Comparaisons de séquences biologiques sur architecture massivement multi-cœurs

Bioinformatics Sequence Comparisons on Manycore Processors

THÈSE

présentée et soutenue publiquement le 21 décembre 2012

pour l'obtention du

Doctorat de l'Université de Lille 1 – Sciences et Technologies

(spécialité informatique)

par

Tuan Tu Tran

Composition du jury

Rapporteurs :	Dominique Lavenier, DR CNRS	IRISA, CNRS, Univ. Rennes 1, INRIA
	Bertil Schmidt, Professor	Universität Mainz, Allemagne
Examinateurs :	Mathieu Giraud, CR CNRS, *co-encadrant de thèse*	LIFL, CNRS, Univ. Lille 1, INRIA
	Arnaud Lefebvre, MdC	LITIS, Univ. Rouen
	Nouredine Melab, Professeur	LIFL, CNRS, Univ. Lille 1, INRIA
	Jean-Stéphane Varré, Professeur, *directeur de thèse*	LIFL, CNRS, Univ. Lille 1, INRIA

UNIVERSITÉ DE LILLE 1 – SCIENCES ET TECHNOLOGIES
ÉCOLE DOCTORALE SCIENCES POUR L'INGÉNIEUR
Laboratoire d'Informatique Fondamentale de Lille — UMR 8022
U.F.R. d'I.E.E.A. – Bât. M3 – 59655 VILLENEUVE D'ASCQ CEDEX
Tél. : +33 (0)3 28 77 85 41 – Télécopie : +33 (0)3 28 77 85 37 – email : direction@lifl.fr

History of this document

- 12 March 2013: final version
  - minor updates throughout the document
  - acknowledgments
- 6 December 2012:
  - corrections from the reporters
  - updates to chapter 6, new sections 6.4.4 and 6.4.5
  - minor updates throughout the document
- 1 November 2012: minor updates to chapter 5
- 9 October 2012: document sent to the reporters

This PhD thesis can be found online at http://www.lifl.fr/bonsai/doc/phd-tuan-tu-tran.pdf.

March 12, 2013

Contents

Acknowledgment

This thesis would not have completed without an excellent direction of my two supervisors, Prof. Jean-Stéphane Varré and Dr. Mathieu Giraud at the BONSAI team. During my research project, I received tremendous help and valuable advice from both of them. Their accompaniment has been a great source of inspiration and motivation on my way learning to be a researcher. I'm extremely indebted to both of them.

I would also like to express my sincere gratitude to Prof. Dominique Lavenier and Prof. Bertil Schmidt for their valuable insights which helped me upgrade my thesis. Heartfelt thanks also go to Prof. Nouredine Melab and Dr. Arnaud Lefebvre for their presence and valuable comments during my thesis defense.

I appreciate a lot the help, kindness and warm friendship of my colleagues in the BONSAI team, in INRIA Lille - Nord Europe, France, as well as those in the High Performance Computing Center, Hanoi University of Science and Technology, Vietnam.

I will never forget the warm support and sincere help from my Vietnamese friends in Lille, who have been like my brothers and sisters.

Last but not least, I would like to express my greatest gratitude to my family and my fiancée, for being my resource of strength so that I can overcome all the difficulties and the challenges of the PhD journey. I am proud that I am worthy with your love.

Introduction

In recent years, the trend of producing *processors with multiple cores*, such as multicore central processing units (CPUs) and manycore graphic processing units (GPUs), has made parallel computing more and more popular. The computing power and the parallel architecture of today's GPUs can be compared as those of the supercomputers of the last decade. Hundreds of industrial and research applications have been mapped onto GPUs.

In bioinformatics, with the advances in *High-Throughput Sequencing* technologies (HTS), the data to analyze grows even more rapidly than before, requiring efficient algorithms and execution platforms. Many bioinformatics problems are related to *similarities studies* between genetic sequences, and require efficient tools which can compare sequences made of hundred millions to billions base pairs. Using common *short words* called *seeds*, then extending to full alignments, the *seed-based heuristic* alignment tools have shown improved speed with relatively high accuracy. This strategy is now one of the mainstream approaches to design applications over large genetic databases.

⋆ ⋆ ⋆

This thesis focuses on the design of **parallel data structures and algorithms** that can be **efficiently mapped on the GPUs to solve the problem of approximately multiple pattern matching**. This problem can be used in the "neighborhood filtering phase" of the seed-based heuristics tools to study the similarities in genetic data. This thesis contains 7 chapters.

- **Chapter 1** gives the **background knowledges** related to massively parallel manycore computing in bioinformatics. After the definition and the introduction of general features of modern manycore processors, such as the GPUs, this chapter provides the basics of OpenCL, which is the programming language used through the thesis. The bioinformatics applications which have been mapped onto GPUs in the recent years are also presented.
- The main method of creating indexes for large genetic sequences is then introduced in **Chapter 2**: "Given a large genetic sequence, how can we keep the occurences of each seed so that their neighborhoods can be retrieved and compared efficiently?" We choose the "neighborhood indexing approach", which consists in storing the neighborhoods along with the position of each seed occurrence in the sequence. This chapter discusses the **framework of neighborhood indexing** with two main problems: the data structures to keep the neighborhoods and the algorithms to do the approximate matching in the

requirement of efficient implementation on GPUs. The core of this thesis is then built on this framework:

  - **Chapter 3** implements the **direct neighborhood matching** approach which stores the neighborhoods of each seed occurrence as a flat list. The input pattern is directly compared with the elements of the list, either using **mflBPR** (our approximate pattern matching bit-parallel algorithm for a *set of fixed length words* adapted from the work of [Wu and Manber, 1992a]) or applying a traditional **binary search (BS)** algorithm on the set of degenerated patterns.
  - **Chapter 4** proposes another solution in which the neighborhood list of each seed is **indexed with perfect hash functions (PH)**. Thanks to the BDZ_{PH} algorithm [Botelho, 2008], a neighborhood can be retrieved and compared in constant time, at a very small cost in storage space.
  - The **performance results** of these three solutions (mflBPR, BS, and PH) are analyzed and discussed in **chapter 5**.

- The approach of neighborhood indexing is further developed into **MAROSE: a prototype of read mapper for manycore processors**, which is the content of **Chapter 6**. It is a direct application of our work to build a potential high-performance tool to map genetic sequences onto genomes.
- Finally, **chapter 7** gives **conclusions** and proposes some **perspectives** for future works.

Chapter 1

Background

The background of this thesis is concerned with massively parallel manycore computing in bioinformatics. The Graphics Processing Units (GPUs) will be our representative of the manycore processors and the Open Computing Language (OpenCL)[1] will be our programming language. The first section deals with the evolution of computer architecture and presents the hardware and programming models of GPUs. The second section is a brief technical introduction of OpenCL, and the third section introduces some applications of GPUs in bioinformatics.

Contents

[1] OpenCL is a trademark of Apple Inc., used under license by Khronos.

1.1 New hardware models, new programming languages

This section explains the trend of manufacturing multicore and manycore processors since the early 2000s and the massively computing potential of the GPUs. It also introduces the technical improvements that leads to the interest on general purpose computation on GPU (GPGPU).

1.1.1 From traditional processors to multicore and manycore processors

A Central Processing Unit (CPU), or a computer processor, plays a role of a *"brain"* in a computer: it dispatches the input instructions, loads and stores the input data, executes the instruction with the corresponding data and stores the output results. A CPU can be traditionaly programmed within a "serial programming model": the instructions in a program are dispatched and executed sequentialy according to their orders.

The famous *Moore's Law* states that *the number of transitors on a chip doubles every two years* [Moore, 1965, Moore, 1975]. This was consistent with what was observed since 1965, and *"more than a natural observation, this is a self-fullfilling prophecy that drives the semiconductor industry"* [Varré et al., 2011, Chapter 1.1].

The continuous improvement of CPU computer power has always been driven by the Moore's Law, enabling more complex operators and better architectures (instruction pipelines, superscalar architectures, out-of-order executions...).

However, between the years 1965 and 2000, the *higher frequencies* are another important factor that also explained the improvement of computing power, doubling every 18 – 24 months in this period [Shalf et al., 2009]. Since the beginning of 2000s, the frequency of CPUs has not increased anymore. Indeed, increasing the clock frequency is more and more difficult and expensive due to heat dissipation issues [Shalf et al., 2009].

New hardware models. However, the famous Moore's Law is still *"alive"* regarding the increase of the number of transistors. How can be these transistors turned into computed power?

- The first solution is to multiply the *cores* on the chip [Shalf et al., 2009, Asanovic et al., 2006, Asanovic et al., 2009]. It has led to the developements of **multicore CPUs** since the last decade. The mainstream CPU now evolves more with a multiplication of the core number than with an improvement of the core architecture. *"The industry buzzword "multicore" captures the plans of doubling the number of "standard core" per die with every semiconductor process generation, starting with the single processor."* [Shalf et al., 2009, page 43]. A good example of a current CPU is depicted on Table 1.1: the CPUs of the Intel Nehalem microarchitecture has up to 8 cores that are full-featured processors sharing a common die and the L3 cache.

- Another method is to *"adopt the "manycore" trajectory, which employs simpler core running at modestly lower clock frequencies. Rather than progressing from 2 to 4 to 8 cores with the multicore approach, a manycore design would start with hundreds of core and progress geometrically to thousand of cores over time"* [Shalf et al., 2009, page 43]). In the recent years, the **Graphics Processing Units (GPUs)**, which are today the main representative of the manycore processors, have gained a high interest of developments,

firstly to satisfy the needs of the game and cinema industries. Table 1.1 describes two examples of GPU: the NVIDIA Fermi family and the AMD Evergreen family. They both show a higher number of "cores" and "processing elements" that will be discussed later. Although GPU programming is still a complex task, nowadays GPU applications are not limited to the graphics processing fields.

The two solutions are today not so different: *"recent trends blur the line between GPUs and CPUs: CPUs have more and more cores, and cores in GPUs have more and more functions"* [Varré et al., 2011, Chapter 1.2].

Parallel and programming models. These devices, multicore CPU and manycore/GPU processors, have challenged the traditional "serial programming" method and may be developed with several strategies for parallelism:

- *Inter-core parallelism* is all that will be achieved between independent cores (and is somewhat analogous to what exists in a grid). This parallelism can be defined in the programs by using explicit multithread programming or with the help of high-level frameworks such as OpenMP. Inter-core parallelism can also happen with the automatic execution of differents tasks simultaneously on cores of the same machine thanks to the scheduler of the operating system. This parallelism naturally applies to both multicore CPU and manycore/GPU processors.
- *Intra-core level* involves parallelism inside a core – for example through a SIMD (Simple Instruction Multiple Data) model, which means several Processing Elements controlled by a same instruction flow. Another example is *out-of-order executions* where independent instructions can be executed simultaneously, regardless of their orders in the program. Again, this parallelism applies to both multicore CPU and manycore/GPU processors. However, *it is fundamental for the performance of GPU/manycore processors* because they have a much higher PE/cores ratio than a mainstream CPU (Table 1.1).

To further exploit the advantages of the modern GPUs and CPUs, there is thus a need for new programming languages that can adapt to the variety of processor architectures, dealing with the ever increasing number of cores and the heterogeneity of the computing environments (see Section 1.1.3).

Task and data parallelism. Generally, a program consists of one or multiple computational tasks. When different tasks process different chunks of data in an independent way, these tasks can run concurrently. If the computing system has enough computional resources so that each task can map onto one independent processing elements, the concurrent tasks can run simultaneously: this is the *task parallelism*.

This parallelism can be used both in a *corse-grained* way (inter-core parallelism), dispatching tasks on independent cores. Moreover, if the computations are very regular and can be described with a unique instruction flow, it is further possible to use a *fine-grained* intra-core parallelism, one data chunk being be further divided to be processed simultaneously by different processing elements (Figure 1.2): this is the *data parallelism*. In the case of the GPU, the large number of available processing elements will thus mean a high computing power.

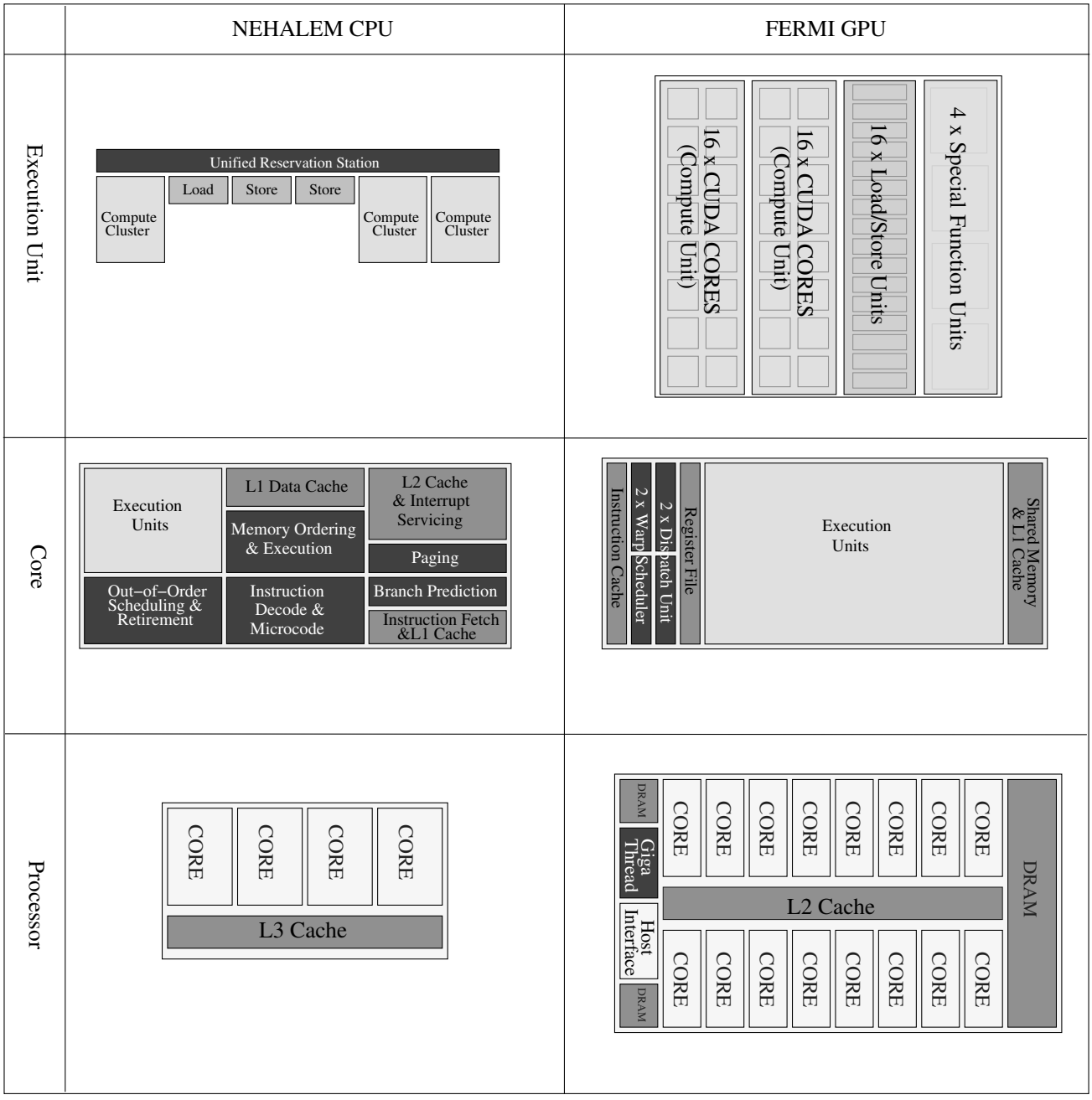

Figure 1.1: Architecture comparation between Intel NEHALEM CPU and NVIDIA Fermi GPU at three levels: processor, core and execution unit. *(Based on [Semin, 2009] and [NVIDIA Corp, c]).*

	NVIDIA Fermi GPU	**AMD Evergreen GPU**	**Intel Nehalem CPU**
Compute Resource			
Compute Unit (Name, Max Number)	Streaming Multiprocessor 16	SIMD Engine 20	Thread 16
Processing Element (Name, Max Number)	CUDA cores 16 x 32-wide SIMD	Processing Element 20 x 16-wide SIMD x 5-wide VLIW	Thread 16x2x4-wide SIMD
Total	512	1600[a]	108
Max Clock Frequence (MHz)	1401	850	2270
In-core Memory			
Shared Memory (KB)	48 or 16	32	x
L1 Cache (KB)	16 or 48	8	64
L2 Cache (KB)	x	x	256
Out-core Memory			
L2 Cache (KB)	768	512	x
L3 Cache (MB)	x	x	4 - 12
Processor Memory			
Speed (MHz) Max Size (GB)	1848 6	1200 1	x x
Type	GDDR5	GDDR5	x

Table 1.1: Technical features of NVIDIA GPU, AMD Evergreen GPU and Intel Nehalem CPU.

[a]The number of processing elements here is based on the official documents of the hardware vendor (see page 22 for more details)

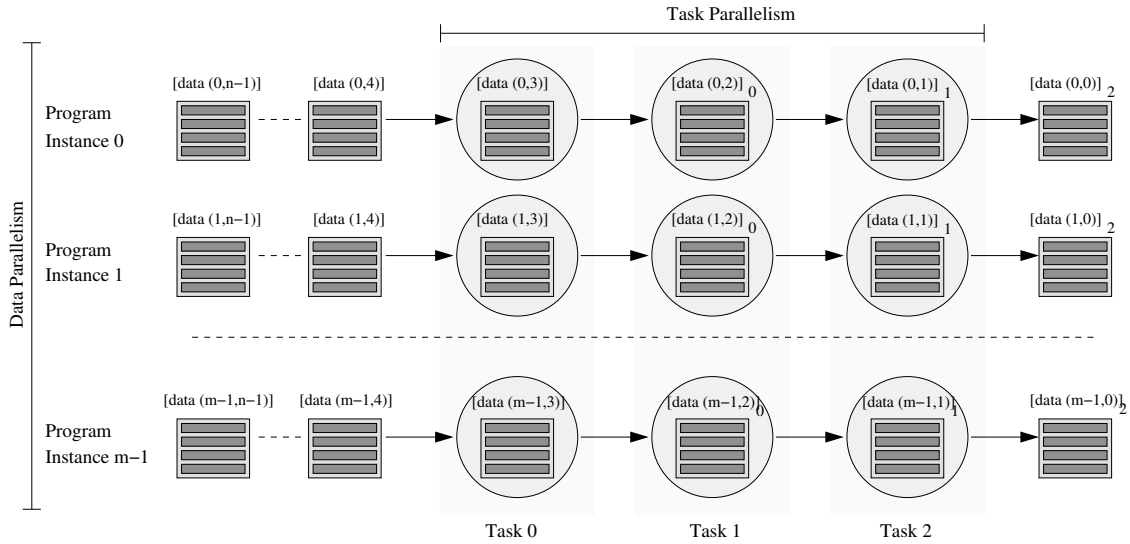

Figure 1.2: The Task parallelism and the Data parallelism. The program is launched as m instances, processing n data chunk groups simultaneously. The data chunks in each group are being streamed into the program to process sequently by all the tasks in the program. $[data(i,j)]_k$ means the data chunk j of group i after being processed by task 0, task 1, ..., task k.

⋆ ⋆ ⋆

According to the technical definition of NVIDIA, *"a GPU is a single chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second"*[2]. Initially, each type of "engines" in GPU was responsible for one specific task such as vertex shading, pixel shading, etc. These tasks are linked sequentially, as the output of one is the input of another, forming the "graphics pipeline". The main role of the GPUs is to efficiently process the huge volume of input graphics data on its embeded graphics pipeline.

This family of processors were created to serve the graphics processing tasks, but for the last ten years they have been also widely used with other types of computations, especially to accelerate the scientific applications. The usage of GPUs for tasks others than graphics processings are called the **General Purpose Computation on GPU (GPGPU)**[3].

As the GPU is today the most available manycore processor, **it is choosen to be the focus of this thesis**. The following sections further explain why the GPU has gained much interest in the current years.

[2]Information and definition can be found in the website of NVIDIA, at: http://www.nvidia.com/object/gpu.html and at http://www.nvidia.com/page/geforce256.html

[3]The collection of the bioinformatics applications which are implemented on GPUs is presented and studied in Section 1.3. For the GPGPU in the other fields, the NVIDIA CUDA Zone (http://developer.nvidia.com/category/zone/cuda-zone) and the Research category of the GPGPU website (http://gpgpu.org/category/research) give many links to CUDA applications.

1.1.2 Improvements in GPU architecture

The widely used *graphics processing algorithms* in the industry usually contain a large number of calculating operations executed in the same instruction flow. The GPU was primarily designed to match these requirements as the majority of the transitors in a core is used for the execution units (Figure 1.1). Moreover, the memory bandwidth of the video memory inside the GPU is very high[4], sufficent for the huge number of data read/write accesses required by a parallel execution of graphic operations.

However, in the GPUs of the first generation, the stages in the graphics pipeline were not programmable and could only be *configurated*. It means that these stages operated fixed programs, and only the input arguments could be changed. The GPU development rapidly reached the second generation, in which some pipeline stages can be *programmed*. These programmable stages, called the *shaders*[5], are usually classified into two groups: for vertex processing (vertex shader) and for pixel fragment processing (pixel shader or fragment shader) [Blythe, 2006].

In the first and second generations, the cores inside the GPUs are organised as discrete groups, relating to each type of shaders. The main advantage of this architecture is that the cores in the same group have the specialized designs and instruction set in order to maximize the processing capabilities. But it can also be a serious problem if there is the disbalance in the computation requirements at each stage. The free cores in one group can not be used to execute the shaders in other groups, causing a waste in computional resources [Owens et al., 2008].

In the years 2006 and 2007, there was a number of changes in the graphics processing field, from both the software and hardware sides, that directed the GPUs to the next generation with **unified shader architecture**. Some facts illustrate this trend:

- Starting from the Direct3D's Shader Model 4.0 or the OpenGL's GLSL 3.3, the different shader types share the *same instruction set*;
- Starting from the AMD Radeon HD 2000 or the NVIDIA 8000 series, the cores in the GPUs have the *same hardware design* and can thus be used for any type of shaders.

With these changes, unified shaders are now more and more flexible, being capable of executing a wide range of different codes.

1.1.3 New programming languages for GPU and manycore processors

Along with the development of hardware architectures, there was also a number of changes in GPU and manycores programming languages. Even when GPUs were only used for graphics computations, there was an evolution from the initial assembly languages to more C-like languages and APIs (such as NVIDIA Cg, OpenGL GLslang, Microsoft HLSL, Direct3D...). However, these languages were only popular among the experts in graphics processing with an extensive knowledge of the librairies and the hardware features [Buck et al., 2004].

The release of Brook in 2004 as *"a system for general-purpose computation on programmable graphics hardware"* [Buck et al., 2004, Abstract] can be considered as one of the first attempts to

[4] A comparation (from NVIDIA) between the memory bandwidth between the some GPUs and CPUs, from 2003 to 2010, can be found in [NVIDIA Corp, 2012, page 8]

[5] In terms of graphics processing, "shader" means "graphics functions". For example, the definition of the "Pixel shader" in the website of NVIDIA is that: "*A Pixel Shader is a graphics function that calculates effects on a per-pixel basis.*" (http://www.nvidia.com/object/feature_pixelshader.html)

make GPU programming easier, for both the graphics processing experts *and the programmers from the other fields*. The early works to map scientific applications onto the GPUs usually used Brook, as in [Charalambous et al., 2005] or [Horn et al., 2005].

CUDA. Two years later[6], the emergency of NVIDIA's Compute Unified Device Architecture (CUDA) [NVIDIA Corp, a] rapidly accelerated the development of GPGPU trend and caused an explosion of GPGPU publications in the recent years.

To implement applications on CUDA, developers can use programming languages such as C for CUDA, HLSL, OpenCL, etc [NVIDIA Corp, 2009a]. Up to now, C for CUDA is still the most widely used GPU programming languages with a lot of supports:

- Helpful toolkits[7], including integrated development environment with debuggers and profilers,
- Optimized programming libraries for common functions in high-performance computing, such as cuFFT (Fast Fourier Transform), or cuBLAS (Basic Linear Algebra Subroutine),
- Plenty of guides and documents, hundreds of source code examples, and a wide user community.

It should be noted that in this thesis, we simply use the terminology "CUDA" for "C for CUDA", for example when describing the applications in Sections 1.3 and 6.2.3.

However, C for CUDA only allows the GPGPU applications to be executed on NVIDIA hardwares: there is a need for a more general standard that can program in a heterogenous enviroment.

OpenCL. The Open Computing Languge (OpenCL) *"is an open industry standard for programming a heterogeneous collection of CPUs, GPUs and other discrete computing devices organized into a single platform"* [Khronos Group, 2010, p.21]. Since 2008, OpenCL has the aim of becoming an open standard for heterogenous computing, including for GPGPU.

OpenCL is now managed by the Khronos Group [Khronos Group, 2008] and is supported by many companies and institutions. Up to the middle of 2012, there are 5 implementations of OpenCL:

- NVIDIA: NVIDIA GPU Computing SDK, for NVIDIA GPUs [NVIDIA Corp, d]
- AMD: AMD Accelerated Parallel Processing SDK, for AMD GPUs and multi-core CPUs [AMD Inc,]
- Apple: As a feature of operating system from Mac OS X v10.6, Snow Leopard [Apple Inc,]
- IBM: OpenCL Development Kit, for PowerPC CPU [IBM,]
- Intel: Intel OpenCL SDK, for multi-core CPU [Intel Corp, a]

The details of OpenCL will be presented in the next section.

[6] NVIDIA announced CUDA in November 2006, released the first beta CUDA SDK in the first quater of 2007
[7] http://developer.nvidia.com/cuda/cuda-toolkit

Other languages. Another category of manycore programming solution try, like OpenMP, to mix regular Fortran, C/C++ code with directives, such as OpenHMPP[8], and, more recently, OpenACC[9]. Other languages may be proposed in the following years as the field is rapidely evolving.

⋆ ⋆ ⋆

At the beginning of this thesis, in 2009–2010, we decided to use OpenCL since it was a promising standard. Indeed, we managed to run the same OpenCL code both on NVIDIA GPUs, on CPUs through the AMD SDK, and also (to a lesser extent) on some AMD GPUs (see on page 118). The following section will thus present in more the details the OpenCL architecture. However, many of these concepts can also apply to other existing languages (such as C CUDA) and potentially to future languages for manycore processors programming.

Remarks on C for CUDA. On the implementation of NVIDIA, it should be noted that both OpenCL and the progamming language "C for CUDA" function as a "device level programming interface" for the CUDA parallel computing architecture [NVIDIA Corp, 2009a]. The main difference is that while OpenCL interacts with the CUDA driver through the "OpenCL driver", C for CUDA directly interacts with the CUDA driver.

C for CUDA is usually "ahead" OpenCL as it is designed specially for the NVIDIA GPUs, while OpenCL is an open standard, which is limited to common features of modern manycore processors. It means that, on the homogenous NVIDIA platform, OpenCL may not be as efficient as C for CUDA if the applications require advanced features such as the remote direct memory access (RDMA) between GPUs or the dynamic parallelism of CUDA 5 [Harris, 2012].

All the algorithms and data structures proposed in this thesis are implemented with fundamental concepts. But when these algorithms and data structures are applied to a real application (see Chapter 6), some implementations with C for CUDA could be slightly more efficient. Nevertheless, we decided to keep OpenCL for his portability.

In general, we can say that C for CUDA benefits from specialized NVIDIA platforms, while OpenCL has the advantages of portability over heterogenous platforms.

1.2 Manycore high-performance computing with OpenCL

The OpenCL architecture consists of 4 models, which will be further discussed in the next sections:

- Platform model (Section 1.2.1): This section will describe the general architecture of an OpenCL compute platform as a host with one or multiple "computing devices", and explain the hierarchy of the computing elements inside a device and the mapping of OpenCL platforms onto different processor architectures;

[8] http://www.openhmpp.org/en/OpenHMPPConsortium.aspx
[9] http://www.openacc-standard.org/

- Execution model (Section 1.2.2): This section will describe how a computing kernel can be run as multiple instances on the platform and explain the portability of an OpenCL program on different type of computing devices;
- Memory model (Section 1.2.3): This section wll present different types of memory regions which can be used by an OpenCL program and describe how these regions are mapped onto the physical parts of the device;
- Programming model (Section 1.2.4): This section will introduce data-parallelism and task-parallelism as two available programming models for the developement of OpenCL applications.

Finally, Section 1.2.5 will explain some guidelines for efficient manycore programming. Again, these guidelines are not limited only to OpenCL and can be applied to any applications that run on current manycore processors.

OpenCL platforms used in this thesis. In order to clarify how the OpenCL standard run on heterogenous computing architectures from different vendors, we will use three examples: the NVIDIA's Fermi generation GPUs [NVIDIA Corp, c], the AMD's Evergreen generation GPUs [AMD Inc, 2011b] and the Intel's Nehalem microarchitecture CPUs [Semin, 2009]. These selections are consistent with the hardwares used for the experiments in this thesis: an NVIDIA GeForce GTX 480 GPU, an ATI Radeon HD5870 GPU and an Intel Xeon E5520 CPU.

This means that, on the five OpenCL implementations cited in page 18), only the NVIDIA's and the AMD's implementations will be introduced and analysed (Figure 1.3):

- NVIDIA implements the OpenCL standard as a *"device level programming interface"* for the CUDA parallel computing architecture [NVIDIA Corp, 2009a, NVIDIA Corp, 2012].
- AMD implements the OpenCL standard in the AMD Accelerated Parallel Processing SDK (AMD APP SDK) [AMD Inc, 2011a], which supports both AMD GPUs and multi-cores CPUs from any vendor.

Comparing these three platforms in the light of the OpenCL architecture allows us to explain the difference of the hardware models between these GPU/CPU processors.

Remarks on the use of terminologies. Up to the mid 2012, the OpenCL standard is still being developped, so the terminologies may not be stable because different papers use differents terminologies. There are also differences between the documents of the same vendor. For example, in a technical white paper of AMD, published in june 2011, [AMD Inc, 2011c], the *streaming core (SC)* was considered as the *processing element (PE)*. But in the AMD APP SDK OpenCL programming guide [AMD Inc, 2011a], published in december 2011, the *SC* is no longer considered as the basic *PE*. Logically, the *SC* is further divided into other levels with each sub-SC containing which are considered as the *PE*.

In this thesis, the OpenCL specification of the Khronos Group [Khronos Group, 2010][10], the book "Heterogenous Computing with OpenCL" [Gaster et al., 2011], the AMD APP SDK

[10] In november 2011, the Khronos Group released the OpenCL 1.2 specification, but all the implementation used in this thesis are based on OpenCL 1.1.

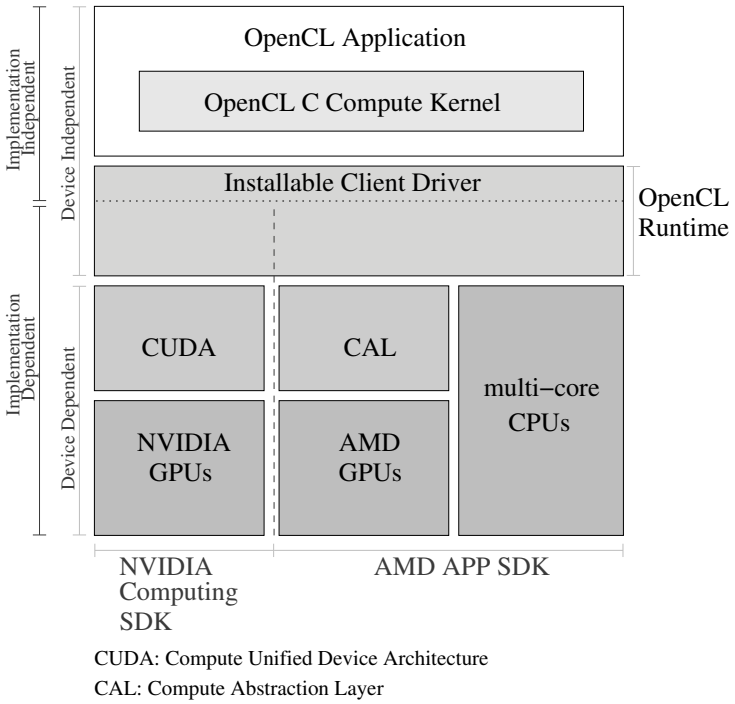

Figure 1.3: OpenCL implementations on different platforms.

OpenCL programming guide [AMD Inc, 2011a], the paper of [Gummaraju et al., 2010] and the paper of [Zhang et al., 2011] are used as the main reference for the use of terminologies.

1.2.1 The OpenCL platform model

An OpenCL platform consists of a **host** connected to one or more **compute devices**, each of which contains one or more **compute units**. Each compute unit consists of one or more **processing elements**. This hierachical model gives an abstract logical view over an heterogenous and scalable computing system:

- The compute devices can be any kind of *"processing units"*, from different vendors, such as the Intel CPUs, the AMD GPUs, the NVIDIA GPUs, the AMD APUs, etc. Any compute device can be added to or removed from the computing system.
- The number of *"cores"* in each compute device can vary in a very wide range. Based on the design of the vendor and the hardware driver, the *"core"* can be further divided into processing elements. The scheduling and executing of the computing tasks in the compute units and the processing elements are transparent for the programmer.

The mapping of the OpenCL platform onto different architectures is based on both the physical and logical features of the devices. It depends also on the implementation approach of the SDK vendors. The mappings of the OpenCL platform onto Intel Nehalem microarchitecture CPUs, AMD Evergreen generation GPUs and NVIDIA Fermi generation GPUs are described in Figure 1.4 and Table 1.2:

Compute device	Compute Unit	Processing Element
NVIDIA Fermi GPU	Streaming Multiprocessor (SM)	Streaming Processor (SP, or CUDA core)
AMD Evergreen GPU	SIMD Engine	Processing Element (PE)
Intel Nehalem CPU	Thread (Logical Core)	Thread

Table 1.2: Mapping of OpenCL platform model onto Intel Nehalem CPU, AMD Evergreen GPU and NVIDIA GPU.

- **Fermi GPUs, NVIDIA Computing SDK.** An NVIDIA Fermi GPU contains upto 16 streaming multiprocessors. Each streaming multiprocessor consists of 32 "CUDA cores".
- **Evergreen GPUs, AMD APP SDK.** An AMD Evergreen GPU contains upto 20 SIMD engines. Each engine consists of 16 stream cores, in each of which there are 5 processing elements. It means that there are 80 processing elements in an "AMD Evergreen SIMD engine".
- **Intel Nehalem CPUs, AMD APP SDK.** At the *physical* view, there are upto 8 cores in an Intel Nehalem CPU. Thanks to the Intel's Hyper-Threading Technology (HT) [Intel Corp, 2002], it reaches 16 *logical* cores. In Intel's offical technical specification documents, the HT based *logical cores* are called *threads*. In the OpenCL implementation of AMD APP SDK, the work-items are executed in turn by a thread (further presented in 1.2.2). This approach is the same with the Intel's OpenCL implementation [Intel Corp, b]. Thus, for the Intel Nehalem CPUs, the thread is also the processing element.

At the physical view, the AMD SIMD engine is in the same level with the NVIDIA streaming multiprocessor, thus, the AMD stream core is comparable to the NVIDIA CUDA core. But as described above, the AMD stream core is further divided into a deeper level, while the CUDA core is not, even there are also 2 computational elements inside each CUDA core, the FP Unit and the INT Unit. The main reason for the difference between the logical view of the two vendors is that an AMD SIMD engine is a vector processor, where all processing elements can execute the same operation simultaneously on multiple data elements, while an NVIDIA streaming multiprocessors is a scalar processor where, at a point of time, either the FP Unit or the INT Unit is used to execute an instruction[11].

Thus, the difference in types of processor results in a signicicant difference between the numbers of "cores" in the technical specifications of the comparable GPUs of AMD and NVIDIA. For example, the ATI Radeon HD 5870 has 1600 "cores" while the NVIDIA GeForce GTX 480 has 480 "cores", even these two GPUs seem to tie in the computing performance.

[11]The multiprocessors of the NVIDIA changed from vector to scalar since the GeForce 8 series [NVIDIA Corp, 2006].

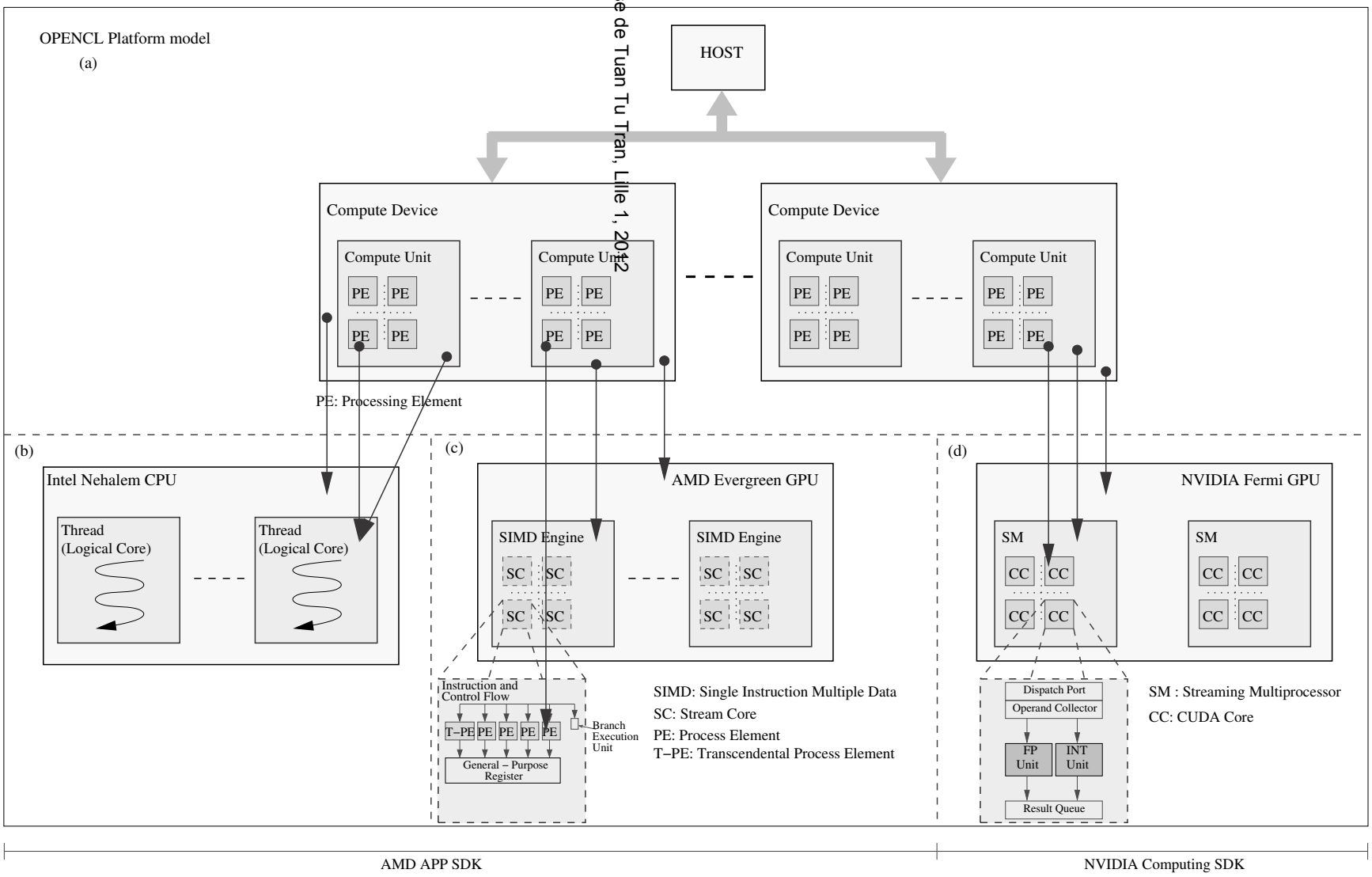

Figure 1.4: (a) The OpenCL platform model. The mapping of OpenCL platform model onto (b) Intel Nehalem microarchitecture CPU, (c) AMD Evergreen generation GPU and (d) NVIDIA Fermi generation GPU. The architectures of the AMD stream core and the NVIDIA CUDA core are cited from [AMD Inc, 2011b] and [NVIDIA Corp, c].

1.2.2 The OpenCL execution model

An OpenCL program contains two parts: a **host program** that runs on the **host** and one or more **kernels** that runs on the **compute devices**. The host program aims to set up the OpenCL context and to provide the host-device interaction mechanism. The kernel will be run as a number of instances which are executing independently on the **processing elements** of the compute device.

The execution of an OpenCL program in the host side within the OpenCL context. The OpenCL context is an *abstract container* created on the host. A context can include one or more following resources: computing devices, command queues, memory objects, event object, program objects and kernels:

- **Computing devices**: as described in the platform model, a host connects to a set of computing devices. When creating a context, it must specify the list of selected devices from this set. An OpenCL context can be only created from the devices that belong to the same platform. In the case of multiple platforms co-existing in the machine, this heterogenous computing environment can be created and monitored by multiple contexts in the host program.
- **Command queues**: the OpenCL command queue is the mechanism which provides the interaction between host and devices. One command queue is associated with only one device, but one device can be associated with one or more command queues.
- **Memory objects**: the OpenCL memory objects are the "encapsulated container" presenting the data which can be transfered between the host and the device.
- **Event objects**: the OpenCL event objects are the mechanisms which allow the profiling of the submitted commands in the command queue and represent the dependencies between these commands.
- **Program objects**: the program objects represent the OpenCL C source code that can be compiled and run on the computing devices[12]. To avoid the confusion with the whole OpenCL program which consists of both the host program and the kernels, from now on, this part of source code will be called the **kernel program**. The OpenCL kernel program usually be compiled into the device specific instructions in the run time, allowing the execution in the heterogenous enviroment.
- **Kernels**: The OpenCL kernel is a function contained in the program and is a unit of execution that can be run on a device.

OpenCL provides a runtime layer called "installable client driver" (ICD). With the ICD, one can select, at runtime, a platform from heterogeneous computing environment (which may contain different OpenCL platforms from different vendors, Figure 1.3). An OpenCL program can be first compiled with one of these implementations, and the OpenCL APIs are linked only to the ICD[13]. When this program is launched, the host program can retrieve the list of platforms

[12]The OpenCL C programming language, used to implement the kernels that run on the computing devices, is based on the ISO/IEC 9899:1999 C language specification (C99 specification).

[13]The common ICD is implemented as libOpenCL.so in both AMD APP SDK and NVIDIA Computing SDK.

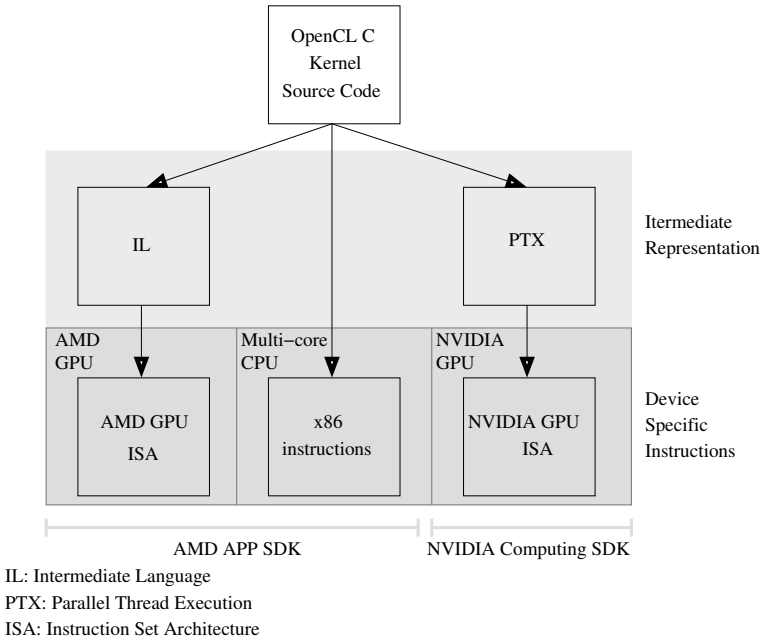

Figure 1.5: The compilation of OpenCL C Kernel on different platforms.

as well as the available devices in order to create the context. For each selected platform or device, the OpenCL program will transparently link to the dynamic library interface of the asscociated vendor's SDK (Figure 1.3)[14].

