Ph.D. in Electronic and Computer Engineering
Department of Electrical and Electronic Engineering
University of Cagliari

Grid and High Performance Computing Applied to Bioinformatics

Emanuele Manca

Advisor:
Prof. Giuliano Armano

PhD Course Coordinator:
Prof. Fabio Roli

Curriculum: ING-INF/05

XXVI Cycle, Academic Year 2013-2014

to my children, Niccolò and Asia.

Abstract

Recent advances in genome sequencing technologies and modern biological data analysis technologies used in bioinformatics have led to a fast and continuous increase in biological data. The difficulty of managing the huge amounts of data currently available to researchers and the need to have results within a reasonable time have led to the use of distributed and parallel computing infrastructures for their analysis. In this context Grid computing has been successfully used. Grid computing is based on a distributed system which interconnects several computers and/or clusters to access global-scale resources. This infrastructure is flexible, highly scalable and can achieve high performances with data-compute-intensive algorithms.

Recently, bioinformatics is exploring new approaches based on the use of hardware accelerators, such as the Graphics Processing Units (GPUs). Initially developed as graphics cards, GPUs have been recently introduced for scientific purposes by reason of their performance per watt and the better cost/performance ratio achieved in terms of throughput and response time compared to other high-performance computing solutions.

Although developers must have an in-depth knowledge of GPU programming and hardware to be effective, GPU accelerators have produced a lot of impressive results.

The use of high-performance computing infrastructures raises the question of finding a way to parallelize the algorithms while limiting data dependency issues in order to accelerate computations on a massively parallel hardware.

In this context, the research activity in this dissertation focused on the assessment and testing of the impact of these innovative high-performance computing technologies on computational biology. In order to achieve high levels of parallelism and, in the final analysis, obtain high performances, some of the bioinformatic algorithms applicable to genome data analysis were selected, analyzed and implemented. These algorithms have been highly parallelized and optimized, thus maximizing the GPU hardware resources. The overall results show that the proposed parallel algorithms are highly performant, thus justifying the use of such technology.

However, a software infrastructure for workflow management has been devised to provide support in CPU and GPU computation on a distributed GPU-based infrastructure. Moreover, this software infrastructure allows a further coarse-grained data-parallel parallelization on more GPUs. Results show that the proposed application speed-up increases with the increase in the number of GPUs.

Acknowledgments

I am grateful to many people who have contributed towards shaping this thesis. At the outset, I would like to express my sincere gratitude and appreciation to my advisor, Associate Professor Giuliano Armano, for his advice during my doctoral research enterprise, for the past four years. I thank him for believing in me and for providing me with the opportunity to work with a talented team of researchers. His talent and achievements serve as inspiration and model for any researcher.

I would also like to express my sincere gratitude to my co-advisor Andrea Manconi. I have to thank him for all the support, the suggestions, the patience, and every kind of help that he gave to me. His observations and comments helped me to establish the overall direction of the research and to move forward with investigation in depth. This thesis would not have been possible without his support and in-sights.

I am grateful to Luciano Milanesi, the coordinator of the CNR Interdepartmental Bioinformatics Research Network, for establishing the high-level scientific environment were I had the opportunity of expending these years as researcher. He has constantly forced me to remain focused on achieving my goals.

I thank my colleagues, especially Alessandro Giuliani, Mario Franco Locci, Emanuele Tamponi, Francesca Fanni, Amir Mohammad Amiri, Mohammad Reza Farmani, Matteo Baire, Andrea Addis, Eloisa Vargiu, Alessandro Orro, Francesco Maxia and Filippo Ledda for creating good working environment and for always smiling with me.

I am also grateful to the coordinator of the PhD program, Prof. Fabio Roli, for his efforts, and to his collaborator Carla Piras, for their helpfulness.

I acknowledge the funding support from the CNR-ITB (Italian National Research Council at the Institute of Biomedical Technologies), Regione Sardegna and my University, for providing scholarships to pursue my doctoral studies.

I am really grateful to my family for supporting and encouraging me during these four years.

Last, but not least, I would like to dedicate this thesis to my children, Niccolò and Asia.

Statement of Authorship

I hereby certify that this PhD thesis has been composed by myself and describes my own work unless otherwise acknowledged in the text. All references and verbatim extracts have been quoted and all sources of information have been specifically acknowledged. It has not been accepted in any previous application for a degree.

Emanuele Manca

Publications

Parts of this thesis have been published in the journal papers listed below, which will be referred to by their roman numerals:

- I A. Manconi, A. Orro , E. Manca, G. Armano, L. Milanesi. (2012). G-SNPM - A GPU-based SNP mapping tool. In Proceeding of the 12th International Workshop on Network Tools and Application for Biology (NETTAB 2012 "Integrated Bio-Search"), pp. 138-139
- II A. Manconi, A. Orro, E. Manca, G. Armano, L. Milanesi. (2014). A tool for mapping Single Nucleotide Polymorphisms using Graphics Processing Units. BMC Bioinformatics, 15(Suppl 1):S10
- III A. Manconi, A. Orro, E. Manca, G. Armano, L. Milanesi. (2014). GPU-BSM: A GPU-Based Tool to Map Bisulfite-Treated Reads. PLoS ONE 9: e97277
- IV E. Manca, A. Manconi, A. Orro, G. Armano, L. Milanesi. (2014). CUDA-Quicksort: An Improved GPU-based Implementation of Quicksort. Submitted to Concurrency and Computation: Practice and Experience. Manuscript CPE-14-0292
- V A. Manconi, A. Orro, E. Manca, G. Armano, L. Milanesi. (2014). G-CNV: A GPU-based Tool for Preparing Data to Detect CNVs with Read Depth Methods. Front. Bioeng. Biotechnol. 3:28. doi: 10.3389/fbioe.2015.00028

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Bioinformatics is an interdisciplinary research area that involves the use of techniques including applied mathematics, informatics, statistics, computer science, artificial intelligence, chemistry and biochemistry to solve biological problems usually at molecular level. The ultimate goal of bioinformatics is to uncover and decipher the richness of biological information hidden in the mass of data and to obtain a clearer insight into the fundamental biology of organisms. The beginning of the twenty-first century has been characterized by an explosion of biological information. The avalanche of data grows daily and arises as a consequence of advances in the fields of molecular biology and genomics and proteomics. The need of managing this huge amount of complex data and to obtain results within a reasonable time have increasingly led to the use of Grid and High Performance Computing (HPC) for their analysis.

The research activity in this dissertation has focused on the use of Grid and HPC infrastructures for solving computationally expensive bioinformatics tasks, where, due to the very large amount of available data and the complexity of the tasks, new solutions are required for efficient data analysis and interpretation.

1.2 Motivations and Objectives

Recent advances in genome sequencing technologies and modern biological data analysis technologies used in bioinformatics have led to a fast and continuous increase in biological data, also shown by the exponential increase in the Genbank genome sequences and in the Protein Data Bank (PDB) structures over the last ten years (see Figure 1.1). The difficulty of managing the huge amounts of data currently available to researchers and the need to obtain results within a reasonable time have led to the use of distributed and parallel computing infrastructures for their analysis. In this context Grid computing has been successfully used.

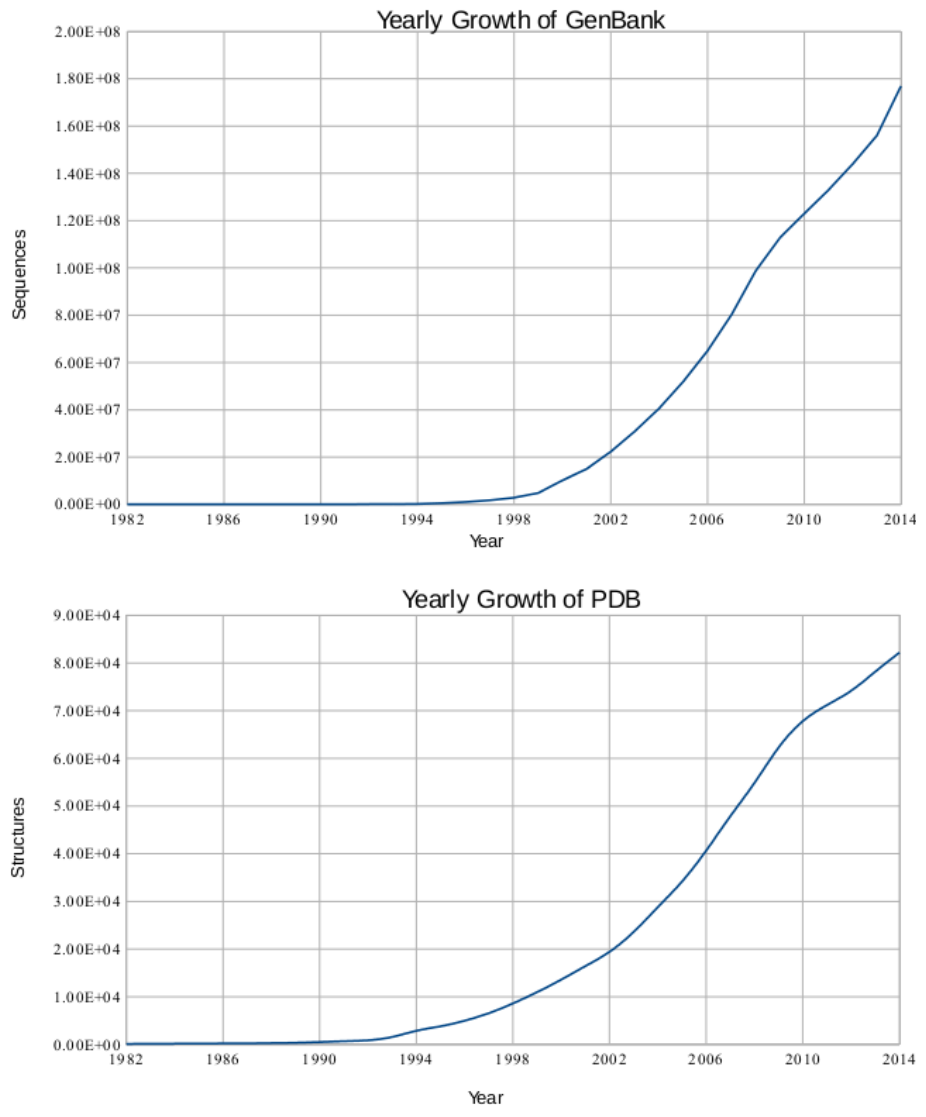


Figure 1.1: **Yearly Growth of GenBank genome sequences/Protein Data Bank (PDB) structures**

Grid computing is a form of distributed computing that combines geographically distributed resources to create a high throughput computing infrastructure [50]. This global infrastructure facilitates the sharing of resources and access to a large-scale computational platform that would otherwise be unavailable. The original motivation for the Grid was the need for a distributed computing infrastructure allowing for coordinated resource sharing and problem-solving in dynamic, multi-institutional environments. Problem solving of advanced science and engineering problems with emphasis on collaborative and multi-disciplinary applications requires the coordinated and well organized interaction of groups of individuals and organizations. This has led to the concept of a virtual organization (VO) [51] which represents the mode of use of Grids. Examples of VOs could include the members of a research group in a university or physicists collaborating in an international experiment. A Virtual Organization has the responsibility of providing a scalable, flexible and secure environment for researchers [191]. It must conform to a set of open standards and protocols when developing Grid solutions.

An example of virtual organization based on the Grid technology whose infrastructure is dedicated or involved with bioinformatics is the *Enabling Grids for E-sciencE project* (EGEE) [54] which brings together scientists and engineers from more than 240 institutions in 45 countries world-wide to provide a seamless Grid infrastructure for e-Science. The EGEE project provides researchers in academia and industry with access to a production-level Grid infrastructure, independent of their geographic location. The biomedical applications area is a broad scientific field which has been divided in three different sectors in the EGEE project. The medical imaging sector targets the computerized analysis of digital medical images. The bioinformatics sector targets gene sequences analysis and includes genomics, proteomics and phylogeny. The drug discovery sector aims at helping to speed-up the process of finding new drugs through in-silico simulations of proteins structures and dynamics.

Grid infrastructures can execute a variety of tasks from many research project areas such as high-energy physics, bioinformatics and chemistry. These applications may be also distributed in nature and require high-throughput or fast data processing capabilities. Applications are run on the resources best suited to perform them. A data-intensive task might be executed on a remotely located supercomputer, while a less-demanding task might run on a smaller, local machine. The assignment of computing tasks to computing resources is determined by a scheduler, and ideally this process is hidden to the end user. Coordinating applications on Grids can be a complex task, especially when coordinating the flow of information across distributed computing resources. Grid workflow systems have been developed as a specialized form of a workflow management system designed specifically to compose and execute a series of computational or data manipulation steps, or a workflow, in the Grid context.

Many bioinformatics projects are based on Grid infrastructures, such as:

- ROSETTA project for protein folding design and docking [187] – it is a distributed computing project for protein structure prediction on BOINC Desktop Grid, run by the Baker laboratory at the University of Washington. Goal of this project is the use of distributed computing to predict and design protein structure and protein complexes with the help of about sixty thousand active volunteered computers processing at 83 teraFLOPS on average.

- Projects for the identification of genes and regulatory patterns [183] – this project uses evolutionary algorithms implemented in a Grid computing infrastructure to create computational models of gene-regulatory networks based on observed microarray data.

In addition to Grid computing, bioinformatics has recently showed a growing interest in innovative HPC technologies, such as hardware accelerators. Accelerators were initially developed as CPU coprocessors intended for the execution of specific data-compute-intensive tasks. State-of-the-art accelerators are based on a SIMD (Single Instruction Multiple Data) or MIMD (Multi Instruction Multiple Data) many-core architecture, affording a very high level of parallelism. In particular, bioinformatics is exploring new approaches based on the use of Graphics Processing Units (GPUs). Driven by the increasing industry demand for real-time, high-definition 3D graphics, the GPU hardware accelerator has evolved into a highly-parallel, multi-threaded, and many-core processor. It is preliminary dedicated to efficiently support the graphics shader programming model, in which a program for one thread draws one vertex or shades one pixel fragment. The GPU excels with fine-grained, data-parallel workloads consisting of thousands of independent threads executing vertex, geometry, and pixel-shader program threads concurrently. GPUs have been recently introduced for scientific purposes by reason of their performance per watt and the better cost/performance ratio achieved in terms of throughput and response time compared to other HPC solutions. The speed of many data-processing-intensive applications can be increased just by using the GPUs as co-processors to CPU to accelerate its general purpose computations that were once handled by the CPU alone. NVIDIA and AMD are the main GPU manufacturers with a long list of different models and features. These companies have been producing different platforms that can use parallel computing architectures to utilize the GPU stream processors, in tandem with the CPU, to significantly accelerate any computing process. The use of GPUs for scientific purposes encouraged graphics card manufacturers to develop a new unified graphics and computing GPU architecture and introduce new programming models dedicated to general purpose computations, such as CUDA by NVIDIA and OpenCL Accelerated Parallel Processing (APP) by AMD, which provide low-level or direct access to the multi-threaded computational resources and associated memory bandwidth of GPUs, exposing them as large arrays of parallel processors.

Even if their main limitation and difficulty is that the developer must have an in-depth knowledge of GPU programming and hardware, GPU accelerators have produced a lot of impressive results.

Recently, GPUs have been successfully used to parallelize some important bioinformatic algorithms. Examples thereof are:

- G-BLASTN – a GPU-based tool for multiple sequence alignment [206];
- GENIE – a software package for gene-gene interaction analysis in genetic association studies using multiple GPU or CPU cores [38];
- CUDASW++ – a GPU-based tool for sequence database searching [118].

The use of high-performance computing infrastructures thus raises the question of finding a way to parallelize the bioinformatic algorithms while limiting data dependency issues in order to accelerate computations on a massively parallel hardware.

In this context, the research activity in this dissertation focused on the assessment and testing of the impact of these innovative HPC technologies on computational biology. In order to achieve high levels of parallelism and, in the final analysis, obtain high performances, some of the bioinformatic algorithms applicable to genome data analysis were selected, analyzed and implemented.

1.3 Thesis Structure

The rest of the thesis is structured as follows:

Chapter 2 – This chapter gives an overview of parallel computing exploring different parallel architectures, while showing some important aspects of each. It also explains in great detail the NVIDIA GPU architectures and CUDA programming model used to implement the bioinformatic applications described in the next chapters.

Chapter 3 – This chapter describes a GPU-based tool aimed at mapping nucleotide sequences representative of a single-nucleotide base polymorphism on a reference genome. This chapter quotes [II] (see Publications above).

Chapter 4 – This chapter describes a GPU-based tool aimed at mapping bisulfite-treated reads with the aim of detecting methylation levels of cytosines. This chapter quotes [III] (see Publications above).

Chapter 5 – This chapter describes CUDA-Quicksort, a GPU-based implementation of the quick-sort. This chapter quotes [IV] (see Publications above).

Chapter 6 – This chapter describes G-CNV, a GPU-based tool for preparing data to detect CNVs with read depth methods. This chapter quotes [V] (see Publications above).

Chapter 7 – This chapter gives an overview of the gUSE science gateway and the BOINC Desktop Grid, highlighting some important aspects of each. It also illustrates the Distributed GPU- and CPU-based Infrastructure system implemented through gUSE and BOINC. Finally, it describes how the proposed tools have been ported to Distributed GPU- and CPU-based Infrastructure.

Chapter 8 – This chapter gives a summary of the thesis and highlights future work.

Chapter 2

GPU Computing

GPUs are hardware accelerators that are increasingly used to deal with computationally intensive algorithms. From an architectural perspective, GPUs are very different from traditional CPUs. Indeed, the latter are devices composed of few cores with lots of cache memory able to handle a few software threads at a time. Conversely, the former are devices equipped with hundreds of cores able to handle thousands of threads simultaneously, so that a very high level of parallelism can be reached. GPGPU (General Purpose Computing on Graphics Processing Units) is a methodology for high-performance computing that combines CPUs with GPUs to deal with data parallel and throughput intensive algorithms. As CPUs are more effective than GPUs for serial processing, they are used to perform serial parts of the algorithm, whereas GPUs are used to perform parts of the algorithm where processing of large blocks of data is done in parallel. The main disadvantage of using GPUs is related with the effort required to code algorithms. To take advantage of the GPU technology, algorithms must be coded to reflect the architecture of these hardware accelerators. Incorporating GPU support into existing codes is very difficult and typically requires significant changes of the code and of the algorithm.

This chapter gives an overview of parallel computing exploring different parallel architectures, while showing some important aspects of each. It also explains in great detail the NVIDIA GPU architectures and CUDA programming model used to implement the bioinformatic applications described in the next chapters.

2.1 Parallel computing overview

Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved in parallel. There are several different forms of parallelism: instruction-level, data, and task parallelism. Parallel processors can be roughly classified according to the level at which the hardware supports parallelism. Actually, parallelism has been employed for many years, mainly in high performance applications, but it has received growing concern recently due to the physical constraints preventing clock frequency scaling. As power saving has become a concern in recent years, parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multi-core and many-core processors.

Nowadays the traditional single-core processors are being replaced by multi-core or many-core processors.

Multi-core processors are characterized by a processing unit containing a relatively small number of cores, where two or multiple independent processing elements are typically integrated onto a single integrated circuit die. Intel i7 processor is an example of a multi-core processor. This processor may have either 2 or 4 cores on chip. The ability of multi-core processors to increase applications performance is strongly constrained by the use of multiple threads within applications. It relies on effective exploitation of multiple-thread parallelism. To exploit their full potential, applications will need to move from a single to a multi-threaded model. Threads can be much smaller and still be effective, and in addition to that, the automatic parallelization is more feasible, and allows scheduling the workload across multiple cores. This scheduling is performed by the operating system, which maps threads/processes to different cores. The main limitation of multi-core processors is due to their low scalability. As adding more cores to the on-chip bus results in an information traffic jam limiting the benefits of multiple cores.

A many-core processor is a highly integrated complex system in which the number of cores is large enough that traditional multi-processor techniques are no longer efficient (this threshold is somewhere in the range of several tens of cores). In fact, all data must pass through the same path. An examples of many-core is the Tilera processor [131] that addresses the multi-processor scalability problem with a architecture that can harness the processing power of hundreds of cores on a single chip, it provides general purpose many-core processors with 16 to 100 identical processor cores (tiles) interconnected with on-chip network.

One of the many-core architectures that received more interest during the last years are GPUs. GPUs are hardware accelerators that are increasingly used to deal with computationally intensive algorithms. Driven by the increasing industry demand for real-time, high-definition 3D graphics, the programmable GPUs have evolved into a highly parallel, multi-threaded, and many-core processor. As shown in Figure 2.1, GPU devices lead the way in peak computing performance. It should be noted that the performance levels shown in Figure 2.1 are very rarely achieved by real-world applications, but are speeds theoretically attainable by such devices. As shown in Figure 2.1 the performance increase have been huge with multi-core CPUs too, but the raw computing capabilities have increased most on the GPU.

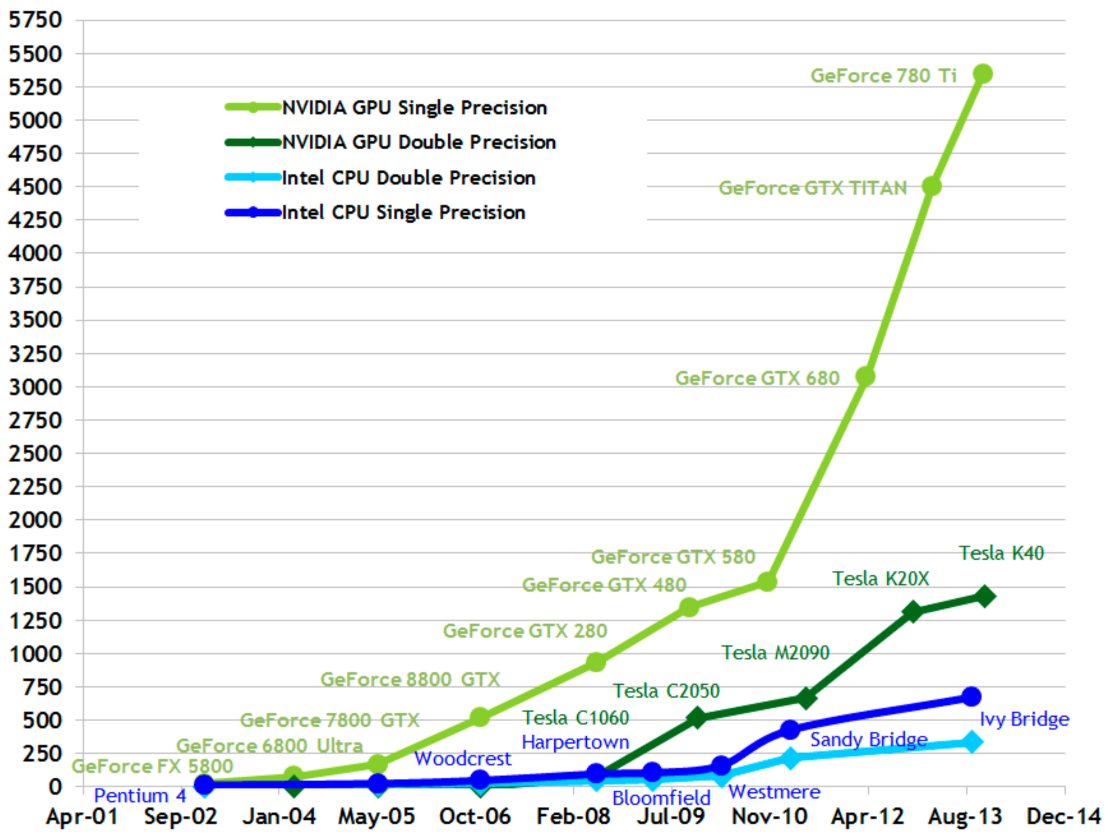


Figure 2.1: **CPU and GPU compute capabilities.** Image from [151]

GPUs have been recently introduced for scientific purposes by reason of their performance per watt and the better cost/performance ratio achieved in terms of throughput and response time compared to other parallel computing solutions. With such large amounts of raw computational power theoretically attainable with GPU devices, a growing number of scientists have begun porting their algorithms to GPU-based computing systems.

Currently there are two dominating hardware vendors in the field of general-purpose GPU computing – NVIDIA and AMD. These companies have been producing different platforms that can use parallel computing architectures to utilize the GPU processors, in tandem with the CPU, to significantly accelerate computing process. NVIDIA and AMD have alternative proprietary GPU platforms each of which is compatible only with their own hardware. Specifically, NVIDIA's distributes the CUDA (Compute Unified Device Architecture) Toolkit [151] while AMD distributes the OpenCL Accelerated Parallel Processing (APP) SDK [17], both provide low-level or direct access to the multi-threaded computational resources and associated memory bandwidth of GPUs, exposing them as large arrays of parallel processors. APP SDK by AMD uses OpenCL (Open Computing Language)[141], a vendor-independent standard, which has been designed to allow HPC application developing independently from hardware.

2.2 Architecture models of GPUs

The exponentially increasing performance levels of GPU devices has largely been driven by the video game industry. Interactive 3D video games demand a very high level of data throughput and an absolutely staggering number of floating-point operations per second. Consider that for a SXGA (1280x1024) display, there is a total of about 1.3Million individual pixels. With a commonly desired frame rate of 30fps, there is a worst-case scenario of having to compute > 39Million pixel-values every second with each value requiring multiple floating point and memory read/write operations. With such high demands for throughput, programmers in the computer graphics community adopted thread-level parallelism as the dominant paradigm for producing satisfactory results and GPU manufactures followed suit by building hardware that could realize the performance benefits of this paradigm.

A high-level comparison between modern CPU and GPU architecture is given in Figure 2.2. As shown in figure the CPU architecture is composed of few cores with lots of cache memory able to handle a few software threads at a time. Indeed, the GPU architecture is equipped with hundreds of cores able to handle thousands of threads simultaneously. The main difference is that most of the GPU transistors are used for strict computation while much of the CPU transistors are used for caching, branch prediction, out-of-order execution optimization during run-time that has become part of the optimizing pipeline for a general-purpose processor. The reason to this evolution is that the GPU have evolved to run the needed graphic rendering pipeline at high throughput, containing many parallel operations that are totally

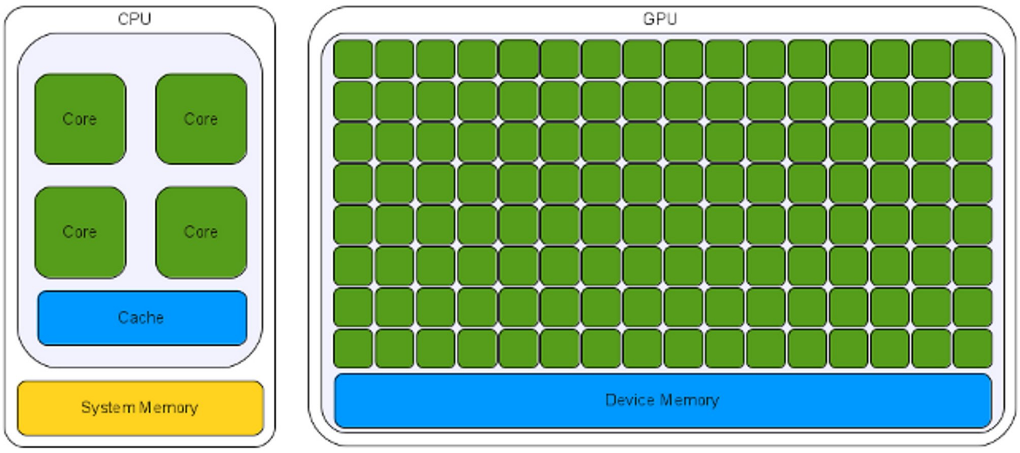


Figure 2.2: **Multi-core and many-core processors** Multi-core processors as CPUs are devices composed of few cores with lots of cache memory able to handle a few software threads at a time. Conversely, many-core processors as GPUs are devices equipped with hundreds of cores able to handle thousands of threads simultaneously.

independent of each other.

2.2.1 NVIDIA GPUs

NVIDIA has been very vocal in promoting the GPU-based approach to parallel computing. NVIDIA GPU computing solutions enable the necessary transition to fast and energy-efficient parallel computing power. The GPU architecture is built around a scalable array of multithreaded Streaming Multiprocessors (SM)(see Figure 2.3). It is viewed as a compute device capable of executing a huge number of threads in parallel. GPUs can operate as co-processors of the main CPU (or host); so, data-independent, compute-intensive portions of applications running on the host are off-loaded into the GPU (or device).

In the NVIDIA GPU-based architecture, parallelization is obtained through the execution of tasks in a number of CUDA cores. Cores are grouped in streaming multiprocessors that execute in parallel (see Figure 2.4). Each core has both floating-point and integer execution units and all cores in a streaming multiprocessor execute the same instruction at the same time. This computational paradigm, called SIMT (Single Instruction Multiple Thread), can be considered as an advanced form of the SIMD (Single Instruction Multiple Data) paradigm. The code is executed in groups of 32 threads, called warps. The dispatching and scheduling of each thread in a

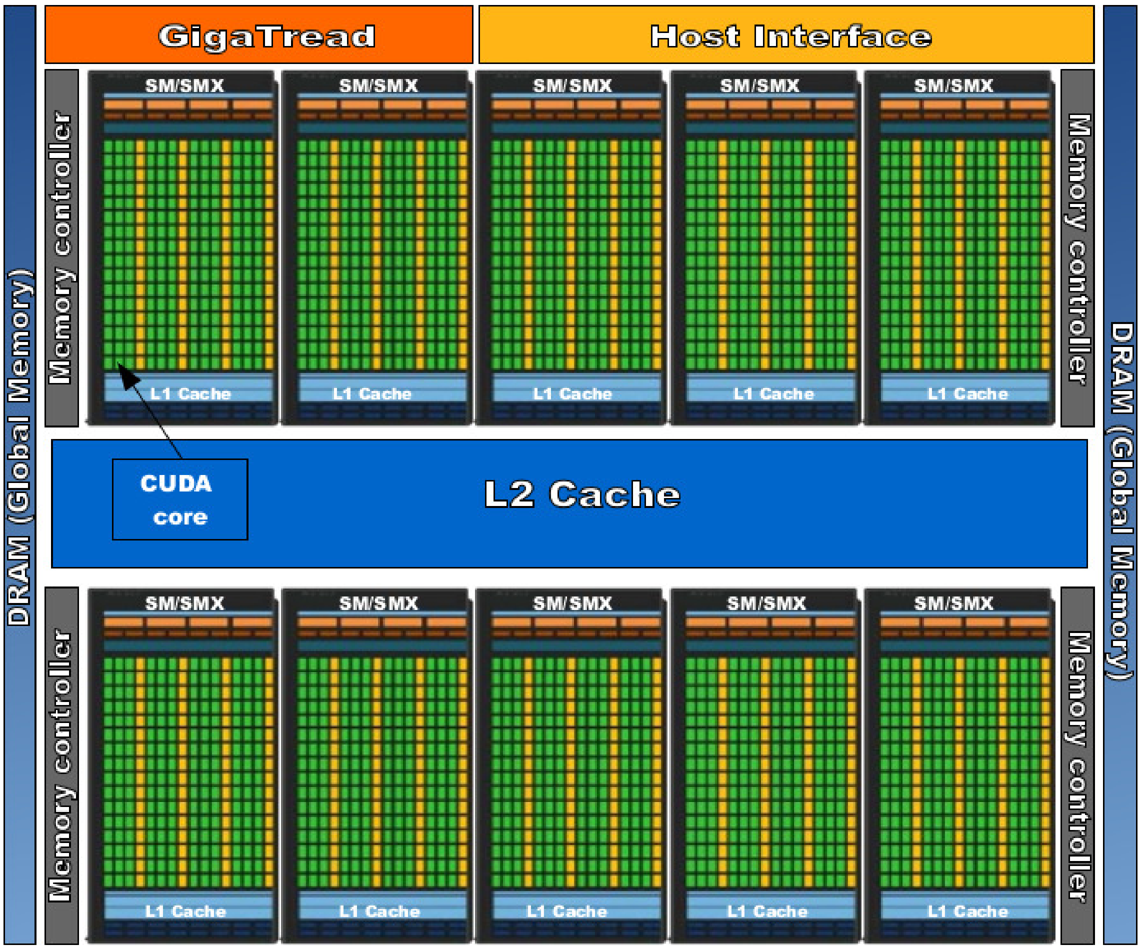


Figure 2.3: **GPU Architecture**. The NVIDIA GPU is a many-core processor equipped with hundreds of cores able to simultaneously handle thousands of threads.

warp, are managed individually for each SM by a warp scheduler unit and a dispatch unit (see Figure 2.4). The GPU architecture supports a DRAM (Dynamic Random Access Memory) memory, with a low-latency L2 cache and a high-bandwidth L1 cache per streaming multiprocessor. Each multiprocessor has also a programmable L1 cache called shared memory, managed by a specific software.

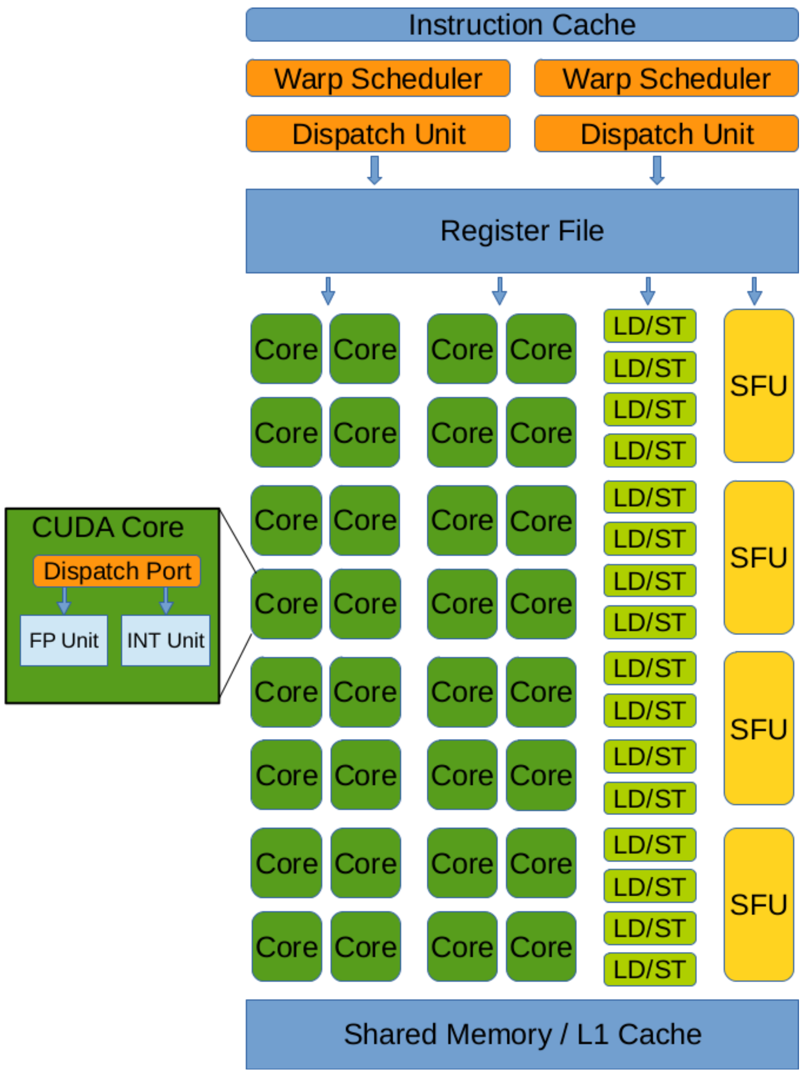


Figure 2.4: **Stream Multiprocessor of Fermi Architecture.** Image derived from [58].

GPU Memory Hierarchy

NVIDIA GPUs have several memory spaces available each with its own benefits and limitations. Effectively understanding and appropriate use of these memory spaces is essential for achieving acceptable performance on the GPU. The fastest available memory for GPU computation is represented by device registers. Registers are divided among all threads which reside simultaneously on the SM. Each multiprocessor also has a region of shared memory space, which performs almost as fast as registers. The on-chip shared memory has very low access latency and high bandwidth similar to an L1 cache (see Figure 2.4). The shared memory allows the parallel threads to run on the cores in a SM to share data without sending them over the system memory bus. GPUs have DRAM called global memory with a high access latency when compared to registers or shared memory.

The SM provides load/store instructions to access the global memory. It coalesces individual accesses of parallel threads in the same warp into fewer memory-block accesses when the addresses fall in the same block and meet alignment criteria. A SM takes few clock cycles to issue one memory instruction for a warp. When accessing global memory, there are, in addition, hundreds of clock cycles of memory latency. Because global memory latency can be hundreds of processor clocks, programs may copy the data to shared memory when it must be accessed multiple times by a thread block. GPU load/store memory instructions use integer byte addressing to facilitate conventional compiler code optimization. The large thread count in each SM, together with support for many outstanding load requests, helps to cover load-to-use latency to the global memory. The modern architecture GPUs also provide atomic read-modify-write memory instructions, facilitating parallel reductions and parallel-data structure management.

Hardware details

Different modern architecture GPUs are available from NVIDIA (e.g., Fermi, Kepler and Maxwell).

With the launch of Fermi GPU in 2009, NVIDIA ushered in a new era in the HPC industry [58]. This architecture consists of a SM array. Each SM includes 32 cores, 16 load/store units, four special-function units, a 32K-word register file, 64K of local SDRAM split between cache and shared memory, and thread control logic. Each core has both floating-point and integer execution units. This architecture has up to 6GB of GDDR5 memory.

In 2012, with the Kepler GPU [149], NVIDIA raises the bar for the HPC industry, yet again. The main changes available in the Kepler GPU architecture include:

- Dynamic parallelism – it supports GPU threads launching new threads, so that the GPU can work more autonomously from the CPU by generating new work for itself at run-time. The dynamic parallelism allows the implementation of recursive algorithms.

- Hyper-Q – it enables CPU cores to initiate work on the same GPU simultaneously. Hyper-Q increases the total number of connections (work queues) between the host and the Kepler GPU compared to the single connection of the Fermi GPU.

- SMX architecture – it provides a new streaming multiprocessor design optimized for performance per watt. Each SMX contains 192 CUDA cores (up from 32 cores in Fermi), 64K-word register, 32 special function units (SFU), and 32 load/store units. Unlike Fermi SM, which has two scheduler and two dispatch units (see Figure 2.4), each SMX features four warp schedulers and eight instruction dispatch units, allowing four warps (groups of 32 threads) to be issued and executed concurrently. The quad scheduler selects four warps, and two independent instructions per warp can be dispatched each cycle.

In 2014, NVIDIA implements a new GPU architecture called Maxwell [152], especially designed for high power saving (see Table 2.1). The first Maxwell GPU, called GM107, has been designed for using in power limited environments. Maxwell GM107 provides few consumer-facing additional features. NVIDIA increased the amount of L2 cache reducing the memory bandwidth needed. NVIDIA also changed the streaming multiprocessor design of Kepler, naming it SMM. The layout of SMM units is partitioned so that each of the four warp schedulers controls isolated CUDA cores, load/store units and special function units, unlike Kepler, where the warp schedulers share the resources. SMM allows for a finer-grain allocation of resources than SMX, saving power when the workload is not optimal for shared resources.

Table 2.1 shows a comparison among these architectures. As shown in the table, the Kepler architecture offers a huge leap forward in power efficiency, achieving up to 3x the performance per watt of Fermi. The performance per watt for single-precision operation achieved by Kepler and Maxwell GPUs is similar, but the latter outperforms the first in terms of power saving. As to peak performance, Kepler GPUs outperform both Fermi and Maxwell architectures.

Chip name	**GF110**	**GK110**	**GK110b**	**GK210**	**GM107**
Product name	GeForce GTX 570	Tesla k20	Tesla k40	Tesla k80	GeForce GTX 750Ti
Architecture	Fermi	Kepler	Kepler	Kepler	Maxwell
Peak performance [TF]	1.40(SP)	4.10(SP) 1.17(DP)	5.36(SP) 1.43(DP)	8.74(SP) 2.91(DP)	1.30(SP) 40GF(DP)
Wattage (TDP)	219W	225W	235W	300W	60W
Performance per watt[GF/W]	6.4(SP)	18.24(SP) 5.21(DP)	22.82(SP) 6.08(DP)	29.13(SP) 9.70(DP)	21.86(SP) 0.66(DP)
Onboard GDDR5 Memory	2GB	5GB	12GB	24GB	2GB
Memory bandwidth [GB/s]	152	208	288	480	86

Table 2.1: **Comparison among Fermi, Kepler and Maxwell Architectures.**

2.2.2 AMD GPUs

AMD is the main competitor of NVIDIA in the HPC industry. It has recently launched a new GPU Architecture called Graphics Core Next (GCN)[18]. GCN Architecture is also AMD's first design specifically engineered for general computing. Representing the cutting edge of AMD's graphics expertise, GCN GPUs are more than capable of handling workloads and programming languages-traditionally exclusive to the main processor. Coupled with the dramatic rise of GPU-aware programming languages like C++ and OpenCL.

GCN is a RISC SIMD architecture, it replaces AMD's TeraScale family of VLIW (Very Long Instruction Word) SIMD architectures used since the Radeon HD 2000 Series. It is used in 28 nm graphics chips in the Radeon HD 7000, HD 8000 and Rx 200 series. The heart of GCN is the new Compute Unit (CU) design. CUs are the basic computational building blocks of the GCN Architecture. Each CU includes 4 separate SIMD engines, each of consisting of up to 16 ALUs (see Figure 2.5). In NVIDIA parlance, a CU can be thought of as a streaming processing cluster, a SIMD Engine can be thought of as a streaming processing, a SIMD ALU as four CUDA cores. Each ALU could execute bundles of 4 or 5 independent VLIW instructions, and is fully IEEE-754 compliant for single-precision and double-precision floating point operations. SIMD units can handle groups of 64 threads (called wavefronts) and execute one wavefront at a time as happens to warps (groups of 32 threads) in NVIDIA GPU. Each of these SIMD units executes different operations simultaneously, but all threads within a wavefront execute the same instruction. This means that a GCN GPU with 32 CUs, such as the AMD Radeon HD 7970, can be working on up to 8192 (32 CUs x 4 SIMDs x 64 threads) threads at a time.

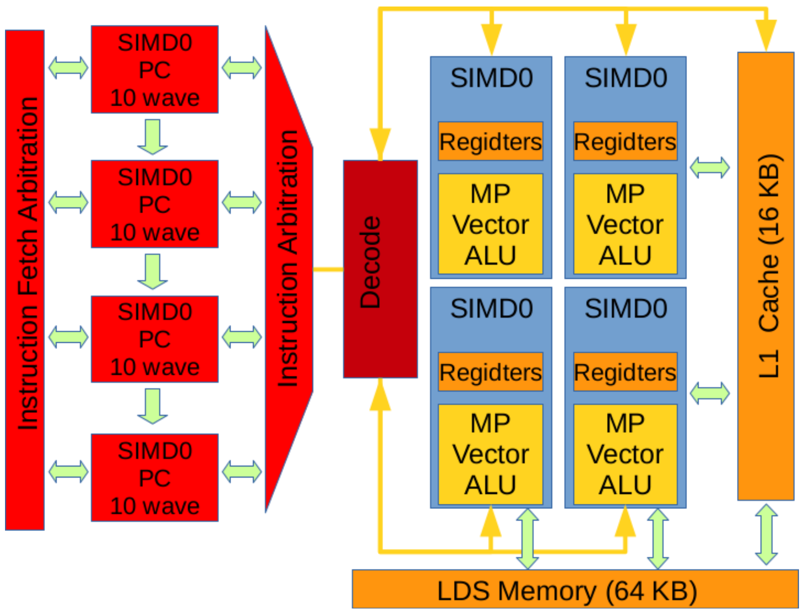


Figure 2.5: **Compute Unit of AMD GPU Architecture.** Image derived from [18].

The GCN architecture supports a DRAM memory, with a low-latency L2 cache and a high-bandwidth 32KB L1 cache shared by a cluster of up to 4 CUs (see Figure 2.5). The fastest available memory for GPU computation is device registers. Each CU contains 8KB of scalar registers that are divided into 512 entries for each SIMD that has also 64 KB of personal registers. The scalar registers are shared by all wavefronts on the SIMD, this is essential for wavefront control flow, for example, comparisons will generate a result for each of the 64 threads in a wavefront. Like NVIDIA GPU streaming multiprocessor each CU has a 64KB region of shared memory space, which performs almost as fast as registers. This shared memory called Local Data Share (LDS) has very low access latency and high bandwidth (see Figure 2.5). LDS memory is an explicitly addressed memory that acts as a third register file specifically for synchronization and communication within a work-group of threads.

Table 2.2 shows a comparison between the best AMD and NVIDIA GPU architectures. The overall performance achieved by AMD and NVIDIA GPUs is really similar. As shown in the table, the performance per watt achieved by NVIDIA Tesla K80 outperforms AMD GPUs. As to peak performance for single precision operations, AMD Radeon R9 295X2 outperforms NVIDIA architectures. As to peak performance for double-precision operations, NVIDIA GPUs outperform AMD GPUs.

Vendor	AMD	AMD	NVDIA	NVIDIA
Product name	Radeon HD 8990	Radeon R9 295X2	Tesla k80	GeForce GTX Titan Z
Peak performance [TF]	8.19(SP) 2.04(DP)	11.46(SP) 1.43(DP)	8.74(SP) 2.91(DP)	8.12(SP) 2.70(DP)
Wattage (TDP)	375W	500W	300W	375W
Performance per watt[GF/W]	21.86(SP) 5.46(DP)	22.90(SP) 2.86(DP)	29.13(SP) 9.70(DP)	21.76(SP) 7.20(DP)
Onboard GDDR5 Memory	6GB	8GB	24GB	12GB
Memory bandwidth [GB/s]	288x2	320x2	240x2	336.4x2

Table 2.2: **Comparison between AMD and NVIDIA GPU Architectures.**

2.3 GPGPU Programming Models

General-purpose computing on GPU (GPGPU) is the use of a GPU – which typically handles the computations for computer graphics – to perform computations in applications typically handled by the CPU. A GPGPU programming model enables the programmer to write and execute programs according to a defined execution and memory model that takes advantage of the underlying massively parallel hardware architecture provided by the GPU.

The idea of using a GPU programming model for general purpose computing is a concept that dates back nearly two decades. Initially, utilizing GPU devices for the execution of non-graphics related algorithms was a very difficult task. Essentially, programmers were forced to use graphics application programming interfaces (APIs) such as OpenGL or Direct3D to gain access to the GPU chip. The main strategy for doing GPGPU computing was to find clever ways to fit some target algorithms into computer graphics abstractions that are compatible with the graphics API being used.

The process of doing GPGPU computing changed dramatically with the release of NVIDIA's CUDA toolkit in 2007 and and OpenCL (vendor-independent) in 2008. CUDA can only be used on NVIDIA GPUs, while OpenCL is an open standard and is aimed at heterogeneous computing platforms. These programming models allowed programmers to ignore the underlying graphical concepts in favor of more common high-performance computing concepts. With the introduction of the CUDA and OpenCL general-purpose computing APIs, in new GPGPU codes it is no longer necessary to map the computation to graphics primitives.

In the early days, NVIDIA realized that the interest in using GPU devices for general purpose computing was growing and wished to capitalize on this emerging market. The goal was to provide a specially extended version of some high-level programming language that would allow programmers to gain direct access to GPU devices without needing in-depth knowledge of computer graphics algorithms and techniques. The result was the CUDA toolkit that allows programmers to write parallel code in a specially extended version of the C programming language for parallel execution on most modern NVIDIA GPUs. CUDA allows for a much more generic parallel programming model than was possible in earlier generation GPUs and allows programmers to use common parallel programming abstractions such as parallel threads, barrier synchronization, and atomic operations in GPU-based code.

OpenCL was initiated by Apple and is now maintained by the Khronos group. It is an open and royalty-free standard for performing heterogeneous computing. The programming models of OpenCL and CUDA are quite similar, except that OpenCL also supports parallel task programming and is designed to run on many different kinds of devices. Indeed, it can be used to run code for multi-core CPUs, GPUs and other devices supporting the OpenCL specification.

Programming models like CUDA and OpenCL give a fine-grained optimization at the cost of good level of abstraction. Tools using CUDA or OpenCL consist of host code running on CPU and separate code snippets that are compiled for the current system GPU. The host code orchestrates allocation and deallocation of memory buffers on the GPU, executing programs and copying data to and from the device. None of these tools are really high-level, and none of them is as easy to program as making a standard C program on the CPU. Since CUDA and OpenCL are both fairly low-level programming models, more high-level approaches have also been released in the last few years including compiler directive-based programming models like OpenACC.

The following sections offer a detailed description of the CUDA programming model and a short introduction to the OpenACC and OpenCL programming models.

2.3.1 CUDA Programming Model

CUDA was first introduced in November 2007 with the aim to make it easier to create GPU computing programs by avoiding using traditional shader languages meant for graphics programming such as DirectX and OpenGL. Instead CUDA uses the C/C++ programming language with syntax extensions and library functions to express parallelism, data locality, and thread cooperation mapping to the underlying hardware architecture. NVIDIA recently released CUDA version 6.5 which includes the CUDA toolkit, samples, and a unified CUDA enabled driver for NVIDIA devices. CUDA programs are compiled using the NVIDIA LLVM-based C/C++ compiler called nvcc. The two most important concepts of the CUDA programming model is its execution and memory model.

The CUDA execution model (see Figure 2.6) can be described as follow: the GPU creates an instance of the kernel program that is made of a set of threads grouped in blocks in a grid. Each thread has a unique ID within its block and a private memory and registers, and runs in parallel with others threads of the same block. All threads in a block execute concurrently and cooperatively by sharing memory and exchanging data. A block, identified by a unique ID within the block grid, can execute the same kernel program with different data that are read/written from a global memory. Each block in the grid is assigned to a streaming multiprocessor in a cyclical manner.

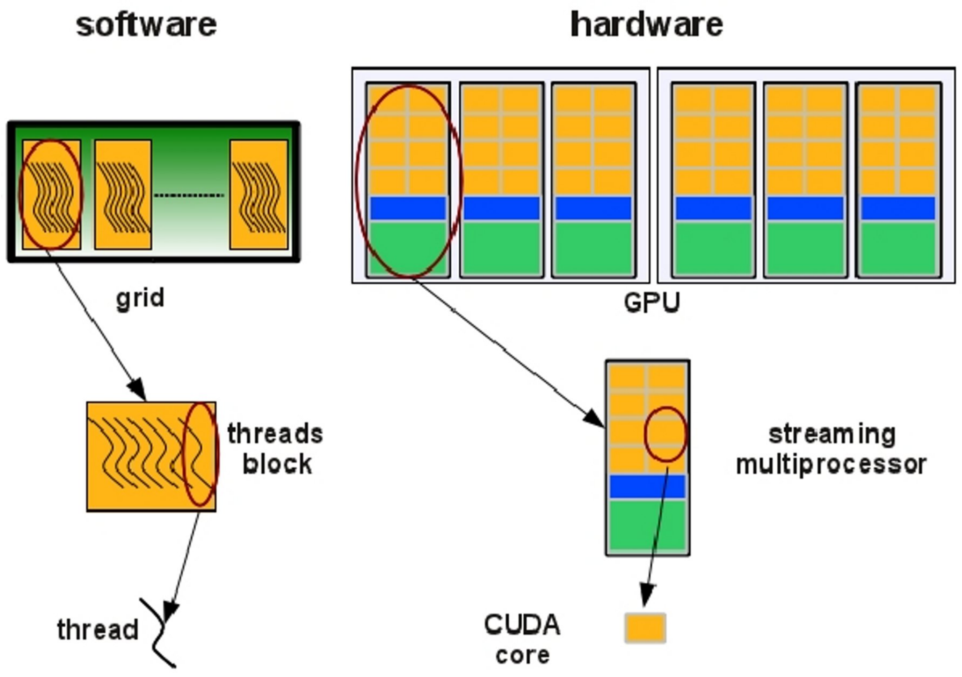


Figure 2.6: **CUDA execution model** Threads are grouped in blocks in a grid. Each thread has a private memory and runs in parallel with the others in the same block.

