UNIVERSITÉ LIBRE DE BRUXELLES
Faculté des Sciences
Département d'Informatique

Fast dynamic predecessor queries on GPU

Youri Hubaut

Promotor : John Iacono

Master Thesis in Computer Sciences

Academic year 2017 - 2018

"*Celerius quam asparagi cocuntur.*"

Gaius Suetonius Tranquillus, 87 AD

Acknowledgment

I would like to thank my master thesis advisor, John Iacono, for his encouragements and research ideas that led to this thesis. This work simply wouldn't have been possible without him, its critical thinking and clever remarks.

To Jean Cardinal, Stefan Langerman and Jan Lemeire who accepted to be members of the jury for this thesis and who aroused interest in the different aspects and themes of this work.

To the Computer Science departement of Université Libre de Bruxelles (U.L.B.), which provided a fertile research environment. I would like to thank its members, both students and faculty, for their valuable ideas and insights that have immensely broadened my education.

In closing, I would also like to thank all those who to some various degree have helped me with their interests and encouragements.

Contents

Chapter 1

Introduction

1.1 Background and objectives of the thesis

In recent years, we have observed a certain stagnation in the computing power of our processors. Moore's law is no longer observed as a result of technical and material issues that are increasingly difficult to overcome; we are also approaching physical limits of our creation. A proposed solution to these problems was the introduction of multi-core processors. However, taking full advantage of the power offered by these new devices is challenging and new algorithms need to be developed since it is not always trivial to transform a sequential algorithm in a parallel one.

Even more recently, the idea came up of using graphics cards which offer high parallelization capacity at a very low cost. Further research has been carried out in this direction to find out the theoretical limits of such devices and what could be done to maximize our implementation.

In this perspective, we will try to adapt an existing data structure to this new programming paradigm. We will see what are the main advantages and disadvantages of this approach and for which purposes it can be used. We will also consider ways to make it highly concurrent and therefore take full advantage of the computational power offered by graphics cards.

We therefore propose a trip to the world of graphic cards during which we will address many themes related to this environment. We will start by explaining the more general context and how such devices work, then we will investigate how data can be accessed through their natural parallelism. We can then focus on the data structure we want to adapt to this paradigm, first in a sequential fashion. We will then return to one of its main components in order to resolve one of the problems related to the concurrency of such a data structure.

1.2 Structure of the thesis

In this work, we will begin by specifying the context in a more precise way and we will define the objectives targeted and pursued. We will then retrace a brief history of computational models linked to parallelism and, in particular, those that are easily exploitable in order to study the complexity of algorithms on graphic cards. Next, we will briefly explain how current graphic cards work, the abstractions they provide and the aspects to which attention should be paid. We will continue by presenting a synthesis of different results obtained for a particular computation model.

We will then start on the main interest of this work which consists of adapting a data structure to this new paradigm and we will discuss briefly its origins. We will first present the capabilities of such a structure in a sequential framework, which will simplify reasoning and implementation. This data structure allows classic operations related to trees: insertions, deletions, search, predecessor and successor queries and therefore corresponds to the "dictionary" family. It can also be used as a heap or a priority queue.

But first, we will tackle another theme which is intrinsically linked to data structures; the way of accessing data by its patterns plays a crucial role in its overall efficiency and we will see that the graphics card model offers a certain range of possibilities. We will see that some accesses have very poor raw characteristics, but which, if exploited intelligently, can ultimately remain competitive with others.

We will then discuss the different ways to implement this data structure and its intrinsic difficulties. Once the implementations are complete, we will be able to present the effectiveness of the various operations offered by this structure in comparison to another fair and classical data structure.

Then, we will come back to an essential component of this data structure that are hash tables. We will try to compare different possible implementations and their relative performance in a highly concurrent framework. Finally, we will adapt our data structure to incorporate elements of concurrence and discuss opportunities for improvement and potential applications. Finally, we will propose a comparison of the performances proposed by such a structure in comparison with another, developed recently, and proposing the same characteristics of dictionnary.

It will then only be necessary to provide a summary of the various conclusions that we have reached with more details in their respective chapters. This will be the opportunity to provide a final conclusion on the desirability and place of such a data structure.

1.3 Contributions

In the following, we try to group our main contributions in a single list in order to present the general interest of the work and its associated advancements:

1. A state of the art gathering many results obtained for the PEM computation model with detailed explanations for some algorithms or complexity[3].
2. A brief study of memory behaviours according to access patterns on GPU[4].
3. An adaptation of an existing data structure within the framework of graphic cards[6].
4. A study of several implementations for concurrent hash tables on GPU[7].
5. Adaptation of the X-fast tries to concurrent framework[8].
6. Analysis of the concurrent X-fast tries on GPU[8.4].

1.4 Notation

The next list describes several symbols that will be later used within the body of the document. We will clarify in due course whether some symbols require further details.

Symbol	Meaning
N	number of elements, problem size.
P	number of processors.
M	size of the local cache.
B	size of memory block transfered between global memory and local cache.
w	word size in word-RAM computation model.
u	size of the universe, cardinality of a bounded set of integers.
V	number of vertices of a graph.
E	number of edges of a graph.
$\lceil x \rceil$	ceil function, smallest integer larger than x.
$\log_b x$	base-b logarithm of x; if b is not specified, we use $b = 2$.
$n!$	Factorial function.
$\binom{n}{k}$	Binomial coefficient.
$f(n) = O(g(n))$	the complexity of $f(n)$ is upper bounded by $g(n)$ as $n \to \infty$. Formally, if it exists c and n_0 such that $\|f(n)\| \leq c\|g(n)\| \ \forall n \geq n_0$
$f(n) = \Omega(g(n))$	the complexity of $f(n)$ is lower bounded by $g(n)$ as $n \to \infty$. Formally, if it exists c and n_0 such that $\|f(n)\| \geq c\|g(n)\| \ \forall n \geq n_0$
$f(n) = \Theta(g(n))$	the complexity is bounded both above and below by $g(n)$ as $n \to \infty$. Formally, if it exists c_1, c_2 and n_0 such that $c_1\|g(n)\| \leq f(n) \leq c_2\|g(n)\| \ \forall n \geq n_0$
scan(N)	optimal number of I/Os to scan N items.
sort(N)	optimal number of I/Os to sort N items.
CPU	Computing Processing Unit.
GPU	Graphics Processing Unit.
w.h.p.	with high probability.

Chapter 2

Context and Computation Model

In this part of the work, we will start by going over the general context of this problem and will take the opportunity to highlight the key factors of this kind of issues in order to propose solutions or, at least, try to resolve them. We will them follow by giving a brief summary of the numerous computation models related to parallelism to arrive at those more adapted to the representation of graphic cards. We will then give a high level explanation of how graphics cards work, their associated terminology and the issues that are intrinsically linked to them.

2.1 General context

Moore's law continues to be observed as a result of the various improvements both in the transistor, by a reduction of its size, and in the architecture, with increasingly complex realizations, which improve the computational power-energy ratio. Nevertheless, we see that it is increasingly difficult to stand up to different physical barriers [73]. One solution to this problem is the multiplication of computation units, among others such as quantum computers, 3D layering, new materials or photo-electronics. It can bypass the Pollack's Rule[1] to provide better yields. But that's not the only and single advantage:

The necessary computational power can be adjusted as best as possible (decrease the frequency or switch off the unit). This helps to balance the workload and distribute heat more evenly. This can also offer more resiliency, easier design and more robust architecture, less prone to bugs.

However, this flexibility is paid for by smaller and therefore less efficient individual computation units. This means that more parallelization is needed to achieve the same results, leading to communication problems for memory access and message transmission [32]. It becomes all the more important to be able to offer algorithms that can run as independently as possible in order to take full advantage of all the parallelism offered.

Then comes Amdahl's law, which presents a very defeatist observation on the parallelism capacity of algorithms and their relative speed-up, the quantity of sequential part of the algorithms limits very strongly the theoretical maximum possible speed-up; or Gustafson's law, which wants to be more optimistic, by pointing out that the increase in computational power will also make it possible to respond to problems of greater size. In the following equations, S represents the theoretical speedup in latency of the execution of the whole task, p is the percentage of the execution workload which can me

[1] Performance increase due to microarchitecture advances is roughly proportional to the square root of the increase in complexity [32].

made parallel, n the number of elements which can run in parallel and s is the latency acceleration of the execution of the part of the task benefiting from the improvement of the system resources.

$$S(n) = \frac{1}{(1-p)+\frac{p}{n}} \qquad S(n) = 1 - p + sp$$

Amdhal's law Gustafson's law

It is within this framework that we have sought to develop a data structure by adapting it to the context of graphic cards. This structure allows classic operations that appear in many algorithms and presents very interesting theoretical and technical characteristics. It also seeks to study a new way, that of using primitive elements of parallelism together in order to respond to the same problem and thus benefit from the additional computational power offered by such a practice.

The problem can be summarized as follows: we have new hardware with a very simplified architecture that offers massive parallelism. We therefore want to make the most of the available performance by exploiting both these new architectures and the very large number of processors. It should be noted that these devices also propose very specific notions to which it is necessary to pay attention.

2.2 Model of computation

Consequently, people began to look at how to parallelize algorithms and compare them through computational models to determine which ones performed best. The notions of complexity have been adapted to this new context and we have seen the emergence of many algorithms often presenting very elegant solutions. In this section, we will give a quick review of different models of computation which exist to compute the complexity of algorithms on parallel systems. There are countless of them with as many variations, we will limit ourselves to a "canonical" subset, the most famous ones and that will lead us to a particular computation model.

2.2.1 PRAM

This model was introduced in 1978 by S. Fortune and J. Wyllie [51, 99]. They knew that we could not keep increasing performances of computers forever and therefore that one solution to this problem was to introduce parallelism in the computations. But, at that time, not many tried to determine what was the theoretical power of such machines.

The parallel random access machine (PRAM) can really be seen as an extension of the classical RAM model [43]. It describes an unbounded set of processors, with an infinite global memory and a set of input registers. Each processor has its own accumular, unbounded local memory, program counter and a flag which indicates whether or not the processor is running; there are also a list basic of instructions (load, store, add, jump, read) and some specific (fork and halt). The idea was to remain relatively close to the programming languages of the time while presenting a certain abstraction necessary to calculate the complexity of the various algorithms.

Its main specifities are the fork instruction, which initializes an inactive processor to start at a label, and the halt to stop running, other instructions are assimilated to the assembler's expression capacities. Traditionally, several processors are allowed to read

simultaneously the same data but only one can write at a time on a specific cell; this assumption is often supported by the majority of materials designed as MIMD and is one of the possibilities offered by the model which can propose exclusivity or concurrency as well on the readings as on the writings since each instruction is executed in a cycle of three phases: a read (if any), a computation and a write (if any).

But most importantly, it defines classes of complexity where execution is determined on termination of the first processor. Hence, the time to accept an input is the minimum over all the computations from the first processor. Therefore, deterministic and non deterministic $T(n)$-TIME-PRAM classes are classically defined, some equivalences were also proven: $T(n)$-TIME-PRAM = $T(n)$-SPACE or N-$cT(n)$-TIME-PRAM = N-$2^{cT(n)}$-TIME (for $T(n) \geqslant \log(n)$) [51].

For the first theorem, one should observe that the number of machine configurations in $T(n)$ algorithm is bounded by $2^{dT(n)}$. Then the equivalence is described as follow, the first processor is able to launch $2^{dT(n)}$ processes in $O(T(n))$, each holding a different number representing one configuration. They all decode their given configuration, compute the next one and encode it. Finally, to determine if the word is accepted from the initial configuration, it is sufficient for each processor to search for the successor configuration of the successor iteratively until we reach the termination state from the initial one (path-doubling strategy); all these operations can be performed in $O(T(N))$.

For the second, a similar argument can be used, each processor can guess one symbol of computation. The first n processors check initial tape configuration, the others check that the symbol they consider corresponds well to that of the transition from the previous configuration and neighboring symbols. The latest processors must also verify that the state is accepting.

Many theoretical results on the bounds of algorithms were also obtained on models with slight variations in comparison to the original one [68]. However, this model has some characteristics that make it unrealistic in practice. Indeed, there is no limit on the number of processors available on the machine. All basic operations are in constant time while some are obviously slower than others. Each processor can access any data and from anywhere, there is no difference between shared and distributed memory. Resource access management is simply ignored, there are no data contentions for instance. But it has the advantage of offering a working environment close to the mental conception of the problem.

2.2.2 BSP

Bulk Synchronous Parallel (BSP) was introduced in the late eighties by L. Valiant [92]. It was designed to lie inbetween hardware and programming models, in the same way that the von Neumann machine is comparable to the Turing machine. This computer consists of three parts: *components* able to perform processing (asynchronously) and memory operations, a *router* which dispatchs the messages between pairs and a primitive (like a barrier) which allows synchronisation of all or only a subset of processors. This aims to symbolize the three main steps involved in any parallel algorithm, concurrent computing, communication and synchronization. This model thus makes it possible to represent other aspects which intervene in this type of algorithm and does not concentrate only on the computational complexity of the problem.

A computation, in this model, is described as a sequence of *supersteps*. Each superstep is divided into three stages, a first where each component is allocated a task consisting of some local computation, then a communication phase is realized in order to

propagate the results thanks to message transmissions and (implicitly) message arrivals from other components. Finally, after each period of L time units, a global check is made to determine whether the system managed to accomplish the work within the allotted time. If it has, the machine can proceed the next superstep. Otherwirse, a new period is allocated to terminate the current superstep. Those L units of time are called *periodicity* and can be controlled at runtime. The lower bounds are set by the material capacities while the upper ones are rather software, by design, and related to the granularity of the solution since to achieve the optimal processor utilization, in each superstep, each processor should be assigned a task of approximately L steps.

Some constants are defined like the communication bandwidth, the size of the local memory, the number of local operations per period or the maximal number of messages per superstep; some considerations about timings and relative proportions are also established. Hence, the cost of an algorithm is defined as the sum of three terms, the cost of the longest running local computation, the cost of communication inbetween processes and the cost of the synchronisation barrier (latency) at the end of the superstep. Each of them being summed on the number of supersteps needed to solve the problem.

The model knew many variations on the same theme and was recently updated to include several layers of cache and the notion of optimality without paying attention about the parameters, also known as *oblivious* [93].

Note that the MapReduce model, introduced by Google in 2004 [48], is based on similar concepts and can be reduced to it [84]. It focuses mainly on problems related to Big Data and provides a more practical framework. The data is first loaded and distributed between the different processors. Each then applies a function *map* that transforms the local data and writes the result into a temporary storage area. The results are then redistributed on the basis of keys, also produced during the map operation, so that all the data belonging to the same key are sent to the same processor, this is the big *shuffle*. Finally, each processor reworks those new data by key in parallel, *reduce*. There are thus two "BSP supersteps" in one in this model.

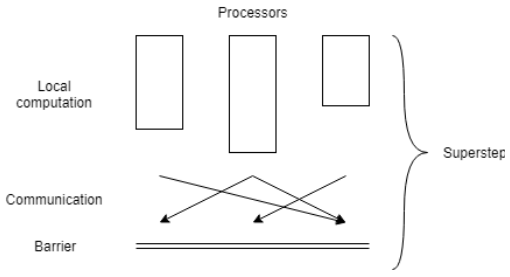


Figure 2.1: BSP

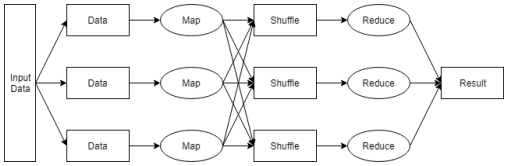


Figure 2.2: MapReduce

2.2.3 LogP

The canonical PRAM model, even though it proposes nice theoretical results, is not representative of real-life computers due to its zero communication delay, synchronism among all the execution steps or infinite bandwidth. On the opposite, BSP may represent a too large variety of machines whereas we see a convergence towards a same architecture for distributed computing: many complete computers connected by a communication network. LogP model proposed in 1993 by Culler and al. [47] tried to take place inbetween those two models. It focuses on high level abstraction of the machine

and target essentially communication (message-passing style).

Some parameters are defined such as an upper bound on latency L incurred by sending a message from a source to a destination, an overhead o linked to the inactivity of the processor while sending or receiving messages, a gap g defined as the minimum time between two consecutive messages and P the number of processes. This model assumes that the network has a finite capacity ($\frac{L}{g}$ messages from any one processor to any other), asynchronous communication and messages are expected to be small,

The BSP and LogP models focus on different concepts, they do not aim at the same purpose. For BSP, barrier synchronisation and routing of messages are primitives and is much likely used to design algorithm. Whereas LogP proposes a better control of resources with loose synchronisation. But, more importantly, they propose a comparable power of computation to design algorithms and so BSP tends to be easier to manage and provides a more convenient programming abstraction [30].

2.2.4 PDM

Aggarwal and Vitter introduced in 1988 [2] the external memory model (EM) or disk access machine (DAM) which counts the number of block I/Os to and from an external source of memory. The idea was that, at the time, the memories were too small to hold the entire dataset, so it was necessary to regularly fetch the data from an external source (i.e. magnetic tapes). EM algorithms explicitly control data placement and transfer to reduce the number of data queries to be made and their inherent slowness, disk accesses are 1 million times slower than the execution of one instruction on modern computer.

Once the desired data location has been found on the tape, accessing neighboring data is very simple. And since we also tend to want to access information located nearby (as in the case of a for loop), it can be interesting to amortize the relatively long initial delay to access a data by transferring a large contiguous group of data items at a time. We use the term *block* to refer to the amount of data transferred to or from one disk in a single I/O operation.

With an analoguous spirit, Vitter and Shriver introduced the parallel disk model (PDM) which combines the notion of locality of references and the parallel accesses. All the processors read in main memories and many blocks of data can be swapped from different disks in parallel. Those models really captured the essential notions to exploit the locality as well as the load balancing to deserve our programs. Many results were achieved through these models and Vitter compiled many algorithms, with their analysis and their associated data structure in one interesting book [97]. Interests about technical difficulties and their practical solutions are also suggested but the main interest relies on sequential algorithms.

PDM uses the following main parameters: N the problem size, M the internal memory size, B the block transfer size, D the number of independent disk drives and P the number of processors.

Such that, in a single I/O, each of the D disks can simultaneously transfer a block of B contiguous data items. The data needs to be placed in internal memory M in order to be able to perform the appropriate processing. The parallelism proposed by this model must be considered more as a weaker form, since the memory is distributed, each processor has its own internal memory and attached set of disks. It could be seen more as P copies of EM machines.

2.2.5 PEM

As the external memory model (and its brother "cache-oblivious") brings the notion of cache and memory transfers to the RAM model, PEM realizes the same idea but with PRAM model. Bender et al., tried in 2005 [29], to combine the notion of cache-oblivious with parallelism but were more focused on one specific data structure, a B-tree. The underlying idea was that the cache can be viewed as a bounded memory which can read or write from an external and global memory.

Finally, L. Arge, M. T. Goodrich, M. Nelson and N. Sitchinava, in 2008 [12], came up with Parallel External Memory (PEM) model. It can be viewed as a single-disk external memory shared by several processes all owning a private cache.

This model differs fundamentally from the previous model of computation as it keeps some shared memory. There is only one external data source that can be accessed by any processor. Nonetheless, each processor has its own internal memory, which introduces thus problems related to concurrency and data consistency. Be aware that no mechanism of synchronisation or communication among processors are provided and further extensions exist to treat the concurrent reads and writes.

It also makes the assumption that the performances are bounded by the I/O and its intrinsic latency. This model is defined by few parameters, N the problem size, P the number of processors, M the cache size and B the block size. So the parameters are essentially the same than the previous one, the only difference is that there is only one disk.

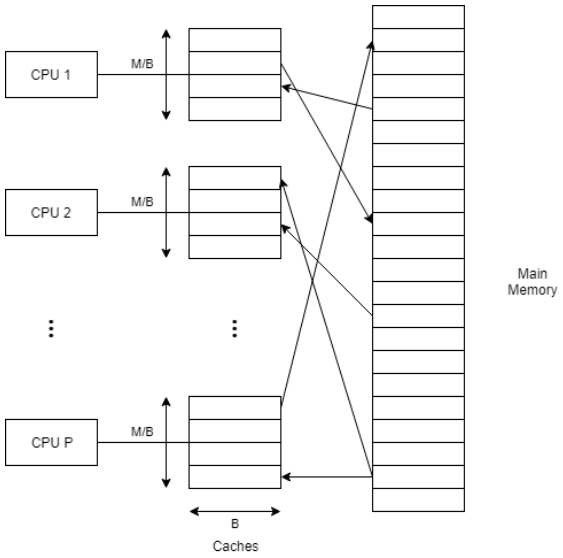


Figure 2.3: PEM model

2.2.6 TMM

Threaded Many-core Memory model of computation was described in 2014 by L. Ma et al. [71], it wants to stay relatively close to the architecture found in GPUs. This model explicitly models the large number of threads per processor and the memory latency to slow memory. It mainly proposes finite constraints on the execution (on the contrary of the model PRAM which does not bound the number of processors) while not targeting a precise hardware or trying to calibrate the performance model.

It is defined by six parameters linked to I/O complexity: the time for a global memory access, the number of processors, the number of elements transfered per block, the size of the local memory attached per group, the number of cores per group and the maximal number of threads per core.

As well as other parameters which are determined by the algorithm and more in relation to time complexity and PRAM notions. We find the total amount of work and per thread, the number of threads per core and the amount of memory space required per each thread. It is one of the many models which exist, but it offers both time and I/O complexity analysis. One can therefore expect stronger optimality criteria, if the algorithm is both optimal in time and in I/Os.

2.2.7 Quantitative model

More recently, people try to model the architecture of the GPU and their relative operations more accurately [20, 65]. The idea was to find out bottlenecks in the implementation through static analysis and to provide tips for the developpers. Highlight operations that are supposed to be slow such that the user can be able to optimize them; or propose metrics to guide compiler optimization passes.

Hence, they go much more further in the concepts which intervene in the modelisation and provide more adhoc comparisons of implementations with the relative timing of each instruction. They represent the code through a work flow graph which feeds the computation and memory pipelines. Warps, SIMD instructions and other factors (like the number of threads, the block size, the cache size...) which may come into account while tuning performances are also taken into consideration. Considerations for which more explanation will be provided in the following section[2.3].

2.3 GPU

Graphics processor units (GPU) were specialized pieces of hardware mainly dedicated to the treatment of images and matrices application. Now, they provide massive parallelism with low power consumption and which can be used for many algorithms. It is also known by its acronym; GPGPU (General-Purpose computing on Graphics Processing Units). The goal of GPU computing is to achieve the highest performance for data-parallel problems through a massive parallel algorithm.

In the early 2000s, processing units were built upon a programmable architecture, permitting to the user to define their own running execution (the so-called “shaders”). People tried to adapt the graphical notions to more classical and scientific problems but it was difficult to realize from a technical point of view. Little by little, the idea to use GPUs thanks to their low power consumption, high parallelism and low cost led people to try to overcome these difficulties. Hence, some first solutions were proposed with a more flexible programming language [36].

However, the way the GPUs work differ fundamentally from the CPUs. Indeed, the structure is inherently massively parallel, with no classical notion of stack (and thus no function stack), and with a different notion of memory layout (*coarse-grained data parallelism*), hence cache. Designing GPU-efficient algorithms is therefore challenging and requires some adaptation. Nevertheless, the architecture provides natural independence in the number of physical processing units available or how execution order of threads is scheduled.

In order to execute a program capable of running on a graphics card, a *kernel*, it is required to write API-compliant code; there are two main API, namely, CUDA and OpenCL (which will disappear to be integrated into Vulkan). To "simplify", these use two different terminologies, we will use mostly the terminology that accompanies the CUDA environment in the rest of this work but in this part, we will present both.

