

JOLOKIA $C++$: AN ANNOTATION BASED COMPILER FRAMEWORK FOR GPGPU

A Thesis Submitted

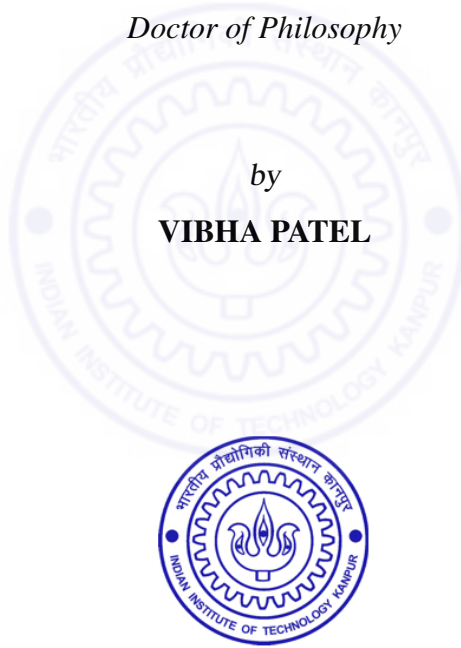
in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

by

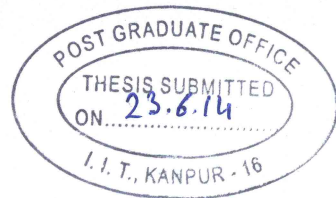
VIBHA PATEL



to the

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR, INDIA**

June 2014



CERTIFICATE

It is certified that the work contained in the thesis entitled “*JolokiaC++ : An Annotation based Compiler Framework for GPGPU*” by *Vibha Patel* has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

(Late) Dr. Sanjeev Aggarwal

Professor,

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur,
Kanpur-208016, INDIA.

Dr. Harish Karnick

Professor,

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur,
Kanpur-208016, INDIA.

Dr. Amey Karkare

Assitant Professor,

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur,
Kanpur-208016, INDIA.

Dr. Vivek Sarkar

Professor of Computer Science,

E.D. Butcher Chair in Engineering,

Rice University

Houston, Texas, USA

Abstract

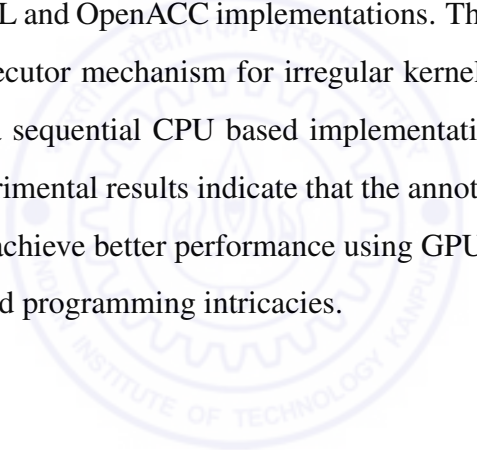
We present the design and implementation of a generic annotation based compiler framework, JolokiaC++, which generates high quality CUDA (Compute Unified Device Architecture) code for GPUs. The framework abstracts the details of the underlying hardware using annotations, thus allowing an end-user to write parallel programs without detailed knowledge about the hardware. The end-user can extract an acceptable level of performance from GPU hardware without worrying about low level details of the hardware like data allocation, memory organization and communication overhead.

The ultimate goal of the framework is to increase productivity without compromising performance. The proposed key ingredients to achieve the goals of productivity and performance are implicit and explicit annotations supported by task-level data flow analysis and operation-level data flow analysis. JolokiaC++ can also optimize irregular data applications on GPUs. We developed extensions for the generic parallel constructs that allow portable and efficient programming of codes with irregular accesses on the GPU.

We evaluate and show the effectiveness of our framework on kernels with regular and irregular accesses. The regular access kernels include Blackscholes, Matrix-Vector multiplication, Matrix-Matrix multiplication, Jacobi 1D & 2D, Heat 2D, Vector Addition and Convolution. We evaluated the performance of regular kernel on Nvidias GeForce 770 using CUDA version 5.5. The inspector-executor composition for irregular accesses in our framework is evaluated by generating synthetic data for aggregation benchmarks: MOL-DYN, IRRREG and NBF. We present experimental results from compiling the irregular

data kernels for execution on Fermi GTX 480, Tesla C1060 and Tesla K20c GPUs. The speedup of optimized JolokiaC++ implementation for regular kernels ranges from 0.89 - 4242.21 as compared to OpenCL which ranges from 0.42 - 571.59 and OpenACC which ranges from 1.08 - 526.08. The speedup of shared memory composition of irregular kernels ranges from 0.59 - 11.23 as compared to the composition without shared memory implementation which ranges from 0.5 - 7.74. The speedup of optimized JolokiaC++ when compared with hand-written OpenCL and OpenACC implementation for regular kernels ranges from 0.52 - 17.13 and 0.199 - 79.56 respectively.

The speedup of almost all the regular kernels using optimized JolokiaC++ is better when compared with OpenCL and OpenACC implementations. The shared memory composition of the inspector/executor mechanism for irregular kernels performs reasonably well when compared with a sequential CPU based implementation and without shared memory composition. Experimental results indicate that the annotation based framework can help domain experts to achieve better performance using GPUs without knowing the details of the architecture and programming intricacies.



Acknowledgements

At this moment of accomplishment, first of all I pay homage to my guide, Late. Prof. Sanjeev K. Aggarwal. This work would not have been possible without his guidance, support and encouragement academically as well as personally. He has been a wonderful mentor and a facilitator. Under his guidance I successfully overcame many difficulties and learned a lot. Despite of his extremely busy schedule, he has always accommodated me whenever I needed his guidance. His unflinching courage and conviction will always inspire me, and I hope to continue to work with his noble thoughts. I wish his soul roots in peace and solace in the heaven. I can only say a proper thanks to him through my future work.

I am also extremely indebted to my guide Prof. Harish Karnick, for picking me up as a student at the critical stage of my Ph.D. I warmly thank my co-guide, Prof. Amey Karkare, for his valuable advice, constructive criticism and his extensive discussions around my work. I would like to express my deepest gratitude to my co-guide, Prof. Vivek Sarkar, for his support and guidance in my thesis. I am thankful to all the faculty members of the department to provide a research conducive environment.

My life as a PhD student has been full of fun, thanks to my dear friends Umarani, Surya, Rohit, Purushottam, Amrita, Ajitha, Saiful, Puneet, Pawan and the entire class of MTech 2008. Thanks to the administrative staff members for making all the paper work hassle-free. Thanks to the lab staff members for technical and infrastructural support. I especially thank Research-I foundation for funding my visits to conferences. I

acknowledge the financial support for my PhD from the Ministry of Human Resources and Development.

I am grateful for the unconditional and unquestioning support from my loving husband throughout my work. My very existence is indebted to my parents and I thank them to make me what I am today. I would like to pay high regards to my brother and sisters for their sincere encouragement and inspiration throughout my research work and lifting me uphill this phase of life.

Last but not the least, I thank the God almighty for giving me strength all my life.





Dedicated

to

My Family Members & Respected Teachers



Contents

List of Tables	xiv
List of Figures	xvi
List of Algorithms	xix
List of Abbreviations	xxi
1 Introduction	1
1.1 Key Challenges for General Purpose Programming on GPU	3
1.1.1 Lack of Memory Hierarchy Management support	3
1.1.2 Lack of Language Support	4
1.1.3 Thread Block Synchronization on GPU	4
1.1.4 Inadequate Parameter Modeling Support	4
1.2 GPU Programming using JolokiaC++	5
1.3 Our Contributions	5
1.4 Organization of the Thesis	8
2 Background and Related Work	9
2.1 Basic Terms	9
2.2 Background	10
2.2.1 GPU Parallel Computing Architecture	10

2.2.2	Execution Model	11
2.2.3	CUDA Programming Model	12
2.3	Related Work	12
3	Language Design	21
3.1	Design of JolokiaC++ Framework	23
3.1.1	JolokiaC++ Programming Model	24
3.1.2	JolokiaC++ Annotations	25
3.1.3	Task-Level Data Flow Graph	30
3.1.4	Operation-Level Data Flow Graph	31
4	Regular Applications	37
4.1	Memory Access Operators	38
4.2	Compiling high level JolokiaC++ constructs	42
4.2.1	Task-Level Data Flow Analysis for Optimizing Communication .	46
4.2.2	Operation-Level Data Flow Analysis for Optimizing Memory Ac- cess	47
5	Irregular Data Applications	55
5.1	Preliminaries of the Framework	58
5.2	Optimization of Irregular Applications	59
5.2.1	Flow Analysis Framework for GPU	61
5.2.2	Code Generation using Sparse Polyhedral Framework	65
5.2.3	Executor Code Generation	73
5.2.4	Inter-Block synchronization using lock-free barrier	74
5.3	Empirical Search for Selection of Optimal Tile Size and Scheduling Policy	76
6	Performance Measurement	79

6.1	Experimental Methodology and Performance Results for Regular Access	
	Kernels	79
6.1.1	Ease of Programming	81
6.1.2	Performance Results and Discussion	84
6.2	Experimental Evaluation of Irregular Access Kernels	85
7	Conclusion and Future Work	101
	Appendices	103
A	Annotation Grammar Specification	105
	Publications	119





List of Tables

3.1	CPU-GPU Memory Space Management Annotations	28
6.1	Test Set and their Configuration	80
6.2	Execution time of benchmarks in milliseconds	83
6.3	Speedup of the benchmarks over sequential CPU implementations	83
6.4	Configuration of GPUs	85
6.5	Execution time of Sequential (Seq) and Parallel (Par) hand-coded CPU implementation in milliseconds	86
6.6	Execution Time of IRREG in milliseconds	87
6.7	Execution Time of MOLDYN in milliseconds	88
6.8	Execution Time of NBF in milliseconds	89
6.9	Average Reordering Time in milliseconds	89



List of Figures

2.1	CUDA Parallel Computing Architecture and Programming Model	11
3.1	Compilation Framework of JolokiaC++	22
3.2	Translation spaces in JolokiaC++	24
3.3	Translation of JolokiaC++ code	25
3.4	Task-Level Data Flow Graph	32
4.1	Memory Access Patterns and their Respective Operator Annotations with their Linearized Representation.	38
4.2	Inter and Intra-thread access of elements in absence of scratchpad anno- tation	41
4.3	Inter and Intra-thread access of elements in presence of scratchpad anno- tation	42
4.4	Memory Access Pattern of Naive Matrix-Vector Multiplication without Scratchpad Annotation	51
4.5	Memory Access Pattern of Matrix-Vector Multiplication using Scratch- pad Annotation	52
4.6	Operation-Level Data Flow Tree for Matrix-Vector Multiplication	53
4.7	Composed Task and Operation-Level Data Flow Tree	54
5.1	Irregular Memory Access	56

5.2	Compiler Framework for Optimizing Irregular Applications	58
5.3	Loop Flow Graph for the Code in Listing 5.1	60
5.4	Scatter/Gather for Simplified MOLDYN Kernel	66
5.5	Compile Time Composition of Inspector	73
6.1	Performance of Matrix Multiplication	90
6.2	Performance of Matrix Vector Multiplication	91
6.3	Performance of Jacobi 2D	92
6.4	Performance of Heat 2D	93
6.5	Performance of Convolution	94
6.6	Performance of 1D Benchmarks	95
6.7	Scatter plot of 1000 molecules	96
6.8	Performance of IRREG kernel	97
6.9	Performance of MOLDYN kernel	98
6.10	Performance of NBF kernel	99
6.11	Data Transfer Overhead	100

List of Algorithms

4.1	Determining Block and Grid Dimension for linearized array	43
4.2	Global Task-Level Data Flow Analysis	48
4.3	Task-Level Data Flow Variables Analysis	48
4.4	Memory Access Optimization	50
5.1	Loop Flow Analysis	62
5.2	Global Flow Analysis	63
5.3	Automatic Shared Memory Tiling	64
5.4	First Touch Policy	69
5.5	Wavefront Generation	72
5.6	Wavefront Regularization	72
5.7	Intra-Loop Memory Access Mechanism	74
5.8	Lock-free Interblock Barrier	76



List of Abbreviations

GPU	Graphics Processing Unit
GPGPU	General-Purpose Computing on Graphics Processing Units
CUDA	Compute Unified Device Architecture
OpenACC	Open Accelerators
OpenCL	Open Computing Language
SM	Streaming Multiprocessor
SP	Streaming Processor
SIMT	Single Instruction Multiple Thread
DRAM	Dynamic Random Access Memory
AST	Abstract Syntax Tree
BLAS	Basic Linear Algebra Subprograms
TFG	Task-Level Data Flow Graph
TFV	Task-Level Data Flow Variable
AO	Array Object
OFG	Operation-Level Data Flow Graph
LFV	Loop Flow Variable
LFG	Loop Flow Graph
AP	Array Part
FT	First Touch
FPGA	Field Programmable Gate Array



Chapter 1

Introduction

Processor designers have turned towards architectures with increased degree of explicit parallelism in response to the challenges faced by frequency scaling. Today's hardware offerings range from general purpose chips with a few cores to many cores graphics processors (GPUs) that support large-scale data parallel computations. Graphics Processing Units (GPUs), with many-core architecture have led the race due to their floating point performance. However, adopting these architectures has been a process plagued with legacy issues. The problem of partitioning existing single-threaded applications to maximally utilize available multiple cores has been a challenging issue. Languages like OpenCL [11] and CUDA [2] greatly improved speed and responsiveness for a wide spectrum of applications by providing a standard interface for general-purpose programming of GPUs. However, using these languages effectively requires explicit management of numerous low-level details which involves use of communication and synchronization constructs. This burden makes GPU programming difficult and error-prone, preventing wide access to these powerful devices for most programmers. While high level abstractions serve well for parallel programming, their semantics is based predominantly on what happens in sequential architectures. As a result compilers for these languages provide limited scalability in performance for complex programming architectures like GPUs.

An alternate to writing parallel programs is to have a parallelizing compiler that can automatically parallelize sequential applications. Auto-parallelization differs from parallel programming in that the programmer does not have to worry about using parallel constructs in the program [12, 41]. It is the compiler's responsibility to take advantage of the parallelism existing in the underlying architecture. The advantage of this approach is that existing/legacy applications need not be modified, e.g. applications just need to be recompiled with a parallel compiler. Therefore, programmers need not learn new programming paradigms. However, this is achieved at the cost of reduced performance improvement.

The work presented in this thesis is based on a third approach, which is an annotation based approach. We present the design and implementation of a generic annotation based compiler framework that can be used to program GPGPUs where the programmer is only required to have a clear idea about those parts of a program that must be parallelized. We call our framework *JolokiaC++*. *JolokiaC++* uses annotations to extend the C++ language to program GPUs. A *JolokiaC++* programmer can exploit the parallelism of GPUs using annotations, without writing complex low level code as required by other mainstream approaches (OpenCL or CUDA). The *JolokiaC++* compiler translates a high-level annotated C program to efficient low-level CUDA code. This makes GPUs more accessible while effectively exploiting their computational power. We demonstrate the utility of *JolokiaC++* by providing a comparison between *JolokiaC++* code and CUDA code to show the ease of programming as also the performance improvement obtained when using *JolokiaC++*.

GPUs have become widely used for general-purpose computation, and have the potential to achieve high peak compute rates. This appealing property comes from the massively parallel architecture of GPUs. However, this leads to high sensitivity in their throughput to the presence of irregularities in memory access patterns in an application. Irregularities in an application may degrade GPUs performance by as much as an order of

magnitude. When the memory access pattern is regular, the GPUs perform extremely well using high level programming models like OpenACC [10, 4]. However, a large number of interesting applications have irregular data access patterns. Efficient parallelization of codes with irregular accesses on the GPU is still a challenging problem. Hence, it is important to develop mechanisms which can help generate efficient parallel code for applications with irregular memory accesses. We overcome this problem by generating schedules which can regroup data and iterations of loop kernel in such a way that the number of consecutive independent iterations is maximized for execution on streaming multiprocessors.

1.1 Key Challenges for General Purpose Programming on GPU

We summarize the key challenges for obtaining high performance from general purpose GPU code as they are the main focus of our proposed framework.

1.1.1 Lack of Memory Hierarchy Management support

Performance of applications is highly dependent upon the efficient utilization of memory hierarchy. Unlike cache based systems the memory hierarchy of GPUs is under control of the programmer. GPUs use scratch-pad memory which requires explicit instructions to move data from global memory. Lack of compiler support to manage scratch-pad memory is a motivating factor to automate this process. Further, random access in irregular kernels leads to non-coalesced global access which reduces memory bandwidth utilization in GPU.

1.1.2 Lack of Language Support

GPUs lack high level language support. Programmers need to know architectural intricacies to effectively utilize GPU features. We provide compile time and runtime support for a generic parallel construct like *for* in *JolokiaC++* to make it work on GPUs. We provide operators like *gpuIn* and *gpuOut* to simplify the data transfer between the CPU and GPU. In addition to this we introduce several operator annotations to guide the process of memory hierarchy optimizations.

1.1.3 Thread Block Synchronization on GPU

In CUDA, `__syncthreads()` is the barrier function which ensures proper intra-block communication. However, there is no explicit support for inter-block communication. Currently, this type of data communication occurs via global memory. It is followed by barrier synchronization via the CPU. That is, the barrier is implemented by terminating the current kernel's execution and relaunching the kernel. This is an expensive operation. To overcome this problem we use a lock-free barrier implementation for inter-block communication.

1.1.4 Inadequate Parameter Modeling Support

Blocks and threads are the basic unit of execution on GPUs. However, the programmer has to experiment with a number of blocks and threads to get optimal performance. We provide annotations to model the number of threads per block to gauge the performance. To support modeling of parameters on GPU, we provide *tile* and *scratchpad* operators.

1.2 GPU Programming using JolokiaC++

In this section, we illustrate the usefulness of programming with JolokiaC++ using an example.

Example 1.1: Listing 1.1 presents JolokiaC++ code to add two vectors, each containing 32 bit floating point numbers. A, B and C in the code are representatives for float arrays in C space. The parameters to *gpuIn* are the objects that are to be copied from CPU to GPU, while parameters to *gpuOut* are the objects to be copied from GPU to CPU. The *tile* operator provides information required for determining block and grid dimensions.

The snippet in Listing 1.2 shows the CUDA code required to perform the same task. It is also the code we expect the translator to generate while compiling the program in Listing 1.1. Notice the amount of low level detail that is hidden by the high level annotations in Listing 1.1. This illustrates the advantage of using JolokiaC++ as it allows the programmer to write simple high level code without compromising performance on a multicore architecture like GPU. ■

The annotations like *gpuIn*, *gpuOut* are provided by the user to assist JolokiaC++ compiler in code generation. In addition to these explicit annotations, JolokiaC++ can automatically infer information associated with array objects as implicit annotations to allow accessing array elements through primitive array types.

1.3 Our Contributions

We have made the following contributions:

- We describe the design and implementation of JolokiaC++, a compiler framework to generate CUDA code from high-level language abstractions provided through implicit and explicit annotations. We explore compiler techniques to recognize high-level abstractions to exploit their semantics for parallelization.

```

void vecAdd(f32Array &A,f32Array &B,f32Array &C){
    A = B + C
}
int main(){
    f32Array A(N),B(N),C(N);
    /*---- Task call site ----*/
    #pragma jolokia gpuIn(A,B,C) gpuOut(A)
        tile(BLK1,1,1)
    vecAdd(A,B,C);          /* GPU Task */
    return 0;
}

```

Listing 1.1: JolokiaC++ Code for Vector Addition

```

#define N 4096

__global__ void vecAdd(int *A, int *B, int *C){
int idx = threadIdx.x + blockIdx.x * blockDim.x;
if(idx < N){
    C[idx] = A[idx] + B[idx];
}
}

int main (void){
    int Host_a[N], Host_b[N], Host_c[N];          //Host array
    int *Device_a , *Device_b, *Device_c;        //Device array
    // Initialize Host array
    ...
    // Declare and initialize grid and block dimensions
    ...
    //Allocate the memory on the GPU
    cudaMalloc((void **)&Device_a, N*sizeof(int));
    cudaMalloc((void **)&Device_b, N*sizeof(int));
    cudaMalloc((void **)&Device_c, N*sizeof(int));
    //Copy Host array to Device array
    cudaMemcpy(Device_a, Host_a, N*sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(Device_b, Host_b, N*sizeof(int), cudaMemcpyHostToDevice);

    //Make a call to GPU kernel
    vecAdd <<< gridDim, blockDim>>> (Device_a, Device_b, Device_c );
    //Copy back to Host array from Device array
    cudaMemcpy(Host_c, Device_c, N*sizeof(int), cudaMemcpyDeviceToHost);
    //Free the Device array memory
    cudaFree(Device_a);
    cudaFree(Device_b);
    cudaFree(Device_c);
    return 0;
}

```

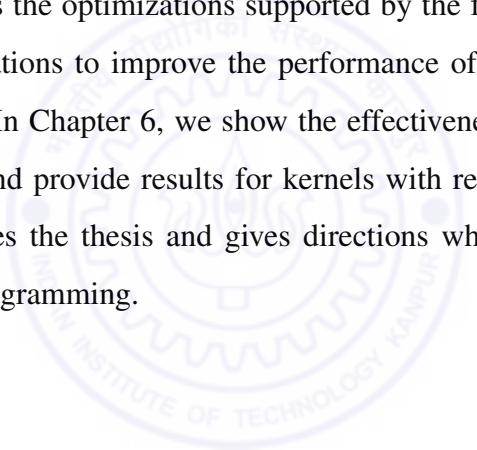
Listing 1.2: CUDA Code for Vector Addition

- We present a compiler system extension to automate parallelization of data parallel regular kernels on GPUs. We introduce a set of automatic optimizations for GPU architectures, that includes memory optimizations that improve locality, reduce bank conflicts, and permit vectorization. The framework analyzes the operators supported by the language, identifies the off-chip memory access patterns, and optimizes the memory accesses through vectorization and coalescing to achieve high data access bandwidth. These optimizations are implemented in JolokiaC++.
- We present a compiler and runtime system extension to automate parallelization of irregular kernels with subscripted subscripts for execution on GPUs. We use a combination of compile time analysis and composition of runtime data and iteration reordering transformations to optimize the performance of irregular kernels with irregular memory accesses on GPUs.
- We evaluate the performance of the code generated by JolokiaC++ for kernels like Blackscholes, Matrix-Vector multiplication, Matrix-Matrix multiplication, Jacobi 1D and 2D, Heat 2D, Vector Addition and Convolution. We also evaluate the effectiveness of the framework for irregular memory access kernels: IRREG, MOLDYN and NBF. The performance of the kernels is presented both in terms of execution time and speedup. The speedup of optimized JolokiaC++ implementation for regular kernels ranges from 0.89 - 4242.21 as compared to OpenCL which ranges from 0.42 - 571.59 and OpenACC which ranges from 1.08 - 526.08. The speedup of shared memory composition of irregular kernels ranges from 0.59 - 11.23 as compared to the composition without shared implementation which ranges from 0.5 - 7.74. The speedup of almost all the regular kernels using optimized JolokiaC++ is better when compared with OpenCL and OpenACC implementations. The shared memory composition of the inspector/executor mechanism for irregular kernels performs reasonably well when compared with a sequential CPU based implementa-

tion and without shared composition. The speedup of optimized JolokiaC++ when compared with hand-written OpenCL and OpenACC implementation for regular kernels ranges from 0.52 - 17.13 and 0.199 - 79.56 respectively.

1.4 Organization of the Thesis

The rest of the thesis is organized as follows. Background and Related work is discussed in Chapter 2. Chapter 3 describes the design and implementation of the language for annotations in JolokiaC++. Chapter 4 describes how to use language construct for regular kernels. It also discusses the optimizations supported by the framework. In Chapter 5, we describe the optimizations to improve the performance of irregular kernels with indirect memory accesses. In Chapter 6, we show the effectiveness of our approach on various benchmark codes and provide results for kernels with regular and irregular accesses. Chapter 7 concludes the thesis and gives directions which can be pursued to further improve GPGPU programming.



Chapter 2

Background and Related Work

Multi-core and many-core architectures have emerged as an elegant solution to meet the increased computational requirements of current applications while avoiding problems like chip overheating. The GPU is probably the dominant massively parallel architecture that is available. Nevertheless, taking advantage of this architecture has been a major concern for traditional serial programmers. Writing parallel codes for existing/new problems to work on current GPUs is a non-trivial task. One of the most difficult tasks in moving from serial to parallel programming is in fact adopting a completely new mindset. CUDA (Compute Unified Device Architecture) programming models high performance implementations for general purpose computational tasks on NVIDIA GPUs. However, manual development of optimized CUDA code for efficient data access is non-trivial. Hence, source to source translation of sequential programs to efficient multi-threaded CUDA programs is of interest for GPGPU programmers. In this chapter we present aspects which will help understand our work and relate it to other work in the field.

2.1 Basic Terms

We define some basic terms used throughout the thesis.

Definition 2.1. (Kernel) A function loaded onto the device (GPU) by the host (CPU) command is called a *kernel*.

Definition 2.2. (Annotation) An *annotation* in the JolokiaC++ programming language is a form of syntactic metadata used to annotate variables, operators, and functions.

Definition 2.3. (Pragma) A *pragma* is an explicit annotation interpreted differently in order to address the requirements of GPU hardware.

Definition 2.4. (Stencil Codes) *Stencil Codes* are computations that involve repeated updating of values connected with points on a multi-dimensional grid, utilizing only the values in a set of neighboring points.

2.2 Background

As our work develops annotation based compiler framework for GPUs, and generates CUDA code for such devices, we discuss the GPU parallel computing architecture and CUDA programming model in detail in this section.

