December 2013

Hardware Acceleration Technologies in Computer Algebra: Challenges and Impact

Sardar Anisul Haque
The University of Western Ontario

Supervisor
Dr. Marc Moreno Maza
The University of Western Ontario

Graduate Program in Computer Science

A thesis submitted in partial fulfillment of the requirements for the degree in Doctor of Philosophy

Hardware Acceleration Technologies in Computer Algebra: Challenges and Impact

(Thesis format: Monograph)

by

Sardar Anisul Haque

Graduate Program
in
Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

Abstract

The objective of *high performance computing (HPC)* is to ensure that the computational power of hardware resources is well utilized to solve a problem. Various techniques are usually employed to achieve this goal. Improvement of algorithm to reduce the number of arithmetic operations, modifications in accessing data or rearrangement of data in order to reduce memory traffic, code optimization at all levels, designing parallel algorithms with smaller span or reduced overhead are some of the attractive areas that HPC researchers are working on.

In this thesis, we investigate HPC techniques for the implementation of basic routines in computer algebra targeting hardware acceleration technologies. We start with a sorting algorithm and its application to sparse matrix-vector multiplication for which we focus on work on cache complexity issues. Since basic routines in computer algebra often provide a lot of fine grain parallelism, we then turn our attention to many-core architectures on which we consider dense polynomial and matrix operations ranging from plain to fast arithmetic. Most of these operations are combined within a bivariate system solver running entirely on a graphics processing unit (GPU).

Keywords. High Performance Computing, Cache complexity, Parallel algorithms, Many core machines, multi-core machines, Computer algebra.

Acknowledgments

I would like to thank my thesis supervisor Dr. Marc Moreno Maza in the department of Computer Science at the University of Western Ontario. His helping hands toward the completion of this research work were always extended for me. He consistently helped me on the way of this thesis and guided me in the right direction whenever he thought I needed it. I am grateful to him for his excellent support to me in all the steps of successful completion of this research.

I want to thank Dr. Wei Pan of Intel Corporation for discussions around our CUDA implementation for condensation method in the finite field case. Many thanks to Dr. Jürgen Gerhard of Maplesoft for his help during my internship. My sincere thanks to Dr. Shahadat Hossain in the Department of Computer Science at the University of Lethbridge for taking the time to share his thoughts on sparse matrices with me.

I want to thank Dr. Yuzhen Xie, Dr. Changbo Chen in the Department of Computer Science at the University of Western Ontario for providing me help and sharing their knowledge.

All my sincere thanks and appreciation go to all the members from our Ontario Research Center for Computer Algebra (ORCCA) lab, Computer Science Department for their invaluable teaching support as well as all kinds of other assistance.

Many thanks to the members of my committee Dr. Éric Schost, Dr. Michael Bauer, Dr. Kenneth A. McIsaac of the University of Western Ontario and Dr. Yuxiong He of Microsoft Research for their reading of this thesis and comments.

Finally, I would like to thank all of my friends and family members for their consistent encouragement and support.

To you, I dedicate this thesis, Tuie.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

This thesis deals with the implementation of basic routines in computer algebra targeting multi-core and many-core architectures. We consider routines from linear algebra (sparse and dense) and polynomial system solving. In contrast to their counterpart in numerical computing, these routines perform calculations in an exact and complete way. As a consequence, they are highly demanding in computer resources, time and memory. This often limits the impact of computer algebra software to problems of moderate size. However, the abundant computing power of hardware acceleration technologies suggests that much harder problems could be attacked with symbolic computation.

With respect to standard high-performance computing challenges, computer algebra low-level routines fall into the following categories.

(P1): Memory access patterns are highly irregular and work count is essentially proportional to the number of memory accesses. Typical examples are sparse matrix arithmetic and sparse polynomial arithmetic.

(P2): The amount of work is much larger than the amount of reads/writes while memory access patterns are rather regular. Typical examples are dense matrix arithmetic and dense polynomial arithmetic. While these routines allow for fine grain parallelism, certain complex memory access patterns (like for Fast Fourier Transform algorithms) make these operations not so suitable for multi-cores.

Problems of the first kind are more suitable for multicore architectures while problems of the second kind are eligible for many-core accelerators (like Graphics Processing Units).

In this thesis, we are interested in developing tools for analyzing algorithms and implementation techniques targeting hardware acceleration technologies. On multi-

cores, we consider operations that are not suitable for many-cores, due to large data size (a frequent issue in computer algebra) and that pause challenges in terms of memory transfer. For such operations, we propose pre-processing techniques that reshape the input data so as to reduce memory transfer when those operations are applied to the reshaped data. Cache complexity analysis and experimentation confirm the effectiveness of the proposed techniques. To be more specific, our work is driven by classical problem from linear algebra: improving data locality in sparse matrix vector (SpMxV) multiplication. This problem is hard as the permutation of rows or columns of a sparse matrix to maximize locality in SpMxV multiplication is NP-hard [59]. In Chapter 6, we propose a reordering algorithm for sparse matrices that improves the data locality during (SpMxV) multiplication. In each test-case, we re-arrange the input data and show that the cost of this re-arrangement can be amortized against the cost of calculations with the input data, such as linear system solving by iterative methods (conjugate gradient, etc.). We provide cache complexity analysis whose favorable results are confirmed experimentally. As a by-product of this research, we propose a new integer sorting algorithm, which is suitable for large sparse objects. This algorithm is described in Chapter 5.

On many-cores, we consider operations that are both data intensive and compute intensive, which is another frequent feature of computer algebra calculations, as mentioned above. For such operations, we propose a computational model for designing algorithms targeting many-cores, with a focus on reducing parallelism overheads. We present a model of multithreaded computation namely many-core machine model (in Chapter 3) that combines the fork-join and SIMD parallelisms, with an emphasis on estimating parallelism overheads, so as to reduce scheduling and communication costs in GPU programs. We have applied this model and successfully reduced parallelism overheads for several basic routines in polynomial algebra. For polynomial multiplication, our theoretical analysis allows us to reduce parallelism overheads due not only to data transfer but also to code divergence, see Chapter 8. For the Euclidean algorithm, our running time estimates match those obtained with the Systolic VLSI Array Model ([9]). Meanwhile, our CUDA code implementing this optimized Euclidean algorithm runs within the same estimate analyzed by our model for input polynomials with degree up to 100,000. This is reported in Chapter 9.

In Chapter 10, we propose a parallel algorithms for performing subproduct tree construction, evaluation and interpolation and report on their implementation on many-core GPUs. We enhance the traditional algorithms for polynomial evaluation and interpolation based on subproduct-trees, by introducing the notion of a subinverse

tree. For subproduct-tree operations, we demonstrate the importance of adaptive algorithms. That is, algorithms that adapt their behavior to the available computing resources. In particular, we combine parallel plain arithmetic and parallel fast arithmetic.

In Chapter 7, we present a GPU implementation of the condensation method for computing the determinant of a matrix. To the best of our knowledge, this is the first study of the parallelization of this algorithm. We consider both matrices with finite field coefficients and floating point number coefficients. Notably, the latter case exhibits favorable behavior in terms of numerical stability.

All our GPU code is freely available in source at `www.cumodp.org`.

Chapter 2

Background

Until the advent of multi-core and many-core architectures, algorithms subject to effective implementation on personal computers were often designed with *algebraic complexity* as the main complexity measure and with sequential running time as the main performance counter [40, 41, 42, 13, 20]. Nevertheless, during the past 40 years, the increasing gap between memory access time and CPU cycle time, in favor of the latter, brought another important and practical efficiency measure: *cache complexity* [36, 17]. In addition, with parallel processing becoming available on every desktop or laptop, the *work* and *span* of an algorithm expressed in the fork-join multithreaded model [18, 13] have become the natural quantities to compute in order to estimate *parallelism*.

These complexity measures (algebraic complexity, cache complexity, parallelism) are defined for computation models that largely simplify reality. On many-core architectures, several phenomena (parallelism overhead, synchronization among threads of all thread blocks, utilization of all multiprocessors, etc.) limit the performances of applications which, theoretically, have a lot of opportunities for concurrent execution.

2.1 Random access machine (RAM) model

The RAM is a simple model of computation which is used to measure the run time of an algorithm by counting up the number of steps it takes on a given problem instance. Unlike *Turing machine*, which could not access the memory immediately without accessing all intermediate cells, it can access the arbitrary memory in a single step process. The memory considered, in this model, is unbounded and has the capability to store arbitrarily large integers in each of its memory cells. This model

can be programmed in some specified but arbitrary programming language. Some of the properties of this model are as follows:

- Each "simple operation" like addition, subtraction, multiplication, assign, branching, calling, etc. takes exactly 1 time step.
- Loops and procedures are considered to be the composition of many single-step operations.
- Each memory access takes exactly one time step, and we have as much memory as we need. The RAM model takes no notice of whether an item is in cache or on the disk, which simplifies the analysis.

A common problem of this model is that it is too simple, that is, these assumptions make the conclusions and analysis too hard to believe in practice. For instance, multiplying two numbers does not have the same cost as adding two numbers, which clearly violates the first assumption of the model. Memory access times also differ greatly depending on whether data are available in cache or on memory or on the disk, which violates the third assumption. However, in spite of having such restrictions, this model does not provide misleading results for the real world problems, since this only assumes a simple abstract model of computation. Furthermore, robustness of the RAM model enables us to analyze algorithms in a machine-independent way.

2.2 The PRAM model

The *Parallel Random Access Machine* (PRAM) is a natural generalization of RAM. It has unbounded number of processors $P_0, P_1, P_2, \cdots$. Each of these processors has unbounded private *local memory*, which is a sets of registers. Unlike RAM model, it does not have tapes. The computing capability of each processor is the same as RAM. These processors can communicate with each other via *shared memory* (*global memory*) $M[0], M[1], M[2], \cdots$, which is also unbounded. Note that, it is the only way, by which a processor can communicate with another processor. Each processor can access shared memory in unit time, unless there is a conflict.

In Figure 2.1, a PRAM model is shown. The input of a PRAM program consists of n items stored in $M[0], \ldots, M[n-1]$. The output of a PRAM program consists of n' items stored in n' memory cells, say $M[n], \ldots, M[n+n'-1]$. A PRAM instruction executes in a 3-phase cycle:

1. *Read* (if needed) from a shared memory cell,
2. *Compute* locally (if needed),

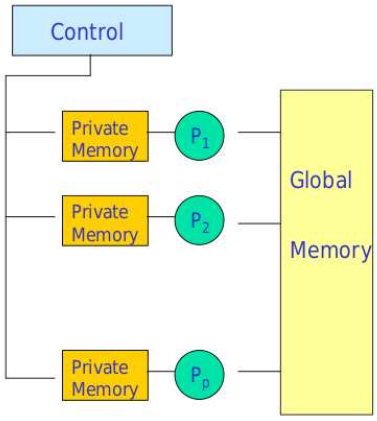

Figure 2.1: The PRAM model.

3. *Write* in a shared memory cell (if needed).

All processors execute their 3-phase cycles *synchronously*. Processor P_0 has a special *activation register* specifying the maximum index of an active processor. Initially, only P_0 is active; it computes the number of required active processors and loads this number in the activation register. Then the corresponding processors start executing their programs. Computations proceed until P_0 halts, at which time all other active processors are halted.

The PRAM Model is attractive for designing parallel algorithms because of the following reasons.

- It is *natural*: the number of operations executed per one cycle on p processors is at most p.
- It is *strong*: any processor can read or write any shared memory cell in *unit time*.
- It is *simple*: ignoring any communication or synchronization overhead.

This natural, strong and simple PRAM model can be used as a benchmark. If a problem has no feasible (or efficient) solution on a PRAM then it is likely that it has no feasible (or efficient) solution on any parallel machine. The PRAM model is an idealization of existing shared memory parallel machines. It ignores lower level architecture constraints (memory access overhead, synchronization overhead, inter-communication throughput, connectivity, speed limits, etc.)

2.2.1 Parallel time, efficiency and speedup factor

The *Parallel Time*, denoted by $T(n,p)$, is the time elapsed from the start of a parallel computation to the moment where the last processor terminates, on an input data of size n, and using p processors. $T(n,p)$ takes into account computational steps (such as adding, multiplying, swapping variables), routing (or communication) steps (such as transferring and exchanging information between processors). The *parallel efficiency*, denoted by $E(n,p)$, is

$$E(n,p) = \frac{\mathsf{SU}(n)}{pT(n,p)},$$

where $\mathsf{SU}(n)$ is a lower bound for a sequential execution. Observe that we have $SU(n) \leq pT(n,p)$ and thus $E(n,p) \leq 1$. One also often considers the *speedup factor* defined by

$$S(n,p) = \frac{\mathsf{SU}(n)}{T(n,p)}.$$

2.2.2 Different types of PRAM models

It is natural to have conflicts in accessing shared memory for some applications. To resolve this issue and synchronize the parallel execution, some mechanism has to be defined for concurrent read and write access conflicts to the same shared memory cell. Some of the basic submodels of PRAM are given below.

Exclusive Read Exclusive Write (EREW). No two processors are allowed to read or write the same shared memory cell simultaneously.

Concurrent Read Exclusive Write (CREW). Simultaneous reads of the same memory cell are allowed, but no two processors can write the same shared memory cell simultaneously.

Concurrent Read Concurrent Write (CRCW). Simultaneous reads and writes of the same memory cell are allowed. CRCW can be divided further based on concurrent writes.

PRIORITY Concurrent Read Concurrent Write (PRIORITY CRCW). Simultaneous reads of the same memory cell are allowed. Processors are assigned fixed and distinct priorities. In case of write conflict, the processor with highest priority is allowed to complete WRITE.

ARBITRARY Concurrent Read Concurrent Write (ARBITRARY CRCW). Simultaneous reads of the same memory cell are allowed. In case of write conflict, one

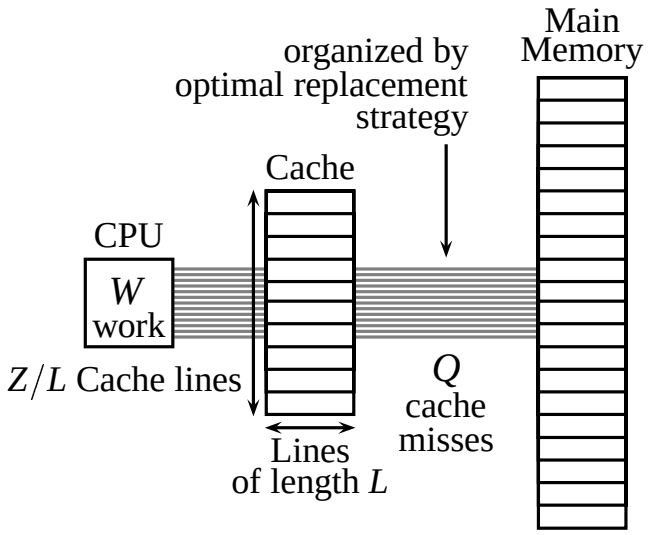

Figure 2.2: The ideal-cache model.

randomly chosen processor is allowed to complete WRITE. An algorithm written for this model should make no assumptions about which processor is chosen in case of write conflict.

COMMON Concurrent Read Concurrent Write (COMMON CRCW). Simultaneous reads of the same memory cell are allowed. In case of write conflict, all processors are allowed to complete WRITE iff all values to be written are equal. An algorithm written for this model should make sure that this condition is satisfied. If not, the algorithm is illegal and the machine state will be undefined.

2.3 The ideal cache model

The *cache complexity* of an algorithm aims at measuring the (negative) impact of memory traffic between the cache and the main memory of a processor executing that algorithm. Cache complexity is based on the *ideal-cache model* shown in Figure 2.2. This idea was first introduced by Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran in 1999 [17]. In this model, there is a computer with a two-level memory hierarchy consisting of an ideal (data) cache of Z words and an arbitrarily large main memory. The cache is partitioned into Z/L *cache lines* where L is the length of each cache line representing the amount of consecutive words that are always moved in a group between the cache and the main memory. In order to achieve *spatial locality*, cache designers usually use $L > 1$ which eventually mitigates

the overhead of moving the cache line from the main memory to the cache. As a result, it is generally assumed that the cache is *tall* and practically that we have

$$Z = \Omega(L^2).$$

In the sequel of this thesis, the above relation is referred as the *tall cache assumption*.

In the ideal-cache model, the processor can only refer to words that reside in the cache. If the referenced line of a word is found in cache, then that word is delivered to the processor for further processing. This situation is literally called a *cache hit*. Otherwise, a *cache miss* occurs and the line is first fetched into anywhere in the cache before transferring it to the processor; this mapping from memory to cache is called *full associativity*. If the cache is full, a cache line must be evicted. The ideal cache uses the optimal off-line cache replacement policy to perfectly exploit *temporal locality*. In this policy, the cache line whose next access is furthest in the future is replaced [5].

Cache complexity analyzes algorithms in terms of two types of measurements. The first one is the *work complexity*, $W(n)$, where n is the input data size of the algorithm. This complexity estimate is actually the conventional running time in a RAM model [1]. The second measurement is its *cache complexity*, $Q(n; Z, L)$, representing the number of cache misses the algorithm incurs as a function of:

- the input data size n,
- the cache size Z, and
- the cache line length L of the ideal cache.

When Z and L are clear from the context, the cache complexity can be denoted simply by $Q(n)$.

An algorithm whose cache parameters can be tuned, either at compile-time or at runtime, to optimize its cache complexity, is called *cache aware*; while other algorithms whose performance does not depend on cache parameters are called *cache oblivious*. The performance of cache-aware algorithm is often satisfactory. However, there are many approaches which can be applied to design optimal cache oblivious algorithms to run on any machine without fine tuning their parameters.

Although cache oblivious algorithms do not depend on cache parameters, their analysis naturally depends on the alignment of data block in memory. For instance, due to a specific type of alignment issue based on the size of block and data elements (See Proposition 1 and its proof), the cache-oblivious bound is an additive 1 away

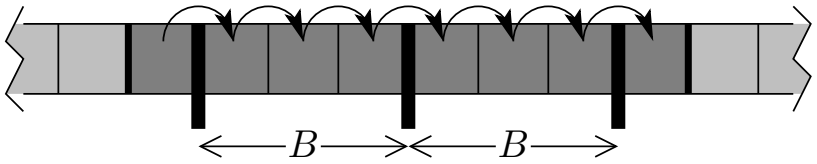

Figure 2.3: Scanning an array of $n = N$ elements, with $L = B$ words per cache line.

from the external-memory bound [36]. However, such type of error is reasonable as our main goal is to match bounds within multiplicative constant factors.

Proposition 1. *Scanning n elements stored in a contiguous segment of memory with cache line size L costs at most $\lceil n/L \rceil + 1$ cache misses.*

Proof ▷ The main ingredient of the proof is based on the alignment of data elements in memory. We make the following observations.

- Let (q, r) be the quotient and remainder in the integer division of n by L. Let u (resp. w) be the total number of words in a fully (not fully) used cache line. Thus, we have $n = u + w$.
- If $w = 0$ then $(q, r) = (\lfloor n/L \rfloor, 0)$ and the scanning costs exactly q; thus the conclusion is clear since $\lceil n/L \rceil = \lfloor n/L \rfloor$ in this case.
- If $0 < w < L$ then $(q, r) = (\lfloor n/L \rfloor, w)$ and the scanning costs exactly $q + 2$; the conclusion is clear since $\lceil n/L \rceil = \lfloor n/L \rfloor + 1$ in this case.
- If $L \leq w < 2L$ then $(q, r) = (\lfloor n/L \rfloor, w - L)$ and the scanning costs exactly $q + 1$; the conclusion is clear again.

◁

2.4 The multi-core machine model

A multi-core architecture consists of a multi-core processor, which is a single computing component with two or more independent processors called "cores". These cores are the basic units that perform read and execute program instructions. These instructions are ordinary CPU instructions like *add*, *move data*, and *branch*. But, importantly, the multiple cores can execute multiple instructions at the same time, which enhance the overall speed of the program execution in the way of parallel computing. A many-core processor is also a multi-core processor in which the number of cores is large enough that traditional multiprocessor techniques are no longer efficient. Manufacturers typically integrate the cores onto a single integrated circuit die, known as a chip multiprocessor or CMP, or onto multiple dies in a single chip package.

The cores in a multi-core architecture can be connected *tightly* or *loosely*. For instance, cores may or may not share caches, and they may implement inter-core communication techniques such as message passing or shared memory. Common network topologies are used to interconnect cores, including bus, ring, two-dimensional mesh and crossbar. *Homogeneous* multi-core systems include only identical cores, whereas, *heterogeneous* multi-core systems have cores which are not identical in practice. Cores on multi-ore systems may implement architecture features such as instruction level parallelism (ILP), vector processing, SIMD or multithreading, similar to those of single-processor systems.

The advantages of multi-core architecture include the fact that *cache coherency* circuitry operates at a much higher clock-rate than in distributed systems where the signals have to travel off-chip. That is, signals between different CPUs (cores) travel shorter distances, and therefore those signals degrade less. As a result, these higher-quality signals with high frequency allow more data to be transferred within a short time period. Moreover, a multi-core processor usually uses less power than multiple coupled single-core processors, this is because of the reduced power required to drive off-chip signals. Furthermore, the cores share some circuitry, like the L2 cache and the interface to the front side bus (FSB). Also, multi-core design produces a product with lower risk of design error than devising a new wider core-design.

Although there are lots of advantages of multi-cores, writing multithreaded programs for this architecture remains quite challenging. Maximizing the utilization of the computing resources in this architecture requires adjustments both to the operating system (OS) support and to existing application software. Also, the performance of multi-core processors to execute applications depends on the use of multiple threads within applications. Finally, raw processing power is not the only constraint on system performance. Several processing cores sharing the same system bus and as a result memory bandwidth limits the real-world performance advantage. If a single core is about to consume whole memory-bandwidth, then for the dual-core, it improves only 30% to 70% of its performance. If memory bandwidth is not a problem, upto 90% improvement is possible. Moreover, if communication between the CPUs is the negligible factor, then it would be possible for an application to execute faster on two CPUs than on one dual-core, which would count as much as 100% improvement.

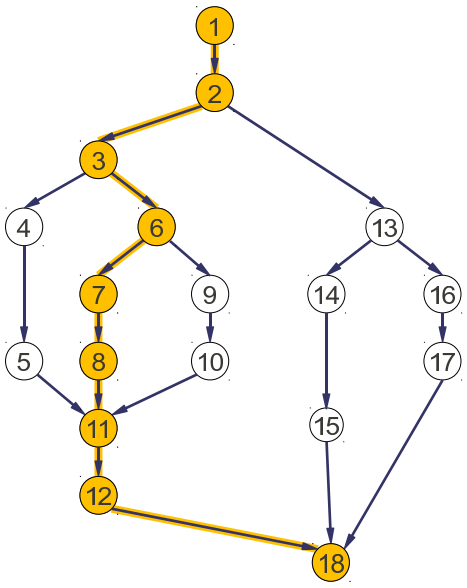

Figure 2.4: A directed acyclic graph (dag) representing the execution of a multithreaded program. Each vertex represents an instruction while each edge represents a dependency between instructions.

2.5 The fork-join parallelism model

The `Cilk++`[1] concurrency platform [7, 18, 45, 15] provides a simple theoretical model called the *fork-join parallelism model* or *dag (direct acyclic graph) model* of multithreading for parallel computation. This model represents the execution of a multithreaded program as a set of nonblocking threads denoted by the vertices of a dag, where the dag edges indicate dependencies between instructions. See Figure 2.4.

In the `Cilk++` terminology, a thread is a maximal sequence of instructions that ends with a `spawn`, `sync`, or `return` statement. These statements are used to denote respectively:

- an *execution flow forking*,
- a *synchronization point*, at which currently running threads must join before the execution flow proceeds further,
- the return point of a function.

A correct execution of a `Cilk++` program must meet all the dependencies in the dag, that is, a thread cannot be executed until all the depending treads have completed. The order in which these dependent threads will be executed on the processors is determined by the *scheduler*.

[1] `http://www.cilk.com`

`Cilk++`s scheduler executes any `Cilk++` computation in a nearly optimal time, see [18] for details. From a theoretical viewpoint, there are two natural measures that allow us to define parallelism precisely, as well as to provide important bounds on performance and speedup which are discussed in the following subsections.

2.5.1 The work law

The first important measure is the *work* which is defined as the total amount of time required to execute all the instructions of a given program. For instance, if each instruction requires a unit amount of time to execute, then the work for the example dag shown in Figure 2.4 is 18.

Let T_P be the fastest possible execution time of the application on P processors. Therefore, we denote the work by T_1 as it corresponds to the execution time on 1 processor. Moreover, we have the following relation

$$T_p \geq T_1/P, \tag{2.1}$$

which is referred as the *work law*. In our simple theoretical model, the justification of this relation is easy: each processor executes at most 1 instruction per unit time and therefore P processors can execute at most P instructions per unit time. Therefore, the *speedup* on P processors is at most P since we have

$$T_1/T_P \leq P. \tag{2.2}$$

2.5.2 The span law

The second important measure is based on the program's *critical-path length* denoted by T_∞. This is actually the execution time of the application on an infinite number of processors or, equivalently, the time needed to execute threads along the longest path of dependency. As a result, we have the following relation, called the *span law*:

$$T_P \geq T_\infty. \tag{2.3}$$

2.5.3 Parallelism

In the fork-join parallelism model, *parallelism* is defined as the ratio of work to span, or T_1/T_∞. Thus, it can be considered as the average amount of work along each point of the critical path. Specifically, the speedup for any number of processors cannot be

greater than T_1/T_∞. Indeed, Equations 2.2 and 2.3 imply that speedup satisfies

$$T_1/T_P \leq T_1/T_\infty \leq P.$$

As an example, the parallelism of the dag shown i n Figure 2.4 is $18/9 = 2$. This means that there is little chance for improving the parallelism on more than 2 processors, since additional processors will often starve for work and remain idle.

2.5.4 Performance bounds

For an application running on a parallel machine with P processors with work T_1 and span T_∞, the `Cilk++` *work-stealing scheduler* achieves an expected running time as follows:

$$T_P = T_1/P + O(T_\infty), \tag{2.4}$$

under the following three hypotheses:

- each strand executes in unit time,
- for almost all parallel steps there are at least p strands to run,
- each processor is either working or stealing.

See [18] for details.

If the parallelism T_1/T_∞ is so large that it sufficiently exceeds P, that is $T_1/T_\infty \gg P$, or equivalently $T_1/P \gg T_\infty$, then from Equation (2.4) we have $T_P \approx T_1/P$. From this, we easily observe that the work-stealing scheduler achieves a nearly perfect linear speedup of $T_1/T_P \approx P$.

2.5.5 Work, span and parallelism of classical algorithms

The work, span and parallelism of some of the classical algorithms in the fork-join parallelism model is shown in Table 2.1.

2.6 Systolic arrays

Systolic arrays are matrix-like regular rows of basic data processing units called cells. Each of these cells relies on arriving data from different directions in the array at regular intervals and being combined [11]. The data streams, which are entering and leaving the ports of the array, are generated by *auto-sequencing memory units* called ASMs. In embedded systems, it is also possible that these data streams be input from and/or output to external components.

Algorithm	Work	Span	Parallelism
Merge sort	$\Theta(n \log_2(n))$	$\Theta(\log_2(n)^3)$	$\Theta(\frac{n}{\log_2(n)^2})$
Matrix multiplication	$\Theta(n^3)$	$\Theta(\log_2(n))$	$\Theta(\frac{n^3}{\log_2(n)})$
Strassen	$\Theta(n^{\log_2(7)})$	$\Theta(\log_2(n)^2)$	$\Theta(\frac{n^{\log_2(7)}}{\log_2(n)^2})$
LU-decomposition	$\Theta(n^3)$	$\Theta(n \log_2(n))$	$\Theta(\frac{n^2}{\log_2(n)})$
Tableau construction	$\Theta(n^2)$	$\Omega(n^{\log_2(3)})$	$\Theta(n^{0.415})$
FFT	$\Theta(n \log_2(n))$	$\Theta(\log_2(n)^2)$	$\Theta(\frac{n}{\log_2(n)})$

Table 2.1: Work, span and parallelism of classical algorithms.

Matrix multiplication might be a good example of the design of systolic algorithm, where one matrix is fed in a row at a time from the top of the array and is passed down the array. The other matrix is fed in a column at a time from the left hand side of the array and passes from left to right. In order to be seen each processor as a whole row and a whole column, dummy values are often passed in when they are not like so. Finally, the multiplication result is stored in the array and can now be output a row or a column at a time, flowing down or across the array.

Lots of applications of systolic arrays include faster input processing, scalability, high throughput etc. The cells are organized in such a way that it can simultaneously process the input, that is, its processing is faster than the conventional computing architecture. Also, this architecture can easily be extended to many more processors according to the requirements of the application. Moreover, systolic arrays offer a way to take certain exponential algorithms and use hardware to make them linear.

The disadvantages of systolic arrays include its complicated design and implementation of hardware and software, highly cost of hardware compared to uniprocessor system, highly specialized for particular applications, difficult to build the system etc.

In the perspective of this thesis, systolic arrays are important since they provide the best known work-efficient parallel algorithm for computing GCDs of univariate polynomials [10]. By work-efficient, we mean that the work is the same complexity class as the Euclidean Algorithm.

Chapter 3

Many-core Machine Model

We propose a model of computations which aims at capturing parallelism overheads (such as communication and synchronization costs) of programs written for modern GPU architectures. We establish a Graham-Brent theorem for this model so as to estimate running time of programs running on p streaming multiprocessors. We evaluate the benefits of our model with three applications. In each case, our model is used to optimize a program parameter controlling overhead.

This chapter is a joint work with M. Moreno Maza and N. Xie.

3.1 Introduction

Designing efficient algorithms targeting implementation on hardware acceleration technologies (multi-core processors, graphics processing units (GPUs), field-programmable gate arrays) creates major challenges for computer scientists. A first difficulty is to define models of computations retaining the features of actual computers that have a dominant impact on program performance. This implies to specify not only the appropriate complexity measures for algorithms but also the relevant parameters for the theoretical machine executing those algorithms. Once different algorithmic solutions for a given problem and a given model of computations are available, a second difficulty is to combine those complexity measures in order to select the "best" algorithm.

In the fork-join parallelism model [6] two complexity measures (the work and the span) and one machine parameter (the number of processors) can be combined in results like the Graham-Brent theorem ([6, 24]) or the Blumofe-Leiserson theorem (Theorems 13 & 14 in [7]) so as to compare algorithm running time estimates. A

variant of this latter theorem is actually supporting successfully the implementation of the parallel performance analyzer called `Cilkview` [35] on multi-core architectures.

With many-core processors, in particular GPUs, one needs to integrate SIMD (Single Instruction Multiple Data) processing into the model. The PRAM model ([66, 22]) has this flavor but it does not have the task-parallelism dimension which is necessary to represent the relations between the different kernels of an application written with the Compute Unified Device Architecture (CUDA) [58]. In addition, the PRAM model fails to retain important features of actual computers related to memory traffic, such as cache complexity ([18, 19]). This latter notion has been proved to be very useful on single-core and multi-core multiprocessors.

An attempt to integrate memory contention into the PRAM model has been made with the QRQW (Queue Read Queue Write) PRAM, defined in [23] by Gibbons, Matias and Ramachandran. The Authors also enhance the Graham-Brent theorem. However, they unify in a single quantity time spent in arithmetic operations and time spent in read/write accesses. We believe that this unification is not appropriate for recent many-core processors, such as GPUs, for which the ratio between one read/write access to the global memory and one floating point operation can be in the 100's.

In a recent paper, Ma, Agrawal and Chamberlain [48] introduce the TMM (Threaded Many-core Memory) model which retains many important characteristics of GPU-type architectures, including several machine parameters such as throughput and coalesced granularity. Moreover, while their running time estimate on P cores is not a Graham-Brent theorem, TMM analysis can order algorithms from slow to fast for many different settings of those machine parameters.

Many works, such as [49, 47], targeting code optimization and performance prediction of GPU programs are related to our work, though these papers do not define an abstract model in support of algorithm analysis.

In this chapter, we propose a many-core machine model (MMM) which aims at optimizing algorithms targeting implementation on GPUs. We insist on the following aspects:

- *Two-level DAG programs.* Defined in Section 3.2, this feature captures the two levels of parallelism (fork-join and SIMD) of CUDA-like programs.

- *Parallelism overhead.* We introduce this complexity measure in Section 3.2.3 with the objective of analyzing communication and synchronization costs.

- *A Graham-Brent theorem.* We combine three complexity measures (work, span

and parallelism overhead) and two machine parameters (size of local memory and data transfer throughput) in order to estimate the running time of an MMM program on p streaming multiprocessors. This result is Theorem 1 in Section 3.2.4.

To demonstrate and evaluate the benefits of our model, we consider three applications for which we have realized an implementation reported in [31]. In each case, the parallelism overhead (and also the work, to a lesser extent) depends on a program parameter. For the first two applications, namely polynomial division and the Euclidean Algorithm (see Chapter 9),

this parameter controls the amount of data transfer between global memory and local memories. For the third application, polynomial multiplication (see Chapter 8)

this parameter controls the amount of branch divergence (see [25] for optimization techniques related to this performance issue) which can also be seen as a parallelism overhead.

For each of these three applications, we apply the following strategy.

1. We determine a value of this program parameter that minimizes parallelism overhead.

2. We check that the work overhead introduced by this optimization technique remains very low. In fact, this work overhead is typically 30% of the work of the non-optimized algorithm.

3. We use our version of Graham-Brent theorem to show that the estimated running time (on p streaming multiprocessors) of the optimized algorithm is asymptotically smaller than that of the non-optimized algorithm. In fact, this speedup is typically a factor of 2, which is confirmed by the experimental study of [31].

Finally, we observe that, in our model, the Euclidean Algorithm reaches the running estimates predicted by the Systolic VLSI Array Model [9]. At the same time, the CUDA code implementing the Euclidean Algorithm developed with our model runs within the same estimate for input polynomials with degree up to 100,000, as reported in [31].

3.2 A many-core machine model

A well-known method for optimizing CUDA programs is to transfer data from the global memory to the local memories in order to reduce redundant memory accesses

with low latency and high throughput. One of the main reasons for this optimization is the fact that global memory latency is approximately 400 to 800 cycles, while local memory latency is only a few cycles. This memory latency difference, when not properly taken into account, may have a dramatic negative impact on program performance. As mentioned in the introduction, this hardware feature of GPUs cannot be captured by the well-studied PRAM model. Indeed, any memory access, as well as any integer arithmetic operation, is performed in *unit time* on a PRAM machine.

This and other limitations of the PRAM model have motivated variants of this model, including our work. Another motivation is the new programming model supported by NVIDIA[1] Kepler architecture, which allows algorithms to run entirely on the device (GPU) without host (CPU) interactions. The model of parallel computations presented in this paper aims at capturing communication and synchronization overheads of programs written for modern GPU architectures, such as NVIDIA Fermi and NVIDIA Kepler.

As specified in Sections 3.2.1 and 3.2.2 below, our many-core machine model (MMM) retains many of the key characteristics of modern GPU architectures and the CUDA programming model. However, in order to support algorithm analysis, with an emphasis on parallelism overheads, as defined in Section 3.2.3, an MMM machine admits a few simplifications and limitations with respect to an actual GPU device. We justify those choices in Section 3.2.5 and explain how more general models can be reduced to ours.

3.2.1 Many-core machine characteristics

Architecture. An MMM machine possesses an unbounded number of *streaming multiprocessors* (SMs) which are all identical. Each SM has a finite number of processing cores and a fixed-size local memory. An MMM machine has a 2-level memory hierarchy, comprising an unbounded global memory with high latency and low throughput while the SM local memories have low latency and high throughput.

