CRINK:Improved 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
Kamna
Roll Number : 13111024

under the guidance of
Dr. Amey Karkare

Department of Computer Science and Engineering
Indian Institute of Technology Kanpur
June, 2015

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

CERTIFICATE

It is certified that the work contained in this thesis entitled "*CRINK:Improved Automatic CUDA code generation for affine C programs.*", by *Kamna(Roll Number 13111024)*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Amey Karkare 16/06/2015

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

June, 2015

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

Abstract

A significant number of applications involve regular kernel.These kernels contain a large amount of iterations which often take tremendous amount of time. Hence, it has been necessary to create efficient parallel codes corresponding to these problems.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 we automate the process of creating parallel code for compute-intensive portions of the program.

We develop a tool CRINK , an end-to-end code transformation system, to convert sequential C programs to CUDA prorgam. To analyse the performance,tool has been tested on standard benchmarks and datasets,where we observed that computation time reduces by a significant amount as the number of threads increases.

Dedicated to

my family, my advisor and my friends.

Acknowledgement

I would like to express my sincere gratitude towards my thesis supervisor Dr.Amey Karkare for his constant support and invaluable suggestions in giving a proper direction to my efforts.He patiently listened to my problems and gave valuable insights.I feel motivated every time I attend his meeting.I express my sincere thanks and respect to him.

I would also like to thank all the faculty members of Computer Science and Engineering Department for providing the state-of-art facilities.

I would like to thank my parents, and all family members for their immense support.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 acknowledge Chandra Prakash for reading the draft of the thesis and suggesting improvements.

Last but most profoundly, I thank almighty God for reasons too numerous to mention.

Kamna

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

Graphics Processing unit (GPU) has traditionally been used for computer graphics and image processing as their highly parallel structure makes them more effective to be computed on a GPU with a large number of cores designed for handling multiple tasks simultaneously, rather than a general-purpose CPU which has very few cores optimized for sequential serial processing. GPU-accelerated computing is the use of a graphics processing unit (GPU) together with a CPU to accelerate computation intensive applications.

GPU-accelerated computing offers unprecedented application performance by offloading compute-intensive portions of the application to the GPU, while the remainder of the code still runs on the CPU. From a user's perspective, applications simply run significantly faster[Nvi].

CUDA is a parallel programming platform created by NVIDIA and implemented on GPU,which allows parallel computations with a large number of parallel threads.It is used to execute instruction sets on GPU. Using CUDA, GPUs are used for General purpose computing on Graphics Processing Unit (GPGPU)[Cud12]. GPGPU is the use of a GPU to perform computation in applications traditionally handled by the central processing unit (CPU).

In this thesis, we propose CRINK, an end-to-end code transformation tool using ROSE compiler, to automatically generate a CUDA program from a given sequantial C program. This tool aims to save the user from learning deep implementation

details of CUDA by providing the generated parallel counterpart of their sequential program.

We test and analyze the performance of CRINK over a number of standard benchmarks.

1.1 Problem Statement

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 the Nvidia **nvcc** compiler. The tool parallelizes the regular kernels in the program. We transform the compute-intensive portions of the program so as to be able to run them parallely on GPU , thereby converting the programs to CUDA C programs.

In this thesis,we have proposed tool named CRINK which is used to convert Sequential C program to CUDA program,which parallelizes the expensive parts and it is then tested over a number of standard benchmarks.

1.2 Motivation

A significant number of applications involve regular kernel. These kernels contain a large amount of iterations which often take tremendous amount of time. Hence, for a program with large number of iterations,if we can find some parallelization technique which can take iterations that do not depend on other iterations and execute them in parallel, then we can reduce the computation time by a significant amount. But if these kernels will have some intra loop dependencies(e.g. iteration (i,j) depending upon the iteration (i+k1,j+k2)) then blindly executing them in parallel on the GPU will result in an incorrect output. So we need some proper mechanism to parallelize such kind of loops. In this thesis, we have used some techniques like cycle shrinking to handle these dependencies with in a loop, which partition the dependent interations into group of indepedent iterations which can be executed in parallel on a GPU.

Example 1 illustrates intra loop dependencies.

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

Example 1 contains some intra-loop dependencies i.e. some iterations depend upon the value of previous iterations. It is natural for a programmer to write a sequential program for any application. But our automatic parallelization tool will help the programmer to run the program in parallel using multiple threads on a *GPU accelerated* system thereby reducing the computation time by a significant amount. So, we propose a technique using which we can partition the dependent iterations into a group of independent iterations which can be executed in parallel on a *GPU accelerated* system.

CRINK serves as a ROSE plugin which auotmatically generates a parallel CUDA program given an input sequential C program.

1.3 Related Work

A lot of research has been done in past few years in the field of automatic parallelization of regular kernels. 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] But manual developement of the parallel code using these programming model is still cumbersome.

CUDA-CHiLL[Rud10] system are based on the idea of a transformation script that performs source-to-source transformations. It uses a polyhedral model to allow for powerful composition of transformations.Lee et al.[LME09, LE10] presented a compiler framework for translating standard OpenMP shared-memory programs into CUDA-based GPGPU programs,while Baskaran et al. [BRS10] proposed a fully

automatic source to source code transformation system from c to CUDA for affine programs.

1.4 Outline of our Solution

CRINK takes sequential C programs as input and generates CUDA program as output.The tool basically consists of following phases:

- **Compilation Phase:** For this purpose we use ROSE compiler(source to source translator).
- **Loop Normalization Phase:** Loop is said to be normalized if the lower bound is zero and the iterator increases by one. If loop does not satisfy this condition, then we need to normalize the loop, as some of the dependence tests are applicable only if a loop is normalized.
- **Dependence testing Phase:** This phase checks whether dependencies exist within loops using either gcd[PK99] or banerjee[AK04] test.
- **Parallelism extraction Phase:** This phase uses Cycle Shrinking [Pol88] or Extended cycle shrinking[SBS95] to partition the dependent iterations of the loop into group of independent iterations.
- **Code Generation Phase:** This phase generates the CUDA C code based on the information collected from previous phases.

The output code is compiled using the Nvidia **nvcc** compiler. We have tested the tool over various standard benchmarks using standard datasets and it has been observed that the computation time of the parallel program reduces as the number of threads increase.

1.5 Thesis Organization

Rest of the thesis is organized as follows:

Chapter 2, Background, contains all the necessary background information to understand the tool like affine programs,loop normalization, dependence test, cycle shrinking, CUDA, ROSE compiler.

Chapter 3,Implementation Details,discuss the various stages like compilation phase, Dependence test, Parallelism Extraction, Code Generation in detail with the help of an example

Chapter 4,Experiments and results,discuss the performance of CRINK by varying the number of threads.

Chapter 5,Conclusion and Future Work,concludes the work of the thesis and scope of future work.

Chapter 2

Background

This chapter will focus on some of the pre-requisites needed to understand CRINK implementation details. This chapter is concluded with a discussion on ROSE compiler which has been used to design CRINK . Detailed background can be found in [AK04, WSO95, Pol88, SBS95, SK10, KMP+96].

2.1 Affine Programs

Programs containing only affine loops are called affine programs. Whereas affine loops are the loops in which array index and loop bounds are the linear functions of loop index variables, hence their memory access pattern is always known at compile time. It will become more clear with the help of following example.

Example 2.
```
for(j=4;j<50;j++)
    b[j+8]=b[j+3];
```

In this example, access pattern of array, **b** is a linear function of the loop index variable, **j** and is known at compile time.

2.2 Loop Normalization

A loop is said to be normalized if the lower bound is zero and its iterator increments by one.

Example 3.
```
for(j=4;j<50;j++)
    b[j+8]=b[j+3];
```

In this example, the loop is not normalized. Loop normalization algorithm is given below:

Algorithm 1 Loop normalization Algorithm(L_0)[AK04]

Input: A loop L_0 with L , U and S as the lower bound, upper bound and step size respectively.

1: Let i be the unique compiler-generated loop induction variable.
2: S_1: Replace the loop header for L_0
3: DO I = L,U,S
4: with the adjusted loop header
5: DO i= 1,(U-L+S)/S;
6: S_2: Replace each reference to I within loop by
7: i*S-S+L;
8: S_3: Insert a finalization assignment
9: I=i*S-S+L;
10: immidiately after the end of the loop.

2.3 Data Dependence

There exists dependency between two iterations i and j ,if $i \leq j$ and

- i^{th} iteration is reading a memory location which was written in j^{th} iteration, known as RAW (Read after write).
- i^{th} iteration is writing a memory location which was read in j^{th} iteration, known as WAR(Write after read).
- i^{th} iteration is writing a memory location which was written in j^{th} iteration, know as WAW(Write after write)

So,we need data dependence test to find which part of the code is independent and which part of the code are inter-dependent.

2.4 Data Dependence Test

A lot of dependence tests are proposed in the data dependence literature, out of which CRINK tool uses GCD test[PK99] and Banerjee's test[AK04].All these tests compare in terms of accuracy and efficiency. These dependence tests always approximate results on conservative side i.e. a dependence can exist even 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[PK99] test is arguably the most basic dependence test. GCD test is based on a theorem of elementary number theory which states that there exists an integer solution for a linear equation if the greatest common divisor(GCD) of the coefficients 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. However if condition does apply then a dependence does not necessarily exist.In this case GCD test gives a *may be* answer.
Consider the equation:

$$a_1x_1 + ... + a_nx_n = a_0 \tag{2.1}$$

Equation 2.1 has an integer solution iff GCD(a_1,....a_n) divides a_0.

Example 4.
```
for(int i=0;i<100;i++){
    a[2*i]=b[i];
    c[i]=a[4*i+1];
}
```

In example 4,The GCD of (2,4) is 2 and dividend is 1. As 2 can not divide 1,so there is no dependency between the two statements in the loop.

