Multidimensional Parallelization for Streaming Text Processing Applications Based on Parabix Framework

by

Dan Lin

M.Sc., Simon Fraser University, 2010
B.Sc., Beihang Univeristy, 2006

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Fall 2017

Approval

Name:	**Dan Lin**
Degree:	**Doctor of Philosophy**
Title:	***Multidimensional Parallelization for Streaming Text Processing Applications Based on Parabix Framework***
Examining Committee:	**Chair:** Keval Vora Assistant Professor

Arrvindh Shriraman
Senior Supervisor
Associate Professor _____

Robert Cameron
Co-Supervisor
Professor _____

Thomas Shermer
Supervisor
Professor _____

Dr. Nick Sumner
Internal Examiner
Assistant Professor _____

External Examiner
External Examiner
Professor
Department of Computing Science
University of Western Ontario _____

Date Defended: Dec 15 2017

Abstract

Streaming text processing is important for transforming and analyzing the rapidly growing data in modern society. Unfortunately, text processing software written using the sequential byte-at-a-time processing model fails to take full advantage of the resources available on modern processors for many reasons, including significant branch misprediction penalties due to the input-dependent branching structures of text processing applications, cache miss penalties for table-based operations applied per byte of input, and logical complexity that makes it difficult to process more than a single-byte at a time. Common solutions to process text streams that have dependencies from start to end may involve state machine or recursive algorithms, which are generally considered hard to parallelize and hence ill-suited for multicore or manycore processors. However, the Parabix approach to text processing has recently been shown to offer a promising alternative, based on the concept of bitwise data parallelism: a transform representation of text that uses the full width of available processor registers at a density of one bit per input byte. This dissertation investigates the further development of the Parabix framework to incorporate multidimensional parallelization, combining Parabix methods with several different models of multithreading such as task parallelism, data parallelism and pipeline parallelism as well as with GPU-based SIMT processing. A form of data-pipeline parallelism is developed and shown to be beneficial for text-processing applications even with strong sequential state dependencies. Compilers for both pure pipeline parallelism and data-pipeline parallelism are developed and integrated into Parabix framework to provide automated multithreading support for any Parabix applications. Methods for task parallelism and data parallelism are also developed, but need to be customized by the programmer for specific applications. GPU support is added to Parabix framework by translating LLVM IR into PTX, which can be compiled into binary code and run on GPU devices. Programmers can simply choose to use NVPTX driver instead of CPU driver for code that is executed on GPU. Several applications based on Parabix are implemented and tested with different parallelization techniques to analyze the advantages and limits of multidimensional parallelization extensions of Parabix framework. In data-pipeline mode, we are able to achieve 215% speedup compared with the sequential version on a quad-core machine. The GPU implementation has its limitations but can give up to 310% speed-up.

Keywords: text processing; parallel programming; dynamic compilation; SIMD; GPU; LLVM

Acknowledgements

I am very grateful to my supervisors Arrvindh Shriraman and Robert Cameron. They gave me lots of support for my Ph.D. research. I would also like to thank my uncle for helping me preparing my thesis defense. This paper is dedicated to my mother who always loves me without any condition.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Modern information societies are defined by large amounts of data mostly represented in text form. Applications built on gathering, filtering, searching or analyzing such data become more and more important.

Many commercial companies are facing problems caused by the rapidly growing data. Google grew from processing 100 terabytes of data a day in 2004 to processing 20 petabytes a day by [32]. Facebook stores, accesses and analyzes more than 30 petabytes of user-generated data; the size is continually increasing since there are about 100 terabytes of data uploaded to Facebook every day [1]. Roughly 6,000 tweets are tweeted on Twitter every second which corresponds to over 500 million tweets per day [3].

Beyond commercial needs, scientific breakthroughs are asking for more powerful and advanced computing capabilities in order to deal with massive datasets. For example, the research of DNA sequencing has created a large amount of text data. Billions of sequence reads with hundreds of bases per read are often produced in a single instrument run [14]. Efficient sequence alignment of all these reads to a reference genome is in great need.

As performance challenges and energy consumption have become a big concern in both the scientific world and real business, researchers have put more effort into improving the processing speed and reducing the energy cost. Many parallelization techniques have been developed to take advantage of the continually evolving hardware designs. Some text processing applications allow simple data division for parallel processing while others may require sequential processing algorithms due to the data dependencies. This dissertation focuses on streaming text processing applications, which we define using the following properties.

- Data dependencies from start to end. Every chunk of data is potentially dependent on its previous chunk. Breakpoints in the dependencies occur irregularly; identifying them generally requires preprocessing.

- Large volumes of data. The rapidly growing data is a trend and becomes a bottleneck in many streaming text processing applications.
- High performance expectations. Efficiency is a primary concern in streaming text processing applications. Sometimes there could be a realtime constraint.

Large volumes of data and high performance expectations motivates the studies of parallelization strategies for streaming text processing applications. A direct instinct might be using cloud computing or computer clustering to process large scale of data. However, the data dependencies of streaming text processing increase the difficulty for distributing the data. For example, the conventional method of implementing regular expression matching uses non-deterministic or deterministic automata that are inherently sequential. Parallelization of automata has been explored by previous studies [22, 35, 27], but often has a large overhead due to the speculative simulation. Sequence alignment is also a hard-to-parallelize problem since it is naturally defined by a recursive algorithm. XML processing is another example that cannot be parallelized by a naive data partition because of the tree-oriented representation.

To address these performance challenges on a single machine, our prior work developed Parabix (Parallel Bit Streams), a new transform representation of text that allows programmers to implement text streaming applications in parallel with SIMD registers [8, 10, 7, 13]. Parabix methods are fundamentally oriented toward single core processing; it is the purpose of this dissertation to investigate the multithreading and GPU support of Parabix applications.

However, programming Parabix applications can be time-consuming and error-prone. Implementing Parabix directly in C/C++ requires the use of SIMD intrinsic libraries and manual processing of the carry bits that are used to address sequential dependencies [8, 10]. The concept of unbounded bit streams was developed to ease the programmers' burden in carry processing [7]. Using arbitrary size integers to represent bit streams, Parabix prototypes can be written in Python. To compile these prototypes to use SIMD instructions, we developed the Parabix tool chain, which consists of two compilers and runtime libraries to implement text streaming applications [31]. Initial work in developing multithreaded Parabix applications was carried out with this tool chain, but was a complex manual process. Programming on GPU was particularly difficult because the code had to be translated to CUDA/OpenCL by programmers.

The work of this dissertation is based on the newer Parabix framework, which takes advantage of the LLVM infrastructure to provide the dynamic compilation of Parabix code [12]. This has been combined with a kernel-based programming model to improve programmer productivity and program robustness. However, this increases the difficulty of combining multithreading techniques or translation to GPU code because the programs built by Parabix framework are eventually represented as LLVM IR before execution. Therefore,

programmers have to know how to create and synchronize the threads in LLVM IR to get a better utilization of multicore machines, and in order to execute the code on GPU device, they also need to understand the conventions and intrinsics of NVPTX back-end for LLVM.

1.2 Contribution

The contributions of this dissertation are summarized as follows.

- We provide multithreading support for Parabix framework, which not only includes two integrated modes, pipeline parallel, and data-pipeline parallel, but also allows programmers to have customized modes, e.g., task parallel and data parallel. The pipeline mode and data-pipeline mode are automated by compilers.
- Applications implemented using Parabix framework can also be executed on GPU. We present the limitations of the GPU extension of Parabix framework and a solution to better utilize GPU resources in one of our applications.
- We implemented multithreaded and GPU versions of several streaming text processing applications using Parabix framework such as regular expression matching, approximate string matching, UTF8 to UTF16 transcoding and base64 encoder. We also provide performance analysis of those applications using different parallelization modes.

1.3 Organization of this Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 provides background material reviewing the existing parallelization techniques and introduces Parabix tool chain and Parabix framework. Chapter 3 describes how Parabix is combined with multithreading techniques and GPU architecture using Parabix tool chain. Chapter 4 describes multithreading support for Parabix framework. Chapter 5 describes GPU extension of Parabix framework and its limitation. Chapter 6 presents the performance results and analysis of several applications that are implemented using Parabix framework and measured with different parallelization methods. Chapter 7 concludes this dissertation.

Chapter 2

Background

2.1 Parallelization Techniques

Parabix is a bit-level parallelization technique that does not rely on any hardware design but can naturally gain performance benefit by using SIMD ISA. We can achieve a further performance improvement of Parabix applications by taking advantage of multicore and GPU parallelization. In this section, we present a few well-known hardware designs that aim to help parallel programming including SIMD, multicores, GPU as well as FPGA. Current Parabix does not apply to FPGA, but there could be a potential benefit with a specialized hardware design.

Parabix framework is a parallel programming model that designed for applying Parabix techniques to improve the performance of streaming text processing applications on a single machine. We introduce several existing parallel programming models in this section. Some of them may design for a different purpose. For example, MapReduce and SYMPLE mainly targets cloud computing or computer clustering.

At the end of this section, we discuss some hard-to-parallelize applications that cannot be easily programmed by the existing models with mainstream parallelization hardware designs but can be implemented using Parabix framework to achieve a substantial performance improvement.

2.1.1 Hardware Parallelization

There exist many techniques that target specific hardware designs. Besides the mainstream multicores, manycore accelerators (e.g., Intel Xeon Phi) and specialized accelerators (e.g., GPU) as well as SIMD instruction set extension also attract extensive attention from software developers that aim at high-performance parallel programming.

Intel Xeon Phi is based on x86 architecture but offers a much larger number of cores than conventional CPUs. An advantage of Intel Xeon Phi is that it does not require processor specific programming languages, which may affect the developer productivity and increase

the maintenance costs. A number of studies have shown performance gains using popular parallel programming models such as OpenMP [42] [48], MapReduce [33].

Streaming SIMD Extensions (SSE) is designed by Intel and first introduced in 1999. SSE offers additional 128-bit registers and provides SIMD instructions that can perform the same operations on multiple data objects. Advanced Vector Extensions (AVX) is then proposed in 2008 with larger register width of 256 bits. Typical SIMD applications include graphics, signal processing, and scientific research. However, using SIMD to parallelize programs that rely on irregular data structures has also been explored and an intermediate language has been developed to implement SIMD parallelized programs [45].

GPU is composed of tens of streaming multiprocessors (SMs), and each SM may contain hundreds of thread processors. This architecture offers extensive compute units and provides massive parallelism. The value of GPU for general purpose computing has been widely recognized as many studies have mapped traditional CPU domain problems to GPU architectures and achieved performance improvement [16]. Critical algorithms (e.g FFT [39]), database operations [21] and a variety of other applications have achieved dramatic speedups with GPUs. In order to fully use the resources, a programming system called Qilin is proposed to distribute works over the CPU and GPU [36]. The heterogeneous system composed of CPU and GPU is explored by other studies as well [58] [17].

Another hardware design that could be used for parallel processing is called field-programmable gate array (FPGA). FPGA is an integrated circuit that allows programmers to design their own logic to speed up a specific algorithm or application. FPGAs have been shown performance benefit in XML parsing [19] [50], regular expression matching [51], DNA sequence alignment [53] and many other applications. A disadvantage of FPGA is the programming difficulty caused by the complexity of fine-grained implementations [24].

2.1.2 Parallel Programming Models

Multi-core techniques are widely applied to nowadays processors. Thus, many parallel programming models and languages/APIs are developed to help creating parallel programs in a quick and efficient manner. This section will introduce several existing models.

OpenMP is a shared-memory multiprocessing programming interface that extends existing languages with a set of directives [46]. Users need to define a parallel region and private variables for this region. The parallel region will be executed by multiple threads using the specified private variables to tell whether a variable should be shared among threads. As Graphics Processing Unit (GPU) and Digital Signal Processor (DPS) become popular in mainstream computers, OpenMP 4.0 Release Candidate 2 provides an extension to support this trend of a combined use of multicore processors and specialized accelerators.[30]

MapReduce is a parallel and distributed programming model that aims for large-scale data-intensive applications. Applications using this model are organized into pairs of Map and Reduce functions. The Map function processes the partitioned input data in parallel

since the data chunks are independent of each other. Key-value pairs are generated during the process so that the Reduce function can merge the pairs that have the same key. A programming API based on MapReduce model called Phoenix is proposed to dynamically schedule the tasks and manage the locality as well as handle the fault tolerance across processor nodes [43].

SYMPLE outperforms MapReduce for User-Defined Aggregations that involve dependencies between data segments [44]. SYMPLE treats the unresolved dependencies as symbolic values and returns a symbolic summary that represents its output as a function of those symbolic values. The summaries for each data segments can be computed by mappers in parallel. A reducer collects the summaries and composes them in the right order to produce the output that matches a sequential execution.

Threading Building Blocks (TBB) is parallel run-time libraries based on the C++ language [18]. In TBB applications, rather than data partitioning, a computation is broken down into tasks and dynamically managed by TBB task scheduler. TBB improves the performance of imbalanced workload by a task stealing algorithm, which attempts to keep threads busy and maximize concurrency.

StreamIt is a parallel programming language that aims to improve the performance of streaming applications [55]. It introduces an important concept called filter as a base unit of computations in StreamIt. StreamIt provides three interconnection types for composing filters into larger stream graphs. The Pipeline is used for sequential task combinations; the SplitJoin is used to handle independent tasks that can be processed concurrently and the FeedbackLoop provides a way to create cycles in the task graph. Based on StreamIt, there is also a compiler system that has been developed to further exploit task, data, and pipeline parallelism and leverages the right combination to achieve good performance [20].

OpenMP is mostly used in programs where the parallelism can be easily found and defined. TBB may generate more code overheads than OpenMP, but is useful for not-well-balanced workloads, nested parallelism, and complex flow graph structures. MapReduce is a good choice for clusters and distributed applications but not good at computational-intensive and iterative computation problems [25].

2.1.3 Hard-to-parallelize Applications

Some text processing applications cannot benefit from traditional techniques and requires special treatment such as speculative partitioning, overlapping and possibly associated with a lightweight pre-processing/post-processing phase. In this section, we focus on three hard-to-parallelize applications, XML parsing, regular expression matching, and approximate string matching.

XML is one of the most widely used standards for information representation and interchange. XML processing in large volume becomes a performance bottleneck in applications such as web services, databases, and other software systems. The fundamental XML pro-

cessing task, XML parsing is exploited for accelerations in many studies. XML Screamer is an efficient XML parser that applies a schema-based compilation to minimize data copying and transformation [28]. Tang et al. use prefetching to reduce the memory accesses, which contribute more than 60% of the total execution cycles [54]. Hardware approaches based on FPGA have shown significant speedup that can achieve a throughput of 1 cycle per byte [19] and 1.7 to 3.1 Gbps [50]. There are also researches that target commodity processors with multi-core systems. A parallel approach for XML parsing proposes a preparsing phase that helps to determine the tree structure and then do the data partition according to the pre-parsing results [34]. Another approach for parallelizing XML parsing process involved a speculatively partitioning and linking the partial node tree structures to a complete node tree [49].

Regular expression matching is a core algorithm used in many applications such as network intrusion detection systems, virus scanners, protein sequence matching (e.g. PROSITE database [23]) and so on. A typical solution for regular expression matching is to use nondeterministic or deterministic automata. However, the computation of an automaton is executed in sequential and thus hard to parallelize. A simultaneous finite automaton (SFA) is proposed that extends automata with a speculative simulation from all the states so that it can be divided at any points and computed in parallel [52]. By applying SFA, regular expression matching can be processed in parallel without overheads but suffers size explosion with large rule sets. SIMD and GPU technologies have also been exploited to implement NFA-based regular expression matching with parallelism [59] [45].

Approximate string matching is used in sequence alignment for DNA, RNA or proteins. It is a time-consuming process as this problem commonly facing a large scale dataset. Myers proposed a bit-vector algorithm that computes the dynamic programming matrix of multiple entries at a time using bit-wise operations [40]. Based on Myers' idea, Chacon et al. developed an approach to compute edit distance in a thread-cooperative way and implemented this approach with CUDA on GPU [14]. National Center for Biotechnology Information (NCBI) has developed a sequence alignment tool called BLAST [5]. Several efforts have been made to parallelize BLAST on multicore processors [6] [41]. As more and more bioinformatics problems proved performance benefit on GPU [47] [37] [56], GPU-BLAST has been built on top of the base BLAST. This GPU version can achieve up to 4 times speedup compared to the single-threaded BLAST in protein alignments [57].

2.2 Parabix Tool Chain

2.2.1 Parabix Overview

Parabix (Parallel Bit Streams) is a new transform representation of text. It transforms byte-oriented data to 8 parallel bit streams known as basis bit streams such that each bit stream comprises of one bit from every byte of the data. By applying simple bitwise logic

to the eight basic bit streams, any character set can be represented by a single bit stream, where each bit tells the existence of a member of the character set at the corresponding position. This enables the applications to processes 128 positions at a time with SIMD register commonly found on commodity processors (e.g., SSE on Intel).

As shown in Figure 2.1, Parabix tool chain consists of two compilers and runtime libraries. The first step is to use character class compiler (ccc) to automatically produce bit stream logic for all the predefined character classes needed in a particular application. The next step is to use Pablo compiler to convert unbounded bit streams logic into C/C++ block-at-a-time processing. Our runtime libraries that support a machine-independent view of basic SIMD operations, as well as a set of core function libraries. We have ported Parabix libraries to a wide variety of processor architectures such as 128-bit SSE operations on the previous generation Intel platforms, 256-bit AVX extensions on the latest Intel processor, and the 128-bit Neon operations on ARM.

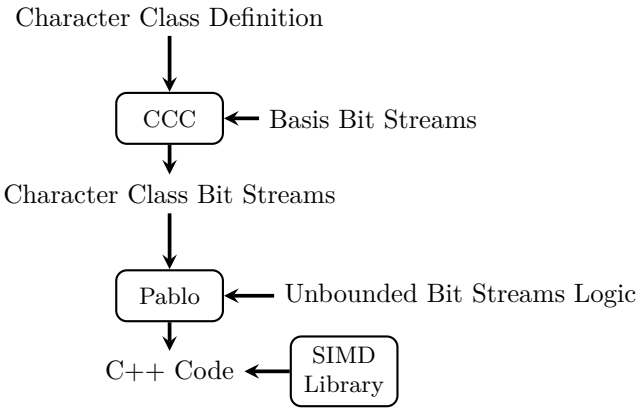

Figure 2.1: Parabix Tool Chain

Many applications have been implemented with Parabix tool chain. Unicode transcoding [8], XML parsing [10, 7, 9] and regular expression matching [13] [12] applications implemented with Parabix have each demonstrated substantial performance improvement as well as energy savings [31].

2.2.2 Character Class Compiler

In the Parabix tool chain, basis bit streams are used as the starting point to construct character-class bit streams in which each 1 bit indicates the presence of a significant character (or class of characters). To construct the basis bit streams, the source data is first loaded in sequential order and then transposed using SIMD pack, shift, and bitwise operations.

As shown in Figure 2.2, the ASCII string "`b7<A>a`" is represented in binary format. By applying transposition, we get 8 basis bit streams, $\mathtt{b}_0, \mathtt{b}_1 \ldots \mathtt{b}_7$ where $\mathtt{b}_0$ represents the first

bit of each character in the string, $\mathtt{b_1}$ represents the second bit of each character in the string, and so forth.

Once the basis bit streams are calculated, we can use a series of bitwise logic operations to calculate character class streams in parallel. For example, a character is an `A` (01000001) if and only if $\neg(b_0 \vee b_2 \wedge b_3 \wedge b_4 \wedge b_5 \wedge b_6) \wedge (b_1 \vee b_7) = 1$. Therefore, by applying this 7 operations to the 8 basis bit streams, we can process 128 positions at a time on SSE. With only 6 positions provided in the example of Figure 2.2, the character class bit stream of `A` is 000100, which marks the fourth position in string "`b7<A>a`".

Ranges of characters sometimes have more advantage than individual characters. For example, the alphabet that includes 52 letters [A-Za-z] can be found with similar formulas with only 11 operations.

The character class compiler is designed to automatically generate the formulas. As shown in Figure 2.3, the input of the character class compiler is a character class definition adapted from the standard regular expression notations. The output is a set of three-address bitwise operations that computes the character class bit streams. The result set is minimized but not guaranteed to be an optimal solution.

STRING	b	7	<	A	>	a
ASCII	01100010	00110111	00111100	01000001	00111110	01100001
$\mathtt{b_0}$	0	0	0	0	0	0
$\mathtt{b_1}$	1	0	0	1	0	1
$\mathtt{b_2}$	1	1	1	0	1	1
$\mathtt{b_3}$	0	1	1	0	1	0
$\mathtt{b_4}$	0	0	1	0	1	0
$\mathtt{b_5}$	0	1	1	0	1	0
$\mathtt{b_6}$	1	1	0	0	1	0
$\mathtt{b_7}$	0	1	0	1	0	1

Figure 2.2: Example of Basis Bit Streams

2.2.3 Pablo Compiler

Once the character class bit streams are computed, we can then write programs by performing Pablo operations including bitwise logic, `Advance`, `ScanThru` and `MatchStar` on these bit streams. Pablo is a language that we defined to operate on unbounded bit streams. The `Advance` operation means to shift the given bit stream one position forward. On little-endian architectures, `Advance` is equaled to shift left by one or adding the operand to itself. The `ScanThru` operator accepts two input parameters, c and m, where c denotes the start positions, and m denotes the marked positions for scanning. The result of the `ScanThru` operation is a bit stream that marks the positions immediately following any run of marker positions. The formula we used to implement `ScanThru` is $(c+m) \wedge \neg m$ [9]. The `MatchStar`

```
 INPUT:  Alpha = [A-Za-z]

OUTPUT:  temp1 = (basis_bits.bit_6 & basis_bits.bit_7)
         temp2 = (basis_bits.bit_5 | temp1)
         temp3 = (basis_bits.bit_4 & temp2)
         temp4 = (basis_bits.bit_4 | basis_bits.bit_5)
         temp5 = (basis_bits.bit_6 | basis_bits.bit_7)
         temp6 = (temp4 | temp5)
         temp7 = ((basis_bits.bit_3 & ~temp3)|(~(basis_bits.bit_3) & temp6))
         temp8 = (basis_bits.bit_1 &~ basis_bits.bit_0)
         Alpha = (temp7 & temp8)
```

Figure 2.3: Character Class Compiler Input/Output

operation is used to find all matches of character class repetitions; it can be computed by $(((m \wedge c) + c) \oplus c) \vee m$ [13].

Figure 2.4 demonstrates a simple application of tag parsing using character class bit streams. We use periods to denote 0 bits so that the 1 bits stand out. In this example, tags start with `<`, end with `>`, and have tag name consists only alphabet characters. If `<` and `>` do not match, an error bit stream E_1 will provide the error positions. To compute the error bit stream, the first step is to apply `Advance` operation to the `LAngle` bit stream and get start positions. The next step is to start with L_0 and apply `ScanThru` to `Alpha` bit stream to locate the expected end positions. If `LAngle` is not found at the expected end positions, a bit will be set at the corresponding position in E_1.

source text			b7<A>ab<err]
Alpha	=	[a-zA-Z]	1..1.11.111.
LAngle	=	[>]	..1....1....
RAngle	=	[<]	....1.......
L_0	=	Advance(LAngle)	...1....1...
L_1	=	ScanThru(L_0, Alpha)	....1......1
E_1	=	$L_1 \wedge \neg$ RAngle	1

Figure 2.4: Tag Parsing in Parabix

In Figure 2.4, it takes only 3 Pablo operations to complete tag parsing. However, on real machines, it is limited by the size of SIMD registers. For example, suppose the register size is 8. `Advance LAngle` can lose one bit at the end. In order to solve this problem, we construct a Carry Queue to hold the carry variables.

Figure 2.5 illustrates the input and output of Pablo compiler. The Pablo code defined in Python is translated into a public C/C++. We chose Python because it supports infinite integers, which can denote the unbounded bit streams. The Carry Queue is declared and

initialized by `CarryDeclare` and `CarryInit`. The unbounded bit stream `Advance` and `ScanThru` operations are translated into block-wise equivalents `BitBlock_advance_ci_co` and `BitBlock_scanthru_ci_co`, which reads the carry from previous block and then set the new carry in Carry Queue for next block. The Pablo compiler also supports conditional and iterative bit stream logic that involves additional carry-test insertion in control branches.

```
INPUT:
def parse_tags(classes):
  L0 = Advance(classes.Langle)
  L1 = ScanThru(L0, classes.Alpha)
  E1 = L1 &~ classes.Rangle

OUTPUT:
struct Parse_tags {
  Parse_tags() { CarryInit(carryQ, 2); }
  void do_block(Classes & classes){
    BitBlock L0, L1, E1;
    L0 = BitBlock_advance_ci_co(classes.Langle, carryQ, 0);
     L1 = BitBlock_scanthru_ci_co(L0, classes.Alpha, carryQ, 1);
    E1 = simd_andc(L1, classes.Rangle);
    CarryQ_Adjust(carryQ, 2);
  }
  CarryDeclare(carryQ, 2);
};
```

Figure 2.5: Pablo Compiler Input/Output

2.3 Parabix Framework

2.3.1 Overview

Parabix tool chain provides a set of tools that help programmers to build Parabix applications. However, it still requires some human effort to implement the application step by step. Therefore, we developed a framework built on LLVM compiler infrastructure, enabled a dynamic compilation of many applications. This framework integrates all of the previous tools and provides a kernel-based programming model.