A *thread* or *work-item* represents one hardware thread of execution, a sequence of instructions. But these are, in practice, executed in groups, called *warp* or *subgroup*, typically of size 32 and who can directly communicate with each other. The underlying idea is that we want to execute multiple threads, sharing the same code, at the same time for parallelism. Thus, as soon as we decode an instruction, we are able to execute it on all of them at once. This technique is called Single Instruction Multiple Thread (SIMT).

A *block* or *group* is formed from a set of warps. This unit of parallelism contains, depending on the hardware, up to 2048 threads or 64 warps that can exchange data through shared memory. They are also attached to a *streaming multiprocessor* which plays the equivalent of a core and which schedules the execution of these warps on the computing unit. Finally, as we have several streaming multiprocessors at our disposal, we can run several blocks at a time in parallel. This set of blocks and threads is called *grid* or *workspace* and designates the whole collection of threads to execute. A grid can therefore represent several thousands of threads and is assigned to a kernel.

The main purpose of these notions is to allow higher abstraction for better scaling without considering technical details about scheduling or their quantities. The architecture is also designed to hide latency, whenever one warp has to wait (the result of a memory access or a long operation), another can take its place. A very large amount of threads is therefore beneficial to this type of architecture, since they are very light and can be easily swapped.

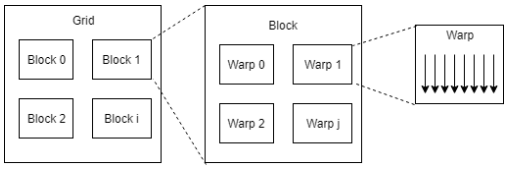


Figure 2.4: Thread hierarchy

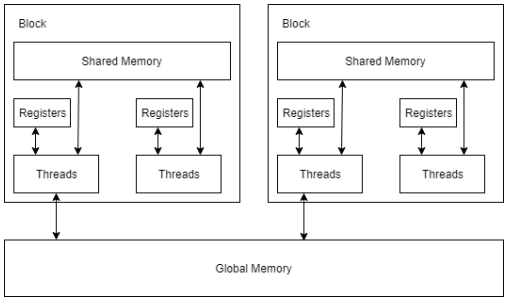


Figure 2.5: Memory hierarchy

Beyond the thread hierarchy, GPU memory is organized into three "main" levels: global, shared/local and private with different sizes and speeds. The global memory is accessible from any thread in the grid, this is the equivalent of our external data source. It typically contains several GB of data and has a bandwidth of over 100GB/s. Shared memory is accessible only by threads being located within a same block, and thus linked to a streaming multiprocessor. It is much smaller, about 32KB, but is incredibly fast, it can peak up to 1300GB/s. Finally, private memory is only accessible within one warp. Its size and speed are not fixed since they can be located in different places, either in registers, shared or global memory, but for communication operations between members of the same warp, we can expect up to 7500GB/s.

Memory is also explicitly managed instead of what we found in an implicit cache

system (like a CPU). This aims to offer the possibility to take advantage of those for our particular problem through caching and prefetching policies which can be specifically designed according to algorithmic or application needs. At the same time, its handling is crucial to obtain the best performance possible [91]. Most of the specific features proposed by those API are only dedicated to this aspect.

The programming work-flow of GPU computing is also different. It defines a host-device relationship between the CPU and GPU. This helps to take advantage of heterogenous computation. The principle is simple, a host program uploads the problem into the device (GPU memory), and then invokes a kernel passing the different parameters of the problem: the number of threads per block as well as the number of blocks, and the arguments of the program. The host program can either work in a synchronous or asynchronous manner, depending if the result from the GPU is currently needed for the next step or not. When the kernel has finished in the GPU, the result data is copied back from device to host [88].

In practice, the essential difference with traditional programming is parallelism management. It is necessary to aim that the processings are the same for all threads. Only, it is not possible in all cases, either because the problem does not lend itself to it, or because the algorithm is not sufficiently adapted. Parallelism can then be controlled more precisely, first by dealing with the problem in blocks and if necessary by warps. By trying each time that there is a minimum of divergence between these different levels of parallelism.

The divergence in the computations is one of the biggest issues. Since an instruction is applied to the entire warp, this implies that each execution flow must be performed one after the other when there is a branching, with a mask indicating whether the results should be conserved or not. If everyone takes the same path, there are no such problems. The more branchings the code has, the slower it will be.

With this crucial aspect, care must be taken to consume the best available resources (registers or memories) in order to allow maximum parallelism and benefit from the locality of the data.

2.3.1 Hardware considerations

Many considerations come into account while implementing algorithms on GPU and which can have significant impact on the performances that will be achieved. We will try to remain as abstract as possible on problems that are highly technical. We must therefore be attentive, when programming, to various concepts. Achieving the maximum possible performance from these tools is often very difficult and it is best to keep different elements in mind. Algorithms can also be designed to rely on these underlying ideas.

Coalesced memory

This notion refers to the ideal scenario of memory accesses where consecutive threads access to consecutive chunks of data. This access pattern helps the data prefetching and, thus, to increase the memory bandwidth, making the implementation more efficient. This also corresponds to the notion of blocks found in the external memory computation model. On the other side, irregular access patterns will lead to reduced performances [23].

Note that the term coalesced is more specifically used when the memory address is aligned on a multiple of the entire cache line and each thread accesses the element associated with its bank (small independant memory cell), the one linked to its identifier

(more practically, this corresponds thus to the case mem[cache line size * k + thread identifier]). We will return to these notions in more details in a forthcoming chapter[4].

Bank conflict

To speed up data access, shared memory is divided into small contiguous memory areas (about 4 bytes) called *bank*. These units guarantee extremely fast accesses at the price of being able to process only one request at a time. So we also have to be cautious about *bank conflicts* which occur when multiple threads attempt to access elements inside the same shared memory bank, those are then serialized leading to poor performances. The hardware splits a memory request with bank conflicts into as many separate conflict-free requests as necessary, decreasing throughput by a factor equal to the number of separate memory requests. The latest hardware versions are capable of broadcasting the data if they are located inside the same word, but accessing different words remains a problem [75].

Thread coarsening

The granularity of the program can really affect the performances. Fine-grained tasks cause greater parallelism and therefore increases the speed-up, however more synchronization or scheduling strategies are also needed, which can have a negative impact on the performance. On the other hand, coarse-grained tasks have lower communication overhead, but often cause a load imbalance. It also allows to reuse same computation among the several tasks in unified manner or to reduce the overhead linked to the latency of the threads scheduling. Thread coarsening consists essentially to reduce the granularity of the algorithm to increase the amount of work per thread. Finding the good granularity is thus a really experimental tuning and the value of such process is usually limited due to the constantly reducing time required to launch a new kernel, schedule new warps or thanks to hardware improvements.

Shared memory and caching

The SIMT multiprocessors have fast private memory shared by all its cooperating threads. And threads are also layered in blocks which can all access to a share and common memory among all of them. Those two levels of cache memory are orders of magnitude faster than the global device memory and thus can be exploided for peak performances.

Padding

Padding is the act to adjust the problem size on the level of the data to get a multiple of the block sizes. There is no more edge corner when considering the last elements of the problems. They fit tightly in the grid and thus require no more special treatment. Of course, the extra dummy data must not affect the original results. With padding, one can avoid putting conditional statements in the kernel that would lead to unnecessary branching [78].

Branching

Branching occurs when there are conditional statements in the kernel, both branches are then executed sequentially. It has a negative impact in performance and should

be avoided whenever possible. The reason why branching occurs is because all threads within a warp execute in a lock-step mode and will run completely in parallel only if they follow the same execution path in the kernel code (SIMT computation); the result is thus discarded for the threads who did not take that path. If any conditional statement breaks the execution into two or more paths, then the paths are executed sequentially. Conditionals can be safely used if one can guarantee that the program will follow the same execution path for a whole warp. Additionally, we can use some logical operators which cause no branching and can be used to make disappear simple conditionals [61].

Memory use vs parallelism

Finally, we said that the architecture conceptually allows an abstraction between the number of blocks and threads we launch and the real hardware capabilities. However, in practice, this is not totally the case, resources are scarce. The shared memory is very small (32KB) and can be shared by thousand threads; there are also only 256 registers but these condition the number of warps executable simultaneously on the same streaming multiprocessor, if we want to have in practice a thousand threads active, we should consume only 32 of them. Moreover, the more active threads there are, the more memory requests will be made and with them the number of cache evictions. It is thus a real trade-off which must be carried out between the number of threads and blocks which one wants to execute and the resources which one can use.

2.4 Design

Designing an algorithm is not a trivial task; even more, in case of parallelism. There are some strategies to help us to create efficient and parallel algorithms but there are also some considerations to take into account. In 1995, Foster [52] identified four key components to design many algorithms.

2.4.1 Partitioning

The first idea which comes to mind when trying to solve some parallel problem is to find out how to split it into several subproblems. The goal is thus to identify whether it is possible to partition either at the level of the data or at the level of the tasks. If the subproblems can work on different data at the same time, we tend to say that there is data-parallelism. On the other hand, when the tasks do not conflict each other, we use the notion of task-parallelism. Those two big families are represented through a large variety of problems. But, data-parallelism seems more in adequation with data-driven programs, like physics simulation or data science. Whereas, task-parallelism arises more often in algorithmetical problems such as graph traversals or flow notions.

2.4.2 Communication

When the method of partitioning has been determined, we still need to consider the communication. This is often divided into two categories: the local and the global communication. Local communation appears when subproblems can easily communicate with their neighbours to continue their work. Global communication, on the opposite, involves broadcast to be able to make the next step. In this phase, all the problems linked to communication and concurrency need to be handled through different and classical

methods as critical sections or barriers to ensure consistency among the workers and reliability in the results.

2.4.3 Agglomeration

Even though, we have nice partitioning with small communication requirements, there may be some imbalances in the resulting computations and it would be nice if we could rebalance the problem while running it. This aspect related to the granularity of the problem. Fine-grained divide the problem into a huge number of jobs but with a lot of communication; coarse-grained perform less jobs but with larger tasks. This is a trade-off between parallelism and communication. Agglomeration seeks thus to find the best compromise of granularity while taking into account the final implementation.

2.4.4 Mapping

With the recent advances in the hardware and algorithmic considerations, it may be interesting to consider how the agglomerations will be mapped to computational processors. The way those are distributed among the cores and their relative order can have significant impact on the performances of the final implementation of the algorithm. We want, of course, to have a direct mapping or, at least, the simplest method to avoid higher hardware complexity and overhead. But, this may lead to counter techniques which enhance balancing. This category really became prominent with distributed memory and GPU computing where exploiting hardware may lead to signifcant improvements.

It is therefore interesting to offer this aspect to developers so that they can choose what suits them best and this results in the consideration or not of notions of warps, blocks or grids in algorithms. Knowing which primitives we are working on allows simplifications that can make the code much more efficient.

2.4.5 Brent's scheduling principle

When we speak of parallelism, the notion of Brent's Scheduling Principle is often mentioned, it synthesizes a logical observation that appears when we study parallel algorithms in a more algorithmic and complexity-related framework.

Some algorithms are said to be time optimal if the number of steps in parallel program is equal to the number of steps in the best sequential algorithm. One should remark that there are two main way to associate the complexity in parallel algorithms:

- Either, we consider the time complexity of a parallel algorithm as the number of steps taken by each processor, denoted by $S(N)$ and called *span*.
- Either, we study the complexity of the whole task as the total number of operations the algorithm performed by every processors, denoted by $W(N)$ and called *work*.

If a parallel algorithm runs in $S(N)$ and uses a total of $W(N)$ operations, it can be simulated on a P-processor PRAM in no more than $T_C(N, P) = W(N)/P + S(N)$ parallel steps. This is known as Brent's Scheduling Principle [56] but, this principle does not apply to PEM model of computation. Since, if we apply the same idea with a round-robin simulation of the processors, this would lead, in worst-case, to $\Theta(\frac{PM}{B})$ factor due to the fact that we need to load the entire memory of each processor to simulate it. There is no equivalent theorem in this framework, the closest states that a PRAM algorithm in time $T(N)$ of N processors and $O(N)$ space can be simulated in $O(T(N)\ \mathrm{sort}(N))$ I/Os [41].

The whole point of Brent's Scheduling Principle is that it provides equivalence between parallel and sequential programs. This mainly means that if an algorithm has a better complexity in the parallel framework, then there exists a sequential program equivalent with the same work complexity since we could easily simulate the parallel one.

Chapter 3

PEM Algorithms

In this part, we will present a review of different and classical algorithms in the PEM model of computation. We will start by presenting the general purpose algorithms, those that serve as building blocks for many other problems. We will follow by those who are related to graphs and their inherent complexity, then will come those related to more geometric problems. We will then present the algorithms linked to the matrices and finally we will come back to sorting ones due to their capital importance.

We would like to emphasize that the PEM computation model focuses on the number of blocks transferred between a supposedly large and slow external memory and a small and fast internal one since the idea is that making a memory request is longer than making a computation. The parallel aspect is translated by several processors, each having their own internal memory and being able to communicate between them through the external memory. Each cache is of size M, is partitioned in blocks of size B and is exclusive to each processor, i.e., processors cannot access other processors' caches. To perform any operation on the data, a processor must have the data in its own cache and the data is transferred between the main memory and the cache in blocks of size B.

The parameters are as follows:

N is the problem size.

M is the internal memory size.

B is the block transfer size.

P is the number of processors.

The model complexity measures the number of parallel block transfers between the main memory and the cache by the processors. For instance, an algorithm reading one block (or different) with each of the P processors simultaneously from the main memory would have an I/O complexity of $O(1)$ and not $O(P)$.

Many of the following algorithms will have the assumption that $M \geq B^{O(1)}$ (or its weaker version $M = \Omega(B^{1+\epsilon})$). This is called the tall-cache assumption, G.S. Brodal and R. Fagerberg prove that without the tall-cache assumption, sorting cannot be performed optimally and permuting elements is not cache-oblivious even under this property [35].

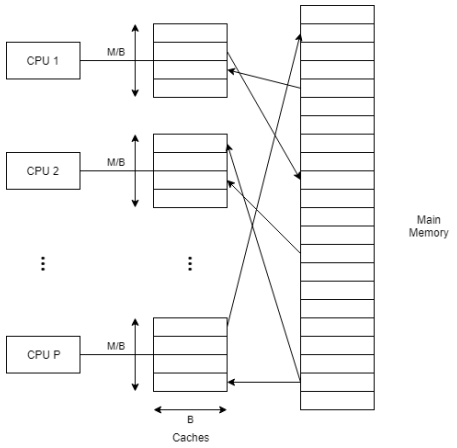


Figure 3.1: PEM model

3.1 General purpose algorithms

We will start by introducing different proven properties for general purpose algorithms. These typically serve as primitives for many other algorithms and their study therefore provides a solid basis for dealing with more complex problems. It is therefore all the more important to have efficient algorithms for them.

3.1.1 Scanning

Scanning is the basic operation which consists simply in consulting each of the elements in a contiguous collection in memory. Naturally, its complexity is bounded by $O(\frac{N}{PB})$. The same applies if we run several scans in parallel, like reversing the input.

3.1.2 Searching

Searching for an item in a sorted collection benefits relatively little from parallelism. Indeed, we can use an argument related to information theory, if there are N elements, our element can be found in N positions, so we need $\Omega(\log N)$ bits of information. And when we read a block, we get $\Omega(\log B)$ bits of information at once, since each block read reveals atomically where the query element fits among those B elements. In our computation model, we are able to read P block simultaneously. Unfortunately, these are not entirely independant since they form an order and many of them will not add more information, this leads to $\Omega(\frac{\log N}{\log PB})$.

3.1.3 Permutation

Permuting data consists to rearrange all the elements into some sequence or ordre. In the PEM model, it takes asymptotically $\text{perm}_P(N, M, B) = \Theta(\min(\frac{N}{P}, \frac{N}{PB} \log_d \frac{N}{B}))$ where d is $\max(2, \min(\frac{M}{B}, \frac{N}{PB}))$ [58]. This result was achieved through extending the problem of bit-matrix-multiply/complement (BMMC) to PEM model of computation. Those BMMC permutations [45] map a source index to a target index by an affine transformation over GF(2), where the source and target indices are treated as bit vectors. This class of permutations appear in several other theoretical problems as sorting, matrix transposition or Fast Fourier Transform.

But, the same argument can be done with a combinatorial extension based on the same argument that A. Aggarwal and J. S. Vitter [2] or with the red blue pebble game of J. W. Hong and H. T. Kung [67]. Other demonstrations are also presented by G. Greiner [58], notably based on a potential function or on program traces.

The first term represents the case where we simply apply the PRAM algorithm ignoring blocks, each time we want to place an element, we may ask to transfer a new block; the second term may correspond to sorting the elements according to some indices.

Hong and Kung

They present a method to obtain lower bounds on the I/O complexity which is based on the computation graph of the algorithm. The computation graph is a directed acyclic graph (DAG) where each node v corresponds to either an input (if the node v does not have any ingoing edges) or either to a computation operation and its related result. An edge represents the depedency of operands.

The idea is to play a game, so called "red-blue pebble", on the computation graph based on some rules. A red pebble represents memory in cache and blue in external. We can put a blue on top of a red or vice-versa which represents a data transfer. We need to have all red pebbles on ingoing edges to compute a new red pebble, we can remove pebbles at any time and we can have at most M red pebbles on the graph. The goal of the game consists to cover some output nodes with blue pebbles according to some blue distribution at start.

They proved that a partitioning of the computation graph yields to a lower bound on the number of I/O operations. Any partition set may have a dominating set (nodes from which there exists a path from the input nodes to this set, the execution flow) of size at most $2M$. And there exists at most $\mathcal{P}(2M)$[1] sets and thus the minimal number of I/Os is at most $M(\mathcal{P}(2M) - 1)$. You can convince yourself that the number of transfers is directly proportional to the total size of the memory and that the number of processors is orthogonal to these notions.

Aggarwal and Vitter

The idea is to bound the maximum number of permutations that can be produced by at most T I/Os, we thus search for the worst-case number of transfers required to perform $N!$ permutations. This result is independent of the number of processors used and is directly proportional since reading two times the same block does not help the complexity. Let's try to estimate the minimum number of memory transfers required. We want to get all permutations of the elements, so $N!$. Only, once we have a block, we can make all the permutations in it without cost and we know that there are N/B blocks, which results in the number of:

$$\frac{N!}{B!^{\frac{N}{B}}}$$

Now, we have to consider what happens when we consider an input or an output of the problem. Each time a new element is considered, the same process must be performed. This action must therefore be repeated N times. We also have to remember that once we get a block, we'll be able to generate its permutation. So, we have to count the number of way to place the block in the cache, which corresponds to the output order of the elements $\binom{M}{B}$ or which block has been read. We arrive at the relation:

[1] Power set

$$(N\binom{M}{B})^T \geq \frac{N!}{B!^{\frac{N}{B}}}$$

Now, taking the logarithm on both side and applying stirling formula, this yields to:

$$T = \Omega(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$$

3.1.4 Gather and scatter

Gather consists to group up the information coming from different memories in a tree-like fashion into a single unit, this leads to $O(\log\ P)$. Of course, we can do the opposite work and spread out one to several memories for the same complexity, this inverse operation is called scatter. We can also remark three different points. First, we can serialize those procedures, second, we need to know all the participants which would imply some order among them and, third, if we allow concurrent reads, scatter can be in $O(1)$.

3.1.5 All-prefix-sum

One of the most classical algorithm in the parallel world is the all-prefix-sum (sometimes called scan). It establishes that:
Given an ordered set A of N elements, the prefix-sum operation returns an ordered set B of N elements, such that $B[i] = \sum_{j=0}^{i} A[j], 0 \leqslant i < N$.

It is a primitive which appears in certain algorithms, like lexically compare strings, polynomial evaluation or radix sort, since it only requires a binary associative operator [31]. And, under the assumption that the input is contiguous in memory, the problem can be solved optimally in $\Theta(\frac{N}{PB} + \log P)$ in PEM model.

The algorithm consists in four phases: we start to compute adjacent elements as $\frac{N}{P}$ parallel sums, we then "up-sweep", then "down-sweep" and we redistribute the results. The sweep phases contribute to the $O(\log P)$ term [85].

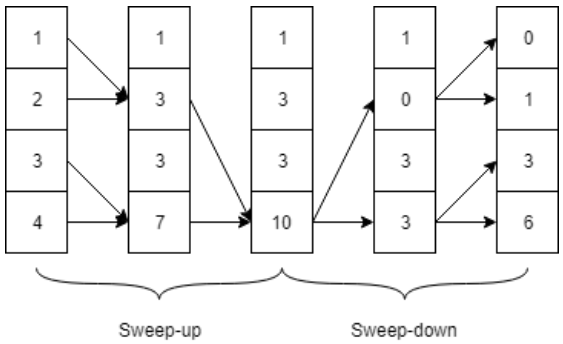


Figure 3.2: G. E. Belloch algorithm - exclusive scan

3.1.6 Multiway partitioning

The multiway partitioning consists to split an unsorted set of N items in d bins such that all the elements in the ith bucket are smaller than the dth pivot and greater than the $d-1$th pivot given $d-1$ sorted pivots in increasing order.

This problem has most of its applications in GPU sorting [40]. Arge et al. proposes a more in depth analysis of the complexity in comparison to the original paper and gets: $O(\frac{N}{PB} + \lceil \frac{d}{B} \rceil \log P + d \log B)$.

The idea is a quite natural. We divide the initial vector in subelements, where we launch our processes. Each processor counts the number of elements smaller/greater than its relative pivot. Then, we compute a general scan to know at which index we will be able to write without having conflicts, elements greater than one pivot must be placed after all the elements which are smaller.

3.1.7 Selection

Given an unordered set of size N and an integer k (with $1 \leq k \leq N$). The selection problem consists to find an item such that it is larger than exactly $k-1$ elements.

The idea is quite similar to the kth sort. We perform a multiway partitioning on the inputs and recurse on the targeted subset. This leads to a complexity in $O(\frac{N}{PB} + \log PB \ \log \frac{N}{P})$; note that it considers an optimal algorithm for sorting in PRAM model when there are less elements than the number of processors [12].

3.2 Graph algorithms

In this part, we will sum up the results for the graph algorithms. These are generally more complex and the existence of massively parallel and optimal algorithms is not always considered as possible. Indeed, they often require a great deal of dependence on results obtained at previous stages. And they mostly have task-parallelism, which is not the most suitable for graphics cards.

3.2.1 List ranking

The list ranking problem consists to: Given a list of N elements, each one identified by a unique address (identifier) and pointing to its successor. Let define the rank as the number of items between the head of the list and the item. The problem resides in finding the ranks of all the elements in the list.

The first parallel algorithm was proposed by J. C. Wyllie [99], it simply consists in a path-doubling strategy, hence it was $O(\log N)$. But, this would lead to a work complexity of $O(N \log N)$ instead of $O(N)$ for the sequential version. R. J. Anderson and G. L. Miller proposes an enhancement by splicing out some elements from the list. This helps to subdivide the list into sublists on which they will apply the Wyllie algorithm. Then, they will need to reconstruct the general order based on the split elements [8].

But, the complexity of list ranking is in $\Theta(\frac{N}{PB} \log_{\frac{M}{B}} \frac{N}{B})$ in PEM model of computation and is thus as complex as sorting a collection due to the distributed nature of the caches and the difficulty to transfer information and its tight link with the permutation problem [66].