2.4.2 Banerjee Test

GCD test[AK04] does not provide any mechanism for bound checking and most common gcd encountered in practice is 1 which divides everything , so to overcome this, Banerjee test was proposed. 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 2.1 can achieve by using some mechanism which is discussed later. If the constant term (a_0) of equation 2.1 does not fall between maximum and minimum vaules(calculated above), then no dependence exists, otherwise a real solution to the linear equation exists and it will return that dependency may exist.

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

$$a^+ = if\ a >= 0\ then\ a\ else\ 0 \tag{2.2}$$

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

$$a^- = if\ a >= 0\ then\ 0\ else\ -a \tag{2.3}$$

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

$$\sum_{k=1}^{d}(A_k I_k\ -\ B_k I_k) = B_0\ -\ A_0 \tag{2.4}$$

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

$$LB_k^{\psi_k} \leq (A_k I_k\ -\ B_k I_k \leq UB_k^{\psi_k} \tag{2.5}$$

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

$$\sum_{k=1}^{d} LB_k^{\psi_k}\ \leq\ \sum_{k=1}^{d}(A_k I_k\ -\ B_k I_k)\ \leq\ \sum_{k=1}^{d} UB_k^{\psi_k} \tag{2.6}$$

$$or,\ \sum_{k=1}^{d} LB_k^{\psi_k} \ \leq\ B_0 - A_0 \ \leq\ \sum_{k=1}^{d} UB_k^{\psi_k} \tag{2.7}$$

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 can not exist.

Lower and upper bounds can be calculated by:

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

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

2.5 Cycle Shrinking

Cycle shrinking [Pol88] is a compiler transformation which is used to parallelize serial loops. This transformation uses the data dependence graph to check whether the exisiting dependencies in the loop allow the loop to execute in parallel without violating any semantics. If no dependence exists, our transformation is simple. Otherwise loops with dependence graphs that do not form strongly connected components can become fully or partially parallel. When the dependence graph forms a cycle, node splitting can be used to break the cycle, assuming that atleast one of the dependency involved is antidependency.

Cycle Shrinking is used to extract parallelism that may be present in perfectly nested loops. With the help of cycle shrinking serial loop is transformed into two perfectly nested loops: an outermost serial and innermost parallel loop.

2.5.1 Characterization of Reduction Factor

Cycle shrinking parallelizes the loop by partitioning the serial loop into two perfectly nested loops:an outermost serial and innermost parallel loop.The two array references that are involved in dependence and includes the source and sink of dependence are called *Reference Pair*. 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.

Example 2 has only one reference pair i.e. `b[j+8]-b[j+3]`. Consider a n-nested loop with indices $J_1, J_2, .., J_n$ and following statement present inside the loop:

S_1: `X[`$J_1 + a_{11}, J_2 + a_{12}, .., J_n + a_{1n}$`]=Y[`$J_1 + a_{21}, J_2 + a_{a22}, .., J_n + a_{2n}$`]`

S_2: `Y[`$J_1 + b_{11}, J_2 + b_{12}, .., J_n + b_{1n}$`]=X[`$J_1 + b_{21}, J_2 + b_{a22}, .., J_n + b_{2n}$`]`

In above Statements, there are two reference pairs;

Z_1: S_1`:X[`$J_1 + a_{11}, J_2 + a_{12}, .., J_n + a_{1n}$`]` - S_2`:X[`$J_1 + b_{21}, J_2 + b_{a22}, .., J_n + b_{2n}$`]`

Z_1: S_2`:Y[`$J_1 + b_{11}, J_2 + b_{12}, .., J_n + b_{1n}$`]` - S_1`:Y[`$J_1 + a_{21}, J_2 + a_{a22}, .., J_n + a_{2n}$`]`

Linear equation for reference pair Z_1 will be:

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

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

$$..$$

$$..$$

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

and linear equation for reference pair Z_2 will be :

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

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

$$..$$

$$..$$

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

The distance vectors for reference pairs Z_1 and Z_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. Final step to calculate reduction factor is $< \lambda_1, \lambda_2, .., \lambda_n >=<$

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

2.5.2 Types of cycle shrinking

Cycle Shrinking speeds up the loop by factor λ called reduction factor as described previously.

I. Simple Shrinking: Dependence cycle is considered separately for each loop in the nest and then algorithm for simple shrinking is applied to each loop separately.

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

Algorithm 2 Simple Shrinking

Input: Consider a n nested loop with loop indices $j_1, j_2, .., j_n$ with $l_1, l_2, .., l_n$ as the lower bound , $u_1, u_2, .., u_n$ as upper bound and $\lambda_1, \lambda_2, .., \lambda_n$ as reduction factor.

DO $j_1 = 0, u_1, \lambda_1$
..
..
  DO $j_n = 0, u_n, \lambda_n$
    Loop Body
  ENDO

Output: Loop after applying simple cycle shrinking is:

DO $j_1 = 0, u_1, \lambda_1$
..
..
  DO $j_n = 0, u_n, \lambda_n$
    DOALL $i_1 = j_1, j_1 + \lambda_1 - 1$
..
..
      DOALL $i_n = j_n, j_n + \lambda_n - 1$
        Loop Body
..
..
  ENDOALL
ENDO

Consider the example :

Example 5.

```
Do I=3,N
   S1:  A[I]=B[I-2]-1;
   S2:  B[I]=A[I-3]*k;
ENDO
```

In the above example, statements S_1 and S_2 are involved in dependence, Reduction factor for this dependence is 2 and 3. But by definition of reduction factor $\lambda_1 = min(\phi_1^1, \phi_2^1) = min(2, 3) = 2$.

Cycle shrinking will shrink the loop by a factor of 2. Therefore, transformed loop will be:

```
DO J=3,N,2
     DOALL I=J,J+1
          S1:  A[I]=B[I-2]-1;
          S2:  B[I]=A[I-3]*k;
       ENDOALL
ENDO
```

II.Extended Cycle Shrinking [SBS95]Simple shrinking deals with the dependence having constant distance and speeds up the loop by reduction factor. For variable dependence distance, we need some improved version of simple shrinking which can handle variable dependence distance and can reduce the number of partitions as well. For this purpose Extended cycle shriking was proposed. Extended Cycle shrinking is basically used for constant dependence distance as well as for variable dependence distance,giving better results as comapred to simple cycle shrinking in case of constant dependence distance.

2.5.3 Reduction Factor for constant dependence distance

The process of calculating reduction factor for constant dependence distance is same as that of Simple cycle shrinking.

2.5.4 Reduction Factor for variable dependence distance

Consider a n-nested loop with loop indices $j_1, j_2, .., j_n$ and reference pair $S_j - S_i$ that contains variable dependence distance:

$S_i : A[a_{10} + a_{11}j_1 + .. + a_{1n}j_n, .., a_{n0} + a_{n1}I_1 + .. + a_{nn}j_n]$

$S_j : A[b_{10} + b_{11}j_1 + .. + b_{1n}j_n, .., b_{n0} + b_{n1}I_1 + .. + b_{nn}j_n]$

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:

$$\lambda_k = \frac{(a_{k0} - b_{k0}) + \sum_{i=1, i \neq k}^{n} (a_{ki} - b_{ki}) * I_i + (a_{kk} - b_{kk}) * I_k}{b_{kk}} \tag{2.8}$$

2.5.5 Extended Cycle Shrinking for Constant Dependence Distance

For an m-nested loop with upper bounds $N_1, N_2, .., N_m$ and reduction factors $\lambda_1, \lambda_2, .., \lambda_m$, extended cycle shrinking algorithm is given below and is explained with the help of example 6

Algorithm 3 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 6.

```
Do I=1,10
    Do J=1,8
        S1: A[I+3,J+4]=B[I,J];
        S2: B[I+2,J+3]=A[I,J];
    ENDO
ENDO
```

Transformed loop after applying shrinking will be:

```
DO K=1,min(⌈10/2⌉, ⌈8/3⌉) + 1
DOALL I=0,1
  START[I]=(K-1)*λ_I
ENDOALL
DOALL I=START[0],min(START[0]+1,10)
  DOALL J=START[1],8
    S1:  A[I+3,J+4]=B[I,J];
    S2:  B[I+2,J+3]=A[I,J];
  ENDOALL
ENDOALL
DOALL I=START[0]+2,N_1
  DOALL J=START[1],min(START[1]+2,8)
    S1:  A[I+3,J+4]=B[I,J];
    S2:  B[I+2,J+3]=A[I,J];
  ENDOALL
ENDOALL
ENDO
```

2.5.6 Extended Cycle Shrinking for Variable Dependence Distance

Algorithm for extended cycle shrinking[Sin14] for variable dependence distance is given below and is explained with the help of example 7

Algorithm 4 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**

Example 7.

```
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];}
```

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

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 4:

$id_1 = 1, id_2 = 1$

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

}

2.6 Compute Unified Device Architecture(CUDA)

NVIDIA introduced CUDA[Cud12] a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in more efficient way than on a CPU.

CUDA comes with a software environment that allows developers to use C as a high-level programming language. Goal of CRINK is to take sequential C program as input and generate parallel CUDA program as output.

Following feature of CUDA are used in this thesis work.

Features:

- CUDA program which generate a large number of threads to execute instructions set in parallel, consists of two levels of parallelism:
  - **Block level**
  - **Thread level**

  CUDA with standard language C consists of:
  - **host code** which is executed by a single thread on the host CPU.
  - **kernel code** which is executed by multiple parallel threads on the GPU.