Programs. An *MMM program* is a directed acyclic graph (DAG) whose vertices are kernels and where edges indicate dependencies, similarly to the instruction stream DAGs of the fork-join multithreaded parallelism model [6]. A *kernel* is a SIMD (single instruction multithreaded data) program decomposed into a number of thread-blocks. Each *thread-block* is executed by a single SM and each SM executes a single thread-block at a time. Similarly to a CUDA program, an MMM program specifies for each

[1] `http://www.nvidia.com/`

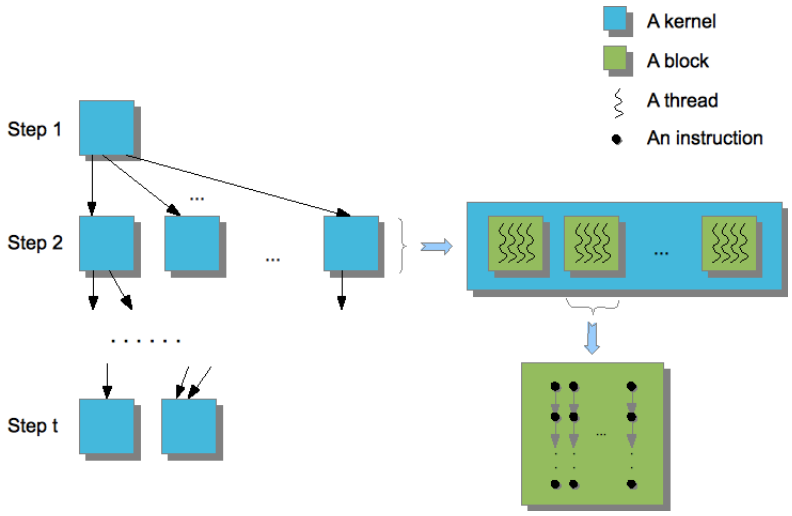

Figure 3.1: Overview of a many-core machine program

kernel call the number of thread-blocks and the number of threads per thread-block. The different types of components of an MMM program are depicted on Figure 3.1.

Scheduling and synchronization. At run time, an MMM machine schedules thread-blocks (from the same or different kernels) onto SMs, based on (1) the dependencies specified by the edges of the DAG and, (2) the hardware resources required by each thread-block. Threads within a thread-block can cooperate with each other via the local memory of the SM running the thread-block. Meanwhile, thread-blocks interact with each other via the global memory. In addition, threads within a thread-block are executed physically in parallel by an SM. Meanwhile, the programmer cannot make any assumptions on the order in which thread-blocks of a given kernel are mapped to the SMs. This restriction allows MMM programs to run correctly on any fixed number of SMs, similarly to a CUDA program.

Memory access policies. All threads of a given thread-block can access simultaneously any memory cell of the local memory or the global memory: read/write conflicts are handled by the CREW (concurrent read and exclusive write) policy. However, read/write requests to the global memory by two different thread-blocks cannot be executed simultaneously. In case of simultaneous request, one thread-block is chosen randomly and served first, then the other thread-block is served.

For the purpose of analyzing program performance, we define two *machine parameters*:

U**:** Time (expressed in clock cycles) to transfer one machine word between global memory and the local memory of any SM. Thus, $1/U$ is a throughput.

Z**:** Size (expressed in machine words) of the local memory of each SM.

To be precise, the throughput $1/U$ satisfies the following property. If r and w are the number of words respectively read and written to the global memory by one thread of a thread-block B, then the total time T_D spent in data transfer between the global memory and the local memory of an SM executing B satisfies

$$T_D \leq (r + w)U. \tag{3.1}$$

We observe that most phenomena that ease or limit data transfer (coalesced accesses to global memory, local memory bank conflicts, partition camping, etc) have an impact on running time which is proportional to the amount of transferred data. This allows us to claim that the throughput $1/U$ combines (or unifies) these different phenomena.

Similarly, the local memory size Z unifies in one parameter different characteristics of an SM and, thus, of a thread-block. Indeed, each of the following quantities is necessarily at most equal to Z: the number of cores of an SM, the number of threads of a thread-block, the amount of words in a data transfer between the global memory and the local memory of an SM.

Relation (3.1) calls for another comment. One could expect the introduction of a third machine parameter, say V, such that, if ℓ is the number of *local operations* (arithmetic operations, reads/writes in the local memory) performed by one thread of the thread-block B, then the total time T_A spent in local operations by an SM executing B would satisfy

$$T_A \leq \ell V. \tag{3.2}$$

As a consequence, for the total running time T of the thread-block B, we would have

$$T = T_A + T_D \leq \ell V + (r + w)U. \tag{3.3}$$

Instead of introducing this third machine parameter V, we let $V = 1$, which is equivalent to a change of coordinates.

3.2.2 Many-core machine programs

As specified above, each MMM program $\mathcal{P}$ is modeled by a directed acyclic graph $(\mathcal{K}, \mathcal{E})$, called the *kernel DAG* of $\mathcal{P}$, where each node $K \in \mathcal{K}$ represents a kernel and each edge $E \in \mathcal{E}$ represents a kernel call which must precede another kernel call. To

be precise, a kernel call can be executed provided that all its predecessors in the DAG $(\mathcal{K}, \mathcal{E})$ have completed their execution.

Recall that each kernel decomposes into one or more thread-blocks and that all threads within a given kernel execute the same serial program, but with possibly different input data. In addition, all threads within a thread-block are executed physically in parallel by an SM. It follows that MMM kernel code needs no synchronization statement, like CUDA's `__syncthreads()`.

This has two consequences. First, each thread in a thread-block is either submitting read/write requests to the global memory or, executing local operations. This justifies Relation (3.3). A second consequence is the fact that the synchronization overheads of an MMM program are included in the scheduling costs of the thread-blocks onto the SMs. We shall assume that those latter costs depend linearly on the number of thread-blocks and the sum over all thread-blocks of the amount of data transferred by one thread. Indeed, this second quantity can be used to estimate the size of the code of a thread-block. Therefore, synchronization overheads of an MMM program can be incorporated in the data transfer time. This *key observation* helps understanding the complexity measures introduced in Section 3.2.3.

Since each kernel of the program $\mathcal{P}$ decomposes into a finite number of thread-blocks, we map $\mathcal{P}$ to a second graph, called the *thread block DAG* of $\mathcal{P}$, whose vertex set $\mathcal{B}(\mathcal{P})$ consists of all thread-blocks of the kernels of $\mathcal{P}$ and such that (B_1, B_2) is an edge if B_1 is a thread-block of a kernel preceding the kernel of B_2 in $\mathcal{P}$. This second graph is associated two important quantities:

$N(\mathcal{P})$**:** number of vertices in the thread-block DAG of $\mathcal{P}$,

$L(\mathcal{P})$**:** critical path length (that is, the length of the longest path) in the thread-block DAG of $\mathcal{P}$.

For the purpose of analyzing program performance, we define five *program parameters*, summarized in Table 3.1.

We also define five algorithm parameters, shown in table 2: n is the input size of a many-core machine program; z is the maximum amount of the local memory per thread-block; q is the number of threads per thread-block; d is the number of thread-blocks needed during a parallel step; and π is the number of parallel steps of a many-core machine program.

Parameter	Description
n	Input size in machine words
z	Maximum number of words of local memory allocated per thread-block
q	The number of threads per thread-block
d	The maximum number of thread-blocks in a parallel step
π	The number of parallel steps

Table 3.1: Algorithm parameters

3.2.3 Complexity measures for the many-core machine model

Consider, as before, an MMM program $\mathcal{P}$ given by its kernel DAG $(\mathcal{K}, \mathcal{E})$. Let $K \in \mathcal{K}$ be any kernel of $\mathcal{P}$ and B be any thread-block of K. We define the *work* of B, denoted by $W(B)$, as the total number of local operations performed by the threads of B. We define the *span* of B, denoted by $S(B)$, as the maximum number of local operations performed by a thread of B. We assume that each thread of B reads r words and writes w words from the global memory. Then, we define the *overhead* of B, denoted by $O(B)$, as $(r + w)U$. The *work* $W(K)$ of the kernel K is defined as the sum of the works of its thread-blocks. The *span* (resp. *overhead*) $S(K)$ (resp. $O(K)$) of the kernel K is defined as the maximum (resp. sum) of the spans (resp. overheads) of its thread-blocks.

We consider now the entire program $\mathcal{P}$. The *work* $W(\mathcal{P})$ of $\mathcal{P}$ is defined as the total work of all its kernels

$$W(\mathcal{P}) = \sum_{K \in \mathcal{K}} W(K).$$

Regarding the graph (K, E) as a weighted-vertex graph where the weight of a vertex $K \in \mathcal{K}$ is its span $S(K)$, we define the weight $S(\gamma)$ of any path γ from the first executing kernel to a last executing kernel as

$$S(\gamma) = \sum_{K \in \gamma} S(K).$$

Then, we define the *span* $S(\mathcal{P})$ of the program $\mathcal{P}$ as

$$S(\mathcal{P}) = \max_{\gamma} S(\gamma).$$

Regarding the graph (K, E) as a weighted-vertex graph, where the weight of a vertex K is its *overhead* $O(K)$, we define the *overhead* $O(\alpha)$ of an anti-chain α of (K, E) as

$$O(\alpha) = \sum_{K \in \alpha} O(K),$$

Finally, we define the *overhead* $O(\mathcal{P})$ of $\mathcal{P}$ as the sum of the $O(\alpha)$'s among all anti-chains α in (K, E), that is,

$$O(\mathcal{P}) = \sum_{\alpha} O(\alpha).$$

Observe that, according to Mirsky's theorem [50], the number π of parallel steps in $\mathcal{P}$ (i.e. anti-chains in $(\mathcal{K}, \mathcal{E})$) is equal to the maximum length of a path in $(\mathcal{K}, \mathcal{E})$ from the first executing kernel to a last executing kernel.

3.2.4 A Graham-Brent theorem with overhead

Theorem 1. *We have the following estimate for the running time T_P of the program $\mathcal{P}$ when executed on p SMs,*

$$T_P \leq (N(\mathcal{P})/p + L(\mathcal{P}))C(\mathcal{P}) \tag{3.4}$$

where $C(\mathcal{P}) = \max_{B \in \mathcal{B}(\mathcal{P})} (S(B) + O(B))$.

The proof is similar to that of the original result. One observes that the total number of *complete steps* (for which p thread-blocks can be scheduled by a greedy scheduler) is at most $N(\mathcal{P})/p$ while the number of *incomplete steps* is at most $L(\mathcal{P})$. Finally, $C(\mathcal{P})$ is an obvious upper bound for the running of every step, complete or incomplete.

3.2.5 Justification of the many-core machine model

In the new programming model of CUDA, a kernel that can be executed is launched by its predecessor at any time, while its predecessor waits to synchronize until the kernel has completed. To analyze this type of programs, we adjust the situation into the way our model can deal with, shown in Figure 3.2. If a child kernel is called within a parent kernel as Figure 3.2(a), we divide the parent kernel into part A and part B, such that part B of the parent kernel starts the same time as the child kernel, and the rest of the parent kernel is part A in the way of Figure 3.2(b). With this effort, our model can simulate the new features of the CUDA programming model.

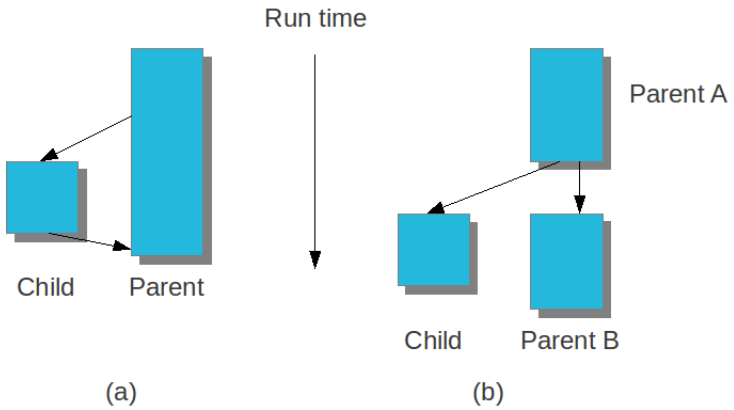

Figure 3.2: Adjust any program into the DAG of many-core machine model

Chapter 4

Cache-oblivious Counting Sort Algorithm

In this chapter, we propose a cache-oblivious counting sort algorithm. Cache complexity estimates of both classical and our proposed cache-oblivious counting sort algorithm are provided considering the ideal cache model. We have implemented these algorithms and compared them by experimentation. Based on those cache complexity results and experimental results, we can say that our cache-oblivious counting sort algorithm is promising.

This chapter is a joint work with M. Moreno Maza.

4.1 Introduction

The counting sort algorithm sorts n non-negative integers in the range $[0, r-1]$ in linear time with respect to $n + r$ considering the RAM model [13]. Algorithm 1 describes the classical counting sort algorithm. Observe that the counting sort algorithm does not require a comparand function. Thus it does not have time complexity lower bound of the form $O(n \log n)$ like any comparison based sorting algorithms.

In this chapter, time and space complexity are computed considering the RAM model with memory holding a finite number of s-bit words, for a fixed s [64]. We assume each of the integers to be sorted can be stored within one machine word. Our cache complexity results are computed considering the ideal cache model [17] with an ideal cache of Z words and for which each cache line holds L words.

The main contribution of this work is a cache-oblivious counting sort algorithm. We also compute the cache complexities of both classical and cache-oblivious counting

sort algorithms. We validate theoretical results by experimentation. Similar work can be found in [61] but the Authors do not make use of the ideal cache model.

The organization of this chapter is as follows. we first describe the classical counting sort algorithm along with its cache complexity in Section 4.2. We then present our cache-oblivious counting sort algorithm along with its cache complexity in Section 4.3 followed by experimental results in Section 4.4. We conclude the chapter with some remarks on our approach.

4.2 The classical counting sort algorithm

Algorithm 1 is the classical counting sort algorithm. Proposition 2 computes the cache complexity of this algorithm.

Algorithm 1: CountingSort(A, n, r)

Input: A is an array of length n that holds n integers in the range $[0, r-1]$.
Output: Array B of length n, where the integers in A are sorted.

```
1  initialize an array C of length r of with zeros;
2  for i = 0; i < n; i = i + 1 do
3      C[A[i]] = C[A[i]] + 1;
4  t = 0;
5  for i = 0; i < r; i = i + 1 do
6      c = C[i];
7      C[i] = t;
8      t = t + c;
9  for i = 0; i < n; i = i + 1 do
10     B[C[A[i]]] = A[i];
11     C[A[i]] = C[A[i]] + 1;
12 return B;
```

Proposition 2. *Given an array A of length n that holds n non-negative integers within the range $[0, r-1]$, the total number of cache misses in Algorithm 1 is at most $3n + 2n/L + 2r/L + 4$, where both n and r are small enough such that none of A or B or C can be stored into the cache entirely.*

Proof ▷ We follow the pseudo-code of Algorithm 1 and count the number of cache misses.

Initializing C with zeros in line 1: This involves traversing C in a regular way (i.e. one slot after another from left to right). Thus, this causes $r/L + 1$ cold misses.

Computing frequency of the integers of A into C in lines 2-3: This involves traversing A in a regular way. However, C is accessed n times in an irregular way. So the total number of cache misses is $(n/L+1)+n$ in the worst case. The latter n cache misses are due to both capacity and conflict misses in accessing C irregularly.

Computing the cumulative frequency in C in lines 5-8: This involves traversing C in a regular way, thus causing $r/L+1$ cache misses.

Creating sorted array B in lines 9-11: Finally, populating the sorted array B involves traversing the three arrays A , B and C. The array A is accessed in a regular fashion. However, the n accesses to C and B are irregular. So the total number of cache misses is $(n/L+1)+2n$ in the worst case. The latter $2n$ cache misses are due to both capacity and conflict misses in accessing C and B irregularly. □

Irregular access make the performance of counting sort algorithm poor in terms of cache misses, for large n or r, as we can see in Section 4.4. In Section 4.3, we propose a cache-oblivious counting sort algorithm in order to reduce the cache complexity of this algorithm.

4.3 Cache-oblivious counting sort algorithm

In Proposition 3, we first estimate the number of cache misses of Algorithm 1 with small r. This leads us to the notion of a *bucketed array* introduced in Definition 1. We show the cache complexity is reduced during counting sort algorithm if the input A has this property. Finally, we propose a preprocessing step (Algorithm 2) which rearranges the integers in A in such a way that it can be bucketed.

Proposition 3. *The cache complexity of Algorithm 1 is at most $3n/L+r+r/L+3$ for $r < Z/(1+L)$, where n is small enough such that none of A or B can be stored into the cache entirely.*

Proof ▷ We traverse C in a regular way, in line 1 of Algorithm 1, in order to initialize it by zeros. Thus it causes $r/L+1$ cold misses. Observe that C is stored entirely into the ideal cache after this step. It is possible because $r < Z/(1+L)$. The total number of cache misses for traversing A for computing frequency of the integers in lines 2-3 is $n/L+1$. Because this traversal is also regular. We do not observe any cache misses for accessing C in this step. Because the ideal cache evicts a cache line that stores elements of A whenever it is full due to the cache replacement policy of it. For the same reason, we do not observe any cache misses for accessing C in lines 5-8.

In lines 9-11, the cache misses due to accessing A exhibit the same cache complexity as described in the proof of Proposition 2. Writing sorted array B means writing (in a linear traversal) r consecutive arrays, of possibly different sizes, but with total size n. Thus, because of possible misalignments between those arrays and their cache-lines, this writing procedure can yield at most $n/L + r$ cache misses (and not just $n/L + 1$). It is possible if the ideal cache has at least r cache lines. We know the ideal cache has Z/L cache lines and we have $r < Z/(1 + L) < Z/L$. Observe C is stored entirely into the cache before the algorithm enters line 9. To keep C stored entirely into the ideal cache during the execution of lines 9-11 and to observe at most $n/L + r$ cache misses for writing the sorted array B, we require an ideal cache where,

$$(r + rL) < Z. \tag{4.1}$$

Equation 4.1 holds for our case, because dividing both sides by $(1 + L)$ yields

$$r < Z/(1 + L).$$

□

From Proposition 3, we can say that counting sort is ideal for sorting integers with small range. The hypothesis of Proposition 3 leads us to introduce the following notion.

Definition 1. *Given an array A of length n with non-negative integer entries in the range $[0, \ell m - 1]$, where ℓ and m are positive integers. We say that A is* m-bucketed *if for all $i = 0, 1 \cdots, (\ell - 1)$, the integers in the sub-range $[im, (i + 1)m - 1]$ are kept together in A. In other words, for all $j = 0, 1 \cdots, (\ell - 1)$, every integer of A lying in the sub-range $[jm, (j + 1)m - 1]$ appears in A before every integer of A lying in the sub-range $[(j + 1)m, \ell m - 1]$.*

In Proposition 4, we estimate the cache complexity of the counting sort algorithm when the input array A is *m-bucketed*.

Proposition 4. *Let the input array A be* m-bucketed *as described in Definition 1, where $m < Z/(1+L)$. The cache complexity of Algorithm 1 is at most $3n/L+4r/L+(m+2)\ell+4$, where n is small enough such that none of A or B can be stored into the cache entirely.*

Proof ▷ Algorithm 1 initializes C with zeros, computes cumulative frequencies in C in line 1 and lines 5-8 respectively. These two steps exhibit the same cache complexity

as described in the proof of Proposition 2. Moreover while counting the frequencies of the integers in lines 2-3 and creating the sorted array B in the last for-loop of the algorithm, the traversals of A are also regular. Thus it has the same cache complexity as in the proof of Proposition 2. Together we have $2n/L + 2r/L + 4$ cache misses for accessing A and C twice in a regular fashion.

For lines 2-3, accessing C for computing the frequency of the integers in A means traversing ℓ consecutive arrays one after another, of possibly different sizes (at most m), but with total size r. In lines 9-11, we need to access C in the same fashion. Thus the total number of cache misses is at most $2r/L + 2\ell$ for accessing C in these two for-loops.

Let us consider a sub-range $[im, (i+1)m - 1]$ of A for $i = 0, 1 \cdots, (\ell - 1)$. Let n_i be the total number of integers in this sub-range. From the proof of Proposition 3, we can say that, the cache complexity for writing into B due to this sub-range is at most $n_i/L + m$. So in total we have at most $n/L + m\ell$ cache misses for writing B. □

Now, we describe our cache-oblivious counting sort algorithm. Our proposed algorithm has a preprocessing step, stated in Algorithm 2. This algorithm calls Algorithm 3 iteratively. Assumption 1 below gathers two relations among the quantities r, Z and L. The first one is made in a sake of simplicity. The second one is taken from Definition 1.

Assumption 1. *We assume that r is of the form $r = m^{u+1}$, for some non-negative integer u and a positive integer m such that we have $m < Z/(1 + L)$.*

Algorithm 2: PreprocessingCounting(A, n, m, r)

Input: A is an array of length n holding non-negative integers in the range $[0, r - 1]$ and m is a positive integer as defined in Assumption 1.

Output: the array A which was overwritten such that A is now m-bucketed.

```
1 r' = r;
2 while r' > m do
3 |   G = PartitionFurther(A, n, m, r, r');
4 |   r' = r'/m;
5 |   copy G into A;
6 return A;
```

The cache complexity of Algorithm 3 is given in Proposition 6.

Proposition 5. *Algorithm 3 works correctly.*

PROOF ▷ For simplicity, we describe the proof for the first bucket. Consider the sub-array $A[0 \dots j]$, where $0 \leq j < n$, stores the first bucket or the sub-range $[0, r' - 1]$. The first iteration of the top for-loop of Algorithm 3 is responsible for rearranging these integers such that $A[0 \dots j]$ is m-bucketed into $G[0 \dots j]$.

We follow the pseudo-code of Algorithm 2 for the first iteration and check its correctness. Observe each integer in $A[0 \dots j]$ is treated as an integer in the range $[0, m - 1]$ when we apply *floor* function after division in both line 9 and 16. In lines 8-10, we compute the frequency of integers in $A[0 \dots j]$ into F. In lines 11-14, we compute the cumulative frequencies in F as in lines 5-8 of Algorithm 1. Finally, in lines 15-19, we write into the sub-array $G[0, \dots, j]$ as m-bucketed integers in $A[0 \dots j]$ with the help of F.

The other $r/r' - 1$ iterations in Algorithm 2 can be proved in the same way. □

Algorithm 3: PartitionFurther(A, n, m, r, r')

Input: An array A of length n holding n positive integers in the range $[0, r - 1]$ and a positive integer m as defined in Assumption 1. Moreover, the array A is assumed to be (r/r')-bucketed.

Output: An array G of length n holding the same n entries as A but re-ordered in a such way such that G is $((r/r')m)$-bucketed.

```
1  y = r/r';
2  d = r'/m;
3  create an array F of length m;
4  x = 0;
5  G is an array of length n;
6  for (i = 0; i < y; i = i + 1) do
7      make all entries of F as 0;
8      for (j = x; j < n ∧ A[j] < r'(i + 1); j = j + 1) do
9          F[⌊(A[j] − r'i)/d⌋] = F[⌊(A[j] − r'i)/d⌋] + 1;
10     t = 0;
11     for (j = 0; j < m; j = j + 1) do
12         c = F[j];
13         F[j] = t;
14         t = t + c;
15     for (j = x; j < n ∧ A[j] < r'(i + 1); j = j + 1) do
16         k = ⌊(A[j] − r'i)/d⌋;
17         G[F[k] + x] = A[j];
18         F[k] = F[k] + 1;
19         x = x + 1;
20 return G;
```

Proposition 6. *In Algorithm 3, we can observe at most* $3n/L+m/L+(r/r')(2+m)+1$ *cache misses, where* n *is small enough such that none of* A *or* G *can be stored into the cache entirely.*

PROOF ▷ We follow the pseudo-code of the algorithm and count the number of cache misses during the first iteration of the top for-loop of Algorithm 3. Like the proof of Proposition 5, consider the sub-array $A[0 \dots j]$, where $0 \leq j < n$, stores the first bucket or the sub-range $[0, r' - 1]$. The first iteration of the top for-loop of Algorithm 3 is responsible for rearranging these integers such that $A[0 \dots j]$ is m-bucketed into $G[0 \dots j]$.

We first initialize F with zeros in line 7. This involves traversing F in a regular way. Thus, this causes $m/L + 1$ cold misses. In lines 8-9, we compute the number of integers in $A[0 \dots j]$ that falls into each bucket. This involves traversing $A[0 \dots j]$ in a regular way. So the number of cache misses for accessing $A[0 \dots j]$ is at most $j/L + 1$ during this step. As $m < Z/(1 + L)$, like in the proof of Proposition 3, we do not observe any cache misses for accessing F. For the same reason, we do not observe any cache misses for accessing F in the rest of the execution of this algorithm.

Finally, in lines 15-19, when we write the m-bucketed integers in $G[0 \dots j]$, we need to traverse $A[0 \dots j]$ in a regular fashion again. Moreover, we need to write the m-bucketed integers into $G[0 \dots j]$. This refers to write (in a linear traversal) m consecutive arrays, of possibly different sizes, but with total size j. From the proof of Proposition 3, we can say that the cache complexity for writing into $G[0 \dots j]$ is at most $j/L + m$. Thus the total number of cache misses for accessing $A[0 \dots j]$ and $G[0 \dots j]$ in this iteration is at most $3j/L + m + 2$. So the cache complexity for accessing A and G in Algorithm 3 is at most $3n/L + (m+2)(r/r')$, as the top for-loop runs for r/r' times.

□

The execution of Algorithm 2 is considered as the preprocessing step of counting sort algorithm. Proposition 7 gives the total number of cache misses during preprocessing step of counting sort algorithm.

Proposition 7. *In worst case, the cache complexity of Algorithm 2 is* $3nu/L+mu/L+(2+m)\frac{m^u-1}{m-1}+u$, *where* n *is small enough such that none of* A *or* G *can be stored into the cache entirely.*

PROOF ▷ Algorithm 2 calls Algorithm 3 for u times as $r = m^{u+1}$. In the j-th call of Algorithm 3 from Algorithm 2 for $j = 0, \dots, u - 1$, the value of r' is r/m^j.

Thus $r/r' = m^j$. From Proposition 6, we can say the worst case cache complexity of Algorithm 3 is

$$3n/L + m/L + m^j(2+m) + 1. \tag{4.2}$$

Thus the worst case cache complexity of Algorithm 2 is

$$3nu/L + mu/L + +u + (2+m)\sum_{j=0}^{u-1} m^j. \tag{4.3}$$

We obtain the worst case cache complexity of Algorithm 2 by applying geometric summation rule in Equation 4.3.

□

Finally, Proposition 8 counts the total number of cache misses for our proposed cache-oblivious counting sort algorithm.

Proposition 8. *Given an array A of length n that holds n non-negative integers within the range $[0, r-1]$. We preprocess A by Algorithm 2. After this preprocessing, A becomes m-bucketed. We then apply Algorithm 1 to sort A. By cache-oblivious counting sort algorithm, we refer to this preprocessing step and applying the classical counting sort algorithm. In worst case, the total cache misses of our proposed cache oblivious counting sorting algorithm is $O(\frac{n \log r}{L} + r)$, where n is small enough such that none of A or B (for Algorithm 1 and 2) or G (for Algorithm 2) can be stored into the cache entirely.*

Proof ▷ Following Proposition 7 and 4 the total number of cache misses of our proposed cache-oblivious counting sort algorithm is

$$3n(u+1)/L + mu/L + u + (2+m)\left(\frac{m^u - 1}{m-1} + \ell\right) + 4r/L + 4. \tag{4.4}$$

The dominating terms of Equation 4.4 are $3n(u+1)/L$ and $(2+m)(\frac{m^u-1}{m-1} + \ell)$. As $r = m^{u+1}$, and $m < Z/(1+L)$, we can write $3n(u+1)/L = O(\frac{n \log r}{L})$. The other dominating term of the equation can also be written as $(2+m)(\frac{m^u-1}{m-1} + \ell) = O(r)$, for the same reasons and $\ell = r/m$.

□

n	classical counting sort	cache-oblivious counting sort (preprocessing + sorting)
100000000	13.74	4.66 (3.04 + 1.62)
200000000	30.20	9.93 (6.16 + 3.77)
300000000	50.19	16.02 (9.32 + 6.70)
400000000	71.55	22.13 (12.50 +9.63)
500000000	94.32	28.37 (15.71 + 12.66)
600000000	116.74	34.61 (18.95 + 15.66)

Table 4.1: CPU times in seconds for both classical and cache-oblivious counting sort algorithm.

4.4 Experiments

We run our experiments on an Intel(R) Core(TM) i7 CPU @ 2.93GHz. It has L2 cache of 8MB[1]. For smaller n, it is expected that we might not get enough speed up. So we take large n and choose $r = n$. We fix $m = 10^6$. We limit r (or n) such that $n, r < m^2$. Thus we need to call Algorithm 3 from Algorithm 2 one time. Table 4.1 shows the CPU time for different n. CPU time for cache-oblivious algorithm includes preprocessing time.

4.5 Conclusion

The preprocessing step of our proposed cache-oblivious counting sort algorithm relies on division operation, which is computationally expensive. It makes the implementation slower. Moreover, if r is big, we have to call Algorithm 3 a number of times. Each step involves with $O(n)$ division operations. That is why, in our experiments, we keep our r small enough such that we need to call Algorithm 3 one time. Still, our implementation gives a way to make the counting sort algorithm efficient and it has room for further improvement. If anyone needs a stable sort routine like counting sort, then he or she might be benefited with this implementation.

One interesting conclusion of this work is, though classical counting sort algorithm has linear complexity, we can do preprocessing on input to reduce cache misses and the cost of the preprocessing can be amortized by the savings from reduced cache misses. In Chapter 5, we propose a new integer sorting algorithm. In our proposed

[1] `http://www.intel.com`

sorting algorithm, we need a stable sort in intermediate steps. Our cache-oblivious counting sort algorithm can be a candidate for that.

Chapter 5

A New Integer Sorting Algorithm

Consider the following problem.

Problem 1. *(Sorting big integers) Consider n integers such that any two of those integers cannot be compared in constant time (counting machine word operations) on the targeted computer. In particular, at least one of those integers requires more than one machine word of storage. The proposed problem is to design an efficient algorithm for sorting those n integers.*

In this chapter, we propose a new sorting algorithm to solve this problem. Our algorithm is a practical algorithm in the sense that it can be implemented on a real computer by using programming languages like C or C++.

This chapter is a joint work with M. Moreno Maza.

5.1 Introduction

Sorting is one of the most fundamental problems in computer science. It is well known that a lower bound time complexity of any comparison-based sorting algorithm is $O(n \log\, n)$ (counting comparisons) where n is the number of keys to be sorted. This time complexity result assumes that there exists a comparand function that works in constant time for those keys. Otherwise, assuming that each key is represented by m bits this lower bound becomes $O(f(m)n \log n)$, where $f(m)$ is the cost of the comparand function for any pair of keys.

However, we still do not know the lower bound for integer sorting. This sorting is important because all objects are represented by binary strings on real computer systems. The interpretation of this binary string can be an integer. Some examples include floating point number representation, binary reflected Gray code or character

strings [2]. Kirkpatrick and Reisch in [39] describe the following problem regarding integer sorting.

Problem 2. *"For what ranges of inputs can we construct practical $o(n \log n)$ integer sorting algorithms?"*

In [2], the authors ask a similar question on integer sorting.

Problem 3. *"Can integers be sorted in linear expected time for all word lengths?"*

In [27], [26] [2], integer sorting problem are discussed. In these works, integers that require a fixed number of words are considered.

In this chapter, we propose a practical sorting algorithm for large integers. The organization of the chapter is as follows. We first define some notations and symbols in Section 5.2. We follow these notations and symbols throughout the chapter. In Section 5.3, we compute the cost for a comparand function that can pick the larger object between two binary objects. Section 5.4 describes our proposed sorting algorithm. In Section 5.5, we describe the time and space complexity of our sorting algorithm. Conclusions are given in Section 5.6.

5.2 Notations

Suppose we have a list of n non-negative integers $U = [a_0, \ldots, a_{n-1}]$, that we want to sort. Each integer a_i, for $i = 0, \ldots (n-1)$, is represented by m bits, where $a_i[0]$ is the most significant bit and $a_i[m-1]$ is the least significant bit. Let $1/p$ be the probability that a bit of a binary representation of an integer in U is 1.

First disagree bit. Let a_i and a_j be two m bit integers, for $0 \leq i, j < n$. If a_i and a_j are not equal then there exists an integer $0 \leq k < m$, for which, $a_i[k] \neq a_j[k]$ and $a_i[s] = a_j[s]$ for each $s < k$. For obvious reason, we call k the *first disagree bit* of a_i and a_j.

Sparse integers. By sparse integer, we mean an integer that has fewer 1s in its binary representation. In this chapter, we assume that a sparse integer is stored by its bit indices those are 1s.

Representation of a_i. For our proposed algorithm, each integer a_i in $U = [a_0, \ldots, a_{n-1}]$ is represented by three values $a_i.g$, $a_i.w$, $a_i.z$ and a list of positive integers $a_i.v$ described below.

1. Define $a_i.g$ as the position of a_i in U. Note that, for any permuted array L of U, suppose $L[j].g = i$ for $0 \leq j < n$. It means the j-th integer in L is the i-th integer in U. Furthermore observe $a_i.g = i$ for each i.

2. The meaning of $a_i.w$ is technical and will be presented in Definition 2.

3. Similarly, $a_i.z$ is defined in Definition 2.

4. The list $a_i.v$ stores, in ascending order, the indices of the 1s in the binary representation of a_i. For example, let, $m = 10$ and $a_i = 50$, which is 0000110010 in 10-bit binary. Then, $a_i.v = [4, 5, 8]$.

   The number of bit operations required to create such list for an m-bit integer is $O(m)$. Data structures for sparse binary objects usually store the information about the 1s only. For such data structures, this list can be created directly. Algorithm 4 describes how our proposed sorting algorithm can interact with this list. Observe that the expected number of words required to store $a_i.v$ is $O(m/p)$.

Algorithm 4: ExploreV$(a_i.v, u, m)$

Input: $a_i.v$ is the list of indices of 1s in integer a_i. a_i is an m bit integer and u is an integer.

Output: return the index of u-th 1 (if present in v_i) in a_i. Return -1 or m in exceptional cases.

```
1 if u = −1 then
2     return −1;
3 if u ≥length(a_i.v) then
4     return m;
5 return a_i.v[u];
```

5.3 Cost of the comparand function for large integers

In this section, we estimate the cost of a comparand function that works with two m-bit integers a_i and a_j for $0 \leq i, j < n$. Lemma 1 estimates the value of their first disagree bit. Corollaries 1 and 2 estimate the cost of comparand functions for both dense and sparse integers based on this lemma, respectively. These corollaries also

estimate the lower bounds of the time complexity of the comparison based sorting algorithms inspired by Lemma 1.

Lemma 1. *The expected value of the first disagree bit of two integers a_i and a_j, for $0 \leq i, j < n$ and $a_i \neq a_j$, is $\frac{p^2}{2(p-1)}$, where $1/p$ is the probability that a bit of a binary representation of an integer is* 1.