Kernels are computational abstractions for data processing. A program can be implemented as one or more kernels that are executed in a pipeline. Each kernel has a struct that holds the kernel states that are used to record to the data dependencies between data chunks.

As shown in Figure 2.6, Parabix framework consists of several core modules. Pablo Generator is first proposed in a regular expression matching application called ICgrep [12].

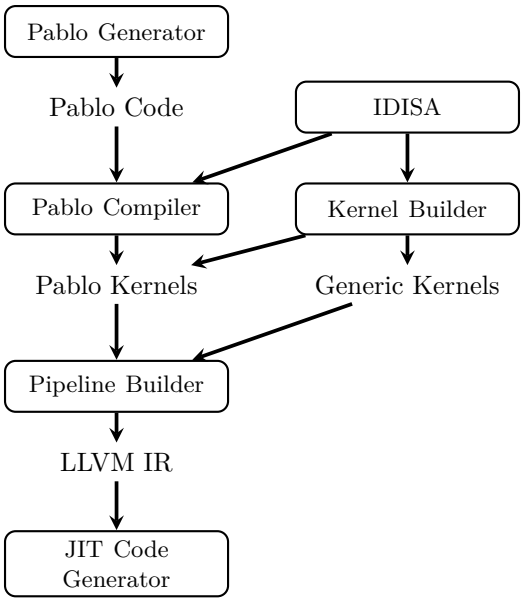

Figure 2.6: Parabix Framework

In ICgrep, Pablo Generator is named as RegEx Compiler, which parses the regular expression and transforms the generated abstract syntax tree (AST) into Pablo operations. There are other Pablo Generators designed for different applications. For example, Appendix A.1 shows the Pablo code generated by Pattern Compiler built for Editd, which is a Parabix application that will be discussed in Section 6.3. Programmers can also write their own Pablo code. Pablo Compiler translates Pablo code into LLVM IR and dynamically generates Pablo Kernels with the support of Kernel Builder. LLVM IR is a low-level code representation in Static Single Assignment (SSA) form, which can be translated into executable code by JIT Code Generator. Kernels that require more than parallel bit stream operations can be created with Kernel Builder directly. Pipeline Builder chains all the kernels and run them against the input data block by block. The block size can be specified by the user or automatically selected by IDISA Builder. Appendix A.6 shows an example of building an application with Parabix framework.

2.3.2 IDISA Builder

IDISA is a uniform programming model that first designed to support inductive doubling algorithms [11]. It provides a general solution for SIMD operations at all power-of-2 field widths up to the full SIMD register width on a target machine. The SIMD Library discussed in Parabix Tool Chain is also constructed based on this model.

As shown in Figure 2.7, IDISA Builder is implemented based on IRBuilder. IRBuilder is an LLVM module that provides a uniform API for creating instructions and inserting them into a basic block. IDISA Builder extends IRBuilder to support a set of core function

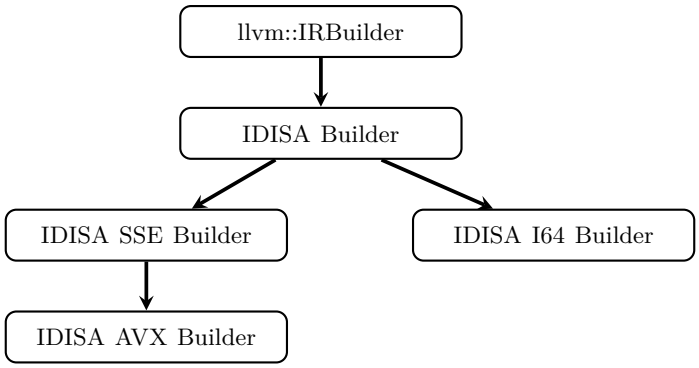

Figure 2.7: Inheritance Diagram for IDISA Builders

libraries for Parabix framework. Based on IDISA Builder, we optimize or specialize some of the library functions according to the target architecture.

IDISA Builder is also capable of detecting the existing architecture. For example, on Intel architectures, if AVX2 exists, it will select AVX2 Builder rather than SSE Builder by default, which tends to give a better performance in most of the Parabix applications. The block size will be automatically set to 256 for AVX and 128 for SSE. For architectures that do not support SIMD instructions, there is also an I64 Builder available to use general 64-bit registers.

2.3.3 Pablo Compiler

Different from the Pablo compiler in Parabix Tool Chain, which translates Python into C/C++, the Pablo compiler that integrated into Parabix Framework translates Pablo code into LLVM IR.

We also implemented a Carry Manager associated with Pablo Compiler so that all the Pablo operations that involving carry propagation such as `Advance`, `ScanThru` and `MatchStar` as well as conditional (if) and iterative (while) logic that requires carry test can be translated into LLVM IR. The Carry Manager calculates the total number of carries needed by the program and inserts the carry data into kernel struct, which will be dynamically allocated when kernel instance is created. Appendix A.2 shows the LLVM IR code of Editd kernel generated by Pablo Compiler and Kernel Builder using the Pablo code in Appendix A.1.

2.3.4 Kernel Builder

To build a kernel, we first need to define the inputs and outputs. To be more specific, the programmer should provide the type definitions of input buffers, input scalars, output buffers and output accumulators.

We use stream sets $N \times M$ to define input and output buffers where N denotes the number of streams in the set, and M denotes code unit size. For example, in ICgrep, we have three kernels, transposition kernel, regular expression matching kernel and matches scanning kernel. Transposition kernel transposes a byte stream into 8 basis bit streams. Therefore, the input of transposition kernel is a single stream of 8-bit code unit represented by Stream Set 1×8, and the output is a set of 8 parallel bit streams represented by stream set 8×1. Regular expression matching kernel computes the character classes based on basis bit streams and then performs the matching logic, which generates two bit streams, one bit stream for line breaks and one bit stream for matches. Thus, the output of regular expression matching kernel is represented by stream set 2×1. Matches scanning kernel scans the bit stream of matches and displays the lines where matches are found. This kernel takes the output of regular expression matching kernel as its input and does not have any output buffers.

After defining the inputs and outputs, the programmer needs to write kernel logic for a single block. Kernel Builder creates a kernel struct that by default consists of a segment number, a termination signal, and pointers to all the inputs and outputs. If a kernel is built with Pablo compiler, carry data will be automatically added into the kernel struct. Other kernels may require programmers to define kernel states by themselves to hold the dependency data. Kernel Builder will also generate a kernel initialization function, a default final block processing function and a segment processing function that iterates through the block-at-a-time kernel logic for one segment of the input data where the segment size can be user-defined.

2.3.5 Pipeline Builder

Pipeline Builder builds the complete program from a chain of kernels. It creates the input and output buffer defined in kernels and allocates them if it is not an external file buffer. If the input is an external file buffer (e.g., transposition kernel), a buffer pointer will be set to the memory space that holds the data of input. After preparing the input and output buffers, the Pipeline Builder will create kernel instances in which kernel structs are allocated. Then the segment processing function of each kernel will be called in a sequential order until a termination signal is set to true.

Programmers do not need to apply the Pipeline Builder while building a Parabix application. The code will be automatically generated by the Pipeline Builder used in Parabix Driver, which provides an interface that let programmers declare the input/output buffers and kernel instances, and thus offers the execution order of all the kernels. Appendix A.3 shows the LLVM IR code of the main program generated by the Pipeline Builder.

The basic Pipeline Builder can only build a sequential program. In order to make a better utilization of the multiprocessor and execute the kernels in parallel, we extend the Pipeline Builder with multithreading support, which will be discussed later in Section 4.1.

Chapter 3

Multi-Dimensional Parallelism

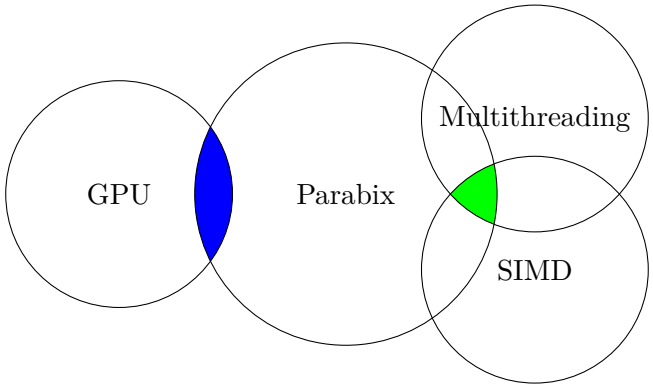

Figure 3.1: Multi-Dimensional Parallelism

3.1 Parabix + SIMD + Multithreading

Parabix is a technique that explores that parallelism by transposing the data into parallel bit streams. The performance of Parabix applications can achieve speedup with conventional registers and instruction set but can gain further improvement with a larger register width and specialized SIMD instruction set.

Parabix uses one bit to represent one position and can process W positions at a time where W is the register width. Ideally, when the register size is doubled, the performance of Parabix code should get a 2X speedup. However, some instructions might not be directly supported by a SIMD instruction set. For example, on SSE2, immediate shift operations of field width 128-bit has to be simulated by shift operations of field width 64-bit if the shift value is not multiple of 8 and takes a total of 2.3 instructions on average.

The first generation of AVX has focused primarily on floating point operations whereas 256-bit integer operations and shifts are not supported. Therefore, even though Parabix

can benefit from the 3-operand form introduced by AVX, it does not improve significantly and in some cases suffers degradation when comparing with SSE2 [31].

The extension to 256-bit integer SIMD operations becomes available in AVX2 instruction set. Combining Parabix with AVX2 has shown a promising performance improvement [13]. Other combinations, for example, Neon and Parabix also gives some interesting results [31].

Parabix can be limited but also benefit from specialized SIMD instruction set. For example, the SIMD `pack` instruction provided in both SSE and AVX series can be used to optimize the transposition from serial byte stream to parallel bit streams [10].

The usage of Parabix along with SIMD does not have any conflict with applying multi-threading to an application to achieve further performance improvement. For applications that are inherently sequential, we can consider a pipeline parallel strategy that organizes the program as a sequence of stages where each stage passes the processed data segment to the next stage and takes a new segment after its prior stage complete processing of that data segment. By assigning the stages to different threads, we can achieve parallel processing of sequential programs. Figure 3.2 showed the data dependency of a four stage pipeline where each column represents the processing of one data segment.

Take Parabix XML Parser as an example, it is constructed by 11 passes and partitioned into four stages [31]. Each of the passes is evaluated separately so that they can be grouped in a way to achieve an optimal load balance. Figure 3.3 showed an ideal case of a four stage pipeline where each stage consumes the same amount of time. When the first thread T_1 starts processing stage `A` of segment N, the second thread T_2 can begin to process stage `B` of segment $N-1$. T_n always wait for T_{n-1} to finish processing its next data segment because of the data dependency. The shared data is held in a circular buffer, which can help to save memory space and reduce cache misses. Suppose we create a circular buffer that can store M data segments, when T_1 finish processing segment N, it has to make sure that T_4 finish processing segment $N-M+1$ so that it could continue to process segment $N+1$ because it reuses the memory space that was taken by segment $N-M+1$.

The pipeline parallel strategy requires neither speculation nor pre-processing. However, in real applications, this technique can hardly get a full balanced division and is slowed down by the stalling time. The Parabix XML parser was able to achieve a 2X speedup on a quad-core machine [31], whereas ICgrep, an implementation that supports Unicode regular expression matching, only gets an average of 30% improvement on selecting test cases by a three stages division [12]. A segment pipeline parallelism strategy that reduces the waiting time caused by imbalance will be discussed in Section 4.1.2.

3.2 Parabix + GPU

Parabix technique can also be combined with GPU architecture. In order to achieve high performance on GPU architecture, thousands of threads must be launched simultaneously.

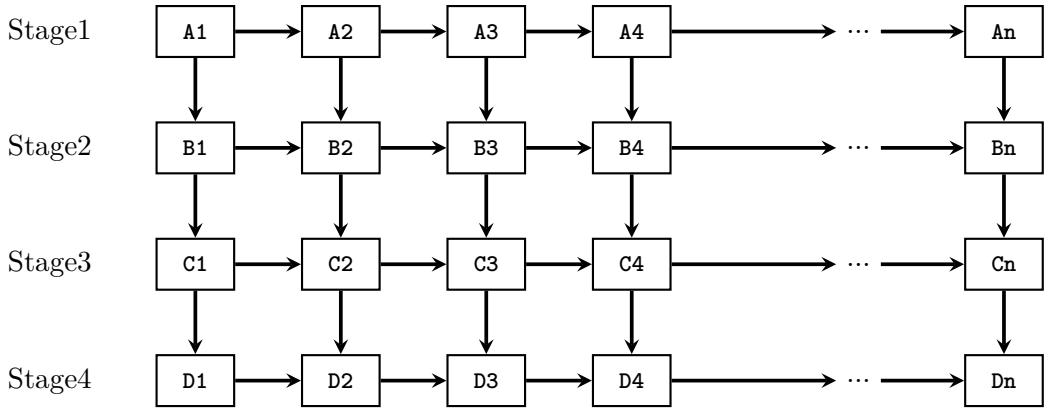

Figure 3.2: Data Dependency of Pipeline Stages

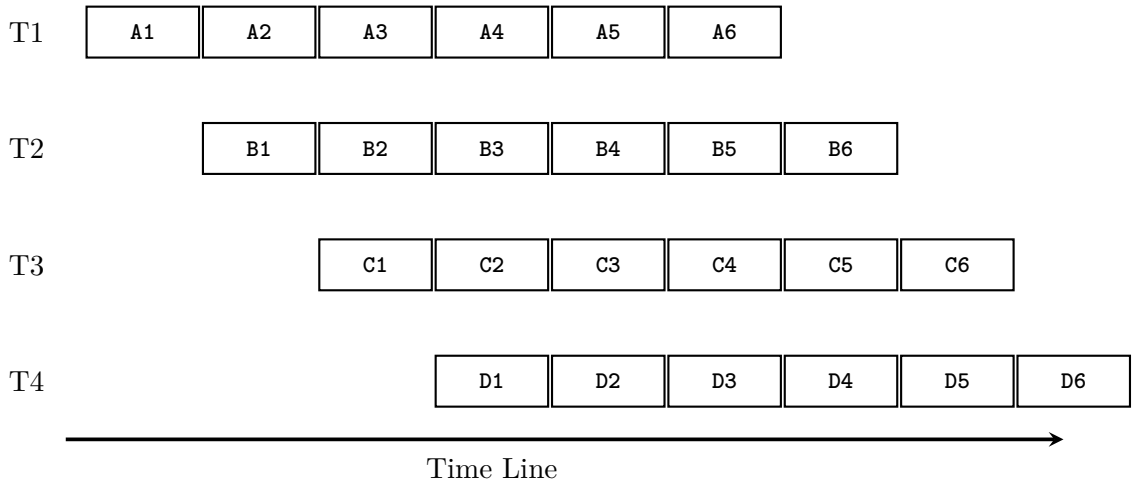

Figure 3.3: A Balanced Case of Four Stage Pipeline

Blocks of threads are divided into warps, which normally consists of 32/64 threads. A warp can be considered as a vector where each thread must execute the same instruction but with different sets of data. Therefore, we need long stream shift and addition for one or more warps to execute Parabix operations such as `Advance`, `ScanThru` and `MatchStar`.

Algorithm 1 and Algorithm 2 [13] shows the pseudo-code of long stream `Shift` and `Addition` function on GPU. We assign 64 threads to each block and use 64-bit values to hold input data. The carry propagation of `Shift` function is handled by *Carry*, which is allocated in shared memory and can be accessed by all the threads within the same block. `Barrier` function ensures that all the carry values have been calculated before merging to the final result. `Addition` function is more complicated than `Shift` as it may involve carry propagation not only limit to the adjacent threads. We create another variable *Bubble*, also allocated in shared memory, to record the possibility of continuous propagation. We use a six-step parallel-prefix style process to gather the *Carry* and *Bubble* bits as

shown in line 11-15 of Algorithm 2. This can be improved by the CUDA intrinsic function, `__ballot(int v)`, which returns a mask in which each bit indicates whether the argument `v` is true for the corresponding thread in the warp. `OR` function shown in Algorithm 3 is needed by control flow testing; because the predicates of a group of threads must be combined together.

With these three basic algorithms, we can implement all the Pablo operations such as `Advance`, `ScanThru` and `MatchStar` for GPU platforms. Therefore, any Pablo program can be translated into GPU programming language. However, using only one block of threads is a waste of GPU resources. Therefore we applied an overlapping strategy to partition the input data of ICgrep so that 4k of threads can be executed simultaneously [13]. Pipeline parallelism can also be implemented on GPU but limited by the number of stages that could be arranged. For example, if a program can only be divided into 3 stages, then at most 3 SMs can be utilized in this process.

Using data parallelism for Parabix applications might not be able to gain a huge benefit from GPU architecture in some cases because of the cost of large data transfers and of global memory accesses. On the other hand, compute-intensive applications make a better use of the resource. For example, in DNA sequence alignment, thousands of sequence reads are mapped to one reference genome so that we can assign each block of threads to process different sequence reads. More details will be discussed in Section 6.3.

Algorithm 1 Long Stream Shift on GPU

1: **function** SHIFT($threadID, Val, shftAmount, Carry, BlockCarry$)
2: $Carry[0] = BlockCarry$
3: $ShftVal = Val << shftAmount$
4: $Carry[threadID + 1] = Val >> (64 - shftAmount)$;
5: BARRIER()
6: $BlockCarry = Carry[64]$
7: $Result = ShftVal \vee Carry[threadID]$
8: **return** $Result, BlockCarry$

Algorithm 2 Long Stream Addition on GPU

1: **function** ADDITION($threadID, Val_1, Val_2, Carry, Bubble, BlockCarry$)
2: $Sum = Val_1 + Val_2$
3: $Gen = Val_2 \wedge Val_1$
4: $Prop = Val_1 \oplus Val_2$
5: $Carry[threadID] = ((Gen \vee (Prop \wedge \neg Sum)) \wedge (1 << 63)) >> (63 - threadID)$
6: **if** $Sum + 1 == 0$ **then**
7: $Bubble[threadID] = 1 << threadID$
8: **else**$Bubble[threadID] = 0$
9: BARRIER()
10: $offset = 32$
11: **while** $offset > 0$ **do**
12: $Carry[threadID] = Carry[threadID] \vee Carry[threadID \oplus offset]$
13: $Bubble[threadID] = Bubble[threadID] \vee Bubble[threadID \oplus offset]$
14: $offset = offset >> 1$
15: BARRIER()
16: $CarryMask = (Carry[0] << 1) \vee BlockCarry$
17: $BubbleMask = Bubble[0]$
18: $S = (CarryMask + BubbleMask) \wedge \neg BubbleMask$
19: $Inc = S \vee (S - CarryMask)$
20: $BlockCarry = (Carry[0] \vee (BubbleMask \wedge Inc)) >> 63$
21: $Result = Sum + ((Inc >> threadID) \wedge 1)$
22: **return** $Result, BlockCarry$

Algorithm 3 Long Stream Or on GPU

1: **function** OR($threadID, Val, Carry$)
2: $Carry[threadID] = Val$
3: BARRIER()
4: $offset = 32$
5: **while** $offset > 0$ **do**
6: $Carry[threadID] = Carry[threadID] \vee Carry[threadID \oplus offset]$
7: $offset = offset >> 1$
8: BARRIER()
9: **return** $Carry[0]$

Chapter 4

Multithreading Support for Parabix Framework

4.1 Integrated Parallelization Modes

Programmers do not need to put any effort to gain the potential performance improvement from the integrated modes. It is built into Parabix framework and can be automatically generated and compiled. As shown in Figure 4.1, we extend pipeline builder to support two parallelization modes, pipeline parallelism, and data-pipeline parallelism. Parallel Pipeline Builder is in charge of generating the thread function for every kernel and creating a parallel pipeline function that assigns the kernel functions to the created threads.

4.1.1 Pipeline Parallelism

Parabix framework focuses on streaming text processing applications, which are inherently sequential and cannot be easily divided into independent data segments. Pipeline Parallelism is not limited by the data dependencies and can be applied to most of the streaming text processing applications.

In section 3.1, we discussed the pipeline parallel strategy implemented in C/C++. We used a lock-free queue to synchronize between the stages. In Parabix framework, we can treat the kernels as a sequence of stages. But the kernel level communication means we have to dynamically generate the code in LLVM IR.

In order to build connection and synchronization between the threads, Parallel Pipeline Builder has to understand the relationship between the kernels. The dependencies can be more complicated than Figure 3.2 has shown. Therefore, we construct a producer table and a consumer table by analyzing the input and output buffers of all the kernels. We define kernel A is a producer of kernel B and kernel B is a consumer of kernel A if kernel B has at least one input that is calculated by kernel A.

For example, in ICgrep, the pipeline consists of 7 kernels, SourceKernel, S2P kernel, LineBreak kernel, RequiredStream kernel, CC kernel, Grep kernel and ScanMatch kernel. The input and output buffer of these kernels are listed in Table 4.1. Using the information from this table, we can calculate the producer table and consumer table for ICgrep as shown in Table 4.2 and Table 4.3 With the help of producer and consumer table, we can retrieve the information of processed data from related kernels and generates the synchronization between them.

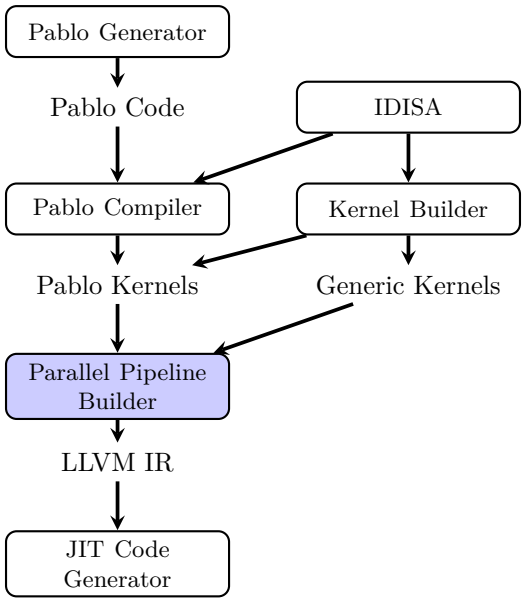

Figure 4.1: Pipeline Parallelsim Support of Parabix Framework

Kernel Name	Input Buffers	Output Buffers
Source Kernel		ByteStream
S2P Kernel	ByteStream	BasisBits
LineBreak Kernel	BasisBits	LineBreaks
RequiredStream Kernel	BasisBits	RequiredStreams
CC Kernel	BasisBits	CharClasses
Grep Kernel	CharClasses, LineBreaks, RequiredStreams	MatchResults
ScanMatch Kernel	LineBreaks, MatchResults	

Table 4.1: ICgrep Kernel Information

Algorithm 4 shows the pseudo-code of `PipelineThread` function in Parallel Pipeline Builder that is used to generate thread function for each kernel under pipeline parallelism mode. The input parameter n represents the kernel id. The local variable $SegNo$ that represents the number of the next segment to be processed is initialized to 0 (line 2). $SegNo$ is also stored in kernel struct and used for thread synchronizations. Reads and

Kernel ID	Kernel	Producer Kernels
0	Source Kernel	
1	S2P Kernel	0
2	LineBreak Kernel	1
3	RequiredStream Kernel	1
4	LineBreak Kernel	1
5	Grep Kernel	2, 3, 4
6	ScanMatch Kernel	2, 3

Table 4.2: Producer Table

Kernel ID	Kernel	Consumer Kernels
0	Source Kernel	1
1	S2P Kernel	2, 3, 4
2	LineBreak Kernel	5, 6
3	RequiredStream Kernel	5
4	CC Kernel	5
5	GrepKernel	6
6	ScanMatch Kernel	

Table 4.3: Consumer Table

writes of $SegNo$ from kernel struct requires atomic loads and stores with memory fence. We use acquire fence for all the atomic loads and release fence for all the atomic stores. The acquire fence prevents memory reordering of the read-acquire with any read or write operation that follows it in program order. The release fence prevents memory reordering of the write-release with any read or write operation that precedes it in program order. $TerminationSignal$ is also a member of kernel struct. It will be set to true once the kernel finish processing its last segment. Before kernel process one segment with function `KernelDoSegment` (line 10), we have to run tests to prevent kernels from executing if there is insufficient input data or output space. For each consumer kernels of $kernel_n$, we first read its $SegNo$ as $consumerSegNo$ and compares with local $SegNo$. If there is not enough space, that is, the difference between $consumerSegNo$ and $SegNo$ is larger than the number of segments in the buffer, the thread will repeat this test until the consumer kernel release the memory space and update its $SegNo$ (line 5,6). To check if there is sufficient input data, we need to go through each producer kernel of $kernel_n$ and see if the $SegNo$ of the producer kernel is larger than the local $SegNo$. If not, the thread will repeat the tests until all the producer kernels finish processing the current segment (line 8, 9). Once the input data and output space are ready, the thread will call `KernelDoSegment`, increase the local $SegNo$ by 1 and update the $SegNo$ of $kernel_n$. Appendix A.4 shows the pipeline thread function generated by the Parallel Pipeline Builder based on this algorithm.