2.2.1 GPU Parallel Computing Architecture

The GPU parallel computing architecture contains a set of multiprocessors. Each streaming multiprocessor (SM) contains a set of processing cores called streaming processors (SPs). Fig. 2.1(a) shows the GPU parallel computing architecture with different memory spaces: global memory, shared memory, constant cache, texture cache, and registers. The off-chip global memory is a large memory and has very high latency. The shared memory is an on-chip memory present in each SM and is organized into banks. When multiple addresses belonging to the same bank are accessed at the same time, it results in bank conflict. Each SM has a set of registers associated with it. The constant and texture memories are read-only regions in the global memory space and they have on-chip read-only

caches. Accessing constant cache is faster, but it has only a single port and hence it is beneficial when multiple processor cores load the same value from the cache. Texture cache has higher latency than constant cache, but it does not suffer greatly when memory read accesses are irregular and it is also beneficial for accessing data with spatial locality.

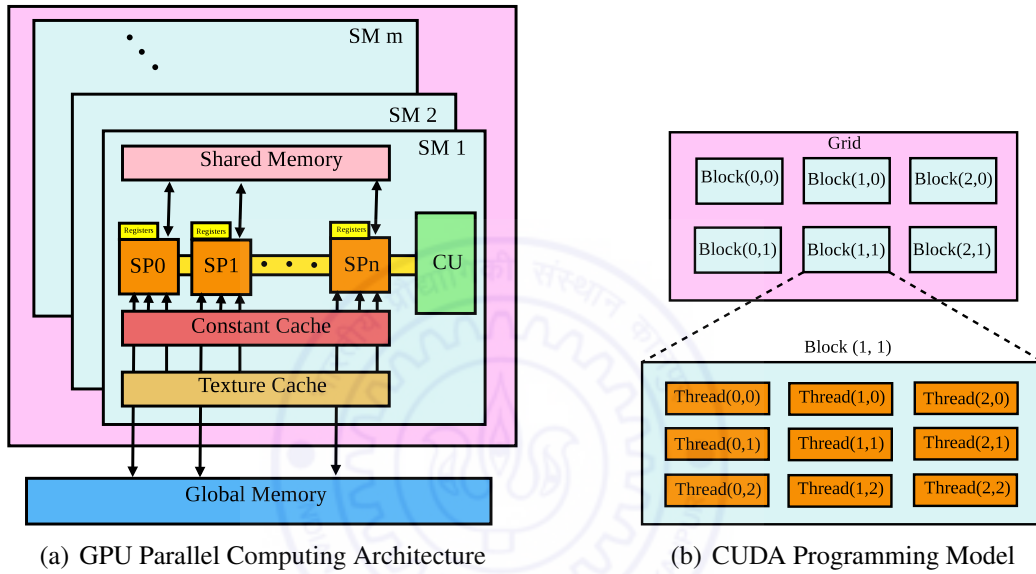


Figure 2.1: CUDA Parallel Computing Architecture and Programming Model

2.2.2 Execution Model

The parallel portions of an application are executed on the device (GPU) as kernels, with one kernel executed at a time. A CUDA kernel launches a grid of thread blocks (as shown in Figure 2.1(b)), a group of threads that should be executed concurrently. Each thread block consists of several warps, which are much smaller groups of threads. A warp is the smallest unit of hardware execution. The SM executes instructions from a warp in an SIMT (Single- Instruction Multiple-Thread) fashion. In SIMT execution, a single instruction is fetched and all the threads in the warp execute the same instruction in lockstep, except when there is control divergence.

2.2.3 CUDA Programming Model

CUDA is NVIDIA's parallel computing architecture for GPUs. It is an interface which enables programmers to access the highly parallel hardware of programmable GPUs. CUDA is an extension of the C programming language, with the CUDA runtime library providing a collection of routines for device memory management, host-device stream synchronization, and execution control functions (among others). CUDA is a data parallel SIMT (Single Instruction Multiple Thread) architecture, in which the same programmer defined kernels execute on all launched threads. These threads are launched in batches of blocks and grids, where blocks are collections of threads and grids are collections of blocks.

Kernel Performance Tuning

When all threads of a warp execute a load, if all accessed locations fall into the same section, only one DRAM request will be made and the access is fully coalesced. When the access locations spread across burst section boundaries, coalescing fails, multiple DRAM requests are made and the access is not fully coalesced. Efficient use of global memory (DRAM) bandwidth is one of the important performance considerations for massively parallel processors.

2.3 Related Work

GPUs are widely used for general-purpose computation and have the potential to achieve high peak compute rates. Programming models such as NVIDIA's Compute Unified Device Architecture (CUDA) [2, 3] and Khronos Group's Open Compute Language (OpenCL) [11, 35] facilitate general purpose programming for GPUs through APIs that expose the low-level details of the device architecture to the programmer. The programmer is expected to manually tune low-level code for a specific device in order to fully exploit its processing capability. On the other hand, OpenACC [10, 4] provides high level

abstractions for accessing GPUs but it is unable to give good gains for codes with irregular memory access patterns. Rapidmind [58] is a parallel development framework that allows the user to write parallel programs using standard C++ language. The framework uses dynamic compilation to execute the program in parallel and targets x86, Cell BE and GPGPUs. PeakStream [53] is a parallel development framework similar to RapidMind. PeakStream programs are compiled into an intermediate language using a custom compiler and then just-in-time compiled at execution time. Both, RapidMind and PeakStream are not publicly available.

Sarkar et. al. [68] describes a matrix and non-matrix based generic framework for representing iteration-reordering transformations. Li et. al. [46] presents a transformation framework which performs Λ -transformations like permutation, skewing and reversal, as well as a transformation called loop scaling. An algebraic representation based unified data and control transformation for distributed shared memory machines is presented by Cierniak et. al. [25]. The advances in automatic parallelization and optimization of programs are to large extent based on the use of polyhedral model [19, 32, 21]. Baskaran et. al. [16, 15] gave a polyhedral model based compiler framework for affine loops. Their framework performs an empirical search for determining best loop transformation parameters, which includes loop tiling sizes and unrolling factors.

Polyhedral Parallel Code Generation (PPCG) for CUDA with multilevel tiling strategy and a code generation scheme for the parallelization and locality optimization of imperfectly nested loops is introduced by Verdooaege et. al. [74]. In order to overcome the load imbalance which may occur due to pipeline fill-ups and drain delay in GPUs, two new parallelism exposing transformations are proposed by Di et. al. [30]. An annotation based CUDA-free interface to implement stencil methods on GPU hardware is introduced by Unat et. al. [73]. Meng et. al. [49] presents a data-flow driven GPU performance projection for multi-kernel transformations. The transformation framework requires users to provide CPU code skeletons for a sequence of parallel loops. The frame-

work can then automatically identify opportunities for multi-kernel transformations and data management. Majeti et. al. [47] presents a compiler-driven data-layout transformation framework for heterogeneous platforms. The data layout framework is integrated with the data parallel construct, `forasync` of Habanero-C, and enable the same source code to be compiled with different data layouts for various architectures. The framework requires the programmer or an auto-tuner to provide a schema of the data layout.

Programming Integrated Parallel System (PIPS) [8], an automatic parallelization system, can be used for source to source program optimization, program compilation, automatic parallelization etc. PIPS accepts programs written in C or Fortran77. It supports analysis techniques such as data flow, control flow, inter procedural analysis and dependence analysis with support for generating code for multiple architectures. Ongoing work in PIPS includes adding support for modern frameworks such as CUDA and OpenCL and inclusion of programming languages like Fortran90/95 and C99. These tools aim at simplifying the task of extracting parallelism for the programmer without compromising the maximum achievable performance gain. These tools work well for affine array access patterns with well understood semantics of popular benchmarked code.

OpenACC extends the familiar face of OpenMP pragma programming to encompass co-processors. It is a set of directive-based extensions to C, C++ and Fortran that allows code with annotations for offloading from a CPU host to an attached accelerator. When using OpenACC, the programmer has to manually annotate the source code with some pragmas that expose parallelism and might steer some data mapping. The PGI compiler then generates CUDA code for the GPU. The performance of the resulting mapping depends both on the quality of the code generated by the tool and the ability of the programmer in setting all the required pragmas. The annotations supported in OpenACC are explicit pragma based annotations. On the other hand, the annotations in JolokiaC++ include both explicit and implicit annotations. The implicit annotations are designed to exploit fine grain parallelism using operation-level data flow analysis. The explicit an-

```

#pragma acc data copyin(A[0:N * N], B[0:N * N]) copyout(C[0:N * N])
{
    #pragma acc region if(accelerate)
    {
        #pragma acc loop independent
        for (int i = 0; i < N; i ++){
            #pragma acc loop independent
            for (int j = 0; j < N ; j ++ ){
                float sum = 0;
                #pragma acc loop seq
                for (int k = 0; k < N ; k ++ ) {
                    sum += A[i * N + k] * B[k * N + j];
                }
                C[i * N + j ] = sum;
            }
        }
    }
}

```

Listing 2.1: Code snippet of Matrix Multiplication using OpenACC

notations in JolokiaC++ work at coarser level using task-level data flow analysis. Code Listing 2.1 and 2.2 shows the code snippet for Matrix multiplication using OpenACC and JolokiaC++ respectively.

CUDA-lite, an annotation based tool to automatically generate CUDA code from given annotated ANSI C code is presented by Ueng et. al. [71]. The annotations developed in the tool are complex and difficult to use. One of main reasons behind this is the fact that compiler analysis is not integrated in the tool. Samadi et. al. [67] proposed an adaptive input aware compilation system, called Adaptic, which automatically generates optimized CUDA code for a wide range of input sizes and dimensions from a high-level algorithmic description. The compiler framework proposed by Yang et. al. [76] optimizes GPGPU programs using a set of novel compiler techniques to improve GPU memory usage and distribute workload in threads and thread blocks. A mathematical model to capture and categorize memory access patterns of affine loops to improve the performance of GPU memory subsystem is presented by Jang et. al. [40]. Oancea et.

```

void matrix_mult(f64Array &A,f64Array &B,f64Array &C){
    f32Scalar pval;
    _for(int i = 0; i < N; i++){
        _for(int j = 0; j < N; j++){
            pval += aview(A, N, i, row) * aview(B, N, j, column);
            C(i,j) = pval;
        }
    }
}

#pragma jolokia gpuIn(A,B,C) gpuOut(C) tile(16,16,1)
                    scatchpad(A,16,16,1) scatchpad(B,16,16,1)
matrix_mult(A,B,C);

```

Listing 2.2: Code snippet of Matrix Multiplication using JolokiaC++

al. [52] proposed a fully automatic approach to loop parallelization using a novel logical inference technique. An annotation guided dynamic compilation is presented by Grant et. al. [34]. A work on annotating user defined abstraction for enforcing traditional compiler optimizations is presented by Quinlan et. al. [55]. Broadway compiler [37] performs high-level semantic encapsulation of PLAPACK library functions. A new programming interface called OpenMPC is proposed by Seyong et. al. [44], to provide an abstraction to the complex CUDA programming model. Further, to overcome some of the inefficiencies of the original OpenMPC tuning scheme a new tuning strategy, called Modified IE (MIE) is proposed by Sabne et. al. [64].

Unlike the existing approaches, we present a novel annotation based compiler framework JolokiaC++, that has the potential to accomplish high performance for existing/new legacy codes to work on the GPU. A JolokiaC++ programmer can exploit GPU parallelism using annotations, without writing complex low level code as required by other mainstream approaches (OpenCL or CUDA). The JolokiaC++ compiler translates a high-level annotated C++ program to efficient low-level CUDA code. This makes GPUs more accessible while effectively exploiting their computational power.

Compile time data dependence analysis of loops accessing subscripted arrays is not

always adequate to exploit hidden parallelism that may manifest itself only at runtime. Traversing data dependences at run-time is necessary for some run-time reordering transformations. Taking this into consideration, a formal composition of runtime data and iteration reordering transformation at compile time to improve cache performance is given by Strout et. al. [69]. Irregularities in code have been widely studied for high performance computing on General-Purpose Graphics Processing Units (GP-GPUs or GPUs for short) [45, 76, 78]. A considerable amount of research to find efficient ways for the compile and run-time support based on the inspector-executor technique is developed by Das et. al. [28]. Researchers have developed run-time data dependence analysis to handle non-affine memory references [54].

Saltz et. al. [65, 27] describe a set of index set transformations that can efficiently solve a form of irregular problems that are start-time schedulable for distributed memory machines. Parallelization of Fortran and C programs with irregular access patterns is supported by the CHAOS [1] runtime library which is specifically targeted towards distributed memory systems that supports message passing or distributed shared memory. Software schemes presented in [50, 60, 62, 59] analyze dependence structure of the code accessing subscripted subscripts at runtime and try to run parts of it in parallel protected by synchronization. Rus et. al. [63] take this further by adding the ability to traverse all data dependences at run-time if necessary. They perform a hybrid (static and dynamic) data dependence analysis inter-procedurally. [48, 31] contributes in improving the memory hierarchy performance for irregular applications using data and computation reordering. A new approach of slice classification for automatic generation of optimized parallel code for N-body simulations is covered in [29]. Other software schemes [36] speculatively run the code in parallel and later recover if a dependence violation is detected.

The massive data parallelism offered by recent architectural enhancements in GPU's comes at the cost of a complex programming model. Researchers have developed several

strategies for overcoming the compilation challenges on the GPU. An auto-parallelization framework for optimization of Affine loop nests on GPGPUs is presented by Baskaran et. al. [17]. The C-to-CUDA by Baskaran et. al. [17], is the first automatic source-to-source compiler based on PLuTo [20]. A preliminary result of an annotation based automated tool for reducing GPU programming complexity is presented by Ueng et. al. [71]. Compiling irregular applications on a cell broadband processor using the inspector-executor approach has been presented by Bhatotia et. al. [18]. Strout et. al. [69, 43] presents the Sparse Polyhedral Framework (SPF) for generating efficient inspector and executor code for multi-core CPUs. The study of an irregular application's implementation on a graphics pipeline is covered in [72]. An extension to StarSs programming model for platforms with multiple GPUs proposed by Ayguade et. al. [14] provides an alternative programming model for exploiting functional parallelism based on building a task dependence graph at run-time with the help of explicit annotations. A comparative study of data-driven and topology-driven implementations of graph algorithms is given by Nasre et. al. [51]. They also devised hybrid approaches that combine both the techniques which outperform each of the two individually.

A speculative parallelization based mechanism to execute iterations of DOACROSS loops on GPU is described by Feng et. al. [33]. In their approach, the misspeculation check is also performed on the GPU. In case of misspeculation, the incorrectly executed iterations are identified and executed on the CPU if there are other misspeculated iterations depending on them; otherwise they are executed again on the GPU. Paragon [66] identifies possibly data parallel loops and runs them speculatively on the GPUs and also runs them sequentially on the CPU. In case of misspeculation, the data generated by the GPUs is ignored and the data generated by the CPU is used. Anantpur et. al. [13] work differs from both these approaches as they do not execute any iterations speculatively on the GPU. In their approach, the loop is executed sequentially on the CPU while the dependence computation is going on, so that in case if the number of levels in the loop is too

high to benefit from running on the GPU, the GPU run can be stopped and the data from the CPU run can be used. Kim et. al. [42] used a profiling based approach to speculatively parallelize loops on a cluster. It tries to optimize the communication and validation overheads. Jablin et. al. [39] presents an inspector executor based approach to handle GPU-CPU communication. However, we use an inspector executor based approach to handle the memory hierarchy of the GPU.

We present a compilation framework for applications with irregular accesses which can simplify general purpose programming of such applications on a GPU. We develop a framework which allows high level specification and control of computational granularity along with a good load balancing mechanism for scheduling and mapping of iterations and data respectively. We used the concept of dynamic scratch pad memory management which is well-known in the context of embedded systems [70] to manage the shared memory of a streaming multiprocessor. Prior work on run-time parallelization on multiple processors includes data and iteration partitioning for irregular applications [61]. In contrast to these, the work presented here extends the compiling support for the GPU in line with the inspector-executor paradigm. We propose the use of an automatic code generator for memory communication and run-time parallelization to deal with irregular kernels on the GPU.



Chapter 3

Language Design

Moore's law states that the number of transistors on a chip will double every 18 to 24 months, and this has been true for the last 40 years. It appears to continue for another decade. But Moore's observation has been simplified to mean a doubling of performance every 18 to 24 months, and that too has been true until now, but not anymore. This resulted in transition to multi and many core processors. As a result, high performance computing will have to make much more effective and efficient use of high degrees of parallelism available through multiple cores and GPU architectures. This requires the development of more sophisticated application code. The complex programming model of GPUs greatly increases the compile-time optimization difficulties for applications. While high level abstractions serve well for parallel programming, they are usually ill suited for the complex programming model offered by GPUs. As a result compilers for such languages provide limited scalability in performance.

In recent years, OpenMP has transitioned from being solely focused on shared-memory systems to include accelerators, embedded systems, multicore and real-time systems. OpenMP 4.0 [5, 6] includes support for accelerators, SIMD constructs to vectorize both serial as well as parallelized loops, error handling, thread affinity, and tasking extensions.

OpenMP 4.0 API provides several extensions to its task-based parallelism support.

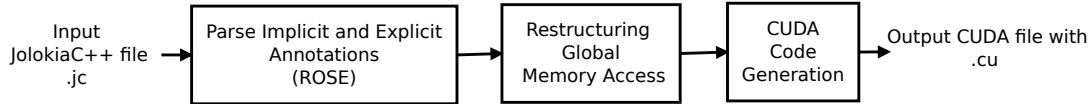


Figure 3.1: Compilation Framework of JolokiaC++

Tasks can be grouped to support deep task synchronization and task groups can be aborted to reflect completion of cooperative tasking activities such as search. Task-to-task synchronization is also supported through the specification of task dependency. Our approach in task-level data flow analysis is very similar to that supported in OpenMP 4.0, however, it is adapted to the context of our language. And similarly, the operation-level data flow graph is also a data flow graph but at the level of operations (statements) inside a task. So, the two graphs are at different granularities.

In this Chapter, we describe the design and implementation of a generic annotation based compiler framework that can be used to program GPGPUs. The programmer is only required to have a clear idea about the parts of programs that will be parallelized, and requires very little knowledge of hardware architecture, and can gauge the performance to an acceptable level on GPUs. Unlike the existing approaches, JolokiaC++ uses annotations for extending C++ to program GPUs. A JolokiaC++ programmer can exploit the parallelism of a GPU using annotations and without writing complex low level code as required by other mainstream approaches like OpenCL or CUDA. JolokiaC++ compiler translates high-level annotated C program to efficient low-level CUDA code. This makes GPUs more accessible while effectively exploiting their computational power.

3.1 Design of JolokiaC++ Framework

JolokiaC++ enhances C++ with an annotation language to enable the user to write GPU programs without worrying about the low level details of the underlying architecture. The high level annotation language allows a user to express data parallelism by specifying operations at the aggregate data collection level. The flow of control in the program for dynamic data types is made explicit at compile time.

Figure 3.1 shows the high level compilation framework implemented for JolokiaC++. The application developer writes a program in JolokiaC++ using explicit annotations specified through pragmas. These annotations provide implementation specific information about the language extensions to the compiler in a simple declarative manner. Further, implicit annotations are added to the language to provide high level abstractions for vector and scalar accesses. The semantic interpretation for these implicit annotations (associated with array objects) allows accessing array elements through primitive array types. We also introduce a way to specify the range of elements of the collection object by means of the *tile* operator. The *tile* operator enables automatic code generation for inter-thread memory access, thereby exploiting parallelism within a kernel by spawning threads/thread blocks.

We use a source-to-source compiler infrastructure, ROSE [57], to explore compiler techniques to recognize high-level abstractions and to exploit their semantics for automatic parallelization. Our compiler framework takes as input annotated C code that is functionally correct, but does not include any lower level device-specific performance optimizations. The framework analyzes operator annotations to determine off-chip memory access patterns, and optimizes these accesses through vectorization and coalescing to achieve high data access bandwidth.

In summary, the process of translation involves interpreting the semantics of the pre-defined operators and data types supported by the annotation language and generating the corresponding code for CUDA enabled GPUs using the hints provided through explicit

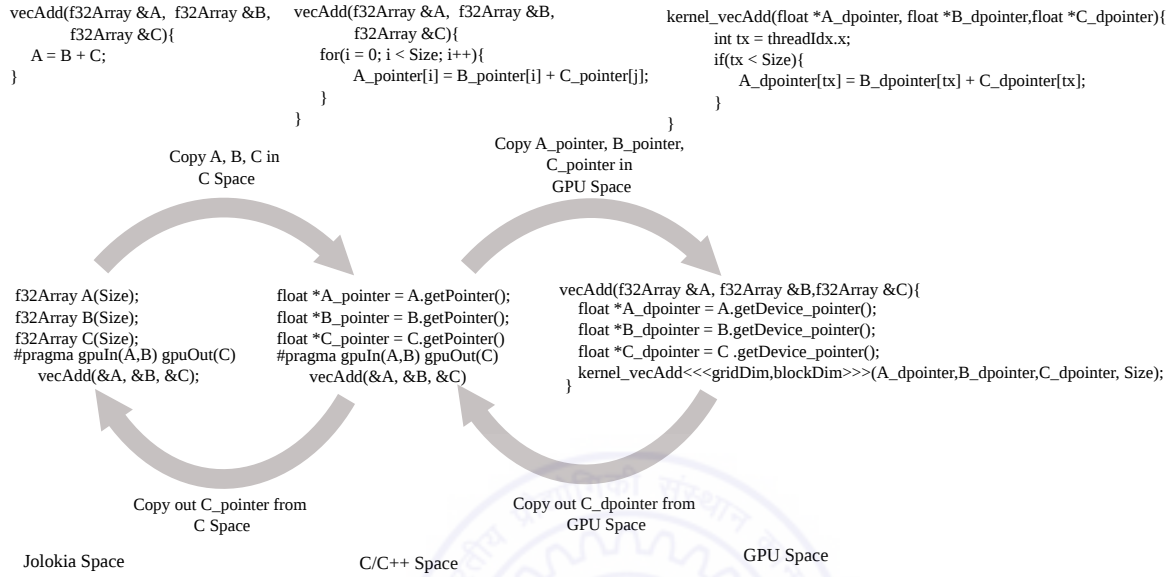


Figure 3.2: Translation spaces in JolokiaC++

annotations.

3.1.1 JolokiaC++ Programming Model

JolokiaC++ source code is intended to be compiled to CUDA, so that the resulting executable code runs on a CUDA enabled GPU. The JolokiaC++ virtual machine provides sufficient abstraction to the programmer to hide the communication details of the programming model. It provides unified view of different address spaces associated with CPU and GPU. This allows the programmer to focus on algorithms rather than dealing with the communication intricacies involved in using different memory spaces required for the execution of program on GPU.

To support this model JolokiaC++ introduces three spaces: 1) Jolokia space 2) C space and 3) Device space. These are depicted in Figure 3.2. Different views for scalar array

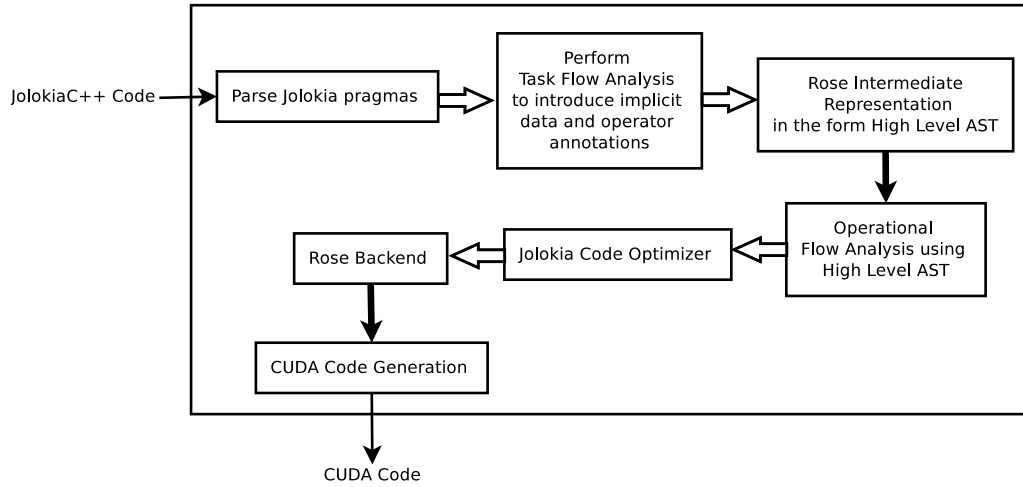


Figure 3.3: Translation of JolokiaC++ code

objects are created in different spaces as per the aliasing information provided through annotations. The use of a simple scalar object corresponds to the creation of a thread local variable in the kernel, resulting in use of streaming multiprocessors' registers. Pragmas are preprocessed to generate the annotations required for creating an alias on the device. The process of translation is shown in Figure 3.3.

3.1.2 JolokiaC++ Annotations

Explicit annotations exist in the input program in the form of data annotations, operator annotations or as pragma specifiers. Implicit annotations are implemented to simplify the translation of explicit annotations. To recognize annotations, we build an annotation grammar and the corresponding high-level ASTs using ROSE. This grammar is similar to the base language grammar. We added constructs to represent specific user-defined functions, data-structures, user-defined types etc in the grammar. The compiler recognizes the annotations within an application in much the same way it recognizes the syntax of the base language.

```
void SAXPY(f64Scalar &A, f64Array &X, f64Array &Y, f64Array &S){  
  S1: S = A * X + Y;  
}  
  
main(){  
  f64Array X(N);  
  f64Array Y(N);  
  f64Array S(N);  
  f64Scalar A;  
  
  ...  
  
  /* Task call site */  
  #pragma jolokia gpuIn(X,Y,S) gpuOut(S) tile(BLK,1,1)  
    SAXPY(A, X, Y, S);          /* GPU Task */  
}
```

Listing 3.1: Data Annotation Example

```
void SAXPY(f64Scalar &A, f64Array &X, f64Array &Y, f64Array &S){  
  int xlen = X.length;  
  double *s = S.get_pointer();  
  double *x = X.get_pointer();  
  double *y = Y.get_pointer();  
  double a = A.get_data();  
  
  ...  
  
  for (i = 0; i < xlen; i++){  
    s[i] = a * x[i] + y[i];  
  }  
}
```

Listing 3.2: Semantic Interpretation of Statement S1 in Listing 3.1

3.1.2.1 Data Type annotations

Explicit data annotations like `f32Array`, `f64Array`, `f32Scalar` provide data abstraction for vectors and scalars of primitive type. The implicit annotations `array_view` and `is_array` are associated with data annotations. We associate attributes like *dimension*, *length* and *element* using implicit annotation `array_view` to create a view of an array in C space. The scalar variable *dimension* holds the dimension of an array. The attributes *length* and *element* are collection variables. The attribute *length* is used to hold the length of an array in each dimension. Every occurrence of `f32Array` object is replaced by an *element* which is indexed by integer parameters. This annotation replaces object reference in Jolokia space to arrays in C by making use of the view created in C space. The `is_scalar` annotation is used to convey that `f32Scalar` has C scalar semantics with *selem* attribute. The attribute *selem* is used to hold the value of scalar variable. The annotations for scalar data types is created in order to provide flexibility in describing operator annotations.