The OpenCL kernel programs is built by the compiler of the selected vendor's SDK. The kernel compilations on the AMD APP SDK and the NVIDIA Computing SDK are described in Figure 1.5. For x86 multi-core CPUs, the kernel program is compiled into the x86 instructions. For AMD GPUs and NVIDIA GPUs, there are two compilation levels. First, the kernel program is compiled into an intermediate representation: AMD's IL (Intermediate Language) or NVIDIA's PTX (Parallel Thread Execution). This intermediate representation is further just-in-time (JIT) compiled into the device specific instruction set architecture (ISA), which will be run on the GPUs.

Generally, the execution of an OpenCL program has three main steps: (1) create the context, (2) compile the kernel program, (3) monitor the devices to do the computing works by submiting the commands (host-device data transfer, kernel execution, etc) to the command queue.

The mapping of the OpenCL kernel instances into the OpenCL index space. When the host submits a kernel to execute, the OpenCL runtime defines an N-dimensional index space called **NDRange**. The number of the dimensions of an NDRange can be 1, 2 or 3. An instance of the kernel, running on the computing device, is called a **work-item**. Each work-item, corresponding to one point in NDRange, executes the same code source on its given data. An NDRange of size G provides the execution space in which there are G work-items intended to be executed concurrently.

[14]The NVIDIA GPU Computing SDK's driver is provided in `libCUDA.so`. The ADM APP SDK's driver is provided in `liboclamd32.so` and `liboclamd64.so`.

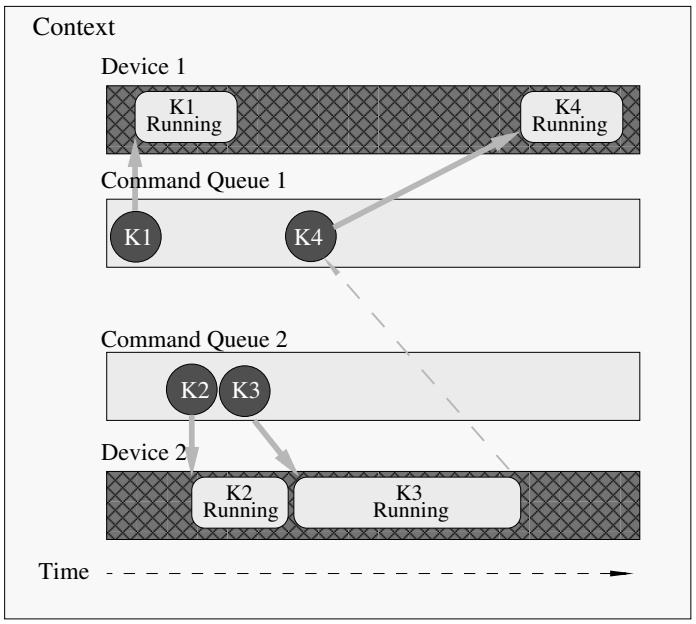

Figure 1.6: Example of the execution of an OpenCL program. An OpenCL context consists of 2 devices, associated with 2 command queues. The OpenCL program is a collection of 4 kernels: K1, K2, K3, K4. K1, K2, K3 are independent from each other, except that K4 is dependent on K3. The kernels are submited and scheduled to run on the devices through the command queues. The event object associated with the submission of K3 maintains the dependency between K4 and K3 as well as the synchronization between the executions of the commands in different command queues.

At the view point on the whole index space, each work-item can communicate with the others in the global level. To support the local cooperations level, work-items are organized into **work-groups**. The data exchanges and the synchronizations between the work-items within the same work-group are simpler and more efficient than at the global level. Globally, the work-item and the work-group are assigned a unique identifier (ID). Inside the work-group, the work-item is also given the local ID.

The execution of the OpenCL kernel instances in the compute device. In both the NVIDIA Computing SDK and the AMD APP SDK, each work-group is mapped onto one compute unit, and one compute unit can host one or more work-groups. This means that each work-item is executed by one processing element, while one processing element may execute one or more work-items. At the device level, the execution of work-items on the GPUs and the multi-core CPUs are different.

The GPU compute unit schedules and executes the work-group as multiple sub-groups of N work-items. In the AMD APP SDK, these sub-groups are called the **wavefronts** and, in the NVIDIA Computing SDK, they are called the **warps**. The size of the wavefronts or the warps depends on the device. For example, the ATI HD 5800 series GPUs of the AMD Evergreen family contain the 64 work-item wavefronts and the NVIDIA Fermi GPUs contain the 32 work-item warps. The work-items in the same wavefronts or the same warps always execute the same instruction. Thus, the total number of the instructions needed to finish a kernel program increases linearly with the number of divergent code paths. A compute unit can support multiple

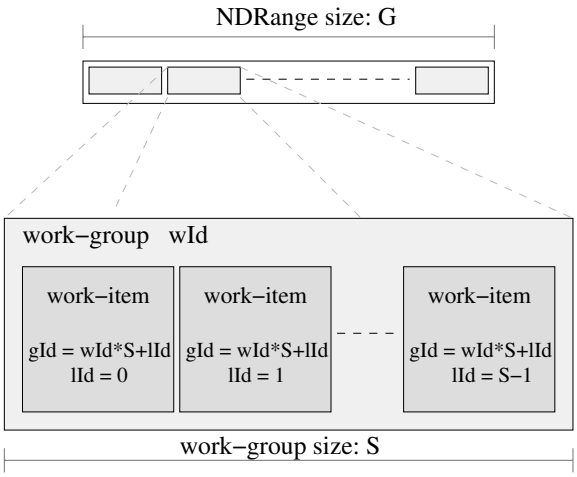

Figure 1.7: Example of an 1-dimensional NDRange which contains G work-items, each with the unique global ID (gID) as well as the local ID (lID). The work-items are organized in work-groups of size S. Each work-group has a unique group ID (wID).

OpenCL	CUDA
work-item	thread
work-group	thread-block
NDRange	grid of thread-blocks

Table 1.3: The correlation between the execution model terminologies of the OpenCL standard and of the CUDA architecture

wavefronts/wraps simultaneously[15]. From now on, we use the notation **warp** to represent both the NVIDIA warp and the AMD wavefront, except the cases that need explicit distiguishment.

For multi-core CPUs, the AMD APP SDK maps the whole work-group to one thread. Generally, the kernel program can be divided into multiple sequential sections by the synchronization barriers. Each section is executed by each work-item in turn before changing to the next section. One thread can run multiple work-groups (Figure 1.8 a)[16].

1.2.3 The OpenCL memory model

The OpenCL memory model is an abstract memory system with 4 distinct regions:

- **Global memory** is with full read/write access for all the work-items in the computing device. This region also plays the role of host-device data exchange.
- **Constant memory** is read-only accessible to all the work-items in the computing device. It is used for unchanged data that are written from the host and read simultaneously by

[15] For exeample, the Streaming Multiprocessor of the NVIDIA Fermi GPUs can support upto 48 warps simultaneously [NVIDIA Corp, b].

[16] The more detail description of the OpenCL work-group execution by the CPU thread can be found in [Gummaraju et al., 2010, section 4.1] and [Gaster et al., 2011, Chapter 6].

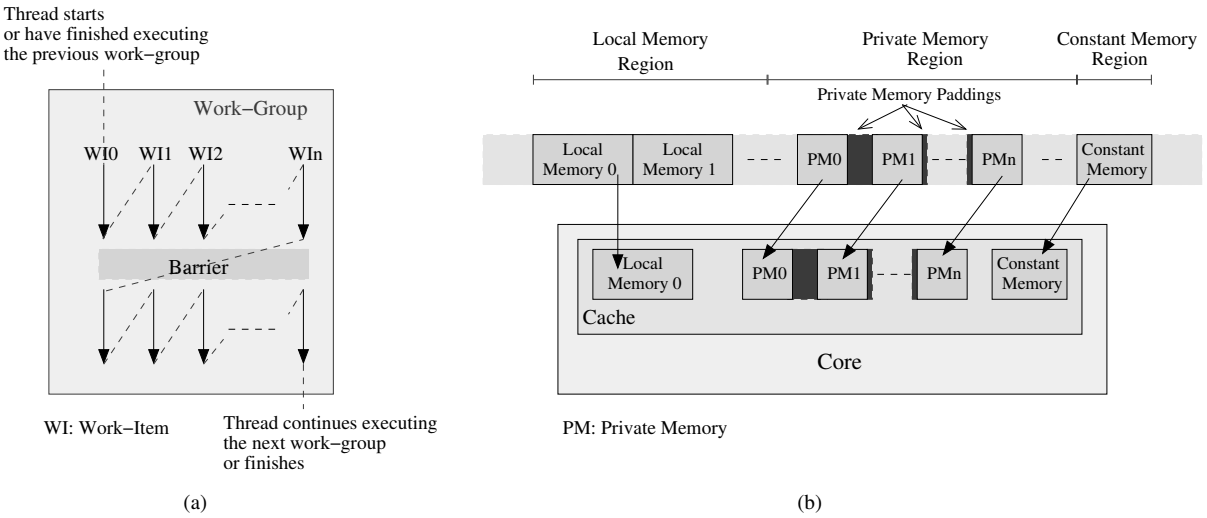

Figure 1.8: The AMD APP SDK implementation of OpenCL on multicore CPUs. (a): The execution of work-items of the work-group in the single thread. (b): The mappings of the local memory, private memories and constant memory associating to a work-group, from the main memory of the host to the cache inside the CPU core.

	Global	**Constant**	**Local**	**Private**
Host	Dynamic allocation read/write	Dynamic allocation read/write	Dynamic allocation no access	No allocation no access
Kernel	No allocation read/write	Static allocation read-only	Static allocation read/write	Static allocation read/write

Table 1.4: Memory region - allocation and memory access capabilities (*table cited from "The OpenCL Specification, version 1.1" of the Khronos Group [Khronos Group, 2010]).*

all work-items.

- **Private memory** belongs to a work-item. This type of memory can only be accessed (read/write) by a work-item that owns it, and contains either variables declared inside the kernel codes or non-pointer arguments of the kernels.
- **Local memory** is assigned for a work-group and is shared among the work-items of this group. It can be dynamically allocated from the host program as the argument of the kernel or be statically allocated as a variable declared inside the kernel program. The host can not directly read from or write to this region: Only the work-items of the considered work-group can do the transfers between the local memory and the three other types of memory.

The mapping of the OpenCL memory platform onto each type of device depends on the memory system of the device and on the implementation of the vendor. For the GPUs, the global memory and the constant memory are mapped to the off-chip video memory of the GPUs. The private memory is usually mapped to the registers, but with the AMD GPUs, it

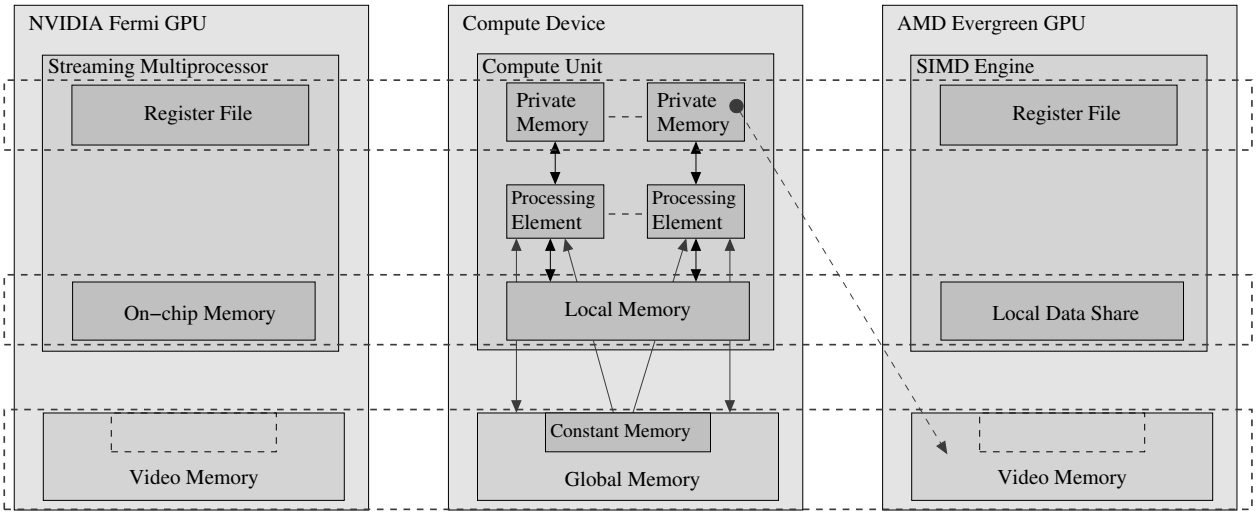

Figure 1.9: Mapping the OpenCL memory model (centre) onto NVIDIA Fermi GPU (left) and AMD Evergreen GPU (right). With AMD Evergreen GPUs, the private memory can be mapped on to either the registers or the video memory.

can be also mapped to the video memory (in the cases of private arrays and spilled registers [Gaster et al., 2011]).

The local memory on the GPUs of the two vendors is both mapped to the on-chip memory inside each compute unit[17]. For the AMD Evergreen GPUs, this type of memory is called "local data share" (LDS), inside each SIMD engine, and has the size of 32 KB. For the NVIDIA Fermi GPUs, the 64 KB on-chip memory inside each Streaming Multiprocessor is used to host both the local memory and the L1 cache.[18].

Comparing to the off-chip memory, the on-chip memory is much larger in size, but has smaller bandwidth and higher latency.

The AMD APP SDK maps all types of OpenCL memory onto 4 continous distinct regions in the main memory of the host, outside the multi-core CPUs. But there are also mappings from local memory, the private memory and the constant memory regions in the main memory to the cache in each CPU core (Figure 1.8 b)[19].

1.2.4 The OpenCL programming model

The host-device interactions through the command queues and the simultaneous execution of the kernel instances, as the work-items in the N-dimensional index space, allow the development of an OpenCL as the hybrid of 2 parallel programming models:

[17]The on-chip memory inside the AMD Evergreen GPUs and NVIDIA Fermi GPUs is the scratchpad memory [Banakar et al., 2002]. It is also called the "programable cache"

[18]The on-chip memory of the Fermi GPUs is configurable, the size of local memory and of L1 cache can be either (48 KB, 16 KB) or (16 KB, 48 KB).

[19]The more detail descrition of mapping OpenCL memory model onto the main memory and the CPU caches can be found in [Gummaraju et al., 2010, section 4.2] and [Gaster et al., 2011, Chapter 6].

OpenCL	**CUDA**
global memory	global memory
private memory	local memory
local memory	shared memory
constant memory	constant memory

Table 1.5: The correlation between the memory model terminologies of the OpenCL standard and of the CUDA architecture

- Data parallelism.
- Task parallelism.

Data parallelism. At the global view, a kernel program is executed as a Single Program Multiple Data (SPMD) application: the same code is applied to different data. As one work-group is usually scheduled to run on one computing unit, one compute device can simultaneously process multiple work-units (instances of the kernel). However, as described in 1.2.2, the work-items that belong to the same wavefronts/warps always execute the same instruction, even if there are divergent code pathes[20]. It means that the execution of the work-items in the same wavefronts/warps follows the Single Instruction Multiple Data (SIMD) model.

The organization of the work-items into work-groups and the different levels of the memory model support multiple degrees of data parallelism. There following features of OpenCL should be noticed to achieve a good parallel strategy:

- The communications and synchronizations among the work-items in the same work-group are supported by the on-chip local memory and "barrier" OpenCL functions.
- Among the work-groups, the communications can only done by the out-chip global memory.
- OpenCL does not support global synchronizations among work-groups.

So, it would better to implement the fine-grain parallelism on the work-items inside the same work-group, and the coarse-grain parallelism are on "inter-work-group" level.

Task parallelism. As an OpenCL kernel can be considered as one task, an OpenCL program is the form of a processing pipeline which consists of multiple tasks. The parallelism of the tasks and the multiple instances executing of the processing pipeline are shown in Section-1.1.1. The dependencies and the synchronizations between the tasks are defined and maintained by the event objects. There are two levels for processing the tasks simultaneously in the running time enviroment:

- Inter-device task parallelism: the tasks can be submitted to different compute devices to be processed parallely. For example, in Figure 1.6, two kernels $K1$ and $K2$ run parallely on Device 1 and Device 2.

[20] More detail about the branch divergence can be found in 1.2.5.1.

- Intra-device task parallelism: In modern GPUs, multiple kernels can run concurrently in one compute device[21]. Depending on the global sizes of the kernels and the number of the processing elements, at one point in time, one or multiple kernels can be executed by the computing device.

1.2.5 Optimizing code for manycore OpenCL programming

This section discusses two factors that can significally influence the performance of an OpenCL application when running on GPUs: the global memory access and the branch divergence[22]. These two factors are analyzed and in particular cases which are further presented in Chapter-2, Chapter-3 and Chapter-4.

1.2.5.1 Branch divergence

As the running instance of the kernel code, a work-item consisits of multiple *instructions* that are dispatched to run on a processing element. In the GPU computing unit, the work-items are not processed independently but grouped into *warps*[23]. Moreover, the processing elements are also organized into the *lanes*. The scheduler at the computing unit level decides which instruction of which warp is executed by which lanes, thus the work-items belongs to the same warp always execute the same instruction simultaneously (as known as SIMD, Single Instruction Multiple Data).

In the run time, the data differences between the work-items can lead to different execution paths. This problem is called the **branch divergence**, and has a large impact on the performance of GPU applications, as *all possible paths are dispatched sequentially* to be executed. Having the SIMD behavior, the warp serially passes through all the available paths of the kernel code while disabling work-items that are not in each one. These work-items converge back when all the paths are completed [Lindholm et al., 2008] (Figure 1.10).

Moreover, there is a waste in the use of processing elements: the total number of dispatched instructions is higher than the number needed to process them serially [AMD Inc, 2011a, Chapter 1].

1.2.5.2 Random global memory access

GPUs have a large off-chip memory, on which located the global memory, with high bandwidth but high latency (1.2.3)[24]. This type of memory is advantageous when the frequence of access transactions is small but the data amount of each transaction is large and in continuous region. As long as the memory access pattern is optimized, it can effectively handled hundreds or thousands simultaneous data read or write transactions [Jang et al., 2011]. But, in the case of high frenquency light weight memory accesses to the random region in the global memory,

[21] For the NVIDIA GPUs, begin from the Fermi family, a GPU can execute the kernels in the same context concurrently [NVIDIA Corp, c, page 18].

[22] The complete guides about optimization of OpenCL implementations on GPUs can be found in [NVIDIA Corp, 2009b] and [AMD Inc, 2011a].

[23] As mentioned in page 26, the notation *warp* is used to represented both the NVIDIA warp and the AMD wavefront.

[24] The bandwidth and the latency of the GPU off-chip memory is compared with the main memory of the computer.

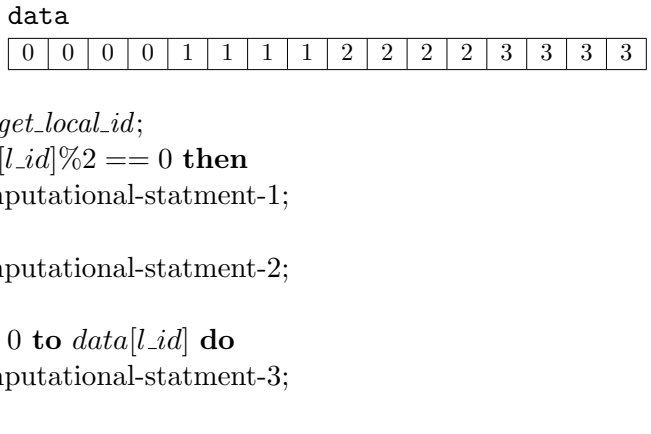

Figure 1.10: Example of branch divergence on a work-group of 16 work-items. *get_local_id* is the command to get the local identifier of the work-item in the work-group. All the work-items have to pass througth both paths in the conditional statement `if-then-else` and 4 iterations of the `for-loop`, even in each pass not all of them have to execute the corresponding computational statement.

it can cause the botleneck in data transfer due to the high latency and seriously decrease the performance of the application on the GPU. However, for the GPUs of NVIDIA Fermi or AMD Evergreen class, there is a cache for data transfer from the global memory.

As recommended in the OpenCL programming documents such as [NVIDIA Corp, 2009b, Chapter 3] or [AMD Inc, 2011a, Chapter 4], the global memory should be accessed in the *coalesced pattern* where the the work-items in the same wavefront/warp read from or write to the contiguous data elements. Moreover, the local memory, which is located in the very fast on-chip memory (1.2.3), should be used to temporarily store the computing data in order to reduce the number of global memory accesses.

Depending on the application and the size of the data, it is not always possible to apply these memory access optimizations. In this thesis, one of the key features of data structure design is to reduce as much as possible the *number of random global memory accesses*, which is the main idea of the neighborhood indexing approach, presented in Chapter-2. The use of local memory with coalesed data tranfer is also discussed in 3.1.4.

1.3 GPU in Bioinformatics

The main purpose of this section is to list the bioinformatics applications that have been mapped onto GPUs. It can be considered as an extension from [Varré et al., 2011] with the addition of the bioinformatics GPGPU publications since 2009.

A note on speedups. In this section, we report many different studies from various fields of bioinformatics, with different methodologies for parallelizing, experimenting and evaluating the speedups. Moreover, the GPUs to do the experiments are also in the wide range and span several architecture generations. Generally, it is very difficult to fairly measure these speedups – for example, with GPU/CPU comparison, what is the base reference (CPU?, multi-core CPU? multi-core CPU with SIMD instructions? grid of CPUs?), and was this CPU code

really optimized?

In addition, the raw times, in seconds, could be interpreted differently depending on the objective of application – for example, in our Chapter 5, we will use a normalized measure for solving approximate pattern matching (see page 84). Finally, it lacks the independent benchmarking suite for these solutions. *The speedups reported here should thus not be taken absolutely and should not be used to compare these different studies*, but rather as a raw indication given by the different authors.

Contents of this section. Many bioinformatics high-performance studies concern *sequence similarities*: this domain was previously dominated by pairwise alignment (Section 1.3.1), either by exact dynamic programming or with heuristics. With the advances in High-Throughput Sequencers (HTS), sequence similarities are now used in fruitful new domain of research (1.3.2). We also list other GPU studies in sequence algorithms (1.3.4), proteomics (1.3.5), data mining (1.3.6) and cell simulation (1.3.7).

⋆ ⋆ ⋆

Note that the motivation for studying similarities between genomic sequences will be presented in the next chapter (page 43), focusing on the core of this thesis – the filtering phase of seed-based heuristics.

The following pages do not aim to explain all cited problems, it is more a preview of what has already done with GPUs in bioinformatics – and what speedups the authors usually report.

1.3.1 Pairwise alignment

The tools presented here find similarities between two genomic sequences, either by exact dynamic programming or, like in the rest of this thesis, within a seed-based heuristics.

1.3.1.1 Pairwise alignment by exact dynamic programming

Smith-Waterman. The main algorithm for computing local similarities between genomic sequences is Smith-Waterman [Smith and Waterman, 1981] which runs in quadratic time over the length of the sequences. This algorithm is very regular and was often parallelized in the three last decades on various platforms (dedicated hardware such as FPGA, CPU with SIMD, grids, etc) [Lavenier and Giraud, 2005]. One of the most efficient parallelizations of this algorithm should be the SIMD implementation of [Farrar, 2007], in which the author proposed an *"striped query profile"* technique to reduce the number of iteration loops over the dependent data. When running as a single thread on a 2.0 GHz Xeon Core 2 Duo, it achieved the speed about 3 billion cell updates per second.

The Smith-Waterman alorithm was thus an interesting target of choice for GPU parallelization. In 2006, [Liu et al., 2006b] proposed the GPU implementation of the algorithm by using OpenGL. The anti-diagonals of the dynamic programming matrix are processed simutaneously.

The implementation on a GeForce 7800 GTX has a reported speedup of almost 16× than OSEARCH and 8× than SSEARCH[25]. The first Smith-Waterman CUDA implementation should be the work of [Manavski and Valle, 2008] which was published in 2008. Their implementation on a single GeForce 8800 GTX was reported to be 3.5× to 4.7× faster than the experiments of [Liu et al., 2006b]. In the same year, but some months later, [Munekawa et al., 2008] published another CUDA Smith-Waterman implementation which also uses the anti-diagonal based parallelization strategy but with improved memory assigment and data reuse schemes. On a single GeForce 8800 GTX, the implementation of [Munekawa et al., 2008] was reported to be 3.1× faster than the one in [Manavski and Valle, 2008] and 6.4× faster than the one in [Liu et al., 2006b].

In 2009, [Striemer and Akoglu, 2009] also showed that [Manavski and Valle, 2008] was still highly CPU dependent. They proposed GSW, the absolute GPU dependent Smith-Waterman implementation. In comparison with the serial version of SSEARCH, GSW has a reported peak speed-up of 23× on a Tesla C870[26]. At the same year, [Liu et al., 2009a] released CUDASW++ which has a lot of memory access optimizations to gain the sequencing speed. Moreover, there are two parallelization strategies in CUDASW++: the intra-task and the inter-task, which are used for the long and short queries respectively[27]. In comparision with the work of Manavski and Valle, on the same dual-GPU GeForce GTX 295, CUDASW++ had the speedup of up to 10×. In 2010, the same authors, [Liu et al., 2010b], released the upgraded version (CUDASW++2.0), with the optimized SIMT (Single Instruction Multi Thread) algorithm and the virtualized SIMD (Single Instruction Multi Data) vector programming model. On an NVIDIA GeForce GTX 295, CUDASW++2.0 was faster than CUDASW++ from 1.45 × to 1.72 ×. Independently, [Hains et al., 2011] showed that the intra-task kernel of CUDASW++, which deals with the long queries, have a great impact on the overall performance of the genome database aligning and thus, they proposed their upgraded kernel. On an NVIDA C2050, the improved version of [Hains et al., 2011] increased the performance of at most 39.3. There were also other CUDA Smith-Waterman implementations in 2009 presented by [Ligowski and Rudnicki, 2009] and by [Ling et al., 2009]. The peak performance of the implementation of [Ligowski and Rudnicki, 2009] on a dual NVIDIA 9800 GX2 was 4.1× higher than that of [Manavski and Valle, 2008] on a dual NVIDIA GeForce 8800 GTX. [Ling et al., 2009] did not focus on the running speed but on the length of the queries. In most cases, their implementation was slower than those of [Manavski and Valle, 2008] and of [Munekawa et al., 2008] but it could align the sequence of any length[28].

In 2010, [Dohi et al., 2010] (include all 3 authors of [Ling et al., 2009]) published the CUDA Smith-Waterman implementation with deeply optimizations on both algorithmics side and hardware side. In the peak performance, this implementation was about 1.45× faster than CUDASW++ [29]. In this year, there was the first report of an OpenCL Smith-Waterman implementation [Razmyslovich et al., 2010]. On an NVIDIA GeForce GTX 260, the authors announced

[25] OSEARCH and SSEARCH are two Smith-Waterman implementations in the FASTA program [Lipman and Pearson, 1988].

[26] Striemer and Akoglu did not do the direct comparation between their works with the one in [Manavski and Valle, 2008]. They explained that the mapping in [Manavski and Valle, 2008] was the combination of CPU and GPU, and it could not align more than 400 residues on the GPU.

[27] In CUDASW++, the defaut threshold to differentiate long and short queries is 3072 residues.

[28] Both [Manavski and Valle, 2008] and [Munekawa et al., 2008] have the limit on the query length. On the testing platform of [Ling et al., 2009], the limits were 2500 and 2048 respectively.

[29] It means that the performance of the implementation of [Dohi et al., 2010] is approximately equal to that of CUDASW++2.0

the speedup of 3× over CUDASW++2.0.

In 2011, there were others publications with CUDA implementations : [Hasan et al., 2011b], [Hasan et al., 2011a] and of [Zheng et al., 2011]. The implementation of Hasan et al, in comparation with CUDASW++2.0 on a NVIDIA GeForce GTX 275, in the case of fully optimization was reported to be about 1.13× faster but was 1.52× slower in the less optimized implementation.

Up to mid of 2012, the Smith-Waterman should thus be the most favorite bioinformatics algorithm to be mapped on the GPU.

Needleman-Wunsch. The Needleman-Wunsch algorithm was actually published before the Smith-Waterman algorithm [Needleman and Wunsch, 1970]: it consists in finding the global alignment between two genetic sequences. This alogorithm also has quadric time complexity over the length of the sequences.

In 2008, [Che et al., 2008] introduced the CUDA implementation of the Needleman-Wunsch algorithm [Needleman and Wunsch, 1970] as a case study in an article about the performance exminations of some general-purpose applications. With the anti-diagonal parallelization scheme, on an NVIDIA GeForce GTX 260, this implementation was reported to be up to 2.9× faster than the naïve serial CPU version. In 2010, [Siriwardena and Ranasinghe, 2010] studied different levels of memory access methods on the CUDA compatible GPUs in order to find the efficient ways to implement the Needleman-Wunsch algorithm. On an NVIDIA GeForce 8800 GT, their implementation had a reported speedup of up to 4.2× over the CPU version. Recently, in 2012, [Farivar et al., 2012] carefully examinated the CUDA implementation of the Needleman-Wunsch algorithm. They proposed the solutions for two serious problems of high memory consumption and of diverging SIMD flows when mapping the algorithm to the GPUs. On a NVIDIA GeForce GTX 275, this implementation was reported to be up to 8× faster than a multi-thread implementation which run on a quad-core Intel Core i7-920 CPU.

Traceback. The traceback, also called backtrack, is the process to get the full alignment from a dynamic programming matrix. Many high-performance tools do not implement this phase, or fall back to the CPU, arguing that, in many applications, there are very few such alignments at the output of the algorithm.

In 2011, [Blazewicz et al., 2011] published a CUDA implementation of both Smith-Waterman and Needleman-Wunsch with an efficient traceback routine. They did the comparision with the corresponding implementations in the EMBOSS package [Rice et al., 2000][30]. On a NVIDIA GeForce GTX 280, their implementations had a reported speedup of at most 68× with the Needleman-Wunsch and 108× with the Smith-Waterman. Their experiments on multiple GPUs also achieved a linear speedup.

1.3.1.2 Pairwise alignment with seed-based heuristics

The tools in this section use *"seed-based heuristics"* to execute the pairwise aligment between large sequences, with the trade-off between the speed and the sensibility. More details about this technique can be found in the next chapter (Section 2.1.2).

[30] In the EMBOSS package, the implementation of Needleman-Wunsch algorithm named *needle*, the implementation of Smith-Waterman algorithm named *water*.

BLAST. As one of the most popular bioinformatics applications, BLAST [Altschul et al., 1990] is an interesting target to be mapped onto the GPUs. There are some variants in the BLAST family, but generally, they all have four steps: (1) Hit detection, (2) Ungapped alignment, (3) Gapped alignment and (4) Gapped alignment for traceback. In fact, the first and second steps are usually intergrated into one kernel.

- [Ling and Benkrid, 2010] developed a CUDA implementation of gapped BLAST with 2 kernels. The first kernel does the first and second steps of BLAST. The host evaluates the output ungapped alignment and sends to the second kernel, which does the third step. On an NVIDIA GeForce 8800 GTX, it was reported to be from 1.7× to 2.7× faster than NCBI-BLAST.

- [Vouzis and Sahinidis, 2011] released a CUDA implementation named GPU-BLAST that is based directly on the source code of NCBI-BLAST[31]. GPU-BLAST only maps the hit detection and gapped extension to the GPU in one kernel. The speedup of GPU-BLAST over NCBI-BLAST on a NVIDIA Fermi C2050 was reported to be between 3× and 4×.

- In 2010, Liu published a Tesla-GPU version of BLASTP (named CUDA-BLASTP) on the website of NVIDIA [Liu, 2010]. The speed up of this CUDA-BLASTP version on a dual NVIDIA C1060 computer was reported to be at most 10× over NCBI-BLAST 2.2.22. As discussed lately in [Xiao et al., 2011], this version parallelized only the first two steps of BLAST. This work was further improved in [Liu et al., 2011c] with the paralleliza-tion of the third step of BLAST in a second kernel. On a NVIDIA GeForce GTX 295, this implementation had a reported speedup of at most 10.0× over NCBI BLAST 2.2.22. Using MPI (Message Passing Interface), OpenMP (Open Multi-Processing) and CUDA, [Liu et al., 2011b] developed the more parallelized version named mpiCUDA-BLASTP that can run on the clusters of GPU nodes. This implementation is a colaboration of different parallelization levels: on the cluster nodes (message passing), on the multicore CPUs (multi-threaded) and on the GPUs. On a cluster which consisted of 2 Tesla S1060 quad-GPU computing systems, their implementation was reported to be 1.6× and 6.6× faster than GPU BLAST 1.0-2.2.24 of [Vouzis and Sahinidis, 2011] when run with a single GPU and with 6 GPUs respectively.

- [Xiao et al., 2011] deeply experimented the mappings of BLASTP on the Tesla and Fermi NVIDIA GPUs. Their authors proposed and implemented 5 GPU BLASTP versions with diffirent optimization techniques and carefully evaluated the speedup in the parallelization of the each in the first 3 steps of BLAST. The highest reported speedup between their GPU implementation and their CPU implementations was 6×. In comparision with the Tesla version of CUDA-BLASTP [Liu, 2010] on a NVIDIA C1060, their implementation was reported to be from 1.8× to 2.0× faster[32].

- [Lin et al., 2011] used a CUDA GPU kernel for the "matching word" step of Mercury BLAST [Jacob et al., 2008]. This step bases on the Bloom filter [Bloom, 1970] and is originally accelerated by the FPGAs. In comparision with the CPU version which ran on an AMD Opteron quad-core, the reported speedup was approximately 35×.

[31] Thus, the implementation of [Vouzis and Sahinidis, 2011] maintains the indentical results as of NCBI-BLAST. As mentioned in the article, this version of GPU-BLAST is only for BLASTP

[32] [Xiao et al., 2011] did the comparision with the very first version of CUDA-BLASTP, which parallelizated only step-1 and step-2 of BLAST.

PLAST. Based on the idea of BLAST, [Nguyen, 2008] developed PLAST (Parallel Local Alignment Search Tool), specialized for the alignment between genetic sequence banks. The GPU version of PLAST parallelized the ungapped extension and the small gapped extension over the flanking region around each detected hit. On a platform with dual NVIDIA C870 GPU, it achieved the speedup from 5.38× to 10.13× over the NCBI TBLASTTN, which ran as 2 threads on a 2.6 GHz Xeon Core 2.

MUMmer. While the BLAST family used the fixed-length seed in the "hit detection" step, MUMmer [Delcher et al., 1999] [33] can detect the hit of maximal length, or the *maximal unique matching* (the MUM), by using the suffix tree [Weiner, 1973]. Once the MUMs between two sequences are found, MUMmer uses the Smith-Waterman algorithm to do the full extend (*seed-and-extend*). [Schatz et al., 2007] published MUMmerGPU 1.0, which mapped the MUM detection step on to the GPUs. The suffix tree is flattened to keep in the 2D texture memory of the GPUs. On an NVIDIA GeForce 8800 GTX, this implementation was reported to be from 3.4× to 3.8× faster than the serial MUMmer. This implementation was further improved [Trapnell and Schatz, 2009] with a reported speedup of 13× over the serial version.

1.3.2 Algorithms for High-Throughput Sequencers (HTS)

High-Throughput sequencers will be presented in Section 6.1. Such sequencers have drawn a lot of interest in recent years, notably with the tools for read mapping and read assembly. The applications presented here takes as input a large set of millons of reads (very short sequences, about tens to hundreds residuces) and thus requires a large amount of computing power to be solved.

Read mapping. In all resequencing projects, the main problem is to locate reads on a reference genome (such as the human genome). Read mapping will be extensively discussed in Chapter 6, with a summary of all available CPU and GPU read mappers (see page 99).

Read assembly. When there is no base genome reference, the main problem is to reconstruct a genome (or several genomes in the case of metagenomics) from the reads (*de-novo* assemly). As described in [Shi et al., 2010], the existing assemblers can be classified as the overlap graph based and the de-Bruijn graph based. The de-Bruijn based assemblers consists of two main stages: finding the exact overlaps of a given length from the input reads, and then, constructing and finding the contigs in the graphs by using the found overlaps.

The study [Mahmood and Rangwala, 2011] implemented a CUDA version of the second stage of the other de-Bruijn based assembler: Euler-SR [Pevzner et al., 2001]. The authors mapped onto GPU all 3 steps in the second stage: the graph construction, the Euler tours finding and the contigs indentifying steps. On an NVIDIA Quadro FX 5800, the new version, called GPU-Euler, is reported to have a peak speedup of about 5× over Euler-SR.

Read correction. Sequencing errors in reads has may have a great influence on the result of assemblers. An additional step of *read correction* is thus often implemented before the asssembler. Accelerating this step on the GPUs is the cumulating work of the research

[33] MUMmer have been upgraded into: MUMmer 2.0 [Delcher et al., 2002], MUMmer 3.0 [Kurtz et al., 2004]).

group at the Nanyang Technological University, from [Shi et al., 2009] to [Shi et al., 2010] and to [Liu et al., 2011e]. Using CUDA, the authors implemented the spectral alignment error correction which used Bloom filter for membership testing. The GPU based error correction stage then can be used in any existing de-Bruijn graph based assemblers. The final result, named DecGPU [Liu et al., 2011e], is the GPU based fine grain parallel error correction, that are used as the core of the bigger coarse grain parallel framework using OpenMP and MPI which can run on the cluster of GPU nodes. Comparing with hSHREC, the other error correction algorithm, the speedup of DECGPU was reported to be about 22×. Moreover, DECGPU showed high quality correction results when integrating with two assemblers: Velvet and ABySS.

1.3.3 Motif/model discovery and matching

Algorithms in this category also look for similarities in sequences, but modelize these similarities through different models, which can range from a simple regular expression to more elaborated matrices or probabilistic models.

Motif discovery Motif discovery problem can be defined as *"given a set of unaligned genetic of protein sequences, indentify and characterize shared motifs which are the families of subsequences having some biological property of interest"* [Bailey and Elkan, 1993]. One of the most popular softwares to solve this problem is MEME ([Bailey and Elkan, 1993], [Bailey and Elkan, 1994]), which bases on iterative approache exploiting probabilistic matching models.

[Liu et al., 2010a] proposed CUDA-MEME with the highest speedup reported on an NVIDIA GeForce GTX 280 was 20.5× over the serial MEME 3.5.4. This work was then improved into mCUDA-MEME [Liu et al., 2011d] as a combination implementation of MPI, OpenMP and CUDA. On a cluster of 2 Tesla S2050 computing systems, which consists of 8 GPUs, mCUDA-MEME was reported to run at most 8.3× faster than a parallel MEME 4.4.0 on a cluster of 32 CPUs.

DNA transcription factors binding side locating. The transcription process from the DNA to the RNA can be studied by discovering the possible binding sites for the TF proteins[34] along the DNA. Position Weight Matrices (PWMs) are used as a step in a pattern matching with a score threshold process to locate these putative TF binding sites. A CUDA parallelization of PWM algorithms of [Giraud and Varré, 2010], on an NVIDIA GeForce GTX 280 was reported to have speedup of more than 10× than a serial implementation.

HMMER. Modeling a set of sequences in a same family can be done with probabilistic approaches, in which Hidden Markov Models (HMM) are a very popular method. The HMMs, built from the given set of sequences can be further used to evaluate the similarity between any member in the set and the query sequence.

In the HMMER package[35], the most intensive calculation stage in the search is the observation sequence generations using the P7Veterbi dynamic programming algorithm. The HMMER implementations on GPUs, reported by [Horn et al., 2005], [Walters et al., 2009] and

[34] TF, which stands for Transcription Factors, is a types of proteins which can bind themself to the DNA to help enabling the transcription of genes.

[35] http://hmmer.janelia.org

[Ganesan et al., 2010], focus on accelerating this stage. The study [Horn et al., 2005] should be the first GPU version of HMMER, written in BrookGPU and run on both NVIDIA and ATI cards with the peak speedup over the serial version, obtained with the ATI R520, was reported to be about 35×. Both using CUDA, [Walters et al., 2009] and [Ganesan et al., 2010] reported a speedup of 38× and more than 100× on an NVIDIA 8800 GTX Ultra and on a Tesla C1060, respectively.

1.3.4 Other sequence-based algorithms

Multiple sequence alignment. Multiple sequence alignment (MSA) can be considered as an extension of pairwise sequence alignment: the goal is to align a set of query sequences between themselves. One very popular MSA method is Clustal-W [Thompson et al., 1994, Larkin et al., 2007], which has 3 main steps. It uses the Smith-Waterman algorithm to compute the distance matrix between all input sequence pairs (step 1) in order to build the tree (step 2) and finally to guide the progressive alignment (step 3).

The computation of the distance matrix, which is the most time consuming step in Clustal-W, is the interest target to map onto GPU as in [Liu et al., 2006a] and in [Ling et al., 2011]. Using the same technique than the GPU implementation of the Smith-Waterman algorithm in [Liu et al., 2006b], the study [Liu et al., 2006a] developed GPU-ClustalW by using OpenGL. On an NVIDIA GeForce 7800 GTX, the first step of GPU-ClustalW was reported to be 10× faster than that of the serial version of ClustalW, bringing an overall speedup of 6×. Using the same approach, and basing on the intra-task parallelization strategy of [Liu et al., 2006a] but with improvements, [Ling et al., 2011] proposed a CUDA implementation which, on an NVIDIA GeForce 8800 GTX, was reported to be 20× faster than the serial ClustalW.

[Liu et al., 2009b] proposed MSA-CUDA, an upgraded version of GPU-ClustalW, which maps all 3 steps of ClustalW onto the GPU. The technique to parallelize the second step is the same as in [Liu et al., 2009c], the other work of the same authors. The third step parallelization technique is very similar to the first step. On an NVIDIA GeForce GTX 280, for long, average and short sequences MSA-CUDA is reported to be 36×, 18× and 11× faster than a serial ClustalW, respectively.

Phylogeny. Phylogeny tools aim at laying out a set of sequences into an evolutionary tree. They often begin with a pariwise comparison between the sequences. [Charalambous et al., 2005] proposed the BrookGPU version of RAxML [Stamatakis et al., 2005, Stamatakis, 2005]. On an NVIDIA 5700 LE, the GPU version was reported to be 1.2× faster than the CPU version. Although this speedup was not so high, the work of [Charalambous et al., 2005] is usually considered one of the first GPU bioinformatics applications.

[Suchard and Rambaut, 2009] developed other GPU algorithms to calculate the phylogenetic likelihoods of molecular sequence data. On a combination of three NVIDIA GeForce GTX 280 GPUs, their speedup was reported to be up to 90× over the optimized CPU-based computation.

Finally, [Ying et al., 2011] proposed the OpenCL implementation of DNADIST, a DNA distance matrix computation used in the phylogenetic tree reconstruction of the PHYLIP phylogeny inference package [Felsenstein, 2010]. On a dual AMD HD5850 and on an NVIDIA C2050, this implementation has the speedup reported of 24 × and 16× over the serial implementation, respectively.