The memory model is logically partitioned into four regions: global memory, local memory, texture memory, and constant memory. Global memory is available to all threads and is persistent between GPU calls. Local memory is available only to individual threads and is used as a backup when the compiler is unable to fit requested data into the device's registers, in which case registers are said to spill to local memory. Texture memory is read-only with a small cache optimized for manipulation of textures. Constant memory, as the name implies, is also a read-only region which also has a small cache. Finally, host memory (the system's main memory) is available indirectly and relatively slowly to the GPU. This memory space is only available to the GPU when copied over the PCI-Express bus to the GPU's device memory. For details see subsection 2.3.1.2.

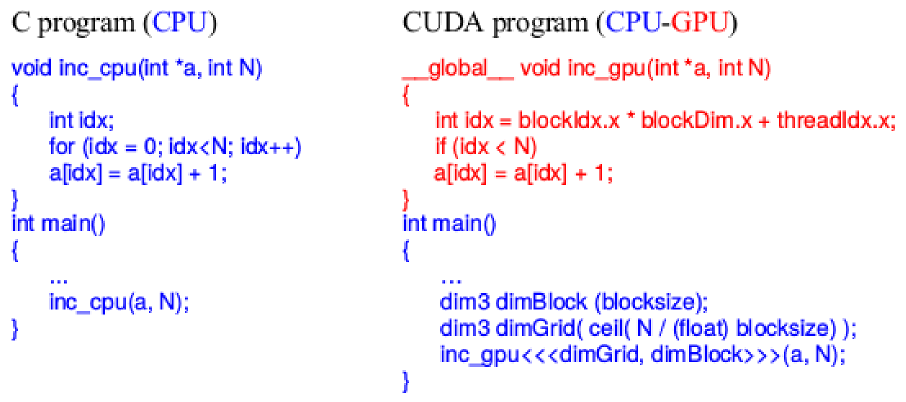
2.3.1.1 Execution model

A GPU computing program written in CUDA consists of function that can run on either the CPU, also called the host, or the GPU which is called the device. The functions executed on the GPU are commonly referred to as CUDA kernels and can be executed in parallel across threads on the device and also asynchronous from the host. This parallel execution is not only bound to the execution of a specific kernel, but also allows the execution of a multiple of different kernels at once, where the number of kernels that can be executed is limited by the amount of SMs on the GPU. To specify if a function should be executed on the host or the device the qualifiers shown in Table 2.3 are used.

Qualifiers	**Callable From**	**Executes On**
__device__	Device	Device
__global__	Host	Device
__host__	Host	Host

Table 2.3: **CUDA Function Qualifiers.** Table shows from where the functions with the specific qualifiers are callable from and from where the functions are executes on. The *__host__* qualifier is usually omitted since it is the default if no other qualifier is specified.

A kernel is executed in parallel across a set of parallel threads, which are grouped into parallel thread blocks. When a kernel is launched it is initiated on the GPU as a grid of parallel blocks with each thread within a thread block executing an instance of the kernel. Threads in a grid are organized into a two-level hierarchy. At the top level, the grid is organized as a two-dimensional array of blocks where all blocks have the same number of threads. At the second level, each thread block is organized as a three-dimensional array of threads. When the host code invokes a kernel, it sets the grid and thread block dimensions via execution configuration parameters (see Figure 2.7).



C program (CPU)

```
void inc_cpu(int *a, int N)
{
    int idx;
    for (idx = 0; idx<N; idx++)
    a[idx] = a[idx] + 1;
}
int main()
{
    ...
    inc_cpu(a, N);
}
```

CUDA program (CPU-GPU)

```
__global__ void inc_gpu(int *a, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N)
    a[idx] = a[idx] + 1;
}
int main()
{
    ...
    dim3 dimBlock (blocksize);
    dim3 dimGrid( ceil( N / (float) blocksize) );
    inc_gpu<<<dimGrid, dimBlock>>>(a, N);
}
```

Figure 2.7: **Increment Array Example in C and CUDA**.

The first parameter of the execution configuration specifies the dimensions of the grid in terms of number of block. The second specifies the dimensions of each block in terms of number of threads. Each such parameter is a *dim3* type, which is essentially a C struct with three unsigned integer fields (x, y and z). Because grids are 2D arrays of block dimensions, the third field of the grid dimension parameter is ignored. Because all threads in a grid execute the same kernel function, they rely on unique coordinates to distinguish themselves from each other and to identify the appropriate portion of the data to process. These threads are organized into two-level hierarchy using unique coordinates – *blockIdx* for block index and *threadIdx* for thread index – assigned to them by CUDA runtime system. The *blockIdx* and *threadIdx* appear as built-in, preinitialized variables that can be accessed within kernel functions. When a thread executes the kernel function, references to the *blockIdx* and *threadIdx* variables return the coordinates of the thread (see Figure 2.7). Additional built-in variables, *gridDim* and *blockDim*, provide the dimension of the grid and the dimension of each block respectively. It is the responsibility of programmer to use these variables in kernel functions so the threads can properly identify the portion of the data to process. This model of programming compels the programmer to organize threads and their data into hierarchical and multidimensional organizations.

Once a kernel is launched, the CUDA runtime system generates the corresponding grid of thread blocks. Each block of the grid is cyclically assigned to a Streaming Multiprocessor (SM) in arbitrary order (see Figure 2.6). Up to 16 blocks can be assigned to each SM in the Kepler architectures as long as there are enough resources to satisfy the needs of all of the blocks. In situations with an insufficient amount of any one or more types of resources needed for the simultaneous execution of 16 blocks, the CUDA runtime automatically reduces the number of blocks assigned to each SM until the resource usage is under the limit. The runtime system maintains a list of blocks that need to execute and assigns new blocks to SMs as they complete the execution of blocks previously assigned to them. One of the SM resource limitations is the number of threads that can be simultaneously tracked and scheduled. In the Kepler device, up to 2048 threads can be assigned to each SM. This could be in the form of 8 blocks of 256 threads each, 16 blocks of 128 threads each, etc. It should be obvious that 32 blocks of 64 threads each is not possible, as each SM can only accommodate up to 16 blocks.

Once a block is assigned to a SM, it is further divided into 32-thread units called warps. The warp scheduler of a SM selects a warp that is ready to execute and issues the instruction to the threads of the warp. The SM maps each thread of a warp to one thread core (see Figure 2.6), so at any given clock cycle, each thread of a warp executes the same instruction, but operates on different elements of the data set in parallel (data-parallel model SIMT). A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken; disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjointed code paths [147].

At any time, the SM executes only a subset of its resident warps for execution. This allows the other warps to wait for long-latency operations without slowing down the overall execution throughput of the massive number of execution units. When an instruction executed by the threads in a warp must for the result of a previously initiated long-latency operation, the warp is not selected for execution. Another resident warp that is no longer waiting for results is selected for execution. If more than one warp is ready for execution, a priority mechanism is used to select one for execution. This mechanism of filling the latency of expansive operations with work from other threads is often referred to as latency hiding. The selection of ready warps for execution does not introduce any idle time into the execution timeline, which is referred to as zero-overhead thread scheduling. With warp scheduling, the long waiting time of warp instructions is hidden by executing instructions from other warps. For example if a warp is waiting for memory access, the scheduler can perform a zero-cost, immediate context switch to another warp.

Threads synchronization

CUDA allows threads in the same block to coordinate their activities using a barrier synchronization function, *__syncthreads()*. When a kernel function calls ensure that all threads in a block have completed a phase of their execution of the kernel before any moves on the text phase. The barrier synchronization function is highly used when threads within a same block must coordinate their communication with each other by writing or reading per-block memory at a synchronization barrier.

Barrier synchronization is not allowed among threads in different blocks. Threads in different blocks can only safely synchronize with each other by terminating the kernel and starting a new kernel for the activities after the synchronization point. Therefore, the CUDA runtime system can execute blocks in any order relative to each other because none of them must wait for each other. This flexibility enables scalable implementations as shown in Figure 2.8.

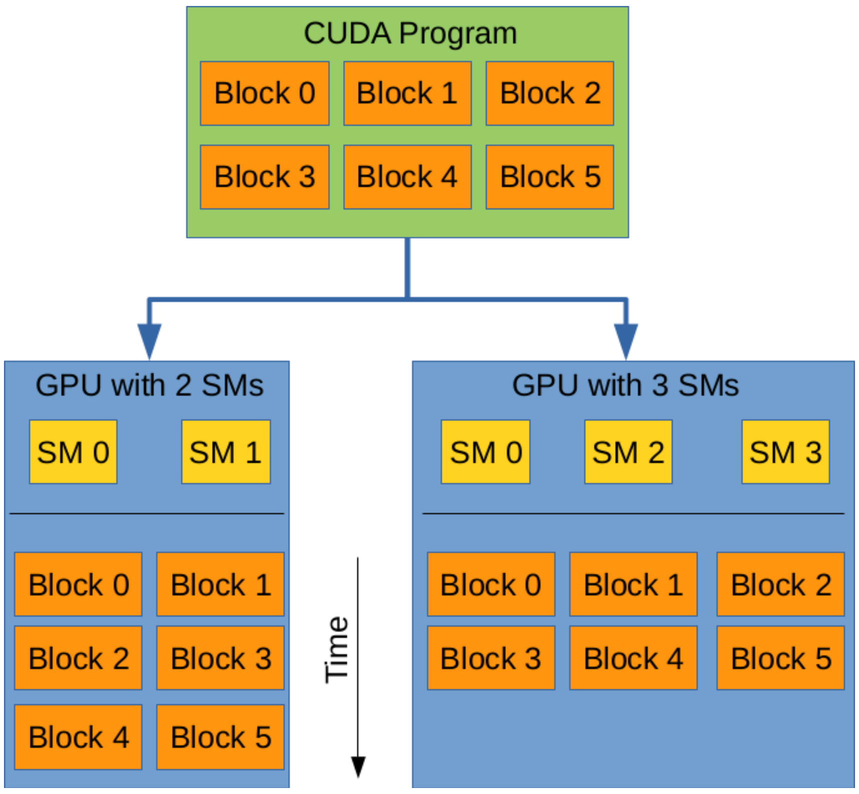


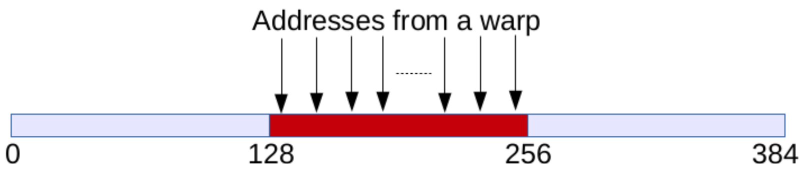
Figure 2.8: **Transparent scalability for CUDA programs**. Image derived from [151].

In a low-cost implementation with only a few execution resources, one can execute a small number of blocks at the same time. In a high-end implementation with more execution resources, one can execute a large number of blocks at the same time. Both execute exactly the same application program with no change to the code. The ability to execute the same application code on hardware with different numbers of execution resources is referred to as transparent scalability, which reduces the burden on application developers and improves the usability of applications.

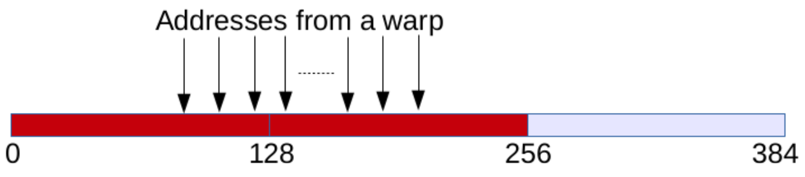
2.3.1.2 Memory model

In addition to the CUDA execution model the CUDA memory model, describing the GPU memory hierarchy and memory access, is also very important to consider when creating GPU computing programs. With traditional CPUs data is stored in a large main memory that has a high-latency memory access. This high-latency memory access can become a performance constraint when executing programs where the computational problem is memory and not compute bound. To hide this latency the data is moved closer to the CPU with the use of fast and very low-latency on-chip caches. The use of these caches is transparent and all memory access is uniform, which means that the programmer does not need to explicitly handle the use of caches. It can however be beneficial to limit the workloads of the program to the size of the caches since this will decrease cache misses and access to the main memory.

Similar to the CPU the GPU also has a large high-latency main memory called the global memory, caches that are called local memory (compute capability 2.0 and above), and registers. However, due to the number of computation cores on a GPU there is a very limited space for caches on the chip, and therefore the hiding of latency must instead be achieved through high bandwidth utilization by concurrently executing threads. The achievable bandwidth utilization is determined by restrictions to the memory access pattern of thread warps which enables the coalescing of memory access into as few accesses transactions as possible (see Figure 2.9). In details when all threads in a warp execute a load instruction, the hardware detects if the threads access consecutive global memory locations. That is, the most favorable access pattern is achieved when the same instruction for all threads in a warp accesses consecutive global memory locations (see Figure 2.9). In this case, the hardware combines, or coalesces, all of these accesses into a consolidated access to consecutive DRAM locations. For example, for a given load instruction of a warp if thread *0* accesses location *N*, thread *1* accesses location *N+1*, thread *2* accesses location *N+2*, etc., then all of these accesses will be coalesced, or combined into a single request for all consecutive locations when accessing the global memory. Such coalesced access allows the DRAMs to deliver data at a rate close to the peak global memory bandwidth.



(a) Coalesced access - all threads access one memory segment



(b) Unaligned sequential addresses that fit into two memory segments

Figure 2.9: **Access pattern to global memory**. Image derived from [150]

In addition the the global memory, caches, and registers, the GPU also has a shared memory, constant memory and a texture memory. The shared memory is a small and fast low-latency on-chip memory available on each SM that has its own address space, and can be accessed by all threads in a thread block allocated to the SM. The shared memory is divided into banks which can be accessed simultaneously by as many threads as it has banks. However, if multiple threads try to access the same banks at the same time bank conflicts can occur, forcing the bank access to be serialized and thereby much slower. Such bank conflicts can be avoided by block synchronization using the *__synchthreads()* barrier function within a kernel. If there are no bank conflicts the shared memory can have a similar performance as registers. Shared memory must be explicitly declared by the programmer and it is commonly used for communication and sharing frequently used data between threads. The constant and texture memory is located on the device (off-chip) alongside the global, local, and constant memory. The constant memory is used to store constant global variables, and the texture memory is traditionally used for graphics textures, but can also be used to store more general structures. The texture memory also has the added benefit of hardware units that perform interpolation when accessing the stored data, clamping and wrapping memory accesses to prevent out-of-bounds faults, and spatial caching.

Figure 2.10 shows an illustration of the CUDA memory model and the connection between memory and GPU grid, thread blocks, threads and the CPU host. As the figure shows the CPU can access the GPUs global, constant, and texture memory. This access is done through the PCI-Express bus and is often viewed as a major bottleneck in GPU computing because of the latency involved. It is therefore recommended to limit the transfer of data between the host and the device, and if possible only at the initialization and end of a GPU programs execution. Figure 2.10 also shows that global, constant and texture memory can be accessed by all the threads in every threads block, but shared memory is restricted to all threads within a thread block and that both local memory and registers are private for each thread.

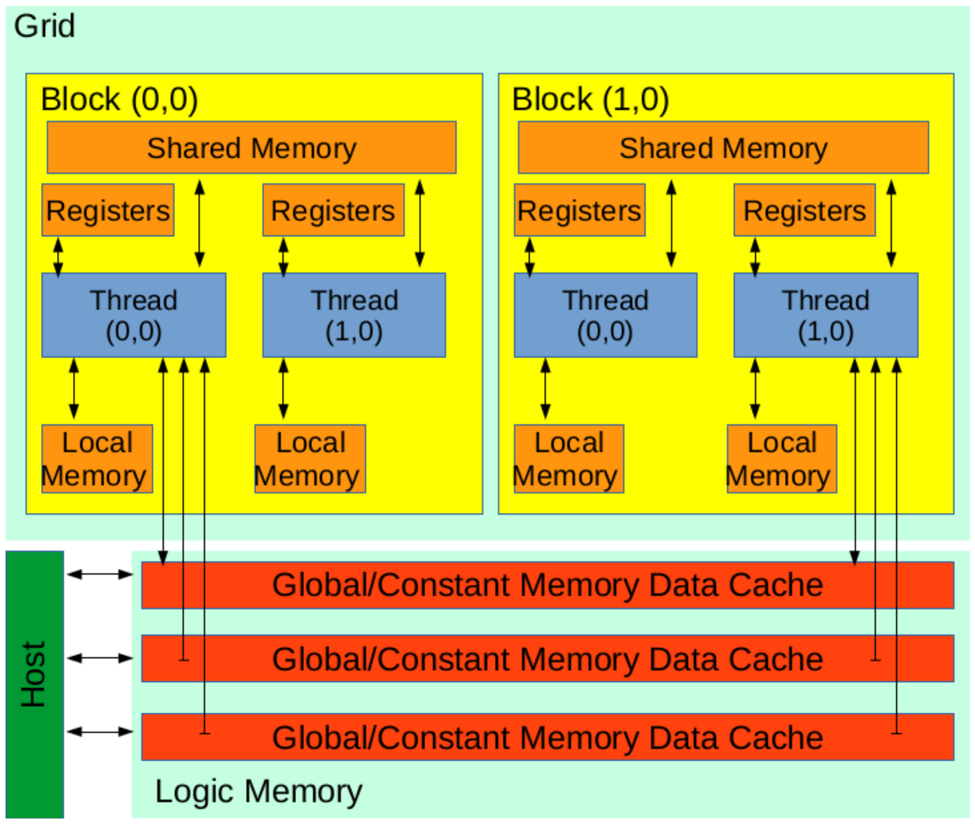


Figure 2.10: **The CUDA memory model**. Image derived from [151]

Similar to the kernel qualifiers also has a set of qualifiers to indicate what type of memory a variable is allocated to. For variables declared in the global scope (outside the scope of a kernel) the qualifier *__device__* is used, for variables in shared memory the qualifier *__shared__*, and for constant variables the *__constant__* qualifier.

Allocation of memory on the device from the host is possible by using the CUDA function *cudaMalloc()*, and freed by using *cudaFree()*. Memory can also be copied by using the CUDA function *cudaMemcpy()* with a keyword defining if the data should be copied from host to host, host to device, device to host, or device to device.

2.3.2 OpenCL

OpenCL is an open standard targeted for general-purpose parallel programming on different types of processors. The goal of OpenCL is to provide software developers a standard framework for easy access to heterogeneous processing platforms. The focus is to enable parallel programming on any hardware that supports parallel execution, including GPUs, CPUs, DSPs, and FPGAs. The OpenCL standard specifies a set of API and a programming language based on C. For the purpose of this thesis it is only necessary to describe the OpenCL concepts rather than the technical details. The technical details of the OpenCL framework can be found in the OpenCL specification [141].

Under the OpenCL programming model, computation can be done in data parallel, task parallel, or a hybrid of these two models. The main focus of the OpenCL programming model is the data parallel model, where each thread (called work-item in OpenCL) works on a data item - effectively implementing SIMD. The task parallel model can be realized by enqueuing the execution of multiple kernels, where only one work-item for each kernel is created. Even though some of the GPUs support this model, this is highly inefficient model for GPUs. It is possible to have a hybrid model where multiple kernels each with multiple work-items are enqueued for execution at the same time.

OpenCL and CUDA are similar in their execution and memory model when used with a GPU, but differ in how kernels are created. In OpenCL kernels are created in separate program files that are collected in program objects that are compiled at runtime into kernel objects which are passed to the device driver for optimization. The OpenCL syntax is also generally a bit more verbose then the CUDA syntax and uses different terminology for execution constructs and components. The different terminology used in CUDA an OpenCL is reported in Table 2.4).

The OpenCL programming model is based on a virtual architecture model mapped on really device only during phase compiling. Virtual architecture model called OpenCL platform model is composed of one or more Compute Units (CU) that correspond to CUDA streaming multiprocessors (see Figure 2.11). However, a CU can also correspond to CPU cores or other types of execution units in compute accelerators such as DSPs and FPGAs. Each CU, in turn, has one or more Processing Elements (PE), which correspond to the cores in CUDA (see Figure 2.11).

CUDA	OpenCL	Description
SM	Compute Unit	A cluster of Core
CUDA Corer	Processing Element	Executing a single instruction
Thread	Work-item	Single instance of a kernel
Thread Block	Work-group	Group of work-items/threads
Kernel	Kernel	Parallel code
Grid (of thread blocks)	NDRange	Array of threads blocks
Global Memory	Global Memory	main memory
Shared Memory	Local	Memory Shared within a work-group
Local Memory	Private Memory	Thread memory

Table 2.4: **CUDA vs. OpenCL terminology.**

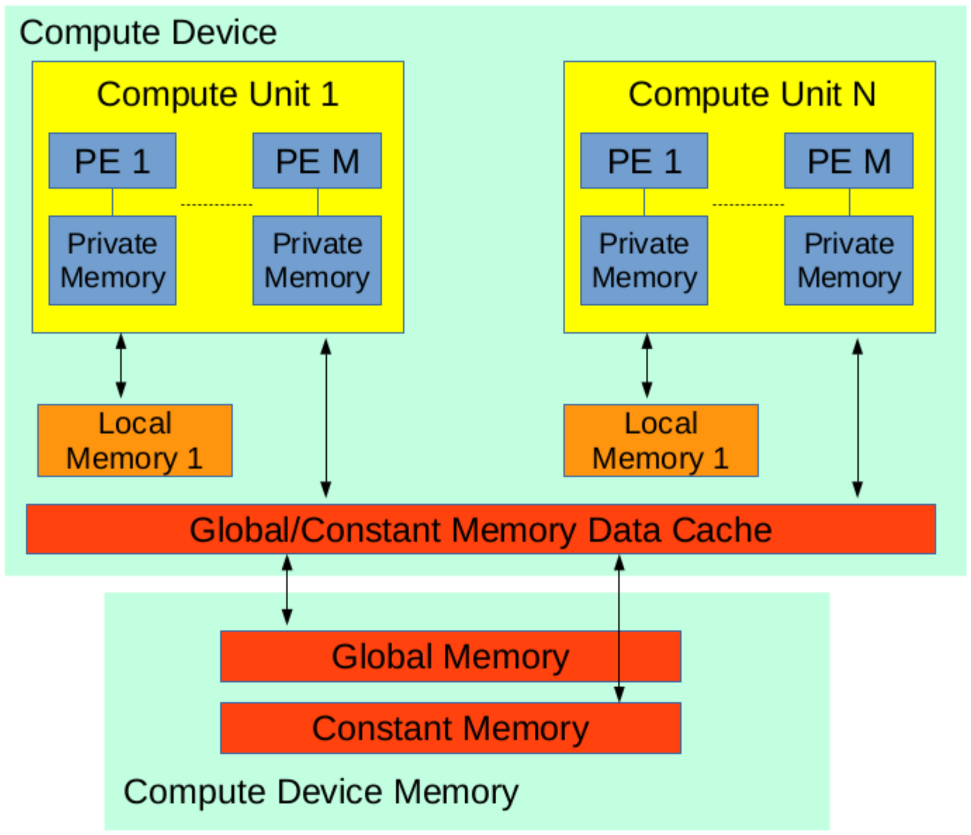


Figure 2.11: **OpenCL platform model**. Image derived from [95]

Execution model

Like for CUDA, the execution model of an OpenCL application has two components. One part, called the kernel, executes on one or more devices, and the other part that executes on the host that manages the execution of kernels. In OpenCL, devices are managed through *contexts*. In order to manage one or more devices in the system, the OpenCL programmer first creates a *context* that contains these devices. Then, to submit work for execution by a device, the host program must create a command queue for the device. Once a command queue is created, the host code can perform a sequence of API function calls to insert a kernel along with its execution configuration parameters into the command queue. When the device is available for executing the next kernel, it removes the kernel at the head of the queue for execution. Therefore, using the OpenCL API, the host part of the application creates a context object and the other objects under it (i.e. kernel object, program object, memory objects, and command queues object). Each device in the context has an associated command queue, and kernel execution and memory transfer are coordinated using the command queue. There are three types of commands that can be issued. Memory commands are mainly used to transfer memory between the host and the device. Kernel commands are issued to start the execution of kernels on the device. Synchronization commands can be used to control the execution order of the commands. Once the commands are scheduled on the queue, there are two possible execution modes. The commands can be executed in-order, meaning the previous command on the queue must finish execution for a command to start execution. The other option is for the commands to execute out-of-order, where commands do not wait for previously queued commands to finish. However, explicit ordering can be enforced in an out-or-order queue by synchronization commands.

A virtual N-dimensional indexed space is defined for the execution of the kernel, and one kernel instance (i.e. a work-item) is executed for each point in this indexed space ($1 \leq N \leq 3$). All the work-items (threads) execute the same code, however, they usually work on different data and their execution path through the code can diverge. Each work-item is assigned a global ID that is unique across the indexed space. Equal numbers of work-items are grouped together to form work-groups (thread blocks), with all the work-groups having the same dimensions. The work-item in a work-group has a unique ID, and has also access to shared local memory (see Figure 2.11). It is important to note that with proper device support, the total number of work-items can be much greater than the number of processing elements present in a device. Through API calls an application can find out the maximum number of work-items a device supports.

Memory model

The host manages the kernels and the memory objects under a *context* through command queues. The memory objects are visible to both the host and the kernels, and used to transfer data between the host and the device. The host creates memory objects, and through the OpenCL API allocates memory on the device for them.

The memory model in OpenCL is divided into four types (see Figure 2.11):

- Global Memory: all work-items have read-write access to this memory region. Usually the input data for the work-items are written to this region by the host, and the computed output data is written there by the work-items.
- Constant Memory: this is a read-only global memory accessible to all work items. The host part of the application allocates and initializes this memory region.
- Local Memory: it is the local memory of a work-group. All the work-items in a work-group shares this memory region. This memory allows work-items to communicate with each other within a work-group.
- Private Memory: this memory region represents the local variables of the kernel instance. Each work-item has its own copy of the local variables and they are only visible to the work-item.

2.3.3 OpenACC Version 1.0

Both the CUDA and OpenCL programming models feature a function rich and low level API for GPU programming, giving the programmer fine grained control over the execution and memory management of a GPU program. Even though this fine grain control can enable the programmer to create very fast and efficient programs, it can also make it difficult to program complicate software design, and tie code a specific device or vendor. As a solution to this, recent approaches try to improve programmability with an abstraction to GPU programming by making the compiler responsible for many of the low-level programming tasks through compiler directive-based high-level APIs such as OpenACC.

OpenACC is a open programming standard developed by NVIDIA, *The Portland Group*, CAPS, and Cray, for parallel computing on heterogeneous systems using compiler directives. The OpenACC standard was first released as version 1.0 [145] in November 2011 and it describes a API for GPU programming that includes compiler directives, library routines and environment variables. In March 2013 draft for OpenACC version 2.0 [146] was released and is currently open for public comment. The OpenACC API and its directives can currently be used in combination with the C, C++, and Fortran programming languages and aims to be portable across operating systems, host CPUs and accelerators.

The primary goal of OpenACC is to simplify programming applications on heterogeneous systems that are composed of general purpose processors, such as CPUs, with attached accelerator devices, such as GPUs. This simplification is done by using directives to indicate what loops or region of code should be executed on the accelerator in parallel and letting the compiler create the needed target code. This gives the programmer a simple start to GPU programming without the need to think as much about the execution and memory model, but the OpenACC API also has features for a more explicit control of the program execution. It is also important to note that the use of directives does not make parallel programming easier in general, as it is still up to the programmer to identify parallel regions or modify existing code to make it suitable for parallelism.

Since OpenACC is only a standard describing the API, it is up to developers to create compilers with OpenACC support and decide how the GPU program should be created. A common approach is to perform a source-to-source translation of the regions marked by the OpenACC directives to CUDA or OpenCL target code built for the appropriate architecture.

Execution model

OpenACC is generally aimed at heterogeneous systems that has a host processor, such as a CPU, and an attached accelerator device, such as a GPU. On such systems a program is usually executed on the host until a compute intensive region that can be executed in parallel is reached. Such regions are typically called parallel regions and usually contains one or more work-sharing loops that can be executed as a kernel. When such a region is reached the host can offload the region to the accelerator for fast parallel execution. After the execution is finished the results are transfer back to the host again. Because of this behavior OpenACC has a host-directed execution model.

The host must allocate device memory, initiate data transfer, send code to the device, pass needed arguments to the parallel region, wait for the execution to complete, transfer results back again, and deallocate the device memory. Most of this process is done by the OpenACC compiler, but it can also be specified in more detail by the programmer.

In OpenACC accelerators executes parallel regions that are similar to CUDA kernels, which typically contains single or nested work-sharing loops. These regions are indicated by the use of directives *#pragma acc kernels* or *#pragma acc parallel*, that can either be executed synchronously or asynchronously.

The parallel regions can be executed with three levels of parallelism, called gang, worker, and vector. Generally these are mapped to an architecture that is a collection of processing elements, where there is one or more processing element per node, each processing element is multithreaded, and each processing element can execute vector instructions. The mapping between gang, worker, and vector is implementation-dependent, but typical mappings to a CPU might be that gang is the number of CPUs, worker is the number of CPU cores, and vector is the number of SIMD instructions per core. On CUDA a typical mapping is that gang corresponds to the number of thread blocks in a grid, vector is the number of threads per thread block, and worker is locked to the warp size. If no specific number of gangs, workers, and vectors is specified the compiler will select values that matches the size of the computing domain and the underlying hardware architecture. However, there are no guarantees that the values selected by the compiler result is the optimal choice, and in some cases it might be wise to tune the code to fit particular target architecture.

Memory model

Since OpenACC is mainly targeting heterogeneous systems with a host CPU and a attached accelerator device, such as an GPU, the memory of the host and accelerator is usually completely separated. All data allocation and data movement between host memory and accelerator device memory must be performed by the host through runtime library calls. Besides the host and device memory OpenACC also has the notion of private and shared memory, where the private memory is usually hardware-managed caches and the shared memory is software-managed cache like the shared memory on CUDA. All of these memory concepts are implicit and managed by the compiler, based on compiler directives declared by the programmer.

The directives can describe the allocation and data movement by telling the compiler that memory should be allocated in device memory, data should be copied from the host to the device and back again, only copied from the host to the device, only copied to the host from the device, or that the data already exists in the device memory.

In a typical parallel region data is copied to the device from the host at the start of the region, and copied back to the host when the region ends. However, when there are multiple parallel regions that all work on the same data in sequence the copying of data back and forth between each region might slow down performance. It is therefore possible to use the OpenACC runtime API to allocate memory at the start of the program and pass the device memory pointers to the parallel regions, allowing the data to stay on the device through the programs execution. In addition the compiler also creates barrier constructs to prevent simultaneous access to the same memory that might result in memory coherence issues.

2.4 Discussion and Conclusions

Nowadays, many scientific and engineering applications tend towards using more specialized high-performance processor technologies over conventional CPU processors. Some of the graphics computing solutions have been expanding their intended use in current and future graphic systems, as well as in high-performance computing systems (Such as, IBM Cell, Intel Xeon, and NVIDIA and AMD GPUs).

Semiconductor capability and advances in fabrication process have increased for both CPUs and HPC platforms, but the main growth disparity is due to architectural differences: CPUs have large caches and they are optimized for high performance on sequential code or coarse-grained multi-thread, focusing branch prediction and out-of-order execution. On the other hand, GPUs that focus on highly parallel general purpose computations achieve higher arithmetic intensity. Thus, it increases the system performance without compromising the overall system power consumption.

Table 2.5 shows a comparison among the best CPUs, HPC co-processors, and GPU architectures. The comparison is very complex because all of them have different features, but if we compare them in terms of peak performance and performance per watt, GPUs perform most efficiently. However, Cell/BE is cheaper than other accelerators. The AMD GPU hardware implementations are architecturally different from the NVIDIA's one but, on average, performances are very similar. Making the best choice of platforms greatly depends on applications. If an application contains a massive amount of independent data and still needs to gain performance while maintaining the power consumption low, then GPU is a better choice. Since the GPU is a massively multi-threaded parallel machine with high-end shared memory, it is a great choice for fine-grained parallelism. However the main limitation and difficulty is that the developer must have an in-depth knowledge of GPU programming and hardware. For this reason the graphics card manufacturers have developed a new

	CPU	**Coprocessor**		**GPU**	
Vendor	**Intel**	**Intel**	**IBM**	**NVIDIA**	**AMD**
Product name	Xeon E5-2687v3	Xeon Phi 3120A	PowerX Cell 8i	Tesla k80	Radeon R9 295X2
Peak performance [TF]	0.39(SP)	0.71(SP)	0.20(SP) 0.10(DP)	8.74(SP) 2.91(DP)	11.46(SP) 1.43(DP)
Wattage (TDP)	160W	300W	260W	300W	500W
Performance per watt[GF/W]	2.4(SP)	2.36(SP)	0.76(SP) 0.38(DP)	29.13(SP) 9.70(DP)	22.90(SP) 2.86(DP)
Memory bandwidth [GB/s]	68	240	32	240x2	320x2
Price	2145$	1695$	1100$	5000$	1499$

Table 2.5: **Comparison among CPU, HPC Coprocessors, and GPU architectures**.

unified graphics and computing GPU architecture and introduce new programming models dedicated to general-purpose computations, which provide low-level or direct access to the multi-threaded computational resources and associated memory bandwidth of GPUs. As to programming models, the vendor-independent OpenCL is used by both AMD and NVIDIA, which also uses its proprietary language CUDA. OpenCL draws heavily on CUDA in the areas of supporting a single code base for heterogeneous parallel computing, data parallelism, and complex memory hierarchies. On the other hand, it has a more complex platform and device management model that reflects its support for multi-platform and multi-vendor portability. Whereas the OpenCL standard is designed to support code portability across devices produced by different vendors, such portability does not come free. OpenCL programs must be prepared to deal with much greater hardware diversity and thus will exhibit more complexity. As a result, a portable OpenCL code may not be able to achieve its performance potential on any of the devices. In particular, as shown in [89], in the specific case of NVIDIA GPUs, CUDA kernel execution is consistently faster than OpenCL's, despite the two implementations run nearly identical code. The main reason for CUDA outperforming OpenCL is that CUDA builds kernels using a compiler thus obtaining a better optimized code than OpenCL, which builds them at runtime. Moreover, the most recent CUDA releases offer more features than OpenCL. For example, CUDA allows SM register sharing among threads and implementation of recursive algorithms is only available on OpenCL starting from OpenCL 2.0 and is not supported in NVIDIA drivers.

The bioinformatics applications detailed in the following chapters have been parallelized on NVIDIA GPUs using CUDA, as the NVIDIA GPUs were among the most performant at the time they were bought and CUDA outperforms OpenCL on NVIDIA GPUs.

Chapter 3

G-SNPM: A GPU-based SNP Mapping Tool

Single Nucleotide Polymorphism (SNP) genotyping analysis is very susceptible to SNPs chromosomal position errors. SNPs mapping data are provided along the SNP arrays without any necessary information to assess in advance their accuracy. Moreover, these mapping data are related to a given build of a genome and need to be updated when a new build is available. As a consequence, researchers often plan to remap SNPs with the aim to obtain more up-to-date SNPs chromosomal positions.

Specialized tools as LiftOver, AssemblyConverter, and the NCBI Genome Remapping Service have been devised to project the coordinates of genomic regions from a given build to another build of a genome. These tools are very useful to update chromosomal coordinates between different reference sequences; however they might be unable to perform a given conversion between different assemblies. In fact, these tools typically allow only a limited set of assembly-assembly conversion combinations. Then, it might be impossible to use them to update SNPs positions on a given build of a genome. Moreover, new positions obtained using these tools are strongly related to the initial positions provided by the vendor. Unfortunately, if a SNP has been wrongly mapped by the vendor, the error will be spread to the updated position. Finally, these tools are specialized to convert coordinates from a build to another and do not permit to remap a SNP against the same reference build to look for discrepancies with the vendor positions. Researchers use also tools as BLAST or BLAT to analyze SNP probe positions and/or to update them to the genome or to the transcriptome. However, using tools as BLAST or BLAT to update thousands or millions of SNPs is a very expensive task in terms of computing time.

G-SNPM is a tool devised to map a short sequence representative of a SNP against a reference DNA sequence in order to find the physical position of the SNP in that sequence. In G-SNPM each SNP is mapped on its related chromosome by means of an automatic three-stage pipeline. In the first stage, G-SNPM uses the GPU-based short-read mapping tool SOAP3-dp to parallel align on a reference chromosome its related sequences representative of a SNP. In the second stage G-SNPM uses another short-read mapping tool to remap the sequences unaligned or ambiguously aligned by SOAP3-dp (in this stage SHRiMP2 is used, which exploits specialized vector computing hardware to speed-up the dynamic programming algorithm of Smith-Waterman). In the last stage, G-SNPM analyzes the alignments obtained by SOAP3-dp and SHRiMP2 to identify the absolute position of each SNP. Used to remap the SNPs of some commercial chips, G-SNPM has been able to remap without ambiguity almost all SNPs. Based on modern GPUs, G-SNPM provides fast mappings without worsening the accuracy of the results. G-SNPM can be used to deal with specialized genome wide association studies, as well as in annotation tasks that require to update the SNP mapping probes.

3.1 Introduction

GWAS have shown that genetic variants are often responsible of traits expressed in phenotypes. Genetic variants may be associated with the cause (e.g., [132]) or with the predisposition (e.g., [186]) of a disease, and may determine individual drug responses (e.g., [46]). SNPs are the most common type of genetic variant in human genome. More than 10 million SNPs are estimated to be in the human genome [28]. The scientific community has placed a great interest in the analysis of SNPs, widely exploiting their knowledge in GWAS [188, 171, 72]. Hence, different public resources have been devised to share their knowledge (e.g., dbSNP [174], the International HapMap Project [57], the 1000 Genomes Project [8]), as well as specialized tools for SNP calling (e.g. MAQ [111], SOAPsnp [112], SNVMix [63]) and SNP analysis (e.g., FAST-SNP [202], SNPLims [154], SNPInfo [197], SNPranker 2.0 [136]). In this context, SNP genotyping arrays represent an important tool for genetic analysis. It should be pointed out that the reliability of the genotype-phenotype associations that may be discovered analyzing SNPs is strongly related to the accuracy of the data that describe them. In particular, SNP genotyping analysis is very susceptible to SNPs chromosomal position annotation errors. In fact, wrongly mapped SNPs may in some cases affect data analysis and lead to erroneous conclusions. An interesting study about wrongly mapped SNPs in commercial SNP chips, and on their possible functional consequences, has been presented in [45]. In this work, SNPs of various chips have been remapped using highly sensitive alignment parameters against their reference genomes, with the goal to highlight discrepancies between the found genomic positions and those provided by the chip vendors. These discrepancies highlighted that more sensitive aligner parameters should be used to achieve an accurate alignment instead of retrieving a partial best alignment with extra SNPs, indels or less SNP flanking sequence aligned. This suggests that researchers should closely examine how mapping data have been obtained, with the goal of analyzing their accuracy and if necessary taking into account the opportunity to update them. However, mapping data are provided to the users along the SNP chips, omitting any information about the algorithm and the parameter settings used to obtain them. Then, meticulous researchers often plan to remap the SNPs to obtain more accurate chromosomal positions before performing association studies. In general, when a new build of a genome is available it might be productive to re-analyze the data of old genotyping experiments while exploiting the new reference sequences. In this case, as the mapping data of SNP chips are related to a given build of the genome under consideration (irrespective of their original accuracy), chromosomal positions need to be updated according to the newest build. Moreover, in genotyping analysis often researchers need to merge genetic datasets coming from different genotyping platforms, which in turn use different sets of SNPs to represent genetic polymorphisms. To this end, it is necessary to know the exact position of a SNP in a chromosome and update this information when new builds of the reference genome are available.

Specialized tools as LiftOver [4], AssemblyConverter [1], and the NCBI Genome Remapping Service [5] have been devised to project the coordinates of genomic regions from a given build to another build of a genome. These tools are very useful to update chromosomal coordinates between different reference sequences; however they might be unable to perform a given conversion between different assemblies. In fact, these tools typically allow only a limited set of assembly-assembly conversion combinations. Then, it might be impossible to use them to update SNPs positions on a given build of a genome. Moreover, new positions obtained using these tools are strongly related to the initial positions provided by the vendor. Unfortunately, if a SNP has been previously wrongly mapped by the vendor, the error will be spread to the updated position. Finally, these tools are specialized to convert coordinates from a build to another and do not permit to remap a SNP against the same reference build to look for discrepancies with the vendor positions.

Researchers use tools as BLAST [15] or BLAT [91] to analyze the SNP probes positions and/or to update them to the genome or to the transcriptome. For instance, some researchers highlighted that many of the Illumina probes have unreliable original annotations and defined a pipeline that exploits both BLAST and BLAT to perform complete genomic and transcriptomic re-annotation of the probe sequences [23]. AffyProbeMiner [117] is a platform-independent tool that uses all RefSeq mature RNA protein coding transcripts and validated complete coding sequences in GenBank [25] to regroup the individual probes into consistent probe sets to remap them to the correct sets of mRNA transcripts exploiting a local implementation of the BLAT server. The Bioconductor [56] package named *altcdfenvs* has been used to investigate how probes found on Affymetrix microarrays were matching on more recent curated collections of human transcripts. Experiments showed that not all the probes matching a reference sequence were consistent with the grouping of probes by the manufacturer of the chips [55]. However, using tools as BLAST or BLAT to update thousands or millions of SNPs is a very expensive task in terms of computing time.

In this work, an improved version of G-SNPM (standing for GPU-SNP Mapping) [155] was presented, an accurate and very fast tool devised to cope with the problem of updating SNPs chromosomal positions. Written in Python, G-SNPM is mainly based on the SOAP3-dp [122] short-read mapping tool to exploit the computation power of modern GPUs.

G-SNPM is available at the following address http://www.interomics.eu/sp1-wp2.

3.2 Related work

Several tools have been devised to perform short-read mappings. Without aiming to be exhaustive, some of the most popular solutions are cited, as MAQ [111], RMAP [179, 178], Bowtie [104], BWA [108], CloudBurst [169], and SHRiMP2 [166, 40]. MAQ maps short sequence reads to a reference genome by calculating the probability of a read alignment to be correct, and consensus genotype calling with a model that incorporates correlated errors and diploid sampling. It supports gapped alignment and can align reads up to 128bp. RMAP uses quality scores to provide accurate ungapped alignments. In so doing, it exploits two different mapping criteria. A first criterion is based on a simple count of mismatches between a read and the aligned genomic region, while a second criterion makes use of the base-call quality scores. By manipulating the quality-score cutoff, the second criterion provides another means of adjusting sensitivity and specificity. In particular, it allows positions to contribute when they are of high-quality, but not be penalizing if they are low-quality. Bowtie is a memory-efficient short-read aligner that exploits the Burrows-Wheeler Transform (BWT) to index the genome allowing only ungapped alignments. BWA is another tool that exploits the BWT to index the reference sequences. It can also provide gapped alignments, while Bowtie cannot. It consists of three algorithms (i.e., BWA-backtrack, BWA-SW and BWA-MEM), devised to perform both short and long read alignments. CloudBurst is a parallel seed-and-extend read-mapping tool able to align reads with a specified number of differences, including both mismatches and indels (insertions/deletions). It exploits the open-source Hadoop [190] implementation of MapReduce [41] to parallelize the execution using multiple computing nodes. SHRiMP2 exploits specialized vector computing hardware to speed-up the Smith-Waterman [180] dynamic programming algorithm. It is a multi-core short-read mapping tool that enables the alignment of reads with extensive polymorphism and sequencing errors. A comparative study aimed at assessing the accuracy and the runtime performance of different state-of-the-art Next-Generation Sequencing (NGS) read alignment tools highlighted that among all SOAP2 [113] is the one that showed the higher accuracy [165]. Exhaustive reviews of the tools cited above can be found in the literature (e.g., [22]).

In general, the mentioned solutions exploit some heuristics to find a good compromise between accuracy and running time. Recently, GPU-based solutions have been proposed to cope with different bioinformatics problems [125, 203, 119, 175]. GPUs have also been exploited to cope with the exponentially increasing throughput of NGS. In particular, the computational power of these hardware accelerators is helping researchers to speed the short-read mapping process without compromising accuracy and sensitivity. Lately, the GPU-based short-read mapping tools Barracuda [96], CUSHAW [120], SOAP3 [116] and SOAP3-dp have been proposed to the scientific community. Experimental results show that SOAP3, which is the GPU evolution of SOAP2, outperforms the popular tools BWA and Bowtie. When tested to align millions of 100-bp read pairs to the human genome, it resulted at least 7.5 times faster than BWA, and 20 times faster than Bowtie. Moreover, SOAP3 does not exploit heuristics and it is able to align correctly slightly more reads than BWA and Bowtie. SOAP3 is able to align a read to a reference sequence with up to four mismatches while it does not support gapped alignments. Lately, the SOAP3 research team released SOAP3-dp, a new version of the aligner that exploits dynamic programming to support gapped alignments. Compared with BWA, Bowtie2 [103], SeqAlto [140], GEM [128], and the previously mentioned GPU-based aligners, SOAP3-dp is two to tens of times faster, while maintaining the highest sensitivity and lowest false discovery rate on Illumina reads with different lengths. Table 3.1 summarizes the described tools.

Name	**Mapping Strategy**	**Indels Support**	**Quality evalutation**	**GPU-based**
Barracuda	BWT-based indexing of the reference	Yes	Yes	Yes
BWA	BWT-based indexing of the reference	Yes	Yes	No
Bowtie	BWT-based indexing of the reference	No	Yes	No
CUHSHAW2	BWT-based indexing of the reference	Yes	Yes	Yes
CloudBurst	Hash the reads	Yes	No	No
MAQ	Hash the reads	No	Yes	No
RMAP	Hash the reads	Yes	Yes	No
SHRiMP2	Hash the reads	Yes	Yes	No
SOAP2	BWT-based indexing of the reference	Yes	Yes	No
SOAP3	BWT-based indexing of the reference	No	No	Yes
SOAP3-dp	BWT-based indexing of the reference	Yes	No	Yes

Table 3.1: **Short-read mapping tools. A summary of some of the most popular short-read mapping tools.**

3.3 Methods

G-SNPM is a tool that maps a sequence representative of a SNP against a reference sequence in order to find the absolute position of the SNP in that sequence. For genotyping analysis a SNP is represented by a oligonucleotide probe for each possible allele. In turn, these probes can be synthetically described by a regular expression obtained by combining the flanking sequences of a SNP with a grouping construct that represents its possible alleles (e.g., GCACTCTCACATG-GATTAGGGAATTA[CG]ATGCAGACCTCCTGCACAACTGCCC). Since public repositories as dbSNP provide short and fixed length flanking sequences, it has been assumed that typically the probes used to design a SNP chip are represented by short sequences. Starting from this consideration, a short-read mapping tool could be successfully used to cope with the SNP mapping task.

In the following of this section, firstly existing state-of-the-art short-read mapping tools are introduced. Then, the strategy used is outlined, devised to deal with SNP mapping problems. Successively, the adopted alignment constraints are discussed. Finally, the minimal hardware and software equipment required to use G-SNPM are briefly resumed.

3.3.1 The implemented strategy

As previously seen, a SNP can be synthetically represented by means of a regular expression R that uses a single grouping construct to describe the possible alleles. However, short-read mapping tools are not designed to work with sequences described by a regular expression with specialized constructs. Then, two trivial approaches could be used to map a SNP with a short-read mapping tool. As for the former approach (see Figure 3.1), the probe sequences related to the alleles of a given SNP are dealt with separately in the alignment process. In other words, each probe sequence is aligned against a reference sequence independently from the others using the same mapping tool and identical setting parameters. After that sequences have been aligned, results are merged and analyzed to detect and eventually update the SNPs mapping positions. As for the second approach (see Figure 3.2), the probe sequences related to the alleles of a given SNP are dealt with simultaneously in the alignment process. To this end, a single sequence must be used to represent the probes related to a SNP. This sequence can be obtained by substituting the grouping construct in R that describes the possible alleles with a aNy symbol that represents any possible nucleotide. In so doing, the expressiveness of the new sequence increases with respect to that of the starting one, while its information content decreases. In this case, results obtained by aligning the new sequence against a reference sequence must be analyzed to filter out false positive alignments: i.e., those alignments for which the aNy symbol that represents the SNP does not match with one of the possible alleles for that SNP. Only after this step alignments can be analyzed to update SNPs mapping positions. This approach can significantly reduce the computational load needed to perform the alignment task. For instance, for biallelic SNPs it will be almost halved with respect to the first approach. Basically, G-SNPM uses this approach to align a sequence representative of a SNP by means an automatic three stage pipeline (see Figure 3.3).

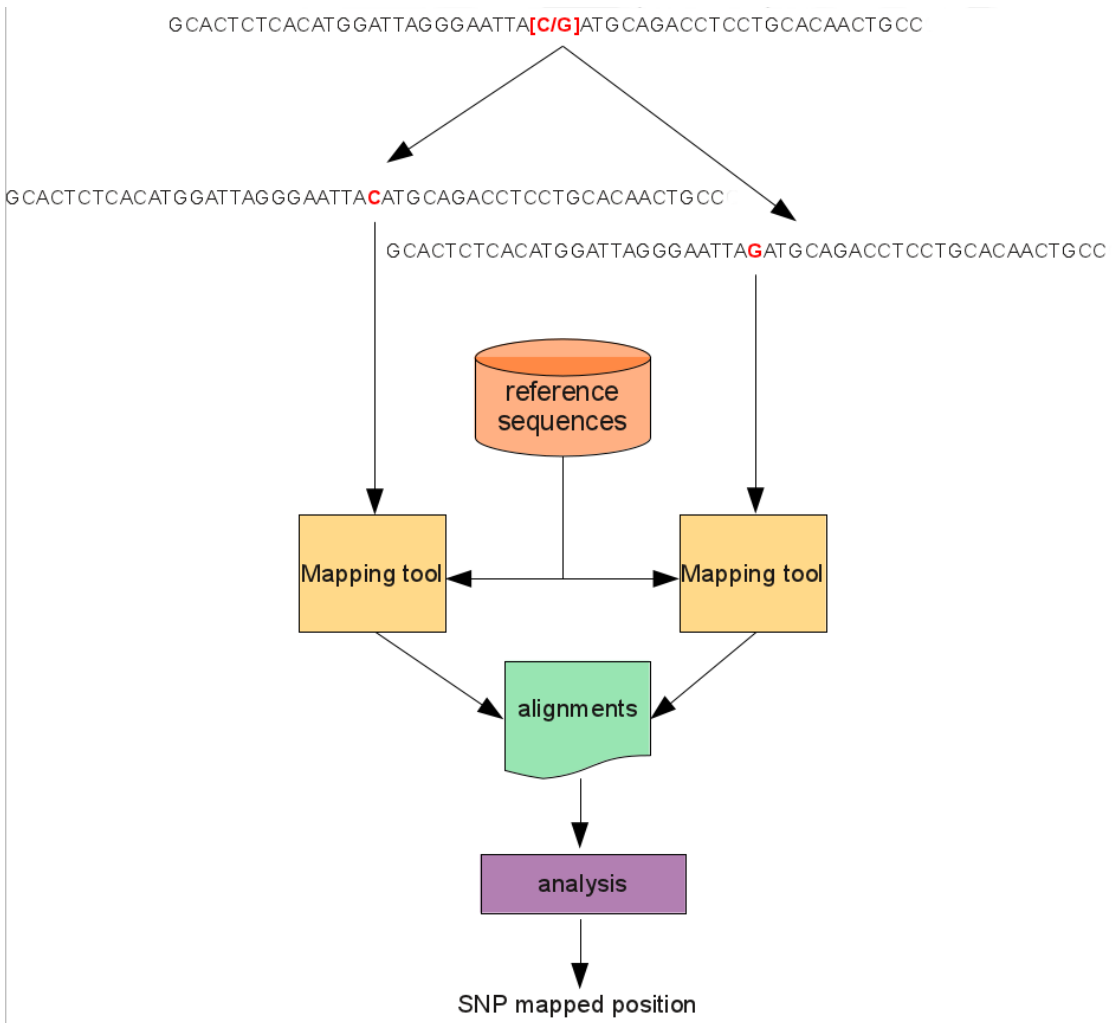

GCACTCTCACATGGATTAGGGAATTA[C/G]ATGCAGACCTCCTGCACAACTGCC
GCACTCTCACATGGATTAGGGAATTACATGCAGACCTCCTGCACAACTGCC
GCACTCTCACATGGATTAGGGAATTAGATGCAGACCTCCTGCACAACTGCC
reference sequences
Mapping tool
Mapping tool
alignments
analysis
SNP mapped position

Figure 3.1: **Use of two sequences to represent a SNP: two sequences are separately aligned for a SNP**. After the alignment, results are analyzed to calculate the absolute position of the SNP.