  Parallel portion of the program are executed on device known as kernels.Each Kernel is executed by many threads at a time and a grid is the entire array thread generated by a kernel launch. A grid is divided into blocks and further each block is divided into threads as shown in figure 2.1. A grid can have at most 65535 number of blocks. Each block may have at most 512 threads. Hence, maximum 512*65535 threads can be generated by parallelized program at a time.

- 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 and default value for this data type is 1.

- Instructions which are executed by GPU, can access only those memory locations which are located on GPU device. Hence, to access a data, it needs to be transfered on device before executing the instructions.

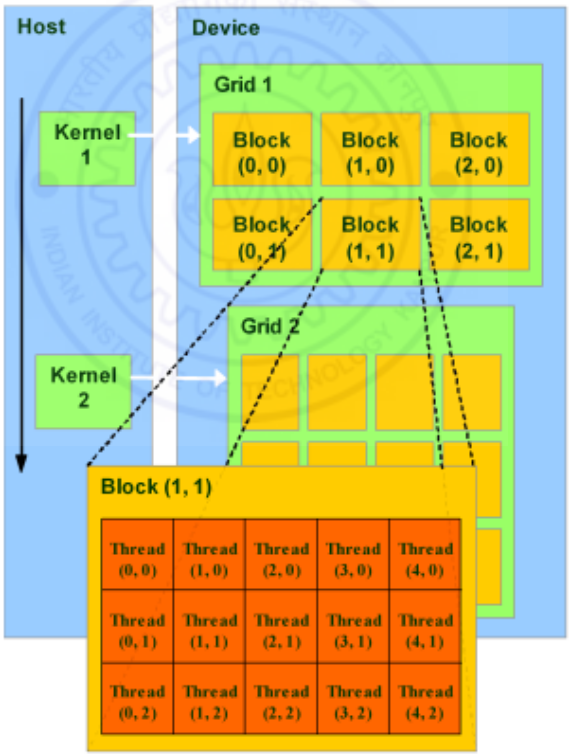


Figure 2.1: CUDA Programming Model[cud]

2.6.1 CUDA C programming Syntax

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

- **cudaMalloc()**
  - **Syntax:**`cudaMalloc ((void **)Ptr, size_t size)`
  - **Usage:**It is used to allocate memory on device global memory. Two parameters are required by this function, address of a pointer to the allocated object and size of the object.
- **cudaMemCpy()**
  - **Syntax:** `cudaMemcpy (void *dest, const void *source, size_t size, cudaMemcpyHostToDevice)`
  - **Usage:** It 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
- **cudaThreadSynchronize()**

  Threads can be synchronized using this function .
- To create a kernel, we declare it as a global function, as following,

  __global__ void function name(arguments)function specifications
- CUDA kernel is called from host code as:

  `functionName<<<noOfBlock,noOfThread>>>(Parameters)` specifying the number of threads and blocks.
- The thread id and block id for all three dimensions of the thread, on the device, can be computed as following:

```
thread_x = threadIdx.x;        block_x = blockIdx.x;
thread_y = threadIdx.y;        block_y = blockIdx.y;
```

```
thread_z = threadIdx.z;        block_z = blockIdx.z;
```

2.7 ROSE Compiler Infrastructure

ROSE[DQWc] is an open source compiler infrastructure for building tools that can read and write source code in multiple languages (C/C++/Fortran). ROSE is used for building source-to-source translators. ROSE makes it easy to build tools that read and operate on source code from large scale applications.

ROSE works by reading the source code and/or binary and generating an Abstract Syntax Tree (AST). The AST forms a graph representing the structure of the source code and is held in the memory. The nodes used to define the AST graph are an intermediate representation(IR). ROSE also provides mechanisms to traverse and manipulate the AST. Finally ROSE provides mechanisms to regenerate source code from the AST.

Chapter 3

Implementation Details

The tool takes a sequential C program as input and outputs a corresponding CUDA program involving various phases. This chapter describes all these phases *Compilation Phase, Loop Normalization, Dependence Testing Phase, Parallelism Extraction Phase and Code Generation Phase* in detail.

3.1 Compilation Phase

In this phase input code is passed to ROSE Compiler.[ros] ROSE reads the input program and generates an Abstract Syntax Tree (AST) of the input code. The nodes used to define the AST are an intermediate representation (IR) of the input program. We traverse and manipulate the AST in various phases according to our need.

3.1.1 Loop Normalization

Loop is said to be normalized if the lower bound is zero and its iterator increments by 1. There are some dependency detection tests that consider the loop to be normalized. Therefore performing dependence test on non-normalized loop will give incorrect results, so loop normalization is required before going for further phases. Below are the two examples of non-normalized loop:

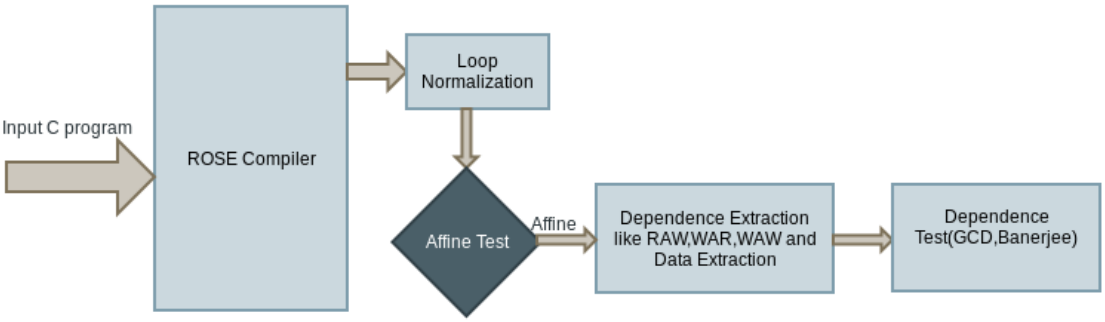


Figure 3.1: Compilation Phase

- **Case 1**

  Example 8.
  ```
  for(i=10;i>=0;i--)
      a[i]=b[i];
  ```

  Above example represents reverse loop.After normalization, the above loop will look like the following:

  ```
  for(i=0;i<=10;i++)
  a[10-i]=a[10-i];
  ```

- **Case 2**

  Example 9.
  ```
  for(i=7;i<50;i+=3)
      a[i]=b[i]+7;
  ```

  Here, the lower bound of the loop is non-zero and its iterator increments by 3. Therefore, this loop is not normalized. The normalized form of above example will look like:

  ```
  for(i=0;i<(50-7)/3;i++)
      a[i*3+7]=b[i*3+7]+7;
  ```

  After this step, everything will be processed on the normalized loop.

- Function used for Loop normalization in CRINK :
  - **Syntax:** `void loop_normalization(SgNode* forloop,set<string> var)`
  - In the above function SgNode represents the node in intermediate representation(IR) of AST. Here we are looking for SgNode of type SgForStatement(represents the FOR statements in AST) and var is used to collect all the array references in the loop.Using this function,we are normalizing the FOR loop.

3.1.2 Affine Test

This step checks whether the given input program contains an affine or a non-affine loop. Further phases will be carried out only for the affine loops.

Function used for Affine test in CRINK :

- **Syntax:** `bool isPotentiallyAffine(SgExpression* expr)`
- In the above function SgExpression represents the notion of an expression. Expressions passed during function call are array indices and loop bounds and they are checked if they are affine or not.

3.1.3 Data Extraction

ROSE generatesQ an Abstract Syntax Tree (AST),which ensures that input program is error free and following data are extracted from the nodes in AST.

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

After this, data structure is created to store all the values extracted in this phase.

Function used for Data Extraction in CRINK :

- **Syntax:** `void ref_function(SgNode* for_loop,SgFunctionDefinition *defn )`

- In the above function SgNode represents the node in the AST of the intermediate representation(IR) of input c program. Here we are looking for SgNode of type SgForStatement(represents the FOR statements in AST). SgFunctionDefinition represents the scope of FOR statement.

3.1.4 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 in the statements inside FOR loop.Using this information we identify the dependence(like WAW, RAW, WAR) in the statements. 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)

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

Functions used for Dependence Extraction(RAW,WAW,WAR) in CRINK :

- **Syntax:** `void calculate_intersection_RAW(loop* node)`

  - In the above function loop represents data structure that contains all the necessary information required(read and write array references within statements of for loop) for calculating RAW.

- **Syntax:** `void calculate_intersection_WAW(loop* node)`

  - In the above function loop represents data structure that contains all the necessary information required for calculating WAW.

- **Syntax:** `void calculate_intersection_WAR(loop* node)`

  - In the above function loop represents data structure that contains all the necessary information required for calculating WAR.

3.2 Dependence Test

Consider a loop with large number of iterations which, if executed sequentially, would become bottleneck to the computation time of the program. To reduce the computation time, we parallelize the loop so that many iterations can execute in parallel and hence the loop takes less computation time. But this can only happen if the statements within the loop does not form dependence with each other. 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, we need some kind of dependence test to analyze it. If dependence exists, then we need to find out which iterations can execute in parallel; the process taking place in parallelism extraction phase. But if dependence does not exist, then CUDA code will be generated because each iteration can execute in parallel on GPU. Dependence test 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 it does not.

The tool provides two different loop dependence test i.e. *GCD,Banerjee.*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 `-test:gcd`(for GCD test) and `-test:banerjee`(for Banerjee test).

Function used for Dependence Test in CRINK :

- **Syntax:** `void Dependence_Testing_Interface(char *test_name, loop* write,SgNode* loop,int loop_number)`

- Here test_name indicates the name of the test gcd or banerjee. write is a data structure of type loop which stores read and write references with in a particular loop.