Bit-Parallel Pattern Matching. Bit-parallel algorithms are efficient for some problems of approximate pattern matching (see Section 3.1.2). [Li et al., 2011] developed GPU-Agrep, a CUDA version of the approximate pattern matching program Agrep of Wu and Manber [Wu and Manber, 1992b], [Wu and Manber, 1992a]. On a NVIDIA GeForce GTX 285 and to search in the whole human genome, their GPU implementation was reported to be 70× faster than their multi-thread implementation that run on a quad-core CPU[36].

Suffix Tree and Suffix Array. Suffix trees and arrays are data structures enabling fast exact matchings; they are very efficient on CPUs. [Encarnaijao et al., 2011] maps both the suffix tree and the suffix array to the GPUs. Their work is limited only to the exact matching problem. On an NVIDIA GeForce GTX 580, their implementation was reported to have a speed-up of at most 85× over the CPU version.

RNA folding. Starting from sequence, RNA folding aims to find the secondary structure of RNA (which can be further used to study the teritary structure). of the RNAs. Folding algorithms include the basic Nussinov algorithm [Nussinov et al., 1978], based on the maximinization of base pair, and extensions based on the energy minimization, such as the complete "Turner folding model" [Matthews et al., 1999].

- [Rizk and Lavenier, 2009] implemented the GPU version of the mfold/unafold packages [Zuker, 2003], which does the RNA folding with the Turner model. On an NVIDIA GTX 280, this implementation was reported to be 17× faster than the serial verison.
- [Chang et al., 2010] mapped the Nussinov algorithms onto GPU and with an NVIDIA Tesla C2050, the authors reported a peak speedup of 290× over the serial version.

Generic dynamic programming. Finally, [Steffen and Giegerich, 2005] developed the Algebraic Dynamic Programming (ADP) as the framework to encode and generate automatically the C source code for different dynamic programming problems. This framework is able to encode several problems previously presented, such as sequence comparison or RNA folding. The CUDA backend of ADP was released in [Steffen et al., 2009].

1.3.5 Proteomics

Algorithms in this category deals with the 3D structure of proteins.

Protein structure alignment.

- [Stivala et al., 2010] developed the GPU implementation of protein structure and substructure searching which used simulated annealing. On an NVIDIA GTX 285, the speedup over the serial version was reported up to 34 × and it was proved to be one of the most accurate and fastest methods while compare with the other existing programs.

[36] GPU-Agrep can process the input patterns whose lengths are up to 64 with the Levenshtein edit distance is up to 9.

- Using Brook+, [Hung et al., 2011] proposed GPU-Q-J to accelerate the root mean square deviation calculation between the coordinates of two structure, which is very common used in structural biology. On an ATI 4770, GPU-Q-J was reported to be hundreds of times faster than the existing CPU-based methods.

- ppsAlign of [Pang et al., 2012] is a framework of parallel protein structutre alignment. The authors implemented 5 different CUDA kernels to map the 3 most time-consumed steps in the framework on to the GPU. On an NVIDIA Tesla C2050, ppsAlign was reported to be faster than other protein structure alignment methods which run on the CPU: 35.9× over TM-align [Zhang and Skolnick, 2005], 64.7× over Fr-TM-algin [Pandit and Skolnick, 2008] and 40.3× over MAMMOTH [Ortiz et al., 2002].

Protein docking. Protein docking is the computational task to study the prefer positions when the other molecular binds with the protein to form a stable complex. It usually requires very intensive calculations, as the 3D structures of the two molecules must be examined.

- [Friedrichs et al., 2009] mapped all the basic steps of the general protein docking pipeline onto GPU, with BrookGPU for the ATI devices and CUDA for the NVIDIA devices. In some cases, their implementation was reported to be up to 700× faster than the corresponding serial implementation.

- One of the main step in protein docking is to calculate the correlation between the surface of the protein with the other molecular by mearsuring the solvation energies. [Dynerman et al., 2009] implementated a CUDA code name CUSA et CUDE to compute the solvent accessible surface and the corresponding desolvent. On an GTX 280, these CUDA versions were reported to be up to 2× faster than the serial versions.

- [Ritchie and Venkatraman, 2010] proposed Hex as the first exhaustive Fast Fourier Transform based protein docking application thanks to the use of GPU. On an NVIDIA GTX 285, Hex was reported to be 45× faster than the corresponding serial version and is 2× faster than the reference docking tool ZDOCK 3.0.1.

1.3.6 Genetics or biological data mining

Genome-Wide Association Studies. According to the definition in the website of the United State's National Human Genome Research Institute[37], *"a genome-wide association study (GWAS) is an approach that involves rapidly scanning markers across the complete sets of DNA, or genomes, of many people to find genetic variations associated with a particular disease."*. One of the popular methods to implement the GWAS is the epistatis analysis which studies the interactive effects between genetic variants. [Sinnott-Armstrong et al., 2009] focused on the gene-gene interaction identifying problem with the multifactor dimensionality reduction (MDR) method of [Ritchie et al., 2001]. MDRGPU is implemented with pyCUDA and on three NVIDIA GeForce GTX 280 GPUs, it ran 140× faster than the serial version. Using the same method but with the improvements in mapping the MDR algorithm onto the GPU, [Kwon et al., 2010] proposed cuGWAM which can reach 2.9× speedup over MDRGPU. [Jiang et al., 2009] proposed epiCUDA, the framework to detect the epistasis with the other method than the MDR for

[37] http://www.genome.gov/20019523

GWAS. On an NVIDIA GTX 280, epiCUDA was reported to have a speedup of at most 25.7× over the CPU version.

Protein clustering. The Markov clustering algorithm (MCL) [Van Dongen, 2008], a tool for finding clusters in a network, can be applied to the protein interaction networks to determine protein family, as in the work of [Enright et al., 2002]. Bustamam et al proposed CUDA-MCL [Bustamam et al., 2010a], [Bustamam et al., 2010b] which on an NVIDIA GeForce GTX 285 has a reported speedup upto 9× over the corresponding serial implementation.

1.3.7 Cell simulation

- [Roberts et al., 2009] proposed a new cellular automata (CA) method to perform the whole cell reaction-diffusion simulation in long time-scale and under *in-vivo*[38] conditions. Although the first performance was not as good as the theoretical performance, their implementation was analysed for further improvements. Moreover, by comparing between an NVIDIA FX 5600 and an NVIDIA GeForce GTX 280, it was reported that their implementation could achieve a speedup which was proportional with the speed of GPUs: as a GeForce GTX 280 is 1.67× (602/325 MHz) faster than a FX 5600 in clock speed, the performances were from 1.8× to 2.4× higher.

- [Chalkidis et al., 2011] mapped the biological network modeling with the hybrid functional Petri Nets on to the GPUs with CUDA. On an NVIDIA GeForce GTX 285, their implementation was reported to be 18× faster than the serial implementation and when applying to simulate the cell boundary formation of a model containing 1600 cells, the speedup was reported to be approximately 7×.

- [Falk et al., 2011] simulated the signal transduction processes within a cell. On an NVIDIA GeForce 465 GTX, one step of this simulation was reported to be 12.4× faster than the reference OpenMP version which run on an AMD Athlon 64X2 Dual Core.

1.4 Conclusion

This chapter starts with a brief overview of the hardware models and the programming model of modern multicore and manycore processors. As the main presentatives of current manycore processors, the Graphical Processing Units (GPU) are selected to be the focus of this thesis. The massively computating performance of GPUs and the technical developements that have made GPU programming more and more popular are then studied and presented. OpenCL, the programming language used throughout this thesis, is also introduced.

We saw that many applications of GPU in bioinformatics were already proposed on GPUs, mostly on sequence comparisons. This thesis will also focus on sequence comparison, but on a particular problem, namely *the approximate pattern matching*, which may have several applications in sequence similarity tools. This problem will be discussed in the next chapters, from the description of the framework to the implementations and benchmarks.

[38] "in-vivo" means: "within a living organisim"

Chapter 2

Seed-based Indexing with Neighborhood Indexing

Seed-based heuristics have proved to be efficient for studying similarity between genetic databases with billions of base pairs. This thesis focuses on **algorithms and data structures for the filtering phase in seed-based heuristics**, with an emphasis on **efficient GPU/manycores implementation**. This chapter explains the ideas between seed-based heuristics, then presents the general framework of neighborhood indexing, which will be further implemented in chapters 3 and 4.

Contents

2.1 Similarities and seed-based heuristics

This section contains 2 parts. The first part is an introduction on similarity studying and aligment algorithms for genomic sequences. The second part presents the "seed-based heuristics", which are an efficient approach to compute alignements, and also the main target for improvement of this thesis.

2.1.1 Similarities between genomic sequences

"Similarities between genomic sequences are often traces of common ancestry, and the study of distances between species teaches us about the history of the evolution" [Pisanti et al., 2011, Introduction]. Indeed, homological breeds diverge from the common ancestry by the process of

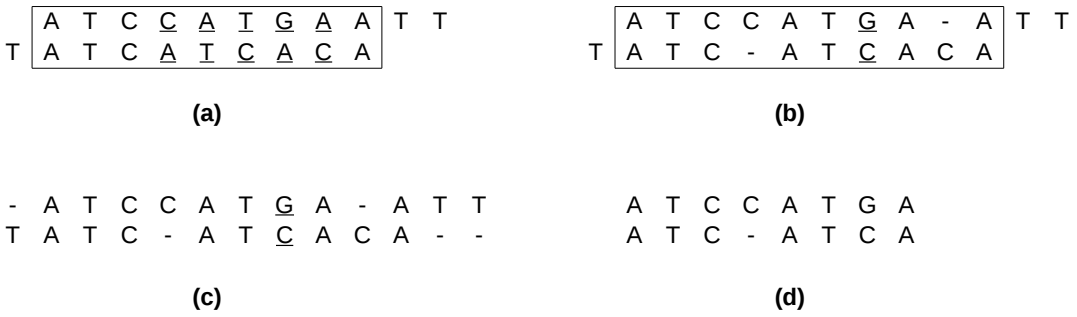

Figure 2.1: Examples of edit distance, global alignment, and local alignment between two sequences $S =$ ATCCATGAATT and $T =$ TATCATCACA. For 2 subsequences, $S' =$ ATCCATGAA in S and $T' =$ ATCATCACA in T, the Hamming distance between S' and T' is 5 (a) while the Levenshtein distance between them is 3 (b). For the whole S and T and with the score system is: $+1$ for matching, -1 for substitution and -2 for indel, the best global alignment has a score of -1 (c), the best local alignment has a score of 1 (d).

mutation and natural selection. Some basic mutational processes in the genetic sequences are the change of a residue *(substitution)*, the addition of a residue *(insertion)* or the removal of a residue *(deletion)*. The natural selection screens the mutations, thus some types of mutational processes may be more frequent than others [Durbin et al., 1998]. The homologies between the breeds can be studied from the similarities between genomic sequences.

Even when one does not directly study evolution, similarities are useful to predict homologies between sequences: similar sequences may have similar functions. Given an unknown sequence, searching similar sequences in databases may help to understand its function. Finally, similarities are at the core of tools for resequencing (see the prototype read mapper, chapter 6) where the short genetic fractions collected from the new breeds are mapped to the well known genomes to study their relations.

Formally, the **edit distance** between 2 words is the minimal number of **edits** needed to tranform one word into the other. The distance is the **Hamming distance** if only substitutions are allowed, and the **Levenshtein distance** if substitutions, insertions and deletions (so-called indels) are allowed. More elaborated **alignment distances** can be defined by assigning different **scores** for each type of point mutation (figure-2.1, a and b).

From a computational point of view, the computation of the similarity can be done with **alignment algorithms**. Given two genetic sequences and a score system, those algorithms are able to find the most optimal **global** or **local alignment** (depending on the type of analysis we do) and output the score (which provides an evalutation of the similarity) and an *alignment* (which provides a description of the similarity) (figure-2.1, c and d).

The score system, used to quantify the similarity, is usually defined based on the expert knowledge of the sequences under study in combination with the statistical models [Korf et al., 2003, chapter 4]. Some popular score systems for protein comparison are PAM (**P**oint **A**ccepted **M**utation) matrices [Dayhoff et al., 1978] and BLOSUM (**Blo**cks of Amnino Acid **Su**bstitution **M**atrix) matrices [Henikoff and Henikoff, 1992].

2.1.2 Seed-based heuristics

Well known dynamic programming based algorithms as [Needleman and Wunsch, 1970] and [Smith and Waterman, 1981] are able to find the best global and local alignments (respectively). But they are not suitable for long sequences due to their quadratic time complexity of $\mathcal{O}(mn)$ with two sequences of length m and n (figure-2.2, a). In practice, there is a need for being able to compute alignments between sequences of hundred millions to billions of base pairs. For example, the re-sequencing process usually maps tens to hundreds millions short reads[39] to the human genome which contains 25 chromosomes with a total of 2.7 billions base pairs, or even to databanks of all known sequences (Genbank[40] revision 191 contains 143 billions of base pairs for 156 millions of sequences).

The **seed-based heuristics algorithms**, such as FASTA [Lipman and Pearson, 1988] and BLAST [Altschul et al., 1990], were proposed in the late 1980s to increase efficiency against pure dynamic programming algorithms.

Seed-based heuristics are based on the assumption that two similar sequences share some identical parts. Thus, seed-based heuristics approaches consist in using relatively short words for comparing sequences to "anchor" an alignment. Those short words are named *seeds*. Once a common seed has been identified in both sequences, further *extension* is realized to get the full local alignments. The pair of common seed occurrences in two sequences is called a **candidate**. In this context, a **good candidate** means a candidate that leads to the local aligment with a score greater than or equal to a choosen threshold.

The advantage of the seed-based heuristics approach is that it can significally reduce the *search space*. With two sequences of length m and n, the size of the search space becomes $\mathcal{O}(r \times m' \times n')$, where r is the number of candidates, and m' and n' might be much less than m and n (figure-2.2, b). In practice, as the seed size is extremely shorter compared to the sequence length, the number of candidates might be numerous, containing both true positive ones and false positive ones. To improve selectivity a **filtering phase** is added to the whole seed-based heuristics aligment process. The filtering phase is usually applied to the flanking regions around the seeds: the **neighborhoods**. This phase is called: **"neighborhood filtering"** (figure-2.2, c).

The seed design, which is not addressed in this thesis, has a large impact on the sensitivity and selectivity of the seed-based heuristics alignment algorithms. For the example shown in figure-2.2, choosing the seed `AT` leads to the miss of the alignment in figure-2.2, d.

⋆ ⋆ ⋆

The design of the seeds and the full extension algorithms is not addressed here: see [Brown, 2008] for a survey of seeding for sequence alignment. This thesis focuses on *neighborhoods* of the seeds, and, more precisely, on *"the algorithms and data structures to index the seed and to do neighborhood filtering"*. We discuss in the following sections two problems:

[39] In this context, short reads mean the genetic sequences of tens to hundreds base pairs
[40] http://www.ncbi.nlm.nih.gov/genbank/

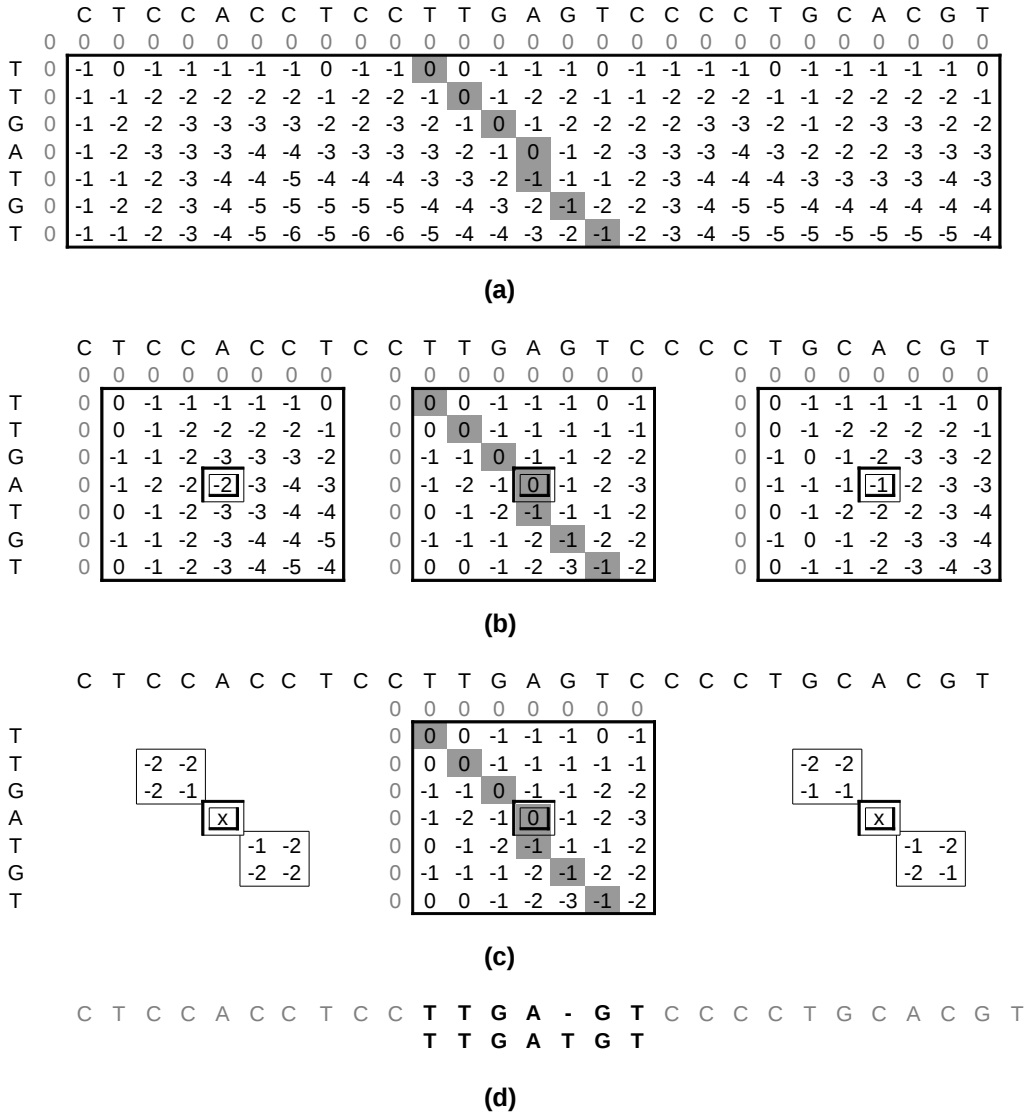

Figure 2.2: Comparision between Smith-Waterman algorithm and seed-based heuristics alignment algorithm in finding the occurences with at most 1 error of sequence $T =$ `TTGATGT` of length $m = 7$ in sequence $S =$ `CTCCACCTCCTTGAGTCCCCTGCACGT` of length $n = 27$. The score system is: 0 for matching, -1 for substitution and indels. We are here in the case of finding the best semi-global alignment, that is the best alignment between the whole sequence T and a subsequence of S. The rectangles describe the search space, thus the number of calculations needed to be performed and the memory needed to be used. (a) is the search space of dynamic programming algorithm on sequences S and T, that is $m \times n$; (b) is the search space when choosing the word `A` for the seed and then applying a dynamic programming around the candidates , that is $k \times m^2$ with k the number of occurrences of the seed is S; (c) is the search space when choosing the same seed and applying the neighborhood filtering phase with two flanking regions of size $\ell = 2$ around the candidates, that is $k \times 2 \times \ell^2$. (d) is the result, which finds an alignment with one deletion between T and the subsequence $TTGAGT$ in S.

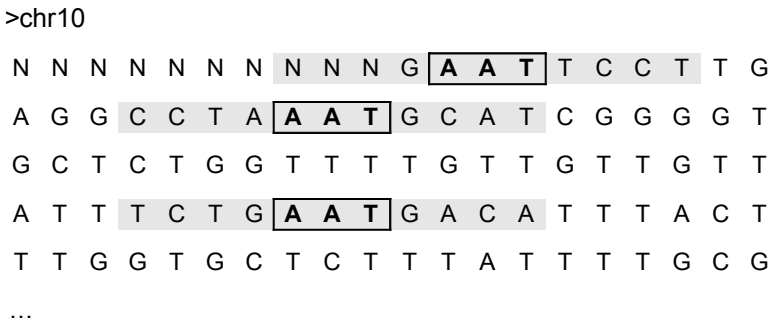

Figure 2.3: Example of seed's neighborhoods in the first 100 residues of the chromosome 10 with seed length $u = 3$ and neighborhood length $\ell = 4$. For each occurrence of the seed s (here AAT), the left and right flanking regions of 4 residues are considered as the neighborhoods.

- Seed and neighborhood retrieval (section-2.2): how to store and retrieve the occurrences and the neighborhoods of a given seed? *As in [Peterlongo et al., 2008], we choose the neighborhood indexing to efficiently access the neighborhoods.*
- Approximate neighborhood matching (section-2.3): once the candidate occurrences are retrieved with their neighborhoods, how to implement the neighborhood filtering? *The following chapters will then describe the solutions we adopted to solve those problems using manycore processors.*

2.2 Seed and neighborhood indexing

A seed s is simply defined as a contiguous word of length u over Σ. There are $|\Sigma|^u$ such different seeds. In this section, we discuss the methods to keep the list of the occurrence positions of each seed in a sequence t over Σ and their associated neighborhoods. All implementations in this thesis are used for DNA sequences, thus $\Sigma = \{A, C, G, T\}$ and $|\Sigma| = 4$.

For each occurrence of the seed in the sequence S, the two flanking regions of size ℓ will be named the *neighborhoods* (figure-2.3). In our work, only the right neighborhood, denoted by q, will be processed: We thus consider the sequence sq of size $u + \ell$. In practice, the selection of a flanking region to be indexed and the usage of this indexation depends on the design of the application. An example is further described in chapter-6. Our work can also be straightforwardly extended for the applications which need both left and right region indexing.

2.2.1 Offset and neighborhood indexing

The usual scheme to index the seeds consists to store the positions (in the sequence t) of all occurrences of each seed in a data structure (figure 2.4, left), which is called **offset indexing** in [Peterlongo et al., 2008]. Such an indexing is used in many bioinformatics tools, such as SSAHA2 ([Ning et al., 2001]), BLAT ([Kent, 2002]) or RMAP ([Smith et al., 2008]). For each query position, each candidate returned by the *seed matching* phase leads to an iteration of the

neighborhood filtering phase. This iteration accesses some neighborhoods of the positions. These memory accesses are *random*, *i.e.* unpredictable and non contiguous. As described on page 31, such accesses are not efficiently cached and require high latencies, for both CPUs and GPUs.

A way to reduce the computation time is thus to avoid as far as possible such random memory accesses. In [Peterlongo et al., 2008], a **neighborhood indexing** approach has been proposed. The idea is to directly store in the index the neighborhoods of size ℓ for every seed occurrence (figure 2.4, right). Thus all neighborhoods corresponding to a seed are obtained through a single contiguous memory access. Obviously, this index is **redundant**, as every character of the text will be stored in the neighborhoods of ℓ different seeds: the neighborhood indexing enlarges the size of the index (see below). However, it can improve the computation time by reducing the random memory access. In [Peterlongo et al., 2008], the authors claimed that the neighborhood indexing speeded up the execution time by a factor ranging between 1.5 and 2 over the offset indexing. The neighborhood indexing approach are used to implement PLAST, the aligning tools for protein sequences[41] ([Nguyen and Lavenier, 2008], [Nguyen and Lavenier, 2009] and [Nguyen, 2008]) or GASSST, the short read mapper ([Rizk and Lavenier, 2010]). Both PLAST and GASSST create the index for sequences in the running time, as a data preprocessing step. In applications we have in mind, the indexes of the sequences are created, stored on the hard disk so that they can be loaded to be reused.

Considering GPUs, the neighborhood indexing approach has also two advantages:

- Avoiding as much as possible the random data exchanges between global and local memories of the GPU.
- Allowing coalesced accesses of the work-items to global memory of the GPU.

As the neighborhood length is fixed, it is also an advantage to efficiently design the memory access pattern and the kernels. The high throughput data should be uniform and the algorithms should avoid as much as possible branch divergences, which could be caused by the heterogeneity in the input data sizes.

Size overhead of the neighborhood indexing. An obvious drawback of the neighborhood indexing is that additional memory is required to store neighborhoods. If n is the size of the sequence to be indexed:

- With the offset indexing, each offset takes $\lceil \log n \rceil$ bits, thus the index size is $n \times \lceil \log n \rceil$ bits;
- With the neighborhood indexing, and considering the nucleotide alphabet which require 2 bits to store each character, the overall index size is equal to $n \times (\lceil \log n \rceil + 2 \times \ell)$ bits, where ℓ is the length of the neighborhood.

The ratio between the overall index sizes of the neighborhood indexing and the offset indexing is thus $r_\ell = 1 + \frac{2 \times \ell}{\lceil \log n \rceil}$ [Peterlongo et al., 2008]. For alignment purposes, it is common to round $\lceil \log n \rceil$ to offsets of an integer number of bytes. Table-2.1 shows the comparision between two indexing approaches on a nucleotide sequence of 100 MB where the neighborhood lengths are $4, 8$ and 16. *The overhead of using neighborhood indexing is thus not so large.*

[41] For protein sequences, $|\Sigma| = 20$.

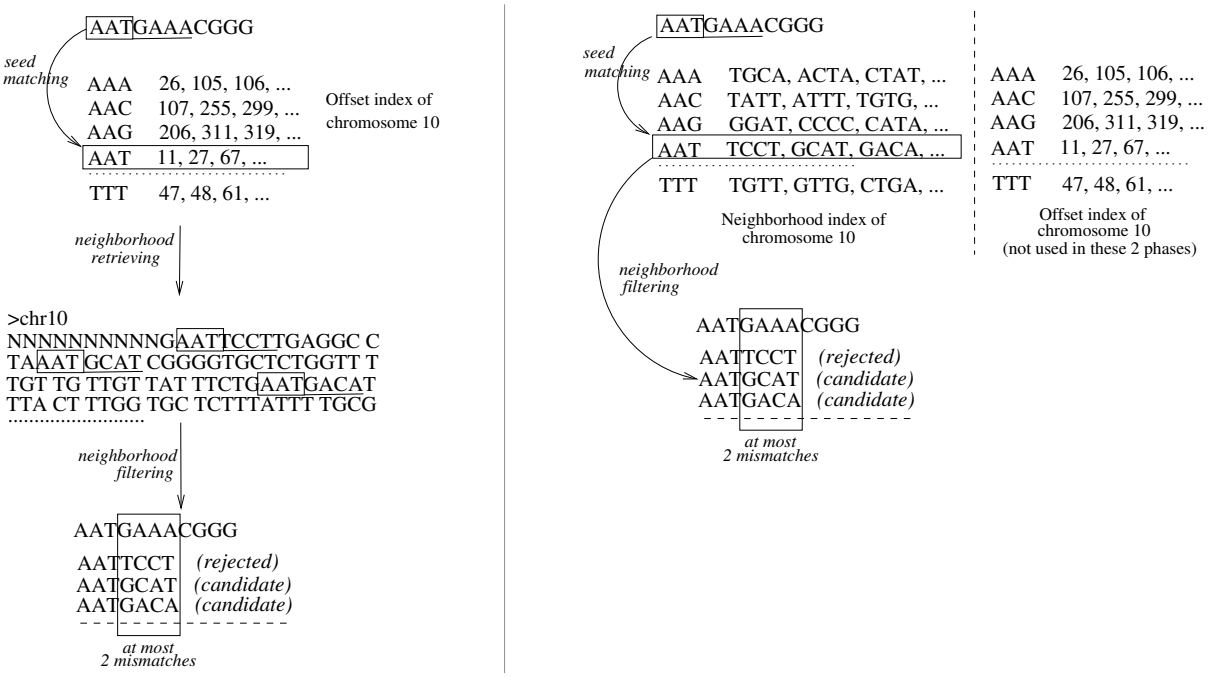

Figure 2.4: Examples of using offset indexing (left) and neighborhood indexing (right) to search for the pattern $P = \mathsf{AATGAAACGGG}$ with at most 2 substitutions in the human chromosome 10. The indexes of the chromosome is created with seed length $u = 3$ (length of s) and neighborhood length $\ell = 4$ (length of q). During the seed matching phase, the seed s (here AAT) is used to access to the index, then the filtering phase get neighborhoods of this seed to compare them against q (here GAAA) with a chosen threshold of 2 mismatches. (left) With **offset indexing approach**, it requires one memory access (to the chromosome) per occurrence to get the corresponding neighborhood, thus the random memory accesses. (right) With **neighborhood indexing approach**, all neighborhoods are stored in the contigous region, along with the offset of each seed occurrence as the independent offset index. One unique memory access allows to retrieve all the data needed by the filtering phase. Note that the offset index is also required in the neighborhood indexing approach, but it is not used in the neighborhood filtering phase.

ℓ	offset indexing (MB)	neighborhood indexing (MB)	ratio
4	400	500	1.25
8	400	600	1.5
16	400	800	2

Table 2.1: The sizes of offset indexing and neighborhood indexing of a nucleotide sequence of $100MB$. The overhead of neighborhood indexing depends on the length of neighborhood (ℓ). 32 bit machine word is used as the `unit` to represent the offset and to keep the neighborhoods.

2.2.2 Data structures for neighborhood indexing

We now present the data structures that we will use to store the neighborhoods. We will use a simple "general structure" which is a flat list of occurrences, and a "reduced structure" which contains a further level of indirection to regroup identical neighborhoods.

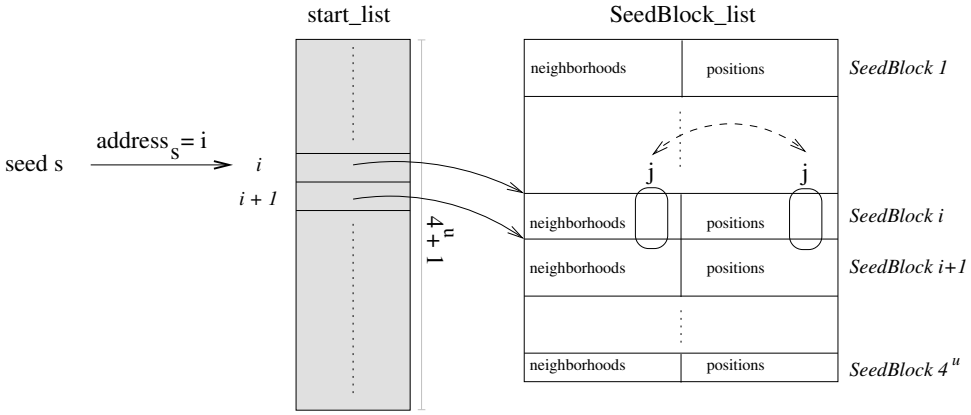

Figure 2.5: General structure of the index. It is the simplest structure, with 1 *start_list* and 1 list of SeedBlocks. From a seed s, we compute its address i. Then `start_list`$[i]$ gives the address of the neighborhoods of s in the SeedBlock_list: its SeedBlock. Each SeedBlockis a sequence of neighborhoods foloowed by the sequence of their positions. An example is given figure-2.6.

General Structure. For each seed s, the list of all its neighborhoods (and their positions) are kept in a contiguous region called SeedBlock(s). As the size of seed is u, there are 4^u SeedBlocks (figure-2.5).

We use an array called the **start_list** to keep the start positions of each SeedBlock. The address in the `start_list` associated to the seed s is a key computed from the seed:

$$\text{address}_s = \sum_{i=0}^{u-1} s[i] \times |\Sigma|^i$$

It requires only the seed key to access to the whole SeedBlocks of the given seed, from `start_list`$[\texttt{address}_s]$ to `start_list`$[\texttt{address}_s + 1]$, so we must use one additional element to keep the end position of the last SeedBlocks. Thus, the `start_list` has $4^u + 1$ elements.

In practice, we use the **machine word** as an `unit` for storing data. Here, we use the assumption that both neighborhood and position can be stored in one unit[42]. The more compact case, where multiple neighborhoods can be kept in one unit, is further presented in Section 3.1.4. In each SeedBlock, there is a $1-1$ relation between a neighborhood and its positions. If they are $\mathcal{N}$ neighborhoods of a SeedBlock, then the size of the SeedBlock is:

$$S_{\text{SeedBlock}} = 2 \times \mathcal{N} \times S_{\texttt{unit}}$$

where $S_{\texttt{unit}}$ is the size of a machine word (either 32 or 64 bits).

[42]For the alphabet $\Sigma = \{A, C, G, T\}$, a character is represented by 2 bits, so a machine word of size 32 can store a neighborhood of length up to 16. It is the case for all implementations in this thesis.

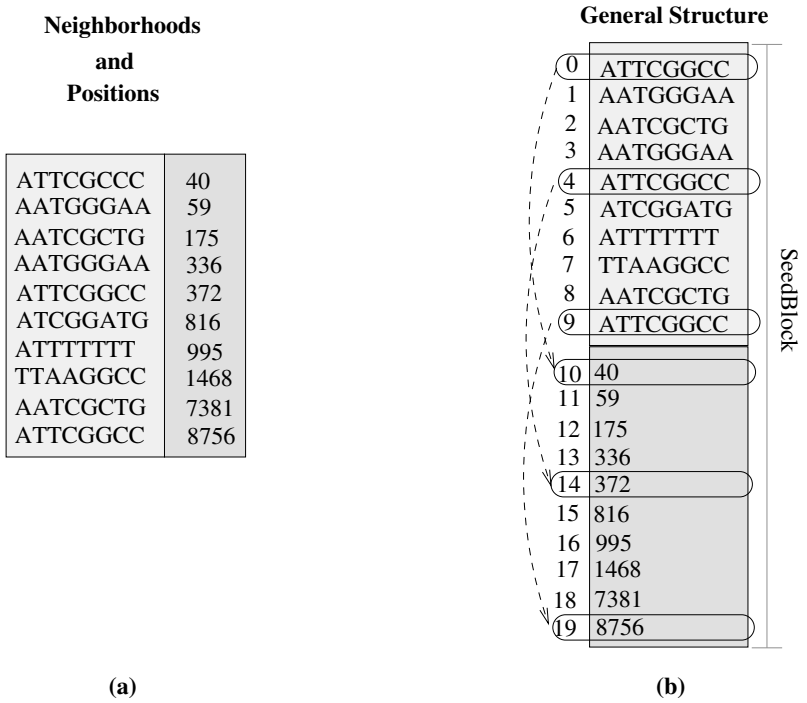

Figure 2.6: Example of a SeedBlock in the **general structure**. (a) is the neighborhoods and the seed occurrence positions, (b) is the corresponding SeedBlock made of a sequence of neighborhoods and their corresponding positions.

Reduced Structure. The disavantage of the general structure is that the neighborhoods might be redundant. We will consider a **reduced structure** where the neighborhoods are sorted and kept in the list as unique keys. The trade-off is that the data structure is now more complex, as it must divide the SeedBlock into the block of neighborhoods (the nb_block) and the block of positions (the pos_block).

As there is no longer the $1-1$ relations between a neighborhood and its position, it requires another list as the pointer to the sub-block which contains the positions of all occurrences of each neighborhood (the nb_pos_block) (figure-2.7).

Indexed Block Structure. In general structure and reduced structure, the neighborhoods are kept in flatten lists. To apply approximate matching methods, the lists must be traversed. In order to speedup the matching time, the lists can be also indexed themselves. In this case, another list is added to keep the neighborhood list index information. This leads to the **indexed block structure**. Details about this data structure and a solution using this data structure with perfect hashing are further presented in chapter-4.

2.3 Approximate neighborhood matching

The neighborhood filtering phase verifies all candidates returned from the seed matching phase by comparing the neighborhoods associated with this pair of seed occurrences. To validate a candidate, one can compute the *edit distance* between two neighborhoods (see page 44) and select it or not based on a given threshold.

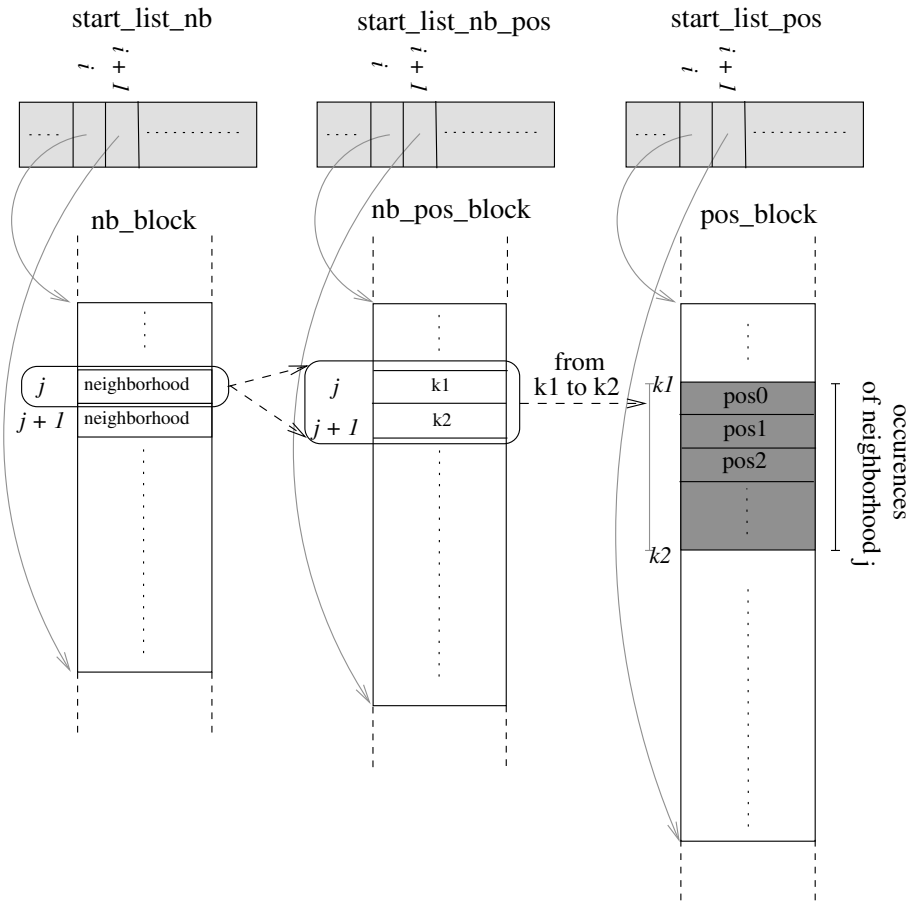

Figure 2.7: Reduced structure of the index. The data stored in a SeedBlock is now divided into 2 parts: the nb_block for the neighborhoods and the pos_block for the positions. The link between a nb_block and a pos_block is made by another list: the nb_pos_block, which stores the begining and ending positions of the occurrences. Given a seed s, we compute its address i. From `start_list_nb[`i`]` and `start_list_nb[`$i+1$`]`, we obtain the list of the neighborhood os s in nb_block. For the j-th neighborhood we obtain at positions j and $j+1$ in nb_pos_block the positions of the occurrences in pos_block. An example of the data for a SeedBlock in this structure is given in figure-2.8.

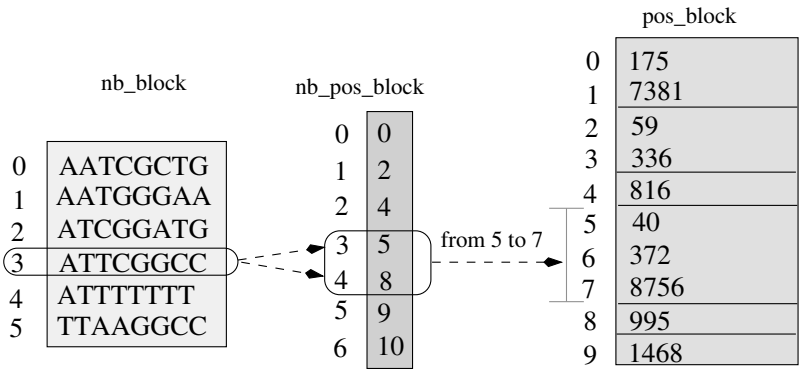

Figure 2.8: Example of a SeedBlock in **reduced structure**, based on the example of general structure in figure-2.6.

This phase can be implemented as an approximate pattern matching process over all pairs coming from two sets of neighborhoods. One neighborhood set can play the role of the texts while the other can play the role of the patterns and each pattern can be queried in turn. It is related to a generic basic problem where a *pattern* is queried against one or several *texts*, allowing some *errors*. This is the core problem of this thesis, formalized as follows:

Problem 1 (Approximate neighborhood matching) *Given a pattern q and a parameter e, find all words in a set of fixed-length words such that the edit distance with q is at most e.*

This problem can be solved following two approaches:

- **Approximate neighborhood matching based on exact matching.** From the input pattern q, the set of all patterns $\Pi_e(q)$ is generated by iterations. Each **degenerated pattern** is queried against the text with an exact matching algorithm. This method is simple, but is obviously exponential, the set $\Pi_e(q)$ having a size $\mathcal{O}(|\Sigma|^e)$. Nevertheless, taking parallelism into account, this "naive" approach for approximate matching could be the most powerful. It seems the matching process of each degenerated pattern is easy to implement and to launch simulaneously and independently on the processing elements of an OpenCL device. We will use this strategy in 3.2. A drawback of such an approach is that it can only be used for small values of e. Thus in this thesis, we will experiment it with small Hamming distance, for $e = 1$ or 2.
- **Approximate neighborhood matching by dedicated algorithm.** More generally, one can design specific algorithms able to to find the Levenshtein or other distance between two words [Navarro and Raffinot, 2002]. Of course, although it is more general than the above method, it requires suitable algorithms for the GPUs, which ideally should terminate after a predictible number of iterations and contain few branch condition statements. The BPR algorithm, which is based on bit-parallelism (3.1.1) is selected as it can satisfy these requirements.

2.4 Conclusion

Seed-based heuristics alignment process is based on three phases:

1. seed indexing,
2. neighborhood filtering,
3. and full extension.

We focus in this work on the *redundant neighborhood indexing approach* that can be effeciently used to accelerate the *filtering phase*. The main ideas in data structures and approximate pattern matching methods were also presented. The following chapter-3 will show how we use the general index structure together with an extension of the Wu-Manber bit parallelism approximate matching algorithm as well as the benefits of the reduced index structure to implement a binary search strategy. The usage of the indexed block structure using perfect hashing functions will be presented in chapter-4.

Chapter 3

Direct Neighborhood Matching

This chapter brings two solutions to the problem-1 (Section-2.2) using the index structures where **the neighborhoods are kept in the index in flatten lists**. The first part, which was presented at *Parallel Bioinformatics Conference (PBC)* [Tran et al., 2011], is about the usage of Bit Parallel Row-wise **(BPR)**, an approximate pattern matching bit-parallel algorithm proposed in [Wu and Manber, 1992a] and adapted by us for a *set of fixed length words*. The second part uses the binary search strategy.

Contents

3.1 Bit-parallel rowise algorithm

General structure and reduced structure introduced in the previous chapter are just flatten lists of the neighborhoods and of the occurrences of the seeds (page 50). It is well suited for data parallelism as each element is independent with the others. BPR is a powerful tool for approximate pattern matching using the Levenshtein distance between two words of length less than a machine word. Without branch divergence, BPR is also an efficient algorithm to process massively parallel data.

3.1.1 Bit-parallel approximate pattern matching

We start this section with the problem of generic *approximate pattern matching* between a pattern $q = q_1q_2...q_v$ of length v and a text t over an alphabet Σ as: *"Given a pattern q and a*

parameter e, find all the occurrences of q in the text t such that the distance between q and its occurrence in t is at most e."

The main difference from this problem with the core problem of the thesis ("approximate neighborhood matching", defined on page 51) is that here we look for matches of q anywhere in t (that may be here a large text) while the core problem searches for matches of q in a set of fixed-length words.

