CRINK: Automatic CUDA code generation for affine C programs

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Master of Technology

by
Akanksha Singh
Roll Number : 12111008

under the guidance of
Dr. Amey Karkare and Dr. Sanjeev Kumar Aggarwal

Department of Computer Science and Engineering
Indian Institute of Technology Kanpur
June, 2014

CERTIFICATE

It is certified that the work contained in this thesis entitled
"*CRINK: Automatic C to CUDA code generation for affine programs.*",
by *Akanksha Singh(Roll Number 12111008)*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Amey Karkare 5/6/2014

(Dr. Amey Karkare)
Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur
Kanpur-208016

June, 2014

Abstract

Parallel programming has largely evolved as an efficient solution to a large number of compute intensive applications. Graphics Processing Unit (GPUs), traditionally designed to process computer graphics, are now widely applied to process large chunks of data parallely in many computationally expensive applications. While developing parallel programs to run on parallel computing platforms, such as CUDA, OpenCL, etc. requires knowledge of platform-specific concepts, it becomes very convenient if the process of parallelizing compute intensive sections of the program can be automated.

We develop a tool CRINK, an end-to-end code transformation system, to convert sequential C programs to their parallel counterparts in CUDA. CRINK targets to parallelize the expensive sections (sections within loops) of the program while converting C programs to CUDA C programs. It incorporates handling of both irregular and regular kernels. We use concepts of Cycle Shrinking and Extended Cycle Shrinking for parallelism extractions and loop transformations.

To analyse the performance, we run CRINK over the expensive sections taken from ZERO_RC, SPEC, SANDIA_RULES, Treepack and Higbie standard benchmarks. Analysis is done over 66 varied configurations of the benchmarks and datasets where we observe that drastic drops in computation times are achieved as the number of threads are increased while execution of the code transformed by CRINK.

Dedicated to

my family, my advisor and my friends.

Acknowledgement

I would like to express my sincere gratitude to both my thesis supervisors, Dr. Amey Karkare and Late Dr. Sanjeev Kumar Aggarwal. Sanjeev sir's guidance and support helped me develop proper understanding of the subject. His guidance helped me to approach the problem through the right direction during the initial stages of my thesis work. Though I started working under Amey sir in the middle of my thesis duration, he was always very motivating and helpful. His guidance and encouragements enabled me to bring this thesis to a successful completion. I express my sincere thanks and respect to both of them.

I am also thankful to all my friends of Computer Science Department, for their continuous support, both technical and emotional, during my stay at IIT Kanpur. I am grateful to all of them for creating memories that are really unforgettable.

I am deeply and forever indebted to my family for their encouragement, support and love throughout my life. It is their well wishes which always gave me inspiration and courage to face all challenges and made my path easier.

Akanksha Singh

Contents

Abstract **ii**

List of Figures **ix**

List of Algorithms **x**

List of Tables **xi**

1 Introduction **1**

1.1 Objective 2

1.2 Motivation of Work 2

1.3 Outline of the Solution 3

1.4 Contribution of the Thesis 4

1.5 Thesis Organization 4

2 Background **6**

2.1 Compute Unified Device Architecture(CUDA) 6

2.2 Affine and Non-Affine Programs 10

2.3 Data Dependence 11

2.4 Data Dependence Test 15

2.5 Cycle Shrinking 18

2.6 Extended Cycle Shrinking 26

3 Code Generation Framework **32**

3.1 Compilation Phase 32

3.2 Dependence Test 36
3.3 Parallelism Extraction 38
3.4 Code Generation 41

4 A tour to CRINK **45**
4.1 Singly Nested Loops 45
4.2 Multi-Nested Loops with Constant Dependence Distance 54
4.3 Multi-Nested Loops with Variable Dependence Distance 68

5 Literature Survey **75**
5.1 Automatic Parallelization 75
5.2 Automatic Code Generation for GPGPUs 77

6 Performance Evaluation **79**
6.1 Experimental Setup 79
6.2 Testing Setup 80
6.3 Standard Datasets 80
6.4 Performance Analysis 81
6.5 Results Analysis 95

7 Conclusion and Future Work **96**

Bibliography **98**

List of Figures

1.1 An abstract view of CRINK 3

2.1 CUDA Programming Model 7
2.2 CUDA Memory Model 9
2.3 Data Dependence in Loops 13
2.4 Different Partition for Example 7 27

3.1 Compilation Phase 33
3.2 Dependence Testing 37
3.3 Parallelism Extraction 39
3.4 Code Generation 41

4.1 Upper and Lower Bound Constraints 56

6.1 Plot for CSphd 83
6.2 Plot for EVA 83
6.3 Plot for barth5 84
6.4 Plot for p2p-Gnutella04 84
6.5 Plot for HEP-th-new 84
6.6 Plot for NotreDame_www 84
6.7 Plot for CSphd 84
6.8 Plot for EVA 84
6.9 Plot for barth5 85
6.10 Plot for p2p-Gnutella04 85

6.11 Plot for HEP-th-new 85
6.12 Plot for NotreDame_www 85
6.13 Plot for CSphd 85
6.14 Plot for EVA 85
6.15 Plot for barth5 86
6.16 Plot for p2p-Gnutella04 86
6.17 Plot for HEP-th-new 86
6.18 Plot for CSphd 86
6.19 Plot for EVA 86
6.20 Plot for barth5 87
6.21 Plot for p2p-Gnutella04 87
6.22 Plot for HEP-th-new 87
6.23 Plot for NotreDame_www 87
6.24 Plot for CSphd 87
6.25 Plot for EVA 87
6.26 Plot for barth5 88
6.27 Plot for p2p-Gnutella04 88
6.28 Plot for HEP-th-new 88
6.29 Plot for NotreDame_www 88
6.30 Plot for CSphd 88
6.31 Plot for EVA 88
6.32 Plot for barth5 89
6.33 Plot for p2p-Gnutella04 89
6.34 Plot for HEP-th-new 89
6.35 Plot for NotreDame_www 89
6.36 Plot for CSphd 89
6.37 Plot for EVA 89
6.38 Plot for barth5 90
6.39 Plot for p2p-Gnutella04 90

6.40 Plot for HEP-th-new 90
6.41 Plot for NotreDame_www 90
6.42 Plot for CSphd 91
6.43 Plot for EVA 91
6.44 Plot for barth5 91
6.45 Plot for p2p-Gnutella04 91
6.46 Plot for HEP-th-new 91
6.47 Plot for NotreDame_www 91
6.48 Plot for CSphd 92
6.49 Plot for EVA 92
6.50 Plot for barth5 92
6.51 Plot for p2p-Gnutella04 92
6.52 Plot for HEP-th-new 92
6.53 Plot for NotreDame_www 92
6.54 Plot for CSphd 93
6.55 Plot for EVA 93
6.56 Plot for barth5 93
6.57 Plot for p2p-Gnutella04 93
6.58 Plot for HEP-th-new 93
6.59 Plot for NotreDame_www 93
6.60 Plot for CSphd 94
6.61 Plot for EVA 94
6.62 Plot for barth5 94
6.63 Plot for p2p-Gnutella04 94
6.64 Plot for HEP-th-new 94
6.65 Plot for NotreDame_www 94

List of Algorithms

1 Cycle Shrinking for Simple Loops 21

2 Simple Shrinking 22

3 Selective Shrinking 23

4 True Dependence Shrinking 25

5 Extended Cycle Shrinking for Constant Dependence Distance 28

6 Extended Cycle Shrinking for Variable Dependence Distance 30

List of Tables

6.1 Standard benchmarks used 81

6.2 Performance analysis for different configuration of standard benchmarks and datasets 83

Chapter 1

Introduction

Graphics Processing unit (GPU) was traditionally designed for 3D graphics application but because of increasing demand for high performance computing (HPC), GPUs are now efficiently used in preference to CPUs for most of the computationally expensive applications. GPUs and CPUs are different from each other in the way they process data. CPUs have fewer cores to process the task sequentially while GPUs have thousands of cores that can process the task parallely. *GPU accelerated computing* takes the benefits of both GPU and CPU. The compute intensive part of a program is made to run on the GPU while the remaining portion keeps executing on CPU. Nowadays, GPUs are widely used in embedded systems, Mobile phones, Personal Computers etc.

Compute Unified Device Architecture(CUDA)[Cud12] is a parallel programming model which is created by Nvidia and implemented to interact with GPUs. CUDA provides software environment to program developers for parallel computing. Using CUDA, GPUs are made available to be used for general processing, commonly known as General purpose computing on Graphics Processing Unit (GPGPU). CUDA is available to software developers through CUDA accelerated libraries and the extensions to some basic programming languages like C, C++, Fortran etc.

CUDA provides a multi-threaded model for general purpose computation on GPUs. Manual implementation of parallel code using CUDA is still more complex than parallel programming model like OpenMP, MPI etc as it involves better learn-

ing and understanding of CUDA architecture. The issue increases the need of some tool that could convert sequential programs into their parallel counterparts. In this thesis, we propose CRINK , an end-to-end code transformation tool. CRINK provides compiler support to facilitate the automatic code generation of parallel CUDA programs from given input sequential programs.

We test and analyze the performance of CRINK over a number of standard benchmarks, the results of which have been detailed in Chapter 6

1.1 Objective

The objective of this thesis is to create a tool that can automatically convert an input C program into CUDA C program, which can then be compiled using nvcc (Nvidia LLVM-based C/C++ compiler). To accomplish this, we needed to transform the compute intensive portions of the program so as to be able to run them parallely, thereby converting the programs to CUDA C programs. The goal of the thesis is to achieve reductions in computation times of applications by transforming them to CUDA C programs and being able to leverage the benefits of their parallel execution.

1.2 Motivation of Work

There are significant number of applications that contain regular or irregular kernels. They may consist of a number of iterations which, if executed sequentially, would become bottleneck to their computation times. Executing these iterations on GPU reduces the execution time by providing highly parallel multi-core processing system which processes large blocks of data efficiently. CUDA provides a platform to execute these iterations parallely with the help of various threads that execute the instruction in SIMD fashion. However, these kernels might contain some inter loop dependencies — dependencies which exist across loop iterations. Parallel execution of such kernels without taking care of such inter loop dependencies may lead to incorrect results. Example 1 illustrates inter loop dependencies.

Example 1.
```
for(i=3;i<N1;i++)
for(j=4;j<N2;j++){
  x[i][j]=y[i-3][j-4];
  y[i][j]=x[i-2][j-3];
}
```

Example 1 gives a simple case, where, for array variable x, iteration (i, j) depends upon the iteration $(i-2, j-3)$ and for array variable y, iteration (i, j) depends upon the iteration $(i-3, j-4)$. However, in practice, a large number of application programs contain much more complex dependencies across loop iterations.

Therefore, we needed special mechanisms to handle inter loop dependencies while parallelization. We present a technique that partitions dependent iterations into groups of independent iterations, such that each independent group can be made to execute parallely on GPUs.

1.3 Outline of the Solution

Automatic C to CUDA conversion system takes sequential C programs as input and performs the following phases one by one as outlined in the Figure 1.1. Figure 1.1 gives an abstract view of the whole tool.

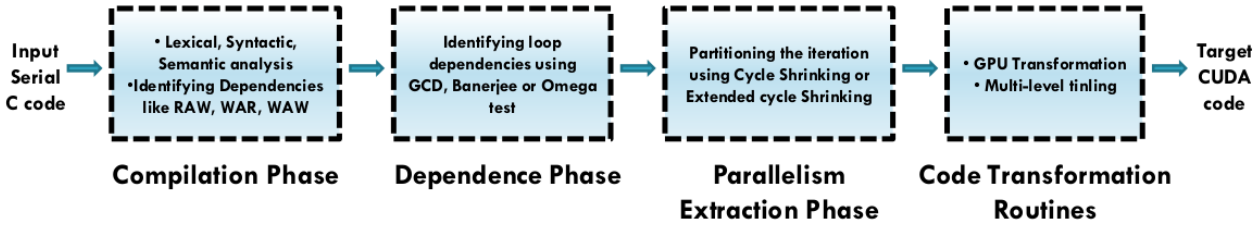


Figure 1.1: An Abstract View of CRINK .

The tool, largely consists of following phases:

- **Compilation phase:** All the lexical, syntactic and semantic analysis are carried out in this phase.

- **Dependence testing phase:** This phase checks whether dependencies exist within loops using gcd[PK99], banerjee[AK04] or omega[KMP$^+$96] test.
- **Parallelism extraction phase:** This phase uses Cycle Shrinking [Pol88] or Extended cycle shrinking[SBS95] technique to partition the loop iterations into the group of independent iterations that can execute parallely on GPU.
- **Code Generation phase:** This phase generates the CUDA C code based on the information extracted from the third phase.

The output code is compiled using the compiler which supports CUDA architecture and parallelization is carried out at run time.

1.4 Contribution of the Thesis

In this thesis, we proposed and implemented a tool which provides automatic transformation of C programs into paralllel CUDA programs using partition based approach for parallelism extraction. The input program can contains an affine or a non-affine kernel. We have tested the tool over various standard benchmarks using standard datasets and observes that the computation time of the generated parallel program reduces as the number of threads increases while its execution.

1.5 Thesis Organization

Rest of the thesis is organized as follows:

Chapter 2, Background, describes the background information required to understand the tool like CUDA, affine and non-affine programs, dependence test, cycle shrinking.

Chapter 3, C to Cuda code generation Framework, discusses the various stages of the tool like compilation phase, dependence testing, parallelization extraction and code generation routines in detail.

Chapter 4, Methodology, gives the overview of the techniques and algorithms used

in the thesis. It also describes the implementation details of the tool.

Chapter 5, Experiments and Results, discusses the various experiments done on the benchmarks using standard datasets.

Chapter 6, Literature Survey, discusses the literature survey done during the course of the thesis work.

Chapter 7, Conclusion and Future work, concludes the work of the thesis and the scope of future work.

Chapter 2

Background

This chapter provides an overview of the concepts required to understand this thesis. The definitions and notations required to understand the theoretical aspects of this dissertation are entirely covered in this chapter. Detailed background can be found in [AK04, WSO95, Pol88, SBS95, SK10, KMP+96].

2.1 Compute Unified Device Architecture(CUDA)

CUDA[Cud12, SK10] a is a parallel computing architecture developed by Nvidia. It provides a platform to perform parallel computation on Graphics accelerator and thereby enables considerable increase in computing performance by harnessing the power of GPUs. CUDA architecture exposes the general purpose computation on GPUs as their primary capability. CRINK goal is to generate CUDA C program that takes less time as compared to the traditional sequential program. CUDA C is an extension to the industry standard C language to allow heterogeneous programs. CUDA with standard language C consists of:
host code which executes by a single thread, and
kernel code executed by multiple parallel threads.

2.1.1 CUDA Programming Model

A CUDA program consists of different portions that are executed on CPU and on GPU. Kernel is launched and its code is executed on GPU device by multiple threads. Each kernel is executed as Grid of Thread Blocks. A Thread Block is a collection of threads that cooperate by synchronizing their execution and by efficiently accessing the respective block memory. Every thread is free to take any unique path and cannot communicate with the threads from the other blocks.

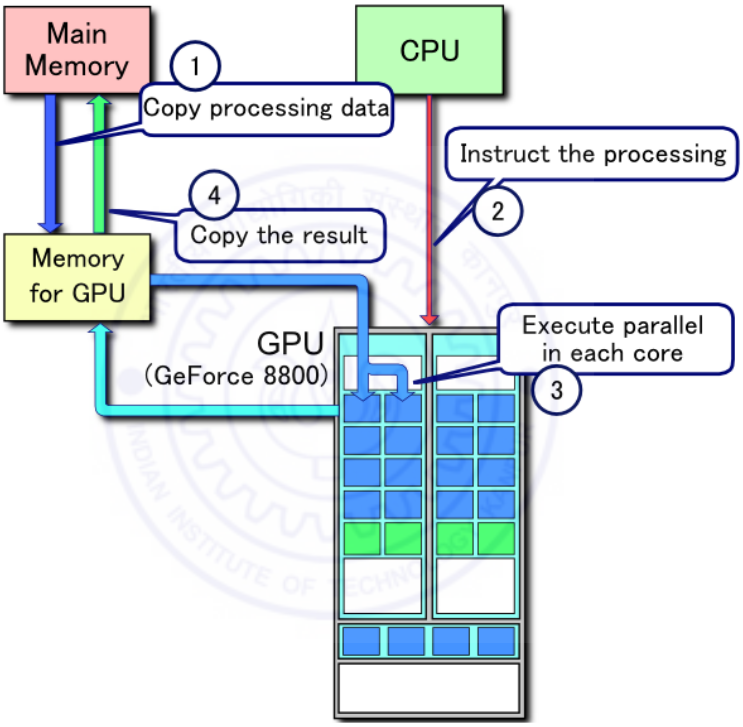


Figure 2.1: CUDA Programming Model[cud].

A grid and block can be of one, two or three dimension and each of its element is a block and thread respectively. Following are the built in CUDA variables:

girdDim.x,y,z gives the number of blocks in each direction

blockDim.x,y,z gives the number of threads in each direction

blockIdx.x,y,z gives the block index within a grid

threadIdx.x,y,z gives the thread index within a block

These variables are of dim3(integer vector) type. If a variable is declared as dim3

type, then any component unspecified has default value 1.

2.1.2 CUDA Memory Model

CUDA threads can access different memories during their execution as described in Figure 2.2. Each thread has its own local memory and each block has its own shared memory. There is a global memory from which every thread can access the data. There are two other read-only memories, i.e. constant memory and texture memory. CUDA architecture provides below listed memories:

- *Local memory* of each thread is allowed to be accessed by that particular thread only. Each thread is executed by streamline processor. Therefore, local memory can be accessed by only one streaming processor. Local memory is present on the device, therefore it has high latency and low bandwidth.

- *Shared memory* is shared among the threads within a block. Each block is executed on a multiprocessor. Therefore, shared memory can be accessed by all streaming processors within a multiprocessor. Shared memory is present on the chip hence it is faster then the local and global memoy.

- *Global memory* is the largest memory in terms of volume. It is present on the device, and hence, has high bandwidth. Global memory can be accessed by any thread within any block.

- *Constant memory* is a read only memory. It resides on device memory. Constant memory is used for the data that will not change over the course of kernel execution.

- *Texture memory:* Just like constant memory, texture memory is also a read only memory that provides good performance, higher bandwidth only when data access have certain patterns.

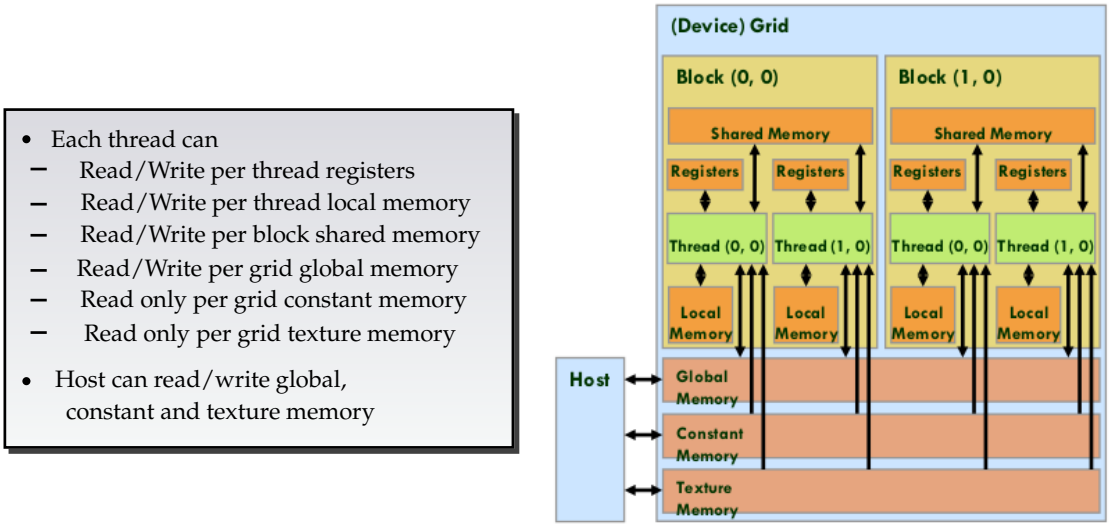


Figure 2.2: CUDA Memory Model

2.1.3 CUDA C programming Syntax

Following are the CUDA APIs and CUDA kernel syntax that will be used in this thesis work:

- **cudaMalloc()** is used to allocate memory on device global memory. It requires two parameters: address of a pointer to the allocated object and size of the object. For example, `cudaMalloc ((void **)devPtr, size_t size)` allocates memory to devPtr on device of size `size`.
- **cudaMemCpy()** is used for memory data transfer. It requires four parameters:

  1. Pointer to source
  2. Pointer to Destination
  3. Size of the data to be transfer
  4. Type of transfer i.e. Host to Host, Host to Device, Device to Device, Device to Device

  For example, `cudaMemcpy (void *dst, const void *src, size_t count, cudaMemcpyHostToDevice)` copies source `src` from host to destination `dst` on device.

- **cudaFree()** frees object from device global memory. For example, `cudaFree (void *devPtr)` frees the memory allocated to `devPtr` on device.

- Threads can be synchronized using `cudaThreadSynchronize()`.
- CUDA kernel is defined as `__type qualifier__ functionName(Parameters) Kernel Body`. Unlike C functions, CUDA kernel is executed by various threads parallely.
- CUDA kernel is called from host code as `functionName<<<noOfBlock, noOfThread>>>(Parameters)`. This specifies the number of threads that execute the kernel. A grid can contain maximum 65535 blocks and a block can contain maximum 512 threads.

2.2 Affine and Non-Affine Programs

Affine loops are the loops in which referenced array subscripts and loop bounds are a linear function of the loop index variables. The memory access patterns within the statements contained in the affine loops, can thus, always be known at compile time. The programs containing affine loops are called affine or regular programs.

Example 2.
```
for(i=4;i<50;i++)
    a[i+6]=a[i-2];
```

Here, access pattern of array variable a, is a linear function of loop index. In contrast to affine loops, non-affine loops are the ones where the memory access patterns cannot be determined at compile time, since the array subscript expressions and loop bounds do not form linear functions of the loop index variables. The programs containing non-affine loops are called non-affine or irregular programs. For example,

Example 3.
```
for(i=2;i<20;i++)
    a[b[i]]=a[c[i]]+a[d[i]];
```

Here, subscript of array variable a, depends on the value of i_{th} iteration of array `b,c,d`, which can not be known at compile time. Therefore, it is a non-affine loop.

2.3 Data Dependence

A data dependency is a situation in which the data used by a program statement (instruction) depends upon the data created by other statement. The technique used to discover data dependencies among statements (or instructions) is called dependence analysis.

Definition 1. There is a data dependence from statement S_1 to statement S_2 (statement S_2 depends on statement S_1) if and only if[AK04] :

1. both statements access the same memory location and at least one of them stores into it and

2. there is a feasible run-time execution path from S_1 to S_2

There are three types of data dependencies:

1. **Flow Dependence** occurs when a variable is assigned in one statement (say S_1) and is used in subsequently executed statement (say S_2). It is denoted by $S_1\delta_f S_2$.

2. **Anti Dependence** occurs when a variable is used in one statement (say S_1) and is reassigned in subsequently executed statement (say S_2). It is denoted by $S_1\delta_a S_2$.

3. **Output Dependence** occurs when a variable is assigned in one statement (say S_1) and is reassigned in subsequently executed statement (say S_2). It is denoted by $S_1\delta_o S_2$.

In this thesis, we deal with nested loops. The following definitions and notations are required to understand the theoretical aspects of data dependence:

- Consider a m-perfectly nested loop with $(I_1, I_2, .., I_m)$ denoting the respective loop indices. Since every loop is normalized, 0 and $(N_1, N_2, .., N_m)$ are the lower and upper bounds of perfectly nested loops.