3.3 Parallelism Extraction

If loop carried dependencies exist,then next step is to extract the parallelism. Since dependency exists, loop iterations can not be executed in parallel. The aim of this phase is to extract parallelism out of the sequential loop by partitioning the iteration space into groups of iterations that are independent of each other and hence can execute in parallel.

The tool uses *Cycle shrinking* [Pol88] and *Extended cycle shrinking* [SBS95] for paralleism extraction. The basic cycle shrinking offers *Simple* 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`
Extended cycle shrinking for constant dependence distance	`extShrinking1`
Extended cycle shrinking for variable dependence distance	`extShrinking2`

Function used for Parallelism Extraction in CRINK :

- **Syntax:** `void CycleShrinking( char *shrinking,loop* write,SgNode* n,int loop_number,set<string> var)`

- Here shrinking indicates the type of shrinking; write is a data structure of type loop which stores read and write references within a particular FOR loop n.

3.3.1 Compute Distance Vectors

Distance vector is the distance between two consecutive array references involved in dependency.These array references can be of the following type:

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

Using read and write reference variables in the loop, all possible reference pairs are found. Thereafter the 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 : `x[i][j]=y[i-1][j-1]+x[i-2][j-6];`

S_2 : `a[i][j]=x[i-3][j-2];`

Here, for array x there are two reference pairs: R_1 : $\{S_1 : x[i][j] - S_1 : x[i-2][j-6]\}$ and R_2 : $\{S_1 : x[i][j] - S_2 : x[i-3][j-2]\}$. So the distance vector for R_1 and R_2 are $< \phi_1^1, \phi_1^2 >=< 2, 6 >$ and $< \phi_2^1, \phi_2^2 >=< 3, 2 >$ respectively. Therefore the actual distance vector for array $a[\ ][\ ]$ is $< 2, 2 >$.

3.3.2 Calculating Reduction Factor

If for cycle shrinking user gives `simple` or `extShrinking1` as input then we need to calculate Reduction factor after calculating distance vector. Reduction factor is the minimum distance of each coordinate among all the corresponding coordinates of every distance vectors.

3.3.3 Compute Data Dependence Vector

If for cycle shrinking user gives `extShrinking2` as input then we need to calculate Data Dependence Vector for each reference pair using equation 2.8.

3.4 Code Generation

Final phase of CRINK generates CUDA program as output. If dependence exists , then we perform cycle shrinking on input code and output of cycle shrinking viz. reduction factor and data dependence vector, acts as an input for code generation phase. Otherwise we do not perform cycle shrinking and CRINK direclty jumps to this phase after dependence testing. With the help of reduction factor and data dependence vector,loop iterations are partitioned into set of independent iterations that can execute in parallel. Figure 3.2 shows the various routines involved in code generation.

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

Code generation for kernel function declaration: This is the initial step of code generation phase, it generate the code corresponding to the declaration of kernel function. The kernel function declaration will give the information about kernel name, number of arguments and their respective type.

Code generation of C code (above FOR loop): The next step is to generate the code for the sequential portion of the input program which is above the FOR loop .

Kernel variable code generation and memory allocation: Kernel execution needs device variables allocated on device, so this step generates the target code for declaration of these variables on host and allocates the memory for these variables in 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

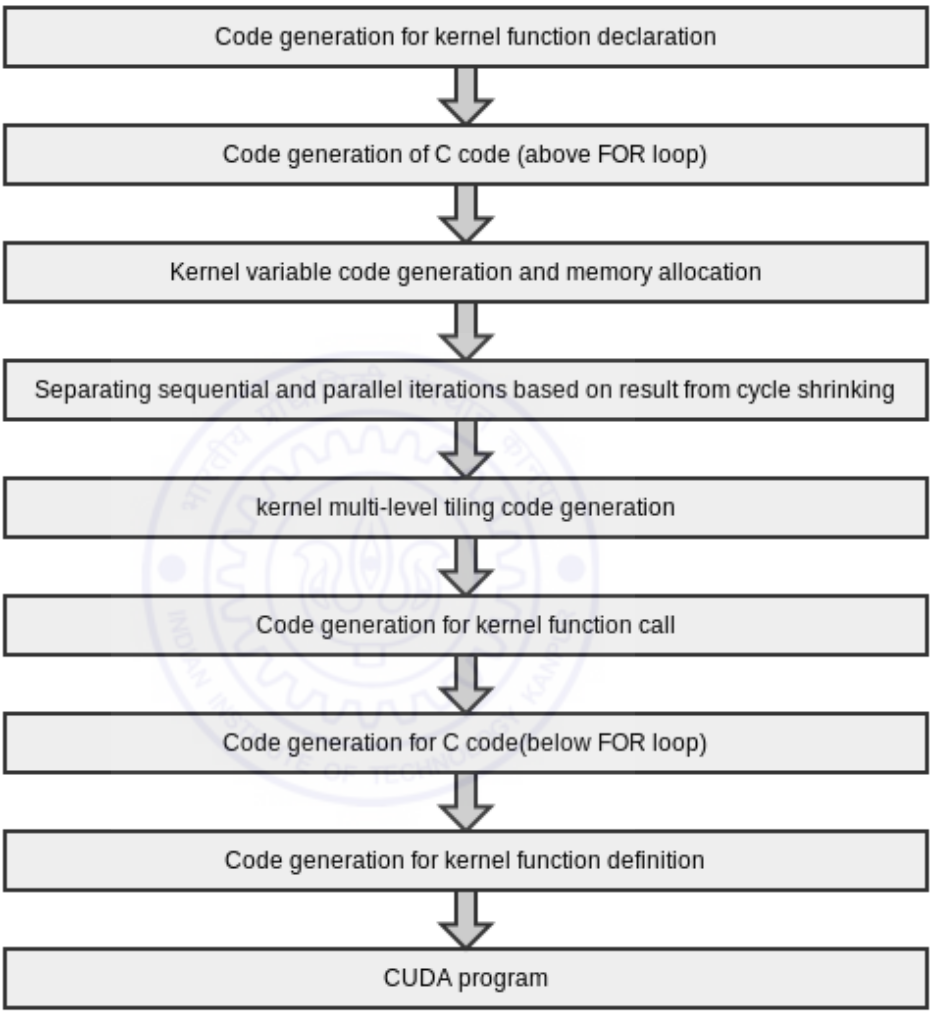


Figure 3.2: Code Generation Phase.

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 result from cycle shrinking: This step uses the result of cycle shriking phase(Reduction factor and DDV) and generates the code for the loop transformation.

Code generation for kernel function definition: This step generates the code for kernel function call. This function will specify the number of threads and blocks allocated to the function .

Code generation for C code (below FOR loop): This step of the code gener-

ation phase generates the code for the portion below FOR loop in the sequential C program.

Code generation for kernel function definition: This is the last step of code generation phase.It generates the code for the kernel function definition. In function body, we require indices to access the data available on device. So,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 processing all these steps, the final CUDA code is ready for execution on GPU.

3.5 How to use CRINK

In this section, we have briefly defined the overall working of CRINK with the help of an example by considering various user inputs for dependence test and cycle shrinking. As already discussed, CRINK uses GCD [WSO95] or Banerjee [AK04] for dependence test and Cycle shrinking [Pol88] or Extended cycle shrinking [SBS95] for parallelism extraction.

3.5.1 Working of tool explained with the help of an example

Firslty to run tool type the following command with choices for Dependency test and cycle shrinking:

```
./main_file -test:gcd/banerjee -shrink:simple/extShrinking1/extShrinking2
```

Consider the below FOR loop:

Example 10.

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

Process of converting the sequential input program to parallel CUDA program will pass through following phases:

Compilation Phase

In this phase input code is compiled using ROSE compiler.ROSE reads the input program and generate an Abstract Syntax Tree (AST) representing the structure of input program.

- **Loop Normalization**

  If loop is not normalized,then we need to normalize the FOR loop. In Example 10, loop is not normalized, so we apply loop normalization algorithm. After normalizing loop, we get:

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

- **Affine test**

  Now we need to check whether loop is affine or non-affine. In Example 10,array indices are linear function of loop variable i, so the loop is affine.

Dependence Test

To find whether intra-loop dependencies exist or not, we perform dependence test. Based on the user input i.e.`gcd` or `banerjee`, this phase identifies the dependency within a loop. After performing both tests on Example 10, following are the results corresponding to various user inputs:

- `gcd`: The linear equation for the normalized form of Example 10 is:

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

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

- `banerjee`: For banerjee test, we calculate upper and lower bounds for different direction vectors, for Example 10 these bounds will be:

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

It is clear that value of $(B_0 - A_0)$ does not satisfy any of the lower and upper bound constraints, so no dependence exists with any of the direction vector.

Parallelism Extraction

Based on the user input for cycle shrinking,we calculate the reduction factor as per follow:

- `simple` or `extShrinking1`: If user inputs simple or extShrinking1,then we calculate reduction factor,which is used to speed up the loop. Because there is only one reference pair i.e. $A[j+1] - A[j+11]$ in this example, both the distance vector and reduction factor (λ) are same. Therefore, reduction factor (λ)=10.

- `extShrinking2`: Extended cycle shrinking for variable dependence distance can only be applied to loops that contain array indices of the form $aj \pm b$,where $a > 1$ but for the array indices of Example 10, $a = 1$. Therefore, we can not apply this shrinking on the FOR loop.

Code Generation

Code generation process involves various routines which are discussed in detail in Chapter 3. The reduction factor calculated during parallelism extraction phase is taken as input to this phase. After applying all the routines,we get CUDA program corresponding to the input program.

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

- If user inputs `gcd` for dependence test and `simple` for cycle shrinking. Then GCD will detect dependence in the loop as already discussed in *Dependence Test* phase and calculate the reduction factor. Hence simple shrinking will

transform the loop as shown in Code 3.1.