This problem can be solved by passing the characters in the text through a matching automaton, built from the pattern. The most simple type of automaton, used in this chapter, is the **prefix automaton** which can recognise all the prefixes of q with at most e errors. They are a nondeterministic automaton with $(v+1) \times (e+1)$ states, which are virtually organised as a matrix M of $e+1$ rows and $v+1$ columns (figure-3.1):

- Once a state $M_{i,j}$ is active, it means the prefix $q_1 q_2 ... q_{j-1}$ is recognized with i errors.
- For each state $M_{i,j}, 0 \leq i \leq e+1, 1 \leq j \leq v$, there are 2 (Hamming distance) or 4 (Levenshtein distance) transitions:
  - The matching transition, which reads the character q_j and changes to the state $M_{i,j+1}$. It means that the prefix $q_1 q_2 ... q_j$ is recognized with i errors.
  - The substitution transition, which reads any character in Σ, and changes to the state $M_{i+1,j+1}$. It means that the prefix $q_1 q_2 ... q_j$ is recognized with $i+1$ errors.
  - (only for Levenshtein distance) The deletion transition, which reads nothing (denoted by ϵ) and changes to the state $M_{i+1,j+1}$. It means that once the the prefix $q_1 q_2 ... q_{j-1}$ is recognized with i errors, the prefix $q_1 q_2 ... q_j$ can be automatically recognized by deleting the character q_j, thus with $i+1$ errors.
  - (only for Levenshtein distance) The insertion transition, which reads any character in Σ and changes to the state $M_{i+1,j}$. It means that the character q_j is inserted to the the prefix $q_1 q_2 ... q_{j-1}$, thus the prefix $q_1 q_2 ... q_j$ is recognized with $i+1$ errors.
- $M_{i,v+1}, 0 \leq i \leq e+1$, are the finite states, which have only (for the Levenshtein distance) an insertion transition. Once the finite state $M_{i,v+1}$ is active, the pattern is matched with i errors.
- $M_{1,1}$ is the initial state with the self-loop to express the travere of all the characters in the input text.

As this matching automaton is in the nonderterministic form, it must be either transformed into the deterministic one (with a possibly exponential number of states) or run by simulating updates of states [Holub, 2002]. One of the automaton state simulation methods is **bit-parallelism**, which emerged in the early 90's. This approach consists in taking advantage of the parallelism of bit operations, **encoding the states of a matching automaton into a machine word** seen as a bit array so that they can be updated simultaneously by one operation. Ideally, these algorithms divide the complexity by w, where w is the length of a machine word. The Shift-or algorithm for exact pattern matching [Baeza-Yates and Gonnet, 1989] is one of the first algorithms using this paradigm. In 1992, Wu and Manber [Wu and Manber, 1992a] proposed an approximate matching algorithm. The Wu-Manber algorithm (called **BPR**, for Bit-Parallelism Row-wise, in [Navarro and Raffinot, 2002]) allows substitution, insertion and deletion errors, and was implemented in the `agrep` software.

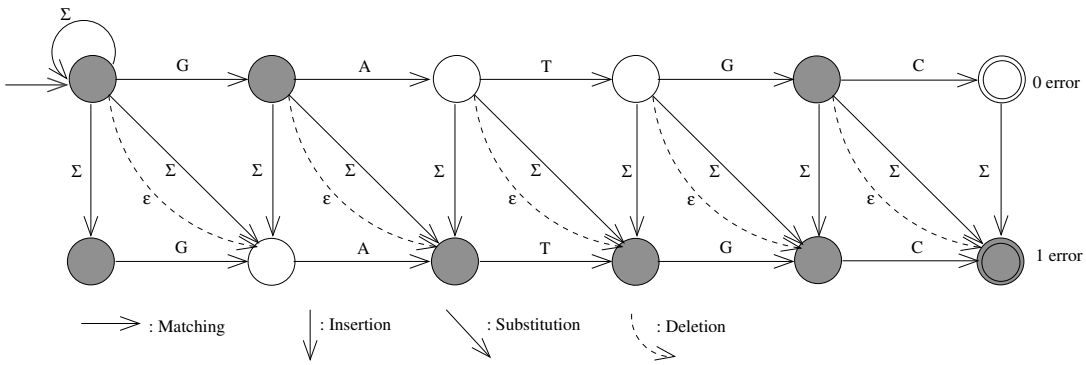

Figure 3.1: Example of the nondeterministic finite matching automaton of pattern $q = \mathsf{GATGC}$ which allows upto 1 error in the Levenshtein distance. The automaton has 2 "rows", corresponding to the exact matching case and the 1-error matching case. There are 4 types of transitions: matching (horizontal arrows), insertion (vertical arrows), substitution (diagonal arrows) and deletion (dash curved diagonal arrows). The shaded states are the active ones after parsing the text $t = \mathsf{GATG}$. One approximate matching with 1 deletion from the pattern is recognized.

Many other bit-parallel algorithms have been designed. One can refer to [Navarro and Raffinot, 2002] or [Simone and Therry, 2013] for reviews on the subject. BPR has been reported as the best unfiltered algorithm in DNA sequences, for low error levels and short patterns (p. 182 of [Navarro and Raffinot, 2002]). We thus focused on this algorithm instead of theoretically better ones such as BNDM, implemented in the `nrgrep` software [Navarro and Raffinot, 2002]. Moreover, BPR is more regular than other solutions, simple to implement on GPUs without creating divergent branches (further discussed in 5.2.5). The two following sections will present details of this algorithm and our extension for multiple fixed length words.

3.1.2 Bit-parallel rowise (BPR) algorithm

We now present the Bit Parallel Rowise (BPR) algorithm proposed in [Wu and Manber, 1992a]. This algorithm simulates each row in the matching nondeterministic finite automaton **(NFA)** as a bit vector. All states can be updated simultaneously by a combination of bit operators.

BPR Exact matching. The pattern q of length v is encoded over a bit array R of length v. Characters of the text t are processed one by one, and we denote by $R^{[j]}$ the value of R once the first j letters of the text have been read. More precisely, the i^{th} bit of $R^{[j]}$ equals 1 if and only if the first i characters of the pattern $(q_1...q_i)$ match exactly the last i characters of the text $(t_{j-i-1}...t_j)$. The first bit of $R^{[j]}$ is thus just the result of the matching of $(q_1 = t_j)$, and, when $i \geq 2$, the i^{th} bit $R^{[j]}(i)$ of $R^{[j]}$ is obtained by:

$$R^{[j]}(i) = \begin{cases} 1 & \text{if } R^{[j-1]}(i-1) = 1 \text{ and } (q_i = t_j) \text{ (match)} \\ 0 & \text{otherwise} \end{cases}$$

With bitwise operators and (&) and shift ($\ll$), this results in Algorithm 1.

The *pattern bitmask* B is a table with $|\Sigma|$ bit arrays constructed from the pattern, such that $B[t_j](i) = 1$ if and only if $(q_i = t_j)$. This algorithm works as long as $v \leq w$, where w is the length of the machine word, and needs $\mathcal{O}(z)$ operations to compute all $R^{[j]}$ values, where z is the length of the text.

$$\begin{cases} R^{[0]} & \leftarrow & 0^v \\ R^{[j]} & \leftarrow & \left((R^{[j-1]} \ll 1) \mid 0^{v-1}1\right) \ \& \ B[t_j] \end{cases}$$

Algorithm 1: Exact Bit-Parallel Matching

BPR Approximate matching. To generalize the matching up to e errors, we now consider $e+1$ different bit arrays $R_0, R_1, ...R_e$, each one of length v. The i^{th} bit of $R_k^{[j]}$ equals 1 if and only if the first i characters of the pattern match a subword of the text finishing at t_j with at most k errors, leading to Algorithm 2. Due to insertion and deletion errors, the length of a match in the text is now in the interval $[v-e, v+e]$. This algorithm works as long as $v+e \leq w$, but now takes $\mathcal{O}(ez)$ time. An illustration of this algorithm is given figure-3.2.

$$\begin{cases} R_k^{[0]} & \leftarrow & 0^{v-k}1^k \\ R_0^{[j]} & \leftarrow & \left((R_0^{[j-1]} \ll 1) \mid 0^{v-1}1\right) \ \& \ B[t_j] \quad \text{(match)} \\ R_k^{[j]} & \leftarrow & \underbrace{\left((R_k^{[j-1]} \ll 1) \ \& \ B[t_j]\right)}_{\text{(match)}} \mid \underbrace{R_{k-1}^{[j-1]}}_{\text{(insertion)}} \mid \underbrace{(R_{k-1}^{[j-1]} \ll 1)}_{\text{(substitution)}} \mid \underbrace{(R_{k-1}^{[\mathbf{j}]} \ll 1)}_{\text{(deletion)}} \mid \underbrace{0^{v-k}1^k}_{\text{(init)}} \end{cases}$$

Algorithm 2: BPR Matching

This algorithm can easily be changed in the cases that not all 3 types of edits are allowed. For example, if only the substitution is allowed (the Hamming distance), we remove the components corresponding to the deletion and the insertion, thus the update of $R_k^{[j]}$ is:

$$R_k^{[j]} \leftarrow \underbrace{\left((R_k^{[j-1]} \ll 1) \ \& \ B[t_j]\right)}_{\text{(match)}} \mid \underbrace{(R_{k-1}^{[j-1]} \ll 1)}_{\text{(substitution)}} \mid \underbrace{0^{v-k}1^k}_{\text{(init)}}$$

3.1.3 Multiple fixed length bit-parallel rowise (mflBPR) algorithm

In this section, we propose an extension of BPR to solve the core problem of "Approximate Neighborhood Matching" (defined on page 51), taking account of memory accesses into consideration. We will call this algorithm **mflBPR**, for "multiple fixed length Bit Parallel Rowise". Formally, we compare here a pattern q of length v against a collection of words t^1, t^2, ... t^n of length $\ell = v+e$, allowing at most e errors (substitutions, insertions, deletions).

The existing bit-parallel algorithms for multiple pattern matching (reviewed in Section 6.6 of [Navarro and Raffinot, 2002]) match a set of patterns within a large text. Our setup is different, as we want to match one pattern with several texts. Of course, one could reverse the multiple pattern matching algorithms and build an automaton on a set of all neighborhoods. This would result in a huge automaton, and the algorithm would not be easily parallelizable.

The idea of mflBPR is to store h fixed-length words into a machine word, so h matchings can be done simultaneously. As in BPR, to have a matching up to e errors, we consider $e+1$ different bit arrays $R_0, R_1, ...R_e$, but each one is now of size vh, that is h slices of v bits. If

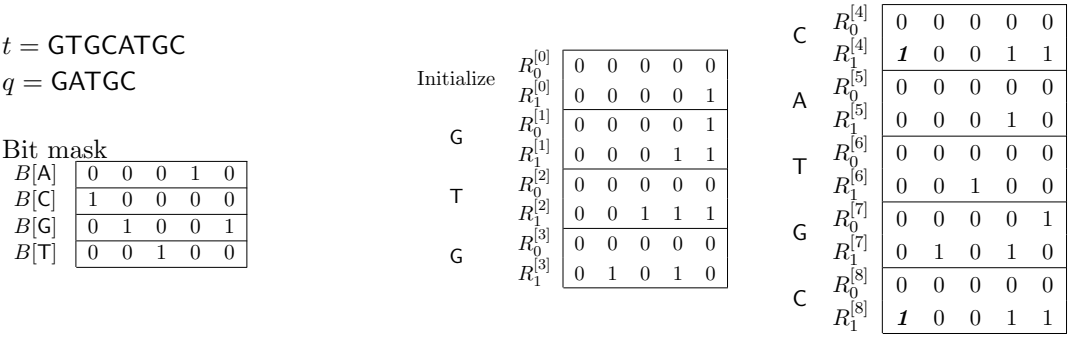

Figure 3.2: Example of Bit Parallel Rowise matching algorithm (BPR), searching the patern $p = \mathsf{GATGC}$ in the text $t = \mathsf{GTGCATGC}$. On the left is the bit mask corresponding to p. On the right is the updates of simulation matrix R, corresponding to the approximate matching automaton in figure-3.1, after the read of each character in t. There are two matches with 1 error. The first match is found after reading the character $t_{(4)}$, starts from $t_{(1)}$ to $t_{(4)}$: matched with 1 deletion from the pattern. The second match is found after reading the character $t_{(8)}$. There are 2 possibilities for this match: (1) starts from $t_{(5)}$ to $t_{(8)}$: matched with 1 insertion to the pattern or (2) starts from $t_{(4)}$ to $t_{(8)}$: matched with 1 substitution in the pattern.

$1 \leq r \leq h$ and $1 \leq i \leq v$, the i^{th} bit of the r^{th} slice of $R_q^{[j]}$ equals 1 if and only if the first i characters of the pattern match the last i characters of the r^{th} text $(t_{j-i-1}^r ... t_j^r)$ with at most k errors. We thus obtain Algorithm 3.

$$
\begin{cases}
R_k^{[0]} \leftarrow (0^{v-k}1^k)^h \\
R_0^{[j]} \leftarrow \left((R_0^{[j-1]} \ll 1) \mid (0^{v-1}1)^h \right) \ \& \ \widehat{B}[\widehat{t_j}] \quad \text{(match)} \\
R_k^{[j]} \leftarrow \underbrace{\left((R_k^{[j-1]} \ll 1) \ \& \ \widehat{B}[\widehat{t_j}] \right)}_{\text{(match)}} \mid \underbrace{R_{k-1}^{[j-1]}}_{\text{(insertion)}} \mid \underbrace{(R_{k-1}^{[j-1]} \ll 1)}_{\text{(substitution)}} \mid \underbrace{(R_{k-1}^{[j]} \ll 1)}_{\text{(deletion)}} \mid \underbrace{(0^{v-k}1^k)^h}_{\text{(init)}}
\end{cases}
$$

Algorithm 3: Multiple Fixed-Length BPR (mflBPR) Matching

Figure 3.3 shows a run of this algorithm. Compared to BPR, the initialization is $(0^{v-k}1^k)^h$ instead of $0^{v-k}1^k$. This initialization puts 1's at the k first bits of each slice, thus overriding any data shifted from another slice. Moreover, to allow better memory efficiency:

- The set of n fixed-length words $t = \{t^1, t^2, ..., t^h\}$ is stored and accessed through a stripped layout, as each access j returns the j^{th} characters of every word: $\widehat{t_j} = t_j^1 t_j^2 ... t_j^h$
- The block mask $\widehat{B}[t_j^1 t_j^2 ... t_j^h] = B[t_j^1]B[t_j^2]...B[t_j^h]$ is thus now larger, having $|\Sigma|^h$ bit arrays (instead of $|\Sigma|$). As in BPR, the computation of this table still depends only on the pattern. This table is somehow redundant, but at least it now allows the match of h characters with one unique memory access: This is designed to fit our manycore hardware.

mflBPR works as long as $vh \leq w$, where w is the length of the machine word, and needs $\mathcal{O}(ez/h)$ operations to compute all $R_k^{[j]}$ values, where z is the total length of all texts. Comparing to the BPR algorithm, there are h times less operations. Of course, the limiting factor is again the size of a machine word.

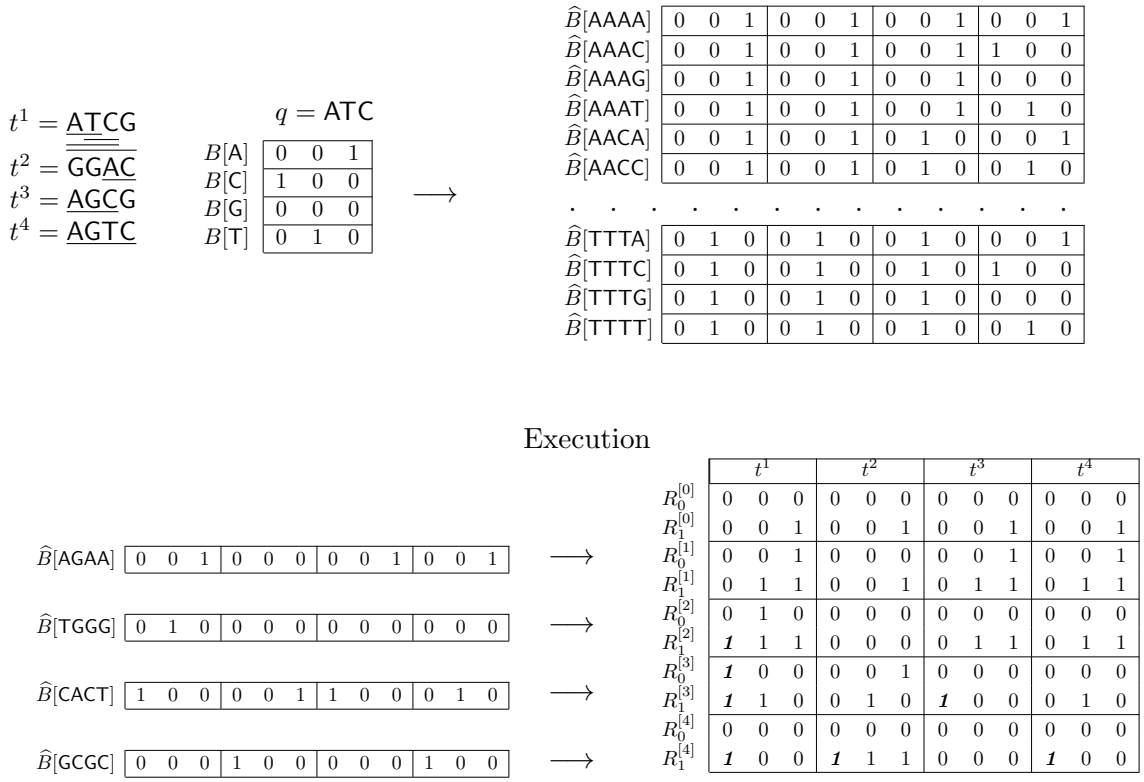

Figure 3.3: Execution of the algorithm 3 on 12-bit machine words. The pattern $q = \mathsf{ATC}$, of length $v = 3$, is compared against $h = 4$ words with up to $e = 1$ error. The block mask $\widehat{B}$ of q is generated based on the bit mask B. The text data $t = \{\mathsf{ATCG}, \mathsf{GGAC}, \mathsf{AGCG}, \mathsf{AGTC}\}$ is stored in a stripped layout as $\{\mathsf{AGAA}, \mathsf{TGGG}, \mathsf{CACT}, \mathsf{GCGC}\}$. After 2 iterations, there is one approximate match for t^1 (AT, one insertion). After 3 iterations, there is one exact match for t^1, and one approximate match for t^3 (AGC, one substitution). After 4 iterations, there are three approximate matches, for t^1 (ATCG, one deletion), t^2 (AC, one insertion) and t^4 (AGTC, one deletion).

3.1.4 Implementing BPR and mflBPR on OpenCL devices for approximate neighborhood matching

Let w the size of the machine word, ℓ the size of the neighborhood and b the number of bits encoding a character in Σ, the maximal number of neighborhoods in a machine word is:

$$h = \left\lfloor \frac{w}{\ell \times b} \right\rfloor$$

In the framework of using neighborhood indexing for the seed-based heuristic alignment, the lists of the neighborhoods and of the occurrences of a seed s are kept in the $\mathsf{SeedBlock}(s)$ (see section 2.2.2). With $\mathcal{N}$ the number of neighborhoods, and a machine word of 32 bits (4 bytes) as the `unit` to store both the neighborhood and the position, the total size of a $\mathsf{SeedBlock}$ is:

$$S_{\mathsf{SeedBlock}} = \left(\left\lfloor \frac{\mathcal{N}}{h} \right\rfloor + \mathcal{N} \right) \times S_{\texttt{unit}}$$

ℓ	h	nbBlock size (MB)	posBlock size (MB)	SeedBlock size (MB)
4	4	0.5	2	2.5
8	2	1	2	3
16	1	2	2	4

Table 3.1: The maximal number of neighborhoods (h) in a 32 bit machine word and the sizes of a SeedBlock for the general index of a nucleotide sequence of length 128Mbp in 3 cases of neighborhood length with seed length $u = 4$.

The position pos_i of the i^{th} neighborhood is calculated as:

$$\text{pos}_i = \left(\left\lfloor \frac{\mathcal{N}}{h} \right\rfloor + i \right) \times S_{\texttt{unit}}$$

The average value of a SeedBlock depends on the length of the text (n) and the seed length (u): $S_{\mathsf{SeedBlock}} \sim n/4^u$. Table-3.1 gives examples of h values and $S_{\mathsf{SeedBlock}}$ size for the general index of a 128Mbp nucleotide sequence, with $u = 4$ and for several ℓ.

The problem-1 can be solved by applying either BPR or mflBPR over all the elements in the neighborhood list partition of the SeedBlock. This index is intrinsically parallel, as the neighborhoods are processed independently. Figure-3.4 depicts the diagram of using BPR or mflBPR on general index structure over OpenCL device. All the index data are precomputed and transferred only once to the device. Then the application runs looping on each query:

- The pattern pre-processing as well as the SeedBlock(s) position retrieving is done on the host.
- The block bitmask $B(q)$ or $\widehat{B}(q)$ and the positions of SeedBlock(s) are sent to the global memory of the device.
- The device is devoted to the neighborhood filtering phase. Depending on the size of SeedBlock(s), several comparing cycles may be run (dashed arrow on figure-3.4). In each comparing cycle, neighborhoods are distributed over different work groups and loaded in the local memory of each work-group, and processed by several work-items. The positions of the matching neighborhoods are then written back to a result array in the global memory. They are transferred back to the host once all comparing cycles are finished.

3.2 Binary search (BS)

This section is about approximate pattern matching based on exact matching. We use the binary search strategy to find the occurrences of all substitution patterns (thus considering the Hamming distance). The patterns are generated from the original word from the list of neighborhoods. This approach is suitable only for the cases of small Hamming distance, over a small alphabet, and we will see in chapter 5 that it is highly efficient in these cases.

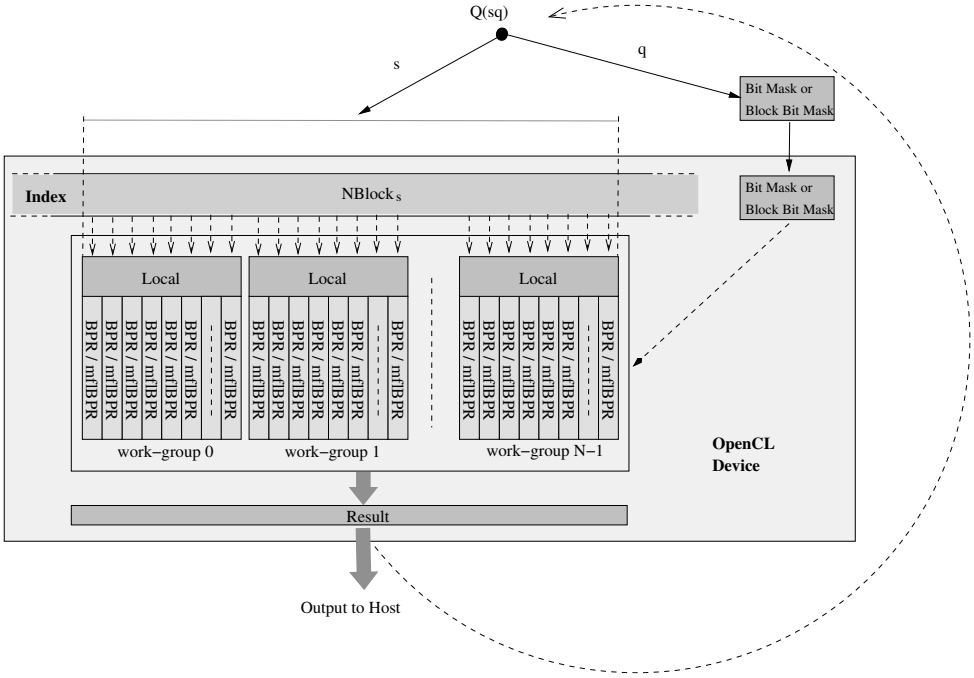

Figure 3.4: Using BPR and mflBPR with OpenCL devices. The index is written and kept in the global memory of the OpenCL device. For each query sq, the bit mask or block bit mask corresponding to q is calculated and transfered to the OpenCL device. The SeedBlock corresponding to s is divided and written to the local memory of the work-groups. Each work-item in a work-group processes a number of independent neighborhood simultaneously. The input queries are processed in turns, represented by the dashed line arc in the diagram.

3.2.1 Generating degenerated patterns

Given a pattern q of length v and a parameter $e \geq 0$ over an alphabet Σ, we have to generate the set of degenerated patterns $\Pi_e(q)$ which have a Hamming distance of at most e with q. It can be solved by substituting the characters in e different positions of the pattern q. We denote by $\mathrm{SP} \subset [1, v]^e$ the set of combinations of e positions of the pattern. Each element $(i_1, \ldots, i_e)$ of SP denotes a set of positions to be substituted in q. The size of SP is:

$$|\mathrm{SP}| = \binom{v}{e} = \frac{v!}{e!(v-e)!} = \mathcal{O}(v^e)$$

For each substituting position in the pattern q, there are $|\Sigma| - 1$ different characters to be changed in turn. Thus there are:

$$|\Pi_{=e}(q)| = |\mathrm{SP}| \times (|\Sigma| - 1)^e = \mathcal{O}(v^e \cdot |\Sigma|^e)$$

degenerated patterns with *exactly* e substitution errors. To avoid branch divergence in the algorithm, and to generate patterns with $< e$ errors, we will also subtitute each character by itself, giving $|\Sigma|$ different characters for each position. It means that we allow redundant computations to have a simpler algorithm which do not require conditional statements. We denote by SC the set of combinations of e letters from Σ. An element $(\alpha_1, \ldots, \alpha_k) \in \Sigma^e$ of SC means that the character q_{i_k} will be substituted by α_k. The size of SC is:

$$|\mathrm{SC}| = |\Sigma|^e$$

	1	2	3
4	16	96	256
8	32	448	3584
16	64	1920	35840

Table 3.2: The number of degenerated patterns with $1, 2, 3$ substitution errors for $v = 4, 8, 16$

Each couple of elements in SP×SC will map to an elemnt of $\Pi_e(q)$. For example, let $q = \mathsf{AGCTAAGT}$, $\Sigma = \{\mathsf{A},\mathsf{C},\mathsf{G},\mathsf{T}\}$:

- $((1,3),(\mathsf{T},\mathsf{G}))$ produces the degenerated pattern **T**G**G**TAAGT (2 errors from q)
- $((1,2),(\mathsf{A},\mathsf{A}))$ produces the degenerated pattern **AA**GTAAGT (1 error from q)

This mapping is not injective, as all degenerated pattern with $< e$ errors will be generated several times : $((2,5),(\mathsf{A},\mathsf{A}))$ produces the same degenerated pattern with 1 error. Nevertheless the size of $\Pi_e(q)$ can be bounded by:

$$|\Pi_e(q)| \leq |\mathrm{SP}| \times |\mathrm{SC}| = |\mathrm{SP}| \times |\Sigma|^e = \mathcal{O}(v^e \cdot |\Sigma|^e)$$

In practice, all elements of $\Pi_e(q)$ can thus be generated in turn by iterations with the complexity of $\mathcal{O}(v^e \cdot |\Sigma|^e)$. The parallel solution to generate and process $\Pi_e(q)$ is presented in the following section. As shown in table-3.2, the number of degenerated patterns grows exponentially: it will become eventually greater than the maximal number of work-items in each work-group. Of course, one could also serialize the process on some work-items, but we choose to implement this technique with a unique work-item for each element of SP × SC. We choose to consider only the case with $0 \leq e \leq 2$. When the Hamming distance is greater than 2, we can still use BPR allowing only substitutions.

3.2.2 Implementating binary search on OpenCL devices

Given a query in the form of sq, where s is the seed part and q is the neighborhood part. The key of s is used to locate the $\mathsf{SeedBlock}(s)$ on which all elements of $\Pi_e(q)$, generated from q, are queried by the binary search. The whole work which contains $|SP| \times |SC|$ independent processing elements can be divided as follows:

- The whole work is processed by a block of |SC| work-groups;
- Each work-group processes a whole set |SP| for one element in SC;
- Each work-item processes one element of SP for the element of SC of its work-group.

Parallelizing the Hamming-distance degenerated patterns Once a kernel is launched as multiple instances in the OpenCL index space, each work-item can use the native command to get its local identifiers (l_id) and the identifiers of its work-group (gr_id). These two identifiers are used to calculate the elements of SP and SC.

- $e = 0$: exact matching case, $\Pi_e(q) = \{q\}$.

- $e = 1$: $|\text{SP}| = v$ and $|\text{SC}| = \Sigma$.
  The element of SP × SC for a given work-item is:

$$(l\_id \ \texttt{mod} \ v), \quad (gr\_id \ \texttt{mod} \ |\Sigma|)$$

- $e = 2$: $|SP| = \frac{v(v-1)}{2}$ and $|SC| = |\Sigma|^2$.

  To compute efficiently the position of substituing characters, we precompute a complementary data structure, called the **position substituting list (L_{Pos})**. The construction of L_{Pos} only depends on v. Let M_{sub} be a $v \times v$ matrix ranging from $(0, 0)$ to $(v-1, v-1)$ such that $M_{sub}(i, j) = iv + j$, we construct L_{Pos} as a 1 dimension array ranging from 0 to $|SP| - 1$ made of the upper right triangular values of M_{sub}, excluding the main diagonal. Let $gr\_sub\_id = gr\_id \ \texttt{mod} \ |SC|$, the element of SP × SC for a given work-item is:

$$(L_{Pos}[l\_id] \ \texttt{div} \ v, L_{Pos}[l\_id] \ \texttt{mod} \ v), \quad (gr\_sub\_id \ \texttt{div} \ |\Sigma|, gr\_sub\_id \ \texttt{mod} \ |\Sigma|)$$

  An example for $v = 4$ is given on figure-3.5. The advantage of this method is that L_{Pos} depends only on v and e, which are constants for the kernel. L_{Pos} can thus be built in the host and then transfered to the kernel.

In all these cases, each work-item can build its element of SP × SC $\Pi_e(q)$ by some basic operators as `mod` and `div` in constant time. Once this element is known, the work-item can do the binary search in time $O(\log(\mathcal{N})$. This technique could be extended for error numbers $e > 2$, but again the exponential size of $\Pi_e(q)$ would become a limiting factor.

Usage with OpenCL devices Figure-3.6 depicts the usage of binary search with the Hamming distance patterns generating model on the OpenCL devices. All the index data are precomputed and transferred only once to the device. The input queries, each in the form of sq, are divided into batches of size N. The constants related to the configuration of the kernel such as: |SP|, |SC|, L_{Pos}, *etc.* are calculated in the host and written to the constant memory region of the device. Then, the application runs looping on each batch of queries:

- Each work-item follows those steps:
  - Use the key of s to get the address of the corresponding SeedBlock(s).
  - Generate the degenerated pattern from q based on its local identifier and group identifier.
  - Search for the degenerated pattern in SeedBlock(s) using the binary search.
- The positions of the matching neighborhoods are then written back to a result array in the global memory before being transfered back to the host.

Comparing with the usage of BPR and mflBPR on OpenCL devices, this strategy has both advantages and disadvantages:

- The search can be done in time $\mathcal{O}(v^e \log(\mathcal{N}))$, where $\mathcal{N}$ is the size of the SeedBlock (see Table 3.3). For small Hamming distance, the v^e factor can be efficiently parallelized, as presented above;
- The size of a SeedBlock can be greater than the size of the local memory. In this case, it leads to random accesses to the global memory when doing the binary search.

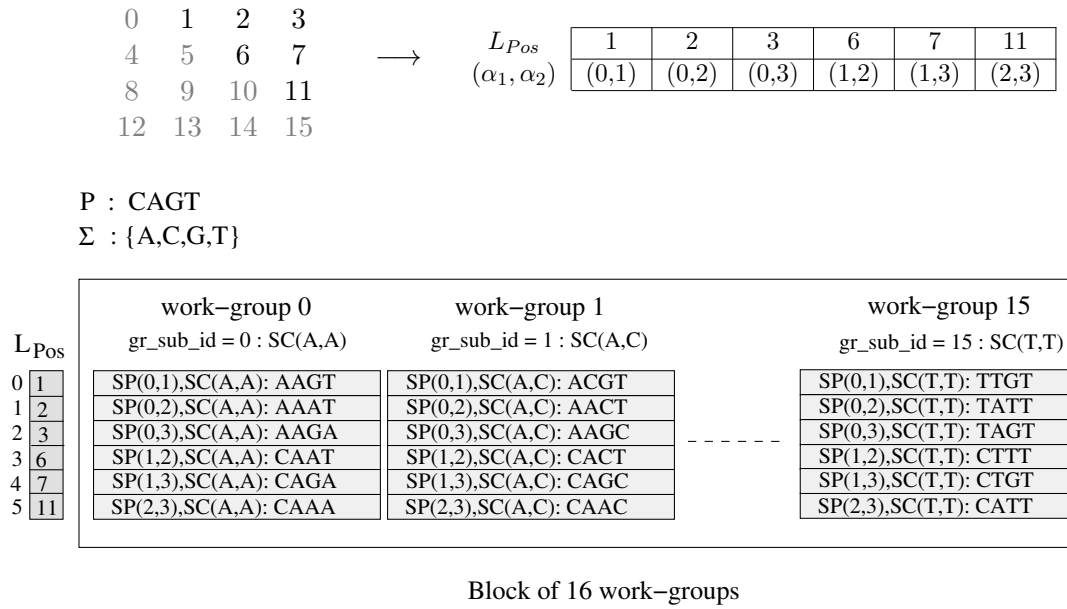

Figure 3.5: Parallelizing the Hamming-distance pattern generating model for $e = 2$ errors and a pattern of size $v = 4$ (*note that in computing, all character position start from 0*). The substituting list L_{Pos} has 6 elements whose values are the upper-right diagonal triangle of a 4×4 matrix. For a pattern of size 4 over $\Sigma = \{A, C, G, T\}$, we use a block of 16 work-groups, corresponding to elements (α_1, α_2) of SC: $\{(A, A), \ldots, (T, T)\}$. Each work-group contains 6 work-items with the local identifier from 0 to 5. This figure shows an example of generating all edited patterns which have the Hamming-distance of at most 2 with the pattern $q = \mathsf{CAGT}$.

	Total time complexity (`uint` operations)	Global memory access
Generate P_e	$\mathcal{O}(v^e)$	1
Binary Search	$\mathcal{O}(v^e \cdot \log(\mathcal{N}))$	$\leq v^e \cdot \log(\mathcal{N})$

Table 3.3: The time complexity, the number of `uint` operators and the memory access number of a binary search query. This complexities are divided by v^e when using several work-items as described in the text.

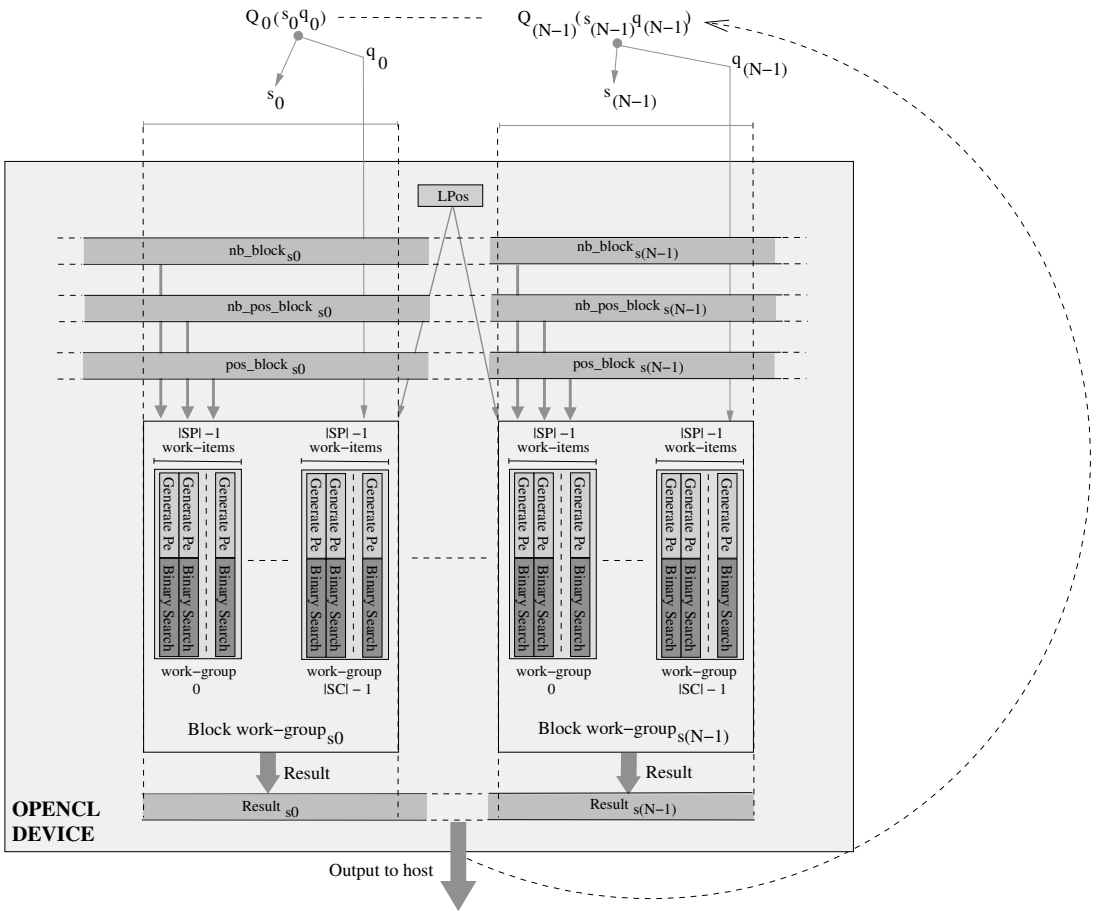

Figure 3.6: Using binary search with reduced index on OpenCL devices. L_{Pos} is written to the constant memory, the other structures of the reduced index: `nb_block`,`nb_pos_block`,`pos_block` (2.2.2) are written to the global memory. The input queries are divided into batches of size N, which are processed in turns as described by the dashed line arc in the diagram.

3.3 Conclusion

Considering data structures where the SeedBlocks are stored in flatten lists, the approximate pattern matching against these lists of neighborhoods can be done simultaneously on the OpenCL device by either a dedicated algorithm such as BPR or the use of an exact matching algorithm for the set of all degenered patterns at a given Hamming distance. We proposed and implemented a new algorithm, mlfBPR, a parallel extension for BPR. Results of this algorithm and comparison with BPR will be discussed in chapter 5. We will see that mflBPR outperforms BPR, with a speedup close the theoretical ratio h. However, in the case of small Hamming distances, which is the case of many applications such as for the read mapper presented in chapter 6, the binary search performs better than both BPR and mflBPR.

Keeping neighborhoods in flatten list in the general structure and the reduced structure has the disadvantage that the SeedBlocks has to be traversed to find the occurrences, either exhaustively or dichotomously. Thus the compute time depends on the size of the SeedBlock of the considered seed. The next chapter will propose to index also the neighborhoods.

Chapter 4

Neighborhood Indexing with Perfect Hash Functions

We first remember that the core problem of this thesis is to efficiently retrieve and (approximatively) compare neighborhoods stored in a seed-based index.

In the previous chapter, we investigated solutions when the neighborhoods of a given seed were kept in a flatten lists sorted or not. This chapter presents another approach. We will use hashing in order to *index the neighborhoods*. This means that we now use a two-stage index: the main index, given a seed s, returns as before a neighborhood block $\mathsf{SeedBlock}(s)$, but this block is further stored in a indexed way. To allow very efficient querying, in constant time, we will use techniques of *perfect hashing*.

The chapter starts with an introduction on perfect hashing techniques, then explains how this is implemented to solve approximate neighborhood matching.

Contents

4.1 Motivation

In the previous chapter, given a seed s, the neighborhood block $\mathsf{SeedBlock}(s)$ needs to be traversed to apply the matching algorithms. In these 2 cases, the processing time depends on $\mathcal{N}$, the size of $\mathsf{SeedBlock}(s)$: either $\mathcal{O}(\mathcal{N})$ or $\mathcal{O}(\log(\mathcal{N}))$ for exact matching, and either $\mathcal{O}(\mathcal{N})$ or $\mathcal{O}(\log(\mathcal{N}) \cdot v^e)$ for approximate pattern matching.

The idea of this chapter is to further index the neighborhoods in order to be able to retrieve the occurrences of a neighborhodd in constant time. This would allow us to compute the exact matching in $\mathcal{O}(1)$ time and the approximate matching in $\mathcal{O}(v^e)$ time. One good technique is to use *hash functions*: given a neighborhood q, a hash function can effectively gives a hashed value $addr = h(q)$ in constant time, "where" we can store occurrence information.

The problem of hash collisions. The disadvantage of using hash functions is the eventuality of **collisions** where multiple neighborhoods q are hashed to the same $addr$. How can the collisions be handled ?

1. Usually, librairies using hashing techniques explicitely handle collisions. A common solution is to use a linked list at the value $addr$. Another solution is the linear hashing [Litwin, 1980], which continues generating a new hashed value $addr'$ from the current value $addr$ by a deterministic function, until collision is ended. In both solutions, the problem is that the access time to the value does not remain constant, and can be different between elements. For implemention on GPUs with the diagram presented figure-3.6 page 66, this drawback can cause branch divergence because of a difference in the number of random accesses to the global memory, like the binary search.

2. Some techniques further lower the expected number of collisions. For example, when one only wants to test the appartenance of key to a set, *Bloom filters* reduce the number of collisions by combining several hash functions [Bloom, 1970]. For Bloom filters, the query of the existence of a key in a set (the filter step) can be done in constant time. Although this techniques avoids false negatives, it leads to some false positives. Such filters need further post-processing to effectively access the values and to check the false positives. Moreover, another disadvantage of Bloom filters is that the addresses (or the positions) of the keys in the set are not stored. It means that *to check the false positives of the filter Bloom and to obtain the address of the true existence keys, it require to do an additional searching process over the set (the search step).* Even if the number of the keys needed to be search could be much less than the original queried keys, it leads to the following problems, especially considering implementations on GPUs:

   - The disadvantages of the direct neighborhood matching approach (chapter-3) still remain. Even if the number of keys that passed through the filter is very few, it still causes branch divergence, thus there could be a lot of waste due to work-items that terminate after the filter phase and have to wait for the other ones (in the same warps) doing the searching step.
   - In cases where the number of keys that passed the filter is numerous, it leads to waste time for the filter step.

We made some preliminary tests with Bloom filters in cooperation with the BPR/mflBPR as the search step, but the performance was very poor (results not shown). Moreover, it leads

to additional complexity to the program structure.

Related works on GPU-bashed Hashing. A number of works on mapping popular hasing algorithms onto GPU have been anounced, such as "Open adressing" [Alcantara, 2011, chapter 3], "Chaining" [Alcantara, 2011, chapter 4], "Cuckoo hashing" [Alcantara, 2011, chapter 5,6], "Multidimensional linear dynamic hashing" [Liu et al., 2012b]. One point of focus of these works is on the speed of the hash table building phase and on subsequent dynamic updates. However, the need for collision handle in the key retrieval phase still remains (as in the original serial hashing algorithms). Although the queries can be executed in parallel, there is always the possibility of nondeterministic accesses to the hash table, which can lead to both random memory accesses and branch divergence problems.

In addition to such problems, these works are different from the main purpose of this thesis in that the index is constructed only once without requiring dynamics update: we focus here more on the key retrieval phase, which must be efficiently parallelized on the GPU.

