Combining Data Parallelism and Task Parallelism for Efficient Performance on Hybrid CPU and GPU Systems

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science (By Research)
in
Computer Science and Engineering

by

Aditya Deshpande
200802011
aditya.deshapande@research.iiit.ac.in

Center for Visual Information Technology
International Institute of Information Technology
Hyderabad - 500 032, INDIA
April 2014

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "Combining Data Parallelism and Task Parallelism for Efficient Performance on Hybrid CPU and GPU Systems" by Aditya Deshpande, has been carried out under my supervision and is not submitted elsewhere for a degree.

Date

Adviser: Prof. P J Narayanan

To my mother and father

Acknowledgments

This dissertation would not have been possible without the continuous guidance, support and help of my adviser Prof. P J Narayanan. The many discussions that I have had with him, on topics pertaining to research and otherwise, have not only taught me about algorithmic design, parallel computing, graphics and computer vision, but have also allowed me to understand scientific temper and research methodology. His open-door policy and the ability to always find time out of his busy schedule ensured that I was constantly able to drive my research forward. I am very grateful for everything that he has taught me and hope that I am able to put it to good use in the future.

I would like to thank my fellow lab mates and/or co-authors – Siddharth, Rajvi, Srinath, Saurabh, Parikshit, Chandrashekhar, Revanth, Krishna, Rohit, Pawan, Rohan, Harshit, Aditya and Apurva – with them I have had many intellectual discussions and they always ensured a jovial atmosphere in the lab. In my long stay at IIIT, I was fortunate enough to make many great friendships, which I hope will last a lifetime. Ankit, Aravindh, Archit, Harshit, Ishan, Kaustav, Mayank, Mihir, Mohit, Navni, Nikhil, Sanidhya, Saumay, Shipra, Varun, Vipul – I thank you for always being there for me, to celebrate my happy moments, to support me during the tough times. I would also like to thank my mother, father and brother, for it was they who put up with my erratic work schedules, with my long periods of not visiting home etc. It is only because of their support and encouragement that I find the strength to pursue my dreams. Lastly, I would like to thank all the professors, seniors, juniors, staff, vendors, everyone in IIIT Hyderabad, for you have inspired me by what you do, taught me many things, helped me and guided me at different points in my journey here.

Abstract

In earlier times, computer systems had only a single core or processor. In these computers, the number of transistors on-chip (i.e. on the processor) doubled every two years and all applications enjoyed free speedup. Subsequently, with more and more transistors being packed on-chip, power consumption became an issue, frequency scaling reached its limits and industry leaders eventually adopted the paradigm of multi-core processors. Computing platforms of today have multiple cores and are parallel. CPUs have multiple identical cores. A GPU with dozens to hundreds of simpler cores is present on many systems. In future, other multiple core accelerators may also be used.

With the advent of multiple core processors, the responsibility of extracting high performance from these parallel platforms shifted from computer architects to application developers and parallel algorithmists. Tuned parallel implementations of several mathematical operations, algorithms on graphs or matrices on multi-core CPUs and on many-core accelerators like the GPU and CellBE, and their combinations were developed. Parallel algorithms developed for multi-core CPUs primarily focussed on decomposing the problem into a few independent chunks and using the cache efficiently. As an alternative to CPUs, Graphics Processing Units (GPUs) were the other most cost-effective and massively parallel platforms, that were widely available. Frequently used algorithmic primitives such as sort, scan, sparse matrix vector multiplication, graph traversals, image processing operations etc. among others were efficiently implemented on GPU using CUDA. These parallel algorithms on the GPU decomposed the problem into a sequence of many independent steps operating on different data elements and used shared memory effectively.

But the above operations – statistical, or on graphs, matrices and list etc. – constitute only portions of an *end-to-end* application and in most cases these operations also provide some inherent parallelism (task or data parallelism). The problems which lack such task or data parallelism are still difficult to map to any parallel platform, either CPU or GPU. In this thesis, we consider a few such difficult problems – like Floyd-Steinberg Dithering (FSD) and String Sorting – that do not have trivial data parallelism and exhibit strong *sequential dependence* or *irregularity*. We show that with appropriate design principles we can find data parallelism or fine-grained parallelism even for these tough problems. Our techniques to break sequentiality and addressing irregularity can be extended to solve other difficult data parallel problems in the future. On the problem of FSD, our data parallel approach achieves a speedup of $10\times$ on high-end GPUs and a speedup of about $3-4\times$ on low-end GPUs, whereas previous work by Zhang et al. dismiss the same algorithm as lacking enough parallelism for GPUs. On string sorting, we achieve

a speedup of around $10 - 19\times$ as compared to state-of-the-art GPU merge sort based methods and our code will be available as part of standard GPU Library (CUDPP).

It is not enough to have a truly fine-grained parallel alogrithm for only a few operations. Any end-to-end application consists of many operations, some of which are difficult to execute on a fine-grained parallel platform like GPU. At the same time, computing platforms consist of CPU and a GPU which have complementary attributes. CPUs are suitable for some heavy processing by only a few threads i.e. they prefer task parallelism. GPUs is more suited for applications where large amount of data parallel operations are performed. Applications can achieve optimal performance by combining data parallelism on GPU with task parallelism on CPU. In this thesis, we examine two methods of combining data parallelism and task parallelism on a hybrid CPU and GPU computer system: (i) *pipelining* and (ii) *work sharing*. For pipelining, we study the Burrows Wheeler Compression (BWC) implementation in Bzip2 and show that best performance can be achieved by pipelining its different stages effectively. In contrast, a previous GPU implementation of BWC by Patel et al. performed all the tasks (BWT, MTF and Huffman encoding) on the GPU and it was $2.78\times$ slower than CPU. Our hybrid BWC pipeline performs about $2.9\times$ better than CPU BWC and thus, about $8\times$ faster than Patel et al. For work sharing, we use FSD as an example and split the data parallel step between CPU and GPU. The Handover and Hybrid FSD algorithms, which use work sharing to exploit computation resources on both CPU and GPU, are faster than the CPU alone and GPU alone parallel algorithms.

In conclusion, we develop data parallel algorithms on the GPU for difficult problems of Floyd-Steinberg Dithering, String Sorting and Burrows Wheeler Transform. In earlier literature, simpler problems which provided some degree of data parallelism were adapted to the GPUs. The problems we solve on GPU involve challenging sequential dependency and/or irregularity. We show that in addition to developing fast data parallel algorithms on GPU, application developers should also use the CPU to execute tasks in parallel with GPU. This allows an application to fully utilize all resources of an end-user's system and provides them with maximum performance. With computing platforms poised to be predominantly hetergoneous, the use of our design principles will prove critical in obtaining good application level performance on these platforms.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In earlier times, computer systems had only a single core or processor. The number of transistors on-chip (i.e. on the processor) doubled every two years. Intelligent use of these transistors by computer architects led to faster hardware. The clock speed of these processors kept improving. As a result, all applications enjoyed free speedup (popularly known as *free ride* [53]) every few years. This phenomena was termed as the Moore's Law. With more and more transistors being packed on-chip, power consumption became an issue, frequency scaling reached its limits and industry leaders eventually adopted the paradigm of multi-core processors [3]. Computing platforms of today have multiple cores and are parallel. CPUs have multiple identical cores. A GPU with dozens to hundreds of simpler cores is present on many systems. In future, other multiple core accelerators may also be used.

With the advent of multiple core processors, the responsibility of extracting high performance from these parallel platforms shifted from computer architects to application developers and parallel algorithmists. Tuned parallel implementations of several mathematical operations, algorithms on graphs or matrices on multi-core CPUs and on many-core accelerators like the GPU and CellBE, and their combinations were developed [4, 36, 45, 82]. Intel Math Kernel Library [36] was developed as a suite of standard operations viz. fast fourier transform, LU, Cholesky and QR decomposition, random number and probability distribution generator, splines and interpolation etc. that made the best use of multi-core CPU to provide optimum performance. Parallel algorithms developed for multi-core CPUs primarily focussed on decomposing the problem into a few independent chunks and using the cache efficiently. The development of a framework like CUDA for GPUs, made it possible to express data parallelism easily. This allowed computation resources on graphics processors to be useful for general purpose operations. Frequently used algorithmic primitives such as sort [70, 14, 28, 27], scan [70], sparse matrix vector multiplication [6], graph traversals [29] among others were efficiently implemented on GPU using CUDA. A suite of statistical, data structures, arithmetic, image, signal processing related primitives for the GPU, similar to Intel MKL, was developed and distributed with NvPP [59] and other libraries viz. Thurst [32], CUDPP [16]. These parallel algorithms on the GPU decomposed the problem into a sequence of many independent steps operating on different data elements. These steps could be per-

formed on multiple simple cores of the GPU and they used the shared memory (for every small group of threads) effectively.

But the above operations – statistical, or on graphs, matrices and list etc. – constitute only portions of an *end-to-end* application and in most cases these operations also provide some inherent parallelism (task or data parallelism). Matrix operations like QR decomposition, matrix multiplication or sparse matrix vector multiplication involved performing same operations across different data elements of the matrix. Though the nodes vary at every step, many graph traversals involved performing the same operations on different nodes without any dependence between the nodes at every step. Many image processing operations such as filtering, color conversion, fast fourier transform were by its very nature data parallel, i.e. operations on all pixels were independent of each other. This led to these operations being amongst the ones adapted early to all available parallel platforms. But, the problems which lack such task or data parallelism are still difficult to map to any parallel platform, either CPU or GPU. In this thesis, we consider a few such difficult problems – like Floyd-Steinberg Dithering and String Sorting – that do not have trivial data parallelism and exhibit strong *sequential dependence* or *irregularity*. We show that with appropriate design principles we can find data parallelism or fine-grained parallelism even for these tough problems. Also, for high performance in applications, it is not sufficient to have only a data parallel algorithm for some step. An application will consist of many other sequential steps. The best performance can be achieved by combining the data parallel step on the GPU with other steps on CPU using task parallelism, as is demonstrated for Floyd-Steinberg Dithering and Lossless Data Compression in this thesis. In addition to finding data parallelism for difficult problems, the focus of this thesis is to examine the techniques of achieving the right mix of data parallelism and task parallelism on a hybrid CPU and GPU computer system. By a hybrid CPU and GPU system, we mean a computer system which consists of a multi-core CPU and a many-core GPU. In Section 1.1 we examine the design principles to obtain data parallelism on difficult problems and Section 1.2 talks about our methods to combine data parallel algorithm with task parallelism for maximum performance on a hybrid CPU and GPU system.

1.1 Design Principles for Data Parallelism

The difficulty to get good performance on end-to-end application using data parallelism results from the fact that most applications have operations which are *inherently sequential* or *irregular*. We develop design principles to break sequentiality (Section 1.1.1) and address irregularity (Section 1.1.2). On problems that are inherently sequential, it is difficult to identify independent operations on different data elements. For irregular problems, it is difficult to allocate appropriate quantity of work to every thread such that the load is balanced. Thus, developing fast implementations which exploit the parallelism of many-core GPU is difficult for problems with sequentiality and irregularity. They lack the inherent parallelism that was found in simpler operations on matrices, signals and graphs which were ported to GPUs easily. In this thesis, we develop data parallel algorithms and some design principles for these

difficult problems. Our design principles allow us to extract fine-grained parallelism, even on problems which show the characteristics of sequentiality and/or irregularity.

1.1.1 Breaking Sequentiality

The Error Diffusion Dithering application, specifically Floyd-Steinberg Dithering (FSD) algorithm, exhibits sequentiality. In Chapter 3, we develop parallel algorithm for FSD and solve the problem of sequentiality. We note that, though error distribution scheme of FSD implies a dependency of even the last pixel on the first, our analysis allows us to find pixels that can be processed independently. This error distribution scheme imposes a constraint that pixels separated by a knight's move can be processed independently. We start with a wavefront at top left of the image and process a knight's move pixel boundary per iteration in parallel. We perform these parallel processing steps on the independent pixels till we reach the bottom right of image (and achieve the final output). Our data parallel approach achieves a speedup of $10\times$ on high-end GPUs and a speedup of about $3-4\times$ on low-end GPUs (Figure 3.10 and 3.12). The pattern of data dependency that is found in FSD is commonly observed in many dynamic programming problems. Most Dynamic Programming problems involve filling a matrix such that any element is dependent on some or all of its neighbors and the final answer is obtained after processing the last element. The operations at each element can also be non-linear. This is similar to the threshold step of FSD. Techniques, similar to the ones developed by us viz. tiling, coalesced memory accesses etc. have been applied for mapping the problems of Dynamic Programming to the GPU [80]. Thus, beyond high performance on the application of FSD, the use of our techniques can also help accelerate many Dynamic Programming applications using GPU. There are also other image processing operations such as causal filtering, summed area tables etc. which exhibit the same sequentiality as FSD. Our techniques can be extended to develop efficient solutions for these problems.

1.1.2 Addressing Irregularity

String sorting involves sorting long and mostly variable length keys. Since any two given keys match to different lengths, the work performed by any two threads is not uniform. Also, arbitrary memory accesses need to be performed depending on the ordering between the input keys. Algorithms where work given to different threads is different and the memory accesses are arbitrary depending on the input, are popularly known as irregular algorithms. Thus, string sorting exhibits the characteristics of an irregular algorithm. In Chapter 4, we develop a fast parallel algorithm for this irregular problem of string sorting. The approach we develop involves mapping the operations of string sort to fast standard primitives of fixed-length sort, scatter and scan. Highly tuned implementations are available on the GPUs for all these standard primitives and their use provides high performance in any application. The challenges of irregularity are efficiently handled within these primitives. For example, in radix sort developed by Merrill and Grimshaw [55], they carefully design it to be compute bound and hide the latency incurred

on account of arbitrary memory accesses during shuffling of buckets. We leverage the intelligent design of primitives and map the original irregular problem to a sequence of steps involving only standard primitives (which have an efficient solution). This primitive-based approach will allow for fast development of parallel and high performing applications for many other irregular problems. Also, any improvements to the primitives will be directly inherited by these applications without requiring re-design. Our string sort approach achieves a speed up of around $10 - 19\times$ as compared to state-of-the-art GPU merge sort based methods on difficult datasets (Table 4.2). We extend our string sorting algorithm to efficiently solve another irregular problem of Burrows Wheeler Transform in Chapter 5. Our Burrows Wheeler Transform implementation is the first to achieve speed up on the GPU (Figure 5.5).

1.2 Combining Data Parallelism and Task Parallelism

It is not enough to have a truly fine-grained parallel algorithm for only a few operations. Any end-to-end application consists of many operations, some of which are difficult to execute on a fine-grained parallel platform like GPU. At the same time, computing platforms consist of CPU and a GPU which have complementary attributes. CPUs are optimized for running a sequential code, most of the hardware (or transistors) are dedicated to finding instruction level parallelism, perform branch prediction etc. CPUs are suitable for some heavy processing by only a few threads i.e. they prefer task parallelism. The sequential portions of the application should thus be performed on CPUs. In GPUs, most transistors are used for more direct problem solving purposes. Thus GPUs offer high performance benefits using all these transistors for data parallel operations. Applications can achieve optimal performance by combining data parallelism on GPU with task parallelism on CPU. In this thesis, we examine two methods of combining data parallelism and task parallelism on a hybrid CPU and GPU computer system: (i) *pipelining* (Section 1.2.1) and (ii) *work sharing* (Section 1.2.2).

1.2.1 Pipelining

As discussed earlier, an end-to-end application is typically made up of many tasks. Not all tasks are amenable for fine-grained parallel processing by the GPU. An end-to-end application can achieve maximum throughput by mapping each of its tasks to the appropriate compute platform and overlapping them efficiently. This task scheduling or pipelining leads to optimal resource utilization and provides maximum throughput. In Chapter 5, we take a look at the commonly used end-to-end application of Lossless Data Compression. We study the Burrows Wheeler Compression (BWC) implementation in Bzip2 and show that best performance can be achieved by pipelining its different stages effectively. For compute intensive tasks (i.e. BWT) we develop a data parallel algorithm on GPU and we perform other tasks (merge, MTF, Huffman encoding) on CPU. We efficiently overlap the CPU and GPU computations and ensure that resources do not remain idle. In contrast, a previous GPU implementation of BWC by

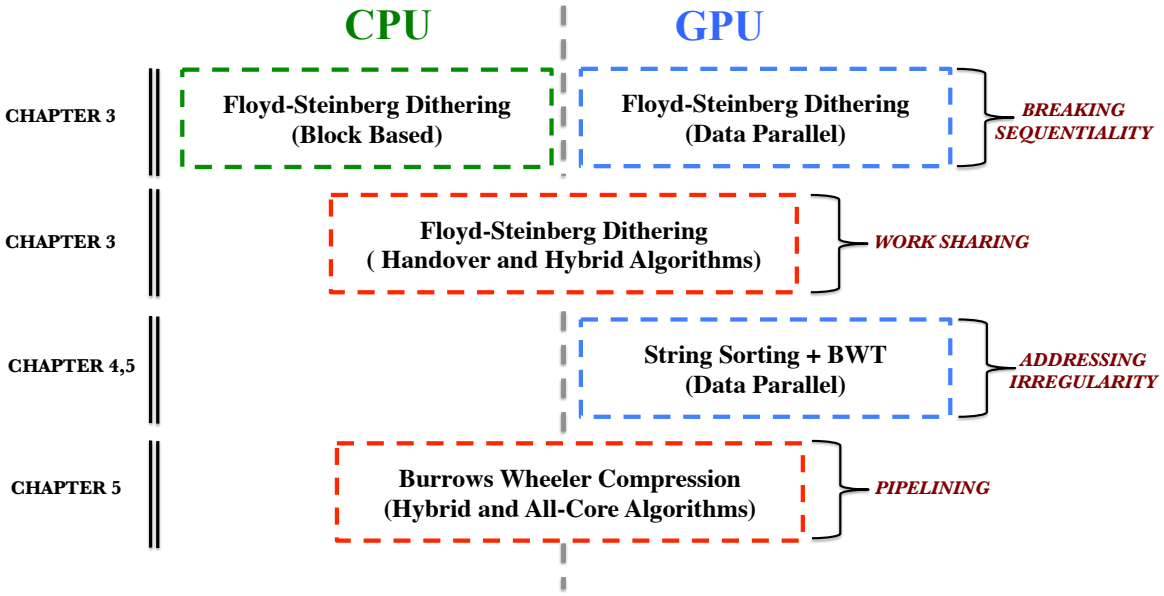


Figure 1.1: Outline of the Thesis.

Patel et al. [62] performed all the tasks (BWT, MTF and Huffman encoding) on the GPU. Each of the task was slower on the GPU and their resulting implementation was about $2.78\times$ slower than CPU. Using our pipelining strategies we develop a hybrid BWC and achieve a $2\times$ speedup over CPU on the same problem (Figure 5.6). This hybrid BWC approach uses only a single CPU core, whereas the CPU still has more cores. To use them effectively, we perform BWC tasks on a few blocks (using best sequential implementation of BWC) on idle CPU cores in parallel with our hybrid BWC. We call this the all-core BWC, which uses all cores of the CPU as well as the many core GPU. We improve nearly $2\times$ speedup using hybrid BWC to $4.87\times$ with our all-core BWC on a high-end platform. Detailed results for all-core BWC could be found in Table 5.2 and 5.3.

1.2.2 Work Sharing

If the data-parallel step is performed on the GPU alone and the problem does not have enough number of other tasks, the CPU will remain idle. In such a scenario, we can split the data parallel step between CPU and GPU itself. This leads to none of the resources remaining idle and provides better throughput for the problem at hand. In Chapter 3, for the problem of FSD, we first develop a data parallel algorithm using data-dependency to our advantage. We observe that the parallelism is low towards the start and end for this data parallel algorithm. We solve these low parallelism portions on the CPU and GPU solves only the high parallelism portions. This is called the Handover FSD algorithm. We further extend this to the Hybrid FSD algorithm. In Hybrid FSD algorithm, the CPU does a portion of the work even when GPU is operating in the high parallelism region. This Hybrid algorithm involves concurrent processing by both CPU and GPU. Though a transfer of few bytes of memory is required per iteration to keep the

CPU and GPU synchronized, we show that the overall performance improves even with the memory transfer overhead. Since we split the same data parallel FSD step across CPU and GPU, we call this work sharing instead of pipelining. This extension is also possible for the problem of string sorting. We can split the buckets for sorting between CPU and GPU, either at the start or after every iteration. The buckets could be split depending on the relative speeds of processing of the two processors. This string sort which uses both CPU and GPU will provide even better performance than the current GPU alone algorithm. Similar to string sort, our idea of work sharing can be extended to other data parallel algorithms in the future. Today, there is an increased interest in unifying the virtual memory address space for CPU and GPU. This unification will make memory transfer between CPU and GPU even faster making work sharing data parallel algorithms necessary for high performance.

1.3 Outline

The outline of this thesis is given in Figure 1.1. We develop data parallel algorithms to address sequentiality and irregularity. We combine these data parallel algorithms with task parallelism to get the best results on a hybrid CPU and GPU computer system. Chapter 3 has details of our data parallel algorithm for Floyd-Steinberg Dithering (FSD) which exhibits the characteristic of sequentiality. Chapter 4 describes our data parallel algorithm for String Sorting which is an irregular algorithm. The String Sorting algorithm is further extended to perform difficult Burrows Wheeler Transform (BWT) and then finally, Burrows Wheeler Compression (BWC) in Chapter 5. We show that for BWC, efficient pipelining of tasks is critical to achieve good performance. Chapter 3 also describes our techniques for splitting the data parallel algorithm of FSD and performing some work on the CPU using work sharing. In both cases, FSD and BWC, the algorithms that use both the CPU and GPU efficiently (indicated by red boxes in Figure 1.1) give the best performance.

1.4 Contributions

The major contributions of this thesis are as follows:

1. We develop data parallel algorithms on the GPU for difficult problems of Floyd-Steinberg Dithering, String Sorting and Burrows Wheeler Transform. In earlier literature, simpler problems which provided some degree of data parallelism were adapted to the GPUs. The problems we solve on GPU involve challenging sequential dependency and/or irregularity.

2. We develop a coarse parallel algorithm for FSD on CPU and a data parallel algorithm for FSD on GPU. We utilize the complementary attributes of CPU and GPU to share the work of data parallel step between both. The Hybrid and Handover FSD algorithms which perform this work sharing achieve better performance than the CPU only and GPU only parallel FSD algorithms.

3. The string sorting algorithm developed by us outperforms the state-of-the-art string sorting approach on GPU by a large margin. String sorting is a vital primitive in many applications like compression, pattern matching, data mining etc. and our work will help accelerate them all.

