Efficient Algorithmic Techniques for Heterogeneous Computing

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

MS
in
Computer Science Engineering

by

Shashank Sharma
200802034
shashank.sharma@research.iiit.ac.in

International Institute of Information Technology
Hyderabad - 500 032, INDIA
January 2017

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "Efficient Algorithmic Techniques for Heterogeneous Computing" by Shashank Sharma, has been carried out under my supervision and is not submitted elsewhere for a degree.

Date

Adviser: Prof. Dr. Kishore Kothapalli

Dedicated to my parents.

Acknowledgments

This thesis is a result of inputs and contribution from various people. I owe my deepest gratitude to my guide, Dr. Kishore Kothapalli, for his guidance and suggestions throughout the duration of this work. He has supported me throughout my thesis with his patience and knowledge whilst allowing me the room to work in my own way. He helped to shape the direction of this work, filled in many of the gaps in my knowledge, and helped me towards solutions. His enthusiasm, constant encouragement and availability to discuss ideas have made working with him a true pleasure. I would also thank my mentor Dip Sankar Banerjee for his encouragement and positive criticism at times when I felt frustrated and moving ahead looked difficult. Our discussions in and outside the lab has helped shaped this thesis the way it is.

Abstract

Today, general-purpose GPUs(GPGPUs) have become very ubiquitous and available in something as simple as a personal laptop. These GPUs can be utilized to a great extent for their huge parallelism such as NVIDIA GT520 having 48 cores and GTX580 having 512 cores that we use for our implementations. On such devices Single Instruction Multiple Data(SIMD) operations can be performed using NVIDIA CUDA programming model. GPUs for general purpose computing has brought forth several success stories with respect to time taken, cost, power, and other metrics. However, accelerator based computing has significantly relegated the role of CPUs in computation. As CPUs evolve and also offer matching computational resources, it is important to also include CPUs in the computation. We call this the hybrid computing model. Indeed, most computer systems of the present age offer a degree of heterogeneity and therefore such a model is quite natural.

With such a hybrid model in mind, in this thesis, we try to explore ways to harvest the benefits of hybrid computing. In this thesis we explore two such ways namely Dynamic Load Balancing where we split the work among GPU and CPU dynamically and the other technique is graph pruning to recursively remove 1-degree nodes before applying an algorithm on a much smaller resultant graph and later post-processing the results to the complete graph.

The first two chapters propose hybrid solution to two of major application in GPU computing such as Lattice Boltzmann Method(LBM), a class of computational fluid dynamic problem and Ray Casting, an image processing primitive. Later we propose a light-weight, low overhead, and completely dynamic framework that addresses the load balancing problem of heterogeneous algorithms. Our framework will be applicable for workloads which have a few simple characteristics such as having a collection of largely independent tasks that are easily describable. To show the efficacy of our framework, we consider two different heterogeneous computing platforms, and two of the above mentioned workloads: LBM and ray casting. For each of these workloads, we demonstrate that using our framework, we can identify the proportion of work to be allotted to each device up to $\pm 8\%$ on average with reference to the base hybrid solution. Further, solutions using our framework require no more than 5% additional time on average compared to best possible load assignment obtained via empirical search.

In the later part of the thesis we introduce graph pruning as a technique that aims to reduce the size of the graph. Certain elements of the graph can be pruned depending on the nature of the computation. Once a solution is obtained on the pruned graph, the solution is extended to the entire graph.

We apply the above technique on a very fundamental graph algorithm: Single Pair Shortest Path that we later generalize to solve All Pairs Shortest Paths (APSP). To validate our technique, we implement our algorithms on a heterogeneous platform consisting of a multicore CPU and a GPU. On this platform, we achieve an average of 35% improvement compared to state-of-the-art solutions. Such an improvement has the potential to speed up other applications reliant on these algorithms.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Accelerator Based Computing

The whole world of parallel computing underwent a change when graphics practitioners found that GPUs can be used for many general purpose computations. The term of General Purpose Computing on Graphics Processing Units (GPGPU) was coined during that time. Although NVIDIA led the initial development of the GPGPU program, soon significant contributions were made by industry houses and scientists from several fields of research. Today, OpenCL is the dominant open source general purpose computing language for accelerators, although the dominant proprietary framework is still NVIDIAs Compute Unified Device Architecture (CUDA).

In parallel to the GPUs, the Cell BE initiative from IBM also made a big impact in this new world. The POWER4 processor which was also designed fundamentally for aiding graphics and games, was re-designed to perform general purpose computations and renamed as the Cell. It was heavily used in the Sony PlayStation gaming consoles and also contributed towards many teraflop supercomputers such as the IBM Roadrunner.

Parallel computing using accelerator based platforms has gained widespread research attention in recent years. Accelerator based general purpose computing, however, relegated the role of CPUs to second-class citizens where a CPU sends data and program to an accelerator and gets the results of the computation from the accelerator. As CPUs evolve and narrow the performance gap with accelerators on several challenge problems, it is imperative that parity is restored by also bringing CPUs in the computation process.

1.2 Hybrid Computing

Hybrid computing seeks to simultaneously use all the computational resources on a given tightly coupled platform. We envisage a multicore CPU plus accelerators such as GPUs as one such realization as shown in Figure 1.1. The CPU on the left with six cores is connected to a many-core GPU on the right. In our work, we use two variants of the model shown in Figure 1.1 by choosing different CPUs

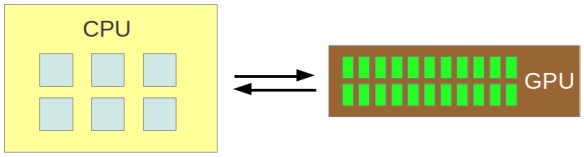


Figure 1.1 A tightly coupled hybrid platform.

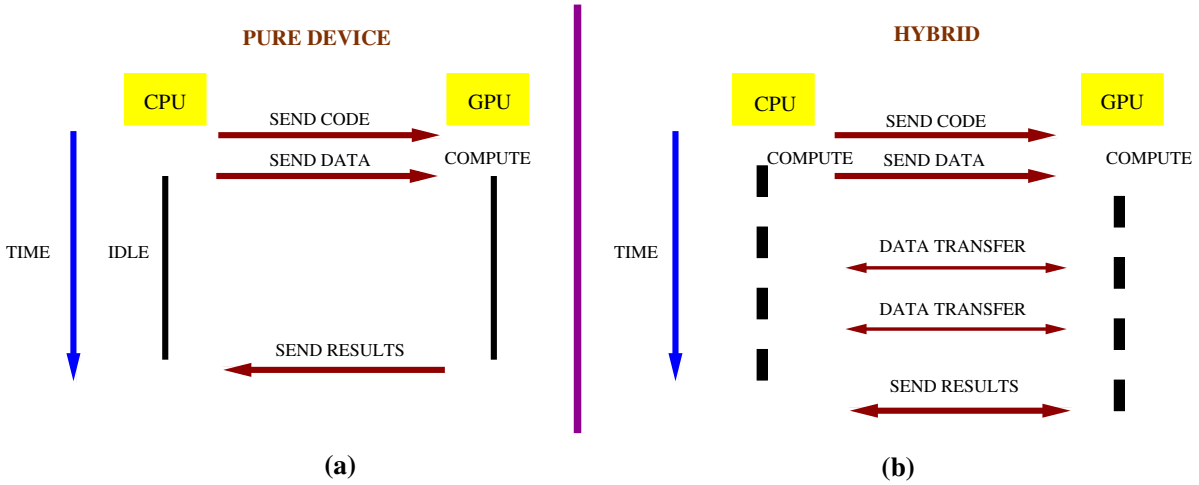


Figure 1.2 A view of hybrid multicore computing. Figure (a) shows the conventional accelerator based computing where the CPU typically stays idle. Figure (b) shows the hybrid computing model with computation overlapping between the CPU and the GPU.

and GPUs. Figure 1.2(b) illustrates the use of a hybrid computing model where both the devices are used optimally.

The case for hybrid computing on such a platform can be made naturally. Computers come with a CPU, at least a dual-core at present, and is expected to contain tens of cores in the near future. Graphics processing units are traditionally used to process graphics operations and most computers come equipped with a graphics card that presently has several GFLOPS of computing power. Moreover, commodity production of GPUs has significantly lowered their prices. Hence, an application using both a multicore CPU and an accelerator such as a GPU can benefit from faster processing speed, better power and resource utilization.

Further, it is believed that GPUs are not well suited for computations that offer little SIMD parallelism, and have highly irregular memory access patterns. For various reasons, CPUs do not suffer greatly on such computations. Thus, hybrid computing opens the possibility of novel solution designs that take heterogeneity into account. Hence, we posit that hybrid computing has the scope to bring the benefits of high performance computing to also desktop and commodity users.

We distinguish between our model of hybrid computing with other existing models as follows. We consider computational resources that are tightly coupled. Supercomputers such as the Tianhe-2 that use a combination of CPUs and GPUs do not fall in our category. The issues at that scale would overlap with

some of the issues that arise in hybrid computing, but have other unique issues such as interconnection network and its cross-section bandwidth, latency, and the like.

Hybrid computing solutions on platforms using a combination of CPUs and GPUs are being studied for various problem classes such as dense linear algebra kernels [5, 43], maximum flows in networks [21], list ranking [47] and the like. Most of these works however have a few limitations. In some cases, for example [21, 47, 40] computation is done only either on the CPU or the GPU at any given time. Such a scenario keeps one of the computing resource idle and hence is in general wasteful in resources. Secondly, we explore a diverse set of workloads so as to highlight the advantages and limitations of hybrid multicore computing. In a departure from most earlier works, we study hybrid computing also on low end CPU and GPU models and see the viability of hybrid computing on such platforms.

Accelerator based computing using many-core architectures such as the GPUs and the IBM Cell BE has been successful in pushing application performance on commodity systems (cf. [37, 11, 34]). Further, GPUs have now become ubiquitous even on commodity systems due to their low price and low power consumption. However, it is projected that improvements in multicore CPU technology will further drive the performance of CPUs so as to scale challenges such as the power wall, the memory wall, and the like. Further, it is strongly believed that future generation computer systems shall be heterogeneous in nature with a mix of multicore CPUs and special purpose accelerators. Hence, it becomes important to design and implement efficient algorithms that work seamlessly on heterogeneous systems. This is termed as heterogeneous computing or hybrid computing in various recent works (cf. [7, 31, 42, 18, 6]). Heterogeneous computing has the scope to offer improved resource usage apart from faster algorithms.

However, heterogeneous algorithm design and implementation is fraught with a host of challenges. Current heterogeneous architectures impose severe limits on bandwidth available for communicating across the devices. This poses serious concerns to algorithm design and implementation. Synchronization between devices poses yet another difficulty. Therefore, at present, designing and implementing heterogeneous algorithms comes with a lot of hand-crafting. Nevertheless, heterogeneous computing on commodity platforms is gaining large scale research attention in recent years. Such algorithms have been designed recently for several challenging problems in parallel computing including graph BFS [13, 31, 22], dense matrix computations [51], sorting [8], and the like. Many of the above-cited works spread the entire computation across the computational devices. In some cases, this is followed by a post-processing phase that combines the outputs of the individual computations [7, 31, 8, 38, 42]. This approach of designing heterogeneous algorithms can be called as the work partitioning approach.

However, due to differing architectures and execution characteristics, the amount of work each device can do in a given computation can vary significantly across the devices. Therefore, using the work partitioning approach in the design and implementation of heterogeneous algorithms brings up the problem of load balance across devices. As heterogeneous algorithms should aim to improve the resource usage, it is required that proper load balance mechanisms are deployed.

Current approaches to address this problem include either a static work partitioning or using an analytical model to determine the appropriate work partitions. In the former case, one typically uses a standard dataset for a given workload and arrives the proportion of work allocation. This approach, used in [31, 7, 42, 18] to name a few, is very limiting as it fails to capture any properties of a different instance before deciding the work partition. In the latter case, one proposes an analytical model that captures the time taken by each device on a given input. This model can then be used to identify the work proportion. This approach is used in [31, Section IV.C]. However, the latter approach is not generic and is known to work only for very specific problems or specific input instances of a problem [31]. Thus, existing approaches fall short of a complete solution.

Thus, a fundamental problem in heterogeneous computing is to propose generic mechanisms that can help address the issue of load balancing in heterogeneous algorithms designed using the work partitioning model [16]. In this thesis, we propose a simple and light-weight mechanism for the same. Our mechanism has the special property that it can offer load balancing even under dynamic conditions, and is suitable for a variety of application workloads which exhibit some common characteristics. We also validate our proposal against two different workloads: Lattice Boltzmann Method (LBM), and Ray Casting (RC). These workloads are chosen for their wide-ranging applications and importance. These workloads have been part of several studies on benchmarks and are also included in the recent highly-influential work of Lee et al. [29]. We note that our approach can be applied to design heterogeneous algorithms for other workloads also. Our work thus indicates that efficient dynamic load balancing approaches can be designed with relatively little additional effort. While we focus on CPU + GPU heterogeneous computing platforms in this thesis, our framework can be easily adapted to other heterogeneous computing platforms.

