Transforming and Optimizing Irregular Applications for Parallel Architectures

Jing Zhang

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science and Application

September 28, 2017

Blacksburg, Virginia

Keywords: Irregular Applications, Parallel Architectures, Multi-core, Many-core, Multi-node, Bioinformatics

Transforming and Optimizing Irregular Applications for Parallel Architectures

Jing Zhang

ABSTRACT

Parallel architectures, including multi-core processors, many-core processors, and multi-node systems, have become commonplace, as it is no longer feasible to improve single-core performance through increasing its operating clock frequency. Furthermore, to keep up with the exponentially growing desire for more and more computational power, the number of cores/nodes in parallel architectures has continued to dramatically increase. On the other hand, many applications in well-established and emerging fields, such as bioinformatics, social network analysis, and graph processing, exhibit increasing irregularities in memory access, control flow, and communication patterns. While multiple techniques have been introduced into modern parallel architectures to tolerate these irregularities, many irregular applications still execute poorly on current parallel architectures, as their irregularities exceed the capabilities of these techniques. Therefore, it is critical to resolve irregularities in applications for parallel architectures. However, this is a very challenging task, as the irregularities are dynamic, and hence, unknown until runtime.

To optimize irregular applications, many approaches have been proposed to improve data locality and reduce irregularities through computational and data transformations. However, there are two major drawbacks in these existing approaches that prevent them from achieving optimal performance. First, these approaches use *local optimizations* that exploit data locality and regularity locally within a loop or kernel. However, in many applications, there is hidden locality across loops or kernels. Second, these approaches use *"one-size-fits-all" methods* that treat all irregular patterns equally and resolve them with a single method. However, many irregular applications have complex irregularities, which are mixtures of different types of irregularities and need differentiated optimizations. To overcome these two drawbacks, we propose a general methodology that includes a taxonomy of irregularities to help us analyze the irregular patterns in an application, and a set of adaptive transformations to reorder data and computation based on the characteristics of the application and architecture.

By extending our adaptive data-reordering transformation on a single node, we propose a data-partitioning framework to resolve the load imbalance problem of irregular applications on multi-node systems. Unlike existing frameworks, which use *"one-size-fits-all"* methods to partition the input data by a single property, our framework provides a set of operations to transform the input data by multiple properties and generates the desired data-partitioning codes by composing these operations into a workflow.

Transforming and Optimizing Irregular Applications for Parallel Architectures

Jing Zhang

GENERAL AUDIENCE ABSTRACT

Irregular applications, which present unpredictable and irregular patterns of data accesses and computation, are increasingly important in well-established and emerging fields, such as biological data analysis, social network analysis, and machine learning, to deal with large datasets. On the other hand, current parallel processors, such as multi-core CPUs (central processing units), GPUs (graphics processing units), and computer clusters (i.e., groups of connected computers), are designed for regular applications and execute irregular applications poorly. Therefore, it is critical to optimize irregular applications for parallel processors. However, it is a very challenging task, as the irregular patterns are dynamic, and hence, unknown until application execution. To overcome this challenge, we propose a general methodology that includes a taxonomy of irregularities to help us analyze the irregular patterns in an application, and a set of adaptive transformations to reorder data and computation for exploring hidden regularities based on the characteristics of the application and processor. We apply our methodology on couples of important and complex irregular applications as case studies to demonstrate that it is effective and efficient.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my Ph.D. advisor Prof. Wu-chun Feng for his continuous support of my Ph.D. study and research. I appreciate his invaluable contributions of time, insights, ideas, and funding to make my Ph.D. experience productive and stimulating.

Besides my advisor, I would also like to thank my dissertation committee members: Dr. Hao Wang, Prof. Ali R. Butt, Prof. Liqing Zhang, Dr. Heshan Lin for their time, valuable comments, suggestions, and feedback.

I would also like to all current and former members of the SyNeRGy lab for their valuable feedback and stimulating discussions.

I would also like to thank my parents and my wife for their unconditional love, care, and encouragement all my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Currently, processor manufacturers have turned to increase the number of cores to improve the overall performance of a processor, as it is no longer feasible to improve single-core performance through increasing its operating clock frequency due to physical limits [2]. For example, the recent Intel Xeon processors feature up to 28 cores per socket, while AMD processors feature 32 cores per socket in its latest generation architecture. Along with this trend, many-core architectures, which can feature an extremely high number of cores (thousands of cores), are becoming popular in high-performance computing (HPC) due to their high throughput and power efficiency. For example, in the recent Top500 list [3], seven (7) of the top 10 supercomputers use many-core processor architectures as accelerators. Furthermore, to keep up with the exponentially growing need for more and more computational power, the number of compute nodes in HPC systems have continued to dramatically increase. For example, the recent No. 1 system from the November 2017 Top500 List (i.e., Sunway TaihuLight) has 40,960 nodes, which is more than double the number of nodes of the previous top system with 16,000 nodes (i.e., Tianhe-2).

On the other hand, many applications in well-established and emerging fields, such as bioinformatics, social network analysis, and graph processing, exhibit increasing irregularities in memory access, control flow, and communication patterns. Though modern architectures have multiple techniques, such as smart hardware prefetching, large caches, and branch prediction, to mitigate the impact of these irregularities, many irregular applications still perform poorly on current parallel architectures, as their irregularities exceed the capabilities of these techniques. For example, the smart hardware prefetcher [4] in Intel CPUs (specifically, Intel Sandy Bridge CPUs or later) can pre-load data into the cache when successive cache misses occur in the last-level cache (LLC), but it requires that the memory accesses are in the same memory page. However, irregular applications could have random memory accesses with large strides across pages.

Therefore, it is critical to eliminate irregularities in applications. However, it is a very challenging task, as the irregularities are dynamic and unpredictable, and hence, unknown until runtime and can even change during the computation. To optimize irregular applications, previous studies [5, 6, 7, 8, 9] propose multiple solutions to map irregular applications onto parallel architectures through reordering data and computation. However, these approaches have two major limitations. First, while past work makes local optimizations to exploit the local regularity and locality within a loop or a kernel. However, many irregular applications have locality that is hidden across loops or kernels. Second, the *"one-size-fits-all" methods* that treat irregularities in an application equally and resolve them with a single method. However, many applications have complex irregularities, which are mixtures of different types of irregularities and need differentiated optimizations.

To overcome these two drawbacks, we present a general methodology to better analyze and optimize irregular applications. First, we propose a taxonomy of irregularities that classifies irregularities into four classes based on the relationship of computation and data structures to help us identify their causes and complexities. From simple to complex, the four classes are as follows: 1) *Single-Data-Single-Compute (SDSC)*, where a single function operates on a single data structure, 2) *Multiple-Data-Single-Compute (MDSC)*, where a single function operates on multiple data structures, 3) *Single-Data-Multiple-Compute (SDMC)*, where a single data structure is operated by multiple functions, and 4) *Multiple-Data-Multiple-Compute (MDMC)*, where multiple functions operate on multiple data structures. Second, we propose three general transformations: (1) *interchanging*, which changes the execution order in an application to exploit global locality and regularity across loops or kernels, (2) *decoupling*, which divides a complex irregular kernel into small kernels with simple irregularity, and resolves them with differentiated optimizations, and (3) *reordering*, which bridges separate kernels of different patterns with data reordering. Third, we propose an adaptive data reordering transformation that provides the optimal data reordering pipeline based on the characteristics of the application and architecture.

For irregular applications on multi-node systems, we propose a balanced data-partitioning framework by extending our adaptive reordering transformation. Unlike with the previous methods, which use the "one-size-fits-all" approach to partition data by a single property (i.e., data size), we determine a set of common input-data operations, including *sort*, *group*, *split* and *distribute*, to reorder and re-distribute the input data by multiple properties and propose a framework that can automatically generate desired data-partitioning codes to improve the load balance of irregular applications by composing these operations.

To demonstrate our methodology, we analyze and optimize couples of important and complex irregular applications on different parallel architectures as case studies.

1.1 Optimizing Irregular Applications for Multi-core Architectures

Current multi-core architectures (i.e., multi-core CPUs) have the sophisticated memory hierarchy and control mechanism to tolerate irregular memory accesses and control flows. However, the irregular memory accesses in irregular applications usually have large strides and unpredictable patterns, which exceed the capabilities of current hardware. It can result in substantial cache and TLB (translation lookaside buffer) misses, which can adversely impact performance. Therefore, many studies [10, 11, 12] propose advanced cache mechanisms to alleviate the effects of irregular memory accesses by detecting and exploring the spatial and temporal locality in the running program. However, these techniques need hardware modifications.

Meanwhile, multiple software approaches have been proposed to improve data locality for multi-core architectures through data and computation reordering. However, these approaches have limited effectiveness and efficiency due to several drawbacks. First, many approaches [13, 14, 15, 16] use static methods, which can hardly deal with dynamic irregular patterns. Second, many dynamic approaches [5, 6, 17, 18] only use *local optimizations* to exploit data locality within a loop or kernel. Third, these dynamic approaches use the *"one-size-fits-all" method* that uses a single method to resolve different types of irregularities. In this dissertation, we analyze and optimize two complex irregular applications on multi-core architectures as case studies to show how to resolve these drawbacks by our methodology.

1.1.1 Case Study - Optimizing Short Read Alignment on Multi-core CPUs

The first application is *short read alignment*, which is responsible for mapping short DNA sequences (i.e., reads) to a long reference genome. It is a fundamental step in next-generation sequencing (NGS) analysis. In this dissertation, we focus on short read alignment tools that use the Burrows-Wheeler Transform (BWT), which are increasingly popular due to their small memory footprint and flexibility. Despite extensive optimization efforts, the performance of these tools still cannot keep up with the explosive growth of sequencing data. Through an in-depth performance characterization of Burrows-Wheeler Aligner (BWA) [19], a popular BWT-based aligner on multi-core architectures, we demonstrate that BWA is limited by memory bandwidth due to the irregular memory access patterns on the reference genome index. Specifically, the search kernel of BWA, which belongs to the *Multiple-Data-Single-Compute* class, reveals extremely poor locality due to its irregular memory access pattern and thus suffers from heavy cache and TLB misses, which can result in up to 85% of stalled cycles. We then propose a locality-aware implementation of BWA, called LA-BWA [20], to explore hidden data locality via *interchanging* the execution order of the kernel and *reordering* memory accesses with the binning method. However, the preliminary optimization method has a couple of drawbacks, limiting the performance gain: 1) the interchanging and binning transformations break the data locality in the original algorithm, which can result in high TLB misses; 2) the preliminary bin structure has high memory pressure, which could limit the scaling of the algorithm. Thus, we propose an optimized binning method with a compressed and cache-oblivious bin structure. Experimental results show that our optimized approach can reduce LLC misses by 30% and TLB misses by 20%, resulting in up to a 2.6-fold speedup over the original BWA implementation.

1.1.2 Case Study - Optimizing Sequence Database Search on Multi-core CPUs

The other application is *sequence database search*, which is responsible for searching similar sequences for a query sequence in a sequence database. In this dissertation, we focus on BLAST (Basic Local Alignment Search Tool), which is a ubiquitous tool for database search due to its relatively fast heuristic algorithm delivering fairly high accuracy. However, because of BLAST's heuristic algorithm, it possesses heavy irregular control flows and memory accesses in order to narrow down the search space. Through an in-depth performance analysis of the original BLAST, we find that indexing the database instead of the query can better exploit the caching mechanism on modern CPU architectures. However, the database-indexed search with existing BLAST search techniques will suffer more from irregularities, leading to performance degradation, since it needs to align a query to millions of database sequences at a time rather than a single database sequence iteratively. Moreover, due to different challenges and characteristics between query indexing and database indexing, the existing techniques for query-indexed search are inefficient for the database-indexed search. To address these issues, we propose a methodology that first determines that the database-indexed BLAST algorithm is a *Multiple-Data-Multiple-Compute* class problem, and then accordingly proposes a re-factored BLAST algorithm, called muBLASTP, for modern multi-core CPUs. The key optimizations are that we first *decouple* the two interactive phases of the original BLAST algorithm into separate kernels, and then *reorder* data between the divided kernels with an efficient data reordering pipeline of filtering-sorting. Experimental results show that on a modern multi-core architecture, namely Intel Haswell, the multithreaded muBLASTP can achieve up to a 5.1-fold speedup over the multithreaded NCBI BLAST using 24 threads.

1.2 Optimizing Irregular Applications for Many-core Architectures

To achieve massively parallel computational power, many-core architectures feature a great number of compute cores but have simple control-flow mechanisms and cache mechanisms. Therefore, many-core architectures are highly sensitive to irregularities in both memory accesses and control flows. This makes the optimizations of irregular applications on many-core architectures more challenging. To eliminate irregularities for many-core architectures (i.e., GPUs), previous studies [7, 8, 9] use data and computation reordering to improve data locality and reduce branch divergence. However, these approaches have two major limitations (i.e., *local optimizations* and *"one-size-fits-all" methods*) that limit their effectiveness and efficiency, especially for complex irregular applications. To overcome the above limitations, we apply our general methodology to the BLAST algorithm (with query indexing) on a GPU as a case study and resolve the irregularity in the existing dynamic parallelism approaches.

1.2.1 Case Study - Optimizing Sequence Database Search on a GPU

Though recent studies have utilized GPUs to accelerate the BLAST algorithm for searching protein sequences (e.g., BLASTP), the complex irregularities in the BLAST algorithm prevent these approaches from applying fine-grained parallelism to achieve better performance. To address this, we present a fine-grained approach, referred to as cuBLASTP, to optimize the most time-consuming phases (i.e., hit detection and ungapped extension). In the optimization, we first *decouple* the two phases, which have different irregular patterns, into separate kernels, then map each kernel on a GPU with fine-grained parallelism, and then connect the two kernels with an efficient data *reordering* pipeline of binning-sorting-filtering for GPUs. Compared with the latest GPU implementation, cuBLASTP can deliver up to a 2.8-fold speedup for the overall performance.

1.2.2 Case Study - Optimizing Irregular Applications with Adaptive Dynamic Parallelism on a GPU

On recent GPU architectures, dynamic parallelism, which enables the launching of kernels from the GPU device without CPU involvement, provides a new way to improve the performance of irregular applications by generating child kernels dynamically to reduce workload imbalance and improve GPU utilization. In general, dynamic parallelism provides an easy way to *decouple* kernels to resolve the workload imbalance problem. However, in practice, dynamic parallelism generally does not improve performance due to the high kernel launch overhead and low occupancy. Consequently, most existing studies focus on mitigating the kernel launch overhead. As the kernel launch overhead has been progressively reduced due to algorithmic redesigns and hardware architectural innovations, the organization of subtasks to child kernels becomes a new performance bottleneck, which is overlooked in the literature.

In this dissertation, we perform an in-depth characterization of existing software-level approaches for dynamic parallelism optimizations on the latest available GPUs. We observe that the current approaches of subtask aggregation, which use the "one-size-fits-all" method that treats all subtasks equally, can underutilize the resources and degrade overall performance, as different subtasks require various configurations for the optimal performance. To address this problem, by taking advantage of statistical and machine-learning techniques, we propose a performance modeling and task scheduling tool that can 1) analyze the performance characteristics of subtasks to identify the critical performance factors, 2) predict the performance of new subtasks, and 3) generate the optimal aggregation strategy for new subtasks. Experimental results show that the implementation with the optimal subtask aggregation strategy can achieve up to a 1.8-fold speedup over the state-of-the-art task aggregation approach for dynamic parallelism.

1.3 Optimizing Irregular Applications on Multi-node Systems

On multi-node systems, load imbalance (or computational skew) is a fundamental problem. To tackle the problem of computational skew, many multi-node frameworks provide advanced mechanisms. For example, MapReduce [21] provides speculative scheduling to replicate the last few tasks of a job on different compute nodes. Furthermore, many mechanisms, including [22, 23, 24, 25, 26, 27, 28, 29, 30], are also proposed to mitigate skew by optimizing task scheduling, data partitioning, or job allocation. Although these runtime methods are able to handle skew to a certain extent, they cannot achieve near-optimal performance for irregular applications because the load balancing of many irregular applications not only relies on a single property (i.e., data size) but many other properties, such as the algorithm applied on the data and data distribution.

1.3.1 Case Study - Optimizing Irregular Applications with Adaptive Data Partition on Multi-node Systems

In our research, we propose *PaPar* [31], a framework that can generate balanced data partitioning codes for irregular applications on multi-node systems. In this framework, we first identify a set of common data operations, including *sort*, *group*, *split*, and *distribute*, as building blocks to reorder and redistribute the input data by multiple properties, and then provide an easy interface to construct a workflow of data partitioning with these building blocks. Finally, PaPar will map the workflow sequence to the multi-node systems. In our evaluation, we use two irregular multi-node applications (i.e., muBLAST [32] and PowerLyra [33]) to evaluate the performance and programmability of PaPar. Experimental results show that the codes generated by PaPar can produce the same partition quality as the original applications with less partitioning time.

1.4 Organization of this Dissertation

The remainder of this dissertation is organized as follows:

- Chapter 2 introduces background information, which discusses the characteristics of parallel architectures and irregular applications.
- Chapter 3 discusses the related works about existing optimization methods of irregular applications on parallel architectures.
- Chapter 4 introduces our methodology for irregular applications, involving our irregularity taxonomy, general transformations, and adaptive optimizations.
- Chapter 5 presents the optimizations of two irregular applications as case studies for multi-core architectures. First, we optimize Burrows-Wheeler Aligner (BWA) on multi-core architectures with *interchanging* and *reordering* transformations to improve data locality, presented in Section 5.1. In Section 5.2, we study the BLAST algorithm for protein sequences, and present a database-indexed search algorithm, called muBLASTP, which uses *decoupling* and *reordering* transformations to improve data locality and performance.
- Chapter 6 presents the optimizations of irregular applications on many-core architectures. In Section 6.1, we propose a re-factored BLAST algorithm for NVIDIA GPU, called cuBLASTP,

with *decoupling* and *reordering* transformations. In Section 6.2, we propose an optimized dynamic parallelism mechanism that *decouples* aggregated subtasks into separate kernels by their properties to improve GPU utilization. Moreover, we present performance models for dynamic parallelism to generate the optimal aggregation strategy.

- Chapter 7 presents the PaPar framework for automatically generating balanced data partitioning to alleviate the imbalance problem of irregular applications on multi-node systems.
- Chapter 8 presents the conclusion of the dissertation.

Chapter 2

Background

This chapter provides the background of this dissertation, including an introduction to current parallel architectures, and brief descriptions of irregular applications as case studies in this dissertation.

2.1 Parallel Architectures

In this section, we briefly introduce the characteristics of current parallel architectures, including multi- and many-core architectures, and multi-node systems, and the impacts of irregularities on these parallel architectures.

2.1.1 Multi-core Architectures

Fig. 2.1 shows the organization of a typical multi-core architecture. There are multiple identical cores connected together inside a chip packaging. With a shared unified memory space, different cores can share information and communicate each other by accessing the same memory locations. To handle the large gap between the processor and memory speed, multi-core architectures have a complex memory hierarchy. As Fig. 2.1 shown, the L1 cache is dedicated to each core, while the L2 cache can either be shared by a subset of cores or dedicated to each core. The L3 cache is large and shared by all cores in the processor. To speed up address translation for virtual memory systems, each core also has a dedicated Translation-Lookaside Buffer (TLB) for caching the address translation results.

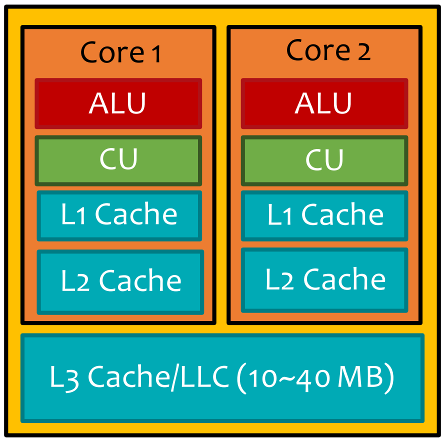


Figure 2.1: Example of a multi-core CPU architecture

The memory hierarchy takes the advantage of the data locality in a program. In particular, there are two types of locality, i.e., temporal and spatial locality. Temporal locality supposes that recently accessed address will be accessed in the near future, e.g. the accesses in loops or stacks. Spatial locality supposes that the addresses close to recent accesses will be accessed in the near future, e.g., sequential memory accesses on arrays.

Irregular applications can cause problems of latency and bandwidth [18]. The latency problem is due to poor temporal and spatial reuse that can result in elevated cache and TLB misses. The bandwidth problem is due to indirect references. In particular, when a data block is fetched into the memory hierarchy, the items within a block are only referenced few times before the block is evicted due to conflict and/or capacity misses, even though the block will be referenced later.

2.1.2 Many-core Architectures

Since it is hard to achieve high core count on multi-core architectures due to complex architectural designs, many-core architectures have been developed with hundreds or thousands of simple cores. Thanks to their high power efficiency and throughput, many-core architectures are increasingly popular in high performance computing (HPC) systems. In the recent Top500 list [3], 7 of the top 10 supercomputers uses GPUs or Intel MICs as accelerators.

To fit so many cores in a single processor, many-core architectures shrink non-computational-related resources, such as cache hierarchy, hardware prefetcher, and control flow units. As a result, compared with multi-core architectures, many-core architectures are more sensitive to irregular memory accesses and control flows. Below we use both NVIDIA and AMD GPUs as examples to introduce GPU architectures and explain the impacts of irregularities on many-core architectures.

2.1.2.1 NVIDIA CUDA Architecture

As Fig. 2.2 shown, a NVIDIA GPU contains a set of Streaming Multiprocessors (SMs or SMXs), each which consists of multiple Scalar Processors (SPs).

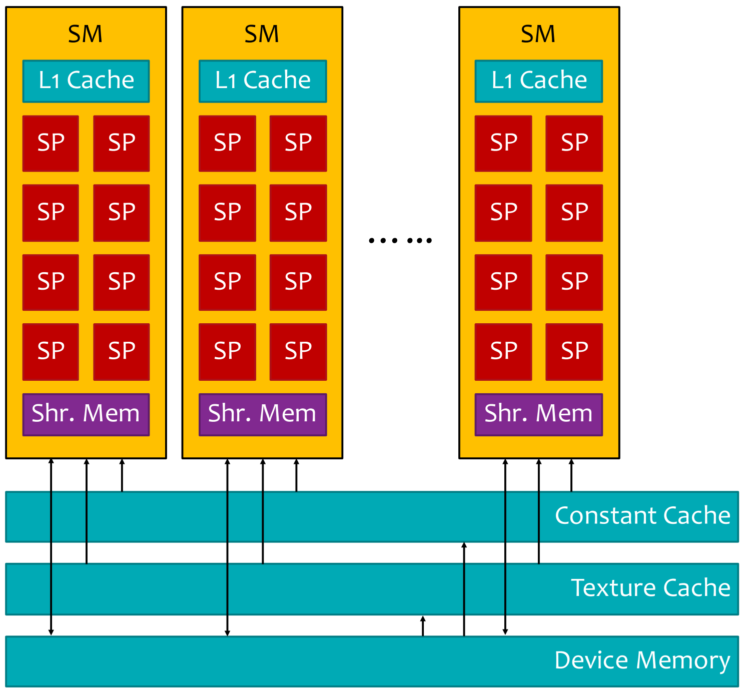


Figure 2.2: NVIDIA CUDA architecture

In each SM, all SPs must execute the same instruction in the same clock cycle. Otherwise, some SPs have to wait. This execution model is called “Single-Instruction, Multiple-Thread (SIMT)”. A group of threads (32 threads) that run together on an SM is called a warp. For each warp, the hardware handles divergent control flow by splitting threads into two warps and executing the two warps separately. Thus, a subset of threads (with a wasted slot) will skip the branch, and the other set of threads will take the branch (Fig. 2.3), called “branch divergence”, which significantly underutilize computing resources.

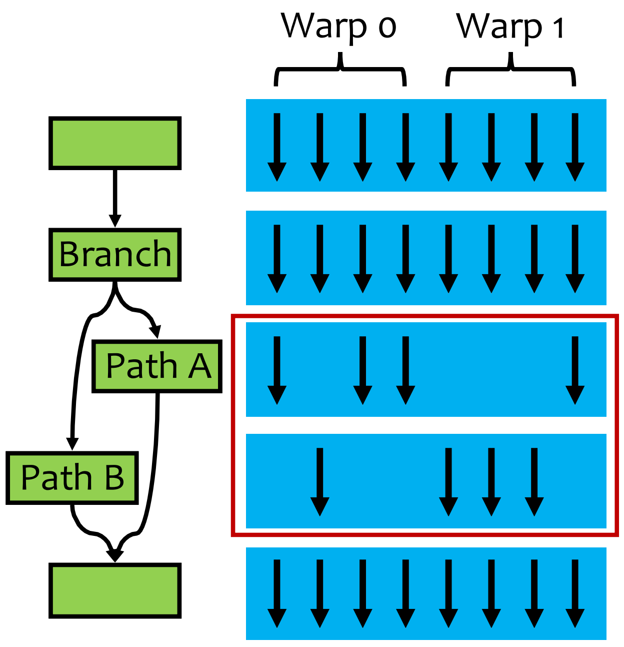
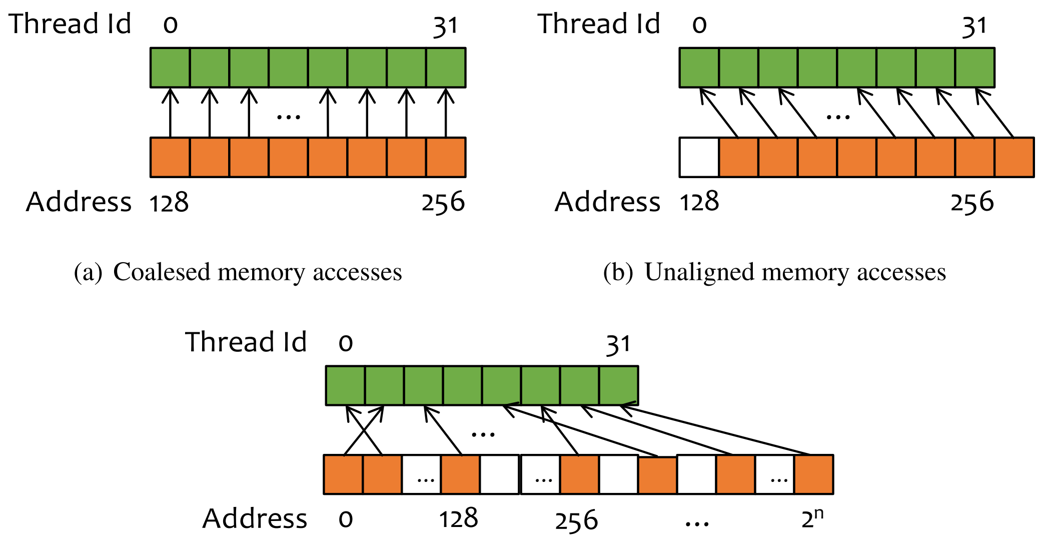


Figure 2.3: Example of branch divergence on a GPU

There are several levels of memory on the NVIDIA GPU (Fig. 2.2), each has distinct read and write characteristics. Overall, there are two types of memory on the GPU, i.e., on-chip and off-chip memory. On-chip memory, such as the register file and shared memory, has low access latency but small size. Off-chip memory, such as global memory, constant memory and texture memory, has much larger size but high access latency.

To efficiently access data in global memory, read/write operations must be coalesced. Specifically, if the memory accesses of a warp fall into an aligned 128-byte segment (32 single precision words), the hardware can read/write the data for the warp with a single memory transaction (Fig. 2.4(a)). If memory addresses are unaligned, i.e., across two 128-byte fragments, the hardware has to issue two transactions to load the data (Fig. 2.4(b)). Even worse, if irregular memory accesses of a warp crosses multiple segments, the hardware has to issue multiple transactions and load the data serially, which can result in the underutilization of memory bandwidth (Fig. 2.4(c)).



(a) Coalesed memory accesses (b) Unaligned memory accesses

(c) Memory accesses across N fragments

Figure 2.4: Examples of coalesced and irregular global memory access patterns

2.1.2.2 AMD GPU Architecture

Next, we take the latest AMD Radeon Vega 64 GPU (with AMD Vega architecture) as an example to introduce AMD GPU architectures. Fig. 2.5 illustrates that the AMD Vega 64 GPU contains 64 compute units (CUs), where each CU consists of 4 SIMD units, a dedicated L1 cache, and local memory. Threads in a kernel will be distributed across these CUs and processed by SIMD units. These 64 CUs are organized into four shader engines (SEs) with 16 CUs per SE. All CUs share L2 cache and memory channels through a crossbar.

Similar with NVIDIA GPUs, during the execution, threads on each CU will be divided into 64-thread wavefronts as basic execution units and processed by SIMD units concurrently. If threads in a wavefront take different execution paths, it will cause branch divergence. If threads within a warp access non-consecutive memory addresses, non-coalesced memory accesses occur, resulting in memory divergence.

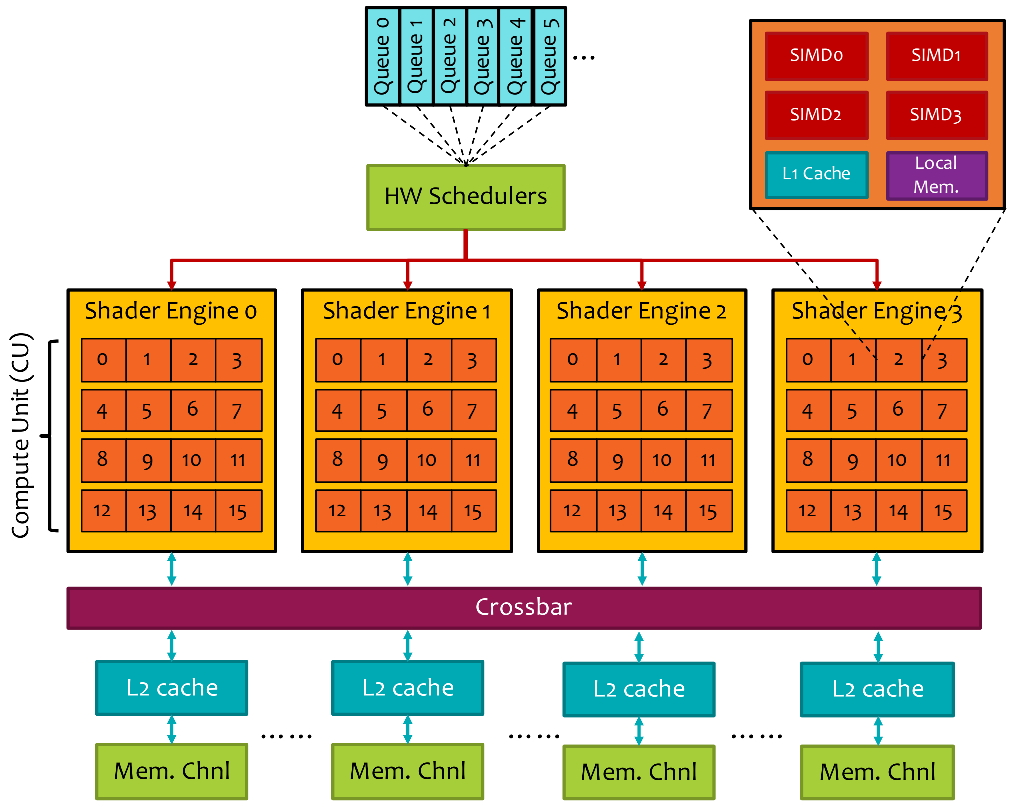


Figure 2.5: AMD GCN architecture [1]

2.1.2.3 GPU Programming Model

NVIDIA CUDA Programming Model CUDA [34] is a programming model provided by NVIDIA. As Fig. 2.6 shown, in a CUDA program, the computing system consists of the *host* that refers to the traditional CPU and its memory, and the *device* that refers to the GPU and its memory. Host codes executed by the CPU can call CUDA functions, i.e., GPU kernels. A kernel runs a large number of threads in parallel on the GPU. The threads are grouped into blocks, called thread blocks, where threads within a block cooperate via the shared memory, atomic operations, and barrier synchronization. All blocks in a grid (i.e., a kernel) have the same number of threads. Thread execution on the GPU follows a SIMT model, where threads of a block running on an SM are partitioned into small groups (i.e., warps) and execute the same instruction simultaneously. Each thread has a unique ID that it uses to decide what data to deal with.

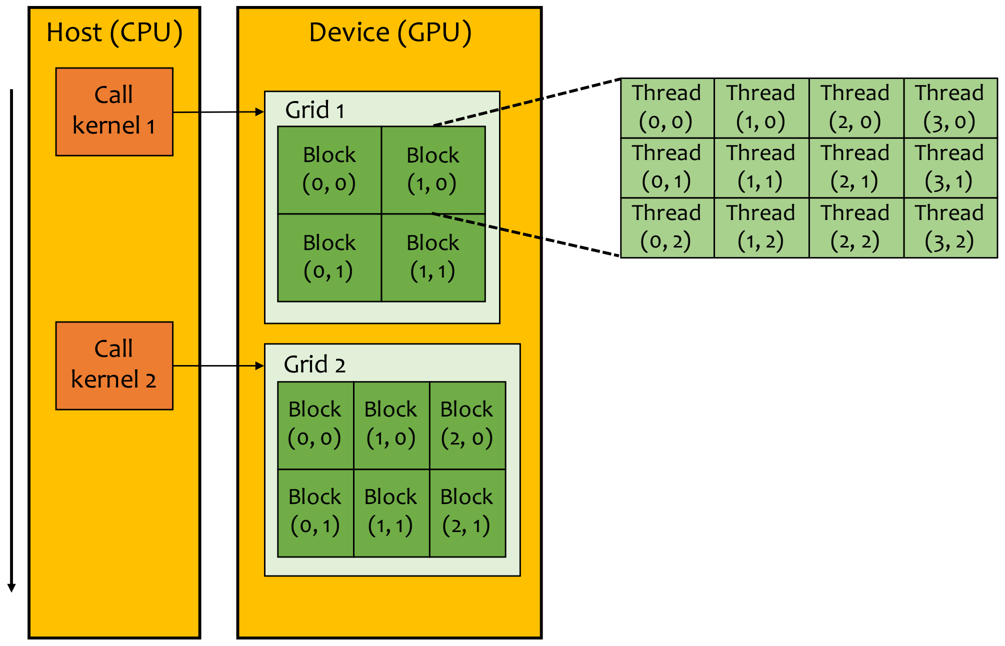


Figure 2.6: Example of the CUDA programming model

OpenCL Programming Model OpenCL (Open Computing Language) [35], which is defined by the Khronos Group, is a standard for parallel computing on heterogeneous systems. OpenCL kernels can be executed with multicore CPUs, AMD and NVIDIA GPUs, and DSPs (Digital Signal Processor) [36]. Similar to CUDA, in OpenCL programs, the *host* program executed by the CPU, while OpenCL kernels are executed on the devices. Threads (i.e., work-items) in an OpenCL kernel are grouped into workgroups, and each workgroup will be processed on an SM or CU that shares local memory.

AMD GPU Computing Software Stack

Radeon Open Compute platform (ROCm) ROCm [37] is an open-source platform created by AMD for GPU computing. ROCm supports multiple programming languages, such as HCC C+ and HIP, and OpenCL. In addition, the recent ROCM runtime (version 1.6) offers a new feature that allows the end user to provision individual CUs for kernel execution.

Asynchronous Task and Memory Interface (ATMI) ATMI [38] is a runtime framework and programming model for heterogeneous CPU-GPU systems. It provides a uniform API to create task graphs on both CPUs and GPUs. ATMI provides a sophisticated programming model to describe and fully control the high-level tasks simply by using a few predefined C-style structures.

2.1.2.4 Dynamic Parallelism in GPUs

Dynamic Parallelism (DP) is a feature, supported by both AMD and NVIDIA GPUs, to allow a GPU kernel to launch additional GPU kernels at the device side without CPU involvement (Fig. 2.7). Specifically, a parent kernel can launch child kernels and optionally synchronize on the completion of child kernels to consume the output of the child kernels.

In recent studies [39], Dynamic Parallelism has been used to improve the performance of irregular

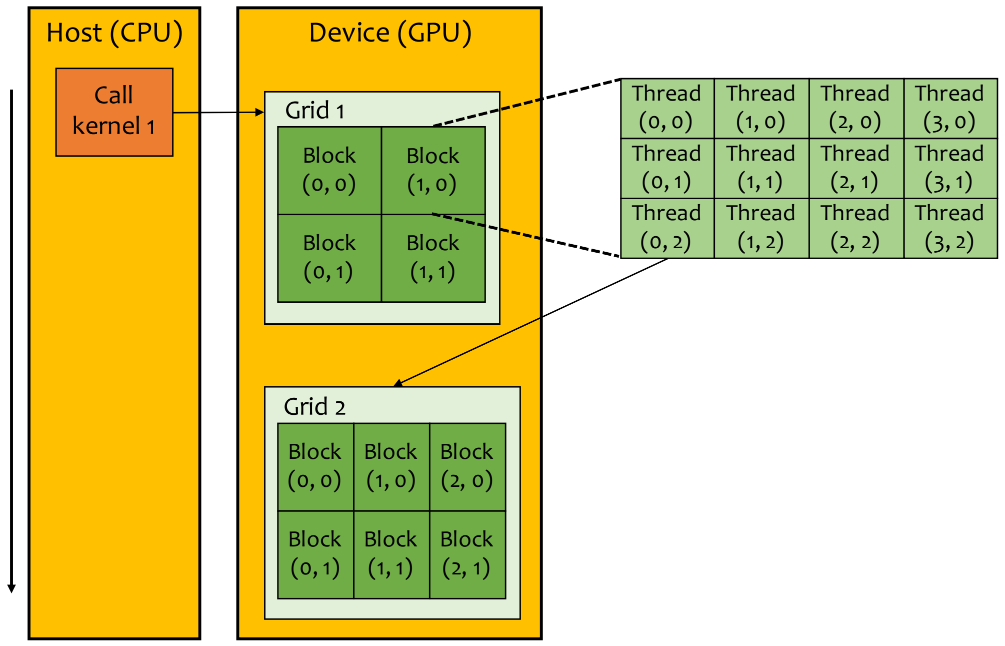


Figure 2.7: Example of Dynamic Parallelism

applications by alleviating workload imbalance and other irregularities. For example, as shown in Fig. 2.8, dynamic parallelism allows overloaded threads in the parent kernel to offload their subtasks to new child kernels (Line 5). And then, subtasks will be processed in fine-grained parallelism by multiple threads (Line 10) in child kernels, which can better exploit GPU compute and memory bandwidth.

2.1.3 Multi-node Systems

Nowadays, the dataset size of applications has been growing at an exponential rate. Therefore, we need to connect multiple machines (i.e., compute nodes) together to store and process the huge amount of data.

```
__kernel parent_kernel() {
  int tid = get_global_id(0);
  type *this_subtask = subtasks[tid];
  if(this_subtask->size >= THRESHOLD)
    kernel_launch(child_kernel, this_subtask);
  else
    process(this_subtask);
}

__kernel child_kernel(type *this_subtask) {
   int tid = get_global_id(0);
   // process this_subtask
   ...
}
```

Figure 2.8: Example of dynamic parallelism for resolving workload imbalance in irregular applications

2.1.3.1 Message-Passing Interface (MPI)

Message Passing Interface (MPI) [40] is a standardized and portable message-passing system based on the consensus of the MPI Forum. MPI is a message-passing parallel programming model, in which through cooperative operations between processes, data are moved from one process to another. The standard defines the interface for users to write portable message-passing programs. The MPI standard includes a set of functions that support Point-to-Point communication, Collective communication, Process groups, Process topologies and Datatype manipulation. Moreover, MPI also provides remote-memory access operations, dynamic process creation, and parallel I/O. MPI is a specification, not an implementation: there are several open-source and efficient implementations of MPI, such as OpenMPI [41], MVAPICH [42], MPICH [43], etc.

2.1.3.2 MapReduce

MapReduce [21] is a programming model and an associated implementation for simplifying large-scale data processing on commodity clusters. It was originally invented by Google, and now there are multiple implementations in different programming languages. Hadoop is a popular open-source implementation developed by Yahoo, being a part of Apache Hadoop [44].

A typical MapReduce job is composed of a *Map* and *Reduce* phase. The *Map* phase splits input datasets into data blocks, and processes them in parallel, and generates key-value pairs (KVP) based on user-defined functions. And then, the MapReduce framework internally sorts the pairs by the key and passes the sorted pairs to the *Reduce* phase. The *Reduce* phase computes and generates a new set of key-value pairs. These new key-value pairs can be output to users, or be input as another MapReduce job. The traditional file system could be a bottleneck as thousands of processors will access to the same file at the same time. Therefore, Google developed the robust distributed Google File System (GFS) [45] to support efficient MapReduce execution. The files in the GFS are automatically partitioned into fixed size blocks, which are distributed and replicated across multiple nodes. The file system is designed to provide high bandwidth, but has high latency for individual file operations, as most of the file operations in MapReduce are bulk read and write.

2.2 Irregular Applications

Irregular applications pertain in many domains, including both well established and emerging areas, such as machine learning, social network analysis, bioinformatics, and computer security. Though these applications have a significant degree of latent parallelism, it is difficult to scale on current parallel architectures due to their irregular and unpredictable memory access and computation patterns. Moreover, their data sets are difficult to be partitioned and balanced. In this dissertation, we select couples of complex and important irregular applications from bioinformatics as case studies.

2.2.1 Irregular Applications in Bioinformatics

As many bioinformatics problems are difficult to be solved optimally within polynomial time, heuristic methods are widely used to get near-optimal results in reasonable time. For example, with the advent of next-generation sequencing (NGS), which dramatically reduces the cost and time of DNA sequencing, the growth rate of sequence database has outpaced Moore's law, more than doubling each year (Fig. 2.9). But, heuristic methods employ irregular data structures (e.g., hash tables, suffix trees, lookup tables, etc.) and data-dependent conditional statements, which can cause complex irregular patterns. Below we introduce two important applications in bioinformatics, i.e., short read alignment and database search, as case studies to investigate the complex irregularities.

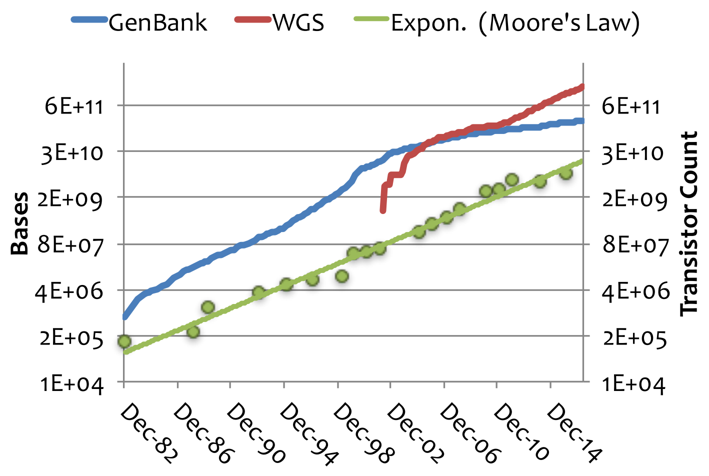


Figure 2.9: Growth rate of sequence databases vs. Moore's law

2.2.1.1 Short Read Alignment

In next-generation sequencing (NGS), millions or billions of DNA strands are sequenced in parallel, producing a huge amount of short DNA sequences, called reads. Mapping these reads to one or more reference genomes, referred to as short read alignment (Fig. 2.10), is fundamental in NGS data analysis, such as Indel detection [46]. Thus, tens of short read alignment tools have been developed over recent decades. In general, these tools can be classified into two categories: 1) hash table based tools, and 2) suffix/prefix tries based tools.

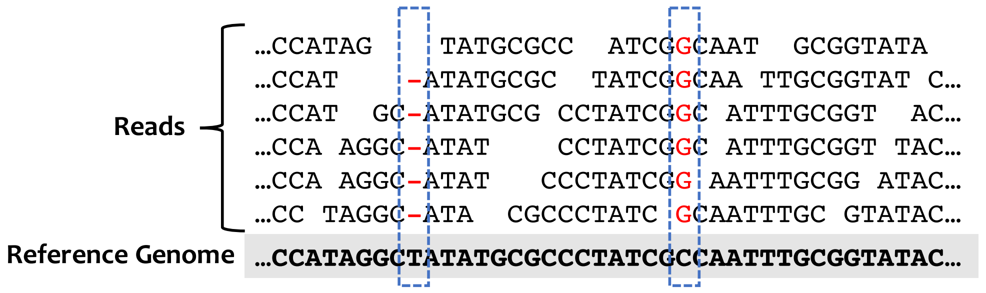

Reads
...CCATAG TATGCGCC ATCGGCAAT GCGGTATA
...CCAT -ATATGCGC TATCGGCAA TTGCGGTAT C...
...CCAT GC-ATATGCG CCTATCGGC ATTTGCGGT AC...
...CCA AGGC-ATAT CCTATCGGC ATTTGCGGT TAC...
...CCA AGGC-ATAT CCCTATCGG AATTTGCGG ATAC...
...CC TAGGC-ATA CGCCCTATC GCAATTTGC GTATAC...
Reference Genome
...CCATAGGCTATATGCGCCCTATCGCCAATTTGCGGTATAC...

Figure 2.10: Example of short read alignment

Hash table based tools [47, 48, 49] follow the seed-and-extend paradigm. The main idea is that the algorithm first rapidly finds short exact matches with fixed length (i.e., seeds) between the query and the reference genome by looking up a hash table, which contains all positions of fixed-length short sequences in the reference genome, and then extends and joins seeds without a gap, and finally refines high-quality ungapped extensions with dynamic programming. These tools can be fast and accurate. However, they usually are very memory consuming. For example, the hash table of the human genome can be tens of gigabytes.

Compared with hash table-based tools, tries-based tools, which use suffix/prefix tries to find short matches, have much smaller memory footprint. For example, the compressed index based on Burrows-Wheeler transform (BWT) only need 4 gigabytes for the human genome. Thus, tries-based tools, especially BWT-based tools [19, 50, 51], become increasingly popular. In this dissertation, we focus on Burrows-Wheeler Aligner (BWA), which is a popular BWT-based short-read alignment tool well optimized for multi-core architectures.

Burrows-Wheeler Aligner The Burrows-Wheeler Aligner (BWA) is based on the Burrows-Wheeler Transform (BWT), a data compression technique introduced by Burrows and Wheeler [52] in 1994. The main concept behind BWT is that it sorts all rotations of a given text in lexicographic order and then returns the last column as the result. The last column, i.e., the BWT string, can be easily compressed, because it contains many repeated characters. Similar to other BWT-based mapping tools, BWA uses the FM-index [53], a data structure built atop the BWT string that allows for fast string matching on the compressed index of the reference genome. In BWA, exact matching of a read (string) is done by a *backward search* [54], which essentially performs a top-down traversal on the prefix tree index of the reference genome. The backward search stage accounts for the vast majority of the execution time.

A brief description of the backward search in BWA is as follows.[1] For the string X, let $a \in \Sigma$ be the letter being considered and $c[a]$ be the number of symbols in $X[0, n-2]$ that are lexicographically smaller than a and $Occ(i, a)$ is the number of occurrences of a in the BWT string of X based on current position i. $c[a]$, $Occ(i, a)$ and the BWT string form the FM-index. String matching with the FM-Index tests if W is a substring of X, which is done by following a proven rule that $\underline{R}(aW) \leq \overline{R}(aW)$ if and only if aW is a substring of X:

$$\underline{R}(aW) = c[a] + Occ(\underline{R}(W) - 1, a) + 1$$

$$\overline{R}(aW) = c[a] + Occ(\overline{R}(W), a)$$

Iteratively applying the above rule, we get a narrowing search range declared by $\underline{R}(aW)$ and $\overline{R}(aW)$ (k and l in Algorithm 1) until $\overline{R}(aW)$ is less or equal to $\underline{R}(aW)$.

As Algorithm 1 shown, the occurrence calculation, i.e., the Occ function, of a is the core function in backward search. A trivial solution of implementing the Occ function is counting the occurrences of a in all previous position of the BWT string. This solution is inefficient when the BWT string is large. A widely accepted optimization, also used by BWA, is to break the whole BWT string into millions of small buckets and record pre-calculated counts of A/C/G/T for each bucket. BWA packages these pre-calculated counts along with the BWT string by inserting them at the head

[1]We use the same notations as the original BWA paper [19].

Algorithm 1 Original Backward Search

```
Input: $W$: sequence reads
Output: $k$ and $l$ pairs
1:  for all $W_j$ do
2:      $k = 0, l = |X|$
3:      for $i = len - 1$ to 0 do
4:          $a \leftarrow W_j[i]$
5:          $k \leftarrow c[a] + Occ(k - 1, a) + 1$
6:          $l \leftarrow c[a] + Occ(l, a)$
7:          if $k > l$ then
8:              output $k$ and $l$
9:              break
10:         end if
11:     end for
12: end for
```

of each BWT bucket. In this way, the occurrence calculation can be reduced to counting the occurrences within a bucket, which can be done in constant time. For example, in Fig. 2.11, the occurrence number of C in the position 3 at the bucket 1 equals to 31, fetched from the head of the bucket, plus 2, which is the count of C in the bucket before the position. This optimized BWT table is called FM-index, which is proposed by Ferragina and Manzini in 2001.

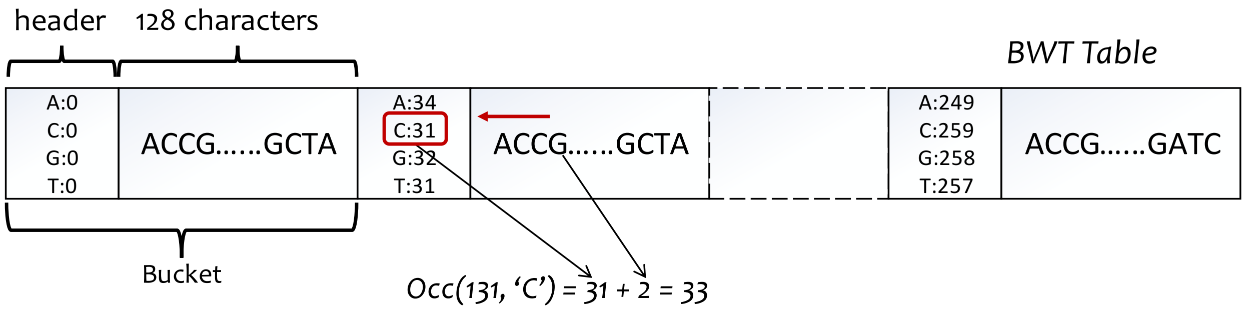


Figure 2.11: Memory layout of the BWT table

Based on the FM-index, the Occ function in BWA has three steps as shown in Algorithm 2: 1) getting the bucket location based on the input i, 2) fetching the pre-calculated count for letter a at the header of the bucket, and 3) counting the occurrences of a in the bucket and returning the sum of the local count and the pre-calculate count. Note that the memory-access location in the BWT table is determined by the input i.

Algorithm 2 Occ function

```
Input: $i$: $k$ or $l$ values; $a$: letter in reads
Output: $n$: occurrences of $a$
1: $p \leftarrow getBucket(i)$                ▷ Step 1
2: $n \leftarrow getAcc(p, a)$                ▷ Step 2
3: $n \leftarrow n + calOcc(p, a)$            ▷ Step 3 return $n$
```

2.2.1.2 Sequence Database Search

Sequence database search is responsible for finding the similarity between the query sequence and subject sequences in the database. The similarities can help to identify the function of the new-found molecule, since similar sequences probably have the same ancestor, share the same structure, and have a similar biological function. Sequence database search is also used outside of bioinformatics. For example, sequence database search is widely used into cybersecurity for data leak detection [55, 56, 57].

