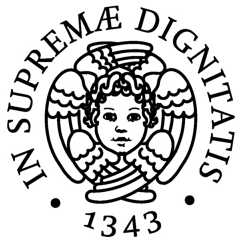


Università degli Studi di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

The Impact of Novel Computing Architectures on Large-Scale Distributed Web Information Retrieval Systems

Gabriele Capannini

Supervisors

Ranieri Baraglia
Dino Pedreschi
Fabrizio Silvestri

Referees

Fabio Crestani
Manuel Prieto Matias

(SSD) INF/01

... Ciò che dobbiamo imparare a fare, lo impariamo facendolo.

(Aristotele)

Abstract

Web search engines are the most popular mean of interaction with the Web. Realizing a search engine which scales even to such issues presents many challenges. Fast crawling technology is needed to gather the Web documents. Indexing has to process hundreds of gigabytes of data efficiently. Queries have to be handled quickly, at a rate of thousands per second. As a solution, within a datacenter, services are built up from clusters of common homogeneous PCs.

However, Information Retrieval (IR) has to face issues raised by the growing amount of Web data, as well as the number of new users. In response to these issues, cost-effective specialized hardware is available nowadays. In our opinion, this hardware is ideal for migrating distributed IR systems to computer clusters comprising heterogeneous processors in order to respond their need of computing power. Toward this end, we introduce K-model, a computational model to properly evaluate algorithms designed for such hardware.

We study the impact of K-model rules on algorithm design. To evaluate the benefits of using K-model in evaluating algorithms, we compare the complexity of a solution built using our properly designed techniques, and the existing ones. Although in theory competitors are more efficient than us, empirically, K-model is able to prove because our solutions have been shown to be faster than the state-of-the-art implementations.

Acknowledgments

I strongly desire to thank a long list of people for helping me in achieving this result. I would like to thank my supervisors Ranieri Baraglia and Fabrizio Silvestri. I am particularly grateful to Ranieri Baraglia whose advises started from my M.Sc. Thesis and to Fabrizio Silvestri whose advises during my Ph.D. helped me in finding interesting problems, always encouraging me to ask the best to myself. I also would like to thank all my co-authors with which I share part of the results of this thesis: Ranieri Baraglia, Claudio Lucchese, Franco Maria Nardini, Raffaele Perego, Fabrizio Silvestri from ISTI-CNR and Laura Ricci from Pisa University. I am also very grateful to the colleagues of the High Performance Computing Laboratory of the ISTI-CNR in Pisa for the precious reminders, discussions, tips, and moments we had since I was a Master student there. I can not forget my colleagues with which I shared the room C64: Franco Maria Nardini, Gabriele Tolomei, Davide Carfì, and Diego Ceccarelli for the wonderful life experience spent in our room and all around the world. I also would like to thank my family and all my friends. The achievement of this important result would not have been possible without the continuous support of them, which always encouraged me in any moment, and helped me whenever I needed it. A warm thank to a special woman, Katalin, whose support during the last years has been very precious.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

This PhD thesis aims to study the potentiality of unconventional computing resources consisting of novel many-core processors[1] to build large-scale distributed Web Information Retrieval (WebIR) systems.

Web Search Engines (WSEs) are the main way to access online content nowadays. Web data is continuously growing, so current systems are likely to become ineffective against such a load, thus suggesting the need of software and hardware infrastructures in which queries use resources efficiently, thereby reducing the cost per query [1]. Given the high demand of computational resources by WebIR systems, many-core processors (also referred as *manycores* in the rest of this document) can be a promising technology. In fact, there have been efforts aimed at developing basic programming models like Map-Reduce[2] on manycores. Recently, Bingsheng *et al.* [2] designed Mars, a Map-Reduce framework on graphics processors, and Kruijf *et al.* [3] presented an implementation of Map-Reduce for the Cell architecture.

The peculiarity of manycores is to merge multiple cores in a single processor package where the supporting infrastructure (interconnect, memory hierarchy, etc.) is designed to provide high levels of scalability, going well beyond that encountered in multi-core processors, compromising single-core performance in favor of "parallel throughput". These cores may be the same (a homogeneous many-core processor) or different (a heterogeneous many-core architecture). Novel manycores born to solve specific problems, such as the graphical ones. Consequently, their exploitation to solve a wider range of problems requires to redesign the related algorithms for such architectures.

The design of efficient algorithms for such architectures is also made more difficult by the lack of computational models that properly abstract

[1] Graphics Processing Units and Cell BE are an example of such architectures.

[2] Map-Reduce is a distributed programming framework originally proposed by Google for developing Web search applications on a large number of commodity CPUs.

their features. Models that do not give a realistic abstraction, such as PRAM [4] for multiprocessor computers, may lead to unrealistic expectations for the designed algorithms, and thus to evaluate theoretically optimal algorithms that are not optimal on any realistic machine [5]. We believe that the challenge of effectively and efficiently programming manycores can be addressed by defining a computational model which captures the essential common features determining the real many-core architecture performance.

1.1 Context & Motivations

People use WSEs daily to access information and services on the Web. As a consequence, Information Retrieval (IR) has to face issues raised by the growing amount of information, as well as the number of new users. Realizing a search engine which scales even to today's Web presents many challenges. Fast crawling technology is needed to gather the Web documents and keep them up to date. Storage space must be used efficiently to store indexes and, optionally, the documents themselves. The indexing system has to process hundreds of gigabytes of data efficiently. Queries have to be handled quickly, at a rate of thousands per second. Moreover, for redundancy and fault tolerance, large search engines operate multiple, geographically distributed datacenters. Large-scale replication is also required to reach the necessary throughput. As a solution, within a datacenter, services are built up from clusters of homogeneous PCs [6]. The type of PC in these clusters depends upon price, CPU speed, memory and disk size, heat output, and physical size. In particular, power consumption and cooling issues can become challenging. Specifically, the typical power density for commercial datacenters is much lower than that required for PC clusters. This leads to the question of whether it is possible to reduce the power usage per server. First, to reduce the power usage is desirable, but it has to come without a corresponding performance penalty for applications. Second, the lower-power server must not be considerably overpriced in order to do not outweigh benefits of the saved power.

In response to these issues, cost-effective specialized hardware (originally designed for different purposes) is available nowadays, namely manycores. Over the past two decades, microprocessor designers have focused on improving the performance of a single thread in a desktop processing environment by increasing frequencies and exploiting instruction level parallelism using techniques such as multiple instruction issue, out-of-order issue, and branch prediction. This has led to an explosion in microprocessor design complexity and made power dissipation a major concern [7, 8]. For these reasons, new generation of microprocessors (not only specialized ones, but also classical CPUs) aims to incorporate an ever-increasing number of "cores" on the same chip, compromising single-core performance in favor of the total amount of work done across multiple threads of execution [9]. In our

opinion, this makes those architectures ideal for migrating distributed IR systems to computer clusters comprising heterogeneous processors in order to respond their need of computing power and to reduce power usage.

For example, a standard CPU requires 65W and is capable of 20 GFLOPS, this means a ratio of 3.25 Watt per GFLOPS. The same calculus for GPUs and Cell BE leads respectively to 70W/250GFLOPS = 0.28W/GFLOPS, and 150W/500GFLOPS = 0.3W/GFLOPS. As a matter of fact, some scientific communities already use graphics processors for general purpose high performance computing, and the proposed solutions outperform those based on classical processors in term of computing time but also energy consumption [10, 11].

Finally, further incentives for using manycores are offered from those companies proposing novel hardware and software approaches. Since some years ago, programming techniques for special purpose accelerators had to rely on APIs to access the hardware, for example OpenGL or DirectX. These APIs are often over-specified forcing programmers to manipulate data that are not directly relevant, and the related drivers take critical policy decisions, such as where data resides in memory and when it is copied, that may be suboptimal. In the last years, due to the trend of media market, the demand for rendering algorithms is rapidly evolving. For the companies producing hardware, that means to redesign every time new hardware able to run novel algorithms as well as provide higher throughput. Such processors require significant design effort and are thus difficult to change as applications and algorithms evolve. The demand for flexibility in media processing motivates the use of programmable processors, and the existing demand for non-graphical APIs sparked the same companies into creating new abstractions that are likely to achieve some type of longevity. So, novel hardware and software solutions are capable of performing stream processing programs as well as facing computational-intensive problems better than in the past.

1.2 Additional Research Areas

During my PhD, I worked in different research areas. The main topic (in which I spent most of my effort) is the exploitation of unconventional computing architectures for solving problems typical in IR. Furthermore, I have also collaborated with other research groups whose research areas are: *job scheduling on distributed computing platforms* and *efficiency in Web search results diversification*. In the rest of this section the results of these studies are briefly described, and a more detailed presentation of them is given in Chapter 5.

Job Scheduling on Distributed Computing Platforms. Such activity was devoted to define a framework for scheduling a continuous stream

of sequential and multi-threaded batch jobs on large-scale distributed platforms, made up of interconnected clusters of heterogeneous machines. This framework, called Convergent Scheduling [12, 13], exploits a set of heuristics that guides the scheduler in making decisions. Each heuristics manages a specific problem constraint, and contributes to compute a "matching degree" between jobs and machines. Scheduling choices are taken to meet the QoS requested by the submitted jobs, and optimizing the usage of hardware and software resources.

Moreover we also propose a two-level scheduler [14] that aims to schedule arriving jobs respecting their computational and deadline requirements. At the top of the hierarchy a lightweight Meta-Scheduler classifies incoming jobs balancing the workload. At cluster level a Flexible Backfilling algorithm carries out the job machine associations by exploiting dynamic information about the environment.

Efficient Diversification of Web Search Results. Concerning the diversification of results returned by WSEs, we particularly deal with efficiency. The conducted studies aim to extend a search architecture based on additive machine learned ranking systems [15] with a new module computing the diversity score of each retrieved document. Our proposed technique consists in an efficient and effective diversification algorithm [16, 17] based on knowledge extracted from the WSE query logs.

1.3 Document Organization

The rest of the document is organized as follows. Chapter 2, describing the state of the art, is divided into three parts: many-core architectures, computational models concerning manycores, and the architecture of today's Web search engines. In Chapter 3 describes K-model, a computational model aiming to properly gather manycores performance constraints. Furthermore, Section 3.2 and Section 3.3 illustrate two case studies concerning efficient sorting and parallel prefix sum computation, respectively. For each case study, the related problem is initially addressed in a theoretical way and the most promising solutions are compared; the section also shows how experimental results confirm the theoretical ones and how the resulting techniques can be effectively applied to different WebIR aspects. Conclusions, possible future works, the list of my publications, and the detailed description of the results obtained in additional research areas are described in the last chapther.

CHAPTER 2

The State of the Art

Manycores are programmable accelerators classified as high-parallel shared-memory architectures capable of performing high performance computations [18]. For this reason, manycores can be a viable solution to effectively and efficiently face the computing needs of a distributed Web IR system. Unfortunately performance expectations cited by vendors and in the press are frequently unrealistic for our purpose due to very high theoretical peak rates, but very low sustainable ones. Given the fact that these kind of processors are usually designed with a special purpose, their general purpose exploitation requires careful design-of-algorithms. In this regard the literature lacks a computational model able to adequately abstract manycores.

The rest of this chapter is organized as follows. Section 2.1 gives an overview on the state of the art in manycores. It describes some nowadays available cost-effective architectures for stream computing and the relative software tools. Section 2.2 describes a number of computational models divided in two classes: sequential and parallel. Due to the relevance of the memory management aspect, the presentation of each class of models is organized grouping them on this aspect. For each model an analysis of its suitability to manycores is provided. Section 2.3 describes some groups of WSE components that are characterized by a high computational load as well as the need of a higher level of performance.

2.1 Target Computing Architectures

All the architectures we are presenting expose similar features. This mainly depends on the specialized purpose they are projected for. Each of them is suitable for computing heavy stream of data faster than CPUs by exploiting a high parallel but less flexible architecture. This is due to the fact that on

manycores the main chip-area is dedicated to computational units, on the contrary, standard CPUs use this space for other mechanisms (e.g. multiple instruction issue, out-of-order issue, and branch prediction). On the one hand, these mechanisms avoid programmers to directly take care of some hard aspects, thus they make easier to get the highest performance reachable from the processor. On the other hand, these architectural choice has the consequence of increasing the GFLOPS reachable by many-core processors but it also makes more difficult to reach the maximum level of performance than with the standard CPUs.

On all the architectures under consideration, the need of computing power has been faced by exploiting some common and comparable characteristics. This is enough to define a unique model able to abstract the underlying real architectures, in order to make possible to design an algorithm for a wider range of hardware. Successively, we plan to enrich this common model with a set functions to measure the time requested by a computation.

In the following, the first two sections describe some relevant characteristics and issues arising from the stream programming model and the single-instruction multiple-data (SIMD) architecture. These are the two main classes in which the processors considered can be included. Then, we will give a briefly introduction to some of these architectures and the available programming tools.

2.1.1 Single-Instruction Multiple-Data Architecture

SIMD machines are classified as processor-array machines: a SIMD machine basically consists of an array of computational units connected together in some sort of simple network topology [19, 20]. This processor array is connected to a control processor, which is responsible for fetching and interpreting instructions. The control processor issues arithmetic and data processing instructions to the processor array, and handles any control flow or serial computation that cannot be parallelized. Processing elements can be individually disabled for conditional execution: this option give more flexibility during the design phase of an algorithm.

Although SIMD machines are very effective for certain classes of problems, they do not perform well universally. This architecture is specifically tailored for data computation-intensive work, and it results to be quite "inflexible" and perform poorly on some classes of problems[1]. In addition, SIMD architectures typically scale better if compared to other types of multiprocessor architecture. That is to say, the desirable price/performance ratio when constructed in massive arrays becomes less attractive when constructed on a smaller scale. Finally, since SIMD machines almost exclusively rely on very simple processing elements, the architecture cannot leverage low-cost advances in microprocessor technology.

[1] For example, these architectures cannot efficiently run control-flow dominated code.

Up until recent years, a combination of these factors have led SIMD architecture to remain in only specialized areas of use.

2.1.2 Stream Processing

Related to SIMD, the computer programming paradigm, called *stream processing* [21], allows some applications to more easily exploit a limited form of parallel processing. Such applications can use multiple computational units, such as the floating point units on a GPU or alike processors, without explicitly managing allocation, synchronization, or communication among those units.

The stream processing paradigm simplifies parallel software and hardware by restricting the parallel computation that can be performed. Given a set of data (a *stream*), a series of operations (*kernel* functions) are applied to each element in the stream. Uniform streaming, where one kernel function is applied to all elements in the stream, is typical. Kernel functions are usually pipelined, and local on-chip memory is reused to minimize external memory usage.

Since the kernel and stream abstractions expose data dependencies, compiler can fully automate and optimize on-chip management tasks. Stream processing hardware can use scoreboarding[2], for example, to launch DMAs at runtime, when dependencies become known. The elimination of manual DMA management reduces software complexity, and the elimination of hardware caches reduces the amount of die area not dedicated to computational units such as ALUs.

Stream processing is essentially a data-centric model that works very well for traditional GPU-type applications (such as image, video and digital signal processing) but less so for general purpose processing with more randomized data access (such as databases). By sacrificing some flexibility in the model, the implications allow easier, faster and more efficient execution. Depending on the context, processor design may be tuned for maximum efficiency or a trade-off for flexibility.

Stream processing is especially suitable for applications that exhibit three application characteristics [22]:

- *Compute Intensity*, the number of arithmetic operations per I/O or global memory reference. In many signal processing applications today it is well over 50:1 and increasing with algorithmic complexity.

- *Data Parallelism* exists in a kernel if the same function is applied to all records of an input stream and a number of records can be processed simultaneously without waiting for results from previous records.

[2] Method for dynamically scheduling a pipeline so that the instructions can execute out of order when there are no conflicts with previously-issued incomplete instructions and the hardware is available.

- *Data Locality* is a specific type of temporal locality common in signal and media processing applications where data is produced once, read once or twice later in the application, and never read again. Intermediate streams passed between kernels as well as intermediate data within kernel functions can capture this locality directly using the stream processing programming model.

2.1.3 Target Architectures

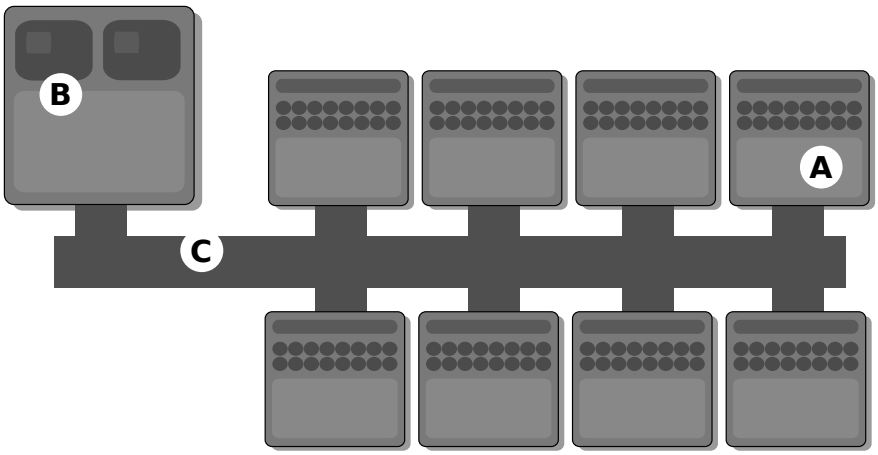
The first two architectures introduced in the following are commodities large-scale diffused, namely Cell BE and GPUs. The last one is the processor proposed by Intel and named Larrabee. Larrabee is not yet available and the related documentation is often confounding. However we are interested to introduce Larrabee as a possible processor to take into consideration for possible future works. On the base of our knowledge, it is possible to rank the following architectures in term nominal computing power. The most powerful processor seems to be represented by GPU, followed by Cell BE; on the contrary Cell BE seems to be more flexible. About Larrabee, we have no data about its capabilities.

Cell Broadband Engine

The Cell Broadband Engine (Cell/B.E.) [23] microprocessor is the first implementation of a novel multiprocessor family that conforms to the Cell/B.E. Architecture (CBEA). The CBEA and the Cell/B.E. processor are the result of a collaboration between Sony, Toshiba, and IBM, which formally began in early 2001. Although the Cell/B.E. processor is initially intended for applications in media-rich consumer-electronics devices, such as game consoles and high-definition televisions, the architecture has been designed to enable fundamental advances in processor performance. These advances are expected to support a broad range of applications in both commercial and scientific fields.

The CBEA has been designed to support a broad range of applications. The first implementation is a single-chip multiprocessor with nine processor elements that operate on a shared memory model, as shown in Figure 2.1. In this respect, the Cell/B.E. processor extends current trends in PC and server processors. The most distinguishing feature of the Cell/B.E. processor is that, although all processor elements can share or access all available memory, their function is specialized into two types: (*i*) Power Processor Element (PPE), (*ii*) Synergistic Processor Element (SPE).

The Cell/B.E. processor has one PPE and eight SPEs. The PPE contains a 64-bit PowerPC ArchitectureTM core [24]. It complies with the 64-bit PowerPC Architecture and can run 32-bit and 64-bit operating systems and applications. The SPE is optimized for running compute-intensive single-instruction, multiple-data (SIMD) applications. It is not optimized



A indicates a Synergistic Processor Element (SPE), **B** indicates the Power Processor Element (PPE), and **C** indicates the memory Element Interconnect Bus (EIB) connects all processor elements.

Figure 2.1: Schema of Cell/B.E Architecture.

for running an operating system. The SPEs are independent processor elements, each running their own individual application programs or threads. Each SPE has full access to shared memory, including the memory-mapped I/O space implemented by multiple-data (SIMD) applications. It is not optimized for running an operating system.

The SPEs are independent processor elements, each running their own individual application programs or threads. Each SPE has full access to shared memory, including the memory-mapped I/O space implemented by direct memory access (DMA) units. There is a mutual dependence between the PPE and the SPEs. The SPEs depend on the PPE to run the operating system, and, in many cases, the top-level thread control for an application. The PPE depends on the SPEs to provide the bulk of the application performance.

The SPEs support a rich instruction set that includes extensive SIMD functionality. However, like conventional processors with SIMD extensions, use of SIMD data types is preferred, not mandatory. For programming convenience, the PPE also supports the standard PowerPC Architecture instructions and the vector/SIMD multimedia extensions.

The PPE is more adept than the SPEs at control-intensive tasks and quicker at task switching. The SPEs are more adept at compute-intensive tasks and slower than the PPE at task switching. However, either processor element is capable of both types of functions. This specialization is a significant factor accounting for the order-of-magnitude improvement in peak computational performance and chip-area and power efficiency that the Cell/B.E. processor achieves over conventional PC processors.

The more significant difference between the SPE and PPE lies in how they access memory. The PPE accesses main storage (the effective address space) with load and store instructions that move data between main storage and a private register file, the contents of which may be cached. PPE memory access is like that of a conventional processor technology, which is found on conventional machines. The SPEs, in contrast, access main storage with DMA commands that move data and instructions between main storage and a private local memory, called local storage (LS). An instruction-fetches and load and store instructions of an SPE access its private LS rather than shared main storage, and the LS has no associated cache. This three-level organization of storage (register file, LS, and main storage) is a radical break from conventional architecture and programming models. The organization explicitly parallelizes computation with the transfers of data and instructions that feed computation and store the results of computation in main storage.

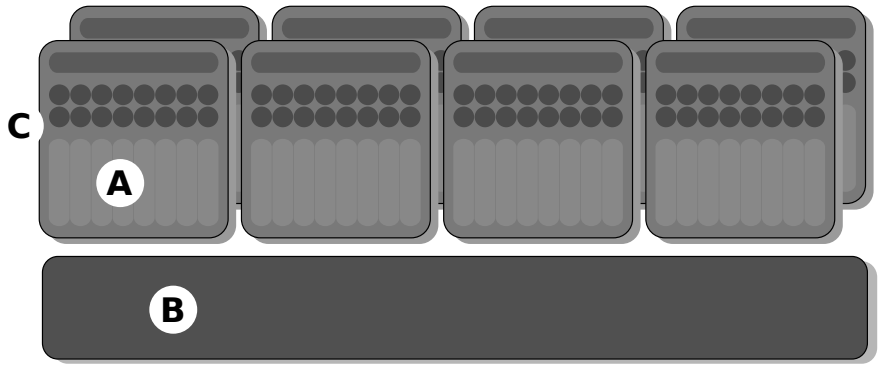
Graphic Processing Units

The difference with the Cell is that Cell is nominally cheaper, considerably easier to program and it is usable for a wider class of problems. This is because the Cell BE architecture is closer to the one of standard CPUs. Existing GPUs can already provide massive processing power when programmed properly but this is not exactly an easy task.

The GPUs we consider in this work are related to a novel generation of programmable graphics processors[25] that is built around a scalable array of streaming multiprocessors (SMs). Each multiprocessor contains a set of scalar processors (referred as "cores" or streaming processors). Furthermore, each processing core in an SM can share data with other processing cores in the SM via the shared memory, without having to read or write to or from an external memory subsystem. This contributes greatly to increased computational speed and efficiency for a variety of algorithms. Based on traditional processing core designs that can perform integer and floating-point math, memory operations, and logic operations, each processing core is multiple pipeline stages in-order[3] processor.

GPUs include a substantial portion of die area dedicated to processing, unlike CPUs where a majority of die area is dedicated to onboard cache memory. Rough estimates show 20% of the transistors of a CPU are dedicated to computation, compared to 80% of GPU transistors. GPU processing is centered on computation and throughput, where CPUs focus heavily

[3] For contraposition in the field of computer engineering, out-of-order execution is a paradigm used in most high-performance processors to make use of instruction cycles that would otherwise be wasted by a certain type of costly delay. In this paradigm, a processor executes instructions in an order governed by the availability of input data, rather than by their original order in a program. In doing so, the processor can avoid being idle while data is retrieved for the next instruction in a program, processing instead the next instructions which is able to run immediately.



A indicates the shared memory, **B** indicates the external memory, and **C** indicates the array of streaming multiprocessors.

Figure 2.2: Schema of GPU architecture.

on reducing latency and keeping their pipelines busy (high cache hit rates and efficient branch prediction[4]).

In the following, the main constraints for obtaining the best performance from the architecture are introduced. Perhaps the most important performance consideration in programming for is "coalescing" external memory accesses. external memory loads and stores issued in a computational step by streaming processors are coalesced by the device in as few as one transaction when certain access requirements are met. To understand these access requirements, external memory should be viewed in terms of aligned segments of consecutive words. Because of this possible performance degradation, memory coalescing is the most critical aspect of performance optimization of device memory.

Because it is on-chip, shared memory is much faster than local and external memory. In fact, shared memory latency is roughly $100\times$ lower than external memory latency when there are no bank conflicts that waste its performance. To achieve high memory bandwidth for concurrent accesses, shared memory is divided into equally sized memory modules, called banks, that can be accessed simultaneously. Therefore, any memory load or store of n addresses that spans n distinct memory banks can be serviced simultaneously, yielding an effective bandwidth that is n times as high as the bandwidth of a single bank.

However, if multiple addresses of a memory request map to the same memory bank, the accesses are serialized. The hardware splits a memory

[4] In computer architecture, a branch predictor is a technique that tries to guess which way a branch (e.g. an `if-then-else` structure) will go before this is known for sure. The purpose of the branch predictor is to improve the flow in the instruction pipeline. Branch predictors are crucial in today's pipelined microprocessors for achieving high performance.

request that has bank conflicts into as many separate conflict-free requests as necessary, decreasing the effective bandwidth by a factor equal to the number of separate memory requests. The one exception here is when all threads in a half warp address the same shared memory location, resulting in a broadcast. To minimize bank conflicts, it is important to understand how memory addresses map to memory banks and how to optimally schedule memory requests.

Finally, any flow control instruction (`if`, `do`, `for`, ...) can significantly impact the effective instruction throughput by causing streaming processors driven by the same instruction unit to diverge, that is, to follow different execution paths. If this happens, the different executions paths have to be serialized, increasing the total number of instructions executed. When all the different execution paths have completed, the differing execution paths converge back to the same one.

Many Integrated Core

Many Integrated Core (MIC) technology represents Intel's entry into the HPC processor accelerator sector[5], as the company attempts to perform an end-run around GPU computing.

Furthermore Intel [26] offers the following description of the MIC architecture: "The architecture utilizes a high degree of parallelism in smaller, lower power, and single threaded performance Intel processor cores, to deliver higher performance on highly parallel applications. While relatively few specialized applications today are highly parallel, these applications address a wide range of important problems ranging from climate change simulations, to genetic analysis, investment portfolio risk management, or the search for new sources of energy."

As suggested by its name, MIC is essentially an x86 processor which puts a number of cores on a single chip. The MIC architecture has more cores (but simpler ones) than a standard x86 CPU and an extra-wide SIMD unit for heavy duty vector math. As such, it's meant to speed up codes that can exploit much higher levels of parallelization than can be had on standard x86 parts.

2.1.4 Software Tools

The use of computer graphics hardware for general-purpose computation has been an area of active research for many years, the first example was Ikonas [27]. Several non-graphics applications, including physics, numerical analysis, and simulation, have been implemented on graphics processors in order to take advantage of their inexpensive raw compute power, and high-bandwidth memory systems. Unfortunately, such programs must rely

[5] The first MIC co-processor to hit the commercial market, and the one slated for use in Stampede, is codenamed "Knights Corner" and will feature over 50 cores.

on OpenGL [28] or DirectX [29] to access the hardware. These APIs are simultaneously "over-specified", in the sense that one has to set and to manipulate data that is not relevant for his purpose, and the drivers that implement these APIs make critical policy decisions, such as where data resides in memory and when it is copied, that may be suboptimal.

An increasing demand for non-graphical APIs sparked a lot of research into creating other abstractions, and there are currently three emerging technologies that are likely to achieve some type of longevity. These are ATI Stream [30], [31] and Nvidia CUDA [32].

ATI Stream SDK

The ATI Stream SDK (namely, Software Development Kit) consists of a large, collection of code examples, compilers and run-time libraries. Central to the collection is the high-level language Brook+ [30] which is based on C/C++. Brook+ enables the writing of CPU code and syntactically simple GPU kernel functions. Once this is done, the Brook+ compiler divides the code into CPU and GPU components for the standard C++ compiler and the kernel compiler respectively. The kernel compiler produces AMD Intermediate Language code that is then interpreted by the stream runtime to Compute Abstraction Layer (CAL) code. The CAL code is then translated into optimised device specific code for the various stream processors available.

Code written directly in CAL can be optimised much more than Brook+ code and so it has the potential for greater performance increases. However, CAL code has a complicated syntax and structure, making it harder to learn and use. However, with support for OpenCL, ATI's GPU platform could become an even more significant part of GPGPU computing.

Nvidia's CUDA SDK

Like ATI Stream SDK, also the SDK provided by Nvidia for its GPUs consist of a large, collection of code examples, compilers and run-time libraries. Thanks to our past experiences with CUDA, let us to put in evidence some constraints imposed by the model.

Clearly the CUDA model is "restricted", mainly for reasons of efficiency. Threads and thread blocks can be created only by invoking a parallel kernel, not from within a parallel kernel. Task parallelism can be expressed at the thread-block level, but block-wide barriers are not well suited for supporting task parallelism among threads in a block. To enable CUDA programs to run on any number of processors, communication between different blocks of threads, is not allowed, so they must execute independently. Since CUDA requires that thread blocks are independent and allows blocks to be executed in any order. Combining results generated by multiple blocks must in general be done by launching a second kernel. However, multiple thread

blocks can coordinate their work using atomic operations on the out-chip memory.

Recursive function calls are not allowed in CUDA kernels. In fact, recursion is unattractive in a massively parallel kernel because providing stack space for all the active threads, would require substantial amounts of memory. To support a heterogeneous system architecture combining a CPU and a GPU, each with its own memory system, CUDA programs must copy data and results between host memory and device memory. The overhead of CPU/GPU interaction and data transfers is minimized by using DMA block-transfer engines and fast interconnects.

OpenCL

OpenCL (Open Computing Language) is a framework for developing and executing parallel computation across heterogeneous processors (CPUs, GPUs, Cell-type architectures and other hardware). The OpenCL framework is made up of an API for coordinating parallel computation across heterogeneous processors, and a cross platform, C-based programming language that contains extensions for parallelization. OpenCL supports both data-parallel and task-parallel programming models and is designed to operate efficiently with graphics APIs such as OpenGL or DirectX. The key feature of OpenCL is that it is designed not as a GPU programming platform, but as a parallel platform for programming across a range of computational devices. This makes it different to either Nvidia's CUDA or ATI's Stream which are proprietary and designed to only work on their respective hardware platforms.

