

Automatic conversion of source code for C to CUDA C

By

Parth R. Trivedi

Roll No: 10MCEC25



**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INSTITUTE OF TECHNOLOGY**

NIRMA UNIVERSITY

AHMEDABAD-382481

May 2012

Automatic conversion of source code for C to CUDA C

Major Project

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology in Computer Science and Engineering

By

Parth R. Trivedi

Roll No: 10MCEC25



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INSTITUTE OF TECHNOLOGY

NIRMA UNIVERSITY

AHMEDABAD-382481

May 2012

Undertaking for Originality of the Work

I, Parth R. Trivedi, Roll. No. 10MCEC25 , give undertaking that the Major Project entitled Automatic conversion of source code for C to CUDA C submitted by me, towards the partial fulfillment of the requirements for the degree of Master of Technology in Computer Science & Engineering of Nirma University, Ahmedabad, is the original work carried out by me and I give assurance that no attempt of plagiarism has been made. I understand that in the event of any similarity found subsequently with any published work or any dissertation work elsewhere; it will result in severe disciplinary action.

Signature of Student

Date: -----

Place: -----

Certificate

This is to certify that the Major Project entitled "Automatic conversion of source code for C to CUDA C" submitted by Parth R. Trivedi (10MCEC025), towards the partial fulfillment of the requirements for the degree of Master of Technology in Computer Science and Engineering of Nirma University, Ahmedabad is the record of work carried out by him under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Dr. S.N. Pradhan
Guide, Professor,
Department Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Prof. D. J. Patel
Professor and Head,
Department of Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Dr. K. Kotecha
Director,
Institute of Technology,
Nirma University, Ahmedabad

Acknowledgements

It gives me great pleasure in expressing thanks and profound gratitude to my guide Dr. S.N. Pradhan, Professor and M.Tech. Coordinator, Department of Computer Science and Engineering, Institute of Technology, Nirma University, Ahmedabad for his valuable guidance and continual encouragement throughout part one of the Major project. I am heartily thankful to him for his time to time suggestions and the clarity of the concepts of the topic that helped me a lot during this study.

I would like to extend my gratitude to Prof. D.J.Patel, H.O.D., Department of Computer Science and Engineering, Institute of Technology, Nirma University, Ahmedabad for fruitful discussions and valuable suggestions during meetings and for their encouragement.

I would like to thank Dr. Ketan Kotecha, Honorable Director, Institute of Technology, Nirma University, Ahmedabad for providing basic infrastructure and healthy research environment.

I would like to thank Prof. Samir Patel who gave me his precious time for discussions and evaluation of my project.

I would also thank my Institution, all my faculty members in Department of Computer Science and my colleagues without whom this project would have been a distant reality. Last, but not the least, no words are enough to acknowledge constant support and sacrifices of my family members because of whom I am able to complete my dissertation work successfully.

- Parth R. Trivedi

10MCEC25

Abstract

As today in so many fields, computation is the main part of the algorithm and takes too much time in execution of the algorithm, it is necessary to parallelize the computation or reduce execution time. GPUs are widely used in (HPC) High Performance Computing. To achieve speedup, either we can increase clock frequency or multiple computation cores on the same chip. The clock speeds have reached the physical limit, so the use of many cores is the only way left to achieve speedup. As the GPU is growing demand of the Game Industry and large scientific computations, efforts have been made to take advantages to gain maximum utilization of the GPUs in computation. Though GPUs are widely used in Supercomputers today, they are not code transparent because one has to sit and code the algorithms in CUDA C to run them on GPU. So if we can have some middleware that converts the C programs to CUDA, the end user gets transparency.

I tried to develop a prototype compiler using ANTLR in visual studio that converts the C programs in CUDA C language. The thesis describes the literature survey in CUDA, different performance optimization strategies to reduce execution time, the Pattern approach to develop a translator for source code to source code translation on the basis of selection of codes using patterns, platforms to code such translator and platform comparison and choice and algorithm of translation. The Compiler Architecture and its implementation details are widely described in thesis. The thesis describes implementation of the complete C2CUDATranslator, testing and analysis of the developed compiler. The compiler takes input of C program and generates CUDA program. The thesis demonstrates the pattern approach for language to language translation and the compiler flow architecture. C2CUDAranslator covers a new way or a framework to implement new analysis algorithms to detect dependencies in the code. The thesis also covers neural network design for compiler learning and optimization of the translated code. The Neural Network helps compiler to take decision for selection of transformation and translation.

Finally, the thesis covers outcome of the compiler, converted programs list, evaluation using parboil benchmark suite, performance graph of converted programs. It is concluded that the C2CUDATranslator saves 95% of the development time in selected cases.

Contents

Undertaking for Originality of the Work	iii
Certificate	iv
Acknowledgements	v
Abstract	vi
Contents	x
List of Tables	xi
List of Figures	xiii
Abbreviations	xiv
1 Introduction	1
1.1 Objective of the Work	2
1.2 Scope of the Work	2
1.3 Motivation of the Work	2
1.4 Thesis Organization	2
2 Literature Survey	4
2.1 Design goal	5
2.1.1 Advantages	6
2.2 Limitation	7
2.3 CUDA Programming Model	7
2.4 CUDA Memory Model	9
2.5 Hardware Implementation: Memory Architecture	11
2.6 Hardware Implementation: Execution Model	12
3 Performance Optimization Strategies	13
3.1 Maximize Utilization	14
3.1.1 Application Level	14
3.1.2 Device Level	14
3.1.3 Multiprocessor Level	15
3.2 Maximize Memory Throughput	16

3.2.1	Data Transfer between Host and Device	16
3.2.2	Optimize Memory Access Patterns	19
3.3	Execution configuration optimization	21
3.3.1	Occupancy	22
3.4	Maximize Instruction Throughput	23
4	Problem Definition	24
4.1	Part 1: C level Parallelization	24
4.1.1	Loop interchange transformation	24
4.1.2	Loop embedding transformation	25
4.1.3	Pattern Structure	26
4.1.4	Condition Block	28
4.1.5	Result Block	28
4.2	Part 2: C to CUDAC Transformation	30
4.2.1	Examples	30
4.2.2	Patterns	31
4.2.3	Use of #pragma unroll	34
4.3	Issues	34
4.4	Use of Patterns	35
5	Automatic code conversion Algorithm	37
5.1	The Algorithm	37
6	Different Compiler Platforms	39
6.1	Introduction to ANTLR	39
6.2	Introduction to Open64	40
6.3	Introduction to SUIF	41
7	Platform Comparison and reason of choice	42
7.1	What exactly does ANTLR 3 do?	42
7.1.1	ANTLR 3	44
7.1.2	Target languages	44
7.1.3	Why should I use ANTLR 3?	44
8	Compiler	45
8.1	Compiler Modules	46
8.2	C2CUDATranslator Phases	47
8.3	C2CUDATranslator Flow	47
8.4	Preprocessor	49
8.5	Code Transformation in Kernel Region	49
8.6	Compiler Style	50
8.7	Pattern Learning	51
8.8	Dependency Checker	53
8.8.1	Dependency Analysis	54
8.8.2	Range Check	56
8.8.3	Recursion Check	56
8.9	Data Structures Used	57

8.10 Interfaces used	58
8.11 C2CUDATranslator Features	58
8.12 Grammar Example	59
8.13 Pattern Example	62
8.14 Class Diagram	63
8.15 Support Function Example	65
9 Optimization	66
9.1 How to use Visual Profiler?	66
9.1.1 Getting started for optimization	66
9.2 Maximize Memory Throughput	67
9.2.1 Insertion of keywords for placing in memory	67
9.2.2 Use of CUDAMemSet	67
9.2.3 Asynchronous Transfers and Overlapping Transfers with Computation	67
9.3 Decision Making	69
9.4 Artificial Neural Network	69
9.4.1 Compiler Learning	70
10 Outcome	71
10.1 Input	72
10.2 Output	74
10.2.1 Output Screenshot	74
10.2.2 Output in words	75
10.2.3 Parallelization Achieved	78
10.2.4 Output of Converted Program (C2CUDATranslator)	80
10.2.5 Output of Converted Program (Handwritten)	80
11 Evaluation	81
11.1 CUDA benchmarks for evaluating the C2CUDATranslator compiler	81
11.1.1 Evaluation of C2CUDATranslator	82
12 Conclusion	83
12.1 Conclusion	83
13 Future Scope	84
13.1 Future Scope	84
13.1.1 Upcoming Features	84
13.1.2 Pre-processor phase	84
13.1.3 Parallelization/Analysis Phase	85
13.1.4 Compiler for OpenMP	85
13.1.5 Compiler Learning	85
Web References	86
References	88
Index	89

A	Output Screenshots	90
B	User's Guide	91
B.1	Getting Started	91
B.1.1	Input and Output of C2CUDATranslator	91
B.2	C2CUDATranslator input Details	91
B.2.1	Kernel Outlining	91
B.2.2	Kernel Variables	92
B.2.3	Kernel local Variables	93
C	Developer's Guide	94
C.1	Getting Started	94
C.1.1	C2CUDATranslator Project Structure	94
C.2	C2CUDATranslator development Details	95
C.2.1	Analysis Framework	95
C.2.2	Translation Framework	95
C.2.3	Use framework	95

List of Tables

I	C2CUDA Translator Features	59
I	CUDA benchmarks for evaluating the C2CUDAranslator compiler	82
I	Upcoming Features	84

List of Figures

2.1	CUDA Software Stack	5
2.2	CUDA programming model	8
2.3	CUDA Memory model	9
2.4	A set of SIMD processor with memory architecture	11
3.1	A coalescing memory access	20
3.2	coalescing memory access: divergent warp	20
3.3	Non- Sequential memory access	21
3.4	A coalescing memory access	22
3.5	A coalescing memory access	22
3.6	A coalescing memory access	23
6.1	ANTLRWORKS	40
6.2	open64 Phases	41
8.1	General Compiler Phases	45
8.2	Compiler Modules	46
8.3	Flow of the C2CUDA Translator	48
8.4	Code Transformation in Kernel Region	50
8.5	Compiler Style	51
8.6	Pattern Learning	52
8.7	Dependency Checker	54
8.8	Banerjee Wolfe Test Function	55
8.9	Recursion Check	56
8.10	Recursion Check Function	57
8.11	NFA Example	58
8.12	Grammar Example	60
8.13	Grammar Tree Example	61
8.14	Pattern Example	62
8.15	Corresponding Pattern Example	62
8.16	Class Diagram 1	63
8.17	Class Diagram 2	64
8.18	Support Function Example	65
9.1	Pattern Neural Design	69
9.2	Compiler Learning	70

10.1	Output of C2CUDATranslator	74
10.2	Knapsack Grid	79
10.3	Output of Converted Program	80
10.4	Output of Handwritten Program	80
11.1	Evaluation of C2CUDATranslator	82
C.1	C2CUDATranslator Project Structure	95

Abbreviations

ANSI	American National Standards Institute
ANTLR	ANother Tool for Language Recognition
AST	Abstract Syntax Tree
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DARPA	Defense Advanced Research Projects Agency
DFA	Deterministic Finite Automaton
DRAM	Dynamic Random Access Memory
FFT	Fast Fourier Transform
GFLOP	Giga FLoating-point OPeration
GPGPU	General-Purpose computing on Graphics Processing Units
GPU	Graphics Processing Unit
IEEE	Institute of Electrical and Electronics Engineers
IR	Intermediate Representation
LR	Left to Right
MIPS	Microprocessor without Interlocked Pipeline Stages
MPEG	Moving Picture Experts Group
NaN	Not a Number
NFA	Nondeterministic Finite Automaton
NSF	National Science Foundation
NVCC	NVIDIA C Compiler
openMP	Open Multi-Processing

PCCTS	Purdue Compiler Construction Tool Set
SIMD	Single Instruction Multiple Data
SUIF	Stanford University Intermediate Format

Chapter 1

Introduction

Because of the demands of game industry, Graphics Processing Units (GPUs) have evolved from application-specific units for 3D scene rendering into highly parallel and programmable multi pipelined processors that can satisfy extremely high computational requirements at low cost. The fact that the performance of graphic processing units (GPUs) is much bigger than the central processing units (CPUs) of now-a-days [1] is hardly surprising. GPUs were formerly focused on such limited field of computing graphic scenes. Within the course of time, GPUs became very powerful and the area of use dramatically grew. So, we can come together on the term General Purpose GPU (GPGPU) denoting modern graphic accelerators. The driving force of rapid rising of the performance is computer games and the entertainment industry that evolves economic pressure on the developers of GPUs to perform a vast number of floating-point calculations within the unit of time. The research in the field of GPGPU started in late 70's [7]. Today's fastest GPUs can deliver a peak performance in the order of 500 GFLOPS [5], more than four times the performance of the fastest x86 quad-core processor. This thesis introduces a source to source transformation of c programs to CUDA Architecture. It also finds out dependencies and performs optimization for peak performance gain. Automatic evolution of kernels, independent code finding, Loop unrolling, Memory coalescing and thread scheduling are main part of concerns. IR level optimization and higher level optimizations patterns finding is important issue that may be covered by this thesis. Thesis describes parallelization patterns and CUDA C extensions from C to find out transformation rules. At least one can generate template using this

transformation rules.

1.1 Objective of the Work

The objective of this research is to provide a better automation of the language translation of CUDA GPU to obtain better performance and to save manual efforts and time to learn the SDK and to prepare a program for CUDA environment.

1.2 Scope of the Work

The scope of this work is to make a translator that performs the language to language conversion for c program to CUDA C and reduces the development time.

1.3 Motivation of the Work

To identify the independent computational tasks from the given C/C++ source code.

Increase the computation power by distributing the independent tasks to all the available processing elements. For this we have to parallelize the sequential code and break them in units so kernels can be made in CUDA.

1.4 Thesis Organization

The rest of the thesis is organized as follows.

Chapter 2, *Literature survey on CUDA* , describes history of CUDA. It also describes CUDA Programming Model, Memory Architecture, and Hardware implementation.

Chapter 3, *Performance Optimization Strategies* , describes various performance optimization strategies specific to CUDA, which are used to get the maximum utilization of available resources.

chapter 4, *Explanation of the Definition*, methods for performance optimization, steps to be performed with the selected method, benefit of the software.

Chapter 5, *Automatic code conversion Algorithm*, proposed algorithm and logic for code conversion and source to source translation.

Chapter 6, *Different Platforms*, describes different platforms to develop.

Chapter 7, *Platform Comparison and reason of choice*, describes comparisons of the platforms and platform choice for thesis.

Chapter 8, *Compiler Flow*, describes the flow and architecture of the translator.

Chapter 9, *Optimization*, describes the optimizations possible in translated program and decision taking neural design to make a decision.

Chapter 10, *Outcome*, describes the inputs and outputs of the translator.

Chapter 11, *Evaluation*, describes the evaluation of the translator based upon converted programs vs benchmark programs.

Finally, in **chapter 12** concluding remarks and in **chapter 13** scope for future work is presented.

Chapter 2

Literature Survey

In November 2006, NVIDIA introduced CUDA(Compute Unified Device Architecture), a general purpose parallel computing architecture [5] - with a new parallel programming model and instruction set architecture - that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. NVIDIA GPUs with the new Tesla unified graphics and computing architecture run CUDA C programs and are widely available in laptops, PCs, workstations, and servers. The CUDA model is also applicable to other shared-memory parallel processing architectures, including multicore CPUs [8].

NVIDIA CUDA SDK has been designed for running parallel computations on the device hardware: it consist of a compiler, host and device runtime libraries and a driver API. CUDA software stack is composed of several layers: a hardware driver (CUDA Driver), an API and its runtime (CUDA Runtime), two higher-level mathematical libraries (CUDA Libraries) of common usage as shown in fig 2.1.

GPU performance is influenced by the architectural organization of the hardware platform. NVIDIA suggests that achieving the highest GPU occupancy and optimizing the use of the memory hierarchy are the two main factors behind GPU performance [2]. In fact, both of them are related since maximizing the occupancy can help to cover latency during global memory loads. Researchers represent several experiments aimed at analyzing their relative importance. Results indicate that code transformations that target efficient memory usage are the major determinant of actual performance. Overall, they ensure the

best performance even if some resources remain underutilized. Therefore, maximizing occupancy should be examined at a later stage in the compilation process, once data related issues have been properly addressed. NVIDIA compiler NVCC can optimize code but the best optimized code is one should write at assembly level. But it looks very difficult in big algorithms and projects. So to find out occupancy is important issue.

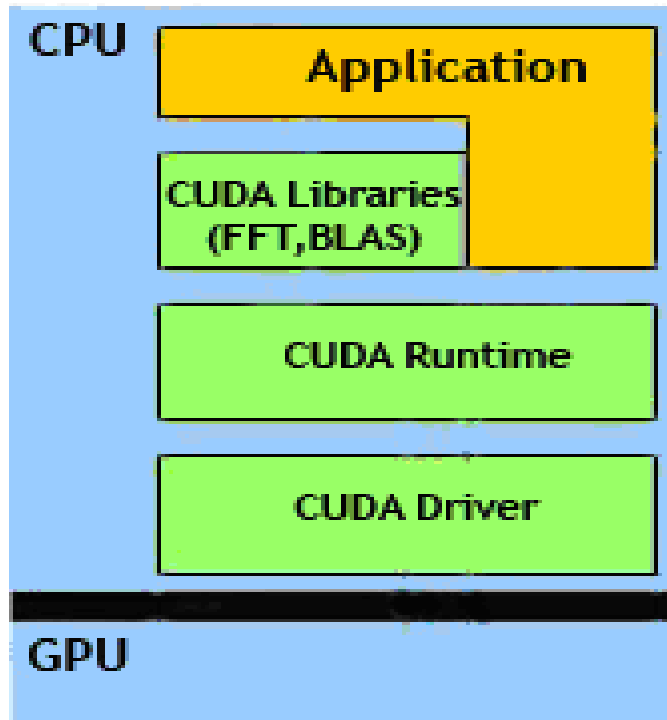


Figure 2.1: CUDA Software Stack

2.1 Design goal

- Scale to 100's of cores, 1000's of parallel threads.
- Focus on the task of parallelization of the algorithms rather than spending time on their implementation.
- Support heterogeneous computation where applications use both the CPU and GPU. Serial portions of applications are run on the CPU, and parallel portions are run on

to the GPU.

- Enable heterogeneous systems (CPU + GPU) CPU and GPU are separate devices with separate DRAMs [2].
- Generate a template based on calculated occupancy.
- Conversion of C code in a way it fits in the CUDA C template.
- Optimize the source code and measure the performance GPU & CPU of the program [4].

2.1.1 Advantages

Advantages of CUDA over the traditional approach to GPGPU computing:

- More efficient data transfers between system and video memory.
- Faster downloads and read backs to and from the GPU.
- Scattered reads - code can read from arbitrary addresses in memory.
- Shared memory - CUDA exposes a fast shared memory region (16KB in size) that can be shared amongst threads. This can be used as a user-managed cache, enabling higher bandwidth than is possible using texture lookups.
- Full support for integer and bitwise operations.
- Support for integer texture lookups.
- Programming interface of CUDA applications is based on the standard C language with extensions, which facilitates the learning curve of CUDA operations.
- Support for integer texture lookups.
- Programming interface of CUDA applications is based on the standard C language with extensions, which facilitates the learning curve of CUDA.

2.2 Limitation

- Threads should be running in groups of at least 32 for best performance, with total number of threads numbering in the thousands. Branches in the program code do not impact performance significantly, provided that each of 32 threads takes the same execution path; the SIMD execution model becomes a significant limitation for any inherently divergent task.
- Texture rendering is not supported.
- It uses a recursion-free, function-pointer-free subset of the C language, plus some simple extensions. However, a single process must run spread across multiple disjoint memory spaces, unlike other C language runtime environments.
- For double precision there are no deviations from the IEEE 754 standard. In single precision, Denormals and signalling NaNs are not supported; only two IEEE rounding modes are supported and those are specified on a per instruction basis rather than in a control word and the precision of division square root are slightly lower than single precision.

2.3 CUDA Programming Model

A CUDA program consists of one or more phases that are executed on either the host (CPU) or a device such as a GPU. The GPU is viewed as a compute device: that is a coprocessor to the CPU(host), has its own DRAM(device Memory), Runs many threads in parallel.[4] Data parallel portion of application are executed on the device as kernels which run in parallel on many threads. Difference between GPU and CPU thread [8] are:

- GPU threads are extremely lightweight and require very little creation overhead.
- GPU needs 1000s of threads for full efficiency where as multicore cpu needs only a few.