Algorithm 4 Pipeline

```
1: function PipelineThread(n)
2:     SegNo = 0
3:     while TerminationSignal is false do
4:         for each consumerK ∈ ConsumerKernels do
5:             while SegNo > consumerSegNo + BufferSegments do
6:                 atomic load acquire consumerSegNo from consumerK
7:         for each producerK ∈ ProducerKernels do
8:             while SegNo >= producerSegNo do
9:                 atomic load acquire producerSegNo from producerK
10:        KernelDoSegment(SegNo)
11:        SegNo = SegNo + 1
12:        atomic store release SegNo to Kernels[n]
13:        load TerminationSignal from Kernels[n]
14:    ThreadExit
```

In chapter 4, Figure 3.3 showed a balanced case of a four stage pipeline where the kernels can immediately processing the data from next segment. However, in many applications, the time consumed by each stage is different. The performance of pipeline parallelism can be considerably reduced by the imbalanced workloads.

Figure 4.2 gives an example of a pipeline that has three kernels A, B, C. The output of kernel A is the input of kernel B and the output of kernel B is the input of kernel C. Kernel A takes C_A (3 time slots) to execute one segment of data, kernel B takes C_B (4 time slots) and kernel C takes C_C (2 time slots). Since kernel C is depended on kernel B, the thread that executes kernel C cannot run faster than the thread that executes kernel B. Therefore, the processing speed of this pipeline is limited by kernel B. A lot of CPU time is wasted as shown by the dot lines between kernel C executions.

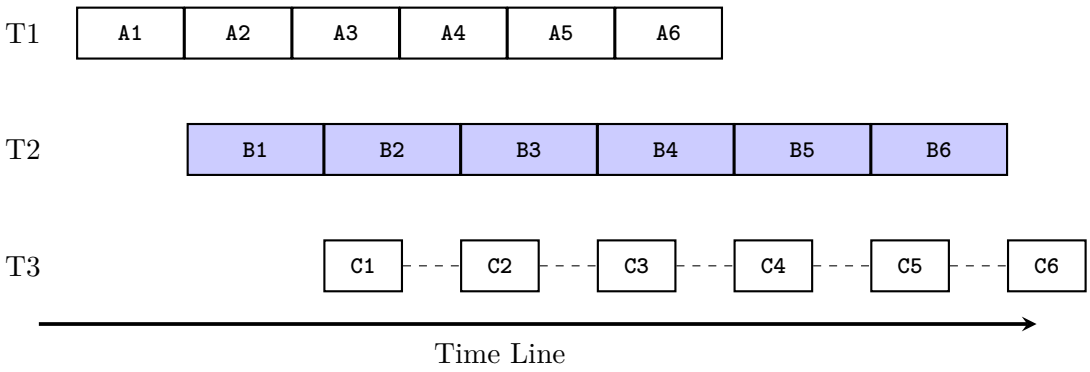

Figure 4.2: Pipeline Parallel

Let us assume the number of physical cores on the CPU is more than the number of kernels, the total processing time of M segments can be computed by $C_T = C_A + C_B \times M +$

$C_C \approx C_B \times M$. If the processing time of the sequential version is P_s, then the processing time of the pipeline parallel version can be computed by $P_e = P_s \times I$ where I is the imbalance factor of a pipelined program that can be calculated by equation 4.1 in which N is the total number of the kernels in the pipeline and K_n represents the performance of kernel n measured in CPU cycles per segment. Note that the overhead of thread synchronization and the possibility of generating more cache misses is not considered in this estimated processing time.

$$I = \frac{max(K_1, K_2, ...K_N)}{K_1 + K_2 + ... + K_N} \tag{4.1}$$

Equation $P_e = P_s \times I$ can not be used to compute the cases that do not have sufficient CPU resources. For example, if the pipeline is running on a dual-core machine, one of the thread has to wait for its turn to be executed and cause more stalling time for synchronization. In this case, $P_e < P_s \times I$. However, the processing time is still affected by the imbalance factor and can be expressed as $fn(I)$. The larger the imbalance factor is, the longer processing time it is required.

One way to improve the performance degradation caused by imbalance factor is to group the kernels and make a better distribution of the workloads. For example, as discussed in section 3.1, the 11 kernels in Parabix XML parser are grouped into 4 stages each processed by one thread. In this case, the imbalance factor I is only 0.31 where the optimal number of 4 threads is 0.25. (I is calculated using results from [31].) However, this solution requires the programmer to know the cost of each kernel very well or pre-measure their performance before grouping them. This method can also be limited by the dependencies between the kernels. For example, in Figure 4.2, we cannot group kernel A and kernel C because kernel C is dependent on kernel B.

4.1.2 Data-pipeline Parallelism

In order to address the load-balancing deficiencies of conventional pipeline parallelism as described in the previous subsection, in this section we describe a hybrid data-pipeline parallelism mode and its automated compiler support. Other approaches to overcoming limitations of pure pipeline parallelism that have been investigated in the context of StreamIt [20] and Cilk [29]. Although those frameworks are not designed to deal with the kinds of arbitrary-length and fine-grained dependencies found in streaming text processing applications, we will briefly contrast them to the data-pipeline approach at the end of this section.

Data-pipeline combines data parallelism and pipeline parallelism such that each thread will process different data segments in a pipeline. Let us suppose the total number of threads is T and $t \leq T$, then the data segments executed by thread T_t are B_t, B_{t+T},

B_{t+T*2}... Kernels are processed in a sequential order within a thread. However, a kernel cannot start processing block $m+1$ before it finishes processing block m in other threads.

The same example using data-pipeline parallelism is shown in Figure 4.3. Thread T_1 execute block 1, 3, 5 and thread T_2 execute block 2, 4, 6. When T_2 finish processing the second block of kernel A, it stalls and waits for kernel B to finish processing the first block. This generates only one slot of waiting time and the rest of the CPU time is fully utilized.

In this example, data-pipeline parallelism shows a better utilization of CPU resources compared with the conventional pipeline parallelism. We can get this improvement whenever the imbalanced factor I is less than $1/T$. However, if we increase the number of threads to 3, as shown in Figure 4.4, $I = C_B/(C_A+C_B+C_C) \approx 0.44$ will be larger than $1/T = 0.33$. This generates many slots of waiting time, and the performance is again dominated by kernel B.

A better CPU utilization does not necessarily give a better performance. However, when the CPU resource is limited, for example, if Figure 4.3 and Figure 4.4 are implemented on a dual-core machine, the CPU utilization becomes crucial to the performance.

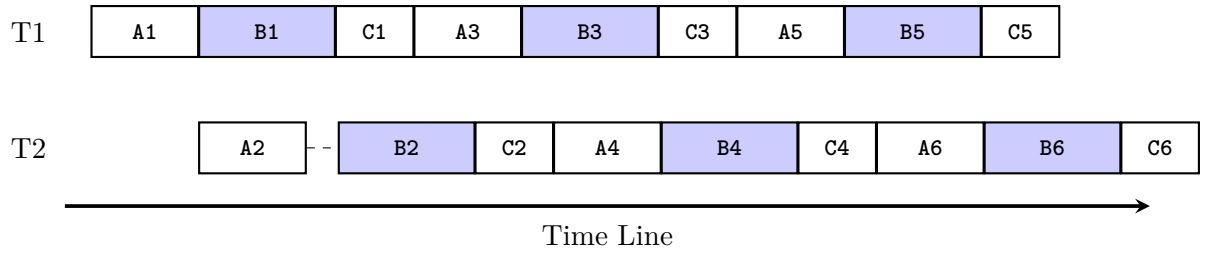

Figure 4.3: Data-pipeline Parallel with 2 Threads

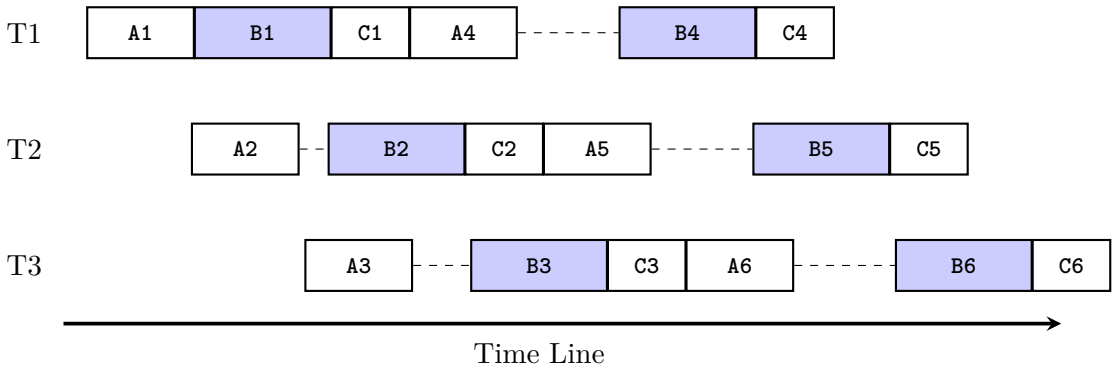

Figure 4.4: Data-pipeline Parallel with 3 Threads

When $I \leq 1/T$, the processing time of the data-pipeline parallel version can be computed by $P_e = P_s/T$ where P_s is the processing time of the sequential version. This is an optimal performance improvement that can be achieved by multithreading. It cannot be achieved

when the number of threads T we assigned is more than the number of kernels N. However, the condition $I \leq 1/T$ is not guaranteed by $N > T$; we could still face the problem of the imbalanced workloads in this case.

As illustrated in Figure 4.5, the time slots taken by kernel A is now changed to 1. The overall performance is limited by the processing speed of kernel B. The total processing time can be computed by equation $P_e = P_s \times I$ where I is the imbalance factor of the pipeline. We can see that in this example the processing time is the same as using conventional pipeline, but the number of processors required is less and therefore makes a better utilization of the CPUs.

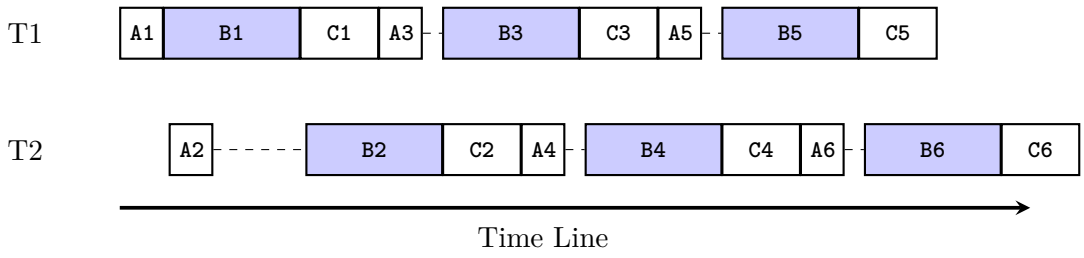

Figure 4.5: Data-pipeline Parallel with 2 Threads

By summarizing the two situations, we can compute the processing time of applying data-pipeline parallelism with equation 4.2. Note that this equation also applies to the case that $T \geq N$, because $P_s \times I$ is always larger than P_s/T when $T \geq N$.

$$P_e = max(\frac{P_s}{T}, P_s \times I) \tag{4.2}$$

Idealy, we would like to see that $P_e = P_s/T$. Therefore, we can try to reduce the imbalance factor so that $I \leq 1/T$. This can be achieved by splitting up the kernels that consume more CPU cycles in the pipeline to reduce $max(K_1, K_2, ...K_N)$. In general, for data-pipeline parallelization mode, the more kernels, the better.

Data-pipeline parallelism is supported by Parallel Pipeline Builder as well. It does not rely on the data dependencies between kernels because the kernels are executed in sequential order within a thread. Therefore, to get a proper input for a kernel K_n, we do not need to check its producer kernel but to make sure that K_n has finish processing segment m before it starts processing segment $m+1$. There is also no need to check consumer kernels. Segment m and $m+T$ are processed by the same thread and cannot be executed together. Therefore, no more than t segments of data can be written to an output buffer at the same time. Kernels always have enough space to store its outputs as long as the size of the output circular buffers is larger than the number of threads.

Algorithm 5 shows the pseudo-code of `DataPipelineThread` function in Parallel Pipeline Builder that is used to generate thread function under data-pipeline parallelism mode. The

input parameter *tid* represents the thread ID. The local variable $SegNo$ is initialized to *tid* and will be increased by T each time the kernel finish processing one segment. For each kernel in the pipeline, we first need to check if the previous segment has been processed. This test is implemented by comparing the local $SegNo$ with the $SegNo$ read from kernel struct and renamed as $processedSegNo$ (line 7). The thread will repeat this test until $SegNo$ match with $processedSegNo$. After the kernel finish processing a segment with `KernelDoSegment`, $TerminationSignal$ of all the kernels are combined with logical `OR` (line 12). If any of the kernels reach the end and sets its $TerminationSignal$ to true, the thread will call the `ThreadExit` function. Appendix A.5 shows the data-pipeline thread function generated by the Parallel Pipeline Builder based on this algorithm.

Algorithm 5 Data-Pipeline

```
1:  function DataPipelineThread(tid)
2:      SegNo = tid
3:      T = false
4:      while T is false do
5:          for each K ∈ Kernels do
6:              while SegNo ≠ processedSegNo do
7:                  atomic load acquire processedSegNo from K
8:              KernelDoSegment(SegNo)
9:              SegNo = SegNo + T
10:             atomic load acquire TerminationSignal from K
11:             atomic store release SegNo to K
12:             T = T|TerminationSignal
13:     ThreadExit
```

Further optimization of data-pipeline parallelism can be done to those data independent kernels. For example, the S2P kernel that transposes byte-oriented data to parallel bit stream can process data segments in parallel and does not need to wait for the result from the previous segment. Therefore, we can remove the thread synchronization test (line 6,7) and reduce the overhead. This optimization may also improve P_e (equation 4.2) in the cases that the S2P kernel is the bottleneck in the pipeline and affects the imbalance factor I.

A compiler system based on StreamIt language has been developed to leverage the right combination of task, data, and pipeline parallelism [20]. One of the strategies combines data and pipeline parallelism where the filters (similar to kernels in Parabix framework) that have state dependencies can be assigned to different processors and executed in parallel with others. A greedy partitioning heuristic is applied to optimize the load balancing when assigning filters to cores. This requires the throughput of each filter known by the programmer or computable by the compiler. The implementation also needs a separate communication stage to shuffle data between buffers that read or written by the filters since the data segments are processed by different cores.

Another approach Cilk-P extends Cilk parallel-programming model to support on-the-fly pipeline parallelism, which allows a more flexible structure of the pipeline built dynamically as the program executes [29]. Cilk-P applies the bind-to-item approach [38] where the threads carry the item through the pipeline to the last stage or until an unresolved dependency is encountered. Cilk-P implementations have shown competitive performance results compared with TBB but require hand-compilation.

4.2 Customized Modes

Programmers get to decide how to parallelize with customized modes. Both task parallelism and data parallelism can be implemented with Parabix framework as customized modes.

4.2.1 Task Parallelism

Task parallelism refers to performing tasks concurrently across multiple processors. In this paper, we define the parallel tasks as independent processes that have no communication between each other. Therefore, the implementation can focus on arranging the tasks to different threads and synchronization does not need to be considered during the task processing.

Multiple Input Files

One useful case that task parallelism can apply is when multiple input files need to be processed. For example, in ICgrep, the user might want to search an regular expression against all the files in a specified directory. As shown in Figure 4.6, we first compile the pattern into Pablo kernel and then schedule the tasks so that they load in different files and process them in a sequential order. Once a thread finishes processing one input file, it will immediately load the next one in the queue.

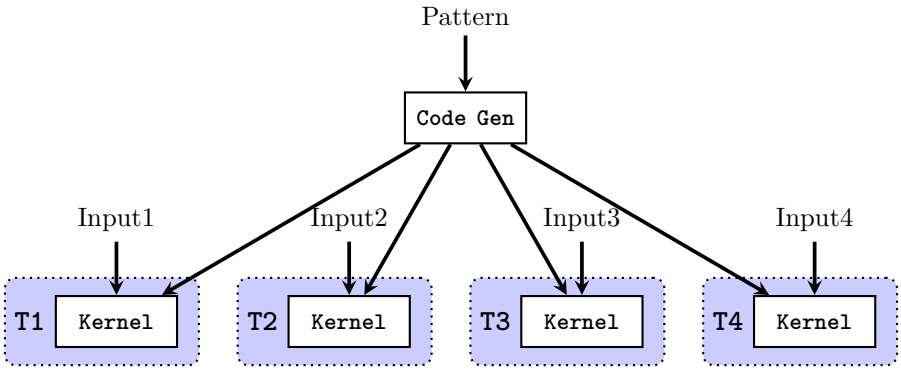

Figure 4.6: Task Parallelism for Multiple Input Files

Multiple Independent Kernels

Another useful case for task parallelism is when we have multiple independent kernels. However, most of the Parabix applications are relying on the new representation of text. Therefore, we need to generate a separate pipeline which includes one kernel called S2P that transpose the byte-oriented input data to parallel bit streams.

As shown in the example of Figure 4.7, multiple patterns need to be matched. Before initializing the tasks, we must first execute the S2P kernel to calculate the bit streams. Each task is not only running the matching process but also in charge of compiling different patterns and generate the corresponding kernel pipeline. In some cases, we might need a merging kernel that combines the matching results.

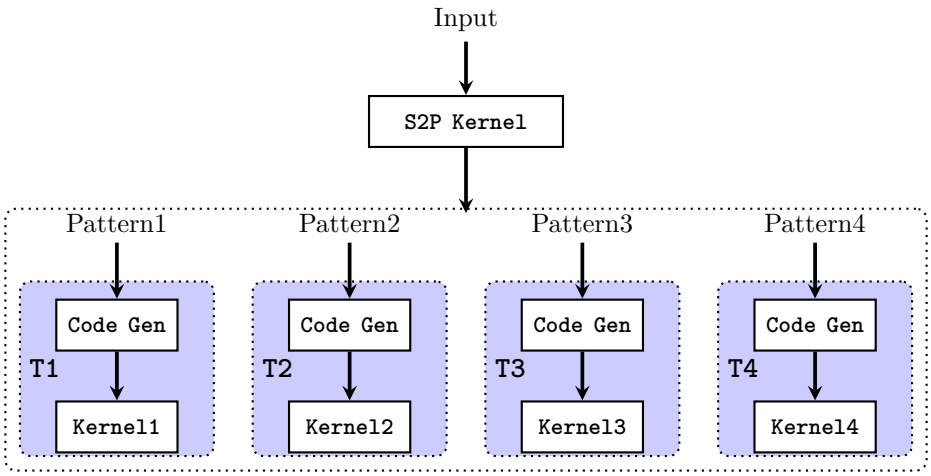

Figure 4.7: Task Parallelism for Multiple Independent Kernels

4.2.2 Data Parallelism

Data parallelism focuses on distributing the data across different processors. In fine-grained data parallelism, the data is evenly divided and hence facilitates load balancing on multi-processor systems. For example, in Figure 4.7, we can apply fine-grained data parallelism to S2P kernel and speed up the program by concurrently transpose the input data.

However, in some applications, the partition is more complicated because of data dependencies. Therefore, the data cannot be equally divided up and generally a pre-process that helps find the proper division points is required before applying a coarse-grained data parallelism. Figure 4.8 showed the data-parallel strategy that we integrated into Parabix framework.

For example, in ICgrep, the target is to match regular expressions to each line of the input text data. If we cut a line in the middle, we could miss some of the match positions. Therefore we choose to divide the data at the line breaks so that we do not need to handle the across boundary cases. During the pre-processing step, all the line break positions are

calculated first. Then we can equally group the line break positions and use the selected positions as division points.

On CPU, we create threads and let each of them reads one chunk of data and execute the same pipeline kernels. On GPU, we provide the division points to each work group so that they can load the corresponding input data from global memory.

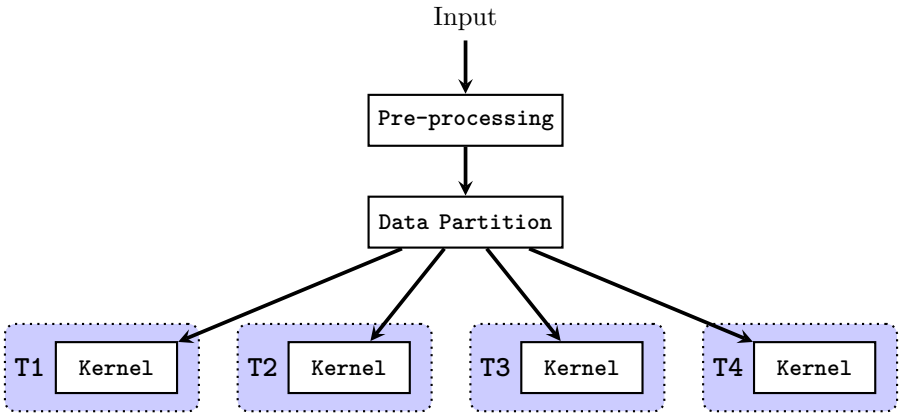

Figure 4.8: Coarse Grained Data Parallelism

Chapter 5

GPU Support for Parabix Framework

5.1 NVPTX Back-end

Parallel Thread eXecution (PTX) is an intermediate representation for Nvidia GPUs. The NVIDIA CUDA Compiler first translates code written in CUDA to PTX, and then compiles it into a binary code that can be run on GPU device. In order to execute Parabix kernels on GPU, we need a PTX generator that converts LLVM IR to PTX instructions and a CUDA driver that executes the PTX code on the GPU device and communicates with the CPU host. An overview of the general usage of NVPTX back-end is provided by LLVM user guide [4]. Additional detail can be found in the NVVM IR Specification [2]. This section will give an introduction to the material that is related to building the Parabix framework for GPU.

5.1.1 Conventions and Intrinsics

Address Spaces

In PTX, we need to specify the address space for each memory allocation. By default, the address space is 0, which represents the private memory that is only available for the thread. As shown in Figure 5.1, address space 3, the shared memory can be accessed by all threads within a group. Global memory, defined by address space 1, is visible to all threads and allows memory copy from host memory space. The NVPTX back-end uses 6 different address space mappings, Figure 5.1 only lists the address space that is required to enable the GPU support for Parabix frameowork.

Address Space	Memory Space
0	Private
1	Global
3	Shared

Figure 5.1: Address Space Mapping

Special Register Intrinsic	CUDA Builtin
`llvm.nvvm.read.ptx.sreg.tid.x`	`threadId`
`llvm.nvvm.read.ptx.sreg.ctaid.x`	`blockIdx`

Figure 5.2: Special Register Mapping

Special Registers

In order to retrieve different sets of data for parallel processing, each thread needs to read its threadId and blockIdx from special registers. Figure 5.2 lists the special register intrinsics that map to CUDA builtins.

Barrier and Memory Fence

`llvm.nvvm.barrier0` is equivalent to function call `__syncthreads()` in CUDA. It ensures that all threads in the thread block have reached this point before executing the next instruction. `llvm.nvvm.barrier0.or` not only enables a memory fence but also evaluates predicate for all threads of the block and returns non-zero if and only if any of predicate is non-zero. Their equivalent CUDA functions are listed in Figure 5.3.

5.1.2 Constructing GPU Kernel

Several basic components are needed to build a GPU kernel with LLVM IR. First, we need to specify the data layout and target triple. The data layout string determines the size of common data types, which could be either 32-bit or 64-bit. The target triple is `nvptx-nvidia-cuda` for 32-bit and `nvptx-nvidia-cuda` for 64-bit.

The second component is device functions and kernel functions. Device functions can be called only from the device code whereas the kernel functions are called by the host

Barriers Intrinsic	CUDA Builtin
`lllvm.nvvm.barrier0`	`__syncthreads()`
`llvm.nvvm.barrier0.or`	`__syncthreads_or()`

Figure 5.3: Barriers Intrinsic Mapping

code. Here, we refer device and host to GPU and CPU respectively. NVPTX back-end uses metadata to mark a function as the kernel function. This is also the last component required for constructing the GPU kernel.

5.2 Basic Design

Figure 5.4 shows the four modules that have been added to Parabix framework in order to support applications on GPU architecture, NVPTX IDISA Builder, NVPTX Driver, NVPTX Generator, and NVPTX Executor. These four modules are available only if user enabled CUDA Compilation and CUDA is installed on the OS and can be found by Parabix.

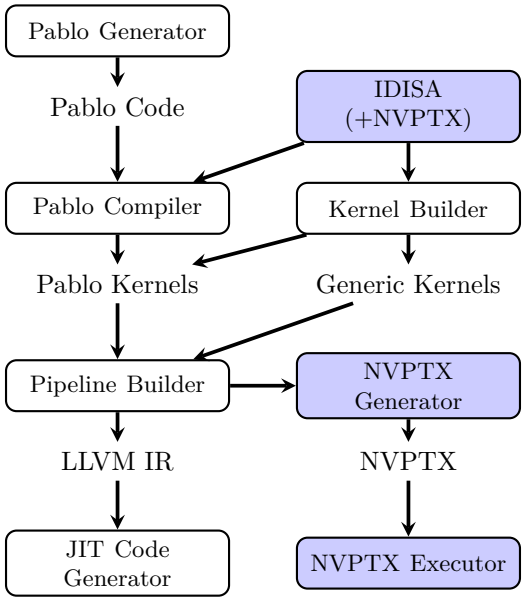

Figure 5.4: GPU Support for Parabix Framework

5.2.1 IDISA NVPTX Builder

Parabix uses 64-bit integer as a common data type for GPU kernel code. Therefore, IDISA NVPTX builder is inherited from IDISA I64 Builder. However, there are a few functions that need to be specialized including `bitblock_advance`, `bitblock_add_with_carry`, `bitblock_any` and `bitblock_mask_from` because the block size is no longer 64-bit but 64×64-bit as we define the number of threads per block to be 64.