4. We use our GPU string sort and develop additional techniques on top of it to perform suffix sort step of Burrows Wheeler Transform (BWT) on GPU. Our BWT implementation achieves speedup on GPU for the first time.

5. We efficiently pipeline the GPU BWT step with other steps of Burrows Wheeler Compression (BWC) on CPU. Our pipelined BWC implementation outperforms the highly tuned CPU implementation in Bzip2. BWC is routinely used by end-users and our code will be beneficial to them.

6. We show that in addition to developing fast data parallel algorithms on GPU, application developers should also use the CPU to execute tasks in parallel with GPU. This allows an application to fully utilize all resources of an end-user's system and provides them with maximum performance.

Chapter 2

GPU and Hybrid Computing

Graphics applications traditionally required high computation and they also exhibited data parallelism in all their operations. Dedicated hardware called Graphics Processing Units (GPUs) were developed to serve the needs of these graphics applications. The operations performed during graphics pipeline were fixed and could be hardwired into the graphics processor. Thus, initial GPUs were fixed function and allowed no flexibility in performing computations. A few graphics systems viz. Pixar's RenderMan showed the benefits of allowing flexibility in computations. With more flexibility being added to GPUs, general purpose computations also became possible on graphics processors. Graphics Processing Units (GPUs) became cost-effective, massively parallel platforms available. Researchers have thus always been interested in harnessing their power for general purpose computation. Prior to development of frameworks such as CUDA [58], people programmed the shaders in a graphics pipeline intelligently to perform general purpose computations. With CUDA it became easier to express data parallelism and exploit the computation resources on the graphics processor for general purpose computations. Since then there has been growing interest in developing data parallel algorithms on the GPU platform. In this chapter, we will first look at data parallel algorithms that were developed only for the GPU alone. The work done in this area falls in the category of GPU Computing (Section 2.1). After that, we also look at algorithms that used both GPU and the controlling CPU for additional performance benefits. This paradigm of using CPU along with its co-processor GPU for computation is popularly known as Hybrid Computing (Section 2.2). This chapter only consists related work for the broader areas of GPU and Hybrid computing, the relevant related work for each chapter has been discussed in each of the chapters. The techniques we develop to address irregularity and break sequentiality and solve problems of Floyd-Steinberg Dithering and String Sorting, are a contribution to the area of GPU Computing. Our work in pipeline and work-sharing for fast performance on Burrows Wheeler Compression and Floyd-Steinberg Dithering, extends the ideas in Hybrid Computing literature.

2.1 GPU Computing

Horn developed the first scan primitive for GPU architecture [35]. They used the Brook programming language (predecessor to CUDA, OpenCL) to implement their scan, which provided limited features. They proposed more flexibility required from the GPU architecture such as central running sum registers and ability to send data to other parts of frame buffer etc. This and other similar work (on GPUs with limited support for data parallelism) were important in arriving upon ideas that constituted a useful data parallel framework like CUDA. Sengupta et al. later developed the first work optimal $O(N)$ scan algorithm on the GPU. They exploited the features offered by CUDA viz. shared memory, synchronized thread warp etc. for high performance. Their scan primitive was useful in developing fast solutions for many other problems such as quicksort, sparse matrix vector multiplication and tridiagonal matrix solver [70].

Sorting is a fundamental operation in computer science. It was critical to develop sorting algorithms for the GPU. Sorting is a frequently occurring primitive in most applications and sending data to-and-fro between CPU and GPU for every sort operation (so that sort could be performed on CPU) would be an expensive overhead. It would have hampered the use of GPUs for general purpose computation. Thus, there was high interest and vast literature for developing sorting algorithms on the GPU platform. The earliest sorting algorithms on GPUs involved using a sorting network which solved the problem in fixed steps and with fixed parallelism [9, 13, 40, 64]. After this, Kipfer and Westermann [41] made better use of GPUs computational resources and showed improvements over the bitonic sort of Buck and Purcell [9]. Their sorting algorithm preserved the sorted order of nearly sorted arrays, thus offering additional benefits. Sengupta et al. [70] using their scan primitive, for the first time, developed a CUDA accelerated GPU version of quicksort and radix sort. Their quicksort was much slower than radix sort. Then, Cederman and Tsigas [14] developed a fast quicksort on GPU that outperformed the previous state-of-the-art quicksort and bitonic sort implementations of Sengupta et al. [70] and Kipfer and Westerman [41] respectively . They minimize the book keeping and inter-block synchronization required in earlier GPU quicksort for fast performance. Satish et al. developed the fastest radix sort on GPU which outperformed an 8-core CPU by $4\times$ [69]. They carefully designed their radix sort to have high fine-grained parallelism and they also reduced the global communication required between threads. In the same article, Satish et al. also presented a GPU merge sort algorithm. Though, their GPU merge sort was slower than radix sort, it was still the fastest comparison based sorting algorithm presented. Merrill and Grimshaw improved upon the radix sort of Satish et al. by mapping the operations to fast scan primitives [56]. Their radix sort adjusted its granularity (digit size) adaptively to fit the underlying GPU architecture. They introduced kernel fusion and serialization of threads for appropriate steps to reduce the global memory accesses, which have been helpful techniques to accelerate many other applications. Their radix sort is the fastest sort implementation till date and it is available as part of Thrust Library [32]. Sample sort was also implemented on GPUs [43, 63]. External memory sorting algorithms were also developed for the GPU [26].

Apart from sorting, many other linear algebra operations: LU decomposition, sparse matrix multiplication etc. and graph operations: BFS, APSP, SSSP etc. were accelerated using the massive parallelism offered by GPUs. Bell and Garland developed a sparse matrix-vector multiplication (SpMV) algorithm on GPU that outperformed the previous state-of-the-art quad-core CPU runtime by $10\times$ [6]. Subsequent efforts also improved the performance on SpMV on GPUs [51, 65]. GPUs were also used for accelerating operations on graphs. Harish et al. demonstrated fast implementation of BFS, SSSP, APSP using GPUs. Buluc et al. improved the performance of these path problems on GPUs in their work [10]. Fast algorithms for operations on linked list such as list ranking [66] and prefix computation [78] were developed using CUDA. With time many such primitives from different domains were accelerated on GPUs. Their code was made available to be used as part of standard libraries such as cuBLAS, cuFFT, NPP, Magma, cuSparse, Thrust, GPP (computational geometry). More details about these libraries and primitives could be found at `https://developer.nvidia.com/gpu-accelerated-libraries`.

2.2 Hybrid Computing

A mix of processors are already present on today's computer. Future computer systems are bound to maintain similar heterogeneity of processors. Thus, efficient use of these heterogeneous platforms will be critical for obtaining runtime benefits on applications. In this section, we will discuss work in hybrid computing that has been developed to utilize all resources on these heterogeneous computing platforms effectively. Since heterogeneous systems with CPU and GPU are more prevalent, most work we will discuss uses these systems as target heterogeneous platforms.

Tomov et al. did some pioneer work on using hybrid CPU and GPU systems for additional performance gains [75]. They presented a hybrid LU factorization algorithm for a multi-core CPU + GPU system. The performance benefits offered by extension of a previous GPU-alone LU factorization to a hybrid CPU+GPU algorithm made a strong case for looking at CPU and GPU as co-processors to solve a problem. Their algorithm performs balanced processing on both CPU and GPU for best performance. They also address the communication issues observed when using the CPU and GPU together. In subsequent work, Agullo et al. developed new CPU/GPU hybrid LU factorization kernels. On a multi-GPU and CPU system they achieved a 1TFlops/s LU factorization throughput.

Choudhary et al. develop a fast implementation for the compute intensive sparse bundle adjustment on a hybrid CPU and GPU computer system [15]. They completely overlap the expensive computations such as Jacobian and Schur complement on the GPU with other simpler operations on the CPU viz. L2-error from error vector etc. They achieve about $30 - 50\times$ speedup over CPU alone implementation. Their work shows the importance of pipeling the tasks appropriately for good performance on a CPU and GPU system. We extend this pipelining technique in our work to get high performance on common end-to-end applications. Kumar et al. [50] in their work investigate different heuristics to split the work between CPU and GPU for the problem of sparse matrix multiplication. They show a much improved

performance over Intel MKL using a GPU alone. On a hybrid CPU and GPU computer system they are able to improve their GPU alone speedup of $3.5\times$ (over Intel MKL) further to $5.5\times$. Banerjee and Kothapalli develop hybrid algorithms for the problems of list ranking and graph connected components [5]. These hybrid algorithms offer performance benefits of 50% and 25% respectively over the best known GPU alone implementations for the corresponding problems. They perform appropriate task mapping and discuss the larger issues of synchronization between the CPU and and GPU in their work.

Leist et al. [44], Hawick and Playne [30] examine the interplay of cross-calling kernels and host components on a hybrid multi-core CPU and GPU system for graph algorithms of clustering co-efficients and component labelling respectively. Hong et al. propose a method for BFS where they choose the best implementation at each step: a simple sequential execution or multi-core execution or GPU execution [34]. Their algorithm uses the best platform for every step on a hybrid system and gives good performance. This is similar to the idea of work sharing a data parallel step between both CPU and GPU. The RSA encryption also showed improved throughput when the idle CPU cores were used for computation along with the GPU [22]. A comprehensive survey detailing many applications and benefits of hybrid computing could be found in [42].

Chapter 3

Error Diffusion Dithering

Many image filtering operations have ample parallelism. This is because each input pixel creates one output pixel value, either independently or by processing a small neighborhood around itself in the input image. The fact that output pixels depend only on a few input pixels, and that the latter do not change as a result of processing, allows all pixels to be processed in parallel. Such embarrassing parallelism is not available if the operation has a causal dependence on the output of previous pixels. In such a scenario, the output value of "later" output pixels depend on a mix of input pixel values and values of output pixels computed "earlier". A causal processing order for 2D images can be defined typically by choosing one corner as the first pixel and its opposite corner as the last in processing order. The processing can then proceed along rows or along columns. The processing step on each pixel may be a linear operation or can involve non-linear computations. Linear operations are easy to parallelize, as the output at each pixel can be written as a linear combination of the input values of some or all "past" pixels. The pixel with the longest dependency decides the running time. This may not be possible if the operation is non-linear. The operations then need to be performed strictly in order, leading to sequential dependence, as a closed form expression is not possible. This potential dependency of even the last pixel on the very first makes such operations inherently *sequential* and *difficult* to parallelize. These algorithms thus exhibit the characteristic of *sequentiality* discussed in Section 1.1.1 and we develop a data parallel algorithm for it. As part of this thesis, in addition to the data parallel algorithm, we develop algorithms with the right mix of data parallelism (on GPU) and task parallelism (between CPU and GPU). The use of our techniques can help solve other problems which exhibit similar sequentiality.

A typical example of the progressive non-linear operation discussed above is *Error Diffusion Dithering*. Several other image processing operations involve non-linear, progressive processing of pixels viz causal filtering. But, we use Error Diffusion Dithering as the sample application, because optimal error diffusion algorithm poses the maximum challenge for a parallel implementation. As shown in Figure 3.1, dithering is a technique used to create an illusion of a higher color depth in images using a limited color palette. It provides a much better approximation using a limited color palette compared to the naive thresholding methods. It approximates other colors not available in the color palette using a spatial distribution of available colors, as the human visual system averages the colors in a neighbor-

(a) Input Image (256 Colors) **(b) Threshold Image (2 Colors)** **(c) Dithered Image (2 Colors)**

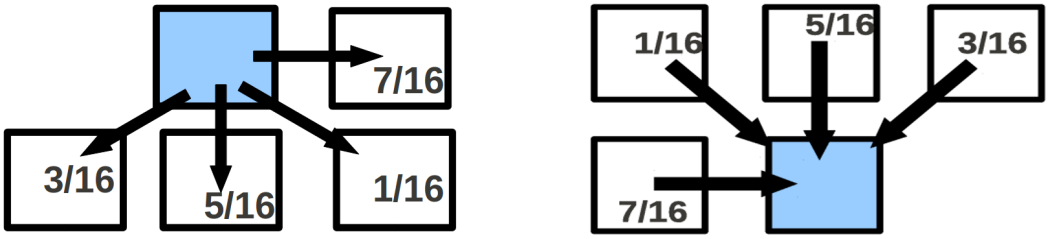
Figure 3.1: The Dithered (2 Color) Image is a much better approximation of Input (256 Color) Image as compared to naive Threshold (2 Color) Image.

hood. Applications of dithering vary from printing, display on small devices like cellphones, computer graphics [77], visual cryptography [46], image compression [74], etc. Among the class of dithering algorithms, error diffusion dithering algorithms are popular due to their high quality output. The Floyd-Steinberg Dithering (FSD) is an optimal error diffusion algorithm [23]. It generates better results than other dithering methods. Its traditional sequential CPU implementation has $O(mn)$ time complexity (m, n being the height and width of the image in pixels). The error distribution scheme (Figure 3.2(a)) processes pixels from left to right and top to bottom. This introduces a sequential dependency of the last pixel (bottom right) on the first (top left). Consequently, FSD problem is difficult to parallelize [57].

In this chapter, we present strategies to perform FSD on multi-core CPUs (Section 3.3), many-core GPUs (Section 3.4), as well as a combination of CPU and GPU (Section 3.5). Our results will show the benefits of combining data parallelism and task parallelism (i.e. using both CPU and GPU available on a given system). We show that high performance can be obtained by using appropriate, problem-dependent data layout and by minimizing communications, especially in a hybrid (CPU+GPU) setting. The hybrid algorithm is especially promising on low-end computers used by millions of people, such as a laptop or an off-the-shelf commodity desktop with a decent CPU and a mid or low-end GPU. Practical parallel applications have to focus on such low-end platforms to bring the benefits of parallelism to a huge number of common users and not merely the users of expensive HPC systems.

3.1 Background and Related Work

We will briefly discuss the Floyd-Steinberg Dithering algorithm (Section 3.1.1) and previous efforts to develop parallel algorithms for it (Section 3.1.2).



(a) Flow of errors outside from the center pixel

(b) Flow of errors into any pixel from neighboring pixels

Figure 3.2: Flow of errors in Floyd-Steinberg Dithering.

Algorithm 1 Sequential Floyd-Steinberg Dithering

Input: Image $\mathbf{I_{in}}$
Output: Image $\mathbf{I_{out}}$
for $i = 1 \rightarrow n$
  for $i = 1 \rightarrow m$
    $\mathbf{I_{out}}(i, j)$ =NearestColor($\mathbf{I_{in}}(i, j)$)
    $err = \mathbf{I_{in}}(i, j) - \mathbf{I_{out}}(i, j)$
    $\mathbf{I_{in}}(i, j+1) \quad += err \times \frac{7}{16}$
    $\mathbf{I_{in}}(i+1, j-1) \quad += err \times \frac{3}{16}$
    $\mathbf{I_{in}}(i+1, j) \quad += err \times \frac{5}{16}$
    $\mathbf{I_{in}}(i+1, j+1) \quad += err \times \frac{1}{16}$
  end **for**
end **for**

3.1.1 Floyd-Steinberg Dithering Algorithm

The basic FSD approach proceeds as follows: For each pixel, the accumulated error from its neighbors (explained later) and its own pixel value is summed. This value is quantized to the nearest value available in the color palette. The residual quantization error is then distributed to the neighboring pixels in fractions shown in Figure 3.2(a). Each pixel receives quantization errors from its neighbors and these form the pixels accumulated error (Figure 3.2(b)). The process starts with the top-left (or first) pixel. The quantization at each pixel makes the output a non-linear function of all past pixel values. The algorithm is summarized in Algorithm 1.

3.1.2 Previous work on Parallel FSD

Metaxas [57] was one of the first to parallelize Error Diffusion Dithering algorithm. They parallelized the FSD algorithm with $2m + n$ parallel steps compared to mn required for the sequential version, for an $m \times n$ image. An optimal scheduling order ensured that a pixel was processed as soon as all its data dependencies were met. This scheduling order is same as the knights chessboard move

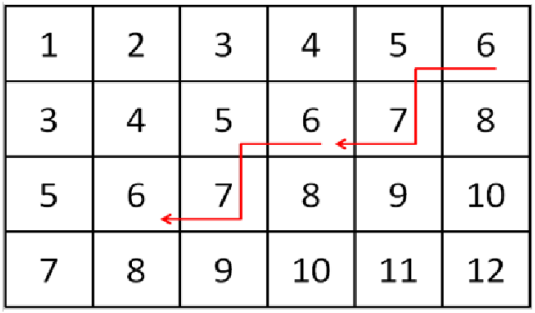


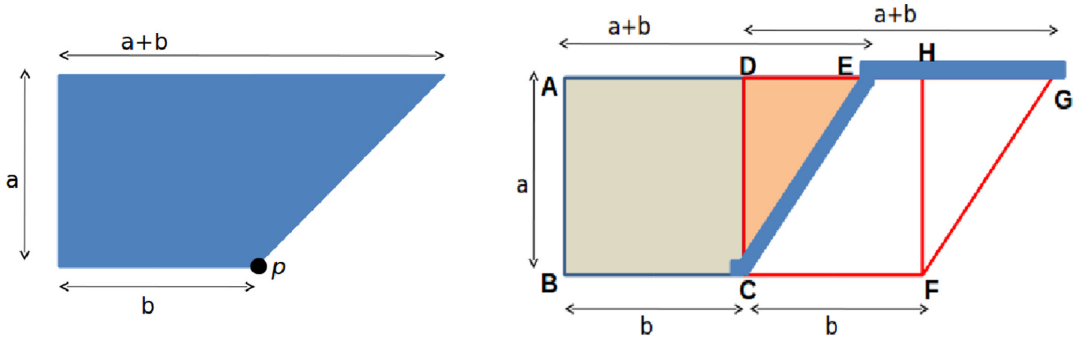
Figure 3.3: Each box is a pixel and label n denotes the iteration in which the pixel can be scheduled for processing. All pixels of same label are independent of each other.

pattern that we discuss later. The use of a linear array of processors was proposed, with each processor processing three pixels in a row, followed by the pixels in the next row. After a processor processed three pixels, it transmitted errors to the neighboring processor and activated them for processing. This effort was mainly directed at the PRAM processing model and is not scalable to the many-core (GPU) processing model of today.

Zhang et al. took an altogether different approach to parallelizing Error Diffusion Dithering [81]. They dismissed the FSD algorithm for dithering due to the low possible parallelism. They used a pinwheel dithering algorithm [45] instead, which has much more inherent parallelism. The image is divided into blocks of two types in a checkerboard fashion. All blocks of the same type can be processed in parallel and independently of each other. First, blocks of one type were processed followed by processing the other type of blocks. Metrics that take into account the human visual system show that the raster-order FSD approach provides better results compared to the serpentine order error diffusion used in the pin-wheel algorithm [33]. We parallelize the original FSD algorithm as it has perhaps the most challenging data dependency pattern that reduces parallelism. Thus, techniques developed for it and the lessons learned from it will be useful for many operations that are currently considered difficult for parallel processing

3.2 Finding Parallelism in FSD

It is interesting to see the flow of errors in FSD. Each pixel (except the ones on the border) needs the error values from 4 neighboring pixels before it can be processed (Figure 3.2(b)). Edge pixels depend on fewer neighboring pixels as others are outside the image boundary. Thus, for any pixel, a maximum of 4 neighboring pixels need to be processed before processing itself. In a sequential implementation, the rows can be processed from left to right, starting with the first row on the top. The error distribution pattern induces a long chain of data dependency. The last pixel of the image in causal order depends on the first, in theory. The quantization at each pixel is a non-linear operation. The access pattern introduces the following scheduling constraint for each pixel (i, j):



(a) Trapezoidal block structure of single block.

(b) Adjacent blocks, ECFG requires pixels shaded with blue to be processed previously.

Figure 3.4: Block structure and their dependency in Coarse Parallel FSD algorithm on CPU.

$$T(i,j) > max\{T(i-1,j-1), T(i-1,j), T(i-1,j+1), T(i,j-1)\} \tag{3.1}$$

$T(i,j)$ denotes the time/iteration at which pixel (i, j) is scheduled for processing. Thus for a truly optimal scheduling, each $T(i,j)$ should be one unit greater than maximum of its dependencies, viz, $T(i-1,j-1)$, $T(i-1,j)$, $T(i-1,j+1)$, $T(i,j-1)$. Optimal scheduling constraint results in an ordering pattern given in Figure 3.3. This pattern is the source of available parallelism, as the label indicates the iteration in which the pixel can be scheduled. All pixels with label n can be processed in parallel and independent of each other, provided that all pixels with label less than n have been processed. The number of pixels that can be processed in parallel increases as we approach the diagonal of the image. The pixels that can be processed in parallel are arranged in a knights move order in chess. The maximum parallelism for an $m \times n$ image is min$\{m, n/2\}$.

3.3 Coarse Parallel FSD Algorithm on CPU

We now describe a block based implementation of FSD algorithm suitable for multi-core CPUs with a small number $(2-8)$ of cores. The pixels are grouped together into trapezoidal blocks, each block has a structure shown in Figure 3.4(a). Two adjacent blocks $ABCE$ and $ECFG$ are shown in Figure 3.4(b). The shaded region in Figure 3.4(b) shows the pixels that need to have completed their processing before the region $ECFG$ can be processed. Also, due to the pattern of error distribution the bottom-right corner pixel (at the vertex F) can be processed only after all the remaining pixels are processed. The triangle CDE is considered a part of the neighboring trapezoidal block (i.e. first from left), the subsequent blocks are similar to parallelogram $ECFG$. The shaded blue region indicates the pixels on the boundary that should be processed before the current block $ECFG$ is scheduled for processing.

This blue region shows a dependency on neighboring blocks (left, top-left, top, top-right) similar to that of pixels (Figure 3.2(b)). Thus we can use the same knights move order for parallel scheduling of blocks. Figure 3.3 can be thought of as giving the optimal scheduling order for blocks, each of which can

Algorithm 2 Coarse Parallel Floyd-Steinberg Dithering on CPU

Input: Image $\mathbf{I_{in}}$ *// height: m, width: n*
Input: Block Height a, Block Width b
Output: Image $\mathbf{I_{out}}$
for $i \rightarrow 1$ to $(2 \times \frac{m}{a}) + \frac{n}{b}$
  $pRow, pCol \leftarrow$ get-primal-block(m, n, a, b, i)
  omp-set-threads(t)
  # pragma omp parallel for
  for $j = 0 \rightarrow totalBlocks$
    $row = pRow - j$
    $col = pCow - 2 \times j$
    dither-block$(row, col, \mathbf{I_{in}}, \mathbf{I_{out}})$
  end **for**
end **for**

be processed by a thread using the sequential FSD approach. For a multi-core CPU, the thread creation overhead is high. The optimal scheduling of threads is limited by the number of cores on the CPU. A lot of time may be spent on context switching if the number of threads is high compared to the number of cores. It is better to create a few heavy-weight threads as a result. We use each thread to process one or more blocks, keeping the overall number of threads small. This suits the CPU architecture well and gives good results.

