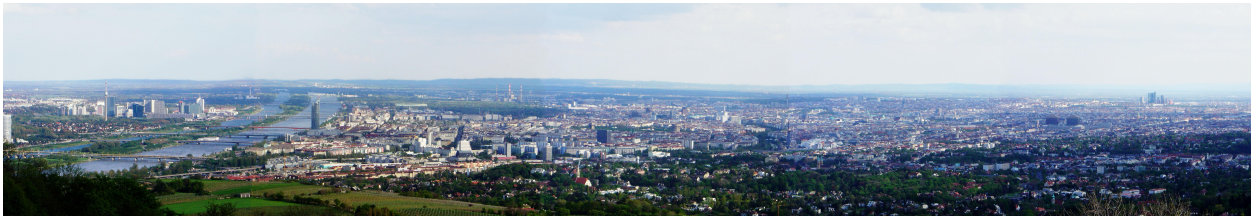
Proceedings

International Workshop on GPUs and Scientific Applications (GPUScA 2010)



Vienna, Austria **September 11, 2010**

In conjunction with International Conference on Parallel Architectures and Compilation Techniques (PACT 2010), Vienna, Austria, September 11-15, 2010.

Eduard Mehofer, Markus Schordan, Dan Quinlan, Beniamino Di Martino (Eds.)

Proceedings

International Workshop on GPUs and Scientific Applications (GPUScA 2010)

Vienna, Austria

September 11, 2010

Eduard Mehofer, Markus Schordan, Dan Quinlan, Beniamino Di Martino (Eds.)

Proceedings available as technical report.
Department of Scientific Computing,
University of Vienna.
http://www.par.univie.ac.at/publications/download/TR-10-3.pdf

Technical report TR-10-3
Department of Scientific Computing
University of Vienna
September 2010

Preface

Welcome to the International Workshop on GPUs and Scientific Applications (GPUScA 2010) in Vienna! The workshop takes place in conjunction with PACT 2010 - the annual International Conference on Parallel Architectures and Compilation Techniques. The purpose of the workshop is to bring together GPU experts with computational science experts.

GPUs are cost-effective platforms for computational intensive applications providing tremendous peak performance. However, it is a major challenge to deliver the intrinsic performance of such architectures to end applications. The workshop addresses programming approaches and key techniques to leverage the computing power of GPUs.

Based on 3 reviews per submission, 8 high-quality papers were selected for presentation and are included in the workshop proceedings. The accepted papers reflect the multidisciplinary character and the broad spectrum of the field. The technical program consists of parallel programming technology papers and application papers covering algorithmics, image processing, physical phenomena, and computational biology. The presentation of the papers is arranged in three sessions.

The session devoted to application papers with strong algorithmic aspects consists of the paper 'Improving the GPU-based collision check procedure for distributed crowd simulations' presenting an algorithm for GPU-based crowd simulations; the paper 'Fast GPU perspective grid construction and triangle tracing for exhaustive ray tracing of highly coherent rays' proposing an algorithm for tracing nuclear radiation; the paper 'Solving planted motif problem on GPU' addressing a problem from computational biology that was ported to GPUs.

The session devoted to application papers with strong domain aspects consists of the paper 'Scalability of color-based segmentation of football players over GPUs' studying the scalability of a real-time image processing application; the paper 'Fluid simulation with CUDA using the Lattice Boltzmann Method' describing the realization of a physical problem on GPUs; the paper 'A framework for GPU accelerated deformable object modeling' presenting a framework for simulating the deformation of objects.

The session devoted to papers targeting parallel programming technology consists of the paper 'ViennaCL - a high level linear algebra library for GPUs and multi-core CPUs' presenting a library which supports linear algebra routines for GPUs; the paper 'Dynamic work scheduling for GPU systems' addressing efficient scheduling techniques for GPUs.

The preliminary workshop proceedings are published as technical report TR-10-3 of the Department of Scientific Computing, University of Vienna (URL http://www.par.univie.ac.at/publications/download/TR-10-3). Extended versions of the best papers will be published after the event in the International Journal of High Performance Computing and Networks (IJHPCN) and International Journal of High Performance Computing Applications (IJHPCA) depending on the topic of the respective paper.

It is our pleasure to announce Vivek Sarkar for the keynote address, whose talk is entitled 'Towards a Portable Execution Model for Extreme Scale Multicore Systems'. An abstract of the keynote address opens the proceedings.

We would like to thank the program committee members and the reviewers for their hard work and the excellent cooperation. Also special thanks to all authors of submitted papers for their interest and their contributions to the success of the workshop. Finally, we are grateful to the PACT chairs for their support of the workshop.

Vienna, September 2010

Eduard Mehofer, Markus Schordan, Dan Quinlan, Beniamino Di Martino

Workshop Organization

Workshop Chairs

Eduard Mehofer	University of Vienna
Markus Schordan	University of Applied Sciences Technikum Wien
Dan Quinlan	Lawrence Livermore National Laboratory
Beniamino Di Martino	Second University of Naples

Program Committee

Alok Choudhary	Northwestern University, USA
Thomas Fahringer	University of Innsbruck, AUT
Michael Gerndt	Technical University Munich, GER
Beniamino Di Martino	Second University of Naples, ITA
Eduard Mehofer	University of Vienna, AUT
Michael O'Boyle	University of Edinburgh, GBR
Bernhard Scholz	University of Sydney, AUS
Markus Schordan	University of Applied Sciences Technikum Wien, AUT
Michael Schwarz	Max Planck Institute for Informatics, GER
Xipen Shen	College of William and Mary, USA
Apan Qasem	Texas State University, USA
Dan Quinlan	Lawrence Livermore National Laboratory, USA
Michael Wimmer	Vienna University of Technology, AUT
Qing Yi	University of Texas at San Antonio, USA

Additional Reviewers

Pasquale Cantiello
Tim Dawborn
S. M. Farhad
Lang Hames
Mauro Iacono
Herbert Jordan
Prabhat Kumar
Berkin Ozisikyilmaz
Heiko Schmidt
Heiko Studt
Peter Thoman
Eddy Zhang

Table of Contents

Keynote

Towards a Portable Execution Model for Extreme Scale Multicore Systems

Vivek Sarkar, E.D. Butcher Professor in Engineering, Rice University

Abstract. Computer systems anticipated in the 2015 – 2020 timeframe are referred to as Extreme Scale because they will be built using homogeneous and heterogeneous many-core processors with 100's of cores per chip. These systems pose new critical challenges for software in the areas of concurrency, energy efficiency and resiliency. Unlike previous generations of hardware evolution, this shift towards many-core computing will have a profound impact on software. These software challenges are further compounded by the need to enable parallelism in workloads and application domains that have traditionally not had to worry about multiprocessor parallelism in the past. A recent trend towards Extreme Scale systems is the use of graphics processor units (GPUs) to obtain order-of-magnitude performance improvements relative to general-purpose CPU's. Unfortunately, hybrid programming models that support multithreaded execution on CPU's in parallel with CUDA execution on GPU's prove to be too complex for use by mainstream programmers and domain experts in a portable fashion, especially when targeting platforms with multiple CPU cores and multiple GPU devices.

An execution model serves a valuable role in providing a shared conceptual view for all stakeholders in a computing platform ecosystem. Successful execution models from the past (e.g., Von Neumann, vector parallelism, SMP parallelism, Bulk-Synchronous parallelism) were built on primitives that were well matched with past device, architecture, and software technology trends, but are mismatched to future multicore systems where performance has to be driven by parallelism and constrained by energy. In this talk, we identify key primitives that we believe will be necessary for a successful execution model for future extreme scale systems with heterogeneous accelerators such as GPUs. We will discuss the portability of these execution model primitives based on their ability to support multiple programming models and their amenability to be mapped to a wide range of extreme scale hardware.

We present early experiences with these execution model primitives in the Habanero Multicore Software Research project at Rice University which targets mainstream homogeneous and heterogeneous multicore systems, and discuss future directions in the context of the NSF Expeditions project on the Center for Domain-Specific Computing (http://www.cdsc.ucla.edu/) which targets embedded systems with an initial focus on the medical imaging domain. Both projects takes a two-level approach to programming models, with a higher-level macro-dataflow model based on Intel Concurrent Collections (CnC) for parallelism-oblivious domain experts, and a lower-level task-parallel model based on the Habanero-Java and Habanero-C languages for parallelism-aware developers. We discuss language, compiler and runtime implementation challenges that must be overcome to efficiently support these primitives on future mainstream and embedded multicore systems. To address the hybrid programming challenge for domain experts, we extend CnC with CUDA steps to obtain a model called CnC-CUDA. The CnC-CUDA extensions discussed in this talk include multithreaded steps for execution on GPUs, and automatic generation of data and control flow between CPU steps and GPU steps.

Links

[1] Habanero Multicore Software Research project (`http://habanero.rice.edu`)
[2] Habanero Concurrent Collections download (`http://habanero.rice.edu/cnc`)
[3] Habanero Java download (`http://habanero.rice.edu/hj`)
[4] Overview article on "Software Challenges at Extreme Scale" (`http://www.scidacreview.org/1001/html/software.html`)

Improving the GPU-based Collision Check Procedure for Distributed Crowd Simulations*

Guillermo Vigueras, Juan M. Orduña, Miguel Lozano
Departamento de Informática
Universidad de Valencia
Spain
juan.orduna@uv.es

José M. Cecilia, José M. García
Dpto. Ingenería y Tecnología de Computadores
Universidad de Murcia
Spain
{chema, jmgarcia}@ditec.um.es

ABSTRACT

The computing capabilites of current Graphics Processor Units (GPUs) have been used by many distributed applications for performing general purpose computations. In particular, the capabilites of many-core GPUs have been used in crowd simulations not only for enhancing the crowd rendering, but also for performing collision check and even for simulating the whole crowd. Nevertheless, these applications can still significantly increase their throughput if the GPU capabilities are fully exploited.

In this paper, we propose a new algorithm for GPU-based collision check in distributed crowd simulations. Unlike other collision check algorithms in the literature, the absence of both sorting procedures and atomic operations in the proposed method significantly reduces the computing workload of the collision check procedure, while keeping the crowd simulation consistent. The performance evaluation results show that the execution time required for the proposed method is significantly lower than previous methods based on sorting, increasing the crowd simulation throughput accordingly.

1. INTRODUCTION

The computing capabilites of current Graphics Processor Units (GPUs) have been used by many distributed applications for performing general purpose computations [11]. In particular, these capabilites have been used in crowd simulations, a special case of Virtual Environments where the avatars are autonomous agents instead of user-driven entities. Each of these agent-based entities can have its own goals, knowledge and behavior [14]. The computational cost of multiagent crowd simulations exponentially increases with the number of agents in the system, requiring a scalable design that can support huge amounts of agents (of different orders of magnitude) by simply adding more hardware. A distributed system architecture has been proposed to tackle these requirements [7, 17, 16]. That architecture consists of a distributed system where some of the computing nodes contain a distributed Action Server controlling the simulation. The rest of the computers host a set of agents implemented as threads of a single process. That architecture was shown efficient enough to support simulations up to tens of thousands of complex agents with plausible graphic quality. However, this distributed scheme can be still improved by fully exploiting the potential of new many-core architectures like GPUs.

Since the processing of the collision checks submitted by agents represents the most time consuming task in the distributed action server [17], in a previous work we implemented a basic distributed server for crowd simulations using an on-board GPU [18]. That GPU-based basic implementation used the particle algorithm [10] for performing parallel collision checks. Nevertheless, crowd simulations can still significantly increase their throughput if the GPU capabilities are fully exploited. In this paper, we propose a new GPU-based algorithm to perform the collision check procedure in distributed crowd simulations. Unlike other collision check procedures in the literature, the absence of both sorting procedures and atomic operations in the proposed method significantly reduces the computing workload of the collision check procedure, while keeping the consistency of the crowd simulation. The performance evaluation results show that the execution time required for the proposed method is significantly lower than previous methods based on sorting, increasing the system throughput accordingly.

The rest of the paper is organized as follows: Section 2 describes the distributed system for crowd simulation where the proposed GPU-based algorithm would be integrated. Section 3 briefly describes the related work about parallel architectures for crowd simulation. Section 4 gives an overview of the CUDA programming model. Next, Section 5 shows the proposed algorithm, as well as other improvements of existing methods for comparison purposes. Section 6 shows the performance evaluation results for the different approaches considered. Finally, section 7 shows some conclusion remarks.

*This work has been jointly supported by the Spanish MICINN and the European Commission FEDER funds under grants Consolider-Ingenio 2010 CSD2006-00046 and TIN2009-14475-C04.

2. A DISTRIBUTED SYSTEM FOR CROWD SIMULATION

In previous works, we proposed an architecture that can simulate large crowds of autonomous agents at interactive rates [7, 17, 16]. In that architecture, the crowd system is composed of many Client Computers, that host agents implemented as threads of a Client Process, and one Action Server (AS). The AS is executed in one computer and is responsible for checking the actions (eg. collision detection) sent by agents [7]. In order to avoid server bottleneck, the simulation world was partitioned into subregions and each one assigned to one parallel AS [17]. A scheme of this architecture is shown in Figure 1. This figure shows how the 2D virtual world occupied by agents (black dots) is partitioned into three subregions, and each one managed by one parallel AS (labeled in the figure as AS_x). Each AS is hosted by a different computer. Agents are execution threads of a Client Process (labeled in the figure as $Client_x$) that is hosted on one Client Computer. The computers hosting client and server processes are interconnected. Each AS process hosts a copy of the Semantic Database (SDB) that contains information about the simulated world. However, each AS exclusively manages the part of the database representing its region. In order to guarantee the consistency of the actions near the border of the different regions (see $agent_k$ in figure 1), the ASs can collect information about the surrounding regions by querying the servers managing the adjacent regions. Additionally, the associated Clients are notified about the changes produced by the agents located near the adjacent regions by the ASs managing those regions.

Each action requested by an agent requires a collision test in the corresponding AS. This test is computed based on the Area of Interest (AOI) of the agent. If the AOI of the considered agent does not intersect with the region border (eg. $agent_1$ in Figure 1), the corresponding AS updates the semantic database with the new location and notifies all the local CPs about that change. If, on the contrary, the AOI of the considered agent intersects with the region border (eg. $agent_k$ in Figure 1), then the adjacent servers are queried. Only if all the servers answer positively the requested action is allowed, and the semantic database is updated. In this case the queried adjacent servers are also notified about the change, in order to guarantee the consistency among all the SDB copies. The architecture shown in Figure 1 allows to simulate large crowds of autonomous agents providing a good scalability. However, this architecture can also benefit from the GPU capabilities for simultaneously checking the collision requests received from agents [18].

3. RELATED WORK

Some proposals have been made last year for exploiting the capabilities of multi-core and many-core architectures in crowd simulations. In this sense, a new approach has been presented for the CellBe processor to distribute the load among the processing elements [13]. Other work uses graphics hardware to simulate crowds of thousands of individuals using models designed for gaseous phenomena [2]. Recently, some authors have started to use GPU in an animation context (particle engine) [5], and there are also some proposals for running simple stochastic agent simulations on GPUs [8, 12]. However, these proposals are not suitable

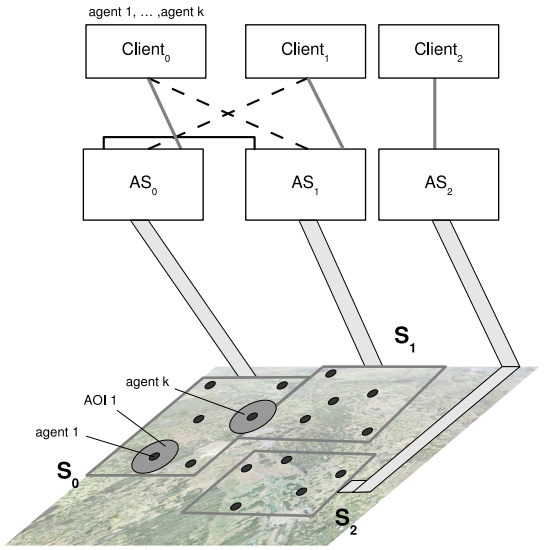


Figure 1: General scheme of the distributed architecture for crowd simulation

to simulate complex agents, including a cognitive model, at interactive rates.