A kernel is executed as a grid of thread blocks. A thread block is a batch of thread that can cooperate with each other by efficiently sharing data through shared memory, and

synchronizing their execution for hazard free shared memory accesses. There is a limit to the number of threads per block, since all threads of a block are expected

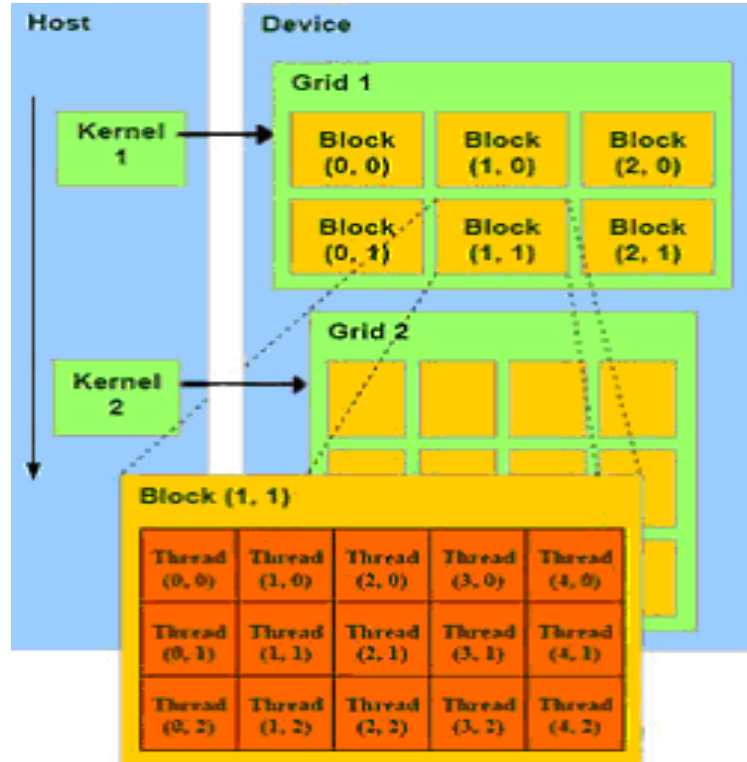


Figure 2.2: CUDA programming model

to reside on the same processor core and must share the limited memory resources of that core. Blocks are organized into a one-dimensional or two-dimensional grid of thread blocks as illustrated by Figure 2.2. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system. It must be possible to execute thread block in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores, enabling programmers to write code that scales with the number of cores. For efficient cooperation, the shared memory is expected to be a low-latency memory near each processor core (much like an L1 cache) and thread synchronization is expected to be lightweight.

2.4 CUDA Memory Model

CUDA threads may access data from multiple memory spaces during their execution as illustrated by Figure 2.3. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory. There are also two additional read-only memory spaces accessible by

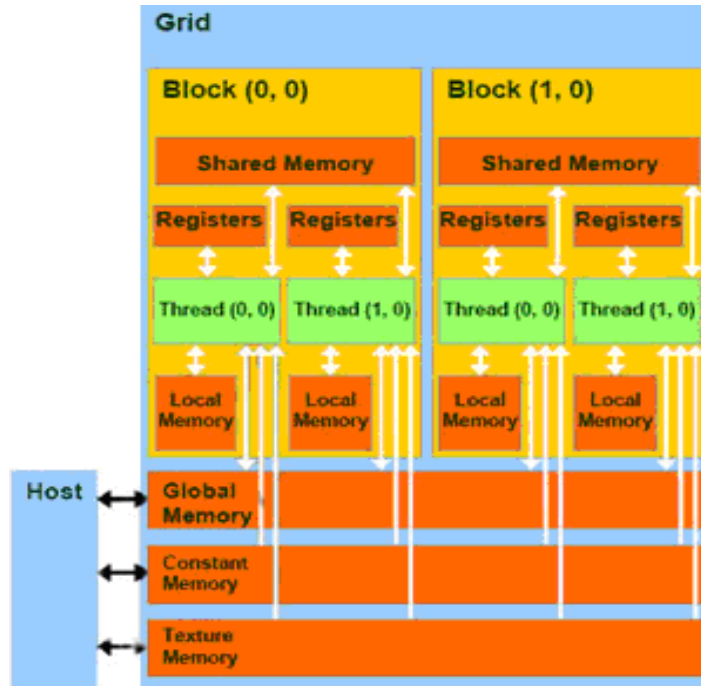


Figure 2.3: CUDA Memory model

all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages. Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats. The global, constant, and texture memory spaces are persistent across kernel launches by the same application [5].

Local Memory: is small volume of memory, which can be accessed only by one streaming processor. The local memory space resides in device memory, so local memory accesses have same high latency and low bandwidth as global memory accesses.

Local memory accesses only occur for some automatic variables. Automatic variables that the compiler is likely to place in local memory are [8]:

- Arrays for which it cannot determine that they are indexed with constant quantities,
- Large structures or arrays that would consume too much register space,
- Any variable if the kernel uses more registers than available (this is also known as register spilling).

Global Memory : the largest volume of memory available to all multiprocessors in a GPU, from 256 MB to 1.5 GB in modern solutions (and up to 4 GB in Tesla). It offers high bandwidth, over 100 GB/s for top solutions from NVIDIA, but it suffers from very high latencies (several hundred cycles). Non-catchable supports general load and store instructions, and usual pointers to memory.

Global memory resides in device memory and device memory is accessed via 32-, 64-, or 128-byte memory transactions. These memory transactions must be naturally aligned: Only the 32-, 64-, or 128-byte segments of device memory that are aligned to their size (i.e. whose first address is a multiple of their size) can be read or written by memory transactions.

Shared Memory: is 16-KB memory shared between all streaming processors in a multiprocessor. Because it is on-chip, the shared memory space is much faster than the local and global memory spaces.

To achieve high bandwidth, shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously. Any memory read or write request made of n addresses that fall in n distinct memory banks can therefore be serviced simultaneously, yielding an overall bandwidth that is n times as high as the bandwidth of a single module.

Constant Memory: is a 64 KB, read only memory for all multiprocessors. It's cached by 8 KB for each multiprocessor. The constant memory space resides in device memory. A constant memory request for a warp is first split into two requests, one for each half-warp, that are issued independently. A request is then split into as many separate requests as there are different memory addresses in the initial request, decreasing throughput by a

factor equal to the number of separate requests. The resulting requests are then serviced at the throughput of the constant cache in case of a cache hit, or at the throughput of device memory otherwise. This memory is rather slow latencies of several hundred cycles, if there are no required data in cache.

Texture Memory: space resides in device memory and is cached in texture cache, so a texture fetch costs one memory read from device memory only on a cache miss, otherwise it just costs one read from texture cache. The texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture addresses that are close together in 2D will achieve best performance. Also, it is designed for streaming fetches with a constant latency; a cache hit reduces DRAM bandwidth demand but not fetch latency.

2.5 Hardware Implementation: Memory Architecture

The local, global, constant and texture spaces are regions of device memory. Each multiprocessor has following memory space as shown in fig 2.4

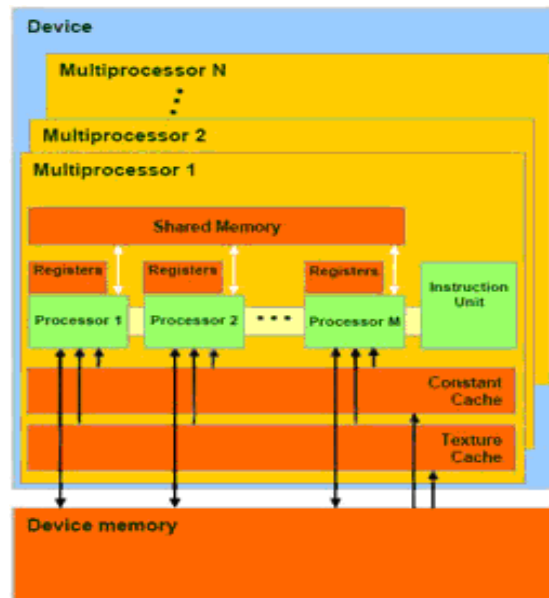


Figure 2.4: A set of SIMD processor with memory architecture

- A set of 32-bit registers per processor.

- On-chip shared memory where the shared memory space resides.
- A read-only constant cache to speed up access to the constant memory space.
- A read-only texture cache to speed up access to the texture memory space.

2.6 Hardware Implementation: Execution Model

- Each thread block of grid is split into warps that get executed by one multiprocessor, device processes only one grid at a time.
- Each thread block is executed by only one multiprocessor so that the shared memory space resides in the on chip shared memory and registers are allocated among the threads.
- A multiprocessor can execute several blocks concurrently. Shared memory and registers are allocated among the threads of all concurrent blocks. So, decreasing shared memory usage and register usage increases number of blocks that can run concurrently.

Chapter 3

Performance Optimization Strategies

Performance optimization revolves around four basic strategies:

- Maximize parallel execution to achieve maximum utilization;
- Optimize memory usage to achieve maximum memory throughput;
- Optimize Execution configuration;
- Optimize instruction usage to achieve maximum instruction throughput.

Which strategies will yield the best performance gain for a particular portion of an application depends on the performance limiters for that portion, optimizing instruction usage of a kernel that is mostly limited by memory accesses will not yield any significant performance gain. Optimization efforts should therefore be constantly directed by measuring and monitoring the performance limiters, for example using the CUDA profiler. Also, comparing the floating-point operation throughput or memory throughput - whichever makes more sense of a particular kernel to the corresponding peak theoretical throughput of the device indicates how much room for improvement there is for the kernel.

3.1 Maximize Utilization

To get the maximum utilization of the available resources, application should be parallelized in such a way that application keeps various components of the system busy most of the time.

3.1.1 Application Level

At a high level, the application should maximize parallel execution between the host, the devices, and the bus connecting the host to the devices, by using asynchronous functions calls and streams. It should assign to each processor the type of work it does best: serial workloads to the host; parallel workloads to the devices [11].

For parallel execution program is divided into threads, this threads need to share data with each other, there are two cases:

- If this threads belong to same block, they should use `syncthreads()` and share data through shared memory.
- If threads belong to different blocks, they must share data through global memory. In this case two separate kernel invocations are required, one for writing to and one for reading from global memory.

The second case adds extra overhead of kernel invocations and also increases global memory traffic. Its occurrence should therefore be minimized by mapping the algorithm to the CUDA programming model in such a way that the computations that require inter-thread communication are performed within a single thread block as much as possible.

3.1.2 Device Level

At a lower level, the application should maximize parallel execution between the multiprocessors of a device [13].

For devices of compute capability 1.x, only one kernel can execute on a device at one time,

so the kernel should be launched with at least as many thread blocks as there are multiprocessors in the device. For devices of compute capability 2.0, multiple kernels can execute concurrently on a device, so maximum utilization can also be achieved by using streams to enable enough kernels to execute concurrently.

3.1.3 Multiprocessor Level

At an even lower level, the application should maximize parallel execution between the various functional units within a multiprocessor [13].

To maximize utilization, a GPU multiprocessor relies on thread-level parallelism. Utilization is therefore directly dependent on the number of resident warps. At every instruction issue time, a warp scheduler selects a warp that is ready to execute, if any, and issues the next instruction to the active threads of the warp. The number of clock cycles it takes for a warp to be ready to execute its next instruction is called latency, and full utilization is achieved when the warp scheduler always has some instruction to issue for some warp at every clock cycle during that latency period, or in other words, when the latency of each warp is completely hidden by other warps. How many instructions are required to hide latency depends on the instruction throughput.

If all input operands are registers, latency is caused by register dependencies. In the case of a back-to-back register dependency (i.e., some input operand is written by the previous instruction), the latency is equal to the execution time of the previous instruction and the warp scheduler must schedule instructions for different warps during that time. If warp is waiting at some memory fence or synchronization point at that time warp is not ready to execute its next instruction. A synchronization point can force the multiprocessor to idle as more and more warps wait for other warps in the same block to complete execution of instructions prior to the synchronization point. Having multiple resident blocks per multiprocessor can help reduce idling in this case, as warps from different blocks do not need to wait for each other at synchronization points.

The number of blocks and warps residing on each multiprocessor for a given kernel call depends on the execution configuration of the call, the memory resources of the multiprocessor, and the resource requirements of the kernel. To assist programmers in choosing

thread block size based on register and shared memory requirements, the CUDA Software Development Kit provides a spreadsheet, called the CUDA Occupancy Calculator, where occupancy is defined as the ratio of the number of resident warps to the maximum number of resident warps [13].

The performance of an application also depends on the kernel code. The number of threads per block should be chosen as a multiple of the warp size to avoid wasting of computing resources with under-populated warps as much as possible.

3.2 Maximize Memory Throughput

Memory optimizations are the most important area for performance. The goal is to maximize the use of the hardware by maximizing bandwidth. Bandwidth is the best served by using as much fast memory and as little slow-access memory as possible [14]. This section discusses best way to set up data items to use the memory effectively.

3.2.1 Data Transfer between Host and Device

There are various ways to transfer data between host and device, which method provides best performance depend on the type of application and type of data, the application has to process.

Minimize data transfer between the host and the device.

Applications should be structure in such a way that minimizes data transfer between the host and the device. One way to accomplish this is to move more code from the host to the device, even if that means running kernels with low parallelism computations. Intermediate data structures may be created in device memory, operated on by the device, and destroyed without ever being mapped by the host or copied to host memory.

Group transfers

Also, because of the overhead associated with each transfer, batching many small transfers into a single large transfer always performs better than making each transfer separately.

Page-Locked Data Transfers

On systems with a front-side bus, higher performance for data transfers between host and device is achieved by using page-locked host. In addition, when using mapped page-locked memory, there is no need to allocate any device memory and explicitly copy data between device and host memory. Data transfers are implicitly performed each time the kernel accesses the mapped memory. For maximum performance, these memory accesses must be coalesced as with accesses to global memory. Assuming that they are and that the mapped memory is read or written only once, using mapped page-locked memory instead of explicit copies between device and host memory can be a win for performance.

On systems where device memory and host memory are physically the same, any copy between host and device memory is not required at that time mapped page-locked memory should be used. Applications can check whether a device is integrated or not by calling `cudaGetDeviceProperties()` and checking the integrated property or checking the `CU_DEVICE_ATTRIBUTE_INTEGRATED` attribute using `cuDeviceGetAttribute()`. `cudaMallocHost()` allows allocation of page-locked ("pinned") host memory. Page-locked memory enable highest `cudaMemcpy` performance up to 3.2 GB/s on PCI-e x16 Gen1, and 5.2 GB/s on PCI-e X16 Gen2.

Allocating too much page-locked memory can reduce overall system performance, so we need to test our system and application to find their limits.

Asynchronous Transfers and Overlapping Transfers with Computation

We can transfer data between host and device using `cudaMemcpy()`, which is the blocking transfers, whereas `cudaMemcpyAsync()` provides non-blocking transfers in which control is returned immediately to the host thread. The asynchronous transfer requires pinned host memory and `streamID` (A stream is a sequence of operations that are performed in order on the device).

We can overlap data transfers and computation using asynchronous transfers. There are two ways to do this first, on all CUDA-enabled devices; it is possible to overlap host computation with asynchronous data transfers and with device computations. For example, Listing 3.1 demonstrates how host computation in the routine `cpu Function()` is performed while data is transferred to the device and a kernel using the device is executed.

The last argument to the `cudaMemcpyAsync()` function is the stream ID, which in this case uses the default stream, stream 0. The kernel also uses the default stream, and it will not

begin execution until the memory copy completes; therefore, no explicit synchronization is needed. Because the memory copy and the kernel both return control to the host immediately, the host function `cpuFunction()` overlaps their execution. In Listing 4.1, the memory copy and kernel execution occur sequentially.

```
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, 0);
kernel<<<grid, block>>>(a_d);
cpuFunction();
```

Second, on devices that are capable of concurrent copy and execute, it is possible to overlap kernel execution on the device with data transfers between the host and the device. Whether a device has this capability is indicated by the `deviceOverlap` field of a `cudaDeviceProp`. On devices that have this capability, the overlap once again requires pinned host memory, and, in addition, the data transfer and kernel must use different, non-default streams (streams with non-zero stream IDs). Non-default streams are required for this overlap because memory copy, memory set functions, and kernel calls that use the default stream begin only after all preceding calls on the device (in any stream) have completed, and no operation on the device (in any stream) commences until they are finished. Listing 3.2 illustrates the basic technique.

```
cudaStreamCreate(&stream1); cudaStreamCreate(&stream2);
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, stream1);
kernel<<<grid, block, 0, stream2>>>(otherData_d);
```

Listing 3.2 Concurrent copy and execute

In above listing, two streams are created and used in the data transfer and kernel executions as specified in the last arguments of the `cudaMemcpyAsync` call and the kernel's execution configuration. This technique could be used when the data dependency is such that the data can be broken into chunks and transferred in multiple stages, launching multiple kernels to operate on each chunk as it arrives.

Zero Copy

Zero copy is a feature that was added in version 2.2 of the CUDA Toolkit [14]. It allows GPU threads to directly access host memory. For this purpose, it requires mapped pinned (non-pageable) memory. On integrated GPUs, mapped pinned memory is always gives best

performance because it avoids redundant copies as integrated GPU and CPU memory are physically the same. Zero copy can be used in place of streams because kernel-originated data transfers automatically overlap kernel execution without the overhead of setting up and determining the optimal number of streams. The host code in Listing 3.3 shows how zero copy is typically set up.

```
float *a_h, *a_map; ....
cudaGetDeviceProperties(&prop, 0);
if(!prop.canMapHostMemory) exit(0); cudaSetDeviceFlags(cudaDeviceMapHost);
cudaHostAlloc((void*)&a_h, nBytes, cudaHostAllocMapped);
cudaHostGetDevicePointer((void *)&a_map, (void *)a_h, 0);
kernel<<<gridSize, blockSize>>>(a_map);
```

Listing 3.3 Zero-copy host code

In above code, `cudaGetDeviceProperties` is used to check that device supports mapping host memory to the device’s address space. `cudaSetDeviceFlags()` are used to map page-locked memory. `cudaHostAlloc()` is used to allocate page-locked mapped memory, and `cudaHostGetDevicePointer()` is used to get the pointer to the mapped device address space.

3.2.2 Optimize Memory Access Patterns

Performance of the system also depends on bandwidth. Bandwidth is one of the most important factor for performance. It goes on to calculate theoretical bandwidth which is in the order of hundreds of gigabytes per second. Where as Effective bandwidth can vary by an order of magnitude depending on access pattern, so need to optimize access patterns to get:

Coalesced global memory accesses

The simultaneous global memory accesses by each thread of a half-warp (16 threads on G80) during the execution of a single read or write instruction will be coalesced into a single access if:

- The size of the memory element accessed by each thread is either 4, 8, or 16 bytes.
- The elements form a contiguous block of memory.
- The Nth element is accessed by the Nth thread in the half-warp.
- The address of the first element is aligned to 16 times the element's size.
- Coalescing happens even if some threads do not access memory (divergent warp).

Figures 3.1 - 3.5 shows the different coalescing and Non-coalescing global memory access patterns.

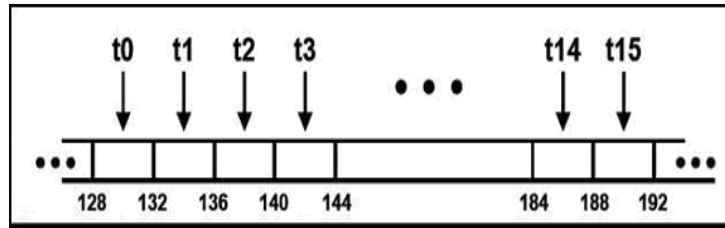


Figure 3.1: A coalescing memory access

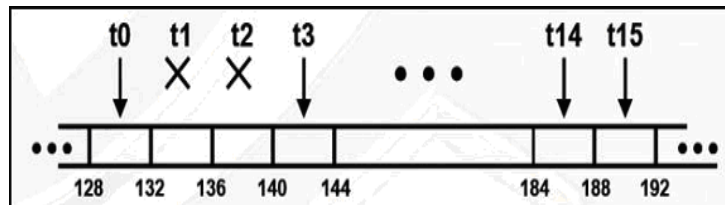


Figure 3.2: coalescing memory access: divergent warp

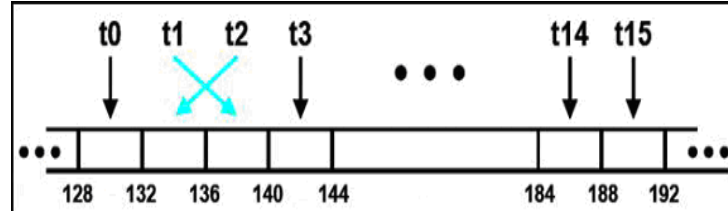


Figure 3.3: Non- Sequential memory access

We have to follow following guideline to avoid non-coalescing memory access.