The main idea consists to create independent sets via 3-coloring ($\frac{N}{3}$ size), through the forward/backward edges algorithm or deterministic coin tossing ($\frac{N}{4}$ size), solve the subproblems and recompose the solution [41]. This operation of decomposition and reconstruction of the list is called brigde-out/in, it consists in making a copy of the original list, sort the original by successor identifier, scanning through original and the copy together to obtain successor information and sort back the modified original list by identifier [66].

3-coloring can be done as followed, we color alternatively forward pointers in red and blue, and backwards, in green and blue; each node will have one color at the exception of head/tail who have two, it remains to color them as the head of the list. In practice, we need to put the nodes in a priority queue based on the index, because we can't access any element in any order, this would result in too many transfers, and a priority queue can have a complexity of $O(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B})$ [10].

Behind this curious problem, there is a fundamental operation which has many applications: broadcasting an information.

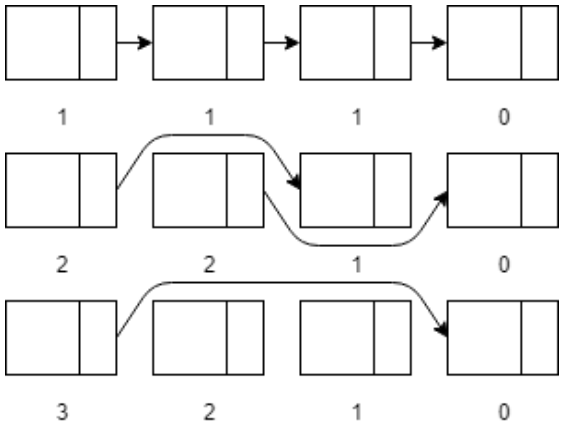


Figure 3.3: List ranking

3.2.2 Euler tour

R. Tarjan and U. Vishkin introduced the Euler tour technique [90]; the Euler tour is defined as being able from one vertex of the graph to visit every edges exactly twice, in both direction. Hence, it can be easily mapped to a list ranking problem and this technique was used to solve many theoretical problems leading to several and various results to trees.

The bottom idea is that we can use the prefix sum algorithm through the Euler tour technique. This allows us to represent our tree structure as a flat collections of elements and apply our more advanced operations on all the elements without worrying on the accesses.

3.2.3 Tree contraction

Tree contraction was introduced by G .L. Miller and J. H. Reif in 1989 [76] and has been used to design many and efficient parallel algorithms due to the possibility to represent the problems as a tree. The algorithm requires two primary operations:

- Rake which removes all the leafs of a tree (merging childless nodes together).
- Compress consists to identify every vertex v_i with v_{i+1} when i is odd and v_i has only one child which is not leaf.

Both operations are successively repeated until the tree is reduced to one and unique node. The maximal number of iterations needed is bounded by $O(\log N)$. This helps us

to work with any tree and guarantee $O(\log N)$ operations even with degenerated cases like linked list.

Many algorithms are based on this algorithm, the *Arithmetic Expression Evaluation*, the *Tree isomorphism* or to find the *3-connected components* of a graph [77]. Those present a complexity in $O(\text{sort}_P(N))$ with $P \leq \frac{N}{B^2 \log B}$.

3.2.4 Lowest common ancestors

The lowest common ancestor (LCA) of two nodes v and w in a tree T is the lowest (i.e. deepest) node that has both v and w as descendants.

D. Harel and R. E. Tarjan showed that LCA queries can be answered in constant time after only linear preprocessing of the tree through *Heavy path decomposition* [59] but that data structure may be hard to implement. Arge et al., proposed to consider a simple $(\frac{M}{B})$-ary search tree which guarantees $O(\log_{\frac{M}{B}} \frac{N}{B})$ levels. The K queries are treated after sorting the items, so that all of the queries can be answered by scanning the search tree a constant number of times through reduction of the problem to range minimum query (RMQ). The complexity is thus expressed as $O((1 + \frac{K}{N})\text{sort}_p(N))$ I/Os.

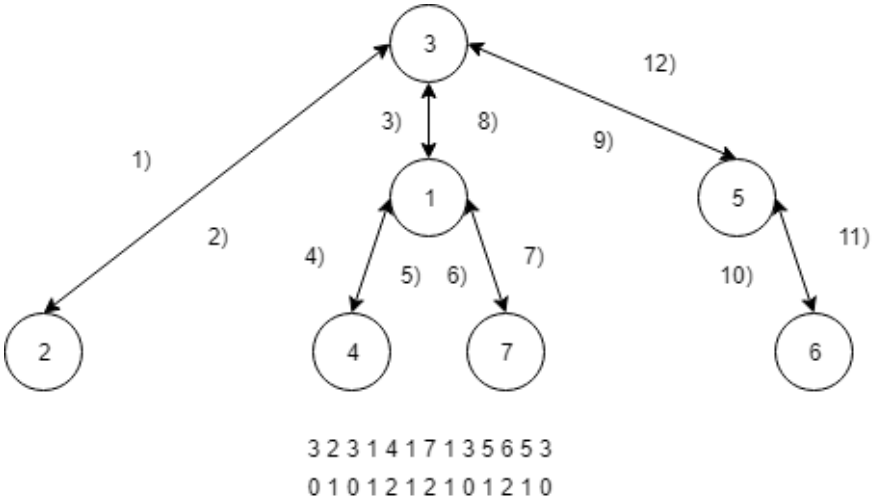


Figure 3.4: LCA - RMQ: Euler tour

3.2.5 Connected and biconnected components, ear decomposition and minimum spanning tree

Arge et al. [13] propose also many theoretical results in links to these problems:

Finding a minimum spanning tree, the connected components, the biconnected components and performing the ear decompositon can all be solved on the undirected connected graphs $G = (V, E)$ can be solved in $O(\text{sort}_p(|V|) + \text{sort}_p(|E|) \log(\frac{|V|}{PB})$ I/Os in the PEM model using up to $P \leq \frac{|V|+|E|}{B^2 \log^2 B}$ processors.

If the graph is sparse and closed under contraction, the connected components, the minimum spanning tree (if G is connected), the biconnected components and ear decomposition (if G is biconnected) can be computed in $O(\text{sort}_p(|V|))$ I/Os with $P \leq \frac{|V|+|E|}{B^2 \log^2 B}$.

Those results are extension of the work of Chiang et al. [41] in external memory model. They relied on many other algorithms which would be out of the scope. These

algorithms are often based on the Minimum Spanning Forest problem, where we divide the graphs among the processors. They all define a priority queue based on the weights of the concerned nodes and apply a Prim-like algorithm [11]. N. Zeh proposes a work in which he gathers many results with demonstrations and detailed explanations for the sequential I/O algorithms [101].

3.3 Geometric problems

A classical set of problems and algorithms are the geometric ones. They appear in a wide range of problems, often in somewhat concealed forms, but offer frequently elegant solutions. Many of the sequential algorithms are based on divide-and-conquer and lead in a relatively straightforward manner to efficient parallel algorithms.

3.3.1 Interval stabbing counting and 1-D range counting

Given I, a set of intervals, and S, a set of points on the real line, with $|I| + |S| = N$. The *interval stabbing counting* problem consists to compute the number of intervals containing each point of S. The *1-D range counting problem*, is the opposite, to compute the number of points contained in each interval.

These two problems can be solved in $O(\text{sort}_p(N))$ I/Os. But, in the case where the inputs are already sorted by their value for S and by the end of the interval for I, interval stabbing counting is in $O(\frac{N}{PB})$ and the 1-D range counting is bounded by $O(\text{sort}_p(|I|) + \frac{|S|}{PB})$.

The idea to count the number of intervals containing a point is to assign a weight of 1 to every left interval endpoint, a weight of -1 to every right one and 0 to every point in S. And then apply a prefix sum in $O(\frac{N}{PB})$.

On the other hand, compute the number of points within the intervals is the difference of the prefix sums of its endpoints after assigning a weight of 1 to every point and 0 to every interval endpoint. The two prefix sums are trivial, but the difference is not. We must first extract the set of interval endpoints from the point list using a compaction operation and sort the resulting list to store the endpoints of each interval consecutively; this operation takes $O(\text{sort}_p(|I|) + \frac{|S|}{PB})$. They also came up with an optimal algorithm in $O(\frac{(N+K)}{PB} + \log P)$ when $P \leq min(\frac{N}{B \log^2 N}, \frac{N}{B^2})$ and where K are the queries but which is way more complex [5].

3.3.2 2-D weighted dominance counting

The problem is posed as follows: Given a set of points ($q \in S$), each of them associated with a weight, the goal is to compute the total weight of all points 2-dominated by each point in S. The 2-D dominance is defined as such: given two points in the plane $q_1 = (x_1, y_1)$ and $q_2 = (x_2, y_2)$, q_1 1-dominates q_2 if $y_1 \geq y_2$ and 2-dominates if it both 1-dominates and $x_1 \geq x_2$.

This problem looks quite surprising, but some others reduced to it [18]. The problem, by it-self, can be solved using $O(\text{sort}_p(N))$ I/Os in the PEM model.

The first step of the algorithm consists to sort the points along their x-coordinates and partition this collection into vertical slabs σ_i, each containing $\frac{N}{P}$ points. Then, in parallel, we sort the slabs along the y-axis, we get a new list $U(\sigma_i)$ and we associate to each one two new weights: $W^1_{\sigma_i}(q)$ and $W^2_{\sigma_i}(q)$ which are the total weights of the

points within σ_i that q 1- and 2-dominates, respectively. It then remains to group up the information. The obtained lists can be merged together using a d-way cascading merge procedure where we have a d-ary tree where the leaves are the σ_i. At each tree node v, we compute a new y-sorted list $U(v)$ which is the result of the subnodes. While we are doing this operation, we can also compute the new weights $W_v^1(q)$ and $W_v^2(q)$ since those can be expressed as the sum of their predecessors in the merged lists. At the end, the root r of the tree has $U(r) = S$ and $W_r^2(q)$ is the total weight of the dominated points [4].

3.3.3 Distribution sweeping

In computational geometry, a sweep line or plane sweep is an algorithmic tool which is used to solve many problems. One can imagine that a line is swept accross the plane and stops at some points. The idea is to construct the solution bit by bit while we meet new points to finally produce our solution. This concept has been extended to parallel problems in 1985 and it is more well-knowed as distributed sweeping [18]. It can be rougly expressed as subdividing recursively the plane into slabs such that they contain about the same number of elements, with $min(\sqrt{\frac{N}{P}}, \frac{M}{B})$. But this framework may be hard to implement due to parallelism of line sweeping and due to the expected good load balancing if we consider output sensitive algorithms [3].

D. Ajwani, N. Sitchinava and N. Zeh propose an optimal algorithm to solve three problems: *Orthogonal line segment intersection*, *Batched orthogonal range reporting* and *Rectangle intersection reporting* [5]. The optimal complexity achieved is in $O(\text{sort}_p(N) + \frac{K}{PB})$ but is really complex. It can be easier to duplicate horizontal segments which cross different partitions and then apply the standard algorithm, this would lead to a $O(\text{sort}_p(N + K))$ where K is the number of duplicates segments.

To realize such results, they extended the external memory data structure called *Buffer Tree* which provide amortized complexity for queries and which can be used as a priority queue [9]. This structure can be seen a specialization of the B-trees, the classical balanced trees, as R-trees are for range queries. They assign $\Theta(\frac{1}{PB} \log_{\frac{M}{B}} \frac{N}{B})$ credits to each operations and modify the structure to maintain the invariant when the buffer is full, it is then emptied and the sub-nodes are impacted [87].

They also propose a solution to the problems of: *Lower envelope of a set of non-intersecting 2-D line segments*, *Convex hull of a 2-D point set*, and *maxima of a 3-D point set*. All of these are in $O(\text{sort}_p(N))$, provided $P \leq \frac{N}{B^2}$ and $M = B^{O(1)}$. Convex hull simply consists to subdivide the problem into pair wise convex hull problems through Graham Scan technique and then compose the solution finding the tangent [19]. The other problems can be solved using a technique close to 2-D weighted dominance through clever change in the labels [17].

3.4 Matrix algorithms

While considering matrix algorithms, it is worth to consider the way the elements are laid out in memory. Sparse matrices are designed such that only non-zero entries are stored as a triple of value and row, column. For a dense matrices, we usually distinguish the three following layouts:

- Column major layout: The records are ordered by column first, and by row index within a column.

- Row major layout: The records are order row-wise first, and column-wise within a row.
- Recursive layout or zig-zag: Such layouts are often helpful to construct efficient cache-oblivious algorithm. Many applications involve (recursive) space filling curves to define the ordering of records.

3.4.1 Matrix compaction and transposition

The problem consists to transpose a matrix A of total size $N_x \times N_y = N$, composed as P rows A_i divided in d sub-arrays, such that it procudes a new matrix $A^{'}$, with d rows and P subdivisions. The segmentation is given through a matrix M whose dimensions are $P \times d$ where $M[i,j]$ gives the size of the subarray.

A. Aggarwal and J. S. Vitter point out that transposition is a special case of permutation. Indeed, one can imagine that the matrix is broken down into several subgroups that will be used to create the final result. And then, it is necessary to merge the results through a fusion procedure, with the intuition that the same subgroups remain together. Hence a natural bound, for dense matrix, of $\Omega(\frac{N}{PB} \log_d \min(B, N_x, N_y, \frac{N}{B}))$ with $d = \max(2, \min(\frac{M}{B}, \frac{N}{PB}))$ [58]. Cantazaro et al. propose another solution to this problem which has the advantage to be bank-conflict free [38].

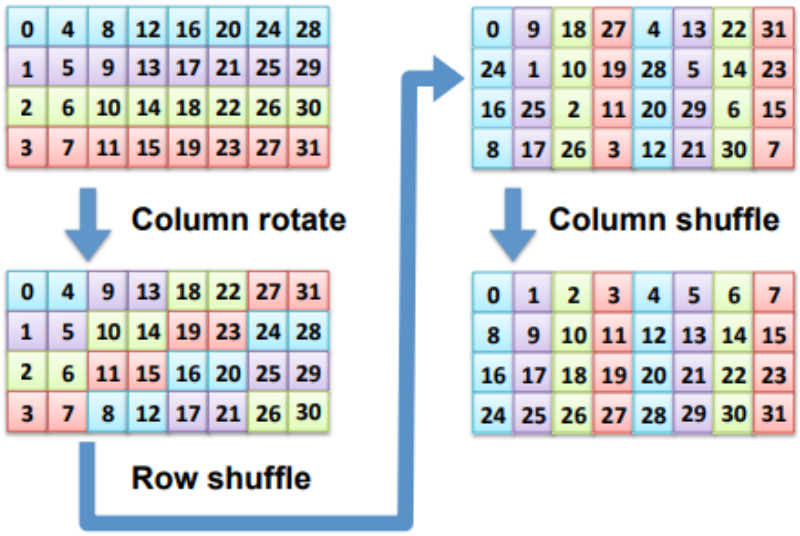


Figure 3.5: Catanzaro transposition - image extracted from Catanzaro et al. [79], doi:10.1145/2692916.2555253

3.4.2 Matrix-vector multiplication

Performing a *Sparse matrix-vector multiplication* can be bound in $\Omega(\frac{wH}{PB})$ with H the number of non-empty entries and w the number of vectors to multiply. And computing the bilinear form adds a factor of $O(\log \min(\frac{\sqrt{N}}{B}, \frac{P}{w}))$ since we can compute the partial scalar product when we are writing the resulting vector of the matrix multiplication and then gather the result [58]. This extends the work of Bendal et al. [26] to parallel

machines. Those sparse notions have a lot of application since they can have deep connection with graph problems [100].

3.4.3 Matrix multiplication

If we apply the standard matrix multiplication algorithm for two dense matrices, with the first matrix in row-major order and the second in column-major, for each element, we would have to do two scans, which would end up in $O(\frac{N^3}{PB})$ which is not optimal. Indeed, it is feasible to achieve $\Theta(\frac{N^3}{BP\sqrt{M}})$ [22] for the classical algorithm, this coincides with one of the first result achieved by Hong and Kung. One should remark that multiple rows and columns are read several times, leading to inefficiency. The solution consists to store the elements of the matrices recursively, each time subdividing the 4 corners one after the other with a layout like $\text{layout}(Z) = \text{layout}(Z_{11})\text{layout}(Z_{12})...\text{layout}(Z_{22})$.

The complexity can be described by this system:

$$\begin{cases} MT(N) = 8MT(\frac{N}{2}) + O(\frac{N^2}{B}) \\ MT(\sqrt{\frac{M}{3}}) = O(\frac{M}{B}) \end{cases}$$

Indeed, each corner of the resulting matrix is obtained as the sum of the product of two sub-matrices, we have 4 corners with 2 products. The sums are independent of the recursion and cost $O(\frac{N^2}{B})$ since we scan along X and Y. Finally, when the submatrices can be entirely contained in the cache, and as they are stored continuously, the worst case is to go through all these blocks, $O(\frac{M}{B})$. Now, due to master theorem, we need to count the number of leaves, since the sum costs more than the recursion, there are thus $8^{\log N/\sqrt{M/3}} = (N/\sqrt{M/3})^3$ leaves. The total cost is finally:

$$(\frac{N}{\sqrt{M/3}})^3 O(\frac{M}{B}) = \Theta(\frac{N^3}{M^{3/2}})O(\frac{M}{B}) = \Theta(\frac{N^3}{B\sqrt{M}})$$

Sparse-dense matrix multiplication has a complexity dependent of the number of processors that we may offer, $\Omega(\frac{H\sqrt{N_z}}{PB\sqrt{M}})$ with $P \leq HN_z/M^{\frac{3}{2}}$. Sparse-Sparse multiplication is still an open question, but we know that the complexity is bound by $\Omega(\frac{H_1H_2N}{PB}\log\frac{N}{\min(H_1,H_2)B)})$ where H_1 (respectivelly H_2) is the average number of non-zero elements per column. Ballard et al. [21] regroup the results for the classical matrix decomposition (SVD, LU, ...).

3.5 Sort

Sorting is a building block for many algorithms. Indeed, countless of those take for precondition a sorted collection. Hence, it is important to try to achieve the best performances possible to make benefit all the others. It is also intented to design them such that they benefit the most from the hardware. GPUs are thus good candidates to gain efficiency since we can perform many operations at the same time, like comparisons.

One of the first results achieved for the PEM model of computation was the complexity bound of this problem which are inbetween $\Omega(\min(\frac{N}{P}, \frac{N}{PB}\log_{\frac{M}{B}}\frac{N}{B}) + \log\frac{N}{B})$ and $O(\frac{N}{PB}\log_{\frac{M}{B}}\frac{N}{B})$ with $P \leq \frac{N}{B^2}$ and $M = B^{O(1)}$ [12].

3.5.1 Sorting networks

One classical notion when talking about parallel sorting is the use of sorting networks, with the classical Batcher's odd-even mergesort or bitonic mergesort [24]. But, even though, these sorting networks are not optimal, they provide good runtimes since they can be efficiently hardware-implemented. Hence, they grant low constant factors, good locality and no bank conflicts but they can not be used practically to sort large arrays. A certain type of sorting network is also at the origin of the complexity of permutation in External Memory model [67].

3.5.2 Shearsort

Shearsort can be viewed as an analogue to the sorting networks but viewing the problem as a matrix. Elements are first sorted by rows, alternating between ascending and decreasing. Then, follow by sorting column-wise and repeat those two steps up to $\Theta(\log(N))$ times [86]. It also benefits from being bank-conflict free.

3.5.3 Radix sort

When we are interested in the task of sorting elements, we have to be aware that we intend most of the time comparison-based. But there are some sort algorithms which do not rely on that and use other kind of information. One of the most well known is the radix sort for integer-like elements. It consists to use the propriety of the elements to directly sort them, hence it only keeps to do it in a clever way to avoid scattering and scanning [83].

3.5.4 Distribution sort

The distribution sort consists to partition the unsorted set of N elements into d buckets, each being recursively sorted and concatenated to obtain the final result, it can be viewed as a generalization of the quicksort. The algorithm, by it-self, randomly sample elements of the data to get $\sqrt{d}$ pivots, partition the data and calls it-self on those subsets. When there are not enough data left to recurse one again, we perform a simple sort. One heuristic is used to determine the pivots, in link to the theoretical results of A. Aggarwal and J.S. Vitter [2], it consists to sample a part of the elements that we want and construct on them our pivot distribution.

Its complexity is in $O(\frac{N}{PB}\log_{\frac{M}{B}}\frac{N}{B})$ when the number of elements $N \geq PB$ and $M \geq B^c$ for some $c > 1$. Otherwise, its expression is way more complex [12].

3.5.5 Multiway merge sort

Casanova et al. proposed in 2017, a new algorithm called: *Multiway Mergesort* (MMS) [37]. It is the first sorting algorithm for the GPU that is asymptotically optimal in terms of global memory accesses and without any bank conflicts in shared memory. This algorithm intervenes while trying to solve two problems. Indeed, previous algorithms benefited from coalesced memory accesses but they were not trying to reduce the total number of accesses and were less cautious with the banks which may appear in the merging phase.

The proposed algorithm is based on the classical notion of merge sort, which can be parallelised efficiently using partitioning [83]. Instead of two sublists merged together

multiple ones are treated at the same time. To allow parallelism, it also uses a notion of partitioning which splits some sorted list by the median pivot. Then, it tries to merge efficiently avoiding bank conflicts and trying to maximize memory coalescing. It remains to sort basis elements which will be source of the recursion for the multiway merging algorithm, this is achieved by a shearsort [86].

Let us return a little on the complexity of the classic merge sort, with the fusion of two lists at once. The complexity is described by this system:

$$\begin{cases} MT(N) = 2MT(\frac{N}{2}) + O(\frac{N}{B}) \\ MT(M) = O(\frac{M}{B}) \end{cases}$$

Since we recurse on both sides and, to merge two lists, we need to perform one scan on two arrays, so $O(\frac{N}{B})$. Now, we remark that the height of the recursion is bounded by $O(\log N - \log M) = O(\log \frac{N}{M})$ since once we reach the cache size, everything becomes essentially free. Finally, the amount of work is the same for each level of the tree, hence $O(\frac{N}{B} \log \frac{N}{M})$. But this is not optimal, since we don't use efficiently all the cache available to us when merging the lists, we could merge up to $\frac{M}{B}$ lists at the same time. The coefficient "2" becomes $\frac{M}{B}$ and we finally reach the optimal bound of $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$. Now, to adapt the result to parallelism, we just need to find out a way to split the data and merge the lists in $O(\frac{N}{PB})$. But merging lists is not as trivial as it sounds [57].

Chapter 4

Memory accesses

We will now address a completely different topic and return to our main concern, which is to adapt a data structure to the graphics card. We wanted to start by studying memory behavior. Indeed, this is a crucial element when developing a data structure where we try to minimize the number of steps to find an element as well as the number of cells or cache lines that we must consult to guide our queries. As a matter of fact, the computation model on GPU proposes minimum units of parallelism that group the elements by warp, a same instruction is executed by, generally, 32 threads at the same time and the same goes for the memory access instructions. We can therefore study the variety of accesses that the whole warp can perform at the same time.

4.1 Type of accesses

We wanted to study the behavior of 4 main different types of access. First, how long it took to search for one and only one item. This will allow us to define our base case, and compare it with other accesses, get the minimal running time to issue those actions. Secondly, we wanted to know what the impact was if all the elements of the same warp tried to access the same element. Indeed, Cuda indicates us that the accesses will then be serialized. Third, Cuda recommends accessing coalesced or at least adjacent addresses, so threads will all access contiguous elements in memory but not necessarly coalesced [60]. Finally, the question arose for random accesses, if each thread accessed a different and non-contiguous element in memory which would represent the worst scenario among these. The other possible types of access are of less interest to us since they seem to be placed in intermediate positions and not sufficiently relevant.

