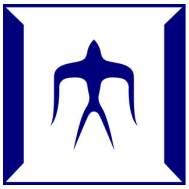
論文 / 著書情報
Article / Book Information

題目(和文)	大規模並列ヘテロジニアス環境におけるメモリ構造を考慮したソーティングおよび配列アラインメントの最適化
Title(English)	Memory-Conscious Optimizations for Sorting and Sequence Alignment for Massively Parallel Heterogeneous Architectures
著者(和文)	DROZD Aleksandr
Author(English)	Aleksandr Drozd
出典(和文)	学位:博士(理学), 学位授与機関:東京工業大学, 報告番号:甲第9420号, 授与年月日:2014年3月26日, 学位の種別:課程博士, 審査員:松岡　聡,遠藤　敏夫,山下　真,脇田　建,渡辺　治
Citation(English)	Degree:Doctor (Science), Conferring organization: Tokyo Institute of Technology, Report number:甲第9420号, Conferred date:2014/3/26, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Memory-Conscious Optimizations for Sorting and Sequence Alignment for Massively Parallel Heterogeneous Architectures



Aleksandr Drozd

Graduate School of Information Science and Engineering

Tokyo Institute of Technology

A thesis submitted for the degree of

PhilosophiæDoctor (PhD)

2014 February

Academic Advisor: Satoshi Matsuoka

Reviewer: Toshio Endo

Reviewer: Ken Wakita

Reviewer: Osamu Watanabe

Reviewer: Makoto Yamashita

Day of the defense: January 6th, 2014

Abstract

This work addresses the issue of improving performance of data-intensive algorithms on modern computing architectures. Two case studies are presented: sorting and sequence alignment.

For sorting we developed CPU and GPU implementations of most-significant digit radix sort algorithm using different parallelization strategies on different execution stages to optimize the use of system resources and workload balance. To overcome the limitations of PCIe bus bandwidth in the GPU version we used communication-reducing strategy. Our solution achieves sorting rates up to $6 * 10^7$ keys per second sorting throughput with good scalability.

For the sequence alignment we adapted Burrows Wheeler transform based index to reduce overall memory footprint on GPUs. A mathematical model of computation and communication costs was developed to optimize memory partitioning for index and queries. Performance evaluation shows more than ten-fold performance gain per device.

Acknowledgements

I would like to give my most sincere appretiation to people who made this dissertation possible. My advisors, colleagues, dear friends - well, these are largely intersecting sets.

Спасибо!

ありがとうございます！

Contents

List of Figures

LIST OF FIGURES

List of Tables

Glossary

DNA Molecule that encodes the genetic instructions used in the development and functioning of all known living organisms and many viruses

Gene The basic physical unit of heredity; a linear sequence of nucleotides along a segment of DNA that provides the coded instructions for synthesis of RNA, which, when translated into protein, leads to the expression of hereditary character.

Genome Entirety of an organism's hereditary information.

GPU Originally specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display.

Nucleotides Organic molecules that serve as the monomers, or subunits, of nucleic acids like DNA and RNA.

Sequence Alignment a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences.

1

Introduction

1.1 Computational Science and HPC

Invention of computers revolutionized science and technology. Through history people invented new instruments which allowed them to work faster and make new discoveries which, in their turn, led to creation of even better instruments. As of now, the top of this pyramid is the computer. Since the appearance of the first electronic computing machines they have been used for calculation-intensive scientific simulations (1).

Modern computers have come a long way; currently even the simplest home systems have more computational power than the computers used in the first space explorations. Computers have already enabled people to create more efficient vehicles, new ways of communication, new life-saving drugs, etc. The rate of progress is increasing, and it requires increasingly precise simulations, which requires more and more powerful computers.

Another factor that came into play with computerization of society is that people are now generates huge amounts of information. We have exabytes of data from World Wide Web, sensor networks, hight-throughput DNA sequencing machines, and the Large Hadron Collider, to name a few. Where do we get the computational power to process all this data?

Originally the microprocessor technology was pampering us with reasonable performance improvement due to frequency scaling and more powerful instruction sets. In 1965 Intel co-founder Gordon E. Moore described the trend of doubling of the number of transistors on integrated circuits approximately every two years(2). Over the his-

tory of computing hardware this trend proved to be true so far, though even G. Moore himself admits that there are fundamental barriers in miniaturization levels (since we can not go beyond the atomic level), and the problems like heat dissipation efficiency prevent us from getting even to the atomic level.

Processor vendors are already out of room for driving clock speeds and straight-line instruction throughput higher, but the overall performance of computing systems continues to grow steadily. This is achieved via parallelizm (3), a form of computation coupled with hardware architecture capable of carrying out many calculations simultaneously. Typically large problems can be divided into smaller ones, which are then solved in parallel.

1.2 Motivation

Supercomputers are the state of the art in high-performance computing. Modern supercomputers employ immense parallelizm. First supercomputers in 1960-1970s used only a few processors, but already in the 1990s we had machines with thousands of processors. By the end of the 20th century massively parallel supercomputers featured tens of thousands of processors.(4).

The computational power grows in Moore's law-like fashion. Today the most powerful computers in the world achieved petaFLOP-level performance, performing quadrillions of operations per second. As of November 2013, China's Tianhe-2 supercomputer is the fastest in the world at 33.86 petaFLOPS.

It is the supercomputers that allow us to perform numerical simulation of experiments we would never be able to conduct in real life and to analyse enormous amounts of data that human brain could never process on its own. Allowing us to do more with less, computers created a new breed of "computational science".

Figure 1.1 shows current and projected performance of systems in Top 500 list. We are talking now about achieving symbolic barrier of exa-scale performance in several years to come.

However, all this computational power is not easy to use. Traditional processors provide performance increase with little effort from computer programmers, but supercomputers are posing lots of problems. It is very difficult to program efficiently for supercomputers (5). Many existing algorithms and data structures are not scalable for

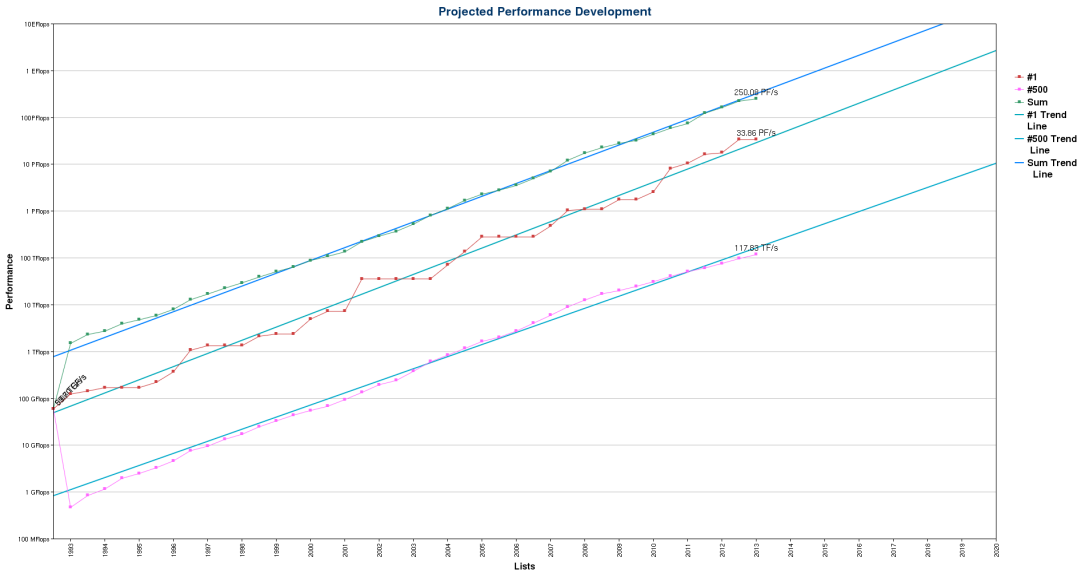


Figure 1.1: Projected Supercomputers Performance

large number of nodes available on Petascale HPC systems. So, it is often necessary to modify an existing algorithm or develop new ones, which would efficiently exploit high performance of a Petascale systems;

- Inefficient communications patterns can become a bottleneck for increasing performance.
- Applications become increasingly complicated, combining a variety of languages, libraries, programming models, data structures and algorithms. Therefore we need techniques which would provide code flexibility and portability.
- Load imbalance becomes increasingly critical when hundreds of thousands of threads are kept waiting to get the result form just one thread.

The architecture of supercomputers becomes more and more heterogeneous. If we take a look at the head of the Top 500 list we would find that most of the worlds fastest supercomputers are based on hybrid (GPU+CPU) architecture. Tsubame 2.5, the supercomputer installed in Tokyo Tech, is featuring such a hybrid architecture. It was upgraded from TSUBAME2, a production supercomputer operated by Global Scientific

Information and Computing Center (GSIC), Tokyo Institute of Technology in cooperation with industrial partners, including NEC, HP, NVIDIA, Microsoft, and Voltaire. Since fall 2010 it has been one of the fastest and power-efficient supercomputers in the world. Tsubame 2.5 boasts 2.4 PFlops peak performance achieved by aggressive GPU acceleration, which allows scientists to enjoy faster computing than ever.

It is also worth mentioning that in terms of energy efficiency GPUS are considered to be one of the greenest architectures up to date (6). Due to the fact that hybrid systems show high performance, performing parallel tasks with comparatively little power, we believe that heterogeneous architecture with GPUs is a strong candidate for future Exascale systems.

In fact, all top 10 systems of November 2013 Green 500 list are GPU-based. The list is headed by Tsubame KFC system - the prototype of Tsubame 3.0 supercomputer to be installed in Tokyo Institute of Technology in 2015.

However, hybrid architectures introduce new difficulties and challenges into programming for supercomputers. GPU accelerators add another level to the memory hierarchy and utilize a different execution model. Specialized programming environments are needed to use GPUs. Moreover, programming approaches for GPU substantially differ from those for traditional architectures, which makes research in this area highly relevant for many applied fields.

1.3 Problem Statement

To sum up, mankind has now accumulated unprecedented amounts of data which needs to be processed in various scientific and applied tasks. In parallel to the growth of Big Data, we witness the growth of massive parallel heterogeneous architectures. However, their usage in data-intensive areas requires major adaptations in many core algorithms.

One of the new knowledge areas that particularly needs supercomputing is computational biology that deals with mathematical modeling and computational simulations of biological systems. Computational genomics is one of the particularly data-intensive subfields; one human genome (7) takes gigabytes of data, and now there are projects that collect big banks of genomes (8)

One of the fundamental routines in computational genomics is sequence alignment - arranging the sequences of DNA, RNA, or protein to identify regions of similarity

that may be a consequence of functional, structural, or evolutionary relationships between the sequences.(9). This work addresses the issue of efficient sequence alignment on GPU-based massively parallel heterogeneous architectures. In this work, we adapt Burrows Wheeler transform based index to reduce overall memory footprint on GPUs and also develop a mathematical model of computation and communication costs optimize memory partitioning for index and queries.

Another challenging area in supercomputing is the optimization of sorting algorithms, which is particularly important in the context of Big Data. Most particularly, the sorting of strings has received less attention than sorting of numerical data, and this work addresses this gap. We analyse candidate sorting algorithms with reference to their useability for particular data types and architectures and efficient data movement across memory hierarchies. We then focus on the most-significant digit radix sort algorithm which allows for sorting of variable length keys.

1.4 Contributions

For sorting we developed CPU and GPU implementations of most-significant digit radix sort algorithm using different parallelization strategies on different execution stages to optimize the use of system resources and workload balance. To overcome the limitations of PCIe bus bandwidth in the GPU version wef used communication-reducing strategy. Our solution achieves sorting rates up to $6x10^7$ keys per second sorting throughput with good scalability.

We develop CPU and GPU implementations of , optimizing the use of system resources and workload balance by using different parallelization strategies on different execution stages.

For the sequence alignment we adapted Burrows Wheeler transform-based index to reduce overall memory footprint on GPUs. We introduced efficient CUDA implementation of pattern matching algorithm and verified its fitness to GPU architecture. We exploited the interplay between the Burrows Wheeler Transform and Suffix Arrays to minimize data transfers throught the PCIe bus. Finally, we developed a a mathematical model of computation and communication costs to optimize memory partitioning for index and queries. Performance evaluation shows more than ten-fold performance gain per device.

1.5 Thesis Outline

Before describing the main contributions of this thesis, we provide some background information HPC and parallel programming, particularly on CUDA and GPU architecture, overview known algorithm optimization strategies, and provide some examples of their successful implementations.

Chapter 3 addresses the problem of sequence alignment. We start by providing a general introduction to the problem domain of computational genomics and a formal definition of the problem. We describe in detail our data structure and analyse the chosen algorithm in terms of its efficiency on GPU architecture. Particular attention is given to the problem of reduction of costs of moving data between host and GPU memories.

Chapter 4 presents the classification of sorting algorithms with respect to the data types and fitness of the algorithms to particular architectures. We focus on the most-significant digit radix sort, one of the few algorithms that are applicable to sorting of string data. This chapter also discusses our performance model: load balancing, scalability, and communication expenses.

Chapter 5 concludes this work. It outlines the main findings from implementation experience and performance evaluation of our solutions, some of which are relevant not only to the applied areas under consideration, but also to the supercomputer programming in general. We and also discuss the implications of our findings for computational biology and string data analysis and suggest some directions for future work in optimization of data-intensive algorithms for hybrid architectures.

2

Background

2.1 High Performance Computing

History of HPC goes back to the middle of 20th century. Computers of that time had performance about KFLOPS and were rather sequential in the meaning that they executed one instruction after previous one is completed. In the 70s the new architecture, called RISC or *restricted (reduced) instruction set computer* appeared. Its load/store approach allowed designers to increase the size of the register set and increase internal parallelism, i.e. allowed to execute multiple instructions and multiple parts of the instructions simultaneously. That gave rise to performance level around the MFLOPS.

In the 80s new vector computers was developed, which idea was to issue one instruction and have that instruction take be applied on a whole sequence of data. And in the 80s we saw performance level approaching a GFLOPS.

Starting from the 90s we have parallel computing taking place in a rather large way. While the first supercomputers of 1970s had only few processors, in the 90s some machines used thousands of processors, and by the end of the 20th century, massively parallel supercomputers with tens of thousands of off-the-shelf processors were the norm.

Today we are at the PFLOPS point where we have super-scalar, special purpose parallel machines, which can use over 100,000 processors connected by fast connections. Supercomputer sector is the fastest growing one in the world of technology, and it is currently worth some 25Billion USD(10). Japan, China and the USA are currently

holding the lead in the world of Supercomputing, but there is an acute shortage of the skills and applications required to make the most of these machines.

2.1.1 Programming Supercomputers

Programming for parallel architectures is more difficult than traditional sequential way. First of all, there are much more things to take care off. Programmer has not only to express the algorithm, but move data from the point where it is originated to the point, where it is used.

Process interaction can roughly be classified as shared memory or message passing models. In the first model processes can see global address space where they can read and write simultaneously. Synchronization mechanisms like locks or semaphores might be required to assure right order of memory accesses and correctness of the program.

In a message passing model, parallel tasks exchange data by sending messages to one another. These communications can be asynchronous or synchronous. In this case processes can synchronize by waiting for messages. Both approaches are prone to dead-lock and race conditions - making parallel programming more difficult.

Task other important concepts are Task parallelism and Data parallelism. A task-parallel model focuses on processes, which can be behaviourally distinct, which emphasizes the need for communication. It is usually classified as MIMD/MPMD or MISD.

A data-parallel model (usually classified as SIMD/SPMD) implies performing a set of tasks or operations on different elements of partitions of data set. Data is typically regularly structured in an array or similar structure and the operations performed in the data are independent for separate elements. In a shared memory system, the data will be accessible to all, but in a distributed-memory system it will divided between memories and worked on locally.

There are now numerous parallel and concurrent programming languages, libraries, APIs, and parallel programming models. We uses OpenMP for programming shared-memory intra-node parts of algorithms and MPI for communication between nodes.

OpenMP (Open Multi-Processing) is, perhaps, the most wide-spread technology for programming shared memory systems. It extends C, C++, and Fortran languages with so called pragmas, which are supported by most of the modern compilers and do not disrupt program execution if the compiler happens to not support OpenMP standard. Today OpenMP works on a wide variety of processor architectures, including

accelerators. It works under many operating systems, including AIX, Solaris, HP-UX, GNU/Linux, Mac OS X, and Windows platforms. OpenMP also contains library routines, and environment variables that influence run-time behavior. (11).

There are other libraries for implementing parallel algorithms for shared memory processors: Threading Building Blocks, Massive Threads, QThreads etc. Most of them support light-weight threading model - unlike operating system heavy-weight threads they share same hardware context (registers etc) and thus context switching can be executed faster.

The fundamental primitive in this libraries/languages and corresponding runtime environments is a task - a sequence of instructions that can be executed independently by one of the available "worker" threads. New tasks are created in the task queue usually local for the current threads and then can be executed by any available worker thread, this logic is control by the runtime component called scheduler.

For the inter-node communication typically MPI (Message Passing Interface) is used. It is standardized and portable message-passing system for a wide variety of parallel computers. It originates in 1991 from an effort of a group of researchers from academia and industry and eventually became "de-facto" standard fort programming distributed-memory supercomputers (12).

MPI includes language-independent communications protocol and defines the syntax for the of library routines providing MPI API. It can be used to program parallel systems of various architecture and network topology and is available for Fortran, C and $C++$ programming languages.

Now there are many implementation of MPI available, most of them mature and well-tested, suited for industrial usage. Many of these implementation are free or in the public domain. MPI is widely in software industry for the development of scalable and portable large-scale parallel applications.

2.1.2 Application Performance Challenges

Number of processors, total FLOPS and network bandwidth does not guarantee high performance for a particular application. In practice many applications run on supercomputers utilize only a small share of available computational performance. This difficulty is signified by the existence of Gordon Bell prize that is awarded every year to the high-performance computing applications that achieve.

Through the history of HPC we see how the paradigms are shifting on building and programming supercomputers (13). For example, originally power consumption was not considered a problem, but transistors were expensive. Now we talk about *power wall* and struggling to build more power-efficient systems and adopt appropriate programming models.

With fewer cores on chip / chips in the system in the past occasional hardware errors were not a problem, but now the drop in feature size and increase in the number of cores dramatically increase error rates. It created a need to consider fault-tolerance as another great challenge, in the context of design if hardware, software infrastructure and user software itself.

Performance improvements does not go with equally steady pace for different hardware components of the system. For example IO bandwidth improves by at least the square of the improvement of the latency. Single core performance almost stopped to grow, so did the clock rates.

Individual nodes of a cluster are typically multi-core / multi-processor shared memory systems programmed with OpenMP and programmers sometimes adopt simplified approach to such systems as to symmetric multiprocessors. But in fact NUMA effects are taking place in almost all contemporary microprocessors because of core-local caches etc. NUMA, or Non-uniform memory access is a computer memory design when the memory access time depends on the memory location relative to the processor. Under NUMA, a processor can access its own local memory faster than non-local memory (memory local to another processor or memory shared between processors). The benefits of NUMA are limited to particular workloads, notably on servers where the data are often associated strongly with certain tasks or users.

Instruction-level parallelism, which used to be revealed by compiler without much effort from a programmer - now yields diminishing return (*ILP wall*).

As a result we can see the change of perception of what is considered "successful parallelization" - any speedup on a parallel system is often considered a success.

Another shift is happening in a sense of what application require supercomputer performance. While classical "supercomputer" tasks like dense end sparse linear algebra problems, spectral methods, N-body simulations, finite-element methods etc, are still very important and require a lot of FLOPS, other tasks from the areas of combinational logic, graph algorithm, machine learning - are also becoming "popular". Many of this

tasks rely on integer rather then floating-point computations and a very communication-intensive.

The demand for such computation is coming particularly from the growing abundance of data generated collected bu modern IT infrastructure.

2.2 GPU Computing

If we take a look at the head of the Top 500 list we would find that most of the worlds fastest supercomputers are based on hybrid architecture. And from the application side there is a corresponding rise in use of accelerators (14). In this work we mainly focused on GPUs.

Graphics Processing Units (GPUs) are high-performance many-core processors capable of very high computation and data throughput. Initially they were designed to speed up the rendering of complex graphics, e.g. for video game applications, where the same algorithm is applied to many elements like polygons. This make GPUs design originally more parallel, than traditional CPUs. First GPUs were difficult to program, but todays GPUs are general-purpose parallel processors with support for accessible programming interfaces and industry-standard languages like C and are actively used for the various scientific applications.

The use of graphics processors for general purpose computation (GPGPU) is particularly attractive as their performance is improving faster than that of typical CPUs. Unlike traditional CPUs, which are designed to make serial tasks run as quickly as possible, GPUs execute multiple parallel tasks in a power-efficient manner. Therefore GPU computing becoming more widely used in demanding customer applications and high performance computing.

Recently GPUs have become very popular accelerators for HPC systems. Among the world's fastest systems many now are GPU-accelerated. One example petascale-class supercomputer based in hybrid architecture is the Tokyo Techs TSUBAME 2.5 with Nvidia Fermi GPUs. It was recently upgraded from TSUBAME 2, a production supercomputer operated by Global Scientific Information and Computing Center (GSIC), Tokyo Institute of Technology in cooperation with industrial partners, including NEC, HP, NVIDIA, Microsoft, and Voltaire. Since fall 2010 it has been one of the fastest and power-efficient supercomputers in the world. TSUBAME 2.5 boasts

2.4 PFlops peak performance achieved by aggressive GPU acceleration, which allows scientists to enjoy faster computing than ever.

The GPUs have increased the performance of several benchmarks (e.g. the sparse finite difference Himeno benchmark), multiple libraries and many applications.

2.2.1 GPGPU History

First GPUs emerged in 1997, when Nvidia released the RIVA 128 3D single chip graphics accelerator for games and 3D visualization applications (15)).

The only way to program devices of that time was to use Microsoft Direct3D and OpenGL. GeForce 256 was introduced in 1999 and it was the first GPU, a single-chip graphic processor, which contained a configurable 32-bit floating-point vertex transform and lighting processor, and a configurable integer pixel-fragment pipeline, programmed with OpenGL and Microsoft DirectX 7 APIs. In 2001 the GeForce 3 GPUs became more flexible and programmable.

The new GeForce FX and GeForce 6800 were highly multithreaded and used Cg programs, DX9 and OpenGL. The Cg language was the first language for programming GPUs, which provided scalable programming models for the programmable floating-point vertex and pixel-fragment processors of GeForce FX, GeForce 6800, and subsequent GPUs. Developing Cg language allowed to use GPUs to compute scientific simulations and other general-purpose GPU (GPGPU) computations. This GPGPU computing programs showed significant acceleration and high performance, but were very complicated to write. Programming general-purpose computation with Cg, however, was very not straight-forward task.

In 2006 NVIDIA introduced the GeForce 8800, which were the first unified graphics and computing architecture. In addition to using Dx10 and OpenGL the GeForce 8800 were programmable in C with the CUDA parallel computing model. Starting form the 2007, when NVIDIA introduced the Tesla C870, D870 and S870 GPU cards, it became possible to build personal supercomputers by adding multiple GPU cards. In 2009 the third-generation Fermi GPU computer architecture, which significantly increased double precession performance, was introduced by NVIDIA. From that time the programmer can use not only C, but C++, Fortran, OpenCL, and DirectCompute languages to program the GPUs.

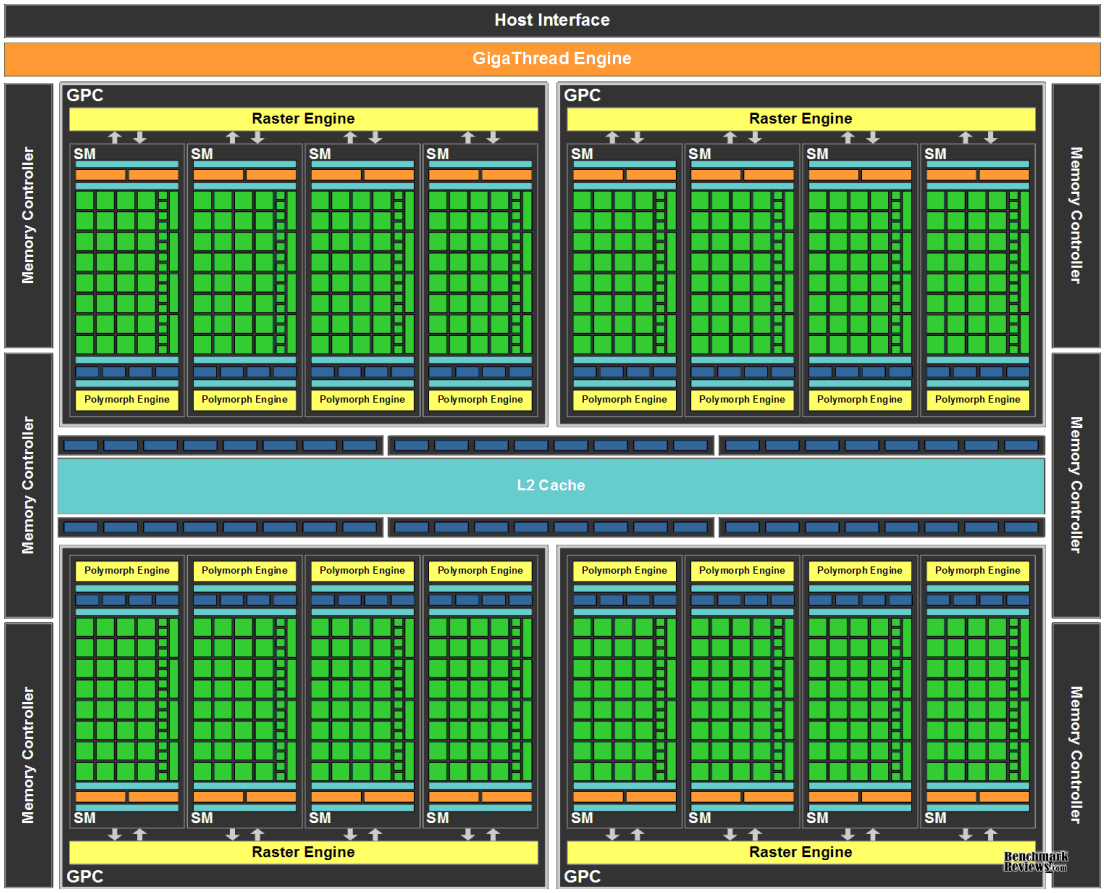


Figure 2.1: CUDA architecture

2.2.2 GPU Architecture

GPUs are good massively parallel computation thanks to its several hundreds of cores. While classic CPU architecture is optimized for low-latency access to cached data sets, the GPU architecture is optimized for high data parallel throughput computation. The GPU cores are all managed by a thread manager, that can spawn and manage tens of thousands of threads simultaneously. Figure 2.1 shows the Fermi GPU Architecture. The cores of the GPU (512 for the Fermi) have one Arithmetic Logic Units (ALU) and one Floating Point Unit (FPU). A group of 32 cores plus a shared memory block (48/16KB) makes a Streaming Multiprocessor (SM). Four of these SMs plus a Raster Engine make a Graphic Processor Cluster (GPC). Finally, the Fermi GPU is composed by 4 GPCs.