- For irregular read patterns, texture fetches can be a better alternative to global memory reads.
- If all threads read the same location, use constant memory.
- For sequential access patterns, but a structure of size 4, 8, or 16 bytes:

Use a Structure of Array in stand of Array of Structures as shown in fig. 3.6.

Force structure alignment using $\text{align}(X)$, where $X = 4, 8, \text{ or } 16$.

Use shared memory to achieve coalescing. (divergent warp).

Maximize the use of shared memory Because it is on-chip, shared memory is much faster than local and global memory. In fact, uncached shared memory latency is roughly 100x lower than global memory latency-provided there are no bank conflicts between the threads.

3.3 Execution configuration optimization

To get the maximum performance it is important to design the application to use threads and blocks in a way that get maximum utilization of available resources. A key concept in

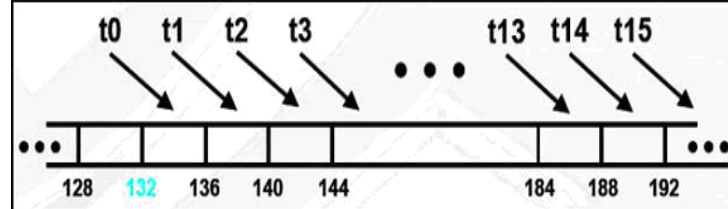


Figure 3.4: A coalescing memory access

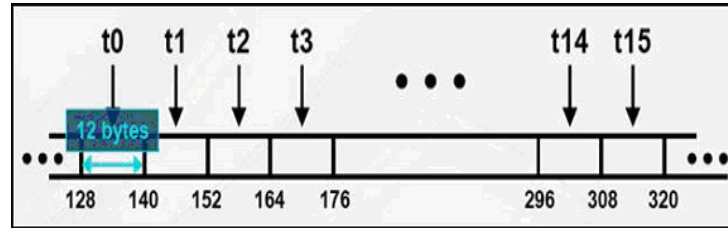


Figure 3.5: A coalescing memory access

this effort is occupancy, which is explained in the following sections.

3.3.1 Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use [11]. Higher occupancy does not always equate to higher performance - there is a point above which additional occupancy does not improve performance. However, low occupancy always interferes with the ability to hide memory latency, resulting in performance degradation.

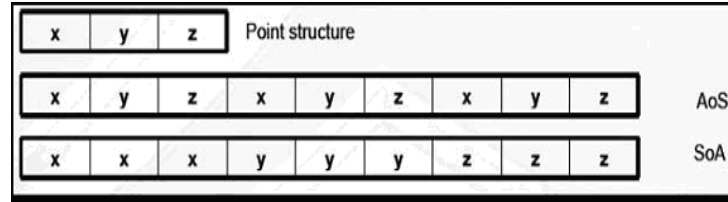


Figure 3.6: A coalescing memory access

3.4 Maximize Instruction Throughput

If programmer knows, how instructions are executed then it is possible to apply low level optimizations that can be useful. It is good practices to apply lower level optimization after all higher-level optimization have been completed. To maximize instruction throughput the application should [13]:

- Minimize the use of arithmetic instructions with low throughput; this includes trading precision for speed when it does not affect the end result, such as using intrinsic instead of regular functions, single-precision instead of double precision, or flushing the normalized numbers to zero.
- Minimize divergent warps caused by control flow instructions.
- Reduce the number of instructions, for example, by optimizing out synchronization points whenever possible or by using restricted pointers.

Here, throughput is given in number of operations per clock cycle per multiprocessor. For a warp size of 32, one instruction results in 32 operations. Therefore, if T is the number of operations per clock cycle, the instruction throughput is one instruction every $32/T$ clock cycles. All throughputs are for one multiprocessor. They must be multiplied by the number of multiprocessors in the device to get throughput for the whole device.

Chapter 4

Problem Definition

To optimize process we may like to make faster algorithm. But compiler on GPU NVCC may fail in converting proper optimized assembly code. To solve this one may write whole algorithm in assembly language. But it looks difficult for huge projects. We may prepare the tool that converts the C code in CUDA C but with proper optimization.

For .cu files we specify a header file, containing host functions in it. Then, include that header file in other .cu or .cpp files. The linker will do the rest. It is nothing different than having multiple plain .cpp files in project. It may possible to star with some example programs in c and to convert them in CUDA C and finding transformation rules.

4.1 Part 1: C level Parallelization

4.1.1 Loop interchange transformation

Before Parallelization:

```
for ( i = 0 ; i < 200 ; i++ )  
{  
  for ( j = 0 ; j <= 100 ; j++ )  
  {  
    A[i][j] = A[i-1][j] + B[i][j];  
  }  
}
```

After Parallelization:

```
for ( j = 0 ; j <= 100 ; j++ )
{
  for ( i =0 ; i < 200 ; i++ )
  {
    A[i][j] = A[i-1][j] + B[i][j];
  }
}
```

4.1.2 Loop embedding transformation

Before Parallelization:

```
for (i=0;i<100;i++)
{
  do_something();
}
....
void do_something()
{
  procedure body
}
```

After Parallelization:

```
....
do_something2();
....
void do_something2()
{
  int i;
  for(i=0;i<100;i++)
  {
    procedure body
```



```

}
}

```

Both above transformations differ quite substantially, and therefore, using standard approaches, two separate programs would have to be written for each of them. This can be avoided if we properly organize the process of applying a transformation.

The translation is divided into 3 distinct stages:

- Code selection stage: In this stage the engine searches for code that has a strictly specified structure (that matches a specified pattern). Each fragment that matches this pattern is a candidate for the transformation.
- Conditions checking stage: Transformations can pose other (non-structural) restrictions on a matched code fragment. These restrictions include, but are not limited to, conditions on data dependencies and properties of index variables.
- Transformation stage: Code fragments that matched the specified structure and additional conditions are replaced by new code, which has the same semantics as the original code.

4.1.3 Pattern Structure

PATTERN

```
{
```

description of the code selection stage

```
}
```

CONDITIONS

```
{
```

additional constraints

```
}
```

RESULT

```
{
```

description of the new code

```
}

```

Pattern block for the loop interchange transformation

```
PATTERN

```

```
{
VAR x, y;
for (x=EXPR(1);BOUND(1,x);STEP_EXPR(2,x))
{
for (y=EXPR(1);BOUND(2,y);STEP_EXPR(3,y))
{
STMTLIST(1);
}
}
}
```

Pattern block for the loop embedding transformation

```
PATTERN

```

```
{
VAR x;
for (x=EXPR(1);BOUND(1,x);STEP_EXPR(2,x))
{
PROCCALL(1, p_pat);
}
}
void p_pat()
{
STMTLIST(1);
}
```

4.1.4 Condition Block

A conditions block contains additional conditions that are imposed on code that was elected in the previous stage. These conditions can be divided into 3 categories:

- **Type properties:** This category includes requirements such as "is this expression a constant?", "does this statement contain a procedure call?" or 'does this expression have side effects?'.
- **Structural properties:** In case of some structural properties it is not desirable to have to specify them in the pattern block. Most importantly, if we had to specify the existence or absence of control transfer statements (break, continue and return statements) in the pattern block, we would severely restrict the generality of the descriptions of transformations. Therefore, extra conditions that test for these structural properties are necessary.
- **Data dependencies::** Data dependencies are vital in testing for the legality of code transformations. Depending on the type of transformation we must be able to test or different following properties of data dependencies.

Condition block for the loop interchange transformation

CONDITIONS

```
{
stmtlist_has_no_unsafe_jumps(1);
not(dep(* direction=(<,>)
between stmtlist 1 and stmtlist 10));
}
```

4.1.5 Result Block

The result block uses the same part of the transformation language as used in the pattern block. Also a number of the result block specific elements are allowed. These include for example support for creation of new variables in the transformed code. Also strict

control over the data type of these newly created variables is possible. The examples below demonstrate the transformation block specification.

Result block for the loop interchange transformation

```

RESULT
{
  VAR x, y;
  for (y=EXPR(1);BOUND(2,y);STEP_EXPR(3,y))
  {
    for (x=EXPR(1);BOUND(1,x);STEP_EXPR(2,x))
    {
      STMTLIST(1);
    }
  }
}

```

Result block for the loop embedding transformation

```

RESULT
{
  PROCCALL(1, p_transformation);
}
void p_transformation()
{
  VAR x;
  for (x=EXPR(1);BOUND(1,x);STEP_EXPR(2,x))
  {
    STMTLIST(1);
  }
}

```

4.2 Part 2: C to CUDA Transformation

4.2.1 Examples

C Program structure

```
#include <stdio.h>

int add (int,int); /* function prototype for add */

void main() {
    printf("%d ",add(3));
}

int add(int i, int j)
{
    return i+j;
}
```

CUDA C Program structure #include <stdio.h>

```
#include <cuda.h>

#define N 512

__global__ void add( int*a, int*b, int*c )

int main( void )
{
    int *a, *b, *c; //host copies of a, b, c
    int *dev_a, *dev_b, *dev_c; //device copies of a, b, c
    int size = N * sizeof( int); //we need space for 512 integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( size );

    random_ints( a, N );
```

```

random_ints( b, N );
// copy inputs to device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice);
// launch add() kernel with N
add<<<N,N >>>( dev_a, dev_b, dev_c);
// copy device result back to host copy of c
cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost);
free( a );
free( b );
free( c );
cudaFree( dev_a);
cudaFree( dev_b);
cudaFree( dev_c);
return 0;
}
__global__ void add( int*a, int*b, int*c )
{
int index = threadIdx.x + blockIdx.x * blockDim.x;
c[index] = a[index] + b[index];
}

```

This is sample program for adding two numbers.

4.2.2 Patterns

Program Structure

C Program

```
#include"stdio.h"
```

```
.... Function declarations .
```

```

void main()
{
int i;// var declarations
.... Function() ....
}
functions that run on CPU

```

CUDA C Program

```

#include"stdio.h"
#include"cuda.h"
.... __host__Function declarations
__global__ kernel Function declarations
....
int void main()
{
int i;// var declarations ....
Function() ....
Kernel function<<>>() ....
return 0;
}
Functions that run on CPU
Kernel functions that run on GPU

```

Declaration

a.> Variable Declaration C Program

```
int a, b;
```

CUDA C Program

```

int *a, *b, *c; //host copies of a, b, c
int *dev_a, *dev_b,*dev_c;//device copies of a, b, c
cudaMalloc( (void**)&dev_a, size );
cudaMalloc( (void**)&dev_b, size );

```

```
cudaMalloc( (void**)&dev_c, size );
```

```
a = (int*)malloc( size );
```

```
b = (int*)malloc( size );
```

```
c = (int*)malloc( size );
```

b.> Function Declaration

C Program

```
int add (int,int); /* function prototype for add */
```

CUDA C Program

```
__global__ void add( int *a, int *b, int *c ) ;
```

texture declaration

```
texture<Type, Dim, ReadMode> texRef;
```

- Type specifies the type of data that is returned when fetching the texture; Type is restricted to the basic integer and single-precision floating-point types and any of the 1-, 2-, and 4-component vector types;
- Dim specifies the dimensionality of the texture reference and is equal to 1, 2, or 3; Dim is an optional argument which defaults to 1;
- ReadMode is equal to `cudaReadModeNormalizedFloat` or `cudaReadModeElementType`; if it is `cudaReadModeNormalizedFloat` and Type is a 16-bit or 8-bit integer type, the value is actually returned as floating-point type and the full range of the integer type is mapped to $[0.0, 1.0]$
- For unsigned integer type and $[-1.0, 1.0]$ for signed integer type; for example, an unsigned 8-bit texture element with the value `0xff` reads as 1; if it is `cudaReadModeElementType`, no conversion is performed; ReadMode is an optional argument which defaults to `cudaReadModeElementType`.

4.2.3 Use of `#pragma unroll`

The `#pragma unroll` directive however can be used to control unrolling of any given loop. It must be placed immediately before the loop and only applies to that loop. It is optionally followed by a number that specifies how many times the loop must be unrolled.

```
#pragma unroll 5 for (int i = 0; i < n; ++i)
```

the loop will be unrolled 5 times. The compiler will also insert code to ensure correctness (in the example above, to ensure that there will only be n iterations if n is less than 5, for example). It is up to the programmer to make sure that the specified unroll number gives the best performance.

`#pragma unroll 1` will prevent the compiler from ever unrolling a loop. If no number is specified after `#pragma unroll`, the loop is completely unrolled if its trip count is constant, otherwise it is not unrolled at all.

4.3 Issues

- a. One issue was that the CUDA compiler does not like long expressions. In initial tests, the evolved programs were written as a single expression. However, when the length of the expression was increased the compilation time increased dramatically. This is presumably because they are difficult to optimize.
- b. Another issue is that functions with many input variables will cause the compilation to fail, with the compiler complaining that it had been unable to allocated sufficient registers. In initial development, we had passed all the inputs in the training set to each individual - regardless of if the expression used them. This worked well for small numbers of inputs, however the training set that was used to test the system contains 41 columns. The solution to this problem was to pass the function only the inputs that it used. However, this requires each function to be executed with a different parameter configuration. Conveniently, the CUDA.Net interface does allow this, as the function call can be generated dynamically at run time. The other issue here is that all/many inputs may be needed to solve a problem. It is hoped that this is a

compiler bug and that it will be resolved in future updates to CUDA.

c. Restrictions

- (1) `__device__` and `__global__` functions do not support recursion.
- (2) `__device__` and `__global__` functions cannot declare static variables inside their body.
- (3) `__device__` and `__global__` functions cannot have a variable number of arguments.
- (4) `__device__` functions cannot have their address taken; function pointers to `__global__` functions, on the other hand, are supported.
- (5) The `__global__` and `__host__` qualifiers cannot be used together.
- (6) `__global__` functions must have void return type.
- (7) Any call to a `__global__` function must specify its execution configuration.
- (8) `__global__` function is asynchronous, meaning it returns before the device has completed its execution.
- (9) A call to a `__global__` function parameters are currently passed via shared memory to the device and limited to 256 bytes.

4.4 Use of Patterns

- Patterns help us describe expert solutions to parallel programming

- They give us a language to describe the architecture of parallel software.
- They provide a roadmap to the frameworks we need to support general purpose programmers.
- And they give us a way to systematically map programming languages onto of parallel algorithms thereby comparing their range of suitability.

Chapter 5

Automatic code conversion Algorithm

This algorithm is developed to convert a normal program written in C programming language which runs on a single CPU into a complete CUDA C program which can run properly on CUDA architecture and can get performance benefits of the architecture.

5.1 The Algorithm

The proposed algorithm is as bellow.

Algorithm 5.1 The Proposed Algorithm

- 1 Read C file of the input
- 2 Find out all the variables and kernel variables
- 3 Find out the global variables and included files
- 4 Create Symbol Table
- 5 Find out declared functions
- 6 If it is a main function Than
- 7 Write it as main function
- 8 Else
- 9 Write it as it is
- 10 Find out functional dependence
- 11 If there is a inter functional dependence Than
- 12 Write the function definition as device function

- 13 Else
- 14 write the function as it is
- 15 Read main.cu file line by line
- 16 Analyze code and calculate BlockGrid and BlockDim
- 17 Write appropriate CUDAMalloc and CUDAMemcpy code at respective place
- 18 Find out the function calls in the definition and insert them in function list
- 19 Find pragma kernel regions and convert them into kernel calls
- 20 analysis with dependency checker
- 21 Based on dependency flags call transformation routines
- 22 Call appropriate kernel threads instead of function calls
- 23 Include the header file to all the main.cu file.
- 24 Compile the main.cu file using nvcc compiler
- 25 Run the main file

Chapter 6

Different Compiler Platforms

Now, the translation process contains lots of complex processes like token generation, scanning, parsing, compilation, code generation, code modification, code analysis, dependency analysis, code optimization etc.

So it was impossible to make a compiler in c itself for c and do it all manually in one year by just only one student! So I better started for looking for compiler frameworks or compiler development environment that provides me some basic data structures like DFAs, NFAs or link list trees and interfaces to access those huge data structures to make my translator.

I found some open source compiler platforms listed below:

6.1 Introduction to ANTLR

ANTLR, ANother Tool for Language Recognition, is a language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a variety of target languages. [21]

ANTLR has a sophisticated grammar development environment called ANTLRWorks.

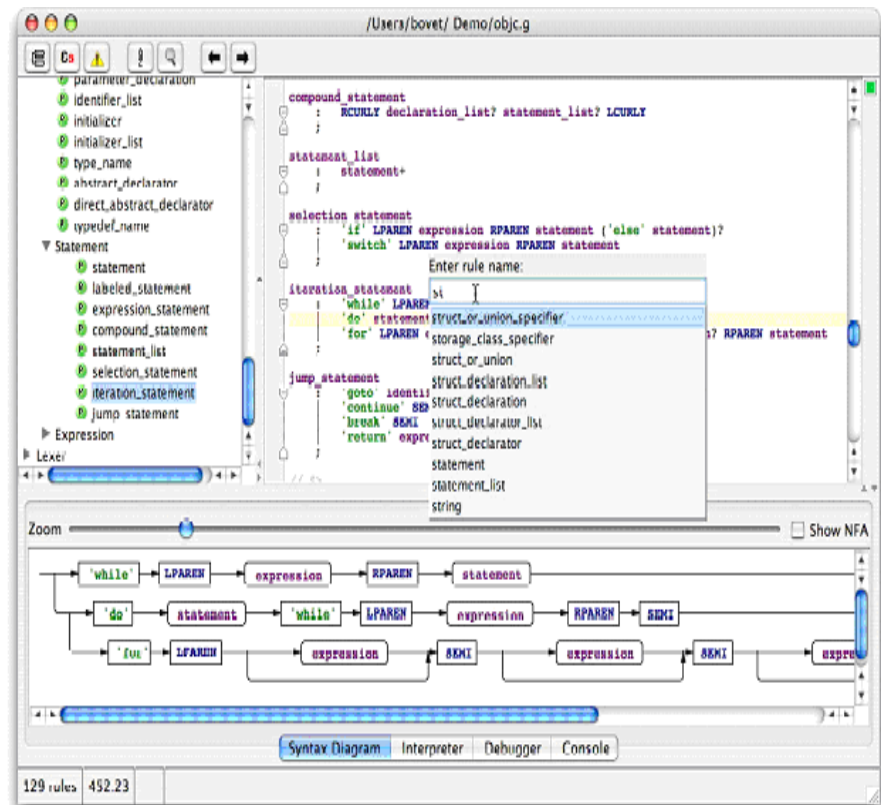


Figure 6.1: ANTLRWORKS

6.2 Introduction to Open64

Open64 has been well-recognized as an industrial-strength production compiler. It is the final result of research contributions from a number of compiler groups around the world. Formerly known as Pro64, Open64 was initially created by SGI from SGI's MIPSPro compiler. [24]

Open64 includes advanced interprocedural optimizations, loop nest optimizations, global scalar optimizations, and code generation with advanced global register allocation and software pipelining. [24]

A considerable amount of work has been performed with the Open64 compiler infrastructure over the last few years. There were lots of phases and passes those could be enabled or disabled in compiler when I installed it. Open64 is WHIRL based and it converts WHIRL file to C. I tried to convert knapsack using open64. It created whirl file to me. But to work with it team and time is needed.

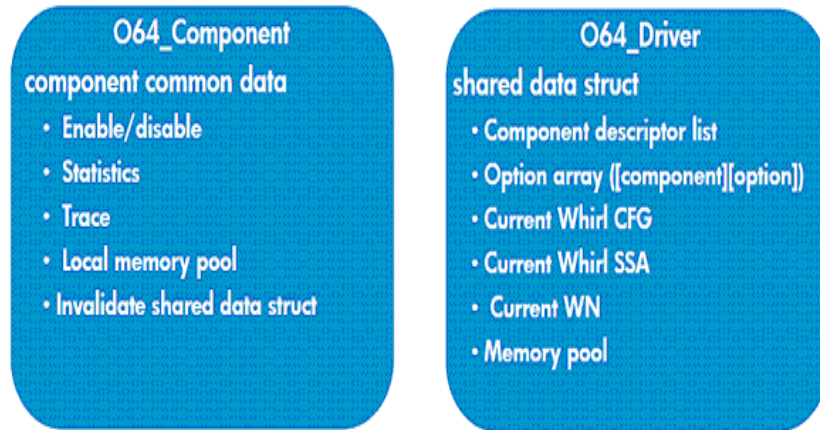


Figure 6.2: open64 Phases

6.3 Introduction to SUIF

The SUIF (Stanford University Intermediate Format) compiler, developed by the Stanford Compiler Group, is a free infrastructure designed to support collaborative research in optimizing and parallelizing compilers. It is a part of the national compiler infrastructure project funded by DARPA and NSF. [25]

The SUIF 2 compiler infrastructure project is co-funded by DARPA and NSF. It is a new version of the SUIF compiler system, a free infrastructure designed to support collaborative research in optimizing and parallelizing compilers. It is currently in the beta test stage of development. Because of a change in the file format, you will need to rebuild any SUIF2 intermediate files before running a SUIF 2.2 pass on them. [25]

But SUIF does not provide loop transformation and works with only affine partitions.