Other proposals show efficient GPU implementations of particle simulations [10] or parallel global pathfinding [1] using the CUDA programming environment. These works propose efficient models for a single GPU. On the contrary, this paper proposes a distributed implementation that can use as many GPUs as necessary, each one hosted by an Action Server, to perform the collision check process. In order to solve the GPU-based collision check problem, different implementations have been proposed [6, 20], based on hierarchical data structures and sorting. However, the computational cost of these proposals were shown efficient to solve problems like ray tracing but not for agent based simulation. Finally, another work proposes a GPU implementation for searching the k nearest neighbors in order to solve the collision check problem [4]. Nevertheless, this work does not assess the scalability of the method with the number of entities considered in the neighbors search.

4. CUDA PROGRAMMING MODEL

The Compute Unified Device Architecture (CUDA) programming model for GPU architectures covers both hardware and software features for performing computations on the GPU as a data-parallel computing device without the need of mapping them to a graphics API [9]. The hardware interface of CUDA consists of a parallel SIMD architecture, where thousands of threads run in parallel. These cores are organized as a given number of multiprocessors (SMs), each one having a set of 32-bit registers, constants and texture caches, and 16 KB of on-chip shared memory as fast as local registers (one cycle latency). At any given cycle, each core executes the same instruction on different data (SIMD), and communication among multiprocessors is performed through global memory.

CUDA consists of a set of C language library functions that the programmer can use to specify the structure of a CUDA

program. A CUDA program consists of two subprograms: The CPU (or *host*) subprogram and the GPU (or *device*) subprogram. The former prepares the execution on the GPU, moving data from main memory to the GPU memory, setting up all the parameters involved in the execution, and launching the code that is executed on the GPU by each thread.

The GPU subprogram consists of a set of kernels. Kernel execution is decomposed into blocks that run logically in parallel (they are physically executed only if there are resources available on the GPU). A block is a group of threads assembled by the developer which is mapped to a single multiprocessor. This group of threads can share 16 KB of memory and they can synchronize among them through barrier primitives. However, the communication among threads of different blocks is only performed through global memory, and the traditional way to synchronize them is terminating a kernel launch. All the threads are internally grouped into *warps*. A warp is a collection of threads that can run concurrently (with no time sharing) on all of the multiprocessors. The developer can determine the number of threads to be executed (up to a limit intrinsic to CUDA), but if there are more threads than the warp size, then they are time-shared on the actual hardware resources. Any thread can have access to all the GPU memory in the CUDA programming model, but there is a performance boost when threads access data located in shared memory, which is explicitly managed. Therefore, large data structures must be stored in the global memory and often-used data structures must be stored in the shared memory, in order to efficiently use the GPU's computational resources. This issue is particularly important in the collision check algorithms for crowd simulations.

5. COLLISION CHECK ALGORITHMS

The collision detection problem has been addressed in many areas like Computer Graphics, Computer Animation, Agent based Simulation, etc. A collision among agents within a crowd simulation occurs when the volume occupied by one agent intersects with the one occupied by other agent (this problem can be reduced to a two dimensional environment considering the 2D shape that represents each agent instead of its volume). Usually, the simulated scenario is divided by means of a n-dimensional grid in order to efficiently solve the collision check problem. In this way, only the agents contained in a given grid cell and the agents contained in the neighboring cells are checked. A naive GPU implementation of this grid (called *collision grid*) consists of defining a static array and assigning each grid cell to each position of this array. The mapping of agents to grid cells is performed by a spatial hashing method, depending on the cell size and the position of agents. Since many agents can fall within the same cell and GPU threads can simultaneously update the same memory address, atomic operations are needed in order to keep consistency [9]. However, atomic operations cause a performance penalty, increasing the execution time of the collision check procedure. Due to this penalty, other approaches based on sorting have been shown to obtain better performance than static approaches based on atomic operations [10, 3].

In a previous work, we implemented a collision check procedure for crowd simulations using an on-board GPU [18]. This algorithm consists of five steps, each one implemented as a GPU kernel. Figure 2 shows a scheme of the five steps and the data structures involved in this algorithm, as well as the input and output of each step. The upper part of Figure 2 shows a snapshot of a 2D grid, composed of sixteen cells containing six agents. In the lower part, this Figure shows the values of the data structures corresponding to that snapshot for each step of the algorithm. First, the hashing of the agents within the *collision grid* is performed, determining on which cell is located each agent (more than one agent can be assigned to the same cell). The result is an array containing the cell identifier assigned to each agent. Second, the sorting of the previous array based on the cell identifier (in increasing order) is performed, in order to allow the GPU threads to efficiently access to this grid. Third, the data structure containing the agents positions are also sorted to match the same cell order established in the second step. As a result, all the agents located in the same cell are in adjacent positions of the data structure. Fourth, a data structure representing the collision grid is computed. This structure allows a fast access to the agents located in each grid cell, and it consists of a sparse array. Finally, the last kernel is the collision check algorithm. This algorithm finds in which cell is located each agent and which other agents are located in the same or the neighboring cells (that is, the possible collisions are checked). In order to perform this task, it uses the data structure computed in the fourth step. The result is an array whose elements are a collision flag for each agent. We have denoted this algorithm as the *Baseline* implementation. Since this implementation is the basic translation of a GPU-based method for collision tests [10, 3], we have developed an improved implementation of that algorithm as a reference for comparison purposes.

5.1 Improved Baseline Algorithm

The *Baseline* algorithm is composed by five kernels. In order to improve the baseline algorithm, the first step is to determine which kernels are the most time consuming. We have measured the percentage of the global execution time consumed by each kernel for a simulation of one million agents. These measurements are shown in Figure 3. This figure shows that the most time consuming kernel is the one performing the collision check, consuming 63% of the total time. That is, this kernel does not take advantage of the GPU memory hierarchy in the *Baseline* version, since it only accesses the global memory.

Each agent checks its neighborhood in the collision check kernel. This data locality can be exploited by using the on-chip GPU memories. Concretely, the input arrays of the kernel performing the collision check can be bound to the texture memory. Hence, neighbor cells are cached and they can be fetched from the texture memory instead of the device memory, increasing the memory bandwidth. We have considered this as the first improvement of the baseline algorithm, and we have denoted it as the *texture memory* optimization. On other hand, data locality can be exploited by using the shared memory along with a tiling technique [19]. We define tiles within the collision grid in such a way that collisions can be independently checked by each GPU block, avoiding inter-block synchronization. We propose the ordering of the collision grid cells in global memory based on

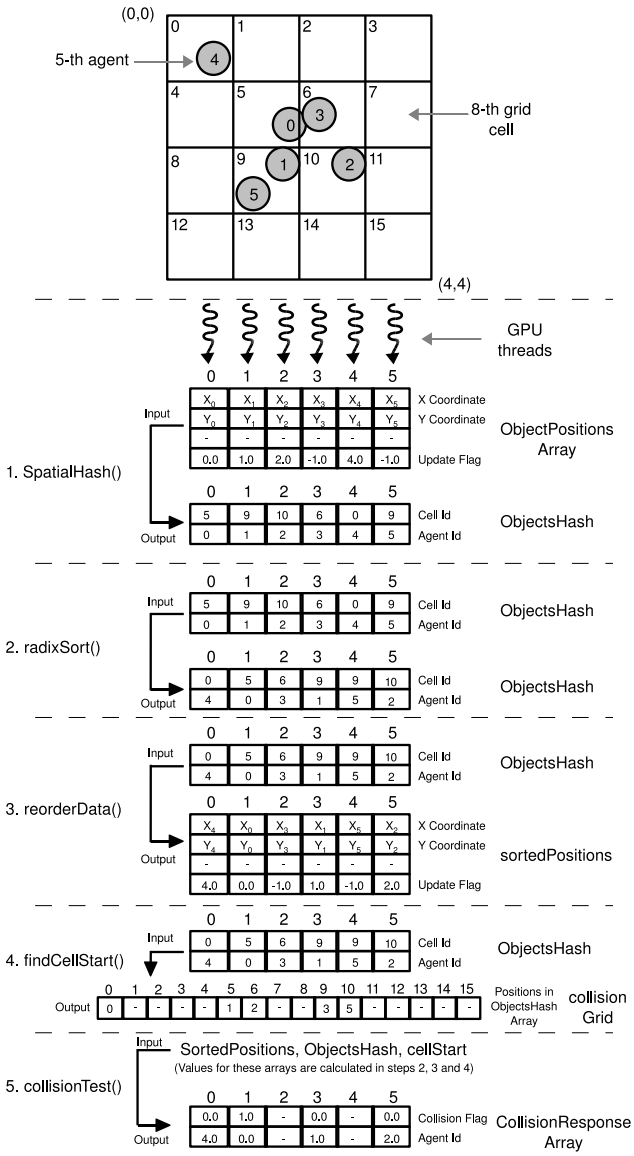


Figure 2: Baseline algorithm for GPU-based collision check

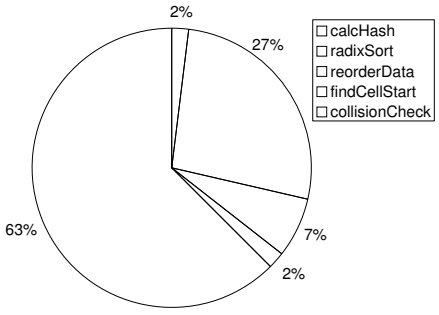


Figure 3: Percentage of execution time required by the kernels for the baseline version.

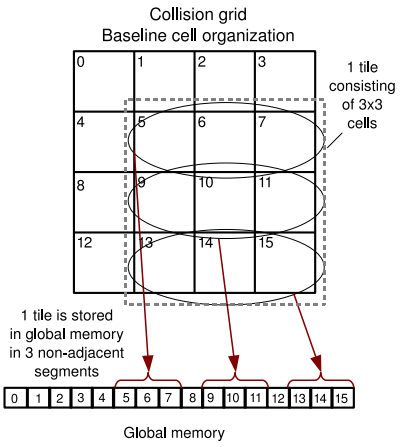


Figure 4: Grid mapping to global memory in the baseline version

the tile organization. In this way, all threads in a GPU block collaborate in loading the assigned tile from global memory to shared memory, obtaining a coalesced access and reducing the number of accesses to device memory. This memory layout also avoids bank conflicts in the access to shared memory. In order to illustrate this improvement, Figure 4 shows the memory access pattern of the baseline algorithm, while Figure 5 shows the memory access pattern of the improved baseline algorithm. Both figures show a collision grid with sixteen cells. Figure 4 shows how a given tile consisting of 3x3 cells (from cell 5 to cell 15 except cells 8 and 12) is stored in global memory. It can be seen that the neighboring cells are stored in non-adjacent memory segments (cells 8 and 12 are interleaved within the tile segments) preventing coalesced accesses to global memory.

A tile in the improved algorithm consists of 3x3 cells, as in the case of the baseline algorithm. Figure 5 shows how the improved algorithm replicates those cells that are in the border of a tile. In this figure, the numbers in the middle of each cell denotes the cell number in the collision grid, while the small numbers in the corners of each cell denote the replicas of that cell in each tile. For example, the cell number 3 is replicated as cell 4 in the first tile, cell 12 in the second tile, cell 19 in the third tile, and cell 27 in the fourth tile. The advantage of this data replication consists of having all the cells belonging to a given tile linearly ordered in the same global memory segment, as shown in the lower part of Figure 5. Therefore, all threads in a warp (half-warp) can linearly access to the same global memory segment and load the data into shared memory obtaining a coalesced access. We have denoted this improved organization along with the use of shared memory as the *shared memory* optimization. A key parameter in the shared memory optimization is the tile size, since it determines the number of threads in each block. In turn, this number of threads must be an entire multiple of the warp size in order to obtain a good performance. The tuning of the tile size should be experimentally performed. Concretely, a tile size of 16x16 (256 threads per block) has provided the best result for populations ranging from 10.000 to 1.000.000 agents.

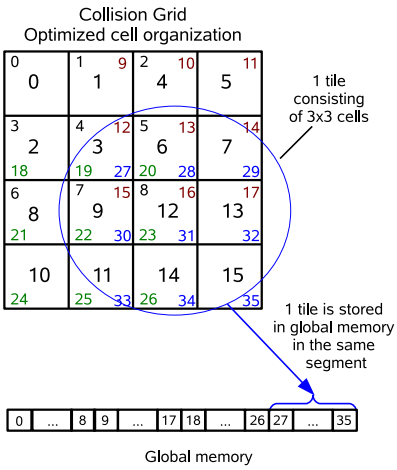


Figure 5: Grid mapping to global memory in the improved version

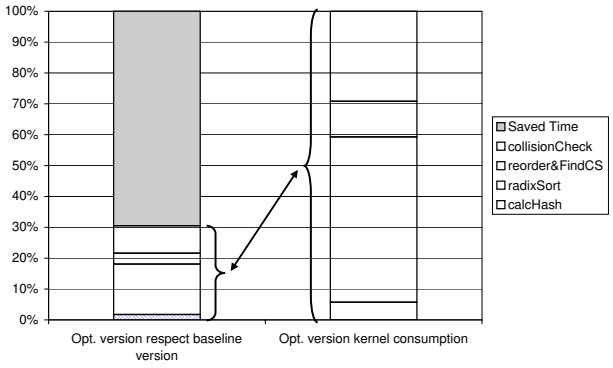


Figure 6: Percentage of execution times required by the kernels in the baseline optimized version.

Besides the collision check kernel, the kernel performing the radix sort is the second most time consuming kernel in the baseline algorithm (see Figure 3). In order to improve the performance, the radix sort procedure used in the baseline algorithm can be replaced by the fastest published version of this sorting algorithm [15]. Finally, although the execution time for the rest of the kernels are less significant than the previous ones, some optimizations can be performed on them. The kernels corresponding to the third and fourth steps in the *Baseline* algorithm can be merged into a single one, as there are no global synchronization requirements between them. Therefore, the cost of synchronization can be saved. Furthermore, the shared memory can be used by the fourth kernel, taking advantage of the data locality and improving the global memory bandwidth.

In order to show the improvements achieved by the optimized version of the Baseline algorithm, Figure 6 shows the impact of the optimizations in terms of percentages of the execution time (being 100% the total execution time of the Baseline algorithm on the left bar). This bar shows that the effect of the optimizations represents a reduction of a 70% in the global execution time respect to the Baseline version. The right bar in Figure 6 zooms in the results obtained for the improved version. In this version, the most time consuming kernel is the *radixSort*, with a 54% of the global execution time for the optimized version. For this reason, we propose a new algorithm to perform the collision check that is not based on sorting.

5.2 A New GPU-Based Algorithm for Collision Check

We propose an algorithm that avoids the sorting step in the collision check procedure. In order to achieve this goal, we use a static grid. Nevertheless, if many agents fall within the same grid cell and they try to write into the same memory address, atomic operations are needed. In order to avoid the performance penalty caused by atomic operations, we propose a different approach in which the size of each grid cell is fixed in such a way that the simulation consistency is guaranteed. Concretely, the consistency is guaranteed if

$$\sqrt{L^2 + L^2} = D = 2R \quad (1)$$

where L is the size of the side of a grid cell, D is the diagonal of a grid cell and R is the radius of the agents. When the distance between two agents is less or equal to twice the agent radius ($2R$) a collision occurs. For that reason the condition in Equation 1 establishes that all the agents falling in the same cell will collide since the maximum distance within a cell is the diagonal of the cell (i.e. $D = 2R$). In this way the condition in Equation 1 implicitly performs the collision detection for agents trying to move to the same cell. In that situation the consistency can be guaranteed by allowing the movement of one agent and forbidding the rest of the movements. It must be noticed that the selection of the agent to perform the movement can be done in a non-deterministic fashion since agent-based simulations evolve in this way.