GPU devices has own GDDR memory which is physically placed on the same extension card and typically is faster then main RAM, though smaller in size.

New chips based on Kepler architecture comprise more then 7 billion transistors and provide over a teraFLOP of double precision throughput and even better power efficiency. Each of the Kepler GK110 SMX units feature 192 CUDA cores, and each core has fully pipelined floatingpoint and integer arithmetic logic units. Kepler retains the full IEEE 7542008 compliant single and doubleprecision arithmetic introduced in Fermi, including the fused multiplyadd (FMA) operation.

2.2.3 Programming GPUs

Currently there are several different libraries and tools, which allow to make GPU development easier and more productive. This include CUDA C/C++ (16), (CUDA for Fortran is also available from the Portland Group (PGI) CUDA, OpenCL (17), OpenHMP (18) and OpenACC (19).

OpenACC is a programming standard for parallel computing developed by Cray, CAPS, Nvidia and PGI to simplify parallel programming of heterogeneous CPU/GPU systems. It resembles OpenMP in a way that programmer can annotate source code to identify the areas that should be accelerated using compiler directives and additional functions.

There is an ongoing effort to merge OpenACC standard into OpenMP specification to create a common specification which extends OpenMP to support accelerators in a future release of OpenMP. These efforts resulted in a technical report[5] for comment and discussion timed to include the annual Supercomputing Conference (November 2012, Salt Lake City) and to address non-Nvidia accelerator support with input from hardware vendors who participate in OpenMP.

Open Computing Language (OpenCL) is a language based on a C99 standard for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays and other processors. OpenCL itself is used for writing *kernels* -functions that execute on OpenCL devices, and kernels are called from a user program written in C or other supported language with provided application programming interfaces (APIs).

OpenCL is an open standard maintained by the non-profit technology consortium Khronos Group. It has been adopted by Apple, Intel, Qualcomm, Advanced Micro Devices (AMD), Nvidia, Altera, Samsung, Vivante and ARM Holdings. Academic researchers have investigated automatically compiling OpenCL programs into application-specific processors running on FPGAs, and commercial FPGA vendors are developing tools to translate OpenCL to run on their FPGA devices. OpenCL can also be used as an intermediate language for directives-based programming such as OpenACC.

These two standards aim for the support of a wide range of accelerators and processor architecture and as a trade-off they sacrifice some platform-specific nuances. We use CUDA for programming GPUs as it is mature and widely-used industry standard and, compared to other GPU programming frameworks, CUDA allows for the most detailed control of micro-architectural details of the program execution.

2.2.4 Programming with CUDA

CUDA parallel programming model enables NVIDIA GPUs to execute programs, written in C, C++, Fortran and other languages, it extends the language with a set of abstractions for expressing parallelism. This lets the developer do not change the language of the sequential code, and make a highly scalable code, by using CUDA extensions, which significantly reduce complexity and the development time.

A CUDA program is organized into a host program, consisting of one or more sequential threads running on the host CPU, and one or more parallel kernels that are suitable for execution on a parallel processing device like the GPU (20). Kernel executes a scalar sequential program on a set of parallel threads. The programmer organizes these threads into a grid of thread blocks (Fig. 2.2).

The threads of a single thread block are allowed to synchronize with each other via barriers and have access to a high-speed, per-block shared on-chip memory for interthread communication (Fig. 2.3). Threads from different blocks in the same grid can coordinate only via operations in a shared global memory space visible to all threads. CUDA requires thread blocks be independent, meaning that a kernel must execute correctly no matter the order in which blocks are run, even if all blocks are executed sequentially in arbitrary order without preemption. This restriction on the dependencies between blocks of a kernel provides scalability. It also implies that

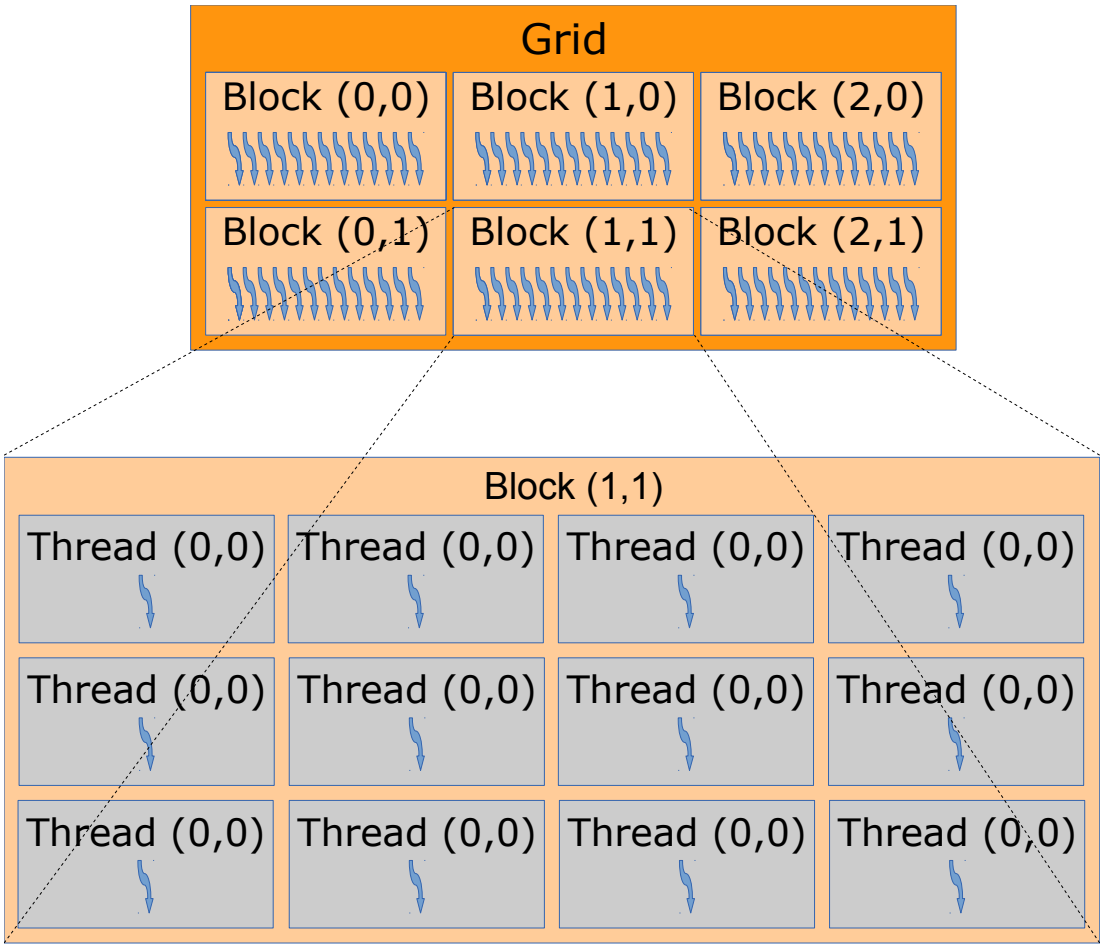


Figure 2.2: CUDA Execution Model

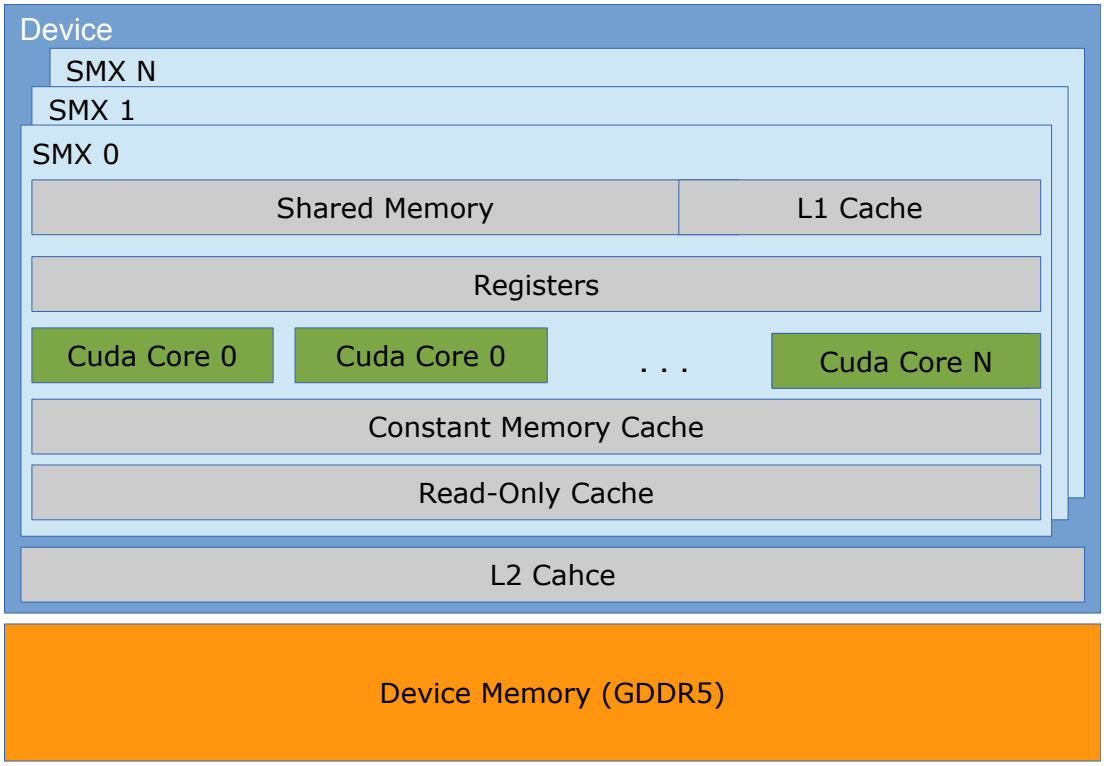


Figure 2.3: CUDA Memory Hierarchy

the need for global communication or synchronization amongst threads is the main consideration in decomposing parallel work into separate kernels. The details of the CUDA programming model are available in NVIDIAs CUDA Programming Guide (16).

The management of the data through the memory hierarchy of the GPU is a very important element to optimize GPU applications. Finally, when the kernel execution is finished, the data is copied back from the device to the host. Fermi GPUs not only achieve good performance in single-precision but also in double-precision and they include ECC to tolerate bit-flip errors.

Some of the frequently used GPU primitives are available as Thrust library. It is based on $C++$ Standard Template Library (STL). For some simple applications Thrust allows programmers to implement high performance parallel applications with minimal programming effort through a high-level interface that is fully interoperable with CUDA C. Thrust provides a collection of data parallel primitives such as scan, sort, and reduce, which can be composed together to implement more complex algorithms.

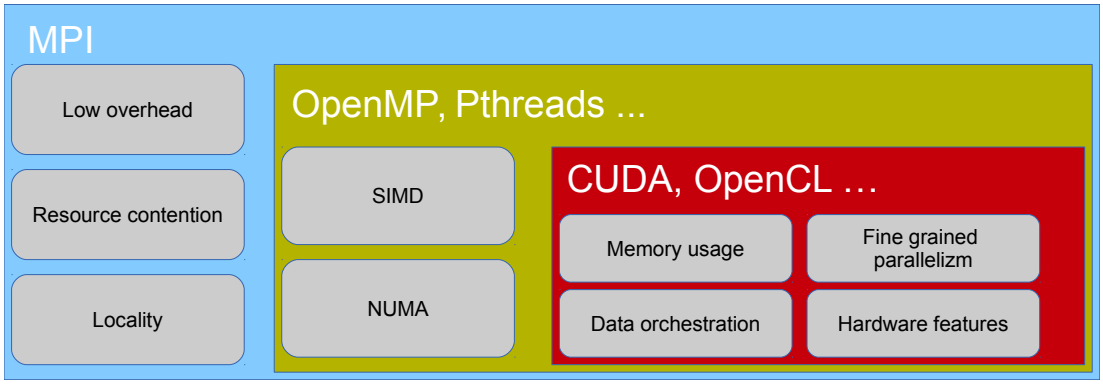


Figure 2.4: Stack of Parallel Programming Technologies

Recent Kepler architecture and corresponding updates in CUDA allowed for Dynamic Parallelism which simplifies GPU programming by allowing programmers to easily accelerate all parallel nested loops resulting in a GPU dynamically spawning new threads on its own without going back to the CPU. Another new feature of CUDA programming model was so called Hyper-Q, which allows multiple CPU cores to simultaneously utilize a single Kepler GPU, dramatically advancing programmability and efficiency.

2.2.5 Challenges of GPU Programming

Hybrid GPU-accelerated architectures introduce new level of difficulty to the parallel programming. First of the stack of technologies required to program for these architectures gets one extra level. Typically programmers have to use MPI to distribute work among cluster nodes and to exchange messages between processes. Then they use OpenMP or other threading language/library to utilize multiple cores of the CPU or multiple CPUs on the motherboard. Then they have to use GPU-specific language for programming the accelerator: CUDA, OpenCL, OpenACC or any other alternative (fig. 2.4).

This also introduces additional complexity to the code, makes it harder to write, to read, to maintain and to debug. But the problem is not only how to write a working error-free program, but how to make it efficient in therms of using all the potential GPU performance.

The first problem is extra levels of memory hierarchy introduced by GPUs. The data

lying in the DRAM, main memory accessed by the CPU, is not visible by the device. Programmer has to explicitly transfer this data and it takes time as the transfer happens through the PCIx bus. When the computation per unit of data is relatively big - the transfer does not cause noticeable performance degradation, but if compute/data move ration is not very high - GPU's computational power might be of little use. Moreover, when there are multiple GPUs in the system PCIe bandwidth is divided between these devices.

GPU memory, in turn, is not monolithic (fig. 2.3). There are typically several chips of fast GDDR memory on the board containing so called device memory. From the programmer's point of view this is so called *global memory* - the one accessible to all symmetric multiprocessors of GPU. This is the only memory which can be accessed from the host, largest in size and slowest in access, however typically GDDR memory offers higher throughput than DRAM. More then that, the throughput depends in access pattern - if threads of the warp are accessing adjacent memory words, this accesses can be performed simultaneously, effectively multiplying throughput. On modern GPU devices size of global memory reaches 12 gigabytes, which is still several times smaller than typically available to CPU.

Then on the GPU chip itself there is level two (L2) cache. This cache is also shared between all symmetric multiprocessors, is faster than the global memory but much smaller in size, about several megabytes.

Every SMX also has it's own several levels of memory hierarchy private to this SMX. The *local* memory and L1 cache are placed on the same physical memory close the SMX and are visible to the threads of the same block. Local memory and L2 cache are faster then L2 cache and respectively than the global memory, but are much smaller in size, 64 kilobytes on recent GPUs.

In addition to L2 and L1 caches GPUs have constant memory and read-only cache. Like many of the nuances of GPU architecture these are inherited from previous tailoring for computer-graphics computation where textures were accessed read-only and with 2-dimensional addressing. At the dawn of GPGPU computing programmers had to literally represent their data as images to get benefits from GPU texture subsystems. Now programmer can just specify that certain block of data is accessed read-only and compiler will translate it into corresponding low-level instructions.

Finally there are registers, the fastest memory on GPU. Registers are local for the threads and typically used for storing variables from cuda kernels. If there are not enough registers in the system - *register spilling* happens and negatively affects performance.

As we can see there are many things just from the memory hierarchy are which can help or prevent achieving high GPU performance. Some algorithms with high computation/memory access ratio and algorithms with regular memory access patterns are obviously "GPU-friendly". Some algorithms can me implemented in a form that can meet GPU peculiarities. Some algorithms have very irregular memory access patterns and perform it frequently - these applications are difficult to efficiently implement for GPU.

The other potential cause of problems is the execution model itself. As we discussed before, modern GPUs evolved from those GPU for computer graphics with fixed-function pipelines. Computer graphics applications were not so sensitive to latency of individual instruction execution but required hight throughput. This throughput was achieved by massive parallelism, while the individual core is not very efficient compared to CPU core. So if there is any sequential part in GPU code - the negative affect on performance would be orders of magnitude bigger compared with CPU, where all programs typically have some sequential parts.

Parallel threads on GPUs are executed in *warps* of 32 threads. Each threads in the warp executes same instruction, so some researchers say that "many CUDA core" mentioned in NVidia's white papers are virtually several SMX cores with 32-degree wide SIMD instructions. If there is branching operator in the thread and one thread of the warp executes one branch another threads executes second branch - than both threads in fact would iterate over the instructions of these two branches. This is called *branch divergence* and can dramatically decrease GPU performance. In worst case, effectively lose factor 32x in performance if one thread needs expensive branch, while rest do nothing.

It is noteworthy that there are some gradual improvements in GPU architectures which are done to make GPUs less "picky" for the applications which can be run on them. We see that GPU memory access can now be partially coalescent - i.e. if only some threads of the warp are addressing adjacent words - then these threads will enjoy combined load or store operation while other threads will issue individual

memory access instructions. Overall global memory size is increasing from one device to another.

Next generations of accelerators are predicted to have unified virtual memory which will allow to avoid explicit data transfers between host and device. Stacked DRAM promises an order of magnitude higher memory throughput and integrating general-purpose core into the device will allow for control logic to run on the same device as the computational kernels.

Still GPU programming is extremely challenging and while new architectural features will surely make some GPU programming easier in some aspects - they will also introduce new nuances to be aware of when implementing algorithms for future devices.

3

Sequence Alignment

3.1 Introduction to the Problem Domain

3.1.1 Genes and DNA

In most living organisms the genetic instructions used in their development and functioning are stored in the long polymeric molecules of Deoxyribonucleic acid or DNA.

Most DNA molecules are double-stranded helices, consisting of two long biopolymers which in their turn consist of simpler units called nucleotides. Nucleotides are formed with four nucleobased (guanine, adenine, thymine, and cytosine, coded G, A, T, and C, respectively), and a backbone made of alternating sugars (deoxyribose) and phosphate groups (related to phosphoric acid). The nucleobases (G, A, T, C) are attached to the sugars (21).

It is the sequence of the four nucleobases along the backbone that encodes biological information. Under the genetic code, RNA strands are translated to specify the sequence of amino acids within proteins. These RNA strands are initially created using DNA strands as a template in a process called transcription.

The genetic code is the set of rules by which information encoded within genetic material (DNA or mRNA sequences) is translated into proteins by living cells. Biological decoding is accomplished by the ribosome which links amino acids in the order specified by mRNA. Ribosome uses transfer RNA (tRNA) molecules to carry amino acids and to read the mRNA, three nucleotides at a time. The genetic code is highly similar among all organisms and can be expressed in a simple table with 64 entries.

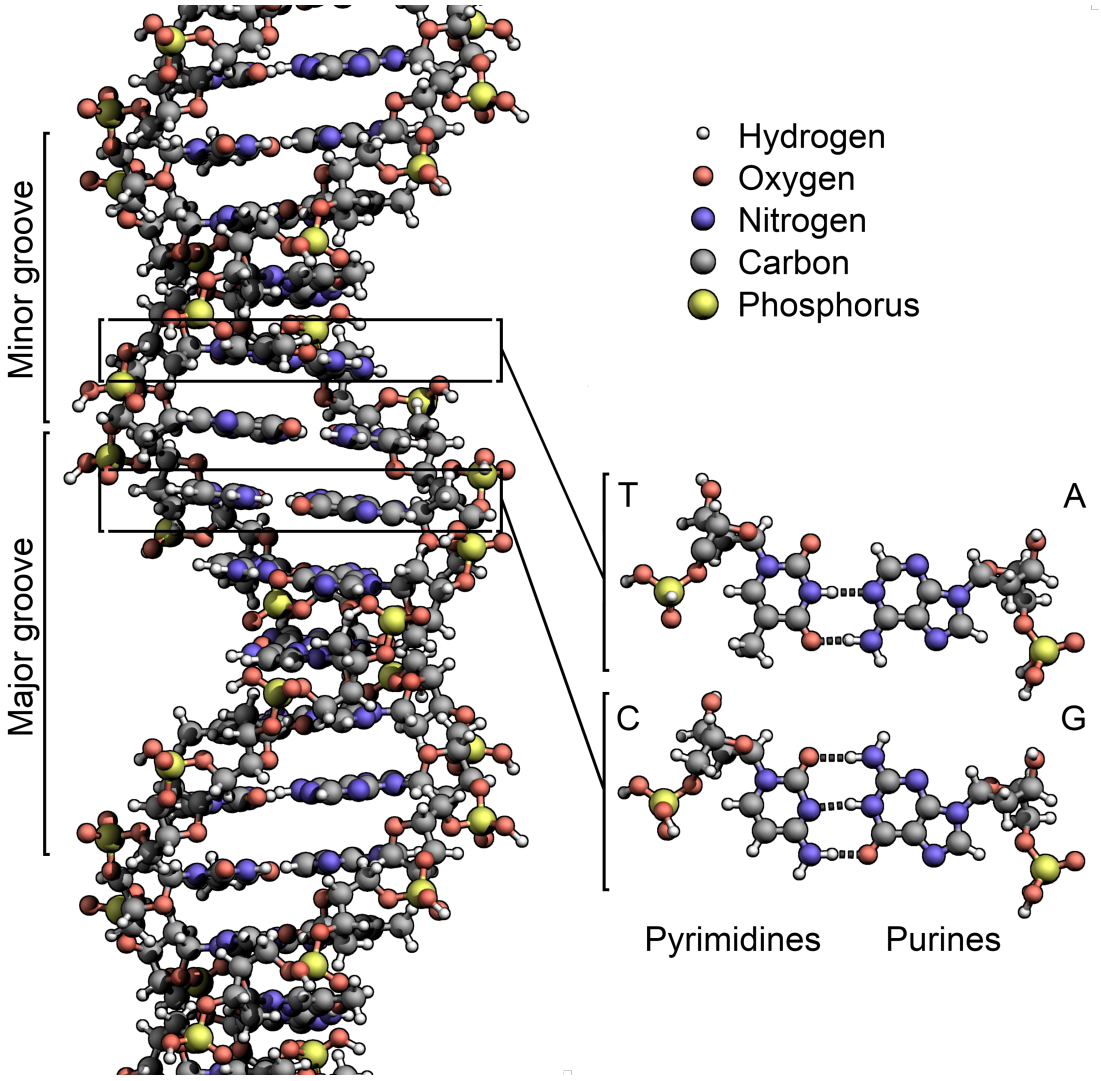


Figure 3.1: DNA Structure

The code defines how sequences of nucleotide triplets, called codons, specify which amino acid will be added next in the process of protein synthesis. With some exceptions, a three-nucleotide codon in a nucleic acid sequence specifies a single amino acid. Because the vast majority of genes are encoded with exactly the same code (see the RNA codon table), this particular code is often referred to as the canonical or standard genetic code, or simply the genetic code, though in fact some variant codes have evolved. For example, protein synthesis in human mitochondria relies on a genetic code that differs from the standard genetic code.

Not all nucleotides in DNA encode proteins, in fact only small share of all nucleotides has this function. This stretches of DNA are called *genes*, other nucleotides (regulatory regions) control *gene expression* - determining when and under what conditions genes are actually transcribed and eventually turned into proteins.

The human genome consists of approximately 3 billion DNA base pairs and is estimated to carry 20,00025,000 protein coding genes. To decipher information contained in a DNA molecule we need to determine the order of nucleotides, the elementary building blocks of DNA that are also called bases. This task is important for many emerging areas of science and medicine.

3.1.2 DNA sequencing

DNA sequencing is the process of determining the exact order of nucleotides within a DNA molecule. This term is used to refer to any method or technology that is used to determine the order of the four bases (adenine, guanine, cytosine, and thymine) in a strand of DNA. The invention of faster DNA sequencing methods has greatly accelerated biological and medical research and discovery.