Also if OpenCL has the ability to obtain a unique code for more than one architecture, it is difficult that one algorithmic solution can efficiently spread on a large range of different architectures. Also if OpenCL compiler is able to optimize the code for different types of processor, it does not mean that the same algorithmic approach is the optimal one for all architectures (i.e. general purpose cache-based processor, GPUs SIMD based architecture and Cell BE). Likely we can achieve a common optimal approach in some cases, but we believe that it is not true in general. For example, on GPUs, some relevant aspects of design are left to the experience of the developer, rather than to be hidden by hardware mechanisms as on CPUs.

2.2 Computational Models

The simplified and abstract description of a computer is called a *computational model*. Such a model can be used as a base to estimate the suitability of one computer architecture to various applications, the computation complexity of an algorithm and the potential performance of one program on various computers. A good computational model can simplify the design of an algorithm onto real computers. Thus, such computational model is sometimes also called "bridging model" [33].

This section presents some of the main computational models, so that we can outline lacks and common aspects among them and the architectures presented in Section 2.1. Due to the number of models, they are divided into sequential, and parallel subset. Successively each family is presented according to the memory model of their targeting parallel computers.

2.2.1 Sequential Models

The first bridging model between the sequential computer and algorithms was RAM. The concept of a *Random Access Machine* (RAM) starts with the simplest model of all, the so-called counter machine model. Cook *et al.* [34] introduced a formal model for random access computers and argue that the model is a good one to use in the theory of computational complexity. RAM consists of a finite program operating on an infinite sequence of registers. Associated with the machine is the function $\ell(n)$ which denotes the time required to store the number n. The most natural values for $\ell(n)$ are identically 1, or approximately $\log |n|$.

Memory Hierarchy in Sequential Model

Large computers usually have a complex memory hierarchy consisting of a small amount of fast memory (registers) followed by increasingly larger amounts of slower memory, which may include one or two levels of cache, main memory, extended store, disks, and mass storage. Efficient execution of algorithms in such an environment requires some care in making sure the data are available in fast memory most of the time when they are needed.

HMM-BT. The *Hierarchical Memory Model* (HMM) of computation [35] is intended to model computers with multiple levels in the memory hierarchy. Access to memory location x is assumed to take time $\lceil \log x \rceil$. In order to solve problems efficiently in such an environment, it is important to stage the flow of data through the memory hierarchy so that once the data are brought into faster memory they are used as much as possible before being returned to the slower memory.

An HMM is a RAM where access to memory location requires an access time that is measured by an arbitrary, monotone, non decreasing function. An HMM is defined with unlimited number of registers, and each operation on them has unitary cost.

Successively, Aggarwal *et al.* [36] propose the hierarchical memory extended with block transfer (BT). In addition a contiguous block can be copied in unit time per word after the startup time. Specifically, a block-copy operation copies t memory locations between two disjoint memory intervals in $f(x) + t$ time. In this way, for example, the transfers from disks are charged with the time spent for moving the heads on the platters surface, plus a cost proportional to the length of the block to copy.

Also for GPUs and Cell, to move blocks of contiguous data leads benefits, but in this case no mechanic part is involved in this operation. Specifically, for GPUs out-chip memory bandwidth is used most efficiently when the simultaneous memory accesses can be gathered into a single memory transaction. To obtain the maximum bandwidth, these blocks have to reach a specific minimum size by coalescing the accesses. Differently, BT "suggests" to move as longer as possible blocks.

This behavior is proved to leads no kind of advantage in the practice. Figure 2.3 shows the time needed for transferring 64M integers by using block of varying size that is denotes the variable t occurring in BT cost function. The results of the experiment show that above a specific threshold, transfers on GPUs do not take advantage by increasing the block size used for the transfer.

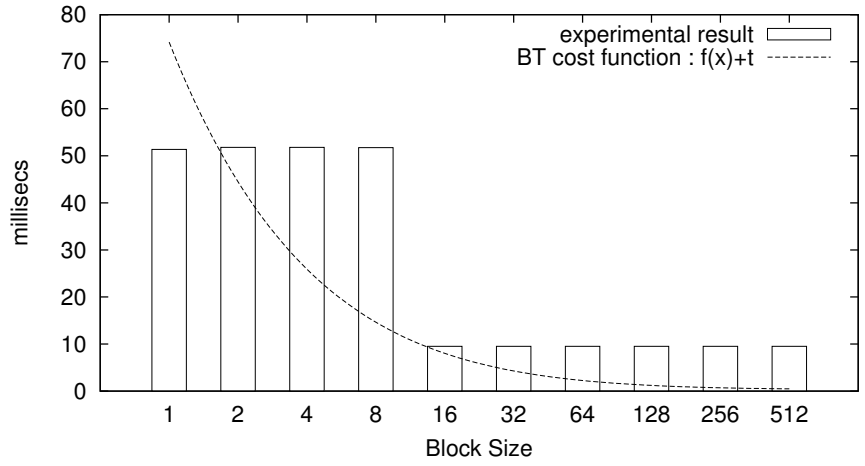


Figure 2.3: Time for transferring an array using varying size blocks on GPU.

Concerning CBEA, for load/store accesses (from SPE to local store to main memory) we report in Figure 2.4 the result of the test Jimenez*et al.* [37]. For DMA transfers the authors have evaluated both DMA requests of a single data chunk (labeled DMA-elem in the experiments), and DMA requests for a list of data chunks (labeled DMA-list). In the first case, the SPE is responsible for programming the Memory Controller (MC) for each individual DMA transfer. In the second case, the SPE provides a list of DMA commands to the MC, and the MC takes care of all requests without further SPE intervention. We have varied the size of DMA chunks from 128 Bytes to 16 KB (the maximum allowed by the architecture). While it is possible to program DMA transfers of less than 128 Bytes, the experiments show a very high performance degradation.

Cache-aware algorithm. In their paper, Aggarwal *et al.* [38] analyzed the sorting problem. Their work focused on the amount of resources are

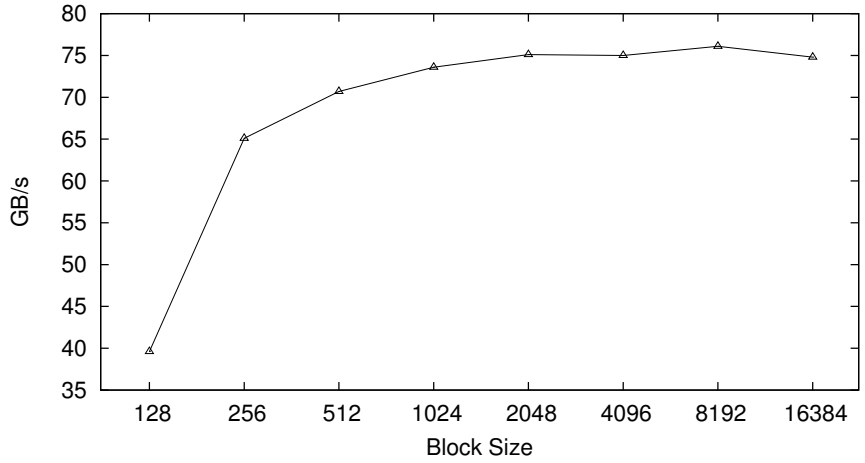


Figure 2.4: Bandwidth for transferring an array using varying size blocks on CBEA.

consumed by external sorts, in which the file is too large to fit in internal memory and must reside in secondary storage.

They examined the fundamental limits in terms of the number of I/Os for external sorting and related problems in current computing environments. They assumed a model with a single central processing unit, and the secondary storage as a generalized random-access magnetic disk.

Each block transfer is allowed to access any contiguous group of B records on the disk. Parallelism is inherent in the problem in two ways: each block can transfer B records at once, which models the well-known fact that a conventional disk can transfer a block of data via an I/O roughly as fast as it can transfer a single bit.

Cache-oblivious algorithm. Frigo *et al.* [39] presents cache-oblivious algorithms that use both asymptotically optimal amounts of work, and asymptotically optimal number of transfers among multiple levels of cache. An algorithm is cache oblivious if no program variables dependent on hardware configuration parameters, such as cache size and cache-line length need to be tuned to minimize the number of cache misses.

The authors introduce the Z, L ideal-cache model to study the cache complexity of algorithms. This model describes a computer with a two-level memory hierarchy consisting of an ideal (data) cache of Z words of constant size, and an arbitrarily large main memory. The cache is partitioned into cache lines, each consisting of L consecutive words that are always moved together between cache and main memory.

The processor can only reference words that reside in the cache. If the referenced word belongs to a line already in cache, a cache hit occurs, and the word is delivered to the processor. Otherwise, a cache miss occurs, and the line is fetched into the cache. If the cache is full, a cache line must

be evicted. An algorithm with an input of size n is measured in the ideal-cache model in terms of its work complexity $W(n)$ and its cache complexity $Q(n, Z, L)$, that is the number of cache misses it incurs as a function of the size Z and line length L of the ideal cache.

The ideal-cache model does not conform to the architectures mainly studied, which exhibits a different memory hierarchy. In fact, target processor architecture is internally equipped with local memory instead of cache. Adopting local memory approach, the movements are managed by software, on the contrary in cache-based architecture this aspect is automatically managed by the underlying support. So if the support forces control, software cannot be oblivious.

Local memory approach forces the programmer to bear the effort of synchronizing, sizing, and scheduling the computation of data and its movement from out-chip memory to the on-chip one. This can be done by moving data located in different addresses composing a specific access pattern. This capability is impossible to realize with caches, where the hardware hides this operation by automatically replacing cache lines missed.

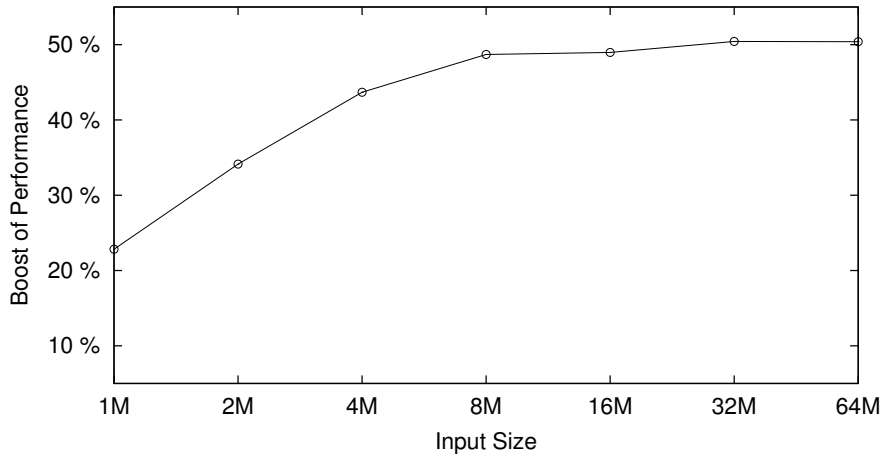


Figure 2.5: Boost of performance obtained applying a properly defined access pattern to sorting GPU-based algorithm.

Eventually, considering the off-chip memory, from our experience on designing sorting algorithm for GPUs [40] and from the results obtained by other researchers [41], we argued that minimizing the amount of data transfered from/to off-chip memory (i.e. the number of cache-misses for any given cache line length L) does not guarantee the best performance, see Figure 2.5. The best results has been reached by augmenting the overall amount of datatransfers from/to off-chip memory, and optimizing the access pattern used. On the contrary, the previous version of our algorithm privileges the minimization of such quantity obtaining lower performance.

2.2.2 Parallel Models

Parallel computational models can be classified into three generations according to the memory model of their targeting parallel computers. The first class is the shared memory parallel computational model, the second generation is the distributed memory parallel computational models, also including some distributed shared memory models, the third generation is the hierarchical memory parallel computational model.

Shared Memory in Parallel Model

With the success of the RAM model for sequential computation, a natural way to model a parallel computation is to extend this model with parallel processing capability. A set of independent processors that share one common global memory pool instead of one processor. From this phase of our investigation we expect to find some models or hints in order to well-describe the main constraints and features for the parallelism exposed by the architectures studied for this PhD thesis.

PRAM & Asynchronous PRAM. A Parallel Random Access Machine (PRAM) [4] consists of an unbounded set of processors $P_0, P_1, \ldots$, an unbounded global memory, a set of input registers, and a finite program. Each processor has an accumulator, an unbounded local memory, a program counter, and a flag indicating whether the processor is running or not.

With respect to the instruction set available on RAM, the fork instruction executed on P_i, provides to replicate the state of P_i into the processor P_j. An halt instruction causes a processor to stop running.

Simultaneous reads of a location in global memory are allowed, but if two processors try to write into the same memory location simultaneously, the PRAM immediately halts. All three steps are assumed to take unit time in the model. The PRAM model assumes that all processors work synchronously and that inter-processor communications are essentially free.

Unlike the PRAM, the processors of an Asynchronous PRAM [42] run asynchronously, i.e. each processor executes its instructions independently of the timing of the other processors. A synchronization step among a set S of processors is a logical point in a computation where each processor in S waits for all the processors in S to arrive before continuing in its local program.

Historically, PRAM models are the most widely used parallel models. For the architectures considered in this PhD thesis, PRAM is inaccurate because it hides details which impact on the real performance of an algorithm, for example it does not consider the time required for communication as well as synchronization aspect.

Module Parallel Computer & QRQW PRAM. Mehlhorn *et al.* [43] consider algorithms which are designed for parallel computations in which processors are allowed to have fairly unrestricted access patterns to the shared memory. The shared memory is organized in modules where only one cell of each module can be accessed at a time. Their research was motivated by the Ultracomputer project: a machine capable of 4096 processors as many memory modules.

The model employs p processors which operate synchronously and N common memory cells. The common memory is partitioned into m memory modules. Say that at the beginning of a cycle of this model the processors issue R_j requests for addresses located in the cells of module j, $0 \leq j < m$. Let $R_{max} = max\{R_j | 0 \leq j < m\}$. Then the requests for each module are queued in some order and satisfied one at a time. So a cycle takes R_{max} time. The authors assume that immediately after the simulation of a cycle is finished, every processor knows it.

Similarly, Gibbons *et al.* [44] presented the queue-read, queue-write (QRQW) PRAM model, which permit concurrent reading and writing, but at a cost proportional to the number of readers/writers to a memory location in a given step.

The QRQW PRAM model consists of a number of processors, each with its own private memory, communicating by reading and writing locations in a shared memory. Processors execute a sequence of synchronous steps, each consisting of three phases for each processor i: read r_i, compute c_i, and write w_i. Concurrent reads and writes to the same locations are permitted in a step. In the case of multiple writers to a location x, an arbitrary write to x succeeds in writing the value present in x at the end of the step. Consider a QRQW PRAM step with maximum contention k, and let $m = max\{r_i, c_i, w_i\}$ for the step. Then the time cost for the step is $max\{m, k\}$, and the time for an algorithm is the sum of its step.

These models seem to properly describe some features of GPUs. Specifically, the models correctly charge the cost of the computation with the time spent by a bank of memory in order to resolve conflicts accessing the on-chip shared memory. Clearly this is only one of the key features I want capture for GPUs and Cell BE.

BPRAM. Aggarwal *et al.* [45] define a Block PRAM with two parameters p, ℓ as follows. There are p processors each with a local memory of unbounded size. In addition, there is a global memory of unbounded size. A processor may access a word from its local memory in unit time. It may also read a block of contiguous locations from global memory, and it may write a block into contiguous locations of the global memory. Such an operation takes time $\ell + b$ where b is the length of the block, and ℓ is the startup time or the latency, and it is a parameter of the machine. Any number of processors may access the global memory simultaneously, but in an exclusive read/write fashion. In other words, blocks that are being accessed

simultaneously cannot overlap; concurrent requests for overlapping blocks are serviced in some arbitrary order. The input initially resides in global memory, and the output must also be stored there.

Raising from the merging of PRAM and HMM-BT also this model seem to be not able to define on aspects of our set of architectures. First of all, memory of unbounded size assumption does not conform our set of architectures. Moreover, the approach on the cost for accessing the memory do not reflect the effective time spent in real cases of study as the test result reported in Figure 2.3 shows.

HPRAM. The Hierarchical PRAM model, proposed by Heywood *et al.* [46], consists of a dynamically configurable hierarchy of synchronous PRAMs, or equivalently, a collection of individual synchronous PRAMs that operate asynchronously from each other. A hierarchy relation defines the organization of synchronization between independent PRAMs. The set of allowable instructions are any PRAM-instruction and the partition-instruction. The partition-instruction adds a controlled form of asynchrony to the model. Specifically, a partition-step in an algorithm splits the set of processors into disjoint subsets and assigns a synchronous PRAM algorithm to execute on each of them. Each subset of processors is a synchronous PRAM operating separately from the others.

The latency $\ell(P)$ and the synchronization cost $s(P)$ are functions of the number P of processors being communicated amongst and synchronized. The latency $\ell(P)$ has traditionally been correlated with the diameter d_P of the network interconnecting P processors in the architecture. The synchronization cost is typically $d_P \leq s(P) \leq d_P \cdot \log P$. Let T and C denote the number of computation and communication steps, respectively, the complexity of the sub-PRAM algorithm is $T + C \cdot \ell + (T + C) \cdot s(P)$.

Unlikely HPRAM lacks of any consideration about a cost function for charging the data transfers. However, the partition of processors can be a viable method to abstract the model the pattern used to distribute data among processors.

Distributed Memory in Parallel Model

It also exists several parallel computational models assuming a distributed memory paradigm and the processor communicates through message passing. They are interesting for our purpose, for example, to support a heterogeneous machine made of CPUs and GPUs. In fact, in this kind of machine each type of processor has its own memory, then a program have to copy data and results between CPU's memory and GPU's memory. The following models has been red in order to obtain some hints regarding a model for a heterogeneous architecture.

BSP & D-BSP. The Bulk-Synchronous Parallel model (BSP), proposed by Valiant [47], is defined as the combination of three attributes: a number of components, each performing processing and/or memory functions, a router that delivers point-to-point messages, and facilities for synchronizing all or a subset of the components at regular intervals of L time units where L parameter specifies the periodicity. A major features of the BSP model is that it provides to the programmer the option to avoid the effort of managing memory, assigning computation and, performing low-level synchronization.

A BSP computation consists of a sequence of supersteps. In each superstep, at each component is allocated a task consisting of some combination of local computation steps, message transmissions and (implicitly) message arrivals from other components. After each period of L time units, a global check is made to determine whether the superstep has been completed by all the components. If it happens, the machine proceeds to the next superstep. Otherwise, the next period of L units is allocated to the unfinished superstep. The synchronization mechanism can also be switched off for any subset of the components; sequential processes that are independent of the results of processes at other components should not be slowed down unnecessarily.

The message transmission is accomplished by the router, which can send and receive h messages in each superstep (shortly *h-relation*). The cost of realizing such relation is assumed to be $g \cdot h + s$ time units, where g can be thought as the latency for each communication and s denotes the startup cost. If the length of a superstep is L and $g \cdot h \gg s$, then L local operations and a $\lfloor L/g \rfloor$-relation message pattern can be realized.

De la Torre *et al.* [48] proposed D-BSP that extends a BSP by adding a collection of subset of submachines, each one composed by processors. Furthermore, each submachine has the possibility to synchronize and to exchange messages among its processors. A D-BSP computation is recursively defined as a sequence of terminal/structural metastep. The time charged for a sequence is the sum of the times charged for its metasteps. For each metastep is defined a submachine s, and on its end, all processors in s synchronize.

If the metastep is terminal, each processor in the submachine s computes its local data and exchange the results within s. The time charged is the maximum number of local computations executed by all processors in s plus the time occurred to realize a session of communications in s (that is an h-relation in BSP terminology above).

Otherwise, in a structural metastep for the submachine s, for a given set of metasteps $\{\mu_i\}_{1 \leq i \leq r}$, where each μ_i is a metastep of an independent machine s_i in s, the processors in each s_i behave according to μ_i. In this case, the time charge is the maximum of the time of each metastep plus the startup latency of s.

Both the models are oriented to describe in particular synchronization

aspects. Nevertheless, BSP is not able to capture submachine locality, and charges all processors as the busiest one, so for some algorithms, this exaggerates the total amount of data exchanged and overestimate the runtime. For example, in GPUs case, a stream element that ends its computation is immediately replaced with a new one, without synchronizations with others kernel-processor. Furthermore the communications for the next stream element can happen concurrently with the unfinished computation of others processors.

LogP. Culler *et al.* [49] developed a model of distributed-memory multiprocessor architecture in which processors communicate by point-to-point messages. The model, called LogP, specifies the performance characteristics of the interconnection network, but does not describe the structure of the network.

The main parameters of the model are: L, an upper bound on the latency incurred in communicating a message from its source module to its target module; o, the overhead, defined as the length of time that a processor is engaged in the transmission or reception of each message; during this time, the processor cannot perform other operations; g, the gap, defined as the minimum time elapsed between two consecutive message transmissions/receptions for a processor; P, the number of processor/memory modules.

Furthermore, it is assumed that the network has a finite capacity, such that at most $\lceil L/g \rceil$ messages can be in transit from any processor or to any processor at any time. If a processor attempts to transmit a message that would exceed this limit, it stalls until the message can be sent without exceeding the capacity limit.

The model is asynchronous, so the processors work asynchronously and the latency of any message is unpredictable, but is bounded above by L in the absence of stalls. Because of variations in latency, the messages directed to a given target module may not arrive in the same order as they are sent. In analyzing an algorithm, the key metrics are the maximum time and the maximum space used by any processor. In order to be considered correct, an algorithm must produce correct results without stalls. In estimating the running time of an algorithm, this constraint is assumed satisfied.

This model is built for point-to-point type of messages. This type of communication is not exploitable because it is contradictory with the characteristics of the streaming programming model. Specifically, each element is ran only once locally to the kernel-processor. Moreover, inherently to GPUs, each the order followed to execute all elements in the stream is not known a priori, so it is not possible to know if a kernel is already elapsed or not.

Memory Hierarchy in Parallel Model

There are several distributed memory models that incorporate the memory hierarchy into the analysis of parallel computational models. This is due to the different gap of speed that exist considering processor and memory.

P-HMM & P-BT. The P-HMM and P-BT are two hierarchical memory models augmented to allow parallel data transfer. Specifically, each memory component is connected to P larger and slower memory components at the next level. The extension adopted in their paper is to have P separate memories connected together at the base level of each hierarchy. They assume that the P base memory level locations are interconnected via a network.

Vitter *et al.* [50, 51] provide optimal algorithms for different problems in terms of the number of input/outputs (I/Os) required between internal memory and secondary storage. Their two-level memory model gave a realistic treatment of parallel block transfer, in which during a single I/O each of the P secondary storage devices (disks) can simultaneously transfer a contiguous block of B records. To be realistic, the model requires that each block transfer must access a separate secondary storage device.

Parallelism appears in this model in two basic ways: records are transferred concurrently in blocks of contiguous records, in this case the seek time is a dominant factor in I/O; the second type of parallelism arises because P blocks can be transferred in a single I/O.

LogP-HMM. From an examination of existing models, Li *et al.* [52] observed that a void exists in parallel models that accurately treat both network communication and multilevel memory. The authors proposed LogP-HMM model to fill this gap by combining two kinds of models together, and by adding more resource metrics into either model.

The resulting model consists of two parts: the network part and the memory part. The network part can be any of the parallel models such as BSP and LogP, while the memory part can be any of the sequential hierarchical memory models such as HMM and UMH.

A LogP-HMM machine consists of a set of asynchronously executing processors, each with an unlimited local memory. The local memory is organized as a sequence of layers with increasing size, where the size of layer i is 2^i. Each memory location can be accessed randomly; the cost of accessing a memory location at address x is an access cost function $f(x)$, usually $\log x$. The processors are connected by a LogP network at level 0. In other words, the four LogP parameters, L, o, g and P, are used to describe the interconnection network. A further assumption is that the network has a finite capacity such that at any time at most, $\lfloor L/g \rfloor$ messages can be in transit from or to any processor.

From the authors we take the hint to approach our modelling problem by extending an existing parallel model with others. In such way we can cope the lacks of the different existing models by exploiting the results achieved by other authors. It remain the effort to formalize additional resource metrics into either model involved.

UPMH. Alpern *et al.* [53] introduced the Uniform Memory Hierarchy (UMH) model to capture performance relevant aspects of the hierarchical nature of computer memory. A sequential computers memory is modelled by the authors as a sequence $\langle M_0, M_1, \ldots \rangle$ of increasingly large memory modules that defines a *memory hierarchy* MH_σ. Computation takes place in M_0, so as to M_0 can model a computers central processor, while M_1 might be cache memory, M_2 main memory, and so on.

A memory module M_u is a triple $\langle s_u, n_u, l_u \rangle$. Intuitively, M_u holds n_u blocks, each consisting of s_u data items. The bus B_u copies atomically a block to or from level-$u+1$ in l_u cycles. All buses may be active simultaneously. The transfer costs of buses in a UMH are given by a function $f(u)$ giving the transfer cost in cycles per item of B_u.

The authors also give the notion of *communication efficiency* of a program. It is the ratio of its RAM complexity to its UMH complexity in term of the problem size. A program is said to be communication-efficient if it is bounded below by a positive constant. Otherwise, the program is communication-bound.

In the same paper, the authors parallelized the UMH model to handle parallelism. A module of the Parallel Memory Hierarchy (PMH) model can be connected to more than one module at the level below in the hierarchy, giving rise to a tree of modules with processors at the leaves. Finally, between two levels of the tree, the model allows point-to-point and broadcast communications.

Formally, a Uniform Parallel Memory Hierarchy, $UPMH_{\alpha,\rho,f(u),\tau}$ is a PMH forming a uniform τ-tree of $\langle \rho^u, \alpha\rho^u, \rho^u f(u) \rangle$ memory modules defined for UMH model.

This model introduced by the authors is suitable to formalize the systems we are interested. Tree representation of the system is sufficiently flexible to well-describe more than one kind of architecture, and the different levels of memory. Unlikely the model needs to be specialized, hopefully merging on the leaves not simple processor but other models better representing the type of processor the architecture is provided.

DRAM. The RAM(h, k) [54] model is a computational model with take into particular consideration the instruction level parallelism (k ways) and memory hierarchy (h levels). The authors argued that the complexity analysis of algorithms should include traditional computation complexity (time and space complexity) and new memory access complexity to reflect the

different memory access behavior on memory hierarchy of different implementations.

Unlike the UMH model that gives a detailed architectural model and scheduling of data transfer, RAM(h, k) accounts for the different memory hierarchy level by their different real measurable memory access cost under different memory access patterns of an algorithm including temporal and spatial locality.

The computer is modeled as one RAM with different memory access cost on each level under different memory access patterns. In RAM(h, k), the traditional RAM model with one level unit access cost memory and single operation per cycle processing unit was extended to the RAM(h, k) model which has non-uniform access cost h-level memory hierarchies and k concurrent operations per cycle processing unit.

DRAM(h, k) consists of p independent RAM(h, k) processors each with its own local memory, and it can be viewed as the combination of the RAM(h, k) and LogP models.

Network-Oblivious. Biliardi *et al.* [55] introduced the network-oblivious algorithm $\mathcal{A}$ for a given computational problem Π. Let n be a suitable function of the input size for Π, a network-oblivious algorithm $\mathcal{A}$ for Π is designed for a complete network $M(n)$ of n Processing Elements (PEs), $PE_0, \ldots, PE_{n-1}$, each consisting of a CPU and an unbounded local memory. $\mathcal{A}$ consists of a sequence of steps labeled in the $[0, \log n)$ range.

For $0 \leq i < \log n$ and $0 \leq j < n$, in an i-step, PE_j can perform operations on locally held data, and send data only to any PE_k whose index k has the same i most significant bits of the index j.

In order to analyze $\mathcal{A}$s communication complexity on different machines, the authors introduce the machine model $M(p, B)$, where the parameters p and B are positive integers. $M(p, B)$ is essentially an $M(p)$ with a communication cost function parametrized by the block size B.

Furthermore, the authors introduce the definition of *optimality*. Let *communication complexity* of an algorithm be the sum for all steps of the maximum number of blocks sent/received by a single PE in one step, a network-oblivious algorithm $\mathcal{A}$ for a problem Π is optimal if, for any instance of size n and for every $p \leq n$ and $B \geq 1$, the execution of $\mathcal{A}$ on an $M(p, B)$ machine yields an algorithm with asymptotically minimum communication complexity among all algorithms for Π on $M(p, B)$.

2.3 Web Search Engines

A Web Search Engine (WSE) is designed to give the access to information on the Web. WSEs work by storing information about many web pages. These pages are retrieved by a Web *crawler*, which is an automated browser used as a mean for providing up-to-date data. Web crawlers are mainly used to

create a copy of all the visited pages for later processing by a search engine that will *index* the downloaded pages. Data about web pages are stored in an index database for use in later *queries*. A query can be a single word. The purpose of an index is to allow information to be found as quickly as possible.

This chapter focuses a possible group of services that require an higher computing power, thus that can be usefully executed on novel architectures. For each service the relative subsection gives a briefly introduction to the problem, and points out how the new architectures could impact. In this phase we investigate how to parallelize common solutions at-the-state-of-the-art, and we propose the use of alternative solutions that could give better effective results in exchange of an higher computational load.

2.3.1 Infrastructure Overview

For redundancy and fault tolerance, large search engines operate multiple, geographically distributed datacenters. Within a datacenter, services are built up from clusters of commodity PCs. The type of PC in these clusters depends upon price, CPU speed, memory and disk size, heat output, reliability, and physical size [6]. The total number of servers for the largest engines is now reported to be in the hundreds of thousands.

Within a datacenter, clusters or individual servers can be dedicated to specialized functions, such as crawling, indexing, query processing, snippet generation, link-graph computations, result caching, and insertion of advertising content.

Large-scale replication is required to handle the necessary throughput. For example, if a particular set of hardware can answer a query every 500 milliseconds, then the search engine company must replicate that hardware a thousandfold to achieve throughput of 2000 queries per second. Distributing the load among replicated clusters requires high-throughput, high-reliability network front ends.

Currently, the amount of Web data that search engines crawl and index is on the order of 400 terabytes, placing heavy loads on server and network infrastructure. Allowing for overheads, a full crawl would saturate a 10-Gbps network link for more than 10 days. Index structures for this volume of data could reach 100 terabytes, leading to major challenges in maintaining index consistency across datacenters. Copying a full set of indexes from one datacenter to another over a second 10-gigabit link takes more than a day.