In this case we can easily argue the case of many-core vs multi-core computing, where a many-core is a combination of many low compute cores bundled in a chip and a multi-core is a chip where a small number of high-power cores are packed. There are several scientific computations which have high compute requirement, but are aided by graphical simulations which require low compute but a higher number of working threads. Hence, we can intuitively make an assumption regarding the presence of many large scale applications, that are not entirely suitable for accelerator type computations. There has to be some kind of a trade-off that must exist in order to cater to all types of applications. In particular we may look at some cases where there has been sufficient proof to show that some operations like the atomic operations are not very well suited for the accelerators. Computations such as histograms which are not time consuming ones in the CPU often suffer in GPUs. In paper [36], the authors have shown an implementation of histogram on GPUs where they have dealt with the complications of efficiently computing histograms. The authors have showed the complexity that is involved in the accumulation stage of all the bins and he limited precision which is available. The CPUs, however have proved time and again their efficiency in performing operations such as atomics, gather and scatter. It is the development of the dedicated hardware units for these operations that enable the CPU to excel in their implementations.

1.3 Platform

GPUs						
Device	**Cores**	**# of SMs**	**Clock**	**Global Memory**	**L2 Cache**	**Threads**
GTX580	512	16	1.54 GHz	1535 MB	768 KB	1024
GT520	48	1	1.62 GHz	1024 MB	64 KB	1024
CPUs						
Device	**# of Cores**	**SIMD width**	**Clock**	**Last Level Cache**	**L1 Cache**	**Threads**
i7 980x	6	4	3.4 GHz	12 MB	32 KB	12
Core 2 Duo	2	2	2.8 GHz	3 MB	32KB	4

Table 1.1 The specifications for the different GPUs and CPUs used in our experiments.

1.3.1 Hetero-Low

Our low-end heterogeneous platform resembles a commodity desktop computing environment more closely. The Hetero-Low platform is a combination of an Intel Core 2 Duo E7400 CPU along with an NVIDIA GT520 GPU. The CPU and the GPU are connected via a PCI Express version 2.0 link. This link supports a data transfer bandwidth of 8 GB/s between the CPU and the GPU. To program the GPU we use the CUDA API Version 4.1, and we use OpenMP specification 3.0 to program the CPU along with ANSI C.

The GT520 is a stand-alone graphics processor having 48 computing cores and 1 GB of global memory. Each of the compute cores are clocked at 810 MHz. The GPU on an average give a sustained performance of 77.7 GFLOPS and consume about 29W of power. In this system both the processors are of a comparable performance range and hence provide a more realistic platform for experimenting the heterogeneous programs.

The Intel Core 2 Duo CPU is one of the earliest multicore offerings from Intel and was released in the year 2008. It has 2 cores with hyper-threading and each core is clocked at 2.8 GHz. The CPU consists of a 3 MB L2 cache and the maximum power consumption is around 65 W. The CPU was designed entirely for commodity PCs and gives a sustained performance of about 20 GFLOPS.

1.3.2 Hetero-High

The Hetero-High heterogeneous platform we used is a coupling of the two devices, the Intel i7 980 and the Nvidia GTX 580 GPU. The CPU and the GPU are connected via a PCI Express version 2.0 link. This link supports a data transfer bandwidth of 8 GB/s between the CPU and the GPU. To program the GPU we use the CUDA API Version 4.1. The CUDA API Version 4.1 supports asynchronous concurrent execution model so that a GPU kernel call does not block the CPU thread that issued this call. This also means that execution of CPU threads can overlap a GPU kernel execution.

The GTX 580 GPU is a current generation Fermi micro-architecture from NVIDIA that has 16 symmetric multi-processors (SM) with each SM having 32 cores for a total of 512 compute cores. Each compute core is clocked at 1.54 GHz. Each SM has a hardware scheduler that schedules 32 threads at a time. This group is called a *warp* and a half-warp is a group of 16 threads that execute in a SIMD fashion. Each of the cores of the GPU now has a fully cached memory access via an L2 cache, 768 KB in size. In all, the GTX 580 has a peak single precision performance of 1.5 TFLOPS.

Along with the GTX 580, we use an Intel i7 980x processor as the host device. The 980x is based on the Intel Westmere micro-architecture. This processor has six cores with each core running at 3.4 GHz and with a thermal design power of 130 W. With active SMT(hyper-threading), the six cores can handle twelve logical threads. The L3 cache has a size of 12 MB. The L1 cache size is 64 KB per core and the size of the L2 cache is 256 KB. Other features of the Core i7 980x include a 32 KB instruction and a 32 KB data L1 cache per core and the L3 cache is shared by all 6 cores. The i7 980 CPU has a peak throughput of 110 GFlops.

1.4 Results

For this thesis we study three applications on GPU as well as CPU + GPU hybrid platforms as mentioned in Section 1.3.

For Lattice Boltzmann Method we implement pure GPU as well as two different approach to hybrid implementation as given in Chapter 3 namely static work partition and the work queue model as described in Section 3.4.2. In our Hybrid algorithms for Lattice Boltzmann Method we achieve around 20% improvement on Hetero-low and around 10% in high end both over the pure GPU implementation. We later compare our hybrid implementation on static partition based on empirical data and dynamic work split model. We note a performance difference of less than 2% in both hetero-high and hetero low platforms thus laying a possibility of generalising this method.

We see similar improvements for Ray Casting, on hetero-low we get 30% while on hetero-high its around 5% over pure GPU implementation. This comes from the fact that our CPU is being utilized to the most optimum work division, that we achieve by trial and error. We compare the baseline implementation based on empirically determined work split with the dynamic worksplit model and note a performance difference of 5% on hetero-high and around 10% on hitero-low thus indicating applicability and efficacy of our mode. Hetero Low has relatively close CPU-GPU performance difference. Which is relatively higher in hetero-high as we see later in Chapter 3 and 4. In Chapter 5 we implement a pre-processing step in solving APSP by recursively removing one degree nodes before applying the actual algorithm. Our method shows approximately 44% improvement on average over some real world graphs taken from [1] over pure GPU implementation without pre-procesing. We show analysis over varying the number of iterations of leaf pruning and percentage of nodes removed.

Chapter 2

Related Work

There has been considerable interest in GPU computing in the last decade. Some of the notable early works include scan [37], spmv [9], sorting [32], and the like. Other modern architectures that have been studied recently include the IBM Cell and the multi-core machines. In this these we focus on hybrid algorithms where we use a CPU and GPU.

One of the recent work of Lee et al. [29] claims that GPU computing can offer on average only 3x performance advantage over a multicore CPU on a range of 14 workloads deemed important for throughput oriented applications. Lattice Boltzmann Method and Ray Casting being two of the 14 workloads, we study their hybrid implementations as when both CPU and GPU join hands we can get much better performance in comparison to CPU or GPU alone.

Hybrid computing is gaining popularity across application areas such as dense linear algebra kernels [5, 43, 3], maximum flows [21], graph BFS [22] and the like. The aim of this thesis is to however establish the potential of hybrid computing by dividing the work loads between the otherwise idle CPU and the GPU. Further, in some of these works, (cf. [22, 21, 48]), while both the CPU and the GPU are used in the computation, one of the devices is idle and while the other is performing computation. In contrast, we seek solutions where both the devices are simultaneously involved in the computation.

One of the works on dynamic task management was proposed by Song et al. in [38]. In this paper, the authors describe a dynamic task scheduling system for distributed memory systems which does not have any dependence on process cooperation. They design a dynamic runtime system for linear algebra libraries like PBLAS and ScaLAPACK and uses algorithms to resolve data dependencies without process cooperation. They test their solution on three well known linear algebra routines such as the Cholesky Factorization, LU Factorization and QR Factorization.

Lattice Boltzmann Equation was well studied for implementation by [12] for different types of fluid. We see a detailed implementation for D3Q13 lattice by Tolke et al. [41] on a desktop level GPU NVIDIA GeForce 8800. In this they achieve a teraflop performance using plain GPU implementation. This was subsequently improved by Habich et al. [19] for a D3Q19 implementation that we try to solve in our work. Again the work here is purely on GPU while the authors mention that we can achieve further optimizations using CPU + GPU hybrid implementation.

The implementation in Ray Casting is compared to baseline implementation of Lurig et al. [30] where they identify the best work split empirically, we try to implement our dynamic work split model and compare the performance.

Many recent works in parallel computing have focused on graph algorithms. Few among them include [14, 23, 48, 31]. One of the first results of BFS,APSP using GPUs is the work of Harish et al. [20]. Subsequent improvements to [20] centered around the use of heterogeneous computing such as a recent work [18] partitions the graph so that low degree vertices are processed on the GPU and the high degree vertices are processed on the CPU.

The all-pairs-shortest-paths (APSP) problem is a fundamental graph algorithm with several applications. We see a GPU implementation of the same in Harish et al. [20] where he tried to solve a parallel version of Dijkstra's algorithm. We compare the results of our implementation with Katz et al. []

In [33], the authors try to solve the maximum clique problem on large sparse graphs using pruning techniques. Their main idea is to prune the vertices that strictly have fewer neighbors than the size of the maximum clique already computed. These vertex can be exempted from computation as even if a new clique is found, its size would not be greater than the maximum one that is already computed. We use similar technique in APSP problem and compare results by varying number of pruning steps.

Chapter 3

Lattice Boltzmann Method

3.1 Introduction

Lattice Boltzmann Method, LBM for short, is a class of computational fluid dynamics for fluid simulation. It is a numerical method to solve the Nevier-Stokes equation for incompressible Newtonian fluids [12, 39]. This discrete Boltzmann equation is solved to simulate the flow of Newtonian fluid by using stream and collision functions. LBM has importance in various scientific simulation applications such as advection, diffusion, flow simulation, heat transfer and calculating permeability. Any kind of incompressible material that flows in any viscous medium can be studied using Lattice Boltzmann Method.

We here present a hybrid solution that give improvements over plain GPU approach as shown by Kothapalli et al. [28]. There are many works on improving and optimizing LBM on GPUs. But this is the first hybrid solution. We first studied the static partitioning of work loads [28] and later a dynamic architecture aware solution which dynamically determines the threshold as it runs. The static partitioning was also studied in two forms where first we tried to split workload on the basis of 19 velocity functions but speedups were limited because of less precision in workload split and dependencies as we will study in Section 3.4. The later approach is a particle based threshold split which we further use for dynamic threshold approach.

3.2 Lattice Boltzmann Equation

LBM is applied on a limited number of particles which simulates their streaming and collision in each iteration. The intrinsic particle interaction throughout the viscous medium leads to a good study of flow behavior applicable across the greater mass. In our experiment we study two different models on Lattice Boltzmann Method i.e. Bhatnagar-Gross-Krook (BGK) [10] and Multi-Relaxation Time(MRT). MRT scheme provides more accurate simulation compared to the less computation intensive BGK scheme.

In our work we study the D3 Q19 Lattice Boltzmann Equation for fluid simulation where we have three Dimensional fluid lattice and each particle has interactions with 19 particles around it noted by the 19 directional velocity vectors as shown in Figure 3.1. The D3 Q19 velocities are given by the vector

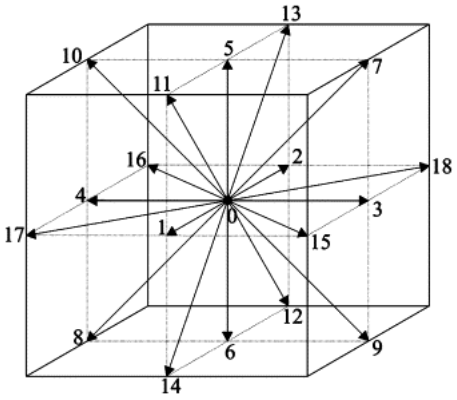


Figure 3.1 A three dimensional lattice with 19 speed vectors. This model is called D3 Q19. This figure is taken from [44].

$$\xi_i = \begin{cases} \{0,0,0\}^T & \text{for } i = 0 \\ \{\pm 1,0,0\}^T, \{0,\pm 1,0\}^T, \{0,0,\pm 1\}^T & \text{for } i = 1,2,\cdots,6 \\ \{\pm 1,\pm 1,0\}^T, \{\pm 1,0,\pm 1\}^T, \{0,\pm 1,\pm 1\}^T & \text{for } i = 7,8,\cdots,18 \end{cases} \tag{3.1}$$

The velocity and density can be obtained from distributions as

$$\rho = \sum_{i=0}^{Q-1} f_i \tag{3.2}$$

$$\mathbf{u} = \frac{1}{\rho} \tag{3.3}$$

Lattice Boltzmann equations are given by a two step procedure, collision and streaming.

Collision is done at time $t + \delta t$ as

$$f_i^t(\vec{x}, t + \delta t) = f_i(\vec{x}, t) + \jmath_i \text{ for } i = 0, 1, \cdots, 18. \tag{3.4}$$

Streaming at time $t + \delta t$ and position $x + \delta x$ is given by

$$f_i(\vec{x} + \delta x, t + \delta t) = f_i^t(\vec{x}, t + \delta t) \text{ for } i = 0, 1, \cdots, 18. \tag{3.5}$$

Here $\jmath_i$ is the collision operator which is different for BGK and MRT.

The term f_i represent the i^{th} speed vector out of 19 directional vectors, and f_{eq} is the equilibrium values which approximate Maxwellian distributions. These are calculated as follows:

$$f_i^{eq} = w_i \cdot \rho \cdot \left(1 + 3(\xi_\mathbf{i}.\mathbf{u}) + \frac{\mathbf{9}}{\mathbf{2}}(\xi_\mathbf{i}.\mathbf{u}) - \frac{\mathbf{3}}{\mathbf{2}}(\mathbf{u}.\mathbf{u})\right) \tag{3.6}$$

where ρ is the density as shown above.
Further, we have $w_0 = \frac{1}{3}$, $w_1, \cdots w_6 = \frac{1}{18}$, and $w_7, \cdots, w_{18} = \frac{1}{36}$, and u is the velocity.