As a result of using the condition in Equation 1 to define the cell size, more neighbor cells will have to be queried during the collision check. Since the side of a cell can be shorter than $2R$, not only the closest neighbor cells must be queried but also those cells that are one cell distant. We denote this set of cells as *extended neighbor cells*. Nevertheless, in spite of the higher number of neighbor cells accessed, the performance can be improved by loading these cells from global memory only once and store them on shared memory.

Using the consistency condition (equation 1), we have defined a new collision check algorithm consisting of four steps, each one containing one GPU kernel call. In this new algorithm there is an array (denoted as *CollisionResponseArray*) containing a pair *(collision flag, agent identifier)* in each position. Another array called *ObjectPositionsArray* contains the agents positions, and the array *collisionGrid* contains in each position three elements. The first element indicates the current step of the simulation. The second element stores an agent id indicating which is the target cell for that agent. The third element stores an agent id indicating which is the source cell for that agent. Agents positions are copied by the CPU onto device memory and then the collision check test is launched. Once the test is finished the result is returned back to the CPU by copying the *CollisionResponseArray*. The actions performed in each step of the new algorithm

are illustrated in Figure 7. This figure shows an example of the whole process, including the data structures involved as both input and output of each step. The upper part of this figure shows a snapshot of a 2D grid, composed of sixteen cells containing four agents at given locations. In the lower part, this figure shows the data structures with the values corresponding to that snapshot for each step of the algorithm described above. The actions performed in each step are the following ones:

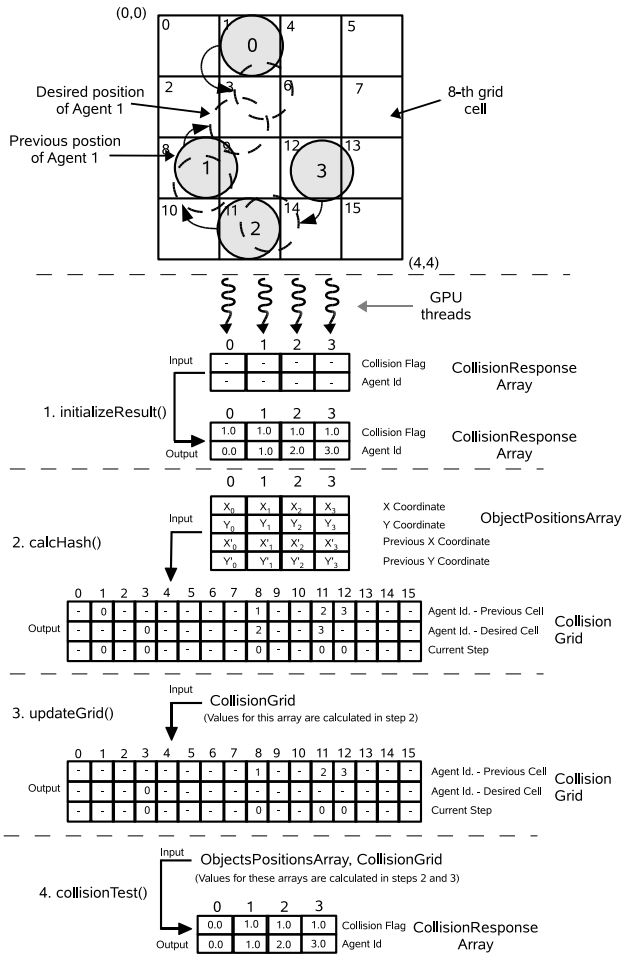


Figure 7: New algorithm for collisions check on the GPU

1. In the first step, the *collisionResponse* array is initialized indicating that there are collisions for all agents (see Figure 7). This initialization is necessary because one agent can overwrite other agent when falling in the same cell. Overwritten agents can detect the collision by means of this initialization step.

2. In the second step, the hashing to determine the target and the source cell for each agent position stored in *ObjectPositionsArray* is performed. Each thread writes a *step identifier* and the agent identifier in both the source and target cells. Since agents move at the same time in a simulation cycle of our tests, all the movements in the same cycle share a common *step identifier*. This identifier allows to determine whether the information within a cell is correct or it contains obsolete data. We use this *step identifier* to avoid using the function *cudaMemset( )*. The execution time of this function significantly increases the global execution time, specially when the size of the array to be cleared grows. The hashing performed in this step by the *calcHash* kernel is shown in Figure 7. Since cell 1 is the previous one for Agent 0 and it wants to move to cell 3, Agent 0 writes its identifier in these cells in the corresponding slot. Agent 2 moving from cell 11 to cell 8 and Agent 3 moving from cell 12 to cell 11, write their identifiers in the corresponding slots in these cells. Also Agent 1 writes its identifier in the proper slot of cell 8 (the source cell of Agent 1) but the value for the target cell (cell 3) is overwritten with the value stored by Agent 0 when the kernel *calcHash* finishes. All agents share the *step identifier* 0, since this is the first movement of each agent.

3. The third step of the new algorithm consists of agents detecting whether their desired movements are possible or not. If the desired movement of an agent was overwritten in the previous kernel or generates a collision, it means that the desired position is not possible. In this case, the collision grid is updated in the following way. Agents which desired movement was finally written, clean their identifier from their source cell. However if the movement of an agent is not possible it checks whether its source cell is the target cell of other agent. In such case the overwritten agent notifies that the desired movement is not possible. It must be noticed that restoring the previous position cannot lead to an inconsistent situation, since the initial scenario is collision free (i.e. position restore is possible), and for each cycle the agents positions are updated keeping the consistency. In Figure 7, Agent 0 cleans its identifier from its source position, cell 1. On the other hand, Agent 1 notifies to Agent 2 that its desired movement to cell 8 is not possible. Also Agent 2 notifies to Agent 3 that the desired position of the latter agent generates a collision.

4. Finally, the collision check is performed in the fourth step. For each grid cell, if the agent identifier stored in that cell is written in the *Desired Cell* slot then its *extended neighbor cells* are queried to detect a collision. If no collision is detected, then the collision flag in *collisionResponse* array is set to 0, indicating that there is no collision. If the movement is the previous one, then the collision flag is not overwritten, since the desired position generates a collision. Figure 7 shows that the collision for agent 1 is detected. The collision for agent 2 and agent 3 are also detected, since they are notified about it.

The algorithm described above performs global synchronization through finishing the second kernel launch. In this way, in the third kernel the overwritten agents are restored to their previous positions and the consistency of the simulation is kept. We have implemented a version of this algorithm using atomic operations for comparison purposes. This new version consists of merging the second and third steps in a single kernel. In order to merge these two steps,

atomic operations are needed (the global synchronization achieved through the second kernel termination should be performed by using atomic operations). However, the advantage of saving one kernel launch at the cost of using atomic operations should be analyzed.

6. PERFORMANCE EVALUATION

This section shows the performance evaluation of the GPU algorithms for collision check described in section 5. Our performance tests are based on different configurations of the simulated scenario, varying the number of agents, in order to evaluate the scalability of each algorithm version. We use random agent movements for evaluation purposes. Concretely, one hundred random movements are computed per agent, using the agent identifiers as the seed for the random generation, in order to obtain reproducible results. The execution times reported below are the aggregated time obtained for all the movements performed by all the agents considered for each simulation. Since the considered algorithm should scale up with the physical parallelism available on the GPU, we have considered different NVIDIA Tesla GPUs: the Tesla C870 (16 SMs) and Tesla C1060 (30 SMs).

Figure 8 and Figure 9 shows the overall execution time for the different collision check implementations on different graphic cards. These figures show on the X-axis the number of agents considered for the simulations. The Y-axis shows the aggregated execution time obtained for each collision check method. Figure 8 shows the results for the Tesla C870 platform. The new version using atomic operations has not been tested for this platform, since it does not support this kind of operations. As it could be expected, the greatest differences arise for the largest population size, that is, one million agents. We use the texture memory to decrease the use of the device memory in the first optimization, obtaining 50% of reduction in the execution time respect to the Baseline version. In the second optimization, the shared memory is used along with the new organization of the collision grid in global memory, in such a way that a coalesced access to device memory is guaranteed. This optimization obtains 70% of reduction in the execution time with respect to the Baseline version. Nevertheless, the proposed technique achieves the best results, obtaining 85% of reduction in the execution time.

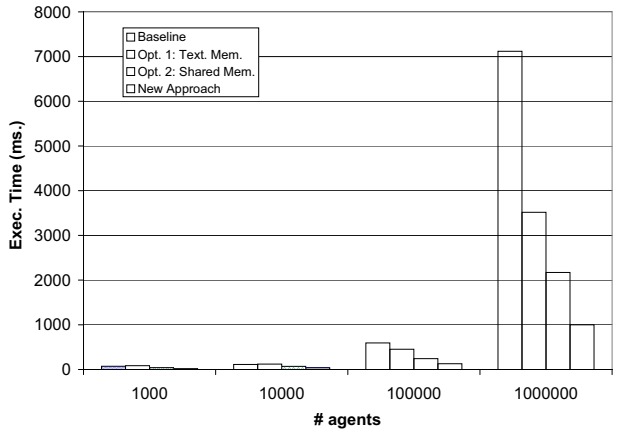


Figure 8: Execution times on Tesla C870 card

Figure 9 shows the execution times obtained for the Tesla C1060 card. In this case, the effects of the texture memory optimization hardly arise. The reason is that for this card the global memory access algorithm has been improved respect to the C870 platform [9], allowing to obtain more coalesced accesses. Therefore, the baseline algorithm requires much shorter execution times than for the case of the C870 card. The optimization that uses shared memory allows a decrease in the execution time of 53% with respect to the baseline version for a crowd size of one million agents. Nevertheless, the proposed algorithm achieves the best execution times, with a reduction of 65% when using atomic operations and around 75% without using atomic operations. If Figure 8 and Figure 9 are compared, then it can be seen that the execution times are inversely related to the number of SMs available on the cards.

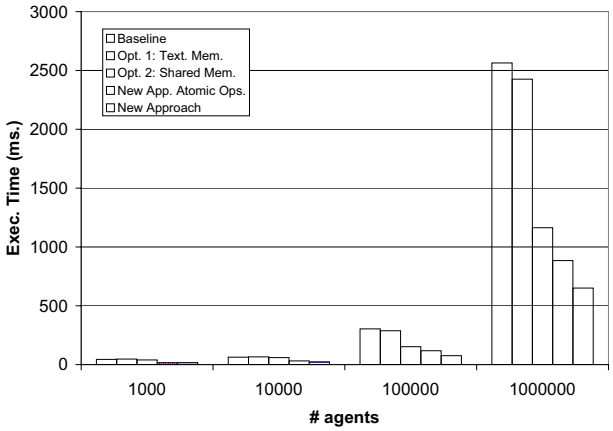


Figure 9: Execution times on Tesla C1060 card

In order to show that these execution times are directly related to the workload generated by each method, we have measured the throughput of the different versions in terms of number of collisions checked per second. Figures 10 and 11 show the collisions check rates obtained when increasing the number of agents for both the Tesla C870 and C1060 cards. Figure 10 and Figure 11 show that the proposed method without atomic operations performs the highest numbers of collision checks per second for all the population sizes. These figures also show that the collisions check rate performed by the proposed method significantly increases with the number of available SMs on the GPU, assessing the scalability of this method.

7. CONCLUSIONS

In this paper, we have proposed a new algorithm for GPU-based collision check in distributed crowd simulations. Unlike other collision check algorithms in the literature, the absence of both sorting procedures and atomic operations in the proposed method significantly reduces the computing workload of the collision check procedure while keeping the consistency of the crowd simulation. The performance evaluation results show that the execution times required for the proposed method are significantly lower than the ones of the methods used for comparison purposes, since the latter ones are based on sorting. Also, the results show that the number of collision checks per second achieved by the proposed method are the highest ones, showing that

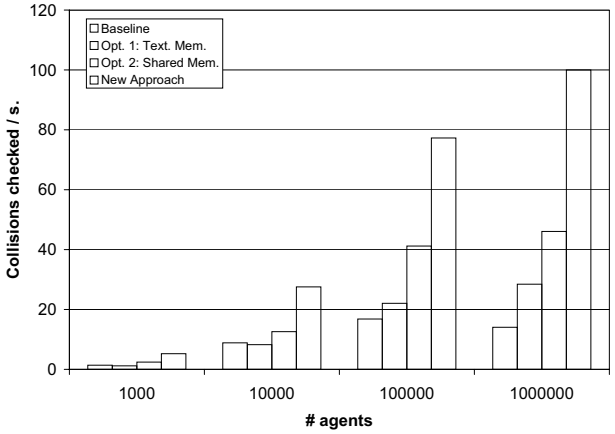


Figure 10: Collisions rate on Tesla C870 card

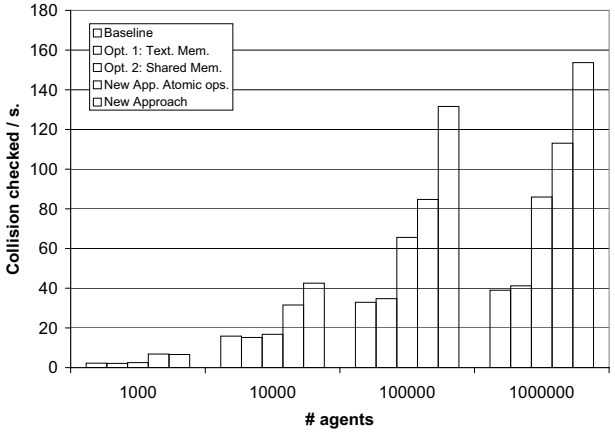


Figure 11: Collisions rate on Tesla C1060 card

the proposed method allows a higher throughput. Finally, the performance evaluation results show that the proposed method properly scales up with the number of multiprocessors available in the GPU.

8. REFERENCES

[1] A. Bleiweiss. Gpu accelerated pathfinding. In *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 65–74, 2008.

[2] N. Courty and S. R. Musse. Simulation of large crowds in emergency situations including gaseous phenomena. In *CGI '05: Proceedings of the Computer Graphics International 2005*, pages 206–212, 2005.

[3] U. Erra, B. Frola, V. Scarano, and I. Couzin. An efficient gpu implementation for large scale individual-based simulation of collective behavior. In *Proceedings of HiBi 2009*, pages 51–58, Oct. 2009.

[4] V. Garcia, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using gpu. In *CVPR Workshop on Computer Vision on GPU*, Anchorage, Alaska, USA, June 2008.

[5] L. Latta. Building a million particle system. In *In Proc. of Game Developers Conference(GDC-04)*, 2004.

[6] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh construction on gpus. *Comput. Graph. Forum*, 28(2):375–384, 2009.

[7] M. Lozano, P. Morillo, J. M. Orduña, V. Cavero, and G. Vigueras. A new system architecture for crowd simulation. *J. Netw. Comput. Appl.*, 32(2):474–482, 2009.

[8] M. Lysenko and R. M. D'Souza. A framework for megascale agent based model simulations on graphics processing units. *Journal of Artificial Societies and Social Simulation*, 11(4):10, 2008.

[9] NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. 2008.

[10] NVIDIA Corporation. *Particles Example. NVIDIA CUDA SDK*, 2008. Ver. 2.1.

[11] Owens, D. John, Luebke, David, Govindaraju, Naga, Harris, Mark, Kruger, Jens, Lefohn, E. Aaron, Purcell, and J. Timothy. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[12] K. S. Perumalla and B. G. Aaby. Data parallel execution challenges and runtime performance of agent simulations on gpus. In *SpringSim '08: Proceedings of the 2008 Spring simulation multiconference*, pages 116–123, New York, NY, USA, 2008. ACM.

[13] C. Reynolds. Big fast crowds on ps3. In *Proceedings of the ACM SIGGRAPH symposium on Videogames*, pages 113–121, New York, NY, USA, 2006. ACM.

[14] C. W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34, New York, NY, USA, 1987. ACM.

[15] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In *Proceedings of IEEE IPDPS '09*, pages 1–10, 2009.

[16] G. Vigueras, M. Lozano, J. M. Orduña, and F. Grimaldo. A comparative study of partitioning methods for crowd simulations. *Journal of Applied Soft Computing*, 10(1):225 – 235, 2010.