- $S_i(I_1, I_2, .., I_m)$ denotes the i_{th} indexed statement surrounded by m perfectly nested loops, where I_j is the index of j^{th} loop.
- $S_i < S_j$ implies that statement S_i will execute before statement S_j.
- $S_i(i_1, i_2, .., i_m)$ represents an instance of statement of S_i.
- $W(S_i(i_1, i_2, .., i_m))$ and $R(S_i(i_1, i_2, .., i_m))$ gives the set of variables that are written and read respectively in S_i.
- If statements S_i and S_j are involved in dependence, such that $S_i < S_j$, and the data used or created in S_i is recreated or used in S_j respectively, then the statement S_i is said to be *dependence source* and S_j is said to be *dependence sink*.
- Two statements $S_i(I_1, I_2, .., I_m)$ and $S_j(I_1, I_2, .., I_m)$ are said to be involved in a dependence $S_i \delta S_j$ only if there exists an instance $S_i(i_1, i_2, .., i_m)$ and $S_j(i_1, i_2, .., i_m)$, such that, $S_i(i_1, i_2, .., i_m) < S_j(i_1, i_2, .., i_m)$ becomes true and following exists:
  1. Flow dependence: $W(S_i(i_1, i_2, .., i_m)) \bigcap R(S_j(i_1, i_2, .., i_n)) \neq \phi$
  2. Anti dependence: $R(S_i(i_1, i_2, .., i_m)) \bigcap W(S_j(i_1, i_2, .., i_m)) \neq \phi$
  3. output dependence: $W(S_i(i_1, i_2, .., i_m)) \bigcap W(S_i(i_1, i_2, .., i_m)) \neq \phi$ where $S_i(I_1, I_2, .., I_m)$ and $S_j(I_1, I_2, .., I_m)$ are the dependence source and sink respectively.

2.3.1 Data Dependence in Loops

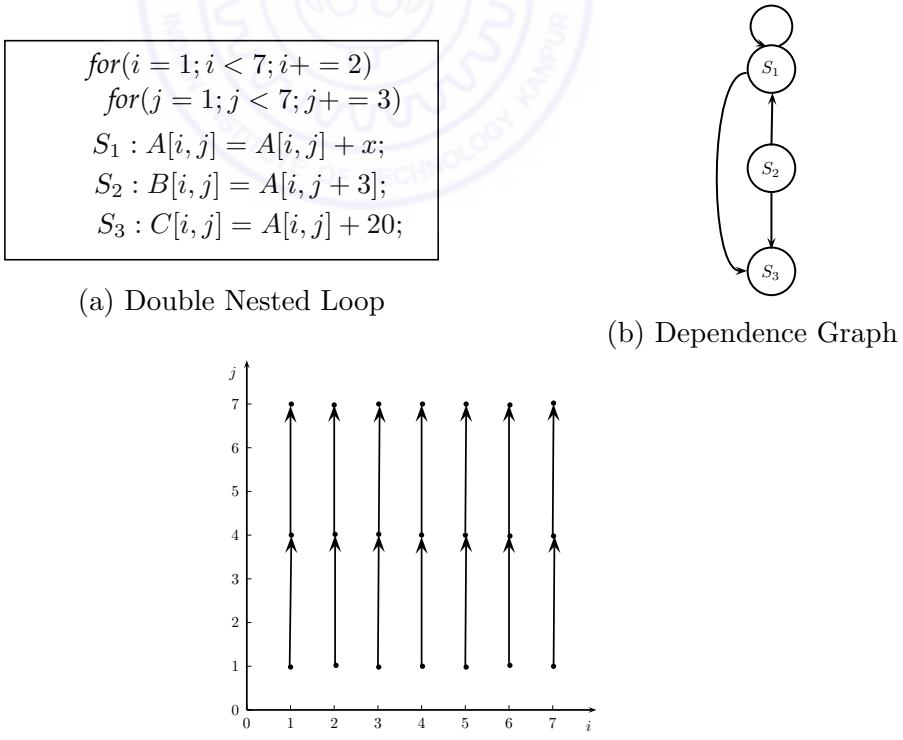
The statements within loops are executed multiple times leading to dependence flows from one instance of executing statement to an instance of other statement or the same statement. To represent dependence flows, *Dependence Graphs* [WSO95, AK04] are used.

Definition 2. Dependence Graphs are directed graphs that represent dependencies between statements within a loop, where vertices of the graph are the various statements within the loops and an edge among two vertices represent the presence of dependence between them.

Figure 2.3b represents dependence graph corresponding to the loop shown in figure 2.3a.

2.3.2 Iteration Space

The Iteration space of a loop provides the information regarding flow of the dependence. It contains one point from every iteration. To represent the dependence of any statement in one iteration to any statement in other iteration, an arrow is marked from the source iteration to the sink iteration. This creates an *iteration space dependence graph*. Figure 2.3c gives the iteration space for the example shown in figure 2.3a.



(a) Double Nested Loop

(b) Dependence Graph

(c) Iteration Space

Figure 2.3: Data Dependence in Loops

2.3.3 Iteration Vector

One obvious way to label the iteration space is to use index variable iteration vectors i.e., using the values of the loop index variables as the iteration vector. If we consider a m-nested loop with loop indices $(I_1, I_2, .., I_m)$, then iteration vector i for a particular iteration can be given as:

$$i = \begin{pmatrix} i_1 \\ i_2 \\ . \\ . \\ . \\ i_m \end{pmatrix}$$

where $i_k,\ 1 \leq k \leq m$ represents the iteration number for the loop at nesting level k.

2.3.4 Dependence Distance

Dependence distance is the difference between source and target iteration vectors of a dependence relation. The dependence distance is itself a vector denoted by d as:

$$d = i^T - i^S \tag{2.1}$$

where i^T and i^S are the target and source iteration vectors of a dependence relation. An instance of statement p with iteration vector is represented by $S_p[i]$. If $S_q[i + d]$ depends on $S_p[i]$, then the dependence relation can be denoted as:

$$S_p \delta^*_{(d)} S_q \tag{2.2}$$

2.3.5 Direction Vector

The direction vector of a dependence relates the source and target iteration vectors. If d is the dependence distance between statement S_1 of i_{th} iteration and statement

S_2 of j_{th} iteration, then direction vector $D(i,j)$ can be defined as:

$$D(i,j) = \begin{cases} " < " & if d(i,j)_k > 0 \\ " = " & if d(i,j)_k = 0 \\ " > " & if d(i,j)_k < 0 \end{cases} \tag{2.3}$$

2.4 Data Dependence Test

Data dependence analysis is an essential part of compiler optimization. It tells the compiler about the fragments of code which are independent (and hence can execute in parallel) and the fragments of code which contain dependencies. A lot of dependence tests are proposed in data dependence literature, out of which this thesis uses GCD test[PK99], Banerjee's test[AK04] and Omega test[KMP+96]. In all these tests, there are trade-offs between accuracy and efficiency. Thus, these data dependence tests always approximate results on conservative side i.e., a dependence can exist if independence can not be proved. Data dependence testing is equivalent to integer linear programming and therefore can not be solved generally.

2.4.1 GCD Test

GCD test is the most basic dependence test is also used as an initial step in many tests. GCD test is based on a theorem of elementary number theory which states that there exists an integer solution for a linear equation if greatest common divisor(GCD) of coefficient on left hand side evenly divides the constant term at right hand side. If this condition does not hold, it means that there is no integer solution for the linear equation and hence dependence does not exist. For Example 2, the linear equation will be:

$$ai_1 + bi_2 = (d - c) \tag{2.4}$$

$$i_1 + i_2 = (6 - 2)$$

In equation 2.4, a, b are the coefficients of loop variables and c, d are the constant terms in the array subscript. Therefore, an integer solution will exist if $GCD(a,b)$

evenly divides *(d-c)*. So from lexical and syntactic analysis, we will be able to get the read and write dependency variable and will extract the value of a, b, c and d. If *GCD(a,b)* divides *(d-c)* then the dependency exists, otherwise not.

2.4.2 Banerjee Test

The Banerjee test is based on the Intermediate Value Theorem. The test calculates the minimum and maximum value that the left hand side of linear equation 2.7 can achieve. If the constant term $(B_0 - A_0)$ of equation 2.7 does not fall between these extreme values, then no dependence exists, otherwise a real solution to the linear equation exists and hence, like GCD it will also return the maybe answer.

GCD Test is inefficient because most common GCD is 1 which divides everything. Also, GCD indicates dependence even when the solution for dependence equation exists outside the loop limits. Therefore, Banerjee Test tries to find the solution of dependence equation under the constraints of direction vector and loop limits.

Let a be a real number. The positive part of a, denoted by a^+, is given by following expression:

$$a^+ = if\ a >= 0\ then\ a\ else\ 0 \tag{2.5}$$

The negative part of a, denoted by a^-, is given by following expression:

$$a^- = if\ a >= 0\ then\ 0\ else\ -a \tag{2.6}$$

The dependence equation of a statement present inside d nested loops is given as:

$$\sum_{k=1}^{d}(A_k I_k\ -\ B_k I_k) = B_0\ -\ A_0 \tag{2.7}$$

For each k, find the lower and upper bounds such that:

$$LB_k^{\psi_k} \leq (A_k I_k\ -\ B_k I_k \leq UB_k^{\psi_k} \tag{2.8}$$

where $LB_k^{\psi_k}$ is the direction vector. After taking summation we get:

$$\sum_{k=1}^{d} LB_k^{\psi_k} \;\leq\; \sum_{k=1}^{d}(A_k I_k \;-\; B_k I_k) \;\leq\; \sum_{k=1}^{d} UB_k^{\psi_k} \tag{2.9}$$

$$or,\; \sum_{k=1}^{d} LB_k^{\psi_k} \;\leq\; B_0 - A_0 \;\leq\; \sum_{k=1}^{d} UB_k^{\psi_k} \tag{2.10}$$

If either $\sum_{k=1}^{d} LB_k^{\psi_k} \;>\; B_0 - A_0$ or $\sum_{k=1}^{d} UB_k^{\psi_k} \;<\; B_0 - A_0$ is true, then the solution does not exist within the loop constraints and therefore dependency may not exist.

Following are the equations required for calculating the lower and upper bounds:

$$\begin{aligned}
LB_i^{<} &= -(a_i^- + b_i)^+(U_i - 1) + [(a_i^- + b_i)^- + a_i^+]L_i - b_i \\
UB_i^{<} &= (a_i^+ - b_i)^+(U_i - 1) - [(a_i^+ - b_i)^- + a_i^-]L_i - b_i \\
LB_i^{=} &= -(a_i - b_i)^- U_i + (a_i - b_i)^+ L_i \\
UB_i^{=} &= (a_i - b_i)^+ U_i - (a_i - b_i)^- L_i \\
LB_i^{>} &= -(a_i - b_i^+)^-(U_i - 1) + [(a_i - b_i^+)^+ L_i + a_i \\
UB_i^{>} &= (a_i - b_i^-)^+(U_i - 1) - [(a_i - b_i^-)^- L_i + a_i
\end{aligned}$$

where L_i and U_i are the lower and upper bounds of i^{th} loop.

2.4.3 Omega Test

The Omega test is based on a combination of the least remainder algorithm and Fourier-Motzkin variable elimination (FMVE) [DCE73]. Omega test begins with a derivation of Knuth's least remainder algorithm that converts the linear equalities and inequalities into the linear inequalities. An extension to standard FMVE is used to determine if the resulting system of linear inequalities has an integer solution or not.

In Omega calculator, a text based interface is given as input to Omega library which manipulates integer tuple relations and set such as:

$$\{[i,j] \rightarrow [j,j'] : 1 \leq i < j < j' \leq n\} \; and \; \{[i,j] : 1 \leq i < j \leq n\}$$

For Example 2, input file for Omega calculator will be:

$T := [i_1, i_2] : 4 <= i_1, i_2 <= 50 \;\&\&\; i_1 + 6 = i_2 - 2;$

$T;$

and the corresponding output will be:

Omega Calculator $v2.1$ (*based on Omega Library* $2.1,\ July,\ 2008$) :

$T := [i_1, i_2] : 4 <= i_1, i_2 <= 50 \;\&\&\; i_1 + 6 = i_2 - 2;$

#

$\#T;$

$[i_1, i_2 + 8] : 0 <= i_1 <= 38$

#

2.5 Cycle Shrinking

Cycle shrinking [Pol88] is a compiler transformation which is used to parallelize perfectly nested loops in which array access functions in the loop ststement are affine function of loop indices and global parameters. It performs certain transformations that parallelize these loops. These transformations use the data dependence graph of the loop to identify whether the existing dependency allows the loop to be parallelized without violating their semantics. If dependency does not exists, then the transformation is quite simple. In some cases when loops are complex, certain tests have to be performed to determine whether the pattern of dependence allows loop parallelization. Loops whose dependence graph do not form a strongly connected component, can be fully or partially parallelizable. If the dependence graph contains a cycle, then node splitting is performed to break the cycle, assuming atleast one of the dependence is anti dependence.

Cycle Shrinking is used to extract parallelism that may be present in perfectly nested loops. It is useful in cases where the dependence cycle of loop involves dependencies with distances greater than 1. It transforms a serial loop into two perfectly nested loops: an outermost serial and innermost parallel loop.

2.5.1 Characteriztion of Reduction Factor

Cycle shrinking parallelizes the loop by partitioning the loop into groups of iterations that are independent to each other. The partitioning is done on the basis of distance vectors. This method finds out the minimum dependence distance (among all *Reference Pairs*) that can transform the loop without altering its overall result and hence expedite the loop by a factor of lambda(λ), called reduction factor.

The two array references that are involved in dependence and includes the source and sink of dependence are called *Reference Pair*. Example 2 has only one reference pair i.e. `a[i+6]-a[i-2]`. Consider a n-nested loop with indices $I_1, I_2, .., I_n$ and following statement present inside the loop:

S_1: `A[`$I_1 + a_{11}, I_2 + a_{12}, .., I_n + a_{1n}$`]=B[`$I_1 + a_{21}, I_2 + a_{a22}, .., I_n + a_{2n}$`]`

S_2: `B[`$I_1 + b_{11}, I_2 + b_{12}, .., I_n + b_{1n}$`]=A[`$I_1 + b_{21}, I_2 + b_{a22}, .., I_n + b_{2n}$`]`

In above example, there are two reference pairs;

R_1: S_1`:A[`$I_1 + a_{11}, I_2 + a_{12}, .., I_n + a_{1n}$`]` - S_2`:A[`$I_1 + b_{21}, I_2 + b_{a22}, .., I_n + b_{2n}$`]`

R_1: S_2`:B[`$I_1 + b_{11}, I_2 + b_{12}, .., I_n + b_{1n}$`]` - S_1`:B[`$I_1 + a_{21}, I_2 + a_{a22}, .., I_n + a_{2n}$`]`

Linear equation for reference pair R_1 will be:

$$I_{1_1} + a_{11} = I_{1_2} + a_{21}$$

$$I_{2_1} + a_{12} = I_{2_2} + a_{22}$$

$$..$$

$$..$$

$$I_{n_1} + a_{1n} = I_{n_2} + a_{2n}$$

and following are the linear equation of R_2:

$$I_{1_1} + b_{11} = I_{1_2} + b_{21}$$

$$I_{2_1} + b_{12} = I_{2_2} + b_{22}$$

$$..$$

$$..$$

$$I_{n_1} + b_{1n} = I_{n_2} + b_{2n}$$

The distance vectors for reference pairs R_1 and R_2 are $< \phi_1^1, \phi_2^1, .., \phi_n^1 >=< a_{11} - a_{21}, a_{12} - a_{22}, .., a_{1n} - a_{2n} >$ and $< \phi_1^2, \phi_2^2, .., \phi_n^2 >=< b_{11} - b_{21}, b_{12} - b_{22}, .., b_{1n} - b_{2n} >$ respectively. Finally reduction factor can be calculated as $< \lambda_1, \lambda_2, .., \lambda_n >=<$

$min(|\phi_1^1|, |\phi_1^2|), min(|\phi_2^1|, |\phi_2^2|), .., min(|\phi_n^1|, |\phi_n^2|) >$.

2.5.2 Simple Loops

Consider a dependence cycle where all dependence have same distance which should be greater than 1. In such cases, cycle shrinking will speed up the loop by reduction factor.

Case 1. Consider a singly nested serial loop with m statement $S_1, S_2, .., S_m$ that are involved in a dependence cycle, such that $S_1\delta S_2\delta....S_m\delta S_1$. All m dependence have a constant distance which is greater than 1. For constant dependence , the array subscript expression of elements are of form $I \pm a$, where $a \geq 0$ and I is the loop index. Loop transformations by this case can be obviated by the below example:

Example 4.
```
DO I=1 to N
   S1:  A(I+K)=B(I)-1;
   S1:  A(I+K)=B(I)-1;
ENDO
```

Above is the example of loop with constant dependence distance. Transformed loop of Example 4 using cycle shrinking(here, $\lambda = k$ will be like:

```
DO I=1, N, K
  DOALL J=1, I+K-1
    S1:  A(J+K)=B(J)-1;
    S1:  B(J+K)=A(J)+C(J);
  ENDOALL
ENDO
```

Case 2. This case handles the loops where distance of each dependence is constant but distance between dependencies are different. Consider cycle $S_1\delta_1 S_2, ..., S_k\delta_k S_1$ where ϕ_i is the distance of i_{th} dependence. Then, without loss of generality, assume that $\phi_1 \geq \phi_2 \geq, .., \geq \phi_n$.

Example 5.
```
DO I=1, N
  S1:  X(I)=Y(I)+Z(I);
  S2:  Y(I+3)=X(I-4)*W(I);
ENDO
```

In Example 5, $\phi_1 = 4,\ \phi_2 = 3$. Acccording to section 2.5.1 , λ gets reduced by a factor of $\lambda = min\{\phi_1,\ \phi_2\} = 3$. Therfore, the transformed loop will be:

```
DO J=1, N, 3
  DOALL I=J, J+2
    S1:  X(I)=Y(I)+z(I);
    S1:  Y(I+3)=X(I-4)*W(I);
  ENDOALL
ENDO
```

Algorithm 1 explains the loop transformation on simple loops.

Algorithm 1 Cycle Shrinking for Simple Loops

Input: A loop with I and N as the lower and upper bound, loop body and the reduction factor λ
Output: Reconstructed loop
1: **DO** $I = 0, N, \lambda$
2: **DOALL** $I_1 = I, I + \lambda - 1$
3: Loop Body
4: **ENDOALL**
5: **ENDO**

2.5.3 Complex Loops

The above two cases only handle the program with singly nested loop. To handle the nested loops, cycle shrinking provides three different types of shrinking. These shrinking are different to each other in the way they calculate individual and true distances to compute reduction factor:

I. Simple Shrinking: In this, dependence cycle is considered separately for each loop in the nest. Cycle shrinking is then applied to each individual loop in the nest similar to the way shrinking is applied for simple loop.

Algorithm 2 describes the method of simple shrinking for nested loops:

Algorithm 2 Simple Shrinking

Input: A m nested loop with loop indices $I_1, I_2, .., I_m$, 0 and $N_1, N_2, .., N_m$ as the lower and upper bound respectively, loop body and the reduction factor $\lambda_1, \lambda_2, .., \lambda_m$.

Output: Reconstructed loop

DO $I_1 = 0, N_1, \lambda_1$
..
..
  DO $I_m = 0, N_m, \lambda_m$
    DOALL $J_1 = I_1, I_1 + \lambda_1 - 1$
..
..
      DOALL $J_m = I_m, I_m + \lambda_m - 1$
        Loop Body
..
..
  ENDOALL
ENDO

Consider the below example similar to example 1 in Chapter 1,

Example 6.

```
Do I=3,N1
  Do J=4,N2
    S1:  x[I][J]=y[I-3][J-4];
    S2:  y[I][J]=x[I-2][J-3];
  ENDO
ENDO
```

Here, statements S_1 and S_2 are involved in dependence, such that $S_1 \delta S_2$ and $S_2 \delta S_1$. Dependence distance for the two dependences are $< \phi_1^1, \phi_2^1 >=< 2, 3 >$ and $< \phi_1^2, \phi_2^2 >=< 3, 4 >$. Cycle shrinking will shrink the outer and inner loop by a factor of $\lambda_1 = min(\phi_1^1, \phi_2^1) = min(2, 3) = 2$ and $\lambda_2 = min(\phi_1^2, \phi_2^2) = min(3, 4) = 3$. Therefore, transformed loop will be:

```
DO I=3,N1,2
  DO J=4,N2,3
    DOALL I1=I,I+1
      DOALL J1=J,J+2
```

```
        S1:  x[I1][J1]=y[I1-3][J1-4];
        S2:  y[I1][J1]=x[I1-2][J1-3];
      ENDOALL
    ENDOALL
  ENDO
ENDO
```

II. Selective shrinking: Selective shrinking also considers each component of the distance vectors separately as in case of simple shrinking. A m-nested loop will have m different dependence cycles associated with each loop. Each dependence in a cycle is labeled with the corresponding element of its distance vector. Selective shrinking computes the reduction factor $\lambda_i (i = 1, 2, ..., k)$ for each loop in nest starting from the outermost loop. The process stops when for some $1 \leq j \leq k$, $\lambda_j \geq 1$, j^{th} loop in the nest is blocked by a factor of λ_j. In addition, all loops nested inside the j^{th} loop are transformed to `DOALL`.

Algorithm 3 gives the design of complex loop transformation by selective shrinking.

Algorithm 3 Selective Shrinking

Input: A m nested loop with $L_1, L_2, .., L_m$ and $U_1, U_2, .., U_m$ as the lower and upper bound respectively, loop body and the reduction factor $\lambda_1, \lambda_2, .., \lambda_m$. Let say there exists a $\lambda_k > 1$.

Output: Reconstructed loop

DO $I_1 = 0, U_1, \lambda_1$
..
..
  DO $I_k = 0, U_k, \lambda_k$
  DOALL $J_1 = I_1, I_1 + \lambda_1 - 1$
..
..
    DOALL $J_k = I_k, I_k + \lambda_k - 1$
    DOALL $I_{k+1} = L_{k+1}, U_{k+1}$
..
..
      DOALL $I_m = L_m, U_m$
        Loop Body
  ENDOALL
ENDO

Transformed loop of Example 6 using selective shrinking will be as follows:

```
DO I=3,N1,2
  DOALL I1=I,I+1
    DOALL J=4,N2
      S1:  x[I1][J]=y[I1-3][J-4];
      S2:  y[I1][J]=x[I1-2][J-3];
    ENDOALL
  ENDOALL
ENDO
```

As stated earlier in selective cycle shrinking, if for any loop, $\lambda > 1$ becomes true, then all the loops next to it will be converted to `DOALL`. In example 6, for outermost loop, $\lambda = 2$. Therefore, all the inner loops will become `DOALL`.

III. True dependence shrinking(TD shrinking): In this, only true distances are used to compute the reduction factor. Each dependence in the dependence cycle is labeled by its true distance. Cycle shrinking is then applied as if the nested loop was a single loop. True dependence shrinking treats a multidimensional iteration space as a linear space. Consider a k-nested loop with upper bounds $N_1, N_2, .., N_m$ and ϕ_r be the r^{th} dependence distance, then the true distance Φ_{ij} between statement i and j is defined as:

$$\Phi_{ij} = \sum_{r=1}^{k} \phi_r \prod_{m=r+1}^{k} N_m \tag{2.11}$$

For solving Example 6 using true dependence shrinking, firstly true distance needs to be calculated using Equation 2.11. This true distance will give the total number of iterations that can execute in parallel in one step. True distance for Example 6 are:

$$\Phi_{12} = \phi_1^1(N_2 - 4 + 1) + \phi_2^1 = 2(N_2 - 3) + 3$$

$$\Phi_{21} = \phi_1^2(N_2 - 4 + 1) + \phi_2^2 = 3(N_2 - 3) + 4$$

Now the reduction factor $\lambda = min(\Phi_{12}, \Phi_{21}) = 2(N_2 - 3) + 3$

Algorithm 4 explains the method of true dependence shrinking for complex loops:

Algorithm 4 True Dependence Shrinking

Input: Two nested loop I, J with L_1, L_2 and U_1, U_2 as the lower and upper bound respectively, loop body and the reduction factor λ.