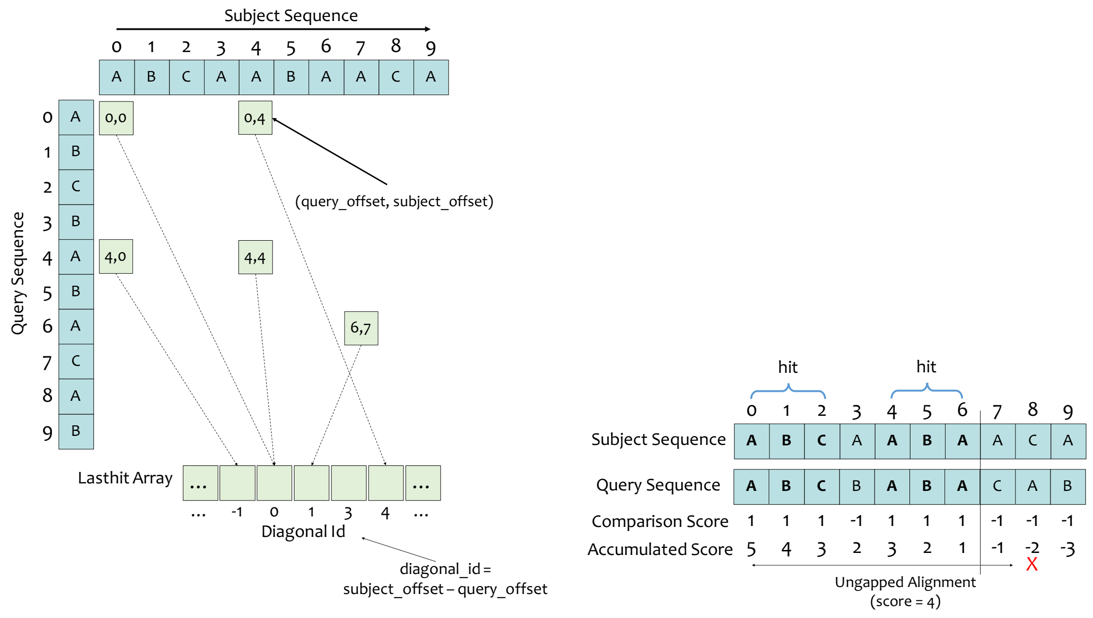
The dynamic programming algorithm is, e.g., Smith-Waterman algorithm, used for the optimal alignment of two sequences. Though the Smith-Waterman algorithm is well optimized on parallel architectures [58, 59, 60], the execution time is proportional to the product of the lengths of the two sequences, which is too slow for database search. Therefore, many tools use fast heuristic methods to improve search performance by pruning the search space based on the seed-and-extend paradigm. In this dissertation, we use BLAST (Basic Local Alignment Search Tool) as a case study.

Basic Local Alignment Search Tool BLAST is a family of programs to approximate the results of the Smith-Waterman algorithm. Instead of comparing the entire sequence, BLAST uses a heuristic method to reduce the search space. With only a slight loss of accuracy, BLAST executes significantly faster than the Smith-Waterman. In this dissertation, we focus on BLAST for protein sequence search, called BLASTP, which is more complicated than the other variants, e.g., BLASTN for nucleotide sequence search.

The BLASTP algorithm consists of the four stages as below:

Hit detection finds high-scoring short matches (i.e., hits) between the query sequence and the subject sequence from the database. The index, which is built on the query, records the positions of short segments with fixed length W, called words. The hit detection scans the subject sequence and searches each word in the query index to find the hits. Typically, W is 3 in BLASTP, and the words can be overlapped. For example, in Fig. 2.12(a), *ABC* at the position 0 and *BCA* at the position 1 are overlapping words in the subject sequence. To improve the accuracy, the neighboring words, which contains the word itself and the similar words to the word, are also considered to be hits. For example, the neighboring words *ABC* and *ABA* are treated as a hit to each other in Fig. 2.12(a).

Two-hit ungapped extension first finds the pairs of hits close together, and then extends hit pairs to basic alignments without gaps. The ungapped extension algorithm uses an array, called lasthit array *lasthit*, to update the last found hit for each diagonal. When a hit is found, the algorithm computes its distance to the last hit. If the distance is less than the threshold, the ungapped extension is triggered in both backward and forward directions. For example, in Fig. 2.12(a), when the hit (4,4) is found in the diagonal 0, the algorithm checks the distance to the last hit (0,0) in the same diagonal and triggers the ungapped extension, which ends at the position (7,7). Then, the ending position will be written back to the position 0 of the last hit array. Fig. 2.12(b) shows the details of the ungapped extension. Each step, the algorithm compares the differences between the corresponding characters from the query sequence and subject sequence, which is represented by a score. The ungapped extension stops when the accumulated score drops T ($T = -2$ in this example) below the maximum score.


Subject Sequence
0 1 2 3 4 5 6 7 8 9
A B C A A B A A C A
Query Sequence
0 A
1 B
2 C
3 B
4 A
5 B
6 A
7 C
8 A
9 B
0,0
0,4
(query_offset, subject_offset)
4,0
4,4
6,7
Lasthit Array
... -1 0 1 3 4 ...
Diagonal Id
diagonal_id = subject_offset – query_offset
hit
hit
0 1 2 3 4 5 6 7 8 9
Subject Sequence A B C A A B A A C A
Query Sequence A B C B A B A C A B
Comparison Score 1 1 1 -1 1 1 1 -1 -1 -1
Accumulated Score 5 4 3 2 3 2 1 -1 -2 -3
X
Ungapped Alignment
(score = 4)

(a) Hit detection (b) Ungapped extension

Figure 2.12: Example of the BLAST algorithm for the most time-consuming stages — hit detection and ungapped extension.

Gapped extension performs a gapped alignment with dynamic programming on the high-scoring ungapped regions to determine if they can be part of a larger, higher-scoring alignment.

Traceback re-aligns the top-scoring alignments from the gapped extension using a traceback algorithm, and produces the top scores. The ranked results will be returned to the user.

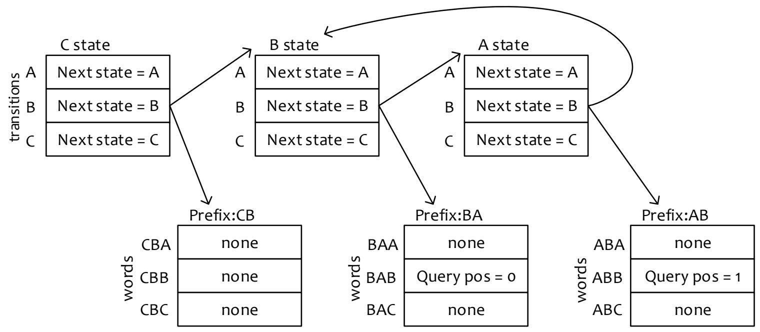
Based on [61], where 100 queries are randomly chosen from the NR protein database [62] and profiled, hit detection, ungapped extension and gapped extension consume the most time, taking nearly 90% of the total execution time. Thus, our work focuses on the optimizations of these three phases.

Below we describe the core data structures used in hit detection and ungapped extension: deterministic finite automaton (DFA) [63], position-specific scoring matrix (PSS matrix or PSSM), and scoring matrix.

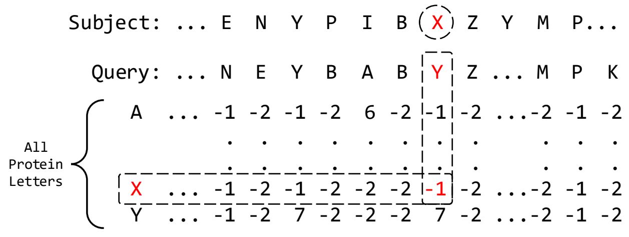
The *DFA* provides a general method for searching one or more fixed- or variable-length strings expressed in arbitrary, user-defined alphabets. In BLAST, the query sequence is decomposed into fixed-length short words and converted into a DFA. As an example, Fig. 2.13(a) shows the portion of DFA structure that is traversed with the example subject sequence “CBABB” processed (the word length is 3) and query sequence “BABBC”. First, the letter C is read, and the current state is set to C. Because the next letter is B, the next state of the DFA transitions to the B state. Simultaneously, the DFA provides a pointer to the CB prefix words to retrieve the query positions for the word CBA. Because the position for CBA in the DFA constructed from $BABBC$ is “none,” there is no hit found for CBA. Likewise, for the next letter A in $CBABB$, the DFA transitions to the A state and provides a pointer to the BA prefix words to retrieve the query positions for the word BAB, which is in the position 0 of $BABBC$, and so on.

The *PSS matrix* is built from the query sequence. As shown in Fig. 2.13(b), a column in the PSS matrix represents a position in the query sequence, and the scores in the rows indicate the similarity of all possible characters (i.e., amino acid) to the character in the column of the query sequence. So, the score for X in the subject sequence and Y in the query sequence is -1. By checking the PSS matrix, the BLAST algorithm can quickly identify the similarity between two characters in corresponding positions of the two sequences.

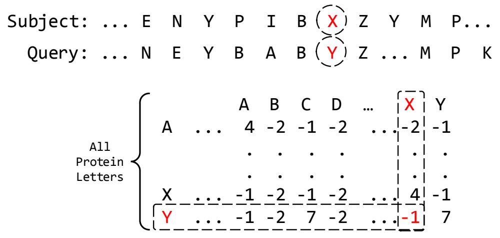
The *scoring matrix* is an alternative data structure of the PSS matrix. This matrix has a fixed and smaller size than the PSS matrix because the elements in the columns represent words instead of positions in the PSS matrix. The drawback in using this scoring matrix is that more memory accesses are needed. For example, to compare the same pair of characters as above, Fig. 2.13(c) shows the algorithm must first load the letter X from the subject sequence *and* Y from the query sequence, and then it can retrieve the score of -1 from the column X and row Y.



(a) Hit detection via DFA [63]*


Subject: ... E N Y P I B X Z Y M P...
Query: ... N E Y B A B Y Z ... M P K
All Protein Letters
A ... -1 -2 -1 -2 6 -2 -1 -2 ...-2 -1 -2
X ... -1 -2 -1 -2 -2 -2 -1 -2 ...-2 -1 -2
Y ... -1 -2 7 -2 -2 -2 7 -2 ...-2 -1 -2

(b) Scoring via the PSS Matrix [64]


Subject: ... E N Y P I B X Z Y M P...
Query: ... N E Y B A B Y Z ... M P K
All Protein Letters
A B C D … X Y
A ... 4 -2 -1 -2 ... -2 -1
X ... -1 -2 -1 -2 ... 4 -1
Y ... -1 -2 7 -2 ... -1 7

(c) Scoring via the Scoring Matrix [64]

Figure 2.13: Core Data Structures in BLAST. In Fig 2.13(a), $W = 3$ and the example query sequence is $BABBC$, and the example subject sequence is $CBABB$.

Chapter 3

Related Work

3.1 Optimization Methods of Irregular Applications on Parallel Architectures

In this section, we introduce the existing optimization methods of irregular applications on parallel architectures.

3.1.1 Optimization Methods for Multi-core Architectures

Since irregular memory accesses are a major performance issue on multi-core architectures, many studies focus on how to improve data locality. In the early study, Leung and Zahorjan [13] discuss how to choose a proper data layout to match the access pattern for a better spatial locality in nested loops. They propose a technique, called *array restructuring*, that improves the spatial locality exhibited by array accesses in nested loops. Specifically, the technique can determine the proper layout of array elements in memory that matches the given memory access pattern to maximize locality. Kandemir et al. [14, 15] propose an approach to improve the global data locality of nested loops via transforming both loop and data layouts. The authors claim that pure loop transformations are restricted by data dependencies and may not be successful in optimizing imperfectly nested loops. On the other hand, the data transformation on an array can affect all the references to that array in all loop nests. Thus, they propose an integrated approach that employs both loop and data transformations. Similarly, Boyle et al. [16] propose a framework to improve the locality of nested loops via combining loop and data transformations. However, these approaches use static methods at compile time. They can hardly resolve dynamic irregularities, where data access patterns remain unknown until runtime.

For the applications with dynamic irregularities, such as graph processing, sparse matrix operations, etc., many studies use dynamic methods that reorder data at runtime. Chen et al. [5] propose a dynamic approach for improving data locality through two methods: 1) grouping data, which will be accessed in the near future, and 2) packing data, which accesses in the same cache line. Strout et al. [6] propose an approach that performs data and iteration reordering transformations as minimal linear arrangements, and provide a corresponding metric that predicts performance and the cache behavior for selecting the optimal iteration-reordering heuristics. Pichel et al. [65] deal with irregular memory accesses in sparse matrix codes on multi-core CPUs via reordering data. As determining the optimal data layout is a classic NP-complete optimization problem, the authors solve it as a graph problem, i.e., *Traveling Salesman Problem*, using the *Chained Lin-Kernighan* heuristic.

Locality optimizations have also been developed for sparse linear algebra. The Reverse Cuthill-McGee (RCM) method can improve locality in sparse matrices by reordering columns using a reverse breadth-first search (BFS) [66, 67]. Al-Furaih and Ranka also study data partitioning using graph partitioning algorithm (METIS) and BFS to reorder data in irregular codes [68]. They conclude METIS yields better locality than BFS. They also evaluate different partition sizes for METIS and find partitions equal to cache size yielded the best performance. However, they do not consider the computation reordering or processing overhead.

Ding and Kennedy use dynamic copying of data elements based on the loop traversal order and show major improvements in performance [69]. They can automatically achieve most of their transformations in a compiler, using user-provided annotations. For adaptive codes, they reapply transformations after every change. Mellor-Crummey et al. use a geometric partitioning algorithm (RCB) based on space-filling curves to map multidimensional data to memory [70]. They also block computation using methods similar to tiling. Mitchell et al. improve locality using the bucket sorting method to reorder loop iterations in irregular computations [71]. They improve the performance of two applications, CG and IS, from NAS benchmarks, and a medical heart simulation. Bucket sorting works only for computations with a single irregular access per loop iteration. Strout et al. provide a uniform framework to handle locality transformations and improve the performance of irregular reductions by using sparse tiling [72]. Hwansoo Han and Chau-Wen Tseng characterize and compare the locality transformations for irregular codes [73]. And then, they develop parameters to guide both geometric (RCB) and graph partitioning (METIS) algorithms and develop a new graph partitioning algorithm based on hierarchical clustering (GPART) which achieves good locality with low overhead [74]. And then, they propose an adaptive method for exploiting the data locality for irregular scientific codes with Z-SORT reordering [75].

3.1.2 Optimization Methods for Many-core Architectures

Unlike with multi-core architectures, many-core architectures (i.e., GPUs and Intel MICs) are highly sensitive to irregularities in both memory accesses and control flows. Thus, many studies investigate both irregular memory accesses and control flows. To investigate the impacts of irregularities on GPU architectures, Burtscher et al. [76] define two measures of irregularity called control-flow irregularity and memory-access irregularity, and study the difference between irregular GPU kernels and regular kernels with respect to the two measures via the performance profiling on a suite of 13 benchmarks. Through quantitative studies, they make three important discoveries: 1) irregularity at the warp level varies widely, (2) control-flow and memory-access irregularities are highly independent of each other, and (3) most kernels, including regular ones, exhibit some irregularities. Baskaran et al. [77] propose a compiler framework for automatic parallelization and performance optimization of affine loop nests on GPUs. The framework optimizes the irregular applications in three ways: 1) performing a program transformation for efficient data access from GPU global memory, 2) determining the optimal padding factors for conflict-minimal data accesses from GPU shared memory, and 3) searching the optimal parameters for unrolling and tiling. Sung et al. [7] propose a compiler approach to transform the data layout of structured-grid applications on GPUs for a given model of the memory system. Jang et al. [78] present a methodology that optimizes memory performance on data-parallel architectures. In particular, the methodology first uses a mathematical model to analyze the memory access patterns inside nested loops, and then applies data transformation techniques for vector-based architectures (e.g., AMD VLIW GPUs) or uses an algorithmic memory selection approach for scalar-based architectures (e.g., NVIDIA GPUs), respectively. Merrill et al. [79] propose Breath-First Search (BFS) on GPUs with fine-grained thread blocks to adaptively explore the neighbors of vertices for better SMX utilization and fine-grained load balance. However, these studies use static methods, which only deal with the static irregular pattern at compile time.

For dynamic irregular patterns, previous studies use dynamic methods that reorder data and computation at runtime. Sung et al. [80] focus on the global memory accesses for GPU applications that access data in the Array-of-Structure (AoS) layout, and propose the Array-of-Structure-of-Tiled-Array (ASTA) that transforms the data layouts on-the-fly using in-place marshaling to reduce irregular memory accesses. To allow developers to leverage the benefits of ASTA with minimal effort, this work also provides a user-friendly automatic transformation framework. Zhang et al. [8] propose a dynamic approach, called G-Streamline, to eliminate the irregular memory accesses and control flows in GPU programs. G-Streamline utilizes two basic transformation mechanisms: 1) data reordering that creates a new data array, and stores the original data into the duplicated array in the regular order; 2) job swapping that packs data into the warp-sized buckets. To hide the overhead of data transformation, G-Streamline pipelines the data transformation on the CPU and the kernel execution on the GPU. Che et al. [9] propose an API, called Dymaxion, to help programmers to resolve the irregular memory accesses in GPU programs with hints about memory access patterns. In particular, the framework proposes a set of memory remapping functions for commonly used data layouts and access patterns in scientific applications. Instead of reordering data, Novak et al. [81] resolve the branch divergences in loops on GPUs via iteration scheduling that artificially delays and aggregates the selected iterations. Hou et al. [82] propose a novel auto-tuning framework that automatically finds the most efficient parallelizing strategy to achieve high-performance SpMV.

3.1.2.1 Optimizations with Dynamic Parallelism

Other than the conventional transformations, some research works use spawning dynamic subtasks via dynamic parallelism on the GPU to resolve irregularities. Wang et al. [39] characterize the benefits and overheads of dynamic parallelism for irregular applications. There are two major drawbacks in current dynamic parallelism mechanisms — high kernel launch overhead and low occupancy.

Therefore, there are multiple studies proposed to improve the efficiency of dynamic parallelism via subtask aggregation, which consolidates small kernels into coarse kernels. Wang et al. propose Dynamic Thread Block Launch (DTBL) [83], a hardware-based kernel aggregation that buffers subtasks in aggregation tables. To further improve the efficiency of dynamic parallelism, this work is enhanced by a locality-aware scheduler [84]. Orr et al. [85] also provide a kernel aggregation scheme in hardware for fine-grained tasks. However, the two approaches require hardware modification. Other than hardware-based kernel aggregation, there are multiple compiler-based approaches using kernel aggregation to reduce the number of kernel launches. Gupta et al. [86]

introduce the Persistent Thread (PT) programming style on GPUs, which occupies all the SMXs with a number of thread blocks, and dynamically generates tasks to improve workload balance across the SMXs. The same technique is also used in Pagoda framework [87] for GPU multiprogramming. Yang et al. propose CUDA-NP [88], which is a compiler-based approach for exploring nested parallelism via using slave threads in a thread block to process subtasks. Chen et al. [89] propose another compiler-based approach, called "Free Launch" that reuses the parent threads as persistent threads to process the child tasks. Wu et al. [90] propose a kernel aggregation for child kernels at three different granularities, including warp, block, and kernel-level. A similar work [91] is proposed by Hajj that aggregates kernels at the same three granularities and overlaps child kernel execution with the parent kernel via dispatching child kernels ahead of child tasks ready. Tang et al. [92] present "SPAWN" to improve the performance of parallelism by balancing subtasks in the parent and child kernels. However, all these software solutions mainly focus on reducing kernel launch overhead regardless of child kernel performance.

3.1.3 Optimization Methods for Multi-node Systems

Data partitioning and load balancing are important for parallel computations, especially for multi-node systems. Over the past few years, many efforts have been taken to explore the load imbalance (i.e., skew) problem. The speculative scheduling is the basic method of MapReduce that can speculatively relaunch last few tasks on other nodes. Zaharia et al. [22] propose the "LATE" scheduling algorithm that speculatively launches tasks having the longest estimation remaining time for heterogeneous systems. The Mantri system [23] restarts the task having the inconsistent runtime. The Flexslot system [93, 94] dynamically changes the numbers of slots for stragglers.

Skewtune [24] mitigates the skew for MapReduce applications by identifying the straggler, repartitioning its unprocessed input data, and rescheduling data to other nodes. Libra [28] determines that the keys having more values may become a performance bottleneck in the reduce stage and proposes a solution to repartition large keys with a new sampling method. OLH [25] proposes a key chopping method and a key packing method to split large keys and group medium keys, respectively. TopCluster [26] proposes a distributed monitoring framework to 1) capture the local data distribution on each mapper, 2) identify the most relevant subset data, and 3) approximate the global data distribution. This method provides complete information for appropriate skew-tacking methods. Although these mechanisms can mitigate the skew without the modification of applications, the effort to improve partitioning algorithms is still valuable, because application-specific partitioning methods can get better performance and scalability as illustrated in SkewReduce [95], PowerLyra [33], and Polymer [96].

SkewReduce [95] proposes a cost function based framework for spatial feature extraction applications manipulating multidimensional data. PowerLyra [33] is a graph computation and partitioning engine for skew graphs. The hybrid-cut method is proposed to partition input data. Polymer [96] is a graph processing engine for the NUMA compute node. A differentiated partitioning and allocation mechanism can put graph data into the local memory bank, and a NUMA-aware mechanism can convert random accesses on local memory to sequential accesses on remote memory.

3.1.4 Performance Modeling on Parallel Architectures

There are a large number of studies on performance modeling of traditional parallel architectures (i.e., multi-core CPUs). Lee and Brooks [97, 98] utilize the piecewise polynomial regression model to perform accurate performance prediction on a large uniprocessor design space of about one billion points. Ipek, et al. [99, 100] propose a performance prediction model of memory, core, and CMP design spaces with artificial neural networks. Marin and Mellor-Crummey predict application behavior via semi-automatically measuring and modeling program characteristics with properties of the architecture, properties of the binary, and application inputs [101]. Their toolkit provides a set of predefined functions and allows the users adding their customized functions. Carrington, et al. [102] develop a framework that predicts scientific computing performance and evaluates the framework for HPL and an ocean modeling simulation. Kerbyson, et al. [103] propose a predictive analytical model that can determine the performance and scalability of the applications of adaptive mesh refinement.

In recent years, with the increasing popularity of GPUs, there are a large number of research works [104, 105, 106] on performance analysis of GPU architectures. Many approaches utilize machine learning for performance and/or power modeling based on GPU hardware performance counters. For example, Zhang et al. [107] propose a statistical approach to identify the relationship between the characteristics of a kernel on a GPU and the performance and power consumption. To characterize the GPU performance, Souley et al. [108] propose a statistical model based on the Random Forest algorithm to characterize and predict the performance of GPU kernels. Rogers et al. [109] characterize the effect of the warp-size on NVIDIA GPUs. Stargazer [110] is an automated GPU performance framework based on stepwise regression modeling. Eiger [111] is an automated statistical methodology for modeling program behavior on different architectures. Though considerable attention has been focused on performance models to provide performance analysis and prediction on GPU architectures, none of them address the subtasks of dynamic parallelism in GPUs, which are tiny, irregular, and many in numbers.

3.2 Irregular Applications on Parallel Architectures

In this section, we introduce the related works of two important irregular applications, i.e., BWA and BLAST, which are case studies in the dissertation.

3.2.1 Burrow-Wheeler Transform based Alignment

Currently, most optimization studies on BWT-based mapping tools focus on accelerators. For instance, BarraCUDA [50], proposed by Petr Klus and Simon Lam, is a GPU-accelerated mapping tool adapted from BWA. BarraCUDA achieves a 3-fold speedup with 8 NVIDIA GPUs over a 12-core CPU. Another GPU-based short read aligner based on BWT released by Liu and Schmidt, CUSHAW [112], achieves a 4-6x speedup with 2 NVIDIA GPUs over a 4-core CPU. Recently, Torres and Espert propose another GPU-based alignment algorithm [113], which is three times faster than Bowtie and four times faster than SOAP2. Liang You et al. release optimized BWA for Intel Xeon Phi co-processors, which can achieve up to a 1.2x speedup over the 48 cores CPUs.

Besides many-core acceleration, there have been multiple studies [114, 115] in optimizing BWT-based alignment with FPGA (Field-Programmable Gate Array). Our research, on the other hand, focuses on optimizing BWT-based alignment on multi-core CPUs by remapping the algorithm to better exploit the caching mechanism of modern processors. In addition, our study performs in-depth performance characterization of BWA, which has not been reported by previous work.

Irregular memory accesses has also been observed for hash-table based short-read mapping tools. Wang and Tang [116] propose a memory optimization of hash-index for NGS by reordering memory access and compressing the hash-table. Another cache-oblivious algorithm based on a hash-table index, called mrsFAST, is proposed by Hach and Hormozdiari [117]. Our research work is the first to investigate the locality-aware implementation of BWT-based alignment, which involves more complicated data structures and more sophisticated memory-access patterns than hash-table based tools.

3.2.2 BLAST: Basic Local Alignment Search Tool

Many studies have conducted to improve the performance of BLAST tools because of its compute- and data-intensive nature. NCBI BLAST+ [118, 119] uses pthreads to speed up BLAST on a multi-core CPU. On CPU clusters, TurboBLAST [120], ScalaBLAST [121], and mpiBLAST [122] have been proposed. Among them, mpiBLAST is a widely-used one based on NCBI BLAST. With efficient task scheduling and scalable I/O subsystem, mpiBLAST can leverage tens of thousands processors to speed up BLAST. To achieve higher throughput on a per-node basis, BLAST has also been mapped and optimized onto various accelerators, such as FPGAs [123, 124, 125] and GPUs [126, 127, 128, 129, 130, 64, 131]. Relative to FPGAs, the work of Mahram et al. [125] is notable for its co-processing approach, which leverages both the CPU and FPGA to accelerate BLAST.

3.2.2.1 BLAST on GPUs

CUDA-BLASTN [132] is the first implementation of BLAST on a GPU for the nucleotide sequence alignments. After that, CUDA-NCBI-BLAST [128] is published for the protein sequence alignment. However, without GPU architecture-specific optimizations, CUDA-NCBI-BLAST only achieves up to a 2.7-fold speedup on an NVIDIA G80 GPU over a single-core Intel Pentium 4 CPU. Shortly thereafter, GPU-NCBI-BLAST [129] built on NCBI BLAST is proposed. The most time-consuming phases, including hit detection and ungapped extension, is ported to the GPU. With the same accuracy as NCBI BLAST, the authors report approximately a 4-fold speedup using an NVIDIA Fermi GPU over a single-threaded CPU implementation and a 2-fold speedup over a multi-threaded CPU implementation on a hexacore processor. CUDA-BLASTP [130] is proposed to use a compressed DFA for hit detection and an additional step that sorts the subject sequences by their lengths to improve the load balance. CUDA-BLASTP also ports the gapped extension on the GPU. GPU-BLASTP [64] improves the load balance further via a runtime work-queue design, where a thread could grab the next sequence after processing the current subject sequence. GPU-BLASTP also provides a two-level buffering mechanism, which writes the output of the ungapped extension first to a local buffer of each thread, and then to a global buffer if the local buffer is full. This mechanism avoids global atomics to write the output of different sequences, whose sizes could not be determined in advance. Most recently, G-BLASTN [131], based on NCBI-BLAST, is released for the nucleotide sequence alignment. With optimizations on GPU and parallelism on the CPU, including multithreading and vectorization, G-BLASTN achieves up to a 7-fold overall speedup over the multithreaded NCBI-BLAST for nucleotide sequence search on a quad-core CPU. Because BLASTN has already been implemented as a fine-grained algorithm, BLASTN does *not* have the challenges of BLASTP when mapped to GPU architectures.

3.2.2.2 Query Indexed BLAST

Since the hit detection is a time-consuming part of BLAST, to achieve higher throughput, couples of index data structures have been developed to boost the hit detection. For example, Deterministic Finite Automaton(DFA), which is introduced by FSA-BLAST [63], is multiple times smaller than the traditional lookup table and more cache-conscious. There are many studies that optimize the DFA structures for regular expression matching [133, 134, 135, 136]. To improve cache performance, NCBI-BLAST also introduces a couple of optimizations into the lookup table [137]. First is a *PV* array (presence vector), which uses a bit array to present if a cell in the lookup table contains query positions. The second is the thick backbone, where couples of query positions are embedded into the lookup table as there are few query positions for a cell. However, all these techniques are designed for the query derived index, which has many empty entries and thin entries (few positions in an entry). For the database index, as there are millions of positions of words from all subject sequences in the database, the lookup table will have zero empty entries, and every entry/word contains tons of positions.

3.2.2.3 Database Indexed BLAST

Instead of the hit detection based on the index delivered from the query, a serial of alternative approaches perform the hit detection based on the database index, such as SSAHA [138], CAFE [139], BLAT [140] and MegaBLAST [141]. Existing studies suggest that database-based index generally can deliver better performance, however, less sensitive than the BLAST algorithm [142, 143]. SSAHA and BLAT, for example, are significantly fast for finding near-identical matches. To reduce memory footprint and search space, both tools build indexes of non-overlapping words from the database, which leads to extremely fast search but compromised sensitivity. In particular, BLAT builds the database index with non-overlapping words of length W. With this approach, the size of database index is significantly reduced, roughly $\frac{1}{W}$ the size of an index with overlapping words. However, it requires a matching region of $2W-1$ bases between two sequences for guaranteeing to detect it. The CAFE is another search tool supporting protein sequence with database index, but the search method and scoring phase are substantially changed. Moreover, CAFE uses the large word size, which reduces the number of words that need to be compared, but impacts the sensitivity. MegaBLAST is a NCBI-BLAST variant based on the database index. MegaBLAST speeds up the search for highly similar sequences by using a large word size ($W = 28$), and thus reducing search workload and index size. But MegaBLAST only supports nucleotide sequences, as the authors claimed that it is very challenging to support protein sequence based on their design, such as the increased alphabet size, small word, and two-hit ungapped extension.

Chapter 4

Methodology

In general, there are three steps to optimize an irregular application for a parallel architecture (Fig. 4.1): (1) analyzing irregularities in the application, (2) exploiting the locality and regularity in the application, and (3) mapping the optimizations of the application to the target parallel architecture. Accordingly, our methodology provides three major techniques: (1) *Irregularity Taxonomy*, which classifies irregular applications into four classes based on the relationship between functions and data structures to help us analyze the causes and complexities of irregularities in the application; (2) *General Transformation*, which provides three general transformations on the computation and data structures to fully exploit the locality and regularity within the application; (3) *Adaptive Optimization*, which maps the transformations of the application to the target architecture based on the characteristics.

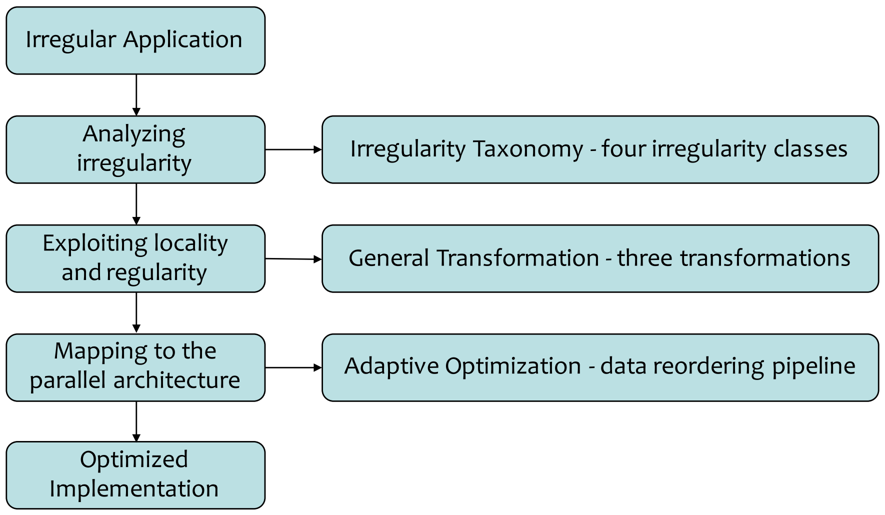

Irregular Application
Analyzing irregularity
Exploiting locality and regularity
Mapping to the parallel architecture
Optimized Implementation
Irregularity Taxonomy - four irregularity classes
General Transformation - three transformations
Adaptive Optimization - data reordering pipeline

Figure 4.1: Architecture of our methodology

4.1 Irregularity Taxonomy

Irregularities could have a variety of causes, complexities, and influences, which make analysis and optimization extremely difficult. To simplify the analysis of the irregularity in an application, we propose a taxonomy for irregular applications that classifies irregularities into four classes based on the relationship between functions and data structures. Irregularities in a class could have similar causes and complexities. Therefore, we can generalize the optimizations for each class. Here we borrow the terminology from Flynn's taxonomy [144] that is used for the classification of architectures. As shown in Fig. 4.2, from simple to complex, the four classes are *Single-Data-Single-Compute (SDSC)*, *Multiple-Data-Single-Compute (MDSC)*, *Single-Data-Multiple-Compute (SDMC)*, and *Multiple-Data-Multiple-Compute (MDMC)*. Below we give details of each class.

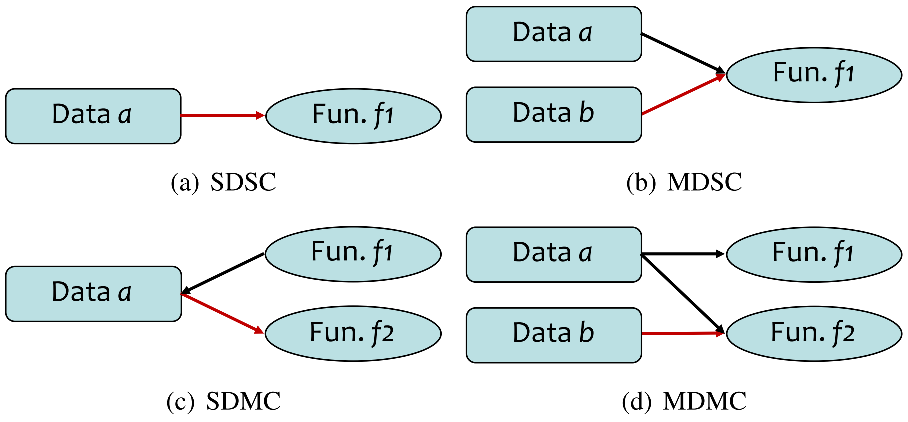


Figure 4.2: Examples (data flow diagrams) of irregularity classes. The red lines indicate the irregular memory accesses.

4.1.1 Single-Data-Single-Compute (SDSC)

The first class is *Single-Data-Single-Compute (SDSC)*, where a function operates on a single data structure. As an example shown in Fig. 4.2(a), the function $f1$ has irregular memory accesses on the data structure a. *SDSC* is a simple and common class, which generally occurs in irregular applications with an irregular data structure, such as graph traversal algorithms, sparse matrix operations, etc. In this class, the poor locality of memory accesses on the data structure is the performance bottleneck. Therefore, our optimizations focus on improving the data locality of the irregular data structure.

4.1.2 Multiple-Data-Single-Compute (MDSC)

The second irregularity class is *Multiple-Data-Single-Compute (MDSC)*, where a single function operates on multiple data structures. Fig. 4.2(b) gives an example of the MDSC class, where the function $f1$ operates on data structures a and b. In the example, the function $f1$ has regular memory accesses on the data structure a, but irregular memory accesses on the data structure b. *MDSC*'s irregularities are common in the search applications with index data structures, such as hash tables, lookup tables, search trees, etc. During the computation, the application loads the input item one by one with consecutive memory accesses and searches each input item in the index structure with irregular memory accesses. For example, the Burrow-Wheeler Aligner (BWA) algorithm (Section 5.1), the algorithm scans each short DNA sequence and checks the random positions in the BWT index.

The optimization for this class is more complex than the SDSC class. Besides the irregular memory accesses on the data structure b, we also need to take care of the interference between accesses on data structures a and b. Therefore, our optimizations should not only alleviate irregular memory accesses on the data structure b, but also avoid the interference between data structures a and b.

4.1.3 Single-Data-Multiple-Compute (SDMC)

The third irregularity class is *Single-Data-Multiple-Compute (SDMC)*, where multiple functions operate on a single data structure. Fig. 4.2(c) illustrates an example of the SDMC class, where the data structure a is operated by functions $f1$ and $f2$ as the output and input, respectively. However, the function $f1$ outputs the data structure a in a pattern (e.g., row-major order), but the function $f2$ consumes the data structure a in another pattern (e.g., column-major order). Therefore, the two functions have incompatible access patterns on the data structure a, which results in irregular memory accesses. This kind of irregularities could happen in the applications with multiple stages. For example, the BLAST algorithm (Section 5.2 and 6.1) has hit detection and ungapped extension stages operate on the hit structure. The hit detection generates hits in column-major order, but the ungapped extension processes hits in diagonal-major order. To resolve this kind of irregularity, we can reorder the data layout of the data structure a from the function $f1$ to fit the function $f2$ or unify the access patterns of functions $f1$ and $f2$.

Besides irregular memory accesses, the SDMC class also can cause irregular control flows (i.e., branch divergence on GPUs), when we map the applications of the SDMC class on a GPU. For example, as shown in Fig. 4.3, the execution of the function $f2$ depends on the result of the function $f1$. In the case, lane 0 (i.e., thread 0 with the warp) executes the function $f2$, while lane 1 does not. Therefore, lane 1 has to wait until lane 0 finishing the execution of the function $f2$. It can significantly underutilize the computation resource. To resolve this kind of irregularity, we can also reorder the data structure a to unify the compute patterns of threads/lanes within a warp.

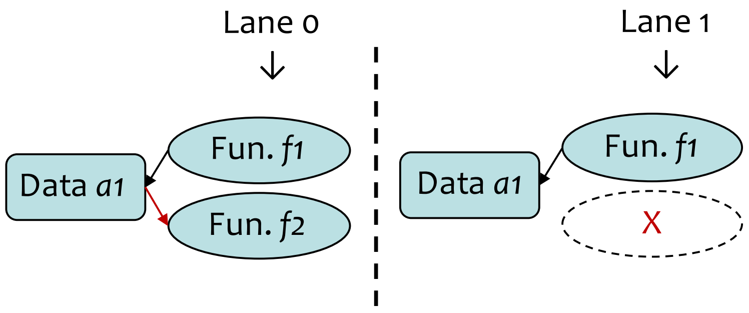


Figure 4.3: Example of branch divergence in SDMC

4.1.4 Multiple-Data-Multiple-Compute (MDMC)

The last irregularity class is *Multiple-Data-Multiple-Compute (MDMC)*, where multiple functions operate on multiple data structures. *MDMC* is the complex class where multiple irregular patterns coexist. For example, in Fig. 4.2(d), functions $f1$ and $f2$ operate on data structures a and b. In the example, the kernel has the incompatible access patterns between functions $f1$ and $f2$ on the data structure b, which occurs in the *SDMC* class, and also has the interference between accesses on data structures a and b, which occurs in the *MDSC* class. Such irregularity is common in complex algorithms (e.g., heuristic algorithms) having multiple stages that switch back and forth depending on the current status. For example, the BLAST algorithm uses heuristics methods where multiple alignment stages switch back and forth relying on the current alignment score. To deal with irregularities in this class, we first use the divide-and-conquer method to decouple the complex kernel into separate simple kernels, and then apply differentiated and fine-grained optimizations on each kernel, and then connect them together with data transformation.

4.2 General Transformation

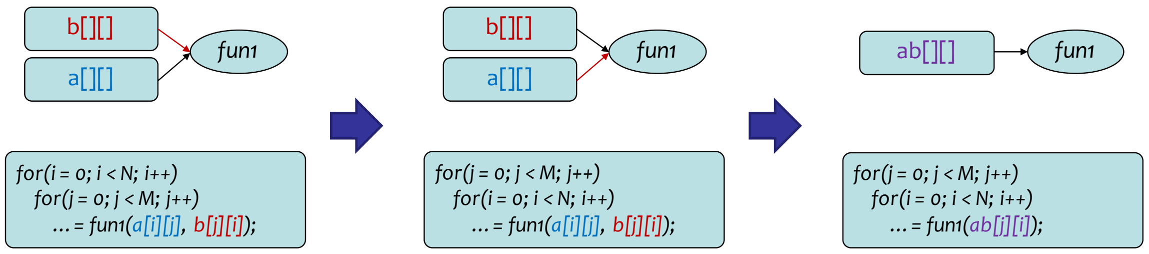
Based on the analysis of irregularity classes above, we abstract three general transformations, including *interchanging*, *reordering* and *decoupling*, to transform the computation and data for exploiting the locality and regularity in irregular applications. Below we detail each transformation and its challenges.

4.2.1 Interchanging

The *interchanging* transformation is to change the execution order of kernels/applications to exploit the locality and regularity across functions or kernels within them. Fig. 4.4 shows an example of the interchanging transformation for the MDSC class. In the example, the original kernel (Fig. 4.4(a)) has the function $fun1$ operate on arrays a and b, which both are stored in row-major order. However, the function $fun1$ accesses the array b in column-major order, which can result in strided memory accesses on the array b. Furthermore, we find the memory accesses on the array b are the performance bottleneck of the kernel. Therefore, to improve the data locality on the array b, we can exchange the execution order of the function $fun1$, i.e., interchanging the loops (Fig. 4.4(b)). After that, the memory accesses on the array b become to row-major order, which can improve the performance of the kernel due to better locality.

However, the interchanging transformation has a major limitation that it will alter the memory access pattern in the original algorithm, which may break the original locality and limit the performance gain. For example, in Fig. 4.4(b), after interchanging the loops, the memory accesses on the array a become to column-major order, which breaks the locality on the array a. To resolve this side effect, along with the interchanging transformation on the computation, we also apply the transformation correspondingly on the associated data structures. As Fig. 4.4(c) shown, in this example, we can combine arrays a and b into a single array ab where each element consists of the elements from the corresponding positions of arrays a and b. After that, the elements from arrays a and b at the same position will be referenced together, which improves the locality of both arrays

a and b.



(a) Original kernel (b) Interchanged kernel (c) Optimized Kernel

Figure 4.4: Example of optimizing irregularity with the *Interchanging* transformation.

4.2.2 Reordering

The *reordering* transformation is to reorganize data at runtime to make the data layout fit for the access pattern of the kernel/function. In our methodology, we use the reordering transformation to bridge two kernels/functions with different access patterns. For example, as a typical SDMC kernel shown in Fig. 4.5(a), the function $fun1$ outputs the array a in column-major order, but the function $fun2$ processes the array in row-major order. As a result, the function $fun2$ has irregular memory references on the array a. To solve this irregularity, we can add the *reordering* between functions $fun1$ and $fun2$ to transform the layout of the array a from row-major order to column-major order to eliminate the irregular memory accesses of the function $fun2$.

Besides resolving irregular memory accesses, the reordering transformation also can be used to deal with the irregularity in computation, e.g., workload imbalance or branch divergence. Fig. 4.6 shows an example of resolving branch divergence in a GPU kernel via reordering tasks. In the example, threads within a warp have variable-size tasks, resulting in branch divergence. After re-

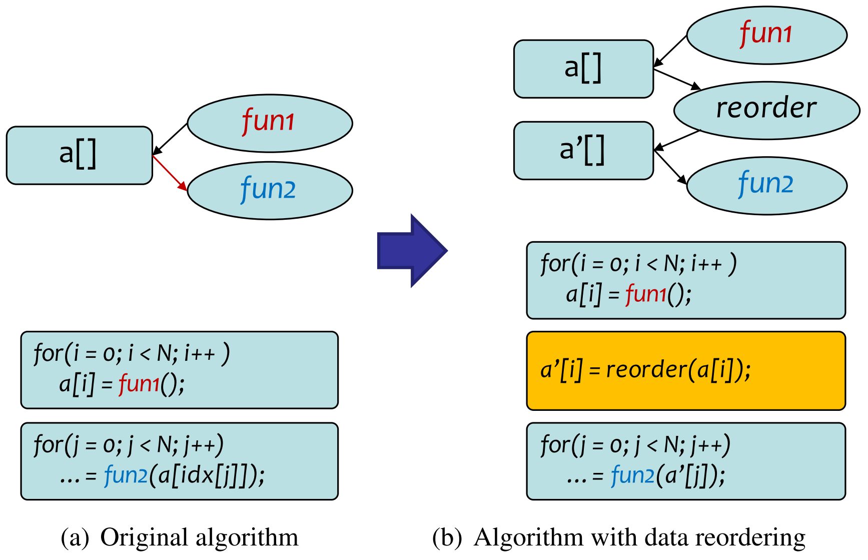


Figure 4.5: Example of optimizing irregularity with the *Reordering* transformation.

ordering threads by task size, the threads in a warp will have similar-size tasks and less divergence (Fig. 4.6(b)).

However, as the reordering transformation is dynamic, whose performance is critical to the overall performance gain, we need to minimize the overhead of the reordering transformation. In this dissertation, we provide adaptive optimization on the reordering transformation, which uses application-specific data reordering pipelines with architecture-aware mapping, presented in Section 4.3.

4.2.3 Decoupling

The *decoupling* transformation is to divide a complex kernel into small kernels with simple patterns, and then we can apply differentiated optimizations and fine-grained parallelism on each

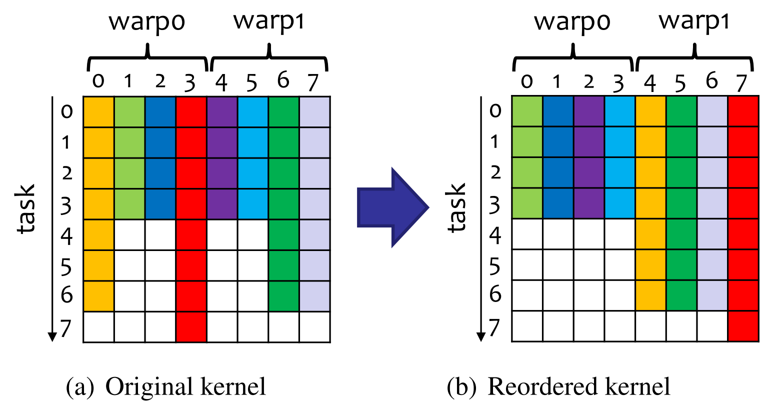
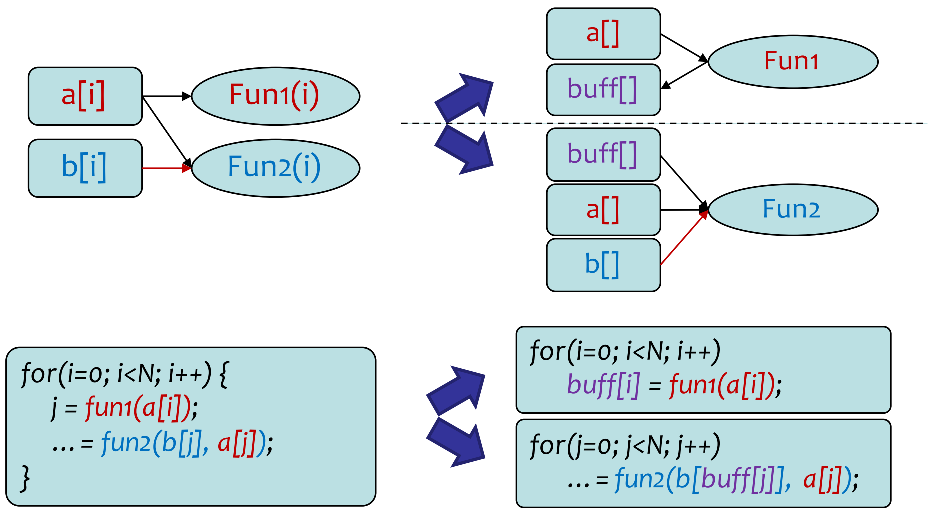


Figure 4.6: Example of resolving branch divergence in a GPU kernel with the *Reordering* transformation.

kernel. Fig. 4.7 illustrates a typical kernel of the MDMC class. In the example, both functions $fun1$ and $fun2$ operate on the array a, while the function $fun2$ operates on the b array. In this complex kernel, there are a couple of irregularities coexist: 1) the interference (i.e., cache contention) between accesses on arrays a and b. 2) the incompatible access patterns between functions $fun1$ and $fun2$ on the array b. It is difficult to resolve these two kinds of irregularities at the same time. Therefore, we divide the two functions into two separate kernels. After that, the first subkernel only contains the function $fun1$, operating on the array a, while the second subkernel only contains the function $fun2$, processing the outputs of the first subkernel with the arrays a and b. After decoupling, we only have the simple irregularity, i.e., incompatible access patterns between functions $fun1$ and $fun2$ on the array a, which can be easily solved by the reordering transformation. In this way, we simplify the irregular patterns and avoid the contention between functions $fun1$ and $fun2$.

Beyond resolving irregular memory accesses, the *decoupling* transformation also can be used to



```
for(i=0; i<N; i++) {
    j = fun1(a[i]);
    ... = fun2(b[j], a[j]);
}
```

```
for(i=0; i<N; i++)
    buff[i] = fun1(a[i]);
```

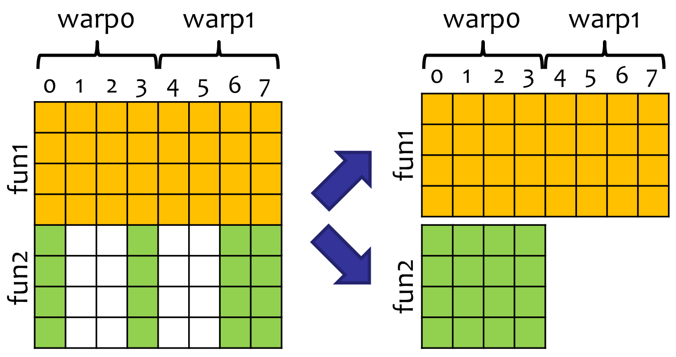
```
for(j=0; j<N; j++)
    ... = fun2(b[buff[j]], a[j]);
```

(a) Original kernel (b) Decoupled kernels

Figure 4.7: Example of optimizing irregularity with the *Decoupling* transformation.

deal with branch divergence in GPU kernels. For example, in Fig. 4.8, the GPU kernel contains functions $fun1$ and $fun2$. The execution of the function $fun2$ relies on the outputs of the function $fun1$, which can result in branch divergence in the kernel. To resolve this, we split functions $fun1$ and $fun2$ into separate kernels. After that, each kernel only contains a single function without branch divergence.

However, the decoupling transformation also has a drawback. It needs to buffer all intermediate results from the first kernel and pass to the second kernel, which could result in high memory pressure, especially for data-intensive applications, which can limit the scaling of the algorithm. For example, in the BLAST algorithm, the hit detection stage could generate a huge number of hits as intermediate results, which can be tens of gigabytes. To minimize the intermediate data size, we provide a set of techniques, including *data compression*, which reduces bits by identifying and eliminating statistical redundancy, *data blocking*, which limits intermediate data size in each iteration, and *data filtering*, which kicks out useless data. We will discuss these techniques in the case studies.



(a) Original kernel (b) Decoupled kernel

Figure 4.8: Example of resolving divergence in a GPU kernel with the *Decoupling* transformation.

4.3 Adaptive Optimization — Adaptive Reordering Pipeline

As discussed above, the reordering transformation is a dynamic, having runtime overhead. To obtain the optimal overall performance, in this section, we focus on minimizing the reordering overhead. To achieve this, we propose the *data reordering pipeline*, which combines multiple data reordering methods as build blocks to create a pipeline based on the demands of the applications, and *adaptive mapping*, which maps the pipeline onto the target architecture based on the characteristics of the architecture. Furthermore, to allow developers and researchers to build the optimal data reordering pipeline with less efforts, we present the concept of an *automatic framework* using machine learning techniques to direct the build of the data reordering pipeline.

4.3.1 Data Reordering Pipeline

Currently, there are many data reordering methods with various granularities, complexities, and use cases. However, the data structures and memory access patterns in irregular applications could be complex where a single reordering method cannot achieve the optimal effectiveness and efficiency. Therefore, we provide the data reordering pipeline that combines multiple reordering methods to reorder data step by step. As the example shown in Fig. 4.9, the pipeline starts with the coarse-grained data reordering method (e.g., the binning algorithm), and then switches to the fine-grained method (e.g., the sorting algorithm) when the input data get sufficiently small, and finally turns to the refinement method (e.g., the filtering algorithm) to adjust the sorted data.

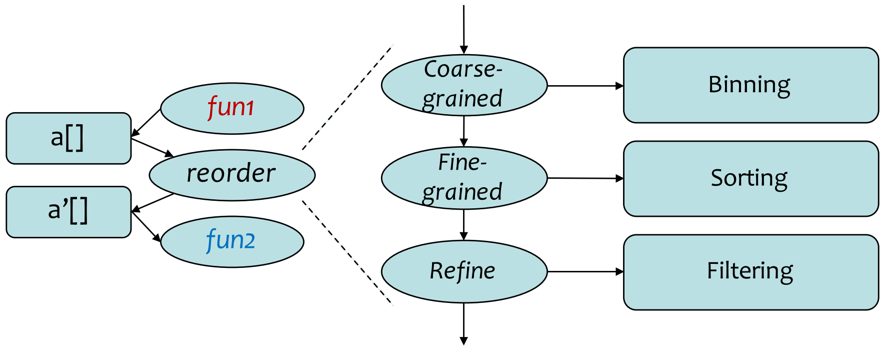


Figure 4.9: Example of the data reordering pipeline

However, building a proper data reordering pipeline for an irregular application is not trivial. As Table 4.1 shown, from coarse-grained to fine-grained, we have three reordering methods, including *binning*, *sorting* and *filtering*, where each has different complexity and effectiveness. Below we will discuss the details of each data reordering method and the placement of each one in the pipeline.