4.2 Experiment

We designed a relatively simple experiment. We start by reserving a large amount of memory, 2GB. Then, each thread accesses its own element according to its defined access policy. 16 million accesses (2^{24}) are made in order to saturate the memory and avoid cache phenomena. Those positions are determined by a pseudo-random function (hash function) based on the current step in the algorithm and an initial random offset. A very simple operation is then performed with the loaded element, an incrementation. The experiment is performed on objects of different sizes: 4 bytes, 8 bytes, 16 bytes and 32 bytes in order to see the impact of size evolution on performance.

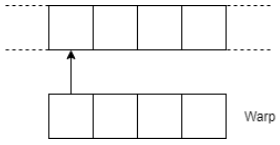


Figure 4.1: Only one

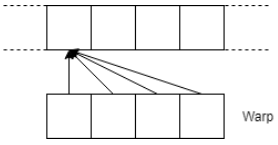


Figure 4.2: Same one

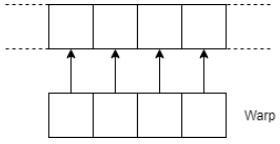


Figure 4.3: Adjacent

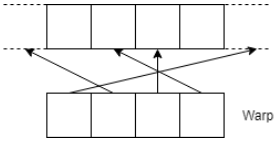


Figure 4.4: Random

Figure 4.5: Different pattern of accesses

We performed the experiment in various configurations by varying the number of warps and blocks, in order to obtain the Cartesian product from 1 to 32 (2^5) for warps and 1 to 1024 (2^{10}) for blocks, by doubling steps (matrices of 5×10). We collected the result of 10 runs, each receives a different original offset for the position, and performed 10 iterations each time, to reduce the cost of launching one kernel. Various statistical elements are then collected, in particular moments or quantiles.

4.3 Results

As the results depend on many factors: the type of access, the size of the data, the number of warps and the number of groups, we present a non-exhaustive view of the results. Please note the scales vary from one result representation to another. We will try to extract the key elements of the experiment and present them and we will quickly move on to elements that we have found less relevant.

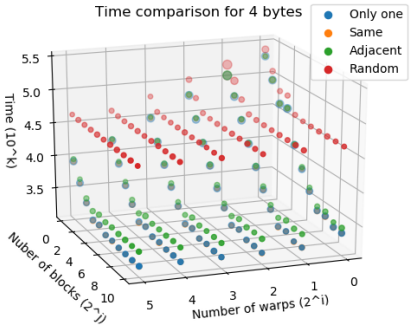


Figure 4.6: Time to access to 4 bytes

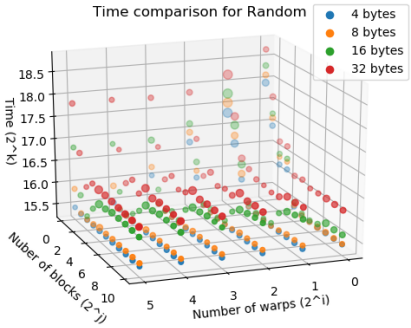


Figure 4.7: Time for random accesses

Several remarks can be made on the first and raw results:

- Access times stabilize relatively quickly, and adding new warps or groups does not save time beyond one configuration. This corresponds quite naturally to the saturation of the bandwidth. Indeed, we are not able to issue more accesses than the memory is able to offer in the same time interval.
- We achieve the maximal bandwidth possible in all the cases if we consider that the elements are loaded in cache line of 128 bytes. We also remark that the practical bandwidth is lower than the theoretical one (we get about 100GB/s instead of the 112GB/s), this may be due to the inner latency of these instructions (12 cycles for issue latency - time to start the instruction - and 172 for the whole latency - time before being able to start a new instruction - as reported by the gpuperformance tool [1]).
- When the random accesses on 16 and 32 bytes exceed a certain configuration (number of warps $\times$ number of blocks $\geq$ constant), the access time increases slightly to stabilize with a higher variance. This may be due to a stronger eviction in the cache due to the pressure on it which is increasing.

These raw data are not really surprising by themselves but when we compare the performances between them, we can observe some other aspects where the results are quite surprising:

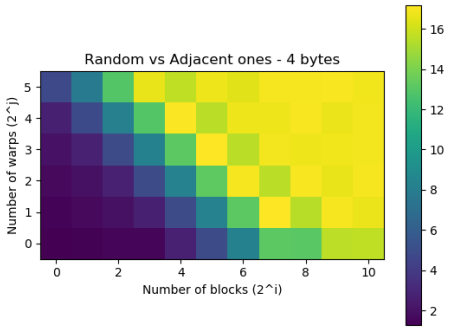


Figure 4.8: Slowdown between adjacent and random accesses on 4 byte elements

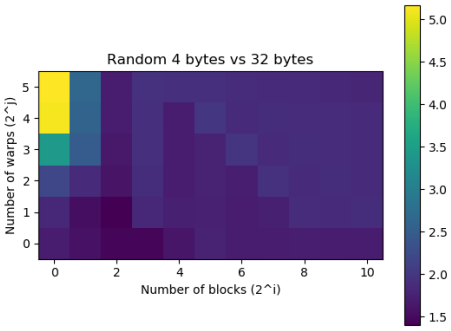


Figure 4.9: Slowdown between random accesses on 4 bytes and 32 bytes

Indeed, we would expect to have a slowndown factor of 32 since the random data are spread randomly in 32 different cache lines, but we only get 16. This is due to the fact that the elements are not coalesced and thus we need to load 2 cache lines in average for adjacent strategy. But if we increase the size of the data, the slowing factor becomes less and less important, with sizes of 8 and 16 bytes, we are only at a factor of about 12 and 8 respectively and when we go to 32 bytes (which is quite large), the numbers collapse to 3.5. It becomes to be more interesting to perform random accesses than the equivalent amount of sequential.

Curiously, we observe that the time to obtain the data is noticably sublinear with the data size and not directly proportional, this would suggest that a mecanism of prefetching is provided by default and, in the case of random accesses, the cost is essentially free

since the cache lines are already loaded. The upper left corner of the figure 4.9 remains a complete mystery.

From this small experience, we can conclude a very important point. Accessing the data in an adjacent manner is always more interesting but the gap becomes smaller and smaller compared to random if you need a lot of data. This is a direct consequence of a rather logical phenomenon. The bandwidth remains the same no matter what happens, what changes mainly is the efficiency of the data loaded. It is better to consult all the cells obtained during a random access and thus compact the data in this case, whereas the sequential case will be better on average.

In addition, and in the course of this work, we have noticed several times that we have been confronted with memory dependency problems. The mechanism used to hide the latency of instructions was not sufficiently effective and a significant amount of time could be lost on this aspect, which also led to data dependency and execution issues. This was especially the case for hash tables.

Chapter 5

X-Fast Tries

When we leave the realm of data structures purely based on comparing keys, we may expect to reduce the complexity of several operations, in an analoguous manner than we can achieve linear complexity with counting sort for instance in $O(N + K)$ for integer values instead of the general lower bound $\Omega(N \log N)$ in comparison model. In this chapter, we will present a quite old data structure defined in word-RAM computation model and some of its variations.

5.1 van Emde Boas trees

We shall first introduce the so-called *van Emde Boas* trees which support dictionary-like operations in $O(\log \log N)$ worst-case time. But it requires that the elements are integers within the range 0 to $N - 1$, with no duplicates allowed. As it will introduce some confusions, we will redefine some notions:

- We will denote the set of possibles integers as the universe U: $\{0, 1, ..., u - 1\}$.
- u will be the universe size, and is often intented to be a power of two 2^w where w is understood as the word size defined in word-RAM models.

This data structure behaves like a set but owns two main other operations: the predecessor ($\max(\{e|e < x, e \in S\})$ and the successor; it can thus be used as a dictionary or a priority queue [96]. But it consumes a lot of memory $O(u)$.

We will present it in a different fashion than the one used by van Emde Boas et al. [96] but in the way that we can retrieve it by Cormen et al. [44].

5.1.1 Direct mapping & stratified tree

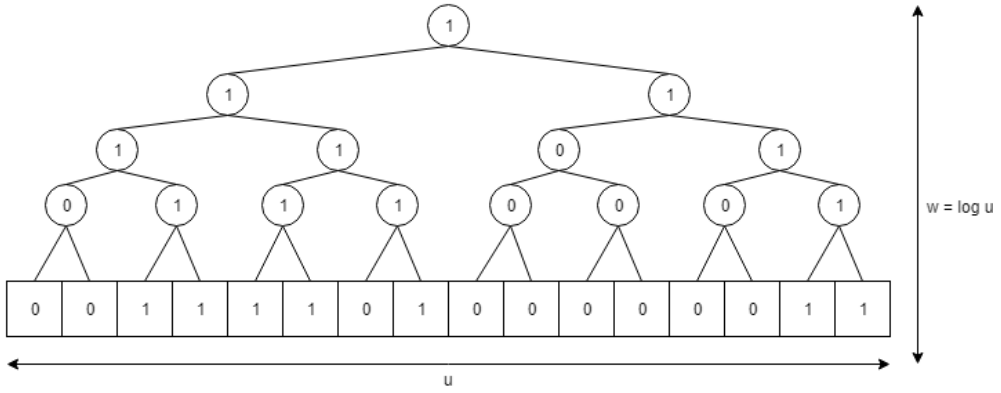
With a really huge amount of memory, we could save all the elements of the set as a bit array where the ith position determines whether the i element is set. This grants natural $O(1)$ for membership testing, insert and delete operations but finding the predecessor or successor could lead to $\Theta(u)$.

We can cut the space search for predecessor/successor by constructing a tree such that the node is marked as 1 if none of its children is 0 and 0 otherwise (see Figure 5.1). The membership is still in $O(1)$ but all the other operations became potentially $O(\log u)$. The idea consists to bound the height of the tree which will fix the amount of recursion and thus the complexity. van Emde Boas [94] came up, in 1977, with the solution to

decompose the problem in *clusters* divided in m *galaxies* of size k. This cluster will be used to sum up the information of the whole subtree. One can remark that we can easily determine in which tree the element will go based on its index $\lfloor \frac{x}{k} \rfloor$.

Now, to achieve the $O(\log \log u)$, we need to cut recursively the space in cluster of $O(\sqrt{u})$. This leads to: $T(u) \leq T(\sqrt{u}) + O(1) \rightsquigarrow T(u) = O(\log \log u)$. Then, we will have to ensure that the algorithms do not need to perform two recursions, which would lead to $O(\log u)$ instead of $O(\log \log u)$, we must absolutely recurse on one and unique path in the tree. We will also remark that this structure has a lot of historical background [95].

Figure 5.1: Stratified tree



van Emde Boas trees

The van Emde Boas trees is a data structure which can be seen as a tree with high degree and defined recursively. It is composed of two elements:

- Clusters: Each cluster holds $\sqrt{u}$ subclusters and this recursively until it contains all the elements of the universe.
- Summary: Each cluster is summarized by one element which will help to guide the path to the predecessor or successor element. It is also recursively defined since the summary summarizes the information of its subtree.

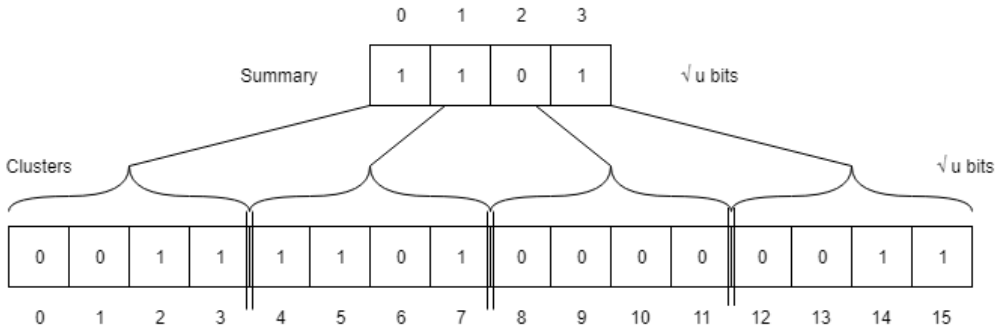
The whole point is that we can access these helper structures in constant time due to the definition of our tree where the index of the cluster can be easily computed at each step. Hence, the cluster and its relative offset is defined as $(\frac{x}{\sqrt{u}}, x \bmod \sqrt{u})$ which in the case of binary numbers leads to: $(\frac{x}{2^{\frac{w}{2}}}, x \bmod 2^{\frac{w}{2}})$ which are roughly a shift and a mask operations.

K. Melhorn and S. Näher [74] proved that we can achieve $O(N)$ memory if we use hash tables which map prefix of the integers to ordered linked-list nodes while preserving complexity with high probability. An idea similar to Y-fast tries can also be applied for this purpose.

5.1.2 X/Y-fast tries

The problem with the *van Emde Boas* trees is that they consume a huge amount of memory. D. E. Willard [98], in 1982, proposed two solutions to this problem based on the stratified tree idea. The first part of its answer is called the *X-fast* trie, the idea

Figure 5.2: vEB idea



is rather simple, only store the present elements, whose bit are set to one. We save the path to the node (with left as 0 and right as 1, for example) in a dynamic hash table, called level-search structure, and we associate it with the minimal/maximal value of the subtree. So instead of directly jumping to the subtree, we ensure the presence thanks to the hash table. This technique consumes $O(N \log u)$ memory to store each elements and their binary representation (note that we can spare some space since a common prefix/suffix is shared). Also, we may need to update a whole path, up to $O(\log u)$ values but we can achieve predecessor/succesor queries in $O(\log \log u)$ with high probability and search for one specific elements in $O(1)$. We can also maintain a linked list of the entire set of occupied leaves, this allows that given node x, we can find the successor and the predecessor in $O(1)$.

Second, we gain a lot of space, but we lose the $O(\log \log u)$ on insert and delete. Hopefully, the *Y-fast* trie are there. They consist of two layers, the summit of the trie is made of a *X-fast* trie such that it holds $\Theta(\frac{N}{\log u})$ elements and the leaves are other canonical binary seach trees of $O(\log u)$ elements where one representant is placed in the upper trie. This thus achieves the $O(N)$ in space and the operations are bound by $O(\log \log u)$ in the bottom trees since the update time is dominated by the leaf structures. Complexity is exchanged throughout the structure by amortizing it into the terminal elements.

Finally, let's conclude by saying that this data structure even though has nice properties, we may prefer to implement a *X-fast* trie which are simpler to code and more efficient in practice. We need also to emphasize on the fact that the performances of this tree depends heavily on the hash table performances and the time to insert one element can thus vary by a huge factor. Notice that the complexity is not optimal, it is expected to be $O(\frac{\log \log u}{\log \log \log u})$ for $N^{O(1)}$ space [82]. Other data structures based on similar concepts have been developed to address the problem of the predecessor or the dynamic least common ancestor [33, 25]. You may also be interested in the little brother of this data structure, defined in the AC^0 computation model (constant depth circuit which behaves as word-RAM like without multiplication), and called *fusion tree* [54].

5.2 X-fast trie

In summary, X-fast trie can be seen as a kind of binary tree where each level corresponds to a set of values sharing the same common prefix. Traditionally, the left child corresponds to the addition of a 0 to the prefix and the right, to a 1. There are thus $O(\log u)$ levels in the trie. All the elements are stored at the leaves (conceptually sorted) and the

internal nodes are stored only if leaves exist in their subtree. The key idea of the data structure is to represent each level through a hash table which will grant operations in $O(1)$ w.h.p.. Not only the presence of a prefix is stored but also the minimum/maximum of the subtree.

This data structure is intended to be used as an ordered dictionary with those five operations but can also be used as a heap or priority queue:

- Find(k): find the value associated with the given key.
- Insert(k, v): insert the given key/value pair.
- Delete(k): remove the key/value pair with the given key (if it exists).
- Predecessor(k): find the key/value pair with the largest key less than or equal to the given key.
- Successor(k): find the key/value pair with the smallest key larger than or equal to the given key.

Find:

When we search for one item, we only need to perform one query on the leaf-level to get our answer. This allows access to data in $O(1)$ which is quite interesting.

Insert:

When we want to insert a new key/value pair in the trie, we must do several operations: First, we search for the lowest level which shares a common prefix with our element to insert. The path is then completed to lead to the leaf and the key/value pair is inserted at the bottom while updating a double-linked list gathering the leaf elements. Finally, the upper nodes are updated to take into account the new key as described for the predecessor/successor operations.

Delete:

We delete the element at the very bottom and move up the path to delete all prefixes that have no children other than the curent element you want to remove. We also need to update the predecessor/successor information in internal nodes in accordance.

Predecessor/Successor:

In order to find the predecessor of a key in the trie, we must first look for the node that shares the largest common prefix with our query. Then, it is enough to refer to the value pointed by this node. The whole idea lies in this fact that each node keeps, if necessary, either the largest element of the subtree if there is no right child, or the smallest if there is no left child. Since we know we will be looking for the lowest node in the trie every time. Depending on the search for the successor or predecessor, it will sometimes be necessary to access the next item in the doubly chained list of elements at the bottom level, in case there is another branch that does not belong to the subtree that we are browsing.

We point out that only the find queries are $O(1)$ operations, insertions and deletions require in the worst case $O(\log u)$ since we would need to either insert or destroy each

intermediate nodes. Predecessor and successor queries can be made in $O(\log \log u)$ if we perform a binary search on the levels of the trie.

We will end by expressing that, on a theoretical point of view, it is easy to parallelize the data structure by making queries on each level independently and then gathering the results or updating the nodes if required. From a practical point of view, this will turn somewhat more complicated. We will return very largely to these problems in a future chapter[8].

Chapter 6

X-fast trie implementation

After having seen the different concepts and perceived what we wanted to obtain, we can start to implement our problem in order to see what it really is, what the performance can be and if it can be really interesting. We will begin by discussing in more detail the X-fast tries themselves, with focus on the different possible implementations and their various advantages. We will follow by presenting the results we obtained in comparison to another data structure that proposes simular properties and which is widely used. We will end by giving the appropriate conclusions and the possibilities for improvement as well as the main interests and defects.

We first address them in the sequential framework, this will allow us to identify the main difficulties and implementation details. Once those work properly and we will start to see the first results, we will then be able to see what will be needed to adapt them to the concurrent framework.

6.1 X-fast trie

Following the various observations we made in the two previous chapters, we finally embarked on different implementation for the X-fast tries. All share a common base, the trie is seen, on the one hand, as a set of intermediate levels that allow to navigate in the trie and determine which is the predecessor or successor of an element depending of its prefix and, on the other hand, a last level actually containing the key-value pairs.

All of these rely on the use of a hash table, based on the open-addressing principle and with the linear probing strategy for the implementation, we will come back to these notions in the forthcoming chapter[7]. We have thought about different ways to build and operate on these and we have identified three main implementation families:

6.1.1 Binary search

In order to find out from which height the new nodes should be inserted, a simple dichotomous search is performed on the levels. This introduces a natural complexity in $O(\log \log u)$ on the search for the predecessor/successor. However, the entire warp is used for this task since this treatment is inherently sequential. Once the lowest node has been found, all that remains is to complete the trie and update the parents as long as necessary.

We have tried to show the different steps performed by these different implementations with their respective time indication. The search for nodes is symbolized by the

black color, the insertion of new nodes is tinted green and the update of the parents is red. The arrows represent which levels are actually accessed. And the number defines the order of execution, a same number indicating parallelism of the task.

For this, we decided that the whole warp should contribute to the task, the idea was that when we do our research for a particular new location or object in the hash table, the different threads will access the following probes at once. This allows all threads to perform the same task and therefore there is no divergence and thus variance is minimized. This is done at the cost of more total data access. We update the parents as long as the minimum/maximum of their subtrees is not affected by our new element.

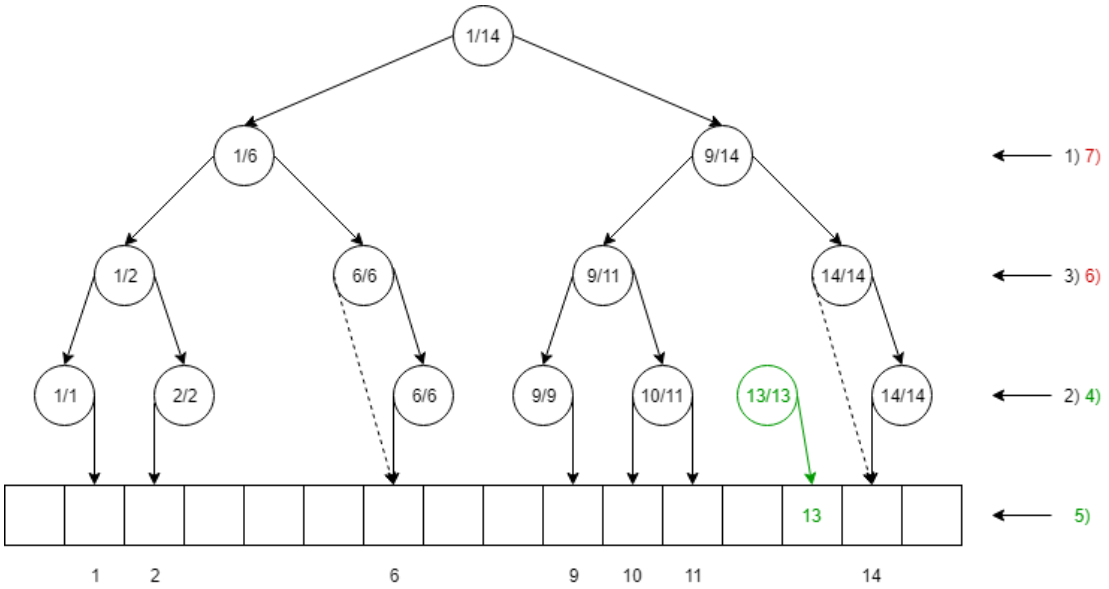


Figure 6.1: Binary strategy

6.1.2 Warp search

The previous technique is unnecessarily inefficient, all queries that resulted in the absence of a prefix cannot be used effectively and the gain provided by a successful query is marginal. This is why, and following the observations on memory accesses, we opted for each thread to make an independent request on a particular level. All levels are therefore read at once and thanks to the election mechanisms in warps, the lowest level is determined at once.

The voting functions in warps allow all threads in a warp to perform a broadcast operation followed by a reduction in a single step and at very high speed. These take as input an integer that will serve as predicate, all threads will then compare their own value with that provided, the comparison results are combined and sent to each participant. It is then enough to recover the position of the bits set to one to know for which threads the predicate was true.

Now that we have found the lowest node, the first part will update the parents simultaneously with each time only one thread associated to a level and the other part will insert the new ones in the same way. However, care must be taken to resize these hash tables. Indeed, either we can resize the hash table when we deem it necessary or we can work with “amortized” operations, each time we insert an element, we take the

opportunity to transfer elements from the old table to the new one. In our case, we opted for the first approach which is simpler to implement and a preliminary step is thus taken to determine whether there is still enough space. We will come back to this aspect later.