Output: Reconstructed loop

1: $N = U_1 - L_1 + 1$
2: $M = U_2 - L_2 + 1$
3: **DO** $I_1 = 1, NM, \lambda$
4: $I_L = (K/M)$
5: $I_U = ((K + \lambda)/M)$
6: $J_L = K\%M$
7: $J_U = M - ((K + \lambda)\%M)$
8: **DOALL** $J = J_L, M$
9: Loop Body with indices I_L and J
10: **ENDOALL**
11: **DOALL** $I = I_L, I_U$
12: **DOALL** $J = L_2, U_2$
13: Loop Body with indices I and J
14: **ENDOALL**
15: **ENDOALL**
16: **DOALL** $J = L_2, J_U$
17: Loop Body with indices I_U and J
18: **ENDOALL**
19: **ENDO**

Therefore according to the Algorithm 4, Example 1 will be modified as:

```
DO K=1,NM,λ
  I_L=(K/M)
  I_U=((K+λ)/M)
  J_L=(K%M)
  J_U=M-((K+λ)%M)
    DOALL J=J_L,M
      S1:  x[I_L][J]=y[I_L-3][J-4];
      S2:  y[I_L][J]=x[I_L-2][J-3];
    ENDOALL
    DOALL I=I_L,I_U
      DOALL J=L1,N2
        S1:  x[I][J]=y[I-3][J-4];
        S2:  y[I][J]=x[I-2][J-3];
```

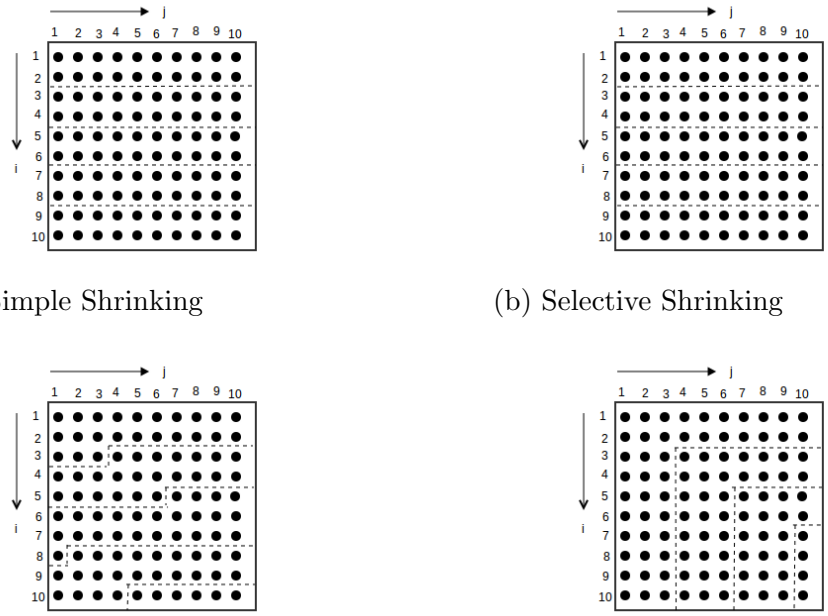
```
        ENDOALL
    ENDOALL
    DOALL J=L2,JU
      S1:  x[IU][J]=y[IU-3][J-4];
      S2:  y[IU][J]=x[IU-2][J-3];
    ENDOALL
ENDO
```

2.6 Extended Cycle Shrinking

Cycle shrinking reconstructs a loop to speed up the original loop by the reduction factor. Cycle shrinking does this by partitioning the loop iterations into groups. However, it will be of no use if the number of groups are large. Therefore, there can be two improvements in basic cycle shrinking approach. First would be to reduce the number of partitions and second would be to handle the loops that contain variable dependence distances. To incorporate these two improvements, Extended cycle shrinking [SBS95] was introduced.

Example 7.

```
for(i=3;i<N1;i++)
for(j=4;j<N2;j++){
  x[i][j]=y[i-3][j-4];
  y[i][j]=x[i-2][j-3];
}
```



(a) Simple Shrinking

(b) Selective Shrinking

(c) True Dependence Shrinking

(d) Extended Cycle shrinking

Figure 2.4: Different Partition for Example 7

2.6.1 Characterization of Reduction Factor

Consider the statement in Example 1 with constant dependence distance:

$$S_1 : x[i][j] = y[i-3][j-4]$$

$$S_2 : y[i][j] = x[i-2][j-3]$$

The process of calculating distance vector and reduction factor for extended cycle shrinking for constant dependence distance is exactly same as for cycle shrinking. Therefore, $\lambda_1 = min(\phi_1^1, \phi_2^1) = 2$ and $\lambda_2 = min(\phi_1^2, \phi_2^2) = 3$.

Now, consider a m-nested loop with loop indices $I_1, I_2, .., I_m$ and reference pair $S_j - S_i$ that contains variable dependence distance:

$S_i : A[a_{10} + a_{11}I_1 + .. + a_{1m}I_m, .., a_{m0} + a_{m1}I_1 + .. + a_{mm}I_m]$

$S_j : A[b_{10} + b_{11}I_1 + .. + b_{1m}I_m, .., b_{m0} + b_{m1}I_1 + .. + b_{mm}I_m]$

Extended cycle shrinking for variable dependence distance use data dependence vector (DDV) for computing reduction factor. The data dependence vector is calculated using the following equation:

$$\lambda_k = \frac{(a_{k0} - b_{k0}) + \sum_{i=1, i\neq k}^{m}(a_{ki} - b_{ki}) * I_i + (a_{kk} - b_{kk}) * I_k}{b_{kk}} \tag{2.12}$$

2.6.2 Extended Cycle Shrinking for Constant Dependence Distance

For a m-nested loop with upper bounds $N_1, N_2, .., N_m$ and reduction factors $\lambda_1, \lambda_2, .., \lambda_m$, extended cycle shrinking partitions this loop into minimum among $(\lceil N_i/\Phi_i \rceil \,|1 \leq i \leq m, \Phi_i \neq 0)$ groups. For instance, $peak_k$ be the first point of k_th group and $START[i]$ be the i_{th} coordinate of that group.

Following is the algorithm for extended cycle shrinking(constant dependence distance):

Algorithm 5 Extended Cycle Shrinking for Constant Dependence Distance

Input: A m nested loop $I_1, I_2, .., I_m$ with $L_1, L_2, .., L_m$ and $U_1, U_2, .., U_m$ as the lower and upper bound respectively, loop body and the distance vectors $\Phi_1, \Phi_2, .., \Phi_m$.
Output: Reconstructed loop

DO $K = 1, min\{\lceil N_i/\Phi_i \rceil \,|1 \leq i \leq m, \Phi_i \neq 0\} + 1$
  DOALL $I = 0, m - 1$
    $START[I] = (K - 1) * \Phi_i$
  ENDOALL
  $r = 1$
while $(r \leq m)$ **do**
  $i = 1$
  while $(i \leq m)$ **do**
    Introduce m nested DOALL loops based on following condition for each loop:
    if $i < r$ **then**
      DOALL $I_r = START[r] + \Phi_r, N_r$
    end if
    if $i = r$ **then**
      DOALL $I_r = START[r], min\{START[r] + \Phi_r - 1, N_r\}$
    end if
    if $i > r$ **then**
      DOALL $I_r = START[r], N_r$
    end if
  end while
end while
ENDO

Example 1 using Algorithm 5 will be transformed as given below:

```
DO K=1,min(⌈N1/2⌉ ,⌈N2/3⌉) + 1
  DOALL I=0,1
    START[I]=(K-1)*λI
```

```
    ENDOALL
    DOALL I=START[0],min(START[0]+2,N1)
      DOALL J=START[1],N2
        S1:  x[I][J]=y[I-3][J-4];
        S2:  y[I][J]=x[I-2][J-3];
      ENDOALL
    ENDOALL
    DOALL I=START[0]+2,N1
      DOALL J=START[1],min(START[1]+3,N2)
        S1:  x[I][J]=y[I-3][J-4];
        S2:  y[I][J]=x[I-2][J-3];
      ENDOALL
    ENDOALL
  ENDO
```

2.6.3 Extended Cycle Shrinking for Variable Dependence Distance

Algorithm for extended cycle shrinking for variable dependence distance is given below:

Algorithm 6 Extended Cycle Shrinking for Variable Dependence Distance

Input: A two dimensional loop with L_1, L_2 and U_1, U_2 as the lower and upper bound respectively, loop body and the distance vectors $< \Phi_1, \Phi_2 >$. Each distance vector is a function of loop indices I_1, I_2.

Output: Reconstructed loop

1: $id_1 = 1, id_2 = 1$
2: **while** $((id_1 < U_1)\&\&(id_2 < U_2))$ **do**
3: $nextid_1 == \lfloor min\{phi_1(id_1, id_2)\}\rfloor$
4: $nextid_2 == \lfloor min\{phi_2(id_1, id_2)\}\rfloor$
5: **doall** $I_1 = id_1, minnextid_1, U_1$
6: **doall** $I_2 = id_2, U_2$
7: Loop body
8: **endoall**
9: **endoall**
10: **doall** $I_1 = nextid_1, U_1$
11: **doall** $I_2 = id_2, minnextid_2, U_2$
12: Loop body
13: **endoall**
14: **endoall**
15: **end while**
16: **endo**

In above algorithm, id_1, id_2 and $nextid_2, nextid_2$ marks the *peak* of two consecutive groups. Considering the below example:

Example 8.

```
for(i=3;i<N1;i++)
for(j=4;j<N2;j++){
   x[3*i+5][3*j+7]=y[i-3][j-4];
   y[3*i+8][2*j]=x[2*i-2][j-3];
}
```

There are two reference pair i.e. $R_1 : x[3 * i + 5][3 * j + 7] - x[2 * i - 2][j + 3]$ and $R_2 : y[3 * i + 8][2 * j] - y[i + 3][j - 4]$ present in above example. For $id_1 = 1$ and $id_2 = 1$ distance vectors will be $< \phi_1^1, \phi_2^1 >=< 4, 6 >$ and $< \phi_1^2, \phi_2^2 >=< 7, 5 >$ and hence the reduction vector are $\lambda_1 = 4$ and $\lambda_2 = 5$. Therefore, below is the reconstructed loop using Algorithm 6:

$id_1 = 1, id_2 = 1$

`while`$((id_1 < N_1)\&\&(id_2 < N_2))$ {

$nextid_1$`=min(((3-2)*`id_1`+(5+2))/2,((3-1)*`id_2`+(8-3))/1)`

$nextid_2$=min(((3-1)*id_1+(7-3))/1,((2-1)*id_2+(0+4))/1)

doall $I = id_1, min(nextid_1, N_1)$

doall $J = id_2, N_2$

S_1: x[3*I+5][3*J+7]=y[I+3][J-4];

S_2: y[3*I+8][2*J]=x[2*I-2][J+3];

ENDOALL

ENDOALL

DOALL $I = nextid_1, N_1$

DOALL $J = id_2, min(nextid_2, N_2)$

S_1: x[3*I+5][3*J+7]=y[I+3][J-4];

S_2: y[3*I+8][2*J]=x[2*I-2][J+3];

ENDOALL

ENDOALL

$id_1 = nextid_1$

$id_2 = nextid_2$

ENDO

}

Chapter 3

Code Generation Framework

This chapter gives the overview of the different phases involved in the end-to-end parallelization tool. These are *Compilation Phase, Dependence Testing Phase, Parallelism Extraction Phase and Code Generation Phase*. Figure 1.1 presented the abstract view of Automatic C to CUDA code generation and the overall flow of the transformation system. In this chapter, we describe various steps and methods of each phase in detail.

3.1 Compilation Phase

This very first phase of the tool takes a sequential C program as input, performs lexical and syntactic analysis and extracts the information needed for the further phases. Figure 3.1 shows the flow of compiler front end of the system.

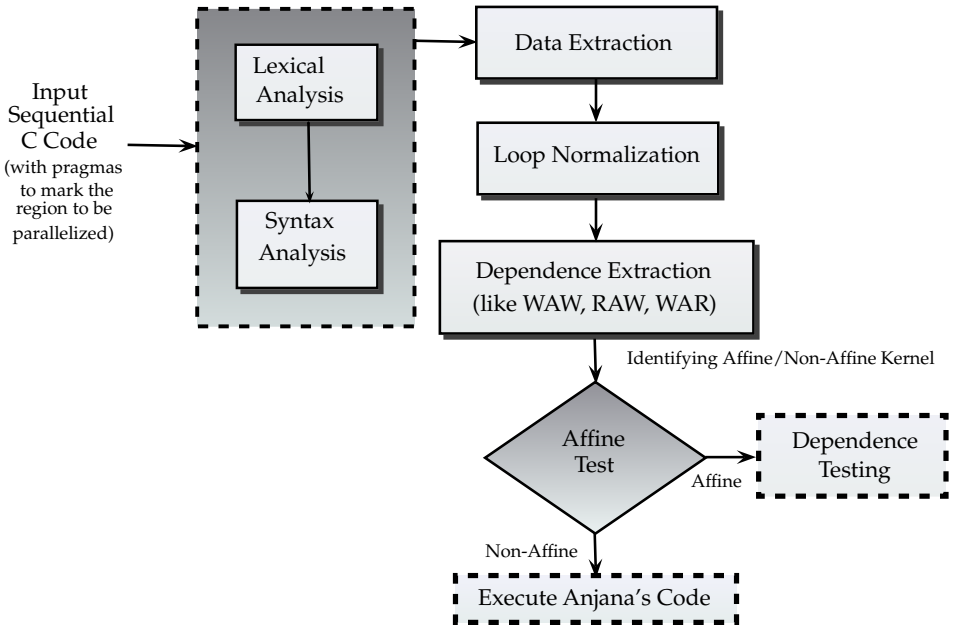


Figure 3.1: Compilation Phase.

3.1.1 Lexical Analysis

The *Lexical analysis* phase scans and segments the input program into small meaningful units i.e., *token*. Before giving input to the lexical analyzer following points must be ensured:

- The input file should be a C program. Any program other than that can not be scanned by the lexical analyzer.
- The tool assumes that input program will be syntactically correct otherwise it will generate error. Also it should be semantically correct since otherwise, it might generate semantically incorrect program.
- The program can have affine or non-affine perfectly nested loop. To generate the CUDA code for non-affine loops [Sin13] is used.
- The region that needs to be parallelized should be enclosed within # pragmas, otherwise, the tool would not be able to spot the regions of interest and would generate error.

The scanner generates the tokens only for the region which is marked under pragmas and for the variable declarations. The tokens for variable declarations will give the type, size and value(if present) of the variables, and the tokens of the marked region will give the information about the loop index, its boundaries and the statement enclosed in loop.

3.1.2 Syntactic Analysis

Syntactic analysis, also known as *Parsing*, takes tokens generated from the lexical analyzer and matches the grammar rules defined in *yacc* file. Since it is already mentioned in lexical analysis that token will be generated only for marked regions and for variable declaration, grammar rules are defined only for these sections.

3.1.3 Data Extraction

Syntactic analysis creates parse tree if grammar rules corresponding to the token generated from the lexer matches. The process of data extraction works parallel to the syntactic analyzer. Creation of parse tree ensures that the input program is error free and hence following data can be extracted from the parse tree:

- Variable name, type, size and value if variable is initialized.
- Information about loop indices, bounds, relational operator and increment/decrement operator and value.
- Data about statement present inside the *for* loop and their subscript values.

Following data structures are created for above extracted data:

- List of statements inside the loop
- A list of variables that are used in the statements enclosed within perfectly nested loop.
- List of loops with the details about their index, bounds, relational operator and increment/decrement operator and value.

3.1.4 Loop Normalization

Normalized loop is a standard form of loop that starts from 0 and increments by 1. There are some dependence tests that consider the loop upper and lower bound while identifying the dependence in loop. Therefore performing dependence test in non-normalized loop will give incorrect results and loop normalization is required before going for further phases. Below are the two examples for non-normalized loop:

Example 9.
```
for(i=n;i>=0;i--)
    a[i]=b[i]+d[i];
    d[i]=a[i]+x[i];
```

Above is an example of reverse loop as it has n and 0 as lower and upper bound values respectively. After normalization, the above loop will look like the following:

```
for(i=0;i<=n;i++)
    a[n-i]=a[n-i]+x[n-i];
```

Example 10.
```
for(i=3;i<50;i+=5)
    a[i]=a[i]+x[i];
```

Here, the loop initial value is non-zero and increment value is not 1. Therefore, this loop is not normalized. The normalized form of above example will be:

```
for(i=0;i<(50-3)/5;i++)
    a[i]=a[i]+x[i];
```

Loop normalization process takes the information about loop from data extraction step and identifies whether the loop is normalized or not. If not, then necessary changes are carried out on the loop. After this step, everything will be processed on the normalized loop.

3.1.5 Dependence Extraction

Dependence extraction takes the data structures as input that are generated in data extraction phase. It extracts out the variable referenced and their corresponding write and read references made in the statement . Using this information it identifies the dependence(like WAW, RAW, WAR) between the statements present inside the perfectly nested loops. Dependence extraction finds out the following:

- List of all the statement variables that have write reference or read reference and are involved in some dependence.
- List of write references that comes after a read reference(WAR dependence) or a write reference(WAW dependence).
- List of all read references that come after a write reference(RAW dependence)

For Example 9, dependence extraction gives the following information:

- List of statement variables: a, b, d, x
- List of write references: $a[i], d[i]$
- List of read references: $d[i], a[i]$

The output of dependence extraction will be used by further phases of the tool.

3.1.6 Affine Test

This step checks whether the given input program contains an affine or an non-affine loop. If the program contains an affine loop then the next phase will be dependence testing, otherwise an existing approach [Sin13] is executed to convert sequential C program into CUDA program.

3.2 Dependence Test

Consider a m-nested loop with large number of iterations which, if executed sequentially, will take a huge computation time. So the idea is to parallelize the loop so that

many iterations can execute in parallel and hence the loop takes less time compared to sequential execution. But this can only happen if the statement within loop does not form dependence with respect to loop. If loop statement indulges in dependence which is carried by loop (i.e., *loop carried dependence*) then all iterations can not execute in parallel. Therefore, loop dependence analysis is performed to find out if dependence exists or not. If dependence exists, then there should be some technique that will find out which iterations can execute in parallel and process takes place in parallelism extraction pahase. But if dependence doesn not exist, then CUDA code will be generated because each iteration can execute in parallel on GPU. Figure 3.2 shows the loop dependence analysis phase.

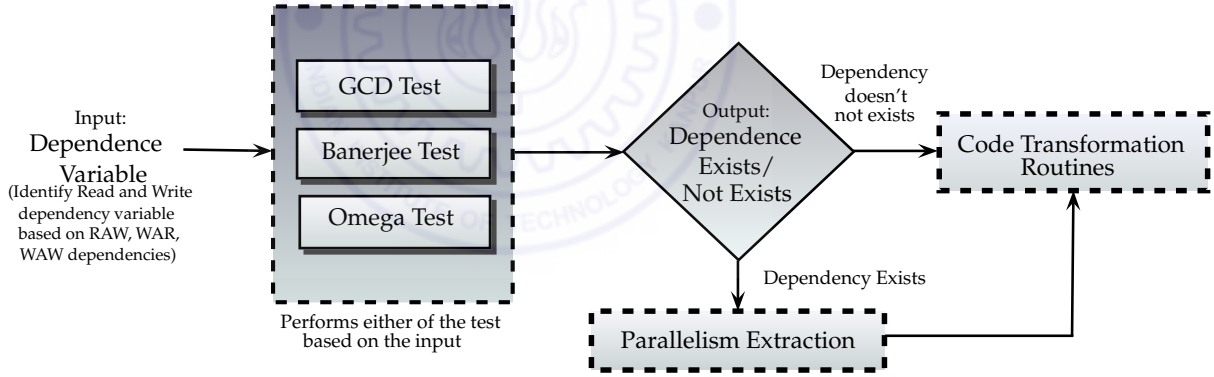


Figure 3.2: Dependence Test.

This phase takes read and write references of the variables involved in the dependence (like WAW, RAW or WAW) as an input from the compilation phase. The output of dependence testing is *Yes* if dependence exists and *No* if not.

The tool provides three different loop dependence test i.e. *GCD, Banerjee*

and *Omega* test. User can configure which test to run using command line option. The user will have to enter his choice from command line and the choices are `--depTest=gcd`(for GCD test), `--depTest=banerjee`(for Banerjee test) and `--depTest=omega`(for omega test).

3.3 Parallelism Extraction

If dependency exists in input program, then next phase would be parallelism extraction. Since dependency prevails, each loop iteration can not execute in parallel to other iteration. The purpose of this phase is to extract parallelism out of the sequential loop by partitioning the iteration space into groups of iterations. Iterations within a group are independent of each other and therefore all iterations within a group can execute in parallel. While the iterations from different groups can have dependency on each other, two groups cannot execute in parallel. Hence, only one group will execute at a time on GPU. Figure 3.3 shows the different steps involved in parallelism extraction phase. Seeing that dependency exists, the tool will require read and write reference variables as input to identify the distance vectors.
The tool uses *Cycle shrinking* [Pol88] and *Extended cycle shrinking* [SBS95] for paralleism extraction. The basic cycle shrinking offers three versions for shrinking i.e. *Simple*, *Selective* and *True dependence shrinking* and the extended cycle shrinking provides two versions i.e. extended cycle shrinking for *constant dependence distance* and for *variable dependence distance*. Here also, user decides which cycle shrinking to perform by giving any of the following options from command line:

Cycle shrinking type	Command line option
Simple shrinking	`simple`
Selective shrinking	`selective`
True dependence shrinking	`true_dependence`
Extended cycle shrinking for constant dependence distance	`extShrinkingConst`
Extended cycle shrinking for variable dependence distance	`extShrinkingVar`

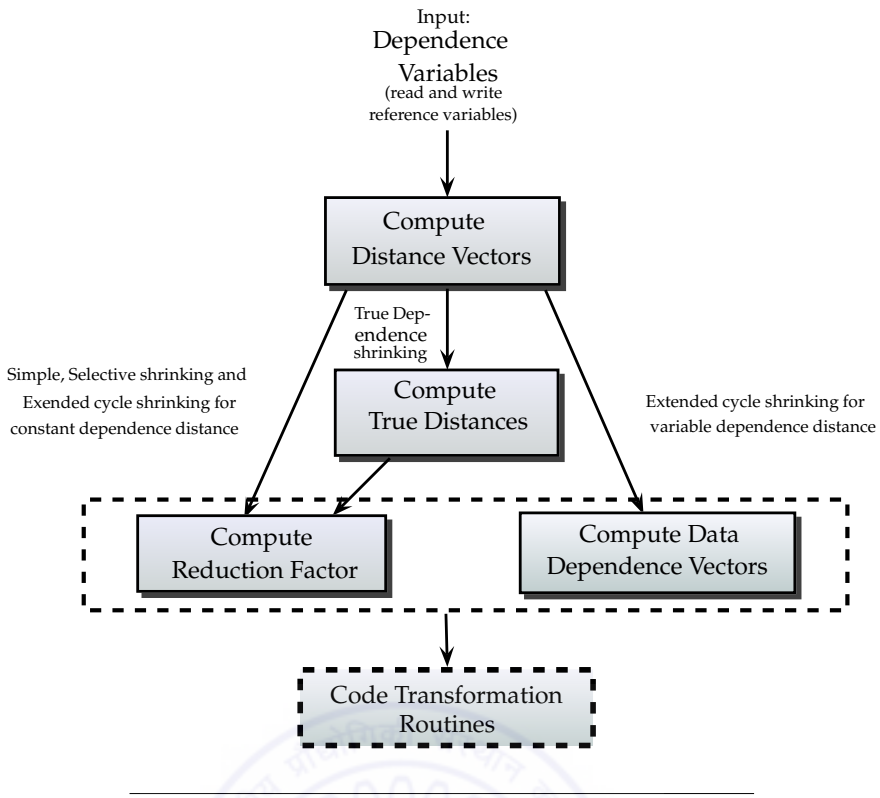


Figure 3.3: Parallelism Extraction.