We use OpenMP for thread creation and management on the multi-core CPUs. The first occurrence of a block of label n, on a row-wise left to right and top to bottom traversal, is termed as the primal block. Hence, by definition, for all blocks with label n, the primal block is the one with the minimum row number. Once the location of primal block i.e. row and col is obtained for an iteration, we can get the rows and columns for the other blocks by simply $row = row + k$ and $col = col - 2k$, where $k = [1, 2, 3, \cdots]$ till row and col are within the range of the image. The pixels within a block are dithered sequentially in a scan-line manner. Any block requires only the errors from the boundaries of the neighboring blocks (Refer Figure 3.4(b)). The pseudo-code for block-based dithering is given in Algorithm 2.

3.4 Fine-Grained Data Parallel FSD Algorithm on GPU

Operating at the level of pixels (and not blocks) allows us to have more fine-grained parallelism. Pixel-based implementation of FSD is better suited to the massively parallel architecture of the GPU. Due to availability of larger number of cores, it is best to create a large number of light-weight threads on the GPU. We let each thread process a single pixel of the image. The sequential dependence places strict constraints on how the threads can be scheduled. In iteration k, only as many threads can be actively used as there are pixels with label k (Figure 3.3). A GPU kernel operates on the pixel data and the error values from already processed pixels stored in the global memory. The output pixel value and

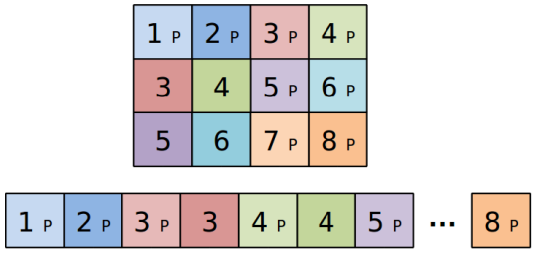


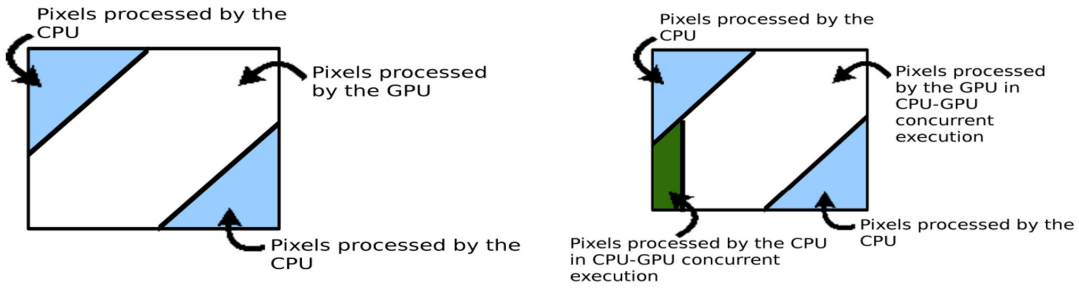
Figure 3.5: Knight's order storage for an image, here p indicates primal block.

the residual error are written to the global memory. The kernel at each thread reads the residual errors of its neighbors, adds the correct fraction to own pixel value, performs quantization, and writes the output pixel value and residual error. The thread number and the iteration number uniquely determine the pixel (i, j) to be processed by the thread.

3.4.1 Addressing Uncoalesced Memory Access

The reading of error values from the neighbors will be very inefficient if the image is stored in the usual row-major or column-major order due to the uncoalesced memory access pattern. For instance, in Figure 3.3, consecutive threads process the 3 pixels with label 6 as shown. Their neighbors are not consecutive in memory and the uncoalesced memory access will be very slow. Optimum memory access times can be achieved by reordering the image. It can be seen that when the left neighbor is accessed by each thread, the memory locations accessed are related by a knights move pattern (with difference in 1 row and 2 columns). We can reorder the image using the iteration number (labels in Figure 3.3) and a pre-decided ranking (top to bottom) within the pixels of same label.

Figure 3.5 shows the mapping from a 3×4 image to a 12-element 1D representation. This re-ordering can be done effectively on the GPU using shared memory by adapting the method used to transpose a matrix [67]. We make use of a simple embarrassingly parallel kernel with $m \times n$ threads to perform this operation. In practice, the time required by other steps is significantly larger than this conversion time. After reordering, each thread of each iteration processes its pixel in a similar way. But, it can be seen that the access to error values from the neighbors as well as access to own pixel value, is totally coalesced in the new converted order, with consecutive threads (or threads in a warp) always accessing consecutive memory locations. Thus, a single memory request can suffice to service the I/O queries of a thread warp. The output pixel and error values are written in the same order also, using coalesced write operations. This maintains efficient memory accesses for subsequent iterations also. The final resulting image needs to be transformed in the reverse using a similar kernel. The FSD computation proceeds in parallel for each iteration of Figure 3.3 resulting in a separate GPU kernel invocation per iteration.



(a) CPU-GPU Handover

(b) CPU-GPU Hybrid

Figure 3.6: FSD using both CPU and GPU. The pixels processed by CPU are shown in blue and green and pixels processed by GPU are white regions within the image.

3.5 Mixing Task Parallelism and Data Parallelism for FSD

The previous implementations used the CPU or the GPU alone. Better performance can be obtained by using both these resources together. A few operations are performed on CPU in addition to the data parallel operations on GPU for best performance. We now discuss two approaches for the same.

3.5.1 Handover FSD Algorithm

The number of pixels that can be processed in parallel (called *parallelism*) is low in the early iterations and in the iterations at the very end. The parallelism slowly increases in the order 1, 1, 2, 2, 3, 3, $\cdots$ until the maximum value of $min(m, n/2)$. Depending on the image dimensions, many iterations can have this maximum value. Thereafter, the parallelism reduces slowly in the reverse order to $\cdots$, 3, 3, 2, 2, 1, 1. If the number of threads is low, the amount of time required for kernel setup is more than the actual processing time of the kernel. We therefore let the CPU process the iterations in the beginning and towards the very end, before handing over the computations to the GPU (Figure 3.6(a)). The *Handover FSD Algorithm* algorithm has the following three stages:

1. Process initial part (top left of the image) on CPU until the parallelism exceeds a threshold and then handover the computation to GPU.

2. Process subsequent pixels on GPU until the parallelism falls below the same threshold (towards the bottom-right part of the image) and then handover the computation back to CPU.

3. Process end part on CPU sequentially. As will be demonstrated by our results, this algorithm involving a handover between CPU and GPU performed better than the algorithm where GPU alone was used for processing all the iterations. Note that GPU alone algorithm is a special case of the Handover FSD algorithm, with CPU immediately handing over the computation to GPU.

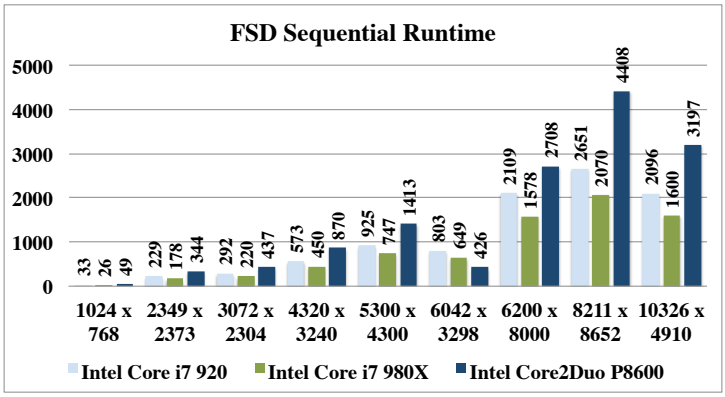


Figure 3.7: FSD sequential implementation timings (in milliseconds).

3.5.2 Hybrid FSD Algorithm

In the Handover FSD algorithm (Section 3.5.1), the CPU and GPU do not operate concurrently. This results in the GPU remaining idle when CPU does the processing and vice-a-versa. On machines like a laptop with a GPU, considerable computing resources are idle at all times. Fast processing of large images needs the mobilization of all resources. Step 1 and 3 of the handover algorithm is best done on the CPU alone as described above. The step 2 can be jointly performed by CPU and GPU. We do this by partitioning pixels of each iteration (recollect that pixels of each iteration can be processed totally independently) between CPU and GPU, with CPU processing a fixed number of pixels (indicated by green region in Figure 3.6). After each iteration, along the border separating CPU and GPU execution, a few residual error values need to be sent either from the CPU to the GPU or in the reverse direction. These errors need to be sent before the next iteration begins. The transfer of error values satisfies the data dependency for the next iteration. We use *zero copy* feature on Nvidia GPUs for the transfer as the data involved is only a few bytes. This is a feature that enables the GPU threads to directly access pinned host memory on CPU. This gives a better runtime than stream transfer methods like *cudaMemcpy* for transferring small amounts of data [60]. Our hybrid algorithm uses all the resources, CPU and GPU on a given compute system and provides optimal performance.

3.6 Experimental Results

We present the results for all the algorithms discussed above. The focus is on real-time processing of large images on ordinary computers using all available computing resources. We also show results on high-end computing resources such as high-end GPUs and 6-core CPUs to show the efficacy of our methods. The CPUs we use are: Intel Core 2 Duo P8600 (2 physical cores), Intel Core i7 920 (4 physical cores, 8 with HyperThreading [49]), Intel Core i7 980x (6 physical cores, 12 with HyperThreading). We disabled the TurboBoost feature on the Core i7 processors to ensure a steady clock speed. We use the GPUs Nvidia 8600M GT (32 stream processors), Nvidia GTX 480 (480 stream processors, Fermi

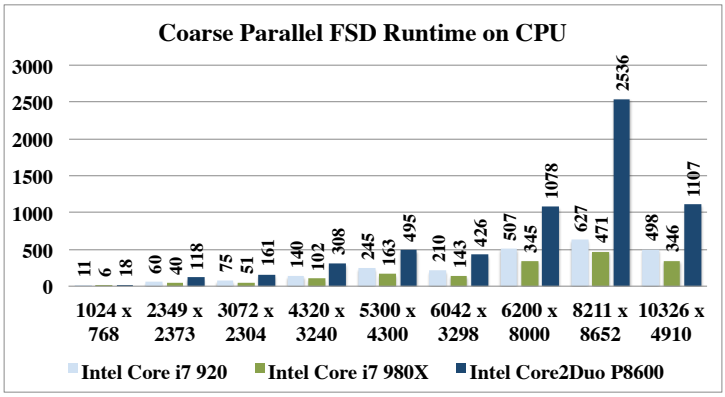


Figure 3.8: Runtime (in milliseconds) for Coarse Parallel FSD on CPU.

architecture), and the Tesla T10 (240 stream processors). The input is an 8-bit gray level image and the output is a binary image. The conversion to knight order (Figure 3.5) for the 1024×768 and 6042×3298 images takes 4ms and 99ms respectively on 8600M. Since this time is minimal, we do not include it in the runtime for the results presented below. The runtime for the sequential FSD algorithm (Algorithm 1) on images of different sizes is given in Figure 3.7.

3.6.1 Results: Coarse Parallel FSD on CPU

We used OpenMP for handling thread creation on multicore CPUs. We varied the block size and the number of threads independently of each other and checked timings for the combinations. A strong dependence between the number of cores in the CPU and the maximum number of threads used is seen, and hence we can assume the timings to be independent of the block size. For a given image, we calculate the block size from the number of threads. For n threads, we use the block size which will have n blocks in parallel for the maximum number of iterations. We are able to get a 3 to $4\times$ speedup over the purely sequential implementation. The number of threads which gives minimum timing is roughly 1.5 to $2\times$ the number of CPU cores for CPUs with Intel HyperThreading Technology [49], and exactly equal to the number of cores for other CPUs. By using this observation, we can compute the optimal number of threads needed for a certain CPU (depending on number of cores) and from this we can calculate the block size for any image automatically. Consider the 1024×768 image (Figure 3.8). For the Core 2 Duo P8600, we see that the speedup is $2\times$ compared to the sequential code, when 2 threads are used. For the Core i7 980x, the speed-up is around $4\times$ (with 9 threads).

3.6.2 Results: Fine-grained Parallel FSD on GPU

As noted earlier, the pixel based implementation on the GPU alone is a special case of CPU-GPU handover algorithm and thus, we do not give detailed results for it separately. The time required for 1024×768 image on 8600M is about 48 milliseconds (ms). Similarly for a 6042×3298 image, we

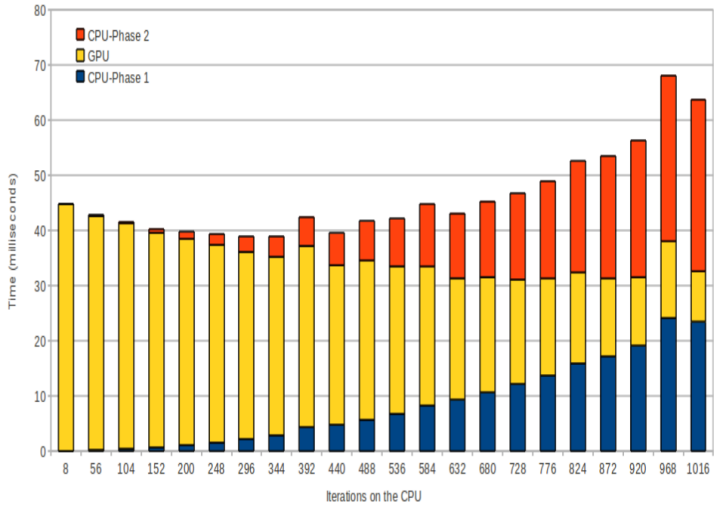


Figure 3.9: CPU-GPU Handover times (in milliseconds) for different # iterations processed on CPU at the start and end.

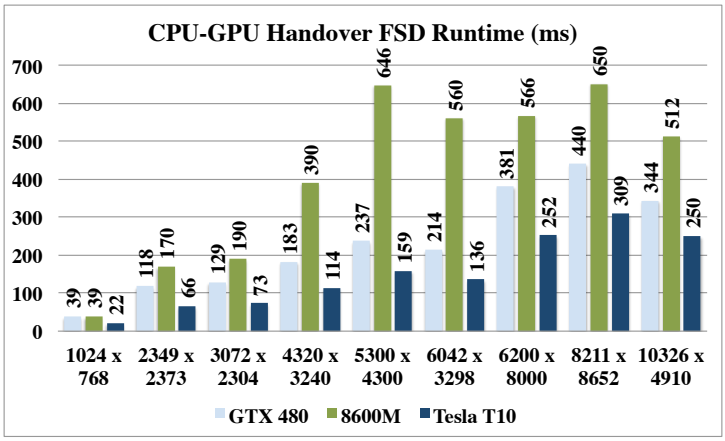


Figure 3.10: Runtime (in milliseconds) for CPU-GPU Handover FSD algorithm on different images.

need about 576ms on 8600M. These values are higher than the corresponding values needed for a CPU-GPU handover. In fact, as discussed earlier, our results reaffirm that to improve performance some of the initial and final iterations (with low parallelism) must be offloaded to the CPU.

3.6.3 Results: Handover FSD Algorithm

The Handover FSD algorithm uses the GPU resources well. We vary the number of iterations handled by the CPU and GPU. In essence we modify the parameter iteration number n, which is the number of iterations after which the CPU hands over control to the GPU, and the number of iterations the CPU handles towards the end. So an iteration value of n means that the CPU handles n iterations at the beginning, n iterations at the end and the GPU handles the iterations in between. Figure 3.9 plots of

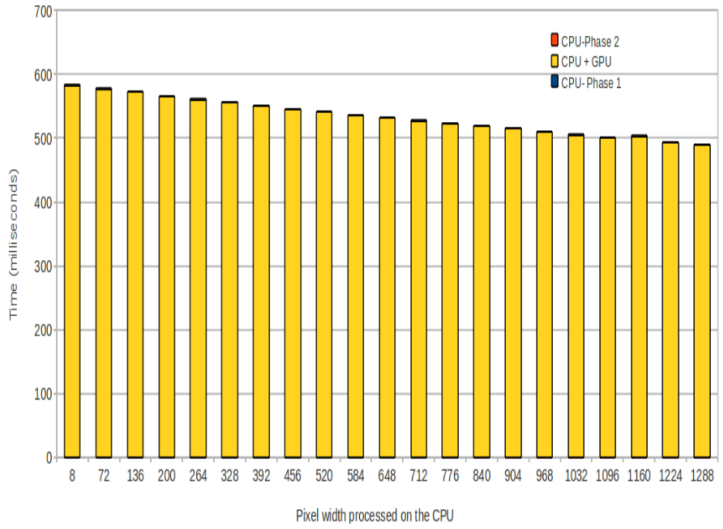


Figure 3.11: CPU-GPU Hybrid FSD times (in milliseconds) for different width of pixel boundary processed on CPU.

total time v/s iteration number at which the switch was performed. The nature of this graph indicates that there is an optimum value of the number of iterations which should be performed on the CPU. Increasing or decreasing this value, results in an increase in the running time. Also as seen in Figure 3.10, the time required to dither various images is the lowest for the Tesla T10. These timings are considerably lower than the sequential timings (Figure 3.7).

3.6.4 Results: Hybrid FSD Algorithm

The hybrid algorithm uses both CPU and GPU during step (2) of the processing. We vary the number of pixels handled by CPU in this step. This changes the boundary of CPU-GPU concurrent execution part of the hybrid algorithm. Figure 3.11 gives time for different amounts of load handled by the CPU in concurrent CPU+GPU phase. The results demonstrate the existence of an optimum value for pixel width processed by CPU that reduces the total overall processing time. The overall timings for various images on the different GPUs are shown in Figure 3.12.

3.7 Discussion

Our Handover and Hybrid FSD approaches which combine task parallelism with data parallelism perform better than GPU alone, data parallel approach. This results from the fact that we are able to utilize all available computational resources i.e. CPU and GPU. On high-end hardware, our speed up is around $10\times$ and on low-end hardware, after exhaustive utilization resources, the speed up is 3 to $4\times$, compared to a standard sequential implementation, shown in Figure 3.7. Our approaches involving the GPU work better for large images, since the kernel setup time, the memory copy time, etc., become

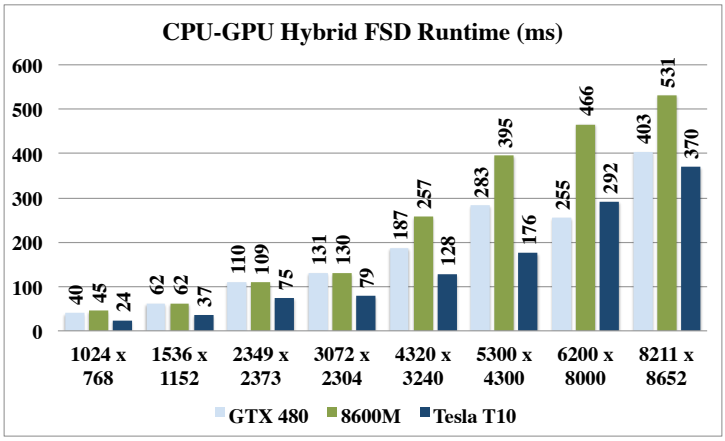


Figure 3.12: Runtime (in milliseconds) for CPU-GPU Hybrid FSD algorithm on different images.

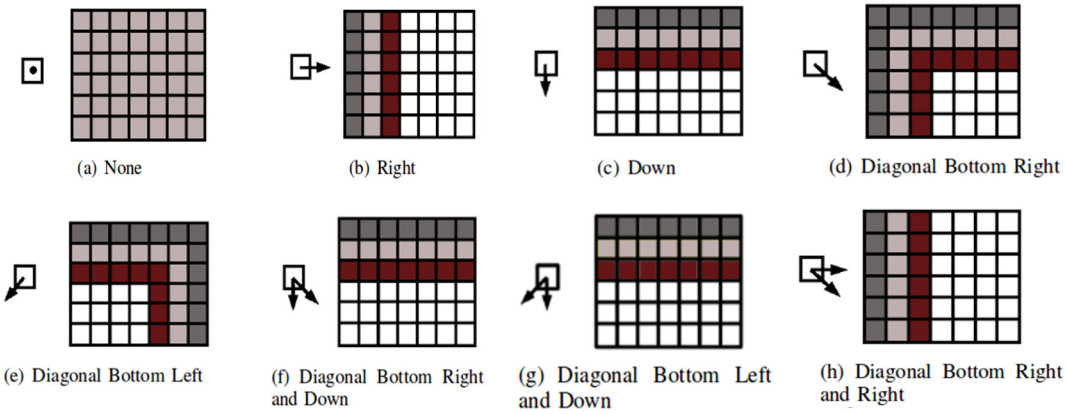


Figure 3.13: Various types of Data Dependencies and resulting parallelism. Pixels shaded with the same color can be processed together in parallel.

overheads for small images. Although it may seem that in some cases, the block-based CPU algorithm performs better than the Handover or Hybrid FSD algorithms, it must be remembered that multi-core CPUs used in the block-based CPU results are relatively high-end CPUs (with 4-6 cores). On the other hand GPU as well as the CPU in the Handover or Hybrid FSD results is a commodity or low-end hardware. Thus, we demonstrated that problems with long sequential dependency, like Floyd Steinberg Dithering (FSD), can be efficiently implemented in parallel even on low-end hardware. Figure 3.13, shows different types of data dependency patterns. As shown by the shaded regions in the figure, we see that our analysis provides us with groups of independent pixels for data parallel processing.

Chapter 4

String Sorting

Sorting is an important operation in data processing. Fast and efficient sorting has been available on the GPU for a few years. Radix sort performs the best on the GPU and its performance has improved steadily since its introduction [56, 69]. The Thrust template library contains fast sort implementations, with radix sort by Merrill and Grimshaw being the fastest [56]. Thrust sort can be enhanced with a custom comparison operator [32]. Sorting is a vital data parallel primitive on the GPU and is a critical component of several algorithms [2, 31, 24, 54, 76]. The use of standard primitives opens the door for automatic performance improvements when the primitive's performance improves due to an improved implementation or an improved architecture, both of which happens with sort on the GPU. A primitive based implementation can be thus be adapted efficiently to newer generation of architectures that differ significantly.