PROOF ▷ For any two m-bit integers a_i, a_j and randomly chosen bit $s \in \{0, \ldots, (m-1)\}$, define the two probabilistic events SUCCESS ($a_i[s] \neq a_j[s]$) and FAILURE ($a_i[s] = a_j[s]$). The probability of the event SUCCESS is $2(\frac{1}{p} \cdot (1 - \frac{1}{p}))$ or $\frac{2(p-1)}{p^2}$. Consider a succession of experiments, whereby we increase s from 0 until we obtain a SUCCESS. Obviously the random variable for the value of s giving the first SUCCESS is geometrically distributed with expected value $\frac{p^2}{2(p-1)}$. This corresponds to the expected first disagree bit. □

Corollary 1. *The comparand function for dense integers is expected to work in $O(\frac{p^2}{2(p-1)})$ bit operations. So the lower bound time complexity of any comparison based sorting algorithm for sorting n dense integers is $O(\frac{p^2}{2(p-1)} n \log n)$.*

Corollary 2. *The comparand function for sparse integers is expected to work in $O(\frac{p}{2(p-1)})$ bit operations because the expected number of 1s in $\frac{p^2}{2(p-1)}$ bits is $\frac{p}{2(p-1)}$. So the lower bound time complexity of any comparison based sorting algorithm for sorting n sparse integers is $O(\frac{p}{2(p-1)} n \log n)$.*

5.4 A new sorting algorithm

Our sorting algorithm works by partitioning the given integers in U successively until all parts are singleton. Each partition can be viewed as a list of lists of integers, which we order by certain rules described in Section 5.4.1.

Let $A^0 = [U]$ be our first list of lists (it has one list). Assume $A^k = [A^k_0, A^k_1, \ldots A^k_x]$, for $0 \leq k$ and $0 \leq x$, is the k-th partition (list of lists of integers). In Definition 2, we define the values $a_i.w$ and $a_i.z$ mentioned earlier for integers found in the lists of A^k.

Definition 2. *Define $a_i.z = h$ when $a_i \in A^k_h$, for $h \in \{0, \ldots, x\}$. Let $a_s, a_t \in A^k_h$ and assume they satisfy* ExploreV$(a_s.v, u, m)$ =ExploreV$(a_t.v, u, m)$ *where $u \in \{-1, \ldots, (k-1)\}$. Then, we define $a_s.w =$* ExploreV$(a_s.v, k-1, m) =$ ExploreV$(a_t.v, k-1, m) = a_t.w$.

5.4.1 Creating A^{k+1} from A^k

We assign $a_i.z = 0$ and $a_i.w = -1$ for all $i \in \{0, \ldots, (n-1)\}$ in A^0. Assume $A^k = [A^k_0, A^k_1, \ldots A^k_x]$, for $0 \leq k$ and $0 \leq x$, is constructed. We describe how we can create A^{k+1} from A^k step by step below.

1. Form the array L from A^k in the same order as they appeared in A^k. For example, if $A^k = [[a_2, a_0][a_1, a_3]]$ then $L = [a_2, a_0, a_1, a_3]$.

2. Form the list L' by sorting the integers in L, by any stable sort algorithm, based on their $\mathsf{ExploreV}(L[r].v, k, m)$ values in descending order for $0 \leq r < n$. We propose that we can use the counting sort algorithm for this purpose.

3. To obtain A^{k+1} from L', we use Algorithm 5. See Proposition 9 for the correctness of this algorithm.

Algorithm 5: $\mathsf{Create}(L', n, k, m)$

Input: L', an array of n integers of m bits. This list is sorted in descending order based on $\mathsf{ExploreV}(L'[r].v, k, m)$ for $0 \leq r < n$.
Output: A^{k+1}, list of lists of integers.

```
1  initialize A^{k+1} as an empty list;
2  s = 0;
3  d = 0;
4  while s < n do
5      for r = s; r < n; r = r + 1 do
6          if L'[s].z == L'[r].z ∧ ExploreV(L'[s].v, k, m) == ExploreV(L'[r].v, k, m)
           then
7              L'[r].w = ExploreV(L'[r].v, k, m);
8          else
9              insert [L'[s], ··· , L'[r − 1]] as the d-th list of A^{k+1};
10             d = d + 1;
11             Break;
12     s = r;
   /* Observe for any two integers a_i and a_j, we have a_i.z = a_j.z, if
      both of the integers are from the same list of A^{k+1}.          */
13 sort the lists in A^{k+1} based on a_i.z values by ascending order using any stable
   sort algorithm;
14 update the a_i.z values of all integers in all lists of A^{k+1};
15 return A^{k+1};
```

Proposition 9. *Algorithm 5 works correctly.*

PROOF ▷Consider any two integers a_i and a_j, for $0 \leq i, j < n$, such that they are in a list A^k_s of A^k, for $0 \leq s$ and $0 \leq k$. Assume these two integers are in a list A^{k+1}_t for $0 \leq t$. If $\mathsf{ExploreV}(a_i.v, k-1, m) = \mathsf{ExploreV}(a_j.v, k-1, m) = a_i.w = a_j.w$ (before the execution of Algorithm 5) then $\mathsf{ExploreV}(a_i.v, k, m) = \mathsf{ExploreV}(a_j.v, k, m) = a_i.w = a_j.w$ (after the execution of Algorithm 5) because of the execution of line 7 of Algorithm 5. Again if $a_i.z = a_j.z = s$ in (before the execution of Algorithm 5) then $a_i.z = a_j.z = t$ (after the execution of Algorithm 5) because of the execution of line 14 of Algorithm 5. □

In Proposition 10, we describe the relationship between the order of the lists in A^k and the sorted order of U.

Proposition 10. *Assume $A^k = [A^k_0, A^k_1, \ldots A^k_x]$, for $0 \leq k$ and $0 \leq x$. Let A^k_s and A^k_t be two lists of A^k such that $0 \leq s < t \leq x$ then all integers in A^k_s are smaller than any integer in A^k_t.*

PROOF ▷ We proof it by induction.

(Base case): This is true in A^0. Because we have one list in A^0 which is U.

(Inductive step): Assume it is true in $A^q = [A^q_0, A^q_1, \ldots A^q_y]$, for $0 \leq q$ and $0 \leq y$. Consider any two lists A^q_s and A^q_t for $0 \leq s < t \leq y$. Thus all integers from A^q_s are smaller than any integer in A^q_t. Let $[A^{q+1}_{s1}, \ldots, A^{q+1}_{s1+b}]$ and $[A^{q+1}_{t1}, \ldots, A^{q+1}_{t1+c}]$ be two sub-lists of A^{q+1} obtained by partitioning A^q_s and A^q_t respectively for some positive integers $s1, t1, b, c$. Thus all integers from sub-list $[A^{q+1}_{s1}, \ldots, A^{q+1}_{s1+b}]$ are smaller than any integer in $[A^{q+1}_{t1}, \ldots, A^{q+1}_{t1+c}]$.

Line 14 of Algorithm 5 ensures that lists in each of such sub-list are placed consecutively in A^{q+1}. It also ensures that $[A^{q+1}_{s1}, \ldots, A^{q+1}_{s1+b}]$ comes before $[A^{q+1}_{t1}, \ldots, A^{q+1}_{t1+c}]$ in A^{q+1}.

Again for any such sub-list, for example $[A^{q+1}_{t1}, \ldots, A^{q+1}_{t1+c}]$, the order by which each of the list is created and inserted into A^{q+1} in the while-loop of Algorithm 5 depends on the q-th 1s of their integers. Thus all integers in any list are smaller than any integer of the next list in the same sub-list.

So Proposition 10 is also true in A^{q+1}. □

In Proposition 11, we compute the time complexity of creating A^{k+1} from A^k.

Proposition 11. *We can create A^{k+1} from A^k by $O(n+(k+1)p)$ bit operations using Algorithm 5.*

PROOF ▷ We can create the list of integers L from A^k in $O(n)$ time as described in Section 5.4.1. The values in $\mathsf{ExploreV}(a_i.v, k, m)$ for $0 \leq i < n$ are expected to be in the range $[0, (k+1)p]$. We can create L' from L using the counting sort algorithm in $O(n + (k+1)p)$ time. The while-loop in Algorithm 5 runs in $O(n)$ time. We propose that we can use the counting sort algorithm in line 14 of this algorithm. The counting sort algorithm can sort the lists in A^{k+1} in $O(n)$ time. Finally, we can updates z values of all integers in $O(n)$ time. □

Our proposed sorting algorithm is described in Proposition 12.

Proposition 12. *Our proposed sorting algorithm creates list of lists $A^0, A^1, \ldots$ successively. It terminates whenever one of the two following conditions holds.*

1. *Every list of the list A^ℓ, for some non-negative integer ℓ, has a single element.*
2. *$a_i.w = m$ for all integers in A^ℓ.*

Let L be a list of integers obtained from A^ℓ. The order of the integers in L is the same as they appeared in A^ℓ. Then the integers in U can be found sorted in ascending order in L.

PROOF ▷ We proof it for two different cases below.

(Case 1, when every list in A^ℓ has single integer): It is easy to follow from Proposition 10.

(Case 2, when $a_i.w = m$ for all $i = 0 \ldots, n-1$): Observe in this case, any further calls to Algorithm 5 returns the same list of lists. All integers in a list of A^ℓ are same. □

In the case where our input U has duplicate values, according to Proposition 12, our proposed algorithm terminates after calling Algorithm 5 for τ times, where τ is the maximum number of 1s in any integer. As a result, our implementation becomes inefficient. To resolve this performance issue, we limit the number of times we need to call Algorithm 5. This is presented in Proposition 15. Proposition 16 describes how we can get the sorted list in this case.

Proposition 13. *Our proposed algorithm is a stable sorting algorithm.*

PROOF ▷ Observe that, in creating L as defined in Section 5.4.1, we preserve the given order (U) for the duplicate integers. Moreover we apply stable sorting algorithm in creating L' and ordering the lists in A^{k+1} in Algorithm 5. For both cases, we preserve the given order for duplicate integers too. □

5.5 Complexity

In Remark 1, we describe one important observation that helps us to compute the memory requirement for our proposed sorting algorithm given in Proposition 14.

Remark 1. *Our proposed algorithm creates the list of lists of integers successively. Once it creates A^{k+1} from A^k, it does not need to store A^k anymore.*

Proposition 14. *We need $O(n + nm/p)$ words of storage for our proposed sorting algorithm.*

PROOF ▷ The expected number of words required to store $a_i.v$, for $i \in \{0, \ldots, n-1\}$, is $O(m/p)$. Accordingly, we can store a list of lists of integers in $O(n + nm/p)$ words of storage. □

Proposition 15 describes the number of times ℓ, we need to call Algorithm 5 to be sure that all lists in A^ℓ have $O(1)$ integers. In Proposition 16, we describe the cost to order each list in A^ℓ to find the sorted order of the integers.

Proposition 15. *It is expected that each list of A^ℓ has $O(1)$ integers where $\ell = O(\log_p(n))$.*

PROOF ▷ It is expected that any list A^k_h in A^k has at most $O(\frac{n}{p^k})$ integers. So after calling Algorithm 5 $O(\log_p(n))$ times, we expect that all lists in the list of lists of integers have $O(1)$ integers. □

Proposition 16. *Let A^ℓ, be a list of lists where each of the list has $O(1)$ integers as described in Proposition 15. We need $O(1)$ comparison operations to order each list in A^ℓ. Thus, we need $O(n\frac{p^2}{2(p-1)})$ and $O(n\frac{p}{2(p-1)})$ bit operations to sort n non-negative dense and sparse integers respectively.*

PROOF ▷ It follows from Corollary 1 and Corollary 2. □

Proposition 17. *The time complexity of our proposed sorting algorithm to sort n non-negative dense integers is $O(n \log n + \frac{p \log n(\log n+1)}{2} + n\frac{p^2}{2(p-1)})$.*

PROOF ▷ Following from Proposition 11, the number of bit operations required for computing $A^{\log_p(n)}$ is

$$O\left(\sum_{j=1}^{\log_p(n)} (n + jp)\right).$$

For dense integers, where p is small, we can say $\log_p(n) \approx \log n$. Finally, from Proposition 16, we can compute the cost to sort the integers in all lists of $A^{\log_p(n)}$.

It should be noted that, we do not consider the cost for creating $[a_0.v, \ldots, a_{n-1}.v]$, which is required for dense integers. □

Our proposed algorithm may not be suitable for dense integers for two main reasons. First, we need to compute $[a_0.v, \ldots, a_{n-1}.v]$ in advance, which is expensive. Second, the cost of comparison based sorting algorithm for dense integers given in Corollary 1 might already be good enough for practical purposes. Still, our proposed algorithm has some properties given below which might be useful in practice.

- Our sorting algorithm is stable.
- We can call Algorithm 5 for a number of times to make each list of integers small. Then each of the list can be sorted independently (or in parallel) by any comparison based sorting algorithm. Thus, Algorithm 5 can be used as a preprocessing step of the sorting algorithm.

Proposition 18. *The time complexity of our proposed sorting algorithm to sort n non-negative sparse integers is $O(n + p + n\frac{p}{2(p-1)})$, where $\log_p(n) = O(1)$.*

Proof ▷ It follows from Propositions 11 and 16. □

5.6 Conclusion

Based on theoretical complexity, we can say our algorithm is more suitable for sorting large sparse integers. It can be modified to sort other type of objects as well. For example, in Chapter 6 we apply this algorithm to sort binary reflected Gray codes. Moreover it can be used as a preprocessing step in sorting of dense integers. In our proposed algorithm, we suggest applying the counting sort algorithm as a stable sorting algorithm for intermediate sorting. Cache-oblivious counting sort algorithm of Chapter 4 can be used for this purpose.

Chapter 6

Cache Friendly Sparse Matrix-vector Multiplication

This work is motivated by the challenges posed, in terms of data locality, by large and unstructured matrices occurring in sparse linear algebra. Our goal is to minimize the cache complexity of sparse matrix-vector multiplication. In a previous work, we experimentally observed that, for an input matrix S, column reordering based on binary reflected Gray code was a practically efficient preprocessing phase, which could be amortized against repeated multiplications of S by a dense vector [29].

In this chapter, we provide a theoretical foundation for the above observation. If S counts n columns, m rows and has a total number τ of non-zero entries and if S is sufficiently sparse, we show that the columns and rows of S can be reordered in $O(\tau)$ bit operations, using the RAM model with memory holding a finite number of w-bit words, for a fixed w. This reordering of columns and rows is inspired by binary reflected Gray code. We establish a cache complexity result for sparse matrix-vector multiplication when the sparse matrix is reordered by our proposed method.

We report numerical experiments which confirm our theoretical results. In particular, we include data for a simulation of the ideal cache model for verifying our cache complexity estimates.

This chapter is a joint work with S. Hossain and M. Moreno Maza.

6.1 Introduction

Sparse matrix-vector multiplication, or SpMxV, is an important kernel in scientific computing. For example, the conjugate gradient method is an iterative linear system solving process where multiplication of the coefficient matrix S with a dense vector x

is the main computational step accounting for as much as 90% of the overall running time. While the total number of arithmetic operations (involving non-zero entries only) to compute Sx is fixed, reducing the probability of cache misses per operation by preprocessing S remains a challenging area of research. This preprocessing is done once and its cost is amortized by repeated multiplications. Computers that employ cache memory to improve the speed of data access rely on the reuse of data that is brought into the cache memory. The challenge is to exploit data locality especially for unstructured problems like modeling data locality, which in this context is hard [68].

Pinar and Heath [59] propose column reordering to make the non-zero entries in each row contiguous. However, column reordering for arranging the non-zero entries in contiguous location is NP-hard [59]. In a considerable volume of work [38, 29, 59, 69, 71] on the performance of SpMxV on modern processors, researchers propose optimization techniques such as the reordering of the columns or rows of S to reduce indirect access and improve data locality, and blocking to reduce memory load and loop overhead. In [37], the authors describe a number of applications of sparse matrix-vector multiplication.

Here, we present a new row-and-column permutation algorithm, based on binary reflected Gray codes, that runs in *linear time* with respect to the number of non-zero entries.

To evaluate these results, we have realized an implementation of our algorithm and analyzed its performance on a set of well-known test matrices. Our experimental results are coherent with our theoretical estimates and demonstrate the performance gains rendered by our permutation algorithm.

The organization of this chapter is as follows. In Section 6.2, we discuss some preliminary materials followed by our proposed re-ordering algorithm in Section 6.3. We analyze our preprocessing algorithm in Section 6.4 and present the experimental results in Section 6.5.

6.2 Background

In this section, we review some of the data structures and introduce some notations used in this chapter.

6.2.1 Compressed row storage scheme (CRS)

Storage schemes used for unstructured sparse matrices usually involve some form of indirect indexing of its non-zero elements via auxiliary data structures. For example, the compressed row storage (CRS) scheme [4] uses two auxiliary arrays, `colind` of length τ (the number of non-zero elements) and `rowptr` of length $m+1$ where m is the number of rows of S. This is the most common storage scheme for sparse matrices. The three arrays required to store the sparse matrix S are described below.

1. `value`: for storing the non-zeros of S row-by-row,
2. `colind`: for storing the column index of each non-zero, and
3. `rowptr`: for storing the index of the first non-zero of each row in the `value` array.

6.2.2 SpMxV with CRS scheme

Sample code for computing $y = Sx$ under CRS scheme is given in Algorithm 6. In this algorithm, accesses to vector y and all three arrays of CRS are regular. But the accesses to the vector x might be irregular because the column indices of each row may not be consecutive. A large number of cache misses might occur during the accessing of x which may make the SpMxV very slow in practice.

6.2.3 Compressed column storage scheme (CCS)

This scheme is the same as CRS except that the non-zeros are stored column-by-column. Like CRS, three arrays are used in the compressed column storage scheme (CCS) to store sparse matrix S are described below.

1. `value`: for storing the non-zeros of S column-by-column,
2. `rowind`: for storing the row index of each non-zero, and
3. `colptr`: for storing the index of the first non-zero of each column in the `value` array.

6.2.4 Notations

We consider a sparse matrix S with arbitrary sparsity structure having m rows $(r_0, \ldots, r_{m-1})$, n columns $(c_0, \ldots, c_{n-1})$ and τ non-zero elements. Here, $s_{i,j}$ refers

to the entry of S which is at the i-th row and the j-th column. We denote by ρ^r and ρ^c the average number of non-zeros in a row and a column respectively. Let $1/p$ be the probability that an element of S is non-zero. Throughout this chapter, we assume that m and n are positive integers of machine word size, or smaller. We assume that $\tau > m + n$ and $\min(m, n) > p$. Time and space complexity estimates are given for the RAM model with memory holding a finite number of w-bit words, for a fixed w [64]. Cache complexity is measured by considering the ideal cache model described in Chapter 2.

Algorithm 6: SpMxV(`value`, `colind`, `rowptr`, x)

Input: `value`, `colind`, `rowptr` are three arrays that represents S in CRS and dense vector x

Output: vector y, where $y = Sx$

```
1 for all i = 0, 1, ..., m − 1 do
2     y[i] = 0;
3 for i = 0, 1, ..., m − 1 do
4     for k = rowptr[i] to rowptr[i + 1] − 1 do
5         j = colind[k];
6         y[i]+ = value[k] ∗ x[j];
7 return y;
```

6.2.5 Binary reflected Gray code

A q-bit binary reflected Gray code [43] is a Gray code denoted by G^q and defined by $G^1 = [0, 1]$ and

$$G^q = [0G_0^{q-1}, \ldots, 0G_{2^{q-1}-1}^{q-1}, 1G_{2^{q-1}-1}^{q-1}, \ldots, 1G_0^{q-1}], \text{ for } q > 1,$$

where G_i^q is the i-th binary string of G^q and $0 \leq i < 2^q$. We call i the rank of G_i^q in G^q. For example, $G^2 = [00, 01, 11, 10]$ and $G^3 = [000, 001, 011, 010, 110, 111, 101, 100]$. So, the rank of 011 in G^3 is 2. For details please see [43].

6.2.6 Sorting of binary reflected Gray codes

In this chapter, we develop a new row and column permuting algorithm based on *binary reflected Gray code* for sparse matrices. We call it BRGC *ordering*. For our proposed reordering algorithm, we consider each non-zero of S as 1. We also consider each column of S as a binary reflected Gray code in G^m. Like in Section 5.2, we consider the bits from row 0 and $m-1$ as the most and least significant bits respectively.

In this section, we explain how we can sort binary reflected Gray codes in descending order of their ranks by our proposed sorting algorithm described in Chapter 5. From the mathematical definition of binary reflected Gray code in Section 6.2.5, we can describe Corollary 3 which is the basis for sorting binary reflected Gray codes.

Corollary 3. *Let G_i^q and G_j^q be two different binary reflected Gray codes in G^q. Let their* first disagree bit *(see Section 5.2) be h for $0 \leq h < q$. Assume that the h-th bit of G_j^q has* 1*. If the number of* 1*s in G_i^q or G_j^q before h-th bit is even (odd), we can conclude $j > i$ $(i > j)$.*

Proposition 19 describes how we can modify our proposed sorting algorithm in Chapter 5 to sort binary reflected Gray codes according to their ranks.

Proposition 19. *Our proposed sorting algorithm in Chapter 5 can sort binary reflected Gray codes in descending order according to their ranks with one modification. While creating A^{k+1} from A^k (see Section 5.4.1), we need to form array L and apply a stable sort algorithm on L to obtain L' in ascending order only when k is even.*

Proof ▷ It follows from Corollary 3. □

It should be noted that, we do not use any well-established sorting algorithm, like quick sort, for this purpose. We can implement quick sort algorithm available in C++ STL. The reasons for not using these technique are given below.

1. Our reordering algorithm, which is described later of this chapter, is not just a sorting of columns considering their ranks in binary reflected Gray codes.

2. We have already seen in Chapter 5, our proposed sorting algorithm is suitable for sparse objects.

6.3 Proposed reordering method

Our reordering algorithm proceeds by several consecutive intermediate reorderings, all of which permute the columns except the second one which permutes the rows. Algorithm 7 describes our proposed reordering algorithm. We explain this algorithm step by step with their time and memory complexity. Below we describe Lemmas 2, 3, 4, and 5, which help us understand the time complexity of our proposed ordering algorithm.

Lemma 2. *Let a sparse matrix S be given in* CRS *(*CCS*). The total number of bit operations required to compute the* CCS *(*CRS*) representation of S from its* CRS *(*CCS*) representation, is $O(\tau)$, where τ is the total number of non-zeros in S.*

PROOF ▷ We show how to convert from CRS to CCS scheme of S by applying $O(\tau)$ bit operations. The reverse conversion follows a similar method. We can determine the number of non-zeros in each column from CRS. It involves traversing `colind` which requires $O(\tau)$ bit operations. We store this information in an array F of length n, where $F[i]$ is the number of non-zeros of column c_i, for $i \in \{0, \ldots, n-1\}$. We can create another array D of length n, where $D[i]$, stores the total number of non-zeros in columns $\{c_0, \cdots, c_{i-1}\}$. It is possible to create D from F by applying $O(n)$ bit operations. Observe that, we can create `colptr` of CCS from D. We traverse S again using `colind` and `value` of CRS. During this traversal, for each non-zero $s_{j,i}$, we can compute how many non-zeros we have in S for columns $\{c_0, \ldots, c_{i-1}\}$ from D, for $j \in \{0, \ldots, m-1\}$. We can also keep track of the number of non-zeros that we have seen so far for each column during this traversal using another temporary array. These two pieces of information help us identify the indices of non-zeros and place into `value` and `rowind` of CCS. It also requires $O(\tau)$ bit operations. □

Lemma 3. *When we convert* CRS *(*CCS*) of S to* CCS *(*CRS*) by the method stated in the proof of Proposition 2, the row (column) indices for each column (row) in* CCS *(*CRS*) are sorted in ascending order.*

PROOF ▷ While we are scanning the non-zeros of S for the second time, for a column (row), the order by which the non-zeros appear depends on their row (column) indices. □

Lemma 4. *Let a sparse matrix S be given in* CRS *(*CCS*). Suppose we want to permute the columns (rows) of S by a given column (row) permutation. In order to update CRS (CCS) according to the new column (row) permutation, we need to perform $O(\tau)$ operations.*

PROOF ▷ We need to traverse `colind` (`rowind`) and change each column (row) index according to the new column permutation. □

Lemma 5. *Let a sparse matrix S be given in CRS (CCS). Suppose we want to permute the rows (columns) of S by a given row (column) permutation. In order to update CRS (CCS) according to the new row (column) permutation, we need to perform $O(\tau)$ operations.*

PROOF ▷ It follows from Lemma 2 and 4. □

Our reordering strategy is stated in Algorithm 7. It is important to understand that in this preprocessing step each non-zero entry is treated as 1. So, in Algorithm 7,

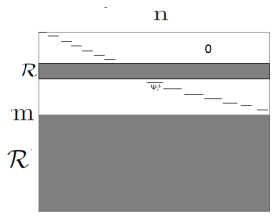

Figure 6.1: After initial column ordering.

we consider each of the columns of S as a binary string of G^m. We apply Algorithm 5 in Chapter 5 for our proposed reordering strategy. For each column c_i, for $i \in \{0, \ldots n-1\}$, we can easily create $c_i.v$, indices of 1s (see Section 5.2), from the CCS representation of S. Algorithm 7 order the rows and columns of S according to BRGC ordering. We describe each step of Algorithm 7 below.

6.3.1 Initial column ordering

Lines 1-3 of Algorithm 7 are responsible for doing the initial permutation of the columns of S. First, we create A^0 considering each column of S as an m-bit binary string following Section 5.4.1. We need to apply $O(\tau)$ bit operations to get A^0. Finally, we can compute A^1 from A^0 using Algorithm 5, with the modifications stated in Proposition 19. It requires $O(n+p)$ bit operations. The initial column ordering is the order of the columns as they appeared in A^1. We need to update both CRS and CCS data structures of S according to this initial column ordering. We also need to be sure that the column (row) indices of non-zero entries for each row (column) in CRS (CCS) are sorted in ascending order. According to Lemma 3, we can perform these data structure updates by applying $O(\tau)$ bit operations. Each of the data structures requires $O(\tau)$ words to be stored. The time complexity of this step is $O(\tau)$, as we assume that $n > p$. The space complexity of this step is also $O(\tau)$.

In Figure 6.1, we have shown what the sparse matrix looks like after this step.

6.3.2 Row ordering

After the initial column ordering, we perform a row permutation, which is stated by Algorithm 8. The intention of this row permutation is to place some more non-zeros for which we can build some regular patterns. These patterns may reduce the cache misses during SpMxV. Algorithm 8 describes this row permutation procedure. We

Algorithm 7: BRGC(CRS(S), CCS(S), m, n, b, t)

Input: CRS(S) and CCS(S) are the CRS, CCS representation of sparse matrix S that has m rows and n columns, b is a positive integer, where $b > 0$ and t is a positive number where $0 < t < 1$.

Output: CRS(S), the CRS representation of sparse matrix S after reordering of rows and columns.

1 compute A^0 considering each column of S as a binary object;
2 compute A^1 from A^0 using Algorithm 5 with the modifications stated in Proposition 19;
3 update CRS(S) and CCS(S) according to the column permutation found in A^1 and we need to be sure that the column (row) indices for each row (column) in CRS (CCS) are sorted in ascending order;
4 Γ = RowOrdering(A^1, CRS(S), CCS(S), m);
5 update $c_i.v$ (for $i = 0, \cdots, (n-1)$, where c_i is the i-th column of S after initial column permutation), CRS(S) and CCS(S) according to the row permutation found in Γ;
6 merge(A^1, b, m);
7 $k = 1$;
8 **while** true **do**
9 break if the number of lists in A^k is greater than tn;
10 create A^{k+1} from A^k using Algorithm 5 with the modifications stated in Proposition 19;
11 $k = k + 1$;
12 update CRS(S) according to the column permutation found in A^k;
13 return CRS(S);

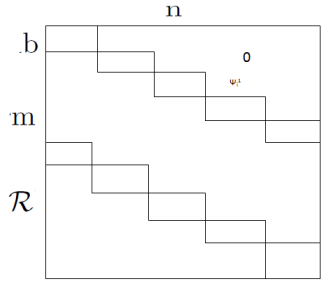

Figure 6.2: After row permutation.

stress the fact that not all rows participate to this row permutation. To be more precise, we need the following notion.

The function SelectedRows(A^1, m) returns precisely the set of rows of S that participate (set R) and do not participate (set $\mathcal{R}$) in creating A^1. By participation, we mean, a row participates in A^1 if and only if at least one non-zero of that row participates in the stable sorting routine which is required to create L' from L (see Section 5.4.1). In the sequel, we denote this set by $\mathcal{R}$. Thus both R and $\mathcal{R}$ are subsets of $\{r_0, \ldots, r_{m-1}\}$. We define R as a set of rows defined as $\{r_0, \ldots, r_{m-1}\} - \mathcal{R}$. The time complexity of SelectedRows(A^1, m) is $O(m)$. We have shown $\mathcal{R}$ rows in Figure 6.1.

Algorithm 8 calls Algorithm 9 and returns the row permutation Γ. Algorithm 9 first determines an ordering for the rows that are in R and then computes an ordering for the rows that are in $\mathcal{R}$. The former step is done in a straightforward manner through Lines 1 to 4 of Algorithm 9. The latter step is detailed below.

We first need to understand the following definitions.

Definition 3. *Let $R_i(j)$ be the column index of the j-th non-zero element in the i-th row of A for $0 \leq i < n$ and $0 \leq j < n$. Getting $R_i(j)$ for a single non-zero from the* CRS *representation of a sparse matrix is expensive. However, we can identify the column indices of all non-zeros of S one-by-one from the* CRS *representation, which requires $O(\tau)$ bit operations.*

Definition 4. *We say that a list A^1_h of A^1 is the* owner *of the non-zero referred by $R_i(j)$ if the non-zero belongs to a column in A^1_h. Furthermore, we denote A^1_h as the* winner *of the i-th row if no other list in A^1 owns more nonzeros from the i-th row than A^1_h. In Algorithm 9, we call a routine called* owner$(R_i(j), A^1)$*, which returns the* owner *of the nonzero referred by $R_i(j)$.*

Now we are ready to describe lines 5 to 21 of Algorithm 9. At Line 5, we initialize an array of queues called `winlist`, whose size is equal to the number of lists in A^1. As all column indices of non-zero entries for a row in CRS are in ascending order, after the execution of the for-loop (between line 6 and line 21), the queue `winlist`$[k]$ contains all the indices of the rows won by the list A^1_k, where k is a positive integer. Lines 22 to 25 determine the ordering of the rows from $\mathcal{R}$ as follows. Let i and j be the indices of two distinct rows in $\mathcal{R}$, that are won by two different lists A^1_{s1} and A^1_{s2}, respectively, where $0 \leq \mathsf{s1} < \mathsf{s2}$. Then in Γ, i appears before j. Rows that are won by the same list can be placed in arbitrary order in Γ, though in our algorithm we dedicate k queues for this purpose. The time complexity of Algorithm 9 is $O(\tau)$.

Each of the data structures used in row ordering requires $O(\tau)$ words to be stored. Figure 6.2 shows how a sparse matrix looks like after row permutations.

Algorithm 8: RowOrdering(A^1, CRS(S),CCS(S), m)

Input: A^1, the list of lists of columns of S described in Definition 2, CRS(S) and CCS(S) are the CRS and CCS representation of sparse matrix S respectively and m is the row dimension of S

Output: Γ, the row permutation vector

1 $[\mathcal{R}, R] = \text{SelectedRows}(A^1, m)$;
2 return $\text{RowPerm}(A^1, R, \mathcal{R}, \text{CRS}(S))$;

6.3.3 Algorithm merge(A^1, b, m)

Let b be an integer, where $b > 0$. In this routine, we re-create A^1. In our new A^1, a list A^1_i contains all columns for which the following property satisfies $\{\lfloor \text{Explore}(c_q.v, 0, m)/b \rfloor = i | q \in \{0, \ldots, n-1\}\}$. The objective of this redefinition is to make the cardinality of each list of A^1 bigger. Once we re-create A^1, we need to update $[c_0.v, \ldots, c_{n-1}.v]$. Let c_q be a column which is in A^1_i.

- First, delete all entries of $c_q.v$ that are smaller than $(i+1)b$.
- Second, insert i as the first entry of $c_q.v$.

We need $O(\tau)$ bit operations to complete this algorithm. Because re-creation of A^1 can be done by $O(n)$ bit operations but the modifications of the list of indices $[c_0.v, \ldots, c_{n-1}.v]$ requires $O(\tau)$ bit operations. All data structures required for this algorithm have memory complexity of $O(\tau)$.

6.3.4 Iterative column ordering

In this step, we need to create A^k for $k = 2, \cdots$, until the number of lists in the list is greater than tn for $0 < t < 1$. We fix it as 0.9. Considering the sparse matrices found in the *University of Florida sparse matrix collection*[1], we can say $\log_p(n) = O(1)$. Thus according to Proposition 15, we need few iterations to break the while-loop of Algorithm 7. We can create A^{k+1} from A^k using Algorithm 5 with the modifications stated in Proposition 19 by applying $O(n+p)$ bit operations. So the time complexity of this step is expected to be $O(n+p)$, and the space complexity of this step is $O(\tau)$.

[1] `http://www.cise.ufl.edu/research/sparse/`

Algorithm 9: RowPerm($A^1, R, \mathcal{R}$, CRS(S))

Input: A^1, the list of lists described in Definition 2, R and $\mathcal{R}$ are the sets of rows of S described in Section 6.3.2 and CRS(S) is the CRS representation of S

Output: Γ, the new row permutation

```
1  j = 0;
2  for all row r_i in R do
3      Γ[j] = i ;
4      j = j + 1;
5  initialize winlist as howManySets(A^1) empty queues;
   /* Let howManySets(A^1) be a routine that returns the number of
      lists in A^1.                                                  */
6  for all row r_i in R that has at least one non-zero do
7      q = 1;
8      k = 1;
9      k_max = 1;
10     l_max = owner(R_i(0), A^1);
11     l = l_max;
       /* Let length(r_i) be the number of non-zeros in i-th row of S.
          */
12     while q < length(r_i) do
13         while l == owner(R_i(q), A^1) and q < length(r_i) do
14             k = k + 1;
15             q = q + 1;
16         if k_max < k then
17             k_max = k;
18             l_max = l ;
19         k = 1;
20         l = owner(R_i(q), A^1);
21     Push i into the winlist[l_max] ;
22 for all k = 0 ··· (howManySets(A^1) − 1) do
23     while queue winlist[k] is not empty do
24         Γ[j] = pop(winlist[k]) ;
25         j = j + 1;
26 The rows in R that does not have any non-zero are added in Γ at the end in
   arbitrary order.
27 return Γ;
```

6.4 Complexity

In this section, we discuss the time and memory complexity of Algorithm 7. Furthermore, we discuss the cache complexity of Algorithm 6 when S is preprocessed BRGC ordering.

6.4.1 Time complexity

Lemma 6. *Let S be a sparse matrix with m rows and n columns that has τ non-zeros. Let $1/p$ be the probability that an entry of S is non-zero, where $\log_p(n) = O(1)$. The time complexity of Algorithm 7 is $O(\tau)$ with $b > 0$ and $0 < t < 1$.*

PROOF ▷ The time complexity of each step of Algorithm 7 such as initial column ordering (Section 6.3.1), row ordering (Section 6.3.2) and $\mathsf{merge}(A^1, b, m)$ (Section 6.3.3) is $O(\tau)$. The last step called iterative column ordering requires $O(n+p)$ bit operations (Section 6.3.4). Since, $n > p$, the conclusion follows. □

6.4.2 Memory complexity

Lemma 7. *The memory complexity of Algorithm 7 is $O(\tau)$.*

PROOF ▷It follows from the fact that each data structure used for our preprocessing step requires $O(\tau)$ words of storage. □

6.4.3 Cache complexity

We first describe different types of non-zeros in S after BRGC ordering. We need this classification to explain the cache complexity of SpMxV.

Classification of non-zeros

We classify the non-zeros of S into five categories after applying the BRGC algorithm. These are α, β, γ, δ and λ non-zeros.

- α **non-zeros:** In $\mathsf{merge}(A^1, b, m)$, we delete some entries from the list of indices $[c_0.v, \cdots, c_{n-1}.v]$. α non-zeros include the non-zeros referred by those deleted entries.