3.3.1 Compute Distance Vectors

Distance vector gives the distance between the two consecutive access of an array. These successive access can be any of the following:

- write after write access
- read after write access
- write after read access

It takes read and write reference variables as input and then finds out all possible reference pairs. Now distance vector can be calculated using these reference pairs. If any variable is involved in more than one reference pair then each coordinate of actual distance vector for that variable will take the minimum among the corresponding coordinates of all the distance vector belonging to that variable. Consider the following statement:

S_1 : `a[i][j]=b[i-2][j]+a[i-3][j-5];`

S_2 : `x[i][j]=a[i-4][j-2];`

Here, for array a there are two reference pairs: R_1 : $\{S_1 : a[i][j] - S_1 : a[i-3][j-5]\}$

and R_2 : $\{S_1 : a[i][j] - S_2 : a[i-4][j-2]\}$. So the distance vector for R_1 and R_2 are $< \phi_1^1, \phi_1^2 >=< 3, 5 >$ and $< \phi_2^1, \phi_2^2 >=< 4, 2 >$ respectively. Therefore the actual distance vector correspond to $a[][]$ is $< 3, 2 >$.

3.3.2 Compute True Distance (TD)

Calculating distance vector is the basic requirement of parallelism extraction. After doing this, depending on the user preferred choice for cycle shrinking different parameters are calculated. If user input is `true_dependence` then *true distance* will be calculated. For any dependence, the total number of loop iterations between its source and sink is given by true distance. True distance is calculated using equation 2.11.

3.3.3 Calculating Reduction Factor

After calculating distance vector, if user input is `simple` or `selective` or `extShrinkingConst` then the reduction factor(λ) is calculated. As reduction factor is the minimum distance of each coordinate among all the corresponding coordinates of every distance vectors. The i^{th} value of λ gives the total number of iterations of loop at i^{th} level that can run in parallel.

If the user input is `true_dependence`, then reduction factor will be calculated after computing true distance. Because the true dependence shrinking imitates an m-nested loop as a singly nested loop, the reduction factor in case of true dependence shrinking contains a single value that gives the total number of iterations of this single loop that can execute in parallel.

3.3.4 Compute Data Dependence Vector

Data dependence vector is calculated for each reference pair. If the user input is `extShrinkingVar` then data dependence vector (DDV) are calculated using equation 2.12.

3.4 Code Generation

This is the last phase of the tool, that generates the final CUDA code as output. If dependence exists then result of parallelism extraction, i.e. reduction factor or data dependence vector, acts as an input for the code generation phase. Otherwise the tool directly jumps to this phase after dependence testing. Using reduction factor and DDVs the loop iterations are partitioned into groups of independent iterations. Figure 3.4 shows the various routines involved in code generation.

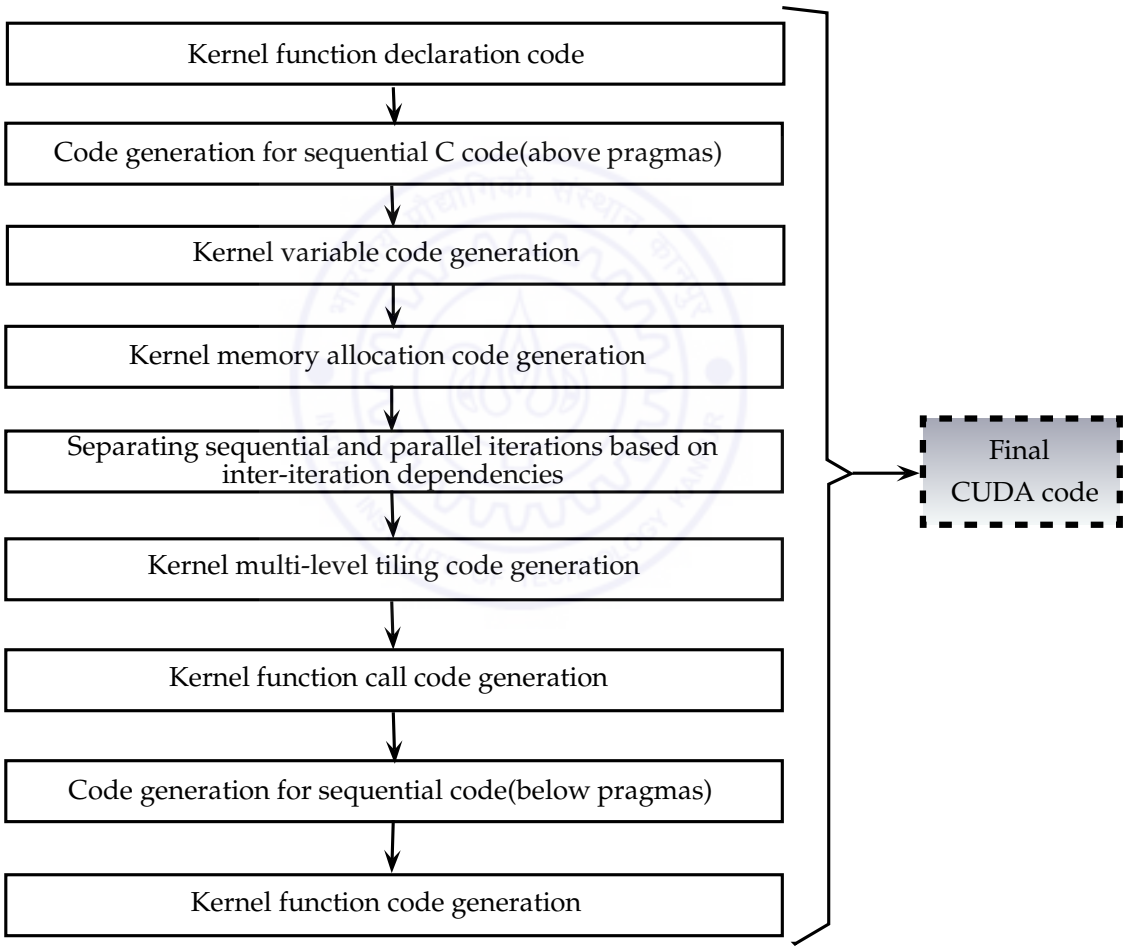


Figure 3.4: Code Generation Phase.

Following are the code generation routines that will generate different portions of the target CUDA code:

Kernel function declaration code: This is the initial step which will generate

the code corresponding to the declaration of kernel declaration. The kernel function declaration will give the information about kernel name, number of arguments and their respective type.

Code generation for sequential C code(above pragmas): The input program contains pragmas to mark the region which is compute intensive. The next step is to generate the code for the portion of the input program above pragmas.

Kernel variable code generation: Since device variables are required for the kernel execution that has memory allocated on device, this step will generate the target code for declaration of these variables on host.

Kernel memory allocation code generation: This generates the code for allocating memory to device variable declared in previous step on GPU. It allocates the space for these variables on global memory.

Kernel multi-level tiling code generation: This generates the code for identifying the number of thread and blocks required to execute the instruction on GPU. Also, if data becomes too large to accommodate in one launch of kernel, then *tiling* is required. Tiling partitions the data into tiles such that the size of each tile can fulfill the demand of threads and blocks in a kernel launch. Consider an array a of size N. Calculation of number of threads, blocks and tiles is given below:

```
int NTHREAD=512, NBLOCKS=65535;
int NUM_THREADS = N, NUM_BLOCKS=1, int NUM_TILE=1;
dim3 _THREADS(512);
dim3 _BLOCKS(1);
if(NUM_THREADS < NTHREAD){
            _THREADS.x=NUM_THREADS;}
```

```
else{
          _THREADS.x=NTHREAD;
          NUM_BLOCKS=NUM_THREADS/512;
          if(NUM_BLOCKS<NBLOCK)
               _BLOCKS.x=NUM_BLOCKS;
           else{
               _BLOCKS.x=NBLOCK;
               int temp=NUM_BLOCKS;
               NUM_TILE=(temp % NBLOCK == 0)?(NUM_BLOCKS/NBLOCK):
((NUM_BLOCKS/NBLOCK)+1);}}
```

Separating sequential and parallel iterations based on inter-iteration dependencies : This step generates the code for the loop transformation performed by cycle shrinking

Kernel function call code generation: It generates the code for calling kernel function. This will specify the number of threads and blocks allocated to the function for execution and the actual parameters of the function.

Code generation for sequential code(below pragmas): This step of the code generation phase generates the code for the portion below pragmas in the source program. The code below and above pragmas has nothing to do with the GPU as it will execute on CPU.

Kernel function code generation: This is the last step that generates the code for the kernel function definition. In kernel body, indices are required to access the data available on device. Indices are computed using CUDA variable. Code for calculating an index is given below:

```
int index = blockDim.x*blockIdx.x + threadIdx.x
```

In case of tiling,

```
int index = gridDim.x*blockDim.x*_CUDA_TILE + blockDim.x*blockIdx.x +
            threadIdx.x
```

Here, `_CUDA_TILE` specifies the number of tile.

After the above routines, the final CUDA code is ready for execution on GPU.

Chapter 4

A tour to CRINK

This chapter gives outline of the overall working of CRINK . As already discussed, this research uses GCD[WSO95], Banerjee[AK04], Omega[KMP+96] for dependence test and Cycle shrinking[Pol88], Extended cycle shrinking[SBS95] for parallelism extraction. This chapter explains the code transformation process in detail with the help of some examples and by considering various user inputs, for instance, what happens if user wants to perform Omega test and use true dependence shrinking for parallelism extraction or to use Banerjee test and extended cycle shrinking for dependence test and parallelism extraction etc.

4.1 Singly Nested Loops

Consider the below singly nested loop with only one statement:

Example 11.
```
for(j=1;j<=10;j++)
  A[j]=A[j+10]+10;
```

Then converting the sequential input into parallel CUDA program will pass through following phases:

Compilation Phase

The very first phase of CRINK is *Compilation*. This step performs scanning and parsing of input program which checks whether the input C program is syntactically

correct or not. The next stage is to normalize the loop if its lower bound is some non-zero value and the loop is incremented/decremented by the value greater than 1. The above example consists of non-normalized loop, therefore the *loop normalization* stage normalizes the loop as given below:

```
for(j=0;j<=9;j++)
  A[j+1]=A[j+11]+10;
```

After normalization, the next task is to identify whether the loop is *affine* or *non-affine*. Since the array indices are linear function of loop variables, loop in Example 11 is affine.

Dependence Test

Based on the user input i.e.`gcd`, `banerjee` or `omega`, this phase identifies the dependency within a loop. Following are the results corresponding to various user inputs:

- `gcd:` The linear equation for the normalized form of Example 11 is:

$$j_1 + j_2 = (11 - 1)$$

  Therefore the $gcd(1, 1)$ evenly divides 10 and hence dependency exists.

- `banerjee:` For banerjee's test, the lower and upper bound for different direction vector will be:

$$LB_j^{<} = -9 \leq B_0 - A_0 = 10 \leq UB_j^{<} = -1$$
$$LB_j^{=} = 0 \leq B_0 - A_0 = 10 \leq UB_j^{=} = 0$$
$$LB_j^{<} = 0 \leq B_0 - A_0 = 10 \leq UB_j^{>} = 0$$

  Since the value of $(B_0 - A_0)$ does not satisfy any of the lower and upper bound constraints, no dependency exists with any of the direction vector.

- `omega:` The input file for omega calculator will be:

```
T1 := {[j11,j21]:  0 <= j11,j21 <= 9 && 1+j11 = 1+j21+10};
```

```
T1;
```

and corresponding output file will be:

```
# Omega Calculator v2.1 (based on Omega Library 2.1, July, 2008):
# T1 := [j11,j21]:  0 <= j11,j21 <= 9 && 1+j11 = 1+j21+10;
#
# T1;


{[j11,j21] :  FALSE }


#
```

The line containing {`[j11,j21] : FALSE` } specifies that there exists no dependency.

Hence only GCD test assumes that dependence exists while Banerjee and Omega come up with the solution that dependency does not exist.

Parallelism Extraction

Below is the calculation of reduction factor based on the user input for cycle shrinking:

- `simple` or `extShrinkingConst`: Because there is only one reference pair i.e. $A[j+1] - a[j+11]$, the distance vector becomes the reduction factor(λ). Therefore, $\lambda = 10$.

- `selective` or `true_dependence`: Selective and true dependence shrinking transform only nested loops. Howver, Example 11 consists of singly nested loop. Therefore, these two shrinking approaches cannot be applied on the above example.

- `extShrinkingVar`: Extended cycle shrinking are only applied to loops that contain array indices of the form $aj \pm b$ and the array indices of Example 11 belongs to $j \pm b$ form. Therefore, this shrinking also cannot be applied.

Code Transformation

Code transformation consists of several routines that perform various transformations on source C program to generate parallel CUDA code. The routines required for code generation are already discussed in Chapter 3 "Code Generation Framework". In parallelism extraction phase, it gets clear that for Example 11, only simple shrinking and extended cycle shrinking for constant dependence distances can be used for code generation while selective shrinking, true dependence shrinking and extended cycle shrinking for variable dependence distance cannot be used for code transformation as their conditions are not getting fulfilled.

Following are the scenarios corresponding to various user inputs for dependence test and code transformation:

- If user wants to use `gcd` and `simple` for dependence test and parallelism extraction respectively. Then GCD will detect dependence in the loop as already discussed in *Dependence Test* phase and hence simple shrinking will transform the loop as shown in Code 4.1.

Code 4.1: Loop Transformation using Simple Shrinking

```
int _SZ_A_1 = 100;
int *_DEV_A;
// Allocating device memory to the kernel variable
cudaMalloc((void**) &_DEV_A, sizeof(int)*_SZ_A_1);
// Copying Kernel variable from host to device
cudaMemcpy(_DEV_A, A, sizeof(int)*_SZ_A_1, cudaMemcpyHostToDevice);
int _NUM_THREADS = 100;
float _NUM_BLOCKS=1;
int _NUM_TILE=1;
dim3 _THREADS(512);
dim3 _BLOCKS(1);
// Tiling and declaring threads and blocks required for Kernel
```

```
    Execution
if(_NUM_THREADS < _NTHREAD)
        _THREADS.x=_NUM_THREADS;
else{
         _THREADS.x=_NTHREAD;
        _NUM_BLOCKS=(_NUM_THREADS % _NTHREAD ==
            0)?(_NUM_THREADS/_NTHREAD):((_NUM_THREADS/_NTHREAD)+1);
        if(_NUM_BLOCKS<_NBLOCK)
                _BLOCKS.x=_NUM_BLOCKS;
        else{
                _BLOCKS.x=_NBLOCK;
                int temp=_NUM_BLOCKS;
                _NUM_TILE=(temp % _NBLOCK ==
                    0)?(_NUM_BLOCKS/_NBLOCK):((_NUM_BLOCKS/_NBLOCK)+1);}}
int _CUDA_TILE;
// Code transformation through Simple cycle shrinking
for(j=0;j<=99;j+=100)
for(_CUDA_TILE=0;_CUDA_TILE<_NUM_TILE;_CUDA_TILE++){
        _AFFINE_KERNEL<<<_BLOCKS,_THREADS>>>(_DEV_A, _SZ_A_1, 1, j,
            0, 99, _CUDA_TILE);
        cudaDeviceSynchronize();}
// Copying Kernel variable from device to host
cudaMemcpy(A, _DEV_A, sizeof(int)*_SZ_A_1, cudaMemcpyDeviceToHost);
// Releasing the memory allocated to kernel variable
cudaFree(_DEV_A);
```

Kernel definition for above code:

```
__global__ void _AFFINE_KERNEL(int* A,int _SZ_A_1,int phi_count, int
    CUDA_j, int CUDA_L_j,int CUDA_U_j, int _CUDA_TILE){
        int j = gridDim.x*blockDim.x*_CUDA_TILE +
            blockDim.x*blockIdx.x + threadIdx.x;
```

```
        if((CUDA_j<=j)&&(j<(CUDA_j+100))&&(j<=CUDA_U_j)){
                A[1+j]=A[1+j+100]+10;}}
```

- If user enters `gcd` for dependence test and either `selective` or `true_dependence` for loop reconstruction, then the tool generates a warning message stating *Oops!! Selective/ True Dependence shrinking can't be applied on Single nested loop* and exits from the transformation system.

- If the user input is `gcd` and `extShrinkingConst` the parallel code generated will be as shown in Code 4.2.

  Code 4.2: Loop Transformation using Extended Cycle Shrinking for Constant Dependence Distance

```
int _SZ_A_1 = 100;
int *_DEV_A;
// Allocating device memory to the kernel variable
cudaMalloc((void**) &_DEV_A, sizeof(int)*_SZ_A_1);
// Copying Kernel variable from host to device
cudaMemcpy(_DEV_A, A, sizeof(int)*_SZ_A_1, cudaMemcpyHostToDevice);
int _NUM_THREADS = 100;
float _NUM_BLOCKS=1;
int _NUM_TILE=1;
dim3 _THREADS(512);
dim3 _BLOCKS(1);
// Tiling and declaring threads and blocks required for Kernel
    Execution
if(_NUM_THREADS < _NTHREAD)
        _THREADS.x=_NUM_THREADS;
else{
         _THREADS.x=_NTHREAD;
        _NUM_BLOCKS=(_NUM_THREADS % _NTHREAD ==
```

```
            0)?(_NUM_THREADS/_NTHREAD):((_NUM_THREADS/_NTHREAD)+1);
        if(_NUM_BLOCKS<_NBLOCK)
                _BLOCKS.x=_NUM_BLOCKS;
        else {
                _BLOCKS.x=_NBLOCK;
                int temp=_NUM_BLOCKS;
                _NUM_TILE=(temp % _NBLOCK ==
                    0)?(_NUM_BLOCKS/_NBLOCK):((_NUM_BLOCKS/_NBLOCK)+1);}}
// Code transformation through Extended cycle shrinking for constant
dependence distance
int ID_1, ID_2, START[1];
int _CUDA_TILE;
int Phi[1]={100};
int loopUpperLimits[1]={99};
for(ID_1=1;ID_1<=99/100+1;ID_1++){
        for(ID_2=0;ID_2<1;ID_2++){
                if(Phi[ID_2]>=0)
                        START[ID_2]=(ID_1-1)*Phi[ID_2];
                else
                        START[ID_2]=loopUpperLimits[ID_2]+(ID_1-1)*Phi[ID_2];}
        for(_CUDA_TILE=0;_CUDA_TILE<_NUM_TILE;_CUDA_TILE++){
                _AFFINE_KERNEL<<<_BLOCKS,_THREADS>>>(_DEV_A, _SZ_A_1,
                    START[0], MIN(START[0]+100, 99), _CUDA_TILE);
                cudaDeviceSynchronize();}}
// Copying Kernel variable from device to host
cudaMemcpy(A, _DEV_A, sizeof(int)*_SZ_A_1, cudaMemcpyDeviceToHost);
// Releasing the memory allocated to kernel variable
cudaFree(_DEV_A);
```

Kernel definition for above code:

```
__global__ void _AFFINE_KERNEL(int* A,int _SZ_A_1,int CUDA_L_j,int
```

```
CUDA_U_j, int _CUDA_TILE){
    int j = gridDim.x*blockDim.x*_CUDA_TILE +
        blockDim.x*blockIdx.x + threadIdx.x;
    if((CUDA_L_j<=j)&&(j<=CUDA_U_j)){
            A[1+j]=A[1+j+100]+10;}}
```

- As already mentioned in parallelism extraction phase, extended cycle shrinking for variable dependence distance cannot be applied to the loop containing array index of the form $j \pm b$. Therefore, for user input `extShrinkingVar` in place of `extShrinkingConst` in previous case, a warning message gets generated saying *Oops!! Wrong input. Please give simple, selective, true_dependence or extShrinkingConst as input.*

- Because banerjee and omega test does not detect any dependence in the loop, irrespective of any user input for parallelism extraction the parallel code shown in Code 4.3 will be generated.

Code 4.3: Loop Transformation when Dependency does not exists

```
int _SZ_A_1 = 100;
int *_DEV_A;
// Allocating device memory to the kernel variable
cudaMalloc((void**) &_DEV_A, sizeof(int)*_SZ_A_1);
// Copying Kernel variable from host to device
cudaMemcpy(_DEV_A, A, sizeof(int)*_SZ_A_1, cudaMemcpyHostToDevice);
int _NUM_THREADS = 100;
float _NUM_BLOCKS=1;
int _NUM_TILE=1;
dim3 _THREADS(512);
dim3 _BLOCKS(1);
// Tiling and declaring threads and blocks required for Kernel
```

```
    Execution
if(_NUM_THREADS < _NTHREAD)
        _THREADS.x=_NUM_THREADS;
else{
         _THREADS.x=_NTHREAD;
        _NUM_BLOCKS=(_NUM_THREADS % _NTHREAD ==
            0)?(_NUM_THREADS/_NTHREAD):((_NUM_THREADS/_NTHREAD)+1);
        if(_NUM_BLOCKS<_NBLOCK)
                _BLOCKS.x=_NUM_BLOCKS;
        else {
                _BLOCKS.x=_NBLOCK;
                int temp=_NUM_BLOCKS;
                _NUM_TILE=(temp % _NBLOCK ==
                    0)?(_NUM_BLOCKS/_NBLOCK):((_NUM_BLOCKS/_NBLOCK)+1);}}
int _CUDA_TILE;
for(_CUDA_TILE=0;_CUDA_TILE<_NUM_TILE;_CUDA_TILE++){
        _AFFINE_KERNEL<<<_BLOCKS,_THREADS>>>(_DEV_A, _SZ_A_1, 0, 99,
            _CUDA_TILE);
        cudaDeviceSynchronize();}
// Copying Kernel variable from device to host
cudaMemcpy(A, _DEV_A, sizeof(int)*_SZ_A_1, cudaMemcpyDeviceToHost);
// Releasing the memory allocated to kernel variable
cudaFree(_DEV_A);
```

Kernel definition for above parallel code:

```
__global__ void _AFFINE_KERNEL(int* A,int _SZ_A_1,int CUDA_L_j,int
    CUDA_U_j, int _CUDA_TILE){
        int j = gridDim.x*blockDim.x*_CUDA_TILE +
            blockDim.x*blockIdx.x + threadIdx.x;
        if((CUDA_L_j<=j)&&(j<=CUDA_U_j)){
                A[1+j]=A[1+j+100]+10;}}
```

4.2 Multi-Nested Loops with Constant Dependence Distance

Example 11 was a quite simple one that contains only one loop with a single statement inside and also only one reference pair. We consider a more complex example that could cover more user input scenarios.

Example 12.
```
for(i=3;i<=23;i++)
  for(j=5;j<=20;j++)
    X[i+5][j+1]=Y[i-1][j-2]+X[i][j-1];
    Y[i+2][j+2]=X[i+2][j-1]+10;
```

CRINK takes the C program containing the above loop as input and will pass through following phases:

Compilation Phase

Compilation phase starts with lexical and syntactic analysis to ensure that the input program does not contain any syntax error. At the time of parsing, data is extracted from the tokens if the respective grammar rule matches. Since example 12 contains non-normalized loop, below is the normalized form generated by the tool.

```
for(i=0;i<=20;i++)
  for(j=0;j<=15;j++)
    X[i+8][j+6]=Y[i+2][j+3]+X[i+3][j+4];
    Y[i+5][j+7]=X[i+5][j+4]+10;
```

In normalized form the array indices of X and Y are linear function of loop variable, hence this example consists of *affine* loop.