Bioinformatics involves manipulation, searching, and analysis of biological data, including DNA sequence data. The development of techniques to store and search DNA sequences have posed new tasks for such areas of computer science as string searching algorithms, machine learning and database theory.[144] String searching or matching algorithms which search for an occurrence of a sequence of letters inside a larger sequence of letters were developed for processing specific sequences of nucleotides.

The first advances in reading biological data were made in RNA sequencing. In 1972 Walter Fiers and his coworkers at the University of Ghent (Ghent, Belgium) identified

and published the first complete gene (22). In 1976 they also presented the complete genome of Bacteriophage MS2(23).

Several notable advancements in DNA sequencing were also made in the 1970s. Frederick Sanger developed rapid DNA sequencing methods at the MRC Centre, Cambridge, UK and published a method for "DNA sequencing with chain-terminating inhibitors" in 1977.(24) Walter Gilbert and Allan Maxam at Harvard also developed sequencing methods, including one for "DNA sequencing by chemical degradation".(25, 26) In 1973, Gilbert and Maxam reported the sequencing of 24 basepairs using a method known as wandering-spot analysis.[9] Advances in sequencing were aided by the concurrent development of recombinant DNA technology, allowing DNA samples to be isolated from sources other than viruses.

The first full DNA genome to be sequenced was that of bacteriophage $\phi X174$ in 1977(27). Medical Research Council scientists deciphered the complete DNA sequence of the Epstein-Barr virus in 1984, finding it to be 170 thousand base-pairs long.

Leroy E. Hood's laboratory at the California Institute of Technology announced the first semi-automated DNA sequencing machine in 1986.[citation needed] This was followed by Applied Biosystems' marketing of the first fully automated sequencing machine, the ABI 370, in 1987. By 1990, the U.S. National Institutes of Health (NIH) begun large-scale sequencing trials on Mycoplasma capricolum, Escherichia coli, Caenorhabditis elegans, and Saccharomyces cerevisiae at a cost of US$0.75 per base. Meanwhile, sequencing of human cDNA sequences called expressed sequence tags was started in Craig Venter's lab, in an attempt to capture the coding fraction of the human genome.(28) In 1995, Venter, Hamilton Smith, and his colleagues at The Institute for Genomic Research (TIGR) published the first complete genome of a free-living organism, the bacterium Haemophilus influenzae. The circular chromosome contains 1,830,137 bases and its publication in Science journal(29) marked the first published use of whole-genome shotgun sequencing, eliminating the need for initial mapping efforts. By 2001, shotgun sequencing methods had been used to produce a draft sequence of the human genome.(7, 30)

Several new methods for DNA sequencing were developed in the mid- to late 1990s. In 1996, Pl Nyrn and his student Mostafa Ronaghi at the Royal Institute of Technology in Stockholm published their method of pyrosequencing.(31) A year later, Pascal

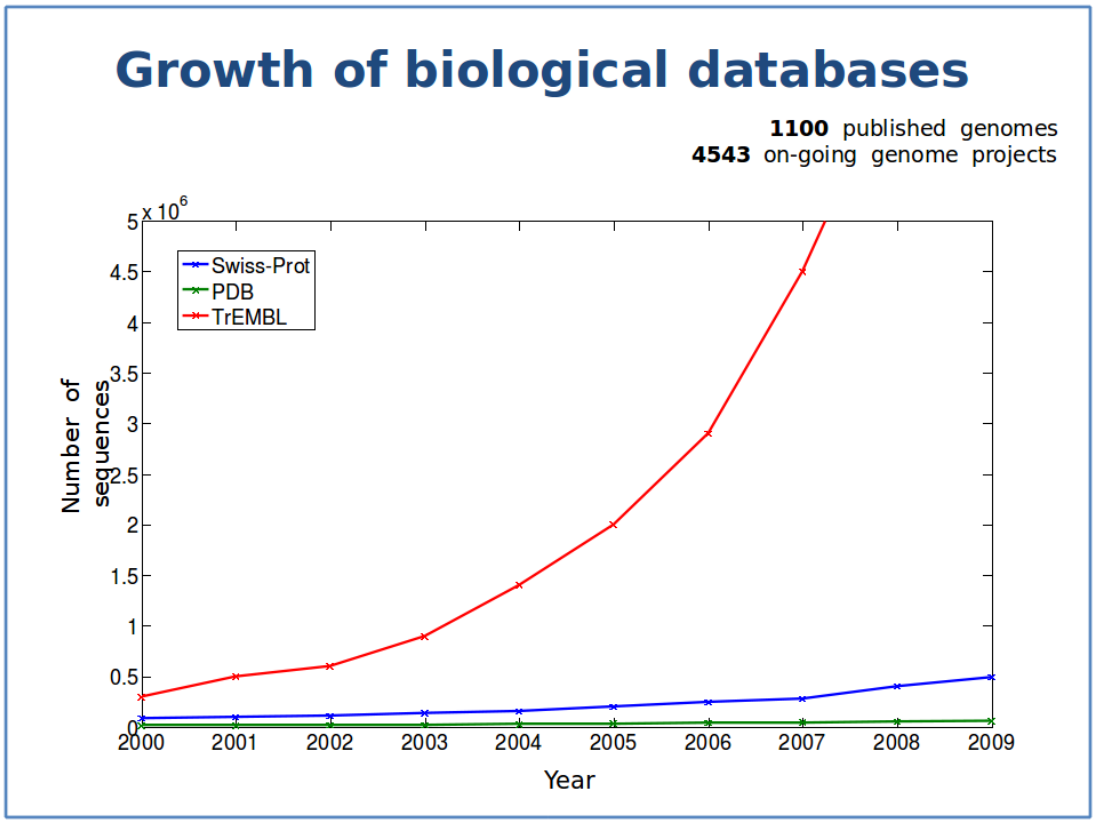


Figure 3.2: The growth of biological databases

Mayer and Laurent Farinelli submitted patents to the World Intellectual Property Organization describing DNA colony sequencing.(32) In 2000 Lynx Therapeutics published and marketed "Massively parallel signature sequencing", or MPSS. This method incorporated a parallelized, adapter/ligation-mediated, bead-based sequencing technology; it became the first commercially available "next-generation" sequencing method, although no DNA sequencers were sold to independent laboratories.(33) In 2004, 454 Life Sciences marketed a parallelized version of pyrosequencing.(34) The first version of their machine reduced sequencing costs 6 times compared to automated Sanger sequencing.(35)

The progress in sequencing technologies continues, now we have a spectre of bext-gen sequencing techinques, including semiconductor highly parallel sequencing machines It all leads to increasing amount of DNA data (see figure 3.2) and

3.1.3 Practical Applications and -omics

The possibility to look into the contents of DNA opened a door for new fields in medicine, biotechnology, anthropology and other social sciences.(36) Next-generation genomic technologies allow clinicians and biomedical researchers to drastically increase the amount of genomic data collected on large study populations. When combined with new informatics approaches that integrate many kinds of data with genomic data in disease research, allowing researchers to better understand the genetic bases of drug response and disease.(37)

Availability of sequenced genomes of many organisms enabled and boosted research in gene expression and regulation, protein structure prediction, mechanisms of genome evolution and many others.

Sequenced genomes allowed for creation of the first software simulation of the entire organism, performed by researchers from Stanford University and J. Craig Venter Institute.(38) They created a simulation of Mycoplasma genitalium, smallest free-living organism with 525 genes.

Researchers developed the software model With data from more than 900 scientific papers reported on the bacterium, with different modules mimicing the various functions of the cell and then are integrated together into a whole simulated organism. The simulation runs on a cluster of 128 computers, recreates the complete life span of the cell at molecular level, reproducing the interactions of molecules in cell processes including metabolism and cell division. Coincidently, the simulation for a single cell division takes around 10 hours, about the same time the living cell takes in its natural environment.

The silicon cell will act as computerized laboratories that could perform experiments which are difficult to do on an actual organism, or could carry out procedures much faster. The applications will include faster screening of new compounds, understanding of basic cellular principles and behavior.

The English-language neologism omics informally refers to a field of study in biology ending in -omics, such as genomics, proteomics or metabolomics. The related suffix -ome is used to address the objects of study of such fields, such as the genome, proteome or metabolome respectively. The suffix -ome as used in molecular biology refers to a

totality of some sort; similarly omics has come to refer generally to the study of large, comprehensive biological data sets.

There are two notable projects related to this field which really highlight the tremendous acceleration of advance in genomic technologies and the current flood of genomic data.

3.1.4 Human Genome Project

The Human Genome Project was the largest collaborative biological project. It started in 1987 and was planned for 15 years (39). The aim was to determine the sequence of chemical base pairs which make up DNA, and to identify and map the approximately 20000 - 25000 genes of the human genome from both a physical and functional standpoint.

It took unprecedented cooperation between researchers from many countries and billions of dollars of funding before the first draft of the first rough drafts became available in 2000. The Human Genome Project was declared complete in April 2003 with the final sequencing mapping of the human genome.

Although this was reported to be 99% of the human genome with 99.99% accuracy a major quality assessment of the human genome sequence was published on May 27, 2004 indicating over 92% of sampling exceeded 99.99% accuracy which is within the intended goal(40). The data gathered by Human Genome Project is the combined "reference genome" of a small number of anonymous donors, not of sequence of every individual's genome. It is available on the internet and there are many ongoing works using HGP data.

Even before the completion of the project lots of practical result emerged from it. Genetic tests to detect predisposition to the variety of illnesses like breast cancer, cystic fibrosis, liver diseases and many others became available from a number of companies. Scientists improved understanding of etiologies for cancers, Alzheimer's disease etc.

Meanwhile, the progress in sequencing technology let us run much more data-intensive projects, aiming for example to identifying differences among individuals, involving single-nucleotide polymorphisms and the HapMap.

3.1.5 1000 Genomes Project

In January 2008 the 1000 Genomes Project was started with the aim of developing a detailed catalogue of human genetic variation which in turn can be used for association studies relating genetic variation to disease. Scientists planned to sequence the genomes of at least one thousand anonymous participants from a number of different ethnic groups within the following three years, using newly developed technologies which were faster and less expensive.

The primary goal of this project was to create a complete and detailed catalogue of human genetic variations, which in turn can be used for association studies relating genetic variation to disease. Secondary goals includes the support of better SNP and probe selection for genotyping platforms in future studies and the improvement of the human reference sequence.

For this project samples were collected from the populations of Yoruba in Ibadan, Nigeria; Japanese in Tokyo; Chinese in Beijing ; Utah residents with ancestry from northern and western Europe; Luhya in Webuye, Kenya; Maasai in Kinyawa, Kenya; Toscani in Italy; Peruvians in Lima, Peru; Gujarati Indians in Houston; Chinese in metropolitan Denver; people of Mexican ancestry in Los Angeles; and people of African ancestry in the southwestern United States.

In 2010, the project finished its pilot phase, which was described in detail in a publication in the journal Nature.(41) In October 2012, the sequencing of 1092 genomes was announced in a Nature publication.(8) Data generated by the 1000 Genomes Project is widely used by the genetics community, making the first 1000 Genomes Project one of the most cited papers in biology.

During the production phase joined sequencing rate reached 10 billion bases or 2.5 human genomes per day which is a groundbreaking capacity. The total dataset includes more then 6 trillion DNA bases, which is 60 times more than collected in previous 25 years.

The work on interpretation of genome data is still in its initial stages. It is anticipated that detailed knowledge of the human genome will provide new avenues for advances in medicine and biotechnology.

3.2 Sequence Alignment

This work focuses on the pairwise local DNA sequence alignment problem. It is extremely computationally intensive as constant progress in sequencing technology leads to ever-increasing amounts of data to be processed. We target GPU-based systems that have been shown to allow for greater performance in sequence processing tasks due to their extreme parallel capacities (42).

3.2.1 Definition

There are different understanding of what to call sequence alignment. The main idea can be expressed as arranging two symbolic representations of DNA or protein sequences next to one another so that their most similar elements are juxtaposed.

Many bioinformatics tasks depend upon successful alignments and typically regions of similarity indicate some functional, structural, or evolutionary relationships between the sequences (9). Generally they fall into two categories: global alignments and local alignments. Global alignment can be interpreted as a form of global optimization that "forces" the alignment to span the entire length of all query sequences. Local alignments identify regions of similarity within long otherwise divergent sequences.

There is a variety of specifications of a task and used algorithms depending on an actual biological problem. Comparing sequences of different species can reveal common ancestry, build phylogenetic tries, detect single nucleotide polymorphism etc.

3.2.2 Dynamic Programming

The formally correct method, yielding best alignment for any sequences is based on dynamic programming. For global alignment it is the NeedlemanWunsch algorithm presented in 1970 (43) and for local alignment - The SmithWaterman algorithm, (44) which uses roughly the same basic idea.

The NeedlemanWunsch algorithm works by constructing a two-dimensional matrix F with one column for each character in sequence A, and one row for each character in sequence B. Thus, if we are aligning sequences of sizes n and m, the amount of memory used is in O(nm). There are some optimizations reported to decrease the memory consumption but the time required is still $O(nm)$. The matrix is filled based on similarity.

The matrix elements F_{ij} are be assigned to be the optimal score for the alignment of the first i= $0, \ldots, n$ characters in A and the first $j = 0, \ldots, m$ and then recursively as $F_{ij} = \max(F_{i-1,j-1} + S(A_i, B_j),\ F_{i,j-1} + d,\ F_{i-1,j} + d)$ To find the alignment the second stage of the algorithms start from the bottom right cell, and compare the value with the three possible sources (Match, Insert, and Delete above) to see which it came from.

The SmithWaterman algorithm sets negative F matrix cells a to zero, which makes local alignments possible. Backtracking is done starting from the highest scoring matrix cell and proceeds until a cell with zero score. Optimal local alignment allows for a correct alignments in regions of low similarity even between distantly related biological sequences. It is noteworthy that CUDA implementation of Smith-Watherman algorithm (CUDASW++) was among the first highly successful bio-informatics applications for GPU, it was highlighted on NVIDIA website for a long time.

Dynamic programming methods yield optimal alignments, but require quadratic time and memory space to execute. There were multiple optimizations proposed, but still time complexity is very prohibitive for aligning large sequences.

3.2.3 Faster Methods

Due to the high space and time complexity dynamic programming approaches are used only for relatively short sequences. Usually proteins or protein-coding DNA regions. For biological reasons different nucleotides in this regions have different mutation probability and because of these substitution matrices are used to assign corresponding scores to substitutions, and a gap penalty for matching an amino acid in one sequence to a gap in the other.

Another popular workflow is searching large-scale databases of annotated genomes, queries are usually gene-coding sequences and researchers are looking for homologous genes. In this case less precise but much more efficient heuristic algorithms or probabilistic methods are being used.

Word methods are typical representatives of this family, they identify series of short, nonoverlapping words in the sequence and match them to the database. FASTA and BLAST (45) algorithms are, perhaps, the most well known of this kind.

FASTA method was introduced in 1985 by David J. Lipman and William R. Pearson The heuristic algorithm observes the pattern of word hits and switches to slow but more

precise Smitwh-Waterman alignment for potential matches. User can specify the search word length - for slower but more sensitive search if the given length is shorter. The legacy of FASTA method is now ubiquitous FASTA storage format, which we used in our implementation as well. The detailed description of the format will be given in the following chapters.

The BLAST, or Basic Local Alignment Search Tool, is actually a family of algorithms optimized for particular types of queries. The BLAST program was introduced in 1990 by Stephen Altschul, Warren Gish, Webb Miller, Eugene Myers, and David J. Lipman form the National Institutes of Health (46). BLAST is following roughly the same logic as FASTA, using a word search of length k, but evaluates only the most significant word matches. It provides a faster alternative to FASTA without sacrificing much accuracy.

BLAST can be used for several purposes:

- to identify sequences from unknown species
- to map a unknown location of sequence based on other sequences from the same species
- to locate common genes in two related species and map annotations from one organism to another
- to construct a phylogenetic tree, however purpose-built computational phylogenetic methods do it better.

The algorithms mentioned above work extremely well on when aligning single proteins or genomic DNA sequences containing a single gene. On the whole-genome level the problem of size emerge. Also for closer related species the search for very similar nucleotide matches is becoming increasingly important. Newer generation of alignment programs tailored for this use typically use index based on a Burrows Wheeler Transform to address the size challenge. They can use genome-scale target database. Example alignment programs are BWA(47), SOAP(48), and Bowtie(49).

3.2.4 Whole-Genome Structural Alignment

More importantly, due to the recent advances in sequencing technology now we have multitude of genomes from the same species, which are extremely similar and the problem of whole genome-level structural comparison emerges.

All the previous algorithms were designed primarily to discover insertions, deletions and point mutations, but not to look for the large-scale structural changes that can be discovered in whole-genome comparisons, such as differences in tandem repeats and large scale reversals. Such a comparison can be based on first identifying regions of 'maximum unique match' (MUM), or the longest subsequence that occurs in both query sequence. And then by using other techniques to analyse structural changes. Software called MUMMer was introduced in 1999 to deal with this task. Matching algorithm was build upon a suffix tree data structure(50).

In this work we focus on performing the search for maximal exact matches in huge sequences, more closely related works and our approach will be described in the following section in more details.

3.2.5 Read Alignment

Another important form of sequence alignment is short read alignment. As described before, modern sequencing techniques split the DNA molecule into pieces that are also called reads. Reads are processed separately to increase the sequencing throughput. Then they are aligned to the reference sequence to determine their position in the molecule.

Read alignment is extremely computationally intensive, as a complete genome of such complex organisms as humans is billions of bases long, and the amount of reads data produced by sequencing machines is usually an order of magnitude bigger(51)(52).

Reads produced by different sequencing technologies can differ in length and error characteristics. Overall, the method of finding maximal unique matches can be used for read alignment, particularly MUMmer software is known to be used for this task.

3.2.6 Defining our Problem

We focus on a problem of finding maximal unique matches ib a big sequences, as it is important for the emerging whole-genome structural comparison problem and can also

be applied for read alignment.

Technically MUM alignment is a substring matching operation: we search for a pattern of length m in reference string of length n, where $n >> m$. Straight-forward naive approach has daunting asymptotic performance of $O(mn)$, so aligning is typically done in two stages:

- Index is build from the reference DNA sequence;
- Each read is matched against the reference sequence using its index.

Based in the data structure selected different matching algorithms are available. In next section we make a survey of existing solutions that use different types of search index are briefly discussed in the following section.

To reduce memory consumption we propose using matching algorithm based on Burrows Wheeler Transform. This algorithm is mainly used for data compression, but possibility of pattern matching using this transform was recently described(53). Index based on BWT is more than ten times smaller than index based on suffix array. We perform an analysis of how this algorithm fits GPU characteristics and do model implementation to see if we can actually get significantly better execution time with this smaller memory footprint algorithm.

3.3 Aproaches to Finding MUMs

3.3.1 Suffix Trees and MUMmer

The theoretically fastest search algorithm uses suffix tree, as index and has computational complexity $O(m)$ (where m is query length) for matching one query to the reference(54).

Suffix tree (also called PAT tree) were first introduced by Weiner in 1973 (55), as a compressed *trie* containing all the suffixes of the given text as their keys and positions in the text as their values. Donald Knuth characterized suffix tree concept as "Algorithm of the Year 1973".

Later on efficient algorithms for suffix tree construction were introduced by McCreight in 1976 (56) , and by Ukkonen in 1995.(57) Ukkonen provided the first online-construction of suffix trees, now known as Ukkonen's algorithm. Finally, Farach (1997)

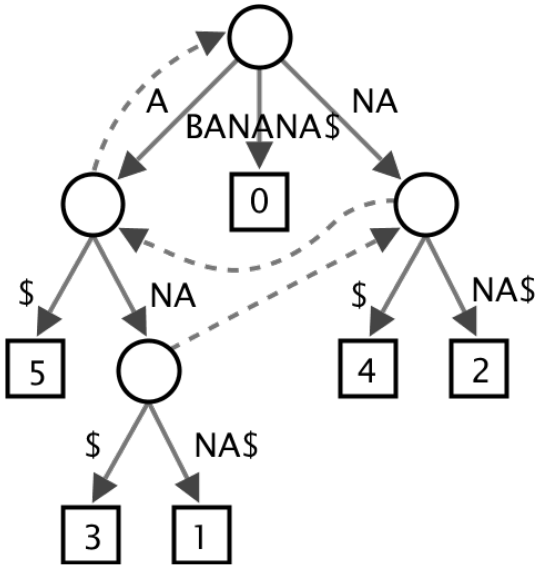


Figure 3.3: Suffix tree

gave the first suffix tree construction algorithm that is optimal for all alphabets which became the basis for new algorithms for constructing both suffix trees and suffix arrays, for example, in external memory, compressed, succinct, etc.(58)

Suffix trees allow a particularly fast implementation of many important string operations, like string search or finding the longest repeated substring and are often used in bioinformatics applications, searching for patterns in DNA or protein sequences (which can be viewed as long strings of characters). The ability to search efficiently with mismatches might be considered their greatest strength.

Also in read alignment we usually search for the longest possible match up to some minimal match length. Instead of repeating search for each subquery the suffix tree can incorporate additional links that connects related suffixes. Thus it allows to search for all subqueries of a given query in $O(m)$ time.

Sequence alignment software called MUMmer was developed on the basis of suffix tree data structure(50). MUMmer is a system for rapidly aligning entire genomes, whether in complete or draft form, it can align of very large DNA and amino acid sequences. Later on it was refactored into GPU-version called MummerGPU, and its authors claimed up to 10 times speed-up over the CPU version (59).

While the suffix tree asymptotic space complexity is linear, the constant multiplier under $O(N)$ (where N is reference length) is very big, between 22.4n and 32.7n bytes for DNA sequences (60), so the memory consumption becomes a serious performance issue on big workloads.

3.3.2 Suffix Arrays and MUMmerGPU++

While theoretically fastest search algorithm uses suffix tree, its space complexity makes it inefficient for big references(54). There were successful attempts to decrease memory footprint of matching algorithm or even to trade computational complexity for space consumption. In MummerGPU++ the authors replaced search algorithm based on suffix tree with one based on suffix array, which lead for another performance improvement(61).

Suffix array is an array of integers giving the starting positions of suffixes of a string in lexicographical order. This data structure was introduced in 1990 by Manber and Myers as a simple, space efficient alternative to suffix trees (62).

1	2	3	4	5	6	7	8	9	10	11	12
m	i	s	s	i	s	s	i	p	p	i	$

Let $S = s_1, s_2, ..., s_n$ be a string and let $S[i, j]$ denote the substring of S ranging from i to j.

The suffix array A of S is now defined to be an array of integers providing the starting positions of suffixes of S in lexicographical order. This means, an entry $A[i]$ contains the starting position of the i-th smallest suffix in S and thus for all $1 < i \leq n$: $S[A[i-1], n] < S[A[i], n]$.

Twelve suffixes: "mississippi$", "ississippi$", "ssissippi$", and so on down to "i$" and "$" that can be sorted into lexicographical order to obtain:

The process is illustrated in the figure 3.4, LCP is the the longest common prefix array, storing the lengths of the length of longest common prefixes between pairs of consecutive suffixes in the suffix array.

This approach, however, has a very bad performance. If we sort all the string with comparison-based sorting it will require $\mathcal{O}(n \log n)$ comparison operations, and comparison of two suffixes requires $\mathcal{O}(n)$ time, giving us overall asymptotic execution

Index	Sorted suffix	LCP
12	$	0
11	i$	0
8	ippi$	1
5	issippi$	1
2	ississippi$	4
1	mississippi$	0
10	pi$	0
9	ppi$	1
7	sippi$	0
4	sissippi$	1
6	ssippi$	1
3	ssissippi$	3

Figure 3.4: Constructing Suffix Array

time estimation as $\mathcal{O}(n^2 \log n)$. Sligtly better performance is possible to achieve using three-way radix quicksort (developed by Sedgewick and also known as multikey quicksort)(63).

Theoretically fastest way to construct a suffix array is via the suffix tree traversal, and for the suffix tree linear-time construction algorithms are available, as we mentioned in the previous section. However, this approach has a unsatisfying space requirement due to the necessity to store the suffix tree.

Suffix array construction in linear time without using suffix tree is called *direct suffix sorting* problem, and now we have many (in fact, many classes) of direct suffix sorting algorithms. These approaches are all exploiting the fact that strings which are being sorted are all related to each other.

The first approaches were based on the prefix doubling strategy, described by of Karp, Miller Rosenberg (1972) (64). Suffixes are bucketed into groups bu common prefixes, and the length of prefixes is doubles every iteration until each prefix is unique and provides the rank of the associated suffix.

The second group of suffix array construction algorithms is recursive and roughly follows suffix tree construction logic. Subset of suffixes are recursively sorted and then used to infer a suffix array of the remaining suffixes.

The third important group is induced copying algorithms, favouring iteration of recursion but also using the idea of sorting a subset of suffixes ind then inducing the order of remaining suffixes. The SA-IS algorithm of Nong, Zhang and Chan (2009)(65) was among the first to achieve $\Theta(n)$ minimal asymptotic complexity, little extra working space requirement and being fast in practice. In fact The detailed survey on a diverse group of suffix array construction algorithms can be found in (66).

Suffix arrays are closely related to suffix trees:

- Suffix arrays can be constructed by performing a depth-first traversal of a suffix tree, if edges are visited in the lexicographical order of their first character.
- A suffix tree can be constructed in linear time by using a combination of suffix and LCP array.

