Toward Better Computation Models for Modern Machines

Dissertation

zur Erlangung des Grades des
Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

vorgelegt von

Tomasz Jurkiewicz

Saarbrücken
2013

Tag des Kolloquiums:
30. Oktober 2013

Dekan der Naturwissenschaftlich-Technischen Fakultät I:
Univ.-Prof. Dr. Mark Groves
Saarland University, Saarbrücken, Germany

Prüfungsausschuss:
Prof. Dr. Dr. h.c. Wolfgang J. Paul (Vorsitzender des Prüfungsausschusses)
Saarland University, Saarbrücken, Germany
Prof. Dr. Dr. h.c. mult. Kurt Mehlhorn
Max Planck Institute for Informatics, Saarbrücken, Germany
Prof. Dr. Ulrich Meyer
Institute for Computer Science
Goethe University Frankfurt am Main, Germany
Dr. Martin Hoefer (Akademischer Beisitzer)
Max Planck Institute for Informatics, Saarbrücken, Germany

Berichterstatter:
Prof. Dr. Dr. h.c. mult. Kurt Mehlhorn
Max Planck Institute for Informatics, Saarbrücken, Germany
Prof. Dr. Ulrich Meyer
Institute for Computer Science
Goethe University Frankfurt am Main, Germany
Prof. Dr. Sandeep Sen
Department of Computer Science & Engineering
Indian Institute of Technology, Delhi, India

In theory there is no difference between theory and practice;
In practice there is.

Jan L. A. van de Snepscheut

Abstract

Modern computers are not random access machines (RAMs). They have a memory hierarchy, multiple cores, and a virtual memory. We address the computational cost of the address translation in the virtual memory and difficulties in design of parallel algorithms on modern many-core machines.

Starting point for our work on virtual memory is the observation that the analysis of some simple algorithms (random scan of an array, binary search, heapsort) in either the RAM model or the EM model (external memory model) does not correctly predict growth rates of actual running times. We propose the VAT model (virtual address translation) to account for the cost of address translations and analyze the algorithms mentioned above and others in the model. The predictions agree with the measurements. We also analyze the VAT-cost of cache-oblivious algorithms.

In the second part of the paper we present a case study of the design of an efficient 2D convex hull algorithm for GPUs. The algorithm is based on *the ultimate planar convex hull algorithm* of Kirkpatrick and Seidel, and it has been referred to as *the first successful implementation of the QuickHull algorithm on the GPU* by Gao et al. in their 2012 paper on the 3D convex hull. Our motivation for work on modern many-core machines is the general belief of the engineering community that the theory does not produce applicable results, and that the theoretical researchers are not aware of the difficulties that arise while adapting algorithms for practical use. We concentrate on showing how the high degree of parallelism available on GPUs can be applied to problems that do not readily decompose into many independent tasks.

Zusammenfassung

Moderne Computer sind keine Random Access Machines (RAMs), da ihr Speicher hierarchisch ist und sie sowohl mehrere Rechenkerne als auch virtuellen Speicher benutzen. Wir betrachten die Kosten von Adressübersetzungen in virtuellem Speicher und die Schwierigkeiten beim Entwurf nebenläufiger Algorithmen für moderne Mehrkernprozessoren.

Am Anfang unserer Arbeit über virtuellen Speicher steht die Beobachtung, dass die Analyse einiger einfacher Algorithmen (zufällige Zugriffe in einem Array, Binärsuche, Heapsort) sowohl im RAM Modell als auch im EM (Modell für externen Speicher) die tatsächlichen asymptotischen Laufzeiten nicht korrekt wiedergibt. Um auch die Kosten der Adressübersetzung mit in die Analyse aufzunehmen, definieren wir das sogenannte VAT Modell (virtual address translation) und benutzen es, um die oben genannten Algorithmen zu analysieren. In diesem Modell stimmen die theoretischen Laufzeiten mit den Messungen aus der Praxis überein. Zudem werden die Kosten von Cache-oblivious Algorithmen im VAT Modell untersucht.

Der zweite Teil der Arbeit behandelt eine Fallstudie zur Implementierung eines effizienten Algorithmus zur Berechnung von konvexen Hüllen in 2D auf GPUs (Graphics Processing Units). Der Algorithmus basiert auf dem *ultimate planar convex hull algorithm* von Kirkpatrick und Seidel und wurde 2012 von Gao et al. in ihrer Veröffentlichung über konvexe Hüllen in 3D als die *erste erfolgreiche Implementierung des QuickHull-Algorithmus auf GPUs* bezeichnet.

Motiviert wird diese Arbeit durch den generellen Glauben der IT-Welt, dass Resultate aus der theoretischen Informatik nicht immer auf Probleme in der Praxis anwendbar sind und dass oft nicht auf die speziellen Anforderungen und Probleme eingegangen wird, die mit einer Implementierung eines Algorithmus einhergehen.

Wir zeigen, wie der hohe Grad an Parallelität, der auf GPUs verfügbar ist, für Probleme nutzbar gemacht werden kann, für die eine Zerlegung in unabhängige Teilprobleme nicht offensichtlich ist.

Diese Arbeit ist in englischer Sprache verfasst.

Acknowledgments

First, I would like to thank Kurt Mehlhorn who not only supervised me, but also let me make all the mistakes that needed to be made and then suggested how to fix them. Second, I thank all of you who in one way or another inspired me to pursue the subject of this thesis.

I am obliged to all the staff members of the Max Planck Institute for Informatics for maintaining a homely atmosphere. I am especially grateful to Krista Ames for proofreading the next couple dozen pages and making many, many useful comments.

Special thanks go to all the board game geeks around and to the people who take care of the trees growing inside the institute building.

Contents

1 Introduction

The role of models of computation in algorithmics is to provide abstractions of real machines for algorithm analysis. Models should be mathematically pleasing and have a predictive value. Both aspects are essential. If the analysis has no predictive value, it is merely a mathematical exercise. If a model is not clean and simple, researchers will not use it. The standard models for algorithm analysis are the RAM (random access machine) model [Shepherdson and Sturgis, 1963] and the EM (external memory) model [Aggarwal and Vitter, 1988] shortly summarized in section 1.1.

The RAM model is by far the most popular model. It is an abstraction of the von Neumann architecture. A computer consists of a control and a processing unit and an unbounded memory. Each memory cell can hold a word, and memory access as well as logical and arithmetic operations on words take constant time. The word length is either an explicit parameter or assumed to be logarithmic in the size of the input. The model is very simple and has a predictive value.

Modern machines have virtual memory, multiple processor cores, an extensive memory hierarchy involving several levels of cache memory, main memory, and disks. The external memory model was introduced because the RAM model does not account for the memory hierarchy, and hence, the RAM model has no predictive value for computations involving disks.

In part I we introduce an extension to the RAM model that handles virtual memory, the VAT model that is a tool for predicting asymptotic behavior of programs with low locality. The research described mostly overlaps with the content of [Jurkiewicz and Mehlhorn, 2013].

In part II we present a case study of adapting a sequential algorithm to CUDA, which is a modern massive-parallel computing device. We show how

CUDA differs from the classic parallel models, and put a groundwork for future models that would accurately describe this type of machines. Fragments of the writeup were available on the author's webpage and were cited in the followup work as [Jurkiewicz and Danilewski, 2010] and quoted in [Gao et al., 2012] as *the first successful implementation of the QuickHull algorithm on the GPU*. Current writeup is extended and updated.

In part III we mention author's other contributions, that do not fit the subject of this thesis.

1.1. The Random Access Machine and the External Memory Machine

A RAM machine consists of a central processing unit and a memory. The memory consists of cells indexed by nonnegative integers. A cell can hold a bitstring. The CPU has a finite number of registers, in particular an accumulator and an address register. In any one step, a RAM can either perform an operation (simple arithmetic or boolean operations) on its registers or access memory. In a memory access, the content of the memory cell indexed by the content of the address register is either loaded into the accumulator or written from the accumulator. Two timing models are used: in the unit-cost RAM, each operation has cost one, and the length of the bitstrings that can be stored in memory cells and registers is bounded by the logarithm of the size of the input; in the logarithmic-cost RAM, the cost of an operation is equal to the sum of the lengths (in bits) of the operands, and the contents of memory cells and registers are unrestricted.

An EM machine is a RAM with two levels of memory. The levels are referred to as cache and main memory or memory and disk, respectively. We use the terms cache and main memory. The CPU can only operate on data in the cache. Cache and main memory are each divided into blocks of B cells, and data is transported between cache and main memory in blocks. The cache has size M and hence consists of M/B blocks; the main memory is infinite in size. The analysis of algorithms in the EM-model bounds the number of CPU-steps and the number of block transfers. The time required for a block transfer is equal to the time required by $\Theta(B)$ CPU-steps. The hidden constant factor is fairly large, and therefore, the emphasis of the analysis is usually on the number of block transfers.

Part I

Virtual Memory Translation

2 The Cost of Address Translation

This research started with a simple experiment. We timed six simple programs for different input sizes, namely, permuting the elements of an array of size n, random scan of an array of size n, n random binary searches in an array of size n, heapsort of n elements, introsort[1] of n elements, and sequential scan of an array of size n. For some of the programs, e.g., sequential scan through an array and quicksort, the measured running times agree very well with the predictions of the models. *However, the running time of random scan seems to grow as $O(n \log n)$, and the running time of the binary searches seems to grow as $O\big(n \log^2 n\big)$, a blatant violation of what the models predict.* We give the details of the experiments in Section 2.1.

Why do measured and predicted running times differ? Modern computers have virtual memories. Each process has its own virtual address space $\{0, 1, 2, \ldots\}$. Whenever a process accesses memory, the virtual address has to be translated into a physical address. *The translation of virtual addresses into physical addresses incurs cost.* The translation process is usually implemented as a hardware-supported walk in a prefix tree, see Section 2.2 for details. The tree is stored in the memory hierarchy, and hence, the translation process may incur cache faults. The number of cache faults depends on the locality of memory accesses: the less local, the more cache faults.

We propose an extension of the EM model, the VAT (Virtual Address Translation) model, that accounts for the cost of address translation, see Section 2.3. We show that we may assume that the translation process makes optimal use of the cache memory by relating the cost of optimal use with the cost under the

[1]Introsort is the version of quicksort used in modern versions of the STL. For the purpose of this paper, introsort is a synonym for quicksort.

LRU strategy, see Section 2.3. We analyze a number of programs, including the six mentioned above, in the VAT model and obtain good agreement with the measured running times, see Section 2.4. We relate the cost of a cache-oblivious algorithm in the EM model to the cost in the VAT model, see Section 2.5. In particular, algorithms that do not need a tall-cache assumption incur no or little overhead. We close with some suggestions for further research and consequences for teaching, see Section 2.7.

Related Work: It is well-known in the architecture and systems community that virtual memory and address translation comes at a cost. Many textbooks on computer organization, e.g., [Hennessy and Patterson, 2007], discuss virtual memories. The papers by Drepper [Drepper, 2007, Drepper, 2008] describe computer memories, including virtual translation, in great detail. [Advanced Micro Devices, 2010] provides further implementation details.

The cost of address translation has received little attention from the algorithms community. The survey paper by N. Rahman [Rahman, 2003] on algorithms for hardware caches and TLB summarizes the work on the subject. She discusses a number of theoretical models for memory. All models discussed in [Rahman, 2003] treat address translation atomically, i.e., the translation from virtual to physical addresses is a single operation. However, this is no longer true. In 64-bit systems, the translation process is a tree walk. Our paper is the first that proposes a theoretical model for address translation and analyze algorithms in this model.

2.1. Some Puzzling Experiments

We used the following seven programs in our experiments. Let A be an array of size n.

- permute: for $j \in [n-1..0]$ do: $i := \text{random}(0..j)$; swap$(A[i], A[j])$;
- random scan: $\pi :=$ random permutation; for i from 0 to $n-1$ do: $S := S + A[\pi(i)]$;
- n binary searches for random positions in A; A is sorted for this experiment.
- heapify
- heapsort
- quicksort
- sequential scan

On a RAM, the first two, the last, and heapify are linear time $O(n)$, and the others are $O(n \log n)$. Figure 2.1 shows the measured running times for these programs divided by their RAM complexity; we refer to this quantity as *normalized operation time*. More details about our experimental methodology are available in Subsection 2.1.2. If RAM complexity is a good predictor, the normalized operation times should be approximately constant. We observe that two of the linear time programs show linear behavior, namely, sequential access and heapify, that one of the $\Theta(n \log n)$ programs shows $\Theta(n \log n)$ behavior, namely, quicksort, and that for the other programs (heapsort, repeated binary search, permute, random access), the actual running time grows faster than what the RAM model predicts.

How much faster and why?

Figure 2.1 also answers the "how much faster" part of the question. Normalized operation time seems to be a piecewise linear in the logarithm of the problem size; observe that we are using a logarithmic scale for the abscissa in this figure. For heapsort and repeated binary search, normalized operation time is almost perfectly piecewise linear, for permute and random scan, the piecewise linear should be taken with a grain of salt.[2] The pieces correspond to the memory hierarchy. *The measurements suggest that the running times of permute and random scan grow like $\Theta(n \log n)$ and the running times of heapsort and repeated binary search grow like $\Theta\big(n \log^2 n\big)$.*

2.1.1. Memory Hierarchy Does Not Explain It

We argue in this section that the memory hierarchy does not explain the experimental findings by determining the cost of the random scan of an array of size n in the EM model and relating it to the measured running time. Let s_i, $i \geqslant 0$ be the size of the i-th level C_i of the memory hierarchy; $s_{-1} = 0$. We assume $C_i \subset C_{i+1}$ for all i. Let ℓ be such that $s_\ell < n \leqslant s_{\ell+1}$, i.e., the array fits into level $\ell + 1$ but does not fit into level ℓ. For $i \leqslant \ell$, a random address is in C_i but not in C_{i-1}, with probability $(s_i - s_{i-1})/n$. Let c_i be the cost of accessing an address that is in C_i but not in C_{i-1}. The expected total cost in the external memory model is equal to

$$T_{\mathrm{EM}}(n) := n \cdot \left(\frac{n - s_\ell}{n} c_{\ell+1} + \sum_{0 \leqslant i \leqslant \ell} \frac{s_i - s_{i-1}}{n} c_i \right) = n c_{\ell+1} - \sum_{0 \leqslant i \leqslant \ell} s_i (c_{i+1} - c_i).$$

This is a piecewise linear function whose slope is $c_{\ell+1}$ for $s_\ell < n \leqslant s_{\ell+1}$. The slopes are increasing but change only when a new level of the memory hierarchy

[2] We are still working on a satisfactory explanation for the bumpy shape of the graphs for permute and random access.

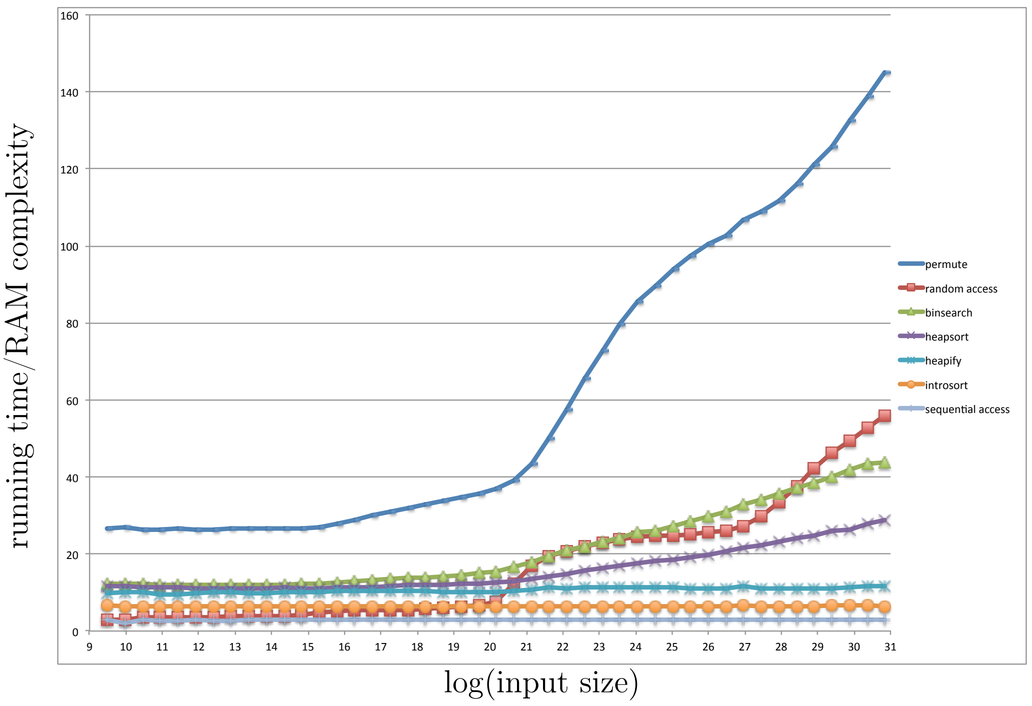

Figure 2.1. The abscissa shows the logarithm of the input size. The ordinate shows the measured running time divided by the RAM-complexity (normalized operation time). The normalized operation times of sequential access, quicksort, and heapify are constant, the normalized operation times of the other programs are not.

is used. Figure 2.2 shows the measured running time of random scan divided by EM-complexity as a function of the logarithm of the problem size. Clearly, the figure does not show the graph of a constant function.[3]

2.1.2. Methodology

Programs used for the preparation of Figure 2.1 were compiled by gcc in version "Debian 4.4.5-8", and run on Debian Linux in version 6.0.3, on a machine with an Intel Xeon X5690 processor (3,46 GHz, 12MiB[4] Smart Cache, 6,4 GT/s QPI). The caption of Figure 2.2 lists further machine parameters. In each case, we performed multiple repetitions and took the minimum measurement for each considered size of the input data. We chose the minimum because we are estimating the cost that must be incurred. We also experimented with average

[3]A function of the form $(x \log(x/a))/(bx - c)$ with $a, b, c > 0$ is convex. The plot may be interpreted as the plot of a piecewise convex function.

[4]KiB and MiB are modern, non-ambiguous notations for 2^{10*2} and 2^{10*3} bytes, respectively. For more details, refer to http://en.wikipedia.org/wiki/Binary_prefix.

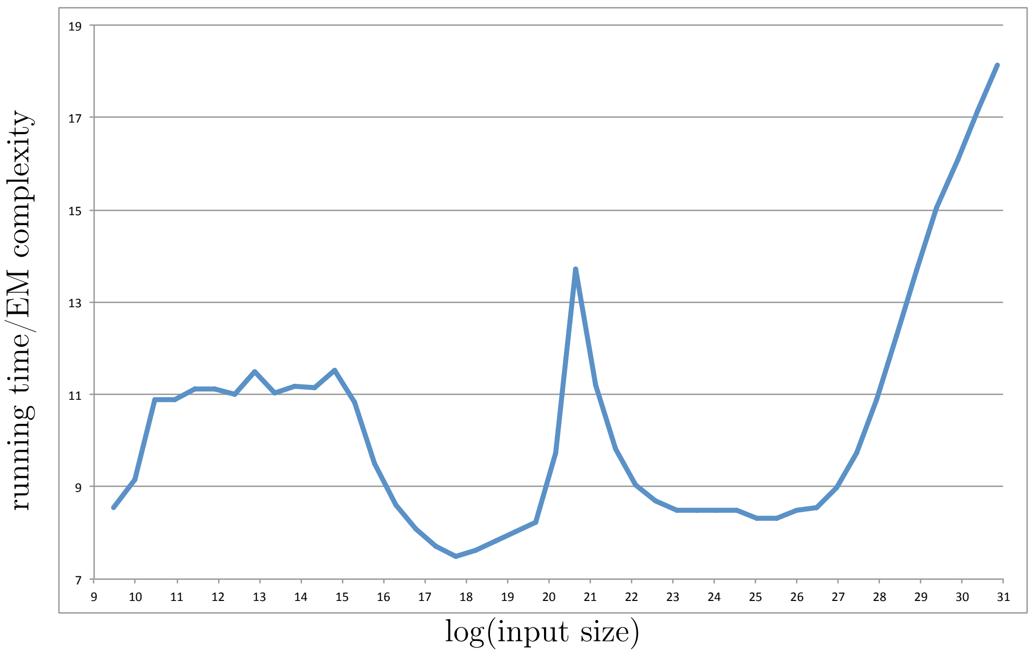

Figure 2.2. The running time of random scan divided by the EM-complexity. We used the following parameters for the memory hierarchy: the sizes are taken from the machine specification, and the access times were determined experimentally.

Memory Level	Size	log(maximum number of elements)	Access Time in Picoseconds
L1	32kiB	12	4080
L2	256kiB	15	4575
L3	12MiB	20,58	9937
RAM			38746

or median; moreover, we performed the experiments on other machines and operating systems and obtained consistent results in each case. We grew input sizes by the factor of 1.4 to exclude the influence of memory associativity, and we made sure that the largest problem size still fitted in the main memory to eliminate swapping.

For each experiment, we computed its normalized operation time, which we define as the measured execution time divided by the RAM complexity. This way, we eliminate the known factors. The resulting function represents cost of a single operation in relation to the problem size.