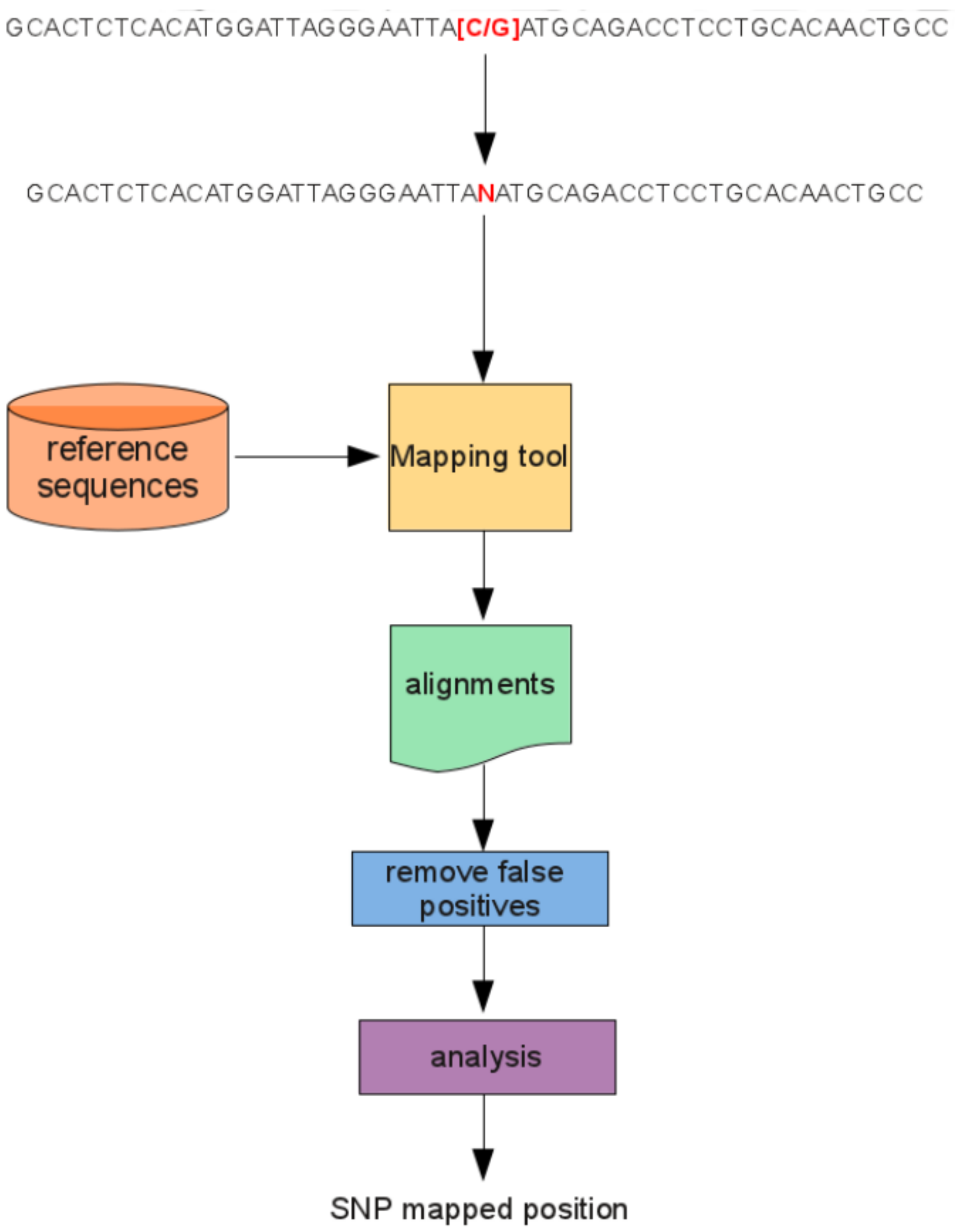

GCACTCTCACATGGATTAGGGAATTA[C/G]ATGCAGACCTCCTGCACAACTGCC
GCACTCTCACATGGATTAGGGAATTANATGCAGACCTCCTGCACAACTGCC
reference sequences
Mapping tool
alignments
remove false positives
analysis
SNP mapped position

Figure 3.2: **Use of two sequences to represent a SNP: only a sequence is aligned for a SNP**. After the alignment results are analyzed to remove those false positives and to calculate the absolute position of the SNP.

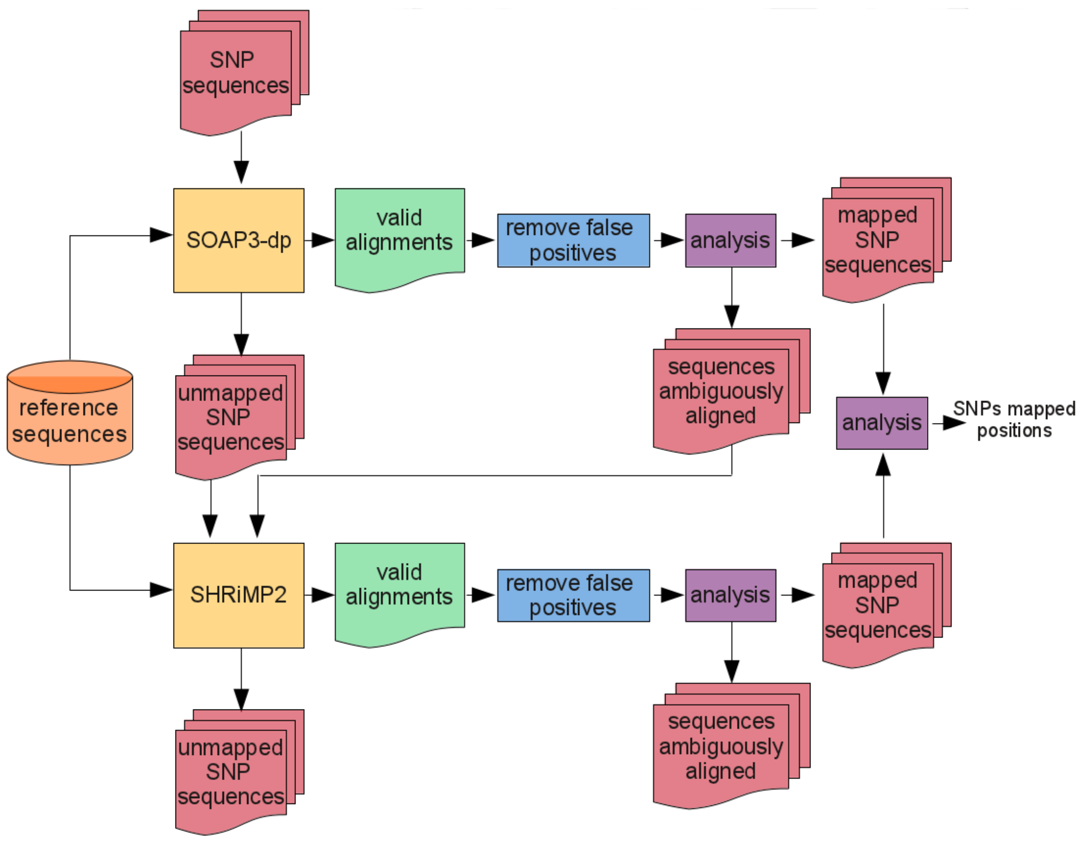


Figure 3.3: **G-SNPM mapping strategy**. G-SNPM exploits a three-stage pipeline to update the chromosomal position of a SNP. In the first stage, SOAP3-dp is used to unambiguously map a SNP against a reference sequence. Unmapped or ambiguously mapped SNPs are remapped at the second stage by exploiting SHRiMP2. At the third stage, mapped SNP sequences are analyzed to identify the SNP chromosomal position.

First stage of the pipeline

G-SNPM uses the GPU-based SOAP3-dp short-read mapping tool to align a sequence related to a SNP against its related chromosomal sequence. Typically, a short-read mapping tool is used to map a read against the overall genome. In fact, the genome region from which the read has been generated from the sequencer is unknown. To reduce the running time G-SNPM uniquely aligns each SNP against the reference chromosomal sequence shown in the mapping data of the chip. In fact, it is very unlikely that a SNP has been mapped to a wrong chromosome. Then, since SOAP3-dp exploits the BWT to index a reference sequence, it is necessary to index separately each chromosomal sequence involved in the mapping task.

In general, the alignment process can generate one of three possible results. In particular, also depending on the setting parameters, SOAP3-dp:

i. provides a unique alignment;

ii. provides multiple alignments;

iii. is unable to find an alignment with respect to the given constraints.

As previously explained, the adopted mapping strategy requires that G-SNPM analyzes the resulting alignments to filter out false positives. During the alignment, SOAP3-dp aligns each aNy symbol in a sequence as a mismatch against any possible nucleotide in the reference sequence. Therefore, G-SNPM *i*) analyzes each alignment to look for false positives, *ii*) removes them, and then *iii*) updates the edit distance of those alignments classified as true positives. To detect a unique SNP chromosomal position, a unique alignment must be considered valid. To this end, G-SNPM analyzes all valid alignments of each SNP sequence to detect the best hit and discard the others. Basically, the best hit might be detected by calculating the score alignment of each hit and selecting the best. However, G-SNPM analyzes a more complex score. In particular, it detects the best hit by analyzing the BWA-like MAPQ score provided with the last releases of SOAP3-dp that is intended to indicate confidence of read placement accuracy. This score assigns a Phred-like mapping quality score to each read based on match uniqueness, sequence identity, end-pairing, and inferred insert size.

Second stage of the pipeline

It is aimed at refining the mapping process. At this stage, G-SNPM tries to remap those SNPs (if any) that have not been mapped at the first stage of the pipeline;

in other words, those SNPs for which SOAP3-dp has not been able to provide valid alignments for their representative sequence and/or those SNPs for which G-SNPM has not been able to find unambiguous mapping chromosomal positions (i.e., SNPs for which SOAP3-dp found multiple valid alignments with the same mapping quality score). G-SNPM uses the Smith-Waterman based short-read mapping tool SHRiMP2 to perform this stage of the pipeline. As for the first stage, also in this stage G-SNPM adopts an identical policy to detect and discard false positives alignments that might be found by SHRiMP2, while exploiting the SHRiMP2 mapping quality score to detect the best alignment. At the end of this stage, G-SNPM reports those SNPs for which SHRiMP2 has been unable to find a unique valid alignment of their representative sequences or an unambiguous SNP chromosomal position.

Third stage of the pipeline

G-SNPM analyzes unique valid alignments of each successful mapped SNP to calculate the absolute position of each SNP. An output file is generated, containing for each SNP, its name, the related chromosome, the original SNP position, and the mapped SNP position. Moreover, information about the alignment as the strand, and the CIGAR string are also provided. Then, the pipeline is re-executed to map against the overall genome *i*) those SNPs that G-SNPM has been unable to map against a unique chromosomal sequence and *ii*) those SNPs unmapped by the chip vendor.

In G-SNPM reference DNA sequences are accepted in standard FASTA format, whereas SNPs must be represented by using two files: a FASTA file with the representative reads of the SNPs, and another flat file with information about the SNP, in particular the original absolute SNP position and its alleles. Currently, automatic generation of these files is provided for SNP probes of the Illumina Chip. G-SNPM analyzes Illumina files to automatically generate the previously described files for each chromosome.

3.3.2 Alignment constraints

G-SNPM defines different mapping constraints at the first and second stage of its pipeline, according to the different two mapping tools exploited.

First stage

Typically, due the time required to find an alignment, short-read mapping tools allow to set some parameters to limit the maximum alignments allowed for read sequence. For instance, by default Bowtie allows only one alignment for read sequence. In general, this limitation might affect the quality of the final results, especially when no sensitive alignment parameters are imposed. Short-read mapping tools that exploit modern GPUs allow to easily by-pass the limitations of this constraint. By default, SOAP3-dp generates up to 1000 alignments for read. This is deemed to be a good constraint and it was not modified in G-SNPM. However, users can easily modify it to decrease, increase, or avoid the upper limit to the alignments that may be found for each sequence.

As already pointed out, SOAP3-dp is the evolution of SOAP3 that exploits dynamic programming to support indels in alignments. Depending on whether dynamic programming is enabled or not, SOAP3-dp will generate gapped or ungapped alignments. When dynamic programming is enabled, SOAP3-dp performs the alignment in two steps. In the first step it looks for ungapped alignments that meet a given constraint on the allowed number of mismatches. Up to 4 mismatches are allowed for this step. In the second step, it exploits dynamic programming to look for gapped alignments. By default, in the first step SOAP3-dp allows up to 2 mismatches to speed-up the overall alignment process. However, G-SNPM modifies this constraint to allow alignments with up 4 mismatches. Users can decreases this value in G-SNPM.

Second stage

SHRiMP2 is an accurate short-read mapping tool that has been designed to parallelize the alignment process on multi-core CPUs. By default SHRiMP2 uses only a CPU-core. Then, to speed-up the analysis performed at this stage, G-SNPM assigns all available CPU-cores to SHRiMP2. In particular, it automatically detects the number *N* of available CPU-cores, and then runs SHRiMP2 on *N-1* cores; a CPU-core is reserved to the operating system. However, it is possible to set manually how many CPU-cores must be assigned to SHRiMP2.

Depending on the number of available CPU cores, it might be useful to limit the maximum number of alignments for sequence, with the aim to reduce the overall mapping time. However, it should be noted that most SNPs are successfully mapped at the first stage of the pipeline. So, the activation of the second stage is sporadic and involves only some SNP sequences. It was deemed useful not imposing any limitation on the number of alignments at this stage, to prevent any worsening of the overall accuracy of G-SNPM. At this stage, SHRiMP2 is enabled to allow ungapped alignments. Alignment score and penalties are those of default of SHRiMP2 (i.e., match score = 10; mismatch penalty = 15, gap open penalty = 33, gap extend penalty = 33). It is possible to change these values to meet user constraints.

3.3.3 Requirements

G-SNPM works on linux based systems with a custom installation of Python and equipped with a CUDA (Compute Unified Device Architecture) enabled GPU-card. It was tested on two families of NVIDIA GPU cards. In particular tests have been carried out on the NVIDIA FERMI architecture based GTX 480 card, and on the NVIDIA Kepler architecture based k10 and k20c cards. Currently, SOAP3-dp can be run on CUDA-3.2 and CUDA-4.2 releases, while no support for the CUDA 5.0 release has been provided yet. It is suggested to scientists interested to use G-SNPM to install the CUDA-4.2 release.

3.4 Results

To assess G-SNPM, it was used in the task to remap about *i*) 1.2 millions of SNPs of the Illumina Chip HumanOmni 1S (version 1) aligned by the chip vendor on the build 37.1 of the human genome, *ii*) 370 thousands of SNPs of the Illumina Chip CNV370 (version 3) aligned on the build 36.1 of the human genome, and *iii*) 318 thousands of SNPs of the Illumina Chip HH300 (version 2) also aligned by the chip vendor on the build 36.1 of the human genome. Experiments have been mainly executed *i*) to highlight discrepancies in respect in map positions provided by the chip vendor, and *ii*) to assess the capability of G-SNPM to deal with the mapping problem. In the following of this section, firstly both the hardware configuration and the short-read mapping tool releases exploited to carry out experiments are briefly summarized. Then, it is described the way data have been prepared, so that a scientist can easily reproduce experiments. Finally, results are presented and discussed.

3.4.1 Hardware and Software Configuration

Experiments described hereinafter have been carried out on a 12 cores Intel Xeon CPU E5-2667 2.90GHz with 128 GB of RAM. An NVIDIA Kepler architecture based Tesla k20c card with 0.71 GHz clock rate and equipped with 4.8 GB of global memory has been exploited to execute SOAP3-dp. Moreover, the following software releases were used: SOAP3-dp rel. 2.3.116 and SHRiMP2 rel. 2.2.3.

3.4.2 Data Preparation

The *.csv* file version of the Manifest of the analyzed chips was downloaded from the Illumina website. Then, the Illumina parser was used, which is distributed together with G-SNPM, to automatically generate the working files used by G-SNPM. Successively, the builds 36.1, 37.1 and 37.3 of the human genome were downloaded from the NCBI Reference Sequence Database [162]. Then, G-SNPM-Builder (also distributed along G-SNPM) was used to build the BWT indexes required in the first stage of the pipeline.

3.4.3 Analysis of Mapped SNPs

G-SNPM was used to perform two different experiments. As for the former, it was used to remap the SNPs of each chip against the same genome build previously used by the chip vendor. This experiment permits to put into evidence and to analyze possible discrepancies between the SNPs positions obtained with G-SNPM and those provided by the chip vendor. As for the second experiment, first the G-SNPM was used to remap the SNPs against the newest build 37.3 of the human genome and then, the reliability of the updated positions was analyzed. Table 3.2 reports some details about the SNPs of the analyzed chips. As for the HumanOmni 1S chip, it was observed that the vendor provided the positions of 1.180.662 SNPs. As the overall number of SNPs was 1.185.662 no information about the position of 5.314 SNPs was provided. The chip vendor provided the positions of all the 373.397 SNPs of the CNV370 chip, version 3, and of all the 318.237 SNPs of the HH300 chip, version 2.

Remapping SNPs against the same reference sequence used by the chip vendor

Table 3.3 summarizes results obtained remapping SNPs with G-SNPM against the same reference sequences used by the chip vendor. In the table are reported: *i*) the overall number of SNPs mapped using G-SNPM, *ii*) the number of those uniquely mapped, *iii*) the number of SNPs for which G-SNPM has been unable to find any alignment, and *iv*) the number of SNPs for which the proposed tool found positions

CHIP name	hg build	SNPs	unmapped SNPs
HumanOmni 1S	37.1	1.185.976	5.314
CNV370 ver 3	36.1	373.397	0
HH300 ver 2	36.1	318.237	0

The first column reports the name of the chips and the second the reference build of the human genome used by the chip vendor to map the SNPs. The third and fourth column report the overall number of SNPs of the chip and the number of them unmapped by the chip vendor, respectively.

Table 3.2: **Analyzed chips**

.

that differ from those provided by the chip vendor. As for the chip HumanOmni 1S, G-SNPM has been able to remap 4.460 of the 5.314 SNPs for which the chip vendor did not provide any mapping position. Most of these SNPs have been mapped at the first stage of G-SNPM. In particular, they have been mapped by SOAP3-dp looking for ungapped alignments and without exploit any heuristic. Only 35 of these SNPs have been mapped looking for gapped SNPs. In the last column of Table 3.3 is reported that 4.626 SNPs have been differently mapped with G-SNPM. It should be observed that this value includes also the 4.460 SNPs mapped only by G-SNPM.

		SNPs			
CHIP name	**hg build**	**mapped**	**uniquely mapped**	**unmapped**	**differently mapped**
HumanOmni 1S	37.1	1.185.122	1.185.118	854	4.626
CNV370 ver 3	36.1	373.397	373.382	0	14.391
HH300 ver 2	36.1	318.237	318.237	0	1.822

A summarization of the discrepancies observed remapping the SNPs with G-SNPM against the same reference builds previously used by the chip vendor to detect the SNPs positions. The first and the second column report the name of the chip and its reference build, respectively. The third column reports the overall number of SNPs mapped using G-SNPM, whereas the fourth column reports the number of them that are uniquely mapped. The fifth column reports the number of SNPs for which G-SNPM did not provide any valid alignment. Finally, the sixth column reports the number of mapped SNPs for which G-SNPM provided different positions with respect to those detected by the chip vendor.

Table 3.3: **Results obtained using G-SNPM to remap the SNPs against the same reference build used by the chip vendor.**

Analyzing the SNPs mapped by the chip vendor, only 166 of them have been mapped differently with G-SNPM, one on a different chromosome. As for the other chips, G-SNPM mapped uniquely against their related reference build almost all SNPs. Experimental results shown that G-SNPM mapped differently 14.391 SNPs (7 on a different chromosome) of the chip CNV370, version 3, and 1.822 SNPs (none on a different chromosome) of the chip HH300, version 2. Also for these chips G-SNPM mapped almost all SNPs without considering gapped alignments. The differences between the SNPs mapped by G-SNPM with respect those mapped by the chip vendor could be attributed to differences in the alignment algorithms and settings. As reported in the background section, different works have proved that often unreliable positions are provided along the chip, typically due to the fact that not very accurate alignment were obtained. The algorithm and alignment settings used by the vendor were unknown. Then, it was difficult to compare the accuracy of the proposed tool with the one of the vendor. In any case G-SNPM is very accurate. Being based on SOAP3-dp, it looks for ungapped alignments with up to four mismatches without exploiting any heuristics. It is worth pointing out that only a very low percentage of SNPs positions have been calculated starting from gapped alignments and that almost all sequences representative of the SNPs have been uniquely mapped. As for the SNPs of the HumanOmni 1S mapped by G-SNPM and for which the chip vendor did not provide any position, it can be supposed that either no valid alignment have been found for them or, conversely, that multiple valid alignments have been found making impossible to unambiguously map these SNPs. As for the 854 SNPs unmapped also by the proposed tool, it is assumed that G-SNPM tried to map them using some heuristics that did not permitted to find valid alignments.

CHIP name	**hg build**	**mapped SNPs**	**uniquely mapped SNPs**	**unmapped SNPs**
HumanOmni 1S	37.3	1.185.108	1.185.103	868
CNV370 ver 3	37.3	373.374	373.371	23
HH300 ver 2	37.3	318.217	318.216	20

The first and the second columns report the name of the chip and its reference build, respectively. The third column reports the overall number of SNPs mapped using G-SNPM, whereas the fourth column reports the number of them uniquely mapped. The fifth column reports the number of SNPs for which G-SNPM did not provide a valid alignment.

Table 3.4: **Results obtained using G-SNPM to remap the SNPs against the build 37.3 of the human genome.**

Remapping SNPs against the build 37.3 of the human genome

Table 3.4 summarizes results obtained remapping SNPs with G-SNPM against the build 37.3 of the human genome. It should be observed that results are slightly different from those obtained remapping the SNPs against the same build used by the chip vendor. Results show that G-SNPM has been unable to remap some SNPs previously mapped against the oldest builds. As for the chip HumanOmni 1S, almost all SNPs unmapped by the chip vendor have also been mapped against the newest build of the genome. In particular, G-SNPM has been unable to find a valid alignment for 868 SNPs (i.e., 14 SNPs more than in the previous experiment). For the other SNPs unmapped by the vendor, G-SNPM found that they map to the same positions in both builds. As for the other chips, G-SNPM has been unable to find a valid alignment for 23 SNPs of the chip CNV370, version 3, and for 20 SNPs of the chip HH300, version 2. As for the unmapped SNPs, it is possible that, *i*) due to the refinement of the reference sequence, some SNPs are no longer present in the latest build or that *ii*) the refinement of the reference sequence required complex gapped alignments that G-SNPM is unable to find, due to the procedures adopted in the two stages of its pipeline. As in the previous experiment, almost all SNPs have been mapped at the first stage of G-SNPM, while looking for ungapped alignments.

To analyze the reliability of the proposed tool, a comparison was made of the SNPs positions on the build 37.3 obtained with G-SNPM with *i*) those obtained using a genome remapping tool, and with *ii*) those retrieved by dbSNP. As for the first comparison, the NCBI Genome Remapping Service was used because at the time of writing of this thesis it is the only assembly-assembly converter tool able to project features from the build 36.1 to the build 37.3, whereas neither the NCBI Genome Remapping Service nor the UCSC LiftOver and Ensembl AssemblyConverter services are currently able to project features from the build 37.1 to the build 37.3. Therefore, this experiment has not been performed for the chip HumanOmni 1S. The NCBI Genome Remapping Service projects the coordinates of a chromosomal region between two different builds of a genome. In this case, the aim is to project against the build 37.3 the coordinates of those regions that contain the SNPs in the build 36.1. Assuming that the SNPs positions provided by the chip vendors are correct, it is possible to identify these regions retrieving the sequences representative of the SNPs, their relative positions within these sequences, and their absolute positions within the chromosome sequence. This information is present in *.csv* files of the Manifest of the chips analyzed for this study. Table 3.5 summarizes results obtained with the NCBI service. It should be observed that it has been unable to convert the coordinates of several regions if compared with the number of SNPs unmapped by G-SNPM. In particular, it has been unable to project the coordinates of 212 SNPs of the CNV370 chip, version 3, and the coordinates of 28 SNPs of the HH300 chip, version 2. Typically, regions are unmapped either as they are deleted in the new reference or as intersects multiple chains. Moreover, it was analyzed if the SNPs mapped with G-SNPM fall in the regions that have been projected with the NCBI service. Results reported in Table 3.6, show that G-SNPM mapped 7.296 SNPs of the chip CNV370, version 3, in different regions of those obtained with the NCBI service, as well as 454 SNPs of the chip HH300, version 2. Differences might be related to the fact that G-SNPM looks for the nucleotide present in the SNP position and discard those alignments that do not match with one of the possible alleles for the SNP. As the NCBI service does not perform this check, it can report also wrong regions. As for the second comparison, the SNPs of the HumanOmni 1S chip were differently analyzed from those of the chips CNV370 and HH300. In particular, the SNPs of the HumanOmni 1S chip unmapped by the vendor were retrieved from dbSNP. Only 47 of them have a rsID whereas the others have been derived from the 1000 Genomes Project (kgp identifiers). The SNPs with kgp identifiers were converted to rsIDs in dbSNP132 using MegaBLAST [204] to align against the database the sequences representative of the SNPs. It was observed that only 859 of 5.314 SNPs were present in dbSNP132 and all of them with multiple positions. Only a little percentage of them validated. For about half of these KGP SNPs, and for all SNPs in the chips with rsID In dbSNP were found the same positions obtained with the proposed tool. As for the other chips, all SNPs mapped by G-SNPM were searched on dbSNP. About 281 thousands SNPs of the CNV370 chip and about 238 thousands SNPs of the HH300 chip were present in dbSNP. It was observed that G-SNPM did not provide identical SNPs positions for 1.447 SNPs of the CNV370 chip and for 1.281 SNPs of the HH300 chip. As for the SNPs for which G-SNPM provided different positions, It was observed that dbSNP reports longer flanking sequences that those reported by the vendor. This can be related to the different mappings of G-SNPM as well as the regions unprojected by the NCBI Genome Remapping Service.

Performance Analysis

Table 3.7 summarizes the performance of G-SNPM in terms of overall mapped SNPs and running time. Results are reported for all experiments performed and are distinct according to the mapping option. As previously explained, G-SNPM tries to remap against the overall genome sequence those SNPs that have been unmapped against the same chromosomal sequence detected by the chip vendor. In these cases,

CHIP name	**projected regions**	**unprojected regions**
CNV370 v. 3.0	373.185	212
HH300 v. 2.0	318.209	28

A summarization of the results observed converting from the build 36.1 to the build 37.3 of the human genome the coordinates of the regions containing the SNPs detected by the chip vendor. The first column reports the name of the chip, whereas the second and the third report the number of regions successfully projected against the build 37.3 and the number of regions for which the NCBI service has been unable to provide any conversion, respectively.

Table 3.5: **SNPs chromosomal regions projected with the NCBI Genome Remapping Service against the build 37.3 of the human genome.**

CHIP name	**regions differently remapped**
CNV370 ver 3	7.296
HH300 ver 2	454

The table shows for each analyzed chip the number of SNPs remapped with G-SNPM against the build 37.3 of the human genome whose positions did not fall inside the regions obtained with the NCBI Genome Remapping Service.

Table 3.6: **Comparison between G-SNPM and the NCBI Genome Remapping Service.**

analysis at the second stage of G-SNPM can require a very long running time. G-SNPM by default tries to align these SNPs only at the first stage. To force the second stage alignment, users must specify the *"D"* option. In the table, results are summarized for both cases. It should be observed that the running time greatly increases when the *"D"* option is used. Only a small percentage of SNPs is further mapped against the overall genome sequence at the second stage of G-SNPM. The time for mapping the SNPs of chip HH300, version 2, do not change after activating this option *"D"*, as all SNPs are in fact mapped at the first stage. Moreover, the table shows that G-SNPM aligns almost 1.2 million of SNPs of the HumanOmni 1S chip faster than the almost 370 thousands SNPs of the CNV370 chip, version 3, and the almost 318 thousands SNPs of the HH300 chip, version 2. Justification must be sought in the fact that in the HumanOmni 1S chip almost all SNPs are mapped at the first stage of G-SNPM. As for the others, G-SNPM required more time to try to map SNPs at the second stage. Table 3.8, summarizes the number of sequences that G-SNPM tried to align at the second stage of the pipeline and its related processing time. Results shown in Table 3.8 highlight the presence of a considerable imbalance with respect to the number of sequences processed at the first stage (for instance considering the HumanOmni 1S chip, G-SNPM processed about 1.2 millions of SNPs against the build 37.1 in 20 minutes, of which 13 minutes to process 17 sequences at the second stage).

		option D disabled		option D enabled	
CHIP name	**reference build**	**mapped SNPs**	**global time**	**mapped SNPs**	**global time**
HumanOmni 1S	37.1	1.184.688	20m	1.185.118	1h 34m
HumanOmni 1S	37.3	1.185.031	19m	1.185.103	1h 30m
CNV370 v. 3.0	36.1	373.382	56m	373.382	2h 5m
CNV370 v. 3.0	37.3	373.367	52m	373.371	2h 2m
HH300 v. 2.0	36.1	318.237	29m	318.237	29m
HH300 v. 2.0	37.3	318.216	37m	318.216	37m

The table is divided in two parts. The first summarizes the performance of G-SNPM when only its first stage has been used to remap against the overall genome sequence those SNPs previously unmapped against the same chromosomal sequence detected by the chip vendor (option "D" disabled). The second part of the table summarizes the performance of G-SNPM when both stages have been used to remap against the overall genome sequence those SNPs previously unmapped against the same chromosomal sequence detected by the chip vendor (option "D" enabled).

Table 3.7: **Overall analysis of mapped SNPs and running time**

CHIP name	**reference build**	**sequences analyzed**	**time**
HumanOmni 1S	37.1	17	13m
HumanOmni 1S	37.3	17	12m
CNV370 v. 3.0	36.1	56	41m
CNV370 v. 3.0	37.3	81	49m
HH300 v. 2.0	36.1	10	22m
HH300 v. 2.0	37.3	36	27m

A summarization of the performance in terms of running time at the second stage of the G-SNPM. The table shows the number of sequences that G-SNPM tried to align at the second stage and the time required to align them. It is evident a considerable imbalance of the processing time between the first and the second level. The table summarizes the performance with option "D" disabled.

Table 3.8: **Analysis of the performance at the second stage of G-SNPM**

3.5 Discussion and Conclusions

G-SNPM is a useful and powerful tool that can simplify the work of researchers that plan to remap the SNPs chromosomal positions before to perform any GWAS. Typically, researchers use sequence alignment tools as BLAST or BLAT to update the mapping position of a SNP to a genome or a transcriptome. However, no generalized and/or computationally efficient solutions have been proposed to address this problem. G-SNPM is the only general-purpose tool devised to deal with the mapping of SNPs. Being based on modern GPUs, it exploits the computational power of these hardware accelerators to guarantee a very fast mapping without compromising the accuracy. G-SNPM can be easily integrated in specialized pipelines and workflows devised to cope with specialized GWAS, as well as annotation tasks that requires to remap the SNP probes.

Chapter 4

GPU-BSM: A GPU-Based Tool to Map Bisulfite-Treated Reads

Cytosine DNA methylation is an epigenetic mark implicated in several biological processes. Bisulfite treatment of DNA is acknowledged as the gold standard technique to study methylation. This technique introduces changes in the genomic DNA by converting cytosines to uracils while 5-methylcytosines remain nonreactive. During PCR amplification 5-methylcytosines are amplified as cytosine, whereas uracils and thymines as thymine.

Two main protocols have been developed to construct bisulfite-treated libraries for high-throughput sequencing. These protocols, methylC-seq and BS-seq, mainly differ in the PCR amplification procedure. In methylC-seq libraries are generated in a directional manner: a single amplification step is performed, so that reads are related to the forward (+FW) or to the reverse (-FW) direction of the bisulfite-treated sequence. Libraries generated using the methylC-seq protocol are termed directional. In BS-seq, two amplification steps are performed, so that bisulfite reads may be related to four different directions of the bisulfite-treated sequence: forward Watson strand (+FW) and its reverse complement (+RC), forward Crick strand (-FW) and its reverse complement (-RC). Libraries generated using the BS-seq protocol are termed non-directional.

The main limitation of the widely used whole-genome bisulfite sequencing (WGBS) is related to its cost, which is very high. Reduced representation bisulfite sequencing (RRBS) is an alternative and cost-effective technique used to study methylation. In RRBS, DNA genome is first digested using specific restriction enzyme to enrich for CpGs. Then, the DNA fragments are size-selected and subsequently, as for WGBS, treated with bisulfite to be sequenced. Hence, in RRBS, only specific CpG-rich regions are considered.

To detect the methylation levels, reads treated with the bisulfite must be aligned against a reference genome. Mapping these reads to a reference genome represents a computational challenge mainly due to i) the increased search space and ii) the loss of information introduced by the bisulfite treatment.

To deal with this computational challenge the GPU-based tool GPU-BSM has been devised. GPU-BSM is a tool able to map bisulfite-treated reads from both WGBS and RRBS, and to estimate methylation levels, with the goal of detecting methylation. Due to the massive parallelization obtained by exploiting graphics cards, GPU-BSM aligns bisulfite-treated reads faster than other cutting-edge solutions, while outperforming most of them in terms of unique mapped reads.

4.1 Introduction

Regulation of gene expression is a very complex process controlled by multiple factors, including epigenetic ones. Epigenetics studies changes in gene expression that do not involve changes in the underlying DNA sequence [84]. Specifically, it refers to functionally relevant modifications that permit the genes of an organism to express themselves differently. Cytosine DNA methylation is a stable epigenetic mark that plays a very important role in several biological processes, including genomic imprinting, and is often responsible of phenotypic expressions (e.g., cancer) [43]. It involves the addition of a methyl group to the cytosine DNA nucleotides (see Figure 4.1). Mechanisms of epigenetic regulation include methylation at CpG islands in the promoter region of the gene. In many disease-causing processes gene promoter CpG islands acquire abnormal hypermethylation [85], which results in transcriptional silencing that can be inherited by daughter cells upon cell division. Three main approaches (i.e., endonuclease digestion, affinity enrichment and bisulfite conversion) [102] have been developed to analyze DNA methylation and various molecular biology techniques, as probe hybridization and sequencing, can be used to identify methylated cytosines in genomic DNAs treated with one of these approaches.

Bisulfite treatment of DNA [52] is considered the gold standard technique to study methylation. This technique introduces specific changes in the DNA sequence, depending on the methylation status of individual cytosine residues. Genomic DNA is modified by converting cytosines to uracils, while 5-methylcytosines remain nonreactive. In particular, during PCR amplification, only 5-methylcytosines are amplified

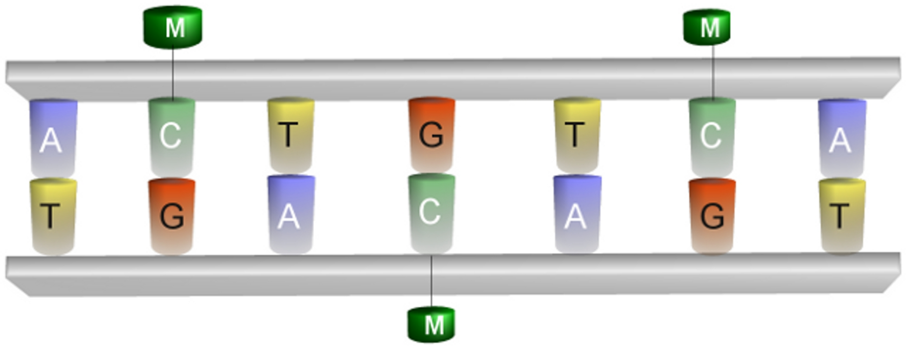


Figure 4.1: **Cytosine DNA Methylation.** Cytosine DNA methylation is a epigenetic mechanisms that affects gene expression. It involves the addition of a methyl group to the cytosine DNA nucleotides.

as cytosine, whereas uracils and thymines are amplified as thymine. Two main protocols have been developed to construct bisulfite-treated libraries for high-throughput sequencing. These protocols, methylC-seq [115] and BS-seq [39], mainly differ in the PCR amplification procedure. In methylC-seq libraries are generated in a directional manner: a single amplification step is performed, so that reads are related to the forward (+FW) or to the reverse (-FW) direction of the bisulfite-treated sequence. Libraries generated using the methylC-seq protocol are termed directional. In BS-seq, two amplification steps are performed, so that bisulfite reads may be related to four different directions of the bisulfite-treated sequence: forward Watson strand (+FW) and its reverse complement (+RC), forward Crick strand (-FW) and its reverse complement (-RC) (see Figure 4.2). Libraries generated using the BS-seq protocol are termed non-directional.

The main limitation of whole-genome bisulfite sequencing (WGBS) is related to its cost, which is very high. Reduced representation bisulfite sequencing (RRBS) [135] is an alternative and cost-effective technique used to study methylation. In RRBS, DNA genome is first digested using specific restriction enzyme to enrich for CpGs. Then, the DNA fragments are size-selected and subsequently, as for WGBS, treated with bisulfite to be sequenced. Hence, in RRBS, only specific CpG-rich regions are considered.

To calculate the methylation levels, bisulfite-treated reads are aligned against a reference genome. Mapping these reads to a reference genome represents a computational challenge mainly due to *i*) the increased search space and *ii*) the loss of information introduced by the bisulfite treatment. As for the former issue, considering that bisulfite affects only cytosines, non complementary Watson and Crick strands are

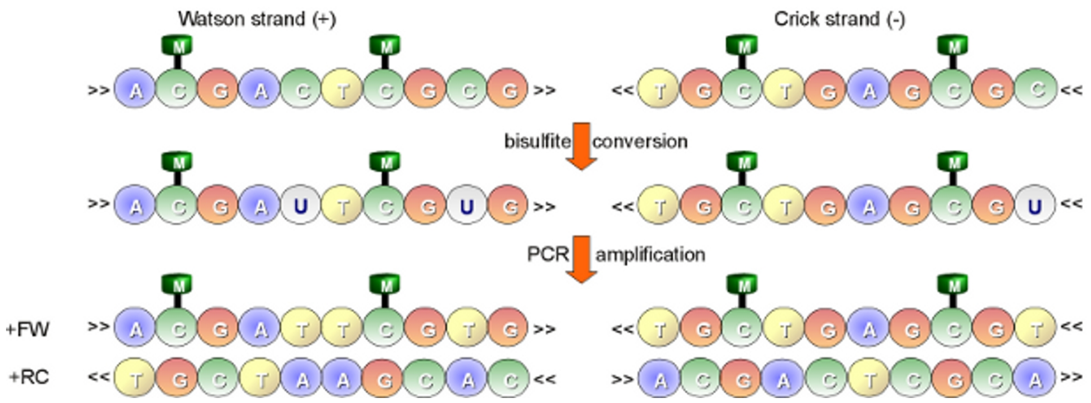


Figure 4.2: **Bisulfite treatment.** Two main type of libraries can be generated, directional and non-directional. As for directional libraries, a single amplification step is performed so that reads are related either to the forward (+FW) or to the reverse (-FW) direction of the bisulfite-treated sequence. Conversely, as for non-directional libraries, two amplification steps are performed, so that bisulfite reads may be related to four different directions of the bisulfite-treated sequence: forward Watson strand (+FW) and its reverse complement (+RC), forward Crick strand (-FW) and its reverse complement (-RC).

generated. As previously highlighted, this implies that PCR amplification of both strands will produce up to four different strands and the bisulfite treated read can be derived from any of these strands. Moreover, the alignment process is further complicated by the asymmetric mapping between cytosines and thymines. In fact, a thymine in a read can be mapped to a cytosine in the reference genome, but the inverse is not allowed (see Figure 4.3). As for the latter issue, it should be pointed out that only a very small portion of cytosines is methylated in mammalian [164], making more difficult the alignment process along the reference genome.

Some tools have been proposed in the literature to address this mapping challenge. These tools can be divided in two classes, according to the strategy adopted to deal with the asymmetric mapping between cytosines and thymines. Tools belonging to the first class are specifically designed to perform alignments by allowing cytosines and thymines in the reads to match with cytosines in the reference sequence. By contrast, tools in the second class adopt an unbiased strategy that reduces the complexity of involved sequences converting cytosines to thymines. In so doing, sequences are represented with a simplified 3-letter nucleotide alphabet and alignments can be carried out with common and available short-read mapping tools. Alignments obtained exploiting this strategy must be post-processed to avoid those ambiguous and false positives. Tools in the first class provides the highest

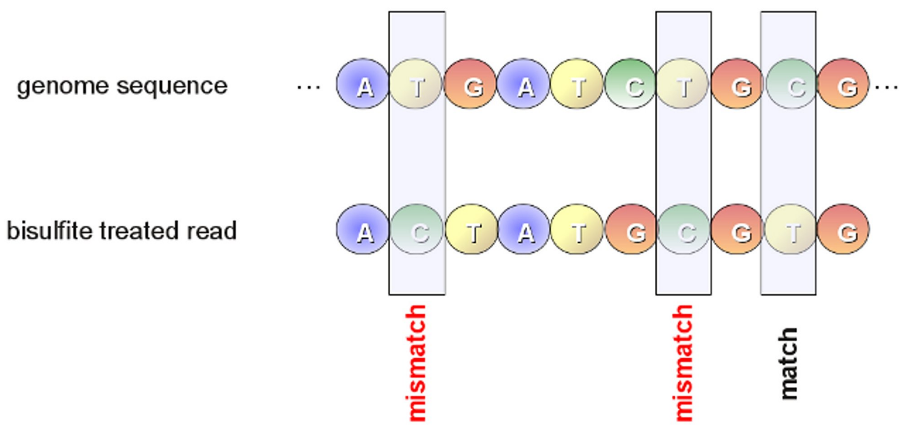


Figure 4.3: **Asymmetric mapping.** Due to the bisulfite treatment, unmethylated cytosines are converted to thymines during the PCR amplification. This conversion must be take into account during alignment by allowing an asymmetric mapping. A thymine in a read mapped to a cytosine in the reference genome sequence is considered as a match, whereas a thymine in the genome sequence mapped to a cytosine in a read is considered as a mismatch.

mapping efficiency. However, it should be observed that with the mapping strategy adopted by these tools, methylated read sequences will be aligned with greater efficiency than unmethylated ones. This means that tools in this class can overestimate methylation levels. By contrast, tools in the second class provide a slightly reduced mapping efficiency whereas alignment of reads is unaffected by their methylation state [101].

With no claim of being exhaustive, BSMAP/RRBSMAP [194][193] (it is pointed out that the latest release of RRBSMAP has been merged into BSMAP) and segemehl [157] are cited as tools of the first class, and BS-Seeker/BS-Seeker2 [36][65], Bismark [100], and BRAT-BW [67] are of the second class. BSMAP, applies to the reads a reverse bisulfite conversion, converting thymines to cytosines only at cytosine positions in the reference genome; then, it maps the masked reads to the genome. RRBSMAP, was the first tool specifically tailored for RRBS libraries. Based on suffix arrays, segemehl was the first tool able to take into account indels (insertions/deletions) in alignments of bisulfite-treated reads. Its high speed and accuracy are obtained by means of multi-threading, and with a very high memory consumption compared to those of others state-of-the-art tools. BS-Seeker performs a 3-letter alphabet reduction by converting cytosines to thymines on the FW reads and on both strands of the reference genome. Then, it uses the Bowtie [104] short-read alignment tool to map the converted FW reads against the converted Watson and Crick strands. In the event that reads are generated from the BS-seq protocol, a guanine to adenine conversion is performed on the RC of both reads and reference genome. Bowtie is then used to map the converted RC reads to the converted RC of the Watson and Crick strands. BS-Seeker runs in parallel the different instances of Bowtie. A final post-processing phase is performed to detect false positive alignments and methylation. BS-Seeker2 is an updated version of BS-Seeker that can also map reads from RRBS. Furthermore, BS-Seeker2 supports gapped global and local alignments with the newest multi-threading Bowtie2 [103] release. Bismark is an alternative tool able to map bisulfite-treated reads generated with both WGBS and RRBS. Similarly to BS-Seeker2, Bismark uses Bowtie2 and Bowtie to map preprocessed reads with and without indels supports respectively. Unlike from BS-Seeker2, Bismark does not support local alignments when used with Bowtie2. BRAT-BW uses the same strategy adopted by BS-Seeker and Bismark, while efficiently implementing the FM-index [47] in terms of memory occupancy. In general, due to the computational effort that may be required to cope with this mapping task, these tools present one or more implicitly or explicitly imposed constraints on the alignment process (e.g., number of mismatches, number of hits for reads, indels support). Table 4.1 reports a summarization of some features of the cited tools.

tool	**3-letter**	**mismatches**	**indels support**	**hits/reads**	**WGBS-RRBS**
Bismark	Yes	Unlimited	Yes*	Unlimited	Yes
BSMAP	No	15	Yes	1000	Yes
BS-Seeker	Yes	3	No	2	only WGBS
BS-Seeker2	Yes	Unlimited	Yes*	2	Yes
BRAT-BW	Yes	Unlimited	No	Unlimited	only WGBS
segemehl	No	Unlimited	Yes	Unlimited	only WGBS

Some bisulfite-treated reads mapping tools listed according to some relevant features. The second columns indicates whether the corresponding tool adopts a 3-letter conversion strategy. The third column reports the maximum number of mismatches allowed for the read. The fourth column reports whether the corresponding tool supports gapped alignments. The fifth column reports the maximum number of hits allowed for a read. The sixth column reports whether the corresponding tool supports WGBS and RRBS protocols.

* Using Bowtie2.

Table 4.1: **Bisulfite-treated reads mapping tools.**

In this work, GPU-BSM (standing for GPU-BiSulfite reads Mapping) is presented, an accurate and very fast tool devised to map bisulfite-treated reads and to estimate methylation levels. Written in Python, GPU-BSM exploits the 3-letter nucleotide alphabet reduction strategy and it is mainly based on SOAP3-dp [122], a short-read mapping tool able to take advantage of the computational power of modern Graphics

Processing Units (GPU). GPU-BSM has been designed to support ungapped and gapped (global and local) alignment with libraries generated with both WGBS and RRBS. Currently, GPU-BSM can be run parallelized on up to 4 different GPU cards. The massive parallelization obtained by means of GPUs enables GPU-BSM to map bisulfite-treated reads without imposing stringent limitations on the alignment process.

4.2 Methods

Based on the 3-letter nucleotide alphabet reduction strategy, GPU-BSM implements an approach similar to the one adopted by similar tools as BS-Seeker, Bismark, and BRAT-BW. In particular, similarly to other tools, GPU-BSM uses a third-part short-read mapper (i.e., SOAP3-dp) to align 3-letter converted reads. In the following of this section, it is first given a short introduction to GPUs and to existing state-of-the-art short-read mapping tools. Then, a new strategy is presented, devised to deal with the bisulfite-treated reads mapping problem. Subsequently, the adopted alignment constraints are discussed. Finally, the hardware and software equipment required to use GPU-BSM are briefly resumed.

4.2.1 The implemented strategy

Reads alignment may be very expensive in terms of both computing time and exploited hardware resources. Modern short-read mapping tools try to speed the alignment *i*) by parallelizing the overall process, and *ii*) by using ad-hoc heuristics. Parallelization could considerably accelerate the alignment, but it is often limited by the available hardware resources (i.e., CPU cores and memory). On the other hand, the adoption of heuristics may degrade sensitivity and affect the quality of the final results. As already pointed out, the computational challenge is heightened in the process to map bisulfite-treated reads, in which to map a read two or four different alignments must be performed according to the used protocol. Massive parallelization that may be obtained exploiting GPUs has been successfully used to address the short-read mapping problem and it is deemed that it may be exploited to address also the mapping of bisulfite-treated reads. In fact, GPU-BSM uses the GPU-based SOAP3-dp mapping tool to align bisulfite-treated reads.

Initially, GPU-BSM creates two sequences from the forward genomic strand. The first sequence is obtained by converting cytosines to thymines, whereas the second sequence is obtained by converting guanines to adenines. These sequences are created differently, depending on the sequencing technique used to generate the analyzed library. As for WGBS libraries, sequences are created from the original forward genomic strand, whereas for RRBS libraries they are created from a modified version to take into account the sequencing parameters. In particular, GPU-BSM modifies the genomic strand masking those DNA fragments that do not meet the sequencing parameters. In so doing, GPU-BSM notably improves the computing time required to align RRBS libraries.

Directional and non-directional libraries are treated differently. To map reads from a directional library, GPU-BSM performs two different alignments using SOAP3-dp (see Section 3.2). The first alignment is obtained by converting cytosines to thymines in the reads and then aligning them to the first sequence. The second is obtained by converting guanines to adenines in the reverse complement of the reads and then aligning them to the second sequence (see Figure 4.4). To map reads from a non-directional library, GPU-BSM performs four different alignments. In addition to the alignments performed for a directional library, GPU-BSM uses SOAP3-dp to map the reverse complement of the reads with cytosines converted to thymines to the first sequence, and the reverse complement of the reads with guanines converted to adenines to the second sequence. Then, GPU-BSM analyzes the mapped reads, detecting and removing ambiguous reads and those that are in fact false positives (see Figure 4.5). Those reads for which *i*) a best match exists for at least two of two/four alignments performed according to the exploited library or *ii*) at least two best hits exist for a single alignment are considered ambiguous. However, users interested in these mappings can disable this filtering option. To detect false positives, GPU-BSM calculates the number of mismatches of the mapped reads using the 4-letter nucleotide alphabet. Note that, due to the bisulfite treatment, a thymine in a read can be aligned to a cytosine in the reference sequence. Similarly, a guanine in the reverse complement of a read can be aligned to an adenine in the reference sequence (see Figure 4.6).