Suffix array takes $N \log N$ bits in general case, whereas the original text takes $N \log |\Sigma|$ bits where Σ is the alphabet. In practical implementation we talk about linear space complexity with constant multiplier under $O(n)$ being 9 bytes per symbol in case of 64-bit implementation.

Search complexity for suffix array is $O(m + \log n)$ where m is the length of query and n is the length of reference. Binary search algorithm can be used for a suffix array.

Evaluation of MummerGPU++ showed that on references over 100MB the memory limit is still taxing performance, since it leads to splitting the index into small pieces to fit into GPU memory and repeating search for each part. Search complexity does not depend (or depends very little) on index size, so splitting index in chunks increases computation time linearly. Copying index and queries to the device also takes its share of time of time. We will provide a more detailed analysis of time consumed by data transfers later on.

3.4 Implementing BWT-based Aligner

As the chief way to increase performance we propose using an algorithm with lesser memory footprint. Such an algorithm can be based on Burrows-Wheeler transform and some additional data structures (FM-Index) instead of suffix array. BWT was introduced in 1994 by Burrows and Wheeler(67) and was used mainly for data compression. There are some recent sequence alignment solutions using BWT, some of them are not

	F		L
mississippi$	$	mississipp	i
ississippi$m	i	$mississip	p
ssissippi$mi	i	ppi$missis	s
sissippi$mis	i	ssippi$mis	s
issippi$miss	i	ssissippi$	m
ssippi$missi	m	ississippi	$
sippi$missis	p	i$mississi	p
ippi$mississ	p	pi$mississ	i
ppi$mississi	s	ippi$missi	s
pi$mississip	s	issippi$mi	s
i$mississipp	s	sippi$miss	i
$mississippi	s	sissippi$m	i

Figure 3.5: Constructing Burrows-Wheeler transform

parallel (Bowtie (49)), some are using GPUs, but for different class of alignment (48). Also in (68) authors discuss the potential of using GPUs for exact sequence matching on single GPU.

3.4.1 Burrows-Wheeler Transform

BWT was introduced by Burrows and Wheeler(67) in 1994. It transforms reoccurring patterns in the string into continuous runs of a single symbol, so this transform is used as a part of compression algorithms such as bzip2 and is sometimes (incorrectly) called "block-sorting compression". BWT itself does not compress data - i.e. the number of characters (or bytes) remains the same, it just permutes the order of symbols.[1]

The Burrows-Wheeler Transformation of a text T, BWT(T), is constructed as follows: The Burrows-Wheeler Matrix of T is the matrix whose rows are all distinct cyclic rotations of T$ sorted lexicographically. It is shown on the right sight of 3.5 for word *mississippi*. BWT(T) is the sequence of characters in the rightmost column (marked L) of the matrix(67).

[1]We use the word symbol as the transform is easier to understand on example of text data, but generally speaking any number of consecutive bits can be a symbol

Obviously there is no need to store and sort all the rotations as strings - this will reqire $O(2^n)$ space and would not be efficient. Instead and array of references to positions in the original string can be used. Such array $R[1..n]$ would have $O(n)$ space requirement and the size of the transformed text is identical to the original, containing exactly the same characters but in a different order.

In the example in figure 3.5 we used special symbol ($) to indicate the end of the string. This leads to increasing the character space by one, but in real implementation can be avoided by maintaining special pointer to the first or last symbols. Alternatively BWT can be constructed in its bijective variant with special technique to avoid infinite comparison of looped word.

Sorting of all the rotations of the input is basically what brings together characters that occur in similar contexts. This is what makes BWT useful in compression algorithms. Typically move-to-front transform and run-length encoding are used together with BWT. For example a piece of Burrows-Wheeler Transform of all of Shakespeare's Hamlet will look like

nnnnnnnnnnnnnnnnnnntnnnnnnnhnnngnnnnnnnnjnnnnnhdnnng
nnnnonnNnnnhhNnnnnnnnnnntnnhnnnnnnnnnnnnnnnNnndnnnhnn
nnnNnnnnnnnnnnnnnnnnnnnnonntnnNNnnnnnnnndngnnnnnnn
nnnnnnnNnnnnnnnngnnnnnnnnnnnnnnnnnngnnnnnnnnonnnnnn
nnnNNnlnnnhnnnnnnnnnnntdbdnnrrmnnmnmnnnuoccpppppdnr
rDolBbbdddodbbBddbbddbdBdbbdbdDddddBbbbbdDbubbdbdbB

and just by run-length encoding the first line can be compressed as "19nt7nh3ng8nj5nhd3ng"

Run-length encoder takes advantage of runs of identical characters in a sequence by replacing them with a singe data value and count, rather than as the original run. The "Move-to-front transform" is working by replacing each symbol in the data with its index in the stack of recently used symbols. These techniques are strongly featured in much of the BWT research, , however, most of the compression is achieved with the arithmetic coding, and there is some evidence that even the run-length encoding isn't necessary if the arithmetic coding is done correctly, although RLE can make implementation simpler and faster (69).

We can notice some properties of BWT Matrix:

- Every column of M_T , hence also the transformed text L, is a permutation of T$.
- In particular, the first column of M_T , called F, is obtained by lexicographically sorting the characters of T$ (or, equally, the characters of L).
- There is a strong relation between M_T and the suffix array built on T.

Another very important property of BWT is that it is reversible - otherwise decompression would not be possible. The original data can be re-generated only using symbols from the last column of the matrix - i.e. BWT itself. The simplest way to perform and understand reverse transformation is the following. Using only the last column we can recreate the first one - by sorting symbols. Then last and the first column put together will give us all the pairs of symbols in the text. By sorting this list of pairs we will rebuild first two column of the BWT matrix. Following the same logic we will eventually reconstruct all the matrix and the row with sentinel symbol at the end would be the original text.

Better approach to the reverse transformation is based on a property called called LF mapping: the i^{th} occurrence of character X in the last column of the BWT matrix corresponds to the same character in original text as the i^{th} occurrence of X in the first column. By counting the number of symbols in F and L we can consequently reconstruct the original data symbol by symbol as shown in the figure 3.6.

3.4.2 Compressed Suffix Arrays

Possible alternative to BWT-based index can be using compressed suffix array. It is possible to decrease space consumption of suffix array from $O(n \log n)$ to $O(n)$ bits, particular constants might differ depending on implementation and usually depend on the entropy of the text. First so called *compressed suffix array* approach, first presented by Grossi and Vitter in 2000(70).

The core idea is to use a *successor array* usually called Ψ such that $SA[x] = SA[\Psi[x]] - 1$. Suffix array can be regenerated from Ψ and also Ψ has a good potential for compression. Because suffixes are lexicographically sorted the Ψ array consists of series of monotonically increasing sequences - two consecutive entries in Ψ will be increasing if the corresponding suffixes start from the same prefix.

F	L	
i_1	p_1	
i_2	s_1	
i_3	s_2	
i_4	m_1	
m_1	i_1	
p_1	p_2	$m_1 \rightarrow i_4 \rightarrow s_4 \rightarrow s_2 \rightarrow i_3 \rightarrow s_3 \rightarrow s_1 \rightarrow i_2 \rightarrow p_2 \rightarrow p_1 \rightarrow i_1$
p_2	i_2	
s_1	s_3	
s_2	s_4	
s_3	i_3	
s_4	i_4	

Figure 3.6: Reverse Transformation

As a simplest way to compress such a sequence delta-encoding in blocks can be used. In this case elements of compressed array are stored as $\Psi[i] - \Psi[i-1]$ Such encoding, however, would allow only for sequential access to elements in blocks. To allow for random access Elias-Fano encoding cab be used. It is slightly more intricate and requires some auxiliary data structures.

it is possible to search compressed suffix array directly - without reconstruction of the original suffix array and even without looking into the original text. One elegant implementation od compressed suffix array with such capabilities was proposed by Sadakane (71) soon after Grossi and Vitter original work.

Array Ψ can also be used for faster than logarithmic search of matches in a suffix array. Let's define array C with $C[p]$ being the number of occurrences of symbols less then p in the text. Using array C it is possible to determine the range of suffix array entries containing suffixes starting from certain letter - just by taking the element of C corresponding to the first letter of the target suffix and the adjacent element.

Using Ψ and C it is possible to iteratively narrow down the range of suffix array corresponding to the last, then last two and so on symbols from the end of the target suffix until we reach the first symbol.

The size of an index based on compressed suffix array is comparable to one based of BWT, but if we use straightforward encoding - the BWT gives us constant compression

```
i:=p, c:=P[p],
First:=C[c]+1, Last:=C[c+1];
while ((First <= Last)
            and (i >= 2)) do
   c:=P[i-1];
   First:=C[c]+Occ(c,First-1)+1;
   Last:=C[c]+Occ(c,Last);
   i:=i-1;
       if (Last<First)
              then return no matches
     else return <First,Last>.
```

Figure 3.7: Procedure Backward_search.

ratio and ability to subdivide index easily. Search algorithm for compressed suffix array and for BWT indexes are very similar, search algorithm for BWT-based index is described in the next section.

Memory-partitioning optimizations proposed in further sections are agnostic of underlying data structures and would be equally applicable to indexes based on BWT and compressed suffix array.

3.4.3 Search Algorithm

It has transpired that the BWT is useful for a lot more than compression because it contains an implicit sorted index of the input string. Eventually full-text indexing and pattern-matching algorithms based on BWT were discovered.

We adopted backward search algorithm proposed by Manzini and Ferragina (53) for our GPU implementation. Here *Occ* is the number of occurrences of given symbol before given position in transformed sequence. Array C contains total number of occurrences of each symbol.

Backward_search procedure (fig. 3.7) uses LF mapping to calculate in rounds the rows of the matrix that begin with progressively longer suffixes of the query string.

3.5 Implementation

The running time of the Backward_search procedure is dominated by the cost of evaluating $Occ(c, q)$. If we build a two-dimensional array OCC such that $OCC[c][q] = Occ(c, q)$ the backward search procedure runs in $O(m)$ time and it requires $O(|\Sigma| n \log n) = O(n \log n)$ bits.

The result of the Backward_search procedure is not the position(s) of matches in the reference sequence but the range of elements in the corresponding suffix array, containing indexes of actual matches in the reference. We suggest using suffix array on a host (which usually has enough memory to store it entirely) to decipher output of Backward_search procedure in $O(1)$ time. While it is possible to resolve positions of matches using the transformed text and OCC, generating all match positions on GPU will provide unpredictable amount of results per query, i.e. each execution thread will need to use unpredictable amount of device memory, and that is unsuitable for CUDA execution model. It will also cause additional overhead for moving data from device to host. To decipher search results on the host side we simply iterate suffix array elements bound by backward search procedure output values.

To compress BWT we make use of the fact that DNA sequences have a very small alphabet (only four symbols), that allows for straightforward encoding, two bit for each symbol. Such compression is almost as efficient as bzip-like typically used with BWT and offers such benefits as the absence of worst-case degradation and the possibility to estimate memory space required for index before the transform. In real application queries can contain extra symbols other than A,C,G and T, like n for inconclusive read, but we do not need to compress queries. Otherwise we could use 4bits for coding extra symbols.

To compress OCC we split the transformed text into buckets of arbitrary size. For each bucket we will store the number of occurrences of each symbol in the transformed text before the first symbol of this bucket. For example, in 64 bit implementation for buckets of 32 symbols we will need 8 bits per symbol to store compressed OCC and 8 consequent memory reads to count the number of occurrences for any symbol. It gives us 10 bits of index per 8 bits of reference sequence and it is possible to change this ratio by varying OCC bucket size. 64 bit suffix array need 17bytes of memory, which is 13.5 times bigger.

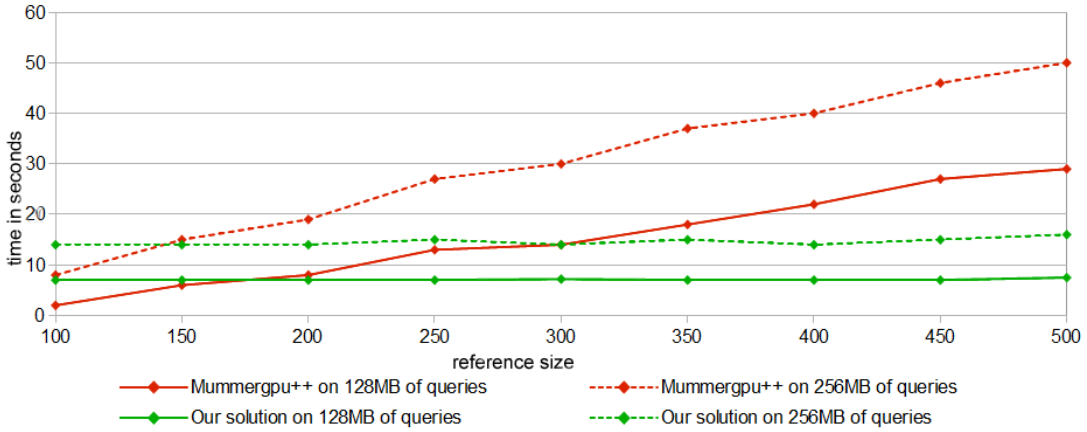


Figure 3.8: Effect of memory partitioning

3.5.1 Preliminary Performance Evaluation

We chose CUDA as target architecture as it is de facto standard for GPGPU programming. The algorithm was implemented in C++ for CUDA programming language. Experimental implementation takes reference and a set of named queries in FASTA format as input. Output is a set of positions in the reference where queries are mapped and for each query the position of the first match character.

By merely replacing suffix array with BWT we already achieved 3-4 times performance improvement for cases where the size of data is too big to fit in memory for suffix-array based software but can be processed in one pass with our approach. Fig.3.8) show how increasing reference size affects performance whether index can (BWT) or can not (suffix array) fit into GPU memory. We used NVIDIA Tesla 2050 card (2.6Gb memory) on the machine with 2.67GHz 4 cores Intel Core i7 920 CPU and 12GB of RAM running under CentOS 5.4.

The program executes in following phases:

1. Build index from reference or load pre-built index.

2. Load query set.

3. Move index and queries to GPU.

4. Align queries to reference using its index.

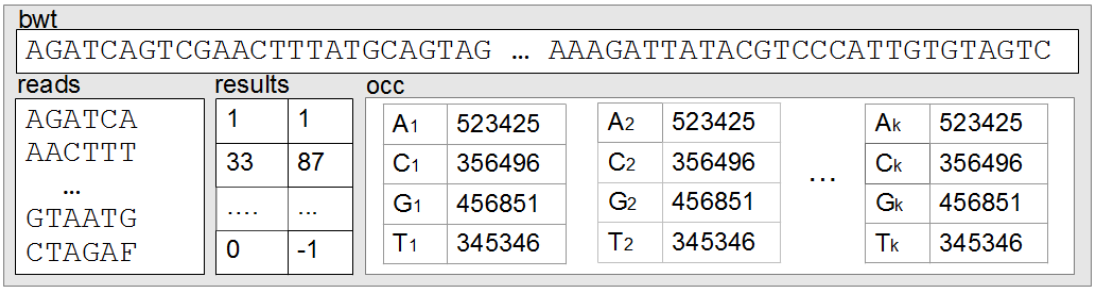


Figure 3.9: Data in GPU Memory

ion 0.99d (TeX Live 2013/Debian) The top-level auxiliary file: thes

5. Copy results to host.

6. Output results.

The CUDA kernel that performs the query search is an almost straight-forward implementation of procedure Backward_search, where each thread is processing its own query independently. Each thread stores results in its own preallocated global memory and accesses the reference index only by reading. Therefore there are no race conditions and no need for synchronization. Performance profiling showed that major share of time is consumed by loading data from global memory. On references over 100mb MummerGPU++ starts to subdivide index and loses performance, while with our approach index up to several gigabytes (i.e. complete human genome) can be stored in GPU memory.

3.5.2 Partitioning Big Workloads

Index based on BWT is small enough to store the entire human genome in memory of modern GPU devices (up to 6Gb in recent models) which is enough for aligning the DNA material from a known source, such as a sample from human biological material that needs to be aligned only against the human genome. This allows us to fully exploit parallel capacities of GPU without having to split the index and process it chunk by chunk. Yet there are situations when the size of index as well as the amount of query data considerably exceed device memory size. For example, in metagenomics DNA samples are taken from environmental samples and might be required to be aligned against a range of reference genomes, total size of which can me much bigger than device memory size.

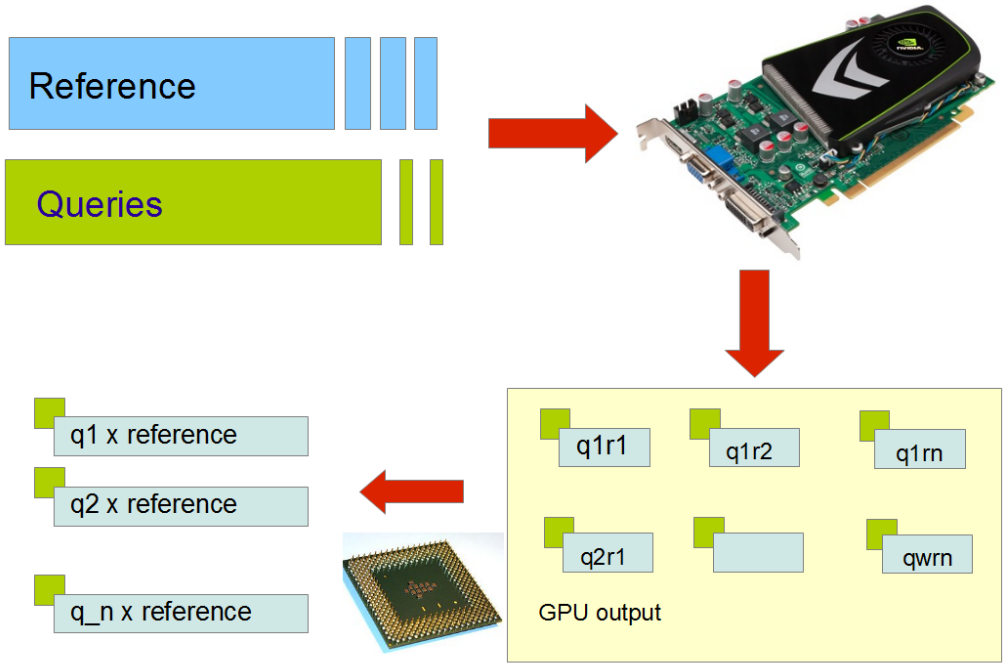


Figure 3.10: Dealing with Bif Workloads

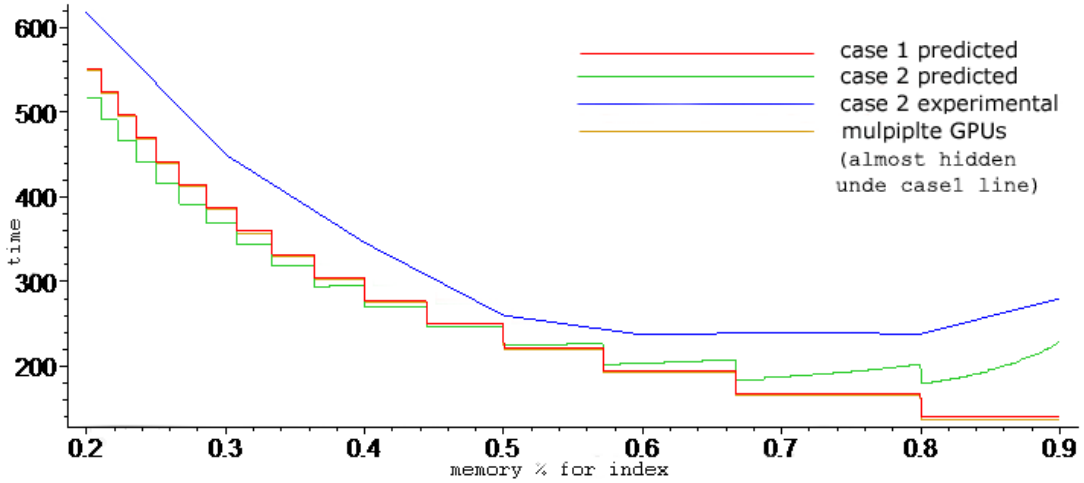


Figure 3.11: Performance evaluation

In this case we need to somehow split the index as well as the queries and process them chunk by chunk, aligning each chunk of query set against each chunk pf index. It is not obvious how to do this decomposition, in which proportions to allocate memory for index and for queries and in what order to process them. In the following section we describe our performance model for this process, so as to see if it is possible to find the optimal balance and get maximal performance. (72) (73)

3.5.3 Performance Model and Workload Balancing on single-GPU

The theoretical complexity of matching algorithm itself is $O(q)$, where q is query length. In case of sequential execution increasing number of queries to process obviously increases execution time in the same linear manner.

So we can say that the overall execution time depends linearly on the overall size of query set. Parallel execution on a GPU device will show little or no increase of computation time until the number of queries si big enough to occupy all cores and saturate memory bus. Having 512 CUDA cores on Fermi card multiplied by roughly esteemed number of 256 thread, multiplied by 100(the number of bases in one query) - gives us about 10mb of query data to be processed in one kernel launch, which is much smaller than the typical amount of available onboard memory and even more negligible compared to the overall workload size. So we will consider performance to be linearly dependent on query size for overall GPU performance, assuming index and query can

fit entirely into GPU memory and keeping in mind that we do not want to make query load less than 10mb.

Let us call memory size S_{mem}, index size S_{idx} and query set size S_{qry}. The overall execution time consists of the computation time itself and the time spent on moving data between host and device: $T = T_{cmp} + T_{mem}$. This formula assumes the worst case scenario when there is no overlapping between computation and data transfers. Cases where such overlapping is possible will be discusses below.

Let's take a look at T_{cmp}. Suppose we have to split the index into N_{idx} chunks of size P_{idx} each and the query set into N_{qry} chunks of P_{qry} bytes. There is an obvious correlation between N_{idx} and N_{qry}, but for the time being we shall not include it in the model to keep it simpler. We have to match each chunk of query set against each part of index, one such iteration (kernel launch) taking $C * P_{qry}$ time as complexity does not depend on index size. We have to repeat the matching procedure for each part of index and for each part of query set, which gives as execution time $T_{cmp} = C * N_{idx} * N_{qry} * P_{qry} = C * S_{qry} * N_{idx}$. As the overall size of S_{qry} is constant it seems preferable to allocate greater share of memory for storing index.

Now let's consider the communication expenses of moving index and query set parts from host to device. We have two basic options here. One option is to place one part of index on device, processing all subsets of query set one by one and then doing the same procedure for next part of index. The other option is to do the matching vice versa, i.e. matching one part of query set against all parts of index and then proceed to the next chunk of query set.

In the first case we need to copy P_{idx} bytes for each part of index, then N_{qry} times P_{qry} bytes of query subsets which equals to S_{qry} bytes and then to repeat this process N_{idx} times. Given host-to-device transfer bandwidth β communication will take $T_{mem} = \beta(P_{idx} + S_{qry}) * N_{idx} = \beta S_{idx} + \beta S_{qry} N_{idx}$ time. The overall time will be $T = C * S_{qry} * N_{idx} + \beta S_{idx} + \beta S_{qry} N_{idx} = (C + \beta) S_{qry} N_{idx} + \beta S_{idx}$. Obviously the only variable parameter here is N_{idx} and we maximize it by keeping the size of index chunk as big as possible. At the same time the size of query set chunks goes to zero, that is to the minimal amount required to saturate GPU memory bus. As the overall time does not depend on number of query chunks, we can split each chunk into two parts without affecting time, and copy one part while processing another. Having typically $\beta < C$ we can completely hide the communication cost.

For the second case using the same logic we get $T = C * S_{qry} * N_{idx} + \beta S_{qry} + \beta S_{idx} * N_{qry}$ overall execution time.

In the second case we copy one part of query set of P_{qry} bytes and the whole index part by part (S_{idx} bytes). Then we repeat it N_{qry} times which gives us $\beta(P_{qry} + S_{idx}) * N_{qry} = \beta S_{qry} + \beta S_{idx} * N_{qry}$ time. The overall time will be $T = C * S_{qry} * N_{idx} + \beta S_{qry} + \beta S_{idx} * N_{qry}$. In this case the performance gained through smaller amount of index chunks is concealed by losses caused by more frequent transfers of index.

Let α be the share of memory occupied by index. Then each chunk of index will use αS_{mem} bytes and each chunk of queries $(1 - \alpha)S_{mem}$ bytes. We will have to split index into $N_{idx} = S_{idx}/\alpha S_{mem}$ chunks and query set into $N_{qry} = S_{qry}/(1 - \alpha)S_{mem}$ chunks. Figure 3.11 shows how variation of α changes the overall execution time and that the first case allows for a potentially higher performance.

Actual value of C is retrieved form experiment and it depends on many parameters, like minimal required match length etc, but the asymptotic behaviour will be the same. Performance of test implementation on big workloads confirms the predicted model (figure 3.11).

So in the first case the overall performance increases as the index size is increased. This process continues up to the point where the memory remaining for queries is enough to run kernels with full memory saturation, which is relatively small and is not shown in figure3.11.

In the second case we increase index size up until the point where communication expenses of repeating transfers of big index chunks are equal to the time spent on processing queries on extra number of index chunks. Maximal performance is better in the first case and it seems preferable from the point of view of pure GPU productivity. Moreover, it allows us to overlap communication and computation, as we can split queries without much penalty making performance even closer to ideal.