The canonical way of showing asymptotic equivalency of functions f and g is to show that $\lim_{x\to\infty} \frac{f(x)}{g(x)}$ is bounded between two positive constants. In case of higher polynomials, the fraction usually converges to a constant quickly enough to be clearly visible even on very rough charts. Since we deal with logarithms,

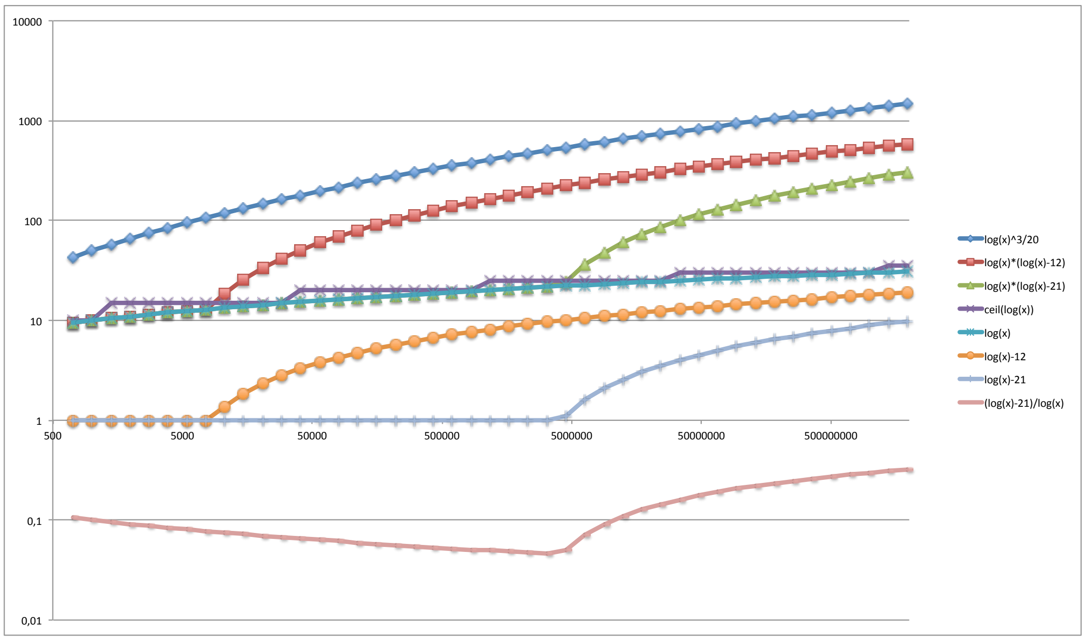

Figure 2.3. Functions of a logarithmic flavor related to our measurements. $\log(x)$ in descriptions stands for $\max(\log(x), 1)$ since running times could never be negative. Functions in the legend are ordered according to values of the functions for the greatest argument on the chart. In this ordering, the top function is $\Theta(\log^3 x)$, the next two are $\Theta(\log^2 x)$, the following four are $\Theta(\log x)$, and the last one is $\Theta(1)$. However, the shape of the functions suggests a completely different classification. Moreover, the last function shows that a $\Theta(1)$ function can be confused with a logarithm when it is composed of translated logarithms.

constant additive factors tend to zero slowly enough that they significantly influence the measurements. Therefore, even for the highest values on n possible on our machine, data is too scarce to prove asymptotic tendencies. To illustrate the issue, we present different functions of a logarithmic flavor in Figure 2.3.

Since we cannot reliably read asymptotic tendencies of measurements, we instead concentrate on the shape of the graphs on a logarithmic scale for the functions' domains. Let $\ell := \log x$, with this approach, functions of form $x \mapsto a\log^k(bx) + c$ are drawn in the form of $\ell \mapsto a\ell^k + c'$. In particular, $a\log(bn) + c$ becomes a linear function that is easy to identify. Linear plots are apparent on the chart in Figure 2.1. Our educated guess is that the slowdown is caused by the Virtual Memory Translation. Let us explain this mechanism and then argue that it indeed is the cause of our observation.

2.2. Virtual Memory

Virtual addressing was motivated by multi-processing. When several processes are executed concurrently on the same machine, it is convenient and more secure to give each program a linear address space indexed by the nonnegative integers. However, theses addresses are now virtual and no longer directly correspond to physical (real) addresses. Rather, it is the task of the operating system to map the virtual addresses of all processes to a single physical memory. The mapping process is hardware supported.

The memory is viewed as a collection of pages of $P = 2^p$ cells. Both virtual and real addresses consist of an *index* and an *offset*. The index selects a page and the offset selects a cell in a page. The index is broken into d segments of length $k = \log K$. For example, for one of the most commonly used 64 bit addressing modes on processors of the AMD64 family (see http://en.wikipedia.org/wiki/X86-64) the numbers are: $d = 4$, $k = 9$, and $p = 12$; the remaining 16 bits are used for other purposes. Logically, the translation process is a walk in a tree with outdegree K; this tree is usually called the page table [Drepper, 2008, Hennessy and Patterson, 2007]. The walk starts at the root; the first segment of the index determines the child of the root, the second segment of the index determines the child of the child, and so on. The leaves of the tree store indices of physical pages. The offset then determines the cell in the physical address, i.e., offsets are not translated but taken verbatim. Here quoting [Advanced Micro Devices, 2010]:

> "Virtual addresses are translated to physical addresses through hierarchical translation tables created and managed by system software. Each table contains a set of entries that point to the next-lower table in the translation hierarchy. A single table at one level of the hierarchy can have hundreds of entries, each of which points to a unique table at the next-lower hierarchical level. Each lower-level table can in turn have hundreds of entries pointing to tables further down the hierarchy. The lowest-level table in the hierarchy points to the translated physical page.
>
> Figure 2.4 on page 22 shows an overview of the page-translation hierarchy used in long mode. Legacy mode paging uses a subset of this translation hierarchy. As this figure shows, a virtual address is divided into fields, each of which is used as an offset into a translation table. The complete translation chain is made up of all table entries referenced by the virtual-address fields. The lowest-order virtual-address bits are used as the byte offset into the physical page."

Due to its size, the page table is stored in the RAM, but nodes accessed during the page table walk have to be brought to faster memory. A small

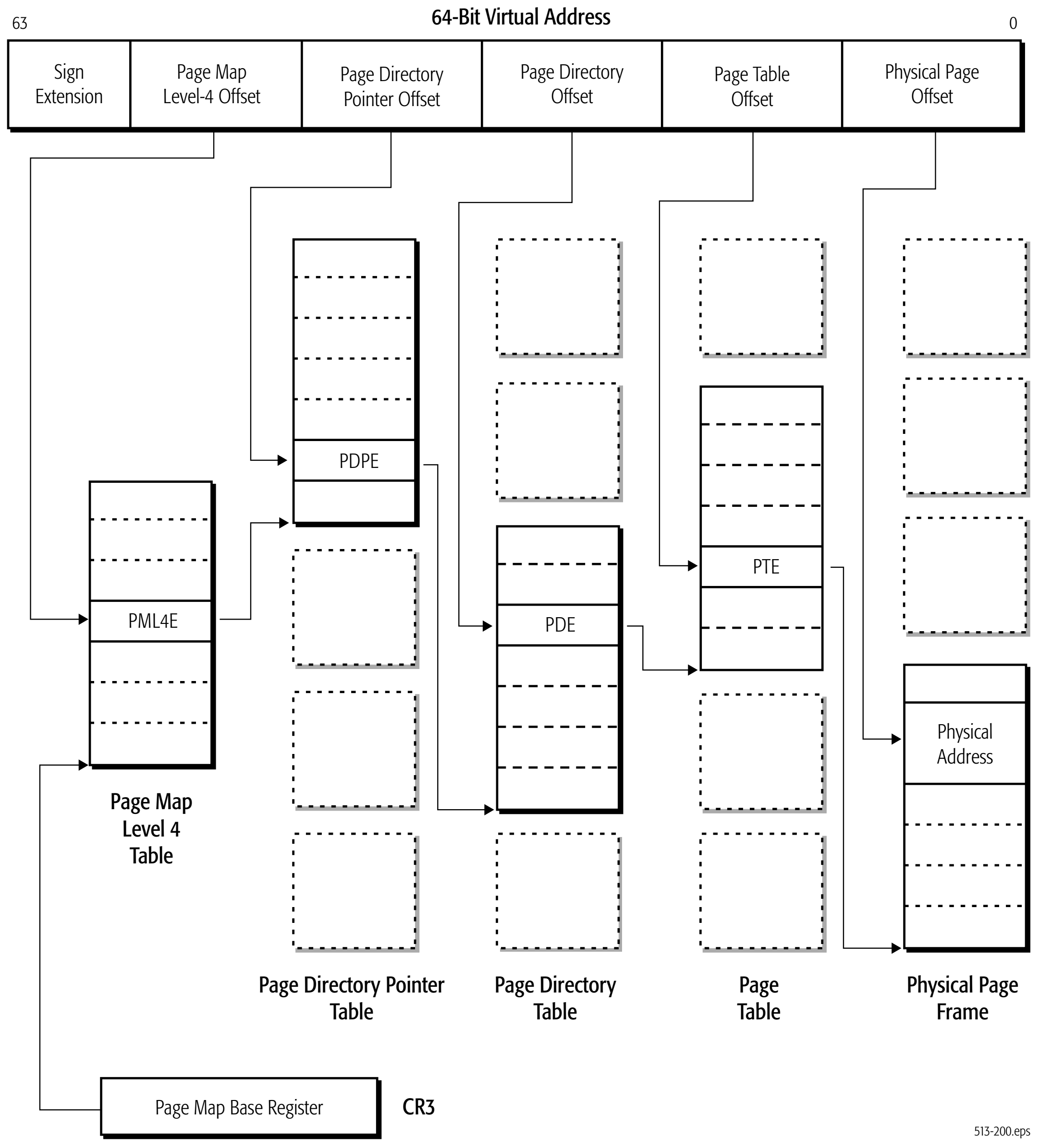

Figure 2.4. Virtual to Physical Address Translation in AMD64, figure from [Advanced Micro Devices, 2010]. Note that levels of the prefix tree have distinct historical names, as this system was originally not designed to have multiple levels (Page Map Level 4 Table; Page Directory Pointer Table; Page Directory Table; and Page Table).

number of recent translations is stored in the translation-lookaside-buffer (TLB). The TLB is a small associative memory that contains physical indices indexed by the virtual ones. This is akin to the first level cache for data. Quoting [Advanced Micro Devices, 2010] further:

> "Every memory access has its virtual address automatically translated into a physical address using the page-translation hierarchy. *Translation-lookaside buffers* (TLBs), also known as *page-translation caches*, nearly eliminate the performance penalty associated with page translation. TLBs are special on-chip caches that hold the most-recently used virtual-to-physical address translations. Each memory reference (instruction and data) is checked by the TLB. If the translation is present in the TLB, it is immediately provided to the processor, thus avoiding external memory references for accessing page tables.
>
> TLBs take advantage of the *principle of locality*. That is, if a memory address is referenced, it is likely that nearby memory addresses will be referenced in the near future."

2.2.1. Virtual Address Translation Does Explain It

The most sensible way to find out how the Virtual Address Translation affects the running time of programs would be to switch it off and compare the results. Unfortunately, no modern operating system provides such an option. The second best thing to do is to increase the page size (ultimately to a single page for the whole program) in order to decrease the number of translations and cost of each of them. This is what we did. However, while hardware architectures already support pages sized in gigabytes, operating systems do not. Quoting [Hennessy and Patterson, 2007]:

> "*Relying on the operating systems to change the page size over time.*
>
> The Alpha architects had an elaborate plan to grow the architecture over time by growing its page size, even building it into the size of its virtual address. When it came time to grow page sizes with later Alphas, the operating system designers balked and the virtual memory system was revised to grow the address space while maintaining the 8 KB page.
>
> Architects of other computers noticed very high TLB miss rates, and so added multiple, larger page sizes to the TLB. The hope was that operating systems programmers would allocate an object to the largest page that made sense, thereby preserving TLB entries. After a decade of trying, most operating systems use these "superpages" only for handpicked functions: mapping the display memory or other I/O devices, or using very large pages for the database code."

We certainly believe that operating systems will at some point be able to use big pages more frequently. However, using greater pages leads to problems. The main concern is conserving space. Pages must be correctly aligned in memory, so bigger pages lead to a greater waste of memory and limited flexibility while paging to disk. Another problem is that since most processes are small, using bigger pages would lengthen their initialization time. Therefore, current operating systems kernels provide only basic, nontransparent support for bigger pages. The *hugetlbpage* feature of current Linux kernels allowed us to use pages of moderate size of 2MiB on AMD64 machines. The following links together serve as a relatively correct and complete guide that we have used to take advantage of this feature.

- http://linuxgazette.net/155/krishnakumar.html
- https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt
- https://www.kernel.org/doc/Documentation/vm/hugepage-shm.c
- http://man7.org/linux/man-pages/man2/shmget.2.html

What this feature does is attach a final real address one level higher in the memory table. This slightly decreases cache usage, decreases the number of nodes needed in each single translation but one, and finally, increases the range of addresses covered by the related entry in the TLB by 512.

We rerun the *permute*, *introsort* and *binsearch* on a special machine, with and without use of the big pages. Figure 2.5 clearly shows that use of bigger pages can introduce a speedup. In other words, virtual address translation introduces a significant slowdown that can be partially reduced by use of the bigger pages. While for some applications this can be the only solution, for others, there are algorithmic solutions independent of the system configuration capabilities. However, one needs to be careful while constructing a model that efficiently aids with work on such algorithms. Current computers are highly sophisticated machines with many features. Each single feature requires a lot of attention to be modeled properly. We will concentrate on the virtual address translation — a feature that we believe leads to the greatest analysis discrepancies for sequential algorithms. While memory translation is not the only cost factor of a memory access, to our best knowledge, other processes are either asymptotically insignificant, or strongly correlated to the memory translation. However, there is no reason to believe that other significant cost factors will not appear in the future.

The Model Design

First and foremost, while there is a visible slowdown after L2 cache becomes too small for our data, the translation cost becomes significant only when the amount of memory in use is large enough to exceed L3 cache. Hence, we will consider only one level of memory cache. Please note that modern machines do

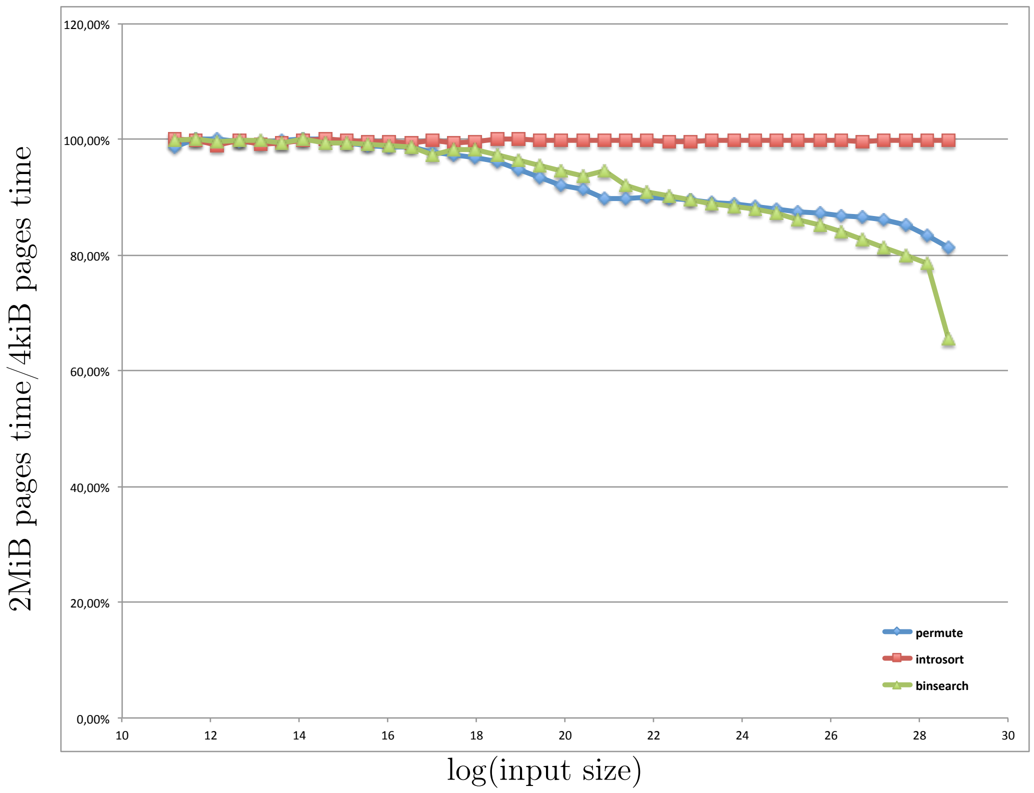

Figure 2.5. The abscissa shows the logarithm of the input size. The ordinate shows the measured running time with use of the 2MiB pages as a percentage of the running time with 4kiB pages.

not have a separate cache for the memory table (although it has been considered). Moreover, every translation is followed by a memory access, hence, one can see the RAM memory as just one additional level of the translation tree (even though the branching factor differs on the lowest level). Therefore, there is no need for a separate count of the traditional cache misses. Similarly, real machines use TLB to avoid the tree walk in the memory table altogether. For simplicity, we will only use one type of cache.

Finally, we design the model as an independent extension to the RAM model. This way, it can be coupled with other (for instance parallel) models as well, with little or no modification. The model in the current form applies to various computer architectures (even though it was developed in the context of x64 machines), too precise modeling would remove this advantage. In addition, modern computers also use other types of symbolic addresses that up to a level resemble page tables. Internet domain addresses are translated to IP addresses by DNS servers, and then to MAC addresses by routers. File paths are translated to cylinders, heads, and sectors on the disk using B-trees. While all those mechanisms differ significantly, we believe our research on virtual addressing in RAM might be of interest in these related fields as well.

2.3. VAT, The Virtual Address Translation Model

VAT machines are RAM machines that use virtual addresses. Virtual addresses were motivated by multiprocessing. If several programs are executed concurrently on the same machine, it is convenient and more secure to give each program a linear address space indexed by the nonnegative integers. However, now the addresses are virtual. They no longer correspond directly to addresses in the physical memory. Rather, the virtual memories of all running programs must be simulated with one physical memory.

We concentrate on the virtual memory of a single program. Both real (physical) and virtual addresses are strings in $\{0, \ldots, K-1\}^d \{0, \ldots, P-1\}$. The $\{0, \ldots, K-1\}^d$ part of the address is called an *index*, and its length d is an execution parameter fixed a priori to the execution. It is assumed that $d = \lceil \log_K(\text{last used address}/P) \rceil$. The $\{0, \ldots, P-1\}$ part of the address is called page offset and P is the page size. The translation process is a tree walk. We have a K-nary tree T of height d. The nodes of the tree are pairs (ℓ, i) with $\ell \geqslant 0$ and $i \geqslant 0$. We refer to ℓ as the layer of the node and to i as the number of the node. The leaves of the tree are on layer zero and a node (ℓ, i) on layer $\ell \geqslant 1$ has K children on layer $\ell-1$, namely the nodes $(\ell-1, Ki+a)$, for $a = 0 \ldots K-1$. In particular, node $(d, 0)$, the root, has children $(d-1, 0)$, $\ldots$, $(d-1, K-1)$. The leaves of the tree store page numbers of the main memory of a RAM machine. In order to translate virtual address $x_{d-1} \ldots x_0 y$, we start in the root of T, and then follow the path described by $x_{d-1} \ldots x_0$. We refer to this path as the *translation path* for the address. The path ends in the leaf $(0, \sum_{0 \leqslant i \leqslant d-1} x_i K^i)$. Let z be the page index stored in this leaf. Then $zP + y$ is the memory cell denoted by the virtual address. Observe that y is part of the real address.

The translation process uses a translation cache TC that can store W nodes of the translation tree.[5] The TC is changed by insertions and evictions. Let a be a virtual address, and let $v_d, v_{d-1}, \ldots, v_0$ be its translation path. Translating a requires accessing all nodes of the translation path in order. Only nodes in the TC can be accessed. The translation of a ends when v_0 is accessed. The next translation starts with the next operation on the TC.

The *length of the translation* is the number of insertions performed during the translation, and the *cost of the translation* is τ times the length. The length is at least the number of nodes of the translation path that are not present in the TC at the beginning of the translation.

[5]In real machines, there is no separate translation cache. Rather, the same cache is used for data and for the translation tree.

2.3.1. TC Replacement Strategies

Since the TC is a special case of a cache in a classic EM machine, the following classic result applies.

Lemma 2.1 ([Sleator and Tarjan, 1985, Frigo et al., 2012]). *An optimal replacement strategy is at most by factor 2 better than LRU*[6] *on a cache of double size, assuming both caches start empty.*

This result is useful for upper bounds and lower bounds. LRU is easy to implement. In upper bound arguments, we may use any replacement strategy and then appeal to the Lemma. In lower bound arguments, we may assume the use of LRU. For TC caches, it is natural to assume the initial segment property.

Definition 2.2. *An **initial segment** of a rooted tree is an empty tree or a connected subgraph of the tree containing the root. A TC has the **initial segment property (ISP)** if the TC contains an initial segment of the translation tree. A TC replacement strategy has ISP if under this strategy a TC has ISP at all times.*

Proposition 2.3. *Strategies with ISP exist only for TCs with $W > d$.*

ISP is important because, as we show later, ISP can be realized at no additional cost for LRU and at little additional cost for the optimal replacement strategy. Therefore, strategies with ISP can significantly simplify proofs for upper and lower bounds. Moreover, strategies with ISP are easier to implement. Any implementation of a caching system requires some way to search the cache. This requires an indexing mechanism. RAM memory is indexed by the memory translation tree. In the case of the TC itself, ISP allows to integrate the indexing structure into the cached content. One only has to store the root of the tree at a fixed position.

2.3.2. Eager Strategies and the Initial Segment Property

Before we prove an ISP analogue of Lemma 2.1, we need to better understand the behavior of replacement strategies with ISP. For classic caches, premature evictions and insertions do not improve efficiency. We will show that the same holds true for TCs with ISP. This will be useful because we will use early evictions and insertions in some of our arguments.