The selection of using perfect hash functions. We finally decided to use another technique, by using *perfect hashing functions* which allow *no collision at all*. In this case, the test of an exact match of a neighborhood in $\mathsf{SeedBlock}(s)$ can be done exactly in constant $\mathcal{O}(1)$ time, with a fixed number of memory accesses. Moreover, it fits particularly well on the GPU, because it requires *intensive but homogeneous* computational operations.

Formally, let $\mathcal{U}$ be the universe of keys (in our context, the neighborhoods, that is words of length v), let $\mathcal{S} \subseteq \mathcal{U}$ be a subset of n **keys** and let $\mathcal{V} = [0..m-1]$ be an interval of integers called the **values**. A **hash function** is a function $h : \mathcal{U} \to \mathcal{V}$ that maps the set of keys $\mathcal{S}$ into $\mathcal{V}$. A **perfect hash function** is a one-to-one hash function: two different keys maps two different values. If $m = n$, then we say that h is a **minimal perfect hash function**.

Once the elements in the set are indexed by a perfect hash function, a $\mathcal{O}(1)$ access is guaranted but the size of the list used to store the elements may not be optimal (further discussed in Section 4.2.5). A minimal perfect hash function also guarantees that the final list of elements is of minimal size. However, computing and using minimal perfect hash functions requires an additional data structure and some further computing steps, particularly for the algorithm used in this thesis: BDZ_{PH}, which will be presented in the following section. It is the main reason why we choose to use perfect hash functions, instead of minimal ones.

Section 4.2 will explain how to build perfect hash function with the $\text{CHM}_{\text{PH}}/\text{BDZ}_{\text{PH}}$ algorithm. Then we will see, in Section 4.3, how to use these functions for our problem of approximate neighborhood matching.

4.2 Random hypergraph based algorithms for constructing perfect hash functions

This section is a brief summary about the methods for designing perfect hash function based on random hypergraph. We describe here the BDZ_{PH} algorithm [Botelho et al.,], which is largely based on the the CHM_{PH} algorithm [Majewski et al., 1996] (Note that in this work, we use from the BDZ algorithm only the mapping and assigning steps which lead to obtain a perfect hash function and not a minimal one.)

4.2.1 Key idea

Let be a graph $G = (\mathcal{V}, E)$, where

- the vertices $\mathcal{V}$ are the choosen interval of values.
- the edges E are randomly built (by hashing) from the set of keys $\mathcal{S}$

Finding a perfect hash function exactly means **finding a one-to-one function** mapping an edge $e = (v_0, v_1)$ to a vertex $phf(e)$. A practical way to do that is an **assignation**, that is selecting, for each edge e, a value $phf(e) = v_j$ with either $j = 0$ or $j = 1$.

If the graph is **acyclic**, this is fairly simple to find such assignation: one picks one edge containing a vertex of degree 1 and assign this vertex, removing the edge, and one iterates. Moreover, it is easy to remember this assignation, by storing only one bit of information by vertex : $j = (g(v_1) + g(v_2)) \texttt{ mod } 2$, where the g maps any vertex to 0 or 1.

The problem with regular graphs is that random acyclic graphs are not so common – in fact, it requires to have $|\mathcal{V}| > 2E$ in order that a random graph be acyclic with a probability of almost 1 [Majewski et al., 1996].

A better solution is to use this idea with **hypergraphs**. The major property of such graphs is that random hypergraphs are very common, at a cost of a few vertices: 3-hypergraphs with $|\mathcal{V}| > 1.23E$ will be enough to have an "acyclic" property with a probability of almost 1 [Majewski et al., 1996].

4.2.2 g-Assignation in a hypergraph

We begin with some definitions, following [Majewski et al., 1996] and [Botelho, 2008].

Definition 1 *Let r be an integer greater than 2, a* ***hypergraph*** *$G = (V, E)$, denoted by $r-$graph, is the generalization of a standard undirected graph where each edge connects between 2 and r vertices : $e = \{v_0, v_1 ... v_d\}$ with $2 \leq d \leq r$.*

Definition 2 *A $r-$graph is* **acyclic** *if and only if some sequence of repeated deletions of edges containing at least one vertex of degree 1 yields a graph with no edges.*

Definition 3 *A* **k-partite** *$r-$graph is a $r-$graph whose vertices can be partitioned into k disjoint sets so that no two vertices within the same set are linked by an edge.*

The perfect hash function building of BDZ_{PH} can be formalized as follows.

Given an undirected r-partite $r-$graph[43] $G = (V, E), |E| = n, |V| = m$, find an assignment g to all vertices in V such that for each edge $e = \{v_0, v_1, \ldots, v_{r-1}\} \in E$, the function

$$phf(e) = v_j$$

[43] In theory, the algorithm of building perfect hash function of BDZ_{PH} can apply for any $r-$graph. However, in practice, [Botelho, 2008] uses the Jenkins hash functions which creates in parallel three values in different ranges (4.2.4). Thus, in BDZ_{PH}, the $r-$graph is always r-partite

where

$$j = \left(\sum_{i=0}^{r-1} g[v_i] \right) \texttt{ mod } r \tag{4.1}$$

is a one-to-one function.

The function $phf : \mathcal{S} \rightarrow \mathcal{V}$ is a **perfect hash function** which maps each key in $\mathcal{S}$ to a unique value in the interval $\mathcal{V} = [0, |\mathcal{V}| - 1]$.

With the assumption that the hypergraph G is acyclic, [Czech et al., 1992] proposed an algorithm, denoted here by G-ASSIGNING, to solve the computation of the assignment g with time complexity $\mathcal{O}(|\mathcal{V}| + r \times |E|)$. We thus have a method to build a perfect hash function from a $r-$graph. The task is now to design an algorithm to build an *acyclic* $r-$graph.

4.2.3 Randomly building acyclic $r-$graph

[Havas et al., 1994] and [Majewski et al., 1996] conducted intensive studies on the probability of having a random acyclic $r-$graph with the following important conclusion:

> "For a random $r-$graph $G = (V, E)$, starting a constant ratio $c = \frac{|V|}{|E|}$, the probability that G is acyclic is greater than zero." When $|E|$ closes to infinity, this probability closes to 1.

A **random** $r-$graph $G = (V, E), |E| = n, |V| = m$ is here a graph that the selections of r vertices $v_0, v_1, ..., v_{r-1} \in V$ for each edge $e \in E$ are random.

The minimum value of c depends on r and is calculated in [Majewski et al., 1996, Chapter 4]. The best value known is for $r = 3$, with $c = 1.23$. It means that, with a suitable ratio between the number of edges and the number of vertices, after a finite number of attempts (outputing a random $r-$graph), we can obtain an acyclic $r-$graph which then can be applied the G-ASSIGNING algorithm to find the table g and then the perfect hash function phf.

In our own experiments, as soon as $|V| \geq 1.23|E|$, we never had to do more than 1000 attempts.

To implement this method, it remains to solve the two following practical problems:

1. **How to construct a random** $r-$graph**?** Ideally, the function to select the vertices for each edge of the $r-$graph needs to be absolutely random. In practice, a family of universal hash functions [Carter and Wegman, 1977] can be used with an acceptable limit in randomness. [Czech et al., 1992] used a family of universal hash functions, denoted here by $\mathcal{H} = \{h_0, h_1, ..., h_{\text{r-1}}\}$, which need $\mathcal{O}(log(m))$ space to store and that can be evaluated in constant time. The procedure GRAPH-CREATING($\mathcal{S}$,$\mathcal{H}$) outputs a random $r-$graph $G(V, E)$ for the set $\mathcal{S}$ of keys as follows:

   $V = \emptyset$, $E = \emptyset$;
   for *each* $k \in S$ **do**
   $\quad e = \{h_0(k), h_1(k), ..., h_{r-1}(k)\}$;
   $\quad E = E \cup e$;
   $\quad V = V \cup \{h_0(k), h_1(k), ..., h_{r-1}(k)\}$;
   end

2. **How to check if an $r-$graph is acyclic?** Based on the definition-2, [Havas et al., 1994] proposed an algorithm with a time complexity of $\mathcal{O}(rn+m)$, denoted here by ACYCLIC-TESTING(G), to verify if an $r-$graph G is acyclic (details of the algorithm are not shown).

4.2.4 Jenkins hash functions

The algorithm works with any "universal" hash functions [Czech et al., 1992]. BDZ_{PH} uses the universal hash function family developed by Jenkins [Jenkins, 1997]. We call the functions in this family by the **Jenkins hash functions** and we denote them by $\mathcal{H}_{\mathcal{J}}$ in the following. As mentioned in [Botelho, 2008], although the Jenkins hash functions have not been proved theoretically they have very good performance in practice. Morever, there are two key advantages of using $\mathcal{H}_{\mathcal{J}}$ that support the implementation:

- One can generate in parallel three random 32-bit integers (or 64-bit integer, depending on the size of the machine word) in constant time. This feature well support the case of random 3-graph. Thus, the random $r-$graph, created by BDZ_{PH}, is always r-partite.
- The functions in $\mathcal{H}_{\mathcal{J}}$ are represented only by the *seed* (denoted here by J_{seed}). It is the 32-bit (or 64-bit) integer, which is used as the initialization for the mixture of bits to create the random values.

Starting from a set of keys $\mathcal{S}$ and a seed J_{seed}, we denote by JENKINS-GRAPH-CREATING($\mathcal{S}$,J_{seed}) the function that creates the random $r-$graph G by using $\mathcal{H}_{\mathcal{J}}$. As $|\mathcal{S}|$ is fixed, the time complexity of JENKINS-GRAPH-CREATING($\mathcal{S}$,J_{seed}) is $\mathcal{O}(1)$.

4.2.5 Complete $\text{CHM}_{\text{PH}}/\text{BDZ}_{\text{PH}}$ algorithm

We now have all the components needed to implement the algorithm. It leads to the BDZ_{PH} algorithm to generate perfect hash functions:

Data: $\mathcal{S}$
Result: Table g and seed J_{seed}
repeat
  J_{seed} = random seed
  G = JENKINS-GRAPH-CREATING($\mathcal{S}$,J_{seed})
until ACYCLIC-TESTING(G);
g = G-ASSIGNING(G);

The time complexity of the entire algorithm is thus $\mathcal{O}(1)+\mathcal{O}(rn+m)$. Once the perfect hash function have been created, each key in the data set $\mathcal{S}$ can be saved in the list at the position *addr*. The query of the existence of a key in this list can be done in $\mathcal{O}(1)$ time.

4.2.6 An example of the BDZ_{PH} algorithm with $r = 3$

We now give an example of using BDZ_{PH} to index a set of keys S and to query against the indexed set L. We re-use the input key set S from the example of reduced list (Figure 2.6). We chose $r = 3$ and the seed from the Jenkins family hash functions $J_{\text{seed}} = 13$.

Computation of *phf*. We first start by creating the perfect hash function for S. We thus compute the 3−graph made of 6 edges (the set of keys contains 6 keys) and 9 vertices. The list of edges is given in Figure 4.1.

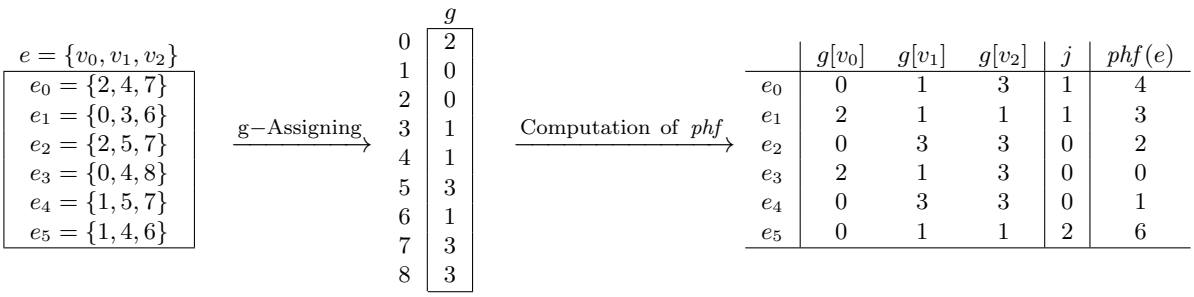

Figure 4.1: Example of the computation of *phf* for 6 keys.

We do not give the details of how the graph is computed. Once the graph is built, we are able to compute the g table (details not given). Note that in g, there are three elements whose values are 3 (at position 5, 7, 8). As 3 is the initialization value of all the elements in g, it means that these elements are not changed when g is computed.

The function *phf* can now be computed. For example, for edge $e_0 = \{2, 4, 7\}$ we compute j following equation 4.1, that is $j = (g[2] + g[4] + g[7])$ `mod` $3 = (0 + 1 + 3)$ `mod` $3 = 1$. Thus $phf(e_0) = v_1 = 4$. Computation for the other edges are given figure 4.1. We verify that we obtained a different value for each edge.

Using *phf* to index the set of keys. Thanks to *phf* each number between 0 and the size of the set of keys S is assigned a unique value as seen before. Indexing consists in assigning the *phf* value to each key. Because the number of vertices is greater than the number of keys, we have to pad the empty slots with a key present in the set S (the first one for example). This is depicted figure 4.2.

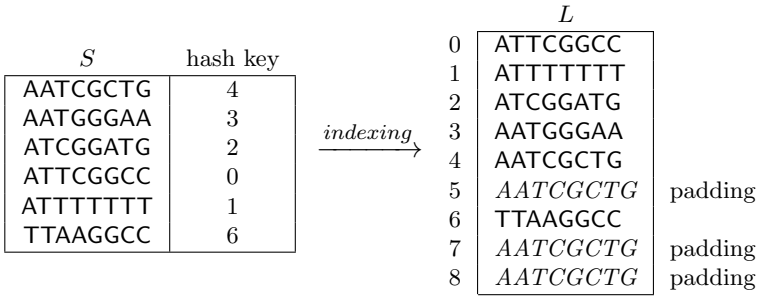

Figure 4.2: Example of indexing using perfect hashing

Querying using an index built with perfect hashing. We present 2 examples of querying against the index. For the query word ATTTTTTT, we compute the corresponding edge which is $e = \{1, 5, 7\}$ and then $j = (g[1] + g[5] + g[7])$ `mod` $3 = 0$ and $phf(e) = v_0 = 1$. The corresponding hash key is thus 1, and $L(1) =$ ATTTTTTT which is equal to the query word: There is a match.

For the query word AATCGCTC, we compute the corresponding edge which is $e = \{1, 3, 7\}$

and then $j = (g[1] + g[3] + g[7]) \bmod 3 = 1$ and $phf(e) = v_1 = 3$. The corresponding hash key is thus 3 and $L(3) = \mathsf{ATTCGGCC}$ which is different from the query word: There is no match.

4.3 Using perfect hashing functions for approximate neighborhood matching

As we mentionned in chapter-2, we will use another data structure in which the lists are indexed themselves. Thanks to this additionnal information we will speedup the matching time. The way of indexing the lists will use perfect hashing.

4.3.1 Using $\mathrm{BDZ_{PH}}$ to create indexed block structure

The Indexed Block Structure $\mathsf{SeedBlock}(s)$ contains all the lists and blocks as in the Reduced Structure together with further informations (see figure-4.3 and figure-4.4):

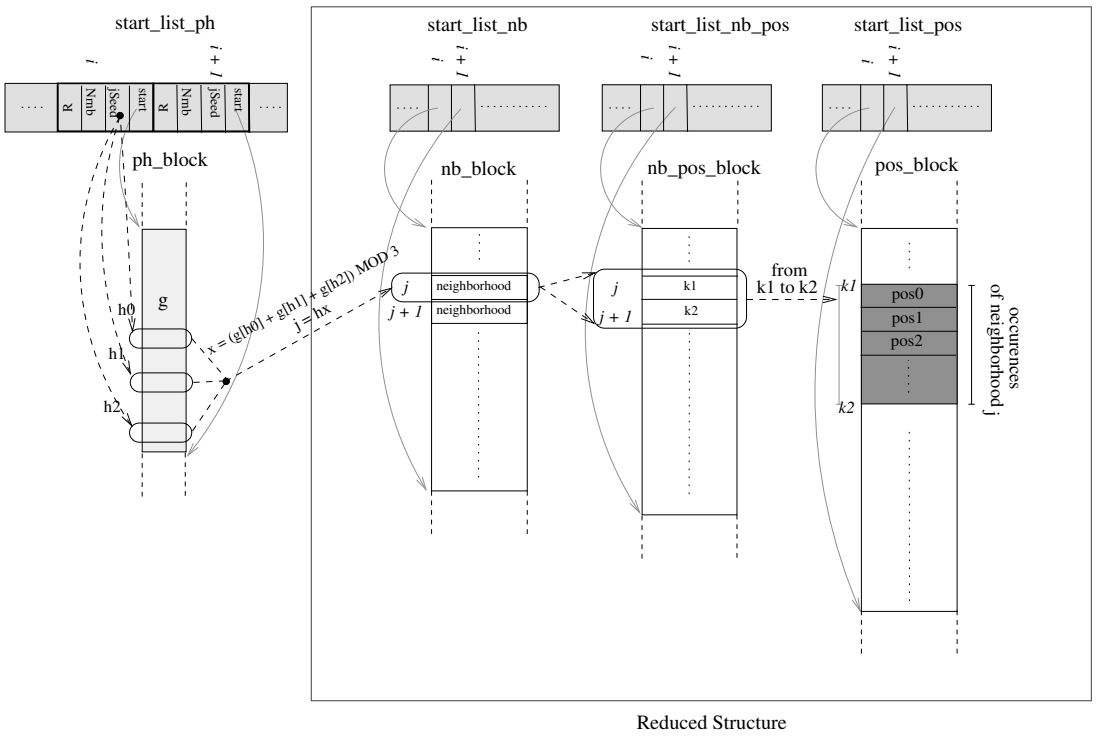

Figure 4.3: Indexed Block Structure together with the $\mathrm{BDZ_{PH}}$ algorithm on the SeedBlock. Compared to the reduced structure (figure-2.7, right on this figure), the ph_block (which contains the table g) and the start_list_ph (which contains the start position of the table g and 3 data: JSeed, N_{rnb} and R) are added. More details about JSeed, N_{rnb} and R can be found in page 76.

- The list of neighborhoods, as in the Reduced Structure. These neighborhoods are now sorted according to their phf values. As the hash function is not minimal, the size of the hash table M is:

$$M = \lceil (1.23 \times N)/3) \rceil \times 3 \ (element)$$

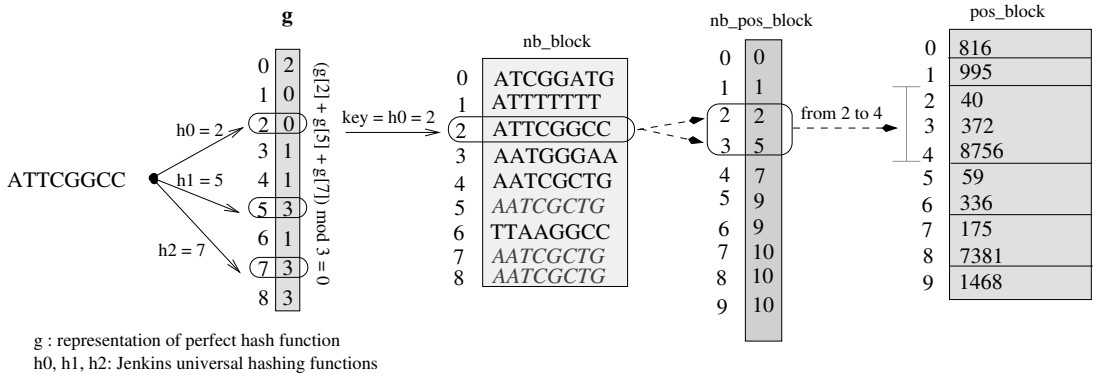

Figure 4.4: Example of indexed block structure by BDZ_{PH} based on the same example than before (figure-2.6 and figure-4.2.6). For a query $Q(sq)$, its index is calculated and then the matching is verified by comparing q (here ATTCGGCC) with the neighborhood kept in the address associating to this index. The occurrences and their positions of the query are retrieved in the same process with the reduced structure.

It leads to ($N_{\text{pad}} = M - N$) phantom values that are not mapped by any key. As a pattern that is not really in the SeedBlock can be hashed to any address in the table, including to these phantom values, we added N_{pad} elements into the SeedBlock with the value of the first existing neighborhood. It does not yield false positive matching results because of the difference between the value of the padding element and the query itself that can be hashed to this address.

- The family of the universal hash functions $\{h_0, h_1, h_2\}$. With the Jenkins hash functions, all this family can be stored with only the seed (`jenkins_seed`, one unsigned int value, which can be kept in 1 word).

- The assignment table g, of size $|g| = \lceil M/4) \rceil$ $(byte)$, as we can use 2 bits to present a element in g.

- Two supplement data are added to the data structure:

  - The number of actual elements (excluding the phantom values): N_{rnb}
  - The ratio between N_{rnb} and the total elements (including the phantom values): R. In theory, it is about 1.23×. But in practice, it must be rounded up. This value is kept in order to avoid additional computational operation in the query phase.

The total size is thus

$$(M + 3) \times S_{unit} + \lceil M/4) \rceil \ (byte)$$

that is an overhead of

$$(3 + N_{pad}) \times S_{unit} + \lceil M/4) \rceil \ (byte)$$

compared to the data structures used in the previous chapter.

Finally, as the list of neighborhoods is in the same format than in the previous chapter, it could be used as well with mflBPR, especially in cases where a large number of errors are allowed.

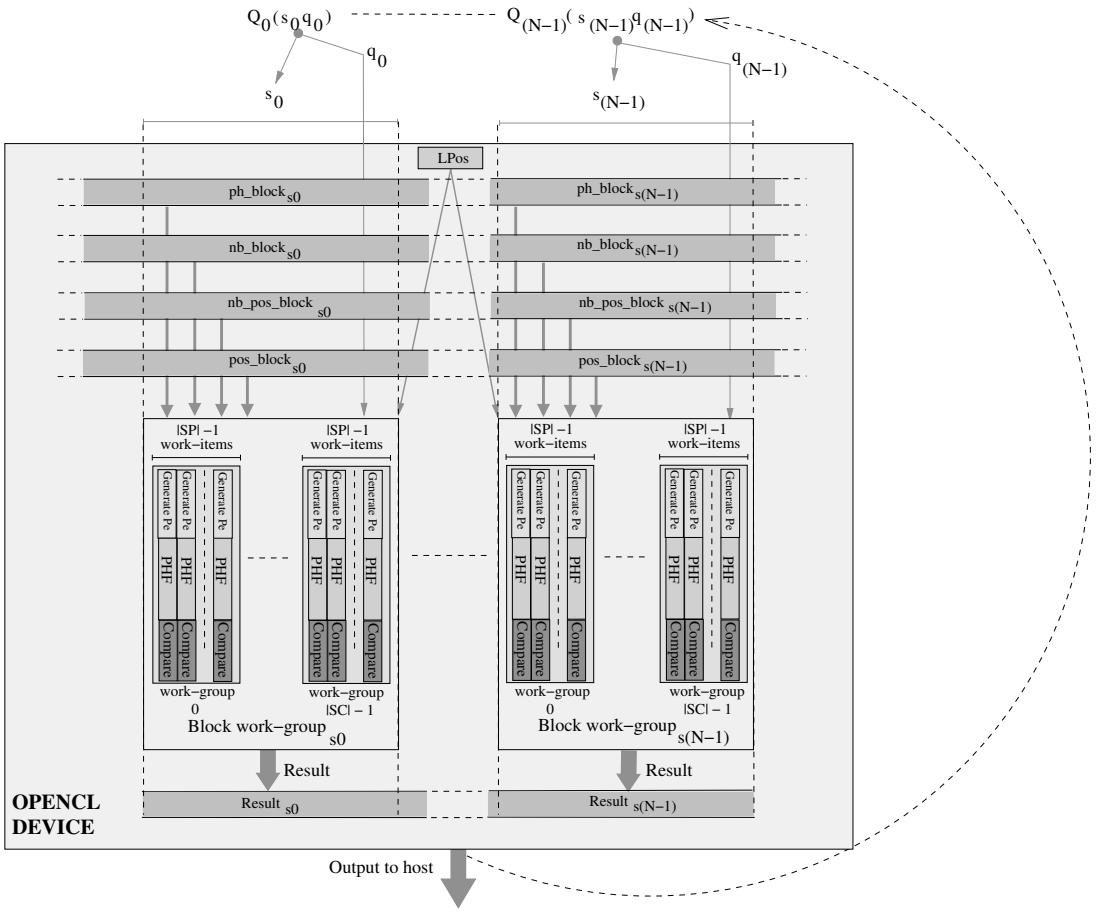

Figure 4.5: Using BDZ_{PH} and Indexed Block Structure on OpenCL devices. The work-item must calculate the address of the neighborhood which then is compared with the corresponding P_e.

4.3.2 Implementing $\mathrm{BDZ_{PH}}$ query on OpenCL devices

Figure-4.5 depicts the usage of perfect hashing with the Hamming-distance patterns generating model on the OpenCL devices. All the index data are precomputed and transferred only once to the device. The input queries, each in the form of sq, are divided into batches of size N. The constants related to the configuration of the kernel such as: $|SP|$, $|SC|$, L_{POS}, etc. are calculated in the host and written to the constant memory region of the device. Then, the application runs looping on each batch of queries:

- Each work-item follows these steps:
  - It uses the key of s to get the addresses of the corresponding SeedBlock.
  - It generates the degenerated pattern from q based thanks to its local identifier and group identifier.
  - It uses the Jenkins hash functions h_0,h_1,h_2 and the table g to calculate the address of the degenrated pattern.
  - It compares the degenerated pattern with the element in the calculated address.
- The positions of the matching neighborhoods are then written back to a result array in the global memory in order to be transfered back to the host.

	Time complexity	`uint` operations	Global memory access
Generate P_e	$\mathcal{O}(v^e)$	v^e	1
PHF	$\mathcal{O}(1)$	54	3
Compare	$\mathcal{O}(1)$	1	1

Table 4.1: The time complexity, the number of `uint`operators and the memory access number of a $\mathrm{BDZ_{PH}}$ query.

As shown in Table 4.1, although the work-items need to do one more step (comparison with the degenrated pattern) compared to binary seach, this approach is more suitable for the GPU because the number of computing operations and memory accesses is fixed, and there is no branch divergence during the execution of a work-items.

4.4 Conclusion

This chapter presented an approach different from that in Chapter-3 which consists in index the set of neighborhoods of the seeds. For this we use the very efficient technique of perfect hashing which allows us to query a neighborhood q in the SeedBlock in constant time. The performance of this approach will be analysed and compared with the other approaches in Chapter-5.

Chapter 5

Performance Results

Chapter-3 and chapter-4 described several solutions for approximate neighborhood matching: bit-parallel (mflBPR), binary search (BS) and perfect hashing (PH). This chapter presents the performance measurements when experimenting these solutions on a GPU and on a multicore CPU. Firstly, we will explain the methodology. Then, we will present measurements and discuss about the gain of parallel computing. The results are analyzed to show the impact of several parameters of the method on the performances, and a comparison between the three solutions is presented.

Contents

5.1 Benchmarking environments and methodology

This section introduces the setup for the experiments of each solution (table-5.1) and the units used to evaluate the performance results.

5.1.1 Benchmarking environments

We benchmarked our solutions on two types of OpenCL devices: a GPU and a multicore CPU. In all cases, the same OpenCL code was used, but with different OpenCL libraries on the two platforms, leading to two environments:

oclGPU: GPU, NVIDIA GTX 480 (30×16 cores, 1.4 GHz, 1.5 GB RAM), with the OpenCL library NVIDIA GPU Computing SDK 1.1 beta.

oclCPU: CPU, Intel Xeon E5520 (8 cores, 2.27 GHz, 8 MB cache), with the OpenCL library AMD APP SDK 2.4.

All programs were compiled using `GNU g++` with the `-O3` option. The host computer had 8 GB RAM.

Moreover, we also tested a pure C++ "CPU serial" version of mflBPR, leading to a third environment:

serialCPU: CPU, 1 core of an Intel Xeon E5520 (8 cores, 2.27 GHz, 8 MB cache)

It should be noted that:

- As described in chapter-1, our work focuses more on GPUs than on multicore CPUs. Nevertheless, we made benchmarking on multicore CPUs by taking advantage of the portability of OpenCL on different types of processors. But all the analyses, designs and implementations of our work are specialized and customized for GPUs, especially for NVIDIA Fermi GPUs.
- While there are actually 4 physical cores in an Intel Xeon E5520, upto 8 parallel threads can run simultaneously on this CPU (thus 8 "logical cores") thanks to the Intel 's Hyper-Threading Technology (see section 1.2.1).
- Although these implementations can run on an ATI Radeon HD5780, the performances were very low in our implementation (see page 118). We thus only report here the benchmarks on the NVIDIA 480 GTX and on the Intel Xeon E5520.

5.1.2 Experiments setup

Table 5.1 summarizes the experiments setup described below.

Genetics data and indexes. Indexes were computing using the first 100 Mbp of the human chromosome 1. Remember that the indexes can be laid out with a full list of neighborhoods (general structure) or with an additional table to regroup identical neighborhoods (reduced structure, see page 50). The following implementations were benchmarked:

- The mflBPR solution using the general structure;
- The binary search solution using the reduced structure;
- The perfect hashing solution using the indexed block structure.

	Bit-parallel (mflBPR)	Binary Search (BS)	Perfect Hashing (PH)
Text Length	First 100 Mbp of the human chromosome 1		
Index Structure	general	reduced	indexed block
Number of Patterns	100	1000	1000
Seed Length	3, 4, 6	4	4
Neighborhood Length	4, 8, 16		
Index Size (MB)	500 – 800	400.5 – 1094.14	400.7 – 1360.53
Number of errors (e)	0 – 3 (Levenshtein)	0 – 2 (Hamming)	0 – 2 (Hamming)
Benchmarking environment	oclGPU, oclCPU, serialCPU	oclGPU, oclCPU	oclGPU, oclCPU

Table 5.1: Experiments on 3 solutions: mflBPR, binary search (BS), perfect hashing (PH) on the first 100 Mbp of the human chromosome 1. More details about the index sizes can be found in Table A.1 and Table A.2.

Even if the mflBPR solution can work on reduced structure, we chose the general structure as it is the most simple case to implement.

The binary search solution can also work on general structures[44]. In this case, the size of SeedBlock may be larger due to the redundancy. Morover, it requires also more calcultations to get all the occurrences and their positions as the binary match return only one occurrence.

In order to analyse the speedup of the mflBPR solution over BPR, we created another version of general structure where several neighborhoods are not packed into a unique word[45].

Number of patterns. With the mflBPR solution, we ran searches on 100 successive queries, but we saw no significant difference between 1, 10, 100 or 1000 queries, as soon as enough computations hide the transfer times. With the binary search and perfect hashing solutions, we ran searches with 1000 successive queries.

Error Numbers. With the mflBPR solution, the error number e is from 0 (exact matching) to 3. With the binary search and perfect hashing solutions, the error number is from 0 to 2 (table-5.1) as the number of the degenerated patterns exceeds the maximal number of work-items in each work-group from $e \geq 3$ (see page 62).

5.1.3 Performance units

Device time. Generally, the running times of the experiments are used to evaluate the speed of the solutions. As described pages 60, 63 and 79, the index is loaded only one time to the global memory of the OpenCL devices. The time measured, the **device time**, denoted here by

[44] To apply the binary search, the SeedBlock in general structure must be sorted, but it also allows for the redundancy

[45] To benchmark BPR, we used the mflBPR code, setting the *neighborhood per word (h)* parameter to 1. An optimized BPR-only implementation could be slightly more efficient.

T_{dev}, is the sum of 3 phases[46]:

1. Transfer times from host to device for the input queries;
2. Compute time of the OpenCL kernel;
3. Transfer times from device to host for the output result.

It should be noted that the benchmark of the binary search solution and the perfect hashing solution presented in this chapter do not include the running time of the position retrieval stage (as described in page 64 and page 79). More precisely, the T_{dev} of these 2 solutions are only the running time of the approximate matching stage, which may make their performance (measured here) to appear higher than that of these actual solutions. It can slightly impact the performance comparision of these 2 solutions with the mflBPR solution (see 5.2.5). However, it does not cause any problem for what are discussed in 5.3. In addition, the full implementations of the binary search solution and the perfect hashing solution are confirmed by the executions of the $\texttt{Seed} - \texttt{Filter}$ kernel of the real application of short read mapper (MAROSE) (see chapter 6).

Neighborhood matched per second. Once an input pattern sq has been processed against the SeedBlock of a seed s, which contains $\mathcal{N}_s$ neighborhoods, we will say that *"$\mathcal{N}$ neighborhoods have been processed"*. Thus we will measure the performance of our algorithms in terms of *neighborhoods processed per second*, denoted here by nps, representing the number of neighborhoods (n) per second: n/s.

For mflBPR, nps really reflects the number of neighborhoods which are loaded and compared to the neighborhood part q in the pattern. But for binary search and perfect hashing, it is not exactly this number that is measured because the algorithms do not compare all the neighborhoods in the SeedBlock with q. However, it remains a good measure to describe the quantity of index information processed.

Finally, as the binary search solution is experimented on reduced structures and the perfect hashing solution is on indexed block structures, the number of neighborhoods is not the same as in general structures. It can lead to an over-estimation of the true computing performance of these solutions.

In order to compare the performance between all our solutions, we thus used the size of the SeedBlock in the corresponding general structure, before being reduced to calculate nps. For an experiment which takes the set of pattern P as an input:

$$nps = \frac{\sum_{s \in P} \mathcal{N}_s}{T_{dev}} \ (n/s)$$

where $\mathcal{N}$ is the neighborhood number of the SeedBlock in the general structure.

5.2 Performance measurements

In this section, the experiments results of mflBPR, binary search (BS) and perfect hashing (PH), are presented. For mflBPR, we give both running times and neighborhood per second.

[46]These times are measured by profiling the return events of the corresponding OpenCL commands: clEnqueueWriteBuffer, clEnqueueNDRangeKernel and clEnqueueReadBuffer.

For BS and PH, we show only the nps, as the running time is directly related to this measure (explanations in page-84). The purpose of this section is to explain the performance of these solutions in different cases.

5.2.1 Performances of bit-parallel solutions (BPR/mflBPR)

Speedup of mflBPR over BPR using the serialCPU environment To analyse the speedup of mflBPR over BPR, we use the serialCPU environment to run mflBPR on two types of general index: one with multiple neighborhoods packed in machine words (mflBPR) and the standard approach with only one neighborhood per machine word (BPR). Figure-5.1 depicts the experiments result of the case that the seed length = 4 and the neighborhood lengths = 4, 8 and 16.

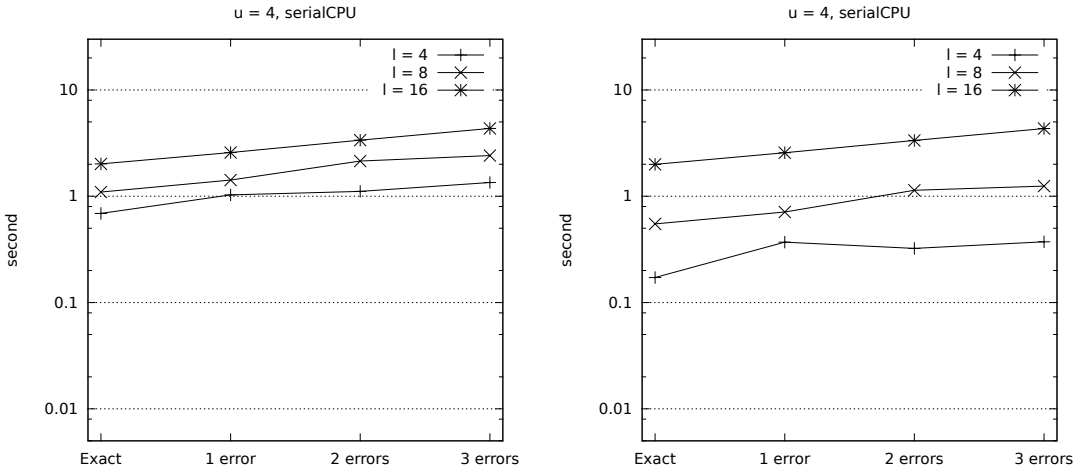

Figure 5.1: Running time of BPR (left) or mflBPR (right), both on CPU serial version, $u = 4$, $\ell = 4, 8, 16$.

With 32-bit integers and 2 bits to encode each character, the numbers of neighborhoods kept in each word is respectively 4, 2 and 1 with neighborhoods lengths 4, 8 and 16. Thus the theoretical gains are 4, 2 and 1, respectively. As depicted on the figures, running the serial version on CPU for BPR ($\ell = 16$) and mflBPR leads to performance gains for mflBPR compared to BPR ranging from 2.73× to 3.92× for words of length 4, and from 1.89× to 2.06× for words of length 8, which are very close to the 4× and 2× theoretical gains.

Performance of mflBPR. We investigate the performance of mflBPR. As said before, we also have an implementation of mflBPR for the serialCPU environment. We thus compare mflBPR on the three environments. Results are depicted figure-5.2 (running time) and figure-5.3 (nps).

In the most simple instance (neighborhoods of size 8, no error), the serial CPU implementation peaks at 59 Mn/s, the oclCPU at 189 Mn/s, and the oclGPU at 3693 Mn/s. In this case, using OpenCL brings speed-ups of about 3.2× on CPU and about 62× on GPU (thus the speedup about 19× between oclGPU and oclCPU).

In the same setup, the offset indexing peaks at 4.0 Mn/s on serial CPU, 108 Mn/s on OpenCL CPU and 1706 Mn/s on OpenCL GPU (data not shown).

When the number of errors rises, performance degrades in both implementations. On small neighborhoods, starting from $e = 2$, the performance of both GPU and CPU versions are limited

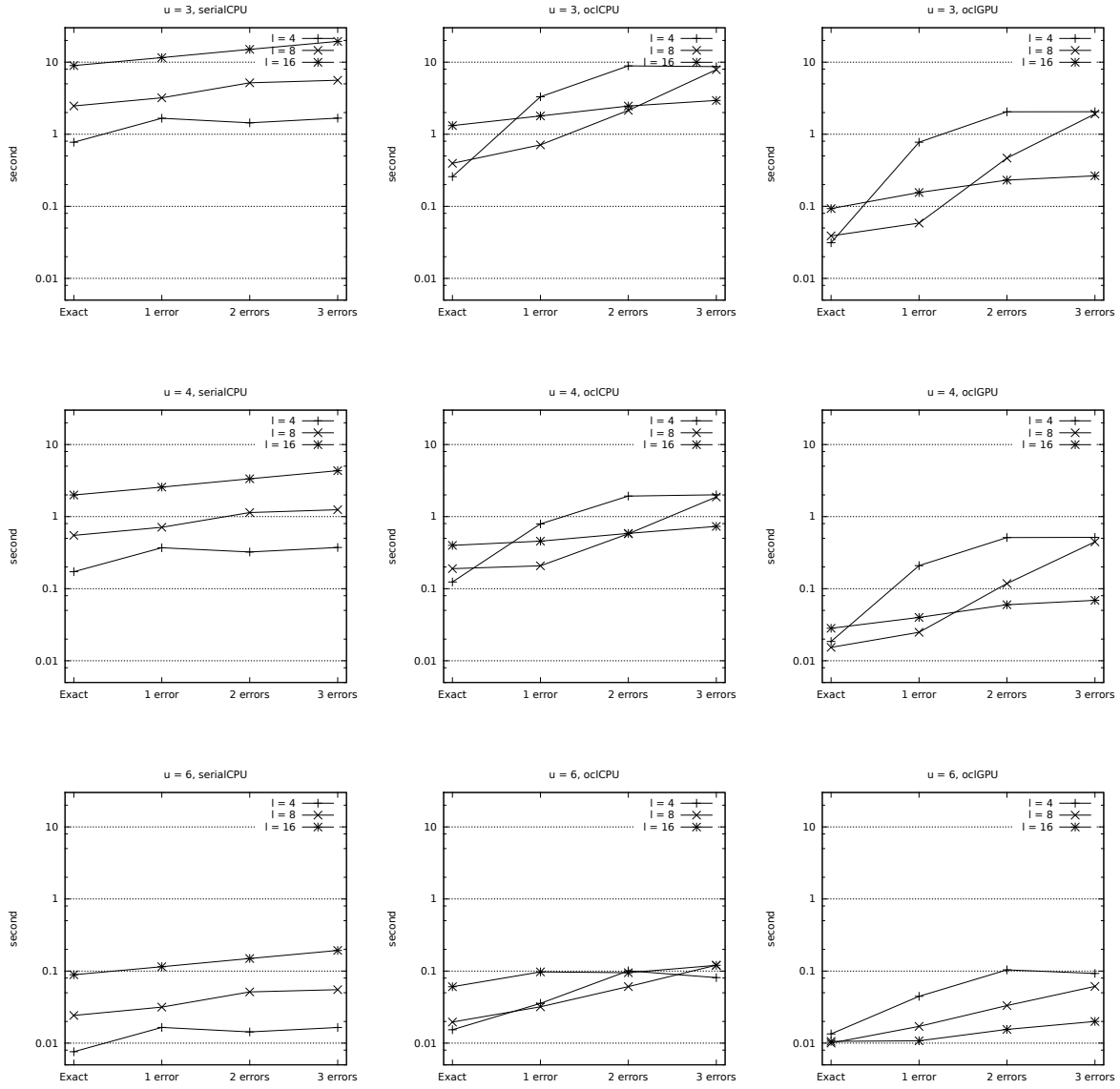

Figure 5.2: Running time of mflBPR with the neighborhood length 4, 8 and 16. Seed length is 3, 4 and 6 (from top to bottom). Three benchmarking environments were used: serialCPU, oclCPU and oclGPU (from left to rigth). For each environment, there are three different curves, corresponding to different neighborhood lengths (4, 8 and 16).