Dependence Test

For the above normalized loop, following are the results of carrying out various dependence test:

- `gcd:` GCD finds out the dependence with the help of linear equation of form:

$$ai_1 + bi_2 = (d - c) \tag{4.1}$$

Dependence exists if *gcd(a,b)* evenly divides *(d-c)*, where *a,b* are the coefficients of loop variables and *d,c* are the constant terms. In example 12, there are three reference pairs:

1. Linear equation for reference pair $X[i+8][j+6] - X[i+3][j+4]$ are:

$$i_1 + i_2 = (8 - 3)$$
$$j_1 + j_2 = 6 - 4$$

2. Linear equation for reference pair $X[i+8][j+6] - X[i+5][j+4]$ are:

$$i_1 + i_2 = (8 - 5)$$
$$j_1 + j_2 = 6 - 4$$

3. Linear equation for reference pair $Y[i+5][j+7] - Y[i+2][j+3]$ are:
   $i_1 + i_2 = (5 - 2)$
   $j_1 + j_2 = 7 - 3$

In all above linear equations *gcd(a,b)* divides *(d-c)*, GCD detects dependence in the loop.

- `banerjee:` If user's choice for dependence test is banerjee, then lower and upper bounds for different direction vectors needs to be calculated. Figure 4.1 shows the upper and lower bound constraints for each reference pair present in example 12. In figure 4.1, it is clear that the value of $(B_0 - A_0)$ in all reference pairs satisfy the constraints for direction vector '>'. Therefore, dependence exists with direction vector '>' only.

- `omega:` For omega dependence test, input file for omega calculator is:

```
T1 := [i11,i21,j11,j21]:  0 <= i11,i21 <= 20 && 0 <= j11,j21 <= 15
&& 3+i11+2 = 3+i21-1 && 5+j11+2 = 5+j21-2;
```

$$LB_i^< = -20 \leq B_0 - A_0 = 3 \leq UB_i^< = -1$$
$$LB_i^= = 0 \quad \leq B_0 - A_0 = 3 \leq UB_i^= = 0$$
$$LB_i^> = 1 \quad \leq B_0 - A_0 = 3 \leq UB_i^> = 20$$
$$LB_j^< = -15 \leq B_0 - A_0 = 2 \leq UB_j^< = -1$$
$$LB_j^= = 0 \quad \leq B_0 - A_0 = 2 \leq UB_j^= = 0$$
$$LB_j^> = 1 \quad \leq B_0 - A_0 = 2 \leq UB_j^> = 15$$

(a) For Reference Pair $X[i+8][j+6] - X[i+3][j+4]$

$$LB_i^< = -20 \leq B_0 - A_0 = 5 \leq UB_i^< = -1$$
$$LB_i^= = 0 \quad \leq B_0 - A_0 = 5 \leq UB_i^= = 0$$
$$LB_i^> = 1 \quad \leq B_0 - A_0 = 5 \leq UB_i^> = 20$$
$$LB_j^< = -15 \leq B_0 - A_0 = 2 \leq UB_j^< = -1$$
$$LB_j^= = 0 \quad \leq B_0 - A_0 = 2 \leq UB_j^= = 0$$
$$LB_j^> = 1 \quad \leq B_0 - A_0 = 2 \leq UB_j^> = 15$$

(b) For Reference Pair $X[i+8][j+6] - X[i+5][j+4]$

$$LB_i^< = -20 \leq B_0 - A_0 = 3 \leq UB_i^< = -1$$
$$LB_i^= = 0 \quad \leq B_0 - A_0 = 3 \leq UB_i^= = 0$$
$$LB_i^> = 1 \quad \leq B_0 - A_0 = 3 \leq UB_i^> = 20$$
$$LB_j^< = -15 \leq B_0 - A_0 = 4 \leq UB_j^< = -1$$
$$LB_j^= = 0 \quad \leq B_0 - A_0 = 4 \leq UB_j^= = 0$$
$$LB_j^> = 1 \quad \leq B_0 - A_0 = 4 \leq UB_j^> = 15$$

(c) For Reference Pair $Y[i+5][j+7] - Y[i+2][j+3]$

Figure 4.1: Upper and Lower Bound Constraints

```
T1;
T2 := [i11,i21,i22,j11,j21,j22]:  0 <= i11,i21,i22 <= 20 &&
0 <= j11,j21,j22 <= 15 && 3+i11+5 = 3+i21+2 && 5+j11+1 = 5+j21-1 &&
3+i11+5 = 3+i22 && 5+j11+1 = 5+j22-1;
T2;
```

and the output of omega calculator is given below:

```
# Omega Calculator v2.1
(based on Omega Library 2.1, July, 2008):
# T1 := [i11,i21,j11,j21]:  0 <= i11,i21 <= 20 && 0 <= j11,j21 <=
15 && 3+i11+2 = 3+i21-1 && 5+j11+2 = 5+j21-2;
#
# T1;

[i11,i11+3,j11,j11+4]:  0 <= i11 <= 17 && 0 <= j11 <= 11
```

```
#
# T2 := [i11,i21,i22,j11,j21,j22]:  0 <= i11,i21,i22 <= 20
&& 0 <= j11,j21,j22 <= 15 && 3+i11+5 = 3+i21+2 && 5+j11+1 = 5+j21-1
&& 3+i11+5 = 3+i22 && 5+j11+1 = 5+j22-1;
#
# T2;

[i11,i11+3,i11+5,j11,j11+2,j11+2]:  0 <= i11 <= 15 && 0 <= j11 <=
13

#
```

The line `[i11,i11+3,j11,j11+4]:  0 <= i11 <= 17 && 0 <= j11 <= 11` and `[i11,i11+3,i11+5,j11,j11+2,j11+2]:  0 <= i11 <= 15 && 0 <= j11 <= 13` of omega output ensures that dependency exists.

Parallelism Extraction

The purpose of parallelism extraction is to find out the reduction factor by which loop can speed up. In case of simple, selective and extended cycle shrinking for constant dependence distance, reduction factor is calculated for each loop. In example 12there are three reference pairs, so three distance vectors need to be calculated. Therefore, computation of distance vectors is given below:

For reference pair $X[i+8][j+6] - X[i+3][j+4] : < \phi_1^1, \phi_2^1 >=< 5, 2 >$

For reference pair $X[i+8][j+6] - X[i+5][j+4] : < \phi_1^2, \phi_2^2 >=< 3, 2 >$

For reference pair $Y[i+5][j+7] - Y[i+2][j+3] : < \phi_1^3, \phi_2^3 >=< 3, 4 >$

Therefore, reduction factor for simple, selective and extended cycle shrinking for constant dependence distance is $< \lambda_1, \lambda_2 >=< 3, 2 >$. While for true dependence shrinking reduction factor based on true distance. True distance for reference pairs $R_1 :\ X[i+8][j+6] - X[i+3][j+4]$, $R_2 :\ X[i+8][j+6] - X[i+5][j+4]$ and $R_3 :\ Y[i+5][j+7] - Y[i+2][j+3]$ are following respectively:

$$\Phi_{R1} = \phi_1^1(U_j - L_j) + \phi_2^1 = 3(15 - 0) + 2 = 47$$
$$\Phi_{R_2} = \phi_1^2(U_j - L_j) + \phi_2^2 = 5(15 - 0) + 2 = 77$$
$$\Phi_{R_3} = \phi_1^3(U_j - L_j) + \phi_2^3 = 3(15 - 0) + 4 = 49$$

Hence, reduction factor(λ) for true dependence is $min(\Phi_{R_1}, \Phi_{R_2}, \Phi_{R_3}) = 47$. For this example also, extended cycle shrinking for variable dependence distance is not applicable because of the same reason.

Code Transformation

Code transformation consists of various routines that generate the parallel CUDA code corresponding to the input C program. This phase also carries out the transformations required in various shrinking approach to reconstruct the loop. As all the dependence test i.e., GCD, Banerjee and Omega detected dependence in the loop, hence, irrespective of any user input for dependence test, the generated parallel code for any particular shrinking approach will not change. Therefore, Code 4.4, 4.5, 4.6 and 4.7 show the parallel CUDA code generated for user input `simple`, `selective`, `true_dependence` and `extShrinkingConst` respectively.

Code 4.4: Loop Transformation using Simple Shrinking

```
int _SZ_Y_2 = 150;
int _SZ_Y_1 = 150;
int _SZ_X_2 = 150;
int _SZ_X_1 = 150;
int *_DEV_Y;
// Allocating memory to Kernel Variable and copying them on device
cudaMalloc((void**) &_DEV_Y, sizeof(int)*_SZ_Y_2*_SZ_Y_1);
cudaMemcpy(_DEV_Y, Y, sizeof(int)*_SZ_Y_2*_SZ_Y_1,
    cudaMemcpyHostToDevice);
int *_DEV_X;
cudaMalloc((void**) &_DEV_X, sizeof(int)*_SZ_X_2*_SZ_X_1);
cudaMemcpy(_DEV_X, X, sizeof(int)*_SZ_X_2*_SZ_X_1,
```

```
    cudaMemcpyHostToDevice);
int _NUM_THREADS = 22500;
float _NUM_BLOCKS=1;
int _NUM_TILE=1;
dim3 _THREADS(512);
dim3 _BLOCKS(1);
// Tiling and declaring threads and blocks required for Kernel Execution
if(_NUM_THREADS < _NTHREAD)
{
        _THREADS.x=150;
        _THREADS.y=150;
}
else {
        _NUM_BLOCKS=(_NUM_THREADS*1.0)/256;
        _BLOCKS.x=_BLOCKS.y=ceil(sqrt(_NUM_BLOCKS));
        _THREADS.x=_THREADS.y=ceil(sqrt(22500.0/(_BLOCKS.x*_BLOCKS.y)));
        int temp=_NUM_BLOCKS;
        if(_NUM_BLOCKS>_NBLOCK)
                _NUM_TILE=(temp % _NBLOCK ==
                    0)?(_NUM_BLOCKS/_NBLOCK):((_NUM_BLOCKS/_NBLOCK)+1);
}
int _CUDA_TILE;
// Code transformation through Simple cycle shrinking
for(i=0;i<=137;i+=3)
for(j=0;j<=135;j+=2)
for(_CUDA_TILE=0;_CUDA_TILE<_NUM_TILE;_CUDA_TILE++){
        _AFFINE_KERNEL<<<_BLOCKS,_THREADS>>>(_DEV_Y, _SZ_Y_2, _SZ_Y_1,
            _DEV_X, _SZ_X_2, _SZ_X_1, 2, i, j, 0, 137, 0, 135, _CUDA_TILE);
        cudaDeviceSynchronize();}
// Copying Kernel variable from device to host
cudaMemcpy(Y, _DEV_Y, sizeof(int)*_SZ_Y_2*_SZ_Y_1,
```

```
    cudaMemcpyDeviceToHost);
cudaMemcpy(X, _DEV_X, sizeof(int)*_SZ_X_2*_SZ_X_1,
    cudaMemcpyDeviceToHost);
// Releasing the memory allocated to kernel variable
cudaFree(_DEV_Y);
cudaFree(_DEV_X);
```

Kernel definition for above parallel CUDA code:

```
__global__ void _AFFINE_KERNEL(int* Y,int _SZ_Y_2,int _SZ_Y_1,int* X,int
    _SZ_X_2,int _SZ_X_1,int phi_count, int CUDA_i, int CUDA_j, int
    CUDA_L_i,int CUDA_U_i, int CUDA_L_j,int CUDA_U_j, int _CUDA_TILE){
        int i = gridDim.x*blockDim.x*_CUDA_TILE + blockDim.x*blockIdx.x +
            threadIdx.x;
        int j = gridDim.y*blockDim.y*_CUDA_TILE + blockDim.y*blockIdx.y +
            threadIdx.y;
        if((CUDA_i<=i)&&(i<(CUDA_i+3))&&(i<=CUDA_U_i)){
        if((CUDA_j<=j)&&(j<(CUDA_j+2))&&(j<=CUDA_U_j)){
                X[(3+i+5)*_SZ_X_1+5+j+1]=Y[(3+i-1)*_SZ_Y_1+5+j-2]
        +X[(3+i)*_SZ_X_1+5+j-1];
                Y[(3+i+2)*_SZ_Y_1+5+j+2]=X[(3+i+2)*_SZ_X_1+5+j-1]+10;}}}
```

Code 4.5: Loop Transformation using Selective Shrinking

```
int _SZ_Y_2 = 150;
int _SZ_Y_1 = 150;
int _SZ_X_2 = 150;
int _SZ_X_1 = 150;
int *_DEV_Y;
// Allocating memory to Kernel Variable and copying them on device
cudaMalloc((void**) &_DEV_Y, sizeof(int)*_SZ_Y_2*_SZ_Y_1);
cudaMemcpy(_DEV_Y, Y, sizeof(int)*_SZ_Y_2*_SZ_Y_1,
    cudaMemcpyHostToDevice);
```

```
int *_DEV_X;
cudaMalloc((void**) &_DEV_X, sizeof(int)*_SZ_X_2*_SZ_X_1);
cudaMemcpy(_DEV_X, X, sizeof(int)*_SZ_X_2*_SZ_X_1,
    cudaMemcpyHostToDevice);
int _NUM_THREADS = 22500;
float _NUM_BLOCKS=1;
int _NUM_TILE=1;
dim3 _THREADS(512);
dim3 _BLOCKS(1);
// Tiling and declaring threads and blocks required for Kernel Execution
if(_NUM_THREADS < _NTHREAD){
        _THREADS.x=150;
        _THREADS.y=150;}
else {
        _NUM_BLOCKS=(_NUM_THREADS*1.0)/256;
        _BLOCKS.x=_BLOCKS.y=ceil(sqrt(_NUM_BLOCKS));
        _THREADS.x=_THREADS.y=ceil(sqrt(22500.0/(_BLOCKS.x*_BLOCKS.y)));
        int temp=_NUM_BLOCKS;
        if(_NUM_BLOCKS>_NBLOCK)
                _NUM_TILE=(temp % _NBLOCK ==
                    0)?(_NUM_BLOCKS/_NBLOCK):((_NUM_BLOCKS/_NBLOCK)+1);}
int _CUDA_TILE;
// Code transformation through Selective cycle shrinking
for(i=0;i<=137;i+=3)
for(_CUDA_TILE=0;_CUDA_TILE<_NUM_TILE;_CUDA_TILE++){
        _AFFINE_KERNEL<<<_BLOCKS,_THREADS>>>(_DEV_Y, _SZ_Y_2, _SZ_Y_1,
            _DEV_X, _SZ_X_2, _SZ_X_1, 2, i, j, 0, 137, 0, 135, _CUDA_TILE);
        cudaDeviceSynchronize();}
// Copying Kernel variable from device to host
cudaMemcpy(Y, _DEV_Y, sizeof(int)*_SZ_Y_2*_SZ_Y_1,
    cudaMemcpyDeviceToHost);
```

```
cudaMemcpy(X, _DEV_X, sizeof(int)*_SZ_X_2*_SZ_X_1,
    cudaMemcpyDeviceToHost);
// Releasing the memory allocated to kernel variable
cudaFree(_DEV_Y);
cudaFree(_DEV_X);
```

Kernel definition of above parallel code:

```
__global__ void _AFFINE_KERNEL(int* Y,int _SZ_Y_2,int _SZ_Y_1,int* X,int
    _SZ_X_2,int _SZ_X_1,int phi_count, int CUDA_i, int CUDA_j, int
    CUDA_L_i,int CUDA_U_i, int CUDA_L_j,int CUDA_U_j, int _CUDA_TILE){
        int i = gridDim.x*blockDim.x*_CUDA_TILE + blockDim.x*blockIdx.x +
            threadIdx.x;
        int j = gridDim.y*blockDim.y*_CUDA_TILE + blockDim.y*blockIdx.y +
            threadIdx.y;
        if((CUDA_i<=i)&&(i<(CUDA_i+3))&&(i<=CUDA_U_i)){
        if((CUDA_L_j<=j)&&(j<=CUDA_U_j)){
                X[(3+i+5)*_SZ_X_1+5+j+1]=Y[(3+i-1)*_SZ_Y_1+5+j-2]
        +X[(3+i)*_SZ_X_1+5+j-1];
                Y[(3+i+2)*_SZ_Y_1+5+j+2]=X[(3+i+2)*_SZ_X_1+5+j-1]+10;}}}
```

Code 4.6: Loop Transformation using True Dependence Shrinking

```
int _SZ_Y_2 = 150;
int _SZ_Y_1 = 150;
int _SZ_X_2 = 150;
int _SZ_X_1 = 150;
int *_DEV_Y;
// Allocating memory to Kernel Variable and copying them on device
cudaMalloc((void**) &_DEV_Y, sizeof(int)*_SZ_Y_2*_SZ_Y_1);
cudaMemcpy(_DEV_Y, Y, sizeof(int)*_SZ_Y_2*_SZ_Y_1,
    cudaMemcpyHostToDevice);
int *_DEV_X;
```

```
cudaMalloc((void**) &_DEV_X, sizeof(int)*_SZ_X_2*_SZ_X_1);
cudaMemcpy(_DEV_X, X, sizeof(int)*_SZ_X_2*_SZ_X_1,
    cudaMemcpyHostToDevice);
int _NUM_THREADS = 22500;
float _NUM_BLOCKS=1;
int _NUM_TILE=1;
dim3 _THREADS(512);
dim3 _BLOCKS(1);
// Tiling and declaring threads and blocks required for Kernel Execution
if(_NUM_THREADS < _NTHREAD){
        _THREADS.x=150;
        _THREADS.y=150;}
else{
        _NUM_BLOCKS=(_NUM_THREADS*1.0)/256;
        _BLOCKS.x=_BLOCKS.y=ceil(sqrt(_NUM_BLOCKS));
                _THREADS.x=_THREADS.y=ceil(sqrt(22500.0/(_BLOCKS.x*_BLOCKS.y)));
        int temp=_NUM_BLOCKS;
        if(_NUM_BLOCKS>_NBLOCK)
                _NUM_TILE=(temp % _NBLOCK ==
                    0)?(_NUM_BLOCKS/_NBLOCK):((_NUM_BLOCKS/_NBLOCK)+1);}
int _CUDA_TILE;
// Code transformation through True dependence shrinking
int lambda=407;
int id_1,id_2,id_3,id_4,id_5;
int UB_1=137-0;
int UB_2=135-0;
for(id_1=1;id_1<=(UB_1*UB_2);id_1+=lambda){
        id_2=(id_1/UB_2);
        id_3=((id_1+lambda)/UB_2);
        id_4=(id_1%UB_2)-1;
        id_5=UB_2-((id_1+lambda)%UB_2);
```

```
        for(_CUDA_TILE=0;_CUDA_TILE<_NUM_TILE;_CUDA_TILE++){
                _AFFINE_KERNEL_1<<<_BLOCKS,_THREADS>>>(_DEV_Y, _SZ_Y_2,
                    _SZ_Y_1, _DEV_X, _SZ_X_2, _SZ_X_1, id_2, id_4, UB_2,
                    _CUDA_TILE);
                cudaDeviceSynchronize();}
        for(_CUDA_TILE=0;_CUDA_TILE<_NUM_TILE;_CUDA_TILE++){
                _AFFINE_KERNEL<<<_BLOCKS,_THREADS>>>(_DEV_Y, _SZ_Y_2,
                    _SZ_Y_1, _DEV_X, _SZ_X_2, _SZ_X_1, id_2, id_3, 0, 135,
                    _CUDA_TILE);
                cudaDeviceSynchronize();}
        for(_CUDA_TILE=0;_CUDA_TILE<_NUM_TILE;_CUDA_TILE++){
                _AFFINE_KERNEL_1<<<_BLOCKS,_THREADS>>>(_DEV_Y, _SZ_Y_2,
                    _SZ_Y_1, _DEV_X, _SZ_X_2, _SZ_X_1, id_3, 0, id_5,
                    _CUDA_TILE);
                cudaDeviceSynchronize();}}
// Copying Kernel variable from device to host
cudaMemcpy(Y, _DEV_Y, sizeof(int)*_SZ_Y_2*_SZ_Y_1,
    cudaMemcpyDeviceToHost);
cudaMemcpy(X, _DEV_X, sizeof(int)*_SZ_X_2*_SZ_X_1,
    cudaMemcpyDeviceToHost);
// Releasing the memory allocated to kernel variable
cudaFree(_DEV_Y);
cudaFree(_DEV_X);
```

Kernel definition for above parallel CUDA code:

```
__global__ void _AFFINE_KERNEL_1(int* Y,int _SZ_Y_2,int _SZ_Y_1,int*
    X,int _SZ_X_2,int _SZ_X_1,int i, int id_1, int UB_1, int _CUDA_TILE){
        int j = gridDim.y*blockDim.y*_CUDA_TILE + blockDim.y*blockIdx.y +
            threadIdx.y;
        if((id_1<=j)&&(j<=UB_1)){
                X[(3+i+5)*_SZ_X_1+5+j+1]=Y[(3+i-1)*_SZ_Y_1+5+j-2]
```

```
        +X[(3+i)*_SZ_X_1+5+j-1];
                Y[(3+i+2)*_SZ_Y_1+5+j+2]=X[(3+i+2)*_SZ_X_1+5+j-1]+10;}}


__global__ void _AFFINE_KERNEL(int* Y,int _SZ_Y_2,int _SZ_Y_1,int* X,int
    _SZ_X_2,int _SZ_X_1,int CUDA_L_i,int CUDA_U_i, int CUDA_L_j,int
    CUDA_U_j, int _CUDA_TILE){
        int i = gridDim.x*blockDim.x*_CUDA_TILE + blockDim.x*blockIdx.x +
            threadIdx.x;
        int j = gridDim.y*blockDim.y*_CUDA_TILE + blockDim.y*blockIdx.y +
            threadIdx.y;
        if((CUDA_L_i<=i)&&(i<=CUDA_U_i)){
        if((CUDA_L_j<=j)&&(j<=CUDA_U_j)){
                X[(3+i+5)*_SZ_X_1+5+j+1]=Y[(3+i-1)*_SZ_Y_1+5+j-2]
        +X[(3+i)*_SZ_X_1+5+j-1];
                Y[(3+i+2)*_SZ_Y_1+5+j+2]=X[(3+i+2)*_SZ_X_1+5+j-1]+10;}}}
```