Sorting variable length keys is still a challenge on the GPU. The most typical problem is sorting of strings of arbitrary length, which is a common occurrence in different fields. Variable length sorting implies different threads will perform different work (in form of comparing keys), making load balancing between them *difficult*. Also, sorting inherently involves dependency of an element to many other elements making it *challenging* to develop parallel algorithms for it. Based on the input keys, data undergoes shuffling memory accesses are arbitrary. Variable load and arbitrary accesses makes the problem of string sorting irregular (Section 1.1.2). In this chapter, we develop a data parallel algorithm to efficiently solve this irregular problem of string sorting. Our string sort is built over existing sort primitives on the GPU. This will result in future performance improvements in the primitives to translate to string sorting with no redesign. The developers of Thrust suggest the use of a custom string-comparator to perform string sorting by a merge sort algorithm. The performance of this depends critically on the comparator implementation and is not fast enough perhaps due to repetitive comparisons and data loading that occur at every merge step. Davidson et al. presented the first string sort implementation on the GPUs using an efficient merge sort for long and variable length keys [17]. Their performance is good, but scalability to large datasets may not be straightforward using merge sort compared to radix sort.

In this chapter, we present a fast and efficient GPU string sort that achieves better performance than both the above methods on large datasets. Our method is simple and scalable. We present exhaustive

results on several datasets to demonstrate the applicability and scalability of our method. We use the standard benchmark datasets as well as our own to demonstrate scaling to larger problem sizes. The average after-sort tie-length quantifies the difficulty of sorting. We present results on datasets which a cover a wide spectrum of these tie lengths. We also propose the percentage of the computation time spent on standard performance primitive as a measure of adaptability of GPU applications. Thrust primitives dominate the string sort time in our approach (on an average 70%). We also analytically estimate the expected runtime for our algorithm; the experimental results are within realistic limits of these expected runtimes. Our string sort achieves a speedup ranging from 1.8 to 19.7 over the current methods. Our speed up is particularly good on challenging datasets with large strings and long ties. We obtain a sorting throughput of 83 MKeys/s on a dataset of 1 million random strings. Our approach can scale to an order of magnitude larger input size than the previously reported GPU string sort. On a 10 million words dataset we achieve a throughput of 65 MKeys/s. Our code is available for public use and it has also been accepted into the standard CUDPP library for GPU primitives[1].

4.1 Related Work

We review the relevant sorting and string sorting algorithms for the CPU and the GPU in following sections.

4.1.1 CPU String Sorting Algorithms

Most sorting algorithms are designed with the assumption that input keys are a few characters or integers (i.e. fixed length keys). For keys which span more than a few integers or characters (i.e. strings), comparison based sorting algorithms can use an iterative comparator between given keys. Radix sorting algorithms can create and recursively sort buckets starting from most significant to the least significant integer or character. Comparison based sorts will perform $\Omega(Nlog(N))$ comparisons, each iterating over two strings to resolve ties. Naive radix sort procedures will shuffle the entire string in each step. Both of these could be expensive in practice.

Several specialized string sorting algorithms have been developed over time. Most popular and efficient amongst such approaches are: mutli-key quicksort [8], Burstsort [72, 73] and MSD (most significant digit) radix sort [37]. These typically use a combination of two or more of the standard sorting algorithms augmented with a few additional steps for performance. Burstsort uses *burst-trie* data structure and a standard sorting algorithm [8, 52]. It organizes strings into small buckets by inserting them into a burst-trie, such that these buckets can be sorted within the CPU cache memory. The sorted buckets are lexicographically ordered amongst them and need not be merged later. Kärkkäinen and Rantala engineer an efficient radix sort for strings which involves repetitive application of radix sorting

[1]Code available at http://web.iiit.ac.in/∼adity.deshapandeug08/stringSort/ and integrated with CUDPP at https://github.com/aditya12agd5/cudpp

from most significant digit to the least significant one [37]. They develop *counting* based methods which require pre-computing the bucket size. To achieve this, they need to make two passes over the data per radix sort step, where the first pass computes bucket sizes and next pass scatters the data. They also develop one-pass *dynamic* methods where buckets are generated and resized on the fly. They resort to simpler sorting viz. insertion sort when buckets are small enough. They avoid shuffling entire strings by manipulating pointers, cache successive characters of strings, and use supra-alphabets (2 byte characters instead of 1 byte) for small buckets, to reduce the sorting steps.

Our algorithm comes under the category of counting based method of [37], with actual implementation exploiting the efficient primitives on the GPU. The MSD radix sort creates buckets in a depth first manner, while we show that a breadth-first pattern is more suited for exploiting the parallelism on the GPU. Also, our implementation can compare longer length of prefixes (maximum of about 8 characters long) per radix sort step as compared to the two character limit of MSD radix sort. In Table 4.6 and Section 4.3.6, we show that our GPU algorithm gives significant speed up ($4\times$ - $17\times$) compared to both Burstsort and MSD radix sort algorithms on standard benchmark datasets.

4.1.2 GPU String Sorting Algorithms

The GPU has emerged as a massively parallel accelerator for problems that have a strong data parallel flavor. Several high-performance sorting algorithms have been designed for it. Cederman and Tsigas developed an implementation of Quicksort [14], sample sort was implemented by Leischner et al. [43]. Satish et al. developed a radix sort on GPU which outperformed an 8-core CPU by $4\times$ [69]. Their performance benefits from the reducing of scattered writes to global memory by making use of on-chip shared memory. They also developed a fast merge sort, which merged small blocks that fit in the on-chip shared memory. Merrill and Grimshaw later developed a radix sort using fast scan primitives which is the fastest sort today [56]. They introduced optimizations like adapting the radix sort granularity (digit size) to fit the underlying GPU architecture, kernel fusion and serialization of threads for appropriate steps to reduce the global memory accesses. All these implementations assume a fixed key size of 32/64 bits and cannot handle variable or long keys.

Thrust also provides efficient primitives for fixed key length radix sort and merge sort [32]. Developers of Thrust suggest creating a user-defined string class and a custom iterative comparator to extend their sort to strings[2]. Thrust supports radix sort only for basic datatypes and resorts to a merge sort when provided with user defined types and a custom comparator. Radix sort primitives are significantly faster as compared to merge sort and such an approach fails to exploit them.

Recently, Davidson et al. developed an efficient merge-sort based string sorting procedure that handles variable length keys well [17]. They prefer register packing (by creating a limited number of threads) against over-utilization of GPU. Input was divided into blocks of fixed size, $m = 1024$, and $m/8$ threads (per SM) were created to sort 8 elements each through a carefully designed bitonic sorting network. Each thread then modified its search-space (neighboring 8, 16, 32 and so on elements)

[2] http://goo.gl/mlwlZ

to create the final sorted block of size m. In the next step, on each GPU SM two blocks were merged which doubles the size of the output block and halves the number of blocks. At the end when blocks to be merged are few, they use a scheme where all threads (or cuda blocks) cooperatively merge them. For variable length keys, they initially sort the first 4 characters and load successive characters from global memory in case of ties. Each part of the string is compared multiple times (during every merge step) and each comparison involves accessing the high latency global memory. As the merge procedure nears conclusion, they observe that, comparisons are made between more similar strings and ties take longer to resolve. Resolving ties involves accessing high latency global memory and also causes thread divergence, these issues cannot be easily solved. Their implementation shows a good sorting performance of 70M Strings/sec on a dataset of 1 million strings when ties are few and slows down by $4-5\times$ when tie length increases. Though the GPU radix sort primitives are relatively faster, their string sort again uses a merge sort. They avoid radix sort because "costs of radix sort algorithm that involves direct manipulation (i.e. shuffling) of keys will scale with key length".

We introduce a different category of efficient string sorting algorithms for the GPU, based on fast fixed length radix sort. Instead of comparing two strings at a time in the traditional iterative manner, we show that we can efficiently compare all strings in a column-wise manner from first to the last character. Our approach uses a way to limit the shuffling to only string indexes and small prefix strings at each step, while performing all operations with fast parallel primitives. Radix sort based string sorting procedure allows us to compare the characters of the string only once (from left to right), thus avoiding repetitive global memory accesses. Also, on practical datasets we see that the length of ties do not exceed more than a few hundred characters and results in Section 4.3.4, 4.3.3 show that, our GPU algorithm gives better performance and scalability as compared to the merge sort based string sorting methods of Davidson et al. and the Thrust Library.

4.2 Radix Sort Based String Sort

The input setup stores all strings in the global memory as a single contiguous array, delimited by null characters (Figure 4.1), accompanied by an index array. The pointers to the strings are indices in this global string array. Fixed length radix sort primitives are the fastest amongst all others. One way to use these primitives for performing string sorting is to load entire strings as keys and perform the radix sort operation. This will recursively create buckets from the first to last character, but will involve shuffling the entire string keys at each step. Such an extensive data movement will form a huge bottleneck on the GPUs.

We develop an approach that is outlined in Algorithm 3. We exploit the efficiency of radix sort while reducing data movement by sorting records of string bytes as the key and string index as the value (line 10). Each step uses a few bytes of each string starting at a fixed offset from the left as the key; the offset is 0 for the first step (line 4). The fixed length radix sort primitive from the library is used in each step. Strings with a common prefix so far come in adjacent positions after sorting. Strings with

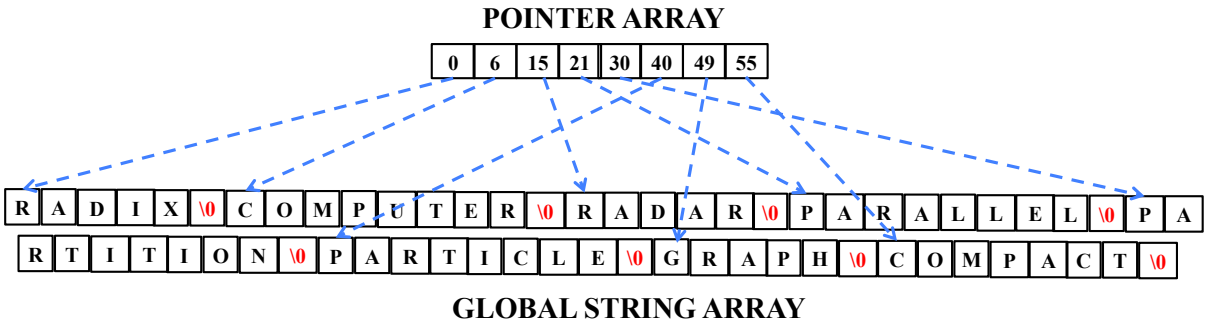


Figure 4.1: The strings in the global string are delimited by null characters. The pointer array contains indices of the starting position of each string in the global string array. This pointer array is shuffled during the sort to obtain the sorted order of strings. The example we consider in this paper contains the set of strings : **radix, computer, radar, parallel, partition, particle, graph, compact.** We use this same set of strings to illustrate our sorting procedure.

unique prefixes will be singletons and would already be in their place in the sorted output. These can be marked and removed from further processing (line 11). For the remaining strings, we assign *segment ids* (stored in the Seg array) beginning with 0 for the lexicographically smallest and increment by 1 whenever the next string differs. Common prefix strings are contiguous and get the same *segment id*. Segment assignment is performed using a scan primitive after marking in parallel the locations where adjacent sorted records have different keys. Each segment represents a bucket; strings belonging to it will only shuffle among themselves without crossing segments in the final output. The *segment id*, thus, encodes the history of the sorting steps till the current one, which allows us to discard the currently sorted prefixes and load successive bytes in their place for further sorts (line 17). We do further sorts on records with a key consisting of the *segment id* on the left and next few bytes from each string on the right and a value consisting of the string pointers. The tuple of *segment id* and successive characters forms a proxy for the entire prefix of each string. The pointers in the value, together with the offset are used to load successive characters in every sort step.

Our approach is illustrated in Figure 4.2, where we sort the set of strings: *radix, computer, radar, parallel, partition, particle, graph and compact*, organized in memory as shown in Figure 4.1. In this example, we load 2-character prefix strings before every sort step. After the first sort step we see that the prefix *gr* is a singleton, thus the position of *great* in output array is fixed as 3. The other prefix strings *co, pa, ra* are assigned incremental segment ids. We then use the value part which consists of pointers of the strings to load successive 2-character prefixes for each string. The next sort is performed on the tuple of segment ids and the newly loaded 2-character prefix. This generates more singletons *ra, da, di* which fixes the positions of the strings *parallel, radar, radix* respectively. The same process repeats for one more iteration after which all strings become singletons and we obtain the sorted order.

Algorithm 3 Our GPU String Sorting

```
1: Input: String Array G, Index Array I
2: Output: Shuffled Index Array O.
3: k = optimal key length                                   // 8 for our platform
4: M ← load_prefix(G,k, 0)
5: offset ← k                                               // load next prefix starting at offset
6: Seg ← [0, 0, ··· , 0]                                    // Only one segment
7: segBytes ← compute_bytes(Seg)                            // 0 initially
8: K ← pack_keys(M, Seg, segBytes)
9: repeat
10:    radix_sort(key: K, value: I)
11:    F ← mark_singletons(K, O)                            // F = Flag, O = Output
12:    // above step also writes index of singletons to output
13:    D ← prefix_scan(F)                                   // D = Destination Array
14:    K, I ← scatter(K, I, flag: F, dest : D)              // compaction
15:    Seg ← generate_segments(K)
16:    segBytes ← compute_bytes(Seg)
17:    M ← load_prefix(G, k− segBytes, offset)
18:    offset ← offset +k− segBytes
19:    K ← pack_keys(M, Seg, segBytes)
20: until no segments left
21: Output : Shuffled Index Array O
```

4.2.1 Singleton Elimination

A parallel kernel performs the singleton elimination. First a flag array is created with 0 if a prefix is a singleton – that is, it is different from its predecessor and its successor – and 1 otherwise. An exclusive prefix sum of the flag gives us the destination indices for each remaining string. We compact the original records with flag of 1 using a scatter primitive and the destination indices. Singleton elimination reduces the problem size for the subsequent iterations and reduces the memory requirements. In practice, singleton elimination improves the string sort performance by $1.1\times$ to $4.5\times$ as shown in Table 4.3. The use of library primitives for scan and scatter helps in scaling to other systems. Our approach thus can handle strings of much larger problems than has been shown in the past on the GPU as can be seen from our results.

4.2.2 Optimal Key Length and Adaptive Segment Ids

The radix sort primitives support keys of lengths 1, 2, 4 and 8 bytes efficiently on current Nvidia GPUs. Beyond 8 bytes, the values are tupled together and compared with 2 or more standard comparisons [7]. Figure 4.3 shows the normalized sorting time per byte of the key on an Nvidia GeForce GTX 580 GPU for different key lengths. Beyond 8-byte keys, there is a sharp increase in this normalized per byte sorting cost. This suggests that the individual sorting steps should use 8-byte keys for maximum

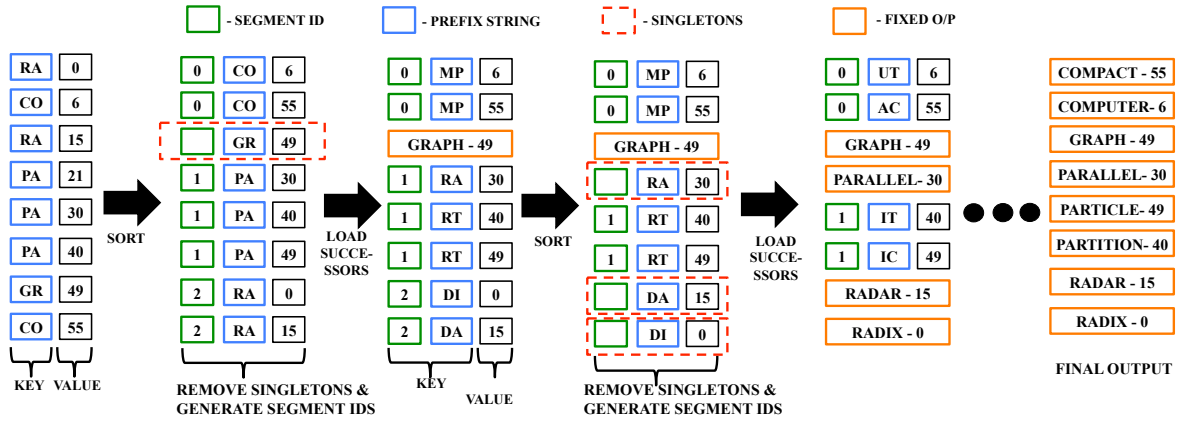


Figure 4.2: Illustration of our basic GPU sorting algorithm. In this example, we load two-character prefix strings in each step. The steps of fixed-length sorting, removing singletons, generating segment ids and loading successive prefix strings are performed until we obtain the final output (i.e. all strings are singletons).

performance. This number may be different on other accelerators and on future Nvidia GPUs. Our algorithm should use appropriate key-lengths on each.

We pack the segment ids and prefix strings into the key. We can use 4 bytes to represent the segment id and the rest for subsequent string bytes. This will process the strings slower when the number of segments are small. We adapt to the number of segments and use the minimum number of bytes to represent the segment id at each step. This varies from 0 bytes for the first step (all strings are in the same segment trivially) to 3 bytes in practice in our examples. (The problem size itself was less than 2^{24} except in one case.) The remaining bytes are used to load the prefix strings. This adaptive scheme allows us to compare more number of characters in each step, reducing the number of iterations. On low entropy data the adaptive scheme allows us to have fewer bytes for the segment id, which implies more characters are compared per sort step and more new segments can be identified. For high entropy data, segment id consumes more bytes and relatively lesser number of characters are compared per sort step. This is still reasonable because strings are already highly segmented and only a few more comparisons are needed to resolve ties. The results in Table 4.4 show that we get significant speed up with the adaptive scheme.

4.3 Experimental Results

We use efficient parallel primitives provided by the Thrust Library (v1.6.0) [7]. Our setup has Nvidia GeForce GTX 580 GPU (compute v2.0) and we use CUDA software version 4.0. We also demonstrate our results on Nvidia Tesla K20C (compute v3.5), with the Kepler architecture. For K20C, we use CUDA software version 5.0. We present the performance of our string sorting algorithm on different

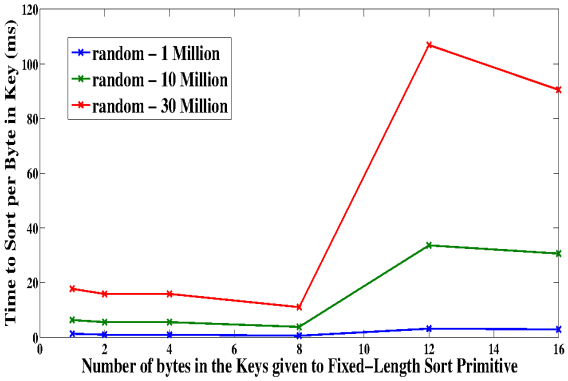


Figure 4.3: Performance of Thrust's fixed-length sort primitive with varying key length.

datasets (Table 4.1) in this section. The runtimes that we measure for all CPU and GPU algorithms are typically in milliseconds and do not include the File I/O time.

4.3.1 Thrust performance on varying key size

Figure 4.3 presents the performance of Thrust's fixed-length radix sort primitive for different key lengths. The key length changes from 1 to 16 bytes. The total time divided by the key length in bytes, to sort 1 to 30 million random keys plus 4-byte value per key record, is shown in the figure. The per byte sorting time reduces as key length varies from 1 byte to 8 bytes. Beyond 8 bytes, Thrust shifts to a slower merge sort algorithm [7]. This results in a sharp increase in the per byte sort time. The optimal per byte sort time is thus obtained for 8 byte keys. We, therefore, fix the key size to 8 bytes for the GPU radix sort operations we perform. Please note that this parameter may vary on other GPUs and certainly on other accelerators. The k parameter in Algorithm 3 can be set to the optimum value when adapting our method to other architectures.

4.3.2 Datasets

The details of the datasets used in our experiments are given in Table 4.1. The top 8 (artificial-2 to genome) are standard datasets used in the CPU string sorting literature [37, 73]. They have data of varying size from natural sources such as list of english words, urls, genome sequences as well as synthetic ones such as list of repeated strings of same character (artificial-2), list of repeated strings of same character but varying lengths (artificial-5), random strings (random), words formed from a few characters (artificial-4), etc. Good performance on these datasets indicates the robustness of the sorting algorithm. The GPUs are particularly good to process larger datasets. Hence, we create additional datasets such as *pc-filelist* and *sentences*. The *pc-filelist* was created by listing all (about 2 million) files on a typical server class machine starting with the root character "/". This was replicated 5 times with a different prefix for each copy (viz. "node1/" to "node5/") to obtain a dataset of 10 million strings.

Dataset	Details	Size (MB)	After Sort Tie Length	
			Max.	Avg.
artificial-2	10^6 same strings of A repeated ≈ 100 times	98	101	101
artificial-4	10^7 strings of characters a to i	162	80	13
artificial-5	10^6 strings containing A but varying length (1 to 100)	50	100	50
dictcalls	100187 strings of opcodes	2.2	35	15
random	10^6 random strings of length ≈ 100	97	5	2
words	$10^7 \approx 10$ million words with no duplicates	118	48	7
url	$10^7 = 10$ million strings containing urls	305	215	29
genome	$31623000 \approx 30$ million strings of a, t, g, c	302	9	8
sentences	≈ 1.8 million sentences from gutenberg project	124	63	17
pc-filelist	≈ 10 million strings containing filepaths	656	180	59

Table 4.1: Details of the datasets used in our experiments. We create and use sentences and pc-filelist to particularly benchmark our code on typical datasets with high number of ties. The other datasets are standard datasets used in previous string sorting literature [37, 73].

Dataset	Thrust Comparator Sort			Our GPU String Sort						Speed Up	α
	Mem. Setup	Sort	Total	Mem. Setup	Iterations (k)	Sort (t1)	Scatter/ Scan (t2)	CUDA Kernels (t3)	Total		
artificial-2	122	2652	2744	91	15	35	31	61	219	12.5	3.6
artificial-4	113	5699	5813	106	15	197	44	87	436	13.3	1.6
artificial-5	97	1808	1906	60	17	69	24	20	175	10.8	1.6
dictcalls	81	32	113	50	6	7	3	1	62	1.8	1.5
random	182	75	257	70	1	6	1	5	84	3.0	2
words	105	2016	2122	97	9	97	22	35	252	8.4	1.5
url	155	8559	8714	153	37	416	102	264	936	9.3	1.8
genome	155	14064	14219	275	2	189	92	164	720	19.7	2.3
sentences	131	895	1026	79	11	65	18	36	199	5.1	1.8
pc-filelist	230	9003	9234	225	33	705	164	313	1409	6.5	1.6

Table 4.2: Comparison of runtime (in milliseconds) of the Thrust Comparator based string sort and our GPU string sort. The table shows the split in runtimes for different steps of our string sorting algorithm. The value $\alpha = (t1 + t2 + t3)/t1$.