Table 4.1: Data Reordering Methods

Reordering method	Algorithms	Granularity	Time Complexity
Binning	One-level binning Hiearchical binning	Coarse-grained	$O(n)$
Sorting	Radix sort Merge sort Quicksort ...	Fine-grained	$\sim O(n \log n)$
Filtering	Array-based filtering Scan-based filtering	Refinement	$O(n)$

- **Binning** is a coarse-grained data reordering method that groups data into bins/buckets by keys. Since the binning method has low computational complexity (i.e., $O(n)$), we can use it to reorder large data roughly to improve data locality. First, we can use the binning method alone to group the memory accesses of adjacent memory locations into the same bin to improve data locality of the application directly. For example, in the optimization of the BWA algorithm on CPUs (Section 5.1), we use the binning method to group the access on the same bucket of the index into a bin to improve the data locality on the index. Second, we can use the binning to pre-process large input data into small blocks prior to fine-grained reordering methods. For example, in the optimization of the BLAST algorithm on a GPU (Section 6.1, we first group hits by diagonal id with the binning, and then sort each bin by position id. In this way, we can both reduce the computational overhead of the fine-grained methods (i.e., sorting) and improve its data locality.

- **Sorting** is a fine-grained data reordering method, which can provide precious data reordering, but has higher computational complexity (average $O(n \log(n))$ than the binning method.

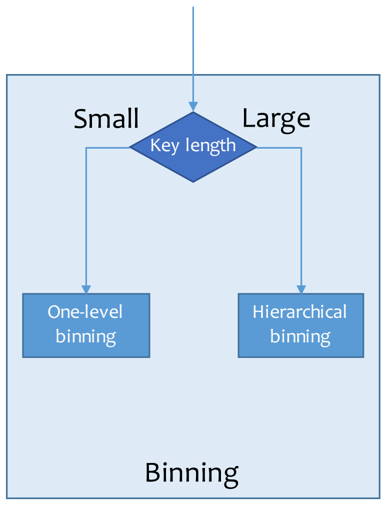
We can use the sorting to reorder data for kernels that have high requirements on memory access patterns. For example, GPU architectures, whose performance highly relies on the memory throughput, require coalesced memory accesses where all threads access consecutive and aligned memory locations. Thus, we need to use the sorting rather than the binning to achieve the coalesced memory accesses for better performance improvement. However, due to the high complexity of the sorting, we generally place the sorting into the pipeline after binning and filtering instead of using it alone.

- **Filtering** provides refinement on data with a small overhead ($O(n)$). It is mainly responsible for eliminating some data without changing the order of data as binning and sorting. For example, in our optimization of the BLAST algorithm on a GPU, after sorting, there are non-extendable hits, which make the memory accesses interleaved. Therefore, we scan the sorted data to determine non-extendable hits and filter them out. In this case, we can use the filtering to refine the reordered data, and compact memory accesses. The filtering also can be used to pre-process the data before sorting to reduce the complexity of the sorting. For example, in our optimization of the BLAST algorithm on CPUs, we perform the filtering to eliminate non-extendable hits before the sorting to reduce the overhead of the sorting.

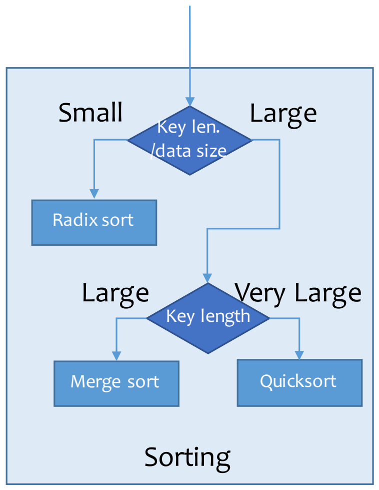
4.3.2 Adaptive Mapping

As Table 4.1 shown, each data reordering method could have multiple algorithms and implementations where each could have its suitable scenario. To achieve the optimal performance, as shown in

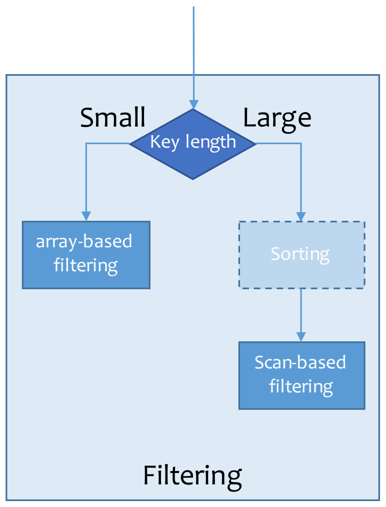
Fig. 4.10, we provide the adaptive mapping with the awareness of the application and architecture. Specifically, the adaptive mapping will select the proper algorithm and implementation for each data reordering method based on the characteristics of the application and architecture. Below we will discuss the adaptive mapping for each reordering method.



(a) Binning



(b) Sorting



(c) Filtering

Figure 4.10: Adaptive mapping and optimizations for *binning*, *sorting* and *filtering*

4.3.2.1 Binning

As the binning method is memory intensive, the performance of the binning algorithm highly relies on the memory throughput. To achieve the optimal performance, ideally, we need to fit the bin structure into the cache to reduce memory accesses. However, for the data with a large number of keys, it requires a large number of bins for all possible keys, which cannot fit into the cache and degrade the performance. Therefore, we provide a hierarchical binning algorithm that breaks the long keys in multiple segments and reorders the data segment by segment to make sure the bin structure for each level can fit into the cache. Thus, as shown in Fig. 4.10(a), our binning method uses the one-level binning algorithm for a short key length, and turns to use the multiple-level binning for a large key length.

4.3.2.2 Sorting

The optimization of the sorting reordering method is more complex than binning and filtering, since there are many sorting algorithms with variable characteristics. There are multiple factors to consider when selecting a proper sorting algorithm, such as data size, key length, and architectural characteristics. Below we use the three typical high performance sorting algorithms as case studies to discuss the selection of sorting algorithms.

Table 4.2: Characteristics of sorting algorithms

Sorting algorithm	Time Complexity	Space Complexity	Parallelism
Radix sort	$O(nw)$	$O(n+w)$	Thread-level (medium)
Merge sort	$O(n\log(n))$	$O(n)$	Thread/data-level (high)
Quicksort	$O(n\log(n))$	$O(\log(n))$	Thread-level (medium)

- **Radix sort** [145] is the non-comparison based sort algorithm, which has the lowest time complexity $O(wn)$, where w is the average key length, n is the number of elements. But radix sort has irregular memory access patterns, requiring high memory bandwidth. Moreover, radix sort requires fixed and small size keys, and more memory $O(n+w)$. Because of these characteristics, in our methodology, we use radix sort for small datasets and the datasets with short and fixed key length.

- **Merge sort** is a fast parallel sorting algorithm for large data with the time complexity of $O(n \log(n))$, and the space complexity of $O(n)$. Since Merge sort exhibits massive thread- and data-level parallelism, the state of the art studies [146, 147, 148, 149, 150, 151, 152, 153] show that parallel and vectorized merge sort with can deliver high and stable performance for large datasets on modern CPUs and GPUs.

- **Quicksort** is an efficient sorting algorithm for large datasets with good data locality and low storage complexity ($O(\log(n))$). However, quicksort is lack of data-level parallelism, compared with merge sort. Thus, in our work, we use quicksort for large datasets with very long keys, which are not suitable for data-level parallelism.

Based on the characteristics of sorting algorithms above, as the example shown in Fig. 4.10(b), we can build a decision tree to select the optimal sorting algorithm. If the data size is small enough, where all data can fit into the cache, we can use radix sort for low complexity. Otherwise, we select merge sort or quicksort. Between merge sort and quicksort, if the key length is very large (i.e., 128bit), we choose quicksort for better locality. Otherwise, we choose merge sort for better parallelism.

4.3.2.3 Filtering

Similar with the binning, the performance of the filtering highly relies on the size of the filtering data structure, which is mainly affected by the key length. The array-based method uses a large array to record the status of each key, and filter out unnecessary data by tracking the status of each key. However, the memory accesses on the array are irregular. On a CPU, which has a large cache, the array can easily fit into the cache to avoid irregular memory accesses. But, since GPUs have the small cache, the irregular accesses on the global memory can lead to poor performance. To resolve this problem, we can use the scan-based filtering with the sorting. After sorting, the data with the identical key are grouped into contiguous memory locations. Thus, we can perform the scan-based filtering via scanning and comparing the adjacent data with regular memory accesses (details are presented in Section 6.1). As a result, as shown in Fig. 4.10(c), our filtering method will use the array-based filtering for small keys with the large cache, and switch to the scan-based filtering for large keys with the small cache.

4.3.3 Automatic Framework

Though we provide guidelines above, it is still not easy to build a proper data reordering pipeline, as it requires substantial efforts with professional expertise and experience on the application and the architecture. It needs to evaluate possible data reordering pipeline strategies and select the optimal data reordering strategy for the target architecture. To simplify this process, we propose a concept of the adaptive framework that utilizes machine learning techniques to build performance models of the data reordering methods and direct developers and researchers to choose the optimal data reordering pipeline strategy.

Figure 4.12 illustrates a high-level depiction of the framework, which consists of three phases: *Performance Measurement* that runs the micro-benchmarks of provided data reordering algorithms with different parameters, and collects performance numbers on the target architecture, *Performance Modeling* that applies machine learning techniques on the collected performance numbers and generates performance models for reordering algorithms, and *Pipeline Generator* that searches the optimal pipeline strategy based on the predicted performance provided by performance models. Below we discuss the design of each phase.

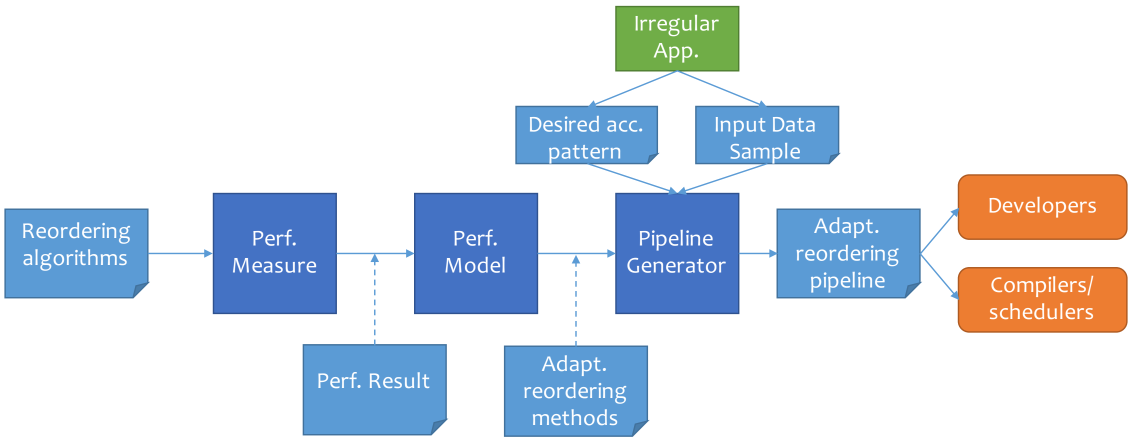


Figure 4.11: Architecture of the adaptive data reordering framework

Performance Measurement In this phase, we run a set of micro-benchmarks of data reordering algorithms with varying input parameters, such as data size, key length, and data distribution. During the benchmarks, the performance numbers, including execution time and performance counter data of the target architectures will be collected via profilers or profiling APIs.

Performance Modeling Based on the performance numbers collected in the previous phase, we can utilize machine learning techniques to build performance models for each data reordering method. For example, as Fig. 4.12 shown, we build performance models for each data reordering method that can predict the performance for the given input and parameter. Based on the predicted performance, we can select the optimal data reordering algorithm for different inputs. Moreover, we can build a decision tree based the performance models to direct the selection of the reordering algorithm for each data reordering method. For example, the model of the binning method generates a decision tree that can tell us the right binning algorithm for the specified key length and return the estimated performance. If the key length is large than the threshold T_b1, the binning structure cannot fit into the cache; the model will choose the hierarchical binning algorithm. And then, we can build the performance model of the data reordering pipeline via combining the performance models of data reordering methods. (Section 6.2 provides a concrete example of building performance models for determining the optimal strategy for dynamic parallelism on a GPU.)

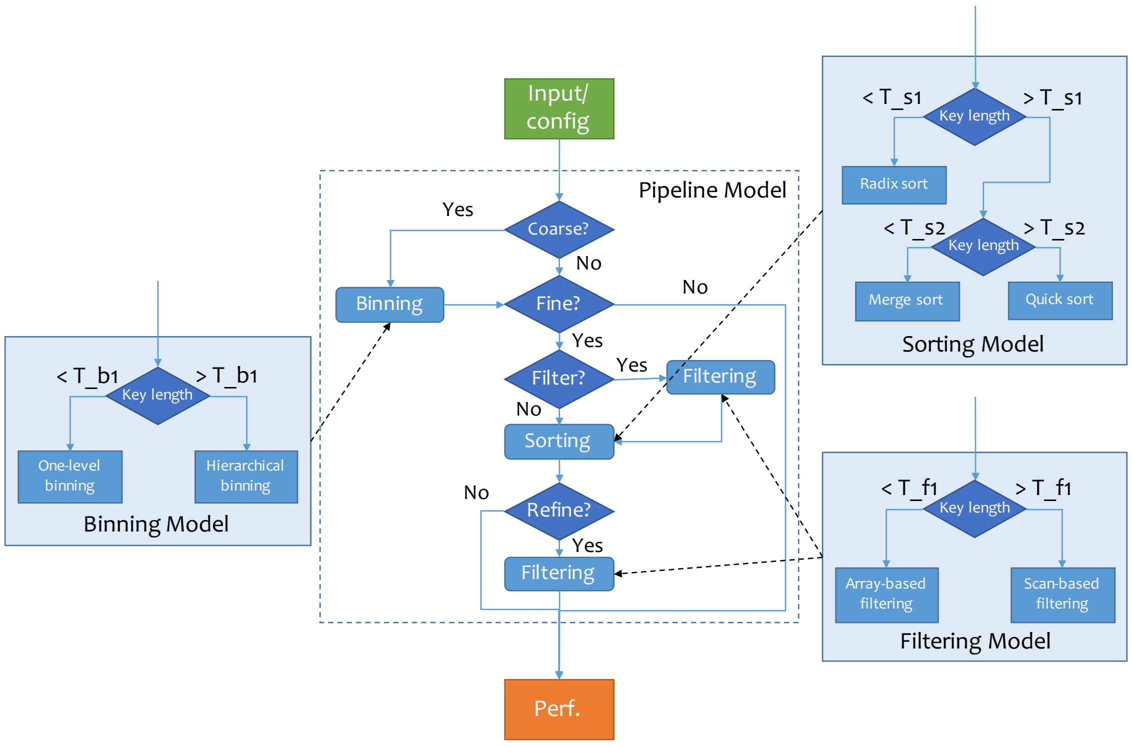


Figure 4.12: Example of the model of adaptive data reordering methods.

Pipeline Generation Based on the performance models above, we can search all data reordering pipeline strategies, and determine the optimal one based on the predicted performance. More specifically, we evaluate the overhead and performance gain of each data reordering pipeline strategy and select the optimal data reordering pipeline with the maximum overall performance gain after transformation costs.

4.4 Case Studies

To demonstrate our methodology, we choose couples of important irregular applications from different domains on different parallel architectures as case studies. Table 4.3 lists all the case studies in the dissertation with the irregularity class and transformations applied to them. From the table, we can see these case studies cover all irregular classes and transformations. Moreover, in the case study of adaptive DP (i.e., Dynamic Parallelism), we provide an example of building performance models for performance analysis and prediction using machine learning techniques. In the case study of PaPar, we provide an example of automatic code generation for irregular applications.

Table 4.3: Case Studies of Optimizing Irregular Applications

Case Study	Application	Architecture	Irregularity Class	Transformation
LA-BWA (Sec. 5.1)	Short read alignment	CPUs	MDSC	Interchanging Reordering
muBLASTP (Sec. 5.2)	Sequence search	CPUs	MDMC	Decoupling Reordering
cuBLASTP (Sec. 6.1)	Sequence search	GPUs	SDMC	Decoupling Reordering
Adaptive DP (Sec. 6.2)	Graph algorithm Sparse Matrix Ops	GPUs	SDSC, MDSC	Adaptive Decoupling
PaPar (Sec. 7.1)	Graph algorithm Sequence search	Multi-node	N/A	Reordering with Auto-Generation

Chapter 5

Optimizing Irregular Applications for Multi-core Architectures

5.1 LA-BWA: Optimizing Burrows-Wheeler Transform-Based Alignment on Multi-core Architectures

5.1.1 Introduction

Recently, next-generation sequencing (NGS) technologies have dramatically reduced the cost and time of DNA sequencing, making possible a new era of medical breakthroughs based on personal genome information. A fundamental task, called short read alignment, is mapping short DNA sequences, also called reads, that are generated by NGS sequencers, to one or more reference genomes, which are really big. Many short read alignment tools based on different indexing techniques have been developed during the past couple of years [154]. Among them, alignment tools based on the Burrows-Wheeler Transform (BWT), such as BWA [19], SOAPv2 [155], and Bowtie [51] have become increasingly popular because of their superior memory efficiency and support of flexible seed lengths. The Burrows-Wheeler Transform is a string compression technique that is used in compression tools such as bzip2. Using the FM-index [54], a data structure built atop the BWT, BWT-based alignment tools allow fast mapping of short DNA sequences against reference genomes with a small memory footprint.

State-of-the-art BWT-based alignment tools are well engineered and highly efficient. However, the performance of these tools still cannot keep up with the explosive growth of NGS data. In this study, we first perform an in-depth performance analysis of BWA, one of the most widely BWT-based aligners, on modern multi-core processors. As a proof of concept, our study focuses on the exact matching kernel of BWA, because inexact matching is typically transformed into exact matching in BWT-based alignment. Our investigation shows that the irregular memory access pattern is the major performance bottleneck of BWA. Specifically, the search kernel of BWA is a typical irregular pattern in the *MDSC* class, which shows poor locality in its memory access pattern, and thus suffers very high cache and TLB misses. To address these issues, we propose a locality-aware design of the BWA search kernel, which *interchanges* the execution order to exploit the potential locality across reads, and *reorders* memory accesses to better take advantage of the caching and prefetching mechanism in modern multi-core processors. Experimental results show that our improved BWA implementation can effectively reduce cache and TLB misses, and in turn, significantly improve the overall search performance.

Our specific contributions are as follows:

1. We carry out an in-depth performance characterization of BWA on modern multi-core processors. Our analysis reveals crucial architecture features that will impact the performance of BWT-based alignment.

2. We propose a novel locality-aware design for exact string matching using BWT-based alignment. Our design refactors the original search kernel by grouping together search computation that accesses adjacent memory regions. The refactored search kernel can significantly improve memory access efficiency on multi-core processors.

3. We evaluate the optimized BWA algorithm on two different Intel Sandy Bridge platforms. Experimental results show that our approach can improve LLC misses by 30% and TLB misses by 20%, resulting in up to a 2.6-fold speedup over the original BWA implementation.

5.1.2 Performance Characterization of BWA

In order to understand the performance characteristics of BWA, we collect critical performance counter numbers, such as branch misprediction, I-Cache misses, LLC misses, TLB misses, and microcode assists, using Intel VTune [156]. Fig. 5.1 shows the breakdown of cycles impacted by different performance events. As we can see, the percentage of stalled cycles is overwhelmingly high (more than 85%). Clearly, cache misses and TLB misses are the two major performance bottlenecks of backward search. Together, the two account for over 60% of all cycles. A closer look at the profiling data shows that the main source of these misses is the Occ function, which is the core function in backward search and accounts for over 80% of total execution time. Based on profiling numbers, within the Occ function, the stalled cycles caused by cache misses account for 55% of overall cycles, and TLB misses caused stalled cycles occupy 41%. Thus, our optimization strategy focuses on how to optimize the memory access of the Occ function.

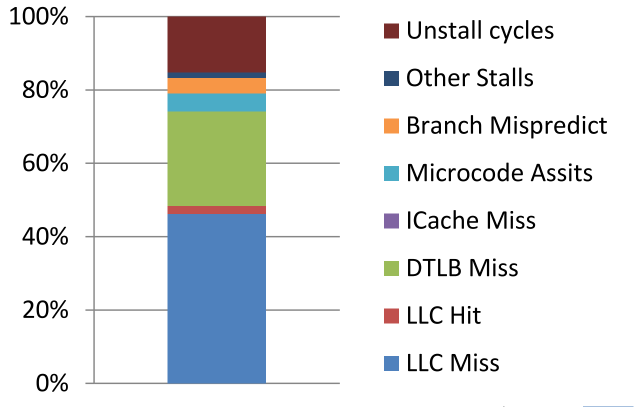


Figure 5.1: Breakdown of cycles of BWA with Intel Sandy Bridge processors

As we mentioned in Section 2.2.1.1, in the Occ function (Algorithm 2), the input i (k or l in backward search) determines the access location in the BWT table. In order to further understand the memory access pattern of the Occ function, we trace the buckets that need to be accessed in calculating ks when searching an input read. As shown in Fig. 5.2, the access location in the BWT table jumps irregularly with large strides. Also, there seems to be little locality between consecutive access locations. Thus, we can classify the BWA kernel into the *MDSC* class, where the irregular memory access is the main reason for the high cache-miss rate. Furthermore, as the capacity of TLB is limited, large strides (e.g., larger than the 4K page size) over the BWT table can cause high TLB misses.

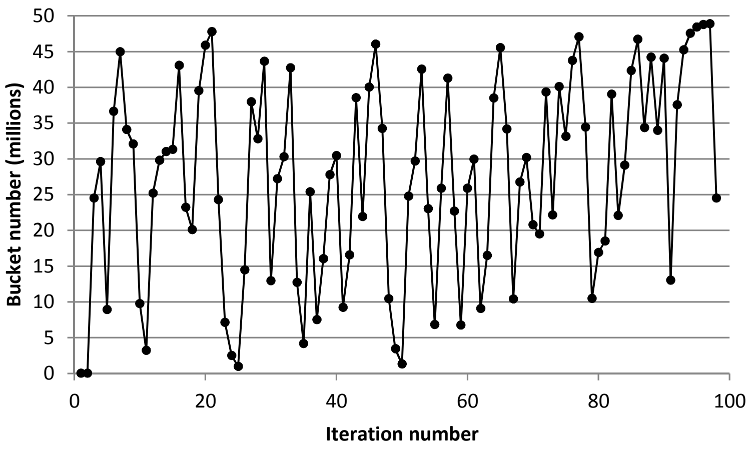


Figure 5.2: The trace of k in backward search for a read

Clearly, the backward searches of individual reads suffer from poor locality. However, we observe the potential locality across processing of different reads. To reduce I/O overhead, BWA loads millions of reads into memory as a batch. It is highly probable that multiple bucket accesses from different reads will fall into the same memory region. This observation is the main motivation of our optimizations, which will be presented in Section 5.1.3.

5.1.3 Optimization

In order to improve memory-access efficiency in BWT-based alignment, we propose a locality-aware backward search design, which exploits the locality of memory accesses to the BWT table from a batch of reads. Specifically, as discussed in Section 5.1.2, the computation of occurrences, i.e., the Occ function, is the main source of cache and TLB misses.

5.1.3.1 Exploit Locality with Interchanging

As shown in Algorithm 3, our design batches the occurrence computation from different reads by interchanging the inner and outer loops of the original BWA implementation. After interchanging, in each outer iteration, we compute the occurrences of the position in all reads. And then, we can exploit the hidden locality across reads via reordering memory accesses by grouping together the occurrence computation that accesses the adjacent buckets of the BWT table.

5.1.3.2 Reordering Memory Access with Binning

To reorder memory accesses, our design maintains a list of bins, where each bin corresponds to several consecutive buckets in the BWT table. The binning method can improve the data locality with rough data reordering of low overhead. Designing a highly efficient binning algorithm here is challenging as it involves many competing factors. For instance, while binning can help improve memory access in occurrence computation, it also introduces extra memory accesses that can lead to undesirable cache and TLB misses. The design is also complicated by the complex memory hierarchy and prefetching mechanisms of modern processors.

Memory-Efficient Data Structure The preliminary data structure of a bin entry is depicted in the left picture in Fig. 5.3. k, l and r_id are the input of the refactored Occ function, where k and l are corresponding top and $bottom$ in the original BWA implementation, and r_id is the id of the read being processed. Besides the three prerequisite variables, we add a small character array for data preloading to help reduce memory access overhead (discussed in Section 5.1.3.3).

Such a bin entry requires 28 bytes to store. However, because of the data structure alignment, each element should occupy a multiple of the largest alignment of any structure member with padding, i.e., actually requires 32 bytes in memory. For a large batch of reads, there can be millions of entries, which can consume gigabytes of memory. To preserve the memory-efficiency of BTW-based alignment, we optimize the preliminary data structure as follows.

First, we observe an interesting property of k and l that can help shrink the data structure of a bin entry. In the original BWA implementation, the k and l are 64-bit integers, occupying 16 bytes in total. However, for the human genome, the maximum values of k and l are less than 2^{34}. Therefore, storing k or l only requires 33 bits, wasting the remaining 31 bits (in a 64-bit integer). To improve memory utilization, we pack k and l into a single 64-bit integer such that k takes the first 33 bits, and l is represented as an offset to k in the remaining 31 bits. By doing so, the size of the bin structure can be significantly reduced. However, such a design requires the offset between k and l to be less than 2^{32}. Extensive profiling using data from the 1000 Genome Project [157] shows that the distance between k and l is always less than 2^{31} except the first iteration (example statistics are shown in Fig. 5.4(a). This can also be explained in theory because the FM-index mimics a top-down prefix tree traversal, and as such, the distance between k and l decreases quickly as more letters are matched. Based on this observation, we package k and l by just skipping the first iteration. In addition, the trend shown in Fig. 5.4(b) implies that our method can be easily extended and used for larger genomes by skipping more initial iterations.

Second, cc is a small character array used to temporally store sub-sequences of reads. As the letters in the sequence reads are A, T, C, G and a few other reserved letters, we can use 4 bits to present a

instead of 1 byte. By doing so, the 8-byte small char array can be packed into 4 bytes, i.e., a 32-bit integer.

The optimized data structure of a bin entry is shown in Fig. 5.3(b). With the aforementioned optimization, the size of an entry reduces by half, thus greatly improving memory efficiency. This can also improve cache performance as more entries can fit into a cache line.

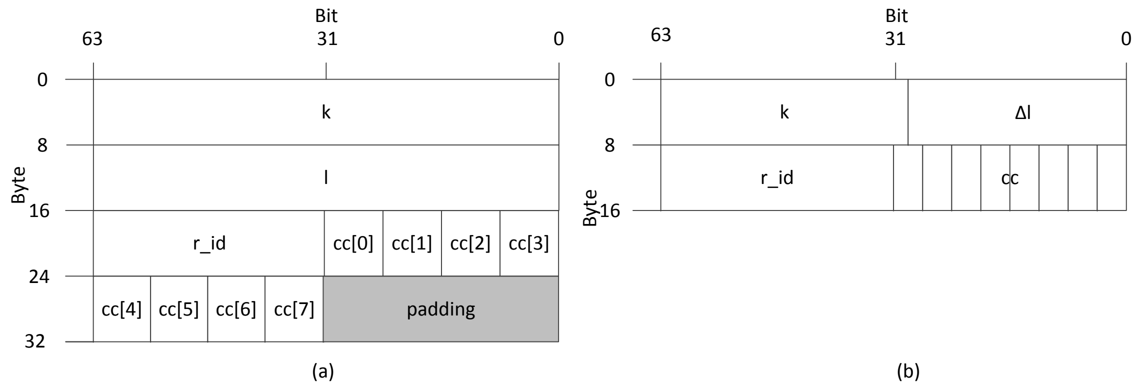


Figure 5.3: The layout of data structure of one element: preliminary (a), and optimized (b)

Bin Buffer Allocation The memory allocation of the bin buffer is complicated by the fact that the number of entries in each bin varies significantly. Dynamic memory allocation can help workaround this variance but will introduce non-trivial overhead with frequent allocation requests. On the other hand, static allocation can reduce memory allocation overhead, but can lead to memory wastage. To achieve a balance between memory utilization efficiency and runtime overhead, we adopt a hybrid approach—which statically allocates a fixed buffer for each bin and uses a large pool to be stored overflow items.

By carefully analyzing the distribution of bucket accesses to the BWT table, we find that the number of accesses of individual buckets is more evenly distributed after the first few iterations.

Since searching a read always begins with the same and k and l values, the access locations of the BWT table when searching different reads are almost the same for the first iteration. Based on the above observation, our implementation skips the first few iterations before starting the binning process and uses the average number of accesses across all buckets as the size of the preallocated buffer.

5.1.3.3 Cost-Efficient Binning

Compared to the original BWA implementation, our design involves extra computation in the binning process. It is critical to minimize the compute overhead of binning, to avoid offsetting the benefit from memory access reordering. To this end, our implementation simply right-shifts k and l to get the corresponding bin bucket numbers. However, the binning process can still introduce non-trivial overhead because it needs to be performed for every calculation of k and l. To further improve the binning efficiency, we leverage an interesting property from the BWT alignment; that is, the distance between k and l narrows fast as the search progresses. Fig. 5.4(b) shows statistics of the distance between k and l for representative input reads. As we can see, in matching reads of length 100, for most iterations (more than 80), k and l fall in the same bucket in the BWT table. This is because backward search mimics a top-down traversal over the prefix tree. In fact, this property was also used in the original BWA implementation to improve data reuse; the Occ function is optimized for the case where k and l fall in the same bucket to eliminate duplicated data loading from the BWT table. Based on this observation, our design applies binning only to k, which reduces the binning computation by half.

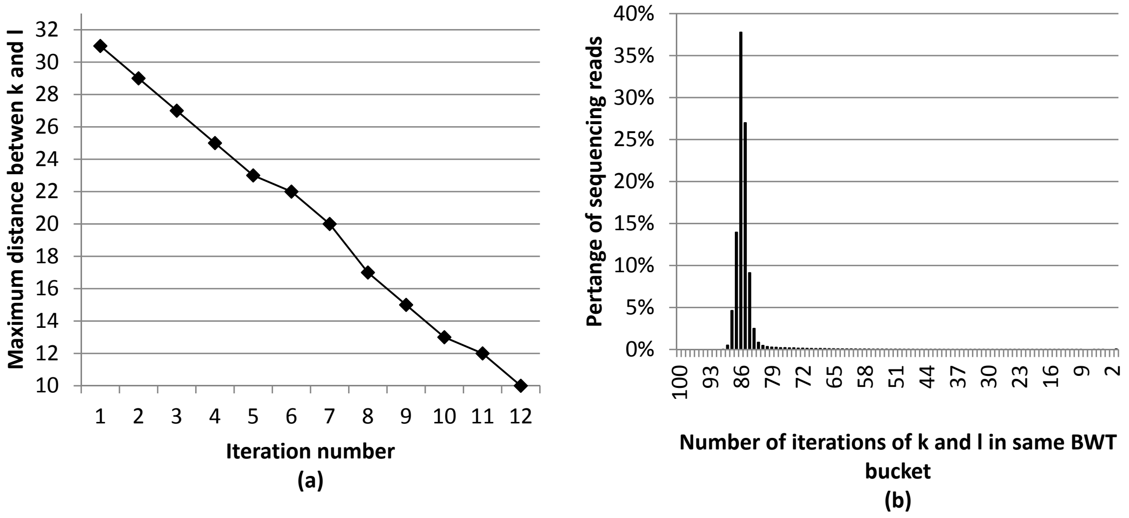


Figure 5.4: Properties of distribution of k and l (read length 100); (a) the maximum distance between k and l in a given iteration; (b) number of iterations in which k and l in same BWT bucket

Algorithm 3 Optimized Burrows-Wheeler Aligner Kernel

Input: W: sequence reads

Output: k and l pairs

```
1:  for $i = len' - 1$ to 0 do
2:      for all $bin_x$ do
3:          for all $e_j$ in $bin_x$ do
4:              if $i$ mod $cc\_size = 0$ then
5:                  preload $cc\_size$ letters from reads to $e_j.cc$
6:              end if
7:              $a \leftarrow get\_a(e_j.cc, i \bmod cc\_size)$
8:              $ok \leftarrow Occ(e_j.k - 1, a)$
9:              $ol \leftarrow Occ(e_j.l, a)$
10:             $e_j.k \leftarrow C[a] + ok + 1$
11:             $e_j.l \leftarrow C[a] + ol$
12:             if $e_j.k > e_j.l$ then
13:                 output as result
14:             else
15:                 $y \leftarrow get\_bin\_number(e_j.k)$
16:                 fill $e_j$ into $bin_y$
17:             end if
18:         end for
19:     end for
20: end for
```

Reducing Binning Overhead with Data Preload Although the basic binning algorithm can effectively improve locality of memory accesses, it does not come for free. The first two columns in Table 5.1 show the comparison between the original BWA and the preliminary binning implementations in cache misses and TLB misses for a representative input file. Surprisingly, while reordering memory access through binning can effective reduce the number of cache misses, it introduces more TLB misses.

Table 5.1: Performance numbers of original backward search, preliminary binning algorithm and optimized binning algorithm with a single thread on Intel Desktop CPU and batch size 2^{24}.

	LLC Misses (milli)	TLB misses (milli)	Execution Time (sec)
Original	70	27	3.60
Preliminary binning	56	59	3.28
Binning with preload	53	23	2.60

With careful profiling, we find that the extra TLB misses are caused by indirect references on the sequence reads; when backward search needs to fetch the next character from a read, it uses the read id r_id to locate the corresponding buffer storing the read sequence. As shown in Fig. 5.5(a), in the original algorithm, the access on a sequencing read is sequential, and thus, fetching read sequence data can benefit from the prefetching mechanism available in modern processors. However, in the preliminary binning design, due to loop interchange and memory reordering by buckets, memory accesses on the letter at the same location of different reads are random as shown in Fig. 5.5(b). As a consequence, we violate the data locality of accesses on reads in the original BWA algorithm, prefetching of sequence data from a read cannot be reused, causing more frequent accesses to read data. As a batch of reads typically occupies several hundreds of megabytes, accessing such a memory space with large strides cause an overflow of the TLB cache.

To mitigate this issue, we add a small character array in each bin entry to periodically store letters loaded from sequence reads. As the character array is embedded in every entry, it will be loaded in the cache when the corresponding k and l are processed, thus greatly reducing TLB misses in fetching the read data. As shown in the third column of Table 5.1, the enhanced binning design can significantly reduce cache misses without incurring extra TLB misses.

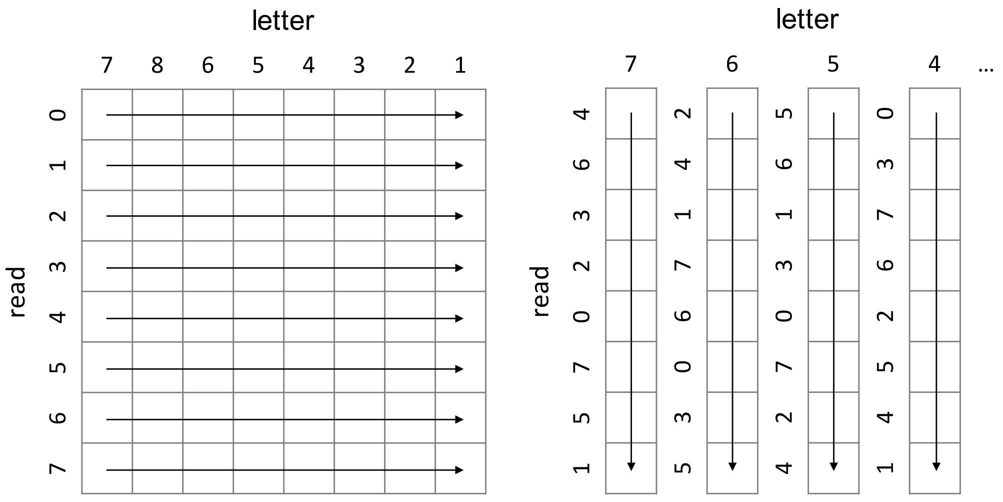


Figure 5.5: Access pattern of backward search in original (left) and binning BWA (right): each box presents a block of data; arrows show the memory access direction.

Fig. 5.6 shows the execution profile of the enhanced binning algorithm, collected using Intel VTune. Compared to Fig. 5.1, the number of non-stall cycles improves from 15% to 30%. Also, the stall cycles caused by TLB misses are greatly reduced. The differences in the execution profiles suggest that our memory-access reordering design is effective in improving memory access efficiency.

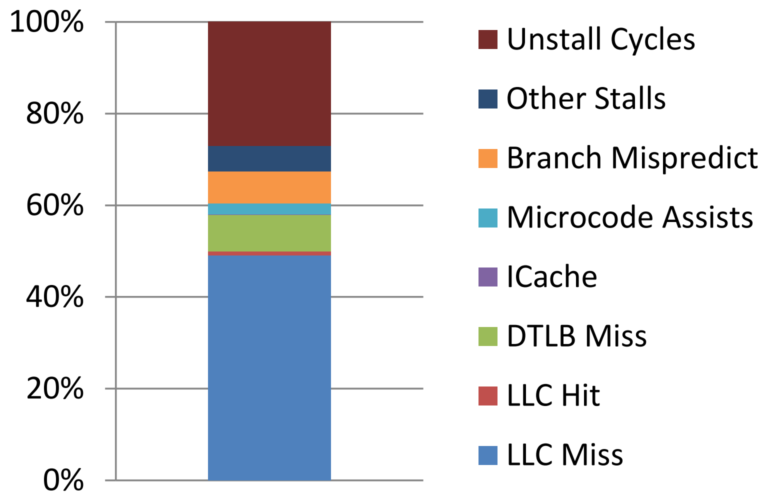


Figure 5.6: The breakdown of cycles of optimized BWA algorithm

5.1.3.4 Multithreading Optimization

Multi-core architectures add more complexity to our design. False sharing of data between threads can cause thrashing and severely impact performance. A straightforward approach to parallelize our binning design is to have each thread maintain a separate bin and work independently. The disadvantage of such a design is that the memory bandwidth of a multi-core processor cannot be efficiently utilized because there is no data sharing between threads. Therefore, in our design, all threads share the same bin structure. A design challenge then lies in how to efficiently synchronize between different threads.

To minimize synchronization overhead, our design maintains two copies of the bin structure. In the beginning, one copy of the bin structure stores all initial values and is marked as read-only. Another copy of the bin structure is marked as write-only. Extensive profiling shows that the processing time of each bin is about the same. Therefore, our design uses a static task-allocation approach, where all bins in the read-only structure are evenly distributed among all of the threads.

When processing a bin, each thread computes the k and l of each entry and places them in the corresponding bin of the write-only structure. The read-only and write-only structures are swapped in the next iteration. Such a design reduces the synchronization overhead as there is no need to coordinate accesses to the read-only structure. For the write-only structure, one index is maintained for each bin to mark the last entry in the bin. Thus, a new entry can be safely placed at the end of the bin by executing an atomic add on the index associated with the bin. Our profiling shows that such a design incurs very low overhead, partly because the contention on a particular bin is typically low.

5.1.4 Performance Evaluation

We evaluate the performance of our implementation in three aspects: impact of software configuration, the impact of micro-architecture, and scalability.

5.1.4.1 Experiment Setup

In order to evaluate the impact of variance of the micro-architecture, particularly cache size, two different Intel Sandy Bridge CPUs are used in our experiments: (1) Intel Core i5 2400 is a high-performance quad-core microprocessor with high clock frequency; (2) Intel Xeon E5-2620, a hex-core processor designed for servers, has a lower clock frequency, but is integrated with a large on-chip L3 cache. To eliminate effects of Hyper-threading (HT) on cache performance, we disable HT on the Intel Xeon E5-2620 via BIOS setting.

While our experiments focus on human genome sequencing, none of our analysis is specific to such a genome and easily carry over to other genomic datasets as well. We use sequence datasets from the GenBank database. The read queries used in this thesis are from the 1000 Genome Project. To evaluate the impact of read lengths, we choose 4 read queries with different lengths. In the remaining experiments, we use a read query with 100bp as default input.

5.1.4.2 Impact of Software Configuration

In the optimized BWA algorithm, there are three important parameters: (1) preloaded data size - the number of letters in sequence reads preloaded; (2) bin range - the range of bucket access grouped into a bin; (3) batch size - the number of sequence reads loaded into memory to be processed. To achieve the optimal configuration of these parameters, we quantify the impacts of the three parameters in this section.

5.1.4.3 Preloading Data Size

The size of preloading data determines the frequency of preloading data. A larger preloaded data size implies less indirect references, but fatter elements and larger memory footprint. In Fig. 5.7(a), we can see that when the preloaded data size increases from 4 to 32, the performance improves slightly and peaks at 16.

5.1.4.4 Bin Range

The bin range determines the granularity of memory reordering. A smaller bin range indicates that bucket accesses are more in-order. However, the overhead of binning increases as the bin range reduces. As the cache in modern CPU can be up to several megabytes, which can contain millions of elements, the elements in one bin are unlikely to be evicted before the next bucket is accessed. As we can see in Fig. 5.7(b), the overhead of binning increases with decreasing bin range causing the performance to suffer noticeable degradation when the bin range is reduced to 16.

5.1.4.5 Batch size

Batch size is a critical parameter, significantly influencing the overall performance. In Fig. 5.7(c), we observe that increasing the batch size dramatically improves cache performance, and consequently the overall application performance. But, increasing batch size barely impacts the performance of original BWA. A large batch size allows more bucket accesses, and more accesses fall into a bin, increasing the possibility that multiple bucket accesses hit the same cache line. Due to memory space limitation, we can maximally get a 2.6-fold speedup with 16 GigaBytes memory. If further increasing batch size with larger memory, we can achieve more performance gain.

5.1.4.6 Impacts of Read Length

To clarify the impact of read length, we compare the performance of the original and optimized versions of BWA with different read lengths. We notice that the difference of read lengths has

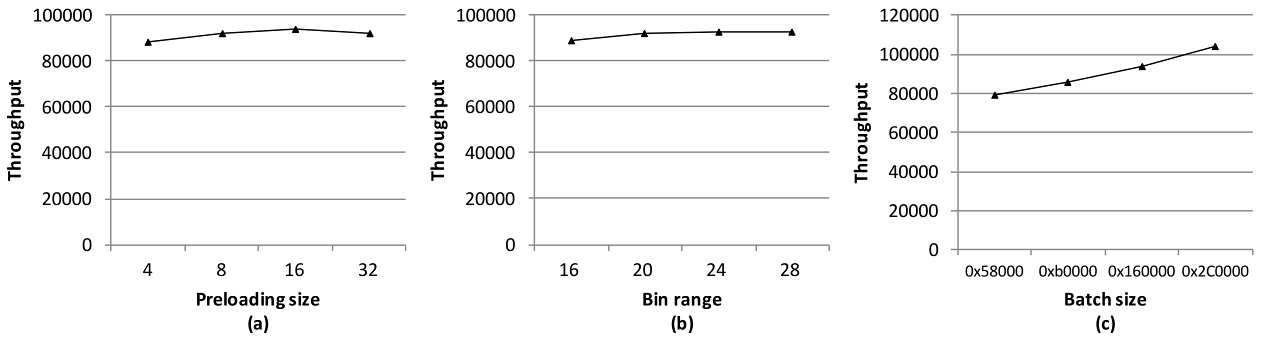


Figure 5.7: Throughput (reads per second) of optimized BWA with different software configurations: (a) preloaded data size, (b) bin range, (c) batch size. In each figure, we change a parameter while fixing the other two parameters.

little influence on the speedup of the optimized BWA algorithm as shown in Table.5.2; that is, the speedup is stable with different read lengths on both single-thread and multi-threaded tests.

Table 5.2: Performance of original and optimized BWA with different read length

	SRR003084(36bp)			SRR003092(51bp)		
	orig(s)	opt(s)	speedup	orig(s)	opt(s)	speedup
single thread	6.21	4.04	1.43	6.87	4.65	1.44
multithreads	2.4	1.87	1.28	3.79	3.14	1.21
	SRR003196 (76bp)			SRR062640(100bp)		
	orig(s)	opt(s)	speedup	orig(s)	opt(s)	speedup
single thread	12.79	8.93	1.54	20.54	14.26	1.48
multithreads	1.21	0.89	1.36	1.29	0.99	1.30

5.1.4.7 Impacts of Micro-Architectures

Micro-architectures can differ in several aspects. In this work, we mainly focus on cache size. To understand the effect of the variation of cache size, we profile both the original and optimized backward search on the two Intel CPU architecture models described in Section 7.1.5.1.

As shown in Fig. 5.8, the speedup on the Intel Xeon CPU is not better than that on Intel i5 CPU, despite the larger cache on the Intel Xeon. This is because our optimization mainly improves spatial locality of the algorithm, which is sensitive to cache line size rather than cache size itself. Furthermore, the higher single-core performance on Intel i5 benefits our optimized algorithm. If we restrict the frequency of Intel i5 to 2GHz, which is the same as the Intel Xeon, the speedup drops to close to that achieved by the Intel Xeon (Fig.5.8).

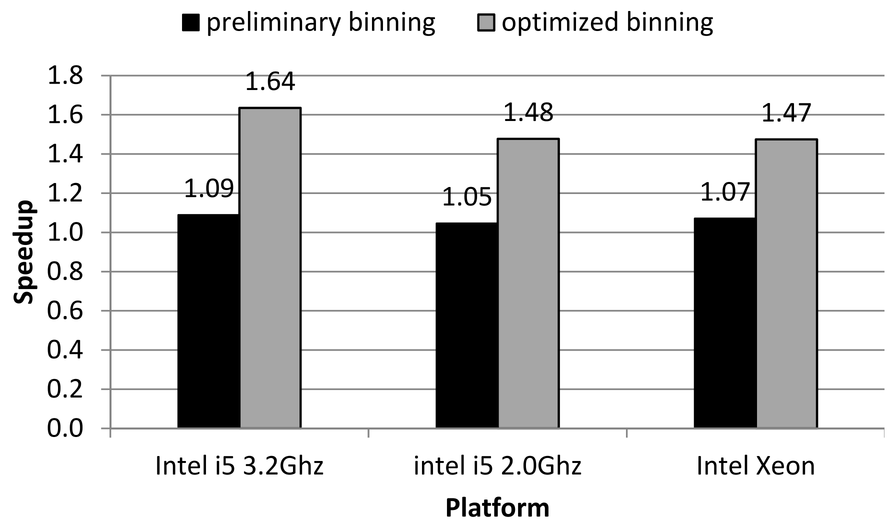


Figure 5.8: Speedup of optimized BWA over original BWA on different platforms with a single thread

5.1.4.8 Scalability

Fig. 5.9 shows the strong and weak scalability of the optimized BWA algorithm. We notice that the weak scaling of the optimized algorithm is pretty close to ideal, with an approximately 10% loss of scalability going from 1 thread to 6 threads. Strong scaling numbers show a 4.5X speedup with 6 cores (that is, a parallel efficiency of 75%).

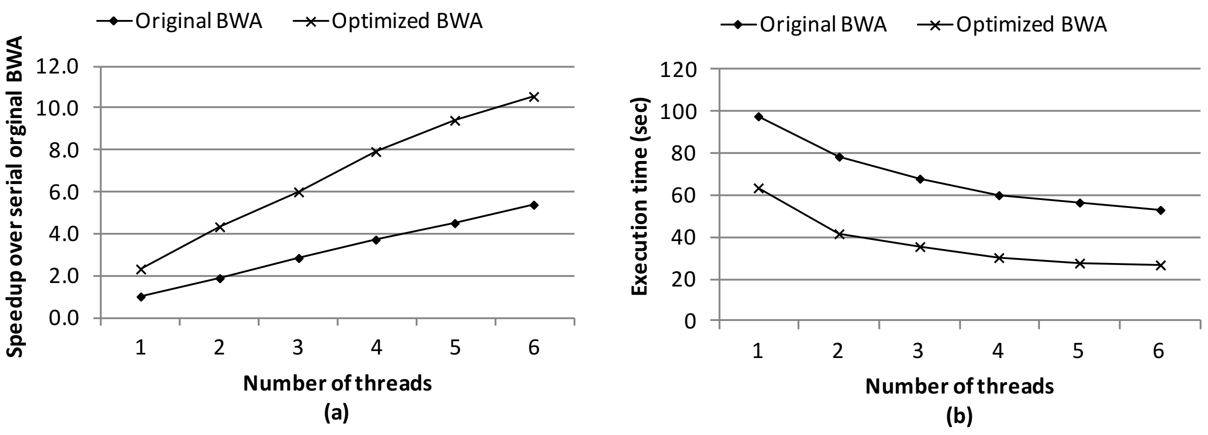


Figure 5.9: Scalability of optimized BWA: (a) and (b) are strong and weak scaling on Intel Xeon platform

We further analyze the loss of scalability for strong scaling in Table 5.3, through a more detailed architectural analysis. We notice that the multithreaded version suffers more cache and DTLB misses due to interference among threads, thus resulting in some loss of performance.

Table 5.3: Performance numbers of optimized Occ function

	Cache miss(milli)	DTLB miss(milli)	CPI
multi-threads	127	87	2.077
single thread	109	49	1.589

5.1.5 Conclusion

In this work, we first present an in-depth performance characterization with respect to the memory access pattern and cache behavior of BWT-based alignment. We then propose a well-designed optimization approach to improve data locality of backward search via binning. Our optimized BWA algorithm achieves up to a 2.6-fold speedup and a good weak scaling on multi-core architectures.

5.2 muBLASTP: Eliminating Irregularities of Protein Sequence Search on Multi-core Architectures

5.2.1 Introduction

The Basic Local Alignment Search Tool (BLAST) [119] is a fundamental algorithm in life sciences that compares a query sequence to the sequences from a database, i.e., subject sequences, to identify sequences that are most similar to the query sequence. The similarities identified by BLAST can be used to infer functional and structural relationships between the corresponding biological entities, for example.

Although optimizing BLAST is a rich area of research using multi-core CPUs [118, 158], GPUs [126, 132, 130, 131], FPGAs [123, 125], and clusters and Clouds [120, 122, 159, 160, 161, 121, 162], BLAST is still a major bottleneck in biological research. In fact, in a recent human microbiome study that consumed 180,000 core hours, BLAST consumed nearly half the time [163]. It still requires urgent attention in higher level applications.

BLAST adopts a heuristic method to identify the similarity between the query sequence and subject sequences from the database. Initially, the query sequence is decomposed into short words of the fixed length; and the words are converted into the query index, i.e., a lookup table [137] or a deterministic finite automaton (DFA) [61], to store the positions of words in the query sequence. BLAST reads the words from the subject sequence and identifies high scoring short matches, i.e., hits, from the query index. If two or more hits near enough to each other, BLAST forms the local alignment without insertions and deletions, i.e., gaps, (called two-hit ungapped extensions), and then generates the further extension based on the local alignments but allows the gaps. Although such a heuristics can efficiently eliminate unnecessary search space, it makes the execution of the program unpredictable and the memory access pattern irregular, leading to the limited scope of SIMD parallelism and the increase of the trips to the memory.

With the advent of next-generation sequencing (NGS), the exponential growth of sequence databases is arguably outstripping the ability to analyze the data. In order to deal with huge databases, a range of recent approaches of BLAST build the index based on the subject sequences instead of the input query [140, 138, 141, 139, 32]. Although these alternatives that build the database index in advance and reuse it during the search for multiple queries can improve the overall performance, there are more challenges in the parallel design on multi-core processors. In fact, most of the tools use longer, non-overlapping, or non-neighboring words to reduce the size of database index, and consequently reduce the number of hits and extensions, that also reduces irregular memory accesses. However, as reported by [142, 164, 165], they compromise the sensitivity and accuracy compared to the query indexed methods.

In this work, following the existing heuristic algorithm, we first implement a database-indexed BLAST algorithm that includes the overlapping and neighboring words, to provide exactly the same accuracy as the query-indexed BLAST i.e., NCBI-BLAST. Then, we identify that directly using the existing heuristic algorithms on the database-indexed BLAST will suffer further from irregularities: when it aligns a query to multiple subject sequences at the same time, the ungapped extension, which is the most time-consuming stage, will access the memory randomly across different subject sequences. Even worse is that the penalty from random memory access cannot be offset by the cache hierarchy even on the latest multi-core processors. To eliminate irregularities in the BLAST algorithm, which is a complex *MDMC* class problem, we propose muBLASTP, a multi-threaded and multi-node parallelism of BLAST algorithms for protein search. It includes three major optimization techniques: (1) *decoupling* the hit detection and ungapped extension to avoid the contention between the two phases, (2) *sorting* hits between decoupled phases to remove the irregular memory access and improve data locality in the ungapped extension, (3) *pre-filtering* hits not near enough ahead of sorting to reduce the overhead of hit sorting.