Long Stream Shift/Addition

`bitblock_advance` and `bitblock_add_with_carry` are based on the algorithms of long stream shift and addition described in Section 3.2. Instead of writing these two algorithms in GPU programming language, IDISA NVPTX Builder provides functions that can generate

them in LLVM IR. Figure 5.5 showed the generated `Advance` function based on Algorithm 1. The shared data that used to handle carry propagation is constructed as a global variable `carry` allocated in address space 3. We declare $N + 1$ elements for `carry` where N is the number of threads per group. In the example shown in Figure 5.5, we set each group to have 64 threads. Every thread saves its own carry bit to one of the elements, and the blockCarry takes the extra position.

```
2  define { i64, i64 } @Advance(i32 %id, i64 %val, i64 %shftAmount, i64 %blockCarry) {
3  entry:
4    %c0 = getelementptr [65 x i64], [65 x i64] addrspace(3)* @carry, i32 0, i32 0
5    store i64 %blockCarry, i64 addrspace(3)* %c0
6    %0 = shl i64 %val, %shftAmount
7    %1 = add i32 %id, 1
8    %2 = getelementptr [65 x i64], [65 x i64] addrspace(3)* @carry, i32 0, i32 %1
9    %3 = sub i64 64, %shftAmount
10   %4 = lshr i64 %val, %3
11   store i64 %4, i64 addrspace(3)* %2
12   call void @llvm.nvvm.barrier0()
13   %c64 = getelementptr [65 x i64], [65 x i64] addrspace(3)* @carry, i32 0, i32 64
14   %blockCarryOut = load i64, i64 addrspace(3)* %c64
15   %5 = getelementptr [65 x i64], [65 x i64] addrspace(3)* @carry, i32 0, i32 %id
16   %carryVal = load i64, i64 addrspace(3)* %5
17   %6 = or i64 %0, %carryVal
18   %7 = insertvalue { i64, i64 } undef, i64 %6, 0
19   %8 = insertvalue { i64, i64 } %7, i64 %blockCarryOut, 1
20   ret { i64, i64 } %8
21 }
```

Figure 5.5: LLVM IR for Advance Function

Control Flow Testing

Earlier work uses Algorithms 3 to calculate the predicate for a group of threads. It is now simplified by a single intrinsic function `llvm.nvvm.barrier0.or`, which takes a 32-bit integer as input parameter and returns non-zero if and only if any of the input parameters from a thread is non-zero. The test variable must first be compared with a null value of the same type and then cast to a 32-bit integer.

EOF Mask Calculation

Parabix kernels provide a `DoFinalBlock` function in which programmers can apply the EOF mask to the last block of the input. On GPU, threads in a warp must execute the same instruction. In order to avoid branches generated by locating the thread that needs EOF mask, the mask is calculated for every thread in the group and then applied to all the final blocks. We define the final blocks as full block, empty block or target block. The EOF mask of full blocks are *AllZeros*, the EOF mask of empty blocks are *AllOnes* and the EOF mask of the target block is calculated by *AllOnes* shifted by bytes left in the last block. If

we let P be the position of the last byte and N be the number of threads in a group, the EOF Mask can be calculated by the following formular.

$$\text{EOF Mask} = (AllOnes << (P\%N)) \wedge (ThreadId == (P \div N)) \vee (ID > (P \div N))$$

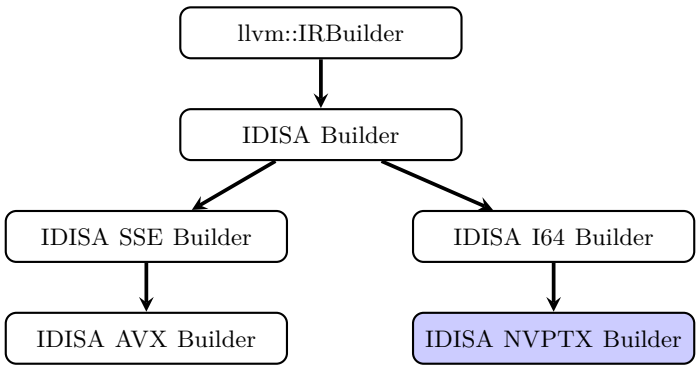

Figure 5.6: Inheritance Diagram for IDISA Builders

5.2.2 PTX Generator

The PTX Generator translates the LLVM IR into PTX code. The similarity between these two representations makes it possible to implement most of the translations with simple mappings [26]. Figure 5.7 shows the generated `Advance` function in PTX by compiling the input of LLVM IR shown in Figure 5.5. Most of the operations have direct mappings from LLVM IR to PTX. For example, `call void @llvm.nvvm.barrier0()` is translated into `bar.sync 0`. A not straightforward example, `getelementptr`, which retrieves pointer by index is compiled into memory address calculations in PTX.

LLVM provides a tool called LLC that compiles LLVM source inputs into machine language for a specified architecture. We made some modification and integrated this tool into Parabix framework. The three basic components discussed in Section are needed in order to construct a valid input for PTX Generator. The code generated by Pipeline Builder can be considered as a device function. This device function is called by a kernel function that first reads input data from global memory and then stores the output after the calculation.

5.2.3 NVPTX Executor

The NVPTX Executor loads the PTX code into a CUDA module and specifies the name of kernel function that matches the one created in LLVM IR. Before launching the kernel, we need to allocate device memory and copy the data from CPU memory to GPU global memory. This could be a simple transfer of the entire input buffer but can also involve data partitioning in complicated cases that require programmer's work. CUDA Driver uses

```
.visible .func  (.param .align 8 .b8 func_retval0[16]) Advance(
        .param .b32 Advance_param_0,
        .param .b64 Advance_param_1,
        .param .b64 Advance_param_2,
        .param .b64 Advance_param_3
)
{
        .reg .s32       %r<5>;
        .reg .s64       %rd<16>;

        ld.param.u32    %r1, [Advance_param_0];
        ld.param.u64    %rd1, [Advance_param_3];
        st.shared.u64   [carry], %rd1;
        mov.u64         %rd2, carry;
        ld.param.u64    %rd3, [Advance_param_1];
        ld.param.u64    %rd4, [Advance_param_2];
        cvt.u32.u64     %r2, %rd4;
        shl.b64         %rd5, %rd3, %r2;
        add.s32         %r3, %r1, 1;
        mul.wide.s32    %rd6, %r3, 8;
        add.s64         %rd7, %rd2, %rd6;
        mov.u64         %rd8, 64;
        sub.s64         %rd9, %rd8, %rd4;
        cvt.u32.u64     %r4, %rd9;
        shr.u64         %rd10, %rd3, %r4;
        st.shared.u64   [%rd7], %rd10;
        bar.sync        0;
        ld.shared.u64   %rd11, [carry+512];
        mul.wide.s32    %rd12, %r1, 8;
        add.s64         %rd13, %rd2, %rd12;
        ld.shared.u64   %rd14, [%rd13];
        or.b64          %rd15, %rd5, %rd14;
        st.param.b64    [func_retval0+0], %rd15;
        st.param.b64    [func_retval0+8], %rd11;
        ret;
}
```

Figure 5.7: Generated PTX for Advance Function

C/C++-style runtime APIs that are similar to conventional CUDA program but does not require compiling with NVCC. Therefore it can be integrated into Parabix framework and compiled with GCC/G++.

5.2.4 NVPTX Driver

The NVPTX Driver extends Parabix Driver to build the program that can be compiled into PTX and executed on GPU devices. Programmers can declare the input/output buffers and kernel instance the same way as using Parabix Driver for a CPU program. The basic components such as data layout, target triple and metadata are automatically created by the NVPTX Driver. However, the programmers have to write the code to assign different input data to different threads and choose the memory location to store the output results by using the special register intrinsic in Table 5.2.

5.3 GPU Support Limitations

While the IDISA NVPTX Builder is capable of handling Pablo code generally, there are some important limitations.

5.3.1 Dynamic Memory Allocation

LLVM IR supports C/C++ function calls by linking the functions to the ExecutionEngine. In ICgrep, we call an extern C function to report the matched lines. However, on GPU, the device functions cannot call functions to execute on the host. We need to allocate memory space on GPU to store the data of which the size can only be known at runtime.

Dynamic global memory allocation is supported by devices of compute capability 2.x and higher. We can write a malloc function in PTX and add it to the NVPTX IDISA builder as an inline assembly. However, in ICgrep, the report function is only called when a match position is found. It is highly inefficient if the results are sparsely distributed in the input text file which keeps most of the thread inactivate. The current Parabix framework does not support dynamic global memory allocation. The result bit streams are sent back to CPU for scanning.

5.3.2 Limited Memory Space

We have to be very careful when using the memory space on GPU. The input data that is transferred to GPU cannot exceed the limit of global memory. Large files can be partitioned into smaller chunks, but the carry data has to be kept in the global memory and passed to the next chunk. This is not dynamically supported by the current Parabix framework and has to be taken care of by the programmers.

5.3.3 Long Stream Advances

We implemented Algorithm 1 in the IDISA NVPTX Builder, which operates on a 64×64 bit streams. Advances of more than 64×64 positions need further work and cannot be handled by the current IDISA NVPTX Builder.

5.3.4 Manual Kernels

With the support of IDISA NVPTX Builder, the dependencies between threads can be handled for all the Pablo Kernels. However, manual kernels that have state dependencies may not be able to be properly built for GPU. In this case, specialized library functions may be required to extend the current IDISA NVPTX Builder.

Chapter 6

Case Studies

6.1 Experimental Platform and Methodology

Most of our performance measurements were done on a quad-core machine shown in Table 6.1. The GPU platform is a Nvidia GEFORCE hosted on an Intel Core i3 dual-core machine. Table 6.2 lists the basic information of this graphic card. Both platforms run the 64-bit Ubuntu 15.04 with GNU C/C++ compiler at version 4.9.

Physical Cores	4
Threads	8
Base Clock	3.4 Ghz
Instruction Cache	32 KB
L3 Cache	8 MB

Table 6.1: Intel Core i7

CUDA Cores	6×128
Base Clock	1024 Mhz
Shared Memory	96K
Global Memory	2G

Table 6.2: GEFORCE GTX 950

The Parabix framework includes a built-in kernel cycle counter, by which users can see the performance of each kernel measured in CPU cycle per byte. The total processing time is measured with `perf`, which is a lightweight profiler included in the Linux kernel. All of the programs are optimized to use SSE2 instruction by default. On GPU, we use the CUDA event API that includes calls to compute the elapsed time in milliseconds between two recorded events. We take the average processing time of 3 runs. We exclude the compilation time of generating LLVM IR for individual kernels and the compilation time that translates LLVM IR to PTX.

6.2 ICgrep

ICgrep is a Parabix based application that searches files for regular expressions in the same way as other grep applications do. ICgrep has shown a significant performance improvement over comparative implementations, such as grep, nrgrep, agrep and pcregrep on commodity processors [13]. In this section, we compare ICgrep under different parallelization mode including task parallelism, pipeline parallelism and data-pipeline parallelism on a multicore CPU. We also include the performance comparison of the GPU implementation versus CPU implementation.

In the previous work of ICgrep [13], we only measured the performance of running single regular expressions. Multiple regular expressions are handled as one regular expression and processed in one kernel. For example, suppose we have 4 regular expressions, a*, b*, ab* and (ab)*. They will be concatenated into a*|b*|ab*|(ab)* before passing to RE compiler, which is the Pablo generator of ICgrep.

The advantage of the concatenation is that the optimizer can find more common subexpression in the generated Pablo code. However, this complex Grep kernel may become the bottleneck of a pipeline, which could limit the performance benefit that might be achieved from either pipeline parallelism or data-pipeline parallelism.

We introduce the concept of multi-grep kernel that divides up the regular expression set into groups so that a*, b*, ab* and (ab)* can be compiled separately into 4 Grep kernels. We can also set the number of regular expression per group to 2 and generate 2 Grep kernels that process a*|b* and ab*|(ab)* respectively.

The basic ICgrep pipeline have seven kernels, Source kernel, S2P kernel, LineBreak kernel, RequiredStream kernel, CC kernel, Grep kernel, and ScanMatch kernel. A Merge kernel is needed for combining the match results from Grep kernels when multiple Grep kernels exist in the pipeline. Distributing the work of the Grep kernel to more than one kernel can balance the workloads and may improve the performance of the two integrated parallelization modes.

6.2.1 Test Cases

Commonly Used Regular Expressions

Table 6.3 lists five regular expressions used in our previous work [13] (three of them are selected from Benchmark of Regex Libraries). The 15 input text files are downloaded from Wikibooks. Most of the tests are run on a 620MB concatenated version (wikibooks.xml) except the experiment that is done for task parallelism which requires multiple input files.

Name	Expression
@	`@`
Date	`([0-9][0-9]?)/([0-9][0-9]?)/([0-9][0-9]([0-9][0-9])?)`
Email	`([^ @]+)@([^ @]+)`
URIOrEmail	`([a-zA-Z][a-zA-Z0-9]*)://([^ /]+)(/[^ ]*)?|([^ @]+)@([^ @]+)`
Xquote	`["']|&quot;|&apos;|&#0*3[49];|&#x0*2[27];`

Table 6.3: Test Expressions

Spam Rules

In order to see how ICgrep scales with a large regular expression set under different parallelization modes, we introduce a spam rule set that is extracted from Apache SpamAssassin standard ruleset version 3.4.0, which is available in `http://www-us.apache.org/dist/spamassassin/source/` and Debian mailing lists that can be checked out from `svn://svn.debian.org/svn/pkg-listmasterat` at revision 450. The combined rule set includes over three thousand rules represented by regular expressions.

We use two concatenated input text files. One of them is 32MB and selected from the SpamAssassin public corpus available in `http://spamassassin.apache.org/old/publiccorpus/`. The other one is 1.4GB and chosen from the Enron e-mail dataset, which is available in `https://www.cs.cmu.edu/~./enron/`.

6.2.2 Results and Analysis

Pipeline vs Data-pipeline

We compare the processing time of the sequential version of ICgrep and the two integrated parallelization modes, conventional pipeline parallelism (CP) and data-pipeline parallelism (DP). We use the five test cases listed in Table 6.3 and the input text file wikibooks.xml.

As shown in Figure 6.1, conventional pipeline parallelism can only achieve 18% improvement in its best test case `URIOrEmail` and shows a performance degradation in another test cases `@`. This shows that the overhead generated by CP mode can sometimes be more than the benefit it can gain.

To understand the reason the why conventional pipeline parallelism does not perform well in ICgrep, we measured the performance of each kernel under different test cases using Parabix kernel cycle counter as shown in Table 6.4.

We define the performance speedup P of a parallelized program over the sequential version using the following formula. (T_s represents the processing time the sequential version. T_p represents the processing time of the parallelized version.)

$$P = \frac{T_s - T_p}{T_p} \times 100\%$$

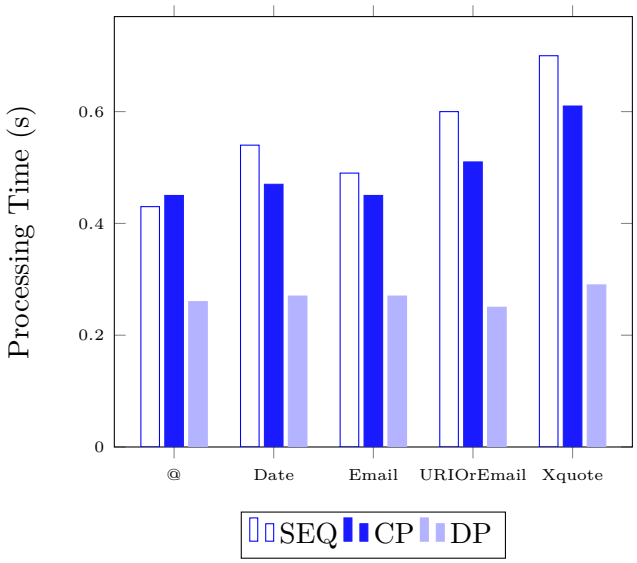

Figure 6.1: Comparison between Pipeline and Data-pipeline Mode

	Source	S2P	LineBreak	RequiredStream	CC	grep	ScanMatch
@	0.00	0.89	0.36	0.41	0.09	0.20	0.25
Date	0.00	0.89	0.36	0.41	0.14	0.75	0.25
Email	0.00	0.89	0.36	0.41	0.14	0.52	0.25
URIOrEmail	0.00	0.89	0.36	0.41	0.24	0.94	0.25
Xquote	0.00	0.89	0.36	0.41	0.57	1.12	0.25

Table 6.4: Kernel Profile (Cycles/Byte)

Table 6.5 shows the performance speedup P and the imbalance factor I calculated using formula 4.1 for all the test cases. We can see that the smaller the imbalance factor is, the better performance we can get with conventional pipeline parallelism.

	@	Date	Email	URIOrEmail	Xquote
I	0.405	0.317	0.346	0.304	0.311
Speedup	-5%	15%	9%	18%	15%

Table 6.5: Imbalance Factor and Performance

Different from conventional pipeline parallelism, all of the test cases that run in the data-pipeline mode using 4 threads achieves a significant performance improvement (Figure 6.1).

Using the formula 4.2, we calculate the processing time for the five test cases under data-pipeline mode with thread number 2, 3 and 4. The results are shown in Table 6.6.

	@	Date	Email	URIOrEmail	Xquote
2-threads	0.21	0.27	0.24	0.30	0.35
3-threads	0.17	0.17	0.17	0.20	0.23
4-threads	0.17	0.17	0.17	0.18	0.21

Table 6.6: Estimated Minimum Processing Time (s)

The real processing time will be a little bit longer than the estimated minimum processing because of the overhead, which includes cache misses caused by reading and writing to shared data between threads and branch mispredictions caused by controlling the kernels based on their dependencies. If we compare the real processing time shown in Figure 6.2 with the estimated minimum processing time shown in Table 6.6, we can find a similar pattern between the two.

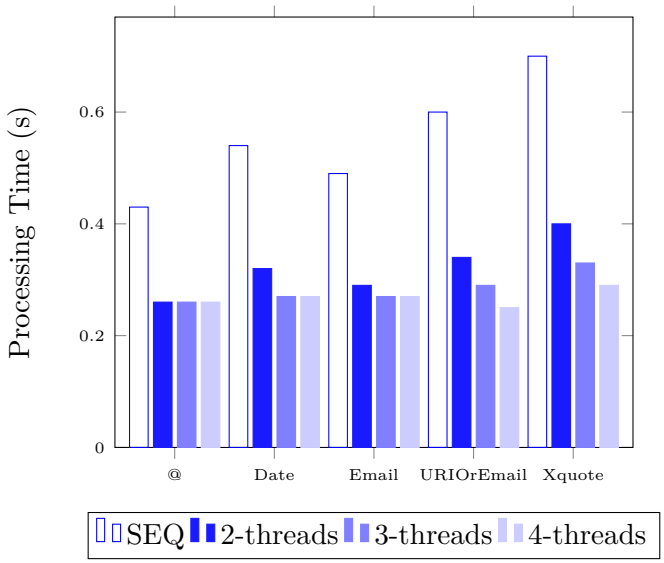

Figure 6.2: Real Processing Time

Multi-grep Kernel

Multi-grep kernel is designed to balance the workloads when multiple regular expressions are run in a single execution. We select all the test regular expressions in Table 6.3 except `@` and measure the performance of all the kernels using three grouping strategies, 1 regular expression per kernel, 2 regular expressions per kernel and 4 regular expressions per kernel.

Since the performance of Source kernel, S2P kernel, LineBreak kernel, RequiredStream kernel and ScanMatch kernel are the same under the 3 different circumstance and very close to the results in Table 6.4, Table 6.7 only lists the results of three kernels, CC kernel, Grep kernel and Merge kernel measured in the sequential mode. The sum of all the kernels is

calculated and provided in the table as well. We can see a performance degradation as we distributed the workloads to more Grep kernels. But it also shows that when using only one Grep kernel, that is 4 regular expressions per kernel in this example, the Grep kernel becomes the bottleneck in the pipeline and the cost of Grep kernel is more than twice of any of the rest of the kernels. By applying multi-grep kernel strategy, the bottleneck is reduced from 1.92 Cycles/Byte to 1.06 Cycles/Byte.

	CC1	Grep1	CC2	Grep2	CC3	Grep3	CC4	Grep4	Merge	sum
4 re/kernel	0.85	1.92								4.51
2 re/kernel	0.21	0.75	0.83	1.57					0.06	5.16
1 re/kernel	0.13	0.69	0.13	0.54	0.23	0.72	0.52	1.06	0.06	5.80

Table 6.7: Kernel Profile (Cycles/Byte)

Figure 6.3 shows the processing time under data-pipeline mode with different number of threads. With 1 regular expression per kernel, we can achieve up to 185% speedup whereas without using the multi-grep kernel, we can only achieve 90% speedup. The multi-grep version outperforms the single-grep version as the number of the threads increases to 4.

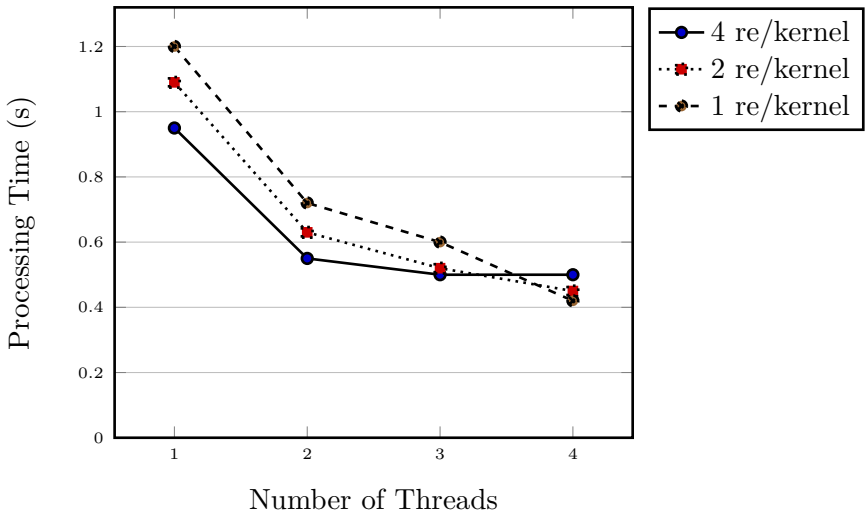

Figure 6.3: Multiple Regular Expressions

Spam Rules

To find out how ICgrep scales to larger numbers of regular expressions, we measured the performance of ICgrep with the spam rule set that contains 3027 rules. we also created two subsets with 100 rules and 1000 rules that are randomly selected from the 3027 rules.

Before applying any of the parallelization methods, we first compare the sequential version of ICgrep with one of the existing application pcregrep since ICgrep has not been tested with a large regular expression set in our previous work.

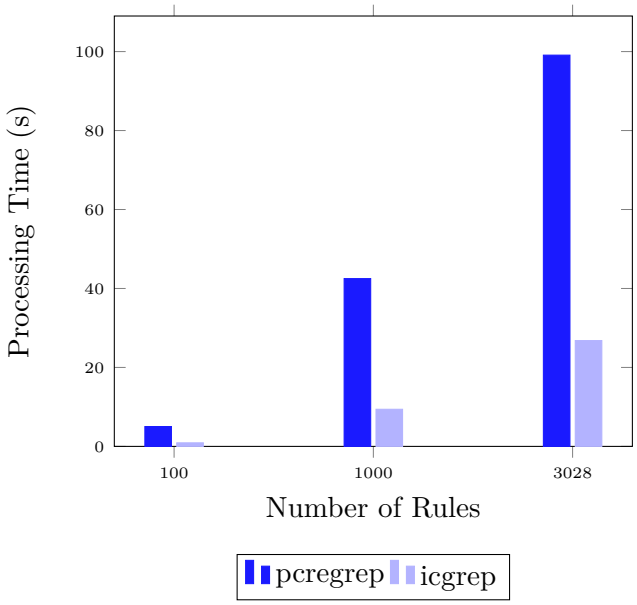

Figure 6.4: SpamAssassin public corpus (32 MB)

Figure 6.4 and Figure 6.5 show results from our experiment with different input files. As seen in the figures, ICgrep outperforms pcregrep in every test, especially on the larger input file. The rate of increase of the processing time is less than the rate of increase of the number of rules, which demonstrates the scalability of ICgrep with a large set of regular expressions.

We applied multi-grep kernel to the spam rules. In this case, the Grep kernel is obviously the bottleneck in the pipeline. However, the performance improvement of data-pipeline parallelism is not only limited by the imbalanced workloads but also limited by the number of physical cores. As long as we divide the rules into groups so that none of the groups reaches one fourth of the entire workload, we do not expect stalls due to workload imbalance.

However, we did not randomly pick the number of regular expressions per kernel. Because if the group size is too small, we lose the advantage of optimizing the generated Pablo code, and if the group size is too big, the generated code may become so large that instruction cache misses become a problem.