Code 4.7: Loop Transformation using Extended Cycle Shrinking for Constant Dependence Distance

```
int _SZ_Y_2 = 150;
int _SZ_Y_1 = 150;
int _SZ_X_2 = 150;
int _SZ_X_1 = 150;
int *_DEV_Y;
// Allocating memory to Kernel Variable and copying them on device
cudaMalloc((void**) &_DEV_Y, sizeof(int)*_SZ_Y_2*_SZ_Y_1);
cudaMemcpy(_DEV_Y, Y, sizeof(int)*_SZ_Y_2*_SZ_Y_1,
    cudaMemcpyHostToDevice);
int *_DEV_X;
cudaMalloc((void**) &_DEV_X, sizeof(int)*_SZ_X_2*_SZ_X_1);
cudaMemcpy(_DEV_X, X, sizeof(int)*_SZ_X_2*_SZ_X_1,
    cudaMemcpyHostToDevice);
```

```
int _NUM_THREADS = 22500;
float _NUM_BLOCKS=1;
int _NUM_TILE=1;
dim3 _THREADS(512);
dim3 _BLOCKS(1);
// Tiling and declaring threads and blocks required for Kernel Execution
if(_NUM_THREADS < _NTHREAD){
        _THREADS.x=150;
        _THREADS.y=150;}
else{
        _NUM_BLOCKS=(_NUM_THREADS*1.0)/256;
        _BLOCKS.x=_BLOCKS.y=ceil(sqrt(_NUM_BLOCKS));
        _THREADS.x=_THREADS.y=ceil(sqrt(22500.0/(_BLOCKS.x*_BLOCKS.y)));
        int temp=_NUM_BLOCKS;
        if(_NUM_BLOCKS>_NBLOCK)
                _NUM_TILE=(temp % _NBLOCK ==
                    0)?(_NUM_BLOCKS/_NBLOCK):((_NUM_BLOCKS/_NBLOCK)+1);}
// Code transformation through Extended cycle shrinking for constant
    dependence distance
int ID_1, ID_2, START[2];
int _CUDA_TILE;
int Phi[2]={3, 2};
int loopUpperLimits[2]={137, 135};
for(ID_1=1;ID_1<=MIN(137/3, 135/2)+1;ID_1++){
        for(ID_2=0;ID_2<2;ID_2++){
                if(Phi[ID_2]>=0)
                        START[ID_2]=(ID_1-1)*Phi[ID_2];
                else
                        START[ID_2]=loopUpperLimits[ID_2]+(ID_1-1)*Phi[ID_2];}
        for(_CUDA_TILE=0;_CUDA_TILE<_NUM_TILE;_CUDA_TILE++){
                _AFFINE_KERNEL<<<_BLOCKS,_THREADS>>>(_DEV_Y, _SZ_Y_2,
```

```
                _SZ_Y_1, _DEV_X, _SZ_X_2, _SZ_X_1, START[0],
                MIN(START[0]+3, 137), START[1], 135, _CUDA_TILE);
            cudaDeviceSynchronize();}
        for(_CUDA_TILE=0;_CUDA_TILE<_NUM_TILE;_CUDA_TILE++){
            _AFFINE_KERNEL<<<_BLOCKS,_THREADS>>>(_DEV_Y, _SZ_Y_2,
                _SZ_Y_1, _DEV_X, _SZ_X_2, _SZ_X_1, START[0]+3, 137,
                START[1], MIN(START[1]+2, 135), _CUDA_TILE);
            cudaDeviceSynchronize();}}
// Copying Kernel variable from device to host
cudaMemcpy(Y, _DEV_Y, sizeof(int)*_SZ_Y_2*_SZ_Y_1,
    cudaMemcpyDeviceToHost);
cudaMemcpy(X, _DEV_X, sizeof(int)*_SZ_X_2*_SZ_X_1,
    cudaMemcpyDeviceToHost);
// Releasing the memory allocated to kernel variable
cudaFree(_DEV_Y);
cudaFree(_DEV_X);
```

Kernel definition for above code:

```
__global__ void _AFFINE_KERNEL(int* Y,int _SZ_Y_2,int _SZ_Y_1,int* X,int
    _SZ_X_2,int _SZ_X_1,int CUDA_L_i,int CUDA_U_i, int CUDA_L_j,int
    CUDA_U_j, int _CUDA_TILE){
        int i = gridDim.x*blockDim.x*_CUDA_TILE + blockDim.x*blockIdx.x +
            threadIdx.x;
        int j = gridDim.y*blockDim.y*_CUDA_TILE + blockDim.y*blockIdx.y +
            threadIdx.y;
        if((CUDA_L_i<=i)&&(i<=CUDA_U_i)){
        if((CUDA_L_j<=j)&&(j<=CUDA_U_j)){
                X[(3+i+5)*_SZ_X_1+5+j+1]=Y[(3+i-1)*_SZ_Y_1+5+j-2]
        +X[(3+i)*_SZ_X_1+5+j-1];
                Y[(3+i+2)*_SZ_Y_1+5+j+2]=X[(3+i+2)*_SZ_X_1+5+j-1]+10;}}}
```

4.3 Multi-Nested Loops with Variable Dependence Distance

Till now, example for extended cycle shrinking for variable dependence distance remains uncovered. The above two examples only cover simple, selective, true dependence shrinking and extended cycle shrinking for constant dependence distance. Therfore, here we unveil the example for extended cycle shrinking for variable dependence shrinking:

Example 13.
```
for(i=1;i<=21;i++)
  for(j=1;j<=21;j++)
    X[3*i+5][2*j+4]=Y[2*i-2][j+2]+23;
    Y[3*i+4][3*j+4]=X[i-1][j+4];
```

Generating parallel CUDA code for the program containing this loop will pass through the following phases:

Compilation Phase

The compilation phase of this example takes place in a same way as it happened for example 11 and 12. After lexical, syntactic analysis and data extraction, loop normalization is carried out. Since the lower limit of the loop in this example is non-zero, it is non-normalized. After normalization this example becomes:

```
for(i=0;i<=20;i++)
  for(j=0;j<=20;j++)
    X[3*i+8][2*j+6]=Y[2*i][j+3]+23;
    Y[3*i+7][3*j+7]=X[i][j+5];
```

This loop is also affine because all the array indices are linear functions of loop variable.

Dependence test

Among GCD, banerjee and omega any of the test can be used to deduce dependency. There are two reference pairs in the normalized form of example 13 i.e., $R_1 : X[3 *$

$i+5][2*j+4] - X[i][j+5]$ and $R_2 : Y[3*i+4][3*j+4] - Y[2*i][j+3]$. Linear equation for reference pair R_1 is given below:

$$i: \ 3*i_1 + 5 = i_2$$
$$j: \ 2*j_1 + 4 = j_2 + 5$$

Comparing these two linear equations with equation 4.1, *gcd(a,b)* evenly divides *(d-c)*. Now linear equation for reference pair R_2 becomes as follows:

$$i: \ 3*i_1 + 4 = 2*i_2$$
$$j: \ 3*j_1 + 4 = j_2 + 3$$

For this equation also *gcd(a,b)* divides *(d-c)*, hence dependence exists in loop because of both X and Y array. Therefore, if user input is `gcd`, then it detects dependence in the loop. To detect dependence using banerjee test, upper and lower bounds for every reference pair needs to be calculated. Following are the lower and upper bounds with respect to direction vector $(<, =, >)$. For reference pair R_1,

$$LB_i^{<} = -20 \leq B_0 - A_0 = 8 \leq UB_i^{<} = 37$$
$$LB_i^{=} = 0 \leq B_0 - A_0 = 8 \leq UB_i^{=} = 40$$
$$LB_i^{<} = 3 \leq B_0 - A_0 = 8 \leq UB_i^{>} = 60$$
$$LB_j^{<} = -20 \leq B_0 - A_0 = 1 \leq UB_j^{<} = 18$$
$$LB_j^{=} = 0 \leq B_0 - A_0 = 1 \leq UB_j^{=} = 20$$
$$LB_j^{<} = 2 \leq B_0 - A_0 = 1 \leq UB_j^{>} = 40$$

For refernece pair R_2,

$$LB_i^{<} = -40 \leq B_0 - A_0 = 7 \leq UB_i^{<} = 17$$
$$LB_i^{=} = 0 \leq B_0 - A_0 = 7 \leq UB_i^{=} = 20$$
$$LB_i^{<} = 3 \leq B_0 - A_0 = 7 \leq UB_i^{>} = 60$$
$$LB_j^{<} = -20 \leq B_0 - A_0 = 4 \leq UB_j^{<} = 37$$
$$LB_j^{=} = 0 \leq B_0 - A_0 = 4 \leq UB_j^{=} = 40$$
$$LB_j^{<} = 3 \leq B_0 - A_0 = 4 \leq UB_j^{>} = 60$$

As the value of $(B_0 - A_0)$ lies within each bound constraints of R_1 and R_2, therefore dependency exists.

If user opts for `omega`, then *omega_input.in* file is generated that acts as an input for omega calculator:

```
T1 := [i11,i21,j11,j21]:  0 <= i11,i21 <= 20 && 0 <= j11,j21 <= 20
&& 3+3*i11+4 = 2+2*i21-2 && 3+3*j11+4 = 1+j21+2;
T1;
T2 := [i11,i21,j11,j21]:  0 <= i11,i21 <= 20 && 0 <= j11,j21 <= 20
&& 3+3*i11+5 = 1+i21-1 && 2+2*j11+4 = 1+j21+4;
T2;
```

The output of omega calculator will be:

```
# Omega Calculator v2.1 (based on Omega Library 2.1, July, 2008):
# T1 := [i11,i21,j11,j21]:  0 <= i11,i21 <= 20 && 0 <= j11,j21 <= 20
&& 3+3*i11+4 = 2+2*i21-2 && 3+3*j11+4 = 1+j21+2;
#
# T1;

[i11,i21,j11,3j11+4]:  7+3i11 = 2i21 && 5 <= i21 <= 20
&& 0 <= j11 <= 5

#
# T2 := [i11,i21,j11,j21]:  0 <= i11,i21 <= 20 && 0 <= j11,j21 <= 20
&& 3+3*i11+5 = 1+i21-1 && 2+2*j11+4 = 1+j21+4;
#
# T2;

[i11,3i11+8,j11,2j11+1]:  0 <= i11 <= 4 && 0 <= j11 <= 9

#
```

The line [i11,i21,j11,3j11+4]: 7+3i11 = 2i21 && 5 <= i21 <= 20 && 0 <= j11 <= 5 and [i11,3i11+8,j11,2j11+1]: 0 <= i11 <= 4 && 0 <= j11 <= 9 ensure that dependency exists.

Parallelism Extraction

The array indices of example 13 belong to the form $(ai \pm b)$. This implies that the dependence distance varies with the change in loop iteration. Hence simple, selective, true dependence shrinking and extended cycle shrinking for constant dependence distance cannot be applied to this example as these approaches are applicable to the loop containing constant dependence distance. Therefore, only extended cycle shrinking for variable dependence distance is applicable for this example. To extract parallelism out of the loop, data dependence vectors(DDV) are required for each reference pair. As already discussed in Chapter 2, data dependence vector for refernce pair $S_i : A[a_{10} + a_{11}I_1 + .. + a_{1m}I_m, .., a_{m0} + a_{m1}I_1 + .. + a_{mm}I_m]$ - $S_j : A[b_{10} + b_{11}I_1 + .. + b_{1m}I_m, .., b_{m0} + b_{m1}I_1 + .. + b_{mm}I_m]$ is calculated by below equation:

$$\phi_k = \frac{(a_{k0} - b_{k0}) + \sum_{i=1, i\neq k}^{m} + (a_{kk} - b_{kk})}{b_{kk}} \tag{4.2}$$

Using equation 4.2, DDV for reference pairs $R_1 : X[3 * i + 5][2 * j + 4] - X[i][j + 5]$ and $R_2 : Y[3 * i + 4][3 * j + 4] - Y[2 * i][j + 3]$ are calculated:

$$R_1 : < \phi_1^1, \phi_2^1 > = < \frac{(5-0)+(3-1)}{1}, \frac{(4-5)+(3-2)}{1} >$$
$$R_2 : < \phi_1^2, \phi_2^2 > = < \frac{(4-0)+(3-2)}{2}, \frac{(4-3)+(3-1)}{1} >$$

Code Transformation

As specified in above phase, only extended cycle shrinking for variable dependence distance can be used to extract parallelism. Therefore, only user input `extShrinkingVar` can be used to generate parallel CUDA code for the given sequential C program while for other user input such as `simple`, `selective`, `true_dependence` and `extShrinkingConst`, a warning message gets generated saying "Oops!! Code Generation for Variable Dependence Distance can only happen through Extended Cycle Shrinking". Code 4.8 shows the code generated by extended cycle shrinking for variable dependence distance.

Code 4.8: Loop Transformation using Extended Cycle Shrinking for Variable Dependence Distance

```
int _SZ_Y_2 = 300;
int _SZ_Y_1 = 300;
int _SZ_X_2 = 300;
int _SZ_X_1 = 300;
int *_DEV_Y;
// Allocating memory to Kernel Variable and copying them on device
cudaMalloc((void**) &_DEV_Y, sizeof(int)*_SZ_Y_2*_SZ_Y_1);
cudaMemcpy(_DEV_Y, Y, sizeof(int)*_SZ_Y_2*_SZ_Y_1,
    cudaMemcpyHostToDevice);
int *_DEV_X;
cudaMalloc((void**) &_DEV_X, sizeof(int)*_SZ_X_2*_SZ_X_1);
cudaMemcpy(_DEV_X, X, sizeof(int)*_SZ_X_2*_SZ_X_1,
    cudaMemcpyHostToDevice);
int _NUM_THREADS = 90000;
float _NUM_BLOCKS=1;
int _NUM_TILE=1;
dim3 _THREADS(512);
dim3 _BLOCKS(1);
// Tiling and declaring threads and blocks required for Kernel Execution
if(_NUM_THREADS < _NTHREAD){
        _THREADS.x=300;
        _THREADS.y=300;}
else{
        _NUM_BLOCKS=(_NUM_THREADS*1.0)/256;
        _BLOCKS.x=_BLOCKS.y=ceil(sqrt(_NUM_BLOCKS));
        _THREADS.x=_THREADS.y=ceil(sqrt(90000.0/(_BLOCKS.x*_BLOCKS.y)));
        int temp=_NUM_BLOCKS;
        if(_NUM_BLOCKS>_NBLOCK)
                _NUM_TILE=(temp % _NBLOCK ==
```

```
                0)?(_NUM_BLOCKS/_NBLOCK):((_NUM_BLOCKS/_NBLOCK)+1);}
int _CUDA_TILE;
// Code transformation through Extended cycle shrinking for variable
   dependence distance
int ID_1=0, next_ID_1, ID_2=0, next_ID_2;
while((ID_1<79)&&(ID_2<89)){
       next_ID_1 = MIN((((8)+(2)*ID_1+(0)*ID_2)/(1)),
           (((7)+(1)*ID_1+(0)*ID_2)/(2)));
       next_ID_2 = MIN((((1)+(0)*ID_1+(1)*ID_2)/(1)),
           (((4)+(0)*ID_1+(2)*ID_2)/(1)));
       for(_CUDA_TILE=0;_CUDA_TILE<_NUM_TILE;_CUDA_TILE++){
              _AFFINE_KERNEL<<<_BLOCKS,_THREADS>>>(_DEV_Y, _SZ_Y_2,
                  _SZ_Y_1, _DEV_X, _SZ_X_2, _SZ_X_1, ID_1, MIN(next_ID_1,
                  79), ID_1, 89, _CUDA_TILE);
              cudaDeviceSynchronize();}
       for(_CUDA_TILE=0;_CUDA_TILE<_NUM_TILE;_CUDA_TILE++){
              _AFFINE_KERNEL<<<_BLOCKS,_THREADS>>>(_DEV_Y, _SZ_Y_2,
                  _SZ_Y_1, _DEV_X, _SZ_X_2, _SZ_X_1, next_ID_1, 79, ID_2,
                  MIN(next_ID_2, 89), _CUDA_TILE);
              cudaDeviceSynchronize();}
       ID_1=next_ID_1;
       ID_2=next_ID_2;}
// Copying Kernel variable from device to host
cudaMemcpy(Y, _DEV_Y, sizeof(int)*_SZ_Y_2*_SZ_Y_1,
   cudaMemcpyDeviceToHost);
cudaMemcpy(X, _DEV_X, sizeof(int)*_SZ_X_2*_SZ_X_1,
   cudaMemcpyDeviceToHost);
// Releasing the memory allocated to kernel variable
cudaFree(_DEV_Y);
cudaFree(_DEV_X);
```

Kernel definition of above parllel code:

```
__global__ void _AFFINE_KERNEL(int* Y,int _SZ_Y_2,int _SZ_Y_1,int* X,int
    _SZ_X_2,int _SZ_X_1,int CUDA_L_i,int CUDA_U_i, int CUDA_L_j,int
    CUDA_U_j, int _CUDA_TILE){
        int i = gridDim.x*blockDim.x*_CUDA_TILE + blockDim.x*blockIdx.x +
            threadIdx.x;
        int j = gridDim.y*blockDim.y*_CUDA_TILE + blockDim.y*blockIdx.y +
            threadIdx.y;
        if((CUDA_L_i<=i)&&(i<=CUDA_U_i)){
        if((CUDA_L_j<=j)&&(j<=CUDA_U_j)){
                X[(3+3*i+5)*_SZ_X_1+2+2*j+4]=Y[(2+2*i-2)*_SZ_Y_1+1+j+2]+23;
                Y[(3+3*i+4)*_SZ_Y_1+3+3*j+4]=X[(1+i-1)*_SZ_X_1+1+j+4];}}}
```

Chapter 5

Literature Survey

This chapter gives the survey of related work done in the domain of this thesis. It list out the research performed in automatic parallelization, loop transformation, code generation. Also these work helps in finding out the area that is remain unexplored in all of them.

5.1 Automatic Parallelization

Parallel programming starts with the emergence of computing. The need of parallelization arises because of the application that contains loops, array references and takes huge computation time when executed sequentially. A lot of research has been done in the field of parallelizing compilers and automatic parallelization of programs. Earlier the work was mainly performed for the program written in FORTRAN [ZBG88, LP98, AK87] but later on because of the emerging need of industry towards C/C++ programming languages considerable research are done towards automatic parallelization of the programs written in these language. Automatic parallelization get widespread because it does not requires any effort from the programmer side for parallelization and optimization of programs.

5.1.1 Polyhedral Transformation Framework

Polyhedral compiler framework provides a powerful framework for program analysis and transformation. Polyhedral model identify the transformation to be performed on the nested loops by viewing the statement instance as an integer point in a well define space called polyhedron of statement. Various automatic parallelization techniques [Bon13, BBH+08, BBK+08b, PBB+10] uses polyhedral model for program transformation. Tools like PLUTO[BHRS08, Rud10, VCJC+13] provides a fully automatic polyhedral source to source transformation system that optimize regular program for paralllelization and locality. Some research [PBB+10] combines the iterative and model driven optimization to present a search space which is expressed using polyhedral model. This search space contains complex loop transformation like loop fusion, loop tiling, loop interchanging and loop shifting.

5.1.2 Loop Transformation Framework

Many compute intensive application spend their execution time in nested loops. One way to parallelize these loops is to perform various transformations like loop unrolling, interchange, skewing, distribution, reversal. A lot of research [LP94, AGMM00] has been done in the field of automatic parallelization using loop transformation. Hall et al. [HCC+10] proposed loop transformation recipes which acts as an high level interface to code transformation and code generation. This interface is a part of auto-tuning framework that presents a set of implementation for a computation from which it automatically selects the best implementation. M.F.P. O'Boyle and P.M.W. Knijnenburg [OK99] develops an optimization framework that combines the data and loop transformation to minimize parallelization overhead on distributed shared memory machines. Their work also proves that only data or only loop transformation will not be able to give efficient parallelization.

5.1.3 Other transformation framework

Other than polyhedral and loop transformation framework there are several other transformation framework [CA04, VRDB10, KBB+07, BBK+08a, PBB+10, GZA+11, GT14] that helps in automatic parallelization. Tranformations are not only for the FORTRAN or industry standard C/C++ programming language but there are frameworks for other languages like for JAVA [CA04]. Automatic parallelization are required for the complex nested loops involves in various applications. Parallelization for stencil computation [KBB+07] focuses on the issues affecting parallel execution of tiled iteration space that are embedded into the perfectly nested iteration space. It introduce the concept of overlapped tiling and split tiling that eliminate the inter-tile dependencies. Vandierendonck et al. proposed a parallax infrastructure [VRDB10] to parallelize irregular pointer intensive applications. They presented a light weight programming model to identify the thread level parallelism. It adds annotation to the program that describe its properties which a static compiler cannot find out. Also several research has been done in optimizing affine loop nests for different memory architectures [Bon13, BBK+08a].

5.2 Automatic Code Generation for GPGPUs

Because of the demand of multiple processing element on a single chip, the need of multicore architecture emerges out. Graphic processing unit(GPUs) are one of them and a powerful computation system. Traditionally GPUs only handles the graphics related application, but now they are going to be used for general purpose computations called as GPGPU. Even though GPGPU provides a parallel system to application developers but programming the application is complex in GPGPU. Many programming models are presented for application developement so that they can run on GPUs like CUDA[SK10], OpenCL[MGMG11] etc. But manual developement of the parallel code using these programming model is still cumbersome than the parallel programming language like OpenMP. Several research has

been done in the area of code generation framework but they all suffered from the scalability challenge. Baskaran et al.[BRS10] proposed a fully automatic source to source code transformation system from c to CUDA for affine programs. While Lee et al.[LME09, LE10] presented a compiler framework to perform program optimization and translation of OpenMP application into CUDA based GPGPU application. They have taken the advantage of OpenMP programming paradigm for CUDA programming. CUDA-CHILL [Rud10] proposed a compiler framework and code generation for optimization of sequential loops. Different from the above works this thesis develops an end to end code transformation tool that convert and optimize sequential regular or irregular programs written in C language to generate efficient parallel CUDA program.

Chapter 6

Performance Evaluation

This chapter outlines the experimental evaluation of CRINK . Design and implementation of which has already been discussed in Chapter 3 and 4. This chapter will discuss the results of various experiment performed on code generated by CRINK . The generated CUDA code is executed on the GPU machines and their performance in terms of execution time are illustrated through graphs.

6.1 Experimental Setup

The performance of generated parallel CUDA code is tested on GPU platform. The GPU machine used for experiment is *Tesla C1060* which is Nvidia's third brand of GPUs. Tesla C1060 has 240 cores and 4GB memory with compute capability 1.3. All the experiments are performed on *gpu01.cc.iitk.ac.in*, *gpu02.cc.iitk.ac.in* and *gpu04.cc.iitk.ac.in* server which has nvcc as Nvidia CUDA compiler installed on it. The tool has been tested for various standard benchmarks and some other kernels with the help of some standard datasets. The testing of various kernel have also put *reduction factor* into consideration i.e. experiments has been performed for various kernel by modifying the reduction factor.

6.2 Testing Setup

The tool has been tested for the loops taken from Higbie (the collection of vectorizable loops) [Hig], various standard programs available on [jbu] and standard benchmarks SPEC. Since for each kernel different parallel CUDA code can be generated, following factors are taken into considerations while evaluating results for each kernel:

- User input for dependence test i.e `gcd`, `banerjee` and `omega`. Because for a particular program one test may detect dependence and other may not, therefore different parallel code will be generated.
- If dependence exists, then depending upon the user input for parallelism extraction i.e. `simple`, `selective`, `true_dependence`, `extShrinkingConst` and `extShrinkingVar` different parallel code gets generated.

6.3 Standard Datasets

To test the performance of our tool, we have used the standard datasets [Dat]. These datasets are the two dimensional sparse matrix collection created by University of Florida. The dataset used in this thesis are listed below:

1. PajekCSphd [Paj]
2. PajekEVA [Paj]
3. PajekHEP-th-new [Paj]
4. SNAPp2pGnutella04 [SNA]
5. Nasabarth5 [Nas]
6. BarabasiNotreDame_www [Bar]

6.4 Performance Analysis

This section will inspect each standard kernel and its corresponding parallel CUDA code generated by CRINK . It will examine the result when the CUDA code is executed on GPUs for different standard dataset and for different λ value.

Table 6.1 list out the standard benchmarks used in this thesis and gives the corresponnding description:

Benchmarks	Description
ZERO_RC	C library to find out the solution for the system of non-linear equation
Higbie	Collection of various vectorizable loops written by {Lee Higbie}
Treepack	C library to performs some calculation on tree
Sandia Rules	C library to generates a wide range of quadrature of various orders
SPEC	Benchmark designed to test the CPU performance.

Table 6.1: Standard benchmarks used

Table 6.2 shows the performance of various benchmarks on standard datasets. It presents the execution time for different parallelism extraction methods correspond to each benchmark. The λ value indicates the reduction factor with respect to corresponding cycle shrinking approach. As discussed earlier in Chapter 2, calculation of λ value for true dependence shrinking depends upon the *true distance* and for simple, selective and extended cycle shrinking for constant dependence distance depends on *distance vectors*. In Table 6.2, *for loop* ID refers to the benchmark file line number in which the for loop occurs.

