“Acceleration of Compute-Intensive Applications using Compute Unified Device Architecture (CUDA)”

by

Ali Zaidi (az206)

MSc. in Advanced Computing
Individual Project Report

Dr. Oskar Mencer
Dr. Paul Kelly

September 2007
Abstract

CPUs (Central Processing Units) are designed to deliver acceptable levels of performance for a broad spectrum of software applications. This is evident from the diverse set of resources boasted by modern CPUs. These resources can be coarsely categorized into data-caching and data-processing resources. Inevitably there are applications or parts of an application which pose excessive demands on one of these categories, creating a performance bottleneck. In case of applications or parts of an application characterized by a majority of data-processing operations can be delegated to accelerators such as GPUs (Graphic Processing Units) and FPGAs (Field Programmable Gate Arrays) working alongside the CPU. Generally, the amount of memory that needs to be transferred to/from the accelerator determines the feasibility of this delegation.

In this project, we will develop a software tool capable of estimating the potential speed-up gained by delegating parts of application to execute on an accelerator. This estimate is based on the amount of memory which needs to be transferred to/from the accelerator if the designated parts of the application are to execute there. We run this tool on the CFP95 benchmark suite resulting in estimated speed-up figures for the respective benchmarks. Subsequently, we modify these benchmarks and execute the designated parts of the applications on a GPU working alongside the CPU as an accelerator. Finally, we compare the speed-up estimates with the actual speed-ups achieved using the GPU in order to verify the speed-up estimation tool and gain valuable insight into the dynamics of the acceleration process.
Acknowledgements

I would like to acknowledge the contribution of Kubilay Atasu and Carlos Tavarez during this work. As always, Dr. Oskar Mencer was ever present with his advise when needed.
# Table of Contents

Chapter 1 **Introduction**  
1.1 Proposal 1  
1.2 Motivation 2  
1.3 Background 4  
1.3.1 Implementations using 'Hard-Wired' devices 4  
1.3.2 Implementations using programmable devices 4  
1.3.3 Implementations using Reconfigurable devices 5  
1.3.4 Hybrid Implementation 5  
1.3.5 Estimation of Acceleration 12  
1.4 Document Overview 13

Chapter 2 **Speed-up Estimation**  
2.1 Introduction 14  
2.2 Architecture 16  
2.3 Testing 20

Chapter 3 **Experiments with Benchmarks**  
3.1 Introduction 21  
3.2 Implementation 22  
3.2.1 Tomcatv 26  
3.2.2 Swim 27  
3.2.3 Su2cor 28  
3.2.4 hydro2d 29  
3.2.5 mgrid 31  
3.2.6 applu 32  
3.2.7 turb3d 33  
3.2.8 apsi 33  
3.2.9 fppp 34  
3.2.10 wave5 35
Chapter 4 Experiments with user Application

4.1 Algorithm Description
   4.1.1 Face-Region Search
   4.1.2 Eyes-Region Search
   4.1.3 Eyes-State Search

4.2 Software Implementation
4.3 Hardware Implementation
4.4 GPU Implementation

Chapter 5 Results

5.1 SPEC Benchmark Results
   5.1.1 memBoundarAcc tool results
   5.1.2 Speed-up Estimation Results
   5.1.3 Speed-up Results

5.2 Drowsy Driver Detection system results
   5.2.1 memBoundarAcc tool results
   5.2.2 Speed-up Estimation Results
   5.2.3 Speed-up Results

Chapter 6 Conclusion and Future Work

6.1 Conclusion
6.2 Evaluation
6.3 Future Work

Bibliography

Appendix A 3S Instrumentation Framework
   A.1 Introduction
   A.2 Getting Started
      A.2.1 General Instrumentation
      A.2.2 SPEC Instrumentation
   A.3 Tool Development

Appendix B memBoundaryAcc tool code

Appendix C f2c (Fortran to C)
   C.1 Introduction
   C.2 f2c Conversions
      C.2.1 Name Conversions
      C.2.2 Type Conversions
C.2.3 Return Value Conversions 73
C.2.4 Argument List Conversions 73
C.3 User Options 73

Appendix D **CUDA Architecture** 75
   D.1 Introduction 75
   D.2 CUDA Architecture 76
   D.3 CUDA Programming Model 78

Appendix E **tomcatv CUDA code** 83
   E.1 tomcatv.c 84
   E.2 tomcatv.cu 86
   E.3 tomcatv_kernel.cu 88

Appendix F **ASC (A Stream Compiler)** 90
   F.1 Introduction 90
   F.2 Programming Model 91
   F.3 User Guide 92

Appendix G **Drowsy Driver Detection code** 93

Appendix H **Drowsy Driver Detection ASC code** 98

Appendix I **Drowsy Driver CUDA code** 100
   I.1 algo.c 100
   I.2 algo.cu 103
   I.3 algo_kernel.cu 104
List of Figures

Chapter 1 Introduction

Figure-1.1 The Liquid Circuit Project flyer 3
Figure-1.2 An illustration of the extraction of performance critical program blocks form s/w and their implementation on h/w platform. 7
Figure-1.3 FPU (Floating point Operations) per Second for CPUs and GPU 9
Figure 1.4 Differences in CPU and GPU Architectures. 9
Figure-1.5 Comparison of execution times for CPU and hybrid implementations. 11

Chapter 2 Speed-up Estimation

Figure-2.1 The operation of the memBoundaryAcc tool. 15
Figure 2.2 The architecture of the memBoundaryAcc tool. 16
Figure-2.3 Flow chart for the _3S_memBoundaryAcc function 18

Chapter 3 Experiments with Benchmarks

Figure-3.1: Work-flow chart of the experiments conducted on the SPEC benchmark suite 24
Figure-3.2 Branch ratio of the CFP95 suite. 25
Figure-3.3 Average ticks per block entry for the CFP95 suite. 25

Chapter 4 Experiments with user Application

Figure-4.1 Illustration of various steps of Drowsy Driver Detection algorithm 41
Figure-4.2 Scanning operation in a graphical device 42
Chapter 5 **Results**

Figure-5.1: Memory Transferred to/from program blocks designated for execution on the accelerator.

Figure-5.2: Percentage Distribution of Non-Accelerated Execution times of the SPEC benchmarks.

Figure-5.3: Execution time Estimates using the memBoundaryAcc tool.

Figure-5.4: Estimated Speed-up vs. Actual Speed-up

Figure-5.5: Optimal Estimated Speed-up vs. Optimal Actual Speed-up

Figure-5.6: Memory Transferred to/from program blocks designated for execution on the accelerator.

Figure-5.7: Percentage Distribution of Non-Accelerated Execution times.

Figure-5.8: Execution time Estimates using the memBoundaryAcc tool

Figure-5.9: Estimated Speed-up, Actual Speed-up, Optimal Estimated Speed-up and Optimal Actual Speed-up

Appendix D **CUDA Architecture**

Figure-D.1: The CUDA Architecture software stack

Figure-D.2: Thread addressing scheme of CUDA architecture.

Figure-D.3: CUDA memory hierarchy in the order of proximity to the processors
Chapter 1

Introduction

1.1 Proposal

Acceleration of compute-intensive applications is an active topic of research in HPC (High Performance Computing) community. The philosophy behind this concept is that different parts of an application pose different demands on the execution platform for optimal execution. The emphasis in the majority of modern CPUs (Central Processing Units) is on data-caching and flow-control capabilities. These CPUs are well equipped for handling non-sequential patterns in code as well as data. On the other hand, most applications have some compute-intensive, data-parallel code which can benefit immensely from execution on special devices. These special devices primarily include GPUs (Graphic Processing Unit) and FPGAs (Field Programmable Gate Arrays) employed as co-processors along the CPU. With respect to context these devices are called Accelerators.

Identification and isolation of sections of code which can benefit from execution on accelerators, followed by their implementation and control is an involved process. Successful automation of this process has the potential to unlock many wonderful possibilities and therefore is the subject of significant research endeavours.
In this project, we attempt to take the first step i.e. identification of program blocks which can benefit from execution on an accelerator. In order to achieve our objective we develop a software tool which can analyse program blocks and estimate the maximum achievable speed-up if they are executed on an accelerator. We use this software tool to estimate the speed-ups for numerous program blocks from the SPEC95 floating-point benchmarks.

In addition, we attempt to manually transform these program blocks to run on a GPU. This exercise have three major benefits. The first benefit is proof-of-concept for later research. The second benefit is the invaluable insight into transformation of program blocks for subsequent execution on accelerator, leading to possible automation of this process. The third and final benefit is functional verification of the speed-up estimation tool developed as a part of this project.

1.2 Motivation

This project promises to be the ground-breaking work for the liquid circuits project being conducted at the Computer Architecture Research Group at the Department of Computing. The goal of the Liquid Circuits Project is to investigate Automated Dynamic Hardware Acceleration for Compute-Intensive Applications. The mission statement reads:

“Imagine your PC would change it's circuits to match the programs you are running, like a liquid adapting to it's container.” [1]

Although the ultimate aim for the Liquid Circuits Project is to employ FPGAs as accelerators for compute-intensive applications. However, during the initial phase, the focus of the research is on the identification and isolation of program blocks which can benefit from accelerated execution. Therefore it is perfectly viable to abstract the concept of accelerators and employ GPUs as accelerators for verification of initial research. Moreover, the evolution of GPUs towards becoming acceptable solution for fast execution of compute-intensive, data parallel program sections is an exciting new development which should be incorporated into the wider scope of the Liquid Circuits Project.
The Liquid Circuit Project being conducted at the Computer Architecture Research Group promises to deliver a PC that can change its circuit depending on the software being executed [1].
1.3 Background

The primary methods employed in contemporary computing applications for the implementation of computational algorithms can be classified into three categories.

1.3.1 Implementations using 'Hard-Wired' devices

The first method employs 'Hard-Wired' devices for the implementation of algorithms. These devices include Application Specific Integrated Circuits (ASICs), configured to perform a certain computation, or several off-the-shelf, Large scale integration (LSI) or Medium Scale Integration (MSI) devices, connected on a Printed Circuit Board (PCB) to perform the required operations. Of these two, ASICs delivers better performance. This is largely due to the fact that the signal propagation delays are significantly smaller for on-chip connections as compared to PCB connections. However, the ASIC is prone to incur higher Non-Recurring Engineering (NRE) costs. Both of these solutions have a slight disadvantage when it comes to flexibility. In case of any design changes or upgrades, the cost for replacing the ASIC or redesigning the PCB is a significant factor which weighs against these solutions.

1.3.2 Implementations using programmable devices

The second method for the execution of algorithms is to use 'Programmable' microprocessors. The immediate advantage is that of flexibility. Microprocessors execute a set of instructions to perform the requisite computations. Any change in functionality of this computation can be easily incorporated by changing the set of instructions by reprogramming the device. However, there is a certain price to pay for this flexibility in terms of device performance and throughput. This is due to the fact that the very nature of the microprocessor allows for a sequential execution of tasks. The microprocessor must read each instruction from the memory, decode it, and only then executes it. The execution phase contributes towards the overall computational goal, while the fetch and decode cycles for every instruction can be termed as execution overheads. Additionally, the instruction set of a microprocessor is determined during the architectural design phase. Therefore, any attempt towards implementing an algorithm is limited by the instruction set for the particular microprocessor, although the control for the flow of instructions to be executed rests with the programmer.
1.3.3 Implementations using Reconfigurable devices

The diverse advantages and disadvantages of the implementation of computational algorithms using hard-wired and programmable solutions have resulted in a technology gap between the two domains. The last decade have witnessed the emergence of 'Reconfigurable' devices, which promise to bridge the gap between the traditional solutions by matching the performance of ASICs and being as flexible as microprocessors. Although several devices are included in the family of reconfigurable devices, however, Field Programmable Gate Arrays (FPGAs) are claimed by many to be the flag bearers of the potential of reconfigurable logic devices and generally assert the technological trends which shape the industry.

Reconfigurable devices, comprises of generic arrays of computational elements termed as 'Logic Blocks' whose functionality is determined by a configuration bit stream. The logic blocks are connected by sets of routing resources. These routing resources are implemented by on-chip 'Crossbar Switch Matrices' which, in turn are also configurable. Computational algorithms can be realized by dividing them into logic components which can be mapped directly on to the logic blocks. Subsequently, the logic blocks contributing towards the computational algorithm are connected by the configurable crossbar switch matrices to form the necessary circuit.

1.3.4 Hybrid Implementation

Owing much to the diversity of the devices discussed above, it is impossible to determine precedents in terms of performance. A recent trend is to mix and match the various platforms in accordance with the requirements of the program. This essentially means that different sections of the program are executed on different platforms. The choice of platform depends upon the characteristics of the program section under consideration. For example, if the program is highly data-parallel, it might benefit from execution on a GPU with the capability to execute hundreds of threads simultaneously. On the other hand, if the program section is compute-intensive, and therefore straining the ALUs (Arithmetic Logic Units) in the processor, it might be beneficial to execute this section on an FPGA. Barring extremely rare cases, the bulk of the program is run on a CPU owing to the excellent data-caching and program-flow capabilities of modern processors. However, certain special sections of the program as discussed earlier might reap enormous benefits by changing the architecture of the execution platform. Enter the world of accelerators.
Majority of the research in accelerator technology is employing two execution platforms to accelerate the application: namely FPGAs and GPUs. In following paragraphs, we provide a brief description of each platform, its architecture, strengths and weaknesses.

**FPGA**

FPGAs and reconfigurable computing have been shown to accelerate a variety of applications. The paper at reference [2] lists the speed-ups achieved for implementing various algorithms in FPGA. By careful analysis of these works, we can categorize the circumstances in which the traditional microprocessors or the contemporary reconfigurable devices are preferable. For example, when the function and data granularity to be computed are well-understood and fixed, and when the function can be economically implemented in the limited silicon, dedicated hardware provides the most computational capacity per unit area to the application. On the other hand, if we are limited spatially and the function to be computed has a high operation and data diversity, we are forced into reusing limited active space and accepting limited instruction and data bandwidth. In this case, conventional microprocessors are most effective since they dedicate considerable space to on-chip instruction storage in order to minimize off-chip instruction traffic while executing descriptively complex tasks.

In other words, it can be stated that the implementation of computational algorithms in configurable devices is accomplished in 'Space Domain'. On the other hand, the implementation of computational algorithms in microprocessors are executed in 'Time Domain'. Unlike the microprocessors which broadcast instructions to the functional units on every clock cycle, instructions in reconfigurable devices are locally configured, allowing the reconfigurable device to compress instruction stream distribution and thus deliver more instructions into active silicon on each cycle. Reconfigurable devices provide a large number of separately programmable, relatively small computational units allowing for the execution of a greater range of computations per unit time. Resources such as memories, crossbar switch matrices and logic blocks are distributed rather than being centralized in large pools. Independent, local access allows for the utilization of on-chip resources in parallel, thus avoiding performance bottlenecks resulting from a large, central resource pool. Therefore, it can be stated that the control portions of the algorithm call for microprocessor implementation while the computationally expensive 'performance critical' portions of the algorithm should benefit from the custom hardware options provided by FPGA as shown in Figure-1.2.
Figure-1.2 An illustration of the extraction of performance critical program blocks form software and their implementation on hardware platform. Notice that the source code can be in any programming language. This is possible because the program blocks are being extracted at the assembly level. Notice that the execution time of the final implementation depends upon the bandwidth of the I/O interface.
The past few years have seen the transformation of the GPU (Graphic Processing Unit) from a chip intended to support the CPU for tasks such as 3D and texture rendering etc. to a general purpose computational powerhouse. This meteoric rise in computational power is evident from Figure-1.3.

This computational power is delivered by multiple cores with access to high memory bandwidth with an emphasis on data processing rather than data caching and flow control. This distinctive architecture was intended for the compute intensive, highly parallel graphic rendering applications. Figure-1.4 illustrates the differences between CPUs and GPUs.

More specifically, the GPU is well-suited to SIMD (Single Instruction Multiple Data) applications in which the same program is executed on multiple data elements in parallel. Usually the program performs operations which have high arithmetic intensity i.e. ratio of arithmetic to memory access and control instructions. Therefore, there is less demand for sophisticated instruction and data caching, flow control and branch prediction mechanisms. The precious silicon resources can therefore be utilized to provide a much higher computational throughput. This throughput is usually delivered by multiple execution cores enabling simultaneous processing of multiple data elements.