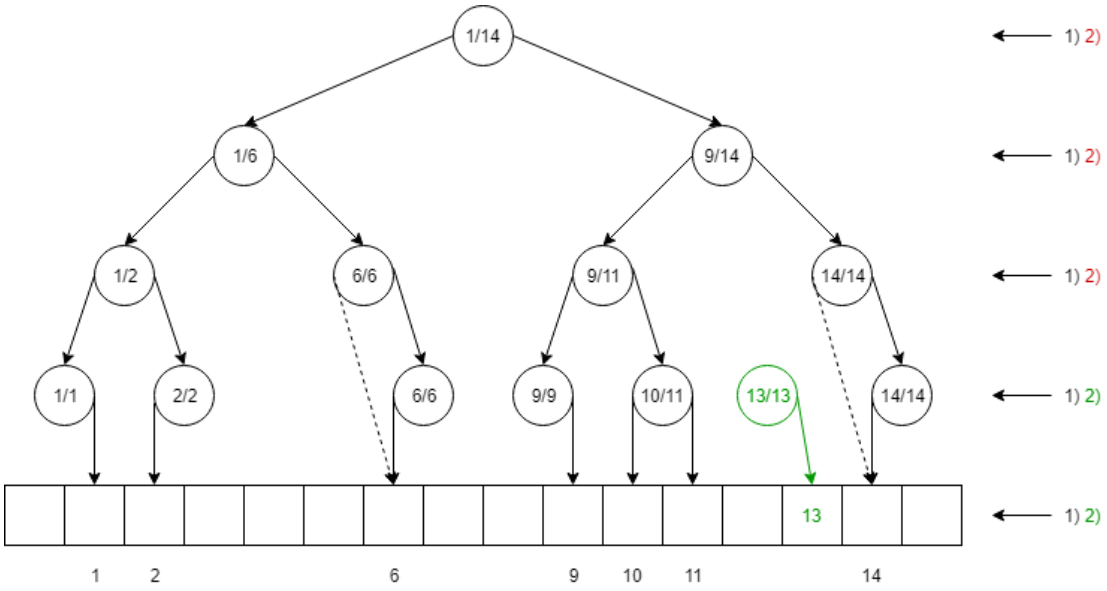


Figure 6.2: Warp strategy

6.1.3 Group search

After implementing the warp search idea, it seemed quite natural to look for a subset of the trie instead of one and only one element. Those will be stored as a simple array in their associated parent, to mimick a structure close to the one we find out in van Emde Boas layouts which grant search in less than $O(4 \log_B N)$ memory transfers [27]. This would store fewer elements in total and therefore consume less memory, but it is not the only advantage. This also reduces the number of line caches needed to fill all queries and also decreases the likelihood of falling into a worst case scenario with a degenerate item search case in a hash table. In addition, warp access will respond to more levels and therefore require a lower number of total accesses in the end. If we use 64-bit keys, the warp strategy will require two accesses whereas the group strategy, even with a group size of 1, only one will be enough to fullfill the request.

Here, we define group size as the number of levels and therefore children that are grouped into a single entity that will serve as a value. One bit and all its prefix will serve as the key. Care must be taken to update these elements in accordance with their memory layout. It is also necessary to ensure that both operations of update and insertion are done within the same group when the lowest node falls inside a group. Warp search can be seen as a special case of group search when the group size is equal to 0.

6.1.4 Experiment

We set up an experimental protocol in order to compare the various possible strategies to implement an X-fast trie. This consists in pre-allocating a large memory area, attaching to it a memory allocator (so called “linear”) that works by assigning the next memory

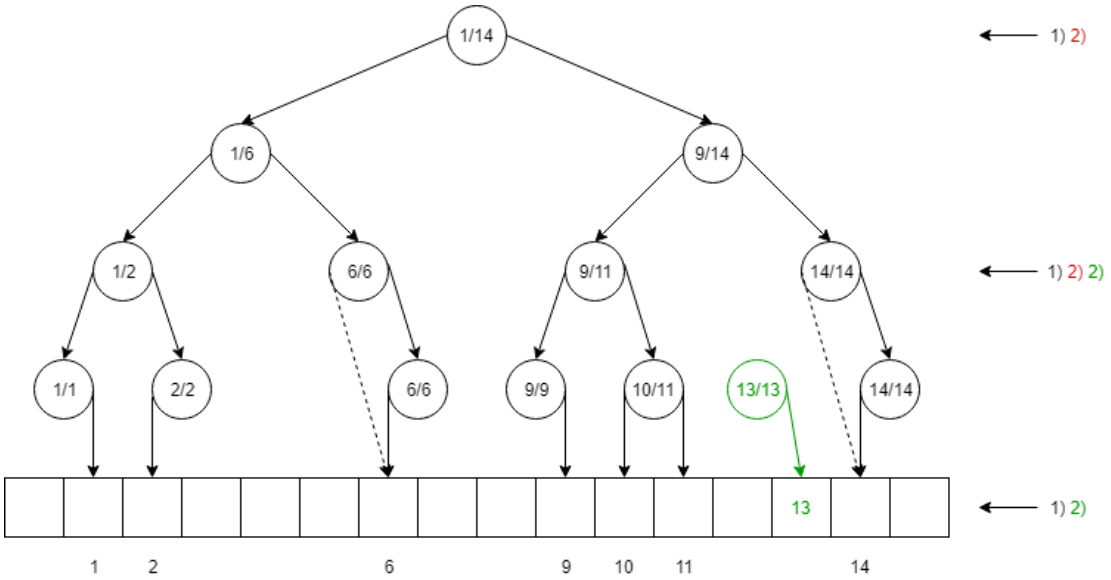


Figure 6.3: Group strategy - group of size 1

area each time, so the allocations are constant time and disallocation operations are equivalent to no operations, and from which we will pick for our problem. We then build our tries by preallocating enough memory to contain all our insertions, this is done in order to provide a more independent result on the hash tables used. We will see that there is some flexibility on hash table and that more complex implementations can remain competitive and interesting in practice.

Once our tries are well built, variables correctly initialized and so on, we insert a large amount of elements and we measure the time taken on an average of 10 runs with 10 iterations. Each run receives its associated and random seed which will be used to feed a pseudo random generator (hash function) based on the current index of the insertion and the associated seed which will generate the distribution of keys for the elements to insert. We do similar work to test search operations. We pre-construct our tries and pre-fill them with enough elements, and we measure the time taken to answer all our successful requests, both by making requests by warp (the 32 threads are used to answer the same key) and by thread (each thread looks for the value associated with a different key). Finally, the same experiment is done but for the predecessor and successor queries; the look-up keys are created as a combination of the random numbers and the original seed, we may expect to have low exact match on the keys. Remark that only one warp is active in those experiments, no concurrency is made.

6.1.5 B+ Tree

In order to see the performance of our data structure, it was interesting to implement another one that shares the same operations and similar properties. We opted for B+ trees [42] which is the trie version of the classical B-trees, introduced by R. Bayer and E. M. McCreight in 1971. Indeed, it is a very classical associative data structure which has the operations which interest us the insertions/deletions/searches and predecessor/-successor queries, which has been widely studied for its good properties [27] and which was ported successfully on GPU [50]. B-trees can be seen as generalizations of binary search trees where instead of having only two children, we have B children; it is also

self-balancing. This naturally gives complexities in $O(\log_B N)$ for many dynamic operations.

We decided to present B+-trees instead of B-trees because they seem, for us, conceptually simpler and the algorithms are essentially the same. They have the advantage of being cleaner and not mixing two different notions. The letter m will design the number of children in one node in this section.

A B+-tree is a rooted tree having two kind of nodes:

- Internal node: are made of $m-1$ keys ordered with m associated pointers linking either to another internal node or a leaf node. Those keys are meant to separate the ranges of keys stored in each subtree.
- Leaf node: are made of m keys ordered with m associated values.

The B+-trees obey these following properties [42]:

- Every node has at most m children.
- Every non-leaf node (except root) has at least $\lceil \frac{m}{2} \rceil$ children.
- The root has at least two children if it is not a leaf node.
- A non-leaf node with k children contains $k-1$ keys.
- All leaves appear at the same level.

Searching in a B+ tree is similar to searching in a binary search tree, except instead of making a binary decision at each node, we make a multiway branching decision based on the number of children in that node and our look-up key. A simple binary search can be done on the keys of one node.

Inserting a new element into a B+-tree is more complicated than inserting a key into a binary search tree. We start with a similar step where we look for the leaf position where to insert our new element and we insert the new key into an existing leaf node. However, we cannot insert an element when the leaf is full. For that, a special operation called soberly "split" is carried out, it consists in dividing the number of children of the leaf in two new leaves. The middle key will be moved to the parent internal node to identify the split point between the two new trees. But if the parent is also full, we have to divide it before we can insert the new key, and so we might end up dividing over the entire height of the tree.

Deletion is a bit trickier, there are two main strategies, either we locate and delete the item, then restructure the tree to retain its invariants or while going down the tree, we restructure the nodes such that the invariants are kept and when we encounter the final element, we can simply remove it without breaking any invariants. We must also pay attention to two particular cases, if the key is used as a separation element between two nodes or if this is the minimum/maximum of a leaf-node. In these two cases, it will also be necessary to update the parent's keys.

The essential difference between B-trees and B+-trees is this mixture between values and pointers in intermediate nodes, otherwise their structure is really similar. We opted for the B+ trees variant instead of the simple B trees because they allow to group keys and pointers and thus to guarantee a better distribution of data within the levels but

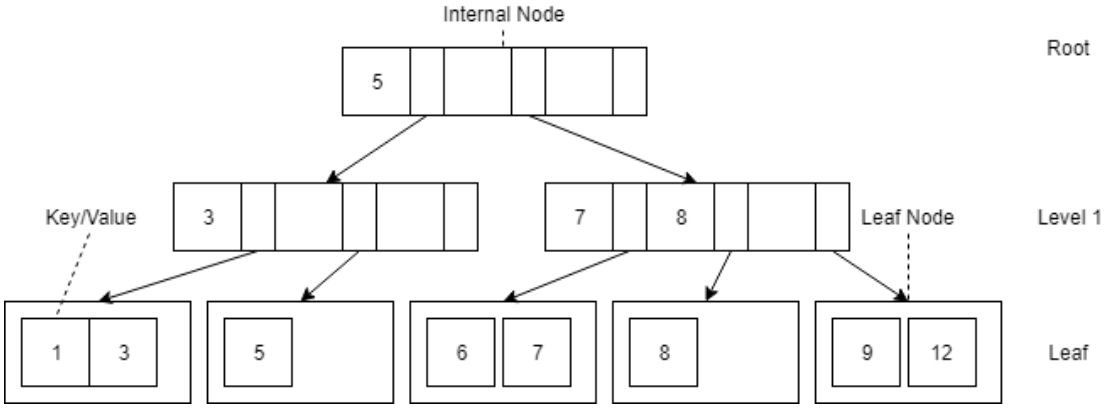


Figure 6.4: B+Tree representation with $m = 3$

keys are then copied at several places. The distinction between values and pointers is also more clear and avoid some cumbersome detail of implementation. We opted to set the number of children of one node to $m = 32$, because it allows to get the child or the position to insert at once thanks to the voting system in the warps. Each thread can possibly compare with its associated key in the node at once.

6.1.6 Results

We will start by presenting the results obtained for the insertions, we will follow by the two different search operations and we will end with the predecessor/successor queries. Each time, we will discuss the results, notice the potential improvements or elaborate on points which seem more relevant to our case. The graphs should be interpreted as the mean time taken to perform the number of operations (described by the head of the row) with the standard deviation between parentheses and the strategy per column, the intermediate rows represent the ratio between the time taken by the B-tree and the X-fast trie strategy (the lower the number is, the better the strategy is). All these times are expressed in microseconds (µs):

Insertions

We performed the experiment twice, once with 32-bit keys and once with 64-bit keys. After running our experiments and extracting the data here is what we got:

We can make different observations in view of the results obtained:

- The strategy of doing a dichotomous search (Binary) on levels is always slower. Indeed, it is very inefficient in the way the elements are accessed. The gain brought by adjacent accesses does not compensate for their very large number, random accesses are more interesting in this case.

- The time to insert elements is almost linear in all cases, there is a big exception for B-trees when you go from 8192 (2^{13}) elements to 16834 (2^{14}). Indeed, this corresponds to the transition between three and four levels in the tree with all the rebalancing that this induces due to the fact that with used B-trees with $m = 32$ (to correspond to the size of a warp).

Table 6.1: Time to insert K elements with all these strategies with 32-bit keys (in µs) mean time (standard deviation) - intermediate lines represent slowdown factor

Insertion	B-tree	Binary	Group1	Group2	Group3	Group4	Warp
1024	3064 (30)	6053 (21)	1691 (13)	1695 (16)	1736 (11)	1830 (16)	1593 (13)
		1.98	0.55	0.55	0.57	0.6	0.52
2048	6024 (17)	11823 (45)	3438 (30)	3434 (22)	3554 (20)	3740 (15)	3242 (23)
		1.96	0.57	0.57	0.59	0.62	0.54
4096	11986 (15)	22974 (60)	6884 (18)	6969 (24)	7148 (26)	7529 (25)	6517 (13)
		1.92	0.57	0.58	0.6	0.63	0.54
8192	23916 (30)	44515 (127)	13682 (41)	13856 (34)	14257 (23)	15043 (35)	12972 (28)
		1.86	0.57	0.58	0.6	0.63	0.54
16394	65411 (181)	86178 (223)	27219 (40)	27726 (40)	28571 (52)	30107 (55)	26091 (49)
		1.32	0.42	0.42	0.44	0.46	0.4
32768	133339 (254)	167413 (444)	55193 (81)	56032 (85)	57656 (84)	60546 (88)	52708 (72)
		1.26	0.41	0.42	0.43	0.45	0.4
65536	268613 (581)	320465 (932)	114746 (172)	116848 (158)	119413 (169)	124226 (169)	107993 (173)
		1.19	0.43	0.44	0.44	0.46	0.4

Table 6.2: Time to insert K elements with all these strategies with 64-bit keys (in µs)

Insertion	B-tree	Binary	Group1	Group2	Group3	Group4	Warp
1024	3079 (34)	6743 (26)	1893 (13)	1900 (11)	1966 (12)	2087 (20)	1837 (17)
		2.19	0.61	0.62	0.64	0.68	0.6
2048	6131 (16)	12947 (40)	3749 (31)	3818 (29)	3949 (26)	4110 (29)	3647 (32)
		2.11	0.61	0.62	0.64	0.67	0.59
4096	12181 (24)	24979 (61)	7413 (30)	7555 (20)	7840 (22)	8157 (19)	7197 (28)
		2.05	0.61	0.62	0.64	0.67	0.59
8192	24330 (32)	48336 (114)	14914 (31)	14987 (26)	15552 (29)	16384 (28)	14360 (22)
		1.99	0.61	0.62	0.64	0.67	0.59
16394	67532 (179)	93456 (209)	29936 (44)	30062 (43)	31192 (38)	32945 (68)	28925 (30)
		1.38	0.44	0.45	0.46	0.49	0.43
32768	137071 (297)	181747 (442)	60658 (78)	61541 (106)	63654 (89)	66154 (95)	58915 (72)
		1.33	0.44	0.45	0.46	0.48	0.43
65536	279735 (544)	349635 (824)	126038 (165)	126811 (179)	130452 (166)	135358 (198)	121929 (184)
		1.25	0.45	0.45	0.47	0.48	0.44

- The time to perform 64-bit insertions isn't much higher, about 3 percent for B-trees and 5 percent for the others (GroupK and Binary), except for warp insertions which go up to 8 percent. The time is expected to be roughly constant for B-trees since their operation is independent of the size of the key. Group strategies benefit from the fact that a single warp allows access to all 63 levels at once since we ask for at least two levels at once, while the warp strategy requires two accesses.

- More importantly, GroupK and Warp strategies show a real gain over B-trees when a large number of insertion requests are made. This supports the observations made on memory access[4] and the idea that B-tree requires $O(\log_B N)$ sequential accesses while X-fast tries only require $O(\lceil \frac{\log U}{32} \rceil)$ accesses but random. The standard deviation also tends to be lower than for B-trees since potential rebalancing is avoided.

- Group strategy with group size of 1 or 2 and warp strategy presents the best performances for this operation and presents a large improvement compared to the B-tree reference structure.

Search

And this is what we collected for the search operation for those both experiments, those per warp where all the threads in a warp are used to find an element and those per thread where each thread searches for its own element:

Table 6.3: Time to get K elements with 64-bit keys (in µs)

Table 6.4: By warp

Search by warp	B-tree	Warp
1024	1960 (38)	781 (22)
		0.4
2048	3812 (31)	1579 (14)
		0.41
4096	7480 (15)	2997 (18)
		0.4
8192	14866 (22)	5904 (17)
		0.4
16394	46063 (111)	11663 (11)
		0.25
32768	91482 (213)	23237 (22)
		0.25
65536	186031 (383)	46180 (23)
		0.25

Table 6.5: By thread

Search by thread	B-tree	Warp
1024	256 (31)	101 (16)
		0.39
2048	462 (48)	142 (19)
		0.31
4096	879 (92)	180 (8)
		0.2
8192	1651 (153)	329 (5)
		0.2
16394	3354 (197)	690 (7)
		0.21
32768	6769 (245)	1606 (30)
		0.24
65536	14642 (270)	4278 (35)
		0.29

Let's start by saying that we mentioned the results only for one of the strategies implemented for X-fast tries since all have the same underlying hash table for the elements at the bottom. The differences in results were therefore marginal (less than 0.1%) and not significant according to a paired t-test and common sense. Once again, we obtained very interesting results:

- "search by warp" for X-fast tries is much faster than for B-trees, the standard deviation is also much smaller since we have to do $O(\log_B N)$ access for B-trees and $O(1)$ for X-fast tries with high probability by the intrinsic properties of these structures.
- "search by threads" are also magnitude faster for a lower standard deviation. This is explained by the number of accesses required for B-trees which are $O(\log_B N * \log_2 B)$ since we have to perform a dichotomous search on the keys while they remain in $O(1)$ for X-fast tries.
- The speed-up ratio to access data per thread compared to warp is only about 12 for B-trees and over 12 for X-fast tries. This is consistent with the observation that random access is still more attractive than performing the equivalent number of sequential accesses.
- An increasing standard deviation can be expected for B-trees as a function of the number of queries made, given the number of possible branches in thread accesses greater than for hash table accesses which remain consistent and proportional to the load factor.

- The time required to access items per thread is not directly proportional to the number of queries made due to the characteristics of our hash table, the greater the load factor, the longer can be expected to retrieve one element.

Predecessor/Successor

Next, we need to analyze the results related to the requests of predecessors and successors:

Table 6.6: Predecessor queries on 64-bit keys

Predecessor	BTree	Binary	Group1	Group2	Group3	Group4	Warp
1024	2634 (33)	8235 (34)	2851 (24)	4729 (29)	4842 (38)	4984 (25)	4249 (37)
		3.13	1.08	1.8	1.84	1.89	1.61
2048	5145 (18)	16342 (41)	5629 (14)	9507 (35)	9775 (42)	10016 (41)	8520 (33)
		3.18	1.09	1.85	1.9	1.95	1.66
4096	10124 (20)	32645 (50)	11387 (19)	19227 (47)	19659 (59)	19946 (63)	17208 (48)
		3.22	1.12	1.9	1.94	1.97	1.7
8192	20157 (19)	65276 (39)	23387 (28)	38884 (44)	39746 (42)	40349 (49)	35276 (39)
		3.24	1.16	1.93	1.97	2.0	1.75
16394	53144 (121)	130700 (66)	49046 (36)	79847 (66)	82120 (77)	83651 (72)	73536 (78)
		2.46	0.92	1.5	1.55	1.57	1.38
32768	105540 (212)	262535 (97)	108955 (78)	171596 (148)	174387 (101)	178055 (127)	159783 (134)
		2.49	1.03	1.63	1.65	1.69	1.51
65536	214314 (571)	525281 (164)	245071 (853)	371696 (610)	376686 (496)	378355 (534)	347530 (745)
		2.45	1.14	1.73	1.75	1.77	1.62

Table 6.7: Successor queries on 64-bit keys

Successor	BTree	Binary	Group1	Group2	Group3	Group4	Warp
1024	2381 (25)	8137 (31)	2850 (35)	4655 (30)	4750 (39)	4929 (41)	4156 (26)
		3.42	1.2	1.96	2.0	2.07	1.75
2048	4680 (20)	16135 (35)	5623 (31)	9388 (52)	9577 (24)	9885 (41)	8310 (54)
		3.45	1.2	2.01	2.05	2.11	1.78
4096	9174 (19)	32270 (47)	11355 (33)	18982 (48)	19221 (60)	19642 (74)	16845 (58)
		3.52	1.24	2.07	2.1	2.14	1.84
8192	18305 (18)	64561 (51)	23319 (31)	38360 (38)	38878 (76)	39730 (99)	34595 (52)
		3.53	1.27	2.1	2.12	2.17	1.89
16394	49485 (122)	129250 (77)	48912 (30)	78756 (103)	80659 (110)	82431 (101)	72079 (74)
		2.61	0.99	1.59	1.63	1.67	1.46
32768	98362 (264)	260552 (249)	108610 (97)	169961 (173)	171513 (202)	176271 (142)	156794 (162)
		2.65	1.1	1.73	1.74	1.79	1.59
65536	200266 (392)	521108 (173)	244131 (872)	368136 (702)	370038 (699)	374094 (721)	346341 (642)
		2.6	1.21	1.83	1.84	1.86	1.73

The results are somewhat surprising. We can observe different elements:

- All times are directly proportional to the number of elements queried, which is expected.

- The time required to perform predecessor/successor queries in B-trees is roughly equivalent to the search operations. This is quite natural by the nature of tree search. There is also a slight difference between the two types of queries since the same dichotomous search was performed while in one case one looks for a lower bound and in the other an upper bound on the keys. The treatment is therefore a bit dissimilar between the two cases, but can easily become the same. While the treatment is the same in both the case for the X-fast tries.

- X-fast tries are significantly slower than B-trees by a factor 2 ! This result is somewhat surprising when you consider that, for the X-fast tries, the time taken to insert elements is roughly the same. Accessing the different levels seems to be quite logically the limiting factor.
- We were expecting a certain slowness related to this global look-up followed by the two potential hash table searches (when we search for the predecessor and there is no left subtree for example) that are being done but not to this point. Perhaps an element escaped our re-reading of the code and a pessimisation slipped in. The other explanation being that it is the rebalancing in the tree which strongly degraded the performances of B-tree. Operation that simply does not exist for queries.
- One should not be worried due the presence of an exacerbated difference in performance between size 1 and 2 groups; but this is simply due to the fact that we can not fit all the elements in a single cache line with group of size more than 1. The results are roughly equivalent with queries on 32-bit keys and only the strategy of grouping by size of 1 seems really effective.

Memory consumption

Finally, all that remains is to analyze the memory consumption of such data structure. We have tried to make a approximate calculation on the memory consumption of these different structures. Let us begin by describing the assumptions we have made:

- The size of the keys for both B+-trees and X-fast tries is the same, and this applies to the internal nodes as well as the leaves or the intermediate and bottom levels. If we have an X-fast tries of 32 levels, the keys are 32 bits, or 4 bytes.
- We set the size of the pointers contained in the intermediate nodes to 8 bytes.
- The hash tables used are twice as large as the number of elements they can accommodate. And the resizing takes place when the load factor reaches 50% (in order to simplify the calculations and to have only powers of 2).
- The size of the values was fixed at 4 bytes but this data only intervenes for the leaves or the bottom level.
- The B-tree contains $m = 32$ children per node.