For the *sentences* dataset, we extract sentences from nearly 40 e-books of the Gutenberg project. Large after-sort tie length is an indicator of the difficulty of sorting a dataset [17]. These strings have many ties to each other since many files share common base directory path and many sentences have similar beginning. This makes them difficult, yet not artificial, inputs for string sorting. Their large size also allows us to test for the scalability of our approach. These datasets as well as our code will be available for others to use.

4.3.3 Performance of our GPU String Sort

In Table 4.2, we compare the performance of our radix sort based GPU string sort to Thrust's comparator based string sort described in section 4.1.2. We use the implementation of Thrust comparator

based method that is available as part of a suffix sorting code[3]. This implementation is very slow as it does not support bulk copy of the user-defined string class. Copying one string at a time is the bottleneck. We enhanced the code to support bulk transfers so as to facilitate its running on large datasets. Table 4.2 shows that we obtain a speed ups ranging from 1.8 to $19.7\times$. We analyze and justify this performance in the following sections.

Sort Time and Total Time. Table 4.2 shows the split of the times taken by different steps of the string sort for all the datasets. The memory setup time includes the time for allocating memory on GPU as well as doing all the data transfers between CPU and GPU. We also measure the time taken by the sort primitive ($t1$), scatter/scan primitives ($t2$) and the CUDA kernels that perform the remaining operations ($t3$). We see from the results that sorting is the most expensive step on all large datasets. The only exception is artificial-2 dataset, which sorts arrays of equal values each time. Thrust handles this specially by short-circuiting the sort and provides very fast sorting [56]. Typically, if we exclude the memory setup time, we see that the total time varies in a small band of 1.5 to $2.3\times$ the time taken for radix sort. We denote this by a factor α, shown in the last column of Table 4.2. In practice, as of today, we can empirically expect α to be bound by about 2.5 for our algorithm.

Expected vs. Achieved Time. Given that the basic sorting primitive takes on average time t per iteration, the total time for an iteration can be estimated to be $\alpha \times t$. With k iterations, then estimated total time is $\alpha \times t \times k$. If the throughput of fixed-length radix sort primitive is given in p MKeys/s and if the string sorting problem has size N million strings, then the estimated time per iteration is $1000/p \times N$ milliseconds. Thus, the expected total time for string sort in milliseconds is

$$T_{exp}(ms) = \alpha \times 1000/p \times N \times k \approx t1 + t2 + t3. \tag{4.1}$$

Fixed-length radix sort primitive of thrust offers a sorting performance of 1GKeys/s = 1000 MKeys/s [56]. But in practice, on our architecture, we achieve a throughput of 500 to 760 MKeys/s from the radix sort primitive, thus we fix $p = 750$ in equation 4.1 and calculate the expected runtime for $\alpha = 2.5$. The results in Figure 4.4 show that we achieve better runtime than the expected time on artificial-4, words, pc-filelist and url datasets (each of which has 10 million strings and also long ties). Such an analysis, shows that we can predict the performance of the string sorting algorithm based on a fixed-length sorting primitive. This prediction needs parameters such as input size (N), performance of the fixed-length sort primitive (p) and max ties in the input (i.e. some indicator of the value of k). Our results in Figure 4.4, show that our achieved performance is within practical limits of the expected performance in many cases and it justifies the speed up that we obtain over other methods.

Singleton Elimination and Adaptive Segment ID. Removal of singletons progressively reduces the sorting problem size. This allows the fixed-length sorting primitive to perform better. In equation 4.1, this translates to reducing the impact of N in successive iterations. Table 4.3 shows that we achieve a

[3]https://github.com/bzip2-cuda/bzip2-cuda/tree/master/lib

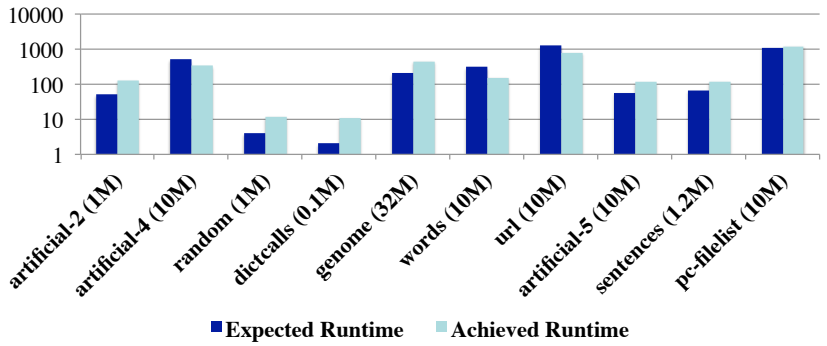


Figure 4.4: Achieved vs. Expected runtime (in milliseconds) for our radix sort based GPU string sorting algorithm.

Dataset	With Singleton Elimination - Time (ms)	Without Singleton Elimination - Time (ms)	Speed Up
artificial-2	367	357	0.9
artificial-4	601	2237	3.7
artificial-5	299	317	1.1
dictcalls	28	38	1.3
random	23	31	1.3
words	423	1340	3.1
url	1160	5331	4.5
sentences	219	408	1.8
pc-filelist	1723	4791	2.7

Table 4.3: Comparison of runtime (in milliseconds) of our GPU string sorting algorithm with and without the optimization of singleton removal. To decouple the optimizations and study them separately we maintain a fixed segment id size in above experiments. The runtime improves with singleton removal.

speed up between 1.1 to 4.5 for different datasets after eliminating singletons. There is no speed up for the artificial-2 dataset, since it consists of equal strings and no singletons are found. For url, artificial-4 and pc-filelist datasets, which have large number of strings and long ties we see that our singleton elimination technique performs particularly well giving us a speed up of 4.5, 3.7 and 2.7 respectively. Using the minimum number of bytes for the segment id field enables us to reduce the total number of iterations as more number of characters are compared per iteration. This reduces k in equation 4.1. Table 4.4 shows that we achieve a further speed up of $1.4\times$ to $3.3\times$ by using the adaptive scheme for segment id bytes. Both, these optimizations aim at reducing the expected runtime for our GPU algorithm and also yield good results in practice.

Thrust Comparator based String Sort. The enhanced Thrust sort for strings using a custom comparator performs poorly (Table 4.2). This is due to the use of the slower merge sort as well as the loading successive characters from the high-latency global memory to resolve ties. These accesses need to be

Dataset	Fixed Size Segment Id		Adaptive Size Segment Id		Speed Up
	# Iterations	Total Time	# Iterations	Total Time	
artificial-2	26	367	15	127	2.8
artificial-4	21	601	15	328	1.8
artificial-5	26	299	17	113	2.6
dictcalls	9	28	6	11	2.5
random	2	23	1	12	1.9
words	13	423	9	154	2.7
url	54	1160	37	782	1.4
genome	3	1488	2	445	3.3
sentences	16	219	11	119	1.8
pc-filelist	46	1723	33	1182	1.4

Table 4.4: Comparison of runtime (in milliseconds) of our GPU string sorting algorithm with and without the optimization of adaptive size for segment bytes. This shows that our adaptive scheme reduces the number of iterations and provides us a significantly better runtime.

performed repeatedly per string in *every* merge step. Our method loads the string from start till the ties are resolved only once per sort step. Our approach makes full use of the fact that the fixed-length radix sort primitive ensures limited global memory accesses [56]. Reducing the high latency global memory accesses and exploiting fast primitives of radix sort and scan, help us achieve much better performance than the comparator based string sort of Thrust, Table 4.2 shows that we achieve a speed up of 5 to 10 on large datasets.

Time spent on Thrust Primitives. Figure 4.5 presents the fraction of total execution time that is spent using Thrust primitives of sort, scan and scatter. This measure is an indicator of the adaptability of our methods to newer implementations and architectures. For large datasets with long ties (i.e., url, aritificial-4, sentences, pc-filelist) we see that greater than 65% of the time is used by Thrust primitives. On an average the Thrust primitives use 70% of the execution time. These results show that our algorithm without any redesign can benefit well from any future improvements to these primitives or the basic architecture.

4.3.4 Comparison to Davidson et al. [17]

We compare with the method by Davidson et al. using the CUDPP code from them[4] [17]. This code is still under development and lacks a few optimizations mentioned in their paper. It presently does not scale to the input size that we use for most datasets. On the wiki-words dataset they used, we obtained a runtime of 20 ms, while they reported about 14 ms in their paper [17]. This is an easy dataset, with very few ties (average ≈ 5) and only 1 million elements. Hence, the small difference in runtime is not very significant. Scalability to large datasets is a particular strength of our approach since it is totally

[4]http://code.google.com/p/cudpp/source/checkout

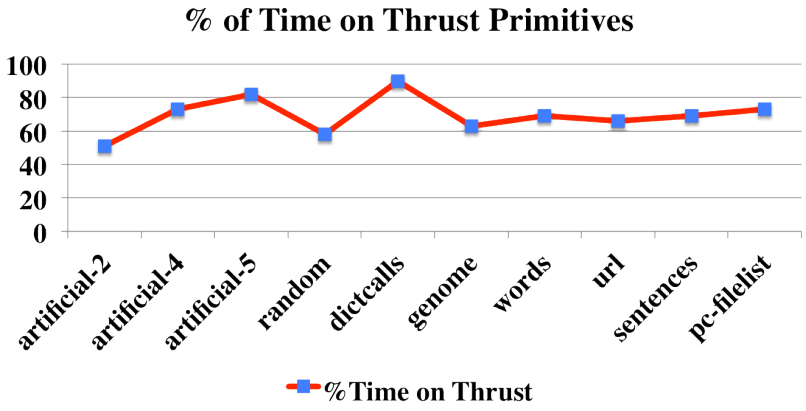


Figure 4.5: The % of total time (w/o memory setup) used for execution by the Thrust sort, scatter and scan primitives. On an average, 70% of the total time is utilized by Thrust primitives.

Dataset	**GTX 580** *(Fermi)*	**K20C** *(Kepler)*	**Speed Up**
artificial-4	436	472	0.92
url	936	899	1.04
genome	720	642	1.21
pc-filelist	1409	1121	1.25
pc-filelist $\times 2$	-	2574	-

Table 4.5: Comparison of runtime (in milliseconds) on the Kepler K20C and GTX 580 GPU. pc-filelist dataset is replicated twice ($\approx 20Mstrings$) to create pc-filelist $\times 2$. K20C can process this large input because of its high global memory, which is not possible on GTX 580.

based on the radix sort primitive, as opposed to the slower merge sort they use. On 1 million random strings and a larger words dataset with 10 million strings, their implementation takes 27 ms and 2100 ms respectively. Our string sort takes only 12 ms and 154 ms respectively on these datasets, we get a speed up of 2 to $13\times$. The comparison will be more meaningful only when a fully optimized and stable version of their code is available.

4.3.5 Performance on the Kepler GPU

Table 4.5 compares the performance of our string sort on Nvidia K20C and Nvidia GTX 580 GPU. K20C is of a different architecture family (kepler) and has a relatively slower clock speed of 706 MHz as compared to 772 MHz of GTX 580 (fermi). But, it has a global memory of $\approx 5GB$ and can process much larger inputs than GTX 580 ($\approx 2GB$ global memory). It also has 2496 cores, while the GTX 580 has only 512. In our experiments on sorting random integer data with the Thrust sort primitive, we see that the K20C performs only marginally better as compared to GTX 580. Thus, using the K20C we achieve a speedup of 1.21 and 1.25 on genome and pc-filelist datasets respectively. On an even larger dataset, pc-filelist $\times 2$, created by concatenating the pc-filelist dataset twice, K20C gives a runtime of

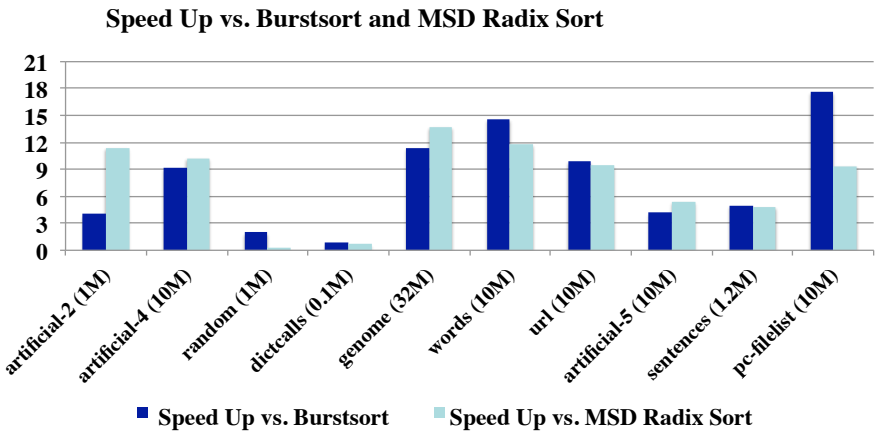


Figure 4.6: Speedup by using our GPU string sorting algorithm against state-of-the-art CPU algorithms for string sort. For these experiments, we use the Nvidia GTX 580 GPU and Intel Core2Duo E7500 CPU.

2.57s. The same dataset takes about 38.1s and 27.8s on the CPU using Burstsort and MSD radix sort respectively (i.e. speed up of 14 and 10). Also, the global memory limit in GTX 580 GPU is prohibitive for processing this large input. Thus, the new K20C GPU gives slightly better performance and is scalable to much larger inputs as compared to GTX 580. Note, since our code is primarily primitive based, no tuning of the thread/block grid parameters is required to achieve speed up on Kepler. Also, any future improvements to the primitives that result from new features viz. dynamic parallelism, hyperQ etc. of the Kepler architecture will be directly inherited by our string sort.

4.3.6 Comparison with CPU Algorithms

Table 4.6 compares the performance of our GPU string sorting algorithm with the state-of-the-art CPU algorithms for string sorting: Burstsort [72, 73] and MSD radix sort [37]. For Burstsort, we use the code provided by the authors of [72]. For MSD radix sort we use the efficient code available as part of a standard string sorting library[5]. The speedup is very significant except on the very simple dictcalls dataset. This gives a feel of the speed up that can be expected using a GPU for those who use string sorting on the CPU.

4.3.7 Performance with After-Sort Tie Length

The sorted ties histogram quantifies the difficulty of sorting a given dataset [17, 37]. For each string, the average of number of prefix characters that match to the strings just before and after it in the sorted order is called its tie-length. This measure is an upper bound on the number of prefix characters that can match to the given string from the entire dataset. The histogram of these tie-lengths is called the sorted

[5]https://github.com/rantala/string-sorting

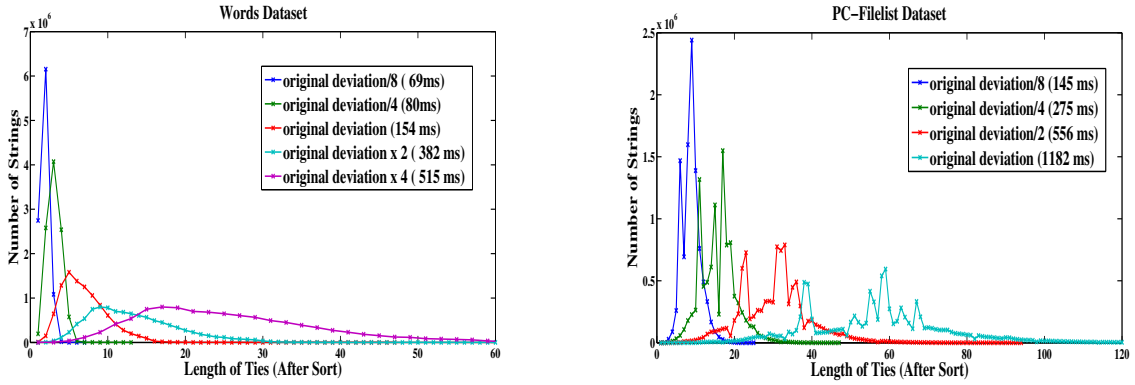


Figure 4.7: In this figure, we vary the deviation of the sorted ties histogram for two datasets: words (less ties) and pc-filelist (high ties). To reduce the deviation by a factor of k, we take every k^{th} character of each string. To increase deviation by a factor of k, each character of the string is replicated k times. The runtimes for our GPU algorithm are indicated in the legend. Datasets having histograms with low deviation are easier to sort than those with high deviation. The runtime varies linearly with the change in deviation, indicating we can even handle inputs with high ties as efficiently as possible.

ties histogram. If the sorted ties histogram is short (i.e. deviation is small) and steep, many strings have small tie lengths. Such an input is easy to sort. On the other hand, if it has a large spread (i.e. deviation is large), there are many strings with long ties and the dataset is difficult to sort.

The words dataset has strings that are smaller in length. The average tie length is 7 characters. To study the performance of our algorithm with respect to average tie-length, we modify the sorted ties histogram of this dataset as follows. We increase the deviation by a factor of k by replicating every character of the given string k times. Similarly, we reduce the deviation by a factor of k by taking every k^{th} character of each string. Note, area under the curve remains the same for all these modified histograms. From Figure 4.7, we see that when deviation changes by a factor of $1/8$, $1/4$, 2 and 4, the runtime changes by $0.4\times$, $0.5\times$, $2.4\times$ and $3.3\times$ respectively. Our runtime demonstrates a near-linear scaling with deviation. On reducing the deviation below $1/4$, the tie length becomes so low that only 1-2 sort operations are required. Similarly for the pc-filelist dataset, we change deviation by factors of $1/2$, $1/4$ and $1/8$ and obtain runtimes that are $0.47\times$, $0.23\times$ and $0.12\times$ the original runtime. The pc-filelist dataset was designed as a stress test for our algorithm, with a large number of strings (10 million) and very long ties. A near-linear performance on different spreads of histograms for this dataset shows that our algorithm can handle the entire spectrum of sorted ties histogram efficiently.

The pc-filelist dataset is such that we use up all memory available on the GPU to perform the sort. Increasing the deviation will involve replicating the characters and will require more memory. To handle such cases, we can modify our approach to stream successive prefixes of all strings to the GPU memory periodically to substitute the used prefixes. This can be explored in our future work and will allow our algorithm to scale to even larger input size. It is not as straightforward to scale merge sort algorithms,

because they will require entire strings to be in memory to perform iterative comparisons at least for the final merge step.

4.4 Discussion

We developed an efficient data parallel algorithm for the irregular problem of string sorting, based on the fast radix sort primitive in the paper. We got a speed up of more than 10 over the best GPU string sorting approaches. Our approach scales to larger datasets easily. Reducing memory movement, removing singleton segments early, and consuming maximum number of string bytes in each sort-step are the key factors behind our high performance. Most of the time was spent on optimized primitives from available libraries. We presented results on a variety of natural and synthetic datasets. The analytical expected runtime matched the actual time quite closely.

Our method is scalable to extreme conditions in the future, when the set of strings to be sorted does not fit the GPU memory. Our method can handle this by streaming parts of the dataset from the CPU as and when needed, as we need to access the string strictly from the left to the right only. Strings can thus be divided into sections by columns and streamed to the GPU. The streaming can overlap with the computations to get very high throughput in string sort.

Chapter 5

Lossless Data Compression

In this chapter, we present a parallel *all-core* implementation of an end-to-end lossless data compression application, specifically, the *Burrows Wheeler Compression* (BWC) algorithm. We combine data parallel algorithm of Burrows Wheeler Transform (BWT) with other steps on CPU using task parallelism. We solve pipelining issues discussed in Section 1.2.1 for the end-to-end application of BWC and develop a all-core BWC that optimally uses both CPU and GPU for high performance.

BWC is a popular, open compression scheme built on Burrows Wheeler Transform (BWT) [11]. It is used widely to compress regular files, system software, gene sequences, etc. BWC typically gives 30% smaller compressed files compared to LZW based schemes [1]. Compression schemes are among the hardest to parallelize on many-core architectures like the GPU due to their irregularity. The approach developed by us scales to exploit all cores – CPU, GPU and others – present on a given computer. In contrast, only block-parallel BWC approaches on multi-core CPUs have resulted in speedup previously [25]. A recent GPU BWC effort performed slower than a single core CPU [62]. We develop an intra-block, fine-grained BWC algorithm on GPU that outperforms the state-of-the-art CPU implementation by Seward [71]. We also present an all-core framework where inter-block parallelism is exploited to divide the tasks among multiple computing stations of the CPU and we use intra-block parallelism on the fine-grained (many-core) architecture of the GPU. Our results show significant performance benefits as well as effective load balancing using all cores on the system. The main contributions of our work are given below.

- We develop a fast BWT algorithm on the GPU that is built on radix sort (Section 5.2.1 and 5.4.1). We extend the GPU string sort approach developed by us in Chapter 4 [1] [19] to address the high tie-lengths common in compression datasets and use it to perform suffix sort step of BWT.

- BWT and its inverse are often used in pairs. This allows us a way to speedup BWT by modifying the strings to reduce high tie-lengths. We introduce a string perturbation step to increase speed at a slight reduction in compression ratio (Section 5.2.2). The known perturbations can be reversed after decompression to recover the original data block.

[1] Code available at http://web.iiit.ac.in/ adity.deshapandeug08/stringSort/ and https://github.com/aditya12agd5/cudpp

- We partition and efficiently pipeline the tasks between CPU core and the GPU, with naturally serial tasks performed on the CPU. The controlling CPU thread performs its tasks totally overlapped with the GPU computations, resulting in a fast hybrid BWC algorithm (Section 5.2.3).

- We develop an *all-core computation framework* to exploit all compute cores on a computer system. For task/work partitioning, we present a simple strategy using *work queues* that can solve the pipelining challenge in several applications. We use our hybrid BWC approach on the compute stations with GPUs and the optimal BWC code from Seward [71] on the CPU compute stations. Section 5.3 describes our all-core framework and its application to BWC in more detail. We modify the state-of-the-art BWC implementation to support large block sizes and concurrent processing of blocks by CPU and GPU. Our code is available for public use. [2]

- We demonstrate significant speedup as well as better compression ratios on a set of challenging datasets (Section 5.4.2 and 5.4.3). Our hybrid BWC is the first to report a speedup on the GPU. The all-core BWC produces linear speedup using only multi-core CPUs, while being compatible with present Bzip2 standards. The all-core BWC using both CPU cores and GPU gives better runtime (than multi-core CPU BWC), while balancing the load between the GPU and the CPUs.

Our implementation scales well on different combination of compute cores available on a system, providing optimal performance. On an Intel Core i7, our all-core implementation achieves a $3.06\times$ speedup, which improves to $4.87\times$ when an Nvidia GTX 580 is included (Table 5.2). On a low-end Intel Core2Duo, our all-core BWC achieves a $1.22\times$ speedup which improves to a $1.67\times$ speedup with the addition of Nvidia GTX 280 (Table 5.3). We believe that the techniques and the lessons from this work will motivate future work in designing all-core implementations for other end-to-end applications. A suite of similar end-to-end applications that exploit every compute core will truly allow the common user to enjoy the benefits of parallel computing on the desktop.