However, in this model we do not take into account the fact that results of matching of each subset of queries against each part or index need to be merged with each other. In the first case we have to store results of matching against each part of index somewhere until we process all queries and it will tax CPU-side memory/storage. This approach is completely inapplicable in a situation where queries are being streamed from some source (i.e. a sequencing machine) and we need to process each query block

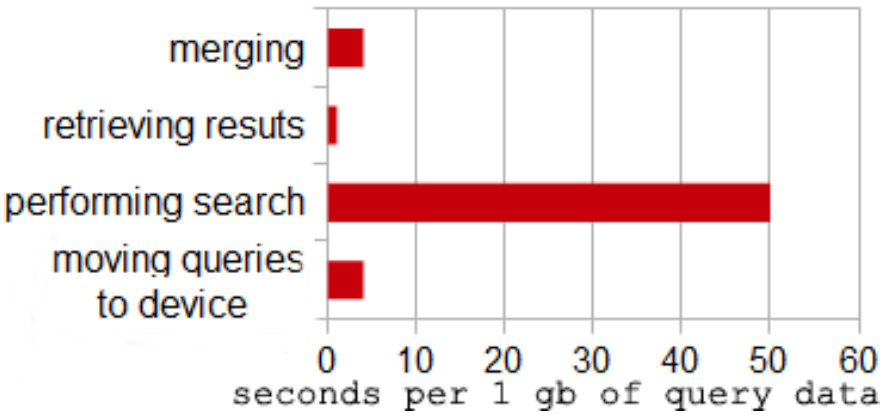


Figure 3.12: Performance details

as it comes so we have to stay with worst case model - or we can try using multiple GPUs.

3.5.4 Multiple GPUs

Index chunk distribution among multiple GPU devices allows for smaller amount of repeatedly loaded index chunks per device. Ideally index chunks are not being moved at all. In this case theoretical performance in terms of pure GPU productivity will be even better, though not significantly, than that provided by the first approach on a single GPU device. On each device we spend $C * S_{qry} + \beta S_{qry}$ time for moving and processing all queries (once again, overlapping is possible in this case).

In perspective of the whole application two problems can occur. Copying data to multiple devices can be slower than copying to single device. But even in the worst case, when data can only be moved to GPU devices one by one, it will not be an issue: since when devices are initially filled with workloads one by one, we can keep this asynchronism in execution and all succeeding data transfers.

Another potential problem is than merging results can take up more time than matching itself.

The process of deciphering and joining results consists of following stages. We get the ranges of suffix array elements as output of each GPU matching routine and restore actual positions of matches in reference sequence. For each device output we will have such list of positions. Then we need to merge these lists together and sort resulting list. It does indeed look like time consuming routine, but it obviously has $O(N_{idx})$

complexity, the same as complexity of search procedure itself. The exact multiplier depends on implementation, CPU characteristics and average number of matches for each query. However, given realistic search output, our sequential test implementation performed merging of 8 chunks of one million results in less then one second, which is definitely faster than processing corresponding amount of data on GPU (fig. 3.12). In previous experiments we used queries of 100 bases long, so 1 million results correspond to 100Mb of query data. In tests on both real and generated sequences multi GPU performance per device was same as for single GPU case 1. We performed benchmarking on one of the Tsubame 2.0 supercomputer nodes with 2 six-core Intel Xeon X5670 CPUs and 54GB of RAM running under SUSE Linux Enterprise Server 11 SP1 for this test. The node has three NVIDIA Tesla 2050 GPUs connected with 16 lanes of PCI Expression 2 on it. We used 100 bases long queries and set minimal match length to 40 bases. For 6GB reference sequence aligning efficiency per device was 3.55 million bases per second for single GPU and 3.7 for multi GPU implementation when all 3 devices were used. So 3 GPUs compared to single one gave us 3.11 times speed-up, i.e. 1.04 efficiency. Optimal number of devices is equal to the number of index chunks of optimal size. Increasing number of GPUs further will negatively affect the efficiency as index chunk size will be decreased.

3.6 Conclusion

Read alignment is basically a string matching problem and is typically done by building index of a reference and then matching queries against it. There are several types of indexes and corresponding match algorithms which were being used for alignment problem. We made a survey of existing solutions (50),(59),(61), and found that memory limitation is the performance bottleneck in all cases. Workload size for both reference sequence and query set can dramatically surpass available device memory and each index subdivision into smaller chunks to fit into memory simply doubles execution time. For example human genome contains approximately 3 billion of bases. Suffix array (array of integers giving the starting positions of suffixes of a string in lexicographical order) needs 9 bytes per base, so it will require 27 gigabytes of memory, while top modern GPUs have about 6GB. To index bigger references 64 bit integers are required and suffix array space complexity will be 17 bytes per base.

Faster and faster computing systems are developed every day to cope with ever-increasing complexity of problems that emerge in various areas of science and technology. Performance growth comes from technological advancements and mainly form architectures facilitating parallel data processing in various forms (i.e. recently GPUs). At the same time algorithms known to solve particular tasks themselves have many possibilities of improvement, taking into consideration fact that overall performance comes not just from better algorithm, but also on how it fits certain peculiarities of hardware platform and different patterns of data distribution in heterogeneous systems. GPUs and clusters of GPUs have recently become one of the main threads of supercomputing. Their computational characteristics are different from those of traditional systems and they are relatively new to software developers, which makes the above-stated issues even more important. Also while some applications have a pretty uniform data model, like those solving various matrix-based mathematical problems, in other applications data model itself is heterogeneous and its decomposition requires a profound study of balancing storage and distribution of workload parts so that we could better meet the platform characteristics and improve the overall performance.

Better software performance does not necessarily come from computational complexity of underlying algorithms. Choice of particular data structures and corresponding algorithms depends on how they meet characteristics and features of target hardware. This is particularly true for GPU devices.

This work shows that using more compact data structures can lead to performance improvement in short read alignment problem. We refactored MummerGPU++, previous highly-efficient GPU exact-matching read alignment software by replacing suffix array with BWT and rewriting the corresponding search algorithms and get 3-4 times performance improvement. The analysis of application behavior for the case of workload size considerably exceeding device memory proves that higher performance can me achieved by intelligent strategy for data decomposition. We also showed that best performance per device for read alignment problem can be achieved by using multiple GPUs, and the optimal number of GPU devices for a particular task can be estimated from reference size.

As the chief way to increase performance we propose using an algorithm with lesser memory footprint. Such an algorithm can be based on Burrows-Wheeler transform and some additional data structures (FM-Index) instead of suffix array. BWT was

introduced in 1994 by Burrows and Wheeler(67) and was used mainly in compression algorithms such as bzip2 as it transforms reoccurring patterns in the string into continuous runs of a single symbol, but it can be also used for pattern matching.

The second one is the performance model of possible memory utilization strategies. This model allowed us to find best proportions and succession of memory allocations and data transfers to maximize overall performance. We found that optimal performance is possible to achieve by using multiple GPU devices.

4

String Sorting

4.1 Introduction

Sorting, or ordering the sequence of items in ascending or descending order, is one of fundamental and most widely studied algorithmic problems in computer science. According to Donald Knuth (74) "computer manufactures in 1960s estimated that more than 25 percent of the running time on their computers was spend on sorting, when all their customers were taken into account. In fact, there were many installations in which the ask of sorting was responsible for more then half of the computing time."

Solving togetherness problem, matching items in two lists, binary search by key - are common example of how sorting can be useful. Sorting routines are widely used standalone for storage and manipulation of data, and also often serve as a basis for more complex algorithms in various areas from graph and spatial data processing to molecular biology.

With data becoming Big Data, efficiency of sorting algorithms becomes of increasing importance. Another important consideration is the data becomes more diverse in nature, particularity bigger volumes of non-numeric data are emerging (75) and require for specialized algorithms (76).

There are two ways to improve sorting performance: optimization of the algorithms themselves and their adaptation to massively parallel hybrid architectures. An efficient algorithm should combine workload balancing, economic data transfers and overall computation model that is adequate to strengths and limitations of underlying hardware

platform. Sorting algorithms also make a valuable case study of how to attack parallel programming challenges in general.

This work addresses the issue of efficient sorting of strings on multi-core processors, which has not received as much attention as sorting of numeric data. We describe our approach to parallelization of MSD radix sort and discuss its the applicability to GPU and traditional CPU architectures. Further we compare CPU and GPU implementations with regards to the efficiency of the sorting algorithm on varying key length.

4.1.1 Approaches to Sorting

Let's say we are given a list of N records $R_1, R_2, ...R_N$, containing some data and keys $K_1, K_2, ...K_N$. The goal of sorting is to arrange these records according to the ordering relation "$<$" applied to keys. Since the sorting algorithms deal only with keys and are agnostic about the additional values we will focus on sorting a list of N keys, sometimes called items. If the keys are big it makes sense to move pointers to keys instead of moving bulky keys themselves, this approach is called *address table sorting*. In case when the keys are multi-character arrays or strings, such approach is dictated by the data representation itself, as string arrays are typically stored in memory as two arrays - of pointers and symbols belonging to each sting in contiguous memory block.

Ordering relation "$<$" should satisfy following conditions for any key values a, b, c:

- Only one of the responsibilities $a < b, a = b, b < a$ is true. (trichotomy)
- If $a < b$ and $b < c$, then $a < c$ (transitivity)

We can notice that these conditions are equivalent to mathematical concept of *total ordering*. The sorting is called *stable* if we make the further requirement that records with equal keys should retain their original relative order.

Most classical sorting algorithms are build only on the assumption that the "$<$" relation is the only thing we know about the data type of keys. We will call such algorithms *comparison based*. Their advantage is that they can be applied to any data type if the comparison operator is provided. In modern programming languages these algorithms are implemented as template or generic code, which user can parametrize with own comparison operation implementation or provide a reference to appropriate library routine.

algorithm	guarantee	average	extra space	assumption of keys
insertion	$N^2/2$	$N^2/4$	no	comparable
selection	$N^2/2$	$N^2/2$	no	comparable
merge	$N \log n$	$N \log N$	N	comparable
quick	$1.39N \log N$	$1.39N \log N$	$c \log N$	comparable
LSD radix	WN	WN	$N + R$	digital

Table 4.1: Frequency of instruction in the inner loop

Simple approaches which can be build upon a comparison operation are well known of comparison based algorithms are *insertion sort* (items are considered one at a time, and each new item is inserted into the appropriate position relative to the previously-sorted items) and *bubble sort* (if two items are found to be out of order, they are interchanged. This process is repeated until no more exchanges are necessary).

A fundamental limit of comparison sorting algorithms is that they require *linearithmic* time - $O(N \log N)$ in the worst case. Two wll-known classical comparison-based sorting algorithms that achieve this asymptotic complexity are merge-(77) or quicksort(78).

However this formula allows us to estimate only the amount of comparison operations needed to complete the sort and not the actual performance time. So, for these sorts execution time depends on the cost of comparison operation, and that depends on the actual data. For example, lexicographic sorting of strings requires comparison of many symbols, and that makes the complexity of the sorting dependent on what is called the longest common prefix (LCP) = $\frac{1}{n-1} \sum_{i=0}^{n} (LCP(S_i, S_{i+1}))$ where $LCP(S_i, S_{i+1})$ is the number of symbols two adjacent strings in have in common.

On the other hand, better performance is possible on real-world data (such as almost-sorted data), and algorithms not based on comparison, such as counting sort, can have better performance. Although many consider sorting a solved problem asymptotically optimal algorithms have been known since the mid-20th century useful new algorithms are still being invented.

4.1.2 Sorting by Counting

Sorting by counting, or *key-index counting* is a simple primitive which is used as a

building block for string sorting algorithm and it also provides a good and simple example how better sorting performance can be achieved if we tailor particular data type.

This method is based on the idea that the jth key in the final sorted sequence is greater than exactly j 1 of the other keys (74, 79). Let's assume input to be a collection of n non-negative integer key whose maximum value is at most σ.

The algorithm iterates over the items, computing a histogram of the number of times each key occurs within the input collection. It then performs a prefix sum computation (a second loop, over the range of possible keys) to determine, for each key, the starting position in the output array of the items having that key. Finally, it loops over the items again, moving each item into its sorted position in the output array.

Algorithm 1 Key-index counting

```
procedure SORT(input, N)
    for i ∈ (0..N) do                                   ▷ counting
        count[input[i]] ← count[input[i]] + 1
    end for
    for i ∈ (0..σ) do                                   ▷ offsets
        offset[i] ← Σ_0^{i−1} count[i]
    end for
    for i ∈ (0..N) do                                   ▷ moving
        output[offset[input[i]]] ← input[i]
        output[offset[input[i]] ← output[offset[input[i]] + 1
    end for
    return output
end procedure
```

Algorithm 1 shows the pseudocode for counting sort. *input* here is the input array of integers and σ is the maximum key value. The first loop counts keys in the input array to the array *cound*, this process is also called building a *histogram*. The second loop builds the array of the number of items with key less than ith, which is the same as the first index at which an item with key i should be stored in the output array. This process is also called *prefix sum*.

In our simplified pseudocode the prefix sum computation looks redundant in a sense that it iterates over the *count* array $O(\sigma^2)$ times, in practice linear-time implementation

is possible and pretty straightforward.

The *output* is an array of ordered items. From the pseudo-code we can see that such implementation of counting sort is *stable* and requires $O(N)$ time to execute - indeed, it performs only two loops over the N input elements. Strictly speaking, it require Therefore the time for the whole algorithm is the sum of the times for these steps, $O(N + \sigma)$ time, but in practical case σ is significantly less than N. The spatial complexity of the algorithm is also $O(N + \sigma)$.

4.1.3 Sorting Longer Keys

As we can see, sorting algorithms can deliver faster performance for certain data types, up to linear performance in the ideal case, when we take into account the additional knowledge about keys data type. On the other hand ignoring it can cause serious performance degradation, like in the case of string, when comparison operation does not take constant time.

One of the algorithm that we focus on is a radix sort which is a build on the top of key-index counting

Radix sort comes in two flavors: sort that starts from the most significant digit (MSD radix sort) or from the least significant digit (LSD radix sort). The term "digit" is used because radix sort is generally applied to integers; it actually refers to any amount of bits in the binary representation of the number. However, since this paper describes application of radix sort to strings, we shall hereafter use the term "symbol" rather than "digit".

LSD radix sort is perhaps the most commonly used one; it performs well on short-length keys such as integer numbers. The most efficient implementation of this algorithm is now part of CUDA SDK(80). However, LSD sort is bound to short keys of fixed length, which does not cover many types of data.

The reason why LSD radix sort is bound to short keys is that it starts from the rightmost symbol and proceeds to the previous one while maintaining stability of the sort, and then the algorithm is repeated until the first symbol is reached. On relatively long keys this approach would not be efficient because comparing the first several symbols should be enough to determine the order of strings. Moreover, with a long key the number of iterations goes up, and the performance decreases accordingly.

MSD radix sort does not have this problem in that it starts from the leftmost symbol and then moves up to the next symbol only for the strings the order of which is not yet determined. It can be viewed as bucket sort because this process basically consists of recursive distribution of strings into buckets: at first all strings are placed into different buckets depending on their first symbol, and then the strings inside each bucket are partitioned again by the next symbol. This process is fairly intuitive, but its recursive nature makes it challenging to implement on GPU.

Another algorithm that can be useful for sorting strings is *Three-way radix quicksort* or *multikey quicksort*, which is the combination of MSD radix and quick sorts.

4.1.4 Parallel Sorts

Sorting algorithms have been studied extensively, and there have also been numerous attempts to develop parallel approaches to sorting. Comparison-based sorts are the most commonly used and applicable to various kinds of data. The most efficient algorithms are based on divide-and-conquer approach and are tricky to parallelize efficiently. We now have parallel versions of quick sort (81) and merge sort(82, 83) , among others. There is also bitonic mergesort sort (84) which was developed to be more paralellization-friendly. The efficiency of certain algorithms and their implementation is also relative to the type of data being sorted and underlying hardware architecture. Most of these sorting algorithms are memory-bound, and, for distributed memory systems, communication-bound.

SIMD architectures are putting even more limitations on what can be implemented and, like the recently popular GPUs, provide totally different performance trade-offs. Distribution sorts which are inherently efficient for certain type of data have been especially successfully implemented on GPU. Radix sort which utilizes thread parallelizm and hight memory throughput was reported to be highly efficient on GPU (85). They have also presented a quicksort implementation for GPU with inferior performance. To the best of our knowledge, the most efficient GPUu radix sort is currently the one from Thrust library presented by Merril and Grimshaw (80).

Comparison-based sorts were also implemented on GPU. Purcell et al. (86) presented bitonic merge sort on GPUs based on the work by Kapasi et al. (87). Greß et al. (88) used the sorting technique presented in the Bilardi et al. paper (89) to implement GPU adaptive bitonic sort. Another GPU sort based on bitonic was implemented by

Govindaraju et al. (90). Later they presented a hybrid CPU and GPU solution using bitonic-radix sort in Tera-Sort challenge(91). An approach that combines several algorithms was presented by Sintorn et al. (92); their solution splits the data with a bucket sort and then uses merge sort on the resulting blocks. Finally, there were more successful attempts to implement quick sort on GPU (93). There are ongoing efforts to optimize comparison-based algorithms for new architectures, e.g. by using vector instructions of modern processors (94).

However, all the above-mentioned radix sorts perform better on numerical data, since they are LSD radix sorts and can not work with long keys; and none of comparison-based sorts are efficient for string data. The one algorithm that is known for high performance on strings is MSD radix sort (74). There is also 3-way radix quicksort presented by Bentley and Sedgewick (63, 95), which is even more efficient due to more optimal use of caching. However, at present, to the best of our knowledge, there are no parallel implementations of radix sort that could handle long string keys. This work addresses this gap. Our solution is based on MSD radix sort which is less complex and more GPU-friendly, and equally efficient on the initial stages of the algorithm (while the bucket are relatively big). On the later stages, as buckets get smaller, we are switching to the 3-way radix quicksort.

4.2 Parallelizing String Sort

In previous section we made a survey on different sorting algorithm and found two which are appropriate for sorting string data - MSD radix sort and multikey quicksort. Now we will introduce this algorithm in greater detail and provide our strategy for building efficient parallel implementation.

4.2.1 MSD Radix Sort

As we briefly mentioned, MSD radix sort is using so called key-indexed counting which and can be outlined as follows:

1. count frequencies of each symbol using key as index
2. compute offsets
3. access cumulates using key as index to find record positions

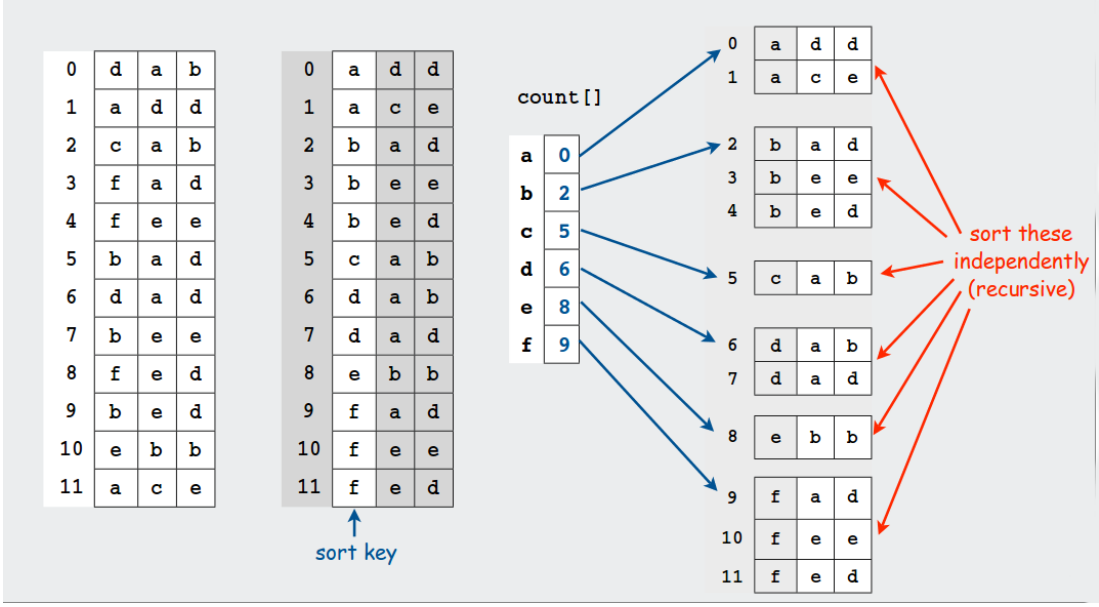


Figure 4.1: MSD radix sort

4. copy back into original array

To perform MSD radix sort we partition input items into σ buckets according to first character and then sort all strings that start with each character (see fig. 4.1). σ here is the size of the alphabet.

MSD radix sort can be implemented *in place*, i.e. without the requirement for auxiliary array for keys, but as we are moving only pointers to keys (strings), not the keys themselves - the overhead of using auxiliary array is not significant. On the other hand, such implementation allows for *stable* sort.

The alg. 2 shows pseudo-code for our implementation.

Here S is the array of strings and $S[i][j]$ denotes j_{th} symbol of i-th string and S_{aux} is auxiliary array. Though we use double-buffering technique, S and S_{aux} are storing only pointers to strings, so the increase in memory consumption is not significant. C is the array of counters for each letter of the alphabet and O is the array of pointers to the beginning of each bucket. N is the number of strings being sorted and σ is alphabet size. d denotes sorting depth, i.e. the position of symbol we use for partitioning strings into buckets.

Algorithm 2 MSD Radix Sort

```
procedure SORT(S, l, r, d)
    for i ∈ (0..N) do                                   ▷ counting
        C[S[i][d]] ← C[S[i][d]] + 1
    end for
    for i ∈ (0..σ) do                                   ▷ offsets
        O[i] ← ∑_0^{i−1} C[i]
    end for
    for i ∈ (0..N) do                                   ▷ moving
        S_aux[O[S[i][d]]] = S[i]
        O[S[i][d]] ← O[S[i][d]] + 1
    end forS = S_aux
    for i ∈ (0..σ) do                                   ▷ recursion
        if (i! = EOL)and(C[i] > 1) then
            Sort(S, O[i] − C[i], O[i], d + 1)
        end if
    end for
end procedure
```

4.2.2 3-Way Radix Quicksort

MSD radix sort provides theoretically best performance in term of how many times do we need to examine every symbol of every string. However, the real implementation requires creating counter variables for each possible symbol if alphabet, while the bucket if strings that is being sorted may contain only certain subset of this symbols as the current key.

The way to avoid this and slightly increase the performance was proposed by Bentley and Sedgewick in an algorithm called 3-way radix quicksort, or multikey quicksort (95). This algorithm is a combination of radix sort and quicksort.

Quicksort is a textbook divide and conquer comparison-based algorithm developed by Tony Hoare in (78) 1961. It chooses a partitioning element, permutes the elements such that lesser elements are on one side and greater elements are on the other, and then recursively sorts the two subarrays. Original version of the algorithm used binary partitioning method, placing lesser elements on the left and greater elements on the right, but equal elements could appear on either side.

This approach was exhibiting suboptimal performance when sorting sets containing many equal keys. Eventually, 3-way quicksort was developed to avoid this problem. 3-way quicksort partitions items into 3 groups - less then, breater then and equal to the pivot element (96).

It is interesting to mention, that the MSD radix sort algorithm is isomorphic to the *trie* data structure. The same way, quicksort is essentially building *binary search tree* abd 3-way quik sort - *ternary search tree*

3-way radix quick sort works like this

- It picks an element from the array (the pivot) and consider the first character (key) of the string (multikey). Then it partition the remaining elements into three sets: those whose corresponding character is less than, equal to, and greater than the pivot's character.
- Recursively sort the "less than" and "greater than" partitions on the same character.
- Recursively sort the "equal to" partition by the next character (key).

Given we sort using bytes or words of length W bits, the best case is O(KN) and the worst case O(2KN) or at least O(N2) as for standard quicksort, given for unique keys N¡2K, and K is a hidden constant in all standard comparison sort algorithms including quicksort. This is a kind of three-way quicksort in which the middle partition represents a (trivially) sorted subarray of elements that are exactly equal to the pivot.

4.2.3 Implementing 3-Way Radix Quicksort Without Swaps

Classical quicksort implementation is based on swapping: algorithm scans items form left to right, compare them to the pivot and moves to appropriate partition. Partition pointers are updated accordingly. This approach makes quicksort algorithm *unstable* and also it makes it difficult to parallelize.