Chapter 7

Platform Comparison and reason of choice

My choice to work on the platform was ANTLR because the alternatives did not fit.

- Predecessor (Polaris) only works on Fortran77
- SUIF is for C; last major update in 2001 so fails in handling pragmas
- Open64's 5 IRs are more complex than I needed

So the best way was to write my own but not possible without support of LEX and YACC on LINUX. I found some related work on code translation that was on open64 and was a big PHD Project. So I tried to make a prototype compiler using simple grammar. And also I found that students and some of the researchers avoid use of linux because compiler becomes too complex that it becomes difficult to manage by a single person. So I jumped to ANTLR.

7.1 What exactly does ANTLR 3 do?

ANTLR reads a language description file called a grammar and generates a number of source code files and other auxiliary files. Most uses of ANTLR generates at least one (and quite often both) of these tools:

- **Lexer** : This reads an input character or byte stream (i.e. characters, binary data, etc.), divides it into tokens using patterns you specify, and generates a token stream as output. It can also flag some tokens such as whitespace and comments as hidden using a protocol that ANTLR parsers automatically understand and respect.
- **Parser** : This reads a token stream (normally generated by a lexer), and matches phrases in your language via the rules (patterns) you specify, and typically performs some semantic action for each phrase (or sub-phrase) matched. Each match could invoke a custom action, write some text via String Template, or generate an Abstract Syntax Tree for additional processing.

ANTLR's Abstract Syntax Tree (AST) processing is especially powerful. If you also specify a tree grammar, ANTLR will generate a Tree Parser for you that can contain custom actions or String Template output statements.

Most language tools will:

- a. Use a Lexer and Parser in series to check the word-level and phrase-level structure of the input and if no fatal errors are encountered, create an intermediate tree representation such as an Abstract Syntax Tree (AST),
- b. Optionally modify (i.e transform or rewrite) the intermediate tree representation (e.g. to perform optimizations) using one or more Tree Parsers, and
- c. Produce the final output using a Tree Parser to process the final tree representation. This might be to generate source code or other textual representation from the tree (perhaps using String Template) or, performing some other custom actions driven by the final tree representation.

Simpler language tools may omit the intermediate tree and build the actions or output stage directly into the parser.

7.1.1 ANTLR 3

ANTLR 3 is the latest version of a language processing toolkit that was originally released as PCCTS in the mid-1990s. As was the case then, this release of the ANTLR toolkit advances the state of the art with its new LL* parsing engine. ANTLR provides a framework for the generation of recognizers, compilers, and translators from grammatical descriptions. ANTLR grammatical descriptions can optionally include action code written in what is termed the target language (i.e. the implementation language of the source code artifacts generated by ANTLR).

When it was released, PCCTS supported C as its only target language, but through consulting with NeXT Computer, PCCTS gained C++ support after 1994. PCCTS's immediate successor was ANTLR 2 and it supported Java, C# and Python as target languages in addition to C++.

7.1.2 Target languages

ANTLR 3 already supports Java, C#, Objective C, C, Python and Ruby as target languages. Support for additional target languages including C++, Perl6 and Oberon (yes, Oberon) is either expected or already in progress. This is all due in part to the fact that it is much easier to add support for a target language (or customize the code generated by an existing target) in ANTLR 3.

7.1.3 Why should I use ANTLR 3?

Because it can save my time and resources by automating significant portions of the effort involved in building language processing tools. It is well established that generative tools such as compiler. Compilers have a major, positive impact on developer productivity. In addition, many of ANTLR v3's new features including an improved analysis engine, its significantly enhanced parsing strength via LL* parsing with arbitrary lookahead, its vastly improved tree construction rewrite rules and the availability of the simply fantastic. AntlrWorks IDE offers productivity benefits over other comparable generative language processing toolkits. [21]

Chapter 8

Compiler

When we want to make a translator we must have to think about each and every thing very carefully. Analysis phase of the project consists whole language recognition analysis and a grammar for whole language needs to be considered. On ANTLR website the grammar for C is given but it is LR and lots of recursion is called. So I made my own grammar.

For translation we need to tell the compiler that "at this rule we need to do something". This means for each rule that parses the input program there must be translation rule. In my C2CUDATranslator there is similar like this. At each rule of parser translator works in parallel with it.

General Compiler phases include:

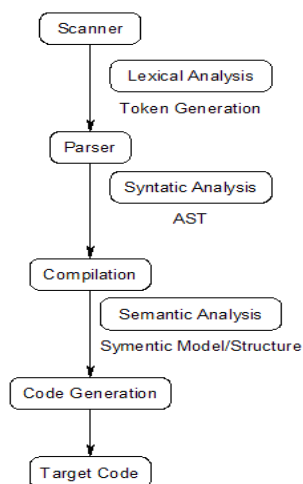


Figure 8.1: General Compiler Phases

8.1 Compiler Modules

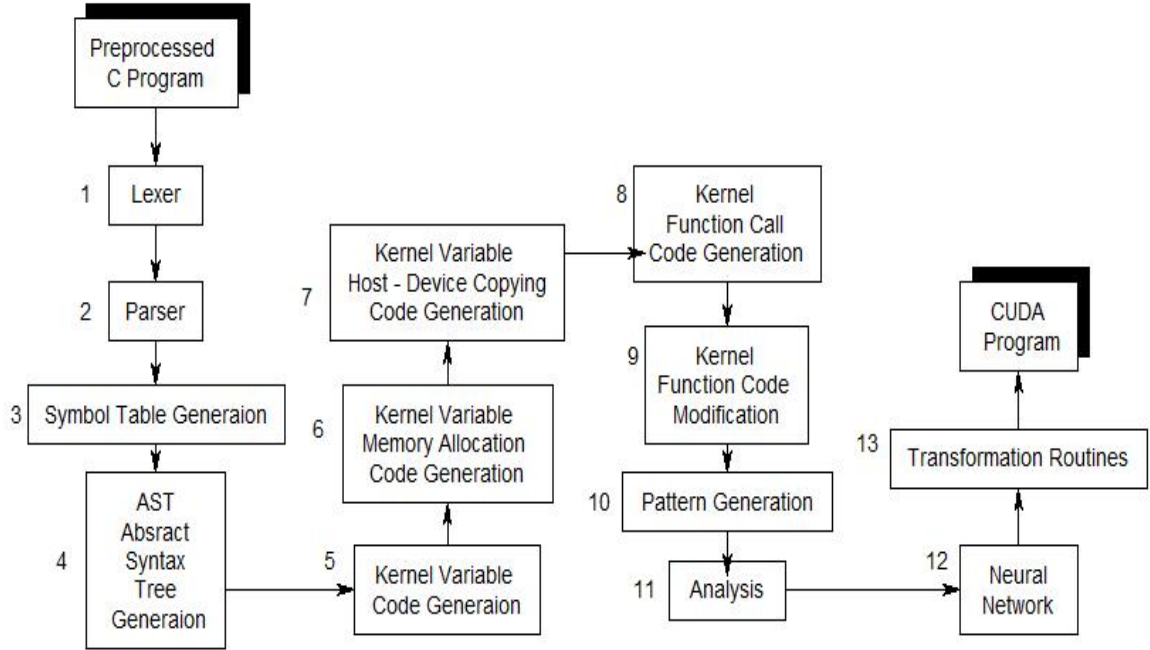


Figure 8.2: Compiler Modules

When we start any project, we try to divide it in small projects parts called modules. It is a process to divide complexity, so that one can work at lower complexity because main big task is divided in small subtasks. When a developer sits to convert a C program to CUDA program, he can read C file and can understand file. But how a computer can understand a C language? Languages are made by grammars. Grammars form language. So I need to write a grammar to make a computer understand a C file input. Modules started with Lexer and Parser in ANTLR and translator code begin in ANTLR. The figure shows how the project goes on module by module.

Language is wide domain. There are infinite members in a this language domain. A developer can write anything. By the time, more and more algorithms will be introduced and developed. So the conversion process will become inhomogeneous and according to their codes and data, the decision for parallelization and optimization will change. But this change is inhomogeneous by nature. Developer can decide this decision but how a computer can decide? So the neural network comes in later modules.

8.2 C2CUDataTranslator Phases

- a. C level Optimization
- b. Preprocessing (Proper Spacing and Bracketing)
- c. Parallelization
 - (1) Dependency Analysis
 - i. Benenergy Wolf Test
 - ii. External Memory Reference (Range Check)
 - iii. Recursion Check
 - (2) Code Transformation in kernel region (Loop Splitting, Shared Memory etc.)
- d. Translation
- e. Optimization
 - (1) Reduction in Host to Device Communication
 - (2) Code Level Optimization
 - (3) Memory Allocation changes

8.3 C2CUDataTranslator Flow

The flow of the translator shows the overall functionality and proper compiler structure. The input is C file which is pre-processed input to translator. The input is given to C Parser which is generated using ANTLR grammar and contains two files lexer and parser. Lexer generates tokens and using IToken interface the parser rules are parsed and the code is checked using the parser. The symbol table is generated. The preprocessor outlines the kernel by pragma pack. Before starting the kernel region the line with "#pragma kernel_start" comes. So the compiler can know that the next statements are of kernel region which will be ported on the GPU. The kernel is finished with the statement "#pragma kernel_end". The translator uses symbol table and converts the kernel region with kernel function.

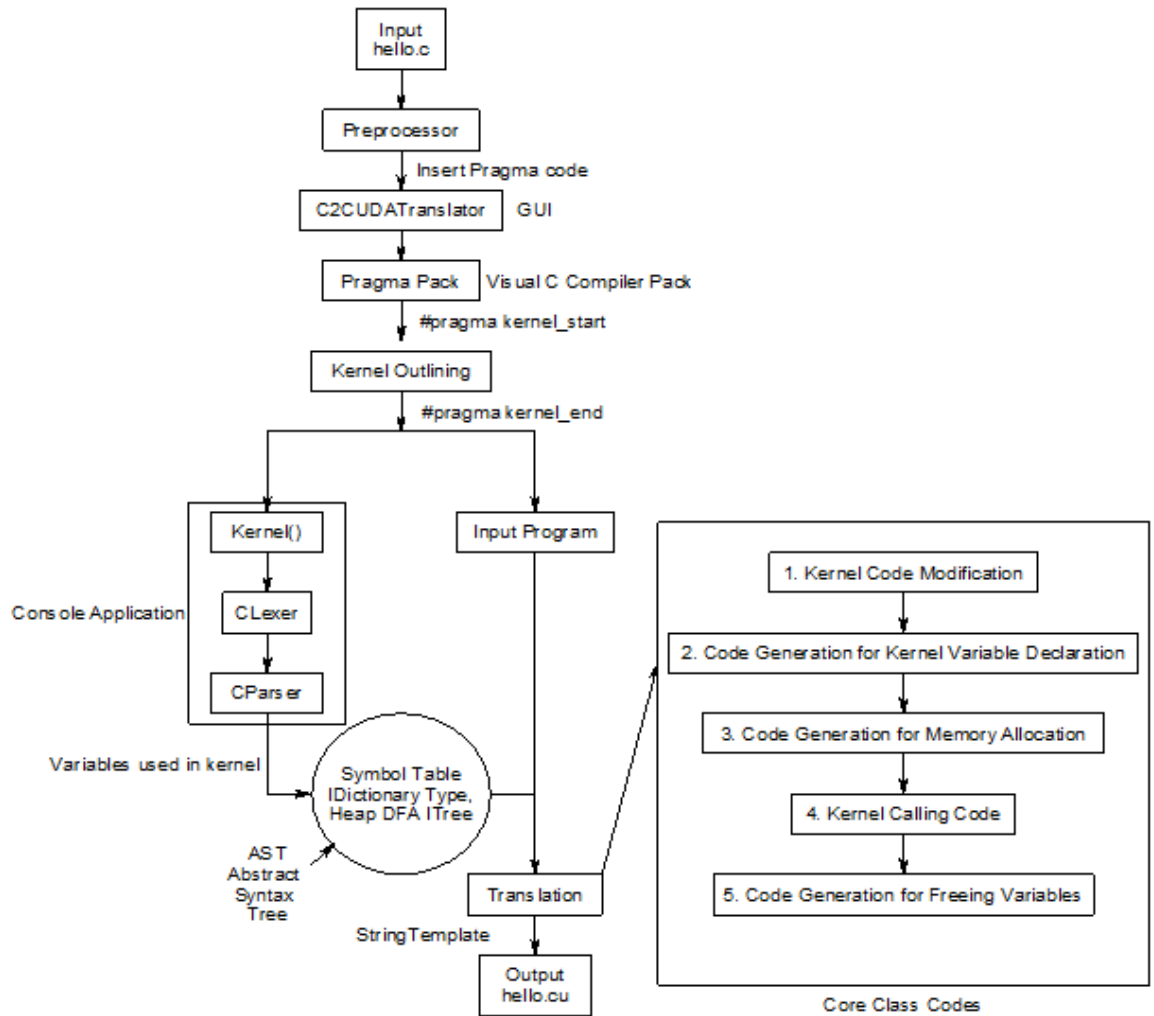


Figure 8.3: Flow of the C2CUDA Translator

For kernel code writing the translator performs 5 steps shown in figure above. They are nothing but the proposed patterns in previous sections. The translator generates codes for memory allocation on both host and devices and auto generates required pointers and variables. After that it ports the kernel region in kernel function. After parallelization is added it will be ported on many threads and blocks.

Finally, the CUDA file code is generated and extra functions in .c file are as it is in .cu file.

8.4 Preprocessor

In Pre-Processor phase I will try to implement a pre-processor that restricts users to use some of the compiler commands because my compiler is not ANSI C Compiler. So if suppose user will enter unsafe codes like "`#ifdef _cplusplus`" than my compiler will fail to perform the action that original ANSI C Compiler will do. And also some proper spacing and bracketing is important. So that we can understand the code. So I will try to make a C Pre-processor that will be inserted before the Scanner as shown in Compiler flow figure.

8.5 Code Transformation in Kernel Region

The kernel region identification is done by use of pragmas. There are two pragmas defined in my C2CUDA Translator.

- a. `#pragma kernel_start`
- b. `#pragma kernel_end`

Kernel region will be ported on GPU. So the code in the kernel contains CUDA code. That contains the related parallel version of the given code. The kernel call contains pointers to the variables. So that the input variables to kernel and output variables can be copied from / to host from / to device using `CUDAMemCpy` Function.

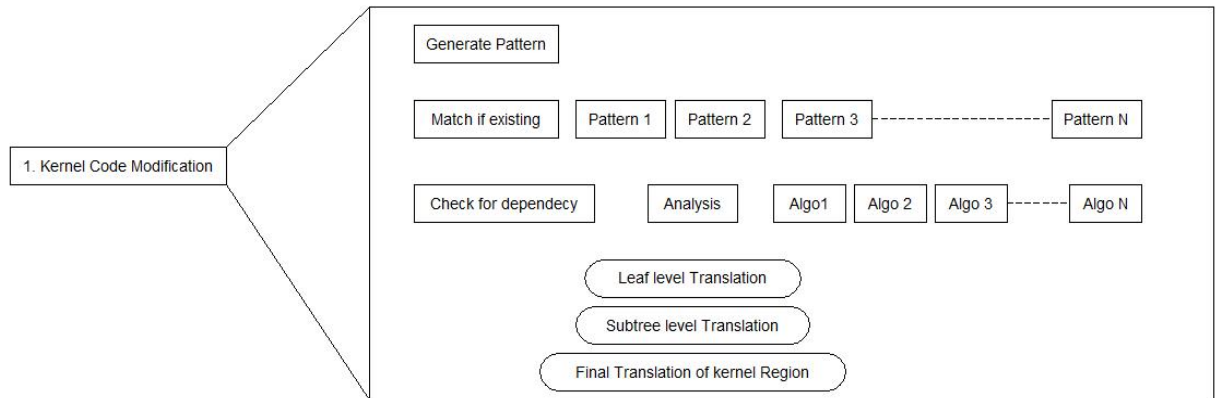


Figure 8.4: Code Transformation in Kernel Region

8.6 Compiler Style

For kernel region compiler has a unique style. There are some projects that start but never reach to the end. The algorithms for parallelization are more and time to add all of them are not enough for thesis. So I tried to implement a mechanism in which one can add more algorithms if one wants to.

Here compiler generates patterns for the input program. By the time we can see there are N numbers of programs and infinite. New programs and algorithms will be introduced to the CUDA by the time. So there will be N number of patterns.

Patterns are the heart of the compiler. Pattern describes the mechanism for the compiler like virus signature does in Antivirus software. New virus are always generated and corresponding signatures are also made in Antivirus. Similarly new patterns can be added later and so on.

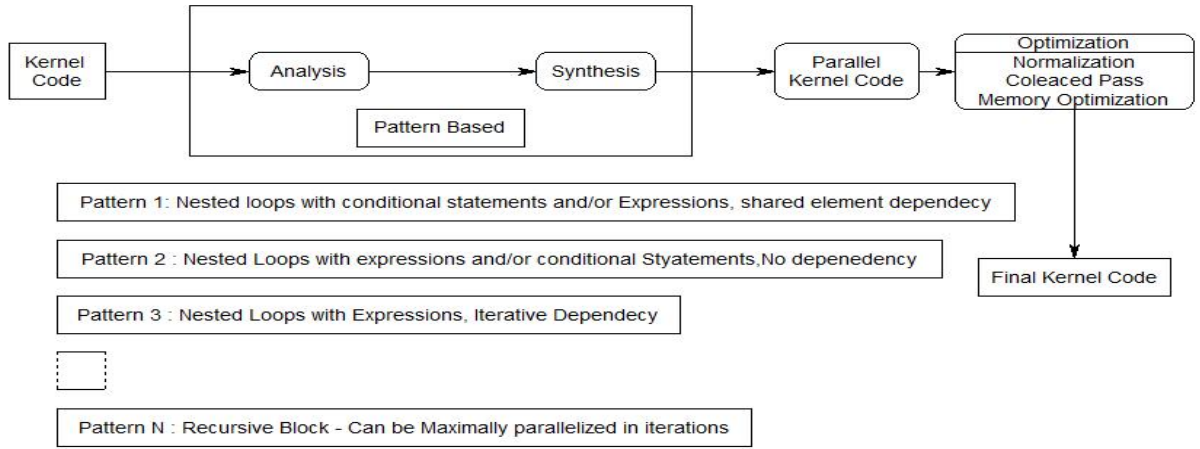


Figure 8.5: Compiler Style

8.7 Pattern Learning

In C2CUDATranslator the modules are divided such that future work can be implemented. Every programmer needs to learn language. This compiler is new developer for us. It learns the language in the form of Grammar given. Grammar provides syntax and semantic understanding to it.

Now each developer needs to learn parallelization fundamentals. Compiler has Analysis framework in which it has functions to check dependency, loop splitting and all.

Now new developer starts converting programs one by one and learns with experience. So, the modules are made like first syntax translation then kernel outlining and with this kernel code, the input will be given to Pattern Generator. This Pattern Generator generates pattern block for the pattern. While generating pattern it will also perform analysis at leaf, sub tree and tree level. Different algorithms are called at different level. This sets flags of the algorithms. This flags makes Condition Block of the Pattern.

Now if the pattern is new then compiler can either generate the code maximum it can transform and ask the developer to check whether the transformed code is correct or not. If not then the correct Transformation Block is made and compiler can use it for the next time.