2.3.2 Crawling

The aim of a Web crawler is exploring the Web. It is a computerized "robot" that connects to responding computer systems, follows links to documents, and compiles an index of those links and the information available via the links. But due to the huge volume of Web pages and limitations of hardware,

a search engine can only fetch a fraction of the Web. Therefore, which pages should be downloaded first is crucial. Usually this problem may take many factors into consideration. No doubt that the quality of the downloaded page set is one of the most important factors. In common sense, people always prefer to download the most important pages first.

For example, when a Web crawler prepares to download a page set, it starts with an initial set of seeds. Then all the URLs that are parsed from the seed pages are added into the URL waiting list for further visit. A main problem is how to choose the pages that should be visited next. This is so-called *crawling ordering strategy*. There are several representative crawling ordering strategies [56] such as Breadth-First (BF), Backlink-Count (BC), Batch-Pagerank (BPR) and Larger-Sites-First (LSF).

BF method selects page according to the order of the URL in the list. This method is very easy to implement and can work effectively in most situations. BC always visits the page with the most backlink count first, that is to say, the number of backlink is the metric of page quality.

BPR uses the Web graph of pages that have been downloaded so far to compute the Pagerank values of pages in the list, and then chooses the page with the highest score for further visit. This is a time consuming process although it sounds good in discovering high quality pages. LSF is mainly based on the assumption that a large-scale site may have a high possibility of high quality.

From the results obtained by Baeza-Yates, the strategies BPR and LSF have better performance than the other strategies. However, it prejudices small sites and the definition of "large" can be difficult [57]. Thus LSF has better scalability making it more suitable for large scale distributed crawlers. In a real setting, this strategy should include mechanisms to avoid spam pages, for example to check for near-duplicate pages to avoid giving high rank to sites that create artificial loops on themselves.

These last considerations can lead to prefer BRP to LSF, if we are able to bound the hard computational load imposed by the method.

2.3.3 Indexing

Each crawled document is converted into a set of word occurrences called hits. For each word the hits record: frequency, position in document, and some other information.

Indexing can also be considered as a "sort" operation on a set of records representing term occurrences [1]. Records represent distinct occurrences of each term in each distinct document. Sorting efficiently these records using a good balance of memory and disk usage, is a very challenging operation. In the last years it has been shown that sort-based approaches [58], or single-pass algorithms [59], are efficient in several scenarios, where indexing of a large amount of data is performed with limited resources.

Sort-based approach first makes a pass through the collection assembling

all term-docID pairs. It then sorts the pairs with the term as the dominant key and docID as the secondary key. Finally, it organizes the docIDs for each term into a postings list (it also computes statistics like term and document frequency). For small collections, all this can be done in memory.

With main memory insufficient, we need to use an external sorting algorithm [60]. For acceptable speed, the main requirement of such algorithm is that it minimizes the number of random disk seeks during sorting. One solution is the Blocked Sort-Based Indexing algorithm (BSBI). BSBI segments the collection into parts of equal size, then it sorts the termID-docID pairs of each part in memory, finally stores intermediate sorted results on disk. When all segments are sorted, it merges all intermediate results into the final index.

A more scalable alternative is Single-Pass In-Memory Indexing (SPIMI). SPIMI uses terms instead of termIDs, writes each block's dictionary to disk, and then starts a new dictionary for the next block. SPIMI can index collections of any size as long as there is enough disk space available. The algorithm parses documents and turns them into a stream of term-docID pairs, called tokens. Tokens are then processed one by one.

For each token, SPIMI adds a posting directly to its postings list. Instead of first collecting all termID-docID pairs and then sorting them (as BSBI does), each postings list is dynamic. This means that its size is adjusted as it grows. This has two advantages: it is faster because there is no sorting required, and it saves memory because it keeps track of the term a postings list belongs to, so the termIDs of postings need not be stored.

When memory has been exhausted, we write the index of the block (which consists of the dictionary and the postings lists) to disk. Before doing this, SPIMI sorts the terms to facilitate the final merging step: if each block's postings lists were written in unsorted order, merging blocks could not be accomplished by a simple linear scan through each block. The last step of SPIMI is then to merge the blocks into the final inverted index.

SPIMI, which time complexity is lower because no sorting of tokens is required, is usually preferred with respect to BSBI that has higher time complexity.

2.3.4 Query Processing

WSEs are facing formidable performance challenges as they need to process thousands of queries per second over billions of documents. To deal with this heavy workload, current engines use massively parallel architectures of thousands of machines that require large hardware investments.

At the state of the art in query processing on GPUs, we are aware only of the writing of Ding *et al.* [61]. They investigated new ways to build such high performance IR systems based on GPUs. Their contribution was to design a basic system architecture for GPU-based high performance IR, and to describe how to perform highly efficient query processing within such

architecture. Preliminary experimental results based on a prototype implementation suggest that significant gains in query processing performance might be obtainable with such an approach.

Modern WSEs use ranking functions based on many features (for example Pagerank) that are combined into an overall scoring function using machine-learning techniques. So query processing is typically divided into two phases: an initial one uses a fairly simple ranking function, such as BM25 together with some global document score such as Pagerank, to select a set of candidate documents; in the second phase, the machine-learned scoring function is applied to only these candidates to select top results.

Specifically, using the Rice-coding the authors apply parallel prefix-sum to bit array in order to retrieve the most significant part of the "gap". A bit-grained array probably implies that each GPUs core has to accesses to the same word during the parallel prefix-sum. Specifically one memory bank having to satisfy the more than one request, has serialized them augmenting the total computational time.

Applying PForDelta, the authors use globally three arrays to store the gaps. Proposed solution recursively uses PForDelta to code lowest bits of each gap, and the offset of each exception. However from the article it is not clear how the eventual exceptions in the most inner PForDelta encoding are managed. The decompression phase, in this case is charged not only for conflicts on the on-chip memory, but also for the double decoding phase and eventually for the exceptions. Also from the results presented, decompression speed on CPUs and GPUs put in evidence similar level of performance (see Table 2.1).

Algorithm	CPU	GPU
Rice	310.63	305.27
PForDelta	1165.13	1237.57

Table 2.1: Decompression speed on CPU and GPU in millions of integer per second.

For the reasons wrote above, we believe it can exist a margin for improvements that can be reached by taking into account that the GPU's "core" has a behavior similar to the QWQR PRAM.

Moreover, in the same paper, the presentation of the approach to the TAAT method lacks of particular. So it is not possible make deep consideration about possible improvements. It could be useful, also for DAAT approach, deeply investigate on method for computing top-k results. About this operation we are aware of previous works: Neumann *et al.* [62] introduced novel optimization methods for top-k aggregation queries in distributed environments. This optimization proposed computes data-adaptive scans for different input sources on a tree structure. Furthermore, TPUT

[63] and KLEE [64] methods have to be evaluated for their implementation on GPUs.

2.3.5 Postings List Compression

Encoding lists of integers effciently is important for many applications in different fields. Inverted indexes of WSEs keep the lists of postings compressed in order to exploit the memory hierarchy. In this subsection we show two of the most interesting solution for this problem: the Vector of Splits Encoding and PForDelta (mentioned in the previous subsection).

In [65] the authors present *Vector of Split Encoding*, hereinafter VS, a novel class of encoders designed to efficiently represent lists of integers. The method works by splitting the list in variable-length blocks and by encoding any integer in each block by using a fixed number of bits (namely, the number of bits required to represent the largest integer of the block). VS optimally partitions the lists in blocks via dynamic programming.

PForDelta [66], hereinafter P4D, encodes lists of k consecutive integers. The method firstly finds the smallest b such that most (e.g. $\sim 90\%$) of the integers in the list are $\leq 2^b$. Then, it encodes them by storing each integer as a b-bit entry. Each entry is then packed within a of $k \cdot b$ bits, with k usually equal to a multiple of the word size so as to obtain word aligned blocks. Those integers that do not fit within b bits are treated as exceptions and stored differently. For example by storing the part of a binary representation that exceeds the b bits into a separate array with a reference to the position of the related block element. Each exceptions is then concatenated to the original codeword during the decoding phase.

CHAPTER 3

K-model

Nowadays computing architectures exploiting "manycores" processors are available. As discussed in Section 2.1, Graphics Processing Units, Sony's Cell/BE processor, and next-generation CPUs are able to compute hundreds of arithmetic and logical operations at the same time. The key feature of all these architectures is the use of a massive collection of SIMD computing cores integrated in a single silicon chip. The computational elements of such cores share a small on-chip data-memory. Furthermore, all the SIMD processors are connected to a global off-chip memory. Such novel manycore processors are programmable accelerators classified as high-parallel shared-memory architectures capable of performing high performance computations. An important aspect is that as processor building technology advances, GPUs and CPUs differently exploit the number of available additional transistors. GPUs are specialized for compute-intensive computations, and therefore designed to devote more transistors to data processing rather than data caching and flow control. As a result, the parallelism adopted by GPUs is devoted to roughly augment the number of arithmetical operations issued at the same time, and the bandwidth of the communication channels. On the other hand, CPUs offer more flexible programming models and techniques making CPU programming a task easier than GPUs programming. Furthermore, related to SIMD processors, there is the stream programming model, see Section 2.1.2. Briefly, according to such programming model, a stream program organizes data as a *stream* and expresses all computations as *kernel*. A stream is a homogeneous sequence of elements composed by a subset of the input dataset which are, in turn, computed independently. A kernel consumes a set of input streams, performs a computation, and produces a set of output streams. Streams passing among multiple computation kernels form a stream program.

As discussed in Section 2.2 many computing models have been proposed

in the past to analyze sequential and parallel algorithms. For different reasons these models are not able to accomplish our purposes. For instance, their target architectures are too different from manycores, or the level at which they abstract the architecture is too high, they tend to do not consider important aspects of computations. As an example, the widely known PRAM model [4] can be considered a unifying parallel paradigm. However, it is far from being accurate: for example it does not consider the "cost" of addressing the different levels of the real memory hierarchy that, instead, has a great impact on performance. Despite the last consideration, PRAM has been applied to some recently proposed GPU algorithms [67]. Indeed, as we describe in this section, experimental results contradict theoretical results that are not done using a computational model specifically designed for GPUs. As a consequence, we aim to define a computational model able to capture the architectural features described in detail in Section 2.1 and summarized in the following:

- *serial control.* Due to the SIMD nature of these architectures, algorithm designers have to face the serial control programming effort. On the one hand this architectural solution permits to multiply the FLOPS of the processors for a small chip area. On the other hand to reach the peak of performance we must synchronize the instruction flows performed by the computational elements related the same instruction unit.

- *internal memory.* The SIMD processors considered in Section 2.1 comprises one instruction unit, a collection of single precision pipelines executing scalar instructions, and a local shared data-memory. This memory is divided into independent modules, which can be accessed simultaneously. If two addresses of a memory request fall in the same module, there is a conflict, thus the accesses have to be serialized, penalizing the latency of the overall units of the multiprocessors (MP).

- *external memory.* Finally, the architecture is equipped with its own external memory that is off-chip, and can be addressed by each MP during a computation. The latency spent to complete each data transfer depends on the number of memory transactions issued to access to the different memory segments addressed.

Eventually, Table 3.1 resumes the main computing models analyzed in the previous section to resumes what models can be exploited to properly summarize the target peculiarities. Since none of these models is able to cover all aspects, we properly define a new one. In [68] we propose *K-model*, a computational model developed to capture all the features of GPUs, and alike architectures. In the K-model, an algorithm is evaluated by measuring the *time complexity* and the *computational complexity*. The former is the *latency* of each instruction an algorithm calls, summed over all the instructions called. The computational complexity, instead, can be thought as the

classical sequential complexity assuming we are simulating the execution of the algorithm on a serial RAM. To the best of our knowledge K-model is the first paper attempting to propose a computational model specifically targeting manycores.

	serial control	*internal memory*	*external memory*
(P)RAM[34, 4]	–	–	–
V-RAM[69]	Yes	–	–
(P)HMM[35]	–	–	–
(P)BT[36]	–	–	Yes
QRQW[44]	–	Yes	–
BSP[47]	Yes	–	–
UPMH[53]	–	–	Yes
Cache Oblivious[39]	–	–	Yes

Table 3.1: Matching between computational models and architectural features. If 'Yes' the model in the corresponding row represents properly the feature in the corresponding column.

3.1 Definition

K-model [68] is designed to have an abstract machine able to capture the key features observed in the novel generation of processing units. This class of processors is specialized for compute-intensive, highly parallel computation, exactly what graphics rendering is about, therefore more transistors are devoted to data processing rather than data caching and flow control.

Due to the homogeneity of available processors and the unpredictability with which a stream element is assigned to a processor, the abstract machine referred in the K-model appears simplified w.r.t. the real architecture. In fact real devices are in general composed of an array of SIMD processors so as to improve the overall scalability of the architecture when their number grows. On the contrary, k-model restricts the processors array to only one. This because the benefit of having more SIMD processors on the real architectures does not concern the design of the algorithm which should aim at the effective computation of the single stream element. In other words, the core of the K-model is the evaluation of the "parallel-work" performed on each stream element by one SIMD processor, disregarding the number of real processors.

Before discussing the notion of complexity, we introduce the K-model to study the complexity of algorithms. This model, which is illustrated in Figure 3.1, consists of a computer with an array of k scalar execution units ($E[1..k]$) linked to a unique instruction unit (IU) which provides to dispatch the instructions to the scalar units. This forces the work of the scalar units to be synchronized, because only one instruction is issued at each step.

The memory hierarchy consists of an *external memory* and a local memory made of a set of *local registers*, and a *internal memory* of σ locations equally divided into k parallel modules (also referred as banks).

The evaluation of an instruction issued by IU in the different execution paths composing the whole instruction flow of the kernel is based on two criteria: *latency* and *length*. Length is simply the number of units involved in the execution of an instruction issued by the IU, then it will be in the range $[1..k]$. Latency, instead, is measured in terms of the type of instruction issued.

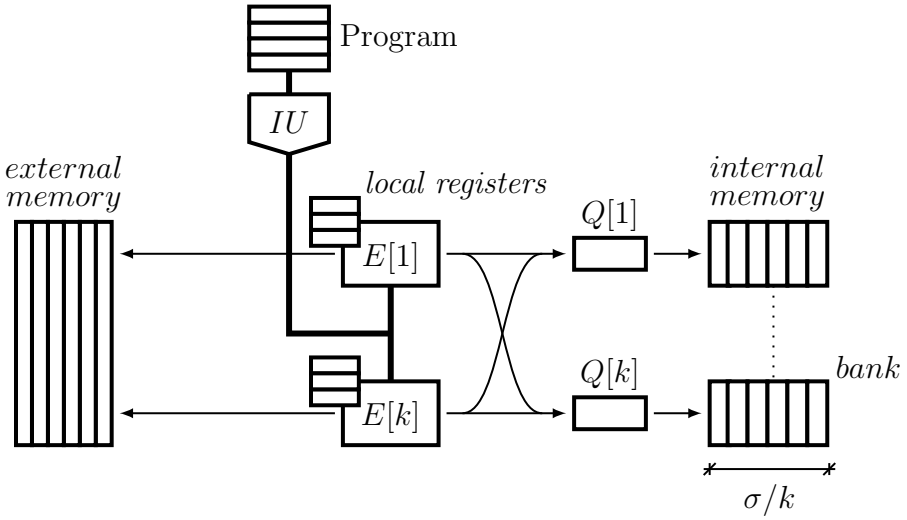


Figure 3.1: K-model architecture schema.

The K-model evaluates the latency of the arithmetical instructions has a cost equal to 1. Instead, the time required to perform a data-access instruction is evaluated proportionally to the level of contention it generates. Whenever an instruction addresses the internal memory with no bank conflict or a register, the latency of the instruction equals 1. Otherwise, the latency of a data-access instruction is equal to the highest number of requests incoming to one of the k memory banks. In fact, whenever the IU issues an instruction accessing the internal memory (at most) k requests performed by the execution units $E[1..k]$ are collected by the queues $Q[1..k]$, each of them managing its connected memory module. In order to provide the contents of memory locations requested, the memory modules work independently by returning the data once at a time. As a consequence, because all the accesses have the same latency and because the next instruction will be issued only when all requests are satisfied, the time required to completely perform an instruction accessing the internal memory is equal to the size of the queue that has collected more requests: $\max\{\, |Q[i]| \,\}_{\forall i \in [1..k]}$.

However, the evaluation of the "internal work" induced by a stream element is not the unique purpose of K-model. A relevant aspect to take into

consideration is the pattern used to access the external memory in order to fetch the input data and then to flush the results. Since the cost of accessing the external memory is one order of magnitude higher than the other type of data-accesses, and due to the parallelism among the instructions performed on the local memory and the access to the external one, we separately measure the number of *memory transactions* issued to access to the external memory. This measure aims at taking into account coalescing-degree of the memory accesses requested by the k scalar processors. In particular, the aspect to take into account in this type of evaluation is not the amount of data transfered, but the number of memory transactions. Independently of the number of external memory locations addressed, the latency of the transfer is defined by the time required to perform of such transaction. Thus, to increase the data transfer efficiency we have to organize the accesses so as to request the locations lying in the same k-size segment in same transaction (or in the least number of them).

3.1.1 Performance Indicators

Concerning the architecture previously described, we denote the time required to perform any given algorithm by defining two performance indicators, namely $T()$ and $G()$. The first one, $T()$, evaluates the work performed inside the processor and is function of latency of the instructions previously defined. Instead, $G()$ aims at taking into account external memory accesses by measuring the number of memory transactions issued to access to the external data.

Moreover, a further indicator has been defined to evaluate the efficiency of a algorithm in term of the computational elements $E[\,]$ usage-degree. On the one hand, $T()$ represents the latency per instruction call, summed over the instructions issued, and, on the other one, $W()$ is the length induced by each instruction, summed over the instructions issued.

Note that $T()$ can be thought as the parallel complexity of the algorithm assuming a collection of k scalar processors, whereas $W()$ can be thought as the serial complexity, assuming we are simulating the execution on a serial RAM [34]. Generally, we are interested in the minimization of $T()$, so as to obtain faster execution; $W()$, instead, is more relevant during the efficiency analysis of the solution we are evaluating. For example, considering the value of $W()$ over $k \cdot T()$ ratio gives us the level of exploitation of the architecture performing the computation of a stream element: values closer to 1 are indicative of a good efficiency.

Eventually, in order to evaluate an algorithm in the K-model, we have to evaluate the stream to compute by multiplying the complexities of a generic stream element by the stream size. Whenever the algorithm is made of more kernels, the whole complexity is defined as the sum over the complexities of the different kernels.

The schema concerning to the internal-work has also a graphic interpre-

tation that can clarify the intention of K-model. In fact, we can consider each instruction as a bin of an histogram that represents the entire kernel execution on a stream element. Concerning each bin, its height is proportional to the length of the instruction as previously described; its width, instead, is proportionally to the related instruction latency. In this point of view, the function $T()$ is denoted by the final width of the histogram and efficiency is given by the ratio computed by dividing the area of histogram, i.e. $W()$ function over the area of the rectangle which base equals $T()$ and height equals k. Therefore, efficiency can be evaluated as $W()/[k \cdot T()]$.

Example. Let us consider the following instantiation of the naïve parallel algorithm to sum a list of elements. Given $k = 4$ scalar execution units, compute the sum of the elements of an array x of $n = 2k$ entries. Algorithm 1 is a sequence of $\log n$ instructions: `(r)` represents the operation of reading data, `(+)` represents the scalar add, and `(w)` represents the write-back operation.

Algorithm 1 Naïve parallel algorithm to sum a list of elements.

1: **For** $d = 0$ **To** $\log n - 1$ **Do**
2: **For** $k = 0$ **To** $n - 1$ **Step** 2^{d+1} **Parallel Do**
3: $a \leftarrow x[k + 2^{d+1} - 1]$, $b \leftarrow x[k + 2^d - 1]$; `// (r)`
4: $a \leftarrow a + b$; `// (+)`
5: $x[k + 2^{d+1} - 1] \leftarrow a$; `// (w)`
6: **End Parallel Do**
7: **End Do**
8: **Return** $x[n - 1]$;

In Figure 3.2, the time and computational complexities are represented. Each bin has a width equal to the latency of the relative operation. For example the smaller width of the `(+)` bars with respect to the `(r)` ones states that `(+)` has a shorter duration than `(r)`. In particular, `(r)` operations have latency equal to 2 because in during the related steps at least one queue of $Q[1..k]$ has collected 2 distinct accesses. The height of each bar is equal to the number of units performing the current instruction. Obviously, since the approach exploits a tree-based computation schema, at each iteration the number of elements to sum halves as the number of involved units.

The case above is toy example, however reduction like that often occurs as building block in many algorithms. The example is particularly useful to show the mean of $T(n, k)$ performance indicator. Let us briefly compute their formulas. Algorithm 1 is composed of $\log n$ steps and the latency of every one is $\geq k$ due to the data access pattern used to address the internal memory. In fact, the naïve implementation leads to concentrate the load/store requests on a subset of queues $Q[\,]$ and, as a consequence, the execution of the corresponding instruction requires more time. In this particular case we can conclude that $T(n, k) = k \cdot \log n$. The main aspect

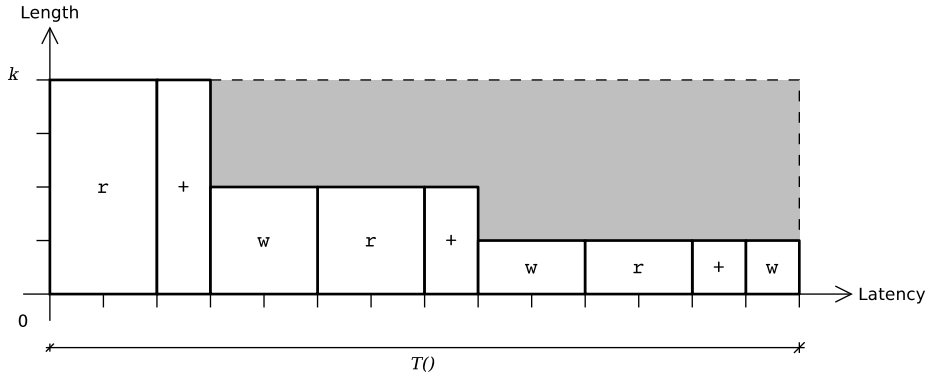


Figure 3.2: Temporal behavior of the parallel sum naïve implementation.

to observe is that, using a naïve approach, we waste k parallel couple of modules, i.e. $(E[i], Q[i])\ _{\forall i \in [1..k]}$, at each step. In practice, performing each step in k time units, in most cases sequential architecture performs better than the parallel one and it cannot be observed adopting a not-properly-defined model. To complete the example we also compute the last two remaining performance indicators. As defined above, $W(n)$ is related to the RAM complexity and it corresponds to $O(n)$. Concerning $G(n, k)$ and by assuming the n values are sequentially stored in the external memory, $n/k = 2k/k = 2$ memory transactions are performed to get the input values into the internal memory and 1 further transaction whenever the sum has to be copied back to the external memory. In this case, the study of $G()$ is not particularly interesting so it losses some of its significance. However in the next case study (Section 3.2) is shown as it is fundamental in more complex algorithm.

3.2 Case Study: Sorting

In [70] we digress from the motivation of sorting efficiently a large amount of data on modern GPUs to propose a novel sorting solution that is able to sort in-place an array of integers. In fact, sorting is a core problem in computer science that has been extensively researched over the last five decades, yet it still remains a bottleneck in many applications involving large volumes of data. Furthermore, sorting constitutes a basic building block for Large Scale Distributed Systems for IR. First of all, as we show in Section 2.3.3, sorting is the basic operation for indexing. Large scale indexing, thus, required scalable sorting. Second, the technique we are introducing here is viable for Distributed Systems for IR since it is designed to run on GPUs that are considered as a basic building block for future generation data-centers [71]. Our bitonic sorting network can be seen as a viable alternative for sorting large amounts of data on GPUs.

During our research we studied a new function to map Bitonic Sorting

Network (BSN) on GPU exploiting its high bandwidth memory interface. We also present this novel data partitioning schema that improves GPU exploitation and maximizes the bandwidth with which the data is transferred between on-chip and off-chip memories. It is worth noticing that being an in-place sorting based on bitonic networks our solution uses less memory than non in-place ones (e.g. [67] and [72]), and allows larger datasets to be processed. Space complexity is an important aspect when sorting large volume of data, as it is required by large-scale distributed system for information retrieval (LSDS-IR).

To design our sorting algorithm in the stream programming model, we started from the popular BSN, and we extend it to adapt to our target architecture. Bitonic sort is one of the fastest sorting networks [73]. Due to its large exploitation bitonic sorting is one of the earliest parallel sorting algorithms proposed in literature [73]. It has been used in many applications. Examples are the divide-and-conquer strategy used in the computation of the Fast Fourier Transform [74], WebIR [40, 75], and some new multicasting network [76].

The main contributions described in this section are the following:

- We perform a detailed experimental evaluation of state-of-the-art techniques on GPU sorting and we compare them on different datasets of different size and we show the benefits of adopting in-place sorting solutions on large datasets.

- Taking into account the performance constraints of our novel computational model [68], we design a method to improve the performance (both theoretical and empirical) of sorting using butterfly networks (like bitonic sorting). Our theoretical evaluation, and the experiments conducted, show that following the guidelines of the method proposed improve the performance of bitonic sorting also outperforming other algorithms.

This section is organized as follows. Section 3.2.1 discusses related works inherent sorting. Section 3.2.2 describes the new function to map BSN on GPU we propose. Section 3.2.3 and Section 3.2.4 presents the results obtained in testing the different solutions on synthetic and real dataset. Section 3.2.5 presents the conclusions and discusses how to evolve in this research activity.

3.2.1 Related Work

In the past, many authors presented bitonic sorting networks on GPUs [77], but the hardware they use belongs to previous generations of GPUs, which does not offer the same level of programmability of the current ones.

Since most sorting algorithms are memory-bound, it is still a challenge to design efficient sorting methods on GPUs.

Purcell *et al.* [78] present an implementation of bitonic merge sort on GPUs based on an original idea presented by [79]. Authors apply their approach to sort photons into a spatial data structure providing an efficient search mechanism for GPU-based photon mapping. Comparator stages are entirely realized in the fragment units[1], including arithmetic, logical and texture operations. Authors report their implementation to be compute-bound rather than bandwidth-bound, and they achieve a throughput far below the theoretical optimum of the target architecture.

In [80, 81] it is shown an improved version of the bitonic sort as well as an odd-even merge sort. They present an improved bitonic sort routine that achieves a performance gain by minimizing both the number of instructions executed in the fragment program and the number of texture operations.

Greß *et al.* [82] present an approach to parallel sort on stream processing architectures based on an adaptive bitonic sorting [83]. They present an implementation based on modern programmable graphics hardware showing that they approach is competitive with common sequential sorting algorithms not only from a theoretical viewpoint, but also from a practical one. Good results are achieved by using efficient linear stream memory accesses, and by combining the optimal time approach with algorithms.

Govindaraju *et al.* [84] implement sorting as the main computational component for histogram approximation. This solution is based on the periodic balanced sorting network method [85]. In order to achieve high computational performance on the GPUs, they used a sorting network based algorithm, and each stage is computed using rasterization. Later, they presented a hybrid bitonic-radix sort that is able to sort vast quantities of data, called GPUTeraSort [77]. This algorithm was proposed to sort record contained in databases using a GPU. This approach uses the data and task parallelism to perform memory-intensive and compute-intensive tasks on GPU, while the CPU is used to perform I/O and resource management.

Cederman *et al.* [67] show that GPU-Quicksort is a viable sorting alternative. The algorithm recursively partition the sequence to be sorted with respect to a pivot. This is done in parallel by each GPU-thread until the entire sequence has been sorted. In each partition iteration, a new pivot value is picked up and as a result two new subsequences are created that can be sorted independently by each thread block. The conducted experimental evaluation point out the superiority of GPU-Quicksort over other GPU-based sorting algorithms.

Sengupta *et al.* [72] present a Radix-sort and a Quicksort implementation based on segmented scan primitives. Authors presented new approaches to implement several classic applications using this primitives, and show that this primitives are an excellent match for a broad set of problems on parallel hardware.

Wassenberg *et al.* [86] present a fast radix sorting algorithm that builds

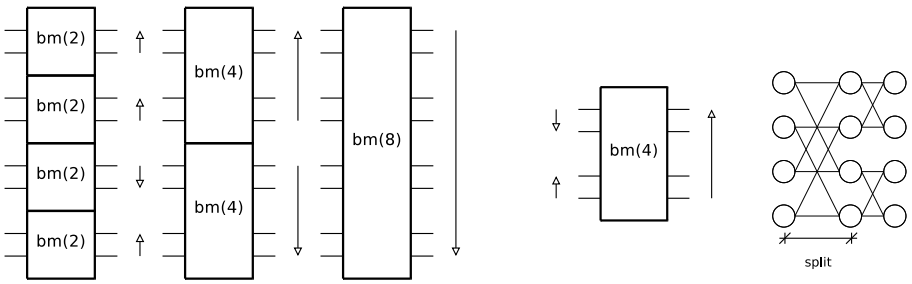
[1]In addition to computational functionality, fragment units also provide an efficient memory interface to server-side data, i.e. texture maps and frame buffer objects.

upon a microarchitecture-aware variant of counting sort. Taking advantage of virtual memory and making use of write-combining yields a per-pass throughput corresponding to at least 89% of the system's peak memory bandwidth. It also compares favorably to the reported performance of GPU-based algorithm when data-transfer overhead is included.

3.2.2 K-Model Oriented Solution

A sorting network is a mathematical model of a sorting algorithm that is made up of a network of wires and comparator modules. The sequence of comparisons thus does not depend on the order with which the data is presented. The regularity of the schema used to compare the elements to sort makes this kind of sorting network particularly suitable for partitioning the elements in the stream programming fashion, as K-model requires.

In particular, BSN is based on repeatedly merging two bitonic sequences[2] to form a larger bitonic sequence [87]. On each bitonic sequence the bitonic split operation is applied. After the split operation, the input sequence is divided into two bitonic sequences such that all the elements of one sequence are smaller than all the elements of the second one. Each item on the first half of the sequence, and the item in the same relative position in the second half are compared and exchanged if needed. Shorter bitonic sequences are obtained by recursively applying a binary merge operation to the given bitonic sequence [73]. The recursion ends and the sequence is sorted when the input of the merge operation is reduced to singleton sequences. Figure 3.3 shows graphically the various stages described above.



(a) A BSN made up of Bitonic Merging Network (b) Bitonic Merging Network

Figure 3.3: (a) Structure of a BSN of size $n = 8$. With bm(x) we denote bitonic merging networks of size x. The arrows indicate the monotonic ordered sequence. (b) Butterfly structure of a bitonic merge network of size $n = 4$.