We can approach quicksort in a different way, inpsired by MSD radix sort: let's iterate onve over all the items in the array count how many of them are less then, greater then, or equal to pivot. Then based on these three counters we can figure out the exact space for three partitions on the second iteration of the loop move items to

Algorithm 3 3-Way Radix Quicksort

```
procedure SORT(S, lo, hi, d)
    if lo < hi then
        return
    end if
    lt ← lo, gt ← hi
    v ← charAt(a[lo], d)
    i ← lo + 1
    while i <= gt do                    ▷ 3-way partitioning (using d th character)
        t ← charAt(a[i], d)
        if t < v then
            exch(a, lt++, i++)
        else if (t > v) then
            exch(a, i, gt−−)
        else
            i++
        end if
    end while
    Sort(a, lo, lt − 1, d)
    if v >= 0 then
        Sort(a, lt, gt, d + 1)
    end if
    Sort(a, gt + 1, hi, d)
end procedure
```

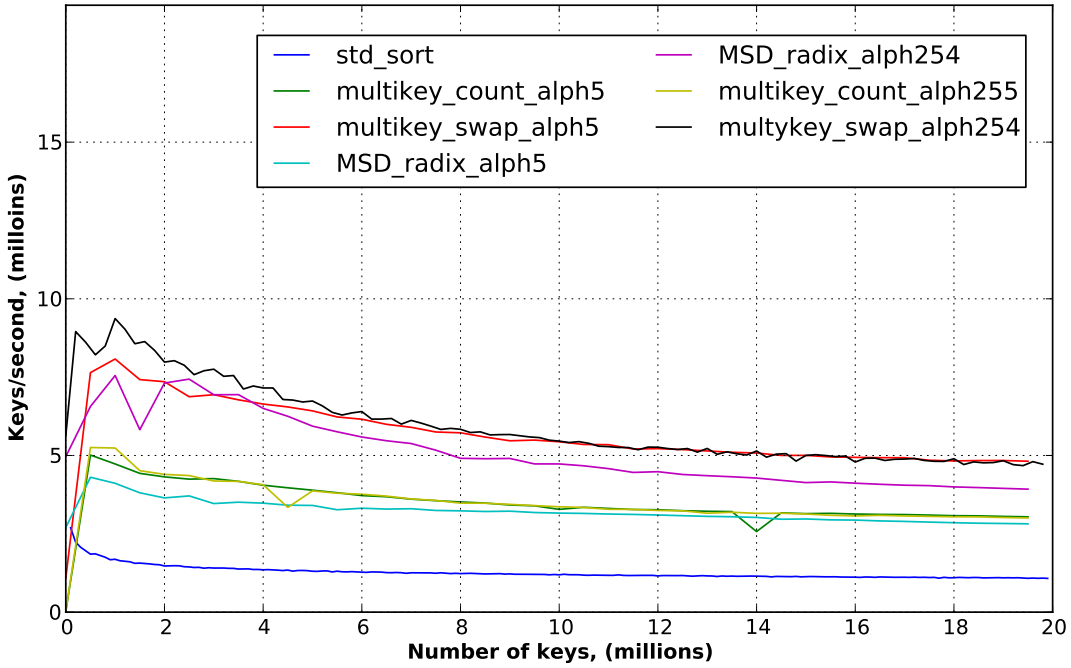


Figure 4.2: Performance of sequential algorithms

appropriate places. Such implementation is stable and can be parallelized with the technique we propose in the next chapter.

This approach requires auxiliary array to temporarily place items and these doubles memory consumption - so it can be less attractive as a substitute of classical quicksort.

In case 3-Way radix quicksort, however, we are targeting long keys and efficiently sort only pointers to these keys, which makes same spaw-less approach quite appropriate.

4.2.4 Parallelization Scheme

The naive approach to parallelization of a recursive algorithm would be to use task parallelism for every recursion branch (fig 4.3). Thus we will be doubling the number of parallel threads on every level of recursion. To partition strings N into the buckets we need to scan N symbols and then partition each of the sub-buckets the same number of symbols in total until the process start encountering empty buckets, i.e. the amount of workload is the same for every iteration and takes $O(N)$ time in total. If the first iteration is only done by one thread and the second by σ (alphabet size) parallel threads

etc. - possible speed-up of such an implementation is obviously limited and existing implementations show limited scalability (97).

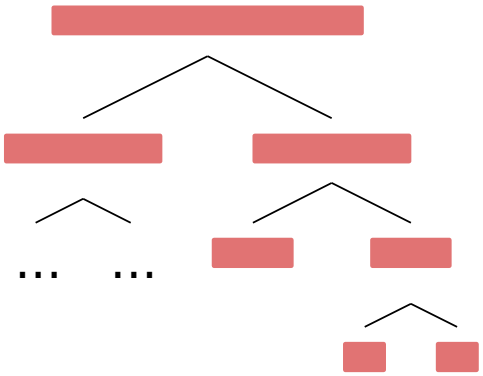


Figure 4.3: Recursive Bucketing: First Iteration Has Maximum Work

This approach is even less efficient for GPU than for the classical multi-core architecture, since one thread on a GPU is relatively slower and high performance is achieved only when thousands of threads are running in parallel.

Another approach would be to parallelize every iteration of the algorithm, as is typically done for LSD radix sort. To build each bucket we are basically counting symbols and then moving string pointers according to the counters. Counting can be efficiently parallelized within multiple threads when the workload is split into chunks for each thread to process.

The problem with this approach is that it performs well in the beginning of the recursive execution when the buckets are relatively big, but as they get smaller processing small amounts of data with multiple threads becomes a waste of resources. However, by this time we already have enough buckets to make use of the model in which one thread or a small group of threads are processing one bucket.

Combining the two approaches would allow us to keep all the processor cores busy the entire time of execution. We suggest starting with parallel partitioning strings into buckets and then, when buckets are small enough, continuing to process each bucket independently in parallel. In the case of many-core CPU we seamlessly switch from one model to another and continue sorting until all the strings are in place. But on GPU having many small unsorted buckets creates performance issues due to branch divergence rates; so we suggest to use a hybrid approach in which the last stages of sorting are always performed on CPU. Moreover, when the recursion branches are independently executed by parallel threads we can seamlessly switch from MSD radix sort to 3-way radix quicksort algorithm.

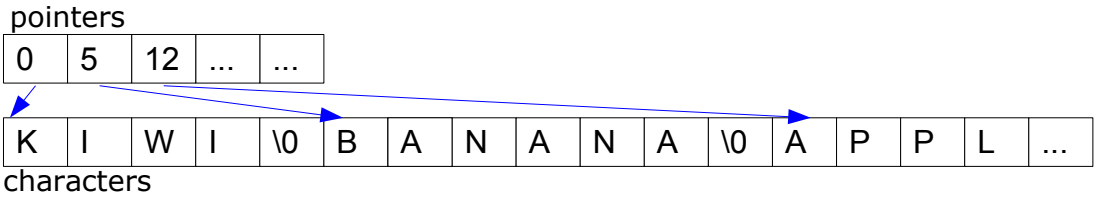


Figure 4.4: String Array in Memory

For the second stage of the sorting Three-way MSD Radix Quick-sort algorithm can be used interchangeably with MSD Radix sort. Also known as multikey quicksort it is a combination of radix sort and quicksort. Algorithms picks an element from the array (the pivot) and consider the first character (key) of the string (multikey). Then remaining elements are partitioned into three buckets: those whose corresponding character is less than, equal to, and greater than the pivot's character. "less than" and "greater than" partitions are sorted recursively on the same character, while for the "equal to" partition the algorithm proceeds to the next character (key). Given we sort using bytes or words of length W bits, the best case is $O(KN)$ and the worst case $O(2^K N)$ or at least $O(N^2)$ as for standard quicksort, given for unique keys $N < 2^K$, and K is a hidden constant in all standard comparison sort algorithms including quicksort. This is a kind of three-way quicksort in which the middle partition represents a (trivially) sorted subarray of elements that are exactly equal to the pivot.

The following section describes our implementations for GPU and CPU and discusses different aspects of our model that also influence performance, such as data transmission costs, workload balance, and divergent branching in sorting code.

4.3 Implementation

For the many-core CPU implementation we used OpenMP. Our GPU solution is implemented in CUDA, NVIDIA's programming platform for general-purpose computing on GPUs which is the current industry standard.

One of the resources we could use to increase the efficiency of radix sort is to process groups of symbols instead of one symbol at a time. The amount of symbols that could be processed simultaneously is only limited by the available memory and the length of the alphabet. Generally speaking, the amount of buckets in radix sort with N symbols

in the alphabet is equal to: S^M where S is the alphabet size and M is the length of the group. Assuming that we use 64-bit integers as counters it is easy to estimate how much memory the program will require for storing buckets: $8 * S^M$ bytes.

Furthermore, shorter alphabets with the same amount of memory will allow for processing of more symbols per iteration. A good example of an area that uses such data is genomics, where the alphabet consists of four nucleotides coded A, C, G, and T (some databases also use N for inconclusive read results). In such a case sorting six symbols at a time would require only $4^6 * 8$ bytes which would take up only 15 Kb, which is insignificant compared to the gigabyte-sized strings to be sorted. Moreover, this amount of buckets is already sufficient to start the parallel sorting by buckets on the next stage.

4.3.1 CPU Implementation

For data-parallel part (*counting* and *move*) we can use OpenMP loop parallelization to split the workload between threads. Each thread is then accumulating its own array of counters for each symbol or a group of symbols. The next step is to perform reduction on the resulting counters. Since OpenMP does not support reduction in arrays we perform it manually in parallel fashion, each thread having access to local counters of all the other threads. Obviously, we multiply the amount of memory we need for the counters by the number of threads, which further limits the length strings we can process. Also with more threads we have to reduce more data (although this is partly compensated by the fact that we have more threads). Being memory-bound, this stage exhibits work time inflation, but still the reduction stage is much smaller than the counting stage, and the performance degradation is insignificant.

In the second stage of the algorithm we use OpenMP task parallelism. Here we switch to 3-way radix quicksort instead of MSD radix sort.

4.3.2 GPU Implementation

The execution model on GPU is very different from that of traditional multi-core systems. While GPUs provide a much higher level of parallelism (new cards boast as many as 512 SM cores per die), programs are executed in the so-called Single Instruction Multiple Threads (SIMT) paradigm. This means that threads on the same multiprocessor are performing the same instructions at the same time, and this does not allow us

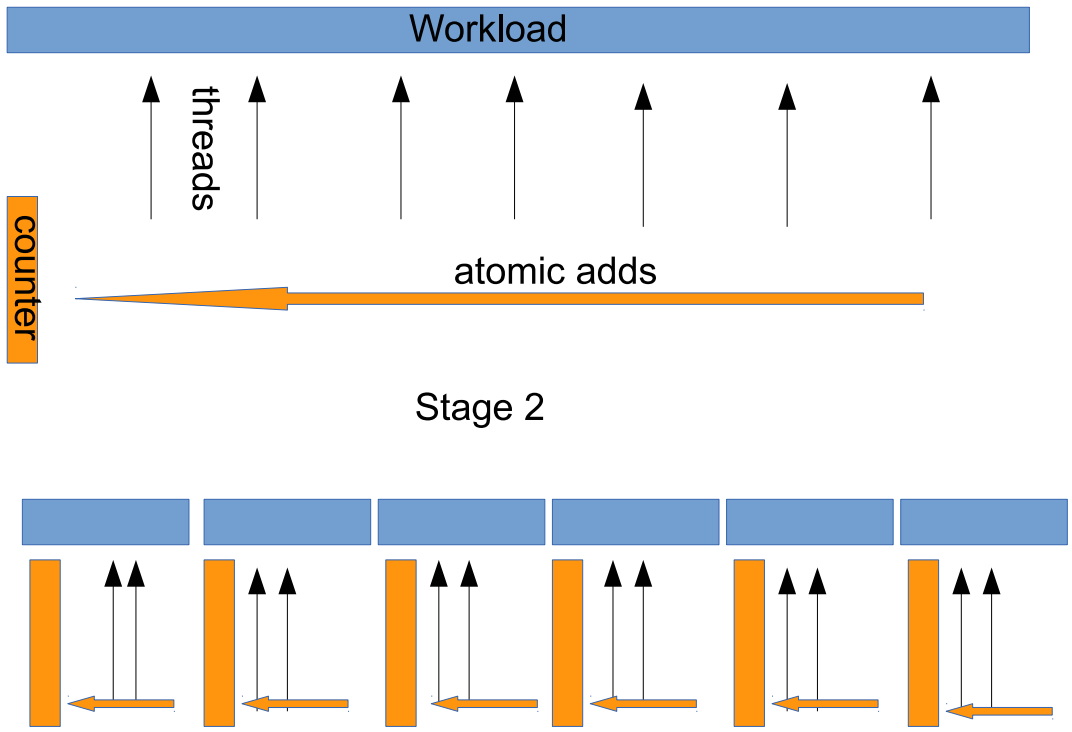


Figure 4.5: First stages of sorting kernel

to run independent tasks on different cores (although it is possible to launch several parallel kernels at the same time).

Recursive algorithms are challenging to implement on GPUs because, as of now, only the newest Kepler architecture currently support recursion, and that support is rather limited. Recursive launch of kernels is used to bring control of their execution entirely to GPU, but it is not meant to be used in highly recursive algorithms. Another hardware characteristic to be considered is that programs on GPU can address only on-board GPU memory which has relatively high bandwidth, but the data has to be transferred from host memory and back, which is relatively slow.

As described earlier, our algorithm is executed in three stages:

(1) processing the first (biggest) buckets in data-parallel fashion; (2) processing the resulting (smaller) buckets in combination of data-parallel and task-parallel paradigms; (3) sorting the remaining small buckets with CPU threads.

To implement the first stage of the algorithm we made heavy use of atomic opera-

tions which allowed us to avoid reduction and to thus decrease memory usage (which would be significant with many counters per thread). This approach does not slow down our performance due to highly efficient implementation of atomic operations in the newest GPU architectures.

As was mentioned above, the efficiency of radix sort could be increased by sorting groups of symbols instead of one symbol at a time. With the use of atomic operations, the amount of symbols that could be processed simultaneously is only limited by the available memory and the length of the alphabet. Generally speaking, the amount of buckets in radix sort with N symbols in the alphabet is equal to: S^M where S is the alphabet size and M is the length of the group. Assuming that we use 64-bit integers as counters it is easy to estimate how much memory the program will require for storing buckets: $8 * S^M$ bytes.

The second stage of the algorithm repeats the same logic, but with smaller groups of threads independently processing different buckets. This is more efficient, since the buckets get smaller at this stage. On the other hand, since the groups of threads now have their own counters this limits the amount of groups that can be executed in parallel. We balance these parameters to keep GPU cores saturated.

At the point when the distribution of buckets exhibits high workload imbalance which we cannot cope with on GPU, and the sort is continued on CPU. There we can use 3-way radix quicksort which is a more advanced version of the same algorithm.

4.4 Performance Analysis and Optimization

We used the following hardware configuration for performance evaluation. System 1:

- CPU: Intel(R) Xeon(R) CPU E5-2687W 0 @ 3.10GHz, 16 cores with hyper-threading, 2 sockets
- OS: CentOS release 6.4 (Final)
- Memory: 126 GB

System 2:

- Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz, 6 cores with hyper-threading, 1 socket

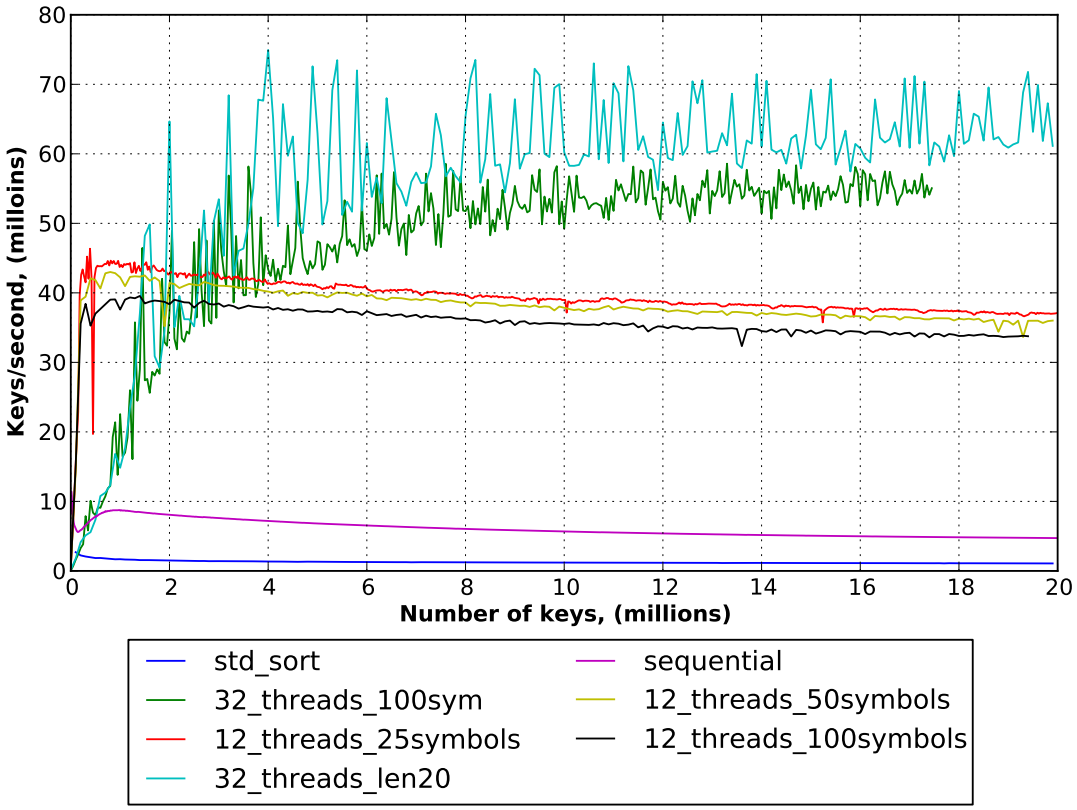
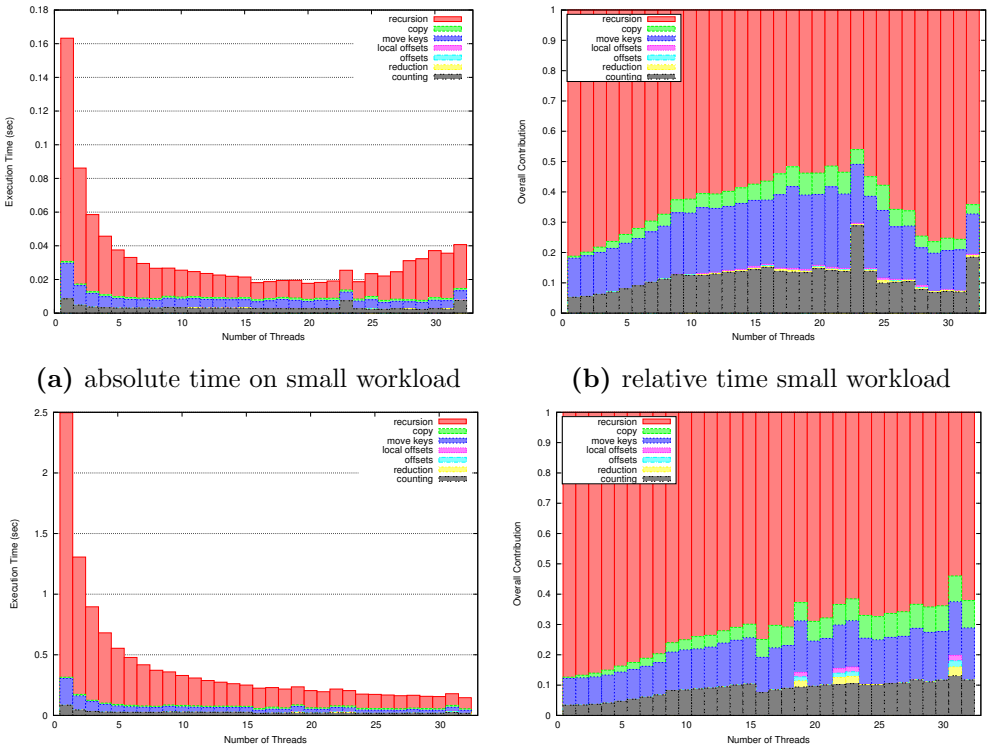


Figure 4.6: Sorting throughput

- OS: Scientific Linux release 6.1 (Carbon)
- Memory: 126 GB
- GPU: Tesla K20c
- GPU compute capability: 3.5
- GPU memory: 4.6 GB

4.4.1 CPU Implementation Performance

First of all we evaluated overall performance of the implementation in terms of sorting throughput. Figure 4.6 shows sorting throughput on different number of keys. We observed stable performance when all the threads are placed on the cores belonging to one socket and irregular, though insignificant jumps when threads are scattered



(a) absolute time on small workload

(b) relative time small workload

(c) absolute time on big worlkoad

(d) relative time on big workload

Figure 4.7: Time spent in different phases

across the cores. Analysis of time spent for each phase of the algorithm shows that the reduction phase is causing synchronization overheads.

While measuring how the implementation responds to the increase of threads number we used numactl tool to control thread allocation and forced threads to first utilize all cores of the first socket and then start using the second one. Figure 4.10 shows exact time spent in both stages of the algorithm for 1-socket system and Figure 4.7 gives more details about each phase for relatively small (1 million of keys) and sufficiently large (10 million) workloads for 2-sockets system. Figure 4.9 shows overall performance scaling for 1-socket and 2-sockets systems.

To understand the performance of the parallel CPU implementation, we instrumented the code to track the execution of each phase as well as the work executed within each phase. For instrumentation we used low-overhead instrumentation library called LoI[1]. LoI collects timings for kernels and phases using only timestamp counters,

[1] `https://bitbucket.org/mpericas/loi-pub`

and it reports information on average kernel times, variation between kernel times of different threads, phase execution times, and overheads. In this context, *overheads* refers to the amount of time within a phase in which threads are not executing kernels. It is a measure of both runtime overheads (such as OpenMP API calls) as well as load imbalance.

By evaluating average kernel times for single-threaded and multithreaded executions it is also possible to measure the work time inflation suffered by the kernels due to resource sharing. Both factors can be encoded as two factors OVR_N and WTI_N which represent the execution stretch compared to *ideal* serial execution at N threads:

$$T_{\text{parallel}} = \frac{T_{\text{serial}}}{N} \times \mathrm{OVR}_N \times \mathrm{WTI}_N$$

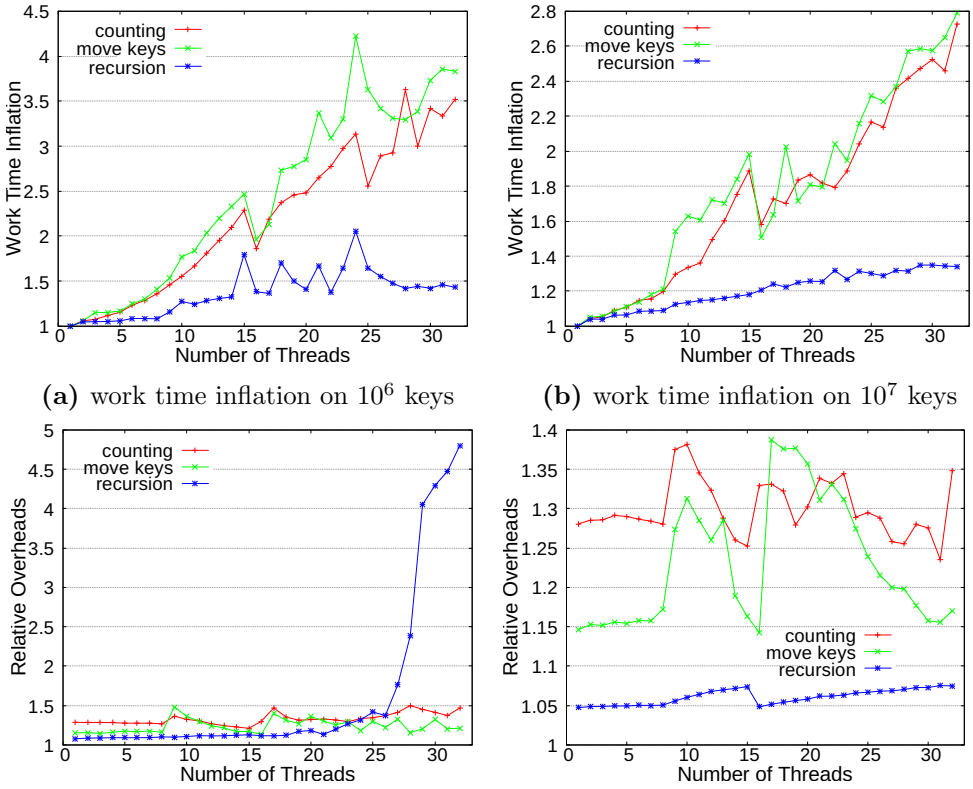
By measuring these values we can obtain a fast qualitative analysis of the bottlenecks present in each phase of the current code.

The figures 4.8 show the results obtained when instrumenting the three main phases and kernels of the code: *counting*, *move keys* and *recursion*. As we can see, data-parallel phases are more subjected to work time inflation both on small and large workloads task-parallel due to more intensive use of synchronization.

Straight-forward task-parallel implementation version of the algorithm showed inferior performance and scalability. We also implemented parallel version of 3-way radix quick sort in a similar mixed-parallelism fashion with similar performance.

4.4.2 GPU Implementation Performance

For the GPU implementation analysis of hardware counters done with Compute Visual profiler shows that sort kernel is memory bound and uses only few percent of available memory bus bandwidth. Organizing memory access patterns in a way that writes and loads are or at least localized is a one of the fundamental optimizations for CUDA kernels. In our algorithms though, we are examining i_{th} symbol of every variable-length string and strings are occupying continuous span in global memory. It is very difficult to organize efficient memory access in this context. Few things we can to do are two disable L2 cache with compiler options and prioritize L1 cache size over available shared memory with CUDA API call. Then instead of loading consequent symbols one by one for reading the prefix of the string we do one 32 bytes memory load into local buffer and then iterate over its. This gave us 15-20 performance improvement.



(a) work time inflation on 10^6 keys

(b) work time inflation on 10^7 keys

(c) overheads on 10^6 keys

(d) overheads on 10^7 keys

Figure 4.8: Performance details

Figure 4.10 shows exact time spent in each phase of the algorithm. *copy* is moving string from host to device, *count* is counting the number of occurrences for each prefix, offsets" is prefix sum of the counters and "move_keys" denotes scanning strings for second time and placing corresponding pointers according to offsets.