We will start by explaining the calculation performed for B+-trees:
If we have N randomly distributed elements in the leaves, then we have $\lceil \frac{N}{B} \rceil$ leaf nodes. Since internal nodes have potentially up to B children, the tree size is simply $\log_B N$ and there are thus $\lceil \frac{N}{B} \rceil * \log_B N$ elements in total. The number of intermediate nodes is therefore this quantity minus the number of leaves. So the total amount of memory is proportional to:

$$\lceil \frac{N}{B} \rceil * B * (\text{key size} + \text{value size}) + (\lceil \frac{N}{B} \rceil * \log_B N - \lceil \frac{N}{B} \rceil) * B * (\text{key size} + \text{pointer size})$$

And now for the X-fast tries:
Under the previous assumptions, we know that the size of a hash table containing N

element corresponds to the next power of 2. Second, the intermediate level i contains at most 2^i elements since only the first i bits are read and, in group strategies, those nodes have at most 2^k (where k is the group size) children layout in van Emde Boas fashion. If we define the height of the tree as w; the computation resumes to:

```
def memory_consumption(N, k, w):
    inode = 0
    for i in range(1, w - 1, k + 1):
        inode += min(next_power_of_2(N), 2 ** i)
    return 2 * ((key size + 2 ** k * key size) * inode \
        + next_power_of_2(N) * (key size + value size))
```

Now if we ask to draw these two functions with fixed parameters in comparison with the actually required memory, here is what we get:

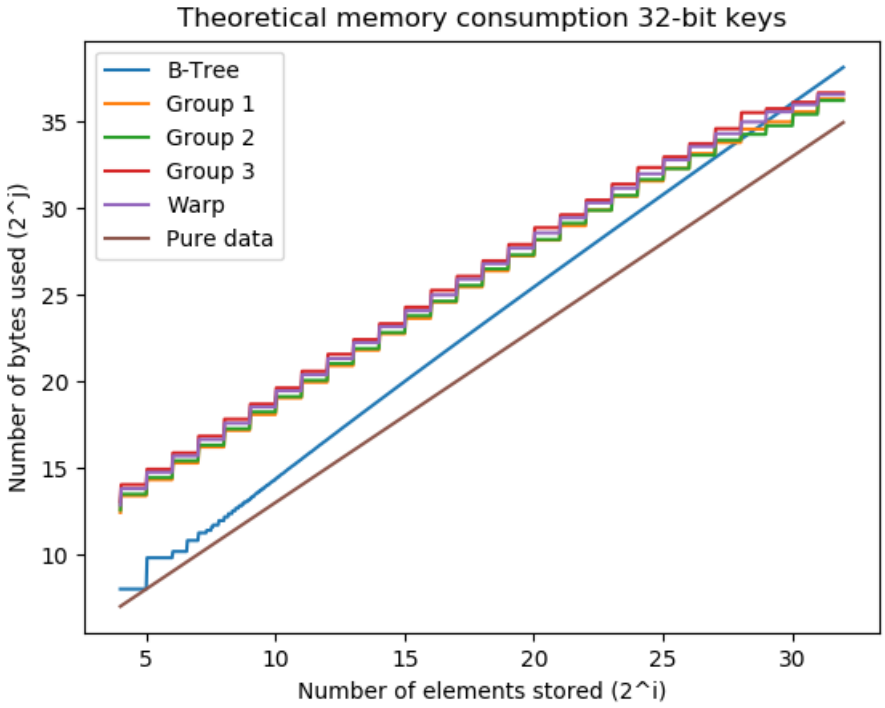


Figure 6.5: Memory consumption in bytes with $m = 32$ and 32-bit keys

We observe that X-fast tries consume proportionally much more memory than B+-trees and that it takes a lot of elements before they are competitive. At first glance, it's not obvious that this trend exists but let's not forget that B+-trees have a $O(\log_B N)$ factor compared to linear. Second, hash tables tend to have a consumption directly proportional to the number of items $\Theta(n)$. However, the problem is that we initially have a very large number of hash tables that require resizing but this proportion is decreasing since those located near the root have reached their final size and will not be able to grow any further. Here, group size of 1 and 2 manage to reduce overall memory consumption compared to other strategies for X-fast tries.

Conclusion

The data structure seems to offer some really interesting features. The insertion and search requests are much faster than the reference structure used here which proposes some common properties. However, time is wasted on requests from predecessors/successors but within acceptable time frames. It is thus really adapted if the requests for insertions or obtentions are more common than those of predecessor/successor. Note that it also seems difficult to propose insert operations or predecessor/successor requests if we only have one available thread for the X-fast tries while it remains feasible with B-tree.

But before we get any more excited, we must first realize that the results obtained for X-fast tries are terribly dependent on the hash table used. Here, enough space has been allocated in advance to avoid a possible resizing of the table that could drastically reduce performance. In our experimental case, we used an open-addressing hash table with linear probing and a 75 percent load factor. We also notice that the time to obtain elements is not linear in the number of queries, this corresponds to the fact that the number of probes required to find an element can be expressed on average as a function of the load factor. On the other hand, the B-tree node allocation policy is also very simplified and can be summarized as a single operation (an addition on a pointer).

The results were also obtained in a very particular case where only one warp was being executed; which is very far from the case of traditional use of a graphics card. This is therefore more of a proof of concept since the structure was not built to be concurrent. But let's not be so defeatist, the concurrent possibilities of X-fast tries are higher than for B-trees given the problems related to rebalancing and root management. Much literature exists on effective ways to make hash tables concurrent [72] but less on the B-trees seen their inherent difficulty [34].

Chapter 7

Hash Table

In this chapter, we will return to the primitive components of our data structure that are hash tables. We will come back to the different possible implementations and will try to compare them in the concurrent framework. This will allow us to define a hash table that can be used as a base block in our X-fast trie to test parallelism capabilities for our data structure. We will focus mainly on insertion and search capabilities based on a single thread since this kind of operations will be the most useful to us.

7.1 General context

Hash tables play an important role in many algorithms and data structures. It is a fundamental data structure which can be used at many places. They can be seen as a bunch of slots where elements will be stored and every possible item from the universe will be mapped to one of the slots thanks to a hash function. Nevertheless, there can be collisions since the number of available slots is very much smaller than the size of the universe and thus each location can have multiple items mapped to them. Classical techniques of collision resolution must be adapted to GPUs due to their nature highly parallel:

- Serialization: of the operations, indeed we cannot use locking mechanism on GPU and we must thus find solution without those and with the help of atomic operations. Those algorithms and data structures are often qualified as *lock-free* if it guarantees system-wide progress or *wait-free* if it also per-thread progress.
- Memory accesses: threads are intended to be used together to benefit from coalesced memory but the inherent nature of hash table makes it hard since the elements are randomly spread.
- Probing: another big aspect is the warp notion, all the threads will have to wait for the slowest one, hence the worst-case number of probes required.

This structure is characterized by a memory-intensive task linked to the presence of numerous random accesses and, as GPUs present a very interesting memory bandwidth, it makes them good candidates for such structures. The goal is then to make the most of the parallelism offered, either through a data parallelism between the threads, or a work as little divergent as possible between all the elements of the same warp. These

parallelism elements can also be used to simplify probing by performing all accesses related to the following probes in one treatment. Finally, it is also interesting to reduce the number of atomic operations that are carried out. These induce a not insignificant cost when there are so numerous but they are also the only primitives ensuring the validity of the accesses on GPU.

There exist several ways to construct those data structures:

- Perfect Hashing: The best way to handle $O(1)$ operations and not to have troubles with collisions is to avoid collisions thanks to perfect hashing. M. L. Fredman, J. Komlós and E. Szemerédi [53] introduced a simple construction at the price of space efficiency $O(N^2)$. The problem consists essentially to find a hash function that will not cause collisions. There are $\binom{N}{2}$ pairs who can collide each with probability N^2, hence there is $\binom{N}{2}/N^2 < \frac{1}{2}$ chance that there is a potential collision. Otherwise the other idea is to do a perfect hashing on $O(N)$ elements and do the same thing on collisions (combining with the chaining idea), this allows to make it dynamic [49].

- Open Addressing: A simple technique to implement, in order to insert an item, a series of probes is performed until an empty slot is found. There are three common strategies to probe: linear, quadratic and double hashing. Those probes must set the value atomically if it the key is empty, care must be taken with deletion. Problems also appear when the load factor is too important and the variance on the queries can be quite large. Resizing the table is also not trivial due to the expected lazy initialization of the new table [55]. The expected number of probes is bounded by $1/(1 - \frac{N}{m})$ due to the hypergeometric probability where m is the number of slots in total.

- Chaining: An alternative is to use a set of buckets and add elements in a linked-list fashion. But traversing pointers is generally not efficient. This approach can be completed by grouping the elements into larger packets [15] but variance can still be high if there are not enough buckets. Nevertheless, the possibilities of lock-free algorithms are numerous and they also guarantee the validity of iterators combined with resource management (garbage collector). The expected work is in $O(1 + \frac{N}{m})$ where m is the number of buckets/chains. On average, for all the insertions, we get: $\frac{1}{N}\sum_{i=0}^{N-1}(1 + \frac{i}{m}) = O(1 + \frac{N}{m})$

- Cuckoo: it ensures a small number of probes by limiting the number of slots an item can reside. The idea is that the elements are exchanged from one table to another each time a collision occurs. However, we may need to reconstruct the table entirely. There are some technical difficulties to implement those but they present fair results [6]. It is possible to show that if the load factor is less than $\frac{1}{2}$, the expected number of evictions is constant and the longest path is bounded by $O(\log N)$. Beyond, cycles may appear and there are specific solutions to solve those issues.

7.2 Implementation

We decided to implement different hash tables, in order to see, in practice, which models offered the best characteristics for our specific problem on graphic card. This also makes it possible to compare theoretical performance with that obtained in practice. All implementations will aim to offer the same properties with the same uniqueness of elements

and the same guarantees on reading/writing to be more fair; if there are simultaneous accesses (read and write) for the same key, the behaviour is undefined. We will mainly be interested in search and insertions operations per thread and not on a warp basis.

7.2.1 Open addressing

These hash tables are very simple to implement. We start by filling them with a sentinel value representing the absence of value. When an element is inserted, based on the index obtained by the hash function and its relative offset depending on the strategy used, it is replaced atomically by the new key if the current value is still empty. However, if insertions were limited to this treatment, the results would not be as expected. Indeed, we would be able to read values that have not yet been written, which would lead to undefined behavior. We therefore chose to work with a second sentinel value that represents a key/value being inserted. The element will be accessible after the key has been changed by its final value. The tables are static, all the memory needed to hold the keys has already been pre-allocated, so you won't have to pay extra to transfer the elements from the old table to the new one in a amortized way.

7.2.2 Chaining

For the chaining policy, we have opted for a modified model where the elements are not stored in a simple linked-list but grouped by packet through a linked-list, like the idea that can be found in data structures of type deque. When several threads attempt to access this list, they attempt to atomically increment the number of items stored in the packet to determine where they should place their result. If there is not enough space left in the packet, a new node is allocated and they restart their operation to this new node. The number of buckets was determined as being twice the number of elements to insert divided by the number of elements in a linked-list node that fits on a single cache line.

7.2.3 Cuckoo

Cuckoo hashing is more complicated despite a rather simple general concept, we insert an element at a certain position and if the position is already occupied, we expel it in order to replace it by our new element. The old one is then inserted into another table where the same process is repeated until the system stabilizes. The main advantage of this table is that the number of possible positions where an element can be located is limited by the number of tables used by this technique unlike open addressing or chaining, which can lead to degenerate cases and linear complexities. Tables of the same size have been used to simplify the design but there are variants where the tables are not all of the same size.

In our case, we have opted for two different strategies to address some of the problems inherent in this hash table. Indeed, it is very likely that we fall into a cycle where we seek to insert in a loop the same subset of elements. It is then necessary to rebuild a new table by rehashing all the elements. To limit the occurrence of such problems and to avoid this significant cost, there are several solutions: the simplest is to use more than two tables, usually three or four. Each with its own hash function and we alternate cyclically between them when inserting. The other consists in creating something close to the chaining, one cell does not store only one element at a position but several of them and when there is no more place, one is expelled, generally the last one to work as

a queue. Let's also mention the existence of "stash", a small area where we store all the keys that could not be successfully inserted [70] and its adaptation to GPU [69].

7.2.4 Remark

All these implementations are intended to be lock-free, unfortunately, if we insert this sentinel value representing an intermediate state synonymous with a current addition of an element, we have to wait until this operation is finished before we can continue our algorithm. This wait results in a primitive called *spinlock*, an active wait on the value until a new value is set, which can greatly increase the contention and therefore the access time to the elements. Care must also be taken to pay close attention to the spinlock model used. Indeed, since threads are executed by warp, it is necessary to make sure that the processing making the situation evolve is carried out before the active wait. It is therefore necessary to play on the way the operations are scheduled in order to obtain the desired behavior and not a brutal dead lock by implementing the standard schema.

7.3 Experiments

Once we have implemented all these tables, we can start by comparing their relative performances. We have therefore set up a simple experimental protocol to test the different properties and actions that interest us. Here, we focused on three types of possible actions, inserting a new element, obtaining a value and searching for an element not present in the table. Deletion operations were not considered due to the excessive analogy of the processing carried out in relation to search operations. We then ran all the experiments in the same framework, 32 warps and 32 blocks, all threads doing the same operation on different data. Finally, we took the statistics related to the execution of 30 runs and this on 10 iterations. We decided to perform the experiments under a load factor of 50%, it implies that we preallocate enough memory to hold twice our data, and for a number of operations between 65 thousands (2^{15}) up to 8 millions (2^{23}). We will represent the mean obtained with the standard deviation as a dot in the following.

7.3.1 Insertion

We will start by presenting the results obtained for the insertions:

Different results can be seen on these graphs:

- Cuckoo hashing with two tables seems to present catastrophic results compared to other models. We also decided to stop its experiment after having exceeded the million of insertions seen the time necessary for the action (attention to the log plot). A very important remark must be made a priori, we present here the mean, but the median is in reality only 1.34 times larger than for the insertion with cuckoo and 3 tables, it should lie in reality somewhere between 3 and 4 tables. This also implies that one can fall in strongly degenerated cases where the active waiting leads to much contention, which makes explode the execution time.

- The very essence of cuckoo operation leads to a much stronger sequentiality of actions, and as one may be led to transfer many table elements to table, this induces very strong adverse cases. A more adapted tuning of the parameters as well on the level of the size of the tables as in the functions of hashage employed can partially mitigate these problems.

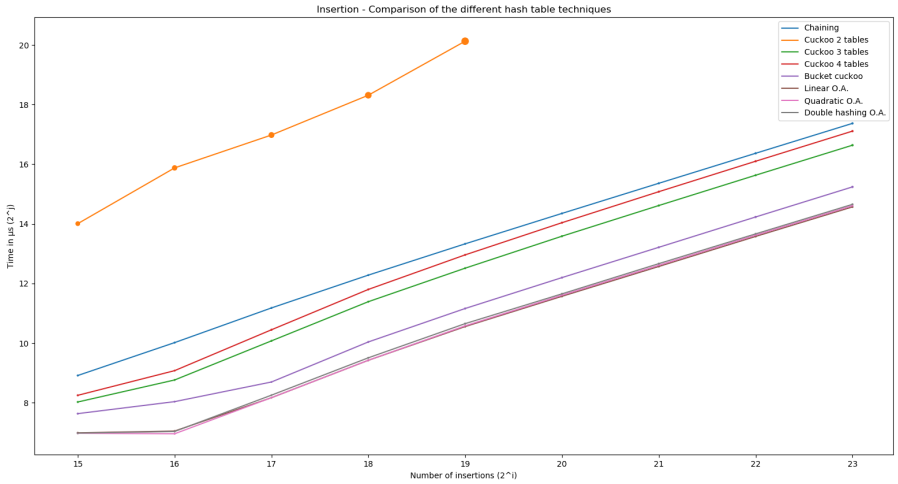


Figure 7.1: Insertions in hash table 32-bit key/value

- Chaining proposes surprisingly low performances which can be explained by the mode of access to the data and the indirection necessary to access the data, one has a low efficiency in the use of the cache.
- We can also note that the cuckoo with more than 2 tables and buckets of size 4 leads to a noticeable acceleration of performance. So, once a cached line has been loaded, its access becomes essentially free.
- Finally, those who use notions related to open addressing fare best. They even fit in a pocket handkerchief; no noticeable difference, mainly due to the relatively low load factor, 50%.

7.3.2 Search

We can then move on to the two search operations, one successful and the other looking for an element that does not exist:

Again, many remarks to be made:

- Chaining and cuckoo with bucket present the worst results of all these techniques. The performance is quite logical since these techniques are based on a similar scheme where all elements associated with a value must be observed before moving to the next node/table. A lot of processing is then done and the divergence in the same warp is all the greater as the number of possible branching may exist.
- When there are few elements, cuckoo hashing seems to offer a real alternative to open addressing with a lower standard deviation and a slightly lower but still significant average (by unpaired t-test). But on a larger number the interest seems tenuous. Indeed, the locality of the data and their relative efficiency play a crucial role in access times. Accessing the entire cache line even if loaded randomly and concurrently by each thread remains comparable to lower efficiency but with sequential accesses.

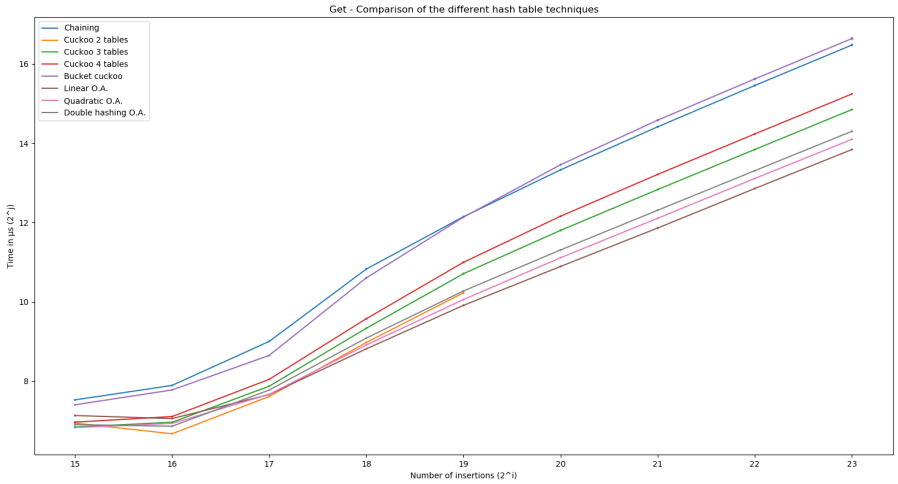


Figure 7.2: Successful search

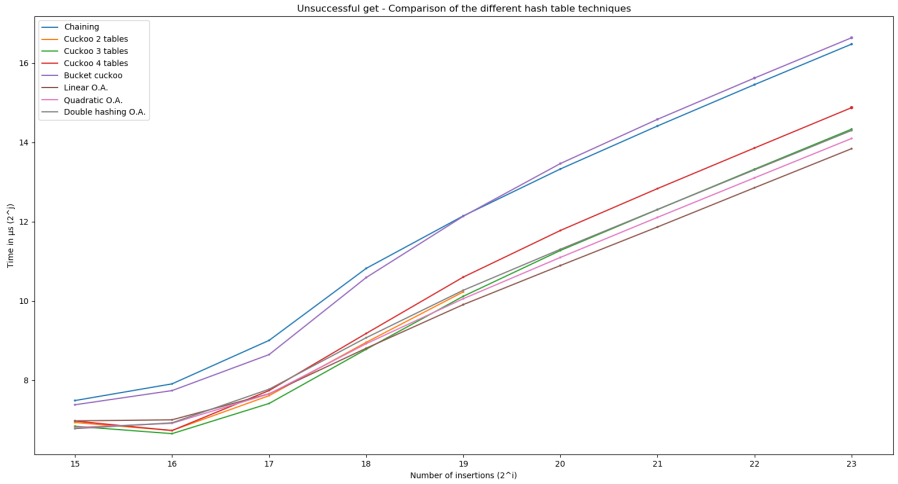


Figure 7.3: Unsuccessful search

- A notable and curious difference exists between the two cases of searches. If the element exists, cuckoo with 2 tables shows the best access times. On the contrary, if the element is absent, 3 tables seem slightly more effective. This phenomenon is somewhat bizarre and difficult to explain. It seems logical that searching in fewer tables takes less time since you have to load less data in the end but the other case is a real mystery.

- Finally, a linear probing seems to be the best in our case (50% of load factors), we then find quite naturally quadratic and double hashing since quadratic offers at the beginning, a greater probability to remain in the same cache line, whereas with double hashing, it becomes practically nil. It is funny to note that cuckoo hashing and double hashing seem to offer the same performance, which would indicate that

an equivalent number of data are read in both cases, i.e. 2.

7.3.3 Conclusions

Cuckoo hashing with just two tables is not a viable alternative in the context of graphics cards. Unless the static insertion approach is chosen as originally presented [6]. Working with 3 tables (at least) or with buckets seems a much better alternative but buckets have the disadvantage of having relatively slow access times to the elements and, in the end, one would prefer to use 3 tables if the cuckoo approach is really necessary. This has the advantage of having a bounded time on search operations compared to the other two families and the standard deviation is slightly lower, which is better suited to real time conditions. It is also the technique that can afford a high load factor without paying too much on the resulting performances.

Chaining does not seem to offer any interest by itself, if we obviously omit the possibilities of resizing the data structure and its flexibility in memory. It is much easier to offer strong guarantees on the validity of operators in this context.

Open addressing seems to be the big winner of our test, both in terms of insertions and searches. However, the removal policy is a little more complex and it may be necessary to clean this table regularly, which induces a cost in fine. Hopscotch hashing [64] presents a very interesting variant to these techniques since it consists in limiting the region in which an element can be located by exchanging the position of two elements as long as they remain sufficiently close to their original position. This would introduce a cost on insertions but would help reduce the worst case in search operations. Moreover, the linear variant of open addressing seems to win hands down in each of the tested conditions, we will opt for it when implementing our X-fast trie.

Finally, the results are more or less the same using 64-bit keys, but there is a greater decrease in performance for linear open addressing than in any other case due to the greater number of memory requests made. But the latter remains a winner in all cases. Finally, in order to give perspectives on the possible performances obtained, our implementation allows the insertion of 325 million elements per second and to recover the value associated with 572 million keys per second. See also, Heer et al. [63] which regroup all the results obtained for hash tables on GPU.

Chapter 8

Parallel X-fast tries

We have previously seen in detail the X-fast tries, their general concept and implementation strategy, in a sequential framework[5]. We can try to adapt it better to the paradigm of graphic cards and thus take advantage of all the computing power offered by them. Indeed, our implementation was for the moment purely sequential in order to see if it presented only a theoretical or real interest. Following the results, we looked at the question of making this data structure concurrent and since they are based on hash tables, it was interesting to come back to these different notions. In this chapter, we will attempt to combine these two aspects (the X-fast tries and concurrent hash tables) into one to see what could be achieved for more exploitable purposes. This will also allow us to determine whether this structure is still relevant in a highly parallel context.

8.1 General ideas

Let's take two minutes to ponder on what is absolutely necessary to make our data structure concurrent. When we insert an element, we must start by looking for the lowest node in the tree where the difference is made, where the common prefix starts to diverge between the element we want to add and all those already inserted. First, update all parent nodes as long as necessary in order to always keep the minimum and maximum value of the entire subtree, a way to access the predecessor or successor in a constant time. Second, insert all nodes that will form the path to the leaf with the increasing prefix. Finally, it remains to insert the element at leaf level and update its direct predecessor and successor.

In the remainder of this chapter, we will consider essentially bulk operations, we will concentrate first on a set of the same operations carried out simultaneously and not a mixture of these. That is to say, we will later be interested in reading, inserting and deleting operations that will take place simultaneously, interleaved.

As a preamble, we will point out that reading / writing in the different levels of the tree cannot be done simultaneously, a certain latitude is allowed due to the different positions at which an element can be located in a hash table, hence a variable time to retrieve it. This is probably the most annoying aspect of proving that the invariants are well respected.

Look for the lowest level in the tree shouldn't be a problem. Indeed, this step is limited to the membership property and it is very likely that the previous elements are fully inserted in the tree. The only embarrassing case is when levels have not yet been inserted or are currently being inserted and only a subset of them can be determined to

exist. Only, this one has a reduced impact, since we only look where the split occurs. But we must keep in mind that we can insert several elements belonging to the same subtree at the same time, or more specifically, several times the same element.