The pseudo-code of a sequential BSN algorithm is shown in Algorithm 2. Figure 3.4, instead, shows an instantiation of a fan-in 16 BSN.

[2]A bitonic sequence is composed of two sub-sequences, one monotonically non-decreasing and the other monotonically non-increasing.

Algorithm 2 BitonicSort (A)

1: $n \leftarrow |A|$
2: **For** $s = 1$ **To** $\log n$ **Do**
3: **For** $c = s - 1$ **To** 0 **Step** -1 **Do**
4: **For** $r = 0$ **To** $n - 1$ **Do**
5: **If** $\frac{r}{2^c} \equiv \frac{r}{2^s} \pmod 2 \wedge A[r] > A[r \oplus 2^c]$ **Then** SWAP($A[r]$, $A[r \oplus 2^c]$)
6: **End Do**
7: **End Do**
8: **End Do**

To design our sorting algorithm in the stream programming model, we start from the original parallel BSN formulation [73] and we extend it to follow the K-model guidelines. In particular, the main aspect to consider is to define an efficient schema for mapping items into stream elements. Such mapping should be done in order to perform all the comparisons involved in the BSN within a kernel. The structure of the network, and the constraint of the programming model, indeed, disallow the entire computation to be performed within one stream. Firstly, the number of elements to process by the merging step increases constantly (as it is shown in Figure 3.3). Furthermore, due to the unpredictability of their execution order, the model requires the stream elements to be independently computable. This implies that each item has to be included into at most one stream element, see Figure 3.5. Following these constraints, the set of items would then be partitioned (and successively mapped) into fewer but bigger parts as shown in Figure 3.6. For example, referring to the last merging step of a BSN all the items would be mapped into a unique part. This is clearly non admissible since the architectural constraints limit the number of items that can be stored (and shared) locally (i.e. the size of a stream element). In particular in the K-model, such limit is fixed by σ that is the amount of memory available for each stream element, see Figure 3.1.

In our solution we define different partition depending on which step of the BSN we are. Each partitioning induces a different stream. Each stream, in turn, needs to be computed by a specific kernel that efficiently exploits the characteristic of the stream processor.

Since each kernel invocation implies a communication phase, such mapping should be done in order to reduce the communication overhead. Specifically, this overhead is generated whenever a processor begins or ends the execution of a new stream element. In those cases, the processor needs to flush the results of the previous computation stored in the local memory, and then to fetch the new data from the off-chip memory. Taking into account the K-model rule, depending on the pattern used to access the external memory, the measure denoting the time required for such transfer can increase up to k times and translating in an increase of up to one order

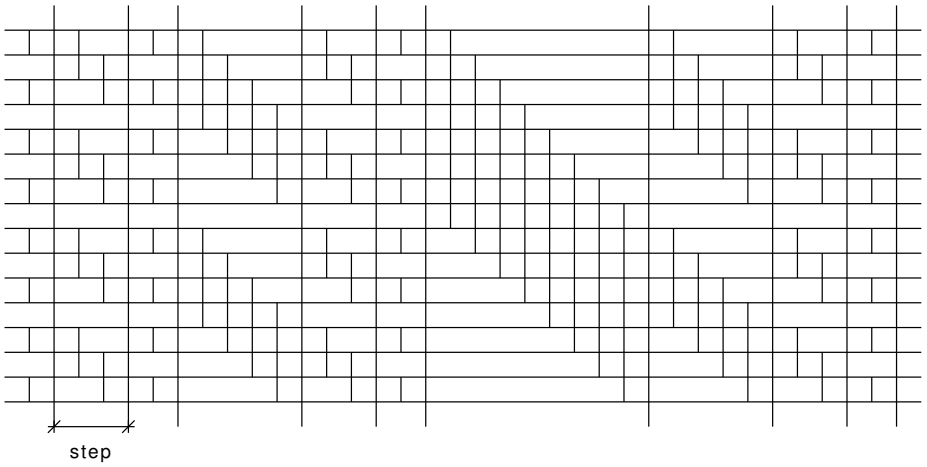


Figure 3.4: Example of BSN for 16 elements. Each comparison is represented with a vertical line that link two elements, which are represented with horizontal lines. Each step of the sort process is completed when all comparisons involved are computed.

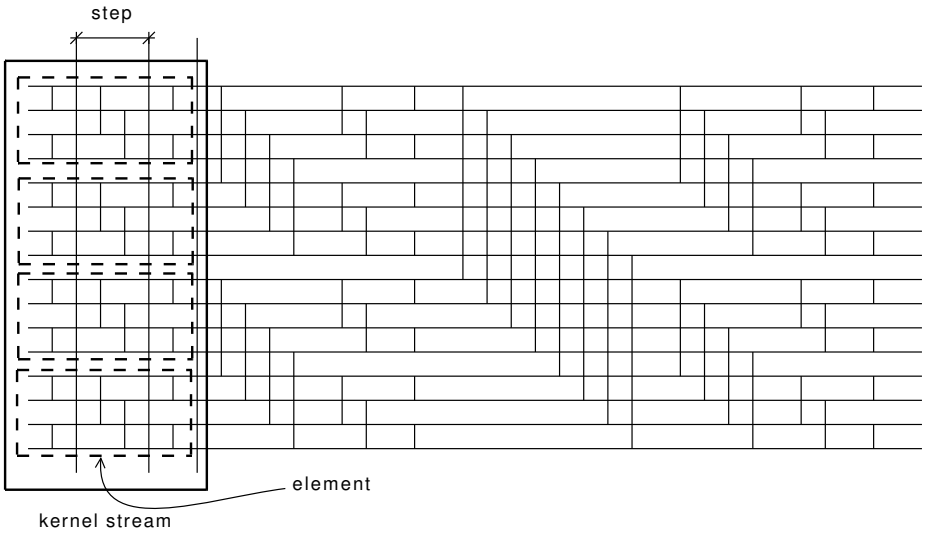


Figure 3.5: Example of a kernel stream comprising more steps of a BSN. The subset of items composing each element must perform comparison only inside itself.

of magnitude when measured on the real architecture.

Resuming, in order to maintain the communication overhead as small as possible, our goals are: (*i*) to minimize the number of communications between the on-chip memory and the off-chip one, (*ii*) to maximize the bandwidth with which such communications are done. Interestingly, the sequential version of the bitonic network algorithm exposes a pattern made up of repeated comparisons. It turns out that this core set of operations

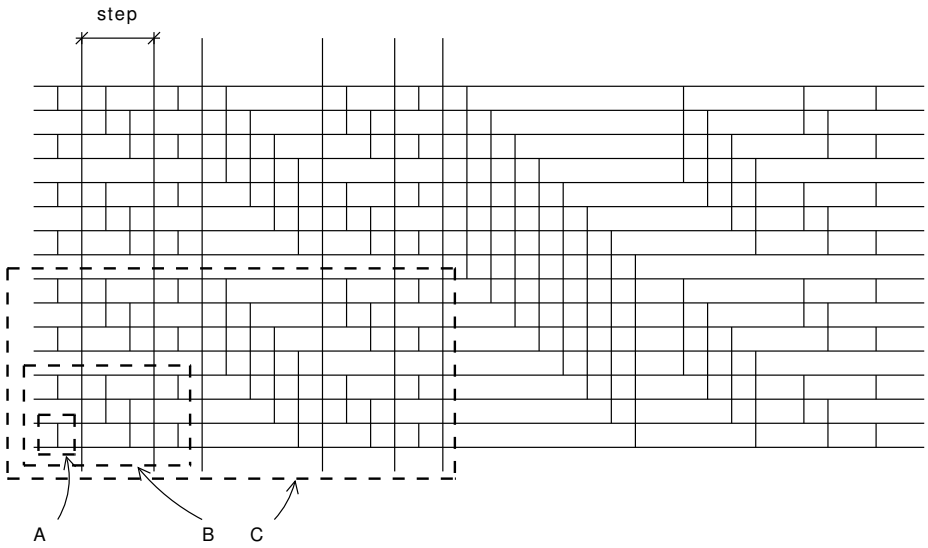


Figure 3.6: Increasing the number of steps covered by a partition, the number of items included doubles. A, B and C are partitions respectively for local memory of 2, 4 and 8 locations.

can be then optimally reorganized in order to meet the two goals above described.

Let us describe how a generic bitonic network sorting designed for an array A of $n = 2^i$ items, with $i \in \mathbb{N}^+$, can be realized in K-model.

In order to avoid any synchronization, we segment the n items in such a way each part contains all the items to perform some steps without accessing the items in other parts. Since the items associated with each stream element have to be temporarily stored in the on-chip memory, the number of items per part is bounded by the size of such memory. In the follow, we show the relation between the number of items per part, and the number of steps each kernel can perform. This relation emerges from the analysis of Algorithm 2.

Briefly, to know how many steps can be included in the run of a partition, we have to count how many distinct values the variable c can assume. First of all, by the term *step* we refer to the comparisons performed in the loop at line 4 of Algorithm 2. Furthermore, let c and s be the variables specified in Algorithm 2, the notation $step_{\bar{s},\bar{c}}$ represents the step performed when $c = \bar{c}$ and $s = \bar{s}$. At each step, the indexes of the two items involved in the comparison operation are expressed as a function of the variable c.

Claim 1. *Within a $step_{s,c}$ two elements are compared, if and only if, the binary representation of their relative indexes differ only by the c-th bit.*

Proof. By definition of bitwise $\oplus$ the operation $r \oplus 2^c$, invoked at line 5, corresponds to flipping the c-th bit of r, in its binary representation. □

The claim above gives a condition on the elements of the array A involved

in each comparison of a step. Given an element $A[r]$ at position r this is compared with the one whose position is obtained by fixing all the bits in the binary representation of r but the c-th one which is, instead, negated. The previous claim can be extended to define what are the bits flipped to perform the comparisons done within a generic number of consecutive steps, namely k, called *k-superstep*[3]. This definition straightforwardly follows from the Algorithm 2, and it is divided in two cases, specifically for $k \leq s$ and $k > s$.

Definition 1 (Γ-sequence)**.** *Within the k-superstep starting at* $step_{s,c}$*, with* $1 \leq k \leq s$*, the sequence* Γ *of bit positions that Algorithm 2 flips when it performs the comparisons is defined as follows:*

$$\Gamma = \begin{cases} \Gamma_{\geq} = [c,\, c-k) & \text{if } c > k \\ \Gamma_{<} = [s,\, s-k+c+1) \cup [c,\, 0] & \text{otherwise} \end{cases}$$

The sequence basically consists of the enumeration of the different values taken by c in the k-superstep considered. It is worth being noted that the values assigned to c in the k steps are distinct because of the initial condition $k \leq s$. Now, let us consider the behavior of the Algorithm 2 when $s < k$. In particular, let us restrict to the definition of Γ in steps from $step_{1,0}$ to $step_{k,0}$. Since c is bounded from above by $s < k$, for each considered step c can only assume values in the range $(k, 0]$. Note that, in this case, the number of steps covered by flipping the bit positions contained in the sequence is $\frac{1}{2}k(k+1)$, instead of k.

Definition 2 (Γ_0-sequence)**.** *The sequence* $\Gamma_0 = (k, 0]$ *corresponds to bit positions that Algorithm 2 flips when it performs the comparisons within the* $\frac{1}{2}k(k+1)$ *steps starting from* $step_{1,0}$*.*

To resume, given a generic element $A[r]$, with $0 \leq r < n$, and considering a superstep of the bitonic network, the only bits of r flipped by Algorithm 2 to identify the corresponding elements to compare with are those identified by the sequence Γ of bit positions. Then, bit positions that do not occur in Γ are identical for the elements compared with $A[r]$ in such superstep. By definition of the Γ-sequence, we can retrieve the following claim.

Claim 2. *Let* $A[r]$ *and* $A[q]$ *be two elements of A. Given a superstep and its* Γ*-sequence,* $A[r]$ *and* $A[q]$ *belong to the same partition if and only if* $\forall i \notin \Gamma.\ r_{[i]} = q_{[i]}$*, where the notation* $r_{[i]}$ *denotes the i-th bit of the binary representation of r.*

From the previous claims, we can also retrieve the size of each partition as function of Γ.

[3]In the rest of the section we denote a sequence of integers by putting the greater value on the left of the range. For example, the sequence formed by the elements in $\{i |\, m \leq i < M\}$ is denoted by $(M, m]$.

Lemma 1. *Each part is composed by* $2^{|\Gamma|}$ *items.*

Proof. By induction on the length of Γ. When $|\Gamma| = 1$, an item is compared with only another one, by Claim 1. So each part is made up of 2 items. For the inductive step, let us consider the next step in the superstep. Each of the previous items is compared with an element not yet occurred, due to the new value of c that implies to flip a new bit position. Since each item forms a new pair to compare, the number of items to include in the part doubles, namely it is $2 \times 2^{|\Gamma|} = 2^{|\Gamma|+1}$. □

From the above lemma, and because each partition covers all the elements of A, it follows directly that

Corollary 1. *The number of parts for covering all the comparisons in the superstep is* $2^{\log n - |\Gamma|}$.

The previous claim can be extended to define the Γ-partition procedure.

Definition 3 (Γ-partition)**.** *Given a* k*-superstep, the relative* Γ*-partition is the set of parts* $\mathcal{P} = \{p_i\}$*, for* $0 \leq i < 2^{\log n - |\Gamma|}$ *where each part is constructed by means of Algorithm 3.*

Algorithm 3 BUILDPARTITION (A, n, k, Γ)

```
1: /* create a bit-mask corresponding to the fixed log n − k bits
   whose positions are not in Γ */
2: j = 0, m = 0;
3: For b = 0 To log n − 1 Do
4:   If b ∉ Γ Then m_[b] = i_[j], j = j + 1;
5: End Do
6: /* populate the partition using the bit-mask m defined in the
   previous step */
7: For e = 0 To 2^k − 1 Do
8:   j = 0, r = m;
9:   For b = 0 To ⌈log n⌉ − 1 Do
10:    If b ∈ Γ Then r_[b] = e_[i], i = i + 1;
11:  End Do
12:  p_i = p_i ∪ A[r];
13: End Do
```

Let us show an example of how Γ-partition works.

Example. Let us consider a bitonic network at $step_{2,2}$, and the k-superstep of length $k = 2$. Since, $\bar{c} \geq k - 1$, then $\Gamma = \Gamma_{\geq} = [\bar{c}, \bar{c} - k) = [2, 0) = [2, 1]$. We firstly create for each part p_i a bit-mask m_i corresponding to the fixed $\lceil \log n \rceil - k$ bits whose positions are not in Γ, namely:

$$p_0 \Rightarrow m_0 = \mathbf{0}\,00\,\mathbf{0} \quad p_1 \Rightarrow m_1 = \mathbf{0}\,00\,\mathbf{1} \quad p_2 \Rightarrow m_2 = \mathbf{1}\,00\,\mathbf{0} \quad p_3 \Rightarrow m_3 = \mathbf{1}\,00\,\mathbf{1}$$

Where the bold-font bits refer the bits do not occur in Γ. Now, we populate each partition by using the relative bit-mask m_i defined in the previous step:

$$p_0 = \begin{cases} \mathbf{0}\,00\,\mathbf{0} \\ \mathbf{0}\,01\,\mathbf{0} \\ \mathbf{0}\,10\,\mathbf{0} \\ \mathbf{0}\,11\,\mathbf{0} \end{cases} \quad p_1 = \begin{cases} \mathbf{0}\,00\,\mathbf{1} \\ \mathbf{0}\,01\,\mathbf{1} \\ \mathbf{0}\,10\,\mathbf{1} \\ \mathbf{0}\,11\,\mathbf{1} \end{cases} \quad p_2 = \begin{cases} \mathbf{1}\,00\,\mathbf{0} \\ \mathbf{1}\,01\,\mathbf{0} \\ \mathbf{1}\,10\,\mathbf{0} \\ \mathbf{1}\,11\,\mathbf{0} \end{cases} \quad p_3 = \begin{cases} \mathbf{1}\,00\,\mathbf{1} \\ \mathbf{1}\,01\,\mathbf{1} \\ \mathbf{1}\,10\,\mathbf{1} \\ \mathbf{1}\,11\,\mathbf{1} \end{cases}$$

Now, let us make some consideration about the communication overhead discussed above. Each time we perform a stream, the computation is charged of the time spent to fetch all n elements divided among the different parts, then to write them back. In order to minimize this overhead, we need to minimize the number of streams needed to cover all the network, i.e. to maximize the number of steps performed within each partition. Because each Γ-sequence is made up of $2^{|\Gamma|}$ items, see Lemma 1, and in the K-model the data of each part has to fit in the local memory of σ locations, the optimal size for Γ is $\log \sigma$. Then, each Γ-partition forms a stream that feeds an appropriate kernel. Due to the mapping we design, each part is modeled as a bitonic network (see Algorithm 4). It is possible to show that such a modeling allows to always keep the k executors active. At the same time, the contention to access the k on-chip memory banks is balanced. Note that, in the K-model rule, by balancing the contention the latency of the accesses is reduced because the maximum contention is lower.

Algorithm 4 RUNSTREAMELEMENT (A_p, Γ)

```
1:  For Each id ∈ [0, k − 1] Parallel Do
2:    n = log2(σ)
3:    For i = 0 To n − 1 Do
4:      c = Γ[i]
5:      For j = id To n/2 − 1 Step k Do
6:        p = InsAt (j, 0, c)
7:        q = InsAt (j, 1, c)
8:        Compare&Swap (A_p[p], A_p[q])
9:      End Do
10:   End Do
11: End Parallel Do
```

The pseudo-code in Algorithm 4 discards some side aspects, to focus the main technique. In particular it takes a part (A_p) and the related Γ-sequence (Γ), then performs all due comparisons in-place. The procedure InsAt (N, x, p) inserts the bit x at the position c of the binary representation of N, for example InsAt $(7, \mathbf{0}, 1) = 11\mathbf{0}1 = 13$. The procedure Compare&Swap performs the comparisons between the argument elements and, if needed, swaps them. Note that, each RUNSTREAMELEMENT execution is free from conditional branch instructions. This is a very important feature

for a SIMD algorithm, avoiding, in fact, divergent execution paths that are serialized by the (single) instruction unit of the processor.

In a previous work we argued that minimizing the number of data-transfers is not enough [40, 75]. In particular, in the cache-based model, proposed by [39], the bandwidth needed to replace a cache-line, in the case of cache-miss, is constant. Following the K-model rules [68], the memory bandwidth is fully exploited when simultaneous memory accesses can be coalesced into a single memory transaction. This means that it is possible to reduce the latency of data transfer by reorganizing in the proper manner the accesses to the off-chip memory.

In the rest of the section we will refer a sequence of k consecutive addresses with the term *k-coalesced set*, and we will say that a part, or the associated Γ-sequence, satisfies the *k-coalesced condition* when its values are related only to sets that are k-coalesced. Specifically, for Definition 3, such a Γ-sequence satisfies the *k-coalesced condition* when it contains all the values in the range from 0 to $\log k - 1$.

Let us analyze the k-coalesced condition in the K-model. By definition of Γ-sequence, when we fall into a $\Gamma_<$ case and $c > \log k$, the k-coalesced condition is verified because the Γ-partition accesses to 2^{c+1}-coalesced subsets of positions. When $c \leq \log k$, and we are still in the $\Gamma_<$ case, we need to access to longer consecutive sequences of addresses to satisfy the k-coalesced condition. On the other hand when we fall into a $\Gamma_\geq$-sequence, no consecutive addresses are included in the relative partitions, because the value 0 cannot be included in such type of sequence, for Definition 1. Eventually, the Γ_0 sequence is composed of a unique sequence of contiguous addresses.

To satisfy the k-coalesced condition for all the generated Γ-partitions, we move some pairs of items from a part of the current partition to another part. The aim is to group in the same memory transaction items having consecutive addresses, whenever we need longer sequences of consecutive memory addresses. To do that, each Γ-sequence is initialized with the values in the range $]\log k, 0]$. Then, the values of c related to the next steps to perform are pushed in the Γ-sequence as far as it contains $\log \sigma$ distinct values.

This operation augments the coalescing-degree of the data transfer, still it forces to remove from Γ some elements related to the next values of c. The best possible communication bandwidth is attained at the cost of decreasing the length of some supersteps. This means to perform more supersteps to cover the whole bitonic network.

3.2.3 Evaluation & Experiments

The solution we propose is evaluated theoretically and experimentally by comparing its complexity and performance with those obtained by [67] with their version of quick-sort (hereinafter referred to as Quicksort), and the radix-sort based solution proposed by [72] (hereinafter referred to as Radix-

sort). Quicksort exploits the popularly known divided-and-conquer principle, whereas Radixsort exploits the processing of key digits.

Theoretical Evaluation

BSN. The number of steps to perform is $(\log^2 n + \log n)/2$. To estimate the number of *memory transaction* needed to compute a sorting network for an array of size n, we have to count the number of Γ-partitions needed to cover all the network. That means to know how many stream elements are computed, then the number of fetch/flush phases, and the number of memory transactions.

From Definition 2, it follows that the first partition covers the first $(\log^2 \sigma + \log \sigma)/2$ steps.

Let us call $stage_s$ the loop at line 2 of Algorithm 2. In the remaining steps $s > \sigma$, $\log n - \log \sigma$ stages remain, and each of them has the last Γ-partition covering $\log \sigma$ steps. On the other hand the $s - \log \sigma$ steps are performed with partitions covering $\log(\sigma/k)$ steps. Resuming, the number of partitions needed to cover all the network is

$$1 + \sum_{s=\log \sigma + 1}^{\log n} \left(\left\lceil \frac{s - \log \sigma}{\log(\sigma/k)} \right\rceil + 1 \right) = O\left(\frac{\log^2 n}{\log k} \right)$$

Since, each element fetches and flushes only coalesced subset of elements, the number of transactions is

$$O\left(\frac{n}{k} \cdot \frac{\log^2 n}{\log k} \right)$$

The *time complexity* is

$$O\left(\frac{n \log^2 n}{k} \right)$$

as it is obtained by Algorithm 4 which equally spreads the contentions among the k memory banks and maintains active all elements.

Regarding the *computational complexity* it is known and it is

$$O(n \log^2 n)$$

Quicksort. It splits the computation in $\log n$ steps. For each step it performs three kernels. In the first one, it equally splits the input and counts the number of elements greater than the pivot, and the number of the elements smaller than the pivot. In the second, it performs twice a parallel prefix sum of the two set of counters in order to know the position where to write the elements previously scanned. In the final kernel, it accesses to the data in the same manner that in the first kernel, but it writes the elements to the two opposite heads of an auxiliary array beginning at the positions calculated in the previous kernel.

The first kernel coalesces the access to the elements and, since the blocks are equally sized, also the computation is balanced. Then the counters are flushed, and the second kernel starts. Supposing that $n/k < \sigma$, each prefix sum can be computed within a unique stream element. Consequently, for each prefix sum we need n/k^2 *memory transactions* to read n/k counters. The *time complexity* is logarithmic in the number of elements, on the contrary the *computational complexity* is linear. Last kernel is similar to the first one, except for flushing the data into the auxiliary array. In particular, because each thread accesses to consecutive memory locations, the main part of the requests is not coalesced, requesting one memory transaction per element.

Table 3.2.3 contains the evaluation of the three type of kernel in the K-model. In order to compute the complexity of the whole algorithm, the sum of such formulas have to be multiplied by $\log n$.

	memory transactions $G()$	*time complexity* $T()$	*computational complexity* $W()$
kernel #1	$n/k + 2$	n/k	n
kernel #2	$4n/k^2$	$4 \cdot \log \frac{n}{k}$	$2n/k$
kernel #3	$n/k + n$	n/k	n
OVERALL	$O(n \log n)$	$O(\frac{n}{k} \log n)$	$O(n \log n)$

Radixsort. It divides the sequence of n items to sort into h-sized subsets that are assigned to $p = \lceil n/h \rceil$ blocks. Radixsort reduces the data transfer overhead exploiting the on-chip memory to locally sort data by the current radix-2^b digit. Since global synchronization is required between two consecutive phases, each phase consists of several separate parallel kernel invocations. Firstly, each block loads and sorts its subset in on-chip memory using b iterations of binary-split. Then, the blocks write their 2^b-entry digit histogram to global memory, and perform a prefix sum over the $p \times 2^b$ histogram table to compute the correct output position of the sorted data. However, consecutive elements in the subset may be stored into very distant locations, so coalescing might not occur. This sacrifices the bandwidth improvement available, which in practice can be as high as a factor of 10.

In their experiments, the authors obtained the best performance by empirically fixing $b = 4$ and $h = 1024$. That means each stream is made up of $\lceil n/1024 \rceil$ elements. Once the computation of a stream element ends, the copies of the sorted items may access up to $O(2^b)$ non-consecutive positions. Finally, considering 32-bit words, we have $32/b$ kernels to perform. This leads to formalize the total number of *memory transactions* performed as follows:

$$O\Big(\frac{32}{b} \cdot \frac{n}{h} \cdot 2^b\Big)$$

Regarding *computational* and *time complexity*, Radixsort does not use expensive patterns and it does not increase the contention in accessing shared memory banks. Therefore, the *time complexity* is given, by $b \cdot n/k$, and the *computational complexity* is linear with the number of input-elements, i.e. $b \cdot n$.

Experimental Evaluation

The experimental evaluation is conducted by considering the execution time and amount of memory required by running BSN, Quicksort and Radixsort on different problem size. The different solutions have been implemented and tested on an Ubuntu Linux Desktop with an Nvidia 8800GT, that is a device equipped with 14 SIMD processors, and 511 Megabytes of external memory. The compiler used is the one provided with the Compute Unified Device Architecture (CUDA) SDK 2.1 [88]. Even if the CUDA SDK is "restricted" to Nvidia products, it is conform to the K-model. To obtain stable result, for each distribution, 20 different arrays were generated.

According to [89], a finer evaluation of sorting algorithms should be done on arrays generated by using different distributions. We generate the input array according to uniform, gaussian and zipfian distributions. We also consider the special case of sorting an all-zero array[4]. These tests highlight the advantages and the disadvantages of the different approach tested. The computation of Radixsort and BSN is based on a fixed schema that uses the same number of steps for all type of input dataset; on the contrary, Quicksort follows a divide and conquer strategy, so as to perform a varying number of steps depending on the sequence of recursive procedures invoked. The benefits of the last approach are highlighted in the all-zero results.

The experiments confirm our theoretical ones. Figure 3.7 and Figure 3.8 show the means, the standard deviation, and the maximum and the minimum of the execution time obtained in the conducted tests. Radixsort results to be the fastest and this is mainly due to its complexity in terms of the number of memory transactions that it needs, see Table 3.2. This confirms our assumption that the number of memory transactions is dominant with respect to the time complexity measure, i.e. $T()$. This is particularly true whenever the cost of each operation is small if compared to the number of memory access operations (like in the case of data-intensive algorithms). Radixsort, in fact, has a $O(n)$ number of memory transactions, that is smaller than $O(n \log^2 n/k \log k)$ of the BSN and than $O(n \log n/k)$ of the Quicksort. Considering the specifications of real architectures, which related to the parameter $k \simeq 16$ of the K-model, and considering the capacity of the external memory available on real devices (order of Gigabytes), Quicksort results to be the least performing method analyzed. Figure 3.9 shows the behaviour of $G()$ for the different solutions.

[4] All elements are initialized equal to 0.

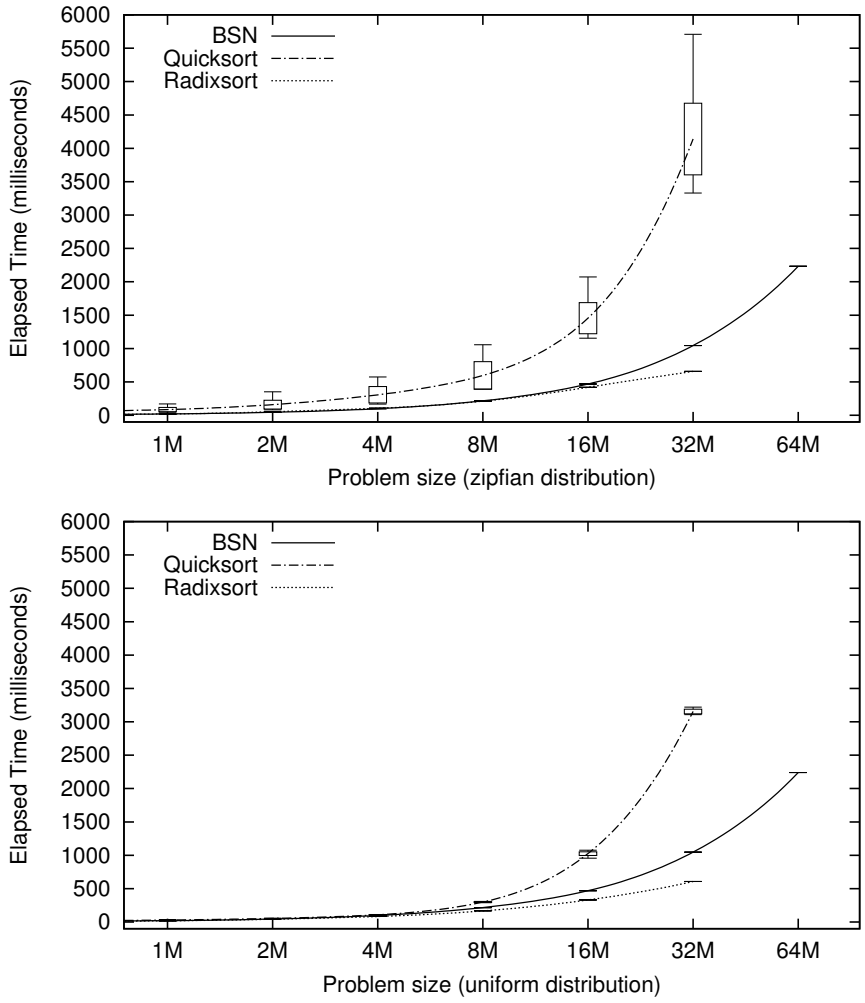


Figure 3.7: Elapsed sorting time for varying input size generated with zipfian and uniform distributions.

A last word has to be spent regarding the memory consumption of the methods. Quicksort and Radixsort are not able to sort large arrays, as it is pointed out, see Table 3.2. Being an in-place solution, in fact, BSN can thus devote all the available memory to store the dataset. This has to be carefully considered since sorting large datasets will require less passes than the other solutions. They need, in fact, to split the sorting process in more steps, then to to merge the partial results. Moreover, merge operation may require further transfers for copying the partial results to the device memory if this operation is performed on manycores. Otherwise, CPU can perform the merging step, but exploiting a bandwidth lower than the GPU's one.