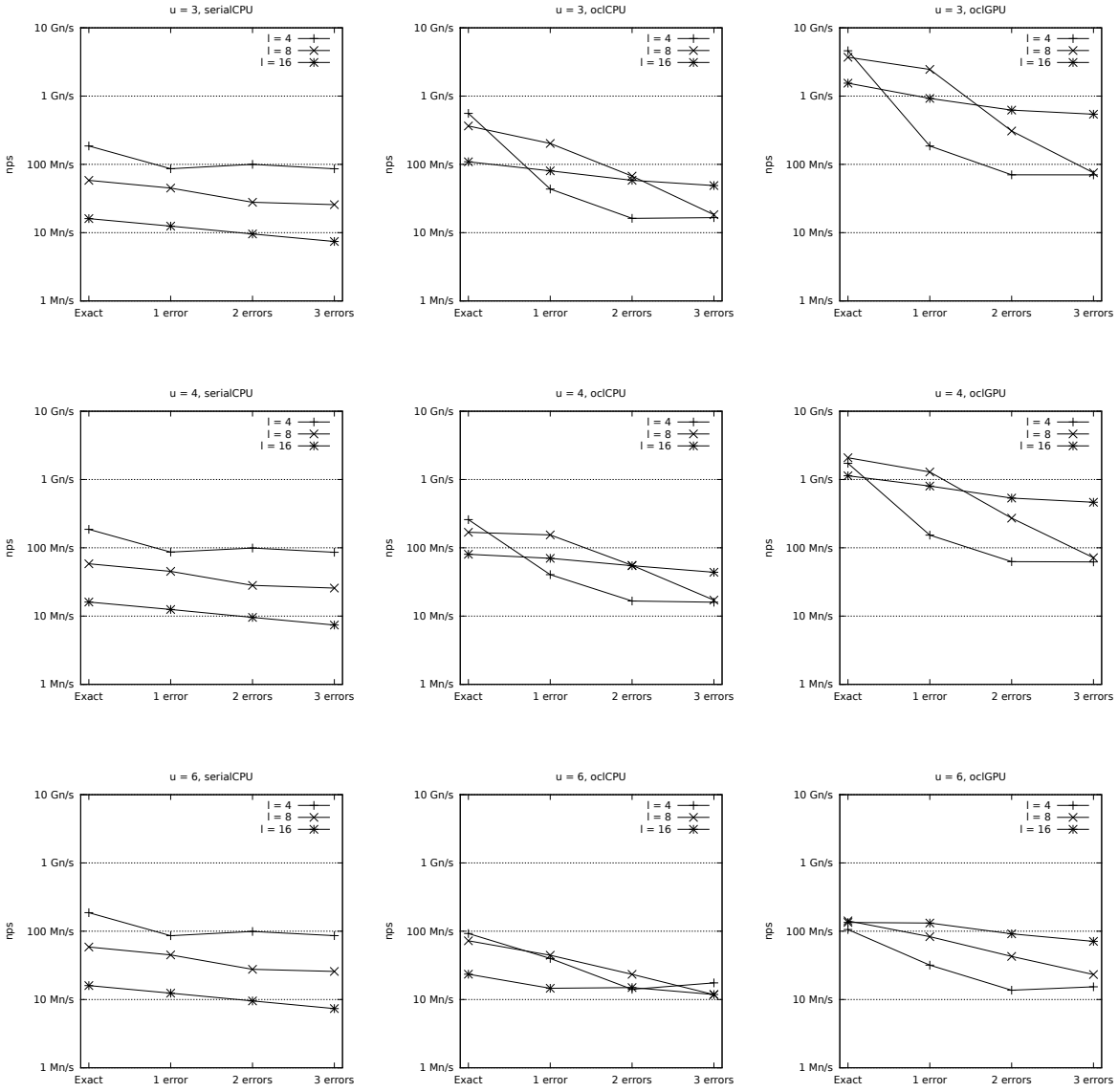

Figure 5.3: Performance of mflBPR in *neighborhoods per second* with neighborhood length 4, 8 and 16. Seed length is 3, 4 and 6 (from top to bottom). Three benchmarking environments are used: serialCPU, oclCPU and oclGPU (from left to right). For each environment, there are three different curves, corresponding to different neighborhood lengths (4, 8 and 16).

by the number of matches in the output. In these cases, almost all queries are matched, causing a bottleneck when each work-item writes its results to the global memory using an atomic function[47].

However, even in the worst case (7.5 Mn/s, CPU serial implementation, 3 errors, seed length 4 and neighborhood length 16), using the neighborhood indexing takes less than 0.06 s for parsing a chromosome of length 100 Mbp, while non-indexed approaches using bit parallelism takes 0.9 s with `agrep` and 0.7 s with `nrgrep`.

5.2.2 Performance of binary search (BS)

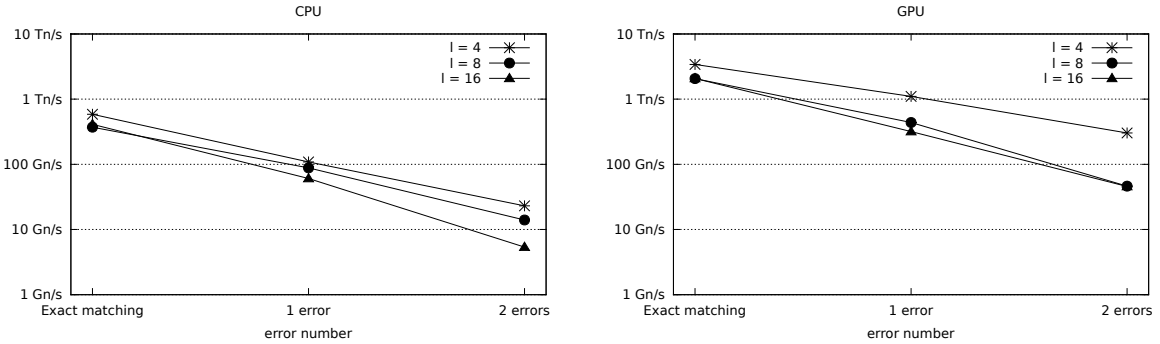

Figure 5.4: Performance of binary search on the neighborhood indexing on CPU (left) and GPU (right), with a seed length of 4 and a neighborhood length of 4, 8 and 16. Indexes created from 100 MB. The number of queries is 1000.

Figure 5.4 shows performance of the binary search on the neighborhood length of 4, 8 and 16 with seed length 4. Those results show that the performance increases when the length of the neighborhood decreases: $\ell = 4$ is the best, then $\ell = 8$ and $\ell = 16$ (explanations can be found in 5.3.2)

Figure-5.5 shows that the GPU is in the worst case 3.3× faster than the multicore CPU ($\ell = 16$, 1 error) and in the best case 13.1× faster ($\ell = 16$, exact matching). The speedup increases with the number of the errors with $\ell = 4$ and 16, but decreases with $\ell = 8$. We have not found the reason for this problem, which could require further experiments on multicore CPUs.

5.2.3 Performance of perfect hashing (PH)

Figure-5.6 shows the performance of the perfect hashing solution with neighborhood lengths 4, 8 and 16 considering a seed length of 4. Generally, the performance profile is very similar to the binary search one: the performance clearly depends on the length of the neighborhood (5.2.2), and again the best result is obtained with neighborhood length 4 and the worst for length 16.

Figure-5.7 shows the speedup of the GPU over the multicore CPU. The GPU is in the worst case 4.5× faster than the CPU ($\ell = 4$, exact matching) and in the best case 29× ($\ell = 4$, 2 errors) faster. This result demonstrates the advantage of the GPU over the CPU when running applications which require intensive calculations without divergent branches.

The same than the binary search solution, there is also the abnormality in the speedup when the number of errors increases, now with $\ell = 16$ (5.2.2). This should require more experiments

[47] The atomic function used here is `atomic_inc`

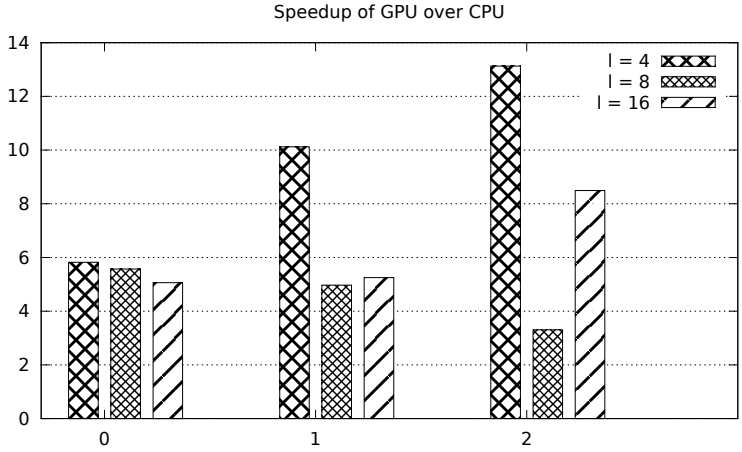

Figure 5.5: Speedup of GPU over CPU of querying by binary search on the neighborhood indexing (for the performances presented in figure 5.4).

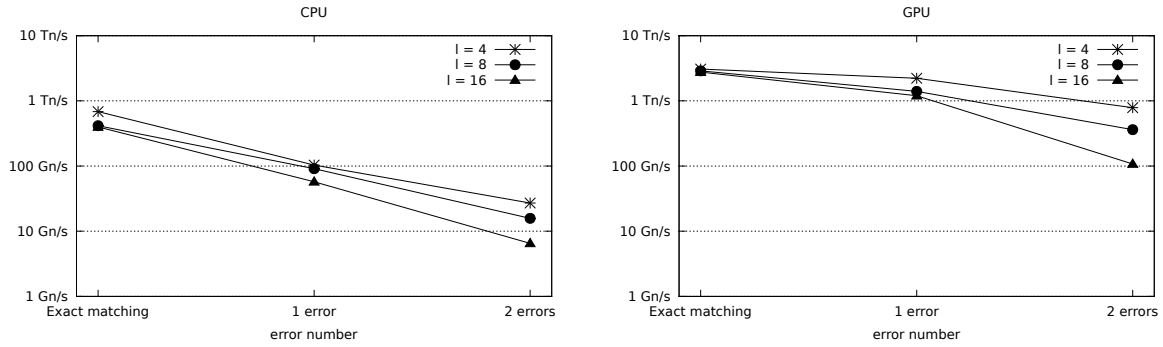

Figure 5.6: Performance of query by perfect hashing on the neighborhood indexing on CPU (left) and GPU (right), with a seed length of 4 and a neighborhood length of 4 , 8 and 16. Indexes created from 100 MB. The number of queries is 1000.

on multicore CPUs to find the reason.

5.2.4 Efficiency of parallelism on binary search and on perfect hashing

By comparing with the corresponding serial version, mflBPR showed striking speedups with the OpenCL version both on multicore CPU and GPU. For the binary search and perfect hashing approaches, we did not implement the serial versions. Nevertheless we are able to estimate the increase of running time which depends on $|\Pi_e(q)|$. As depicted in figure-5.8, the growth of the running time is always less than that of $|\Pi_e(q)|$. Thus we can conclude that OpenCL versions should improve serial version.

5.2.5 Performance comparisions between approaches

Figure-5.9 shows the performance comparisons between our three solutions on the oclGPU benchmarking environment. Seed length u was set to 4 and the neighborhood length ℓ varies from 4 to 16.

In all cases, using perfect hashing leads to the best solution. Binary search follows and mflBPR ends. Measured in nps, on the GPU, the perfect hashing solution can be up to 10×

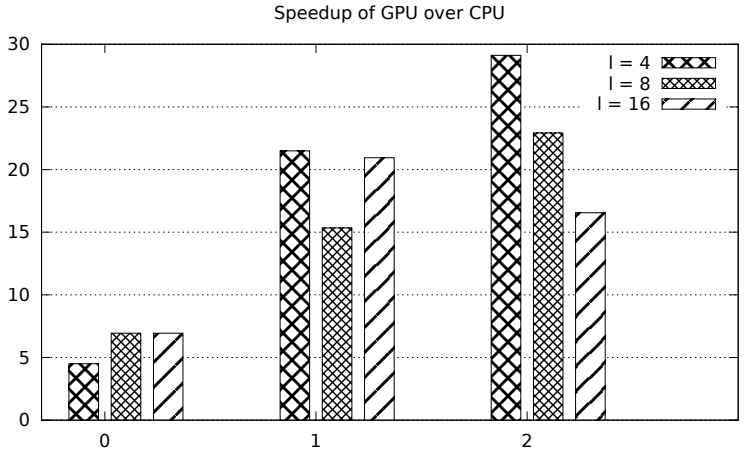

Figure 5.7: Speedup of GPU over CPU of querying by perfect hashing on the neighborhood indexing for the performances presented in figure-5.6.

better than binary search (explanations in 5.3.3). Compared to mflBPR, perfect hashing is up to 1000× faster.

In fact, mflBPR solution has to traverse all the neighborhoods in the list, while the perfect hashing solution needs only 3 accesses to the table g and 1 access to the list of neighborhoods. This explains the gain obtained with perfect hashing.

5.3 Discussion

We have presented raw results of performance for mflBPR, binary search and perfect hashing approaches. This section analyses the performance of these methods against their arguments:

- The length of the seed.
- The length of neighborhood.
- The number of errors.

We seek to compare global memory accesses and branch divergences of the three methods. Table 5.2 summarizes data that are discussed in the three next sections.

5.3.1 Impact of the seed length

The length of seed (u) is related to the number of seed occurrences. Thus the number of neighborhoods and the size of SeedBlock. For a text of length n, the average number of neighborhoods in a SeedBlock is:

$$\mathcal{N} \sim \frac{n}{4^u}$$

As the length of text used for all experiments is 100 MB, about 2^{27} MB, $\mathcal{N} \sim 2^{19}$.

The length of the seed does not have any influence on the time complexity of the perfect hashing method.

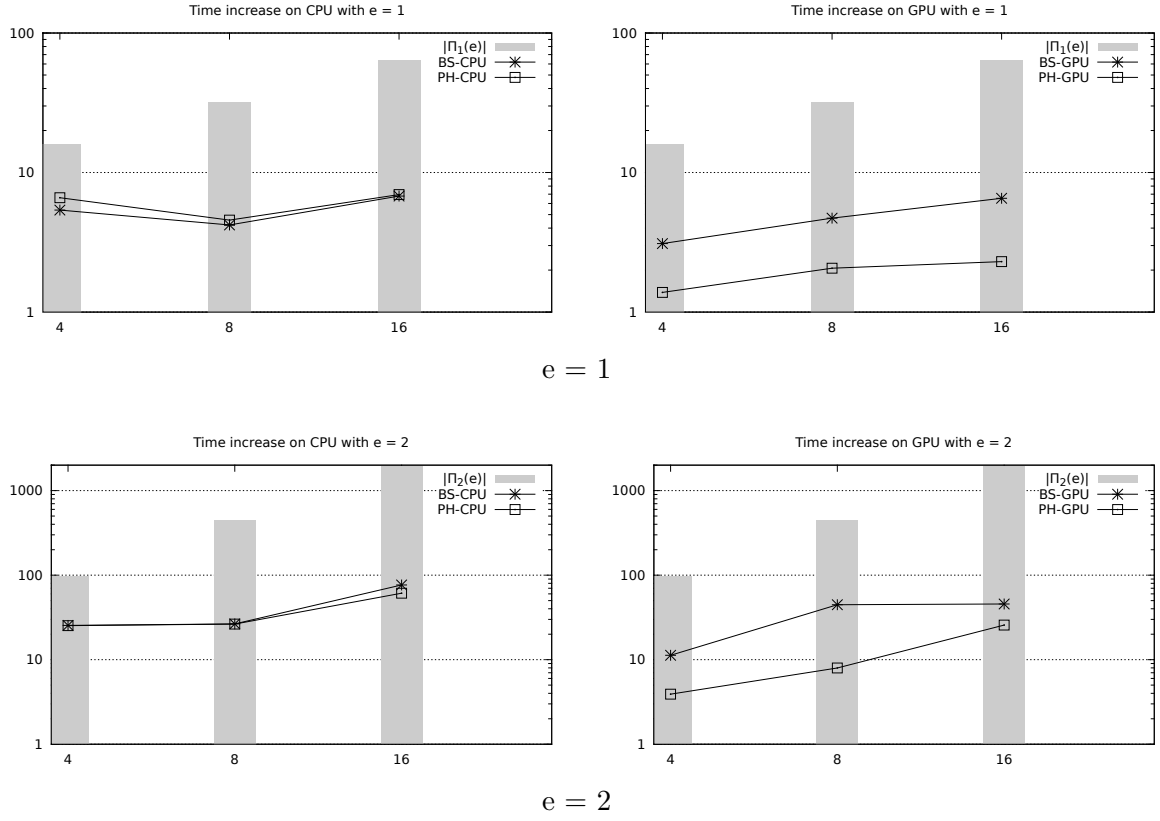

Figure 5.8: The running times of binary search (BS) and perfect hashing (PH) compared to the growth of the number of degenerate patterns $|\Pi_e(q)|$. For $e = 1$ (top) and $e = 2$ (bottom) on oclCPU environment (left) and oclGPU environment (right).

But, for the two other methods of direct neighborhood matching, $\mathcal{N}$ is an argument of the time complexity (table-5.2). Indeed, as a number of elements have to be traversed during the searching, the number of global memory accesses could be important.

mflBPR has to process all the neighborhoods in the SeedBlock, but it has the advantage of coalesed access pattern and the use of local memory as the temporary memory storage. As the index segment in the local memory is divided to process among the work-item in the same work-group, we can choose a number of neighborhoods to be processed with a work-group as a multiple of its work-item number. In this case, there is no branch divergence.

The binary search method uses randomly accesses to the global memory. As the binary search process can terminate at any iteration, there is a high probability of branch divergence. In the worst case, all the work-items in the same warp have to pass through $\log(\mathcal{N})$ iterations.

It means that even if the mflBPR has to process much more elements than the binary search, it still benefits from the advantages of contiguous memory accesses and the absence of branch divergences while accessing to the neighborhood data.

5.3.2 Impact of the neighborhood length

The length of neighborhood (ℓ) is one of the arguments in the time complexity of mflBPR (but it does not cause any divergence branch). In addition, it can also change the number of the

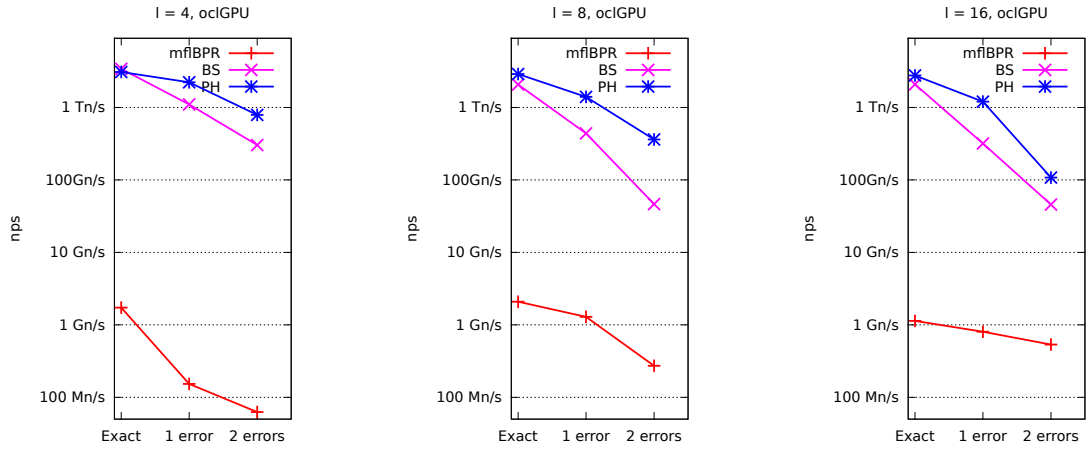

Figure 5.9: Performance comparisons between the mflBPR, the binary search and the perfect hashing solutions on the oclGPU benchmarking environment, for seed length $u = 4$ and neighborhood length $\ell = 4, 8, 16$ (from left to right).

	mflBPR	binary search	perfect hashing
Time complexity	$\mathcal{O}(\mathcal{N} \cdot ev/h)$	$\mathcal{O}(\log(\mathcal{N}) \cdot v^e)$	$\mathcal{O}(v^e)$
Access to data (fixed-length words)	$\mathcal{N}$ contiguous	$\leq \log(\mathcal{N}) \cdot v^e$ random	v^e random
`uint` operations for each step	$7 \times e$	3	54
Access to index			3 random
Index of 100 MB ($n \sim 2^{27}$), $u = 4, \lvert\Sigma\rvert = 4$			
SeedBlock size	2^{19}		
Global memory access	2^{19} (contiguous)	$19 \cdot v^e$ (random)	$(1+3) \cdot v^e$ (random)
Performance range (GPU, nps)	$10^7 - 10^{10}$	$10^{10} - 10^{12}$	$10^{11} - 10^{12}$
Performance range (CPU, nps)	$10^6 - 10^9$	$10^9 - 10^{11}$	$10^9 - 10^{11}$

Table 5.2: Comparations between mflBPR, binary search and perfect hashing methods .

neighborhood kept in one word (h). Thus ℓ affects the number of global memory accesses. As a result, the algorithm runs faster on smaller neighborhoods than on longer ones. The performance of mflBPR (5.2.1), in the case of exact matching (to avoid the impact of the error number), is a good example for this argument (figure-5.2 and figure-5.3).

We continue to discuss the binary search and perfect hashing in the context of exatch neighborhood matching, in order to isolate effect of the number of errors. Theoretically, the performance of both the binary search and perfect hashing approaches is independent from ℓ, as the neighborhood is always treated as an `unsigned int` or a string of 4 characters. However, as in figure-5.4, for the binary search, the running time of $\ell = 4$ is about $1.5\times$ faster than of $\ell = 8$ and $\ell = 16$. The reason is that we used the reduced structure, where the maximal number of neighborhoods in the SeedBlock depends on ℓ. For $\ell = 4, 8, 16$, these values are $2^8, 2^{16}, 2^{32}$, respectively. In the case of $\ell = 16$, with the limit in the text size $n \approx 2^{27}$, $\mathcal{N}$ is about 2^{19} (see section 5.3.1). It leads to a change in the maximal numbers of iterations when doing the binary search, either 8 or 16 or 19 (the $\log_2$ of the number of neighborhoods) which results in the observable difference between the case of $\ell = 4$ and the others.

Figure-5.6 also shows that the perfect hashing is independent of the size of SeedBlock with a running times very similar for $\ell = 4, 8, 16$ on the GPU. However, for the experiments on the multicore CPU, there is a difference between $\ell = 4$ and $\ell = 8, 16$. The reason for this difference could be the automatic usage of the L1 cache of the CPU when the size of SeedBlock fit in this cache ($64KB$).

In the case of approximate pattern matching, for the binary search and the perfect hashing methods, the length of neighborhood indirectly affects the performance because the number of degenerate patterns with errors $\Pi_e(q)$ depends on ℓ (see section 3.2.2). As we used one work-item to process one degenerate pattern, the size of the work-group is in $O(\ell^2)$. Large neighborhood sizes will thus imply more simultaneous random global memory access requests and also more possibility of having branch divergence.

5.3.3 Impact of the error threshold

The performance of mflBPR directly depends on the error number (e), but it does not cause any change neither in the number of global memory accesses nor in the possibility of having branch divergence.

For the binary search and the perfect hashing methods, as for the length of neighborhood, the error number contributes exponentially to the number of work-items needed – $\mathcal{O}(v^e)$. It also contributes to the growth of random global memory accesses and the possiblity of having branch divergence (see section 3.2.2). The combination of these two disadvantages, from both e and ℓ, leads to a decrease in performance dur to the increase of the number of error in all three cases of $\ell = 4, 8, 16$, as depicted in figure-5.4 and figure-5.4.

Figure-5.10 compares the performances between the binary search (5.2.2) and perfect hashing for the two benchmarking environments: oclGPU and oclCPU. On the multicore CPU, the performance of binary search and perfect hashing is almost similar. However, on the GPU, except for the case of exact matching where the performances of these two approaches are almost equal, perfect hashing always have a better speedup over binary search, in the best case $7.8\times$ faster ($\ell = 8$, 2 errors). It means than random global memory accesses and branch divergence less impact perfect hashing than binary search. Indeed, as the number of errors grows, the number of degenerated patterns grows faster and thus the number of queries in the

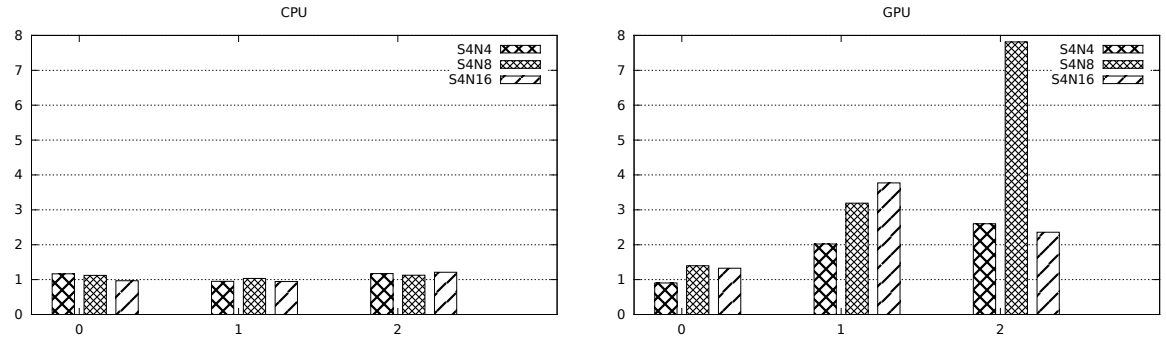

Figure 5.10: Speedup of query by perfect hashing over binary search on the neighborhood indexing on CPU(left) and GPU (right). The speedups are calculated based on the performances presented in figure 5.4 and figure 5.6.

index is larger. The binary search strategy is disadvantaged by its higher number of branch divergences and memory accesses compared to perfect hashing.

5.4 Conclusion

In this chapter, we demonstrated the efficiency of using OpenCL and GPUs to speed-up the neighborhood filtering phase extension in seed-based heuristics: in all our experiments, the implementation on GPU is at most 19× (mflBPR) or 13.1× (BS) or 29× (PH) faster than on CPU (even when using multiple cores on the CPUs through the OpenCL code). Moreover, both on GPU and on CPU, the PH and BS solutions are far more efficient than the direct BPR/mflBPR approaches. Finally, we can draw some general conclusions:

- mflBPR is a general solution for approximate neighborhood matching, allowing any number of errors and potentially complex edit operations. However, it is slowler than the binary search and the perfect hashing.
- In our experiments, the perfect hashing approach is the best solution for neighborhood matching with small Hamming distance. This is consistent with the theoretical analysis, as the perfect hashing requires only constant time for each work-item.

Note that the error model of BS and PH is not the same as the one of mflBPR, which can be used with edit distance. A fair comparison of performance between BS/PH and mflBPR could be approached in two ways:

- mflBPR can be run with Hamming distance (see section 3.1.2). Even if this leads to a reduction in the number of bit operators in the update operations, the whole list of neighborhoods in the SeedBlock must still be traversed. The runtimes should be a little faster, but it could stille not compete with BS and PH.
- It could be possible to run BS and PH with edit distance, by adding the insertions and deletion to each position in the error generating model (section 3.2.1). But this could be efficient only for very small distances, as the number of edits needed for each position is $|2 \times \Sigma + 1|$[48], instead of $|\Sigma|$ [Klus et al., 2012].

[48] $|\Sigma|$ edits for substitution, $|\Sigma|$ edits for insertion and 1 edit for deletion.

With the analyses of the impacts of the 3 factors u, ℓ and e, we showed that the manycore processors such as the Fermi GPUs have a good scability on our core problem, but are very sensitive on problems such as branch divergence and random global memory accesses.

Chapter 6

MAROSE
A Prototype of a Read Mapper for Manycore Processors

In the previous chapters, we explored several solutions for the efficient retrieval and approximate comparisons of the indexed neighborhoods, which can accelerate the filtering phase in seed-based heuristics applications for genetic similarity study (see page 43). This chapter presents the developements of these solutions into the whole process of mapping short genetic segments to a reference genome (**read mapping**). Many read mappers were proposed in the five last years thanks to the advances in sequencing technologies. In this chapter, we attempt to build a parallel read mapper using GPUs and OpenCL.

After an overview of available read mappers and their algorithmic foundations, we describe the feature analyses, the design and the implementation of our short read mapper prototype using OpenCL named **MAROSE** (Massive Alignments between Reads and Sequences). Finally, we present experiments of applying MAROSE on benchmarks involving the complete human genome, and compare against other read mappers that were evaluated by [Schbath et al., 2012]. A submission for a journal article on this readmapper is in preparation.

Contents

6.1 Motivations

Recent advances in sequencing technologies have led to the emergence of new fields of research in bioinformatics such as **re-sequencing** [Bentley, 2006] or **metagenomics** [NRC, 2007]. New sequencers are able to produce gigabytes of sequence data in short time at low cost. For example, the average cost of generating one million DNA base pairs in January 2012 is about 0.09$[49].

Such high-throughput applications require the design of new pairwise aligment tools capable of aligning a very large number of short sequences, namely **reads**, against a long reference sequence or a set of reference genomes depending on the application. This alignment process is called **read mapping**.

The length of reads often varies from tens to hundreds bases (for example: $50 - 100$ for Illumina and SOLiD sequencing, $300 - 400$ for 454 sequencing[49]), and the number of reads produced for sequencing a genome is very large: from millions to billions depending of the length of the genome and the accuracy (for example, the Illumina Hiseq system can produce 3 billions reads per run[50]).

The set of reference sequences is usually a genome bank made of billions base pairs. For example, the size of the human genome from the assembly 37.1 of the United States National Center for Biotechnology (NCBI)[51] is 2.7 Gbp[52].

The traditional sequence aligment tools such as FASTA [Lipman and Pearson, 1988], BLAST [Altschul et al., 1990], BLAT [Kent, 2002], MUMmer [Delcher et al., 1999] or HMMER[53] are not dedicated to process such a quantity of reads. Despite their efficiency both in computation time and in the quality of the results, they are not efficient enough to deal with big data. It thus requires another type of alignment tools able to map the large number of short reads to the genome bank in acceptable time authorizing some reduction of accuracy. Those tools, the **read mappers**, take as input a set of relatively short sequences (of length between tens and hundreds base pairs) and a set of long sequences named references, and output all the possible alignments between the whole read and the subsequences of the references. In recent years, many such softwares have been designed. We review some of them in the Section 6.2 and in the Table 6.1, with a special focus for GPU read mappers and "seed-and-extend" read mappers.

Our research and experiments on neighborhood indexing and approximate matching (see the previous chapters) can be directly applied to create the filtering phase of a seed-based read mapper. We thus propose a prototype, MAROSE, to evaluate the use of the techniques proposed in this thesis (Section 6.3), which will be tested and compared against some read mappers on real datasets (Section 6.4).

[49] Data cited from http://www.genome.gov/sequencingcosts/
[50] Data cited from http://www.illumina.com/Documents/products/Illumina_Sequencing_Introduction.pdf
[51] http://www.ncbi.nlm.nih.gov/projects/genome/assembly/grc/human/data/index.shtml
[52] This genome bank is further used to evaluate our read mapper (6.4.1)
[53] http://hmmer.janelia.org

6.2 Methods for read mappers

This section contains three parts. The first part introduces and classifies several recent read mappers (table 6.1). MAROSE, our prototype read mapper, is developed using seed-based heuristics (2.1.2), so the second part studies some read mappers using the same technique. The third part is about GPU read mappers as MAROSE is implemented to run on manycore processors such as the GPUs.

6.2.1 Classification of read mappers

One of the common points between these read mappers is the use of indexation in order to accelerate the search. Different methods of indexation led to plenty of the read mapping tools. The most recent surveys and evaluations on existing tools such as [Schbath et al., 2012], [Li and Hormer, 2010], [Horner et al., 2010], [Bao et al., 2011], *etc.* uses the indexing algorithms as the base for the classification of the read mappers. On the other hand, the sequences to be indexed by the tools are either the genomes or the reads. The Table 6.1 lists some read mappers, classified according to their *algorithms* and to their *indexing target*.

Algorithms classification. We follow here the classification proposed by [Schbath et al., 2012]. Considering indexing algorithms, there are **hashing-based** and **Burrow-Wheeler Trasform (BWT)-based** read mappers.

- **Hashing-based read mappers.** The principle of these read mappers consists in building a hash table for seeds (either contiguous k-mers or spaced seeds). To evaluate the occurrence of the whole read, several techniques may be used:
  - Similarly to the seed-based heuristics presented in Section 2.1.2, the positions of the seeds could be used to do a full alignment in order to calculate a matching score between the read and the corresponding segment of the genome, that is the *seed-and-extend technique*.
  - Another technique does not use all of the alignments: a simple combination of the occurence of the seeds allows to decide if it is an occurrence or not, precise or with errors, based on the *pigeon hole* principle.
- **Burrow Wheeler Transform-based read mappers.** Index data structures for hashing-based algorithms are very large. The Burrow Wheeler Transform (BWT) algorithms enables a smaller memory footprint. In this type of read mappers, the index is a *suffix-array* of the data (genome or set of reads), which would be further applied the BWT to get a compressed data structure allowing the search in $\mathcal{O}(p+occ\,log^{\epsilon}u)$ [Ferragina and Manzini, 2000]. This approach constructs a very compact index and avoids multiple queries in case of a repetition but it is not so efficient when errors are allowed.

Indexing target classification. Finally, the indexation can be applied for either the reference genomes or the set of input reads, which led to **genome indexing (GI)** and **reads indexing (RI)** (see Table 6.1). Indexing a large genome always requires more time and space. But this disavantage is acceptable because the indexes are created only once and stored in the hard disk

	Hashing based		BWT based		Others
	GI	RI	GI	RI	
Before 2008					
SSAHA2. Sequence Search and Alignment by Hashing Alogrithm [Ning et al., 2001]	✓				
2008					
MAQ. [Li et al., 2008a]		✓			
SOAP. Short Oligonucleotide Alignment program [Li et al., 2008b]	✓				
RMAP. [Smith et al., 2008]		✓			
ZOOM. Zillions of Oligos Mapped [Lin et al., 2008]		✓			
SeqMap. [Jiang and Wong, 2008]		✓			
2009					
segemeh1. Read mapper using enhanced suffix array, with mismatch, insertion, deletion [Hoffmann et al., 2009]					SA[a]
SOAP2. Short Oligonucleotide Alignment program 2 [Li et al., 2009]			✓		
BFAST. [Homer et al., 2009]	✓				
MPSCAN. Multi-Pattern Scan [Rivals et al., 2009]					Trie[b]
SHRiMP. The SHort Read Mapping Pakage [Rumble et al., 2009]		✓			
Bowtie. [Langmead et al., 2009]			✓		
BWA. Burrow-Wheeler Alignment tools (for short reads) [Li and Durbin, 2009]			✓		
PASS. [Campagna et al., 2009]	✓				
CloudBurst. [Schatz, 2009], the cloud computing version using Hadoop of RMAP		✓			
MOM. Maximum oligonucleotide mapping [Eaves and Gao, 2009]	✓	✓			
EMBF. [Wang et al., 2009]	✓	✓			
2010					
BWA-SW. Burrow-Wheeler Aligner's Smith-Waterman Alignment (for long reads) [Li and Durbin, 2010]		✓			
GPU-RMAP. [Aji et al., 2010], the massively parallel version of RMAP [Smith et al., 2008]		✓			
Novoalign. [Technologies,] commercial software, the free trial is available.		✓			
GASSST. Global Alignment Short Sequence Search Tools [Rizk and Lavenier, 2010]	✓				
2011					
SARUMAN [Blom et al., 2011]		✓			
SOAP3 [Liu et al., 2011a]			✓		
BWT-GPU [CIPF,]			✓		
BWT-GPU BWT based exact GPU read mapper [Chen and Jiang, 2011]			✓		
2012					
BarraCUDA. [Klus et al., 2012], the GPU version of **BWA**			✓		
CUSHAW. [Liu et al., 2012a]			✓		
GPU Exact Alignment. [Torres et al., 2012]			✓		
NGM2 [CIBIV,]		✓			
Bowtie2 [Langmead and Salzberg, 2012]			✓		
CUSHAW2 [Liu and Schmidt, 2012]			✓		

[a] Search by using the suffix array of the genome
[b] Reads are indexed by a trie-like structure.

Table 6.1: List of read mappers and classification. GI: Genome Indexing, RI: Read Indexing.

in order to be reused. Moreover, in the case of millions of reads, the size of an index on the reads may be similar to the one of an index on the genome.

6.2.2 A focus on read mappers with seed-and-extend heuristics

In the readmappers using hashing-based techniques to index seeds, some of them use full seed-based heuristics as presented in Section 2.1.2, with an extension to the neighborhood regions of the matching seeds in order to select the good candidate to do the full extension. This type of read mapper is also called "seed-and-extend" or "BLAST-like" with 3 main phases of seeding, neighborhood filtering and extending. The Table 6.2 summarizes the techniques used by three of those read mappers: PASS, EMPF and GASSST.

	seeding	filtering	extending
PASS [Campagna et al., 2009]	GI	PST	specialized dynamic programming
EMPF [Wang et al., 2009]	GI and RI	FVF	Smith-Waterman
GASSST [Rizk and Lavenier, 2010]	GI	PST and FVF	Needleman-Wunsch

Table 6.2: Table of seed-and-extend short read mappers, which generally consist of 3 phases: seed, filter and extend (GI: Genome Indexing; RI: Read Indexing; PST: Precomputed Score Table; FVF: Frequency Vector Filter)

The significal differences between these 3 read mappers is in the techniques of the filtering phase, which are applied on the flanking regions around the seed (the neighborhoods, see 2.1.2)

- PST, *"precomputed score tables"*, which uses 2-dimension matrices that keep the Lenvenshtein distances between 2 fixed length words.
- FVF, *"frequency vector filter"*, which bases on the counting of the difference number of each character between 2 fixed length words.

In comparison with these read mappers, MAROSE uses the approximate neighborhood matching techniques developed in the previous chapters to implement the filtering phase.

6.2.3 A focus on GPU read mappers

Along with the trend in GPGPU, in the recent years, there have been works and publications on GPU read mappers. We now focus of the read mappers using GPUs.

6.2.3.1 Hashing-based GPU read mappers

GPU-RMAP. [Aji et al., 2010] developed GPU-RMAP, a CUDA massively parallel version of RMAP [Smith et al., 2008]. GPU-RMAP parallelized the **mapping step**, which accounts for more than 98% of the running time of RMAP. GPU-RMAP allows only substitutions, but can be applied to unlimited number of errors.

GPU-RMAP indexes the reads. The genome, which is divided into segments, and the hash table of the reads are transfered to the GPU. The mapping step is implemented as 2 kernels:

- Kernel 1. Each thread scans the *corresponding segment*, looks-up and scores.
- A synchronization step after Kernel 1 distributes the reads among the threads of Kernel 2.
- Kernel 2. Each thread inspects the *respective read* in the list of correspoding sites, scores and chooses the best-mapped site for each read.

The authors discuss the use of *binary search* instead of the *hashing* technique, with collisions resolved by chaining, in the original RMAP tool. In GPU, the binary search on the look-up table, which is optimized with the hierachy of GPU memory, is better than hashing.

SARUMAN. [Blom et al., 2011] proposed SARUMAN (*Semiglobal Alignment of Short Reads Using CUDA and NeedMAN-Wunsch*). SARUMAN indexes the reads, selects the candidates by using a *q-gram lemma* (mapping phase) and aligns them with the corresponding segment of the reference genome with semi-global Needleman-Wunsch algorithm (aligning phase). The mapping phase run on the host, and the candidates are processed simultenously on the GPU. SARUMAN allows both substitutions and indels.

6.2.3.2 Burrow Wheeler Transform (BWT)-based GPU read mappers

BWT-based exact GPU read mapper. [Chen and Jiang, 2011] proposed an approach for high throughput exact matching based on BWT and GPUs. The reference genomes are indexed using BWT, the input reads are sorted and combined before searching on the BWT. The authors implemented both a CPU and a GPU version, with a speedup of at most 2.46× on a NVIDIA Tesla C2050.

BarraCUDA. [Klus et al., 2012] implemented the GPU version of the BTW searching phase of BWA[Li and Durbin, 2009]. To fit with the limit of the on-chip shared memory, the authors changed the traditional *breath-first-search* of BWA with a *different bound depth-first-search*. Moreover, the reads are divided into short fragments of size 32 to be serially processed through a pipeline of kernels to further exploit task-parallelism.

⋆ ⋆ ⋆

As this field of research is rapidely evolving, new read mappers were recently proposed. These GPU read mappers are also either hashing-based such as NGM2 [CIBIV,] or BWT-based such as CUSHAW [Liu et al., 2012a], SOAP3 [Liu et al., 2011a], BWT-GPU [CIPF,] and the GPU Exact Alignment of [Torres et al., 2012].

However, according to our knowledge, up to the middle of 2012, there is not any publication about a GPU read mapper which utilizes the full seed-based heuristics with all 3 phases of the seed-and-extend approach. Looking on the two hashing-based GPU read mappers presented in section 6.2.3, GPU-RMAP does not use the full extension and SARUMAN does not use the neighborhood filtering phase.

The following section presents MAROSE, our seed-and-extend GPU read mapper which does the filtering phase by using the neighborhood indexing approach.

6.3 MAROSE: Massive Alignments between Reads and Sequences

This section describes the design and implementation of MAROSE. Firstly, we introduce the general process of MAROSE to map a read onto a sequence. Then we present the OpenCL implementation to explain the design of the two kernels.

6.3.1 Mapping a read to a sequence by MAROSE

MAROSE maps the reads in accordance with the definition:

Given a read r, a sequence t and a parameter $a_e \geq 0$, find all the occurrences of r in t such that the Levenshtein distance between r and its occurrences in t is at most a_e[54].

Once an occurrence of a read r is found at position *pos* in the sequence t, we say that *"r is mapped to t at pos"*. Once a read is mapped to any position in a sequence, we say that there is a *hit*.

A DNA sequence is made of residues in {A, C, G, T, N}. For any ambiguous nucleotide N, we chose here to randomly change it to either A or C or G or T . Thus all the reads and sequences are finally written over an alphabet $\Sigma = \{A, C, G, T\}$.

The sequence t is indexed with the seed length u and the neighborhood length ℓ. The resulting index, denoted by I_t, is created once and stored in the hard disk. Each input read[55] is then processed independently through two phases (see Figure 6.1):

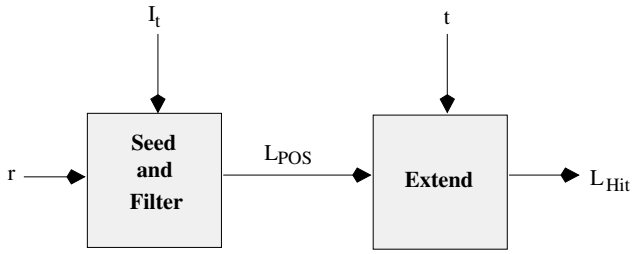

Figure 6.1: Process of mapping a read r onto the sequence t. I_t is the neighborhood index of t, L_{CPos} is the list of the putative position to do the full alignment of r on t. L_{Hit} is the list of hits of r as the result of the whole process.

Phase 1: Seed and Filter phase.

The input read is divided into a set of consecutive overlapping patterns $\{s_i q_i\}$. Each pattern is the concatenation of a seed s and a neighborhood q, where $|s| = u$ and $|q| = \ell$. For a read of length ℓ_r, there are thus $\ell_r - u - \ell + 1$ such patterns (Figure 6.2).

For each pattern sq, q is matched with at most f_e errors with the corresponding list of neighborhoods in SeedBlock(s) (Figure 6.3, a). Here f_e stands for *"filtering error"*. Each matching of q leads to one putative alignment position on the genome, called the *absolute candidate position* C_{ap}.

[54] In this definition, a_e stands for *"aligning error"* which is further discusted in 6.3.2.2

[55] Note that for a read r, we also implement the mapping process with its *reverse complement*, which is created in the running time. The read, and its reverse complement are processed independently.

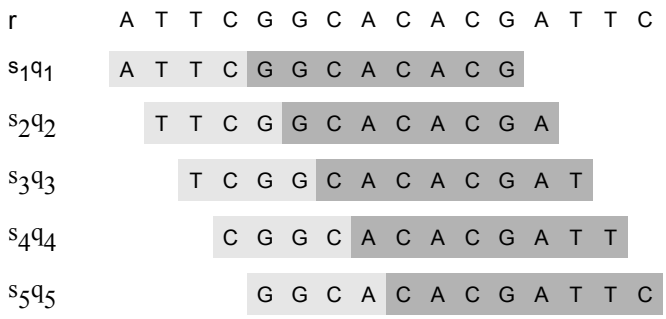

Figure 6.2: An example for processing an input read r of length $\ell_r = 16$ into a set of consecutive overlapping patterns $\{s_i q_i\}$. With a seed length of $u = 4$ and a neighborhood length of $\ell = 8$, the read $r =$ ATTCGGCACACGATTC of length 16 is processed into a set of $16 - 4 - 8 + 1 = 5$ patterns $\{s_1 q_1, ..., s_5 q_5\}$.

As the patterns overlap along the read, multiple patterns may match to one C_{ap}. After a "multiplication removing" step, we store only once each candidate position in the C_{ap} result list (Figure 6.3, b)[56]. The result of this phase is a list of unique C_{ap}, denoted by L_{CPos}.

Phase 2: Extend phase.