Table 6.8 shows the profile of the 3027 rules run against SpamAssassin public corpus. As seen in this table, with a larger group size, the program executed fewer instructions but suffered a higher instruction cache misses rate. We choose 10 regular expressions per group and use this number in all of our experiments for spam rules.

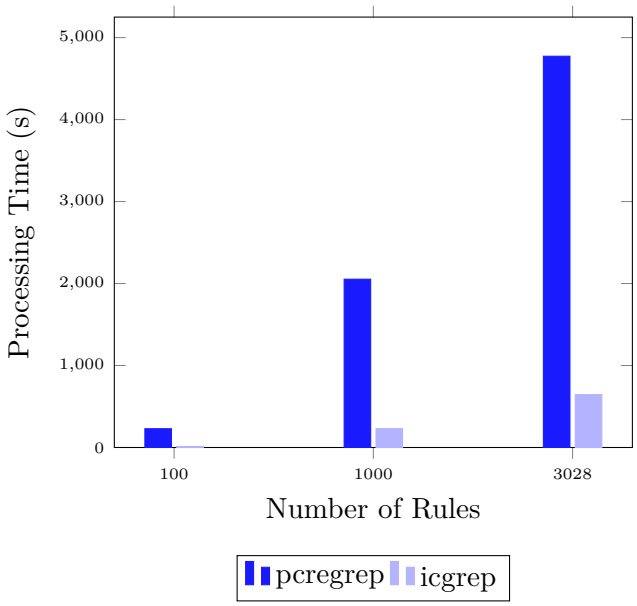

Figure 6.5: Enron Email Dataset (1.4 GB)

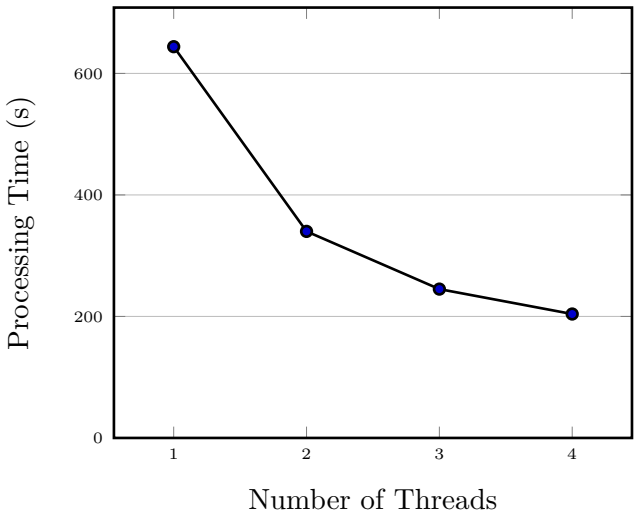

Figure 6.6: Spam Rules with Data-pipeline

	5 re/kernel	10 re/kernel	20 re/kernel	30 re/kernel
Instructions	147.8G	124.5G	111.8G	110.2G
I-Cache Misses	1.3G	1.7G	2.7G	3.3G
Processing Time	27.1s	26.8s	27.6s	32.8s

Table 6.8: Performance Analysis of Different Group Size

Figure 6.6 shows the results of running the whole spam rule set under data-pipeline mode for Enron email dataset. The rule set is divided into 303 groups, which means there are 310 kernels in the pipeline. ICgrep shows a good scalability with data-pipeline parallelism and is able to achieve 215% speedup with 4 threads on a quad-core machine.

Multithreading vs. SIMD

Parabix was first designed to take advantage of the SSE2 instruction set. ICgrep has shown performance benefit with AVX2, a newer instruction set architecture with larger register width. To find out the performance improvement that can be achieved by double the register size versus double the number of cores for applications built on Parabix framework, we ran some experiments that compare a multithreaded version (data-pipeline) of ICgrep using general registers (64-bit) with a sequential version that uses SSE2 instructions (128-bit).

Figure 6.7 shows the results of running the regular expression set in Table 6.3. As seen in this figure, using SSE2 instruction set outperform the multithreaded version with 2 threads and close to the result with 4 threads. However, in the figure that runs the experiments for the spam rules against Enron email dataset (Figure 6.8), SSE2 only achieves 10% speedup whereas the 2 threads data-pipeline version achieves 90% speedup.

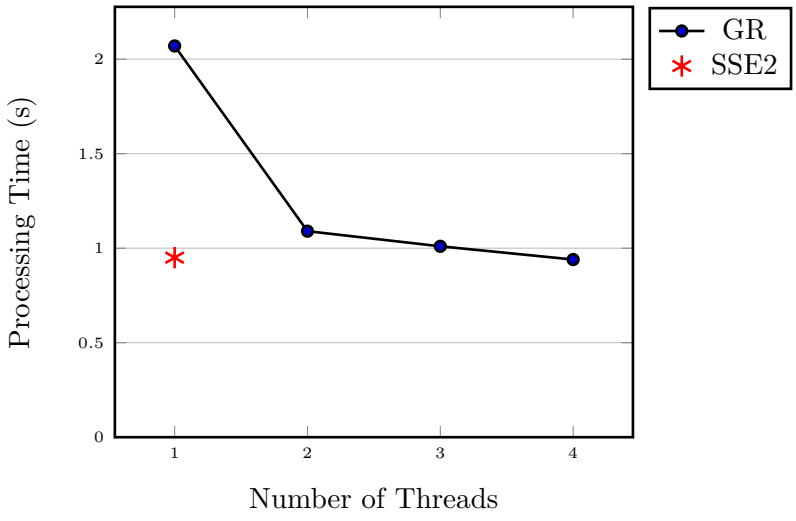

Figure 6.7: Table 6.3 Test Cases

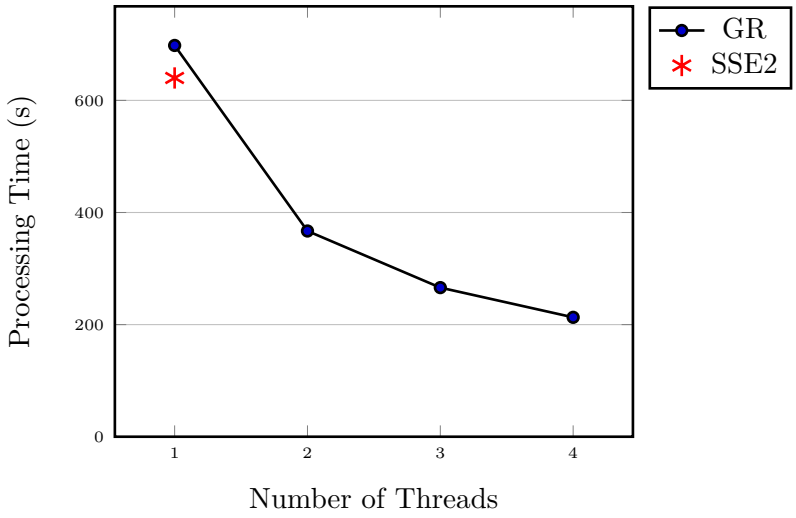

Figure 6.8: SpamAssassin Rules

To understand the huge differences between the two test sets, we measured the performance of each kernel using Parabix kernel cycle counter. Table 6.9 uses the same test cases as Figure 6.7 does. As shown in this table, LineBreak kernel, RequiredStream kernel and CC kernel is able to get about 100% speedup with SSE2 because they are mostly comprised of bitwise logic operations that can easily benefit from a larger register width. By taking advantage of the specialized SSE2 pack operation, S2P kernel is able to get more than 100% speedup. This is why we see a significant speedup even though the rest of the kernels cannot get 100% speedup.

However, the situation is different for spam rules. In this case, the cost of kernels other than CC and Grep is trivial and can hardly affect the performance speedup. Therefore, for ICgrep, the performance speedup of using SSE2 instead of general register is dependent on the workload of CC kernel and Grep kernel, which vary for different regular expression or regular expression set.

	Source	S2P	LineBreak	RequiredStream	CC	Grep	ScanMatch
GR	0.00	4.06	0.67	0.76	1.9	2.19	0.28
SSE2	0.00	0.91	0.32	0.38	0.83	1.86	0.23

Table 6.9: Kernel Profile (Cycles/Byte)

Task Parallelism with Multiple Input Files

We implement task parallelism for ICgrep so that the 15 input text files are processed in different threads. Task parallelism can also be affected by the imbalanced workloads. As shown in Figure 6.9, we can get an average of 65% speedup with 2 threads and 120%

speedup with 3 threads. However, increasing the number of threads from 3 to 4 does not give us much performance benefit. The reason is that one of the input text files is 196MB, which is about 1/3 of the entire input set. The thread that processes this big file becomes the bottleneck and we cannot further improve the performance even with more threads and more physical cores.

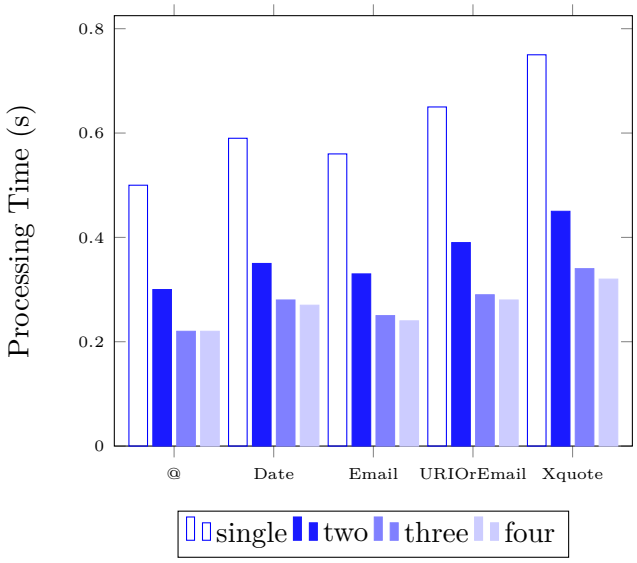

Figure 6.9: Task Parallelism with Mutiple Input Files

Task Parallelism + Data-pipeline Parallelism

As shown in Figure 6.2 and Figure 6.9, both task parallelism and data-pipeline parallelism could not fully utilized the CPU resources but for different reasons. One is caused by the imbalanced workloads of different kernels, the other one is caused by the imbalanced workloads of input files.

However, programmers can easily combine these two parallelization strategies with Parabix framework. Figure 6.10 shows the performance results of data-pipeline parallelism only (4 threads), task parallelism only (4 threads) and compares the results with the combination of data-pipeline parallelism (2 threads) plus task parallelism (4 threads). Further performance improvement is achieved for all of the test cases.

GPU Implementation

The GPU version of ICgrep is implemented using NVPTX driver provided by Parabix framework. The processing time of the GPU version consists of two parts: 1. The data transmission time that includes the cost of sending the input data from host to device and

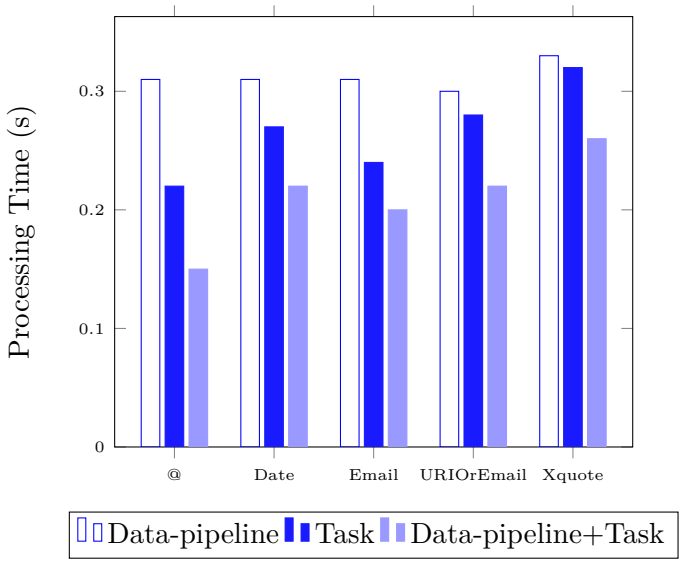

Figure 6.10: Task Parallelism + Data-pipeline Parallelism with Mutiple Input Files

the cost of retrieving the output data from device to host. 2. The kernel execution time. (Here, the kernel is referring to the routine compiled for GPU)

Figure 6.11 and Figure 6.12 show the performance comparison of the sequential CPU version versus the GPU version. In Figure 6.11, we use the five test cases in Table 6.3. This figure not only shows the processing time of the two versions but also provides the kernel execution time separately as GPU-K. As seen in this figure, the data transmission is about 0.21 second for a 620MB file, which is more than the kernel execution time of all the test cases. If we only count the kernel execution time, the GPU version achieves an average of 260% speedup. If we include the data transmission time, we can only get an average of 50% speedup.

Figure 6.12 shows the experiment results of spam rules. We use the smaller input text file SpamAssassin public corpus as we are limited by the global memory on GPU. The data transmission time for this text file is only 16ms, which is trivial compared to the kernel execution time. However, we do not see much benefit by increasing the amount of work per input file. We hit another obstacle, the instruction cache misses. Table 6.12 shows the sizes of the generated PTX files. With 200 spam rules, the PTX code size is about 13.6MB and cannot be handled by our experimental graphic card.

Number of Rules	10	50	100	200
Size	0.8MB	2.0MB	3.0MB	13.6MB

Table 6.10: PTX Code Size

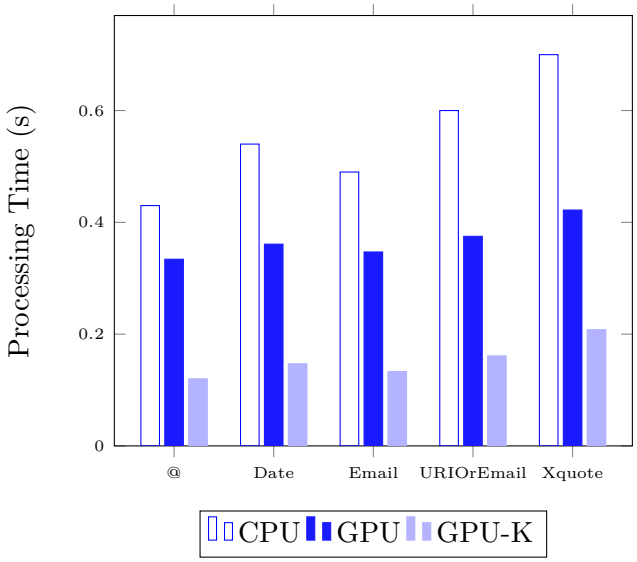

Figure 6.11: Single Regular Expressions on GPU

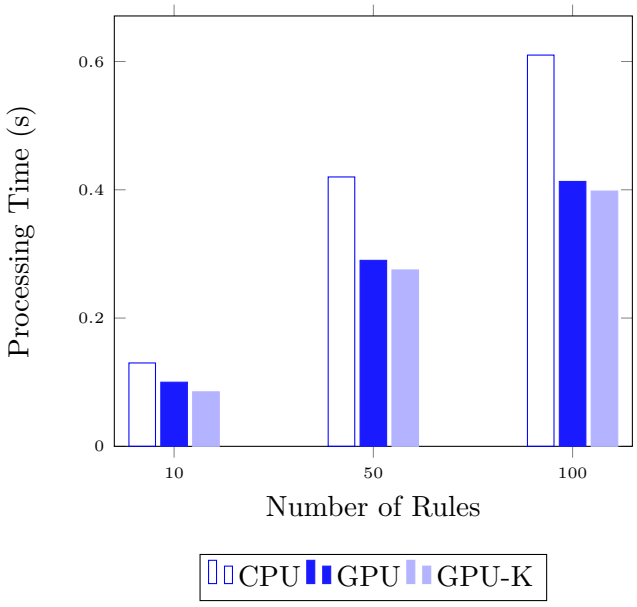

Figure 6.12: Spam Rules on GPU

6.3 Editd

Edit distance is used in approximate string matching problem to find all locations where a pattern matches a substring of a text with certain differences. It is a very important algorithm in bioinformatics problems and requires the fast computation. We developed a Parabix based application called Editd that can be used to map sequence reads to a reference genome within a given error rate. Editd consists of at least 3 kernels, CC kernel, Mapping kernel and Scan kernel. CC kernel generates the 4 Character Class bit stream for A, C, G, T. Mapping kernel runs the core algorithm that will be discussed later in this section. Scan kernel scans the match positions from the result bit stream computed by Mapping kernel.

6.3.1 Test Cases

The reference genome is downloaded from UCSC Genome Browser (`http://hgdownload.soe.ucsc.edu/goldenPath/hg19/bigZips/chromFa.tar.gz`). We choose a 250MB sequence for experiments on CPU and a 5MB sequence for experiments on GPU.

The sequence reads are selected from the Pacbio dataset (`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR306/SRR306842/SRR306842_1.fastq.gz`) in which the reads length varies from 35 base pairs to 35k base pairs. To find out how Editd scales with reads of different sizes, we select 3 reads that are 100 bps, 1000 bps, and 10000 bps.

An experiment for mapping to the human genome showed that 95% of 1000-base reads with a 15% error rate mapped to the correct location in the genome [15]. We choose $r = 15\%$ as our target error rate in our experiments to find the potential mappings. The edit distance d can then be calculated by $r \times m$ where m is the length of the reads to be mapped with.

6.3.2 Base Algorithm

A typical solution to avoid recursive computation is to use dynamic programming, which turns the problem into the calculation of an integer matrix that is used to store the distance between two substrings.

We use $C[i][j]$ to denote the edit distance between suffix of text $T[0...i-1]$ and pattern $P[0...j-1]$. Equation 6.1 gives the definition of $C[i][j]$.

$$C[i][j] = \begin{cases} j \text{ if } i = 0 \\ i \text{ if } j = 0 \\ \min \begin{cases} C[i-1][j-1] + \begin{cases} 0 \text{ if } T[i-1] = P[j-1] \\ 1 \text{ if } T[i-1] \neq P[j-1] \end{cases} \\ C[i][j-1] + 1 \\ C[i-1][j] + 1 \end{cases} \end{cases} \tag{6.1}$$

We develop an algorithm to compute the matrix with parallel bit streams. Let P_i be the bit stream marking text positions that match the character at pattern position i and $E_{i,j}$ be the bit stream marking matches after pattern position i with less or equal to j edits. $E_{i,j}$ can be calculated by the following equation 6.2.

$$E_{i+1,j} = (E_{i,j} >> 1 \wedge P_{i+1}) | (E_{i,j-1} >> 1 \wedge \neg P_{i+1})) | E_{i,j-1} | (E_{i+1,j-1} >> 1) \tag{6.2}$$

Suppose the length of the pattern is m, then all the match positions within edit distance d can be found by scanning bit stream $E_{m,d}$.

If $E_{i,d}$ becomes an empty bit stream, then $E_{i+1,d}$ must be an empty bit stream as well. In this case, there is no need to continue the calculation of the next pattern position. Therefore, the algorithm can be further optimized by adding a condition test to reduce the calculation after position i if an empty bit stream is found.

Algorithm 6 shows the pseudocode of the base algorithm with the optimization. AllOnes represents the bit stream that is fully marked.

Algorithm 6 Parallelized Edit Distance Calculation

```
1:  function ParabixEditDistance(P)
2:      E[0,0] = P_0
3:      if j > 0 then
4:          E[0,j] = AllOnes
5:      for i from 0 to m − 1 do
6:          E[i+1,0] = Advance(E[i,0]) ∧ P_{i+1}
7:          for j from 1 to d + 1 do
8:              E[i+1,j] = Advance(E[i,j]) ∧ P_{i+1}
9:              |Advance(E[i,j−1]) ∧ ¬P_{i+1}
10:             |E[i,j−1]
11:             |Advance(E[i+1,j−1])
12:         if E[i+1][d] == 0 then
13:             break
```

6.3.3 Extended Algorithm

The time complexity of algorithm 6 is affected by the edit distance d. To improve the performance of the base algorithm when a higher error rate is allowed, we build a two-level filter based on two observations.

1. If a pattern is divided into segments of size $(d'+1)/ErrorRate - 1$ and none of the segments can be mapped with edit distance less or equal than d', then there are no matches for the pattern with the given error rate (d' can be any integer that is smaller than d).

2. Within length of $PatternLen * (ErrorRate + 1)$ in reference string, the accumulated score V of each segment should be at least $TotalSegments \times (d' + 1) - PatternLen \times ErrorRate$. The score v of each segment is equal to d' subtracted by the edit distance at that candidate position.

As shown in Algorithm 2, the pattern is first divided into small segments based on observation 1. The first level filter maps each segment and the reference text with a small edit distance and then combines the results together.

In the second filter, we first calculate n, which represents the number of the segments that are required to match with the given edit distance. Then we can check if we have that many matches within length $PatternLen * (ErrorRate + 1)$.

To rule out more false positives, we can check whether those matches are in segment order. As a middle step, we can at least check if the matches are from unique segments. This part is not implemented yet.

Algorithm 7 Filter using Smaller Edit Distance

```
1:  function EditDistanceFilter(P)
2:      S ← PatternDivision(P)
3:      for all segment ∈ S do
4:          resultStream = resultStream or ParabixEditDistance(segment)
5:      matchPositions ← StreamScanner(resultStream)
6:      V = TotalSegments * d' − PatternLen * ErrorRate
7:      startPos = pos_0
8:      sum = v_0
9:      i = 0
10:     n = 0
11:     count = 0
12:     while i < TotalPositions do
13:         if pos_i − startPos > PatternLen * (ErrorRate + 1) then
14:             sum = sum + v_i
15:             i = i + 1
16:         else
17:             if sum > V then
18:                 count = count + 1
19:             sum = sum − v_n
20:             n = n + 1
21:             startPos = pos_n
```

6.3.4 Editd-I

We build a pattern compiler based on algorithm 6 to generate Pablo code and then compile the Pablo code to Pablo kernel in LLVM IR. For the extended algorithm, we have to generate one Mapping kernel for every pattern segment.

For example, suppose we have 4 character class bit streams A, C, G, T computed by CC compiler. The Pablo code that calculates E[1][0] for pattern segment `AC` is:

```
e_1_0 = (C & pablo.Advance(A, 1))
```

For a different pattern segment `GT`, the code generated by pattern compiler is different:

```
e_1_0 = (T & pablo.Advance(G, 1))
```

As mentioned in the previous section, we have limited resource on GPU and the program size becomes an obstacle for the GPU version. Therefore, we implement Editd-I that uses the same Mapping kernel for all the patterns. We store the 4 character class bit streams A, C, T, G into CC array and use the sixth and seventh bits of character A, C, T, G as the index of CC array. Then we can calulate E[1][0] with the code shown below:

```
e_1_0 = (CC[(p[1] >> 1) & 0x3] & pablo.Advance(CC[(p[0] >> 1) & 0x3], 1))
```

However, this code is not supported by Pablo compiler. We have to use the kernel builder to implement the base algorithm and manually handle the carries for the advance functions.

6.3.5 Results and Analysis

Base Algorithm vs. Myers

Figure 6.13 shows the performance comparison between Myers and Editd. In this experiment, we use the 100 bps read and test it with different edit distance. As seen in this figure, Editd outperforms Myers' algorithm when edit distance is smaller than 12 but is slower than Myers' for a larger edit distance. The performance of Myers' algorithm is unrelated to edit distance. But as we can see from algorithm 6, the time complexity of Editd is $O(md)$ where $d = m \times error\_rate$. Therefore the processing time of Editd grows exponentially as we increase the edit distance d. To improve the performance of Editd on larger edit distance, we developed the extended algorithm to filter out a large portion of the positions by running the base algorithm on pattern segments with smaller editd distance.

Grouping by Prefix

In the extended algorithm, the reads are partitioned into pattern segments. Table 6.11 gives the number of pattern segments for the 3 test reads when $d' = 2$. Since the Pablo compiler is able to optimize Pablo code by reducing common subexpressions, we can group the pattern segments by their prefix to improve the performance of the extended algorithm. Table 6.11 shows the number of groups when different prefix length is applied.

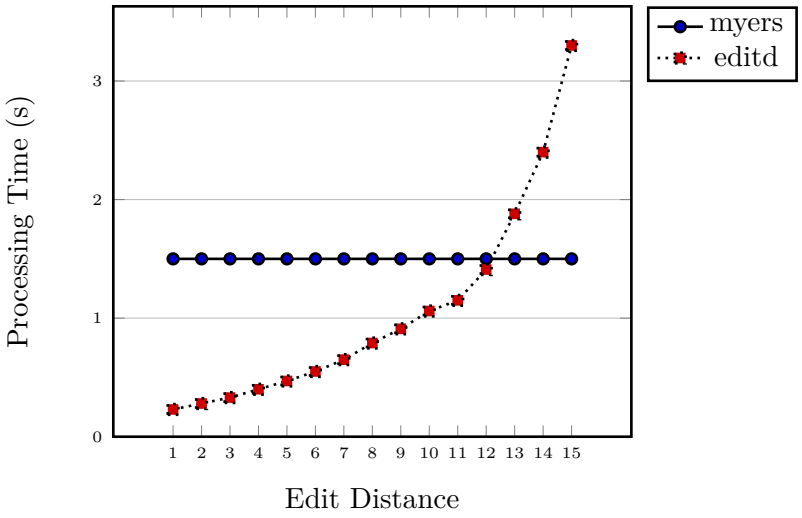

Figure 6.13: Comparison with Myers

	100 bps	1000 bps	10000 bps
Number of Pattern Segments	4	52	526
Total Groups (prefix = 1)	3	4	4
Total Groups (prefix = 2)	4	15	16
Total Groups (prefix = 3)	4	34	64
Total Groups (prefix = 4)	4	49	201
Total Groups (prefix = 5)	4	50	390

Table 6.11: Grouping by Prefix

Table 6.12 shows the performance comparison between the base algorithm and the extented algorithm of Editd with error rate 15%, that is $d = 15$ for the 100 bps read. The extended algorithm uses $d' = 2$ and achieves 450% speedup compared with the base algorithm. In the test read, two pattern segments have the same prefix when prefix length is 1. By grouping these two pattern segments, Editd is able to gain another 8% speedup. For the 100 bps read, there is no benefit of further increasing the prefix length.