3.2.1 Bhatnagar-Gross-Krook (BGK)

The BGK model assumes that the distributions relax at a constant rate toward equilibrium value f_i^{eq}.
Collision term for BGK

$$\jmath_i = \frac{1}{\tau}(f_i^{eq} - f_i) \tag{3.7}$$

here τ is the relaxation rate which relates to fluid viscosity (η) as,

$$\eta = \frac{1}{3}(\tau - \frac{1}{2}) \tag{3.8}$$

3.2.2 Multi-Relaxation Time

The BGK term approximates that all non-conserved hydrodynamic modes relax at a constant rate, which might not always be the case therefore Multi-Relaxation Time scheme takes into consideration Hydrodynamic modes that relax at different rates. Hydrodynamic modes are given by

$$\hat{f}_i = \sum_{q=0}^{18} \alpha_{i,q} f_q \tag{3.9}$$

where α is a constant for each particle mode f_q
Collision term for MRT

$$\jmath_i = \sum_{i=0}^{18} \alpha^*_{q,i} \lambda_i (f_i^{eq} - \hat{f}_i) \tag{3.10}$$

The $\alpha^*_{q,i}$ are the matrix inverse of transformation coefficients $\alpha_{i,q}$ and they map the moments back to distribution space. λ_i are the relaxation parameters chosen to minimize viscosity dependence of fluid flow. given by

$$\lambda_1 = \lambda_2 = \lambda_9 = \lambda_{10} = \lambda_{11} = \lambda_{12} = \lambda_{13} = \lambda_{14} = \lambda_{15} = \lambda_A = \frac{1}{\tau}$$

$$\lambda_4 = \lambda_6 = \lambda_8 = \lambda_{16} = \lambda_{17} = \lambda_{18} = \lambda_B = \frac{8(2 - \lambda_A)}{8 - \lambda_A}$$

3.3 Experimental Setup

Our experimental setup consists of around 1 million points distributed uniformly across a 3D space of N_x, N_y, N_z. N_y and N_z are fixed and N_x is varied to increase the volume of fluid particles. These points are released in horizontal direction to hit a solid sphere. The flow is then simulated for a large number of iterations till a lot of points pass over the sphere. Then an average of 100 iterations is taken to compare the results. We run the experiment for 500,000 points to 1,500,000 points that are initialized in a 3D cuboidal space.

3.4 Implementation

We need two arrays, one to store the coordinates of points and other to store the 19 velocity vectors $(f_0, f_1, \cdots f_18)$ for each points as given implemented by [19, 41]. The coordinates are mapped to the graphics unit as a Vertex Buffer Object in OpenGL. Where as the velocity array is initialised such as all the N points for f_i are stored consecutively as shown in Figure 3.2. Here as the array is 1 dimensional, index in of Point $P_{i,j,k}$ is calculated as $(i \times N_y \times N_z) + (j \times N_z) + k$ which is further stored for 19 directions as $d_i \times totalpoints + in$ thus all the threads accessing a memory at any given time finds the data stored consecutively this increases efficiency as shown in [49] and Figure 3.2.

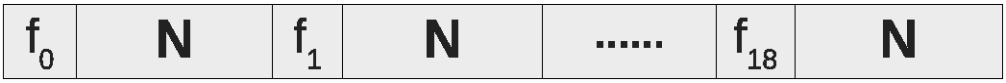


Figure 3.2 Representation of Array

We study and optimize LBM on CPU, GPU individually and then develop a hybrid solution.

In the LBM method, the collision and streaming steps are done for each particle in parallel. Further, the computations are independent across the particles and also on the 19 velocity vectors. We can either divide the workload on particles on the basis of X- direction or we can divide on $f_0 to f_4 on CPU$ and $f_5 to f_1 8$ on GPU.

In the first case where work is divided on the basis of 19 velocities, The dependency is such that f_0 to f_4 should be together. This puts a performance limit in hybrid implementation as the work is split in ratio 5:14 and if the architecture is such that the CPU is any slower than this ratio our performance degrades.

The second method is to divide the work on the basis of points on one of the axis. Here we divide points on X-Axis. This method helps in much precise work split but the computation in the streaming step of particle p_x, p_y, p_z depends on particle p_{x+1}, p_y, p_z which can be solved by asynchronous transfer of p_{x+1} and p_{x-1} values to the CPU and the GPU respectively before the CPU, GPU reaches their respective work split boundaries. Here the boundaries are determined using our work split model.

The work load is split between CPU and GPU as per our model. And within each CPU and GPU the points are divided on the X-Axis, a work unit thus is succinctly described as those corresponding to a set of contiguously numbered particles. The computation in an iteration numbered $i + 1$ uses the values generated in iteration i, for $i = 1, 2, \cdots$. Each iteration indicates 1 time step and executes the 3 functions namely, collision, checking_boundaries, and stream. After that the coordinates are computed and mapped to the Vertex Buffer Object (VBO) and is plotted on a OpenGL window. The independent nature of the work units makes this workload suitable for the work queue model.

For a given number of particles, we first apply the collision function which calculates $f(x, t + \delta t)$ and then apply the stream function which calculates $f(x + \delta x, t + \delta t)$. The boundary collisions are treated as elastic collisions. We used two different LBM models BGK (Bhatnagar-Gross-Krook)[10] and MRT (Multi-Relaxation Time).

The whole simulation runs inside OpenGL loop which we run for sufficient iterations to get a clear picture of particle behaviour before and after collision.

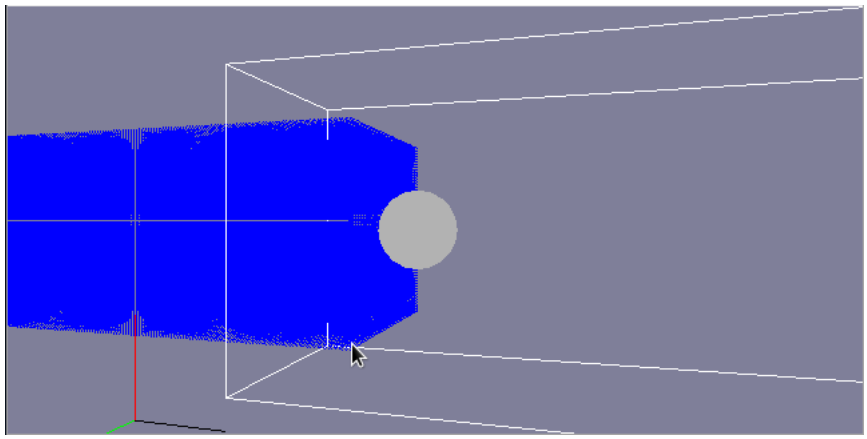


Figure 3.3 Points before hitting the sphere.

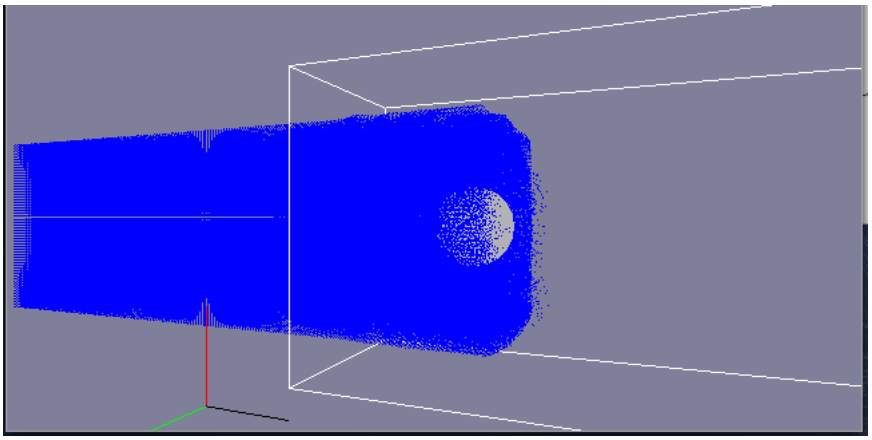


Figure 3.4 After hitting the sphere.

3.4.1 Static Partitioning

Static partitioning is a simple method to divide the work between CPU and GPU. In order to determine this ratio we run the task individually on CPU and GPU. Let's suppose the time taken is t_{CPU} and t_{GPU}.

We can compute effective time t_{sp}, the time taken when both run optimally as

$$\frac{1}{t_{sp}} = \frac{1}{t_{CPU}} + \frac{1}{t_{GPU}}$$
$$t_{sp} = \frac{t_{CPU} \times t_{GPU}}{t_{CPU} + t_{GPU}}$$

Hence any partitioning can not give results better than this. Now we fix the split ratio based on $\frac{t_{CPU}}{t_{GPU}}$.

3.4.2 Work Queue

In this section, we present a generic description of our framework. To this, end, we first consider a few characteristics of workloads for which our framework is suitable. The characteristics we seek are:

- Independent work units: It must be the case that the computation can be broken down into independent subproblems. We call these subproblems as *work units*. It is not necessary that the work units have identical computational requirement.

- Easily describable work units: We seek workloads where it is easy and succinct to describe independent sub problems. For instance, a work unit could correspond to processing a contiguous set of elements, say rows in a matrix.

- Minimal or no post-processing: We seek that the solution to the entire problem be a (near)-immediate consequence of the solutions to the independent work units. So, there should be little post-processing involved.

Some of these characteristics are similar to divide-and-conquer or partitioning design methodologies. Parallel algorithm design has also exploited the above properties to design efficient algorithms (cf. [24]). Using our framework, we also show that algorithm engineering of heterogeneous algorithms can benefit from the above characterization.

We now describe our framework in the context of heterogeneous CPU + GPU systems for ease of exposition. Our proposed solution for dynamic load balancing of the above category of workloads envisages that the multiple threads of the CPU and the GPU share a queue that contains several work units. The individual threads can access the work queue to fetch the next work unit for which computation is still pending. We also have CPU threads and the GPU access the work queue from either end so that there is no need to, in most cases, synchronize accesses to the work queue by the CPU and the GPU. However, CPU threads have to access the queue in a concurrent fashion. Given the low number of CPU threads that one can use, we employ a simple locking mechanism on the CPU `front` variable. See also Figure 3.5 for an illustration.

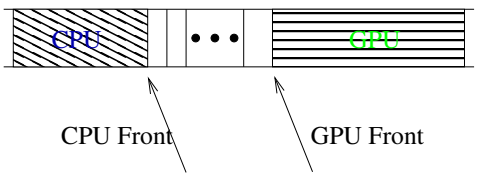


Figure 3.5 Work queue showing the CPU and the GPU front pointers. The shaded region at both ends corresponds to the work processed by the CPU and the GPU respectively.

Note from our description that it is likely that the CPU and the GPU `front` pointers move at different rates. This indicates that the size of the work unit for a CPU thread and a GPU kernel can vary depending on the workload. The ideal size of the work unit for a CPU thread and a GPU kernel can be estimated based on the nature of the workload and other parameters. Using our framework requires one to address a few optimizations for improved performance. Some of these are mentioned below.

- Minimal Synchronization: Firstly, we seek to minimize the synchronization requirement when accessing the queue. For this, we make the queue double-ended. The CPU and the GPU dequeue work units from either ends. The only synchronization required between the CPU and the GPU is for dequeuing the last work unit from the queue. Having a double ended queue also ensures that it is easy to maintain the state of the queue correctly at all times. In practice, we also notice that the synchronization requirement is also most non-existent. This optimization is possible for workloads that possess the characteristic that work units are independent.

- Reducing the Overhead of Queue operations: Secondly, the overhead of queue operations should be kept at a minimum so that heterogeneous algorithms using our framework have a runtime as close to the best possible runtime. For this, we envisage that the queue can actually be maintained only logically, and not physically. So, one need not actually fill the queue with work units before the start of the computation. Initially, the `front` pointer on the CPU side is at work unit 1, and the `front` pointer on the GPU side is at work unit n, assuming there are n work units in total. The progress of the computation can be described by storing the location of the queue `front` pointer at each end. The computation is said to finish when the `front` pointer on the GPU side meets, or crosses, the `front` pointer on the CPU side. This optimization is possible whenever the workload has the characteristic that work units are succinctly describable.

- Other Program Optimizations: We also introduced several optimizations in implementing our framework. For instance, it is well known that there is a small overhead associated with launching a GPU kernel from a host device. To keep this overhead low, we actually launch the GPU kernel only once irrespective of the number of work units that the GPU works. The GPU kernel interacts with the host to fetch multiple work units without exiting execution on the device.

3.4.3 Dynamic Partitioning

In our implementation, we consider up to 1.5 million particles in three dimensions, and simulate their flow around a solid sphere. The size of the work unit is determined empirically. The CPU works on particles numbered $1, 2, \cdots$, and GPU works on particles with numbers $n, n-1, \cdots$, divided into chunks split on the x-axis. The work units are prepared accordingly. The LBM computation is iterative in nature. Since the computation in each iteration is identical, we use the work queue framework from Section 3.4.2 for the first iteration. This allows us to identify the proportion of work to be allotted to

the CPU and the GPU respectively. This proportion is used in all the subsequent iterations. This lets us save the overhead of our framework over multiple iterations.

3.5 Algorithm

The algorithm for Lattice Boltzmann Equation is run over multiple iterations where each iteration is a time step as given by Algorithm 1 below.