If we continue recursive sorting past the level when buckets are getting small enough - high branch divergence starts causing performance degradation. We found that optimal cut off level is about 6-8 symbols for small alphabets (like 4 symbols of genomic data) and 3-5 symbols for longer alphabets.

But the definite bottleneck for GPU implementation is moving data to and from the device -it takes considerable share of time and it grows proportionally to the length of the key. Figure 4.11 shows maximal performance we can get depending on the size of the key. Top green line shows maximal throughput if we only have to copy data to GPU. Line labelled *copy_kernel* accounts for moving sorted pointers back and also one kernel launch. Kernel launch overhead is about 20 microseconds even if it does not perform any work and we obviously need to launch at least one kernel. So this is the best possible performance for such an implementation.

CPU implementation, on the other hand, has fixed performance irrelevant to the length of the keys - only to the overall number and statistical properties like the size of the alphabet.

To overcome this impediment we used the following technique. We chopped of first symbols of every string and repartitioned them into new memory block along with the pointers to original location. Then we transferred only this part to GPU and performed MSD radix sort there. Though the keys are seemingly fixed-length now, at least for the GPU part - we can not use LSD radix sort, as its every iteration is oblivious of previous iterations and those information about partitioning is not preserved. After N iterations of MSD radix sort on the other hand we have strings sorted by N first symbols and also start and end position of every bucket as a by-product of an algorithm.

This re-partitioning of strings of course is bringing additional overhead, but it is justified by overall performance improvement except for extremely short keys. We also parallelized re-partitioning process on CPU using OpenMP, although this process is memory-bound and does not scale much. New distribution of execution time is shown on Figure 4.13b.

4.4.3 Alternative Approach to GPU String Sorting

Another work on implementing MSD Radix sort for GPU emerged after the submission of the first draft of this thesis. (98)

While using the same core algorithm authors propose different implementation. To maintain good workload balance all the keys on the same level of recursive bucketing tree are processed simultaneously.

To enable this, instead of addressing parts of keys in the contiguous memory block where keys are stored, authors move two a portion of a key to the new memory block, append the pointer to the first byte of the key, and extra field with bucket id to differentiate between buckets. Then thrust radix sort primitives are applied to sort these two symbols with the respect of bucket ids segmented scan. Then buckets of size one or zero are excluded from the list, bucket ids are updated. Finally next two symbols are loaded into the same space and the process is repeated until no buckets are left. This approach requires auxiliary space for copies of pointers, fragments of the keys and bucket ids.

The algorithm provides a very different and interesting approach to implementing MSD radix sort. In a way it *emulates* standard recursive bucketing by keeping buckets id's and pay for better workload balance with additional memory transfers. Also, used of thrust prefix sum primitives allowed for a short clean implementation, particularly by "outsourcing" grid/block side tuning to the library.

Our evaluation of this solution showed that it delivers good sorting throughput on randomized reasonably short (about 20 bytes) keys - but as the length of the keys grows the PCIe communication overheads start to dominate over sorting efficiency. Secondly, the Thrust segmented scan primitive is not very efficient when there are many segments inside the warp of 32 threads. Simple example, when the input data set contains randomized keys and each key is repeated two times, i.e. there are many buckets of two strings requiring deep recursion to sort - solution by Deshp and Narayanan shows significant performance degradation down to about $15*10^6$ keys/second, which is several times slower than our implementation.

4.4.4 Skewed Keys

The algorithm itself is optimal in a sense that it observes only as many characters as necessary to distinguish unique keys and does it constant number of times - two in our case. If the keys are distributed so that they form big amount of small buckets - algorithm switches to task-parallel model and sort them separately. If there are big buckets left - they are processed in the same thread-parallel manner, both parts of the algorithm showed good performance.

In the worst case we have to examine every symbol of every key, this can happen is we have square matrix of N keys of size N, filled with the same character. In this case for all comparison-based algorithm complexity of comparison operation would be $O(N^2)$ and the best overall performance we can get would be $O(N^2 \log N)$, so MSD Radix Sort still delivers the superior performance in this case.

The key point is that algorithm can dynamically choose different branches of execution depending on the current bucket size and thus utilize SIMD-fashion parallelism inside the recursive bucketing routing if it is needed while sequentially processing smaller buckets to avoid task creation overheads.

4.5 Conclusion

To the best of our knowledge, this is the first[1] attempt to parallelize a sorting algorithm efficient for the processing string data. We presented our implementations of MSD radix sort for two parallel architectures (CPU and GPU). Our solution features a two-stage algorithm that balances different parallelization strategies to achieve good scalability. Performance analysis confirmed that MSD radix sort can be efficiently parallelized and achieve high performance on string data, especially on data with shorter alphabets.

Our implementation also showed that when the keys are longer, MSD radix sort outperforms competing algorithms such as merge sort.

We analysed performance to validate our approach and used locality optimizations. For GPU implementation we identified host-to-device communication overhead as a bottleneck and introduced a communication-reducing strategy to overcome this issue.

[1] Work by Deshp and Narayanan(98) was published after the submission of the first draft of this thesis

More research needs to be done on testing MSD radix sort on other parallel architectures, such as MIC. Another possible direction for further work is testing the applicability of this algorithm to distributed memory systems.

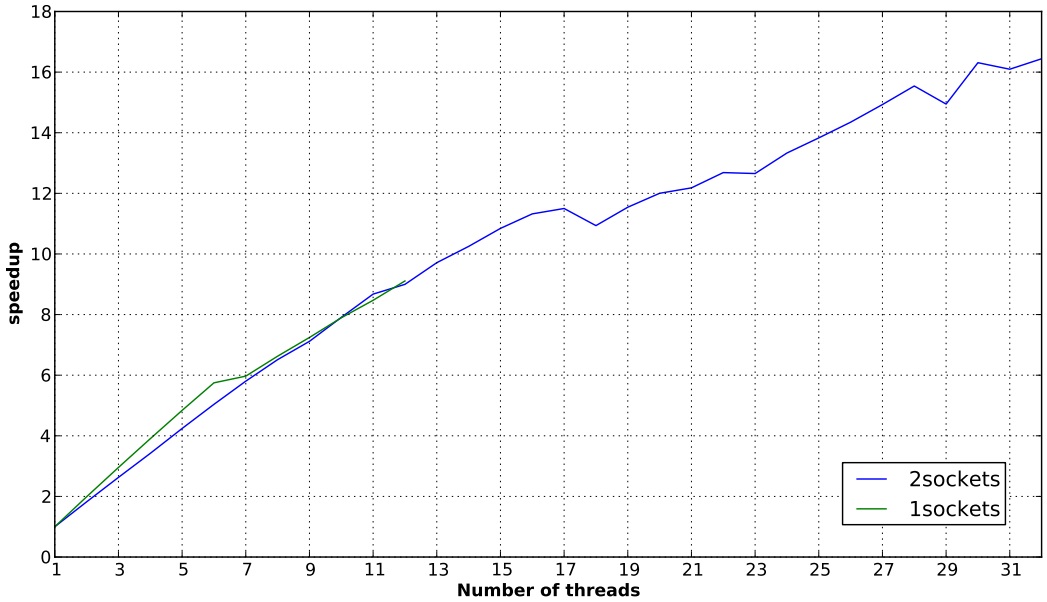


Figure 4.9: Scaling

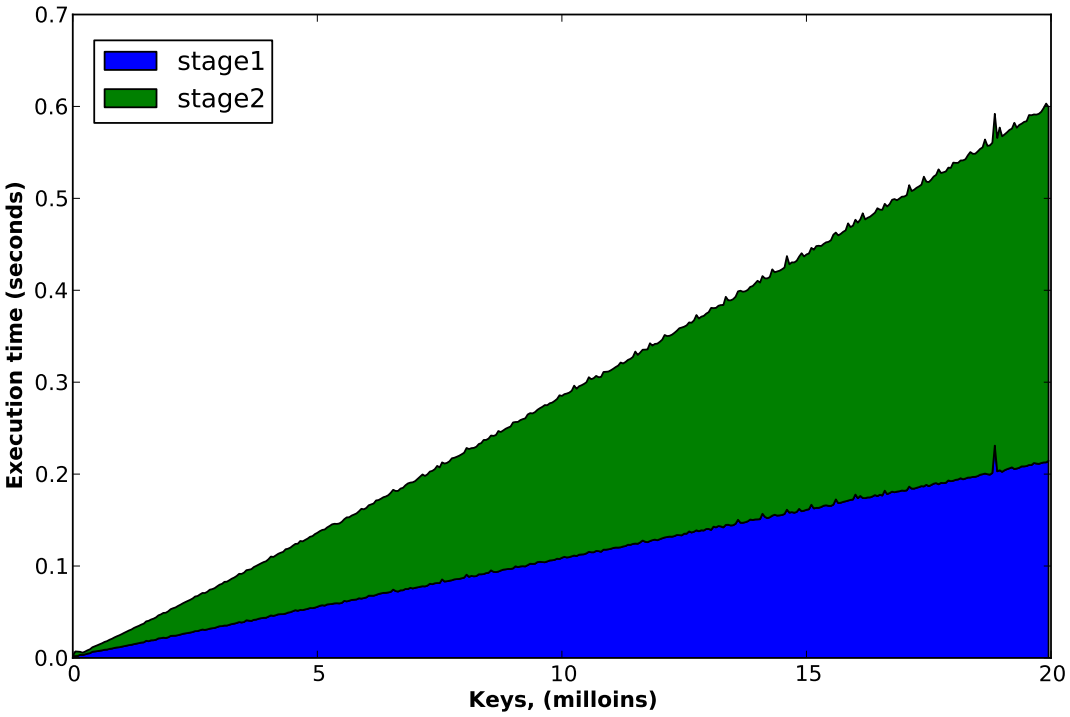


Figure 4.10: Execution time breakdown for CPU implementation

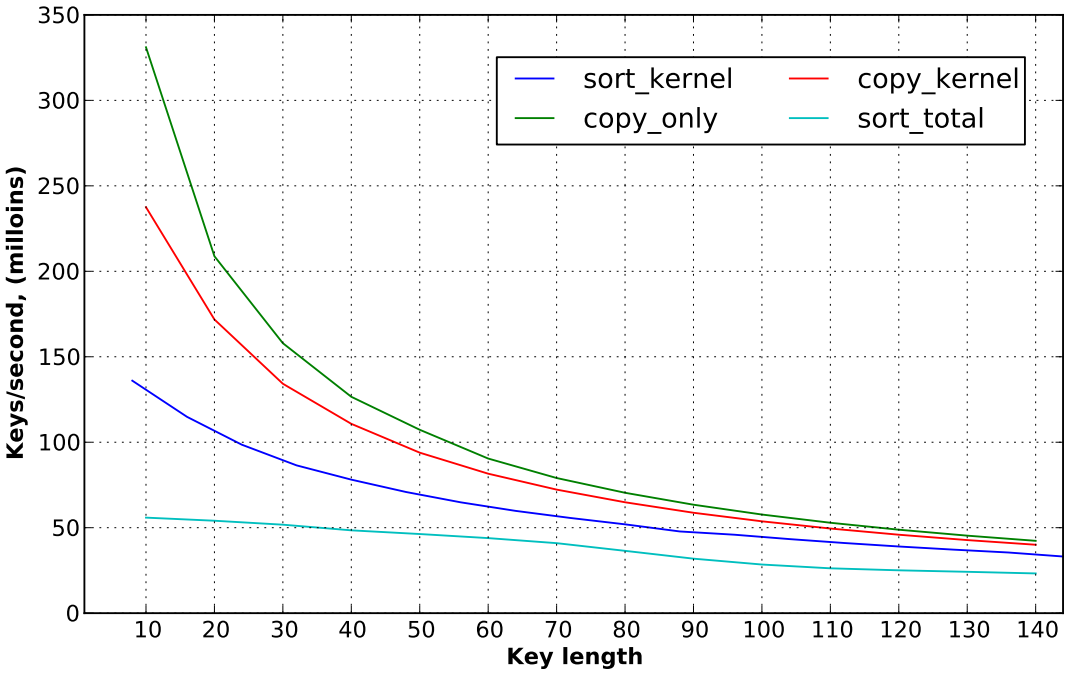
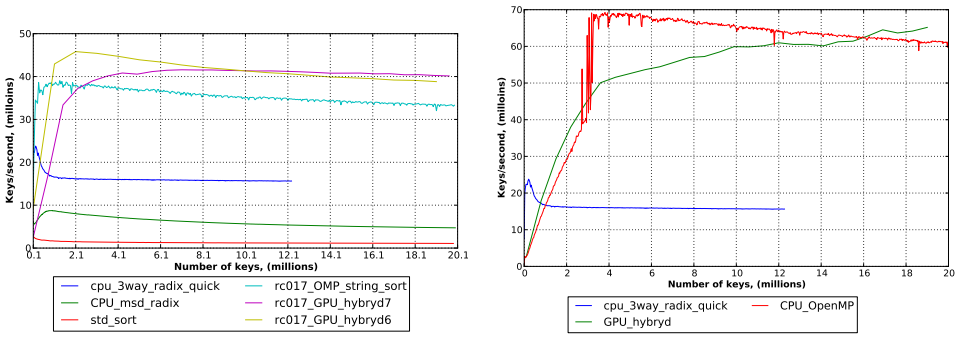
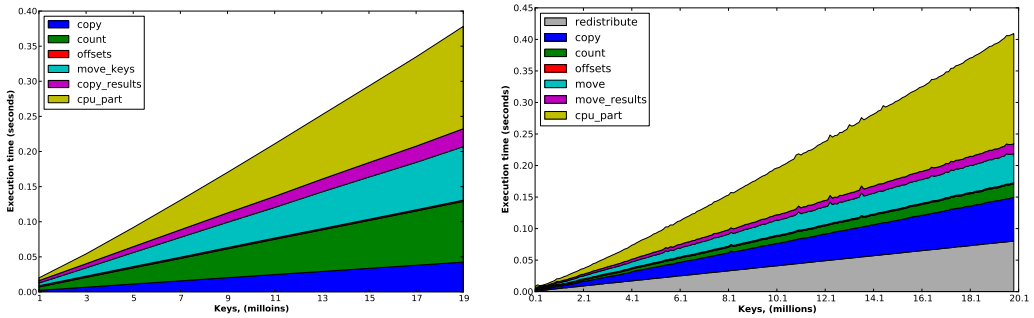


Figure 4.11: Correlation of performance and key length



(a) absolute time on small workload **(b)** relative time small workload

Figure 4.12: GPU sorting throughput



(a) first implementation **(b)** Implementation with re-partitioned strings

Figure 4.13: GPU execution time breakdown

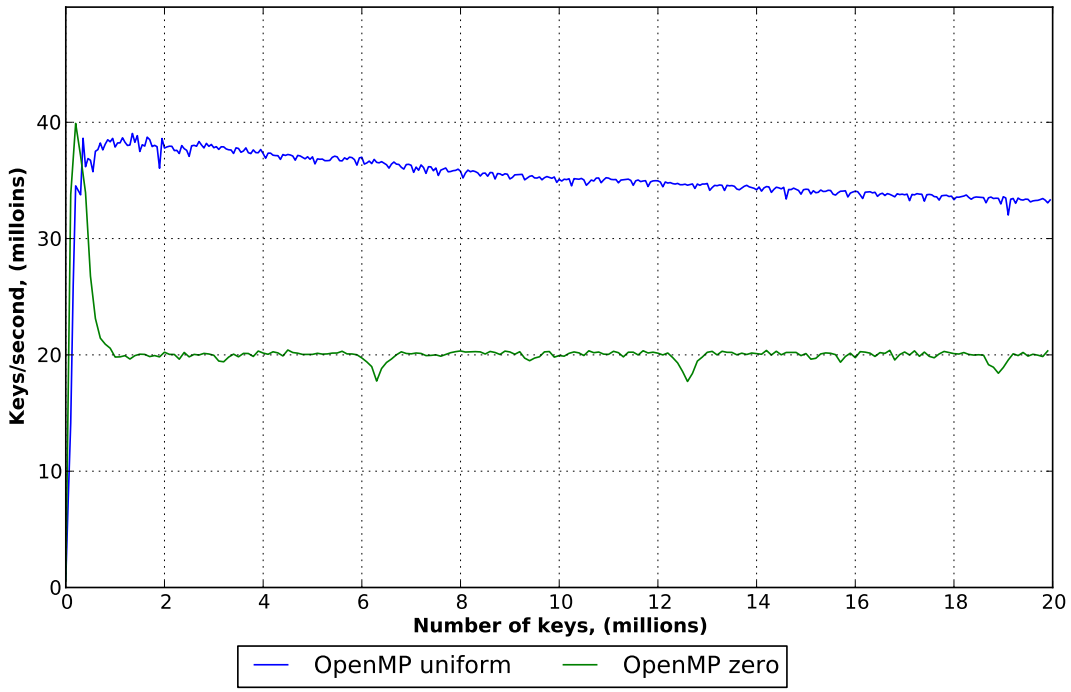


Figure 4.14: Sorting keys with zero-distribution

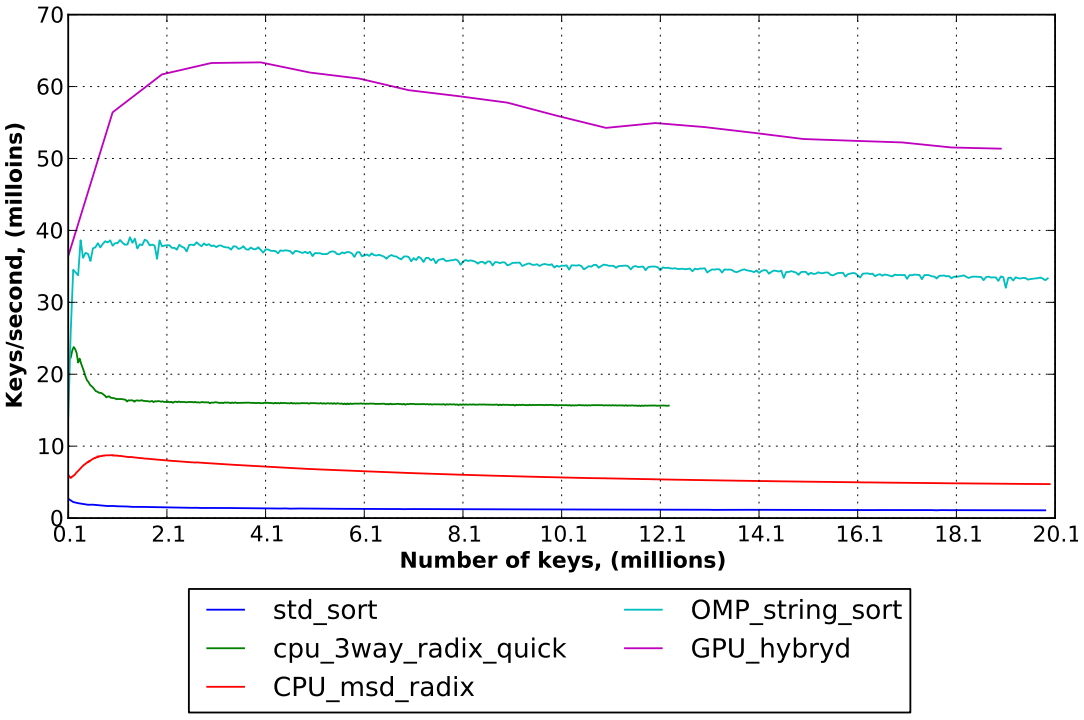


Figure 4.15: Sorting throughput of improved implementation

5

Discussion and Conclusions

In this work we have presented case studies of two important data-intensive problems of computational science: sequence alignment and string data sorting. In additional to theoretical analysis and mathematical modelling, we implemented our ideas for GPU architecture.

This section outlines the main findings from implementation experience and performance evaluation of our solutions, some of which are relevant not only to the applied areas under consideration, but also to the supercomputer programming in general. We and also discuss the implications of our findings for computational biology and string data analysis and suggest some directions for future work in optimization of data-intensive algorithms for hybrid architectures.

5.1 Main Findings

The main application-specific findings from our two case studies can be summarized as follows.

For the string sorting problem, we have shown that MSD radix sort can be efficiently parallelized and achieve high performance on string data. We have shown that this algorithm is particularly beneficial for data with shorter alphabets, like the one found in genomics or proteomics. We have also found which diapason of key lengths can be efficiently sorted on GPU and how we can apply the key-splitting strategy to sort keys of any length with a hybrid approach.

For the sequence alignment problem, our implementation has shown that using more compact data structures can lead to performance improvement in short read alignment problem. In our case, replacing suffix array with BWT in MummerGPU++ and rewriting the corresponding search algorithms increased the performance by 3-4 times.

We showed that smart strategy for allocation shares of available memory for different parts of a workload can further boost performance and that the best performance per device for read alignment problem can be achieved by using multiple GPUs. The optimal number of GPU devices for a particular task can be estimated from reference size.

The problems presented in this dissertation involve different algorithms for performing different tasks on different kinds of data. Nevertheless, in addition to application-specific results we can also observe some general tendencies which include the following.

Firstly, the efficiency of the algorithm highly depends on how well it matches a given architecture. This makes the choice of the algorithm a key decision in writing a data-intensive application.

Secondly, not all algorithms can efficiently use the benefits of GPU acceleration throughout the time of execution. This means that hybrid approaches can often lead to better results than relying only on GPUs.

Thirdly, our GPU implementations allowed us to identify the host-to-device communication overhead as a bottleneck which requires special strategies to reduce the time costs.

In this chapter we will focus on the three general tendencies outlined above, which present interest not only for the applied areas under consideration (computational biology and string data sorting), but also to other data-intensive areas of supercomputer programming.

5.2 Fitness of Algorithms and Architectures

Our results show that it is important to select the algorithm and data structure so that it would match the given architecture. It is often the case that several algorithms can be applied to solve a particular problem. Such areas include matrix multiplication, numeric methods, and, in our case, pattern matching and sorting. In some cases even

higher complexity of the algorithm itself can allow for a better overall performance by allowing, e.g., better cache efficiency or lower host-to-device communication expenses.

This point is well illustrated by choosing the basic data structure for our implementation of sequence alignment algorithm. While suffix tree is theoretically the fastest data structure for substring matching, high memory consumption is making its practical implementation less efficient, particularly for GPUs. Using BWT allowed us to increase the performance by several times as it is much smaller than suffix array and, subsequently, suffix tree. Its small size also allows us to process the same amount of data in one pass. With the low amount of memory available on GPUs, other data structures would have forced us to split the data to be processed in several chunks.

The reason for efficiency of BWT in our implementation is not only memory efficiency but

5.3 Host-to-device Communication Expenses

Performance degradation due to high costs of data transmission between different levels in memory hierarchies is clearly a performance bottleneck for many applications. This was evident in the case of string sorting: moving long keys through the PCIe bus was making the otherwise efficient standard version of Most Significant Digit radix sort algorithm slower than its sequential counterpart.

Therefore programming for GPUs requires being conscious about data movements to make the algorithm much more efficient. We see several possible ways to reduce the time spent on data transfers.

- Careful selection of the algorithm. With GPUs, it makes sense to decide on a relatively computationally complex algorithm if it allows to reduce communication costs.

- Compressing the data before it is sent (given that one does not spend too much time on compressing and decompressing).

- Overlapping of computation and communication. In our sequence alignment implementation sending a chunk of query data was possible when another chunk is being processed.

- Algorithm-specific ways to reduce communication costs might be available depending on the task.

The last point is illustrated by our implementation of the sequence alignment method. The data that we are processing consists of two different part: reference and queries. Both parts have to be split in chunks to fit the device memory and be processed in pairwise fashion.

The analysis of application behavior for the case of workload size considerably exceeding device memory proves that higher performance can be achieved by intelligent strategy for data decomposition.

5.4 Hybrid Algorithms

As described above, GPUs and traditional architecture provide different performance characteristics, and different algorithms can exhibit drastically different fitness to a given architecture. Some algorithms are so well adapted to GPUs that the role of CPU in hybrid-architecture tailored solution consists only in loading data, transferring it into GPU and then retrieving the results back. However, in other cases both sides can contribute substantially to the overall application performance.

The present study highlights three such characteristic cases:

1. When algorithm is composed of several sub-routines, and some of this routines are well-suited for GPU architecture, and some are not, but can be efficiently executed on CPU.

2. When CPU is used to compress/uncompress data to minimize the host-to-device communication overhead.

3. When algorithm exhibits different efficiency on particular architectures during different stages of execution.

These thee cases represent possible nuances rather that distinct, non intersecting cases, and can co-occur in one problem. Such was our case with string sorting: different architectures were used at different execution stages, and CPU was used to smartly reformat data to decrease data transfers.

The first scenario is, perhaps, the most straightforward. GPU-friendly tasks are running on GPU, while highly recursive tasks or tasks with high memory requirements are running on the host side. In our example the suffix array is constructed on CPU. Linear-time suffix array construction algorithms are available, but they are difficult to implement on GPU due to their highly recursive nature.

Another good example can be found in two GPU-accelerated implementations in Fast Multipole Method, a mathematical technique that was developed to speed up the calculation of long-ranged forces in the n-body problem. One implementation was presented by Hu et al. in 2011 (99), and another - in 2012 by Lashuk (100). FMM algorithm is composed from several phases and authors determined which phases are GPU-friendly, particularly the so-called *P2P* phase (the native N-body computation).