Example 3.1: To exemplify the role of explicit data annotations, consider a JolokiaC++ code snippet given in Listing 1. Here **X**, **Y**, and **S** are declared as `f64Array` type variables and **A** is declared as `f64Scalar` type variable. Semantically, each array object declared in this code is interpreted as a single *dimensional* array of *length* N which can store N 64-bit floating point *elements*. Based on the number of parameters passed to an array object, the dimensions and the lengths corresponding to these dimensions are determined for each dynamically created array of appropriate type (`double` for this example). For the scalar object **S** in the code snippet, a 64-bit floating point (`double`) scalar variable is created.

To illustrate the significance of introducing scalar types, let us consider the statement S1 in Listing 1. Here, **a** is the scalar element (*selem*) associated with **A**. Also, **x**, **y** and **s** are the dynamic arrays associated with **X**, **Y** and **S** respectively. The semantic interpretation of statement S1 is as shown in Listing 3.2. In order to interpret operations applied between scalar and vector types (e.g * operation between **A** and **X**) we introduce data type annotations for scalar variables. This allows us to carry out various operation

Annotation	Semantics
dalloc	Allocates memory on GPU space for the object passed as parameter to the annotation.
dcpy	Annotation to copy data from CPU space to GPU space for the object passed as parameter to it.
on_entry	Returns a device pointer associated with object passed as parameter to the annotation.
on_exit	Annotation to copy data from CPU space to GPU space for the object passed as parameter to it.

Table 3.1: CPU-GPU Memory Space Management Annotations

between scalar and array type variables. ■

3.1.2.2 Operator annotations

Operator annotations describe the semantics of the functions that operate on the data abstractions used for creating scalar and vector data types. For every function call that follows a Jolokia pragma, a function operator entry is created in an annotation file. The *on_entry* and *on_exit* annotations are placed within the function operator for each object instance passed to the *gpuIn* and *gpuOut* parameter of a Jolokia pragma. The *gpuIn* pragma specification provides information regarding the array object instance being read by the kernel function. Similarly, *gpuOut* is used for providing information regarding the writing of data into the array object instance. Using this information, we build a graph based on the operations performed on objects of array type. This graph is similar to an abstract syntax tree in a compiler. The graph provides sufficient information to optimize the intermediate representation of the code for the target architecture.

The *scratchpad* annotation is provided to specify the size of shared memory and to restructure memory access for coalesced global memory access. The *aview* operator annotation provides linearized access to multi-dimensional arrays for vectorization. The *stride* and *shift* operators allow the programmer to specify different memory access patterns. The *modify_array* annotation is implemented to allow modifications to an alias

```

#define BLK 256
/* Task definition */
void add(f64Array &A, f64Array &B, f64Array &C){
    A = B + C
}

main(){
    f64Array A(N);
    f64Array B(N);
    f64Array C(N);

    ...

    /* Task call site */
    #pragma jolokia gpuIn(A,B,C) gpuOut(A) tile(BLK,1,1)
        add(A,B,C);          /* GPU Task */
}

```

Listing 3.3: Example for *gpuIn*, *gpuOut* and *Tile* annotation

created in C space. The *alias* annotation describes aliasing relationships between the inputs and results. The *on_entry* annotation allows retrieval of an object's view in GPU space. The *on_exit* annotation is concerned with reflecting the changes made in the GPU space to an array created in C space. The *kernel* annotation is provided for composition of GPU kernels in the code. The *tile* annotation provides sufficient information required to launch a kernel on the GPU. The implicit CPU-GPU space mapping annotations and their semantics is presented in Table 3.1.

Example 3.2: In the example shown in Listing 3.3, we illustrate the use of *gpuIn* and *gpuOut* annotations associated with *jolokia* pragma. Each array type object can either have associated dynamic arrays, one on device (GPU) and one on host (CPU) each or it can have an associated dynamic array only on host. The dynamic array associated with objects **A**, **B** and **C** are required for invoking the device kernel associated with *add* function. This requires insertion of *on_entry* annotation for each array object within the

```
void stencil(f32Array &A, f32Array &B){
    B = shift(A,-1) + shift(A,0) + shift(A,1)
}
```

Listing 3.4: Shift Operator Example

```
void stencil(f32Array &A, f32Array &B){
    for(i = 0; i < len; i++){
        B[i] = A[i-1] + A[i] + A[i+1];
    }
}
```

Listing 3.5: Semantic Interpretation of Shift Operator

function annotation. This would allow us to obtain device pointer associated with the every dynamic array existing on CPU. The result of vector addition stored in the device array associated with **A** is returned back to the CPU by passing it as a parameter to *gpuOut*. ■

Example 3.3: Listing 3.3 also demonstrates the use of *tile* annotation associated with jolokia pragma. It is applied to define parameters (grid and block dimension) required to launch the kernel on the GPU. ■

Example 3.4: Listing 3.4 illustrates the use of shift operator. Its semantic interpretation on CPU space is shown in Listing 3.5. ■

3.1.3 Task-Level Data Flow Graph

Before describing the task-level data flow graph we present an example to motivate its need. Listing 3.6 presents a snippet of JolokiaC++ implementation of data parallel code. A naive translation that does not take into account the flow of data among the statements executed on CPU and on GPU is presented in Listing 3.7. In this code the unnecessary *cudamalloc* and *cudamemcpy* calls are generated which could be inefficient or incorrect. Further, what we would like to generate the code similar to what a CUDA expert would

write. A sample of such a code is shown in Listing 3.8. To do so, we need to analyse the flow of data in the JolokiaC++ program. This is done using the task-level data flow graph as defined next, after a few helper definitions.

Definition 3.1. (GPU Task) The Jolokia pragma along with the function call that immediately follows it is defined as a GPU task.

Definition 3.2. (CPU Task) Maximal set of consecutive statements which do not contain any jolokia pragma is a CPU task. This set of statements are targeted for execution on the CPU.

Definition 3.3. (Task-Level Data Flow Graph) Let \mathcal{P} be a JolokiaC++ program. The nodes in a task-level data flow graph of \mathcal{P} represents CPU or GPU tasks, and the edges represent flow of data from one node to another. An edge from node n_1 to node n_2 exists in the task-level data flow graph of \mathcal{P} , if n_1 follows n_2 in program order. Note that, even though a GPU task is a function call, it represents a single node in the task-level data flow graph.

Example 3.5: The task-level data flow graph for the JolokiaC++ code given in Listing 3.6 is shown in Figure 3.4. Here, *add pragma* and *scale pragma* nodes correspond to *add* and *scale* functions that immediately follow a jolokia pragma. A node for the CPU task *print* also exists in the graph. To analyse the body of the function corresponding to a GPU task we use an operation-level data flow graph as described in the next section. ■

3.1.4 Operation-Level Data Flow Graph

The operation-level data flow graph for a GPU task function \mathcal{F} is a graph where nodes correspond to the statements of \mathcal{F} , and the edges represent flow of data from one node to other. We use operation-level data flow graph to analyze the operations performed on

```

/* Task definition */
void add(f64Array &A, f64Array &B, f64Array &C){
    A = B + C
}

void scale(f64Array &A, f64Scalar &S){
    A = S * A;
}

main(){
    ...
    /* Task call site */
    #pragma jolokia gpuIn(A,B,C) gpuOut(A) tile(BLK1,1,1)
        add(A,B,C);          /* GPU Task */
    print(A);                 /* CPU Task */
    #pragma jolokia gpuIn(A,S) gpuOut(A) tile(BLK2,1,1)
        scale(A,S);          /* GPU Task */
}

```

Listing 3.6: Task-Level Data Flow Example

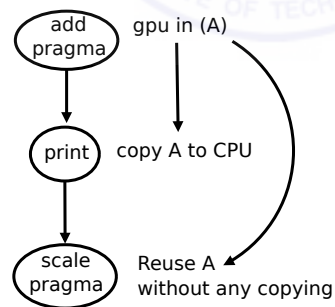


Figure 3.4: Task-Level Data Flow Graph

```

__global__ void add(float *A, float *B, float *C)
int idx = threadIdx.x + blockIdx.x * blockDim.x;
if(idx < N){
    A[idx] = B[idx] + C[idx];
}
}

__global__ void scale(float *A, float S)
int idx = threadIdx.x + blockIdx.x * blockDim.x;
if(idx < N){
    A[idx] = S * A[idx];
}
}

cudaMalloc((void **)&Device_A, N*sizeof(float));
cudaMalloc((void **)&Device_B, N*sizeof(float));
cudaMalloc((void **)&Device_C, N*sizeof(float));
//Copy Host array to Device array
cudaMemcpy(Device_A, Host_A, N*sizeof(int),
            cudaMemcpyHostToDevice));
cudaMemcpy(Device_B, Host_B, N*sizeof(int),
            cudaMemcpyHostToDevice));
cudaMemcpy(Device_C, Host_C, N*sizeof(int),
            cudaMemcpyHostToDevice));

//Make a call to GPU kernel
add<<<gridDim, blockDim>>>(Device_A, Device_B, Device_C);
//Copy back to Host array from Device array
cudaMemcpy(Host_A, Device_A, N*sizeof(float),
            cudaMemcpyDeviceToHost);
for(i = 0; i < N; i++)
    printf("%f",Host_A[i]);
cudaMalloc((void **)&Device_A, N*sizeof(float));
cudaMemcpy(Device_A, Host_A, N*sizeof(int),
            cudaMemcpyHostToDevice));
//Make a call to GPU kernel
scale<<< gridDim, blockDim>>>(Device_A, host_S);
//Copy back to Host array from Device array
cudaMemcpy(Host_A, Device_A, N*sizeof(float),
            cudaMemcpyDeviceToHost);
//Free the Device array memory
cudaFree(Device_A);
cudaFree(Device_B);
cudaFree(Device_C);

```

Listing 3.7: A Naive Translation of Task-Level Data Flow Example

```

__global__ void add(float *A, float *B, float *C)
int idx = threadIdx.x + blockIdx.x * blockDim.x;
if(idx < N){
    A[idx] = B[idx] + C[idx];
}
}

__global__ void scale(float *A, float S)
int idx = threadIdx.x + blockIdx.x * blockDim.x;
if(idx < N){
    A[idx] = S * A[idx];
}
}

cudaMalloc((void **)&Device_A, N*sizeof(float));
cudaMalloc((void **)&Device_B, N*sizeof(float));
cudaMalloc((void **)&Device_C, N*sizeof(float));
//Copy Host array to Device array
cudaMemcpy(Device_A, Host_A, N*sizeof(int),
            cudaMemcpyHostToDevice));
cudaMemcpy(Device_B, Host_B, N*sizeof(int),
            cudaMemcpyHostToDevice));
cudaMemcpy(Device_C, Host_C, N*sizeof(int),
            cudaMemcpyHostToDevice));

//Make a call to GPU kernel
add<<<gridDim, blockDim>>>(Device_A, Device_B, Device_C);
//Copy back to Host array from Device array
cudaMemcpy(Host_A, Device_A, N*sizeof(float),
            cudaMemcpyDeviceToHost);
for(i = 0; i < N; i++)
    printf("%f", Host_A[i]);

//Make a call to GPU kernel
scale<<< gridDim, blockDim>>>(Device_A, host_S);
//Copy back to Host array from Device array
cudaMemcpy(Host_A, Device_A, N*sizeof(float),
            cudaMemcpyDeviceToHost);
//Free the Device array memory
cudaFree(Device_A);
cudaFree(Device_B);
cudaFree(Device_C);

```

Listing 3.8: Translation after Task-Level Data Flow Analysis

objects of scalar and vector type. The analysis involves identifying the sequence of statements that are called when instantiation of threads happen within a **subtask**. A subtask corresponds to the subset of tasks targeted for execution on a streaming multiprocessor of a GPU. It represents one iteration of the innermost implicit `for` loop indicated through operator annotation. This operation flow analysis helps us to exploit data parallelism for threads within a subtask.



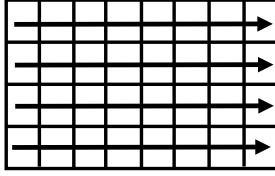


Chapter 4

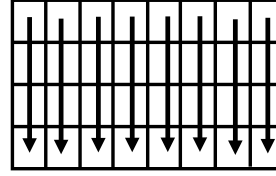
Regular Applications

Memory accesses which are linear combinations of loop index variables are quite common in a variety of applications like dense-matrix linear algebra, finite-difference PDE solvers, image processing and scans/joins in relational databases. Array references with such access are called affine array accesses. Loops with affine array accesses are a natural consequence of the emergence of massively multi-threaded data-parallel computing platforms. Today's GPU computing platforms are designed with a heterogeneous memory architecture comprising multiple memory spaces where each space has specific characteristics. Mapping data arrays to the most appropriate memory space, based on their associated memory access patterns, can have a huge impact on overall performance.

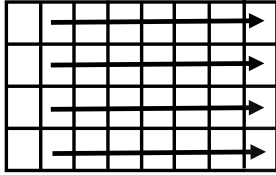
We present a technique that uses operator annotations to systematically characterize affine access in loop nests. We present a methodology that optimizes memory performance of data-parallel architectures. The goal is to convey implementation-specific information to the compiler in a simple declarative manner through the hints provided by annotations, which helps to improve the performance of the given code. We evaluate the performance of our framework by considering BLAS and stencil codes.



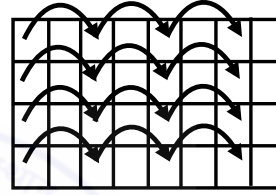
$\text{aview}(A, \text{width}, i, \text{row})$ or
 $\text{aview}(A, \text{width}, i) = A[i * \text{width} + j]$
 (a) Row major access of elements



$\text{aview}(A, \text{height}, i, \text{column}) = A[j * \text{height} + i]$
 (b) Column major access of elements



$\text{shift}(A, 0, C) = A[i * \text{width} + (j + C)]$
 (c) Row major shifted access of elements



$\text{stride}(A, 1, C) = A[i * \text{width} + (C * j)]$
 (d) Strided access of elements

Figure 4.1: Memory Access Patterns and their Respective Operator Annotations with their Linearized Representation.

4.1 Memory Access Operators

In this section, we present operator annotations that capture the memory access pattern present in a loop nest. The operator annotations *aview*, *stride* and *shift* are meant to provide the necessary information to represent affine memory access patterns. This in turn is used for optimizations presented in Section 4.2.2. Figure 4.1 presents examples of memory access patterns along with the associated operator annotations.

Operator *aview*

The row linear and column linear memory access patterns refer to accesses in which a dimension of an array is contiguously accessed with respect to iteration space as shown in Fig. 4.1a and Fig. 4.1b respectively. The *aview* operator provides composition for row/-column linear memory access patterns. It provides a linearized view of array elements associated with the object and has the following four parameters:

1. The collection object.
2. The limiting condition for the implicit loop iterator.
3. Iterator associated with the direction of access.
4. The direction of access.

The signature of the *aview* operator is as follows:

```
aview(DataType objectName, int numberOfColumns,  
      int explicitIterator, int accessDirection)
```

Operator *shift*

A shifted memory access pattern refers to an access in which a dimension of an array is contiguously accessed with respect to the iteration space, but this contiguous access is shifted by some constant. The use of the *shift* operator and its access pattern is shown in Fig. 4.1c. The first parameter corresponds to the collection object. The second and third parameters are used to provide the amount of shift in the row and column directions respectively. The signature of *shift* operator is as follows:

```
shift(DataType objectName, int rowShift, int colShift)
```

Operator *stride*

The use of the *stride* operator allows non-unit stride memory access. Fig. 4.1d illustrates its use. The first parameter corresponds to the collection object. The second and third parameters are used to provide the stride value for specific directions. The signature of the *stride* operator is as follows:

```
stride(DataType objectName, int rowStride, int colStride)
```

Example 4.1: The operators *shift* and *stride* can be combined to generate different access patterns. Listing 4.1 demonstrates some interesting combinations of these operators. ■

Example 4.2: To describe our programming model, we start with the matrix multiplication code example provided in Listing 4.2. The framework parallelizes a loop nest by associating one logical thread with some number of points in the iteration space of the nest. It then partitions and maps the logical threads onto physical ones, guided by annotations that the programmer employs to tune the code. The annotated code, using the *aview* operator and the data parallel `_for` loop, is shown in Listing 4.3. The data parallel `_for` loops (*i* and *j*) in the code get mapped to threads when parallelized. When used in conjunction with the *tile* operator, these loops correspond to a 2D thread configuration. ■

Example 4.3: The transformation framework provides linearized access for each *aview* operator as shown in Listing 4.4. Using *aview* and *tile* results in inter-thread and intra-thread access of elements in block $((i - tx)/BLOCK_SIZE_X, (j - ty)/BLOCK_SIZE_Y)$ by thread (tx, ty) for array **A**, **B** and **C** as shown in Figure 4.2. The high level semantic interpretation of column linear memory access operation (memory access of **B** in Listing 4.3) results in coalesced global memory access by threads in a warp. The use of the data parallel `_for` loop iterators (*i* and *j*) for accessing the elements of **C** indicates each thread accessing one element of the matrix. The row linear memory access of **A** leads to uncoalesced global memory access by threads in the warp. Using these interpretations our code generator generates naive CUDA code for GPU. ■

```
shift(stride(A,2,3),0,-4) = A[2*i][3*j-4] = A[(2*i)*width+(3*j-4)]
```

Listing 4.1: Combination of Shift and Stride operator annotations

```
for(int i = 0; i < M; i++)
  for(int j = 0; j < N; j++)
    for(int k = 0; j < N; j++)
      C[i][j] += A[i][k] * B[k][j];
```

Listing 4.2: Serial matrix multiplication code

```
void matrix_mult(f64Array &A,f64Array &B,f64Array &C){
  f32Scalar pval;
  _for(int i = 0; i < M; i++)
    _for(int j = 0; j < N; j++){
      pval += aview(A, N, i, row) * aview(B, P, j, column);
      C(i,j) = pval;
    }
}
#pragma jolokia gpuIn(A,B,C) gpuOut(C) tile(16,16,1)
                      scratchpad(A,16,16,1) scratchpad(B,16,16,1)
matrix_mult(A,B,C);
```

Listing 4.3: Annotated Matrix-Multiplication Code

```
aview(A,N,i,row) = A[i * N + k]
aview(B,P,j,column) = B[k * P + j]
C(i,j) = C[i * M + j]
```

Listing 4.4: Linearized access of annotations for A, B, and C shown in Listing 4.3

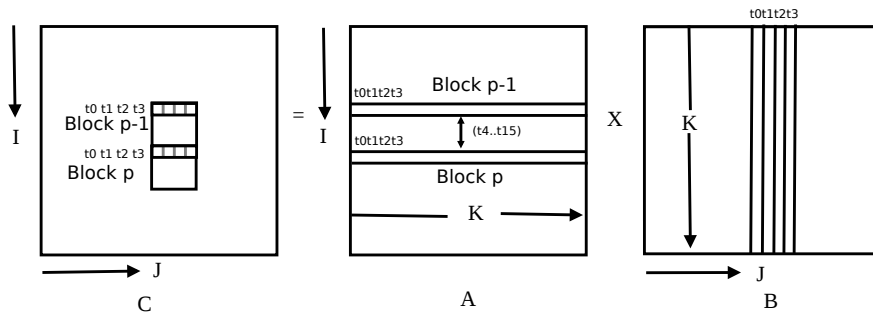


Figure 4.2: Inter and Intra-thread access of elements in absence of scratchpad annotation

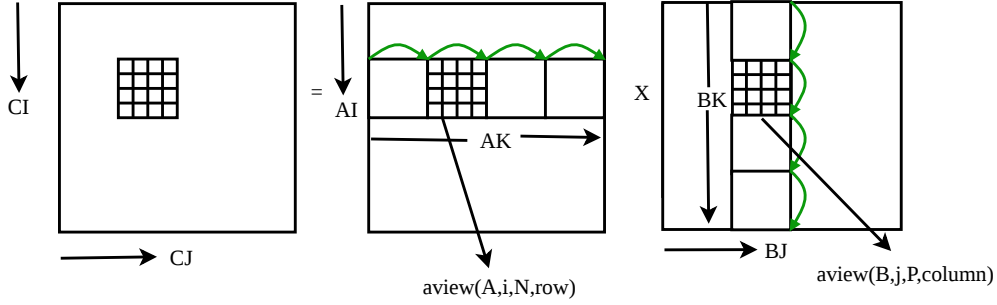


Figure 4.3: Inter and Intra-thread access of elements in presence of scratchpad annotation

Example 4.4: The provision of *scratchpad* along with *aview* and *tile* introduces intra-thread memory access composition for shared memory access. The inter and intra-thread access of elements in the presence of *scratchpad* annotations is shown in Figure 4.3. The use of *scratchpad* annotation allows coalesced global memory access of **A**, **B** and **C** by threads in the warp. The high level semantic interpretation of row and column linear memory access operations in the presence of *scratchpad* annotations lead to coalesced global memory access by threads in the block in their respective direction. ■

4.2 Compiling high level JolokiaC++ constructs

In this section, we discuss the translation performed by JolokiaC++ guided by implicit and explicit annotations. The compiler performs task-level data flow analysis to optimize the communication between CPU and GPU. This is discussed in subsection 4.2.1. The operation-level data flow analysis, discussed in subsection 4.2.2, allows composition of the memory hierarchy optimization in the kernel functions. The operation-level data flow analysis used in the transformation framework is based on the work presented by Vegdoolaege et. al. [74].

Algorithm 4.1: Determining Block and Grid Dimension for linearized array

Input: num_of_elem, array_data_type**Output:** BLOCK_SIZE, GRID_DIM

Retrieve SHARED_MEM_SIZE, MAX_NUM_THREAD at runtime

if \exists *tile operator annotation* **then**| BLOCK_SIZE \leftarrow tile(BLOCK_X,...)**else**| **if** \exists *Scratchpad annotation with stride parameter* **then**| BLOCK_SIZE \leftarrow min(MAX_NUM_THREAD, num_of_elem, Scratchpad
| Parameter) SHARED_SIZE \leftarrow scratchpad(Obj,SBLOCK,...)| **else**| **if** SHARED_MEM_SIZE/sizeof(array_data_type) > MAX_NUM_THREAD)| **then**| BLOCK_SIZE \leftarrow min(DEFAULT_SIZE(), MAX_NUM_THREAD,
| num_of_elem)| **else**| BLOCK_SIZE \leftarrow min(DEFAULT_SIZE(),
| SHARED_MEM_SIZE/sizeof(array_data_type), num_of_elem)GRID_DIM \leftarrow (num_of_elem + BLOCK_SIZE - 1)/BLOCK_SIZE**return;**

```

float sum = 0;
for(int j = 0; j < width; j++)
    sum += A[i][j] * X[j];
X[i] = sum;
}

```

Listing 4.5: Matrix-Vector Multiplication using C

```

void stencil(float A[][], float B[][], int N){
    int i, j;
    for(int i = 0; i < N; i++){
        for(int j = 0; j < N; j++){
            B[i][j] = A[i-1][j] + A[i+1][j] + A[i][j]
                    + A[i][j-1] + A[i][j+1];
        }
    }
}

void copy(float A[][], float B[][], int N){
    for(int i = 0; i < N; i++){
        for(int j = 0; j < N; j++){
            A[i][j] = B[i][j];
        }
    }
}

// Function call site
for(t = 0; t < iter; t++){
    stencil(A,B,N);
    copy(A,B,N);
}

```

Listing 4.6: Stencil code using C

Example 4.5: To illustrate the significance of both analyses we start with the Matrix-Vector multiplication and Stencil code written in C shown in Listings 4.5 and 4.6 respectively. The corresponding *JolokiaC++* code is shown in Listings 4.7 and 4.8. Note that both codes do not have low level details of target architecture. The code written in *JolokiaC++* is similar to C code except for one or more pragmas. The compiler analyzes the annotated function and identifies the parameters for kernel invocation by the process given in Algorithm 4.1. It also checks the off-chip memory access patterns, and optimizes memory accesses through coalescing and shared memory to achieve high data ac-

```

_for(int i = 0; i < height; i++){
    f32Scalar sum = 0;
    sum = sum + aview(A,i,width) * X;
    X(i) = sum;
}

```

Listing 4.7: Matrix-Vector Multiplication using JolokiaC++

```

// Function definition
void stencil(f32Array &A, f32Array &B, int N){
    int i, j;
    _for(int i=0; i< N; i++){
        _for(int j=0; j < N; j++){
            B(i,j) = shift(A,-1,0) + shift(A,1,0) + shift(A,0,0)
                    + shift(A,0,-1) + shift(A,0,1);
        }
    }
}
void copy(f32Array &A, f32Array &B){
    A = B;
}
// Function call site
for(t = 0; t < iter; t++){
    #pragma jolokia gpuIn(A,B) gpuOut(B) tile(32,8,1)
        stencil(A,B,N);
    #pragma jolokia gpuIn(A,B) gpuOut(B) tile(32,8,1)
        copy(A,B);
}

```

Listing 4.8: Stencil code using JolokiaC++ Shift Operator Annotation

cess bandwidth. The parameter required for kernel invocation is generated by the process described in Algorithm 4.1 at runtime. The input required for this process is provided through explicit annotations specified by the pragmas. The vectorized access patterns specified by the *aview* operators provide necessary information for optimizing the GPUs memory hierarchy. ■

Note that both the program (Listing 4.7 and Listing 4.8) do not have low level details of target architecture. The code written in JolokiaC++ is similar to C code except for an extra pragma. The compiler analyzes the pragma annotated function and identifies the

parameters for kernel invocation by the process given in Algorithm 4.1. It also checks the off-chip memory access patterns, and optimizes memory access through coalescing and shared memory to achieve high data access bandwidth. The parameter required for kernel invocation is generated by the process described in Algorithm 4.1 at runtime. The input required for this process is provided by explicit annotations specified by the pragmas. The vectorized access patterns specified by *aview* operators provide the necessary information for optimizing the GPU's memory hierarchy.