Algorithm 1 MainLoop

```
1: Initialise Points array.
2: Initialize directinal velocity array using equation 3.6
3: Initialize OpenGL functions
4: Start OpenGL Loop
5:     stream()
6:     collide()
7:     boundarycheck()
8:     plot the points in Gl window.
```

Streaming step as explained in Equation 3.5 is nothing but simulating points in space by propagating the velocities in corresponding vector directions.

Algorithm 2 stream

```
1: for all points p_{i,j,k}
2:     for all directions d_i
3:         copy f value of the next point in the direction pointed by the vector 3.5.
```

Checkcollide function checks for boundaries and in those cases performs an elastic collision where velocities are reversed.

Algorithm 3 checkcollide

```
1: for all points p_{i,j,k}
2:     check if point collides with sphere.
3:         apply elastic collision property u=-u
```

Collision function is where we apply BGK or MRT as given by Equations 3.7 and 3.10.

3.6 Results

In our experiments we use two platforms namely Hetero-Low and Hetero-High as described in Section 1.3. First we see the speedups between GPU, Static - Hybrid and workqueue on LBM BGK on Hetero Low

Algorithm 4 collide

1: **for all** points $p_{i,j,k}$
2: calculate resultant velocity in x,y,z directions.
3: calculate current position in 3 dimensions.
4: apply collision as given in equations 3.4 and 3.6
5: apply BGK or MRT collision operator as given in equations 3.7 and 3.10

Points in thousand	static split %	Pure GPU time	Hybrid 1	Speedup %	dynamic Split %	Hybrid 2	speedup %
500	19.44	4.01	3.22	19.70	18.33	3.33	16.96
750	20.74	6.19	4.90	20.84	20.37	5.03	18.71
1000	21.11	8.13	6.41	21.16	20.83	6.52	19.75
1250	20.00	10.03	8.01	20.14	19.11	8.12	19.04
1500	20.37	11.968	9.52	20.45	19.81	9.64	19.45

Table 3.1 Performance table of LBM BGK experiments.

In our experiments we use three dimensional particles with coordinates generated uniformly at random. The number of particles is varied from 500 K to 1.5 M. We simulate flow of these points around a solid sphere for 100-400 iterations for each set of points. As it gives sufficient time to hit the sphere and flow around it. For results we plot the averaged value for 100 iterations.

The baseline algorithm in our case is a purely GPU algorithm. We then build a hybrid algorithm that identifies the proportion of work to be allotted to the CPU and the GPU empirically. We them compare the dynamic hybrid algorithm and plot them as shown in Figure 3.6, 3.7, 3.9 and 3.8. As can be seen from Figure 3.12 and 3.13, the speedup difference varies around 3-5% and converges as we increase the number of particles. To see the benefit of our framework from Section 3.4.2, we also measured the absolute difference in the proportion of work allotted to the CPU and the GPU in our algorithm and that of the baseline algorithm. The results of this comparison is shown in Figure 3.10 and 3.11. We see here since the partitioning is limited to a minimum of $(N_y * N_z)$ particles since we split on X direction and our minimum increment size is restricted. Hence we see a very little change in split ratio. Which in turn is responsible for the less overhead over the static hybrid solution.

In the step that checks for whether a particle has reached a boundary or is in collision with sphere we apply inelastic collision properties by reversing the direction of each speed vector. Since different particles may or may not qualify for collision different blocks will have different timing for this step but the time is negligible in comparison to stream and collide and hence can be ignored.

We also see that performance degrades for MRT. This is because the collide function becomes huge due to extra outer for loop which makes GPU a little more efficient and the hybrid solution gets closer to GPU.

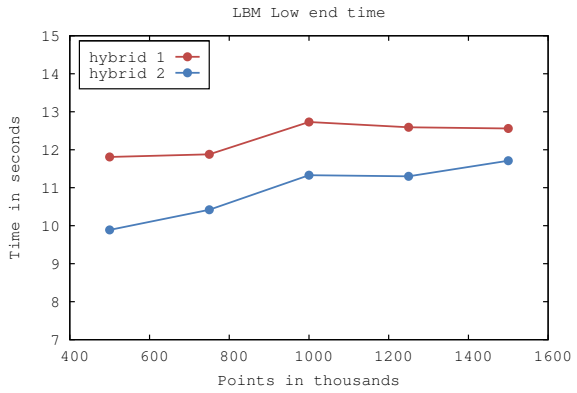


Figure 3.6 Comparison of LBM BGK speedup of Hybrid 1 and Hybrid 2 over pure GPU implementation in high end.

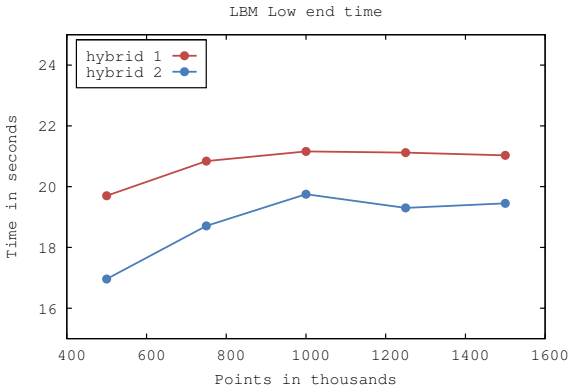


Figure 3.7 Comparison of LBM BGK speedup of Hybrid 1 and Hybrid 2 over pure GPU implementation in low end.

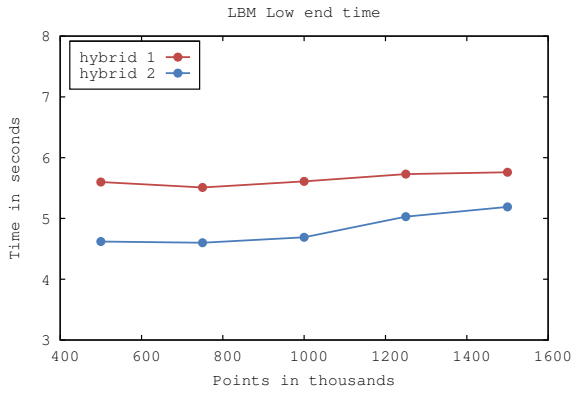


Figure 3.8 Comparison of LBM MRT speedup of Hybrid 1 and Hybrid 2 over pure GPU implementation in high end.

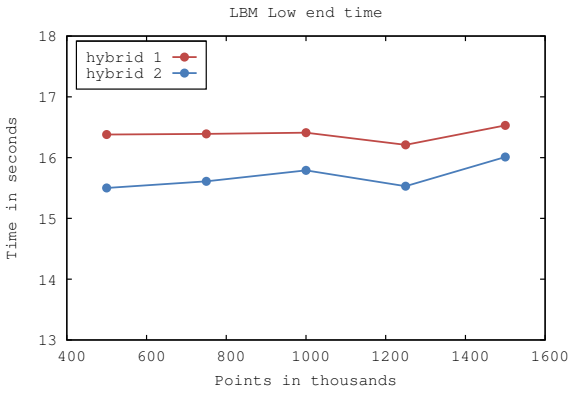


Figure 3.9 Comparison of LBM MRT speedup of Hybrid 1 and Hybrid 2 over pure GPU implementation in low end.

A low % difference in split ratio as seen in Figures 3.10 and 3.11 hints towards the optimality we achieve in static partition.

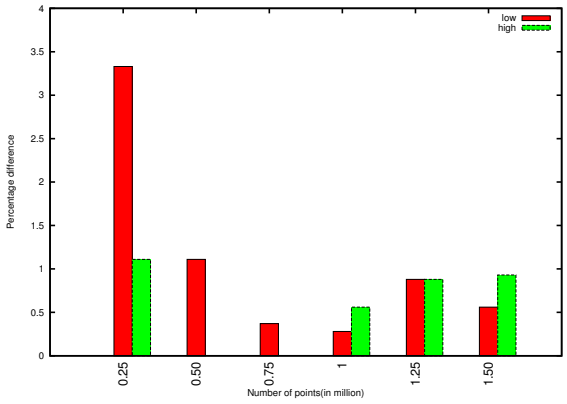


Figure 3.10 Comparison of LBM BGK split ratio of Hetero-High and Hetero-Low over static implementation.

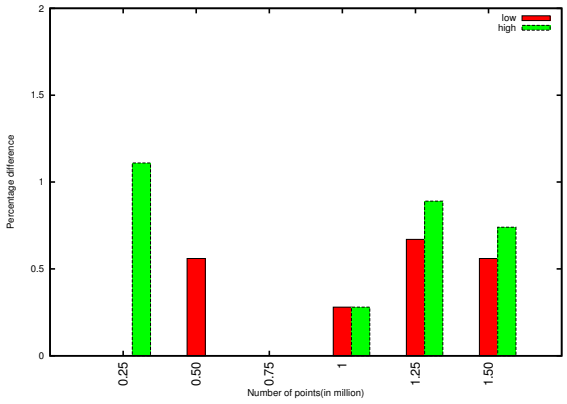


Figure 3.11 Comparison of LBM MRT split ratio of Hetero-High and Hetero-Low over static implementation.

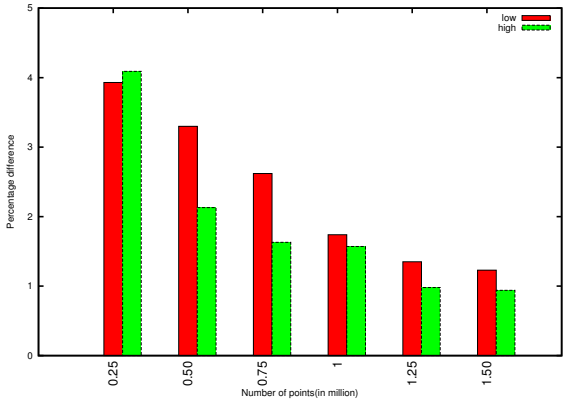


Figure 3.12 Comparison of LBM BGK speed up of Hetero-High and Hetero-Low over static implementation.

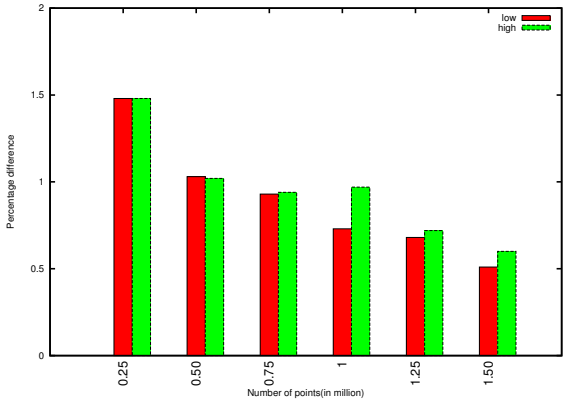


Figure 3.13 Comparison of LBM BGK speed up of Hetero-High and Hetero-Low over static implementation.

Chapter 4

Ray Casting

4.1 Introduction

Ray Casting is an important visual application, used to visualize 3D datasets, such as CT scan data used in medical imaging. High quality image generation algorithms, known as Ray Casting, cast rays through the volume, performing compositing of each voxel into a corresponding pixel, based on voxel opacity and color. Since all rays perform the computations independently, the problem is very much portable for parallel architectures. Tracing multiple rays using SIMD is challenging because rays can access non-contiguous memory locations resulting in incoherent and irregular memory accesses.

We have developed optimized algorithm for both CPU and GPU implementation and then moved to a hybrid version further optimizing into a Work-Queue framework. For this we have worked upon Implicit Adaptive Ray Casting Technique and did a comparative study to benchmark the performance of CPUs and GPUs then proved that hybrid platforms offer better performance. The algorithm is tested on NVIDIA and Intel platforms. The later work queue approach tries to implement an architecture aware algorithm whereby we need not depend on any pre-assessment of work split ratio.

One of the recent works for Ray Casting on modern architectures is that of Lurig et al. [30]. The algorithm is note-worthy for its simplicity and portability to parallel architecture. The algorithm from [30] can be divided into three stages. In the first stage of the algorithm, an interpolation function is calculated for each tetrahedron. In the second stage, ray entrance calculations are performed for the rays to be traversed. In the final stage, the rays are then traversed through the polygon to accumulate color and intensity values that are used for final rendering.

4.2 GPU Ray Casting

The complete Ray Casting Algorithm consists mainly of the following steps

- Preprocessing
- Initializing the interpolation function

- First hit calculation
- Ray Casting through the tetrahedra mesh

In the preprocessing stage, triangles are generated from the tetrahedra information and are processed in order to separate the front facing surface triangles from rest of the triangles. The interpolation function for each tetrahedron is generated from the intensity values and the coordinate values given for each vertex of a tetrahedron. The Surface triangles are ray traced to get the first hit point/triangle. This is then used to find the first tetrahedron intersected, which is used in the next step of traversing through the tetrahedra mesh. The Ray is initialized from the first hit point and traversed through the entire mesh to keep accumulating intensity values from the interpolation function. Ray is traversed until it goes out of the mesh, at which point the process terminates.

4.2.1 Preprocessing

The data provided is in the form of vertices of tetrahedron and their intensity values. Hence the first step is to generate a list of triangles from the list of tetrahedrons. Next, the vertex index for each triangle is sorted in ascending order and then the whole list of triangles is sorted based on the vertices index value.

This is done in order to separate the outer surface triangles from the inner triangles. A triangle having same vertices index as the triangle below or above is an inner triangle, otherwise its a surface triangle. Also, this leads to generating of the duplicate list for each triangle. The duplicate list contains the index of the other tetrahedron where the same triangle occurs and is -1 in case of surface triangle.