Definition 2.4. *A replacement strategy is **lazy** if it performs an insertion of a missing node, only if the node is accessed directly after this insertion, and performs an eviction only before an insertion for which there would be no free cell*

[6]LRU is a strategy that always evicts the Least Recently Used node.

otherwise. In the opposite case the strategy is **eager**. *Unless stated otherwise, we assume that a strategy being discussed is lazy.*

Eager strategies can perform replacements before they are needed and can even insert nodes that are not needed at all. Also, they can insert and re-evict, or evict and re-insert nodes during a single translation. We eliminate this behavior *translation by translation* as follows. Consider a fixed translation and define the sets of **effective evictions and insertions** as follows.

$$EE = \{evict(a) : \text{there are more } evict(a) \text{ than } insert(a) \text{ in the translation.}\}$$
$$EI = \{insert(a) : \text{there are more } insert(a) \text{ than } evict(a) \text{ in the translation.}\}$$

Please note that in this case “there are more” means “there is *one* more” as there cannot be two $evict(a)$ without an $insert(a)$ between them, or two $insert(a)$ without $evict(a)$.

Proposition 2.5. *The effective evictions and insertions modify the content of the TC in the same way as the original evictions and insertions.*

Proposition 2.6. *During a single translation while a strategy with ISP is in use:*

1. *No node from the current translation path is effectively evicted, and all the nodes missing from the current translation path are effectively inserted.*
2. *If a node is effectively inserted, no ancestor or descendant of it is effectively evicted. Subject to obeying the size restriction of the TC, we may therefore reorder effective insertions and effective evictions with respect to each other (but not changing the order of the insertions and not changing the order of the evictions).*

Lemma 2.7. *Any eager replacement strategy with ISP can be transformed into a lazy replacement strategy with ISP with no efficiency loss.*

Proof. We modify the original evict/insert/access sequence *translation by translation*. Consider the current translation and let EI and EE be the set of effective insertions and evictions. We insert the missing nodes from the current translation path exactly at the moment they are needed. Whenever this implies an insertion into a full cache, we perform one of the lowest effective evictions, where lowest means that no children of the node are in the TC. There must be such an effective eviction because, otherwise, the original sequence would overfull the cache as well. When all nodes of the current translation path are accessed, we schedule all remaining effective evictions and insertions at the beginning of the next translation; first the evictions in descendant-first order and then the insertions in ancestor-first order. The modified sequence is operationally equivalent to the original one, performs no more insertions, and does not exceed cache size. Moreover, the current translation is now lazy. □

2.3.3. ISLRU, or LRU with the Initial Segment Property

Even without ISP, LRU has the property below.

Lemma 2.8. *When the LRU policy is in use, the number of TC misses in a translation is equal to the layer number of the highest missing node on the translation path.*

Proof. The content of the LRU cache is easy to describe. Concatenate all translation paths and delete all occurrences of each node except the last. The last W nodes of the resulting sequence form the TC. Observe that an occurrence of a node is only deleted if the node is part of a latter translation path. This implies that the TC contains at most two incomplete translation paths, namely, the least recent path that still has nodes in the TC and the current path. The former path is evicted top-down and the latter path is inserted top-down. The claim now easily follows. Let v be the highest missing node on the current translation path. If no descendant of v is contained in the TC, the claim is obvious. Otherwise, the topmost descendant present in the TC is the first node on the part of the least recent path that is still in the TC. Thus, as the current translation path is loaded into the TC, the least recent path is evicted top-down. Consequently, the gap is never reduced. □

The proof above also shows that whenever LRU detaches nodes from the initial segment, the detached nodes will never be used again. This suggests a simple (implementable) way of introducing ISP to LRU. If LRU evicts a node that still has descendants in the TC, it also evicts the descendants. The descendants actually form a single path. Next, we use Lemma 2.7 to make this algorithm lazy again. It is easy to see that the resulting algorithm is the ISLRU, as defined next.

Definition 2.9. ***ISLRU*** *(Initial Segment preserving LRU) is the replacement strategy that always evicts the lowest descendant of the least recently used node.*

Due to the construction and Lemma 2.7, we have the following.

Proposition 2.10. *ISLRU for TCs with $W > d$ is at least as good as LRU.*

Remark 2.11. *In fact, the proposition holds also for $W \leqslant d$, even though ISLRU no longer has ISP in this case.*

2.3.4. ISMIN: The Optimal Strategy with the Initial Segment Property

Definition 2.12. ***ISMIN*** *(Initial Segment property preserving MIN) is the replacement strategy for TCs with ISP that always evicts from a TC the node that is not used for the longest time into the future among the nodes that are* *not on the current translation path and have no descendants. Nodes that will never be used again are evicted before the others in arbitrary descendant–first order.*

Theorem 2.13. ISMIN *is an optimal replacement strategy among those with ISP.*

Proof. Let R be any replacement strategy with ISP, and let t be the first point in time when it departs from ISMIN. We will construct R' with ISP that does not depart from ISMIN, including time t, and has no more TC misses than R. Let v be the node evicted by ISMIN at time t.

We first assume that R evicts v at some later time t' without accessing it in the interval $(t, t']$. Then R' simply evicts v at time t and shifts the other evictions in the interval $[t, t')$ to one later replacement. Postponing evictions to the next replacement does not cause additional insertions and does not break connectivity. It may destroy laziness by moving an eviction of a node directly before its insertion. In this case, R' skips both. Since no descendant of v is in the TC at time t, and v will not be used for the longest time into the future, none of its children will be added by R before time t'; therefore, the change does not break the connectivity.

We come to the case that R stores v until it is accessed for the next time, say at time t'. Let a be the node evicted by R at time t. R' evicts v instead of a and remembers a as being **special**. We guarantee that the content of the TCs in the strategies R and R' differs only by v and the current special node until time t' and is identical afterwords. To reach this goal, R' replicates the behavior of R except for three situations.

1. If R evicts the parent of the special node, R' evicts the special node to preserve ISP and from then on remembers the parent as being special. As long as only Rule 1 is applied, the special node is an ancestor of a.

2. If R replaces some node b with the current special node, R' skips the replacement and from then on remembers b as the special node. Since a will be accessed before v, Rule 2 is guaranteed to be applied, and hence, R' is guaranteed to save at least one replacement.

3. At time t', R' replaces the special node with v, performing one extra replacement.

We have shown how to turn an arbitrary replacement strategy with ISP into ISMIN without efficiency loss. This proves the optimality of ISMIN. □

We can now state an ISP-aware extension of Lemma 2.1.

Theorem 2.14.

$$\mathrm{MIN}(W) \leqslant \mathrm{ISMIN}(W) \leqslant \mathrm{ISLRU}(W) \leqslant \mathrm{LRU}(W) \leqslant 2\mathrm{MIN}(W/2),$$

where MIN *is an optimal replacement strategy and $A(s)$ denotes a number of insertions performed by replacement strategy A to an initially empty TC of size $s > d$ for an arbitrary but fixed sequence of translations.*

Proof. MIN is an optimal replacement strategy, so it is better than ISMIN. ISMIN is an optimal replacement strategy among those with ISP, so it is better than ISLRU. ISLRU is better than LRU by Proposition 2.10. LRU$(W) <$ 2MIN$(W/2)$ holds by Lemma 2.1. □

2.3.5. Improved Relationships

Theorem 2.14 implies LRU$(W) \leqslant$ 2ISLRU$(W/2)$ and ISMIN$(W) \leqslant$ 2MIN$(W/2)$. In this section, we sharpen both inequalities.

Lemma 2.15. LRU$(W + d) \leqslant$ ISLRU(W).

Proof. d nodes are sufficient for LRU to store one extra path, hence, from the construction, LRU on a larger cache always stores a superset of nodes stored by ISLRU. Therefore, it causes no more TC misses because it is lazy. □

Theorem 2.16. ISMIN$(W + d) \leqslant$ MIN(W).

In order to reach our goal, we will prove the following lemmas by modifying an optimal replacement strategy into intermediate strategies with no additional replacements.

Lemma 2.17. *There is an eager replacement strategy on a TC of size $W + 1$ that, except for a single special cell, has ISP and causes no more TC misses than an optimal replacement strategy on a TC of size W with no restrictions.*

Lemma 2.18. *There is a replacement strategy with ISP on a TC of size $W + d$, that causes no more TC misses than a general optimal replacement strategy on a TC of size W.*

Since ISMIN is an optimal strategy with ISP, Theorem 2.16 follows from Lemma 2.18.

In the remainder of this section, some lemmas and theorems require the assumption $W > d$, and some do not. However, even for the latter theorems, we sometimes only give the proof for the case $W > d$.

2.3.6. Belady's MIN Algorithm

Recall that Belady's algorithm MIN, also called the clairvoyant algorithm, is an optimal replacement policy. The algorithm always replaces the node that will not be accessed for the longest time into the future. An elegant optimality proof for this approach is provided in [Michaud, 2007]. MIN does not differentiate between nodes that will not be used again. Therefore, without loss of generality, let us from now on consider the descendant–first version of MIN. For any point in time, let us call all the nodes that are still to be accessed in the current translation **the required nodes**. The required nodes are exactly those nodes on the current translation path that are descendants of the last accessed node (or the whole path if the translation is only about to begin).

Lemma 2.19. *1. Let w be in the TC. As long as w has a descendant v in the TC that is not a required node,* MIN *will not evict w.*

2. If $W > d$, MIN *never evicts the root.*

3. If $W > d$, MIN *never evicts a required node.*

Proof. Ad. 1. If v will be accessed ever again, then w will be used earlier (in the same translation), and so, MIN evicts v before w. If v will never be accessed again, then MIN evicts it before w because it is the descendants–first version. Ad. 2. Either the TC stores the whole current translation path, and no eviction occurs, or there is a cell in the TC that contains a node off the current translation path; hence, the root is not evicted as it has a non-required descendant in the TC. Ad. 3. Either the TC stores the whole current translation path, or there is a cell c in the TC with content that will not be used before any required node; hence, no required node is the node that will not be needed for the longest time into the future. □

Corollary 2.20. *If $W > d$,* MIN *inserts root into the TC as a first thing during the first translation, and never evicts it.*

Lemma 2.21. *If $W > d$,* MIN *evicts only (non-required) nodes with no stored descendants or the node that was just used.*

Proof. If MIN evicts a node on the current translation path, it cannot be a descendant of the just translated node (Lemma 2.19, claim 3), it also cannot be an ancestor of the just translated node (Lemma 2.19, claim 1). Hence, only the just translated node is admissible. If the algorithm evicts a node off the current translation path, it must have no descendants (Lemma 2.19, claim 1). □

Lemma 2.22. *If* MIN *has evicted the node that was just accessed, it will continue to do so for all the following evictions in the current translation. We will refer to this as* ***round robin*** *approach.*

Proof. If MIN has evicted a node w that was just accessed, it means that all the other nodes stored in the TC will be reused before the evicted node. Moreover, all subsequent nodes traversed after w in the current translation will be reused even later than w if at all. In case of $W > d$, the claim holds by Lemma 2.21. □

Corollary 2.23. *During a single translation,* MIN *proceeds in the following way:*

1. *It starts with the* **regular phase** *when it inserts missing nodes of a connected path from the root up to some node w, as long as it can evict nodes that will not be reused before just used ones.*

2. *It switches to the* **round robin phase** *for the remaining part of the path.*

It is easy to see that for $W > d$, in the path that was traversed in the round robin fashion, informally speaking, all gaps move up by one. For each gap between stored nodes, the very TC cell that was used to store the node above the gap now stores the last node of the gap. Storage of other nodes does not change. This way, the number of nodes from this path stored in the TC does not change either. However, it reduces numbers of stored nodes on side paths attached to the path.

2.3.7. Proof of Lemma 2.17

We introduce a replacement strategy RRMIN[7]. We add a special cell rr to the TC, and we refer to the remaining W cells as **regular TC**. We will show that the cell rr allows us to preserve ISP in the regular TC with no additional TC misses. We start with an empty TC, and we run MIN on a separate TC of size W on a side and observe its decisions.

We keep track of a partial bijection[8] φ_t on the nodes of the translation tree. We put one timestamp t on every TC access and one more between every two accesses in the regular phase of MIN. We position evictions and insertions between the timestamps, at most one of each between two consecutive accesses. At time t, φ_t maps every node stored by MIN in its TC to a node stored by RRMIN in its regular TC. Function φ_t always maps nodes to (not necessarily proper) ancestors in the memory translation tree. We denote this as $\varphi_t(a) \sqsubseteq a$, and in case of proper ancestors as $\varphi_t(a) \sqsubset a$. We say that a is a witness for $\varphi_t(a)$.

Proposition 2.24. *Since the partial bijection φ_t always maps nodes to ancestors, for every path of the translation tree,* RRMIN *always stores at least as many nodes as* MIN.

In order to prove the Lemma 2.17, we need to show how to preserve the properties of the bijection φ_t and ISP. In accordance with Corollary 2.23, MIN inserts a number of highest missing nodes in the regular phase and uses the round robin approach on the remaining ones.

[7]Round Robin MIN
[8]A partial bijection on a set is a bijection between two subsets of the set.

Let us first consider the case when MIN has only the regular phase and inserts the complete path. In this case, we substitute evictions and insertions of MIN with these described below.

Let MIN evict a node a. If $\varphi_t(a)$ has no descendants, RRMIN evicts it. In the other case, we find $\varphi_t(b)$ a descendant of $\varphi_t(a)$ with no descendants on his own. RRMIN evicts $\varphi_t(b)$, and we set $\varphi_{t+1}(b) \coloneqq \varphi_t(a)$. Clearly, we have preserved the properties of φ_{t+1}[9], and ISP holds.

Now let MIN insert a new node e. At this point, we know that both RRMIN and MIN store all ancestors of e. If RRMIN has not yet stored e, RRMIN inserts it, and we set $\varphi_{t+1}(e) \coloneqq e$. If e is already stored, it means it has a witness $\varphi_t^{-1}(e)$ that is a proper descendant of e. We a find a sequence $e \sqsupset \varphi_t^{-1}(e) \sqsupset \varphi_t^{-2}(e) \sqsupset \ldots \sqsupset \varphi_t^{-k}(e) = g$ that ends with g RRMIN has not stored yet. Such g exists because φ_t^{-1} is an injection on a finite set and is undefined for e. We set $\varphi_{t+1}(h) \coloneqq h$ for all elements of the sequence except g. RRMIN inserts the highest not stored ancestor f of g, and we set $\varphi_{t+1}(g) \coloneqq f$. Note that the inserted node f might not be a required node. Properties of φ_t are preserved, and RRMIN did not disconnect the tree it stores. Also, RRMIN performed the same number of evictions and insertions as MIN. Note as well that for all nodes on the translation path, φ_t is identity. Finally, Proposition 2.24 guarantees that all accesses are safe to perform at the time they were scheduled.

Now let us consider the case when MIN has both regular and round robin phase. Assume that the regular phase ends with the visit of node v. At this point, MIN stores the (nonempty for $W > d$ due to Corollary 2.20) initial segment p_v of the current path ending in v; it does not contain v's child on the current path, and it contains some number (maybe zero) of required nodes. Starting with v's child, MIN uses the round robin strategy. Whenever it has to insert a required node, it evicts its parent. Let ℓ_r and ℓ_{rr} be the number of evictions in the regular and round robin phase, respectively.

RRMIN also proceeds in two phases. In the first phase, RRMIN simulates the regular phase as described above. RRMIN also performs ℓ_r evictions in the first phase and φ_t is the identity on p_v at the end of the first phase; this holds because φ_t maps nodes to ancestors and since MIN contains p_v in its entirety at the end of the regular phase. Let d' be the number of nodes on the current path below v; MIN stores $d' - \ell_{\mathrm{rr}}$ of them at the beginning of the round robin phase, which it does not have to insert, and it does not store ℓ_{rr} of them, which it has to insert. Since φ_t is the identity on p_v after phase 1 of the simulation and maps the $d' - \ell_{\mathrm{rr}}$ required nodes stored by MIN to ancestors, RRMIN stores at least the next $d' - \ell_{\mathrm{rr}}$ required nodes below v in the beginning of phase 2 of the simulation. In the round robin phase, RRMIN inserts the required nodes missing from the regular TC one after the other into rr, disregarding what MIN does. Whenever MIN replaces a node a with its child g, in case of $W > d$ we fix φ_t by

[9] φ_{t+1} is equal to φ_t on all arguments not explicitly specified.

setting $\varphi_{t+1}(g) := \varphi_t(a)$. By Proposition 2.24, RRMIN does no more evictions than MIN. Therefore, as it also preserves ISP in the regular TC, Lemma 2.17 holds.

2.3.8. Proof of Lemma 2.18

In order to prove the lemma, we will show how to use additional d regular cells in the TC to provide functionality of the special cell rr while preserving ISP in the whole TC. We run the RRMIN algorithm aside on a separate TC of size $W + 1$, and we introduce another replacement strategy, which we call LIS[10], on a TC of size $W + d$. LIS starts with an empty TC where d cells are marked. LIS preserves the following invariants.

1. The set of nodes stored in the unmarked cells by LIS is equal to the set of nodes stored in the regular TC by RRMIN.
2. The set of nodes stored in the marked cells by LIS contains the node stored in the cell rr by RRMIN.
3. Exactly d cells are marked.
4. LIS has ISP.
5. No node is stored twice (once marked, once unmarked).

Whenever RRMIN can replicate evictions/insertions of LIS without violating the invariants, it does. Otherwise, we consider the following cases.

1. Let RRMIN in the regular phase evict a node a that has marked descendants in LIS. Then, LIS marks the cell containing a and unmarks and evicts one of the marked nodes with no descendants that does not store the node stored in rr. Such a node exists because the only other case is that the marked cells contain all nodes of some path excluding the root, and the leaf is stored in rr. Therefore, a is the root, but the root is never evicted due to ISP.
2. In the regular phase, RRMIN inserts a node c to an empty cell while LIS already stores c in a marked cell. In this case, LIS unmarks the cell with c and marks the empty cell.
3. In the round robin phase, RRMIN replaces the content of the cell rr, LIS (if needed) replaces the content of an arbitrary marked node with no descendants that is not on the current translation path. Since the root is always in the TC and there are d marked cells, such a cell always exists. ISP is preserved, as the parent of this node is already in the TC.

[10]Lazy strategy preserving the Initial Segments property

At this stage, if we drop notions of φ_t and marked nodes, LIS becomes an eager replacement strategy on a standard TC. Therefore, we can use Lemma 2.7 to make it lazy. This concludes the proof of Lemma 2.18.

Remark 2.25. *We believe that the requirement for d is essentially optimal. Consider the scenario when we access subsequent cells uniformly at random. Informally speaking,* MIN *will tend to permanently store first $\log_K(W)$ levels of the translation tree because they are frequently used and will use a single cell to traverse the lower levels. In order to preserve ISP, we need $d - \log_K(W) + 1$ additional cells for storing the current path. Not uniform random patterns should lead to even higher requirements. This does not seem to give much more room for improvement.*

Conjecture 2.26. *The strategy of storing higher nodes (Lemma 2.17) and using extra d cells to not evict nodes from the current translation path (Lemma 2.18) can be used to add ISP to any replacement strategy without efficiency loss.*

2.4. Analysis of Algorithms

In this section, we analyze the translation cost of some algorithms as a function of the problem size n and memory requirement m. For all the algorithms analyzed, $m = \Theta(n)$.

In the RAM model, there is a crucial assumption that usually goes unspoken, namely, the size of a machine word is logarithmic in number of memory cells used. If the words were shorter, one could not address the memory. If the words were longer, one could intelligently pack multiple values in one cell. This technique can be used to solve NPC problems in polynomial time. This effectively puts an upper bound on n, namely, $n < 2^{\text{word length}}$, while asymptotic notations make sense only when n can grow to infinity. However, this is not a bound on the RAM model, it merely shows that to handle bigger inputs, one needs more powerful machines.

In the VAT model there is also a set of assumptions on the model constants. The assumptions bound n by machine parameters it in the same sense as in the RAM model. However, unlike in the RAM model, they can hardly go unspoken. We call them the *asymptotic order relations* between parameters. The assumptions we found necessary for the analysis to be meaningful are as follows:

1. $1 \leqslant \tau d \leqslant P$; moving a single translation path to the TC costs more than a single instruction, but not more than size-of-a-page instructions, i.e., if at least one instruction is performed for each cell in a page, the cost of translating the index of the page can be amortized.

2. $K \geqslant 2$, i.e., the fanout of the translation tree is at least two.

3. $m/P \leqslant K^d \leqslant 2m/P$, i.e., the translation tree suffices to translate all addresses but is not much larger. As a consequence, $\log(m/P) \leqslant d \log K = dk \leqslant 1 + \log(m/P)$, and hence, $\log_K(m/P) \leqslant d \leqslant 1/k(1 + \log(m/P))$.

4. $d \leqslant W < m^\theta$, for $\theta \in (0,1)$, i.e., the translation cache can hold at least one translation path, but is at least significantly smaller than the main memory.

Sequential Access: We scan an array of size n, i.e., we need to translate addresses b, $b+1$, $\ldots$, $b+n-1$ in this order, where b is the base address of the array. The translation path stays constant for P consecutive accesses, and hence, at most $2n/P$ indices must be translated for a total cost of at most $\tau d(2 + n/P)$. By assumption (1), this is at most $\tau d(n/P + 2) \leqslant n + 2P$.