4.2.1 Task-Level Data Flow Analysis for Optimizing Communication

The communication between CPU-GPU is a common source of errors for manual parallelization of code and limits the applicability of automatic parallelization. To add to this, cyclic communication dramatically increases the execution time of a program by an order of magnitude. It prevents the system from efficiently parallelizing programs that launch many GPU functions. A global task-level data flow analysis is performed on the task-level data flow graph to identify cyclic communication patterns in the input code. The analysis of the task-level data flow graph also helps in preserving the semantics of the data flow between tasks. It takes off with a sequential JolokiaC++ code calling parallel GPU codes without any CPU-GPU communication. All variables share a single common Jolokia namespace with no distinction between GPU and CPU memory spaces. For each function that immediately follows Jolokia pragma, the compiler creates a list of live-in array objects. An array object is live-in if it is passed to the *gpuIn* annotation directly or is a global variable used by the GPU. It makes use of in-built def-use analysis and liveness analysis to compose the decision for allocation and transfer of data between the CPU and GPU. The basic steps performed in task-level data flow analysis are presented in Algorithm 4.2. It uses the task-level data flow variables determined by task-level data flow variable analysis presented in Algorithm 4.3. For each array object in the *ALLARRAY* set, an implicit annotation *dalloc* for allocation of memory on the device is inserted before the

first Jolokia pragma. An *on_entry* annotation is inserted for each parameter of the *gpuIn* annotation before the *kernel* annotation within the function annotation. The *on_entry* annotation allows access to the device memory by returning a pointer to the device memory location. The entries *dalloc*, *dcopy* and *on_exit* for allocation and data transfer are created through task-level data flow analysis. The compiler constructs a task-level data flow graph (TFG) and uses the task-level data flow variables (TFVs) information gathered through the process presented in Algorithm 4.3 to identify the communication pattern for optimization. Further, a postorder traversal on TFG is performed to identify the program points which can hoist the *dalloc* and *dcopy* out of the loop bodies and up in the task graph. If the Jolokia pragma exists within a loop the compiler promotes the allocation and copy operation for the variables that exist in the COPYIN set by placing the *dalloc* and *dcopy* above the loop. An implicit *on_exit* annotation is inserted after the loop for each object passed as parameter to the *gpuOut* annotation. Further, the memory allocated to all the arrays associated with array objects in the ALLARRAY set is released through the *release* operator.

4.2.2 Operation-Level Data Flow Analysis for Optimizing Memory Access

In this section, we discuss operation-level data flow analysis to enable efficient use of the memory hierarchy. The compiler, after performing task-level data flow analysis, resorts to operation-level flow analysis for mapping non-scalar data to different memory locations in the device. The use of operator annotations within a loop allows the framework to extract information about regional array accesses. The operation-level data flow analysis captures frequent usage of an array by keeping track of its reads and writes, thus exhibiting temporal locality. This concept is based on the work presented in [74]. It scans the immediate representation from memory access operators and applies the corresponding transformation using a pattern matching approach. The use of the *aview* operator in a data

Algorithm 4.2: Global Task-Level Data Flow Analysis

Input: Global program information of AO (Array Objects), set of JPP (Jolokia pragma points) τ

Output: Task-Level Data flow variables(TFVs)

Perform postorder traversal of TFG to insert $dmalloc \forall ALLARRAY$

Insert $dcopy \forall GIN(1) \cup GOUT(1)$

$ALLMODEF(1) = \phi$

for $t = 2 \rightarrow \tau$ **do**

if GPU task $p \in Pred(t)$ **then**

COPYOUT(t) : The GPU array objects which need to be copied on CPU

$COPYOUT(t) = GOUT(p) \cap USE(t)$

ALLMODEF(t) : The array objects modified on or before CPU task t

$ALLMODEF(t) = ALLMODEF(t-1) \cup MODEF(t)$

if GPU task $s \in Succ(t)$ **then**

COPYIN(s) : The GPU array objects which need to be copied on GPU

$COPYIN(t) = GIN(s) \cap ALLMODEF(t)$

$ALLMODEF(t) = ALLMODEF(t) \setminus COPYIN(t)$

return

Algorithm 4.3: Task-Level Data Flow Variables Analysis

Input: Set of Task-Level Data Flow Graph(TFG) CPU nodes ω , Set of Task-Level Data Flow Graph(TFG) GPU nodes τ , Global program information of Array Objects(AO)

Output: Local Flow Variables (LFVs)

$ALLARRAY = \{\phi\}$

for $t = 1 \rightarrow \tau$ **do**

GIN(t) : The array objects passed as parameters to $gpuIn$ in t .

GOUT(t) : The array objects passed as parameters to $gpuOut$ in t .

ALLARRAY : All array objects that exist on GPU memory

$ALLARRAY = ALLARRAY \cup GIN(t)$

for $k = 1 \rightarrow \omega$ **do**

MODEF(k) : The array objects modified/defined in or before k

USE(k) : The array objects used in k

return

parallel `_for` loop is interpreted as inter-thread memory access of an array within a loop indicating the reuse of an array. The *aview* operator when used in conjunction with the *scratchpad* annotation results creation of subarray being read in the shared memory. The

optimizer performs a *def-use* analysis to retrieve information about when to transfer data between on-chip and device memory.

The operator annotations in the source code aid the compiler front end to generate the equivalent Abstract Syntax Tree (AST) fragment to be substituted into the application's AST. This allows test codes containing operator annotations for the transformations to be built separately, which in turn helps us to introduce optimizing transformations into the applications. This involves performing operation-level data flow analysis on the transformations represented through operator annotations for extracting reuse information from array accesses. The operation-level data flow analysis makes use of in-built *data dependence* analysis, *def-use* analysis and *liveness* analysis of ROSE to decide whether it is safe for a loop nest to parallelize or not. The data dependence analysis in ROSE implements the transitive dependence analysis algorithm published by Yi, Adve and Kennedy [77]. We discuss the data structure used for operation flow analysis in the next section.

4.2.2.1 Operation-Level Data Flow Tree (OFT)

In this section, we introduce the basic terms and the data structure that play a key role in analyzing and transforming kernel codes. The annotation based code is parsed into a high level intermediate representation, called the Operation-Level Data Flow Tree (OFT). Each node of the OFT corresponds to a perfectly nested loop (which are loop nests in which all assignment statements are contained in the innermost loop) statement in the code. The innermost loops represent the leaf nodes. An imperfectly nested outer and inner loop become parent and child nodes respectively. Each data parallel node (Explicit loop or EL) records the loop iterators and their ranges using the information provided through the *tile* operator or through default parameters. It also records the loop body as a sequence of code statements using linearize access for the array portions. Compared to the AST in compilers whose leaf nodes are individual operators and operands, the OFT representation works at a much higher level. OFT encapsulates knowledge about

Algorithm 4.4: Memory Access Optimization

Input: Local program information of Array Objects(*AO*), Memory Access Pattern(*MAP*), Read and Write access of *AO*

Output: Memory Location(*ML*)

if \exists *scratchpad* annotation **then**

- Let **R** be the set of **AO** passed as parameters to *scratchpad* annotations
- Let **T** denote the program points to insert shared memory load and store
- if** *The loop is completely nested with single IL* **then**
 - Load data from global memory to shared memory before the first access of any $AO \in R$
 - Store data from shared memory to global memory at or after the last write in the sequence
- else**
 - Perform postorder traversal of OFT and determine **T** using inter-loop data dependences

else

- Let **G** be the group of *AO*'s at the leaf nodes
- Enumerate the *MAP* of the leaf node to determine the average reuse of *AO*
- if** \exists *shift operator with average reuse greater than 1 and read only access* **then**
 - Use texture fetch for each access of **AO**
- else**
 - Use global memory access

return

flexibilities in the code structure and performance related behaviors through high level operator annotations. Algorithm 4.4 presents the process to identify the different memory access points used for optimizing the memory accesses.

To illustrate its applicability, let us reconsider the C code of Matrix-Vector multiplication shown in Listing 4.5. The corresponding *JolokiaC++* code shown in Listing 4.7. It shows the use of the *aview* operator to generate vectorized code for a GPU. The code generated by *JolokiaC++* code generator, without *scratchpad* annotations, is shown in Listing 4.9. In this code, the vectorization is across multiple rows, which would result in each thread processing one row at a time. This is pictorially shown in Figure 4.4 where all the array accesses are from global memory. However, we note that instead of each thread processing a single row, vectorizing within a single row will improve the performance


```

int i = blockIdx.x * BLOCK_SIZE + threadIdx.x;
float sum = 0;
if(i < height){
    for (j = 0; j < width; j++){
        sum = sum + A[i * width + j] * X[j];
    }
    Y[i] = sum;
}

```

Listing 4.9: Naive Matrix-Vector Multiplication CUDA Code

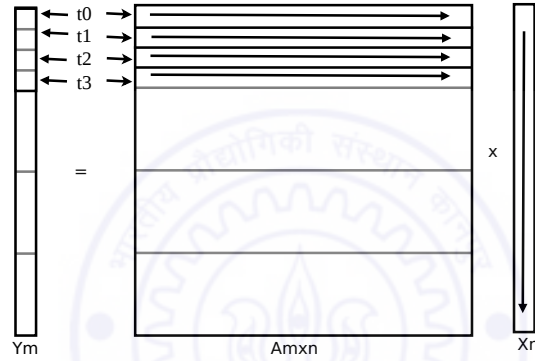


Figure 4.4: Memory Access Pattern of Naive Matrix-Vector Multiplication without Scratchpad Annotation

for this program. Adding a *scratchpad* annotation enforces that multiple threads process a tiled shaped subtask using shared memory. This requires explicit synchronization between the threads of the tiled block loaded on the Streaming Multiprocessor (SM). The optimized code with shared memory access is generated using the array access analysis provided by Algorithm 4.4. It uses the OFT shown in Figure 4.6 to identify the program point to compose shared memory accesses. The memory access pattern of coalesced global memory access code generated with the *scratchpad* annotation is shown in Figure 4.5. Adding the annotation *scratchpad*(A,16,16,1) to the code in Listing 4.7 generates the code shown in Listing 4.10.

An important use of the *shift* operator is in stencil computations as shown in Listing

```

__shared__ float shared_A[TILE][TILE];
i = blockIdx.x * BLOCK_SIZE + threadIdx.x
itx = threadIdx.x

if( i < height){
  for (j = 0; j < width; j = j + TILE){
    for(ii = 0; ii < TILE; ii = ii+1)
      shared_A[ii][itx]=A[(ii+i-itx)*width+(j+itx)];
    __syncthreads();
    for (k = 0; k < TILE; k = k+1){
      float a = shared_A[itx*TILE+k];
      float x = X[k+j];
      sum += a * x;
    }
    __syncthreads();
  }
  Y[i]=sum;
}

```

Listing 4.10: Matrix-Vector Multiplication Code with Coalesced Access of A

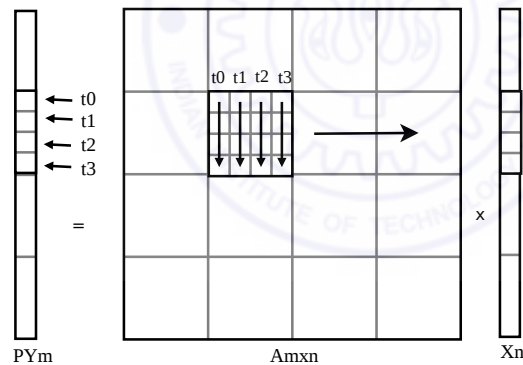


Figure 4.5: Memory Access Pattern of Matrix-Vector Multiplication using Scratchpad Annotation

4.8. Multiple occurrences of the *shift* operator applied on the same array is interpreted as the reuse of that array in stencil code. The use of the *shift* operator allows the compiler to extract data dependence information from the OFT. The OFT of Listing 4.8 is shown in Figure 4.7. Performing postorder traversal of the operation-level data flow tree allows the JolokiaC++ optimizer to optimize the use of the memory hierarchy through identifications

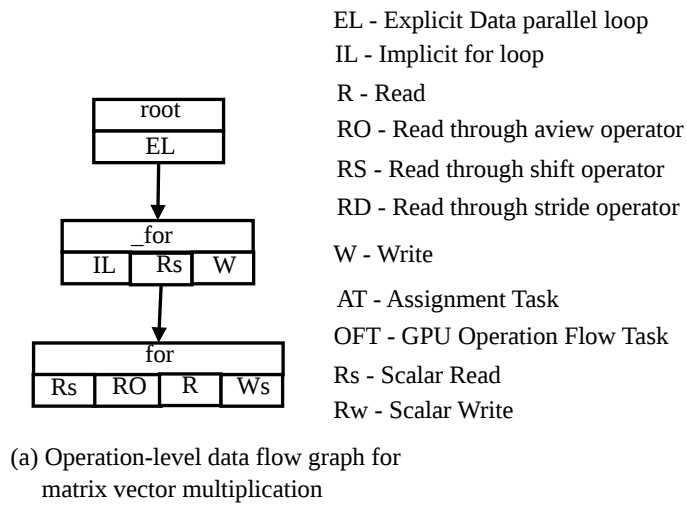
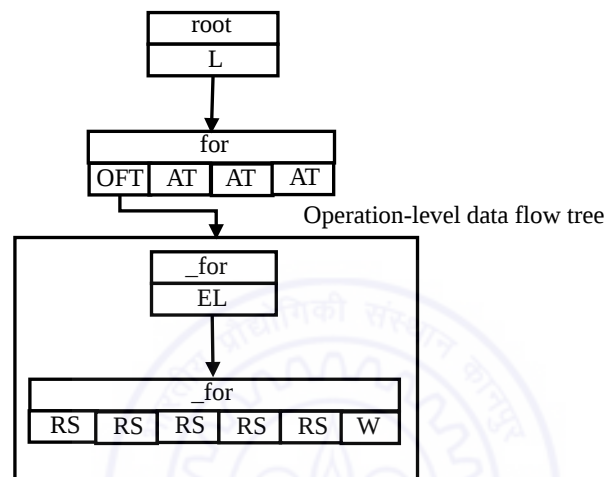


Figure 4.6: Operation-Level Data Flow Tree for Matrix-Vector Multiplication

of program points for different types of accesses.



(b) Task-level data flow graph and Operation-level data flow tree for Stencil code

Figure 4.7: Composed Task and Operation-Level Data Flow Tree

Chapter 5

Irregular Data Applications

Many scientific and engineering applications use sparse data structures or indirect memory references which leads to dynamic irregular memory accesses. Dynamic irregular memory accesses are a special class of irregular data references whose access patterns are not known at compile time. In this work, we deal with a class of dynamic irregularities with irregular memory access in loops that may cause cross-iteration dependences at runtime.

Example 5.1: Figure 5.1 shows an example having irregular memory accesses. The memory access patterns of $Y[\text{left}[i]]$ and $Y[\text{right}[i]]$ is determined from the runtime values of $\text{left}[i]$ and $\text{right}[i]$. Here, there is a possibility of dynamic cross-iteration dependence when Y is read in one iteration and written in another. Since the memory access patterns are unknown at compile time, it is not possible to identify the dynamic dependences at compile time. Therefore, it is not possible for the compiler to parallelize it. Being dynamic, these references are especially hard to tackle, making effective exploitation of GPUs difficult. ■

The massively parallel architecture of a GPU makes it possible to extract enormous computational performance but at the cost of complex programmability. At any time, hundreds or thousands of threads may try to issue reads or writes, and these accesses

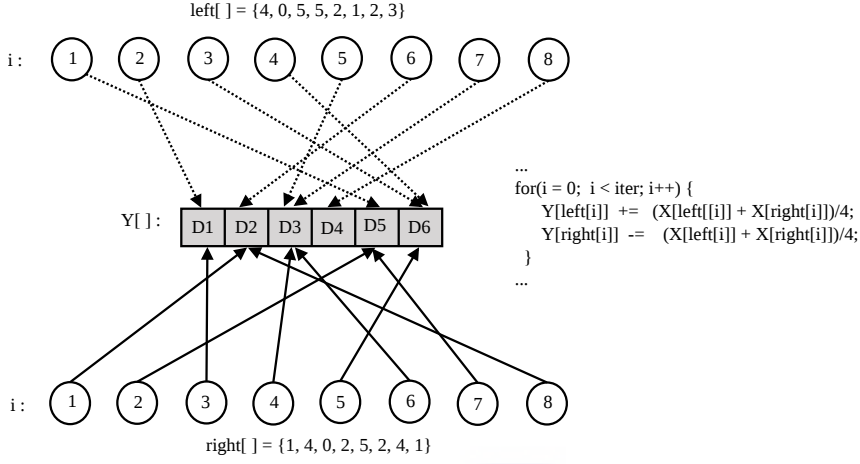


Figure 5.1: Irregular Memory Access

are serialized if the threads generate access patterns that are not optimized for the given memory organization. This makes applications running on data-parallel architectures extremely sensitive to irregularity in memory access patterns. Our framework (see fig. 5.2) maps concurrent language constructs of *JolokiaC++* to CUDA enabled GPUs for irregular data applications.

We have built *JolokiaC++* compiler framework for GPU based on a perspective to handle dynamic irregularities such as those illustrated in Example 1, with a motive to overcome inferior mapping between threads and data. This perspective leads to enhancing the thread-data mappings at runtime. We have developed a compile-time and runtime system that automatically orchestrate communication and computation, maps data to the GPU memory hierarchy, and tunes the kernel code to deliver considerable end-to-end performance. To support the system, we accurately identify the non-contiguous memory access points, called scatter and gather, in the computational kernel using compile time data flow analysis. Our composition involves simulating the shared memory of GPU as a cache. Further, we perform program transformations using Sparse Polyhedral Framework

to facilitate the actual data communication and computation through the introduction of uninterpreted function symbols. We also exploit situations to reuse communication schedules for amortizing the cost of the runtime data-flow analysis. We developed and tested various scheduling policies to eliminate cross-iteration dependences to provide coalesced access to threads of GPU to improve efficiency. To the best of our knowledge, the integration of compile time scatter and gather operation through program flow analysis for data dependence abstraction and program transformation is the first attempt towards automatic parallelization of irregular application on GPUs.

We now introduce the properties of GPU memory access which are of concern for irregular applications. The global memory in modern GPUs comprises of a large number of continuous segments. A typical global memory read or write operation takes 400 to 800 clock cycles. Therefore, coalescing global memory accesses is one of the most important optimizations. The load or store instructions will lead to coalesced accesses by a set of threads if these threads are within a warp and the words accessed by them lie in a single segment. An irregular reference refers to a load or store instruction, at which, the data requested by a warp happens to lie on multiple memory segments, causing more memory transactions than necessary. Because a memory transaction incurs a latency of hundreds of cycles, irregular references often degrade the effective throughput of a GPU significantly. We try to overcome this problem by generating schedules which can regroup data and iterations of loop kernels in such a way that the number of consecutive independent iterations is maximized for execution on streaming multiprocessors. The work presented in this chapter shows how a JolokiaC++ programmer can harness the computational capability of a GPU without writing complex low-level code for irregular kernels.

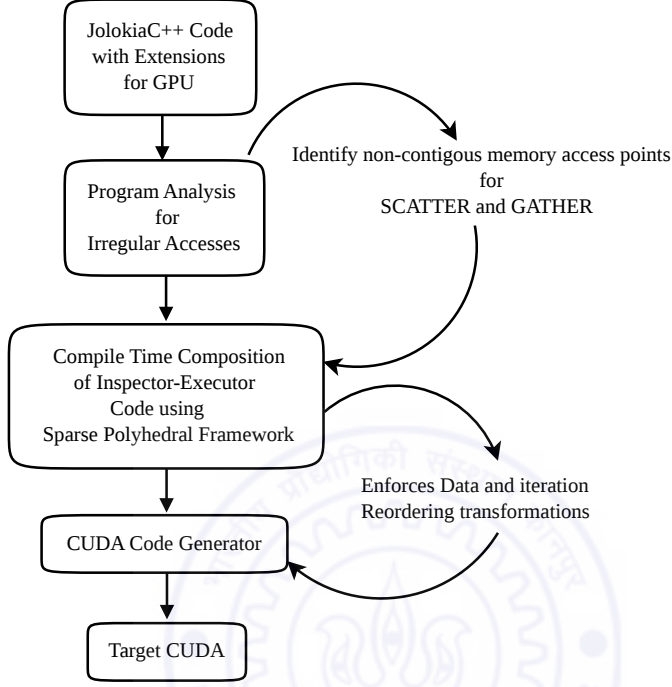


Figure 5.2: Compiler Framework for Optimizing Irregular Applications

5.1 Preliminaries of the Framework

First, we define some concepts used in the transformation framework. These concepts are explained using Examples 5.3 and 5.5 later in this chapter.

Definition 1 (Array Part). An array part (AP) $A[B(1:n)]$ consists of the array A and index array B with lower bound 1 and upper bound n .

Definition 2 (Loop Flow Graph (LFG)). A loop flow graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ of the program \mathcal{P} is a control flow graph, where the node $n \in \mathcal{N}$ represents a loop with indirectly accessed arrays or an entry/exit pad node. The edge $e \in \mathcal{E}$ denotes flow of control from one node to another.

Definition 3 (Wavefront). The set of concurrently executable loop iterations is a wave-

front.

Example 5.2: Listing 5.1 shows a code example. In this code, we have

- two inner loops l_1 and l_2 ;
- one entry and one exit pad, l_0 and l_3 ;
- two array names, x and y ;
- six array parts: $x1 = x[\text{IDENTITY}[1:\text{numMol}]]$, $x2 = x[\text{left}[1:\text{numIter}]]$, $x3 = x[\text{right}[1:\text{numIter}]]$, $y1 = y[\text{IDENTITY}[1:\text{numMol}]]$, $y2 = y[\text{left}[1:\text{numIter}]]$, $y3 = y[\text{right}[1:\text{numIter}]]$.

Here `IDENTITY` is an array¹ with the property `IDENTITY[i] = i`. The LFG for this code is shown in Figure 5.3. ■

5.2 Optimization of Irregular Applications

The transformation framework bears similarities with the work presented in [38] for distributed memory parallel machines, in which flow analysis is performed on the loop flow graph for communication of array parts. We use this framework to generate the executor's communication call for *GATHER* and *SCATTER* operations. We also enable the specification of run-time reordering transformations at compile time through the Sparse Polyhedral Framework [43]. This involves representing indirect memory references and run-time generated data together with iteration reordering using uninterpreted function symbols [54] which allow the composition of inspector and executor code.

In summary, we compose the runtime data and iteration reordering at compile time after analyzing the loop flow graph for communicating array parts. We perform loop

¹Note that `IDENTITY` is only a conceptual entity used for consistency of notations.

```

for i = 1 to numIter
l0:
#pragma jolokia tile(512,1,1) gpuIn(x,y)
  _for j = 1 to numMol nowait
l1:    x[j] = y[j]
  endfor
#pragma jolokia tile(256,1,1) gpuIn(x,y,left,right) gpuOut(y)
  _for k = 1 to numInter wait
l2:    y[left[k]] += (x[left[k]] + x[right[k]])/4
        y[right[k]] -= (x[left[k]] + x[right[k]])/4
  endfor
l3:
endfor

```

Listing 5.1: Simplified MOLDYN Kernel

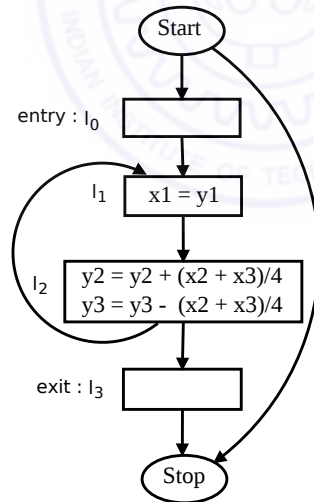


Figure 5.3: Loop Flow Graph for the Code in Listing 5.1

dependence analysis at compile-time in order to accomplish this. The compiler transforms the kernel, after analyzing the kernel's data access patterns using flow analysis algorithms covered in this section.

At run-time, wavefronts of concurrently executable loop iterations are identified by the inspector code. Using this wavefront information, loop iterations are reordered to increase parallelism. We use topological ordering to implement the scheduling mechanisms that perform run-time reordering transformations. This leads to partitioning of the index set into disjoint subsets of wavefronts, such that work pertaining to all indices in a wavefront may be carried out in parallel. We perform data reordering to provide spatial locality and to reduce non-coalesced access of global memory. The executor code generator is responsible for scheduling explicit memory communications between the global memory and the shared memory. We base the generation of the inspector and executor code on the specification of irregular computations and run-time reordering transformations presented in [43].

5.2.1 Flow Analysis Framework for GPU

We implement the analysis framework using a lattice of bit vectors. We obtain components of data flow equations by application of local flow analysis (Algorithm 5.1). The application of global flow analysis (Algorithm 5.2) leads to propagation of knowledge about the communication characteristics of the loops in the flow graph. Further, result flow variables are computed to describe the parts to be gathered before entering loop l or after leaving loop l .