On the other hand, our BSN approach is comparable to Radixsort and it is always faster than Quicksort, mainly because the mapping function proposed allows the full exploitation of the available memory bandwidth.

Table 3.2 measures the memory contention, and the number of divergent paths. The first value measures the overhead due to the contention on

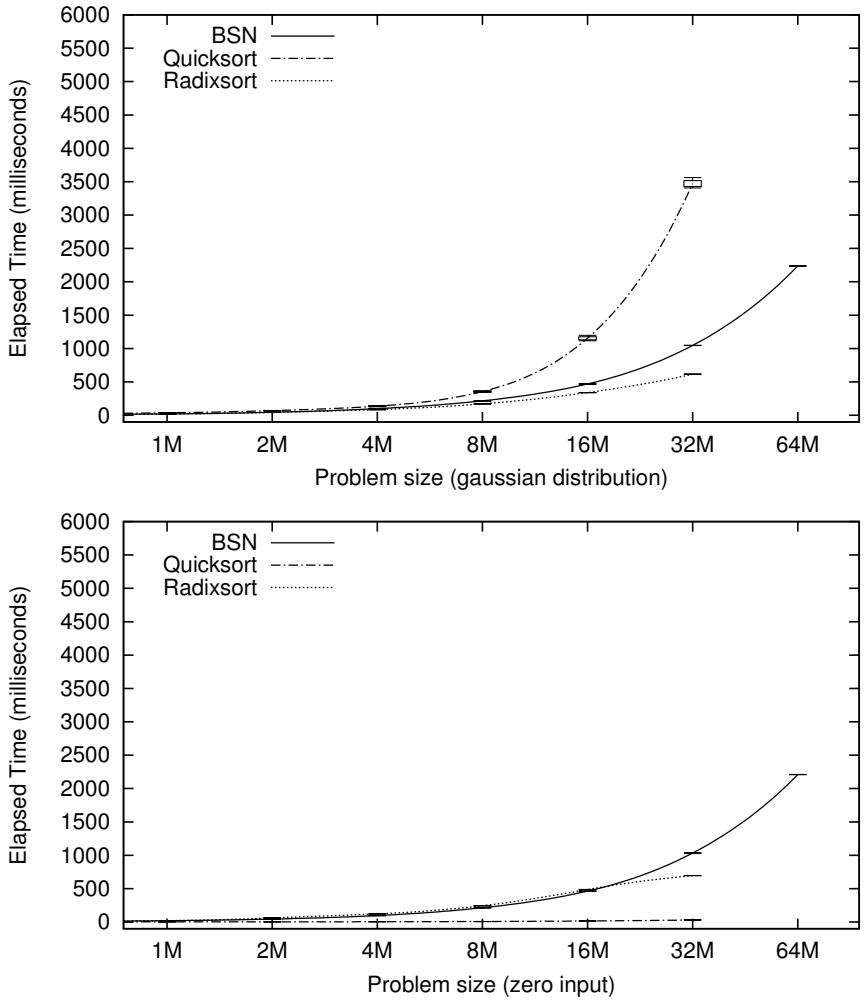


Figure 3.8: Elapsed sorting time for varying input size generated with gaussian and all-zero distributions.

the on-chip memory banks as K-model expects. The second value measures how many times threads of the multiprocessors can not work simultaneously. These two last metrics together show the efficiency of the algorithms tested. Keeping low both values corresponds to a better exploitation of the inner parallelism of the SIMD processor. All the memory banks and all the computational, in fact, are allowed to work simultaneously.

Moreover, since direct manipulation of the sorting keys as in Radixsort is not always allowed, it is important to provide a better analysis of the comparison-based sorting algorithms tested. Due to the in-place feature and due to the best performance resulting from the test conducted, BSN seems more preferable than Quicksort. Furthermore, BSN exposes lower variance in the resulting times, it is equal to zero in practice. On the contrary, depending on the distribution of the input data, Quicksort's times are affected by great variance. Probably, this is due to how the divide-et-impera tree grows depending on the pivot chosen, and on the rest of the

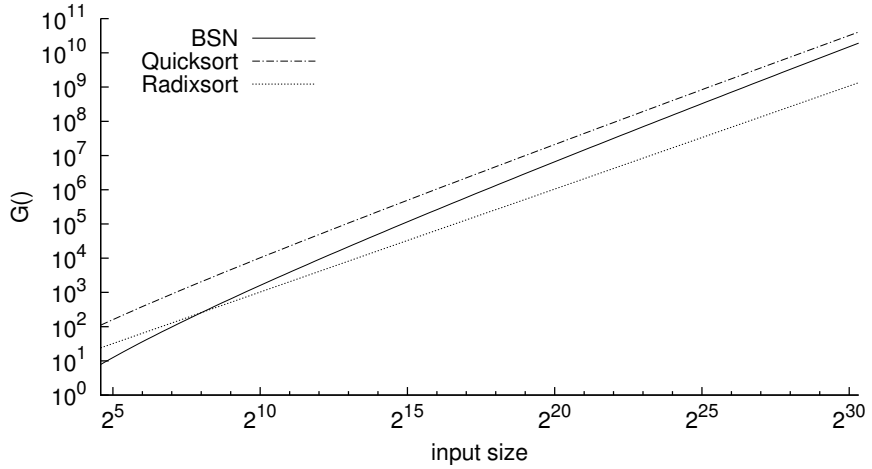


Figure 3.9: Number of memory transactions required by BSN, Radixsort, and Quicksort algorithims.

Problem Size	Solution	Memory Transactions	Memory Contention	Divergent Paths
2^{20}	Bitonicsort	796800	34350	0
	Quicksort	4446802	123437	272904
	Radixsort	965791	132652	29399
2^{22}	Bitonicsort	4119680	151592	0
	Quicksort	18438423	379967	1126038
	Radixsort	3862644	520656	122667
2^{24}	Bitonicsort	20223360	666044	0
	Quicksort	85843422	1379155	1126038
	Radixsort	15447561	2081467	492016
2^{26}	Bitonicsort	101866786	2912926	0
	Quicksort	n.a.	n.a.	n.a.
	Radixsort	n.a.	n.a.	n.a.

Table 3.2: Performance of BSN, Radixsort and Quicksort in terms of number of memory transactions, memory contention, and number of divergent paths. Results are related to uniform distribution. "n.a." means that computation is not runnable for lack of device memory space.

input. For example, on system based on multi-devices (i.e. more than one GPU), this result increases the overhead of synchronization among the set of available devices.

3.2.4 Indexing a Real Dataset

This section describes the results obtained by indexing a collection of documents crawled on Web. In practice, our Web crawler has generated varying

size collections up to 26 million documents of about 200 KB per document. Then, such datasets have been used as input to evaluate the benefits of plugging a GPU-based sorter in a Blocked Sort-Based Indexer (BSBI).

We expect that using the GPU leads to reduce the execution time of the indexing process for two main reasons: (i) GPUs perform the sort process faster than CPUs, (ii) Within BSBI, CPU and GPU can be exploited in pipeline fashion. This means that each block of pages is first parsed by the CPU, then the pairs sorting, which is the most costly part of the elaboration, is performed independently by the GPU while the CPU carries out a new block of *termId-docId* pairs.

INDEX$_{cpu}$ (D) ::

1: $n \leftarrow 0$
2: `while` $D \neq \emptyset$ `do`
3: $n \leftarrow n + 1$
4: $B \leftarrow$ NextBlock()
5: Parse(B)
6: Sort(B)
7: toDisk(B, f_n)
8: $D \leftarrow D \setminus B$
9: $f_{out} \leftarrow$ Merge($f_1, \ldots, f_n$)

INDEX$_{gpu}$ (D) ::

1: $n \leftarrow 0$
2: `while` $D \neq \emptyset$ `do`
3: $n \leftarrow n + 1$
4: $B \leftarrow$ NextBlock()
5: Parse(B)
6: GpuSort(B)
7: toDisk(B, f_n)
8: $D \leftarrow D \setminus B$
9: $f_{out} \leftarrow$ Merge($f_1, \ldots, f_n$)

INDEX$_{gpu^+}$ (D) ::

- CPU(D) ::
  1: `while` $D \neq \emptyset$ `do`
  2: $B \leftarrow$ NextBlock()
  3: Parse(B)
  4: `SyncSend`(B, `toGpu`)
  5: $D \leftarrow D \setminus B$
  6: `SyncSend`($\emptyset$, `toGpu`)
  7: `Recv`(n, `fromGpu`)
  8: $f_{out} \leftarrow$ Merge($f_1, \ldots, f_n$)
- GPU() ::
  1: $n \leftarrow 0$
  2: `while` $B \neq \emptyset$ `do`
  3: `Recv`(B, `fromCpu`)
  4: `if` $B \neq \emptyset$ `then`
  5: $n \leftarrow n + 1$
  6: GpuSort(B)
  7: toDisk(B, f_n)
  8: `SyncSend`(n, `toCpu`)

Figure 3.10: Description of the different versions of the indexer architecture.

With respect to CPU, GPU offers a considerable computing power, but its exploitability is bounded by the available memory size. In fact, as said in the previous section, in our study we used a GPU is equipped with 511 Megabytes of memory, which corresponds to the maximum pairs block size usable in our tests (blocks are referred as B in Figure 3.10). This aspect could affect the overall computational time of indexing. Indeed, given a collection of documents (D), the procedure that merges the pairs blocks has a greater number of files ($f_{1\ldots n}$) to join, and this number affects the merge algorithm complexity of a logarithmic factor (detailed complexity analysis is given in the following). However, the time spent for merging

is dominated by the latency for data transferring from/to the disk. Then, because the whole data to transfer (namely the *termId-docId* pairs) is the same in both the approaches (i.e. the CPU-based and the GPU-based ones), we expect that the time needed by the two solutions for the merging phase is similar.

In order to quantitatively evaluate the contribution given by a GPU-based sorter we developed three different indexing algorithms: (i) a CPU-base indexer, called INDEX$_{cpu}$, (ii) an indexer that alternates the parsing phase on the CPU, with the sorting phase on the GPU, called INDEX$_{gpu}$; (iii) an indexer that performs in pipeline fashion, called INDEX$_{gpu^+}$. Figure 3.10 shows such algorithms expressed in pseudo code. In the two last cases we chose BSN as sort algorithm.

Table 3.3 shows the indexer computational times by varying the size of the collection. The computational time spent by each indexer version has been split in two parts: the time needed to create the n sorted files and the time spent for their merging, represented by the "Parse+[Gpu]Sort" and "Merge" columns, respectively. Concerning the two GPU-based methods, "Merge" column is shown once because the solutions perform the same merge procedure on the same input.

Figure 3.11 shows the overall computational times of the three algorithms when running our case of study. They point out the computational time benefits obtained by using the graphics co-processor. It can be seen that the sorting phase is no more the "bottleneck" of the indexer. On the contrary, considering the GPU-based approaches the computational time is dominated by the latency of the merging phase. In fact, given the n files ($f_{1\ldots n}$ in Figure 3.10) to merge the complexity of parsing is linear in the size of B, that is $O(\, n \cdot |B|\,)$. Furthermore, the complexity of the merging phase is equal to $O(\, n \cdot |B| \cdot \log n\,)$, because n-way merger is realized with an heap composed of as many elements as the files to merge are. At each step Merge procedure extracts the minimum pair x from the heap and a new element, taken from the source file of x, is inserted. The extraction has constant time,

$\|D\|$	INDEX$_{cpu}$		INDEX$_{gpu}$	INDEX$_{gpu^+}$	INDEX$_{gpu,\, gpu^+}$
	Parse+Sort	Merge	Parse+GpuSort	Parse+GpuSort	Merge
0.81	42.9	16.8	12.6	9.0	19.8
1.63	94.5	35.5	24.9	20.2	42.0
3.27	185.2	88.7	52.0	38.0	100.9
6.55	394.5	193.4	107.9	73.9	244.6
13.10	783.5	482.3	221.3	151.5	599.8
26.21	1676.8	1089.2	456.4	346.8	1291.7

Table 3.3: Comparison of the computational times referring the different approach for the indexer architecture (times are in seconds, $|D|$ in millions of documents).

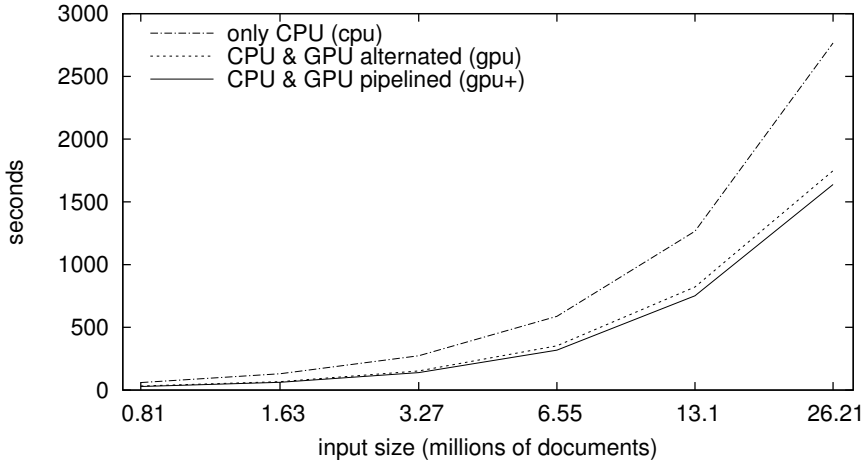


Figure 3.11: Elapsed time for different indexer architectures.

but the insertion is $O(\log m)$ given a m-size heap, then the complexity of the merging phase corresponds to the number of pairs collected multiplied by $\log n$, that is $O(n \cdot |B| \cdot \log n)$. This leads the complexity of merging to be higher than the one corresponding to parsing.

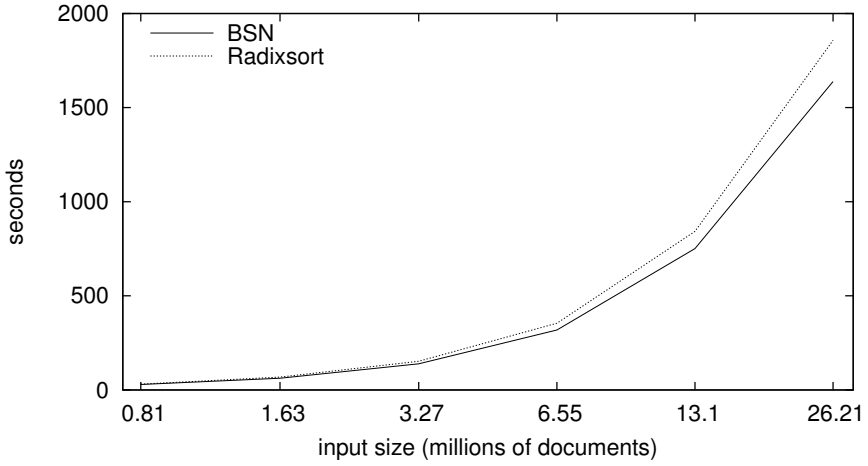


Figure 3.12: Performance of INDEX$_{gpu^+}$ by varying GpuSort() solutions.

Figure 3.12 shows the computational times obtained in the last test we have conducted to compare different versions of INDEX$_{gpu^+}$ by varying the GPU sorting algorithm. To this end, we did not consider GPU-based Quicksort because the results shown in the previous section point out that such solution is both slower and space inefficient. Also if Radixsort is a bit faster than our solution, due to its space complexity, we are forced to use blocks made up of half documents with respect to the number we can sort using BSN. As consequence, the merging phase is more costly due to the increased number of files to merge.

3.2.5 Conclusions

In this section we initially propose a new mapping of Bitonic Sorting Network on stream processing architectures. We start from its traditional algorithm, and we extend it to adapt to our target architecture. Bitonic Sorting Network is one of the fastest sorting networks to run on parallel architectures. The proposed solution was evaluated both theoretically and experimentally, by comparing its complexity and performance with those obtained by two others state-of-the-art solutions.

The theoretical algorithms complexity was evaluated by using K-model a novel computational model specifically designed to capture important aspects in stream processing architectures. The experimental evaluation was conducted using input streams generated according to different distributions. This kind of experiments highlighted the behavior of the analyzed algorithms particularly regarding the variance of the performance obtained for different data distributions. Regarding the execution time, our solution obtains results comparable to the other competitors. However, our solution is capable of working in-place (implemented without any auxiliary array in addition to the one required for the input) and it is able to better exploit the memory interface bandwidth available on GPUs as well as the computing power of their parallel cores. These reasons make the proposed solution viable for sorting large amounts of data.

Accordingly the results of the experiments, we have chosen Bitonic Sorting Network and Radixsort to develop an indexer prototype to evaluate the possibility of using an hybrid CPU-GPU indexer in the real world. The time results obtained by indexing tests are promising and suggest to move also other indexing-phases onto manycore architectures.

3.3 Case Study: Parallel Prefix Sum

In this section we focus on the relevance of prefix sum operation[5] as building block to effectively exploit the parallelism. Given a list of elements, after scan execution each position in the resulting list contains the sum of the items in the operand list up to its index. Blelloch [90] defines all-prefix-sums operation as follows:

Definition 4. *Let $\oplus$ be a binary associative operator with identity $\mathcal{I}$. Given an ordered set of n elements:*

$$[\ a_0,\ a_1,\ \ldots,\ a_{n-1}\ ]$$

the exclusive scan operation returns the ordered set:

$$[\ \mathcal{I},\ a_0,\ (a_0 \oplus a_1),\ \ldots,\ (a_0 \oplus \ldots \oplus a_{n-2})\ ]$$

[5] also known as scan, prefix reduction, or partial sum.

In literature, many application fields exploit scan to merge the partial results obtained by different computations. In particular, general purpose GPU algorithms largely apply this operation as atomic step in many solutions. For example, some of the fastest sorting algorithms (e.g. Radixsort [72], Quicksort [67], and others) perform scan operation many times for reordering the values belonging to different blocks. Ding*et al.* [61] use prefix sum to implement their PForDelta decoder for accessing postings lists, and a number of other applications refer to this operation [91, 92].

Concerning the computational complexity on a sequential processor, prefix sum operation is linear in the size of the input list. Instead when the computation is performed in parallel the complexity becomes logarithmic [93, 94, 95]. On the one hand, the sequential solution is more easy to implement and works in-place. As a consequence, if the available degree of parallelism is low, the approaches based on the sequential solution could be still applied without major performance loss. On the other hand, it is straightforward that, augmenting the parallelism degree, the efficient prefix sum computation becomes a key aspect of the overall performance. This is the case of Graphics Processing Units (GPUs), Sony's Cell BE processor, and next-generation CPUs. They consist of an array of single-instruction multiple-data (SIMD) processors and allow to execute a huge number of arithmetic operations at the same time. Related to SIMD, there is the stream processing (see Section 2.1.2). It is a programming paradigm that allows some applications to easy use multiple computational units by restricting the parallel synchronization or communication that can be performed among those units. Given a set of data (*stream*), a series of operations (*kernel*) is computed independently on each stream element. Furthermore, also due to the performance constraints related to the complex memory hierarchy, the design of efficient algorithms for these processors requires more efforts than for standard CPUs.

An important contribution of this section is to provide a comparison between the main existing solutions for computing prefix sum and our original access pattern. In particular, for each solution, we provide both theoretical and experimental analysis. We describe and justify the main existing algorithms within K-model (see Section 3.1), which is a computational model defined for the novel massively parallel computing architectures such as GPUs and alike processors. The theoretical results point out that our approach efficiently satisfies all the performance constraints featured by K-model, and show the benefits and the limits of each considered solution. Finally, the experimental results validate the theoretical ones by showing that our access schema is up to 25% faster than the other considered solutions.

This section is organized as follows: Section 3.3.1 analyses the solution can be considered suitable for GPUs or computational models exposing performance constraints that are similar to the main ones considered by K-model (e.g. QRQW-model [44]). In Section 3.3.2 we make some general consideration on the tree-based approaches to define a metric for their eval-

uation. In Section 3.3.2 our access schema is defined. Section 3.3.3 shows both theoretical and experimental analysis. Section 3.3.4 describes an effective encoding schema that achieve impressive decoding performance by exploiting GPUs.

3.3.1 Related Work

Due to its relevance, scan has been deeply studied from researchers to devise faster and more efficient solutions. However, only few of these are relevant for our purpose, namely the studies conducted for general purpose computation on GPUs. Furthermore, we attempted to include in our study also those solutions proposed for QRQW PRAM [44] because the behavior of its memory equals the IM one, which is the performance constraint mainly taken into account in this study. Unfortunately, to the best of our knowledge, none has been found.

Horn*et al.* [93] propose a solution derived by the algorithm described in [95]. This solution, represented in Algorithm 5, performs $O(n \log n)$ operations. Since a sequential scan performs $O(n)$ sums, the $\log n$ extra-factor could have a significant effect on the performance hence it is considered work-inefficient. Note that the algorithm in [95] is designed for the Connection Machine architecture [96, 69] that is different from the ours, with an execution time dominated by the number of performed steps rather than work complexity. From this point of view, that consists in considering the number of loops performed at line 1 of Algorithm 5, the solution has a step-complexity equal to $O(\log n)$.

Algorithm 5 Work-inefficient scan.

Require: n-size array $A[\,]$
1: **for** $d \leftarrow 1$ **to** $\log n$ **do**
2: **for all** k **in parallel do**
3: **if** $2^d \leq k < n$ **then**
4: $A[k] \leftarrow A[k] + A[k - 2^{d-1}]$
5: **end if**
6: **end for**
7: **end for**

Harris*et al.* [94] proposed a work-efficient scan algorithm that avoids the extra factor of $\log n$. Their idea consists in two visits of the balanced binary tree built on the input array: the first one is from the leaves to the root, the second starts at the root and ends at the leaves, see Figure 3.13.

Since binary trees with n leaves have $\log n$ levels and 2^d nodes at the generic level $d \in [0, \log n - 1]$, the overall number of operations to perform is $O(n)$. In particular, in the first phase the algorithm traverses the tree from leaves to the root computing the partial results at internal nodes, see lines 1÷6 of Algorithm 6. After that, it traverses the tree starting at the

root to build the final results in place by exploiting the previous partial sums, see lines 8÷15 of Algorithm 6.

Algorithm 6 Work-efficient scan.

Require: n-size array $A[\,]$
1: **for** $d \leftarrow 0$ **to** $\log_2(n) - 1$ **do**
2: **for** $k \leftarrow 0$ **to** $n - 1$ **by** 2^{d+1} **in parallel do**
3: $i \leftarrow k + 2^d - 1$
4: $A[i + 2^d] \leftarrow A[i + 2^d] + A[i]$
5: **end for**
6: **end for**
7: $A[n - 1] \leftarrow 0$
8: **for** $d \leftarrow \log_2(n) - 1$ **down to** 0 **do**
9: **for** $k \leftarrow 0$ **to** $n - 1$ **by** 2^{d+1} **in parallel do**
10: $i \leftarrow k + 2^d - 1$
11: $temp \leftarrow A[i]$
12: $A[i] \leftarrow A[i + 2^d]$
13: $A[i + 2^d] \leftarrow A[i + 2^d] + temp$
14: **end for**
15: **end for**

Despite the work-efficiency, Algorithm 6 is not yet fully efficient considering the metrics defined in Section 3.1, i.e. the function $T()$. Due to the data access pattern used, the solution is inclined to increase the latency for accessing to the shared data-memory IM. In fact, considering a generic step, different scalar processors require to access elements belonging to the same memory bank. As a consequence, the most part of the IM accesses are serialized so as to augment the time required to perform an instruction.

To avoid most memory bank conflicts, Harris *et al.* insert some empty positions in the memory representation of the input list. Specifically, their solution stores an input list element a_i at the array $A[\,]$ position as follows:

$$a_i \mapsto A[\, i + \lfloor i/k \rfloor \,] \tag{3.1}$$

Therefore the authors avoid the "periodicity" with which the same subset of memory banks are accessed in Algorithm 6. On the one hand, they balance the memory banks workload by applying this technique. On the other hand, the memory space required for representing the input in memory grows of an extra-factor that equals n/k. Let us consider, as proof, that an n-size list $\{a_0, \ldots, a_{n-1}\}$ is represented by the array A as follows: $a_0 \mapsto A[0]$ and $a_{n-1} \mapsto A[\, n-1+\lfloor \frac{n-1}{k} \rfloor \,]$. So, the required space is $n+\lfloor \frac{n-1}{k} \rfloor = O(n+\frac{n}{k})$, which corresponds to an extra factor of $O(n/k)$. Last but not the least, this solution does not scale with the size of the input list because, as shown in Section 3.3.2, this technique is effective on lists having up to k^2 elements.

3.3.2 K-Model Oriented Solution

The previous section emphasizes the weak points of Algorithm 6. In particular, such algorithm consists in traversing twice the binary tree built on the input list. Before introducing our pattern, let us point out two aspects that permit to define a "metric" for the analysis of the different solutions:

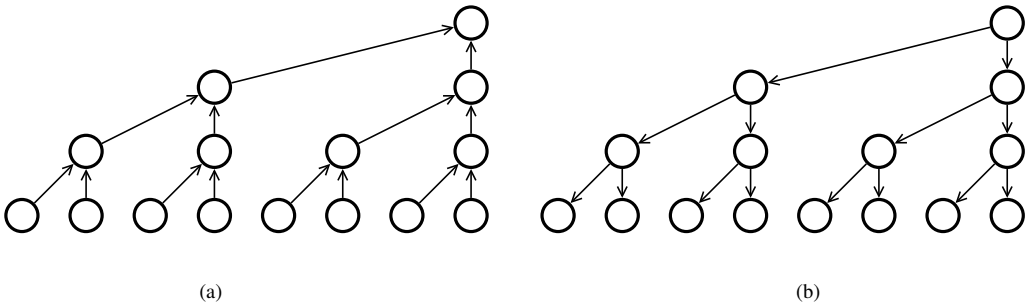


Figure 3.13: Types of tree-traversal performed by Algorithm 6: (a) bottom-up and (b) top-down.

(i) Let us consider the sets of accesses made at each phase of Algorithm 6: the first phase consists of a bottom-up tree-traversal, see Figure 3.13a, the second phase consists in the same procedure performed in a top-down manner, see Figure 3.13b. The two sets of accesses are equivalent. The only difference is the reversed order adopted to traverse the tree. So, by defining an optimized access pattern for the first visit, we have a method to efficiently perform the second visit too. In other words, if a pattern generates a conflict in the shared memory by addressing a pair of elements in the first visit, the same conflict will occur, for the same pair, in the second visit too. As a consequence, we can reduce the analysis of an access pattern to one phase: in our case we chose the bottom-up tree-traversal.

(ii) The second point concerns the operations performed on the shared data-memory, i.e. load and store. By induction on the tree levels: the nodes storing the results of the operations performed at the level i correspond to the operands to add at the level $i+1$. Thus, by reducing the number of conflicts generated to store the results at a generic level of the tree, we automatically reduce the conflicts generated to read the operands at the next level of the visit. This second aspect permits us to focus our analysis on only one type of operation (i.e. store) instead of considering both the store and the load ones.

Resuming, in the rest of the section, we face the problem of defining a data access pattern able to equally spread sequences of k store operations on k different memory banks.

This section describes our access pattern [97]. The main advantages are: (i) our solution is the optimal in space[6], namely $O(1)$; (ii) the proposed schema perfectly balances the workload among the internal memory modules for any given input size; (iii) the solution is based on the tree-based solution and inherits its work efficiency.

In order to reduce the memory contention our solution redirects the sum results so as to refer to distinct memory modules, instead of applying a function like (3.1) to stretch the memory representation of the input list. Our schema works by considering subsets of k consecutive store-operations. As shown in Figure 3.14, given a sequence of k sums to calculate, the algorithm alternatively stores the result of each one of the first $k/2$ sums by replacing the value of the operand with the smallest index (denoted as *left-sum*) and it stores the results of the remaining $k/2$ sums at position of the operand with the greatest index (denoted as *right-sum*). Then, we iteratively apply this pattern to each k-size sequence at all levels of the tree. When the tree-traversal is ending, the number of nodes belonging to a tree level becomes smaller than k and the algorithm performs only left-sums.

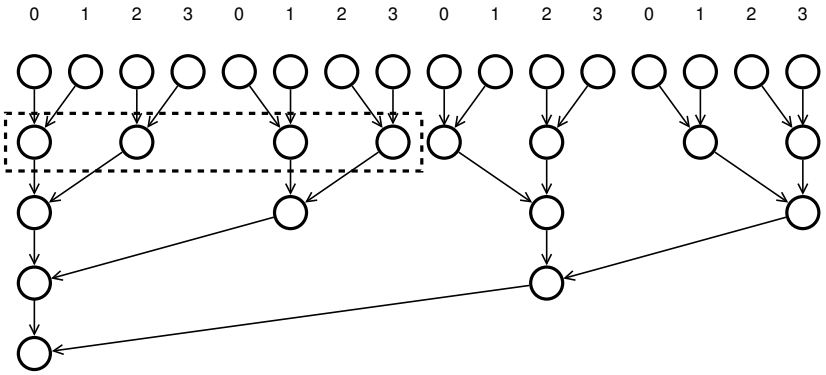


Figure 3.14: Representation of our pattern applied to a tree with $n = 16$ leaves allocated in a memory of $k = 4$ banks.

In this manner, each sequence of k operations addresses k distinct memory banks. From K-model point of view, this means to eliminate the contention on IM. Equation (3.2) defines the approach depicted in Figure 3.14, namely it discriminates between left-sums and right-sums. Basically, this function consists in permuting the set of memory banks related to a sequence of sums to compute.

$$\textsc{LeftRight}_h(x,\ m) = \underbrace{\textsc{Rst}_h(x \cdot 2^m)}_{\textbf{base}} + \underbrace{\textsc{Rot}_h^m(x)}_{\textbf{offset}} \tag{3.2}$$

Let k be a power of 2, given an element at position i, the least significant $\log k$ bits of i identify the memory bank where the element is stored.

[6] This result is still more relevant considering that the real architectures exploits small sized shared data-memories.

Equation (3.2) consists in applying a left circular-shift to the least significant $\log k$ bits of the element positions involved in the computation. In particular, ROT is applied on a sequence of $2 \cdot k$ addresses referring k pairs of operands. If the addresses in the initial sequence are uniformly spread on k distinct memory banks, we obtain a new sequence of k positions belonging to k distinct memory banks that are arranged as Figure 3.14 shows. Since at the first step of the tree traversal the operands are already equally spread among the k banks, no conflicts are generated accessing them. Therefore, applying Equation (3.2) we maintain the desired property, i.e. avoid the conflicts on the memory banks during the execution of Algorithm 6.