Many applications that process large data sets such as arrays can use a data-parallel programming model to speed up the computations. In 3D rendering large sets of pixels and vertices are mapped to parallel threads. Similarly, image and media processing applications such as post-processing of rendered images, video encoding and decoding, image scaling, stereo vision, and pattern recognition can map image blocks and pixels to parallel processing threads. In fact, many algorithms outside the field of image rendering and processing are accelerated by data-parallel processing, from general signal processing or physics simulation to computational finance or computational biology.
Figure-1.3 FPU (Floating point Operations) per Second for CPUs and GPU [3]. The growth in computational power of CPUs is considered meteoric, however it dwarfs in comparison to the growth of GPUs.

Figure-1.4 Differences in CPU and GPU Architectures. CPU devotes significantly more resources to data-caching while GPU devotes more resources to data-processing.
In a non-ideal world however, only a limited number of applications can benefit by using the data-parallel programming model. The major reason behind this limitation is data dependencies in the programs. For example, consider the simple problem of the Fibonacci series (1, 1, 2, 3, 5, 8, 13 ...) by using the following formula:

\[ F(k+2) = F(k+1) + F(k) \]

This is an example of an inherently non-parallel problem. This is so because the calculation of the Fibonacci series by the above formula would entail dependant computations instead of the requisite independent computations. The calculation of the \( k+2 \)th term depends upon the calculation of \( k+1 \)th as well as \( k \)th term. Therefore every computation is dependant on the previous two computations and therefore cannot be run in parallel. The program listing for the Fibonacci series is shown in listing 1.1. A quick glance at the listing will reveal that every iteration of the for loop is dependant upon the previous two iterations, hence the name loop-carried dependency.

```c
int F[SIZE];
F[0] = F[1] = 1;
for(int i=2; i<SIZE; i++)
    F[i] = F[i-1] + F[i-2];
```

Listing 1.1 The Fibonacci series

Another reason for the limited number of applications that can benefit by using the data-parallel programming model is a term called parallel overhead. It is defined as the time required for housekeeping activities necessary for running the multiple threads. Parallel overhead can include factors such as Task start-up time, synchronizations, Data communications, software overhead imposed by the development environment and the operating system and task termination time. Moreover in case of GPU one also has to watch out for the amount of time required to transfer the data from the main memory to the GPU memory as well as the time required to transfer the results back from the GPU memory to the main memory as illustrated by Figure-1.5.
Figure-1.5 Comparison of execution times for CPU and hybrid implementations. Hybrid implementation generally reduces the computation time but results in communication overheads due transfer of data to/from FPGA and GPU
1.3.5 *Estimation of Acceleration*

The implementation of the algorithm on hybrid platform must take into account the time to transfer the data to be processed to the reconfigurable device as well as the time to accumulate the results. These timings can have a significant effect on determining the feasibility of the hybrid implementation as shown in Figure-1.5. This feasibility can be immensely useful because prior transferring to the accelerator, the code section need to be compiled accordingly. This process can take a significant amount of time to accomplish as in the case of FPGAs.

We define an ideal accelerator, as one which executes the allocated section of the code in virtually zero-time. In practical terms, the term zero-time takes a slightly different shade. For instance, in the case of an FPGA accelerator, the data stream being sent to the FPGA, the computations being executed on the FPGA and the results stream coming from the FPGA has considerable potential to be overlapped in time. This potential has significant implications in our case, because it implies that the zero-time phenomenon is not just a theoretical simplification but in the world of accelerators, it can be realized in reality.

The result of the zero-time simplification are significant. For instance, we can now state that if we can determine the following values, we can estimate the speed-up gained by executing a section of the code on an accelerator:

- The amount of memory being transferred to/from the program section being executed in a generic accelerator
- The speed of the bus with which the accelerator and the CPU are connected

The bus speed is relatively easier to find. Knowing the amount of memory being transferred to/from the accelerator during the course of execution requires some work to be done. Enter the memBoundaryAcc software developed as part of this project. The design and development of this tool is described in detail in the next Chapter.
1.4 Document Overview

Chapter 2
Describes the development of the Speed-up estimation software. It presents the architecture of the software as well as describing the algorithm employed for speed-up estimation. It provides brief description of the various functions comprising the software. In addition, it also discusses the testing framework developed in conjunction with the software.

Chapter 3
Describes in detail the experiments conducted on the SPEC95 floating-point benchmarks. During these experiments, we attempted to transform portions of the code from the SPEC95 benchmarks to execute on GPU. The characteristics of the respective benchmarks related to this transformation are also discussed in detail.

Chapter 4
In addition to the benchmarks, we also conduct similar experiments on the Drowsy Driver Detection software developed as part of a research project.

Chapter 5
Presents the results of our work. This includes the execution time distribution of various SPEC benchmarks as well as the Drowsy Driver Detection system. The speed-up estimates and the actual results are also compared.

Chapter 6
Concludes this report by providing an evaluation of the work and pointing in possible directions for future work.

In addition, there are several appendices which provide tutorials for the various tools and techniques employed during this work. In addition there are a few appendices which, provide selected source code listings.
Chapter 2

Speed-up Estimation

2.1 Introduction

The first part of the project was to design and develop the Speed-up Estimation tool chain. This tool chain is to become part of the 3S program instrumentation framework. 3S is characterized by low overhead as compared to similar tools such as Valgrind while having the capability to produce the same type of reports as Valgrind. 3S framework is designed to instrument x86 assembly programs. It adds instrumentation stubs in the x86 assembly. These instrumentation stubs communicate with a library of 3S tools and pass them useful runtime information about program flow and other useful parameters such as the number of ticks the last block took to execute. 3S instrumentation stubs can be placed around the basic program blocks such as loops and calls as well as around every instruction of the target program. The former is called program block level instrumentation while the latter is known as instruction level instrumentation. The speed-up estimation tool chain works at the instruction level. It comprises the two components:

1. 3S tool to determine the memory transferred to/from the program section designated for accelerated execution on a co-processor.
2. Speed-up report generator script.
The 3S tool for determining the memory transferred to/from the program section designated to execute on a co-processor sets out to achieve the following objectives:

1. Determine the amount of memory transferred to/from the program section designated to execute on the co-processor.
2. Determine whether the memory transferred to/from the program section belongs to the stack or the heap.
3. Determine the number of times the program section is entered.

The tool for determining the memory transferred to/from the program section is named as memBoundaryAcc tool. Figure-2.1 illustrates the function of this tool. Listing-2.1 shows a sample report from the tool.

Figure-2.1 The operation of the memBoundaryAcc tool. The tool determines the amount of memory which is required to be transferred to/from the marked program section. Given the bus speed, the Speed-up can be easily calculated.

Listing-2.1 A sample report from the memBoundaryAcc tool

Where:

- Entry is the number of times the program section in consideration is entered.
- StackIn and StackOut is the amount of memory transferred from/to the stack during execution of the FPGA block respectively.
- HeapIn and HeapOut is the amount of memory transferred from/to the stack during execution of the FPGA block respectively.
2.2 Architecture

The code for the memBoundaryAcc tool is presented in Appendix B. Figure-2.2 shows the architecture of the memBoundaryAcc tool. It can be observed that the keywords used to mark the section of the code which is designated for accelerated execution are BEGIN_FPGA_N and END_FPGA_N. Presently, the maximum value of N is three i.e. three code sections can be marked for accelerated execution.

![Diagram of the architecture of the memBoundaryAcc tool. The program block designated for accelerated execution is marked by the BEGIN_FPGA_N and END_FPGA_N markers as shown.](image)
The 3S program instrumentation framework requires its tools to contain a standard function which is called after every program block in case of block level instrumentation and after every instruction in case of instruction level instrumentation. The name of this function should be \_3S\_<Tool Name>\_ In addition, following the recommended practices of modular design, we define several helper functions which perform the necessary operations required by the tool. Description of the various functions is provided in the following paragraphs.

**void enter_fpga_block_3s()**

This function handles an entry into the FPGA block by incrementing the FPGA depth counter and making sure that the FPGA Depth is greater than zero. Moreover, it also increments the FPGA Entry counter. The FPGA Entry counter is present in the memBoundaryAcc report.

**void exit_fpga_block_3s()**

This function handles the exit from an FPGA block by decrementing the FPGA depth variable and making sure that the FPGA depth variable is greater than zero.

**bool insideFPGA()**

This function returns true if we are inside an FPGA block.

**void* getAddress(const std::string& s)**

This function returns the address in standard hexadecimal notation which starts from 0x followed by the respective hexadecimal digits.

**void getBounds()**

This function gets the start and end addresses for stack and heap memory.

**bool closerToStack(void* address)**

This function returns true if the address in the argument lies closer to the stack in the memory.
bool isStack(void* address)

This function returns true of the address in the argument appears to belong to the stack.

void _3S__memBoundaryAcc()

This function implements the functionality of the tool and is placed as part of the 3S instrumentation stub in the assembly file. Figure-2.3 shows the flow chart for this function.

Figure-2.3 Flow chart for the _3S__memBoundaryAcc function
Following information utilized in the algorithm is provided by the 3S framework:

- Type of the symbol for which control is passed to the memBoundaryAcc tool i.e. program block or instruction.
- Whether the instruction is categorized as a memory access instruction.
- What is the address in memory which the instruction tries to access.
- Whether the instruction is reads or writes the memory or does both.

In case of a memory read instruction, we first check whether the memory read is taking place inside the program block marked by the BEGIN_FPGA_N and END_FPGA_N markers or not. If the memory is being read inside the marked program block, we record the address in a map data structure. The key of the map is the address while the entry is the number of times the location is read. Moreover, if the location is being read for the first time, we also increment the appropriate counter i.e. StackIn if the memory belongs to the stack or HeapIn otherwise. Recording only the first access to the memory is important because the accelerator is supposed to have some local memory so that a variable accessed more than once inside the marked program block only needs to be transferred once. If the memory read lies outside the marked program block for the first time, we check if the data being read has been written in the FPGA block. If this is the case, we increment the appropriate counter i.e. StackOut if the address in question lies in the proximity of the stack or HeapOut otherwise.

In case of a memory write instruction, we again check if the memory write is taking place inside the marked program block or not. If this is the case, we record the memory write in another map data structure reserved for memory writes inside the marked program block. The key to the map is the address of the memory being accessed while the value is the number of times the memory is being accessed.

```c
void _3S__memBoundaryAcc_report()
```

This function is responsible for writing the memBoundaryAcc report containing the information collected during the execution of the instrumented program.
2.3 Testing

In order to verify the functionality of our tool, we have developed a set of test programs. Each test programs comprise of three program blocks

- Program block A initializes vectors to be processed in the program block. The amount of memory for these vectors represents the DATA parameter. This parameter can be varied to test various scenarios.

- Program block B is designated for accelerated execution and is marked by the FPGA_BEGIN_N and FPGA_END_N markers. Part of the DATA parameter which is utilized in the program block B represents the MEM_IN parameter. The result vector produced is represented by the parameter called RESULT.

- Program block C utilizes part of the RESULT vector produced in the program block B. This part is called the MEM_OUT parameter.

A total of six test programs were written. Table-2.1 shows the various cases tested by these programs as well as the results produced by the memBoundaryAcc tool.

<table>
<thead>
<tr>
<th>TEST</th>
<th>DATA</th>
<th>MEM_IN</th>
<th>RESULT</th>
<th>MEM_OUT</th>
<th>ENTRY</th>
<th>TOOL OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>20</td>
<td>10</td>
<td>10</td>
<td>1</td>
<td>Correct</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>50</td>
<td>25</td>
<td>25</td>
<td>1</td>
<td>Correct</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>1</td>
<td>Correct</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>20</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>Correct</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>20 (x2)</td>
<td>10</td>
<td>10</td>
<td>1</td>
<td>Correct</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td>20</td>
<td>10</td>
<td>10</td>
<td>5</td>
<td>Correct</td>
</tr>
</tbody>
</table>

Table-2.1 Various permutations of test data used for testing the tool
Chapter 3

Experiments with Benchmarks

3.1 Introduction

For reasons mentioned in the first chapter, we experimented with the SPEC95 benchmark suite [4]. The work-flow of these experiments is shown in Figure-3.1. We employed NVIDIA GPU to accelerate the benchmarks. Incidentally, the speed-up figures estimated by the tool is an upper-limit to the acceleration that can be achieved by employing a suitable accelerator. The GPU may or may not be a suitable accelerator depending upon the program we want to accelerate. Generally, in case of GPU, the more parallel threads of execution we can execute simultaneously, the higher the speed-up. Therefore, program in which the execution core is characterised by loops with loop carried dependencies (especially loop carried dependencies of small dependence distance), generally do not yield high speed-up figures when accelerated with the GPU. Detailed explanation of loop carried dependencies is provided in Appendix D.
The various benchmarks of SPEC95 suite can be categorized as legacy code and therefore, they cannot gain any benefits from contemporary microprocessor features for high performance computing such as Superscalar/Vector processing support such as Intel SSE Extensions and Multiple Cores. Moreover, these software are not optimized for execution on modern microprocessors architecture extensions. They were designed to run with much smaller caches and memory. Such legacy programs represent a distinct advantage of using accelerators. The critical loops inside these programs can be executed on accelerators to gain significant speed-ups.

We have employed the CFP95 subset of the SPEC95 benchmark suite. Our selection can be justified by the fact that the acceleration of compute-intensive program blocks is ideally suited for scientific and multimedia applications which benefit from vectorization. CFP95 represent a cross-section of such applications.

A brief summary of the respective CFP benchmarks along with some control flow parameters such as branch ratio and ticks per block entry is provided in subsequent sections. Branch ratio is defined as the number of taken branches divided by the number of not-taken branches. In case of CFP95 benchmarks, a high value of the branch ratio will indicate that the program has many loops. Ticks per block entry is defined as the average time spent in each program block. It gives an idea of how large the program blocks are.

### 3.2 Implementation

Before implementing the benchmarks to run on the GPU, we need to perform two tasks. Firstly, we convert the benchmark from double precision to single precision, followed by an attempt to translate the program to C programming language using f2c tool. The translation to C promises to facilitate the analysis and execution of suitable portions of the program on GPU. The conversion from double precision to single precision is necessary because the target GPU only supports single precision arithmetic.

After these conversions, we compile the benchmark. The compilation process usually produces errors because the Fortran code contains significant number of calls to double precision version of library routines. In order to compile the program, we need to replace these calls to double precision library routines from their single precision counterparts.
After compiling, we attempt to run the benchmarks. In some cases, the execution fails resulting in a runtime exception such as NaN (Not a Number) or a Floating Point exception. In other cases, the program executes but produces the wrong results. These benchmarks are implemented on the GPU however, we exercise caution in relying on the speed-up figures produced by these benchmarks. This is so because many benchmarks in the CFP95 suite implement algorithms which rely on the double precision arithmetic to converge. Converting the benchmark from double precision to single precision may therefore alter the execution profile of these benchmarks resulting in the benchmark running abnormally faster or slower in some case. Several CFP95 benchmarks produced results which were sufficiently accurate. Naturally, the accuracy of the results is a relative matter. However, in our case we are not particularly concerned about getting extremely accurate results as long as the execution profile of the benchmark does not show abnormal behaviour.

In order to gain some insight into the characteristics of the CFP95 benchmarks, we analysed them using the 3S instrumentation framework. Figure 3.2 and 3.3 shows the results of our analysis.
Figure-3.1: Work-flow chart of the experiments conducted on the SPEC benchmark suite
Figure-3.2 Branch ratio of the CFP95 suite. Branch ratio is the ratio between taken and not-taken branches during program execution.

Figure-3.3 Average ticks per block entry for the CFP95 suite. Average ticks per block entry is a measure of the average size of the program blocks.
3.2.1 **Tomcatv**

Tomcatv is a double precision floating point mesh generation program which does little I/O and is described to be 90 - 98 % vectorizable. The various control flow statistics of tomcatv are shown in Figure-3.2 and Figure-3.3. Following are some of the important observations regarding the statistics:

- tomcatv comprise a single main function doing all the computing. The core comprise of a nested loop computing the residuals among a few other loops.
- It has a relatively modest branch ratio of 2.6 which signifies that this benchmark has few loops.
- The average ticks per block entry figure is the highest among CFP95 benchmarks – 130. This means that it has computationally large program blocks which are good for streaming.

We attempt to convert the tomcatv benchmark from double precision to single precision, followed by an attempt to translate the program to C programming language using f2c tool. Both of these attempts succeed apparently, as shown by the respective program outputs for the original Fortran version of the program as well as the single precision C version.