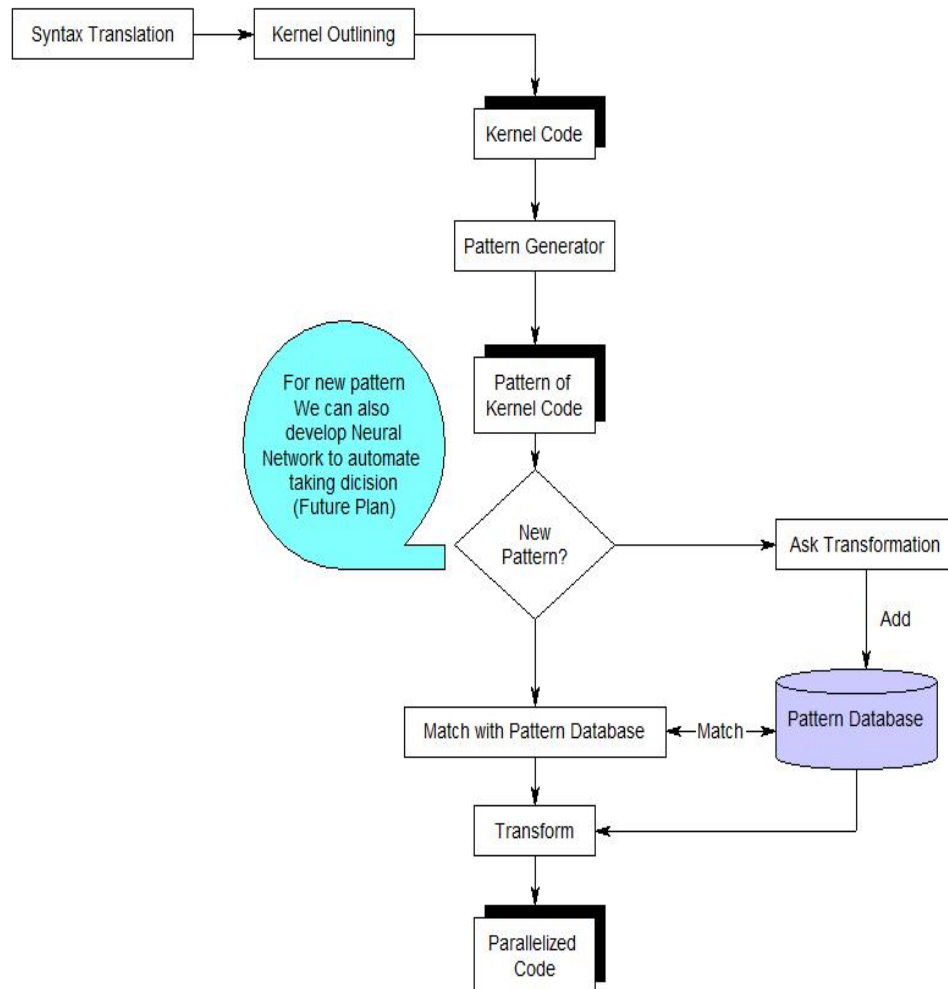


Figure 8.6: Pattern Learning

Once the pattern has been added to the database the compiler will autonomously convert it if it arrives next time as input. The compiler can learn in this way. We can have neural schema design for supervised learning. The weights can be assigned to patterns and transformation functions. These weights will be updated each time the new program gets converted. Maybe as the time goes on compiler will be having enough set of patterns which is subset of all the patterns that can be used to generate every kind of big patterns. Additionally the compiler can decide which function to call for optimal parallelization. Suppose there are more than two ways to parallelize than which one is best. We also learn

like this. The weights will be useful for decision.

8.8 Dependency Checker

Dependency Checker is the module of the compiler that is main parallelization computing brain of the compiler. In parallelization the problem is dependency. And to check it various algorithms exists in compilers.

Here in CUDA only kernel region needs to be parallelized so we have to check dependency only in kernel region. We can use AST generated by the compiler and while traversing through it to make pattern in pattern generator we can modify the pattern with different dependency analysis algorithms from `C2CUDATranslator.analysis` namespace. We have implemented flags mechanism to generate condition block of the pattern. In this mechanism the algorithm can be called and can be checked at any level of AST and respective flags can be set. According to the flag value the transformation is applied in Pattern. Condition Block is checked before transformation. The flag represent type of the dependency and compiler can identify the action to take for translation.

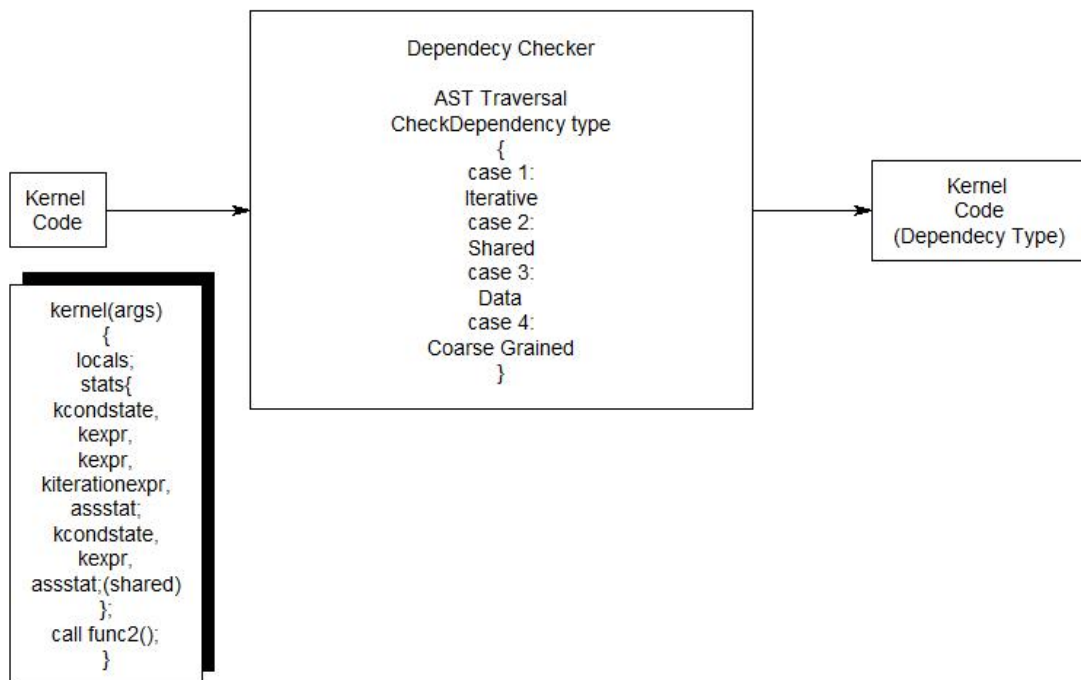


Figure 8.7: Dependency Checker

8.8.1 Dependency Analysis

Banerjee Wolf Test

The Banerjee Test finds dependency among iterations of loops. Banerjee Wolfe test is used to determine statement data dependence, subject to direction vectors, in automatic vectorization/parallelization of loops.

- If $a*i_1 - b*i'_1 = c$ has a solution, does it have a solution within the loop bounds for a given direction vector (i) or (=) in this case)?
- For our problem, does $i_1 - i'_1 = -1$ have a solution
 - For $i_1 = i'_1$, then it does not (no (=) dependence).

- For $i1 < i'1$, then it does ($<$) dependence).

This test returns boolean value that tells whether the expression is having iterative dependency or not. Suppose we pass i and $i + 1$ as two indexes then it will perform $i - (i+1)$. So, i will be canceled out and 1 will be remaining. Therefore, iterative dependency exists.

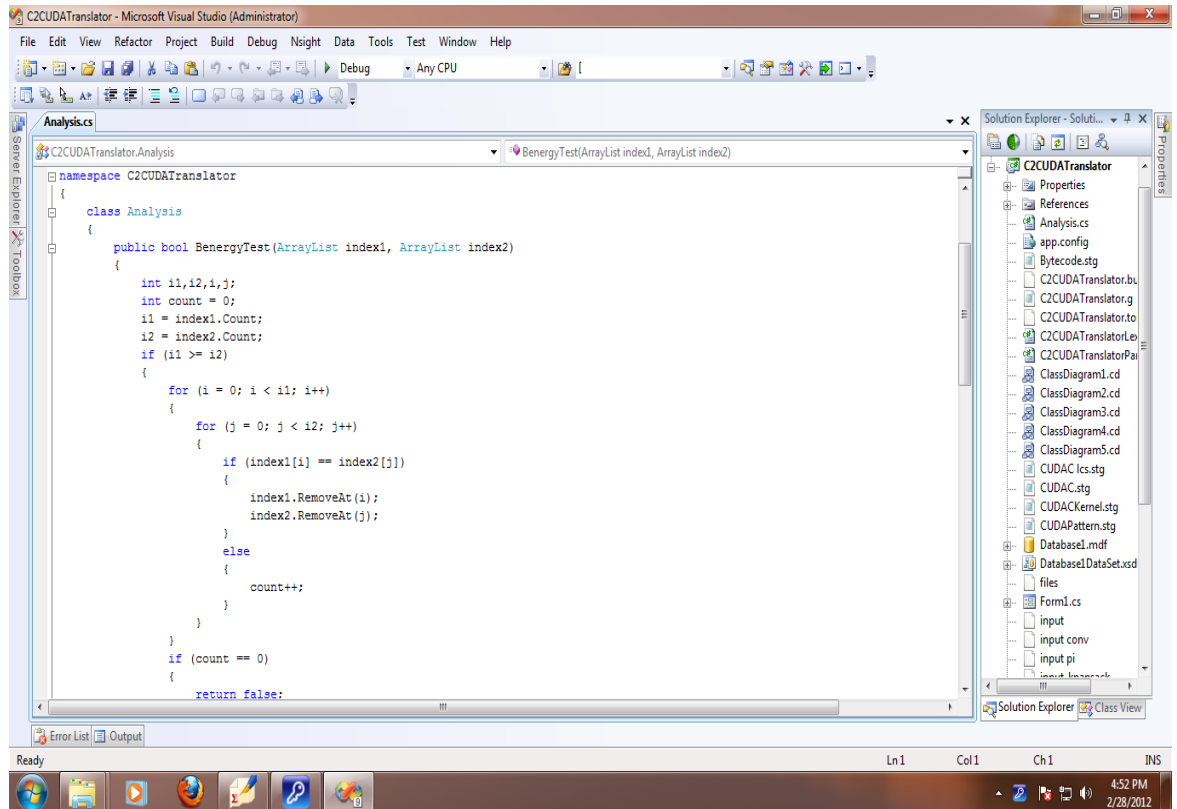


Figure 8.8: Banerjee Wolfe Test Function

8.8.2 Range Check

Scope management is very crucial task in compiler design. In my grammar I have defined scopes. They holds the local data. The term local data means the data used in that particular scope only. Range check analysis refers to searching the external memory references in the scope and values of the variables in proper range.

8.8.3 Recursion Check

Recursion means function dependent on itself. It is the issue when function calls itself in its own scope. In C2CUDATranslator I have one arraylist which maintains stack of called functions in the program. I have defined a condition in which it checks if the new called function is itself calling function by checking in arraylist. If it is already in arraylist then the flag for recursion will be set to true.

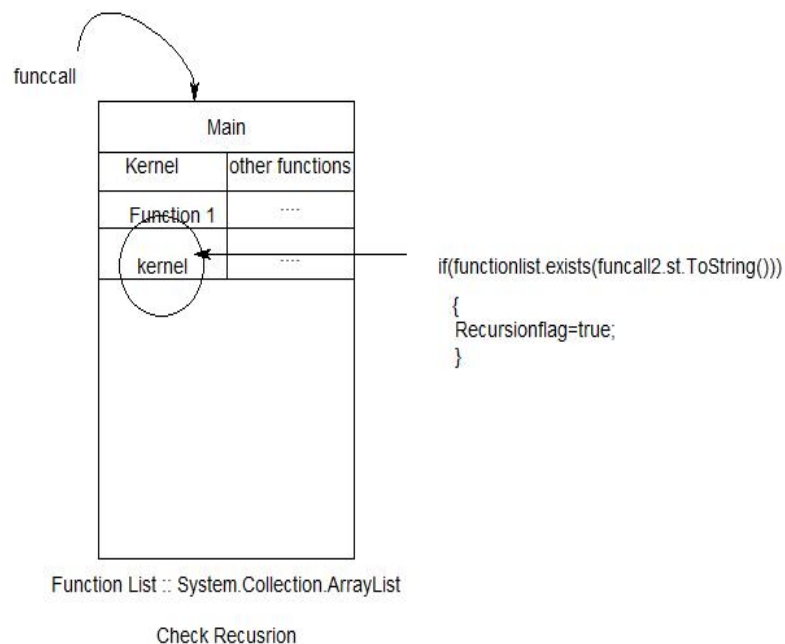


Figure 8.9: Recursion Check

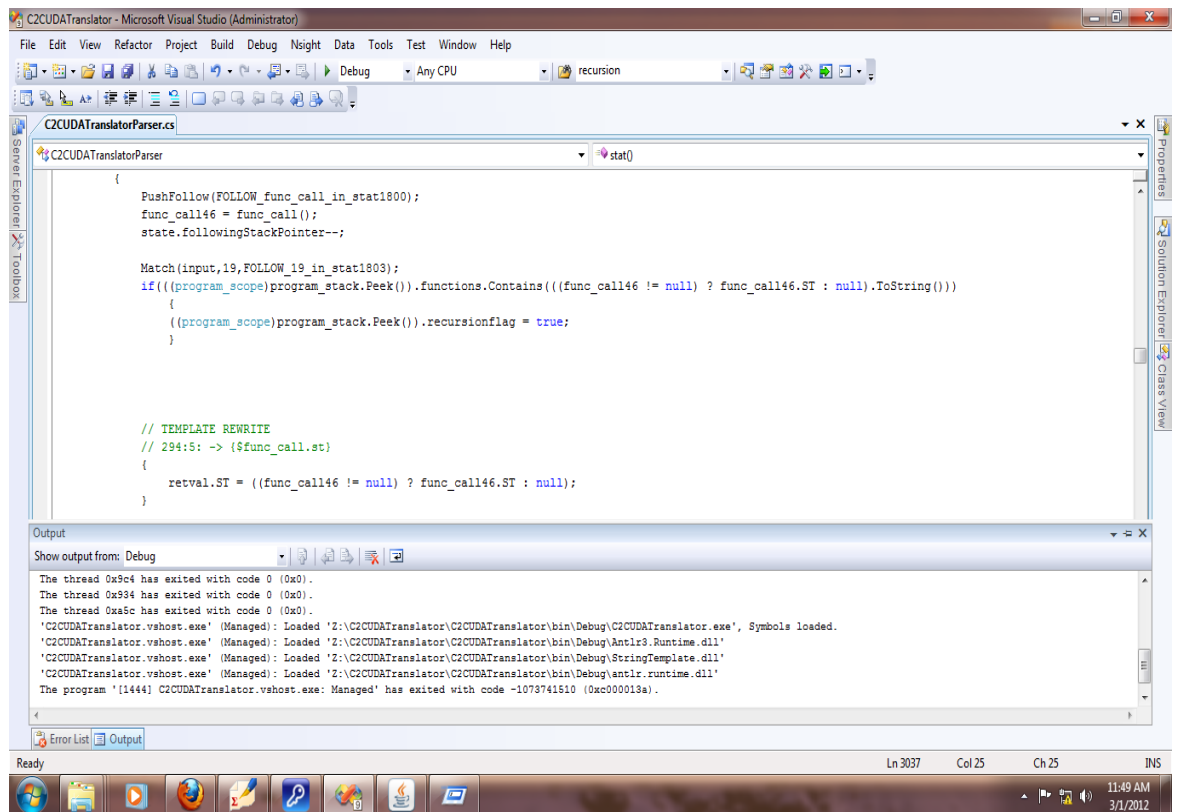


Figure 8.10: Recursion Check Function

8.9 Data Structures Used

- Stack
- Heap
- AST - Abstract Syntax Trees
- Array List
- Tokens
- Files
- DFA - Deterministic Finite Automaton
- Symbol Table
- NFA - Non Deterministic Finite Automaton

- *.stg - Group Files to store Intermediate Templates

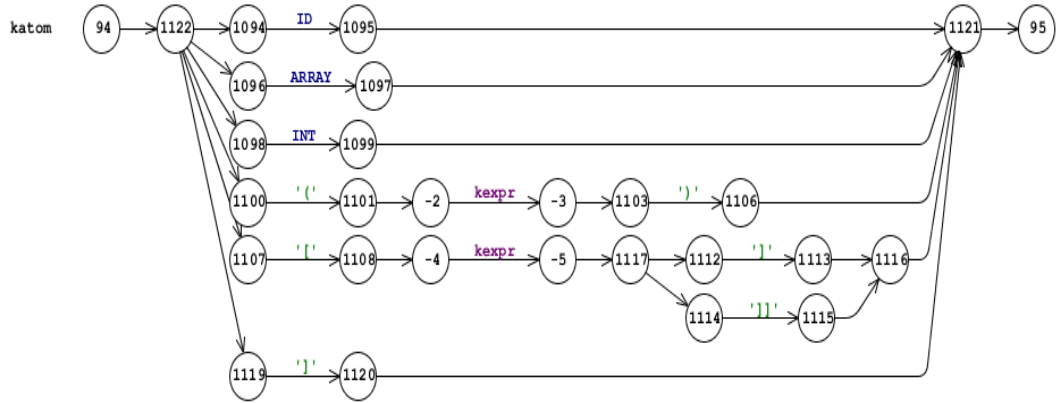


Figure 8.11: NFA Example

8.10 Interfaces used

- **IList:** Access to Array list to peek tokens
- **IToken:** Access to token
- **ISymbol:** Access to Symbols, Identifiers in Symbol Table
- **ILocal:** Local variables in Current Range or Function
- **IFunction:** Access to Function blocks defined in input

8.11 C2CUDATranslator Features

Table I: C2CUDA Translator Features

Sr.No	Features	Completed
1	Symbol Table of kernel region and program	YES
2	Required translation for kernel region	YES
3	Extra code generation for kernel variables as pointers	YES
4	Memory allocation variables code generation (Host, Device)	YES
5	Code generation for cudaMemcpy	YES
6	Kernel Parameter Passing	YES
7	Kernel Parameters Prefix Adding	YES
8	Iteration Statements (if, while ,do while loops)	YES
9	Simple printf Statements	YES
10	Complex printf Statements	YES
11	Double Arrays	YES
12	Pitch Arrays	YES
12	Pointers and double pointers	YES
13	Functions	YES