Next,we determine the normal for each triangle. The normal is calculated in the following way:

- For each triangle in the surface list, find the corresponding tetrahedron.
- Find the 4^{th} vertex of the tetrahedron.
- Find the cross product : $(\vec{v}_2 - \vec{v}_1) \times (\vec{v}_3 - \vec{v}_1)$
- Substitute the 4^{th} vertex of the tetrahedron to find the final direction of the normal

The next step is to apply back-face culling on the generated list of surface triangles. This is achieved by:

- Find the direction vector for rays from camera center to the three vertices of the triangle.
- Find the dot product of the rays with the normal of the triangle.
- If all three comes out to be positive (angle between -90° and 90°), then the triangle can be culled / removed.
- Maintain an array which stores 1 for triangles which are not front facing.

- Rearrange to get the final list of front facing triangles.

We apply projection and model view matrix on the vertices of the generated front facing triangles to get the initial bounds on the rays which need to be ray traced in the first hit stage. Thus, we do not need to trace all the rays, but only those which lie in the bounds.

4.2.2 Initialization of the interpolation function

The calculation of interpolation function involves the intensity values at the four vertices of the tetrahedron and their coordinates. It is computed for every tetrahedron in the dataset. It is of the following form:

$$f(x, y, z) = a + bx + cy + dz \tag{4.1}$$

The calculation reduces down to four linear equations in four variables and is solved by applying Cramers rule. This is performed in parallel in the GPU for each tetrahedron in the dataset.

$$\begin{pmatrix} 1.0 & x_1 & y_1 & z_1 \\ 1.0 & x_2 & y_2 & z_2 \\ 1.0 & x_3 & y_3 & z_3 \\ 1.0 & x_4 & y_4 & z_4 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{pmatrix} \tag{4.2}$$

4.2.3 First Hit Calculation

Prior to moving through the tetrahedra mesh and accumulating intensity values, we need to find the first triangle intersected by each ray. The algorithm is detailed in Algorithm 5. This process is performed in parallel for each ray on the GPU. Each triangle is first loaded into the shared memory of the GPU and then tested for intersection with every ray. The use of shared memory leads to a much higher speedup as compared to loading from global memory. This stage generates a much higher ratio of CPU:GPU time because of the use of shared memory. The intersection test used for ray-triangle intersection is given in Algorithm 6.

4.2.4 Ray Casting through the tetrahedra mesh

Upon finding the first intersection with a tetrahedron, the ray is traversed in front to back manner through the tetrahedra mesh. The ray is checked for intersection with rest of the three triangles of the current tetrahedron except the ray entrance triangle. The test outputs the next triangle getting intersected and also the t parameter value, which is then used to find the exit point of the ray for the tetrahedron. Intensity values for a tetrahedron is found by integrating the values given by the interpolation function at the start and end point of a tetrahedron for each ray. The integration is performed using the trapezoid rule. The process is repeated until the ray exits the mesh. The overall algorithm is given in Algorithm 7.

4.3 Implementation

We developed a hybrid version of the overall algorithm. This is done in order to additionally utilize the power of modern CPUs. The schematic for the hybrid implementation is given in Figure 4.1. Since the computational time required for initializing the interpolation function is small, it is performed purely on the GPU. This gives CPU a bit of a head start to start the raytrace kernel early, so as to process more number of rays, thereby decreasing the overall time. The total number of rays which are tested for first intersection are divided in a ratio α determined later.

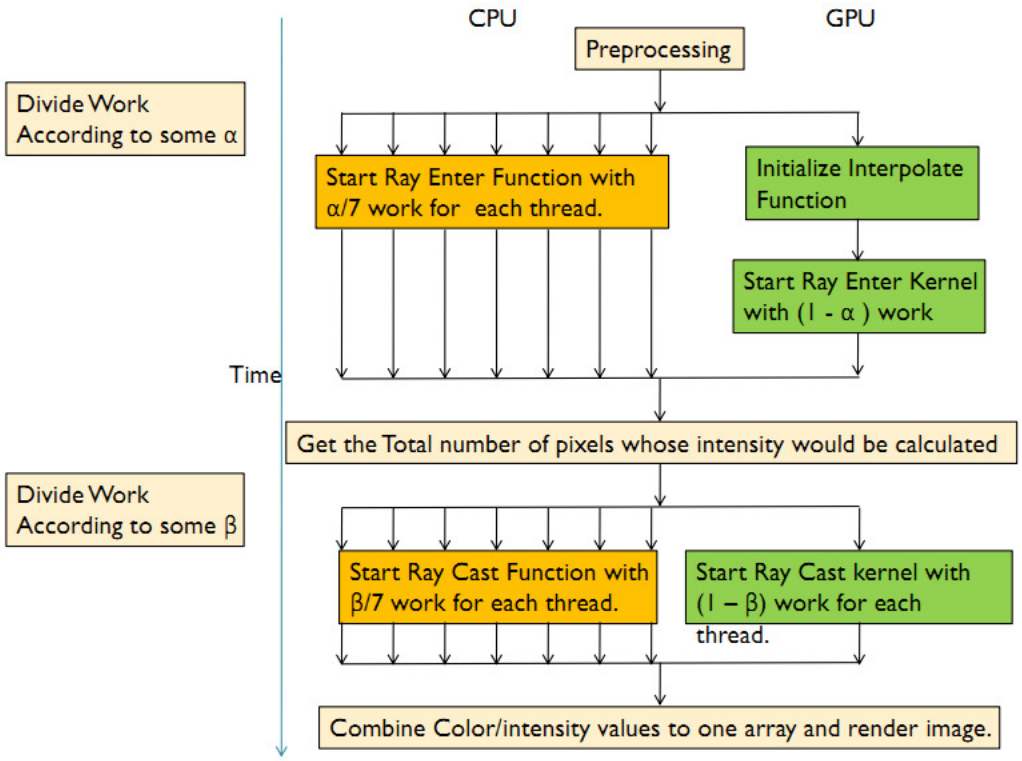


Figure 4.1 Hybrid Implementation.

We experimentally found that keeping total number of CPU threads to be twice the number of cores gives the best results. Hence, $\frac{\alpha}{2*N-1}$ amount of work is performed by CPU threads, where N is total number of cores One thread is used for calling the GPU kernels. By carefully optimizing the CPU code, we are able to get good speed-ups for the system.

The output data after the first stage is then compacted and further divided according to β ratio to be processed on the CPU-GPU in parallel. This step is important as there is a vast difference in α and β values and hence it provides much better speed-up as compared just having one ratio for the overall process. The intensity values for all the processed rays are then again compacted and then rendered on the screen.

Algorithm 5 Overview of GPU First Hit

1: **f**or all rays in parallel on the GPU do
2: mint = FLOAT_MAX
3: triangle_ind = -1
4: **f**or all triangles in the surface triangle list **do**
5: Store triangle vertices in shared memory
6: $tval = triangleintersect(ray_i, triangle_j)$
7: **if** $tval < mint$ **then**
8: $mint = tval$
9: triangle_ind = j
10: **endif**
11: **endfor**
12: **endfor**

Algorithm 6 Ray-Triangle Intersection

1: **function** intersect(origin,direction,v1,v2,v3)
2: $edge1 = v2 - v1$
3: $edge2 = v3 - v1$
4: $s1 = direction \times edge2$
5: $divisor = s1 \cdot edge1$
6: **if** divisor = 0.0 **then**
7: **return** FLOAT_MAX
8: **endif**
9: $invDiv = 1/Divisor$
10: $distance = Origin - v1$
11: $barycCoord_1 = (distance \cdot s1) * invDiv$
12: **if** $barycCoord_1 < 0.0 or barycCoord_1 > 1.0$ **then**
13: **return** FLOAT_MAX
14: **endif**
15: $s2 = distance \times edge1$
16: $barycCoord_2 = (distance \cdot s2) * invDiv$
17: **if** $barycCoord_2 < 0.0 or barycCoord_1 + barycCoord_2 > 1.0$ **then**
18: **return** FLOAT_MAX
19: **endif**
20: $intersectionDistance = (edge2 \cdot s2) * invDiv$
21: **return** intersectionDistance
22: **endfunction**

Algorithm 7 Ray Casting through the mesh

for all Rays getting a first hit in parallel **do**
  $currTri = currtrilist[ray_i]$
  $currTet = tetindlist[currTri]$
  $direction = directionlist[ray_i]$
  $tval = tvaluelist[ray_i]$
  $nexttri = -1$
  $startPoint = Origin + tval * direction$
  while nexttri is not a surface triangle **do**
    $mint = FLOAT\_MAX$
    for rest 3 triangles of the currTet **do**
      $tval = triangleintersect(ray_i, triangle_j)$
      if $tval < mint$ **then**
        $mint = tval$
        $nexttri = j$
      endif
    endfor
    $endPoint = Origin + mint * direction$
    $len = computelength(startPoint, endPoint)$
    $fa = interpolate(startPoint, intercoeff)$
    $fb = interpolate(endPoint, intercoeff)$
    $intensityValue+ = len * \frac{(fa+fb)}{2}$
    $startPoint = endPoint$
    $currTri = duplicate[nextTri]$
    $currTet = tetindlist[currTri]$
  endwhile
end for

Dataset	static split %	Pure GPU time	Hybrid 1	Speedup %	dynamic Split %	Hybrid 2	speedup %
spx	34.71	116.79	71.71	38.60	30.01	87.68	24.92
fighter	31.64	595.12	393.78	33.83	31.00	442.45	25.65
bluntfin	28.49	1895.1	1319.06	30.39	31.00	1367.35	27.85

Table 4.1 Performance table of LBM BGK experiments on low end.

We note that the computation for each ray is indeed independent of the computation with respect to other rays. Further, a work unit in this case may correspond to the computation involved across a bunch of rays. This work unit can also be described succinctly by indexing the starting and ending ray numbers. Also, there is no post-processing involved at the end. Thus, Ray Casting satisfies the characteristics of workloads described in Section 3.4.2.

We use the algorithmic model described in Section 3.4.2. A size of the work unit for the CPU and the GPU is chosen empiricially. We kept the size of the GPU work unit to be six times more than that of the CPU work unit size. This arrangement leads to lesser number of kernel calls while keeping both the CPU and the GPU busy till the very end. The number of rays used depends on the input image. Let the rays be numbered from 1 through n. Applying the framework from Section 3.4.2, the computation corresponding to rays with indices starting from 1 is assigned to the CPU. The computation corresponding to the rays with indices from n backwards is assigned to the GPU. Since we use a multicore CPU, a block of rays allotted to the CPU is divided among the CPU threads equally. The CPU threads wait for the master CPU thread to do this work distribution. This reduces the synchronization overhead at the work queue. However, due to the waiting time of CPU threads, the overall time taken by our algorithm is slightly larger than the time taken by a heterogeneous algorithm that uses a brute-force method to identify the work split percentage and then uses such a work distribution.

4.4 Experimental Setup

We use three different images: spx, fighter and bluntfin in our experiments. These input images have been part of the standard inputs used in most of the recent works on volumetric Ray Casting. The total number of tetrahedrons vary greatly among the three images and hence the results provide a fairly good overview of the algorithm. For the spx, the total number of tetrahedrons is $12,936$, the fighter image has $70,125$ tetrahedrons, and the bluntfin image has $222,414$ tetrahedrons.

4.5 Results

The baseline implementation we use is a heterogeneous implementation of the algorithm of Lurig et al. [30]. The baseline implementation identifies the best work split percentage empirically. The results are reported for 512×512 resolution, with 10-15% screen coverage as shown in 4.1. The results are averaged over multiple iterations on two different heterogeneous platforms as described in Section 1.3. The two of datasets are shown in 4.2 and 4.3

Figure 4.2 Rendered image for SPX dataset.

Figure 4.4 shows the difference in the work split percentage achieved by our implementation with respect to the baseline implementation. As can be seen, the difference in under 5% on all the three images. Figure 4.5 shows the difference in the runtime between our implementation and the baseline implementation. The average idle time of our implementation is also under 2% on both platforms. These results also indicate the applicability and efficacy of our model.

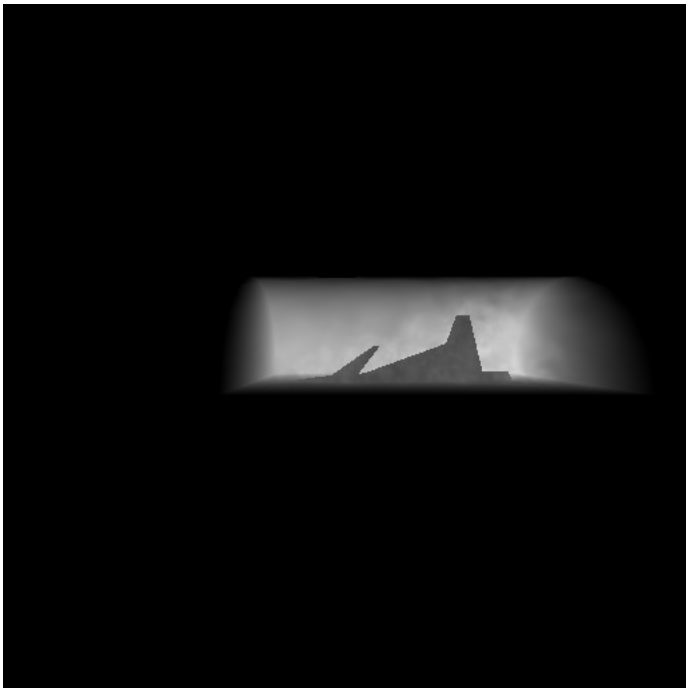


Figure 4.3 Rendered image for fighter dataset.