To take advantage of multiple GPUs, GPU-BSM automatically runs in parallel the two (four) different alignments for directional (non-directional) libraries. In the current release, GPU-BSM performs alignments on up to four GPUs. In particular, it uses up to two GPU cards to perform two different alignments required for reads of directional libraries, whereas it uses up to four cards to perform four different alignments required for reads of non-directional libraries. For machine equipped with a single GPU card, GPU-BSM sequentially performs the different alignments.

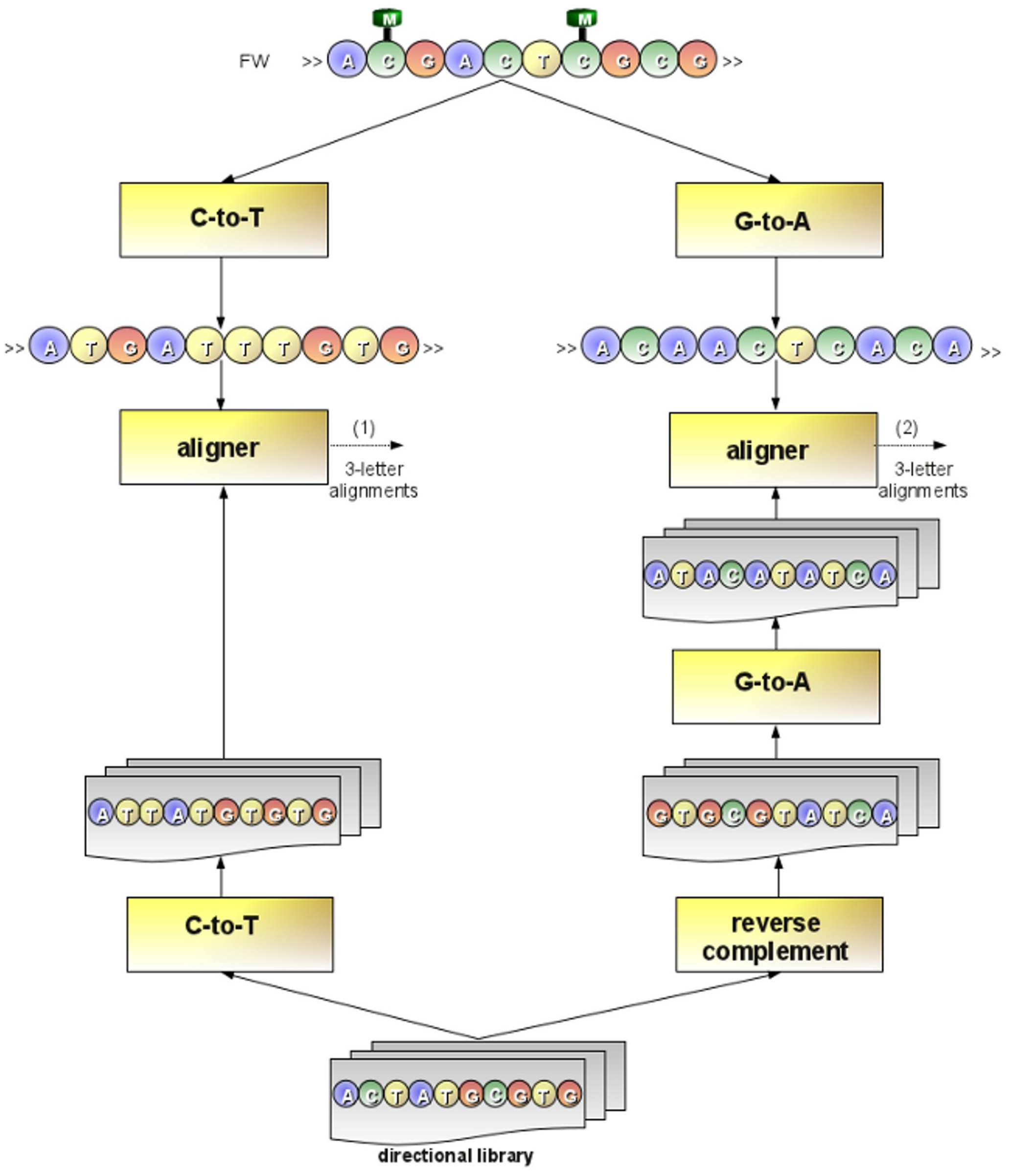


Figure 4.4: **Mapping directional reads.** To map directional reads, GPU-BSM performs two different alignments. As for the former alignment, GPU-BSM maps the reads of the library against the forward strand of the reference genome, after that cytosines have been converted to thymines in all sequences. As for the latter alignment, GPU-BSM maps the reverse complement of the reads against the forward strand of the reference genome, after that guanines have been converted to adenines in all sequences. Finally, all 3-letter alignments obtained for a read (i.e., outputs (1) and (2) in the figure) will be post-processed with the aim to detect and remove those ambiguous and false positives.

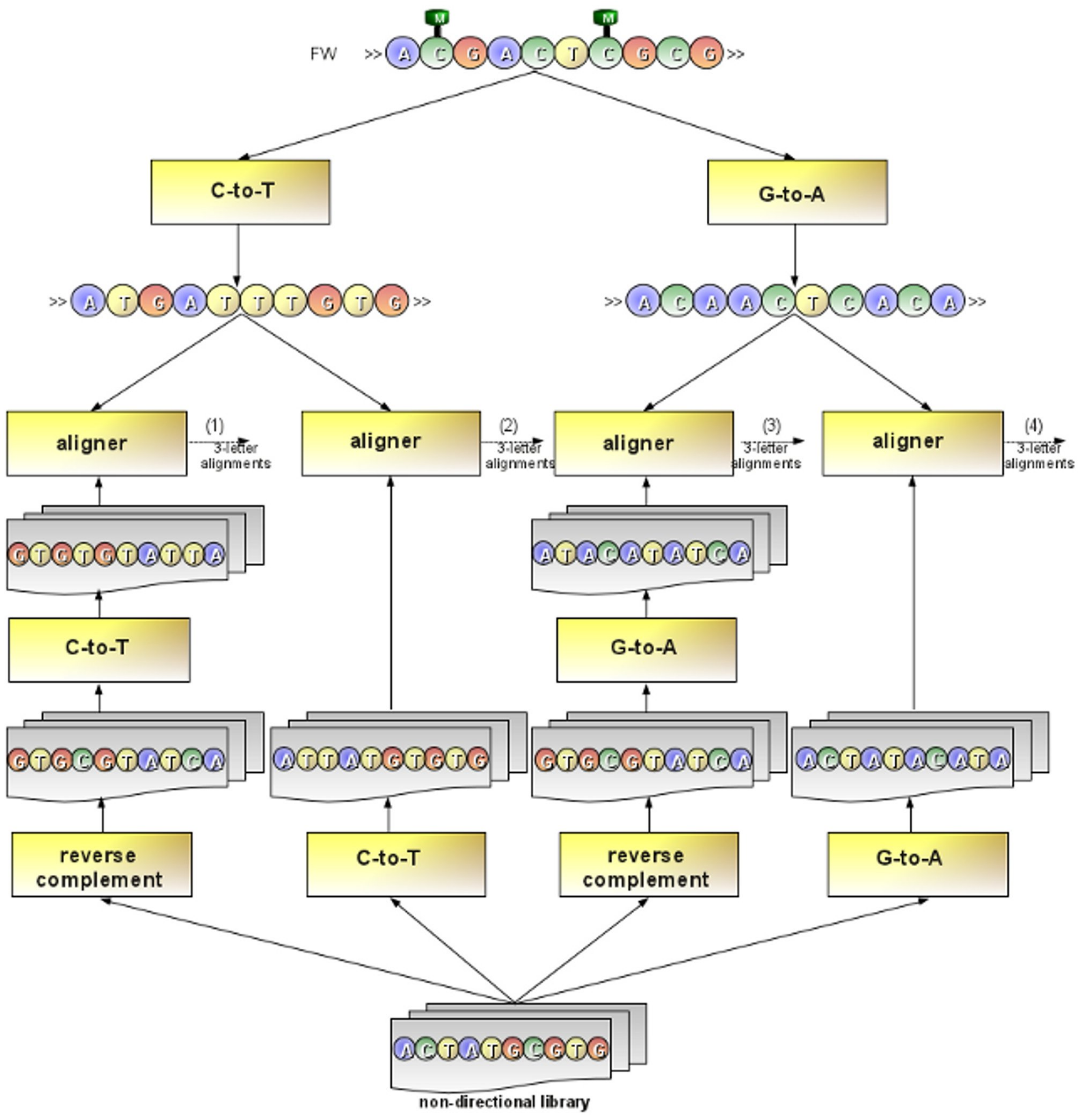


Figure 4.5: **Mapping non-directional reads.** To map non-directional reads, GPU-BSM performs four different alignments. The figure shows that two additional alignments are performed with respect to ones reported in Fig. 4.4 for directional reads. As for the first additional alignment, GPU-BSM maps the reads of the library against the forward strand of the reference genome after that guanines have been converted to adenines in all sequences. As for the second alignment, GPU-BSM maps the reverse complement of the reads of the library against the forward strand of the reference genome after that cytosines have been converted to adenines in all sequences. Finally, all 3-letter alignments obtained for a read (i.e., outputs (1), (2), (3) and (4) in the figure) will be post-processed with the aim to detect and remove those ambiguous and false positives.

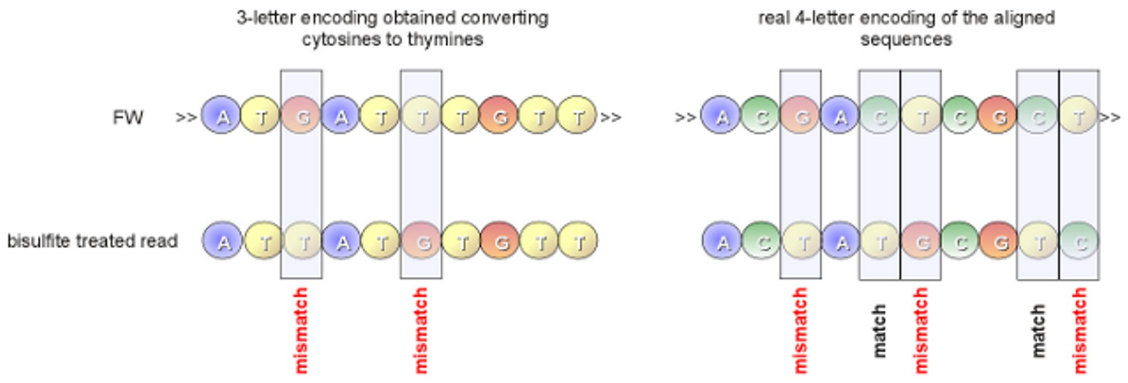

3-letter encoding obtained converting cytosines to thymines
real 4-letter encoding of the aligned sequences
FW
bisulfite treated read
mismatch
mismatch
mismatch
match
mismatch
match
mismatch

Figure 4.6: **False positive alignments.** GPU-BSM aligns reads exploiting a reduced 3-letter nucleotide alphabet. Alignments obtained using this encoding must be processed to look for false positives; i.e., those alignments that in the actual 4-letter nucleotide alphabet do not meet the alignment constraints imposed by the user. A typical case is represented in this figure. A two mismatches alignment obtained with the 3-letter encoding is reported on the left side. The same alignment, reported on the right of the figure with 4-letter nucleotide alphabet, shows three mismatches.

4.2.2 Tool settings

Depending on whether dynamic programming is enabled or not, SOAP3-dp will generate gapped or ungapped alignments. When dynamic programming is enabled, SOAP3-dp performs the alignment in two steps. In the first step, it uses SOAP3 to look for ungapped alignments that meet a given constraint on the allowed number of mismatches. Up to 4 mismatches are allowed for this step and no heuristic is used. In the second step, it exploits dynamic programming to look for gapped alignments. A score threshold defines when to use dynamic programming. It is also possible to skip the first step with the aim to align all reads exploiting the dynamic programming reducing the computing time. By default, in the first step SOAP3-dp allows up to two mismatches to speed-up the overall alignment process. However, GPU-BSM uses SOAP3-dp to aligns reads with up to four mismatches when it looks for ungapped alignments, whereas it does not allow mismatches in the first step when used to look for gapped alignments. It should be pointed out that this constraint refers to the number of mismatches allowed in the alignment when both read and genome are converted using the 3-letter nucleotide alphabet. Users can change this value in GPU-BSM as well as disable ungapped alignments. By default, GPU-BSM generates up to 2 alignments for a read. Users can easily modify this value to decrease, increase, or avoid the upper limit to the alignments that may be found for each sequence. By default, GPU-BSM analyzes only the unique best alignments found by SOAP3-dp. However, GPU-BSM also permits to analyze all valid alignments or all best alignments obtained by SOAP3-dp.

4.2.3 Hardware and Software Requirements

GPU-BSM works on linux based systems, equipped with a custom installation of Python (release>=2.7.3) and with a CUDA enabled GPU-card. It was tested on two families of NVIDIA GPU cards. In particular tests have been carried out on the NVIDIA FERMI architecture based GTX 480 card, and on the NVIDIA Kepler architecture based k10 and k20c cards. Currently, SOAP3-dp can be run on the CUDA-4.2 and CUDA-5.5 releases. As SOAP3-dp has been successfully deployed on some cloud computing services (e.g., Amazon EC2, NIH BioWulf and Tianhe-1A) it is also possible to use the proposed tool on them.

4.3 Results

Experiments have been designed to assess the performances of GPU-BSM to map WGBS and RRBS libraries with both synthetic and real data. In this section, first experiments on synthetic data are introduced, mainly aimed at assessing the reliability of GPU-BSM. Then, evaluation results on real data are presented. Finally, the hardware and software configuration used for experiments is briefly resumed.

4.3.1 Performance evaluation on synthetic data

Synthetic WGBS and RRBS libraries have been generated with the Sherman bisulfite-read simulator (`http://www.bioinformatics.babraham.ac.uk/projects/sherman/`). For the experiments, libraries of different reads length were used. In particular, libraries with reads length of 75 and 120 bp have been generated. Each library consisted of 250 thousands of reads generated from the build 37.3 of the human genome with a uniform bisulfite conversion rate of 50% on both strands. Libraries have been generated simulating the sequencing error rate from 0% to 6% in increments of 2%. So, sixteen libraries were generated: eight synthetic WGBS libraries and eight synthetic RRBS libraries. Specifically, for both WGBS and RRBS, eight libraries were generated: four libraries for reads of length of 75 bp and four libraries for reads of length 120 bp with simulated sequencing errors of 0%, 2%, 4% and 6%, respectively. As for RRBS libraries, an in-silico MspI digestion was performed on the build 37.3, and 40-500 bp fragments were selected.

Sherman simulates sequencing errors using an error rate curve that follows an exponential decay model with the aim to mimic real data. In this way, it will be most likely that the simulated errors are in bases towards the 3' end rather than in bases towards the 5' end.

As for WGBS libraries, GPU-BSM has been compared with Bismark, BSMAP, BS-Seeker2 and segemehl, whereas for RRBS libraries only with Bismark, BSMAP and BS-Seeker2 as segemehl does not support this type of data. To provide an accurate comparison with the other tools, experiments were performed to assess the reliability of GPU-BSM to look for ungapped and gapped alignments. In particular, as for gapped alignments, the performance of GPU-BSM when used to look for global and local alignments was separately assessed. Bismark and BS-Seeker2 have been used with Bowtie to look for ungapped alignments, and with Bowtie2 to look for gapped alignments. BS-Seeker2 with Bowtie2 has been run to look for gapped global and local alignments. BSMAP and segemehl look for gapped global alignments and do not permit to enable or disable this feature.

Tools compared in this work implement different algorithms that do not allow to perform experiments using the same constraints. Then, experiments have been performed setting parameters with the aim to obtain more accurate alignments according to the analyzed library (see Table 4.2). In particular, tools have been run to look for alignments with up to five mismatches. It should be pointed out that Bismark and segemehl do not permit to set the number of mismatches to be allowed; they permit to set the number of mismatches in the seed. Then, in order not to overestimate the performance of these tools, their alignments were analyzed without taking into account those obtained with more than five mismatches.

Very accurate tools will exhibit high precision and high recall (sensitivity). Then, with the goal of providing a rigorous comparison among the tools, the performances of the analyzed tools were compared in terms of unique best mapped reads, precision, and F1. Defined as the harmonic mean between precision *(p)* and recall *(r)*, F1 is a measure that weights equally both metrics. It penalizes systems with a mediocre performance of precision or sensitivity with respect to those that exhibit good performance on both metrics.

tool		
GPU-BSM^u	WGBS	-m 5 –ungapped -l 1
	RRBS	-m 5 –ungapped -l 1 -R -d C-CGG
GPU-BSM^g	WGBS	-m 5 –e2e -l 1
	RRBS	-m 5 –e2e -l 1 -R -d C-CGG
GPU-BSM^{gl}	WGBS	-m 5 -l 1
	RRBS	-m 5 -l 1 -R -d C-CGG
Bismark^u	WGBS	-q –directional
	RRBS	-q –directional
Bismark^g	WGBS	-q –directional –bowtie2
	RRBS	-q –directional –bowtie2
BS-Seeker2^u	WGBS	-m 5 –aligner=bowtie -f sam
	RRBS	-m 5 –aligner=bowtie -f sam -r -c C-CGG -L 40 -U 500
BS-Seeker2^g	WGBS	-m 5 –aligner=bowtie2 –bt2–end-to-end -f sam
	RRBS	-m 5 –aligner=bowtie2 –bt2–end-to-end -r -c C-CGG -L 40 -U 500
BS-Seeker2^{gl}	WGBS	-m 5 –aligner=bowtie2 -f sam
	RRBS	-m 5 –aligner=bowtie2 -r -c C-CGG -L 40 -U 500
segemehl	WGBS	-D 0 -F 1 -H 1
	RRBS	not supported
BSMAP	WGBS	-v 5 -w 2 -r 0
	RRBS	-v 5 -w 2 -r 0 -D C-CGG

Tool settings used to map reads of synthetic libraries. Default settings have been used for not specified parameters.
^{u} *Ungapped alignments. In these experiments Bismark and BS-Seeker2 are used with Bowtie.*
^{g} *Gapped alignments. In these experiments Bismark and BS-Seeker2 are used with Bowtie2.*
^{gl} *Gapped local alignments. In these experiments BS-Seeker2 is used with Bowtie2.*

Table 4.2: **Tool settings used to map synthetic reads.**

4.3.1.1 Performance evaluation on WGBS libraries

Figures 4.7 and 4.8 show the percentage of unique best mapped reads as function of sequencing error for WGBS libraries. In almost all cases GPU-BSM and BS-Seeker2, both run to support local alignments, have been able to map more reads than the other tools. GPU-BSM, when run supporting gapped global alignments was the second tool able to map more reads than the other ones for almost all simulated sequencing errors. In particular, GPU-BSM outperforms the other tools that adopt the same unbiased strategy. As for ungapped alignments, the number of reads mapped by GPU-BSM is comparable with those of Bismark and BS-Seeker2 for simulated sequencing error up to 2%.

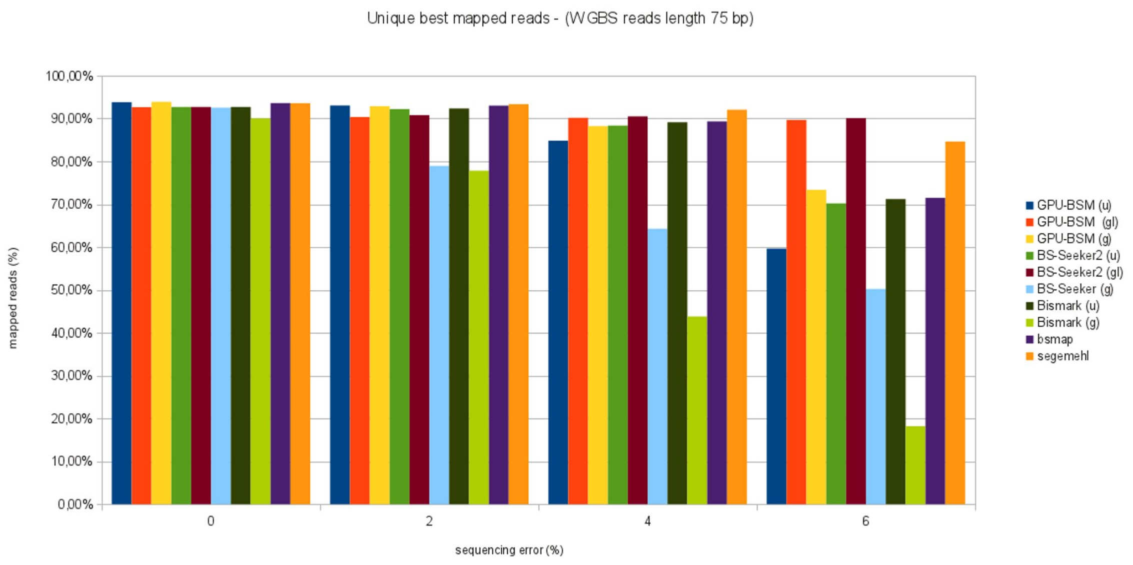


Figure 4.7: **Unique best mapped reads for WGBS libraries with reads length of 75 bp.** The graph represents the percentage of unique best mapped reads obtained for each tool as function of the sequencing error for WGBS synthetic libraries with reads length of 75 bp.

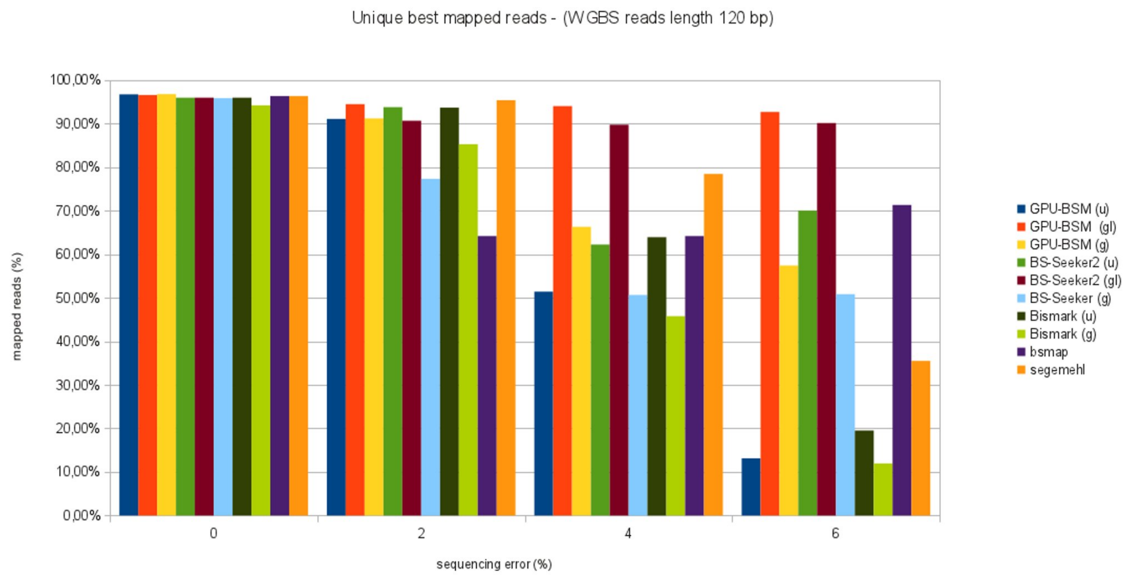


Figure 4.8: **Unique best mapped reads for WGBS libraries with reads length of 120 bp.** The graph represents the percentage of unique best mapped reads obtained for each tool as function of the sequencing error for WGBS synthetic libraries with reads length of 120 bp.

The analysis of precision (see Table 4.3) shows that for local alignments GPU-BSM is more accurate than BS-Seeker2. As for gapped alignments, BSMAP and segemehl outperform the other tools, whereas for ungapped alignments Bismark and BS-Seeker2 are slightly more accurate than GPU-BSM.

	simulated sequencing error			
tool	0%	2%	4%	6%
GPU-BSM^u	93.75%	92.91%	84.48%	58.74%
GPU-BSM^g	93.84%	92.68%	87.89%	72.75%
GPU-BSM^{gl}	92.57%	89.92%	88.64%	89.09%
Bismark^u	92.69%	92.30%	88.99%	70.86%
Bismark^g	90.01%	77.75%	43.50%	17.85%
BSMAP	93.59%	92.96%	89.21%	71.08%
BS-Seeker2^u	92.69%	92.16%	89.36%	88.73%
BS-Seeker2^g	92.52%	78.60%	63.41%	49.28%
BS-Seeker2^{gl}	92.69%	90.08%	89.36%	88.73%
segemehl	93.57%	93.30%	91.96%	84.41%

A) Table reports precision varying the sequencing error from 0% to 6% for 250 thousands of 120 bp reads mapped against the build 37.3 of the human genome.

	simulated sequencing error			
tool	0%	2%	4%	6%
GPU-BSM^u	99.39%	99.16%	98.79%	98.56%
GPU-BSM^g	99.35%	99.09%	99.08%	99.25%
GPU-BSM^{gl}	99.32%	98.62%	98.29%	98.49%
Bismark^u	100%	99.78%	99.57%	98.32%
Bismark^g	100%	99.87%	99.67%	97.76%
BSMAP	100%	99.45%	98.75%	98.07%
BS-Seeker2^u	100%	99.67%	99.65%	98.36%
BS-Seeker2^g	100%	98.61%	98.65%	94.76%
BS-Seeker2^{gl}	100%	95.41%	96.12%	86.55%
segemehl	100%	99.72%	99.50%	99.30%

B) Table reports precision varying the sequencing error from 0% to 6% for 250 thousands of 75 bp reads mapped against the build 37.3 of the human genome.

Table 4.3: **Precision for WGBS libraries with reads length of 75 bp and 120 bp**

F1 measures concerning all experiments on WGBS libraries are reported in Figures 4.9 and 4.10. These graphs show that GPU-BSM, when run to support local alignments, outperforms BS-Seeker2 for all sequencing errors. As for gapped global alignments, GPU-BSM outperforms Bismark and BS-Seeker2 that exploit the same unbiased strategy, whereas for ungapped alignments its performance is comparable with those of Bismark and BS-Seeker2 only for simulated sequencing error of 0% and 2%.

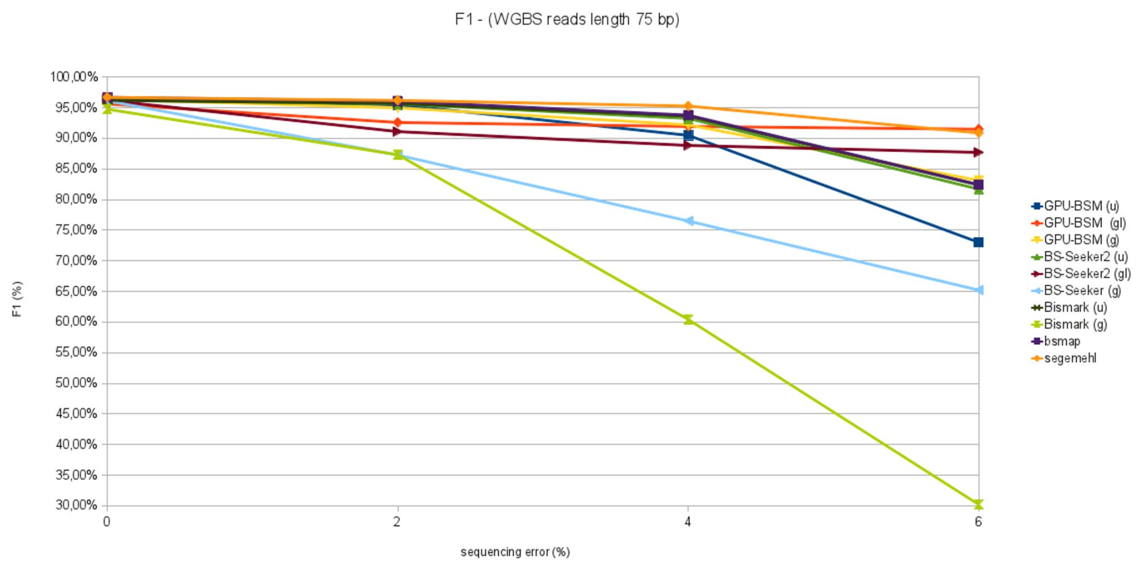


Figure 4.9: **F1 measure analyzing WGBS libraries with reads length of 75 bp.** This figure reports F1 measure varying sequencing error from 0% to 6% for 250 thousands of 75 bp reads mapped against the build 37.3 of the human genome.

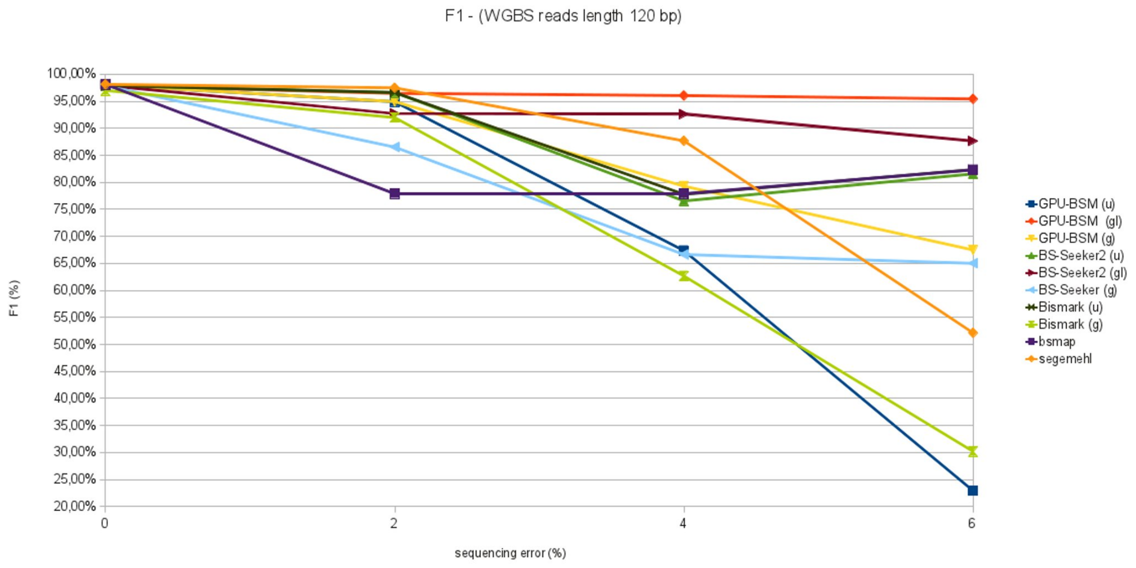


Figure 4.10: **F1 measure analyzing WGBS libraries with reads length of 120 bp.** This figure reports F1 measure varying sequencing error from 0% to 6% for 250 thousands of 120 bp reads mapped against the build 37.3 of the human genome.

Performance evaluation on RRBS libraries

Figures 4.11 and 4.12 show the percentage of unique best mapped reads as function of sequencing error for RRBS libraries. Also in this case, GPU-BSM and BS-Seeker2, both run to support local alignments, have been able to map more reads than the other tools. As for gapped global alignments, in almost all cases BSMAP has been able to map more reads than the other tools, whereas GPU-BSM and BS-Seeker2 mapped more reads than Bismark. As for ungapped alignments, the performance of GPU-BSM is comparable with those of Bismark and BS-Seeker2 for simulated error sequencing up to 2%. For higher simulated sequencing error BS-Seeker2 mapped more reads than the other tools.

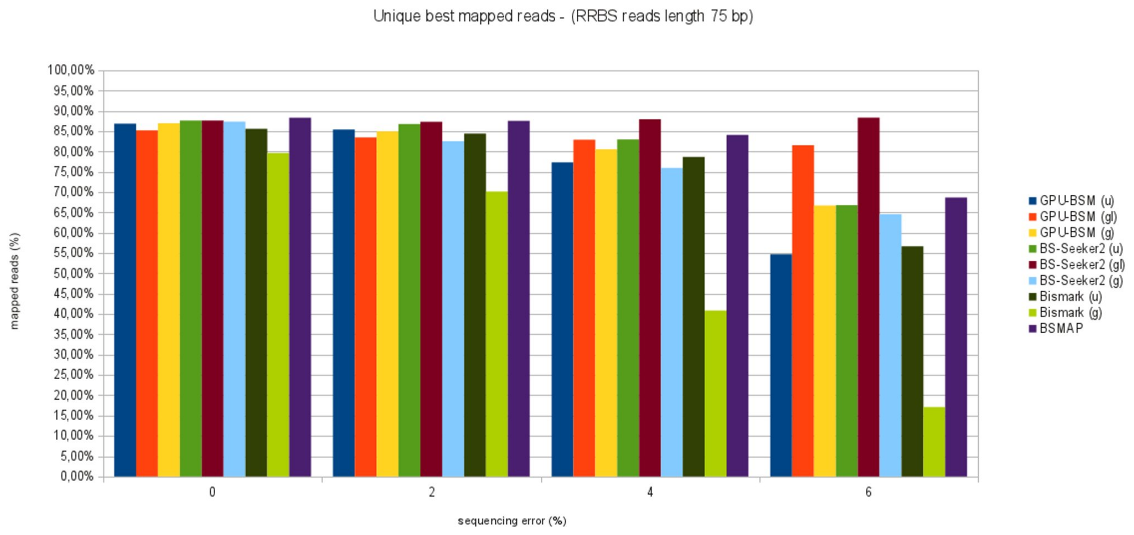


Figure 4.11: **Unique best mapped reads for RRBS libraries with reads length of 75 bp.** The graph represents the percentage of unique best mapped reads obtained for each tool as function of the sequencing error for RRBS synthetic libraries with reads length of 75 bp.

The analysis of precision (see Table 4.4 and Table 4.5) shows that GPU-BSM outperforms BS-Seeker2 when run to look for local alignments. As for gapped alignments, BSMAP and Bismark are more accurate than the other tools for reads of length 75 bp and 120 bp respectively. Bismark shows better precision for ungapped alignments.

F1 measures concerning all experiments on RRBS libraries are reported in Figures 4.13 and 4.14. These graphs show that GPU-BSM, when run to support local alignments, outperforms BS-Seeker2 for all sequencing errors. As for gapped global

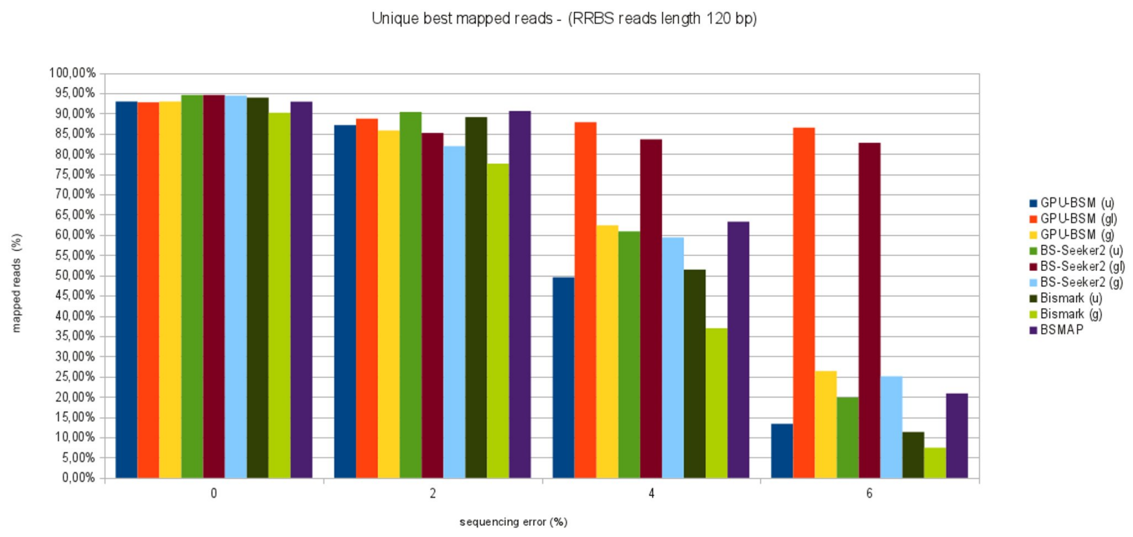


Figure 4.12: **Unique best mapped reads for RRBS libraries with reads length of 120 bp.** The graph represents the percentage of unique best mapped reads obtained for each tool as function of the sequencing error for RRBS synthetic libraries with reads length of 120 bp.

	simulated sequencing error			
tool	0%	2%	4%	6%
GPU-BSM^u	100%	99.26%	98.72%	98.04%
GPU-BSM^g	99.98%	99.02%	98.64%	98.64%
GPU-BSM^{gl}	99.95%	98.02%	97.01%	96.27%
Bismark^u	100%	98.96%	98.60%	98.49%
Bismark^g	100%	99.43%	98.51%	97.38%
BSMAP	99.92%	99.44%	99.10%	98.48%
BS-Seeker2^u	100%	98.72%	97.96%	97.51%
BS-Seeker2^g	100%	97.41%	96.15%	96.36%
BS-Seeker2^{gl}	100%	91.42%	87.37%	85.67%

Table reports precision varying the sequencing error from 0% to 6% for 250 thousands of 75 bp reads mapped against the build 37.3 of the human genome.

Table 4.4: **Precision for RRBS libraries with reads length of 75 bp**

alignments BSMAP outperforms the other tools for simulated sequencing error up to 4%. GPU-BSM outperforms the other tools based on the same mapping strategy for all simulated sequencing errors, and BSMAP for simulated errors of 6%. As for

	simulated sequencing error			
tool	0%	2%	4%	6%
GPU-BSM^u	99.27%	98.88%	98.71%	98.32%
GPU-BSM^g	99.26%	99.08%	99.45%	99.45%
GPU-BSM^{gl}	99.24%	98.74%	98.83%	98.68%
Bismark^u	100%	99.72%	99.67%	99.58%
Bismark^g	100%	99.79%	99.62%	99.26%
BSMAP	99.91%	99.69%	99.44%	99.06%
BS-Seeker2^u	100%	99.66%	99.63%	99.62%
BS-Seeker2^g	100%	99.04%	99.13%	99.26%
BS-Seeker2^{gl}	100%	97.14%	96.30%	95.89%

Table reports precision varying the sequencing error from 0% to 6% for 250 thousands of 120 bp reads mapped against the build 37.3 of the human genome.

Table 4.5: **Precision for RRBS libraries with reads length of 120 bp**

ungapped alignments, BS-Seeker2 outperforms all the other tools.

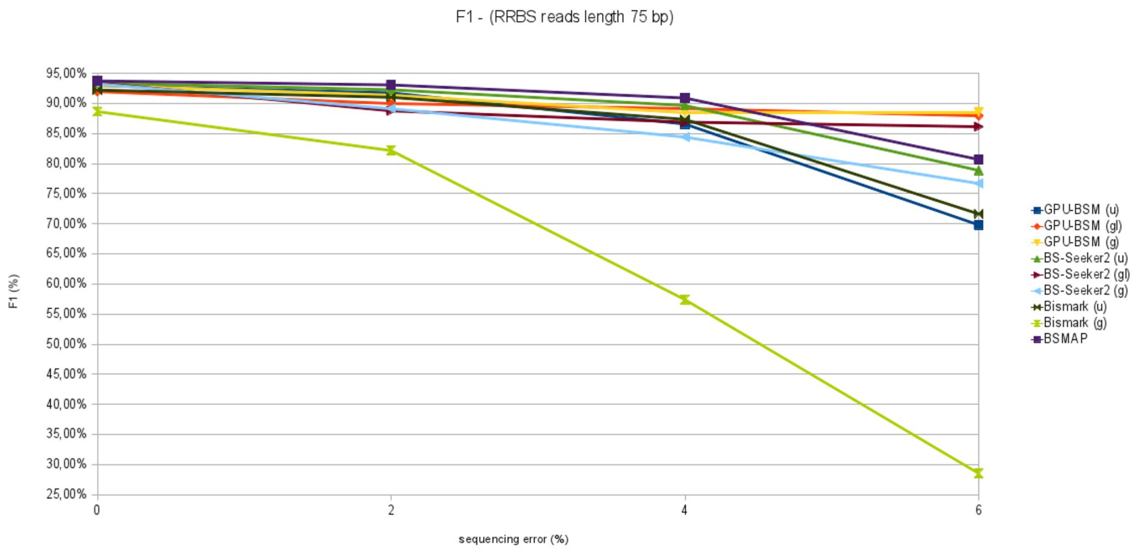


Figure 4.13: **F1 measure analyzing RRBS libraries with reads length of 75 bp.** This figure reports F1 measure varying sequencing error from 0% to 6% for 250 thousands of 75 bp reads mapped against the build 37.3 of the human genome.

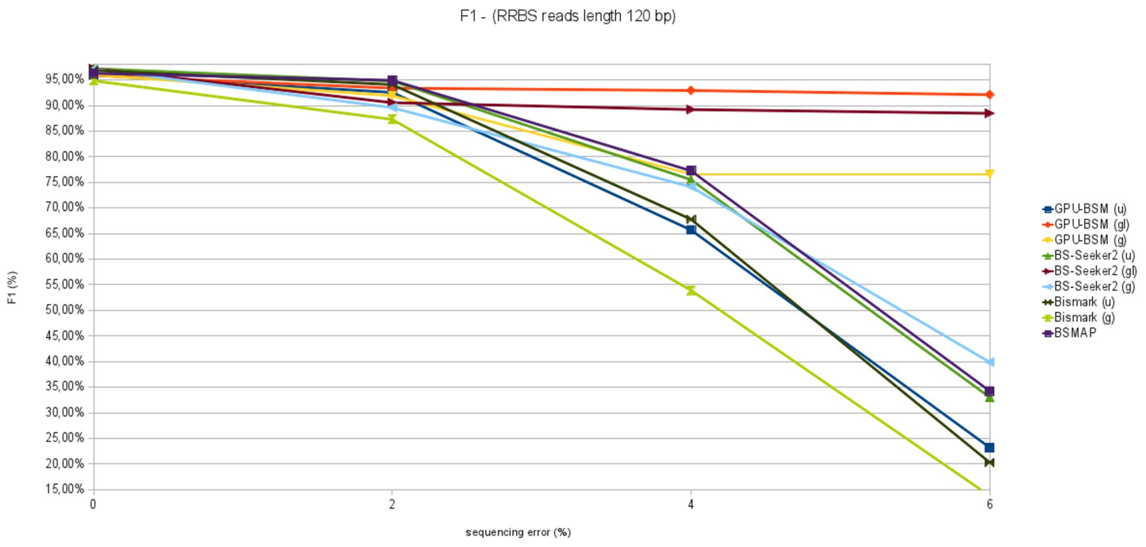


Figure 4.14: **F1 measure analyzing RRBS libraries with reads length of 120 bp.** This figure reports F1 measure varying sequencing error from 0% to 6% for 250 thousands of 120 bp reads mapped against the build 37.3 of the human genome.

4.3.2 Performance evaluation on real data

As for WGBS, to assess the performances of GPU-BSM on real data, it was used to map the reads of two directional libraries obtained by sequencing the human H1 cell line on the Human NCBI genome build 37.3/hg19. The reads of the same real-life libraries analyzed in [36] and [157] were mapped to assess the performance of BS-Seeker and segemehl, respectively: library *SRR019597*, consisting of 5.9 millions of 76 bp reads, and library *SRR019048*, consisting of 15.3 millions of 87 bp reads. Results have been compared with those of Bismark, BSMAP, BS-Seeker2, and segemehl.

As for RRBS, GPU-BSM was used to map against the mus musculus genome (mm9) the reads of the library SRR748751 [173]. The library consists of 11.9 millions of 100 bp reads generated with MspI digestion and selecting fragments of 40-220 bp.

Tables 4.6 and 4.8 summarize experimental results in terms of fraction of unique best mapped reads and computing time for all tools. In the tables have only been reported the percentage of unique best mapped reads for alignments with up to five differences.

Experimental results show that GPU-BSM is a very effective tool for mapping bisulfite-treated reads, as it outperforms almost all analyzed tools. When run to look for ungapped and gapped global alignments, it has been able to map more reads than the other tools in almost all cases. As for unique best mapped reads, its performances are only comparable with those of segemehl for WGBS libraries. GPU-BSM appears to be slightly more effective than segemehl to map reads with few differences. On the other hand segemehl appears to be slightly more effective to map reads with more differences. When run to look for local alignment BS-Seeker2 mapped more reads than GPU-BSM.

As for the computing time, GPU-BSM is definitely the faster tool to map WGBS libraries, and the second to map RRBS libraries. In particular, as for SRR0195957 and SRR019048 WGBS libraries, GPU-BSM ran on a single GPU resulted: *i*) 3.3x/1.25x faster than Bismark and 3x/2.5x faster than BS-Seeker2 when run to look for ungapped alignments; *ii*) 6.6x/4.3x faster than Bismark, 9x/11.2x faster than BS-Seeker2, 3.5x/2.4x faster than segemehl, and 1.4x/3.9x faster than BSMAP when run to look for gapped global alignments; *iii*) 9x/10.6x faster than BS-Seeker2 to map reads with gapped local alignments. As for the SRR748751 RRBS library, GPU-BSM ran on a single GPU resulted: *i*) 1.9x faster than Bismark and 2.8x faster than BS-Seeker2 when run to look for ungapped alignments; *ii*) 2.8x faster than Bismark and 12.3x faster than BS-Seeker2 when run to look for gapped global alignments; *iii*) 7.7x faster than BS-Seeker2 to map reads with gapped local alignments. As for RRBS and gapped global alignments, BSMAP resulted 3.3x faster than GPU-BSM.

4.3.3 Hardware and Software Configuration

Experiments described hereinafter have been carried out on a 12 cores Intel Xeon CPU E5-2667 2.90 GHz with 128 GB of RAM. Two NVIDIA Kepler architecture based Tesla k20c cards with 0.71 GHz clock rate and equipped with 4.8 GB of global memory have been exploited to execute SOAP3-dp rel. 2.3.177.

library	tool	time	percentage of unique best mapped reads					
			$= 0$	≤ 1	≤ 2	≤ 3	≤ 4	≤ 5
SRR019597	GPU-BSM^u	15*/11**m	40.2%	53.0%	58.8%	62.7%	65.8%	66.6%
	GPU-BSM^g	9*/7**m	40.3%	53.7%	60.1%	64.3%	67.8%	70.9%
	GPU-BSM^{gl}	9*/7**m	61.6%	77.7%	83.5%	86.3%	87.8%	88.7%
	Bismark^u	50m	40.3%	52.8%	58.4%	61.9%	64.1%	65.3%
	Bismark^g	1h	39.1%	51.5%	56.9%	57.6%	57.8%	57.9%
	BSMAP	13m	40.0%	52.6%	58.3%	62.0%	64.7%	66.8%
	BS-Seeker2^u	45m	39.9%	52.3%	58.0%	61.8%	64.7%	67.1%
	BS-Seeker2^g	1h22m	38.8%	52.5%	56.7%	59.5.0%	61.9%	64.0%
	BS-Seeker2^{gl}	1h21m	61.1%	77.8%	83.8%	86.8%	88.6%	89.9%
	Segemehl	32m	39.9%	53.3%	59.7%	64.2%	67.9%	71.2%
SRR019048	GPU-BSM^u	44*/32**m	24.7%	33.6%	38.0%	41.4%	44.4%	45.1%
	GPU-BSM^g	23*/19**m	24.7%	34.3%	39.1%	42.9%	46.5%	50.0%
	GPU-BSM^{gl}	22*/19**m	55.2%	69.1%	74.0%	76.6%	78.2%	79.3%
	Bismark^u	55m	24.5%	33.3%	37.4%	39.8%	41.2%	42.0%
	Bismark^g	1h40m	24.0%	32.7%	36.8%	37.4%	37.7%	37.8%
	BSMAP	1h30m	24.6%	33.4%	37.7%	41.0%	43.8%	46.4%
	BS-Seeker2^u	1h51m	24.5%	33.2%	37.4%	40.7%	43.6%	46.2%
	BS-Seeker2^g	4h18m	24.4%	33.2%	37.0%	40.0%	42.9%	45.6%
	BS-Seeker2^{gl}	3h55m	56.2%	71.5%	77.6%	81.2%	83.7%	85.7%
	Segemehl	57m	22.8%	33.4%	38.8%	43.2%	47.2%	51.1%

Performances comparison on real-life libraries among GPU-BSM, Bismark, BSMAP, BS-Seeker2, and segemehl. Two directional libraries are analyzed: *SRR019597*, which consists of 5.943.586 reads of length 76 bp, and *SRR019048*, which consists of 15.331.851 reads of length 87 bp. The first and second column of the table report the library and the name of the tools, respectively. The third column reports the time required to analyze the libraries. Columns 4 to 9 report the percentage of uniquely mapped reads according to the number of mapping differences. Differences are mismatches when the tools are used to look for ungapped alignments, whereas they may be mismatches and/or indels when the tools are used to look for gapped alignments. Computing time for GPU-BSM has been reported running it on a single and on two GPUs. As for multi-threading based tools, computing time has been reported for 12 cores. Tools settings: *i*) GPU-BSM^u -m 5 –ungapped -l 1, GPU-BSM^g -m 5 –e2e -l 1, GPU-BSM^{gl} -m 5 -l 1; moreover for all experiments with GPU-BSM the following settings have been used: -L 76 for SRR019597 and -L 87 for SRR019048, -g 0 to run the experiment on a single GPU (-g 0 -g 1 to run the experiment on two GPUs); *ii*) Bismark^u -q –directional, Bismark^g -q –directional –bowtie2 -p 6***; *iii*) BSMAP -v 5 -w 2 -r 0 -p 12; *iv*) BS-Seeker2^u -m 5 –aligner=bowtie -f sam, BS-Seeker2^g -m 5 –aligner=bowtie2 -f sam –bt2–end-to-end –bt2-p 6***, BS-Seeker2^{gl} -m 5 –aligner=bowtie2 -f sam –bt2-p 6*** *v*) segemehl -F 1 -H 1 -D 0 -A 70 –threads 12.

* *GPU-BSM run on a single GPU*

** *GPU-BSM run on two GPUs*

*** *Bismark and BS-Seeker2 run in parallel two instances of Bowtie2. To ensure that both tools use 12 core it was used the option -p 6/–bt2-p 6 so that each Bowtie2 instance runs with 6 threads.*

Table 4.6: **Performance evaluation on WGBS data**

library	tool	memory
SRR019597	GPU-BSM$^{u/g/gl}$	20.3/20.3/20.3 GB
	Bismark$^{u/g}$	7.7/10.1 GB
	BSMAP	8.3 GB
	BS-Seeker2$^{u/g/gl}$	4.6/7.3/7.3 GB
	segemehl	53 GB
SRR019048	GPU-BSM$^{u/g/gl}$	22.4/40.6/41.6 GB
	Bismark$^{u/g}$	7.7/10.1 GB
	BSMAP	8.3 GB
	BS-Seeker2$^{u/g/gl}$	4.6/7.3/7.3 GB
	segemehl	53 GB
SRR748751	GPU-BSM$^{u/g/gl}$	17.3/27.7/29.5 GB
	Bismark$^{u/g}$	7.7/10.1 GB
	BSMAP	2.1 GB
	BS-Seeker2$^{u/g/gl}$	3.0/3.0/3.0 GB

Peaks of memory required to run experiments on real-life libraries. Data reported in the table shows that GPU-BSM is not very efficient in terms of memory consumption.