Three loops in the program are selected for execution on the GPU. These loops are isolated from the rest of the program by encapsulating them into separate functions called fpga1_d, fpga2_d and fpga3_d respectively. The variables required by these functions to execute are determined by inspection and included in the function parameter list. These functions are placed in a separate file named tomcatv_kernel.cu, to be compiled by the CUDA (Compute Unified Device Architecture) compiler called nvcc. CUDA is an architecture developed by NVIDIA and simplifies the use of GPU as a generic co-processor for executing suitable program sections for the purpose of acceleration.
In addition, three more functions are written, namely fpga1_, fpga2_ and fpga3_. These functions are responsible for communicating the data to the CUDA functions, providing the CUDA functions with the execution profile, collecting the results from the CUDA functions and cleaning up the memory once they have finished execution. These functions are placed in a file called tomcatv.cu. The rest of the tomcatv code is in the file called tomcatv.c. The CUDA implementation of the tomcatv the code is presented in Appendix E and includes some very useful comments describing the various CUDA related features.

In case of fpga1_d, we have a two-dimensional loop without any loop carried dependency. Therefore, we are able to launch axb simultaneous execution threads, where a and b are the number of loop iteration for respective dimensions. However, in case of fpga2_d and fpga3_d, we are limited by loop carried dependencies in one of the loop dimensions. Therefore, we are able to launch parallel threads of execution in only one loop dimension. Table-4.1 shows the various characteristics of the CUDA implementation of tomcatv.

### 3.2.2 Swim

Swim is a scientific benchmark with single precision floating point arithmetic. SWIM stands for Shallow Water Model with 1024 x 1024 grid. The program solves the system of shallow water equations using finite difference approximations on a N1 x N2 grid. The various control flow statistics of swim are shown in Figure-3.2 and Figure-3.3. Following are some of the important observations regarding the statistics:

- The swim core comprise three functions named calc1,calc2 and calc3 which account for 33, 25 and 25 percent of the total execution time respectively.
- The branch ratio of swim is 511 which is among the highest in CFP95 suite and shows that there are quite a few loops with large number of iterations.
- The ticks per block figure is fairly large – 73. This shows that the respective program blocks have a fair amount of computation.
The attempts to reduce the precision of the program from double to single as well as the attempts to translate the original Fortran program into C language are successful as shown by the outputs of the respective programs.

As in case of tomcatv, three loops in the program are selected for execution on the GPU. Just like tomcatv, these loops are isolated to form functions named as fpga1d, fpga2d and fpga3d and placed in a separate file called swim_kernel.cu. The respective supporting functions called fpga1_, fpga2_ and fpga3_ are also written and placed in a file called swim.cu. Both of these files are to be compiled with nvcc. The rest of the swim code is in the file called swim.c and is to be compiled using the gcc.

In case of the swim benchmark, all three CUDA functions contain two-dimensional loops and neither loop has any dependency in any dimension or direction. Therefore, we are able to launch parallel threads of execution in both dimensions of all the loops. Table-4.1 shows the various characteristics of the CUDA implementation of swim.

### 3.2.3 Su2cor

The program is a vectorizable, double precision floating-point arithmetic quantum physics application. Masses of elementary particles are computed in the framework of the Quark Gluon theory. The data are computed with a Monte Carlo method taken over from statistical mechanics. The various control flow statistics of su2cor are shown in Figure-3.2 and Figure-3.3. Following are some of the important observations regarding the statistics:

- The su2cor core comprises two functions named matmat and int2v. These functions account for 30 and 28 percent of the total program execution time respectively.
- The branch ratio for su2cor is approximately 16 which signifies a fair amount of loops.
- Average number of ticks per program block is 82.

The attempts to reduce the precision of the program from double to single as well as the attempts to translate the original Fortran program into C language are successful as shown by the outputs of the respective programs.
Three loops in the program are selected for execution on the GPU. Two of these loops are isolated and placed in separate functions called fpga1d and fpga2d. The third loop is already isolated in a function called bespol. This function is converted to bespold. All these functions are placed in a separate file called su2cor_kernel.cu. In addition the respective supporting functions called fpга1, fpга2 and bespol are also written and placed in a file called su2cor.cu. The extension of these files ensures that they are compiled by the nvcc rather than gcc. The rest of the su2cor code is in the file called su2cor.c and is to be compiled using the gcc.

The fpga1d function is characterized by a for loop which encompasses most of the code. Fortunately this loop does not have a loop carried dependency and therefore, we launch n parallel threads of execution for this loop, where n is the number of loop iterations. The fpga2d function also has a loop encompassing most of the code but this loop has a loop carried dependency with a dependence distance of 1 and therefore, does not have any potential for threads. The bespol function like fpga1d has a loop without any dependency and is converted to run multiple threads. Table-4.1 shows the various characteristics of the CUDA implementation of su2cor.

3.2.4 hydro2d

The program is a vectorizable with double precision floating-point arithmetic. It is an astrophysics application which solved hydrodynamical Navier Stokes equations to compute galactical jets. The various control flow statistics of hydro2d are shown in Figure-3.2 and Figure-3.3. Following are some of the important observations regarding the statistics:

- The hydro2d core comprise of a single function named filter. This function accounts for about 42 percent of the total execution time. This means that su2cor has a good core element.
- The branch ratio for hydro2d is 9.
- Average number of ticks per program block entered is approximately 56, which is modest.

The attempts to reduce the precision of the program from double to single as well as the attempts to translate the original Fortran program into C language are successful as shown by the outputs of the respective programs.
Three loops in the program are selected for execution on the GPU. These loops are isolated and placed in separate functions called fpga1d, fpga2d and fpga3d. All these functions are placed in a separate file called hydro2d_kernel.cu. In addition the respective supporting functions called fpga1, fpga2 and fpga3 are also written and placed in a separate file called hydro2d.cu. The extension of these files ensures that they are compiled by the nvcc rather than gcc. The rest of the hydro2d code is in the file called hydro2d.c and is to be compiled using the gcc.

The fpga1d and fpga2d functions are characterized by a two-dimensional for loop which encompasses most of the code. This loop does not have a loop carried dependency in either dimension and therefore, we launch axb parallel threads of execution for this loop, where a and b are the number of loop iterations in the respective dimensions. The fpga3d function is special because it comprises a series of two sequential two-dimensional for loops. Let the respective number of iterations of these loops be represented by axb and cxd respectively. Then, we launch mxn parallel threads of execution such that m and n are given by Listing 3.1.

\[
m = \max(a, c) \quad n = \max(b, d)
\]

Moreover, the loops are encapsulated within if statements of the form:

if(blockIdx.x <= a)      if(blockIdx.x <= c)
if(blockIdx.y <= b)      if(blockIdx.x <= d)

**Listing-3.1** Formulae for number of parallel threads of execution for the hydro2d benchmark

This is to guard against launching too many threads for the respective loops and as a consequence accessing an out of bounds address for the array. Table-4.1 shows the various characteristics of the CUDA implementation of hydro2d.
3.2.5 mgrid

mgrid is a multi-grid solver for 3D potential grid. mgrid demonstrates the capabilities of a very simple multi-grid solver in computing a three dimensional potential field. Adapted by SPEC from the NAS Parallel Benchmarks with modifications for portability and a different workload. The various control flow statistics of mgrid are shown in Figure-3.2 and Figure-3.3. Following are some of the important observations regarding the statistics:

- The mgrid core comprise two functions named resid and psinv. These functions account for 56 and 24 percent of the total program execution time respectively.
- Branch ratio for mgrid is 35 which signifies a large number of frequent loop iterations.
- Average number of ticks per program block entered is 72.

The attempts to reduce the precision of the program from double to single as well as the attempts to translate the original Fortran program into C language are successful as shown by the outputs of the respective programs.

Three loops in the program are selected for execution on the GPU. These loops are isolated and placed in separate functions called fpga1d, fpga2d and fpga3d. All these functions are placed in a separate file called mgrid_kernel.cu. In addition the respective supporting functions called fpga1, fpga2 and fpga3 are also written and placed in a separate file called mgrid.cu. The extension of these files ensures that they are compiled by the nvcc rather than gcc. The rest of the mgrid code is in the file called mgrid.c and is to be compiled using the gcc.

The fpga1d and fpga2d functions are characterized by a two-dimensional for loop which encompasses most of the code. This loop does not have a loop carried dependency in either dimension and therefore, we launch axb parallel threads of execution for this loop, where a and b are the number of loop iterations in the respective dimensions. The fpga3d function is special because it comprises a series of two sequential two-dimensional for loops. However, unlike hydro2d, these loops are characterized by the same number of iterations, eliminating any need to perform checks for the number of threads launched. Table-4.1 shows the various characteristics of the CUDA implementation of mgrid.
3.2.6 applu

applu is a Computational Fluid Dynamics and Computational Physics benchmark. Solution of five coupled non-linear PDE’s, on a 3-dimensional logically structured grid, using an implicit pseudo-time marching scheme, based on two-factor approximate factorization of the sparse Jacobian matrix. The various control flow statistics of applu are shown in Figure-3.2 and Figure-3.3. Following are some of the important observations regarding the statistics:

- The applu core comprises three functions named buts, blts and jacld. These functions account for 31, 25 and 17 percent of the total execution time respectively.
- The branch ratio is fairly small – 3
- Average ticks per block entry for this benchmark is 59.

The attempts to reduce the precision of the program from double to single as well as the attempts to translate the original Fortran program into C language were successful.

In case of mgrid benchmark, there is no need to isolate the loops identified for execution on the GPU. This is because all these loops are already isolated in separate functions in the original mgrid program. The name of these functions is blts, buts and jacld. In converting these functions to execute on the GPU, their names are changed to blts_d, buts_d and jacld_d respectively and placed in a separate file called mgrid_kernel.cu. The original names of the functions i.e. blts, buts and jacld are taken by supporting functions which in essence are the execution stubs of the respective CUDA functions. These stubs are placed in a separate file called mgrid.cu.

The blts_d and buts_d functions are characterized by a series of several loops with as many as 5 dimensions. However, we are limited by the fact that there are loop carried dependency in the first two dimensions for blts_d as well as buts_d. For jacld_d, we are provided a three-dimensional loop. The outer dimensions of this loop are free from any loop carried dependency and appropriately, we launch axb threads for this loop where a and b are the number of iterations of the outer most loops for the respective dimensions. Table-4.1 shows the various characteristics of the CUDA implementation of applu.
3.2.7 turb3d

Turb3d simulates isotropic, homogeneous turbulence in a cube with periodic boundary conditions in x,y,z coordinate directions. It has a large 1D FFT computational component. It solves the Navier-Stokes equations using a pseudo spectral method. Leapfrog-Crank-Nicolson scheme is used for time stepping. The various control flow statistics of turb3d are shown in Figure-3.2 and Figure-3.3. Following are some of the important observations regarding the statistics:

- The turb3d core comprises three methods named fftz2, dcft and fftz1. These functions account for 30, 22 and 17 percent of the total program execution time respectively.
- The branch ratio for turb3d is 5.71.
- Average ticks per block entry for this benchmark is 52.

The attempts to reduce the precision of the program from double to single precision did not succeed. We get a floating point exception in the output and after several attempt, we are unable to run this benchmark with reduced precision.

3.2.8 apsi

apsi is a scientific benchmark with double precision floating point arithmetic. It solves for the mesoscale and synoptic variations of potential temperature, U AND V wind components, and the mesoscale vertical velocity W pressure and distribution of pollutants C having sources Q. The synoptic scale components are in quasi-steady state balance, while the mesoscale pressure and velocity W are found diagnostically. The various control flow statistics of apsi are shown in Figure-3.2 and Figure-3.3. Following are some of the important observations regarding the statistics:
• The apsi benchmark has the most evenly distributed computational core in the CFP95 suite comprising of three functions named trid, radb4 and rad4. These functions account for approximately 14, 10 and 7 percent of the total execution time of the program respectively.

• The branch ratio is 9.

• Average number of instructions per block entry is 59.

The attempts to reduce the precision of the program from double to single precision do not succeed. We get a NaN (Not a Number) exception in the output and after several attempt, we are unable to run this benchmark with reduced precision.

3.2.9 fppp

A scientific benchmark with double precision floating point arithmetic. Fpppp is a quantum chemistry benchmark which measures performance on one style of computation (two electron integral derivative) which occurs in the Gaussian series of programs. It does very little I/O. The input to the program is found in a file and contains as the first entry, the number of atoms. The various control flow statistics of fpppp are shown in Figure-3.2 and Figure-3.3. Following are some of the important observations regarding the statistics:

• The fpppp core is highly concentrated. It comprises of two functions named twldrv and fpppp. These functions account for 59 and 31 percent of the total program execution time respectively.

• Branch ratio for fpppp is 0.94 which is the lowest among the CFP95 benchmarks and shows and shows that the fpppp benchmark contains very few loops.

• Average number of instructions per block entry for the fpppp benchmark is 94.

The attempts to reduce the precision of the program from double to single precision do not succeed. We get a NaN (Not a Number) exception in the output and after several attempt, we are unable to run this benchmark with reduced precision.
3.2.10 wave5

Scientific benchmark with double precision floating point arithmetic. A two-dimensional, relativistic, electromagnetic particle-in-cell simulation code used to study various plasma phenomena. WAVE solves Maxwell's equations and particle equations of motion on a Cartesian mesh with a variety of field and particle boundary conditions. The benchmark problem involves 750,000 particles on 75,000 grid points for 40 time steps; about 11 M words (32-bit) of memory are required. Considerable indirect addressing dominates the code's runtime. The various control flow statistics of wave5 are shown in Figure-3.2 and Figure-3.3. Following are some of the important observations regarding the statistics:

- The wave5 core comprises of a single function called parmvr. This function accounts for 54 percent of the total program execution time signifying a highly concentrated core.
- Branch ratio for the wave5 benchmark is 30 which signifies a large number of loops and iterations.
- Average number of instructions per program block entered is 56.

The attempts to reduce the precision of the program from double to single as well as the attempts to translate the original Fortran program into C language were not successful. We get a Floating Point exception when the program is executed.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Program Block</th>
<th>Number of Threads</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>tomcatv</td>
<td></td>
<td>n-1 x n-1</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>FPGA_1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FPGA_2</td>
<td>n-1 x 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FPGA_3</td>
<td>n-1 x 1</td>
<td></td>
</tr>
<tr>
<td>swim</td>
<td></td>
<td>n-1 x 1</td>
<td>r</td>
</tr>
<tr>
<td></td>
<td>FPGA_1</td>
<td>cons_1-&gt;m x cons_1-&gt;n</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>FPGA_2</td>
<td>cons_1-&gt;m x cons_1-&gt;n</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>FPGA_3</td>
<td>cons_1-&gt;m x cons_1-&gt;n</td>
<td>None</td>
</tr>
<tr>
<td>s2c2or</td>
<td></td>
<td>n x 1</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>FPGA_1</td>
<td>lvec x 1</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>FPGA_2</td>
<td>n x 1</td>
<td>pchd</td>
</tr>
<tr>
<td></td>
<td>FPGA_3</td>
<td>lvec x 1</td>
<td>None</td>
</tr>
<tr>
<td>hydro2d</td>
<td></td>
<td>n x 1</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>FPGA_1</td>
<td>advc_1-&gt;m x advc_1-&gt;n</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>FPGA_2</td>
<td>advc_1-&gt;m x advc_1-&gt;n</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>FPGA_3</td>
<td>max(advc_1-&gt;m, advc_1-&gt;m)</td>
<td>None</td>
</tr>
<tr>
<td>mgrid</td>
<td></td>
<td>n x n</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>FPGA_1</td>
<td>n-2 x n-2</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>FPGA_2</td>
<td>n-2 x n-2</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>FPGA_3</td>
<td>n-2 x n-2</td>
<td>None</td>
</tr>
<tr>
<td>aplicu</td>
<td></td>
<td>n x n</td>
<td>v</td>
</tr>
<tr>
<td></td>
<td>FPGA_1</td>
<td>n x 1</td>
<td>v</td>
</tr>
<tr>
<td></td>
<td>FPGA_2</td>
<td>n x 1</td>
<td>v</td>
</tr>
<tr>
<td></td>
<td>FPGA_3</td>
<td>cgcon_1-&gt;m x cgcon_1-&gt;n</td>
<td>None</td>
</tr>
<tr>
<td>wave5</td>
<td></td>
<td>n x n</td>
<td>curr_1-&gt;cx, curr_1-&gt;cy, curr_1-&gt;cz</td>
</tr>
<tr>
<td></td>
<td>FPGA_1</td>
<td>n x 1</td>
<td>curr_1-&gt;cx, curr_1-&gt;cy, curr_1-&gt;cz</td>
</tr>
<tr>
<td></td>
<td>FPGA_2</td>
<td>(parcm_1-&gt;l)+1 x (parcm_1-&gt;l)+1</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>FPGA_3</td>
<td>(parcm_1-&gt;l)+1 x (parcm_1-&gt;l)+1</td>
<td>None</td>
</tr>
</tbody>
</table>

**Table-4.1:** Various characteristics of the CUDA implementation of SPEC benchmarks. The number of threads and dependencies are represented in terms of respective program variables which can be looked-up in the program source code.
Chapter 4

Experiments with user Application

In order to observe the forces at work when a developer is trying to accelerate his own applications, we experimented with Drowsy Driver Detector software developed by us as part of an earlier research effort. It is a perfect embodiment of a modern 'real-world', yet unsolved problem in the domain of computer vision which presents the usual challenges associated with high quality, real-time processing of high speed data stream, which in this case is the video feed from the on-board cameras.