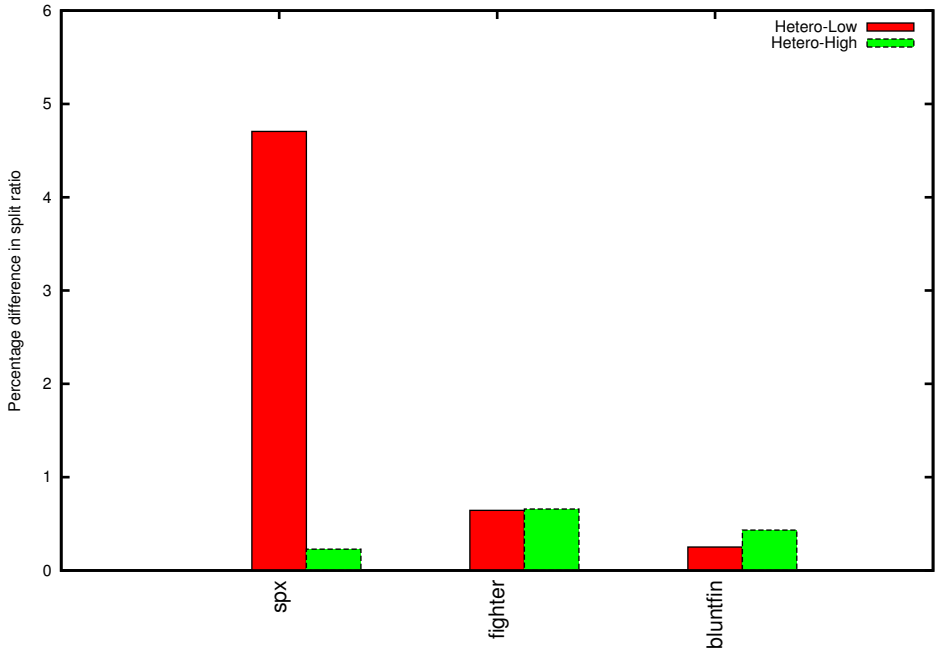


Figure 4.4 Ray Casting split ratio difference.

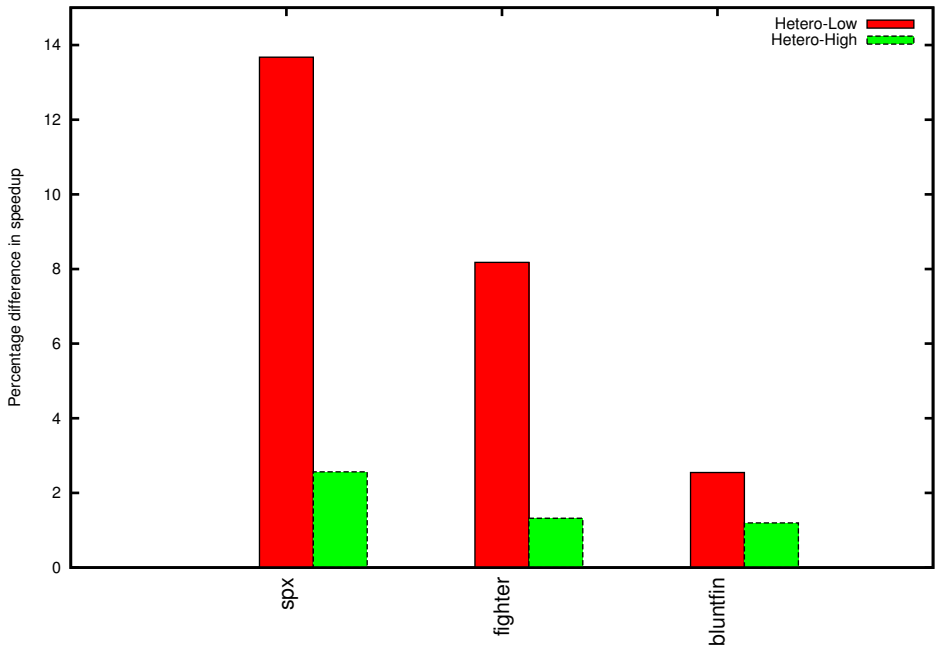


Figure 4.5 Ray Casting speedup difference.

Chapter 5

All Pairs Shortest Paths(APSP)

5.1 Introduction

Graph algorithms find a large number of applications in engineering and scientific domains. Prominent examples include solving problems arising in VLSI layouts, phylogeny reconstructions, data mining, image processing etc. Some of the most commonly used graph algorithms are graph exploration algorithms such as Breadth First Search (BFS), computing components, and finding shortest paths. As the current real life problems often involve the analysis of massive graphs, it is often seen that parallel solutions provide an acceptable recourse.

Heterogeneous algorithms that aim to utilize all the computational devices in a commodity heterogeneous platform have also been designed for graph breadth-first exploration [23, 31, 18]. Most of these use platforms consisting of multicore CPUs and GPUs. All of the above-cited works show an average of 2x improvement over pure GPU algorithms. Most of the above works in general aim at data structure optimizations but largely run classical algorithms on the entire input graph. These algorithms are designed for general graphs whereas the present real world graphs possess markedly distinguishable features such as being large, sparse, and large deviation in the vertex degrees. In Figure 5.3, we show some of the real-world graphs taken from University of Florida Sparse Matrix Collection dataset [1]. As can be seen from Figure 5.3, these graphs have several vertices of very low degree, often as low as 1. For instance, in the case of the graph web-Google, 14% of the vertices have degree 1. Table 5.1 lists other properties of a few real world graphs from [1].

Current parallel algorithms and their implementations [31, 18, 35, 23, 46] do not take advantage of the above properties. For instance, in a typical implementation of the breadth-first search algorithm, one uses a queue to store the vertices that have to be explored next. But, a vertex v of degree 1 that is in the queue will not lead to the discovery of any yet undiscovered vertices. So, the actions of BFS with respect to v such as adding it to the queue, dequeue it, and then realize that there are no new vertices that can be discovered through vertex v are all unnecessary. These actions unfortunately can be quite expensive on most modern parallel architectures as one has to take into account the fact that the queue is

to be accessed concurrently. Similarly, other operations such as checking of the status of a vertex, may be quite disposable.

Parallel computing on graphs however is often very challenging because of their irregular nature of memory accesses. This irregular nature of memory access also stresses the I/O systems of most modern parallel architectures. It is therefore not surprising that most of the recent progress in scalable parallel graph algorithms is aimed at addressing these challenges via innovative use of data structures, memory layouts, and SIMD optimizations [31, 20, 35]. Recent results have been able to make efficient use of modern parallel architectures such as the Cell BE [35], GPUs [31, 23, 20], Intel multi-core architectures [14, 50, 2] and the like. Algorithms running of GPUs have shown standout performance amongst these because of its massive parallelism.

In light of the above paragraph, we envisage that new algorithms and implementation strategies are required for efficient processing of current generation graphs on modern multicore architectures. Such strategies should help algorithms and their implementations benefit from the properties of the graphs. In this thesis, we propose graph pruning as a technique in this direction. Graph pruning aims to reduce the size of the graph by pruning away certain elements of the graph. The required computation is then performed on the remaining graph. The result of this computation is then extended to the pruned elements, if necessary.

In this chapter, we apply the graph pruning technique to All Pairs Shortest Paths(APSP). In this case, we show that pruning pendant nodes iteratively can result in reducing the size of the graph on real-world datasets, by as much as 25% in some cases. This reduction in size helps us achieve remarkable improvements in speed for the above three workloads by an average of 35%.

In graph theory, finding shortest paths in a weighted graph $G(V, E, W)$ is a fundamental and well researched problem. The problem seeks to find the shortest path between any two vertices **u,v** of the graph such that the sum of the weights of the constituent edges is minimized. The All Pairs Shortest Paths(APSP) problem is a generalization where one seeks to find the shortest path between every pair of vertices in the graph.

The most popular solution of the APSP problem is the Floyd-Warshall [45] algorithm which has a $\mathcal{O}(V^3)$ running time and a $\mathcal{O}(V^2)$ space complexity. As the Floyd-Warshall algorithm is generally not well suited for sparse graphs, there are special algorithms designed for sparse graphs [25]. On GPUs, there are very few reported implementations. Notable among these are those of Harish et al. [20], Katz et al. [27], and [45]. In Harish et al. [20], the problem is solved by running a parallel Dijkstra's algorithm [15]. This is shown to be better for sparse graphs. We try to further improve upon by a parallel pendant node removal step before running the parallel Dijkasta's algorithm.

5.2 Implementation

We first prune the pendant nodes in the graph iteratively which is further done in two steps namely parallel find pendant nodes and parallel remove pendant nodes. Notice that for a pendant vertex v with

w as its only neighbor, the shortest path from v to any other vertex u will always pass through w. Therefore, pendant vertices can be safely removed from the graph in the pruning step and the required shortest paths can be easily computed. For instance, for the graph in Figure 5.2, the shortest path from vertex 5 to vertex 4 has to necessarily go through the only neighbor of vertex 5, that is vertex 3. Further, if vertex 3 is removed in the second iteration of Phase I, then the shortest path from vertex 3 to any other vertex has to necessarily go through its only remaining neighbor, i.e., vertex 1.

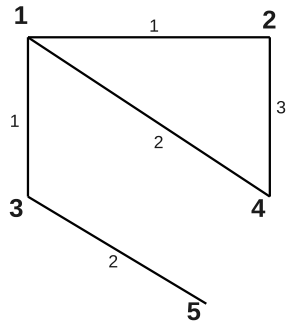


Figure 5.1 APSP example.

Hence, in the first phase we prune the pendant vertices. We remove all the pendent vertices of graph G and obtain G' over a few iterations along with the book keeping of removed nodes along with their iteration number.

Graphs are commonly represented as an adjacency matrix but this representation in sparse graphs wastes a lot of space. Therefore, in this implementation, we use the compact adjacency list which is more popularly known as the compact sparse representation (CSR). Here each vertex points to the starting point of its own adjacency list in the large array of edges. Vertices of Graph $G(V, E)$ are represented as array V_a and another array E_a stores the edges with edges of vertex i+1 immediately following the vertex of edges i for all i in V. As shown in figure 5.2

A separate weight array is maintained for storing the weights. The weight function W on the edges associates a random weight with each edge. As is done in [20], we run a single-source-shortest-paths (SSSP) algorithm from each vertex in graph $G(V, E, W)$. A parallel implementation of Dijkstra's algorithm is used to solve SSSP.

UF Sparse Matrix Collection				
Graph	**Nodes**	**Edges**	**Pendant Vertices**	**r**
internet	124,651	207,214	56,959 (**45%**)	2
dblp_2010	326,183	807,700	87,881 (**26%**)	3
watson_1	386,992	1,055,093	44,637 (**11%**)	2
webbase_1M	1,000,005	3,105,536	80,053 (**8%**)	4
wiki-Talk	2,394,385	5,021,410	176,617 (**7.37%**)	4
web-Google	916,428	5,105,309	134,452 (**14.6%**)	3
rail2586	923,269	8,011,362	117,342 (**12.7%**)	4
tp-6	1,014,301	11,537,419	194,764 (**19%**)	4

Table 5.1 The graphs used for experimentations and their properties. The column heading r in the last column indicates the number of iterations required to remove all pendant vertices.

R-MAT Graphs				
Graph	**Nodes**	**Edges**	**Pendant Vertices**	**r**
rmat_1	131,072	256,784	43,556 (**33.2%**)	1
rmat_2	263,144	554,678	67,345 (**25.6%**)	2
rmat_3	525,288	1,048,515	55,453 (**10.5%**)	2
rmat_4	1,056,534	2,097,345	78,234 (**7.4%**)	3
rmat_5	2,045,266	4,190,223	92,345 (**4.5%**)	4
rmat_6	3,074,344	8,456,240	105,117 (**3.4%**)	3
rmat_7	3,156,834	12,673,552	132,443 (**4.2%**)	4
rmat_8	3,765,223	16,673,993	153,442 (**4.1%**)	4

Table 5.2 The graphs used for experimentations and their properties. The column heading r in the last column indicates the number of iterations required to remove all pendant vertices.

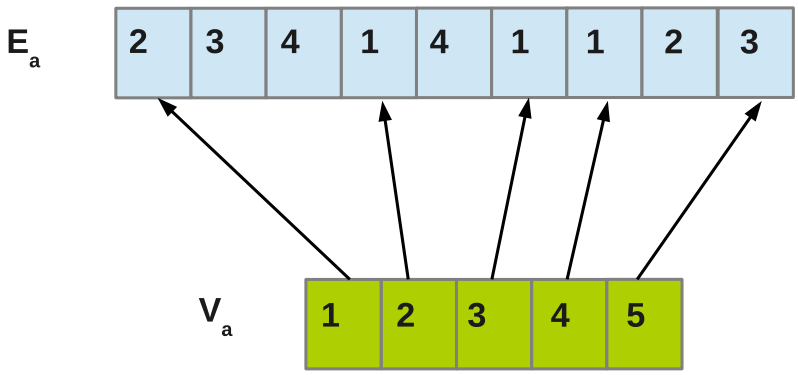


Figure 5.2 CSR representation of Graph $G(V, E)$

5.2.1 Leaf Prunning

In this section, we present a three phase technique, outlined in Algorithm 8, for scalable parallel graph algorithms of real world graphs. In the first phase, called the *preprocessing phase*, we reduce the size of the input graph by removing *redundant* elements of the graph. Once the graph size reduces, the second phase involves using existing algorithms to perform the computation on the smaller graph. In a final phase, we then extend the result of the computation to the entire original graph via quick post-processing, if required.