Table 4.7: **Memory consumption**

library	tool	time	percentage of unique best mapped reads					
			$= 0$	≤ 1	≤ 2	≤ 3	≤ 4	≤ 5
SRR748751	GPU-BSM^u	18*/15**m	24.7%	32.1%	34.6%	35.9%	37.0%	37.1%
	GPU-BSM^g	20*/12**m	24.7%	32.5%	35.1%	35.6%	37.6%	38.3%
	GPU-BSM^{gl}	27*/18**m	41.8%	50.1%	52.4%	53.4%	54.0%	54.3%
	Bismark^u	34m	24.2%	31.1%	33.2%	33.9%	34.3%	34.6%
	Bismark^g	56m	22.3%	28.9%	31.0%	32.2%	32.5%	32.6%
	BSMAP	6m	24.6%	31.6%	33.7%	34.9%	35.7%	36.3%
	BS-Seeker2^u	52m	24.6%	31.7%	33.7%	34.8%	35.5%	35.9%
	BS-Seeker2^g	4h6m	24.5%	31.8%	34.2%	35.6%	36.7%	37.7%
	BS-Seeker2^{gl}	3h29m	48.9%	59.5%	63.4%	65.7%	67.0%	67.9%

Performances comparison on real-life libraries among GPU-BSM, Bismark, BSMAP, and BS-Seeker2. A directional library *SRR748751*, which consists of 11.961.710 reads of length 100 bp has been analyzed. The first and second column of the table report the library and the name of the tools, respectively. The third column reports the time required to analyze the library. Columns 4 to 9 report the percentage of uniquely mapped reads according to the number of mapping differences. Differences are mismatches when the tools are used to look for ungapped alignments, whereas they may be mismatches and/or indels when the tools are used to look for gapped alignments. Computing time for GPU-BSM has been reported running it on a single and on two GPUs. As for multi-threading based tools, computing time has been reported for 12 cores. Tools settings: *i*) GPU-BSM^u -m 5 –ungapped -l 1 -R -d C-CGG, GPU-BSM^g -m 5 –e2e -l 1 -R -d C-CGG, GPU-BSM^{gl} -m 5 -l 1 -R -d C-CGG; moreover for all experiments with GPU-BSM the following setting has been used: -L 100 -g 0 to run the experiment on a single GPU (-L 100 -g 0 -g 1 to run the experiment on two GPUs); *ii*) Bismark^u -q –directional, Bismark^g -q –directional –bowtie2 -p 6***; *iii*) BSMAP -v 5 -w 2 -r 0 -D -C-CGG -p 12; *iv*) BS-Seeker2^u -m 5 –aligner=bowtie -f sam -r -c C-CGG -L 40 -U 220, BS-Seeker2^g -m 5 –aligner=bowtie2 -f sam -r -c C-CGG -L 40 -U 220 –bt2–end-to-end –bt2-p 6***, BS-Seeker2^{gl} -m 5 –aligner=bowtie2 -f sam -r -c C-CGG -L 40 -U 220 –bt2-p 6***.

* *GPU-BSM run on a single GPU*

** *GPU-BSM run on two GPUs*

*** *Bismark and BS-Seeker2 run in parallel two instances of Bowtie2. To ensure that both tools use 12 core it was used the option -p 6/–bt2-p 6 so that each Bowtie2 instance runs with 6 threads.*

Table 4.8: **Performance evaluation on RRBS data**

4.4 Discussion and Conclusions

GPU-BSM is a mapping tool able to align single-end and paired-end reads generated from WGBS and RRBS. GPU-BSM supports both gapped and ungapped alignments. Massive parallelization on GPUs enables GPU-BSM to map reads without stringent limitations on the alignment process. Experimental results shown that GPU-BSM is very accurate and outperforms most of the cutting-edge solutions in terms of unique best mapped reads, while keeping computational time reasonably low.

It is deemed there are further margins of improvement of the overall computing time. The mapping process implemented in GPU-BSM can be represented by a three-stage pipeline. In the first stage, GPU-BSM performs a 3-letter nucleotide alphabet reduction. Successively, the bisulfite-treated reads are mapped against the reference genome. Finally, GPU-BSM analyzes the mapped reads to detect and remove those ambiguous and false positives. Currently, only the second stage of the pipeline has been parallelized on GPU cards. In particular, the mapping process can be run on up to four GPU cards. At the second stage, the gain in terms of computing time resulted nearly linear with increasing the number of GPU cards. Nevertheless, the overall gain is not linear due to the fact that the first and third stages of the pipeline have not yet been parallelized. It is planned to improve GPU-BSM *i*) porting to GPU the third stage of the pipeline and *ii*) extending the parallelization of the second stage to a cluster of GPUs. Porting to GPUs the analysis performed at the third stage is essential to obtain a linear gain of the computing time with increasing the used GPUs. Without this improvement, there will be no benefit from the parallelization of the second stage on a cluster of GPUs. It is estimated that the planned updates of GPU-BSM can notably improve the computing time. This part of the algorithm was implemented with the aim to easily migrate it on GPU. In doing this, data structures were defined mainly devised for massive parallelization on GPU that are not optimized for CPU. This implied a huge amount of memory required to run it. The peaks of memory required from the different tools were reported in Table 4.7 which shows that only segemehl requires more memory than GPU-BSM. GPU-BSM is freely available for non-commercial use under the terms of the Affero GNU General Public License. The current release can be downloaded at the following addresses:

- `http://pypi.python.org/pypi/GPU-BSM/`
- `http://www.itb.cnr.it/web/bioinformatics/gpu-bsm`

Chapter 5

CUDA-Quicksort: An Improved GPU-based Implementation of Quicksort

Sorting is a very important task in computer science and becomes a critical operation for bioinformatics algorithms that make heavy use of sorting algorithms, in particular when dealing with huge amounts of data. Recently, general-purpose computing has been successfully used on GPUs to parallelize some sorting algorithms. Two high-performance GPU-based implementations of the quicksort algorithm were presented in the literature: a CUDA iterative implementation (i.e., GPU-quicksort), and a recursive implementation (i.e., NVIDIA CUDA Dynamic Parallel (CDP) advanced quicksort). CUDA-Quicksort is another iterative implementation of the sorting algorithm designed for NVIDIA GPUs. CUDA-Quicksort is able to outperform both GPU-Quicksort and NVIDIA CDP-Quicksort in terms of computing time (up to four times faster than the first and about three times faster than the latter). CUDA-Quicksort has been used for the implementation of G-CNV, a tool for preparing data to detect CNVs with read depth methods (see Chapter 6).

5.1 Introduction

Sorting is a very important task in computer science and becomes a critical operation for programs that make heavy use of sorting algorithms, in particular when dealing with huge amounts of data. To reduce computational time when sorting large amounts of data, several algorithms have been devised and implemented. Some are problem-specific, while others exploit parallelization strategies. The latter usually has strong dependence on the adopted hardware and software architecture. Nowadays, conventional parallel architectures have the drawback of being complex and expensive, while not conventional ones, like Graphics Processing Units (GPUs), are increasingly used in scientific computing due to their low cost and their high parallelization capabilities. Recently, general-purpose computing has been successfully used on GPUs to parallelize some sorting algorithms. GPU-Quicksort [33] is one of the first GPU-based implementations of the original quicksort algorithm [73]. At the time that GPU-Quicksort has been presented, recursion was not supported by GPUs and their authors devised an iterative implementation of the algorithm. Recently, the NVIDIA laboratories released a recursive GPU-based quicksort implementation called CDP (CUDA Dynamic Parallel) advanced quicksort [3]. CDP-Quicksort has been developed for the latest GPUs based on the Kepler architecture supporting recursion. Both implementations repeatedly perform two steps on the sequence in hand. In the first step a pivot (say P) is picked and the sequence is partitioned, so that several thread groups can work in parallel on different parts of the sequence. Each thread group calculates the coordinates of two partial subsequences by separating the items with value $< P$ from the items with value $> P$. Then, thread groups are synchronized with the aim to merge their partial results. Two subsequences are then created to separate the items with value $< P$ from those with value $> P$. Each thread group moves the items in the appropriate subsequence. The second step starts when the size of each subsequence is so small that the overhead of using quicksort becomes too high. In this case, the GPU-based bitonic sort is used instead.

As an efficient synchronization is hardly achievable given the high quantity of threads used in GPUs, the communication among thread groups and the inter-thread synchronization are critical issues for implementing this algorithm. In GPU-Quicksort, the synchronization problem is addressed through a block-oriented iterative implementation, where each partition is processed by one thread block. To keep the inter-block synchronization low, the first step of the algorithm is divided in two phases: the first phase proceeds as the above-mentioned step, while the second phase starts when the size of a subsequence allows to entirely process the subsequence by one thread block. This phase differs from the first, as here there is no need to partition the sequence and to provide synchronization among blocks. The CDP-Quicksort exploits a warp-oriented recursive implementation instead, which uses an inter-warp synchronization based on atomic primitives [151] (i.e., barrier-functions supported starting from the NVIDIA Fermi architecture).

In this work a new block-oriented iterative GPU-based implementation of the quicksort called CUDA-Quicksort is proposed, which uses atomic primitives to perform inter-block synchronizations while performing an optimized access to the GPU memory. Experimental results show that the proposed implementation outperforms both the GPU-Quicksort and the CDP-Quicksort in terms of computing time. The rest of this work is organized as follow: in the next section, the architecture and programming model of GPUs is presented. Section 5.2 gives an overview of the related work. Section 5.3 describes the proposed implementation of the quicksort algorithm. Section 5.4 illustrates and discusses experimental results. In section 5.5 conclusions and future work are discussed. The software is freely available at http://sourceforge.net/projects/gpu-quicksort.

5.2 Related Work

A first effort to provide a quicksort suitable for GPUs has been proposed in [172]. In this case results were not encouraging: sorting computational time was an order of magnitude slower than other sorting algorithms they used for comparison. The first high-performance implementation of quicksort for GPU, called GPU-Quicksort, has been designed by minimizing the amount of bookkeeping and inter-thread synchronization. It is well known that quicksort is a recursive algorithm based on the divide-and-conquer paradigm. It recursively picks a pivot from a sequence and successively moves items with value lower than the pivot to the "left" and items with value higher than the pivot to the "right". GPU-Quicksort works in a similar manner. It parallelizes quicksort by exploiting a straightforward approach [69] [185] which divides the sequence in hand in different partitions and dynamically assigns them to available processors.

The first step of the GPU-Quicksort repeatedly performs two phases on the given sequence. In the first phase, the algorithm picks a pivot (say P) and partitions the sequence, so that several thread blocks can work in parallel on different parts of the sequence. Each thread in the block iterates through all the data of its assigned partition, keeping track of the number of elements that are greater or lesser than P. This information is stored in two arrays of the shared memory (see Figure 5.1.A). Then, each thread block calculates the GPU-based prefix-sum [68] of these two arrays, so that each thread knows the relative offset where to move items that are higher or lower than the pivot (see Figure 5.1.B). Successively, thread blocks are synchronized, so that each thread block knows the absolute offset where to move items. In the inter-block synchronization, the CPU waits for the completion of each thread block, then calculates another prefix-sum (see Figure 5.1.C). Finally, when each thread of a block knows its offset, the items $< P$ and $> P$ are moved in their respective slice (see Figure 5.1.D). Successively, the items whose value is equal to P are written between the two subsequences. After a fixed number of N iterations, GPU-Quicksort starts the second phase. On average, in this phase the size of each subsequence is such that it can be entirely processed by a thread block. This phase differs from the first, in that there is no need to partition the sequence and to provide synchronization among blocks.

In the second step, when subsequences are very small, the NVIDIA GPU-based bitonic sort is used to obtain the final result.

Recently, NVIDIA proposed a recursive implementation of quicksort for GPU, called CDP advanced quicksort, based on a recent technology called Dynamic Parallelism. It allows a GPU kernel to call another GPU kernel, so that the GPU can work more autonomously from the CPU by generating new work for itself at run-time. The dynamic parallelism allows the implementation of recursive algorithms, unsupported until now. It is only available in the most recent Kepler NVIDIA architectures.

CDP-Quicksort and GPU-Quicksort algorithms are very similar. As in the latter algorithm, CDP-Quicksort partitions the sequence to be sorted. However, in CDP-Quicksort each partition is processed by a thread warp. Therefore, to move $\leq P$ and $> P$ items in their respective subsequences a prior inter-warp synchronization is required. Summarizing, the algorithm proceeds as follows: *i*) each warp counts the number of elements $\leq P$. This number is used to update an atomic counter, so that each warp knows in which slice of the subsequence it can move the items; *ii*) an inter-thread synchronization is performed through a single-operation warp scan (i.e., a warp prefix-sum), so that each thread knows its own offset within this slice; *iii*) when each thread of a warp knows its relative offset, the items are moved in their assigned slice. The same steps are used to move elements $> P$. As in the GPU-Quicksort, when subsequences are very small, the final result is obtained through a GPU-based bitonic sort [6].

GPU-based solutions have also been proposed in literature to efficiently parallelize other sorting algorithms. In [163], a GPU-based bitonic merge sort has been presented, based on the implementation proposed in [88]. In [92] and [93] a bitonic and an odd-even merge sort have been presented. In [62] two sorting solutions have been developed: a solution based on the periodic balanced sorting network [143] and a solution based on the bitonic sorting network [61]. In [64], an approach for parallel sorting on stream processing architectures based on adaptive bitonic sorting, called GPU-ABISort, has been presented. Another algorithm for fast sorting large lists that makes use of GPUs has been presented in [177]. In this work, the authors designed a vector-based mergesort using CUDA, designed to work on four

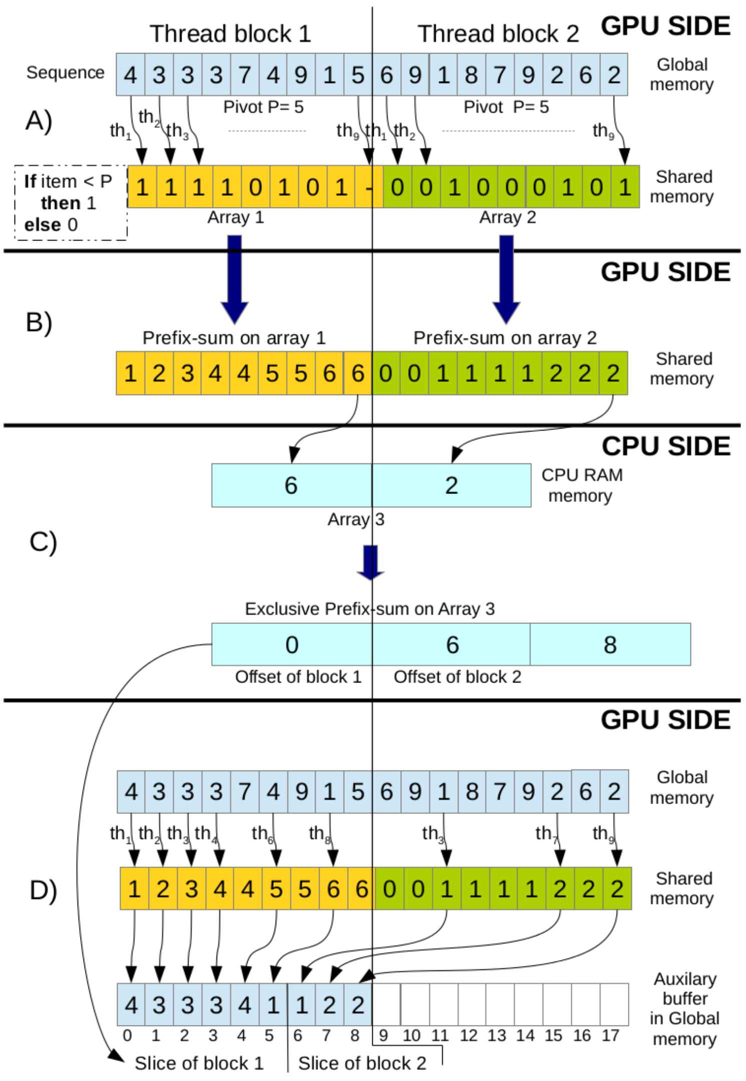


Figure 5.1: **Example of the GPU-Quicksort algorithm for a sequence of 18 elements**. The sequence is partitioned in two 9-thread blocks. A) Each thread compares its related item with P. Then, results of comparison are stored in the shared memory (block 1 in array 1, block 2 in array 2). B) Each thread block calculates the prefix-sum on array 1 and on array 2. C) The CPU waits for the completion of each thread block. Then, the CPU stores in array 3 the number of elements that are lesser than the pivot associated to each block. Finally, the CPU calculates the exclusive prefix-sum on array 3 to calculate the slice offset of each block. D) Each thread of a block gets its offset from the shared memory and moves the thread-associated item $< P$ in its respective slice. The same algorithm is used to move items $> P$.

32-bit floats simultaneously, resulting in a 4 times speed improvement compared to mergesorting. In [107], the authors described the design and implementation of a sample sort algorithm for GPUs CUDA enabled.

5.3 Methods

A new block-oriented iterative GPU-based implementation of the quicksort is proposed, which uses atomic primitives to perform inter-block synchronizations while guaranteeing an optimized access to the GPU memory. The proposed solution is an improvement of GPU-Quicksort. The main difference being that the first step is not divided in two phases. In the first step, a pivot is picked out and the sequence is partitioned to let several thread blocks work in parallel. Initially, thread blocks sort their assigned partition independently from each other (i.e., from the rest of the sequence). In particular, each thread block creates two subsequences in the shared memory to separate items with value lower than the pivot from those whose value is higher. Then, the inter-synchronization is used to get an unambiguous slice of an auxiliary buffer in the global memory where the subsequences will be written. Finally, all subsequences that keep track of items lower than the pivot are merged into a single subsequence. The same task is performed while merging all subsequences that keep track of the items with value higher than the pivot. Then, the items whose value is equal to the pivot are written between the two subsequences. These two subsequences are sorted in parallel, and the sorting process can be started over again on each of them independently. The second step starts when subsequences are very small. GPU-based bitonic sort, based on NVIDIA CUDA Samples 6.0, is used to perform the sorting. Figure 5.2 describes these steps.

5.3.1 Partitioning

In GPU-Quicksort, sequence partitioning is based on the number of thread blocks which is fixed a priori independently from the sequence size. The partition element size depends on the sequence size and is calculated as the ratio of the sequence size to a fixed number of thread blocks. In the proposed solution, the partition element size is fixed a priori and the number of thread blocks depends on the sequence size. The number of thread blocks is calculated as the ratio of the sequence size to a fixed partition element size, which is equal to the size of the shared memory necessary to store elements $< P$ and $> P$. In CUDA-Quicksort the number of threads in a block is lower than the number of items in the partition. In this case, a thread block is

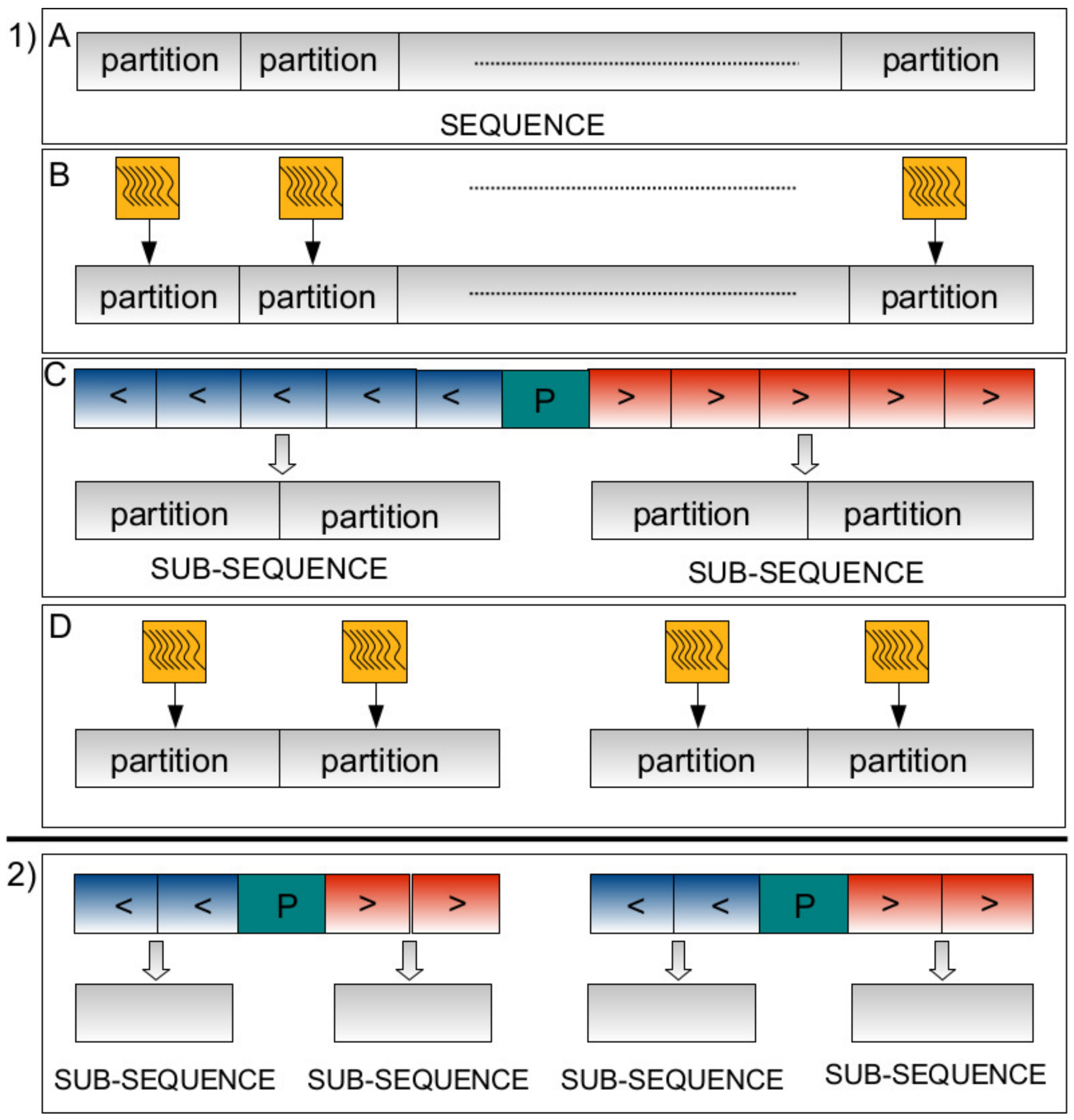


Figure 5.2: **CUDA-Quicksort Algorithm**. 1) First step: a) the sequence is partitioned so that several thread blocks can work in parallel on different parts; b) a thread block is assigned to each different partition element; c) the partial result of each thread block is merged in two subsequences; d) new subsequences are partitioned and assigned to threads blocks; 2) second step: subsequences are so small that each can be assigned to a single thread block. As a final step, bitonic sort is used to finalize the sorting (not represented in the figure).

unable to process a partition in a single run, so it divides and processes the partition in different tiles.

5.3.2 Inter-block Synchronization Using Atomic Primitives

In [33], the need for synchronization among thread blocks has been highlighted. As previously described, they divide the first step of the algorithm in two phases. In the first phase, sequences are typically very large and several thread blocks may work on different partitions. Thread blocks process their partition independently from the other ones, even though they are responsible to contribute to merge their partial results with all other ones. Then, appropriate synchronization among thread blocks is required to merge together the resulting subsequences. Atomic primitives could be used to synchronize thread blocks in this phase. The authors of GPU-Quicksort assessed the opportunity to use them. However, as atomic primitives were not widely supported by GPUs, they decided not to use them with the aim to provide a more generalized solution for GPU. Hence, in their algorithm synchronization is guaranteed by simply waiting the completion of each thread block. This implies that the host is responsible for the synchronization of thread blocks, resulting in a decrease of the overall performance. This effect depends on *i*) the serialization of part of this phase, and *ii*) the increase of the time spent for transferring data from device to host and vice versa. To keep the inter-block synchronization low, a second phase has been implemented. Indeed, the second phase starts after N iterations of the first phase, when the size of each subsequence is generally such that it can be processed by a thread block. This phase differs from the first, as each thread block is assigned its own subsequence, thus eliminating the need for inter-block synchronizations. In so doing, the synchronization issue is dealt with; however there is no guarantee that the generated subsequences will be sufficiently small to be processed in this phase by one thread block. Of course, in the event that a very big subsequence is processed by only one thread block, a decrease of the overall performance occurs. Nowadays, the access to atomic primitives is widely provided by GPUs. In the NVIDIA CDP-Quicksort, partitions are processed by thread warp and the inter-warp synchronization is performed by atomic primitives. This solution has a high atomic issue rate, leading to a decrease of the overall performance. A block-oriented solution is proposed, which uses the atomic primitives to synchronize thread blocks. In doing so, the computing power of the GPU is used, while providing a full parallelization of the first step of the algorithm and reducing this high atomic rate issue.

5.3.3 Optimizing Memory Access

In GPU-Quicksort, each thread block uses a prefix-sum to calculate the new coordinates of the items to be moved to the left or to the right of the pivot. The output of the prefix-sum is stored in the shared memory. An auxiliary region of the global memory is allocated to store the new subsequences in parallel. Then, each thread in a block *i*) accesses the shared memory to read the new coordinates of its assigned item, *ii*) gets the item value in the sequence, and *iii*) writes it to the proper subsequence on the global memory. In doing so, global memory is accessed in write without any wariness of guaranteeing coalesced access. It is commonly known that uncoalesced access to the global memory may substantially affect the overall performance. Coalesced memory access is guaranteed if consecutive threads access consecutive global memory addresses. However, as each thread in a block may be assigned to an item lower or higher than the pivot, consecutive threads may access very distant memory regions (see Figure 5.3.A). The NVIDIA CDP-Quicksort proceeds in a quite similar way, with the difference that calculations are more fine-grained as they are performed at a thread-warp level rather than at a thread-block level. Sequence partitioning in thread warps allows using warp vote functions which permit an inter-thread synchronization without using the shared memory. In particular, in the CDP-Quicksort each thread of a warp exploits a warp prefix-sum to calculate the offset of the items to be moved to the left or to the right of the pivot. Then, when each thread of a warp knows its offset, it writes its items in the appropriate subsequence in the global memory. Also in this case, access to the global memory is uncoalesced. The problem of sorting items is overcame in the shared memory before writing them to the global memory. In particular, in the proposed solution each thread performs the prefix-sum, then: *i*) accesses the shared memory to read the new coordinates of its assigned items; *ii*) gets the items value in the sequence and writes it to the proper subsequence in the shared memory; *iii*) updates the atomic counter and writes its new assigned item to the proper subsequence in the global memory. In this way, consecutive threads in a block read sorted items from the shared memory and write them to consecutive addresses of the global memory (see Figure 5.3.B).

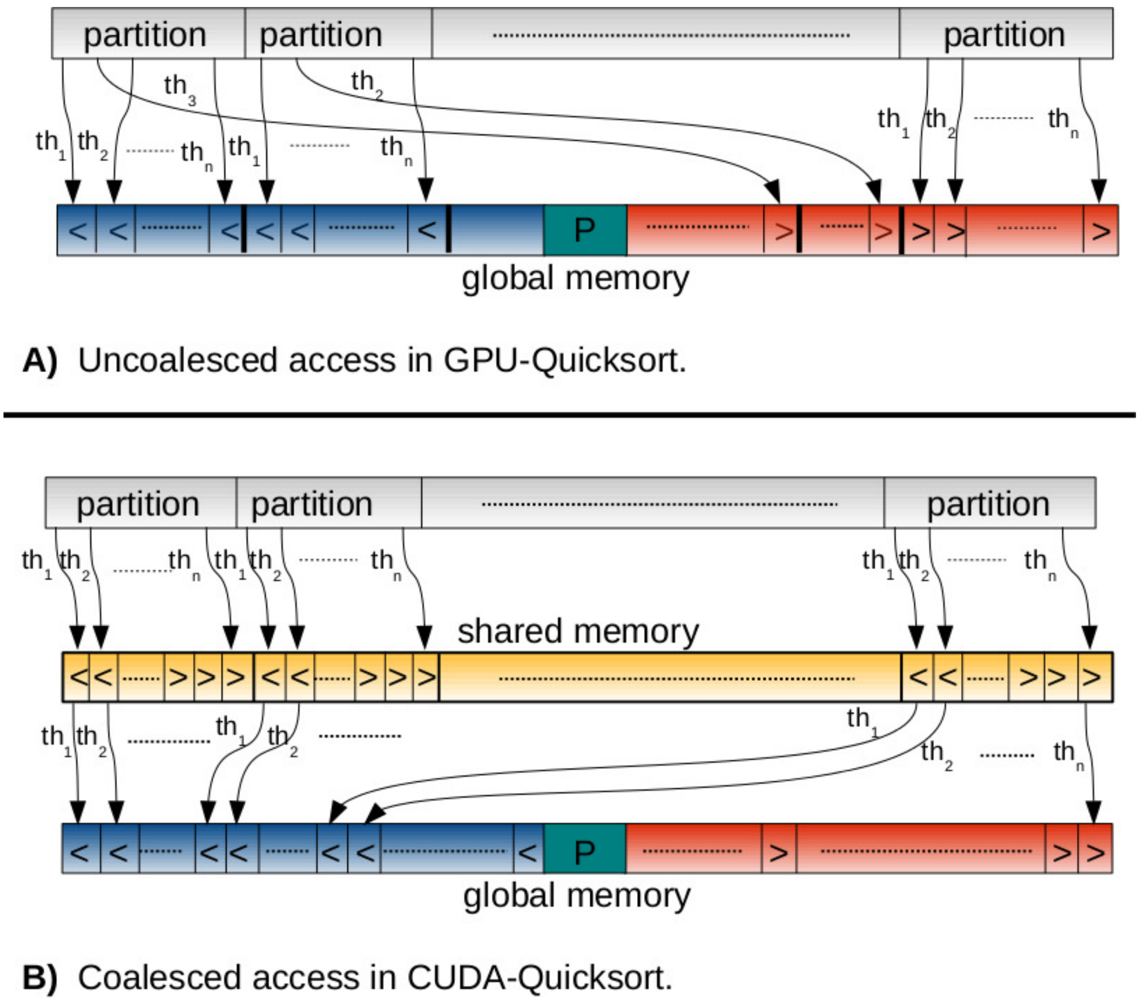


Figure 5.3: **Uncoalesced access in GPU-Quicksort vs.coalesced access in CUDA-Quicksort**. A) Uncoalesced access in GPU-Quicksort. Each thread in a block may move its item in not consecutive global memory addresses. B) Coalesced access in the proposed solution. Items are sorted in the shared memory before being written in the global memory.

5.4 Results

Experiments have been carried out on a 12-core Intel Xeon CPU E5-2667 2.90 GHz and a GPU NVIDIA Tesla Kepler k20. Six sorting benchmarck distributions were used (see Figure 5.4) used in [33] to evaluate GPU-Quicksort.

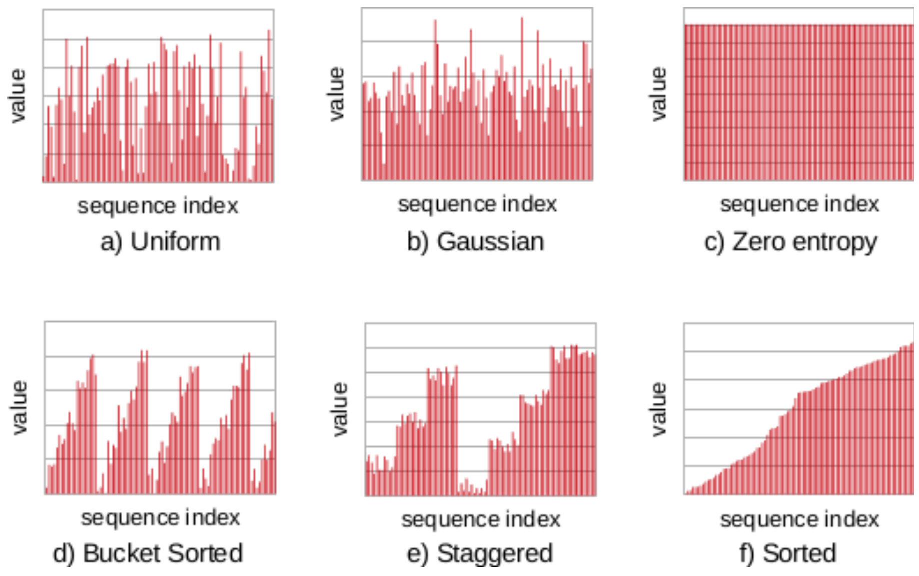


Figure 5.4: **Sorting benchmarks distributions**. a) A uniformly distributed input obtained with random values from 0 to 2^{31}; b) a Gaussian distributed random input created calculating the average of four randomly generated values; c) a zero entropy input, created by setting every value to a random constant value; d) an input sorted into p buckets, such that the first $\frac{n}{p^2}$ elements in each bucket are random numbers in $[0, \frac{2^{31}}{p} - 1]$, the second $\frac{n}{p^2}$ elements in $[\frac{2^{31}}{p}, \frac{2^{32}}{p} - 1]$, and so forth; e) after that a dataset is divided into p partitions. Then, if the partition index *i-th* is $\leq \frac{p}{2}$ ($> \frac{p}{2}$) to their items will be assigned a random value between $(2i-1)\frac{2^{31}}{p}$ and $(2i)(\frac{2^{31}}{p} - 1)$ ($(2i-p-2)\frac{2^{31}}{p}$ and $(2i-p-1)\frac{2^{31}}{p} - 1$); f) sorted uniformly distributed values.

These benchmarks are commonly used in literature to compare the performance of different sorting algorithms [70]. Performance was evaluated on 32-bit integer and 64-bit floating point sequences, varying their size from 1M to 32M elements (only power-of-2 sizes).

To assess to which extent the proposed changes affect the overall performance of the GPU-based quicksort, the average time required to carry out the different tasks of the first step of CUDA-Quicksort and CDP-Quicksort and the first phase of GPU-Quicksort were measured. Summarizing, both the first step of the two first algorithms and the first phase of the last algorithm are composed of the following tasks: *i*) picking out the pivot; *ii*) partitioning the sequence; *iii*) reading subsequences and calculating their coordinates; *iv*) synchronizing thread groups; and *v*) writing subsequences in the global memory; *vi*) preparation for the next quicksort execution. In the assessment performed, summing together the time required to perform the first four tasks was considered appropriate. Figure 5.5 shows a comparison of the time required to perform these tasks in GPU-Quicksort, CDP-Quicksort and CUDA-Quicksort for different sequence sizes. For the sake of brevity, only the behavior for the uniform distribution has been represented.

As shown in Figure 5.5, CUDA-Quicksort performs better than GPU-Quicksort and CDP-Quicksort, in both synchronization and writing tasks. However, it can be observed that in CUDA-Quicksort the time required to perform the first four steps tends to become higher than that required in the GPU-Quicksort for very long sequences (reported *in blue* in Figure 5.5). This worsening is mainly due to the greater quantity of GPU-Quicksort blocks used in the proposed solution, which however provides an outstanding improvement in the fifth step (reported *in red* in Figure 5.5), which yields an overall improvement.

Figure 5.6 shows the single iteration speed-up of the CUDA-Quicksort first step against the CDP-Quicksort first step and the GPU-Quicksort first phase. Figure 5.6 shows that, with a uniform distribution and when sorting 32M elements, it is up to 5.8 times faster than GPU-Quicksort and about 1.9 times faster than CDP-Quicksort. Similar results have been obtained with other distributions, except for the zero entropy one (see Figure 5.6 for details). As highlighted in the same figure, when sorting a uniformly distributed sequence, the algorithm speed-up of CUDA-Quicksort against GPU-Quicksort is about 3.7x. This is mainly due to the fact that the first step of CUDA-Quicksort is only twice faster than the second phase of GPU-Quicksort. On the contrary, the speed-up of the proposed solution algorithm against CDP-Quicksort is about 3.7x. This is mainly due to the fact that the two solutions choose a different pivot (depending on the value of the pivot, the quicksort complexity varies between *nlog* and n^2). CUDA-Quicksort uses a mean value between the minimum and the maximum, while CDP-Quicksort just uses a random value of the sequence.

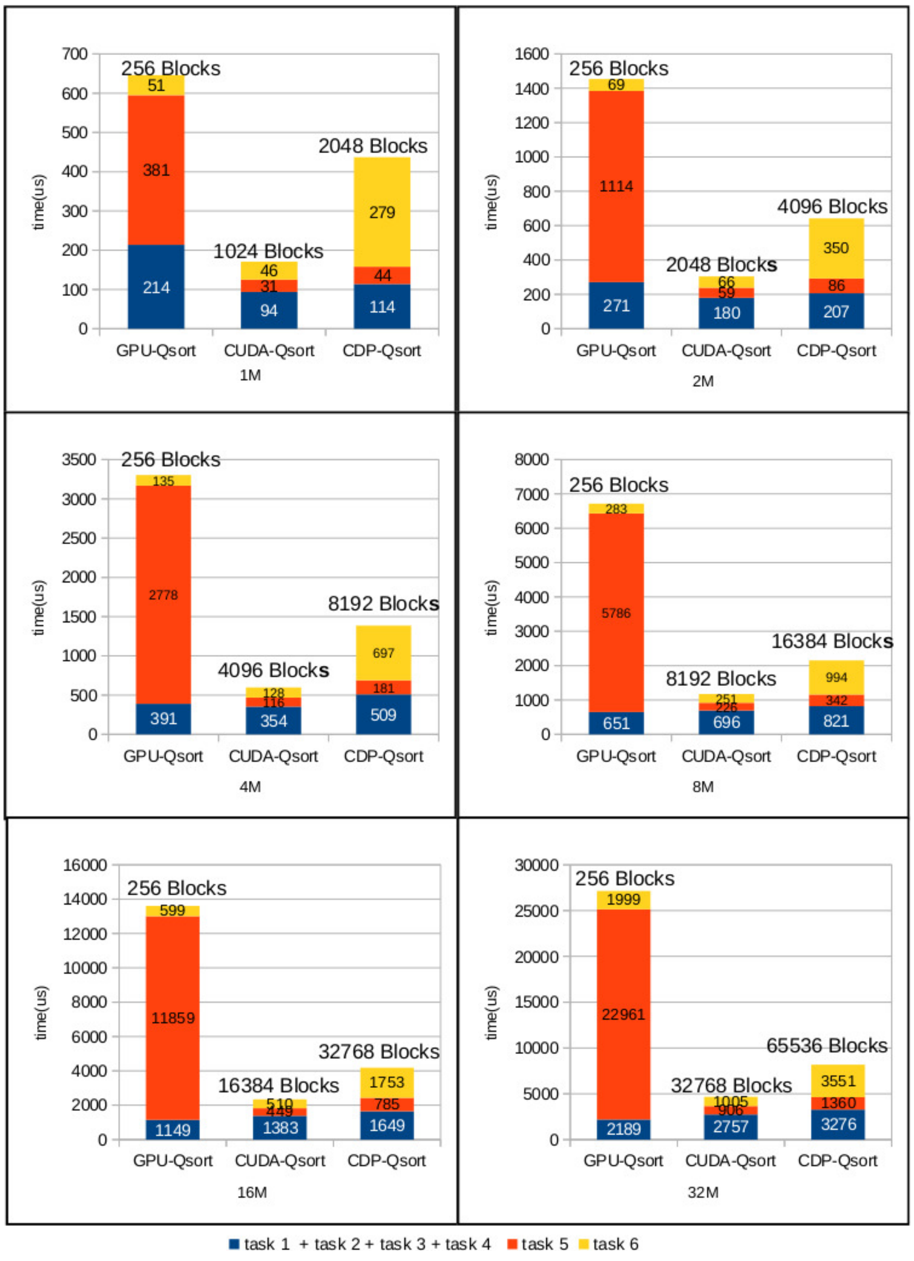


Figure 5.5: **Time required to perform the first step**. *Task 1*: picking out of the pivot; *Task 2*: sequence partitioning; *Task 3*: creation of two subsequences for each thread block; *Task 4*: thread block synchronization; *Task 5*: writing subsequences in the global memory; *Task 6*: preparation for the next quicksort execution. Kernel sizing: GPU-Qsort and CUDA-Qsort [128 threads per block], CDP-Qsort [512 threads per block].

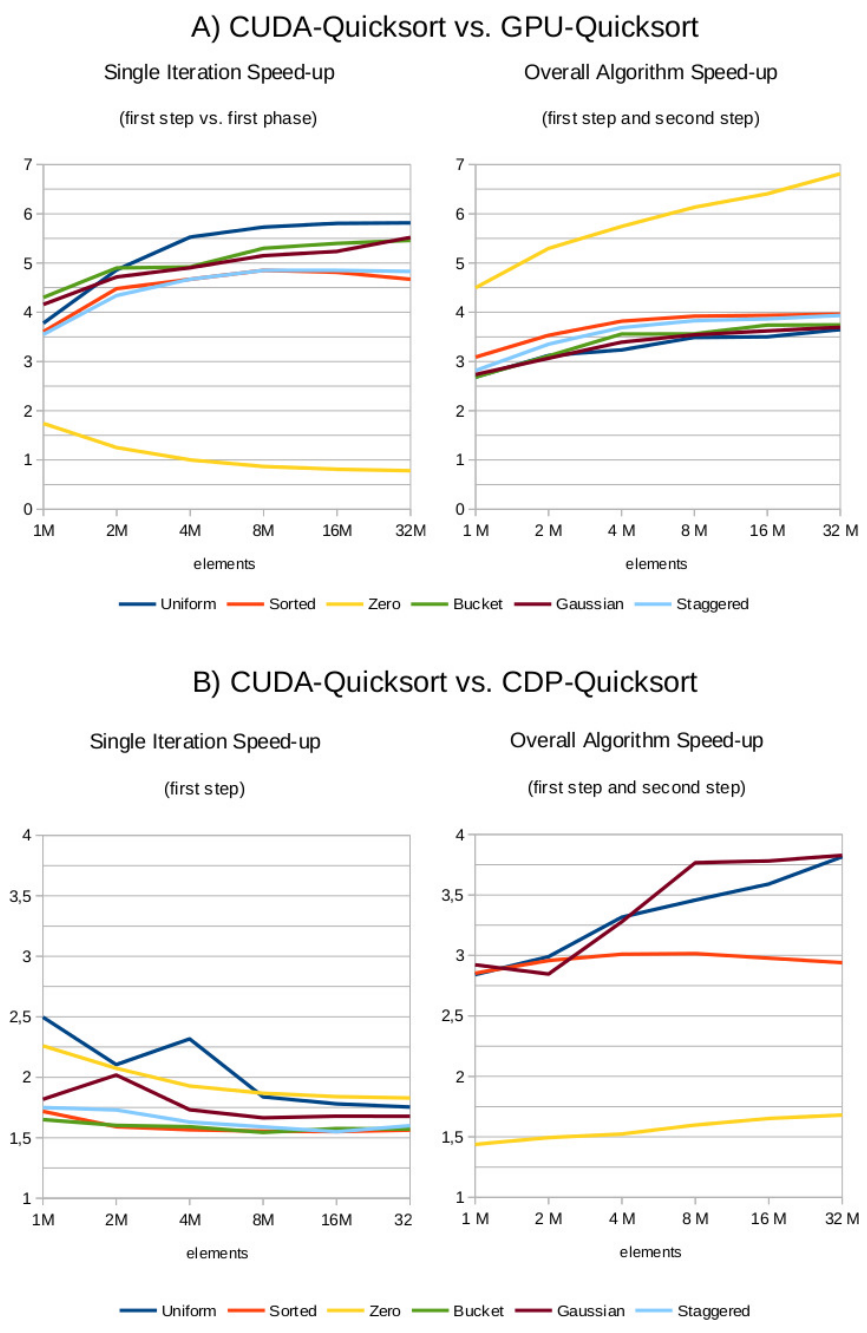


Figure 5.6: **Single iteration speed-up and overall algorithm speed-up**. The single iteration speed-up of the CUDA-Quicksort first step against the CDP-Quicksort first step and the GPU-Quicksort first phase is reported for different benchmarks on the left side. The speed-up of the overall algorithm (first step and second step) is reported on the right side. The gaussian and staggered distributions are not shown in B, as CUDA-Quicksort is 60x faster than CDP-Quicksort.

In order to perform a straightforward comparison between CUDA-Quicksort and CDP-Quicksort and to assess the opportunity of exploiting the CUDA dynamic parallelism, a recursive version of the proposed algorithm called CUDA-R-Quicksort was developed. As for the quicksort single iteration performance, Figure 5.7 shows that the CUDA-Quicksort solution and the CUDA-R-Quicksort solution have a similar computing time and that both are faster than the NVIDIA solution. As for the overall performance, Figure 5.7 shows that the proposed iterative solution is the fastest. For the sake of brevity, only the behaviour for uniform distributions has been represented, being representative of the expected behaviour on the other benchmark distributions.

The performance of CUDA-Quicksort was assessed in comparison with those of the following GPU-based sorting algorithms: GPU-Quicksort, CDP-Quicksort, the radix sort of the Thrust Library [74], based on [168], bitonic sort [24] and merge sort [168]. Bitonic sort and merge sort are provided in the NVIDIA CUDA Samples 6.0. All sorting algorithms used for benchmarking purposes are implemented in CUDA.

Figure 5.8 and Figure 5.9 show that CUDA-Quicksort outperforms almost all cited algorithms. Only the Radix Sort outperforms CUDA-Quicksort when sorting 32-bit and 64-bit items (see Figure 5.10(a)). This was an expected result. For the sake of completeness, let us recall that the computational complexity of the not camparison based radixsort is *O(n)* while that of the quicksort is *O(n log n)*. However, the advantage of the quicksort with respect to the radixsort is that it can be used to sort any set of elements where it is need to compare two elements. When used to sort structured data, radixsort requires to map the items onto keys (e.g., integers) which may not always be possible and the performance can decrease with respect to that of tue quicksort. Therefore, we also performed experiments aimed at comparing both algorithms to sort structured data. In particular, we used both the Thrust Radix Sort and CUDA-Quicksort to sort 96-bit data structure items. Figure 5.10(b) shows that CUDA-Quicksort outperforms the Thrust Radix Sort achieving a speed-up ranging from 1.58x to 2.18x. In the figure have been plot the performance only for the benchmarcks with a uniform distribution of the data. Similar behaviour has been obtained for the other benchmarcks. It should be observed, that 96-bit data structure are not supported by the other sorting algorithms used in the comparison.

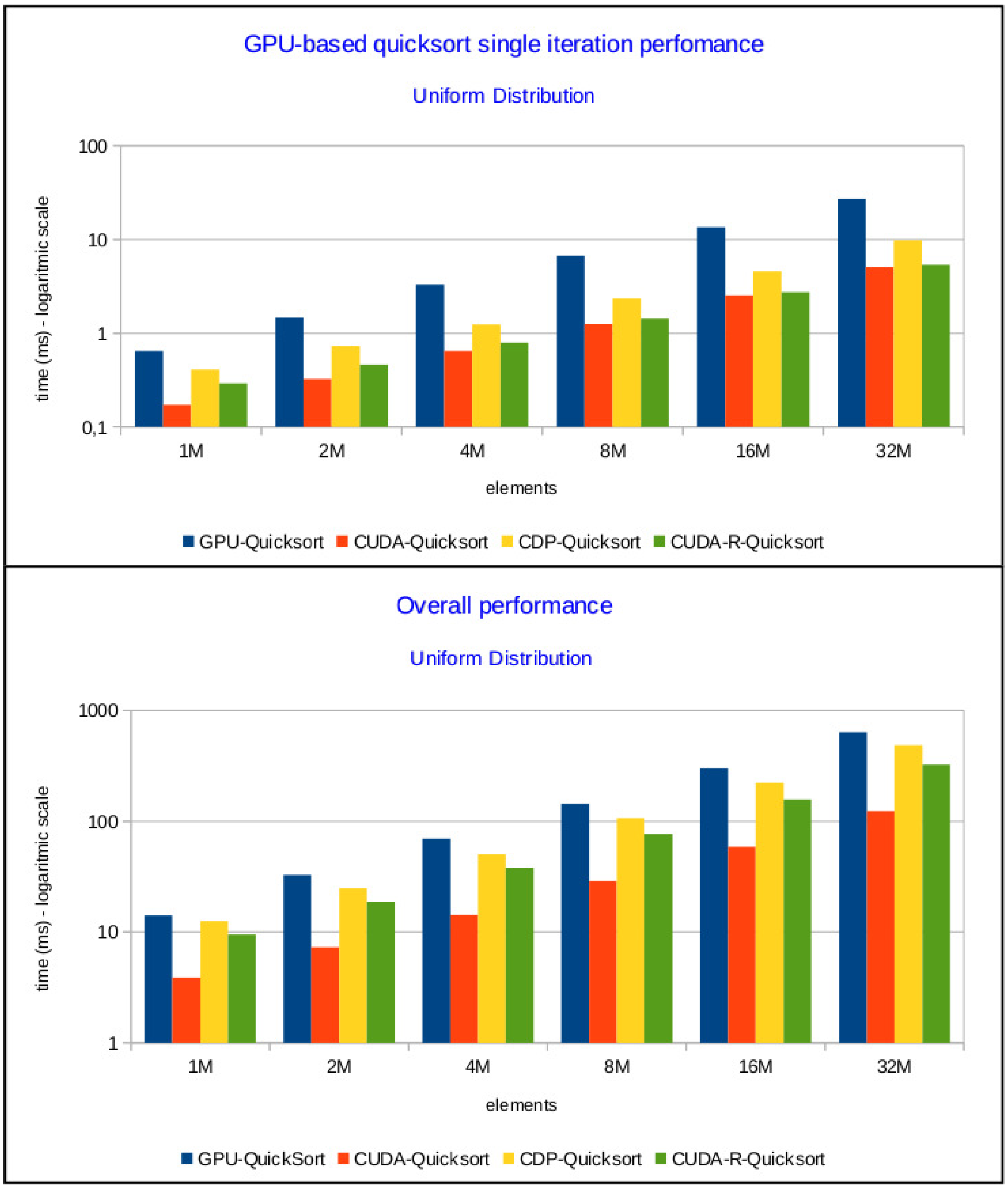


Figure 5.7: **Comparison among CUDA-Quicksort, CUDA-R-Quicksort, GPU-Quicksort and CDP-Quicksort.**

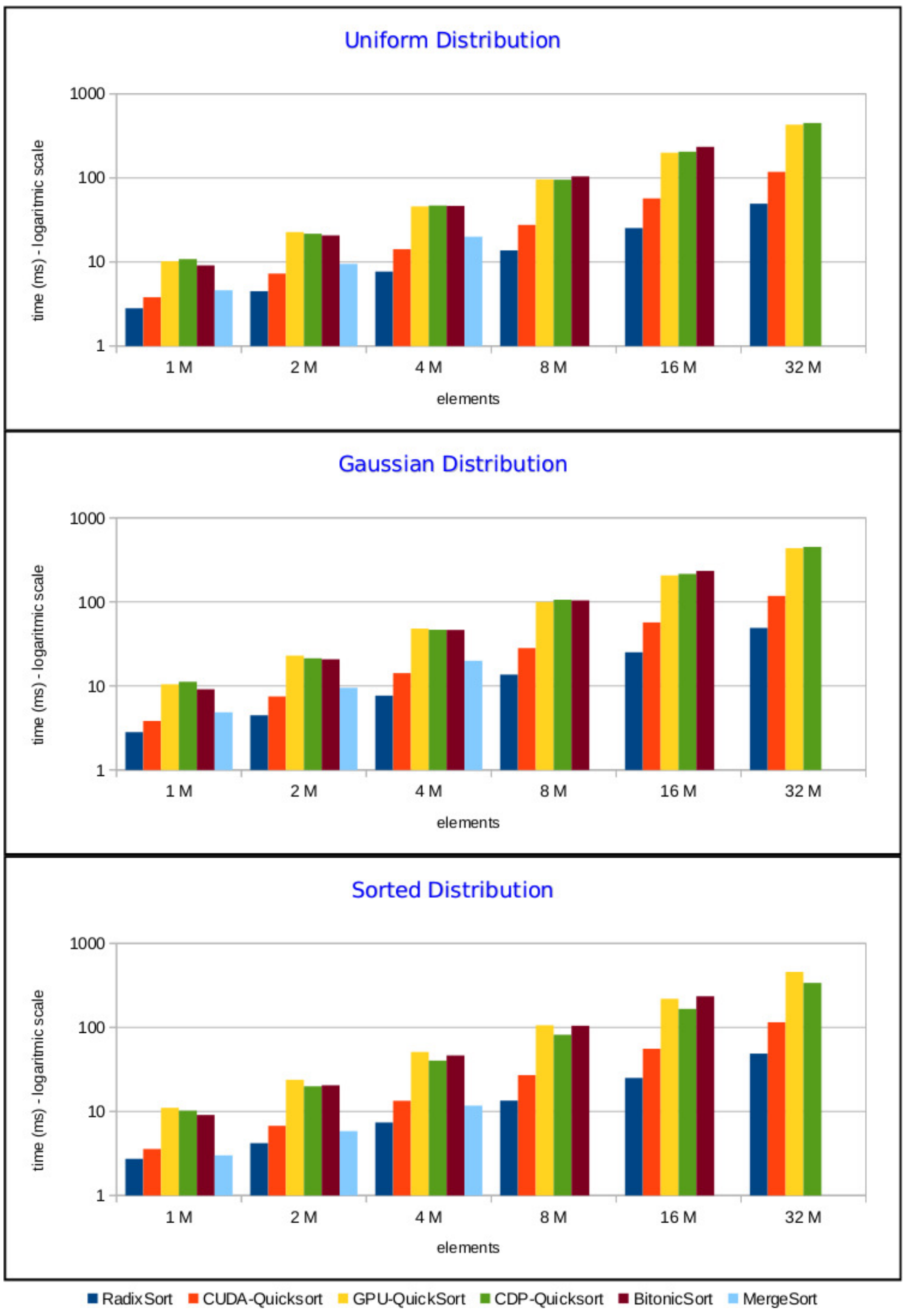


Figure 5.8: **Comparison among CUDA-Quicksort and the other GPU-based sorting algorithms**. MergeSort does not work when sorting 8M elements or more. BitonicSort does not work when sorting 16M elements or more.