Code 3.1: Loop Transformation using Simple Shrinking

```
int _SZ_A_0 = 10;
int *_DEV_A_0;
// Allocating device memory to the kernel variable
cudaMalloc((void**) &_DEV_A_0, 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 = 10;
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<=9;j+=10)
for(_CUDA_TILE=0;_CUDA_TILE<_NUM_TILE;_CUDA_TILE++){
        _AFFINE_KERNEL<<<_BLOCKS,_THREADS>>>(_DEV_A, _SZ_A_1, 1, j,
            0, 9, _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+10))&&(j<=CUDA_U_j)){
                A[1+j]=A[1+j+10]+10;}}
```

- If the user input is `gcd` and `extShrinking1` the parallel code generated will be as shown in Code 3.2.

  Code 3.2: Loop Transformation using Extended Cycle Shrinking for Constant Dependence Distance

```
int _SZ_A_1 = 10;
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 = 10;
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]={9};
for(ID_1=1;ID_1<=9/10+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]+10, 9), _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+10]+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 `extShrinking2` in place of `extShrinking1` in previous case, an error message gets generated saying "*Oops!! Wrong input. Please give simple or extShrinking1 as input*".

- Because banerjee test does not detect any dependence in the loop, irrespective of any user input for parallelism extraction, the parallel code shown in Code 3.3 will be generated.

Code 3.3: Loop Transformation when Dependency does not exists