4.1 Algorithm Description

Our algorithm essentially comprise of three steps: Face-Region Search, Eyes-Region search and Eyes-State detection. These steps are described in the following paragraphs:

4.1.1 Face-Region Search

In the first step, we try to determine the location of the driver's face in the image. The left and right boundaries of the driver's face are determined by thresholding the vertical projection of the image. For a 2D grey scale image of resolution \( n \times m \), vertical projection can be described by the following equation:
\[ PV(x) = \sum_{y=1}^{n} I(x,y) \]  \hspace{1cm} (1)

Where \( I(x,y) \) is the grey scale value of the pixel, while \( x \) and \( y \) are the horizontal and vertical coordinates respectively.

The vertical projection contains a crest if the driver's face is lighter than the background. Otherwise it will contain a trough. This crest or trough can be detected by thresholding the vertical projection as described by the following equation:

\[ PV'(x) = \begin{cases} 255, & \forall PV(x) > PVA \\ 0, & \text{otherwise} \end{cases} \]  \hspace{1cm} (2)

Where \( PVA \) is the average of Vertical Projection given by the following equation:

\[ PVA = \frac{1}{n} \sum_{x=1}^{n} PV(x) \]  \hspace{1cm} (3)

The left boundary of the driver's face is the first white pixel in \( SV'(x) \) represented by \( FL \). The right boundary will be the last white pixel represented by \( FR \) as shown in Figure-4.1.

The top and bottom boundaries of the driver's face are determined by thresholding the horizontal projection of the image. For a 2D grey scale image of resolution \( n \times m \), horizontal projection can be described mathematically by the equation:

\[ PH(y) = \sum_{x=0}^{n} I(x,y) \]  \hspace{1cm} (4)

Where \( I(x,y) \) is the grey scale value of the pixel, while \( x \) and \( y \) are the horizontal and vertical coordinates respectively

The horizontal projection contains a crest if the driver's face is lighter than the background. Otherwise it will contain a trough. This crest or trough can be detected by thresholding the horizontal projection as described by the following equation:
Where $PHA$ is the average of Horizontal Projection given by the following equation:

$$PHA = \frac{1}{m} \sum_{y=1}^{m} PH(y)$$  \hspace{1cm} (6)$$

The upper boundary of the driver's face will be the first white pixel in $PH'(y)$ represented by $FT$. The lower boundary will be the last white pixel represented by $FB$ as shown in Figure-4.1.

**4.1.2 Eyes-Region Search**

The eyes region in the driver's face is determined by thresholding the horizontal projection of the part of image within the facial boundaries as described by the following equation:

$$PH_E(y) = \sum_{x=FL}^{FR} I(x,y) \quad \forall \text{ } FB<y<FT$$ \hspace{1cm} (7)$$

Where $I(x,y)$ is the grey scale value of the pixel, while $x$ and $y$ are the horizontal and vertical coordinates respectively. The threshold operation is described by the following equation:

$$PH'_E(y) = 255, \forall PH_E(x)>PHA_E$$ \hspace{1cm} (8)$$

Where $PHA_E$ is the average of Horizontal Projection within the facial boundaries given by the following equation:

$$PHA_E = \frac{1}{|FB-FT|} \sum_{y=FR}^{FB} PH_E |y|$$ \hspace{1cm} (9)$$
The top and bottom boundaries of the eyes region can be determined by a simple heuristic rule: The eyes region will be represented by the third local minima in the horizontal projection of the driver's face, preceded by the hair and eyebrows. Let $EU$ and $EL$ represent the upper and lower boundaries of the eyes region respectively as shown in Figure-4.1.

4.1.3 Eyes-State Search

Having detected the location of the driver's eyes in the image, we can detect whether the eyes are open or close. This is determined by applying the edge detection operator followed by complexity function.

The Prewitt operator is used for performing edge detection in the eyes region. The Prewitt operator consists of two components: the vertical edge component is calculated with the kernel $K_v$ and the horizontal edge component calculated with the kernel $K_h$. $|K_v|+|K_h|$ gives an indication of the intensity of the gradient in the current pixel.

When the driver's eyes are closed, the eye window only contains the upper and lower eyelids. However, when the eyes are open, the features detected are considerably more complex, consisting of the upper and lower eye lids, the pupils and the iris. These differences in the visual characteristics of the two images can be highlighted by subjecting the eye window to a complexity function defined as:

$$compl = \sum_{i=FL}^{FR} \sum_{j=EL}^{EU} [b(i,j)-b(i,j-1)]\times k(i)$$

(10)

Where $b(i,j)$ is the binary image resulted by the edge detection, while $k(i)$ is a weight constant defined by the following equation:

$$k(i) = \begin{cases} i, & \forall i(x) < m/2 \\ (m-i), & \text{otherwise} \end{cases}$$

(11)

Edge detection and complexity function are applied to determine whether the driver's eyes are open or close. If the driver's eyes are closed for $N$ consecutive frames, the alarm is switched on. The value of $N$ depends upon several parameters such as the automobile type, road conditions and the driver. This problem is solved by training the system at the start. During training, the system records the values of the complexity function with the eyes of the driver open and close.
Figure 4.1 Illustration of various steps of Drowsy Driver Detection algorithm for eyes open and close including boundaries of Face region and eyes region, Projection values and edge detection

4.2 Software Implementation

Software implementation of drowsy driver detection system is used for functional verification of the algorithm as well as a reference for highlighting the speed-up achieved by the hardware implementation. Software implementation employs the low level image processing functions from OpenCV library. It is compiled using the g++ compiler with the highest optimization level. It is executed on an Intel Pentium 4 as well as an AMD Opteron platform.
4.3 Hardware Implementation

Hardware implementation of drowsy driver detection algorithm is realized using the A Stream Compiler (ASC). ASC provides an interface for generating stream architectures to be implemented on FPGA. The ASC interface uses the familiar C++ syntax with ASC specific semantics to simplify this task [5]. The FPGA based computing platform was a prototype board with PCI and DVI interfacing options and two Xilinx Virtex II 2000 FPGAs.

The FPGA implementation of our drowsy driver detection system employs stream architecture to take advantage of the fine-grain parallelism offered by FPGAs. However, the video stream from the camera follows the standard spatial pattern of CRT devices as shown in Figure-4.2. This spatial pattern restricts the calculation of vertical projection of the image until the last row of pixels is received from the video camera. This situation can be resolved in two ways:

- Buffering the image in On-chip or Off-chip RAM
- Performing the remaining steps of the algorithm on subsequent frames

![Figure-4.2 Scanning operation in a graphical device](image)

Both of these methods result in decreasing the performance of the system. Owing to the hard realtime requirements of the drowsy driver detection system, we cannot compromise the performance. Our solution involves dropping the face region search altogether. Instead, we start by searching for the driver's eyes in the image by thresholding the horizontal projection given by the equation:
\[ PH(y) = \sum_{x=0}^{n} I(x,y) \quad (12) \]

Where \( PH(y) \) is the Horizontal Projection of the image, \( I(x,y) \) is the grey scale value of the pixel, while \( x \) and \( y \) are the horizontal and vertical coordinates respectively.

In this case, we do not have the facial boundaries to limit the search space. Therefore the results for this step are comparatively less accurate. We introduce contrast enhancement to restore the accuracy of the results given by:

\[ CH(y) = PH[y] \times Gamma + Beta \quad (13) \]

Where \( CH(y) \) is the Horizontal Projection of the image with enhanced contrast, while \( Gamma \) and \( Beta \) are constants and their values can be found experimentally.

After contrast enhancement, the new values of horizontal projection are subjected to a thresholding operation, which can be defined by the following equations:

\[ CH(y) = \begin{cases} 255, & \forall CH(x) > 255 \\ 0, & \text{otherwise} \end{cases} \quad (14) \]

This thresholding operation marks the relatively dark facial features such as hair, eyes, mouth etc. against the lighter tone of skin. The thresholding is followed by the comparison of the values and search for local minima representing the eye window.

Hardware implementation of the driver drowsiness detection algorithm takes advantages of the fine-grain parallelism in FPGA architecture. This is accomplished by computing the edge detection and complexity function for every pixel in the image in parallel with the eyes region search.
4.4 GPU Implementation

The software implementation of the drowsy driver detection system is analysed for computationally intensive, data-parallel program blocks. Firstly, the execution profile of the program is obtained using gprof in order to identify the functions which contribute the most towards the execution time of the program. We discovered that the execution core of the Drowsy Driver Detection software comprises three functions, getPV, getPH and getCx accounts for 30, 24 and 16 percent of the total program execution time respectively.

As evident from the execution profile, the functions responsible for hogging most of the execution time are getPV, getPH and getCx. These functions account for almost 100 percent execution time for the program. In order to accelerate the program, we will isolate the three functions which account for most of the program execution time and try to get them running on the GPU. However, simply running these functions on the GPU will not get us any acceleration. We have to find threads of parallel execution in these functions and taking advantage of the architecture of the GPU, try to run these threads simultaneously in order to reduce the execution time of the program.

In addition to modifications to the functions identified for accelerated execution on the GPU, we also need to write three more functions responsible for communicating data to the respective functions executing on the GPU, control the execution profile of these GPU functions and accumulate the results from the respective GPU function. The source code for the CUDA implementation of drowsy driver detection system is presented in Appendix J.

The getPV function has a two-dimensional loop in order to access all the pixels in the image which seems to be present in almost every image processing function. The loop has a loop carried dependence in the image width dimension. However, there is no loop carried dependence in the height dimension. Therefore we can have h threads of execution running simultaneously in this function.

The getPH function has a similar two-dimensional loop. However, unlike the getPV function, the getPH function has a loop carried dependency in the height dimension. Therefore we can run w threads of execution running simultaneously.

The getCX function again has the serial two-dimensional loop. In this case however, there is no loop carried dependency in either dimension and we are able to run wxh threads of execution simultaneously.
Chapter 5

Results

5.1 SPEC Benchmark Results

5.1.1 memBoundaryAcc tool results

The results of the memBoundaryAcc tool are shown in Figure-5.1. The memory access report produced by the memBoundaryAcc tool is used to estimate the potential speed-up if the designated program blocks are executed on an accelerator. This speed-up figure is estimated by calculating the time required for the memory transfers to/from the Accelerator.

The program blocks designated for accelerated execution are termed A, B and C as shown in figure. Speed-up estimation is performed for each of these program blocks. Subsequently, the results are combined to give the total speed-up figure.
5.1.2 Speed-up Estimation Results

Figure-5.2 shows the percentage distribution of non-accelerated execution times of the program blocks A, B and C which are designated for execution on the accelerator. The speed-up estimates for the SPEC benchmarks are obtained using the memory access report generated by the memBoundaryAcc tool. Execution time estimates generated using the memBoundaryAcc reports are shown in Figure-5.3. The rest of the program is represented by the term Processor as it runs on the CPU. If program blocks A, B and C comprise a large percentage of the execution time, we are more likely to get good speed-up as in the case of swim and mgrid.
Figure-5.2: Percentage Distribution of Non-Accelerated Execution times of the SPEC benchmarks.
Figure-5.3: Execution time Estimates using the memBoundaryAcc tool. A decrease in execution time signifies speed-up while an increase in execution time signifies slow-down. In this manner, we can determine the feasibility of executing a program block on the accelerator beforehand.
5.1.3 Speed-up Results

These are the actual speed-ups gained by implementing the program blocks A, B and C on the GPU. The memBoundaryAcc tool provides an upper-bound to the speed-up which can be achieved for a given program section. Owing to several practical limitations such as loop carried dependencies limiting the number of threads, delays due to thread synchronization etc. the actual speed-up figures can approach but never exceed the estimated speed-up as evident by Figure-5.4. In this case, the value of actual speed-up closest to the estimated speed-up was achieved for the tomcatv benchmark for which the Actual/Estimated speed-up ratio is 0.98. The lowest value for this parameter observed for the swim benchmark equals 0.2.

Figure-5.4: Estimated Speed-up vs. Actual Speed-up
Figure-5.5 shows the optimal speed-up figures for SPEC benchmarks. The optimal speed-up figure is calculated by executing the program section which resulted in a slow down on the CPU resulting in a speed-up figure of 1 while the other program sections contribute towards faster execution.

**Figure-5.5:** Optimal Estimated Speed-up vs. Optimal Actual Speed-up
5.2 Drowsy Driver Detection system results

5.2.1 memBoundarAcc tool results

The results of the memBoundaryAcc tool are shown in Figure-5.6. The program blocks designated for accelerated execution are termed getPV, getPH and getCx respectively.

![Figure-5.6](image-url)

**Figure-5.6:** Memory Transferred to/from program blocks designated for execution on the accelerator.
5.2.2 Speed-up Estimation Results

Figure-5.7 shows the percentage distribution of non-accelerated execution times of the program blocks getPV, getPH and getCx which are designated for execution on the accelerator. The speed-up estimates for the Drowsy Driver Detection software are obtained using the memory access report generated by the memBoundaryAcc tool. Execution time estimates generated using the memBoundaryAcc reports are shown in Figure-5.8.

Figure-5.7: Percentage Distribution of Non-Accelerated Execution times.
Figure-5.8: Execution time Estimates using the memBoundaryAcc tool
5.2.3 Speed-up Results

These are the actual speed-ups gained by implementing the program blocks A, B and C on the GPU. Owing to several practical limitations such as loop carried dependencies limiting the number of threads, delays due to thread synchronization etc. the actual speed-up figures can approach but never exceed the estimated speed-up as evident by Figure-5.9.

The figure also shows the optimal speed-up results. The optimal speed-up figure is calculated by executing the program section which resulted in a slow down on the CPU resulting in a speed-up figure of 1 while the other program sections contribute towards faster execution.

![Figure-5.9: Estimated Speed-up, Actual Speed-up, Optimal Estimated Speed-up and Optimal Actual Speed-up](image)
Chapter 6

Conclusion and Future Work

6.1 Conclusion

We have developed the memBoundaryAcc software which can analyse programs in which we have identified the blocks which are to be executed on an accelerator and estimates the maximum achievable speed-up. This software works with the 3S program instrumentation framework to determine the amount of memory which needs to be transferred to/from the program block designated for accelerated execution.

We use this software tool to estimate the speed-ups for the SPEC95 floating-point benchmarks. In addition, we attempted to implement the program blocks designated for acceleration to run on a GPU.

Finally we compared the estimated speed-up figures with the actual speed-up figures. The memBoundaryAcc tool provides an upper bound on the speed-up which can be achieved for a particular program block using an accelerator. Therefore, the actual speed-up figures approached but never exceeded the predicted speed-up.
6.2 Evaluation

As described in Chapter 2, the speed-up estimation software runs as a part of the 3S program instrumentation framework. During the experiments, especially with the larger benchmarks we observed that the instrumentation process resulted in extremely slow execution. In case of the Liquid Circuit Project the aim is to analyse binary files while they are being executed for potential speed-ups. This requirement entails a significant improvement in the performance of the 3S instrumentation framework in general and the memBoundaryAcc tool in particular. We have identified ways to deliver this performance. Some are described in the subsequent section.

We were unable to experiment with all the CFP95 benchmarks. This was due to the fact that all the CFP95 benchmarks except swim are double precision. We had to convert these benchmarks to single precision in order to execute them on the GPU which only support single precision arithmetic. We lost 4 benchmarks to these conversions as discussed in Chapter 4.

6.3 Future Work

The performance of the speed-up estimation tool can be significantly improved if the 3S framework can be extended to instrument selected portions of the program. For instance, we can get significantly higher performance if the 3S framework limits the instrumentation to the program blocks designated to run on the accelerators. This would limit the accuracy of our results because we will not be able to identify the intermediate data flow within the program block resulting in significant number of false positives in the memory access records. A possible work-around to this problem is that whenever we instrument a program block, we write the addresses accessed to a file. When we are instrumenting some other program block, we use the information in this file to eliminate some of the intermediate data flow resulting in false positives.