8.12 Grammar Example

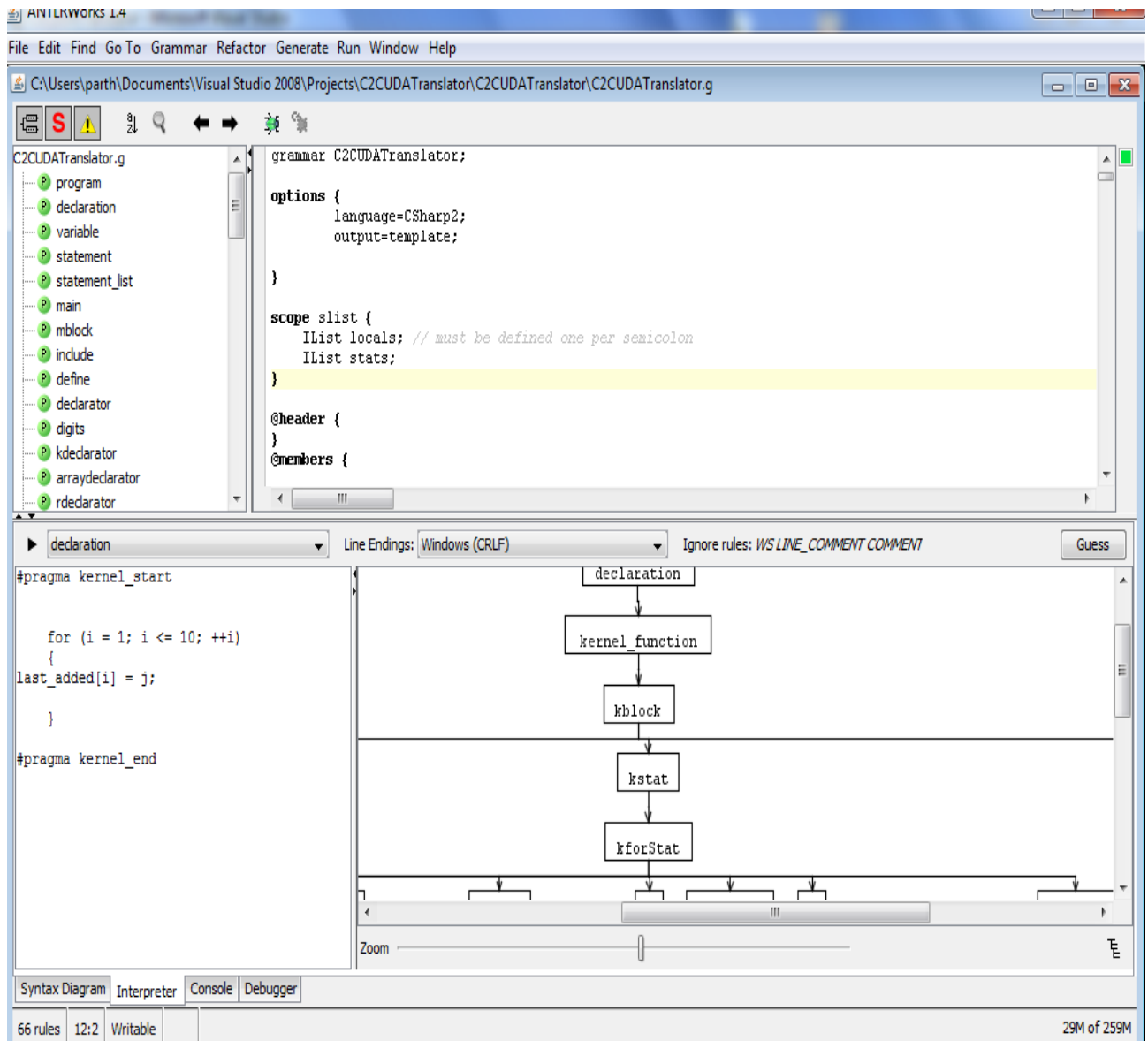


Figure 8.12: Grammar Example

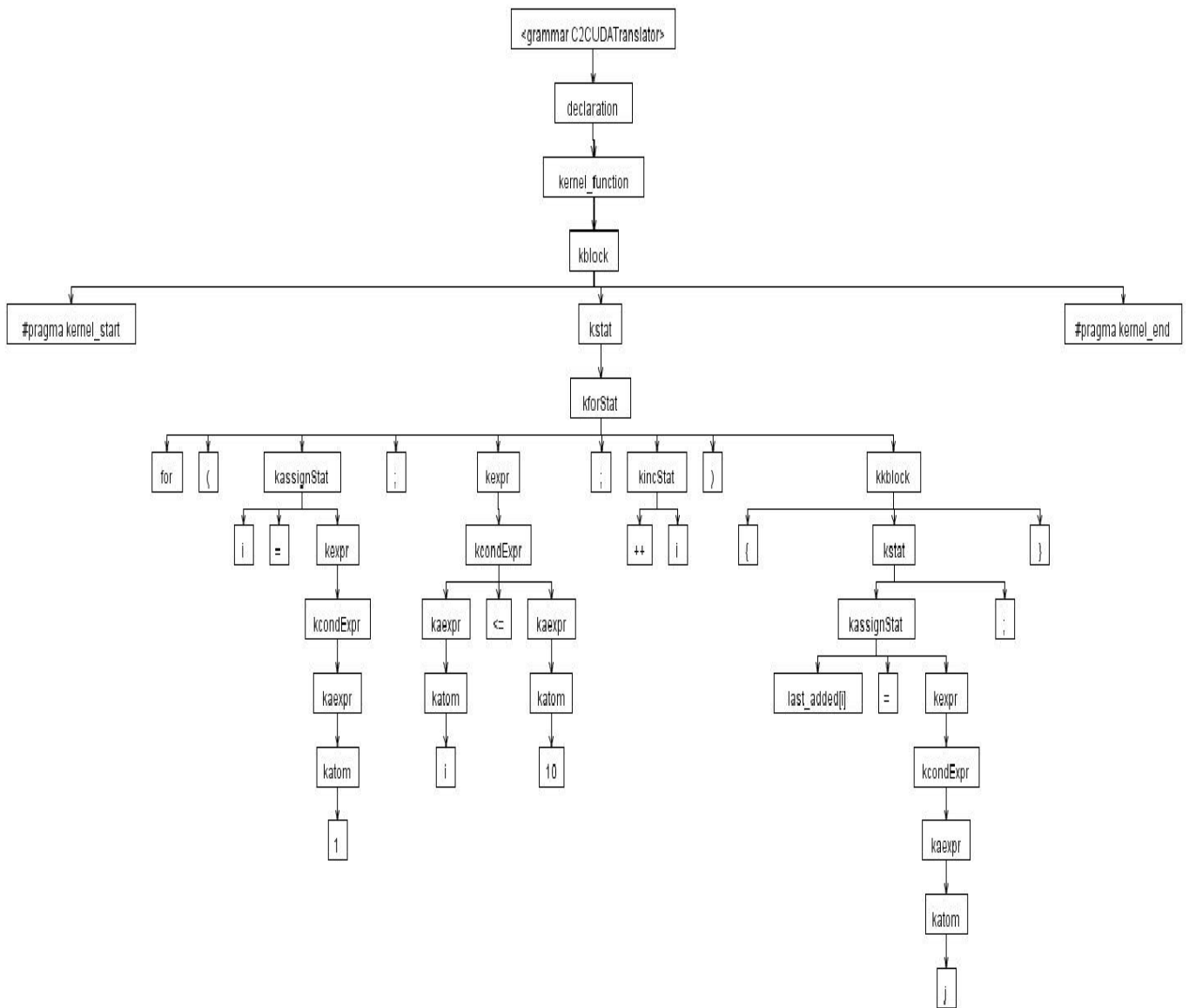
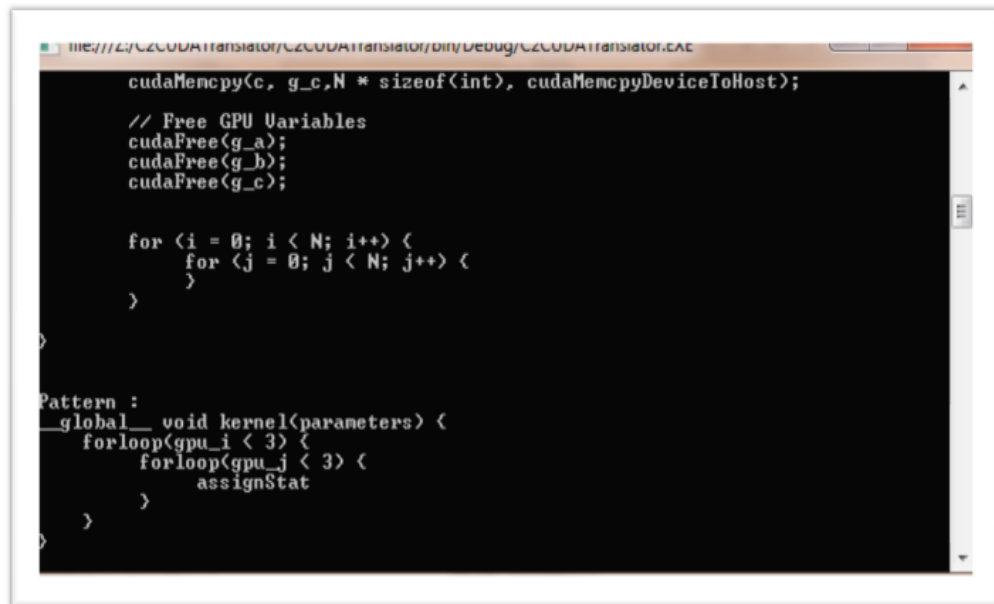


Figure 8.13: Grammar Tree Example

8.13 Pattern Example



```

file:///Z:/C/CUDAtranslator/C/CUDAtranslator/bin/Debug/C/CUDAtranslator.exe

    cudaMemcpy(c, g_c, N * sizeof(int), cudaMemcpyDeviceToHost);

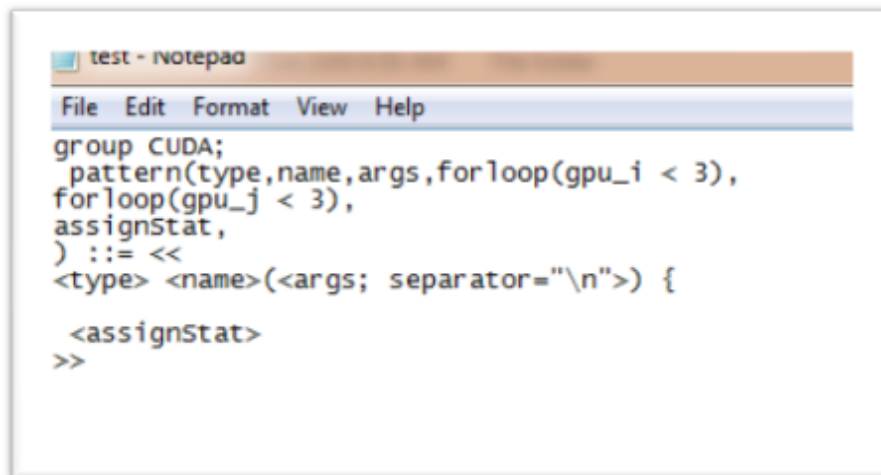
    // Free GPU Variables
    cudaFree(g_a);
    cudaFree(g_b);
    cudaFree(g_c);

    for (<i = 0; i < N; i++> <
        for (<j = 0; j < N; j++> <
        )
    )

Pattern :
__global__ void kernel(parameters) <
    forloop(gpu_i < 3) <
        forloop(gpu_j < 3) <
            assignStat
        )
    )
>

```

Figure 8.14: Pattern Example



```

test - notepad
File Edit Format View Help
group CUDA;
pattern(type,name,args,forloop(gpu_i < 3),
forloop(gpu_j < 3),
assignStat,
) ::= <<
<type> <name>(<args; separator="\n">) {
    <assignStat>
>>

```

Figure 8.15: Corresponding Pattern Example

8.14 Class Diagram

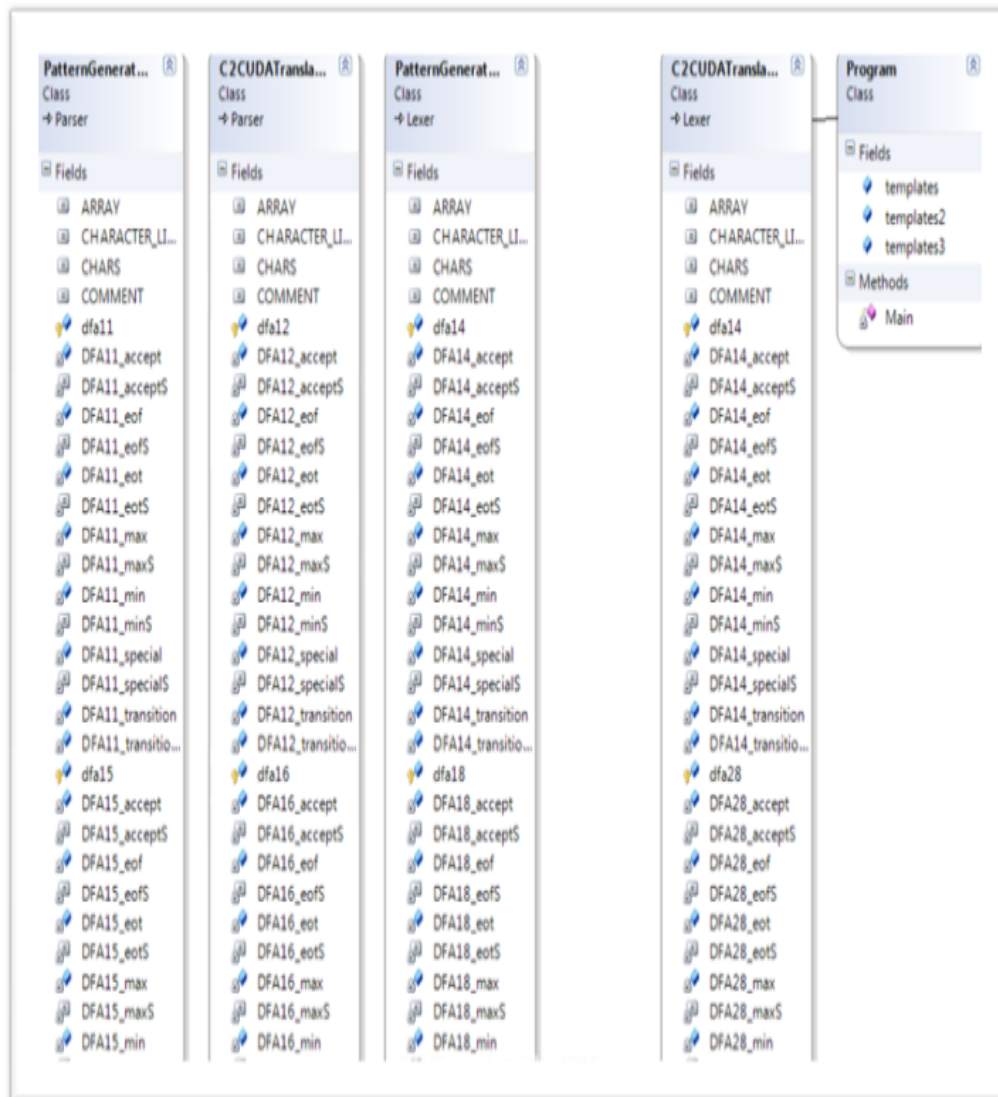
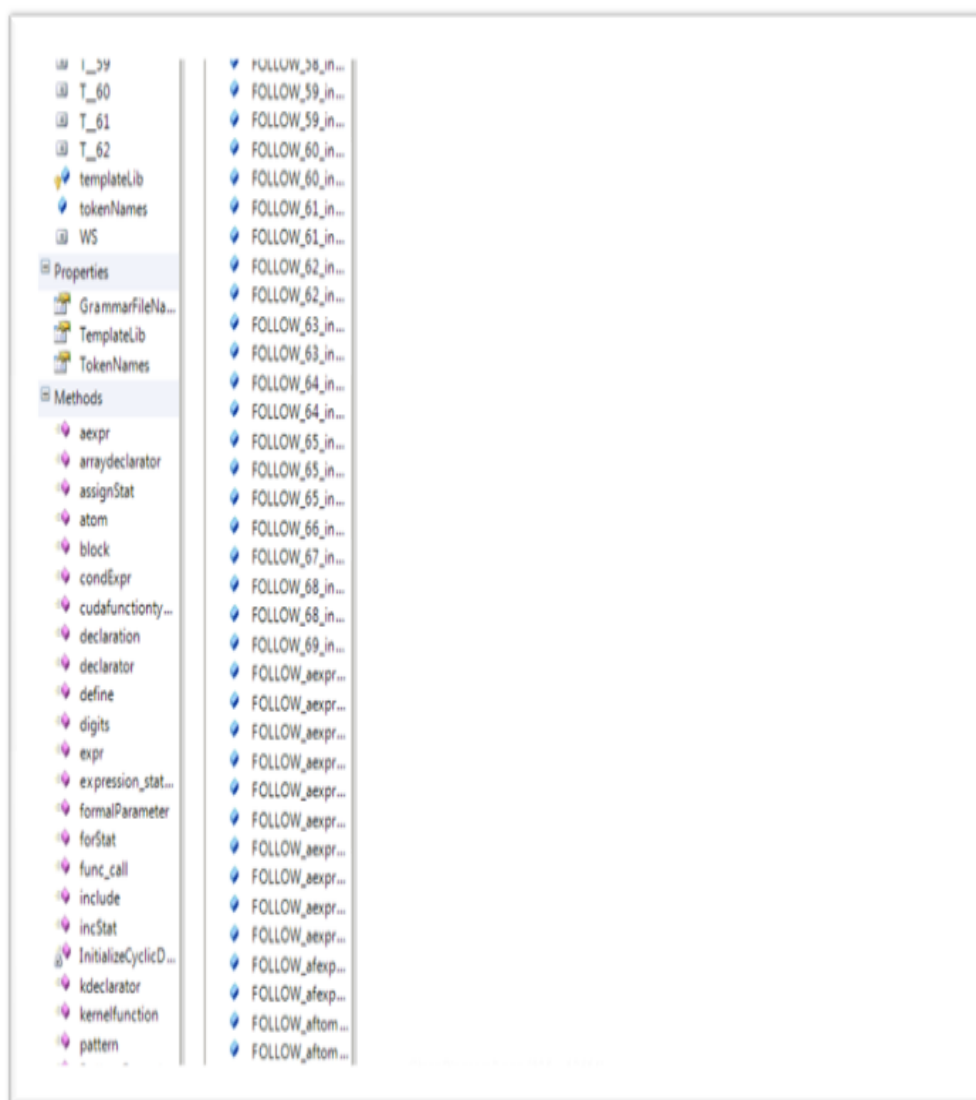


Figure 8.16: Class Diagram 1



8.15 Support Function Example

Support function is the function that supports the compiler in any phase. Here there is an example of the support function that helps compiler in syntax translation. This function adds prefix "gpu_" to all the variables in the kernel region. Hence this function helps parser it is written in the grammar file.

```

string getkernelstr(string getarr)
{
    string temp, result;
    temp = "";
    result = "";
    int i;
    for (i = 0; i < getarr.length ; i++)
    {
        if (getarr[i] == '[' || getarr[i] == ']' || getarr[i] == '+' || getarr[i] == '-' || getarr[i] == '*' ||
getarr[i] == '/')
        {
            temp = "";
            result = result+ getarr[i];
        }

        else
        {
            temp = "gpu_";
            while ((getarr[i] >= 'a' && getarr[i] <= 'z') || (getarr[i] >= 'A' && getarr[i] <= 'Z') || (getarr[i]
== '_' || (getarr[i] == '&') || (getarr[i] == '.'))
            {
                temp = temp + getarr[i];
                i++;
            }
            result = result + temp;
            i=i-1;
        }
    }
    return result;
}

```

Figure 8.18: Support Function Example

Chapter 9

Optimization

NVIDIA provides Visual profiler for profiling the CUDA programs and see how many micro seconds used in each function of the program. Profiler is very accurate and is used by the CUDA developers widely. Without profiling one can not see visually the power of GPUs.

9.1 How to use Visual Profiler?

To use visual profiler first we have to install it from NVIDIA web site. After installing it we can open it from Visual Profiler desktop icon. In session tab we need to specify the location of the exe file of our program.

Note: Do not use functions like "scanf()", "getch()", "printf()" or any I \O related instruction in the program while building the program exe for visual profiler.

9.1.1 Getting started for optimization

To get started with visual profiler first open visual profiler. Go to **Session** tab and specify the location of your project exe. Now Click on **Launch**. Profiler launches your exe 15 times and shows you average of 15 execution times. Profiler allows you to compare the results with previous results in graphs. Now you need to do changes in code and each time you need to check execution time for reduction in micro cycles.

9.2 Maximize Memory Throughput

We have seen different kind of memory throughput optimization in our chapter 3. Generally pinned memory allocation and use of streams provides better optimization. Until necessary the translator avoids use of global memory. But user needs to specify "local" and "shared" before "kernel" phrase in input program.

9.2.1 Insertion of keywords for placing in memory

In C2CUDATranslator there is a collection for Read Only and Write Only variables. So if the variables a and b are used in kernel region but they are read but never updated then the translator will put "__device__ __constant__" before declaring them for Device memory allocation.

Similarly if the variable is frequently updated in so many continuous assignment statements, the translator puts "register" in kernel region before declaration of that variable.

9.2.2 Use of CUDAMemSet

The dramatic reduction in execution time can be made if we use CUDAMemSet in our program. This is function is very useful in initialization of Device Memory. We need to first allocate variables using pinned memory and then we can use this function to initialize the variables. For Example if we want to code a large array and we have zero value initially.

```
cudaMallocHost((void**)&h_A, sizeimg * sizeof(float));
cudaMalloc((void**)&d_A, sizeimg * sizeof(float));
cudaMemset(d_A, 0, sizeimg * sizeof(float));
```

9.2.3 Asynchronous Transfers and Overlapping Transfers with Computation

We can transfer data between host and device using cudaMemcpy(), which is the blocking transfers, whereas cudaMemcpyAsync() provides non-blocking transfers in which control is returned immediately to the host thread. The asynchronous transfer requires pinned host

memory and streamID (A stream is a sequence of operations that are performed in order on the device).

We can overlap data transfers and computation using asynchronous transfers. There are two ways to do this first, on all CUDA-enabled devices; it is possible to overlap host computation with asynchronous data transfers and with device computations. For example, code demonstrates how host computation in the routine `cpuFunction()` is performed while data is transferred to the device and a kernel using the device is executed.