In particular, $\text{ROT}_h^m(x)$ stands for computing m times the left circular-shift of the h least significant bits extracted by the binary representation of x. For example, $\text{ROT}_3^1(12) = \text{ROT}_3^1(1100) = \text{ROT}_3^1(100) = 001 = 1$ and $\text{ROT}_3^2(2) = 001 = 1$. The $\text{RST}_h(x)$ operator resets the h least significant bits of the binary representation of x. For example $\text{RST}_3(13) = \text{RST}_3(1101) = 1000 = 8$. Resuming, an index resulting from Equation (3.2) is composed by two distinct parts: the first part, that is referred as **base**, specifies the most significant part of the address; the least significant $\log k$ bits, that are referred as **offset**, are obtained by applying a circular-shift rotation. The resulting pseudo-code is shown in Algorithm 7.

Algorithm 7 Free-conflict bottom-up tree visit step.

```
1: for d ← 0 to log2(n) − 1 do
2:    for i ← 0 to n/2^(d+1) − 1 in parallel do
3:       x ← A[ LeftRight_{log k}(2i, d) ]
4:       y ← A[ LeftRight_{log k}(2i + 1, d) ]
5:       A[ LeftRight_{log k}(i, d + 1) ] = x + y
6:    end for
7: end for
```

This section ends with some notes about the implementation of our access function on the tested GPUs. The SDK [98] provided with the tested devices consists in an extension of the `C` language. However, neither `C` nor `C++` have a statement for circular-shift of a binary word. However, $\text{ROT}_h^m(x)$ can be emulated via:

```
x &= (1<<h)-1;
return (x>>m) | (x<<(h-m));
```

and the operation $\text{RST}_h(x)$ can be emulated via:

```
return x & (0xFFFFFFFF<<h);
```

Moreover, we can quickly perform the $\text{LEFTRIGHT}_h()$ operation by precomputing and storing in a a set of k constants the displacement to add at each index to address the correct item at each step. During the tree-visit, the k processors ($E[\,]$) perform in parallel k sums. To this end, each of them shifts the proper constant proportionally to the current level of the

tree. The complexity to compute the set of constants is proportional to k and not to the input size. Since these constants depend on k they can be computed at compile-time without overhead for the computation.

3.3.3 Evaluation & Experiments

The solution we propose is evaluated theoretically and experimentally by comparing its complexity and performance with those presented in Section 3.3.1 and Section 3.3.2.

Theoretical Evaluation

In this section we evaluate the different solutions by defining the related complexities discussed in Section 3.1, i.e. $G()$, $W()$ and $T()$. Firstly we show the complexities of the three tree-based solutions varying the data access pattern: the naïve version represented by Algorithm 6, the version exploiting padding, and the one based on our novel solution. These solutions are referred in the remainder of this section as *NoPadding*, *Padding*, *LeftRight*, respectively. Concerning $T()$, the analysis of the different solutions is presented using the metric introduced in Section 3.3.2, i.e. taking into account the number of conflicts generated storing the results of the operations performed during the bottom-up tree traversal. Then we analyze the method proposed for the Connection Machine architecture and adapted for GPUs in [93]. This is referred as *CM*.

Concerning $G(n, k)$, that measure the external memory data-transfers efficiency, it equals n/k for all referred methods. In fact, the allocation in the external memory does not need any particular arrangement because it is already optimal. Since the input elements are coalesced in a vector, each memory transaction fully exploits the communication bandwidth by transferring k elements at a time.

Concerning $W(n)$, let us observe that the number of operations performed by the tree-based solutions does not change by varying the access pattern. We already pointed out that these methods differ only in the schema applied for mapping the operands into the internal data-memory IM. Analyzing the lines $1 \div 6$ of Algorithm 6 and Figure 3.13a, it can be seen that the number of nodes to compute halves at each step. Let n denote the input size and, without loss of generality, suppose n is a power of 2. The overall number of operations performed to traverse the tree is given by (3.3) and it corresponds to $W()$ too.

$$W(n) = \sum_{i=1}^{\log n} \frac{n}{2^i} = n - 1 = O(n) \tag{3.3}$$

Thus, let us define the efficiency of the solutions and ℓ denote the number

of instructions issued by IU, we can define ℓ as follows:

$$\ell_{n,k} = \underbrace{\sum_{i=\log k}^{\log n-1} \frac{2^i}{k}}_{\ell_1} + \overbrace{\log k}^{\ell_2} = \frac{n-k}{k} + \log k \tag{3.4}$$

Given a generic tree level $d \geq \log k$, the number of instructions issued by the unit IU is $\ell_1 = 2^d/k$, with 2^d is the number of nodes composing the level. At the last $\log k$ steps of the tree-visit, the efficiency halves as well as the number of sums to perform, nevertheless IU issues one instruction per step, denoted by ℓ_2.

With both $W(n,k)$ and ℓ, we can evaluate the efficiency of the solutions as $W(n,k)$ divided by $k \cdot \ell$, see Equation (3.5). Without loss of generality, we suppose the parameter k, inherent K-model, is a power of 2 and $k < n$.

$$\frac{W(n,k)}{k \cdot \ell_{n,k}} = \frac{n-1}{k \cdot \ell_{n,k}} = \frac{n-1}{k \cdot \log k + n - k} \tag{3.5}$$

Figure 3.15 represents (3.5) by showing the behaviour of the efficiency as function of $W(n)$ by varying the input size. Smaller input size leads to low efficiency, while smaller values of k lead to faster growth of the efficiency curve.

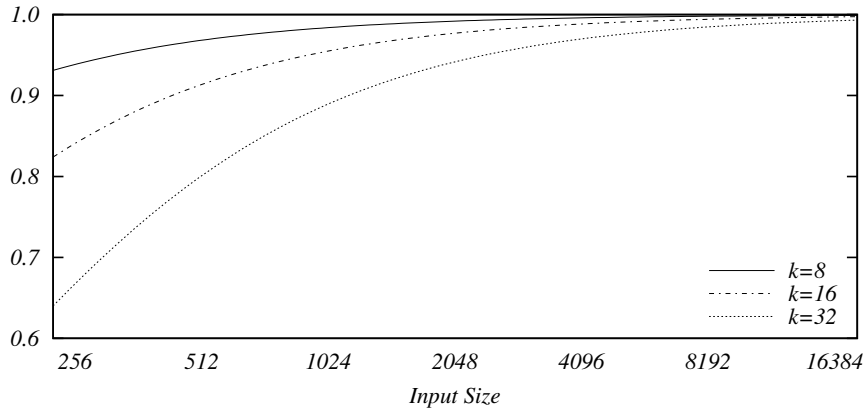


Figure 3.15: Efficiency computed for the tree-based methods with different values of k by varying the input size.

Concerning the CM method, as argued in Section 3.3.1, the number of operations performed by Algorithm 5 is $O(n \log n)$ for any given n-length input list. Therefore $W()$ equals $O(n \log n)$.

As last indicator, we consider $T()$ that denotes the time required to compute a stream element. It depends on the different number of conflicts generated by addressing the internal memory IM so we analyze one solution at a time.

NoPadding. Let n be the input size and $d \in [1,\ \log n]$ indicate a step of the loop represented at the lines $1 \div 6$ of Algorithm 6. The number of operations to compute at each step is $n/2^d$, and the displacement between the positions storing two consecutive results is 2^d. As a consequence, the involved queues $Q[\,]$s are those having index $2^d \mod k$. At each step, the number of operations to perform halves, the queues size doubles and hence the time to perform a step is unvaried because the workload halves as the number of the working memory modules. Since k requests are processed at a time, the time needed to perform the first $\log k$ steps is n/k for each one of them. Only on the last $\log n - \log k$ steps, the maximum queue size does not double because the number of adds becomes too small. Finally $T_{\bar{p}}(n,\ k)$ is:

$$\begin{aligned} T_{\bar{p}}(n,\ k) &= \sum_{d=1}^{\log k} \frac{n}{2^d} \cdot 2^d \cdot \frac{1}{k} + \sum_{d=\log k+1}^{\log n} 2^d \cdot \frac{1}{k} \\ &= \log k \cdot \frac{n}{k} + \sum_{d=1}^{\log n - \log k} 2^d \\ &= n \cdot \frac{\log k + 1}{k} - 1 \end{aligned}$$

Padding. This method is fully efficient until the displacement applied to the indexes is smaller than the number of memory modules, i.e. k in K-model. In particular, for values greater than k, the Function (3.1) generates the same conflicts in IM as the *NoPadding* algorithm does. In particular, let us consider a tree traversal performed on arrays made up of more than $2 \cdot k$ elements. Due to *Padding*, the representation in memory of two consecutive segments of k elements is divided by an empty entry. At the end of the traversal performed on such segments, namely after $\log k$ steps, this technique is no more effective. As a consequence, the computation of the remaining levels generates the same contention-degree of the *NoPadding* method.

Hence, we get the peak of IM bandwidth computing the first $\log k$ steps because all operations are equally spread on the memory banks, see (3.6). Note that such formula is straightforwardly derived by Equation (3.3) because, when the contention is null, the latency of the overall accesses corresponds to $W()/k$. For the remaining steps the complexity function is the same obtained in the *NoPadding* case, i.e. it is $T_{\bar{p}}()$.

$$f_p(n,\ k) = \frac{1}{k} \cdot \sum_{i=1}^{\log k} \frac{n}{2^i} \tag{3.6}$$

The complexity of *Padding* corresponds to the following expression, which is obtained by merging $f_p()$ and $T_{\bar{p}}$.

$$T_p(n,\ k) = f_p(n,\ k) + T_{\bar{p}}\Big(\frac{n}{k},\ k\Big)$$

LeftRight. As argued in Section 3.3.2, this memory access pattern was developed in order to avoid all memory bank conflicts so as to expose the complexity of $T()$ as follows:

$$\begin{aligned} T_{lr}(n,\, k) &= \log k \;+\; \sum_{i=\log k}^{\log n-1} \frac{2^i}{k} \\ &= \log k \;+\; \frac{n-k}{k} \end{aligned}$$

This result corresponds to Equation (3.4) that represents the number of instructions issued by IU. This is the result we expected. In fact, when the set of accesses corresponding to an instruction does not generate any memory conflict then the parallel complexity computed in K-model is asymptotically equal to the number of instruction issued, namely ℓ.

CM. The solution performs $\log n$ steps for a given n-size array. Let $d \in [0,\ \log n - 1]$ denote a step of Algorithm 5, the method performs $n - 2^d$ operations per step. Resuming, the overall amount of sums corresponding to $W()$ is:

$$W_{cm}(n) \;=\; \sum_{d=0}^{\log n-1} n - 2^d \;=\; n \cdot \log n - n + 1$$

In K-model, such solution splits the set of operations performed at each step into session of k operations. Each session can be implemented so as to avoid all conflicts on the memory modules. In practice, we temporarily store the sums of the operands loaded from IM into a local variable. When all sums have been computed, we proceed by writing back the results into IM. As a consequence of that:

$$T_{cm}(n,\, k) \;=\; \frac{W_{cm}(n)}{k}$$

Figure 3.16 is the log-log plot of the $T()$ functions computed for all the solutions taken into account. The parameter k (referred in K-model) has been set equal to 16 that corresponds to the value of the hardware used for the tests so as to make possible to compare theoretical and experimental results. The figure points out that *CM* is the lowest performing solution. Furthermore, until *Padding* is able to perform conflict-free accesses, its performance is comparable to one of *LeftRight* solution. However, the padding technique does not scale well to larger problem sizes. In particular it is effective only when the first $\log k$ steps and the last $\log k$ steps are performed. This means that trees made of at most $\log k + \log k$ levels, namely k^2 items, can be efficiently performed with *Padding*. On the contrary, for input size

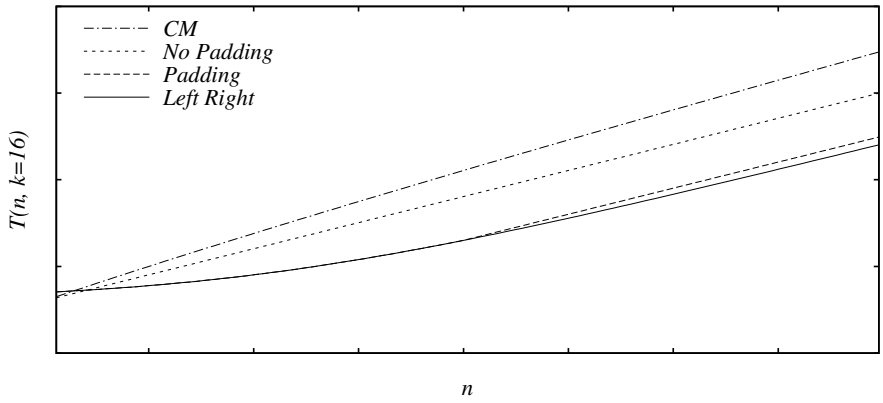


Figure 3.16: Expected behavior of the different solutions in log-log plot. k equals 16 because this value is realistic, i.e. it is related to the hardware used for testing.

greater than k^2 we expect that *Padding* experimental results are wasted due to the serialization of the accesses.

Finally, the following table summarizes the space complexity of each considered algorithm:

Pattern Name	Space Complexity
CM	$O(1)$
NoPadding	$O(1)$
Padding	$O(n/k)$
LeftRight	$O(1)$

In particular, three of them perform scan in linear space and only *Padding* requires an extra factor for the reasons discussed in Section 3.3.1.

Experiental Evaluation

The experiments have been conducted using the CUDA SDK 3.1 on NVIDIA 275 GTX video device. Due to the limited size of the on-chip shared data-memory and considering the standard 32-bit word size, each stream element can compute *blocks* of at most 2048 elements (the same used also for the experiments in [94]). The conducted experiments aim to measure the time required to perform the operation scan in two different cases for all the solutions:

(i) We tested the different approaches by computing an input list that can be stored in only one stream block, i.e. varying the block size in the range from 2^5 up to the maximum size allowed by the architecture, namely 2048 integers. Moreover, to stable results, each stream consists of 10752 blocks, see Figure 3.17. The goal of this test is to compare all the approaches exclusively on a SIMD processor, i.e. how scan is used in [61, 91].

(*ii*) In order to compare the different algorithms, we extend them for computation of large arrays of arbitrary dimensions (e.g. large arrays are scanned in [72, 67], etc.), see Figure 3.18. The basic idea to perform on large lists is simple: we divide the large array into blocks that each can be scanned within a single stream element, scan the blocks, and write the total sum of each block to another array of block-sums. Then we scan the block-sums, generating an array of partial results that are used to "update" all the elements in their respective blocks. In more detail, let n be the number of elements in the input list $in[\,]$, and b be the number of elements processed in a block. We allocate n/b blocks of b elements each. We use the first part of the scan algorithm (i.e. lines 1÷6 of Algorithm 6) to compute each block j independently, storing the resulting scans to sequential locations of an output array $out[\,]$. We make one minor modification to the scan algorithm. Before reset the last element of block j (i.e. line 7 of Algorithm 6), we store the value (the total sum of block j) into an auxiliary array $partial[\,]$. Then we scan $partial[\,]$ in the same manner. After we use $partial[j]$ to initialize the computation of block j using a kernel performing the second part of the scan (i.e. lines 8÷15 of Algorithm 6) on n/b blocks.

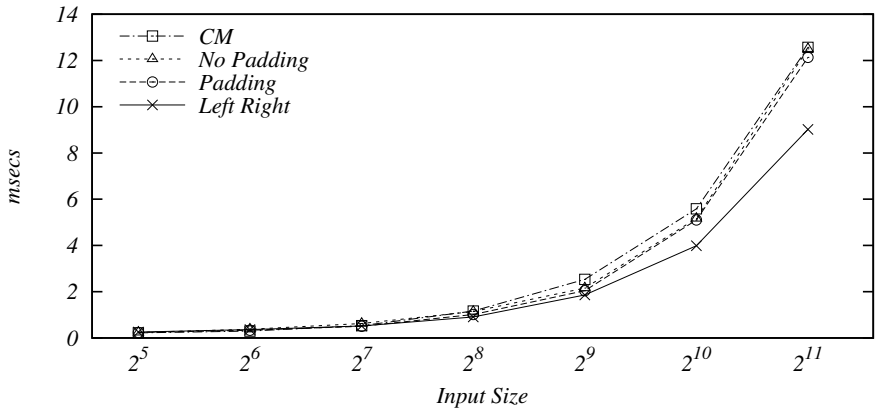


Figure 3.17: Elapsed time for scan on stream blocks of varying size.

The results shown in Figure 3.17 are aligned with the considerations on the theoretical results obtained in the previous paragraph: our solution has been proved to be faster than the competitors. In particular, on input of 2048 items, our solution is 25% faster than *Padding* which was considered the fastest solution for GPUs. This is mainly due to our novel schema that is able to fully exploit the parallelism of the architecture that means, from K-model point of view, to reduce $T(\,)$ with respect to the other methods.

In Figure 3.18, that concerns the computation of large arrays, the benefits of applying our solution are decreased but still permit to outperform the competitors, i.e. we are 18% faster. The main reason is the increased

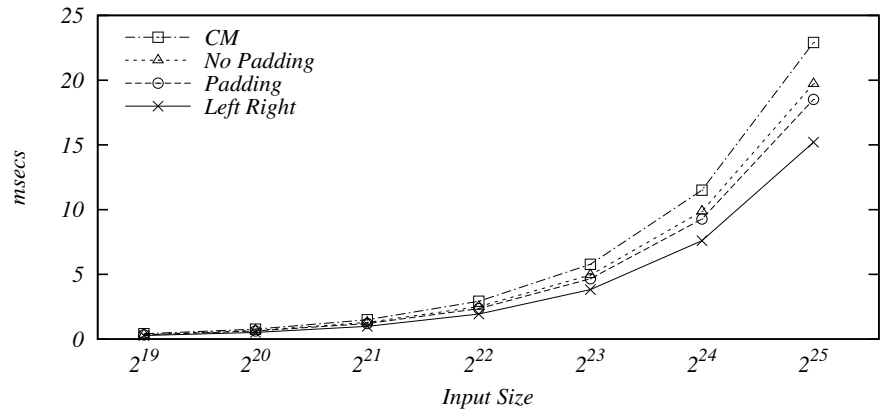


Figure 3.18: Elapsed time for exclusive scan on large arrays of varying size.

number of communications between the array processors and the off-chip memory that dominates the computational time.

3.3.4 Posting Lists Compression

In this section we discuss about the application of GPUs to the design of an efficient posting list decoder. In particular we focus on two methods shown in Section 2.3.5, which are: Vector of Split Encoding [65], hereinafter VS, and the largely used PForDelta [66], hereinafter P4D. The main issue to design an efficient GPU-based decoder is to define an efficient method for retrieving as many as possible postings at the same time. In fact, the target architecture is massively parallel and permits to execute hundreds of instructions on thousands of data at a time. Moreover, to provide an adequate supply of input data and to quickly flush the decoded postings, the GPU is connected to an off-chip memory by a communication channel able to reach a bandwidth greater than the one typical of the standard CPUs. Both these features, namely massively parallelism and communication channel bandwidth, are responsible for the throughput reachable by using GPUs for posting lists decoding. The desirable theoretical peak of performance reachable is often an hard task due to the performance constraints that are proper of this kind of architecture, see Section 2.1. In particular, we focus in this section to minimize the latency of the computation consisting of the posting lists decoding. From K-model point of view, we are interested to reduce both $T()$, denoting the time required to perform the kernel on each SIMD processor, and $G()$, denoting the time needed to fetch encoded postings and to flush the decoded ones.

To optimize the decoder in order to reduce $G()$ does not require particular efforts. This is mainly due to the nature of the data structure involved in the computation. In fact, blocks of both encoded and decoded data are stored in sequences of consecutive addresses allocated in the external memory EM. In order to take advantage of the data arrangement, the decoder

has to be designed in such a way that the SIMD processors can compute each stream element by fetching, manipulating and flushing subsequences of consecutive postings. In this way, on each SIMD processor, the decoding kernel function can handle a stream element by accessing to segments of consecutive addresses, that means by issuing the minimum number of memory transactions. Furthermore this choice is particularly indicated also to compute the original set of doc-id values starting from a list of d-gaps. More precisely, given an ordered list of distinct doc-ids $[\ d_0,\ d_1,\ d_2,\ \dots\ ]$, it is transformed into the equivalent list of d-gaps $[\ d_0,\ d_1-d_0-1,\ d_2-d_1-1,\ \dots\ ]$ before to be encoded. This transformation provides smaller positive integers to encode ensuring a better compression ratio. As a consequence, the rationale is decoding a block made up of consecutive postings on the same SIMD processor. This organization represents the first choice concerning the decoder architecture to solve the problem of how to organize the decoding phase relating to the the stream processing and fully exploiting the bandwidth of the external memory.

We consider, now, the second key aspect concerning the work performed internally by SIMD processors whose latency is denoted by $T()$ in K-model. Given a block of consecutive codewords belonging to the same stream element, we choose to distribute them among the scalar processors module the number of scalar processors composing the SIMD processor. Let us consider VS encoding. According to the VS coding technique, to each bunch of codewords (b) is associated a signature denoting both the number of encoded codewords (n_b) and their size (s_b) in term of number of bits. Given a sequence of these bunches and the related sequence of codewords, the decoding phase has to extract a block of integers from the sequence of codewords according to the signature. To accomplish this task in parallel, two approaches can be adopted:

(i) The first one consists in to assign one bunch of codewords (b) to each processor. According to this approach, each processor only needs to know: (i) the number of bits preceding the first codeword of the assigned bunch and (ii) the number of codewords encoded in the previous bunches. For example, let us suppose to use a set of k scalar processors $\{E[i]\}_{\forall i \in [0,\, k-1]}$ for decoding a set of k bunches of codewords, i.e. $\{b_i\}_{\forall i}$, each one with a signature (n_i, s_i). This solution assigns b_i to $E[i]$ so, for each i, the processor $E[i]$ accesses the block of codewords at position

$$h_i = \sum_{j=0}^{j<i} n_j \cdot b_j \tag{3.7}$$

to extract the first s_i-size codeword. Then it extracts the next s_i bits for the second codeword, and so on until the last n_i-th codeword of b_i.

Furthermore, the codewords extracted will be placed in a subset on consecutive positions in the resulting array starting from the position g_i defined as follows:

$$g_i = \sum_{j=0}^{j<i} n_j \tag{3.8}$$

Considering that the previous sums (h_i and g_i) can be efficiently computed in $O(\log k)$ steps [90], the solution is promising, but it does not optimally balance the workload among the scalar processors. Indeed, the whole process requires each scalar processor spends a time proportional to the largest bunch, i.e. $\max\{n_i\}_{\forall i \in [0,\, k-1]}$.

(*ii*) The alternative solution (the adopted one) consists in to preventively calculate the position of each codeword then to assign each codeword to a scalar processor module the number of processors, i.e. k. As first we compute both the sums h_i and g_i (as defined above) then to each processor $E[i]$ is assigned the processing of codewords at the positions corresponding to $h_i + (i + t \cdot k) \cdot s_i$ for each value of $t \in \mathbb{N}^+$ such that $0 \leq i + t \cdot k < n_i$. In this way we can optimally distribute the workload among the processors smoothing in the different bunches size.

Moreover, let us point out that the same conclusion are achieved also adopting P4D method too. In fact, the extraction of the codewords(exceptions are managed separately) can be processed as in the VS case. The main difference is that, in P4D case, the number of codewords belonging to each bunch is constant. As a consequence, the parameter that defines the size (in number of codewords) of a block encoded with P4D can be tuned as multiple of k so as to reach a perfect workload, that is assigning to all processors the same number of codewords to retrieve.

Designing an Efficient Postings Decoder

Now we focus on the computation of h_i and g_i as defined in the equations (3.7) and (3.8), respectively, for any given n and $i \in [0, n-1]$. After the computation of $[\ g_0,\ \ldots\ g_{n-1}\ ]$ and $[\ h_0,\ \ldots\ h_{n-1}\ ]$, each position in the resulting lists contain the sum of the items in the operand list up to its index. This type of operation consists in a prefix-sum as shown in Definition 4 and in the following: *let $\oplus$ be a binary associative operator with identity $\mathcal{I}$. Given an ordered set of n elements:* $[\ a_0,\ a_1,\ \ldots,\ a_{n-1}\ ]$ *the exclusive scan operation returns* $[\ \mathcal{I},\ a_0,\ (a_0 \oplus a_1),\ \ldots,\ (a_0 \oplus \ldots \oplus a_{n-2})\ ]$.

Depending on the computing architecture taken into consideration, different efficient solutions can be found in literature. Concerning computation on GPUs in general, we propose in Section 3.3.2 a method that balances

the workload among the parallel memory modules and that is able to scale with any given input size. Resuming, the solution is based on the tree-based solution and inherits its work efficiency, i.e. the cost of performing two tree-traversal of a binary tree built on the n-size input list. As a consequence the method performs twice a procedure costing $O(n)$ operations distributed on $2 \cdot \log n$ steps.

However, as argued in [96] regarding to Connection Machine [69] and in Section 3.1 regarding to architectures alike K-model, the time required for the computation depends on the number of instructions issued (and, from K-model point of view, it depends also on their latency) rather than on the number of scalar operations performed. We taken into account the method proposed in [96] that needs $\log n$ steps to be performed instead of $2 \cdot \log n$ steps. The algorithm shown in Algorithm 8 assumes that there are as many processors as data elements. On GPUs this is not usually the case, because the **For Each** instruction in Algorithm 8 is automatically divided into small parallel batches that are executed sequentially on a SIMD processor in stream programming fashion. Since Algorithm 8 performs the scan in place on the array, it does not work because the results of a step are overwritten by the accesses of another step.

Algorithm 8 PPSUM$_k$($A[\,]$)

```
1. For s ← 0 To log k − 1 Step s = s + 1 Do
2.   For Each j ∈ [0, k − 1] In Parallel Do
3.     If j + 2^s < k Then
4.         A[j + 2^s] ← A[j + 2^s] + A[j];
5.     End If
6.   End Do
7. End Do
```

To avoid such limitation, in our case we can "resize" the problem so as to have a solution that works in place and performs the minimum number of steps as Algorithm 8 does. Since in K-model each instruction is provided simultaneously to the k scalar processors $E[\,]$, we can state the granularity of the computation equal to k, which means k corresponds to the number of pairs (n_b, s_b) belonging to the bunch b that are performed in parallel with one scan. Note that, there is no difference between the tree-based solution proposed in Section 3.3.2 and the previous Algorithm 8 in term of asymptotic analysis: both of them are logarithmic. However, because in our case the size of the input list is fixed and equals to k, to adopt Algorithm 8 means to halve the time spent by the scan w.r.t. use a tree-based approach. Because prefix-sum is the complexest procedure of the decoder, the coefficients have a relevant effect on the expression's value for our purpose.

After prefix-sum computation, the procedure EXTRACTBUNCH, which is described in Algorithm 9, provides to extract the encoded codewords

related to a generic bunch b_i composing a block B of k bunches, i.e. $B = [\, b_0,\ \ldots,\ b_{k-1}\,]$. Given a bunch b_i of codewords and its signature n_i and s_i, for each codeword belonging to b_i, a scalar processor $E[j]$, with $j \in [0,\ k-1]$, is aware of the following informations: (i) where the representation of the codeword starts within the codewords bit sequence, i.e. $h_i + c \cdot s_i$; (ii) the length of the codeword to extract, i.e. s_i; (iii) the position of the extracted codeword inside the resulting list, i.e. $g_i + c$.

Algorithm 9 EXTRACTBUNCH(n_i, s_i, h_i, g_i)

$in[\,]$ bit-stream of encoded data.
$out[\,]$ list of decoded integers.
1. **For Each** $j \in [0,\ k-1]$ **In Parallel Do**
2. **For** $c \leftarrow j$ **To** $n_i - 1$ **Step** $c = c + k$ **Do**
3. $begin \leftarrow h_i + c \cdot s_i$;
4. $end \leftarrow begin + s_i$;
5. $out[g_i + c] =$ BITCPY($in[\,], begin, end$);
6. **End Do**
7. **End Do**

In Algorithm 9, the values that c assumes denote the sequence of integers congruent $j \mod k$ for each processing unit $E[j]$; moreover BITCPY($A[\,], b, c$) denotes a macro returning the subsequence of digits in the $A[\,]$ bit list from the position b to c .

Finally, Algorithm 10 resumes the phases for decoding a block of codewords. It represents the atomic procedure we iteratively use for extracting the values encoded in a compressed postings list. As shown in Algorithm 9, the procedure EXTRACTBUNCH is executed k times: once for each bunch of B. As argued above, the parameter k has been chosen as size of B because it is the least integer permitting us to express the most fine-grained computation by maintaining the full efficiency.

Algorithm 10 EXTRACTBLOCK(B)

1. $g[\,] \leftarrow$ PPSUM$_k$($[n_0,\ \ldots,\ n_{k-1}]$);
2. $h[\,] \leftarrow$ PPSUM$_k$($[s_0 \cdot n_0,\ \ldots,\ s_{k-1} \cdot n_{k-1}]$);
3. **For Each** $i \in [0,\ k-1]$ **Do**
4. EXTRACTBUNCH($n_i,\ s_i,\ h_i,\ g_i$);
5. **End Do**

Extension for P4D. Many aspects discussed in the previous section about VS are also valid for designing the P4D decoder. In other words, P4D differs from VS for the following aspects: (i) the n_i parameter is constant in P4D (we refer n_i as $\bar{n}$ to remark that it does not change). In fact, $\bar{n}$ is constant and multiple of the word size so as to align the number of

necessary memory words to the related data transfer; (*ii*) P4D exploits exceptions: in this way P4D exchanges a better compression ratio with lower throughput. Note that, from this point of view, VS can be considered as exclusively composed of exceptions, that means composed of varying-size blocks of codewords.

Concerning the performance and the design of the decoder, to have a unique for n_i leads to a more efficient implementation of the method. In Algorithm 9, by choosing $\bar{n}$ as multiple of k we can use in each step all the k scalar processors $E[\,]$. Contrarily, in the VS case this behavior is not guaranteed because the inner **For** of Algorithm 9 involves, at the last loop, only those processors with index $\leq (n_i \mod k)$. Moreover, in Algorithm 10, if $\bar{n}$ is constant we do not need to compute $\text{PPSUM}_k(\ [n_{0\ldots k-1}]\ )$ because its computation equals to a multiplication, i.e. $g_i = \bar{n} \cdot i$.