The speed-up estimation tool developed as part of this project assumes that the accelerated execution of the designated program block takes zero-time. This assumption simplifies the processing to be done in the tool resulting in significantly better performance. However, if we are to limit the instrumentation to the designated program blocks, we open the possibility of more involved operations to be included in the analysis. For example, we can try and simulate the bus connecting the CPU with the accelerator by including the communications overhead and other such parameters in our analysis. Moreover, we can also establish dependency graphs between the
Memory flowing in and the Memory flowing out of the designated program block. This will allow us to overlap some of the computations with communication resulting in further improvement of the speed-up estimates.

Moreover, the speed-up estimation tool developed as part of this project estimates the speed-up dynamically i.e. while the program is executing. At present, it does not employ any static information in the analysis. This can be a useful direction for future work.

Implementation of the program blocks designated for acceleration on the GPU for the complete CFP95 suite could not be achieved. This was due to the fact that the CFP95 benchmarks employ double precision arithmetic while none of the GPU presently available has the capability to deal with double precision arithmetic. In case of CUDA, the double type was allowed however, any double data was truncated to a float before passing it to the GPU. NVIDIA has announced that it will be launching GPUs with double precision capabilities in the last quarter of 2007. This development has significant implications for our work because we have had to convert the double precision benchmarks to single precision resulting in execution failure, incorrect results and failure to converge in some cases. Therefore, taking advantage of the new double precision enabled GPU can be a useful direction for future work.

Applying loop transformations such as unrolling, fusion and fission in order to improve speed-up potential of programs was not included in the scope of this study. However, during the study we observed that applying loops transformations can result in higher speed-ups for many benchmarks and therefore is a good direction for future work. Of course, we can always achieve the highest speed-ups by transforming the algorithm so that it is optimal for FPGA implementation. However, such transformations are quite complicated and it is not possible or feasible to automate them.
Bibliography

[1] Computer Architecture Research Group at Imperial College (http://comparch.doc.ic.ac.uk/)


[3] NVIDIA CUDA Programming guide


[7] CUDA Release Notes Version 0.8
Appendix A

3S Instrumentation Framework

A.1 Introduction

3S stands for Spacey Stream Splitter. It has a low overhead and the capability to produce the same type of reports as Valgrind. 3S framework is designed to instrument x86 assembly programs. It adds instrumentation stubs in the x86 assembly. These instrumentation stubs communicate with a library of 3S tools to pass them useful runtime information about program flow and other useful parameters such as the number of ticks the last block took to execute.

The instrumentation stubs can be placed around the basic program blocks such as loops and calls. In addition, it can also be placed around every instruction. The former is called program Block-level Instrumentation while the latter is known as Instruction-level Instrumentation. The 3S tools build upon the framework to produce various types of useful reports by employing the information they communicated by the 3S framework. Table-A.1 and Table-A.2 provides a brief description of the various 3S block level instrumentation tools.

The 3S framework works at the assembly level. It is not an object code instrumentation system. This way, 3S does not have to be concerned with its stubs overwriting program instructions or jump targets. In addition it benefits from the basic block identification algorithm already implemented...
in the compiler. Another advantage of working at the assembly level is that the output of the 3S framework is a fully readable assembly code. This adds transparency to the whole procedure of instrumentation. The users can actually see the bits added by the framework.

Moreover, working at the assembly level we benefit from the fact that we do not have to understand the OS dependent executable format and we do not have to deal with the displacement issues that arise when inserting instrumentation stubs into an executable. The framework development and prototyping is also greatly simplified because when working at the assembly level the 3S framework's transformations are simply text transformations that can be applied using the regular expression capabilities of Python or another scripting language.
<table>
<thead>
<tr>
<th>No.</th>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Hotspot</td>
<td>Records the number of ticks and instructions for each block executed</td>
</tr>
<tr>
<td>2</td>
<td>Trace</td>
<td>Blindly writes the name of each block out to a file</td>
</tr>
<tr>
<td>3</td>
<td>Branch</td>
<td>Records branches that occur from the program trace</td>
</tr>
<tr>
<td>4</td>
<td>ControlFlow</td>
<td>Produces a control graph in pdf colouring the nodes in the graph according to how much time is spent in the corresponding block</td>
</tr>
<tr>
<td>5</td>
<td>Fpr02_cfg</td>
<td>Alternative control flow graph, merges sequential blocks into single nodes</td>
</tr>
<tr>
<td>6</td>
<td>Instruction Count</td>
<td>Records the category of instructions executed dynamically</td>
</tr>
<tr>
<td>7</td>
<td>Instruction Count Boundary</td>
<td>Records the category of instructions executed only within the boundary defined by FPGA markers in the original source code</td>
</tr>
<tr>
<td>8</td>
<td>Loopgraph_d</td>
<td>Similar to regex, but produces a pdf graph to illustrate the loops within the trace.</td>
</tr>
<tr>
<td>9</td>
<td>None</td>
<td>Empty tool. Designed to compae 3S to other instrumentation tools</td>
</tr>
<tr>
<td>10</td>
<td>Profile</td>
<td>Records the ticks spent in each block</td>
</tr>
<tr>
<td>11</td>
<td>Regex</td>
<td>Records the program trace and compresses loops using a regular expression syntax</td>
</tr>
<tr>
<td>12</td>
<td>TicksBoundary</td>
<td>Records the ticks spent within each FPGA section</td>
</tr>
</tbody>
</table>

**Table A-1:** Description of block level instrumentation tools
<table>
<thead>
<tr>
<th>No.</th>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Memory</td>
<td>Records the memory access and in which block they occurred</td>
</tr>
<tr>
<td>2</td>
<td>DynamicInstr</td>
<td>Simply counts the number of instructions executed</td>
</tr>
<tr>
<td>3</td>
<td>MemBounadryAcc</td>
<td>Records the amount of memory that is required to be transferred to/from the FPGA section</td>
</tr>
</tbody>
</table>

**Table A-2:** Description of block level instrumentation tools

From an end user's perspective, working with assembly is not as ideal as if the 3S framework worked directly with object code. The main reason for this is that the source code (or assembly) of a program is not always available but the object code always is. However, while the current 3S framework is less generally applicable than a true object code translator, the framework is already considerably more applicable than any single language source code instrumentation solution. This is because the 3S framework is language neutral and the GNU compiler can translate many languages into assembly, including for example: Java, C++, C and Fortran.

**A.2 Getting Started**

The 3S has a command-line interface to allow for ease in use. In addition to the python scripts doing the actual instrumentation, there are several high-level scripts aimed at providing ease of use for frequently instrumented applications such as the SPEC Benchmarks suite. Following are a few of these scripts and their brief description.

**A.2.1 General Instrumentation**

To instrument a target program use:

```
./3SInstrument.py --compiler-opts <comp0pts> --instr-regions 
<sourceFile1> sourceFile2> [tool]
```
--compiler-opts is optional and specifies extra flags to be passed to the compiler

--instr-regions is optional and tells 3S to instrument the sections marked within the src code. The file instrDefs.h must be included and the section in focus marked with the keywords: START_INSTRUMENTATION & END_INSTRUMENTATION

You must specify the full path to the source files. They can be or fortran files. The instrumented executable will be saved as: ./build/target

When you run your instrumented executable, instrumentation results will be written to the working directory. They are named after the tool you used to instrument the program.

### A.2.2 SPEC Instrumentation

A small script to instrument spec benchmarks.

```bash
./instrumentBenchMarks.py --cat <category> --test <benchMarkName> --compileOpts <compileOpts> toolName
```

- **category** is the category to instrument and is `specfp` for floating-point benchmarks and `specint` for integer benchmarks.
- **benchmarkName** is the name of the benchmark to instrument. If this is specified then so must the category.
- **compileOpts** is an optional string passed to the compiler
- **toolName** is the tool to instrument the benchmark with. It is required.

Moreover, If the respective category and benchmarkName are not specified, then all SPEC benchmarks are instrumented.

The resulting executable, both instrumented and non-instrumented, the input/output and small test script (runBenchmark.sh) are placed at:

```
3SPath/build/categoryPath/benchMarkName/
```
Create SPEC Report

This script takes as input a file containing placeholders. Each placeholder represents a different statistic about the the SPEC benchmark. By repeatedly instrumenting the benchmark with various tools the intermediate reports produced are analysed and the placeholders replaced with the actual values calculated from the instrumented benchmark.

`createSpecificReport.py --cat <category> --test <benchmarkName> --compileOpts <compileOpts> --inputReport <inputRpt> --outputReport <outputRpt>`

- `category` is the category to instrument and is given by `specfp` for floating point benchmarks and `specint` for integer benchmarks.
- `benchmarkName` is the name of the benchmark to instrument. If this is specified then so must the category.
- `compileOpts` is an optional string passed to the compiler.
- `inputRpt` specifies the template report file. This contains placeholders to be expanded by the script.
- `outputRpt` specifies the output report file name. This is the input report file but the placeholders have been expanded by the script to actual values.

Some placeholders require sections in the code to be marked as being accelerated using an FPGA. These are usually marked using the macros `BEGIN_FPGA` and `END_FPGA`. However each SPEC benchmark has been analysed and 3 sections of code marked as being possible candidates for acceleration using an FPGA. To select which section to use to calculate the placeholder values, defines are passed to the compiler

`FPGA_MARKERS_ACTIVE_N` uses `BEGIN_FPGA_N` & `END_FPGA_N` where `N` is the number of the FPGA section being analysed.

Generate All Reports

This is a super-script for generating a large report containing both general statistics about each benchmark and also each of the sections marked as being located on the the FPGA.
A.3 Tool Development

Each tool is contained within a separate library. Each library contains a function of the form \_3S\_toolName. When a program is instrumented, calls to the relevant function are inserted at each block (or instruction). The corresponding library is then linked into the final instrumented executable.

The structure is as follows:

./include - Contains header files for the tools.
./libs - Contains source files for the tools.

To make the libraries, do

```
make all
```

The makefile in ./tools/ then makes the various libraries in each of the subdirs. In order to make a library for a tool, the source files need to be specified, this is done in the module.mk files.

In each subdir there is a module.mk file, which details what must be done to build the corresponding library. In the makefile a line including each one of these module.mk files is required.

Note, the subdir, the library and the function must all include the tool name as above.
Appendix B

memBoundaryAcc tool code

```c
#include "tool.h"
#include <map>
#include <fstream>
#include <string>
#include <iostream>
#include <vector>
#include <sstream>
using namespace std;

// this tool uses instruction level instrumentation //
#define _3S_INSTRUMENT_INSTRUCTIONS

// this tool requires FPGA regions to be defined //
#define _3S_FPGA_BOUNDARY_DEFINES

// we record the reads and writes in these maps //
map<void*, unsigned long> reads;
map<void*, unsigned long> writes;

static const char *reportFileNameG = "memBoundaryAcc.3s";
static const char *reportHeaderG = "# MEMORY ACCESSES ACROSS FPGA BOUNDARY.
"
static void _3S__memBoundaryAcc_report();

int FPGADepth = 0;

// memory access counters //
unsigned long stackIn, stackOut, heapIn, heapOut;

// memory limits to determine whether the address lies in stack or heap //
void *stackMin, *stackMax, *heapMin, *heapMax;

extern "C" {
    void _3S__memBoundaryAcc();
    void enter_fpga_block_3s__();
    void exit_fpga_block_3s__();
}
```

66
// handles entry into the FPGA block //
void enter_fpga_block_3s__() {
++FPGADepth;
assert(FPGADepth>=0);
}

// handles exit from the FPGA block //
void exit_fpga_block_3s__() {
--FPGADepth;
assert(FPGADepth>=0);
}

// are we inside an FPGA block? //
bool insideFPGA() {
return FPGADepth>0;
}

// returns address in a standard format //
void* getAddress(const std::string& s) {
std::stringstream addressStream;
addressStream << "0x" << s;
void* address;
addressStream >> address;
return address;
}

// returns the boundaries of stack and heap for the current process //
void getBounds() {
std::stringstream filename;
filename << "/proc/" << getpid() << "/maps";
filename << (filenameString(filename.str()));
std::stringstream addressStream;
addressStream << "0x" << s;
void* address;
addressStream >> address;
}

// Magic number for maximum length of line in maps file
const int maxLineLength=1024;
std::vector<char> lineArray(maxLineLength);
if (mapFile.fail()) {
std::cerr << "Unable to open map file" << std::endl;
exit(1);
}
while(!mapFile.eof()) {
// Magic number for maximum length of line in maps file
const int maxLineLength=1024;
std::vector<char> lineArray(maxLineLength);
mapFile.getline(&lineArray[0], lineArray.size());
const std::string memBegin(line.substr(0, dash));
const std::string memEnd(line.substr(dash+1, space-dash));
if (isStack) {
stackMin = getAddress(memBegin);
stackMax = getAddress(memEnd);
}
if (isHeap) {
heapMin = getAddress(memBegin);
heapMax = getAddress(memEnd);
}
mapFile.close();}
bool closerToStack(void* address)
{
    int heapDiff = (char*)heapMax - (char*)address;
    int stackDiff = (char*)stackMin - (char*)address;

    if (stackDiff < 0) stackDiff = -stackDiff;
    if (heapDiff < 0) heapDiff = -heapDiff;

    if (stackDiff < heapDiff) return true;
    return false;
}

bool isStack(void* address)
{
    if (address >= stackMin && (address <= stackMax)) return true;
    if (closerToStack(address)) return true;
    return false;
}

void _3S__memBoundaryAcc()
{
    static bool initialised = false;
    unsigned long address = 0;

    if (initialised) {
        if (_3S_ISTYPE(*_3S_current_symbol, _3S_SYMBOL_INSTRUCTION)) {
            for (unsigned int i = 0; i < _3S_instruction_parameters; ++i) {
                if (_3S_ISINSTRUCTIONTYPE(_3S_instruction_parameter[i], _3S_INSTRUCTION_MEMORY)) {
                    void* const address = _3S_instruction_parameter[i].value;
                    if (_3S_IOSUBTYPE(_3S_instruction_parameter[i], _3S_INSTRUCTION_READ)) {
                        if (insideFPGA()) {
                            //if the address is being accessed for the first time- record it and //increment the read count, otherwise simply increment the read count
                            reads[address]++;
                            if (reads[address] == 1) {
                                if (isStack(address)) stackIn++;
                                else heapIn++;
                            }
                        }
                        else {
                            //if data is read outside an FPGA that was written in an FPGA
                            if (writes[address] == 1) {
                                if (isStack(address)) stackOut++;
                                else heapOut++;
                            }
                        }
                    }
                }
            }
            if (_3S_IOSUBTYPE(_3S_instruction_parameter[i], _3S_INSTRUCTION_WRITE)) {
                if (insideFPGA()) writes[address]++;
                else writes.erase(address);
            }
        }
    }
    _3S_UPDATE();
    }
else {
    _3S_INITIALISE(_3S__memBoundaryAcc_report);
    getBounds();
    initialised = true;
    }
}
void _3S__memBoundaryAcc_report()
{
    map<void*, unsigned long>::const_iterator iread;
    map<void*, unsigned long>::const_iterator iwrite;
    int entryCount = 1;
    for (iwrite = writes.begin(); iwrite != writes.end(); ++iwrite) {
        if(reads[iwrite->first] > 0 && iwrite->second > 0) {
            if(isStack(iwrite->first)) stackIn--;
            else heapIn--;
            reads.erase(iwrite->first);
        }
    }
    ofstream out(reportFileNameG);
    out << reportHeaderG;
    out << "Entry: " << entryCount << "\t";
    out << "Stack In: " << stackIn << "\t";
    out << "Stack Out: " << stackOut << "\t";
    out << "Heap In: " << heapIn << "\t";
    out << "Heap Out: " << heapOut << "\t";
    out.close();
}

Appendix C

f2c (Fortran to C)

C.1 Introduction

$f2c$ is a Fortran to C converter developed by S. I. Feldman and D. M. Gay from Bell Communications Research working with M. W. Maimone and N. L. Schryer from Carnegie-Mellon University.

Automatic conversion of Fortran to C is desirable in lots of different scenarios. However, the most popular one is when the developers decide to reuse some of the millions of lines of legacy Fortran. Although it is always a possibility to mix C and Fortran code but if one has to make changes to the Fortran code, it gets a bit more tedious. There are quite a few programming tasks which are comparatively more complicated to accomplish in Fortran. For instance:

- Storage Management
- Character Operations
- Arrays of Functions
- Heterogeneous Data-structures
- Operating System calls.
\textit{f2c} is a well-tested tool which has been used to convert various large programs and subroutine libraries to C automatically, including the \textit{PORT3 Subroutine Library}, \textit{MINOS}, and \textit{Schryer's floating-point test} [1]. The floating-point test is of particular interest, as it relies heavily on correct evaluation of parenthesized expressions and is bit-level self-testing.