Benchmarks	For Loop ID	Dependence Test	Dependency Exists	Cycle Shrinking	λ value	Dataset	Runtime (Threads x Blocks)		
							128x1	512x8	512x64
ZERO_RC	zero_rc.c :123	GCD, Banerjee, Omega	Yes	Simple shrinking	2	CSphd	0.16269	0.01262	0.01262
						EVA	2.37350	0.10438	0.04872
						barth5	198.46145	6.90151	0.98608
						p2p-Gnutella04	94.41442	2.80765	0.63796
						HEP-th-new	6488.75220	215.77367	30.74060
						NotreDame_www	44899.2898	5886.7611	213.7435
				Extended cycle shrinking for constant dependence distance	2	CSphd	0.11505	0.02141	0.02141
						EVA	1.55880	0.06632	0.06659
						barth5	127.49446	4.40064	0.53989
						p2p-Gnutella04	54.45787	1.74379	0.36136
						HEP-th-new	4214.08422	131.96046	16.80648
						NotreDame_www	28996.88856	916.99014	116.73602
Higbie	higbie:1510	GCD, Banerjee, Omega	Yes	Simple shrinking	<NV2,NV3>	CSphd	0.05786	0.01839	0.01839
						EVA	0.93374	0.12417	0.07155
						barth5	1.23790	0.15600	0.02076
						p2p-Gnutella04	40.00787	5.01940	0.65543
						HEP-th-new	8847.3863	1075.5344	137.8398
						NotreDame_www	130085.6058	Time out	Time out
				Selective shrinking	NV2*M	CSphd	0.00389	0.00124	0.00124
						EVA	0.03602	0.00504	0.00023
						barth5	1.20648	0.15084	0.02076
						p2p-Gnutella04	0.56067	0.07453	0.00955
						HEP-th-new	40.85149	5.10508	0.65994
						NotreDame_www	380.7156	47.39323	5.98502
				True dependence shrinking	NV2*M+NV3	CSphd	0.01109	0.00347	0.00347
						EVA	0.09166	0.01275	0.00738
						barth5	3.55874	0.44109	0.06383
						p2p-Gnutella04	1.69831	0.20951	0.02956
						HEP-th-new	114.52113	14.34307	1.91028
						NotreDame_www	1041.74659	131.24814	16.76237
				Extended cycle shrinking for constant dependence distance	<NV2,NV3>	CSphd	0.00569	0.00180	0.00180
						EVA	0.04542	0.00628	0.00366
						barth5	1.73036	0.21479	0.02937
						p2p-Gnutella04	0.82107	0.10267	0.01360
						HEP-th-new	54.94935	6.87558	0.88530
						NotreDame_www	496.44444	62.01346	7.82824
Treepack	treepack.c :3543	GCD, Banerjee, Omega	Yes	Simple shrinking	2	CSphd	0.10784	0.01524	0.01524
						EVA	1.57178	0.05871	0.05944
						barth5	127.38594	4.24979	0.55260
						p2p-Gnutella04	53.56174	1.71252	0.34366
						HEP-th-new	4207.57740	131.49243	16.66710
						NotreDame_www	29049.41600	910.74710	115.81394
				Extended cycle shrinking for constant dependence distance	2	CSphd	0.16126	0.024619	0.024619
						EVA	2.36471	0.09487	0.09474
						barth5	195.29583	6.84860	0.95559
						p2p-Gnutella04	82.40414	2.79328	0.62027
						HEP-th-new	6437.51152	213.02812	30.07215
						NotreDame_www	44559.91662	1463.56947	1465.056

Benchmarks	For Loop ID	Dependence Test	Dependency Exists	Cycle Shrinking	λ value	Dataset	Runtime (Threads x Blocks)		
							128x1	512x8	512x64
SANDIA _RULES	sandia_ rules.c:6571	Banerjee	No	NA	NA	CSphd	0.00020	0.00003	0.00003
						EVA	0.00073	0.00004	0.000048
						barth5	0.00648	0.00027	0.00008
						p2p-Gnutella04	0.00421	0.00017	0.00007
						HEP-th-new	0.03725	0.00143	0.00039
						NotreDame_ www	0.09727	0.003432	0.00073
SPEC	blocksort.c :842	GCD, Banerjee, Omega	Yes	Simple shrinking	nblock	CSphd	0.00080	0.00011	0.00011
						EVA	0.00797	0.00024	0.00024
						barth5	0.50741	0.01712	0.00215
						p2p-Gnutella04	0.21617	0.00705	0.00139
						HEP-th-new	16.81098	0.52985	0.06743
						NotreDame_ www	116.26934	3.66369	0.46903
				Extended cycle shrinking for constant dependence distance	nblock	CSphd	0.00072	0.000131	
						EVA	0.00933	0.00040	0.00020
						barth5	0.72144	0.02568	0.00364
						p2p-Gnutella04	0.31060	0.01044	0.00237
						HEP-th-new	23.58767	0.78993	0.11371
						NotreDame_ www	162.97816	5.42086	0.78510

Table 6.2: Performance analysis for different configuration of standard benchmarks and datasets

6.4.1 Results for ZERO_RC Benchmark

For ZERO_RC benchmark, the referenced array in loop is single dimensional. Therefore, simple shrinking and extended cycle shrinking for constant dependence distance are used for parallelism extraction.

Fig 6.1 - 6.6 shows the performance of ZERO_RC benchmark on standard datasets when simple shrinking is used.

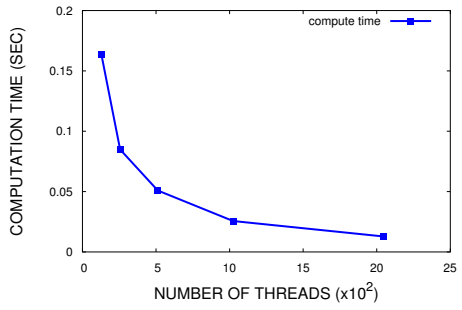


Figure 6.1: Plot for CSphd

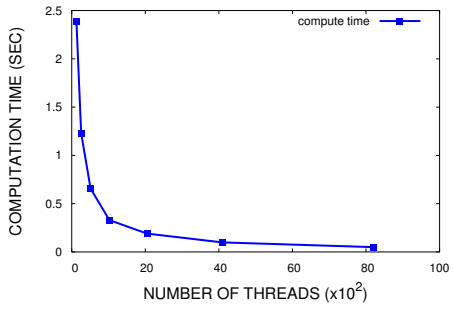


Figure 6.2: Plot for EVA

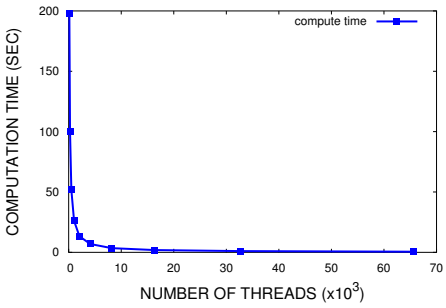


Figure 6.3: Plot for barth5

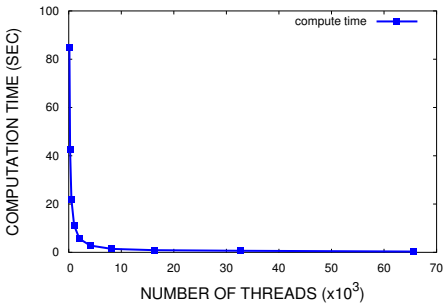


Figure 6.4: Plot for p2p-Gnutella04

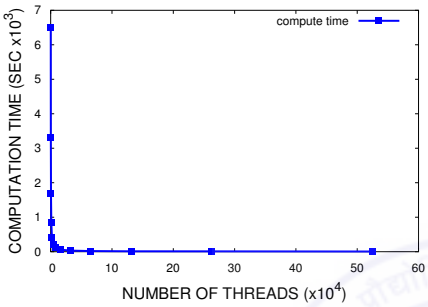


Figure 6.5: Plot for HEP-th-new

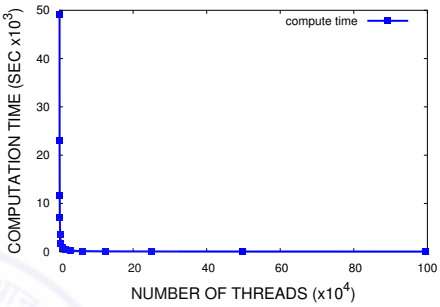


Figure 6.6: Plot for NotreDame_www

Fig 6.7 - 6.12 shows the performance of ZERO_RC benchmark when extended cycle shrinking is used for parallelism extraction.

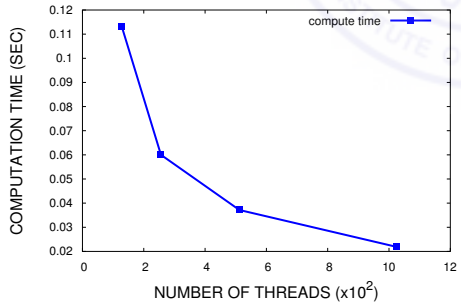


Figure 6.7: Plot for CSphd

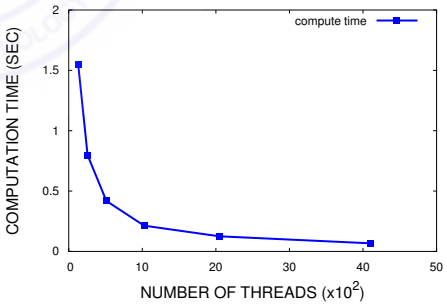


Figure 6.8: Plot for EVA

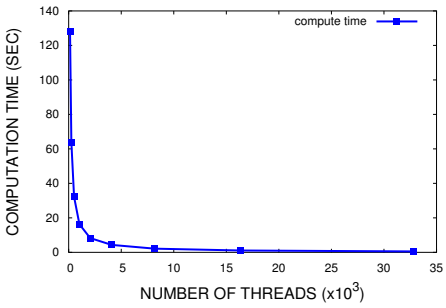


Figure 6.9: Plot for barth5

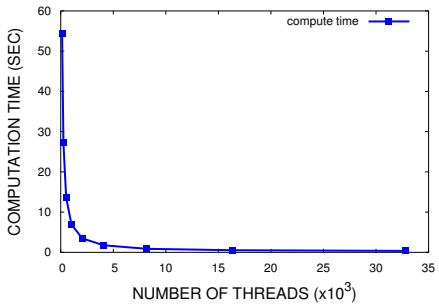


Figure 6.10: Plot for p2p-Gnutella04

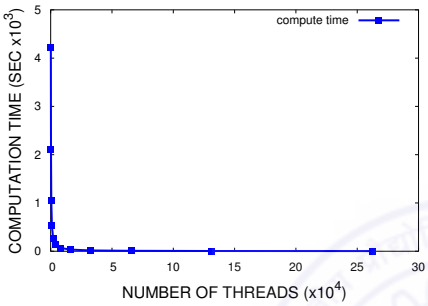


Figure 6.11: Plot for HEP-th-new

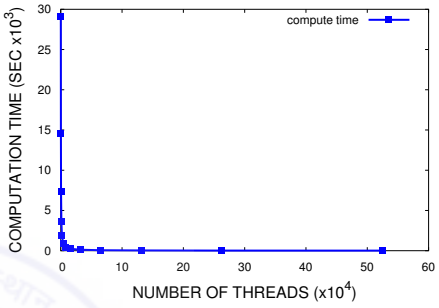


Figure 6.12: Plot for NotreDame_www

6.4.2 Results for Higbie Benchmark

The loop that we have considered in higbie consists of dependence detected by all the dependence test i.e. GCD, Banerjee and Omega. The array referenced in the loop is two dimensional hence, simple, selective, true dependence and extended cycle shrinking for constant dependence shrinking is used for parallelism extraction.

Figure 6.13 - 6.17 represents the results for simple shrinking.

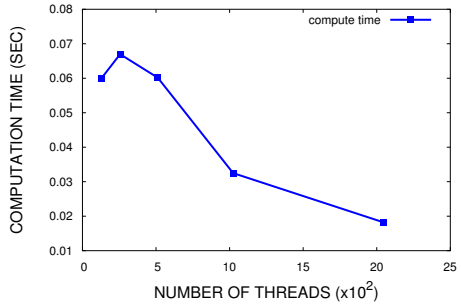


Figure 6.13: Plot for CSphd

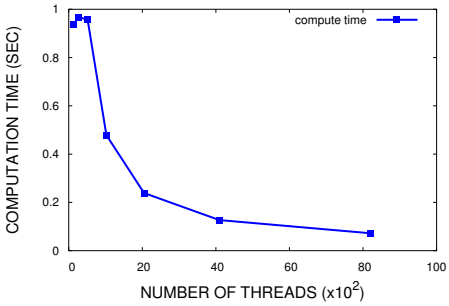


Figure 6.14: Plot for EVA

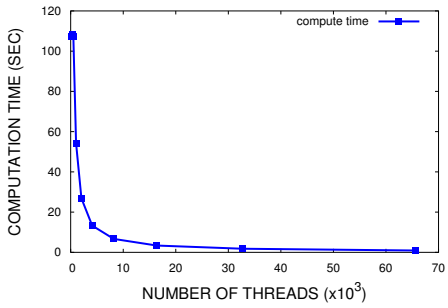


Figure 6.15: Plot for barth5

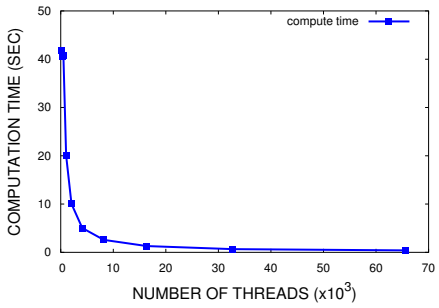


Figure 6.16: Plot for p2p-Gnutella04

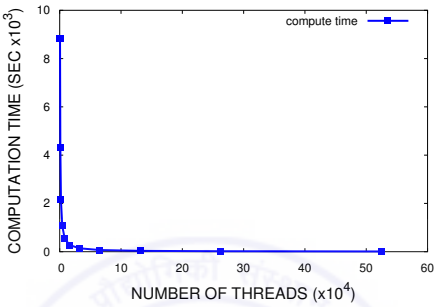


Figure 6.17: Plot for HEP-th-new

Figure 6.18 - 6.23 shows the performance of higbie using selective shrinking.

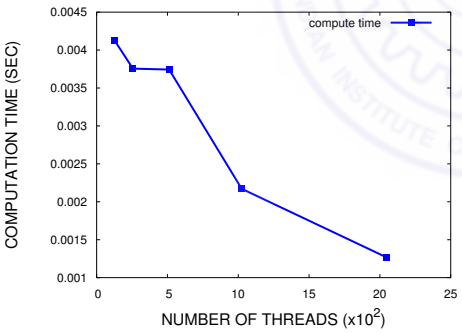


Figure 6.18: Plot for CSphd

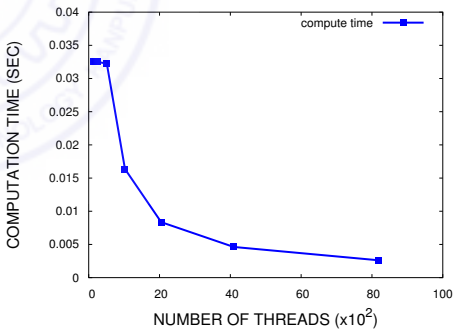


Figure 6.19: Plot for EVA

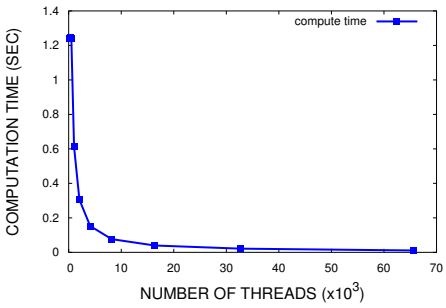


Figure 6.20: Plot for barth5

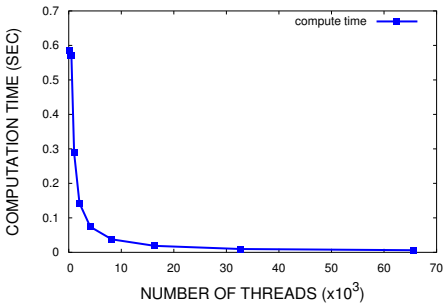


Figure 6.21: Plot for p2p-Gnutella04

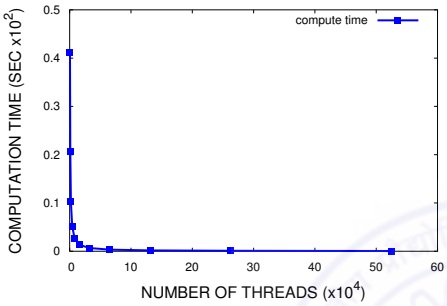


Figure 6.22: Plot for HEP-th-new

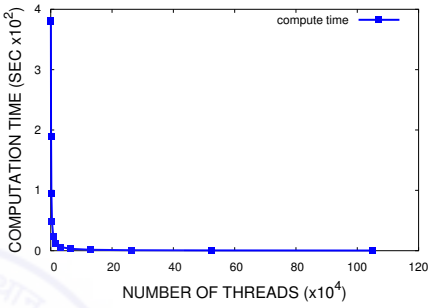


Figure 6.23: Plot for NotreDame_www

Figure 6.24 - 6.29 shows the performance using true dependence shrinking

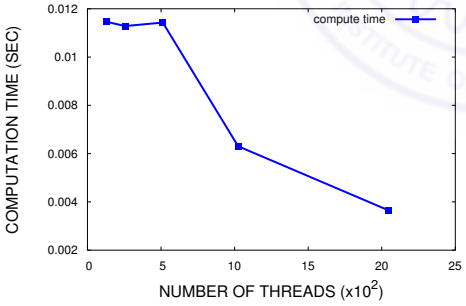


Figure 6.24: Plot for CSphd

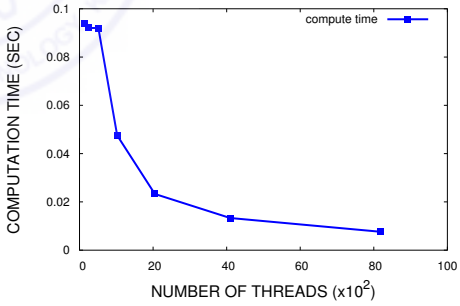


Figure 6.25: Plot for EVA

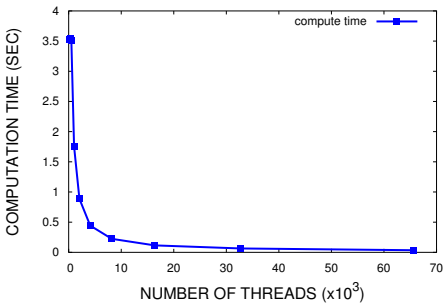


Figure 6.26: Plot for barth5

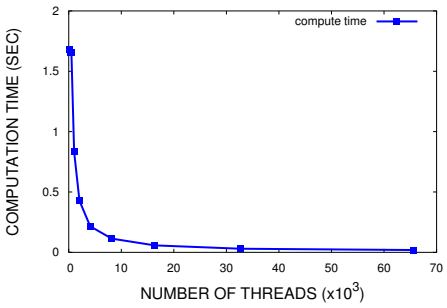


Figure 6.27: Plot for p2p-Gnutella04

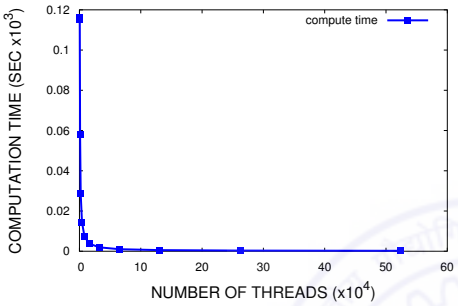


Figure 6.28: Plot for HEP-th-new

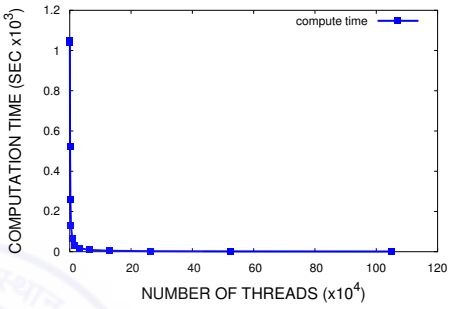


Figure 6.29: Plot for NotreDame_www

Figure 6.30 - 6.35 shows the results for extended cycle shrinking

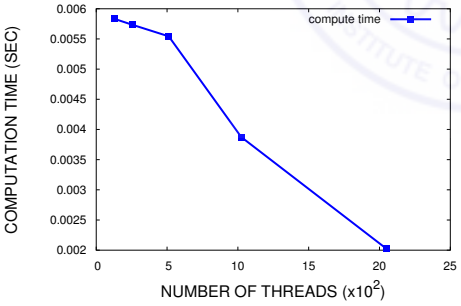


Figure 6.30: Plot for CSphd

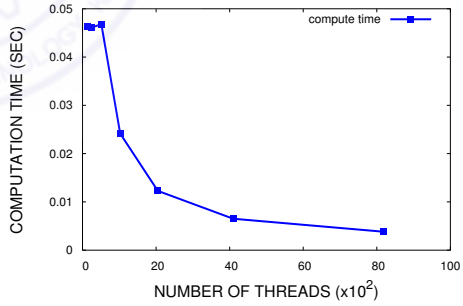


Figure 6.31: Plot for EVA

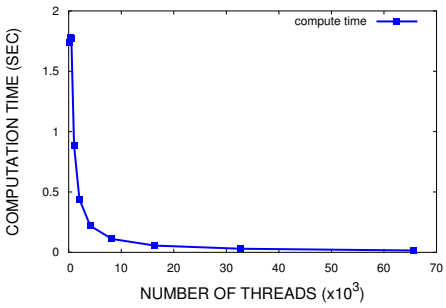


Figure 6.32: Plot for barth5

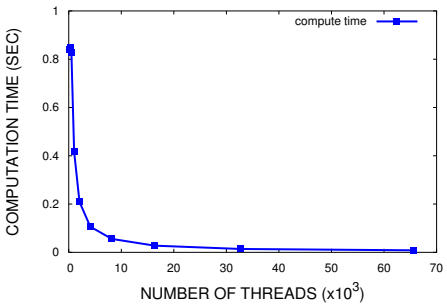


Figure 6.33: Plot for p2p-Gnutella04

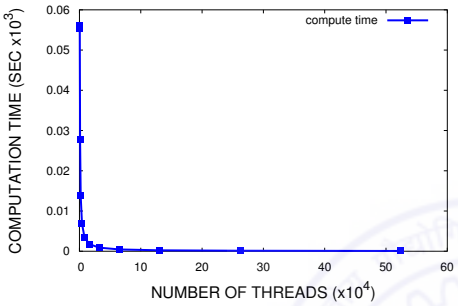


Figure 6.34: Plot for HEP-th-new

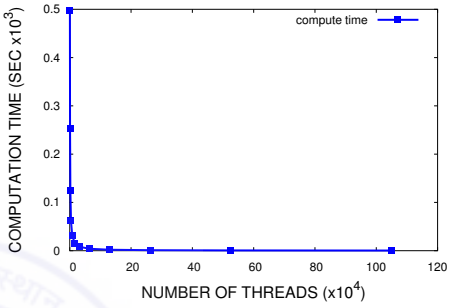


Figure 6.35: Plot for NotreDame_www

6.4.3 Results for Sandia Rules Benchmark

In Sandia Rules, the considered loop does not contain dependency if user choice for dependence test is `banerjee`. This implies that each iteration can execute in parallel. Figure 6.36 - 6.41 shows the performance of Sandia Rules benchmarks on standard dataset.