Regarding to the second point, namely the exception management, we defined a variant of the original method that simplifies the implementation, and improves the throughput of our solution with respect to the throughput reachable with original version. The past P4D [66] requires to define a proper set of exceptions for each bunch b. This approach forces to perform a further computation for extracting each set of exceptions related to a bunch b. In other words, this means to execute a procedure that, after the extraction of the "regular" codewords, extracts and merges the exceptions, requires a time comparable to the time spent to perform Algorithm 10. In particular, since the different bunches have their own number of different size exceptions, we have to perform, for each set of exceptions related to a bunch b, a procedure that is similar to Algorithm 9. To avoid this effort, our solution merges all exceptions into one set. In other words, instead of having a set of exceptions for each bunch b, we define a unique set of exceptions for the block B. In this way we can reduce the overhead for accessing the exceptions by aligning the representation of different sets of exceptions. Each exception of the block B is represented by two parts: the position of the codeword related to the exception in the decoded block B and the value of the exception to concatenate with the related codeword, namely the part of codeword exceeding its "regular" representation. Since the number of codewords per bunch is constant (i.e. $\bar{n}$) and the bunches in B are k, the first part of the exception can be stored in $\log\lceil k+\bar{n}\rceil$. Instead, the bit size of the exceptions depends on the largest exception to represent concerning the values in B.

Synthetic Data Tests

The previous sections describe the decoders designed respectively for VS and P4D and point out the similarities and the differences between the two solutions. The goal of the conducted tests is to measure the influence on the decoding throughput of the following aspects: (*i*) the size of the integer list L to decode; to take advantage from the parallelism of the target architecture,

large L are needed to supply the array of processors. (ii) VS decoder suffers of the variable size of a bunch, i.e. n_i. On the one side this aspect permits to VS to reach a better compression ratio, on the other hand, as argued in the previous section, it means to handle a variable decoding schema that is not full-efficiently performed by the proposed solution. Differently, P4D uses a static memory representation that gives better performance in term of throughput. (iii) greater the binary representation of the encoded codewords is, more would be the time required to fetch such data from the memory, i.e. more memory transactions are necessary. Thus, increasing the value b, which denotes the average size of an encoded codeword in memory, the decoding throughput decreases. We expect to find some correspondences of these behaviors in the experiment results.

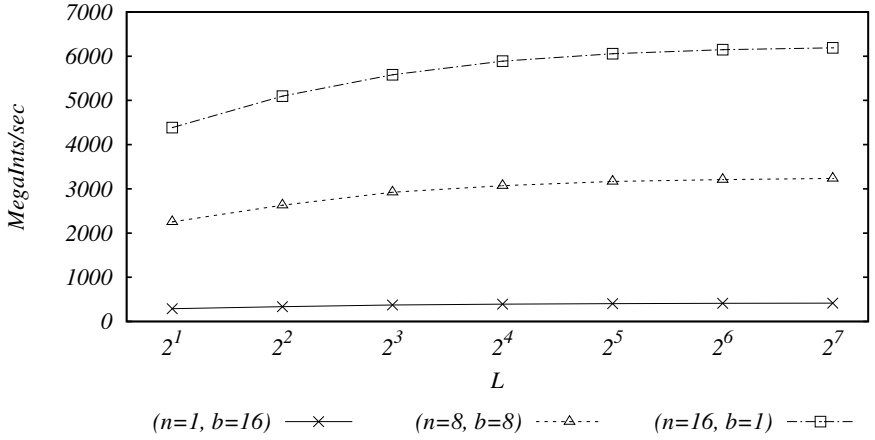


Figure 3.19: Throughput decoding synthetic data by varying the list size (L) and setting n and b with the bound values and the average one. L is divided by 10^5.

VS coding. The first method we analyse is VS. In order to better exhibit the effects of each one of the aspects introduced above, we measure the throughput of the VS by varying, one at a time, the average value of the related parameters, namely L, n, and b. Figure 3.19 shows the results obtained varying the length of the posting lists in a range from $2 \cdot 10^5$ to $2^7 \cdot 10^5$ postings, while n, which denotes the average of the n_i values, and b are set equal to different couple of values. Figure 3.20 and Figure 3.21 show the throughput obtained by varying n and b, respectively. In particular, Figure 3.20 shows interesting results. The test concerns the effect on the throughput of varying the average number of postings contained in the bunches (i.e. n). The related results show that, when n is particularly small, the efficiency of the instructions performed by the loop **For ... Do** in Algorithm 9 decreases as the number of scalar processors involved in the decoding process. The effect is a drastic loss in throughput. Fortunately

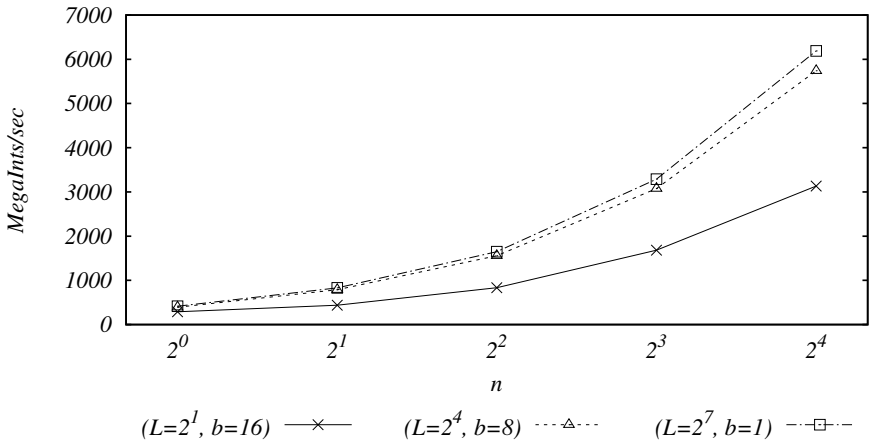


Figure 3.20: Throughput decoding synthetic data by varying the average number of posting per bunch (n) and setting L and b with the bound values and the average one.

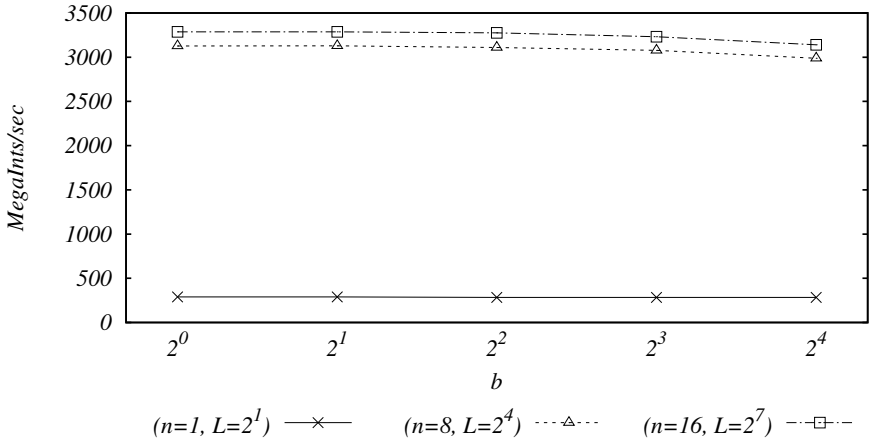


Figure 3.21: Throughput decoding synthetic data by varying the average posting size b and setting n and L with the bound values and the average one.

the probability to have the average number of elements per bunch equal to few units (i.e. 1 or 2) is low. In fact, it means to instantiate a new bunch for each encoded integer, but this implies to obtain a low compression ratio and this is prevented by the VS encoder. In other words, in these cases, we can state that to store the integers in-clear (i.e. one value per memory word) is more convenient.

P4D coding. We provide, now, the results of the tests performed for the P4D solution. Note that, in this case, the parameter n is constant as required from the definition of the method, in particular we defined $n = 32$. Moreover, as a consequence of the solution proposed in the previous sections,

the management of the exceptions can be done in just one pass, namely the latency for their treatment is independent of the number of exceptions defined in the block B to decode.

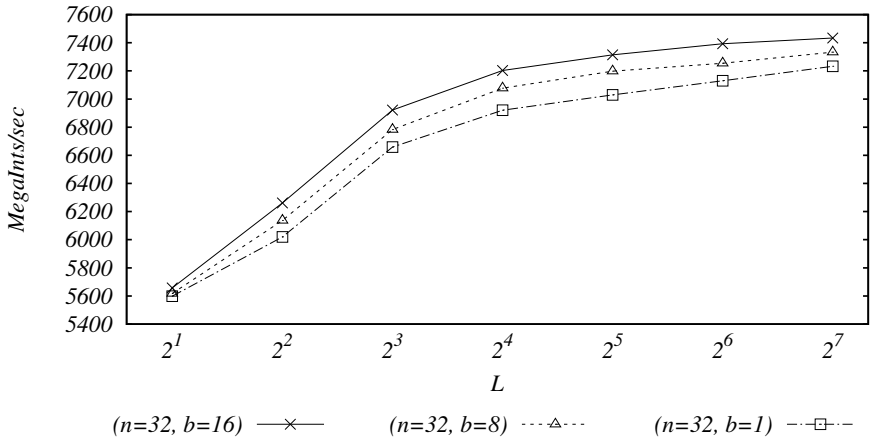


Figure 3.22: Throughput decoding synthetic data by varying the list size L and setting n and b with the bound values and the average one. L is divided by 10^5.

In Figure 3.22 the results are shown. It can be seen that the main parameter affecting the throughput of P4D is L, i.e. the number of encoded integers. When its average value is low, the size of the input is not enough to completely exploit the array of processors within the architecture; as a consequence the overall performance are low. Compared to VS, P4D is able to guarantee an higher throughput. This is mainly due to a more static schema used to encode the integers, thus, as argued in the previous section, also the schema used for the decoding phase is less complex and it ensures an higher throughput.

3.3.5 Conclusion

The scan operation is a simple and powerful parallel primitive function with a broad range of applications. In this section we use K-model as valid tool for the analysis of the efficiency and the complexity of algorithms performing the prefix-sum operation when performed on GPUs. Driven by the definitions of K-model performance indicators, we have defined an access pattern able to minimize the measures inherent to the internal computation performed on such model. Experiments confirmed the theoretical results by reducing the time for computing each stream elements up to the 25% with respect to the other methods existing in literature.

We also described an effective encoding schema that achieve promising performance in decoding by exploiting graphics processing units that are present in today's computers. Experimental evaluation support our claims

and show an impressive throughput of about 6 Giga integers decoded per second.

CHAPTER 4

Final Remarks

This PhD study focuses on evaluating the impact of performing general-purpose computations on *unconventional* processing architectures[1], which were not intended for general-purpose HPC in the first place. The reasons for exploiting unconventional processors in HPC could be raw computing power, good performance per watt, or low cost in general. For example, the computing power of platforms for graphics rendering continuously raises and this motivates the use of these platforms to face the demand of computational resources by WebIR systems. To this end, we proposed a computational model to assess the suitability of the target architectures to various applications as well as the computation complexity of an algorithm. By using our model, we also proposed some novel techniques that improve the performance of different existing algorithms and that prove the need of an adequate model for such unconventional processing architectures.

In the following sections we briefly resume the main steps of this PhD project, the main achieved results, the possible next developments, and the list of publications.

4.1 Conclusions

Driven by the availability of novel affordable massively parallel processors, in this PhD thesis, we aim at *evaluating the impact of, novel, manycore-based architectures on WebIR systems*. For this reason, the last goal was to study, design, and deploy optimal algorithmic solutions to WebIR-related problems that specifically target novel architectures, which are considered, in turn, made up of manycore processors.

[1] e.g. Graphics Processing Units, Sony's Cell/BE processor, etc.

We initially stated how we intend to measure the *impact* of manycore-based systems on WebIR-related problems. The impact of the proposed solutions was measured in term of performance metrics, i.e. the time required for indexing process and the throughput reached decoding a posting list. Then the entire study was decomposed in two different, correlated, research lines.

First we identified some *data-intensive* problems requiring a non-trivial solution to run efficiently on the target architectures. Such solutions require a careful design phase allowing us to find appropriate ways to partition the data, allocate processes, etc. in order to efficiently solve a *data-intensive* problem on architectures that are more targeting *compute-intensive* problems. Second we generalized the lessons learned from the previous activity and then we defined an accurate computational model, called K-model, targeting the systems we are considering. As far as we are aware of, models existing in literature fail to abstract novel processors such as GPUs or the Cell/BE. In particular, the models we analyzed in our state-of-the-art section are either too much general or based on assumptions that incorrectly describe these novel architectures.

A closed-loop feedback process characterized the two research lines. On the one hand, WebIR-related problems provided a test-bed for examining if a technique designed to take into consideration a specific architectural feature is then really effective. In this sense, the conducted tests helped us to refine the abstract model. On the other hand the model was used to design and compare some candidate algorithms in order to develop effective and efficient solutions for the target problems.

In particular, we considered two case studies related to WebIR: indexing and posting list decoding. The studies conducted on the first case led us to investigate on the efficiency of existing approaches for sorting. As a result we defined a new solution capable of fully exploiting the computing power of the available parallel cores as well as the bandwidth of the relative communication channels. This is possible because our solution exploits a novel data-access pattern properly designed to efficiently perform a sorting network in the stream programming model, which is strictly related to manycores. As expected from the analysis of the theoretical evaluation, our solution is capable of performing similarly to the other state-of-the-art solutions on small synthetic datasets. However, our sorting network outperforms the different competitors when it is applied to develop a WebIR indexer, thereby for sorting of larger datasets. This because our approach does not use any auxiliary array in addition to the one required for the input so it is able to sort larger amount of data in a single pass. In the second case study, we attempt to define an efficient approach to perform the parallel prefix sum. As a result we propose a new technique to efficiently perform such operation. With respect to the existing solutions, our technique exploits a novel data access pattern that evenly balances the workload at each step of the computation. Last but not the least, the proposed technique

performs the parallel prefix sum in-place, hence it is more effective on large input data. Successively we developed a posting list decoder that is able to reach peak of throughput (in term of Mega of integers per second) that is one order of magnitude higher than the one obtained on "conventional" processing architectures.

4.2 Future Works

Concerning future works, let us briefly introduce a third possible scenario related to the exploitation of many-core architectures in the WebIR systems. We focus on the possibility of exploiting manycores for diversifying the Web search results[2].

In our previous study [99] we dealt with the adaptation of the pipeline composing the Machine Learned Ranking (MLR) systems to diversifying problem. In practice, a two-phase scoring scheme is usually employed during the online query processing. In the first phase, a simple but inaccurate scoring technique (e.g. BM25) is used for selecting a small subset of potentially relevant documents from the entire collection. In the second phase, the selected documents are scored again by using a complex but more accurate MLR process. The final ranking is determined by the document scores computed in the second phase. In order to perform the second phase on manycores, the pipelined modules should be examined in the K-model point of view, thereby optimizing the performance constraints during the algorithm-design phase. After that, to implement such process on manycores, a MapReduce-based approach could be adopted as follows. Firstly defining an efficient way to perform the functions related to the different pipelined modules on the input documents. Specifically, the document set have to be divided into a stream of elements by taking into account the potential dependencies induced by the existing pipelined functions and the possible early exit techniques [100]. To this end, we learned that a key point is the problem of data-management (in particular the allocation and the representation of the data) in order to maximize the data transfer bandwidth and define an efficient program capable of fully exploiting the parallel computing power. After the computation of the document-score pairs, we have to reduce these pairs in order to return to the user the final list of documents. To this end, the pattern we used in [70] to implement the bitonic merger is a viable solution for implementing this phase.

Furthermore, WebIR includes a number of problems whose solutions can be re-design within the stream processing paradigm, which is related to K-model. This could leads to obtain better performance with respect to the one achievable on conventional architecture. Finally the experience matured by applying K-model in WebIR context can be usefully re-used in other application fields so as to take further advantage of more performing

[2] for more details, see the appendix in Chapther 5

processing units.

4.3 List of Publications

This section lists the publications related to my PhD studies. The first paragraph contains the publications related to the main PhD topic, i.e. the evaluation of the impact of novel computing architectures in WebIR systems. The rest of the section lists the publications related to the results obtained in additional research areas. For each topic, the publications are ordered from the most recent to the oldest one.

The Impact of Novel Computing Architectures in WebIR Systems

1. Gabriele Capannini. "Designing Efficient Parallel Prefix Sum Algorithms for GPUs". *11th IEEE International Conference on Computer and Information Technology (CIT 2011)*, August 2011.

2. Gabriele Capannini, Fabrizio Silvestri, and Ranieri Baraglia. "Sorting on GPUs for large scale datasets: A thorough comparison". *Information Processing & Management*, In Press, Corrected Proof(60), January 2011.

3. Gabriele Capannini, Fabrizio Silvestri, and Ranieri Baraglia. "K-Model: A New Computational Model for Stream Processors". *10th IEEE International Conference on High Performance Computing and Communications*, September 2010.

4. Gabriele Capannini, Fabrizio Silvestri, and Ranieri Baraglia. "Mapping Bitonic Sorting Network onto GPUs". *ComplexHPC Meeting, Universidade Tecnica de Lisboa*, October 2009.

5. Gabriele Capannini, Fabrizio Silvestri, Ranieri Baraglia, and Franco Maria Nardini. "Sorting using BItonic netwoRk wIth CUDA". *7th Workshop on LSDS-IR (SIGIR 2009)*, July 2009.

Job Scheduling

1. Ranieri Baraglia, Gabriele Capannini, Domenico Laforenza, Marco Pasquali, and Laura Ricci. "A Multilevel Scheduler for Batch Jobs on Grids". *The Journal of Supercomputing, 55(11227)*, January 2011.

2. Ranieri Baraglia, Gabriele Capannini, Patrizio Dazzi, and Giancarlo Pagano. "A Multi-criteria Job Scheduling Framework for Large Computing Farms". *Proc. of the 10th IEEE International Conference on Computer and Information Technology*, September 2010.

3. Ranieri Baraglia, Gabriele Capannini, Domenico Laforenza, Marco Pasquali, and Laura Ricci. "A Priority-Based Multilevel Scheduler for Batch Jobs on Grids". *PDPTA. CSREA Press*, July 2009.

4. Baraglia Ranieri, Capannini Gabriele, Laforenza Domenico, Pasquali Marco, and Ricci Laura. "A Two-Level Scheduler to Dynamically Schedule a Stream of Batch Jobs in Large-Scale Grids". *Proceedings of the 17th ACM International Symposium on High Performance Distributed Computing (HPDC '08)*, 2008.

5. Dalibor Klusáček, Hana Rudová, Ranieri Baraglia, Marco Pasquali, and Gabriele Capannini. "Comparison Of Multi-Criteria Scheduling Techniques". *Grid Computing. Springer US*, 2008.

6. Marco Danelutto, Paraskevi Fragopoulou, Vladimir Getov, Ranieri Baraglia, Gabriele Capannini, Marco Pasquali, Diego Puppin, Laura Ricci, and Ariel Techiouba. "Backfilling Strategies for Scheduling Streams of Jobs On Computational Farms". In *Making Grids Work*. Springer US, 2008.

7. Gabriele Capannini, Ranieri Baraglia, Diego Puppin, Laura Ricci, and Marco Pasquali. "A Job Scheduling Framework for Large Computing Farms". In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07. ACM, New York, NY, USA, 2007.

8. Dalibor Klusáček, Hana Rudová, Luděk Matyska, Ranieri Baraglia, and Gabriele Capannini. "Local Search for Grid Scheduling". Doctoral Consortium at the International Conference on Automated Planning and Scheduling (ICAPS'07), Providence, RI, USA, 2007, 2007.

Efficiency in Web Search Results Diversification

1. Gabriele Capannini, Franco Maria Nardini, Raffaele Perego, and Fabrizio Silvestri. "A Search Architecture Enabling Efficient Diversification of Search Results". *DDR-2011 in conjunction with ECIR 2011 – the 33rd European Conference on Information Retrieval*, 18th April 2011.

2. Gabriele Capannini, Franco Maria Nardini, Raffaele Perego, and Fabrizio Silvestri. "Efficient Diversification of Web Search Results". *Proceedings of the VLDB, Vol. 4, No. 7*, April 2011.

3. Gabriele Capannini, Franco Maria Nardini, Raffaele Perego, and Fabrizio Silvestri. "Efficient Diversification of Search Results using Query Logs". In *20th International Conference on World Wide Web (Accepted Poster)*. ACM, Hyderabad, India, March 28 - April 1 2011.

Other Publications

1. Ranieri Baraglia, Gabriele Capannini, Malko Bravi, Antonio Laganà, and Edoardo Zambonini. "A Parallel Code for Time Independent Quantum Reactive Scattering on CPU-GPU Platforms". In *Chemistry and Materials Sciences and Technologies (CMST 2011) in the 11th International Conference on Computational Science and Its Applications (ICCSA 2011)*, Lecture Notes in Computer Science. Springer, University of Cantabria, Santander, Spain, June 20-23 2011.

CHAPTER 5

Appendix

This section shows a more detailed description of the results obtained in the additional research areas investigated during the PhD period. This material is not necessary to the comprehension of the results concerning the main PhD research topic but provides a summary of the results obtained in different areas.

A. Job Scheduling

To build a grid infrastructure requires the development and deployment of middleware, services, and tools. At middleware level the scheduler plays a major role in order to efficiently and effectively schedule submitted jobs on the available resources. The objective of the scheduler is to assign tasks to specific resources maximizing the overall resource utilization and guaranteeing the QoS required by the applications. As depicted in Figure 5.1 we defined a two-level framework for dynamically scheduling a continuous stream of sequential and multi-threaded batch jobs on large-scale grids of interconnected clusters of heterogeneous machines.

At the highest level the **Meta Scheduler** (MS) dispatches jobs to clusters according to a policy for balancing the workload by assigning a job to the less loaded cluster. To estimate the workload on each cluster, an array of max positions is defined for each one. The parameter max identifies the number of possible priorities that can be assigned to submitted jobs. Each priority value corresponds to an array position, which stores the amount of workload due to jobs with the corresponding priority value, plus the amount of load due to jobs with higher priority. According to this estimation, clusters are ranked, and a job is scheduled to the cluster with the smallest rank. The job priority is computed by summing the partial priority values $\Delta_{p,j}$

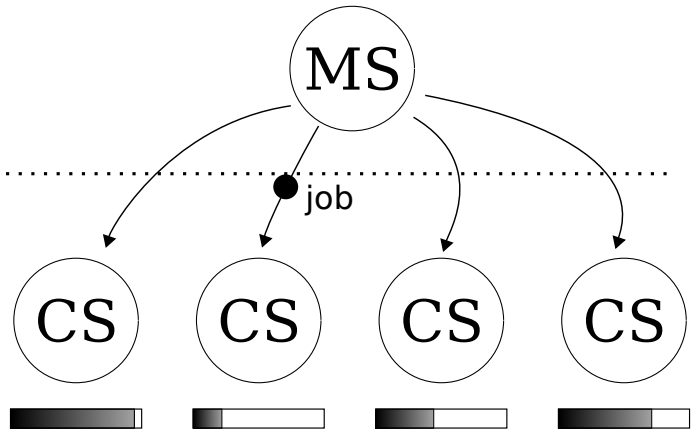


Figure 5.1: Sketch of the two-level scheduling framework.

computed by two heuristics ($H_{deadline}$ and $H_{license}$) each one managing an aspect of the considered problem.

$H_{deadline}$ aims to improve the number of jobs that execute respecting their deadline. Jobs closer to their deadline get a boost in priority as a function of the jobs computational requirements. In particular, the boost $\Delta_{p,j}$ to the priority of a job j is proportional to the proximity of the time at which it has to start its execution to meet its deadline. The boost $\Delta_{p,j}$ is computed as follows. Given a fixed size temporal window the top-level scheduler assigns a job to a class of priority so that only a small subset of the jobs related to the current temporal window has the highest priority. To this end the temporal domain is divided into *max* intervals then each of them has a duration equal to twice the previous interval, which is related to an higher priority class. Depending on the proximity, which is computed as deadline time minus the wall clock time minus the job expected duration, the corresponding job is assigned to a priority class. Finally $\Delta_{p,j}$ is computed as function of its corresponding priority class.

$H_{license}$ computes $\Delta_{p,j}$ to favor the execution of jobs that improve the license usage. $\Delta_{p,j}$ is computed as a function of the number of licenses required by a job. Jobs asking for a higher number of licenses get a boost in preference that gives them an advantage in scheduling. This pushes jobs using many licenses to be scheduled first, this way releasing a number of licenses.

The results concerning the tests conducted [14] show that MS is able to dispatch jobs among underlying clusters, distributing the workload proportionally to the actual cluster computational power, which is the main task of the the highest level scheduling module.

At bottom of the scheduler hierarchy, each cluster is managed by an instance of **Convergent Scheduler** (CS). This framework exploits a job-machine matrix $P^{|N|\times|M|}$ where N denotes the set of jobs to be matched with the machines belonging to the cluster M. Each entry $p_{i,m} \in P$ denotes

the preference degree of a job i on a machine m.

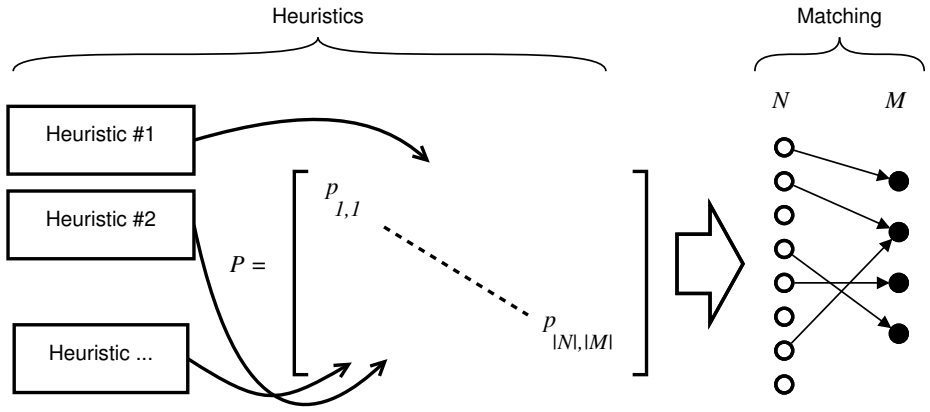


Figure 5.2: Structure of the Convergent Scheduling framework.

As shown in Figure 5.2 the current scheduling framework is structured according to two main phases: Heuristics and Matching. The matrix P constitutes the common interface to the heuristics set. Each heuristics changes priority values in order to compute job-machine matching degrees. The final priority value $p_{i,m}$ is the linear combination of the values computed by each heuristics. Managing the constraints by using distinct heuristics, and structuring the framework in subsequent phases, leads to a modular structure that makes it easier to extend and/or to modify. Moreover, to manage the problem constraints, in [13] we revised and extended the set of heuristics proposed in [12], and exploited by the scheduler for making decisions. The matching phase elaborates the resulting P matrix to carry out a new scheduling plan, which can be updated whenever it is required by the adopted policy (e.g. event-driven policy: variations in N and/or M sets; time-driven policy; a mix of the two previous ones). Each matrix element expresses the preference degree in selecting a machine m to run a job i. The aim of the matching algorithm is to compute the associations job-machine to which correspond a larger preference degree, according to the problem constraints. The elements of P are arranged in descending order, and the new scheduling plan is built by incrementally including the job-machine associations, and updating the available resources. Once an association job-machine takes place, the set of hardware and software resources is updated.

Table 5.1 resumes some of the most significant experimental results obtained from the comparison of our solution, namely CS, and other two of the most popular scheduling techniques used in literature, i.e. Easy Backfilling (Easy BF) [101] and its enhanced version Flexible Backfilling (Flex BF) [102]. The tests have been conducted on GridSim platform [103] by simulating a stream of 1500 jobs and varying the average job inter-arrival time $\mathrm{T_a}$ in the set of simulator time-unit values $\{ 1, 4, 6 \}$ (all details on the test methodology and further experiments are shown in [13]). In particu-

T_a	Scheduling Technique	Resource Usage	Job Slowown	Scheduling Overhead (sec.)	Late Jobs
1	Easy BF	89.31%	2.82	32.4	56.2%
	Flex BF	90.74%	2.79	**31.9**	54.5%
	CS	**94.42**%	**1.79**	32.7	**40.7**%
4	Easy BF	86.66%	1.63	19.3	38.9%
	Flex BF	86.98%	1.59	**18.9**	37.9%
	CS	**92.10**%	**1.18**	21.0	**23.3**%
6	Easy BF	93.30%	1.14	15.3	24.4%
	Flex BF	93.32%	1.11	**14.8**	24.3%
	CS	**98.65**%	**1.03**	20.9	**16.4**%

Table 5.1: Performance of different scheduling techniques in different tests varying the job inter-arrival time (in bold the best results).

lar the table consists of the following columns: (*i*) Resource Usage, which shows the average hardware and software resource utilization percentage; (*ii*) Job Slowdown, which shows the average job slowdown that is calculated, for each job, as $(T_w + T_e)/T_e$, with T_w the time that a job spends waiting to start and/or restart its execution, and T_e the job execution time; (*iii*) Scheduling Overhead, which shows the overall time required to compute the scheduling plans; (*iv*) Late Job, which shows the average percentage of jobs ending after the required deadline. The results clearly show that CS is capable of obtaining a more effective and efficient resource utilization by spending a negligible extra-time.

B. Efficient Diversification of WSE Results

Users interact with WSEs by usually typing a few keywords representing their information need. These keywords, however, are often ambiguous and have more than one possible interpretation [104]. By diversifying search results for covering at least the most popular interpretations of these queries, we would avoid the submission of most of these second-step queries resulting in a twofold advantages: minimizing the risk of dissatisfaction of WSE users, and decreasing the computational load over it. In our formulation, we mostly focus our contribution on presenting *efficient* diversification techniques that would not impact negatively on the quality of the query results. Result diversification is defined as the problem of maximizing the quality of the results returned for ambiguous queries by taking into account the past specializations of such queries available in the query log.

Query Log Based Diversification. We assume that a query log Q is composed by a set of records $\langle\ q_i,\ u_i, t_i,\ V_i,\ G_i\ \rangle$ registering, for each submitted query q_i: (*i*) the anonymous user u_i; (*ii*) the timestamp t_i at

which u_i issued q_i; (iii) the set V_i of URLs of documents returned as top-k results of the query, and, (iv), the set C_i of URLs corresponding to results clicked by u_i.

Users generally query a search engine by submitting a sequence of requests. Splitting the chronologically ordered sequence of queries submitted by a given user into *sessions*, is a challenging research topic [105, 106]. Since session splitting methodologies are out of the scope of this study, we resort to adopt a state-of-the-art technique to devise user logical sessions whose discovering is very important for our result diversification framework. For a complete coverage of the method we refer to the original papers [107]. As a result, by processing the query log Q we obtain the set of logical user sessions exploited by our result diversification solution which is entirely based on information mined from query logs.