At compile time, the data-flow framework analyzes the input kernel to build the memory communication schedule for transfer of data between the global memory and the shared memory. To accomplish this, the compiler has to transform the kernel after analyzing the data access patterns. Our framework for GPU targets analyzes the high level code to determine the points in the program for memory communication. This involves

Algorithm 5.1: Loop Flow Analysis

Input: Set of Loop Flow Graph $LFG \mathcal{G} = (\mathcal{N}, \mathcal{E})$ basic nodes \mathcal{N} , Global Program Information of Array Parts(AP)

Output: Local flow variables (LFV)

Let l stands for an arbitrary loop

p denotes a part of an array e.g. $x(\text{left}(\text{lb}:\text{ub}))$

for $l = 1 \rightarrow \tau$ **do**

GET(l) : The parts read in l from shared memory.

$\{p : \text{stmt in } l \text{ reads part } p\}$

PUT(l) : The parts written by l to shared memory.

$\{p : \text{stmt in } l \text{ writes to part } p\}$

TILE(l) : The parts tiled into blocks. They are large compared

 to the shared memory available on each streaming multiprocessor.

$\{p : \text{parts in } l \text{ require memory more than block size in } l\}$

BUFF(l) : The parts buffered in shared memory while exit from l

$BUFF(l) = (GET(l) \cup PUT(l)) \cap \overline{TILE(l)}$

KILL(l) : The parts that may be made invalid in l by modifying partial or full section of the array.

$\{p : \text{statements in } l \text{ invalidate part } p\}$

return

analyzing the loop flow graph for communication of array parts. The local flow analysis within the framework determines the prefetch and store candidates for a particular loop. As an outcome of this flow analysis, one can accurately describe the array parts to gather before entering loop l and scatter after leaving l . The result variables, i.e. *GATHER* and *SCATTER*, determine where the gather and scatter operations should be placed. For the given example we only use *SCATTER_{ADD}* operation.

To accomplish shared memory optimization, the compiler has to transform the kernel after analyzing the data access patterns. The JolokiaC++ framework analyzes the high level code to determine the points in the program for memory communication. The framework then composes executor code for shared memory access at the identified points. During program execution, the executor code composition examines the data references made by processor and calculates which off-processor data needs to be fetched from global memory and where this data will be stored once it is received in shared mem-

Algorithm 5.2: Global Flow Analysis

Input: Set of *LFG* Basic Nodes τ , Local Flow Variables (*LFV*), Global Program Information of *AP*

Output: Result flow variable

for $l = 1 \rightarrow \tau$ **do**

Global Flow Variable Analysis:

LIVE^{any/all}(*l*) : The part of array needed in *l* along any/all paths starting in *l*.

$$GET(l) \cup \bigcap_{s \in succ(l)} (LIVE^{all}(s)) \setminus KILL(l)$$

$$GET(l) \cup \bigcup_{s \in succ(l)} (LIVE^{any}(s)) \setminus KILL(l)$$

BUFFD(*l*) : The parts of arrays that are already available when entering *l*.

$$BUF(l) \cup \bigcap_{p \in preds(l)} (BUFFD(p) \setminus KILL(l))$$

HOIST(*l*) : The arrays for which a *GATHER* should be hoisted for *l*

$$\bigcap_{p \in preds(l)} (LIVE^{all}(p) \cup BUFFD(p))$$

FETCH(*l*) : The part of arrays that are needed in *l* or in some later loop. It can be hosted before *l*.

$$GET(l) \cup \bigcap_{s \in succs(l)} (HOIST(s) \cap FETCH(s))$$

Result Flow Variable Analysis:

$$1 \leq k \leq MAX_{SM}$$

GATHER(*l*) : It describes parts of array to be gathered before entering *l*

$$\text{foreach } SM_k, \forall i < SHARED_SIZE_x(SM_k) \\ shared_x[i] \leftarrow x[\sigma_{BLK}(SM_k, i)]$$

SCATTER(*l*) : It describes parts of array to be scattered after leaving *l*.

$$\text{foreach } SM_k, \forall i < SHARED_SIZE_y(SM_k) \\ y[\sigma_{BLK}(SM_k, i)] \leftarrow y[\sigma_{BLK}(SM_k, i)] + shared_y[i]$$

return

Algorithm 5.3: Automatic Shared Memory Tiling

Input: Program points for *GATHER* and *SCATTER_{ADD}* with input data arrays

for *Each array A* **do**

Partition all the wavefronts into maximal disjoint sets such that each partition has a subset of elements each of which is non-overlapping with any element in other partitions

for *Each partition of elements* **do**

Find the lower bound of elements in the partition which is an affine function of thread Index and Block Index.

Define the local storage for a partition accessed by elements of array A, with size based on the structure of the data array.

return

ory.

Example 5.3: When given an irregular kernel as shown in Listing 5.1, the result flow analysis determines the candidates for *GATHER* and *SCATTER_{ADD}* operations. Performing local flow analysis on the LFG shown in Figure 5.3, the bits corresponding to *left* and *right* part of array *x* and *y* are set to ‘1’ for loop l_2 . This allows identification of program points for composition of *GATHER* and *SCATTER_{ADD}*. Performing result flow analysis on the LFG shown in Figure 5.3, we get *x* and *y* as the candidates for *GATHER* and *SCATTER_{ADD}* respectively. Since, the bits corresponding to array *x* are set to ‘1’ in the *GATHER* set of loop l_2 , a gather operation for the array *x* is placed at the beginning of loop l_2 . Similarly, the bits corresponding to array *y* are set to ‘1’ in the *SCATTER_{ADD}* set of loop l_2 . This indicates placement of the scatter operation for the array *y* after loop l_2 . This is shown in Figure 5.4. We introduce shared memory optimization at program points (1) and (2) as shown in Listing 5.2 using uninterpreted function symbols. We create local memory storage for each non-overlapping region of the data space of an array that is accessed in a program block, thereby ensuring that data dependence relationships are preserved. ■

The *GATHER* operation at program point (1) performs global memory read of data

```

for i = 1 to numIter
  _for j = 1 to numMol nowait
    S1: x[i] = y[i]
  endfor
  GATHER(x) //-----(1)
  _for k = 1 to numInter wait
    S2: y[left[k]] += (x[left[k]] + x[right[k]])/4
    S3: y[right[k]] -= (x[left[k]] + x[right[k]])/4
  endfor
  SCATTERADD //----- (2)
endfor

```

Listing 5.2: Modified Kernel after Flow Analysis

array x . Localizing the computation and data to one SM (streaming multiprocessor) will tend to improve the performance of the code. Thus, localizing the data array x in the shared memory of each SM will allow the computations in each wavefront to reuse the data stored in shared memory. The result of the computation is stored in the shared memory of each SM to reduce global memory accesses for the store operation. We introduce shared memory reduction at program point (2) to combine the partial result computed by each SM via $SCATTER_{ADD}$. The $GATHER$ and $SCATTER_{ADD}$ operations are performed using shared memory tiling, which is expected to improve the performance. However, if the shared memory is not big enough to accommodate sufficient data, it will lead to global memory access. The detailed description on tiling for shared memory is covered in Section 5.3.

5.2.2 Code Generation using Sparse Polyhedral Framework

Existing loop transformation frameworks for GPUs [17] represent and manipulate iteration spaces as polyhedra and/or unions of polyhedra, which is restricted to loop bounds that are affine functions of outer iterators and symbolic constants. The limitations of this framework leads to conservative decisions of dependence on non-affine memory references occurring in important applications like sparse matrix and unstructured mesh computations. Fortunately, the Kelly and Pugh framework describes non-affine memory

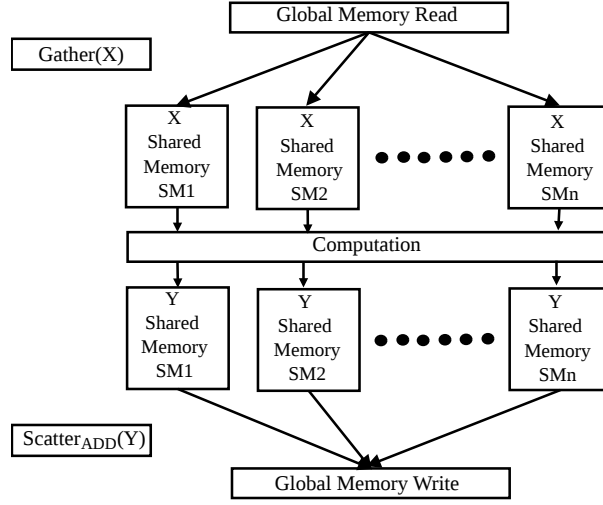


Figure 5.4: Scatter/Gather for Simplified MOLDYN Kernel

references of the type $A[B[i]]$ by using Presburger arithmetic with uninterpreted function symbols. We exploit this ability to specify data mappings between loop iterations and data locations, and dependences between loop iterations when non-affine memory references are involved. We base the generation of the inspector and executor on the specification of irregular computations and run-time reordering transformations using the Sparse Polyhedral Framework. This involves compile time composition of the inspector and executor based on identification of GATHER/SCATTER using the TILE data structure.

5.2.2.1 Compile Time Composition of Inspector

We use the formalization of run-time data and iteration ordering proposed by Strout et. al [69] to compose inspector code for GPUs in our framework. The composition involves exploiting wavefronts for the data parallel **_for** set iterator. The **_for** set iterator with **wait** clause enforces the use of the lock-free barrier described in Section 5.2.4. We collect the data mapping and dependence information via the flow analysis framework covered in Subsection 5.2.1. We then apply intra-loop run-time data and iteration re-ordering transformations. We explain this process for the simplified MOLDYN kernel

(Listing 5.1) in the remainder of this Section.

5.2.2.1.1 Data and Run-time Transformations Mapping The unified iteration space I_0 for the program is the following set:

$$I_0 = \{[i, 1, j, 1] \mid 1 \leq i \leq \text{numIter} \wedge 1 \leq j \leq \text{numMol}\} \\ \cup \{[i, 1, k, q] \mid 1 \leq i \leq \text{numIter} \wedge 1 \leq k \leq \text{numMol} \wedge 1 \leq q \leq 2\}$$

The data spaces associated with the simplified example is given by:

$$x_0 = \{[j] \mid 1 \leq j \leq \text{numMol}\} \\ y_0 = \{[j] \mid 1 \leq j \leq \text{numMol}\} \\ left_0 = \{[k] \mid 1 \leq k \leq \text{numIter}\} \\ right_0 = \{[k] \mid 1 \leq k \leq \text{numIter}\}$$

The data mappings for the MOLDYN example is as follows:

$$M_{I_0 \rightarrow x_0} = \{[i, 1, j, 1] \rightarrow [j]\} \\ \cup \{[i, 2, k, q] \rightarrow [left(k)]\} \\ \cup \{[i, 2, k, q] \rightarrow [right(k)]\} \\ M_{I_0 \rightarrow y_0} = \{[i, 1, j, 1] \rightarrow [j]\} \\ \cup \{[i, 2, k, 1] \rightarrow [left(k)]\} \\ \cup \{[i, 2, k, 2] \rightarrow [right(k)]\} \\ M_{I_0 \rightarrow left_0} = \{[i, 2, j, q] \rightarrow [k]\} \\ M_{I_0 \rightarrow right_0} = M_{I_0 \rightarrow left_0}$$

The dependences $D_{I' \rightarrow I'}$ of the new iteration space are given by:

$$D_{I' \rightarrow I'} = \{T_{I \rightarrow I'}(p_1) \rightarrow T_{I \rightarrow I'}(p_2) \mid p_1 \rightarrow p_2 \in D_{I \rightarrow I}\}$$

where for each $\{p_1 \rightarrow p_2\} \in D_{I \rightarrow I}$, $T_{I \rightarrow I'}(p_1)$ must be lexicographically earlier than

$T_I \rightarrow I'(p_2)$. The new data mapping $M_{I' \rightarrow a}$ for each array a is given by :

$$M_{I \rightarrow a'} = \{q \rightarrow R(m) \mid m \in M_{I \rightarrow a}(q)\}$$

Run-time Data Reordering for GPU We use the topological ordering mechanism (First Touch Policy) introduced by Ding and Kennedy [31] for run-time data reordering. Given a loop with indirect memory references like the k loop in Listing 5.2, run-time data reordering using FT (First Touch) policy can improve the spatial locality, which in turn provides coalesced global memory access. The FT policy traverses the iteration space of the loop in lexicographic order. The first time a loop touches a piece of data, that data is packed into the next location for the new data mapping. The First Touch Policy specialized for the original data mapping $M_{I_0 \rightarrow x_0}$ is given in Algorithm 5.4. It is composed as part of inspector code to perform run-time data reordering. Here, σ_{FT} records the new permutation order of the elements of array x according to the first touch policy. The new data mapping is specified as follows.

$$\begin{aligned} M_{I_0 \rightarrow x_0} = & \{[i, 1, j, 1] \rightarrow [\sigma_{FT}(j)]\} \\ & \cup \{[i, 2, k, q] \rightarrow [\sigma_{FT}(\text{left}(j))]\} \\ & \cup \{[i, 2, k, 1] \rightarrow [\sigma_{FT}(\text{right}(j))]\} \end{aligned}$$

The above mapping leads to run-time data reordering of x and y array which is given by $R_{x_0 \rightarrow y_0} = \{i \rightarrow \sigma_{FT}(i)\}$.

Example of Run-time Data Reordering Composition for GPU The executor code after applying the data reordering transformation on the simplified MOLDYN kernel is as shown in Listing 5.3. Here, each occurrence of $x[\dots]$ and $y[\dots]$ in the original code is replaced by $nx[\sigma_{FT}[\dots]]$ and $ny[\sigma_{FT}[\dots]]$ respectively. Applying first

Algorithm 5.4: First Touch Policy

Input: left, right
Output: Data Reordering
Initialize *tag_firstTouch* array to all -1 and *count* to 0
for $i = 1 \rightarrow \text{numInter}$ **do**
 index1 \leftarrow left[i]
 index2 \leftarrow right[i]
 if *tag_firstTouch*[index1] = -1 **then**
 | *tag_firstTouch*[index1] \leftarrow count
 if *tag_firstTouch*[index2] = -1 **then**
 | *tag_firstTouch*[index2] \leftarrow count
 count \leftarrow count + 1
for $j = 1 \rightarrow \text{numMol}$ **do**
 if *tag_firstTouch*[j] = -1 **then**
 | *tag_firstTouch*[j] \leftarrow count
 | count \leftarrow count + 1
return

touch requires additional code given in Listing 5.4 for the inspector. It is introduced to enforce pointer update, data alignment and iteration alignment. The executor code after the alignment and update is shown in Listing 5.5.

Run-time Iteration Reordering for GPU Iteration-level parallelism is achieved when different iterations from a loop are executed in parallel. However, to ensure that the semantics of the code is retained, execution of the loop iterations must be in accordance with their inter-iteration dependences. Thus, inter-iteration dependences inhibit iteration parallelization. We propose an iteration-level loop parallelization technique with loop transformation to maximize loop parallelism. Our basic idea is to migrate inter-iteration data dependences by regrouping iterations of a loop kernel in such a way that the number of consecutive independent iterations is maximized. We perform iteration reordering at runtime to extract iteration-level parallelism after exploiting spatial locality using the data reordering mechanism. The run-time iteration reordering involves exploiting data parallelism through wavefront generation. This involves modeling the data dependence

```

// Copy data to reordered location
for i = 1 to numMol
  nx[ $\sigma_{FT}$ [i]] = x[i]
  ny[ $\sigma_{FT}$ [i]] = y[i]
endfor

// simplified MOLDYN computation
for i = 1 to numIter
  _for j = 1 to numMol nowait
    nx[ $\sigma_{FT}$ [i]] = ny[ $\sigma_{FT}$ [i]]
  endfor
  _for k = 1 to numInter wait
    ny[ $\sigma_{FT}$ [left[k]]] += (nx[ $\sigma_{FT}$ [left[k]]] + nx[ $\sigma_{FT}$ [right[k]]])/4
    ny[ $\sigma_{FT}$ [right[k]]] -= (nx[ $\sigma_{FT}$ [left[k]]] + nx[ $\sigma_{FT}$ [right[k]]])/4
  endfor
endfor

// Copy data to original location
for i = 1 to numMol
  x[i] = nx[ $\sigma_{FT}$ [i]]
  y[i] = ny[ $\sigma_{FT}$ [i]]
endfor

```

Listing 5.3: Executor Code for Simplified MOLDYN using FT

```

for i = 1 to numInter
   $\sigma_{left}$ [i] =  $\sigma_{FT}$ [left[i]]
   $\sigma_{right}$ [i] =  $\sigma_{FT}$ [right[i]]
endfor

```

Listing 5.4: Additional code for inspector

among the iterations in a loop by converting a single dimension interaction list to a two dimensional interaction list. Pseudo code for wavefront generation is shown in Algorithm 5.5. The modified iteration domain contains iterations in each wavefront which can be executed in parallel. The barrier (shown as dotted line) separates the iteration domain of each pair of consecutive wavefronts. The overall process of wavefront generation is shown in Figure 5.5. The iteration reordering of the j loop and the k loop based on their mappings to the data arrays x and y is specified as follows.

```

// Copy data to reordered location
for i = 1 to numMol
    nx[σFT[i]] = x[i]
    ny[σFT[i]] = y[i]
endfor

// simplified MOLDYN computation
for i = 1 to numIter
    _for j = 1 to numMol nowait
        nx[i] = ny[i]
    endfor
    _for k = 1 to numInter wait
        ny[σleft[k]] += (nx[σleft[k]] + nx[σright[k]]) / 4
        ny[σright[k]] -= (nx[σleft[k]] + nx[σright[k]]) / 4
    endfor
endfor

// Copy data to original location
for i = 1 to numMol
    x[i] = nx[σFT[i]]
    y[i] = ny[σFT[i]]
endfor

```

Listing 5.5: Executor Code for Simplified MOLDYN after Alignment and Update

$$\begin{aligned}
 T_{I_0 \rightarrow I_1} = & \{[i, 1, j, 1] \rightarrow [i, 1, \sigma_{FT}(j), 1] \\
 & \cup [i, 2, k, q] \rightarrow [i, 2, \sigma_{WF}(k), q]\}
 \end{aligned}$$

However, the data mappings after wave-front generation remains unchanged for the given example code.

Example of Run-time iteration Reordering Composition for GPU Given a loop with the access pattern shown in Figure 5.5(b) the wavefront generation process transforms the iteration domain to contain the wavefront shown in Figure 5.5(d). We apply a regularization function given in Algorithm 5.6 on the wavefronts to reduce non-coalesced global memory access for the threads within a block. This is applied before generating schedules for shared memory access.

Algorithm 5.5: Wavefront Generation

Input: left, right, maxWavefronts
Output: wf
for $i = 1 \rightarrow numInter$ **do**
 $wf[i] \leftarrow -1$
 $\sigma_{WF}[i] \leftarrow -1$
for $i = 1 \rightarrow maxWavefronts$ **do**
 $maxIndexEachWF[i] \leftarrow -1$
for $i = 1 \rightarrow numInter$ **do**
 if $\sigma_{WF}[left[i]] > \sigma_{WF}[right[i]]$ **then**
 $currentWF \leftarrow \sigma_{WF}[left[i]] + 1$
 else
 $currentWF \leftarrow \sigma_{WF}[right[i]] + 1$
 $wf[i] \leftarrow currentWF$;
 $\sigma_{WF}[left[i]] \leftarrow currentWF$
 $\sigma_{WF}[right[i]] \leftarrow currentWF$
 $maxIndexEachWF[currentWF]++$
return

Algorithm 5.6: Wavefront Regularization

Input: σ_{WF} , maxIndexEachWF
Output: $n\sigma_{left}$, $n\sigma_{right}$
for $i = 1 \rightarrow numMol * wfFormed$ **do**
 $n\sigma_{left}[i] \leftarrow 0$
 $n\sigma_{right}[i] \leftarrow 0$
 $temp[i] \leftarrow 0$
for $i = 1 \rightarrow numInter$ **do**
 $currentWf \leftarrow \sigma_{WF}[i]$
 $tempMaxEachWf \leftarrow maxIndexEachWF[currentWf] + 1$
 $idx \leftarrow \sigma_{left}[i]$
 $temp[currentWf * numMol + idx] \leftarrow \sigma_{right}[i]$
 $maxIndexEachWF[currentWf] \leftarrow tempMaxEachWf$
for $i = 0 \rightarrow wfFormed$ **do**
 $p \leftarrow 0$
 for $j = 0 \rightarrow numMol$ **do**
 if $temp[i * numMol + j] \neq 0$ **then**
 $n\sigma_{left}[i * numMol + p] \leftarrow j$
 $n\sigma_{right}[i * numMol + p] \leftarrow temp[i * numMol + j]$
 $p \leftarrow p + 1$
return

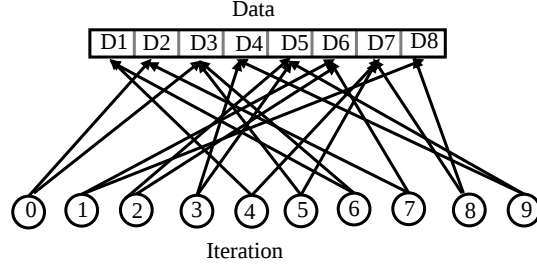
left[10] = {2, 6, 5, 4, 7, 3, 1, 6, 7, 4};
 right[10] = {3, 8, 6, 5, 1, 7, 3, 2, 8, 5};

(a) Original iteration list

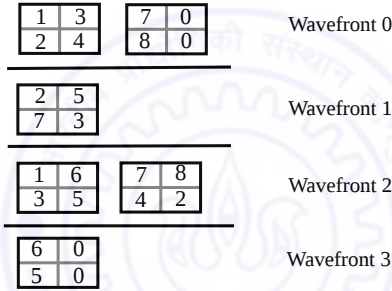
1 --> 8
 2 --> 1
 3 --> 2
 4 --> 6
 5 --> 5
 6 --> 3
 7 --> 7
 8 --> 4

D2 D3 D6 D8 D5 D4 D7 D1

(c) Data Reordering using First touch policy



(b) Original Data and Iteration Access Pattern



(d) Transformed Domain (Wavefront Generation)

Figure 5.5: Compile Time Composition of Inspector

5.2.3 Executor Code Generation

Generating executor code for the transformed computation requires determining loop bounds for the new loop iterators and determining new array access functions within the context of the transformed iteration space. Each **for** loop is transformed to a kernel for execution on the GPU using Algorithm 5.6 and 5.7. To illustrate the process, we consider the iteration space of the code shown in Listing 5.1 for generating executor code for the transformed computation. The kernel surrounded by *GATHER* and *SCATTER* as shown in Listing 5.2 is the candidate for transformation of indirect references performed in *x* and *y*. The original iteration space specification for the single “statement” associated with this

Algorithm 5.7: Intra-Loop Memory Access Mechanism

```

for  $i = 1 \rightarrow numForall$  do
  if  $loop(i)$  is without Indirect access then
    | Generate Regular Blocked code for GPU
  else if  $loop(i)$  contains Indirect access then
    | Use Algorithm 5.3 to generate shared memory tiled code
  return

```

loop is

$$I = \{[i, k] : (0 \leq i < numIter) \wedge (0 \leq k < numInter)\}$$

The iteration space set specification after applying data and iteration reordering and partitioning is

$$\begin{aligned}
 I' = \{[i, GATHER, w, k, SCATTER] : \\
 & (0 \leq i < numIter) \wedge GATHER \\
 & \wedge (0 \leq w < wfNumber)\} \\
 & \wedge (0 \leq k < \sigma(SM_p)) \wedge SCATTER\}
 \end{aligned}$$

The proposed framework generates the executor code as shown in Listing 5.6.

5.2.4 Inter-Block synchronization using lock-free barrier

We experimented by implementing the *wait* clause composition using both lock-free and lock-based barriers. Both implementations provide inter-block synchronization that does not involve the host CPU and thus eliminates the overhead of switching back and forth between the GPU and CPU.


```

for i = 1 to numIter
  GATHER(x)
  for w = 1 to wfNumber
    forall k such that wf[k] = w
      foreach element in GATHER and SCATTER
        t = g(SM)
        if(t in SHARED_X RANGE)
          t1 = SHARED_X[t]
        else
          t1 = x[t]
        endforeach
      endforall
    endfor
  SCATTER(y)
endfor

```

Listing 5.6: Executor code

The basic idea of GPU lock-based synchronization [75] is to use a global mutex variable to count the number of thread blocks that reach the synchronization point. The leading thread will then repeatedly compare the global mutex variable to a target goal value. If it is equal to the goal value, the synchronization is completed and each thread block can proceed with its next wave-front. The goal value is set to $MAX_{SM} * wfNumber$ in the kernel when the barrier function is first called. Our lock-free barrier is outlined in Algorithm 5.8. Our algorithm uses an array *barrCounter* to coordinate synchronization requests from various thread blocks. We map each array element to a thread block to keep track of all the blocks. This mechanism is expected to scale well with increase in the number of thread blocks. We compared our lock-free implementation with the lock-based barrier proposed by Xiao et. al. [75]. We found that our lock-free barrier implementation performs equally well when compared to the lock-based implementation, and is expected to do better with increased contention.

Algorithm 5.8: Lock-free Interblock Barrier

Require: $gNumber$
Ensure: Inter Block synchronization
if $threadIdx = 0$ **then**
 $barrCounter[blockIdx] \leftarrow gNumber$
end if
if $threadIdx < gridDim$ **then**
 $Counter \leftarrow \&barrCounter[threadIdx]$
 repeat
 $\{...\}$
 until $Counter \neq gNumber$
end if

5.3 Empirical Search for Selection of Optimal Tile Size and Scheduling Policy

In this section, we discuss the choice of system parameters such as threads/block used for execution, and the availability of GPU local resources such as shared memory and registers. The number of active threads per block at a given time depends on the amount of shared memory and registers available for execution. We performed tiling for exploiting temporal locality across thread blocks and threads, which in turn reduces the number of loads/stores from global memory. We did an empirical search to find an optimal tile size of threads per block for the tiled loops associated with shared memory access for localizing the access to shared memory. We varied the parameters *tile* and *scratchpad* operator to determine the optimal grid dimension and shared memory size. We found that a shared memory size of 1024 and 512 resulted in optimal performance, with grid dimensions based on the total number of elements for processing. We evaluated 1D and 3D implementation of all the benchmarks with different block dimensions and concluded with 512 threads per block as the most suitable block dimension. We improve the parallelization capability of the GPU by using run-time data and computation ordering performed by the composed inspector. We employ empirical search to pick the optimal scheduling

policy out of various scheduling policies like Random Scheduling, Regularized Block Scheduling, Locality Based Scheduling to model the set iterator for iteration reordering. In Random Scheduling the threads associated with the set iterator access the data placed in random order. We regularize the data access of threads associated with the set iterator by padding the wave-fronts with '0' in the interaction list where molecules are not involved in the interaction. This would lead to coalesced global memory access and reduction in bank conflicts. Further, bringing frequently used data into shared memory allows us to localize the access of data which in turn results in improving performance.