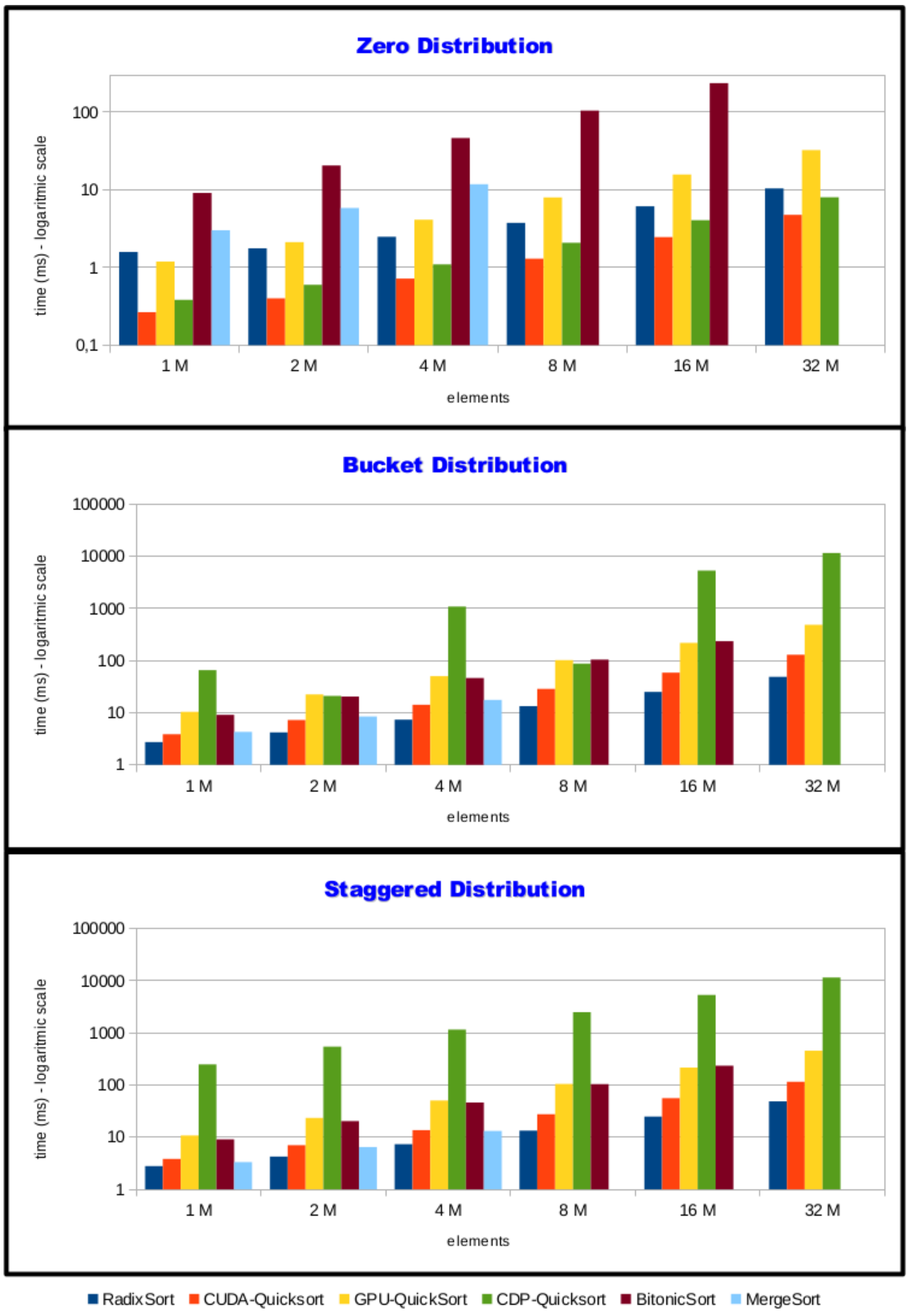
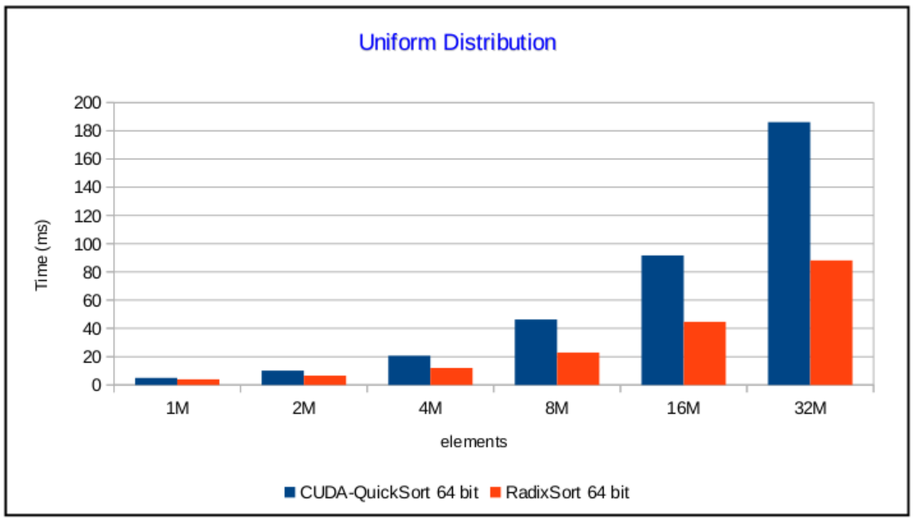
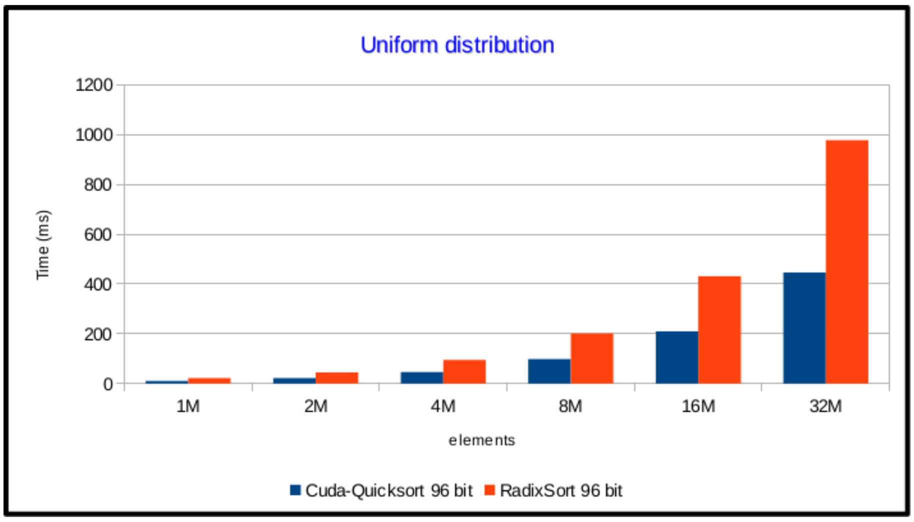


Figure 5.9: **Comparison among CUDA-Quicksort and the other GPU-based sorting algorithms**. MergeSort does not work when sorting 8M elements or more. BitonicSort does not work when sorting 16M elements or more.



(a) Performance on 64-bit floating point sequences.



(b) Performance on 96-bit data structure sequences.

Figure 5.10: **Comparison among CUDA-Quicksort and RadixSort**. Performance was evaluated on 64-bit floating point and on 96-bit data structure sequences.

5.5 Discussion and Conclusions

It is presented a new GPU-based implementation of the quicksort algorithm – designed to take advantage of computing power of modern NVIDIA GPUs. To improve the performance, a block-oriented iterative GPU-based implementation of quicksort was designed. The proposed solution uses atomic primitives to perform inter-block synchronization and is able to optimize the access to global memory. Using atomic primitives in a block-oriented solution allows to outperform the GPU-based quicksort, by reducing the high atomic issue rates of the NVIDIA warp-oriented solution. The increase of the performance is mainly due to a coalesced memory access. Experimental results show that the algorithm is about four times faster than GPU-Quicksort and three times faster than CDP-Quicksort. As for future work, it is planned to redesign the algorithm to provide a new release able to run on multiple GPUs.

Chapter 6

G-CNV: A GPU-based Tool for Preparing Data to Detect CNVs with Read Depth Methods

Copy Number Variations (CNVs) are the most prevalent types of structural variations (SVs) in the human genome and are involved in a wide range of common human diseases. Different computational methods have been devised to detect this type of SVs and to study how they are implicated in human diseases. Recently, computational methods based on high throughput sequencing (HTS) are increasingly used. The majority of these methods focus on mapping short-read sequences generated from a donor against a reference genome to detect signatures distinctive of CNVs. In particular, a class of these methods detect CNVs by analyzing genomic regions with significantly different read-depth from the other ones. Read-depth (RD) is the average number of reads representing a given nucleotide in the reconstructed sequence. The pipeline analysis of these methods consists of four main stages: 1) data preparation, 2) data normalization, 3) CNV regions identification, and 4) copy number estimation.

Data preparation consists of different tasks aimed at assessing the quality of the read sequences, mapping the reads against the reference genome, removing low mapping quality sequences, and sizing the observing window used to calculate the RD signal. Data normalization is aimed at correcting the effect of two sources of bias that affect the detection of CNVs. In particular, it has been proved that correlation exists between RD and the GC-content (the percentage of G and C nucleotides in the observed genomic region); the RD increases with the GC-content of the underlying genomic region. Moreover, it exists a mappability bias due to the repetitive regions in a genome. A read can be mapped to different positions so that ambiguous mappings must be dealt with. After normalization, RD data are analyzed to detect the boundaries of regions characterized by changed copy number. Finally, DNA copy number of each region within breakpoints is estimated.

The first two stages of the analysis pipeline consist of common operations, whereas the last two consist of specific operations for each method. However, it should be pointed out that available tools do not implement most of the operations required at the first and second stage. Typically, these tools start the analysis by building the RD signal from the post-processed alignments. All preparatory operations must be performed by the researchers using third-party tools. Moreover, other tools require annotation files with information about the GC-content that are pre-computed only for some reference genome builds. Only some tools provide limited functionalities to pre-process alignments. For instance RDXplorer and CNV-seq use the samtools to remove low quality mappings and to select the best hit location for each mapped read sequence, respectively.

Most of these operations are data-intensive and can be parallelized to be efficiently run on GPUs. G-CNV is a GPU-based tool devised to perform the common operations required at the first two stages of the analysis pipeline. G-CNV is able to filter low quality nucleotides, to remove duplicated read sequences using CUDA-Quicksort, to map the short-reads, to resolve multiple mapping ambiguities, to build the read-depth signal, and to normalize it. G-CNV can be efficiently used as a third-party tool able to prepare data for the subsequent read-depth signal generation and analysis. Moreover, it can also be integrated in CNV detection tools to generate read-depth signals.

6.1 Introduction

SVs in the human genome can influence phenotype and predispose to or cause diseases [48, 49]. Single nucleotide polymorphisms (SNPs) were initially thought to represent the main source of human genomic variation [167]. However, following the advances in technologies to analyze genome, it is now acknowledged that different types of SVs contribute to the genetic makeup of an individual. SV is a term generally used to refer different types of genetic variants that alter chromosomal structure as inversions, translocations, insertions and deletions [80]. SVs such as insertions and deletions are also referred as CNVs. CNVs are the most prevalent types of SVs in the human genome and are implicated in a wide range of common human diseases including neurodevelopmental disorders [137], schizophrenia [182] and obesity

[27]. Studies based on microarray technology demonstrated that as much as 12% of the human genome is variable in copy number [160], and this genomic diversity is potentially related to phenotypic variation and to the predisposition to common diseases. Hence, it is essential to have effective tools able to detect CNVs and to study how they are implicated in human diseases.

Hybridization-based microarray approaches as a-CGH (array Comparative Genomic Hybridization) and SNP microarrays have been successfully used to identify CNVs [31]. The low cost of a-CGH and SNP platforms promoted the use of microarray approaches. However, as pointed out in [13] microarrays *i*) have limitations in the task of detecting copy number differences, *ii*) provide no information on the location of duplicated copies, and *iii*) are generally unable to resolve breakpoints at the single-base-pair level. Recently, computational methods for discovering SVs with HTS [94] have also been proposed [134]. These methods can be categorized into alignment-free (i.e., de novo assembly) and alignment-based (i.e., paired-end mapping, split read, and read depth) approaches [208]. The former [144, 82] focus on reconstruct DNA fragments by assembling overlapping short reads. CNVs are detected by comparing the assembled contigs to the reference genome. The latter focus on mapping short read sequences generated from a donor against the reference genome with the aim of detecting signatures that are distinctive of different classes of SVs. Mapping data hide useful information that can be used to detect different SVs. Different methods that analyze different mapping information have been devised.

Paired-end mapping (PEM) methods [35, 99, 77, 78, 176, 139] identify SVs/CNVs by detecting and analyzing paired-end reads generated from a donor that are discordantly mapped against the reference genome. These methods allow to detect different types of SVs (i.e., insertions, deletions, mobile element insertions, inversions, and tandem duplications), but they do not allow to detect insertions larger than the average insert size of the library preparations.

Split read (SR) methods [200, 10, 9, 205] are also based on paired-end reads. Unlike PEM methods that analyze discordant mappings, SR methods analyze unmapped or partially mapped reads as they potentially provide accurate breaking points at the single-base-pair level for SVs/CNVs.

Read depth (RD) methods [37, 195, 201, 138, 11, 192, 83] are based on the assumption that the RD in a genomic region depends on the copy number of that region. In fact, as the sequencing process is uniform, the number of reads aligning to a region follows a Poisson distribution with mean directly proportional to the size of the region and to the copy number (see Figure 6.1) [37]. These methods analyze the RD of a genome sequence through non overlapping windows, with the aim of detecting those

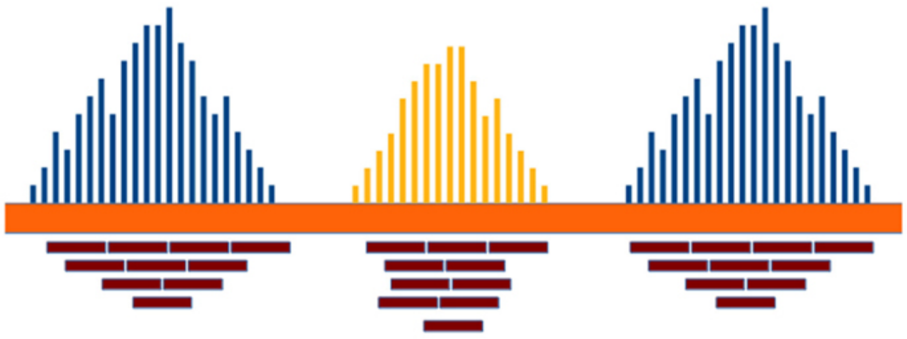


Figure 6.1: **The RD in a genomic region depends on the copy number of that region and follows a Poisson distribution**. Duplicated and deleted regions are characterzed by a RD signal different from that of the other ones.

regions that exhibit a RD significantly different from the other ones. A duplicated region will differ from the other ones for a higher number of reads mapping on it, and then for a higher RD. Conversely, a deleted region will differ from the other ones for a lower number of reads mapping on it, and then for a lower RD. Basically, the analysis pipeline implemented in RD methods consists of four fundamental stages [123]: *i*) data preparation; *ii*) data normalization; *iii*) CNV regions identification; and *iv*) copy number estimation (see Figure 6.2).

Data preparation consists of different tasks aimed at assessing the quality of the read sequences, mapping the reads against the reference genome, removing low mapping quality sequences, and sizing the observing window used to calculate the RD signal. Data normalization is aimed at correcting the effect of two sources of bias that affect the detection of CNVs. In particular, it has been proved that correlation exists between RD and the GC-content [71, 42, 66]; the RD increases with the GC-content of the underlying genomic region. Moreover, it exists a mappability bias due to the repetitive regions in a genome. A read can be mapped to different positions so that ambiguous mappings must be dealt with. After normalization, RD data are analyzed to detect the boundaries of regions characterized by changed copy number. Finally, DNA copy number of each region within breakpoints is estimated.

The first two stages of the analysis pipeline consist of common operations, whereas the last two consist of specific operations for each method. However, it should be pointed out that available tools do not implement most of the operations required at the first and second stage. Typically, these tools start the analysis by building the RD signal from the post-processed alignments. All preparatory operations must be performed by the researchers using third-party tools. Moreover, other tools as ReadDepth [138] require annotation files with information about the GC-content

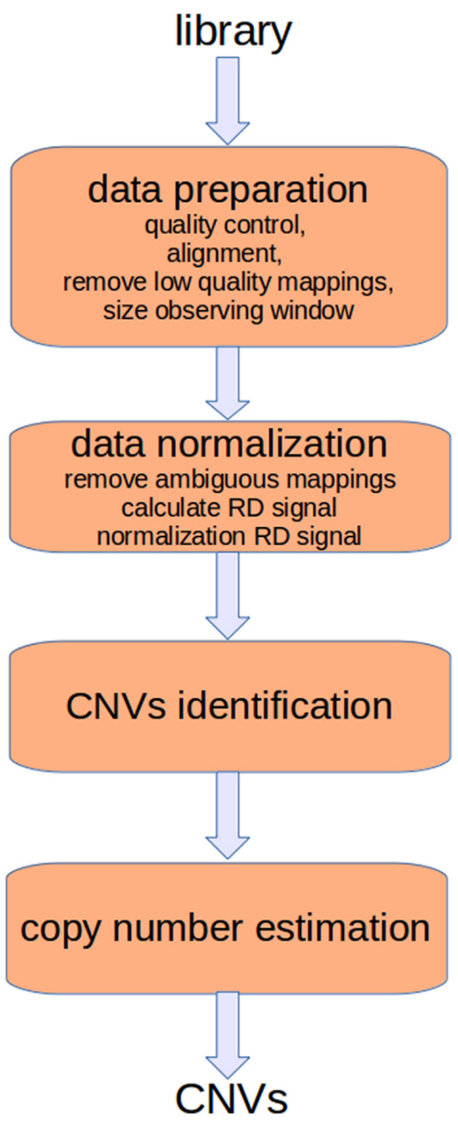


Figure 6.2: **The four main stages of the analysis pipeline of RD-based methods**. The first two stages consist of preparatory operations aimed at generating the RD signal. Sequencing produces artifacts that affects the alignment and consequentely the RD signal. Different filtering operators can be applied to reduce these errors. Moreover, alignments must be post-processed to remove those of low quality and to resolve ambiguities. Finally, the RD signal is calculated taking into account the bias related with the GC-content.

that are pre-computed only for some reference genome builds. Only some tools provide limited functionalities to pre-process alignments. For instance RDXplorer [201] and CNV-seq [195] use the samtools [110] to remove low quality mappings and to select the best hit location for each mapped read sequence, respectively.

Most of these operations are data-intensive and can be parallelized to be efficiently run on GPUs to save computing time. GPUs are hardware accelerators that are increasingly used to deal with computationally intensive algorithms. Recently, GPU-based solutions have been proposed to cope with different bioinformatics problems (e.g., [125, 203, 119, 175, 207, 126, 127]).

In this work G-CNV (GPU-Copy Number Variation) is presented, a GPU-based tool aimed at performing the preparatory operations required at the first two stages of the analysis pipeline for RD-based methods. G-CNV can be used to *i)* filter low quality sequences, *ii)* mask low quality nucleotides, *iii)* remove adapter sequences, *iv)* remove duplicated reads, *v)* map read sequences, *vi)* remove ambiguous mappings, *vii)* build the RD signal, and *viii)* normalize it. Apart the task of removing adapter sequences all the other tasks are implemented on GPU. G-CNV can be used as a third-party tool to prepare the input for available RD-based detection tools or can be integrated in other tools to efficiently build the RD signal.

G-CNV is freely available for non-commercial use. The current release can be downloaded at the following address `http://www.itb.cnr.it/web/bioinformatics/gcnv`.

6.2 Material & Methods

Data preparation and data normalization are crucial operations to properly detect CNVs. It is widely known that sequencing is a process subject to errors. These errors can affect the alignments; hence both the RD signal and the accuracy of the identified CNVs can be affected as well. G-CNV implements filtering operators aimed at correcting some errors related to the sequencing process. In particular, G-CNV is able to analyze the read sequences to filter those read sequences that do not satisfy a quality constraint, to mask low quality nucleotides with a aNy symbol, to remove adapter sequences, and to remove duplicated read sequences. G-CNV uses *cutadapt* [130] to remove adapter sequences. As for the alignment, G-CNV uses the GPU-based short-read mapping tool SOAP3-dp [121]. Low quality alignments are filtered out, while ambiguous mappings can be treated according to different strategies. To build the RD signal, G-CNV builds a RD signal according to a fixed-size observing window. Then, this raw RD signal is corrected according to the GC-content of the observed windows.

In this section, first it is given a short introduction to GPUs. Then, the strategies adopted to cope with the tasks implemented by G-CNV are presented. Finally, the hardware and software equipment required to use G-CNV is briefly recalled.

6.2.1 Quality Control

The sequencing technology has been notably improved. Modern sequencers are able to generate hundreds of millions of reads in a single run and the sequencing cost is rapidly decreasing. Despite this improvement, sequencing data are affected by artifacts of different nature that may strongly influence the results of the research. Hence, the ability to assess the quality of read sequences and to properly filter them are major factors that determine the success of a sequencing project. In particular, as for RD methods, both low quality and duplicated read sequences affect the RD signal and consequently the identification of CNV regions.

Different tools have been proposed for quality control of sequencing data such as NGS QC Toolkit [159], HTQC [199], FASTX-Toolkit [1], FASTQC [2], and Picard [3]. Most of these tools support both Illumina and 454 platforms, while only some of them support CPU parallelization. It should be pointed out that the artifacts generated during the sequencing process and the massive amount of generated reads make quality control tasks difficult and computationally intensive. The massive parallelization that can be provided by GPUs can be used to deal with these computational tasks. Starting from this assumption, G-CNV was integrated with GPU-based operators to filter low quality sequences, to mask low quality nucleotides, and to detect and remove duplicated read sequences. Only the removing of adapter sequences has not yet been implemented on GPU. Currently, these operators are specialized for short-read sequences generated with Illumina platforms.

6.2.1.1 Filtering low quality sequences

FASTQ files report quality values for each sequence. Basically, G-CNV parses these files to identify low quality nucleotides. Nucleotides are classified as of low quality if their quality value is lower than a user-defined threshold. FASTQ files represent

[1]`http://hannonlab.cshl.edu/fastx_toolkit/`

[2]`http://www.bioinformatics.babraham.ac.uk/projects/fastqc/`

[3]`http://broadinstitute.github.io/picard/`

quality values using an ASCII encoding. Different encodings are used depending on the Illumina platform. Illumina 1.0 format encodes quality scores from -5 to 62 using ASCII 59 to 126. From Illumina 1.3 and before Illumina 1.8, quality scores ranges from 0 to 62 and are encoded using ASCII 64 to 126. Starting in Illumina 1.8, quality scores range from 0 to 93 and are encoded using ASCII 33 to 126.

G-CNV performs filtering in three steps. The first step is performed on CPU, whereas the last two steps are massively parallelized on a single GPU. As for the first step, G-CNV analyzes the FASTQ files to detect the Illumina format. Then, the quality values of sequences are decoded according to the detected Illumina format. Finally, G-CNV removes those read sequences that exhibit a percentage of low quality nucleotides that exceed a used defined threshold. As a final result a new FASTQ file is created with the filtered sequences so that the original FASTQ file is preserved.

6.2.1.2 Masking low quality nucleotides

G-CNV can also be used to mask low quality nucleotides. Similarly that for the filtering of low quality sequences, G-CNV performs masking in three steps. The first step is performed on CPU and it is aimed at detecting the Illumina format. Conversely, the last two steps are massively parallelized on a single GPU and are aimed at decoding the quality values sequences according to the Illumina format, and at masking with a aNy symbol those nucleotides with a quality score lower than a user-defined threshold. Then a new FASTQ file is created with the masked nucleotides.

6.2.1.3 Removing adapter sequences

In the current release G-CNV uses *cutadapt* to remove adapter sequences. *Cutadapt* can be used to look for adapter sequences in reads generated with Illumina, 454 and SOLiD HTS machines. Basically, *cutadapt* is able to look for multiple adapters in the 5' and 3' ends according to different constraints (e.g., mismatches, indels, minimum overlap between the read and adapter). It can be used to trim or discard reads in which an adapter occurs. Moreover, it allows to automatically discard those reads that after the trimming are shorter than a given user defined length. All features of *cutadapt* were wrapped in G-CNV.

It should be pointed-out that the current release of *cutadapt* is not parallelized. In order to speed up the removing of the adapters G-CNV splits the original FASTQ files in chunks and runs in parallel an instance of *cutadapt* on each of these chunks. Finally, the output files provided by each instance of *cutadapt* are merged together in a new FASTQ file.

6.2.1.4 Removing Duplicated Read Sequences

Duplicate reads are one of the most problematic artifacts. These artifacts are generated during the PCR amplification. Ideally, duplicates should have identical nucleotide sequences. However, due to the sequencing errors they could be nearly identical [59]. Alignment-based (e.g., NGS QC Toolkit, SEAL [161], and Picard MarkDuplicates) and alignment-free (e.g., FastUniq [196], Fulcrum [29]), CD-HIT [114, 53]) methods have been proposed in the literature to remove duplicated read sequences. Basically, alignment-based methods start from the assumption that duplicated reads will be mapped into a refence genome in the same position. Therefore, in these methods read sequences are aligned against a reference genome and those reads with identical alignment positions are classified as duplicates. It should be pointed out that the final result is affected by both the alignment constraints and the accuracy of the aligner. In alignment-free methods, read sequences are compared among them according to a similarity measure. The reads with a similarity score lower than a given threshold are classified as duplicated.

G-CNV implements an alignment-free method to remove duplicated read sequences from single-end libraries. Like other tools, it implements a prefix-suffix comparison approach. The algorithm has been devised taking into account the per-base error rates of Illumina platforms. Analysis of short read datasets obtained with Illumina highlighted a very low rate of indel errors ($< 0.01\%$) while the number of occurrences of wrong bases increases with the base position [42]. Therefore, G-CNV does not take into account indels and considers as potentially duplicated read sequences those with an identical prefix. Potential duplicated sequences are clustered together (see Figure 6.3), and for each cluster G-CNV compares the suffixes of its sequences. The first sequence of a cluster is taken as a seed and its suffix is compared with those of the other sequences in that cluster. Those sequences identical or very similar to the seed are considered duplicated. Duplicated sequences will be condensed in a new sequence and will be removed from the cluster (see Figure 6.4). Then, the process is iterated for the remaining sequences in the cluster (if any), until the cluster is empty or contains only a read sequence.

In G-CNV, clustering is performed sorting the prefixes of the read sequences. Sort-

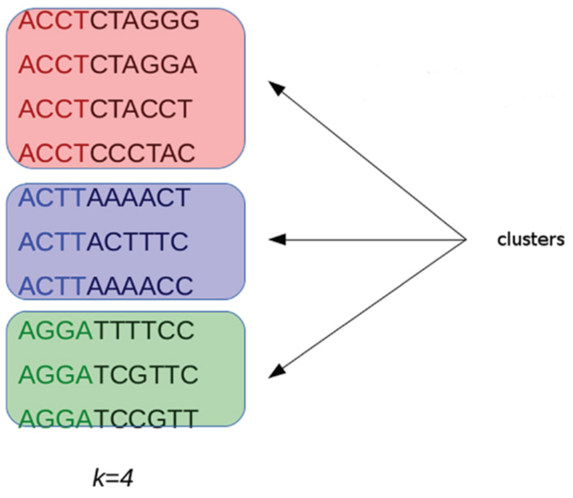


Figure 6.3: **Cluster of identical-prefix short reads**. Short reads with an identical prefix (of fixed length k) are clustered together as potential duplicated sequences. This approach takes into account ther error rates of Illumina platforms. Analysis performed on short reads generated with these sequencing platforms highlighted that ther number of wrong bases increases with the base position.

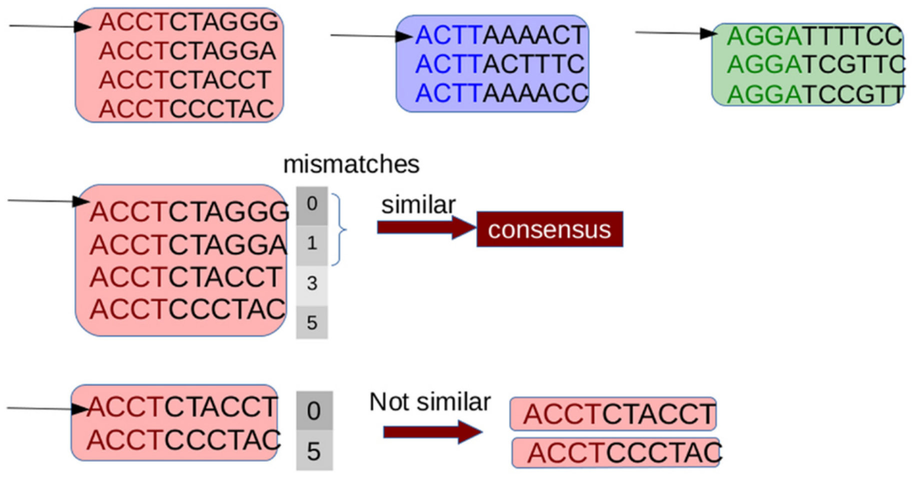


Figure 6.4: **Identification of duplicates through comparison of sequence suffixes.** Suffixes of sequences in a cluster are compared to identify the duplicates. The first read is taken as a seed and its suffix is compared with those of the other ones. Sequences with a number of mismatches lower than a given threshold are considered duplicates of the seed. These sequences are removed from the cluster and are represented with a consensus sequences. Then the process is repeated until the cluster is empty or consists of a single sequence.

ing is performed on a GPU with CUDA-Quicksort [4]. Experimental results shown that CUDA-Quicksort is faster than other available GPU-based implementations of the quicksort. In particular, it results be up to 4 times faster than GPU-Quicksort of [33] and up to 3 times faster than the NVIDIA CDP-Quicksort. As CUDA-Quicksort sorts numerical values the prefixes must necessarily be subject to a numerical encoding. The encoding was devised with the aim to maximize the length of the prefixes that can be compared. In doing this, read sequence prefixes are subject to a dual numerical encoding. Initially, the prefixes were encoded using a base-5 encoding by replacing each nucleotide with a numerical value ranging from 0 to 4 (i.e., $A \rightarrow 0$, $C \rightarrow 1$, $G \rightarrow 2$, $T \rightarrow 3$, $N \rightarrow 4$). Using CUDA-Quicksort to sort items represented with *64 bit unsigned long long int* data type, prefixes of up to 19 nucleotides can be sorted. A longer prefix will exceed the limit for this type of data. However, it is possible to exceed this constraint using a different numerical base to represent the prefixes. In particular, using the base-10 it is possible to represent a number consisting of up 27 digits with a *64 bit unsigned long long int* (see Figure 6.5). Therefore, G-CNV applies this second encoding to maximize the length of the prefixes used for clustering.

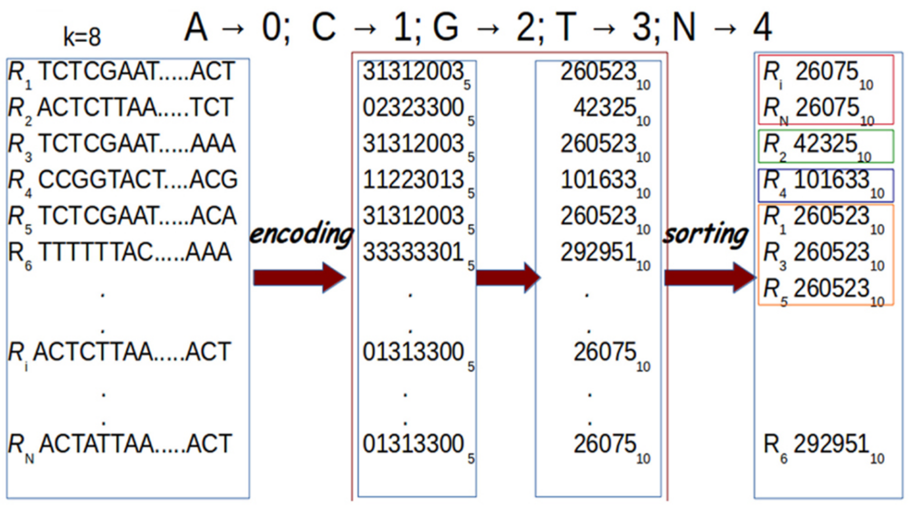


Figure 6.5: **Dual encoding prefixes**. Prefixes are subject to a dual encoding. As for the first encoding, each nucleotide in a prexix is represented with a numerical value from 0 to 4 ($A \rightarrow 0$, $C \rightarrow 1$, $G \rightarrow 2$, $T \rightarrow 3$, $N \rightarrow 4$). Then, these numerical representations are encoded using base-10. Finally sorting is performed for clustering. In the figure, prefixes of lenght *k=8* are represented.

[4]Submitted to Concurrency and Computation: Practice and Experience: manuscript CPE-14-0292 entitled "CUDA-Quicksort: An Improved GPU-based Implementation of Quicksort"

After that the reads have been clustered G-CNV compares their suffixes. This step requires a base-per-base comparison of the nucleotides of the seed read sequence with those of the other reads in a cluster. This approach can require a very high number of base-base comparisons. Let N be the length of the suffixes, and let m be the allowed number of mismatches. In the best case m comparisons must be performed to classify two sequences as not duplicated. In the worst case N-m comparisons must be performed to classify two sequences as duplicated. Apart the high number of comparisons required, this approach is not adapted to be efficiently implemented on GPUs. As GPUs adopt the SIMT paradigm each thread in a block must perform the same operation on different data. Then, G-CNV implements a different comparison method. Suffixes are split into fixed length chunks. Subsequence of each chunk is subjected to the same dual numerical encoding used to represent the prefixes for clustering. Then for each cluster, the numerical difference between the *i-th* chunk of the seed and the related chunk of the other suffixes in a cluster is calculated (see Figure 6.6). The order of magnitude of the difference provides information about the position of the leftmost different nucleotides. Then, the subsequences are cutted corresponding to the mismatch position. The rightmost parts of the mismatch position are maintained and the process is re-iterated.

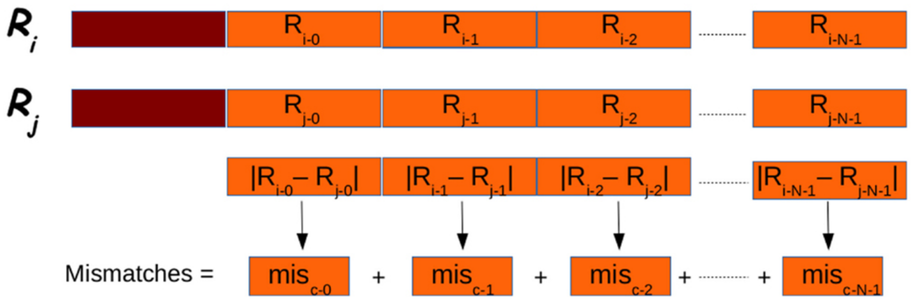

R_i
$R_{i\text{-}0}$
$R_{i\text{-}1}$
$R_{i\text{-}2}$
$R_{i\text{-}N\text{-}1}$
R_j
$R_{j\text{-}0}$
$R_{j\text{-}1}$
$R_{j\text{-}2}$
$R_{j\text{-}N\text{-}1}$
$|R_{i\text{-}0} - R_{j\text{-}0}|$
$|R_{i\text{-}1} - R_{j\text{-}1}|$
$|R_{i\text{-}2} - R_{j\text{-}2}|$
$|R_{i\text{-}N\text{-}1} - R_{j\text{-}N\text{-}1}|$
Mismatches = $mis_{c\text{-}0}$ + $mis_{c\text{-}1}$ + $mis_{c\text{-}2}$ + ... + $mis_{c\text{-}N\text{-}1}$

Figure 6.6: **Chunks of PSuffixes**. PSuffixes (in orange in the figure) are analyzed in chunks. Each chunk is subject to the dual encoding used for prefixes (in red in the figure). The overall number of mismatches if obtained summing the partial number of mismatches obtained for each chunk.

6.2.2 Mapping

It is widely known that mapping of short-read sequences is computationally onerous. Several tools have been devised to deal with short-read mappings. Without claiming to be exhaustive, some of the most popular solutions are cited, i.e. MAQ [111], RMAP [179, 178], Bowtie [104], BWA [108], CloudBurst [169], SOAP2 [113] and SHRiMP [166, 40]. A comparative study aimed at assessing the accuracy and the runtime performance of different cutting-edge next-generation sequencing read alignment tools highlighted that among all SOAP2 was the one that showed the higher accuracy [165]. Exhaustive review of the tools cited above can be found in [22].

In general, the mentioned solutions exploit some heuristics to find a good compromise between accuracy and running time. Recently, the GPU-based short-read mapping tools Barracuda [96], CUSHAW [120], SOAP3 [116] and SOAP3-dp have been successfully proposed to the scientific community. In particular, SOAP3-dp aligns the read sequences in two steps. As for the first step, it looks for ungapped alignments with up to four mismatches without using heuristics. As for the second step, it uses the dynamic programming to look for gapped alignments. Compared with BWA, Bowtie2 [103], SeqAlto [140], GEM [128], and the previously mentioned GPU-based aligners, SOAP3-dp is two to tens of times faster, while maintaining the highest sensitivity and lowest false discovery rate on Illumina reads with different lengths.

Starting from the previous analysis, it was decided to use SOAP3-dp to support read mapping in G-CNV. G-CNV allows to set different parameters of SOAP3-dp that can be useful to properly generate alignments for RD methods. Apart from the constraints on the allowed mismatches, G-CNV allows to set SOAP3-dp parameters able to filter out alignments that are not of interest for the specific RD method. In particular, as different methods presented in the literature filter alignments using different quality mapping scores, G-CNV allows to set a quality mapping threshold on the alignments that must be reported. To set these constraints, G-CNV needs to be able to access the SOAP3-dp files to change the initialization file. Moreover, a short-read may be uniquely aligned or can be aligned to multiple positions onto a genome. Multiple mappings can be related to the alignment constraints or to the nature of the sequenced read. A read sequence can be aligned to multiple positions, as it has been sequenced from repetitive regions or regions of segmental duplication [11]. In the former case alignments are characterized by different alignment scores, whereas in the latter case they are expected to have equal or very similar scores. A common approach to take into account multiple mappings is to randomly select a best alignment. G-CNV allows to report only unique best alignments or a random best alignment.

6.2.3 RD signal

The RD signal depends on the size of the observing window. As methods proposed in the literature suggest different approaches to estimate the window size, G-CNV does not impose it. In G-CNV, the window size is a parameter that must be set by the user.

G-CNV builds the RD signal in two steps. Initially, G-CNV analyzes the genome sequences to build a GC-content signal according to the fixed window size. A GC-signal for each genome sequence will be built. Then, G-CNV splits the mapping for each chromosome sequences, identifies the window where the mappings fall, and calculates a raw RD-signal. By default, the window related to each alignment is identified considering the centering of the read. Finally, G-CNV corrects the RD signal with the same approach proposed in [201] that adjust the RD by using the observed deviation of RD for a given GC percentage according to the following equation:

$$RD'_{w_i} = \frac{\overline{RD}}{\overline{RD}_{GC_{w_i}}} \cdot RD_{w_i} \tag{6.1}$$

where RD_{w_i} is the RD for the i-th window to be corrected, $\overline{RD}$ is the average RD signal, $\overline{RD}_{GC_{w_i}}$ is the average RD signal calculated on the windows with the GC-content found in the i-th window, and RD'_{w_i} is the corrected RD for the i-th window.

6.2.4 Hardware and Software Requirements

G-CNV has been designed to work with NVIDIA GPU cards based on the most recent Kepler architecture. G-CNV works on linux based systems equipped with CUDA (release $\geqslant$ 6.0). It was tested on the NVIDIA Kepler architecture based k20c card. Experiments have been carried-out using the last release of *soap3-dp* (rel. 2.3.177) and of *cutadapt* (rel. 1.7.1).

6.3 Results

Different experiments were performed, aimed at assessing the performance of G-CNV. In particular, its performance was assessed, when used to filter low quality sequences, to mask low quality nucleotides, to remove adapter sequences, to remove duplicated reads, and to calculate the RD-signal. Since G-CNV performs the alignments running *SOAP3-dp* it was deemed not relevant to assess the performance of

G-CNV in this task. Readers are invited to refer the *SOAP3-dp* manuscript for a in depth analysis of the performance of the aligner. Similarly, as G-CNV uses the well known tool *cutadapt* to remove adapter sequences, no test was performed aimed at assessing its reliability in this task. However, experiments were performed aimed at assessing the benefits of the parallelization of *cutadapt* provided with G-CNV.

Experiments have been carried-out on both synthetic and real life libraries. Synthetic reads have been used to assess and compare with other tools the reliability of G-CNV, whereas real life data to assess and compare its performance in terms of both computing time and memory consumption.

Synthetic reads have been generated from the build 37.3 of the human genome using the *Sherman* simulator (`http://www.bioinformatics.babraham.ac.uk/projects/sherman/`). *Sherman* has been devised to simulate HTS datasets for both bisulfite sequencing and standard experiments. To mimic real data it generates synthetic data using an error rate curve that follows an exponential decay model. *Sherman* was used to generate a single-end synthetic library consisting of 1 millions of 100bp reads. Library has been generated simulating a sequencing error of 2% and contaminating the reads with the Illumina singe-end adapter 1 (i.e., ACACTCTTTCCCTACACGACGCTGTTCCATCT). The contamination has been simulated with a normal distribution of fragment sizes. Moreover, since *Sherman* generates identical quality scores for all reads, they were modified to generate a 3% of low quality nucleotides (PHRED value $\leq$ 20) and a 9% of low quality sequences. In the following of this work this dataset will be referred to as the *S1* library.

Since *Sherman* does not permit to control the percentage of duplicates the simulated reads were modified in *S1* to generate a new synthetic library (*S2*) consisting of 30% of duplicated sequences. Read sequences have been duplicated simulating a sequencing error of 2%. The library *S1* has been used to assess the reliability of G-CNV in the task of filtering low quality sequences and masking low quality nucleotides, whereas *S2* has been used to assess the reliability of G-CNV in the task of removing duplicated sequences.

As for real life data experiments have been performed on different libraries generated with Illumina platforms: *i*) SRR001220 consisting of 3.3 millions of 94bp reads; *ii*) SRR001205 consisting of 9.7 millions of 47bp reads; *iii*) SRR005720 consisting of 26.2 millions of 36bp reads; and *iv*) SRR921889 consisting of 50 millions of 100bp reads (see Table 6.1).

Moreover, with the aim to simulate a CNV pre-processing detection analysis two high coverage (30x) whole genome sequencing experiments were simulated. The first experiment have been simulated generating 37 synthetic libraries consisting of 25

Dataset	Library layout	Reads	Read size	Organism	Instrument
SRR001220	Single	3.3M	94bp	Homo Sapiens	Illumina Genome Analyzer II
SRR001205	Single	9.7M	47bp	Homo Sapiens	Illumina Genome Analyzer II
SRR005720	Paired	26.2M	36bp	Homo Sapiens	Illumina Genome Analyzer
SRR921889	Single	50.0M	100bp	Mus Musculus	Illumina HiSeq 2000

The first column reports the name of the dataset. The second column reports the library layout. The third and fourth column report the size of the dataset and the length of the reads, respectively. Organism and sequencing instrument are reported in column fifth and sixth.

Table 6.1: **Real life datasets**.

millions of 100bp reads, and the second generating 9 synthetic libraries consisting of 100 millions of 100bp reads. All libraries have been generated according to the same constraints used to generate *S1*. In the following of this work these datasets will be referred to as *HCS1* and *HCS2*.

In the following of this section the different experiments and present results were described. Finally, the hardware and software configuration used for experiments is briefly resumed.

6.3.1 Filtering low quality sequences

To assess G-CNV in the task of filtering low quality read sequences its performance with those of FASTX-Toolkit and NGS QC Toolkit was compared. Experiments have been performed setting parameters with the aim to filter those sequences with a percentage of low quality (PHRED score < 20) bases $> 10\%$ (see Table 6.2).

A first experiment has been performed on the *S1* synthetic library aimed at assessing and comparing the reliability of G-CNV with the other tools. As expected all tools have been able to filter all low quality sequences. The same experiment has been performed on the real life libraries aimed at assessing the performance of G-CNV in terms of both computing time and memory consumption. It should be pointed out that FASTX-Toolkit does not support parallelization whereas in NGS QC Toolkit parallelization has been implemented in multiprocessing and multithreaded ways. Multiprocessing parallelization was implemented to process multiple files in parallel whereas multithreading paralellization to process in parallel a single file. The FASTQ file is split into chunks, processed in parallel and results are merged at the end. With the aim to provide an in-depth comparison among all tools and to assess as NGS QC Toolkit can scale increasing the CPU cores, the experiments were

tool	commands
GCNV	–mf 20 –pf 90
FASTX-Toolkit	-Q33 -q 20 -p 90
NGS QC Toolkit[1]	N A -l 90 -s 20
NGS QC Toolkit[2]	N A -l 90 -s 20 -c 12

Both FASTX-Toolkit and NGS QC Toolkit consist of different commands. As for FASTX-Toolkit experiments have been performed using the *fastq_quality_filter* command, whereas the *IlluQC_PRLL* has been used for NGS QC Toolkit. The table reports the settings used to run NGS QC Toolkit without exploiting parallelization (NGS QC Toolkit[1]) and parallelized on 12 CPU cores (NGS QC Toolkit[2]).

Table 6.2: **Tools settings used to filter low quality sequences.**

initially run without using parallelization, then experiments have been performed parallelizing the computation on 12 CPU cores.

It should be pointed out that FASTX-Toolkit does not provide support for paired-end libraries. Therefore, it has not been possible to test it with the *SRR005720* dataset. Experimental results show that G-CNV is most effective than the other tools in terms of computing time. Table 6.3 reports computing time and peak of memory required by G-CNV, FASTX-Toolkit, and NGS QC Toolkit to analyze the differet datasets. G-CNV has been 12.4x/7.8x/NA/21.4x faster than FASTX-Toolkit and 24x/21x/26.5x/28.3x faster than NGS QC Toolkit parallelized on 12 CPU cores to filter the read sequences of the SRR001220/SRR001205/SRR005720/SRR921889 dataset. Obviously, the performance of G-CNV improves notably when compared with those of NGS QC Toolkit executed without parallelization. In this case G-CNV has been 154x/120x/125x/175x faster than NGS QC Toolkit to analyze the SRR001220/SRR001205/SRR005720/SRR921889 dataset.

For the sake of completeness, it should be pointed out that NGS QC Toolkit automatically also generates statistics for quality check. Therefore, the computing time reported from NGS QC Toolkit takes into account also the time required to perform these operations.

As for the memory consuption FASTX-Toolkit is undoubtedly the most effective tool. Conversely, G-CNV requires more memory than the other tools. Its performance are only comparable with those of NGS QC Toolkit executed in parallel for the *SRR001220* and *SRR001205* datasets. Experimental results show that the memory required by G-CNV increases with the size of the analyzed library. This is mainly due to the fact that to massively parallelize the computation G-CNV loads

tool	dataset	filtered seq.	time	memory
GCNV	SRR001220	95.3%	5s	0.9GB
	SRR001205	98.3%	11s	1.4GB
	SRR005720	74.7%	48s	4.5GB
	SRR921889	7.9%	1m 10s	10.5GB
FASTX-Toolkit	SRR001220	95.3%	1m 2s	256KB
	SRR001205	98.3%	1m 19s	256KB
	SRR005720	-	-	-
	SRR921889	7.9%	17m 10s	256KB
NGS QC Toolkit[1]	SRR001220	95.3%	12m 52s	0.21GB
	SRR001205	98.3%	22m	0.18GB
	SRR005720	74.7%	1h 40m	0.26GB
	SRR921889	7.9%	3h 25m	0.22GB
NGS QC Toolkit[2]	SRR001220	95.3%	2m	1.4GB
	SRR001205	98.3%	3m 52s	1.4GB
	SRR005720	74.7%	21m	1.3GB
	SRR921889	7.9%	33m	1.9GB

The first and the second column of the table report the tool and the analyzed library, respectively. The third column the percentage of filtered reads. Column fourth reports the computing time required to analyze the different libraries. The fifth column the peak of memory required to perform the analysis.

Table 6.3: **Performance of G-CNV to filter low quality sequences.**

into the memory as many as possible read sequences to maximize the occupancy of the grid of the GPU.

Finally, G-CNV was used to filter the low quality sequences of the *HCS1* and *HCS2* datasets. Filtering has been performed in ~ 20 minutes for the *HCS1* and in ~ 34 minutes for *HCS2*. As for the memory consumption, G-CNV required 5.7GB to analyze *HCS1* and 20.5GB for *HCS2*.