[17] G. Vigueras, M. Lozano, C. Perez, and J. Orduña. A scalable architecture for crowd simulation: Implementing a parallel action server. In *Proceedings of the 37th International Conference on Parallel Processing (ICPP-08)*, pages 430–437, Sept. 2008.

[18] G. Vigueras, J. Orduña, and M. Lozano. *Advances in Practical Applications of Agents and Multiagent Systems*, chapter A GPU-Based Multi-Agent System for Real-Time Simulations, pages 15 – 25. Springer, April 2010.

[19] C. Xu, S. R. Kirk, and S. Jenkins. Tiling for performance tuning on different models of gpus. In *Proceedings of ISISE '09 : Int. Symp. on Information Science and Engineering*, 2009.

[20] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, 27(5):1–11, 2008.

Fast GPU perspective grid construction and triangle tracing for Exhaustive Ray Tracing of Highly Coherent Rays

Lancelot Perrotte CEA-LIST
18 Route du Panorama
92265 Fontenay-aux-Roses, France
lancelot.perrotte@cea.fr

Guillaume Saupin CEA-LIST
18 Route du Panorama
92265 Fontenay-aux-Roses, France
guillaume.saupin@cea.fr

ABSTRACT

In this article, we address the problem of computing, storing and sorting, at an interactive rate, all of the intersections between millions of triangles (a 3D scene) and millions of rays starting from the same point. This paper focuses on the fast GPU construction of a grid in projective space referencing the triangles of a 3D scene. It introduces a fast GPU algorithm used to build a grid of the rays constituting the scene, in the same projective space. This ray-based grid is computed during the initialization of the scene, which allows us to achieve higher performance, and to construct the triangle-based grid in distinct passes for very large scenes, without having to manage memory transfers between CPU and GPU. This algorithm works the same way for both static and dynamic scenes, allowing us to achieve interactive processing of complex and dynamic scenes.

These optimizations are used to speed up the geometrical computations used in the nuclear field to evaluate the impact of radiative sources on an operator. These geometrical computations are similar to those of traditional ray tracing, except that only highly coherent rays are thrown in our application, and that we are looking for all intersections along each ray.

1. INTRODUCTION

Preparing interventions in nuclear field notably implies to evaluate the impact of radiative sources on operators. Lots of computation codes exist to simulate the propagation of radiations, but most of them operate offline. Besides, decreasing computation times in a noticeable way brings interactivity to the user, allowing him to interact with the scene and to intuitively study more appropriate scenarios. For radiation protection purposes, people often take advantage of simplified methods and algorithms, such as the straight line attenuation method with build-up factors [2], which, in most cases, gives results of the same order of magnitude as those produced with more exact methods. In this method, the radiative sources are represented as groups of punctual sources. To compute the radiations received at a given position, rays are traced between this point and the sources. For each ray, all of the intersections with the objects of the scene must be found. Since the treated scenes can have millions of triangles, a very high number of intersections will have to be stored. Even if additional computations are necessary before getting a final result, the bottleneck of the overall simulation always remains in these ray tracing requests.

Two main differences appear between our approach and traditional ray tracing. First of all, this approach does not require the management of secondary (and incoherent) rays. On the other hand, however, the primary rays of the approach are similar to those of ray tracing: coherent rays, all starting from the same point. The second main difference between the two approaches is that all intersections along each ray have to be stored. This requirement, absent of usual ray tracing, is very demanding, as explained later. Moreover, these intersections also have to be sorted, in order to compute depths of material intersected along each ray. These depths will be used as inputs to compute the radiations received at one point.

2. BACKGROUND

2.1 Ray Tracing

The problem of computing intersections between coherent rays and triangles has been investigated a lot. Real-time ray tracing, achieved during the last decade, is greatly due to the use of algorithms performing extremely well for coherent rays, like the introduction of ray packets[30]. Coupled with fully optimized kd-trees [26], algorithms specially optimized for coherent rays have brought impressive performances.

For several years, the BVH has become the best solution for most coherent rays [31], like primary rays [5], because it enables the traversal of very large packets. But in the extreme case where all the rays start from the same point, kd-tree or BVH are not the best structures anymore. [13] defined a grid in perspective space (see Figure 2) that turns out to be the most appropriate structure for primary rays. The special shape of this grid allows a great reduction of the traversal time for each ray, compared with the ones you can get with kd-trees or BVHs.

To deal with dynamic scenes, the acceleration structures have to be updated for each frame of the scene. It is possible to only update some nodes of the tree for BVHs [21], but it does not work for all kinds of dynamic scenes, and can lead

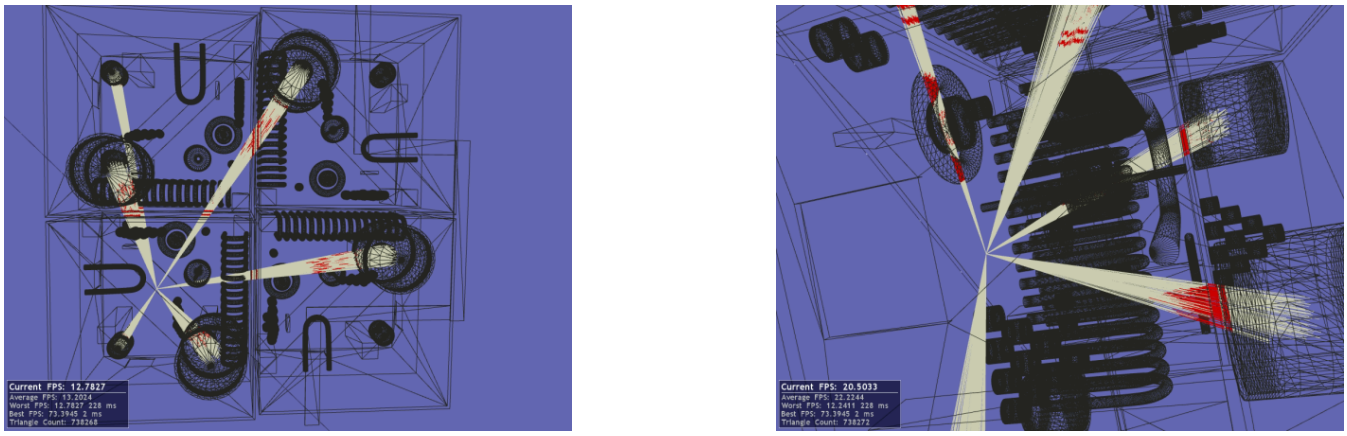


Figure 1: nuclearCase scene (738K triangles) from two different points of view. 600 000 rays are traced, for 6 millions of intersections sorted, in 250 ms on a NVIDIA GTX 295. Rays are colored in white and become red when meeting an object.

to trees of very bad quality after some frames. Interesting attempts exist to update the BVH, and asynchronously rebuild another BVH concurrently to use the new one after some frames [15].

In terms of construction times, great performances have been achieved for BVHs [20] and kd-trees [33], but they are still clearly outperformed by those obtained for grids [18]. The major drawback of the perspective grid is that it has to be constructed for each frame, for both static and dynamic scenes. But thanks to the obtained construction times, this grid still outperforms other approaches for primary rays [13]. Then, this drawback can become an advantage, since the dynamic and static scenes are treated the same way.

2.2 Rasterization

As noted in [13], the perspective grid has many similarities to a Z-buffer. In our case, ray coordinates are known in advance, as in traditional rendering. Therefore, it is natural to try to use the works of the rendering community to solve our problem. Most of the time, to render a 3D scene, triangles are passed through the graphic pipeline, and the first intersection along each ray (here a pixel of the screen) is kept thanks to the Z-buffer [8]. If very high framerates can be obtained with rasterization techniques, they are not applicable to the storing of all of the intersections for each pixel. As a matter of fact, in the rasterization approach, the amount of memory available for each pixel is constant, and limited, so that other ideas have to be found to solve our problem.

In fact, the problem of taking into account every intersection has also received much attention, since it can be essential to solve aliasing problems, or to render transparency effects. The first way to solve this problem is to modify the hardware, and hence modify the storing of the results: the first proposition of such a system was the A-buffer [6], in which all fragments were stored as linked lists. Other solutions have been proposed more recently ([32], [17]), and an implementation of the F-buffer [23] has been made available [12], but was restricted to ATI's graphics hardware.

Other solutions have to be found, since hardware modifications are not an option in this case. The most intuitive may be depth-peeling [10]: In each rasterization pass, the first intersection is stored. Hence, a first pass can be executed, to store the results, and then a new pass, adding a depth test taking into account the results of the previous pass. By repeating this process n times, the n first intersections can be stored for each ray. But, since all intersections have to be stored, the maximum number of intersections for one ray can be very high. The number of passes needed would be equal to the maximum number of intersections for a ray of the scene. This would lead to a number of passes very difficult to predict, and potentially very high.

The same problem exists with the k-buffer [3], which allows to store k intersections by pixel. This technique begins to be less efficient for $k = 8$, which would there again force us to run multiple passes. In order to treat all intersections, the k-buffer technique performs blending operations between fragments, which cannot be used in our case. Moreover, the k-buffer suffers from read-modify-write (RMW) hazards, and the solutions found strongly decrease performance level. These RMW hazards do not exist anymore with the depth peeling improvement proposed by Liu [22]: as for k-buffer, multiple render targets (MRT) buffers are used to store up to 32 intersections for each pixel. But with this solution, intersections too close to each other can be missed. Such a limitation cannot be accepted in our application case.

To sum up, none of the existing techniques completely solves our problem. An other problem comes from the fact that the rasterizer can only be used for regularly spaced rays. In our application, all rays share the same starting point, but the spacing between each other can be completely irregular. Irregular rasterizers have already been proposed ([16], [1]), but they also require a modification of the hardware.

3. ALGORITHM PRESENTATION

The structure that is used here is the perspective grid, since it is the best structure for rays that share the same starting point and since its construction time can be really low. Since all of the intersections have to be found along each ray, the 2D version of this perspective grid has been implemented. Besides, high-level GPU programming languages, such as CUDA[24] and OpenCL[25], now really ease the programming on GPU. It is now simple to perform efficient sort ([27], [29]), scan [28] and stream compaction [4]

operations. Thanks to that, the construction of the grid and the computing of all of the intersections can be done on GPU.

The GPU grid construction algorithm we present here has strong similarities with the ones proposed by [18] and [14]. But our implementation is optimized for the computation of every intersection, and our algorithms allow us to avoid grid storage issues mentioned by [18].

After having chosen the grid in projective space as an acceleration structure, a classical approach would be:

1. Build a grid giving, for each cell, the list of triangles it overlaps,
2. Then, for each ray, find the corresponding cell, and, thanks to the grid, find the corresponding triangles with which intersections need to be tested.
3. Finally, compute the intersections between rays and triangles.

We will explain in this section why we decided to change the structure of the algorithm, build a ray grid, and create partial triangle grids.

3.1 Grid storage issue

First comes a description of the data organization chosen to represent the grid, identical to the one presented in [19]. Working on GPU, it is not possible to have for each 2D cell a vector giving the indices of the triangles intersecting the cell, as the access to these vectors from the GPU kernel would be tedious. Instead, a unique vector, called $trgIds$, is used. It stores contiguously the indices of triangles overlapping each cell. These indices are sorted by index of corresponding cell.

Now, given a cell i, it is necessary to know how to get the list of triangles overlapping this cell. This piece of information is given by a second vector, $cellStartId$. $cellStartId[i]$ gives the position in $trgIds$ where begins the list of triangles that correspond to the cell i. This way, indices of triangles overlapping a given cell i will be found in the vector $trgIds$, between the positions $cellStartId[i]$ and $cellStartId[i+1]$. The construction of the grid thereby consists in the construction of these two vectors, obviously as fast as possible.

The main concern with this approach comes from the sizes of these two vectors, which can vary significantly from one scene to another, or even from one viewpoint to another, and cannot be directly predicted knowing only the number of triangles in the scene. Scenes with numerous large triangles are likely to be difficult to handle. The Sully Scene (804K triangles) is a good example of such a scene. The grid of this scene is composed by more than 10 million cells, which means that after the only building of the grid, and before any intersection test is made, a lot of memory is already used on the graphical unit to store ray and point coordinates, triangle descriptions, and these two vectors.

3.2 Triangle tracing

This is why we decided not to construct this grid, and swap the steps 1 and 2 of the algorithm described at the beginning of this section. Instead of what was presented, we build a grid that stores information about rays. This grid enables us to know, for each cell, which rays fall inside of it. The building of this new ray-oriented grid will be the new initialization step of the algorithm. Then, during the computation step, triangles will be "traced" against the

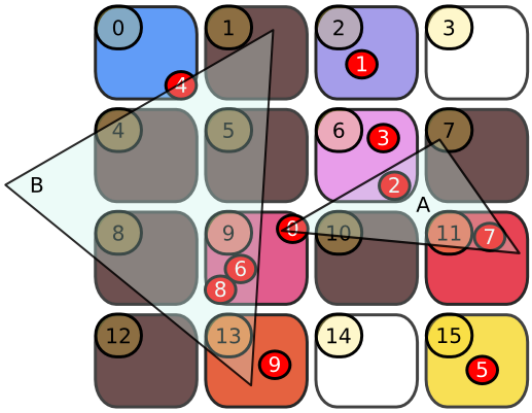


Figure 2: Example of 2 triangles A and B in a perspective view. The associated perspective grid has a resolution of $4*4$, indices of the cells are indicated in the upper-left corner of each cell. Red circles numbered from 0 to 9 are the rays starting from the eye.

ray grid: for each triangle, the cells overlapping the triangle are found, and then intersections are computed with rays overlapping these cells.

By doing this switch, this technique becomes practically equivalent to hierarchical rasterization, which is quite logical, since those approaches face similar problems:

1. Building a fixed regular grid in 2D perspective space is equivalent to a regular tiling of the screen.
2. Stream over triangles, and *build a grid over them* is equivalent to testing triangles against the tiles of the screen.
3. Intersecting rays and triangles within the same cell is equivalent to performing sample tests in screen tiles which overlap the triangle.
4. Generating all hits, rather than just nearest hits: with depth testing optimizations disabled, a rasterizer will always generate all hits.

In fact, this switch basically corresponds to irregular rasterization. It can also be noticed that some of the first attempts to adapt the ray-tracing algorithm on GPU were based on a very similar idea: targeting older-generation GPUs with limited programmability, [7], for instance, also streamed the triangles, testing each against all rays.

The difference with hierarchical rasterization comes here from the storing of all hits, which is made possible with our technique. Also, as stated above, a classical rasterizer cannot be used if the rays are not regularly spaced.

The construction algorithm of the classical projective grid requires a radix sort on a vector which size is the number of couples triangle/cell (noted $nbRefs$) that can be found in the scene. This radix sort is the bottleneck of the overall simulation. Instead, the radix sort used for the ray grid construction will be applied to a vector which size is the number of rays falling in a cell of the grid (noted $nbRaysUsed$). Except for very small scenes, $nbRaysUsed$ will always be far smaller than $nbRefs$ (we need approximately a million of

rays in our scenes, and have found the best resolution to be $1000 * 1000$ for the grid). This will make the radix sort far faster for most cases. The use of the ray grid will also imply sorting the intersections found by index of ray, but, during our tests, the ray grid always remained more efficient. The fact that the ray grid is far smaller than the triangle grid will also reduce the time needed to bind arrays to texture, which can be relevant for very large scenes.

The second main reason for doing this switch is that it is now possible to build the triangle grid in multiple passes. This way, the memory that has been used for a previous group of triangles can be deallocated before treating another bunch of triangles. The only memory that cannot be freed between these passes is the one used to store intersections. But this is not a demanding one, as an intersection is described by 8 bytes : an integer for the ray index, and a single float for the coordinate of the intersection along the ray. Hence, scenes with lots of large triangles can now be treated, and transfers to the CPU mentioned by [18] to avoid memory overflows are no more needed.

The last advantage of this algorithm is that it can be easily used on multi-GPU platforms, as the ray grid just has to be duplicated on each GPU, and then each GPU can handle a unique set of triangles.