The analysis can be sharpened significantly. We keep the current translation path in the cache, and hence, the first translation incurs at most d faults. The translation path changes after every P-th access and hence changes at most a total of $\lceil n/P \rceil$ times. Of course, whenever the path changes, the last node changes. The next to last node changes after every K-th access and hence changes at most $\lceil n/(PK) \rceil$ times. In total, we incur

$$d + \sum_{0 \leqslant i \leqslant d} \left\lceil \frac{n}{PK^i} \right\rceil < 2d + \frac{K}{K-1}\frac{n}{P}$$

TC faults. The cost is therefore bounded by $2P + 2n/d$, which is asymptotically smaller than RAM complexity.

Random Access: In the worst case, no node of any translation path is in the cache. Thus the total translation cost is bounded by τdn. This is at most $\frac{\tau}{k} n(1 + \log(n/P))$.

We will next argue a lower bound. We may assume that the TC satisfies the initial segment property. The translation path ends in a random leaf of the translation tree. For every leaf, some initial segment of the path ending in this leaf is cached. Let u be an uncached node of the translation tree of minimal depth, and let v be a cached node of maximal depth. If the depth of v is larger by two or more than the depth of u, then it is better to cache u instead of v (because more leaves use u instead of v). Thus, up to one, the same number of nodes is cached on every translation path, and hence, the expected length of the path cached is at most $\log_K W$, and hence, the expected number of faults during a translation is $d - \log_K W$. The total expected cost is therefore at least $\tau n(d - \log_K W) \geqslant \tau n \log_K n/(PW) = \frac{\tau}{k} n \log(n/(PW))$, which is asymptotically larger than RAM complexity.

Lemma 2.27. *The translation cost of a random scan of an array of size n is at least $\frac{\tau}{k} n \log(n/(PW))$ and at most $\frac{\tau}{k} n(1+\log(n/P))$.*

Binary Search: We do n binary searches in an array of length n. Each search searches for a random element of the array. For simplicity, we assume that n is a power of two minus one. A binary search in an array is equivalent to a search in a balanced tree where the root is stored in location $n/2$, the children of the root are stored in locations $n/4$ and $3n/4$, and so on. We cache the translation paths of the top ℓ layers of the search tree and the translation path of the current node of the search. The top ℓ layers contain $2^{\ell+1}-1$ vertices, and hence, we need to store at most $d2^{\ell+1}$ nodes[11] of the translation tree. This is feasible if $d2^{\ell+1} \leqslant W$. For the sequel, let $\ell = \log(W/2d)$.

Any of the remaining $\log n - \ell$ steps of the binary search cause at most d cache faults. Therefore, the total cost per search is bounded by

$$\tau d(\log n - \ell) \leqslant \frac{\tau}{k}(1+\log(n/P))(\log n - \ell) = \frac{\tau}{k} \log \frac{2n}{P} \log \frac{2nd}{W}.$$

This analysis may seem unrefined. After all once the search leaves the top ℓ layers of the search tree, addresses of subsequent nodes differ only by $n/2^{\ell}$, $n/2^{\ell+1}$, …, 1. However, we will next argue that the bound above is essentially sharp for our caching strategy. Recall that if two virtual addresses differ by D, their translation paths differ in the last $\lceil \log_K(D/P) \rceil$ nodes. Thus, the scheme above incurs at least

$$\sum_{\ell \leqslant i \leqslant \log n - p} \left\lceil \frac{1}{k} \log \frac{n}{2^i P} \right\rceil \geqslant \sum_{0 \leqslant j \leqslant \log n - \ell - p} \frac{1}{k} \log 2^j \geqslant$$
$$\geqslant \frac{1}{2k}(\log n - \ell - p)^2 = \frac{1}{2k}\left(\log \frac{2nd}{PW}\right)^2$$

TC faults. We next show that it essentially holds true for any caching strategy.

By Theorem 2.14, we may assume that ISLRU is used as the cache replacement strategy, i.e., TC contains top nodes of the recent translation paths. Let $\ell = \lceil \log(2W) \rceil$. There are $2^{\ell} \geqslant 2W$ vertices of depth ℓ in a binary search tree. Their addresses differ by at least $n/2^{\ell}$, and hence, for any two such addresses, their translation paths differ in at least the last $z = \lceil \log_K(n/(2^{\ell}P) \rceil$ nodes. Call a node at depth ℓ *expensive* if none of the last z nodes of its translation path are contained in the TC and *inexpensive* otherwise. There can be at most W inexpensive nodes, and hence, with probability at least $1/2$, a random binary search goes through an expensive node, call it v, at depth ℓ. Since ISLRU is the cache replacement strategy, the last z nodes of the translation path are missing

[11] We use vertex for the nodes of the search tree and node for the nodes of the translation tree.

for all descendants of v. Thus, by the argument in the preceding paragraph, the expected number of cache misses per search is at least

$$\frac{1}{2}\sum_{\ell\leqslant i\leqslant \log n-p}\left\lceil\frac{1}{k}\log\frac{n}{2^iP}\right\rceil \geqslant \sum_{0\leqslant j\leqslant \log n-\ell-p}\frac{1}{2k}\log 2^j \geqslant$$
$$\geqslant \frac{1}{4k}(\log n-\ell-p)^2=\frac{1}{4k}\left(\log\frac{n}{4PW}\right)^2.$$

Lemma 2.28. *The translation cost of n random binary searches in an array of size n is at most $\frac{\tau}{2k}n\left(\log\frac{2nd}{PW}\right)^2$ and at least $\frac{\tau}{4k}n\left(\log\frac{n}{4PW}\right)^2$.*

We know from cache-oblivious algorithms that the van-Emde Boas layout of a search tree improves locality. We will show in Section 2.5 that this improves the translation cost.

Heapify and Heapsort: We prove a bound on the translation cost of heapify. The following proposition generalizes the analysis of sequential scan.

Definition 2.29. ***Extremal translation paths*** *of n consecutive addresses are the paths to the first and last address in the range.* ***Non-extremal nodes*** *are the nodes on translation paths to addresses in the range that are not on the extremal paths.*

Proposition 2.30. *A sequence of memory accesses that gains access to each page in a range causes at least one TC miss for each non-extremal node of the range. If the sequence of pages in the range n is accessed in the decreasing order, this bound is matched by storing the extremal paths and dedicating $log_K(n/P)$ cells in the TC for the required translations.*

Proposition 2.31. *Let n, ℓ, and x be nonnegative integers. The number of non-extremal nodes in the union of the translation paths of any x out of n consecutive addresses is at most*

$$x\ell+\frac{2n}{PK^\ell}.$$

Moreover, there is a set of $x=\lceil n/(PK^\ell)\rceil$ addresses such that the union of the paths has size at least $x(\ell+1)+d-\ell$.

Proof. The union of the translation paths to all n addresses contains at most n/P non-extremal nodes on the leaf level (= level 0) of the translation tree. On level i, $i\geqslant 0$, from the bottom, it contains at most $n/(PK^i)$ non-extremal nodes.

We overestimate the size of the union of x translation paths by counting one node each on levels 0 to $\ell-1$ for every translation path and all non-extremal nodes contained in all the n translation paths on the levels above. Thus, the size of the union is bounded by

$$x\ell + \sum_{\ell \leqslant i \leqslant d} n/(PK^i) < x\ell + \frac{K}{K-1}\frac{n}{PK^\ell} \leqslant x\ell + \frac{2n}{PK^\ell}.$$

A node on level ℓ lies on the translation path of $K^\ell P$ consecutive addresses. Consider addresses $z + iPK^\ell$ for $i = 0, 1, \ldots, \lceil n/PK^\ell \rceil - 1$, where z is the smallest in our set of n addresses. The translation paths to these addresses are disjoint from level ℓ down to level zero and use at least one node on levels $\ell + 1$ to d. Thus, the size of the union is at least $x(\ell + 1) + d - \ell$. $\square$

An array $A[1..n]$ storing elements from an ordered set is heap-ordered if $A[i] \leqslant A[2i]$ and $A[i] \leqslant A[2i+1]$ for all i with $1 \leqslant i \leqslant \lfloor n/2 \rfloor$. An array can be turned into a heap by calling operation *sift*(i) for $i = \lfloor n/2 \rfloor$ down to 1. *sift*(i) repeatedly interchanges $z = A[i]$ with the smaller of its two children until the heap property is restored. We use the following translation replacement strategy. Let $z = \min(\log n, \lfloor (W - 2d - 1)/\lfloor \log_K(n/P) \rfloor \rfloor - 1)$. We store the extremal translation paths ($2d - 1$ nodes), non-extremal parts of the translation paths for z addresses $a_0, \ldots, a_{z-1}$, and one additional translation path a_∞ ($\lfloor \log_K(n/P) \rfloor$ nodes for each). The additional translation path is only needed when $z \neq \log n$. During the siftdown of $A[i]$, a_0 is equal to the address of $A[i]$, a_1 is the address of one of the children of i (the one to which $A[i]$ is moved, if it is moved), a_2 is the address of one of the grandchildren of i (the one to which $A[i]$ is moved, if is moved two levels down), and so on. The additional translation path a_∞ is used for all addresses that are more than z levels below the level containing i.

Let us upper bound the number of the TC misses. Preparing the extremal paths causes up to $2d + 1$ misses. Next, consider the translation cost for a_i, $0 \leqslant i \leqslant z - 1$. a_i assumes $n/2^i$ distinct values. Assuming that siblings in the heap always lie in the same page[12], the index (= the part of the address that is being translated) of each a_i decreasses over time, and hence, Proposition 2.30 bounds the number of TC misses to the number of the non-extremal nodes in the range. We use Proposition 2.31 to count them. For $i \in \{0, \ldots, p\}$, we use the Proposition with $x = n$ and $\ell = 0$ and obtain a bound of

$$\frac{2n}{P} = O\Big(\frac{n}{P}\Big)$$

TC misses. For i with $p + (\ell - 1)k < i \leqslant p + \ell k$, where $\ell \geqslant 1$ and $i \leqslant z - 1$, we use the proposition with $x = n/2^i$ and obtain a bound of at most

$$\frac{n}{2^i} \cdot \ell + \frac{2n}{PK^\ell} = O\Big(\frac{n}{2^i} \cdot \ell + \frac{2n}{2^i}\Big) = O\Big(\frac{n}{2^i}(\ell + 2)\Big) = O\Big(n\frac{i}{2^i}\Big)$$

[12]This assumption can be easily lifted by allowing an additional constant in the running time or in the TC size.

TC misses. There are $n/2^z$ siftdowns starting in layers z and above; they use a_∞. For each such siftdown, we need to translate at most $\log n$ addresses, and each translation causes less than d misses. The total is less than $n(\log n)d/2^a$. Summation yields

$$2d + 1 + (p+1)O\Big(\frac{n}{P}\Big) + \sum_{p<i\leqslant z-1} O\Big(n\frac{i}{2^i}\Big) + \frac{nd\log n}{2^z} = O\Big(d + \frac{np}{P} + \frac{nd\log n}{2^z}\Big).$$

For any realistic values of the parameters, the third term is insignificant, hence, the cost is $O\big(\tau(d + \frac{np}{P})\big)$. We next prove the corresponding lower bound under the additional assumption that $W < \frac{1}{2}n/P$. At least one address must be completely translated; hence the cost of $\Omega(\tau d)$. The addresses in $a_0 \dots a_{p-1}$ assume at least one address per page in the subarray $[n/2..n]$ because a_i can never jump by more than 2^{i+1}. First, the addresses are swept by a_0, then by a_1, and so on, and no other accesses to the subarray occur in the meantime. Hence, if the LRU strategy is in use and $W < \frac{1}{2}n/P$, there are at least $pn/(2P)$ TC misses to the lowest level of the translation tree. This gives the $\Omega\big(\frac{np}{P}\big)$ part of the misses' lower bound. Hence, the total cost is $\Omega\big(\tau(d + \frac{np}{P})\big)$.

2.5. Cache-Oblivious Algorithms

Algorithms for the EM model are allowed to use the parameters of the memory hierarchy in the program code. For any two adjacent levels of the hierarchy, there are two parameters. The size M of the faster memory and the size B of the blocks in which data is transferred between the faster and the slower memory. Cache-oblivious algorithms are formulated without reference to these parameters, i.e., they are formulated as RAM-algorithms. Only the analysis makes use of the parameters. A transfer of a block of memory is called an IO-operation. For a cache-oblivious algorithm, let $C(M, B, n)$ be the number of IO-operations on an input of size, where M is the size of the faster memory (also called cache memory) and B is the block size. Of course, $B \leqslant M$. For this class of algorithms, we have the following upper bound in VAT.

Theorem 2.32. *Consider a cache-oblivious algorithm with IO-complexity $C(M, B, n)$, where M is the size of the cache, B is the size of a block, and n is the input size. Let $a := \lfloor W/d \rfloor$, and let $P = 2^p$ be the size of a page. Then, the number of TC faults is at most*

$$\sum_{i=0}^{d} C(aK^iP, K^iP, n).$$

Proof. We divide the translation cache into d parts of size a and reserve one part for each level of the translation tree.

Consider any level i, where the leaves of the translation tree are on level 0. Each node on level i stands for K^iP addresses, and we can store a nodes. Thus, the number of faults on level i in the translation process is the same as the number of faults of the algorithm on blocks of size K^iP and a memory of a blocks (i.e., size ak^iP). Therefore, the number of TC faults is at most

$$\sum_{i=0}^{d} C(aK^iP, K^iP, n).$$

□

Theorem 2.32 allows us to rederive some of the results from Section 2.4. For example, a linear scan of an array of length n has IO-complexity at most $2 + \lfloor n/B \rfloor$. Thus, the number of TC faults is at most

$$\sum_{i=0}^{d} \Big(2 + \frac{n}{K^iP}\Big) < 2d + \frac{K}{K-1}\frac{n}{P}.$$

It also allows us to derive new results. Quicksort has IO-complexity $O((n/B)\log(n/B))$, and hence, the number of TC faults is at most

$$\sum_{i=0}^{d} O\Big(\frac{n}{K^iP}\log\frac{n}{K^iP}\Big) = O\Big(\frac{n}{P}\log\frac{n}{P}\Big).$$

Binary search in the van Emde Boas layout has IO-complexity $\log_B n$, and hence, the number of TC faults is at most

$$\sum_{i=0}^{d} \frac{\log n}{\log(K^iP)} \leqslant \frac{\log n}{\log P} + \log n \int_0^d \frac{1}{\log P + x\log K}dx$$
$$= \log_P n + \log_K n \ln \log_P(PK^d) \leqslant \log_P n + \log_K n \ln \log_P m.$$

A matrix multiplication with a recursive layout of matrices has IO-complexity $n^3/(M^{1/2}B)$, and hence, the number of TC faults is at most

$$\sum_{i=0}^{d} \frac{n^3}{(aK^iP)^{1/2}K^iP} < \frac{K^{3/2}}{K^{3/2}-1}\frac{n^3}{a^{1/2}P^{3/2}}.$$

For several fundamental algorithmic problems, e.g., sorting, FFT, matrix multiply, and searching, there are cache-oblivious algorithms that match the performance of the best EM-algorithms for the problem [Frigo et al., 2012]. These algorithms are designed such that they show good locality of reference at all scales, and therefore, one may hope that they also show good behavior in the VAT model. As one can infer, Theorem 2.32 gives good (often optimal) results for some typical problems, so the claim about good VAT efficiency can be proved for many algorithms with no effort. In some other cases, results can be even slightly improved by replacing the constant a with a more suitable sequence (a_i).

Unfortunately, some of these fundamental algorithms require the tall-cache assumption $M \geqslant B^2$. The approach we introduced does not extend to this case. The reasoning is as follows. The middle bit of the address corresponds to a chunk of memory of size $\sqrt{n}$. Hence, the tall cache assumption implies that TC must be able to store at once $\Theta(\sqrt{n})$ nodes of the level in the tree related to the middle address bits, (and exponentially more for the higher bits). TCs cannot be expected to be of such size for efficiency reasons.

On the other hand, cache-oblivious algorithms profit from the same locality principle[13] that governs the virtual address translation. Furthermore, the VAT model does not require minimization on each level of translation. Since the cost of a TC miss is equal on each layer of the translation tree, we care only about the total number of them. This gives the VAT extra flexibility that might be a sufficient substitute for the tall cache in case of some cache-oblivious algorithms.

2.6. Commentary

In this section, we describe a number of interesting subjects that extend the scope of our research. In particular, we address here the comments we received from the ALENEX13 program committee and other researchers.

2.6.1. Double Address Translation on Virtual Machines

Nowadays, more and more computation is performed on virtual machines in the clouds. In this environment, address translation must be performed twice, first to the virtual machine addressing space and then to the host. The cost of address translation to host can be as high as $O(\tau \log(\texttt{size of virtual machine}))$. Moreover, big enough virtual machines may require translation for memory tables in the virtual machine, not just for the data. This is independent of the problem input size and significant in the case of random access, but still negligible in the case of sequential access. To test the impact of the double address translation, we timed permutation and introsort on a virtual machine; results are provided in Figure 2.6.

Please note that STL introsort takes actually less time than the permutation generator, even for very small data. This is very surprising at first but means that a high VAT cost is especially harmful for programs launched on virtual machines. Since many cloud systems are meant primarily for computing, the discussed phenomenon should be of primal concern for such environments.

[13]See quotation on page 22.

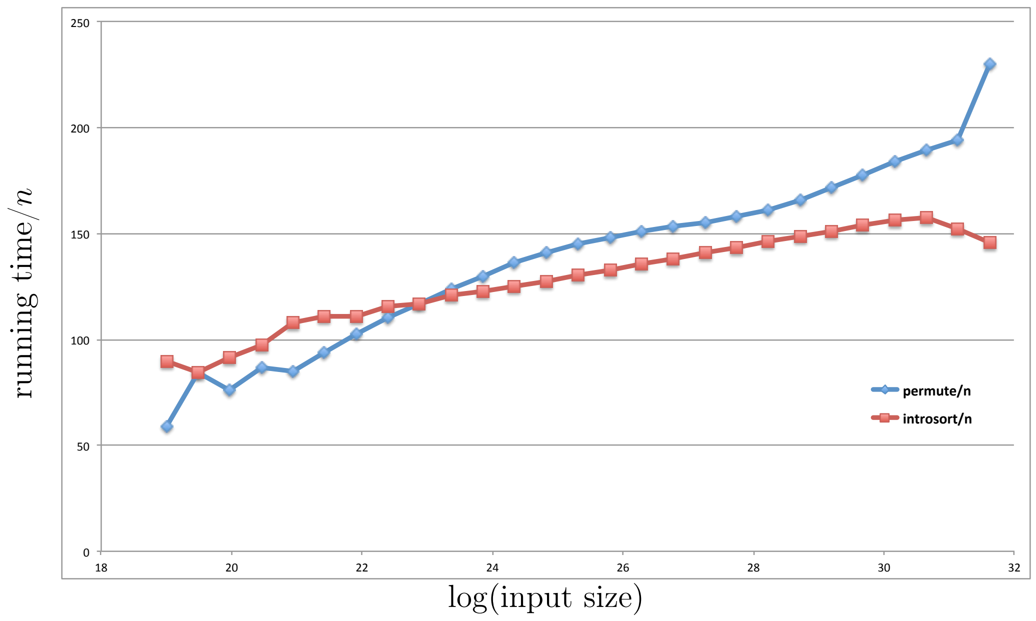

Figure 2.6. Execution times divided by n (*not* the normalized operation time) on a virtual machine with the following specification:
cpu: Intel(R) Xeon(R) CPU X5660 @ 2.67GHz
operating system: Windows 7 Enterprise
compiler: Microsoft Visual Studio 2010

2.6.2. The Model is Too Complicated

While we received comments that the model is too simple, we also received ones saying that the model is too complicated. This impression is probably due to the fact that some of our proofs are somewhat technical. Some arguments simplify if asymptotic notation is used earlier, or if the VAT cost is upper bounded by the RAM cost ahead of time (for sequential access patterns to the memory), or the other way around for the randomized access. However, as this is the first work addressing the subject, we find it appropriate to be more detailed than absolutely necessary. With time, more and more simplifications will appear. Let us briefly discuss a few candidates.

Value of K

There is evidence that for many algorithms, the exact value of K does not matter, and hence, $K = 2$ may be used. In some cases, like repeated binary search, the exact value of K seems to have only a little impact both in theory and practice. In other cases, like permutation, it seems to be the cause of bumps on the chart in Figure 2.1, but the impact is moderate. A notable exception is matrix transpose and matrix multiplication, where the value of K is blatantly visible. The classic matrix transpose algorithm uses $O(n)$ operations, where n is the input size. However, if the matrix is stored and read row by row, the output matrix must be written one element per row. For a square matrix, this means a jump of $\sqrt{n}$ cells between writes, which means $\sqrt{n}$ translations of cost $\Theta(\tau d)$ to produce the first column. As there are $\sqrt{n}$ translations before another element is written to the same row, no translation path can be reused if we consider the LRU algorithm. Therefore, the total VAT cost is $\Theta(\tau n d)$, which is $\Theta(\tau n \log n)$. Figure 2.7 shows that even though the asymptotic growth is intact, the translation cost grows in jumps rather than in a smooth logarithmic fashion. The distance between the jumps appears to be directly related to the value of K, namely, the jump occurs when the matrix dimension is K times greater than during the previous jump. Note that the EM cost of this algorithm is $\Theta(n)$ for $\sqrt{n} \cdot B > M$, and $\Theta(n/B)$ for $\sqrt{n} \cdot B < M$. In fact, the first cost jump is due to this barrier itself.