  Remark 2. *In general, there exists $O(n)$ number of α non-zeros in S.*

- β **non-zeros:** It includes the non-zeros that participate in creating A^2 from A^1.

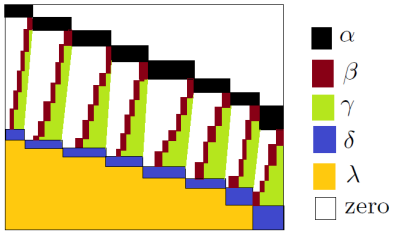

Figure 6.3: The distribution of different types of non-zeros.

Remark 3. *In general, there exists $O(n)$ number of β non-zeros in S.*

From Table 6.1, we found that most of the test matrices have $0.99n$ β non-zeros and all of them have more than n α non-zeros.

δ non-zeros: Every row is won by a list in A^1 during the row permutation. Assume row r_i, for $i \in \{0, \ldots, m-1\}$, is won by A^1_h of A^1. Each non-zero of r_i that is owned by the list A^k_h is called a non-zero of δ type. The total number of δ non-zeros is $O(\mathcal{R})$. In practice, for all matrices considered for experimentations, we found the number of δ non-zeros is $O(n)$.

Remark 4. *A non-zero can be both β and δ non-zero.*

γ non-zeros: A non-zero whose row index is greater than that of the β non-zero of the corresponding column but smaller than that of any δ non-zero of the same column is referred to a γ non-zero.

λ non-zeros: Each of the remaining non-zero is denoted as a λ non-zero.

In Figure 6.3, we can view the different types of non-zeros of a sparse matrix pictorially. After applying Algorithm 7 on S, we obtain some interesting sparsity structures in S. We write a C code to simulate cache misses on the ideal cache model during SpMxV. The number of cache misses for test matrices are shown in Table 6.2.

Cache misses and different types of non-zeros

We analyze the sparsity of a sparse matrix after preprocessing by our proposed algorithm. In conclusion, we come up with the following Proposition 20 about the cache misses during SpMxV.

Proposition 20. *Suppose S has at least $3n$ non-zeros and S is preprocessed by our proposed* BRGC *ordering algorithm (Algorithm 7). For the ideal cache with $Z \geq$*

$2\sqrt{nL}$ *and* $b = \sqrt{nL}/\rho$*, where* $\rho = \rho^r$*, we observe* $3n/L + O(\sqrt{n/L}) + 1$ *cache misses due to accessing* x *during* SpMxV *considering only* α*,* β *and* δ *non-zeros of* S*.*

PROOF ▷Algorithm 7 returns S in CRS after reordering both the rows and columns. Then we apply Algorithm 6 for SpMxV. After applying Algorithm merge(A^1, b, m), the number of elements in each list of A^1 is expected to be ρb. The expected number of lists in A^1 is $n/(\rho b)$.

If we have only α non-zeros in S, let α^{s1} and α^{s2} be two subsets of α non-zeros such that α^{s1} and α^{s2} are the non-zeros from some elements of A^1_{s1} and A^1_{s2} respectively. Assume s1 $<$ s2. It is easy to observe that α^{s1} non-zeros participate in SpMxV before α^{s2}. It is also obvious that the indices of elements x that are multiplied with α^{s1} are smaller than those that are multiplied with α^{s2}. Considering the ideal cache model, we can multiply all α non-zeros with corresponding elements of x by scanning x once. This refers to access a number of consecutive arrays in a irregular fashion one after another, where each array is expected to have ρb entries. The expected number of such arrays is $n/(\rho b)$. This creates $n/L + O(n/\rho b)$ cache misses, if the ideal cache can store ρb non-zeros.

Now, suppose we multiply β non-zeros in S with x only. Let β^h be the non-zeros from a list A^1_h. It is sufficient to have one cache line for these non-zeros during SpMxV in order to avoid conflict or capacity misses on the ideal cache model. Because if we list the column indices of β^h row by row, it becomes a list of integers in descending order. SpMxV for β non-zeros and x refers to accesses (in a regular fashion) a number of consecutive arrays, of possibly different sizes, but with total size n. The expected number of such arrays is $n/(\rho b)$. We can say that the cache complexity to multiply β non-zeros with x is $n/L + O(n/(\rho b))$ due to accessing x on the ideal cache model if the ideal cache has $n/(\rho b)$ cache lines.

Again, suppose we only multiply δ non-zeros of S with x. If we list the column indices of δ non-zeros row by row, it becomes a list of integers in ascending order. So, in SpMxV, we have to scan x once in a regular fashion to multiply β non-zeros with x. It creates $n/L + 1$ cache misses in accessing x on the ideal cache model. It is expected that, the row indices of δ non-zeros are bigger than any β non-zeros.

Finally, considering only α, β and δ non-zeros in S, during SpMxV, one can observe $3n/L + O(n/(\rho b)) + 2$ cache misses in accessing x if we have spaces for storing $\rho b + nL/(\rho b)$ non-zeros of x in the ideal cache model. So the constraint is $Z \geq \rho(b + nL/(\rho b))$. Replacing b by $\sqrt{nL}/\rho$ in the inequality, we obtain $Z \geq (2\sqrt{nL})$. □

Remark 5. *We are not able to state a conclusion on the cache complexity while*

multiplying γ and λ non-zeros with x. We believe some of these non-zeros create some patterns to improve data locality.

Remark 6. *The value of b given in Proposition 20 is maximal for a given Z. To see why, let $\rho b + (nL)/(\rho b) = Z$. This equality can be expressed as a quadratic equation in (ρb) as given below*

$$(\rho b)^2 + nL - Z\rho b = 0. \tag{6.1}$$

The discriminant of Equation 6.1 is $Z^2 - 4nL$. It should be a positive value so that Equation 6.1 has real solutions. So, $Z^2 - 4nL \geq 0$, or $Z^2 \geq 4nL$. We prefer Z as small as possible. So, $Z = 2\sqrt{nL}$. Replacing the value of Z in the quadratic equation, we get $b = \sqrt{nL}/\rho$.

6.5 Experimental results

We select 15 matrices from the University of Florida sparse matrix collection for our experimentation. The basic information for each test matrix is given in Table 6.1. We run our experiments on an *intel core 2 processor Q6600*. It has L2 cache of 8MB and the CPU frequency is 2.40 GHz. Note that other column ordering algorithms reported in [59] and their performances are compared with the BRGC ordering algorithm in [29]. As reported in [29], the BRGC algorithm outperforms these other column ordering algorithms on three different computer architectures.

Table 6.1 shows that for most of the matrices the sum of non-zeros of type α, β and δ is more than $3n$. We compute the cache misses on the ideal cache model and SpMxV time for test matrices with given order, after the BRGC ordering, random ordering, and random ordering followed by the BRGC ordering. By random ordering, we mean a random order of both rows and columns. We present the cache misses and SpMxV time of each setup after normalization with respect to the cache misses and SpMxV time with the given order. We performed 1000 SpMxVs for recording the SpMxV time. These results are shown in Table 6.2. According to the experimentation, considering both cache complexity and SpMxV time, BRGC ordering improves for some matrices and fails to improve for others. In [74], similar improvements were observed, because the given matrices already have some nice sparsity structure. However, if those matrices do not have nice sparsity structures then it could be as worse as random ordering. The column for BRGC ordering after random ordering seems to be prominent in this case.

Matrix name	m	n	τ	ρ	α	β	δ
GL7d17	1548650	955128	25978098	16.0	3.97n	0.99n	1.07n
GL7d19	1911130	1955309	37322725	19.09	3.06n	0.99n	0.53n
wikipedia-20051105	1634989	1634989	19753078	12.08	1.14n	0.50n	2.15n
wikipedia-20070206	3566907	3566907	45030389	12.62	1.14n	0.50n	2.15n
cage14	1505785	1505785	27130349	18.02	7.39n	0.99n	2.21n
cage12	130228	130228	2032536	15.60	6.27n	0.99n	1.991n
GL7d24	21074	105054	593892	5.65	1.58n	0.95n	0.06n
barrier2-10	115625	115625	3897557	33.71	5.93n	0.99n	11.77n
fome21	67748	216350	465294	2.15	1.38n	0.72n	0.56n
hcircuit	105676	105676	513072	4.85	1.28n	0.99n	3.29n
lp_ken_18	105127	154699	358171	2.31	1.59n	0.63n	0.44n
rajat23	110355	110355	556938	5.05	1.87n	0.95n	2.37n
ldoor	952203	952203	23737339	24.92	7.65n	0.61n	0.0n
bcsstk32	44609	44609	1029655	23.08	6.80n	0.96n	0.0n
matrix_9	103430	103430	2121550	20.51	6.48n	1.0n	5.77n

Table 6.1: Test matrices with the number of non-zeros of type α, β and δ.

As shown in Table 6.3, the time required for BRGC ordering is less than that of at most 72 SpMxVs. This cost can be amortized easily in conjugate gradient type algorithms. The total number of iterations required in Algorithm 7 is shown in Table 6.3. It is greater than ρ for some matrices, such as wikipedia-20051105, wikipedia-20070206, ldoor, and bcsstk32. For these matrices (other than ldoor), we have a small fraction of columns that are not singleton after ρ iterations. For ldoor, we require 3ρ iterations, to complete Algorithm 7.

6.6 Conclusion

In this chapter, we propose a new re-ordering algorithm to improve the data locality of a sparse matrix during SpMxV. Our re-ordering algorithm is efficient. The cost of this preprocessing can be amortized easily in conjugate gradient type algorithms. The performance of our pre-processing algorithm on SpMxV is observed for some test matrices, and we believe the experimental data is promising.

Matrix name	Cache misses after BRGC ordering	Cache misses after Random ordering	Cache misses after Random ordering then BRGC ordering	SpMxV time after BRGC ordering	SpMxV time after Random ordering	SpMxV time after Random ordering then BRGC ordering
GL7d17	0.96	1.04	1.02	0.97	1.24	1.04
GL7d19	0.95	1.04	0.90	0.90	1.07	0.98
wikipedia -20051105	1.11	1.35	1.19	0.80	1.38	0.93
wikipedia -20070206	1.07	1.21	1.14	0.82	1.15	1.04
cage14	1.35	7.90	6.25	1.28	5.48	3.79
cage12	1.19	2.59	2.43	1.11	1.55	1.44
GL7d24	0.90	1.19	0.85	1.0	1.16	1.13
barrier2-10	0.24	1.55	0.49	0.94	1.18	1.04
fome21	1.34	2.08	1.43	1.10	1.42	1.30
hcircuit	1.03	1.44	1.28	1.08	1.38	1.23
lp_ken_18	1.08	1.26	1.10	1.33	1.22	1.08
rajat23	1.69	2.52	2.09	1.03	1.28	1.19
ldoor	1.00	27.71	3.55	1.02	4.62	1.44
bcsstk32	1.01	1.92	1.70	1.02	1.13	1.06
matrix_9	0.95	3.72	1.79	1.05	1.46	1.26

Table 6.2: Normalized cache misses on ideal cache model simulator and normalized CPU time for SpMxVs.

Matrix name	R	Iteration number	BRGC cost in terms of (SpMxVs)	Column no. left in BRGC ordering after ρ^c iterations
GL7d17	$0.62m$	3	19.19	-
GL7d19	$0.50m$	3	17.93	-
wikipedia-20051105	$0.78m$	47	18.91	374
wikipedia-20070206	$0.80m$	54	51.85	712
cage14	$0.5m$	4	40.85	-
cage12	$0.51m$	4	20.0	-
GL7d24	$0.14m$	4	33.33	-
barrier2-10	$0.81m$	17	55.86	-
fome21	$0.15m$	2	55.55	-
hcircuit	$0.82m$	4	71.43	-
lp_ken_18	$0.29m$	3	37.04	-
rajat23	$0.77m$	5	60.61	-
ldoor	$0.0m$	71	50.10	274614
bcsstk32	$0.0m$	188	41.67	7299
matrix_9	$0.68m$	4	70.59	-

Table 6.3: Preprocessing time.

Chapter 7

Implementation of Determinant by Condensation Method on GPU

We report on a GPU implementation of the condensation method designed by Abdelmalek Salem and Kouachi Said for computing the determinant of a matrix [63]. We consider two types of coefficients: modular integers and floating point numbers. We evaluate the performance of our code by measuring its effective bandwidth and argue that it is numerical stability in the floating point number case. In addition, we compare our code with serial implementation of determinant computation from well-known mathematical packages. Our results suggest that a GPU implementation of the condensation method has a large potential for improving those packages in terms of running time and numerical stability.

This chapter is based on the paper [30] co-authored with M. Moreno Maza.

7.1 Introduction

The celebrated algorithm of Charles Lutwidge Dodgson [14] (also known as Lewis Carroll) for computing the determinant of a square matrix $A = (a_{i,j} \mid 0 \leq i, j \leq n-1)$ of order n is a popular trick among students. It is, indeed, much easier to perform by hand on paper than the other classical methods, such as those based on minor expansion or Gaussian elimination. This is due to its amazing data traversal pattern. Each transformation step, from one array to the next one, is a streaming process, called a *condensation*. Dodgson's Algorithm can be executed as a stencil computation: the input data array is transformed into its determinant through $n - 1$ successive data arrays. This method suffers, however, from a serious algebraic limitation: it may fail to compute the targeted determinant. Indeed, after each condensation, the

newly generated matrix should have no zero elements in its interior [14] for the next condensation step to take place. The interior of A is the submatrix $\text{int}(A) = (a_{i,j} \mid 0 < i, j < n - 1)$. One can sometimes reduce to this case by combining rows or columns. When this is not possible, the algorithm terminates without producing any answers. In [63], Abdelmalek Salem and Kouachi Said have solved this difficulty by introducing another type of condensation.

One can easily realize that the condensation method (Dodgson's original method and the improved one by Salem and Said) can be executed in parallel. Moreover, we argue in this paper that its data traversal pattern makes it a good candidate for an implementation within a concurrency platform based on data-parallelism such as CUDA.

We report on an implementation of the algorithm described in [63] on GPU using CUDA. We consider two types of coefficients: modular integers and floating point numbers. In the first case, our contribution is to show that the condensation method can be implemented efficiently in terms of memory bandwidth, leading to a very competitive code with respect to popular software packages for computing determinant over finite fields (i.e. with modular integer coefficients). In the floating point case, our contribution is to show that the condensation method can be implemented efficiently in terms of numerical stability. We observe that the condensation method computes, in some sense, a factorization of the determinant. To take advantage of this fact, we use a new algorithm to compute the product of those factors such that, if overflow/underflow can be avoided then computations are ordered in a way that overflow/underflow is indeed avoided. The challenge is to keep the intermediate products within the range of machine floats; our solution is described in Section 7.4.

The organization of the paper is as follows. We describe the condensation method in Section 7.2. Its GPU implementation is presented in Section 7.3 and 7.4 for the finite field and floating point case respectively. Both of these two sections contain the corresponding experimental results. Concluding remarks are in Section 7.5.

7.2 The condensation method

In this section, we first review the condensation method described in [63]. We then analyze the algebraic complexity and cache complexity of this condensation method.

7.2.1 The formula of Salem and Said

As mentioned in the introduction, the authors of [63] have solved the algebraic limitation of Dodgson's Algorithm by introducing another type of condensation, which we summarize below. The input is a square matrix A of order $n > 2$. If the first row of $A = (a_{i,j} \mid 0 \leq i, j \leq n-1)$ is the null vector then the determinant of A is zero and the process terminates. Otherwise, let ℓ be the smallest column index of a non-zero element in the first row of A. The *condensation step* produces a matrix $B = (b_{i,j})$ of order $n-1$ defined by:

$$b_{i,j} = \begin{vmatrix} a_{0,\ell} & a_{0,j+1} \\ a_{i+1,\ell} & a_{i+1,j+1} \end{vmatrix}$$

for $j \geq \ell$ and by $b_{i,j} = -a_{i+1,j}a_{0,\ell}$ for $j < \ell$. The key relation between A and B is the following:

$$\det(A) = \det(B)/(a_{0,\ell})^{n-2} \tag{7.1}$$

We call $a_{0,\ell}$ the *pivot* of the condensation step. Formula (7.1) implies that the condensation method of Salem and Said computes $\det(A)$ as a product of powers of the inverse of the successive pivots. In the finite field case, this product can be accumulated in a variable, that is, this variable is updated after each condensation step. In the floating point number case, these powers can be accumulated in a list so that their product can be performed in a way that overflow/underflow is avoided, if possible, as we shall see in Section series.

7.2.2 The algebraic complexity of the condensation method

Algebraic complexity estimates are given for the RAM model with memory holding a finite number of s-bit words, for a fixed s [65]. Each condensation step involves two matrices: A and B of order n and $n-1$, respectively. Computing B from A requires $2(n-1)^2$ multiplications and $(n-1)^2$ subtractions considering that ℓ refers to the first column (which is the worst case for computing B). The best case happens when ℓ is the last column. When this happens each condensation requires $(n-1)^2$ multiplications. The number of operations involved in finding ℓ is linear in n. The whole algorithm takes at most $n-2$ condensation steps before terminating. So, the total cost for computing the determinant is bounded by $O(n^3)$ arithmetic operations. Moreover, in the worst case, a precise account is $n^3 - 3/2n^2 + 1/2n - 3$, which is comparable to the worst case of Gaussian Elimination.

7.2.3 The cache complexity of condensation method

Cache complexity estimates are given for the ideal cache model. The ideal cache model [17] is a fully associative cache. Its cache replacement policy is optimal in that the cache line to be evicted is one which will be required again furthest in future.

Before estimating the cache complexity of a condensation step, we need to describe the data structures used to represent a square matrix of order n in our implementation. We represent A by a one-dimensional array $\alpha[0, 1, \ldots, n^2-1]$ of size n^2. We use *column major layout.* that is, the sub-array $\alpha[i, i+1, \ldots, i+n-1]$, for $i = 0, n, 2n, \ldots (n-1)n$ represents the i-th column. In particular the element $a_{i,j}$ is stored in $\alpha[i + j * n]$.

Consider an ideal cache of Z words, with cache line of L words. To make the analysis simple, we assume that n is large enough such that one column of A does not fit into the cache. Let α and β be two one-dimensional arrays of size n^2 and $(n-1)^2$, representing the input matrix A and output matrix B of the condensation method, respectively. In each condensation step, matrix B is created from A. Assume that, our algorithm computes B sequentially column-wise. This involves the following data traversals:

- Each element of B is visited only once.
- Each element of A (if it is neither in first row nor in ℓ-th column) is visited only once.
- The ℓ-th column of A is scanned $n-1$ times.
- Each element of the first row of A is visited $n-1$ times consecutively.

It follows that one condensation step incurs $2(n-1)^2/L + n/L + 3$ for α and $(n-1)^2/L+1$ for β, thus $3(n-1)^2/L+n/L+4$ cache misses in total. Summing over the condensation steps for $k = Z + 1 \cdots n$ (that is, those for which one column does not fit it in cache) we obtain

$$\frac{(n-Z)\,(n^2-n+Z^2-Z+Zn+1+4\,L)}{L} \tag{7.2}$$

Therefore, asymptotically, the ratio between the algebraic complexity and the cache complexity is L. This is similar to Gaussian Elimination. However, the condensation method works in a more data-oblivious way: at each condensation step, apart from the search of the pivot, the same operations are performed independently of the data. This regularity pattern facilitates scheduling, in particular hardware scheduling as

it is the case on a GPU. Gaussian Elimination does not have this feature. Indeed, permutations of rows and columns may be needed before proceeding to the next step.

7.3 GPU implementation: the finite field case

As mentioned in the introduction, the GPU implementation reported in this paper handles two types of coefficients, namely modular integers and floating point numbers. In both cases, each coefficient is stored in a fixed number of machine words and hardware arithmetic is used as much as possible for efficiency consideration. This latter property is more difficult to achieve in the case of modular integers and we discuss it in Section 7.3.2 Numerical stability is the challenge of floating point number arithmetic and we address it in Sections 7.4.1 and 7.4.2. Other types of coefficients, such as multi-precision integers, could be also considered and we leave it for future work.

Before describing the issues specific to the finite field case (in other words to modular integers) we present the part of our implementation which is common to both scenarios. More precisely, we discuss in Section 7.3.1 our GPU implementation strategy, in particular the mapping between thread blocks and data. In section 7.3.3, we report on our experimentation with the condensation method for matrices over finite fields. This, of course, is primarily dedicated to evaluate the performance of our GPU implementation, but also to compare it with serial implementations of determinant computation available in computer algebra packages One of our goals is to understand to which extent GPU implementation could improve those packages.

7.3.1 Data mapping

Each condensation step is performed by one CUDA kernel call. The matrices α and β, introduced in Section 7.2.3, are stored in the global memory of GPU. After each condensation step, instead of copying β back to CPU main memory, we simply "swap the pointers" to these arrays.

In practice, we find that the index ℓ is small. So we dedicate one kernel call, with a single thread in a single block, to find ℓ. Once we get ℓ, we compute the $(n-2)$-th power of the inverse of $\alpha[\ell * n]$. We call it *pivot*. We also store the product of the successive pivots in this kernel call.

The kernel performing a condensation step uses one-dimensional blocks and threads. Let T be the number of threads in a block. Each thread is responsible

to compute t elements of the array β (representing B). So the total number of blocks required to compute β is $\lceil (n-1)^2/(Tt) \rceil$. Consider thread i is in block j. Then this thread is responsible for computing $\beta[Ttj + it, Ttj + it + 1, \ldots Ttj + it + t - 1]$.

7.3.2 Finite field arithmetic

Multiplying two elements a, b modulo a prime number p is obviously a key routine. Unlike the case of single and double precision floating point arithmetic, the operation $(a, b, p) \longmapsto (ab) \mod p$, for $a, b, p \in \mathbb{Z}$, is not provided directly by the hardware. This operation is thus an efficiency-critical low-level software routine that the programmer must supply. When p is a machine word size prime, which is the assumption in this paper, two techniques are popular in the symbolic computation community.

The first one takes advantage of hardware floating point arithmetic. We call `double_mul_mod` our implementation of this technique, for which our CUDA code is shown below. The fourth argument `pinv` is the inverse of `p` which is precomputed in floating point.

```
__device__ int double_mul_mod(int a, int b, int p, double pinv) {
   int q  = (int) ((((double) a) * ((double) b)) * pinv);
   int res = a * b - q * p;
   return (res < 0) ? (-res) : res;
}
```

In our implementation, double precision floating point numbers are encoded on 64 bits and make this technique work correctly for primes p up to 30 bits.

The second technique, called the *Montgomery reduction* relies only on hardware integer arithmetic. We refer to Montgomery's paper [51] for details. We have experimented both approaches in [72]. Our CUDA implementation favors the `double_mul_mod` trick.

7.3.3 Experimental results

We generate random integer matrices modulo a prime number p of machine word size. The order of our test matrices varies from 10 to 4000. We conduct all our experiments on a GPU NVIDIA Tesla 2050 C.

We use *effective memory bandwidth* to evaluate our GPU code. The effective memory bandwidth (measured in GB/seconds) of a kernel run is, by definition,

- the amount of data traversed in the global memory of the GPU during the kernel run,

- divided by the running time of the kernel.

Following a principle proposed by Greg Ruetsch and Paulius Micikevicius in [62], we compared the effective memory bandwidth of our kernel to that of a *copy kernel*, that is, a kernel that simply performs one memory copy from one place to another place in the global area of GPU. Such benchmark kernel can be regarded as a good practical measure of the maximum memory bandwidth of a kernel.

For matrix of order of 3000, the effective memory bandwidth of the copy kernel and our condensation method (with modular integer coefficients) on our card are 96 GB/s and 18.5 GB/s respectively.

Our effective memory bandwidth results show that our code is reasonably efficient considering the following two facts:

- the index calculation in our code is not straightforward and
- finite field arithmetic (see Section 7.3.2) is time consuming.

Figure 7.1 reports on the memory bandwidth of our CUDA code for different matrix orders.

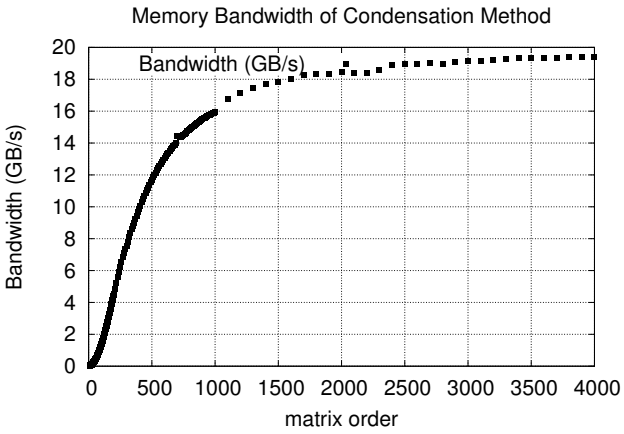

Figure 7.1: Effective memory bandwidth of condensation method.

To conclude this section, we compare our CUDA code for computing determinants over finite fields with its counterpart in two popular mathematical software packages, namely MAPLE[1] and NTL[2].

Figure 7.2 compares the computing time between our CUDA code for the condensation method and the NTL determinant code, both with modular integer coefficients.

[1] `http://www.maplesoft.com/`
[2] `http://www.shoup.net`

It shows clearly that the condensation method in CUDA outperforms NTL code for computing determinant over finite fields. For example, when $n = 4000$, the condensation method in CUDA takes about 9 seconds, while NTL code takes about 850 seconds.

Figure 7.3 compares the computing time between our CUDA code for the condensation method and MAPLE's determinant command over finite fields. It shows clearly that the condensation method in CUDA outperforms MAPLE code for computing determinant over finite fields. For example, when $n = 4000$, the condensation method in CUDA takes about 9 seconds, whereas MAPLE code takes about 90 seconds.

7.4 GPU implementation: the floating point case

In this section, we consider the case of matrices with floating point number coefficients. We adapt our CUDA code described in Section 7.3 to this new scenario. The modifications are described in Section 7.4.1. One potential challenge that we found is to multiply the successive pivots. Mathematically, the problem is to multiply a sequence of floating values where the intermediate results might not be in the range of the floating point number data type while the final results might be. We state the problem and our solution in Section 7.4.2. We conclude the section by providing experimental results.

7.4.1 Finding the pivots

Instead of taking the first nonzero from the left in the first row of A, we choose the nonzero element of the first row that is closest to value 1.0; let use call $p = a_{0,\ell}$ this element. We have verified that this modification of the original algorithm in [63] does not invalidate the expected result, namely the determinant. For simplicity, we are describe the procedure for matrices A and B instead of the arrays α and β. Once p is chosen, all elements in the ℓ-th column are divided by p including $a_{0,\ell}$. Thus we modify Formula (7.1) as follows:

$$\det(A) = \det(B) * p$$

We call p the pivot for this floating point number implementation.

The benefits of the above transformation are as follows.

- $a_{0,\ell}$ becomes 1.0. So we need neither computing the $(n-2)$-th power of it nor performing any division at the end.
- By choosing an element that is the closest to 1.0, we are expecting to reduce the potential of overflow/underflow.

7.4.2 Multiplication of the successive pivots

We first state the problem that we wish to address. Consider an array $x[0, 1, \ldots, k-1]$ of k floating point numbers, encoded by a floating point number data type of fixed precision. Then the problem is to write an algorithm for computing the product $X = \prod_{i=0}^{k-1} x[i]$, assuming that X fits within the range of the given floating point number data type. Our solution to this problem is stated as Algorithm 10 hereafter.

We give a sketch of the proof of Algorithm 10. We observe that multiplications occur at Lines 6, 17 and 25. The multiplication at Line 6 cannot lead to overflow/underflow since $|q1| \leq 1 \leq |q2|$ holds. If Lines 17 or 25 would lead to overflow/underflow, this would bring a contradiction to our hypothesis.

We estimate the running time of Algorithm 10. The first *while* loop runs in linear time with the number of elements in x. The second *while* loop runs $m-1$ times considering there exists m elements in R. Each of the iteration takes $O(m)$ time. So the time complexity of the second *while* loop is $O(m^2)$. Considering the inequality $k \geq m$, the time complexity of the Algorithm is $O(k^2)$.

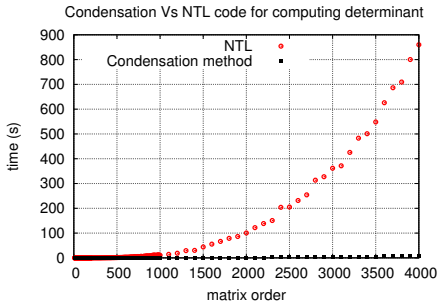

Figure 7.2: CUDA code for condensation method and determinant on NTL over finite field.

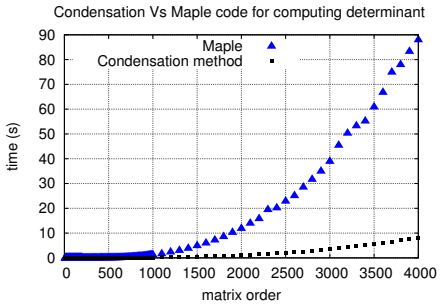

Figure 7.3: CUDA code for condensation method and determinant on MAPLE over finite field.

7.4.3 Experimentation

For the experimentation in the case of floating point coefficients matrices with MAPLE, we use the *Determinant* command of the *LinearAlgebra* package. In this

Algorithm 10: MulSuccPivot(X)

Input: X, list of floating point numbers
Output: R, product of numbers in X.

```
1  Create a stack S1 of elements of x in [−1.0, 1.0];
2  Create a stack S2 of the other elements of x not in S1;
3  while both S1 and S2 are nonempty do
4      q1 = pop(S1);
5      q2 = pop(S2);
6      q = q1 ∗ q2;
7      if q is in [−1.0, 1.0] then
8          push(q, S1);
9      else
10         push(q, S2);
11 if stack S1 is not empty then
12     make a list R with the elements in S1;
13     while R has more than one element do
14         select r1 and r2 in R such that r1 and r2 are closest to 0 and |1.0|
           respectively;
15         r = r1 ∗ r2;
16         delete r1 and r2 from R;
17         insert r into R;
18 else
19     make a list R with the elements in S2;
20     while R has more than one element do
21         select any r1 and r2 in R;
22         r = r1 ∗ r2;
23         delete r1 and r2 from R;
24         insert r into R;
25 return R;
```

case, MAPLE may not have the best possible implementation, since MAPLE's primary purpose is symbolic computation. However, MATLAB[3] has certainly a competitive implementation for floating point coefficients. Indeed, efficiently supporting numerical linear algebra is the primary goal for this cutting-edge software.

For small finite field coefficients, the best serial algorithm is simply Gaussian elimination, which is what MAPLE is using. Therefore, in the case of modular integers, our comparison reported in Section 7.3.3 is also meaningful.

In order to investigate the numerical stability of our GPU implementation of the condensation method, we use the infamous Hilbert matrix $H_{ij} = \frac{1}{i+j-1}$, which is a canonical example of ill-conditioned matrix. This matrix is non-singular, for each value of n. However, for n large enough, any determinant computation of this matrix using a fixed precision floating point number arithmetic returns zero.

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \end{bmatrix}$$

In the tables below, we compare determinant computation of the Hilbert matrix with

- MAPLE using multi-precision floating point number arithmetic (thus software floating point number),
- MATLAB using double-precision floating point number arithmetic,
- our CUDA implementation using double-precision floating point number arithmetic.

We observe that:

1. despite of the use of multi-precision floating point, MAPLE is less accurate than MATLAB and our CUDA implementation (this was checked by computing the exact value of the determinant using rational number arithmetic),
2. our CUDA implementation of the condensation method can compute determinants of much larger order than MATLAB,

[3] `http://www.mathworks.com/`

Matrix order	MAPLE	MATLAB	Condensation method on GPU
5	0.3239712e-11	3.749295e-12	3.74967e-12
6	-0.1037653175e-16	5.367300e-18	5.36556e-18
7	-0.2940657217e-22	4.835803e-25	4.44292e-25
8	-0.2156380381e-28	2.737050e-33	-3.92813e-33
9	-0.1692148341e-35	9.720265e-43	-2.79235e-41
10	0.4704819751e-42	2.164405e-53	-4.44342e-50
15	0.1386122551e-74	-2.190300e-120	-9.47742e-103
20	0.4711757502e-106	-1.100433e-195	3.81829e-164
25	-0.4092672466-139	5.482309e-274	-3.82134e-239
30	-0.2087134536-174	0	-2.50914e-319
35	0.6863051439e-205	-	3.50293e-398
40	0.3354475665e-237	-	-7.42227e-479
70	-0.1605231989e-443	-	-1.42973e-961
100	-0.1344119185e-667	-	1.96009e-1467
200	-0.1635472167e-1423	-	9.43651e-3169
295	-0.1313897019e-2117	-	3.27673e-4811
300	0.4832058492e-2154	-	-1.95564e-4897
320	0.1012376674e-2298	-	7.2904e-4951
340	0.3198288621e-2442	-	-8.67557e-4949*2.08848e-644
360	0.6712616355e-2593	-	9.84118e-4938*8.32678e-1006
380	-0.1532669346e2736	-	-3.28068e-4950*-6.51644e-1341
400	-0.4230797452e-2881	-	-6.19676e-4945*6.56337e-1696
500	-0.1956609252e-3608	-	1.40177e-4939*-2.22223e-3444
600	-0.4139972675e-4335	-	-2.55164e-4950*1.99856e-4945*6.19736e-232
800	0.4570493645e -5853	-	4.25009e-4940*-2.21715e-4940*-2.17891e-3739

Table 7.1: Determinant of Hilbert matrix by MAPLE, MATLAB, and condensation method on both CPU and GPU.

3. our CUDA implementation is also competitive with MATLAB in terms of running time.

7.5 Conclusion

MAPLE and MATLAB commands for computing matrix determinants combine many different state-of-the-art algorithms. On a given input, MAPLE and MATLAB determinant commands choose one of these algorithms by considering the types of the coefficients and the combinatorial properties (size, sparsity) of the input matrix. These choices are heavily tuned since linear algebra is, in the case of MAPLE, at the core

Matrix order	MAPLE	MATLAB	Condensation method on GPU
5	0.004	0	0.000530
6	0.008	0	0.000570
7	0.012	0	0.000595
8	0.008	0	0.000631
9	0.012	0	0.000741
10	0.012	0	0.000447
15	0.016	0	0.000964
20	0.016	0	0.001078
25	0.020	0	0.001271
30	0.024	-	0.001460
35	0.044	-	0.001671
40	0.036	-	0.001896
70	0.188	-	0.003083
100	0.588	-	0.005145
200	5.988	-	0.012488
295	20.733	-	0.023402
300	21.661	-	0.023759
320	26.741	-	0.026633
340	31.677	-	0.029433
360	38.150	-	0.032401
380	46.146	-	0.035940
400	54.099	-	0.038955
500	104.334	-	0.058193
600	187.151	-	0.081969
800	467.541	-	0.147037

Table 7.2: Time(s) required to compute determinant of Hilbert Matrix by MAPLE, MATLAB, and condensation method on both CPU and GPU.

of its symbolic routines while it is, in the case of MATLAB, at the core of the whole system. Therefore, comparing our code against those systems is meaningful.

From our experimental results, it is clear that the condensation method implemented on the GPU is a promising candidate for computing determinants of matrices with both modular integer coefficients and floating point number coefficients.

Though it seems unfair in the first place that we compare our parallel code with serial codes in mathematical packages, our primary objective is to propose parallel algorithms for computing determinants within mathematical software packages, such as MAPLE and MATLAB. Actually, these two systems are already able today to take advantage of multi-core processors and GPUs for certain types of computations. Therefore our objectives are meaningful and motivated by our active cooperation with

the Maplesoft company developing MAPLE. We believe that a GPU implementation of the condensation method can be used to improve the efficiency, in terms of running time and numerical stability, of existing mathematical software packages.