4. RAY GRID CONSTRUCTION

After having exposed the reasons for introducing the ray grid, we present in this section the algorithm (see Figure 3) used to construct the ray grid in perspective space using the GPU. An example of a very simple scene can be seen on Figure 2. Figure 3, presenting the algorithm exposed in the next section, use this case as an example.

The two vectors describing the grid are named $rayIds$ and $cellStartId$, with the same conventions than with the triangle grid (See 3.1). The different steps of the algorithms can be found on the figures, they are numbered like the different subsections.

4.1 Find ray position in projective grid

First step of the algorithm is obvious: each thread works on one ray and finds in which cell it falls.

4.2 Group rays by cellId

The next step of the algorithm consists in sorting couples ray/cell (respectively found in the arrays $rayIds$ and $cellIdOfRay$) by increasing index of cell. This way, the rays that belong to a same cell will be contiguous in memory. We used here the sort implementation of Thrust [11], based upon the latest results in the litterature, as [27] and [29]. This step is in fact the bottleneck of this ray grid construction. It can be noticed that $rayIds$ is already the vector we wanted to produce.

4.3 Count number of rays by cell

Now comes the less obvious parts of the algorithm. To construct the reference vector $cellStartId$ describing where each cell begins in $rayIds$, it is first necessary to generate an array giving the number of rays each cell contains (it will be obtained at step 4.4). This array will be first initialized with zeros. Then will be needed the number of rays in each non-empty cell and the position where to write this number in the array, given by the index of the associated non-empty cell. On a CPU, these informations are easy to find by executing a simple loop on the array $cellIdOfRay$ generated in step 4.2. It is not possible to directly adapt this loop on GPU, because it would produce writing conflicts. To avoid these conflicts, it is necessary to generate two separate vectors, one giving the indices of the non-empty cells, and the other one giving the number of rays in each of these cells, indicating respectively where and what each thread will have to write to produce a correct array at step 4.4.

These two arrays are easily obtained by performing a segmented reduction, with cell indices (from $cellIdOfRay$) as keys and a constant array of ones as values. An efficient implementation of this operation can also be found in Thrust [11] (see thrust::reduce_by_key).

4.4 Write number of rays by cell

Thanks to the previous step, it is now possible to write the values included in $nbRaysByCell$ in the array $cellStartId$, at positions given by $cellIdOfRay$.

4.5 Final writings in cellStartId

Now that the number of rays in each cell is written in $cellIdOfRay$, an exclusive scan on this array provides a vector $cellStartId$ that is exactly the one we wanted to produce.

5. TRIANGLE TRACING

After this initialization comes the phase that could be called triangle tracing. For each triangle, the cells overlapping it are identified. Then, thanks to the ray grid, the potentially intersecting rays are found, and the intersection tests are performed.

5.1 Creation of the list of couples triangle/cell

This step can be seen as a very partial construction of the triangle grid. Here, it will only be a list of the couples triangle/cell found, when the classical algorithm would also have included a sort of these couples by index of cell and other steps to generate the vector of references $cellStartId$. Compared with the ray grid algorithm presented in the previous section, we could say that we just execute here the step 4.1. But this step is in fact more complicated and time-consuming, since each triangle can overlap more than one cell.

The triangles are rasterized via a scanline algorithm. To reduct load balancing issues that could be created by triangles of very different widths, they will be divided into different lines, and then each thread will work on a different line. Since the number of lines for each triangle cannot be known in advance, it is not possible to directly predict where each thread will have to write its own results.

Therefore, each thread is being assigned a triangle, and must count the number of lines (of the grid) it overlaps. The result is written in $nbLines$. An exclusive scan is performed on this vector, so that each thread working on the triangle i will write its results at the position $nbLines[i]$ in the array $startStop$. $startStop[i]$ is a little structure of three integers, so that $startStop[i].start$ and $startStop[i].stop$ are respectively the beginning and ending indices of cells for the line number i in the overall scene. $startStop[i].trgIdx$ is the index of the triangle concerned by this line.

Each thread working on a triangle now scans the lines between the highest and the lowest vertex of the triangle, and stores the $startStop$ associated with each of these lines, beginning to write at position $nbLines[i]$.

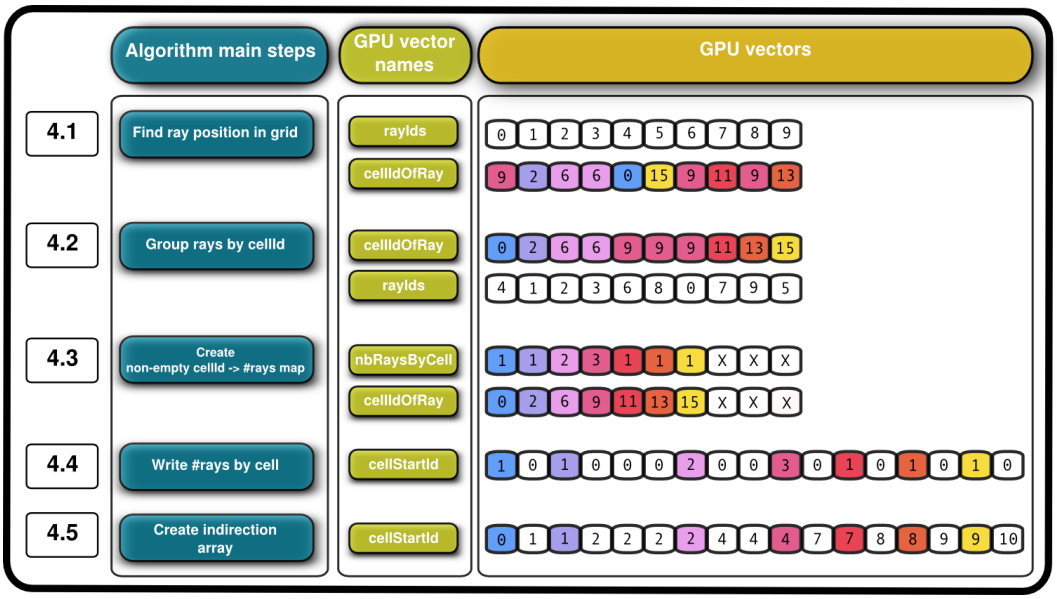


Figure 3: Ray grid construction algorithm. The GPU vectors given as example use the grid presented in Figure 2. *cellStartId* is of size *gridSize + 1*, in order to get the number of rays in the last cell by computing *cellIdOfRay[gridSize] - cellIdOfRay[gridSize - 1]*.

The same work than the one executed for lines is now made for cells. First, a new kernel is run, which counts the number of cells for each line. An exclusive scan on the resulting vector provides the total number of cells, and the position where each thread working on one line will write the corresponding couples cell/triangle. Hence, a new kernel is thrown, in which each thread works on one line, and effectively stores in memory the associated couples triangle/cell.

5.2 Prepare to store intersections

Now that the couples cell/triangle are generated, we just have to find for each cell the rays it contains. Then, the intersection tests will be made with the associated triangle. Since each cell can contain more than one ray, it is necessary to begin by computing the number of tests to be executed for each cell. This is done by using for the first time the ray grid generated in section 4. The number of rays falling on a cell of index i is given by the value $cellStartId[i+1] - cellStartId[i]$. Each thread working on a different couple cell/triangle counts the number of rays contained by its cell. This value is the number of tests that will be executed for the associated couple cell/triangle. An exclusive scan on the generated array provides the positions where each thread will have to store the indices of rays/triangles to be tested.

Then, each thread is being assigned a couple cell/triangle, and, thanks to the previous step, generates the couples ray-triangle to test.

5.3 Compute intersections

To store these results, two arrays $result$ and $rayAndCoord$ are created. If an intersection is found for test number i, $result[i]$ is set to 1, and $rayAndCoord[i]$ to the value $(rayId, t)$, where $rayId$ is the index of the related ray, and t the intersection coordinate. If M is the intersection point, and $\vec{OP}$ the ray, t is defined such that $\vec{OM} = t.\vec{OP}$.

5.4 Removing couples triangle/ray not intersecting

To compute the final list of intersections, elements in the array $rayAndCoord$ unrelated with an intersection are removed.

5.5 Sorting intersections by ray index

The array $rayAndCoord$ is then sorted by increasing index of ray. Then, intersections related to a same ray are sorted by t coordinate. Therefore, $rayAndCoord$ is then the array of intersections that had to be produced.

6. RESULTS AND DISCUSSION

6.1 Implementation

Having described the algorithms used in our GPU exhaustive raytracer, we can now evaluate its performance on several scenes. We use an NVIDIA GeForce GTX 295 (2*896 MB, GPU/shader/memory clocks of 576/1242/1998 MHz), coupled with a 4 core, Intel(R) Xeon(R) X5550 @ 2.67GHz. After the transfer of the data between the CPU and the GPU, all of the computations are done on the GPU. To test the algorithm on different levels of complexity, we use various scenes: Erw6, Fairy Forest, Conference, Sully and NuclearCase. NuclearCase is a scene we designed to control the quality of the mesh: we need very clean meshes grouped by object. Each mesh associated with an object must be closed, so that we do not miss any entry or exit point (only one omission can lead to a completely false result).

6.2 Grid constructions

Experiences showed that the ray grid construction time is almost constant for a given number of rays falling in the grid. It is due to the fact that the bottleneck of this algorithm is the radix sort performed at step 4.2 of the algorithm. The

performance of this sort is very regular for a given number of couples to sort.

A total construction time of 20 ms was obtained for a grid referencing one million rays. These times are a little better than those obtained by [18], for a triangle grid with 1,1 million of references (27 ms for the Conference, 24 for Fairy Forest).

The partial construction time of the triangle grid is far more influenced by the repartition of the different triangles in the scene. Times to generate the list of couples cell/triangle for a resolution of $1024 * 1024$ can be seen on table 1.

Scene	nbTrgs	nbCouples	Time	T	C
Erw6	804	1.3M	14	17K	10.8
Fairy	174K	4.1M	28	160	6.8
Conf.	283K	6.1M	47	166	7.7
Sully	804K	29M	123	152	4.2
Nucl.	738K	28M	210	37	7.5

Table 1: Build statistics for various scenes. nbCouples is the number of couples cell/triangle. The column Time gives the times necessary to generate these couples. T means the time spent per million of triangles, and C the time spent per million of couples. Times are given in milliseconds. Nucl. stands for NuclearCase.

Table 1 clearly shows that the construction time does not depend on the number of triangles of the scene, but on the number of couples triangle/cell that can be found in the triangle-oriented grid. This observation is based on the values of the coefficients C and T, respectively giving the number of milliseconds necessary to treat one million of couples triangle/cell. Contrary to T, the value C seems quite stable in the different scenes. This observation confirms the fact that our algorithm is not limited by the number of triangles in the scene, but by their respective width.

6.3 Exhaustive intersection computation

Scene	nbTests	nbInters	trgTrace	compInter
Erw6	1.3M	1.2M	48.3	56.9
Fairy	4.0M	1.8M	73.3	84.1
Conf.	6.1M	2.7M	125.9	143.5
Sully	29.1M	3.7M	305	325.4
Nucl.	14.0M	10.3M	498.4	517.9

Table 2: Intersection test statistics for 1024*1024 primary rays on various scenes. trgTrace stands for the time spent to execute the complete triangle tracing algorithm presented in section 5, and compInter is the overall time of the simulation (except for the ray grid construction time). Times are given in milliseconds.

Performances of our overall algorithm are reported on table 2. The rays thrown are 1024*1024 primary rays. In such a case, some optimizations like those done in rasterization could have been investigated. But we did not want to modify our algorithm, in order to see how it could work without knowing that the rays are so regularly spaced. The ray construction time is not included in these results, which means that about 20 ms have to be added to *compInter* to get

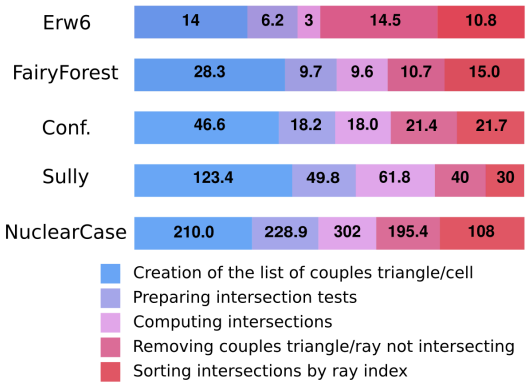


Figure 4: Times for the different stages of the algorithm described in section 5 (in milliseconds). The last step of the algorithm (sorting the coordinates for each ray) is not included.

the overall time of the simulation in each case. This table clearly shows that most of the computation time is spent in the execution of the algorithm described in the section 5. It also has to be noticed that the execution of the program for the NuclearCase scene required two loops of triangles in order to avoid memory overflow.

Figure 4 shows more precisely how the time is used during the execution of this phase. It can be noticed that the intersection requests are not the bottleneck of these computations. The major step of the algorithm is the generation of the couples triangle/cell (rasterization step), but all of the steps participate in the computation time in a noticeable way. The rasterization step could certainly be improved by using more adapted rasterization techniques, such as [9], but this would not modify our results by an order of magnitude. For most of the scenes, we are able to achieve 20 millions of intersections by second, which is quite good, since all these intersections are sorted, and since the generation of all of the intersections produces a lot of overhead to prepare the memory for writing the results.

To compare these results, we also implemented the classical algorithm, *i.e* we generated the triangle grid and then scanned the different rays of the scene. Since these algorithms share lots of common operations, it is very simple to compare them. On the one hand, the classical algorithm builds the triangle grid and generates the list of couples ray/cell. On the other hand, the triangle tracing algorithm builds the ray grid and generates the list of couples ray/triangle. It also has to sort the intersections by index of ray at the end of the computations. Since the time to generate the list of couples ray/cell for the classical algorithm is not relevant (about one millisecond for one million of rays), we just have to compare the triangle grid construction time with the sum of the three steps only executed in our algorithm. The results are given in table 3. First, it has to be noticed that the triangle grid could not be generated on the NuclearCase, because the amount of memory available on the GPU was not enough to generate the grid (985 MB of memory would have been necessary for this case). It reminds that one of the reasons for inverting the classical algorithm was that it would solve these kinds of memory overflow issues. Then, it can be seen that the classical algorithm is

clearly less efficient than the ray grid, especially when the size of the scene increases.

Scene	couplesConstr	sortInters	total	trgGrid
Erw6	14.0	10.8	44.8	33.2
Fairy	28.3	15.0	63.3	69.2
Conf.	46.6	21.7	88.3	108.0
Sully	123.4	30	173.4	408.9
Nucl.	210.0	108.0	318.0	X

Table 3: Comparison between the triangle tracing and the classical algorithm. couplesConstr is the time used to build the couples cell/triangle for the triangle tracing algorithm, and sortInters the time needed to sort the intersections by index of ray. Total results from the sum of these two numbers added to the ray grid construction time. It has to be compared with trgGrid, giving the construction time of the classical triangle grid. Times are given in milliseconds.

This can easily be explained: as seen above, the bottlenecks of the grid constructions are the sorts that have to be done on the couples ray/cell (for the ray grid) and triangle/cell (for the triangle grid). The number of couples triangle/cell will for most cases be far greater than the number of couples ray/cell (at most equal to the number of rays). This can lead to a very longer sorting time for the triangle grid, as can be seen on the Sully case. The drawback of our method is that we have to sort the intersections by index of ray, whereas they are already sorted in the classical algorithm. But, for most cases (especially large scenes), it is not enough to compensate for the time lost in sorting the couples triangle/cell in the classical algorithm.

We decided not to implement a BVH or other classical acceleration structures, because the perspective grid is clearly the structure that implies the fastest traversal time. This allows a fast access to the potentially intersecting rays for each triangle, once the cells it overlaps have been found. With a classical structure such as a BVH, the traversal time for one ray can be very long: in order to compute all intersections for each ray, it would first be needed to determine for each ray how many intersection tests have to be performed, then doing again this traversal to write the associated indices of triangles in memory, and finally run the tests. This would easily lead to load-balancing issues that are avoided here. Moreover, the construction times of the BVHs are clearly too elevated in the case of dynamic scenes, especially compared with our grids.