Chapter 6

Performance Measurement

In this chapter, we evaluate the functioning of our framework on benchmarked codes with regular and irregular access kernels. We present and discuss the performance of the codes using the overall execution time and speedup.

6.1 Experimental Methodology and Performance Results for Regular Access Kernels

We implemented the proposed framework using ROSE [56], a source-to-source compiler infrastructure available for C, C++, FORTRAN programs. The output of JolokiaC++ compiler is compiled by the CUDA compiler, nvcc[2], to generate an executable file for the GPU. We evaluated the effectiveness of our framework using CUDA driver version 6000 on an NVIDIA GeForce GTX 770 using CUDA version 5.5. The test set used for demonstrating the effectiveness of the framework is shown in Table 6.1. We evaluated all the benchmarks with and without *scratchpad* annotations. The Jolokia-C++ naive indicates an evaluation without *scratchpad* annotations, while the Jolokia-C++ shared is with *scratchpad* annotations. We compare the execution time of the code generated by our framework with the OpenACC implementation to determine the speedup over the sequen-

Test set	Time step & Size of Input	Hints for Reuse Pattern	Access pattern
Blackscholes	4000000	No reuse	Linear access
Matrix-Vector Multiplication	131072 * 1024 4096 x 4096	Implicit loop through aview operator	Row Linear access
Matrix-Matrix Multiplication	1024 x 1024 2048 x 2048	Implicit loop through aview operator	Row and Column Linear access
Vector Addition	9000000	No reuse	Linear
Jacobi 1-D	65536 x 65536	Multiple access through shift operator	Shifted Access
Jacobi 2-D	10 X 4096 x 4096 100 x 4096 x 4096	Multiple access through shift operator	Shifted Access
Heat 2-D	20 x 4096 x 4096 100 x 4096 x 4096	Multiple access through shift operator	Shifted Access
Convolution	2048 x 2048 4096 x 4096	Multiple access through shift operator	Shifted Access

Table 6.1: Test Set and their Configuration

tial CPU implementation. We also compare the execution time with the codes available in CUDA-SDK. The sequential implementation is tested on Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz with 32GB RAM, and 8MB L3 shared cache, running UBUNTU 10.04. We used PGI compiler version 13.10 to compile OpenACC codes for evaluation. We investigated eight kernels: one from the PARSEC [7] benchmark suite, three from BLAS and four stencil based codes from the Chombo [26] toolkit, Rodinia [24, 23] benchmark suite and PolyBench [9]. The Blackscholes benchmark from the PARSEC benchmark suite solves a partial differential equation that describes the price of an option over time. Vector-Addition (AXPY), Matrix-Vector multiplication (GEMV) and Matrix-Matrix multiplication (GEMM) are Basic Linear Algebra Subprograms (BLAS) kernels. We have considered 2D heat conduction, Jacobi and Convolution which are stencil-based codes, as the applications to run on the GPU. We chose these applications primarily because the computation to communication ratio is quite high. The computation portion is intensive and it can be distributed to several threads.

The 1-D and 2-D Jacobi solvers are from the Rodina benchmark suite used to determine Lagrangian's of a wide variety of nonlinear differential equations. 2D-Heat Benchmark is from the Chombo toolkit used for modeling heat conduction. It uses the Gauss-Seidel (GS) method for modeling. The GS kernel is repeated until the standard deviation between adjacent 2D matrices is less than a predefined convergence value. In image processing, many operators are based on applying some function to the pixels within a local window. That is, to find the value of an output pixel a window is centered at that location and only pixels falling within this window; are used to calculate the value of that output pixel. 2-D Convolution from PolyBench suite is one such operator which performs weighted average of the pixels within the window to calculate the value of the output pixel.

6.1.1 Ease of Programming

Our goal here is to address those issues that prevent programmers from exploiting the full potential of GPUs. This requires developing programming language constructs and paradigms to express parallelism without explicitly understanding the underlying architecture. We developed an annotation based language constructs that makes GPU programming simpler and less error-prone. To support these annotations we integrate optimizations to shield the programmer from the complexity of the underlying architecture. The input code given in Listing 6.1 illustrates the simplicity of our annotation based language constructs. We asked two programmers, each having one year of CUDA programming experience to evaluate the simplicity and performance of our framework. Programmers appreciated the productivity gains and ease of use associated with the framework. Controlled experiments with more number of users need to be performed to claim the ease of JolokiaC++ over CUDA.

```

blackscholes(f32Array &cRes, f32Array &pRes, f32Array &sPrice,
             f32Array &oStrike, f32Array &oYears, float rFactor,
             float vol, int optN){
    ...
    sqrtYears = sqrt(oYears);
    d1 = logf(sPrice/oStrike) +
        (rFactor + 0.5 * vol * oYears)/(vol * sqrtYears);
    d2 = d1 + Volatility * sqrtOptionYears;
    k1 = 1/(1 + 0.2316419f * abs(d2));
    k2 = (k1 * (X1 + k1 * (X2 + k1 * (X3 + k1 * (X4 + k1 * X5)))));
    cnd1temp = rsqrt2PI * expf(-0.5f * d1 * d1) * k2;
    cnd2temp = rsqrt2PI * expf(-0.5f * d2 * d2) * k2;
    _if (d1 > 0.0f){
        CND1 = cnd1temp;
    }
    else {
        CND1 = 1.0 - cnd1temp;
    }
    _if (d2 > 0.0f){
        CND2 = cnd2temp;
    }
    else {
        CND2 = 1.0 - cnd2temp;
    }
    expRT = expf(-rFactor * oYears);
    cRes = sPrice * CND1 - oStrike * expRT * CND2;
    pRes = oStrike * expRT * (1.0f - CND2)
        - sPrice * (1.0f - CND1);
}

int main(int argc, char **argv)
{
    ...
    #pragma jolokia gpuIn(sPrice,oStrike,oYear)
    gpuOut(pRes,cRes) tile(BLOCK,1,1)
    blackscholes(cRes,pRes,sPrice,oStrike,oYear,
                RiskFactor,Volatility,optN);
}

```

Listing 6.1: Blackscholes Code Snippet in JolokiaC++

Benchmark	Dimensions	CPU	JolokiaC++		OpenCL	OpenACC
			Naive	Shared		
Convolution	2048 x 2048	55.35	28.75	20.58	29.84	51.24
	4096 x 4096	220.94	59.67	45.34	49.18	117.3
Matrix-Matrix Multiplication	1024 x 1024	6500.55	52.21	29.11	70.21	95.57
	2048 x 2048	156338.2	257.39	36.85	273.51	297.18
Matrix-Vector Multiplication	131072 x 1024	375.23	9.55	9.61	30.43	31.61
	4096 x 4096	47.18	16.69	6.03	29.12	31.45
Jacobi 2-D	10 x 4096 x 4096	1695.21	113.28	97.4	413.38	711.54
	100 x 4096 x 4096	16822.56	288.11	179.34	420.12	11261.39
Jacobi 1-D	65536 x 65536	14089.1	447.75	578.34	300.23	282.64
Blackscholes	4000000	785.79	20.95	-	358.99	311.29
Vector-Addition	9000000	42.62	47.87	-	99.41	9.57
Heat 2-D	20 x 4096 x 4096	2500.28	44.02	42.67	429.02	781.13
	100 x 4096 x 4096	16248.29	168.93	134.25	439.43	10681.4

Table 6.2: Execution time of benchmarks in milliseconds

Benchmark	Dimensions	JolokiaC++		OpenCL	OpenACC
		Naive	Shared		
Convolution	2048 x 2048	1.925	2.689	1.855	1.080
	4096 x 4096	3.702	4.873	4.492	1.884
Matrix-Matrix Multiplication	1024 x 1024	124.508	223.295	92.587	68.019
	2048 x 2048	607.384	4242.212	571.600	526.081
Matrix-Vector Multiplication	131072 x 1024	39.299	39.057	12.331	11.869
	4096 x 4096	2.826	7.830	1.620	1.500
Jacobi 2-D	10 x 4096 x 4096	14.965	17.405	4.101	2.382
	100 x 4096 x 4096	58.389	93.803	40.042	1.494
Jacobi 1-D	65536 x 65536	31.466	24.361	46.928	49.849
Blackscholes	4000000	37.511	-	2.189	2.524
Vector-Addition	9000000	0.890	-	0.429	4.454
Heat 2-D	20 x 4096 x 4096	56.800	58.596	5.828	3.201
	100 x 4096 x 4096	96.182	121.030	36.976	1.521

Table 6.3: Speedup of the benchmarks over sequential CPU implementations

6.1.2 Performance Results and Discussion

The graph based analysis performed on the input and output set passed as a parameter to the *gpuIn* and *gpuOut* annotation in Blackscholes allows our framework to completely parallelize the code. Figure 6.6 shows the execution times of Blackscholes. The execution time of code generated by our framework is compared with the implementation that comes with Nvidia CUDA SDK, OpenACC and OpenCL implementations. The OpenCL code used for comparison is without the use of shared memory. The execution time and speedup of all the benchmarks is as shown in Table 6.2 and 6.3 respectively. It is observed from the results that there is minimal overhead of accessing memory through object instantiation.

BLAS kernels

In this experiment, we tuned the performance of AXPY, GEMV and GEMM on NVIDIA GeForce GTX 770. The GEMV and GEMM kernels are tested with and without *scratchpad* annotations. The parallel JolokiaC++ version contains jolokia parallelization directives with necessary parameters to launch and tune the kernel for transformation of code. The experimental results shown in Figure 6.2 demonstrate the performance in terms of execution time for GEMV and GEMM respectively. Here, the unit used to measure the performance is in terms of milliseconds. The corresponding speedup for GEMV and GEMM is shown in Figure 6.1. The y axis in all the graphs is plotted using log scale. The performance gain of JolokiaC++ shared memory implementation of GEMM outperforms the OpenACC, JolokiaC++ naive and CUDA-SDK implementations. However, this is not the case with AXPY as seen in Figure 6.6. It has been observed that OpenACC outperforms most codes when there is a single loop. The JolokiaC++ shared memory implementation of GEMV gives reasonably good performance in comparison to the hand-coded CUDA implementation. The handcoded CUDA implementation outperforms the

Parameters	GeForce GTX 480	Tesla C1060	Tesla K20c
Global Memory (MB)	1536	4096	4800
GPU clock rate (GHz)	1.40	1.30	0.71
Memory clock rate (MHz)	1848	800	2800
Compute Capability	2.0	1.3	3.5

Table 6.4: Configuration of GPUs

JolokiaC++ shared memory implementation due to lack of loop unrolling support in our composition.

Stencil Codes

Most stencils exhibit a high degree of temporal locality because each update operation accesses neighboring values on the grid. Typically, in an n -dimensional stencil there is data reuse along all n dimensions. Since each time step sweeps over the same data grid, stencils also exhibit data locality in the time dimension. A multiple appearance of the array accessed through the **shift** operator in the source code is also considered as temporal reuse of an array. We evaluated the applicability of the *shift* operator on stencil codes like Heat 2-D, Jacobi and Convolution codes using our framework. The execution time for Heat, Jacobi and Convolution is shown in Figure 6.4a, 6.3a and 6.5a. The JolokiaC++ shared memory implementations of Heat 2-D, Jacobi 2-D and Convolution outperforms the OpenACC implementations as shown in Figure 6.4b, 6.3b and 6.5b. However, there is scope to further optimize the code by reusing data through the efficient use of Registers.

6.2 Experimental Evaluation of Irregular Access Kernels

We evaluated our framework on a multicore CPU and manycore GPUs. We present the experimental results from compiling the three kernels (IRREG, MOLDYN and NBF) for

Dimensions	Number of Molecules	IRREG		NBF		MOLDYN	
		Seq	Par	Seq	Par	Seq	Par
1D	10000	20.56	18.23	40.53	28.33	318.88	112.37
	100000	263.55	168.88	518.19	261.61	3342.74	1074.89
	1000000	9789.39	4042.23	16303.55	5052.27	44198.76	12827.68
3D	10000	54.77	45.49	223.59	113.81	224.59	173.05
	100000	661.37	335.15	2420.62	852.28	2463.78	1301.87
	1000000	21764.79	9734.17	43529.65	13627.44	57806.82	17464.79

Table 6.5: Execution time of Sequential (Seq) and Parallel (Par) hand-coded CPU implementation in milliseconds

execution on GeForce GTX 480, Tesla K20c and Tesla C1060. Their configurations are listed in Table 6.4. We use, Intel(R) Core(TM) i3@3.20 GHz with 4 GB RAM and 4 MB (L3 Cache) to measure the performance of sequential and parallel CPU versions of the code.

We evaluated the performance of our framework by generating synthetic data for aggregation benchmarks: MOLDYN, IRREG and NBF. MOLDYN is a computational kernel extracted from the CHARMM simulation package. CHARMM [22] simulates the properties of atoms and molecules in liquid and solid systems. MOLDYN is the main computational kernel which iterates over all interacting atoms and molecules, and updates the forces acting on both interacting entities. IRREG [38] models the unstructured meshes, which are represented by nodes and edges. This benchmark finds application in computational fluid dynamics(CFD). The kernel is an iterative PDE solver. NBF belongs to the GROMOS molecular dynamics code. The NBF [38] kernel extracted from GROMOS tracks the evolution of the n-body particle system based on the force that is applied between the molecule and its interacting partner. An irregular reduction form of access pattern is present in all these benchmarks.

The *With Shared* plot in the graph corresponds to a composition which uses shared memory for *GATHER* and *SCATTER_{ADD}* operations. The *Without Shared* plot in the graph corresponds to a composition without the use of shared memory. The *Parallel*

6.2. EXPERIMENTAL EVALUATION OF IRREGULAR ACCESS KERNELS 87

Benchmarks	Dimensions	Number of Molecules	Geforce GTX 480	Tesla C1060	Kepler K20c
IRREG	1D	10000 100000 1000000	Without Shared		
			40.14	149.95	213.27
			215.62	933.39	780.77
			2238.77	10426.44	5719.97
		10000 100000 1000000	With Shared		
			34.89	120.69	202.23
			175.77	782.67	622.35
			1507.13	6032.54	4995.82
	3D	10000 100000 1000000	Without Shared		
			69.43	338.2	281.08
			524.38	3088.58	1644.53
			5414.6	37273.48	15156.28
		10000 100000 1000000	With Shared		
			53.91	236.97	244.87
			279.89	1737.26	1595.26
			4645.51	15899.41	14633.69

Table 6.6: Execution Time of IRREG in milliseconds

CPU plot corresponds to execution of OpenMP parallel code on the CPU. The execution time of sequential and OpenMP based parallel implementation of IRREG, MOLDYN and NBF is given in Table 6.5. The execution time of IRREG, MOLDYN and NBF is given in Table 6.6, 6.7 and 6.8 respectively. A scatter plot of a sample data with 1000 molecules is shown in Figure 6.7.

Performance

The speedup of IRREG, MOLDYN, and NBF on the test platforms used for experimentation is shown in Figure 6.8, 6.9, and 6.10 respectively. The speedup of execution on GPUs is measured by taking into consideration the runtime data and iteration reordering overhead along with the data transfer and kernel invocation overhead. The data transfer overhead incurred in the overall execution time is shown in Figure 6.11. The reordering overhead incurred in the overall execution time is shown in Table 6.9. The low data trans-

Benchmarks	Dimensions	Number of Molecules	Geforce GTX 480	Tesla C1060	Kepler K20c
MOLDYN	1D	10000 100000 1000000	Without Shared		
			40	149.59	213.18
			225.85	924.46	794.19
			2361.6	9430.16	5740.76
		10000 100000 1000000	With Shared		
			35.88	120.13	202.23
			178.515	782.13	622.35
			1516.81	6030.35	4995.82
	3D	10000 100000 1000000	Without Shared		
			79.02	373.66	285.38
			543.92	3095.13	1666.53
			5423.4	37249.3	15160.88
		10000 100000 1000000	With Shared		
			52.85	236.09	244.87
			275.91	1736.13	1595.26
			4634.23	15900.41	14633.69

Table 6.7: Execution Time of MOLDYN in milliseconds

fer overhead along with higher clock speed in Fermi GTX 480 appears to be the reason for the high speedup compared to the other test platforms used in the experiments.

From the graph shown in Figure 6.8, 6.9, and 6.10, we can infer that the performance of *With Shared* composition is better than *Without Shared* composition. One of the main reasons behind the performance improvement is the introduction of a scheduling mechanism for iteration reordering to expose temporal locality. This involves regularizing one of the two interaction lists and reordering the second interaction list accordingly. The regularized iteration reordering results in coalesced access to one interaction list and its associated data. This plays an important role in providing spatial locality and in turn improves the performance compared to the irregular access implementation on the GPU.

The shared memory tiling exploits temporal locality to gauge the performance on the GPU. However, shared memory size imposes a limitation on achievable performance. Unavailability of data in shared memory results in non-coalesced global memory access,

6.2. EXPERIMENTAL EVALUATION OF IRREGULAR ACCESS KERNELS 89

Benchmarks	Dimensions	Number of Molecules	Geforce GTX 480	Tesla C1060	Kepler K20c
NBF	1D	10000 100000 1000000	Without Shared		
			41.47	150.48	211.65
			213.63	931.74	782.04
			2105.93	10372.95	5730.01
		10000 100000 1000000	With Shared		
			35.32	120.13	202.23
			173.86	782.18	622.35
			1451.34	6031.69	4995.82
	3D	10000 100000 1000000	Without Shared		
			72.55	338.01	284.64
			541.26	3094.27	1669.8
			5546.46	37229.42	15386.37
		10000 100000 1000000	With Shared		
			52.8	237.87	244.87
			282.43	1737.15	1595.26
			4638.25	15900.41	14633.69

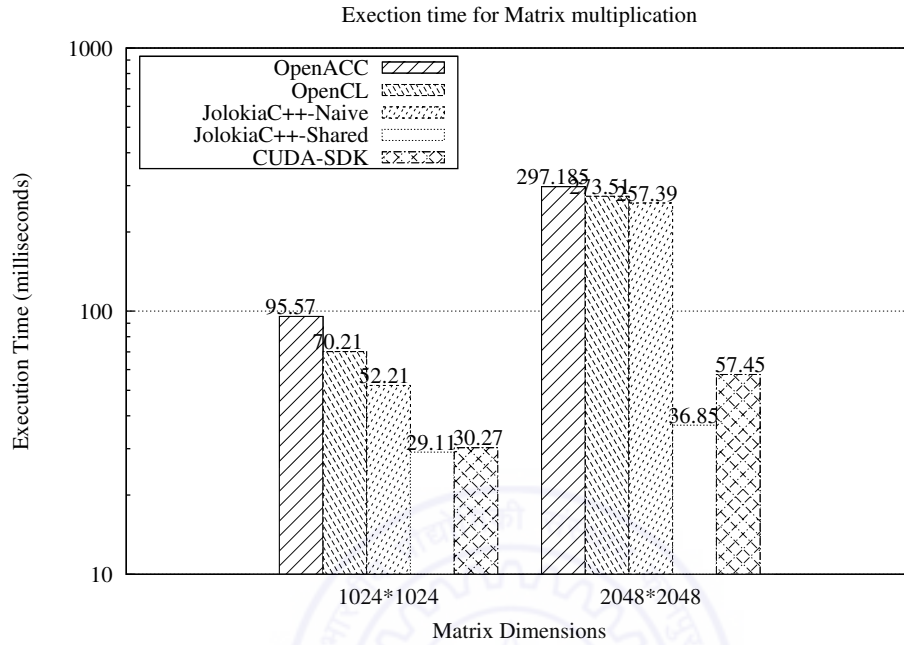
Table 6.8: Execution Time of NBF in milliseconds

Molecules/Interactions	Reorder Time
10000	1.54
100000	20.81
1000000	310.25

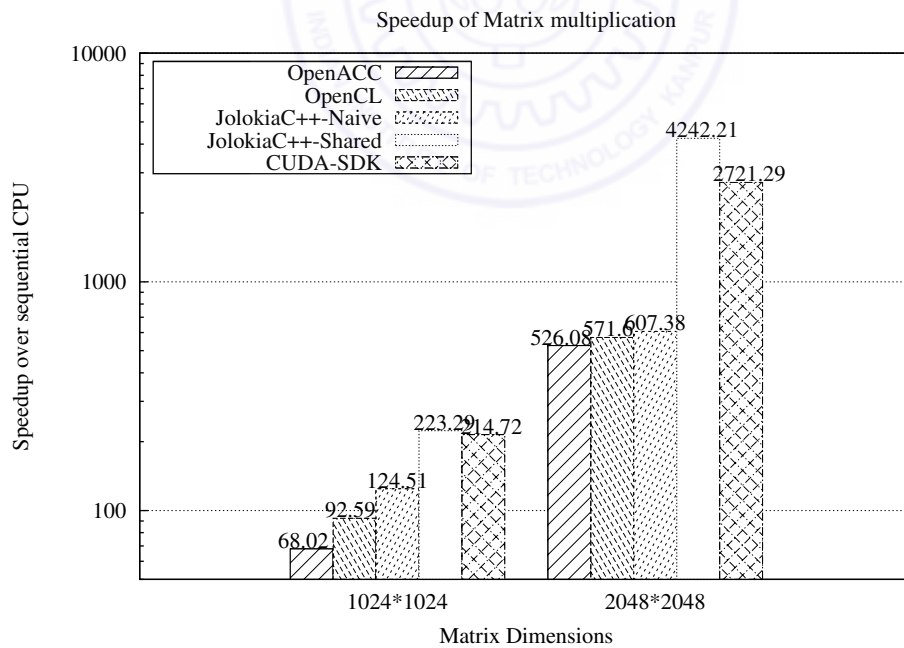
Table 6.9: Average Reordering Time in milliseconds

which can lead to significant performance degradation. However, maximizing the locality based access in the memory hierarchy of the GPU helped us in bridging the memory wall problem to a large extent.

We developed a framework to efficiently and easily port irregular scientific and engineering applications with subscripted subscript. The framework showed appreciable speedup on many-core processors with minimal effort from the programmer. The current implementation is limited to loops with no loop-carried dependences except those used for accumulation of results.