Chapter 8

Implementation of Plain Multiplication for Univariate Polynomials on GPU

In this chapter, we describe an implementation of *plain* univariate polynomial multiplication on many-core machine. In practice, our implementation, using CUDA outperforms an optimized FFT-based implementation (using CUDA also) for fairly large degrees. We analyze our algorithm on our proposed many-core machine model.

This chapter is based on the paper [31] co-authored with M. Moreno Maza.

8.1 Introduction

Let $\mathbb{K}$ be a field and $a, b \in \mathbb{K}[X]$ be two univariate polynomials over a $\mathbb{K}$ and with variable X. Let n and m be positive integers such that $\deg(a) = n - 1$ and $\deg(b) = m - 1$. We assume that each binary arithmetic operation (addition, subtraction, multiplication and division) in $\mathbb{K}$ can be done with a single machine word operation of a streaming multiprocessor (SM) of an MMM program.

$$a = a_{n-1}X^{n-1} + \cdots + a_1X + a_0 \text{ and } b = b_{m-1}X^{m-1} + \cdots + b_1X + b_0, \text{ with } n \geq m. \quad (8.1)$$

We start from the so-called *long multiplication*[1], which computes the product $a \times b$ in the way we all learned integer multiplication in primary school. Multiplying a and b by plain arithmetic requires $O(nm)$ arithmetic operations on RAM model. The cache

[1] http://en.wikipedia.org/wiki/Multiplication_algorithm#Long_multiplication

			$a =$		X^5+	$8X^4+$	$2X^3+$	$2X^2+$	$6X+$	7
			$b =$		X^5+	$2X^4+$	$4X^3+$	X^2+	$3X+$	2
					$2X^5+$	$16X^4+$	$4X^3+$	$4X^2+$	$12X+$	14
				$3X^6+$	$24X^5+$	$6X^4+$	$6X^3+$	$18X^2+$	$21X$	
			X^7+	$8X^6+$	$2X^5+$	$2X^4+$	$6X^3+$	$7X^2$		
		$4X^8+$	$32X^7+$	$8X^6+$	$8X^5+$	$24X^4+$	$28X^3$			
	$2X^9+$	$16X^8+$	$4X^7+$	$4X^6+$	$12X^5+$	$14X^4$				
$X^{10}+$	$8X^9+$	$2X^8+$	$2X^7+$	$6X^6+$	$7X^5$					
$X^{10}+$	$10X^9+$	$22X^8+$	$39X^7+$	$29X^6+$	$55X^5+$	$62X^4+$	$44X^3+$	$29X^2+$	$33X+$	14

Table 8.1: Long multiplication ($n = m = 5$).

complexity of this algorithm is $Q(n, m) = 2 + \frac{2mn+m+n-1}{L} + n(2 + \frac{1}{L})$. In Table 8.1, we show an example of long multiplication.

For simplicity, let, $n = m$. In [54], M. Moreno Maza and W. Pan analyze the span, parallelism and cache complexity of this problem on multi-core machine. They proposed a parallel algorithm for plain polynomial multiplication. According to their analysis the span and cache complexity are $\Theta(n)$ and $O(\frac{n^2}{LZ})$ respectively. The total number of words of storage required is $\Theta(n \log n)$. The total amount of memory accesses, cache misses, allocation and deallocation of memory make the overhead significant such that the algorithm does not perform well compare to the implementation of *Fast Fourier Transforms (FFTs)* based multiplication.

Both the memory access pattern and the for-loop parallelization overheads in the implementation of a FFT based polynomial multiplication restrict linear speedup to input vectors of very large sizes, say 2^{20}, according to [57, 56]. In contrast, serial FFT code provide high-performance even for input vectors of relatively small sizes, say 2^{10}. This is the case with the standard libraries FFTW [16], NTL and Spiral [60]. As a consequence, higher level algorithms, that heavily rely on FFTs in their serial implementation, require additional supporting routines for small/average size problems, when targeting implementation on multi-core architectures. Examples of such higher level algorithms are fast evaluation and fast interpolation based on sub-product tree techniques, see Chapter 10 in [20]. As for serial code on CPUs, parallel code on GPUs for dense polynomial arithmetic relies on a combination of asymptotically fast and plain algorithms. Those are employed for data of large and small size, respectively. Parallelizing both types of algorithms is required in order to achieve peak performances.

Graphics processing units (GPUs) offer a higher level of concurrent memory access than multi-core architectures. Moreover, thread scheduling is done by the hardware,

which reduces for-loop parallelization overheads significantly. Despite of these attractive features, and as reported in [53], highly optimized FFT implementation on GPUs are not sufficient to support the parallelization of higher level algorithms, such as dense univariate polynomial arithmetic. To give a figure, for vectors of size 2^{18} and 2^{26}, speedup factors (w.r.t. a C serial implementation) are 9 and 37 respectively on a NVIDIA Geforce GTX 285 running CUDA 2.2, as reported in [53].

8.1.1 Elements of syntax

In this section, we describe the representation of univariate polynomials on many-core machine model. We also describe syntax for *kernel call* in many-core machine algorithms.

Each polynomial (for example b) is represented by a coefficient array (lets, $b[0, \ldots, \deg(b)]$). The length of the array is the length of the polynomial (for b it is $\deg(b) + 1$). The elements in the array are in order. For example, $b[\deg(b)]$ is the leading coefficient.

All of our kernel has one dimensional threads, and thread blocks. Each thread, has a local variable j which is defined as the *rank* of the thread over all the threads in all thread blocks. It is computed as $t+\mathsf{blockID}\times\ell$, where t and ℓ are the thread id and the number of threads in a thread block respectively. In this paper, we represent rank and thread id by j and t respectively.

We launch a kernel by the following format: $\mathsf{kernelName}\lll c, \ell \ggg (p_0, \ldots, p_{n-1})$. Here, ℓ and c are the threads number in a thread block and the thread blocks number respectively. $\mathsf{kernelName}$ is the name of our kernel. $(p_0, \ldots, p_{n-1})$ are the parameters for this kernel.

8.2 Polynomial multiplication algorithms

Let $d \in \mathbb{K}[X]$ such that $d = ab$. The main idea of the algorithm is to multiply each coefficient of a with every coefficient of b then apply shifted additions among the coefficient multiplications to obtain d.

Our implementation has two phases: *multiplication phase* and *addition phase*. In Section 8.2.1 and 8.2.2 we describe these two phases followed by the analysis of our algorithm. Algorithm 11 is our top level algorithm. We consider a number of assumptions on the input polynomial in order to keep our algorithm simple and clear. First, let x be a small positive integer. Second, let m is divisible by x and $r = \frac{n}{x} = 2^q$

for $q \geq 0$. Third, $(n + x - 1) \geq 2\ell$ and it is divisible by ℓ, where ℓ is the number of threads in a thread block. The algorithm can be modified easily if any of the assumptions does not hold, and it does not change our complexity analysis.

Algorithm 11: PlainMultiplicationGPU(a, b, d, x)

Input: $a, b \in \mathbb{K}[X]$ with $n - 1 = \deg(a)$ and $m - 1 = deg(b)$ and an integer $x \geq 1$.
Output: $d \in \mathbb{K}[X]$ and $d = ab$.

```
1 c = n + x − 1; r = m/x; t = rc;
2 Let M be an array of size t with coefficients in K ;
3 ℓ is the number of threads per block;
4 MulKer⋘ rc/ℓ, ℓ ⋙ (a, b, M, n, x);
5 d is an array that stores ab;
6 for (i = 0; i ≤ log_2 r; i = i + 1) do
7     AddKer⋘ rc/(ℓ2^{i+1}), ℓ ⋙ (M, d, c, x, r, i);
8 return d;
```

8.2.1 Multiplication phase

Algorithm 12 is responsible for completing the multiplication phase. Each thread copies one coefficient from polynomial a and stores the coefficient in local memory A'. A thread block copies x coefficients from b into local memory B'.

In a thread block, multiplications between A' and B' are done by all threads in parallel and the results are written into the appropriate places in M. More precisely, each thread associated with j does x multiplications and $x-1$ additions in this phase, and then save the result in an entry of $M[j]$.

Figure 8.1 describes the computation of array M. Each thread block computes ℓ entries of M, which are created from ℓ coefficients and x coefficients from a and b respectively .

8.2.2 Addition phase

The main idea of adding the intermediate multiplication results into d is similar to *parallel scan* [32]. Algorithm 13 completes this phase. Consider M as a two dimensional array of $r \times c$. For example, coefficients in $\{M[i\,r] \cdots M[i\,r + r - 1]\}$ creates the i-th row. Let all rows of the matrix are unchecked at the very beginning. In one parallel steps, it first makes a number of pairs of consecutive unchecked rows and applies addition operations between coefficients, such that one coefficient is from

Algorithm 12: MulKer(a, b, M, n, x)

Input: $a, b, M \in \mathbb{K}[X]$ and an integer $x \geq 1$.

1 j =blockID blockDim +threadID; t =threadID;
2 Let B' and A' be two local arrays of size x and blockDim respectively with coefficients in $\mathbb{K}$;
3 $i = j \mod (n + x - 1)$;
4 **if** $i \geq n$ **then**
5 $\quad A'[t] = 0$;
6 **else**
7 $\quad A'[t] = a[i]$;
8 **if** $t < x$ **then**
9 $\quad B'[t] = b[\lfloor (j/(n + x - 1)) \rfloor x + t]$;
/* copying from global. */
10 $f = 0$;
11 **for** $(k = 0; k < x \wedge (i - k) \geq 0; k = k + 1)$ **do**
12 $\quad f = f + A'[i - k]\, B'[k]$;
$\quad$ /* both coefficient addition and multiplication operations are done in $\mathbb{K}$. */
13 $M[j] = f$;
/* writing to global memory. */

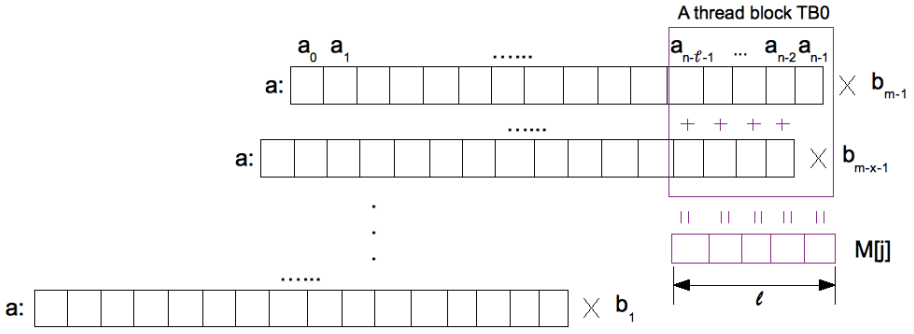

Figure 8.1: Dividing the work of coefficient multiplication among threadblocks.

first row and the other is from the second row and the result is stored into the second row. By first row, we refer to the row that has smaller index than that of the other row in the pair. Note that these additions are all valid addition operations considering the long multiplication algorithm. If there is no such coefficient is found in the second row, we add it with an appropriate coefficient in d. At the end, we mark the first row as checked. We continue this procedure until, all rows are checked. We need $\log_2 r + 1$ number of parallel steps to complete the above procedure. In the last step, there exists only one unchecked row. All coefficients from the unchecked row are

added with the corresponding coefficient in d. Each of parallel steps is completed by one kernel.

In Algorithm 13, during i-th call, does one coefficient addition. First it requires to find a row pair of M (we treat M as a two dimensional matrix of $r \times c$) between which it does the coefficient addition. Let the row indices of a pair be (s, e) for a thread. It can be observed that during i-th call all rows whose index are less than 2^{i-1} were already checked by previous kernels (if any). The difference between the indices in a row pair is 2^i. These rules are used in computing (s, e) in Line 5-6. The next job is to find out coefficients from each row, then add the elements. In Line 3, we compute k, which tells the index of the coefficient from s. The index of the other coefficient in row e is $k + 2^i\, x$. If no such coefficient found in the other row, the coefficient from the first row is added directly with d (in Line 7-8). Note that, the algorithm does not explicitly check the first row. Each thread is accessing the global memory for at most 3 times without storing the coefficients into the local memory.

Algorithm 13: AddKer(M, d, c, x, r, i)

Input: $M, d, \in \mathbb{K}[X]$ and c, x, r, i are positive integers.

1 j =blockID blockDim +threadID; t =threadID;
2 $k = j \mod c$;
3 $q = \lfloor j/c \rfloor$;
4 $s = 2^i - 1 + 2^{i+1}\, q$;
5 $e = s + 2^i$;
6 **if** $k < 2^i\, x$ **then**
7 $\quad d[s\, x + k] = d[s\, x + k] + M[s\, r + k]$;
8 **else**
9 $\quad M[e\, r + k - 2^i\, x] = M[e\, r + k - 2^i\, x] + M[s\, r + k]$;

/* coefficient addition operations are done in $\mathbb{K}$. */

8.2.3 Arbitrary x

We denote by W_{x}, S_{x} and, O_{x}, the work, span and overhead, respectively. Each thread block performs $\ell\,(2\,x - 1)$ arithmetic operations in multiplication phase, and ℓ arithmetic operations in addition phase. Each thread requests at most 3 accesses to the global memory. We obtain the following estimates, where δ stands for $n + x - 1$ and μ stands for $2\,x - 1$,

$$W_{\mathrm{x}} = (2\,m - \frac{1}{2})\,\delta,\ S_{\mathrm{x}} = \mu + \log_2 \frac{m}{x},\ O_{\mathrm{x}} = \frac{3\,\delta\,(2\,m - x)\,U}{x\,\ell}. \tag{8.2}$$

In order to apply Theorem 1, we shall compute the quantities $N(\mathcal{P})$, $L(\mathcal{P})$ and $C(\mathcal{P})$ defined in Section 3.2. We denote them here by N_{x}, L_{x} and C_{x}, respectively. One can easily check that we have

$$N_{\mathrm{x}} = \frac{\delta\,(2\,m - x)}{x\,\ell},\ L_{\mathrm{x}} = \log_2 \frac{m}{x} + 1 \ \text{ and } \ C_{\mathrm{x}} = \mu + 3\,U. \tag{8.3}$$

8.2.4 Comparison of running time estimates

For this application, we use "naive algorithm" as the one obtained by setting $x = 1$ in Algorithm 11. For fixed n and m, the overhead ratio increases as x increases, since we have

$$\frac{O_1}{O_{\mathrm{x}}} = \frac{n\ (2\,m - 1)\,x}{(n + x - 1)\,(2\,m - x)}. \tag{8.4}$$

Next, we observe that this substantial improvement is done at a fairly low expense in terms of work overhead. Indeed, the work ratio is asymptotically constant, since we have

$$\frac{W_1}{W_{\mathrm{x}}} = \frac{n}{n + x - 1}. \tag{8.5}$$

Applying Theorem 1, the running times on p SMs of the naive algorithm and the algorithm with arbitrary x are bounded over by

$$(N_1/p + L_1) \cdot C_1 \ \text{ and } \ (N_{\mathrm{x}}/p + L_{\mathrm{x}}) \cdot C_{\mathrm{x}}. \tag{8.6}$$

and replace m by n. When n escapes to infinity, the ratio R is equivalent to

$$\frac{(1 + 3\,U)\,x}{2\,x - 1 + 3\,U}. \tag{8.7}$$

One can assume $3\,U > 1$, which implies that the above ratio is always greater than 1 as soon as $x > 1$ holds. Therefore, the algorithm with arbitrary x outperforms the naive one.

8.2.5 Experimental results

We have experimented the CUDA implementation of the plain univariate multiplication described in the previous section. We use both *balanced* and *unbalanced* pairs of polynomials, see Table 8.2 and 8.3 respectively. By balanced, following [57], we mean a pair of univariate polynomials of equal degree, which is a favorable case for optimized FFT-based polynomial multiplication.

degree	GPU Plain multiplication	GPU FFT-based multiplication
2^{10}	0.00049	0.0044136
2^{11}	0.0009	0.004642912
2^{12}	0.0032	0.00543696
2^{13}	0.01	0.00543696
2^{14}	0.045	0.00709072

Table 8.2: Comparison between plain and FFT-based polynomial multiplications for balanced pairs ($n = m$) on CUDA.

$\mathsf{degree}(A)$	$\mathsf{degree}(B)$	GPU Plain multiplication
2^{10}	2^{8}	0.00041
2^{11}	2^{8}	0.0005
2^{11}	2^{10}	0.00073
2^{12}	2^{8}	0.00057
2^{12}	2^{10}	0.0011
2^{13}	2^{8}	0.00074
2^{13}	2^{10}	0.0018
2^{13}	2^{12}	0.0061
2^{14}	2^{8}	0.0010
2^{14}	2^{10}	0.0031
2^{14}	2^{12}	0.011
2^{14}	2^{13}	0.02

Table 8.3: Computation time for plain multiplication on CUDA for unbalance pairs ($n \neq m$).

In Table 8.2, we compare the computation time of our CUDA based implementation of parallel plain multiplication with the highly optimized FFT-based multiplication in CUDA reported in [53]. Our implementation outperforms that of FFT-based multiplication until the degree 2^{12}.

FFT-based multiplication may not perform well for unbalanced pairs, see [57] for details. In plain multiplication, this is not true. Computation times for plain multiplication on CUDA of unbalanced pairs are reported in Table 8.3.

8.3 Conclusion

The number of multiprocessors in a GPU is limited. So, we can not expect that the our implementation outperform FFT based multiplication. For our GPU card our code is better than that of FFT based multiplication up to $n = 2^{12}$. On other card, it might be different. We expect a better performance of our algorithm with

GPU card that has more local memory. It is obvious that, at some point, FFT based multiplication outperforms plain multiplication. One important feature of our algorithm is that, we do not need to compute the root of unity. Moreover, in case of FFT based multiplication, both the length of a and b in coefficient representation need to be equal and full power of 2. In plain multiplication, it is not required. The consequence of this feature is that, our implementation have better performance in case of unbalance polynomial.

Chapter 9

Implementation of the Euclidean Algorithm for Univariate Polynomial GCDs on GPU

In this chapter, we describe both plain univariate polynomial division and Euclidean algorithm on many-core machine. We compare our experimental results with the implementations found in NTL. We also analyze our algorithms on our proposed many-core machine model.

This chapter is based on the paper [31] co-authored with M. Moreno Maza.

9.1 Introduction

Let $\mathbb{K}$ be a field and $a, b \in \mathbb{K}[X]$ be two univariate polynomials over a $\mathbb{K}$ and with variable X. We assume that each binary arithmetic operation (addition, subtraction, multiplication and division) in $\mathbb{K}$ can be done with a single machine word operation of a streaming multiprocessor (SM) of an MMM program (like Chapter 8). We assume that b is not zero. Let n and m be positive integers such that $\deg(a) = n - 1$ and $\deg(b) = m - 1$. Thus n and m are the number of terms (null or not) of a and b, respectively. Let q and r be the quotient and the remainder in the Euclidean division of a by b. Thus, (q, r) is a unique couple of univariate polynomials over $\mathbb{K}$ such that $a = b\,q + r$ and $\deg(r) < \deg(b)$ both hold.

$$a = a_{n-1}X^{n-1} + \cdots + a_1X + a_0 \text{ and } b = b_{m-1}X^{m-1} + \cdots + b_1X + b_0, \text{ with } n \geq m. \quad (9.1)$$

We follow the same notations stated in Section 8.1.1 to represent polynomials and kernel calls. We can apply any division algorithm for computing q and r. Plain division is a simple method that we learn in high school. It is an iterative method. Algorithm 14 describes plain univariate division algorithm. As we can see, in each iteration of Algorithm 14, the polynomial a is changing. At the end, the last computed a is returned as the remainder. The step, where we update a in the algorithm (in line 7), is called the *division step*.

Algorithm 14: Division(a, b)

Input: a, b are univariate polynomial.
Output: q, r are the quotient and the remainder respectively.

```
1  n = deg(a);
2  m = deg(b);
3  q, r be two univarate polynomials initialized as zero polynomials.;
4  for (i = 0; i ≤ n − m; i = i + 1)) do
5      if a_{n−i} ≠ 0 then
6          w = a_{n−1−i}/b_{m−1};
7          a = a − b w;
8          q = q + w x^{n−m−i};
9  r = a;
10 return (q, r);
```

Euclidean algorithm is used to compute GCDs. In Algorithm 15, we describe a recursive version of it that calls Algorithm 14. Let g =EuclideanGCD(a, b), then both a and b are divisible by g.

Algorithm 15: EuclideanGCD(a, b)

Input: a, b: univariate polynomial.

```
1  if b = 0 then
2      return a;
3  [q, r] =Division(a, b);
4  return EuclideanGCD(b, r);
```

Algorithm 15 is a sequential algorithm. Indeed, there is no parallel version of this algorithm which would be both sublinear and work-efficient[1]. The best parallel version of the Euclidean Algorithm which is work-efficient, is that for systolic arrays,

[1]Here work-efficient refers to a parallel algorithm in the PRAM model for which the maximum number of processors in use times the span is in the same order as the work of the best serial counterpart algorithm.

a model of computation formalized by H. T. Kung and C. E. Leiserson [44], for which the span is linear [10].

Let, we have a systolic array that has two inputs that take coefficients from a and b in order (starting from leading coefficient). The idea of computing GCD is to place a number of such systolic arrays in series such that, each of it computes one division step. Each of the systolic array also need to store the degrees of the polynomials. For details, we refer [10].

Multiprocessors based on systolic arrays are not so common (as they are quite specialized to certain operations and difficult to build). However, the recent development of hardware acceleration technologies (GPUs, field-programmable gate arrays, etc.) has revitalized researchers interest in systolic algorithms and more generally in optimizing the use of computer resources for low-level algorithms [3, 21, 34].

Systolic array based gcd algorithm can be simulated on many-core machine. The role of one systolic array can be implemented by one multiprocessor. We need to establish synchronization rules among multiprocessors in such a way that a number of multiprocessors can work on different division steps concurrently. Alternatively, we can view the problem orthogonally. That is, instead of parallelizing a number of division steps, we can parallel each division step. We explain the latter technique in this chapter.

All operations except division step involved in computing plain division or Euclidean gcd are serial operations. The division steps are the only places where we can apply parallel operations.

9.2 Importance of plain division and Euclidean algorithm for polynomials with smaller degrees

Consider now the fundamental application of polynomial arithmetic: *solving systems of non-linear equations*. Many polynomial systems encountered in practice have finitely many solutions. Moreover, those systems that can be solved symbolically by computer algebra software, such as MAPLE or MATHEMATICA, have rarely more than 10,000 solutions, see for instance [73]. For this reason, the degrees of univariate polynomials that arise in practice rarely exceed 10,000.

It follows from the above discussion that implementation of a polynomial system solver on multi-cores or many-core machines require efficient parallel polynomial arithmetic in relative low degrees, that is, within degree ranges where FFT-based methods

may not apply due to performance issue. The study conducted in [12] show that univariate arithmetic based on parallel versions of the Algorithm of Karatsuba and its variants are not effective either in the desired degree ranges. This leads us to consider quadratic (or plain) algorithms for dense univariate division and Euclidean algorithm. That is, algorithms which run within $O(d^2)$ coefficient operations for polynomials of degree less than d, meanwhile FFT-based algorithms amount to $O(d\log(d)\log(\log(d)))$ coefficient operations, see the landmark textbook [20] for details.

9.3 Plain division on the GPU

In Section 9.3.1, we present a simple multithreaded algorithm computing (q, r) on an MMM machine. We call this algorithm *naive* since it is a direct implementation of an idea that, at each division step, each thread computes a coefficient of the next intermediate remainder. In Section 9.3.2, we propose a second MMM algorithm with a goal of minimizing overhead. We analyze both algorithms with the complexity measures defined in Section 3.2.3. In Section 9.3.3, we compare these two algorithms by means of Theorem 1. In Section 9.3.4, we compare our implementation of optimized plain division algorithm using CUDA with division function found in NTL.

9.3.1 Naive algorithm

Algorithm 16 repeatedly calls the kernel stated in Algorithm 17. The latter performs one division step in parallel. In a kernel, each thread computes one coefficient of an intermediate remainder polynomial by means of one multiplication and one subtraction in the coefficient field $\mathbb{K}$. Let ℓ be the number of threads in a thread block, we note that each kernel uses $\lceil \frac{m}{\ell} \rceil$ thread blocks. We observe that each thread of a kernel reads/writes 3 to 5 words[2] in the global memory without storing them in the local memory.

We also notice that Algorithm 16 performs exactly $n - m + 1$ consecutive calls to Algorithm 17. Nevertheless, Algorithm 17 works correctly even if, after one division step, the degree of an intermediate remainder drops by more than one. The implementation choice is relevant to dense polynomials, which are our primary interest. In the sparse case, the degree of an intermediate remainder needs to be computed after each division step.

[2]Indeed, in each block, the thread with ID 0 computes f; moreover, the first thread of the first block writes f to the global memory as the coefficient of degree $i - m$ of the quotient q.

We denote by W_{nai}, S_{nai} and O_{nai}, the work, span and overhead of Algorithm 16, respectively. Since each thread block performs $2\,\ell + 1$ arithmetic operations and each thread makes at most 5 accesses to the global memory, we obtain the following estimates, where δ stands for $n - m + 1$,

$$W_{\text{nai}} = \frac{\delta\, m\, (2\,\ell + 1)}{\ell},\ S_{\text{nai}} = 3\,\delta \quad \text{and} \quad O_{\text{nai}} = \frac{5\,\delta\, m\, U}{\ell}.$$

In order to apply Theorem 1, we shall compute the quantities $N(\mathcal{P})$, $L(\mathcal{P})$ and $C(\mathcal{P})$ defined in Section 3.2. We denote them here by N_{nai}, L_{nai} and C_{nai}, respectively. One can easily check that we have

$$N_{\text{nai}} = \frac{\delta\, m}{\ell},\ L_{\text{nai}} = \delta \quad \text{and} \quad C_{\text{nai}} = 3 + 5\,U.$$

In Figure 9.1, we show how one naive division step is done on GPU by our algorithm.

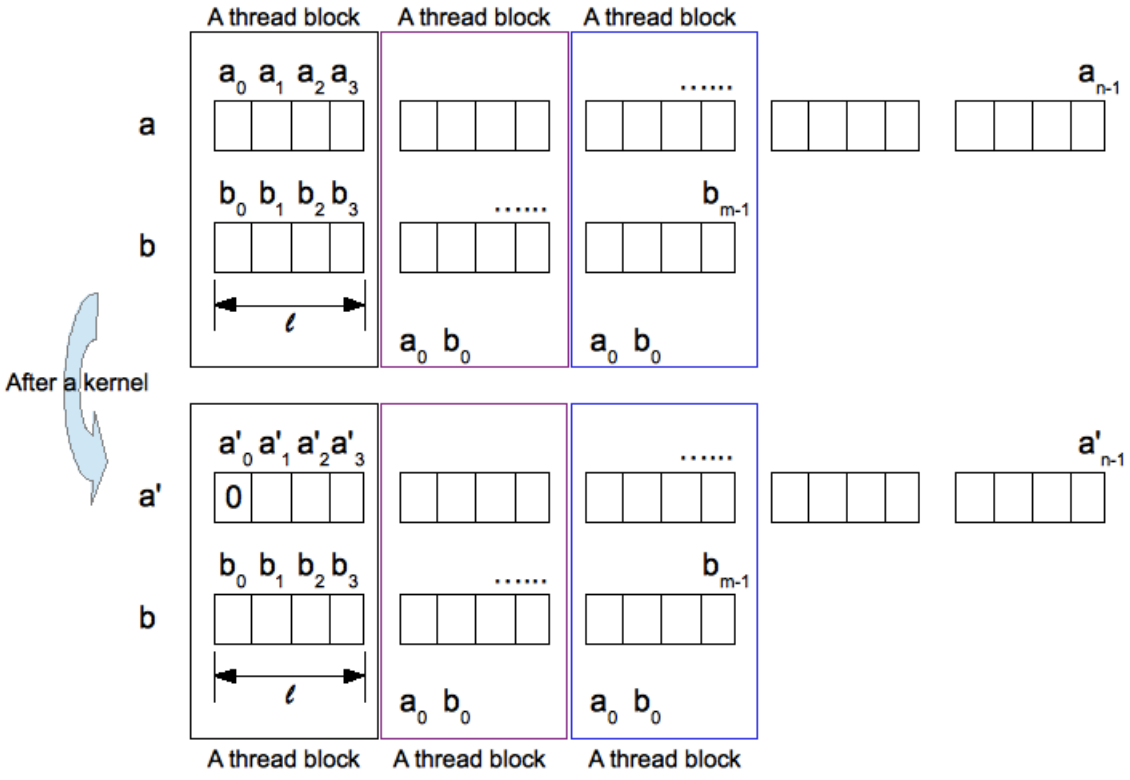

Figure 9.1: A naive division step.

Algorithm 16: NaivePlainDivisionGPU(a, b)

Input: $a, b \in \mathbb{K}[X]$ with $n - 1 = \deg(a) \geq m - 1 = deg(b)$.
Output: $q, r \in \mathbb{K}[X]$ s. t. $a = qb + r$ and $\deg(r) < m - 1$.

Let ℓ be the number of threads in a thread block ;
Let q be array of size $n - m + 1$ with coefficients in $\mathbb{K}$;
Let $c = \lceil m/\ell \rceil$ be the number of thread blocks;
for $(i = (n-1) \ldots (m-1))$ **do**
    NaiveDivKernel$\lll c, \ell \ggg (a, b, q, i, m-1)$;
if $a[0] == \cdots == a[m-1] == 0$ **then**
    return $[q, 0]$;
Compute d the maximum i such that $a[i] \neq 0$ holds ;
Let r be array of size $d + 1$ s.t. $r[i] = a[i]$ for $0 \leq i \leq d$;
return $[q, r]$;

Algorithm 17: NaiveDivKernel(a, b, q, i, d)

Input: $a, b, q \in \mathbb{K}[X]$, $d = \deg(b)$, $i \in \mathbb{N}$, $d \leq i$.

Let blockID, blockDim, threadID be the block id, number of threads per block, thread id respectively;
j =blockID blockDim +threadID;
if $j \leq d$ **then**
    if threadID$== 0$ **then**
        $f = b[d]^{-1}\, a[i]$;
        `/* accessing global memory. */`
        if $j == 0$ **then**
            $q[i - d] = f$;
            `/* writing to global memory. */`
    $a[j + i - d] = a[j + i - d] - b[j]\, f$;
    `/* updating` a `in global memory. */`

9.3.2 Optimized algorithm

Similarly to the scheme in Section 9.3.1, Algorithm 18 repeatedly calls a kernel (Algorithm 19). However, in the kernel, each thread updates a number of coefficients of an intermediate remainder polynomial repeatedly during a number of division steps, thus without synchronizing data among each other thread blocks. The motivation of the new scheme is to minimize the amount of data transferred between global and local memories so as to minimize the overhead.

To be more specific, given an integer $s \geq 1$, Algorithm 19 performs sufficiently many division steps (in fact, at most s) with polynomials a and b such that an output

intermediate remainder is either zero or its degree has been reduced at least by s. To this end, each thread block uses $3\,s$ threads and:

- loads from the coefficients of $X^d, X^{d-1}, \ldots, X^{d-s+1}$ from a (resp. b), that we call the *s-head* of a (resp. b), where d the degree of a (resp. b), see Lines 6-7,
- loads $2\,s$ (resp. $3\,s$) consecutive coefficients of a (resp. b), say $X^e, X^{e-1}, \ldots, X^{e-2s+1}$ ($X^f, X^{f-1}, \ldots, X^{f-3s+1}$) for some positive integer e (resp. f) which depends on the thread and thread-block IDs, see Lines 8-9.

The s-heads of a and b are used to keep track of the leading coefficient of an intermediate remainder during the entire execution of a kernel, see Lines 11-12 and 18-19 of Algorithm 19. This task is achieved by the first s threads of a thread-block and requires $s + (s-1) + \cdots + 1 = \frac{s(s+1)}{2}$ arithmetic operations. Meanwhile, the other $2\,s$ threads update $2s$ coefficients of a, see Lines 20-21, which amounts to $2\,s \cdot 2\,s$ arithmetic operations. Finally, at Lines 16-17 (resp. 22-23) the quotient q (resp. the intermediate remainder a) is updated in the global memory.

We denote the work, span and overhead of the optimized algorithm by W_{opt}, S_{opt} and O_{opt}, respectively. Since each thread requests at most 9 accesses to the global memory, we obtain the following estimates, where δ stands for $n - m + 1$,

$$W_{\text{opt}} = \frac{\delta\, m\, (9\, s + 1)}{4\, s},\ S_{\text{opt}} = 3\,\delta \ \text{ and } \ O_{\text{opt}} = \frac{9\,\delta\, m\, U}{2\, s^2}. \tag{9.2}$$

In order to apply Theorem 1, we shall compute the quantities $N(\mathcal{P})$, $L(\mathcal{P})$ and $C(\mathcal{P})$ defined in Section 3.2. We denote them here by N_{opt}, L_{opt} and C_{opt}, respectively. One can easily check that we have

$$N_{\text{opt}} = \frac{\delta\, m}{2\, s^2},\ L_{\text{opt}} = \frac{\delta}{s} \ \text{ and } \ C_{\text{opt}} = 3\, s + 9\, U. \tag{9.3}$$

In Figure 9.2, we show how optimized division steps is done on GPU by our algorithm.

9.3.3 Comparison of running time estimates

Following the strategy stated in the introduction, we first compare the overheads of the two algorithms. The ratio $O_{\text{nai}}/O_{\text{opt}}$ is $\frac{10}{9}\,\frac{s^2}{l}$. Since we have $2\ell \leq Z$ and $7s \leq Z$, this suggests to replace s and ℓ by $Z/7$ and $Z/2$, respectively in the overhead ratio, leading to

$$\frac{O_{\text{nai}}}{O_{\text{opt}}} = \frac{20}{441}\, Z. \tag{9.4}$$

Algorithm 18: OptimizePlainDivisionGPU(a, b, s)

Input: $a, b \in \mathbb{K}[X]$ with $n-1 = \deg(a) \geq m-1 = deg(b)$ and $s \in \mathbb{N}$.
Output: $q, r \in \mathbb{K}[X]$ s. t. $a = qb + r$ and $\deg(r) < m-1$.