Experimental results show that on a modern multi-core architecture, i.e., Intel Haswell, the multi-threaded muBLASTP can achieve up to a 5.1-fold speedup over the multithreaded NCBI BLAST using 24 threads. In addition to improving performance significantly, muBLASTP produces the identical results as NCBI BLAST, which is important to the bioinformatics community.

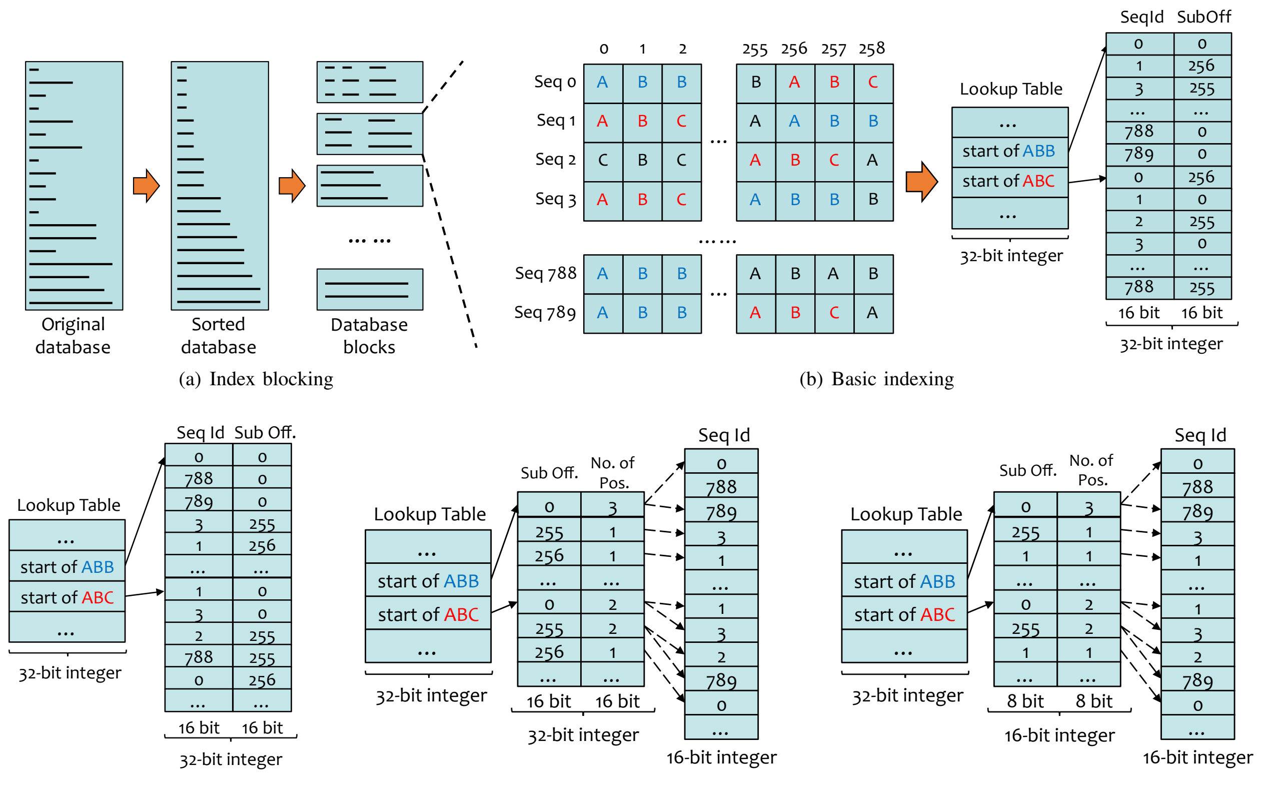
5.2.2 Database Index

One of most challenging components of muBLASTP is the design of the database index. The index should include the positions of overlapping words from all subject sequences of the database, where each position contains the sequence ID and the offset in the subject sequence, i.e., subject offset. For the protein sequence search, the BLASTP algorithm uses the small word size ($W = 3$), a large alphabet size (22 letters), and neighboring words. These factors may make the database index very large, thus we need to design our database index with the following techniques: blocking, sorting, and compression.

5.2.2.1 Index Blocking

Fig. 5.10(a) illustrates the design of index blocking. We first sort the database by the sequence length; partition the database into small blocks, where each block has the same number of letters; and then build the index for each block separately. In this way, the search algorithm can go through the index blocks one by one and merge the high-scoring results of each block in the final stage. Index blocking can enable the database index to fit into main memory, especially for large databases whose total index size can exceed the size of main memory. By shrinking the size of the index block, when the index block is small enough to fit into the CPU cache.

Another benefit of using the index blocking is to reduce the total index size. Without index blocking and assuming a total of M sequences in the database, we need $\log_2 M$ bits to store sequence IDs. After dividing the database into N blocks, each block contains $\frac{M}{N}$ sequences on average. Thus, we only need $\log_2\lceil\frac{M}{N}\rceil$ bits to store sequence IDs. For example, if there are 2^{20} sequences in a database, we need 20 bits to store the sequence IDs. With 2^8 blocks, if each block contains 2^{12} sequences, then we only need a maximum of 12 bits to store the sequence IDs. In addition, because the number of bits for storing subject offsets is determined by the longest sequences in each block, after sorting the database by the sequence length, we can use fewer bits for subject offsets in the blocks having short and medium sequences, and more bits only for the blocks having extremely long sequences. (This is one of the reasons why we sort the database by the sequence length ahead.)



(a) Index blocking

(b) Basic indexing

(c) Index sorting

(d) Index compression - merge

(e) Index compression - increment

Figure 5.10: An example of building a compressed database index. The figure shows the flow from the original database to the compressed index. (a) Index blocking phase partitions the sorted database into blocks. (b) Basic indexing phase generates basic index, which contains positions of all words in the database. (c) Index sorting sorts positions of each word by subject offsets. (d) Index compression-merge merges positions with the same subject offset. (e) Index compression-increment done on the merged positions generates increments of subject offsets and sequence IDs

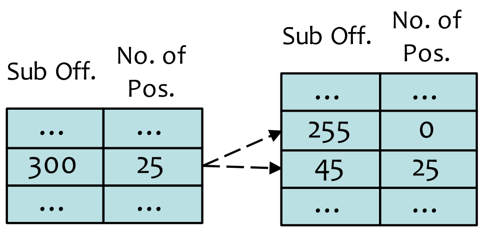
Furthermore, the index blocking allows us to parallelize the BLASTP algorithm via the mapping of a block to a thread on a modern multi-core processor. For this block-wise parallel method to achieve the ideal load balance, we partition index blocks equally to make each block have a similar number of letters, instead of an identical number of sequences. To avoid cutting a sequence in the middle, if the last sequence reaches the cap of the block size, we put it into the next block.

After the database is partitioned into blocks, each block is indexed individually. As shown in Fig. 5.10(b), the index consists of two parts: the lookup table and the position array. The lookup table contains a^w entries, where a is the alphabet size of amino acids and w is the length of the words. Each entry contains an offset to the starting position of the corresponding word. In the position array, a position of the word consists of the sequence ID and the subject offset. For protein sequence search, the BLASTP algorithm not only searches the hits of exactly matched words, but also searches the neighboring words, which are similar words. The query index used in existing BLAST tools, e.g., NCBI BLAST, includes the positions of neighboring words in the lookup table. However, for the database index in muBLASTP, if we store the positions for the neighboring words, the total size of the index becomes extraordinarily large. To address this problem, instead of storing positions of the neighboring words in the index, we put the offsets, which point to the neighboring words of every word, into the lookup table. The hit detection stage then goes through the positions of neighbors via the offsets after visiting the current word. In this way, we use additional stride memory accesses to reduce the total memory footprint for the index.

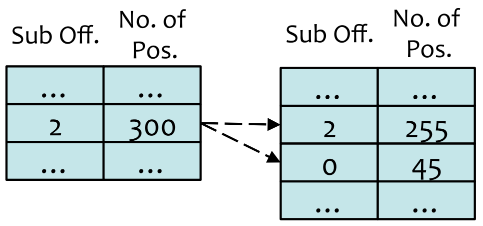
5.2.2.2 Index compression

As shown in Fig. 5.10(b), a specific subject offset for a word may be repeated in multiple sequences. For example, the word "ABC" appears in the position 0 of sequences 1 and 3. In light of this repetition, it is possible to compress the index by optimizing the storage of subject offsets. Next, we sort the position array by the subject offset to group the same subject offsets together, as shown in Fig. 5.10(c). After that, we reduce the index size via merging the repeated subject offsets: for each word, we store the subject offset and the number of positions once and store the corresponding sequence IDs sequentially, as shown in Fig. 5.10(d). After the index merging, we only need a small array for the sorted subject offsets. Furthermore, because the index is sorted by subject offsets, instead of storing the absolute value of subject offsets, we store the incremental subject offsets, as noted in Fig. 5.10(e), and only use eight (8) bits for the incremental subject offsets. Because the number of positions for a specific subject offset in a block is generally less than 256, we can also use eight (8) bits for the number of positions. Thus, in total, we only need a 16-bit integer to store a subject offset and its number of positions.

However, this compressed method presents a challenge. When we use eight (8) bits each for the incremental subject offset and the number of repeated positions, there still exist a few cases that the increment subject offsets or the number of repeated positions can be larger than 255. When such situations are encountered, we split a position entry into multiple entries to make the value less than 255. For example, as shown in Fig. 5.11(a), if the increment subject offset is 300 with 25 positions, then we split the subject offset into two entries, where the first entry has the incremental subject offset 255 and the number of repeated position 0, and the second entry has the incremental subject offset 45 for the 25 positions. Similarly, as shown in Fig. 5.11(b), for 300 repeated number of positions, the subject offset is split into two entries, where the first entry has the incremental subject offset 2 for 255 positions, but the second has the incremental subject offset 0 for an additional 45 positions.



(a) Number of positions overflow



(b) Increment subject offset overflow

Figure 5.11: An example of resolving overflows in the compressed index. (a) Resolving the overflow in the number of positions. (b) Resolving in the incremental subject offsets

5.2.3 Performance Analysis of BLAST Algorithm with Database Index

The existing BLAST algorithm executes the first three stages interactively: once a hit is detected, the algorithm immediately triggers the ungapped extension if the distance is smaller than the threshold, and then issues the gapped extension. For the query-indexed BLAST algorithm, since the subject sequences are aligned one by one to the query, only a lasthit array is needed for the query sequence. Moreover, the protein sequences are generally short, no more than 2K characters. Therefore, we still can achieve good cache performance for the lasthit array method, the query sequence, and the subject sequence, even though the memory access pattern on those data is totally random.

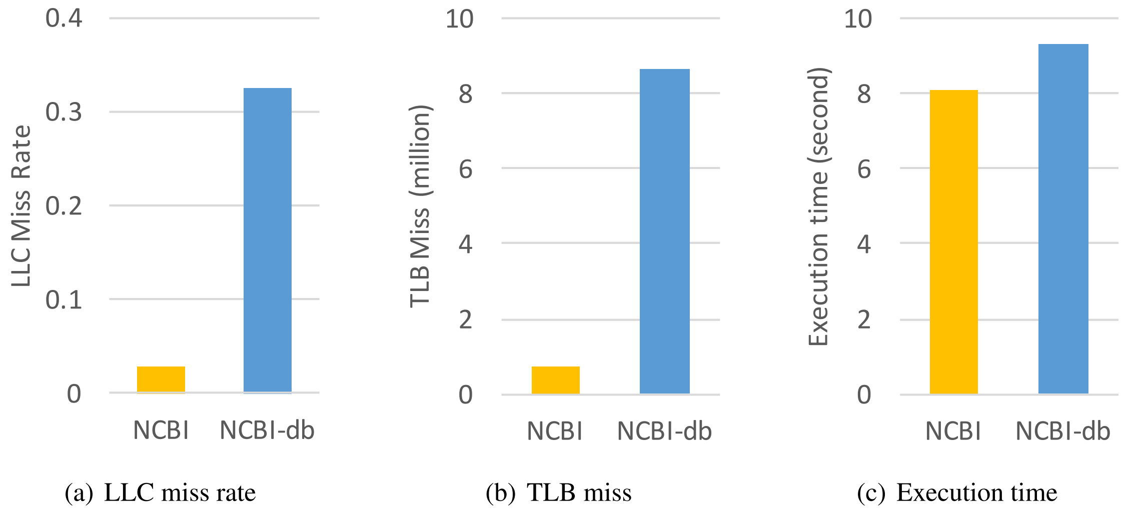


Figure 5.12: Profiling numbers and execution time of the query indexed NCBI-BLAST (NCBI) and the database-indexed NCBI-BLAST (NCBI-db) when search a query of length 512 on the *env_nr* database.

However, irregular memory access patterns in the database-indexed search can lead to a severe locality issue. With an in-depth performance characterization, we identify the database-indexed BLAST algorithm is a *MDMC* problem, which has complex irregularities across multiple functions and data structures. Each word in the database index can include positions from all subject sequences, the algorithm has to keep many lasthit arrays, one for a subject sequence. When the algorithm scans the query sequence successively, a new hit may be located on any lasthit array, and the ungapped extension may be triggered for any subject sequence. As a consequence, the execution path of the program will jump back and forth across different subject sequences, leading to the cached lasthit arrays and subject sequences flushed out from the cache before reuse. Fig. 5.12(a) and 5.12(b) compares the LLC (Last-Level Cache) and TLB (Translation Lookaside Buffer) miss rate, respectively, between NCBI-BLAST with the query index (NCBI) and NCBI-BLAST with the database index (NCBI-db), when searching a real protein sequence of length 512 on the *env_nr* database. Note that NCBI-db (described in Section 5.2.2) uses the database index with overlapping and neighboring words to provide the same results as NCBI-BLAST with the query index. We can see the database index method has much higher LLC and TLB miss rate. As a result, the overall performance with the database index is much worse than that with the query index, as shown in Fig. 5.12(c).

5.2.4 Optimized BLASTP Algorithm with Database Index

5.2.4.1 Decoupling First Three Stages

As discussed in Section 5.2.3, with the database index, the BLAST algorithm have to operate on multiple lasthit arrays simultaneously, because a word can induce multiple hits at different subject sequences. The interleaving execution of hit detection, ungapped extension, and gapped extension will lead to random memory accesses across lasthit arrays and subject sequences. In order to avoid the data swapped in and out the cache without being fully reused, we decouple these three stages. That means after loading an index block, the hit detection will find all hits, and store hits in a temporal buffer. Because the hits for a subject sequence may be distributed randomly in this buffer, we add an additional stage, i.e., hit reordering, before the ungapped extension and the following gapped extension.

A new data structure is introduced to record the hits for fast hit reordering. A hit should contain sequence ID, diagonal ID, subject position (subject offset) and query position (query offset). For the sequence ID and diagonal ID, we pack them into a 32-bit integer as the key, in which the sequence ID uses the higher bits and the diagonal ID uses the lower bits. With this packed key, we just need to sort hits by the key once, and then hits are sorted in the order for both sequence IDs and diagonal IDs. For the subject offset and query offset, since a subject offset or query offset can be calculated with each other with a given diagonal ID as $diagonal\_id - query\_offset$ or $diagonal\_id - subject\_offset$, we just need to keep one of these two offsets, e.g., the query offset, and calculate the other in the ungapped extension (Fig. 5.13). We realize that today's protein databases may contain very long sequences ($\sim 40k$ characters). We don't build the index for such extreme cases. Instead, we use a method proposed recently in [162] to divide the extremely long sequence into multiple short sequences with the overlapped boundaries and use an assembly stage to extend the ungapped extension and gapped extension after finishing the extension inside each short sequence.

5.2.4.2 Hit Reordering with Radix Sort

As shown in Fig. 5.13, the hit detection algorithm will put hits for different subject sequences in successive memory locations in the temporal hit buffer. For the word *ABC*, the hit detection will put the hits (0,0) and (0,4) for the subject sequence 0 in the hit buffer, and then put the hits (0,0), (0,4), and (0,6) for the subject sequence 1 into the following memory locations of the hit buffer. Because the ungapped extension can only operate on hits in the same diagonal of a subject sequence, we have to reorder hits.

There are many sorting algorithms, such as radix sort, merge sort, bitonic sort, and quicksort. Based the analysis in Section 4.3.2.2, radix sort is the best option for the hit reordering due to the following reasons: First, thanks to the index blocking technique, each block has merely hundreds of kilobytes to several megabytes of hits, which can easily fit into the LLC. Therefore, the radix sort does not have high memory bandwidth issue in our case. Second, because we sort the subject sequences when building the database index, each block has the similar length of keys, which is friendly to the radix sort. Third, in the hit detection, the query sequence is scanned from the beginning to the end, and the hits are already in the order of query offsets. Because we need to keep such an order in the key-value sort, the radix sort is a better choice considering the merge sort may lose a little bit performance to achieve the stable sort. There are two ways to implement the radix sort, one is beginning at the least significant digit, called LSD radix sort; and the other is beginning at the most significant digit, called MSD radix sort. Although MSD radix sort has less computational complexity because it may not need to examine all keys, MSD radix sort is too slow for small datasets, e.g., hundred kilobytes in our case. Therefore, we choose LSD radix sort to reorder the hits after the hit detection stage.

Algorithm 4 illustrates the BLAST algorithm on the database index. To achieve better data locality, the algorithm loads index blocks one by one (line 3), and go through all input queries for an index block in the inner loop (line 4). For each query in the inner loop, the hit detection function *hitDetect()* scans the current query, and find hits for all subject sequences in the index block (line

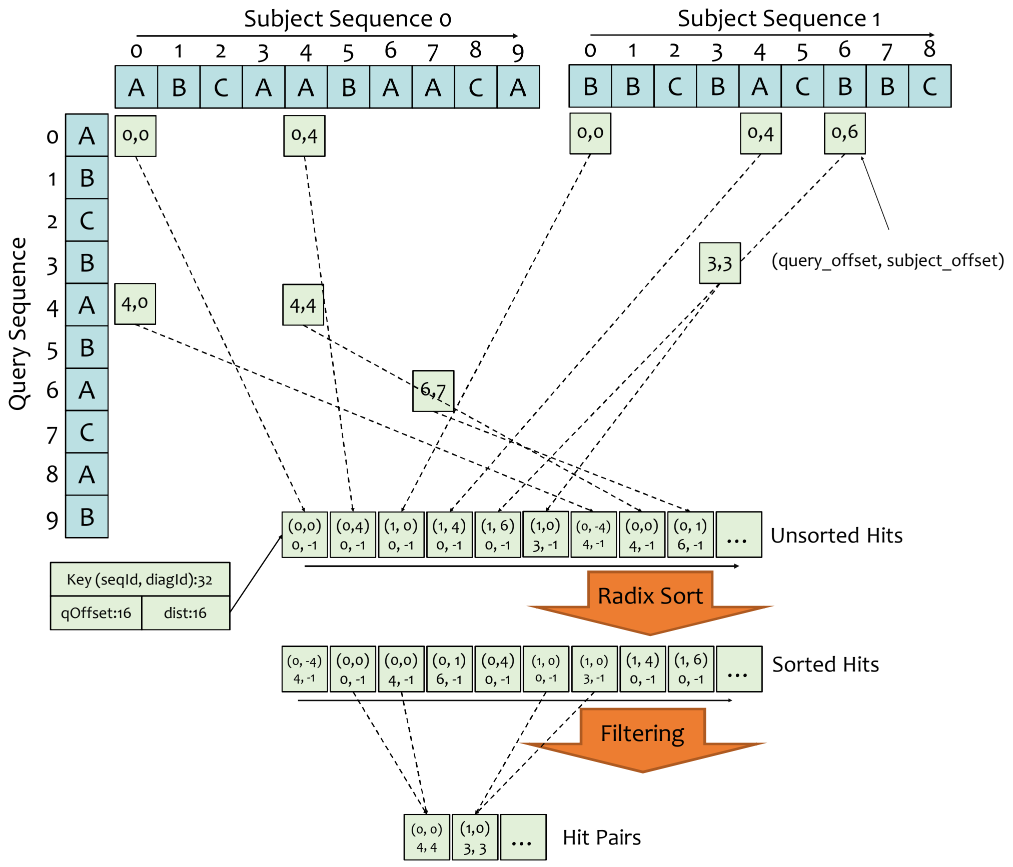


Figure 5.13: Hit-pair search with hit reordering

5). All hits are sorted with the key, including the sequence ID and diagonal ID, by LSD radix sort (line 6). After the hits are sorted, they are passed to the filtering stage (line 9) that picks up the hit pairs near enough along the same diagonal (line 11) and stores them into the internal buffer *HitPairs*. In the ungapped extension, the *for* loop starting from line 20, the hit pairs are extended one by one in the order of subject sequence IDs and diagonal IDs. Thus, this method can reuse the subject sequence during the ungapped extension, while the previous methods cannot, because they issue the ungapped extension immediately within the hit detection and have to jump from a subject sequence to another. Before doing the ungapped extension, the algorithm will also check if the current hit pair is covered by the extension of previous hit pair (line 21). If it is, the algorithm will skip this hit pair.

5.2.4.3 Hit Pre-filtering

Although we have applied the highly efficient radix sort in the hit reordering, the overhead to sort millions of hits per block are not negligible. We introduce a pre-filtering stage before the hit reordering to kick out hits that cannot trigger the ungapped extension. We use the similar idea of the lasthit array: an array is created for a subject sequence to record the current hit in each diagonal; instead of triggering the ungapped extension immediately when a hit pair is detected, the hit pair is put into the hit buffer. Because we only use these lasthit arrays in the hit detection in which we don't access any subject sequence, we do not have the cache swapping issue in the lasthit array method. Fig. 5.14 illustrates the optimized BLAST algorithm with the hit pre-filtering.

Fig. 5.15 illustrates the number of hits that will be sorted in the hit reordering stage with and without the hit pre-filtering. When searching randomly picked 20 input queries on the real protein database *uniprot_sprot*, there are only $3 \sim 4$ percent of hits left after the pre-filtering. As a result, the overhead in radix sort can be reduced dramatically.

Algorithm 5 shows the optimized BLAST algorithm with pre-filtering. In the inner loop, the two-dimensional array *lasthitArr* is used to record the lasthits in every diagonal of subject sequences. When a hit is detected (line 6), the algorithm calculates its diagonal ID and sequence ID (line 7), and accesses the lasthit in this diagonal (line 9). If the distance is smaller than the threshold, this hit pair is stored in the hit pair buffer (line 12). The corresponding position of the lasthit array is

Algorithm 4 database-indexed BLASTP Algorithms with Hit Reordering

```
1:  **Input:** $DI$: database index, $Q$: query sequences
2:  **Output:** $U$: high-scoring ungapped alignments
3:  **for all** database index block $dIdxBlk_i$ in $DI$ **do**
4:      **for all** sequence $q_i$ in $Q$ **do**
5:          $hits \leftarrow hitDetect(dIdxBlk_i, q_i)$
6:          $sortedHits \leftarrow radixSort(hits)$
7:          $reachedPos \leftarrow -1$
8:          $reachedKey \leftarrow -1$
9:          **for all** $hit_i$ in $sortedHits$ **do**
10:             $distance \leftarrow hit_i.qOffset - reachedPos$
11:             **if** $reachedKey == hit_i.key$ and $distance < threshold$ **then**
12:                 $hit_i.dist = distance$
13:                 $HitPairs \leftarrow HitPairs + hit_i$
14:             **end if**
15:             $reachedPos \leftarrow hit_i.qOffset$
16:             $reachedKey \leftarrow hit_i.key$
17:         **end for**
18:         $extReached \leftarrow -1$
19:         $reachedKey \leftarrow -1$
20:         **for all** $hit_i$ in $HitPairs$ **do**
21:             **if** $reachedKey == hit_i.key$ and $extReached > hit_i.qOffset$ **then**
22:                 skip this hit
23:             **else**
24:                 $ext \leftarrow ungappedExt(hit_i, lasthit, S, q_i)$
25:                 **if** $ext.score > thresholdT$ **then**
26:                     $U \leftarrow U + ext$
27:                     $extReached \leftarrow ext.end$
28:                 **else**
29:                     $extReached \leftarrow hit_i.qOffset$
30:                 **end if**
31:             **end if**
32:             $reachedKey \leftarrow hit_i.key$
33:         **end for**
34:     **end for**
35: **end for**
```

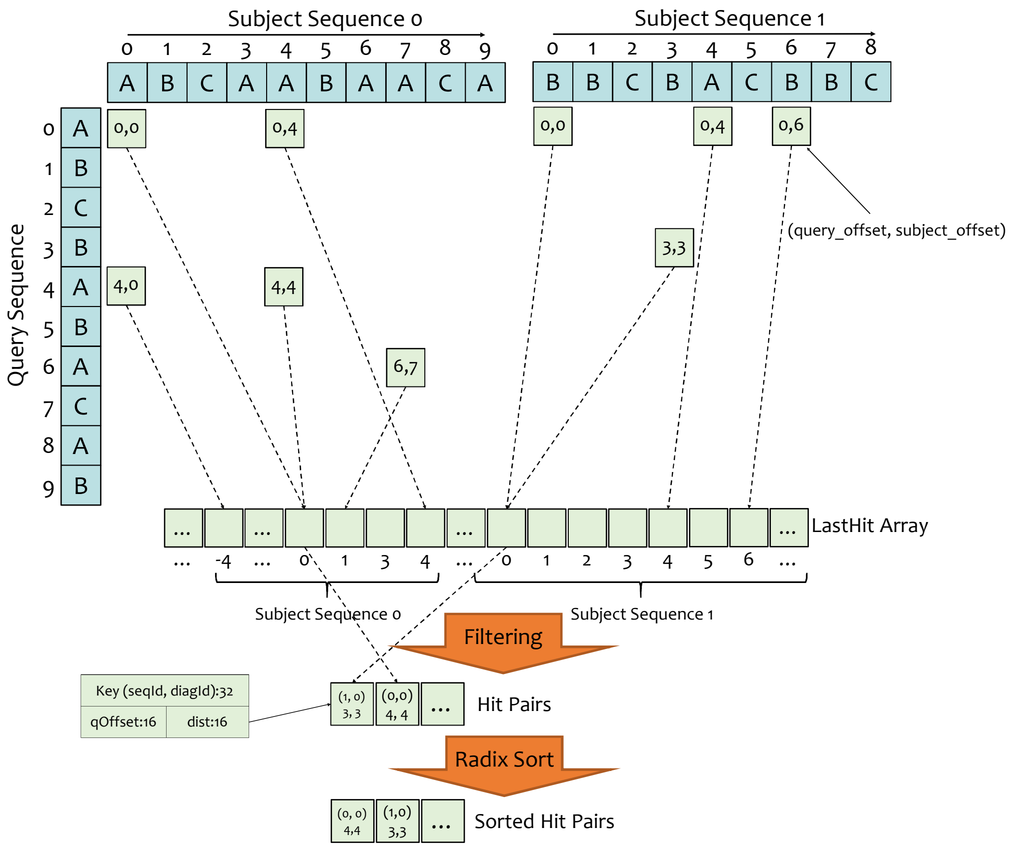


Figure 5.14: Hit reordering with pre-filtering

also updated to the current hit (line 14). After the pre-filtering, all hit pairs will be sorted using the radix sort (line 16). Note that Algorithm 4 also has a filtering stage after the hit reordering (post-filtering) to kick out the hit pairs that cannot trigger the ungapped extension. We apply the pre-filtering in our evaluations to reduce the overhead of hit reordering.

5.2.4.4 Optimizations in multithreading

In the BLAST algorithm, the query sequence is aligned to each subject sequence in the database independently and iteratively. Thus, we can parallelize the BLAST algorithm with OpenMP multithreading on the multi-core processors in a compute node, e.g., our pair of 12-core Intel Haswell

Algorithm 5 database-indexed BLASTP Algorithms with Pre-filtering and Hit Reordering

```
1: **Input:** $DI$: database index, $Q$: query sequences
2: **Output:** $U$: high-scoring ungapped alignments
3: for all database index block $dIdxBlk_i$ in $DI$ do
4:     for all sequence $q_i$ in $Q$ do
5:         $hits \leftarrow hitDetect(dIdxBlk_i, q_i)$
6:         for all $hit_j$ in $hits$ do
7:             $diagId \leftarrow hit.subOff - hit.queryOff$
8:             $seqId \leftarrow hit.seqId$
9:             $lasthit \leftarrow lasthitArr[seqId][diagId]$
10:            $distance \leftarrow hit - lasthit$
11:            if $distance < thresholdA$ then
12:                $hitPairs \leftarrow createHitPairs(hit, lasthit)$
13:            end if
14:            $lasthitArr[seqId][diagId] \leftarrow hit.subOff$
15:        end for
16:        $sortedHitPairs \leftarrow hitSort(hitPairs)$
17:        $extReached \leftarrow -1$
18:        for all $hitPair_i$ in $sortedHitPairs$ do
19:            if $hitPair_i.end.subOff > extReached$ then
20:                $ext \leftarrow ungappedExt(hitPair_i, S, q_i)$
21:                if $ext.score > thresholdT$ then
22:                    $U \leftarrow U + ext$
23:                    $extReached \leftarrow ext.end.subOff$
24:                else
25:                    $extReached \leftarrow hitPair_i.end.subOff$
26:                end if
27:            end if
28:        end for
29:    end for
30: end for
```

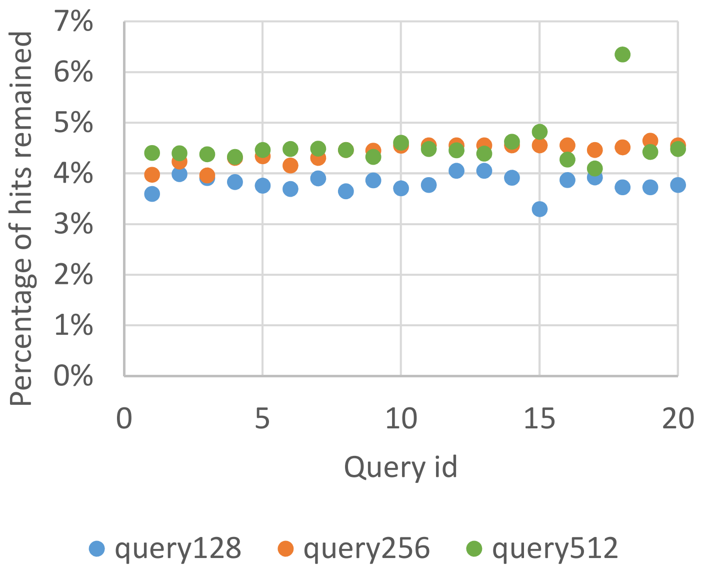


Figure 5.15: Percentage of hits remained after pre-filtering. For different query length — 128, 256 and 512, we select 20 queries from the *uniprot_sprot* database

CPUs or 24 cores in total. However, achieving robust scalability on such multi-core processors is non-trivial, particularly for a data-/memory-intensive program like BLAST, which also introduces irregular memory access patterns as well as irregular control flows. At a high level, two major challenges exist for parallelizing BLAST within a compute node: (1) cache and memory contention between threads on different cores and (2) load balancing of these threads.

Because the alignment on each query is independent, a straightforward approach of parallelization is mapping the alignment of each query to a thread. However, this approach results in different threads potentially accessing different index blocks at the same time. In light of the limited cache size, this approach results in severe cache contention between threads. To mitigate this cache contention and maximize cache-sharing across threads, we exchange execution order, as shown in Algorithm 6. That is, the first two stages, i.e., hit detection and ungapped extension, which share the same database index, access the same database block for all batch query sequences (from

Line 5 to 10). So, we apply the OpenMP pragma on the inner loop to make different threads process *different* query sequences but on the *same* index block. Then, threads on different cores may share the database index that is loaded into memory and even cache. The aligned results for each index block are then merged together for the final alignment with traceback, as shown on Line 9.

Algorithm 6 Optimized multithreaded muBLASTP

```
1:  **Input:** $DI$: database index, $Q$: query sequences
2:  **Output:** $G$: top-scoring gapped alignments with traceback
3:  **for all** database index block $dIdxBlk_i$ in $DI$ **do**
4:      #pragma omp parallel for schedule(dynamic)
5:      **for all** $q_i$ in $Q$ **do**
6:          $hits \leftarrow hitDetect(dIdxBlk_i, q_i)$
7:          $sortedHitPairs \leftarrow hitFilterAndSort(hits)$
8:          $ungapExts \leftarrow ungapExt(sortedHitPairs)$;
9:          $gapExts[i] \leftarrow gapExts + gappedExt(ungappedExts)$;
10:     **end for**
11: **end for**
12: #pragma omp parallel for schedule(dynamic)
13: **for all** $q_i$ in $Q$ **do**
14:     $sortedGapExts[i] \leftarrow SortGapExt(gapExts[i])$
15:     $G \leftarrow gappedExtWithTraceback(sortedGapExts[i])$
16: **end for**
```

For better load balancing, and in turn, better performance, we leverage the fact that we already have a sorted database with respect to sequence lengths. We then partition this database into blocks of equal size and leverage OpenMP dynamic scheduling.

5.2.5 Performance Evaluation

5.2.5.1 Experimental Setup

Platforms: We evaluate our optimized BLASTP algorithm with the database index on modern multi-core CPUs. For the single-node evaluations, the compute node consists of two Intel Haswell Xeon CPUs (E5-2680v3), each of which has 12 cores, 30MB shared L3 cache, and dedicated 32KB L1 cache and 256KB L2 cache on each core. For the multi-node evaluations, we use 128 nodes of the Stampede supercomputer, that was 10th on the Top 500 list of November 2015. Each node of our multi-node evaluations has two Intel Sandy Bridge Xeon CPUs (E5-2680), where each CPU has 8 cores, 20MB shared L3 cache, and dedicated 32KB L1 cache and 256KB L2 cache on each core. All programs are compiled by the Intel C/C++ compiler 15.3 with the compiler flags `-O3 -fopenmp`. All MPI programs are compiled using Intel C/C++ compiler 15.3 and MVAPICH 2.2 library.

Databases: We choose two typical protein NCBI databases from GenBank [62]. The first is the *uniprot_sprot* database, including approximately 300,000 sequences with a total size of 250 MB. The median length and average length of sequences are 292 and 355 bases (or letters), respectively. The second is the *env_nr* database, including approximately 6,000,000 sequences with the total size at 1.7 GB. The median length and average length are 177 and 197 bases (or letters), respectively.

Fig. 5.16 shows the distribution of sequence lengths for the *uniprot_sprot* and *env_nr* databases. We observe that the sizes of most sequences from the two databases are in the range from 60 to

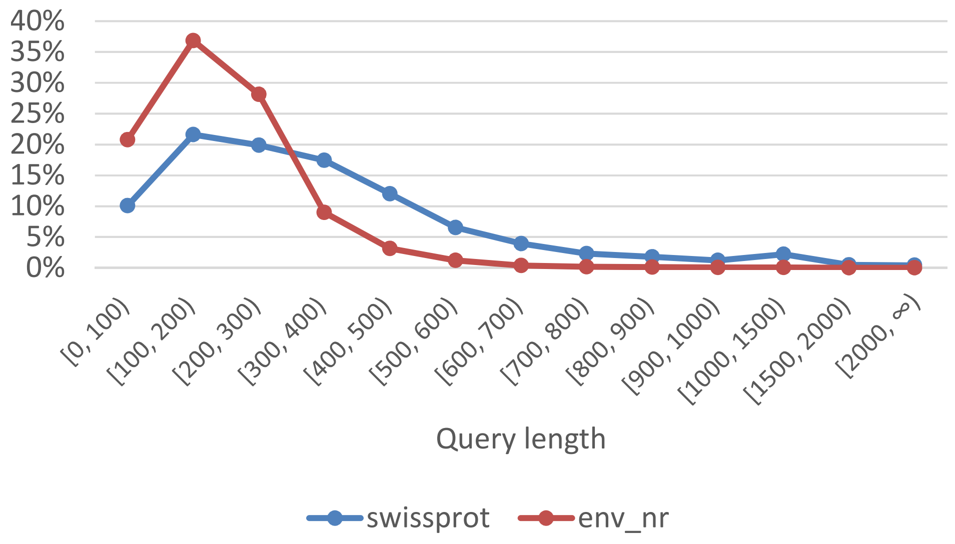
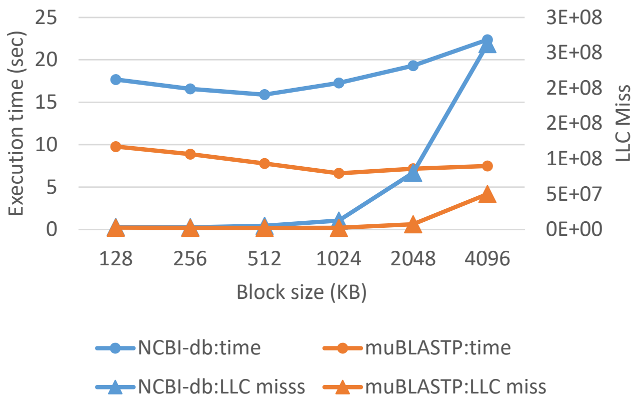


Figure 5.16: Sequence length distributions of *uniprot_sprot* and *env_nr* databases.

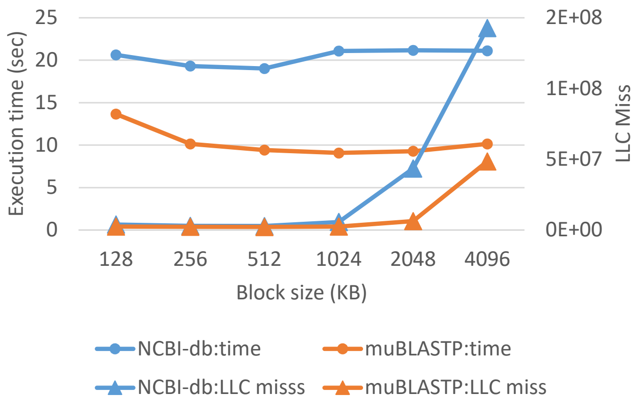
1000 bases and there are few sequences longer than 1000 bases. Similar observations were also reported in previous studies for the protein sequence [166, 167, 142].

Queries: According to the length distribution shown in Fig. 5.16, we randomly pick three sets of queries from target databases with different lengths: 128, 256 and 512. To mimic the real world workload, we prepare the fourth set of queries with the mixed length. This set follows the distribution of sequence length of the target databases. Each set has two batch size: the batch of 128 queries and batch of 1024 queries.

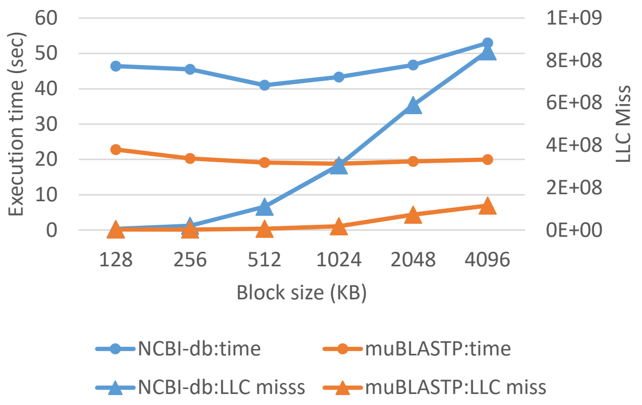
Methods: We evaluate three methods on the single node: the latest NCBI-BLAST (version 2.30) that uses the query index, labeled as **NCBI**; the NCBI-BLAST algorithm with the database index as shown in Section 5.2.2, labeled as **NCBI-db**; and our optimized BLAST, labeled as **muBLASTP**. Note that because there isn't an open sourced BLAST tool using the database index that can get



(a) 128 length



(b) 256 length



(c) 512 length

Figure 5.17: Performance numbers of multi-threaded NCBI-db and muBLASTP on the *uniprot_sprot* database. The batch has 128 queries. The lengths of queries are 128, 256 and 512.

exactly same results of NCBI-BLAST, we implement the second method with our own database index structure but follow the NCBI-BLAST algorithm. On multiple nodes, we compare the MPI version of **muBLASTP** with **mpiBLAST** (version 1.6.0). All performance results in experiments refer to the end to end run times from submitting queries to getting the final results. The database sorting time and index build time is not included since the index only needs to be built once offline for a given database.

5.2.5.2 Performance with Different Block Sizes

To find the best index block size, we evaluate the performance of database indexed methods, i.e., NCBI-db and muBLASTP, with various block sizes for the *uniprot_sprot* database. Fig. 5.17(a) shows the variable performance. We set the batch size to 128, having 128 input queries, change the length of query: 128, 256 and 512, and also change the index block size from 128 KB to 4 MB, corresponding to 32K to 1M positions in each index block. The figures show obvious improvements in execution time of muBLASTP in all cases, the reduced LLC miss rate give a hint of the reason where the performance comes from: much better cache utilization.

With increasing the index block size, we can also see both execution time and the LLC miss rate decrease initially but increase rapidly after the index block size reaches 512 KB. The reason for the decreasing LLC miss rate at the beginning is because of the increasing efficiency of cache usage. For example, if the index block size is 512 KB, there are nearly 128K positions (i.e., each position is stored in a 32-bit Integer). Because a word has 3 amino acids codes (24 codes), there are totally 24^3 (i.e., 13824) possible words. On the average, there are 9 to 10 positions per word (i.e., $128 * 1024/13824$), occupying 36 to 40 bytes. Thus the cache line can be fully utilized with the block size of 512 KB. As a result, if the index block size is smaller than 512 KB, the cache line is underutilized.

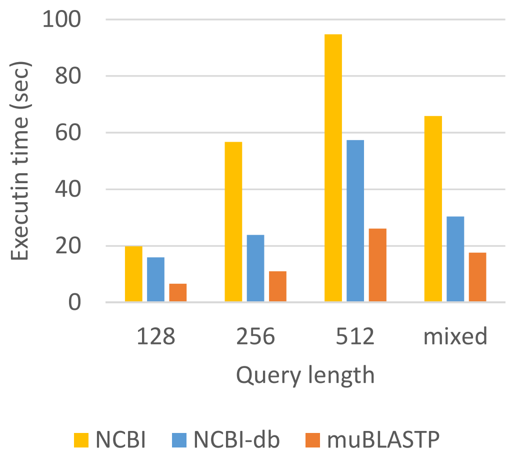
After the block size reaches 1 MB, the index block and lasthit array cannot fit into the LLC cache. Therefore, the LLC miss rate begins to grow. Because the length of the lasthit array is twice of a number of positions, the lasthit array in each thread can roughly occupy 2 MB of the memory, and totally 24 MB for 12 threads. However, there is a 30 MB LLC in our test platform. If the block size is larger than 1 MB, it is possible that the memory access on the lasthit array lead to severe LLC misses because the lasthit array is out of the cache. Without the optimizations of eliminating irregularities, the performance of NCBI-db is reduced much more rapidly than that of muBLASTP.

Based on the discussion above, to fully utilizing hardware prefetcher, we need to select a proper block size to make the index block and the lasthit array can just fit into the LLC cache. Since the lasthit array size for t threads is $t * b * 2$, where b is the block size, for the given LLC cache size L, we can estimate the optimal block size b by $b = L/(t * 2 + 1)$.

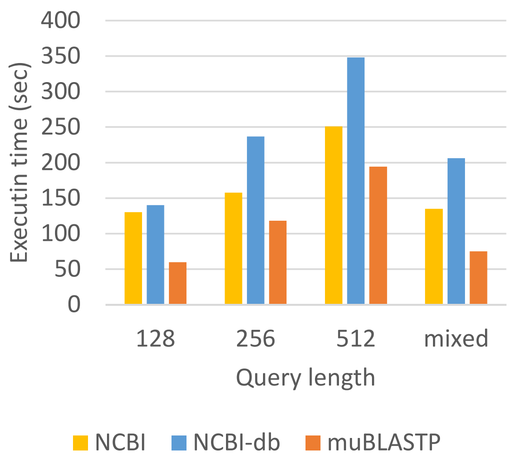
5.2.5.3 Comparison with Multi-threaded NCBI-BLAST

Fig. 6.11 illustrates the performance comparisons of muBLASTP with NCBI and NCBI-db on two types of protein databases. Fig. 5.18(a) and Fig. 5.18(b) show that for the batch of 128 queries, muBLASTP can achieve up to 5.1-fold and 3.3-fold speedups over NCBI on *uniprot_sprot* and *env_nr* databases, respectively. For the batch of 1024 queries, the speedups are 2.5-fold and 2.1-

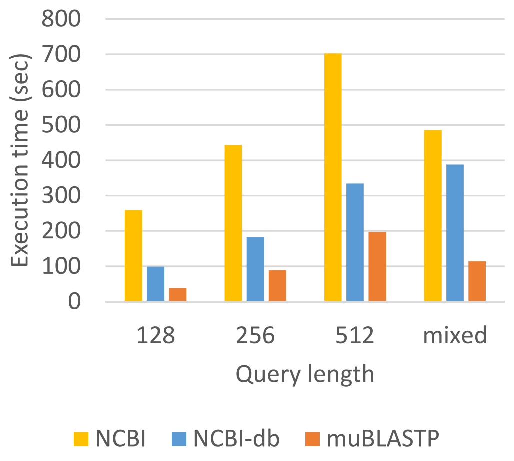
fold, as shown in Fig. 5.18(c) and Fig. 5.18(d). Compared to NCBI-db, muBLASTP can deliver up to 3.3-fold and 3.9 fold speedups on *uniprot_sprot* and *env_nr* databases for the batch of 128 queries, and up to 2.4-fold and 2.7-fold speedups for the batch of 1024 queries.



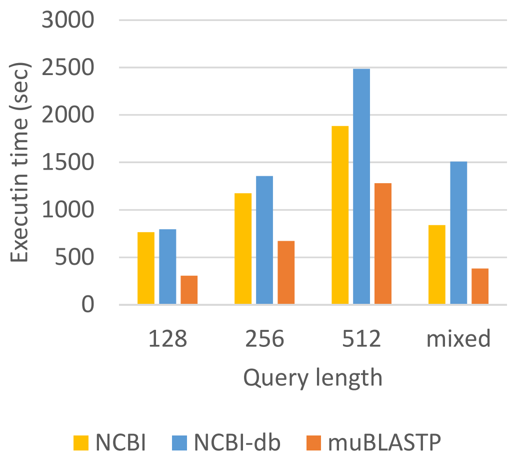
(a) *uniprot_sprot* and batch 128



(b) *env_nr* and batch 128



(c) *uniprot_sprot* and batch 1024



(d) *env_nr* and batch 1024

Figure 5.18: Performance comparisons of NCBI, NCBI-db and muBLASTP with batch of 128 and 1024 queries on *uniprot_sprot* and *env_nr* databases.

The figure also illustrates that for the large database, i.e., *env_nr*, database-indexed NCBI-BLAST (NCBI-db) cannot gain the better performance than the query-indexed NCBI-BLAST (NCBI). This is because of more irregularities in the BLAST algorithm with the larger database index. Our optimizations in muBLASTP are designed to resolve these issues and can deliver better performance than NCBI-BLAST no matter which indexing methods are used.

5.2.6 Conclusion

In this chapter, we present muBLASTP, a database-indexed BLASTP that delivers identical hits returned to NCBI BLAST for protein sequence search. With our new index structure for protein databases and associated optimizations in muBLASTP, we deliver a re-factored BLASTP algorithm for modern multi-core processors that achieves much higher throughput with acceptable memory usage for the database index. On a modern compute node with a total of 24 Intel Haswell CPU cores, the multithreaded muBLASTP achieves up to a 5.7-fold speedup for alignment stages, and up to a 4.56-fold end-to-end speedup over multithreaded NCBI BLAST. muBLASTP also can achieve significant speedups on an older generation platform with dual 6 cores Intel Nehalem CPU, where muBLASTP delivers up to an 8.59-fold speedup for alignment stages, and up to a 3.85-fold end-to-end speedup over multithreaded NCBI BLAST.

Chapter 6

Optimizing Irregular Application for Many-core Achitectures

6.1 cuBLASTP: Fine-Grained Parallelization of Protein Sequence Search on CPU+GPU

6.1.1 Introduction

The *Basic Local Alignment Search Tool (BLAST)* [119] is a fundamental algorithm in the life sciences that compares a query sequence to the database of known sequences in order to identify the most similar known sequences to the query sequence. The similarities identified by BLAST can then be used to infer functional and structural relationships between the corresponding biological entities, for example.

With the advent of next-generation sequencing (NGS) and the increase in sequence read-lengths, whether at the outset or downstream from NGS, the exponential growth of sequence databases is arguably outstripping our ability to analyze the data. Consequently, there have been significant efforts to accelerate sequence-alignment tools, such as BLAST, on various parallel architectures in recent years.

Graphics processing units (GPUs) offer the promise of accelerating bioinformatics algorithms and tools due to their superior performance and energy efficiency. However, in spite of the promising speedups that have been reported for other sequence alignment tools such as the Smith-Waterman algorithm [168], BLAST remains the most popular sequence analysis tool but also one of the most challenging to accelerate on GPUs.

Due to its popularity, the BLAST algorithm has been heavily optimized for CPU architectures over the past two decades. However, these CPU-oriented optimizations create problems when accelerating BLAST on GPU architectures. First, to improve computational efficiency, BLAST employs input-sensitive heuristics to quickly eliminate unnecessary search spaces. While this technique is highly effective on CPUs, it induces unpredictable execution paths in the program, leading to many divergent branches on GPUs. Second, to improve memory-access efficiency, the data structures used in BLAST are finely tuned to leverage CPU caching. Re-using these data structures on GPUs, however, can lead to highly inefficient memory access because the cache size on GPUs is significantly smaller than that on CPUs and because the coalesced memory access is needed on GPUs to achieve good performance.

State-of-the-art BLAST realizations for protein sequence search on GPUs [130, 64, 129, 128] adopt a coarse-grained and embarrassingly parallel approach, where one sequence alignment is mapped to only one thread. In contrast, a fine-grained mapping approach, e.g., using warps of threads to accelerate one sequence alignment, could theoretically better leverage the abundant parallelism offered by GPUs. However, such an approach presents significant challenges, mainly due to the high irregularity in execution paths and memory-access patterns that are found in CPU-based realizations of the BLAST algorithm. Thus, accelerating BLAST on GPUs requires a fundamental rethinking in the algorithmic design of BLAST.

Consequently, we propose cuBLASTP, a novel fine-grained mapping of the BLAST algorithm for protein search (BLASTP) onto a GPU, that improves performance by addressing the irregular execution paths caused by branch divergence and irregular memory access with the following techniques.

- First, we identify the BLAST kernel on a GPU is a *SDMC* class problem that can result in branch divergence and irregular memory accesses. Therefore, we *decouple* the phases in the BLASTP algorithm (i.e., hit detection and ungapped extension) to eliminate branch divergence and parallelize the phases having different computational patterns with different strategies on the GPU or CPU, as appropriate.

- Second, we propose a *data reordering pipeline* of binning-sorting-filtering as an additional phase between the phases of BLASTP to reduce irregular memory accesses.

- Third, we propose three implementations for the ungapped-extension phase with different parallel granularities, including diagonal-based parallelism, hit-based parallelism, and window-based parallelism. Fourth, we design a hierarchical buffering mechanism for the core data structures, i.e., deterministic finite automaton (DFA) and the position-specific scoring matrix (PSS matrix), to explore the new memory hierarchy provided by the NVIDIA Kepler architecture.

- Finally, we also optimize the remaining phases of BLASTP, i.e., gapped extension and alignment with traceback, on a multi-core CPU and overlap the phases running on the CPU with those running on the GPU.

Experimental results show that cuBLASTP can achieve up to a 3.4-fold speedup for the overall performance over the multi-threaded implementation on a quad-core CPU. Compared with the latest GPU implementation - CUDA-BLASTP, cuBLASTP delivers up to a 2.9-fold speedup for the critical phases of cuBLASTP and a 2.8-fold speedup for the overall performance.

6.1.2 Design of a Fine-Grained BLASTP

Here we first analyze the challenges in our coarse-grained BLASTP algorithm on the GPU. Then we introduce our fine-grained BLASTP algorithm. The basic idea is to explicitly partition the phases of BLASTP from within a single kernel into multiple kernels, where each kernel is optimized to run across a *group of GPU threads*. In particular, this is done for hit detection and ungapped extension. We then present our CPU-based optimizations for the two remaining phases, i.e., gapped extension and alignment with traceback.

6.1.2.1 Challenges of Mapping BLASTP to GPUs

Fig. 6.1 shows how hit detection and ungapped extension execute in the default BLASTP algorithm. In the hit detection, each subject sequence in the database is scanned from left to right to generate words; each word, in turn, is searched in the DFA of the query sequence. The positions with similar words found in the query sequence are tagged as hits, with each hit denoted as a tuple of two elements — $(QueryPos, SubPos)$, where $QueryPos$ is the position in the query sequence and $SubPos$ is the position in the subject sequence. For example, the word ABC in the subject sequence is searched in the DFA and found in positions 1, 7, and 11 of the query sequence, which in turn generates the following tuple hits: $(1, 3)$, $(7, 3)$, and $(11, 3)$.