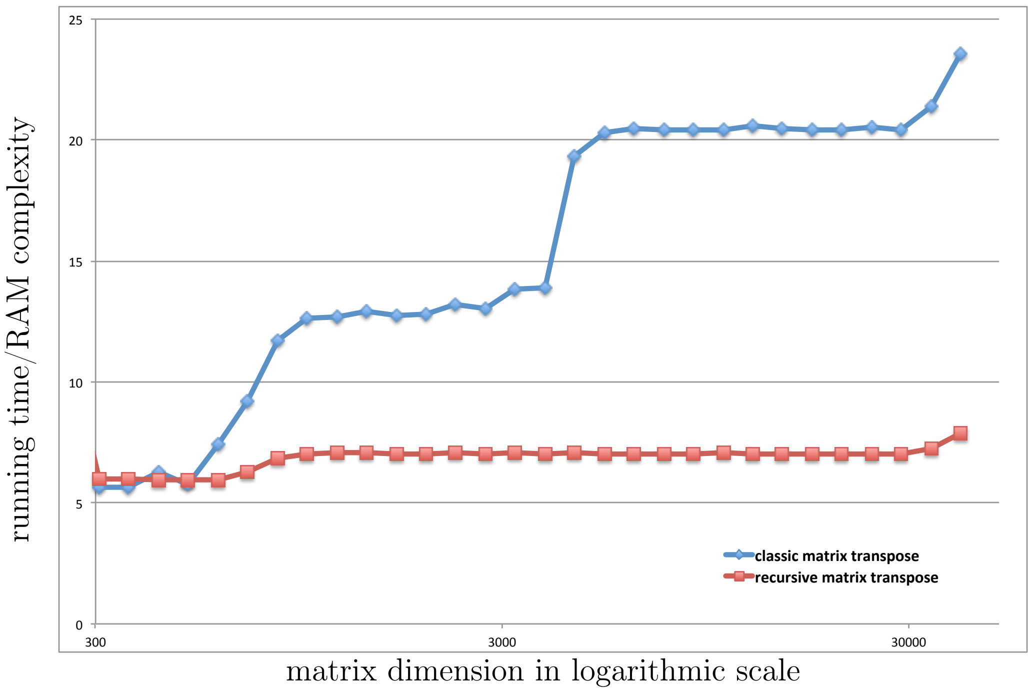

Figure 2.7. Running time of the matrix transpose in row by row layout and in the recursive layout

CAT, or Sequence of Consecutive Address Translations

In our analysis, for many algorithms precisely calculated VAT complexity was much smaller than the RAM complexity. We believe that our approach can bring valuable insight for future research, but some of our results can be obtained in a simpler way. The memory access patterns in the algorithms in question share some common characteristics. There are not too few elements, they are not overspread in the memory, and the accesses are more or less performed in a sequence. We formalize these properties in the following definition.

Definition 2.33. *We call a sequence of ℓ memory accesses a **CAT** (sequence of consecutive address translations) if it fulfills the following conditions:*

- $\ell = \Omega(\tau d)$.
- *On average, $\Omega(\tau)$ elements are accessed per page in the access range.*
- *The pages are used in the increasing or decreasing order. (But accesses in the page need not follow this rule).*
- *Memory accesses from the sequence are separated by at most a constant number of other operations.*

Lemma 2.34. *In case of a CAT, the cost of the address translation is dominated by the cost of RAM operations and therefore negligible. Hence, for CATs, it is sufficient to account for them only in the RAM part of the analysis.*

Proof. We assume the LRU replacement strategy. First, let us assess the cost of translating addresses for all the $O(\ell/\tau)$ pages in the increasing order. The first translation causes d TC misses. Since we allow only a constant number of operations between accesses from the considered sequence, the LRU replacement strategy holds translation path of the last translation when the next one starts. Hence, the addresses to be translated change like in a classic K-nary counter. The amortized cost of an update of a K-nary counter is $O(1)$. Since on average $\Omega(\tau)$ elements are accessed per page, the access range is at most of length $O(\ell/\tau)$, and so the cost of updates is $O(\ell)$. However, we do not start counting from zero, and the potential function in the K-nary counter analysis can reach up to log (the highest number seen), which in our case can reach d. Hence, we need to add the cost of another d TC misses to our estimation. The cost of all translations is therefore equal to $O(\tau d + \ell) = O(\ell)$.

In the definition of a CAT, we do not assume that every page is used exactly once. However, neither skipping values in the counter, nor reusing them causes extra TC misses.

Since the RAM cost is exactly $\Theta(\ell)$, it dominates the translation cost. □

RAT, or Sequence of Random Address Translations

Similarly, algorithms with a high VAT cost share common properties.

Definition 2.35. *A sequence of memory accesses is called a **RAT** (sequence of random address translations) if:*

- *There is a memory partitioning such that each part consists of all memory cells with some common virtual address prefix, and parts are of size at least Pm^θ for $\theta \in (0,1)$.*

- *For at least a constant fraction of the accesses with at least a constant probability, each access is to a part that was not accessed since W TC misses.*

Lemma 2.36. *The cost of a RAT of length ℓ is $\Theta(\tau \ell d)$. It is the same as the cost of the address translations.*

Proof. We assume the LRU replacement strategy. Since parts are of size at least Pm^θ for $\theta \in (0,1)$, a translation of an address from each part uses $\Theta(d)$ translation nodes unique to its translation subtree. Therefore, an access to a part that was not accessed since W TC misses, misses the root of the subtree, and by Lemma 2.8, the access causes $\Theta(d)$ misses. As this happens for at least a constant fraction of the accesses with at least a constant probability, the total cost is $\Theta(\tau \ell d)$. The RAM cost is only $\Theta(\ell)$, which is less than the VAT cost by the order assumption 1. □

2.6.3. The Translation Tree is Shallow

It is true that the height of the translation tree on today's machines is bounded by 4, and so the translation cost is bounded. However, even though our experiments use only 3 levels, the slowdown appears to be at least as significant in practice as the one caused by a factor of $\log n$ in the operational complexity. Therefore, decreasing VAT complexity has a prominent practical significance. Please note that while 64 bit addresses are sufficient to address any memory that can ever be constructed according to known physics, there are other practical reasons to consider longer addresses. Therefore, the current bound for the height of the translation tree is not absolute.

2.6.4. What about Hashing?

We have been asked whether the current virtual address translation system could be replaced with one based on hashing tables to achieve a constant amortized translation time. Let us argue that it is not a good idea. First and foremost, hashing tables sometimes need rehashing, and this would mean the complete blockage of an operating system. Moreover, an adversary can try to increase a number of necessary rehashes. Note that probabilistic guarantees are on the frequencies of the rehashes and the program isolation is insufficient to discard this concern, because an attack can be performed with side-channels like, for example, differential power analysis (see [Tiri, 2007]). Finally, a tree walk is simple enough to be supported by hardware to obtain significant speedups; in case of hashing, this would be not so easy.

On the other hand, simple hash tables can be used to implement efficient caches. In fact, associative memory can be seen as a hardware implementation of a hashing table. If we no longer require from the associative memory that it reliable stores all the previous entries, then associative memories of small enough sizes can be well supported by hardware. This is in fact how the TLB is implemented, and one of the reasons why it is so small.

2.7. Conclusions

We have introduced the VAT model and have analyzed some fundamental algorithms in this model. We have shown that the predictions made by the model agree well with measured running times. Our work is just the beginning. There are many open problems, for example: Which translation cost is incurred by cache-oblivious algorithms that require a tall cache assumption? Virtual machines incur the translation cost twice. What is the effect of this? What is the optimal VAT cost of sorting?

We believe that every data structure and algorithms course must also discuss algorithm engineering issues. One such issue is that the RAM model ignores essential aspects of modern hardware. The EM model and the VAT model capture additional aspects.

Part II

Massive Multicore Parallel

3 Parallel Convex Hull

Currently, we are facing a technological barrier in speeding up processor cores. Multicore systems seem to be the only direction to follow. This can be achieved by putting together a number of traditional CPU cores, which proved effective for many applications. On the other hand, one can use the same number of transistors to build a great number of very simple cores; this is the GPU[1] approach, commonly referred to as many-core. While the future is uncertain, there is one thing we are convinced about. We believe that machines of the future are going to be heterogeneous. They will consist of a number of highly capable cores based on current CPUs and of a great number of simpler cores based on current GPUs. While not every problem can be efficiently solved on GPU, those that can gain speedups counted in orders of magnitude. The hope is that, for many problems, clever algorithms can use the additional computational power to overcome the architectural shortcomings. Unfortunately, many fundamental problems still have no practical solutions for GPUs. We need to understand these machines much better to use them efficiently.

It is hard to predict what the future many-core devices will be like as they are still evolving, but a general trend has already been established. nVidia, while developing their graphic accelerators, as a side effect have also created an efficient massive multicore processor comprising hundreds of so-called CUDA cores and billions of transistors. All nVidia graphic cards released after 2006 support CUDA[2]. Soon after the success of CUDA, companies like nVidia, AMD, Intel and Apple agreed on a common programming interface called OpenCL[3] that

[1]Graphics processing unit, see http://en.wikipedia.org/wiki/GPU
[2]Compute Unified Device Architecture, see http://en.wikipedia.org/wiki/CUDA
[3]Open Computing Language, see http://en.wikipedia.org/wiki/OpenCL

resembles CUDA. One can safely assume that a vast majority of the processors that are to be released in upcoming years will be compatible with CUDA or OpenCL.

Is seems that all possible aspects of parallel algorithms and many-core systems were already discussed in the past. The subject has been well-known in the algorithms community for several decades. It is true that different models of parallel and multicore computations have given researchers a lot of insight into what could be done with multiprocessor machines. Parallelism in its purest forms of SIMD and MIMD has been studied extensively (see [JáJá, 1992]). Memory hierarchy can be described in terms of the External Memory model [Aggarwal and Vitter, 1988] extended with the Cache Oblivious approach [Frigo et al., 2012]. Difficulties with execution control on CUDA due to nondeterministic asynchronous threads (see [Gibbons, 1989]) and need of "barrier synchronization" as the only means of synchronization, like in the Bulk Synchronous Parallel model (see [Valiant, 1990], [Culler et al., 1996]) are well-known as well. Yet, none of the models proposed by theoreticians have received much attention from the software engineering community. Practice shows that existing models are just not close enough to the hardware to be applied easily; only few individual results have been put into practice. The reason for this setback is that the foundations on which these models were designed not only precede the age of multicores, but were founded by forefathers of the field whose expectations do not match the current development. Unfortunately, assembling more fitting models feels too incremental for many researchers to want to get involved. As a result, the applied community works with informal models detached from the theoretical current that are slowly emerging from the publications of CUDA users and programmers (see [Billeter et al., 2009]). It is very clear that only through working with existing technology like CUDA are we actually able to understand how to perform computations efficiently.

Among the theoreticians, the issue is visible as well. While for sequential von Neuman type machines the RAM model feels perfectly natural, there is no natural model for parallel machines. One of the very first attempts to analyze CPU based multicores together with the memory hierarchy is the PEM model [Goodrich et al., 2007]. The very successful MapReduce[4] parallel API is slowly appearing in the consciousness of the theoretical community due to researchers like Michael Goodrich and Eli Upfal. In this part we take on the challenge of making a foothold in the uncharted world of the GPUs.

In the following chapters, we show how to efficiently solve the 2D Convex Hull problem on CUDA, which is currently the most commonly used many-core technology. Prior to our implementation of the Convex Hull algorithm, the problem was considered by the CUDA community to be unsuitable for GPUs. For instance in [Rueda and Ortega, 2008], the authors state:

[4]See [Dean and Ghemawat, 2008] and http://en.wikipedia.org/wiki/MapReduce.

"We have implemented other geometric algorithms in CUDA like (...) convex hull of large meshes but the results have been poor. (...) The problem can hardly be decomposed into simpler independent tasks that can be assigned to the threads."

We show a case study of the problem, a number of techniques we found useful for designing multicore algorithms, and in the analysis we point out the differences between known models and the real machines. In particular we show how to deal with the fact that CUDA is a blend of both SIMD and MIMD, and by design provides a very short cache. We abstain from defining the complete model; instead we take a more axiomatic approach and assume only what is needed for the solution. A good algorithm should work on many models.

3.1. Sequential Algorithm

First, let us show a sequential version of the algorithm we use. In this way, we will be able to more directly describe its advantages while we explain our choice. The algorithm is based on [Chan et al., 1997], [Wenger, 1997], and [Bhattacharya and Sen, 1997], which in turn extend the research of [Kirkpatrick and Seidel, 1986].

The Algorithm 1 finds the leftmost point of the set l and the rightmost point r. It splits points into ones that lie above or below line lr[5]. The upper and lower hull are computed separately by Algorithm 1 and then merged. The algorithm is described from the perspective of the upper hull. First, we find a point belonging to the final convex hull and set it as a pivot. Next, we partition the points using the pivot like in the Quicksort algorithm, along the way discarding some points that do not participate in the hull. The discarding is handled by replacing these points with placeholders. When the partition procedure is complete, the result is available in the original input array, mixed arbitrarily with the correct number of placeholders.

[5]In this chapter, we will repeatedly denote pairs of points like $\{a, b\}$ as ab to indicate that every pair of points is also a segment on the plane.

```
Algorithm 1: ConvexHull
  Input  : T[l..r]: Array segment of 2D points.
  Assert : $l =$ T[l] is the leftmost point of the set; $r =$ T[r] is the
           rightmost point of the set; T[l] and T[r] belong to the final
           convex hull.
1 if(T[l+1..r-1] is empty) break;
2 $m$ := select_pivot_belonging_to_the_convex_hull(T[l..r]);
3 p := lossy_partition(T[l..r], $m$);
4 ConvexHull(T[l..p]);
5 ConvexHull(T[p..r]);
  Result : Array T contains the sequence of the points from the upper convex
           hull appearing in left to right order and separated by placeholders.
```

3.1.1. Select Pivot and Lossy Partition

For the recursion to be effective, we need to assure that the subproblems decrease in size by a constant factor. At the same time, we insist that point m comes from the convex hull. To reach both of these goals at once we implement the helper procedures as follows:

```
Algorithm 2: select_pivot_belonging_to_the_convex_hull
  Input  : T[l..r]: Array segment of 2D points.
1 i := random(0,sizeof(T[l..r])/2);
2 $(a, b)$ := (T[l+2*i],T[l+2*i+1]);
3 return the highest point form T[l..r] in the direction normal to line $ab$
```

```
Algorithm 3: lossy_partition
  Input  : T[l..r]: Array segment of 2D points, $m$: pivoting point.
  Assert : $l =$ T[l] is the leftmost point of the set; $r =$ T[r] is the
           rightmost point of the set; T[l] and T[r] belong to the final
           convex hull.
1 forall the i in [0..sizeof(T[l+1..r-1])/2] do
2     $(p, q)$ := (T[l+1+2*i],T[l+2+2*i]);
3     if $p$ is a convex combination of $\{q, l, m, r\}$ then discard $p$;
4     if $q$ is a convex combination of $\{p, l, m, r\}$ then discard $q$;
5 return partition(T[l..r], $m$) // returns final position of the pivot
```

Checking whether we can discard points in algorithm 3 covers, in particular, the following case. Let pair pq ($p_x < q_x$) lie on the left side of m and have a slope lesser than the slope of ab. Then q would lie below segment pm (see Figure 3.1), and so it would be pruned. Therefore, the right point of any pair with the slope lesser than the slope of ab will either be assigned to the right subproblem or be pruned.

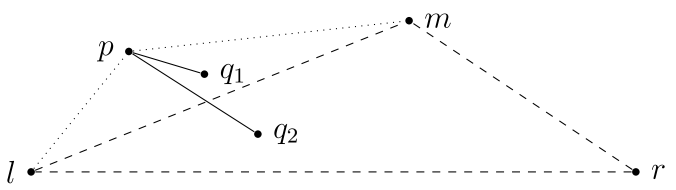

Figure 3.1. In this case, q_1 and q_2 can be safely pruned before recursing to subproblem $[l, m]$.

For pair ab with the median slope, $\frac{1}{4}$ of all points could never appear in the left subproblem. It is worth noticing that there might be only one subproblem if point m returned by `select_pivot_belonging_to_the_convex_hull` is equal to l or r. However, while we do not progress with the extension of the hull here, we actually benefit because there is only one subproblem with a size usually much smaller than the superproblem.

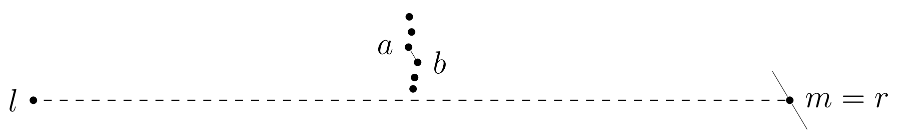

Figure 3.2. Input in which every choice of segment ab leads to point m being one of l or r, unless l or r is one of a or b.

The algorithm can be seen as the `Basic-Randomized-Upper-Hull` algorithm from [Bhattacharya and Sen, 1997] with a slightly improved pruning strategy. Hence, the following theorem holds.

Theorem 3.1. *Algorithm 1 has output sensitive complexity $O(n \log h)$ where n is the size of the input, and h is the size of the convex hull.*

3.2. Problem Analysis

In order to develop an efficient algorithm for CUDA, one needs to first understand CUDA's basic design. Many-core systems are intended to handle massive numbers of parallel threads at the expense of having very restrictive cashing and flow control.

Basic Design

A CUDA device runs a single program on a user defined number of numbered threads. Every 32 consecutive 32-aligned threads (0 to 31, 32 to 63, and so on) constitute a **warp**. From the user point of view, threads in a warp run in SIMD (single instruction multiple data) fashion, with the *arbitrary CRCW* (concurrent read concurrent write), for details see [JáJá, 1992]. All warps have access to the large global memory. A user defined, hardware limited number of consecutive warps constitutes a **block**. Warps in a single block share a cache as well as a so-called shared memory of a rather restrictive size and can be explicitly synchronized. Cache transactions are performed on coalesced, byte-aligned chunks of 32, 64, or 128 bytes. The collection of all blocks is called a **grid**, and a CUDA program is executed as a sequence of grids. The number of threads and the size of a block is a grid parameter. Only the content of the global memory is carried on between the grids.

Warp-Centric Design

Computation is performed on a collection of multiprocessors. Each multiprocessor contains multiple types of scalar processors, each suited for different tasks like memory transactions, floating point operations, etc. The smallest assembly unit in CUDA is the warp. All warps in the same block are guaranteed to be run on the same multiprocessor. Available warps are scheduled instruction by instruction to suitable processors, not necessarily in the program order. As a result, classic notions of scheduling and task preemption are meaningless on CUDA.

Currently, once a block is scheduled on a multiprocessor it cannot be evicted, and number of blocks that can be scheduled at once is limited, therefore, attempt to synchronize between blocks can lead to deadlock. However, scheduling algorithm of blocks is not defined, and can be changed any time. Task completion is guaranteed only if there are no inter-block dependencies.

Use Cases

CUDA is by design most efficient with grids of many small threads that do not communicate and execute identically (at least per warp). Similarly, grids of threads that cooperate in SIMD manner at warp level are efficient. The smallest schedulable unit on CUDA is the warp. To take advantage of these patterns in the first phase of design, we ignore the fact that each warp consists of threads, and we treat it as if it was a scalar processor. This allows us to parallelize by using MIMD techniques. Therefore, we call these grids **MIMD grids**. Only later we use the fact that each warp can perform fairly complicated operations and try to use SIMD techniques to speed up the work that each warp does independently. This approach cleanly separates high and low level parallelization. In the following chapters we will mostly discuss high level parallelization with

MIMD grids. Therefore, unless stated otherwise, when we say *processor* we have a *warp* in mind.

In the case of grids that require more communication, using a single block of maximal size and explicit wall synchronizations allows us to use classic SIMD algorithms and is efficient enough for small data. We call these grids **SIMD grids**.

Unfortunately, in the case of large tasks that require even more communication, one must find a way to split them into multiple MIMD and SIMD grids that store intermediate data in the global memory.

Marriage Before Conquest

There are plenty of classic sequential algorithms to choose from, yet as CUDA is best suited for solving many independent problems; the first approach that comes to mind is Divide and Conquer. In this case, for most of the time there should be enough subproblems to occupy each warp with no need for communication or coordination. However, there are two problems. First, CUDA does not support recursion[6], and hence recursion must be replaced with iteration. In each iteration, the program divides all remaining subproblems and processors, then solves them independently, and finally merges them iteration by iteration. Unfortunately, except for very predictable, very well balanced recursion trees, the level of synchronization achievable with the barrier synchronizations does not appear to be sufficient to achieve enough parallelism while merging solutions.

Fortunately, there is a better way. In [Kirkpatrick and Seidel, 1986], the authors proposed a concept of *marriage before conquest* which is a variant of the *divide and conquer* technique where merging is performed before solving subproblems. The proposed algorithm splits a problem into *left* and *right* halves, finds the (upper) convex hull edge that connects the halves, and only then proceeds to the subproblems. The key to making the algorithm iterizative is the fact that when we have more than one processor working on a marriage before conquest problem, all the subproblems can be launched immediately as if it were a tail recursion.[7] We call this technique a **multiple-tile recursion**, and we discuss it in more detail in Section 3.3. The major advantage of this technique is that there is no need for partial resynchronizations between processors in order to merge solutions.

There is however, one disadvantage to this algorithm from [Kirkpatrick and Seidel, 1986]. While finding a bridge that splits the problem into halves takes only a linear time, it can take a logarithmic number of phases that must be executed in a sequence. The `Basic-Randomized-Upper-Hull` algorithm from [Bhattacharya and Sen, 1997], instead of searching for a bridge, finds a single vertex of the convex hull that splits the problem into subproblems

[6]CUDA 5 supports a limited level of recursion.

[7]Special credit for noticing this analogy goes to Prof. Tony Hoare.

that with a high probability are smaller by a constant. The drawback of this approach is that the vertex found could coincidentally be one of the two already known extremal vertices, therefore the total number of subproblems grows. The benefit is that this form of partitioning always takes only one phase of a linear time, which in turn permits more efficient parallelization.