Mining Specializations from Query Logs. Let q and q' be two queries submitted by the same user during the same logical session recorded in Q. We adopt the terminology proposed in [107], and we say that a query q' is a "specialization" of q if the user information need is stated more precisely in q' than in q. Let us call S_q the set of specializations of an ambiguous/faceted query q mined from the query log. Given the above generic definition, any algorithm that exploits the knowledge present in query log sessions to provide users with useful suggestions of related queries, can be easily adapted to the purpose of devising specializations of submitted queries. Given the popularity function $f()$ that computes the frequency of a query topic in Q, and a query recommendation algorithm $\mathcal{A}$ trained with query log Q, the *Ambiguous Query Detect* algorithm proposed in [17] can be used to detect efficiently and effectively queries that can benefit from result diversification, and to compute for them the set of popular specializations along with their probabilities. The algorithm used learns the suggestion model from the query log, and returns as related specializations, only queries that are present in Q, and for which related probabilities can be, thus, easily computed. Also, this makes it an efficient and effective technique to devise ambiguous queries and their specializations.

Definition 5 (Probability of Specialization)**.** *Let* $\widehat{Q} = \{q \in Q,\ s.t.\ |S_q| > 1\}$ *be the set of ambiguous queries in* Q*, and* $P(q'|q)$ *the probability for* $q \in \widehat{Q}$ *to be specialized from* $q' \in S_q$*.*

We assume that the distribution underlying the possible specialization of an ambiguous query is known and complete, i.e., $\sum_{q' \in S_q} P(q'|q) = 1$, and $P(q'|q) = 0, \forall q' \notin S_q,\ \forall q \in \widehat{Q}$. To our purposes these probability distributions are simply estimated by the following formula:

$$P(q'|q) = \frac{f(q')}{\sum_{q' \in S_q} f(q')}$$

Now, let us give some additional assumptions and notations. $\mathcal{D}$ is the collection of documents indexed by the search engine which returns, for each submitted query q, an ordered list R_q of documents. The rank of document $d \in \mathcal{D}$ within R_q is indicated with $rank(d, R_q)$. Moreover, let d_1 and d_2 be two documents of $\mathcal{D}$, and $\delta : \mathcal{D} \times \mathcal{D} \rightarrow [0,1]$ a distance function having the non-negative, and symmetric properties, i.e. (i) $\delta(d_1, d_2) = 0$ iff $d_1 = d_2$, and (ii) $\delta(d_1, d_2) = \delta(d_2, d_1)$.

Definition 6 (Results' Utility)**.** *The* utility *of a result $d \in R_q$ for a specialization q' is defined as:*

$$U(d|R_{q'}) = \sum_{d' \in R_{q'}} \frac{1 - \delta(d, d')}{rank(d', R_{q'})} \tag{5.1}$$

where $R_{q'}$ is the list of results that the search engine returned for specialized query q'.

Such utility represents how good $d \in R_q$ is for satisfying a user intent that is better represented by specialization q'. The intuition for U is that a result $d \in R_q$ is more useful for specialization q' if it is very similar to a highly ranked item contained in the results list $R_{q'}$. The utility function specified in Equation (5.1) uses a function to measure the distance between documents. Let us define such function $\delta(d_1, d_2)$ as follows:

$$\delta(d_1, d_2) = 1 - cosine(d_1, d_2) \tag{5.2}$$

where $cosine(d_1, d_2)$ is the cosine similarity between the two documents.

Using the above definitions, we are able to define different query-logs-based approaches to diversification. In [17] the Agrawal *et al.* [108] algorithm, and the Santos's xQuAD framework [109] have been adapted by using the above formulation. The following formulation, instead, describes our solution.

The MaxUtiliy-Diversify Problem. The problem addressed in the Agrawal's paper, is actually the maximization of the *weighted* coverage of the categories with pertinent results. The objective function does not consider directly the number of categories covered by the result set; it might be the case that even if the categories are less than $|S_q|$, some of these will not be covered by the results set. This may happen because the objective function considers explicitly how much a document satisfies a given category. Hence, if a category that is a dominant interpretation of the query q is not covered adequately, more documents related to such category will be selected, possibly at the expense of other categories.

We believe, instead, that it is possible to maximize the sum of the various utilities for the chosen subset S of documents by guaranteeing that query specializations are covered proportionally to the associated probabilities $P(q'|q)$. Motivated by the above observation, we define the following problem.

Definition 7 (MAXUTILITY-DIVERSIFY)**.** *Given: query q, the set R_q of results for q, the probabilities $P(q'|q)$ for all the various specializations $q' \in S_q$, the utilities $U(d|R_{q'})$ of documents, and an integer k. Find a set of documents $S \subseteq R_q$ with $|S| = k$ that maximizes*

$$U(S|q) = \overbrace{(1-\lambda)|S_q| \sum_{d \in S} P(d|q)}^{Relevance} + \underbrace{\lambda \sum_{q' \in S_q} P(q'|q) \sum_{d \in S} U(d|R_{q'})}_{Diversity} \quad (5.3)$$

with the constraints that every specialization is covered proportionally to its probability. Formally, let $R_q \bowtie q' = \{d \in R_q | U(d|R_{q'}) > 0\}$. We require that for each $q' \in S_q$, $|R_q \bowtie q'| \geq \lfloor kP(q'|q) \rfloor$.

Our *OptSelect* algorithm aims at selecting from R_q, the k results that maximize the overall utility of the results list. When $|S_q| \leq k$ the results are in someway split into $|S_q|$ subsets each one covering a distinct specializations. The more popular a specialization, the greater the number of results relevant for it. Obviously, if $|S_q| > k$ we select from S_q the k specializations with the largest probabilities.

Since $U(d|R_{q'})$ measures the utility of a document d when the submitted query was q and the intended specialization q', the total utility for each specialization q' equals to the sum of the utilities of all the selected documents. Finally, the sum over all possible specializations of the query q weighted by $P(q'|q)$, gives the possibility to maximize the global utility of the results set R_q over all the considered specializations of the query.

While QL-DIVERSIFY aims to maximize the probability of covering useful categories, the MAXUTILITY-DIVERSIFY aims to maximize directly the overall utility. This simple relaxation allows the problem to be simplified and solved in a very simple and efficient way. Furthermore, the constraints bounding the minimum number of results tied to a given specialization, boost the quality of the final diversified result list, ensuring that the covered specializations reflect the most popular preferences expressed by users in the past.

Therefore, to maximize $U(S|q)$ in Equation (5.3) we simply resort to compute for each $d \in R_q$ the utility of d for specializations $q' \in S_q$ and, then, to *select* the top-k highest ranked documents. Obviously, we have to carefully select results to be included in the final list in order to avoid choosing results that are relevant only for a single specialization. For this reason we use a collection of $|S_q|$ min-heaps each of those keeps the top $\lfloor k \cdot P(q'|q) \rfloor + 1$ useful documents for that specialization. The algorithm returns the set S maximizing the objective function in Equation (5.3). Moreover, the running time of the algorithm is linear in the size of document considered. Indeed, all the heap operations are carried out on data structures having a constant size bounded by k.

Similarly to the other two solutions discussed, the proposed solution is computed by using a greedy algorithm. OptSelect is however computationally less expensive than its competitors. The main reason is that for each inserted element, it does not recompute the marginal utility of the remaining documents w.r.t. all the specializations. The main computational cost is given by the procedure which tries to add elements to each heap related to a specialization in S_q. Since each heap is of at most k positions, each insertion has cost $\log k$, and globally the algorithm costs $O(\ |R_q|\cdot|S_q|\cdot\log k\ )$.

The conducted tests concern the efficacy and the efficiency of our solution of the three studied methods. Regarding to the efficacy, the results reported in the [17] show that OptSelect and xQuAD behave similarly, while IASelect performs always worse. After efficacy, we conducted some tests in the TREC 2009 Web track's Diversity Task framework to evaluate the efficiency. In particular, we measured the time needed by OptSelect, xQuAD and IASelect to diversify the list of retrieved documents. The average time required by the three algorithms to diversify the initial set of documents for the 50 queries of the TREC 2009 Web Track's Diversity Task show that our approach is two orders of magnitude faster than its competitors.

Diversification in Additive MLR Systems. Now, we show how to derive the most likely refinements, and how to use them to diversify the list of results. Our focus is on plugging efficient diversification in additive Machine Learned Ranking (MLR) systems. In modern WSE query response time constraints are satisfied employing a two-phase scoring. The first phase inaccurately selects a small subset of potentially relevant documents from the entire collection (e.g. a BM25 variant). In the second phase, resulting candidate documents are scored again by a complex and accurate MLR architecture. The final rank is usually determined by additive ensembles (e.g. boosted decision trees [15]), where many scorers are executed sequentially in a chain and the results of the scorers are added to compute the final document score.

Here, we show how such a solution needs to be adapted in order to be plugged in a modern MLR system having a pipelined architecture. Let us assume that, given a query q, MLR algorithms are used to rank a set $D = \{d_1, \ldots, d_m\}$ of documents according to their relevance to q. Then the k documents with the highest score are returned. To this end, additive ensembles are used to compute the final score $s(d_i)$ of a document d_i as a sum over many, simple scorers, i.e. $s(d_i) = \sum_{j=1}^{n} f_j(d_i)$, where f_j is a scorer that belongs to a set of n scorers executed in a sequence. Moreover, the set of scorers is expected to be sorted by decreasing order of importance. This because, as argued in [100], if we can estimate the likelihood that d_i will end up within the top-k documents, we can early exit the $s(d_i)$ computation at any position $t < n$, computing a partial final score using only the first t scorers. For these reasons, it is important to define a solution that is fully integrable with the existing systems. Another important aspect to consider

is the cost of each f_j that must be sustainable w.r.t. the others scorers. In particular, we assume that the cost c of computing $f_j(d_i)$ is constant and the total cost of scoring all documents in D is, thus $\mathcal{C}(D) = c \cdot m \cdot n$. For tasks with tight constraints on execution time, this cost is not sustainable if both m and n are high (e.g. $m > 10^5$ and $n > 10^3$ as shown in [100]).

To achieve the previously specified goal, WSE needs some additional modules in order to enable the diversification stage, see Figure 5.3. Briefly, our idea is the following. Given a query q, perform simultaneously both the selection of the documents potentially relevant for q from the entire collection (module `BM25`) and the retrieve of the specializations for q (module `SS`). Assuming that `SS` performs faster than both `DR` and `BM25`, the module $f_{\texttt{DVR}}$ can be placed in any position of the MLR pipeline, i.e. $f_1 \rightarrow_{\ldots} f_n$. The target of $f_{\texttt{DVR}}$ is, then, to exploit Equation (5.1) for properly increasing the rank of the incoming documents as the other pipelined scorers do. Note that in this case, that is different from *OptSelect* running context, the final extraction of top-k documents is left to the MLR pipeline that already performs this operation automatically. In the following, we give more detail on our approach.

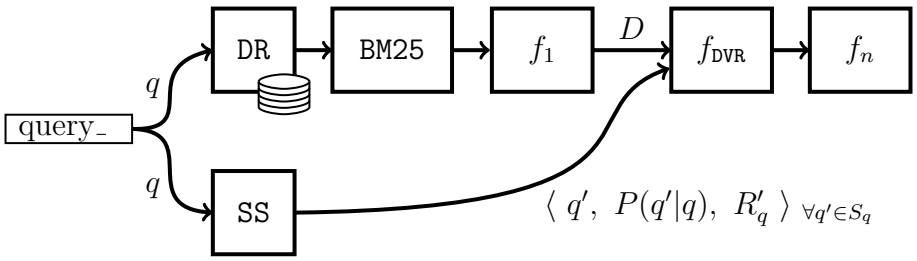


Figure 5.3: A sketch of the WSE architecture enabling diversification.

For any given query q submitted to the engine, we dispatch q to the document retriever `DR` that processes the query on the inverted index, and to the module `SS` that generates the specializations S_q for q. `SS` processes q on a specific inverted index structure derived from query logs: the same proposed in [110]. `SS` returns a set of specializations S_q, a distribution of probability $P(q'|q) \forall q' \in S_q$, and a set $R_{q'} \forall q' \in S_q$ of sketches representing the most relevant documents for each specialization. Concerning the feasibility in space of the inverted index in `SS`, note that each set $R_{q'}$ related to a specialization $q' \in S_q$ is very small compared to the set of whole documents R_q to re-rank, i.e. $|R_{q'}| \ll |R_q|$. Furthermore, using *shingles* [111], only a sketch of a few hundred bytes, and not the whole documents, can be used to represent a document without significant loss in the precision of our method[1]. Resuming, let ℓ be the average size in bytes of a shingle representing a document and let h be the average space needed to store the set S_q of specializations for a query q by using the related inverted index,

[1] note that shingles are already maintained by the WSE for near duplicate document detection.

we need at most ($N \cdot |S_{\widehat{q}}| \cdot |R_{\widehat{q}'}| \cdot \ell \; + \; N \cdot h$) bytes for storing N ambiguous query along with the data needed to assess the similarity among results lists.

Now, let us focus on $f_{\texttt{DVR}}$. As the other modules belonging to the MLR pipeline, also $f_{\texttt{DVR}}$ receives a set of documents D as a stream from its preceding module, scores the elements, then release the updated set. However, contrarily to other diversifying methods analyzed in [17], $f_{\texttt{DVR}}$ is able to compute on the fly the diversity-score for each document d. In fact, exploiting the knowledge retrieved from the query log, our approach does not require to know in advance the composition of D to diversify the query result because `SS` provides the proper mix of different means related to q. In particular, we firstly compute for each $d \in D$ the related shingle. As stated in [111], the related sketch can be efficiently computed (in time linear in the size of the document d) and, given two sketches, the similarity $1 - \delta(d, d')$ of the corresponding documents (i.e. $d \in D$ and each document d' returned by `SS`, i.e. $d' \in R_{q'} \; \forall q' \in S_q$) can be computed in time linear in the size of the sketches. The resulting similarity thus concurs to $U(d|R_{q'})$, i.e. the variation of final score of the document d.

As conclusion, exploiting this approach, the selection of the relevant results to return to the user can be done by simply selecting the top-k documents with the highest score.

Bibliography

[1] R. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri. Challenges on distributed web retrieval. In *ICDE*. IEEE, 2007.

[2] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269, New York, NY, USA, 2008. ACM.

[3] Marc de Kruijf and Karthikeyan Sankaralingam. Mapreduce for the cell b.e. architecture. In *IBM Journal of Research and Development*, 2007.

[4] Steven Fortune and James Wyllie. Parallelism in random access machines. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118, New York, NY, USA, 1978. ACM.

[5] Snyder Lawrence. Type architectures, shared memory, and the corollary of modest potential. *Annual review of computer science*, 1986.

[6] L. A. Barroso, J. Dean, and Holzle. Web search for a planet: The google cluster architecture. *Micro, IEEE*, 23(2):22–28, 2003.

[7] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *Micro, IEEE*, 25(2):21–29, 2005.

[8] John D. Davis, James Laudon, and Kunle Olukotun. Maximizing CMP throughput with mediocre cores. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 51–62, Washington, DC, USA, 2005. IEEE Computer Society.

[9] M. Garland and D.B. Kirk. Understanding throughput-oriented architectures. *Communications of the ACM*, 53(11):58–66, 2010.

[10] S. Collange, D. Defour, and A. Tisserand. Power consumption of GPUs from a software perspective. *Computational Science–ICCS 2009*, pages 914–923, 2009.

[11] S. Borkar and A.A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[12] Gabriele Capannini, Ranieri Baraglia, Diego Puppin, Laura Ricci, and Marco Pasquali. A job scheduling framework for large computing farms. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 54:1–54:10, New York, NY, USA, 2007. ACM.

[13] Ranieri Baraglia, Gabriele Capannini, Patrizio Dazzi, and Giancarlo Pagano. A multi-criteria job scheduling framework for large computing farms. In *Proc. of the 10th IEEE International Conference on Computer and Information Technology*, pages 187–194, September 2010.

[14] Ranieri Baraglia, Gabriele Capannini, Domenico Laforenza, Marco Pasquali, and Laura Ricci. A multi-level scheduler for batch jobs on grids. *The Journal of Supercomputing*, pages 1–18, 2011.

[15] Z. Zheng, H. Zha, T. Zhang, O. Chapelle, K. Chen, and G. Sun. A general boosting method and its application to learning ranking functions for web search. *Advances in Neural Information Processing Systems*, 19, 2007.

[16] Gabriele Capannini, Franco Maria Nardini, Raffaele Perego, and Fabrizio Silvestri. Efficient diversification of search results using query logs. In *20th International Conference on World Wide Web (Accepted Poster)*, Hyderabad, India, March 28 - April 1 2011. ACM.

[17] Gabriele Capannini, Franco Maria Nardini, Raffaele Perego, and Fabrizio Silvestri. Efficient diversification of web search results. *Proceedings of the VLDB, Vol. 4, No. 7*, April 2001.

[18] J. Bovay, B. Henderson Brent, H. Lin, and K. Wadleigh. Accelerators for high performance computing investigation. White paper, High Performance Computing Division - Hewlett-Packard Company, 2007.

[19] Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[20] M. A. Nichols, H. J. Siegel, H. G. Dietz, R. W. Quong, and W. G. Nation. Eliminating memory for fragmentation within partitionable simd/spmd machines. *IEEE Trans. Parallel Distrib. Syst.*, 2(3):290–303, 1991.

[21] S. Rixner. *Stream processor architecture*. Springer Netherlands, 2002.

[22] Brucek Khailany, William J. Dally, Ujval J. Kapasi, Peter Mattson, Jinyung Namkoong, John D. Owens, Brian Towles, Andrew Chang, and Scott Rixner. Imagine: Media processing with streams. *IEEE Micro*, 21(2):35–46, 2001.

[23] J. Pille, C. Adams, T. Christensen, S.R. Cottier, S. Ehrenreich, F. Kono, D. Nelson, O. Takahashi, S. Tokito, O. Torreiter, et al. Implementation of the Cell Broadband Engine in 65 nm SOI Technology Featuring Dual Power Supply SRAM Arrays Supporting 6 GHz at 1.3 V. *Solid-State Circuits, IEEE Journal of*, 43(1):163–171, 2008.

[24] S.P. Song, M. Denman, and J. Chang. The PowerPC 604 risc microprocessor. *IEEE Micro*, 14(5):8–17, 1994.

[25] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[26] K. Skaugen. Petascale to exascale. In *International Supercomputing Conference, Hamburg, Germany*, 2010.

[27] J. N. England. A system for interactive modeling of physical curved surface objects. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 336–340, New York, NY, USA, 1978. ACM.

[28] Opengl, D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, August 2005.

[29] David Blythe. The direct3d 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006.

[30] ATI. http://ati.amd.com/technology/streamcomputing/, 2009.

[31] The Khronos Group. http://www.khronos.org/opencl/, 2009.

[32] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pages 1–14, New York, NY, USA, 2008. ACM.

[33] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[34] Stephen A. Cook and Robert A. Reckhow. Time-bounded random access machines. In *STOC '72: Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 73–80, New York, NY, USA, 1972. ACM.

[35] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 305–314, New York, NY, USA, 1987. ACM.

[36] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Hierarchical memory with block transfer. In *SFCS '87: Proceedings of the 28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 204–216, Washington, DC, USA, 1987. IEEE Computer Society.

[37] D. Jimenez Gonzalez, X. Martorell, and A. Ramirez. Performance analysis of cell broadband engine for high memory bandwidth applications. In *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 210–219. IEEE, 2007.

[38] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[39] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, Washington, DC, USA, 1999. IEEE Computer Society.

[40] Gabriele Capannini, Fabrizio Silvestri, Ranieri Baraglia, and Franco Maria Nardini. Sorting using BItonic netwoRk wIth CUDA. 7th Workshop on LSDS-IR (SIGIR 2009 Workshop), July 2009.

[41] Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Implicit and explicit optimizations for stencil computations. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 51–60, New York, NY, USA, 2006. ACM.

[42] P. B. Gibbons. A more practical pram model. In *SPAA '89: Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 158–168, New York, NY, USA, 1989. ACM.

[43] Kurt Mehlhorn and Uzi Vishkin. Randomized and deterministic simulations of prams by parallel machines with restricted granularity of parallel memories. *Acta Inf.*, 21(4):339–374, 1984.

[44] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. The qrqw pram: accounting for contention in parallel algorithms. In *SODA '94: Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 638–648, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.

[45] A. Aggarwal, A. K. Chandra, and M. Snir. On communication latency in pram computations. In *SPAA '89: Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 11–21, New York, NY, USA, 1989. ACM.

[46] Todd Hervey Heywood. *A practical hierarchical model of parallel computation.* PhD thesis, Syracuse University, Syracuse, NY, USA, 1992.

[47] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[48] Pilar de la Torre and Clyde P. Kruskal. Submachine locality in the bulk synchronous setting (extended abstract). In *Euro-Par '96: Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*, pages 352–358, London, UK, 1996. Springer-Verlag.

[49] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, 1993.

[50] J. S. Vitter and E. A. M. Shriver. Optimal disk i/o with parallel block transfer. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 159–169, New York, NY, USA, 1990. ACM.

[51] Scott J Vitter and A E M Shriver. Algorithms for parallel memory ii: Hierarchical multilevel memories. Technical report, Duke University, Durham, NC, USA, 1993.

[52] Zhiyong Li, Peter H. Mills, and John H. Reif. Models and resource metrics for parallel and distributed computation. In *Proc. 28th Annual Hawaii International Conference on System Sciences*, pages 133–143, 1995.

[53] Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12:12–2, 1994.

[54] Yunquan Zhang. Dram(h,k): A parallel computation model for numerical computing. In *Sixth National Conference on Parallel Computing*, pages 160–165, ChangSha, China, 2000.

[55] Gianfranco Bilardi, Andrea Pietracaprina, Geppino Pucci, and Francesco Silvestri. Network-oblivious algorithms. *Parallel and Distributed Processing Symposium, International*, 0:53, 2007.

[56] Ricardo Baeza-Yates and Mauricio Marin. Crawling a country: Better strategies than breadth-first for web page ordering. In *In Proceedings of the 14th international conference on World Wide Web / Industrial and Practical Experience Track*, pages 864–872. ACM Press, 2005.

[57] Qiancheng Jiang and Yan Zhang. Siterank-based crawling ordering strategy for search engines. In *CIT '07: Proceedings of the 7th IEEE International Conference on Computer and Information Technology*, pages 259–263, Washington, DC, USA, 2007. IEEE Computer Society.

[58] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[59] Nicholas Lester. Fast on-line index construction by geometric partitioning. In *In Proceedings of the 14th ACM Conference on Information and Knowledge Management (CIKM 2005*, pages 776–783. ACM Press, 2005.

[60] C.D. Manning, P. Raghavan, H. Schütze, and Ebooks Corporation. *Introduction to Information Retrieval*, volume 1. Cambridge University Press, 2008.

[61] Shuai Ding, Jinru He, Hao Yan, and Torsten Suel. Using graphics processors for high-performance ir query processing. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 1213–1214, New York, NY, USA, 2008. ACM.

[62] Thomas Neumann, Matthias Bender, Sebastian Michel, Ralf Schenkel, Peter Triantafillou, and Gerhard Weikum. Optimizing distributed top-k queries. In *WISE '08: Proceedings of the 9th international conference on Web Information Systems Engineering*, pages 337–349, Berlin, Heidelberg, 2008. Springer-Verlag.

[63] Pei Cao and Zhe Wang. Efficient top-k query calculation in distributed networks. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 206–215, New York, NY, USA, 2004. ACM.

[64] Sebastian Michel, Peter Triantafillou, and Gerhard Weikum. Klee: a framework for distributed top-k query algorithms. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 637–648. VLDB Endowment, 2005.

[65] F. Silvestri and R. Venturini. VSEncoding: efficient coding and fast decoding of integer lists via dynamic programming. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 1219–1228. ACM, 2010.

[66] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, page 59. IEEE, 2006.

[67] Daniel Cederman and Philippas Tsigas. A practical quicksort algorithm for graphics processors. In *ESA '08: Proceedings of the 16th annual European symposium on Algorithms*, pages 246–258, Berlin, Heidelberg, 2008. Springer-Verlag.

[68] Gabriele Capannini, Ranieri Baraglia, and Fabrizio Silvestri. K-model: A new computational model for stream processors. In *IEEE 12th International Conference on High Performance and Communications (HPCC10)*, Melbourne, Australia, September 2010.

[69] Guy E. Blelloch. *Vector models for data-parallel computing*. MIT Press, Cambridge, MA, USA, 1990.

[70] Gabriele Capannini, Fabrizio Silvestri, and Ranieri Baraglia. Sorting on GPUs for large scale datasets: A thorough comparison. *Information Processing & Management*, In Press, Corrected Proof(60), January 2011.

[71] L.A. Barroso and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 4(1):1–108, 2009.

[72] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In *Parallel and Distributed Processing Symposium, International*, volume 0, pages 1–10, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

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

[74] Naga K. Govindaraju and Dinesh Manocha. Cache-efficient numerical algorithms using graphics hardware. *Parallel Comput.*, 33(10-11), 2007.

[75] Gabriele Capannini, Fabrizio Silvestri, and Ranieri Baraglia. Mapping bitonic sorting network onto gpus. ComplexHPC Meeting, Universidade Tecnica de Lisboa, Portugal, October 19-21 2009.

[76] Majed Z. Al-Hajery and Kenneth E. Batcher. Multicast bitonic network. In *SPDP*, pages 320–326, 1993.

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

[78] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *SIGGRAPH Conference on Graphics Hardware*, 2003.

[79] 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 *In Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*. ACM Press, 2000.

[80] Peter Kipfer, Mark Segal, and Rüdiger Westermann. Uberflow: a gpu-based particle engine. In *HWWS '04*, New York, NY, USA, 2004. ACM Press.

[81] Peter Kipfer and Rüdiger Westermann. Improved GPU sorting. In Matt Pharr, editor, *GPUGems 2*. Addison-Wesley, 2005.

[82] Alexander Greß and Gabriel Zachmann. Gpu-abisort: Optimal parallel sorting on stream architectures. In *IPDPS '06*, April 2006.

[83] Gianfranco Bilardi and Alexandru Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines. Technical report, Cornell University, Ithaca, NY, USA, 1986.

[84] Naga K. Govindaraju, Nikunj Raghuvanshi, and Dinesh Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *SIGMOD '05*, New York, NY, USA, 2005. ACM.

[85] Martin Dowd, Yehoshua Perl, Larry Rudolph, and Michael Saks. The periodic balanced sorting network. *J. ACM*, 36(4), 1989.

[86] J. Wassenberg and P. Sanders. Engineering a multi-core radix sort. *Euro-Par 2011 Parallel Processing*, pages 160–169, 2011.

[87] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[88] NVIDIA. *Compute Unified Device Architecture Programming Guide*, June 2008.

[89] David R. Helman, David A. Bader Y, and Joseph Jj Z. A randomized parallel sorting algorithm with an experimental study. Technical report, Journal of Parallel and Distributed Computing, 1995.

[90] G.E. Blelloch. Prefix sums and their applications. *Synthesis of Parallel Algorithms*, pages 35–60, 1990.

[91] Justin Hensley, Thorsten Scheuermann, Greg Coombe, Montek Singh, and Anselmo Lastra. Fast summed-area table generation and its applications. *Computer Graphics Forum*, 24:547–555, 2005.

[92] Vibhav Vineet, Pawan Harish, and Suryakant Patidar. Fast minimum spanning tree for large graphs on the gpu, 2005.

[93] Daniel Horn. Stream reduction operations for GPGPU applications. In *GPU Gems 2*, pages 573–589. Addison-Wesley, 2005.

[94] Mark Harris, Shubhabrata Sengupta, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In *GH '07*, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[95] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[96] Lewis W. Tucker and George G. Robertson. Architecture and applications of the connection machine. *IEEE Computer*, 21:26–38, 1988.

[97] Gabriele Capannini. Designing Efficient Parallel Prefix Sum Algorithms for GPUs. *11th IEEE International Conference on Computer and Information Technology (CIT 2011)*, 31th August 2011.

[98] NVIDIA Corporation. Programming guide 2.0.

[99] Gabriele Capannini, Franco Maria Nardini, Raffaele Perego, and Fabrizio Silvestri. A search architecture enabling efficient diversification of search results. *DDR-2011 in conjunction with ECIR 2011 – the 33rd European Conference on Information Retrieval*, 18th April 2011.

[100] B.B. Cambazoglu, H. Zaragoza, O. Chapelle, J. Chen, C. Liao, Z. Zheng, and J. Degenhardt. Early exit optimizations for additive machine learned ranking systems. In *Proceedings of the third ACM international conference on Web search and data mining*, pages 411–420. ACM, 2010.

[101] Ahuva W. Muálem and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Trans. Parallel and Distributed Syst.*, 12(6):529–543, June 2001.

[102] Dror D. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Parallel job scheduling, a status report. In *Job Scheduling Strategies for Parallel Processing 10th International Workshop*, pages 1–16, London, UK, Lect. Notes Comput. Sci. vol. 3277, 2004. Springer-Verlag.

[103] R. Buyya and M. Murshed. *GridSim: a Toolkit for Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing*. John Wiley & Sons, Ltd, 2002.

[104] Filip Radlinski and Susan Dumais. Improving personalized web search using result diversification. In *Proc. SIGIR'06*. ACM, 2006.

[105] Ricardo Baeza-Yates. Graphs from search engine queries. In *Proc. SOFSEM'07*, pages 1–8, Harrachov, CZ, 2007.

[106] Rosie Jones and Kristina Lisa Klinkner. Beyond the session timeout: automatic hierarchical segmentation of search topics in query logs. In *Proc. CIKM'08*. ACM, 2008.

[107] Paolo Boldi, Francesco Bonchi, Carlos Castillo, and Sebastiano Vigna. From 'dango' to 'cakes': Query reformulation models and patterns. In *Proc. WI'09*. IEEE CS Press, 2009.

[108] Rakesh Agrawal, Sreenivas Gollapudi, Alan Halverson, and Samuel Ieong. Diversifying search results. In *Proc. WSDM'09*. ACM, 2009.

[109] Rodrygo Santos, Craig Macdonald, and Iadh Ounis. Exploiting query reformulations for web search result diversification. In *Proc. WWW'10*. ACM Press, 2010.

[110] Daniele Broccolo, Lorenzo Marcon, Franco Maria Nardini, Raffaele Perego, and Fabrizio Silvestri. An efficient algorithm to generate search shortcuts. Technical Report 2010-TR-017, CNR ISTI Pisa, July 2010.

[111] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Comput. Netw. ISDN Syst.*, 29:1157–1166, September 1997.