Let $\ell = 3s$ be the number of threads in a thread block ;
Let $c = \lceil m/(2s) \rceil$ be the number of thread blocks;
Let q be array of size $n - m + 1$ with coefficients in $\mathbb{K}$;
for $(i = n-1; i \geq m-1; i = i-s)$ **do**
    OptDivKer$\lll c, \ell \ggg (a, b, q, i, m-1, s)$;
if $a[0] == \cdots == a[m-1] == 0$ **then**
    return $[q, 0]$;
Compute d the maximum i such that $a[i] \neq 0$ holds ;
Let r be array of size $d+1$ s.t. $r[i] = a[i]$ for $0 \leq i \leq d$;
return $[q, r]$;

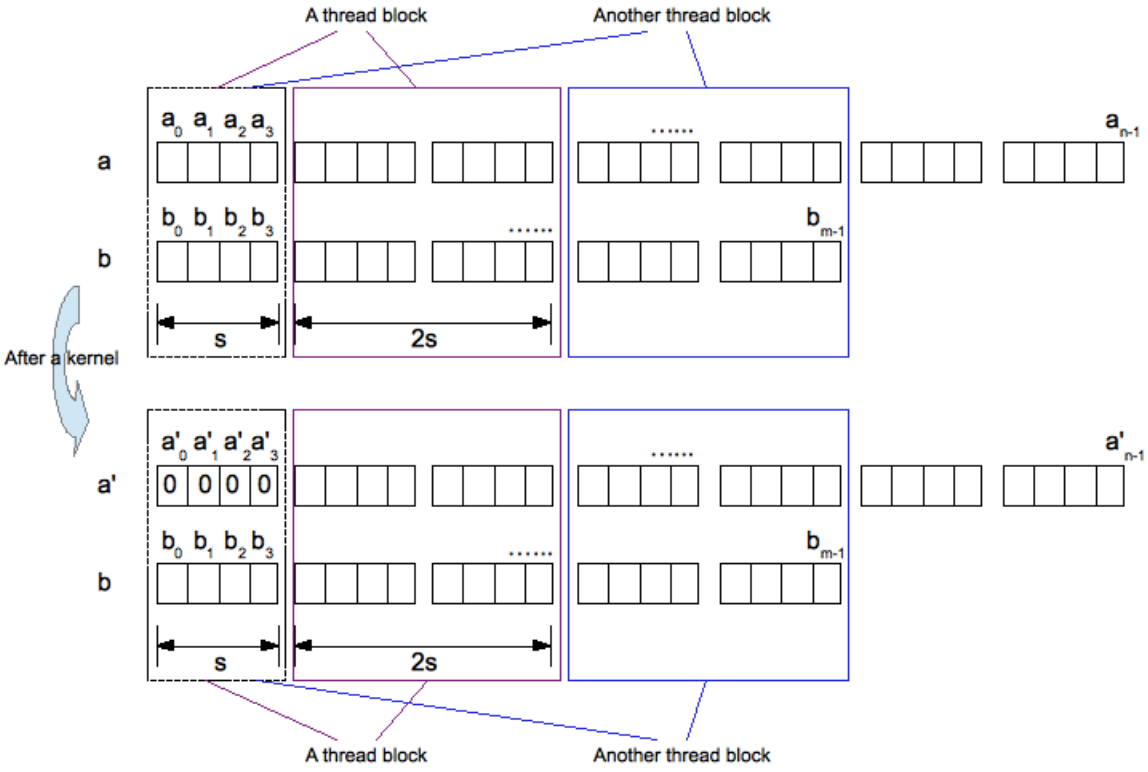

Figure 9.2: Optimize division steps.

Next, we observe that this substantial improvement is done at a fairly low expense in terms of work overhead. Indeed, the ratio $W_{\text{nai}}/W_{\text{opt}}$ is asymptotically constant:

$$\frac{W_{\text{nai}}}{W_{\text{opt}}} = \frac{8\,(Z+1)}{9\,Z+7}. \tag{9.5}$$

Algorithm 19: OptDivKer(a, b, q, i, d, s)

Input: $a, b, q \in \mathbb{K}[X]$, $i \in \mathbb{N}$, $d = \deg(b)$ and $s \in \mathbb{N}$.

```
1  f = b[d]^{-1};
2  Let sAc, sBc, sA, sB be local arrays of size s, s, 2s, 3s respectively each with
   coefficients in K;
3  j =blockID blockDim +threadID; t =threadID;
4  if t < s then
5      sAc[t] = a[i − t]; sBc[t] = b[d − t]; sB[t] = b[d − 2s blockID−t];
6  if t ≥ s then
7      sA[t − s] = a[i − s − 2s blockID−t]; sB[t] = b[d − 2s blockID−t];
   /* Reading from global memory.                                          */
8  for (w = 0; (w < s) ∧ (i + w ≥ d); w = w + 1) do
9      while (w < s) ∧ ( sAc[w] = 0) do
10         w = w + 1;
11     if w ≥ s then
12         break;
13     v =sAc[w] f;
14     if j == 0 then
15         q[i − d − w] = v;
       /* Writing q to global memory.                                      */
16     if w ≤ t < s then
17         sAc[t] = sAc[t]−sBc[t − w] v;
18     if t ≥ s then
19         sA[t − s] = sA[t − s]− sB[t − w] v;
20 if t ≥ s then
21     a[i − s − 2s blockID−t] = sA[t − s];
   /* Writing back a to global memory.                                     */
```

Applying Theorem 1, the running times on p SMs of the naive and optimized algorithms are bounded over by

$$(N_{\text{nai}}/p + L_{\text{nai}}) \cdot C_{\text{nai}} \quad \text{and} \quad (N_{\text{opt}}/p + L_{\text{opt}}) \cdot C_{\text{opt}}. \tag{9.6}$$

After algebraic simplifications, the ratio R of the former by the latter becomes

$$\frac{2}{3} \frac{(3 + 5\,U)\,(m + \ell\,p)\,s^2}{\ell\ (s + 3\,U)\,(m + 2\,s\,p)}. \tag{9.7}$$

After replacing s and ℓ, we obtain

$$\frac{2}{3}\,\frac{(3+5\,U)\,(2\,m+Z\,p)\,Z}{(Z+21\,U)\,(7\,m+2\,Z\,p)}. \tag{9.8}$$

When m escapes to infinity, the ratio R is equivalent to

$$\frac{4}{21}\,\frac{(3+5\,U)\,Z}{Z+21\,U}. \tag{9.9}$$

We observe that this ratio is larger than 1 if and only if $Z > \frac{441\,U}{20\,U-9}$ holds. A natural technological evolution for a many-core machine is to have Z increasing and U decreasing. Thus, the above condition is expected to hold at some point and, in this case, the optimized algorithm is overall better than the naive one.

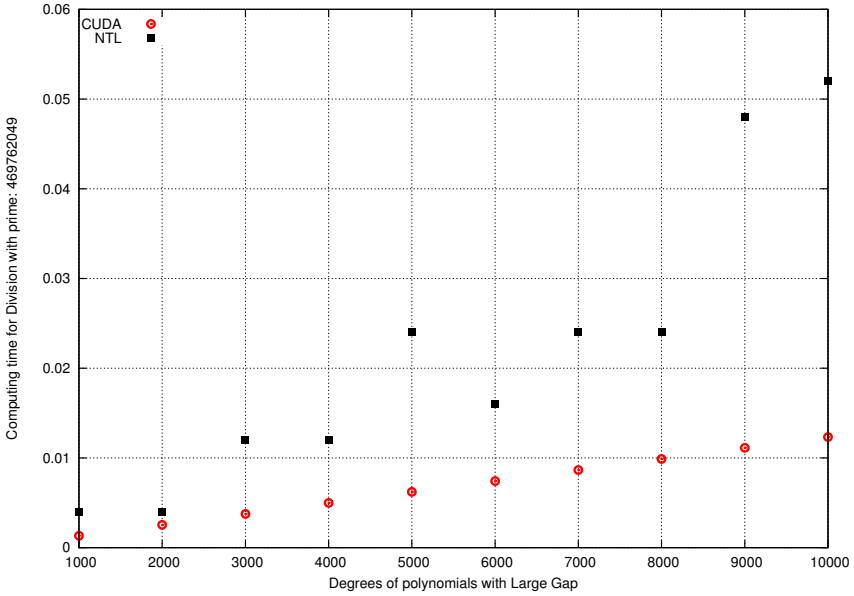

Figure 9.3: Comparison between parallel plain division on CUDA and fast division in NTL for univariate polynomials with large degree gap.

9.3.4 Experimental results of our optimized univariate division on GPU

We have compared the running time of our CUDA implementation of the plain division with the serial C implementation of the fast division (see Chapter 9 in [20]) from the NTL library. The latter algorithm is based on FFT techniques and its work fits within $O(d\log(d)\log(\log(d)))$ coefficient operations, with $d = \max(m, n)$. The input

polynomials used in our experimentation are dense random (univariate) polynomials with coefficients in a finite field whose characteristic is a machine word prime. We use the following primes: 7, 9001 and 469762049. Our GPU code does not depend on the prime while NTL uses different algorithms depending on the prime. For a given degree pattern, the NTL running time varies at most by a factor of 2 from one of our primes to another. The degrees of our input polynomials satisfy $n = 2m$. The running time of our CUDA code outperforms that of NTL by a factor from 3 to 5, for $1,000 \leq n \leq 10,000$, see Figure 9.3.

9.4 Euclidean algorithm on GPU

In Section 9.4.1, we present a simple multi-threaded algorithm that computes $\mathsf{GCD}(a, b)$ for an MMM machine. We call it naive (like Algorithm 16) as this algorithm also performs one division step within one kernel. In Section 9.4.2, we describe another algorithm for an MMM machine that reduces the overhead of data transfer. Finally, in Section 9.4.3, we compare those two algorithms by means of Theorem 1. We also provide the comparison of running time between our optimized implementation of GCD with the corresponding function found in NTL in Section 9.4.4.

9.4.1 Naive algorithm

Algorithm 20 computes $\mathsf{GCD}(a, b)$ in a naive way. Like naive division Algorithm 16, it calls a kernel given in Algorithm 21 that completes one division step. The former algorithm calls the latter one at most $n + m - 2$ times. Let ℓ be the number of threads in a thread block. st is an array of length 2 that stores the degree of a and b. Algorithm 21 is same as Algorithm 17 except it checks the current degree of both a and b to decide which polynomial takes the role as a divisor, and then it completes a division step. Each thread also reads/writes 3 to 5 words in the global memory without storing then in the local memory, and it computes one coefficient of an intermediate remainder. So, the number of active threads in a division step depends on the degree of the divisor polynomial. The active thread, whose rank is maximum, updates st array at the end of the division step correctly. We consider the fact that the degree of an intermediate remainder may be dropped by more than 1. Algorithm 17 works correctly even if the GCD is computed before $n + m - 2$ kernels. After $n + m - 2$ division steps, either polynomial a or b becomes zero polynomial or constant. Algorithm 20 returns the other polynomial as GCD.

We denote by W_{nai}, S_{nai} and O_{nai}, the work, span and overhead of Algorithm 20, respectively. Each thread block performs $2\,\ell + 1$ arithmetic operations, and each thread requests at most 5 accesses to the global memory. We obtain the following estimates, where δ stands for $m + n - 2$ and μ stands for $n + \ell + 1$,

$$W_{\text{nai}} = \frac{m\,(2\,n\,\ell + \mu - 2)}{\ell},\ S_{\text{nai}} = 3\,\delta \ \text{ and } \ O_{\text{nai}} = \frac{5\,m\,U\,\mu}{\ell}.$$

In order to apply Theorem 1, we shall compute the quantities $N(\mathcal{P})$, $L(\mathcal{P})$ and $C(\mathcal{P})$ defined in Section 3.2. We denote them by N_{nai}, L_{nai} and C_{nai}, respectively. One can easily check that we have

$$N_{\text{nai}} = \frac{m\,\mu}{\ell},\ L_{\text{nai}} = \delta \ \text{ and } \ C_{\text{nai}} = 3 + 5\,U.$$

Algorithm 20: NaivePlainGcdGPU(a, b)

Input: $a, b \in \mathbb{K}[X]$ with $n - 1 := \deg(a \geq m - 1 := deg(b)$.
Output: $g \in \mathbb{K}[X]$, s.t. $g =$GCD(a, b).

```
1  int st[] = {deg(a), deg(b)};
2  Let ℓ be the number of threads in a thread block ;
3  Let c = ⌈m)/ℓ⌉ be the number of thread blocks;
4  for (i = 0; i < n + m − 2; i = i + 1) do
5      NaivePlainGcdKernel<<< c, ℓ >>> (a, b, st);
6  if a is a zero or constant polynomial then
7      Compute d the maximum i s.t. b[i] ≠ 0 holds ;
8      Let g be array of size d + 1 with coefficients in K s.t. g[i] = b[i] for
       0 ≤ i ≤ d ;
9  else
10     Compute d the maximum i s.t. a[i] ≠ 0 holds ;
11     Let g be array of size d + 1 with coefficients in K s.t. g[i] = a[i] for
       0 ≤ i ≤ d ;
12 return g;
```

9.4.2 Optimized algorithm

Algorithm 22 is our top level optimize algorithm for computing GCD of a and b. Given a positive integer s, the algorithm calls a kernel iteratively with $\lceil \frac{\deg(b)+1}{s} \rceil$ thread blocks, each of them uses $3\,s$ threads. In Algorithm 23, we present a kernel. Threads in a thread block collectively compute at most s division steps and update

Algorithm 21: NaivePlainGcdKernel(a, b, st)

Input: $a, b \in \mathbb{K}[X]$ and st[] stores the current degree of a and b.

```
1  j =blockID blockDim +threadID; t =threadID;
2  if st[0] ≥ st[1] > 0 ∧ j < st[1] then
3      f = a[ st[0]] b[ st[1]]^{-1}; w = j+st[0]−st[1] ;
4      a[w] = a[w] − b[j] f ;
5      if j ==st[1] − 1 then
6          while (st[0] ≥ 0) ∧ (a[st[0]] = 0) do
7              st[0] = st[0] − 1;
8  else if 0 < st[0] < st[1] ∧ j < st[0] then
9      f = b[ st[1]] a[ st[0]]^{-1}; w = j+st[1]−st[0] ;
10     b[w] = b[w] − a[j] f;
11     if j == st[0] − 1 then
12         while (st[1] ≥ 0) ∧ (b[st[1]] = 0) do
13             st[1] = st[1] − 1;
```

s coefficients from both a and b. After a division step, the degree of the dividend polynomial is decreased by at least one, and then in the next division step, coefficients from the divisor polynomial are adjusted by one or more shift operations. Thus, we need $2\,s$ coefficients from both a and b to be sure that after s division steps we have s coefficients from both a and b correctly. The consecutive thread blocks has s common coefficients from both a and b. That is why the required number of thread blocks is $\min(\frac{\deg(a)+1}{s}, \frac{\deg(b)+1}{s})$. A thread block also needs *s-head* (like Algorithm 19) from both a and b to execute s division steps.

Each thread block, i copies *s-head* coefficients from both a and b. It also copies $2\,s$ other coefficients from both a (b), say $X^{d-i\,s}, X^{d-i\,s-1}, \cdots, X^{d-i\,s-2s}$, where d is the degree of a (resp. b). The first s threads in a thread block completes s division steps with respect to these *s-head*. u and v keeping track of current leading coefficients of a and b respectively. So, in a division step, some threads out of these s threads are not active. These s threads do not need to write back any coefficient. The purpose of those coefficients and those threads are to broadcast the leading coefficient of a and b to other threads only. Once the first s threads compute the current leading coefficient of both a and b, the other $2s$ threads complete a division step for the other coefficients. After s division steps each thread block can write back s coefficients correctly to the global memory.

We denote by W_{opt}, S_{opt} and O_{opt}, the work, span and overhead respectively. Each thread block performs $s + 2\,(s-1) + \cdots + 2\,\frac{s}{2} = \frac{27}{4}\,s^2 + \frac{13}{2}\,s$ arithmetic operations

Algorithm 22: OptimizedPlainGcdGPU(a, b, s)

Input: $a, b \in \mathbb{K}[X]$ with $n - 1 = \deg(a) \geq m - 1 = deg(b)$ and an integer $s > 1$.

Output: $g \in \mathbb{K}[X]$, s.t. $g = \mathsf{GCD}(a, b)$.

```
1  int st[2] = {deg(a), deg(b)};
2  Let ℓ = 3s be the number of threads in a thread block ;
3  Let c = ⌈m)/s⌉ be the number of thread blocks;
4  for (i = 0; i < n + m − 2; i = i + s) do
5  |  OptGcdKer<<< c, ℓ >>> (a, b, s, st) ;
6  if a is a zero or constant polynomial then
7  |  Compute d the maximum i s.t. b[i] ≠ 0 holds ;
8  |  Let g be array of size d + 1 with coefficients in K s.t. g[i] = b[i] for
   |  0 ≤ i ≤ d ;
9  else
10 |  Compute d the maximum i s.t. a[i] ≠ 0 holds ;
11 |  Let g be array of size d + 1 with coefficients in K s.t. g[i] = a[i] for
   |  0 ≤ i ≤ d ;
12 return g;
```

regard to the first s threads. Each thread requests at most 8 accesses to the global memory. We obtain the following estimates, where μ stands for $\frac{345}{16}\,s^2 + \frac{77}{4}\,s$ and ν stands for $\frac{9}{4} + \frac{6}{s}$,

$$\begin{gathered} W_{\text{opt}} = \nu\, m^2 + \left(\frac{9}{2}\,n + \frac{n}{2\,s} + \frac{87}{8}\,s + \frac{23}{2}\right) m - \mu, \\ S_{\text{opt}} = 3\,n + 3\,m \quad \text{and} \quad O_{\text{opt}} = \frac{8\,m\,U\,(n+s)}{s^2}. \end{gathered} \tag{9.10}$$

In order to apply Theorem 1, we shall compute the quantities $N(\mathcal{P})$, $L(\mathcal{P})$ and $C(\mathcal{P})$ defined in Section 3.2. We denote them here by N_{opt}, L_{opt} and C_{opt}, respectively. One can easily check that we have

$$N_{\text{opt}} = \frac{m\,n}{s^2} + \frac{m}{s},\; L_{\text{opt}} = \frac{n}{s} + \frac{m}{s} \quad \text{and} \quad C_{\text{opt}} = 3\,s + 8\,U. \tag{9.11}$$

Algorithm 23: OptGcdKer(a, b, s st)

Input: $a, b \in \mathbb{K}[X]$, an integer $s > 1$ and st[] stores the current degree of a and b.

```
1  Let sAc, sBc, sA, sB be local arrays of size s, s, 2s, 2s respectively with
   coefficients in K;
2  local integers u = v = x = y = 0, f, n = st[0], m = st[1];
3  j =blockID blockDim +threadID; t =threadID;
4  if t < s then
5      sAc[t] = a[n − t];   sBc[t] = b[m − t];
6  if t ≥ s then
7      sA[t − s] = a[n − s blockID−t]   sB[t − s] = b[m − s blockID−t];
   /* copying from global memory.                                          */
8  for (w = 0; w < s; w = w + 1) do
9      if (n ≥ m ∧ m ≥ 0) then
10         if t == 0 then
11             f =sAc[u] sBc[v]^{−1};
12         if (u + t < s) ∧ (v + t < s) then
13             sAc[u + t] =sAc[u + t]−sBc[v + t] f;
14         if (u + t ≥ s) ∧ (v + t ≥ s) then
15             sA[x + t − s] =sA[x + t − s]− sB[y + t − s] f;
16         if t == 0 then
17             while sAc[u] = 0 do
18                 u = u + 1;   x = x + 1;   n = n − 1;
19     if (m ≥ n) ∧ (n ≥ 0) then
20         if t == 0 then
21             f = sBc[v] sAc[u]^{−1};
22         if (u + t < s) ∧ (v + t < s) then
23             sBc[v + t] = sBc[v + t]− sAc[u + t]f ;
24         if (u + t ≥ s) ∧ (v + t ≥ s) then
25             sB[y + t − s] = sB[y + t − s]− sA[x + t − s] f;
26         if t == 0 then
27             while sBc[v] = 0 do
28                 v = v + 1;   y = y + 1;   m = m − 1;
29 if t ≥ s then
30     a[ st[0] − s blockID−t] = sA[t − s];
31     b[ st[1] − s blockID−t] = sB[t − s] ;
       /* writing to global memory.                                        */
32 if j == min(st[0], st[1]) then
33     Update st array with the new degree of a and b;
```

n	m	Optimize GCD on CUDA with s = 512	Naive GCD on CUDA
1000	500	0.010	0.024
2000	1500	0.024	0.058
3000	2500	0.039	0.108
4000	3500	0.053	0.158
5000	4500	0.069	0.203
6000	5000	0.056	0.235
7000	6000	0.066	0.282
8000	7000	0.076	0.324
9000	8000	0.087	0.367
10000	9000	0.097	0.411

Table 9.1: GCD implementation on CUDA with two different values of s.

9.4.3 Comparison of running time estimates

We first compare the overheads of the two algorithms. Since we have $2\ell \leq Z$ and $6s \leq Z$, we replace m, s and ℓ by n, $Z/6$ and $Z/2$, respectively in the overhead ratio,

$$\frac{O_{\text{nai}}}{O_{\text{opt}}} = \frac{5}{48} \, \frac{Z\,(2\,n + 2 + Z)}{6\,n + Z}. \tag{9.12}$$

Next, we also observe the ratio $W_{\text{nai}}/W_{\text{opt}}$ is asymptotically constant, since we have, where μ stands for $115\,Z^3 + 616\,Z^2$,

$$\frac{W_{\text{nai}}}{W_{\text{opt}}} = \frac{(284\,Z + 2)\,n^2 + (Z - 2)\,n}{(1296\,Z + 7488)\,n^2 + (348\,Z^2 + 2208\,Z)\,n - \mu}. \tag{9.13}$$

Thus, the improvement has a fairly low expense. Applying Theorem 1, the running times on p SMs of the naive and optimized algorithms are bounded over by

$$(N_{\text{nai}}/p + L_{\text{nai}}) \cdot C_{\text{nai}} \quad \text{and} \quad (N_{\text{opt}}/p + L_{\text{opt}}) \cdot C_{\text{opt}}. \tag{9.14}$$

When n escapes to infinity, the ratio R is equivalent to

$$\frac{(3 + 5\,U)\,Z}{9\,(Z + 16\,U)}. \tag{9.15}$$

We observe that this ratio is larger than 1 if and only if $Z > \frac{144\,U}{5\,U - 6}$ holds. Thus, the condition is expected to hold and, in this case, the optimized algorithm is overall than the naive one.

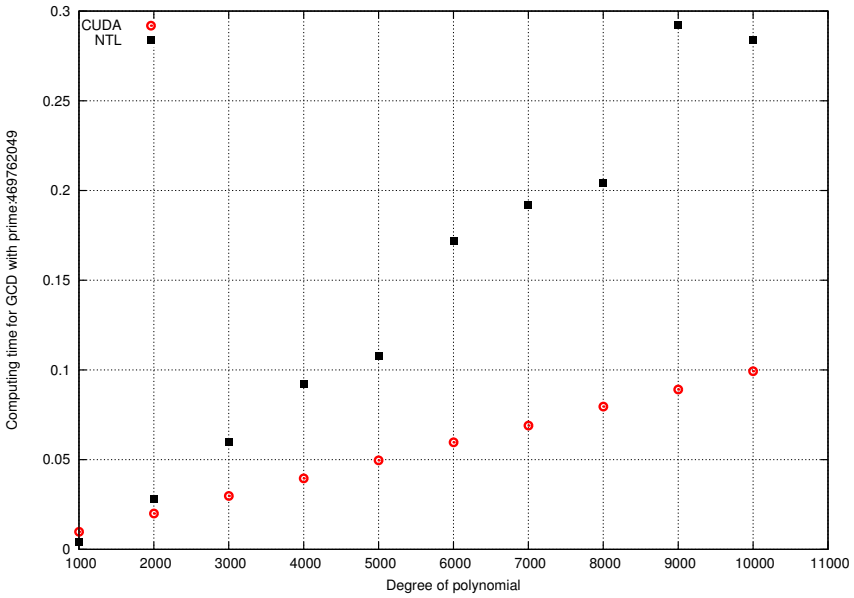

Figure 9.4: Comparison between parallel GCD on CUDA and FFT-based GCD in NTL for univariate polynomials, with the same degree ($n = m$).

9.4.4 Experimental results of our optimized Euclidean algorithm on GPU

We have compared the running time of our CUDA implementation of the Euclidean Algorithm with the serial C implementation of the Half-GCD algorithm (see Chapter 11 in [20]) from the NTL library. The latter algorithm is based on FFT techniques and its work fits within $O(d\log(d)\log(\log(d)))$ coefficient operations, whale that of the former algorithm amounts to $O(d^2)$ coefficient operations, for input polynomials of degree d.

As for the experimentation with the plain division, the input polynomials used in our experimentation are dense random (univariate) polynomials with coefficients in a finite field whose characteristic is a machine word prime. Here again, we use the primes 7, 9001, 469762049 and we observe that our GPU code does not depend on the prime while NTL uses different algorithms depending on the prime. For a given degree pattern, the NTL running time varies at most by a factor of 2 from one of our primes to another.

Figure 9.4 correspond to 469762049. Indeed, we are interested in large primes since they support modular methods for polynomial system solving [73].

As reported by Figure 9.4, our implementation is almost three times faster than that of NTL for polynomials whose degrees range between 1,000 and 10,000. Recall

that this degree range is also what is of interest for the same purpose of polynomial system solving.

The technique of computing s leading coefficients in every thread block, implemented in both division and GCD algorithm, has two major advantages. First, it reduces the number of kernel calls by a factor of s. Second, it reduces the amount of memory transfer between the global and local memory by a factor of s.

In Table 9.1, we compare the computation time between two versions of our CUDA implementation of the Euclidean Algorithm. The first one sets $s = 512$ and the other one does not use at all this technique of computing s leading coefficients in every thread block. In this latter implementation, leading coefficients are kept up-to-date in the global memory such that they can be accessed by every thread block. Thus, in this scheme, every thread block works on a single division step between two updates of the leading coefficient in the global memory.

As mentioned above, the former implementation increases the work but reduces parallelization overheads in a significant manner. Table 9.1, shows that the former method outperforms the latter by a speedup factor varying from 2 to 4.

9.5 Conclusion

Motivated by the implementation of polynomial system solvers over finite fields, we were lead to parallelize plain univariate polynomial arithmetic on GPUs. For the degree range $2^{10} \cdots 2^{18}$, our GPU code for computing polynomial GCDs via the Euclidean Algorithm runs in linear time w.r.t the maximum degree of the input polynomials. Such sizes are sufficient for many applications.

We observed that controlling parallelization overheads (synchronization on data via global memory, number of kernel calls, etc.) was essential for reaching peak performance in our implementation.

Chapter 10

Evaluation and Interpolation of Univariate Polynomial by Subproduct Tree Technique on GPU

We propose parallel algorithms for operations on univariate polynomials (multi-point evaluation, interpolation) based on subproduct tree techniques. We target implementation on many-core GPUs. On those architectures, we demonstrate the importance of adaptive algorithms, in particular the combination of parallel plain arithmetic and parallel FFT-based arithmetic. Experimental results illustrate the benefits of our algorithms.

This chapter is a joint work with F. Mansouri and M. Moreno Maza.

10.1 Introduction

We investigate the use of Graphics Processing Units (GPUs) in the problems of evaluating and interpolating polynomials. Many-core GPU architectures were considered in [67] and [70] in the case of numerical computations, with the purpose of obtaining better support, in terms of accuracy and running times, for the development of polynomial system solvers.

Our motivation, in this work, is also to improve the performance of polynomial system solvers. However, we are targeting symbolic, thus exact, computations. In particular, we aim at providing GPU support for solvers of polynomial systems with

coefficients in finite fields, such as the one reported in [55]. This case handles as well problems from cryptography and serves as a base case for the so-called modular methods [73], since those methods reduce computations with rational number coefficients to computations with finite field coefficients.

Finite fields allow the use of asymptotically fast algorithms for polynomial arithmetic, based on Fast Fourier Transforms (FFTs) or, more generally, subproduct tree techniques. Chapter 10 in the landmark book [20] is an overview of those techniques, which have the advantage of providing a more general setting than FFTs. More precisely, evaluation points do not need to be successive powers of a primitive root of unity. Evaluation and interpolation based on subproduct tree techniques have "essentially" (i.e. up to log factors) the same algebraic complexity as their FFT-based counterparts. However, their implementation is known to be challenging.

In this chapter, we report on the first GPU implementation (using CUDA [58]) of subproduct tree techniques for multi-point evaluation and interpolation of univariate polynomials. The parallelization of those techniques raise the following challenges.

1. The divide-and-conquer formulation of operations on subproduct-trees is not sufficient to provide enough parallelism and one must also parallelize the underlying polynomial arithmetic operations, in particular polynomial multiplication.

2. Algorithms based on FFT (such as subproduct tree techniques) are memory bound since the ratio of work to memory access is essentially constant, which makes those algorithms not well suited for multi-core architectures.

3. During the course of the execution of a subproduct tree operation (construction, evaluation, interpolation) the degrees of the involved polynomials vary greatly, thus so does the work load of the tasks, which makes those algorithms complex to implement on many-core GPUs.

The contributions of this work are summarized below. We propose parallel algorithms for performing subproduct tree construction, evaluation and interpolation. We also report on their implementation on many-core GPUs. See Sections 10.3, 10.5 and 10.6, respectively. We enhance the traditional algorithms for polynomial evaluation and interpolation based on subproduct tree techniques, by introducing the notion of a *subinverse tree*, which we use to accelerate both evaluation and interpolation, see Section 10.4. For subproduct tree operations targeting many-core GPUs, we demonstrate the importance of *adaptive algorithms*. That is, algorithms that adapt their behavior to the available computing resources. In particular, we combine *parallel*

plain arithmetic and *parallel fast arithmetic*. For the former we rely on [31] and, for the latter we extend the work of [53]. The span and overhead of our algorithm are measured considering *many-core machine model* stated in Chapter 3. To evaluate our implementation, we measure the effective memory bandwidth of our GPU code for parallel multi-point evaluation and interpolation On a card with a theoretical maximum memory bandwidth of 148 GB/S, our code reaches peaks at 64 GB/S. Since the arithmetic intensity of our algorithms is also high, we believe that this is a promising result.

All implementation of subproduct tree techniques that we are aware of are serial code only. This includes [8] for $GF(2)[x]$, the FLINT library[33] and the `Modpn` library [46]. Hence we compare our code against probably the best serial C code (namely the FLINT library) for the same operations. On sufficiently large input data and on NVIDIA Tesla C2050, our code outperforms its serial counterpart by a factor ranging between 20 to 30. Experimental data are provided in Section 10.7.

10.2 Background

We review various notions related to subproduct tree techniques. See Chapter 10 in [20] for details. We also specify costs for the underlying polynomial arithmetic used in our implementation. Notations and hypotheses introduced in this section are used throughout this chapter. Let $n = 2^k$ for some positive integer k and let $\mathbb{K}$ be a finite field. Let $u_0, \ldots, u_{n-1} \in \mathbb{K}$. Define $m_i = x - u_i$, for $0 \leq i < n$. We assume that each $u_i \in \mathbb{K}$ can be stored in one machine word.

Subproduct trees. The subproduct tree $M_n := \mathsf{SubproductTree}(u_0, \ldots, u_{n-1})$ is a complete binary tree of height $k = \log_2 n$. The j-th node of the i-th level of M_n is denoted by $M_{i,j}$, where $0 \leq i \leq k$ and $0 \leq j < 2^{k-i}$, and is defined as followed:

$$M_{i,j} = m_{j \cdot 2^i} \cdot m_{j \cdot 2^i + 1} \cdots m_{j \cdot 2^i + (2^i - 1)} = \prod_{0 \leq \ell < 2^i} m_{j \cdot 2^i + \ell}.$$

Note that each of $M_{i,j}$ can be defined recursively as follows.

$$M_{0,j} = m_j \quad \text{and} \quad M_{i+1,j} = M_{i,2j} \cdot M_{i,2j+1}.$$

Observe that the i-th level of M_n has 2^{k-i} polynomials with degree of 2^i. If each element of $\mathbb{K}$ fits within a machine word, then storing the subproduct tree M_n requires at most $n \log_2 n + 3n - 1$ words.

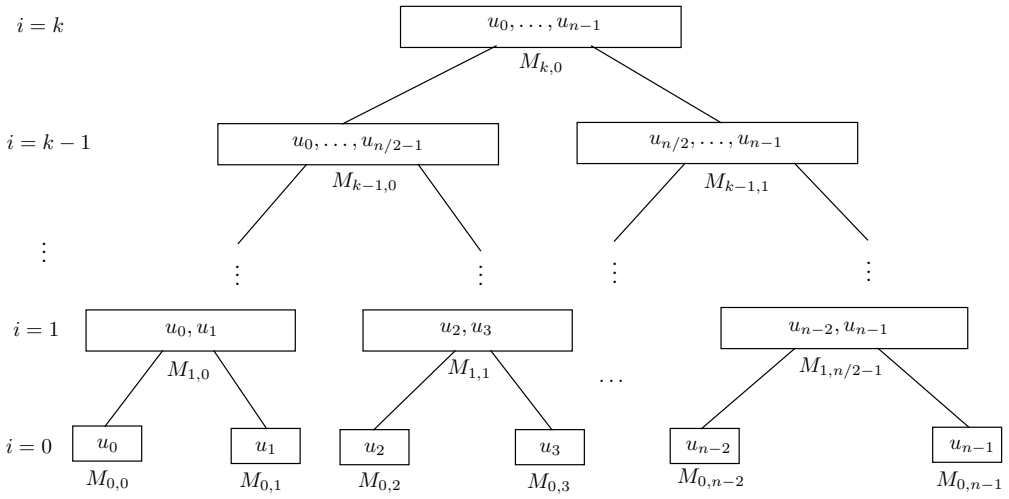

Figure 10.1: Subproduct tree associated with the point set $U = \{u_0, \ldots, u_{n-1}\}$.

Let us split the point set $U = \{u_0, \ldots, u_{n-1}\}$ into two halves of equal cardinality and proceed recursively with each half until it becomes a singleton. This leads to a binary tree of depth $\log_2 n$ having the points $u_0, \ldots, u_{n-1}$ as leaves, depicted on Figure 10.1. Note that the j-th node from the left at level i is labeled by $M_{i,j}$. Algorithm 24 generates the polynomials $M_{i,j}$ in an efficient manner, discussed in Section 10.3.

Algorithm 24: SubproductTree$(m_0, \ldots, m_{n-1})$

Input: $m_0 = (x - u_0), \ldots, m_{n-1} = (x - u_{n-1}) \in \mathbb{K}[x]$ with $u_0, \ldots, u_{n-1} \in \mathbb{K}$ and $n = 2^k$ for $k \in \mathbb{N}$.

Output: The subproduct-tree M_n, that is, the polynomials $M_{i,j} = \prod_{0 \le \ell < 2^i} m_{j \cdot 2^i + \ell}$ for $0 \le i \le k$ and $0 \le j < 2^{k-i}$.

```
for j = 0 to n − 1 do
    M_{0,j} = m_j;
for i = 1 to k do
    for j = 0 to 2^{k−i} − 1 do
        M_{i,j} = M_{i−1,2j} M_{i−1,2j+1};
return M_n;
```

Multi-point evaluation and interpolation. Given a univariate polynomial $f \in \mathbb{K}[x]$ of degree less than n, we define $\chi(f) = (f(u_0), \ldots, f(u_{n-1}))$. The map χ is called the *multi-point evaluation map* at $u_0, \ldots, u_{n-1}$. Define $m = \prod_{0 \le i < n}(x - u_i)$.

When $u_0, \ldots, u_{n-1}$ are pairwise distinct, then

$$\chi : \begin{array}{ccc} \mathbb{K}[x]/\langle m \rangle & \longrightarrow & \mathbb{K}^n \\ f & \longmapsto & (f(u_0), \ldots, f(u_{n-1})) \end{array}$$

realizes an isomorphism of $\mathbb{K}$-vector spaces. The inverse map χ^{-1} can be computed via Lagrange interpolation. Given distinct points $(v_0, \ldots, v_{n-1}) \in \mathbb{K}$, the unique polynomial $f \in \mathbb{K}[x]$ of degree less than n which takes the value v_i at the point u_i for all $0 \leq i < n$ is: $f = \sum_{i=0}^{n-1} v_i s_i m/(x - u_i)$ where $s_i = \prod_{i \neq j,\, 0 \leq j < n} 1/(u_i - u_j)$ and $m = \prod_{0 \leq i < n}(x - u_i)$.

We observe that $\mathbb{K}[x]/\langle m \rangle$ and $\mathbb{K}^n$ are vector spaces of dimension n over $\mathbb{K}$. Moreover, χ is a $\mathbb{K}$-linear map, which is a bijection as soon as the evaluation points $u_0, \ldots, u_{n-1}$ are pairwise distinct.