6.3.2 Masking low quality nucleotides

The performance of G-CNV in the task of masking low quality nucleotides have only been compared with those of FASTX-Toolkit. NGS QC Toolkit does not provide support for this operator. G-CNV and FASTX-Toolkit have been run to mask with aNy symbol the nucleotides with a PHRED quality score < 20 (see Table 6.4). Experiments performed on the *S1* synthetic library shown that both tools have been able to mask all low quality sequences. Experiments performed on real life libraries show that G-CNV outperforms notably FASTX-Toolkit in terms of computing time. Results reported in Table 6.5 show that G-CNV has been 12x/6.8x/5x/13.8x faster than FASTX-Toolkit to analyze the SRR001220/SRR001205/SRR005720/SRR921889 dataset. As previously highlighted FASTX-Toolkit does not support paired-end reads. However, as for the task of masking low quality nucleotides it can be separately used on both the forward and the reverse read sequences. Then, as for the *SRR005720* dataset the Table 6.3 reports the overall computing time required by FASTX-Toolkit to analyze both files.

tool	
GCNV	-m 20
FASTX-Toolkit	-Q33 -q 20 -r N

As for FASTX-Toolkit experiments have been performed using the *fastq_quality_masker* command.

Table 6.4: **Tools settings used to mask low quality nucleotides.**

As for the high coverave simulated sequencing experiments G-CNV masked the low quality nucleotides of *HCS1* in ~ 23 minutes using 7GB of memory, whereas required ~ 39 minutes and 21.9GB of memory for *HCS2*.

tool	dataset	masked nucl.	time	memory
GCNV	SRR001220	24.2%	5s	0.94GB
	SRR001205	43.6%	10s	1.38GB
	SRR005720	21.8%	52s	3.88GB
	SRR921889	3%	1m 15s	12GB
FASTX-Toolkit	SRR001220	24.2%	1m	256KB
	SRR001205	43.6%	1m 8s	256KB
	SRR005720	21.8%	4m 22s	256KB
	SRR921889	3%	17m 20s	256KB

The first and the second column of the table report the tool and the analyzed library, respectively. The third column the percentage of masked nucleotides. Column fourth reports the computing time required to analyze the different libraries. The fifth column the peak of memory required to perform the analysis.

Table 6.5: **Performance of G-CNV to mask low quality nucleotides.**

6.3.3 Removing adapter sequences

As for the task of removing adapter sequences G-CNV has been compared with both FASTX-Toolkit and NGS QC Toolkit. To assess the advantages of the implemented parallelization of *cutadapt* experiments were initially performed running G-CNV without exploiting the parallelization, subsequently parallelizing the computation on 12 CPU cores. Tool settings used to perform these experiments are reported in Table 6.6.

tool	
GCNV^1	–ca-a ACACTCTTTCCCTACACGACGCTGTTCCATCT
GCNV^2	–ca-a ACACTCTTTCCCTACACGACGCTGTTCCATCT –ca-t 12
FASTX-Toolkit	-Q33 -a ACACTCTTTCCCTACACGACGCTGTTCCATCT
NGS QC Toolkit^1	<<ADAPTER FILE>> A
NGS QC Toolkit^2	<<ADAPTER FILE>> A -c 12

As fo FASTX-Toolkit experiments have been performed using the *fastx_clipper* command, whereas the *IlluQC_PRLL* has been used for NGS QC Toolkit. In the table have been reported the settings used to run G-CNV^1 and NGS QC Toolkit^1 without exploiting the parallelization and in multi-threading way (G-CNV^2 and NGS QC Toolkit^2). The table shows the settings used to remove the Illumina Single End Adapter 1. As for the *SRR005720* dataset settings have been modified to remove the Illumina paired-end adapters.

Table 6.6: **Tools settings used to remove adapter sequences.**

Table 6.7 reports results obtained analyzing the real life libraries. Results show that the performance of G-CNV improves notably with parallelization. With parallelization G-CNV has been 6.7x/6.4x/23.4x/2.8x faster to remove the adapter sequences from the SRR001220/SRR001205/SRR005720/SRR921889 dataset. Moreover G-CNV parallelized on 12 CPU cores resulted be 18.2x/11x/-/9.4x faster than FASTX-Toolkit and 11.8x/7.3x/58.3x/6.3x NGS QC Toolkit used exploiting the parallelization to remove the adapters from the SRR001220/SRR001205/SRR005720/SRR921889 dataset. Obviously, also for this task the performance of G-CNV improves when compared with NGS QC Toolkit used without parallelization. In this case G-CNV resulted be 48x/40x/173x/38x faster than NGS QC Toolkit to analyze the SRR001220/SRR001205/SRR005720/SRR921889 dataset. As for the memory consumption FASTX-Toolkit provides better performance than the other tools. However, G-CNV outperforms NGS QC Toolkit. As FASTX-Toolkit does not support paired-end libraries it has not been used to analyze the *SRR005720* dataset.

Finally, when used to remove adapters from the *HCS1* G-CNV required $\sim 50m$ and 250MB of memory, whereas it required $\sim 3h20m$ and 920MB for *HCS2*.

tool	dataset	time	memory
GCNV[1]	SRR001220	1m 14s	17MB
	SRR001205	2m 46s	21MB
	SRR005720	8m 12s	26MB
	SRR921889	17m 11s	20MB
GCNV[2]	SRR001220	11s	0.4GB
	SRR001205	26s	0.46GB
	SRR005720	21s	0.33GB
	SRR921889	6m 10s	0.84GB
FASTX-Toolkit	SRR001220	3m 21s	516KB
	SRR001205	4m 47s	516KB
	SRR005720	-	-
	SRR921889	57m 40s	516KB
NGS QC Toolkit[1]	SRR001220	8m 52s	217MB
	SRR001205	17m 30s	189MB
	SRR005720	1h 48m	269MB
	SRR921889	3h 55m	226MB
NGS QC Toolkit[2]	SRR001220	2m 10s	1.6GB
	SRR001205	3m 10s	1.3GB
	SRR005720	20m 24s	1.13GB
	SRR921889	39m	1.6GB

The first and the second column of the table report the tool and the analyzed library, respectively. Column third reports the computing time required to analyze the different libraries. The fourth column the peak of memory required to perform the analysis.

Table 6.7: **Performance of G-CNV to remove adapter sequences.**

6.3.4 Removing Duplicated Read Sequences

To assess the performance of G-CNV in the task of removing duplicated sequences, its performance were compared with those of Fulcrum. G-CNV implements a very similar algorithm to that implemented in Fulcrum. In particular, similarly to the proposed tool Fulcrum clusters together the reads with a similar prefix and looks for duplicates in the same cluster.

tool	
GCNV	-D << *mis* >> -p << *pref* >>
Fulcrum	-b << *pref* >> -s -t s -c << *mis* >>

Different experiments were performed with different values for both the prefix length and the allowed mismatches. Specific values for the prefixes << *pref* >> and the allowed mismatches << *mis* >> are reported in the tables of the results.

Table 6.8: **Tools settings used to remove duplicated sequences.**

Table 6.8 reports the main parameters that have been used for the experiments. Experiments on the synthetic *S2* library have been performed clustering reads according to a prefix length of 25 bp and looking for identical sequences (i.e., 0 mismatches) and nearly identical sequences with up to 1 mismatch. Results reported in Table 6.9 show that both tools have been able to identify the synthetic duplicate sequences. It should be pointed out that *S2* has been built avoiding to generate mismatches among the duplicated sequences in their first 25bp. As for tests on real life data, experiments were performed on the larger *SRR921889* dataset. Experiments have been aimed at assessing the performance of G-CNV to remove duplicated sequences according to different constraints. In particular, the experiments were performed on both G-CNV and Fulcrum to cluster sequences according to a prefix size of 10 and 25bp and to look for duplicated sequences with up to 1 and up to 3 mismatches. Experimental results are reported in Table 6.10. For each experiment were reported the percentage of removed sequences, the computing time and the peak of memory required for the analysis. Results show that both tools remove a similar percentage of duplicated sequences. However, as for the computing time G-CNV outperforms Fulcrum in all experiments. It should be pointed out that Fulcrum automatically parallelize the computation on all available CPU cores. Therefore, the computing times reported in the table have been obtained running Fulcrum parallelized on 12 CPU cores. Results show that the computing time required by G-CNV depends on both the number of allowed mismatches and the prefix size. The number of sequences that will be classified as duplicated increases with the number of allowed mismatches. Therefore, increasing this value may involves a lower number of sequences comparison. Moreover, the size of a cluster depends on the prefix length. Typically, the size of the clusters increases as the prefix length decreases involving more sequences comparison.

tool	dataset	mismatches	perc. of removed
GCNV	S2	0	0%
	S3	1	30.1%
Fulcrum	S2	0	0%
	S3	1	30.6%

The first column reports the tool. The second column reports the dataset. Column third and forth report the allowed mismatches and the percentage of removed duplicated sequences.

Table 6.9: **Performance of G-CNV to remove duplicated sequences from the synthetic dataset.**

tool	dataset	prefix	mismatches	perc. of removed	time	memory
GCNV	SRR921889	10	1	11.2%	2h	17.3GB
	SRR921889	10	3	11.5%	1h 50m	17.3GB
	SRR921889	25	1	11.9%	16m	17.3GB
	SRR921889	25	3	12.1%	8m	17.3GB
Fulcrum	SRR921889	10	1	11.3%	4h 01m	1.6GB
	SRR921889	10	3	11.4 %	3h 23m	1.6GB
	SRR921889	25	1	11.6%	1h 24m	1.6GB
	SRR921889	25	3	11.9%	1h 33m	1.6GB

The first column reports the tool. The second column reports the length of the prefixes used for clustering. Column third reports the allowed mismatches. The fourth column reports percentage of removed sequences. Column fifth and sixth report the computing time and the memory consumption, respectively.

Table 6.10: **Performance of G-CNV to remove duplicated sequences from the real life dataset.**

For the sake of completeness G-CNV performed the clustering step in ~ 2 seconds for both length of the prefixes, whereas Fuclrum required 13 minutes to cluster the reads according to a prefix of 10bp and 56 minutes to cluster the reads according to prefix length of 25bp. However, it should be pointed out that G-CNV can not be used to cluster reads with a prefix longer than 27bp. Moreover, the clustering phase implemented by G-CNV requires that all prefixes will be loaded into the memory of the GPU device. This implies a constraint on the size of the analyzed library, which depends on the memory of the GPU. As for the memory consumption G-CNV undoubtedly requires more memory than Fulcrum. Also in this case the high memory consumption is due to the need of maximize the occupancy of the grid of the GPU.

Finally, different experiments were performed on the *HCS1* dataset. Experiments have been performed to cluster the reads according to a prefix length of 15bp and 27bp and to look for duplicated with up to 1 and to 3 mismatches. Results are reported in Table 6.11.

mismatches	prefix	time	memory
	15	12h 7m	8.8GB
1	27	5h 33m	6.6GB
	15	3h 20m	8.7GB
3	27	1h 30m	5.7GB

The first column reports the allowed mismatches. The second column reports the length of the prefix used to cluster the reads. Column third and fourth report the computing time and memory consumption, respectively.

Table 6.11: **Performance of G-CNV to remove duplicate sequences from the HCS1 synthetic dataset.**

6.3.5 Generating the RD-Signal

As do not exist other specialized tools to generate the RD signal the performance of G-CNV cannot be assessed and compared with other tools. However, the *FastQC* tool was used (http://www.bioinformatics.babraham.ac.uk/projects/fastqc/) to assess the reliability of G-CNV in the task of calculating the GC-content that is used to normalize the RD signal. FastQC is a tool that provides some quality control checks on HTS data. In particular, it is able to calculate the distribution of the per sequence GC content of the analyzed read sequences.

As G-CNV calculates the GC content of each observed window in the genome sequences, a synthetic library was generated using as reads the subsequences observed with a window of 100bp along the MT chromosome of the human genome (build 37.3). Then, FastQC was used to analyze the GC content of these sequences and the results were compared with those generated by G-CNV. Both tools provided the same distribution of the GC-content. It should be pointed out that it was not possibile to comapare the results with those of FASTX-Toolkit and NGS QC Toolkit as both determine only the perbase GC content. It should be pointed out that the time required by G-CNV was not compared with that required by *FastQC* as it automatically performs several quality checks.

Moreover, to assess the performance of G-CNV to generate a RD-signal an alignment SAM file on the human genome was simulated (build 37.3). The alignment has been simulated by assuming a sequencing experiment on the genome with coverage 30x. Sequencing and alignment errors were not simulated. The SAM file was generated by assuming an ideal aligner able to map the reads uniquely and without errors. In fact, these errors do not affect the computing time to generate the RD-signal; they affect the detection of CNVs. However, for this experiment the main aim was to assess the computing time of G-CNV in the task of generating the RD-signal. G-CNV generated the RD-signal with an observing window of length 100 in less than 1h 56m minutes and required 10.4GB of memory. As for the memory used by G-CNV it depends on the number of alignments in the analyzed genome sequence. G-CNV generates the RD-signal analyzing separately the genome sequences. To maximize the parallelization as many as possible alignments on the analyzed genome sequence are loaded into the GPU.

6.3.6 Hardware and Software Configuration

Experiments described hereinafter have been carried out on a 12 cores Intel Xeon CPU E5-2667 2.90 GHz with 128 GB of RAM and an NVIDIA Kepler architecture based Tesla k20c card with 0.71 GHz clock rate and equipped with 4.8 GB of global memory.

6.4 Discussion and Conclusions

Different RD-based methods and tools have been proposed in the literature to identify CNVs. Typically, these tools do not support most of the preparatory operations for RD analysis. Therefore, a specific analysis pipeline must be built with different third-party tools. G-CNV allows to build the analysis pipeline required to process short-read libraries for RD analysis according to different constraints. However, the added value of G-CNV is the fact that almost all operations are performed on GPUs. In fact, these are data-intensive operations that may require an enormous computing power. GPUs are increasingly used to deal with computational intensive problems. The low cost for accessing the technology and their very high computing power is facilitating the GPUs success. Experimental results show that G-CNV is able to efficiently run the supported operations. However, it should be pointed out, that the current release of G-CNV still has some limitations and/or constraints. In particular, as for removing duplicates, there are two main limitations of the proposed algorithm. As for the former, the current release of G-CNV supports removal of duplicates only for single-end reads. As for the latter, it exists a constraint on the clustering phase. Sorting requires that all prefixes will be loaded into the memory of the GPU device. This implies a constraint on the size of the analyzed library, which depends on the memory of the GPU. With a GPU card equipped with 4.8 GB of global memory, libraries of up to 220M reads can be analyzed. A solution to overcome this constraint is to parallelize the sorting on multiple GPU devices. It is planned to adapt CUDA-Quicksort to run on multiple GPUs. Although, CUDA-Quicksort resulted be the fastest GPU-based implementation of the quicksort algorithm, the Thrust Radix Sort is currently the fastest GPU-based sorting algorithm. However, as for clustering, CUDA-Quicksort was adapted and used as it has been designed to be easily modified to scale on multiple GPUs. Moreover, it is deemed that the overall performance of G-CNV can be improved by implementing the trimming of the adapters on GPUs.

Chapter 7

DGCI: A Distributed GPU- and CPU-based Infrastructure

Science gateways provide an interface between scientists and Distributed Computing Infrastructures (DCIs). These gateways provide a workflow-oriented graphical user interface to create and run complex applications as scientific workflows on DCIs. At the state of the art, no workflow-oriented software infrastructure exists able to efficiently manage and schedule GPU applications on distributed GPU-based infrastructures. As GPUs are always more frequently used by modern bioinformatics applications, a Distributed GPU- and CPU-based Infrastructure (DGCI) system has been designed through a BOINC Grid desktop interfaced to the gUSE science gateway. DGCI allows submission and management of workflows on a wide variety of different DCIs. In particular, it allows the development and use of GPU bioinformatics applications on distributed GPU-based infrastructures, providing an efficient scheduling and distribution of several jobs on more GPUs.

The BOINC (Berkeley Open Infrastructure for Network Computing) Grid desktop is a Grid platform of Volunteer computing, where each volunteer desktop supplies computing resources to the platform. BOINC is made of two elements: i) one server managing job requests and sending and ii) a set of clients (Grid computing units) performing jobs. This platform offers the advantage of being capable of managing and scheduling GPU applications.

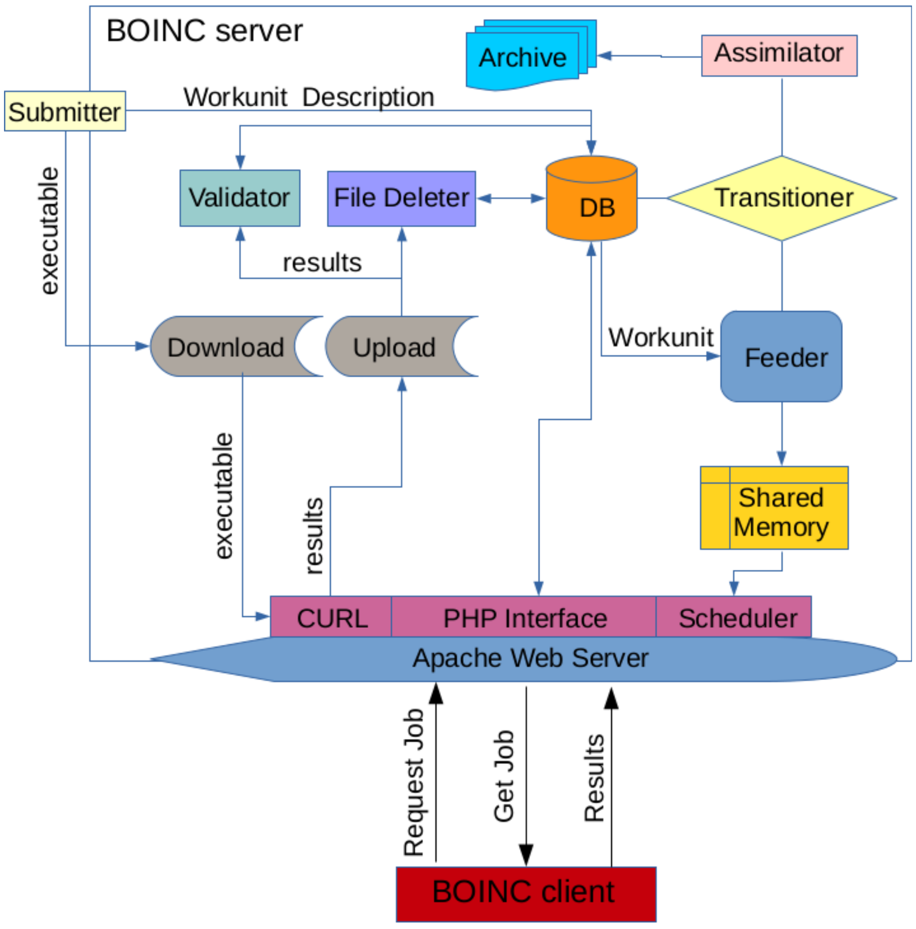
The gUSE (Grid and cloud User Support Environment) science gateway is a portal giving standard access to distributed computing infrastructures as Grid and Cloud. gUSE interfaces with them through a specific application communicating directly with the middleware (managing hardware and software resources) of DCIs. This allows users to easily send jobs and complex applications such as scientific workflows in the chosen computing infrastructure.

The gUSE science gateway interfaced to BOINC operates as follows: through gUSE, users can either create a new workflow or use a workflow already existing in the platform repository. Each job related to a workflow node is then sent to the DCI assigned to the node. In particular, GPU jobs are sent to the clients through a BOINC server based on a specific scheduling policy. Moreover, this DGCI system allows for a further coarse-grained data-parallel parallelization on more GPUs. Indeed, a job generator has been developed, able to split the input of each GPU application in multiple Workunits that will then be sent to the clients for processing. The processing outputs of each Workunit will then be merged by a job collector returning a single output to the user.

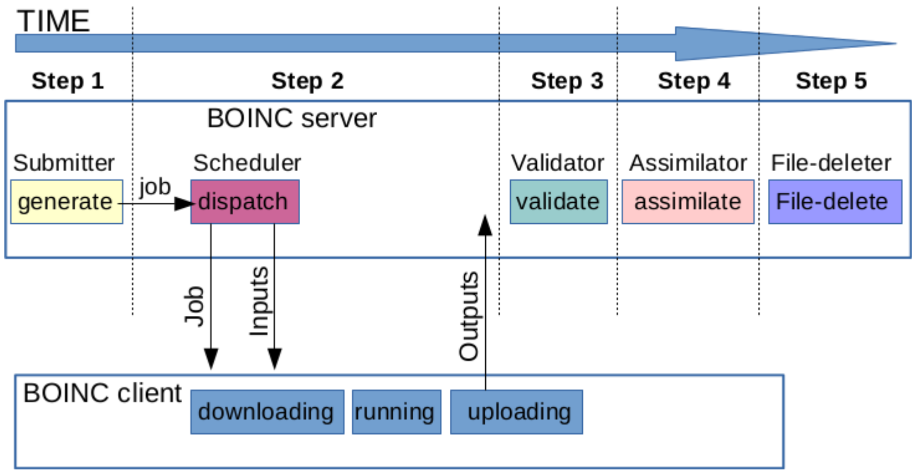
This chapter gives an overview of the gUSE science gateway and the BOINC Desktop Grid, highlighting some important aspects of each. It also illustrates the DGCI system implemented through gUSE and BOINC. Finally, it describes how the G-SNPM, GPU-BSM and G-CNV tools have been ported to DGCI (see Chapters 3, 4 and 6).

7.1 BOINC Desktop Grid

BOINC is an open source client-server middleware system created to allow projects with large computational requirements, usually set in the scientific domain, to use a technically unlimited number of volunteer machines distributed over large physical distances [19]. Created in 2002, BOINC has become one of the most popular volunteer computing middleware systems. The success of BOINC can be attributed to its simplicity and ease of use, as well as to its architecture in general. BOINC in based on a basic client-server model. The BOINC server is mainly used to host scientific projects. Each project on the server has its own applications, database and website and is not affected by the status of other projects. The BOINC client is an application installed on the volunteer host aiming at communicating with the server, mapping the client to one or more projects, organizing computation, executing applications and returning results to the server.



(a) BOINC server daemons architecture



(b) Executions of the demons in the time

Figure 7.1: **BOINC Architecture**. Image derived from [44].

7.1.1 BOINC Architecture

As shown in Figure 7.1, the BOINC architecture is based on a basic client-server model. In summary, the overall flow of BOINC is as follows: i) a BOINC client periodically sends a request to one of its attached projects. The request message specifies the client's platform; ii) the BOINC server uses the project's scheduler to scan the project's database for jobs that can be handled by the client, and returns one or more jobs; iii) the client downloads the files associated with the job(s) and executes it(them); iv) when the jobs are completed, the client uploads the resulting output files to the server. This cycle is repeated indefinitely.

The BOINC client is an application installed on a volunteer host. Upon running the BOINC client for the first time, a series of benchmarks are executed to determine the actual computing throughput capacity of the host. The total resource capacities and available disk space are also recorded. Once connected to a scientific project, the BOINC client will receive an application from the BOINC server for execution. The application itself typically consists of an executable file, which has been previously compiled on the target host platform by the server, and a series of input and output files. During the execution of an application, the BOINC client records the amount of work performed by the volunteer host and issues credits to the user which are published on a BOINC server-managed Website. Credits are calculated by multiplying the application's CPU and GPU time by benchmark scores.

The BOINC client is a simple application. Much of the system complexity resides on the BOINC server. It is used to host scientific projects, create and distribute Workunits, and store and validate results from more than one client. Workunits are instances of a particular application (i.e. a particular scientific task). Storing and distribution of these Workunits and their results are performed at server level: MySQL is used for data storage, while Apache and PHP are used for web access issues - e.g., to allow a volunteer user to modify project preferences or a project administrator to set up the project.

The BOINC server is underpinned by a set of running daemons that create and coordinate project-related items [20] (see Figure 7.1(a)). A set of default daemons are provided. However, additional daemons can be added dependent on the project characteristics and on the functionality required. Once an application developer has created his scientific project, the *Submitter* first creates one o more project Workunits and then stores i) the Workunits description in the database and ii) the executable file and the input files in the *Download* folder (see Figure 7.1(a) and step 1 in Figure 7.1(b)). The *Transitioner* handles the Workunit state transitions. The communication between client and server is implemented through a shared memory. There is a separate shared-memory structure for each running application. This communication is performed by *Feeder* and *Scheduler* daemons. The *Feeder* periodically extracts outbound Workunits from the database and enters them into a shared memory segment. The *Scheduler*, which communicates with the client through XML messages, coordinates outbound Workunits going from the shared memory segment to clients while concurrently dealing with completed Workunits. The *Scheduler* then dispatches the Workunits to different clients (see step 2 in 7.1(b)) by sending the application executable and the input files. Received results are stored in the Upload folder and the *Transitioner* is notified (see Figure 7.1(a). The *Validator* is then instructed to validate these results (see step 3 in 7.1(b)). In order to do that, the server replicates each job, which is then executed on multiple hosts. By comparing the results obtained in different clients, the server checks that no host error or security breach has influenced the results. Credits are issued to hosts only if results are deemed as valid [20]. Once a result has been validated, a Canonical Result (the simplest and best of validated results) is created. Optionally, the *Assimilator* may perform an administrator-defined action such as archiving Canonical Results in a long-term storage (see Figure 7.1(a) and step 4 in Figure 7.1(b)). In order to reduce storage space consumption on the server, the *File Deleter* removes Workunit data files and their results that are no longer required (see Figure 7.1(a) and step 5 in Figure 7.1(b)).

7.2 Grid User Support Environment

gUSE is an open-source science gateway framework that enables users to easy access Grid and Cloud infrastructures [21]. gUSE has been developed by the Laboratory of Parallel and Distributed Systems (LPDS) at the Institute for Computer Science and Control (SZTAKI) of the Hungarian Academy of Sciences. As any science gateways, gUSE provides an interface between a scientist (or community) and distributed computing infrastructures (DCIs) such as supercomputers, clusters, Grids, desktop Grids and Clouds [87]. These infrastructures are accessed by gUSE through a transparent and web-based interface, that provides a general-purpose workflow-oriented framework to compose and run workflows on various DCIs.

High-level representation of the gUSE system is provided by the WS-PGRADE (Web Service – Parallel Grid Run-time and Application Development Environment) Web application, a web portal hosted in a standard portal framework. WS-PGRADE Portal introduces many advanced features both at workflow and architecture levels. It supports simple development and fast submission of distributed applications executed on a large variety of different DCIs. WS-PGRADE provides a powerful workflow editor to compose scientific applications into data-flow based workflow structures. Users select the execution DCI for each workflow node. The workflow is then transparently submitted to the user by the gUSE services.

The gUSE/WS-PGRADE framework can be easily adapted and customized according to the special needs of various user communities, which can develop their application-specific science gateways. The reasons why scientific user communities choose this portal are its very flexible workflow system, user-friendly interface and its capability of submitting and managing workflows on a wide variety of different DCIs.

7.2.1 gUSE Architecture

The gUSE Architecture is developed in a three-tier structure. The main goal of designing the multi-tier architecture of gUSE was to enable adaptable access to many different kinds of DCIs and data storage by different kinds of user interfaces (see Figure 7.2).

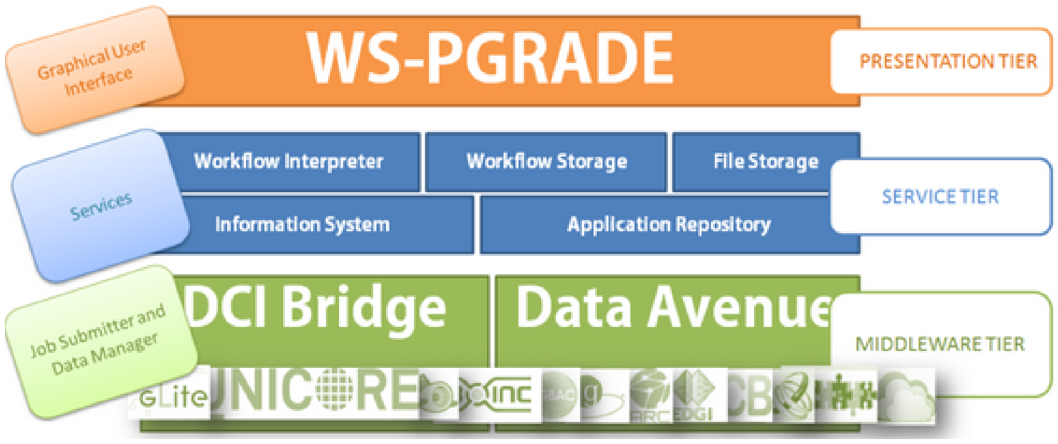


Figure 7.2: **gUSE Architecture.** Image from `guse.hu`.

At the top of the three-tier structure, a *Presentation tier* is provided by the WS-PGRADE Web application. WS-PGRADE is a Liferay-based Web portal that offers a graphical user interface for underlying gUSE services (see *Presentation tier* in Figure 7.2). All the functionalities of the gUSE services are exposed to the users by portlets residing in a Liferay portlet container. This layer can be easily customized and extended according to the needs of the science gateway instances to be derived from gUSE. End users can access WS-PGRADE via Web browser. The HTML pages of WS-PGRADE provide interfaces to generate Grid/Web service applications in four stages. In the first stage a editor is used to create the *Graph* (or *Abstract Workflow*) that contains information only on the graph structure of the workflow. In the second stage a HTML page is used to create the *Workflow* (or *Concrete Workflow*) that contains information both on the graph structure and on the configuration parameters (input file pointers, output file pointers, executable code and target DCI of workflow nodes). In the third stage a HTML page is used to create the *Template*, a workflow containing information on every modifiable parameter of the workflow and indication of whether they are user-settable. They play an important role in the automatic generation of executable workflows in the end-user mode of a WS-PGRADE/gUSE gateway. Finally, in the fourth stage a HTML page is used to create the *Application*, a ready-to-use workflow containing all the *Templates* and embedded workflows.

The middle tier of the gUSE architecture contains the high-level gUSE services (see Service tier in Figure 7.2). The *Workflow Storage* service stores a workflow except for its input files. The local input files and the local output files created during the workflow execution are stored in the *File Storage* service. The *Workflow Interpreter* service is responsible for workflow execution. The *Information System* service holds information for users about running workflows and workflow job status. Users of WS-PGRADE gateways work in isolated workspaces (i.e., they see only their own workflows). In order to enable collaboration among the isolated users, the *Application Repository* service stores the WS-PGRADE workflows in one of their possible features (i.e., *Concrete Workflow*, *Template*, *Application* and *Workflow Project*). The *Workflow Project* feature provides a collaborative workflow development among several workflow developers. Indeed, the *Workflow Project* is not yet a complete workflow and it can be further developed by the people who uploaded it into the *Application Repository* service.

The lowest architecture level, the *Middleware tier*, is provided by the *DCI Bridge* job submission service and the *Data Avenue* service, that is an independent service provided by SZTAKI (see *Middleware tier* in Figure 7.2). *DCI Bridge* is a web service-based application providing standard access to various DCIs. It connects through its DCI plugins to the external DCI resources. When a user submits a workflow, its job components are submitted transparently into the various DCI systems via the *DCI Bridge* service using its standard OGSA Basic Execution Service 1.0 (BES) interface. As a result, the access protocol and all the technical details of the various DCI systems are totally hidden behind the BES interface. The job description language of BES is the standardized Job Submission Description Language (JSDL). The *Data Avenue* service is a file commander tool for data transfer, enabling easy data moving between various storage services (such as Grid, Cloud, cluster, supercomputers) by protocols like: *HTTP*, *HTTPS*, *SFTP*, *GSIFTP*, *SRM*, and *S3*. The *Data Avenue* interface allows you browsing, downloading and uploading data to and from the supported data stores, and moving data easily between them, even if they are accessed by different protocols.

7.3 DGCI implemented through gUSE and BOINC

The Distributed GPU- and CPU-based Infrastructure system has been implemented through a BOINC Grid desktop and gUSE. BOINC offers the advantage of being capable of managing and scheduling GPU applications. gUSE provides an interface between the science community and the distributed GPU-based infrastructure, allowing the development and submission of GPU-based applications. Moreover, as gUSE supports different DCIs, workflows can be created and run on more DCIs thus allowing the integration of GPU-based applications in a workflow node.

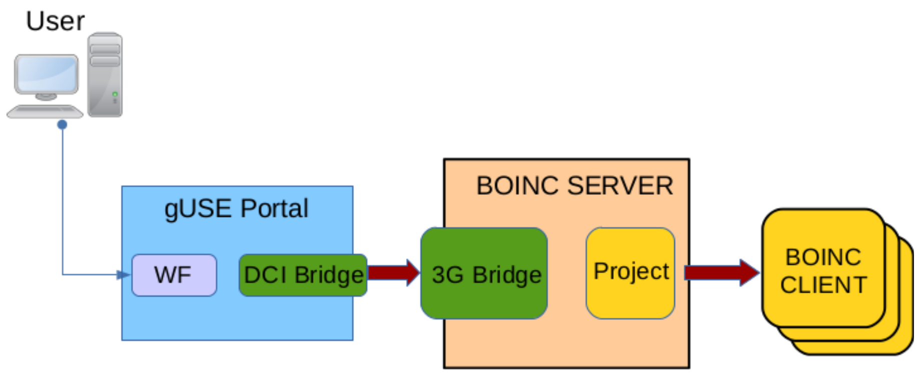


Figure 7.3: **DGCI Architecture.**

This distributed GPU-based infrastructure system is a GPU cluster where each node individually hosts a gUSE portal, a BOINC server and the BOINC GPU clients (see Figure 7.3). This system has been specifically implemented for a local GPU cluster but can be scaled on a GPU Desktop Grid infrastructure. GPU nodes have not been provided by volunteers, but by CNR-ITB (`http://www.itb.cnr.it/web/bioinformatics/home`), which supported this project. These computers, which are located in the Milan CNR-ITB headquarters, are all characterized by the following features: one 12-core Intel Xeon CPU E5-2667 2.90 GHz and four GPU NVIDIA Tesla Kepler k20c.

This system operates as follows: through gUSE, users can either create a new workflow or use a workflow already existing in the platform repository. Each job related to a workflow node is then sent to the DCI assigned to the node. In particular, GPU jobs are sent to the clients through a BOINC server based on a specific scheduling policy. Finally, results are sent through the server to the gUSE portal.

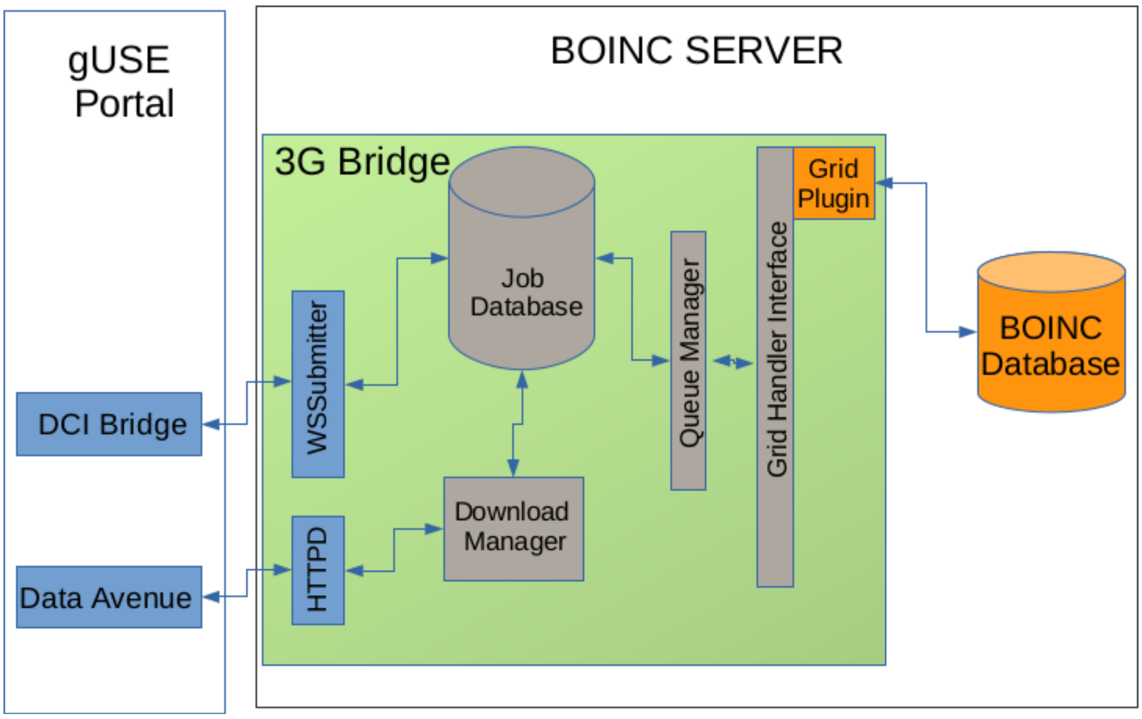


Figure 7.4: **3G Bridge Architecture.** Image derived from `http://doc.desktopgrid.hu/doku.php`

3G Bridge (Generic Grid Grid Bridge), a standard gateway between the various Grid systems (see Figure 7.3), works as the interface between gUSE DCI-Bridge and the BOINC server. 3G Bridge developed by SZTAKI (see `http://doc.desktopgrid.hu`) is used as a mediator between different types of Grid middleware. The main task of 3G Bridge is to transfer jobs received from the source Grid to the target Grid and vice versa. In this case 3G Bridge is used as a gateway between gUSE DCI-BRIDGE and BOINC server. The architecture of the 3G Bridge is composed of (see Figure 7.4):

- a web-service interface called *WSSubmitter*. It is responsible for receiving incoming jobs;
- a *Job Database* to store jobs, their status and their input/output files;

- a *Queue Manager* for job scheduling.
- *Grid Handler* interface and plugins to handle Grid-specific jobs;
- a *Download Manager* for input/output file transmission.

The *WSSubmitter* web service provides job manipulation functionalities as submission, state query, result query, cancel. The *Job Database* stores the job description and input/output files in a relational database. The *Queue Manager* creates a queue for incoming jobs. If connected to more Grids, 3G Bridge will create as many queues as the number of Grids it is connected to. The *Queue Manager* is also responsible for invoking the *Grid Handlers* to perform activities on a particular job (i.e, submit, abort, get results, update status). The *Grid Handler* interface provides a connection among various target Grids via their Grid plugins, which are responsible for communication with the back-end Grid system.

In this case, gUSe DCI-Bridge notifies the *WSSubmitter* of the operation to be performed on the job. If not already present, it send any Workunits and input files through the HTTPD protocol. Data are then stored in the *Job Database*. The *Queue Manager* periodically reads Workunits from the database and transmits them to the Grid Handler interface. This interface will use the BOINC plugin (i.e., the BOINC Submitter) to take Workunits from the queue and insert them into the BOINC database. Moreover, the Grid Handler will notify the BOINC server back-end the operation to be performed. For example, if it is a Workunit submission, the Grid Handler will also send the input files. Input file transfer is immediate, as the 3G Bridge is installed on the same computer as the BOINC server. If the query is the result of a Workunit, the Grid Handler will use the BOINC plugin to take the result from the BOINC server and save it in the Job Database. The *WSSubmitter* web service will then send the job result to the gUSE DCI-Bridge.

7.4 Porting the Proposed Tools on DGCI

gUSE enables parallel execution inside a workflow node as well as among workflow nodes. It is possible to use multiple instances of the same node with different data files (see Figure 7.5). These instances are then simultaneously processed by different Grid nodes. This is a general procedure which is thus valid for a coarse-grained data-parallel parallelization on more GPUs.

Typically, an application data-parallel parallelization requires developing a three-node workflow (see Figure 7.5). The first node is a job generator which is responsible

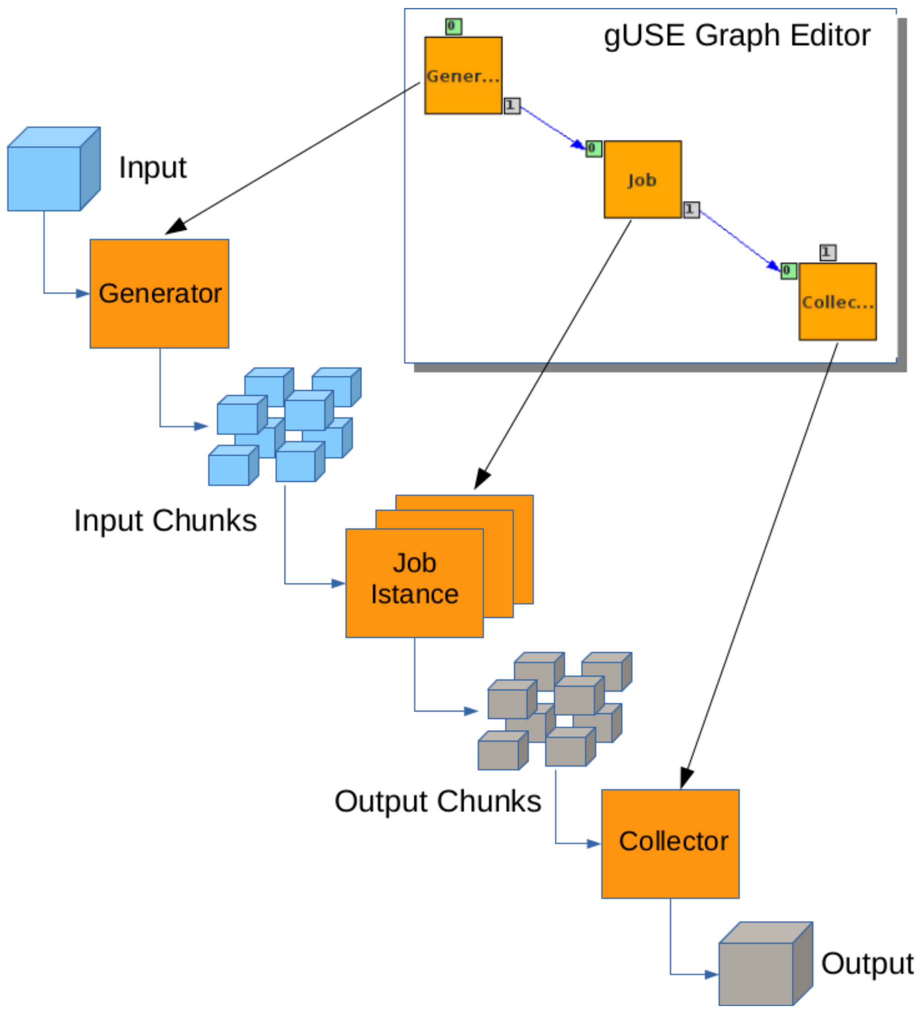


Figure 7.5: **Parallel execution among workflow nodes.**

for dividing the global input data into smaller chunks and sending them to the next node (see Generator in Figure 7.5). The second node is composed of applications (see Job Instance in Figure 7.5). In this node, gUSE creates as many instances as the chunks to be processed by the application. In the specific case of GPU applications processed on the DGCI, these instances are the Workunits sent by gUSE to the BOINC server through the DCI-Bridge. Then, based on a specific scheduling policy, the BOINC server assigns and sends each Workunit to a computer of the grid for processing. Finally, the third node is a collector job, which is responsible for combining the processing outputs of each Workunit to form a global output, so returning one output to the user (see Collector in Figure 7.5).

In general, the number of Workunits in a Grid system is only defined by the optimum parallelization level for a given application, without considering the number of available resources. This procedure can be adopted with a very high number of resources, as the probability to find free resources is very high. In DGCI, available resources are relatively few and when the number of requests is very high there is a very high risk of system congestion, as many jobs are enqueued for a long time before processing. This would cause a performance decrease, thus cancelling the parallelization effects out. This issue has been solved by developing a job generator capable of calculating a suboptmimum parallelization level based on the number of available resources. No automatic parallelization is run on the BOINC server. Indeed, the Workunit generation is left with the user and no middleware functionality notifies the number of available BOINC clients. Therefore, in order to know the number of available resources, the job generator eludes the DCI-Bridge by a direct connection to the BOINC server database. The server replies by sending status information for each client on the network. Finally, the job generator generates the number of Workunits to be simultaneously run by clients considering the application and available resources.

This procedure has been used to execute the parallelization of G-SNPM ((see Chapter 3) and GPU-BSM (see Chapter 4) on more GPUs, whereas G-CNV (see Chapter 6) has been ported to DGCI performing no parallelization.

7.4.1 Porting G-SNPM to DGCI

In order to allow the execution of G-SNPM on more GPUs, a three-node workflow has been developed (see Figure 7.6).

The first node is a job generator having one input and output port (see ports *0* and *1* of Generator in Figure 7.6). The job generator first reads the SNPS file through

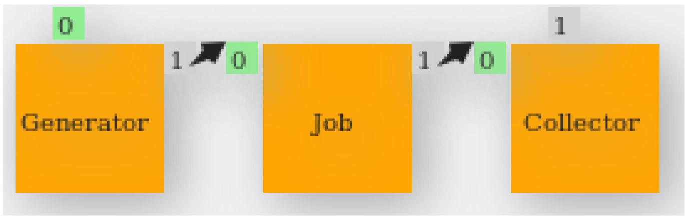


Figure 7.6: **G-SNPM Workflow.**

the input port, then divides the file into smaller file chunks based on the GPUs available on DGCI. Subsequently, the job generator sends SNP file chunks to the next node through the output port.

In the second node, gUSE generates as many G-SNPM instances as the number of SNP file chunks to be processed. Each instance produces alignments which are then sent to the next node in a single file.

In the third node, the collector job combines the processing outputs of each G-SNPM instance to create a single output.

7.4.2 Porting GPU-BSM to DGCI

GPU-BSM can be performed both in single-end mode and in paired-end mode. In order to allow execution on more GPUs, a three-node workflow has been developed (see Figure 7.7).

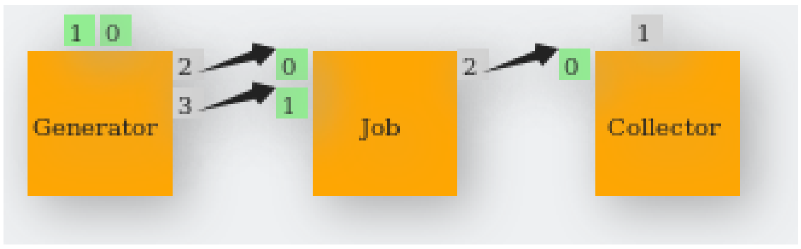


Figure 7.7: **GPU-BSM Workflow.**

The first node is a job generator having two input and output ports (see Generator in Figure 7.7). While working with single-end libraries, only one input and output port is used (i.e, port *0* and *2* of Generator in Figure 7.7), while working with paired-end libraries all input/output ports are used. When working in single-end mode, the job generator divides the bisulfite-treated single-end reads from the input port into smaller reads chunks based on the GPUs available on DGCI. Then, the job generator sends reads chunks to the next node through the output port. In paired-end mode, the job generator performs this step by dividing the bisulfite-treated paired-end reads into smaller reads chunks. It then sends these reads chunks to the next node through the two output ports.

In the second node, gUSE generates as many GPU-BSM instances as the number of reads chunks to be processed. If GPU-BSM is in single-end mode, the single-end reads chunks are read by a single port (i.e, the port *0* of Job in Figure 7.7), while if it is in paired-end mode the paired-end reads chunks are read by the two ports. Each GPU-BSM instance produces alignments which are then sent to the next node in a single file.

In the third node, the collector job combines the processing outputs of each GPU-BSM instance to create a single output.

7.4.3 Porting G-CNV to DGCI

G-CNV has currently been ported to DGCI performing no parallelization on more GPUs. The removal of adapter sequences and duplicated read sequences is performed through the CUDA-Quicksort sorting algorithm (see Section 6.2.1.4). Therefore, the development of a multi-GPU solution of G-CNV for DGCI should be preceded by a CUDA-Quicksort multi-GPU version (see Section 5.5).

In order to allow execution on DGCI, a one-node workflow has been developed. This node represents the G-CNV application having one input and output port. In the input port, G-CNV reads the file containing the read sequence library. Then, in a second task, set at node configuration, G-CNV will generate the relative outputs which will then be sent to the output port in a single file.

7.5 Discussion and Conclusions

DGCI has been mainly devised for submitting and managing workflows on various DCIs. In particular, it allows the development and use of GPU bioinformatics applications on a distributed GPU-based infrastructure. In DGCI, workflows can be created and run on more DCIs, thus allowing the integration of GPU-based applications in a workflow node.

Moreover, the DGCI workflow management system allows a further coarse-grained data-parallel parallelization of GPU applications. Indeed, DGCI enables parallel execution into a workflow node. More instances of the same node can be used with different data files. These instances are then simultaneously processed by different GPUs. This procedure has been used to parallelize G-SNPM and GPU-BSM on more GPUs, whereas G-CNV has been ported to DGCI without performing a multi-GPU parallelization. Results show that the G-SNPM and GPU-BSM execution time decreases with the increase in the number of GPU devices. Figure 7.8 shows the GPU-BSM execution time at varying the number of GPUs. As it can be observed, the execution time decreases with the increase in the number of GPUs. However, this is not a linear decrease as GPU-BSM is not completely ported to GPUs (see Chapter 4). As shown in figure 7.8, the workflow execution time is higher than the node execution time. This is mainly due to the time required for the management of the GPU-BSM node and the transfer of its inputs/outputs being too long compared with the GPU-BSM node execution time. Similar results are obtained when performing G-SNPM on more GPUs. This parallelization procedure is more efficient when the job instances execution time is so long that the time required for their management and input and output transfer becomes insignificant.

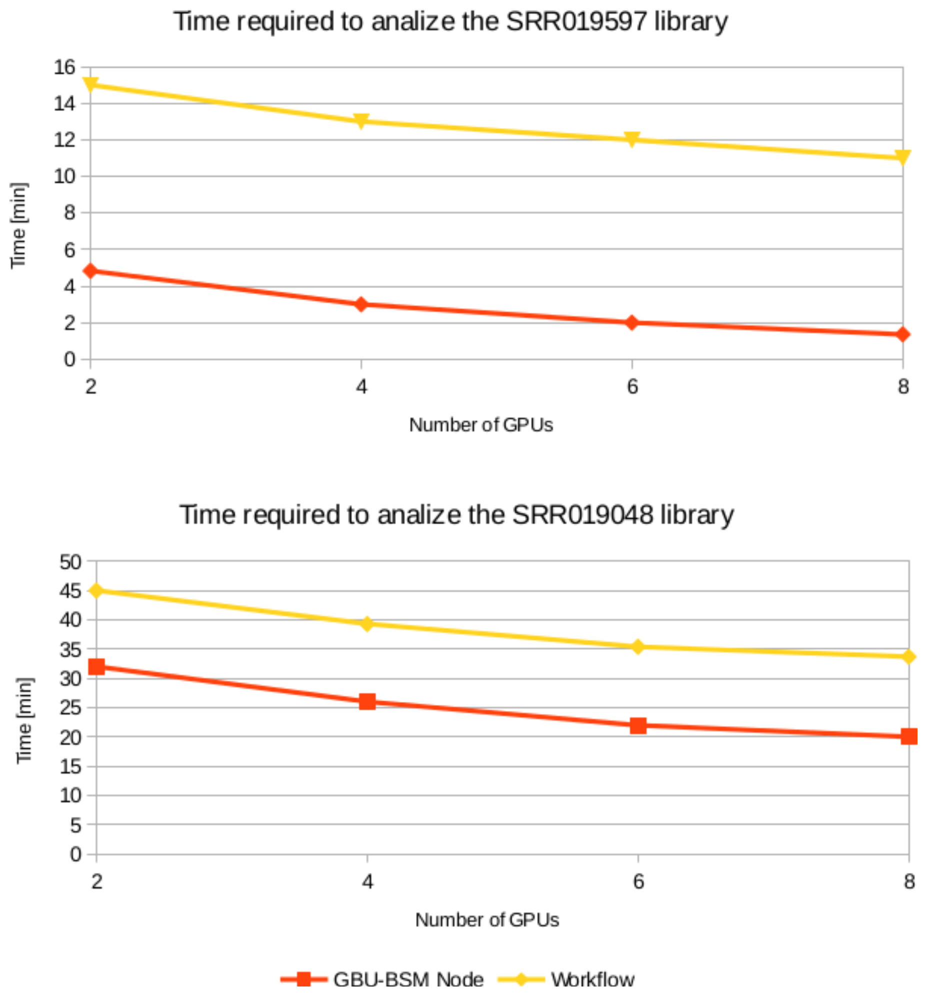


Figure 7.8: **Time required to perform the GPU-BSM workflow at varying the number of GPUs**. Two directional libraries are analyzed: *SRR019597*, which consists of 5,943,586 reads with a length of 76 bp, and *SRR019048*, which consists of 15,331,851 reads with a length of 87 bp. Computing time for the GPU-BSM node and the GPU-BSM workflow has been reported running them on 1-8 GPUs. GPU-BSM settings: *-m 4 –ungapped -l 1*. The generator and collector nodes execution times are not plotted in this figure as they are just few seconds.