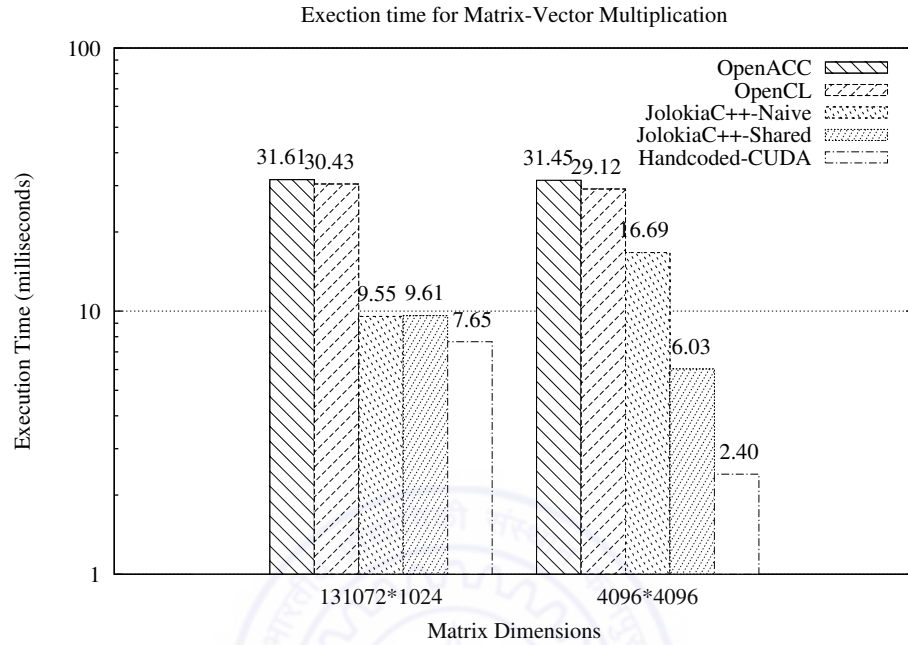


(a) Execution time of Matrix Multiplication

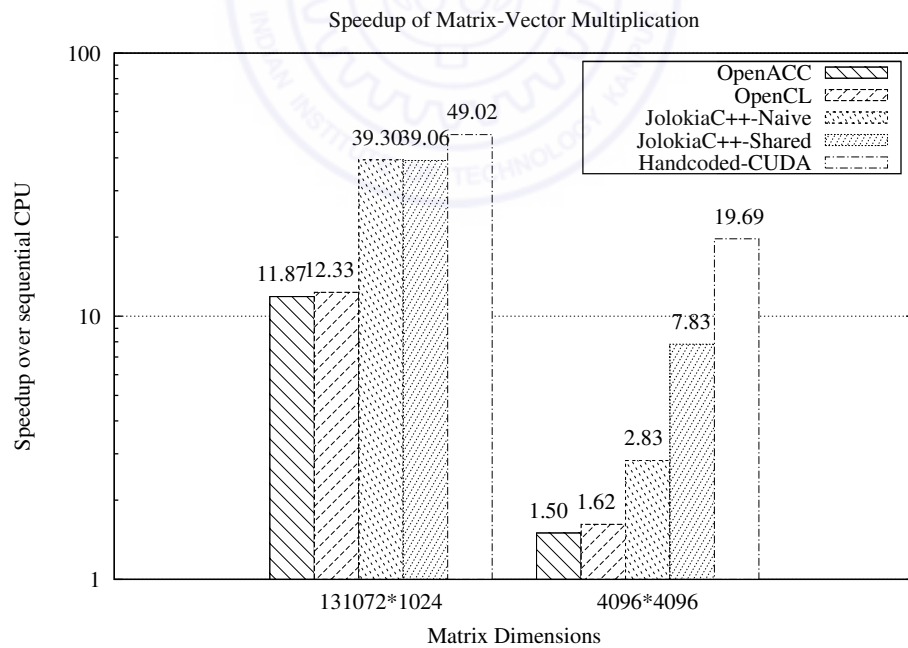


(b) Speedup of Matrix Multiplication

Figure 6.1: Performance of Matrix Multiplication

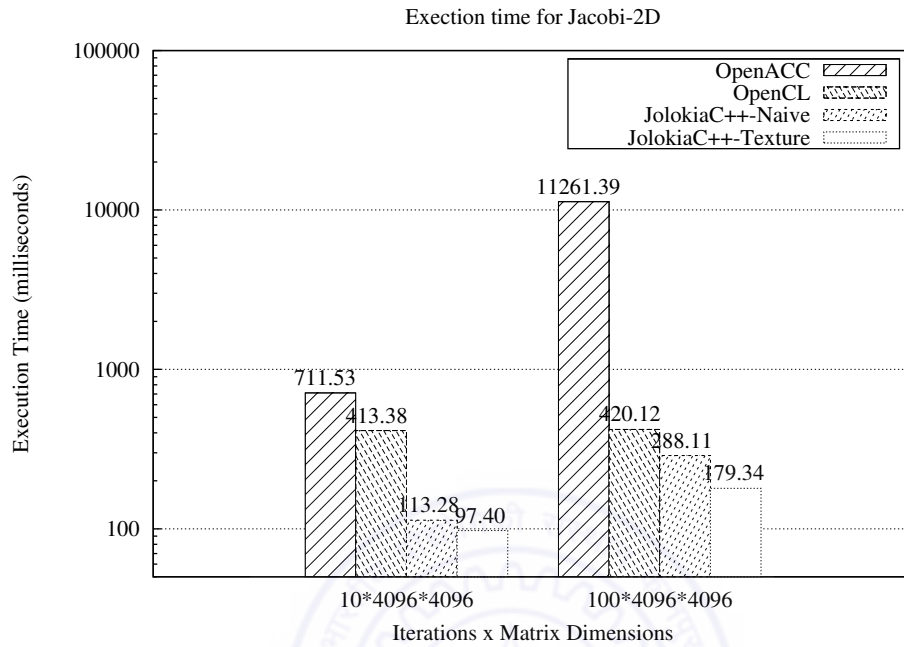


(a) Execution time of Matrix Vector Multiplication

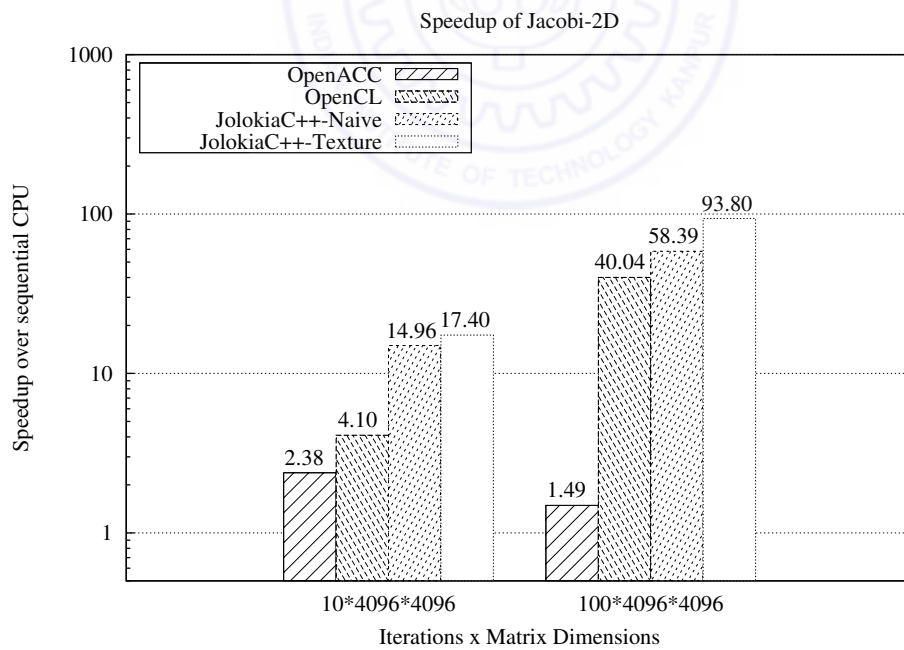


(b) Speedup of Matrix Vector Multiplication

Figure 6.2: Performance of Matrix Vector Multiplication

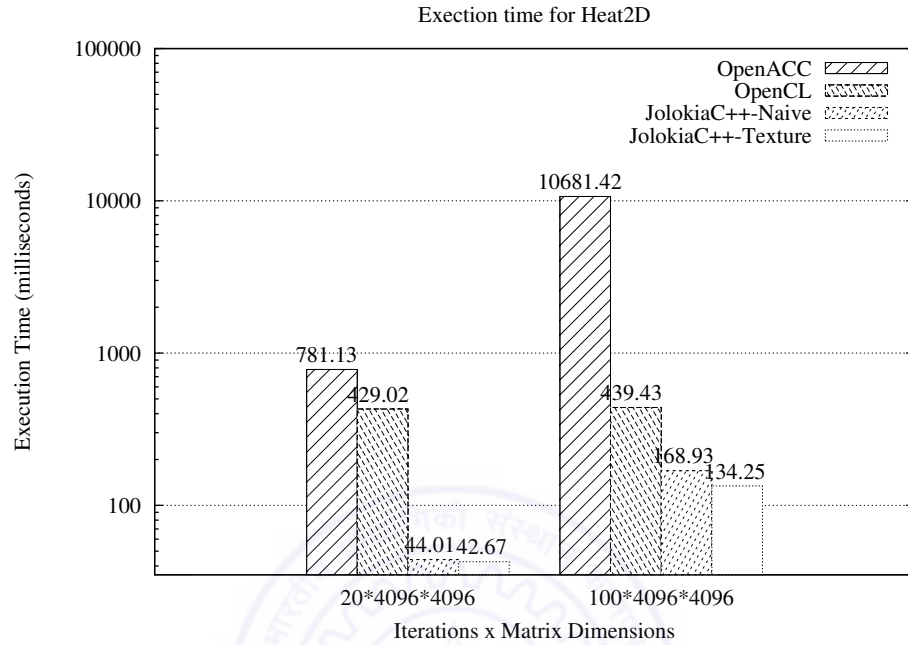


(a) Execution Time of Jacobi 2D

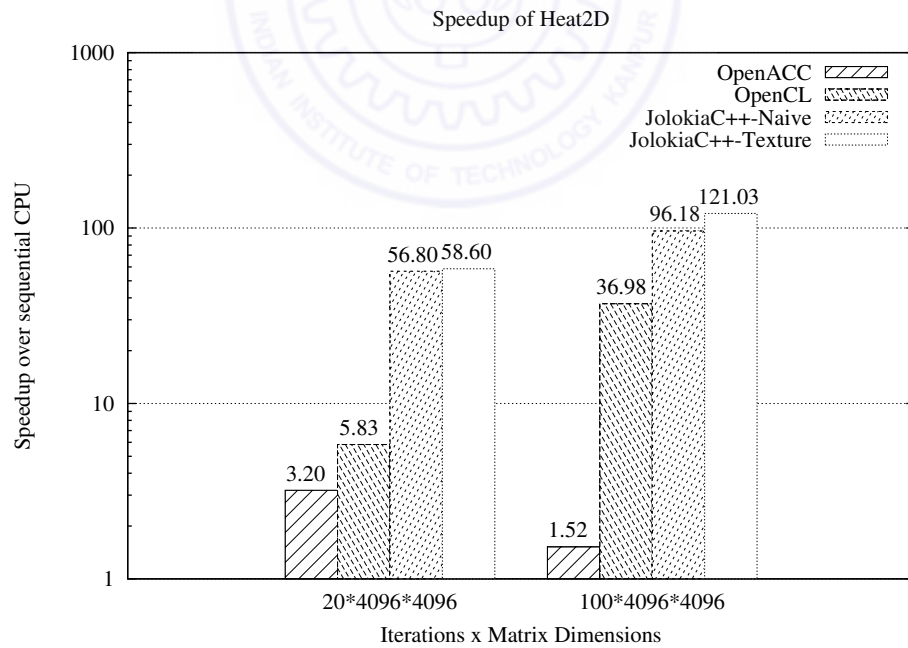


(b) Speedup of Jacobi 2D

Figure 6.3: Performance of Jacobi 2D

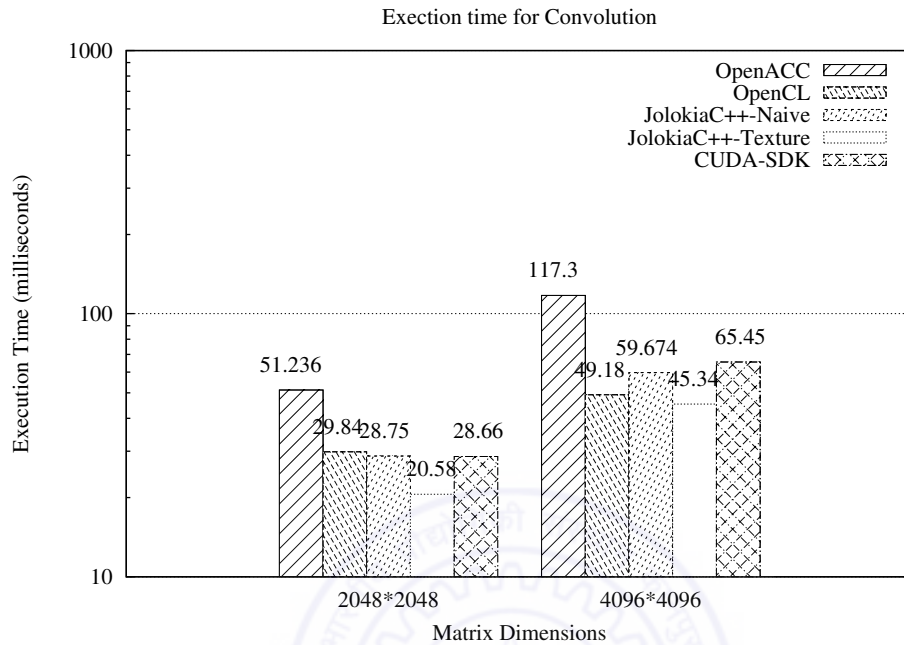


(a) Execution Time of Heat 2D

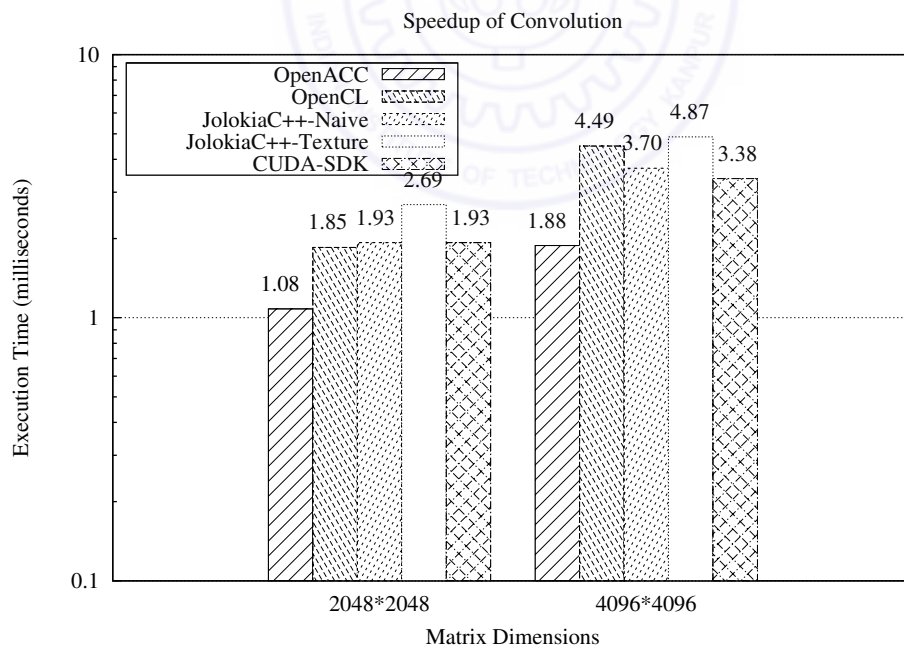


(b) Speedup of Heat 2D

Figure 6.4: Performance of Heat 2D



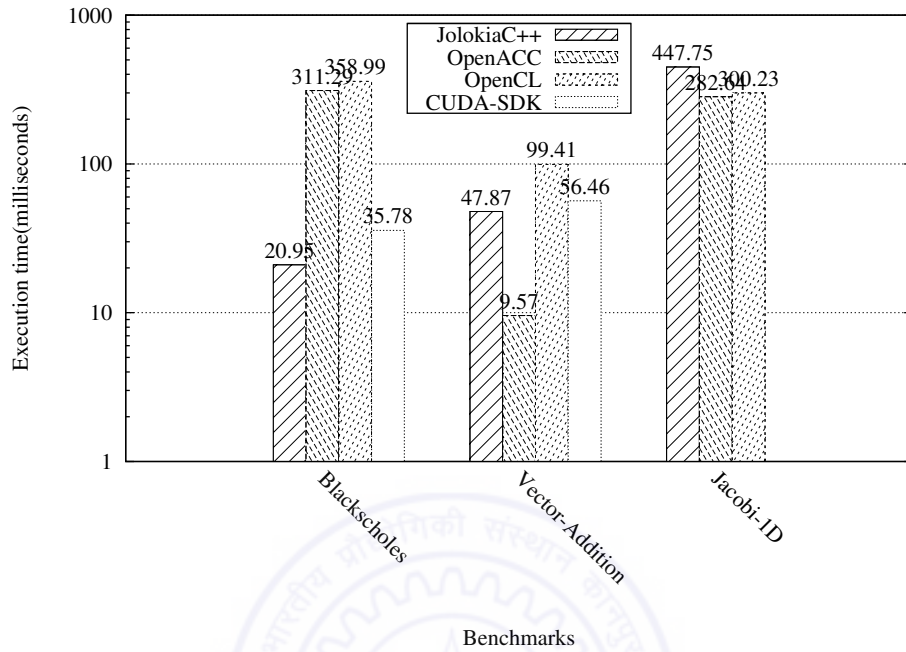
(a) Execution Time of Convolution



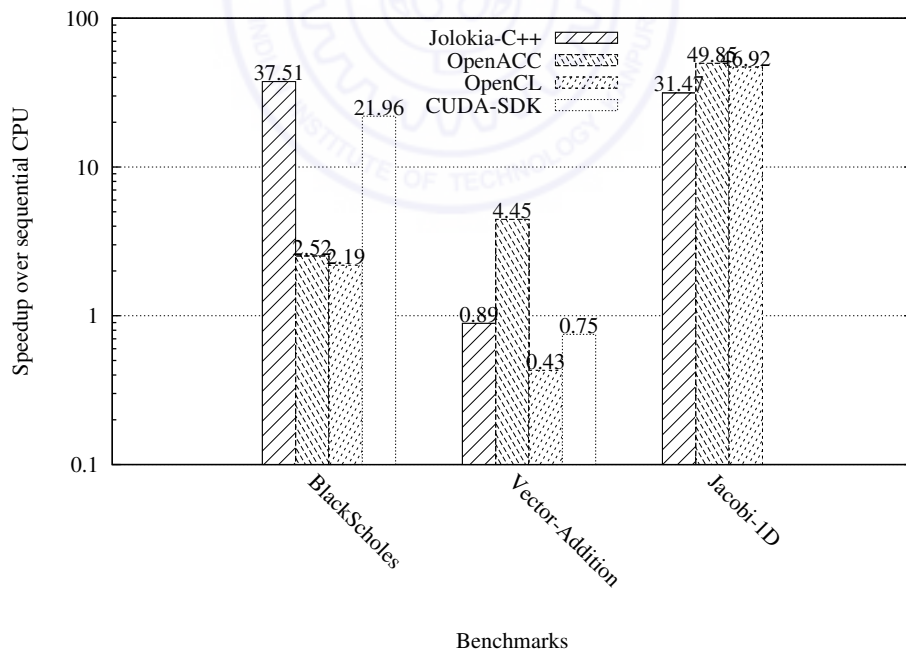
(b) Speedup of Convolution

Figure 6.5: Performance of Convolution

6.2. EXPERIMENTAL EVALUATION OF IRREGULAR ACCESS KERNELS 95



(a) Execution Time of 1D Benchmarks



(b) Speedup of 1D Benchmarks

Figure 6.6: Performance of 1D Benchmarks

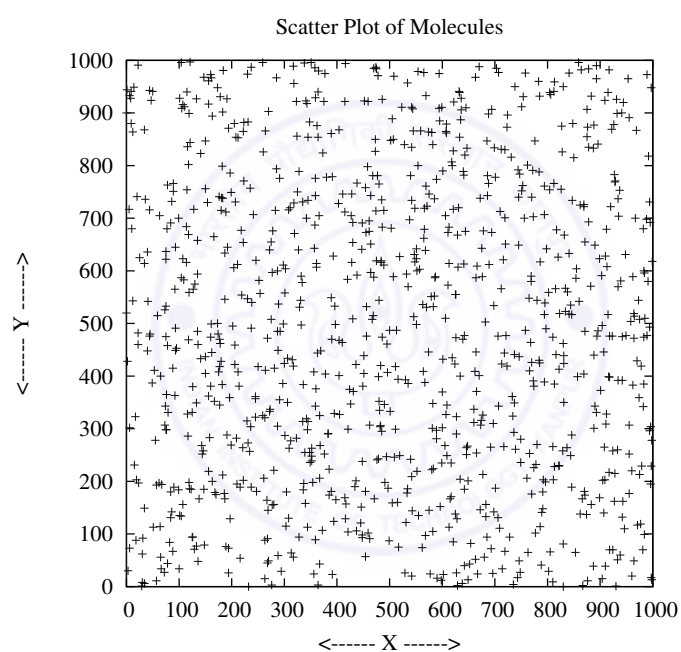
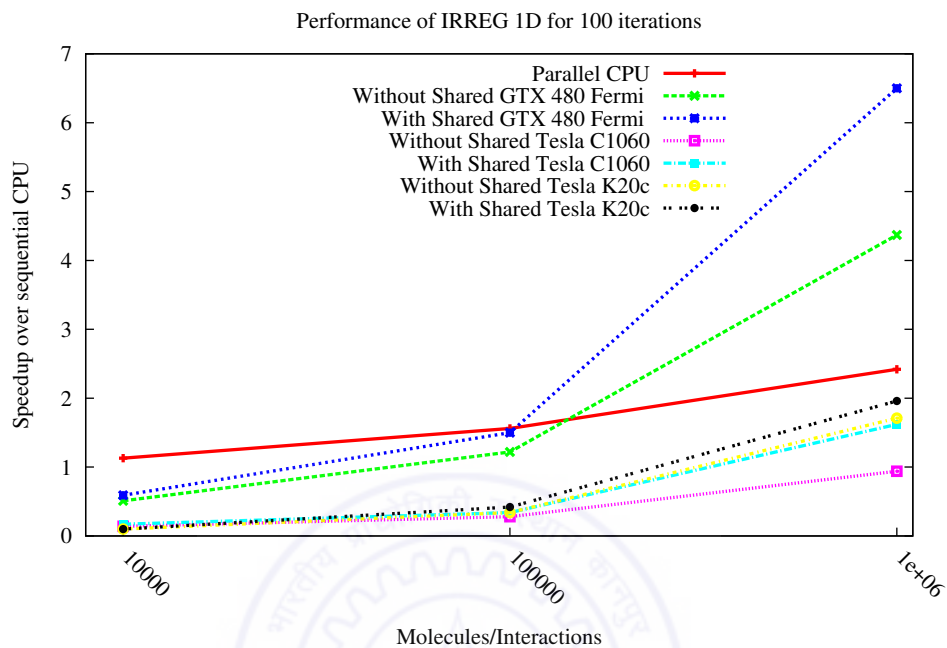
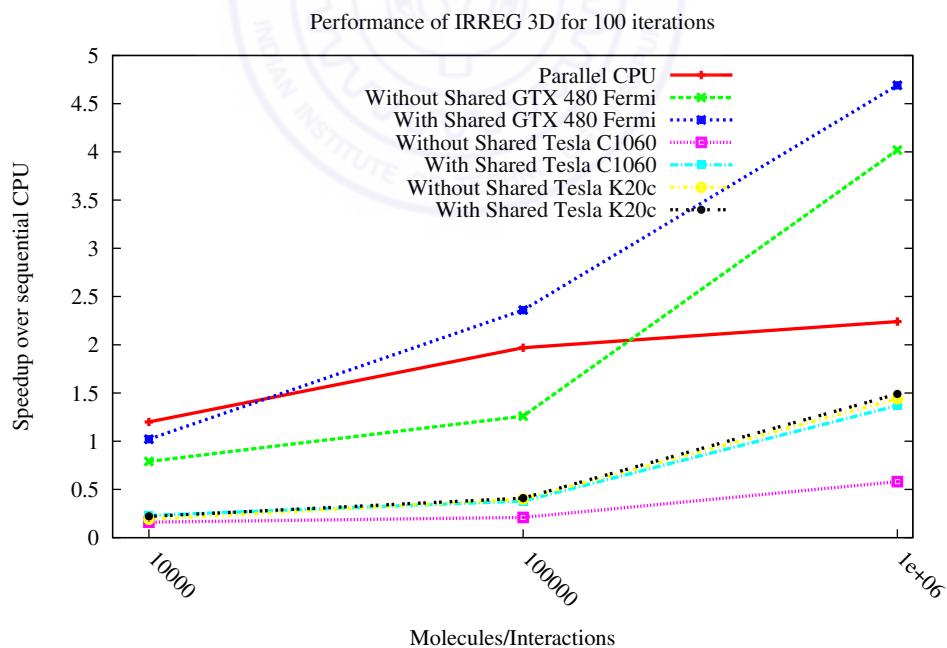