5.1 Background and Related Work

In this section we review related work on BWC, BWT and its sequential and parallel implementations.

5.1.1 Burrows Wheeler Transform

Burrows and Wheeler, developed BWT [11] with following steps: (i) Start with the input string $S[1...N]$ and associated index array $I[1...N]$, initialized to $1...N$, denoting the starting position of each (cyclically shifted) suffix in input string. (ii) Sort all suffixes in lexicographic order along with the corresponding indices I. Final suffix array is a permutation of initial index array $I[1...N]$. (iii) Compute

[2] Code for All-Core BWC available at http://cvit.iiit.ac.in/resources/bzip2GPU/bzip2Cvit.tar.gz

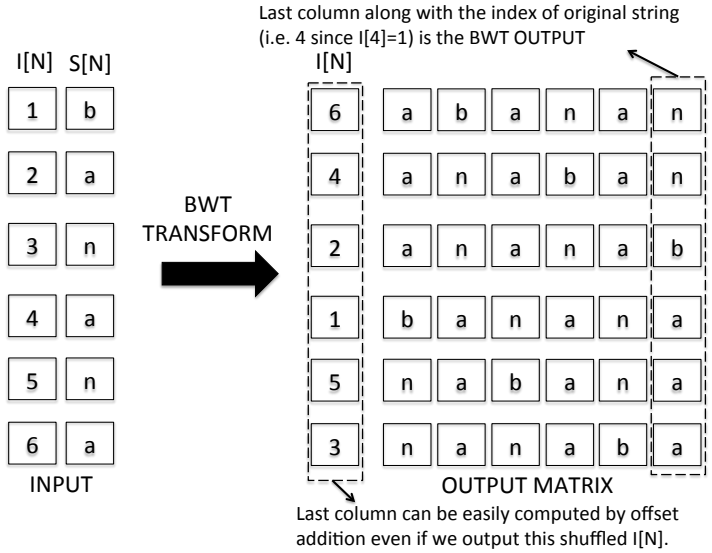


Figure 5.1: Illustration of the Burrows Wheeler Transform on the input string *banana*.

last elements of sorted suffixes and output it along with the index of the original string in the sorted output.

Figure 5.1 shows the application of BWT on an input string `banana`. It operates on all cyclic shifts or suffixes of input string $S[1...6] =$ `banana`. It generates the output matrix containing a sorted list of all cyclically shifted strings. The answer is the last column of output matrix, i.e. `nnbaaa`, appended with number `4`, since the original string occurs at 4^{th} position in the output. In general, the suffix sorting step of BWT is compute intensive because it involves sorting $O(N)$ suffix strings each of length $O(N)$ and these strings also have a high match length (i.e. they share long common prefix). High match length is a characteristic of compression datasets, because compression schemes are typically used when data has redundancy. This redundancy results from long repeating substrings, which causes high match length for suffix strings. For example, if there are 2 same substrings, say a few 1000 characters long, that are repeated in the input, then many suffix strings will have these few 1000 characters matching. Resolving ties between all these strings during sorting is expensive. In practice, cyclically shifted suffix strings match to lengths as high as 10^3 to 10^5 characters within a 9M character block. This shows the compute intensive nature of BWT.

5.1.2 Sequential Burrows Wheeler Compression

BWT was used to devise a lossless BW compression scheme by Burrows and Wheeler [11]. For suffix sort, they performed a radix sort on first two characters (c_1c_2) of all suffixes followed by a modified Quicksort [8] in subsequent iterations. They also developed a special mechanism to handle inputs with long repeated runs on the same character. Their BWC was slow. Seward proposed a method that generated 256 depth one buckets based on the third character (c_3) after the two-character radix sort

[71]. Within each bucket, they sorted only those suffixes starting with c_3c_4 (such that $c_3 \neq c_4$) and cleverly synthesized sorted order for suffixes in other buckets (of the form c_*c_3). This resulted in an efficient and popular Bzip2 file compressor. Incorporating Sadakane's algorithm [68] improved the performance even on worst case inputs. Synthesizing sorted order of new suffixes from previously sorted ones avoids expensive matching and results in good performance of these methods. Such fine-grained synthesis methods require synchronization and are difficult to do on the GPU architecture. Our GPU implementation uses a coarse synthesis technique developed by Kärkkäinen and Sanders [38], wherein after sorting only $(2/3)^{rd}$ of the suffix strings, the rest can easily be sorted and merged.

5.1.3 Parallel Burrows Wheeler Compression

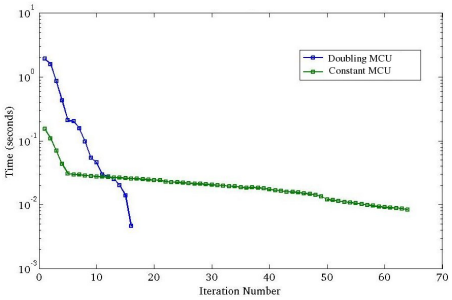
Gilchrist and Cuhadar [25] exploited the inter-block parallelism for linear speedup with multiple CPU cores for BWC. They focussed on naive parallelization without modifying the basic algorithm or making it scale to larger input size.

Previous GPU BWC. The only prior GPU implementation of BWT by Patel et al. [62] repeatedly sorts strings using a variable length key comparison based sort [17]. Their implementation was about $2.78\times$ slower (with BWT step dominating the runtime) than the CPU version due to the inherent difficulty of parallelizing BWC. Another attempt at building parallel BWC was abandoned due to very poor performance [12]. We build our BWT on a GPU string sorting algorithm developed by us [19] that uses radix sort. Our GPU BWT implementation achieves more than $2\times$ speedup. Our implementation scales to larger block sizes giving better compression ratios.

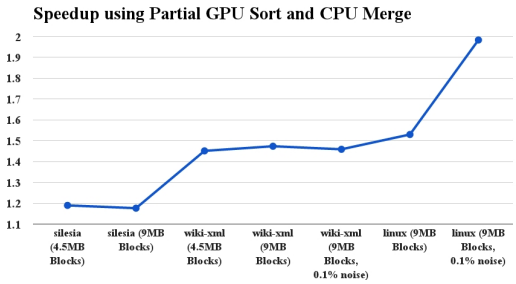
Edwards and Vishkin BWC. In parallel with our work, Edwards and Vishkin [20] developed an intra-block parallel BWC algorithm and compared it with the CPU BWC [21]. Their algorithm is work optimal with an $O(\log N)$ time on architectures with fine-grained parallelism. They demonstrated it on their Explicit Multi-Threading (XMT) architecture but not on multi-core CPUs. In contrast, we demonstrate better performance on multi-core CPUs and GPUs. They report a speedup of 1.8 to $2.8\times$ on XMT-64 (~ 64 cores) platform and 12 to $25\times$ on a simulated XMT-1024 (~ 1024 cores) platform. They use files from *Large Corpus*[3], which are small in size ($< 4.5MB$'s) by today's standards and have a low maximum sorting depth (< 2000). We demonstrate high performance on large datasets with higher sorting depth (10^3 to 10^5) like Enwik8 (96MB) [48], Linux-2.6.11.tar (199MB) [47] and *Silesia Corpus*[4] (203MB, which supersedes *Large Corpus*) [18]. It would be worthwhile to compare the runtime of our algorithm with Edwards and Vishkin [20] on same platforms and same test datasets in the future.

[3] http://corpus.canterbury.ac.nz/descriptions/

[4] http://sun.aei.polsl.pl/ sdeor/index.php?page=silesia



(a) Sorting with Doubling MCU is expensive but requires very few iterations, while Constant MCU sorting is faster but takes very many iterations. Thus, we use constant MCU for first few iterations and then resort to doubling MCU.



(b) Speedup of 1.2 to 2× is obtained by using our partial sort and merge strategy as compared to performing full sorting on different datasets.

Figure 5.2: Performance benefits of Doubling and Partial Sort & Merge optimization.

5.2 CPU and GPU Hybrid BWC Algorithm

Burrows Wheeler Transform (BWT) is the most crucial step of BWC and one that occupies a significant chunk of its runtime. Computing BWT is equivalent to suffix sorting of all suffix strings (shown in Figure 5.1) of the input string. We can treat each suffix as a separate string and perform a string sort. In principle, string sorting can be performed on GPU by using a custom comparator in a GPU merge sort algorithm[5] [62, 17]. This custom comparator iteratively compares two strings. In a merge sort, input is recursively split into small buckets and these buckets can be sorted independent of each other. Parallelism of GPU is also leveraged to perform merging of buckets co-operatively using 2 or more thread blocks/SM's. But, the iterative comparisons discussed earlier are performed at *every* merge step. Moreover, to perform such comparisons during successive merge steps, same string is loaded again and again from the high latency global memory. Threads in a warp diverge because of varying length comparisons. A traditional GPU merge sort based string sort runs $2-5\times$ slower than CPU when used to perform suffix sorting [62]. Thus, instead we use the radix sort based string sorting procedure developed in the previous chapter. Exploiting fast standard GPU primitives and reducing high latency global memory accesses, allows our radix sort based GPU string sort to provide $10\times$ improvement over previous methods. We leverage this fast string sort along with additional techniques to perform BWT step of BWC.

5.2.1 Modified String Sort for BWT

For BWT, all N suffix strings are each of length N and these strings share long matching prefixes with each other. This is because, compression schemes are typically used on datasets with high redundancy or match length (in form of matching substrings). Matching substrings give rise to matching

[5] http://goo.gl/mlwlZ

prefixes for suffix strings. Long matching prefixes results in large number of iterations for the string sort algorithm. The string sorting approach works well only when the input has ties up to a few 100 characters [19]. Suffix sorting step of BWT has relatively much higher number of ties (10^3 to 10^5 characters). In such a scenario sorting using constant-sized MCUs (k characters long) takes too much time. Thus, we perform costly sorts on longer MCUs, but reduce the number of iterations by doubling the MCU length after every iteration. We use the property of suffix strings being cyclically shifted to only sort a subset of strings and generate the sorted order for rest. These two sets of sorted suffix strings are then merged by utilizing idle CPU cycles. These BWT specific optimizations are discussed in more detail in subsequent sections –

Doubling MCU Length. The match length between suffix strings (i.e. length of longest common prefix of all suffix strings) determine the maximum number of fixed-length sorts required. This is referred to as the sorting depth. We use the terms match length and sorting depth interchangeably depending on the context. Large sorting depths result from long substrings repeating many times, which degrades the GPU BWT performance. To address this, we double the MCU length after a few steps. This reduces the number of sort steps as longer substrings are being compared in each successive iteration. As shown in Figure 5.2(a), if we use doubling right from the start, initial sort steps are very costly as compared to constant sized (k character) MCUs. But, the total number of sort steps with doubling are very less. To obtain the best results, we use the faster constant size MCU for the first 16 iterations and then double the MCU length. For example, one input dataset has a sorting depth of 960 characters. We double the MCU length after 16 sort steps. So, we perform $16 * 4 = 64$ character comparisons with 4-byte MCUs. Now, only 7 more sort steps are required to cover the remaining 896 ($8 + 16 + 32 + ... + 512 > 896$) characters. Doubling MCU length after 16 sort steps gives us a speedup of $1.8\times$ (1.06s to 0.58s per block) as compared to using constant sized MCUs in this particular case. Each sort step with longer MCU takes more time compared to the case of constant sized MCU, but the large drop in number of sort steps results in a much improved overall performance.

Partial GPU sorts and CPU merge. In suffix sort, it is not necessary to sort all strings, we can sort only a subset of the original strings and synthesize the sorted order for rest. This synthesis is possible because the input strings are cyclically shifted. Suppose, we sort all strings at indices $i \pmod 3 \neq 0$ (denoted by set $I_{1,2}$), we can generate the sorted order for all suffixes at $i \pmod 3 = 0$ (denoted by set I_0). This is done as follows: we compare the first character of the two I_0 strings, if it is unequal we obtain the sorted order. If it is equal, the strings beginning from next characters of both suffixes correspond to suffixes in $I_{1,2}$, for which we already know the sorted order. Also, it is easy to merge I_0 and $I_{1,2}$, since in at most two comparisons of next characters, we hit two suffixes that belong to $I_{1,2}$. Similar sort and merge approaches have been in practice in the CPU suffix sorting literature [38, 39, 71]. We adapt the efficient approach by [38] to our GPU implementation. We perform the two sorting steps on GPU and move the merge step to CPU as shown in Figure 5.3. This allows us to utilize the idle CPU

cycles while GPU is performing sort operation for subsequent block in BWC. Figure 5.2(b) shows that we obtain a speedup of 1.2 to $2\times$ as a result of using this optimization.

The optimizations discussed above are limited only to the BWT step of BWC. Specific to the problem of BWC, we develop another optimization to forcefully break long ties in the input. This is discussed in Section 5.2.2.

5.2.2 String Perturbation

Large sorting depth comes from repeated long substrings. Runtime can reduce greatly if we can reduce long ties by perturbing the string. This works for BWT based compression schemes because a known perturbation can be undone after decompression. Different perturbations were tried by us. Inserting a random character at fixed positions in the input string worked the best, as it forcefully breaks long ties. For example, on a 4.5M character block of *linux-2.6.11.tar* adding a random character after every 1000^{th} character reduced the maximum sorting depth from 65472 to 960 characters and the average sorting depth from 10078 to 825 characters (Figure 5.5, bottom right). It should be noted that the BWT for this modified input string is not the same as BWT of the original input string. Since BWT and inverse BWT (IBWT) are used in pairs, random characters that occur at fixed positions can be removed to restore the original string after IBWT. The compressed file size increases slightly as we increase the entropy by adding random characters, but our results (Section 5.4.2) show that this increase is reasonable. This optimization also provides us a way to trade-off compression time against compression ratio. Figure 5.5 (bottom right) and Table 5.1 demonstrate the significant improvement ($8.2\times$ speedup for linux-2.6.11.tar with 0.1% perturbation on 9MB blocks) in runtime obtained after using string perturbation. The speedup obtained by string perturbation is very useful on datasets with very high sorting depths viz. *linux-2.6.11.tar*.

5.2.3 Mixing Data Parallelism with Task Parallelism: Hybrid BWC Algorithm

We have developed a hybrid BWC algorithm that partitions task between a single CPU core and the GPU. In the design of our hybrid BWC algorithm we take into account differences between CPU and GPU and map the appropriate operations to the appropriate compute platform. The BWC algorithm consists of three steps: (i) Burrows Wheeler Transform, (ii) Move to Front Transform (MTF), and finally (iii) Huffman encoding. Patel et al. [62] implemented all three stages on the GPU with the hope of performing on the fly compression/decompression during data transfers. All three stages were individually slower on the GPU as compared to the CPU, with BWT step experiencing the maximum slowdown. Typically BWT computation itself takes about 80-90% of the total computation time on the CPU. The MTF and tree building step of Huffman coding are completely serial and it is difficult to extract performance by mapping these to a data-parallel model. Based on these observations, we perform the bulk of BWC computation i.e. BWT operation on the GPU (using steps discussed in Section

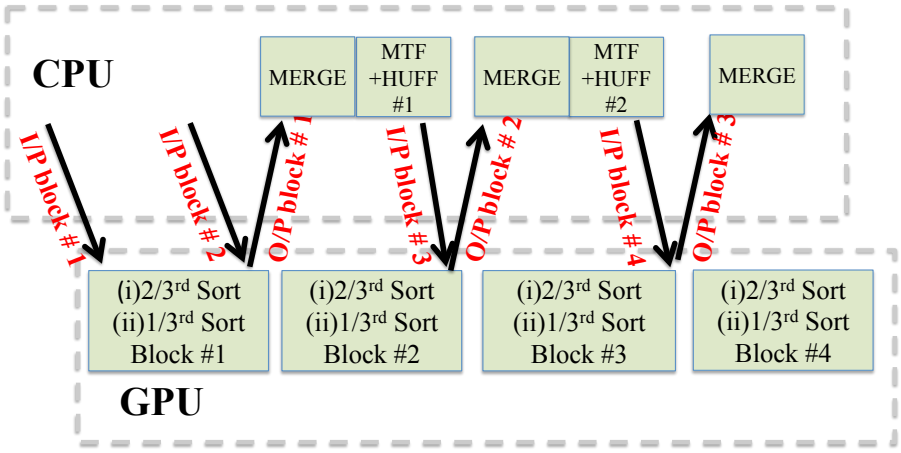


Figure 5.3: An illustration of the CPU+GPU hybrid BWC pipeline. The merge, MTF and Huffman steps are done on the CPU in a fully overlapped manner with the partial sorts on the GPU of succeeding block.

5.2.1) and we perform the remaining computations of MTF and Huffman encoding on the controlling CPU thread. Also note that, as discussed earlier, during BWT the merge step after partial sorts is also performed on CPU. This hybrid BWC is illustrated in Figure 5.3. Barring the last block, the merge, MTF and Huffman operations of all the blocks are performed in a fully overlapped manner with the partial sorts on the GPU. This makes good use of the idle CPU cycles, when GPU is busy doing the sort operation, and provides a throughput for BWC that nearly equals the GPU BWT throughput.

The pseudo code of our hybrid BWC algorithm is given in Algorithm 4. Our algorithm takes as input a file $\mathbf{F}$ (line 1) and splits it into multiple blocks ($[\mathbf{B_1}, \mathbf{B_2}, \cdots, \mathbf{B_n}]$) each of size $\mathbf{N}$ (line 2). These blocks are perturbed (line 3) and then undergo sort steps of BWT on the GPU and remaining merge, mtf and huffman encoding steps on the CPU one after the other (line 4). We create an index array, $\mathbf{I}$, which denotes the starting position of each suffix string (line 5). Note that, since cyclically shifted suffix strings are used during BWT, each of the $\mathbf{N}$ strings also has a length of $\mathbf{N}$. We use modified string sorting method described above, along with doubling MCU optimization to sort strings that occur at positions $\mathbf{I_{1,2}}$ (line 8). After this string sort, $\mathbf{I_{1,2}}$ contains the indices of strings in sorted order. We use this sorted order to non-iteratively synthesize the sorted order for strings at positions $\mathbf{I_0}$ (line 9). The results of partial sorts are handed over to the CPU for performing merge and remaining BWC steps (line 10 to 12), and GPU simultaneously begins the sort steps on the next block. The sort steps on GPU are thus overlapped with other BWC steps on the CPU as shown in Figure 5.3.

In our algorithm for all operations on GPU, we use the fastest sort, scatter and scan primitives. These primitives are tuned for every new architecture and there are also algorithmic improvements which improve their performance. Our GPU BWC, built on these primitives, can directly inherit all these improvements and is adaptable to future architectures without requiring any re-design. The design of our hybrid BWC algorithm makes best use of both CPU and GPU. Table 5.1 and Figure 5.6 show that our hybrid BWC gives a max. $2.9\times$ speedup over standard CPU BWC implementation i.e. Bzip2.

Algorithm 4 Hybrid Burrows Wheeler Compression Algorithm

1: Input: File $\mathbf{F}$
2: $[\mathbf{B_1}, \mathbf{B_2}, \cdots, \mathbf{B_n}]$ = split-blocks($\mathbf{F}$, $\mathbf{N}$) *// split file into size N blocks*
3: $[\mathbf{B_1}, \mathbf{B_2}, \cdots, \mathbf{B_n}] \leftarrow$ perturb-blocks($[\mathbf{B_1}, \mathbf{B_2}, \cdots, \mathbf{B_n}]$, **interval**) *// random char is added after* **interval**
4: **for** $\mathbf{B_p}$ in $[\mathbf{B_1}, \mathbf{B_2}, \cdots, \mathbf{B_n}]$
5: $\mathbf{I} = [\mathbf{1} \cdots \mathbf{N}]$ *// Index Array, denotes starting position of each string*
6: $\mathbf{I_{1,2}} \leftarrow \{\mathbf{I[i]} \quad | \quad \mathbf{I[i]}(\mathbf{mod} \quad \mathbf{3}) = \mathbf{1} \text{ or } \mathbf{2}\}$
7: $\mathbf{I_0} \leftarrow \{\mathbf{I[i]} \quad | \quad \mathbf{I[i]}(\mathbf{mod} \quad \mathbf{3}) = \mathbf{0}\}$
8: $\mathbf{I_{1,2}} \leftarrow$ GPU-modified-string-sort($\mathbf{B_p}, \mathbf{I_{1,2}}$, **lim**) *// MCU doubled after* **lim** *iter.*
9: $\mathbf{I_0} \leftarrow$ GPU-non-iterative-sort($\mathbf{B_p}, \mathbf{I_{1,2}}, \mathbf{I_0}$) *// synthesize sorted order*
/ CPU computation is fully overlapped with asynchronous GPU calls */*
10: $\mathbf{I} \leftarrow$ CPU-merge($\mathbf{B_p}, \mathbf{I_{1,2}}, \mathbf{I_0}$) *// non-iterative merge of two sorted suffix sets*
11: $\mathbf{bwt} \leftarrow$ CPU-offset-addition($\mathbf{I}, \mathbf{B_p}, \mathbf{N}$) $\oplus$ index-of($\mathbf{0}, \mathbf{I}$) *// generate BWT*
12: $\mathbf{result} \leftarrow$ CPU-mtf-huffman($\mathbf{bwt}$) *// perform MTF and Huffman encoding*
13: end **for**

In our hybrid BWC we only use a single CPU core. We further improve our speedup by using the other idle CPU cores through our all-core framework as shown in Tables 5.2 and 5.3.

5.3 The All-Core Framework

The *all-core* framework shown in Figure 5.4 is detailed in this section. A typical computation platform consists of multi-core CPUs, many-core GPUs, and/or other accelerators. The specific platform we focus on consists of a multi-core CPU and a GPU, but the framework extends easily to others. We treat each CPU thread as a compute station or *CoSt*. The number of CoSts can exceed the number of CPU cores with hyperthreading. Each GPU with a controlling CPU thread is another CoSt. The GPU programming models are getting more flexible and multiple CoSts may coexist on a physical GPU in the future. Each CoSt can be assigned a specific task. Each CoSt is free to choose the best possible strategy to perform the given task. A CPU core as a CoSt will attack the problem sequentially while a GPU as CoSt will resort to data-parallelism. For task partitioning we present a simple, but generic strategy based on *work queues* that can be used for several problems. Each CoSt dequeues an appropriate task from the work queue, processes it, and enqueues the results to an output queue. This framework is best suited for applications that process large data, but in independent blocks. There may be some pre-processing before independent blocks are formed and some post-processing to combine the outputs of independent blocks. Several applications fit the work queue model such as video encoding, decoding, and transcoding, lossy or lossless data compression, etc. We confine our attention to BWC compression application in this paper, though extension to other applications is straightforward.