Complexity measures. Since we are targeting GPU implementation, our parallel algorithms are analyzed using an appropriate model of computation introduced in Chapter 3. The complexity measures are the *work* (i.e. algebraic complexity) the *span* (i.e. running time on infinitely many processors) and the *overhead*. This latter measures the total amount of data transfered between the global memory and the local memories.

Notation 1. *The number of operations for multiplying two polynomials with degree less than d using the* plain (or long) multiplication *is $\mathsf{M}_{\text{plain}}(d) = 2d^2 - 2d + 1$. In our GPU implementation, for d small enough, one polynomial multiplication can be done by a single thread block and thus within the local memory of a streaming multiprocessor. In this case, we use $2d+2$ threads for one polynomial multiplication. Each thread copies one coefficient from global memory to the local memory. Each of these threads, except one, is responsible for computing one coefficient of the output polynomial and writes that coefficient back to global memory. So the span and parallelism overhead are $d + 1$ and $2U$ respectively.*

Notation 2. *The number of operations for multiplying two polynomials with degree less than d using* Cooley-Tukey's FFT *algorithmsis: $\mathsf{M}_{\text{FFT}}(d) = 9/2d' \log_2(d') + 4d'$ [57]. Here $d' = 2^{\lceil \log_2(2d-1) \rceil}$. In our GPU implementation, which relies on Stockham FFT algorithm, this number of operations becomes: $\mathsf{M}_{\text{FFT}}(d) = 15d' \log_2(d') + 2d'$ [53]. The span and overhead of our implementation of FFT-based multiplication are $15d' + 2d'$ and $(36d' - 21)U$ respectively.*

Notation 3. *Given $a, b \in \mathbb{K}[x]$, with $\deg(a) \geq \deg(b)$ we denote by $\mathsf{Remainder}(a, b)$ the remainder in the* Euclidean division *of a by b. The number of operations for*

computing the Remainder(a, b)*, by plain division is* $(\deg(b)+1)(\deg(a)-\deg(b)+1)$*. In our GPU implementation, we perform plain division for small degree polynomials, where both* a, b *can be stored into the local memory of a streaming multiprocessor. We use* $\deg(b)+1$ *threads to implement this operation. Each thread reads one coefficient of* b *and at most* $\lceil \frac{\deg(a)+1}{\deg(b)+1} \rceil$ *coefficients of* a*, from the global memory. For the output, at most* $\deg(b)$ *threads write the coefficients of the remainder to the global memory. The span and overhead are* $2(\deg(a)-\deg(b)+1)$ *and* $(2+\lceil \frac{\deg(a)+1}{\deg(b)+1} \rceil)U$*.*

Notation 4. *Given a univariate polynomial* $f \in \mathbb{K}[x]$ *of degree* d *and a non-negative integer* $k \geq d$*, the* reversal *of order* k *of* f *is the polynomial denoted by* $\mathsf{rev}_k(f)$ *and defined as* $\mathsf{rev}_k(f) = x^k f(1/x)$*. In our implementation of this operation, we use one thread for each coefficient of the input and output. So the span and overhead are* 1 *and* $2U$*, respectively.*

Notation 5. *Addition and subtraction between two polynomials of degree* d *can be done within* $d+1$ *coefficient operations. In our implementation, we use one thread per coefficient operation. So the span and overhead are* 1 *and* $3U$*, respectively.*

Notation 6. *Given a univariate polynomial* $f \in \mathbb{K}[x]$*, with* $f(0) = 1$*, and* $\ell \in \mathbb{N}$ *the* modular inverse *of* f *modulo* x^ℓ *is denoted by* $\mathsf{Inverse}(f, \ell)$ *and defined as* $\mathsf{Inverse}(f, \ell)\, f \equiv 1 \mod (x^\ell)$*. Note that* $\mathsf{Inverse}(f, \ell)$ *is unique. Algorithm 25 computes* $\mathsf{Inverse}(f, \ell)$ *using* Newton iteration*. Observe that, this algorithm has* $\lceil \log_2 \ell \rceil$ *dependent steps. In other words, the for-loop cannot be turned to a parallel for-loop.*

Algorithm 25: $\mathsf{Inverse}(f, \ell)$

Input: $f \in \mathbb{R}[x]$ such that $f(0) = 1$ and $\ell \in \mathbb{N}$.
Output: $g_r \in \mathbb{R}[x]$ such that $f g_r \equiv 1 \mod x^\ell$.

```
1 g_0 = 1;
2 r = ⌈log_2 ℓ⌉;
3 for i = 1...r do
4     g_i = (2g_{i-1} − f g_{i-1}^2) mod x^{2^i};
5 return g_r;
```

Remark 7. *We create a Maple worksheet* [1] *for computing the space and algebraic complexity, span and overhead for constructing subproduct tree and subinverse tree (our proposed data structure). We also write formulas to compute algebraic complexity, span and overhead for evaluation and interpolation of univaivariate polynomial.*

[1] available at http://publish.uwo.ca/∼shaque4/

10.3 Subproduct tree

In this section, we study an adaptive algorithm for constructing the subproduct tree $M_n := \mathsf{SubproductTree}(u_0, \ldots, u_{n-1})$ as defined in Section 10.2. Recall that $n = 2^k$ holds for some positive integer k and $u_0, \ldots, u_{n-1} \in \mathbb{K}$.

Both polynomial evaluation and interpolation by subproduct tree techniques depend highly on polynomial multiplication, which brings several implementation challenges.

First of all, it is well-known that for univariate polynomials of low degrees, FFT-based multiplication is most costly than plain multiplication. For this reason, we apply plain multiplication in constructing the nodes of levels $1, \ldots, H$ of in the subproduct tree M_n, where $0 < H \leq k$ is a prescribed threshold, Then, we use FFT-based multiplication for the nodes of higher level.

A second challenge follows from the following observation. Each polynomial in the subproduct tree at level i has length $2^i + 1$ which is not a favorable case for FFT-based multiplication. Fortunately, the leading coefficient of any such polynomial in the subproduct tree is 1. So, it is possible to create $M_{i,j}$ from $M_{i-1,2j}$ and $M_{i-1,2j+1}$, even if we do not store the leading coefficients of the latter two polynomials.

As we will see in Section 10.7 our implementation still has room for improvements regarding polynomial multiplication. For instance, we could consider using an "intermediate" algorithm for polynomials with degree in a "middle range". Such an algorithm could be the one of Karatsuba or one of its variants. However, it is known that these algorithms are hard to parallelize [12].

Definition 5. *Let H be a fixed integer with $1 \leq H \leq k$. We call the following procedure an* adaptive algorithm for computing M_n with threshold H*:*

1. *For each level $1 \leq h \leq H$, the nodes are computed using plain multiplication.*
2. *Then, for each level $H + 1 \leq h \leq k$, the nodes are computed using FFT-based multiplication.*

This algorithm is adaptive in the sense that, on a GPU, each plain multiplication is done by a single streaming multiprocessor (SM) while each FFT-based multiplication is computed by a kernel call, thus using several SMs. In fact, this kernel computes a number of FFT-based products concurrently. Therefore, the algorithm adapts itself to the amount of available resources and, thus, it is adaptive.

Before analyzing this adaptive algorithm, we consider the situation where the subproduct tree M_n is computed by means of a single multiplication algorithm, with

multiplication time[2] $\mathsf{M}(\mathsf{n})$. In this context, Lemma 10.4 in [20] states that the total number of operations for constructing the subproduct tree M_n is at most $\mathsf{M}(n)\log_2 n$ operations in $\mathbb{K}$. Lemma 8 below prepares to the study of our adaptive algorithm.

Lemma 8. *Let $0 \leq h_1 < h_2 \leq k$ be integers. Assume that level h_1 of M_n has already been constructed. The total number of operations in $\mathbb{K}$ for constructing levels $h_1+1, \ldots, h_2$ in M_n is at most $\sum_{i=h_1+1}^{h_2} 2^{k-i} M(2^{i+1})$.*

Proof ▷ Recall that $M(d)$ is an upper bound on the number of operations in $\mathbb{K}$ for multiplying two univariate polynomials of degree less than d. Let $h_1 < i \leq h_2$ be an index. To construct the i-th level, we need 2^{k-i} number of multiplications in degree less than 2^{i+1}. So the total cost to construct for level i is upper bounded $2^{k-i}M(2^{i+1})$. □

We can have an immediate consequence from Lemma 8 by setting $h_1 = 0$ and $h_2 = k$.

Corollary 4. *The number of operations for constructing the M_n is $\sum_{i=1}^{k} 2^{k-i} M(2^{i+1})$.*

Remark 8. *We do not store the leading coefficient of polynomials in M_n of levels $H+1, \ldots, k-1$. So, the length of a polynomial becomes 2^i at level i. The objective of this technique is to reduce the computation time for FFT based multiplication. As the leading coefficient is always 1, we proceed as follows.*

Let $a, b \in \mathbb{K}[x]$ be two monic and univariate polynomials. Let $\deg(a) = \deg(b) = d = 2^e$ for some $e \in \mathbb{N}$. Let $a' = a - x^d$ and $b' = b - x^d$. Then, we have $ab = x^{2d} + a'b' + (a' + b')x^d$.

If we were to compute ab directly the cost would be $O(\mathsf{M}_{\mathrm{FFT}}(2d))$. But if compute it from $a'b'$ using the above formula, then the cost is reduced to $O(\mathsf{M}_{\mathrm{FFT}}(d) + d)$. On the RAM model, this technique saves almost half of the computational time. On a many-core machine, though the cost is not significant in theory, it saves $O(d)$ memory space and also saves about half of the work. In fact, this has a significant impact on the computational time, as we could observe experimentally.

With Corollary 4, we turn our attention to the algebraic and space complexity of our adaptive algorithm. Recall all of the formulas below are computed on the Maple worksheet.

Proposition 21. *The algebraic complexity of the adaptive algorithm for computing M_n with threshold H is given below*

[2]This notion is defined in Chapter 8 of [20]

$$\left(-\frac{17}{2}\,H+2^H+\frac{19}{2}\,\log_2(n)+\frac{15}{2}\log_2(n)^2-\frac{15}{2}\,H^2-\frac{1}{2^H}\right)n.$$

PROOF ▷ We compute the algebraic complexity of constructing M_n with threshold H from Corollary 4. We rely on the cost of polynomial multiplication given in Notations 1 and 2. Note that, we apply the technique described in Remark 8 for FFT-based multiplication to create the polynomials of level $H+1,\ldots,k$ of M_n. According to our Maple worksheet, the algebraic complexity for computing levels $0,1,\ldots,H$ of M_n using plain arithmetic is $\frac{n}{2}\left(19\,\log_2(n)+15\log_2(n)^2-19\,H-15\,H^2\right)$ coefficient operations. For levels $H+1,\ldots,k$ of M_n, the cost is $n(H+2^H-\frac{1}{2^H})$ coefficient operations. We obtain the algebraic complexity for constructing M_n by adding these two quantities. □

Proposition 22. *The amount of machine words required for storing M_n, with threshold H is given below*

$$(-H-2)\left(n+\frac{n}{2^{H+1}}\right)+2nH\left(1+\frac{1}{2^{H+2}}\right)+n\left(\log_2(n)-H+5\right).$$

PROOF ▷ Following our adaptive algorithm, we distinguish the nodes at levels $0,\ldots H$ from those at levels $H+1,\ldots,k$. At level $i\in\{0,\ldots H\}$, the number of coefficients of each polynomial of M_n is 2^i+1 and all those coefficients are stored. We make a formula in our Maple worksheet and compute the total number of coefficients over all polynomials in M_n for level $\{0,\ldots,H\}$, which is $(-H-2)\left(n+\frac{n}{2^{H+1}}\right)+2nH\left(1+\frac{1}{2^{H+2}}\right)+5\,n$.

At level $i\in\{H+1,\ldots,k\}$, we use the implementation technique described in Remark 8, that is, leading coefficients of each polynomial are not stored. So a polynomial at level i requires 2^i words of storage. From the same worksheet, we compute the total number of words required to store polynomials at level $\{H+1\ldots,k\}$, which is $n\left(\log_2(n)-H\right)$.

□

Proposition 23. *Span and overhead of Algorithm 24 for constructing M_n with threshold H using our adaptive method are* span_{M_n} *and* $\mathsf{overhead}_{M_n}$ *respectively, where*

$$\mathsf{span}_{M_n}=\frac{9}{2}\,H-2+2^{H+1}+\frac{15}{2}\,\left(\log_2(n)+1\right)^2-\frac{7}{2}\log_2(n)-\frac{15}{2}\,\left(H+1\right)^2$$

and

$$\mathsf{overhead}_{M_n}=\left(2\,H+\left(18\,\left(\log_2(n)+1\right)^2-35\log_2(n)-18\,\left(H+1\right)^2+35\,H\right)\right)U.$$

PROOF ▷ Let us fix i with $0 \leq i < H$. At level i, our implementation uses plain multiplication in order to compute the polynomials at level $i+1$. Following Notation 1, the span and the parallelism overhead of this process are $H - 2 + 2^{H+1}$ and $2\,HU$, respectively. For level $H \leq i < k$, each thread is participating to one FFT-based multiplication and two coefficient additions (in order to implement the trick of Remark 8). With Notation 2 and 5, we obtain the span and overhead for this step from Maple worksheet as $\frac{15}{2}\,(\log_2(n)+1)^2 - \frac{7}{2}\log_2(n) - \frac{15}{2}\,(H+1)^2 + \frac{7}{2}\,H$ and $\left(18\,(\log_2(n)+1)^2 - 35\log_2(n) - 18\,(H+1)^2 + 35\,H\right)U$ respectively. □

Remark 9. *In order to determine the value H for which the algebraic complexity given in Proposition 21 and the span and overhead given in Proposition 23 are minimized, we computed the algebraic complexity, span and overhead for $k = 5, \ldots, 24$ and $1 \leq H \leq k$. Using our Maple worksheet, we found that, for $H = 7$ and 6 minimize algebraic complexity and span respectively. But the parallelism overhead minimizes for $H = k$. From Notation 1, we realize that we can not do plain multiplication of big polynomials due to the space limitation in local memory. Because, each of the plain multiplication is done on local memory by one thread block without communicating with global memory until the multiplication ends. In fact, for our GPU card on which we run all our experiments, we can not have $H > 8$. Considering all these facts, We have set $H = 8$ for our implementation*

10.4 Subinverse tree

Given a subproduct tree M_n, multi-point evaluation of a polynomial $f \in \mathbb{K}[x]$ of degree less than n, on the point set $\{u_0, \ldots, u_{n-1}\}$ can be done by calling the recursive algorithm $\mathsf{TopDownTraverse}(f, k, 0, M_n, F)$ (Algorithm 26). This algorithm is called with an array F of length n, to which the result is written as $F[i] = f(u_i)$. We implement both *fast division* [20] (as in Algorithm 31) and plain division to compute reminders like $\mathsf{Remainder}(f', M_{i,j})$. Fast division is applied when polynomials are large enough and, actually, can not be stored within the local memory of a streaming multiprocessor. In our implementation, the call $\mathsf{Remainder}(f', M_{i,j})$ is performed by plain division, whenever $i < H$ holds, where H is the threshold of Definition 5. Fast division requires computing $\mathsf{Inverse}(\mathsf{rev}_{2^i}(M_{i,j}), 2^i)$, for $H \leq i \leq k$ and $0 \leq j < 2^{k-i}$, (see Algorithm 25).

Assume that we want to compute in parallel the inverses of a number of different polynomials modulo the same power x, say modulo x^{2^i}. That means that the same algorithm (Algorithm 25) runs on different streaming multiprocessors with different

polynomials. As mentioned in Notation 6, the for-loop in this algorithm can not be converted to a parallel for-loop. Since the first iteration of this for-loop have much less work than the last one, running this for-loop on a streaming multiprocessor under-utilizes computing resources.

To overcome this performance issue we introduce a strategy that relies on a new data structure called *subinverse tree*. In this section, we first define subinverse trees and describe their implementation. We then analyze the complexity of building subinverse trees.

Algorithm 26: $\mathsf{TopDownTraverse}(f', k', h', M_n, F)$

Input: $f' \in \mathbb{K}[x]$ and $\deg(f') \leq 2^{k'} - 1$, $k', h' \in \mathbb{N}$ and F is an array of length n.

```
1 if deg(f') == 0 then
2 |   F[h'] = f';
3 |   return;
4 f_0 = Remainder(f', M_{k-1,2h});
5 f_1 = Remainder(f', M_{k-1,2h+1});
6 TopDownTraverse(f_0, k' - 1, 2h', M_n, F);
7 TopDownTraverse(f_1, k' - 1, 2h' + 1, M_n, F);
```

Definition 6. *Given a subproduct tree* $M_n := \mathsf{SubproductTree}(u_0, \ldots, u_{n-1})$*, the subinverse tree,* InvM_n *associated with* M_n*, is a complete binary tree of the same format as* M_n*. For* $0 \leq i \leq k$*, for* $0 \leq j < 2^{k-i}$*, the* j*-th node of level* i *in* InvM_n *contains the univariate polynomial* $\mathsf{InvM}_{i,j}$ *of degree* $2^i - 1$ *defined by*

$$\mathsf{InvM}_{i,j}\, \mathsf{rev}_{2^i}(M_{i,j}) \;\equiv\; 1 \mod x^{2^i}.$$

Since the purpose of the subinverse tree is to make the fast division efficient, we do not store the polynomials of subinverse tree InvM_n below level H. With this assumption, the total space required to store the subinverse treeInvM_n is given in Proposition 24.

Proposition 24. *Let* InvM_n *be the subinverse tree associated with a subproduct tree* M_n*, with threshold* $H < k$*. Then, the amount of space required for storing* InvM_n*, excluding its root and all levels* $i < H$*, is* $(k - H)n$*.*

Proof ▷ From the Definition 6, we realize the length of $\mathsf{InvM}_{i,j}$ is 2^i. As the total number of polynomials at level i in InvM_n is 2^{k-i}, we need 2^k, that is, n machine words to store all polynomials of level i. Here we are not considering the root of InvM_n to

store because in evaluation or interpolation of an univariate polynomial, we do not need this. □

Let $\mathbb{R}$ be a commutative ring with identity element. Let $a, b, c \in \mathbb{R}[x]$ be three univariate polynomials such that $c = ab$ and $a(0) = b(0) = 1$ hold. Thus, we have $c(0) = 1$. Let $d = \deg(c)+1$ Proposition 25 describes how we can compute $\mathsf{Inverse}(c, d)$ mod x^d from a and b.

Proposition 25. $\mathsf{Inverse}(c, d) \equiv \mathsf{Inverse}(a, d) \cdot \mathsf{Inverse}(b, d) \mod x^d$.

Proposition 25 is the main technical ingredient in creating a subinverse tree. We can rewrite this proposition for the polynomials in M_n in Proposition 26.

Proposition 26. *Let* $\mathsf{InvM}_{i,j}$ *be the* j^{th} *polynomial (from left to right) of the subinverse tree at level* i*, where* $0 < i < k$ *and* $0 \leq j < 2^{k-i}$*. We have the following*

$$\mathsf{InvM}_{i,j} \equiv \mathsf{Inverse}(\mathsf{rev}_{2^{i-1}}(M_{i-1,2j}), 2^i) \cdot \mathsf{Inverse}(\mathsf{rev}_{2^{i-1}}(M_{i-1,2j+1}), 2^i) \mod x^{2^i},$$

where $\mathsf{InvM}_{i,j} = \mathsf{Inverse}(\mathsf{rev}_{2^i}(M_{i,j}), 2^i)$ *from Definition 6.*

The key observation is that computing $\mathsf{InvM}_{i,j}$ requires $\mathsf{Inverse}(\mathsf{rev}_{2^{i-1}}(M_{i-1,2j}), 2^i)$ and $\mathsf{Inverse}(\mathsf{rev}_{2^{i-1}}(M_{i-1,2j+1}), 2^i)$. However, at level $i-1$, the nodes $\mathsf{InvM}_{i-1,2j}$ and $\mathsf{InvM}_{i-1,2j+1}$ are $\mathsf{Inverse}(\mathsf{rev}_{2^{i-1}}(M_{i-1,2j}), 2^{i-1})$ and $\mathsf{Inverse}(\mathsf{rev}_{2^{i-1}}(M_{i-1,2j+1}), 2^{i-1})$ respectively. We describe how we use $\mathsf{InvM}_{i-1,2j}$ and $\mathsf{InvM}_{i-1,2j+1}$ in order to apply Proposition 26 and deduce $\mathsf{InvM}_{i,j}$.

The calculation of the subinverse tree InvM_n can be described in a recursive way. It is constructed in bottom-up fashion like the subproduct tree M_n.

The construction starts with the computation of all polynomials $\mathsf{InvM}_{H,j}$ of level H in InvM_n from the corresponding polynomial $M_{H,j}$ in M_n using Algorithm 25, where $0 \leq j < 2^{k-H}$. Next, we assume that $\mathsf{InvM}_{i-1,j*2}$ and $\mathsf{InvM}_{i-1,j*2+1}$ have already been computed and show how to deduce $\mathsf{InvM}_{i,j}$. We call $\mathsf{OneStepNewtonIteration}(\mathsf{rev}_{2^{i-1}}(M_{i-1,j*2}),\ \mathsf{InvM}_{i-1,j*2}, i-1)$ and $\mathsf{OneStepNewtonIteration}(\mathsf{rev}_{2^{i-1}}(M_{i-1,j*2+1}), \mathsf{InvM}_{i-1,j*2+1}, i-1)$ (see Algorithm 27) so as to obtain $\mathsf{Inverse}(M_{i-1,2j}, 2^i)$ and $\mathsf{Inverse}(M_{i-1,2j+1}, 2^i)$ respectively. Algorithm 27 performs a single iteration of *Newton iteration*'s algorithm. Finally, we perform one truncated polynomial multiplication, as shown in Proposition 26 to obtain $\mathsf{InvM}_{i,j}$. We apply this technique to compute all the polynomials of level i of the subinverse tree.

As we do not store the leading coefficients of the polynomials in the subproduct tree, our implementation of Algorithm 27 is not straightforward. Algorithm 28 is the

modified implementation of Algorithm 27 considering to our specifications. We are going to describe this algorithm below.

Algorithm 27: OneStepNewtonIteration(f, g, i)

Input: $f \in \mathbb{R}[x]$ such that $f(0) = 1$, where $\deg(f) = 2^i$ and $fg \equiv 1 \mod x^{2^i}$.
Output: $g' \in \mathbb{R}[x]$ such that $fg' \equiv 1 \mod x^{2^{i+1}}$.

```
1 g' = (2g − fg^2) mod x^{2^{i+1}};
2 return g';
```

Let $f = \mathsf{rev}_{2^i}(M_{i,j})$ and $g = \mathsf{InvM}_{i,j}$. From Definition 6, $fg \equiv 1 \bmod x^{2^i}$. Note that $\deg(fg) = 2^{i+1} - 1$. Let $e' = -fg + 1$. Thus e' is a polynomial of degree $2^{i+1} - 1$ and from the definition of subinverse tree, we know its least significant 2^i coefficients are zeros. Let $e = e'/x^{2^i}$. So $\deg(e) = 2^i - 1$. In Algorithm 27, we have $g' \equiv g \mod x^{2^i}$. We can compute g' from eg and g. The advantage of working with e instead of e' is that the degree of e' is twice than that of e.

In Algorithm 28, we compute e in the following way,

$$e = -\mathsf{rev}_{2^i}(M_{i,j} \cdot \mathsf{rev}_{2^i-1}(\mathsf{InvM}_{i,j}) - x^{2^{i+1}-1})$$

by means of one convolution and three more polynomial operations. As we do not store the leading coefficient of $M_{i,j}$, we need to do these three additional operations.

Algorithm 28: EfficientOneStep$(M'_{i,j}$,InvM$_{i,j}, i)$

Input: $M'_{i,j} = M_{i,j} - x^{2^i}$, $\mathsf{InvM}_{i,j}$ is a polynomial in subinverse tree.
Output: g, such that $g \ \mathsf{rev}_{2^i}(M_{i,j}) \equiv 1 \mod x^{2^{i+1}}$.

```
1 a =rev_{2^i−1}(InvM_{i,j});
2 b = a − x^{2^i−1};
3 c =convolution(a, M'_{i,j}, 2^i);
4 d =rev_{2^i}(c + b);
5 e = −d;
6 h = e InvM_{i,j} mod x^{2^i};
7 g = hx^{2^i}+InvM_{i,j};
8 return g;
```

Middle product technique is an implementation trick used in Algorithms 25 and 27. This improves the computational time significantly [28]. We do not apply middle product technique directly in constructing subinverse tree. This technique works well for the iterations of Algorithm 25 where the intermediate inverse polynomial g_i is smaller in degree than polynomial f. This is not the case in our Algorithm 28.

Algorithm 29 computes each polynomial, $\mathsf{InvM}_{i,j}$ of InvM_n, where $H < i < k$. It calls Algorithm 28 twice to expand the inverse of both $\mathsf{InvM}_{i-1,2j}$ and $\mathsf{InvM}_{i-1,2j+1}$. Then it multiplies the expanded inverse polynomials and applies a mod operation.

Algorithm 30 is the top level algorithm to create InvM_n. Each $\mathsf{InvM}_{i,j}$ is created by one call to Algorithm 29.

Algorithm 29: InvPolyCompute(M_n,$\mathsf{InvM}_{i,j}$)

Input: M_n, and InvM are the subproduct tree and subinverse tree respectively.
Output: c, such that $c\ \mathsf{rev}_{2^i}(M_{i,j}) \equiv 1 \bmod x^{2^i}$.

$M'_{i-1,2j} = M_{i-1,2j} - x^{2^{i-1}}$;
$M'_{i-1,2j+1} = M_{i-1,2j+1} - x^{2^{i-1}}$;
$a = \mathsf{EfficientOneStep}(M'_{i-1,2j}, \mathsf{InvM}_{i-1,2j}, i-1)$;
$b = \mathsf{EfficientOneStep}(M'_{i-1,2j+1}, \mathsf{InvM}_{i-1,2j+1}, i-1)$;
$c = ab \bmod x^{2^i}$;
return c;

Algorithm 30: SubinverseTree(M_n, H)

Input: M_n is the subproduct tree and $H \in \mathbb{N}$.
Output: subinverse tree, InvM_n

each node in InvM_n for level $0, \ldots H-1$ contains a zero polynomial;
for $j = 0 \ldots 2^{k-H} - 1$ **do**
    $\mathsf{InvM}_{H,j} = \mathsf{Inverse}(M_{H,j}, \deg(M_{H,j}))$;
for $i = H+1 \ldots k-1$ **do**
    for $j = 0 \ldots 2^{k-i} - 1$ **do**
        $\mathsf{InvM}_{i,j} = \mathsf{InvPolyCompute}(M_n, \mathsf{InvM}_{i,j})$;
return InvM_n;

Proposition 27. *For a given subproduct tree, M_n with threshold H, the algebraic complexity for constructing the subinverse tree InvM_n by Algorithm 30 is given below*

$$n\left(-\frac{1}{3\cdot 2^H} + 2 + \frac{16\,4^{2^H}}{3\cdot 2^H} - \frac{2}{2^{H-2^H}} - 10\,\left(-\log_2(n) - 3\log_2(n)^2 + 7\,H + 3\,H^2 + 4\right)\right).$$

Proof ▷ At level H, we need to compute 2^{k-H} polynomials. For each polynomials, we need to call Algorithm 25, with a polynomial of subproduct tree at level H, whose degree is 2^H and the other parameter is 2^H. So the loop in this algorithm runs H times. We apply plain multiplications for this step. with the idea of middle product technique

to make the implementation fast. In middle product technique, we require convolution to compute some coefficients of a polynomial multiplications. In plain arithmetic, we can do the same in a direct way. For example, in the i-th iteration of the for-loop in Algorithm 25 for $i = 2, \ldots, H$, we need to to compute 2^{i-1} coefficients of g_i. We can treat both f and g_{i-1} as polynomials of degree less than 2^{i-1}. Thus this multiplication cost can be expressed as $\mathsf{M}_{\text{plain}}(2^{i-1})$ We need two polynomial multiplications of this type in each iteration. We also need some polynomial subtraction operations too. Observe that computing g_0 and g_1 is trivial in Algorithm 25. We create a formula in our Maple worksheet based on our implementation described with and Notation 1 to compute the total number of coefficient operations for construction the H-th level of InvM_n. According to our Maple worksheet the total number of operation is given below

$$-\frac{n}{3 \cdot 2^H} + 2n + \frac{16\, n\, 4^{2^H}}{3 \cdot 2^H} - \frac{2n}{2^{H-2^H}}.$$

After level H, each polynomial in InvM_n is computed by the equation given in Proposition 26. Note that, when we are constructing the i-th level of subinverse tree, it is assumed that we have all the polynomials at level $i-1$. All polynomial multiplications in these levels are FFT-based. From Algorithm 28 and Proposition 26 along with Notation 2, we compute the total number of operations required to compute the polynomials from level $H+1$ to $k-1$ in our Maple worksheet. It is given below

$$-10\, \left(-\log_2(n) - 3\log_2(n)^2 + 7\, H + 3\, H^2 + 4\right) n.$$

We sum up these two complexity estimates and we get the result. □

Proposition 28. *Given a subproduct tree M_n with threshold H, the span and overhead of constructing the corresponding subproduct tree InvM_n by Algorithm 30 are $\mathsf{span}_{\mathsf{InvM}_n}$ and $\mathsf{overhead}_{\mathsf{InvM}_n}$ respectively, where*

$$\mathsf{span}_{\mathsf{InvM}_n} = 4 \cdot 2^H + 14 + 2 \cdot 4^H + \frac{75}{2}\log_2(n)^2 - \frac{107}{2}\log_2(n) - \frac{75}{2}\, H^2 - \frac{43}{2}\, H$$

and

$$\mathsf{overhead}_{\mathsf{InvM}_n} = U\left(2^{H+1} + 166 + 90\log_2(n)^2 - 255\log_2(n) - 90\, H^2 + 75\, H\right).$$

Proof ▷ The construction of InvM_n can be divided into two steps. First, we compute the polynomials at level H using plain arithmetic by Algorithm 25. During this step, we assign one thread to compute one polynomial of InvM_n. So its span is equal to

the complexity of Newton iteration algorithm that computes inverse of a polynomial of degree 2^H modular x^{2^H}. One kernel call is enough to compute this. Moreover each thread is responsible to copy one polynomial at level H of the subproduct tree from global memory to local memory. The span and overhead that we compute for this step from our Maple worksheet are $4 \cdot 2^H - 2 + 2 \cdot 4^H$ and $\left(2^{H+1} + 1\right) U$ respectively.

Second, we construct level $H + 1, \ldots, (k - 1)$ of InvM_n. As mentioned before, we do not construct the root of the subinverse tree. For a level above H, each thread participates in three FFT-based multiplications and five other coefficient operations (involving shifting, addition, copying). For each of the operations, except FFT-based multiplication, each thread requires accessing at most three times in global memory. So the span and overhead for this step is computed from Notation 2 and 5 using our Maple worksheet and are $\frac{75}{2}\log_2(n)^2 - \frac{107}{2}\log_2(n) - \frac{75}{2}H^2 - \frac{43}{2}H + 16$ and $15\left(6\log_2(n)^2 - 17\log_2(n) - 6H^2 + 5H + 11\right)U$ respectively. □

10.5 Polynomial evaluation

Multi-point evaluation of polynomial $f \in \mathbb{K}[x]$ of degree less than n, for points in $\{u_0, \ldots, u_{n-1}\}$ can be done by *Horner's* rule in $O(n^2)$ time. If we consider parallel architecture to solve this problem, the span becomes $O(n)$. Subproduct tree based multi-point evaluation has better time complexity and span than that.

Algorithm 26 solves multi-point evaluation problem using subproduct tree technique. We construct subproduct tree $M_n := \mathsf{SubproductTree}(u_0, \ldots, u_{n-1})$ with threshold H and corresponding subinverse tree InvM_n. Algorithm 26 requires polynomial division operations. We implement both fast and plain division as described in Section 10.4 for this purpose. In our implementation of fast division given in Algorithm 31, we do not need to compute $\mathsf{inverse}(f, t - s + 1)$, as it is already computed and stored in the subinverse tree. Below level H, we apply plain division algorithm to compute the remainder of polynomials.

Proposition 29. *Given a subproduct tree M_n with threshold H and the corresponding subinverse tree InvM_n, the algebraic complexity of Algorithm 26 is given below*

$$-30\,nH^2 - 46\,nH + 74\,n + 16\,\frac{n}{2^H} + 106\,n\log_2(n) + 30\,n\log_2(n)^2 - 8 + n\,2^{H+1}.$$

Proof ▷ Our adaptive algorithm has two steps. First, we need to call Algorithm 31 for computing the remainder for $k' = k, \ldots, (H + 1)$. We do not need to compute the *inverses of polynomials* as we have InvM_n. All of the multiplications in this algorithm

are FFT-based. We need two multiplications and four other operations (polynomial reversals and subtraction). Following Notations 2, 4 and 6, we obtain the algebraic complexity of this step from our Maple worksheet as

$$-30\,nH^2 - 46\,Hn + 76\,n + 16\,\frac{n}{2^H} + 106\,n\log_2(n) + 30\,n\log_2(n)^2 - 8.$$

Second, while $k' = H, \ldots, 1$ in Algorithm 31, we call plain division algorithm described in Notation 3. The total number of operation that we obtain with this notation from our Maple worksheet for this step is $n\,2^{H+1} - 2\,n$. □

Remark 10. *In [52], the algebraic complexity for solving* multi-point evaluation *is* $7M(n/2)\,\log_2(n) + O(M(n))$ *considering only multiplication cost and ignoring other coefficient operations. In our proposed method, for constructing one polynomial in subinverse tree, we require two polynomial convolutions and two polynomial multiplications between corresponding polynomials (children of that polynomial) from subproduct tree and subinverse tree and one polynomial multiplication. We require two polynomial multiplications in each call to Algorithm 31, if subinverse tree is given. In addition, we need one multiplication to create one polynomial in subproduct tree. So in total, the algebraic complexity for solving* multi-point evaluation *using our proposed method is* $3M(n/2)(\log_2(n)-1) + (2M(n/2) + 4M(n/4) + 4\mathsf{CONV}(n/4)(\log_2(n)-2))$ *considering only the polynomial multiplication and ignoring all other coefficient operations. It should be noted, we start computing the subinverse tree from level* H*. Each leaf of the subinverse tree has a constant polynomial that is* 1*. Now if we compare our complexity estimate with that in [52], (converting the convolution time to an appropriate multiplication time), we might not see any significant differences. In practice, our proposed method should performs better on parallel machine. We do not compare these techniques. We keep it for future work.*

Considering our adaptive strategy, we compute the exact number of operations in solving multi-point evaluation *problem by adding the algebraic complexity of constructing* M_n *along with the corresponding* InvM_n *and Algorithm 26 found in Proposition 21, 27 and 29 respectively.*

Proposition 30. *Given a subproduct tree* M_n *with threshold* H *and the corresponding subinverse tree* InvM_n*, span and overhead of Algorithm 26 are* $\mathsf{span}_{\mathsf{eva}}$ *and* $\mathsf{overhead}_{\mathsf{eva}}$ *respectively, where*

$$\mathsf{span}_{\mathsf{eva}} = -22\,H - 2 + 6\,2^H + 15\log_2(n)^2 + 23\log_2(n) - 15\,H^2$$

and

$$\mathsf{overhead}_{\mathsf{eva}} = \left(2\,H + 36\log_2(n)^2 + 3\log_2(n) - 36\,H^2\right) U.$$

PROOF ▷ From the proof of Proposition 29, we can compute the span and overhead of Algorithm 26, when the value of $k' = k, \ldots, (H+1)$, using Notations 2, 4 and 6 as

$$15\log_2(n)^2 + 23\log_2(n) - 15\,H^2 - 23\,H$$

and

$$3\,U\left(12\log_2(n)^2 + \log_2(n) - 12\,H^2 - H\right)$$

respectively. Once the Algorithm 26 depends on plain arithmetic for division that means when $k' = H, \ldots, 1$, the span and overhead can be computed using Notation 3 as $H-2+6\,2^H$ and $5\,HU$ respectively. We obtain this result from our Maple worksheet too. □

Algorithm 31: FastRemainder(a, b)

Input: $a, b \in \mathbb{R}[x]$ with $b \neq 0$ monic.
Output: (q, r) such that $a = bq + r$ and $\deg(r) < \deg(b)$

```
1  t = deg(a);
2  s = deg(b);
3  if t < s then
4      q = 0;
5  r = a;
6  else
7      f = rev_s(b);
8      g = inverse(f, t − s + 1);
9      q = rev_t(a)g  mod  x^{t−s+1};
       /* rev_t(a) means to replace x by 1/x in a and then multiply a
          with x^t.                                                  */
10     q = rev_{t−s}(q);
11     r = a − bq;
12 return (q, r);
```

10.6 Polynomial interpolation

Recall the Lagrange interpolation 10.2, We call $v_i s_i$, c_i for $0 \leq i < n$. Assume that we have computed $\{c_0, \ldots, c_{n-1}\}$. For generating the result, we call Algorithm 32 which proceeds from leaves of the existing subproduct tree to the root. This is based on the recursive algorithm 10.9 [20].