The last argument to the `cudaMemcpyAsync()` function is the stream ID, which in this case uses the default stream, stream 0. The kernel also uses the default stream, and it will not begin execution until the memory copy completes; therefore, no explicit synchronization is needed. Because the memory copy and the kernel both return control to the host immediately, the host function `cpuFunction()` overlaps their execution. In example below , the memory copy and kernel execution occur sequentially.

```
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, 0);
kernel<<<grid, block>>>(a_d);
cpuFunction();
```

Second, on devices that are capable of concurrent copy and execute, it is possible to overlap kernel execution on the device with data transfers between the host and the device. Whether a device has this capability is indicated by the `deviceOverlap` field of a `cudaDeviceProp`. On devices that have this capability, the overlap once again requires pinned host memory, and, in addition, the data transfer and kernel must use different, non-default streams (streams with non-zero stream IDs). Non-default streams are required for this overlap because memory copy, memory set functions, and kernel calls that use the default stream begin only after all preceding calls on the device (in any stream) have completed, and no operation on the device (in any stream) commences until they are finished.

```
cudaStreamCreate(&stream1); cudaStreamCreate(&stream2);
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, stream1);
kernel<<<grid, block, 0, stream2>>>(otherData_d);
```

But to decide when to use which mechanism is dependent on code and data. If one want to use flag variable whose value decide which memory copying mechanism to use then in C2CUDATranslator it is possible.

9.3 Decision Making

”How humans make decision?” is inhomogeneous in nature. Because each time the program optimization comes, humans check microseconds in visual profiler and can undo the changes if execution time is more. But the challenge is how to make computer aware of optimization strategies and results? How to make a computer to make a decision that where to put this variable, in which memory, or which mechanism would be better, pinned memory or grouped transfers or streams? This is very crucial challenge. But we have Neural Networks to overcome such situations where we need to make a decision support system.

9.4 Artificial Neural Network

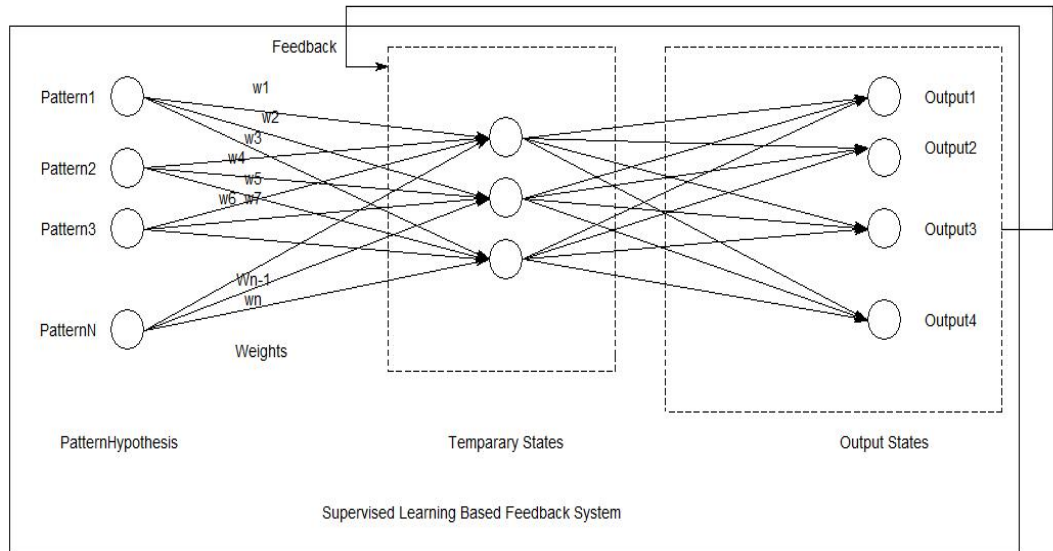


Figure 9.1: Pattern Neural Design

In C2CUDATranslator the modules are divided such that future work can be imple-

mented.

We have neural schema design for supervised learning. The weights are assigned to patterns and transformation functions. These weights will be updated each time the new program gets converted. Maybe as the time goes on compiler will be having enough set of patterns which is subset of all the patterns that can be used to generate every kind of big patterns. Additionally the compiler can decide which function to call for optimized parallelization. Suppose there are more than two ways to parallelize a program than which one is best. We also learn like this. The weights will be useful for decision.

9.4.1 Compiler Learning

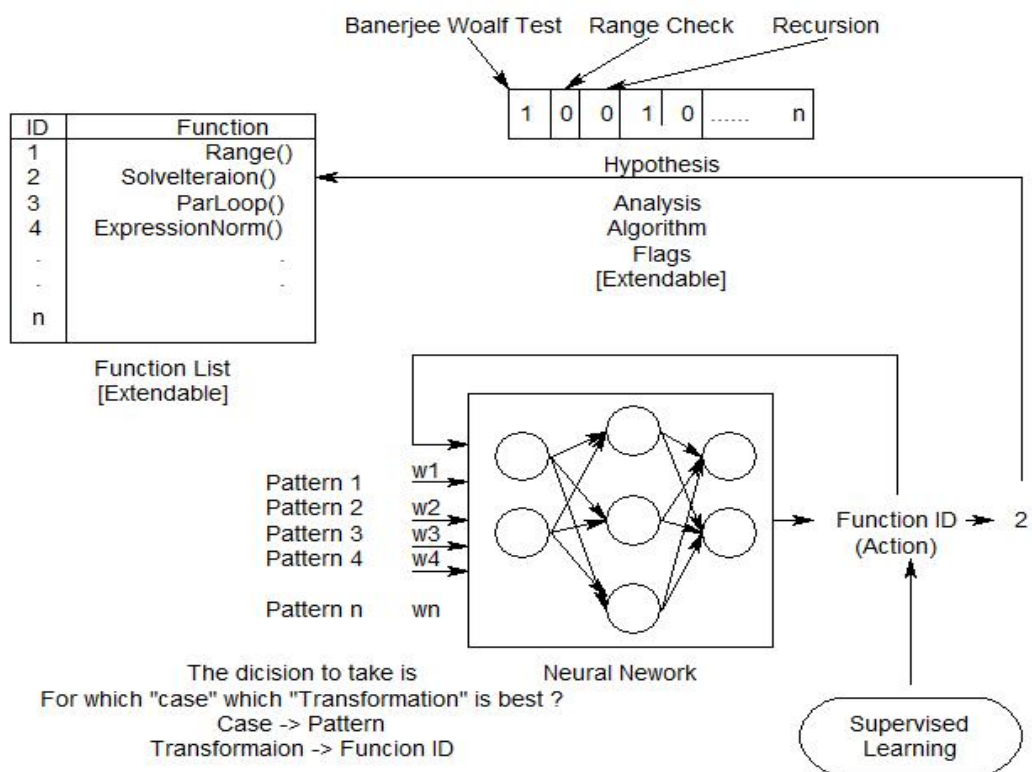


Figure 9.2: Compiler Learning

Chapter 10

Outcome

I tried to translate following algorithms using the C2CUDATranslator. The input to the program is below. The output screen shots are in Appendix A.

- a. Sum of Two Vectors (Mathematical Algorithm)
- b. Binary Search (Searching Algorithm)
- c. Krushkal Algorithm (Dynamic Programming Algorithm)
- d. Longest Common Subsequence (Dynamic Programming Algorithm)
- e. Knapsack Problem (Dynamic Programming Algorithm)
- f. Convolution (Infinite Series Algorithm)
- g. Calculating value of pi (Infinite Series Algorithm)
- h. Blurring Filter (Image Processing Algorithm)
- i. Smothering Filter (Image Processing Algorithm)

Here I have added example of knapsack [26] problem. The outputs are also on project web site [22].

10.1 Input

```

#include <stdio.h>

#define MAXWEIGHT 100

int main( int argc, char *argv [] )
{
    kernel int n=3; /* The number of objects */
    kernel int c[10]; // /* c[i] is the *COST* of the ith object; i.e. what YOU PAY to take the
    object */
    kernel int v[10]; // /* v[i] is the *VALUE* of the ith object; i.e. what YOU GET for taking
    the object */
    kernel int W=10; /* The maximum weight you can take */
    kernel int a[MAXWEIGHT]; /* a[i] holds the maximum value that can be obtained using
    at most i weight */
    kernel int last_added[MAXWEIGHT]; /* I use this to calculate which object were added */
    int i,j;
    int aux;
    c[0]=8; c[1]=16; c[2]=4;
    v[0]=16; v[1]=10; v[2]=7;
    for ( i = 0; i <= W; ++i)
    {
        a[i] = 0;
        last_added[i] = -1;
    }
    #pragma kernel_start
    a[0] = 0;
    for ( i = 1; i <= W; ++i)
    {
        for ( j = 0; j < n; ++j)
        {
            if ( ( c[j] <= i ) && ( a[i] < ( a[i-c[j]] + v[j] ) ) )

```

```

{
a[i] = a[i-c[j]] + v[j];
last_added[i] = j;
}
}
}

#pragma kernel_end

for (i = 0; i <= W; ++i)
{
if (last_added[i] != -1)
{
printf("Weight %d; Benefit: %d; To reach this weight I added object %d (%d$ %dKg) to
weight %d. \n", i, a[i], last_added[i] + 1, v[last_added[i]], c[last_added[i]], i - c[last_added[i]]);
}
else
{
printf("Weight %d; Benefit: 0; Can't reach this exact weight.\n", i);
}
} printf("—\n");

aux = W;
while ( (aux > 0) && (last_added[aux] != -1) )
{
printf("Added object %d (%d$ %dKg). Space left: %d \n", last_added[aux] + 1, v[last_added[aux]],
c[last_added[aux]], aux - c[last_added[aux]]);
aux -= c[last_added[aux]];
}
printf("Total value added: %d$ \n", a[W]);
return 0;
}

```


10.2 Output

10.2.1 Output Screenshot

```

file:///C:/Users/Parth/Documents/Visual Studio 10/Projects/C2CUDATranslator/C2CUDATranslator/...
/*Parth C2CUDA Generated CODE */
#include <stdio.h>
#define MAXWEIGHT 100
int main()
/* Include CUDA runtime library headers */
#include <cuda_runtime.h>
__global__ void kernel<int *g_n,int *g_c,int *g_v, int *g_w,int *g_a,
int *g_last_added> <
    for <int gpu_i = 1; gpu_i <= gpu_W; ++i> <
        for <int gpu_j = 0; gpu_j < gpu_n; ++j> <
            if <<gpu_c[j] <= gpu_i> && <gpu_a[i] < gpu_a[i-c[j]] + gpu_v[j]>> <
                gpu_a[i] = gpu_a[i-c[j]] + gpu_v[j];
                gpu_last_added[i] = gpu_j;
            >
        >
    >
int main(int argc, char *argv[]) <
    int i,j;
    int aux;
    //Kernel Variables
    int n;int *g_n;
    int c[10];int *g_c;
    int v[10];int *g_v;
    int W=10;int *g_w;
    int a[MAXWEIGHT];int *g_a;
    int last_added[MAXWEIGHT];int *g_last_added;
    //CUDA GRID BLOCK SIZE AND NUMBER OF BLOCKS
    int block_size = 1;
    const int N = 10; // Number of elements in arrays
    int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
    c[0] = 8;
    c[1] = 16;
    c[2] = 4;
    c[3] = 16;
    v[0] = 10;
    v[1] = 10;
    for <i = 0; i <= W; ++i> <
        a[i] = 0;
        last_added[i] = -1;
    >
    // Memory Allocation
    g_n = <int *>malloc<sizeof<int>>; // Allocate array on host
    cudaMalloc<<void *>> &n, sizeof<int>; // Allocate array on device
    g_c = <int *>malloc<sizeof<int>>; // Allocate array on host
    cudaMalloc<<void *>> &c, 10 * sizeof<int>; // Allocate array on device
    g_v = <int *>malloc<sizeof<int>>; // Allocate array on host
    cudaMalloc<<void *>> &v, 10 * sizeof<int>; // Allocate array on device
    g_w = <int *>malloc<sizeof<int>>; // Allocate array on host
    cudaMalloc<<void *>> &w, 10 * sizeof<int>; // Allocate array on device
    g_a = <int *>malloc<sizeof<int>>; // Allocate array on host
    cudaMalloc<<void *>> &a, MAXWEIGHT * sizeof<int>; // Allocate array on
device
    g_last_added = <int *>malloc<sizeof<int>>; // Allocate array on host
    cudaMalloc<<void *>> &last_added, MAXWEIGHT * sizeof<int>; // Allocate
array on device
    // Copy Data to device from host
    cudaMemcpy<g_n, n, sizeof<int>, cudaMemcpyHostToDevice>;
    cudaMemcpy<g_c, c, 10 * sizeof<int>, cudaMemcpyHostToDevice>;
    cudaMemcpy<g_v, v, 10 * sizeof<int>, cudaMemcpyHostToDevice>;
    cudaMemcpy<g_w, w, 10 * sizeof<int>, cudaMemcpyHostToDevice>;
    cudaMemcpy<g_a, a, MAXWEIGHT * sizeof<int>, cudaMemcpyHostToDevice>;
    cudaMemcpy<g_last_added, last_added, MAXWEIGHT * sizeof<int>, cudaMemcpyH
ostToDevice>;
    // call kernel
    kernel <<< n_blocks, block_size >>>< int *g_n,int *g_c,int *g_v, int *g
_w,int *g_a, int *g_last_added>;
    // Retrieve result from device and store it in host array
    cudaMemcpy<g_n, g_n, sizeof<int>, cudaMemcpyDeviceToHost>;
    cudaMemcpy<g_c, g_c, 10 * sizeof<int>, cudaMemcpyDeviceToHost>;
    cudaMemcpy<g_v, g_v, 10 * sizeof<int>, cudaMemcpyDeviceToHost>;
    cudaMemcpy<g_w, g_w, 10 * sizeof<int>, cudaMemcpyDeviceToHost>;
    cudaMemcpy<g_a, g_a, MAXWEIGHT * sizeof<int>, cudaMemcpyDeviceToHost>;
    cudaMemcpy<last_added, g_last_added, MAXWEIGHT * sizeof<int>, cudaMemcpyD
eviceToHost>;
    // Free GPU Variables
    cudaFree<g_n>;
    cudaFree<g_c>;
    cudaFree<g_v>;
    cudaFree<g_w>;
    cudaFree<g_a>;
    cudaFree<g_last_added>;
    for <i = 0; i <= W; ++i> <
        if <last_added[i] != -1> <
            printf<<"Weight %d; Benefit: %d; To reach this weight I added ob
ject %d <zd$ zdKg> to weight %d.\n", i, a[i], last_added[i], i, c[last_added[i]], c[
last_added[i]-1]>> <
        >
        else <
            printf<<"Weight %d; Benefit: 0; Can't reach this exact weight.\n"
">> <
        >
    >
    printf<<"---\n">> <
    aux = W;
    while <aux > 0> && <last_added[aux] != -1>> <
        printf<<"Added object %d <zd$ zdKg>. Space left: %d\n", last_added[au
x] + 1, c[last_added[aux]], c[last_added[aux]], aux - c[last_added[aux]]>> <
        aux = c[last_added[aux]];
    >
    printf<<"Total value added: %d\n", a[W]>> <
    return 0;
}

```

Figure 10.1: Output of C2CUDATranslator

10.2.2 Output in words

```

/*Parth C2CUDA Generated CODE */
#include <stdio.h>
#define MAXWEIGHT 100
int main()
/* Include CUDA runtime library headers */
#include <cuda_runtime.h>
__global__ void kernel( int *gpu_n,int *gpu_c,int *gpu_v, int *gpu_W,int *gpu_a, int *gpu_last_added)
{
    gpu_a[0] = 0;
    int gpu_i = threadIdx.x+1;
    for (int gpu_j = 0; gpu_j < gpu_n; ++j)
    {
        if ((gpu_c[j] != gpu_i) && (gpu_a[gpu_i] != gpu_a[gpu_i-c[j]] + gpu_v[gpu_j]))
        {
            gpu_a[gpu_i] = gpu_a[gpu_i-gpu_c[gpu_j]] + gpu_v[gpu_j];
            gpu_last_added[gpu_i] = gpu_j;
        }
    }
}

int main(int argc, char *argv[])
{
    int i,j;
    int aux;
    //Kernel Variables
    int n=3;int *g_n;
    int c[10];int *g_c;
    int v[10];int *g_v;
    int W=10;int *g_W;
    int a[MAXWEIGHT];int *g_a;

```

```

int last_added[MAXWEIGHT];int *g_last_added;

//CUDA GRID BLOCK SIZE AND NUMBER OF BLOCKS
int block_size = 1;
const int N = 10; // Number of elements in arrays
int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);

c[0] = 8;
c[1] = 16;
c[2] = 4;
v[0] = 16;
v[1] = 10;
v[2] = 7;
for (i = 0; i <= W; ++i) {
a[i] = 0;
last_added[i] = -1;
}

// Memory Allocation
g_n= (int *)malloc(sizeof(int)); // Allocate array on host
cudaMalloc((void **) &n,sizeof(int)); // Allocate array on device
g_c= (int *)malloc(sizeof(int)); // Allocate array on host
cudaMalloc((void **) &c,10 * sizeof(int)); // Allocate array on device
g_v= (int *)malloc(sizeof(int)); // Allocate array on host
cudaMalloc((void **) &v,10 * sizeof(int)); // Allocate array on device
g_W= (int *)malloc(sizeof(int)); // Allocate array on host
cudaMalloc((void **) &W,sizeof(int)); // Allocate array on device
g_a= (int *)malloc(sizeof(int)); // Allocate array on host
cudaMalloc((void **) &a,MAXWEIGHT * sizeof(int)); // Allocate array on device
g_last_added= (int *)malloc(sizeof(int)); // Allocate array on host
cudaMalloc((void **) &last_added,MAXWEIGHT * sizeof(int)); // Allocate array on device

```

```

// Copy Data to device from host
cudaMemcpy(g_n, n, sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(g_c, c, 10 * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(g_v, v, 10 * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(g_W, W, sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(g_a, a, MAXWEIGHT * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(g_last_added, last_added, MAXWEIGHT * sizeof(int), cudaMemcpyHostToDevice);

// call kernel
kernel <<< n_blocks, block_size >>>( int *g_n, int *g_c, int *g_v, int *g_W, int *g_a, int
*g_last_added);

// Retrieve result from device and store it in host array
cudaMemcpy(n, g_n, sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(c, g_c, 10 * sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(v, g_v, 10 * sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(W, g_W, sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(a, g_a, MAXWEIGHT * sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(last_added, g_last_added, MAXWEIGHT * sizeof(int), cudaMemcpyDeviceToHost);

// Free GPU Variables
cudaFree(g_n);
cudaFree(g_c);
cudaFree(g_v);
cudaFree(g_W);
cudaFree(g_a);
cudaFree(g_last_added);
for (i = 0; i <= W; ++i) {

```

```

if (last_added[i] != -1){
    printf("Weight %d; Benefit: %d; To reach this weight I added object %d (%d$ %dKg) to
weight %d. \n ",i,a[i],last_added[i] + 1,v[last_added[i]],c[last_added[i]],i - c[last_added[i]]);
}
else {
    printf("Weight %d; Benefit: 0; Can't reach this exact weight.\n",i);
}

}
printf("\n\n");
aux = W;
while ((aux > 0) && (last_added[aux] != -1)) {
    printf("Added object %d (%d$ %dKg). Space left: %d \n", last_added[aux] + 1, v[last_added[aux]],
c[last_added[aux]], aux - c[last_added[aux]]);
    aux -= c[last_added[aux]];
}
printf("Total value added: %d$ \n",a[W]);
return 0;

}

```

10.2.3 Parallelization Achieved

Here, the pattern is

```

    forloop1
{
    forloop2
{
    assStat1 assStat2 }
}

```

In which the assignment statements are two expressions and Banerjee Woalfe Test returns dependency in `assSattel1`. So it depends on previous values. Knapsack problem is dynamic algorithm which is recursive by nature. So here we can check over loop which can be parallelized using thread level parallelization. Here we created 10 threads that runs first Iteration for $j=0$. Here, value of i is assigned `threadIdx.x`.