For each absolute position $C_{ap} \in L_{CPos}$, the read r is aligned with a subsequence t_{sub} of the sequence t, taken as follows (figure 6.3,c):

$$t_{sub} = t_{C_{ap}-a_e}, ..., t_{C_{ap}+\ell_r+a_e} \tag{6.1}$$

This subsequence allows to take into account any alignment with at most a_e insertion or deletion errors. If the alignment score is less than or equal to a_e, **the read r has one hit at position C_{ap} in the sequence t**. This phase outputs a list of absolute positions of the hits, denoted by L_{Hit}. This is the mapping result.

6.3.2 Parallel implementation

We now present the implementation of MAROSE on GPUs as a solution to process a large number of reads and huge reference sequences. MAROSE takes a set of reads $\mathcal{R} = \{r_1, ..., r_n\}$, a set of sequences $\mathcal{T} = \{t_1, ..., t_m\}$, and an alignment error a_e.

The sequences are further divided into k smaller subsequences $t_{i,1}, ..., t_{i,k}$, in order that the corresponding indexes $I_{t_{i,1}}, ..., I_{t_{i,k}}$ fit into the GPU global memory (with their subsequence). We thus finally consider a set of indexes $\mathcal{I}_\mathcal{T} = \{I_{t_{i,1}}, ..., I_{t_{i,k}}\}$.

Each index of $\mathcal{I}_\mathcal{T}$ is loaded in turns, from the hard disk to the main memory of the host and to the global memory of the GPU, to serve in the mapping process with $\mathcal{R}$. $\mathcal{R}$ is also divided into smaller batches of reads, denoted here by $\mathcal{B}_r$. Each batch of read is also loaded in turns to the GPU to map with the subsequence $t_{i,j}$.

[56] The process to remove multiple positions is based on the idea of GASSST [Rizk and Lavenier, 2010]

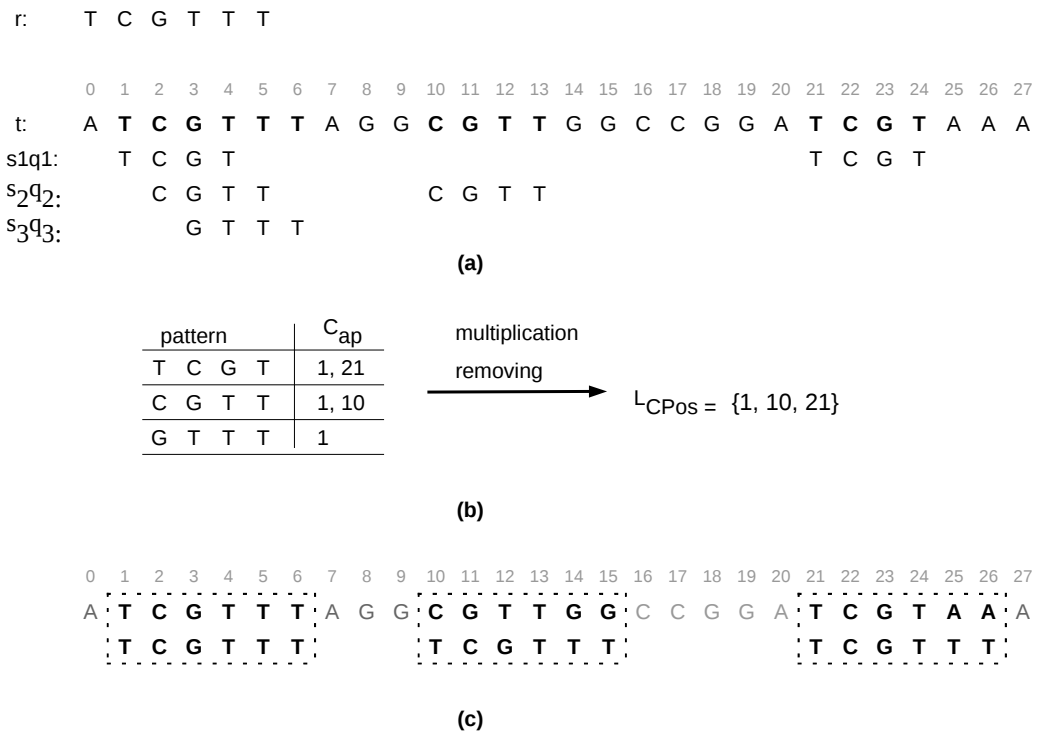

Figure 6.3: Example of exact mapping of between the read r = TCGTTT and the sequence t = ATCGTTTAGGCGTTGGCCGGATCGTAAA, with $u = 2, \ell = 2, f_e = 0$, and $a_e = 0$. (a) is the creation of consecutive patterns $\{sq\}$ and the result of the exact matching of each pattern along the sequence t (using the neighborhood indexing). (b) shows that the candidate position $C_{ap} = 1$ is found 3 times. After the multiplication removing, there are 3 unique candidate positions in the list L_{CPos}. (c) shows the exact alignment between r and the corresponding subsequences in t, from C_{ap} to $C_{ap} + \ell_r$ (as $a_e = 0$).

The execution of MAROSE now can be viewed as the result of *independent mappings between the batch of reads $\mathcal{B}_r$ and the subsequence $t_{i,j}$, using the index $I_{i,j}$*. We call this the "core process" of MAROSE. The rest of this section describes the parallel implementation of this core process, implemented with two OpenCL kernels, the `Seed − Filter` kernel and the `Extend` kernel.

6.3.2.1 The `Seed − Filter` kernel (phase 1)

There are two stages in the `Seed − Filter` kernel, **Filtering** and **Multiplication Removing**.

Stage 1: Filtering: It computes the approximate matching between the patterns sq, generated from the read r, with the corresponding list of neighborhoods in SeedBlock(s) of $I_{i,j}$. The absolute candidate positions (C_{ap}) returned from the match of each sq are kept in the `matching list`, denoted by L_{Match}.

Step 1a: Matching: Each read $r \in \mathcal{B}_r$ is processed independently by one work-group[57]. Each patterns sq is approximately matched with at most f_e substitutions against the nb_block in SeedBlock(s) by one work-item, independently. We implemented this step with two methods detailed in the previous chapters:

- Binary search (BS) with reduced structure,
- Perfect hashing (PH) with indexed block structure.

However, in the current version, we have not implemented the model of generating Hamming-distance patterns (3.2.1), but rather used the following techniques:

- With binary search (BS), the degenerated patterns are created and compared to q sequentially;
- With perfect hashing (PH), the set of combinations of substituting positions (SP) is traversed sequentially. However, for each combination, $|SC|$ degenerated patterns, created by changing the characters in each set of positions, are queried in parallel by using the vector data type.

It means that we only take advantage of the efficiency of the neighborhood retrieving of the index. The advantage of parallel approximate matching has not yet been full exploited.

Step 1b: Positions retrieving: Each match of sq corresponds to a list of positions in pos_block. This step accesses to the nb_pos_block and the pos_block the global memory to retrieving the position in order to calculate the list of C_{ap} of r in the reference sequence. The C_{ap} found by all work-items is gathered to the L_{Match}, thus duplicates could arise, as one C_{ap} may occur multiple times in L_{Match}. L_{Match} is kept in the local memory of the work-group, on which each work-item has a region of fixed size to store its C_{ap}.

Stage 2: Multiplication Removing. A traversal of L_{Match} allows to remove the multiplication of each C_{ap}. The list of unique C_{ap} is stored in L_{CPos}. This is implemented by using a *"sort and reduce"* strategy with 3 steps:

[57] In this version of MAROSE , we experimented with 40,000 work-group for the `Seed − Filter` kernel. Thus, there were 40,000 reads that is mapped simultaneously ($|\mathcal{B}_r| = 40,000$)

Step 2a: Sorting: As L_{Match} is kept in the local memory, it can be accessed by any work-item in the work-group. To sort L_{Match}, we used a parallel version of a *radix sort* where the whole list is shared and sorted together by 16 work-items.

Step 2b: Reducing: After the list of absolute positions is sorted, it is traversed to reduce the multiplications of each C_{ap}. After this step, each read has the list of unique absolute candidate position in the reference sequence (L_{CPos}).

Step 2c: Output writing: L_{CPos} is written to the global memory[58] of the GPU, ready to be processed by the `Extend` kernel.

6.3.2.2 The `Extend` kernel (phase 2)

The `Extend` kernel is implemented with a *banded semi-global alignment* algorithm, which allows us to find any alignment within a a_e Levenshtein distance (substitutions, insertions and deletions). The space complexity of this algorithm is $\mathcal{O}(a_e \cdot \ell_r)$, instead of $\mathcal{O}(\ell_r^2)$ if the full dynamic programming matrix is computed.

The L_{CPos} from all reads $r \in \mathcal{B}_r$, returned by the work-groups, are gathered to create the alignment candidate list, denoted here by L_{AC}. We denote the size of L_{AC} by S_{AC}. Each element $AC(r, C_{ap}) \in L_{AC}$ corresponds to an alignment between a read r and a subsequence t_{sub}, located in the reference sequence from $C_{ap} - a_e$ to $C_{ap} + \ell_r + a_e$.

We launched the `Extend` kernel with $\mathcal{S}_{AC}$ work-items, each work-item process one alignment candidate in L_{AC} independently. The alignment results are gathered into the L_{Hit} list which is returned to the host.

6.4 Results

Following the benchmarks of [Schbath et al., 2012], MAROSE was evaluated on a complete human genome in two cases: exact mapping and approximate mapping allowing 3 substitutions. The following subsections will describe and analyze the experiment results both on performance and on quality.

6.4.1 Experiment data sets and setups

The data sets. To evalutate MAROSE, we used the datasets on the human genome proposed by [Schbath et al., 2012]. This dataset consists of two sets of reads, $\mathcal{H}_0$ and $\mathcal{H}_3$, which are queried against a reference genome bank $\mathcal{H}_{ref}$:

- $\mathcal{H}_{ref}$ is the human genome from the assembly 37.1 of the United States National Center for Biotechnology (NCBI). It contains 25 human chromosomes (1 to 22, X, Y and M) and has a total size of 2.7 Gbp.
- $\mathcal{H}_0$ is a set of 10 millions reads of length 40, extracted uniformly from $\mathcal{H}_{ref}$.
- $\mathcal{H}_3$ was created by adding exactly 3 randoms substitutions to each read in $\mathcal{H}_0$. Thus, there are 10 millions reads of length 40 in $\mathcal{H}_3$.

[58] The OpenCL atomic operation `atomic_add()` is used for this purpose.

	u	ℓ	Reduced		Indexed Block	
			T_{Index}	size (GB)	T_{Index}	size (GB)
$\mathcal{IH}_{U8L16}$	8	16	43m 31s	30.10	91m 21s	37.73
$\mathcal{IH}_{U7L8}$	7	8	34m 55s	27.13	82m 38s	33.57

Table 6.3: The creation time and the size of neighborhood indexes for the human genome bank $\mathcal{H}_{ref}$, which contains 25 sequences for a total size of 2.7 GB.

Experiment	f_e	a_e	Read	Index
$EChr10_0$	0	0	$\mathcal{C}hr10_0$	$\mathcal{IC}hr10_{U8L16}$
$EChr10_3$	1	3	$\mathcal{C}hr10_3$	$\mathcal{IC}hr10_{U7L8}$
EH_0	0	0	$\mathcal{H}_0$	$\mathcal{IH}_{U8L16}$
EH_3	1	3	$\mathcal{H}_3$	$\mathcal{IH}_{U7L8}$

Table 6.4: Four experiments to evaluate MAROSE on the human genome bank $\mathcal{H}_{ref}$.

Indexes on the reference genome. We created two indexes for $\mathcal{H}_{ref}$, denoted by $\mathcal{IH}_{U8L16}$ and $\mathcal{IH}_{U7L8}$, including both reduced structure and indexed block structure. The seed length, the neighborhood length, the total size and creation time of these indexes (T_{Index}) are presented in Table 6.3. More details on the size of these indexes can be found in appendix (Table A.3 and Table A.4).

Benchmarking environment. We used the oclGPU environment (see page 82): all the results are from the experiments on a host with 8 GB RAM and an Intel Xeon E5520 which uses an NVIDIA GTX 480 GPU as an OpenCL device.

Experiment setups. We evaluated MAROSE with the complete human genome $\mathcal{H}_{ref}$ for two cases: exact mapping and approximate mapping with 3 substitutions. These two experiments are denoted here by EH_0 and EH_3 with the parameters presented in Table 6.4.

We will analyze the 5 steps of the seed and filter kernel: matching, position retrieving, sorting, reducing and output writing (6.3.2.1). In order to measure the running time of those 5 steps, we measured the running time of each step in the experiments with the data corresponding only to the chromosome 10. It means that, in $\mathcal{H}_0$ and $\mathcal{H}_3$, we mapped only the reads extracted from the chromosome 10 (denoted here by $\mathcal{C}hr10_0$ and $\mathcal{C}hr10_3$). Both $\mathcal{C}hr10_0$ and $\mathcal{C}hr10_3$ contains $460,544$ reads of length 40. For the index, we used only the indexes of chromosome 10 in $\mathcal{IH}_{U8L16}$ and $\mathcal{IH}_{U7L8}$ (denoted here by $\mathcal{IC}hr10_{U8L16}$ and $\mathcal{IC}hr10_{U7L8}$). It leads to 2 other experiments, denoted by $EChr10_0$ and $EChr10_3$ (see table 6.4).

In each experiments, we run the `Seed − Filter` kernel with both binary search (BS) and perfect hashing (PH) methods.

6.4.2 Running times of the two main kernels

Table 6.5 presents the results of EH_0 and EH_3, which includes:

- T_{K1}: running time of the `Seed − Filter` kernel.

	Experiment	T_{K1} (second)	Candidate	T_{K2} (second)	Hit	T_{rest} (second)	T_{map} (second)
BS	EH_0	2672.60 (65.89%)	6582496108	151.60 (3.74%)	547034208	1231.98 (30.37%)	4056.18 (1h 7m)
PH	EH_0	2402.10 (60.40%)	6582496108	151.60 (3.81%)	547034208	1423.28 (35.79%)	3976.98 (1h 6m)
BS	EH_3	21051.81 (79.73%)	89844054473	3419.95 (12.95%)	811734862	1932.83 (7.32%)	26404.59 (7h 20m)
PH	EH_3	6673.06 (58.27%)	89844054473	3396.09 (29.66%)	811734862	1381.85 (12.07%)	11451.00 (3h 11m)

Table 6.5: Experiment results of exact mapping (EH_0) and approximate mapping with 3 substitutions (EH_3) on $\mathcal{H}_{ref}$.

- Candidate: number of candidates for the extend phase.
- T_{K2}: running time of the `Extend` kernel.
- Hits: number of hits (we have one hit once a read is mapped to any position in a sequence) after the extend phase.
- T_{map}: total running time of the whole application.

It should be noted that T_{K1} and T_{K2} are only the running time of the two kernels[59]. T_{map}, the running time of the whole application, is measured by the `time` command of the Linux operating system. T_{rest}, which includes the times of read preprocessing, index loading, data exchanging between the host and the OpenCL device, mapping results writing, is computed as $T_{map} - T_{K1} - T_{K2}$.

In the case of exact mapping, the running time of BS is not very different from PH. But PH is 2.3× faster than BS in the case of approximate mapping. This comes from a 3.15× speedup of the `Seed` − `Filter` kernel (matching with 1 substition over neighborhoods of length $\ell = 7$).

In all of these experiments, exact and approximate mapping, the majority of running time is for the `Seed` − `Filter` kernel. The time contributions of each step in the kernel are detailed in Table 6.6 for both $ECHr10_0$ and $ECHr10_3$ experiments.

It shows us that the step that mostly consumes computation time in each experiment is the `sorting` step for exact mapping and the `matching` step for approximate mapping, which is not suprising.

For approximate mapping, the running time percentage of the `Extend` kernel is higher than the one of exact mapping (12.95% over 3.74% for BS and 29.66% over 3.81% for PH) because it aligns much more candidates (about 13.65×). In addition, the number of operations of banded alignment with 3 errors is 7× more than the one without errors. We can also computed the average aligning time for a candidate: about $3.81 \times 10^{-8}s$ with 3 errors and about $2.30 \times 10^{-8}s$ for exact aligning. So the processing time grows 1.65× while the number of compute operations grows 7×.

[59] Time measured by profiling the return events of the clEnqueueNDRangeKernel OpenCL command

	BS		PH	
step	$ECHr10_0$	$ECHr10_3$	$ECHr10_0$	$ECHr10_3$
1a. Matching	0.77 (9.62%)	29.22 (66.15%)	0.009 (0.18%)	6.32 (44.98%)
1b. Position retrieving	0.023 (0.29%)	0.78 (1.77%)	0.25 (5.07%)	0.41 (2.92%)
1c. Sorting	6.16 (76.97%)	11.76 (26.62%)	4.12 (83.59%)	4.51 (32.10%)
2a. Reducing	1.02 (12.75%)	2.26 (5.12%)	0.53 (10.75%)	2.59 (18.43%)
2b. Output writing	0.03 (0.37%)	0.15 (0.34%)	0.02 (0.41%)	0.22 (1.57%)
T_{K1} (s)	8.003 (100%)	44.17 (100%)	4.929 (100%)	14.05 (100%)

Table 6.6: Running time of 5 steps in the `Seed` − `Filter` kernel (in seconds) and corresponding percentages relative to the total time of the kernel T_{K1}.

In EH_3, the number of candidates is about 13.65× greater than that of EH_0 while the number of hits is only 1.48× greater than EH_0. It leads to the conclusion that one has to pay attention in the selections of the seed length (u), the neighborhood length (ℓ) and the filter error (f_e). Indeed, with wrong parameters, the `Seed` − `Filter` could outputs *too many candidates* rather than *many good candidates*, resulting in a waste of time during the alignment process. Searching for best values and a good tradeoff should be a rewarding work for the future.

6.4.3 Comparisons with other read mappers, in different platforms

This subsection gives a comparison of the running time and sensibility of MAROSE with the softwares presented in [Schbath et al., 2012]. EH_0 was used to compare 11 softwares (table 6.7) in the case of exact mapping. As MPScan can not map with error, and SOAP2 can only treat up to 2 substitutions, only 9 softwares were used in the case of approximate mapping, with the experiment EH_3.

We evaluated both the speed and the sensibility of MAROSE :

- **Speed**: we compared the indexing time (T_{Index}) and the mapping time (T_{map}). As the sequences are indexed only 1 time we used T_{map} of the PH method to rank the speed.
- **Sensibility**: we used the number of reads which are mapped as the main factor to evaluate the sensibility. In addition, two other factors were used:
  - Unique reads: In $\mathcal{H}_0$, there are $8,877,107$ reads that occur exactly once in $\mathcal{H}_{ref}$[60]. The ability to find all unique reads is a key factor to evaluate the sensibility of a read mapper.
  - Mean number of multiple hits: For the reads that have multiple hits, the mean number of hits was used to evaluate the capacity to find all the occurrences of each read in the targets sequences.

[60] Schbath et al used a dedicate naive algorithm to compute the occurrence number in $\mathcal{H}_{ref}$ of each read in $\mathcal{H}_0$

It should be noted that the platform to run MAROSE and the one to run the other softwares are different. In addition, these softwares ran with only 1 thread on the CPU, while MAROSE run on GPU. Thus, the comparision of speed in this section is here not significant. The T_{Index} in Table 6.7 and Table 6.8 still function as a performance parameter, but the corresponding analysis will focus on the sensibility. A comparison between MAROSE and BWA (version 0.6.2) [Li and Durbin, 2009] on a same platform with multiple threads will be presented in Section 6.4.4.

The `Reference` row in Table 6.7 gives the true numbers of the unique reads, of the multiple hit reads, and the true mean of the number of multiple hits. However, until now we did not have the list of unique reads of this benchmark, so in Table 6.7 and Table 6.8, the value in the unique read column of MAROSE is the number of reads that have only one hit in the results of EH_0 and EH_3. In the current results, a unique read found by MAROSE should be either the true unique read or a read with multiple hits that is mapped only once. We will thus report a unique hit number of MAROSE as the reference with the other read mappers and focus on the number of mapped reads and the mean of multiple hits to analyse the sensibility.

Software	Memory usage (GB)	Indexing time	Mapping time	Non-mapped reads	Mapped reads	Uniquely hit	Multiple hits	
							Number	Mean
SOAP2	51.78	1h 56m	56m	49	9999951	8877061	1122890	653.26
MAROSE	37.73	91m 21s	1h 6m	21	9999979	8877100	1122879	479.26
BWA	2.18	1h 35m	1h 13m	49	9999951	8877061	1122890	722.81
MPSCAN	2.67	1h 20m		26	9999974	8877081	1122893	722.81
Bowtie	7.36	3h 25m	2h 42m	49	9999951	8877061	1122890	722.81
GASSST	57.93	8h 45m		49	9999951	8877061	1122890	722.47
PerM	13.77	13h 05m		115871	9884129	8877068	1007061	126.42
Novoalign	8.12	8m	13h 24m	632	9999368	8877107	1122261	698.63
BFAST	9.68	18h 01m	15h 02m	726332	9273668	8840305	433363	2.96
SSAHA2	9.60	24m	1d 1h	35875	9964125	8886204	1077921	79.52
Reference						8877107	1122893	722.81

Table 6.7: Comparison between MAROSE and others short read mappers in the benchmark [Schbath et al., 2012] in the case of exact mapping. This table is built from the results of experiment EH_0 and the data from Table 3 and Table 4 in [Schbath et al., 2012]. For MAROSE, the index size and the indexing time are the ones of the indexed block structure, and the mapping time is the one of the PH method.

.

Exact mapping (Table 6.7). MAROSE is the second faster (but remember that the speed comparison is here not significant), but has the highest mapped reads number. All 21 non-mapped reads contain at least 1 residue `N`. For 8877100 unique reads, there are 10 reads that are not mapped to the original position (from where they were extracted). It means that there are at least 10 reads which have multiple hits but are mapped only once. The number of multiple hit reads found by MAROSE is 1122879, less than the reference value of 14 reads ($1122893 - 1122879$). It allows us to conclude that the difference in the number of unique reads of MAROSE is between 10 and 14 in comparaison with the referent number. *This difference is very small in relation with the number of reads and can not cause a significal change in the accuracy of the reported results.*

However, the value of the mean of multiple hits shows that we can find about 66.3% of the total hits of the reads that occur more than 1 time in $\mathcal{H}_{ref}$ while BWA, MPSCAN, Bowtie and GASSST can find 100%. This problem is further discussed in the results of the experiment in

approximate mapping (see page 112).

For the running time, Table 6.6 shows that most of the time is taken in the `sorting` step in the `Seed − Filter` kernel. As this step is limited by the local memory size (without dynamic allocation of the GPU), if we keep the structure of MAROSE as in this version, it is difficult to find a solution to improve this point.

Table 6.5 shows that 30.37% or 35.79% of the total running time is for the preprocessing and the data exchanges. It is a disadvantage of MAROSE as large sequences are divided into smaller subsequences so that their indexes fit with the global memory of the GPU (1.5 GB here). The indexes of the subsequences are loaded in turns, from the hard disk to the main memory of the host, and from the main memory to the global memory of the GPU (more details about the number of subsequences in $\mathcal{H}_{ref}$ can be found in Table A.3). However, this problem is only significant when we run MAROSE on a single host and with a GPU with limited global memory. It can be solved by using the latest GPUs with larger memory.

Moreover, the most important thing is that thanks to independent processing of the indexes of the subsequences of MAROSE, it is now very easy to run on a distributed computing platform such as a cluster or a grid, thus providing a high scalibility with input data. *It means that MAROSE exchanges the advantage when running in a single host with the high potential of parallelism in distributed computing platforms.*

Software	Memory usage (GB)	Indexing time	Mapping time	Non-mapped reads	Mapped reads	Uniquely hit	Multiple hits	
							Number	Mean
MA ROSE	33.57	82m 38s	3h 10m	11573	9988427	8424955	1563472	513.79
Bowtie	7.36	3h 25m	9h 57m	49	9999951	8496649	1503302	1161.98
PerM	13.75	12h 25m		186752	9813248	8496655	1316593	147.25
BWA	10.01	1h 38m	17h 04m	49	9999951	8496649	1503302	1161.98
BFAST	9.68	18h 01m	10h 02m	199451	9800549	8476476	1324073	6.17
GASSST	27.14	1d 12h		326598	9673402	8193650	1479752	1139.25
Novoalign	8.12	8m	2d,6h	47	9999953	8699117	1300836	15.12
SSAHA2	9.60	24m	3d 11h	213	9999787	8286416	1713371	6.81

Table 6.8: Comparison between MAROSE and others short read mappers in the benchmark of Schbath et al [Schbath et al., 2012]. in the case of approximate mapping with 3 substitutions. This table is built from the results of experiment EH_3 and the data from Table 5 and Table 6 in [Schbath et al., 2012]. As in Table 6.7, for MAROSE, the index size and the indexing time are the ones of the indexed block structure, and the mapping time is the one of the PH method.

Approximate mapping with 3 substitutions (Table 6.8). MAROSE ranks first in speed (same remark applies), but fifth in the number of mapped reads. This experience highlights the problem of multiple hits already pointed out in the results of exact mapping. The reason for this problem is that the size of the matching list (L_{Match}) and the absolute positions list (L_{CPos}) inside the `Seed − Filter` kernel are small and can not be dynamically allocated. It leads to the miss of good candidates when a read occurs multiple times in the reference sequence as the `matching step` returns a lot of results and causes an overflow in these two lists. It is one of the most critical problems in the current version of MAROSE. We tried to solve with 2 solutions:

- Increase the size of L_{Match}, in the restriction of the size of the local memory (48 KB on the NVIDIA GTX 480).
- Use a *"heuristic index structure"* with a threshold to limit the number of occurrences of *sq*.

But the sensitivity was not significally increased. The reason could be that the number of occurrences of each *sq* is very large while the local memory size of the GPU is very limited. In the case of using global memory to keep the matching results and the candidates, one must modify the structure of current version of our software.

Table 6.6 shows that most of the time is consumed in the `matching` step. As presented in 6.3.2.1, in the current version, the model of generating Hamming-distance patterns (3.2.1) has not been fully applied and the degenerated patterns are created and compared to q sequentially. This leaves scope for significantly improvement of the running time of MAROSE in the case of approximate mapping once this step is fully parallelized.

6.4.4 Comparisons between MAROSE and BWA on a same platform

This section presents the comparison between MAROSE and BWA (version 0.6.2) in the oclGPU experiment environment (page 108). We perform the approximate mapping with 3 substitutions of 460,544 reads of size 40 onto the human chromosome 10 (experiment $ECh r10_3$ in Table 6.4). BWA runs on the Intel Xeon E5520 in two cases: single thread and 8 threads.

According to Table 6.7 and Table 6.8, BWA is one of the most sensitive short read mappers while MAROSE only ranks the fifth. In this section, we only compare the running times, which are reported in Table 6.9.

For BWA, the running time is significantly different between ungapped alignment and gapped alignment. The number of allowed gaps (among the number of errors) is configured by using the `-o` parameter. For the experiment $ECh r10_3$, the gap is not allowed (`-o 0`). In this case, MAROSE runs in about the same time than BWA on 8 threads, and about 6.9× faster than BWA on 1 thread.

When the gaps are allowed, the running time of BWA increases substantially while for MAROSE it does not change thanks to the full extend phase. One conclusion can be drawn from this is that MAROSE can run much more faster than BWA in the case of mapping with gaps, but with a decrease in sensibility.

	MAROSE	BWA					
		`-o 0`		`-o 1`		`-o 2`	
		1 thread	8 threads	1 thread	8 threads	1 thread	8 threads
Running time (s)	31.56	210.97	34.55	635.75	130.42	825.05	161.72
Speedup of MAROSE	–	6.9×	1.09×	20.14×	4.13×	26.64×	4.8×

Table 6.9: Running time comparison between MAROSE and BWA.

6.4.5 Prospective features of MAROSE

We report now preliminary tests on improvements for MAROSE :

- **Non-consecutive patterns**: the successive patterns created from the input read may be not consecutive, but taken with a shift of δ positions to the right. For a seed of size u and a neighborhood of size ℓ, the read r of size ℓ_r is processed into a set of $[(\ell_r - u - \ell)/\delta] + 1$ patterns (Figure 6.4, a). The number of patterns can be significantly reduced, so the speed of the matching phase in the $\texttt{Seed} - \texttt{Filter}$ kernel could be improved but with a trade-off in sensibility.

Moreover, using this feature, we can tune the number of created patterns to fit with the computation and storage capability of the device. It gives to MAROSE a possibility of processing reads of larger sizes.

- **Spaced seeds**: As being proved in various researches such as [Brown, 2008], spaced seeds [Ma et al., 2002] can improve the sensibility of the read mapper. To use spaced seed in MAROSE , the indexes needs to be recomputed as the seeds are changed (Figure 6.4, b).

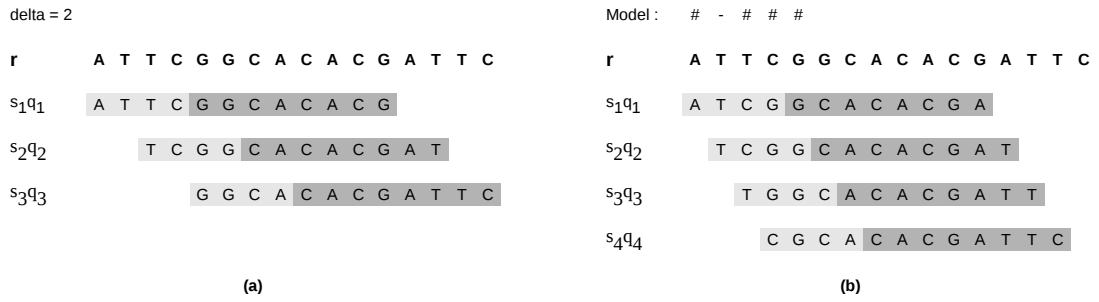

Figure 6.4: Example of non-consecutive patterns and spaced seeds, on a read r of length $\ell_r = 16$. (a) shows the creation non-consecutive patterns from r with the step $\delta = 2$, and (b) shows the creation of spaced seeds with the seed model `#-###`.

The implementation of these two extended features did not critically change the structure of MAROSE , the only modification being in the process of creating the patterns from the input read. Table 6.10 shows the results of experiment $ECHr10_3$ when performed with non-consecutive patterns of step $\delta = 2$ and with a spaced seed of model `####-####`. The result of $ECHr10_3$ in Section 6.4.2 and Section 6.4.4, corresponding to the case of seed model = `########` and $\delta = 1$, is used as a reference. These results confirms the intuitions on these techniques:

- Using non-consecutive patterns, the mapping time is significally improved, but the sensibility is of course reduced;
- Using spaced seeds, the mapping time and the total number of hits are not significaly changed, but the sensibility is improved (as expressed by the decrease in the number of non-mapped reads).

Seed Model	*delta*	Hits	Non-mapped reads	mean of multiple hits	T_{map} (second)
`########`	1	2013786	1345	31.52	30.54
`########`	2	1677504	2592	25.34	22.19
`####-####`	1	2036021	893	31.76	31.19

Table 6.10: Experiments with nonconsecutive patterns and spaced seed

By using non-consecutive patterns, MAROSE is able to treat long reads. We have run experiments with reads that were as long as 400bp. As those experiments are yet under analysis, we are not ready to show comparisons with other read mappers.

6.5 Conclusion

This chapter presents the integration of the solutions presented in the previous chapters into a real application about read mapping. MAROSE, a prototype of short read mapper, demonstrates a striking performance. Even in this preliminary version, in which we only applied a solution for "efficient neighborhood retrieving" and the "approximate neighborhood comparison" was not fully exploited, the speed and the ability to process large data is proved. In addition, MAROSE can be adapted to use other techniques such as the spaced seeds and also has the capability of processing longer reads. Thus, MAROSE is a promising prototype for a powerful GPU read mapper.

Chapter 7

Conclusions and Perspectives

Focusing on parallel data structures and algorithms to solve the problem of massively approximate pattern matching, our work considered neighborhood indexing as the method for building data structures on manycore processors such as the graphical processing units (GPUs). For this purpose, we investigated two main approaches: i) direct matching and ii) matching with indexes created using perfect hashing functions. Algorithms and data structures were implemented and tested, and their performances were analyzed and compared. This work ended with the development of a prototype of a manycore read mapper as an application case.

7.1 Conclusions

Our work started with the background of programming on manycore processors with the Open Computing Language (OpenCL) and the brief summary about the usages of GPUs in bioinformatics. We showed that a number of applications using such devices already exist and that they could bring substantial speedups.

Chapter 2 explained the main target of our work and why we focused on developing methods about approximate matchings between fixed length words. Indeed, such a basic tool involved in numerous applications, especially for the *filtering phase* in *seed-based heuristics* commonly used to compare large genetic sequences. Thus, it was worth an effort to design efficient methods for the GPUs. This problematic leads to the requirements of indexing the occurrences of small words, namely the *seeds*, in large genetic sequences. The neighborhood indexing approach was choosen with adaptation with the behaviours of the *global memory accesses* of the GPUs. Chapter 2 also discussed the features of the algorithms that can be efficiently mapped on GPUs for which the aim is to avoid *divergent branch* problem.

Chapter 3 presented two solutions to the appoximate pattern matching for the neighborhoods kept in the *flatten lists*, what we called direct matching.

- The first solution is the usage of the well-known Bit Parallel Row-wise (BPR) algorithm, an appoximate matching algorithm, we adapted for a *set of fixed length words*: mflBPR. The expriments in chapter 5 showed an improvement in the speedup of mflBPR over BPR, thus proving the effectiveness of our apdatation. Moreover, running on the GPU, our implementation achieved at most 19× faster than the same parallel version on the multicore CPU, 62× faster than the corresponding serial version. Even in the worst case, it

is 15× faster than `agrep`, the original implementation of BPR. Thus, mflBPR should be an excelent solution for high throughput general approximate matching with the Levenshtein distance on GPU, especially for short patterns over small alphabet.

- The second solution seeks to simulate the errors by producing a set of degenerated patterns from an initial pattern. We used the binary search strategy to search in an index which was parallelized by simultaneously matching with the degenerated patterns. The advantage of this solution is the simplicity of the algorithm, as binary search is one of the most popular searching algorithms. Again, performance analysis in chapter 5 showed that this further stage of indexing, for small Hamming distance, could give up to 1000× improvement on a non-indexed solution such as mflBPR.

Chapter 4 proposed a solution where the neighborhoods (inside each SeedBlock) are further indexed. We investigated the usage of perfect hashing functions to do that. While the number of arithmetic operations is more intensive as compared to the 2 previous solutions, this solution is appealing because it can be implemented to obtain a constant time complexity, which is important for application on GPUs. As described in chapter 5, the performance of such approach outweighed the two preceeding solutions, being about 10× faster than the binary search on the experiments with small Hamming distance.

In chapter 5 we presented some performance measurements, analyzing the impact of several parameters. The efficiency of the implementations on the GPUs is proved. The advantages, the disadvantages of each solution in relation with various arguments from both the computing device and the data structure are pointed out. The performance analyses showed that GPUs are able to deal with intensive computations, but are very sensitive to the branch divergence and random accesses to the global memory.

Finally, in chapter 6 we attempted to demonstrate the usage of our indexes and matching strategies in a "real" application. We thus developed a prototype of a parallel read mapper named **MAROSE** (Massive Alignment between Reads and Sequences). Though only a prototype version, the efficiency of MAROSE was proved with the experiments on real data and in comparison with other read mappers: in some cases, MAROSE ranks as one of the fastest read mappers with an acceptable sensitivity.

7.2 Perspectives

Scalability with large data. The solutions in this thesis were designed and customized for current GPUs, but we believe that the principles studied here will also apply to future manycore processors, as soon as some high-throughput data is divided and processed simultaneously on multiple processing elements. Moreover, the distribution of the work-items to the processing elements is transparent for the programmer and is implicit in the source code. The solutions developed in this work have drawn from this and can thus will run on other GPUs/manycores with any number of processing units, without any requirement for critical modifications.

Moreover, when dealing with large data sets, the sequences can be divided into subsequences in order to create neighborhood indexes that can fit in the global memory of GPUs (see what we did to process the complete human genome in chapter 6). As these indexes can be loaded to GPU and processed independently, our solutions have high scalability by either adding GPUs to the running environment or using GPUs with higher computing capability.

Optimization on Other OpenCL Platforms. Writing in OpenCL, our codes can work on different platforms as shown in Chapter 5. But until now, all the designs and implementation was developed with the NVIDIA Fermi GPU in mind. Also, the experiments on multicore CPUs were only carried out to demonstrate the true portability of OpenCL and were not tuned for other modern GPUs or multicore CPUs. That is, the code can be compiled and performed but its efficiency is not the same on all platforms (independently of the device efficiency). For example, the mflBPR was tested on ATI GPU cards (Radeon 5870), but unfortunately gave poor performance (best result peaking at 39.6 Mw/s on a smaller index, not shown). This opens up the problem of writing a unique code that can perform well on different platforms, with different architectures.

One of the perspectives for solving this problem is that we can test and analyze several customizations of the source code and of the launching arguments so that these implementations can work with the best performance on different OpenCL computing devices.

Optimization on the Memory Hierachy of the GPU. Among the three proposed approximate matching methods, only mflBPR takes advantage of the local memory as the neighborhoods are processed independently, allowing a SeedBlock to be divided in smaller segments which are sequentially loaded into the local memory. For the binary search and the perfect hashing methods, even if the number of accesses to the SeedBlock should be much less than that of the mflBPR solution due to the non-contigous and random nature of the accessed neighborhoods, the SeedBlock cannot be divided to be loaded to the local memory.

One of the possible solutions would be to use the size of the local memory as a threshold for the size of a SeedBlock. In this case, some neighborhoods may be lost, leading to false negative results: we will miss some occurrences. It could be worth testing such approach as trade-off between the speed and the accuracy.

Use with Protein Sequences. The work in this thesis is limited to nucleotide sequences ($|\Sigma| = 4$). For the protein sequences ($|\Sigma| = 20$), the size of the index could be theoretically 2.5× larger, or 4× larger in practice. Experiments of neighborhood indexing on protein sequences can be found in [Nguyen and Lavenier, 2009]. Even there could be some overheads of data loading, some advantages of neighborhood indexing on GPUs (such as the prevention of numerous global accesses) will be conserved.

Further Development of the Read Mapper Prototype. The experiments in chapter 6 showed the potential of MAROSE as a powerful high-throughput read mapper. We argue that it is worthwhile pursuing its development with the solutions for the current bottleneck phases: the multiplication reduction phase for exact filtering and approximate neighborhood matching phase for approximate filtering. Moreover, the prospective tests shows that MAROSE could be improved by using other techniques (spaced seeds, q-gram filtering). Finally, it could be also possible to develop a version of MAROSE targeted on longer reads.

Appendix A

Index Sizes

This appendix presents the details of the size of the indexes used in Chapter 5 and Chapter 6.

	$u = 3$			$u = 4$			$u = 6$		
	$\ell = 4$	$\ell = 8$	$\ell = 16$	$\ell = 4$	$\ell = 8$	$\ell = 16$	$\ell = 4$	$\ell = 8$	$\ell = 16$
Size (MB)	500	600	800	500	600	800	500	600	800
Create time (s)	19.83	30.96	58.91	25.11	36.07	60.73	29.31	40.40	65.22

Table A.1: Index size and creation time of general structures for the first 100 MBps of the human chromosome 1.

	$u = 4$, $\ell = 4$		$u = 4$, $\ell = 8$		$u = 4$, $\ell = 16$	
	Reduced	Indexes Block	Reduced	Indexed Block	Reduced	Indexed Block
start_list_nb	0.98 KB	0.98 KB	0.98 KB	0.98 KB	0.98 KB	0.98 KB
nb_block	0.25 MB	0.31 MB	47.46 MB	58.38 MB	347.07 MB	426.9 MB
start_list_nb_pos	0.98 KB	0.98 KB	0.98 KB	0.98 KB	0.98 KB	0.98 KB
nb_pos_block	0.25 MB	0.31 MB	47.46 MB	58.38 MB	347.07 MB	426.9 MB
start_list_pos	0.98 KB	0.98 KB	0.98 KB	0.98 KB	0.98 KB	0.98 KB
pos_block	400 MB	400 MB	400 MB	400 MB	400 MB	400 MB
start_list_ph	-	3.92 KB	-	3.92 KB	-	3.92 KB
ph_block	-	0.077 MB	-	14.59 MB	-	106.73 MB
Total Size (MB)	400.5	400.7	494.9	531.4	1094.1	1360.5
Create Time (s)	59.64	60.47	73.00	84.06	97.92	197.00

Table A.2: Index size and creation time of reduced structure and indexed block structure for the first 100 MBps of the human chromosome 1.

Sequence	Chro-mosome	Size (Mbps)	Segment number
1	10	125.53	3
2	11	125.06	3
3	12	124.27	3
4	13	91.13	2
5	14	84.20	2
6	15	77.57	2
7	16	75.23	2
8	17	74.20	2
9	18	71.20	2
10	19	53.20	2
11	1	214.58	5
12	20	56.75	2
13	21	32.59	1

Sequence	Chro-mosome	Size (Mbps)	Segment number
14	22	33.24	1
15	2	226.70	5
16	3	185.69	4
17	4	178.62	4
18	5	169.47	4
19	6	159.53	4
20	7	147.77	3
21	8	136.01	3
22	9	114.58	3
23	M	0.02	1
24	X	144.06	3
25	Y	24.46	1

Table A.3: List of sequences in the human genome bank ($\mathcal{H}_{ref}$) from the assembly 37.1 of the United States National Center for Biotechnology (NCBI). As the sequences are divided into smaller segments whose maximum size are 50 Mbps, thus there are totally 65 segments.

	$\mathcal{IH}_{U8L16}$ ($u = 8$, $\ell = 16$)		$\mathcal{IH}_{U8L16}$ ($u = 8$, $\ell = 7$)	
	Reduced	Indexes Block	Reduced	Indexed Block
start_list_nb	16.75	16.75	16.75	16.75
nb_block	9930.68	12264.40	8408.37	10391.96
start_list_nb_pos	16.75	16.75	16.75	16.75
nb_pos_block	9947.23	12280.95	8424.92	10408.51
start_list_pos	16.75	16.75	16.75	16.75
pos_block	10902.53	10902.53	10902.53	10902.53
start_list_ph	-	67	-	67
ph_block	-	3074.40	-	2606.30
Total (MB)	30830.69	38639.54	27786.07	34376.30
Create Time	43m21s	91m21s	34m55s	82m38s

Table A.4: Index size and creation time of reduced structure and indexed block structure for the human genome bank ($\mathcal{H}_{ref}$) from the assembly 37.1 of the United States National Center for Biotechnology (NCBI).

Bibliography

[Aji et al., 2010] Aji, A. M., Zhang, L., and Feng, W.-c. (2010). GPU-RMAP: Accelerating Short-Read Mapping on Graphics Processors. In *Proceedings of the IEEE 13th International Conference on Computational Science and Engineering (CSE)*, pages 168 – 175.

[Alcantara, 2011] Alcantara, D. A. F. (2011). *Efficient Hash Tables on the GPU*. PhD thesis, University of California Davis. Supervised by N. Amenta.

[Altschul et al., 1990] Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. (1990). Basic local alignment search tool. *J Mol Biol.*, 215(3):403–413.

[AMD Inc,] AMD Inc. Opencl zone. http://developer.amd.com/zones/OpenCLZone/Pages/default.aspx.

[AMD Inc, 2011a] AMD Inc (2011a). AMD Accelerated Parallel Processing, OpenCL Programming Guide. revision 1.2f.

[AMD Inc, 2011b] AMD Inc (2011b). Evergreen Family Instruction Set Architecture Instructions and Microcode. Reference Guide, revision 1.0e.