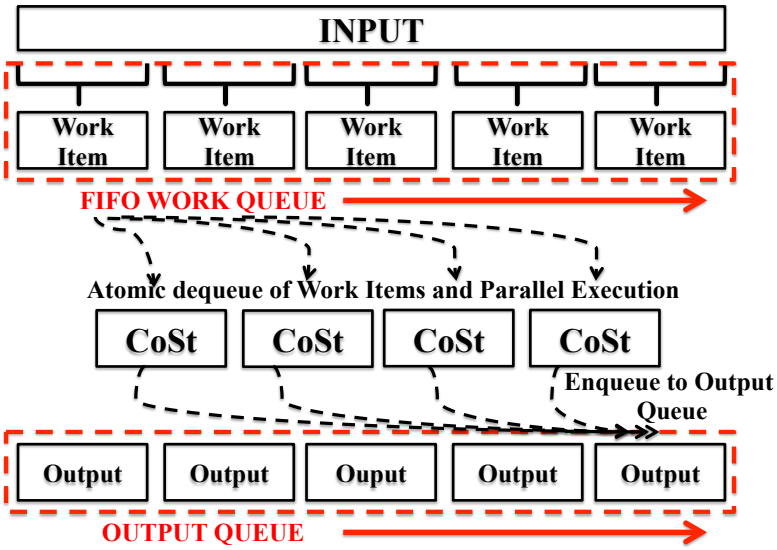


Figure 5.4: Work Queue based all-core framework.

5.3.1 BWC in All-Core Framework

BWC on large data buffers (files or others) is performed by dividing it into blocks which are processed independently. A CoSt processes a block. We add the entire data buffer to a work-queue. Each CoSt, when free, removes a work item of appropriate size from the queue and processes it. When done, it adds the output – a compressed bit stream in case of BWC – to the output queue and gets another item from the queue for processing. A manager thread on CPU performs the post-processing, which involves combining the output bit streams, adding headers, etc., and creates the final output. Each CoSt performs the same task, but uses an implementation that best fits its architecture. Since our target architecture has two types of CoSts, (i) CPU core and (ii) GPU with a single controlling CPU thread, we use two BWC implementations. CPU CoSts use the best sequential implementation of BWC. On the GPU we use our hybrid BWC algorithm. We call this implementation which uses both, CPU (BWC by [71]) and GPU (our hybrid BWC as described in 5.2.3), as the all-core BWC. It is also possible to use only the CPU cores in our all-core framework, this gives us a multi-core BWC (which runs in parallel on multiple CPU cores). The multi-core BWC is similar to parallel implementation by [25], but it is fully compatible with the Bzip2 compression standard. In Table 5.2 and 5.3 we show the speedup achieved using all-core BWC and multi-core BWC. These results show that our all-core BWC improves on the speedup obtained by using only CPU cores in the multi-core BWC. The development of hybrid BWC enables our all-core approach to achieve this additional speedup. Our work-queue based all-core framework keeps all the CoSts busy while there is more work to do. Thus, the framework provides great flexibility in mixing different types of compute stations while balancing the loads on them according to their capacities.

5.3.2 BW Decompression in All-Core Framework

Burrows Wheeler decompression has the same block level parallelism as compression and we use our all-core framework to gain speedup. We develop an all-core implementation of decompression where GPU CoSt performs the Inverse BWT step according to algorithm given by Patel [61]. This algorithm involves 3 steps of sorting (1 byte key), list ranking and scatter. Efficient primitives to perform all these operations are available on the GPU. Since inverse BWT is a simple operation, GPU does not achieve a significant speedup over CPU. Thus, the overall speedup is linear in number of CoSt's used through our all-core framework. BW compression is a tougher problem than decompression and we choose to focus on it in this work.

5.4 Experimental Results

We evaluate the performance of our GPU BWT (Section 5.4.1), hybrid BWC (Section 5.4.2), multi-core and all-core BWC (Section 5.4.3) on different datasets and on different CPU and GPU platforms. The GPU BWC (Algorithm 4), was implemented on Nvidia GPUs using Thrust primitives for sort (key and value), scatter and scan [32]. The following standard datasets for lossless data compression were used in our experiments:

- Enwik8 [48]: The first 10^8 bytes of the English wikipedia dump on March 3, 2006. We also use wikipedia's enwiki-latest-abstract-10.xml (henceforth, referred to as wiki-xml) dataset [79].
- Publicly available source of linux kernel 2.6.11 (199MB) [47].
- Silesia data corpus, a widely used standard data compression benchmark which has large files from various sources viz. database, codes, medical images etc. [18]. We tar (concatenate) all the files and use the tarred file (silesia.tar) as a dataset.

In our results, we benchmark the performance of our GPU BWT against the state-of-the-art BWT on CPU, i.e. BWT implemented in Bzip2 file compressor [71] and we show the effectiveness of our string perturbation optimization for some datasets (Section 5.4.1). We also compare the performance of our hybrid BWC pipeline against single-core CPU BWC (Section 5.4.2). These results show that our hybrid BWC pipeline performs about $2.9\times$ better than the highly tuned CPU BWC and thus, about $8\times$ faster than the previous GPU BWC [62] (which was $2.78\times$ slower than CPU). The reader should note that this is the first time a speedup has been achieved on GPU for the problem of BWC. Our hybrid BWC also gives better compression ratio in lesser time by using large block sizes, as compared to the maximum 900KB block size supported by standard CPU BWC. Finally, we show the speedup we achieve through multi-core BWC and our all-core BWC implementation on a high-end as well as a low-end system (Section 5.4.3). The dataset and the code used for our experiments is available at `http://cvit.iiit.ac.in/resources/bzip2GPU/bzip2Cvit.tar.gz`.

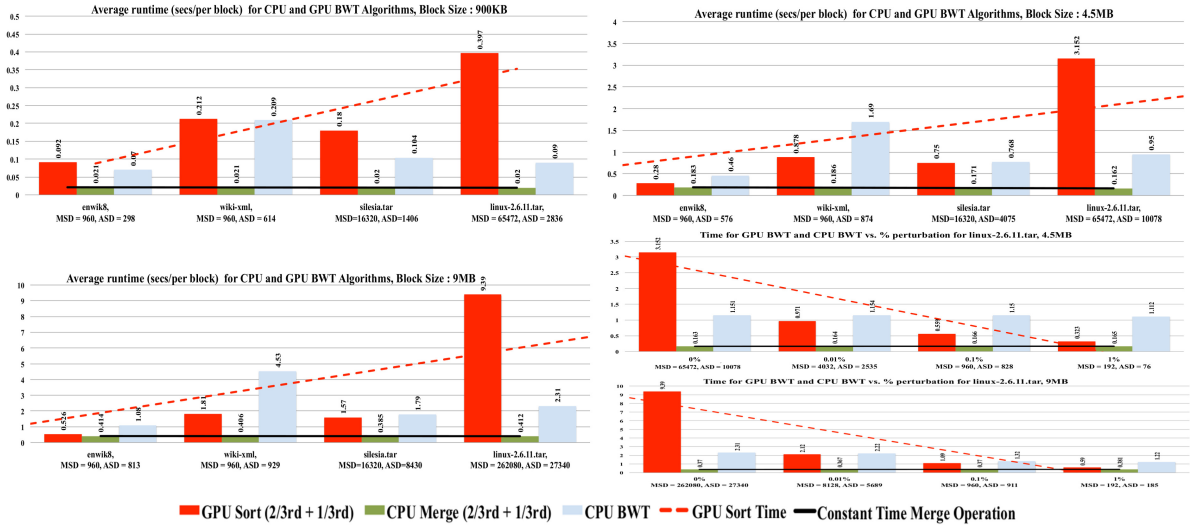


Figure 5.5: The performance of our GPU BWT vs. CPU BWT (Bzip2) for different block sizes and on datasets with different maximum/average sorting depths (MSD/ASD). GPU BWT is $2\times$ faster on large block sizes for enwik8 and wiki-xml datasets. The GPU performance degrades on very high sorting depth datasets (viz. linux-2.6.11.tar) which is addressed using string perturbation (bottom right). With increase in % of string perturbation the sorting depth reduces, and for very high sorting depth (worst-case) linux dataset, beyond 0.01% perturbation the GPU BWT achieves a speedup over CPU.

5.4.1 Results: GPU BWT

We benchmark the performance of both CPU and GPU BWT algorithms by varying block size (900Kb to 9MB) and measuring the runtime on datasets with different sorting depths, i.e. for 9MB blocks the maximum sorting depth varies from 960 (enwik8) to $2,62,080$ (linux-2.6.11.tar). The BWT implementation in Bzip2 on the CPU is considered to be the state-of-the-art and we use it for comparison. Figure 5.5 shows that our GPU algorithm achieves speedup for large block sizes ($> 4.5MB$) since the GPU is not utilized well for smaller ones. We later demonstrate that these large block sizes also provide better compression (Section 5.4.2). Figure 5.5 shows that the runtime for GPU sort operations increases with increase in the sorting depth (indicated by dashed red line). On large block sizes, we achieve a speedup whenever the maximum sorting depth is upper bound by about 10^4. The maximum speedup, of $2.5\times$, is achieved for wiki-xml using 9MB blocks. For datasets, with an order of magnitude higher sorting depth viz. linux-2.6.11.tar, the GPU BWT is about $4\times$ slower than the CPU. This is addressed using string perturbation.

Effect of String Perturbation. In Figure 5.5 (bottom right), we vary the % of perturbation from 0 (0 random characters added) to 1% (1 random character after every 100^{th}) for the high sorting depth (practical worst-case) linux-2.6.11.tar dataset and measure the runtime. As we increase the % perturbation, both the maximum and average sorting depth reduces (since long ties are broken by random characters) and runtime improves (indicated by dashed red line). We see that at 0.01% perturbation itself, the GPU

Dataset (Size)	Block Size	(i) Compression time for our hybrid BWC (s), (ii) Compression time for CPU BWC (s) [71], (iii) Compressed file size in MB's (same for both)			
		0% Perturbation	*0.01% Perturbation*	*0.1% Perturbation*	*1% Perturbation*
enwik8 (96MB)	900KB	**10.07**, 10.81, 27.66	**10.03**, 10.85, 27.70	**9.91**, 10.88, 28.09	**8.87**, 10.97, 31.32
	4.5MB	**7.29**, 13.12, **25.62**	**7.29**, 13.11, **25.67**	**7.31**, 13.10, **26.09**	**7.60**, 13.22, 29.36
	9MB	**8.31**, 15.23, **24.86**	**8.30**, 15.22, **24.91**	**8.33**, 15.82, **25.33**	**8.63**, 15.27, 28.66
wiki-xml (151MB)	900KB	**36.88**, 38.29, 15.29	**36.56**, 38.16, 15.39	**33.85**, 37.63, 16.19	**23.49**, 32.07, 21.89
	4.5MB	**30.42**, 60.78, **13.66**	**30.14**, 60.76, **13.77**	**26.97**, 60.55, **14.55**	**15.96**, 48.52, 19.82
	9MB	**31.51**, 80.76, **13.13**	**31.12**, 80.77, **13.24**	**27.62**, 79.94, **14.04**	**15.79**, 66.12, 19.07
linux-2.6.11.tar (199MB)	900KB	84.86, 24.93, 35.35	48.01, 24.69, 35.46	32.84, 23.21, 36.44	**22.78**, 22.17, 44.19
	4.5MB	133.54, 45.66, **33.10**	41.37, 44.02, **33.23**	**24.17**, 39.88, **34.26**	**14.24**, 26.65, 42.31
	9MB	196.64, 53.59, **32.51**	45.55, 51.77, **32.65**	**23.81**, 32.11, **33.69**	**14.37**, 29.64, 41.81
silesia.tar (203MB)	900KB	39.56, 29.65, 52.06	36.14, 29.69, 52.17	**28.98**, 29.32, 52.97	**23.06**, 27.46, 59.49
	4.5MB	34.60, 39.57, **50.06**	**29.52**, 39.63, **50.19**	**22.97**, 32.67, **51.03**	**16.81**, 36.07, 57.54
	9MB	36.10, 46.73, **49.57**	**28.85**, 46.92, **49.70**	**24.55**, 46.31, **50.55**	**17.74**, 41.94, 57.11

Table 5.1: This table shows impact of block size, string perturbation on runtime and compressed file size (CPU BWC runtime is that of the standard `Bzip2` and the GPU BWC runtime is that of our `hybrid BWC` implementation). Bold values indicate cases where we get either better compression and/or runtime compared to the baseline i.e. standard CPU BWC on the default 900KB blocks (denoted by underline).

performs marginally better than the CPU. Beyond 0.01% perturbation, the speedup of GPU over CPU becomes significant. On linux-2.6.11.tar dataset with 9MB blocks the GPU algorithm goes from being $4\times$ slower to $1.7\times$ faster as the perturbation varies from 0 to 1%.

Time for Merge. Figure 5.5 also shows that the time required for the overlapped CPU merge operation is smaller than the time of the GPU sort step and it is constant for a given input size, no matter what the dataset (indicated by a solid black line). This allows us to fully overlap the merge computation with the sort operation on GPU.

5.4.2 Results: Hybrid BWC

We also measure the runtime of our hybrid BWC (GPU performs the partial sorts and CPU performs the merge, MTF and Huffman in a fully overlapped manner as described in Section 5.2.3) against the state-of-the-art CPU BWC implementation in Bzip2 file compressor. Table 5.1 gives the total runtime and compressed file size on different datasets using different block sizes and with varying percentage of perturbation. Our hybrid BWC is fully compatible with the Bzip2 standard and for any given input it generates the same compressed file as the CPU Bzip2 (provided both algorithms use the same block size and perturbation). Larger block size provides better compression, this is because BWT can now group together characters from a larger area for the MTF and Huffman steps and this has already been demonstrated by Burrows and Wheeler [11]. The performance of our hybrid BWC algorithm improves with increase in block size (except only for linux dataset with no perturbation), while the CPU performance becomes worse. For wiki-xml dataset, from 900KB to 9MB block size (without perturbation)

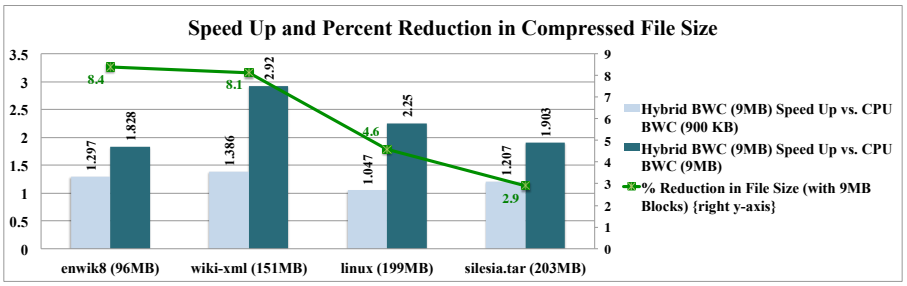


Figure 5.6: Our hybrid BWC (with 9MB blocks) pipeline performs marginally better than CPU BWC with 900KB blocks (which does much less work) and gives max. $2.9\times$ speedup when compared to CPU BWC with 9MB blocks. Using 9MB blocks also gives some gain in compression ratio.

the runtime of our hybrid algorithm improves by 15% (36.88s to 31.51s) while the runtime of the CPU BWC more than doubles (38.29s to 80.76s). Also, the compressed file size reduces by 14% (15.29 to 13.13MB) with 9MB blocks. Thus, our hybrid BWC scales better with block size and can be used to obtain better compression in lesser time as compared to CPU BWC.

To further improve our speedup and address worst-case datasets viz. linux-2.6.11.tar, we use string perturbation. We see that with increase in perturbation (random characters added), the runtime of CPU BWC is nearly same but the runtime of our hybrid BWC improves. Even for the worst-case linux dataset we beat the CPU with perturbation $>= 0.01\%$ and on large blocks. Through perturbation we are adding additional entropy to the input and the compressed file size increases. The current state-of-the-art runtime/compression is provided by CPU BWC running with 900Kb block size and no perturbation (marked by underline). Table 5.1 shows that with 0.1% perturbation and block size greater than $4.5MB$, we obtain better runtime as well as compression (marked by bold values) on all 4 datasets as compared to the state-of-the-art (marked by underline). The light-blue bars in Figure 5.6 show the speedup ($1.04 - 1.38\times$) obtained with respect to BWC on 900KB blocks and green line in the same figure shows the corresponding reduction in compressed file size ($2.9 - 8.4\%$). Note that, in comparison to the state-of-the-art, our hybrid BWC implementation (using 9MB Blocks) is outperforming the CPU, even when the CPU is doing much less work (BWT performance is worse than linear in block size) by using only 900KB blocks. This is a significant improvement over previous GPU implementation which was $2.78\times$ slower and also slightly worse in compression ratio as compared to CPU [62]. Also, to achieve the same compression ratio (using large block size) on the CPU will take much more time as compared to our hybrid algorithm (the speedup when both standard CPU and our hybrid BWC uses 9MB block size is indicated by dark-blue bars in Figure 5.6). In practice, 0.1% perturbation allows us to improve the runtime for our hybrid BWC and still keeps the compressed file size below the state-of-the-art (obtained by CPU BWC on 900KB blocks). The gain ($2.9 - 8.4\%$) in compression ratio is indicated by green line in Figure 5.6. In the results that follow, we fix our block size to 9MB and perturbation to 0.1%.

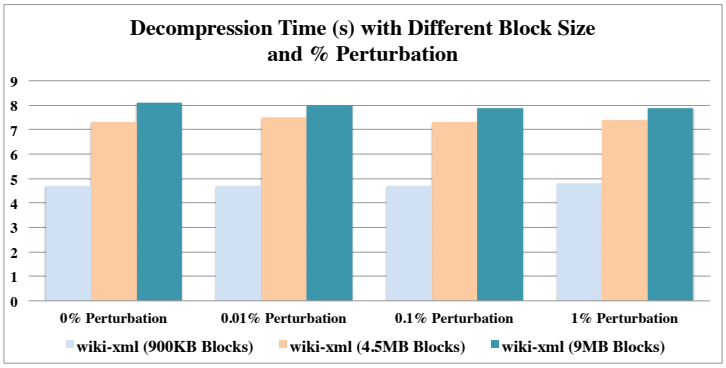


Figure 5.7: The decompression time (about 7s) is relatively small as compared to the compression time (about 30s) for wiki-xml dataset. The runtime increases slightly with increase in block size and % perturbation does not affect the decompression runtime.

Decompression. In Figure 5.7 we measure the change in the runtime for BW decompression with different block size and % perturbation. We see that there is a marginal increase in runtime when the block size increases and no change with perturbation. Thus, our modifications do not have a drastic impact on the decompression runtime. We leverage the block-level parallelism present in BW decompression to gain a linear speedup with CoSts through our all-core framework.

5.4.3 Results: All-Core BWC

We use the work queue based all-core framework and run multi-core BWC (i.e. only the CPU cores) and all-core BWC (which uses both all CPU cores and GPU) on all our datasets. The block size is fixed to 9MB and perturbation to 0.1%, since we obtained optimal performance for these parameters (Table 5.1).

High-End System. In the first setup, we compare the performance of the multi-core and all-core BWC on a high end system comprising of Intel Core i7 920 CPU and Nvidia GTX 580 GPU. The results are given in Table 5.2. The 4-core Intel i7 CPU, with hyper-threading supports 8 threads at a time efficiently and in practice, increasing the number of threads above 8 did not improve the performance. Thus in our experiments we vary the number of CPU threads between 1 to 8. In this range, as expected, for both implementations, the performance generally improves with more threads. Also in Table 5.2, on adding a single additional CPU thread to the CPU+GPU thread, we see that GPU still processes 9 out of 12 blocks, 13 out of 18 blocks for enwik8 and wiki-xml respectively. This reaffirms that our hybrid BWC is 2 times faster as compared to CPU BWC on these datasets and the work-load is balanced according to speed of CoSts. The best runtimes on enwik8, wiki-xml, linux and silesia.tar datasets for the multi-core BWC are $4.9s$, $26.1s$, $9.9s$, $17.4s$ respectively, which improve to $3.9s$ ($1.25\times$), 16.4 ($1.59\times$), $7.7s$ ($1.28\times$) and $11.0s$ ($1.58\times$) using the all-core BWC. If we look at the speedup with respect to the single-core CPU BWC implementation, we see that our all-core BWC achieves a consistent speedup greater

NVIDIA GTX 580 (GPU) + Intel Core i7 920 (CPU)								
Dataset	**Total time for all-core BWC (CPU+GPU) (s), (#blocks processed by GPU / #total blocks)** Different number of CPU threads in addition to one CPU+GPU thread							Speedup (bold) vs. single CPU (underlined)
	0 CPU	**1 CPU**	**2 CPU**	**3 CPU**	**4 CPU**	**6 CPU**	**7 CPU**	
enwik8	8.4 (12/12)	6.5 (9/12)	5.8 (7/12)	4.7 (6/12)	4.7 (5/12)	4.6 (5/12)	**3.9 (4/12)**	4.05
wiki-xml	27.6 (18/18)	23.6 (13/18)	19.6 (10/18)	16.4 **(10/18)**	16.6 (8/18)	18.9 (7/18)	18.6 (6/18)	4.87
linux	23.8 (22/22)	14.3 (13/22)	10.88 (8/22)	9.3 (8/22)	9.1 (7/22)	**7.7 (5/22)**	7.9 (4/22)	4.16
silesia	24 (23/23)	16.8 (16/23)	13.3 (12/23)	12.6 (12/23)	12.7 (11/23)	11.4 (9/23)	**11.0 (8/23)**	4.20
Only Intel Core i7 920 (CPU)								
Dataset	**Total time for multi-core BWC (CPU only) (s)** Different number of CPU threads (No GPU involved)							Speedup (bold) vs. single CPU (underlined)
	1 CPU	**2 CPU**	**3 CPU**	**4 CPU**	**5 CPU**	**7 CPU**	**8 CPU**	
enwik8	<u>15.8</u>	9.1	6.4	5.0	5.0	4.9	**4.9**	3.22
wiki-xml	<u>79.9</u>	44.5	32.3	28.4	25.6	26.2	**26.1**	3.06
linux	<u>32.1</u>	17.6	13.3	10.7	10.0	10.3	**9.9**	3.24
silesia	<u>46.3</u>	30.9	21.5	21.4	21.1	**17.4**	18.4	2.66

Table 5.2: For this table, we use a high-end system with Intel Core i7 CPU and Nvidia GTX 580 GPU. The table shows runtime for all-core BWC (CPU+GPU) and multi-core BWC (CPU only). We use 9MB blocks and 0.1% perturbation, best runtime for each dataset is indicated in bold. We achieve $3.06\times$ speedup with multi-core BWC, which improves to $4.87\times$ with our all-core BWC. Also, if we compare n CPU threads to $n-1$ CPU threads and 1 CPU+GPU thread, the runtimes of latter are better. This again shows our hybrid BWC is faster than CPU BWC.

than $4\times$ for all the datasets compared to the about $3.24\times$ speedup obtained by the multi-core BWC. This shows that our all-core CPU+GPU BWC scales to all the available cores (GPU in addition to the CPU) in the system and provides maximum speedup.