6.4 Experimentations on a nuclear-like scene

Finally, we tried to execute a test case representative of the usages that will be done for nuclear industry. We defined 6 groups of 100 000 points, representing the sources of radiation of the nuclearCase scene. All rays thrown in our test case start from the same point (representing the point where the dose has to be measured), and end at one of the points described in our separate file.

These rays and the overall scene are seen through an external camera. When a ray passes through the air, it is colored in white, and it becomes red when passing through an object (see Figure 1). To detect if a ray enters in an object or leaves it, we consider that the starting point of the rays is not included in any object of the scene. Thus, since the scene is very clean, the first intersection found along a ray can be considered as an entry point in an object, and the next one will be the exit point of this object.

We were able to compute and sort about 6 millions of intersections by step of the simulation, in a time of 400ms. The difficulty here was that 6 grids had to be generated to cover the whole space. To be more efficient, and avoid storing memory for cells that will, for sure, not generate any intersection, we used the bounding boxes of the groups of sources. First step in the computations consisted in testing if the triangles were overlapped by these boxes, and to discard them if not. We also tried to use the fact that the Nvidia GTX 295 is in fact the union of two GPUs. Thus, we used another card to display the graphics, and we tried to distribute the computations over the two GPUs. We tried two different strategies: the first one only consisted in giving three grids to the first GPU, and three others to the second one. The second strategy consisted in arbitrarily splitting the triangles into two groups of same dimension and to make each GPU work on a different group. The first one was the more efficient here: we could achieve a total time of 250 ms for the overall simulation, while we only achieved a time of 350 ms with the second strategy.

This can be explained by the good repartition of the different groups over the scenes in this case. But it is clearly a problem that needs to be investigated in the future.

7. CONCLUSION

We proposed a new GPU algorithm designed to compute every intersection between highly coherent rays and a complex 3D scene. In order to increase performance and avoid memory overflow issues, we introduced the use of a ray grid in perspective space. We postpone the partial building of the triangle grid after the building of the triangle grid. All of the algorithms proposed are designed to efficiently manage the finding of all of the intersections along each ray, which can sometimes lead to solutions fundamentally different from the ones used in traditional ray tracing.

This algorithm can also be very useful in a multi-GPU application, since each GPU can process the computations for a different bunch of triangles. As seen above, a simple repartition of the different triangles between the different GPUs is not very efficient. Strategies for efficiently distributing the triangles over the GPUs could bring much higher performance, and have to be investigated in the future.

8. REFERENCES

[1] T. Aila and S. Laine. Alias-Free Shadow Maps. In *Proceedings of Eurographics Symposium on Rendering 2004*, pages 161–166. Eurographics Association, 2004.

[2] A. Assad, M. Chiron, J.-C. Nimal, C. M. Diop, and P. Ridoux. General formalism for calculating gamma-ray buildup factors in multilayer shields into MERCURE-6 code. In *Proceedings of the Ninth International Conference on Radiation Shielding*, page 493. Japan: Atomic Energy Society of Japan, 1999.

[3] L. Bavoil, S. P. Callahan, A. Lefohn, a. L. D. Comba, Jo and C. T. Silva. Multi-fragment effects on the GPU using the k-buffer. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 97–104, New York, NY, USA, 2007. ACM.

[4] M. Billeter, O. Olsson, and U. Assarsson. Efficient stream compaction on wide SIMD many-core architectures. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 159–166, New York, NY, USA, 2009. ACM.

[5] G. Cadet and B. Lecussan. Coupled Use of BSP and BVH Trees in Order to Exploit Ray Bundle Performance. In *RT '07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 63–71, Washington, DC, USA, 2007. IEEE Computer Society.

[6] L. Carpenter. The A-buffer, an antialiased hidden surface method. *SIGGRAPH Comput. Graph.*, 18(3):103–108, 1984.

[7] N. A. Carr, J. D. Hall, and J. C. Hart. The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[8] E. E. Catmull. *A subdivision algorithm for computer display of curved surfaces.* PhD thesis, 1974.

[9] C. Eisenacher and C. Loop. Data-parallel Micropolygon Rasterization. In S. Seipel and H. Lensch, editors, *Eurographics 2010 Annex: Short Papers*, pages x–x, May 2010.

[10] C. Everitt. Interactive Order-Independent Transparency, NVIDIA. Technical report, 2001.

[11] J. Hoberock and N. Bell. Thrust: A Parallel Template Library, 2009. Version 1.2.

[12] M. Houston, A. J. Preetham, and M. Segal. Graphics Hardware (2005), pp. 1-6 M. Meissner, B.-O. Schneider (Editors) A Hardware F-Buffer Implementation.

[13] W. Hunt and W. R. Mark. Ray-Specialized Acceleration Structures for Ray Tracing. In *IEEE/EG Symposium on Interactive Ray Tracing 2008*, pages 3–10. IEEE/EG, Aug 2008.

[14] P. Ivson, L. Duarte, and W. Celes. GPU-Accelerated Uniform Grid Construction for Ray Tracing Dynamic Scenes. Technical Report 14/09, Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, June 2009.

[15] T. Ize, I. Wald, and S. G. Parker. Asynchronous BVH Construction for Ray Tracing Dynamic Scenes on Parallel Multi-Core Architectures. In *Proceedings of the 2007 Eurographics Symposium on Parallel Graphics and Visualization*, 2007.

[16] G. S. Johnson, J. Lee, C. A. Burns, and W. R. Mark. The irregular Z-buffer: Hardware acceleration for irregular data structures. *ACM Trans. Graph.*, 24(4):1462–1482, 2005.

[17] N. P. Jouppi and C.-F. Chang. Z3: an economical hardware technique for high-quality antialiasing and transparency. In *HWWS '99: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 85–93, New York, NY, USA, 1999. ACM.

[18] J. Kalojanov and P. Slusallek. A parallel algorithm for construction of uniform grids. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 23–28, New York, NY, USA, 2009. ACM.

[19] A. Lagae and P. Dutré. Compact, Fast and Robust Grids for Ray Tracing. *Computer Graphics Forum (Proceedings of the 19th Eurographics Symposium on Rendering)*, 27(4):1235–1244, June 2008.

[20] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH Construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, 2009.

[21] C. Lauterbach, S.-E. Yoon, and D. Manocha. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. In *In Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 39–45, 2006.

[22] F. Liu, M.-C. Huang, X.-H. Liu, and E.-H. Wu. Efficient depth peeling via bucket sort. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 51–57, New York, NY, USA, 2009. ACM.

[23] W. R. Mark and K. Proudfoot. The F-buffer: a rasterization-order FIFO buffer for multi-pass rendering. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 57–64, New York, NY, USA, 2001. ACM.

[24] NVIDIA. Cuda Zone, The resource for CUDA Developers. http://www.nvidia.com/object/cuda_home.html, 2009.

[25] OpenCL. http://www.khronos.org/opencl/, 2009.

[26] A. Reshetov, A. Soupikov, and J. Hurley. Multi-level ray tracing algorithm. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1176–1185, New York, NY, USA, 2005. ACM.

[27] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[28] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, number 3, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[29] E. Sintorn and U. Assarsson. Fast parallel GPU-sorting using a hybrid algorithm. *J. Parallel Distrib. Comput.*, 68(10):1381–1388, 2008.

[30] I. Wald. *Realtime Ray Tracing and Interactive Global Illumination.* PhD thesis, Computer Graphics Group, Saarland University, 2004.

[31] I. Wald, S. Boulos, and P. Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26(1):6, 2007.

[32] C. M. Wittenbrink. R-buffer: a pointerless A-buffer hardware architecture. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 73–80, New York, NY, USA, 2001. ACM.

[33] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time KD-tree construction on graphics hardware. *ACM Trans. Graph.*, 27(5):1–11, 2008.

Solving Planted Motif Problem on GPU

Naga Shailaja Dasari
Old Dominion University
Norfolk, VA, USA
ndasari@cs.odu.edu

Ranjan Desh
Old Dominion University
Norfolk, VA, USA
dranjan@cs.odu.edu

Zubair M
Old Dominion University
Norfolk, VA, USA
zubair@cs.odu.edu

ABSTRACT

(l, d) planted motif problem is defined as: Given a sequence of n DNA sequences, each of length L, find M, the set of sequences(or motifs) of length l which have at-least one d-neighbor in each of the n sequences. Planted motif problem is an important and well-studied problem in computational biology. Motif finding is useful for developing methods to obtain transcription factor binding sites, sequence classification, in developing methods for building phylogenetic trees etc. The planted motif problem is difficult to solve especially for challenging instance sizes (15,5), (17,6), (19,7), and (21,8). The challenging instances are computationally intensive and require large amount of memory. Several serial implementations have been proposed for solving this problem. The time required by these methods for solving large challenge instances is prohibitively expensive. In this paper, we propose a parallel implementation on GPU that solves the challenge instance (21,8) in 1.1 hours. We are not aware of any sequential or parallel method that will solve this challenge instance in better time. Additionally, to the best our knowledge we are not aware of any previous implementation of a parallel method to solve the planted motif problem on GPU.

1. INTRODUCTION

Motif finding is an important and well-studied problem in computational biology [20] [6]. Motif finding is useful for developing methods to obtain transcription factor binding sites, sequence classification, in developing methods for building phylogenetic trees etc. [11]. Finding motifs is a computationally expensive and challenging task. Many variants of motif finding problem can be found in the literature [6]. One set of variants concentrates on finding repeated patterns in a single sequence, and the other set concentrates on finding patterns that appear in multiple sequences. The planted motif problem (PMP) falls in the second category.

The (l, d) planted motif problem can be defined as "Given a sequence of n DNA sequences, each of length L, find M, the set of sequences(or motifs) of length l which have at-least one d-neighbor in each of the n sequences". A d-neighbor of an l-mer(sequence of length l) p is defined as an l-mer that is at a Hamming distance of d or less from p. In the rest of the paper, we refer to l as *enumeration length* and d as *enumeration distance*.

A number of approaches have been proposed to solve the motif finding problem including PMP. Some of these approaches find approximate motifs [14], [2], [16] and others find exact motifs[9], [18], [19], [5], [17], [12], [3], [15], [10], [8]. These approaches can be classified into two types: iterative approaches and combinatorial approaches. Iterative approaches like Gibbs sampling and expectation maximization are based on position weight matrices while combinatorial approaches like MITRA, WINDOWER are based on hamming distances. Planted motif problem defined in this paper is based on hamming distances.

Most approaches to solve PMP are serial in nature and are difficult to parallelize. We had recently proposed a new parallel approach to solve PMP called BitBased approach[7]. BitBased is a simple, easily parallelizable approach. It outperforms all the approaches proposed so far to solve the planted motif problem. In this paper, we show how to implement BitBased on GPU architecture. Iterative approaches like Gibbs sampling [21] and MEME [4] have been implemented on GPU while there are no combinatorial approaches implemented on GPU currently.

BitBased is an enumeration based approach to solving planted motif problem. It uses n' bit arrays, $n' \leq n$, of size 4^l each to find the planted motifs. Each bit in the bit array corresponds to an l-mer. The key idea of BitBased is to enumerate all the l-mers in the input sequences to find their d-neighbors and set the bits corresponding to the d-neighbors in the bit arrays. It then uses the bit arrays to find the planted motifs. It can be noticed that BitBased has high memory requirement. To reduce memory requirement one can use the iterative BitBased approach at the expense of increasing the execution time. Iterative approach works by virtually partitioning the bit arrays into chunks such that a chunk fits in the available memory. We then make multiple passes of the original algorithm to find motifs. The number of passes is determined by the number of virtual partitions. A small chunk size results in increased number of virtual partitions, and thus increasing the overall time to find motifs.

GPUs are becoming increasingly popular in the world of parallel computing. GPUs, which were once used only for graphics, are now being used for different types of applications to achieve high performance. With the advent of CUDA, the task of programming for GPU has become much simple. A GPU is a massively parallel, multi-threaded, manycore processor with hundreds of cores and huge computation power. It can execute thousands of threads concurrently. The programmer must carefully design her application to map to GPU and effectively utilize the hardware.

In this paper we parallelize the BitBased approach[7] for GPU. Though BitBased approach is easily parallelizable, it is challenging to effectively implement it on GPU. The reason being the high memory requirement. We have seen that BitBased uses bit arrays to find planted motifs and that the bit arrays are of size 4^l bits each. And moreover the access to the bit arrays is very scattered. For example, to solve a $(15, 5)$ instance, BitBased needs bit arrays of size 128MB each. Such amount of memory is only available on GPU's global memory. But global memory has very high latencies especially when the access pattern is scattered. In such cases it is highly recommended to use GPU's shared memory. But the shared memory is too small (16KB for Tesla C1060 and S1070) to accommodate the bit arrays. So we use iterative BitBased approach and partition the bit arrays into chunks that fit in shared memory. We then optimize the approach by decreasing the register usage which increases the occupation of the GPU. We also do reordering of shared memory to avoid bank conflicts. .

We have implemented BitBased on NVidia Tesla C1060 which has one GPU device and NVidia Tesla S1070 which has four GPU devices. Tesla C1060 has 30 multi-processors with 8 streaming processor cores each while Tesla S1070 has 960 cores. We tested the (15,5), (17,6), (19,7), (21,8) challenging instances. Tesla C1060 took 8 seconds, 1.52 minutes, 19.7 minutes and 4.5 hours respectively and Tesla S1070 took 3 seconds, 23.9 seconds, 5 minutes and 69 minutes respectively. These are the best timings obtained for planted motif problem so far. We also compare with the results on multicore architecture. We found that a single GPU shows up to 13 to 14 times speed-up and 4 GPU devices shows up to 40 to 60 times speed-up compared to single core CPU.

2. THE BITBASED APPROACH

BitBased approach is a simple, easily parallelizable approach to solving PMP. It is based on exhaustive enumeration of l-mers in the input sequences. Let $S = \{S_i | 0 \leq i \leq n-1\}$ be the set of n input sequences. An l-mer in S_i starting at location j, $0 \leq j \leq L-l$ is represented as $S_i^l\{j\}$. The set of d-neighbors of all the l-mers in S_i is represented by $N_i^{l,d}$. It is easy to see that the set of planted motifs is $M = \bigcap_{i=0}^{n-1} N_i^{l,d}$. Therefore, to find the planted motifs we first need to generate the set of $N_i^{l,d}$, $0 \leq i \leq n-1$, and then find the motifs, i.e. l-mers that are present in all $N_i^{l,d}$, $0 \leq i \leq n-1$. The main issue here is the memory requirement. To see the issue consider $(15, 5)$ instance. For a 15-mer, there can be 853584 number of 5-neighbors. For a sequence of length 600, the size of $N_i^{l,d}$ is 500200224 integers which requires approximately 2GB of memory for a single sequence. To reduce the memory requirement we use bit arrays of size 4^l. Each bit in the bit array corresponds to an l-mer. For example, when $l = 4$ bit 0 represents AAAA, bit 1 represents AAAC, bit 255 represents TTTT assuming A=0, C=1, G=2, T=3. For $(15, 5)$ instance we now require only 4^{15} bits i.e. 128MB of memory for each input sequence. The memory requirement can further be reduced using the approaches mentioned in sections 2.1.1, 2.1.2 and 2.2.

2.1 The basic BitBased approach

The basic BitBased approach consists of two phases, setting bits and finding motifs. In setting bits phase, $N_i^{l,d}$, $0 \leq i \leq n-1$, is generated. $N_i^{l,d}$ is represented using bit arrays. A bit array B_i is assigned to each input sequence S_i, $0 \leq i \leq n-1$. Each l-mer in sequence S_i is enumerated to generate all its d-neighbors and the bits are set in the bit array B_i at the indexes corresponding to the d-neighbors. The index corresponding to an l-mer can be obtained by replacing A by 00, C by 01, G by 10 and T by 11. For example the index corresponding to the 4-mer GACT is 10000111. After setting bits phase, a bit array B_i has a bit set only if the l-mer corresponding to its index is present in $N_i^{l,d}$.