Algorithm 32: LinearCombination$(M_n, c_0, \ldots, c_{n-1})$

Input: Precomputed Subproduct Tree M_n for the points $u_0, \ldots, u_{n-1}$, and $c_0, \ldots, c_{n-1} \in \mathbb{K}$, and $n = 2^k$ for $k \in \mathbb{N}$

Output: $\sum_{0 \le i < n} c_i m/(x - u_i) \in \mathbb{K}[x]$, where $m = \prod_{0 \le i < n}(x - u_i)$

```
for j = 0 to n − 1 do
    I_{0,j} = c_j;
for i = 1 to k do
    for j = 0 to 2^{k−i} − 1 do
        I_{i,j} = M_{i−1,2j} I_{i−1,2j+1} + M_{i−1,2j+1} I_{i−1,2j};
return I_{k,0};
```

Here, $I_{i,j}$ means the intermediate result for corresponding j-th node from the left at level i in the subproduct tree. The degree of $I_{i,j}$ is $2^i - 1$. Finally the corresponding result to the root of the subproduct tree is the polynomial which goes exactly through the points which were used for constructing the subproduct tree. The degree of this unique polynomial will be one less than $m = \prod_{0 \le i < n}(x - u_i)$.

In this section, we study an adaptive algorithm stated in Definition 7 regarding interpolating the unique univariate polynomial over a prime field.

Definition 7. *Let H be a fixed integer with $1 \le H \le k$. We call* adaptive algorithm for computing $I_{k,0}$ with threshold H *the following procedure:*

1. *For every intermediate result $I_{h,j}$ where $1 \le h \le H$ and $0 \le j < 2^{k-h}$, we compute the $I_{h,j}$ using plain multiplication.*

2. *Then, for every intermediate result $I_{h,j}$ for $H + 1 \le h \le k$, we compute the $I_{h,j}$ using FFT-based multiplication.*

In Theorem 10.10 [20], the complexity estimates of the *Linear Combination* is stated as $(\mathsf{M}(n) + O(n)) \log(n)$. Here we present a more precise estimates in Proposition 31.

Proposition 31. *Given a subproduct tree M_n with threshold H, the algebraic complexity Algorithm 32 is given below*

$$20\, n\log_2(n) + 11\, n + 15\, n\log_2(n)^2 + 13\, nH - 15\, nH^2 + n2^{H+1} - n2^{1-H}.$$

Proof ▷ Each polynomial $I_{i,j}$ for $0 \le i < k$ and $0 \le j < 2^{k-i}$ is obtained by two polynomial multiplications and one polynomial addition. For level $i = 0, \ldots, H$,

by plain multiplication and addition, From our Maple worksheet with Notation 1 and 5, we obtain the total number of operations as $3\,Hn + 6\,n + n\,2^{H+1} - n\,2^{1-H}$. For the other levels, we apply FFT-based multiplication. From our Maple worksheet using Notation 2 and 5, we obtain the total number of operations as $5\,n\left(4\log_2(n) + 1 + 3\log_2(n)^2 + 2\,H - 3\,H^2\right)$. □

Finally we use Algorithm 33 in which we first compute $c_0, \ldots, c_{n-1}$, and then we call Algorithm 32. Algorithm 33 is adopted from Algorithm 10.11 [20]. The algebraic

Algorithm 33: FastInterpolation$(u_0, \ldots, u_{n-1}, v_0, \ldots, v_{n-1})$

Input: $u_0, \ldots, u_{n-1} \in \mathbb{K}$ such that $u_i - u_j$ is a unit for $i \neq j$, and $v_0, \ldots, v_{n-1} \in \mathbb{K}$, and $n = 2^k$ for $k \in \mathbb{N}$

Output: The unique polynomial $P \in \mathbb{K}[x]$ of degree less than n such that $P(u_i) = v_i$ for $0 \leq i < n$

1 M_n = SubproductTree$(u_0, \ldots, u_{n-1})$;
2 m' = Derivation of m (the root of M_n);
3 Construct the Subinverse Tree, InvM_n;
4 $\mathsf{si}_0 \ldots \mathsf{si}_{n-1}$ = Evaluate m' at $u_0, \ldots, u_{n-1}$ using Algorithm 26 with M_n and InvM_n;
5 return LinearCombination$(M_n, v_0/\mathsf{si}_0, \ldots, v_{n-1}/\mathsf{si}_{n-1})$;

complexity of *polynomial interpolation* by Algorithm 33 is stated in Remark 11.

Remark 11. *In Algorithm 33, we need to compute* multi-point evaluation *followed by Algorithm 32 for linear combination. In between these two major steps, we compute the* derivation *of the root of the subproduct tree, which can be done by n coefficient operations. So the algebraic complexity of Algorithm 33 is the summation of that of polynomial evaluation (from Remark 10), Proposition 31 and n for the derivation.*

Proposition 32. *Given a subproduct tree M_n with threshold H and the corresponding subinverse tree InvM_n, the Span and overhead of Algorithm 32 are $\mathsf{span}_{\mathsf{lc}}$ and $\mathsf{overhead}_{\mathsf{lc}}$ respectively, where*

$$\mathsf{span}_{\mathsf{lc}} = -\frac{21}{2}\,H - 2 + 2^{H+1} + \frac{15}{2}\log_2(n)^2 + \frac{25}{2}\log_2(n) - \frac{15}{2}\,H^2$$

and

$$\mathsf{overhead}_{\mathsf{lc}} = 4\,H + 18\log_2(n)^2 + \log_2(n) - 18\,H^2.$$

Proof ▷ At level i for $0 \leq i \leq H$, this algorithm does 2^{k-i} polynomial plain multiplications and 2^{k-i-1} polynomial additions. So each thread participates in one coefficient multiplication and one addition. Thus with Notation 1 and 5 from our

Maple worksheet we compute the span and overhead as $2\,H - 2 + 2^{H+1}$ and $5\,HU$ respectively.

For a level,i $(i > H)$ we have same number of polynomial multiplications. But each of the multiplication is done by FFT. As we do not store the leading coefficients for both $M_{i,j}$, we need one more polynomial addition. So a thread participates in one FFT-based multiplication and two coefficient additions. We compute the span and overhead for this step as

$$\frac{15}{2}\log_2(n)^2 + \frac{25}{2}\log_2(n) - \frac{15}{2}\,H^2 - \frac{25}{2}\,H$$

and

$$U\left(18\log_2(n)^2 + \log_2(n) - 18\,H^2 - H\right)$$

respectively from our Maple worksheet using Notation 2 and 5. □

10.7 Experimentation results

In Table 10.1 we compare the time to do *multi-point evaluation* for polynomials with different degrees with FFT-based polynomial multiplication of corresponding degrees are compared. We found the ratios between these two quantities between $[2.24, 4.02]$ on GPU card NVIDIA Tesla C2050. In Corollary 10.8 of [20] this ratio is considered as $\frac{11}{2}$. This is a promising result.

In Table 10.2 and Figure 10.2, we compare two implementations FFT-based polynomial multiplication. The first one is implemented with CUDA [53]. The other one is from FLINT library[3]. We executed our CUDA codes on a Nvidia Tesla M2050 GPU and the other code on the same machine with Intel Xeon X5650 CPU with 2.67GHz clock frequency. From the experimental data, it is clear that, our CUDA codes for FFT-based multiplication does not perform well when the degree of the polynomial is less than 2^{16}. This tells us that we need to implement another fast multiplication algorithm to have better performance in polynomial evaluation and interpolation. We keep it as future work.

In Table 10.3 we compare our implementation of polynomial evaluation and interpolation with that of FLINT library. We found that our implementation does not perform well until the degree of the polynomial is more than 2^{15}. Though we are about 21 times faster than FLINT for higher degree polynomials. We believe if our

[3] `http://www.flintlib.org/`

K	T_{eva}	T_{mul}	$T_{eva}/T_{mul} * k$
10	0.11	0.0049	2.24
11	0.17	0.0051	3.03
12	0.21	0.0060	2.91
13	0.28	0.0061	3.53
14	0.36	0.0069	3.72
15	0.42	0.0070	4.00
16	0.56	0.0087	4.02
17	0.70	0.0111	3.70
18	1.01	0.0163	3.44
19	1.50	0.0256	3.08
20	2.52	0.0438	2.80
21	4.61	0.0862	2.54
22	9.08	0.1654	2.49
23	18.83	0.3416	2.39

Table 10.1: Computation time for random polynomials with different degrees (2^K) and points. All of the times are in seconds.

Degree	GPU (s)	FLINT (s)	Speed-Up
9	0.001	0.001	0.602
10	0.0029	0	0
11	0.0019	0.002	1.029
12	0.0032	0.003	0.917
13	0.0023	0.008	3.441
14	0.0039	0.013	3.346
15	0.0032	0.023	7.216
16	0.0065	0.045	6.942
17	0.0084	0.088	10.475
18	0.0122	0.227	18.468
19	0.0198	0.471	23.738
20	0.0266	1.011	27.581
21	0.0718	2.086	29.037
22	0.1451	4.419	30.454
23	0.3043	9.043	29.717

Table 10.2: Execution times of multiplication

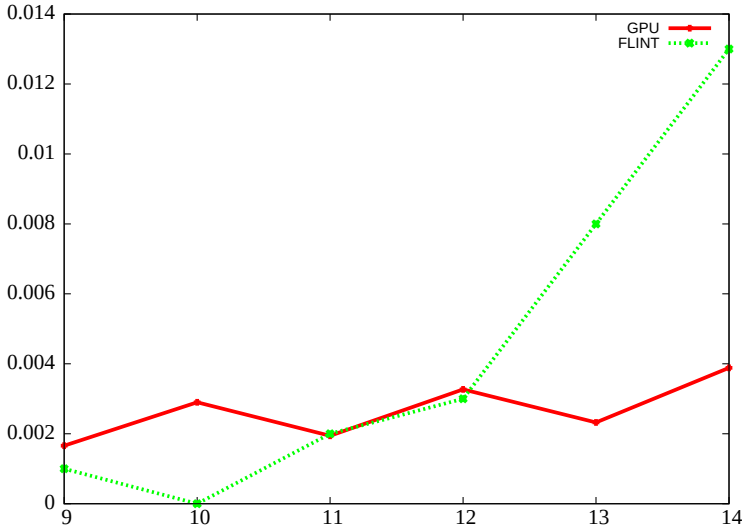

Figure 10.2: Our GPU implementation versus FLINT for FFT-based polynomial multiplication.

	Evaluation			Interpolation		
Degree	GPU (s)	FLINT (s)	Speed-Up	GPU (s)	FLINT (s)	Speed-Up
10	0.0843	0	0	0.0968	0.01	0.1032
11	0.1012	0.01	0.0987	0.1202	0.01	0.0831
12	0.1361	0.02	0.1468	0.1671	0.03	0.1794
13	0.1580	0.07	0.4429	0.1963	0.09	0.4584
14	0.2034	0.17	0.8354	0.2548	0.22	0.8631
15	0.2415	0.41	1.6971	0.3073	0.53	1.7242
16	0.3126	0.99	3.1666	0.4026	1.26	3.1294
17	0.4285	2.33	5.4375	0.5677	2.94	5.1780
18	0.7106	5.43	7.6404	0.9034	6.81	7.5379
19	1.0936	12.63	11.5484	1.3931	15.85	11.3768
20	1.9412	29.2	15.0420	2.4363	36.61	15.0268
21	3.6927	67.18	18.1923	4.5965	83.98	18.2702
22	7.4855	153.07	20.4486	9.2940	191.32	20.5851
23	15.796	346.44	21.9321	19.6923	432.13	21.9441

Table 10.3: Execution times of polynomial evaluation and interpolation.

multiplication routine for polynomials of degrees 2^9 to 2^{13} can improve, we would have better performance in both polynomial evaluation and interpolation in these ranges.

Figure 10.3 and Figure 10.3 compares the polynomial evaluation between our implementation and that of FLINT for lower degrees and higher degrees respectively.

Figure 10.4 and Figure 10.5 compares the polynomial interpolation between our implementation and that of FLINT for lower degrees and higher degrees respectively.

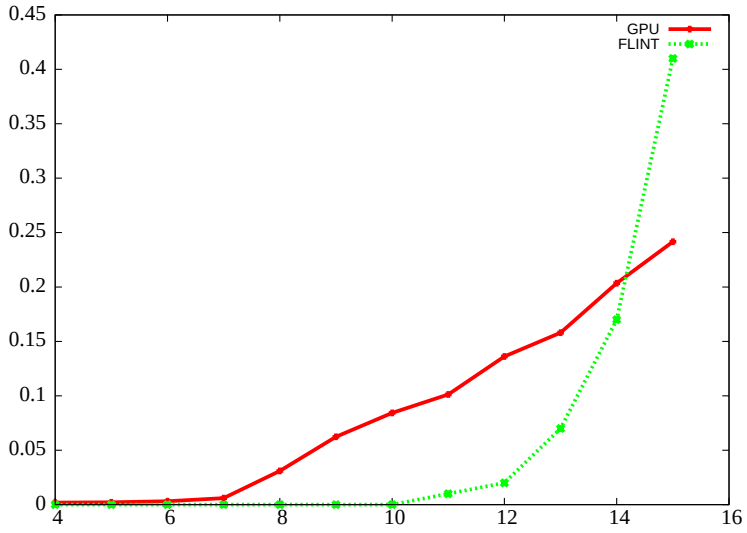

Figure 10.3: Evaluation lower degrees

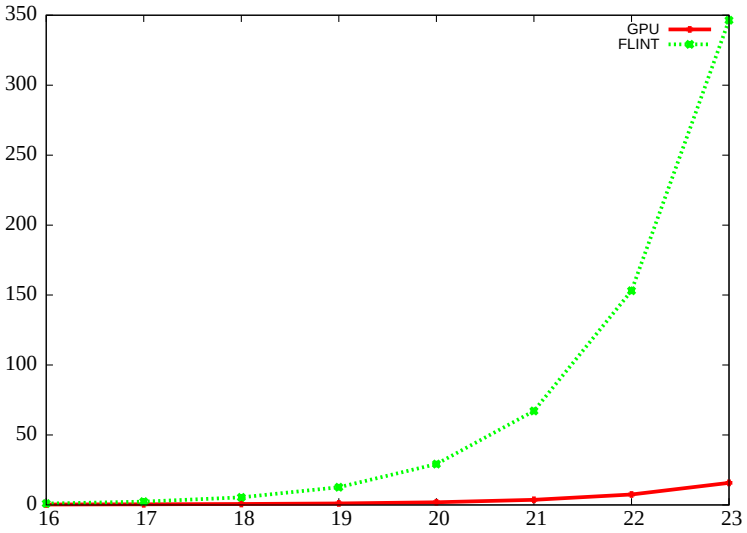

Figure 10.4: Evaluation higher degrees

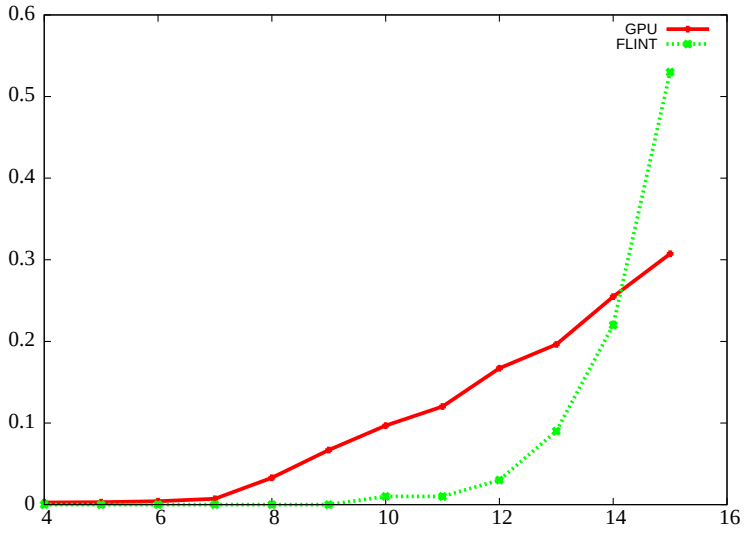

Figure 10.5: Interpolation lower degrees

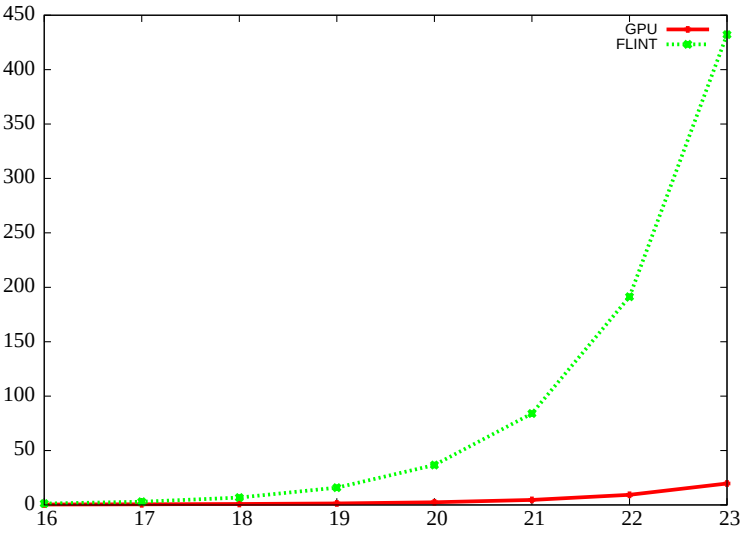

Figure 10.6: Interpolation higher degrees

Degree	Evaluation (GB/S)	Interpolation (GB/S)
4	0.0012	0.0013
5	0.0025	0.0026
6	0.0042	0.0045
7	0.0050	0.0060
8	0.0021	0.0029
9	0.0192	0.0318
10	0.0877	0.1228
11	0.2554	0.3403
12	0.5596	0.7054
13	1.2947	1.6182
14	2.5838	3.1445
15	5.2702	6.3464
16	9.6193	11.4143
17	16.4358	18.7800
18	22.6172	26.7590
19	32.3230	38.7674
20	40.4644	49.0012
21	46.7343	57.0978
22	50.8830	62.4516
23	52.9413	64.2464

Table 10.4: Effective memory bandwidth

One of the major factors of performance in GPU applications is usage of memory bandwidth. For our algorithm this factor is presented for every degree we are computing in the Table 10.4. The maximum memory bandwidth for our GPU is 148 GB/S.

10.8 Conclusion

We presented about implementation of fast evaluation and interpolation of univariate polynomials over a finite field on GPU architectures using techniques such as subproduct tree, subinverse tree, plain arithmetic, FFT-based arithmetic, etc. The results are showing good performance for our implementations, but rooms for improvements still exist; especially proposing new algorithm or better implementation for multiplications at levels 2^9 to 2^{13} of subproduct and interpolation trees instead of using FFT-based multiplications.

Chapter 11

Conclusion

This thesis has been devoted to the design and implementation of some basic routines in computer algebra targeting multi-core and many-core architectures. Driven by these motivations, we have developed new algorithms and implementations to support bivariate polynomial systems solving.

We have investigated and demonstrated, in Chapter 6, a new reordering algorithm for improving the data locality of basic routines dealing with vectors and sparse matrices. In each case, we re-arrange the input data and amortize the cost of this re-arrangement against the cost of calculations with the input data. We provide cache complexity analysis whose favorable results are confirmed experimentally. The cost for this preprocessing step can be easily amortized in conjugate gradient type algorithms. Our reordering algorithm for sparse matrices is based on a new integer sorting algorithm described in Chapter 5. This sorting algorithm can be implemented on a multi-core machine. One of the important key routine in this algorithm is the counting sort algorithm. We developed an efficient implementation of the counting sort algorithm which is cache-oblivious, in Chapter 4.

In Chapter 10, we propose parallel algorithms for performing subproduct tree construction, evaluation and interpolation and report on their implementation on many-core GPUs. We enhance the traditional algorithms for polynomial evaluation and interpolation based on subproduct-trees, by introducing the notion of a subinverse tree. For subproduct-tree operations, we demonstrate the importance of adaptive algorithms. That is, algorithms that adapt their behavior to the available computing resources. In particular, we combine parallel plain arithmetic and parallel fast arithmetic.

We have implemented a condensation method for computing the determinant of a matrix on many-core machine in Chapter 7.

Our proposed abstract computational model in Chapter 3, called *many core machine model* (MMM) is a simple model that captures all important features of a many-core machine. Our model combinines the fork-join and SIMD parallelisms, with an emphasis on estimating parallelism overheads, so as to reduce scheduling and communication costs in GPU programs. We have applied this model and successfully reduced parallelsim overheads for several basic routines in polynomial algebra.

As reported in Chapters 8 and 9, we have demonstrated that polynomial arithmetic with small degrees can be made efficient with plain arithmetic on many-core machines. For polynomial multiplication, our theoretical analysis allows us to reduce parallelism overheads due not only to data transfer but also to code divergence. For the Euclidean algorithm, our running time estimates match those obtained with the Systolic VLSI Array Model [9]. Meanwhile, our CUDA code implementing this optimized Euclidean algorithm runs within the same estimate analyzed by our model for input polynomials with degree up to 100,000.

In this research, we propose some new algorithms and a new computational model. We adopt some data structure and implementation tricks to have efficient implementations that are practical and efficient. We expect that these contributions will be of help to other researchers in high performance computing, parallel computing or symbolic computing areas in their works. All our GPU code is freely available in source at `www.cumodp.org`

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, 1974.

[2] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? In *Proceedings of the 27th annual ACM symposium on theory of computing*, STOC' 95, pages 427–436, New York, NY, USA, 1995. ACM.

[3] R. A. Arce-Nazario, E. Orozco, and D. Bollman. Reconfigurable hardware implementation of a multivariate polynomial interpolation algorithm. *Int. J. Reconfig. Comput.*, vol. 2010:2:1–2:14, 2010.

[4] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. Templates for the solution of linear systems: Building blocks for iterative methods. *SIAM, Philadelphia, PA*, 1994.

[5] L. A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal, 5:78–101*, 1966.

[6] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proceedings of the 25th annual ACM symposium on theory of computing*, STOC' 93, pages 362–371, New York, NY, USA, 1993. ACM.

[7] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, sep 1999.

[8] R. P. Brent, P. Gaudry, E. Thomé, and P. Zimmermann. Faster multiplication in GF(2)[x]. In *Proceedings of the 8th international conference on algorithmic number theory*, ANTS-VIII'08, pages 153–166, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] R. P. Brent and H. T. Kung. Systolic VLSI arrays for polynomial GCD computation. *IEEE Trans. Computers*, 33(8):731–736, 1984.

[10] R. P. Brent, H. T. Kung, and F. T. Luk. Some linear-time algorithms for systolic arrays. In *Proceedings of the IFIP Congress*, pages 865–876, 1983.

[11] A. Brown, editor. *VLSI Circuits and Systems in Silicon.* McGraw-Hill, Inc., New York, NY, USA, 1991.

[12] M. F. I. Chowdhury, M. Moreno Maza, W. Pan, and E. Schost. Complexity and performance results for non FFT-based univariate polynomial multiplication. In *AIP conference proceedings*, volume 1368, pages 259–262, 2011.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.).* MIT Press, 2009.

[14] C. L. Dodgson. Condensation of determinants. In *Proceedings of the Royal Society of London, 15*, pages 150–155, 1866.

[15] M. Frigo, P. Halpern, C. E. Leiserson, , and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proceedings of the 21st annual symposium on parallelism in algorithms and architectures, New York, NY, USA,*, SPAA '09, pages 79–90. ACM, 2009.

[16] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. In *Proceedings of the IEEE, special issue on program generation, optimization, and adaptation*, volume 93, number 2, pages 216–231, 2005.

[17] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th annual symposium on foundations of computer science*, FOCS '99, pages 285 – 297, 1999.

[18] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998.

[19] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *Proceedings of the 18th annual ACM symposium on parallelism in algorithms and architectures*, SPAA' 06, pages 271–280, New York, NY, USA, 2006. ACM.

[20] J. Gathen and J. Gerhard. *Modern Computer Algebra.* Cambridge University Press, 1999.

[21] J. Gathen and J. Shokrollahi. Efficient FPGA-based Karatsuba multipliers for polynomials over F_2. In *Proceedings of the 12th international conference on selected areas in cryptography*, pages 359–369. Springer-Verlag, Inc., 2006.

[22] P. B. Gibbons. A more practical PRAM model. In *Proceedings of the 1st annual ACM symposium on parallel algorithms and architectures*, SPAA' 89, pages 158–168, New York, NY, USA, 1989. ACM.

[23] P. B. Gibbons, Y. Matias, and V. Ramachandran. The Queue-Read Queue-Write PRAM model: Accounting for contention in parallel algorithms. *SIAM J. on Comput.*, 28(2):733–769, 1998.

[24] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. on Applied Mathematics*, 17(2):416–429, 1969.

[25] T. D. Han and T. S. Abdelrahman. Reducing branch divergence in GPU programs. In *Proceeding of GPGPU-4*, pages 3:1–3:8, 2011.

[26] Y. Han. Deterministic sorting in $o(nlog\,logn)$ time and linear space. In *Proceedings of the 34th annual ACM symposium on theory of computing*, STOC' 02, pages 602–608, New York, NY, USA, 2002. ACM.

[27] Y. Han and M. Thorup. Integer sorting in $o(n\sqrt{log\,logn})$ expected time and linear space. In *Proceedings of the 43rd annual IEEE symposium on foundations of computer science, 2002.*, pages 135–144, 2002.

[28] G. Hanrot, M. Quercia, and P. Zimmermann. The middle product algorithm I. *Appl. Algebra Eng., Commun. Comput.*, 14(6):415–438, March 2004.

[29] S. Haque. *A computational study of sparse matrix storage scheme.* M.Sc. thesis, University of Lethbridge, Canada, 2008.

[30] S. Haque and M. Moreno Maza. Determinant computation on the GPU using the condensation method. *Journal of Physics: Conference series*, 341(1):012031, 2012.

[31] S. Haque and M. Moreno Maza. Plain polynomial arithmetic on GPU. In *J. of Physics: Conference series*, volume 385, page 12014. IOP Publishing, 2012.

[32] M. Harris. Optimizing parallel reduction in CUDA (online document, 2007). `http://developer.download.nvidia.com`.

[33] W. Hart. Fast library for number theory: An introduction. In K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, editors, *mathematical software - Proceedings of the 3rd international congress on mathematical software, Kobe, Japan*, volume 6327 of *lecture notes in computer science*, pages 88–91. Springer, 2010.

[34] M. A. Hasan and V. K. Bhargava. Bit-serial systolic divider and multiplier for finite fields GF(2^m). *IEEE Trans. Comput.*, vol. 41, num. 8:972–980, 1992.

[35] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview scalability analyzer. In *Proceedings of the 22nd ACM symposium on parallelism in algorithms and architectures*, SPAA' 10, pages 145–156, New York, NY, USA, 2010. ACM.

[36] J. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the symposium on theory of computing*, pages 326 – 333, 1981.

[37] S. Hossain and T. Steihaug. Sparse matrix computations with application to solve system of nonlinear equations. *Wiley Interdisciplinary Reviews: Computational Statistics*, 5(5):372–386, 2013.

[38] E. Im. *Optimizing the performance of sparse matrix-vector multiplication.* PhD hesis, University of California Berkeley, USA, 2000.

[39] D. G. Kirkpatrick and D. G. Reisch. Upper bounds for sorting integers on random access machines. *Theoretical computer science*, 28:263–276, 1984.

[40] D. E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 3rd Edition.* Addison-Wesley Professional, 1997.

[41] D. E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 3rd Edition.* Addison-Wesley Professional, 1997.

[42] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching, 2nd Edition.* Addison-Wesley Professional, 1998.

[43] D. Kreher and D. Stinson. *Combinatorial Algorithms :Gen., Enum., and Search.* CRC Press, 1999.

[44] H. T. Kung and C. E. Leiserson. *Algorithms for VLSI processor Arrays. In Introduction to VLSI systems.* Addison-Wesley, Reading, MA, 1980.

[45] C. E. Leiserson. The Cilk++ concurrency platform. In *DAC' 09: In Proceedings of the 46th annual design automation conference, New York, NY, USA*, pages 522–527. ACM, 2009.

[46] X. Li, M. Moreno Maza, R. Rasheed, and E. Schost. The modpn library: bringing fast polynomial arithmetic into Maple. *J. Symb. Comput.*, 46(7):841–858, July 2011.

[47] W. Liu, W. Muller-Wittig, and B. Schmidt. Performance predictions for general-purpose computation on GPUs. In *Proceedings of international conference on parallel processing*, ICPP' 07, 2007.

[48] L. Ma, K. Agrawal, and R. D. Chamberlain. A memory access model for highly-threaded many-core architectures. In *Proceedings of international conference on parallel and distributed systems*, ICPADS' 12, pages 339–347, 2012.

[49] L. Ma and R. D. Chamberlain. A performance model for memory bandwidth constrained applications on graphics engines. In *Proceedings of the IEEE international conference on application-specific systems, architectures and processors*, pages 24–31. IEEE Computer Society, 2012.

[50] L. Mirsky. A dual of Dilworth's decomposition theorem. *The American Math. Monthly*, 78(8):876–877, 1971.

[51] P. L. Montgomery. Modular multiplication without trial division. In *mathematics of computation, 44(170)*, pages 519–521, 1985.

[52] P. L. Montgomery. *An FFT Extension of the Elliptic Curve Method of Factorization*. PhD thesis, University of California Los Angeles, USA, 1992.

[53] M. Moreno Maza and W. Pan. Fast polynomial arithmetic on a GPU. *J. of Physics: Conference series*, 256, 2010.

[54] M. Moreno Maza and W. Pan. Plain polynomial multiplication. *Technical Report, University of Western Ontario, Ontario, Canada*, 2010.

[55] M. Moreno Maza and W. Pan. Solving bivariate polynomial systems on a GPU. *J. of Physics: Conference series*, 341, 2012.

[56] M. Moreno Maza and Y. Xie. FFT-based dense polynomial arithmetic on multicores. *In high performance computing systems and applications. 23rd international symposium HPCS 2009, revised selected papers. LNCS 5976, Springer*, pages 378–399, 2009.

[57] M. Moreno Maza and Y. Xie. Balanced dense polynomial multiplication on multicores. *Int. J. Found. Comput. Sci.*, 22(5):1035–1055, 2011.

[58] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[59] A. Pinar and M. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the supercomputing' 99: Proceeding of the 1999 ACM/IEEE*, 1999.

[60] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. In *Proceedings of the IEEE, special issue on program generation, optimization, and adaptation*, volume 93, number 2, pages 232–275, 2005.

[61] N. Rahman and R. Raman. Analysing cache effects in distribution sorting. *J. Exp. Algorithmics*, 5, December 2000.

[62] G. Ruetsch and P. Micikevicius. Optimizing matrix transpose in CUDA. *NVIDIA Corporation*, 2009.

[63] A. Salem and K. Said. Condensation of determinants. Available at `http://arxiv.org/abs/0712.0822`.

[64] J. E. Savage. *Models of Computation.* Addison-Wesley Longman, Boston, MA, USA, 1998.

[65] J. E. Savage. *Models of Computation.* Addison-Wesley Longman, Boston, MA, 1998.

[66] L. J. Stockmeyer and U. Vishkin. Simulation of parallel random access machines by circuits. *SIAM J. Comput.*, 13(2):409–422, 1984.

[67] S. Tanaka, T. Chou, B. Yang, C. Cheng, and K. Sakurai. Efficient parallel evaluation of multivariate quadratic polynomials on GPUs. In *Proceedings of the*

13th international workshop on information security applications, pages 28–42, 2012.

[68] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Proceedings of supercomputing 92*, pages 578–587, 1992.

[69] S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM J. Res. Dev.*, 41(6):711–726, November 1997.

[70] J. Verschelde and G. Yoffe. Evaluating polynomials in several variables and their derivatives on a GPU computing processor. In *Proceedings of the 2012 IEEE 26th international parallel and distributed processing symposium workshops & PhD forum*, pages 1397–1405, Washington, DC, USA, 2012. IEEE Computer Society.

[71] R. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, University of California Berkeley, USA, 2003.

[72] M. Moreno Maza X. Li and W. Pan. Computations modulo regular chains. In *Proceedings of the 2009 international symposium on symbolic and algebraic computation, New York, NY, USA*, SPAA' 09, pages 239–246. ACM, 2009.

[73] D. Xavier, M. Moreno Maza, E. Schost, W. Wu, and Y. Xie. Lifting techniques for triangular decompositions. In *Proceedings of the 2005 international symposium on Symbolic and algebraic computation*, ISSAC' 05, pages 108–115, New York, NY, USA, 2005. ACM.

[74] A. N. Yzelman and R. H. Bisseling. Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods. *SIAM J. Sci. Comput.*, 31(4):3128–3154, July 2009.

Curriculum Vitae

Name:	**Sardar Anisul Haque**
Post-Secondary Education and Degrees:	The University of Western Ontario London, Ontario, Canada PhD in Computer Science, December 2013
	The University of Lethbridge Lethbridge, Alberta, Canada M.Sc. in Computer Science, December 2008
	Islamic University of Technology (IUT) Gazipur, Bangladesh B.Sc. in Computer Science and and Information Technology, October 2002
Work Experience:	Research Assistant, Teaching Assistant University of Western Ontario, London, Canada January 2009 - August 2013
	Research Intern (Mitacs-Accelerate) Maplesoft Inc., Waterloo, Ontario, Canada. September 2012 - April 2013

Honours and Awards:	Province of Ontario Graduate Scholarship, 2012-2013
	Queen Elizabeth II Graduate Scholarships in Science and Technology, 2011 - 2012
	The University of Western Ontario Biocomputing Student Award, May 2009
Selected Publications:	Determinant Computation on the GPU using the Condensation Method With Marc Moreno Maza Journal of Physics: Conference Series. 341 012031, 2012.
	Plain Polynomial Arithmetic on GPU With Marc Moreno Maza Journal of Physics: Conference Series. 385 012014, 2012.
	A Note on the Performance of Sparse Matrix-vector Multiplication with Column Reordering With Shahadat Hossain Proceedings of International Computing Conference held in Fullerton, California,April 2-4, 2009.