3.3. Marriage Before Conquest and Good Distribution

One natural approach to load balancing on multicores is work-stealing. Unfortunately, it is not feasible to implement stealing with the limited execution control that CUDA provides. Yet, we will show that regular enough multiple-tile recursion trees can be executed efficiently. The technique is as follows. We execute a recursion tree level by level, describing all the tasks on the level as a single uniform task that can be synchronized with barrier synchronizations only. Processors used for each task are more or less proportionally assigned to the subtasks they spawn. At some point, the problem gets divided into multiple small subproblems, each assigned to a separate processor (splitting stage). When this happens, the subproblems can be solved independently in a sequential manner (independent stage).

3.3.1. Splitting Stage

The splitting stage must fulfill two conditions to be efficient. First, the idle must be small in each iteration. Second, there should not be too many iterations.

We cannot make any formal statement about idle time on CUDA itself, not knowing the scheduling algorithm. However, we can guarantee the regularity of the tasks we launch in a way that permits competitive scheduling on a wide range of models. We show later that a partition can be computed in work $\Theta(n)$, with the parallel speedup essentially equal to the number of assigned processors P. Under this assumption, it is sufficient if the number of processors assigned to each subproblem is proportional to the size of the input on every level of the recursion.

Lemma 3.2. *Let us consider algorithms that split each problem t into a set of subproblems S such that the total size of the subproblems $\sum_{s \in S} n_s$ does not exceed the size of the superproblem n_t.*

It is possible to assign at least $P_s := \left\lfloor \frac{n_s}{n} P \right\rfloor$ processors to every subproblem s of size n_s, using only processors from their superproblem t.

Proof. The proof is by induction. There are enough processors to assign to the root problem as $\left\lfloor \frac{n}{n} P \right\rfloor = P$. There are also enough processors to assign to subproblems, if we have enough processors in the superproblem, since:

$$\sum_{s\in S}\left\lfloor \frac{n_s}{n}P\right\rfloor \leqslant \left\lfloor \frac{\sum_{s\in S} n_s}{n}P\right\rfloor \leqslant \left\lfloor \frac{n_t}{n}P\right\rfloor .$$

□

Notice that distributing processors, according to the principle above, may assign no processors to tasks of size less than $\lfloor \frac{n}{P} \rfloor$. However, this is fine at this stage because our goal is to split tasks, not to solve them.

Lemma 3.3. *If the splitting strategy on an algorithm realizes lemma 3.2, each subproblem of size at least $\frac{n}{P}$ gets assigned at least half of the processors implied by the perfect proportional split.*

Proof.

$$\frac{P_t}{\frac{n_t}{n}P} \geqslant \frac{\lfloor \frac{n_t}{n}P\rfloor}{\frac{n_t}{n}P} \geqslant \frac{\lfloor \frac{n_t}{n}P\rfloor}{\lfloor \frac{n_t}{n}P\rfloor + 1} \geqslant \frac{1}{2}, \text{ since } \left\lfloor \frac{n_t}{n}P\right\rfloor \geqslant \left\lfloor \frac{n/P}{n}P\right\rfloor = 1$$

□

We claim that distributing tasks in agreement with lemma 3.3 allows for efficient task scheduling on a wide range of models.

Now, let us estimate the number of iterations needed to reduce the problem to a collection of tasks of size no more than n/P. The original algorithm from [Kirkpatrick and Seidel, 1986] guarantees that each subproblem is of size at most $\frac{3}{4}$ of the superproblem. For a uniform random choice, the result is only slightly weaker.

Proposition 3.4. *We say that a partition is α-**shrinking** if both subproblems get no more than fraction $1/\alpha$ of the points for $\alpha > 1$. Every processor needs to participate in no more than $\log_\alpha P$ shrinking partitions before its problem is reduced to size $\frac{n}{P}$.*

Lemma 3.5. *Let us consider a partition procedure that uses P processors in accordance with lemma 3.2 and is α-shrinking with probability at least $1/\beta$. With a probability at least $\frac{1}{2}$ for some constant $\gamma > 0$, all remaining subproblems are no larger than $\frac{n}{P}$ after $\gamma \log_\alpha P$ iterations.*

Proof. By lemma 3.2, no task of size at least $\frac{n}{P}$ is left without a processor. Hence, by proposition 3.4, each processor needs to participate in $\log_\alpha P$ shrinking partitions in order to reduce all subproblems. Let $\mathbf{p}$ be an arbitrary processor. Let X be a random variable that counts the number of α-shrinking partitions after $\gamma \log_\alpha P$ partitions. Let us calculate the probability that $X < \log_\alpha P$. As a partition is α-shrinking with a probability at least $\frac{1}{\beta}$, by Chernoff bound we can say:

$$\mathbb{P}(X < \log_\alpha P) = \mathbb{P}\left(X < \left(1 - \frac{\gamma - \beta}{\gamma}\right)\frac{\gamma}{\beta}\log_\alpha P\right) \leqslant$$
$$\leqslant e^{\frac{\left(\frac{\gamma-\beta}{\gamma}\right)^2 \frac{\gamma}{\beta}\log_\alpha P}{-2}} = P^{-\gamma\cdot\left(\frac{\gamma-\beta}{\gamma}\right)^2\frac{\log_\alpha P}{2\beta}}.$$

For fixed α and β, and sufficiently large P we can choose γ so that $\mathbb{P}(X < \log_\alpha P) < P^{-2}$. Finally, by use of the union probability, we calculate the probability that at least one of P processors fails to shrink its task.

$$P \cdot P^{-2} = P^{-1} < \frac{1}{2} \quad (\text{for } P \geqslant 2)$$

□

Lemma 3.6. *In the convex hull algorithm, a randomly taken pair defines a pivot that splits the problem so that no subproblem has size greater than $\frac{5}{6}$ with a probability at least $\frac{1}{3}$.*

Proof. With probability $\frac{1}{3}$, the slope of the randomly chosen pair is from the middle third. For every pair with a slope smaller than the median, the right point of the pair will not be assigned to the left subproblem. This eliminates $\frac{1}{6}$ of the points. Therefore, no more than $\frac{5}{6}$ points is assigned to the left subproblem (analogically for the right subproblem). □

In the convex hull algorithm, a partition is $\frac{5}{6}$-shrinking with probability at least $\frac{1}{3}$. Applying calculation from the proof of lemma 3.5 yields the following proposition:

Proposition 3.7. *In the convex hull algorithm, all remaining subproblems are no larger than $\frac{n}{P}$ after $7\log_{\frac{6}{5}} P$ iterations of the partition with a probability at least $\frac{1}{2}$.*

3.3.2. Independent Stage

When all remaining subproblems are small enough, we assign each one to a single processor (perhaps more than one per processor) and solve sequentially. All subproblems are of size at most n/P and contain at most h vertices of the convex hull. For this stage to be efficient, none of the independent problems should take significantly more time than others. This can be achieved with any deterministic $O(n \log h)$ algorithm. We conjecture that our algorithm has this property as well, with high probability, but we prove only a simpler related result.

Theorem 3.8. *The expectation of the maximum running time of a collection of P quicksort instances of size ℓ is sharply concentrated around $O(\ell \log \ell)$, assuming that $P < \ell^c$ for some constant $c > 0$.*

Proof. Every processor has an input of size ℓ. From the main theorem of [McDiarmid and Hayward, 1996], we know that the running time of Quicksort is sharply concentrated around its expectation value, here $O(\ell \log \ell)$. In particular, the probability that the running time varies from the expectation by more than factor of ϵ equals

$$\ell^{-2\epsilon \ln\ln \ell - O(\ln\ln\ln \ell)}.$$

Hence, the union probability that any one of the processors takes more time than expected by a factor of ϵ, is bounded by

$$P \cdot \ell^{-2\epsilon \ln\ln \ell - O(\ln\ln\ln \ell)} < \ell^{c-2\epsilon \ln\ln \ell - O(\ln\ln\ln \ell)},$$

which is a sharp bound. □

With $\ell \leqslant \frac{n}{P}$ and $c = 1$ the assumption on P in the theorem 3.8 becomes $P^2 < n$, which is reasonable in practice.

3.3.3. A Couple of Words on Processor Virtualization

By design, the number of threads one can run on CUDA is largely independent of the number of the actual on chip processors. CUDA effectively creates an illusion of as many virtual processors as are needed for a task and simulates them on the available hardware. The greatest drawback of this kind of processor virtualization is its nondeterministic asynchronicity (compare [Gibbons, 1989]). It forces us to resort to barrier synchronizations whenever we need to pass any message between processors. The nondeterministic asynchronicity makes it impossible to implement load balancing systems based on the work-stealing technique widely used in parallel scheduling [Blumofe and Leiserson, 1999]. The only way to implement load balancing is to periodically perform a synchronization, collect all tasks, and redistribute them. However, this design has advantages as well. As noticed by [Valiant, 1990], if we randomly distribute p tasks on p processors, then, with high probability, at least one will get about $\log p / \log\log p$ tasks. However, if we randomly distribute $p \log p$ tasks on p processors, then, with high probability, no one will get more than about $3 \log p$ tasks. Of course, the CUDA scheduling algorithm, while not publicly available, is clearly better than a random one. Therefore, when load distribution is uncertain, splitting problems on more virtual processors than areavailable on the hardware effectively introduces a form of auto-balancing.

3.4. Algorithmic Details

Up to now we have presented an algorithm with a regular iterative structure that admits parallelization. In this section, we show how to parallelize it efficiently. There are many ways to perform certain actions, and there are many performance indicators. Practical experience shows that the most important indicator is work, assuming linear speedup for a limited number of processors. Sacrificing work to further shorten the critical path is impractical. In particular *Nick's Class*[8] is not a class of general practical interest.

> "Thus, parallel algorithms need to be developed under the assumption of a large number of data elements per processor. This has significant impact on the kinds of algorithms that are effective in practice."
> [Culler et al., 1996]

However, there are cases where linear speedup cannot be expected. In particular, parallelization usually leads to additional communication steps. The solution in this case is to hide these between instructions that do gain optimal speedup.

> "Thus, if the algorithm designer can arrange to overlap communication with enough independent computation in each step, the communication latency and bandwidth can be ignored (up to a constant factor)."
> [Culler et al., 1996]

3.4.1. Splitting Stage

The splitting stage consists of a number of iterations and prior preparation. The implementation of the preparation is trivial after the iteration is understood; therefore, we will simply assume the following:

1. Array `T[0..n-1]` contains all the input points for the upper hull problem.

2. `T[0]` is the leftmost point on the plane and `T[n-1]` is the rightmost.

3. We have temporary global arrays `L[0..P-1]`, `V[0..3*(P-1)]`, and `U[0..n-1]`.

Array `L[0..P-1]` will be used to carry on private processor information between grids, in particular the execution control data. Before the first iteration, every processor stores in his private storage the range of its current problem, which is $(0, n-1)$; the range of processors assigned to it $(0, P-1)$; and range of its consecutive part. (We split the array into P possibly equal parts.) We will make an effort to update `L` in a consistent way without the need for additional interprocessor communication.

[8] http://en.wikipedia.org/wiki/NC_(complexity)

Iteration

Each iteration executes as follows. We describe it from the perspective of a single processor $\mathrm{P_c}$. Let $\mathrm{P_{min}}$ and $\mathrm{P_{max}}$ stand for the first and the last processor assigned to the same problem as $\mathrm{P_c}$, respectively.

1. $\mathrm{P_{min}}$ picks a random pair in the current problem and stores it in its control structure.

○ *barrier synchronization* (by slicing the problem into separate grids)

2. $\mathrm{P_c}$ accesses the pair stored in the storage of $\mathrm{P_{min}}$. Next, it sequentially scans its part of `T` for the highest point in the direction normal to the line defined by the pair. Finally, it stores in `V[Pc]` the following pair (the highest point, 0 if $\mathrm{P_c} = \mathrm{P_{min}}$ and 1 otherwise).

○ *barrier synchronization*

3. We run a SIMD grid (see page 57) to compute in array `V` the prefix of the following associative but not-commutative operator. Here, $a, b \in \mathbb{R}^2$, and $y, z \in \{0, 1\}$.
$$(a, y) \lhd (b, z) := (\max\{a \cdot z, b\}, y \cdot z)$$
The operator executed sequentially computes the maximum of all the previous elements in the first element of the pair, but resets whenever it encounters 0 in the second element.

   The maximum here is taken on the position in the direction normal to the line defined by the pair chosen in step 1, under the assumption that 0 is the minimum.

○ *barrier synchronization*

4. $\mathrm{P_c}$ accesses the maximum point stored in `V[Pmax]`. Next, with this point taken as the pivot, the processor sequentially scans its part of `T` and counts how many points should go to which subproblem.

5. We store calculated quantities of the points into `V` so that values for each generated subproblem are stored consecutively, and the number of discarded points is stored between the left and right subproblems. We store the values in the first element of a pair; the second element is 1, unless it is the size of the left subproblem stored by the first processor in the problem.

○ *barrier synchronization*

6. We run a SIMD grid to compute in array `V` the prefix of the following associative but not-commutative operator. Here, $a, b \in \mathbb{R}$, and $y, z \in \{0, 1\}$.
$$(a, y) \lhd (b, z) := (a \cdot z + b, y \cdot z)$$

The operator executed sequentially computes the sum of all the previous elements in the first element of the pair but resets whenever it encounters 0 in the second element. Effectively, the operator determines for each processor where a sequential algorithm could move points while executing a partition.

- *barrier synchronization*

7. $\mathrm{P_c}$ sequentially scans its consecutive part of `T` and uses values from `V` to move elements into their position. Since we cannot perform the partition in-place, we store them in array `U`. In case of discarded elements, we write a copy of the pivot.

- *barrier synchronization*

8. $\mathrm{P_c}$ computes the sizes of the subproblems of its problem and, respecting the good distributions, decides which subproblem to join. If $\mathrm{P_c}$ wrote to `U` data that will get no processor assigned, or copies of the pivot created for discarded elements, then $\mathrm{P_c}$ copies them to the respective cells in array `T`. These cells will not be accessed in any future iteration. Finally, $\mathrm{P_c}$ updates its control structure with its new $\mathrm{P_{min}}$, $\mathrm{P_{max}}$, the range of its next problem and new range of its part.

- *barrier synchronization*

9. We run a SIMD grid to find out whether there is still a problem too big to finish the splitting stage. We also swap pointers to `T` and `U`.

In this approach, subproblems occupy a subset of the memory assigned to their parents, and abandoned space is first filled in with copies of elements from the convex hull chosen as subsequent pivots. Therefore, the output needs to be postprocessed to remove the repetitions. Another approach to this problem is as follows.

Iteration with Balancing

In step 6, we compute a typical prefix sum and do not reserve any space for the discarded elements, but we do reserve space for the abandoned tasks. Hence, we compact the problem in every step, but we always need to completely generate all content of the array `U`. In this approach it is necessary to rebalance the workload in every step since the copying of the abandoned subproblems must be distributed equally. We assume that processor i is responsible for splitting the task that contains the element $i\left\lceil\frac{n}{P}\right\rceil$ and for copying abandoned subproblems that lie between $i\left\lceil\frac{n}{P}\right\rceil$ and $(i+1)\left\lceil\frac{n}{P}\right\rceil$ (if any). Finally, this means that the processor control structures cannot be generated in-place. Therefore, we generate them in a helping array, and the structure for the new processor i is prepared by the processor that stored the cell number $i\left\lceil\frac{n}{P}\right\rceil$ in array `U`, where n is meant in the context of the next iteration (because this is the processor that has all the required information). It could happen that some processors may need to prepare more than a constant number of control structures. This is not a problem because it is possible only if the total number of the points shrinks substantially, which is worthwhile. Moreover, this approach makes it possible to easily readjust the number of processors from an iteration to iteration.

3.4.2. Independent Stage

Once all the tasks are smaller than the required threshold value α, the independent stage starts. Any sequential algorithm can be used; we used the same algorithm as in the splitting stage. However, here we use only a single processor, therefore, the algorithm can no longer be executed as a multiple tail recursion. Hence, we simulate the recursion by an iteration with a stack. The stage is executed as follows.

1. The tasks are compacted (like in the iteration with balancing).

◦ *barrier synchronization*

2. Each processor prepares a privete recursion stack in the global memory to simulate recursion by iteration.

3. Each processor i puts all the problems that have their rightmost point in segment `T[`$i\alpha$`..`$(i+1)\alpha$`-1]` on its recursion stack.

◦ *barrier synchronization*

4. Each processor sequentially solves problems on its recursion stack.

◦ *barrier synchronization*

5. The final result is compacted.

3.4.3. Complexity

Each iteration runs a constant number of SIMD grids to compute parallel prefixes (for details see [JáJá, 1992]). The prefixes can be computed in $\Theta(\log P)$ parallel steps and would take $\Theta(P)$ time if serialized. Each parallel independent scan in MIMD grids has work $\Theta(n)$ and hence takes time at least $\frac{n}{P}$. Therefore, as long as $P^2 < n$, the communication cost can be ignored.

3.4.4. Low Level SIMD Parallelism

On page 56 where we introduced MIMD and SIMD grids, we stated that by default we use the MIMD grids, and for simplicity we pretend that each warp is a single processor. Now it is the time to take this simplification away. The CUDA warp has access to a small cache, a small addressable shared memory, and a relatively high number of private non-addressable registers[9] per thread. This creates a convenient SIMD environment, yet there are differences.

First, the shared memory is limited, and all accesses to the global memory are actually performed through the cache in 64, 128 and 256 byte aligned blocks. Assigning the warps to different consecutive segments is a high level parallelization technique to decrease the number of cache misses. On the low level of parallelization it is the opposite; one needs to interleave the parts assigned to threads so the memory accesses can be coalesced. It is quite apparent that our algorithm can easily coalesce the memory accesses as data is almost always read sequentially.

Second, in the external memory model, one could simply ignore the cost of the operations; in the RAM model, operations are all one cares about. In CUDA, global memory transactions are clearly more expensive than the operations, but the difference is not immense. Therefore, since CUDA can perform memory transactions in parallel with the computation, the usual goal is to make sure that the cost of the computation time is comparable with the time of the memory transactions. The proportion is hard to predict and usually needs to be established experimentally.

Finally, the small amount of memory, the fixed number of processors, and an incomparably greater number of elements to process makes the standard measure of operational complexity unpractical; the game is about a constant. Some classic time-efficient (rather than work-efficient) algorithms apply, but in the end, clever engineering is maybe even more important than the algorithmic insights. Let us discuss some task specific techniques we use.

[9]Namely, we can declare an array in the shared memory and refer to particular cells using integer variables; in the case of registers, there is no such flexibility.

Step 2 of the Splitting Stage

The standard sequential maximum algorithm is as follows.

```
Algorithm 4: sequential maximum
  Input : T[0..n-1]: Array of 2D points, $a$, $b$: chosen slope, $a$.x<$b$.x.
  $x_0$ := $a$.y-$b$.y;
  $y_0$ := $b$.x-$a$.x;
          // ($x_0$, $y_0$) is a vector normal to $ab$ directed upwards.
  ans := T[0];
  i := 1;
  while $i<n$ do
      projection := (T[i].x-ans.x)·$x_0$ + (T[i].y-ans.y)·$y_0$;
      if $projection>0$ then ans := T[i];
      i++;
  return $ans$
```

The obvious way to parallelize it is as follows. Here, P_c is our point-of-view processor, and both processors and cells in the array are indexed from 0.

```
Algorithm 5: parallel maximum
  Input : T[0..n-1]: Array of 2D points, $a$, $b$: chosen slope, $a$.x<$b$.x.
  $x_0$ := $a$.y-$b$.y;
  $y_0$ := $b$.x-$a$.x;
          // ($x_0$, $y_0$) is a vector normal to $ab$ directed upwards.
  ans := T[$P_c$ ];
  i := $P_c$ + size_of_a_warp;
  while $i<n$ do
      projection := (T[i].x-ans.x)·$x_0$ + (T[i].y-ans.y)·$y_0$;
      if $projection>0$ then ans := T[i];
      i := i+size_of_a_warp;
  Result : answer
```

However, there is another way that will allow us to spread the computation for each single point to two threads. Let us assume that array `T` is stored in memory as an alternation of x and y coordinates, and we access the memory cells directly with notation `T[[i]]`. In particular, `T[[4]]=T[2].x`, `T[[6]]=T[3].x`, `T[[7]]=T[3].y`, etc. Let us also introduce a notation `v{thread_id}` that allows us to access the instance of the variable `v` that is local to another thread in the warp[10]. We denote our point-of-view processor as P_c and its companion processor as $\bar{P}_c$, where P_c `xor` $\bar{P}_c = 1$. Finally, let both processors and cells in the array be indexed from 0. Then, we can write the following algorithm.

[10] It can be realized via local memory or in the modern CUDA by the shuffle instructions.

Algorithm 6: another parallel maximum

```
Input  : T[0..n-1]: Array of 2D points, $a$, $b$: chosen slope, $a$.x<$b$.x.
if $P_c = 0 \pmod 2$ then
    c$_0$ := $a$.y-$b$.y                  // the left companion thread
else
    c$_0$ := $b$.x-$a$.x                  // the right companion thread
// Each thread has a half of the normal vector,
ans := T[[P$_c$ ]];                       // and half of an answer.
i := P$_c$ + size_of_a_warp;
while $i<2\cdot n$ do
    projection := (T[[i]]-ans)·c$_0$;     // one of the two components
    if $projection + projection\{\bar{P}_c\}>0)$ then ans := T[[i]];
    i := i+size_of_a_warp;
Result : answer: one of the coordinates of the highest point
```

In this approach, each point is handled by two threads. This slightly increases the total work but has other advantages.