The \textit{f2c} output contains a large number of quite tedious conversions. In particular, Input/Output statements get expanded into a series of calls on routines in \textit{libI77}, \textit{f77}'s I/O library. Therefore, the C output of \textit{f2c} has the potential to be a maintenance nightmare. It would be much more sensible to maintain the original Fortran, translating it anew each time it changed.

\section*{C.2 f2c Conversions}

The conversions from Fortran to C can be coarsely divided into the following broad categories:

- Conversion of Names
- Conversion of Types
- Conversion of Return Values
- Conversion of Function Arguments

These conversions are dealt with by the \textit{f2c} in the following manner:

\subsection*{C.2.1 Name Conversions}

\textit{f2c} allows for names as long as 50 characters long. It also allows the names to contain underscores. In order to avoid conflict with the respective names in the various library routines, \textit{f2c} appends one or two underscores to the names during conversion from C programming language to Fortran. external names, i.e., the names of Fortran procedures and common blocks, have a single underscore appended if they do not contain any underscores and have a pair of underscores appended if they do contain underscores. Some examples of these name conversions are shown in Table C-1.
Table C-1: Some examples of conversion of names from Fortran to C using the f2c tool. Notice that all the names are forced to lower-case. This is because Fortran programming language, unlike C is not case sensitive.

### C.2.2 Type Conversions

The Conversion of various types from Fortran to C using f2c is shown in Table C-2. The C declarations use types defined in f2c.h, a header file upon which f2c’s translations rely.

<table>
<thead>
<tr>
<th>No.</th>
<th>Fortran</th>
<th>C</th>
<th>standard f2c.h</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>integer*2 x</td>
<td>short int x</td>
<td>short int x</td>
</tr>
<tr>
<td>2</td>
<td>integer x</td>
<td>integer x</td>
<td>long int x</td>
</tr>
<tr>
<td>3</td>
<td>logical x</td>
<td>long int x</td>
<td>long int x</td>
</tr>
<tr>
<td>4</td>
<td>real x</td>
<td>real x</td>
<td>float x</td>
</tr>
<tr>
<td>5</td>
<td>double precision x</td>
<td>doublereal x</td>
<td>double x</td>
</tr>
<tr>
<td>6</td>
<td>complex x</td>
<td>complex x</td>
<td>struct { float r, i; } x;</td>
</tr>
<tr>
<td>7</td>
<td>double complex x</td>
<td>doublecomplex x</td>
<td>struct { double r, i; } x;</td>
</tr>
</tbody>
</table>

Table C-2: Various types, their corresponding representation in Fortran and C and the definition of these representations in the standard f2c.h
By the rules of Fortran, integer, logical, and real data occupy the same amount of memory, and double precision and complex occupy twice this amount; \texttt{f2c} assumes that the types in the C column above are chosen (in \texttt{f2c.h}) so that these assumptions are valid.

### C.2.3 Return Value Conversions

If the return type of a Fortran function is integer, logical or double, the corresponding function in C will have the same return type. If the return type of the Fortran function is real, then the return type of the corresponding function in C depends on whether the user has enforced the -R option or not. The workings of the -R option are explained in the next section.

Subroutines in Fortran are invoked as if they were integer-valued functions whose value selects which alternate return. Alternate return arguments (statement labels) are not passed to the function, but are used to do an indexed branch in the calling procedure.

### C.2.4 Argument List Conversions

In Fortran, all arguments are passed by address. In addition, for every non-function argument that is of type character, an argument giving the length of the value is passed in order to cater for String arguments. These string lengths are long integers and are passed by value.

### C.3 User Options

\texttt{f2c} provides a wide array of options for controlling the behaviour of the conversion with ease. Table C-3 shows several options which proved useful in converting the various CFP92 benchmarks from Fortran to C++.
<table>
<thead>
<tr>
<th>No.</th>
<th>Option</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C++</td>
<td>Output C++ code</td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td>Preserve multidimensional subscripts</td>
</tr>
<tr>
<td>3</td>
<td>R</td>
<td>Do not promote REAL functions and operations to DOUBLE PRECISION</td>
</tr>
<tr>
<td>4</td>
<td>U</td>
<td>Honour the case of variable and external names. Fortran keywords must be in lower case</td>
</tr>
<tr>
<td>5</td>
<td>w</td>
<td>Suppress all warning messages, or, if the option is -w66, just Fortran 66 compatibility warnings</td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td>Make local variables automatic rather than static unless they appear in a DATA, EQUIVALENCE, NAMELIST, or SAVE statement</td>
</tr>
<tr>
<td>7</td>
<td>g</td>
<td>Include original Fortran line numbers in #line lines.</td>
</tr>
<tr>
<td>8</td>
<td>P</td>
<td>Write a file of ANSI (or C++) prototypes for definitions in each input. When reading Fortran from standard input, write prototypes at the beginning of standard output</td>
</tr>
</tbody>
</table>

Table C-3: Various options for f2c employed in SPEC95 conversions
Appendix D

CUDA Architecture

D.1 Introduction

CUDA stands for 'Compute Unified Device Architecture'. It is developed by NVIDIA: A market leader in Programmable Graphic Processor technologies. CUDA is an architecture which enables seamless integration of a GPU (Graphic Processing Unit) with CPU (Central Processing Unit) as a co-processor for accelerated execution of computationally intensive, data-parallel code.

Traditionally, GPGPU (General-Purpose Computation on GPUs) has required the use of a graphics API such as OpenGL, which presents an often inconvenient abstraction for general-purpose parallel computation. Therefore, traditional GPGPU applications are difficult to write, debug, and optimize. CUDA facilitates the programmer in issuing and managing computations on the GPU employed as a co-processor for accelerated execution of computationally intensive, data-parallel code without the need of mapping these computations to a graphics API [1].

Presently, CUDA is available for the GeForce-8 series, QuadroFX-5600/4600 and Tesla Solutions. The GeForce-8 GPUs features hardware support for 32-bit (single precision) floating point vector processors, using the CUDA SDK as API. This essentially means that CUDA supports the C
*double* data type, however on *GeForce-8* series GPUs these types will get demoted to 32-bit floats. This is a severe limitation, especially in the case of scientific applications which promises to be one of the major benefactors of the *GPGPU* drive. However, according to NVIDIA, GPUs supporting (64-bit) Double Precision Floating Point arithmetic in hardware will become available in last quarter of 2007. [2]

**D.2 CUDA Architecture**

*CUDA* Architecture comprise of the software stack and the underlying graphics hardware. The software stack in turn comprises several layers as shown in Figure-D.1.

![CUDA Architecture software stack](image)

**Figure-D.1:** The CUDA Architecture software stack

The hardware of the GPUs that support CUDA is designed in a manner which enables extremely light weight CUDA Driver and CUDA Runtime layers. This innovation is extremely beneficial in obtaining high performance levels.
The CUDA APIs comprise several extension to C programming language. This allows for a faster learning curve as most programmers are familiar with the syntax and workings of the C programming language. The CUDA extension to the C programming language are following:

- Function type qualifiers to specify whether a function executes on the host or on the device and whether it is callable from the host or from the device
- Variable type qualifiers to specify the memory location on the device of a variable
- A new directive to specify how a kernel is executed on the device from the host
- Four built-in variables that specify the grid and block dimensions and the block and thread indices

In addition the CUDA API also provides two high-level mathematical libraries which are used frequently in applications targeted for GPU acceleration. These libraries are:

- CUFFT which provides an optimal implementation of FFT (Fast Fourier Transform) routines
- CUBLAS which provides an optimal implementation of BLAS (Basic Linear Algebra Sub-programs).

CUDA allows the programmer to read or write data at any location in the general DRAM memory, just like in a CPU. This results in increased programming flexibility for both scatter and gather memory operations which are vital for parallel applications. In addition, CUDA features an on-chip shared memory with very fast general read and write access which is employed by the threads to share data. Clever use of this memory results in minimizing general DRAM access and therefore the application becoming less dependent on DRAM memory bandwidth.
D.3 CUDA Programming Model

The CUDA architecture allows the programmer to look at the GPU (called device) as a co-processor to the CPU (called host) capable of executing a large number of threads simultaneously. In order to accelerate an application running on the host, the data-parallel, compute-intensive portions are moved to the device.

More specifically, this is done by isolating the data-parallel, compute-intensive portions of the application in functions and then compiling with the CUDA compiler (called nvcc) resulting in a kernel which is then executed on the device.

The memory space of the host and the device is separate and is called host memory and device memory respectively. The programmer can copy data from the host memory to the device memory and the results from the device memory to the host memory by API calls.

The batch of threads that execute the kernel are organized into thread grid as shown in Figure-D.2. Each element of the grid is in turn called a thread block as illustrated in Figure-D.2. The threads in the same thread block can cooperate with each other by sharing data through a fast shared memory and synchronizing their operations via suitable API calls.

Each thread Block has a block ID while each thread within the thread block has a thread ID resulting in a complex addressing system as illustrated in Figure-D.2. For instance, in case of a two dimensional block of size \((D_x, D_y)\), the thread ID of a thread of index \((x, y)\) is \((x + yD_x)\). In case of a three dimensional block of size \((D_x, D_y, D_z)\), the thread ID of a thread of index \((x, y, z)\) is \((x + yD_x + zDxDy)\).

The number of threads inside a block is limited. However, blocks having the same dimension and running the same kernel can be batched together to form a grid as illustrated in Figure-D.2. This convention enables relatively much larger number of threads to be running simultaneously. This comes at the price of reduced cooperation between threads which share the same grid but belong to different thread blocks within the grid. Each block in the grid is identified by the block ID which has an addressing scheme similar to the addressing scheme of threads in a block.
The function that is executed on the device can access the device DRAM and shared memory through the following memory space:

- Read/Write per thread registers
- Read/Write per thread local memory
- Read/Write per block shared memory
- Read/Write per grid global memory
- Read/Write per grid constant memory
- Read/Write per grid texture memory
Figure-D.2: Thread addressing scheme of CUDA architecture. Threads are grouped together in two/three dimensional Blocks. Blocks, in turn are grouped together into two dimensional grids.
**Figure-D.3:** CUDA memory hierarchy in the order of proximity to the processors
The global, constant and texture memory can be Read/Written by the host and are persistent across kernel launches by the same application. The complete memory model is illustrated in Figure-D.3.

The source code for the tomcatv program from the CFP92 benchmark suite is presented in Appendix E. The source code for the same program as implemented in CUDA is presented in Appendix F. The source code contains ample comments and description for help in understanding. It provides a very useful example in how to accelerate legacy application by using a GPU as a co-processor for executing computationally intensive, data-parallel portions of the code.

On the other hand the source code for the Driver Drowsiness Detection system presented in Appendix H and the CUDA version of the source code for the same application presented in Appendix J provides a useful example of how to accelerate your own programs by using a GPU as a co-processor for executing computationally intensive, data-parallel portions of the code.
Appendix E

tomcatv CUDA code

Following is the CUDA source code of the tomcatv benchmark. After converting the benchmark to C language and reducing the precision to 32-bit (float), we ran it through gprof in order to identify the functions which are taking up most of the execution time. As per the Amdahl's Law, we concentrated our efforts on these functions. We started by identifying the critical loops in these functions, followed by isolating these loops in functions of their own named as fpga1_d, fpga2_d and fpga3_d. All the static variables in these function were removed because CUDA does not allow static variables in functions which are supposed to run on the GPU. Next, we placed the __global marker before the function names and placed them in a separate file named as tomcatv_kernel.cu shown in Listing E-1. In addition, we wrote three more functions which were to serve as the execution stubs for the GPU functions. The execution stubs are responsible for transferring the data to the GPU functions, providing them with the respective execution profiles and finally accumulate the results. These functions were names as fpga1_, fpga2_ and fpga3_ and were placed in a file names tomcatv.cu shown in Listing E-2. The code for the CUDA version of tomcatv is presented below. The original tomcatv source code remain in the tomcatv.c file shown in Listing E-1.
E.1  tomcatv.c

```
#include "f2c.h"

static int c__9 = 9;
static int c__1 = 1;
static int c__3 = 3;
static int c__5 = 5;
static int c__513 = 513;
static float c_b36 = 2.0408163265306123;

int MAIN__()
{
    static char fmt_600[] = "(d20.14,d20.14)";
    static char fmt_1100[] = "(/,,
    2-D ITERATION BEHAVIOR(002)");
    static char fmt_1200[] = "(/,
    002     2-D ITERATION BEHAVIOR(002)");
    static char fmt_1300[] = "(1x,i4,e11.4,e11.4)"
    int i__1, i__2, i__3;
    float d__1, d__2, d__3;
    olist o__1;
    int f_open(olist *), s_wsle(cilist *), do_lio(int *, int *, char *, ftnlen), e_wsle();
    int s_stop(char *, ftnlen), int s_rsle(cilist *), e_rsle(), s_rsfe(cilist *),
    do_fio(int *, char *, ftnlen), e_wsfe(cilist *), e_wsfe();

    static float d__[263169];
    static int iter;
    static int main();
    static float x[263169], y[263169], aa[263169], dd[263169], rx[263169], ry[263169],
    dhi, abx, aby, dla, rma[1000], rym[1000];
    static int itact;
    static cilist io___1 = { 0, 6, 0, 0, 0 };
    static cilist io___2 = { 0, 5, 0, 0, 0 };
    static cilist io___8 = { 0, 6, 0, 0, 0 };
    static cilist io___11 = { 0, 6, 0, 0, fmt_600, 0 };
    static cilist io___14 = { 0, 6, 0, 0, fmt_1100, 0 };
    static cilist io___25 = { 0, 6, 0, fmt_1200, 0 };
    static cilist io___26 = { 0, 6, 0, fmt_1300, 0 };
    static cilist io___27 = { 0, 6, 0, fmt_1300, 0 };
    #define d___ref(a_1,a_2) d__[(a_2)*513 + a_1 - 514]
    #define x_ref(a_1,a_2) x[(a_2)*513 + a_1 - 514]
    #define y_ref(a_1,a_2) y[(a_2)*513 + a_1 - 514]
    #define rr_ref(a_1,a_2) rx[(a_2)*513 + a_1 - 514]
    #define ry_ref(a_1,a_2) ry[(a_2)*513 + a_1 - 514]
    o__1.oerr = 1;
    o__1.ounit = 10;
    o__1.ofnlen = 13;
    o__1.ofnm = "TOMCATV.MODEL";
    o__1.ofm = 0;
    o__1.ofl = 16;
    o__1.ofnc = 0;
    o__1.ofnm = 0;
    o__1.oflnk = 0;
    i__1 = f_open(o__1);
    if (i__1 != 0) {
        goto L999;
    }
    goto L1;
```

84
L999:
    s_wsle(&io___1);
    do_lion(&c__9, &c__1, "FILE \"TOMCATV.MODEL\" DOES NOT EXIST; STOP", (ftnlen)41);
    e_wsle();
    s_stop("", (ftnlen)0);
L1:
    s_rsle(&io___2);
    do_lio(&c__3, &c__1, (char *)&n, (ftnlen)sizeof(int));
    do_lio(&c__3, &c__1, (char *)&itact, (ftnlen)sizeof(int));
    do_lio(&c__5, &c__1, (char *)&dlo, (ftnlen)sizeof(float));
    do_lio(&c__5, &c__1, (char *)&dhi, (ftnlen)sizeof(float));
    do_lio(&c__5, &c__1, (char *)&alpha, (ftnlen)sizeof(float));
    e_rsle();
    if (itact > 1000 || n > 513) {
        s_wsle(&io___8);
        do_lio(&c__9, &c__1, "Please recompile: This version is lacking Storage", (ftnlen)50);
        e_wsle();
        s_stop("", (ftnlen)0);
    }
    i__1 = n;
    for (j = 1; j <= i__1; ++j) {
        i__2 = n;
        for (i__ = 1; i__ <= i__2; ++i__) {
            i__3 = s_rsfe(&io___11);
            if (i__3 != 0) {
                goto L990;
            }
            i__3 = do_fio(&c__1, (char *)&x_ref(i__, j), (ftnlen)sizeof(float));
            if (i__3 != 0) {
                goto L990;
            }
            i__3 = do_fio(&c__1, (char *)&y_ref(i__, j), (ftnlen)sizeof(float));
            if (i__3 != 0) {
                goto L990;
            }
            if (i__3 != 0) {
                goto L990;
            }
            /* L10: */
        }
        goto L2;
    }
L990:
    s_wsle(&io___14);
    do_lion(&c__9, &c__1, "\"TOMCATV.MODEL INCOSISTANT; STOP\", (ftnlen)32);
    e_wsle();
    s_stop("", (ftnlen)0);
L2:
    i__2 = itact;
    for (iter = 1; iter <= i__2; ++iter) {
        rm[iter - 1] = 0.;
        rym[iter - 1] = 0.;
        fpga1_(n, c__513, c_b36, x, y, aa, dd, rx, ry);
        i__1 = n - 1;
        for (j = 2; j <= i__1; ++j) {
            i__3 = n - 1;
            for (i__ = 2; i__ <= i__3; ++i__) {
                d__2 = rxm[iter - 1], d__3 = (d__1 = rx_ref(i__, j), abs(d__1));
                rxm[iter - 1] = max(d__2, d__3);
                d__2 = rym[iter - 1], d__3 = (d__1 = ry_ref(i__, j), abs(d__1));
                rym[iter - 1] = max(d__2, d__3);
            }
        }
    }
    /* L80: */
    i__3 = n - 1;
    for (i__ = 2; i__ <= i__3; ++i__) {
        d___ref(i__, 2) = 1. / dd_ref(i__, 2);
    }
    /* L90: */
E.2 tomcatv.cu