	Base	Extended	Extended (prefix=1)
Processing Time (s)	3.3	0.60	0.55

Table 6.12: Performance Improvement of the Extended Algorithm

For longer reads, prefix length can change the grouping results significantly as shown in Table 6.11. The longer the prefix length is, the fewer pattern segments we can have in a group and the less optimization we can get from Pablo compiler. However, shorter prefix length does not necessarily mean better performance.

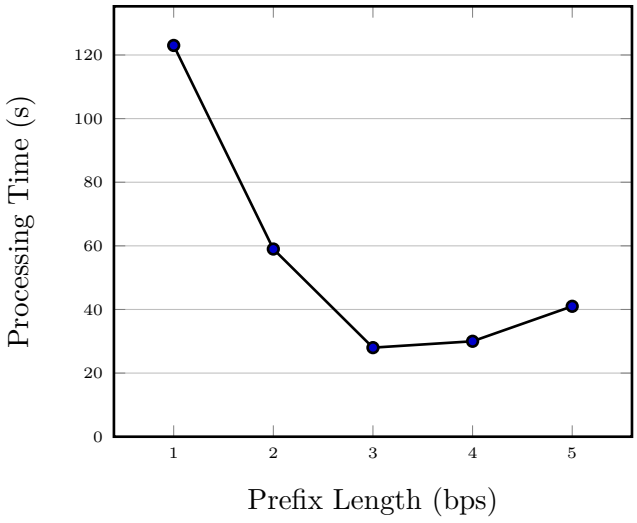

Figure 6.14: Extended Algorithm with Different Prefix Length

Figure 6.14 shows the experiment results of the extended algorithm with different prefix length. As seen in this figure, the best performance is achieved by prefix length 3. The reason that we see a substantial performance degradation rather than improvement for prefix length 1 and 2 is because of the instruction cache misses.

Table 6.13 gives the number of instruction cache misses measured by `perf`. For prefix length 1, the number of pattern segments per group is over a hundred; the LLVM IR generated for each Mapping kernel is over 80k lines whereas the instruction cache size is only 32k. We use prefix length 3 in the rest of our experiments. Note that this might not be the optimal choice for all the reads in Pacbio dataset and users can choose different prefix length with command line option `-prefix`.

prefix=1	prefix=2	prefix=3	prefix=4	prefix=5
22G	15G	0.6G	0.3G	0.3G

Table 6.13: Instruction Cache Misses

Multi-pipeline vs Single-pipeine

We have two implementations for Editd. One is creating a single pipeline that has multiple kernels to process different pattern segments. Another is creating multiple pipelines so that the pattern segments can be processed in different pipelines. The last one can be easily combined with task parallelism. Both implementations can be run under data-pipeline mode. However, since Editd only has 3 basis kernels with multi-pipeline implementation, it may not benefit much from data-pipeline parallelism.

The single-pipeline implementation may suffer the penalty of instruction cache misses especially with a large number of pattern segments. This issue can be solved by increasing the buffer size. Buffer size can be set by users as a command line option to decide the number of blocks processed by every kernel in one iteration.

The multi-pipeline implementation does not have the instruction cache problem as it processes one group of pattern segments for the entire input text before processing the next group.

Figure 6.15 shows the performance results of the two implementations with different buffer sizes. We use buffer size 128 blocks in all of our experiments for Editd and Editd-I.

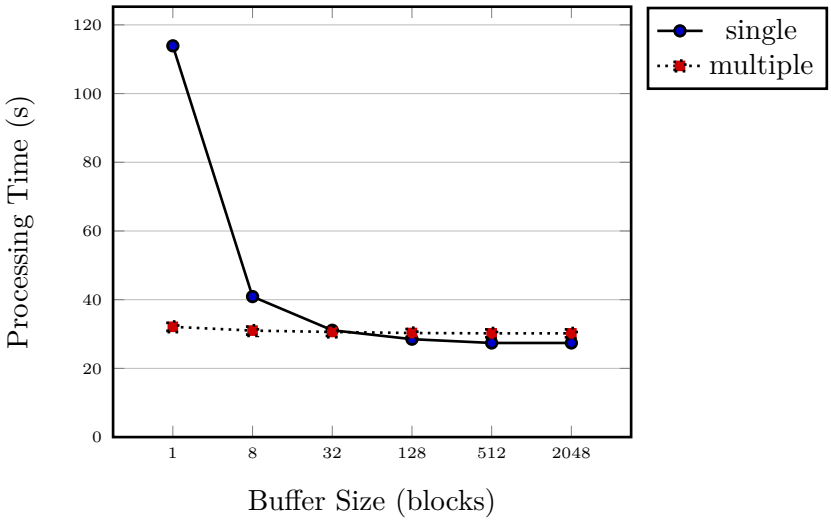

Figure 6.15: Effect of Buffer Size

Extended Algorithm vs. Myers

Figure 6.16 shows the performance results of Editd using the extended algorithm compared with Myers' algorithm at error rate 15%. For the 3 test reads of size 100 bps, 1000 bps, and 10000 bps, Editd achieves an average of 160% speedup and shows a good scalability.

Task Parallelsim vs. Data-pipeline Paralellism

We apply task parallelism based on the multi-pipeline implementation and data-pipeline parallelism based on the multi-pipeline implementation. Figure 6.17 and Figure 6.18 shows the performance comparison of task parallelsim versus data-pipeline parallelsim. They achieve similar performance speedup for reads of different sizes. However, data-pipeline is an integrated mode, that means no extra programming is required to get the performance improvement. Task parallelism also requires the program to have independent tasks whereas data-pipeline parallelism can handle any inherently sequential applications.

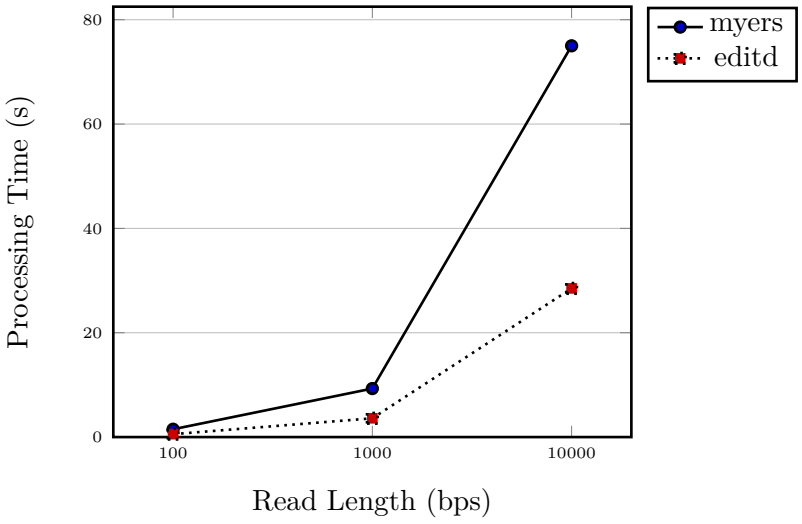

Figure 6.16: Comparison with Myers (r=15%)

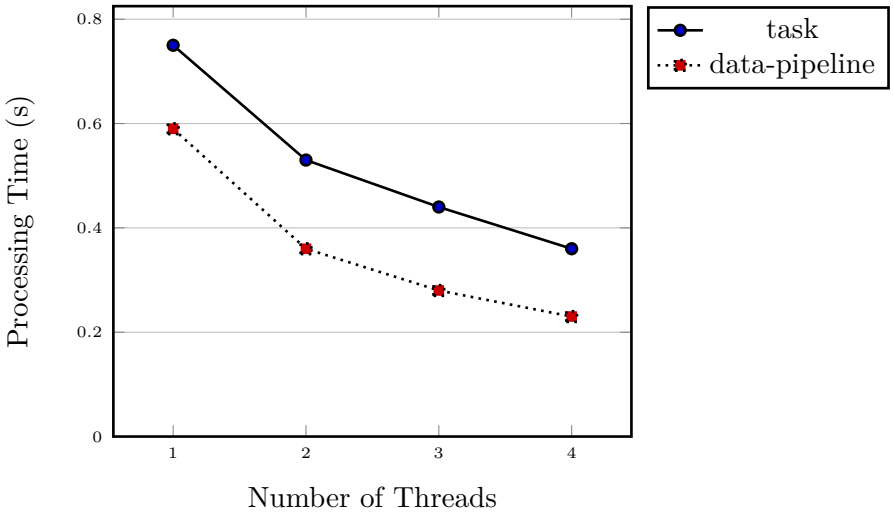

Figure 6.17: Task Parallelism vs Data-pipeline Parallelism (100 bps)

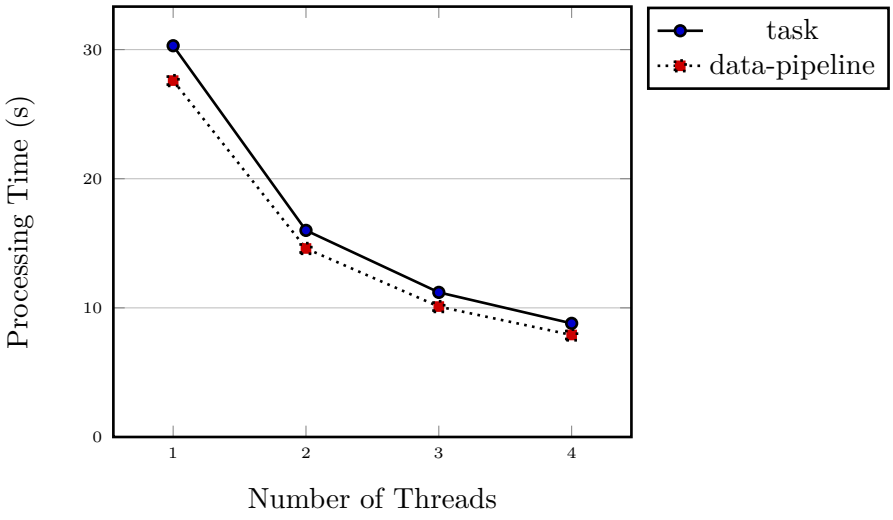

Figure 6.18: Task Parallelism vs Data-pipeline Parallelism (10000 bps)

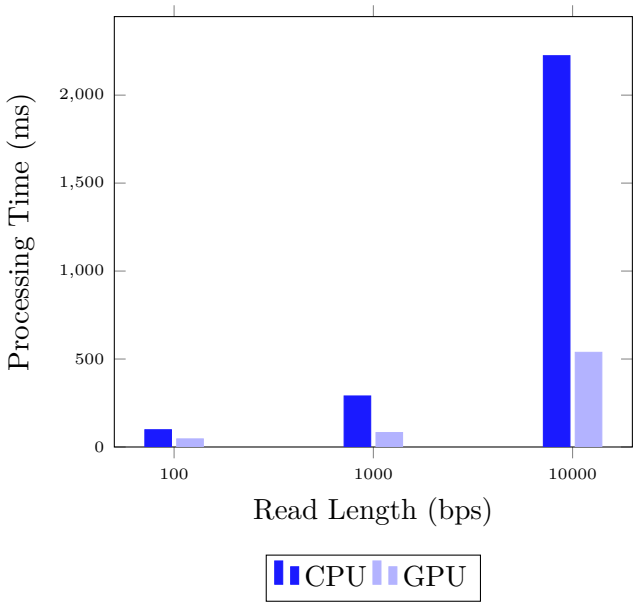

Figure 6.19: Editd on GPU

GPU Implementation

Figure 6.19 shows the performance results of the GPU version of Editd-I compared with the CPU version. The number of work groups depends on the number of pattern segments. Each work group process one group of pattern segments with 64 threads. For the reads of 100 bps, there are only 4 pattern segments and thus only 4 work groups are executed simultaneously. The GPU resources are not fully utilized, and we only see a 110% speedup. For the read of 1000 bps and 10000 bps the GPU version is able to achieve 250% and 310% speedup respectively.

Different from Editd, which is dynamically optimized by Pablo compiler, Editd-I is not fully optimized and could be 20% to 50% slower than Editd. However, in this experiment, we can see that, for compute-intensive applications, which are less affected the extra data transmission time, the GPU version can achieve a substantial improvement if we can smartly using the limited resources.

6.4 U8U16

U8U16 is a UTF8 to UTF16 transcoder based on Parabix techniques. The initial version has shown a significant performance improvement compared with the conventional byte-at-a-time implementations [8].

The latest version implemented with Parabix framework consists of 6 kernels, Source kernel, S2P kernel, U8U16 kernel, Del kernel, P2S kernel and Stdout kernel. The profile of each kernel is shown in Table 6.14. The estimated minimum processing time under data-pipeline mode calculated by equation 4.2 is 50% of T_s for 2 threads, 37% of T_s for 3 threads and 37% of T_s for 4 threads where T_s is the sequential processing time of U8U16. Figure 6.20 shows the actual processing time measured with different number of threads. Using 3 threads and 4 threads gives about the same performance speedup of 150%, which matches the estimated results.

Source	S2P	U8U16	Del	P2S	Stdout
0.00	0.99	1.31	1.87	1.03	3.06

Table 6.14: Kernel Profile (Cycles/Byte)

We also measured the performance of U8U16 under the pipeline mode, which gives only 10% speedup as shown in Table 6.15. The same test cases are processed by iconv, which a command-line program that is used to convert between different character encodings. We can see about 90% speedup of U8U16 over iconv.

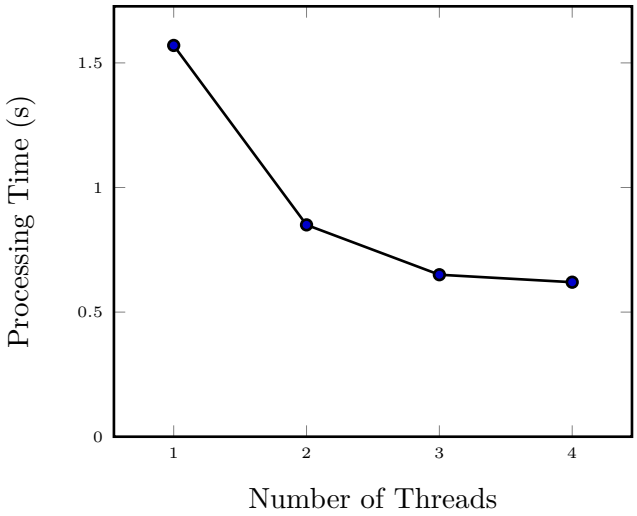

Figure 6.20: u8u16 Parallelization

	iconv	U8U16	data-pipeline	pipeline
Processing Time	2.98s	1.57s	0.62s	1.43s

Table 6.15: Performance Comparisons of U8U16

6.5 Base64

Base64 is an encoding scheme that represents binary data by 64 printable ASCII characters. Every 6 bit in the binary data is corresponding to one of the printable characters. We developed a base64 encoder using Parabix framework and measured the performance of the 5 kernels as shown in Table 6.16.

Source	expand	radix64	base64	Stdout
0.00	0.47	3.69	3.32	1.60

Table 6.16: Kernel Profile (Cycles/Byte)

Compared with the base64 program provided by GNU on Linux system, our encoder is slightly slower. However, with the multi-threading support, we can easily get about 110% speedup with both task parallelism and data-pipeline parallelism. Table 6.17 shows the performance results of GNU's Base64 program and our Based64 under different parallelization modes.

Figure 6.21 shows the experiment results of task parallelism and data-pipeline parallelism with different number of threads. Because of the imbalanced workloads, they both

	Base64 (GNU)	Base64 (Parabix)	data-pipeline	pipeline	task
Processing Time	2.01s	2.24s	1.05s	1.65s	1.08s

Table 6.17: Performance Comparisons of Base64

achieve the best performance with 3 threads and cannot get more benefit by assigning more threads to the program.

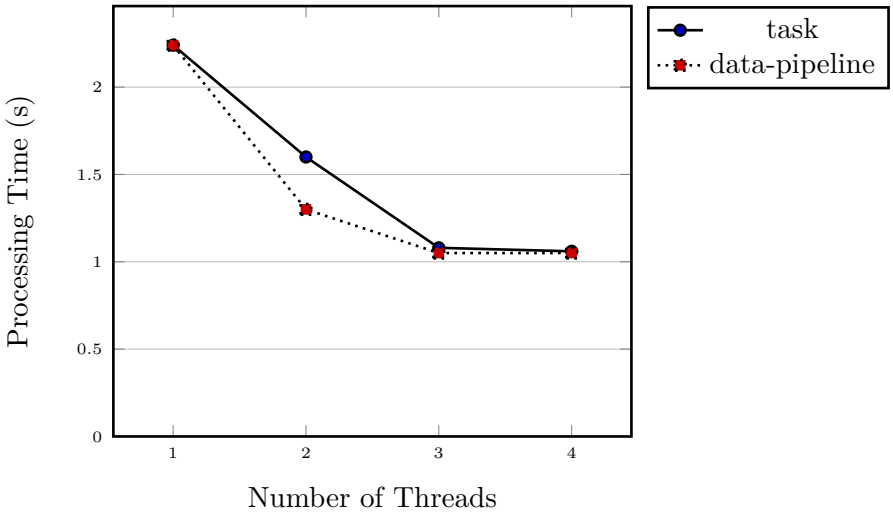

Figure 6.21: Base64 Parallelization

Chapter 7

Conclusion

7.1 Conclusion

Parabix framework is a high performance programming model for streaming text processing applications. SIMD parallelization has been exploited and shown significant performance improvement over the conventional byte-at-a-time implementations. This dissertation investigates multidimensional parallelization that combines Parabix with multithreading on commodity multiprocessors as well as the specialized accelerators GPU. We demonstrated promising performance results with the multithreading supports on the 4 applications, ICgrep, Editd, U8U16 and Base64. The data-pipeline parallelization mode has shown 70% to 215% performance speedup and good scalability. The GPU extension has some limitations and may not benefit all the applications but has shown substantial improvement with compute-intensive implementations. The multi-threading support and GPU extension are also programming friendly, especially for the integrated mode, which does not need any efforts from the programmers.

7.2 Future Work

The current Parabix framework support applications running on either CPU or GPU. There is a potential benefit of distributing the workloads to process data on both CPU and GPU. For example, in ICgrep, when multiple input files are needed to be processed, they can be assigned to different devices (CPU/GPU) and processed in parallel. Another improvement that can be done is optimizing the implementation of the `Addition` Algorithm for GPU. As we discussed in Section 3.2, the parallel-prefix style process can be replaced by ballot function, which already exists in the current IDISA NVPTX Builder but has not been applied yet. As the ballot function only gathers bits from one warp (32 threads) and our work group uses 64 threads, we need to combine the results of two ballot function calls. Data-pipeline parallelization mode can also be improved. For example, we can remove

the thread synchronization test for those data independent kernels as discussed in Section 4.1.2.

Bibliography

[1] A comprehensive list of big data statistics. `http://wikibon.org/blog/big-data-statistics/`.

[2] Nvvm ir specification. `https://wrf.ecse.rpi.edu//wiki/ParallelComputingSpring2015/cuda/nvidia/doc/pdf/`.

[3] Twitter usage statistics. `http://www.internetlivestats.com/twitter-statistics/`.

[4] User guide for nvptx back-end. `http://llvm.org/docs/NVPTXUsage.html`.

[5] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[6] Christiam Camacho, George Coulouris, Vahram Avagyan, Ning Ma, Jason Papadopoulos, Kevin Bealer, and Thomas L Madden. Blast+: architecture and applications. *BMC bioinformatics*, 10(1):421, 2009.

[7] Rob Cameron, Ken Herdy, and Ehsan Amiri. Parallel bit stream technology as a foundation for XML parsing performance. In *International Symposium on Processing XML Efficiently: Overcoming Limits on Space, Time, or Bandwidth*, volume 8, 2009.

[8] Robert D Cameron. A case study in SIMD text processing with parallel bit streams: UTF-8 to UTF-16 transcoding. In *13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 91–98. ACM, 2008.

[9] Robert D Cameron, Ehsan Amiri, Kenneth S Herdy, Dan Lin, Thomas C Shermer, and Fred P Popowich. Parallel scanning with bitstream addition: An XML case study. In *Euro-Par 2011 Parallel Processing*, pages 2–13. Springer, 2011.

[10] Robert D Cameron, Kenneth S Herdy, and Dan Lin. High performance XML parsing using parallel bit stream technology. In *2008 Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, page 17. ACM, 2008.

[11] Robert D Cameron and Dan Lin. Architectural support for SWAR text processing with parallel bit streams: the inductive doubling principle. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 337–348. ACM, 2009.

[12] Robert D Cameron, Nigel Medforth, Dan Lin, Dale Denis, and William N Sumner. Bitwise data parallelism with llvm: The icgrep case study. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 373–387. Springer, 2015.

[13] Robert D Cameron, Thomas C Shermer, Arrvindh Shriraman, Kenneth S Herdy, Dan Lin, Benjamin R Hull, and Meng Lin. Bitwise data parallelism in regular expression matching. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 139–150. ACM, 2014.

[14] Alejandro Chacón, Santiago Marco-Sola, Antonio Espinosa, Paolo Ribeca, and Juan Carlos Moure. Thread-cooperative, bit-parallel computation of levenshtein distance on gpu. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 103–112. ACM, 2014.

[15] Mark J Chaisson and Glenn Tesler. Mapping single molecule sequencing reads using basic local alignment with successive refinement (blasr): application and theory. *BMC bioinformatics*, 13(1):238, 2012.

[16] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using cuda. *Journal of parallel and distributed computing*, 68(10):1370–1380, 2008.

[17] Linchuan Chen, Xin Huo, and Gagan Agrawal. Accelerating mapreduce on a coupled cpu-gpu architecture. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 25. IEEE Computer Society Press, 2012.

[18] Gilberto Contreras and Margaret Martonosi. Characterizing and improving the performance of intel threading building blocks. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 57–66. IEEE, 2008.

[19] Zefu Dai, Nick Ni, and Jianwen Zhu. A 1 cycle-per-byte xml parsing accelerator. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 199–208. ACM, 2010.

[20] Michael I Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *ACM SIGOPS Operating Systems Review*, 40(5):151–162, 2006.

[21] Naga K Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 215–226. ACM, 2004.

[22] Jan Holub and Stanislav Stekr. On parallel implementations of deterministic finite automata. In *CIAA*, volume 5642, pages 54–64. Springer, 2009.

[23] Nicolas Hulo, Amos Bairoch, Virginie Bulliard, Lorenzo Cerutti, Edouard De Castro, Petra S Langendijk-Genevaux, Marco Pagni, and Christian JA Sigrist. The prosite database. *Nucleic acids research*, 34(suppl 1):D227–D230, 2006.

[24] David H Jones, Adam Powell, Christos-Savvas Bouganis, and Peter YK Cheung. Gpu versus fpga for high productivity computing. In *2010 International Conference on Field Programmable Logic and Applications*, pages 119–124. IEEE, 2010.

[25] Sol Ji Kang, Sang Yeon Lee, and Keon Myung Lee. Performance comparison of openmp, mpi, and mapreduce in practical problems. *Advances in Multimedia*, 2015:7, 2015.

[26] Marco Keller. Llvm to ptx backend. 2011.

[27] Yousun Ko, Minyoung Jung, Yo-Sub Han, and Bernd Burgstaller. A speculative parallel dfa membership test for multicore, simd and cloud computing environments. *arXiv preprint arXiv:1210.5093*, 2012.

[28] Margaret G Kostoulas, Morris Matsa, Noah Mendelsohn, Eric Perkins, Abraham Heifets, and Martha Mercaldi. Xml screamer: an integrated approach to high performance xml parsing, validation and deserialization. In *Proceedings of the 15th international conference on World Wide Web*, pages 93–102. ACM, 2006.

[29] I Lee, Ting Angelina, Charles E Leiserson, Tao B Schardl, Zhunping Zhang, and Jim Sukha. On-the-fly pipeline parallelism. *ACM Transactions on Parallel Computing*, 2(3):17, 2015.

[30] Chunhua Liao, Yonghong Yan, Bronis R De Supinski, Daniel J Quinlan, and Barbara Chapman. Early experiences with the openmp accelerator model. In *International Workshop on OpenMP*, pages 84–98. Springer, 2013.

[31] Dan Lin, Nigel Medforth, Kenneth S Herdy, Arrvindh Shriraman, and Rob Cameron. Parabix: Boosting the efficiency of text processing on commodity processors. In *18th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12. IEEE, 2012.

[32] Jimmy Lin and Chris Dyer. Data-intensive text processing with mapreduce. *Synthesis Lectures on Human Language Technologies*, 3(1):1–177, 2010.