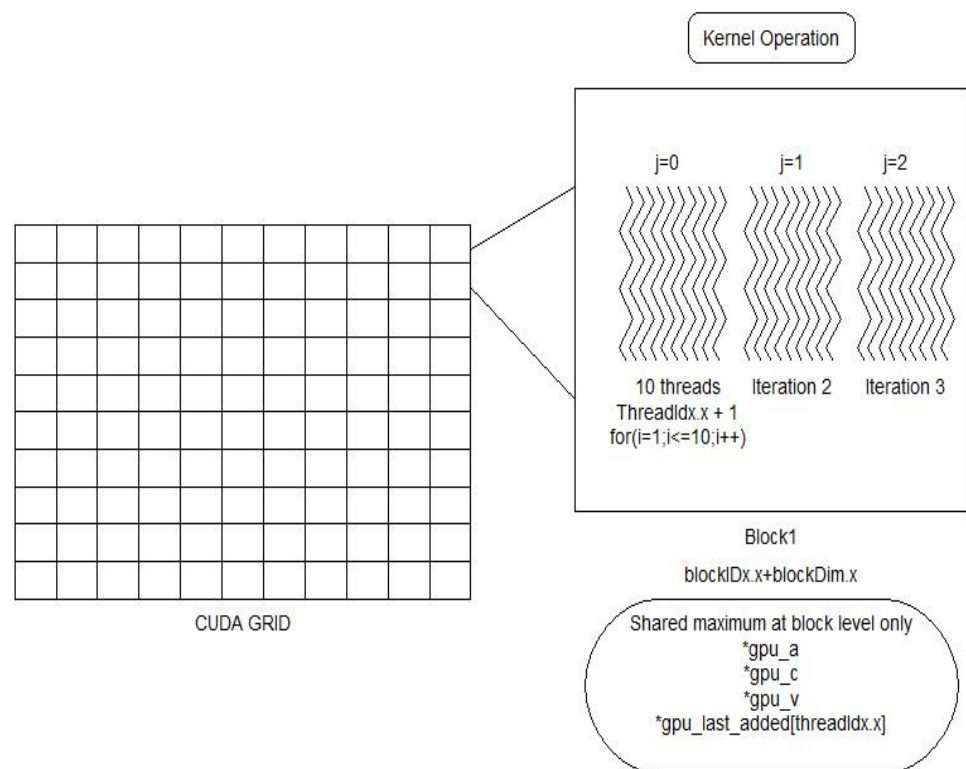


Figure 10.2: Knapsack Grid

10.2.4 Output of Converted Program (C2CUDATranslator)

```

c:\Users\parth\Documents\Visual Studio 2008\Projects\Knapsack\Debug\Knapsack.exe
Weight 0; Benefit: 0; Can't reach this exact weight.
Weight 1; Benefit: 0; Can't reach this exact weight.
Weight 2; Benefit: 0; Can't reach this exact weight.
Weight 3; Benefit: 0; Can't reach this exact weight.
Weight 4; Benefit: 7; To reach this weight I added object 3 <7$ 4Kg> to weight 0
Weight 5; Benefit: 7; To reach this weight I added object 3 <7$ 4Kg> to weight 1
Weight 6; Benefit: 10; To reach this weight I added object 2 <10$ 6Kg> to weight 0
Weight 7; Benefit: 10; To reach this weight I added object 2 <10$ 6Kg> to weight 1
Weight 8; Benefit: 16; To reach this weight I added object 1 <16$ 8Kg> to weight 0
Weight 9; Benefit: 16; To reach this weight I added object 1 <16$ 8Kg> to weight 1
Weight 10; Benefit: 17; To reach this weight I added object 2 <10$ 6Kg> to weight 4.
Added object 2 <10$ 6Kg>. Space left: 4
Added object 3 <7$ 4Kg>. Space left: 0
Total value added: 17$

```

Figure 10.3: Output of Converted Program

10.2.5 Output of Converted Program (Handwritten)

```

c:\Users\parth\Documents\Visual Studio 2008\Projects\Knapsack\Debug\Knapsack.exe
Weight 0; Benefit: 0; Can't reach this exact weight.
Weight 1; Benefit: 0; Can't reach this exact weight.
Weight 2; Benefit: 0; Can't reach this exact weight.
Weight 3; Benefit: 0; Can't reach this exact weight.
Weight 4; Benefit: 7; To reach this weight I added object 3 <7$ 4Kg> to weight 0
Weight 5; Benefit: 7; To reach this weight I added object 3 <7$ 4Kg> to weight 1
Weight 6; Benefit: 10; To reach this weight I added object 2 <10$ 6Kg> to weight 0
Weight 7; Benefit: 10; To reach this weight I added object 2 <10$ 6Kg> to weight 1
Weight 8; Benefit: 16; To reach this weight I added object 1 <16$ 8Kg> to weight 0
Weight 9; Benefit: 16; To reach this weight I added object 1 <16$ 8Kg> to weight 1
Weight 10; Benefit: 17; To reach this weight I added object 2 <10$ 6Kg> to weight 4.
Added object 2 <10$ 6Kg>. Space left: 4
Added object 3 <7$ 4Kg>. Space left: 0
Total value added: 17$

```

Figure 10.4: Output of Handwritten Program

Chapter 11

Evaluation

Compilers are always tested with the help of standards. This standards provides bunch of programs. This programs are handwritten programs converted best optimized and parallelized. The program data are also provided in the benchmarks. We used parboil benchmark which is benchmark suite for CUDA programs. We converted below four benchmark programs to evaluate C2CUDATrnsalsator.

11.1 CUDA benchmarks for evaluating the C2CUDAranslator compiler

There are many parallelization benchmarks, but we choose parboil benchmark to evaluate the C2CUDATranslator because only Parboil benchmark is official CUDA Benchmark. Other benchmark are under process to be converted in CUDA. I converted below programs and run with the data given in parboil benchmark. The performance graph was generated according to the GFlops calculated in the program.

As we can see here we can do further optimizations like we can use CUDAMemcpyAsync and streams to optimize further. The benchmark programs highly optimized as they are handwritten. C2CUDATranslator Optimization phase needs to check additional barriers, use of `#pragma unroll` and CUDAMemcpyAsync with pinned memory.

Table I: CUDA benchmarks for evaluating the C2CUDAranslator compiler

Sr. No	Application	Description	
1	MM	Matrix Multiply	Computes the product of two matrices.
2	HISTO	Saturating Histogram	Computes a moderately large, 2-D saturating histogram with a maximum bin count of 255. Input datasets represent a silicon wafer validation application in which the input points are distributed in a roughly 2-D Gaussian pattern.
3	SAD	Sum of Absolute Differences	Sum of absolute differences kernel, used in MPEG video encoders. Based on the full-pixel motion estimation algorithm found in the JM reference H.264 video encoder.
4	FFT	Fast Fourier Transform	Computes the FFT of a large, 1-dimensional, real-valued array.

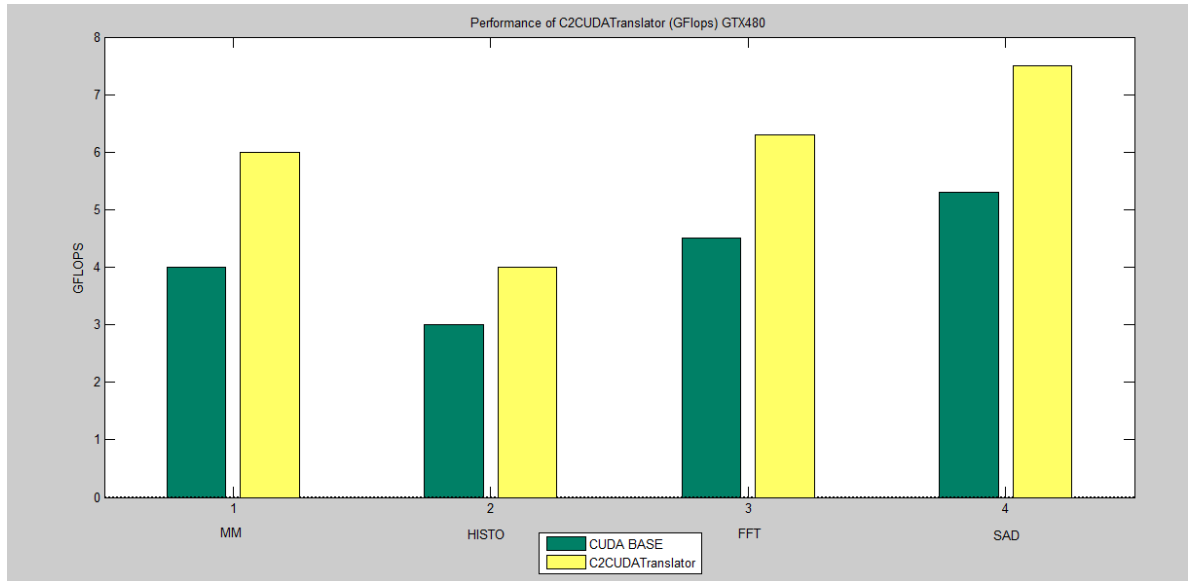


Figure 11.1: Evaluation of C2CUDAranslator

Chapter 12

Conclusion

12.1 Conclusion

Comparison between expected output and actual output:

- As we can see that the tool provides the same program in CUDA language and that is 100% working fine.
- The converted program gives the same output as the C Program.
- Language translation from C 2 CUDA C is working for given programs.

The output of the program converted by C2CUDATranslator is same as the program which converted manually. So we can say that the tool is working properly. The C2CUDATranslator saves 95% of the development /translation time. 5% is more optimizations humans can do further.

Limitations of C2CUDATranslator are:

- Supports ANSI C Only.
- Needs pre-processed input for spaces, comments etc. because we have to remove unsafe statements that my grammar cannot handle such as compiler commands.
- Some more algorithmic analysis needs to be implemented.

Chapter 13

Future Scope

13.1 Future Scope

All features could not be implemented due to time limitations. Some of the features will be implemented in next version of the C2CUDA Translator.

13.1.1 Upcoming Features

Table I: Upcoming Features

Sr. No	Features	Completed
1	Pre-processor	
	1 While pre processing	
	2 Printf pre processing	
	3 variable pre processing	
2	User Interface	

13.1.2 Pre-processor phase

In Pre-Processor phase I will try to implement a pre-processor that restricts users to use some of the compiler commands because my compiler is not ANSI C Compiler. So if suppose user will enter unsafe codes like `"#ifdef _cplusplus"` than my compiler will fail to perform the action that original ANSI C Compiler will do. And also some proper spacing and bracketing is important. So that we can understand the code. So I will try to make a C Pre-processor that will be inserted before the Scanner as shown in Compiler flow figure.

13.1.3 Parallelization/Analysis Phase

More algorithms can be added later on. The list of algorithms :

- a. Array Privatization
- b. Reduction Recognition
- c. Symbolic Expression Manipulators
 - Normalizes and simplifies expressions
 - Examples
 - $1+2*a+4-a$ $5+a$ (folding)
 - $a*(b+c)$ $a*b+a*c$ (distribution)
 - $(a*2)/(8*c)$ $a/(4*c)$ (division)

13.1.4 Compiler for OpenMP

One can also use this compiler for generating openMP code. All that is needed is to insert openMP pragmas before loops.

13.1.5 Compiler Learning

In C2CUDATranslator the modules are divided such that future work can be implemented. We can have neural schema design for supervised learning. The weights can be assigned to patterns and transformation functions. These weights will be updated each time the new program gets converted. Maybe as the time goes on compiler will be having enough set of patterns which is subset of all the patterns that can be used to generate every kind of big patterns. Additionally the compiler can decide which function to call for optimal parallelization. Suppose there are more than two ways to parallelize than which one is best. We also learn like this. The weights will be useful for decision.

Web References

- [15] <http://developer.download.nvidia.com>
- [16] <http://en.wikipedia.org/wiki/CUDA>
- [17] <http://www.nvidia.com>
- [18] <http://parlang.pbworks.com/>
- [19] <http://dl.acm.org/citation.cfm?doid=1995896.1995932>
- [20] http://moss.csc.ncsu.edu/~mueller/cluster/nvidia/2.0/nvcc_2.0.pdf
- [21] <http://wwwantlr.org/>
- [22] <https://sites.google.com/a/nirmauni.ac.in/cudacodes/ongoing-projects/automatic-conversion-of-source-code-for-c-to-cuda-c/review2/knapsack>
- [23] <http://developer.nvidia.com/cuda-libraries-sdk-code-samples>
- [24] <http://www.open64.net/>
- [25] <http://suif.stanford.edu/suif/>
- [26] <http://en.wikipedia.org/wiki/Knapsackproblem>
- [27] <http://en.wikipedia.org/wiki/Binarysearchalgorithm>
- [28] <http://en.wikipedia.org/wiki/Pi>
- [29] en.wikipedia.org/wiki/Convolution
- [30] <http://ait.iit.uni-miskolc.hu/dudas/Pluszok/AQTR.pdf>
- [31] <http://en.wikipedia.org/wiki/Banerjeetest>
- [32] <http://www.cs.colorado.edu/departments/publications/reports/docs/CU-CS-452-89.pdf>
- [33] <https://sites.google.com/a/nirmauni.ac.in/cudacodes/ongoing-projects/automatic-conversion-of-source-code-for-c-to-cuda-c/converted-programs>

References

- [1] Shane Ryoo, Sam S. Stone, "*Optimization principles and application performance evaluation of multithreaded GPU using CUDA*", Center for Reliable and high-performance Computing University of Illinois at Urbana-Champaign NVIDIA Corporation, 2009.
- [2] Setoain1, Christian Tenllado1, Manuel Arenaz, and Manuel Prieto1, "*Towards Automatic Code Generation for GPU architectures*", Computer Architecture Group, Department of Electronics and Systems, University of A Coruna, Spain.
- [3] "*Introducing multithreaded programming:POSIX Threads and NVIDIA's CUDA*".
- [4] B. R. Neha Patil, "*SFast and parallel implementation of image processing algorithm using cuda technology on gpu hardware*", ", tech. rep., Department of Electrical & Computer and Systems Engineering, Rensselaer Polytechnic Institute, Troy, NY 12180-3590.
- [5] D. L. N. Research, "*nvidia gpu architecture & implications*", NVIDIA Corporation 2007.
- [6] V. Rajaraman, C. Siva Ram Murthy, "Parallel Computers Architecture and Programming", Prentice Hall, 2000, ISBN-81-203-1621-5.
- [7] R. Kresch and N. Merhav, "*Fast DCT domain filtering using the DCT and the DST*", HPL Technical Report HPL-95-140, December 1995.
- [8] Shane Ryoo, Christopher I. Rodrigue, Sara S. Baghsorkhi, "*Optimizing the Fast Fourier Transform on a Multi-core Architecture*", 2006-2008.
- [9] David Kirk/NVIDIA and Wen-mei Hwu, "*CUDA Programming Model*", 2006-2008.
- [10] Halfhill, T.R., 2008, "*Parallel Processing With CUDA*", SMicroprocessor Report [Online] Available from: <http://www.MPRonline.com>.
- [11] N. P. Karunadasa & D. N. Ranasinghe, "*On the Comparative Performance of Parallel Algorithms on Small GPU/CUDA Clusters*", University of Colombo School of Computing, Sri Lanka, 2008.
- [12] Jeremy W. Nimmer and Michael D. Ernst, "*Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java*", MIT Lab for Computer Science 200 Technology Square Cambridge, MA 02139 USA..

- [13] Hyung Cook Kim and Edward J. Delp, "*A Comparison of Fixed point 2D 9X7 Discrete Wavelet Transform Implementations*", IEEE ICIP, vol 3, page 7803-7622, 2002.
- [14] Mint model, "*Mint: realizing CUDA performance in 3D stencil methods with annotated C*", [Online] Available from <http://www.sites.google.com/mintmodel>.

Index

Abstract, vi

algorithm, 37

algorithms, 71

Analysis, 54, 85, 95

ANTLR, 39, 44, 47, 94

architecture, 3

AST, 53

benchmark, 81

C2CUDAranslator, 81–83

C2CUDATanslator, 51, 69, 85, 91, 94

C2CUDATranslator, 47, 58, 67, 71, 95

CPU, 6

CUDA, 4, 6, 7, 9, 13, 14, 53

cudaHostAlloc, 19

cudaMallocHost, 67

cudaMemcpyAsync, 18, 67, 68

CUDAMemSet, 67

cudaStreamCreate, 18

Dependency, 53, 54

GPU, 1, 4–7, 15

Kernel, 49

kernel, 7, 15, 18, 19, 59, 68, 92

lexer, 43, 47

NVCC, 5

optimization, 13, 66

Parser, 43

Pattern, 51

pattern, 21, 35

Pattern Structure, 26

Appendix A

Output Screenshots

Please visit <https://sites.google.com/a/nirmauni.ac.in/cudacodes/ongoing-projects/automatic-conversion-of-source-code-for-c-to-cuda-c/converted-programs> [33] for more Output Screenshots.

Appendix B

User's Guide

C2CUDATranslator is very easy to use. Users are required to know C in advance before using C2CUDATranslator.

B.1 Getting Started

B.1.1 Input and Output of C2CUDATranslator

Input.file - input to translator (C file)

Output.file - output of the translator (CUDA file)

First, copy C program to input.file and run the translator. After running the translator copy CUDA code that has been converted or copy content of output.file. Put them in CUDA project. Run the project.

B.2 C2CUDATranslator input Details

B.2.1 Kernel Outlining

The kernel is the code to be ported on the GPU. So, first identify the code you want to run on GPU. It may be some computation code such as loops. Now, write "`#pragma kernel_start`" before that code and at the end of the code write "`#pragma kernel_end`".

Example

```
#pragma kernel_start
for (i = 1; i <= 100; ++i)
{
  for (j = 0; j < 100; ++j)
  {
    a[i][j] = b[i][j] + c[i][j];
  }
}
#pragma kernel_end
```

B.2.2 Kernel Variables

Kernel variables are the variables which are copied to/from host/devices. User can specify them in their code by writing "kernel" in the declaration of that variable.

Example

```
int a[100][100];
int b[100][100];
int c[100][100];
```

a,b & c arrays are used in kernel. So, we will write them as

```
kernel int a[100][100];
kernel int b[100][100];
kernel int c[100][100];
```

B.2.3 Kernel local Variables

If there are some local variables in the kernel code, they does not need to be copied to/from host/device. They can be initialized in GPU. We can write "local kernel" before the declaration statements of them.

Example

```
int n=3;
```

```
int P=100;
```

n & P are local variables used in kernel region. So, we will write them as

```
local kernel int n=3; local kernel int P=100;
```

Appendix C

Developer's Guide

C2CUDATranslator is very easy to work with. Developers are required to know C, CUDA C in detail in advance before using C2CUDATranslator. Additionally they are required to know ANTLR compulsory.

C.1 Getting Started

C2CUDATranslaor is like a framework to develop and test various code analysis algorithms in the field of source code to source code translation.

C.1.1 C2CUDATranslator Project Structure

The design of the project is shown below. Developers can see the Visual 2008 Solution project as below design.

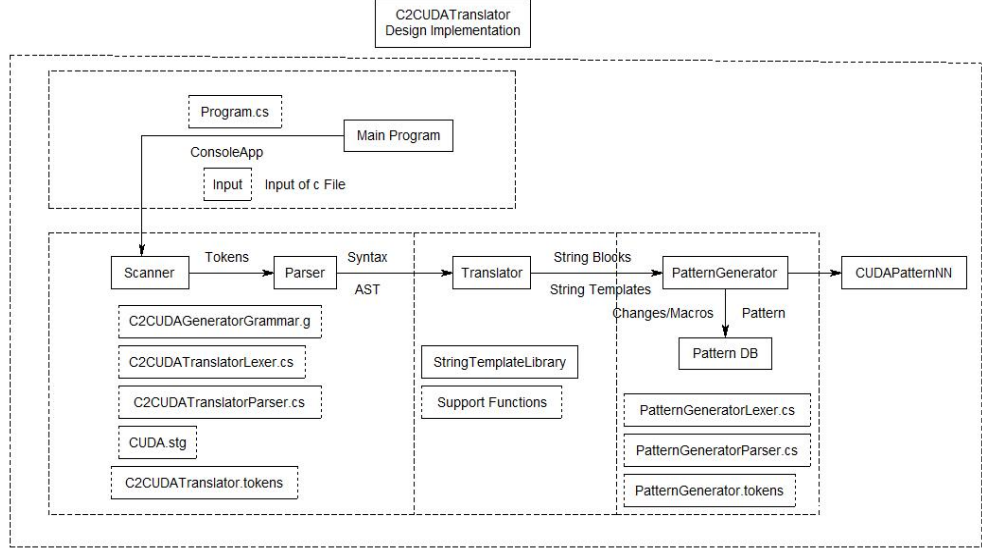


Figure C.1: C2CUDATranslator Project Structure

C.2 C2CUDATranslator development Details

C.2.1 Analysis Framework

In C2CUDATranslator there is a file "Analysis.cs" which contains analysis algorithms. One can write his/her own algorithm in this file as a class or new function and can call in the parser. But the namespace must be C2CUDATranslator.Analysis.

C.2.2 Translation Framework

Similarly, C2CUDATranslator.Translation is the framework that contains classes and codes for translation. Developers will find FOR, BLOCK etc. classes. They may contain properties like LOOP.index, LOOP.IsNestLoop etc.

C.2.3 Use framework

Developers can use collections those contains readonly, write-only variables and can use them in conditions in making analysis algorithms.