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <cutil.h>
#include <tomcatv_kernel.cu>

__global__ void fpga1d(int, int, float, float*, float*, float*, float*, float*, float*);
__global__ void fpga3d(int, int, float*, float*, float*, float*, float*, float*, float*, float*);
__global__ void fpga2d(int, int, float*, float*, float*, float*, float*, float*, float*, float*);

//...
extern "C" int fpga3_(int n, int nmax, float *aa, float *d, float *dd, float *rx, float *ry)
{
    float* aad;
    float* d_d;
    float* ddd;
    float* rxd;
    float* ryd;

    //reserve device memory for data and copy the data to the device
    int size = 513 * 513 * sizeof(float);
    cudaMalloc((void**)&aad, size);
    cudaMemcpy(aad, aa, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**)&d_d, size);
    cudaMemcpy(d_d, d, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**)&dd, size);
    cudaMemcpy(dd, dd, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**)&rxd, size);
    cudaMemcpy(rxd, rx, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**)&ryd, size);
    cudaMemcpy(ryd, ry, size, cudaMemcpyHostToDevice);

    //Execute the function on the device which does the computation
    dim3 dimBlock(1, 1);
    dim3 dimGrid(n-1, 1);
    fpga3d<<<dimGrid, dimBlock>>>(n, nmax, aad, d_d, ddd, rxd, ryd);

    //copy the result back to host
    cudaMemcpy(d, d_d, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(rx, rxd, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(ry, ryd, size, cudaMemcpyDeviceToHost);

    //free device memory
    cudaFree(aad);
    cudaFree(d_d);
    cudaFree(dd);
    cudaFree(rxd);
    cudaFree(ryd);

    return 0;
}

extern "C" int fpga2_(int n, int nmax, float *aa, float *d, float *rx, float *ry)
{
    float* aad;
    float* d;
    float* rxd;
    float* ryd;

    //reserve device memory for data and result and copy the data to the device
    int size = 513 * 513 * sizeof(float);
    cudaMalloc((void**)&aad, size);
    cudaMemcpy(aad, aa, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**)&d, size);
    cudaMemcpy(d, d, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**)&rxd, size);
    cudaMemcpy(rxd, rx, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**)&ryd, size);
    cudaMemcpy(ryd, ry, size, cudaMemcpyHostToDevice);

    //execute the function on the device which does the computation
    dim3 dimBlock(1, 1);
    dim3 dimGrid(n-1, 1);
    fpga2d<<<dimGrid, dimBlock>>>(n, nmax, aad, d, rxd, ryd);

    //copy the result back to host
    cudaMemcpy(rx, rxd, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(ry, ryd, size, cudaMemcpyDeviceToHost);

    //free device memory
    cudaFree(aad);
    cudaFree(d);
    cudaFree(rxd);
    cudaFree(ryd);

    return 0;
}
extern "C" int fpga1d(int n, int nmax, float rel, float *x, float *y, float *aa, float *dd, float *rx, float *ry) {
    float* xd;
    float* yd;
    float* aad;
    float* ddd;
    float* rxd;
    float* ryd;

    // reserve device memory for data and result and copy the data to the device
    int size = 513 * 513 * sizeof(float);
    cudaMalloc((void**)&xd, size);
    cudaMemcpy(xd, x, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**)&yd, size);
    cudaMemcpy(yd, y, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**)&aad, size);
    cudaMemcpy(aad, aa, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**)&ddd, size);
    cudaMemcpy(ddd, dd, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**)&rxd, size);
    cudaMemcpy(rxd, rx, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**)&ryd, size);
    cudaMemcpy(ryd, ry, size, cudaMemcpyHostToDevice);

    // execute the function on the device which does the computation
    dim3 dimBlock(1, 1);
    dim3 dimGrid(n-1, n-1);
    fpga1d<<<dimGrid, dimBlock>>>(n, nmax, rel, xd, yd, aad, ddd, rxd, ryd);

    // copy the result back to host memory
    cudaMemcpy(aa, aad, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(dd, ddd, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(rx, rxd, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(ry, ryd, size, cudaMemcpyDeviceToHost);

    // free device memory
    cudaFree(xd);
    cudaFree(yd);
    cudaFree(aad);
    cudaFree(ddd);
    cudaFree(rxd);
    cudaFree(ryd);

    return 0;
}

E.3 tomtcatv_kernel.cu

#ifndef _CPP_INTEGRATION_KERNEL_H_
#define _CPP_INTEGRATION_KERNEL_H_

__global__ void fpga1d(int n, int nmax, float rel, float *x, float *y, float *aa, float *dd, float *rx, float *ry) {
    float a, b, c__;
    int i__, j;
    float xx, xy, yx, yy, pxx, qxx, pyy, qyy, pxy, qxy;
    i__ = blockIdx.x + 2;
    j = blockIdx.y + 2;

    #define x_ref(a_1,a_2) x[(a_2-1)*nmax + (a_1-1)]
    #define y_ref(a_1,a_2) y[(a_2-1)*nmax + (a_1-1)]
    #define aa_ref(a_1,a_2) aa[(a_2-1)*nmax + (a_1-1)]
    #define dd_ref(a_1,a_2) dd[(a_2-1)*nmax + (a_1-1)]
    #define rx_ref(a_1,a_2) rx[(a_2-1)*nmax + (a_1-1)]
    #define ry_ref(a_1,a_2) ry[(a_2-1)*nmax + (a_1-1)]

    // for (j = 2; j < n; ++j) {
    //     for (i__ = 2; i__ < n; ++i__) {
        xx = x_ref(i__ + 1, j) - x_ref(i__ - 1, j);
        yy = y_ref(i__, j + 1) - y_ref(i__, j - 1);
        xy = x_ref(i__, j) - x_ref(i__, j - 1);
        yx = y_ref(i__ + 1, j) - y_ref(i__ + 1, j - 1);
    // }
    // }

    return 0;
}

#undef x_ref
#undef y_ref
#undef aa_ref
#undef dd_ref
#undef rx_ref
#undef ry_ref
\[
a = (xy * xy + yy * yy) * .25;
b = (xx * xx + yx * yx) * .25;
c__ = (xx * xy + yx * yy) * .125;
\]
\[
aa_ref(i__, j) = -b;
dd_ref(i__, j) = b + b + a * rel;
pxx = x_ref(i__ + 1, j) - x_ref(i__, j) * 2. + x_ref(i__ - 1, j);
pyy = x_ref(i__, j + 1) - x_ref(i__, j) * 2. + x_ref(i__, j - 1);
pxy = x_ref(i__ + 1, j + 1) - x_ref(i__ + 1, j - 1) - x_ref(i__ - 1, j + 1) + x_ref(i__ - 1, j - 1);
yy = y_ref(i__, j + 1) - y_ref(i__, j) * 2. + y_ref(i__, j - 1);
yref(i__, j - 1) + 1) + y_ref(i__, j - 1) + 1);
rx_ref(i__, j) = a * pxx + b * pyy + c__ * pxy;
ry_ref(i__, j) = a * qxx + b * qyy + c__ * qxy;
\]

#endif // #ifndef _CPP_INTEGRATION_KERNEL_H_

89
Appendix F

ASC (A Stream Compiler)

F.1 Introduction

ASC is an interface for writing or automatically generating hardware based on stream architecture for subsequent implementation on reconfigurable logic devices such as FPGAs (Field Programmable Gate Arrays). ASC syntax is based on the popular C/C++ syntax. In addition, ASC proposes a few semantic extensions such as hardware data types and architectural attributes. The C/C++ roots of ASC greatly simplifies the task of programming or automatically generating hardware based on stream architecture.

Generally the portions of the program which can be termed as computationally intensive are good candidates for execution on the customized hardware. Therefore, this hardware works as a co-processor to the traditional CPU (Central Processing Unit) executing the portion of the code designated by the programmer. Generally this customizable hardware is realized by FPGAs. FPGAs are devices which are programmed by setting the contents of a large number of small SRAMs by appropriate values. More specifically, these SRAMs act like Look-up tables for implementing logic functions. In addition to a large number of these SRAMs, an FPGA also has it's share of flip-flops, dedicated fast carry chains and several buses and tristate buffers.
F.2 Programming Model

Unfortunately abstraction of these FPGA primitives in the C/C++ programming model is not straight-forward. This is where ASC comes in. The various layers of ASC bridge the C/C++ programming model with these FPGA primitives, allowing the programmer to designate portions of his C/C++ code to execute on FPGA instead of CPU. The standard operating procedure for writing a program which partially or completely executes on the FPGA is following:

1. Write a conventional software implementation, preferably in C/C++.
2. Extract critical inner loops from the program written in step 1.
3. Map these inner loops to architectural types and attributes which connect the algorithm to the implementation in the FPGA. This usually entails modifying the original C/C++ code to ASC descriptions which consist of C++ syntax and ASC specific semantics. ASC is designed to minimize the amount of effort required for this step, while still keeping the performance close to optimal.
4. Compile the ASC description (the stream.cxx file) including PAM-Blox II and the ASC architecture generation layer (all within C++ libraries) using a standard version of gcc. 5. Run the resulting executable to generate a netlist (usually in EDIF format). The netlist contains a description of the entire circuit consisting of the FPGA primitives (LUTs, flip-flops, etc.), and their interconnections. 
5. The Netlist passes through FPGA vendor CAD tools which take care of mapping the circuit to the particular FPGA, placing the elements into specific locations on the chip, and setting the routes within the FPGA to achieve the circuit described by the input netlist. The FPGA vendor tools generate a configuration file (bitstream) which can be downloaded into the FPGA on demand within milliseconds.
6. Take the program from step 2, and replace the inner loops with ASC runtime calls that usually invoke FPGA device driver commands that send data to and from the FPGA. The resulting program runs on a microprocessor, using the FPGA to execute the critical inner loops at high speed. Thus, in this setup the microprocessor is the master and the FPGA is the slave. The processor sets the configuration of the FPGA and can adapt the FPGA to dynamic events. In theory, the FPGA could also send interrupts to the processor, however, this feature is not exploited in the current version of ASC.
**F.3 User Guide**

The directory structure of ASC is depicted in Table F-1.

<table>
<thead>
<tr>
<th>No.</th>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>doc</td>
<td>Contains the ASC Manual</td>
</tr>
<tr>
<td>2</td>
<td>examples</td>
<td>A few stream examples.</td>
</tr>
<tr>
<td>3</td>
<td>inc</td>
<td>Include files for PamDC, a gate level CAD tool for FPGAs, PAM-Blox II, and ASC</td>
</tr>
<tr>
<td>4</td>
<td>lib</td>
<td>Libraries such as libPamDC.a, libASC.a, etc</td>
</tr>
<tr>
<td>5</td>
<td>bin</td>
<td>Executables of PamDC tools</td>
</tr>
</tbody>
</table>

**Table L-1:** Directory Structure of ASC system

**Listing F-1:** Steps to compile a simple “hello world” example and run a simulation using the values from the data file

```
% cd examples/simple
% ls
data
makefile
stream.cxx
% less stream.cxx
#include "asc.h"
main(int argc, char **argv) {
    STREAM_START; // ASC code start
    // Hardware Variable Declarations
    HWint in(IN);
    HWint out(OUT);
    HWint tmp(TMPL);
    STREAM_LOOP(16);
    tmp = (in << 1) + 55;
    out = tmp;
    printf("Hello World\n");
    STREAM_END; // ASC code end
}
% make stream
% less stream.cxx -lASC -lPamDC ... -o stream.exe
% make sim
stream.exe --quiet --simulate < data
Hello World.
55 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 ...
```
Appendix G

Drowsy Driver Detection code

// System Includes //
#include <stdlib.h>
#include <stdio.h>

// OpenCV Includes //
#include <cv.h>
#include <highgui.h>

// Image Data Structures //
IplImage* in;
IplImage* out;
IplImage* outedge;

// Various Pixels used in algorithms //
CvScalar s, white, black;

// Vertical Projection Array and related variables //
int PV[2000];
int avgPV, left, right;

// Horizontal Projection Array and related variables //
int PH[2000];
int avgPH, top, bottom;

// Miscellaneous Variables //
int avgPHF, upper, lower;
int complexity;

// Function Declarations //
void getPV(void);
void getRL(void);
void getPH(void);
void getTB(void);
void getUL(void);
void getCx(void);
```c
int main(int argc, char *argv[]) {
    // Default values to white and black pixel
    white.val[0] = 255; white.val[1] = 0; white.val[2] = 0;
    black.val[0] = 000; black.val[1] = 0; black.val[2] = 0;

    // process user input
    if(argc<2) {
        printf("Usage: main <image-file-name>\n\n");
        exit(0);
    }

    in = cvLoadImage(argv[1]);
    if(!in){
        printf("Could not load image file: %s\n",argv[1]);
        exit(0);
    }

    // create an image call out and store the input image in it after conversion to grayscale
    out = cvCreateImage(cvSize(in->width,in->height),IPL_DEPTH_8U,1);
    cvCvtColor(in,out,CV_RGB2GRAY);

    // get vertical projection and calculate average
    getPV();

    // get right and left boundaries of face
    getRL();

    // get horizontal projection and calculate average
    getPH();

    // get top and bottom boundaries of face
    getTB();

    // Get top and bottom boundaries of eyes
    getUL();

    // perform edge detection and apply complexity function on eyes
    getCx();

    // super-impose the results of the various filters over the image
    for(int h=0; h<outedge->height; h++) {
        for(int w=0; w<outedge->width; w++) {
            // put lines at the detected facial boundaries
            if(w==left || w==right) cvSet2D(outedge,h,w,white);
            if(h==top || h==bottom) cvSet2D(outedge,h,w,white);

            // put lines at the detected eye-region boundaries
            if(h==upper || h==lower) cvSet2D(outedge,h,w,white);

            // put the horizontal projection values in a strip at the bottom of the image
            if(h > (outedge->height-10)) {s.val[0] = (int)PH[h];
                cvSet2D(outedge,h,w,s);
            }

            // put the vertical projection values in a strip at the right of the image
            if(w > (outedge->width-10)) {s.val[0] = (int)PV[w];
                cvSet2D(outedge,h,w,s);
            }
        }
    }

    // save image to file
    cvSaveImage("openbin.bmp",outedge);

    // release the image data structures
    cvReleaseImage(&in);
    cvReleaseImage(&out);
    cvReleaseImage(&outedge);
    return 0;
}
```
void getPV(void)
{
    // initialize the vertical projection array and the vertical projection average variables
    for(int w = 0; w < out->width; w++) PV[w] = 0;
    avgPV = 0;
    for(int w = 0; w < out->width; w++) {
        for(int h = 0; h < out->height; h++) {
            // get the value of the pixel at (w,h) //
            s = cvGet2D(out,h,w);
            // accumulate the pixel values in the respective projection array location //
            PV[w] += (int)s.val[0];
        }
        // normalize the projection value //
        PV[w] /= out->height;
        // accumulate the normalized projection value //
        avgPV += PV[w];
    }
    // calculate the average projection value //
    avgPV /= out->width;
}

void getRL(void)
{
    int i=0;
    // from the left the first bright spot in the image is the start of the face-region //
    for(; i < out->width; i++) {
        if(PV[i] > avgPV) {
            left = i;
            break;
        }
    }
    // and the last bright spot is the end of the face-region //
    for(; i < out->width; i++) {
        if(PV[i] < avgPV) {
            right = i;
            break;
        }
    }
}