[33] Mian Lu, Lei Zhang, Huynh Phung Huynh, Zhongliang Ong, Yun Liang, Bingsheng He, Rick Siow Mong Goh, and Richard Huynh. Optimizing the mapreduce framework on intel xeon phi coprocessor. In *Big Data, 2013 IEEE International Conference on*, pages 125–130. IEEE, 2013.

[34] Wei Lu, Kenneth Chiu, and Yinfei Pan. A parallel approach to xml parsing. In *2006 7th IEEE/ACM International Conference on Grid Computing*, pages 223–230. IEEE, 2006.

[35] Daniel Luchaup, Randy Smith, Cristian Estan, and Somesh Jha. Multi-byte regular expression matching with speculation. In *RAID*, volume 9, pages 284–303. Springer, 2009.

[36] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 45–55. IEEE, 2009.

[37] Svetlin A Manavski and Giorgio Valle. Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC bioinformatics*, 9(2):S10, 2008.

[38] Michael D McCool, Arch D Robison, and James Reinders. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[39] Kenneth Moreland and Edward Angel. The fft on a gpu. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119. Eurographics Association, 2003.

[40] Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM (JACM)*, 46(3):395–415, 1999.

[41] Hoa Nguyen and Dominique Lavenier. Plast: parallel local alignment search tool for database comparison. *BMC bioinformatics*, 10(1):329, 2009.

[42] Arunmoezhi Ramachandran, Jerome Vienne, Rob Van Der Wijngaart, Lars Koesterke, and Ilya Sharapov. Performance evaluation of nas parallel benchmarks on intel xeon phi. In *2013 42nd International Conference on Parallel Processing*, pages 736–743. IEEE, 2013.

[43] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. Ieee, 2007.

[44] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. Parallelizing user-defined aggregations using symbolic execution. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 153–167. ACM, 2015.

[45] Bin Ren, Gagan Agrawal, James R Larus, Todd Mytkowicz, Tomi Poutanen, and Wolfram Schulte. Simd parallelization of applications that traverse irregular data structures. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–10. IEEE, 2013.

[46] Mitsuhisa Sato. Openmp: parallel programming api for shared memory multiprocessors and on-chip multiprocessors. In *Proceedings of the 15th international symposium on System Synthesis*, pages 109–111. ACM, 2002.

[47] Michael C Schatz, Cole Trapnell, Arthur L Delcher, and Amitabh Varshney. High-throughput sequence alignment using graphics processing units. *BMC bioinformatics*, 8(1):474, 2007.

[48] Dirk Schmidl, Tim Cramer, Sandra Wienke, Christian Terboven, and Matthias S Müller. Assessing the performance of openmp programs on the intel xeon phi. In *European Conference on Parallel Processing*, pages 547–558. Springer, 2013.

[49] Bhavik Shah, Praveen R Rao, Bongki Moon, and Mohan Rajagopalan. A data parallel algorithm for xml dom parsing. In *International XML Database Symposium*, pages 75–90. Springer, 2009.

[50] Reetinder Sidhu. High throughput, tree automata based xml processing using fpgas. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 74–81. IEEE, 2013.

[51] Reetinder Sidhu and Viktor K Prasanna. Fast regular expression matching using fpgas. In *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*, pages 227–238. IEEE, 2001.

[52] Ryoma Sinya, Kiminori Matsuzaki, and Masataka Sassa. Simultaneous finite automata: An efficient data-parallel model for regular expression matching. In *2013 42nd International Conference on Parallel Processing*, pages 220–229. IEEE, 2013.

[53] Euripides Sotiriades, Christos Kozanitis, and Apostolos Dollas. Fpga based architecture for dna sequence comparison and database search. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 8–pp. IEEE, 2006.

[54] Jie Tang, Shaoshan Liu, Chen Liu, Zhimin Gu, and Jean-Luc Gaudiot. Acceleration of xml parsing through prefetching. *IEEE Transactions on Computers*, 62(8):1616–1628, 2013.

[55] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, pages 179–196. Springer, 2002.

[56] Antonino Tumeo and Oreste Villa. Accelerating dna analysis applications on gpu clusters. In *Application Specific Processors (SASP), 2010 IEEE 8th Symposium on*, pages 71–76. IEEE, 2010.

[57] Panagiotis D Vouzis and Nikolaos V Sahinidis. Gpu-blast: using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, 27(2):182–188, 2011.

[58] Guibin Wang and Xiaoguang Ren. Power-efficient work distribution method for cpu-gpu heterogeneous system. In *International Symposium on Parallel and Distributed Processing with Applications*, pages 122–129. IEEE, 2010.

[59] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. Gpu-based nfa implementation for memory efficient high speed regular expression matching. In *ACM SIGPLAN Notices*, volume 47, pages 129–140. ACM, 2012.

Appendix A

Example of Code Generation for Editd

Appendix A shows a portion of the generated code selected from one of the Parabix applications Editd, which has been discussed in Section 6.3. The following code segments are dynamically compiled and generated for a simple case that matches reference genome with sequence "ACGT" for edit distance 2.

A.1 Pablo code of Editd

This is the Pablo code generated by Pattern Compiler based on Algorithm 6 Programmers can choose to display the Pablo code with a command line option `-ShowPablo` provided by Parabix framework.

```
1  advance = pablo.Advance(CC_A,a, 1)
2  e_1_0 = (CC_C,c & advance)
3  advance_1 = pablo.Advance(1, 1)
4  and = (CC_C,c & advance_1)
5  not = (~CC_C,c)
6  and_1 = (advance & not)
7  advance_2 = pablo.Advance(e_1_0, 1)
8  or = (CC_A,a | advance_2)
9  or_1 = (and | and_1)
10 e_1_1 = (or | or_1)
11 and_2 = (advance_1 & not)
12 advance_3 = pablo.Advance(e_1_1, 1)
13 e_2_0 = (CC_G,g & advance_2)
14 and_3 = (CC_G,g & advance_3)
15 not_1 = (~CC_G,g)
16 and_4 = (advance_2 & not_1)
17 advance_4 = pablo.Advance(e_2_0, 1)
18 or_2 = (e_1_0 | advance_4)
19 or_3 = (and_3 | and_4)
20 e_2_1 = (or_2 | or_3)
21 and_5 = (CC_G,g & advance_1)
22 and_6 = (advance_3 & not_1)
23 advance_5 = pablo.Advance(e_2_1, 1)
24 or_4 = (e_1_1 | advance_5)
25 or_5 = (and_5 | and_6)
26 e_2_2 = (or_4 | or_5)
27 e_3_0 = (CC_T,t & advance_4)
```

```
and_7 = (CC_T,t & advance_5)
not_2 = (~CC_T,t)
and_8 = (advance_4 & not_2)
advance_6 = pablo.Advance(e_3_0, 1)
or_6 = (e_2_0 | advance_6)
or_7 = (and_7 | and_8)
e_3_1 = (or_6 | or_7)
advance_7 = pablo.Advance(e_2_2, 1)
and_9 = (CC_T,t & advance_7)
and_10 = (advance_5 & not_2)
advance_8 = pablo.Advance(e_3_1, 1)
or_8 = (e_2_1 | advance_8)
or_9 = (and_9 | and_10)
e_3_2 = (or_8 | or_9)
not_3 = (~e_3_0)
and_11 = (e_3_1 & not_3)
not_4 = (~e_3_1)
and_12 = (e_3_2 & not_4)
```

A.2 LLVM IR of Editd Kernel

This is the generated LLVM IR code of Editd kernel that consists of the kernel struct (line 1), the initialization function (line 4-14) and the segment processing function (line 17-211). Programmers can choose to display the LLVM IR code with a command line option `-ShowIR` provided by Parabix framework.

```
1  %"editd" = type { { <2 x i64>, <2 x i64>, <2 x i64>, <2 x i64>, <2 x i64>, <2 x i64>, <2 x i64>, <2 x i64>, <2 x i64> }, i64, [4 x <2 x i64>]*, [3 x <2 x i64>]*, <2 x i64>, <2 x i64>, { i64, i64** }*, i64, i1, i64, i64 }

3  ; Function Attrs: nounwind
4  define void @"editd_Init"(%"editd"* %self, [4 x <2 x i64>]* %pat_bufferPtr, [3 x <2 x i64>]* %E_bufferPtr, { i64, i64** }* %EConsumerLocks) #0 {
5  entry:
6    store %"editd" zeroinitializer, %"editd"* %self, align 16
7    %0 = getelementptr %"editd", %"editd"* %self, i64 0, i32 2
8    store [4 x <2 x i64>]* %pat_bufferPtr, [4 x <2 x i64>]** %0, align 8
9    %1 = getelementptr %"editd", %"editd"* %self, i64 0, i32 3
10   store [3 x <2 x i64>]* %E_bufferPtr, [3 x <2 x i64>]** %1, align 8
11   %2 = getelementptr %"editd", %"editd"* %self, i64 0, i32 6
12   store { i64, i64** }* %EConsumerLocks, { i64, i64** }** %2, align 8
13   ret void
14 }

16 ; Function Attrs: nounwind
17 define void @"editd_DoSegment"(%"editd"* nocapture %self, i1 %doFinal, i64 %patAvailableItems) #0 {
18 entry:
19   %0 = select i1 %doFinal, i8* blockaddress(@"editd_DoSegment", %"editd_doFinalBlock"), i8* blockaddress(@"editd_DoSegment", %"editd_segmentDone")
20   %1 = getelementptr %"editd", %"editd"* %self, i64 0, i32 1
21   %pat_processedItemCount = load i64, i64* %1, align 4
22   %2 = sub i64 %patAvailableItems, %pat_processedItemCount
23   %3 = lshr i64 %2, 7
24   br label %"editd_strideLoopCond"

26 "editd_strideLoopCond": ; preds = %"editd_strideLoopBody", %entry
27   %pat_processedItemCount1 = phi i64 [ %pat_processedItemCount, %entry ], [ %132, %"editd_strideLoopBody" ]
28   %branchTarget = phi i8* [ %0, %entry ], [ %strideTarget, %"editd_strideLoopBody" ]
29   %stridesRemaining = phi i64 [ %3, %entry ], [ %133, %"editd_strideLoopBody" ]
30   %4 = icmp sgt i64 %stridesRemaining, 0
31   br i1 %4, label %"editd_strideLoopBody", label %"editd_stridesDone", !prof !0

33 "editd_strideLoopBody": ; preds = %"editd_doFinalBlock", %"editd_strideLoopCond"
34   %strideTarget = phi i8* [ %branchTarget, %"editd_strideLoopCond" ], [ blockaddress(@"editd_DoSegment", %resume), %"editd_doFinalBlock" ]
35   %5 = lshr i64 %pat_processedItemCount1, 7
36   %6 = getelementptr %"editd", %"editd"* %self, i64 0, i32 2
37   %pat_bufferPtr = load [4 x <2 x i64>]*, [4 x <2 x i64>]** %6, align 8
38   %7 = and i64 %5, 1
39   %8 = getelementptr [4 x <2 x i64>], [4 x <2 x i64>]* %pat_bufferPtr, i64 %7, i64 0
40   %9 = getelementptr [4 x <2 x i64>], [4 x <2 x i64>]* %pat_bufferPtr, i64 %7, i64 1
41   %10 = getelementptr [4 x <2 x i64>], [4 x <2 x i64>]* %pat_bufferPtr, i64 %7, i64 2
42   %11 = getelementptr [4 x <2 x i64>], [4 x <2 x i64>]* %pat_bufferPtr, i64 %7, i64 3
43   %12 = load <2 x i64>, <2 x i64>* %8, align 16
44   %13 = getelementptr %"editd", %"editd"* %self, i64 0, i32 0, i32 0
45   %14 = load <2 x i64>, <2 x i64>* %13, align 16
```

```
  %15 = lshr <2 x i64> %12, <i64 63, i64 63>
  %16 = shl <2 x i64> %12, <i64 1, i64 1>
  %17 = shufflevector <2 x i64> %15, <2 x i64> <i64 0, i64 undef>, <2 x i32> <i32 ↩
      1, i32 2>
  %18 = shufflevector <2 x i64> <i64 undef, i64 0>, <2 x i64> %15, <2 x i32> <i32 ↩
      1, i32 2>
  %19 = or <2 x i64> %16, %14
  %20 = or <2 x i64> %19, %18
  store <2 x i64> %17, <2 x i64>* %13, align 16
  %21 = load <2 x i64>, <2 x i64>* %9, align 16
  %22 = and <2 x i64> %21, %20
  %23 = getelementptr %"editd", %"editd"* %self, i64 0, i32 0, i32 1
  %24 = load <2 x i64>, <2 x i64>* %23, align 16
  %25 = or <2 x i64> %24, <i64 -2, i64 -1>
  store <2 x i64> <i64 1, i64 0>, <2 x i64>* %23, align 16
  %26 = load <2 x i64>, <2 x i64>* %9, align 16
  %27 = and <2 x i64> %26, %25
  %28 = xor <2 x i64> %26, <i64 -1, i64 -1>
  %29 = and <2 x i64> %28, %20
  %30 = getelementptr %"editd", %"editd"* %self, i64 0, i32 0, i32 2
  %31 = load <2 x i64>, <2 x i64>* %30, align 16
  %32 = lshr <2 x i64> %22, <i64 63, i64 63>
  %33 = shl <2 x i64> %22, <i64 1, i64 1>
  %34 = shufflevector <2 x i64> %32, <2 x i64> <i64 0, i64 undef>, <2 x i32> <i32 ↩
      1, i32 2>
  %35 = shufflevector <2 x i64> <i64 undef, i64 0>, <2 x i64> %32, <2 x i32> <i32 ↩
      1, i32 2>
  %36 = or <2 x i64> %35, %33
  %37 = or <2 x i64> %36, %31
  store <2 x i64> %34, <2 x i64>* %30, align 16
  %38 = load <2 x i64>, <2 x i64>* %8, align 16
  %39 = or <2 x i64> %27, %29
  %40 = or <2 x i64> %39, %37
  %41 = or <2 x i64> %40, %38
  %42 = getelementptr %"editd", %"editd"* %self, i64 0, i32 0, i32 3
  %43 = load <2 x i64>, <2 x i64>* %42, align 16
  %44 = lshr <2 x i64> %41, <i64 63, i64 63>
  %45 = shl <2 x i64> %41, <i64 1, i64 1>
  %46 = shufflevector <2 x i64> %44, <2 x i64> <i64 0, i64 undef>, <2 x i32> <i32 ↩
      1, i32 2>
  %47 = shufflevector <2 x i64> <i64 undef, i64 0>, <2 x i64> %44, <2 x i32> <i32 ↩
      1, i32 2>
  %48 = or <2 x i64> %45, %43
  %49 = or <2 x i64> %48, %47
  store <2 x i64> %46, <2 x i64>* %42, align 16
  %50 = load <2 x i64>, <2 x i64>* %11, align 16
  %51 = and <2 x i64> %50, %37
  %52 = and <2 x i64> %49, %50
  %53 = xor <2 x i64> %50, <i64 -1, i64 -1>
  %54 = and <2 x i64> %53, %37
  %55 = getelementptr %"editd", %"editd"* %self, i64 0, i32 0, i32 4
  %56 = load <2 x i64>, <2 x i64>* %55, align 16
  %57 = lshr <2 x i64> %51, <i64 63, i64 63>
  %58 = shl <2 x i64> %51, <i64 1, i64 1>
  %59 = shufflevector <2 x i64> %57, <2 x i64> <i64 0, i64 undef>, <2 x i32> <i32 ↩
      1, i32 2>
  %60 = shufflevector <2 x i64> <i64 undef, i64 0>, <2 x i64> %57, <2 x i32> <i32 ↩
      1, i32 2>
  %61 = or <2 x i64> %58, %56
  %62 = or <2 x i64> %61, %60
  store <2 x i64> %59, <2 x i64>* %55, align 16
  %63 = or <2 x i64> %54, %22
  %64 = or <2 x i64> %63, %52
  %65 = or <2 x i64> %64, %62
  %66 = load <2 x i64>, <2 x i64>* %11, align 16
  %67 = and <2 x i64> %66, %25
  %68 = and <2 x i64> %49, %53
```

```
  %69 = getelementptr %"editd", %"editd"* %self, i64 0, i32 0, i32 5
  %70 = load <2 x i64>, <2 x i64>* %69, align 16
  %71 = lshr <2 x i64> %65, <i64 63, i64 63>
  %72 = shl <2 x i64> %65, <i64 1, i64 1>
  %73 = shufflevector <2 x i64> %71, <2 x i64> <i64 0, i64 undef>, <2 x i32> <i32 ↩
      1, i32 2>
  %74 = shufflevector <2 x i64> <i64 undef, i64 0>, <2 x i64> %71, <2 x i32> <i32 ↩
      1, i32 2>
  %75 = or <2 x i64> %72, %70
  %76 = or <2 x i64> %75, %74
  store <2 x i64> %73, <2 x i64>* %69, align 16
  %77 = or <2 x i64> %68, %41
  %78 = or <2 x i64> %77, %67
  %79 = or <2 x i64> %78, %76
  %80 = load <2 x i64>, <2 x i64>* %10, align 16
  %81 = and <2 x i64> %80, %62
  %82 = and <2 x i64> %76, %80
  %83 = xor <2 x i64> %80, <i64 -1, i64 -1>
  %84 = and <2 x i64> %83, %62
  %85 = getelementptr %"editd", %"editd"* %self, i64 0, i32 0, i32 6
  %86 = load <2 x i64>, <2 x i64>* %85, align 16
  %87 = lshr <2 x i64> %81, <i64 63, i64 63>
  %88 = shl <2 x i64> %81, <i64 1, i64 1>
  %89 = shufflevector <2 x i64> %87, <2 x i64> <i64 0, i64 undef>, <2 x i32> <i32 ↩
      1, i32 2>
  %90 = shufflevector <2 x i64> <i64 undef, i64 0>, <2 x i64> %87, <2 x i32> <i32 ↩
      1, i32 2>
  store <2 x i64> %89, <2 x i64>* %85, align 16
  %91 = or <2 x i64> %84, %51
  %92 = or <2 x i64> %91, %86
  %93 = or <2 x i64> %92, %88
  %94 = or <2 x i64> %93, %82
  %95 = or <2 x i64> %94, %90
  %96 = getelementptr %"editd", %"editd"* %self, i64 0, i32 0, i32 7
  %97 = load <2 x i64>, <2 x i64>* %96, align 16
  %98 = lshr <2 x i64> %79, <i64 63, i64 63>
  %99 = shl <2 x i64> %79, <i64 1, i64 1>
  %100 = shufflevector <2 x i64> %98, <2 x i64> <i64 0, i64 undef>, <2 x i32> <i32 ↩
      1, i32 2>
  %101 = shufflevector <2 x i64> <i64 undef, i64 0>, <2 x i64> %98, <2 x i32> <i32 ↩
      1, i32 2>
  %102 = or <2 x i64> %99, %97
  %103 = or <2 x i64> %102, %101
  store <2 x i64> %100, <2 x i64>* %96, align 16
  %104 = load <2 x i64>, <2 x i64>* %10, align 16
  %105 = and <2 x i64> %103, %104
  %106 = and <2 x i64> %76, %83
  %107 = getelementptr %"editd", %"editd"* %self, i64 0, i32 0, i32 8
  %108 = load <2 x i64>, <2 x i64>* %107, align 16
  %109 = lshr <2 x i64> %95, <i64 63, i64 63>
  %110 = shl <2 x i64> %95, <i64 1, i64 1>
  %111 = shufflevector <2 x i64> %109, <2 x i64> <i64 0, i64 undef>, <2 x i32> <i32↩
      1, i32 2>
  %112 = shufflevector <2 x i64> <i64 undef, i64 0>, <2 x i64> %109, <2 x i32> <i32↩
      1, i32 2>
  store <2 x i64> %111, <2 x i64>* %107, align 16
  %113 = or <2 x i64> %106, %65
  %114 = or <2 x i64> %113, %108
  %115 = or <2 x i64> %114, %110
  %116 = or <2 x i64> %115, %105
  %117 = or <2 x i64> %116, %112
  %118 = xor <2 x i64> %81, <i64 -1, i64 -1>
  %119 = and <2 x i64> %95, %118
  %120 = xor <2 x i64> %95, <i64 -1, i64 -1>
  %121 = and <2 x i64> %117, %120
  %pat_processedItemCount8 = load i64, i64* %1, align 4
  %122 = lshr i64 %pat_processedItemCount8, 7
```

```
  %123 = getelementptr %"editd", %"editd"* %self, i64 0, i32 3
  %E_bufferPtr = load [3 x <2 x i64>]*, [3 x <2 x i64>]** %123, align 8
  %124 = and i64 %122, 1
  %125 = getelementptr [3 x <2 x i64>], [3 x <2 x i64>]* %E_bufferPtr, i64 %124, i64 0
  store <2 x i64> %81, <2 x i64>* %125, align 16
  %pat_processedItemCount9 = load i64, i64* %1, align 4
  %126 = lshr i64 %pat_processedItemCount9, 7
  %E_bufferPtr10 = load [3 x <2 x i64>]*, [3 x <2 x i64>]** %123, align 8
  %127 = and i64 %126, 1
  %128 = getelementptr [3 x <2 x i64>], [3 x <2 x i64>]* %E_bufferPtr10, i64 %127, i64 1
  store <2 x i64> %119, <2 x i64>* %128, align 16
  %pat_processedItemCount11 = load i64, i64* %1, align 4
  %129 = lshr i64 %pat_processedItemCount11, 7
  %E_bufferPtr12 = load [3 x <2 x i64>]*, [3 x <2 x i64>]** %123, align 8
  %130 = and i64 %129, 1
  %131 = getelementptr [3 x <2 x i64>], [3 x <2 x i64>]* %E_bufferPtr12, i64 %130, i64 2
  store <2 x i64> %121, <2 x i64>* %131, align 16
  %pat_processedItemCount13 = load i64, i64* %1, align 4
  %132 = add i64 %pat_processedItemCount13, 128
  store i64 %132, i64* %1, align 4
  %133 = add i64 %stridesRemaining, -1
  br label %"editd_strideLoopCond"

"editd_stridesDone": ; preds = %"editd_strideLoopCond"
  indirectbr i8* %branchTarget, [label %"editd_doFinalBlock", label %"editd_segmentDone", label %resume], !prof !1

"editd_doFinalBlock": ; preds = %"editd_stridesDone"
  %134 = sub i64 %patAvailableItems, %pat_processedItemCount1
  %135 = zext i64 %134 to i128
  %136 = shl i128 1, %135
  %137 = getelementptr %"editd", %"editd"* %self, i64 0, i32 4
  %138 = bitcast <2 x i64>* %137 to i128*
  store i128 %136, i128* %138, align 4
  %139 = shl i128 -1, %135
  %140 = getelementptr %"editd", %"editd"* %self, i64 0, i32 5
  %141 = bitcast <2 x i64>* %140 to i128*
  store i128 %139, i128* %141, align 4
  br label %"editd_strideLoopBody"

resume:                                           ; preds = %"editd_stridesDone"
  store i64 %patAvailableItems, i64* %1, align 4
  %142 = getelementptr %"editd", %"editd"* %self, i64 0, i32 8
  store i1 true, i1* %142, align 1
  br label %"editd_segmentDone"

"editd_segmentDone": ; preds = %resume, %"editd_stridesDone"
  ret void
}
```