Figure 6.7: Scatter plot of 1000 molecules

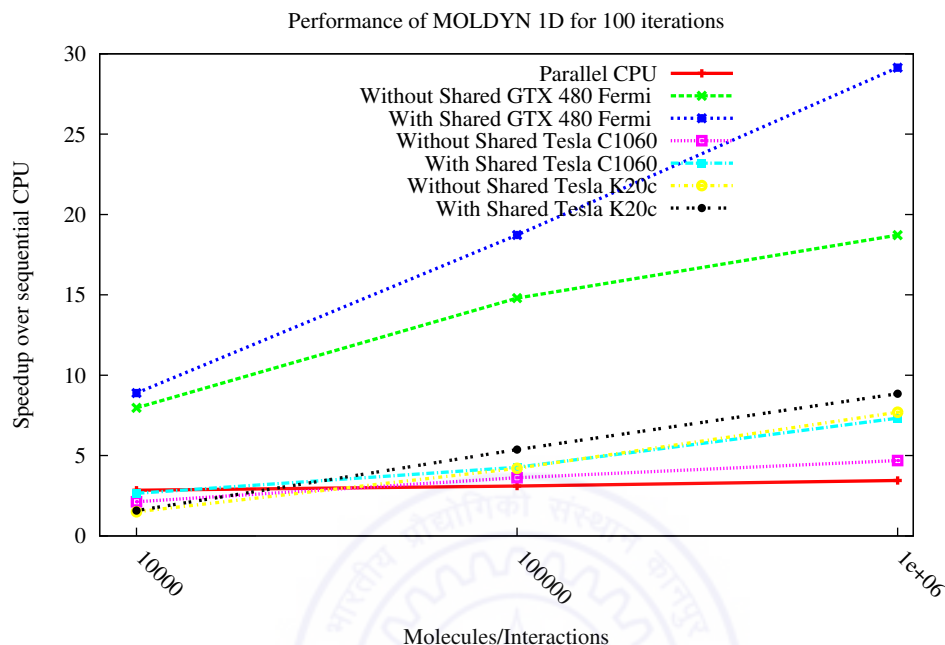


(a) Speedup of IRREG 1D

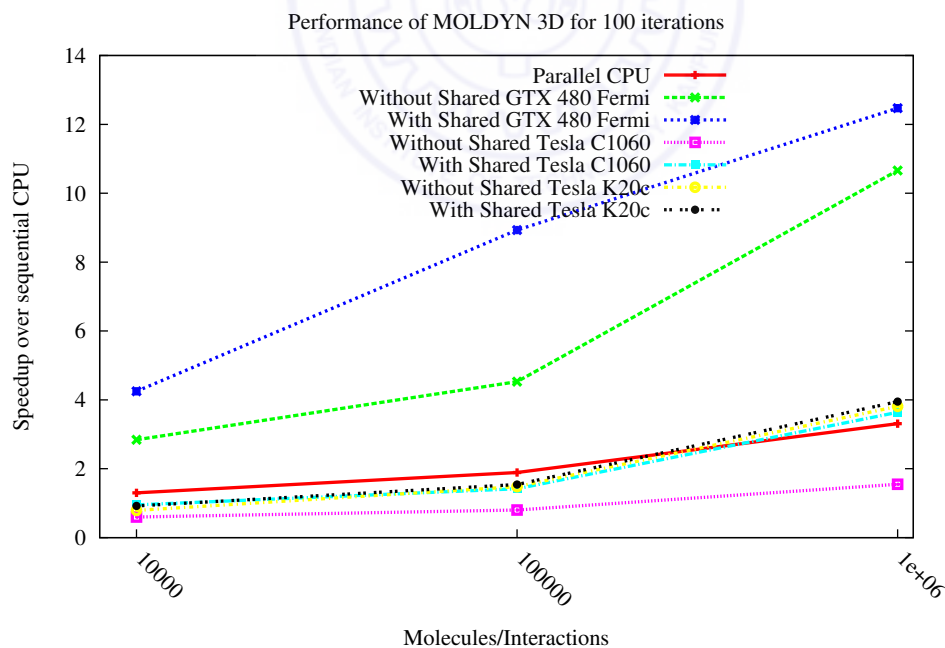


(b) Speedup of IRREG 3D

Figure 6.8: Performance of IRREG kernel

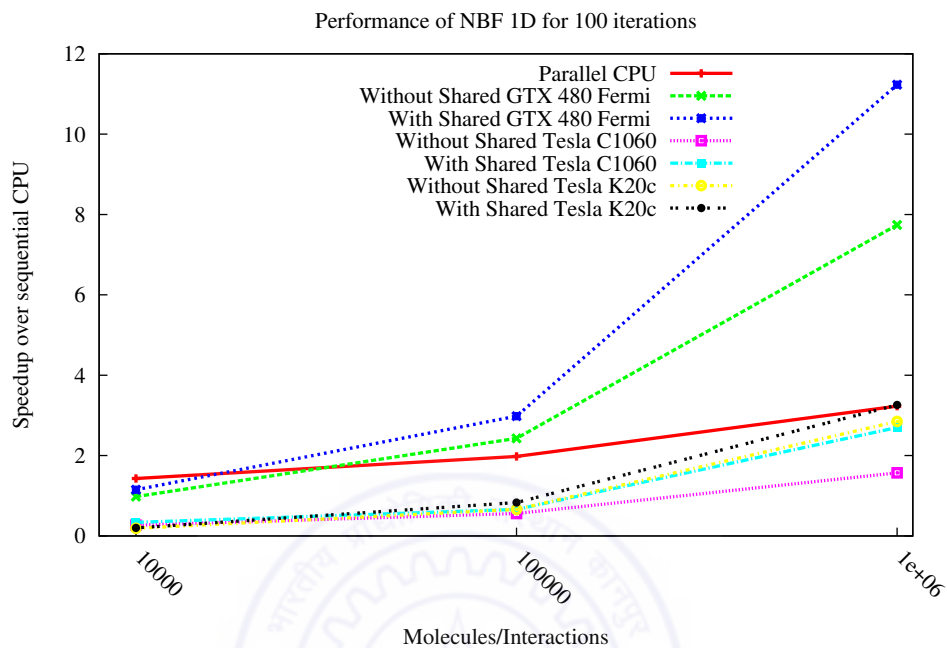


(a) Speedup of MOLDYN 1D

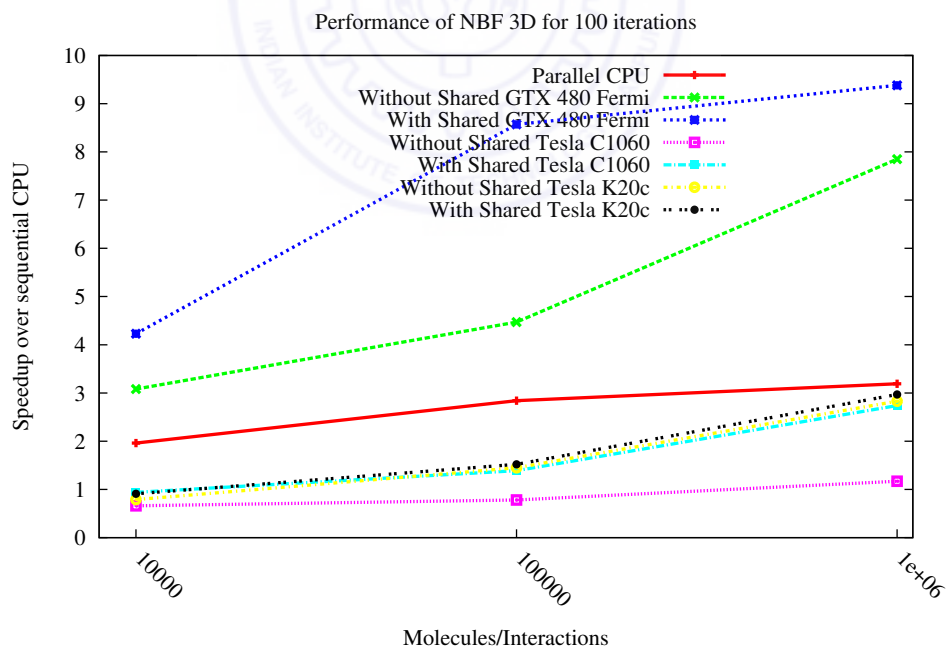


(b) Speedup of MOLDYN 3D

Figure 6.9: Performance of MOLDYN kernel



(a) Speedup of NBF 1D



(b) Speedup of NBF 3D

Figure 6.10: Performance of NBF kernel

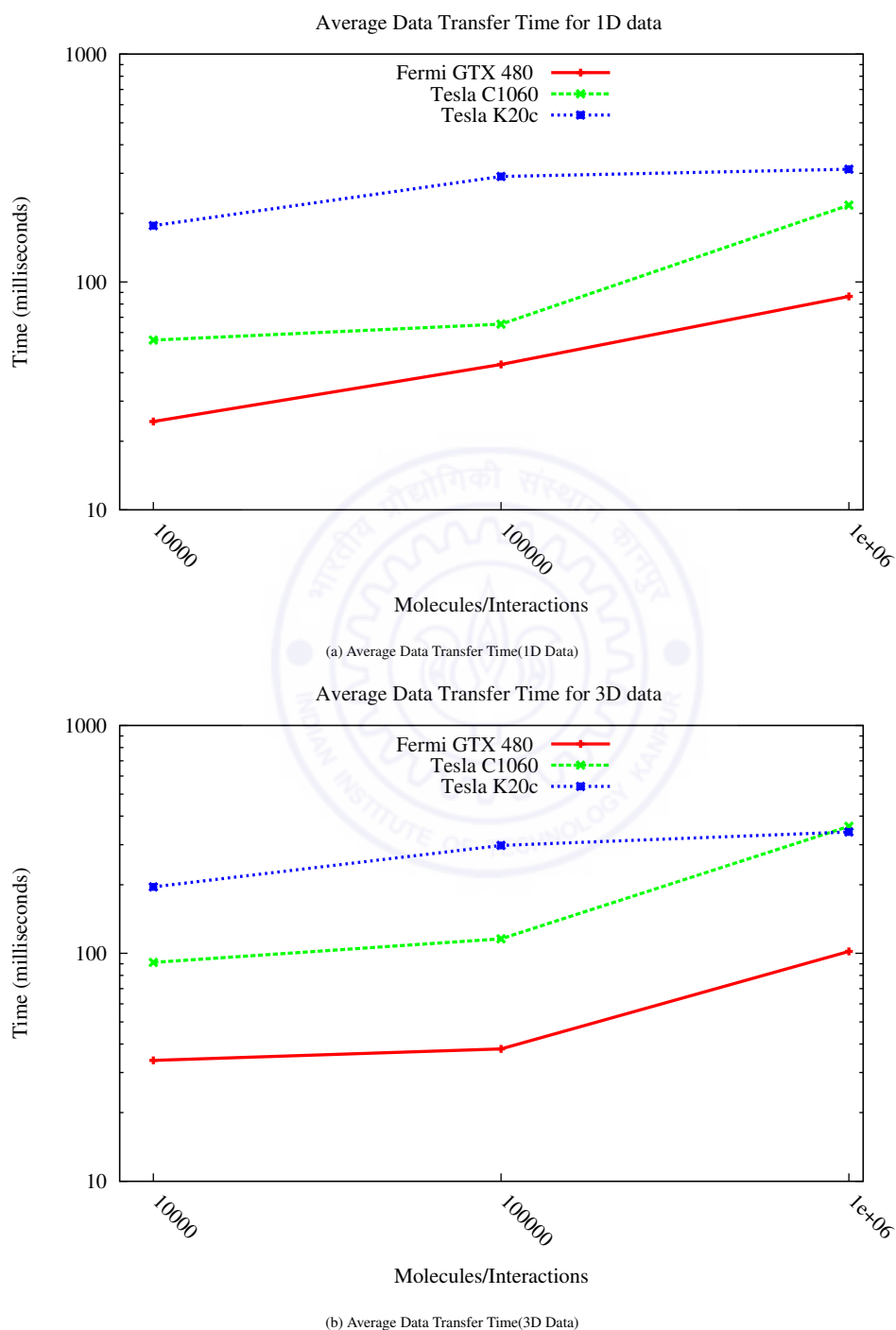


Figure 6.11: Data Transfer Overhead

Chapter 7

Conclusion and Future Work

We presented an annotation based compiler framework which generates CUDA code for GPU. The results show that our framework yields performance within an acceptable range of hand optimized CUDA programs. The annotation based language support is simple enough for a naive programmer to get acquainted with writing code for CUDA enabled GPU. We also investigate improving compiler and run-time support for irregular reductions on GPU. Experimental results indicate that our framework can help domain experts to achieve good performance using GPUs without knowing the details of the architecture and programming intricacies. Improvements provided by JolokiaC++ in run-time data and iteration reordering significantly improve the memory system performance of irregular applications due to improved spatial and temporal locality for the GPU.

We have developed a prototype framework for JolokiaC++ and used it to evaluate the performance on standard benchmarks. While the initial results are promising, the implementation requires work in many aspects to become a finished product. Some of the areas for future investigations are as follows:

- Our current implementation is limited to providing support to two-dimensional array through linearized access using *aview* operator. We would like to extend our framework to provide full-fledged support for using multi-dimensional array.

- Our current implementation does not support jolokia pragmas inside conditional statements (switch-case, if-then-else statement). We would extend the support for it in our language.
- In absence of GPU, our framework generates a sequential code for CPU. We would also like to extend it for generating an efficient parallel code for CPU and other architectures like FPGA, Intel Xeon PHI.
- In practice, many loop nests are imperfectly-nested and existing compilers use heuristics that find a sequence of transformations that convert such loop nests into perfectly-nested ones. These heuristics do not always succeed. We would like to work on designing approaches for tiling imperfectly-nested loop nests. Our current implementation for such loops is restricted to optimizing stencil code using texture fetch. In future, we plan to implement temporal blocking using shared access for better performance.
- Our current implementation for irregular applications is limited to accesses with two level of indirection. In future, we would like work on developing runtime support to handle other categories of irregular applications .

Appendices



Appendix A

Annotation Grammar Specification

The production rules for the annotation based language, JolokiaC++ is given below. The annotation grammar is used to parse the annotations present in annotation file which in turn is used to parse the source code given by the user.

Annotations	→	Annotation
		Annotation ; Annotations
Annotation	→	class Class_anns
		operator Operator_ann
Class_anns	→	Class_name { Class_ann }
Class_ann	→	Class_ann1
		Class_ann1 Class_ann
Class_ann1	→	is_scalar { Scalar };
		is_array { Array };
Array	→	array_view { Array_def };
		array_opt { define { Statements } Array_def };
		array_has_value { Value_def };
Scalar	→	scalar_define { Scalar_attribute };
Operator_ann	→	modify { Var_list }
		read { Var_list }
		on_entry (Name)
		on_exit (Name)
		dcopy (Name)
		dalloc (Name)
		release (Name)
		alias { Var_list }

		allow_alias { Var_list }
		modify_array (Name) { Array_def }
Array_def	→	Array_attributes
		Array_attributes Array_def
Array_attributes	→	Array_attribute = Expression ;
Array_attribute	→	dim
		length (Param)
		elem (Param_list)
Scalar_attribute	→	slem



Bibliography

- [1] MARYLAND: CHAOS Library. http://www.cs.umd.edu/projects/hpsl/compilers/base_chaos.html. Retried: Dec 2012.
- [2] Nvidia CUDA. <http://www.nvidia.com>. Retrieved on: June 2013.
- [3] NVIDIA CUDA Programming Guide,Version 2.3.1. http://developer.download.nvidia.com/compute/cuda/2__3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf. Retrieved on: Jan 2013.
- [4] OpenACC: Directives for Accelerators. <http://www.openacc-standard.org/>. Retieved on: Dec 2013.
- [5] OpenMP Application Program Interface. <http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf>. Retrieved on: Sep 2014.
- [6] OpenMP Application Program Interface, Examples. <http://openmp.org/mp-documents/OpenMP4.0.0.Examples.pdf>. Retrieved on: Sep 2014.
- [7] PARSEC Benchmark Suite. <http://parsec.cs.princeton.edu/index.htm>. Retieved on: Dec 2013.
- [8] PIPS: Automatic Parallelizer and Code Transformation Framework. <http://pips4u.org/>. Retieved on: Jan 2014.

- [9] PolyBench/GPU: Implementation of PolyBench codes for GPU processing. Retrieved on: Dec 2013.
- [10] The OpenACC Application Programming Interface version 2.0. http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf. Retrieved on: Dec 2013.
- [11] The OpenCL Specification version 2.0. <http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf>. Retrieved on: Dec 2013.
- [12] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publisher.
- [13] J. Anantpur and R. Govindarajan. Runtime dependence computation and execution of loops on heterogeneous systems. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–10, Feb 2013.
- [14] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, pages 851–862, 2009.
- [15] Muthu M. Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In *In Proceedings of the 22nd annual international conference on Supercomputing*, pages 225–234, New York, NY, USA, 2008. ACM.
- [16] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic Data Movement and Computation Mapping for Multi-level Parallel Architectures with Explicitly Managed Memories. In *PPoPP*, pages 1–10, New York, NY, USA, 2008. ACM.

- [17] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In *CC*, pages 244–263, 2010.
- [18] Pramod K. Bhatotia, Sanjeev K. Aggarwal, and Mainak Chaudhuri. A Compilation Framework for Irregular Memory Accesses on the Cell Broadband Engine, 2009.
- [19] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.
- [20] Uday Bondhugula, J. Ramanujam, and et al. PLuTo: A practical and fully automatic polyhedral program optimization system. In *IN: PROCEEDINGS OF THE ACM SIGPLAN 2008 CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION (PLDI 08)*, 2008.
- [21] Pierre Boulet, Alain Darte, Georges-André Silber, and Frédéric Vivien. Loop Parallelization Algorithms: From Parallelism Extraction to Code Generation. *Parallel Comput.*, 24(3-4):421–444, May 1998.
- [22] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations. *J. Comp. Chem.*, 4:187–217, 1983.
- [23] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: Accelerating Compute-Intensive Applications with Accelerators . Retieved on: Dec 2013.
- [24] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.

- [25] Michał Cierniak and Wei Li. Unifying Data and Control Transformations for Distributed Shared-memory Machines. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation, PLDI '95*, pages 205–217, New York, NY, USA, 1995. ACM.
- [26] P. Colella, D. T. Graves, J. N. Johnson, H. S. Johansen, N. D. Keen, T. J. Ligocki, D. F. Martin, P. W. McCorquodale, D. Modiano, P. O. Schwartz, T. D. Sternberg, and B. Van Straalen. Chombo Software Package for AMR Applications Design Document. Retrieved on: Dec 2013.
- [27] R. Das, P. Havlak, J. Saltz, and K. Kennedy. Index Array Flattening Through Program Transformation. In *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*, pages 70–70, 1995.
- [28] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *J. Parallel Distrib. Comput.*, 22:462–478, September 1994.
- [29] Mara J. Martn David E. Singh and Francisco F. Rivera. Automatic Generation of Optimized Parallel Codes for N-body Simulations, 2004.
- [30] Peng Di, Ding Ye, Yu Su, Yulei Sui, and Jingling Xue. Automatic Parallelization of Tiled Loop Nests with Enhanced Fine-Grained Parallelism on GPUs. In *Proceedings of the 2012 41st International Conference on Parallel Processing, ICPP '12*, pages 350–359, Washington, DC, USA, 2012. IEEE Computer Society.
- [31] Chen Ding and Ken Kennedy. Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run time. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 229–241, 1999.

- [32] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem: I. One-dimensional Time. *Int. J. Parallel Program.*, 21(5):313–348, October 1992.
- [33] Min Feng, Rajiv Gupta, and Laxmi N. Bhuyan. Speculative parallelization on gpgpus. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 293–294, New York, NY, USA, 2012. ACM.
- [34] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: An Expressive Annotation-directed Dynamic Compiler for C. *Theor. Comput. Sci.*, 248(1-2):147–199, October 2000.
- [35] Kronos Group. The Open Standard for Parallel Programming of Heterogeneous Systems. <http://www.khronos.org/opencl/>. Retrieved on: Dec 2013.
- [36] Manish Gupta and Rahul Nim. Techniques for Speculative Run-Time Parallelization of Loops. In *In Supercomputing 98*, 1998.
- [37] Samuel Z. Guyer and Calvin Lin. An Annotation Language for Optimizing Software Libraries. In *Domain-specific languages*, DSL '99, pages 39–52, New York, NY, USA, 1999. ACM.
- [38] Reinhard von Hanxleden, Ken Kennedy, Charles Koelbel, Raja Das, and Joel H. Saltz. Compiler Analysis for Irregular Problems in Fortran D. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 97–111, London, UK, UK, 1993. Springer-Verlag.
- [39] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic CPU-GPU Communication Management and Optimization. *SIGPLAN Not.*, 46(6):142–151, June 2011.

- [40] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):105–118, 2011.
- [41] Henry Kasim, Verdi March, Rita Zhang, and Simon See. Survey on Parallel Programming Model. In *Proceedings of the IFIP International Conference on Network and Parallel Computing*, NPC '08, pages 266–275, Berlin, Heidelberg, 2008. Springer-Verlag.
- [42] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. SD3: A Scalable Approach to Dynamic Data-Dependence Profiling. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 535–546, Washington, DC, USA, 2010. IEEE Computer Society.
- [43] Alan LaMielle and Michelle Strout. Enabling Code Generation within the Sparse Polyhedral Framework. Technical report, Computer Science Department, Colorado State University, Fort Collins, CO 80523-1873, March 2010.
- [44] 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*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.
- [45] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '09, pages 101–110, New York, NY, USA, 2009. ACM.
- [46] Wei Li and Keshav Pingali. A Singular Loop Transformation Framework Based on Non-singular Matrices. *International Journal of Parallel Programming*, 1992.

- [47] Deepak Majeti, Rajkishore Barik, Jisheng Zhao, Max Grossman, and Vivek Sarkar. Compiler-driven data layout transformation for heterogeneous platforms. In Dieter Mey, Michael Alexander, Paolo Bientinesi, Mario Cannataro, Carsten Clauss, Alexandru Costan, Gabor Kecskemeti, Christine Morin, Laura Ricci, Julio Sahuquillo, Martin Schulz, Vittorio Scarano, StephenL. Scott, and Josef Weidenborfer, editors, *Euro-Par 2013: Parallel Processing Workshops*, volume 8374 of *Lecture Notes in Computer Science*, pages 188–197. Springer Berlin Heidelberg, 2014.
- [48] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings. *Int. J. Parallel Program.*, 29:217–247, June 2001.
- [49] Jiayuan Meng, Vitali A. Morozov, Venkatram Vishwanath, and Kalyan Kumaran. Dataflow-driven GPU Performance Projection for Multi-kernel Transformations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC ’12, pages 82:1–82:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.
- [50] Ravi Mirchandaney, Joel H. Saltz, Joel H. Saltz, Doug Baxter, and Doug Baxter. Run-Time Parallelization and Scheduling of Loops. *IEEE Transactions on Computers*, 40, 1991.
- [51] R. Nasre, M. Burtcher, and K. Pingali. Data-Driven Versus Topology-driven Irregular Computations on GPUs. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 463–474, May 2013.
- [52] Cosmin E. Oancea and Lawrence Rauchwerger. Logical Inference Techniques for Loop Parallelization. In *PLDI, PLDI ’12*, pages 509–520, New York, NY, USA, 2012. ACM.

- [53] Matthew Papakipos. Peakstream many-core computing platform. <http://www.stanford.edu/class/ee380/Abstracts/070926-PeakStream.pdf>. Retrieved on: June 2013.
- [54] William Pugh and David Wonnacott. Nonlinear Array Dependence Analysis, 1991.
- [55] D. Quinlan, M. Schordan, R. Vuduc, and Qing Yi. Annotating user-defined abstractions for optimization. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8 pp.–, April 2006.
- [56] Daniel Quinlan, Markus Schordan, Richard Vuduc, Qing Yi, Thomas Panas, Chunhua Liao, and Jeremiah J. Willcock. ROSE Compiler Infrastructure. <http://rosecompiler.org>. Retrieved on: Jan 2012.
- [57] Daniel J. Quinlan. ROSE: Compiler Support for Object-Oriented Frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.
- [58] RapidMind Inc. Writing Applications for the GPU Using the RapidMindTM Development Platform. <http://www.cs.ucla.edu/~palsberg/course/cs239/papers/rapidmind.pdf>.
- [59] L. Rauchwerger and D.A. Padua. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. *Parallel and Distributed Systems, IEEE Transactions on*, 10(2):160–180, feb 1999.
- [60] Lawrence Rauchwerger, Nancy M. Amato, and David A. Padua. A Scalable Method for Run-Time Loop Parallelization. *IJPP*, 26:26–6, 1995.
- [61] Lawrence Rauchwerger, Nancy M. Amato, and David A. Padua. Run-time methods for parallelizing partially parallel loops. In *Proceedings of the 9th international conference on Supercomputing*, pages 137–146, 1995.

- [62] Lawrence Rauchwerger and David Padua. The Privatizing DOALL Test: Run-Time Technique for DOALL Loop Identification and Array Privatization. In *IN PROCEEDINGS OF THE 1994 INTERNATIONAL CONFERENCE ON SUPERCOMPUTING*, pages 33–43, 1994.
- [63] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. In *Proceedings of the 16th international conference on Supercomputing*, ICS '02, pages 274–284, New York, NY, USA, 2002. ACM.
- [64] Amit Sabne, Putt Sakdhnagool, and Rudolf Eigenmann. Effects of compiler optimizations in openmp to cuda translation. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*, IWOMP'12, pages 169–181, Berlin, Heidelberg, 2012. Springer-Verlag.
- [65] Joel Saltz, Kathleen Crowley, Ravi Mirchandaney, and Harry Berryman. Run-time Scheduling and Execution of Loops on Message Passing Machines. *J. Parallel Distrib. Comput.*, 8(4):303–312, April 1990.
- [66] Mehrzad Samadi, Amir Hormati, Janghaeng Lee, and Scott Mahlke. Paragon: Collaborative Speculative Loop Execution on GPU and CPU. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 64–73, New York, NY, USA, 2012. ACM.
- [67] Mehrzad Samadi, Amir Hormati, Mojtaba Mehrara, Janghaeng Lee, and Scott Mahlke. Adaptive Input-Aware Compilation for Graphics Engines. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 13–22, New York, NY, USA, 2012. ACM.
- [68] Vivek Sarkar and Radhika Thekkath. A General Framework for Iteration-Reordering Loop Transformations (Technical Summary). In *In Proceedings of the*

- ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 175–187, 1992.
- [69] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI '03*, pages 91–102, New York, NY, USA, 2003. ACM.
- [70] Sumesh Udayakumaran and Rajeev Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 276–286, 2003.
- [71] Sain-Zee Ueng, Melvin Lathara, Sara S. Bagsorkhi, and Wen-Mei W. Hwu. Languages and Compilers for Parallel Computing. chapter CUDA-Lite: Reducing GPU Programming Complexity, pages 1–15. Springer-Verlag, Berlin, Heidelberg, 2008.
- [72] Manuel Ujaldon and Joel Saltz. The GPU on irregular computing: Performance issues and contributions. *Computer Aided Design and Computer Graphics, International Conference on*, pages 442–450, 2005.
- [73] Didem Unat, Xing Cai, and Scott B. Baden. Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C. In *Proceedings of the International Conference on Supercomputing, ICS '11*, pages 214–224, New York, NY, USA, 2011. ACM.
- [74] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, January 2013.

- [75] Shucaï Xiao and Wu chun Feng. Inter-block GPU communication via fast barrier synchronization. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.
- [76] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A GPGPU compiler for memory optimization and parallelism management. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation, PLDI '10*, pages 86–97, New York, NY, USA, 2010. ACM.
- [77] Qing Yi, Vikram Adve, and Ken Kennedy. Transforming loops to recursion for multi-level memory hierarchies. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, PLDI '00*, pages 169–181, New York, NY, USA, 2000. ACM.
- [78] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly Elimination of Dynamic Irregularities for GPU Computing. *SIGARCH Comput. Archit. News*, 39(1):369–380, March 2011.



Publications

Selected/Communicated International Journal Articles

1. **Vibha Patel**, Sanjeev K Aggarwal, “A Compiler Framework for Optimization of Irregular Applications on GPGPUs” partially accepted in IET Software Journal.

Selected/Communicated International Paper/Poster

1. **Patel Vibha**, Sanjeev Aggarwal and Amey Karkare, “JolokiaC++ : A Annotation based Compiler Framework for GPGPUs”, accepted for publication in the Sixteenth IEEE Conference on High Performance Computing and Communications (HPCC -2014).
2. Bhavin Patel, **Patel Vibha**, “GPU based Space Partitioning with Image Recognition” accepted for publication in the Seventh International Conference on Contemporary Computing (IC3 - 2014).
3. **Patel Vibha**, Bhavin Patel, “Indexing SURF Features By SVD Based Basis On GPU With Multi-Query Support” accepted for publication in Proceedings of 10th International Conference on Intelligent Computing (ICIC 2014), LNAI, Springer Verlag, Taiyuan, China, August 2014
4. **Patel Vibha**, Sanjeev K Aggarwal, “ArCUDA: An ArBB to CUDA Translator” International Supercomputing Conference 2013, Leipzig, Germany, June 2013
5. Monika Shah, **Vibha Patel**, “ An Efficient Sparse Matrix Multiplication for Skewed Matrix on GPU” 14th IEEE International Conference on High Performance Computing and Communication & 9th IEEE International Conference on Embedded Software and Systems, HPCC-ICISS 2012, Liverpool, United Kingdom, June 25-27, 2012. HPCC-ICISS 2012: 1301-1306