Let G be a large, sparse graph. As mentioned in Algorithm 8, let Prune(G) be a function that can prune certain elements of G. Let G' be the graph that remains after Prune(G). Let A be an algorithm that can compute the desired solution. We then use algorithm A on the graph G'. Let O' be the solution on G'. In a post-processing third phase, we extend the solution O' to a solution O of the entire graph G.

Algorithm 8 $ProcessGraph(Graph\ (\mathcal{V}, \mathcal{E}))$

1: **/* Phase I – Prune */**
2: $G' =$ Prune(G)
3: **/* Phase II – Compute */**
4: $O' = A(G')$
5: **/* Phase III – Extend */**
6: $O =$ Extend(G, O')

We note that if Phase I prunes only a constant fraction of the size of the graph, and one uses a standard algorithm in Phase II, then the asymptotic runtime using the above technique is still unchanged. How-

ever, even such a constant fraction reduction in size can have a considerable impact on the experimental efficacy.

We envisage that different graph algorithms can benefit from corresponding pruning processes in Phase I. Further, Step 1 may also be performed iteratively. Each iteration may prune some nodes after which more nodes may become candidates for pruning in the next iteration. We refer the reader to Algorithm 9 for an illustration. In Algorithm 9, $\mathcal{P}$ refers to a property that vertices that are pruned will satisfy. Similarly, the post-processing in Phase III can also be based on the problem at hand. If Phase I is spread over multiple iterations, then Phase III may also be spread over multiple iterations, possibly in the reverse order of iterations of Phase I.

Algorithm 9 $Prune(Graph\ (\mathcal{V}, \mathcal{E}))$

```
1: for i = 1 to r iterations do
2:     for each vertex v ∈ G do
3:         if v has property P then
4:             Remove v, and all edges incident on v.
5:             Store (v, i) for future re-insertion step.
6:         endif
7:     endfor
8: endfor
```

It is important to note that the property $\mathcal{P}$ can be evaluated quickly. This helps keep the overall time for Phase I small. The time taken by a graph algorithm using our technique will depend on the extent of pruning achieved in Phase I and also the time taken in Phase I and III. As can be noticed, in most cases, there will also be a trade-off between time taken by Phase I and III and that of Phase II. In fact, such a trade-off is observed in the case of list ranking [7].

Some of the properties that may be of interest are the following.

- Pendant nodes: Let us call a node v in a graph G as a pendant node if the degree of v is G is 1. For the three workloads we consider in this paper, we show that a simple pruning based on removal of pendant nodes suffices. This is also the pruning technique used in [17].

- Independent nodes: A subset of nodes is called as an independent set of nodes if they are mutual non-neighbors. This has been used in list ranking algorithms [4] and its recent heterogeneous implementation [48, 7].

- Graph partitioning: Graph partitioning calls for partitioning a graph G into a specified number k equal partitions such that the number of edges that have end points in different partitions is minimized. In an influential work, Karypis and Kumar [26], introduce the coarsening-refinement approach. During the coarsening step, a matching of the current graph is computed and prunes are matched with vertices.

The above examples indicate that various properties $\mathcal{P}$ can have applicability to different problems. Thus, our approach is quite general. It must be noted that all the above examples are not implemented on

modern parallel architectures. In this paper, we show that our technique can be used on modern parallel architectures too.

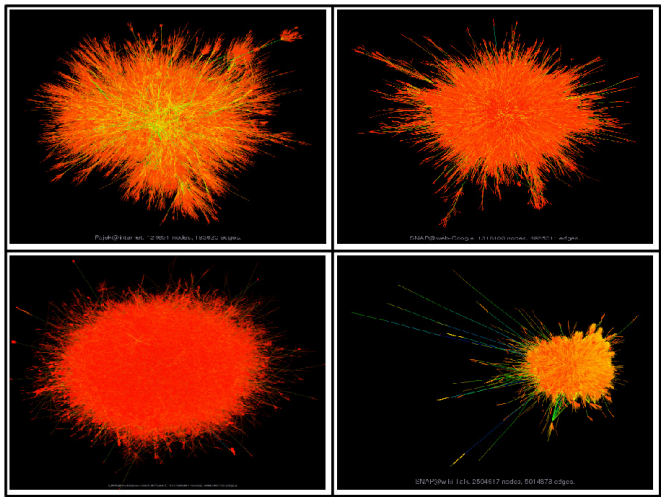


Figure 5.3 A sample of four real world graphs from [1]. On the top-left corner is the graph internet, top-right is the graph web-google, bottom left is the graph webbase_1M, and the bottom-right is the graph wiki-Talk.

5.2.2 Algorithm

In Step 1 we use our leaf pruning method as described in Section 5.2.1. The removed vertices are stored with proper book keeping. In Step 2 we run the algorithm as described in this section, and later we update our results by adding the nodes back into the graph and updating the vertex cost of those pendent vertex by adding their weight to the weight of the parent.

The basic step in SSSP is to select a vertex and update it's neighbors depending upon the minimum cost. but instead of traversing the graph level by level we do multiple calls of parallel Dijkstra's algorithm until there is no change in cost. Hence we do an only $\mathcal{O}(V)$ operation on each of the threads. In the first kernel, the shortest path is determined by setting masks on the neighbor vertices of each vertex. We fetch each vertex from V_a. Each vertex has a corresponding mask. If it is set, then a second kernel is called where it fetches it's cost and then add the weights of its neighbors. If that is less than the cost of neighbor it updates the neighbors costs. This updation is done in a second kernel in order to avoid read/write inconsistencies caused by other vertices. We also use atomic operation to calculate the minimum cost in order to resolve inconsistencies.

Shared memory is used to prefetch all the neighbors of a certain node while it is being processed. The entire execution happens only on the GPU for ease of comparison. (There are no hybrid implementations of APSP reported). An example run of our algorithm is presented in Figure 5.2.2

Algorithm 10 $APSP(Graph\ \mathcal{G}(V, E, W))$

Require: A graph $\mathcal{G}(V, E, W)$
Ensure: Path cost from every vertex to every other vertex.

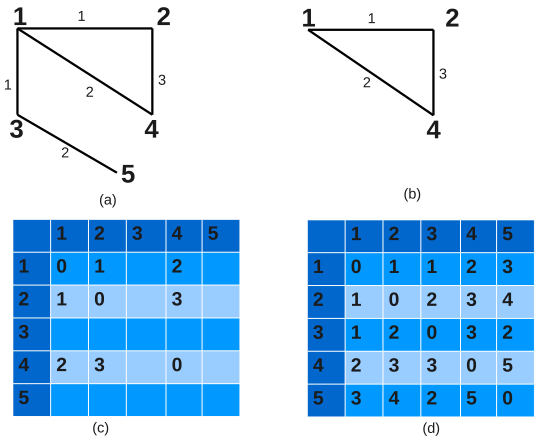
Call Algorithm 9 and shorten the graph.
Create mask array M_a, cost array C_a and updating cost array U_a of size V
for S $\in$ all the vertex from 1 to V
    $M_a[s]$=TRUE
    $C_a[s]$=0
    while M_a not empty
        for each vertex V do in parallel
            Call SSSP_KERNEL1($\mathcal{G}$, Ma ,Ca ,Ua).
            Call SSSP_KERNEL2($\mathcal{G}$, Ma ,Ca ,Ua).
        endfor
    endwhile
endfor
Re-insert the nodes removed in Step 1.

Algorithm 11 $SSSP\_Kernel1(\ \mathcal{G}, M_a, C_a, U_a)$

id = threadIdx.y*gridDim.x+blockIdx.x
if $M_a[id]$
    $M_a[id]$ = FALSE
    for all neighbors N_{id} of vertex id
        if $U_a[N_{id}] > C_a[id] + W_a[N_{id}]$
            $U_a[N_{id}] = C_a[id] + W_a[N_{id}]$
        endif
    endfor
endif

Algorithm 12 $SSSP\_Kernel2(\ \mathcal{G}, M_a, C_a, U_a)$

id = threadIdx.y*gridDim.x+blockIdx.x
if $C_a[id] > U_a[id]$
    $C_a[id] = U_a[id]$
    $M_a[id]$ = TRUE
endif
$U_a[id] = C_a[id]$



	1	2	3	4	5
1	0	1		2	
2	1	0		3	
3					
4	2	3		0	
5					

	1	2	3	4	5
1	0	1	1	2	3
2	1	0	2	3	4
3	1	2	0	3	2
4	2	3	3	0	5
5	3	4	2	5	0

Figure 5.4 APSP example.

5.3 Results

In this section, we present the results of our implementation. We compare our implementation results with those of [27]. The results of [27] are the currently reported best results for finding the shortest path of a given graph on identical platforms.

In Figure 5.5, we run our implementation on a sample of eight real world graphs from the University of Florida dataset [1]. As shown in Figure 5.5, our results are consistently better. This improvement can be attributed to the fact that APSP is a highly computationally intensive workload, roughly of cubic order. Hence, any decrease in the size of the graph reflects prominently in a corresponding decrease in the overall runtime. A similar experiment on the R-MAT graphs from Table is shown in Figure 5.6. It can be noticed from Figure that there is a considerable improvement of 44% on average.

We also study the percentage improvement of our implementation as a function of the percentage of nodes that Phase I can remove. The results of this experiment are shown in Figure 5.7 on the graphs webbase_1M, wiki-Talk and Web_Google from the dataset of [1]. While there are not many irregular operations in the APSP implementation on a GPU, the workload is still computationally heavy. This can be observed from the total runtime from Figures 5.5 and 5.6, where runtimes are of the order of one minute. Hence, pruning even a small fraction of the nodes results can potentially decrease the runtime by a large percentage.

Finally, we also study the trade-off between the number of iterations of Phase I and the overall runtime. Figure 5.8 shows the results of the above experiment on the graph webbase_1M from the dataset of [1]. It can be noticed that the overall time taken decreases over iterations of Phase I and plateaus off at about six iterations for the graph under consideration. The time for Phase I is shown on the right-side Y-axis of Figure 5.8.