```
int _SZ_A_1 = 10;
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 = 10;
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, 9,
            _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+10]+10;}}
```

Chapter 4

Experiments and results

In this chapter, we outline the experimental evaluation of cuda code generated by CRINK .The generated CUDA code is executed on the GPU machines and their performance in terms of computation time versus number of threads are illustrated through graphs.

4.1 Experimental Setup

The 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* 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. We have tested the CRINK on various standard programs available on [jbu] and results are demostrated with the help of graphs.

4.2 Standard Datasets

We have used the standard datasets [Dat] to test the performance of CRINK . These datasets are the 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]

4.3 Benchmarks

- **ARRAY_RETURN benchmark**

  ARRAY_RETURN is a benchmark designed to create arrays in functions and return them in argument list.

- **BACKTRACK_BINARY_RC**

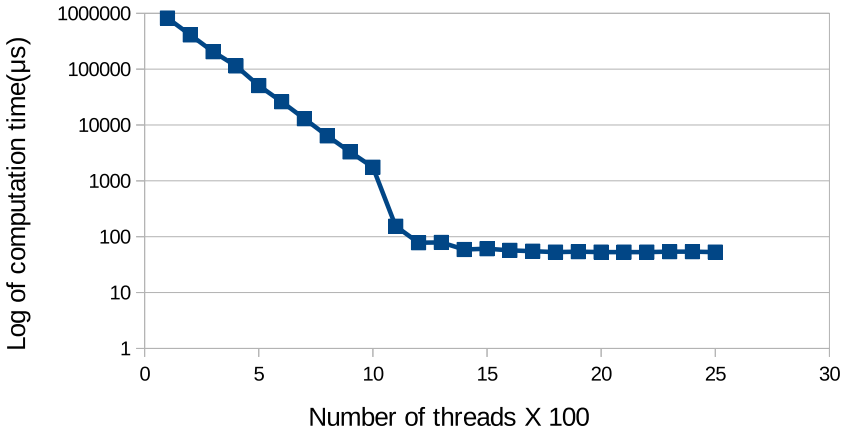
  BACKTRACK_BINARY_RC is a C library which carries out a backtrack search for a set of binary decisions, using reverse communication (RC)

- **CIRCLE_RULE** CIRCLE_RULE is a C library which computes quadrature rules for the unit circle in 2D, that is, the circumference of the circle of radius 1 and center (0,0).

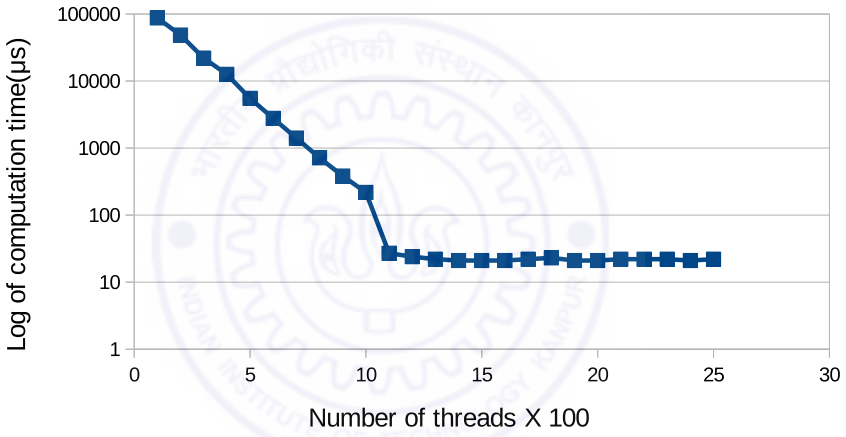
- **VALGRIND** VALGRIND is a directory of C programs which illustrate the use of VALGRIND, a suite of programs which includes a memory leak detector.

4.4 Performance Analysis

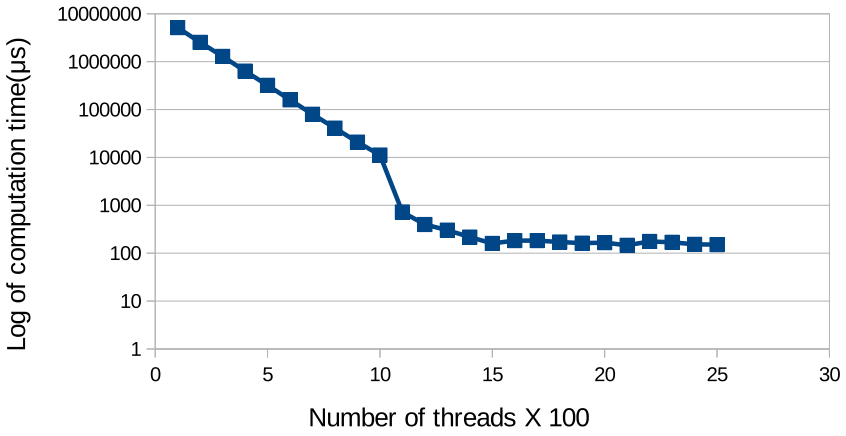
In this section,we calculate the computation time for each standard kernel generated by CRINK for different standard datasets and it has been shown in the following plots that as the number of threads increases, the computation time decreases initially for certain number of threads and later it becomes constant(shown in table 4.1). This happens due to the block overhead.



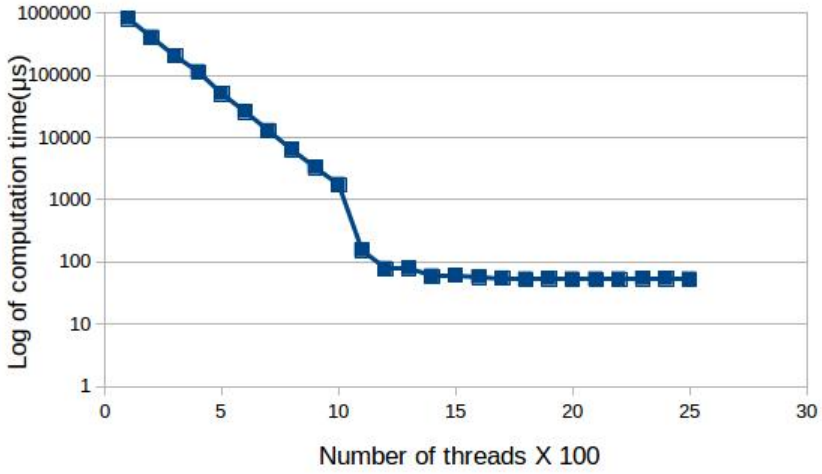
(a) Array Return with CSphd Dataset



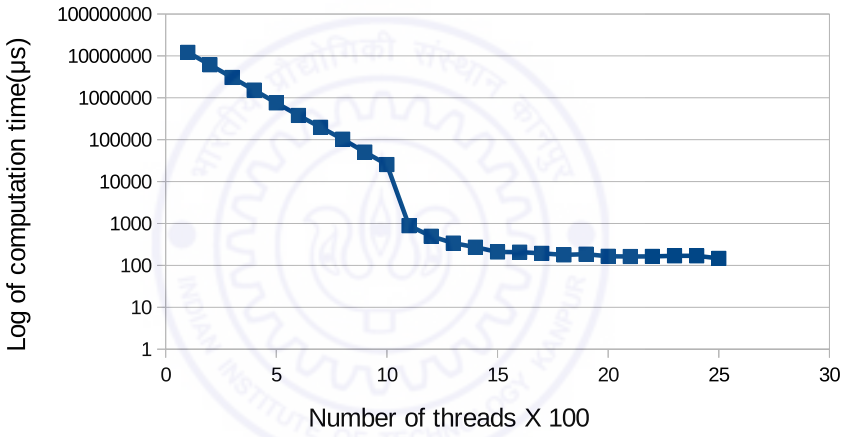
(b) Array Return with EVA Dataset



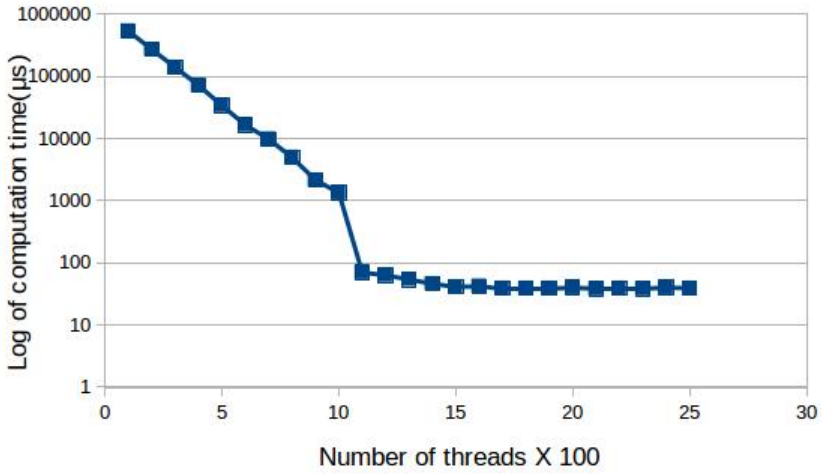
(c) Array Return with hepth Dataset



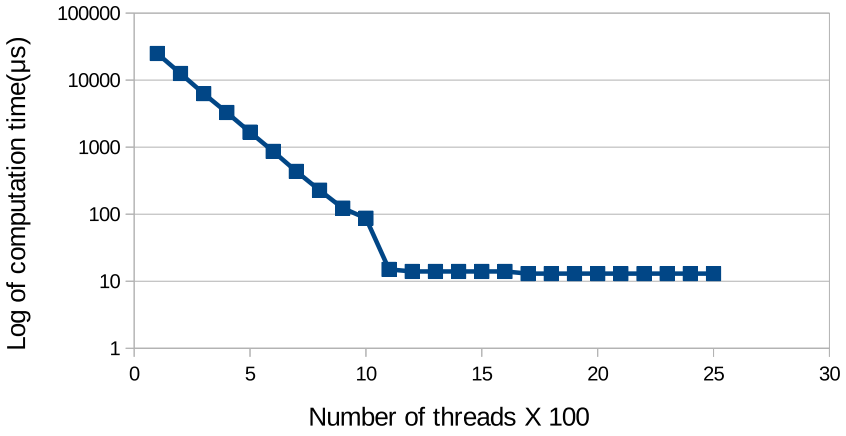
(d) Array Return with Barth5 Dataset



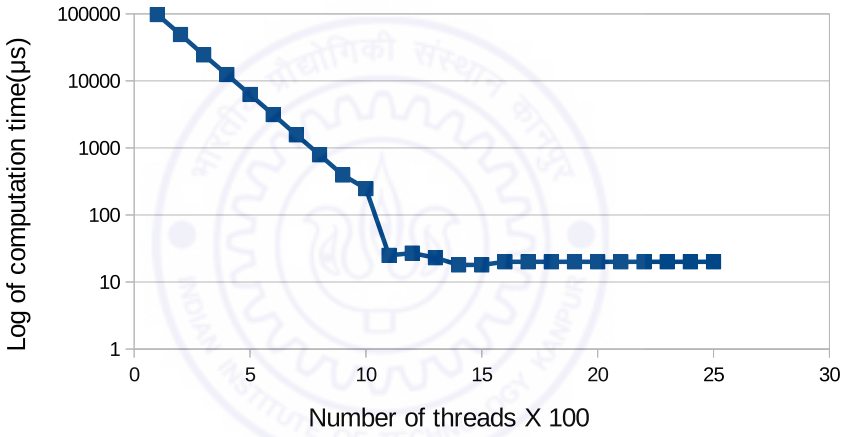
(e) Array Return with NotreDame_www Dataset



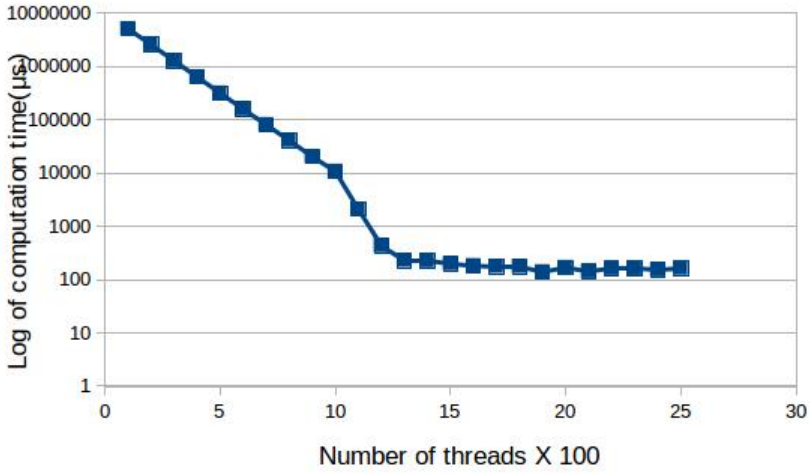
(f) Array Return with p2p-Gnutella04 Dataset



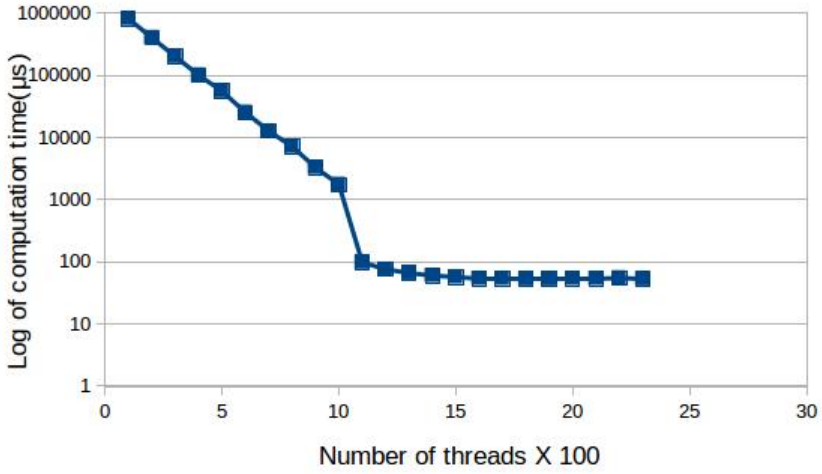
(a) Backtrack_binary_rc with CSphd Dataset



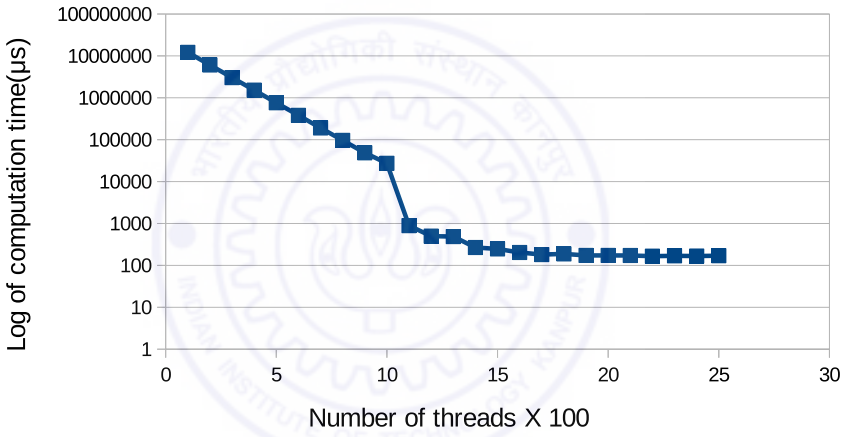
(b) Backtrack_binary_rc with EVA Dataset



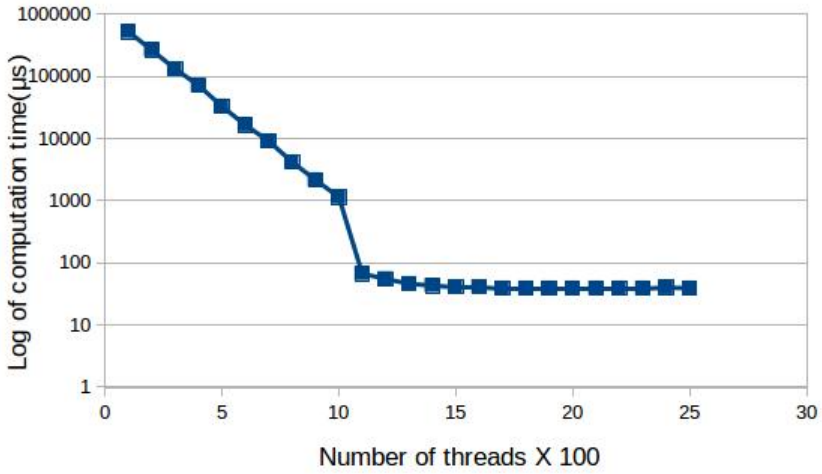
(c) Backtrack_binary_rc with hepth Dataset



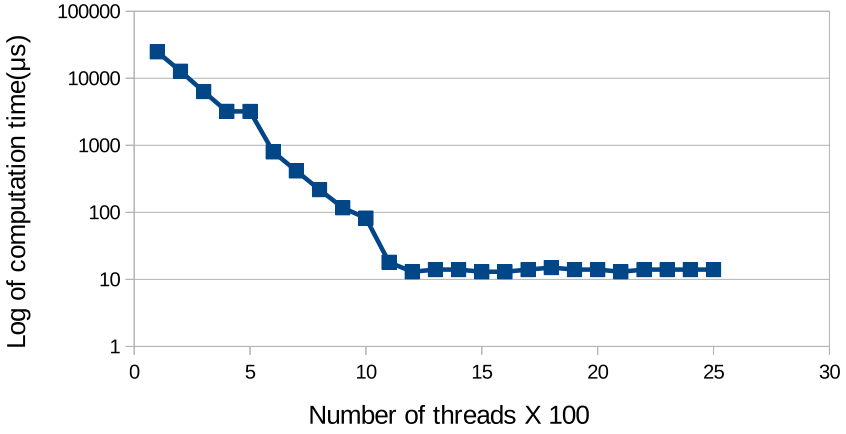