- The number of threads that can be efficiently utilized doubles.
- The amount of shared memory available per point doubles.
- The chances of warp not diverging on conditional statements significantly increases. (It is raised to the power $\frac{1}{2}$.)

This technique is not very beneficial in this step, but it is very useful in others.

Step 4 of the Splitting Stage

Let us assume that p and q both fall to the left subproblem, and they both lie above the line lm, see Figure 3.3. In order to decide whether we can discard one of them, it is sufficient to compute with means of the cross product orientation of the angles $\angle plq$ and $\angle pmq$. If the orientations are the same (the case of q_1), then no point can be discarded; otherwise, one of them is discarded (the case of q_2).

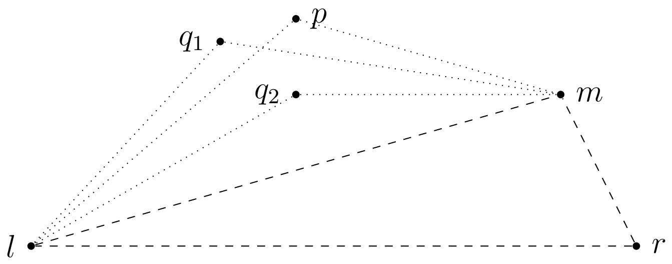

Figure 3.3. Localization of points q_1 and q_2 in respect to point p.

This procedure has enough symmetry to spread the classification of a single pair to multiple threads with the technique we presented in step 2. In order for the data to fit in the limited space provided by the shared memory, we had to spread the computation of each pair into two threads. We chose to keep p and q in separate threads, and we used the following procedure to compute the cross product.

```
1 value:=x·y{$\bar{\mathrm{P}}_\mathrm{c}$}; // equals x{$\mathrm{P}_\mathrm{c}$}·y{$\bar{\mathrm{P}}_\mathrm{c}$}
2 value:=value − value{$\bar{\mathrm{P}}_\mathrm{c}$};
  Result: value: Cross product of vectors $(x, y)$ in $\mathrm{P}_\mathrm{c}$ and $\bar{\mathrm{P}}_\mathrm{c}$
```

Step 7 of the Splitting Stage

The algorithmic part of the solution is very straight forward. From the perspective of a sequential processor, the task is to split a stream of incoming data into two output streams (three if we count the pruned points). The parallel procedure itself is based on computing the prefix sum on the characteristic function of elements belonging to each class. However, is not obvious whether to recompute which class the points belong to in this step, or store the data already computed in step 4 in the global memory. Our implementation a little of both, but this will not necessarily be the most efficient approach for future machines.

> "Sometimes, the best optimization might even be to avoid any data transfer in the first place by simply recomputing the data instead whenever it is needed."
>
> [NVIDIA Corporation, 2008, CUDA programming Guide]

3.5. Implementation

In order to verify the relevance of the proposed model and the algorithm, we implemented it for NVIDIA GPU parallel machines using CUDA. In this section, we provide detailed information on our implementation of the CUDA convex hull. We designed the implementation to work on any GPU of compute capability 1.2. Please note that the algorithm presented in the previous sections was improved over time to be even more efficient on modern CUDA devices. Hence, it slightly differs from the implementation presented here.

We have used an nVidia GTX 285 card, with the following properties:

- $\mathcal{M} = 30$ — Number of multiprocessors;
- $\mathcal{W} = 32$ — Maximal number of concurrent warps on a single multiprocessor;
- $\mathcal{S} = 32$ — Warp size

- $\mathcal{B} = 512$ — Maximal block size — size of a group of threads that can communicate without using expensive global memory and barrier synchronization

- $\mathcal{R} = 16$ — Maximal register-per-thread usage to achieve full occupancy.

For MIMD grids, we could have potentially used up to $P = \mathcal{M} \cdot \mathcal{W} = 960$ warps, each representing an independent processor. Yet, because of the limited amounts of shared memory and registers, we were abble to use only half of this number, namely 480.

For SIMD grids, we launch exactly one block of $\mathcal{B} = 512$ threads.

The whole program consists of four major stages. Let us now describe them, together with the main differences between our CUDA implementation and the theoretical algorithm explained in the previous sections.

Initialization. We search for 4 extreme points — the furthest point in the top-left, top-right, bottom-right, and bottom-left directions. For many input problems, this approach allows us to immediately throw out many points that fall into a quadrilateral formed by these extreme points. Points which remain outside the quadrilateral are partitioned into four initial subproblems. We can run them all at once with a slight adaptation of the upper convex hull algorithm, assigning warps proportionally.

Splitting Stage. First, for every subproblem bounded by points l and r, we select a single random pair (a,b) and determine the pivot m. Next, we overwrite the discarded points with a special value. Only after rejecting points inside the triangle (l,m,r) do we pair the remaining points such that each pair consists of points from the same subproblem. This is substantially different from the theoretical approach where the pairing is fixed before all other operations take place. Theoretically, an adversary can order points such that our pairing will discard much fewer points than the number guaranteed in the original algorithm. However, in practice, this heuristic is very effective as it significantly increases the number of pairs with a potential of discarding a point.

In our implementation, we try to assign processors to subproblems proportionally without abandoning any task, and with an accordance to lemma 3.2. There is a small probability that there is no such split, in which case we rerun the task, unless there are only 3 processors. Finally, the stage is complete when all processors have been separated. This approach was a practical improvement over our previous solution with provable length of $O(\log P)$ iterations. However, for future implementations, we recommend the use of our current strategy described in the previous sections.

Independent Stage: At this point, the problems are small enough that each warp can work independently of all others. Exactly one MIMD grid is launched for this whole stage. The loop is encoded within the procedure, with every warp holding a single stack in the CUDA local memory[11]. Operations on the stack, although expensive, are executed only $O(h)$ times.

At this stage, every processor has exclusive access to the problem it was assigned to; therefore, we can afford a Hoare-style in-place partition. This way, we read and classify each point only once, and we move it to its correct destination immediately thereafter. Other parts of the program are analogical to the collaborative stage.

Finalization: We compact the resulting convex hull points so that they are stored consecutively in the memory.

Now, let us describe each of the major stages in more detail.

3.5.1. Initialization

The program starts with a single pass over the following phases:

Input: `T[0..n]`: An unordered array of 2D points,
`i`: Processor index (global warp index)

1. Conceptually divide the array into groups of size $\mathtt{g} := \lceil \frac{n}{P} \rceil$.
   `b`:=`g`·`i`; `e`:=`g`·`(i+1)`. Assign subarray `T[b..e]` to warp `i`.
2. Launch a MIMD grid: Each warp searches for the top-left, top-right, bottom-right, and bottom-left point in its subarray `T[b..e]` using a simple reduction algorithm.
3. Launch a SIMD grid to find global points: top-left (`A`), top-right (`B`), bottom-right (`C`), and bottom-left(`D`). We form the quadrilateral `ABCD`.
4. We say that point `p` is **above** edge `XY`, if $(\mathtt{Y} - \mathtt{X}) \times (\mathtt{p} - \mathtt{X}) > 0$. In a MIMD grid, we count how many points in the subarray `T[b..e]` are above each of the edges `AB`, `BC`, `CD`, and `DA`. Note that each point can be above only one of the edges at most, since the vertices are the extreme points.
5. In a SIMD grid, we compute the total number of points above each of the edges of the quadrilateral `ABCD`. We create 4 bins for each type of points and reserve an appropriate amount of space in them for each warp.

[11]Local memory is a misleading name for a section of the global memory private to a thread.

6. In a MIMD grid for every point in subarray `T[l..r]`, we recompute the edge above which it is located. We copy the point into the corresponding bin reserved in the previous phase. Points inside the quadrilateral `ABCD` are implicitly discarded.

7. In a SIMD grid, we prepare the GPU to work on the collaborative stage. Given the four bins, we assign a number of warps to work on them, proportional to their size. We assert that at least one warp is assigned to each bin.

Output: The quadrilateral `ABCD`
4 arrays, each consisting of points lying only above an edge `AB`, `BC`, `CD`, `DA`, respectively.

3.5.2. Splitting Stage

As long as there is at least one problem with several warps assigned to it, we proceed as follows:

Input: `i`: Global warp index
`T[0..n]`: An array of 2D points
A set of control variables, separate for every warp.

- l, r — endpoints of the current problem that warp `i` is assigned to.
- `B`, `E` — begin and end indices of points that belong to the problem that warp `i` is working on.
- `b`, `e` — mark the portion of array `T` that current warp is explicitly and exclusively assigned to.

Assert: At least one problem has at least two warps assigned to it.
For every active warp, the range it is exclusively assigned to `(b,e)` lies entirely in the problem range `(B,E)`.
Exclusive ranges (`b`,`e`) for warps assigned to the same problem sum up to the range of the whole problem `(B,E)`.
For every active warp, all points in the range of the problem `(B,E)` are above the **base line** lr.

The following phases are executed only by these warps, which are assigned non-exclusively to a problem. Otherwise, the warp stays idle.

1. We launch a MIMD grid, with every warp selecting a random pair of points from the problem. We ensure that every warp belonging to the same problem selects the same pair. The pair of points form the **pivoting line**. Each warp finds the outermost point in its range from `T[b..e]` in the direction perpendicular to the pivoting line, using a reduction algorithm.

2. A single block finds a single pivoting point m for each problem, using a segmented prefix scan algorithm. m belongs to the convex hull. Note that m can be l or r.

3. In the next MIMD grid, we perform a lossy partition. Each warp counts how many points from `T[b..e]` will fall to the left and right subproblems. For every pair of points falling into the same subproblem, we check whether one can be discarded. If so, its value in the global memory is overwritten by the pivoting point `m`, which guarantees that it will be discarded in the next phase.

4. A SIMD grid finds the sizes of subproblems, based on values reported by each warp. The memory for the new subproblems is reserved, and correct portions of them are assigned to each of the warps.

5. A MIMD grid partitions points from `T[b..e]` to the left and right subproblems. Points are copied into another array `U`. Points inside the triangle (l, m, r) are discarded.

6. Because some points may be dropped, we launch another MIMD grid to clean up new empty space occurring in the output array `U` by setting a special value there. The same part of the memory is cleaned in array `T` as well. From this point, until the finalization step, the empty space will never be referenced.

7. A SIMD grid is launched to reassign warps to new subproblems following the good distribution principle from Section 3.3, but keeping all warps busy, if possible. The kernel updates the control values for all warps.

Finally, pointers to the output array `U` and the input array `T` are swapped.

Output: `T[0..n]`: An array of partitioned 2D points
A set of control variables, separate for every warp, prepared for the next iteration of the algorithm.

3.5.3. Independent Stage

At this point we have enough problems, so that each warp can work independently. Exactly one MIMD grid is launched for this whole stage. Recursion stack is encoded within the grid.

Algorithm 7 provides a detailed pseudo-code close to our CUDA implementation. Let us explain the important points of the code:

① Each warp uses its own stack. Because the warp's fast shared memory is limited, the stack is located in the global memory. Stack operations become quite expensive, but their total number is limited by $O(h)$.

Algorithm 7: `CUDA convex hull for an independent stage`

```
Input : inputProblem, T[b..e]: Array segment of 2D points
stack.push(inputProblem); // ①
while stack not empty do
    problem=stack.pop();
    pivotingLine:=randomPointPair(T[b..e]);
    pivot:=furthestPoint(T[b..e],pivotingLine); // ②
    declare register var p[0..2$\mathcal{S}$]; // ③
    p[0..2$\mathcal{S}$].side:='discard';
    p[0..$\mathcal{S}$].point:=readChunk(T[b..b+$\mathcal{S}$]);
    p[0..$\mathcal{S}$].side:=classify(pivot,p[0..$\mathcal{S}$]); // ④
    empty:=[$\mathcal{S}$..2$\mathcal{S}$];
    subproblem[left,right].size:=0;
    while something more to read do
        p[empty]:=readChunk(T[next chunk]); // ⑤
        p[empty].side:=classify(pivot, p[empty]);
        sort p[0..2$\mathcal{S}$] by p.side: {'left','discard','right'}; // ⑥
        if count(p[0..2$\mathcal{S}$].side=left)≥$\mathcal{S}$ then
            connectInPairsAndDiscardSome(p[0..$\mathcal{S}$]); // ⑦
            subproblem[left].size+=storeChunk(p[0..$\mathcal{S}$]); // ⑧
            empty:=[0..$\mathcal{S}$];
        if count(p[0..2$\mathcal{S}$].side=right)≥S then
            connectInPairsAndDiscardSome(p[$\mathcal{S}$..2$\mathcal{S}$]);
            subproblem[right].size+=storeChunk(p[$\mathcal{S}$..2$\mathcal{S}$]);
            empty:=[$\mathcal{S}$..2$\mathcal{S}$];
    sort p[0..2$\mathcal{S}$] by p.side: {'left','discard','right'};
    if empty≠[0..$\mathcal{S}$] then
        connectInPairsAndDiscardSome(p[0..$\mathcal{S}$]);
        storeChunk(p[0..$\mathcal{S}$]);
    if empty≠[$\mathcal{S}$..2$\mathcal{S}$] then
        connectInPairsAndDiscardSome(p[$\mathcal{S}$..2$\mathcal{S}$]);
        storeChunk(p[$\mathcal{S}$..2$\mathcal{S}$]);
    cleanEmptySpace();
    if subproblem[left].size>1 then stack.push(subproblem[left]);
    if subproblem[right].size>1 then stack.push(subproblem[right]);
```

② The search for the furthest point is performed using a simple reduction algorithm over all the points in the range.

③ We use the register space to hold 2D point data. There are $\mathcal{S}$ threads per warp and each holds two points, forming a virtual array of size $2\mathcal{S}$. Thread i holds points at index i and $i + \mathcal{S}$ of that virtual array.

④ In the classify function, each thread checks independently whether the point it holds falls to the left or right subproblem, or whether it should be discarded.

⑤ If we are reading into the left side of array p (that is, into [0..$\mathcal{S}$]), we take the next unread chunk from array T[b..e] (e.g. [b+$\mathcal{S}$..b+2$\mathcal{S}$]). However, if we are reading onto the right side of array p (into range [S..2$\mathcal{S}$]), we take the next unread chunk counting from the end side of array T[b..e]. In particular, in the first iteration of the while loop, it will be T[e-$\mathcal{S}$..e].

The function readChunk ensures that at the end of the inner while loop, every point is read exactly once.

⑥ The sort is using p.side as a key value, which can take only three values. This is why we use a counting-sort. The points are stored in the register space; we make use of a small amount of shared memory to count the points and then transfer them.

⑦ At this point in the program execution, we are guaranteed that in p[0..$\mathcal{S}$], there are only points which fall into the left subproblem. Now we conceptually connect these points into pairs, matching an even point with the next odd point.

The two threads in parallel can compute necessary cross products to learn if one of the points can be dropped without exchanging the full point information.

If some points get dropped, we mark them as 'discarded' and compact the p[0..$\mathcal{S}$] part of the array.

Analogous operations are performed at lines 21, 24, and 27.

⑧ We perform the partition in-place. Since there is at least one chunk of size $\mathcal{S}$ read from the left and right side of array T[b..e] at all times, we are guaranteed that we can store the computed data back there.

In our case, at line 18, we store data at the beginning of the array and immediately thereafter, by setting "empty" to [0..$\mathcal{S}$], we schedule the read of the next chunk from the front as well, preserving the invariant.

Finally, we increment the size of the left subproblem by the number of points that were actually stored after the pair-pruning. Note that at each write, the value cannot be smaller than $\lfloor \frac{\mathcal{S}}{2} \rfloor$.

Analogous operations, working at the end side of T[b..e], are performed in lines 22-23.

3.5.4. Finalization

We obtain an array `T` containing all the points of the convex hull in a sorted order, interleaved with special markers to indicate an empty space. A stable compaction algorithm is used to obtain the convex hull without the gaps.

3.6. Experiments

Here we present results of the tests we have performed. We created tests containing 10^5, 10^6, and 10^7 points. For each size, we selected random points in a unit square, on a unit disc, and on a unit ring. In the latter case, in theory, all points should belong to the convex hull. In practice, however, since we used 32-bit floating point numbers, many points overlapped and some were not exactly on the ring, resulting in much smaller convex hulls. The best practical result we could relate to is presented in [Srikanth et al., 2009]. The algorithm we refer to is a simple adaptation of a QuickHull algorithm that always chooses as a pivot the point that lies the furthest from the baseline. The simplicity of the approach makes it effective for the random input case, but gives no guarantees for a general case. The authors provide only a very sparse performance report and complexity analysis, and so our comparative analysis is equally brief; for details refer to Figure 3.4. Our approach seems to outperform their implementation by a factor of about 2.

We have also measured the number of global memory reads and writes the program had to perform. In the analysis, we implied that the number of reads and writes per warp is of the optimal order $O\left(\frac{n}{PS}\log h\right)$. The total number of memory operations is then $O\left(\frac{n\log h}{S}\right)$. Figure 3.5 shows how the experimentally obtained number of reads and writes relates to this asymptotic expectation. Finally, in Figure 3.6, we present the number of iterations the program had to perform.

3.7. Future Research

A follow-up work based on our research already exists. While our algorithm does not scale to the 3D case, it is a good starting point. The algorithm described in [Tang et al., 2012] is an analogue to ours, raised to the 3D case. While it does not compute the convex hull, it produces a star-shaped polyhedron in 3D, and in the process prunes many of the internal points. It is designed to be an efficient preprocessing strategy for CPU algorithms. The approach is taken another step

Test	n	h	LEDA	Srikanth	Our implementation
Square	10^5	36	70ms		9ms
	10^6	40	970ms		14ms
	10^7	42	19550ms		58ms
Disc	10^5	160	80ms		12ms
	10^6	344	980ms	13ms	20ms
	10^7	715	19960ms	115ms	73ms
Ring	10^5	31526	220ms		27ms
	10^6	58982	2750ms		53ms
	10^7	101405	45690ms		282ms

Figure 3.4. Absolute run times of the convex hull. Column 'LEDA' shows the performance of a CPU program that computes the convex hull using 64-bit floating point numbers, see [Mehlhorn and Näher, 1995]. Column 'Srikanth' refers to the CUDA implementation reported in [Srikanth et al., 2009]. We do not know exactly how the points are distributed in their tests; we believe they are evenly distributed on a disc.

forward by [Gao et al., 2013] who first compute a star-shaped polyhedron in 3D, and then turn it into a convex hull by flipping concave regions into convex ones.

We also suggest another approach. [Edelsbrunner and Shi, 1991] describes a sequential marriage before conquest 3D convex hull algorithm. It has the same weaknesses and strengths as [Kirkpatrick and Seidel, 1986] for the 2D case, hence, to become truly efficient it needs an efficient pruning strategy. The algorithm has suboptimal complexity $O(n \log^2 h)$, but the marriage before conquest property is required only for the splitting stage. For the independent stage, another algorithm can be used.

While shifting from 2D to 3D is a theoretical challenge, solving the higher dimensional case appears to be more of an engineering challenge, unless new approaches can be found.

A long term goal is to port other fundamental algorithms to CUDA, especially graph algorithms, as they seem to be another class of problems that are hard to split. The technique of partitioning a graph into small distance subgraphs from [Mehlhorn and Meyer, 2002] might be a good starting point for developing a marriage before conquest algorithm in this case.

Finally, nVidia recently designed CUDA devices capable of running multiple grids concurrently. This requires some mechanism of virtual memory that at some point will have to be investigated.

Test	n	h	Memory transactions reads/10^3	writes/10^3	$\frac{n\lceil\log h\rceil}{1000\cdot\mathcal{S}}$
Square	10^5	36	36	13	19
	10^6	40	225	59	188
	10^7	42	2085	453	1875
Disc	10^5	160	55	31	25
	10^6	344	293	153	281
	10^7	715	2500	1265	3125
Ring	10^5	31526	206	127	47
	10^6	58982	1224	693	500
	10^7	101405	11409	6315	5313

Figure 3.5. The number of global memory operations and their relation to our predictions.

Test	n	h	Splitting iterations	$\lceil\log P\rceil$	Independent iterations min	max	avg	total
Square	10^5	36	3	9	0	3	0.04	19
	10^6	40	4	9	0	4	0.06	30
	10^7	42	4	9	0	5	0.06	27
Disc	10^5	160	6	9	0	7	0.2	98
	10^6	344	9	9	0	10	0.45	218
	10^7	715	11	9	0	19	0.94	453
Ring	10^5	31526	14	9	3	123	48	23198
	10^6	58982	12	9	8	256	99	47540
	10^7	101405	13	9	11	542	177	85074

Figure 3.6. Number of program iterations at splitting and independent stages. Each warp performs the same number of steps at the splitting stage, but at the independent stage, the number depends on the size of the problems the warp was assigned to.

4 Parallel Sorting

The reader might have noticed that the previous chapter, while focusing on the convex hull problem, provides all the necessary tools to efficiently implement quicksort on CUDA. In fact, the code below does not contain steps we did not already parallelize in the solution of the convex hull problem.

Algorithm 8: `quicksort`

```
Input : Array segment A[a..b]
1 if(a >= b) break;
2 pivot = select_pivot(A[a..b]);
3 p = partition(A[a..b], pivot);
4 quicksort(A[a..p-1]);
5 quicksort(A[p+1..b]);
```

We implemented this algorithm by removing irrelevant parts of the convex hull implementation, which yielded very decent results. However, readers interested in performance should refer to [Cederman and Tsigas, 2009]. The article presents a dedicated, tuned, and tested implementation of quicksort very similar to our approach.