Updating parents should not be too difficult, either we practice a consistent reading policy on hash tables, that is, we can only read inserted elements, in this case, we are certain that parents exist and it is enough to update the minimum and maximum values of their subtree atomically. Either, when we read information in the hash table, we pass directly those being inserted and we do not wait for them to obtain their final value. This case can be more annoying since we can have a subset of the parent elements, we must then complete those missing or wait until they have a final value to update them.

Completing the subtree to mark the path to the leaves is not a problem as long as a special policy is put in place in the hash tables. The most attentive will have noticed the previous paragraph had the same defect and was subtly erroneous. We have completed the hash table we use to insert a new feature that is found in various implementations and is usually called *upsert* (compound word for "insert or update"). This consists simply in carrying out the same treatment as the insertion with the exception if the element is already present; in this case, a function is applied on the basis of the old and the new value and the result is written. The insertion is thus a particular case of this operation which consists only in writing the new value and discarding the old one.

The real difficulty arises when you want to insert an element at leaf level. Indeed, the operation is not limited to a simple insertion since we must update the predecessor and the successor in order to keep a coherence in the equivalent of this double linked and ordered list. This operation is far from being trivial and requires careful attention to the invariants that one wishes to keep at all times. A simple solution is to use the non-blocking technique of the philosophers' dinner, *compare and swap* as long as we have not blocked our two neighbors and update this information [89]. However, this is not enough because a precedent or successor may have been inserted in the meantime, so care must be taken to keep the smallest interval and block only those. In our case, we considered a slightly different solution where we tried to define ourselves as the predecessor and successor thanks to the "compare and swap" instruction.

8.2 Log structured merge tree (LSM)

After having finished our implementation, we wanted to compare it with another one in order to have a comparison on which to base ourselves and to note or not the interest of such a structure. We looked for a comparable data structure that offers the same features as a dictionary and that exists on the GPU. The choice was not long to decide since there is only one other offering these operations. There are works on Bounding Volume Hierarchy or R-trees but beyond the hash tables, it is somewhat lean cow. There are very few data structures specifically designed for such devices.

The Log Structured Merge tree (which we will name later LSM) is a dynamic dictionary data structure for the GPU. It was developed at the end of 2017 by S. Ashkiani, S. Li, M. Farach-Colton, N. and J. D. Owens as part of a doctoral thesis of the main author [14]. It aims to address three issues. First, it is very difficult to maintain a data structure in a dynamic context, some of the literature is devoted to creating static structure [7, 81], so a total reconstruction is re-done each time; here, updating the structure remains competitive. Second, it aims to propose operations related to dictionaries, namely the canonical ones: insertion/deletion/look up as well as range and count which

can be seen as predecessor queries if we omit one bound. Third, exploit the capabilities of the graphics card while ensuring the correctness of the structure. The invariant proposed by this data structure is very strong.

This data structure is the result of the fusion of two concepts: the Log-structured Merge-tree (LSM) [80] and the Cache Oblivious Lookahead Array (COLA) [28]. The basic idea of a LSM is to contain a set of dictionaries of increasing size. The elements are inserted in the first dictionary and when it is complete, its content is merged with the next one in the list and so on. This has two consequences, to search for an element, you have to look in each dictionary, which induces a cost of $O(\log N \log_B N)$, but to insert elements, you benefit more from the locality of the data since you merge two adjoining levels. In practice, the insertions are faster than for a B-Tree but the queries are slower. COLA, on the other hand, consists simply in having an array sorted instead of dictionaries.

The best way to exploit the parallelism offered by graphics cards for such a data structure is to work by batch. We therefore set a parameter b which corresponds to the number of elements we want to insert at each step and which is equal to the size of the first buffer in this structure. Both update operations are performed by batch, insertions and deletions. Queries do not depend on this parameter and can be performed by a single thread.

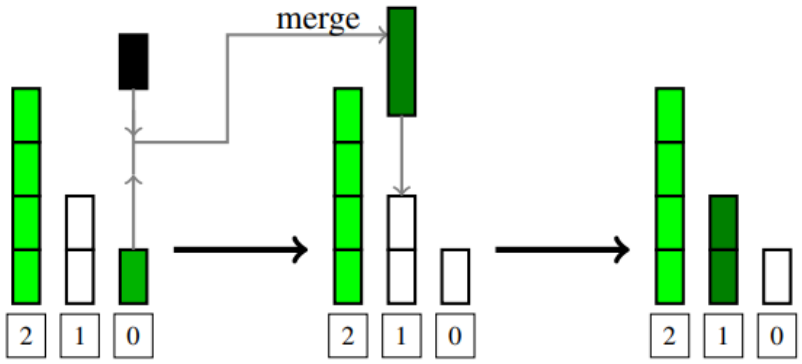


Figure 8.1: Batch insertion with already 5 inserted batches - image extracted from Ashkiani et al. [16], arXiv:1707.05354

Let's first clarify that the different dictionaries used internally by this data structure have sizes that are powers of 2 and multiples of b (i.e. $b2^i$). To insert or delete elements, we first sort the data related to the batch. Then, we always apply the same procedure, we check if the ith level is complete or empty (they cannot be partially filled), if it is empty, we insert all our elements. Otherwise, the elements to be inserted are merged with the elements of the ith level, the two containers become one and the previously filled level is emptied. We then try to insert the result in the next container, and so on until we find an empty dictionary. It should be noted that the filled levels correspond to the bit set to 1 in the binary representation of the number of batches inserted r, and 0 for the empty ones. There are thus $n = br$ elements in the structure. The total amount of work to insert n elements is thus $O(rb \log r)$ since the worst case is complete cascading $O(rb)$ and this occurs with $O(i) \leq O(\log r)$. Observe that duplicates may exist naturally and that deletions correspond to sentinel values sharing the same key. In their implementation, they propose to set the most significant bit to 1 to indicate a deletion, which will serve in the merge procedure.

Queries can be simply implemented through classic lower/upper bound queries on each filled level and leading to natural $O(\log^2 n)$. Note that merge procedures are somewhat difficult to implement in a parallel context, especially when duplicates exist [57].

8.3 Implementation

We will come back to some points linked to these data structures which seem for us important. In practice, LSMs are way easier to implement than X-fast tries.

8.3.1 X-fast trie

We started by looking to see if there was any work on how to make x-fast tries concurrent and we did not come across any results in this direction. So we tried to do what we could to solve the problem.

In practice, making the data structure concurrent does not seem to be an impossible task even if it is somewhat difficult and much attention must be paid to it. We started by making "atomic" the internal nodes of the tree, i.e. if two threads try to insert the same node into the tree, the second will necessarily apply the procedure of merging (upsert) on the data it was in charge of and the one previously inserted. The merge procedure is the same as the node update procedure, atomically replaced in order to keep the key of the minimum and maximum element of the whole subtree.

Note that node updating plays a very important role in this concurrent problem. Indeed, when one wishes to insert a node at the level of the leaves, one wants to have the smallest interval of values which exists and which contains the key of the new element. This avoids having to go through too many elements and therefore minimises the concurrency problems there, by minimising the time spent there. The first step is to insert the element with its closest neighbors into the data structure. This avoids searching for an element that does not yet exist in the leaves but is described in the intermediate nodes.

Now all that remains is to correct it and update its neighbours to indicate its existence. The same procedure is always followed until the situation is resolved. The aim is to reduce the interval while it is possible. We update the predecessor and successor values of the node we want to insert. We make a "compare and swap" on the previous node to put ourselves as successor, idem for the next one, we modify its predecessor. If both succeed, then we're done. Otherwise, the procedure is repeated again. Boundary and degenerate cases are limited by the fact that the structure provides the smallest interval ab initio.

8.3.2 LSM

We tried to obtain the LSM source code from the authors, who politely declined our request. We have therefore sought to replicate it based on the description made in their article which is nevertheless sufficiently clear and precise. We used the same two primitives to sort our data and merge dictionaries. And we hope we have provided an implementation that we feel is relatively fair and sufficiently close to what they have done. The structure being very simple, variations on possible implementations should be minor.

One of the key points of this structure is its inherent sequentiality. It seems difficult to propose a concurrent version or, at least, to have something effective. We have to wait until the treatment between two levels is completely done before we can do anything else.

As there are also very few levels in the structure, a very huge contention will be observed on the first levels making it essentially serialized. It was also easier to use two buffers to merge our data, as suggested in the original paper. So we work on a single block and on only 16 warps out of 32 available. Indeed, the library we use to sort requires some "shared" space (linked to a block) that we did not have enough to support 32 warps per block.

8.4 Experiments

For the experiments, we have again put in place a simple experimental protocol. We tested the three canonical operations: insertion, search and predecessor query, the ones that interest us the most, we can expect an equivalent time for successor or deletion requests. We compared the performance obtained for three different data structures, X-fast tries, B-trees and LSM; each under special conditions.

For the insertions, the B-trees have no concurrent version known as being really effective, we therefore reused the data structure developed in the chapter relating to X-fast tries[6.1.5] and this in a sequential context, i.e. with only one warp in only one block.

For LSM, the situation is different, first, it is difficult to provide true parallelism to this data structure given its inherent sequentiality and, second, for technical reasons, we have not been able to launch more than 16 warps related to too much shared memory use for the sort primitive used.

Finally, for the X-fast tries, the problem was quite different, crashes appeared in some rare case. Of course, we investigated the reasons for these problems, but the bugs did not seem obvious to us. Nonetheless, we have been able to carry out a sufficient number of experiments and we may hope that the results would be representative of those actually obtained.

On the other hand, for queries, there is no reason to deprive ourselves of the parallelism offered by graphics cards, so we decided to use 16 warps and 32 blocks, which should be enough to saturate the streaming processors and therefore get the maximum theoretical performance. All tests were performed with pre-allocated memory and the hash tables had a capacity equal to twice the expected number of elements, thus a load factor of 50%. We collected the results on 20 runs and 5 iterations.

8.5 Results

As usual, we will present each of the experiments that were conducted associated with its relative conclusions. The size of the points represents the standard deviation obtained in the results and their position the mean value.

We'll start with the insertions and we can observe several phenomena:

- The standard deviation is a bit lower for X-fast tries. Indeed, it seems relatively clear that their variance is only dependent on the queries performed on the internal hash tables. But the load factor is relatively low, only 50%, so it seems normal to have a narrow variance. On the contrary, we had already observed that the B-trees had a stronger tendency to have high variances related to the rebalancing necessary to maintain the structure in the tree. Finally, the LSM have an associated variance relatively small in comparison to that of the B-trees, this is related to the cascading merge which must be carried out each time one passes to the next power of 2.

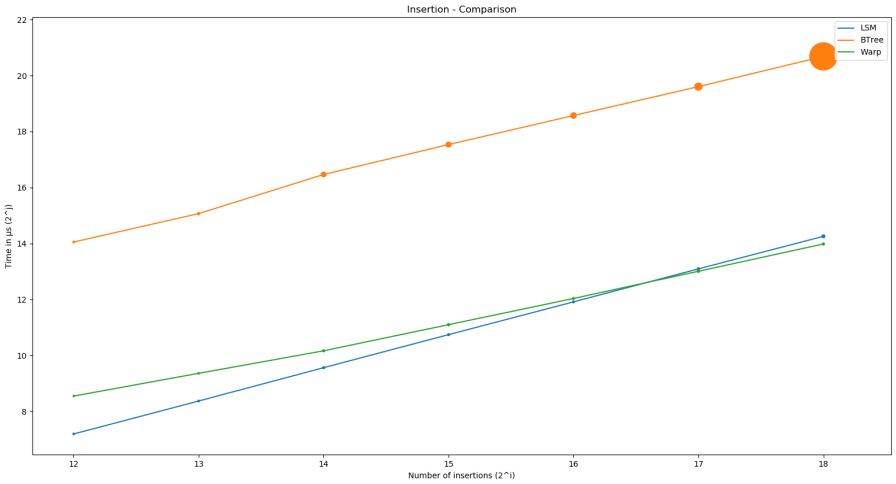


Figure 8.2: 32-bit key insertion

- The gain in concurrency that we have obtained by exploiting more warps for X-fast tries is very interesting. For both LSM and X-fast tries, insertions are almost 100 times faster than for sequential B-trees.
- The LSM offers lower starting constants but the coefficients of the larger orders seem larger. This does not seem so irrelevant since the merge operation is not totally without cost (the cost to sort the original buffer remains marginal [14]). The LSM primitives are incredibly fast but the cascading mecanism remains significant on the performances.
- In this context, insertions in X-fast tries seem to be a good alternative to LSM, following the batch size used of 512; and with greater flexibility, element-wise and not like in a bulk synchronous model. Note that the batch size presented in the original article was quite different, by a factor of more than 32 and that our implementation is not as optimal as their one.

We will continue with the two search operations:
We will start with thread-based queries:
The results are somewhat astonishing:

- Very logically, accessing X-fast tries elements is faster than for B-trees, since it consists of a simple search in a hash table, instead of running through an entire tree.
- But the difference between X-fast tries and LSM is very surprising, they are really close. The main argument is related to the fact that the locality of the data is much stronger in the LSM. The first pivots used during the dichotomous search, have strong chances to be kept in memory since they will be very often requested. We are also in the ideal situation for LSM where only one dichotomous search must be performed, since we have inserted a power of 2 elements ($b2^k$).

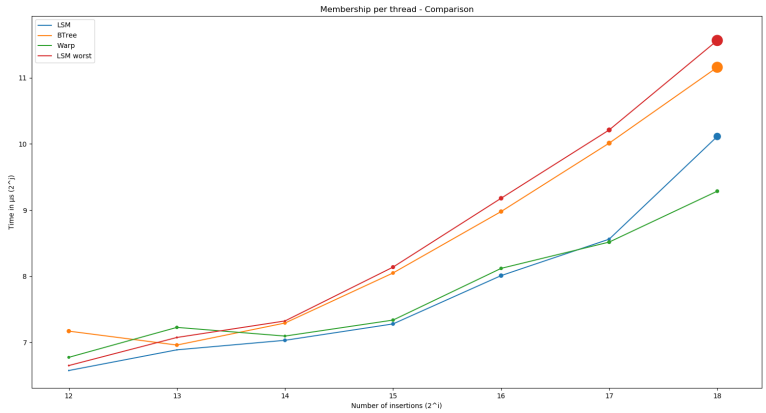


Figure 8.3: Thread-based searches

- When requests are made in the worst case for LSM, which corresponds to a number of insertions which is a power of 2 minus 1 ($b2^k - b$), the performances become comparable to that of a B-tree due to the potential $O(\log N)$ queries made. On average, the LSM remains better than the B-tree.

Now, the warp-based queries where the whole warp contributes to search one element:

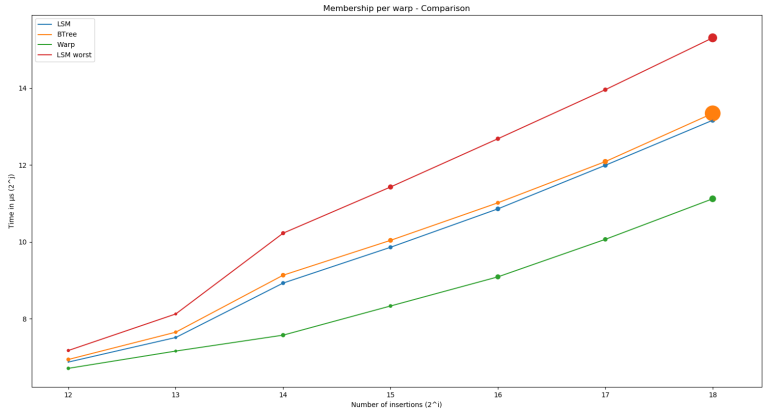


Figure 8.4: Warp-based searches

We observe similar phenomena:

- The best case for LSM is close to B-trees because dichotomous key searches no longer have to be performed. So all that remains is the cost of searching properly and loading the different blocks.
- X-fast tries retain their advantage thanks to the multiple probes performed at the same time in the leaf-level hash table.

Finally, all we have left to observe are the predecessor's queries:

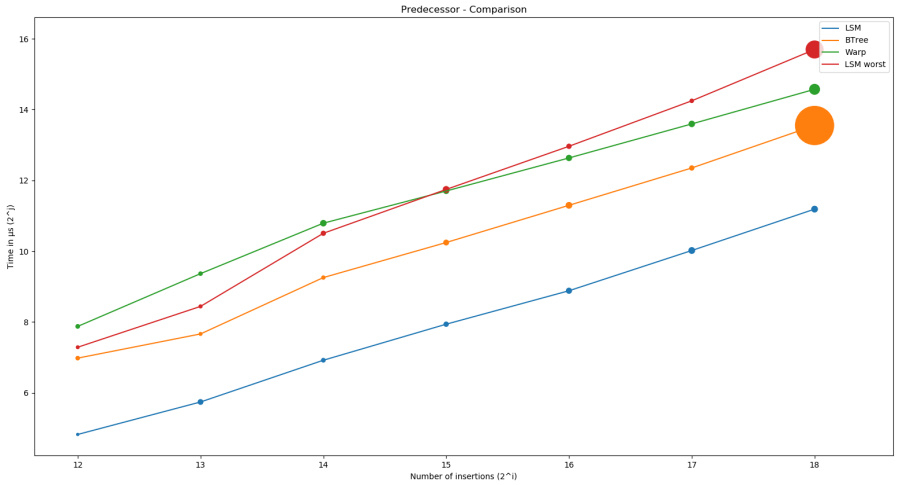


Figure 8.5: Warp-based predecessor queries

Some remarks can be made:

- We notice the same phenomenon as in the sequential framework, X-fast tries are slower than B-trees because of their imposing number of memory requests.
- For the LSM, the performance is roughly equivalent to the search operation, which is quite logical since both use lower bound algorithm.
- We precise that the predecessor search can be done by thread for LSM, whereas for X-fast tries, it can only be done by warp. So we simulated this operation by performing only one of the queries in the warp to make it comparable to other data structures.
- De facto, the time that would be taken to search for the predecessor per thread for LSM would be even faster than pseudo warp-based, and a gain of a factor 8 would not seem excessive.
- Predecessor requests in X-fast tries are faster than those in a LSM in the worst case, but on average they are significantly slower.

8.6 Speed-up

After collecting all these data and results, we asked ourselves what was the scalability of such a data structure. And mainly, what was the influence of the number of blocks and warps on the performances. Indeed, with material advances, we can expect an increase in these resources both in quantity and in speed (cost related to latency, scheduling,...). Will it remain interesting in a more distant future.

So we collected data on the insertion of 131 thousand items (2^{17}) for many configurations. We varied the number of blocks and warps from 1 to 32 and from 1 to 16 respectively by doubling each time their number. We present the mean time associated

Table 8.1: Time to insert 32-bit keys in function of the number of warps and blocks

# Warps / #Blocks	1	2	4	8	16
1	611080.18 (2151.349)	306842.126 (970.405)	154465.447 (203.045)	80001.789 (163.276)	42918.908 (81.063)
	100.0	50.21	25.28	13.09	7.02
2	316926.742 (789.772)	159116.154 (579.524)	80069.378 (101.706)	41476.388 (98.77)	22452.94 (44.331)
	51.86	26.04	13.1	6.79	3.67
4	160547.17 (356.854)	80964.003 (269.402)	40692.377 (88.208)	21198.831 (38.379)	11562.582 (31.429)
	26.27	13.25	6.66	3.47	1.89
8	81903.641 (182.339)	41682.092 (131.041)	21026.405 (63.426)	11649.702 (36.151)	7061.199 (49.369)
	13.4	6.82	3.44	1.91	1.16
16	41745.251 (124.893)	21266.58 (90.145)	11281.267 (51.34)	6674.541 (30.823)	7487.541 (43.459)
	6.83	3.48	1.85	1.09	1.23
32	21303.918 (62.376)	11469.453 (34.3)	6597.327 (17.209)	7461.54 (32.463)	5709.916 (33.266)
	3.49	1.88	1.08	1.22	0.93

with its standard deviation (in parenthesis). The intermediate lines represent the ratio between the time taken for that specific configuration and that with only one block and one warp (and therefore non-concurrent).

The results are self-explanatory:

- Generally speaking, we see the times reduced with the increase in parallelism capacities.
- The times are roughly equivalent on the diagonals, which is relatively logical. And implies that both the increase in the number of blocks and warps contribute to the overall performance improvement.
- The speed-up is far from being linear, we get more than 20% of variation from the theoretical increase for configurations such as $\#warps \times \#blocks \geq 128$. And we're almost at a factor 2 on the last configuration proposed where we reach nearly the maximum possible speed. This may be explained by the fact that we are beginning to reach the maximum possible bandwidth and that the concurrency problems on the double-linked list at the leaf level are not totally without cost.

8.7 Interleaved operations

Finally, we would like to point out that we have only considered bulk operations, where only one type of operation is carried out. This case is obviously very far from reality where a mixture is by no means exceptional. We would therefore like to come back to this point.

We clearly expect that interleaving operations and proving that implementations are correct will not be an easy task. Some mixtures don't seem to pose much of a problem at first glance. Indeed, all those who look for an element in the structure correspond to the classic case of hash tables, hence, we have the standard guarantees. Inserting elements and searching for a predecessor/successor doesn't seem too difficult either if you allow yourself a certain flexibility on the results you want to obtain.

The real concerns come with the removal of elements. Indeed, both at the level of the leaves and the maintenance of the sorted list and at the levels of the intermediate leaves and the maximum/minimum values of the subtree, problems can arise. Perhaps invalidating elements and rebuilding the structure from time to time would be a better solution.

8.8 Conclusions

As we have seen, the results we obtained in the sequential framework are surprising close from those obtained in the concurrent framework. Of course, we expected a certain loss of performance related to the very large number of memory accesses needed to respond to requests, unlike the very limited subset proposed by the LSM, but clearly not in such measures.

The theoretical performance offered by X-fast tries was really important to check and adapt to the context of the graphics cards. This data structure proposes a real alternative to LSM. We want to emphasize that we are far from the number of insertions being announced in their article since we used smaller batch sizes, but our data structure has the advantage of offering insertions by element.

One of the big black spots of this data structure is obviously the need to abuse atomic operations, and these are inherently terribly slow. We count no less than 7 million atomic transactions to insert a hundred thousand elements, however the system can provide more than 7GB/s for these operations. Nonetheless, the performances remain interesting. The other one is the amount of memory needed to hold the structure in memory which is incredibly huge. We stopped our experiments for more than 250 thousands (2^{18}) of insertions since we reach the limits of our available memory.

To resume, we achieve equivalent performance for insertions (10Mop/s) and thread-based searches (300Mop/s). We are faster on searches that use an entire warp (110Mop/s vs 30Mop/s) and we are slower on average on predecessor queries (10Mop/s vs 30Mop/s). Our data structure is of interest if the number of predecessor/successor requests remains lower than the search operations.

Nevertheless, we obtained these results in an ideal case, where all the space had been pre-allocated. One expects to become notoriously slower on insertions in practice. The load factor used in hash tables is relatively low 50%, one can also lose efficiency there.

Besides, it has the advantage of being able to improve itself thanks to scientific or material advances. It also explores a path in research where parallelism can be used at lower cost to find elements effectively. We can hope to see other more advanced structures employing similar techniques, like warp election.

To conclude, we have tried to group in a single table the main differences proposed for the only two existing dynamic dictionaries on GPUs, at this time, in order to have a more theoretical and high level aspect of what is proposed in practice.