In finding motifs phase, the equivalent to $M = \bigcap_{i=0}^{n-1} N_i^{l,d}$ is performed. We perform logical AND operation on the bit arrays to generate a single bit array which can be used to obtain the planted motifs. The final bit array B is obtained by $B = B_0 \bigwedge B_1 \bigwedge ... \bigwedge B_{n-1}$. A bit is set at index j in B only if the bit is set at index j in all the bit arrays B_i, $0 \leq i \leq n-1$. In other words, the l-mer corresponding to the index j is present in all $N_i^{l,d}$, $0 \leq i \leq n-1$ making the l-mer a planted motif. Therefore the planted motifs are nothing but the l-mers corresponding to the indexes in B in which a bit is set.

To reduce the memory requirement further, we use two modifications to the basic approach: Increment motifs and filter motifs. These modifications, if applicable, not only reduce the memory requirement but also improve the performance.

2.1.1 Increment Motifs

This modification is based on the observation that given the set of motifs for $(l-1, d)$ instance, their d-neighbors and corresponding distances in all the n sequences, we can find the motifs for (l, d) instance in $O(n)$ time. Let p be a motif for $(l-1, d)$ instance. Let $(j_0, j_1, ..., j_{n-1})$ and $(d_0, d_1, ..., d_{n-1})$ be the locations of d-neighbors in n sequences and their distances respectively. We can say that $p|R$, $R \in \{A, C, G, T\}$ and '|' is append operation, has a d-neighbor in sequence S_i if it satisfies any of the following conditions: 1. residue at location $j_i + l$ is R. 2. $d_i < d$. For each motif p for $(l-1, d)$ instance, we find if $p|A$, $p|C$, $p|G$, $p|T$ is a motif for (l, d) instance using the above conditions. Therefore to find (l, d) motifs, we can first find (l', d), $l' \leq l$, motifs and then use the above logic incrementally to find (l, d) motifs. With decreasing values of l', the number of (l', d) motifs increase exponentially and hence the time spent in increment motifs. Therefore the value of l' must be carefully chosen.

2.1.2 Filter Motifs

Instead of setting bits and finding motifs for all n sequences, this modification first finds the motifs for n' sequences where $n' \leq n$. These motifs are called candidate motifs. These

candidate motifs are then filtered to find the final planted motifs. This is done by checking each of the candidate motifs if it is present in all the remaining $n - n'$ input sequences. This modification reduces the memory requirement because we now require only n' buffers instead of n buffers. By decreasing the value of n', not only the space requirement decreases but also the time decreases. The reason being that the time taken by BitBased approach is dominated by setting bits phase. By reducing n' we need to set the bits for fewer sequences and hence reducing the time taken. But if the value of n' is chosen to be too low, then the time spent in filtering motifs increases and so the overall time. So it is important to chose an optimum value for n'.

2.2 The Iterative BitBased Approach

This is a crucial modification to the basic BitBased approach and also is the basis for implementing BitBased on GPU. As we have seen previously, BitBased has high memory requirement. It might not always be possible to satisfy such requirement. In such cases, we can use the iterative BitBased approach. Iterative BitBased approach solves the planted motif problem with much less memory requirement but at the expense of increase in time due to the increase in number of operations. Iterative approach works by reusing the available memory to accomplish the required task, which is to find planted motifs. Let l_{max}=max{i $|4^i$ bits of memory can be allocated}. We virtually partition the bit array of size 4^l into $4^{l-l_{max}}$ chunks, each chunk of size $4^{l_{max}}$ bits. In ith iteration, the l-mers of input sequences are enumerated in such a way that the bits are only set in the ith chunk. After finding motifs in ith chunk the same memory is then reused for the $(i+1)$th iteration. Note that when bit array of size 4^l bits is partitioned into $4^{l-l_{max}}$ chunks, the first $l - l_{max}$ residues corresponding to the indexes in a chunk are all the same. For example, when we partition 4^{17} bits into 16 partitions, all the 17-mers corresponding to the indexes in the first chunk start with AA, second chunk starts with AC, and so on. To effectively enumerate the l-mers, we reduce the enumeration length from l to l_{max} as shown in algorithm 1. Note that the more number of chunks the bit array is partitioned into, the less is the enumeration length.

Algorithm 1 IterativeApproach

Input: n, l, l_{max}
Output: M, the set of (l, d) planted motifs

```
Let l_diff = l − l_max
M = ∅
for idx = 0 to 4^l_diff − 1 do
    get the sequence p of length l_diff that corresponds to idx
    {setting the bits in idx^th chunk}
    for i = 0 to n − 1 do
        for j = 0 to L − l + 1 do
            get distance d' between p and S_i^l_diff {j}
            generate N_i^{l_max, d−d'} {j + l_diff}
            for each l_max-mer q in N_i^{l_max, d−d'} {j + l_diff} do
                get index idx' corresponding to q
                set B_i[idx'] = 1
            end for
        end for
    end for
    {finding motifs in idx^th chunk}
    B = B_0 ⋀ B_1 ⋀ ... ⋀ B_{n−1}
    for i = 0 to 4^{l_max} − 1 do
        if B[i] = 1 then
            Let r be the l_max-mer corresponding to i
            Append r to p and add the appended sequence to M
        end if
    end for
    clear all the bit arrays B_0 to B_{n−1}
end for
```

3. OVERVIEW OF GPU

GPU is a massively parallel, multi-threaded, manycore processor. Each GPU device is an array of streaming multiprocessor which in turn consists of a number of scalar processor cores. GPU is capable of running thousands of threads concurrently. It is able to do so by employing SIMT(single-instruction multiple-threads) architecture. The threads are created, scheduled and executed in groups called warps. All the threads in a warp share a single instruction unit. The threads in a GPU are extremely light weight and they can be created and executed with zero scheduling overhead.

CUDA is a parallel programming model that enables programmers to develop scalable applications to be executed on GPU. It exposes a set of extension to C and C++. A CUDA program is organized into sequential host code which is executed on CPU and calls to functions called kernels which are executed on GPU. A kernel contains the device code that is executed by the GPU threads in parallel. CUDA threads can be grouped into thread blocks. Using CUDA one can define the number of blocks and the number of threads per block that can execute a kernel.

3.1 Memory organization

The device RAM is virtually and physically divided into different types of memory: global, local, constant and texture memory. Apart from device RAM the threads can also access on-chip shared memory and registers as shown in figure 1 [13]. Global memory and texture memory have highest latency compared to the other types of memory. A thread has exclusive access to its local memory. All the threads in a block can access on-chip shared memory. All the threads across all thread blocks have access to global, texture and constant memory. Constant and texture memories are read only while global is both read and write.

3.2 Performance considerations

A CUDA program should be properly designed taking advantage of the resources for better performance. Since GPU uses SIMT architecture in which all the threads in a warp use a single instruction unit, the best results can be achieved when all the threads in a warp execute without diverging. When threads diverge they are executed serially, thus decreasing performance.

Global memory has very high latency. But by coalescing the global memory accesses, high throughput can be achieved. For example if the threads in a warp access contiguous address, then only two transactions are issued. But if the

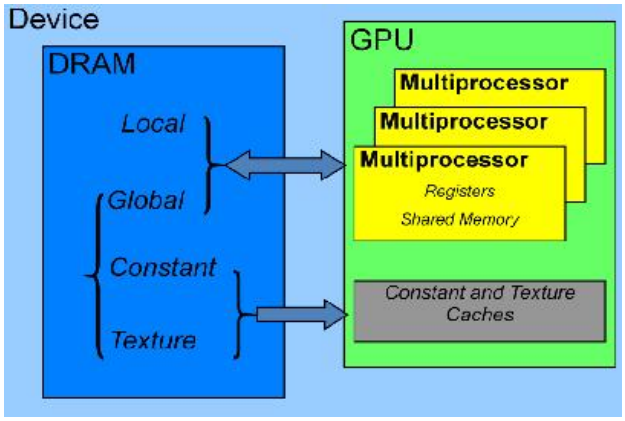


Figure 1: GPU Memory

threads access separate addresses then 32 transactions are issued.

Shared memory is divided into equally sized blocks called banks. If two threads in a half warp access the same bank, this would result in bank conflict and the accesses are serialized thus reducing the effective bandwidth. In order to avoid this, the programmer should try to make sure that the threads in a half warp access different banks.

The memory latencies can be hidden by executing other warps when a warp is paused. So to keep the hardware busy there should be enough active warps. Occupancy is the ratio of number of active warps per multi-processor to the maximum possible number of active warps. If the occupancy is too low, then the memory latency cannot be hidden resulting in performance degradation. So the programmer should try to increase the occupancy to effectively use the hardware.

4. PARALLELIZING BITBASED ON GPU

Though BitBased is a easily parallelizable approach, it is not straight-forward to implement it on the GPU. The main issue is that BitBased has high memory requirement. As we have seen in section 2, it requires 4^l bits of memory for each bit array. Such high amount of memory is only available on the global memory. But global memory has a drawback of high latency. Furthermore, the access pattern of the bit arrays is very scattered making it difficult to use the coalescing feature of the global memory. So to avoid using global memory, we partition the bit arrays into smaller chunks that fit in shared memory. This is similar to the iterative approach discussed in section 2.2. The only difference is that instead of iterating, we assign the task of each iteration to a GPU thread block.

Let t be the number of threads in each block. To solve (l, d) instance we first find l' and n' as explained in [7]. Let l_s=$max\{\text{i} \mid 4^i n'$ bits of memory can be allocated on shared memory$\}$. The bit arrays are partitioned into chunks of 4^{l_s} bits of memory. Each chunk is assigned to a single block. Thus the number of blocks is $4^{l'-l_s}$. The threads in each block enumerate the l-mers in such a way that they generate the d-neighbors only in the chunk of bit arrays assigned to the block. We use the same logic as in iterative approach. Note that the enumeration length here is l_s.

The t threads in a block are responsible for enumerating the l-mers in the input sequences and setting bits in the chunk of bit arrays assigned to the block. The l-mers are distributed among the t threads. The consecutive l-mers are assigned to consecutive threads. After all the threads have finished enumerating the l-mers and setting bits, the threads enter the find Motifs phase. After finding the candidate motifs, they must be filtered by checking if they are present in the remaining $n - n'$ input sequences. We perform this step in a separate kernel called *FilterMotifs* to avoid divergence of threads. So a thread, after finding a candidate motif instead of performing the filtering phase, it writes it to the global memory so that the candidate motif can be accessed in the *FilterMotifs* kernel. To write on to global memory, we use a variable called gIndex. When a thread finds a candidate motif, it first atomically increments gIndex and then writes the candidate motif to the global memory at the index returned by the atomic operation. This is to avoid different threads in different blocks writing to the same index in global memory.

After finding the candidate motifs, filtering them is straight forward. Let c be the number of candidate motifs. For the *FilterMotifs* kernel, we need c/t blocks. The c candidate motifs are equally distributed among the blocks. Within the block, the candidate motifs are further distributed among the threads. Each thread is assigned a candidate motif and it checks if the candidate motif has d-neighbors in the remaining $n - n'$ input sequences which were not considered during finding candidate motifs. If a thread finds that the candidate motif is a planted motif, it writes to the global memory using the same logic explained previously. We improve this implementation by using two modifications: Bit representation and repartitioning and reordering.

4.1 Bit Representation

As we have seen in section 3, each multiprocessor has a limited number of registers. This implementation is limited by the number of registers. Since each thread consumes large number of registers, the number of threads per block is less and hence the occupancy of GPU. To improve the occupancy and performance, we need to reduce the registry usage as much as possible. Each input sequence of length L has $L-l+1$ l-mers. If the input sequence is represented using a character array then an l-mer requires l bytes of memory. Instead we can represent an l-mer using an integer, 2 bits for each residue [1] [17]. For example, the 4-mer CGGA can be represented using an integer whose binary representation is 01101000. By doing so, an l-mer, $l \leq 16$, would need only 4 bytes and $l \leq 32$ would need 8 bytes of memory. So we convert the input character array into an integer array, the integer at index i represents the l-mer starting at location i in the input sequences. By converting into input array, GPU threads only need to read one integer rather than l bytes. This would not only reduce the registry usage by also reduce the I/O time as only an integer need to be read. We use texture binding to read the input sequences.

4.2 Repartitioning and reordering

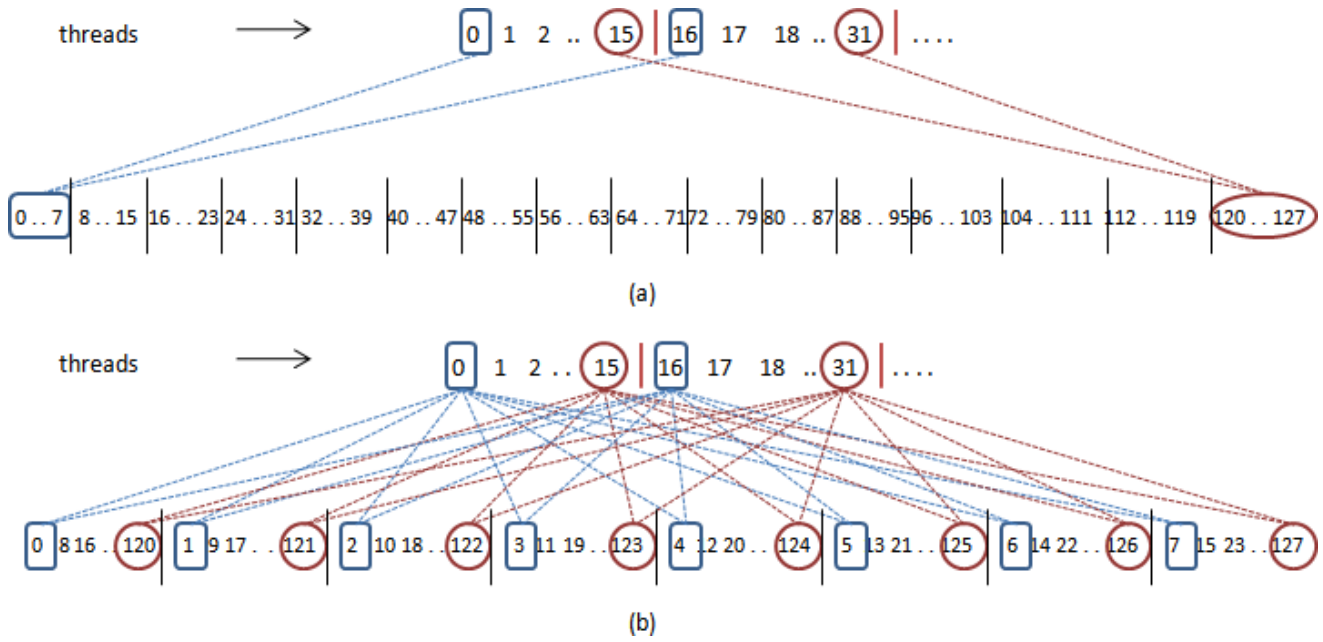


Figure 2: (a) The integer array is partitioned into 16 chunks so that the ith thread in a half warp only accesses ith chunk. (b) The integer array is reordered such that the ith thread in a half warp only accesses ith bank.