A good example of the second scenario is the compression of BWT-based index we use for sequence alignment. Compared to the plain uncompressed index we can achieve up to 4x times better space efficiency. If the workload is too big to be processed in one device memory we have to move data across the PCIe bus several times, thus multiplying the beneficial effect of the compression.

The third case can be observed in Most-Significant Digit radix sort algorithm described in the 3d chapter of this work. This algorithm is recursively partitioning strings into buckets. While buckets are big enough, SIMD (or SIMT) execution model is working well. But when the buckets get smaller, the CUDA code starts to suffer from high branch divergence and poor workload balance.

Finally, it is possible to address these nuances simultaneously. Again, in MSD radix sort the transferring of long keys to device is seriously taxing performance. By splitting keys into segments to be sorted on GPU and to be "finished" on a host side we are solving two problems simultaneously: we separate the parts with good and with poor workload balance to be executed on GPU and on CPU side and we minimize the amount of data to be transferred through the PCIe bus.

Of course, the strategies for a particular implementation depend mostly on the characteristics of a given algorithm, and it is impossible to predict which technique would work best for all problems. Nevertheless, we have outlined some general strategies for implementations for hybrid architectures which could be useful in other specific cases.

5.5 Implications for Computational Biology

New-generation sequencing machines allow for cheaper and faster DNA sequencing, which leads to increasing amount of genomic data. This calls for more efficient software infrastructure for storing and processing this data. Our work demonstrated that GPUs can be helpful for such tasks.

The original MUMmer software is being used in actual work with genetic data (50) . It was further refactored by Schatz et al. (59) to use GPU acceleration. The latest GPU-based implementation by (61) that uses suffix array achieves about $8 * 10^6$ keys/s performance. Using BWT instead of suffix array and smart memory partitioning strategy allowed us to further boost performance by 3 times. At the same time we preserved compatibility with the original MUMmer software, which allows our solution to be directly usable in actual data analysis.

Dealing with genomic data always implies handling gigabytes of information, which requires the algorithms to be efficient in terms of I/O, networking and intra-node communications. Memory-conscious approaches are preferable. We showed that BWT can be efficient and reliable data structure for GPU implementation of exact-match problem, and it could be efficient for other classes of matches, particularly those allowing for mismatches in the sequences.

5.6 Implications for String Data Analysis on GPU

Supercomputers have originally been mostly used for number-crunching (mostly for double precision floating point values). However we now witness the tendency of using supercomputers for processing data; moreover, supercomputers are being converged with data centers. Such data can include genomic sequences, natural language texts, different keys, hashes, etc. This means that we need efficient string-processing algorithms which would be friendly to massively parallel hybrid architectures. This work presented one of such algorithms, namely the string sorting.

We have shown that the benefits of GPU architecture can be used for processing string data. However, to keep it efficient, the programmer must pay particular attention to communication expenses, as the string keys are simply longer than numeric data.

Our implementation has shown that the excessive length of keys makes it prohibitive to utilize GPUs. Two ways to deal with this problem are (a) using GPUs to sort only

shorter keys, and (b) use GPUs to sort only parts of the keys, e.g., the first n symbols. In this work we used the second approach to build an efficient GPU implementation of MSD radix sort.

5.7 Implications for Supercomputer Design

As mentioned above, various non-float data are coming to supercomputers. Processing such data can be more taxing on storage and intra-node communications, and this implies that for supercomputers to be efficient for string crunching, we need faster local storage. We also need faster network and inter-node communications, which are important for numerical data as well.

Another consideration is the metric we use for ranking high performance computing systems. The most widely known and recognized supercomputer rating, Top500 list, is primarily concerned with the double-precision floating-point computation performance. It is built based on High Performance Linpack (HPL) - simple program that factors and solves a large dense system of linear equations using Gaussian Elimination with partial pivoting. HPL benchmark provided good correlation between the ranking and the performance of real-world full-scale applications, but now situation is changing. For many new applications which have much lower computation-to-data-access ratios, access memory irregularly or have fine-grain recursive computations performance do not really correlate with HPL benchmark score.

There are other ratings, such as Graph 500 which target different types of computation, but they are more concerned with communication efficiency. However, even in the domain of numerical simulation we observe the tendency of using the single-precision values at some stages of certain algorithms to maximize the general efficiency. Different systems exhibit different balance of single- and double-precision performance. The Tsubame 2.5 system installed in Tokyo Institute of Technology is inferior to the top Japanese system, the K supercomputer, for double-precision floating-point operations (5.7 TFlop/s VS 10.5 TFlop/s), but is more efficient (at 17.1 TFlop/s VS the same 10.5 TFlop/s) for the single-precision floating-point values.

Recently proposed High Performance Conjugate Gradient (HPCG) benchmark is supposed to address some of this new challenges and provide more adequate rating.(101)

Our case studies highly exemplify this tendency. In fact, they both do not involve floating-point operations at all and are purely integer, and we do not have much statistics on the supercomputer efficiency for this kind of computations.

5.8 Directions for Further Work

The solutions presented in this work were developed for GPUs, currently the most popular accelerator architecture. However more research needs to be done on testing MSD radix sort on other parallel architectures, such as MIC which is steadily growing in presence in the HPC world.

As MICs share many features with GPUs, such as the data transfer through PCIe bus, many of the techniques and findings presented in this works should be applicable to this architecture. However, the execution model of MIC is slightly different from that of GPU, and it might require re-evaluation of some algorithms. At the same time, characteristics of this architecture might open up new ways of optimizations for the sequence alignment and/or string sorting problems.

Since our solution of the sequence alignment problem showed considerable performance gains from replacing suffix array with BWT, it could turn out to be equally useful for other applied tasks in computational biology. There are recent works presenting BWT-based algorithms for inexact matching problem, and these could be efficiently adapted on GPU.

References

[1] Stanislaw Marcin Ulam. **Tribute to John von Neumann**. *Bulletin of the American Mathematical Society*, **64**(3):1–49, 1958. 1

[2] Gordon E Moore et al. **Cramming more components onto integrated circuits**. In *Proceedings of the IEEE*, **86**, pages 82 – 85. McGraw-Hill, January 1998. 1

[3] George S. Almasi. *Highly Parallel Processing (The Benjamin/Cummings series in computer science and engineering)*. Benjamin-Cummings Publishing Co.,Subs. of Addison Wesley Longman,US, 1987. 2

[4] Computer Science, Telecommunications Board, National Research Council, Academy Industry Program, and National Academy of Sciences. *Supercomputers: Directions in Technology and Applications*. National Academies Press, 1989. 2

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

[6] S. Huang, S. Xiao, and W. Feng. **On the Energy Efficiency of Graphics Processing Units for Scientific Computing**. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society. 4

[7] J. C. Venter. **The Sequence of the Human Genome**. *Science*, **291**(5507):1304–1351, Feb 2001. 4, 26

[8] Gil A. McVean, Richard A. Gibbs, Jun Wang, Eric S. Lander, Paul Flice, and all. **An integrated map of genetic variation from 1,092 human genomes**. *Nature*, **491**(7422):56–65, Nov 2012. 4, 30

[9] David W. Mount. *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press, 2001. 5, 31

[10] Horst Gietl Hans W. Meuer. **Supercomputers Prestige Objects or Crucial Tools for Science and Industry?** *Software Development Practice*, 2012. 7

[11] **OpenMP Application Program Interface, Version 4.0**. openmp.org. 9

[12] Ewing Lusk, Nathan Doss, and Anthony Skjellum. **A high-performance, portable implementation of the MPI message passing interface standard**. *Parallel Computing*, **22**:789–828, 1996. 9

[13] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, Katherine A. Yelick, Meetings Jim Demmel, William Plishker, John Shalf, Samuel Williams, and Katherine Yelick. **The Landscape of Parallel Computing Research: A View from Berkeley**. Technical report, TECHNICAL REPORT, UC BERKELEY, 2006. 10

[14] Pavan Balaji and Satoshi Matsuoka. **Guest Editors' Introduction: Special Issue on Applications for the Heterogeneous Computing Era**. *IJHPCA*, **27**(2):87–88, 2013. 11

[15] J. Nickolls and W.J. Dally. **The GPU Computing Era**. *Micro, IEEE*, **30**(2):56 –69, march-april 2010. 12

[16] NVIDIA. *CUDA C Programming Guide*, 2013. 14, 17

[17] Khronos OpenCL Working Group. **The OpenCL Specification, Version 1.1**, 2012. 14

[18] R. Dolbeau, S. Bihan, and F. Bodin. **HMPP: A Hybrid Multi-core Parallel Programming Environment**. *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007), Boston, Massachussets, USA*, 2007. 14

[19] OpenACC. **OpenACC Application Programming Interface**, 2012. 14

[20] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Yao Zhang, and V. Volkov. **Parallel Computing Experiences with CUDA**. *Micro, IEEE*, **28**(4):13 –27, july-aug. 2008. 15

[21] Martin Egli and Wolfram Saenger. *Principles of Nucleic Acid Structure*. Springer, 1983. 23

[22] W. MIN JOU, G. HAEGEMAN, M. YSEBAERT, and W. FIERS. **Nucleotide Sequence of the Gene Coding for the Bacteriophage MS2 Coat Protein**. *Nature*, **237**(5350):82–88, May 1972. 26

[23] W. Fiers, R. Contreras, F. Duerinck, G. Haegeman, D. Iserentant, J. Merregaert, W. Min Jou, F. Molemans, A. Raeymaekers, A. Van den Berghe, and et al. **Complete nucleotide sequence of bacteriophage MS2 RNA: primary and secondary structure of the replicase gene**. *Nature*, **260**(5551):500–507, Apr 1976. 26

[24] F. Sanger, S. Nicklen, and A. R. Coulson. **DNA sequencing with chain-terminating inhibitors**. *Proceedings of the National Academy of Sciences*, **74**(12):5463–5467, Dec 1977. 26

[25] A. M. Maxam and W. Gilbert. **A new method for sequencing DNA.** *Proceedings of the National Academy of Sciences*, **74**(2):560–564, Feb 1977. 26

[26] Gilbert W. **DNA sequencing and gene structure**. Technical report, Nobel lecture, December 1980. 26

REFERENCES

[27] F. Sanger, G. M. Air, B. G. Barrell, N. L. Brown, A. R. Coulson, J. C. Fiddes, C. A. Hutchison, P. M. Slocombe, and M. Smith. **Nucleotide sequence of bacteriophage X174 DNA**. *Nature*, **265**(5596):687–695, Feb 1977. 26

[28] M. Adams, J. Kelley, J. Gocayne, M. Dubnick, M. Polymeropoulos, H. Xiao, C. Merril, A. Wu, B. Olde, R. Moreno, and et al. **Complementary DNA sequencing: expressed sequence tags and human genome project**. *Science*, **252**(5013):1651–1656, Jun 1991. 26

[29] R. Fleischmann, M. Adams, O White, R. Clayton, E. Kirkness, A. Kerlavage, C. Bult, J. Tomb, B. Dougherty, J. Merrick, and et al. **Whole-genome random sequencing and assembly of Haemophilus influenzae Rd**. *Science*, **269**(5223):496–512, Jul 1995. 26

[30] Eric S. Lander, Lauren M. Linton, Bruce Birren, Chad Nusbaum, Michael C. Zody, Jennifer Baldwin, Keri Devon, Ken Dewar, Michael Doyle, William FitzHugh, and et al. **Initial sequencing and analysis of the human genome**. *Nature*, **409**(6822):860–921, Feb 2001. 26

[31] Mostafa Ronaghi, Samer Karamohamed, Bertil Pettersson, Mathias Uhln, and Pl Nyrn. **Real-Time {DNA} Sequencing Using Detection of Pyrophosphate Release**. *Analytical Biochemistry*, **242**(1):84 – 89, 1996. 26

[32] Eric Kawashima, Laurent Farinelli, and Pascal Mayer. **Method of Nucleic Acid Amplification**, October 1998. 27

[33] Sydney Brenner, Maria Johnson, John Bridgham, George Golda, David H. Lloyd, Davida Johnson, Shujun Luo, Sarah McCurdy, Michael Foy, Mark Ewan, and et al. **Gene expression analysis by massively parallel signature sequencing (MPSS) on microbead arrays**. *Nature Biotechnology*, **18**(6):630–634, Jun 2000. 27

[34] Marcel Margulies, Michael Egholm, William E. Altman, Said Attiya, Joel S. Bader, Lisa A. Bemben, Jan Berka, Michael S. Braverman, Yi-Ju Chen, Zhoutao Chen, and et al. **Genome sequencing in microfabricated high-density picolitre reactors**. *Nature*, Jul 2005. 27

[35] Stephan C Schuster. **Next-generation sequencing transforms today's biology**. *Nature Methods*, **5**(1):16–18, Dec 2007. 27

[36] Barry Barnes and John Dupre. *Genomes and What to Make of Them*. University Of Chicago Press, 2008. 28

[37] W. Gregory Feero, Alan E. Guttmacher, Christopher J. O'Donnell, and Elizabeth G. Nabel. **Genomics of Cardiovascular Disease**. *New England Journal of Medicine*, **365**(22):2098–2109, Dec 2011. 28

[38] JonathanR. Karr, JayoditaC. Sanghvi, DerekN. Macklin, MiriamV. Gutschow, JaredM. Jacobs, Benjamin Bolival Jr., Nacyra Assad-Garcia, JohnI. Glass, and MarkusW. Covert. **A Whole-Cell Computational Model Predicts Phenotype from Genotype**. *Cell*, **150**(2):389 – 401, 2012. 28

[39] Charles Delisi. **The Human Genome Project**. *American Scientist*, **76**(5):488–493, September 1988. 29

[40] Jeremy Schmutz, Jeremy Wheeler, Jane Grimwood, Mark Dickson, Joan Yang, Chenier Caoile, Eva Bajorek, Stacey Black, Yee Man Chan, Mirian Denys, and et al. **Quality assessment of the human genome sequence**. *Nature*, **429**(6990):365–368, May 2004. 29

[41] R. M. Durbin, D. L. Abecasis, R. M. Altshuler, and all. **A map of human genome variation from population-scale sequencing**. *Nature*, **467**(7319):1061– 1073, Oct 2010. 30

[42] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron E. Lefohm, and Timothy J.Purcell. **A Survay on General-Purpose Computation on Graphics hardware**. *Computer Graphics Forum*, **26**(1):80–113, 2007. 31

[43] Saul B. Needleman and Christian D. Wunsch. **A general method applicable to the search for similarities in the amino acid sequence of two proteins**. *Journal of Molecular Biology*, **48**(3):443–453, Mar 1970. 31

[44] T.F. Smith and M.S. Waterman. **Identification of common molecular subsequences**. *Journal of Molecular Biology*, **147**(1):195–197, Mar 1981. 31

[45] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. **Basic local alignment search tool**. *Journal of Molecular Biology*, **215**(3):403–410, Oct 1990. 32

[46] D. Lipman and W. Pearson. **Rapid and sensitive protein similarity searches**. *Science*, **227**(4693):1435–1441, Mar 1985. 33

[47] Durbin R Li H. **Fast and accurate short read alignment with Burrows-Wheeler Transform**. *Bioinformatics*, **25**(14):1754–1769, 2009. 33

[48] Ruiqiang Li, Chang Yu, Yingrui Li, et al. **SOAP2: an improved ultrafast tool for short read alignment**. *Bioinformatics*, **15**(25):1966–1967, 2009. 33, 40

[49] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. **Ultrafast and memory-efficient alignment of short DNA sequences to the human genome**. *Genome Biology*, **10**(25), march 2009. 33, 40

[50] Arthur L. Delcher, Simon Kasif, Robert D. Fleischmann, Jeremy Peterson, Owen White, and Steven L. Salzberg. **Alignment of whole genomes**. *Nucleic Acids Res.*, **27**:2369, 1999. 34, 36, 53, 92

[51] Mihai Pop. **Genome assembly reborn: recent computational challenges**. *Briefings in Bioinformatics*, **10**:354, 2009. 34

[52] Jonathan M. Rothberg, Wolfgang Hinz, Todd M. Rearick, et al. **An integrated semiconductor device enabling non-optical genome sequencing**. *Nature*, **475**:348–352, July 2011. 34

[53] Paolo Ferragina and Giovanni Manzini. **Indexing Compressed Text.** *Journal of the ACM*, **53**(4):552–581, 2005. 35, 44

[54] Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997. 35, 37

[55] Peter Weiner. *Linear pattern matching algorithms*, pages 1–11. Institute of Electrical and Electronics Engineers, Oct 1973. 35

[56] Edward M. McCreight. **A Space-Economical Suffix Tree Construction Algorithm**. *Journal of the ACM*, **23**(2):262–272, Apr 1976. 35

[57] R. Giegerich and S. Kurtz. **From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction**. *Algorithmica*, **19**(3):331–353, Nov 1997. 35

[58] Martin Farach. **Optimal Suffix Tree Construction with Large Alphabets**. In *38th IEEE Symposium on Foundations of Computer Science (FOCS '97)*, page 137143, 1997. 36

[59] Michael C Schatz, Cole Trapnell, Arthur L Delcher1, and Amitabh Varshney. **High-throughput sequence alignment using Graphics Processing Units**. *BMC Bioinformatics*, **8**:474, 2007. 36, 53, 92

[60] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. **Replacing suffix trees with enhanced suffix arrays**. *Journal of Discrete Algorithms*, **2**:53–86, 2004. 37

[61] Abdullah Gharaibeh and Matei Ripeanu. **Size Matters: Space/Time Tradeoffs to Improve GPGPU Applications Performance**. In *SC '10 Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010. 37, 53, 92

[62] Udi Manber and Gene Myers. **Suffix arrays: A new method for on-line string searches**. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, 1990. 37

[63] Jon Bentley and Robert Sedgewick. **Sorting Strings with Three-Way Radix Quicksort**. *Dr. Dobbs Journal*, 1998. 38, 63

[64] Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg. *Rapid identification of repeated patterns in strings, trees and arrays*, pages 125–136. Association for Computing Machinery, 1972. 38

[65] Ge Nong, Sen Zhang, and Wai Hong Chan. *Linear Suffix Array Construction by Almost Pure Induced-Sorting*, pages 193–202. Institute of Electrical and Electronics Engineers, Mar 2009. 39

[66] Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin. **A taxonomy of suffix array construction algorithms**. *ACM Computing Surveys*, **39**(2):4–es, Jul 2007. 39

[67] M. Burrows and David J. Wheeler. **A block-sorting lossless data compression algorithm**. Technical Report 124, Digital Equipment Corporation, 1994. 39, 40, 55

[68] Su Chen and Hai Jiang. **An Exact Matching Approach for High Throughput Sequencing Based on BWT and GPUs**. In *2011 IEEE 14th International Conference on Computational Science and Engineering (CSE)*. IEEE Computer Society, 2011. 40

[69] Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. **The engineering of a compression boosting library: Theory vs practice in BWT compression**. Technical report, In Proc. 14th European Symposium on Algorithms (ESA 06, 2006. 41

[70] Roberto Grossi, Jeffrey, and Scott Vitter. **Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract**. In *in Proceedings of the 32nd Annual ACM Symposium on the Theory of Computing*, pages 397–406, 2000. 42

[71] Kunihiko Sadakane. **New Text Indexing Functionalities of the Compressed Suffix Arrays**. *J. Algorithms*, **48**(2):294–313, September 2003. 43

[72] Aleksandr Drozd, Naoya Maruyama, and Satoshi Matsuoka. **Sequence Alignment on Massively Parallel Heterogeneous Systems**. In *in proceedings of IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 2498 – 2501, 2012. 49

[73] Aleksandr Drozd, Naoya Maruyama, and Satoshi Matsuoka. **A Multi GPU Read Alignment Algorithm with Model-Based Performance Optimization**. In *High Performance Computing for Computational Science - VECPAR 2012*, 7851, pages 270–277. Springer-Verlag, 2013. 49

[74] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*, **3**. Addison-Wesley Professional, 2011. 57, 60, 63

[75] Jiang Bian, Umit Topaloglu, and Fan Yu. **Towards Large-scale Twitter Mining for Drug-related Adverse Events**. In *Proceedings of the 2012 International Workshop on Smart Health and Wellbeing*, SHB '12, pages 25–32, New York, NY, USA, 2012. ACM. 57

[76] Beth Plale. **Big Data Opportunities and Challenges for IR, Text Mining and NLP**. In *Proceedings of the 2013 International Workshop on Mining Unstructured Big Data Using Natural Language Processing*, UnstructureNLP '13, pages 1–2, New York, NY, USA, 2013. ACM. 57

[77] Thomas H Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3rd ed.)*. MIT Press and McGraw-Hill, 2009 [1990]. 59

[78] Charles Antony Richard Hoare. **Algorithm 64: Quicksort**. *Communications of the ACM*, **4**(7):321, july 1961. 59, 65

[79] Robert Sedgewick and Michael Schidlowsky. *Algorithms in Java, Third Edition, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1998. 60

REFERENCES

[80] Duane Merrill and Andrew Grimshaw. **Revisiting Sorting for GPGPU Stream Architectures**. Technical Report CS2010-03, University of Virginia, Department of Computer Science, Charlottesville, VA, USA, 2010. 61, 62

[81] Peter Sanders and Thomas Hansch. **Efficient massively parallel quicksort**. In *Proceedings 4th International Symposium, IRREGULAR'97 Paderborn, Germany*, pages 13–24. Springer-Verlag, June 1997. 62

[82] Richard Cole. **Parallel merge sort**. *SIAM J. Comput.*, **17**(4):770–785, August 1988. 62

[83] Manwade K. B. **Article: Analysis of Parallel Merge Sort Algorithm**. *International Journal of Computer Applications*, **1**(1):66–69, February 2010. Published By Foundation of Computer Science. 62

[84] K. E. Batcher. **Sorting networks and their applications**. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, **32** of *AFIPS '68 (Spring)*, pages 307–314, New York, NY, USA, 1968. ACM. 62

[85] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. **Scan Primitives for GPU Computing**. In *Graphics Hardware 2007*, pages 97–106, San Diego, CA, August 2007. ACM. 62

[86] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. **Photon mapping on programmable graphics hardware**. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '03, pages 41–50, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. 62

[87] Ujval J. Kapasi, William J. Dally, Scott Rixner, Peter R. Mattson, John D. Owens, and Brucek Khailany. **Efficient conditional operations for data-parallel architectures**. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pages 159–170, New York, NY, USA, 2000. ACM. 62

[88] Alexander Gress and Gabriel Zachmann. **GPU-ABiSort: Optimal parallel sorting on stream architectures**. In *IN PROCEEDINGS OF THE 20TH IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS 06) (APR*, page 45, 2006. 62

[89] Gianfranco Bilardi and Alexandru Nicolau. **Adaptive Bitonic Sorting: An Optimal Parallel Algorithm for Shared Memory Machines**. Technical report, Cornell University, Ithaca, NY, USA, 1986. 62

[90] Naga K. Govindaraju, Nikunj Raghuvanshi, Michael Henson, David Tuft, and Dinesh Manocha. **A cache-efficient sorting algorithm for database and data mining computations using graphics processors**. Technical report, UNC, 2005. 63

[91] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. **GPUTeraSort: high performance graphics co-processor sorting for large database management**. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 325–336, New York, NY, USA, 2006. ACM. 63

[92] Erik Sintorn and Ulf Assarsson. **Fast parallel GPU-sorting using a hybrid algorithm**. *Journal of Parallel and Distributed Computing*, **68**(10):1381–1388, Oct 2008. 63

[93] Daniel Cederman and Philippas Tsigas. **GPU-Quicksort: A practical Quicksort algorithm for graphics processors**. *J. Exp. Algorithmics*, **14**:4:1.4–4:1.24, January 2010. 63

[94] Kamil Rocki Tian Xiaochen and Reiji Suda. **Register Level Sort Algorithm on Multi-Core SIMD Processors**. In *proceeding of the IA^3 Workshop on Irregular Applications; Architectures & Algorithms*, 2013. 63

[95] Jon Bentley and Robert Sedgewick. **Fast algoprithms for sorting and searching string**. In *Proc. Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 360–369, New Orleans, Luisiana, 1997. ACM/SIAM. 63, 65

[96] Robert Sedgewick. **Implementing Quicksort programs**. *Communications of the ACM*, **21**(10):847–857, Oct 1978. 66

[97] Ralf Hoffmann, Matthias Korch, and Thomas Rauber. **Performance Evaluation of Task Pools Based on Hardware Synchronization**. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, pages 44–, Washington, DC, USA, 2004. IEEE Computer Society. 69

[98] Aditya Deshpande and PJ Narayanan. **Can GPUs Sort Strings Efficiently?** In *IEEE High Performance Computing (HiPC), 2013*, 2013. 79, 80

[99] Qi Hu, Nail A. Gumerov, and Ramani Duraiswami. **Scalable Fast Multipole Methods on Distributed Heterogeneous Architectures**. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 36:1–36:12, New York, NY, USA, 2011. ACM. 91

[100] Ilya Lashuk, Aparna Chandramowlishwaran, Harper Langston, Tuan-Anh Nguyen, Rahul Sampath, Aashay Shringarpure, Richard Vuduc, Lexing Ying, Denis Zorin, and George Biros. **A Massively Parallel Adaptive Fast Multipole Method on Heterogeneous Architectures**. *Commun. ACM*, **55**(5):101–109, May 2012. 91

[101] Jack Dongarra and Michael A Heroux. **Toward a New Metric for Ranking High Performance Computing Systems,**. Technical report, UTK EECS Tech Report and Sandia National Labs Report, June 2013. SAND2013-4744. 93