Table 8.2: Resume comparison of LSM and X-Fast tries

	LSM	X-Fast tries
Insertion	Bulk and Block	Element and Warp
$w = \lvert\text{warp size}\rvert$	$O(\log N)$	$O(\frac{\log u}{w})$
Search	Thread	Thread
	$O(\log^2 N)$	$O(1)$
Predecessor	Thread	Warp
	$O(\log^2 N)$	$O(\frac{\log u}{w})$
Range queries	Yes	No[1]
L output	$O(\log^2 N + L)$	$O(L)$
Concurrency	Difficult	Medium
Memory	Proportional (3N)	Huge (+100N)[2]
Variance	High	Small
Requirements	Comparable keys	Only integers
Implementation	Easy	Hard

[1] Lazy iteration on the elements.
[2] Proportional to $\Theta(3N \log u)$.

Chapter 9

Conclusions

In this chapter, we will come back to the different results we obtained in order to propose a synthesis. We will try to remain as comprehensive as possible and give the main conclusions that have been elaborated in more detail in their related chapter. This aims to close the journey proposed by this work.

9.1 PEM model

We have attempted to provide a summary of the work that has been done under the PEM calculation model. We hope you have perceived the links between I/O complexity and parallelism. Y. J. Chiang had already remarked that algorithms to resolve I/O problems generally corresponded to parallel ones and that stronger connections could exist between the two. One can also see this from another point of view which is that an efficient parallel algorithm is one that effectively divides the problem in order to minimize the number of sequential parts, which results in better data separation and thus better memory usage. We also hope that you will have perceived the beauty of the ideas behind these concepts.

9.2 Memory access

To our great surprise, we discovered that the time required to access the data was far from the expected behaviour. Indeed, the access pattern can have heavy consequences on performance with factors that can be quite very important, we are talking about a magnitude of 10 or more, which was expected. But this advantage quickly fades with wider data, the adjacent access types suffer a very strong degradation of their performance while the random ones hardly see their time decreased. This may make it more attractive to perform multiple random accesses rather than the equivalent number of adjacents since the relationship is not directly linear.

It is important to understand that the loss of performance resulting from the use of random access may be compensated by greater efficiency in accessing the data thus loaded. Adjacent access remains better, but if you have no choice, it is better to use the full cache line loaded in this way, its following accesses are essentially free. You should not be afraid of the loss of raw performances since, in reality, the bandwidth remains the same and there is practically no overhead related to random access.

9.3 X-fast tries

X-fast tries offer a high performance alternative to B-trees. We managed to win orders of magnitude on some operations (insertion and search) but at a slightly higher time cost for predecessor/successor requests. It therefore has a clear advantage if the latter operations are less numerous than the more traditional insertion and search operations. Performance is also strongly linked to the primitive components of this data structure that are hash tables. One can hope that more effective implementations of these will lead to a direct increase in the performance of X-fast tries. Evolution which is more difficult for the B-trees since they are based on a more static structure which can hardly evolve.

It is also interesting to observe the application of this concept, which consists in using all threads of the same warp in order to make a query. Perhaps we will see the emergence of new data structures that are based on the existence of multithreading primitives that make several threads available at once and thus benefit from all the computational contribution offered by them.

9.4 Hash tables

We tried to compare different hash table models to see which one best matched our problem. We came to the conclusion that, surprisingly and for a load factor of 50%, using open addressing with linear probing had equivalent or slightly better performance than cuckoo hashing.

This is related to two phenomena, first, the insertion into the cuckoo hashing is essentially sequential, one is forced to wait to have expelled an element and to have taken its place before another thread can do the same thing. There is a lot of contention possible in this data structure and we saw that setting up multi-table or bucket policies really helped. Second, with such a low load factor, the number of memory cells for open addressing remains relatively small and comparable to cuckoo hashing. What's more, we benefit from the locality of the data thanks to linear probing as well as stronger guarantees on the invalidation of pointers. Of course, we expect the cuckoo hashing to become more interesting as the load factor increases. Chaining is really not a GPU-friendly technique as one would expect.

9.5 Parallel X-fast tries

One of the advantages of X-fast tries is that they are based only on hash tables. This avoids bottlenecks that can occur in tree-related data structures. One can hope to see an emergence of new concurrent hash tables, adapted to GPU, given their more capital interest in research. The possibilities of concurrence thus seem more numerous but, unfortunately, to our knowledge, there is no result on the parallelism of X-fast tries.

So we tried as best we could to offer a concurrent version of this data structure and compared its performance with the B-trees and another recently proposed data structure called LSM. But we can conceive that bulk operations would be easy to implement and prove, problems can certainly occur when mixing different operations at the same time because it will be difficult to ensure consistent reading of data among all the hash tables, one could find oneself in awkward situations with the presence of elements in some but not all levels.

After carrying out our experiments, we came to the conclusion that making X-fast tries concurrent was a more difficult task than expected, we have somehow succeeded in obtaining some results but these seem very convincing especially if we compare them to their sequential version.

Generally speaking, we achieve quite interesting results since we are competitive for the insertions or the searches per warp operations, slower for the predecessor queries but faster on thread-based searches in comparison to the two data structures tested, namely LSM and B-tree. It would also be the first data structure to offer dictionary-type operations in a fully dynamic manner and capable of inserting one element at a time on GPU.

The main problems of this data structure are obviously its excessive memory consumption. Storing the data structure as well as the problem you wish to solve in practice and its additional processing in full in GPU memory is going to be difficult. Hopefully, memory page swap techniques between GPUs and CPUs can partially compensate for this problem, but performance will certainly suffer.

As well as its difficulty of implementation and formal proof. We have tried to propose an implementation that seems fair but we are far from being able to say that it is flawless. Making the data structure concurrent and correct seems to be a real challenge and one can hope that this work will give ideas to new people in this way.

Appendix A

Implementation details

This chapter will be devoted to themes that are related to the purpose of this work but that do not directly support it. This presents more technical aspects, which certainly have an impact on our conclusions, but which are a bit off. They remain important to notify in order to reproduce the experiences but less assiduous readers can stop reading here. We will start by detailing the hardware used in the experiments and then move on to the specific software used. Everything has been developed in C++-14, since this is the maximum version supported by CUDA 9.1.

A.1 Hardware

All experiments were performed on the same computer. This nearly last generation material made it possible to benefit from the lastest innovations, in particular those related to the principle of election in the warps. Here are described the components:

Table A.1: Components

Hardware	
GPU	GTX 1050 ti
CPU	AMD Ryzen 5 1600
Memory	G.SKILL Ripjaws V Series 2x8 GB
SSD	Samsung SSD 850 EVO 250GB
Motherboard	ASRock AB350M Pro4

In practice, this graphics card offers compute capabilities 6.1 with 6 multiprocessors consisting of: 128 CUDA cores for arithmetic operations, 32 special function units for single-precision floating-point transcendental functions, and 4 warp schedulers. Each multiprocessor has a shared memory of 96KB (divided into registers and shared data). There is also a L1 cache for each multiprocessor and a L2 cache shared by all multiprocessors that is used to cache accesses to local or global memory, including temporary register spills.
All this can offer, in theory, up to 2 TFLOPS with a global memory bandwidth of 112GB/s. In practice, the performances are often lower than the theoretical ones announced by the manufacturer.

A.2 Software

The version of these softwares can also have an impact on the performance obtained:

Table A.2: Software versions

Software	
OS	Microsoft Windows 10 Professional 64 bit (build 10.0.16299)
GPU driver	GeForce driver 391.35
CUDA	CUDA Toolkit v9.1.85
Compiler	Visual studio 2017 version 15.4.5 (C++)

We also tested our software under Ubuntu 17.10 but following various problems related to drivers not adapted, most of the development was done under Windows 10.

A.3 Libraries

To develop our software and experiments, we also used three main libraries:

- Catch (2.2.1) [39] is a modern, C++-native, header-only, test framework for unit-tests, TDD and BDD - using C++11, C++14, C++17 and later. It offers many test features and is very simple to use.
- hayai [62] is a C++ framework for writing benchmarks for pieces of code. This library has been very useful for us to collect the performances and timings of our experiments. It also proposes to extract useful information such as statistical moments or quantiles.
- cuda-api-wrappers [46], this library of wrappers around the Runtime API of CUDA is intended to allow us to embrace many of the features of C++ (including some C++11) for using the runtime API - but without reducing expressivity or increasing the level of abstraction (as in, e.g., the Thrust library) in more C++-idiomatic ways.

We were somewhat disappointed not to find a common library that implemented standard algorithms (copy, fill, ...) with warp or block granularity. So we also propose a framework which aims to fill these defects. We also put the code related to X-fast tries in free access in order to avoid people having to rewrite the same data structure and thus save development time. We also use two other libraries for the LSM implementation, namely CUB[1] and moderngpu[2].

[1] https://nvlabs.github.io/cub/
[2] https://github.com/moderngpu/moderngpu/wiki

Index

Bibliography

[1] Gpu performance tool. `http://www.gpuperformance.org/`. Accessed: 2018-03-22. pages 32

[2] Alok Aggarwal, Jeffrey Vitter, et al. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988. pages 8, 19, 28

[3] Deepak Ajwani and Nodari Sitchinava. Empirical evaluation of the parallel distribution sweeping framework on multicore architectures. In *European Symposium on Algorithms*, pages 25–36. Springer, 2013. pages 25

[4] Deepak Ajwani, Nodari Sitchinava, and Norbert Zeh. Geometric algorithms for private-cache chip multiprocessors. In *European Symposium on Algorithms*, pages 75–86. Springer, 2010. pages 25

[5] Deepak Ajwani, Nodari Sitchinava, and Norbert Zeh. I/o-optimal distribution sweeping on private-cache chip multiprocessors. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1114–1123. IEEE, 2011. pages 24, 25

[6] Dan A Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D Owens, and Nina Amenta. Real-time parallel hashing on the gpu. *ACM Transactions on Graphics (TOG)*, 28(5):154, 2009. pages 52, 57

[7] Dan A Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D Owens, and Nina Amenta. Building an efficient hash table on the gpu. In *Gpu Computing Gems Jade Edition*, pages 39–53. Elsevier, 2011. pages 59

[8] Richard J Anderson and Gary L Miller. A simple randomized parallel algorithm for list-ranking. *Information Processing Letters*, 33(5):269–273, 1990. pages 21

[9] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003. pages 25

[10] Lars Arge, Michael A Bender, Erik D Demaine, Bryan Holland-Minkley, and J Ian Munro. An optimal cache-oblivious priority queue and its application to graph algorithms. *SIAM Journal on Computing*, 36(6):1672–1695, 2007. pages 22

[11] Lars Arge, Gerth Stølting Brodal, and Laura Toma. On external-memory mst, sssp, and multi-way planar graph separation. In *Scandinavian Workshop on Algorithm Theory*, pages 433–447. Springer, 2000. pages 24

[12] Lars Arge, Michael T Goodrich, Michael Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 197–206. ACM, 2008. pages 9, 21, 27, 28

[13] Lars Arge, Michael T Goodrich, and Nodari Sitchinava. Parallel external memory graph algorithms. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–11. IEEE, 2010. pages 23

[14] Saman Ashkiani. *Parallel Algorithms and Dynamic Data Structures on the Graphics Processing Unit: a warp-centric approach.* PhD thesis, University of California, Davis, 2017. pages 59, 63

[15] Saman Ashkiani, Martin Farach-Colton, and John D Owens. A dynamic hash table for the gpu. *arXiv preprint arXiv:1710.11246*, 2017. pages 52

[16] Saman Ashkiani, Shengren Li, Martin Farach-Colton, Nina Amenta, and John D Owens. Gpu lsm: A dynamic dictionary data structure for the gpu. *arXiv preprint arXiv:1707.05354*, 2017. pages 60

[17] Mikhail J Atallah and Michael T Goodrich. Efficient parallel solutions to some geometric problems. *Journal of Parallel and Distributed Computing*, 3(4):492–507, 1986. pages 25

[18] Mikhail J Atallah and Michael T Goodrich. Efficient plane sweeping in parallel. In *Proceedings of the second annual symposium on Computational geometry*, pages 216–225. ACM, 1986. pages 24, 25

[19] Mikhail J Atallah and Michael T Goodrich. Parallel algorithms for some functions of two convex polygons. *Algorithmica*, 3(1):535–548, 1988. pages 25

[20] Sara S Baghsorkhi, Matthieu Delahaye, Sanjay J Patel, William D Gropp, and Wen-mei W Hwu. An adaptive performance modeling tool for gpu architectures. In *ACM Sigplan Notices*, volume 45, pages 105–114. ACM, 2010. pages 10

[21] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications*, 32(3):866–901, 2011. pages 27

[22] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Graph expansion and communication costs of fast matrix multiplication. *Journal of the ACM (JACM)*, 59(6):32, 2012. pages 27

[23] Muthu Manikandan Baskaran and Rajesh Bordawekar. Optimizing sparse matrix-vector multiplication on gpus using compile-time and run-time strategies. *IBM Reserach Report, RC24704 (W0812-047)*, 2008. pages 12

[24] Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314. ACM, 1968. pages 28

[25] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone minimal perfect hashing: searching a sorted table with o (1) accesses. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 785–794. SIAM, 2009. pages 36

[26] Michael A Bender, Gerth Stølting Brodal, Rolf Fagerberg, Riko Jacob, and Elias Vicari. Optimal sparse matrix dense vector multiplication in the i/o-model. *Theory of Computing Systems*, 47(4):934–962, 2010. pages 26

[27] Michael A Bender, Erik D Demaine, and Martin Farach-Colton. Cache-oblivious b-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005. pages 41, 42

[28] Michael A Bender, Martin Farach-Colton, Jeremy T Fineman, Yonatan R Fogel, Bradley C Kuszmaul, and Jelani Nelson. Cache-oblivious streaming b-trees. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 81–92. ACM, 2007. pages 60

[29] Michael A Bender, Jeremy T Fineman, Seth Gilbert, and Bradley C Kuszmaul. Concurrent cache-oblivious b-trees. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 228–237. ACM, 2005. pages 9

[30] Gianfranco Bilardi, Andrea Pietracaprina, Geppino Pucci, Kieran T Herley, and Paul Spirakis. Bsp versus logp. *Algorithmica*, 24(3):405–422, 1999. pages 8

[31] Guy E Blelloch. Prefix sums and their applications. 1990. pages 20

[32] Shekhar Borkar and Andrew A Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011. pages 4

[33] Prosenjit Bose, Karim Douïeb, Vida Dujmović, John Howat, and Pat Morin. Fast local searches and updates in bounded universes. *Computational Geometry*, 46(2):181–189, 2013. pages 36

[34] Anastasia Braginsky and Erez Petrank. A lock-free b+ tree. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 58–67. ACM, 2012. pages 50

[35] Gerth Stølting Brodal and Rolf Fagerberg. On the limits of cache-obliviousness. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 307–315. ACM, 2003. pages 17

[36] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 777–786. ACM, 2004. pages 10

[37] Henri Casanova, John Iacono, Ben Karsin, Nodari Sitchinava, and Volker Weichert. An efficient multiway mergesort for gpu architectures. *arXiv preprint arXiv:1702.07961*, 2017. pages 28

[38] Bryan Catanzaro, Alexander Keller, and Michael Garland. A decomposition for in-place matrix transposition. *ACM SIGPLAN Notices*, 49(8):193–206, 2014. pages 26

[39] Catch2, C++ Automated Test Cases in a Header. `https://github.com/catchorg/Catch2`. pages 73

[40] Daniel Cederman and Philippas Tsigas. A practical quicksort algorithm for graphics processors. In *Esa*, volume 8, pages 246–258. Springer, 2008. pages 21

[41] Yi-Jen Chiang, Michael T Goodrich, Edward F Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *SODA*, volume 95, pages 139–149, 1995. pages 15, 21, 23

[42] Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979. pages 42, 43

[43] Stephen A Cook and Robert A Reckhow. Time-bounded random access machines. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 73–80. ACM, 1972. pages 5

[44] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009. pages 34

[45] Thomas H Cormen, Thomas Sundquist, and Leonard F Wisniewski. Asymptotically tight bounds for performing bmmc permutations on parallel disk systems. *SIAM Journal on Computing*, 28(1):105–136, 1998. pages 18

[46] cuda-api-wrappers, Thin C++-flavored wrappers for the CUDA runtime API. `https://github.com/eyalroz/cuda-api-wrappers`. pages 73

[47] 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. In *ACM Sigplan Notices*, volume 28, pages 1–12. ACM, 1993. pages 7

[48] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. pages 7

[49] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auF der Heide, Hans Rohnert, and Robert E Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994. pages 52

[50] Jordan Fix, Andrew Wilkes, and Kevin Skadron. Accelerating braided b+ tree searches on a gpu with cuda. In *Proceedings of the 2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance (A4MMC)*, 2011. pages 42

[51] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118. ACM, 1978. pages 5, 6

[52] Ian Foster. *Designing and building parallel programs*, volume 78. Addison Wesley Publishing Company Boston, 1995. pages 14

[53] Michael L Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0 (1) worst case access time. *Journal of the ACM (JACM)*, 31(3):538–544, 1984. pages 52

[54] Michael L Fredman and Dan E Willard. Blasting through the information theoretic barrier with fusion trees. In *Proceedings of the twenty-second annual ACM symposium on Theory of Computing*, pages 1–7. ACM, 1990. pages 36

[55] Hui Gao, Jan Friso Groote, and Wim H Hesselink. Lock-free dynamic hash tables with open addressing. *Distributed Computing*, 18(1):21–42, 2005. pages 52

[56] Phillip B Gibbons. A more practical pram model. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 158–168. ACM, 1989. pages 15

[57] Oded Green, Robert McColl, and David A Bader. Gpu merge path: a gpu merging algorithm. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 331–340. ACM, 2012. pages 29, 61

[58] Gero Greiner. *Sparse Matrix Computations and their I/O Complexity.* PhD thesis, Technical University Munich, 2012. pages 18, 19, 26

[59] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *siam Journal on Computing*, 13(2):338–355, 1984. pages 23

[60] Mark Harris. Optimizing cuda. *SC07: High Performance Computing With CUDA*, 2007. pages 30

[61] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with cuda. *GPU gems*, 3(39):851–876, 2007. pages 14

[62] HAYAI, Benchmarking framework. `https://github.com/nickbruun/hayai`. pages 73

[63] J Heer, H Leitte, and T Ropinski. Data-parallel hashing techniques for gpu architectures. *Eurographics Conference on Visualization (EuroVis) 2018*, 2018. pages 57

[64] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *International Symposium on Distributed Computing*, pages 350–364. Springer, 2008. pages 57

[65] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 152–163. ACM, 2009. pages 10

[66] Riko Jacob, Tobias Lieber, and Nodari Sitchinava. On the complexity of list ranking in the parallel external memory model. In *International Symposium on Mathematical Foundations of Computer Science*, pages 384–395. Springer, 2014. pages 21

[67] Hong Jia-Wei and Hsiang-Tsung Kung. I/o complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 326–333. ACM, 1981. pages 19, 28

[68] Richard M Karp. A survey of parallel algorithms for shared-memory machines. 1988. pages 6

[69] Farzad Khorasani, Mehmet E Belviranli, Rajiv Gupta, and Laxmi N Bhuyan. Stadium hashing: Scalable and flexible hashing on gpus. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*, pages 63–74. IEEE, 2015. pages 54

[70] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing*, 39(4):1543–1561, 2009. pages 54

[71] Lin Ma, Kunal Agrawal, and Roger D Chamberlain. A memory access model for highly-threaded many-core architectures. *Future Generation Computer Systems*, 30:202–215, 2014. pages 9

[72] Tobias Maier, Peter Sanders, and Roman Dementiev. Concurrent hash tables: Fast and general?(!). In *ACM SIGPLAN Notices*, volume 51, page 34. ACM, 2016. pages 50

[73] Igor L Markov. Limits on fundamental limits to computation. *Nature*, 512(7513):147–154, 2014. pages 4

[74] Kurt Mehlhorn and Stefan Näher. Bounded ordered dictionaries in o (log log n) time and o (n) space. *Information Processing Letters*, 35(4):183–189, 1990. pages 35

[75] P Micikevicius. Performance optimization: programming guidelines and gpu architecture reasons behind them. In *GPU Technology Conference, NVIDA*, 2013. pages 13

[76] Gary L Miller and John H Reif. Parallel tree contraction–part i: Fundamentals. 1989. pages 22

[77] Gary L Miller and John H Reif. Parallel tree contraction part 2: further applications. *SIAM Journal on Computing*, 20(6):1128–1147, 1991. pages 23

[78] Cristobal A Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, 15(2):285–329, 2014. pages 13

[79] Marco S Nobile, Paolo Cazzaniga, Daniela Besozzi, Dario Pescini, and Giancarlo Mauri. cutauleaping: a gpu-powered tau-leaping stochastic simulator for massive parallel analyses of biological systems. *PLoS One*, 9(3):e91963, 2014. pages 26

[80] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996. pages 60

[81] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance gpu ray tracing. In *Computer Graphics Forum*, volume 26, pages 415–424. Wiley Online Library, 2007. pages 59

[82] Mihai Pătraşcu and Mikkel Thorup. Randomization does not help searching predecessors. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 555–564. Society for Industrial and Applied Mathematics, 2007. pages 36

[83] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE, 2009. pages 28

[84] Hermes Senger, Veronica Gil-Costa, Luciana Arantes, Cesar AC Marcondes, Mauricio Marín, Liria M Sato, and Fabrício AB Silva. Bsp cost and scalability analysis for mapreduce operations. *Concurrency and Computation: Practice and Experience*, 28(8):2503–2527, 2016. pages 7

[85] Shubhabrata Sengupta, Mark Harris, and Michael Garland. Efficient parallel scan algorithms for gpus. *NVIDIA, Santa Clara, CA, Tech. Rep. NVR-2008-003*, (1):1–17, 2008. pages 20

[86] Nodari Sitchinava and Volker Weichert. Provably efficient gpu algorithms. *CoRR, abs/1306.5076*, 2013. pages 28, 29

[87] Nodari Sitchinava and Norbert Zeh. A parallel buffer tree. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 214–223. ACM, 2012. pages 25

[88] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010. pages 12

[89] Andrew S Tanenbaum. *Modern operating system*. Pearson Education, Inc, 2009. pages 59

[90] Robert E Tarjan and Uzi Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, 1985. pages 22

[91] Sain-Zee Ueng, Melvin Lathara, Sara S Baghsorkhi, and W Hwu Wen-mei. Cudalite: Reducing gpu programming complexity. In *LCPC*, volume 8, pages 1–15. Springer, 2008. pages 12

[92] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990. pages 6

[93] Leslie G Valiant. A bridging model for multi-core computing. *Journal of Computer and System Sciences*, 77(1):154–166, 2011. pages 7

[94] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information processing letters*, 6(3):80–82, 1977. pages 34

[95] Peter van Emde Boas. Thirty nine years of stratified trees. 2013. pages 35

[96] Peter van Emde Boas, Robert Kaas, and Erik Zijlstra. Design and implementation of an efficient priority queue. *Mathematical systems theory*, 10(1):99–127, 1976. pages 34

[97] Jeffrey Scott Vitter. Algorithms and data structures for external memory. *Foundations and Trends® in Theoretical Computer Science*, 2(4):305–474, 2008. pages 8

[98] Dan E Willard. Log-logarithmic worst-case range queries are possible in space θ (n). *Information Processing Letters*, 17(2):81–84, 1983. pages 35

[99] James C Wyllie. The complexity of parallel computations. Technical report, Cornell University, 1979. pages 5, 21

[100] Carl Yang, Yangzihao Wang, and John D Owens. Fast sparse matrix and sparse vector multiplication algorithm on the gpu. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 841–847. IEEE, 2015. pages 27

[101] Norbert Zeh. I/o-efficient graph algorithms. *EEF Summer School on Massive Data Sets*, 2002. pages 24