[AMD Inc, 2011c] AMD Inc (2011c). OpenCL and the AMD APP SDK. online white paper, realesed by the AMD Staff, http://developer.amd.com/documentation/articles/pages/OpenCL-and-the-AMD-APP-SDK.aspx.

[Apple Inc,] Apple Inc. New Technologies in Snow Leopard. http://www.apple.com/macosx/technology.

[Asanovic et al., 2006] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. (2006). The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

[Asanovic et al., 2009] Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., and Yelick, K. (2009). A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67.

[Baeza-Yates and Gonnet, 1989] Baeza-Yates, R. A. and Gonnet, G. H. (1989). A new approach to text searching. *SIGIR Forum*, 23(3-4):7.

[Bailey and Elkan, 1993] Bailey, T. L. and Elkan, C. (1993). Unsupervised learning of multiple motifs in biopolymers using expectation maximization. In *Machine Learning*, pages 51–80.

[Bailey and Elkan, 1994] Bailey, T. L. and Elkan, C. (1994). Fitting a mixture model by expectation maximization to discover motifs in biopolymers. In *Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology*, pages 28 – 36.

[Banakar et al., 2002] Banakar, R., Steinke, S., Lee, B.-S., Balakrishnan, M., and Marwedel, P. (2002). Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, CODES '02, pages 73–78, New York, NY, USA. ACM.

[Bao et al., 2011] Bao, S., Jiang, R., Kwan, W., Wang, B., Ma, X., and Song, Y.-Q. (2011). Evaluation of next-generation sequencing software in mapping and assembly. *Journal of Human Genetics*, 56:406 – 414.

[Bentley, 2006] Bentley, D. R. (2006). Whole-genome re-sequencing. *Current Opinion in Genetics & Development*, 16(6):545 – 552. Genomes and evolution.

[Blazewicz et al., 2011] Blazewicz, J., Frohmberg, W., Kierzynka, M., Pesch, E., and Wojciechowski, P. (2011). Protein alignment algorithms with an efficient backtracking routine on multiple GPUs. *BMC Bioinformatics*, 12(1):181.

[Blom et al., 2011] Blom, J., Jakobi, T., Doppmeier, D., Jaenicke, S., Kalinowski, J., Stoye, J., and Goesmann, A. (2011). Exact and complete short-read alignment to microbial genomes using Graphics Processing Unit programming. *Bioinformatics*, 27 (10):1351 – 1358.

[Bloom, 1970] Bloom, B. (1970). Space-time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426.

[Blythe, 2006] Blythe, D. (2006). The Direct3D 10 system. volume 25, pages 724–734, New York, NY, USA. ACM.

[Botelho, 2008] Botelho, F. C. (2008). *Near-Optimal Space Perfect Hashing Algorithms*. PhD thesis, Department of Computer Science, Federal University of Minas Gerais. Supervised by N. Ziviani.

[Botelho et al.,] Botelho, F. C., Reis, D. d. C., Belazzougui, D., and Ziviani, N. CMPH - C Minimal Perfect Hashing Library. online website, last updated: 09 June 2012, http://cmph.sourceforge.net/.

[Brown, 2008] Brown, D. G. (2008). *Bioinformatics Algorithms: Techniques and Applications*, chapter A survey of seeding for sequence alignment, pages 126–152.

[Buck et al., 2004] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. (2004). Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786.

[Bustamam et al., 2010a] Bustamam, A., Burrage, K., and Hamilton, N. (2010a). Fast parallel markov clustering in bioinformatics using massively parallel graphics processing unit computing. In *Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on*, pages 116 –125.

[Bustamam et al., 2010b] Bustamam, A., Burrage, K., and Hamilton, N. (2010b). A gpu implementation of fast parallel markov clustering in bioinformatics using ellpack-r sparse data format. In *Advances in Computing, Control and Telecommunication Technologies (ACT), 2010 Second International Conference on*, pages 173 –175.

[Campagna et al., 2009] Campagna, D., Albiero, A., Bilardi, A., Caniato, E., Forcato, C., Manavski, S., Vitulo, N., and Valle, G. (2009). PASS: a program to align short sequences. *Bioinformatics*, 25(7):967 – 968.

[Carter and Wegman, 1977] Carter, J. L. and Wegman, M. N. (1977). Universal classes of hash functions (Extended Abstract). In *Proceedings of the ninth annual ACM symposium on Theory of computing*, STOC '77, pages 106–112, New York, NY, USA. ACM.

[Chalkidis et al., 2011] Chalkidis, G., Nagasaki, M., and Miyano, S. (2011). High Performance Hybrid Functional Petri Net Simulations of Biological Pathway Models on CUDA. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 8(6):1545–1556.

[Chang et al., 2010] Chang, D.-J., Kimmer, C., and Ouyang, M. (2010). Accelerating the Nussinov RNA folding algorithm with CUDA/GPU. In *Signal Processing and Information Technology (ISSPIT), 2010 IEEE International Symposium on*, pages 120 –125.

[Charalambous et al., 2005] Charalambous, M., Trancoso, P., and Stamatakis, A. (2005). Initial Experiences Porting a Bioinformatics Application to a Graphics Processor. *Advances in Informatics*, pages 415–425.

[Che et al., 2008] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., and Skadron, K. (2008). A performance study of general-purpose applications on graphics processors using CUDA. *J. Parallel Distrib. Comput.*, 68(10):1370–1380.

[Chen and Jiang, 2011] Chen, S. and Jiang, H. (2011). An exact matching approach for high throughput sequencing based on bwt and gpus. In *Computational Science and Engineering (CSE), 2011 IEEE 14th International Conference on*, pages 173 –180.

[CIBIV,] CIBIV. NextGenMap. NextGenMap website of Center for Integrative Bioinformatics Vienna, http://www.cibiv.at/software/ngm/NGM/.

[CIPF,] CIPF, B. D. BWT-GPU, website.

[Czech et al., 1992] Czech, Z. J., Havas, G., and Majewski, B. S. (1992). An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43(5):257 – 264.

[Dayhoff et al., 1978] Dayhoff, M., Schwartz, R., and Orcutt, B. (1978). A model of evolutionary change in proteins. *Atlas of Protein Sequence and Structure*, 5(3):345–352.

[Delcher et al., 1999] Delcher, A. L., Kasif, S., Fleischmann, R. D., Peterson, J., White, O., and Salzberg, S. L. (1999). Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376.

[Delcher et al., 2002] Delcher, A. L., Phillippy, A., Carlton, J., and Salzberg, S. L. (2002). Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Research*, 30(11):2478–2483.

[Dohi et al., 2010] Dohi, K., Benkridt, K., Ling, C., Hamada, T., and Shibata, Y. (2010). Highly efficient mapping of the Smith-Waterman algorithm on CUDA-compatible GPUs. In *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on*, pages 29 –36.

[Durbin et al., 1998] Durbin, R., Eddy, S. R., Krogh, A., and Mitchison, G. (1998). *Biological sequence analysis: Probabilistic models of proteins and nucleic acids*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1st edition.

[Dynerman et al., 2009] Dynerman, D., Butzlaff, E., and Mitchell, J. C. (2009). CUSA and CUDE: GPU-Accelerated Methods for Estimating Solvent Accessible Surface Area and Desolvation. *Journal of Computational Biology*, 16(4):523–537.

[Eaves and Gao, 2009] Eaves, H. L. and Gao, Y. (2009). MOM: maximum oligonucleotide mapping. *Bioinformatics*, 25(7):969 – 970.

[Encarnaijao et al., 2011] Encarnaijao, G., Sebastiao, N., and Roma, N. (2011). Advantages and GPU implementation of high-performance indexed DNA search based on suffix arrays. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 49 –55.

[Enright et al., 2002] Enright, A. J., Van Dongen, S., and Ouzounis, C. A. (2002). An efficient algorithm for large-scale detection of protein families. *Nucleic Acids Research*, 30(7):1575–1584.

[Falk et al., 2011] Falk, M., Ott, M., Ertl, T., Klann, M., and Koeppl, H. (2011). Parallelized agent-based simulation on CPU and graphics hardware for spatial and stochastic models in biology. In *Proceedings of the 9th International Conference on Computational Methods in Systems Biology*, CMSB '11, pages 73–82, New York, NY, USA. ACM.

[Farivar et al., 2012] Farivar, R., Kharbanda, H., Venkatraman, S., and Campbell, R. H. (2012). An algorithm for fast edit distance computation on gpus. In *InPar 2012 Innovative Parallel Computing: Foundations & Applications of GPU, Manycore, and Heterogeneous Systems*, San Jose, California. IEEE, IEEE.

[Farrar, 2007] Farrar, M. (2007). Striped Smith–Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23(2):156–161.

[Felsenstein, 2010] Felsenstein, J. (2010). PHYLIP (Phylogeny Inference Package). Version 3.5c, http://cmgm.stanford.edu/phylip/index.html.

[Ferragina and Manzini, 2000] Ferragina, P. and Manzini, G. (2000). Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390 –398.

[Friedrichs et al., 2009] Friedrichs, M. S., Eastman, P., Vaidyanathan, V., Houston, M., Legrand, S., Beberg, A. L., Ensign, D. L., Bruns, C. M., and Pande, V. S. (2009). Accelerating molecular dynamic simulation on graphics processing units. *Journal of Computational Chemistry*, 30(6):864–872.

[Ganesan et al., 2010] Ganesan, N., Chamberlain, R. D., Buhler, J., and Taufer, M. (2010). Accelerating HMMER on GPUs by implementing hybrid data and task parallelism. In *Proceedings of the First ACM International Conference on Bioinformatics and Computational Biology*, BCB '10, pages 418–421, New York, NY, USA. ACM.

[Gaster et al., 2011] Gaster, B., Howes, L., Kaeli, D. R., Mistry, P., and Schaa, D. (2011). *Heterogeneous Computing with OpenCL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

[Giraud and Varré, 2010] Giraud, M. and Varré, J.-S. (2010). Parallel position weight matrices algorithms. *Parallel Computing*, 37 (8).

[Gummaraju et al., 2010] Gummaraju, J., Morichetti, L., Houston, M., Sander, B., Gaster, B. R., and Zheng, B. (2010). Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In *Parallel architectures and compilation techniques (PACT 10), series = PACT '10*, pages 205–216.

[Hains et al., 2011] Hains, D., Cashero, Z., Ottenberg, M., Bohm, W., and Rajopadhye, S. (2011). Improving CUDASW++, a Parallelization of Smith-Waterman for CUDA Enabled Devices. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 490 –501.

[Harris, 2012] Harris, M. (2012). CUDA 5 and Beyond. GPU Technology Conference.

[Hasan et al., 2011a] Hasan, L., Kentie, M., and Al-Ars, Z. (2011a). DOPA: GPU-based protein alignment using database and memory access optimizations. *BMC Research Notes*, 4(1):261.

[Hasan et al., 2011b] Hasan, L., Kentie, M., and Al-Ars, Z. (2011b). GPU-accelerated protein sequence alignment. In *Engineering in Medicine and Biology Society,EMBC, 2011 Annual International Conference of the IEEE*, pages 2442 –2446.

[Havas et al., 1994] Havas, G., Majewski, B. S., Wormald, N. C., and Czech, Z. J. (1994). Graphs, Hypergraphs and Hashing. In *Proceedings of the 19th International Workshop on Graph-Theoretic Concepts in Computer Science*, WG '93, pages 153–165, London, UK, UK. Springer-Verlag.

[Henikoff and Henikoff, 1992] Henikoff, S. and Henikoff, J. G. (1992). Amino acid substitution matrices from protein blocks. *Proc Natl Acad Sci U S A.*, 89(22):10915–10919.

[Hoffmann et al., 2009] Hoffmann, S., Otto, C., Kurtz, S., Sharma, C. M., Khaitovich, P., Vogel, J., Stadler, P. F., and Hackermuller, J. (2009). Fast Mapping of Short Sequences with Mismatches, Insertions and Deletions Using Index Structures. *PLOS Computational Biology*, 5(9):e1000502.

[Holub, 2002] Holub, J. (2002). Bit Parallelism - NFA Simulation. In *Revised Papers from the 6th International Conference on Implementation and Application of Automata*, CIAA '01, pages 149–160, London, UK, UK. Springer-Verlag.

[Homer et al., 2009] Homer, N., Merriman, B., and Nelson, S. F. (2009). BFAST: An Alignment Tool for Large Scale Genome Resequencing. *PLoS ONE*, 4(11):e7767.

[Horn et al., 2005] Horn, D. R., Houston, M., and Hanrahan, P. (2005). Clawhmmer: A streaming hmmer-search implementation. In *SC*, page 11.

[Horner et al., 2010] Horner, D. S., Pavesi, G., Castrignanò, T., D'Onorio De Meo, P., Liuni, S., Sammeth, M., Picardi, E., and Pesole, G. (2010). Bioinformatics approaches for genomics and post genomics applications of next-generation sequencing. *Briefings in Bioinformatics*, 11(2):181–197.

[Hung et al., 2011] Hung, L.-H., Guerquin, M., and Samudrala, R. (2011). GPU-Q-J, a fast method for calculating root mean square deviation (RMSD) after optimal superposition. *BMC Research Notes*, 4(1):97.

[IBM,] IBM. OpenCL - The open standard for parallel programming of heterogeneous systems. http://www.alphaworks.ibm.com/tech/opencl.

[Intel Corp, a] Intel Corp. Intel OpenCL SDK. http://software.intel.com/en-us/articles/opencl-sdk/.

[Intel Corp, b] Intel Corp. Intel OpenCL SDK, User's Guide. revision 1.3.

[Intel Corp, 2002] Intel Corp (2002). Hyper-Threading Technology. *Intel Technology Journal*, 06(01).

[Jacob et al., 2008] Jacob, A., Lancaster, J., Buhler, J., Harris, B., and Chamberlain, R. D. (2008). Mercury BLASTP: Accelerating Protein Sequence Alignment. *ACM Trans. Reconfigurable Technol. Syst.*, 1(2):9:1–9:44.

[Jang et al., 2011] Jang, B., Schaa, D., Mistry, P., and Kaeli, D. (2011). Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures. *IEEE Trans. Parallel Distrib. Syst.*, 22(1):105–118.

[Jenkins, 1997] Jenkins, B. (1997). Algorithm alley: Hash functions. *Dr. Dobb's Journal of Software Tools*, 22(9). online version, http://www.drdobbs.com/database/algorithm-alley/184410284.

[Jiang and Wong, 2008] Jiang, H. and Wong, H. (2008). SeqMap: mapping massive amount of oligonucleotides to the genome. *Bioinformatics*, 24(20):2395 – 2396.

[Jiang et al., 2009] Jiang, R., Zeng, F., Zhang, W., Wu, X., and Yu, Z. (2009). Accelerating genome-wide association studies using cuda compatible graphics processing units. In *Bioinformatics, Systems Biology and Intelligent Computing, 2009. IJCBS '09. International Joint Conference on*, pages 70 –76.

[Kent, 2002] Kent, W. J. (2002). BLAT—The BLAST-Like Alignment Tool. *Genome Research*, 12 (4):656–664.

[Khronos Group, 2008] Khronos Group (2008). OpenCL - The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/.

[Khronos Group, 2010] Khronos Group (2010). The OpenCL Specification, version 1.1. Khronos OpenCL working group, Editor: Aaftab Munshi, Document revision 36.

[Klus et al., 2012] Klus, P., Lam, S., Lyberg, D., Cheung, M., Pullan, G., McFarlane, I., Yeo, G., and Lam, B. (2012). BarraCUDA - a fast short read sequence aligner using graphics processing units. *BMC Research Notes*, 5(1):27.

[Korf et al., 2003] Korf, I., Yandell, M., and Bedell, J. (2003). *BLAST*. O'Reilly.

[Kurtz et al., 2004] Kurtz, S., Phillippy, A., Delcher, A. L., Smoot, M., Shumway, M., Antonescu, C., and Salzberg, S. L. (2004). Versatile and open software for comparing large genomes. *Genome Biology*, 5(R12).

[Kwon et al., 2010] Kwon, M.-S., Kim, K., Lee, S., and Park, T. (2010). cugwam: Genome-wide association multifactor dimensionality reduction using cuda-enabled high-performance graphics processing unit. In *Bioinformatics and Biomedicine Workshops (BIBMW), 2010 IEEE International Conference on*, pages 336 –340.

[Langmead and Salzberg, 2012] Langmead, B. and Salzberg, S. L. (2012). Fast gapped-read alignment with Bowtie 2. *Nature Mathods*, 9:357–359.

[Langmead et al., 2009] Langmead, B., Trapnell, C., Pop, M., and Salzberg, S. L. (2009). Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10 (3).

[Larkin et al., 2007] Larkin, M., Blackshields, G., Brown, N., Chenna, R., McGettigan, P., McWilliam, H., Valentin, F., Wallace, I., Wilm, A., Lopez, R., Thompson, J., 3, T. G., and Higgins, D. (2007). Clustal W and Clustal X version 2.0. *Bioinformatics*, 23(21):2947–2948.

[Lavenier and Giraud, 2005] Lavenier, D. and Giraud, M. (2005). *Reconfigurable Computing*, chapter Bioinformatics Applications. Springer.

[Li and Durbin, 2009] Li, H. and Durbin, R. (2009). Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25 (14):1954 – 1960.

[Li and Durbin, 2010] Li, H. and Durbin, R. (2010). Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26 (5):589 – 595.

[Li and Hormer, 2010] Li, H. and Hormer, N. (2010). A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11 (5):473 – 483.

[Li et al., 2011] Li, H., Ni, B., Wong, M.-H., and Leung, K.-S. (2011). A fast CUDA implementation of agrep algorithm for approximate nucleotide sequence matching. In *IEEE Symposium on Application Specific Processors (SASP 2011)*, pages 74 –77.

[Li et al., 2008a] Li, H., Ruan, J., and Richard, D. (2008a). Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Research*, 18 (11):1851–1858.

[Li et al., 2008b] Li, R., Li, Y., Kristiansen, K., and Wang, J. (2008b). SOAP: short oligonucleotide alignment program. *Bioinformatics*, 24 (5):713–714.

[Li et al., 2009] Li, R., Yu, C., Lam, T.-W., Yiu, S.-M., Kristiansen, K., and Wang, J. (2009). SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25 (15):1966 – 1967.

[Ligowski and Rudnicki, 2009] Ligowski, L. and Rudnicki, W. (2009). An efficient implementation of Smith-Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. In *IEEE International Workshop on High Performance Computational Biology (HiCOMB 2009)*.

[Lin et al., 2008] Lin, H., Zhang, Z., Zhang, M. Q., Ma, B., and Li, M. (2008). ZOOM! Zillions of oligos mapped. *Bioinformatics*, 24 (21):2431 – 2437.

[Lin et al., 2011] Lin, M., Chamberlain, R., Buhler, J., and Franklin, M. (2011). Bloom Filter Performance on Graphics Engines. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 522 –531.

[Lindholm et al., 2008] Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. (2008). NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55.

[Ling and Benkrid, 2010] Ling, C. and Benkrid, K. (2010). Design and implementation of a CUDA-compatible GPU-based core for gapped BLAST algorithm. *Procedia Computer Science*, 1(1):495 – 504. ICCS 2010.

[Ling et al., 2011] Ling, C., Benkrid, K., and Erdogan, A. (2011). High performance Intra-task parallelization of Multiple Sequence Alignments on CUDA-compatible GPUs. In *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, pages 360 –366.

[Ling et al., 2009] Ling, C., Benkrid, K., and Hamada, T. (2009). A parameterisable and scalable Smith-Waterman algorithm implementation on CUDA-compatible GPUs. In *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, pages 94 –100.

[Lipman and Pearson, 1988] Lipman, D. and Pearson, W. (1988). Improved tools for biological sequence comparison. *Proc. Natl Acad. Sci. USA*, 85(8):2444–2448.

[Litwin, 1980] Litwin, W. (1980). Linear hashing: a new tool for file and table addressing. In *Proceedings of the sixth international conference on Very Large Data Bases - Volume 6*, VLDB '80, pages 212–223. VLDB Endowment.

[Liu et al., 2011a] Liu, C.-M., Lam, T.-W., Wong, T., Wu, E., Yiu, S.-M., Li, Z., Luo, R., Wang, B., Yu, C., Chu, X., Zhao, K., and Li, R. (2011a). SOAP3: GPU-based Compressed Indexing and Ultra-fast Parallel Alignment of Short Reads.

[Liu, 2010] Liu, W. (2010). CUDA-BLASTP on Tesla GPUs. http://www.nvidia.com/object/blastp_on_tesla.html.

[Liu et al., 2011b] Liu, W., Schmidt, B., Liu, Y., Voss, G., and Mueller-Wittig, W. (2011b). Mapping of BLASTP Algorithm onto GPU Clusters. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 236 –243.

[Liu et al., 2011c] Liu, W., Schmidt, B., and Muller-Wittig, W. (2011c). CUDA-BLASTP: Accelerating BLASTP on CUDA-Enabled Graphics Hardware. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 8:1678–1684.

[Liu et al., 2006a] Liu, W., Schmidt, B., Voss, G., and Müller-Wittig, W. (2006a). GPU-ClustalW: Using graphics hardware to accelerate multiple sequence alignment. In *IEEE International Conference on High Performance Computing (HiPC 2006)*, volume 4297 of *Lecture Notes in Computer Science (LNCS)*, pages 363–374.

[Liu et al., 2006b] Liu, W., Schmidt, B., Voss, G., Schroder, A., and Muller-Wittig, W. (2006b). Bio-sequence database scanning on a GPU. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 8 pp.

[Liu et al., 2009a] Liu, Y., Maskell, D., and Schmidt, B. (2009a). CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes*, 2(1):73.

[Liu and Schmidt, 2012] Liu, Y. and Schmidt, B. (2012). Long read alignment based on maximal exact match seeds. *Bioinformatics*, 28(18):i318–i324.

[Liu et al., 2010a] Liu, Y., Schmidt, B., Liu, W., and Maskell, D. L. (2010a). CUDA-MEME: Accelerating motif discovery in biological sequences using CUDA-enabled graphics processing units. *Pattern Recogn. Lett.*, 31(14):2170–2177.

[Liu et al., 2009b] Liu, Y., Schmidt, B., and Maskell, D. (2009b). MSA-CUDA: Multiple Sequence Alignment on Graphics Processing Units with CUDA. In *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, pages 121 –128.

[Liu et al., 2009c] Liu, Y., Schmidt, B., and Maskell, D. (2009c). Parallel reconstruction of neighbor-joining trees for large multiple sequence alignments using cuda. In *IEEE International Workshop on High Performance Computational Biology (HiCOMB 2009)*.

[Liu et al., 2010b] Liu, Y., Schmidt, B., and Maskell, D. (2010b). Cudasw++2.0: enhanced smith-waterman protein database search on cuda-enabled gpus based on simt and virtualized simd abstractions. *BMC Research Notes*, 3(1):93.

[Liu et al., 2011d] Liu, Y., Schmidt, B., and Maskell, D. (2011d). An Ultrafast Scalable Many-Core Motif Discovery Algorithm for Multiple GPUs. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 428 –434.

[Liu et al., 2011e] Liu, Y., Schmidt, B., and Maskell, D. L. (2011e). DecGPU: distributed error correction on massively parallel graphics processing units using CUDA and MPI. *BMC Bioinformatics*, 12.

[Liu et al., 2012a] Liu, Y., Schmidt, B., and Maskell, D. L. (2012a). CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows-Wheeler transform. *Bioinformatics*.

[Liu et al., 2012b] Liu, Y., Xi, J., Lai, Z., and Huang, D. (2012b). Batch Records Insertion into Multi-dimensional Linear Dynamic Hashing Table on GPU. *Journal of Computational Information Systems*, 8(10):4293–4301.

[Ma et al., 2002] Ma, B., Tromp, J., and Li, M. (2002). PatternHunter: faster and more sensitive homology search. *BIOINFORMATICS*, 18(3):440–445.

[Mahmood and Rangwala, 2011] Mahmood, S. and Rangwala, H. (2011). GPU-Euler: Sequence Assembly Using GPGPU. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 153 –160.

[Majewski et al., 1996] Majewski, B. S., Wormald, N. C., Havas, G., and Czech, Z. J. (1996). A family of perfect hashing methods. *The Computer Journal*, 39(6):547–554.

[Manavski and Valle, 2008] Manavski, S. and Valle, G. (2008). CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2):S10.

[Matthews et al., 1999] Matthews, D. H., Sabrina, J., Zuker, M., and Turner, D. H. (1999). Expanded sequence dependence of thermodynamic parameters improves prediction of rna secondary structure. *Journal of Molecular Biology*, 288:911–940.

[Moore, 1965] Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8).

[Moore, 1975] Moore, G. E. (1975). Progress in digital integrated electronics. In *Technical Digest 1975. International Electron Devices Meeting, IEEE*, pages 11–13.

[Munekawa et al., 2008] Munekawa, Y., Ino, F., and Hagihara, K. (2008). Design and implementation of the smith-waterman algorithm on the cuda-compatible gpu. In *BioInformatics and BioEngineering, 2008. BIBE 2008. 8th IEEE International Conference on*, pages 1 –6.

[Navarro and Raffinot, 2002] Navarro, G. and Raffinot, M. (2002). *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press. ISBN 0-521-81307-7. 280 pages.

[Needleman and Wunsch, 1970] Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the seach for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):433 – 453.

[Nguyen, 2008] Nguyen, V. H. (2008). *Parallel computing for intensive comparison of genomic sequences*. PhD thesis, École Doctoral Matisse, Université de Rennes 1. Supervised by D. Lavenier.

[Nguyen and Lavenier, 2008] Nguyen, V. H. and Lavenier, D. (2008). Speeding up Subset Seed Algorithm for Intensive Protein Sequence Comparison. In *6th IEEE International Conference on research, innovation & vision for the future*, Ho Chi Minh Ville, Viet Nam.

[Nguyen and Lavenier, 2009] Nguyen, V.-H. and Lavenier, D. (2009). PLAST: parallel local alignment search tool for database comparison. *BMC Bioinformatics*, 10:329.

[Ning et al., 2001] Ning, Z., Cox, A. J., and Mullikin, J. C. (2001). SSAHA: a fast search method for large DNA databases. *Genome Research*, 11 (10):1725 – 1729.

[NRC, 2007] NRC (2007). *The New Science of Metagenomics: Revealing the Secrets of Our Microbial Planet*. The National Academies Press. Committee on Metagenomics: Challenges and Functional Applications, United States National Research Council. Free download version at http://www.nap.edu/catalog.php?record_id=11902.

[Nussinov et al., 1978] Nussinov, R., Pieczenik, G., Griggs, J., and Kleitman, D. (1978). Algorithms for loop matchings. *SIAM Journal of Applied Mathematics*, 35:68–82.

[NVIDIA Corp, a] NVIDIA Corp. NVIDIA CUDA Home,. online website, http://www.nvidia.com/object/cuda_home_new.html.

[NVIDIA Corp, b] NVIDIA Corp. NVIDIA GF100, World's fastest GPU delivering great gaming performance with true geometric system. white paper, version 1.5.

[NVIDIA Corp, c] NVIDIA Corp. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. white paper, version 1.1.

[NVIDIA Corp, d] NVIDIA Corp. OpenCL. http://developer.nvidia.com/opencl.

[NVIDIA Corp, 2006] NVIDIA Corp (2006). NVIDIA GeForce 8800 GPU Architecture Overview. Technical Brief, version 1.1.

[NVIDIA Corp, 2009a] NVIDIA Corp (2009a). NVIDIA CUDA Architecture, Introduction and Overview. version 1.1.

[NVIDIA Corp, 2009b] NVIDIA Corp (2009b). NVIDIA OpenCL best practice guide. version 1.0.

[NVIDIA Corp, 2012] NVIDIA Corp (2012). OpenCL, Programming Guide for the CUDA Architecture. version 4.2.

[Ortiz et al., 2002] Ortiz, A. R., Strauss, C. E., and Olmea, O. (2002). MAMMOTH (Matching molecular models obtained from theory): An automated method for model comparison. *Protein Science*, 11(11):2606–2621.

[Owens et al., 2008] Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., and Phillips, J. (2008). Gpu computing. *Proceedings of the IEEE*, 96(5):879 –899.

[Pandit and Skolnick, 2008] Pandit, S. and Skolnick, J. (2008). Fr-TM-align: a new protein structural alignment method based on fragment alignments and the TM-score. *BMC Bioinformatics*, 9(1):531.

[Pang et al., 2012] Pang, B., Zhao, N., Becchi, M., Korkin, D., and Shyu, C.-R. (2012). Accelerating large-scale protein structure alignments with graphics processing units. *BMC Research Notes*, 5(1):116.

[Peterlongo et al., 2008] Peterlongo, P., Noé, L., Lavenier, D., Nguyen, V. H., Kucherov, G., and Giraud, M. (2008). Optimal neighborhood indexing for protein similarity search. *BMC Bioinformatics*, 9(534).

[Pevzner et al., 2001] Pevzner, P. A., Tang, H., and Waterman, M. S. (2001). A new approach to fragment assembly in DNA sequencing. In *Proceedings of the fifth annual international conference on Computational biology*, RECOMB '01, pages 256–267, New York, NY, USA. ACM.

[Pisanti et al., 2011] Pisanti, N., Giraud, M., and Peterlongo, P. (2011). *Algorithms in Computational Molecular Biology: Techniques, Approaches and Applications*, chapter Filters and Seeds Approaches for Fast Homology Searches in Large Datasets.

[Razmyslovich et al., 2010] Razmyslovich, D., Marcus, G., Gipp, M., Zapatka, M., and Szillus, A. (2010). Implementation of smith-waterman algorithm in opencl for gpus. In *Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on*, pages 48 –56.

[Rice et al., 2000] Rice, P., Longden, I., and Bleasby, A. (2000). EMBOSS: The European Molecular Biology Open Software Suite. *Trends in Genetics*, 16(6):276 – 277.

[Ritchie and Venkatraman, 2010] Ritchie, D. W. and Venkatraman, V. (2010). Ultra-fast FFT protein docking on graphics processors. *Bioinformatics*, 26(19):2398–2405.

[Ritchie et al., 2001] Ritchie, M., Hahn, L., Roodi, N., Bailey, L., Dupont, W., Parl, F., and Moore, J. (2001). Multifactor-dimensionality reduction reveals high-order interactions among estrogen-metabolism genes in sporadic breast cancer. *Am J Hum Genet*, 69:138–147.

[Rivals et al., 2009] Rivals, E., Salmela, S., Kiiskinen, P., Kalsi, P., and Tarhio, J. (2009). MPSCAN: fast localisation of multiple reads in genomes. In *Proc. 9th Workshop on Algorithms in Bioinformatics, Lecture Notes in BioInformatics (LNBI)*, volume 5724, pages 246 – 260. Springer-Verlag.

[Rizk and Lavenier, 2009] Rizk, G. and Lavenier, D. (2009). GPU accelerated RNA folding algorithm. In *Using Emerging Parallel Architectures for Computational Science / International Conference on Computational Science (ICCS 2009)*.

[Rizk and Lavenier, 2010] Rizk, G. and Lavenier, D. (2010). GASSST: Global Alignement Short Sequence Search Tool. *Bioinformatics*, 26(20 2010):2534–2540.

[Roberts et al., 2009] Roberts, E., Stone, J., Sepulveda, L., Hwu, W.-M., and Luthey-Schulten, Z. (2009). Long time-scale simulations of in vivo diffusion using GPU hardware. In *IEEE International Workshop on High Performance Computational Biology (HiCOMB 2009)*.

[Rumble et al., 2009] Rumble, S. M., Lacroute, P., Dalca, A. V., Fiume, M., Sidow, A., and Brudno, M. (2009). SHRiMP: Accurate Mapping of Short Color-space Reads. *PLoS Comput Biol*, 5(5):e1000386.

[Schatz, 2009] Schatz, M. C. (2009). CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363 – 1369.

[Schatz et al., 2007] Schatz, M. C., Trapnell, C., Delcher, A. L., and Varshney, A. (2007). High-throughput sequence alignment using Graphics Processing Units. *BMC Bioinformatics*.

[Schbath et al., 2012] Schbath, S., Martin, V., Zytnicki, M., Fayolle, J., Loux, V., and Gibrat, J.-F. (2012). Mapping reads on a genomic sequence: an algorithmic overview and a practical comparative analysis. *Journal of Computational Biology*, 19(6):796 – 813.

[Semin, 2009] Semin, A. (2009). Inside Intel Nehalem Microarchitecture. In *NOTUR 2009, The 8th Annual Meeting on High Performance Computing and Infrastructure in Norway*.

[Shalf et al., 2009] Shalf, J., Asanovic, K., Patterson, D., Keutzer, K., Mattson, T., and Yelick, K. (2009). The Manycore Revolution: Will HPC Lead or Follow? *SciDAC Review*, 14.

[Shi et al., 2009] Shi, H., Schmidt, B., Liu, W., and Mueller-Wittig, W. (2009). Accelerating Error Correction in High-Throughput Short-Read DNA Sequencing Data with CUDA. In *IEEE International Workshop on High Performance Computational Biology (HiCOMB 2009)*.

[Shi et al., 2010] Shi, H., Schmidt, B., Liu, W., and Müller-Wittig, W. (2010). Quality-score guided error correction for short-read sequencing data using CUDA. *Procedia Computer Science*, 1(1):1129 – 1138.

[Simone and Therry, 2013] Simone, F. and Therry, L. (2013). The Exact Online String Matching Problem: a Review of the Most Recent Results. *ACM Computing Surveys*, 45(2). to appear.

[Sinnott-Armstrong et al., 2009] Sinnott-Armstrong, N., Greene, C., Cancare, F., and Moore, J. (2009). Accelerating epistasis analysis in human genetics with consumer graphics hardware. *BMC Research Notes*, 2(1):149.

[Siriwardena and Ranasinghe, 2010] Siriwardena, T. and Ranasinghe, D. (2010). Accelerating global sequence alignment using CUDA compatible multi-core GPU. In *Information and Automation for Sustainability (ICIAFs), 2010 5th International Conference on*, pages 201 –206.

[Smith et al., 2008] Smith, A. D., Xuan, Z., and Zhang, M. Q. (2008). Using quality scores and longer reads improves accuracy of Solexa read mapping. *BMC Bioinformatics*, 9.

[Smith and Waterman, 1981] Smith, T. F. and Waterman, M. S. (1981). Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197.

[Stamatakis, 2005] Stamatakis, A. (2005). An Efficient Program for Phylogenetic Inference Using Simulated Annealing. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 7 - Volume 08*, IPDPS '05, pages 198.2–, Washington, DC, USA. IEEE Computer Society.

[Stamatakis et al., 2005] Stamatakis, A., Ludwig, T., and Meier, H. (2005). RAxML-III: a fast program for maximum likelihood-based inference of large phylogenetic trees. *Bioinformatics*, 21(4):456–463.

[Steffen and Giegerich, 2005] Steffen, P. and Giegerich, R. (2005). Versatile and declarative dynamic programming using pair algebras. *BMC Bioinformatics*, 6(1).

[Steffen et al., 2009] Steffen, P., Giegerich, R., and Giraud, M. (2009). GPU parallelization of algebraic dynamic programming. In *Parallel Processing and Applied Mathematics / Parallel Biocomputing Conference (PPAM / PBC 09)*.

[Stivala et al., 2010] Stivala, A., Stuckey, P., and Wirth, A. (2010). Fast and accurate protein substructure searching with simulated annealing and GPUs. *BMC Bioinformatics*, 11(1):446.

[Striemer and Akoglu, 2009] Striemer, G. M. and Akoglu, A. (2009). Sequence alignment with GPU: Performance and design challenges. *Parallel and Distributed Processing Symposium, International*, 0:1–10.

[Suchard and Rambaut, 2009] Suchard, M. A. and Rambaut, A. (2009). Many-core algorithms for statistical phylogenetics. *Bioinformatics*, 25(11):1370–1376.

[Technologies,] Technologies, N. Novoalign.

[Thompson et al., 1994] Thompson, J. D., Higgins, D. G., and Gibson, T. J. (1994). CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22(22):4673–4680.

[Torres et al., 2012] Torres, J. S., Espert, I. B., Dominguez, A. T., Hernendez, V., Medina, I., Terraga, J., and Dopazo, J. (2012). Using gpus for the exact alignment of short-read genetic sequences by means of the burrows–wheeler transform. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 99(PrePrints).

[Tran et al., 2011] Tran, T. T., Giraud, M., and Varré, J.-S. (2011). Bit-parallel multiple pattern matching. In *Parallel Processing and Applied Mathematics / Parallel Biocomputing Conference (PPAM / PBC 11)*.

[Trapnell and Schatz, 2009] Trapnell, C. and Schatz, M. C. (2009). Optimizing data intensive GPGPU computations for DNA sequence alignment. *Parallel Computing*, 35(8–9):429 – 440.

[Van Dongen, 2008] Van Dongen, S. (2008). Graph Clustering Via a Discrete Uncoupling Process. *SIAM J. Matrix Anal. Appl.*, 30(1):121–141.

[Varré et al., 2011] Varré, J.-S., Schmidt, B., Janot, S., and Giraud, M. (2011). Manycore high-performance computing in bioinformatics. In Elnitski, L., Piontkivska, H., and Welch, L. R., editors, *Advances in Genomic Sequence Analysis and Pattern Discovery*, page chapter 8. World Scientific.

[Vouzis and Sahinidis, 2011] Vouzis, P. D. and Sahinidis, N. V. (2011). GPU-BLAST: using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, 27(2):182–188.

[Walters et al., 2009] Walters, J. P., Balu, V., Kompalli, S., and Chaudhary, V. (2009). Evaluating the use of gpus in liver image segmentation and hmmer database searches. *Parallel and Distributed Processing Symposium, International*, 0:1–12.

[Wang et al., 2009] Wang, W., Zhang, P., and Liu, X. (2009). Short read DNA fragment anchoring algorithm. *BMC Bioinformatics*, 10 (Suppl 1).

[Weiner, 1973] Weiner, P. (1973). Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (swat 1973)*, SWAT '73, pages 1–11, Washington, DC, USA. IEEE Computer Society.

[Wu and Manber, 1992a] Wu, S. and Manber, U. (1992a). Agrep - a fast approximate pattern-matching tool. In *In Proc. of USENIX Technical Conference*, pages 153–162.

[Wu and Manber, 1992b] Wu, S. and Manber, U. (1992b). Fast Text Searching Allowing Errors. *Communications of the ACM*, 35(10):83–91.

[Xiao et al., 2011] Xiao, S., Lin, H., and chun Feng, W. (2011). Accelerating Protein Sequence Search in a Heterogeneous Computing System. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1212 –1222.

[Ying et al., 2011] Ying, Z., Lin, X., See, S.-W., and Li, M. (2011). GPU-accelerated DNA Distance Matrix Computation. In *Chinagrid Conference (ChinaGrid), 2011 Sixth Annual*, pages 42 –47.

[Zhang et al., 2011] Zhang, Y., Peng, L., B., L., J.-K., P., and J., C. (2011). Architecture Comparisons between NVidia and ATI GPUs: Computational Parallelism and Data Communications. In *In Proceedings of The 2011 IEEE International Symposium on Workload Characterization (IISWC)*.

[Zhang and Skolnick, 2005] Zhang, Y. and Skolnick, J. (2005). TM-align: a protein structure alignment algorithm based on the TM-score. *Nucleic Acids Research*, 33(7):2302–2309.

[Zheng et al., 2011] Zheng, F., Xu, X., Yang, Y., He, S., and Zhang, Y. (2011). Accelerating biological sequence alignment algorithm on gpu with cuda. In *Computational and Information Sciences (ICCIS), 2011 International Conference on*, pages 18 –21.

[Zuker, 2003] Zuker, M. (2003). Mfold web server for nucleic acid folding and hybridization prediction. *Nucleic Acids Research*, 31(13):3406–3415.

List of Figures

List of Tables

Notations

		typical values
t	text (genome)	
$n = \|t\|$	text length	100 Mbps
p	pattern (query)	
$m = \|p\|$	pattern length	4, 8, 16
sq	seed and neighborhood, extracted from the pattern query	
$u = \|s\|$	seed length	4, 6, 8
$v = \|q\|$	neighborhood length	4, 8, 16
$\ell = v + e$	neighborhood length, in the SeedBlock (Hamming)	
$\ell = v$	neighborhood length, in the SeedBlock (Levenshtein)	
$\mathcal{N}$	number of neighborhoods in the SeedBlock	$\sim n/4^u$
q	neighborhood or fixed-length pattern of size v	
$R^{[j]}$	bit vector after processing j characters of the text	
e	number of errors	0, 1, 2
$\Pi_e(q)$	set of degenerated patterns with $\leq e$ errors	
w	size of the machine word	32, 64
$h = \lfloor w/\ell \rfloor$	neighbors per machine word	4, 2, 1
$\mathrm{T}_{\mathrm{dev}}$	device time	
nps	neighborhood per second	

Résumé

Rechercher les similarités entre séquences est une opération fondamentale en bioinformatique, que cela soit pour étudier des questions biologiques ou bien pour traiter les données issues de séquenceurs haut-débit. Il y a un vrai besoin d'algorithmes capables de traiter des millions de séquences rapidement. Pour trouver des similarités approchées, on peut tout d'abord considérer de petits mots exacts présents dans les deux séquences, les *graines*, puis essayer d'étendre les similarités aux *voisinages* de ces graines. Cette thèse se focalise sur la deuxième étape des heuristiques à base de graines : comment récupérer et comparer efficacement ces voisinages des graines, pour ne garder que les bons candidats ?

La thèse explore différentes solutions adaptées aux processeurs massivement multicœurs: aujourd'hui, les GPUs sont en train de démocratiser le calcul parallèle et préparent les processeurs de demain. La thèse propose des approches directes (extension de l'algorithme bit-parallèle de Wu-Manber, publiée à PBC 2011, et recherche dichotomique) ou bien avec un index supplémentaire (utilisation de fonctions de hash parfaites). Chaque solution a été pensée pour tirer le meilleur profit des architectures avec un fort parallélisme à grain fin, en utilisant des *calculs intensifs mais homogènes*. Toutes les méthodes proposées ont été implémentés en OpenCL, et comparées sur leur temps d'exécution. La thèse se termine par un prototype de read mapper parallèle, MAROSE, utilisant ces concepts. Dans certaines situations, MAROSE est plus rapide que les solutions existantes avec une sensibilité similaire.

Mots-clés: bioinformatique, calcul d'haute performance, manycore architecture, GPU, OpenCL, index, heuristics seed-based, approximate pattern matching, short read mapper

---

Abstract

Searching similarities between sequences is a fundamental operation in bioinformatics, providing insight in biological functions as well as tools for high-throughput data. There is a need to have algorithms able to process efficiently billions of sequences. To look for approximate similarities, a common heuristic is to consider short words that appear exactly in both sequences, the *seeds*, then to try to extend this similarity to the *neighborhoods* of the seeds. The thesis focuses on this second stage of seed-based heuristics : how can we retrieve and compare efficiently the neighborhoods of the seeds ?

The thesis proposes several solutions tailored for manycore processors such as today's GPUs. Such processors are making massively parallel computing more and more popular. The thesis proposes direct approaches (extension of bit-parallel Wu-Manber algorithm, published in PBC 2011, and binary search) and approaches with another index (with perfect hash functions). Each one of these solutions was conceived to obtain as much fine-grained parallelism as possible, requiring *intensive but homogeneous* computational operations. All proposed methods were implemented in OpenCL and benchmarked. Finally, the thesis presents MAROSE, a prototype parallel read mapper using these concepts. In some situations, MAROSE is more efficient than the existing read mappers with a comparable sensitivity.

Keywords: bioinformatics, high performance computing, manycore architecture, GPU, OpenCL, indexing, heuristics seed-based, approximate pattern matching, short read mapper