After finding the hits, the BLASTP algorithm starts the ungapped extension. The algorithm uses a global array denoted as *lasthit_arr* to record the hits found in the previous detection for each diagonal. In the ungapped extension, the algorithm checks the previous hits in the same diagonals with the current hits. If the distance between the previous hit and the current hit is smaller than the *threshold*, the ungapped extension continues until a gap is encountered. For example, when the word ABB is processed to generate the hits $(2, 8)$ and $(6, 8)$, the hits in the *lasthit_arr* array for diagonal 2 and diagonal 6 are checked.

Because all the hits in a column are tagged simultaneously, the hit detection proceeds in *column-major order*. However, the ungapped extension proceeds in *diagonal-major order*, where hits in a diagonal are checked from top left to bottom right. Fig. 6.1 also illustrates the memory-access order on the *lasthit_arr* array. With the interleaved execution of hit detection and ungapped extension, memory access on the *lasthit_arr* array is highly irregular.

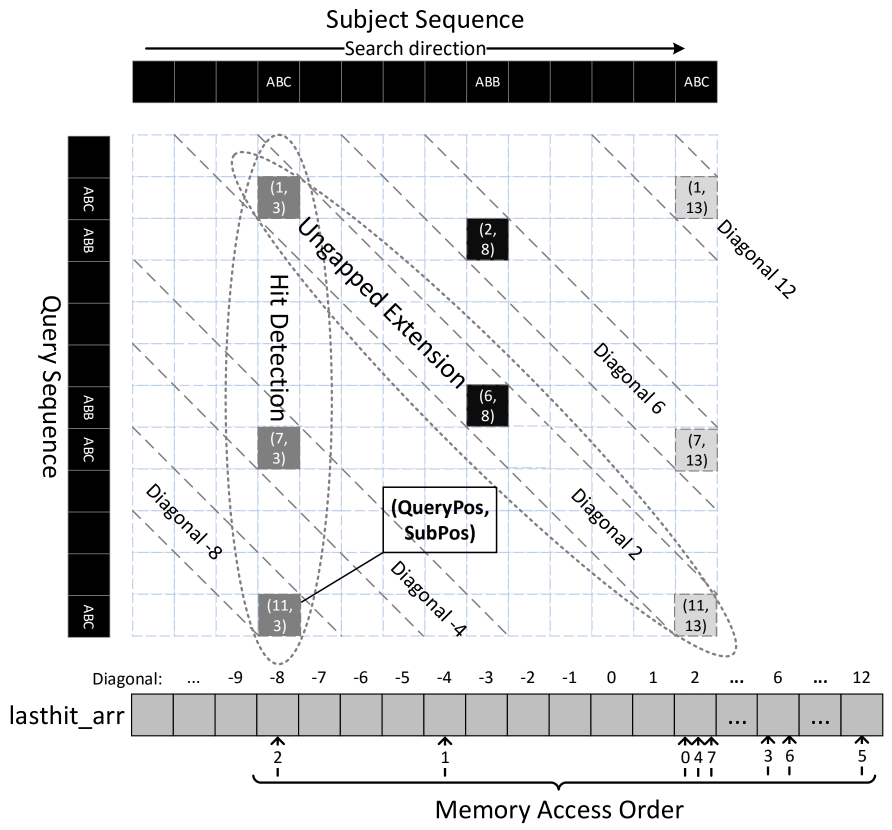


Figure 6.1: BLASTP hit detection and ungapped extension

Algorithm 7 illustrates the traditional BLASTP algorithm, on either CPU or GPU. When a hit is detected, the corresponding diagonal number (i.e., diagonal id) is calculated as the difference of *hit.sub_pos* and *hit.query_pos*, as shown in Line 5. The previous hit in this diagonal is obtained from the *lasthit_arr* array (Fig. 6.1). If the distance between the current hit and previous hit is less than the threshold, the ungapped extension is triggered. After the ungapped extension occurs in the current diagonal, the extended position in the subject sequence is used to update the previous hit in the *lasthit_arr* array. After all hits in the current column are checked in the ungapped-extension phase, the algorithm moves forward to the next word in the subject sequence.

Algorithm 7 Hit Detection and Ungapped Extension

```
Input: $database$: sequence database;
$DFA$: DFA lookup table base on query sequence
Output: $extensions$: results of ungapped extension
 1: for all $sequence_i$ in $database$ do
 2:     for all $word_j$ in $sequence_i$ do
 3:         find $hits$ for $word_j$ in $DFA$
 4:         for all $hit_k$ in $hits$ do
 5:             $diagonal \leftarrow hit_k.sub\_pos - j + query\_length$        ▷ calculate diagonal number
 6:             $lasthit \leftarrow lasthit\_arr[diagonal]$        ▷ get lasthit in the same diagonal
 7:             $distance \leftarrow hit_k.sub\_pos - lasthit.sub\_pos$        ▷ calculate distance to lasthit
 8:             if $distance$ within $threshold$ then
 9:                 $ext \leftarrow ungapped\_ext(hit_k, lasthit)$        ▷ perform the ungapped extension
10:                 $extensions.add(ext)$
11:                 $lasthit\_arr[diagonal] \leftarrow ext.sub\_pos$        ▷ update lasthit with ext position
12:             else
13:                 $lasthit\_arr[diagonal] \leftarrow hit.sub\_pos$        ▷ update lasthit with hit position
14:             end if
15:         end for
16:     end for
17: end for
18: output $extensions$
```

Fig. 6.2 shows how the BLASTP algorithm traditionally maps onto a GPU. It is a coarse-grained approach where all the phases of the alignment between the query sequence and one subject sequence are handled by a dedicated thread on the GPU. Because of the heuristic nature of BLASTP, there exist irregular execution paths in different subject sequences from a sequence database. Since the number of hits that trigger the ungapped extension in different sequences cannot be deduced in advance, branch divergence (and in turn, load imbalance) occurs when using coarse-grained parallelism in BLASTP. For example, while thread 2 works on the ungapped extension, as shown in Fig. 6.2, neither thread 0 nor thread 1 can trigger because in thread 0, there is no hit found in the hit detection, and in thread 1, the distance between the current hit and previous hit is larger than the threshold T. As a result, the branch divergence in this warp cripples the performance of BLASTP on a GPU.

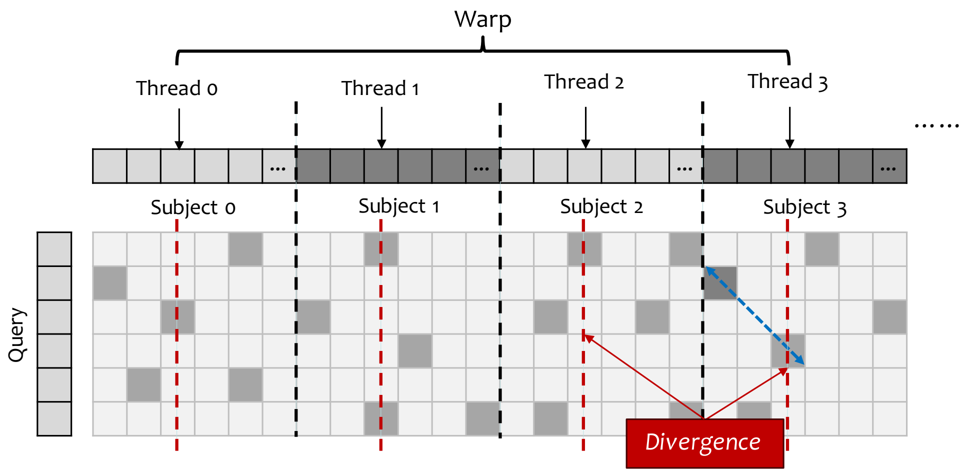


Figure 6.2: Branch divergence in coarse-grained BLASTP

Irregular memory access further impacts the performance of BLASTP on a GPU. Because the current hits can lead to irregular memory access on the previous hits in the *lasthit_arr* array and because each thread has its own *lasthit_arr* when pursuing coarse-grained parallelism for BLASTP, coalesced memory access when the threads of a warp are used for different sequence alignments proves to be effectively impossible.

Even a straightforward *fine-grained* multithreaded approach that uses multiple threads to unfold the "for" loop in Algorithm 7 can also lead to severe branch divergence on a GPU. Why? Due to the uncertainty in both the number of hits on different words and the distance to previous hits along the diagonals. Furthermore, since any position in the *lasthit_arr* array can be accessed during any one iteration, this approach can also cause significant memory-access conflicts. Thus, designing an effective fine-grained parallelization of BLASTP that fully utilizes the capability of the GPU is a daunting challenge. To address this, we decouple the phases of the BLASTP algorithm, use different strategies to optimize each of them, and propose a "binning-sorting-filtering" pipeline based on the method presented in Section 4.3 to reorder memory accesses and eliminate branch divergence, as articulated in the following subsections.

6.1.2.2 Hit Detection with Binning

We first decouple the phases of hit detection and ungapped extension into separate kernels. In our fine-grained hit detection, we use multiple threads to detect consecutive words in a subject sequence and to ensure the coalesced memory access. In addition, because the ungapped extension executes along the diagonals, we re-organize the output results of the hit-detection into diagonal-major order and introduce a binning data structure and bin-based algorithms to bridge the phases of hit detection and ungapped extension. Specifically, we allocate a contiguous buffer in global memory and logically organize this buffer into bins (which will map onto the diagonals) to hold the hits. While a bin could be allocated for one diagonal, we allocate a bin for multiple diagonals to reduce memory usage on the GPU and to allow longer sequences to be handled.

Fig. 6.3 illustrates our approach to the fine-grained hit detection, where *each* word in the subject sequence is scheduled to one thread. A thread retrieves a word from the corresponding position (i.e., column number or id) in the subject sequence, searches the word in the DFA to get the hit positions (i.e., row number or id), and immediately calculates the diagonal numbers as the difference in corresponding column number and row number. For example, thread 3 retrieves word ABC from column 3 of the subject sequence, searches for ABC in the DFA to get hit positions 1, 7, and 11, and calculates the diagonal numbers as 2, -4, and -8, respectively. Since multiple

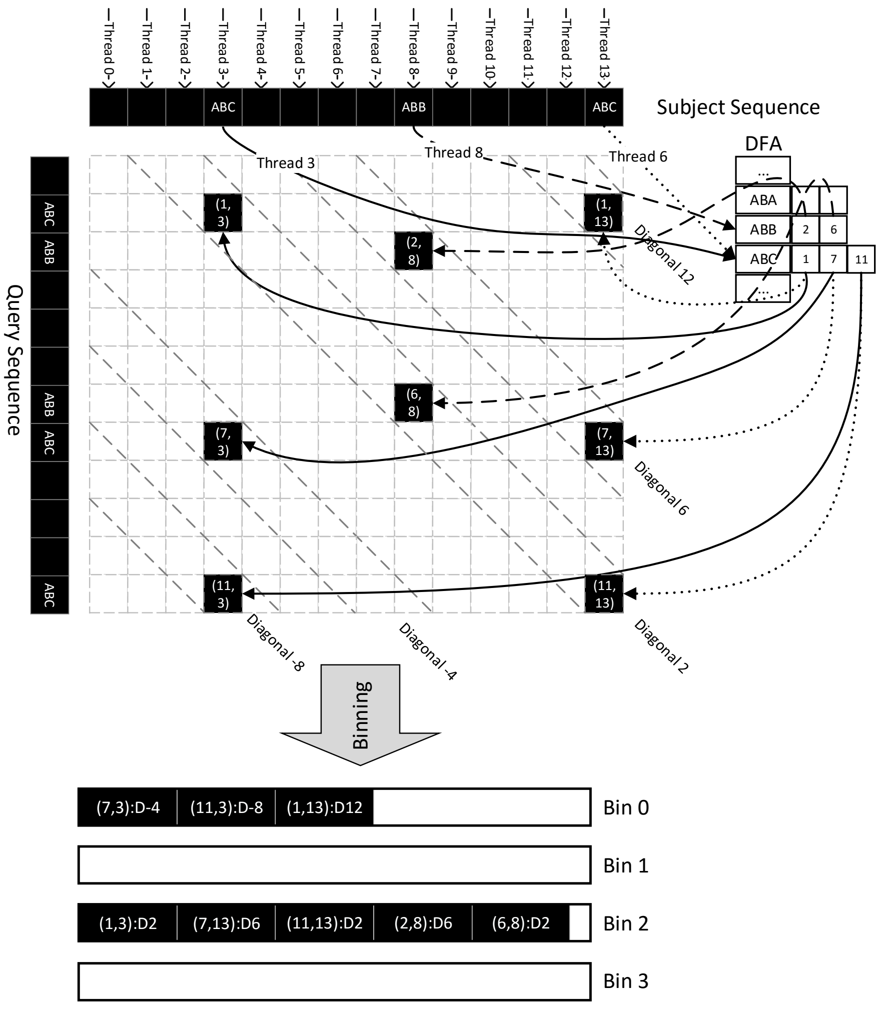


Figure 6.3: Hit detection and binning

threads can write hit positions into the same bin simultaneously, we must use atomic operations to address write conflicts, and in turn, ensure correctness.

Algorithm 8 Warp-based Hit Detection

Input: $database$: sequence database;
DFA: DFA lookup table base on query sequence
Output: $bins$: diagonal-based bins that store hits

```
1:  $tid \leftarrow blockDim.x * blockIdx.x + threadIdx.x$
2:  $numWarps \leftarrow gridDim.x * blockDim.x / warpSize$   ▷ calculate total number of warps
3:  $warpId \leftarrow tid/warpSize$
4:  $laneId \leftarrow threadIdx.x$ mod $warpSize$   ▷ initialize $i$ with $warpId$
5:  $i \leftarrow warpId$
6:  while $database$ has $i$-th sequence do
7:      $j \leftarrow laneId$   ▷ initialize $j$ with $laneId$
8:      while $i$-th sequence has $j$-th word do
9:          find $hits$ of $j$-th word in $DFA$
10:         for all $hit_k$ in $hits$ do
11:             $diagonal \leftarrow hit_k.sub\_pos - j + query\_length$
12:             $binId \leftarrow diagonal$ mod $num\_bins$   ▷ calculate bin number
13:             $curr \leftarrow atomicAdd(top[bin\_id], 1)$   ▷ increment hit counts of the bin
14:             $bins[binId][curr] \leftarrow hit_k$   ▷ store the hit into the bin
15:         end for
16:         $j \leftarrow j + warpSize$   ▷ continue $j + warpSize$-th word
17:     end while
18:     $i \leftarrow i + numWarps$   ▷ continue $i + numWarps$-th sequence
19: end while
20: output $bins$
```

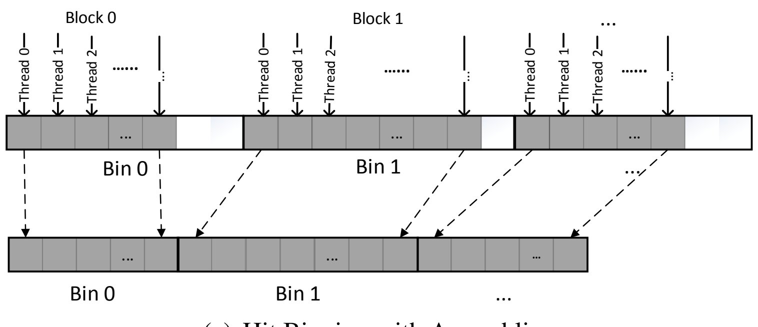
Algorithm 8 describes our fine-grained hit detection algorithm. *num_bins* represents the number of bins, which is a configurable parameter. The algorithm schedules a warp of threads for a sequence based on *warpId*. The word *seq[i][j]* in position *j* of sequence *seq[i]* is handled by the thread with the *laneId j*. For each hit of the word, the diagonal number is calculated and mapped to a bin (Line 12).

The *top* array stores the currently available position in each bin. Using atomic operations on the *top* array in the shared memory, we avoid the heavyweight overhead of atomic operations on global memory. The warp is then scheduled to handle the next sequence after all words in the current sequence are processed.

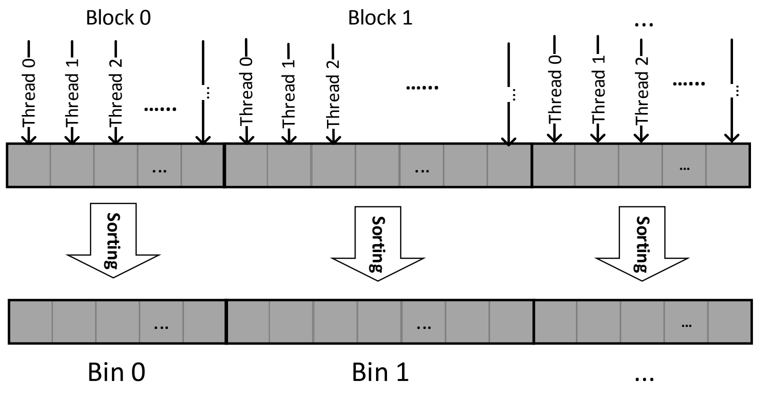
6.1.2.3 Hit Reordering

After the hit detection, hits are grouped into bins by diagonal numbers. Because multiple threads can write hits from different diagonals into the same bin simultaneously, hits in each bin could interleave. For example, Fig. 6.3 shows that hits belonging to diagonal 2 and diagonal 6 interleave. Because the ungapped extension can only extend continuous hits whose distance is less than a threshold, we need to further reorder the hits in each bin to enable contiguous memory access during the ungapped extension. To achieve this, we propose a hit-reordering pipeline that includes *binning, sorting, and filtering*. Fig. 6.4 provides illustrative examples of these three kernels, respectively.

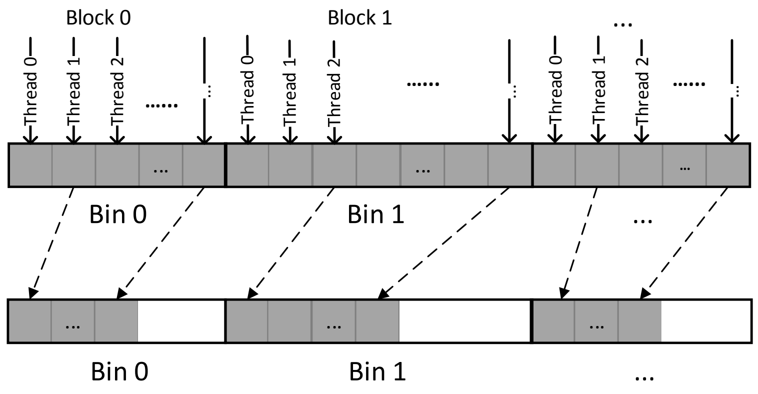
Hit Binning with Assembling: Because it is effectively impossible to get an accurate number of hits for each subject sequence before the completion of the hit-detection, we allocate the maximally possible size (i.e., number of words in the query sequence) as the buffer size of each bin. Though this leads to unused memory in the bins, it offers the promise of high performance as we can use a segmented sort [169] to sort the hits per bin. To maximize the throughput of the sort, the data must be contiguously stored, even if they belong to different segments. Thus, prior to sorting, we launch a kernel that assembles the hits from different bins into a large but contiguous array, as shown in Fig. 6.4(a). Each bin is then processed by a block of threads consecutively for the coalesced



(a) Hit Binning with Assembling



(b) Hit Sorting



(c) Hit Filtering

Figure 6.4: Three kernels for assembling, sorting, and filtering

memory access.

Hit Sorting: A hit includes four attributes: the row number that is the position in the query sequence; the column number that is the position in the subject sequence; the diagonal number that is calculated as the difference of the column number and row number; and the sequence number that is the index of the subject sequence. To unify the attributes and only have to sort once, we propose a bin data structure for the hits. As shown in Fig. 6.5, we pack the sequence number, diagonal number, and subject position into a 64-bit integer. Because the longest sequence in the most recent NCBI NR database [170] contains 36,805 letters, 16 bits is sufficient to record the subject position and 16 bits for the diagonal number, each of which can represent 64K positions. With this data structure, we sort hits in each bin *once* instead of by the diagonal number and subject position, respectively. The packed data structure also can reduce memory accesses.

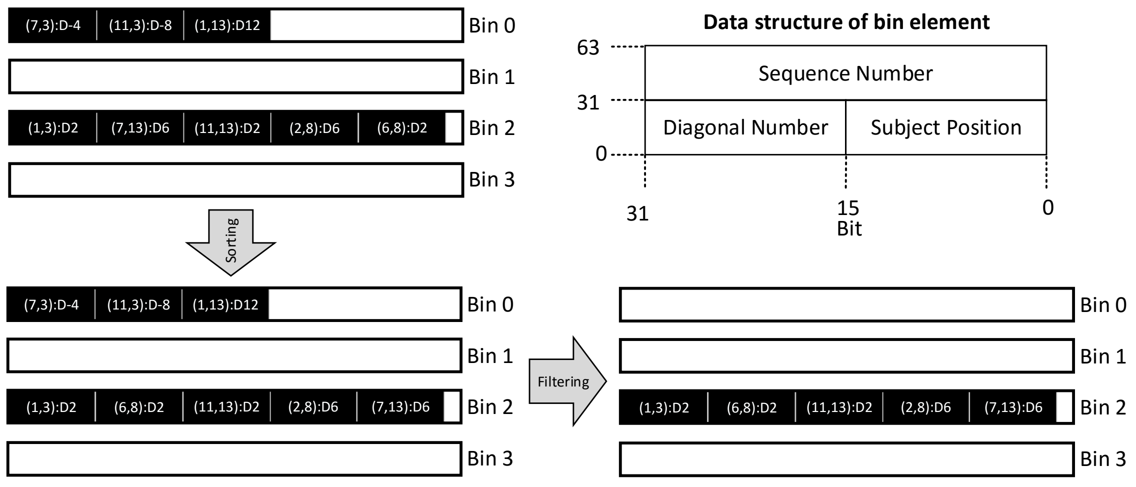


Figure 6.5: Sorting and filtering on the bin structure

Using the segmented sort kernel from the Modern GPU Library [169] by NVIDIA, according to the experiments, we found that as we vary the number of segments for a given data size, the throughput increases as more segments are used. Since the total number of hits after the hit-detection is fixed, we can increase the number of bins to improve sorting performance but at the expense of more memory usage. Because GPU device memory is limited, we must choose an appropriate number of bins to balance the sorting performance and memory storage. We set the number of bins as a configurable parameter in our cuBLASTP algorithm, which relies on many factors, such as the size of device memory and the query length.

Hit Filtering: With the bins now sorted, we introduce hit filtering to eliminate hits whose distances with neighbors are larger than a specified threshold because these hits cannot trigger the ungapped extension. As shown in Fig. 6.4(c), we use a block of threads to check consecutive hits in each bin for the coalesced memory access. We assign a thread for a hit to compare the distance to its neighbor on the left. If the distance to the neighbor is less than the threshold, the hit is kept and passed to the ungapped-extension.

To avoid global synchronization and atomic operations, we write extendable hits into a dedicated buffer that is maintained by each thread block. The overall performance of this additional filtering step is then determined by the ratio of the overhead of hit filtering over the overhead of branch divergence. (Our experimental results show that only 5% to 11% of the hits from the hit-detection are passed to the ungapped extension; thus the overall cuBLASTP performance improves due to this hit filtering.)

6.1.2.4 Fine-Grained Ungapped Extension

After hit reordering, the hits in each bin are arranged in ascending order by diagonals, and the hits that cannot be used to trigger the ungapped extension have been filtered out. Based on the ordered hits, we design a *diagonal-based, ungapped-extension* algorithm, as depicted in Algorithm 9, where *each* diagonal is processed by a thread. So, as shown from Lines 6 to 8, different threads are scheduled to different bins, and threads in a warp are scheduled to different diagonals based on the *warpId*. We then call the *ungapped_ext* function to extend the diagonal until a gap is encountered or the diagonal is ended. *ext* represents the extension result. Because an extension could cover other hits along the diagonal, Line 14 determines if a hit is covered by the previous extension. If the hit is *not* covered by the previous extension, it can be used to trigger an extension. However, this extension method could introduce branch divergent due to various extension length.

Due to the above divergent branching, we propose an alternative fine-grained approach to Algorithm 9 called *hit-based ungapped extension*, as shown in Algorithm 10. This approach seeks to improve performance by trading off divergent branching for redundant computation. Specifically, each thread extends a hit independently. Thus, different hits could have the same extension, which can result in redundant computation and duplicated results. These duplicates are then independently stored on a per-thread basis (Line 13). Unlike Algorithm 9, this algorithm requires a de-duplication step before the remaining phases of gapped extension and alignment with traceback.

Intuitively, which of the two algorithms performs best depends on hits between the query sequence and the subject sequences. If there are too many hits that will be covered by the extension of other

Algorithm 9 Diagonal-based Ungapped Extension

Input: $bins$ binned hits

Output: $extensions$: results of ungapped extension

```
1: $tid \leftarrow blockDim.x * blockIdx.x + threadIdx.x$
2: $numWarps \leftarrow gridDim.x * blockDim.x / warpSize$
3: $warpId \leftarrow tid / warpSize$
4: $laneId \leftarrow threadIdx.x \bmod warpSize$
5: $i \leftarrow warpId$
6: while $i < num\_bins$ do                                ▷ go through all bins by warps
7:     $j \leftarrow laneId$
8:     while $j < bin_i.num\_diagonals$ do                 ▷ process all diagonals in the bin by lanes
9:         $ext\_reach \leftarrow -1$                      ▷ initialize last extension position
10:        for all $hit_k$ in $diagonal_j$ do              ▷ go through all hits in the diagonal
11:            $sub\_pos \leftarrow hit_k.sub\_pos$
12:            $query\_pos \leftarrow hit_k.sub\_pos$
                   $-hit_k.diag\_num$
13:            $seq\_id \leftarrow hit_k.seq\_id$
14:            if $sub\_pos > ext\_reach$ then             ▷ check if the pos has been extended
15:                $ext \leftarrow ungapped\_ext(seq\_id, query\_pos, sub\_pos)$
16:                $extensions.add(ext)$
17:                $ext\_reach \leftarrow ext.sub\_pos$    ▷ update with new extension pos
18:            end if
19:        end for
20:        $j \leftarrow j + warpSize$
21:    end while
22:    $i \leftarrow i + numWarps$
23: end while
24: output $extensions$
```

hits in the diagonal, then diagonal-based the ungapped extension should perform better; otherwise, the hit-based ungapped extension will. However, while hit-based extension eliminates divergent branching, it can create load imbalance. That is because different hits in one diagonal could be extended to different lengths and if (at least) a hit can be extended much longer than other hits, then all other threads in the warp must wait for the completion of the longest extension.

To address the above, we present a window-based extension, as detailed in Algorithm 11. It con-

Algorithm 10 Hit-based Ungapped Extension

Input: bin binned hits

Output: $extensions$: results of the ungapped extension

```
 1: $tid \leftarrow blockDim.x * blockIdx.x + threadIdx.x$
 2: $numWarps \leftarrow gridDim.x * blockDim.x / warpSize$
 3: $warpId \leftarrow tid / warpSize$
 4: $laneId \leftarrow threadIdx.x \bmod warpSize$
 5: $i \leftarrow warpId$
 6: while $i < num\_bins$ do
 7:     $j \leftarrow laneId$
 8:     while $j < bin_i.num\_hits$ do          ▷ process all hits in the bin by lanes in parallel
 9:         $sub\_pos \leftarrow hit_j.sub\_pos$
10:         $query\_pos \leftarrow hit_j.sub\_pos - hit_j.diag\_num$
11:         $seq\_id \leftarrow hit_j.seq\_id$
12:         $ext \leftarrow ungapped\_ext(seq\_id, query\_pos, sub\_pos)$
13:         $extensions.add(ext)$
14:         $j \leftarrow j + warpSize$
15:     end while
16:     $i \leftarrow i + numWarps$
17: end while
18: output $extensions$
```

sists of the following steps: (1) divide a warp of threads into different windows; (2) map a window to a diagonal; and (3) extend hits in a diagonal one by one using a window-sized set of threads at the same time. Because this approach uses a window-sized set of threads to extend a single hit, it can speed up the hit-based extension on the longest extension and reduce the load imbalance that would otherwise more adversely affect performance.

Fig. 6.6 illustrates how computation proceeds in the window-based ungapped extension, along with details on gap detection. A gap can be detected by computing the accumulated score for each extended position from the hit position and then comparing the score change from the highest score along the extension with a threshold. In this figure, we present two windows, each of which extends the IYP hit along the diagonal but in opposite directions.

Algorithm 11 Window-based Ungapped Extension

Input: bin binned hits, $winSize$ size of windows

Output: $extensions$: results of ungapped extension

```
 1: $numBlocks \leftarrow gridDim.x$
 2: $numWin \leftarrow blockDim.x/winSize$                ▷ get number of windows in a thread block
 3: $winId \leftarrow threadIdx.x/winSize$                ▷ get window id
 4: $wLaneId \leftarrow threadIdx.x \bmod winSize$        ▷ get lane id in the window
 5: $i \leftarrow blockIdx.x$
 6: **while** $i < num\_bins$ **do**                      ▷ go through all bins by blocks
 7:     $j \leftarrow winId$
 8:     **while** $j < bin_i.num\_diagonals$ **do**       ▷ go through all diagonals in the bin
 9:         $ext\_reach \leftarrow -1$
10:         **for all** $hit_k$ in $diagonal_j$ **do**    ▷ go through all hits in the diagonal by wins
11:             $sub\_pos \leftarrow hit_k.sub\_pos$
12:             $query\_pos \leftarrow hit_k.sub\_pos - hit_k.diag\_num$
13:             $seq\_id \leftarrow hit_k.seq\_id$
14:             **if** $sub\_pos > ext\_reach$ **then**
15:                 ▷ perform window-based extension
16:                 $ext \leftarrow ungapped\_ext\_win(seq\_id, query\_pos, sub\_pos, wLaneId, winSize)$
17:                 **if** $wLaneId = 0$ **then**
18:                     $extensions.add(ext)$
19:                 **end if**
20:                 $ext\_reach \leftarrow ext.sub\_pos$
21:             **end if**
22:         **end for**
23:         $j \leftarrow j + numWin$
24:     **end while**
25:     $i \leftarrow i + numBlocks$
26: **end while**
27: output $extensions$
```

For brevity, we only discuss the extension to the IYP hit with the right window; the left window is handled concurrently in a similar fashion. First, we map the window-sized set of threads (in this case, 8) along consecutive positions from the hit and then calculate the prefix sum of each position for the *PrefixSum* array using the optimized scan algorithm derived from the CUB library [171]. This prefix sum in the right window produces the highest score of 12, as circled in the *PrefixSum* array.

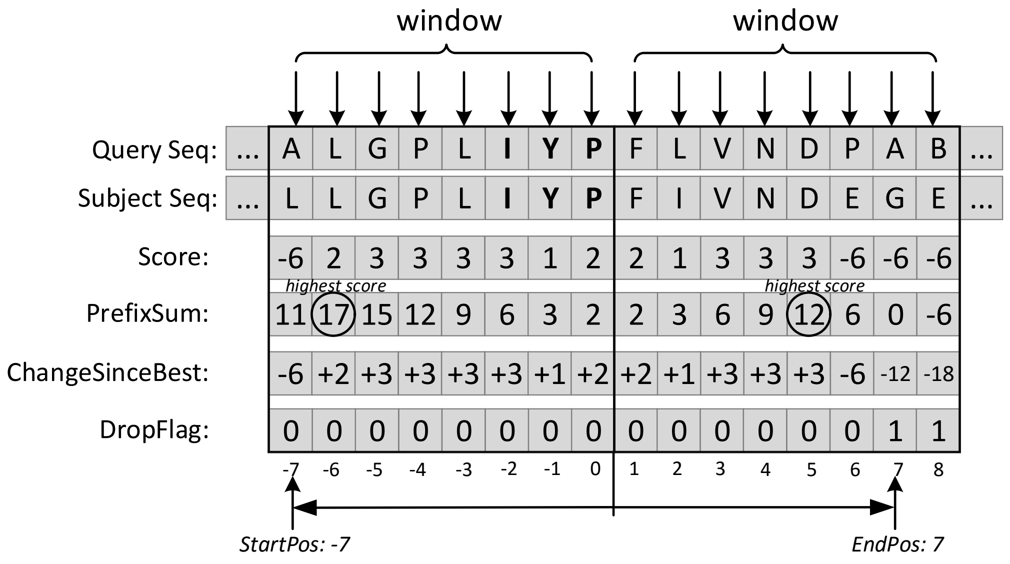
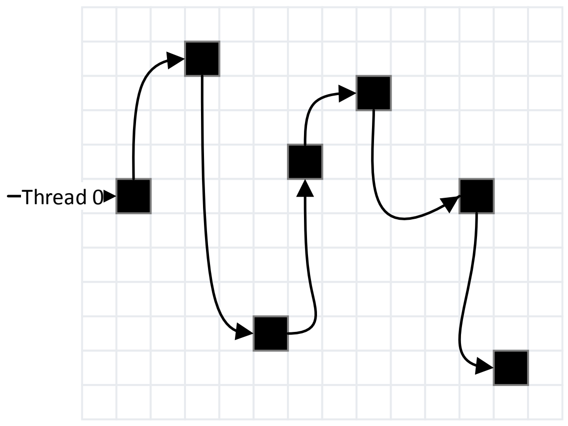
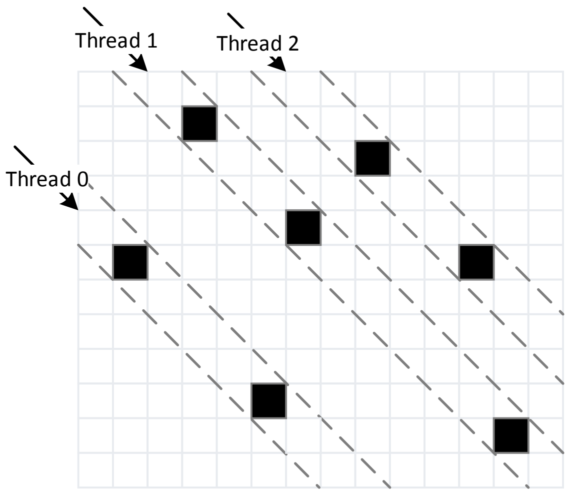

window
window
Query Seq: ... A L G P L I Y P F L V N D P A B ...
Subject Seq: ... L L G P L I Y P F I V N D E G E ...
Score: -6 2 3 3 3 3 1 2 2 1 3 3 3 -6 -6 -6
highest score
highest score
PrefixSum: 11 17 15 12 9 6 3 2 2 3 6 9 12 6 0 -6
ChangeSinceBest: -6 +2 +3 +3 +3 +3 +1 +2 +2 +1 +3 +3 +3 -6 -12 -18
DropFlag: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
-7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8
StartPos: -7
EndPos: 7

Figure 6.6: Example of window-based extension. In this example, the *dropoff* threshold is -10.

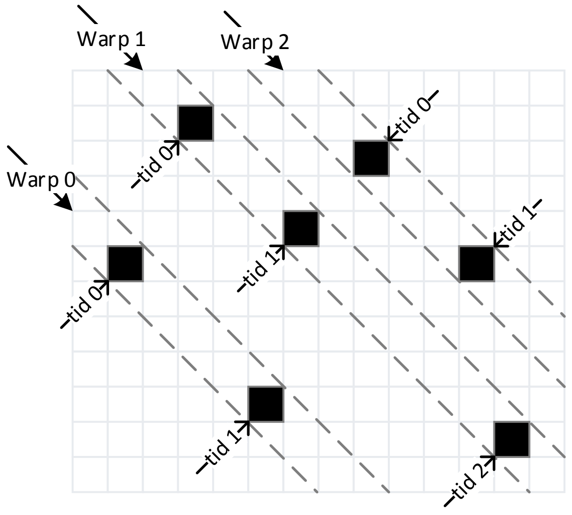
Then, each thread after the position with the highest score calculates the score changed from the highest score while the threads before the highest score position simply record the contribution to the highest score, i.e., the changes from the previous positions. After this step, our window-based algorithm generates the *ChangeSinceBest*. Next, by comparing to the *dropoff* threshold (i.e.,-10, as noted in the figure), the algorithm then generates the *DropFlag* array. If the change is more than the threshold, a “1” is set to denote this position as a gap; otherwise, a “0” is set. If there is a gap, the algorithm then writes the start position and end position of this extension with the highest score into the output of the ungapped extension. If there is no gap in the window like the left window in the figure, the algorithm goes to the next iteration to move the windows forward. (This figure also illustrates the redundant computation in the window-based ungapped extension: even if the gap exists in the middle of the window, all positions of the window have to be checked.)



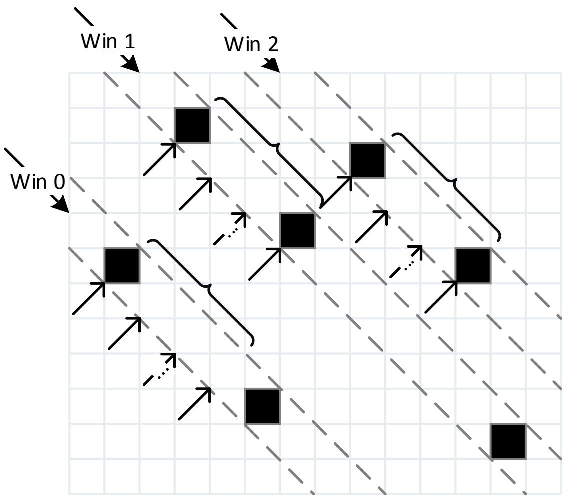
(a) Coarse-grained extension



(b) Diagonal-based extension



(c) Hit-based extension



(d) Window-based extension

Figure 6.7: Four parallelism strategies of the ungapped extension

Algorithm 11 describes the details of the window-based ungapped extension. Because we use a window per diagonal to check hits one by one, we still need to check whether the current hit is covered by the previous extension at Line 14. However, this approach removes the redundant computation that would have otherwise been done with our hit-based extension. As a result, we use a configurable parameter to allow the user to select which the ungapped extension algorithm to execute at runtime: diagonal-based, hit-based, or window-based, as noted in Fig. 6.7.

6.1.2.5 Hierarchical Buffering

To fully utilize memory bandwidth and further improve cuBLASTP performance, we propose a *hierarchical buffering* approach for the core data structure (DFA) used in the hit detection. As shown in Fig. 2.13(a), the DFA consists of the states in the finite state machine and the query positions for the states. Both the states and query positions are highly reused in the hit detection for words in subject sequences. Loading the DFA into the shared memory can improve the data access bandwidth. However, because the number of query positions depends on the length of the query sequence, fetching all positions into the shared memory may affect the occupancy of GPU kernels and offset the improvement from higher data access bandwidth, especially for long sequences. Thus, we load the states that have relatively fixed but small size into the shared memory and store the query positions into constant memory.

On the latest NVIDIA Kepler GPU, a 48-KB read-only cache with relaxed memory coalescing rules provides reusable but randomly accessed data. We allocate the query positions in the global memory but tag them with the keyword “const __restrict” for loading them into the read-only cache automatically.

Fig. 6.8 shows the hierarchical buffering architecture for the DFA on a Kepler GPU. We put the DFA states, e.g., ABB and ABC, into the shared memory. For the first access of ABB from thread 3, the positions are written into bins and loaded into the read-only cache. For the subsequent access of ABB from thread 4, the positions are obtained from the cache.

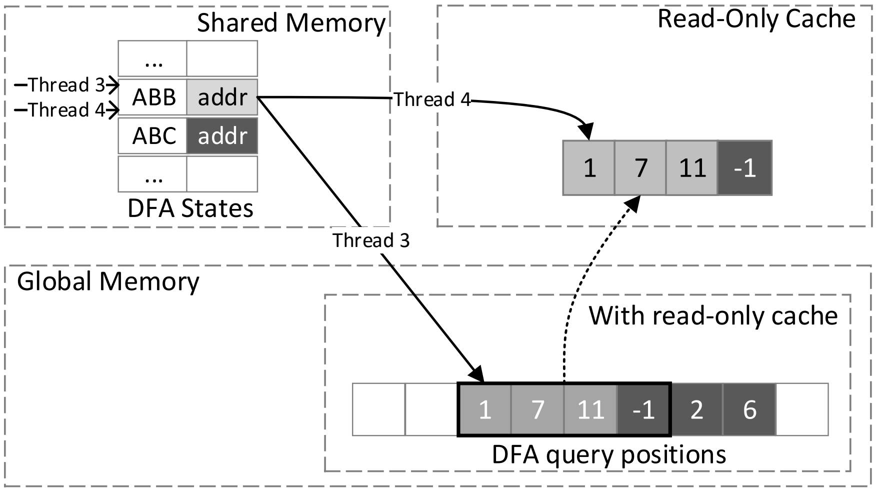


Figure 6.8: Hierarchical buffering for DFA on the NVIDIA Kepler GPU

The PSS matrix is another core data structure that is highly reused in the ungapped extension. The number of columns in the PSS matrix is equal to the length of the query sequence, as shown in Fig. 2.13(b). However, because each column contains 64 bytes (32 rows with 2 bytes for each), the size of the PSS matrix increases quickly with the query length. The 48-KB shared memory cannot hold the PSS matrix when the query sequence is longer than 768.

On the other hand, the scoring matrix can be used to substitute the PSS matrix. For example, the BLOSUM62 matrix, which consists of 32 * 32 = 1024 elements and has a fixed size of only 2 KB (i.e., 2 bytes per element), can be always put into the shared memory. Therefore, for longer query sequences, the BLOSUM62 matrix in the shared memory can provide better performance, even though more memory operations are needed compared with the PSS matrix for short sequences. Thus, we provide a configurable parameter to select PSS matrix or scoring matrix. For the PSS matrix, we put it into the shared memory until a threshold and then we put it into the global memory. For the scoring matrix, we always put it into the shared memory. We will compare the performance using the PSS matrix and the scoring matrix in Section 6.1.4.

6.1.3 Optimizing Gapped Extension and Alignment with Traceback on a Multi-core CPU

After the most time-consuming phases of BLASTP accelerated, the remaining phases, i.e., gapped extension and alignment with traceback, now consume the largest percentage of the total time. Specifically, for a query sequence with 517 characters (i.e., *Query517*), Fig. 6.9 shows that after applied fine-grained optimizations on the GPU, the percentage of time spent on hit detection and ungapped extension is dropped from 80% (FSA-BLAST) down to 52% (cuBLASTP with one CPU). The percentage of time spent on gapped extension and alignment with traceback, however, grows up from 13% to 32% and 5% to 13%, respectively. Thus, it is necessary to optimize these two stages for better overall performance.

In the BLASTP algorithm, only the high-scoring seeds from the *un*gapped extension stage can be passed to the gapped-extension stage. Although the gapped extension on each seed is independent, and the extension itself is compute-intensive, only a small percentage of subject sequences require the gapped extension. If we offload the gapped extension to GPU, CPU will be idle during most

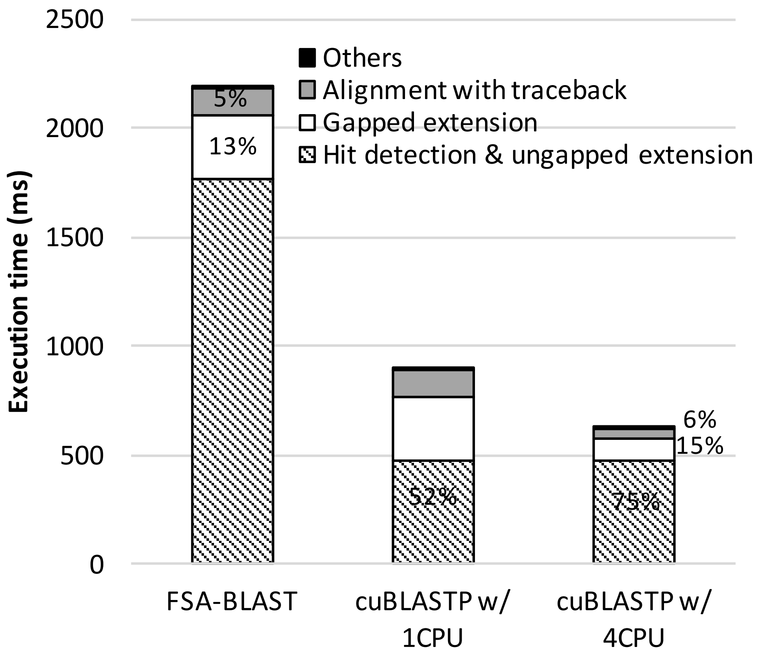


Figure 6.9: Breakdown of execution time for *Query517* on *Swissprot* database

of the BLASTP search. In order to improve the resource utilization of the whole system, i.e., making use of both GPU and CPU, parallelize the gapped extension on CPU is an alternative. Furthermore, though there were several studies proposed to parallelize the gapped extension on GPU, e.g., CUDA-BLASTP, they had to modify the dynamic programming method of the gapped extension on GPU for the performance. As a result, we optimize the gapped extension on CPU with Pthreads. For the alignment with traceback, due to the data dependency and the random memory access, we also optimize it on CPU with multithreading. In order to reduce the overhead of data transfer between CPU and GPU, we design a pipeline to overlap the computations on CPU and GPU, and the data communication on PCIe. Fig. 6.10 illustrates the pipeline design. Once the kernels of hit detection and ungapped extension for one block of the database are finished on GPU, the intermediate data is sent back to CPU asynchronously for the remaining phases. At the same time, the kernels for hit detection and ungapped extension are triggered for the next data block. With the pipeline design, we can overlap the computations on CPU and GPU, and the data transfer on PCIe for different data blocks.

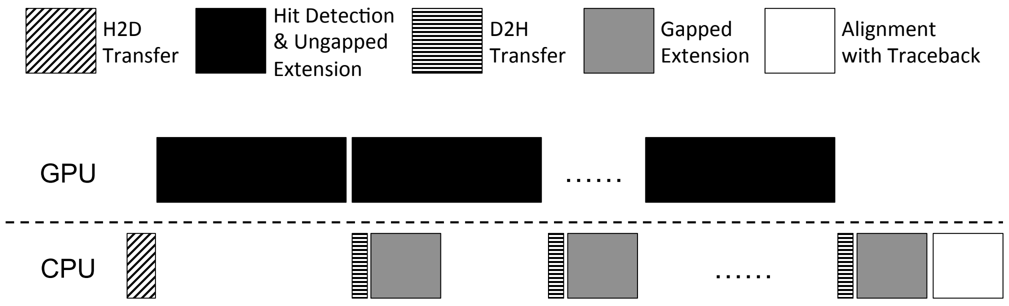


Figure 6.10: Overlapping hit detection and ungapped extension on a GPU and gapped extension and alignment with traceback on a CPU

Fig. 6.9 shows that the multithreaded optimization (cuBLASTP with four CPU threads) significantly improves the gapped extension and the alignment with traceback. Ultimately, the overall performance improvement is more than four-fold over FSA-BLAST. Fig. 6.11 shows multithreaded gapped extension and alignment with traceback exhibiting strong scaling.

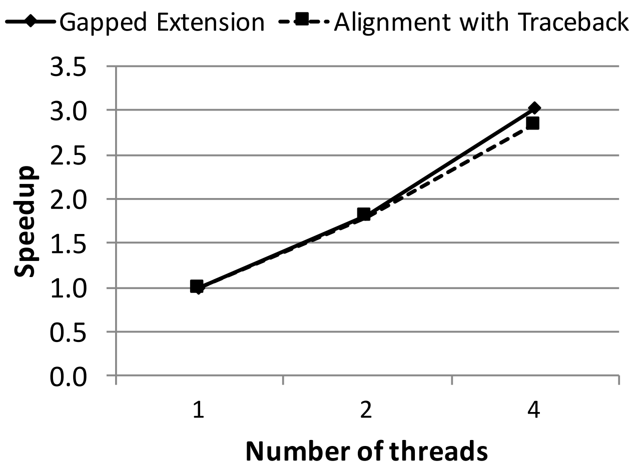


Figure 6.11: Strong scaling for gapped extension and alignment with traceback on a multi-core CPU

6.1.4 Performance Evaluation

We conduct our experimental evaluation on a compute node that includes an Intel Core i5-2400 quad-core processor (with 6MB shared L3 cache and 8GB DDR3 main memory) and a NVIDIA Kepler K20c GPU. The system runs Debian Linux 3.2.35-2 and NVIDIA CUDA toolkit 5.0. For input data, we use two typical NCBI databases [170]. The first database is *env_nr*, which includes about 6-million sequences whose total size is 1.7 GB and where the average length of the sequences is about 200 letters. The second is *swissprot*, which includes over 300,000 sequences with a total size of 150 MB. The average length is 370 letters. For the input query sequences, we choose three sequences, whose lengths are 127 ("query127"), 517 ("query517"), and 1054 ("query1054") bytes, to represent short, medium, and long sequences, respectively.

6.1.4.1 Evaluation of Configurable Parameters

We first evaluate the performance of cuBLASTP kernels with different numbers of bins. Fig. 6.12 shows that the performances of hit sorting and hit filtering can be constantly improved if we increase the number of bins per warp. However, the performance of hit detection drops dramatically after 128 bins. That is, because more bins will use more shared memory to record the current header, and in turn, decrease the occupancy of the kernel. Thus, in order to achieve the maximum overall performance, the optimal number of bins per warp should balance the performance of hit detection with hit sorting and filtering. In our experimental environment, we set the number of bins per warp to 128 for the best overall performance.

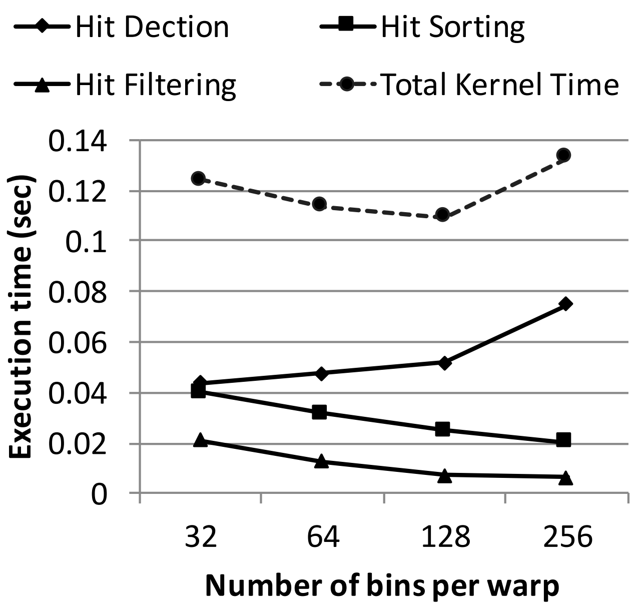


Figure 6.12: Execution time of different kernels with different numbers of bins for *Query517* on *Swissprot* database

Second, in the performance comparison of using the PSS and BLOSUM62 matrix, Fig. 6.13 shows that the PSS matrix performs better for the short sequence (query127) whereas the BLOSUM62 matrix performs better for longer sequences (query517 and query 1054), as reasoned and predicted in Section 6.1.2.5. In short, we observe a –24%, 50%, and 237% improvement in performance when using the BLOSUM62 matrix. As a result, we configure our algorithm to use the PSS matrix for “query127” and the BLOSUM62 matrix for “query517” and “query1057” on NVIDIA Kepler K20c GPU for the following evaluations.