Chapter 8

Conclusions and Future Work

The research activity presented in this dissertation has been focused on the adoption of innovative HPC techniques to deal with bioinformatics challenges. In particular, GPU-driven solutions aimed at using GPU hardware accelerators to solve bioinformatics issues, where new solutions were required for efficient data analysis and interpretation due to the very huge amount of available data and the complexity of tasks. In order to achieve high performances, some of the bioinformatics algorithms applicable to genome data analysis were selected, analyzed and implemented on GPUs. In particular, the following tools and systems were developed:

- **G-SNPM**: a GPU-based tool aimed at mapping nucleotide sequences representative of a single-nucleotide base polymorphism (SNP) on a reference genome (see [II] in Publications). G-SNPM is a useful tool that maps the SNP chromosomal positions on a reference genome. G-SNPM is the only general-purpose tool designed to deal with the mapping of SNPs. It uses modern GPUs to ensure a very fast mapping without compromising accuracy.

- **GPU-BSM**: a GPU-based tool aimed at mapping bisulfite-treated reads with the aim of detecting methylation levels of cytosines (see [III] in Publications). GPU-BSM is a mapping tool for the alignment of single-end and paired-end reads on a reference genome. GPU-BSM supports both gapped and ungapped alignments. Massive parallelization on GPUs enables GPU-BSM to map reads without severe limitations on the alignment process. Experimental results show that GPU-BSM is very accurate and outperforms most of the cutting-edge solutions in terms of unique best mapped reads, while keeping computational time reasonably low.

- **CUDA-Quicksort**: a GPU-based implementation of the quick-sort (see [IV] in Publications); CUDA-Quicksort is a new GPU-based implementation of the quicksort algorithm. It outperforms the two high-performance GPU-based implementations of the quicksort algorithm presented in literature: GPU-quicksort, an iterative implementation, and CDP-Quicksort, a recursive implementation. Experimental results show that algorithm is about four times faster than GPU-Quicksort and three times faster than CDP-Quicksort. CUDA-Quicksort has been used for the implementation of the G-CNV tool as described below.

- **G-CNV**: a GPU-based tool for Preparing Data to Detect CNVs with Read Depth Methods (see [V] in Publications). G-CNV is a GPU-based tool used to filter low-quality read sequences, mask low-quality nucleotides, remove adapter sequences and duplicated read sequences, map short-reads, solve multiple mapping ambiguities, build and normalize the read-depth signal. Experimental results show that G-CNV is more effective than the other tools in terms of computing time. Moreover, G-CNV is the only general-purpose tool designed to generate the read-depth signal.

Moreover, a Distributed GPU- and CPU-based Infrastructure (DGCI) system has been devised to support CPU and GPU computation. This software infrastructure, implemented through gUSE and BOINC, allows the submission and management of workflows on a wide variety of different distributed computing infrastructures. In particular, it allows the development and use of GPU bioinformatics applications on distributed GPU-based infrastructures, providing an efficient scheduling and distribution of several jobs on GPUs. Moreover, the DGCI workflow management system allows a further coarse-grained data-parallel parallelization of GPU applications. G-SNPM and GPU-BSM have currently been ported to DGCI to perform a parallelization on more GPUs. Results show that G-SNPM and GPU-BSM speed-up increases with the increase in the number of GPU devices. However, experiments show that the time required for the management of job instances and the transfer of their inputs/outputs is too long compared with the job instance execution time. Therefore, this parallelization procedure is efficient only when the job instances execution time is so long that the time required for their management and input and output transfer becomes insignificant.

The above-mentioned algorithms have been massively parallelized on NVIDIA GPUs using the CUDA programming model. The NVIDIA GPUs have been used as they were among the most performant at the time they were bought. CUDA, a NVIDIA proprietary language, has been chosen because it outperforms OpenCL on NVIDIA GPUs as the CUDA compiler produces better optimized code. The overall results show that the proposed parallel algorithms are highly performant. However, their portability remains limited to the NVIDIA GPUs, thus forcing users lacking such technology to rest on third-party infrastructures. This issue may be solved by porting the CUDA proposed solutions to OpenCL solutions. Indeed, OpenCL is designed to run on many different kinds of platforms, thus allowing the proposed algorithms to be used on several HPC platforms such as CPUs, DSPs, FPGAs and other GPUs - as the AMD GPUs - whose performance have recently got similar to that of NVIDIA GPUs. Porting CUDA solutions to OpenCL solutions is not a time-consuming task, as these two languages are very similar and automatic tools have been recently developed to perform this task (e.g.CU2CL [2] and Swan [7]).

As for GPU-BSM, it should be pointed out that it has not been completely parallelized on GPUs. GPU-BSM can be represented by a three-stage pipeline and only the second stage of the pipeline has been parallelized on GPUs. In this stage, the gain in terms of computing time resulted nearly linear with the increasing of the number of GPU devices. Nevertheless, the overall gain is not linear due to the fact that the other two stages of the pipeline have not yet been parallelized. A possible future work could be the improvement of GPU-BSM, by redesigning the third stage of the pipeline. Porting to GPUs the analysis performed at the third stage is essential to obtain a linear gain of the computing time with the increase of used GPUs.

As for G-CNV, it uses CUDA-Quicksort to remove duplicated read sequences. It should be pointed out that the current release has some limitations. For example, the library analyzed by the G-CNV to remove duplicated read sequences is limited in size. Indeed, as it is loaded in the GPU memory, its size depends on the memory dimension. So, with a GPU card equipped with 4.8 GB of global memory, libraries of up to 220 M reads can be analyzed. Therefore, a possible future work could be using CUDA-Quicksort on more GPUs to increase the size of the analyzed library. It has been planned to adapt CUDA-Quicksort to run on more GPUs. Although CUDA-Quicksort resulted to be the fastest GPU-based implementation of the quicksort algorithm, the Thrust Radix Sort is currently the fastest GPU-based sorting algorithm. However, as for clustering, the proposed CUDA-Quicksort has been adapted and used on G-CNV as it has been designed to be easily modified to scale on multiple GPUs. When CUDA-Quicksort is used on more GPUs, the size of the analyzed library increases linear with the number of GPU devices.

As for the DGCI system, it has been specifically devised on a local GPU cluster. The GPU cluster computers are not provided by volunteers, but by CNR-ITB, which supports this project. Another future work could be to realize an actual GPU Grid infrastructure (i.e. a geographically distributed infrastructure through volunteer computing). Finally the DGCI workflow management system could be used to develop new CUDA and OpenCL-based bioinformatics applications running on more platforms. In this way, users may exploit all GPU-Grid available resources, from a simple CPU to a cluster of HPC resources (NVIDIA and AMD GPUs, CPUs and FPGAs, etc.).

.

Bibliography

[1] Assemblyconverter. URL: http://www.ensembl.org/tools.html.

[2] Automating cuda-to-opencl translation. URL: http://chrec.cs.vt.edu/cu2cl/.

[3] Cuda toolkit 6.5, documentation. URL: http://docs.nvidia.com/cuda/cuda-samples/index.html.

[4] Liftover. URL: http://genome.ucsc.edu/cgi-bin/hgLiftOver?hgsid=333041007.

[5] Ncbi genome remapping service. URL: http://www.ncbi.nlm.nih.gov/genome/tools/remap.

[6] Parallel bitonic sort algorithm. URL: http://www.tools-of-computing.com/tc/CS/Sorts/bitonic_sort.htm.

[7] Swan: A simple tool for porting cuda to opencl. URL: http://www.multiscalelab.org/swan.

[8] Abecasis, G. R., Altshuler, D., Auton, A., Brooks, L. D., Durbin, R. M., Gibbs, R. A., Hurles, M. E., McVean, G. A., Bentley, D. R., Chakravarti, A., and et al. A map of human genome variation from population-scale sequencing. *Nature 467*, 7319 (2010), 1061–1073.

[9] Abel, H. J., Duncavage, E. J., Becker, N., Armstrong, J. R., Magrini, V. J., and Pfeifer, J. D. Slope: a quick and accurate method for locating non-snp structural variation from targeted next-generation sequence data. *Bioinformatics 26*, 21 (2010), 2684–2688.

[10] Abyzov, A., and Gerstein, M. Age: defining breakpoints of genomic structural variants at single-nucleotide resolution, through optimal alignments with gap excision. *Bioinformatics 27*, 5 (2011), 595–603.

[11] Abyzov, A., Urban, A. E., Snyder, M., and Gerstein, M. Cnvnator: an approach to discover, genotype, and characterize typical and atypical cnvs from family and population genome sequencing. *Genome research 21*, 6 (2011), 974–984.

[12] Advisory Committee on Genetic Modification. *Annual Report.* London, 1999.

[13] Alkan, C., Coe, B. P., and Eichler, E. E. Genome structural variation discovery and genotyping. *Nature Reviews Genetics 12*, 5 (2011), 363–376.

[14] Allen, J. D., Wang, S., Chen, M., Girard, L., Minna, J. D., Xie, Y., and Xiao, G. Probe mapping across multiple microarray platforms. *Brief Bioinform 13*, 5 (2012), 547–554.

[15] Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. Basic local alignment search tool. *J Mol Biol 215*, 3 (1990), 403–410.

[16] AMD and ATI Stream Corporations. *Introduction to OpenCL Programming*, 2010.

[17] AMD and ATI Stream Corporations. *APP SDK: A Complete Development Platform*, 2014.

[18] AMD Corporation. *AMD Graphics Cores Next (GCN) architecture*, 2012.

[19] Anderson, D. P. Boinc: A system for public-resource computing and storage. In *In 5th IEEE/ACM International Workshop on Grid Computing* (2004), pp. 4–10.

[20] Anderson, D. P., Korpela, E., and Walton, R. High-performance task distribution for volunteer computing. In *In Proceedings of the First International Conference on e-Science and Grid Computing (E-SCIENCE '05)* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 196–203.

[21] Balasko, A., Farkas, Z., and Kacsuk, P. Building science gateway by utilizing the generic ws-pgrade/guse workflow system. *Computer Science 14*, 2 (2013), 307.

[22] Bao, S., Jiang, R., Kwan, W., Wang, B., Ma, X., and Song, Y.-Q. Evaluation of next-generation sequencing software in mapping and assembly. *Journal of human genetics 56*, 6 (2011), 406–414.

[23] Barbosa-Morais, N. L., Dunning, M. J., Samarajiwa, S. A., Darot, J. F. J., Ritchie, M. E., Lynch, A. G., and Tavaré, S. A re-annotation pipeline for illumina beadarrays: improving the interpretation of gene expression data. *Nucl Acids Res 38*, 3 (2010), e17.

[24] Batcher, K. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Comput* (1968), V. . Conf., Ed., pp. 307–314.

[25] Benson, D. A., Cavanaugh, M., Clark, K., Karsch-Mizrachi, I., Lipman, D. J., Ostell, J., and Sayers, E. W. Genbank. *Nucl Acids Res 41* (2013), D36–D42.

[26] Bentley, D. R., Balasubramanian, S., Swerdlow, H. P., Smith, G. P., Milton, J., Brown, C. G., Hall, K. P., Evers, D. J., Barnes, C. L., Bignell, H. R., et al. Accurate whole human genome sequencing using reversible terminator chemistry. *Nature 456*, 7218 (2008), 53–59.

[27] Bochukova, E. G., Huang, N., Keogh, J., Henning, E., Purmann, C., Blaszczyk, K., Saeed, S., Hamilton-Shield, J., Clayton-Smith, J., O'Rahilly, S., et al. Large, rare chromosomal deletions associated with severe early-onset obesity. *Nature 463*, 7281 (2009), 666–670.

[28] Botstein, D., and Risch, N. Discovering genotypes underlying human phenotypes: past successes for mendelian disease, future approaches for complex disease. *Nature Genet 33* (2003), 228–237.

[29] Burriesci, M. S., Lehnert, E. M., and Pringle, J. R. Fulcrum: condensing redundant reads from high-throughput sequencing studies. *Bioinformatics 28*, 10 (2012), 1324–1327.

[30] Campbell, P. J., Stephens, P. J., Pleasance, E. D., O'Meara, S., Li, H., Santarius, T., Stebbings, L. A., Leroy, C., Edkins, S., Hardy, C., et al. Identification of somatically acquired rearrangements in cancer using genome-wide massively parallel paired-end sequencing. *Nature genetics 40*, 6 (2008), 722–729.

[31] Carter, N. P. Methods and strategies for analyzing copy number variation using dna microarrays. *Nature genetics 39* (2007), S16–S21.

[32] Carter, S. L., Eklund, A. C., Mecham, B. H., Kohane, I. S., and Szallasi, Z. Redefinition of affymetrix probe sets by sequence overlap

with cdna microarray probes reduces cross-platform inconsistencies in cancer-associated gene expression measurements. *BMC Bioinformatics 6*, 1 (2005), 107.

[33] Cederman, D., and Tsigas, P. A practical quicksort algorithm for graphics processors. *In the ACM Journal of Experimental Algorithmics 4*, 14 (2009).

[34] Cerin, C., and Fedak, G. *Desktop Grid Computing.* Chapman and Hall/CRC, 2012.

[35] Chen, K., Wallis, J. W., McLellan, M. D., Larson, D. E., Kalicki, J. M., Pohl, C. S., McGrath, S. D., Wendl, M. C., Zhang, Q., Locke, D. P., et al. Breakdancer: an algorithm for high-resolution mapping of genomic structural variation. *Nature methods 6*, 9 (2009), 677–681.

[36] Chen, P. Y., Cokus, S. J., and Pellegrini, M. Bs seeker: Precise mapping for bisulfite sequencing. *BMC Bioinformatics 11*, 1 (2010), 203.

[37] Chiang, D. Y., Getz, G., Jaffe, D. B., O'Kelly, M. J., Zhao, X., Carter, S. L., Russ, C., Nusbaum, C., Meyerson, M., and Lander, E. S. High-resolution mapping of copy-number alterations with massively parallel sequencing. *Nature methods 6*, 1 (2008), 99–103.

[38] Chikkagoudar, S., Wang, K., and Li, M. Genie: a software package for gene-gene interaction analysis in genetic association studies using multiple gpu or cpu cores. *BMC Research Notes 4*, 158 (2011).

[39] Cokus, S., Feng, S., Zhang, X., Chen, Z., Merriman, B., Haudenschild, C., Pradhan, S., Nelson, S., Pellegrini, M., and Jacobsen, S. Shotgun bisulphite sequencing of the arabidopsis genome reveals dna methylation patterning. *Nature 452*, 7184 (2008), 215–219.

[40] David, M., Dzamba, M., Lister, D., Ilie, L., and Brudno, M. Shrimp2: sensitive yet practical short read mapping. *Bioinformatics 27*, 7 (2011), 1011–1012.

[41] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

[42] Dohm, J. C., Lottaz, C., Borodina, T., and Himmelbauer, H. Substantial biases in ultra-short read data sets from high-throughput dna sequencing. *Nucleic acids research 36*, 16 (2008), e105–e105.

[43] Esteller, M., and Herman, J. G. Cancer as an epigenetic disease: Dna methylation and chromatin alterations in human tumours. *The Journal of Pathology 196*, 1 (2002), 1–7.

[44] Estrada, T., Taufer, M., and Anderson, D. P. Performance prediction and analysis of boinc projects: An empirical study with emboinc. *Journal of Grid Computing 7*, 4 (2009), 537–554.

[45] Fadista, J., and Bendixen, C. Genomic position mapping discrepancies of commercial snp chips. *PLoS One 7*, 2 (February 2012), e31025.

[46] Fellay, J., Thompson, A. J., Ge, D., Gumbs, C. E., Urban, T. J., Shianna, K. V., Little, L. D., Qiu, P., Bertelsen, A. H., Watson, M., Warner, A., Muir, A. J., Brass, C., Albrecht, J., Sulkowski, M., McHutchison, J. G., and Goldstein, D. B. Itpa gene variants protect against anaemia in patients treated for chronic hepatitis c. *Nature 464*, 7287 (March 2010), 405–408.

[47] Ferragina, P., and Manzini, G. *Opportunistic data structures with applications.* Foundations of Computer Science: Proceedings 41st Annual Symposium on. IEEE: 390-398p, 2000.

[48] Feuk, L., Carson, A. R., and Scherer, S. W. Structural variation in the human genome. *Nature Reviews Genetics 7*, 2 (2006), 85–97.

[49] Feuk, L., Marshall, C. R., Wintle, R. F., and Scherer, S. W. Structural variants: changing the landscape of chromosomes and design of disease studies. *Human molecular genetics 15*, suppl 1 (2006), R57–R66.

[50] Foster, I., and Kesselman, C. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[51] Foster, I., Kesselman, C., et al. The anatomy of the grid: Enabling scalable virtual organizations. *High Performance Computing Applications 15*, 3 (2001), 200–222.

[52] Frommer, M., McDonald, L. E., Millar, D. S., Collis, C. M., Watt, F., Grigg, G. W., Molloy, P. L., and Paul, C. L. A genomic sequencing protocol that yields a positive display of 5-methylcytosine residues in individual dna strands. *Proceedings of the National Academy of Sciences 89*, 5 (1992), 1827–1831.

[53] Fu, L., Niu, B., Zhu, Z., Wu, S., and Li, W. Cd-hit: accelerated for clustering the next-generation sequencing data. *Bioinformatics 28*, 23 (2012), 3150–3152.

[54] Gagliardi, F., Jones, B., et al. Building an infrastructure for scientific grid computing: status and goals of the egee project. *Philos Transact A Math Phys Eng Sci 363*, 1833 (2005), 1729–42.

[55] Gautier, L., Møller, M., Friis-Hansen, L., and Knudsen, S. Alternative mapping of probes to genes for affymetrix chips. *BMC Bioinformatics 5*, 1 (2004), 111.

[56] Gentleman, R. C., Carey, V. J., Bates, D. M., Bolstad, B., Dettling, M., Dudoit, S., Ellis, B., Gautier, L., Ge, Y., Gentry, J., Hornik, K., Hothorn, T., Huber, W., Iacus, S., Irizarry, R., Leisch, F., Li, C., Maechler, M., Rossini, A. J., Sawitzki, G., Smith, C., Smyth, G., Tierney, L., Yang, J. Y. H., and Zhang, J. Bioconductor: open software development for computational biology and bioinformatics. *Genome Biol 5*, 10 (2004), R80.

[57] Gibbs, R. A., Belmont, J. W., Hardenbol, P., Willis, T. D., and et al. The international hapmap project. *Nature 426*, 6968 (2003), 789–796.

[58] Glaskowsky, P. N. The first complete gpu computing architecture. White paper, NVIDIA Corporation, 2009.

[59] Gomez-Alvarez, V., Teal, T. K., and Schmidt, T. M. Systematic artifacts in metagenomes from complex microbial communities. *The ISME journal 3*, 11 (2009), 1314–1317.

[60] Gonzalez, D. L., de Vega, F. F., Trujillo, L., Olague, G., Cardenas, M., Araujo, L., Castillo, P., Sharman, K., and Silva, A. Interpreted applications within boinc infrastructure. In *In Fernando Silva, Gaspar Barreira, and Ligia Ribeiro* (Porto, Portugal, May 2008), I. 2nd Iberian Grid Infrastructure Conference Proceedings, Ed., pp. 261–272.

[61] Govindaraju, N. K., Raghuvanshi, N., Henson, M., and Manocha, D. A cache-efficient sorting algorithm for database and data mining computations using graphics processors. Technical report, University of North Carolina, Chapel Hill, 2005.

[62] Govindaraju, N. K., Raghuvanshi, N., and Manocha, D. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *In Proceedings of the 2005 ACM SIGMOD international conference on Management of data (SIGMOD '05)* (2005), pp. 611–622.

[63] Goya, R., Sun, M. G., Morin, R. D., Leung, G., Ha, G., Wiegand, K. C., Senz, J., Crisan, A., Marra, M. A., Hirst, M., Huntsman, D., Murphy, K. P., Aparicio, S., and Shah, S. P. Snvmix: predicting single nucleotide variants from next-generation sequencing of tumors. *Bioinformatics 6*, 26 (March 2010), 730–736.

[64] Gress, A., and Zachmann, G. Gpu-abisort: optimal parallel sorting on stream architectures. In *In Proceedings of the 20th international conference on Parallel and distributed processing (IPDPS'06)* (Washington, DC, USA, 2006), I. C. Society, Ed., p. 45.

[65] Guo, W., Fiziev, P., Yan, W., Cokus, S., Sun, X., Zhang, M., Chen, P., and Pellegrini, M. Bs-seeker2: a versatile aligning pipeline for bisulfite sequencing data. *BMC Genomics 14*, 1 (2013), 774.

[66] Harismendy, O., Ng, P. C., Strausberg, R. L., Wang, X., Stockwell, T. B., Beeson, K. Y., Schork, N. J., Murray, S. S., Topol, E. J., Levy, S., et al. Evaluation of next generation sequencing platforms for population targeted sequencing studies. *Genome Biol 10*, 3 (2009), R32.

[67] Harris, E., Ponts, N., Le Roch, K., and Lonardi, S. Brat-bw: efficient and accurate mapping of bisulfite-treated reads. *Bioinformatics 28*, 13 (2012), 1795–1796.

[68] Harris, M., Sengupta, S., and Owens, J. D. Parallel prefix sum(scan) with cuda. In *GPU Gems 3*, H. Nguyen, Ed., vol. 3. Addison-Wesley Professional, 2007, ch. 39.

[69] Heidelberger, P., Norton, A., and Robinson, J. T. Parallel quicksort using fetch-and-add. *IEEE Transactions on Computers 39*, 1 (1990), 133–138.

[70] Helman, D. R., Bader, D. A., and Jaja, J. A randomized parallel sorting algorithm with an experimental study. *J. Parallel Distributed Computing 1*, 52 (1998), 1–23.

[71] Hillier, L. W., Marth, G. T., Quinlan, A. R., Dooling, D., Fewell, G., Barnett, D., Fox, P., Glasscock, J. I., Hickenbotham, M.,

Huang, W., et al. Whole-genome sequencing and variant discovery in c. elegans. *Nature methods 5*, 2 (2008), 183–188.

[72] Hirschhorn, J. N., and Daly, M. J. Genome-wide association studies for common diseases and complex traits. *Nature Rev Genet 6*, 2 (2005), 95–108.

[73] Hoare, C. A. R. Quicksort. *Computer Journal 4*, 5 (1962), 10–15.

[74] Hoberock, J., and Bell, N. Thrust: A parallel template library. Technical report, 2010.

[75] Hong, H., Xu, L., Liu, J., Jones, W. D., Su, Z., Ning, B., Perkins, R., Ge, W., Miclaus, K., Zhang, L., Park, K., Green, B., Han, T., Fang, H., Lambert, C. G., Vega, S. C., Lin, S. M., Jafari, N., Czika, W., Wolfinger, R., Doodsaid, F., Tong, W., and Shi, L. Technical reproducibility of genotyping snp arrays used in genome-wide association studies. *PLoS One 7*, 9 (September 2012), e44483.

[76] Hormozdiari, F., Alkan, C., Eichler, E. E., and Sahinalp, S. C. Combinatorial algorithms for structural variation detection in high-throughput sequenced genomes. *Genome research 19*, 7 (2009), 1270–1278.

[77] Hormozdiari, F., Hajirasouliha, I., Dao, P., Hach, F., Yorukoglu, D., Alkan, C., Eichler, E. E., and Sahinalp, S. C. Next-generation variationhunter: combinatorial algorithms for transposon insertion discovery. *Bioinformatics 26*, 12 (2010), i350–i357.

[78] Hormozdiari, F., Hajirasouliha, I., McPherson, A., Eichler, E. E., and Sahinalp, S. C. Simultaneous structural variation discovery among multiple paired-end sequenced genomes. *Genome research 21*, 12 (2011), 2203–2212.

[79] Hunninghake, G. W., and Gadek, J. E. The alveloar macrophage. In *Cultured Human Cells and Tissues*, T. J. R. Harris, Ed. Academic Press, New York, 1995, pp. 54–56. Stoner G (Series Editor): Methods and Perspectives in Cell Biology, vol 1.

[80] Hurles, M. E., Dermitzakis, E. T., and Tyler-Smith, C. The functional impact of structural variation in humans. *Trends in Genetics 24*, 5 (2008), 238–245.

[81] Hwang, K. B., Kong, S. W., Greenberg, S. A., and Park, P. J. Combining gene expression data from different generations of oligonucleotide arrays. *BMC Bioinformatics 5*, 1 (2004), 159.

[82] Iqbal, Z., Caccamo, M., Turner, I., Flicek, P., and McVean, G. De novo assembly and genotyping of variants using colored de bruijn graphs. *Nature genetics 44*, 2 (2012), 226–232.

[83] Ivakhno, S., Royce, T., Cox, A. J., Evers, D. J., Cheetham, R. K., and Tavaré, S. Cnaseg-a novel framework for identification of copy number changes in cancer from second-generation sequencing data. *Bioinformatics 26*, 24 (2010), 3051–3058.

[84] Jaenisch, R., and Bird, A. Epigenetic regulation of gene expression: how the genome integrates intrinsic and environmental signals. *Nat Genet 33* (2003), 245–254.

[85] Jelinek, J., Gharibyan, V., Estecio, M., Kondo, K., He, R., Chung, W., Lu, Y., Zhang, N., Liang, S., Kantarjian, H., Cortes, J., and Issa, J.-P. J. Aberrant dna methylation is associated with disease progression, resistance to imatinib and shortened survival in chronic myelogenous leukemia. *PLoS ONE 6*, 7 (2011).

[86] Jones, X. Zeolites and synthetic mechanisms. In *Proceedings of the First National Conference on Porous Sieves: 27-30 June 1996; Baltimore* (1996), Y. Smith, Ed., Stoneham: Butterworth-Heinemann, pp. 16–27.

[87] Kacsuk, P. *Science Gateways for Distributed Computing Infrastructures.* Springer International Publishing, 2014.

[88] Kapasi, U. J., Dally, W. J., Rixner, S., Mattson, P. R., Owens, J. D., and Khailany, B. Efficient conditional operations for data-parallel architectures. In *In Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (Micro-33)* (2000), pp. 159–170.

[89] Karimi, K., Dickson, N. G., and Hamze, F. A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581* (2010).

[90] Karimi, K., Dickson, N. G., and Hamze, F. A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581* (2010).

[91] Kent, W. J. Blat–the blast-like alignment tool. *Genome Res 12*, 4 (2002), 656–664.

[92] Kipfer, P., Segal, M., and Westermann, R. Uberflow: a gpu-based particle engine. In *In Proceedings of the 2004 ACM SIGGRAPH/Eurographics conference on Graphics hardware (EGGH '04).* (2000), pp. 115–122.

[93] Kipfer, P., and Westermann, R. Improved gpu sorting. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation.*, H. Nguyen, Ed., vol. 2. Addison-Wesley Professional, 2005, pp. 205–222.

[94] Kircher, M., and Kelso, J. High-throughput dna sequencing–concepts and limitations. *Bioessays 32*, 6 (2010), 524–536.

[95] Kirk, D., and Hwu, W. *Programming Massively Parallel Processors: A Hands- on Approach.* Morgan Kaufmann Publishers, 2010.

[96] Klus, P., Lam, S., Lyberg, D., Cheung, M. S., Pullan, G., McFarlane, I., Yeo, G. S., and Lam, B. Y. Barracuda-a fast short read sequence aligner using graphics processing units. *BMC research notes 5*, 1 (2012), 27.

[97] Kohavi, R. *Wrappers for performance enhancement and obvious decision graphs.* PhD thesis, Stanford University, Computer Science Department, 1995.

[98] Koonin, E. V., Altschul, S. F., and Bork, P. Brca1 protein products: functional motifs. *Nat Genet 13* (1996), 266–267.

[99] Korbel, J. O., Abyzov, A., Mu, X. J., Carriero, N., Cayting, P., Zhang, Z., Snyder, M., and Gerstein, M. B. Pemer: a computational framework with simulation-based error models for inferring genomic structural variants from massive paired-end sequencing data. *Genome Biol 10*, 2 (2009), R23.

[100] Krueger, F., and Andrews, S. R. Bismark: A flexible aligner and methylation caller for bisulfite-seq applications. *Bioinformatics 27*, 11 (2011), 1571–1572.

[101] Krueger, F., Kreck, B., Franke, A., and Andrews, S. R. Dna methylome analysis using short bisulfite sequencing data. *Nat Methods 9*, 2 (2012), 145–151.

[102] Laird, P. Principles and challenges of genome-wide dna methylation analysis. *Nat Rev Genet 11*, 3 (2010), 191–203.

[103] Langmead, B., and Salzberg, S. Fast gapped-read alignment with bowtie 2. *Nat Methods 9*, 4 (2012), 357–359.

[104] Langmead, B., Trapnell, C., Pop, M., Salzberg, S. L., et al. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biol 10*, 3 (2009), R25.

[105] LastName1, A., LastName2, A., and LastName2, A. Article title. *Frontiers in Neuroscience 30*, 30 (2013), 10127–10134.

[106] Lee, S., Hormozdiari, F., Alkan, C., and Brudno, M. Modil: detecting small indels from clone-end sequencing with mixtures of distributions. *Nature methods 6*, 7 (2009), 473–474.

[107] Leischner, N., Osipov, V., and Sanders, P. Gpu sample sort. *J. Parallel Distributed Computing* (2009). CoRR abs/0909.5649.

[108] Li, H., and Durbin, R. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics 25*, 14 (2009), 1754–1760.

[109] Li, H., and Durbin, R. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics 25*, 14 (2009), 1754–1760.

[110] Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G., Durbin, R., et al. The sequence alignment/map format and samtools. *Bioinformatics 25*, 16 (2009), 2078–2079.

[111] Li, H., Ruan, J., and Durbin, R. Mapping short dna sequencing reads and calling variants using mapping quality scores. *Genome research 18*, 11 (2008), 1851–1858.

[112] Li, R., Li, Y., Kristiansen, K., and Wang, J. Soap: Short oligonucleotide alignment program. *Bioinformatics 24*, 5 (2008), 713–714.

[113] Li, R., Yu, C., Li, Y., Lam, T.-W., Yiu, S.-M., Kristiansen, K., and Wang, J. Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics 25*, 15 (2009), 1966–1967.

[114] Li, W., and Godzik, A. Cd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences. *Bioinformatics 22*, 13 (2006), 1658–1659.

[115] Lister, R., Pelizzola, M., Dowen, R., Hawkins, R., Hon, G., Tonti-Filippini, J., Nery, J., Lee, L., Ye, Z., Ngo, Q.-M., et al. Human dna methylomes at base resolution show widespread epigenomic differences. *Nature 462*, 7271 (2009), 315–322.

[116] Liu, C.-M., Wong, T., Wu, E., Luo, R., Yiu, S.-M., Li, Y., Wang, B., Yu, C., Chu, X., Zhao, K., et al. Soap3: ultra-fast gpu-based parallel alignment tool for short reads. *Bioinformatics 28*, 6 (2012), 878–879.

[117] Liu, H., Zeeberg, B. R., Qu, G., Koru, A. G., Ferrucci, A., Kahn, A., Ryan, M. C., Nuhanovic, A., Munson, P. J., Reinhold, W. C., Kane, D. W., and Weinstein, J. N. Affyprobeminer: a web resource for computing or retrieving accurately redefined affymetrix probe sets. *Bioinformatics 23*, 18 (2007), 2385–2390.

[118] Liu, Y., Maskell, D. L., and Schmidt, B. Cudasw++: optimizing smith-waterman sequence database searches for cuda-enabled graphics processing units. *BMC Research Notes 2*, 73 (2009).

[119] Liu, Y., Schmidt, B., and Maskell, D. L. 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 (2010), 93.

[120] Liu, Y., Schmidt, B., and Maskell, D. L. Cushaw: a cuda compatible short read aligner to large genomes based on the burrows–wheeler transform. *Bioinformatics 28*, 14 (2012), 1830–1837.

[121] Luo, R., Wong, T., Zhu, J., Liu, C.-M., Zhu, X., Wu, E., Lee, L.-K., Lin, H., Zhu, W., Cheung, D. W., et al. Soap3-dp: fast, accurate and sensitive gpu-based short read aligner. *PloS one 8*, 5 (2013), e65632.

[122] Luo, R., Wong, T., Zhu, J., Liu, C.-M., Zhu, X., Wu, E., Lee, L.-K., Lin, H., Zhu, W., Cheung, D. W., Ting, H.-F., Yiu, S.-M., Peng, S., Yu, C., Li, Y., Li, R., and Lam, T.-W. Soap3-dp: Fast, accurate and sensitive gpu-based short read aligner. *PLoS ONE 8*, 5 (2013), e65632.

[123] Magi, A., Tattini, L., Pippucci, T., Torricelli, F., and Benelli, M. Read count approach for dna copy number variants detection. *Bioinformatics 28*, 4 (2012), 470–478.

[124] Malde, K. The effect of sequence quality on sequence alignment. *Bioinformatics 24*, 7 (2008), 897–900.

[125] Manavski, S. A., and Valle, G. Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC bioinformatics 9*, Suppl 2 (2008), S10.

[126] Manconi, A., Orro, A., Manca, E., Armano, G., and Milanesi, L. Gpu-bsm: A gpu-based tool to map bisulfite-treated reads. *PloS one 9*, 5 (2014), e97277.

[127] Manconi, A., Orro, A., Manca, E., Armano, G., and Milanesi, L. A tool for mapping single nucleotide polymorphisms using graphics processing units. *BMC bioinformatics 15*, 1 (2014), 1–13.

[128] Marco-Sola, S., Sammeth, M., Guigó, R., and Ribeca, P. The gem mapper: fast, accurate and versatile alignment by filtration. *Nature methods 9*, 12 (2012), 1185–1188.

[129] Margulis, L. *Origin of Eukaryotic Cells*. Yale University Press, New Haven, 1970.

[130] Martin, M. Cutadapt removes adapter sequences from high-throughput sequencing reads. *EMBnet. journal 17*, 1 (2011), pp–10.

[131] Mattina, M. Architecture and perfomance of the tile-gx processor family. White paper, Tilera Corporation, 2014.

[132] McPherson, R., Pertsemlidis, A., Kavaslar, N., Stewart, A., Roberts, R., Cox, D. R., Hinds, D. A., Pennacchio, L. A., Tybjaerg-Hansen, A., Folsom, A. R., Boerwinkle, E., Hobbs, H. H., and Cohen, J. C. A common allele on chromosome 9 associated with coronary heart disease. *Science 316*, 5830 (June 2007), 1488–1491.

[133] Medvedev, P., Fiume, M., Dzamba, M., Smith, T., and Brudno, M. Detecting copy number variation with mated short reads. *Genome research 20*, 11 (2010), 1613–1622.

[134] Medvedev, P., Stanciu, M., and Brudno, M. Computational methods for discovering structural variation with next-generation sequencing. *Nature methods 6* (2009), S13–S20.

[135] Meissner, A., Gnirke, A., Bell, G. W., Ramsahoye, B., Lander, E. S., and Jaenisch, R. Reduced representation bisulfite sequencing for comparative high-resolution dna methylation analysis. *Nucleic Acids Res. 33*, 18 (2005), 5868–5877.

[136] Merelli, I., Calabria, A., Cozzi, P., Viti, F., Mosca, E., and Milanesi, L. Snpranker 2.0: a gene-centric data mining tool for diseases associated snp prioritization in gwas. *BMC Bioinformatics 14*, Supp 1 (January 2013), S9.

[137] Merikangas, A. K., Corvin, A. P., and Gallagher, L. Copy-number variants in neurodevelopmental disorders: promises and challenges. *Trends in Genetics 25*, 12 (2009), 536–544.

[138] Miller, C. A., Hampton, O., Coarfa, C., and Milosavljevic, A. Readdepth: a parallel r package for detecting copy number alterations from short sequencing reads. *PloS one 6*, 1 (2011), e16327.

[139] Mills, R. E., Walter, K., Stewart, C., Handsaker, R. E., Chen, K., Alkan, C., Abyzov, A., Yoon, S. C., Ye, K., Cheetham, R. K., et al. Mapping copy number variation by population-scale genome sequencing. *Nature 470*, 7332 (2011), 59–65.

[140] Mu, J. C., Jiang, H., Kiani, A., Mohiyuddin, M., Asadi, N. B., and Wong, W. H. Fast and accurate read alignment for resequencing. *Bioinformatics 28*, 18 (2012), 2366–2373.

[141] Munshi, A. *The OpenCL Specification Version 1.2, Rev. 19*, 2012.

[142] Munshi, A., et al. The opencl specification. *Khronos OpenCL Working Group 1* (2009), l1–15.

[143] nad Perl Y., D. M., L., R., and M., S. The periodic balanced sorting network. *Journal of the ACM 36*, 4 (2009), 738–757.

[144] Nijkamp, J. F., van den Broek, M. A., Geertman, J.-M. A., Reinders, M. J., Daran, J.-M. G., and de Ridder, D. De novo detection of copy number variation by co-assembly. *Bioinformatics 28*, 24 (2012), 3195–3202.

[145] NVIDIA, and Portland, G. *The OpenACC Application Programming Interface, v1.0.* CAPS Enterprise, Cray Inc., 2011.

[146] NVIDIA, and Portland, G. *The OpenACC Application Programming Interface, v2.0.* CAPS Enterprise, Cray Inc., 2013.

[147] NVIDIA Corporation. *Compute-PTX: Parallel thread execution, ISA Version 1.1*, 2007.

[148] NVIDIA Corporation. *Compute unified device architecture programming guide*, 2007.

[149] NVIDIA Corporation. *The Fastest, Most Efficient HPC Architecture Ever Built*, 2012.

[150] NVIDIA Corporation. *CUDA C Best Practices Guide v6.5*, 2014.

[151] NVIDIA Corporation. *CUDA C Programming Guide v6.5*, 2014.

[152] NVIDIA Corporation. *GeForce GTX 750 Ti. Featuring First-Generation Maxwell GPU Technology, Designed for Extreme Performance per Watt*, 2014.

[153] Orengo, C. A., Bray, J. E., Hubbard, T., LoConte, L., and Sillitoe, I. Analysis and assessment of ab initio three-dimensional prediction, secondary structure, and contacts prediction. *Proteins Suppl 3* (1999), 149–170.

[154] Orro, A., Guffanti, G., Salvi, E., Macciardi, F., and Milanesi, L. Snplims: a data management system for genome wide association studies. *BMC Bioinformatics 9*, Suppl 2 (2008), S13.

[155] Orro, A., Manconi, A., Manca, E., Armano, G., and Milanesi, L. G-snpm-a gpu-based snp mapping tool. *EMBnet. journal 18*, B (2012), 138–139.

[156] OtherAuthor, N., and Coauthor, N. S. Article title. *Frontiers in Genetics 30*, 49 (2012), 16417–16418.

[157] Otto, C., Stadler, P., and Hoffmann, S. Fast and sensitive mapping of bisulfite-treated sequencing data. *Bioinformatics 28*, 13 (2012), 1698–1704.

[158] Otto, C., Stadler, P. F., , and Hoffmann, S. Fast and sensitive mapping of bisulfite-treated sequencing data. *Bioinformatics 28*, 13 (2012), 1698–1704.

[159] Patel, R. K., and Jain, M. Ngs qc toolkit: a toolkit for quality control of next generation sequencing data. *PloS one 7*, 2 (2012), e30619.

[160] Perry, G. H., Tchinda, J., McGrath, S. D., Zhang, J., Picker, S. R., Cáceres, A. M., Iafrate, A. J., Tyler-Smith, C., Scherer, S. W., Eichler, E. E., et al. Hotspots for copy number variation in chimpanzees and humans. *Proceedings of the National Academy of Sciences 103*, 21 (2006), 8006–8011.

[161] Pireddu, L., Leo, S., and Zanetti, G. Seal: a distributed short read mapping and duplicate removal tool. *Bioinformatics 27*, 15 (2011), 2159–2160.

[162] Pruitt, K. D., Tatusova, T., and Maglott, D. R. Ncbi reference sequences (refseq): a curated non-redundant sequence database of genomes, transcripts and proteins. *Nucl Acids Res 35*, suppl 1 (2007), D61–D65.

[163] Purcell, J., Donner, C., Cammarano, M., Jensen, H. W., and Hanrahan, P. Photon mapping on programmable graphics hardware. In *In Proceedings of the 2003 Annual ACM SIGGRAPH/Eurographics Conference on Graphics Hardware (EGGH '03)* (2003), pp. 41–50.

[164] Robertson, K. D., and Jones, A. P. Dna methylation: past, present and future directions. *Carcinogenesis 21*, 3 (2000), 461–467.

[165] Ruffalo, M., LaFramboise, T., and Koyutürk, M. Comparative analysis of algorithms for next-generation sequencing read alignment. *Bioinformatics 27*, 20 (2011), 2790–2796.

[166] Rumble, S. M., Lacroute, P., Dalca, A. V., Fiume, M., Sidow, A., and Brudno, M. Shrimp: accurate mapping of short color-space reads. *PLoS computational biology 5*, 5 (2009), e1000386.

[167] Sachidanandam, R., Weissman, D., Schmidt, S. C., Kakol, J. M., Stein, L. D., Marth, G., Sherry, S., Mullikin, J. C., Mortimore, B. J., Willey, D. L., et al. A map of human genome sequence variation containing 1.42 million single nucleotide polymorphisms. *Nature 409*, 6822 (2001), 928–933.

[168] Satish, N., Harris, M., and Garland, M. Designing efficient sorting algorithms for manycore gpus. In *In: Parallel Distributed Processing (IPDPS '09)* (2009), I. I. Symposium, Ed., pp. 1 –10.

[169] Schatz, M. C. Cloudburst: highly sensitive read mapping with mapreduce. *Bioinformatics 25*, 11 (2009), 1363–1369.

[170] Schnepf, E. From prey via endosymbiont to plastids: comparative studies in dinoflagellates. In *Origins of Plastids*, R. A. Lewin, Ed., 2nd ed., vol. 2. Chapman and Hall, New York, 1993, pp. 53–76.

[171] Schork, N. J., Fallin, D., and Lanchbury, J. S. Single nucleotide polymorphisms and the future of genetic epidemiology. *Clin Genet 58*, 4 (2000), 250–264.

[172] Sengupta, S., Harris, M., Zhang, Y., and Owens, J. D. Scan primitives for gpu computing. In *In Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (2007), pp. 97–106.

[173] Sharma, M. D., Huang, L., Choi, J.-H., Lee, E.-J., Wilson, J. M., Lemos, H., Pan, F., Blazar, B. R., Pardoll, D. M., Mellor, A. L., et al. An inherently bifunctional subset of foxp3$^+$ t helper cells is controlled by the transcription factor eos. *Immunity 38*, 5 (2013), 998–1012.

[174] Sherry, S. T., Ward, M.-H., Kholodov, M., Baker, J., Phan, L., Smigielski, E. M., and Sirotkin, K. dbsnp: the ncbi database of genetic variation. *Nucl Acids Res 29*, 1 (January 2001), 308–311.

[175] Shi, H., Schmidt, B., Liu, W., and Müller-Wittig, W. Quality-score guided error correction for short-read sequencing data using cuda. *Procedia Computer Science 1*, 1 (2010), 1129–1138.

[176] Sindi, S., Helman, E., Bashir, A., and Raphael, B. J. A geometric approach for classification and comparison of structural variants. *Bioinformatics 25*, 12 (2009), i222–i230.

[177] Sintorn, E., and Assarsson, U. Fast parallel gpu-sorting using a hybrid algorithm. *J. Parallel Distributed Computing 68*, 10 (1962), 1381–1388.

[178] Smith, A. D., Chung, W.-Y., Hodges, E., Kendall, J., Hannon, G., Hicks, J., Xuan, Z., and Zhang, M. Q. Updates to the rmap short-read mapping software. *Bioinformatics 25*, 21 (2009), 2841–2842.

[179] Smith, A. D., Xuan, Z., and Zhang, M. Q. Using quality scores and longer reads improves accuracy of solexa read mapping. *BMC bioinformatics 9*, 1 (2008), 128.

[180] Smith, T. F., and Waterman, M. S. Identification of common molecular subsequences. *Molecular Biology 147*, 1 (1981), 195–197.

[181] Smith, Y., Ed. *Proceedings of the First National Conference on Porous Sieves: 27-30 June 1996; Baltimore* (Stoneham, 1996), Butterworth-Heinemann.

[182] Stefansson, H., Rujescu, D., Cichon, S., Pietiläinen, O. P., Ingason, A., Steinberg, S., Fossdal, R., Sigurdsson, E., Sigmundsson, T., Buizer-Voskamp, J. E., et al. Large recurrent microdeletions associated with schizophrenia. *nature 455*, 7210 (2008), 232–236.

[183] Swain, M., Hunniford, T., Dubitzky, W., Mandel, J., and Palfreyman, N. Reverse-engineering gene-regulatory networks using evolutionary algorithms and grid computing. *Clinical Monitoring and Computing 19*, 4-5 (2005), 329–337.

[184] Treangen, T. J., and Salzberg, S. L. Repetitive dna and next-generation sequencing: computational challenges and solutions. *Nature Reviews Genetics 13*, 1 (2011), 36–46.

[185] Tsigas, P., and Zhang, Y. A simple, fast parallel implementation of quicksort and its performance evaluation on sun enterprise 10000. In *In Proceedings of the 11th Euromicro Conference on Parallel Distributed and Network-based Processing* (2003), pp. 372–381.

[186] Tuupanen, S., Turunen, et al. The common colorectal cancer predisposition snp rs6983267 at chromosome 8q24 confers potential to enhanced wnt signaling. *Nature Genet 41* (June 2009), 885–890.

[187] Wang, C., Schueler-Furman, O., Andre, I., et al. Rosettadock in capri rounds 6-12. *Proteins 69*, 4 (December 2007), 758–763.

[188] Wang, W. Y. S., Barratt, B. J., Clayton, D. G., and Todd, J. A. Genome-wide association studies: theoretical and practical concerns. *Nature Rev Genet 6*, 2 (2005), 109–118.

[189] White, T. *Hadoop: the definitive guide: the definitive guide.* " O'Reilly Media, Inc.", 2009.

[190] White, T. *Hadoop: The definitive guide.* Sebastopol: O'Reilly Media;, 2012.

[191] Wilkinson, B. *Grid Computing: Techniques and Applications.* Chapman & Hall/CRC Computational Science, 2009.

[192] Xi, R., Luquette, J., Hadjipanayis, A., Kim, T.-M., and Park, P. J. Bic-seq: a fast algorithm for detection of copy number alterations based on high-throughput sequencing data. *Genome biology 11*, Suppl 1 (2010), O10.

[193] Xi, Y., Bock, C., Müller, F., Sun, D., Meissner, A., and Li, W. Rrbsmap: a fast, accurate and user-friendly alignment tool for reduced representation bisulfite sequencing. *Bioinformatics 28*, 3 (2012), 430–432.

[194] Xi, Y., and Li, W. Bsmap: Whole genome bisulfite sequence mapping program. *BMC Bioinformatics 10* (2009), 232.

[195] Xie, C., and Tammi, M. T. Cnv-seq, a new method to detect copy number variation using high-throughput sequencing. *BMC bioinformatics 10*, 1 (2009), 80.

[196] Xu, H., Luo, X., Qian, J., Pang, X., Song, J., Qian, G., Chen, J., and Chen, S. Fastuniq: a fast de novo duplicates removal tool for paired short reads. *PloS one 7*, 12 (2012), e52249.

[197] Xu, Z., and Taylor, J. A. Snpinfo: integrating gwas and candidate gene information into functional snp selection for genetic association studies. *Nucl Acids Res 37*, Suppl 2 (July 2009), W600–W605.

[198] Yang, C.-H., Cheng, Y.-H., Chuang, L.-Y., and Chang, H.-W. Snp-flankplus: Snp id-centric retrieval for snp flanking sequences. *Bioinformation 3*, 4 (2008), 147.

[199] Yang, X., Liu, D., Liu, F., Wu, J., Zou, J., Xiao, X., Zhao, F., and Zhu, B. Htqc: a fast quality control toolkit for illumina sequencing data. *BMC bioinformatics 14*, 1 (2013), 33.

[200] Ye, K., Schulz, M. H., Long, Q., Apweiler, R., and Ning, Z. Pindel: a pattern growth approach to detect break points of large deletions and medium sized insertions from paired-end short reads. *Bioinformatics 25*, 21 (2009), 2865–2871.

[201] Yoon, S., Xuan, Z., Makarov, V., Ye, K., and Sebat, J. Sensitive and accurate detection of copy number variants using read depth of coverage. *Genome research 19*, 9 (2009), 1586–1592.

[202] Yuan, H. Y., Chiou, J. J., Tseng, W. H., Liu, C. H., Liu, C. K., Lin, Y. J., Wang, H. H., Yao, A., Chen, Y. T., and Hsu, C. N. Fastsnp: an always up-to-date and extendable service for snp function analysis and prioritization. *Nucl Acids Res 34*, Suppl 2 (July 2006), W635–W641.

[203] Yung, L. S., Yang, C., Wan, X., and Yu, W. Gboost: a gpu-based tool for detecting gene–gene interactions in genome–wide case control studies. *Bioinformatics 27*, 9 (2011), 1309–1310.

[204] Zhang, Z., Schwartz, S., Wagner, L., and Miller, W. A greedy algorithm for aligning dna sequences. *J Comput Biol 7*, 1-2 (2000), 203–214.

[205] Zhang, Z. D., Du, J., Lam, H., Abyzov, A., Urban, A. E., Snyder, M., and Gerstein, M. Identification of genomic indels and structural variations using split reads. *BMC genomics 12*, 1 (2011), 375.

[206] Zhao, K., and Chu, X. G-blastn: Accelerating nucleotide alignment by graphics processors. *Bioinformatics 30*, Issue 10 (January 2014), 1384–1391.

[207] Zhao, K., and Chu, X. G-blastn: accelerating nucleotide alignment by graphics processors. *Bioinformatics* (2014), btu047.

[208] Zhao, M., Wang, Q., Wang, Q., Jia, P., and Zhao, Z. Computational tools for copy number variation (cnv) detection using next-generation sequencing data: features and perspectives. *BMC bioinformatics 14*, Suppl 11 (2013), S1.

[209] Zvaifler, N. J., Burger, J. A., Marinova-Mutafchieva, L., Taylor, P., and Maini, R. N. Mesenchymal cells, stromal derived factor-1 and rheumatoid arthritis [abstract]. *Arthritis Rheum 42* (1999), s250.