Since this is an implementation of quicksort, the expected work is $\Theta(n \log n)$. Work proved itself to be a very good performance indicator in practice However, unlike in the convex hull, in quicksort a simple comparison is sufficient to decide into which subproblem each element should be passed. Therefore, the number of the memory transactions has high impact on the quicksort's running time. Increasing the number of pivots from 2 to k decreases the I/O cost from $\Theta\left(\frac{n}{S} \log n\right)$ to $\Theta\left(\frac{n}{S} \log_k n\right)$ and is practical if we have enough free shared memory. This raises a natural question: *What is the lower bound?*

4.1. Lower Bound

In order to discuss lower bounds, we need a formal I/O model strong enough to resemble a CUDA machine, but not too strong so it still remains meaningful. We suggest the following warp-centric model.

The machine consists of P processors with a cache of size $c\mathcal{S}$ and a global memory of size $2n$, split into blocks of size $\mathcal{S}$. In each step, each processor (in some globally fixed order) exchanges content of its memory cells with some single block of the global memory. By exchange we mean read a complete block and then write $\mathcal{S}$ elements into it. We assume that elements to be exchanged cannot be modified, only moved and compared.

4.1.1. Bound on Permutation

We permute n distinct elements, allowing for the usage of an extra n memory cells. Any permutation algorithm must be able to output each of the $n!$ possible permutations. Assuming that each block must be accessed at least once, the block can be permuted during its last access with no extra I/O. This way we reduce the number of required outputs to $n!/(\mathcal{S}!)^{n/\mathcal{S}}$. Every memory exchange can introduce $\binom{c\mathcal{S}}{\mathcal{S}}$ possible values to any of the $2n/\mathcal{S}$ blocks. Therefore, to know how many steps t are needed for P processors to permute n elements, we need to solve the following inequality:

$$\frac{n!}{(\mathcal{S}!)^{n/\mathcal{S}}} \leqslant \left(\frac{2n}{\mathcal{S}}\binom{c\mathcal{S}}{\mathcal{S}}\right)^{tP}.$$

Knowing that for large enough values: $(n/e)^n < n!$; $\mathcal{S}^\mathcal{S} > \mathcal{S}!$; $\displaystyle\binom{c\mathcal{S}}{\mathcal{S}} < (ce)^\mathcal{S}$,

we can relax the inequality to: $\displaystyle\left(\frac{n}{e\mathcal{S}}\right)^n < \left(\frac{2n}{\mathcal{S}}(ce)^\mathcal{S}\right)^{tP}$.

Thus: $t > \dfrac{n\log(\frac{n}{\mathcal{S}e})}{P\log(\frac{2n}{\mathcal{S}}(ce)^\mathcal{S})}$.

If $\frac{n}{\mathcal{S}} = \Omega\left((ce)^\mathcal{S}\right)$, then the result is $t = \Omega(\frac{n}{P})$.

If $(ce)^\mathcal{S} = \Omega\left(\frac{n}{\mathcal{S}}\right)$, then the result is $t = \Omega\left(\frac{n}{P\mathcal{S}}\log_c\left(\frac{n}{\mathcal{S}}\right)\right)$.

Theorem 4.1. *The worst case number of I/O turns required to permute n distinct elements is:* $\Omega\left(\min\left\{\frac{n}{P}, \frac{n}{P\mathcal{S}}\log_c\left(\frac{n}{\mathcal{S}}\right)\right\}\right)$

To get the bound that incorporates the possibility of ℓ warps communicating in a block, one needs to multiply c by ℓ.

4.1.2. Bound on Sorting by Comparisons

To sort, we must first learn the correct permutation and then actually permute. To learn the permutation by comparisons, the processor first reads a block. If the block is read for the first time, we learn in which of the $\mathcal{S}!$ possible permutations are elements in the block. This happens $n/\mathcal{S}$ times. Then, we learn in which of the $\binom{c\mathcal{S}}{\mathcal{S}}$ possible relations are read elements to elements already in local memory. By the pigeon hole rule, adversary can always make sure that if we learn in which of r possible permutations is our subset of elements, the number of possible permutations of all elements decreases at most by a factor r. Therefore, to know how many steps t are needed for P processors to learn a permutation of n elements, we need to solve the following inequality:

$$\frac{n!}{(S!)^{n/\mathcal{S}}\binom{c\mathcal{S}}{\mathcal{S}}^{tP}} \leqslant 1$$

Solving the inequality, analogically to the one in Subsection 4.1.1, leads to the result: $t = \Omega\left(\frac{n}{P\mathcal{S}} \log_c\left(\frac{n}{\mathcal{S}}\right)\right)$. Again, to determine the bound that incorporates the possibility of ℓ warps communicating in a block, one needs to multiply c by ℓ.

In Section 4.2, we mention algorithms that can match this bound.

4.1.3. Conclusions

The bounds clearly resemble the onces known for the "External Memory" model. This is not surprising, because the settings have much in common. However, the "External Memory" approach was first introduced to handle the delay of I/O transactions on disks. One of the most fundamental beliefs in the community is that in practice $\binom{M}{B} = \Omega\left(\frac{N}{B}\right)$; in our case, $(ce)^{\mathcal{S}} = \Omega\left(\frac{n}{\mathcal{S}}\right)$. Therefore, the lower bounds on permutations and on sorting by comparisons are equivalent. Thus, linear work sorting algorithms perform poorly on external memories. In the case of CUDA, the formula $(ce)^{\mathcal{S}} \gg \frac{n}{\mathcal{S}}$ looks reasonable at first, but due to the small values of $\mathcal{S}$ and c, the exponential function on the left is not big enough to balance the hidden constants. A practical implementation of radix sort, which is described in [Billeter et al., 2009], is highly regarded in the CUDA community and supports the claim described above.

Another common assumption, especially in the "Cache Oblivious" setting, is the tall cache assumption (see [Frigo et al., 2012]). The assumption says that $M = \Omega\left(B^2\right)$, where M is the size of cache, and B is the size of a memory block. In our case, the assumption would have to be applied to the shared memory and rewritten as $c\mathcal{S} = \Omega\left(\mathcal{S}^2\right)$, which is $c = \Omega\left(\mathcal{S}\right)$. That is in direct opposition to the CUDA design principles. Moreover, sharing the memory by $\mathcal{S}$ scalar processors effectively leaves c memory cells per processor.

The bounds for CUDA are very similar to those of External Memory if one considers only the formulas. However, two of the most basic assumptions about the External Memory are reversed on CUDA. Hence, for many of the problems already considered in literature, cases that seemed to be of little importance for larger but slower memory storages are exactly the ones that are interesting on CUDA.

4.2. Sorting by Sorting in Registers

There are many sorting algorithms known for the External Memory. For fast memory levels, simplicity tends to prevail, because computational complexity (with regard to the asymptotic constant) dominates the cost of the memory access for algorithms with a reasonable memory locality. k-merge sorting based algorithms that minimize the number of memory accesses, especially ones based on buffers and priority queues, proved to be effective for sorting data stored externally in slow memories. In this section we present a simple way of interfacing the external k-merge sort with an arbitrary internal sorting algorithm. The greatest advantage of our approach is that it allows for smooth interfacing algorithms between substantially different architectures. In particular, we will use it to design an efficient many-core sorting algorithm on CUDA.

```
Algorithm 9: $k$-merge
  Input  : $k$ sorted streams organized in blocks of size $B$
  Assert : Memory can fit at least $k$ blocks of size $B$
1 foreach input stream $i$ do
2     rank($i$) = $-\infty$
3 while there is unread input do
4     while there is free space in the memory do
5         read a block $b$ from an input stream $i$ with the smallest rank;
6         rank($i$) = $max(b)$;            // value of the last element
7     sort all elements in the memory;
8     output all elements except for $k-1$ blocks with the largest elements
      (and release space);
9 output remaining elements;
  Result : Sorted output stream
```

Theorem 4.2. *Algorithm 9 merges k sorted streams into one sorted stream.*

Before we proceed with the proof, let us offer this intuition. Note that the blocks are read in precisely the same order as in the k-merge algorithm with heap build on k input buffers. This is the optimal prefetch order (see [Hutchinson et al., 2005]).

Proof. Without the loss of generality, let us assume all elements are strongly comparable. Let us consider a point in time when the memory is full after reading a block and updating its rank. Note that rank(j) is equal to the largest element read from stream j up to this point. Let ℓ be the new stream of the smallest rank. Therefore, all the elements of stream ℓ smaller than or equal to rank(ℓ) have already been read. For each stream $j \neq \ell$, all elements smaller than or equal to rank(j) are already read, and since rank(ℓ) < rank(j), all elements smaller than or equal to rank(ℓ) in all streams are already read. Therefore, it is safe to output all the blocks with elements smaller than or equal to rank(ℓ). Let us show that there is no more than $(k-1)B$ elements larger than rank(ℓ).

Let $j \neq i$, hence rank(j) > rank(ℓ), but before reading the last block of j, the rank of j had had to be smaller than the current rank of ℓ, as otherwise the algorithm would not had read from j. Therefore, from each input stream $j \neq \ell$, at most B elements greater than rank(ℓ) are read, and none from the stream ℓ itself. As there are k streams, all the elements except for at most $B(k-1)$ are smaller than rank(ℓ). □

From the External Memory perspective, the algorithm is already applicable for a memory of size Bk, and is online in the sense that (except for setup) it repeatedly reads one block and then it outputs one block. With a proper choice of internal sorting procedure, the algorithm can already be made efficient on CPU in this setting, but the application we are aiming for is possible only when the memory is of size $Bk + \Omega(Bk)$. In this case, the algorithm needs to read $\Omega(k)$ blocks before outputting $\Omega(k)$ new blocks. However, for this price, we get the flexibility to choose the internal sorting procedure to engineer an efficient algorithm for CUDA.

4.2.1. Internal Warp Sorting

Sorting elements by threads of a single warp is a common primitive in CUDA programming. The current algorithm of choice for this problem is usually bitonic sort[1] in the shared memory. While it theoretically has neither optimal work nor time, its simplicity leads to very efficient implementations. Yet, while registers and shared memory are constructed with use of the same hardware components, use of registers is more constrained, but this in turn makes them even faster than already fast shared memory. The price we pay for the speed is that registers are private to each thread and are non-addressable, that is, it must be known at the compilation time which particular registers are to be used in each instruction.[2]

[1]http://en.wikipedia.org/wiki/Bitonic_sorter

[2]CUDA C language provides a syntactic sugar that allows us to define arrays in registers, yet the indices must still be known at the compilation time.

Therefore, in the past, each data exchange between the threads had to use the shared memory. Additionally, CUDA devices have more registers than the shared memory cells. With the warp shuffle instructions available on the most modern CUDA devices, we can engineer a sorting procedure that uses only registers. The `shuffle` instruction[3] permits the exchange of a variable between threads within a warp without using shared memory. The exchange occurs simultaneously for all active threads within the warp, moving 4 bytes of data per thread. The shuffle instruction is suitable only to a SIMD environment of a single warp. Invoking `shuffle(value, thread ID)` in each thread evaluates to the first argument of this very instruction submitted by the thread with the id specified in the second argument.

2D Sorting, Merging and Partitioning

It is clear that `shuffle` can be used to implement a bitonic sorting network without using the shared memory. Let us assume that the command `horizontal_sort(value)` does exactly that. It takes a value from each thread and returns the smallest value to the first thread, the second smallest to the second thread, etc. In each thread we can internally prepare a sequence of ordered registers that we can sort by another sorting network, we will call it `vertical_sort(sequence of registers)`. Now, let us take a 2D array stored in registers indexed by thread id and register names. With the horizontal and vertical sorting procedures we can easily sort 2D arrays using the technique from [Scherson and Sen, 1989]. We can use this procedure to construct an efficient k-merger using algorithm 9. Additionally, we can use it to easily implement a k-partition using algorithm 10.

4.2.2. Efficient Sorting Algorithms

The internal warp sorting is a primitive that allows us to treat warps as if they were simple processors in MIMD sorting algorithms that are based on mergers and partitioners. To conclude, and hint on interesting subjects for a further experimental study, let us suggest sorting algorithms that seem relatively easy to implement with the set of tools we collected. We assume that $P^2 < n$.

k-Quicksort

Algorithm 8 can be easily adapted to use algorithm 10 to introduce the k-partition. Any sorting algorithm can be used in the independent stage. This leads to an algorithm with the asymptotically optimal I/O cost.

[3]Actually `__shfl()`, `__shfl_up()`, `__shfl_down()` or `__shfl_xor()`.

```
Algorithm 10: $k$-partition
  Input  : Stream organized in blocks of size $B$
  Assert : Memory can fit at least $k$ blocks of size $B$
1 while there is unread input do
2     while there is free space in the registers do
3         read a block from the input stream;
4     sort all elements in the registers using 2D sorting technique;
5     foreach block $b$ in the registers do
6         if b[0] and b[B-1] belong to the same partition $i$ then
7             output $b$ to partition $i$;
8 output remaining elements to their partitions;
  Result : $k$ output streams that are a partition of the input stream in the
           sorting sense of a partition
```

2 Phase Merging

The algorithm is as follows.

1. Split input into P parts and sort independently.
2. Sample the data to get a set of P evenly distributed pivots.
3. Merge parts of each partition by an independent processor.

The first sorting can be done with any sequential sorting algorithm. Very good pivots can be obtained deterministically with the procedure described in [Cole and Ramachandran, 2010]. Finally, merging can be done with Algorithm 9.

Part III

Other Contributions

5 Minimum Cycle Bases

In [Amaldi et al., 2009], we present improved algorithms for finding minimum cycle bases in undirected and directed graphs. For general graphs, the algorithms are Monte Carlo and have a running time of $O(m^\omega)$, where m is the number of edges (arcs) and ω is the exponent of fast matrix multiplication, assuming $\omega > 2$. For planar graphs, the algorithm is deterministic and has a running time of $O(n^2)$, where n is the number of nodes, whereas the previous best running time was $O(n^2 \log n)$. We moreover observe that this algorithm for planar graphs also solves the problem for more specialized classes of cycle bases, namely, integral, totally unimodular, and weakly fundamental cycle bases.

A key ingredient to our improved running times is the insight that the search for minimum cycle bases can be restricted to a subset of at most nm candidate cycles, the so-called isometric cycles, whose total number of edges is bounded above by nm.

The work is extended in [Amaldi et al., 2010], where an $O(m^2 n/\log n)$ algorithm is proposed, and another practical algorithm is evaluated.

Bibliography

[Advanced Micro Devices, 2010] Advanced Micro Devices (2010). AMD64 architecture programmer's manual volume 2: System programming.

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

[Amaldi et al., 2009] Amaldi, E., Iuliano, C., Jurkiewicz, T., Mehlhorn, K., and Rizzi, R. (2009). Breaking the $O(m^2n)$ barrier for minimum cycle bases. In *ESA*, pages 301–312.

[Amaldi et al., 2010] Amaldi, E., Iuliano, C., and Rizzi, R. (2010). Efficient deterministic algorithms for finding a minimum cycle basis in undirected graphs. *Integer Programming and Combinatorial Optimization*, pages 397–410.

[Bhattacharya and Sen, 1997] Bhattacharya, B. K. and Sen, S. (1997). On a simple, practical, optimal, output-sensitive randomized planar convex hull algorithm. *J. Algorithms*, 25(1):177–193.

[Billeter et al., 2009] Billeter, M., Olsson, O., and Assarsson, U. (2009). Efficient stream compaction on wide SIMD many-core architectures. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 159–166. ACM.

[Blumofe and Leiserson, 1999] Blumofe, R. and Leiserson, C. (1999). Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748.

[Cederman and Tsigas, 2009] Cederman, D. and Tsigas, P. (2009). Gpu-quicksort: A practical quicksort algorithm for graphics processors. *J. Exp. Algorithmics*, 14:1.4–1.24.

[Chan et al., 1997] Chan, T. M., Snoeyink, J., and Yap, C.-K. (1997). Primal dividing and dual pruning: Output-sensitive construction of four-dimensional polytopes and three-dimensional voronoi diagrams. *Discrete & Computational Geometry*, 18(4):433–454.

[Cole and Ramachandran, 2010] Cole, R. and Ramachandran, V. (2010). Resource oblivious sorting on multicores. *Automata, Languages and Programming*, pages 226–237.

[Culler et al., 1996] Culler, D., Karp, R., Patterson, D., Sahay, A., Santos, E., Schauser, K., Subramonian, R., and von Eicken, T. (1996). LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85.

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

[Drepper, 2007] Drepper, U. (2007). What every programmer should know about memory. http://lwn.net/Articles/250967/.

[Drepper, 2008] Drepper, U. (2008). The cost of virtualization. *ACM Queue*, 6(1):28–35.

[Edelsbrunner and Shi, 1991] Edelsbrunner, H. and Shi, W. (1991). An O(n $\log^2$ h) time algorithm for the three-dimensional convex hull problem. *SIAM J. Comput.*, 20(2):259–269.

[Frigo et al., 2012] Frigo, M., Leiserson, C., Prokop, H., and Ramachandran, S. (2012). Cache-oblivious algorithms. *ACM Transactions on Algorithms*, pages 4:1 – 4:22. a preliminary version appeared in FOCS 1999.

[Gao et al., 2012] Gao, M., Cao, T.-T., Nanjappa, A., Tan, T.-S., and Huang, Z. (2012). A GPU algorithm for 3D convex hull. Submitted to ACM Transactions on Mathematical Software.

[Gao et al., 2013] Gao, M., Cao, T.-T., Tan, T.-S., and Huang, Z. (2013). Flip-flop: convex hull construction via star-shaped polyhedron in 3D. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D ’13, pages 45–54, New York, NY, USA. ACM.

[Gibbons, 1989] Gibbons, P. (1989). A more practical PRAM model. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 158–168.

[Goodrich et al., 2007] Goodrich, M., Nelson, M., and Sitchinava, N. (2007). Sorting in parallel external-memory multicores. *submitted to WADS*, 2007.

[Hennessy and Patterson, 2007] Hennessy, J. L. and Patterson, D. A. (2007). *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Diego.

[Hutchinson et al., 2005] Hutchinson, D. A., Sanders, P., and Vitter, J. S. (2005). Duality between prefetching and queued writing with parallel disks. *SIAM Journal on Computing*, 34(6):1443–1463.

[JáJá, 1992] JáJá, J. (1992). *An Introduction to Parallel Algorithms.* Addison-Wesley.

[Jurkiewicz and Danilewski, 2010] Jurkiewicz, T. and Danilewski, P. (2010). Efficient quicksort and 2D convex hull for CUDA, and MSIMD as a realistic model of massively parallel computations.

[Jurkiewicz and Mehlhorn, 2013] Jurkiewicz, T. and Mehlhorn, K. (2013). The cost of address translation. In *Proceedings of the Meeting on Algorithm Engineering & Experiments*, New Orleans, USA. Society for Industrial and Applied Mathematics. Invited for publication in the ACM Journal of Experimental Algorithmics (JEA).

[Kirkpatrick and Seidel, 1986] Kirkpatrick, D. G. and Seidel, R. (1986). The ultimate planar convex hull algorithm? *SIAM J. Comput.*, 15(1):287–299.

[McDiarmid and Hayward, 1996] McDiarmid, C. and Hayward, R. (1996). Large deviations for quicksort. *J. Algorithms*, 21(3):476–507.

[Mehlhorn and Meyer, 2002] Mehlhorn, K. and Meyer, U. (2002). External-memory breadth-first search with sublinear I/O. In *ESA*, pages 723–735.

[Mehlhorn and Näher, 1995] Mehlhorn, K. and Näher, S. (1995). Leda: A platform for combinatorial and geometric computing. *Commun. ACM*, 38(1):96–102.

[Michaud, 2007] Michaud, P. (2007). (yet another) proof of optimality for min replacement. http://www.irisa.fr/caps/people/michaud/yap.pdf.

[NVIDIA Corporation, 2008] NVIDIA Corporation (2008). CUDA programming Guide, version 2.0.

[Rahman, 2003] Rahman, N. (2003). Algorithms for hardware caches and TLB. In Meyer, U., Sanders, P., and Sibeyn, J., editors, *Algorithms for Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science*, pages 171–192. Springer Berlin / Heidelberg. 10.1007/3-540-36574-5_8.

[Rueda and Ortega, 2008] Rueda, A. and Ortega, L. (2008). Geometric algorithms on CUDA. *Journal of Virtual Reality and Broadcasting.*

[Scherson and Sen, 1989] Scherson, I. D. and Sen, S. (1989). Parallel sorting in two-dimensional vlsi models of computation. *IEEE Trans. Computers*, 38(2):238–249.

[Shepherdson and Sturgis, 1963] Shepherdson, J. C. and Sturgis, H. E. (1963). Computability of recursive functions. *Journal of the ACM*, 10(2):217–255.

[Sleator and Tarjan, 1985] Sleator, D. and Tarjan, R. (1985). Amortized efficiency of list update and paging rules. *Commun. ACM (CACM)*, 28(2):202–208.

[Srikanth et al., 2009] Srikanth, D., Kothapalli, K., Govindarajulu, R., and Narayanan, P. (2009). Parallelizing Two Dimensional Convex Hull on NVIDIA GPU and Cell BE.

[Tang et al., 2012] Tang, M., yi Zhao, J., feng Tong, R., and Manocha, D. (2012). GPU accelerated convex hull computation. *Computers & Graphics*, 36(5):498 – 506. Early version appeared on Shape Modeling International (SMI) Conference 2012.

[Tiri, 2007] Tiri, K. (2007). Side-channel attack pitfalls. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 15–20, New York, NY, USA. ACM.

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

[Wenger, 1997] Wenger, R. (1997). Randomized quickhull. *Algorithmica*, 17(3):322–329.