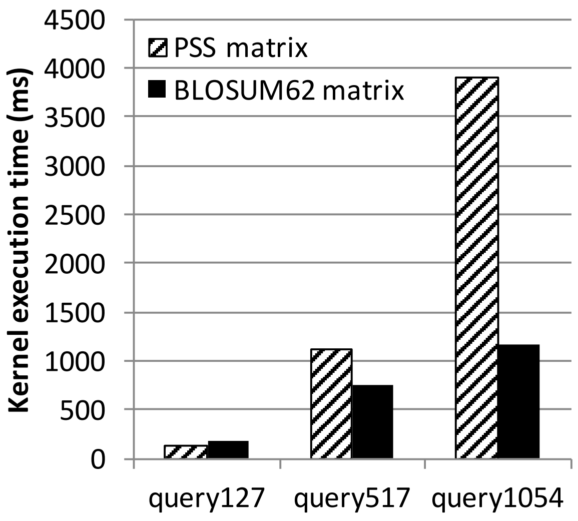
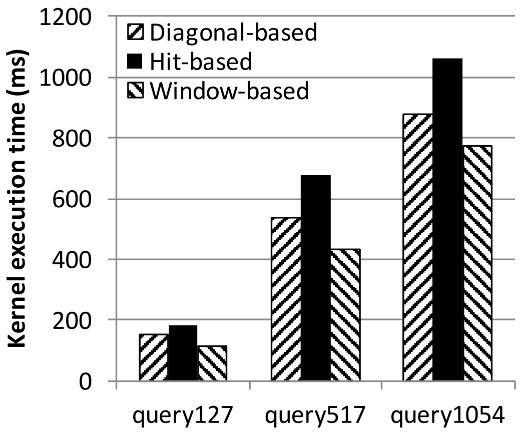


Figure 6.13: Performance with different scoring matrices

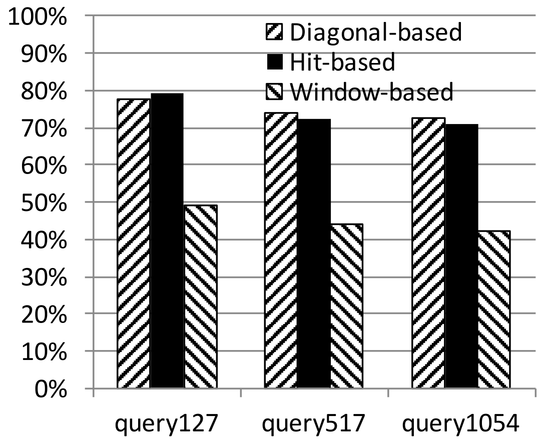
6.1.4.2 Evaluation of our Fine-Grained Algorithms for cuBLASTP: Diagonal-, Hit-, and Window-Based

Fig. 6.14(a) shows that window-based extension delivers 24%, 20%, and 12% better performance for *query127*, *query517*, and *query1054*, respectively, when compared to the diagonal-based extension. Similarly, the window-based extension achieves 38%, 36%, and 27% better performance when compared to the hit-based extension. Fig. 6.14(b) compares the divergence overhead of the three algorithms. The window-based algorithm experiences a significant improvement in divergence overhead when compared with the other two algorithms. As a result, we configure our cuBLASTP algorithm to use the window-based extension for these two databases on the NVIDIA Kepler K20c GPU in the following evaluations.

Fig. 6.15 illustrates that cuBLASTP performance can always improve by adopting our hierarchical buffering mechanism, where the read-only cache is used to store the DFA for the hit detection.



(a) Execution time



(b) Divergence overhead

Figure 6.14: Performance numbers with different extensions

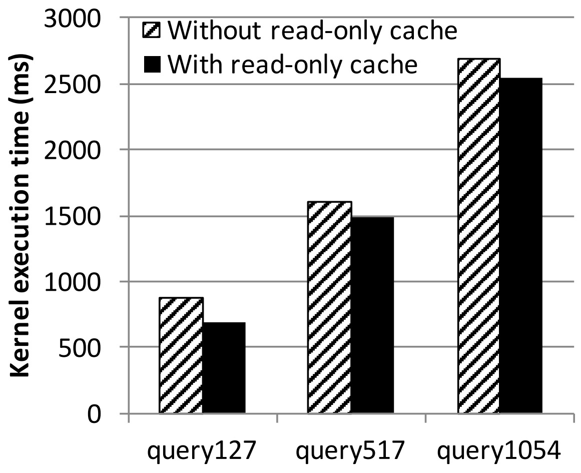


Figure 6.15: Performance with and without read-only cache

6.1.4.3 Performance Comparison to Existing BLASTP Algorithms

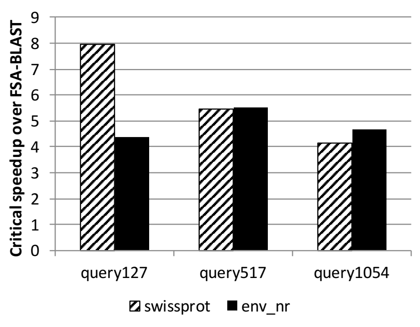
Fig. 6.16 presents the normalized speedup of our fine-grained cuBLASTP over the sequential FSA-BLAST on CPU, the multithreaded NCBI-BLAST on CPU, and the state of the art GPU-based implementations CUDA-BLASTP [130] and GPU-BLASTP [64].

Compared with the single-threaded FSA-BLAST, Fig. 6.16(a) shows that on the *swissprot* and *env_nr* database, cuBLASTP delivers up to 7.9-fold and 5.5-fold speedups for the critical phases of BLASTP, i.e., hit detection and ungapped extension. Fig. 6.16(b) shows that for the overall performance, the corresponding performance improvements using cuBLASTP are 3.6-fold and 6-fold, respectively.

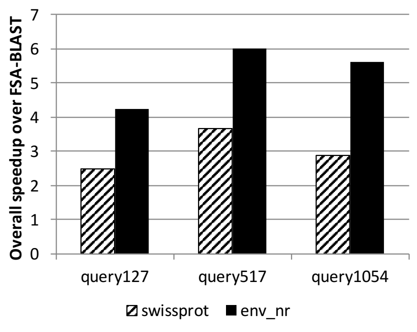
Compared with NCBI-BLAST with four threads, Fig. 6.16(c) shows that on the *swissprot* and *env_nr* database, cuBLASTP delivers up to 2.9-fold and 3.1-fold speedups for the critical phases. Fig. 6.16(d) shows that for the overall performance, the corresponding performance improvements using cuBLASTP are 2.1-fold and 3.4-fold, respectively.

Compared with CUDA-BLASTP on NVIDIA Kepler K20c GPU, Fig. 6.16(e) shows that on the *swissprot* and *env_nr* database, cuBLASTP delivers up to a 2.9-fold speedup and 2.1-fold speedup for the critical phases. Fig. 6.16(f) shows that for the overall performance, including all stages of BLASTP and the data transfer between CPU and GPU, the corresponding performance improvements using cuBLASTP are 2.8-fold and 2.5-fold, respectively.

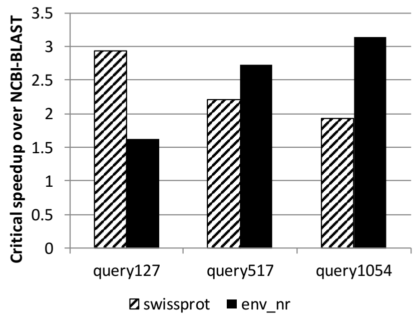
Finally, with respect to GPU-BLASTP, Fig. 6.16(g) shows that on the *swissprot* and *env_nr* database, cuBLASTP achieves up to 1.5-fold and 1.6-fold speedups for the critical phases. Fig. 6.16(h)



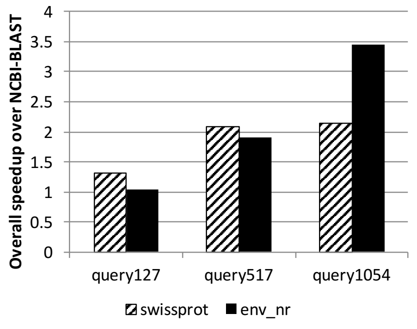
(a)



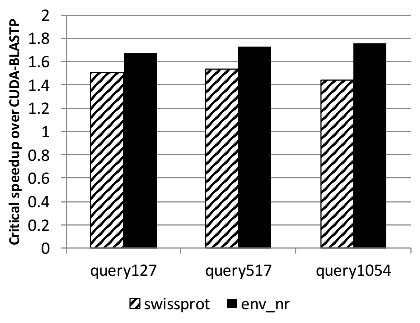
(b)



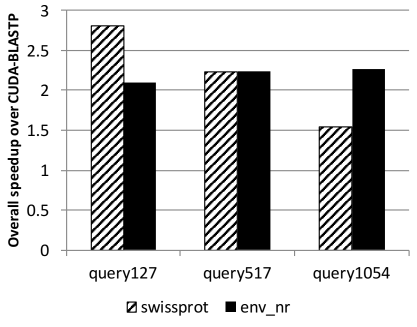
(c)



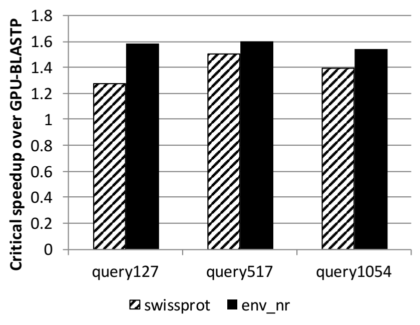
(d)



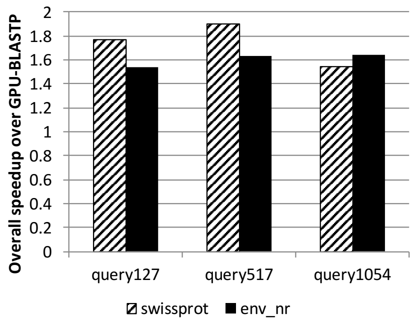
(e)



(f)



(g)



(h)

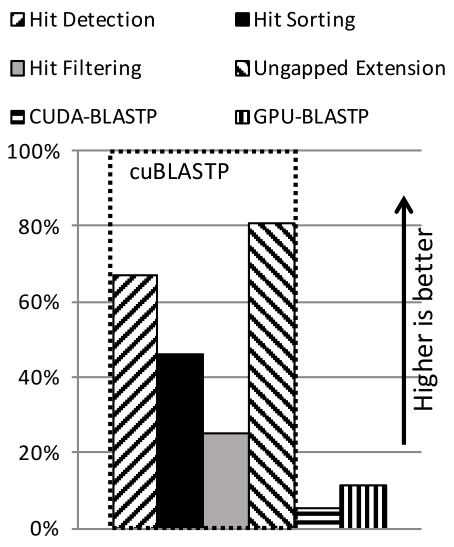
Figure 6.16: Speedup for critical phases and overall performance respectively of cuBLASTP over FSA-BLAST(a-b), NCBI-BLAST with four threads(c-d), CUDA-BLASTP(e-f) and GPU-BLASTP(g-h)

shows that for the overall performance, the corresponding performance improvements using cuBLASTP are 1.9-fold and 1.6-fold, respectively.

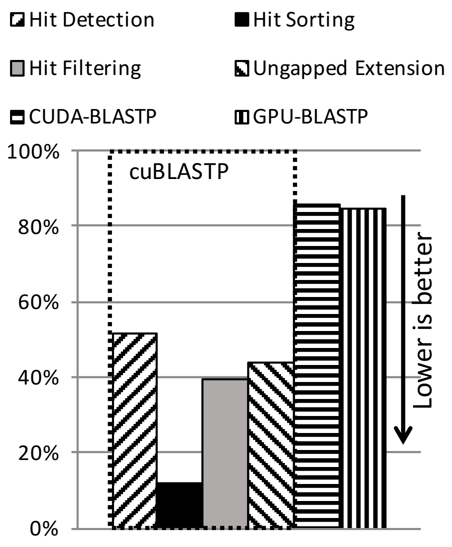
Fig. 6.17(a), 6.17(b), and 6.17(c) show the profiling results of global memory load efficiency, divergence overhead, and occupancy, achieved for cuBLASTP, CUDA-BLASTP, and GPU-BLASTP on the NVIDIA Kepler K20c GPU. Because we observed similar results on other query sequences, we only report the results of "query517" for the *env_nr* database.

Fig. 6.17(a) shows 67.0%, 46.2%, 25.0%, and 81.0% global memory load efficiency for the four respective kernels of cuBLASTP; and only 5.2% for CUDA-BLASTP and 11.5% for GPU-BLASTP, both of them use a single coarse-grained kernel, where both hit detection and ungapped extension are interleaved together. The significantly improved efficiency of our fine-grained kernels comes from the coalesced memory access. In the hit detection, threads in the same warp access positions of subject sequences successively. In sorting and filtering, threads in the same warp access hits in each bin successively; and in the window-based ungapped extension, the window-sized set of threads can access successive positions for one hit to calculate the prefix sum and check the score change. In contrast, neither of the coarse-grained kernels of CUDA-BLASTP or GPU-BLASTP can guarantee such coalesced memory accesses.

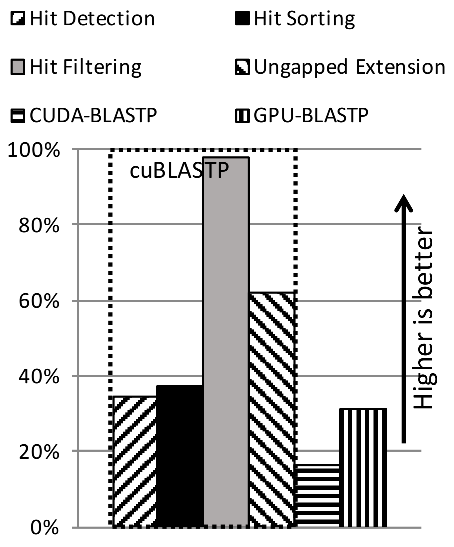
Fig. 6.17(b) and 6.17(c) present the divergence overhead and GPU occupancy, respectively. Our four kernels of cuBLASTP exhibit much lower divergence overhead and higher GPU occupancy than the fused kernels in CUDA-BLASTP and GPU-BLASTP. Fig. 6.20 shows the breakdown of the overall execution time when aligning "query517" on *env_nr* database with cuBLASTP. Although the data transfer between CPU and GPU, and the gapped extension on CPU have the non-



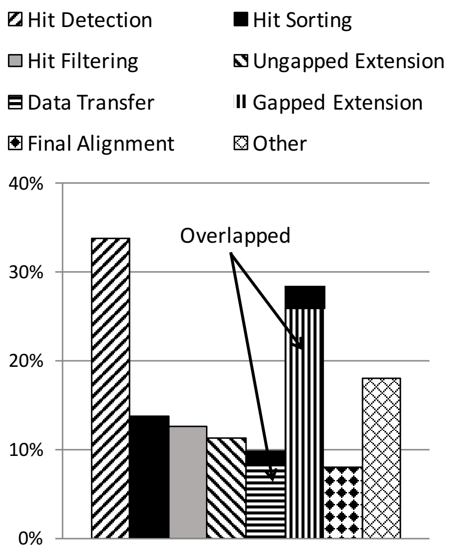
(a) Global memory load efficiency



(b) Divergence overhead



(c) Occupancy achieved



(d) cuBLASTP breakdown

Figure 6.17: Profiling on cuBLASTP, CUDA-BLASTP, and GPU-BLASTP

negligible execution time, we can overlap them with the kernels running on GPU, as shown in the shadowed bars of this figure. We also find after we optimize all stages of BLASTP on GPU and CPU, the remaining part of BLASTP, denoted as "Other" in this figure, can occupy near 18% total execution time. This part includes the time spent on the database read, the DFA and PSS matrix build, and the final results output. We will further investigate the time spent on this part when we extend our research to GPU clusters in the future. Finally, we would like to mention that the output of cuBLASTP is *identical* to the output of FSA-BLAST.

6.1.5 Conclusion

In this chapter, we propose cuBLASTP, an efficient fine-grained BLASTP for GPU using the CUDA programming model. We decompose the hit detection and ungapped extension into separate phases and use different GPU kernels to speed up their performance. To significantly reduce the branch divergence and irregular memory access, we propose binning-sorting-filtering optimizations to reorder memory accesses in the BLASTP algorithm. Our algorithms for diagonal-based and hit-based ungapped extension further reduce branch divergence and improve performance. Finally, we also propose a hierarchical buffering mechanism for the core data structures, which takes advantage of the latest NVIDIA Kepler architecture.

We optimize the remaining phases of cuBLASTP on a multi-core CPU with pthreads. On a compute node with a quad-core Intel Sandy Bridge CPU and a NVIDIA Kepler GPU, cuBLASTP achieves up to a 7.9-fold and 3.1-fold speedup over single-threaded FSA-BLAST and multithreaded

NCBI-BLAST with four threads for the critical phases of cuBLASTP, namely hit detection and ungapped extension, and up to a 6-fold and 3.4-fold speedup for the overall performance, respectively. Compared with CUDA-BLASTP, cuBLASTP delivers up to a 2.9-fold and 2.8-fold speedup for the critical phases of cuBLASTP and for the overall performance, respectively. Finally, compared with GPU-BLASTP, cuBLASTP delivers up to a 1.6-fold and 1.9-fold speedup for the critical phases of cuBLASTP and for the overall performance, respectively.

In summary, our research with cuBLASTP consists of a novel fine-grained method for optimizing a critical life sciences application that has irregular memory-access patterns and irregular execution paths on a single compute node having CPU and GPU.

6.2 Adaptive Dynamic Parallelism for Irregular Applications on GPUs

6.2.1 Introduction

General-purpose graphics processing units (GPGPUs) are widely used to accelerate a variety of applications in different domains. Since GPUs are ideally suited to applications with regular computations and memory access patterns, it is challenging to map irregular applications, e.g., graph algorithms, sparse linear algebra, mesh refinement applications, etc. on a GPU. Dynamic parallelism, supported by both CUDA [34] and OpenCL [35], allows a GPU kernel to directly launch other GPU kernels from the device and without the involvement of the CPU. This feature can potentially improve the performance of irregular applications by reducing workload imbalance between threads, thereby improving both parallelism and memory utilization [39]. For example, during the kernel execution, if some GPU threads have more work than others, new child kernels can be spawned to process these subtasks from the overloaded threads. However, the efficiency of dynamic parallelism is limited by two issues: 1) the high overhead of kernel launch, especially when a large number of child kernels are needed for subtasks; and 2) the low occupancy, especially when the subtasks correspond to tiny kernels that underutilize the computational resources of GPUs.

To address these two issues in dynamic parallelism, multiple solutions [83, 84, 85, 88, 89, 172] have been proposed in both hardware and software. They mainly use the techniques of subtask aggregation, which consolidates small child kernels into larger kernels, hence reducing the number of kernels and increasing the GPU occupancy. However, when the kernel launch overhead has been progressively reduced on the latest GPU architectures, the “one-size-fits-all” approach in the existing studies, where subtasks are aggregated into a single kernel, cannot provide good performance, because those subtasks launched by dynamic parallelism usually require different optimizations and configurations. As a consequence, the organization of subtasks to child kernels becomes more critical to the overall performance, and an adaptive strategy of subtask aggregation that provides differentiated optimizations for subtasks with different characteristics may satisfy dynamic parallelism on the latest GPUs.

However, it is non-trivial to determine the optimal aggregation strategy for subtasks at runtime, because there are many performance factors to be considered, especially the characteristics of subtasks and GPU architectures. To provide a simple system-level solution, in this paper, we propose a performance modeling and task scheduling tool for subtasks in dynamic parallelism to generate the optimal aggregation strategy. Our tool collects the values of a set of GPU performance counters with sampling data and then leverages a couple of statistical and machine learning tools to build the performance model step by step. At the performance analysis phase, we use the statistical analysis on GPU performance counters to identify the most influential performance factors, which can give us the hints of performance critical characteristics and performance bottleneck of subtasks. At the performance prediction phase, based on the most influential performance factors, we establish a performance prediction model to estimate the performance of new subtasks. At the task scheduling phase, the adaptive subtask aggregation strategy launches a set of GPU kernels for subtasks considering the resource utilization, aggregation overhead, and kernel launch overhead. Comparing to the “1-to-1” launching in the default implementations of dynamic parallelism, where one subtask is scheduled to one child kernel, and the “N-to-1” launching in the previous research, where all subtasks are scheduled to an execution entity, e.g., all subtasks to a kernel or thread block or thread warp, our “N-to-M” launching mechanism can provide the most adaptability and fully utilize GPU resources. Our paper has the following contributions:

- We perform an in-depth characterization of existing subtask aggregation approaches for dynamic parallelism on the latest GPU architectures and identify the performance issues.
- We propose a performance model to identify the most critical performance factors and characteristics of subtasks that affect the performance and configurations of subtasks, and predict the performance of new tasks. Based on the prediction model, we design a subtask aggregation model based on the performance model to generate the optimal subtask aggregation strategy.
- In the experiments, we show the accuracy of our performance model by evaluating it with different irregular programs and datasets. Evaluation results show that the optimal aggregation strategy can achieve up to a 1.8-fold speedup over the state-of-the-art subtask aggregation approach.

6.2.2 Background

In this section, we will first introduce the performance counters used in this work, and then briefly introduce the statistical and machine learning techniques for performance modeling and prediction.

6.2.2.1 Performance Counters (PCs)

The hardware performance counters (PCs), which are special-purpose registers built into modern micro-architectures to record the counts of hardware-related events, can help us to perform low-level performance analysis and tuning. In particular, by tracing these PCs, programmers can obtain the correlation between programs and their performance.

Both AMD and NVIDIA provide profiling tools and APIs to access these performance counters. Table 6.1 shows an example of performance counters of NVIDIA GPUs. In this paper, we will utilize these performance counters to establish performance models for performance analysis and prediction.

6.2.2.2 Statistical and Machine Learning Model

In this section, we provide the background of the statistical and machine learning tools that will be used in this paper.

Regression trees and forest Tree-based regression models [173] provide an alternative to the classic linear regression model. It builds decision trees with training datasets and generates the

Table 6.1: Performance counters (NVIDIA Pascal GPU)

Performance Counter	**Description**
warp_execution_efficiency	Ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor expressed as percentage
inst_replay_overhead	Average number of replays for each instruction executed
global_hit_rate	Hit rate for global loads
gld/gst_efficiency	Ratio of requested global memory load/store throughput to required global memory load throughput expressed as percentage
gld/gst_throughput	Global memory load/store throughput
gld/gst_requested_throughput	Requested global memory load/store throughput
tex_cache_hit_rate	Texture cache hit rate
l2_read/write_throughput	Memory read/write throughput seen at L2 cache for all write requests
l2_tex_read/write_hit_rate	Hit rate at L2 cache for all read/write requests from texture cache.
issue_slot_utilization	Percentage of issue slots that issued at least one instruction, averaged across all cycles
ldst_issued/executed	Number of issued/executed local, global, shared and texture memory load and store instructions
stall_not_selected	Percentage of stalls occurring because warp was not selected
issued/executed_ipc	Instructions issued/executed per cycle

classification or regression of the individual trees. Random decision forests (Random Forest) [174] is a popular regression tree model that selects features randomly to avoid the over-fitting issues in decision trees.

Principle Component Analysis Principal component analysis (PCA) [173] is a statistical tool to reduce the number of dimensions by converting a large set of correlated variables into a small set of uncorrelated variables i.e., principal components, where most of the information still remain in the large set. PCA is a technique used to identify the important variables and patterns in a dataset.

Hierarchical Cluster Analysis Hierarchical Cluster Analysis (HCA) [173] is a statistical and data mining tool that builds a hierarchy of clusters for cluster analysis. It provides a measure of correlation between sets of observations. Typically, this is achieved by use of an appropriate metric (such as distance matrices), and a linkage criterion which represents the similarity of sets with the pairwise distances of observations in the sets.

6.2.3 Problems of Dynamic Parallelism

In this paper, we carry out the performance characterization of existing dynamic parallelism approaches to identify their performance issues.

6.2.3.1 Experimental Setup

In this section, we present our experimental setup, including benchmarks, hardware platforms, and software environments.

Benchmark Implementations To identify the performance issues in dynamic parallelism, we choose three typical irregular applications, including *Sparse-Matrix Vector Multiplication* (SpMV), *Single Source Shortest Path* (SSSP), and *Graph Coloring* (GCLR). For each application, we first provide the basic dynamic parallelism implementation that spawns a child kernel per subtask from a thread (Figure 2.8). And then according to the recent publications [83, 90], we build the state-of-the-art subtask aggregation approach that consolidates as many as possible subtasks into a GPU kernel to minimize the kernel launch overhead and improve occupancy. As shown in Figure 6.18, the parent kernel stores all subtasks into a global queue (Line 7), and launches a child kernel for all subtasks, and a subtask is processed by a workgroup for the AMD GPU (or one thread block for the NVIDIA GPU) (Line 17), which was reported to be the best configuration for the graph and sparse linear algebra algorithms.

Dataset Each application has three datasets from the DIMACS challenges [175]: *coPapers*, which has 434,102 nodes and 16,036,720 edges, *kron-logn16*, which has 65,536 nodes and 4,912,142 edges, and *kron-logn20*, which has 1,048,576 nodes and 89,238,804 edges.

```
__global int num_subtasks = 0;
__kernel parent_kernel(type *queue, ...) {
  int tid = get_global_id(0);
  type *this_subtask = subtasks[tid];
  if(this_subtask->size >= THRESHOLD){
    int pos = atomic_add(&num_subtasks, 1);
    queue[pos] = this_subtask;
  }
  else{
    process(this_subtask);
  }
  __global_sync();
  if(tid==0)
    kernel_launch(process_subtasks, queues);
}

__kernel process_subtasks(type *queue, ...) {
   int wg_id = get_group_id(0);
   type *this_subtask = queue[wg_id];
   // process this_subtask
}
```

Figure 6.18: Example of state-of-the-art subtask aggregation

Hardware We evaluate state-of-the-art dynamic parallelism implementations on the latest AMD and NVIDIA GPU architectures. For the AMD GPU platform, called *Vega*, the compute node consists of two Intel Broadwell CPUs (E5-2637v4), and an AMD Radeon RX Vega 64 GPU (AMD Vega architecture). For the NVIDIA GPU platform, called *P100*, the compute node has two Broadwell CPUs (E5-2680v4), and an NVIDIA Tesla P100 GPU (NVIDIA Pascal architecture).

Compiler Each application has OpenCL and CUDA versions for AMD and NVIDIA platform, respectively. OpenCL kernels are compiled and executed with the ROCM 1.6 and ATMI v0.3 on the AMD platfrom. CUDA kernels are compiled and executed with NVIDIA CUDA 8.0 on the NVIDIA platform. CPU-side codes are compiled with GCC 4.7.8.

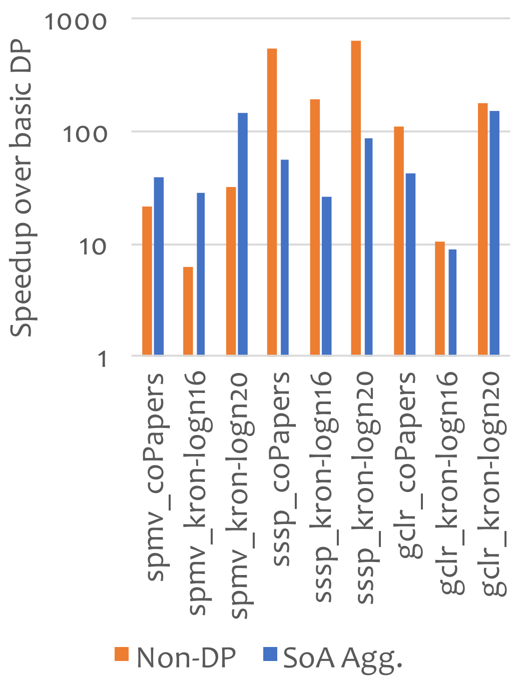
Profilers To get in-depth performance analysis, we use the profilers provided by NVIDIA and AMD to get the performance counters of GPUs. On the NVIDIA platform, we use *nvprof* from CUDA 8.0. On the AMD platform, we use *CodeXL* of version 2.5.

6.2.3.2 Performance Analysis

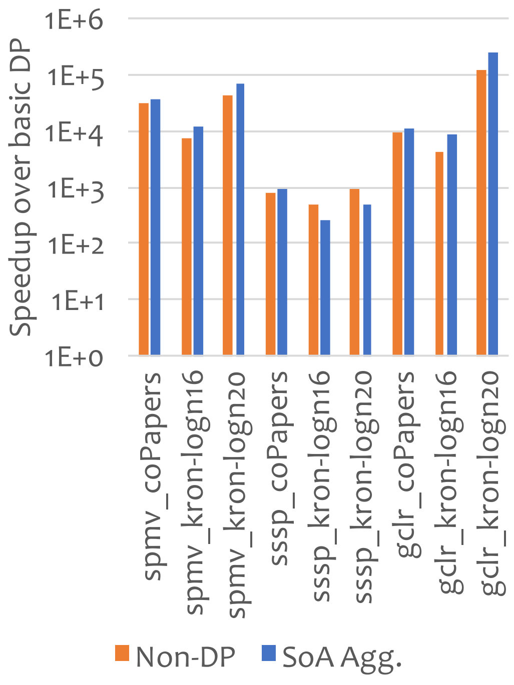
To identify the performance issues in existing approaches, we perform in-depth performance analysis on our driving applications without dynamic parallelism, implementations with the dynamic parallelism, and the dynamic parallelism with state-of-the-art subtask aggregation.

Figure 6.19 illustrates the bad performance of the default dynamic parallelism, compared with the implementations without dynamic parallelism which workload imbalance. This figure also illustrates a huge improvement of using the state-of-the-art subtask aggregation mechanism over the default dynamic parallelism implementation. Moreover, with better workload balance and improved memory access patterns, the state-of-the-art subtask aggregation can also deliver better performance than the implementation without dynamic parallelism (except SSSP benchmarks on the AMD Vega GPU). And we also observe that there are much higher speedups of using the subtask aggregation over the default dynamic parallelism implementations on the NVIDIA P100 GPU than those on the AMD Vega GPU, which is due to the higher kernel launch overhead on the NVIDIA P100 GPU.

Figure 6.20 shows the normalized execution time of child kernels, including kernel launch time and kernel compute time. We can find that with the subtask aggregation mechanism, the kernel



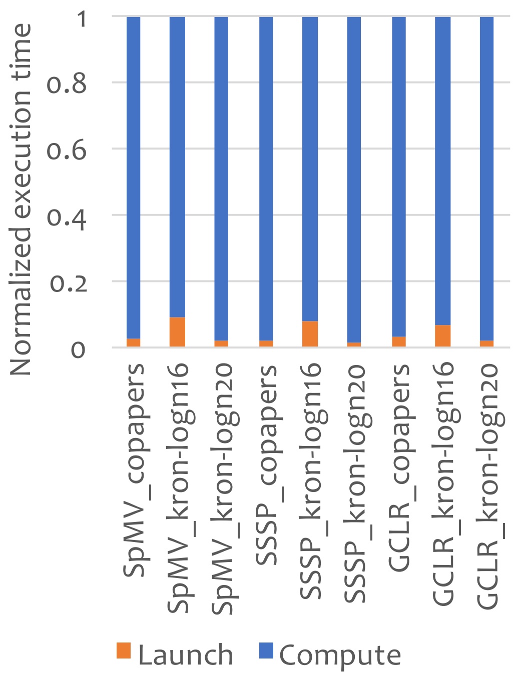
(a) AMD Vega GPU



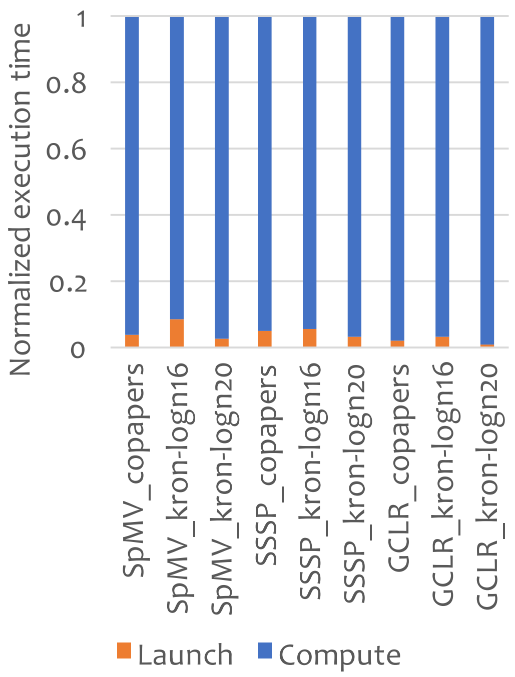
(b) NVIDIA P100 GPU

Figure 6.19: Speedups of the implementations without dynamic parallelism (Non-DP) and the implementations with state-of-the-art subtask aggregation (SoA Agg.) over the default dynamic parallelism implementations (Basic DP).

launch overhead is significantly reduced to be negligible and most of the execution time is spent on the computation of subtasks, especially for large datasets (i.e., *kron-logn20*). Thus, if one wants to improve the overall performance of dynamic parallelism, improving the performance of subtasks is more critical than reducing the launch overhead of child kernels. This is the major reason of why we investigate an adaptive strategy for the subtask aggregation.



(a) AMD Vega GPU

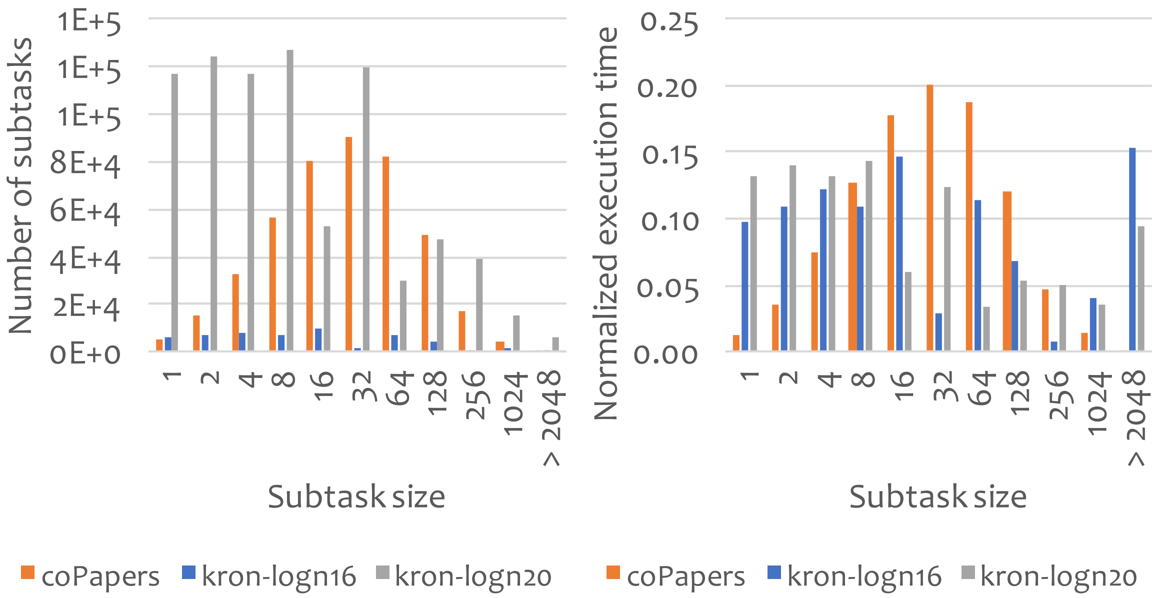


(b) NVIDIA P100 GPU

Figure 6.20: Breakdown of the child kernel execution time in state-of-the art subtask aggregation mechanism (SoA Agg.).

Although the subtask aggregation mechanisms can significantly improve the overall performance of dynamic parallelism, with a deeper investigation, we find that there is a major drawback in existing subtask aggregation mechanisms: they treat all subtasks equally, by using the "one-size-fits-all" methodology, so called "N-to-1" approach, to aggressively aggregate as many as possible subtasks into a single kernel and apply the uniform configuration and parallel strategy for all subtasks. However, we have observed there are highly diverse characteristics in subtasks. Figure 6.21(a) shows that in the SpMV benchmark, the subtask sizes, i.e., corresponding numbers of GPU threads in subtasks, can range from 1 to over 2K; and the distribution of subtask sizes highly depends on the input datasets. We also find although most of the subtasks fall into the range from 1 to 256 in this case, the execution time of large subtasks, i.e., the subtask size > 2048, can take a considerable portion of the total execution time, as shown in Figure 6.21(b).

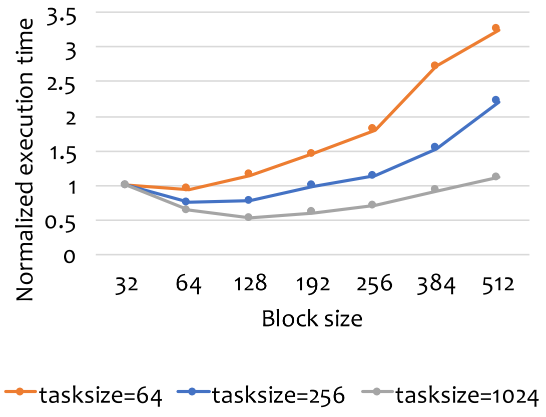
As a result, we carry out a simple evaluation to investigate if we can find different performance when we vary the resource usage, i.e., changing GPU thread block sizes, for different subtask sizes.



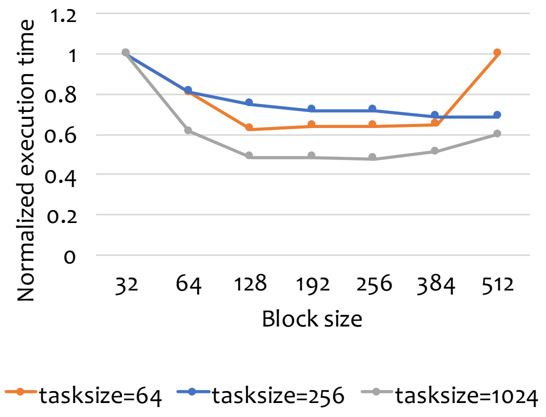
(a) Subtask size (b) Execution time

Figure 6.21: The distribution of subtask size and execution time of SpMV benchmarks. The execution time of each subtask size is normalized to the total execution time.

Figure 6.22 shows that for the subtasks of size 64, 256, and 1024, their performances have obviously affected by the thread block size; and the optimal thread block size is variable with the subtask size and benchmark. The major reason is that when we change the thread block size for a given benchmark, each thread has different workloads and uses different hardware resources, e.g., GPU registers, shared memory, leading to changes in parallelism, occupancy, and resource utilization. As a consequence, the “one-size-fits-all” approach in existing approaches may result in resource underutilization. A more intelligent subtask aggregation strategy is needed.



(a) SpMV



(b) SSSP

Figure 6.22: Performance of SpMV and SSSP subtask with different block size. The execution time is normalized to that of block size = 32.

6.2.4 Adaptive Subtask Aggregation

To obtain the optimal task aggregation strategy for dynamic parallelism, we propose a task aggregation modeling and task scheduling tool that uses statistical analysis and machine learning techniques to establish performance models based on a collection of performance counters with sampling data.

Figure 6.23 shows the high-level depiction of our tool, which consists of four phases: 1) *per-*

formance measurement phase, which collects performance counter data with the sampling data from different input datasets and parameters; 2) *performance modeling* phase, which establishes a performance model for the performance analysis, i.e., determining most important performance counters and subtask characteristics; 3) *performance prediction* phase, which uses the identified important performance counters and characteristics to build a performance prediction model; 4) *aggregation generation* phase, which generates the optimal subtask aggregation strategy based on the performance model by considering subtask performance gain and loss, aggregation overhead, kernel launch overhead, etc. Below we will discuss each phase in details.

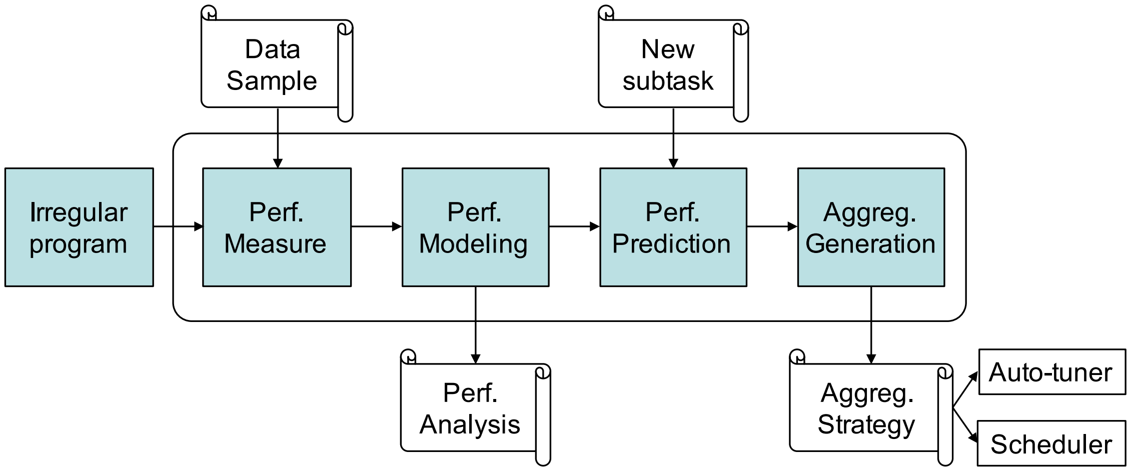


Figure 6.23: Architecture of the adaptive subtask aggregation tool

6.2.4.1 Performance Measurement

Performance Measurement phase is responsible for collecting performance counter data of the irregular program on the target architecture. Since the collection of performance counter data can significantly affect the accuracy of the performance models in the later stages, we carry out the performance measurement by running the subtasks with varying parameters, including different datasets, subtask sizes, and runtime configurations (i.e., thread block (workgroup) size and the number of thread blocks). During the performance measurement, our tool collects performance counter data, and measures the execution time as the response variable. Performance counter data are collected using corresponding performance profilers - *CodeXL* and *nvprof* for AMD and NVIDIA platforms, respectively.

The size and selection of sample data are critical for the accuracy of the performance model. Though more sample data can improve the accuracy of the performance model, over-saturated sample data will significantly increase the data collection time and performance modeling overhead. Moreover, to avoid selection bias, which makes the model is non-representative for new subtasks with unseen characteristics, the data selection should have proper randomization. Therefore, we randomly collect 200 samples with different input parameters and configurations, which are sufficient to build accurate performance models for predicting the optimal aggregation strategy. As a configurable parameter, the number of samples can also be set by users.

6.2.4.2 Performance Modeling

In performance modeling phase, to identify the most important performance factors, we utilize couples of statistical and machine learning tools, including Principal Components Analysis (PCA), Random Forest Regression (RF) and Hierarchical Cluster Analysis (HCA).

Principal Components Analysis (PCA) We first perform PCA analysis on performance counter data, which can help us to identify important performance counters that contribute most to the variance, and also can help us to determine the correlation between these performance counters. Based the importance and correlation, we can reduce the number of performance counters for the later performance modeling to reduce the risk of over-fitting. In this paper, we identify first few important performance counters (< 10) from the top principal components as important variables.

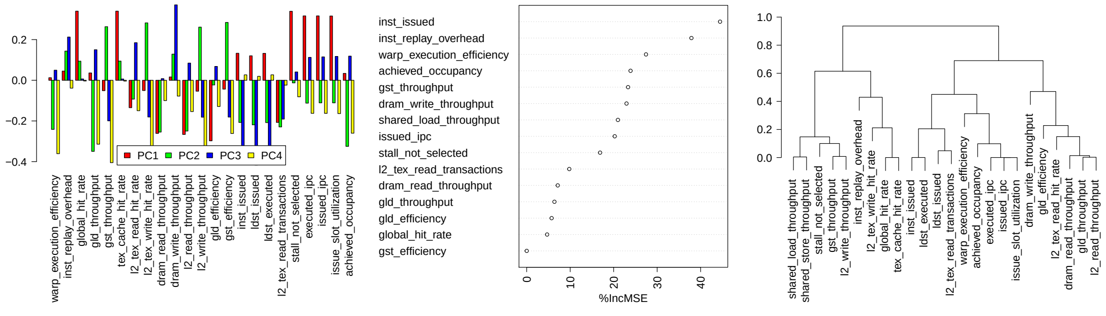
Random Forest Regression After PCA analysis, we apply the Random Forest (RF) model on the performance counter data, and obtain the relative variable importance of RF, which can reveal the influence of a variable to the response variable, i.e., execution time. Through identifying the most important variables (i.e., performance counters), we can determine the performance counters that are strongly correlated to the execution time, which give us hints of the characteristics and performance bottlenecks of subtasks.

Hierarchical Cluster Analysis (HCA) After the Random Forest, we use Hierarchical Clustering Analysis (HCA) to help us get insights of the important performance counters determined by the Random Forest, which can give us hints about the characteristics and performance bottlenecks of subtasks.

Result Analysis In this section, we offer examples of performance analysis with the performance modeling phase.

SpMV Figure 6.24 shows the result of performance modeling of SpMV benchmarks. Based on the PCA results (Figure 6.24(a)), we can identify the most variable performance counters from the top four principle components - PC1, PC2, PC3 and PC4, which account for the most of the variance in the performance counter data. The most variable performance counters are *global_hit_rate*, *tex_cache_hit_rate*, *gld_throughput*, *achieved_occupancy*, *ldst_issued*, *ldst_executed*, *l2_write_throughput* and *gst_throughput*.

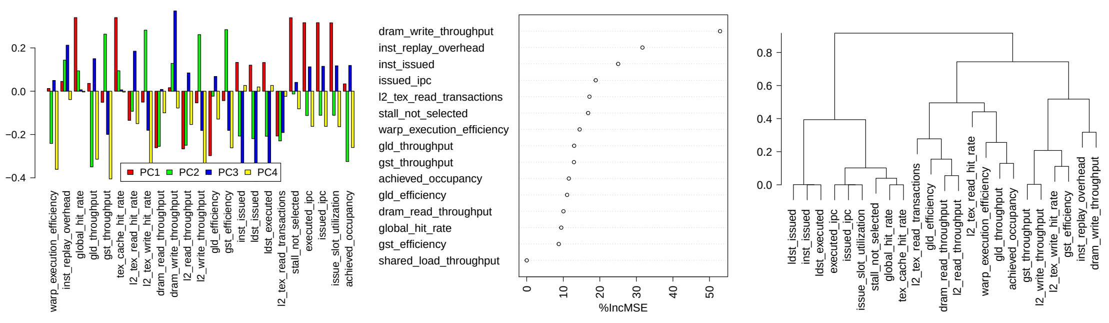
After identifying the most variable performance counters, the Random Forest will be applied to the performance counter data with execution time as response variable to determine the most important performance counters relevant to performance. Figure 6.24(b) shows the *inst_issued* and *inst_replay_overhead* are the two most important performance counters for SpMV benchmarks. Then, we can turn to HCA to get more insights. We can observe that the most relevant performance counter (i.e., *inst_issued*) has strong correlation to *inst_issued* and *ldst_executed*. And the second important performance counter -*inst_replay_overhead* has strong correlation to *global_hit_rate*, *tex_cache_hit_rate* and *l2_tex_write_hit_rate*. It indicates that data locality and the amount of workload have significant impacts on the performance of SpMV.



(a) Factor loadings (b) Variable importance (c) Clustering

Figure 6.24: The result of the performance modeling of SpMV benchmark.

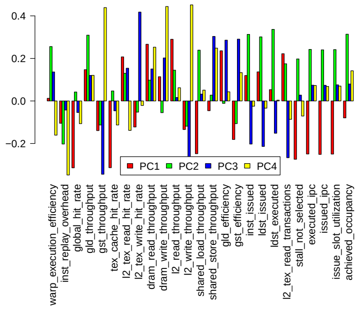
SSSP From the results for SSSP (Figure 6.25(a)), we can observe that the SSSP benchmarks has highly similar PCA results as SpMV benchmarks. However, Figure 6.25(b) shows the most important variable for the time prediction is *dram_write_throughput*. From Figure 6.25(c), we observe that *dram_write_throughput* is strongly connected to *inst_replay_overhead*, *gst_efficiency*, *l2_tex_write_hit_rate* and *gst_throughput*. It can give us a hint that the performance of SSSP is highly relevant to the memory write performance.



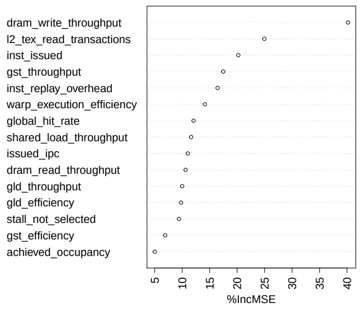
(a) Factor loadings (b) Variable importance (c) Clustering

Figure 6.25: The result of the performance modeling of SSSP benchmark.

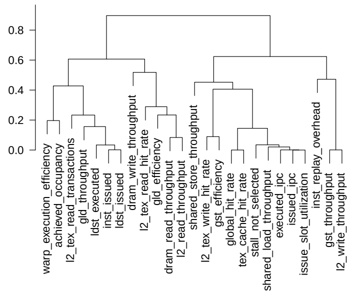
Graph Coloring Figure 6.26 shows the performance modeling result of graph coloring benchmarks. Similar with SSSP benchmarks, *dram_write_throughput* is the most important performance counters for performance prediction. Based on the result of HCA (Figure 6.26(c)), there are high correlation among *dram_write_throughput*, *l2_tex_read_hit_rate*, *gld_efficiency*, and *dram_read_throughput*, which indicate that the performance of graph coloring is highly relevant to the memory read performance.



(a) Factor loadings



(b) Variable importance



(c) Clustering

Figure 6.26: The result of the performance modeling of Graph Coloring (GCLR) benchmark.

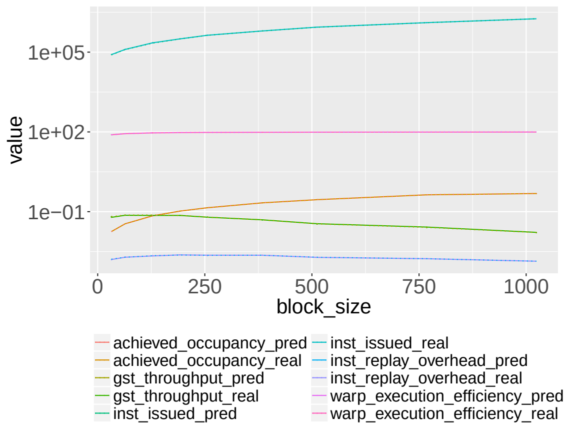
6.2.4.3 Performance Prediction

Based on the performance modeling phase, we can establish a prediction model to estimate the performance of new subtasks.

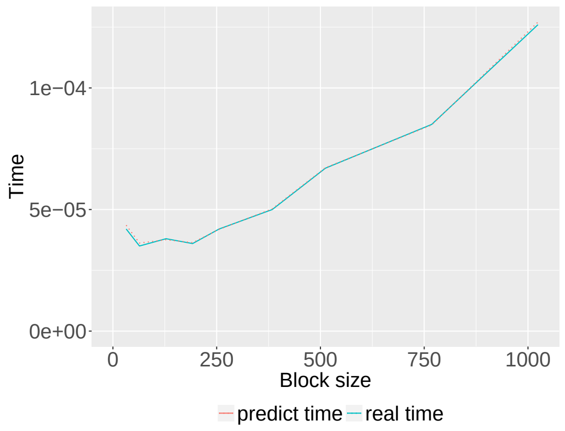
Prediction Model with Random Forest Our general idea for the performance prediction is (1) building dedicated prediction models for each top important performance counters (≤ 5) based on thread blocks size, subtask size, and the number of subtasks, (2) using the top important performance counters to retrain the prediction model for the execution time, (3) and merging the two sets of predict models to predict the execution time for new subtasks with the given subtask size and the number of tasks. With this performance prediction model, we can predict the optimal thread block size for new subtasks. And then, we can perform the initial subtask aggregation that groups subtasks with the same thread block size together, and set the optimal thread block size for each group. But, to achieve the optimal overall performance, we need a more sophisticated subtask aggregation strategy, which will be discussed in the following Section 6.2.4.4.

Result Analysis Figure 6.27 shows an example of performance prediction model for SpMV subtasks with task size = 128, the number of tasks = 64, and varied thread block size. To verify the accuracy of our model, we randomly select 80% of data as training data and use the rest 20% of data as evaluation data.

As Figure 6.27(a) shown, we first predict the top five important performance counters. And then, as Figure 6.27(b) illustrated, we use the predicted value of the top 5 performance counters to estimate the performance of the given SpMV subtasks with varying thread block size.



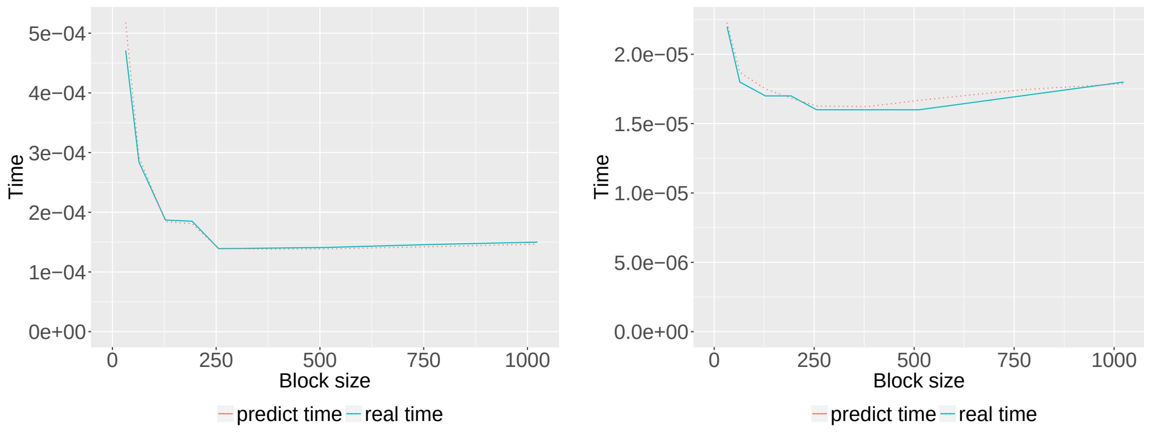
(a) Performance counter prediction



(b) Performance prediction

Figure 6.27: The result of the performance prediction of SpMV benchmark with *kron-logn16* dataset, task size = 128 and number of tasks = 64.