(d) Backtrack_binary_rc with Barth5 Dataset



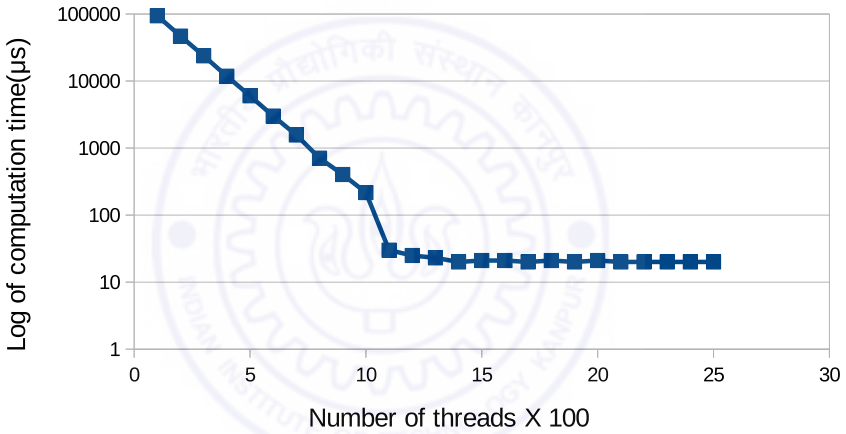
(e) Backtrack_binary_rc with NotreDame_www Dataset



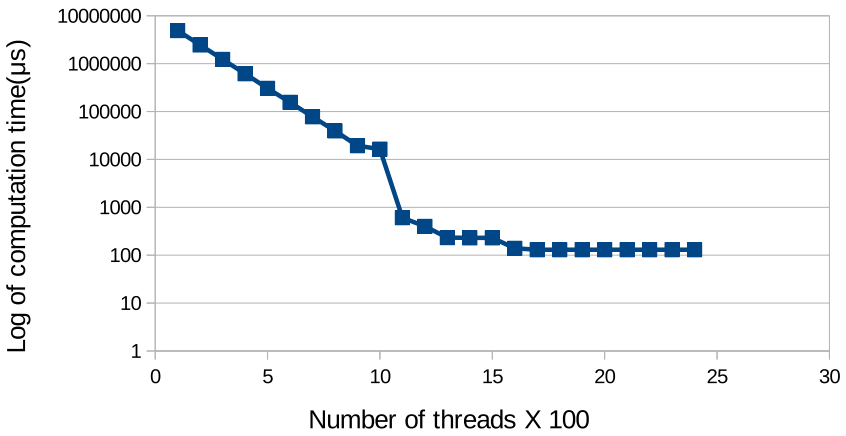
(f) Backtrack_binary_rc with p2p-Gnutella04 Dataset



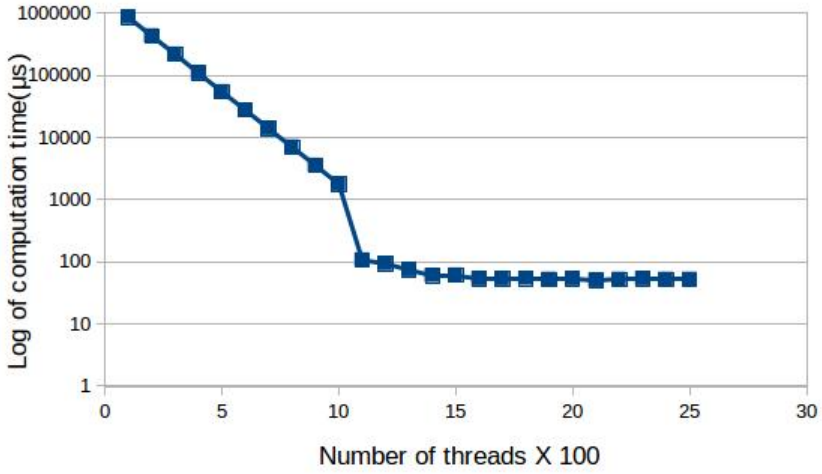
(a) Circle_rule with CSphd Dataset



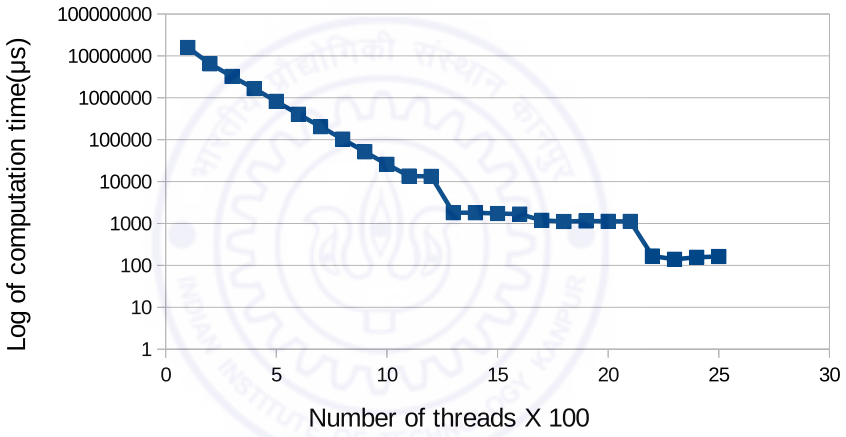
(b) Circle_rule with EVA Dataset



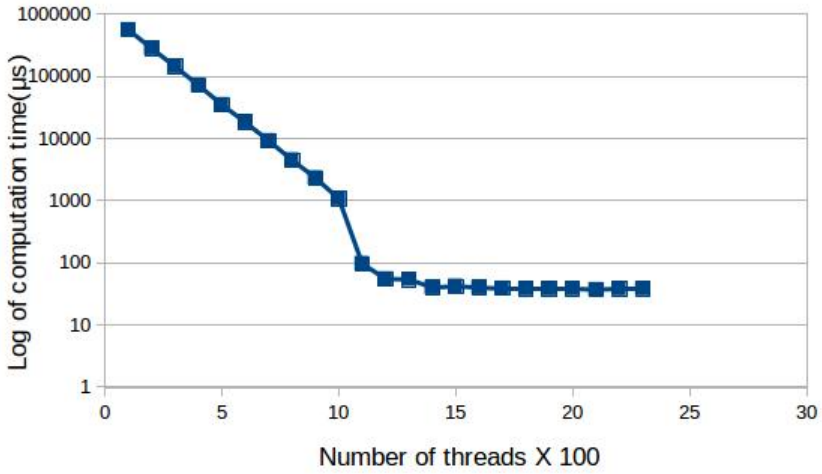
(c) Circle_rule with hepth Dataset



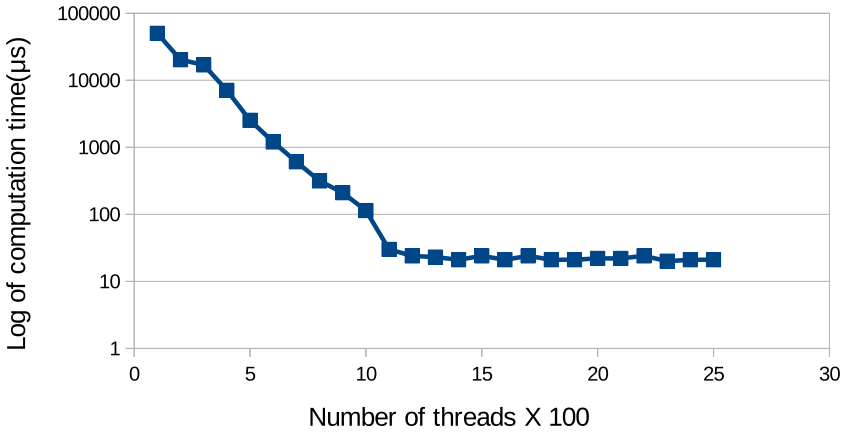
(d) Circle_rule with Barth5 Dataset



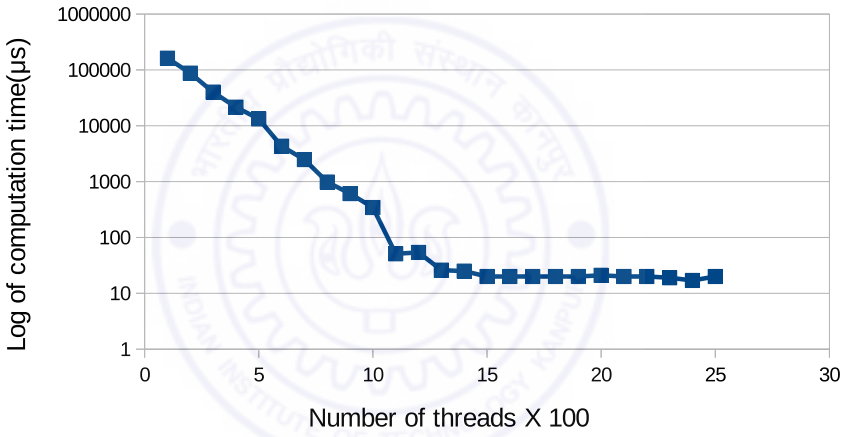
(e) Circle_rule with NotreDame_www Dataset



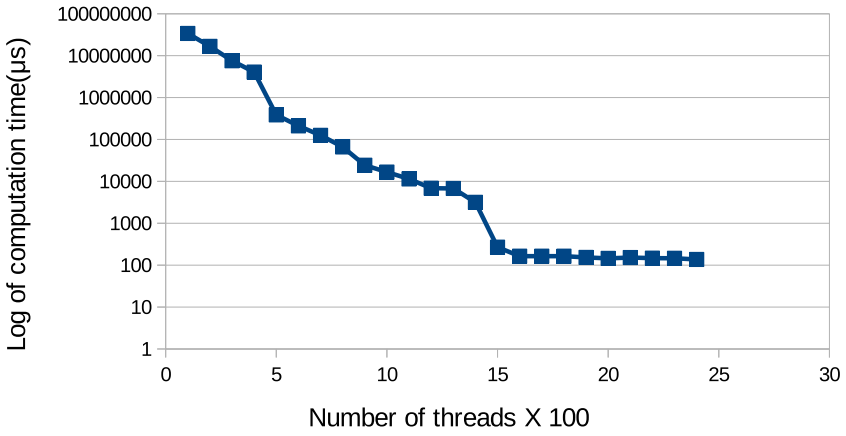
(f) Circle_rule with p2p-Gnutella04 Dataset



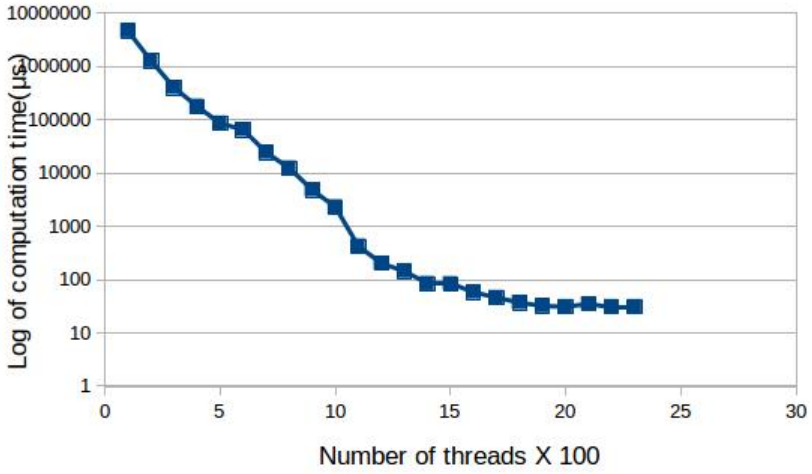
(a) Valgrind with CSphd Dataset



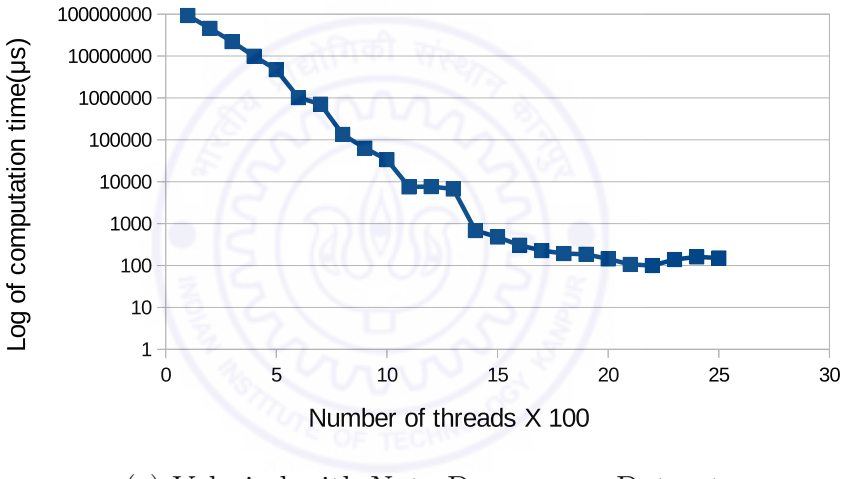
(b) Valgrind with EVA Dataset



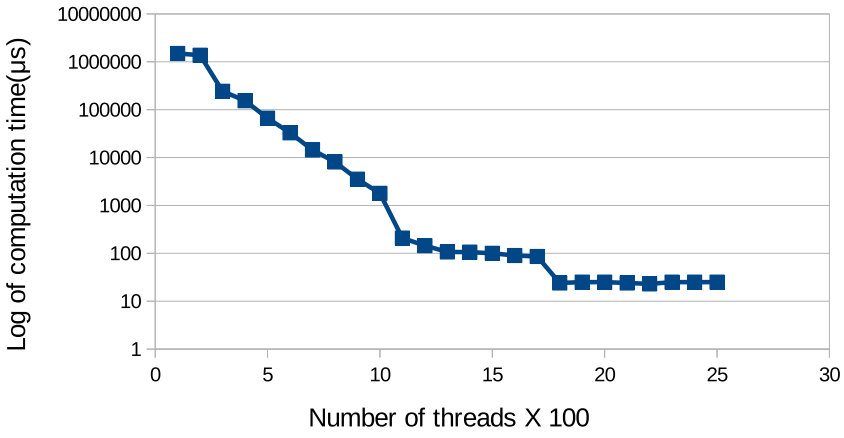
(c) Valgrind with hepth Dataset



(d) Valgrind with Barth5 Dataset



(e) Valgrind with NotreDame_www Dataset



(f) Valgrind with p2p-Gnutella04 Dataset

Benchmarks	Dataset	Time Taken by sequential C program(sec)	Data Transfer Time(sec)	Runtime(ms) (Threads x Blocks)			Limiting threads and corresponding time taken	
				128x1	512x8	512x64	Limiting Threads	Time Taken(ms)
Array_Return	CSphd	0.023106	0.0002	0.206	0.013	0.013	1024	0.013
	EVA	0.087395	0.002	0.716	0.022	0.021	4096	0.021
	barth5	0.821172	0.020000	6.388	0.079	0.057	32768	0.057
	p2p-Gnutella04	0.534134	0.010000	4.962	0.054	0.042	65536	0.039
	HEP-th-new	5.14159	0.130000	40.892	0.303	0.183	8388608	0.153
	NotreDame_www	12.124565	0.200000	102.881	0.341	0.207	16777216	0.147
Backtrack binary_rc	CSphd	0.024971	0.0002	0.228	0.013	0.013	65536	0.013
	EVA	0.097614	0.002	0.791	0.023	0.020	65536	0.020
	barth5	0.816178	0.020000	7.157	0.066	0.054	131072	0.053
	p2p-Gnutella04	0.52386	0.010000	4.193	0.046	0.041	65536	0.039
	HEP-th-new	5.080103	0.130000	40.766	0.229	0.181	1048576	0.141
	NotreDame_www	12.160771	0.20000	96.411	0.487	0.203	2097152	0.166
Circle_rule	CSphd	0.024793	0.0002	0.218	0.014	0.013	16384	0.013
	EVA	0.087395	0.002	0.707	0.023	0.021	16384	0.021
	barth5	0.858775	0.020000	7.013	0.074	0.053	32768	0.053
	p2p-Gnutella04	0.560026	0.010000	4.511	5.467	0.040	65536	0.039
	HEP-th-new	4.900622	0.130000	39.438	0.233	0.139	524288	0.139
	NotreDame_www	15.89101	0.20000	103.284	1.309	1.678	4194304	0.14
Valgrind	CSphd	0.050151	0.0002	0.315	0.023	0.021	8192	0.021
	EVA	0.162255	0.002	0.982	0.026	0.020	32768	0.020
	barth5	4.641323	0.020000	12.366	0.143	0.048	1048576	0.036
	p2p-Gnutella04	1.494457	0.010000	8.089	0.107	0.16	1048576	0.024
	HEP-th-new	33.926026	0.130000	66.263	0.676	1.012	2097152	0.147
	NotreDame_www	92.686521	0.20000	134.338	6.711	0.304	2097152	0.100

Table 4.1: Results on standard benchmarks

Chapter 5

Conclusion and Future Work