void getPH(void)
{
    // initialize the horizontal projection array and the vertical projection average variables//
    for(int i=0; i<out->height; i++) PH[i] = 0;
    avgPH = 0;
    for(int h = 0; h < out->height; h++) {
        for(int w = left; w < right; w++) {
            // get the value of the pixel at (w,h) //
            s = cvGet2D(out,h,w);
            // accumulate the pixel values in the respective projection array location //
            PH[h] += (int)s.val[0];
        }
        // normalize the projection value //
        PH[h] /= (right-left);
        // accumulate the normalized projection value //
        avgPR += PH[h];
    }
    // calculate the average projection value //
    avgPR /= out->height;
}

void getTB(void){
    int i = 0;
    for(; i < out->height; i++) {
        // from the top the first bright spot in the image is the start of the face-region //
        if(PH[i] > avgPH) {
            top = i;
            break;
        }
    }
    // and the last bright spot is the end of the face-region //
    for(; i < out->height; i++) {
        if(PH[i] < avgPH) {
            bottom = i;
            break;
        }
    }
}

void getUL(void){
    avgPHF = 0;
    // calculate the average horizontal projection within the facial boundaries //
    for(int i = top; i < bottom; i++) {
        avgPHF += PH[i];
    }
    avgPHF /= (int)(bottom-top);
    int i = top;
    // from the top the first dark strip will be the hair //
    for(; i < bottom; i++) if(PH[i] < avgPHF) break;
    for(; i < bottom; i++) if(PH[i] > avgPHF) break;
    // the second dark strip will be the eye-brows //
    for(; i < bottom; i++) if(PH[i] < avgPHF) break;
    for(; i < bottom; i++) if(PH[i] > avgPHF) break;
    // the third dark strip will be the eyes //
    for(; i < bottom; i++) {
        if(PH[i] < avgPHF) {
            upper = i;
            break;
        }
    }
    for(; i < bottom; i++) {
        if(PH[i] > avgPHF) {
            lower = i;
            break;
        }
    }
}

void getCx(void){
    // create image data structure //
    outedge = cvCreateImage(cvSize(out->width,out->height),IPL_DEPTH_8U,1);
    // copy the current out image into the new image data structure //
    cvCopyImage(out,outedge);
    CvScalar a11,a12,a13,
    a21,a22,a23,
    a31,a32,a33;
    int lefta= left+1;
    int righta= right-1;
    int uppera= upper+1;
    int lowera= lower-1;
    int asum,bsum,sum;
    int m=(righta-lefta);
    int diff=0;
    CvScalar c1,c2;
for(int h = uppera; h < lowera; h++) {
    for(int w = lefta; w < righta; w++) {
        // get the 3x3 pixel block with the current pixel at the center //
        a11=cvGet2D(out,(h-1),(w-1));
        a21=cvGet2D(out,(h  ),(w-1));
        a31=cvGet2D(out,(h+1),(w-1));
        a12=cvGet2D(out,(h-1),( w  ));
        a22=cvGet2D(out,(h  ),( w  ));
        a32=cvGet2D(out,(h+1),( w  ));
        a13=cvGet2D(out,(h-1),(w+1));
        a23=cvGet2D(out,(h  ),(w+1));
        a33=cvGet2D(out,(h+1),(w+1));

        // apply the prewit edge detection operator //
        asum = (int)(-a11.val[0]+a13.val[0]-a21.val[0]+a23.val[0]-a31.val[0]+a33.val[0]);
        bsum = (int)( a11.val[0]+a12.val[0]+a13.val[0]-a31.val[0]-a32.val[0]-a33.val[0]);

        sum = abs(asum) + abs(bsum);
        if(sum > 255) cvSet2D(outedge,h,w,white);
        else cvSet2D(outedge,h,w,black);

        if(sum > 255) CV_IMAGE_ELEM(outedge, uchar, h, w) = 255;
        else CV_IMAGE_ELEM(outedge, uchar, h, w) = 0;

        c1=cvGet2D(outedge,h,w);
        c2=cvGet2D(outedge,h,w+1);
        diff = (int)(c1.val[0]-c2.val[0]);

        // calculate the complexity function value //
        if(h<(m/2)) complexity += abs(diff)*(h);
        else complexity += abs(diff)*(m-h);
    }
}
}
Appendix H

Drowsy Driver Detection ASC code

```c
#include "asc.h"
#include <math.h>
#include <iomanip>

int main(int argc, char **argv) {
    STREAM_START;
    HWint In(IN, 24, UNSIGNED);
    HWint w(IN, 24, UNSIGNED);
    HWint h(IN, 24, UNSIGNED);
    HWint Sum(REGISTER, 32, UNSIGNED);
    HWint Gum(REGISTER, 32, SIGNMAGNITUDE);
    HWint Yum(REGISTER, 32, SIGNMAGNITUDE);
    HWint Pum(REGISTER, 24, UNSIGNED);
    HWint Ppr(TEMP, 24, UNSIGNED);
    HWint Pdf(TEMP, 25, SIGNMAGNITUDE);
    HWint Edg(REGISTER, 1, UNSIGNED);
    HWint Eye(REGISTER, 4, UNSIGNED);
    HWint Out(OUT, 24, UNSIGNED);
    HWint filter00(TMP, 25, SIGNMAGNITUDE);
    HWint filter01(TMP, 25, SIGNMAGNITUDE);
    HWint filter02(TMP, 25, SIGNMAGNITUDE);
    HWint filter10(TMP, 25, SIGNMAGNITUDE);
    HWint filter11(TMP, 25, SIGNMAGNITUDE);
    HWint filter12(TMP, 25, SIGNMAGNITUDE);
    HWint filter20(TMP, 25, SIGNMAGNITUDE);
    HWint filter21(TMP, 25, SIGNMAGNITUDE);
    HWint filter22(TMP, 25, SIGNMAGNITUDE);
    HWint filterSum(TMP, 25, UNSIGNED);
    HWint filterOut(TMP, 24, SIGNMAGNITUDE);
    HWint comp(REGISTER, 32, UNSIGNED);
    STREAM_OPTIMIZE=LATENCY;
    Sum = 0;

    STREAM_END;
    return 0;
}
```
Sum = IF(w==2,0,Sum+In);
Gum = 0;
Gum = (Sum>>1)-6750;
Yum = 0;
Yum = IF(w==149,Gum,Yum);
Pum = 0;
Pum = IF(Yum>255,255,0);
Ppr = prev(Pum,1);
Pdf = Pum-Ppr;
Edg = IF(Pdf == 0,0,1);
Eye = IF(Edg == 1, Eye+1, Eye);
filter22 = In;
filter21 = prev(filter22,1);
filter20 = prev(filter21,1);
filter12 = prev(filter22,147);
filter11 = prev(filter12,1);
filter10 = prev(filter11,1);
filter02 = prev(filter12,147);
filter01 = prev(filter02,1);
filter00 = prev(filter01,1);
filter20 = -filter20;
filter21 = -filter21;
filter22 = -filter22;
filterSum = filter00 + filter01 + filter02 + filter20 + filter21 + filter22;
comp = IF(Eye==4 && filterOut==0,comp,comp+1);
Out = IF(Eyes==4,filterOut,In);
STREAM_END;
return 0;
Appendix I

Drowsy Driver CUDA code

1.1 algo.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <cv.h>
#include <highgui.h>
#include <math.h>

// variables for default pixel values to be used in the algorithm //
CvScalar s, white, black;

// opencv image data structures //
IplImage* in;
IplImage* out;
IplImage* outedge;

// vertical projection values and average vertical projection //
int PV[2000];
int avgPV;

// horizontal projection values and average vertical projection //
int PH[2000];
int avgPH;

// facial boundaries //
int left;
int right;
int top;
int bottom;

// average horizontal projection within the facial boundaries //
int avgPHF;

// eye region boundaries //
int upper, lower;
```
// pointers for raw image data //
uchar *out_data;
uchar *outedge_data;

// supporting functions //
void getRL(void);
void getTB(void);
void getUL(void);

// execution stubs for CUDA functions //
extern "C" void getCx(uchar*, uchar*, int, int, int, int, int, int, int);
extern "C" int getPV(uchar*, int*, int, int, int);
extern "C" int getPH(uchar*, int*, int, int, int);

int main(int argc, char *argv[])
{
// initialize the default pixel values //
white.val[0] = 255; white.val[1] = 0; white.val[2] = 0;
black.val[0] = 000; black.val[1] = 0; black.val[2] = 0;

// process user input //
if(argc<2) {
    printf("Usage: main <image-file-name>\n\n");
    exit(0);
}

// read the input image //
in = cvLoadImage(argv[1]);
if(!in){
    printf("Could not load image file: %s\n",argv[1]);
    exit(0);
}

// convert image to grayscale //
out = cvCreateImage(cvSize(in->width,in->height),IPL_DEPTH_8U,1);
cvtColor(in,out,CV_RGB2GRAY);

// get raw image data //
cvGetRawData(out, (uchar**)&out_data);

// get vertical projection and calculate average //
uchar* data;
avgPV = getPV(out_data, PV, out->width, out->height);

// get right and left boundaries of face //
getRL();

// get horizontal projection and calculate average //
avgPH = getPH(out_data, PH, out->width, out->height, left, right);

// get top and bottom boundaries of face //
getTB();

// Get top and bottom boundaries of eyes //
getUL();

// perform edge detection and apply complexity function on eyes //
int righta = right-1;
int lefta = left+1;
int uppera = upper+1;
int lowera = lower-1;
outedge = cvCreateImage(cvSize(righta-lefta,lowera-uppera),IPL_DEPTH_8U,1);
cvGetRawData(outedge, (uchar**)&outedge_data);
getCx(out_data, outedge_data, out->width, out->height, lefta, righta, uppera, lowera);

// save output image //
cvSaveImage("openbin.bmp",outedge);

// clean-up memory //
cvReleaseImage(in);
cvReleaseImage(out);
cvReleaseImage(outedge);

return 0;
}
void getRL(void) {
    int i = 0;
    for(; i < out->width; i++) {
        if(PV[i] > avgPV) {
            left = i;
            break;
        }
    }
    for(; i < out->width; i++) {
        if(PV[i] < avgPV) {
            right = i;
            break;
        }
    }
}

void getTB(void) {
    int i = 0;
    for(; i < out->height; i++) {
        if(PH[i] > avgPH) {
            top = i;
            break;
        }
    }
    for(; i < out->height; i++) {
        if(PH[i] < avgPH) {
            bottom = i;
            break;
        }
    }
}

void getUL(void) {
    avgPHF = 0;
    for(int i = top; i < bottom; i++) {
        avgPHF += PH[i];
    }
    avgPHF /= (int)(bottom - top);
    int i = top;
    for(; i < bottom; i++) if(PH[i] < avgPHF) break;
    for(; i < bottom; i++) if(PH[i] > avgPHF) break;
    for(; i < bottom; i++) if(PH[i] < avgPHF) break;
    for(; i < bottom; i++) if(PH[i] > avgPHF) break;
    for(; i < bottom; i++) {
        if(PH[i] < avgPHF) {
            upper = i;
            break;
        }
    }
    for(; i < bottom; i++) {
        if(PH[i] > avgPHF) {
            lower = i;
            break;
        }
    }
}
I.2 algo.cu

#include "algo_kernel.cu"

// declare CUDA functions
__global__ void getCxd(uchar*, uchar*, int, int, int, int, int, int);
__global__ void getPVd(uchar*, int*, int, int, int);
__global__ void getPHd(uchar*, int*, int, int, int, int, int);
extern "C" void getCx(uchar* out, uchar* outedge, int width, int height, int lefta, int righta, int uppera, int lowera);
extern "C" void getPV(uchar* out, int* PV, int width, int height);
extern "C" void getPH(uchar* out, int* PH, int width, int height, int left, int right);

{ // pointers to data in GPU //
  uchar* outd;
  uchar* outedged;

  // transfer data to GPU //
  int size1 = width * height * sizeof(uchar);
  cudaMalloc((void**)&outd, size1);
  cudaMemcpy(outd, out, size1, cudaMemcpyHostToDevice);
  int size2 = (lowera-uppera) * (righta-lefta) * sizeof(uchar);
  cudaMalloc((void**)&outedged, size2);

  // set the execution profile and run the CUDA function //
  dim3 dimGrid(righta-lefta, lowera-uppera);
  dim3 dimBlock(1,1);
  getCxd<<<dimGrid, dimBlock>>>(outd, outedged, width, height, lefta, righta, uppera, lowera);

  // get results from GPU //
  cudaMemcpy(outedge, outedged, size2, cudaMemcpyDeviceToHost);
  cudaFree(outd);
  cudaFree(outedged);
}

{ // pointers to data in GPU //
  uchar* outd;
  int* PVd;

  // transfer data to GPU //
  int size1 = width * height * sizeof(uchar);
  cudaMalloc((void**)&outd, size1);
  cudaMemcpy(outd, out, size1, cudaMemcpyHostToDevice);
  int size2 = width * sizeof(int);
  cudaMalloc((void**)&PVd, size2);

  // set the execution profile and run CUDA function //
  dim3 dimGrid(1,1);
  getPVd<<<dimGrid, dimBlock>>>(outd, PVd, width, height);

  // get the results //
  cudaMemcpy(PV, PVd, size2, cudaMemcpyDeviceToHost);
  cudaFree(outd);
  cudaFree(PVd);

  // calculate the average vertical projection //
  int avg=0;
  for(int i=0; i<width; i++) avg += PV[i];
  return avg/width;
}

{ // pointers to GPU data //
  uchar* out;
  int* PHd;

  // calculate the average //
  int avg=0;
  for(int i=0; i<width; i++) avg += PH[i];
  return avg/width;
}

103
```c
int size1 = width * height * sizeof(uchar);
cudaMalloc((void**)&outd, size1);
cudaMemcpy(outd, out, size1, cudaMemcpyHostToDevice);

int size2 = height * sizeof(int);
cudaMalloc((void**)&PHd, size2);
dim3 dimBlock(1,1);
dim3 dimGrid(height,1);
getPHd<<<dimGrid, dimBlock>>>(outd, PHd, width, height, left, right);
cudaMemcpy(PH, PHd, size2, cudaMemcpyDeviceToHost);
cudaFree(outd);
cudaFree(PHd);

int avg=0;
for(int i=0; i<height; i++) avg += PH[i];
return avg/height;
```

```c
I.3 algo_kernel.cu

#ifndef _CPP_INTEGRATION_KERNEL_H_
#define _CPP_INTEGRATION_KERNEL_H_
#include <cv.h>

__global__ void getCxd(uchar* out, uchar* outedge, int width, int height, int lefta, int righta, int uppera, int lowera)
{
    uchar a11,a12,a13, a21, a23, a31,a32,a33;
    int asum,bsum,sum;
    int widtha = righta-lefta;
    int heighta = lowera-uppera;
    int m=(righta-lefta);
    int diff=0;
    CvScalar c1,c2;
    int w = blockIdx.x + lefta;
    int h = blockIdx.y + uppera;
    //for(int h = uppera; h < lowera; h++) {
    //for(int w = lefta; w < righta; w++) {
        a11=out[(h-1)*width + (w-1)];
        a21=out[(h)*width + (w-1)];
        a31=out[(h+1)*width + (w-1)];
        a12=out[(h-1)*width + (w)];
        a32=out[(h+1)*width + (w)];
        a13=out[(h-1)*width + (w+1)];
        a23=out[(h)*width + (w+1)];
        a33=out[(h+1)*width + (w+1)];
        asum = (int)(-a11+a13-a21+a23-a31+a33);
        bsum = (int)( a11+a12+a13-a31-a32-a33);
        sum = abs(asum) + abs(bsum);
        if(sum > 255) outedge[(h-uppera)*widtha + (w-lefta)] = 255;
        else outedge[(h-uppera)*widtha + (w-lefta)] = 0;
    //}
    //}
}
```

104
__global__ void getPVd(uchar* out, int* PV, int width, int height)
{
    int w = blockIdx.x;
    //for(int w = 0; w < width; w++) {
        for(int h = 0; h < height; h++) PV[w] += (int)out[(h * width) + w];
        PV[w] /= height;
    //}
}

__global__ void getPHd(uchar* out, int* PH, int width, int height, int left, int right)
{
    int h = blockIdx.x;
    //for(int h = 0; h < height; h++) {
        for(int w = left; w < right; w++) PH[h] += (int)out[(h * width) + w];
        PH[h] /= (right-left);
    //}
}

#endif