Figure 6.28 shows the prediction results of SSSP and GCLR benchmarks. In general, we get produce highly accurate performance prediction for SpMV and SSSP benchmarks, and GCLR benchmarks.



(a) SSSP (b) GCRL

Figure 6.28: The result of the performance prediction of SSSP and GCLR benchmark *kron-logn16* dataset, task size = 128 and number of tasks = 64.

6.2.4.4 Aggregation Generation

With the performance prediction model, we can easily determine the optimal configurations (i.e., block size) for new subtasks through searching the configuration space. However, generating the optimal aggregation strategy is not trivial, which has the following challenges.

First, despite the fact that kernel launch time has been continually reduced in the latest GPU micro-architecture and runtime, the kernel launch is not free. Second, the aggregation will reduce kernel launch overhead, but aggregating subtasks with different configurations will result in performance loss by applying non-optimal configuration on subtasks. Third, subtask aggregation also introduces aggregation overhead, e.g., migrating subtasks into the same subtask group. Therefore, we need to balance the kernel launch overhead, aggregation overhead, and subtask performance.

To achieve the optimal overall performance, we build a model that firstly identifies the optimal performance for each subtask (Equation 6.1), and then searches the subtasks, which require the identical or similar configuration (i.e., block size), into a kernel, and uses a subtask aggregation model to estimate the optimal performance after merging subtasks, and then determines if we need to merge the two subtask groups. We estimate the optimal task time by applying the configuration across each other subtask to choose the optimal one (Equation 6.2) for both, and then determine merge or not by considering the kernel launch overhead and aggregation overhead (Equation 6.3). Note that the state-of-the-art aggregation method only considers reducing kernel launch overhead rather than the performance loss of applying the non-optimal configuration.

$$\begin{aligned} conf_A, time_A &= conf\_search(task_A) \\ conf_B, time_B &= conf\_search(task_B) \end{aligned} \tag{6.1}$$

$$time_{child} = min \begin{cases} predict(conf_A, t_B) + time_A \\ predict(conf_B, t_A) + time_B \end{cases} \tag{6.2}$$

$$time_{overall} = min \begin{cases} time_A + time_B \\ time_{child} - time_{kl} + time_{agg} \end{cases} \tag{6.3}$$

6.2.5 Performance Evaluation

In this section, we evaluate the effectiveness of our aggregation model on AMD and NVIDIA GPU platforms with different benchmarks.

6.2.5.1 Performance Comparison between the State-of-the-Art and Optimal Subtask Aggregation

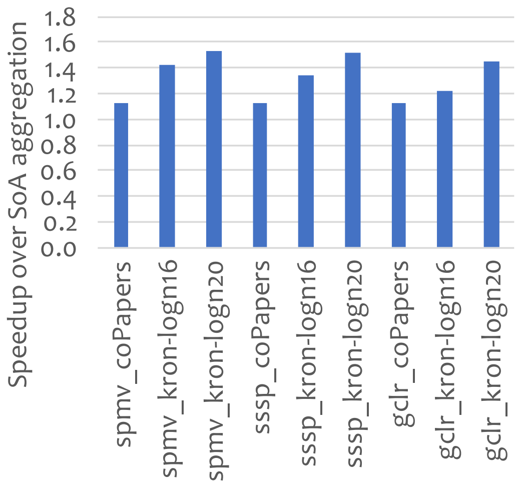
Implementations For each application, we have two implementations:

- **State-of-the-art Subtask Aggregation (SoA Agg.)** is the implementation based current publications that aggressively consolidate as many as possible subtasks into a single kernel.
- **Optimal Subtask Aggregation (Opt. Agg.)** is the implementation based on the optimal subtask aggregation strategy generated by our subtask aggregation model.

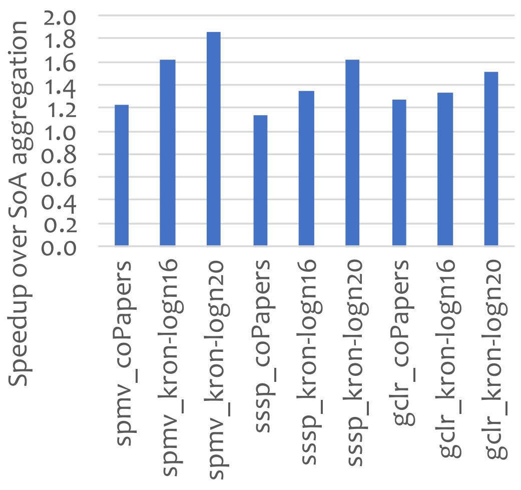
As the performance measurement, performance modeling and performance prediction phase are offline, which just need to run once offline with sampling data for an application. Therefore, in the evaluation, we do not include the execution time of these phases, only include the execution time of the aggregation generation phase, which needs to be performed at runtime.

Performance Figure 6.29 shows the performance comparison between the state-of-the-art aggregation and optimal subtask aggregation. The optimal aggregation strategy can achieve up to a 1.8-fold speedup over the state-of-the-art aggregation on the NVIDIA P100 GPU, and a 1.5-fold speedup on the AMD Vega GPU. For dataset *kron-logn20*, which has higher subtask size diversity, we can achieve higher performance improvement.

Profiling With in-depth profiling, as Figure 6.30 shown, the implementation with the optimal aggregation improves *warp_execution_efficiency* and *global_hit_rate*, which indicates less irregular-



(a) AMD Vega GPU



(b) NVIDIA P100 GPU

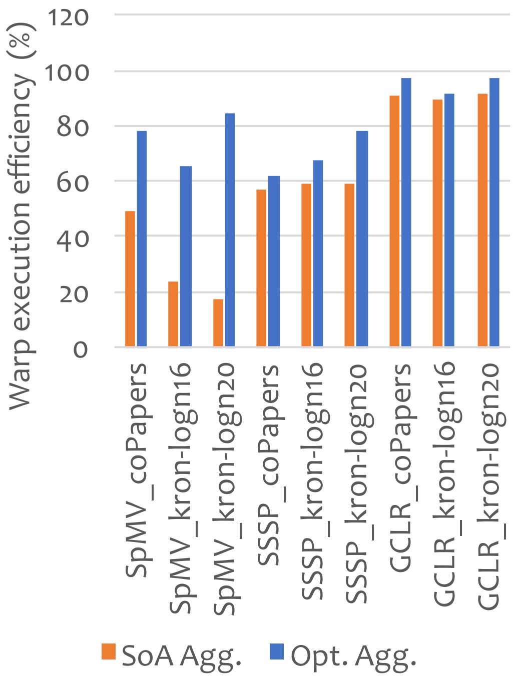
Figure 6.29: Speedup of the optimal subtask aggregation over the state-of-the-art subtask aggregation (SoA Agg.)

ities in control flows and memory accesses. Furthermore, we observe the noticeable improvement in *achieved_occupancy*, which indicates improved resource utilization.

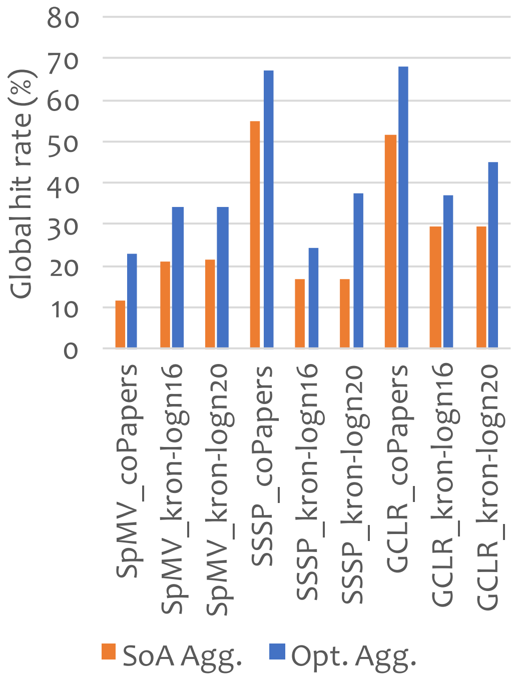
6.2.6 Conclusion

It is challenging to map irregular applications on GPUs due to their irregularities in memory access and computation. Dynamic parallelism supported by AMD and NVIDIA GPUs can potentially improve the performance of irregular applications. However, dynamic parallelism is inefficient on current GPU architectures due to high kernel launch overhead and low occupancy. Therefore, there are multiple studies for improving the efficiency of dynamic parallelism.

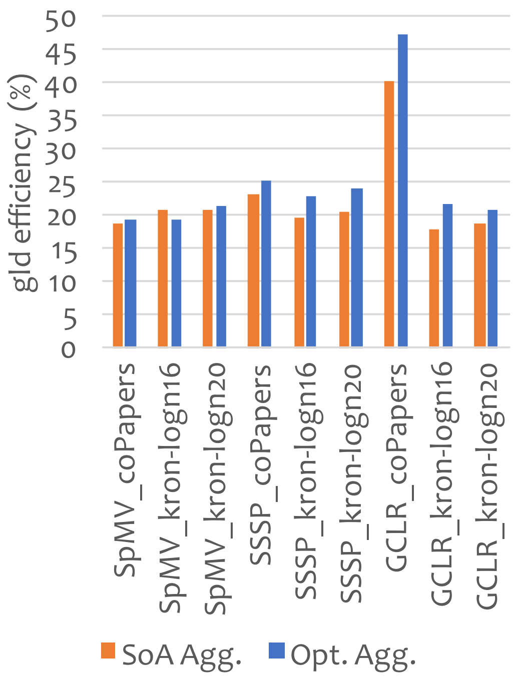
However, with the in-depth performance characterization, existing approaches, which treat all subtasks equally and use uniform configurations, can cause GPU underutilization due to variable char-



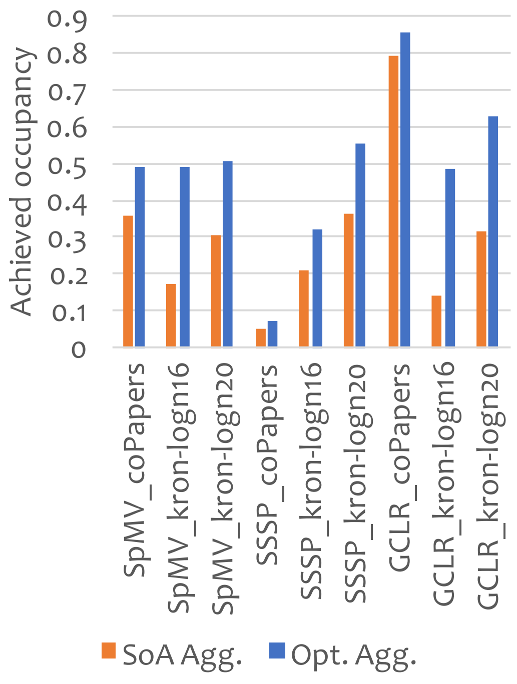
(a) Warp execution efficiency



(b) Global hit rate



(c) Global load efficiency



(d) Achieved occupancy

Figure 6.30: Profiling of the state-of-the-art aggregation (SoA Agg.) and optimal aggregation (Opt. Agg.) on NVIDIA P100 GPUs

acteristics between subtasks. To overcome these drawbacks, we propose a subtask aggregation and scheduling tool that utilizes the statistical and machine learning techniques to establish a set of performance models for performance analysis and prediction of subtasks, and generate the optimal subtask aggregation strategy by considering the subtask performance, kernel launch and aggregation overhead. Experimental results show that the optimal subtask aggregation strategy generated by our tool can achieve up to a 1.8-fold speedup over the state-of-the-art subtask aggregation.

Chapter 7

Optimizing Irregular Applications for Multi-node Systems

7.1 PaPar: A Parallel Data Partitioning Framework for Big Data Applications

7.1.1 Introduction

In the past decade, big data processing systems have been gaining momentum; and scientists have turned to these systems to process large scale and unprecedented data. Most of these systems provide advanced mechanisms to tackle the load imbalance (a.k.a skew), which is a fundamental problem in parallel and distributed systems. For example, MapReduce [21] and its open source implementation Apache Hadoop provides the speculative scheduling to replicate last few tasks of a job on different compute nodes. Many mechanisms, including [22, 23, 24, 25, 26, 27, 28, 29, 176], are also proposed to mitigate skew by optimizing task scheduling, data partitioning, job allocation, etc. Although these runtime methods are able to handle skew to a certain extent without code modification on applications, they can not get the optimal application performance, because the runtime of application not only depends on input data size but also other multiple proprieties. Therefore, many research efforts have been taken to explore the application-specific partitioning methods, including [177, 178, 95, 33, 96]. However, manually writing application-specific partitioning codes requires huge coding efforts. More challenging is the truth that finding the optimal data partitioning strategy is hard even for developers having adequate application knowledge, leading to the iterative and incremental development of design, evaluation, redesign, reevaluation, and so on.

In this research, we target the complexity of developing application-specific data partitioning algorithms and propose *PaPar*, a parallel data partitioning framework for big data applications, to simplify their implementations. We identify a set of common operators used in data partitioning algorithms, e.g., *sort*, *group*, *distribute*, etc., and put them into PaPar as the building blocks. And then we provide a set of interfaces to construct the workflow of partitioning algorithms with these operators. PaPar can parse the configurations of input data types and workflow jobs, and generate the parallel codes after formalizing the workflow as a sequence of key-value operations. Finally, PaPar will map the workflow sequence to the parallel implementations with MPI and MapReduce.

In our evaluation, we use two applications as the case studies to show how to use PaPar to construct user-defined partitioning algorithms. The first driving application is muBLAST [32], an MPI implementation of BLAST for biological sequence search. The second is PowerLyra [33], a computation and partitioning method for skewed graphs. We conduct our experiments on a cluster of 16 compute nodes. Experimental results show that the code generated by PaPar can produce the same partitions as the applications but with less partitioning time. Compared to the multithreaded implementation of muBLASTP partitioning, PaPar can achieve up to 8.6-fold and 20.2-fold speedups for two widely used sequence databases. Compared to the parallel implementation of PowerLyra partitioning, PaPar can also deliver comparable performance for different input graphs.

7.1.2 Background and Motivation

In this section, we first describe our driving applications, summarize the common operators needed in their data partitioning, and then discuss our motivation and design requirements.

7.1.2.1 Driving Applications

muBLASTP: BLAST is a fundamental bioinformatics tool to find the similarity between the query sequence and database sequences. muBLASTP is an MPI implementation of BLAST for protein sequence search. By building the index for each database partition instead of input queries and optimizing the search algorithms with spatial blocking through the memory hierarchy, muBLASTP can achieve better performance than the widely used BLAST implementations, e.g., mpiBLAST [179]. However, the performance of muBLASTP is sensitive to the partitioning methods: because of the nature of heuristics in the search algorithms, the runtime of sequence search depends on the distribution of sequence lengths more than the total size of each partition. The optimized partitioning method [180] tries to satisfy: (1) database partitions have similar numbers of sequences, (2) database partitions have the similar encoded length distribution, and (3) database partitions have the similar total size. Fig. 7.1 illustrates such an implementation. This method manipulates a four-tuple index where each consists of the encoded sequence pointer, the encoded sequence length, the description pointer, and the description length. The partitioning method first sorts the index based on the encoded sequence length and then distributes sequences to different partitions in a cyclic manner.

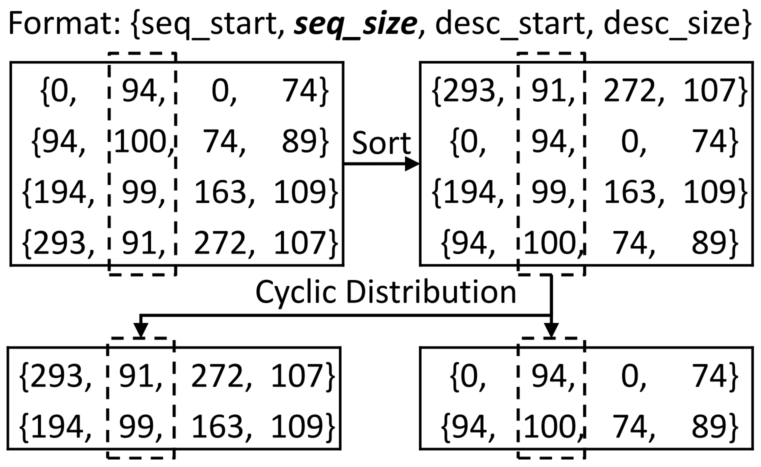


Figure 7.1: The partitioning method in muBLASTP: sort and distribute sequences based on the encoded sequence length.

PowerLyra: PowerLyra is a graph computation and partitioning engine on skew graphs. Other than the vertex-cut and edge-cut partitioning methods, PowerLyra provides a hybrid method to partition graph data. It first does the statistics to generate a user-defined factor, e.g., vertex indegree or outdegree, then splits vertices to a low-cut group and a high-cut group based on this factor, and applies different distribution policies on each group. Integrated with GraphLab [181], PowerLyra can bring significant performance benefits to many graph algorithms, e.g., PageRank, Connected Components, etc. Fig. 7.2 from [33] shows this hybrid-cut method. In this case, PowerLyra uses the vertex indegree to divide the low-cut group and high-cut group with a predefined threshold. For the low-cut group, PowerLyra distributes vertices with all its edges (in-edges) to different partitions; and for the high-cut group, PowerLyra distributes edges of each vertex to different partitions.

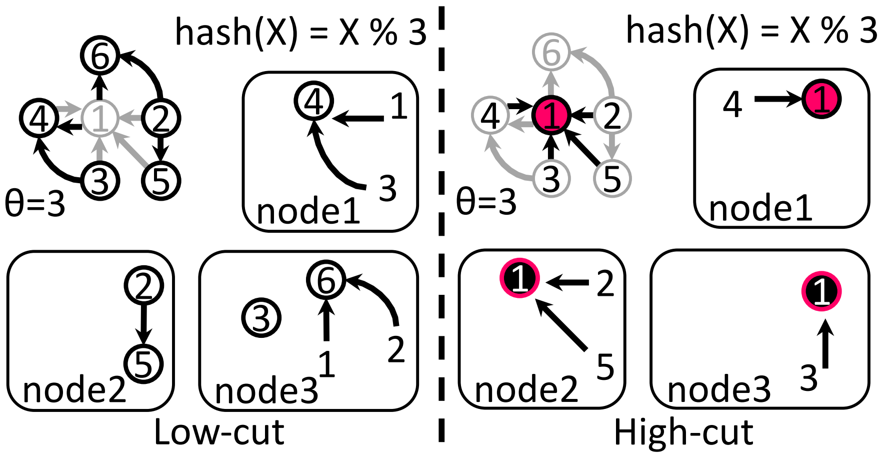


Figure 7.2: The hybrid-cut in PowerLyra: count vertex indegree, split vertices to the low-cut group and high-cut group; for the low-cut, distribute a vertex with all its in-edges to a partition, and for the high-cut, distribute edges of a vertex to different partitions.

7.1.2.2 Motivation

These driving applications illustrate the application-specific methods are necessary for better performance and scalability, even if the underlying systems provide the data partitioning methods. We also observe that there are several common operators used in these two applications to partition data, e.g., the sort operation: muBLASTP needs to sort sequences (as the value) by the encoded sequence length (as the key), and PowerLyra may group the edges belonging to the same in-vertex (as the value) and sort them by the vertex indegree (as the key). Therefore, our motivation is to design a framework to provide such common operators and simplify the implementations of application-specific partitioning algorithms. This task is not straightforward: even if the same operator is needed, the requirements on these operators are very different. For example, for the sort operation, the key and value in muBLASTP can be obtained from input data, i.e., the encoded sequence length and the sequence entry; while in PowerLyra, neither the key (the vertex indegree) nor the value (the grouped edges belonging to the same in-vertex) can be retrieved from input. As a result, the framework must have the capability to concatenate multiple operators, add/delete data attributes, and change data formats on demand. We summarize the design requirements:

- **Correctness:** The framework needs to generate the user-defined partitioning codes. For the same input data, the partitions produced by the framework should be the same to those generated by the original partitioning algorithms.
- **Comprehensiveness:** The framework needs to provide adequate building blocks to construct user-defined partitioning algorithms and be flexible to extend more building blocks as well. Other than the key-value concept for unstructured data, the framework also needs to provide easy to use interfaces to define multiple data types, considering many scientific applications manipulate structured and semi-structured data. Not only processing data from the input file, but the framework also needs to support the in-memory data partitioning, because the intermediate data may require repartitioning and redistribution at runtime.
- **Efficiency:** The generated partitioning codes should be optimized to avoid data partitioning

to be a performance bottleneck. Therefore, the framework needs to adopt the sophisticated techniques from the recent research.

7.1.3 Methodology

Fig. 7.3 shows the high-level architecture of our framework. The user interfaces are two configuration files. One is to describe the input data format, and the other is to describe the data operators in the user-defined partitioning algorithm. By parsing the input data configuration and the workflow configuration, the framework can understand the data structure and set corresponding keys and values for each operator listed in the workflow. Users are allowed to register their own data operator as a new building block by inheriting the *Operator* class and implementing the functionality, which will be discussed in Section 7.1.3.2. The PaPar framework will generate the workflow which will be launched as a sequence of jobs at runtime.

7.1.3.1 Interface for Data Types

To read the structured data, the MapReduce framework, e.g., Hadoop, provides a base class to unify the user interface: users need to implement their own parser for the input data structure by inheriting the Hadoop *InputFormat* class. In this class, users need to implement *getSplits* method to split the input file and generate a list of data blocks, each of which will be assigned to an individual mapper at runtime. Users also need to implement the *getRecordReader* method to extract individual input elements (records) from each split, and set the key and value for the mapper. Although many research projects [182, 183, 184, 185, 186, 187] have leveraged this mechanism to process structured data on MapReduce, we prefer a programming-free method as the interface for user-defined data structures. We provide the *InputData* configuration file to allow users to describe their data structures.

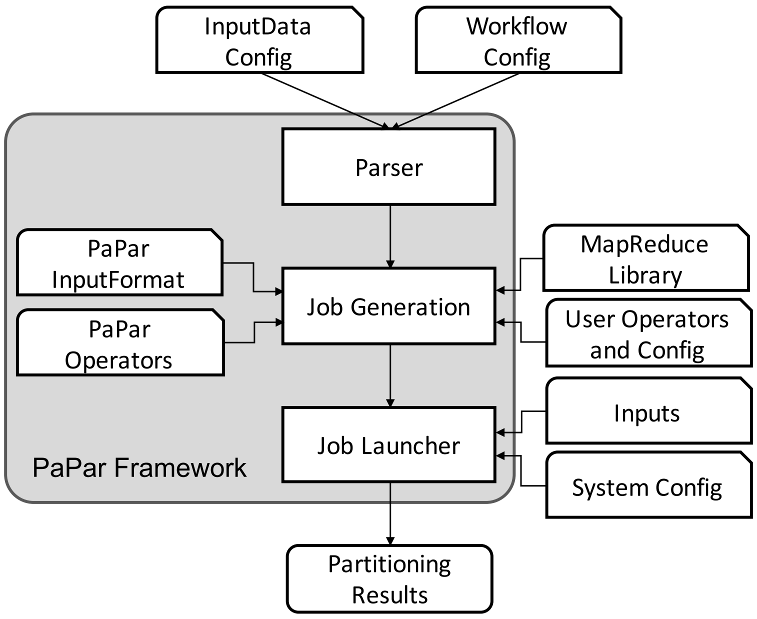


Figure 7.3: The high-level architecture of PaPar framework

Fig. 7.4 shows the example how to describe the BLAST sequence index. The *input_format* and *start_position* sections indicate that BLAST sequence file is a binary file, and the index data starts at 32 bytes. The *element* section describes the index data structure consisting of four 32-bit integers: *seq_start*, *seq_size*, *desc_start* and *desc_size*. According to the configuration file, the parser of PaPar will tell the *InputFormat* class to skip the first 32 bytes of the file, and treat every 16 bytes (4 * 32-bit integers) as an entry. Fig. 7.5 shows the example for the text format used in PowerLyra. The *element* section indicates that each element represents an edge from *vertex_a* to *vertex_b*, separated by the *Tab* character "\t" and ended with the *Enter* character "\n". Similarly, the *InputFormat* class will treat each line in the text file as an entry, and fill two characters from each line to a two-tuple. Note that for derived data types, users may need to declare the nested elements in the configuration file. By providing such a configuration file as an interface, PaPar can support different input data types.

```
<input id="blast_db" name="BLAST Database file">
  <input_format>binary</input_format>
  <start_position>32</start_position>
  <element>
    <value name = "seq_start" type = "integer"/>
    <value name = "Seq_size" type = "integer"/>
    <value name = "desc_start" type = "integer"/>
    <value name = "desc_size" type = "integer"/>
  </element>
</input>
```

Figure 7.4: Data type description for BLAST index

```
<input id="graph_edge" name="edge lists">
  <input_format>text</input_format>
  <element>
    <value name = "vertex_a" type = "String"/>
    <delimiter value="\t"/>
    <value name = "vertex_b" type = "String"/>
    <delimiter value="\n"/>
  </element>
</input>
```

Figure 7.5: Data type description for graph data

7.1.3.2 Operators

We define a set of operators as the building blocks to implement the workflow of desired partitioning algorithms. Users can construct a workflow through the *Workflow* configuration file. For a data partitioning program, we observe that the input and output data formats are usually same, while the formats of intermediate data during partitioning may be different. For example, as discussed in Section 7.1.2.2, the PowerLyra hybrid-cut will count the vertex indegree, which is a new attribute. Based on the behaviors of operators on input data, we define three types of operators. First, the

Basic operators, including *sort*, *distribute*, *split*, *group*, etc., will reorder input data but not add or delete any attribute. For example, the sort operator will move entries from one compute node to another but keep data unchanged. Although multiple basic operators are usually concatenated to construct a workflow, a single basic operator can also be treated as a complete workflow. Second, the *Add-on* operators, e.g., *count*, *max*, *min*, *mean*, *sum*, etc., will add or delete data attributes. Different with the basic operators, the add-on operators themselves can not construct a workflow or a job in the workflow. They need to cooperate with the basic operators. Third, the *Format* operators, e.g., *orig*, *pack*, and *unpack*, can change the data format, but not reorder data or add/delete any attribute. Note that the input and output data discussed in this section refers to the input and output of an operator instead of the input and output files of a partitioning program.

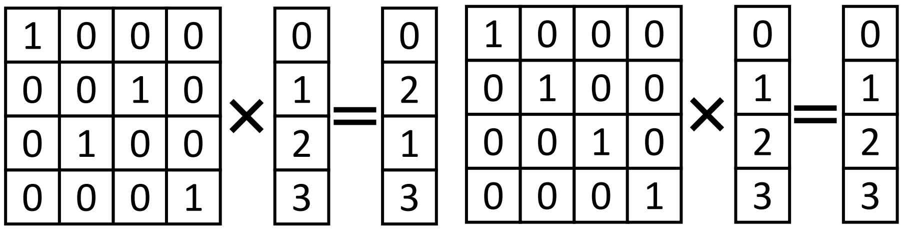
Table 7.1 shows the details of the operators. Most of them will set a field of input data (or intermediate data) as the key and do the computation following the key-value concept. We will present more details with the driving applications in Section 7.1.4. In this paragraph, we focus on the *policy* parameter used in *distribute*, which is an operator not following the key-value concept. In a partitioning algorithm, an entry from the input file is usually put into a partition of output. Although sometimes a entry may be put into multiple partitions for better performance or fault tolerance [188], we discuss the one-to-one mapping like the perfect hash in this research. We design two basic types of policies, i.e., cyclic and block. The partitioning algorithms generated by PaPar will read the parameters *policy* and *numPartitions* from the configuration file at runtime, and formalize the policy to a matrix-vector multiplication operation. We borrow the idea of a domain-specific language (DSL) [189] to define a policy as a permutation matrix: L_m^{km},

Table 7.1: Operators of PaPar workflow

Basic Operator
Sort(String inputPath, String outputPath, Class<?> inputFormat, Class<? extends Format> outputFormat, ValueId key, int flag, Class<? extends AddOn> addOn) Sort data by the given key. ***inputPath***: the path of input data. ***ouputPath***: the path of output data. ***inputFormat***: the format of input data. ***outputFormat***: the format of output data. ***key***: the key for sorting input data. ***flag***: the sorting type; -1: ascending, 1: descending. ***addOn***: add-ons.
Group(String inputPath, String outputPath, Class<?> inputFormat, Class<? extends Format> outputFormat, ValueId key, Class<? extends AddOn> addOn) Group data by the given key. ***inputPath***: the path of input data. ***ouputPath***: the path of output data. ***inputFormat***: the format of input data. ***outputFormat***: the format of output data. ***key***: the key to group input data. ***addOn***: add-ons.
Split(String inputPath, List<String> outputPathList, Class<?> inputFormat, List<? extends Format> outputFormat, ValueId key, SplitPolicy policy, Class<? extends AddOn> addOn) Split data by the given split operation and key. ***inputPath***: the path of the list of inputs. ***outputPathList***: the file list for outputs. ***inputFormat***: the format of the input data. ***outputFormat***: the format of the output data. ***key***: the key for splitting. ***policy***: the policy for splitting data. ***addOn***: add-ons.
Distribute(String inputPath, String outputPath, Class<?> inputFormat, Class<? extends Format> outputFormat, DistrPolicy policy, int numPartitions, Class<? extends AddOn> addOn) Distribute data by the given policy. ***inputPath***: the path of input. ***ouputPath***: the path of output. ***inputFormat***: the format of input data. ***outputFormat***: the format of output data. ***policy***: the distribution policy: cyclic and block. ***numPartitions***: the number of partitions. ***addOn***: add-ons.
Add-on Operator
count(List<T> elements, ValueId key) Count the number of elements of the specific key.
max(List<T> elements, ValueId value) Get the maximum of the specific values of elements.
min(List<T> elements, ValueId value) Get the minimum of the specific values of elements.
mean(List<T> elements, ValueId value) Get the average of the specific values of elements.
sum(List<T> elements, ValueId value) Get the sum of the specific values of elements.
Format Operator
orig(List<T> keyVale) (default) Output data with the input format.
pack(List<T> keyVale) Output data with the packed format.
unpack(List<T> keyValue) Output data with the unpacked format.

$x_{ik+j} \mapsto x_{jm+i}, 0 \leqslant i < m, 0 \leqslant j < k$, which performs a stride-by-m permutation on a vector x having km items. In the distribution policy, x is the input data represented as a vector having km

entries, and m is the stride to permute entries. Fig. 7.6(a) illustrates the example to permute 4 entries with the stride 2 in the cyclic manner. The corresponding permutation matrix is L_2^4. Fig. 7.6(b) illustrates the example for the block policy, which will not permute entries and the matrix is L_4^4. After the permutation, the contiguous data will be sent to two partitions for the distribution. The benefit of using the permutation matrix is to decouple the distribution policies from the workflow when PaPar generates the codes: at the time of code generation, it is not necessary to bind a distribution policy; and at runtime, the parameters *policy* and *numPartitions* will be processed and the permutation matrix will be generated, while the codes of the distribution operator are not changed. At runtime, the matrix-vector multiplication is enforced by multiple mappers in parallel and each mapper only processes its local data distribution based on the multiplication result.



(a) Cyclic matrix L_2^4 (b) Block matrix L_4^4

Figure 7.6: Formalize the distribution polices to matrix-vector multiplication

Though the operators listed in the table are sufficient for most cases, PaPar allows users to define their own operators. Users need to inherit one of these three operator classes, and provide a configuration file to describe the operator. Fig. 7.7 shows an example of customized *sort*. The user needs to specify the class and argument types to tell the framework how to invoke it.

```
<prog id="Sort" type="operator"
      name="MapReduce sort operator">
  <import classpath="/user/mr/sort"
          package="com.mr.sort" class="Sort"/>
  <arguments>
    <param name="inputPath" type="String"/>
    <param name="outputPath" type="String"/>
    <param name="keyId" type="KeyId"/>
    <param name="ascending" type="boolean"
           default="true"/>
  </arguments>
</prog>
```

Figure 7.7: Configuration file for *sort* operator

7.1.4 Case Studies

To demonstrate the capability and usability of PaPar, we use muBLASTP and PowerLyra as case studies.

muBLASTP: Fig. 7.8 shows the workflow configuration file of muBLASTP partitioning. Three parameters listed in the *argument* section are the input file name, output file name, and number of partitions. Two operators *sort* and *distribute* are defined, each of which will be mapped to a job. We use the symbol *$* to represent the variable coming from the output of another operator. For example, for the operator *distribute*, its input comes from the output of operator *sort*, which is labeled as “$sort.outputPath” in the configuration. The optional parameter *num_reducers* is used to launch reducers at runtime. Each operator can use a parameter defined in the workflow arguments or overwrite it in its own parameter section.

Fig. 7.9 illustrates the workflow of muBLASTP partitioning, which sorts input by the encoded sequence length and then distributes elements evenly to partitions with the cyclic policy. This figure follows the MapReduce style. In the figure, the leftmost part shows the index data of muBLASTP. Two jobs are launched in the workflow. The *sort* job sorts entries by using the sequence length *seq_size*. The mappers will shuffle key-value pairs to different reducers according to the range of keys, which is sampled when reading the input. The data sampling will be discussed in Section 7.1.4.1. In this case, as an example, the entries having the key *seq_size* ranging from 90 to 95 are assigned with the reduce-key “1”, and then shuffled to the reducer “1”. The reducers will sort entries by the key *seq_size* and write output data after removing the temporary reduce-key, because the basic operators will only reorder data but not change data as the definition.

The *distribute* job will distribute entries with the cyclic policy. The mappers will enforce the cyclic policy by applying the matrix-vector multiplication in parallel. In this case, each mapper knows there are 4 entries at local and 3 partitions for the output. Therefore, the permutation matrix L_3^4 is generated to permute the entries locally. After that, the mapper will distribute the entries to corresponding partitions. For example, the mapper "0" will send the entries “0” {566, 51, 490, 120}, “3” {1041, 79, 1107, 76} to the partition “0”, the entry “1” {783, 64, 799, 91} to the partition “1”, and so on. Because a reducer is launched to write data for a partition, the reducer id is used as the reduce-key. The reducers of the *distribute* job will write data to the output after removing the temporary reduce-key. Note that, some applications may need to adjust output data. For example, muBLASTP needs to recalculate the start pointers of sequence data and description data. This process has been implemented as a user-defined add-on operator. The algorithm recalculating the muBLASTP index has been discussed in 5.2, and we skip the details in this chapter.

PowerLyra: As introduced in Section 7.1.2.1, the hybrid-cut of PowerLyra will generate the new attributes and use them as the key and value of corresponding operators. Fig. 7.10 shows the configuration file, which concatenates three basic operators — *group*, *split*, and *distribute* — in the

```
<workflow id="blast_partition"
          name="BLAST database partition">
  <arguments>
    <param name="input_path" type="hdfs"
           format="blast_db"/>
    <param name="output_path" type="hdfs"
           format="blast_db"/>
    <param name="num_partitions" type="integer"/>
    <param name="num_reducers" type="integer"
           value="3"/>
  </arguments>
  <operators>
    <operator id="sort" operator="Sort"
            num_reducers="$num_reducers">
      <param name="inputPath" type="String"
             value="$input_path"/>
      <param name="ouputPath" type="String"
             value="/user/sort_output"/>
      <param name="key" type="KeyId"
             value="seq_size"/>
    </operator>
    <operator id="distr" operator="Distribute">
      <param name="inputPath" type="String"
             value="$sort.ouputPath"/>
      <param name="outputPath" type="String"
             value="$output_path"/>
      <param name="distrPolicy" type="DistrPolicy"
             value="roundRobin"/>
      <param name="numPartitions" type="integer"
             value="$num_partitions"/>
    </operator>
  </operators>
</workflow>
```

Figure 7.8: Configuration file for muBLASTP

workflow.

Fig. 7.10 shows the details. The input data represents edges, i.e., *vertex_a* $\rightarrow$ *vertex_b*). The *group* job uses out-vertex *vertex_b* as the key to group edges in the map stage, and uses the add-on operator *count* to add a new attribute on each edge, i.e., vertex indegree, and uses the format operator *pack* to pack output data in the reduce stage. The *split* job then splits the packed entries based on the key *indegree*, which is the new attribute added by the add-on operator *count*. The *split* operator will send the entries, which *indegree* are larger than or equal to *threshold* (i.e., 4 in this example), to the high-degree output, and others to the low-degree output. Note that, for the high-

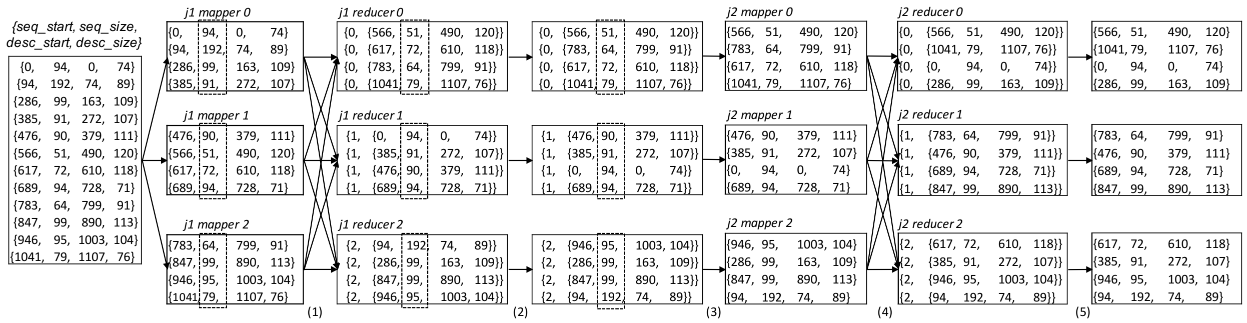


Figure 7.9: The workflow of muBLASTP data partitioning. The *Sort* job will sort the *index* elements by the user-defined key *seq_size* (in the dashed boxes), including: (1) mappers will shuffle data to reducers with the sampled reduce-key; (2) reducers will sort data by the key *seq_size*; (3) store data by removing the reduce-key. The *Distribute* job will distribute the sorted elements to partitions with the cyclic policy, including: (4) mappers will shuffle data to reducers with the generated reduce-key (reducer id); (5) remove the temporary reduce-key.

degree output, the format operator *unpack* is used to unpack data from the packed organization (as shown in the step 5 in the figure). The third job *distribute* will then operate on two different formats of intermediate data and generate two permutation matrices, i.e., L^4_3 for the high-degree and L^3_3 for the low-degree. Note that L^3_3 in this case happens not to permute data, because there are 3 entries for 3 partitions. In a general case, L^M_N will enforce the cyclic distribution when M is larger than N. As the *distribute* is the last step in the workflow, all data will be unpacked to make sure the output has the same format of the input.

7.1.4.1 Implementations

We map our framework on top of Apache Hadoop (2.7.0), MapReduce-MPI (abbr. MR-MPI) [190], and MPI. The interfaces of first two MapReduce systems are similar. On Hadoop, we implement the interfaces of processing structured data by inheriting *InputFormat* class. We implement those operators in Java and generate Hadoop jobs for the workflow. On MR-MPI, an open-source C++

```
<workflow id="hybrid_cut" name="Hybrid-cut">
  <arguments>
    <param name="input_file" type="hdfs"
           format="graph_edge"/>
    <param name="output_path" type="hdfs"
           format="graph_edge"/>
    <param name="num_partitions" type="integer"/>
    <param name="threshold" type="integer"/>
  </arguments>
  <operators>
    <operator id="group" operator="group">
      <param name="inputPath" type="String"
             value="$input_file"/>
      <param name="outputPath" type="String"
             value="/tmp/group" format="pack"/>
      <param name="key" type="KeyId"
             value="vertex_b"/>
      <addon operator="count" key="vertex_b"
             attr="indegree"/>
    </operator>
    <operator id="split" operator="Split">
      <param name="inputPath" type="String"
             value="$sort.outputPath"/>
      <param name="outputPathList"
             type="StringList"
             value="/tmp/split/high_degree,
                    /tmp/split/low_degree"
             format="unpack,orig"/>
      <param name="key" type="KeyId"
             value="$group.$indegree"/>
      <param name="policy" type="SplitPolicy"
             value="{>=,$threshold},
                    {<,$threshold}"/>
    </operator>
    <operator id="distr" operator="Distribute">
      <param name="inputPath" type="String"
             value="/tmp/split/">
      <param name="outputPath" type="String"
             value="$output_path"/>
      <param name="policy" type="distrPolicy"
             value="graphVertexCut"/>
      <param name="numPartitions" type="integer"
             value="$num_partitions"/>
    </operator>
  </operators>
</workflow>
```

Figure 7.10: Configuration file for PowerLyra hybrid-cut

implementation of MapReduce on MPI, we use C++ to implement mappers and reducers by calling MR-MPI interfaces. The MR-MPI library can help us to hide the details of MPI based data shuffle and synchronization. On MPI, we currently use MPI non-blocking interfaces (Isend, Irecv, and Wait) to implement the data shuffle. During the execution of a PaPar-generated partitioner,

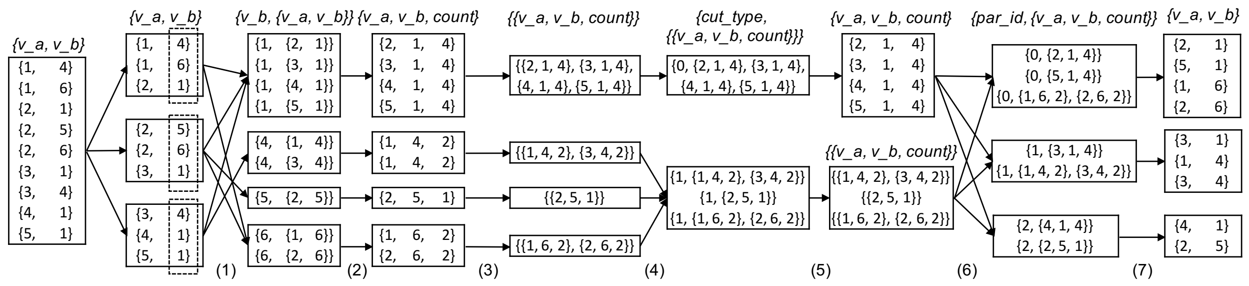


Figure 7.11: The workflow of PowerLyra hybrid-cut algorithm. The *Group* job will group the edges by in-vertex, including: (1) mappers will shuffle data to reducers by setting the in-vertex id as the reduce-key; (2) the add-on operator *count* will add a new attribute indegree for each edge; (3) the format operator *pack* will change the output format to the packed one. The *Split* job will split data into two groups, including: (4) based on the split condition in the configuration file (indegree is larger than or equal to 4 in this case), mappers will set the reducer id as the temporary reduce-key and shuffle data to reducers; (5) based on the different formats of output files, the *unpack* operator is applied on the high-degree part to unpack the data format. The *Distribute* job will distribute the entries in a cyclic manner, including: (6) mappers will shuffle data to reducers by setting the reducer id as the reduce-key; (7) reducers will remove the temporary reduce-key.

the jobs are launched one by one following the order defined in the workflow configuration file. Several important techniques are also implemented as below:

Code Generation: We implement a parser to parse the configuration files and generate the Hadoop or MPI based partitioner by directly calling the backend implementations of operators. This method has been widely used in the code generation from a higher-level description to a lower-level implementation, e.g., from SQL to MapReduce jobs in Apache Hive [191], from SQL to GPU kernels [192], from DSL to SIMD implementations of sorting networks [151], etc. We plan to use an internal representation (IR) [193] to decouple the binding between the frontend and the backend in the future work.

Data Sampling: We implement the data sampling to balance the workload for the reduce stage. For example, for the *sort* operator, the temporary reduce-key corresponding to the range of input data is needed. In order to avoid the imbalance on reducers, we follow the mechanisms proposed in [26] to sample data on every node and approximate to the global data distribution. Based on the distribution of the user-set key and the number of reducers, we set the proper data range for each temporary reduce-key.

Data Compression: This optimization is used to compress the packed data. As shown in the hybrid-cut of PowerLyra, the *group* operator will call the *pack* operator to pack edges having the same in-vertex, resulting in the redundant data in this packed format. As shown in Fig. 7.11, after the step 3, the reducer 0 has the packed data as {{2, 1, 4}, {3, 1, 4}, {4, 1, 4}, {5, 1, 4}}, and the redundant data is 1. This optimization uses the Compressed Sparse Row (CSR) and its transposition Compressed Sparse Column (CSC), which are widely used in sparse matrix computations [194, 195, 196, 197, 198], to compress data. In this case, the CSC format {0, {2, 3, 4, 5}, {4, 4, 4, 4}} is used: 0 is the start pointer of the in-vertex 1, the first vertex in the graph; {2, 3, 4, 5} is the out-vertex id array, and {4, 4, 4, 4} is the value array. Because the value array may include different values (depending on the algorithm to generate the attribute), we do not compress the value array to keep the generality. This optimization can improve the data communication performance, while it highly depends on the input data. We have observed up to 13% improvement for the graph datasets in our evaluation.

7.1.5 Experiments

7.1.5.1 Experimental Setup

We conduct our evaluations on a homogeneous cluster consisting of 16 compute nodes. Each node has two 8-core Intel Xeon E5-2670 (Sandy Bridge) CPUs running at 2.60 GHz, 64 GB memory, and 512 GB local disk. These nodes are linked by 10Gbps Ethernet and a Quad Data Rate (QDR) InfiniBand interconnect. Because both muBLASTP and PowerLyra are implemented in C++, we map PaPar on MR-MPI that leverages the MapReduce concept and the in-memory communication on MPI to provide comparable performance. All codes are compiled with the MVAPICH2 library (version 2.2) and GCC 4.5.3. In all experiments, the execution time is the average time of five runs without I/O time.

In the muBLASTP experiments, two partitioning methods are generated by PaPar. One is the default method to keep the number of sequences in partitions similar. We label it as "block". The other is the optimized method that will sort the index and distribute the sequences in a cyclic manner. We label it as "cyclic". We use two popular protein databases as the test datasets: *env_nr* database and *nr* database. The *env_nr* database consists of about 6,000,000 sequences with the total size at 1.7 GB, and the *nr* database has over 85,000,000 sequences with the size at 53 GB. Most of the sequences in two databases are less than 100 letters. We follow the experimental setups in [32] to randomly pick up sequences from corresponding databases to construct three batches, each of which includes 100 sequences. In the batch "100" and "500", all sequences are less than 100 and 500 letters, respectively; and for the "mixed" batch, we randomly select 100 sequences without the limitation of length.

In the PowerLyra experiments, we generate codes for three types of partitioning methods, "edge-cut", "vertex-cut", and "hybrid-cut" shown in Fig. 7.2 . We choose PageRank as the test algorithm, which computes the rank of vertices in a graph. We use the snapshot version of PowerLyra with the tuned command line parameters downloaded from the PowerLyra website. The threshold parameter of hybrid-cut is set to 200 to divide the vertices into the low-cut or high-cut group. We choose three graph datasets: *Google*, *Pokec* and *LiveJournal*, from SNAP [199]. The datasets are stored in the *EdgeList* format as shown in Fig. 7.5. Table 7.2 shows the statistics of these datasets.

Table 7.2: Statistics of graph datasets

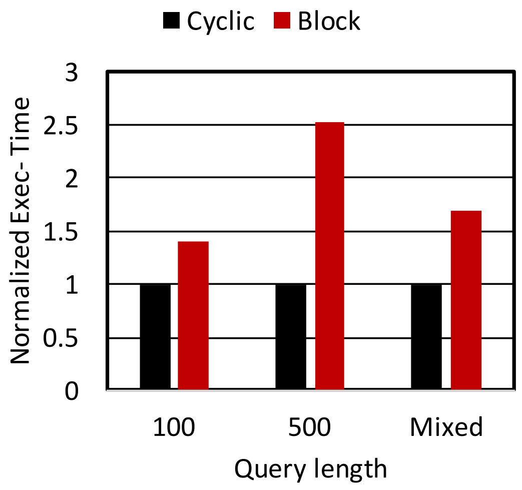
Graph	**Vertices**	**Edges**	**Type**	**Triangles**
Google	875713	5105039	Directed	13391903
Pokec	1632803	30622564	Directed	32557458
LiveJournal	4847571	68993773	Directed	177820130

In our evaluations, we first compare the partitions generated by PaPar and by the partitioning programs of driving applications. The results show that PaPar can produce the same partitions as the driving applications. After that, we present the performance numbers, including the execution time of applications with different partitioning algorithms, the partitioning time on the given input data sets, and the scalability on multiple compute nodes.

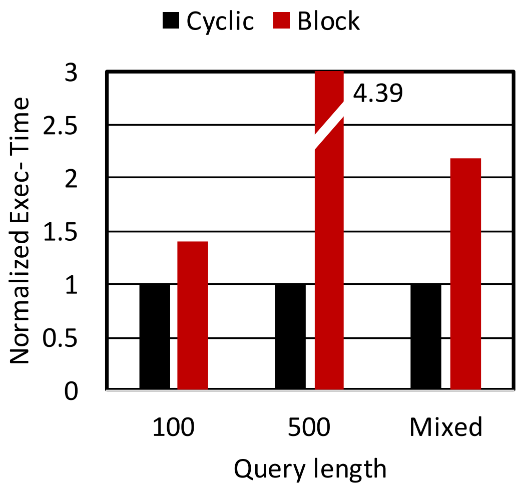
7.1.5.2 Evaluation of BLAST Database Partitioning

Fig. 7.12 shows the normalized execution time of muBLASTP search for three batches on 8 and 16 compute nodes with the cyclic and block policies. muBLASTP follows the MPI + OpenMP programming model, and the best performance can be achieved when binding an MPI process to a CPU (socket) and launch multiple OpenMP threads (8 on our Intel Sandy Bridge CPU) in one MPI process. As a result, on 8 nodes, we produce 16 (8 * 2) partitions; and on 16 nodes, the partition number is 32 (16 * 2). In these figures, the cyclic policy is the clear winner that can bring obvious performance benefits to muBLASTP, no matter which combination of database and batch is used. We also observe that the cyclic policy can achieve more performance benefits for the larger batch, i.e. the batch "500". That means the skew is more significant for the longer queries because they have relatively longer search time.

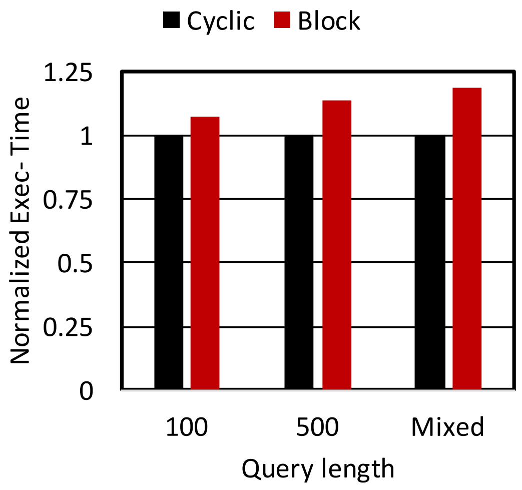
Because the cyclic policy can deliver better performance to muBLASTP search, we compare the partitioning time of PaPar and default muBLASTP partitioning for this policy. Fig. 7.13(a) shows the normalized partitioning time on 16 nodes for the *env_nr* and *nr* databases, respectively. Because the current implementation of muBLASTP partitioning only provides a multithreaded method for the input database [32], it can not scale out on 16 nodes. On the contrary, PaPar can map to MapReduce and MPI implementations, and scale on multiple compute nodes. As shown in the figure, PaPar can achieve 8.6x and 20.2x speedups over default muBLASTP partitioning on 16 nodes for two databases, respectively. Note that even on a single compute node, PaPar is faster, thanks to ASPaS [151], a highly optimized merge sort implementation on multicore processors. We used it in the sort operator implementation. Fig. 7.13(b) shows the scalability up to 16 nodes. Compared to its own single node implementation, PaPar can obtain 7.9x and 14.3x speedups for the *nr* and *evn_nr* databases, respectively.