A.3 LLVM IR of Main Program

This is the generated LLVM IR code of the main program that executes 5 kernels in sequential. One of the kernel, Editd, is shown in Appendix A.3, others are not provided in the Appendix.

```
define void @Main(i32 %fileDescriptor) {
entry:
  %0 = alloca [0 x { [1 x [8 x <2 x i64>]]*, i64, i64 }], align 64
  %.sub = getelementptr inbounds [0 x { [1 x [8 x <2 x i64>]]*, i64, i64 }], [0 x {
       [1 x [8 x <2 x i64>]]*, i64, i64 }]* %0, i64 0, i64 0
  %.sub.repack = getelementptr inbounds [0 x { [1 x [8 x <2 x i64>]]*, i64, i64 }],
       [0 x { [1 x [8 x <2 x i64>]]*, i64, i64 }]* %0, i64 0, i64 0, i32 0
  store [1 x [8 x <2 x i64>]]* null, [1 x [8 x <2 x i64>]]** %.sub.repack, align 64
  %.sub.repack11 = getelementptr inbounds [0 x { [1 x [8 x <2 x i64>]]*, i64, i64
      }], [0 x { [1 x [8 x <2 x i64>]]*, i64, i64 }]* %0, i64 0, i64 0, i32 1
  store i64 0, i64* %.sub.repack11, align 8
  %.sub.repack12 = getelementptr inbounds [0 x { [1 x [8 x <2 x i64>]]*, i64, i64
      }], [0 x { [1 x [8 x <2 x i64>]]*, i64, i64 }]* %0, i64 0, i64 0, i32 2
  store i64 0, i64* %.sub.repack12, align 16
  %1 = call void* @aligned_alloc(i64 64, i64 320)
  %2 = bitcast void* %1 to [8 x <2 x i64>]*
  store [8 x <2 x i64>] zeroinitializer, [8 x <2 x i64>]* %2, align 64
  %3 = call void* @aligned_alloc(i64 64, i64 192)
  %4 = bitcast void* %3 to [4 x <2 x i64>]*
  store [4 x <2 x i64>] zeroinitializer, [4 x <2 x i64>]* %4, align 64
  %5 = call void* @aligned_alloc(i64 64, i64 128)
  %6 = bitcast void* %5 to [3 x <2 x i64>]*
  store [3 x <2 x i64>] zeroinitializer, [3 x <2 x i64>]* %6, align 64
  %7 = alloca %"mmap_source1@8_01", align 64
  %8 = alloca %s2p_01, align 64
  %9 = alloca %ccc_01, align 64
  %10 = alloca %"editd", align 64
  %11 = alloca %scanMatch_01, align 64
  %12 = alloca { i64, i64** }, align 8
  %13 = alloca [1 x i64*], align 8
  %14 = getelementptr %s2p_01, %s2p_01* %8, i64 0, i32 4
  %15 = getelementptr [1 x i64*], [1 x i64*]* %13, i64 0, i64 0
  store i64* %14, i64** %15, align 8
  %16 = getelementptr { i64, i64** }, { i64, i64** }* %12, i64 0, i32 0
  store i64 1, i64* %16, align 8
  %17 = getelementptr { i64, i64** }, { i64, i64** }* %12, i64 0, i32 1
  store i64** %15, i64*** %17, align 8
  call void @"mmap_source1@8_01_Init"(%"mmap_source1@8_01"* nonnull %7, i32
      %fileDescriptor, { [1 x [8 x <2 x i64>]]*, i64, i64 }* %.sub, { i64, i64** }*
       nonnull %12)
  %18 = alloca { i64, i64** }, align 8
  %19 = alloca [1 x i64*], align 8
  %20 = getelementptr %ccc_01, %ccc_01* %9, i64 0, i32 7
  %21 = getelementptr [1 x i64*], [1 x i64*]* %19, i64 0, i64 0
  store i64* %20, i64** %21, align 8
  %22 = getelementptr { i64, i64** }, { i64, i64** }* %18, i64 0, i32 0
  store i64 1, i64* %22, align 8
  %23 = getelementptr { i64, i64** }, { i64, i64** }* %18, i64 0, i32 1
  store i64** %21, i64*** %23, align 8
  call void @s2p_01_Init(%s2p_01* nonnull %8, { [1 x [8 x <2 x i64>]]*, i64, i64 }*
       %.sub, [8 x <2 x i64>]* %2, { i64, i64** }* nonnull %18)
  %24 = alloca { i64, i64** }, align 8
  %25 = alloca [1 x i64*], align 8
  %26 = getelementptr %"editd", %"editd"* %10, i64 0, i32 7
  %27 = getelementptr [1 x i64*], [1 x i64*]* %25, i64 0, i64 0
  store i64* %26, i64** %27, align 8
  %28 = getelementptr { i64, i64** }, { i64, i64** }* %24, i64 0, i32 0
  store i64 1, i64* %28, align 8
  %29 = getelementptr { i64, i64** }, { i64, i64** }* %24, i64 0, i32 1
```

```
  store i64** %27, i64*** %29, align 8
  call void @ccc_01_Init(%ccc_01* nonnull %9, [8 x <2 x i64>]* %2, [4 x <2 x i64>]*↩
      %4, { i64, i64** }* nonnull %24)
  %30 = alloca { i64, i64** }, align 8
  %31 = alloca [1 x i64*], align 8
  %32 = getelementptr %scanMatch_01, %scanMatch_01* %11, i64 0, i32 3
  %33 = getelementptr [1 x i64*], [1 x i64*]* %31, i64 0, i64 0
  store i64* %32, i64** %33, align 8
  %34 = getelementptr { i64, i64** }, { i64, i64** }* %30, i64 0, i32 0
  store i64 1, i64* %34, align 8
  %35 = getelementptr { i64, i64** }, { i64, i64** }* %30, i64 0, i32 1
  store i64** %33, i64*** %35, align 8
  call void @"editd_Init"(%"editd"* nonnull %10, [4 x <2 x i64>]* %4, [3 x <2 x i64↩
      >]* %6, { i64, i64** }* nonnull %30)
  call void @scanMatch_01_Init(%scanMatch_01* nonnull %11, [3 x <2 x i64>]* %6)
  br label %pipelineLoop

pipelineLoop:                                     ; preds = %pipelineLoop, %entry
  call void @"mmap_source1@8_01_DoSegment"(%"mmap_source1@8_01"* nonnull %7, i1 ↩
      false)
  %36 = getelementptr %"mmap_source1@8_01", %"mmap_source1@8_01"* %7, i64 0, i32 7
  %terminationSignal = load i1, i1* %36, align 8
  %37 = getelementptr %"mmap_source1@8_01", %"mmap_source1@8_01"* %7, i64 0, i32 0
  %sourceBuffer_producedItemCount = load i64, i64* %37, align 64
  %38 = getelementptr %"mmap_source1@8_01", %"mmap_source1@8_01"* %7, i64 0, i32 6
  %39 = load atomic i64, i64* %38 acquire, align 16
  %40 = add i64 %39, 1
  store atomic i64 %40, i64* %38 release, align 16
  %41 = getelementptr %s2p_01, %s2p_01* %8, i64 0, i32 0
  call void @s2p_01_DoSegment(%s2p_01* nonnull %8, i1 %terminationSignal, i64 ↩
      %sourceBuffer_producedItemCount)
  %byteStream_processedItemCount1 = load i64, i64* %41, align 64
  %42 = load atomic i64, i64* %14 acquire, align 32
  %43 = add i64 %42, 1
  store atomic i64 %43, i64* %14 release, align 32
  %44 = getelementptr %ccc_01, %ccc_01* %9, i64 0, i32 1
  call void @ccc_01_DoSegment(%ccc_01* nonnull %9, i1 %terminationSignal, i64 ↩
      %byteStream_processedItemCount1)
  %45 = getelementptr %ccc_01, %ccc_01* %9, i64 0, i32 8
  %terminationSignal3 = load i1, i1* %45, align 16
  %46 = or i1 %terminationSignal, %terminationSignal3
  %basis_processedItemCount4 = load i64, i64* %44, align 64
  %47 = load atomic i64, i64* %20 acquire, align 8
  %48 = add i64 %47, 1
  store atomic i64 %48, i64* %20 release, align 8
  %49 = getelementptr %"editd", %"editd"* %10, i64 0, i32 1
  call void @"editd_DoSegment"(%"editd"* nonnull %10, i1 %46, i64 ↩
      %basis_processedItemCount4)
  %50 = getelementptr %"editd", %"editd"* %10, i64 0, i32 8
  %terminationSignal6 = load i1, i1* %50, align 32
  %51 = or i1 %46, %terminationSignal6
  %pat_processedItemCount7 = load i64, i64* %49, align 16
  %52 = load atomic i64, i64* %26 acquire, align 8
  %53 = add i64 %52, 1
  store atomic i64 %53, i64* %26 release, align 8
  %54 = getelementptr %scanMatch_01, %scanMatch_01* %11, i64 0, i32 0
  call void @scanMatch_01_DoSegment(%scanMatch_01* nonnull %11, i1 %51, i64 ↩
      %pat_processedItemCount7)
  %55 = getelementptr %scanMatch_01, %scanMatch_01* %11, i64 0, i32 4
  %terminationSignal9 = load i1, i1* %55, align 32
  %56 = or i1 %51, %terminationSignal9
  %matchResults_processedItemCount10 = load i64, i64* %54, align 64
  %57 = load atomic i64, i64* %32 acquire, align 8
  %58 = add i64 %57, 1
  store atomic i64 %58, i64* %32 release, align 8
  %59 = getelementptr %s2p_01, %s2p_01* %8, i64 0, i32 6
  store i64 %basis_processedItemCount4, i64* %59, align 4
```

```
  %60 = getelementptr %"mmap_source1@8_01", %"mmap_source1@8_01"* %7, i64 0, i32 8
  store i64 %byteStream_processedItemCount1, i64* %60, align 4
  %61 = getelementptr %ccc_01, %ccc_01* %9, i64 0, i32 9
  store i64 %pat_processedItemCount7, i64* %61, align 4
  %62 = getelementptr %"editd", %"editd"* %10, i64 0, i32 9
  store i64 %matchResults_processedItemCount10, i64* %62, align 4
  br i1 %56, label %pipelineExit, label %pipelineLoop

pipelineExit:                                     ; preds = %pipelineLoop
  %63 = call i64 @"mmap_source1@8_01_Terminate"(%"mmap_source1@8_01"* nonnull %7)
  call void @s2p_01_Terminate(%s2p_01* nonnull %8)
  call void @ccc_01_Terminate(%ccc_01* nonnull %9)
  call void @"editd_Terminate"(%"editd"* nonnull %10)
  call void @scanMatch_01_Terminate(%scanMatch_01* nonnull %11)
  call void @free(void* %1)
  call void @free(void* %3)
  call void @free(void* %5)
  ret void
}
```

A.4 LLVM IR of Pipeline Thread Function for Editd Kernel

This is the generated LLVM IR code of one of the thread functions that processes Editd kernel under the conventional pipeline parallelization mode. Five thread functions are generated for this application. Programmers can execute their programs under this parallelization mode using the command line option `-enable-pipeline-parallel`.

```
define internal void @"ppt:editd"(void* %input) {
entry:
  %0 = bitcast void* %input to { %"mmap_source1@8_01"*, %s2p_01*, %ccc_01*, %"editd"*, %scanMatch_01* }*
  %1 = getelementptr { %"mmap_source1@8_01"*, %s2p_01*, %ccc_01*, %"editd"*, %scanMatch_01* }, { %"mmap_source1@8_01"*, %s2p_01*, %ccc_01*, %"editd"*, %scanMatch_01* }* %0, i64 0, i32 2
  %2 = load %ccc_01*, %ccc_01** %1, align 8
  %3 = getelementptr { %"mmap_source1@8_01"*, %s2p_01*, %ccc_01*, %"editd"*, %scanMatch_01* }, { %"mmap_source1@8_01"*, %s2p_01*, %ccc_01*, %"editd"*, %scanMatch_01* }* %0, i64 0, i32 3
  %4 = load %"editd"*, %"editd"** %3, align 8
  %5 = getelementptr { %"mmap_source1@8_01"*, %s2p_01*, %ccc_01*, %"editd"*, %scanMatch_01* }, { %"mmap_source1@8_01"*, %s2p_01*, %ccc_01*, %"editd"*, %scanMatch_01* }* %0, i64 0, i32 4
  %6 = load %scanMatch_01*, %scanMatch_01** %5, align 8
  br label %outputCheck

outputCheck:                                      ; preds = %outputCheck, %doSegment, %entry
  %segNo = phi i64 [ 0, %entry ], [ %segNo, %outputCheck ], [ %13, %doSegment ]
  %7 = getelementptr %scanMatch_01, %scanMatch_01* %6, i64 0, i32 3
  %8 = load atomic i64, i64* %7 acquire, align 8
  %9 = icmp ugt i64 %segNo, %8
  br i1 %9, label %outputCheck, label %inputCheck

inputCheck:                                       ; preds = %outputCheck, %inputCheck
  %10 = getelementptr %ccc_01, %ccc_01* %2, i64 0, i32 7
  %11 = load atomic i64, i64* %10 acquire, align 8
  %12 = icmp ult i64 %segNo, %11
  br i1 %12, label %doSegment, label %inputCheck

doSegment:                                        ; preds = %inputCheck
  %13 = add i64 %segNo, 1
  %14 = getelementptr %ccc_01, %ccc_01* %2, i64 0, i32 8
  %terminationSignal = load i1, i1* %14, align 1
  %15 = icmp eq i64 %13, %11
  %16 = and i1 %15, %terminationSignal
  %17 = shl i64 %segNo, 7
  call void @"editd_DoSegment"(%"editd"* %4, i1 %16, i64 %17)
  %18 = getelementptr %"editd", %"editd"* %4, i64 0, i32 7
  store atomic i64 %13, i64* %18 release, align 8
  br i1 %16, label %exitThread, label %outputCheck

exitThread:                                       ; preds = %doSegment
  call void @pthread_exit(void* null) #1
  unreachable
}
```

A.5 LLVM IR of Data-pipeline Thread Function

This is the generated LLVM IR code of the thread function for the data-pipeline parallelization mode. Programmers can execute their programs under this parallelization mode using the command line option `-enable-data-pipeline-parallel`.

```
define internal void @segment(void* %input) {
entry:
  %0 = bitcast void* %input to { { %"mmap_source1@8_01"*, %s2p_01*, %ccc_01*, %"editd"*, %scanMatch_01* }*, i64 }*
  %1 = bitcast void* %input to { %"mmap_source1@8_01"*, %s2p_01*, %ccc_01*, %"editd"*, %scanMatch_01* }**
  %2 = load { %"mmap_source1@8_01"*, %s2p_01*, %ccc_01*, %"editd"*, %scanMatch_01* }*, { %"mmap_source1@8_01"*, %s2p_01*, %ccc_01*, %"editd"*, %scanMatch_01* }** %1, align 8
  %3 = getelementptr { %"mmap_source1@8_01"*, %s2p_01*, %ccc_01*, %"editd"*, %scanMatch_01* }, { %"mmap_source1@8_01"*, %s2p_01*, %ccc_01*, %"editd"*, %scanMatch_01* }* %2, i64 0, i32 0
  %4 = load %"mmap_source1@8_01"*, %"mmap_source1@8_01"** %3, align 8
  %5 = getelementptr { %"mmap_source1@8_01"*, %s2p_01*, %ccc_01*, %"editd"*, %scanMatch_01* }, { %"mmap_source1@8_01"*, %s2p_01*, %ccc_01*, %"editd"*, %scanMatch_01* }* %2, i64 0, i32 1
  %6 = load %s2p_01*, %s2p_01** %5, align 8
  %7 = getelementptr { %"mmap_source1@8_01"*, %s2p_01*, %ccc_01*, %"editd"*, %scanMatch_01* }, { %"mmap_source1@8_01"*, %s2p_01*, %ccc_01*, %"editd"*, %scanMatch_01* }* %2, i64 0, i32 2
  %8 = load %ccc_01*, %ccc_01** %7, align 8
  %9 = getelementptr { %"mmap_source1@8_01"*, %s2p_01*, %ccc_01*, %"editd"*, %scanMatch_01* }, { %"mmap_source1@8_01"*, %s2p_01*, %ccc_01*, %"editd"*, %scanMatch_01* }* %2, i64 0, i32 3
  %10 = load %"editd"*, %"editd"** %9, align 8
  %11 = getelementptr { %"mmap_source1@8_01"*, %s2p_01*, %ccc_01*, %"editd"*, %scanMatch_01* }, { %"mmap_source1@8_01"*, %s2p_01*, %ccc_01*, %"editd"*, %scanMatch_01* }* %2, i64 0, i32 4
  %12 = load %scanMatch_01*, %scanMatch_01** %11, align 8
  %13 = getelementptr { { %"mmap_source1@8_01"*, %s2p_01*, %ccc_01*, %"editd"*, %scanMatch_01* }*, i64 }, { { %"mmap_source1@8_01"*, %s2p_01*, %ccc_01*, %"editd"*, %scanMatch_01* }*, i64 }* %0, i64 0, i32 1
  %14 = load i64, i64* %13, align 4
  br label %segmentLoop

segmentLoop:                                      ; preds = %scanMatch_01Do, %entry
  %segNo = phi i64 [ %14, %entry ], [ %46, %scanMatch_01Do ]
  %15 = add i64 %segNo, 1
  br label %"mmap_source1@8_01Wait"

"mmap_source1@8_01Wait":                          ; preds = %"mmap_source1@8_01Wait", %segmentLoop
  %16 = getelementptr %"mmap_source1@8_01", %"mmap_source1@8_01"* %4, i64 0, i32 6
  %17 = load atomic i64, i64* %16 acquire, align 8
  %18 = icmp eq i64 %segNo, %17
  br i1 %18, label %"mmap_source1@8_01Completed", label %"mmap_source1@8_01Wait"

"mmap_source1@8_01Do":                            ; preds = %"mmap_source1@8_01Completed"
  call void @"mmap_source1@8_01_DoSegment"(%"mmap_source1@8_01"* %4, i1 false)
  %terminationSignal1 = load i1, i1* %20, align 1
  %19 = getelementptr %"mmap_source1@8_01", %"mmap_source1@8_01"* %4, i64 0, i32 0
  %sourceBuffer_producedItemCount = load i64, i64* %19, align 4
  store atomic i64 %15, i64* %16 release, align 8
  br label %s2p_01Wait

"mmap_source1@8_01Completed":                     ; preds = %"mmap_source1@8_01Wait"
  %20 = getelementptr %"mmap_source1@8_01", %"mmap_source1@8_01"* %4, i64 0, i32 7
```

```
41    %terminationSignal = load i1, i1* %20, align 1
42    br i1 %terminationSignal, label %"mmap_source1@8_01Exit", label %"↩
          mmap_source1@8_01Do"

44  "mmap_source1@8_01Exit":                          ; preds = %"↩
        mmap_source1@8_01Completed"
45    store atomic i64 %15, i64* %16 release, align 8
46    br label %exitThread

48  s2p_01Wait:                                       ; preds = %s2p_01Wait, %"↩
        mmap_source1@8_01Do"
49    %21 = getelementptr %s2p_01, %s2p_01* %6, i64 0, i32 4
50    %22 = load atomic i64, i64* %21 acquire, align 8
51    %23 = icmp eq i64 %segNo, %22
52    br i1 %23, label %s2p_01Do, label %s2p_01Wait

54  s2p_01Do:                                         ; preds = %s2p_01Wait
55    call void @s2p_01_DoSegment(%s2p_01* %6, i1 %terminationSignal1, i64 ↩
          %sourceBuffer_producedItemCount)
56    %24 = getelementptr %s2p_01, %s2p_01* %6, i64 0, i32 0
57    %byteStream_processedItemCount = load i64, i64* %24, align 4
58    store atomic i64 %15, i64* %21 release, align 8
59    br label %ccc_01Wait

61  ccc_01Wait:                                       ; preds = %ccc_01Wait, %s2p_01Do
62    %25 = getelementptr %ccc_01, %ccc_01* %8, i64 0, i32 7
63    %26 = load atomic i64, i64* %25 acquire, align 8
64    %27 = icmp eq i64 %segNo, %26
65    br i1 %27, label %ccc_01Completed, label %ccc_01Wait

67  ccc_01Do:                                         ; preds = %ccc_01Completed
68    call void @ccc_01_DoSegment(%ccc_01* %8, i1 %terminationSignal1, i64 ↩
          %byteStream_processedItemCount)
69    %terminationSignal4 = load i1, i1* %30, align 1
70    %28 = or i1 %terminationSignal1, %terminationSignal4
71    %29 = getelementptr %ccc_01, %ccc_01* %8, i64 0, i32 1
72    %basis_processedItemCount = load i64, i64* %29, align 4
73    store atomic i64 %15, i64* %25 release, align 8
74    br label %"editdWait"

76  ccc_01Completed:                                  ; preds = %ccc_01Wait
77    %30 = getelementptr %ccc_01, %ccc_01* %8, i64 0, i32 8
78    %terminationSignal3 = load i1, i1* %30, align 1
79    br i1 %terminationSignal3, label %ccc_01Exit, label %ccc_01Do

81  ccc_01Exit:                                       ; preds = %ccc_01Completed
82    store atomic i64 %15, i64* %25 release, align 8
83    br label %exitThread

85  "editdWait": ; preds = %"editdWait", %ccc_01Do
86    %31 = getelementptr %"editd", %"editd"* %10, i64 0, i32 7
87    %32 = load atomic i64, i64* %31 acquire, align 8
88    %33 = icmp eq i64 %segNo, %32
89    br i1 %33, label %"editdCompleted", label %"editdWait"

91  "editdDo":  ; preds = %"editdCompleted"
92    call void @"editd_DoSegment"(%"editd"* %10, i1 %28, i64 %basis_processedItemCount↩
          )
93    %terminationSignal7 = load i1, i1* %36, align 1
94    %34 = or i1 %28, %terminationSignal7
95    %35 = getelementptr %"editd", %"editd"* %10, i64 0, i32 1
96    %pat_processedItemCount = load i64, i64* %35, align 4
97    store atomic i64 %15, i64* %31 release, align 8
98    br label %scanMatch_01Wait

100 "editdCompleted": ; preds = %"editdWait"
101   %36 = getelementptr %"editd", %"editd"* %10, i64 0, i32 8
```

```
  %terminationSignal6 = load i1, i1* %36, align 1
  br i1 %terminationSignal6, label %"editdExit", label %"editdDo"

"editdExit": ; preds = %"editdCompleted"
  store atomic i64 %15, i64* %31 release, align 8
  br label %exitThread

scanMatch_01Wait:                                 ; preds = %scanMatch_01Wait, %"editdDo"
  %37 = getelementptr %scanMatch_01, %scanMatch_01* %12, i64 0, i32 3
  %38 = load atomic i64, i64* %37 acquire, align 8
  %39 = icmp eq i64 %segNo, %38
  br i1 %39, label %scanMatch_01Completed, label %scanMatch_01Wait

scanMatch_01Do:                                   ; preds = %scanMatch_01Completed
  call void @scanMatch_01_DoSegment(%scanMatch_01* %12, i1 %34, i64 %pat_processedItemCount)
  %terminationSignal10 = load i1, i1* %48, align 1
  %40 = or i1 %34, %terminationSignal10
  %41 = getelementptr %scanMatch_01, %scanMatch_01* %12, i64 0, i32 0
  %matchResults_processedItemCount = load i64, i64* %41, align 4
  store atomic i64 %15, i64* %37 release, align 8
  %42 = getelementptr %s2p_01, %s2p_01* %6, i64 0, i32 6
  store i64 %basis_processedItemCount, i64* %42, align 4
  %43 = getelementptr %"mmap_source1@8_01", %"mmap_source1@8_01"* %4, i64 0, i32 8
  store i64 %byteStream_processedItemCount, i64* %43, align 4
  %44 = getelementptr %ccc_01, %ccc_01* %8, i64 0, i32 9
  store i64 %pat_processedItemCount, i64* %44, align 4
  %45 = getelementptr %"editd", %"editd"* %10, i64 0, i32 9
  store i64 %matchResults_processedItemCount, i64* %45, align 4
  %46 = add i64 %segNo, 4
  br i1 %40, label %exitThread, label %segmentLoop

exitThread:                                       ; preds = %scanMatch_01Do, %scanMatch_01Exit, %"editdExit", %ccc_01Exit, %"mmap_source1@8_01Exit"
  %47 = icmp eq i64 %14, 0
  br i1 %47, label %ExitProcessFunction, label %ExitThread

scanMatch_01Completed:                            ; preds = %scanMatch_01Wait
  %48 = getelementptr %scanMatch_01, %scanMatch_01* %12, i64 0, i32 4
  %terminationSignal9 = load i1, i1* %48, align 1
  br i1 %terminationSignal9, label %scanMatch_01Exit, label %scanMatch_01Do

scanMatch_01Exit:                                 ; preds = %scanMatch_01Completed
  store atomic i64 %15, i64* %37 release, align 8
  br label %exitThread

ExitThread:                                       ; preds = %exitThread
  call void @pthread_exit(void* null) #1
  unreachable

ExitProcessFunction:                              ; preds = %exitThread
  ret void
}
```

A.6 Build an Application with Parabix Framework

This is the main program the implement a Parabix application with Parabix driver.

```
int main(int argc, char *argv[]) {
  // Create a Parabix Driver.
  ParabixDriver pxDriver("test");

  // Get the IDISA Builder.
  auto & iBuilder = pxDriver.getBuilder();

  // Prepare the input buffer for S2P kernel.
  // This is a byte-oriented stream defined by Stream Set 1 X 8
  StreamSetBuffer * const ByteStream = pxDriver.addBuffer(
    make_unique<SourceBuffer>(iBuilder, iBuilder->getStreamSetTy(1, 8)));
  // Prepare the output buffer for S2P kernel.
  // This is the parallel bit stream defined by Stream Set 8 X 1
  StreamSetBuffer * const BasisBits = pxDriver.addBuffer(
    make_unique<CircularBuffer>(iBuilder, iBuilder->getStreamSetTy(8, 1)));

  // Create S2P kernel instance.
  Kernel * s2pk = pxDriver.addKernelInstance(make_unique<S2PKernel>(iBuilder));
  // Add S2P kernel to the module.
  pxDriver.makeKernelCall(s2pk, {ByteStream}, {BasisBits});

  // You can define more buffers and kernels here.
  // e.g. TestPabloKernel, TestGenericKernel
  // ...

  // Call pipeline buider to generate the pipeline of kernels.
  pxDriver.generatePipelineIR();

  pxDriver.finalizeObject();
}
```

This is the header file that defines a kernel involving Pablo operations.

```
class TestPabloKernel final: public pablo::PabloKernel {
public:
  TestPabloKernel(const std::unique_ptr<kernel::KernelBuilder> & b);
protected:
  // Implement the kernel logic in this method.
  void generatePabloMethod() override;
};
```

This is the header file that defines a generic kernel.

```
class TestGenericKernel final: public BlockOrientedKernel {
public:
  TestGenericKernel(const std::unique_ptr<kernel::KernelBuilder> & b);
protected:
  // Implement the kernel logic in this method.
  void generatePabloMethod() override;
};
```

All the source code can be check out from `http://parabix.costar.sfu.ca/svn/`