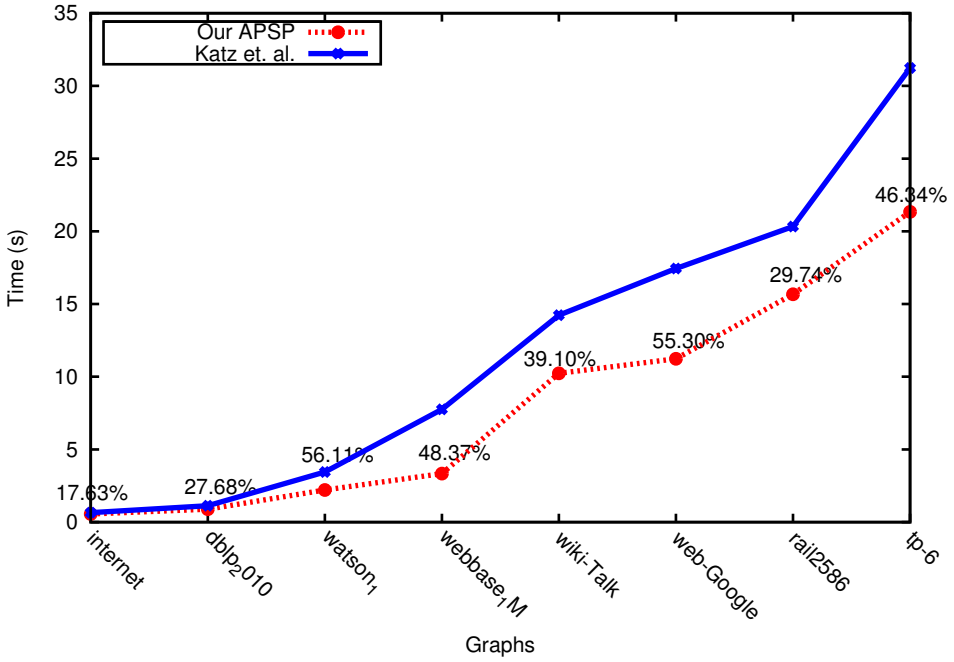


Figure 5.5 APSP timing over University of florida datasets

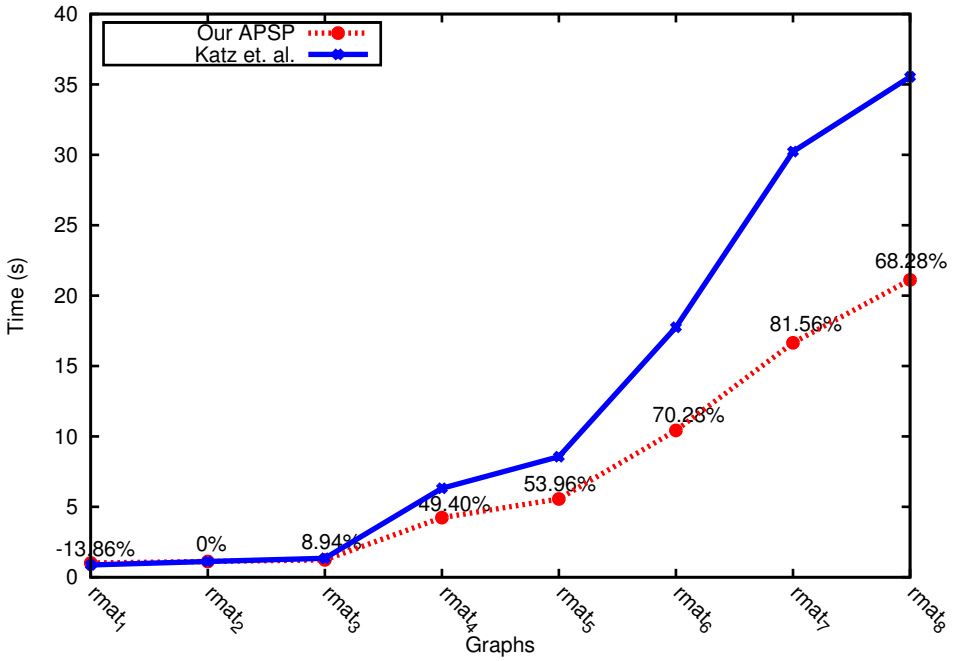


Figure 5.6 APSP timing over random matrices

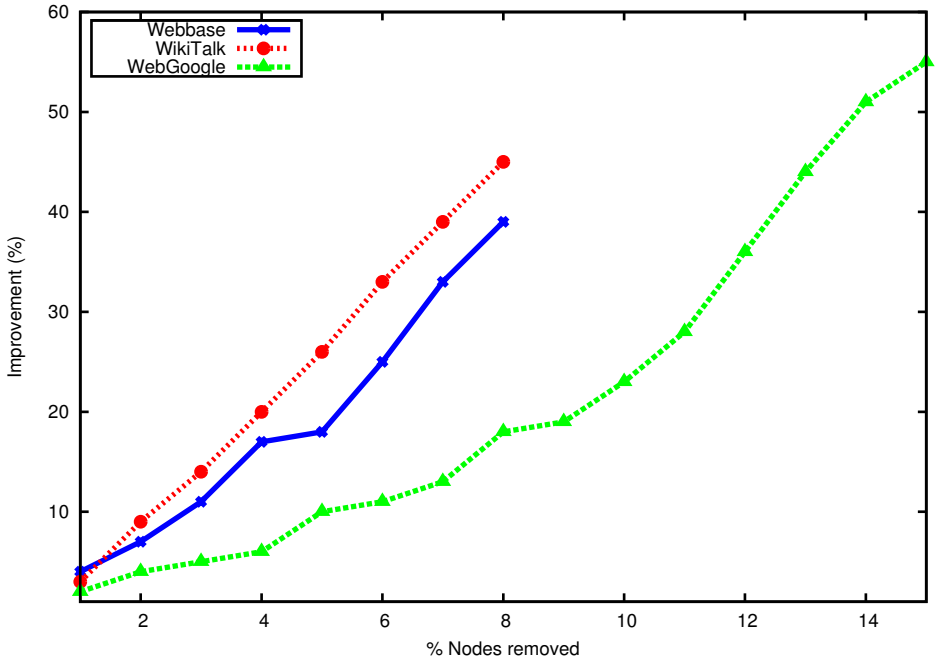


Figure 5.7 APSP speedup improvement over % removed nodes

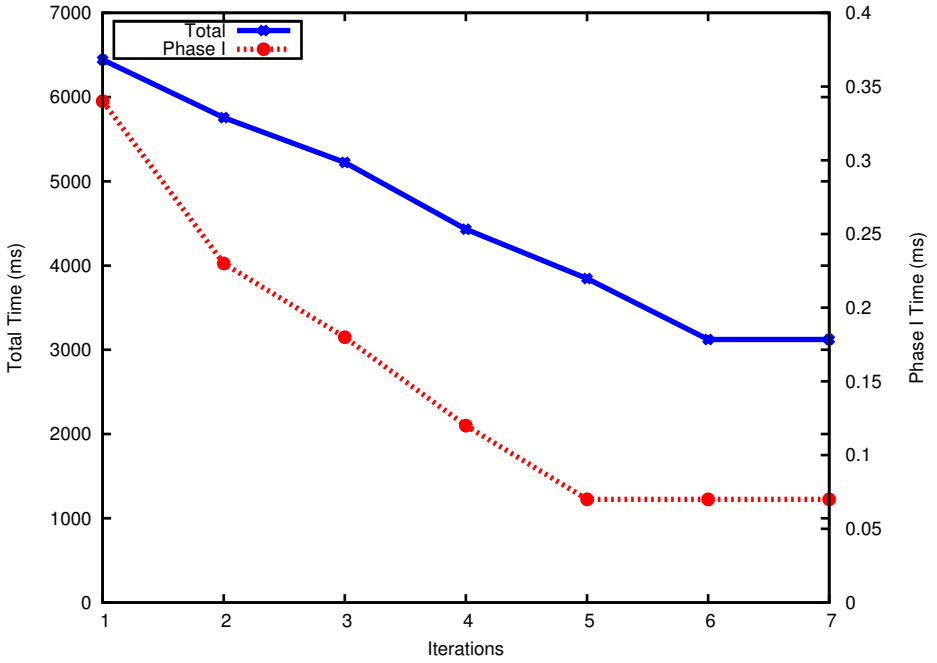


Figure 5.8 APSP timing over no. of iterations

Chapter 6

Conclusions

In the initial part of this thesis we have studied hybrid algorithms for Lattice Boltzmann Method and Ray Casting, and later we proposed and demonstrated a simple and efficient mechanism for dynamic load balancing of heterogeneous algorithms and demonstrated them on LBM and RC respectively. Our results show that our implementations achieve good performance with respect to the plain GPU algorithms and later the work split method which resulted in a more hardware aware solution without losing much in split percentage and runtime compared to corresponding best reported static hybrid implementations.

Our work therefore seems to partly solve the important problem of engineering heterogeneous algorithms using the work partitioning strategy. The nature of our results indicate that employing our framework, one may sacrifice only a small portion of the runtime but can achieve very good load balancing across heterogeneous devices.

We also identified characteristics of workloads that can benefit from our mechanism. In future, we would like to study our mechanism on other emerging heterogeneous computing platforms. Similarly, it would be interesting to apply our framework to other applications which possess the required characteristics.

Later we study the graph pruning as a technique to speed-up large graph algorithms on modern parallel architectures. We applied the technique to the Single Source shortest path(SSSP) and its generic version All Pair Shortest Path(APSP) problem in graph algorithms. Our results indicate that the technique is quite useful, especially for large sparse graphs. In all the applications we studied in this paper, we needed to prune the pendant vertices. In future, we wish to study other problems that will lead to the discovery of other pruning strategies.

Bibliography

[1] "". The University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices/.

[2] V. Agarwal, F. P. D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *Proc. of ACM SC*, page 111, 10.

[3] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, J. Langou, H. Ltaief, and S. Tomov. Lu factorization for accelerator-based systems. In *Computer Systems and Applications (AICCSA), 2011 9th IEEE/ACS International Conference on*, pages 217–224. IEEE, 2011.

[4] R. J. Anderson and G. L. Miller. A Simple Randomized Parallel Algorithm for List-Ranking. *Information Processing Letters*, 33(5):269–273, 1990.

[5] M. Baboulin, J. Dongarra, and S. Tomov. Some issues in dense linear algebra for multicore and special purpose architectures. 2008.

[6] D. A. Bader, V. Agarwal, and K. Madduri. On the Design and Analysis of Irregular Algorithms on the Cell Processor: A Case Study of List Ranking. In *Proc. of IEEE IPDPS*, pages 1–10, 2007.

[7] D. S. Banerjee and K. Kothapalli. Hybrid Algorithms for List Ranking and Graph Connected Components. In *Proc. of 18th Annual International Conference on High Performance Computing (HiPC)*, 2011.

[8] D. S. Banerjee, P. Sakurikar, and K. Kothapalli. Fast, scalable parallel comparison sort on hybrid multicore architectures. In *Proceedings of the third International workshop on Accelerators and Hybrid Exascale Systems (AsHES)*, 2013.

[9] 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*, page 18. ACM, 2009.

[10] P. L. Bhatnagar, E. P. Gross, and M. Krook. A model for collision processes in gases. i. small amplitude processes in charged and neutral one-component systems. *Physical Review*, 94(3):511525, 1954.

[11] L. Chen, Z. Hu, J. Lin, and G. R. Gao. Optimizing the fast fourier transform on a multi-core architecture. In *Proc. of IEEE IPDPS*, pages 1–8, 2007.

[12] S. Chen and G. D. Doolen. Lattice boltzmann method for fluid flows. *Annual Review of Fluid Mechanics*, 30(1):329–364, 1998.

[13] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey. Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency. In *IPDPS*, pages 378–389, 2012.

[14] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey. Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 378–389, 2012.

[15] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. Introduction to algorithms, 2001.

[16] J. Dongarra. Personal communication, 2013.

[17] G. DAngelo, M. DEmidio, D. Frigioni, and V. Maurizio. A speed-up technique for distributed shortest paths computation. In *Computational Science and Its Applications-ICCSA 2011*, pages 578–593. 2011.

[18] A. Gharaibeh, L. B. Costa, E. Santos-Neto, and M. Ripeanu. On graphs, gpus, and blind dating: A workload to processor matchmaking quest. In *in Proc. of IEEE IPDPS*, 2013.

[19] J. Habich, T. Zeiser, G. Hager, and G. Wellein. Performance analysis and optimization strategies for a d3q19 lattice boltzmann kernel on nvidia gpus using cuda. *Adv. Eng. Softw.*, 42(5):266–272, May 2011.

[20] P. Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU using CUDA. In *Proc. of HiPC*, 2007.

[21] Z. He and B. Hong. Dynamically tuned push-relabel algorithm for the maximum flow problem on cpu-gpu-hybrid platforms. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10, 2010.

[22] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Proc. of IEEE PACT*, pages 78–88, 2011.

[23] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Proc. of IEEE PACT*, pages 78–88, 2011.

[24] J. Jaja. *An Introduction To Parallel Algorithms*. Addison-Wesley, 2004.

[25] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24(1):1–13, Jan. 1977.

[26] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing*, Supercomputing '96, Washington, DC, USA, 1996. IEEE Computer Society.

[27] G. J. Katz and J. T. Kider, Jr. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '08, pages 47–55. Eurographics Association, 2008.

[28] 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*, abs/1303.2171, 2013.

[29] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10. ACM, 2010.

[30] C. Lrig, R. Grosso, and T. Ertl. Implicit adaptive volume ray-casting. In *In GraphiCon 97*, pages 114–120, 1997.

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

[32] P. S. N. Leischner, V. Osipov. Gpu sample sort, 2010.

[33] B. Pattabiraman, M. M. A. Patwary, A. H. Gebremedhin, W. keng Liao, and A. N. Choudhary. Fast algorithms for the maximum clique problem on massive sparse graphs. *CoRR*, abs/1209.5818, 2012.

[34] N. Satish, M. Harris, and M. Garland. Designing efcient sorting algorithms for manycore gpus. In *Proc. of IEEE IPDPS*, 2009.

[35] D. P. Scarpazza, O. Villa, and F. Petrini. Efficient breadth-first search on the cell/be processor. *IEEE Trans. Parallel Distrib. Syst.*, 19(10):1381–1395, 2008.

[36] T. Scheuermann and J. Hensley. Efficient histogram generation using scattering on gpus. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 33–37. ACM, 2007.

[37] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan Primitives for GPU Computing. In *Proc. ACM Symp. Graphics Hardware*, pages 97–106, 2007.

[38] F. Song, A. YarKhan, and J. Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*, pages 1–11. IEEE, 2009.

[39] S. Succi. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond (Numerical Mathematics and Scientific Computation)*. Numerical mathematics and scientific computation. Oxford University Press, USA, Aug. 2001.

[40] M. Tang, J.-y. Zhao, R.-f. Tong, and D. Manocha. Gpu accelerated convex hull computation. *Computers & Graphics*, 36(5):498–506, 2012.

[41] J. Tolke and M. Krafczyk. Teraflop computing on a desktop pc with gpus for 3d cfd. *Int. J. Comput. Fluid Dyn.*, 22(7):443–456, Aug. 2008.

[42] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense liner algebra for hybrid gpu accelerated manycore systems. *Parallel Computing*, 12:10–16, Dec. 2009.

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

[44] M. Van der Hoef, M. Ye, M. van Sint Annaland, A. Andrews, S. Sundaresan, and J. Kuipers. Multiscale modeling of gas-fluidized beds. *Advances in chemical engineering*, 31:65–149, 2006.

[45] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya. A blocked all-pairs shortest-paths algorithm. *J. Exp. Algorithmics*, 8, Dec. 2003.

[46] O. Villa, D. Scarpazza, F. Petrini, and J. Peinador. Challenges in mapping graph exploration algorithms on advanced multi-core processors. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10, 2007.

[47] 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, 2010.

[48] Z. Wei and J. JaJa. Optimization of linked list prefix computations on multithreaded gpus using cuda. In *The 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2010.

[49] G. Wellein, T. Zeiser, G. Hager, and S. Donath. On the single processor performance of simple Lattice Boltzmann kernels. *Comput. Fluids*, 35(8-9):910–919, Sept. 2006.

[50] Y. Xia and V. K. Prasanna. Topologically Adaptive Parallel Breadth Frst Search on Multicore-Processors. In *in Proc. PDCS*, 2009.

[51] I. Yamazaki, T. Dong, S. Tomov, and J. Dongarra. Tridiagonalization of a symmetric dense matrix on a gpu cluster. *in the proceedings of the third International workshop on Accelerators and Hybrid Exascale Systems (AsHES)*, may 2013.