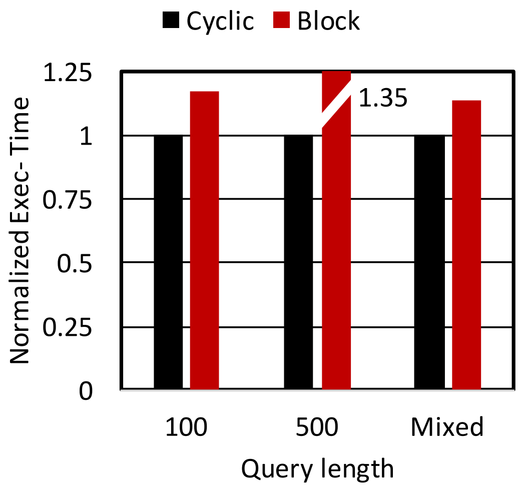
(a) *env_nr* database on 8 nodes



(b) *env_nr* database on 16 nodes

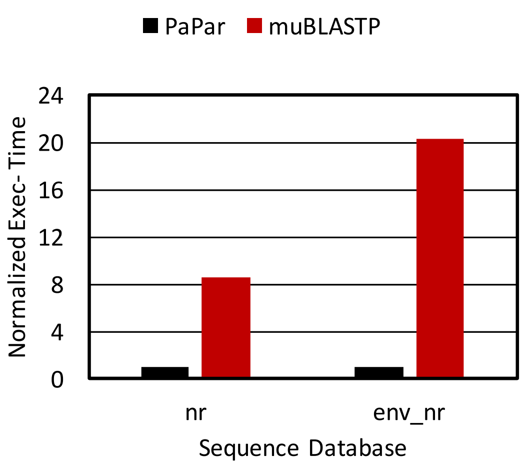


(c) *nr* database on 8 nodes

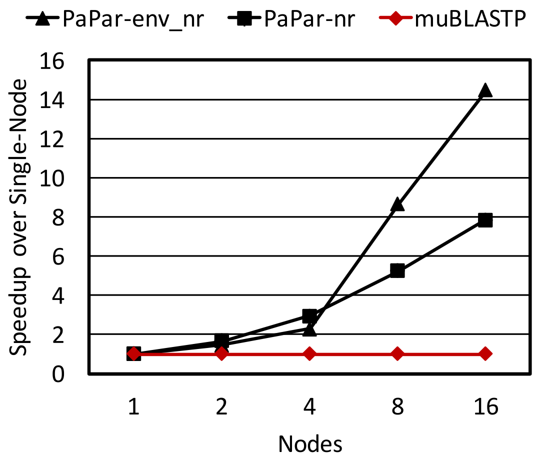


(d) *nr* database on 16 node

Figure 7.12: Normalized execution time of muBLASTP with the cyclic partitioning and block partitioning (normalized to cyclic) on *env_nr* and *nr* databases.



(a) Partitioning time on 16 nodes



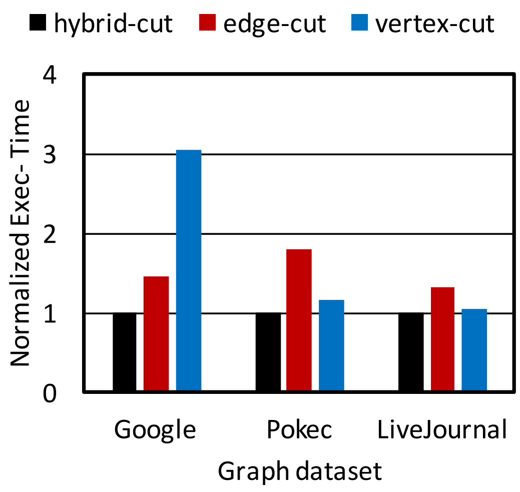
(b) Scalability (up to 16 nodes)

Figure 7.13: Partitioning time (cyclic) for *env_nr* and *nr* databases, and strong scalability of codes generated by PaPar, compared to muBLASTP partitioning program.

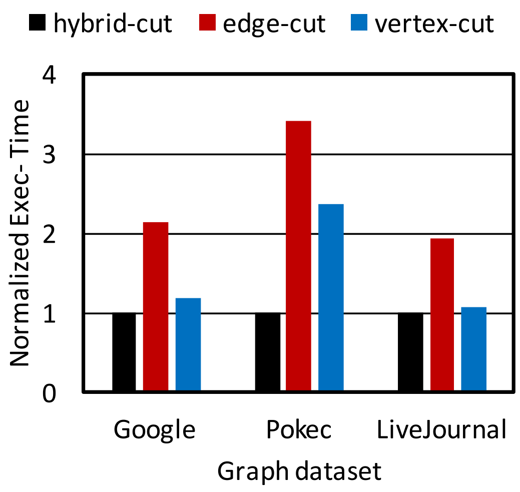
7.1.5.3 Evaluation of Hybrid-Cut Graph Partitioning

Fig. 7.14 shows the normalized execution time of PageRank with "hybrid-cut", "edge-cut", and "vertex-cut" on 8 and 16 nodes. The hybrid-cut can deliver the best performance as we expected. The vertex-cut distributes a vertex with all its in-edges to a partition, which favors the vertices having low-degrees. Because the three datasets in our experiments follow the power law distribution that have much more low-degree vertices, the vertex-cut, instead of the edge-cut, has closer performance to the hybrid-cut.

Fig. 7.15(a) shows the normalized partitioning time of PaPar codes and PowerLyra on 16 nodes for the hybrid-cut. On the Google and Pokec datasets, PowerLyra has the better performance; while PaPar can deliver 1.2x speedup on the LiveJournal dataset. There are several reasons leading to the variable performance comparison. PaPar is mapped on MR-MPI to balance the programmability



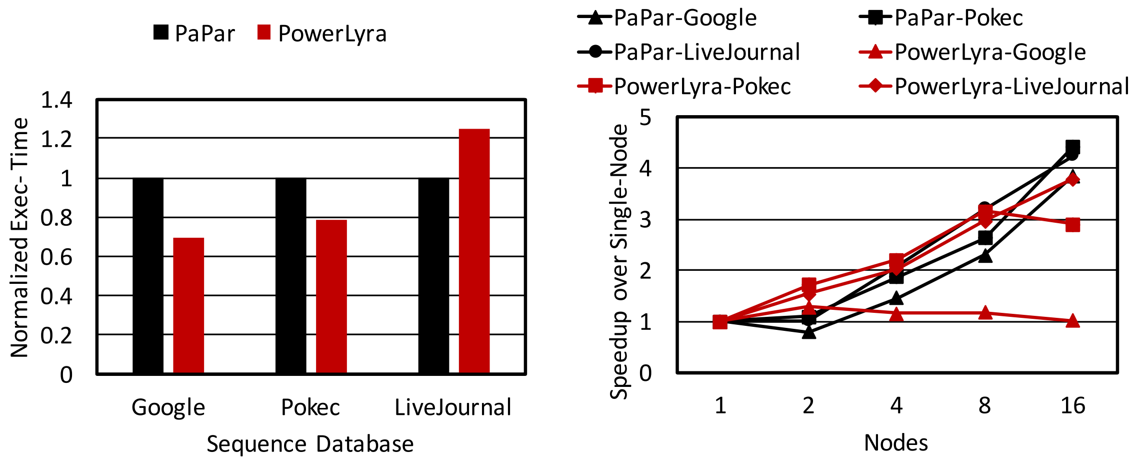
(a) PageRank running on 8 nodes



(b) PageRank running on 16 nodes

Figure 7.14: Normalized execution time of PageRank (with PowerLyra) for hybrid-cut, edge-cut, and vertex-cut (normalized to hybrid-cut).

and performance but without those optimizations on multicore processors used by PowerLyra, e.g., the NUMA-aware data access. Therefore, PowerLyra is faster for the small and medium datasets, where the single node performance counts more. However, such a benefit is offset in the communication intensive case on multiple nodes. Although PowerLyra is integrated with GraphLab on top of MPI, its data shuffle is still based on the socket communication on Ethernet. On the contrary, PaPar maps to MR-MPI that uses MPI instead of socket communication. In our experiments, the MVAPICH2 library can use Remote Direct Memory Access (RDMA) communication on InfiniBand to improve the performance. Furthermore, PowerLyra uses the dynamic approach that calculates scores for low-degree vertices in each partition. This method introduces additional overhead, especially for graphs which vertices cluster together, e.g., the LiveJournal dataset. Fig. 7.15(b) also demonstrates the variable performance. PowerLyra can scale up to 8 and 16 nodes for the Pokec and LiveJournal datasets, respectively, but cannot scale on multiple nodes for the Google dataset; while, PaPar can scale up to 16 nodes for all three datasets.



(a) Partitioning time on 16 nodes (b) Scalability (up to 16 nodes)

Figure 7.15: Partitioning time (hybrid-cut) and strong scalability of codes generated by PaPar framework, compared to PowerLyra.

7.1.6 Conclusion

In this research, we propose the PaPar framework to generate application-specific partitioning algorithms. Taking two configuration files as input, PaPar can formalize the partitioning workflow as a sequence of key-value operations and matrix-vector multiplications, and map to implementations on MPI and MapReduce. We use muBLASTP and PowerLyra as the case studies to show how to generate the user-defined partitioning algorithms with PaPar. Our evaluations illustrate PaPar can generate the same partitions with comparable or less partitioning time.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

Nowadays, the applications in emerging domains have increasing irregularities in memory access, control flow and network communication patterns. However, current parallel architectures, which are designed for regular computation and memory accesses, are inefficient for irregular applications. Therefore, many previous approaches have been proposed to map irregular applications to parallel architectures through reordering data and computation. However, these approaches cannot fully exploit the locality and regularity in irregular application due to *local optimizations* and *"one-size-fits-all" methods*. To overcome these challenges, we propose a general methodology with three techniques 1) a taxonomy for irregular applications that has four irregularity classes based on the relationship between functions and data structures to help us analyze the irregularity in an application, 2) general transformation that provides *interchanging*, *decoupling* and *reordering* transformations to explore hidden locality across loops or kernels in irregular applications, and 3) adaptive optimization that provides the adaptive data reordering pipeline based the characteristics of the application and architecture to achieve the optimal performance. To evaluate our methodology, we analyze and optimize couples of important and complex irregular applications from different domains across parallel architectures as case studies.

8.1.1 Optimizing Irregular Applications on Multi-core Architectures

On multi-core architectures, we first investigate Burrow-Wheeler Aligner (BWA), a popular short read alignment tool on multi-core architectures. Through in-depth performance analysis of BWA, we determine that the BWA kernel belongs to the *MDSC* class, where the irregular memory behavior is the performance bottleneck of such tools due to poor locality. We then propose a locality-aware implementation of BWA with *interchanging* its execution order and *reordering* intermediate data between iterations with a cache-oblivious bin structure to minimize the reordering overhead. Experimental results show that our optimized BWA implementation can reduce LLC misses by 30% and TLB misses by 20%, resulting in up to a 2.6-fold speedup over the original BWA implementation.

We then investigate Basic Local Alignment Search Tool (BLAST), which is another important bioinformatics application. We first develop a BLAST algorithm using database indexing instead of query indexing for better exploiting the cache mechanism. We then demonstrate irregular problems in the database-indexed BLAST algorithm (*MDMC* class) and propose a refactor BLAST algorithm that *decouples* mixed phases, and *reorders* intermediate data between phases with a data reordering pipeline of binning-filtering-sorting. Our optimized BLAST can produce the identical results as the original query-indexed BLAST with up to a 4.41-fold speedup with a single thread, and up to a 5.7-fold speedup for 24 threads.

8.1.2 Optimizing Irregular Applications on Many-core Architectures

Beyond multi-core architectures, we also investigate irregular problems on many-core architectures.

First, we investigate the BLAST algorithm on NVIDIA GPUs. We demonstrate that the irregularity in BLAST on a GPU (i.e., *SDMC* class), especially for protein sequence search, can result in serious performance degradation on a GPU. To overcome these problems, we present a fine-grained parallel approach, referred as cuBLASTP. In cuBLASTP, we resolve the irregular memory accesses and branch divergence via *decoupling* the complex irregular kernel into separate kernels and *reordering* data between kernels with a data reordering pipeline of binning-sorting-filtering. Experimental results show that cuBLASTP can achieve up to a 3.4-fold speedup over the multithreaded NCBI-BLAST, and up to a 2.9-fold speedup over the state-of-the-art GPU-implementation.

Second, we identify the irregular problems in the existing implementations of dynamic parallelism on a GPU that use the "one-size-fits-all" subtask aggregation method. To alleviate irregularities, we use *decoupling* method to redistribute aggregated subtasks into separate kernels by their properties.

To achieve the optimal performance, we present an adaptive subtask aggregation and scheduling tool based on machine learning techniques to generate the optimal parallel strategy. Experimental results show that the optimal subtask aggregation strategy provided by our tool can achieve up to a 1.8-fold speedup over the existing subtask aggregation method.

8.1.3 Optimizing Irregular Applications on Multi-node Systems

On multi-node systems, workload imbalance is a fundamental problem. Though current frameworks provide advanced mechanisms to partition data and resolve workload imbalance, they are inefficient for irregular applications, whose run time depends on not only a single properties i.e., the database size, but also multiple other properties, such as algorithms applied on the data, data locality and data distribution. To effectively resolve workload imbalance of irregular applications, we propose a framework, called PaPar, that can generate the desired data partitioning codes for irregular applications via composing input-data transformations, including *Sort*, *Group*, *Split*, and *Distribute*. We conduct our experiments on a cluster with 16 compute nodes. Experimental results show that the codes generated by PaPar can produce the same partition quality as the application-specific partitioning by manual with less partitioning time.

8.2 Future Work

There are the two possible major extensions to this dissertation. First, our methodology can be extended into a framework that can direct the compilers and runtimes to automatically optimize the irregular applications. Second, our methodology can be extended to resolve "irregularities" in regular applications, such as deep neural network kernels, dense linear algebra algorithms, etc.

8.2.1 Extending Our Methodology into An Automatic Framework

In our methodology, we provide an irregularity taxonomy and a set of general transformations to help us to analyze and optimize irregular applications. However, applying these analysis and optimization techniques in real-world applications is still non-trivial. To simplify the process of analysis and optimization, our methodology could be extended into a framework that can automatically discover and identify the irregularity in an application, and direct compilers and runtimes to generate the optimized implementation.

In Section 6.2, we provide an example of building models for performance analysis and prediction using statistical analysis and machine learning techniques. Similarly, we can build performance models with a set of irregular benchmarks to obtain the correlation between the irregular patterns and the performance counter data [108, 107, 109]. Therefore, we can identify the irregularity patterns in the irregular application based on the performance counter data. And then, with the extension of the performance models in Section 4.3.3, the framework can determine the optimal transformation strategy for the application and architecture to direct the compiler and runtime for generating the optimized implementation.

8.2.2 Extending Our Methodology for Regular Applications

According to [76], irregular is not a binary property, which not only exists in irregular applications but also regular applications. For example, regular applications may have workload imbalance and resource skew. For example, applications in GPU-based scientific computing kernels and deep learning kernels on a GPU may require solving many independent dense matrix operations that are different in size. These matrix operations have various resource usage and parallelism, which requires different tuning parameters. However, existing studies [200, 201, 202] use the "one-size-fits-all" approach that execute all matrix operations with a single kernel and applies the uniform configuration on all operations, which will underutilize the GPU resource and degrade the performance. This problem is very similar with the performance issue in the subtask aggregation for dynamic parallelism. Thus, we can easily extend our subtask aggregation model presented in Section 6.2 to decouple the matrix operations into separate kernels and generate the optimal aggregation strategy.

Furthermore, to improve the utilization of GPUs or other many-core architectures, GPU multiprogramming, which allows two or more kernels from different applications being executed concurrently on a GPU, began to attract a wide attention. However, running two kernels concurrently with different irregular patterns can make the resource sharing and contention very complicated. To generate the optimal aggregation and scheduling strategy for multiprogramming on a GPU, we also can extend our subtask aggregation model with the performance models for multiple applications.

Bibliography

[1] Radeon's Next-generation Vega architecture. `https://radeon.com/_downloads/vega-whitepaper-11.6.17.pdf`, 2017.

[2] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A View of the Parallel Computing Landscape. *Communications of the ACM - A View of Parallel Computing*, 52(10):56–67, October 2009.

[3] TOP500 project. `https://www.top500.org/`, November 2017.

[4] Intel 64 and IA-32 Architectures Optimization Reference Manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, 2016.

[5] Chen Ding and Ken Kennedy. Improving Cache Performance in Dynamic Applications Through Data and Computation Reorganization at Run Time. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 229–241, New York, NY, USA, 1999. ACM.

[6] Michelle Mills Strout and Paul D. Hovland. Metrics and Models for Reordering Transformations. In *Proceedings of the 2004 Workshop on Memory System Performance*, MSP ’04, pages 23–34, New York, NY, USA, 2004. ACM.

[7] I-Jui Sung, John A. Stratton, and Wen-Mei W. Hwu. Data Layout Transformation Exploiting Memory-level Parallelism in Structured Grid Many-core Applications. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT ’10, pages 513–522, New York, NY, USA, 2010. ACM.

[8] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly Elimination of Dynamic Irregularities for GPU Computing. *ASPLOS ’11*, 46(3):369–380, March 2011.

[9] Shuai Che, Jeremy W. Sheaffer, and Kevin Skadron. Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’11, pages 13:1–13:11, New York, NY, USA, 2011. ACM.

[10] Teresa L. Johnson, Matthew C. Merten, and Wen-Mei W. Hwu. Run-time Spatial Locality Detection and Optimization. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 30, pages 57–64, Washington, DC, USA, 1997. IEEE Computer Society.

[11] Sanjeev Kumar and Christopher Wilkerson. Exploiting Spatial Locality in Data Caches Using Spatial Footprints. In *Proceedings of the 25th Annual International Symposium on*

Computer Architecture, ISCA ’98, pages 357–368, Washington, DC, USA, 1998. IEEE Computer Society.

[12] Akanksha Jain and Calvin Lin. Linearizing Irregular Memory Accesses for Improved Correlated Prefetching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 247–259, New York, NY, USA, 2013. ACM.

[13] Shun-tak Leung and John Zahorjan. Optimizing Data Locality by Array Restructuring. Technical report, 1995.

[14] Mahmut Kandemir, Alok Choudhary, Prith Banerjee, and J. Ramanujam. A Matrix-Based Approach to the Global Locality Optimization Problem. *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 0:306, 1998.

[15] Mahmut Kandemir, J. Ramanujam, and Alok Choudhary. Improving Cache Locality by a Combination of Loop and Data Transformations. *IEEE Transactions on Computers*, 48(2):159–167, February 1999.

[16] M.F.P. O’Boyle and P.M.W. Knijnenburg. Integrating Loop and Data Transformations for Global Optimization. *J. Parallel Distributed Comput.*, 62(4):563–590, April 2002.

[17] Trishul M. Chilimbi. Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’01, pages 191–202, New York, NY, USA, 2001. ACM.

[18] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings. *Int. J. Parallel Program.*, 29(3):217–247, June 2001.

[19] Heng Li and Richard Durbin. Fast and Accurate Short Read Alignment with Burrows-Wheeler Transform. *Bioinformatics (Oxford, England)*, 25(14):1754–60, July 2009.

[20] Jing Zhang, Heshan Lin, Pavan Balaji and Wu chun Feng. Optimizing Burrows-Wheeler Transform-Based Sequence Alignment on Multicore Architectures. In *Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 377–384, May 2013.

[21] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2004.

[22] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[23] Ganesh Ananthanarayanan, Srikanth Kandula, Albert G Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, volume 10, page 24, 2010.

[24] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: Mitigating Skew in MapReduce Applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 25–36. ACM, 2012.

[25] Smriti R Ramakrishnan, Garret Swart, and Aleksey Urmanov. Balancing Reducer Skew in MapReduce Workloads using Progressive Sampling. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 16. ACM, 2012.

[26] Benjamin Gufler, Nikolaus Augsten, Angelika Reiser, and Alfons Kemper. Load Balancing in MapReduce based on Scalable Cardinality Estimates. In *Proceedings of the 28th IEEE International Conference on Data Engineering*, pages 522–533. IEEE, 2012.

[27] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: a System for Dynamic Load Balancing in Large-Scale Graph Processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 169–182. ACM, 2013.

[28] Qi Chen, Jinyu Yao, and Zhen Xiao. Libra: Lightweight Data Skew Mitigation in MapReduce. *IEEE Transactions on Parallel and Distributed Systems*, 26(9):2520–2533, 2015.

[29] Shanjiang Tang, Bu-Sung Lee, and Bingsheng He. Dynamic Job Ordering and Slot Configurations for MapReduce Workloads. *IEEE Transactions on Services Computing*, 9(1):4–17, 2016.

[30] Jing Zhang, Hao Wang, Heshan Lin, and Wuchun Feng. Consolidating Applications for Energy Efficiency in Heterogeneous Computing Systems. In *Proceedings of the 2013 IEEE*

10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing, pages 399–406, Nov 2013.

[31] Hao Wang, Jing Zhang, Da Zhang, Sarunya Pumma, and Wuchun Feng. PaPar: A Parallel Data Partitioning Framework for Big Data Applications. In *Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 605–614, May 2017.

[32] Jing Zhang, Sanchit Misra, Hao Wang, and Wu-chun Feng. muBLASTP: Database-Indexed Protein Sequence Search on Multicore CPUs. *BMC Bioinformatics*, 17(1):443, Nov 2016.

[33] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, page 1. ACM, 2015.

[34] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2010. Version 3.2.

[35] The OpenCL Specification v2.0, 2015.

[36] Wu-chun Feng, Heshan Lin, Thomas Scogland, and Jing Zhang. OpenCL and the 13 Dwarfs: A Work in Progress. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 291–294, New York, NY, USA, 2012. ACM.

[37] ROCm - Open Source Platform for HPC and Ultrascale GPU Computing. `https://rocm.github.io/install.html`, 2017.

[38] Asynchronous Task and Memory Interface (ATMI). `https://github.com/RadeonOpenCompute/atmi`, 2017.

[39] Jin Wang and Sudhakar Yalamanchili. Characterization and Analysis of Dynamic Parallelism in Unstructured GPU Applications. In *Proceedings of the 2014 IEEE International Symposium on Workload Characterization*, pages 51–60, 2014.

[40] Message P Forum. MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA, 1994.

[41] OpenMPI. `https://www.open-mpi.org/`.

[42] MVAPICH. `http://mvapich.cse.ohio-state.edu/`.

[43] MPICH. `https://www.mpich.org/`.

[44] Apache. Hadoop. `http://hadoop.apache.org/`.

[45] Apache. HDFS Architecture Guide. `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`.

[46] Da Zhang, Hao Wang, Kaixi Hou, Jing Zhang, and Wu chun Feng. pDindel: Accelerating Indel Detection on a Multicore CPU Architecture with SIMD. In *Proceedings of the 5th IEEE International Conference on Computational Advances in Bio and Medical Sciences (ICCABS)*, pages 1–6, Oct 2015.

[47] Ruiqiang Li, Yingrui Li, Karsten Kristiansen, and Jun Wang. SOAP: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714, 2008.

[48] Andrew D. Smith, Zhenyu Xuan, and Michael Q. Zhang. Using Quality Scores and Longer Reads Improves Accuracy of Solexa Read Mapping. *BMC Bioinformatics*, 9(1):128, Feb 2008.

[49] Heng Li, Jue Ruan, and Richard Durbin. Mapping Short DNA Sequencing Reads and Calling Variants using Mapping Quality Scores. *Genome Res*, 18:1851, 2008.

[50] Petr Klus, Simon Lam, Dag Lyberg, Ming Sin Cheung, Graham Pullan, Ian McFarlane, Giles SH Yeo and Brian YH Lam. BarraCUDA - a Fast Short Read Sequence Aligner Using Graphics Processing Units. *BMC research notes*, 5(1):27, January 2012.

[51] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L. Salzberg. Ultrafast and Memory-Efficient Alignment of Short DNA Sequences to the Human Genome. *Genome biology*, 10(3):R25, January 2009.

[52] Michael Schindler. A Fast Block-Sorting Algorithm for Lossless Data Compression. In *Proceedings of the Conference on Data Compression*, DCC '97, pages 469–, Washington, DC, USA, 1997. IEEE Computer Society.

[53] Udi Manber and Gene Myers. Suffix Arrays: a New Method for On-line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[54] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 390–398, 2000.

[55] Xiaokui Shu, Jing Zhang, Danfeng Daphne Yao, and Wuchun Feng. Fast Detection of Transformed Data Leaks. *IEEE Transactions on Information Forensics and Security*, 11(3):528–542, March 2016.

[56] Xiaokui Shu, Jing Zhang, Danfeng Yao, and Wu-chun Feng. Rapid Screening of Transformed Data Leaks with Efficient Algorithms and Parallel Computing. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, pages 147–149, New York, NY, USA, 2015. ACM.

[57] Xiaokui Shu, Jing Zhang, Danfeng Yao, and Wu chun Feng. Rapid and Parallel Content Screening for Detecting Transformed Data Exposure. In *Proceedings of the 2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 191–196, April 2015.

[58] Kaixi Hou, Hao Wang, and Wu-Chun Feng. AAlign: A SIMD Framework for Pairwise Sequence Alignment on x86-Based Multi-and Many-Core Processors. 2016.

[59] Michael Farrar. Striped Smith–Waterman Speeds Database Searches Six Times over Other SIMD Implementations. *Bioinformatics*, 23(2):156–161, January 2007.

[60] Torbjørn Rognes. Faster Smith-Waterman Database Searches with Inter-Sequence SIMD Parallelisation. *BMC Bioinformatics*, 12(1):221, Jun 2011.

[61] Michael Cameron, Hugh E. Williams, and Adam Cannane. Improved Gapped Alignment in BLAST. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 1(3):116–129, 2004.

[62] Dennis A. Benson, Ilene Karsch-Mizrachi, David J. Lipman, James Ostell, and David L. Wheeler. GenBank. *Nucleic Acids Research*, 41:D36–42, Jan 2013.

[63] Michael Cameron, Hugh E. Williams, and Adam Cannanee. A Deterministic Finite Automaton for Faster Protein Hit Detection in BLAST. *Journal of Computational Biology*, 13(4):965–978, 2006.

[64] Shucai Xiao, Heshan Lin, and Wu-chun Feng. Accelerating Protein Sequence Search in a Heterogeneous Computing System. In *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium*, pages 1212–1222, 2011.

[65] Juan C. Pichel, David E. Singh, and Jesús Carretero. Reordering Algorithms for Increasing Locality on Multicore Processors. In *Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications*, HPCC '08, pages 123–130, Washington, DC, USA, 2008. IEEE Computer Society.

[66] Eun-jin Im and Katherine Yelick. Model-Based Memory Hierarchy Optimizations for Sparse Matrices. In *In Workshop on Profile and Feedback-Directed Compilation*, 1998.

[67] Wai-Hung Liu and Andrew H. Sherman. Comparative Analysis of the Cuthill-McKee and the Reverse Cuthill-McKee Ordering Algorithms for Sparse Matrices. *SIAM Journal on Numerical Analysis*, 13(2):198–213, 1976.

[68] I. Al-Furaih and S. Ranka. Memory Hierarchy Management for Iterative Graph Structures. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 298–302, Mar 1998.

[69] Chen Ding and Ken Kennedy. Improving Cache Performance in Dynamic Applications Through Data and Computation Reorganization at Run Time. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '99, pages 229–241, New York, NY, USA, 1999. ACM.

[70] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings. *Int. J. Parallel Program.*, 29(3):217–247, June 2001.

[71] Nicholas Mitchell, Larry Carter, and Jeanne Ferrante. Localizing Non-Affine Array References. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, PACT '99, pages 192–, Washington, DC, USA, 1999. IEEE Computer Society.

[72] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. Compile-time Composition of Run-time Data and Iteration Reorderings. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '03, pages 91–102, New York, NY, USA, 2003. ACM.

[73] Hwansoo Han and Chau-Wen Tseng. *A Comparison of Locality Transformations for Irregular Codes*, pages 70–84. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[74] Hwansoo Han and Chau-Wen Tseng. *Improving Locality for Adaptive Irregular Scientific Codes*, pages 173–188. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[75] Hwansoo Han and Chau-Wen Tseng. Exploiting Locality for Irregular Scientific Codes. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):606–618, July 2006.

[76] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A Quantitative Study of Irregular Programs on GPUs. In *Proceedings of the 2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 141–151, Nov 2012.

[77] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for Gpgpus. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 225–234, New York, NY, USA, 2008. ACM.

[78] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures. *IEEE Transactions Parallel Distributed Systems*, 22(1):105–118, January 2011.

[79] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU Graph Traversal. *SIGPLAN Not.*, 47(8):117–128, February 2012.

[80] I-Jui Sung, John A. Stratton and Wen-Mei W. Hwu. DL: A Data Layout Transformation System for Heterogeneous Computing. In *2012 Innovative Parallel Computing (InPar)*, pages 1–11, May 2012.

[81] R. Novak. Loop Optimization for Divergence Reduction on GPUs with SIMT Architecture. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1633–1642, June 2015.

[82] Kaixi Hou, Hao Wang, and Shuai Che. Auto-Tuning Strategies for Parallelizing Sparse Matrix-Vector (SpMV) Multiplication on Multi- and Many-Core Processors. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 713–722, May 2017.

[83] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. Dynamic Thread Block Launch: A Lightweight Execution Mechanism to Support Irregular Applications on GPUs. *Proceedings of the 42nd Annual International Symposium on Computer Architecture - ISCA '15*, pages 528–540, 2015.

[84] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. LaPerm: Locality Aware Scheduler for Dynamic Parallelism on GPUs. In *Proceedings of the 43rd International Symposium on Computer Architecture, ISCA 2016*, pages 583–595, 2016.

[85] Marc S. Orr, Bradford M. Beckmann, Steven K. Reinhardt, and David A. Wood. Fine-Grain Task Aggregation and Coordination on GPUs. *Proceedings - International Symposium on Computer Architecture*, pages 181–192, 2014.

[86] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. A Study of Persistent Threads Style GPU Programming for GPGPU Workloads. *2012 Innovative Parallel Computing, InPar 2012*, 2012.

[87] Tsung Tai Yeh, Amit Sabne, Putt Sakdhnagool, Rudolf Eigenmann, and Timothy G. Rogers. Pagoda: Fine-Grained GPU Resource Virtualization for Narrow Tasks. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 221–234, New York, NY, USA, 2017. ACM.

[88] Yi Yang, Chao Li, and Huiyang Zhou. CUDA-NP: Realizing Nested Thread-Level Parallelism in GPGPU Applications. *Journal of Computer Science and Technology*, 30(1):3–19, 2015.

[89] Guoyang Chen and Xipeng Shen. Free Launch: Optimizing GPU Dynamic Kernel Launches Through Thread Reuse. *Proceedings of the 48th International Symposium on Microarchitecture*, pages 407–419, 2015.

[90] Hancheng Wu, Da Li, and Michela Becchi. Compiler-Assisted Workload Consolidation for Efficient Dynamic Parallelism on GPU. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 534–543, May 2016.

[91] Izzat El Hajj, Juan Gomez-Luna, Cheng Li, Li Wen Chang, Dejan Milojicic, and Wen Mei Hwu. KLAP: Kernel Launch Aggregation and Promotion for Optimizing Dynamic Parallelism. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 2016-December, 2016.

[92] Xulong Tang, Ashutosh Pattnaik, Huaipan Jiang, Onur Kayiran, Adwait Jog, Sreepathi Pai, Mohamed Ibrahim, Mahmut T. Kandemir, and Chita R. Das. Controlled Kernel Launch for

Dynamic Parallelism in GPUs. In *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 649–660, Feb 2017.

[93] Yanfei Guo, Jia Rao, Changjun Jiang, and Xiaobo Zhou. FlexSlot: Moving Hadoop Into the Cloud with Flexible Slot Management. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 959–969. IEEE, 2014.

[94] Yanfei Guo, Jia Rao, ChangJun Jiang, and Xiaobo Zhou. Moving MapReduce into the Cloud with Flexible Slot Management and Speculative Execution. *IEEE Transactions on Parallel and Distributed Systems*, 28(3):798–812, 2017.

[95] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skew-Resistant Parallel Processing of Feature-Extracting Scientific User-Defined Functions. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 75–86. ACM, 2010.

[96] Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-aware Graph-structured Analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 183–193, New York, NY, USA, 2015. ACM.

[97] Benjamin C. Lee and David M. Brooks. Illustrative Design Space Studies with Microarchitectural Regression Models. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 340–351, Feb 2007.

[98] Benjamin C. Lee and David M. Brooks. Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction. *SIGPLAN Not.*, 41(11):185–194, October 2006.

[99] Engin İpek, Sally A. McKee, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. Efficiently Exploring Architectural Design Spaces via Predictive Modeling. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 195–206, New York, NY, USA, 2006. ACM.

[100] Engin İpek, Bronis R. de Supinski, Martin Schulz, and Sally A. McKee. An Approach to Performance Prediction for Parallel Applications. In *Proceedings of the 11th International Euro-Par Conference on Parallel Processing*, Euro-Par'05, pages 196–205, Berlin, Heidelberg, 2005. Springer-Verlag.

[101] Gabriel Marin and John Mellor-Crummey. Cross-architecture Performance Predictions for Scientific Applications Using Parameterized Models. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 2–13, New York, NY, USA, 2004. ACM.

[102] Laura Carrington, Allan Snavely, and Nicole Wolter. A Performance Prediction Framework for Scientific Applications. *Future Gener. Comput. Syst.*, 22(3):336–346, February 2006.

[103] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive Performance and Scalability Modeling of a Large-scale Application. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, pages 37–37, New York, NY, USA, 2001. ACM.

[104] Hitoshi Nagasaka, Naoya Maruyama, Akira Nukada, Toshio Endo, and Satoshi Matsuoka. Statistical Power Modeling of GPU Kernels Using Performance Counters. In *Proceedings*

of the 2010 International Conference on Green Computing, GREENCOMP '10, pages 115–122, Washington, DC, USA, 2010. IEEE Computer Society.

[105] Shuaiwen Song, Chunyi Su, Barry Rountree, and Kirk W. Cameron. A Simplified and Accurate Model of Power-Performance Efficiency on Emergent GPU Architectures. In *Proceedings of the 27th IEEE International Symposium on Parallel and Distributed Processing*, pages 673–686, May 2013.

[106] Gene Wu, Joseph L. Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. GPGPU Performance and Power Estimation Using Machine Learning. In *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 564–576, Feb 2015.

[107] Ying Zhang, Yue Hu, Bin Li, and Lu Peng. Performance and Power Analysis of ATI GPU: A Statistical Approach. In *Proceedings of the 6th IEEE International Conference on Networking, Architecture, and Storage*, pages 149–158, July 2011.

[108] Souley Madougou, Ana Lucia Varbanescu, Cees De Laat, and Rob Van Nieuwpoort. A Tool for Bottleneck Analysis and Performance Prediction for GPU-Accelerated Applications. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 641–652, May 2016.

[109] Timothy G. Rogers, Daniel R. Johnson, Mike O'Connor, and Stephen W. Keckler. A Variable Warp Size Architecture. *Proceedings of the 42nd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 43(3):489–501, June 2015.

[110] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. Stargazer: Automated Regression-based GPU Design Space Exploration. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems Software*, pages 2–13, April 2012.

[111] Andrew Kerr, Eric Anger, Gilbert Hendry, and Sudhakar Yalamanchili. Eiger: A Framework for the Automated Synthesis of Statistical Performance Models. In *Proceedings of the 19th International Conference on High Performance Computing*, pages 1–6, Dec 2012.

[112] Yongchao Liu, Bertil Schmidt, and Douglas L. Maskell. CUSHAW: a CUDA Compatible Short Read Aligner to Large Genomes based on the Burrows-Wheeler Transform. *Bioinformatics*, 28(14):1830–1837, July 2012.

[113] Jose Salavert Torres, Ignacio Blanquer Espert, Andres Tomas Dominguez, Vicente Hernende, Ignacio Medina, Joaquin Terraga, and Joaquin Dopazo. Using GPUs for the Exact Alignment of Short-Read Genetic Sequences by Means of the Burrows-Wheeler Transform. *IEEE/ACM Transactions Comput. Biol. Bioinformatics*, 9(4):1245–1256, July 2012.

[114] Jose Martinez, Rene Cumplido, and Claudia Feregrino. An FPGA Parallel Sorting Architecture for the Burrows-Wheeler Transform. In *Proceedings of the 2005 International Conference on Reconfigurable Computing and FPGAs* , RECONFIG '05, pages 17–, Washington, DC, USA, 2005. IEEE Computer Society.

[115] Sebastian Arming, Roman Fenkhuber, and Thomas Handl. Data Compression in Hardware - the Burrows-Wheeler Approach. In Elena Gramatová, Zdenek Kotásek, Andreas Steininger,

Heinrich Theodor Vierhaus, and Horst Zimmermann, editors, *DDECS*, pages 60–65. IEEE, 2010.

[116] Wendi Wang, Wen Tang, Linchuan Li, Guangming Tan, Peiheng Zhang, and Ninghui Sun. Investigating Memory Optimization of Hash-index for Next Generation Sequencing on Multi-core Architecture. In *Proceedings of the 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, pages 665–674, May 2012.

[117] Faraz Hach, Fereydoun Hormozdiari, Can Alkan, Farhad Hormozdiari, Inanc Birol, Evan E Eichler, and S Cenk Sahinalp. mrsFAST: a Cache-Oblivious Algorithm for Short-Read Mapping. *Nature methods*, 7(8):576–577, August 2010.

[118] Christiam Camacho, George Coulouris, Vahram Avagyan, Ning Ma, Jason Papadopoulos, Kevin Bealer, and Thomas L Maddencorresponding. BLAST+: Architecture and Applications. *BMC Bioinformatics*, 10:421, 2009.

[119] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic Local Alignment Search Tool. *J. Molecular Biology*, 215:403–410, 1990.

[120] Robert D. Bjornson, A. H. Sherman, Stephen B. Weston, N. Willard, and J. Wing. TurboBLAST(r): A Parallel Implementation of BLAST Built on the TurboHub. In *Proceedings of the 16th IEEE International Parallel & Distributed Processing Symposium*, page 8, 2002.

[121] C. Oehmen, and J. Nieplocha. ScalaBLAST: A Scalable Implementation of BLAST for High-Performance Data-Intensive Bioinformatics Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 17(8):740–749, 2006.

[122] Aaron E. Darling, Lucas Carey, and Wu-chun Feng. The Design, Implementation, and Evaluation of mpiBLAST. In *Proceedings of the 4th International Conference on Linux Clusters*, 2003. Best Paper: Applications Track .

[123] Arpith Jacob, Joseph Lancaster, Jeremy Buhler, Brandon Harris, and Roger D. Chamberlain. Mercury BLASTP: Accelerating Protein Sequence Alignment. *ACM Transactions Reconfig. Tech. and Syst.*, 1(2):1–44, 2008.

[124] K. Muriki, K. D. Underwood, and R. Sass. RC-BLAST: Towards a Portable, Cost-Effective Open Source Hardware Implementation. In *Proceedings of the 19th IEEE International Parallel & Distributed Proc. Symposium*, 2005.

[125] Atabak Mahram and Martin C. Herbordt. Fast and Accurate NCBI BLASTP: Acceleration with Multiphase FPGA-based Prefiltering. In *Proceedings of the 24th ACM International Conference on Supercomputing*, 2010.

[126] Jing Zhang, Hao Wang, Heshan Lin, and Wu-chun Feng. cuBLASTP: Fine-Grained Parallelization of Protein Sequence Search on a GPU. In *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium*, pages 251–260, 2014.

[127] Jing Zhang, Hao Wang, and Wu-chun Feng. cuBLASTP: Fine-Grained Parallelization of Protein Sequence Search on CPU+GPU. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 14(4):830–843, 2017.

[128] Cheng Ling, and Khaled Benkrid. Design and Implementation of a CUDA-Compatible GPU-based Core for Gapped BLAST Algorithm. In *Proceedings of the 10th International Conference on Computational Science*, 2010.

[129] Panagiotis D. Vouzis and Nikolaos V. Sahinidis. GPU-BLAST: Using Graphics Processors to Accelerate Protein Sequence Alignment. *Bioinformatics*, 27(2):182–188, 2011.

[130] Weiguo Liu, Bertil Schmidt, and Wolfgang Muller-Wittig. CUDA-BLASTP: Accelerating BLASTP on CUDA-enabled Graphics Hardware. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 8:1678–1684, 2011.

[131] Kaiyong Zhao and Xiaowen Chu. G-BLASTN: Accelerating Nucleotide Alignment by Graphics Processors. *Bioinformatics*, 30:1384–1391, 2014.

[132] Ning Wan, Hai-bo Xie, Qing Zhang, Kai-yong Zhao, Xiao-wen Chu, and Jun Yu. A Preliminary Exploration on Parallelized BLAST Algorithm Using GPU. *Computer Engineering & Science*, 31:98–112, 2009.

[133] Xiaodong Yu, Wu-chun Feng, Danfeng (Daphne) Yao, and Michela Becchi. O3FA: A Scalable Finite Automata-based Pattern-Matching Engine for Out-of-Order Deep Packet Inspection. In *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems*, ANCS '16, pages 1–11, New York, NY, USA, 2016. ACM.

[134] Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu-chun Feng, and Michela Becchi. Demystifying Automata Processing: GPUs, FPGAs or Micron's AP? In *Proceedings of the 2017 International Conference on Supercomputing*, ICS '17. ACM, 2017.

[135] Xiaodong Yu and Michela Becchi. GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 18:1–18:10, New York, NY, USA, 2013. ACM.

[136] Xiaodong Yu and Michela Becchi. Exploring Different Automata Representations for Efficient Regular Expression Matching on GPUs. *SIGPLAN Not.*, 48(8):287–288, February 2013.

[137] Jason Papadopoulos. *The Developer's Guide to NCBI BLAST.* NCBI, jun 2008.

[138] Zemin Ning, Anthony J. Cox, and James C. Mullikin. SSAHA: a Fast Search Method for Large DNA Databases. *Genome Research*, 11:1725–1729, Oct 2001.

[139] Hugh E. Williams. Cafe: An Indexed Approach to Searching Genomic Databases. In *Proceedings of the 21st International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 389–389, New York, NY, USA, 1998.

[140] W. James Kent. BLAT–the BLAST-like Alignment Tool. *Genome Research*, 12:656–664, apr 2002.

[141] Aleksandr Morgulis, George Coulouris, Yan Raytselis, Thomas L. Madden, Richa Agarwala, and Alejandro A. Schäffer. Database Indexing for Production MegaBLAST Searches. *Bioinformatics*, 24:1757–1764, 2008.

[142] Michael Cameron. *Efficient Homology Search for Genomic Sequence Databases*. PhD thesis, School of Computer Science and Information Technology, RMIT University, Nov 2006.

[143] Xiao Lei Chen. *A Framework for Comparing Homology Search Techniques*. PhD thesis, School of Computer Science and Information Technology, RMIT University, 2004.

[144] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sept 1972.

[145] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[146] Jatin Chhugani, Anthony D Nguyen, Victor W Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture. *Proceedings of the VLDB Endowment*, 1(2):1313–1324, 2008.

[147] Nadathur Satish, Mark Harris, and Michael Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. In *Proceedings of the 2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–10. IEEE, 2009.

[148] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 351–362, New York, NY, USA, 2010. ACM.

[149] Hiroshi Inoue and Kenjiro Taura. SIMD-and Cache-friendly Algorithm for Sorting An Array of Structures. *Proceedings of the VLDB Endowment*, 8(11):1274–1285, 2015.

[150] Darko Bozidar and Tomaz Dobravec. Comparison of Parallel Sorting Algorithms. *CoRR*, abs/1511.03404, 2015.

[151] Kaixi Hou, Hao Wang, and Wu-chun Feng. ASPaS: A Framework for Automatic SIMDization of Parallel Sorting on x86-based Many-core Processors. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 383–392. ACM, 2015.

[152] Kaixi Hou, Weifeng Liu, Hao Wang, and Wu-chun Feng. Fast Segmented Sort on GPUs. In *Proceedings of the 2017 ACM International Conference on Supercomputing*, ICS '17, pages 12:1–12:10, New York, NY, USA, 2017. ACM.

[153] Kaixi Hou, Hao Wang, and Wu-chun Feng. A Framework for the Automatic Vectorization of Parallel Sort on x86-based Processors. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2018.

[154] Heng Li and Nils Homer. A Survey of Sequence Alignment Algorithms for Next-Generation Sequencing. *Briefings in bioinformatics*, 11(5):473–83, Sep 2010.

[155] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen and Jun Wang. SOAP2: an Improved Ultrafast Tool for Short Read Alignment. *Bioinformatics (Oxford, England)*, 25(15):1966–7, August 2009.

[156] Intel VTune Amplifier XE 2013, Product Brief, August 2012.

[157] A Map of Human Genome Variation from Population-Scale Sequencing. *Nature*, 467(7319):1061–1073, October 2010.

[158] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic Local Alignment Search Tool. *Journal of molecular biology*, 215(3):403–410, October 1990.

[159] Andréa Matsunaga, Maurício Tsugawa, and José Fortes. Cloudblast: Combining MapReduce and Virtualization on Distributed Resources for Bioinformatics Applications. In *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, pages 222–229. IEEE, 2008.

[160] Heshan Lin, Xiaosong Ma, Jeremy Archuleta, Wu-chun Feng, Mark Gardner, and Zhe Zhang. Moon: Mapreduce on opportunistic environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 95–106. ACM, 2010.

[161] Wei Lu, Jared Jackson, and Roger Barga. AzureBlast: A Case Study of Developing Science Applications on the Cloud. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 413–420. ACM, 2010.

[162] Kanak Mahadik, Somali Chaterji, Bowen Zhou, Milind Kulkarni, and Saurabh Bagchi. Orion: Scaling Genomic Sequence Matching with Fine-Grained Parallelization. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 449–460, November 2014.

[163] Sitao Wu, Weizhong Li, Larry Smarr, Karen Nelson, Shibu Yooseph, and Manolito Torralba. Large Memory High Performance Computing Enables Comparison Across Human Gut Microbiome of Patients with Autoimmune Diseases and Healthy Subjects. In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*, XSEDE '13, pages 25:1–25:6, New York, NY, USA, 2013. ACM.

[164] Xie Wu. Improving the Performance and Precision of Bioinformatics Algorithms. Master's thesis, University of Maryland, Aug 2008.

[165] A. Brahme. *Comprehensive Biomedical Physics*, volume 6. Elsevier, first edition, Oct 2014.

[166] Jianzhi Zhang. Protein-Length Distributions for the Three Domains of Life. *Genome Analysis*, 16:107–109, 2000.

[167] Axel Tiessen, Paulino Pérez-Rodríguez, and Luis José Delaye-Arredondo. Mathematical Modeling and Comparison of Protein Size Distribution in Different Plant, Animal, Fungal and Microbial Species Reveals a Negative Correlation between Protein Size and Protein Number, thus Providing Insight into the Evolution of Proteomes. *BMC Research Notes*, 5:85, Feb 2012.

[168] Temple F. Smith and Michael R Waterman. Identification of Common Molecular Subsequences. *J. Molecular Biology*, 147(1):195–197, 1981.

[169] Modern GPU. `http://nvlabs.github.io/moderngpu/`.

[170] NCBI Genbank. `ftp://ftp.ncbi.nlm.nih.gov/genbank/`, 2015.

[171] CUB Project. `http://nvlabs.github.io/cub/`.

[172] Da Li, Hancheng Wu, and Michela Becchi. Nested Parallelism on GPU: Exploring Parallelization Templates for Irregular Loops and Recursive Computations. In *Proceedings of the 44th International Conference on Parallel Processing*, pages 979–988, Sept 2015.

[173] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference and Prediction*. Springer, 2 edition, 2009.

[174] Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, Oct 2001.

[175] DIMACS Implementation Challenges. `https://www.cc.gatech.edu/dimacs10`, 2012.

[176] Nabeel Mohamed, Nabanita Maji, Jing Zhang, Nataliya Timoshevskaya, and Wu-Chun Feng. Aeromancer: A Workflow Manager for Large-Scale MapReduce-Based Scientific Workflows. In *Proceedings of the 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 739–746, Sept 2014.

[177] Amine Abou-Rjeili and George Karypis. Multilevel Algorithms for Partitioning Power-Law Graphs. In *Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium*, pages 10–pp. IEEE, 2006.

[178] Dominique LaSalle and George Karypis. Multi-threaded Graph Partitioning. In *Proceedings of the 27th IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 225–236. IEEE, 2013.

[179] Heshan Lin, Xiaosong Ma, Wuchun Feng, and Nagiza F Samatova. Coordinating Computation and I/O in Massively Parallel Sequence Search. *Parallel and Distributed Systems, IEEE Transactions on*, 22(4):529–543, 2011.

[180] Jing Zhang, Sanchit Misra, Hao Wang, and Wu-chun Feng. Eliminating Irregularities of Protein Sequence Search on Multicore Architectures. In *Proceedings of the 31st IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2017.

[181] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: a Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[182] Saba Sehrish, Grant Mackey, Jun Wang, and John Bent. MRAP: A Novel MapReduce-based Framework to Support HPC Analytics Applications with Access Patterns. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 107–118. ACM, 2010.

[183] Wei Jiang, Vignesh T Ravi, and Gagan Agrawal. A Map-Reduce System with An Alternate API for Multi-core Environments. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 84–93. IEEE Computer Society, 2010.

[184] Tim Kaldewey, Eugene J Shekita, and Sandeep Tata. Clydesdale: Structured Data Processing on MapReduce. In *Proceedings of the 15th international conference on extending database technology*, pages 15–25. ACM, 2012.

[185] Yi Wang, Wei Jiang, and Gagan Agrawal. SciMATE: A Novel MapReduce-like Framework for Multiple Scientific Data Formats. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 443–450. IEEE, 2012.

[186] Matti Niemenmaa, Aleksi Kallio, André Schumacher, Petri Klemelä, Eija Korpelainen, and Keijo Heljanko. Hadoop-BAM: Directly Manipulating Next Generation Sequencing Data in the Cloud. *Bioinformatics*, 28(6):876–877, 2012.

[187] Yi Wang, Arnab Nandi, and Gagan Agrawal. SAGA: Array Storage as a DB with Support for Structural Aggregations. In *Proceedings of the 26th international conference on scientific and statistical database management*, page 9. ACM, 2014.

[188] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, 2012.

[189] Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel. Operator Language: A Program Generation Framework for Fast Kernels. In *Domain-Specific Languages*, pages 385–409. Springer, 2009.

[190] Steven J Plimpton and Karen D Devine. MapReduce in MPI for Large-Scale Graph Algorithms. *Parallel Computing*, 37(9):610–632, 2011.

[191] Yin Huai, Ashutosh Chauhan, Alan Gates, Gunther Hagleitner, Eric N Hanson, Owen O'Malley, Jitendra Pandey, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. Major Technical

Advancements in Apache Hive. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1235–1246. ACM, 2014.

[192] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *Proceedings of the VLDB Endowment*, 6(10):817–828, August 2013.

[193] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F Bacon, and Stephen J Fink. Compiling a High-Level Language for GPUs:(via Language Support for Architectures and Compilers). In *ACM SIGPLAN Notices*, volume 47, pages 1–12. ACM, 2012.

[194] Hao Wang, Weifeng Liu, Kaixi Hou, and Wu-chun Feng. Parallel Transposition of Sparse Data Structures. In *Proceedings of the 2016 International Conference on Supercomputing*, page 33. ACM, 2016.

[195] Weifeng Liu and Brian Vinter. Csr5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 339–350. ACM, 2015.

[196] Naser Sedaghati, Te Mu, Louis-Noël Pouchet, Srinivasan Parthasarathy, and P. Sadayappan. Automatic Selection of Sparse Matrix Representation on GPUs. In *Proceedings of the 29th ACM International Conference on Supercomputing*, ICS '15. ACM, 2015.

[197] Xiaodong Yu, Hao Wang, Wu-chun Feng, Hao Gong, and Guohua Cao. An Enhanced Image Reconstruction Tool for Computed Tomography on GPUs. In *Proceedings of the Computing Frontiers Conference*, CF'17, pages 97–106, New York, NY, USA, 2017. ACM.

[198] Xiaodong Yu, Hao Wang, Wuchun Feng, Hao Gong, and Guohua Cao. cuART: Fine-Grained Algebraic Reconstruction Technique for Computed Tomography Images on GPUs. In *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 165–168, May 2016.

[199] S. N. A. Project. Stanford Large Network Dataset Collection. `http://snap.stanford.edu/data/`.

[200] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack J. Dongarra. *Performance, Design, and Autotuning of Batched GEMM for GPUs*, pages 21–38. Springer International Publishing, Cham, 2016.

[201] Tingxing Dong, Azzam Haidar, Piotr Luszczek, Stanimire Tomov, Ahmad Abdelfattah, and Jack J. Dongarra. MAGMA Batched: A Batched BLAS Approach for Small Matrix Factorizations and Applications on GPUs. 2016.

[202] Erich Elsen. Optimizing RNN Performance - Part I: Investigating Performance of GPU BLAS Libraries, 2015.