Low-End System. In the second setup, we compare the performance of multi-core BWC and all-core BWC on a combination of Intel Core2Duo E6750 CPU with Nvidia GTX 280 and Nvidia Quadro FX 3700 GPUs (Table 5.3). These are less powerful GPUs with limited support for parallelism as compared to Nvidia GTX 580 and we observe a performance comparable or even worse than CPU for our hybrid BWC. Except for the linux dataset, we see that in both cases the all-core BWC improves upon the best speedup obtained by the multi-core BWC. The improvement is relatively more significant for the GTX 280, as it is a more powerful GPU compared to Quadro FX 3700. The best speedup obtained on the GTX 280 setup is $1.67\times$, on the Quadro FX 3700 we achieve $1.48\times$, while the multi-core BWC gives best speedup of $1.22\times$. Though the Nvidia Quadro FX 3700 is on average two times slower compared to

the dual-core CPU, our pipeline is such that it effectively balances the load and still provides an overall speedup with full resource utilization. Also, to investigate the reasons for slowdown on linux dataset, we ran our code while keeping track of the times at which the items are dequeued from the work queue. It so happens that GTX 280 and some CPU threads dequeue the last remaining blocks around 35.2s. The CPU being faster terminates earlier leaving the GPU to be a bottleneck. There is always a possibility of last block going to the slowest CoSt and affecting the runtime. A possible and an interesting way to avoid this would be to learn the behavior of runtimes (during execution of initial work items) of all CoSts and avoid slower CoSts from dequeuing work items at the very end.

5.5 Discussion

In this chapter, we presented an all-core framework to exploit all computing resources available on a user's system. We demonstrated the practical utility of our all-core framework by implementing the Burrows Wheeler Compression (BWC) pipeline. We demonstrated a speedup on BWC on the GPU for the first time. Our hybrid BWC, that optimally uses both CPU and GPU, achieves good speedup over highly tuned CPU BWC. It also handles large block size efficiently, providing better compression ratio along with better runtime as compared to state-of-the-art. Our all-core framework combines task parallelism and data parallelism effectively. It uses all CPU cores and the GPU cores, while balancing the load between all available resources. Everyday users haven't gained much from enhanced computing power of today's parallel processing platforms. This is an area that needs attention as multi-core CPUs, many-core GPUs, and other accelerators become more prevalent. The ideas of all-core framework and results on an *end-to-end* application we presented can improve the application performance on emerging heterogeneous processing platforms. We expect the all-core framework to be useful for applications like video encoding and decoding, other data compression schemes, etc.

NVIDIA Quadro FX 3700 (GPU) + Intel Core2Duo E6750 (CPU)						
Dataset	**Total time for all-core BWC (CPU+GPU) (s), (#blocks processed by GPU / #total blocks)** Different #CPU threads in addition to 1 CPU+GPU thread					Best Speedup (bold) vs. single CPU (underlined)
	0 CPU	**1 CPU**	**2 CPU**	**3 CPU**	**4 CPU**	
enwik8	35.2 (12/12)	22.5 (7/12)	21.8 (5/12)	**21.1 (3/12)**	21.4 (3/12)	1.25
wiki-xml	201.4 (18/18)	114.5 (8/18)	106.5 (6/18)	100.3 (10/18)	**87.62 (4/18)**	1.48
linux	176.5 (22/22)	65.1 (6/22)	51.4 (4/22)	48.9 (4/22)	**47.1 (2/22)**	1.01
silesia	164.5 (23/23)	66.4 (8/23)	64.6 (6/23)	61.4 (4/23)	**57.2 (5/23)**	1.21
NVIDIA GTX 280 (GPU) + Intel Core2Duo E6750 (CPU)						
Dataset	**Total time for all-core BWC (CPU+GPU) (s), (#blocks processed by GPU / #total blocks)** Different #CPU threads in addition to 1 CPU+GPU thread					Best Speedup (bold) vs. single CPU (underlined)
	0 CPU	**1 CPU**	**2 CPU**	**3 CPU**	**4 CPU**	
enwik8	20.5 (12/12)	19.3 (6/12)	**19.8 (5/12)**	20.1 (4/12)	20.0 (3/12)	1.33
wiki-xml	90 (18/18)	**77.4 (12/18)**	77.6 (10/18)	77.8 (8/18)	85.1 (6/18)	1.67
linux	77.3 (22/22)	46.1 (11/22)	40.3 (7/22)	39.8 (6/22)	**39.2 (5/22)**	1.21
silesia	74.3 (23/23)	57.0 (14/23)	**53.3 (10/32)**	53.6 (9/23)	54.0 (10/23)	1.30
Only Intel Core2Duo E6750 (CPU)						
Dataset	**Total time for multi-core BWC (CPU only) (s)** Different number of CPU threads (No GPU involved)					Best Speedup (bold) vs. single CPU (underlined)
	1 CPU	**2 CPU**	**3 CPU**	**4 CPU**	**5 CPU**	
enwik8	26.4	22.4	**21.9**	22.5	22.3	1.20
wiki-xml	129.9	**106.0**	108.3	109.6	108.1	1.22
linux	47.6	**36.2**	37.5	37.7	37.7	1.31
silesia	69.59	55.26	**53.77**	56.1	56.0	1.29

Table 5.3: For this table, we use a low-end Intel Core2Duo E6750 CPU, Nvidia Quadro FX 3700 (low-end) and Nvidia GTX 280 (medium-end) GPUs. The table shows runtimes for all-core BWC (CPU+GPU) and multi-core BWC (CPU only). We use 9MB blocks and 0.1% perturbation, best runtimes for each dataset are indicated in bold. Using all-core BWC on both these setups, allows us to improve on the speedup achieved by the multi-core BWC (except for linux dataset).

Chapter 6

Conclusions

In this thesis, we developed data parallel algorithms for difficult problems of Floyd Steinberg Dithering (FSD) and String Sorting. FSD had a long sequential dependence that made it difficult to develop parallel algorithms. The non-linear dependence on the outputs of some past pixels, reduced parallelism for FSD greatly. We develop a way to study pixel independence as a function of data dependence and find parallelism to solve the problem efficiently. Thus, we use the data dependence to our benefit and find independent elements for data parallel processing. As shown in Figure 3.13, we can handle various types of data dependency by just analyzing the pixels and finding the correct pixels to process in parallel per iteration. A data dependency analysis and appropriate scheduling scheme, similar to one we developed for FSD, will allow many dynamic programming problems to be solved in parallel.

String sorting is an irregular problem on account of variable work performed per thread (during string comparisons) and arbitrary memory accesses performed depending on the ordering between strings. We map this irregular problem of string sorting to fast standard primitives of sort, scatter and scan on the GPU. Our data parallel algorithm built on standard primitives solves the problem of string sorting efficiently and can adapt well to future architectures. It can easily incorporate any algorithmic improvements to standard primitives for fast performance, without requiring re-design. String sorting or sorting long and variable length keys is a bottleneck in many applications viz. Burrows Wheeler Transform, Data Mining, Data Compression. Our algorithm could be leveraged to accelerate all these applications. In this thesis itself, we extend our string sort algorithm to perform suffix sort step of Burrows Wheeler Transform. Earlier, GPU performed slower than even a single core CPU on the problem of Burrows Wheeler Transform. But, now our string sort allows us to obtain a speed up on GPU for the first time. The data parallel algorithm presented for string sort overcomes the challenges of variable quantity of work performed by threads, dependency between elements of the input and repetitive arbitrary global memory access required (since the basic comparison operation requires data beyond that present in the registers). One or more of our techniques can be adapted to suitably address these challenges in other applications.

We not only develop data parallel algorithms for a few problems but also, we combine them with task parallelism on a hybrid CPU and GPU system for better performance. We examine the effects

of this mix of data parallelism and task parallelism for Floyd-Steinberg Dithering (FSD) and Burrows Wheeler Compression (BIC). For the problem of FSD, we develop two implementations – Handover and Hybrid FSD – that use both CPU and GPU. In these two, we use the idea of work sharing and split the data parallel step across CPU and GPU. For the Handover algorithm, CPU operates when the parallelism is low and GPU operates when parallelism is above some threshold value. This ensures we extract the best performance from both CPU and GPU. The overall runtime improves as compared to using only the data parallel algorithm. In the Hybrid algorithm, in addition to Handover, we further split work between CPU and GPU even when the parallelism is high. We show that only a few bytes of memory transfer are required to keep CPU and GPU synchronized. This hybrid algorithm allows for full resource utilization and gives the best runtime. Our ideas of using CPU and GPU together can and should be used in developing data parallel algorithms for other operations in the future.

We also develop a hybrid and all-core algorithm to solve BWC. Our BWC algorithm performs the expensive sort steps on GPU. We perform the sequential merge, MTF and Huffman encoding computations on CPU and overlap them fully with GPU computations. If there are idle CPU cores, we use them to perform BWC in parallel with our hybrid BWC on single CPU core and GPU. Any end-to-end application like the BWC, will have many tasks. In this thesis, we show that one should efficiently pipeline the execution of these tasks on CPU and GPU for best runtime. Even today, the multiple cores of CPU and the many-core GPU remain largely under utilized by common end-to-end applications. These applications like data compression, file search, image decoding, mpeg encoding/decoding etc. primarily use only the single CPU core or only the many-core GPU. As shown by our results, pipelining of different tasks and also work sharing of data-parallel step between CPU and GPU is critical to achieve high performance on end-to-end applications. The use of our techniques to combine data parallelism and task parallelism, will allow the end-user to enjoy the full benefits of hardware available on his computer system.

Related Publications

1. Hybrid Implementation of Error Diffusion Dithering. Proceedings of IEEE International Conference on High Performance Computing (HiPC'11).

2. Can GPUs Sort Strings Efficiently? Proceedings of IEEE International Conference on High Performance Computing (HiPC'13), *Best GPU Paper Award*.

3. Fast Burrows Wheeler Compression Using CPU and GPU. Submitted to ACM Transactions on Parallel Computing (TOPC) *(Under Review)*.

Bibliography

[1] D. Adjeroh, T. Bell, and A. Mukherjee. *The Burrows-Wheeler Transform:: Data Compression, Suffix Arrays, and Pattern Matching*. Springer, 1 edition, July 2008. 41

[2] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta. Real-time parallel hashing on the gpu. *ACM Trans. Graph.*, 28(5):154:1–154:9, Dec. 2009. 25

[3] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, Oct. 2009. 1

[4] D. A. Bader, V. Agarwal, K. Madduri, and S. Kang. High performance combinatorial algorithm design on the cell broadband engine processor. *Parallel Comput.*, 33(10-11):720–740, Nov. 2007. 1

[5] D. Banerjee and K. Kothapalli. Hybrid algorithms for list ranking and graph connected components. In *High Performance Computing (HiPC), 2011 18th International Conference on*, pages 1–10, Dec 2011. 11

[6] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11, New York, NY, USA, 2009. ACM. 1, 10

[7] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for cuda. *GPU Computing Gems Jade Edition*, page 359, 2011. 30, 31, 32

[8] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, SODA '97, pages 360–369, 1997. 26, 43

[9] I. Buck and T. Purcell. A toolkit for computation on gpus. In *GPU Gems*, 2004. 9

[10] A. Buluc, J. R. Gilbert, and C. Budak. Solving path problems on the gpu. *Parallel Computing*, 36(56):241 – 253, 2010. Parallel Matrix Algorithms and Applications. 10

[11] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994. 41, 42, 43, 53

[12] Bzip2Cuda. Github bzip2-cuda, http://bzip2-cuda.github.io/, 2011. 44

[13] D. Callele, E. Neufeld, and Delathouwer. Sorting on a gpu, http://www.cs.usask.ca/faculty/callele/gpusort/gpusort.html. 9

[14] D. Cederman and P. Tsigas. Gpu-quicksort: A practical quicksort algorithm for graphics processors. *J. Exp. Algorithmics*, 14:4:1.4–4:1.24, Jan. 2010. 1, 9, 27

[15] S. Choudhary, S. Gupta, and P. J. Narayanan. Practical time bundle adjustment for 3d reconstruction on the gpu. In *ECCV Workshops (1)*, pages 423–435, 2010. 10

[16] CUDPP. http://cudpp.github.io/cudpp/2.1/, 2013. Version 2.1. 1

[17] A. Davidson, D. Tarjan, M. Garland, and J. D. Owens. Efficient parallel merge sort for fixed and variable length keys. In *Innovative Parallel Computing*, page 9, may 2012. ix, 25, 27, 33, 36, 38, 44, 45

[18] S. Deorowicz. Universal lossless data compression algorithms. *PhD Thesis, Silesian University of Technology*, 2003. 44, 51

[19] A. Deshpande and P. J. Narayanan. Can gpus sort strings efficiently? In *High Performance Computing (HiPC), 2013 International Conference on*, 2013. 41, 44, 46

[20] J. A. Edwards and U. Vishkin. Brief announcement: Truly parallel burrows-wheeler compression and decompression. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 93–96, New York, NY, USA, 2013. ACM. 44

[21] J. A. Edwards and U. Vishkin. Empirical speedup study of truly parallel data compression. *Technical Report, University of Maryland*, 2013. 44

[22] H. M. Fadhil and M. I. Younis. Article: Parallelizing rsa algorithm on multicore cpu and gpu. *International Journal of Computer Applications*, 87(6):15–22, February 2014. Published by Foundation of Computer Science, New York, USA. 11

[23] R. Floyd and L. Steinberg. An adaptive algorithm for spatial grayscale. *Digest of the Society of Information Display*, 1976. 13

[24] K. Garanzha and C. Loop. Fast ray sorting and breadth-first packet traversal for gpu ray tracing. In *Computer Graphics Forum*, volume 29, pages 289–298. Wiley Online Library, 2010. 25

[25] J. Gilchrist and A. Cuhadar. Parallel lossless data compression using the burrows wheeler transform. *Int. J. Web Grid Serv.*, 4(1):117–135, May 2008. 41, 44, 50

[26] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 325–336, New York, NY, USA, 2006. ACM. 9

[27] A. Gress and G. Zachmann. Gpu-abisort: Optimal parallel sorting on stream architectures. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, page 45, 2006. 1

[28] L. K. Ha, J. Krüger, and C. T. Silva. Fast four-way parallel radix sorting on gpus. *Computer Graphics Forum*, 28:2368–2378, 2009. 1

[29] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag. 1

[30] K. A. Hawick and D. P. Playne. Parallel algorithms for hybrid multi-core cpu-gpu implementations of component labelling in critical phase models, 2013. 11

[31] B. He, N. K. Govindaraju, Q. Luo, and B. Smith. Efficient gather and scatter operations on graphics processors. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007. 25

[32] J. Hoberock and N. Bell. Thrust: A parallel template library, 2010. 1, 9, 25, 27, 51

[33] S. Hocevar and G. Niger. Reinstating floyd-steinberg: Improved metrics for quality assessment of error diffusion algorithms. In *Proceedings of the 3rd International Conference on Image and Signal Processing*, ICISP '08, pages 38–45, Berlin, Heidelberg, 2008. Springer-Verlag. 15

[34] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 78–88, Washington, DC, USA, 2011. IEEE Computer Society. 11

[35] D. Horn. Stream reduction operations for gpgpu applications. In M. Pharr, editor, *GPU Gems 2*, pages 573–589. Addison-Wesley, 2005. 9

[36] Intel. Intel math kernel library, http://software.intel.com/en-us/intel-mkl, 2013. 1

[37] J. Kärkkäinen and T. Rantala. Engineering radix sort for strings. In *Proceedings of the 15th International Symposium on String Processing and Information Retrieval*, SPIRE '08, pages 3–14, 2009. xii, 26, 27, 32, 33, 38

[38] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proceedings of the 30th International Conference on Automata, Languages and Programming*, ICALP'03, 2003. 44, 46

[39] D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Combinatorial Pattern Matching*, pages 186–199, 2003. 46

[40] P. Kipfer, M. Segal, and R. Westermann. Uberflow: A gpu-based particle engine, 2004. 9

[41] P. Kipfer and R. Westermann. Improved GPU sorting. In M. Pharr, editor, *GPUGems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 733–746. Addison-Wesley, 2005. 9

[42] K. Kothapalli, D. S. Banerjee, P. J. Narayanan, S. Sood, A. K. Bahl, S. Sharma, S. Lad, K. K. Singh, K. K. Matam, S. Bharadwaj, R. Nigam, P. Sakurikar, A. Deshpande, I. Misra, S. Choudhary, and S. Gupta. Cpu and/or gpu: Revisiting the gpu vs. cpu myth. *CoRR*, 2013. 11

[43] N. Leischner, V. Osipov, and P. Sanders. Gpu sample sort. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10, 2010. 9, 27

[44] A. Leist, K. A. Hawick, and D. P. Playne. Gpgpu and multi-core architectures for computing clustering coefficients of irregular graphs, 2012. 11

[45] P. Li and J. P. Allebach. Block interlaced pinwheel error diffusion. *Journal of Electronic Imaging*, 14(2):023007–023007–13, 2005. 1, 15

[46] C.-C. Lin and W.-H. Tsai. Visual cryptography for gray-level images by dithering techniques. *Pattern Recogn. Lett.*, 24(1-3):349–358, Jan. 2003. 13

[47] LinuxKernel. http://www.kernel.org/pub/linux/kernel/v2.6/, 2005. 44, 51

[48] M. Mahoney. Enwik8, http://mattmahoney.net/dc/text.html, 2006. 44, 51

[49] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. 20, 21

[50] K. K. Matam, S. R. K. B. Indarapu, and K. Kothapalli. Sparse matrix-matrix multiplication on modern architectures. In *HiPC*, pages 1–10, 2012. 10

[51] K. K. Matam and K. Kothapalli. Accelerating sparse matrix vector multiplication in iterative methods using gpu. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 612–621, Washington, DC, USA, 2011. IEEE Computer Society. 10

[52] P. M. McIlroy, K. Bostic, and M. D. McIlroy. Engineering radix sort. *COMPUTING SYSTEMS*, 6:5–27, 1993. 26

[53] W. mei Hwu. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011. 1

[54] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. *SIGPLAN Not.*, 47(8):117–128, Feb. 2012. 25

[55] D. Merrill and A. Grimshaw. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, 21:245–272, 2011. 3

[56] D. Merrill and A. Grimshaw. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, 21(02):245–272, 2011. 9, 25, 27, 34, 36

[57] P. T. Metaxas. Optimal parallel error-diffusion dithering, 1999. 13, 14

[58] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, Mar. 2008. 8

[59] Nvidia. Nvidia performance primitives, https://developer.nvidia.com/npp. 1

[60] Nvidia. Nvidia cuda best practices guide, 2011. 20

[61] R. A. Patel. Parallel lossless data compression on the gpu. Master's thesis, Electrical and Computer Engineering, University of California, Davis, June 2012. 51

[62] R. A. Patel, Y. Zhang, J. Mak, and J. D. Owens. Parallel lossless data compression on the GPU. In *Proceedings of Innovative Parallel Computing (InPar '12)*, May 2012. 5, 41, 44, 45, 47, 51, 54

[63] S. Patidar and P. J. Narayanan. Scalable split and gather primitives for the gpu. Technical report, CVIT, IIIT Hyderabad, 2009. 9

[64] T. Purcell. Ray tracing on stream processors. *PhD Thesis, Stanford University*, 2004. 9

[65] I. Reguly and M. Giles. Efficient sparse matrix-vector multiplication on cache-based gpus. In *Innovative Parallel Computing (InPar), 2012*, pages 1–12, May 2012. 10

[66] M. S. Rehman, K. Kothapalli, and P. J. Narayanan. Fast and scalable list ranking on the gpu. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 235–243, New York, NY, USA, 2009. ACM. 10

[67] G. Ruetsch and P. Micikevicius. Optimizing matrix transpose in cuda. *NVIDIA CUDA SDK Application Note*, 2009. 18

[68] K. Sadakane. A fast algorithm for making suffix arrays and for burrows-wheeler transformation. In *In Proceedings Of The IEEE Data Compression Conference, Snowbird, Utah, March 30 - April 1*, pages 129–138. IEEE Computer Society Press, 1998. 44

[69] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10, 2009. 9, 25, 27

[70] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for gpu computing. In *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. 1, 9

[71] J. Seward. On the performance of bwt sorting algorithms. In *Data Compression Conference*, pages 173–182. IEEE Computer Society, 2000. 41, 42, 44, 46, 50, 51, 53

[72] R. Sinha and A. Wirth. Engineering burstsort: Towards fast in-place string sorting. In C. McGeoch, editor, *Experimental Algorithms*, volume 5038 of *Lecture Notes in Computer Science*, pages 14–27. 2008. 26, 38

[73] R. Sinha, J. Zobel, and D. Ring. Cache-efficient string sorting using copying. *J. Exp. Algorithmics*, 11, Feb. 2007. xii, 26, 32, 33, 38

[74] P. Slavik and J. Prikryl. Dithering as a method for image data compression. *Winter School of Computer Graphics*, 1995. 13

[75] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parallel Comput.*, 36(5-6):232–240, June 2010. 10

[76] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the gpu. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 167–171, New York, NY, USA, 2009. ACM. 25

[77] B. Walter, G. Drettakis, and S. Parker. Interactive rendering using the render cache. In D. Lischinski and G. W. Larson, editors, *Rendering Techniques*, pages 19–30. Springer, 1999. 13

[78] Z. Wei and J. Jaja. Optimization of linked list prefix computations on multithreaded gpus using cuda. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–8, April 2010. 10

[79] Wikipedia. http://dumps.wikimedia.org/enwiki/latest/, 2014. 51

[80] S. Xiao, A. Aji, and W. chun Feng. On the robust mapping of dynamic programming onto a graphics processing unit. In *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, pages 26–33, Dec 2009. 3

[81] Y. Zhang, J. Recker, R. Ulichney, G. Beretta, I. Tastl, I.-J. Lin, and J. D. Owens. A parallel error diffusion implementation on a gpu. In *Proceedings of SPIE, Parallel Processing for Imaging Applications*, volume 7872, Jan. 2011. 15

[82] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. In *ACM SIGGRAPH Asia 2008 papers*, SIGGRAPH Asia '08, pages 126:1–126:11, 2008. 1