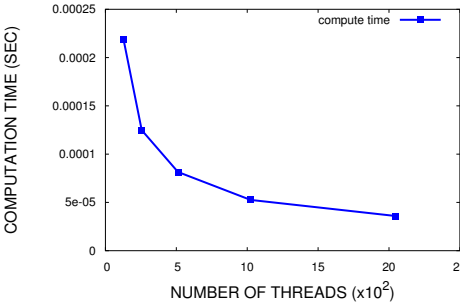


Figure 6.36: Plot for CSphd

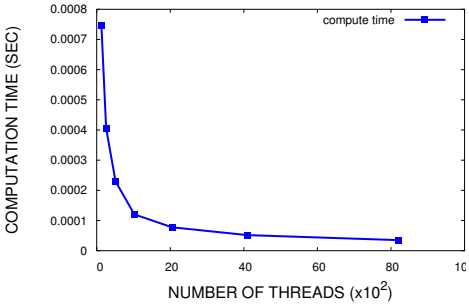


Figure 6.37: Plot for EVA

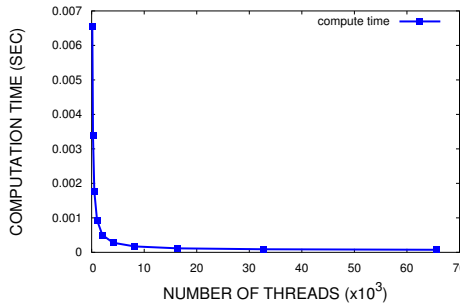


Figure 6.38: Plot for barth5

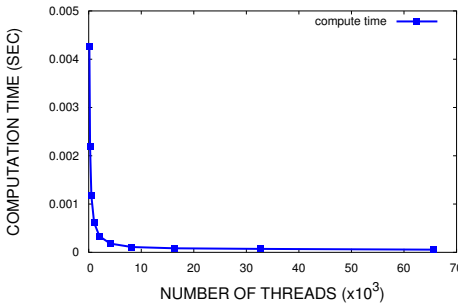


Figure 6.39: Plot for p2p-Gnutella04

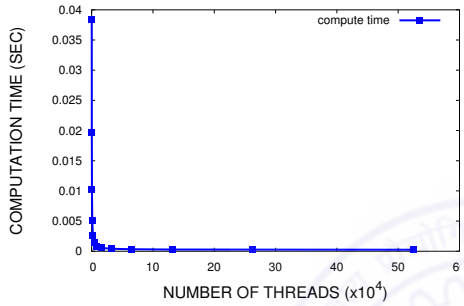


Figure 6.40: Plot for HEP-th-new

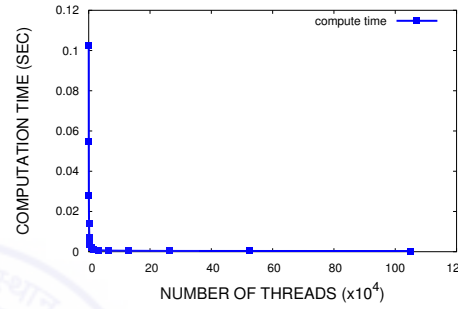


Figure 6.41: Plot for NotreDame_www

6.4.4 Results for Treepack Benchmark

In Treepack benchmark, any user choice for dependence test detects loop dependence. Sine the loop contains single dimensional array with subscript of $i \pm b$ form, only simple and extended cycle shrinking for constant dependence distance can be used for parallelism extraction. Following are the figures that represents the performance of treepack benchmark on various standard datasets, varying number of parallel threads.

Plots for Simple shrinking,

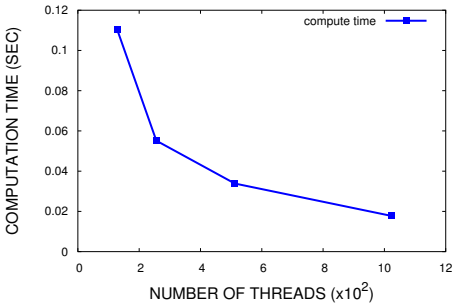


Figure 6.42: Plot for CSphd

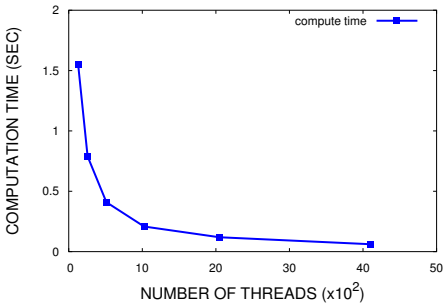


Figure 6.43: Plot for EVA

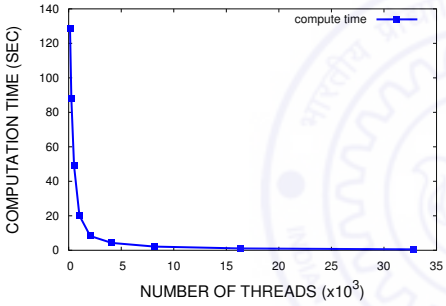


Figure 6.44: Plot for barth5

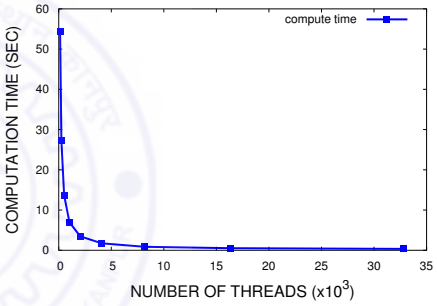


Figure 6.45: Plot for p2p-Gnutella04

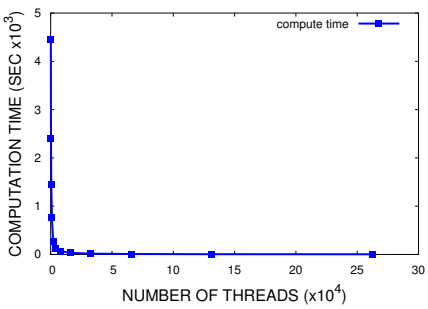


Figure 6.46: Plot for HEP-th-new

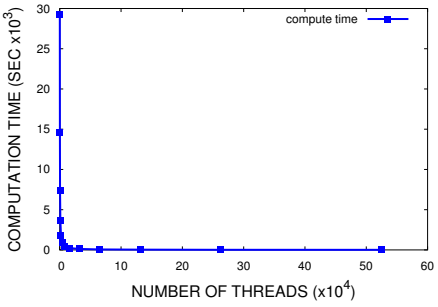


Figure 6.47: Plot for NotreDame_www

Plots for Extended cycle shrinking for constant dependence distance,

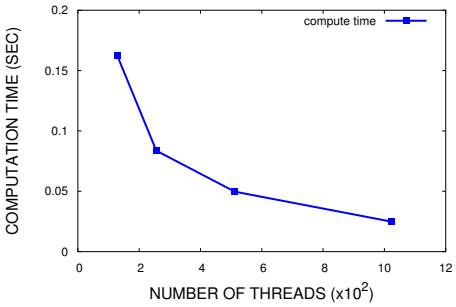


Figure 6.48: Plot for CSphd

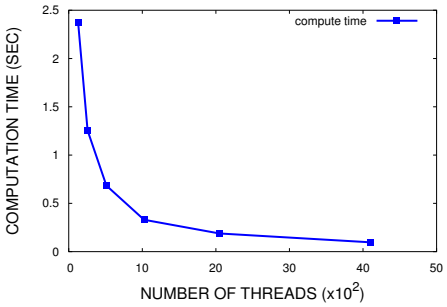


Figure 6.49: Plot for EVA

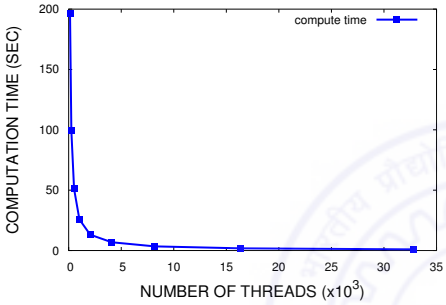


Figure 6.50: Plot for barth5

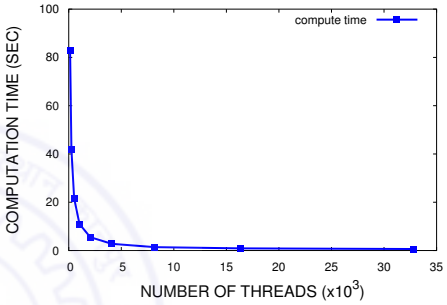


Figure 6.51: Plot for p2p-Gnutella04

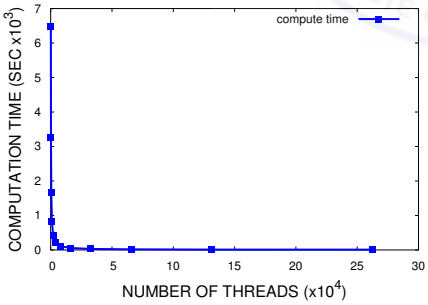


Figure 6.52: Plot for HEP-th-new

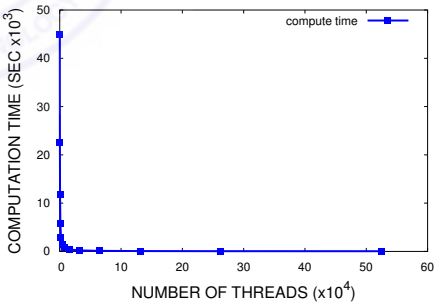


Figure 6.53: Plot for NotreDame_www

6.4.5 Results for SPEC Benchmark

For SPEC benchmark, tool is able to detects the dependence within the loop containing single dimensional array, hence simple and extended cycle shrinking are used for code transformation. Below are the graphs representing the performance

of benchmark with the variation in parallel threads.

Plots for Simple shrinking,

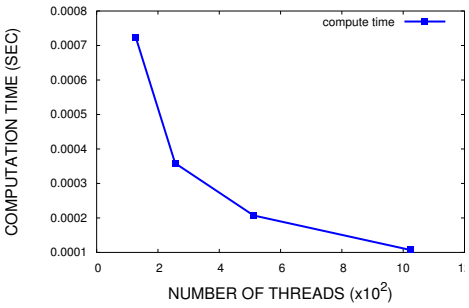


Figure 6.54: Plot for CSphd

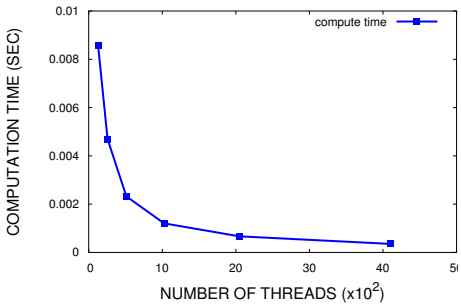


Figure 6.55: Plot for EVA

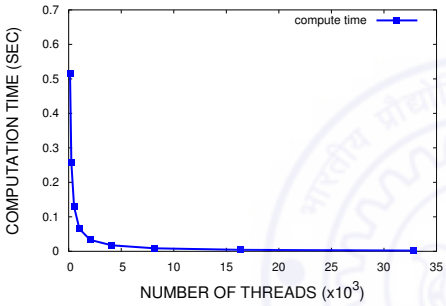


Figure 6.56: Plot for barth5

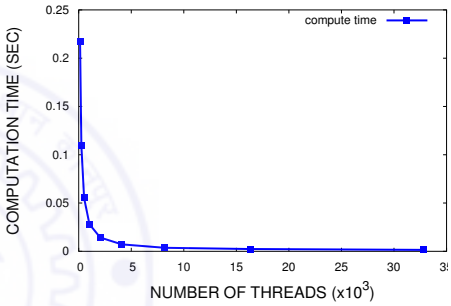


Figure 6.57: Plot for p2p-Gnutella04

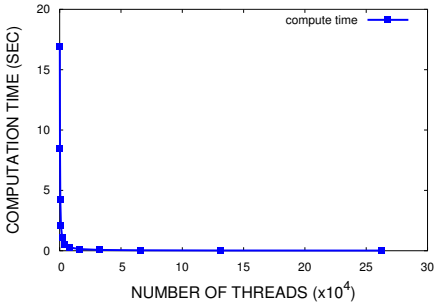


Figure 6.58: Plot for HEP-th-new

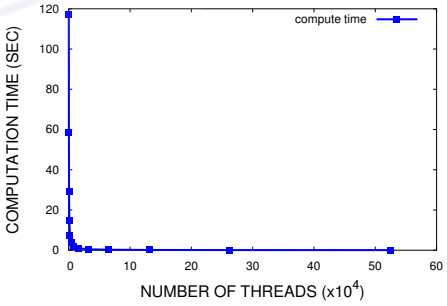


Figure 6.59: Plot for NotreDame_www

Plots for Extended cycle shrinking for constant dependence distance,

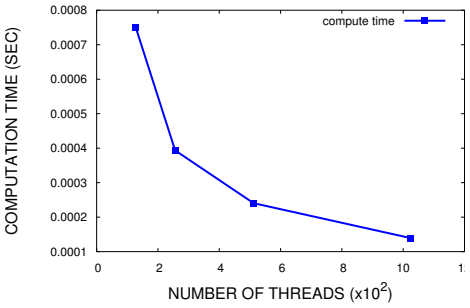


Figure 6.60: Plot for CSphd

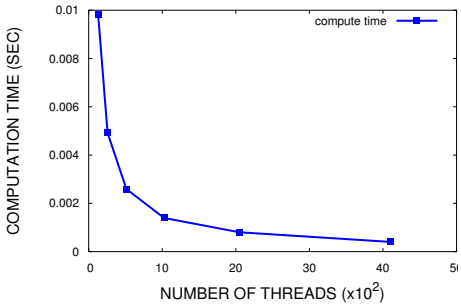


Figure 6.61: Plot for EVA

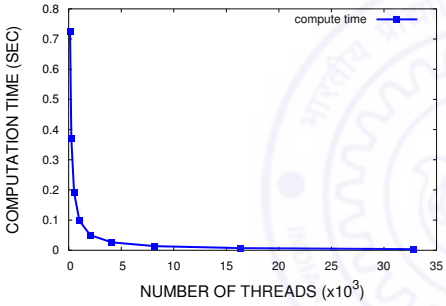


Figure 6.62: Plot for barth5

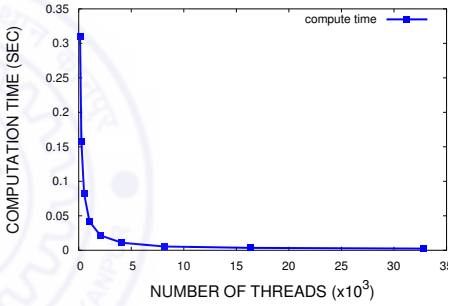


Figure 6.63: Plot for p2p-Gnutella04

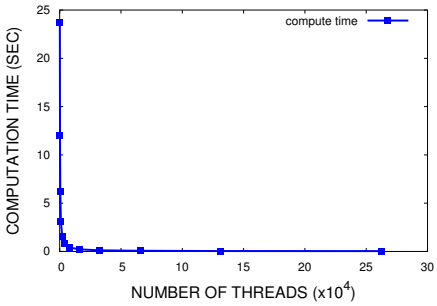


Figure 6.64: Plot for HEP-th-new

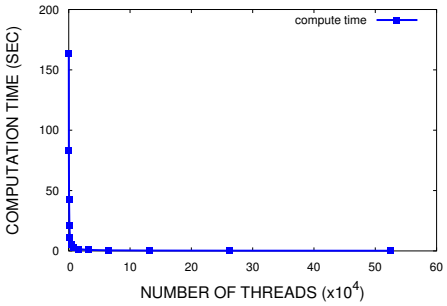


Figure 6.65: Plot for NotreDame_www

6.5 Results Analysis

From the above plots, we conclude that computation time of generated parallel program decreases with the increase in number of threads. Initially when the number of threads are very less, execution of program take huge computation time specially for the large standard datasets like HEP-th-new and NotreDame_www. But as the threads increases the drastic change in the computation time reduction is observed and later on it become constant. When the number of threads becomes very large then execution time of the program become constant, this happens due to the block overhead. As block overhead is negligible for the less number of threads and become significant with respect to computation time when it approaches to very large number.

Chapter 7

Conclusion and Future Work

CRINK is a source to source transformation system that converts an input sequential C program into a parallel program which can run on CUDA platform. The input can be an affine or a non-affine C program. Based upon the dependencies present in the loop within the program, the tool is able to extract parallelism using cycle shrinking [Pol88] and extended cycle shrinking techniques [SBS95]. CRINK uses simple, selective, true dependence, extended cycle shrinking for constant dependence distance and variable dependence distance for loop transformations. The tool is tested on 66 different configurations of standard benchmarks and datasets. Based on these benchmarks, the generated code performs better if dependence distance is large. We are able to conclude that as the number of threads increases, the computation time reduces exponentially, and later on it may become constant for large number of threads.

Various challenges occur while developing this tool, thereby imposing some limitations over CRINK . Some of them are listed below :

1. Not able to handle imperfectly nested loops.
2. It can only handle one loop at a time.
3. Loop should be either affine or non-affine. The tool cannot handle the fusion of affine and non-affine loops.

Future work can include improvements over CRINK , some of which are listed below:

1. Handling imperfectly nested loops.
2. Handling loops that contain both affine and non-affine dependencies.
3. Loop transformation of three or more nested loops using true dependence shrinking.

Bibliography

[AGMM00] Pedro V Artigas, Manish Gupta, Samuel P Midkiff, and José E Moreira. Automatic loop transformations and parallelization for java. In *Proceedings of the 14th international conference on Supercomputing*, pages 1–10. ACM, 2000.

[AK87] Randy Allen and Ken Kennedy. Automatic translation of fortran programs to vector form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(4):491–542, 1987.

[AK04] Randy Allen and Ken Kennedy. *Optimizing compilers for modern architectures*. Elsevier Science, 1st edition edition, 2004.

[Bar] Barabasi dataset. university of florida sparse matrix collection. `http://www.cise.ufl.edu/research/sparse/matrices/Barabasi/`. Accessed: 2014-04-01.

[BBH+08] Uday Bondhugula, Muthu Baskaran, Albert Hartono, Sriram Krishnamoorthy, J Ramanujam, Atanas Rountev, and P Sadayappan. Towards effective automatic parallelization for multicore systems. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–5. IEEE, 2008.

[BBK+08a] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. A compiler framework for optimization of affine loop nests for gpgpus. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 225–234. ACM, 2008.

[BBK+08b] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J Ramanujam, Atanas Rountev, and P Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Compiler Construction*, pages 132–146. Springer, 2008.

[BHRS08] Uday Bondhugula, Albert Hartono, J Ramanujam, and P Sadayappan. A practical and fully automatic polyhedral program optimization system. In *ACM SIGPLAN PLDI*, 2008.

[Bon13] Uday Bondhugula. Compiling affine loop nests for distributed-memory parallel architectures. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 33. ACM, 2013.

[BRS10] Muthu Manikandan Baskaran, Jj Ramanujam, and P Sadayappan. Automatic c-to-cuda code generation for affine programs. In *Compiler Construction*, pages 244–263. Springer, 2010.

[CA04] Bryan Chan and Tarek S Abdelrahman. Run-time support for the automatic parallelization of java programs. *The Journal of Supercomputing*, 28(1):91–117, 2004.

[cud] CUDA (compute unified device architecture). `http://http://en.wikipedia.org/wiki/CUDA`. Accessed: 2014-04-14.

[Cud12] C Cuda. Programming guide. *NVIDIA Corporation (July 2012)*, 2012.

[Dat] University of florida sparse matrix collection. `http://www.cise.ufl.edu/research/sparse/matrices/index.html`. Accessed: 2014-04-01.

[DCE73] George B Dantzig and B Curtis Eaves. Fourier-motzkin elimination and its dual. *Journal of Combinatorial Theory, Series A*, 14(3):288–297, 1973.

[GT14] Diana Göhringer and Jan Tepelmann. An interactive tool based on polly for detection and parallelization of loops. In *Proceedings of Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multi-core Embedded Computing Platforms*, page 1. ACM, 2014.

[GZA+11] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Grösslinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in llvm. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, 2011.

[HCC+10] Mary Hall, Jacqueline Chame, Chun Chen, Jaewook Shin, Gabe Rudy, and Malik Murtaza Khan. Loop transformation recipes for code generation and auto-tuning. In *Languages and Compilers for Parallel Computing*, pages 50–64. Springer, 2010.

[Hig] Lee Higbie. Collection of vecorizable loops. `http://www.netlib.org/benchmark/higbie`.

[jbu] C source codes. `http://people.sc.fsu.edu/~jburkardt/c_src/c_src.html`. Accessed: 2014-04-29.

[KBB+07] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J Ramanujam, Atanas Rountev, and P Sadayappan. Effective automatic parallelization of stencil computations. In *ACM Sigplan Notices*, volume 42, pages 235–244. ACM, 2007.

[KMP+96] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. The omega calculator and library, version 1.1. 0. *College Park, MD*, 20742:18, 1996.

[LE10] Seyong Lee and Rudolf Eigenmann. Openmpc: Extended openmp programming and tuning for gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[LME09] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. *ACM Sigplan Notices*, 44(4):101–110, 2009.

[LP94] Wei Li and Keshav Pingali. A singular loop transformation framework based on non-singular matrices. *International Journal of Parallel Programming*, 22(2):183–205, 1994.

[LP98] Yuan Lin and David Padua. On the automatic parallelization of sparse and irregular fortran programs. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 41–56. Springer, 1998.

[MGMG11] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, and Dan Ginsburg. *OpenCL programming guide*. Pearson Education, 2011.

[Nas] Nasa dataset. university of florida sparse matrix collection. `http://www.cise.ufl.edu/research/sparse/matrices/Nasa/`. Accessed: 2014-04-01.

[OK99] M.F.P. O'Boyle and P.M.W. Knijnenburg. Efficient parallelization using combined loop and data transformations. *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 0:283, 1999.

[Paj] Pajek dataset. university of florida sparse matrix collection. `http://www.cise.ufl.edu/research/sparse/matrices/Pajek/`. Accessed: 2014-04-01.

[PBB+10] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J Ramanujam, and P Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11. IEEE, 2010.

[PK99] Kleanthis Psarris and Konstantinos Kyriakopoulos. Data dependence testing in practice. In *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*, pages 264–273. IEEE, 1999.

[Pol88] Constantine D Polychronopoulos. Compiler optimizations for enhancing parallelism and their impact on architecture design. *Computers, IEEE Transactions on*, 37(8):991–1004, 1988.

[Rud10] Gabe Rudy. *CUDA-CHiLL: A programming language interface for GPGPU optimizations and code generation*. PhD thesis, The University of Utah, 2010.

[SBS95] Ajay Sethi, Supratim Biswas, and Amitabha Sanyal. Extensions to cycle shrinking. *International Journal of High Speed Computing*, 7(02):265–284, 1995.

[Sin13] Anjana Singh. Enabling non-affine kernels on heterogeneous platform. Master's thesis, Indian Institute of Technology, Kanpur, 2013.

[SK10] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[SNA] Snap dataset. university of florida sparse matrix collection. `http://www.cise.ufl.edu/research/sparse/matrices/SNAP/`. Accessed: 2014-04-01.

[VCJC+13] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):54, 2013.

[VRDB10] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. The paralax infrastructure: automatic parallelization with a helping hand. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 389–400. ACM, 2010.

[WSO95] Michael Joseph Wolfe, Carter Shanklin, and Leda Ortega. *High performance compilers for parallel computing*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[ZBG88] Hans P Zima, Heinz-J Bast, and Michael Gerndt. Superb: A tool for semi-automatic mimd/simd parallelization. *Parallel computing*, 6(1):1–18, 1988.