CRINK is a an end to end code transformation system that converts an input sequential C program ,using ROSE compiler,into a parallel CUDA program. The input program can be an affine or a non-affine C program. We are handling affine loops only. The tool is tested on standard benchmarks and datasets. This thesis work concludes that as the number of threads increases, the computation time reduces exponentially for certain number of threads which is shown in Chapter 4 , and later on it may become constant for large number of threads (value of that particular thread after which computation time becomes constant is shown in Table 4.1) .
Various challenges occur while developing this tool, thereby imposing some limitations over CRINK . Future work can include improvements over CRINK , some of which are listed below:

1. Handling non-affine loops.
2. Handling loops that contain both affine and non-affine dependencies.
3. Handling while loops.
4. Handling loops containing multidimensional arrays and nested loops.
5. Handling imperfectly nested loops.

5.1 Our contribution in the thesis

The need of parallelization arises because applications which contains loops, take huge computation time, when executed sequentially. Therefore we need some parallelization techniques to reduce the computation time.

CRINK is automatic source to source parallelization tool. It does not requires any effort from the programmer's side for parallelization and optimization of programs. Manual developement of the parallel code using a programming model is cumbersome,because then user needs to understand the programming language. In our tool CRINK , even if the user does not know about CUDA programming language ,then also he can use it easily. CRINK performs much better than sequential C programs which becomes clear from the table 4.1.

Bibliography

[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.

[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.

[cud] CUDA (compute unified device architecture). `http://ixbtlabs.com/articles3/video/cuda-1-p5.html`. 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.

[DQWc] R Vuduc T Panas Q Yi C Liao D Quinlan, M Schordan and JJ Will-coc. *Rose tutorial: A tool for building source-to-source translators.*

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

[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.

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

[Nvi] What is gpu accelerated computing. `http://www.nvidia.com/object/what-is-gpu-computing.html`.

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

[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.

[ros] *Rose compiler User manual*. Lawrence Livermore National Laboratory.

[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.

[Sin14] Akansha Singh. Crink: Automatic c to cuda code generation for affine programs. Master's thesis, Indian Institute of Technology, Kanpur, 2014.

[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.

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