Table 1: Comparison with multicore

	$(15,5)$			$(17,6)$			$(19,7)$			$(21,8)$		
GPU devices	time (seconds)	speed-up 1 core CPU	speed-up 16 cores CPU	time (seconds)	speed-up 1 core CPU	speed-up 16 cores CPU	time (minutes)	speed-up 1 core CPU	speed-up 16 cores CPU	time (hours)	speed-up 1 core CPU	speed-up 16 cores CPU
1	8	13.5	1.4	91.2	13.6	1.6	19.7	14.3	1.6	4.5	-	1.5
2	4.4	24.5	2.5	46.1	26.8	3.1	9.9	28.5	3.1	2.3	-	3.0
3	3.2	33.6	3.4	31.1	39.7	4.6	6.62	42.6	4.6	1.5	-	4.6
4	2.7	40	4.1	23.9	51.7	6.0	5	56.4	6.1	1.1	-	6.3

We have seen in section 3 that the shared memory is organized into banks. Successive 32-bit words are assigned to successive banks. We implement a bit array using a 32-bit integer array. Therefore successive integers are assigned to successive banks. Each thread executing the kernel enumerates l_s-mers in the input sequence and may set the bits in any of the integer and therefore in any bank resulting in bank conflicts. In order to avoid bank conflicts we repartition the integer array and then reorder the integer array. The integer array, which was once partitioned to fit in the shared memory, is repartitioned into 16 chunks(as there are 16 banks in Tesla). The ith thread in a half warp enumerates the l_s-mers to set the bits in ith chunk. We then reorder the integer array such that the ith thread in a half warp would only access the integers in the ith bank. For example, when $l_s = 6$, each bit array has 4^6 bits and is implemented using an integer array of size 128. We partition the integer array into 16 chunks each of size 8 integers. Figure 2(a) shows the partitioned bit array. The first thread in a half warp(threads 0, 16, 32, ...) only accesses the first chunk i.e. integers 0 to 7. Now we reorder the integers in the bit array such that the integers 0, 1, .., 7 belong to the same bank. Figure 2(b) shows the reordered integer array. It can be seen from the figure that threads 0 and 16 only access the integers in bank 0 and threads 15, 31 only access the threads in bank 15. Therefore there will be no bank conflicts after reordering the integer array.

In addition to avoiding the bank conflicts, repartitioning and reordering has another advantage. Partitioning a bit array into chunks reduces the enumeration length. Because we partition the integer array into 16 chunks, the enumeration length reduces from l_s to $l_s - 2$. Note that the maximum enumeration distance is equal to the enumeration length. For example, when enumeration length is 4, the maximum enumeration distance is 4. So the maximum enumeration distance also decreases by 2. Thus we only need to enumerate to generate $(l_s - 2)$-neighbors instead of l_s-neighbors. This would reduce the registry consumption of each thread and hence we can increase the number of threads per block. Having more threads per block would increase the occupancy resulting in better performance.

5. EXPERIMENTAL RESULTS

We have implemented BitBased on Nvidia Tesla C1060 and Nvidia Tesla S1070 both running at 1.3GHz. C1060 has 30

multiprocessors with 8 scalar processor cores each. S1070 has four GPU devices with 240 cores each. We have tested our code with 20 input sequences of length 600 each. We tested it on random sequences with motifs planted at random positions in the 20 sequences. We have used $n' = 6$ for all our experiments. C1060 and S1070 both have a shared memory of 16KB per processor. As we have described in section 4 we need to find the value of l_s where l_s=max\{i | $4^i n'$ bits of memory can be allocated on shared memory\}. We have found that 6 is the most suitable value for l_s. Table 1 shows the performance results obtained on 1 to 4 GPUs.

We have also experimented the approach using 1 to 120 multiprocessors on Tesla S1070 with only one active block for each multiprocessor and the load is distributed equally among the multiprocessors. It can be seen from Figure 3 that the approach scales well with the number of multiprocessors. We have also collected the results using different number of GPU devices. Figure 4 shows the speed-up of the approach with respect to number of GPU devices. It can be seen clearly that the approach scales well with the increase in number of GPU devices.

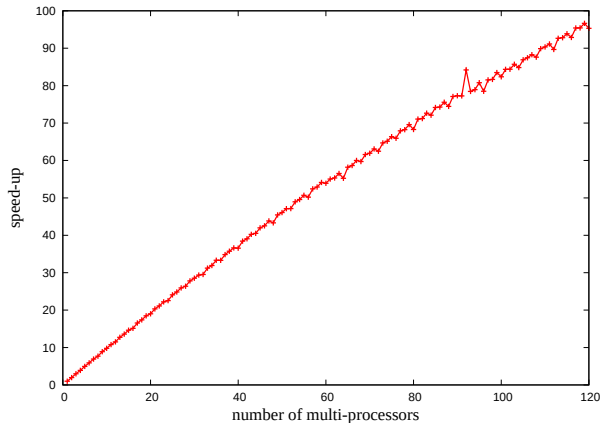


Figure 3: Plot showing the speed-up of the approach with respect to number of multiprocessors.

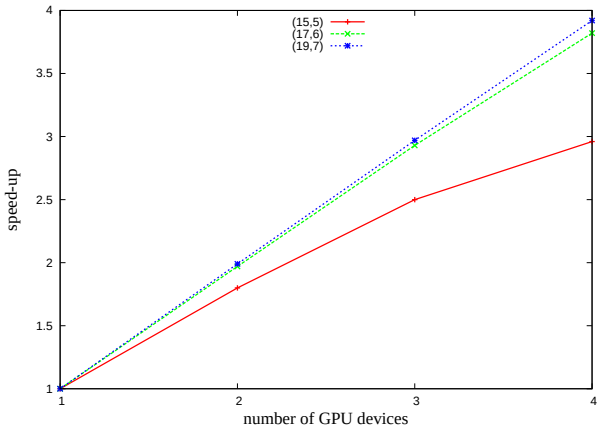


Figure 4: Plot showing the speed-up of the approach with respect to the number of GPU devices.

5.1 Comparison with multicore

The BitBased approach was implemented on a 4 quadcore 2.67 GHz Intel Xeon X5550 machine with a total of 16 cores using 1GB memory. The basic BitBased approach was used for $(15, 5)$ and lower instances and iterative BitBased approach was used for $(17, 6)$ and higher instances. Table 1 shows the results obtained on the multicore machine. It shows the speed-up obtained on GPU with respect to 1 core CPU and 16 cores CPU. The actual results for multicore are discussed in [7]. It can be seen that a single GPU device is 13 to 14 times faster than a single core of Xeon X5550 machine. It performs better than 16 core Xeon machine. 4 GPU devices are 40 to 60 times faster than single core CPU and 4 to 6 times faster than 16 core CPU.

6. CONCLUSION

We presented an efficient parallel approach for solving the planted motif problem on GPU. This approach is modification of a BitBased approach that was originally proposed for Intel based multicore architectures. The BitBased approach had to be modified for GPU architecture. The proposed implementation solves the challenge instance (21,8) of planted problem in 1.1hrs. We are not aware of any sequential or parallel method that will solve this challenge instance in better time. Additionally, to the best our knowledge we are not aware of any previous implementation of a parallel method to solve the planted motif problem on GPU.

7. ACKNOWLEDGEMENTS

This work was partially supported by National Science Foundation grant HRD-0420407.

8. REFERENCES

[1] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[2] J. Buhler and M. Tompa. Finding motifs using random projections. *Journal of Computational Biology*, 9(2):225–242, 2002.

[3] A. M. Carvalho, A. T. Freitas, A. L. Oliveira, and M.-F. Sagot. A highly scalable algorithm for the extraction of cis-regulatory regions. In *APBC*, pages 273–282, 2005.

[4] C. Chen, B. Schmidt, W. Liu, and W. Müller-Wittig. GPU-MEME: Using graphics hardware to accelerate motif finding in DNA sequences. In *PRIB*, pages 448–459, 2008.

[5] F. Y. L. Chin and H. C. M. Leung. Voting algorithms for discovering long motifs. In *APBC*, pages 261–271, 2005.

[6] M. K. Das and H.-K. Dai. A survey of DNA motif finding algorithms. *BMC Bioinformatics*, 8(S-7), 2007.

[7] N. S. Dasari, R. Desh, and Z. M. An efficient multicore implementation of planted motif problem. In *Proceedings of the International Conference On High Performance Computing and Simulation*, pages 9–15, 2010.

[8] J. Davila, S. Balla, and S. Rajasekaran. Space and time efficient algorithms for planted motif search. In *International Conference on Computational Science (2)*, pages 822–829, 2006.

[9] J. Davila, S. Balla, and S. Rajasekaran. Fast and practical algorithms for planted (l, d) motif search. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 4:544–552, 2007.

[10] E. Eskin and P. A. Pevzner. Finding composite regulatory patterns in DNA sequences. In *ISMB*,

pages 354–363, 2002.

[11] T. Ji, K. Gopavarapu, D. Ranjan, B. Vasudevan, C. Sengupta-Gopalan, and M. O'Connell. Tools for cis-element recognition and phylogenetic tree construction based on conserved patterns. In *Computers and Their Applications*, pages 1–6, 2007.

[12] L. Marsan and M.-F. Sagot. Extracting structured motifs using a suffix tree - algorithms and application to promoter consensus identification. In *RECOMB*, pages 210–219, 2000.

[13] NVIDIA. Cuda best practices guide. `http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide.pdf`.

[14] P. A. Pevzner and S.-H. Sze. Combinatorial approaches to finding subtle signals in DNA sequences. In *ISMB*, pages 269–278, 2000.

[15] N. Pisanti, A. M. Carvalho, L. Marsan, and M.-F. Sagot. Risotto: Fast extraction of motifs with mismatches. In *LATIN*, pages 757–768, 2006.

[16] A. L. Price, S. Ramabhadran, and P. A. Pevzner. Finding subtle motifs by branching from sample strings. In *ECCB*, pages 149–155, 2003.

[17] S. Rajasekaran, S. Balla, and C.-H. Huang. Exact algorithms for planted motif problems. *Journal of Computational Biology*, 12(8):1117–1128, 2005.

[18] M.-F. Sagot. Spelling approximate repeated or common motifs using a suffix tree. In *LATIN*, pages 374–390, 1998.

[19] M. Tompa. An exact method for finding short motifs in sequences, with application to the ribosome binding site problem. In *ISMB*, pages 262–271, 1999.

[20] M. Tompa, N. Li, T. Bailey, G. Church, B. De Moor, E. Eskin, A. Favorov, M. Frith, Y. Fu, W. Kent, et al. Assessing computational tools for the discovery of transcription factor binding sites. *Nature biotechnology*, 23(1):137–144, 2005.

[21] L. Yu and Y. Xu. A parallel Gibbs sampling algorithm for motif finding on GPU. *Parallel and Distributed Processing with Applications, International Symposium on*, 0:555–558, 2009.

Scalability of Color-Based Segmentation of Football Players over GPUs

Miguel Ángel Montañés
University of Zaragoza
1 Maria de Luna
Zaragoza, Spain
mmonla@unizar.es

Enrique F. Torres*
Univ. of Zaragoza, HiPEAC
1 Maria de Luna
Zaragoza, Spain
enrique.torres@unizar.es

Jesús Martínez
University of Kingston
Penrhyn Road, Kingston Upon Thames
KT1 2EE, UK
Jesus.Martinezdelrincon@kingston.ac.uk

J. Elías Herrero
University of Zaragoza
1 Maria de Luna
Zaragoza, Spain
jelias@unizar.es

ABSTRACT

In this paper, we study the scalability of a real application to the available number of cores in the *GPU*. Our application is a real-time image processing in which a football player feature extractor based in color patterns obtain feasible measures for tracking system. Since football players are composed for diverse and complex color patterns, a Gaussian Mixture Models *(GMM)* is applied as segmentation paradigm. Optimization techniques have also been applied over the C++ implementation using profiling tools focused on high performance. Time consuming tasks were implemented over NVIDIA's *CUDA* platform, and later restructured and enhanced, speeding up the whole process significantly. Our resulting code is around 4-11 times faster on a low cost *GPU* than a highly optimized C++ version on a central processing unit (CPU) over the same data. The optimized application has been benchmarked over different *GPUs* with different number of cores. Due to data dependencies performance increase 1.4x when doubling number of cores.

1. INTRODUCTION

Real-time image processing systems are specially relevant in Computer Vision. Any advanced image processing application requires a previous extraction of significant features. These features could be used in recognition or tracking systems for several applications. Our proposal is oriented to improve drastically the performance of image segmentation systems. Concretely, we focus on feature extraction and object classification based on those features, not only over prerecorded video sequences but also from live video streaming.

Our method to extract those features consists in an image segmentation according to color information. Segmentation systems are usually a first stage inside an image processing framework. Thus, for instance, results generated by segmentation techniques can be used as input for a tracking algorithm. In the literature, it exists a broad variety of methods for a reliable segmentation of objects in an image. One of the most popular approaches consists in a Gaussian mixture model (*GMM*) in which every object can be represented by one or more Gaussians. This is because most objects are composed of a mixture of different tones associated to a unique color or even of several different colors. Although *GMM* is a successful and broadly used method for feature extraction, its computational cost is a strong handicap for real time applications. The spectacular evolution that CPUs experimented in the past has provided a tool for mitigating the problem. Nevertheless, the progressive slowdown during the last years has stopped this progression whereas it has promoted parallel architectures, such as multi-core, as a solution for increasing the computational power. Unfortunately, most programs are conceived using a serial philosophy. Serial code cannot automatically take advantage of multiple cores to execute itself faster, so that code must be redesigned from a newer parallel point of view.

The *GPU* architecture is optimized for massively parallel processing with peaks up to hundreds of GFLOPS. Recently, in order to take advantage of these high performance computing devices, some extensions to well-known programming languages have been generated, such as *CUDA C* [4]. This language is a set of parallel extensions of the C/C++ programming languages and it is able to interact with a special

*This work was supported in part by grants TIN2010-21291-C02-01, TIN2007-66423 and TIN2007-60625 (Spanish Government and European ERDF), gaZ: T48 research group (Aragón Government and European ESF), Consolider CSD2007-00050 (Spanish Government), and HiPEAC-2 NoE (European FP7/ICT 217068).

hardware interface built into all current NVIDIA *GPUs*.

In the last few years, the amount of scientific application tested over *GP-GPU* has increased [3]. Although generally those researches are focused on specific calculations, they provide an initial idea about the intrinsic potential of this new platform [10]. Particularly, in our field of interest, several studies probe this capability in modern *GPUs* [7]. Traditional methodologies have been implemented, such as pattern recognition algorithms based on textures [6], Gaussian mixture models [9] or image feature extraction techniques [13, 15]. All these examples give an idea of the increase of efficiency that can be achieved thanks to these devices.

In our research, we have developed an application which is able to detect football players in a video sequence. Once they are extracted from background, each player is classified into any of the teams. For classification purposes, a color-based method is employed. Our election has been Expectation Maximization for Gaussian Mixture Models [9]. Since one of our main objectives is to process multi high-resolution cameras, detection and classification processes must be applied on real time in a extremely efficient manner. In order to achieve that, we have adapted and implemented those tasks over *GPU* platform taking advantage of its high parallel computational capability (Section 5).

The evaluation of our implementation has been made over a set of different low cost *GPUs* with 16, 32 and 64 cores to study the scalability of the implementation. These tests have also been run under different CPUs, to clarify as much as possible the real contribution of our implementation.

The outline of the paper is as follows. In Section 2 the hardware infrastructure is described. Section 3 introduces the stages that compose our application and discusses their computational cost. Section 4 explains the computational cost of a prototype implemented in an optimized version in C++ running in a conventional CPU. In section 5, the parallelization as well as the CUDA implementation are detailed. Section 6 presents a comparison between CPU and *GPU* results and its scalability. Finally, conclusions are presented in Section 7.

2. INFRASTRUCTURE

Our approach consists of the processing and classification algorithms for football players in sequences provided from one or multiple cameras, which are installed in a real football stadium. In our infrastructure, we propose a system composed of 8 static high definition digital cameras (resolution 1388x1036) with overlapping fields of view. The cameras are positioned around the stadium as is shown in fig. 1.

This camera distribution has been done in this way because the minimum number of cameras for covering the football field with enough resolution is 8 and the overlapping cameras are crucial to solve occlusions, specially in conflictive areas.

In order to check the performance improvement and scalability that our implementation achieves, we have tested the algorithm over different types of processors and *GPUs*. Thus, three different types of PCs are available for scalability study and a fourth PC is used to confirm results. These equipments are shown in table 1. These equipments are close to the average current processors, giving us a significative sampling of the market. On the other hand, 4 different *GPUs* have been tested too keeping same philosophy. They also fulfill another requirement: since our implementation employs atomic functions to obtain synchronism, we need video cards compatible with *CUDA Compute Capability 1.1* or higher. For every possible combination of both platforms (CPUs and *GPUs*), a scalability study was made.

	Micro	GHz	nVidia	Cores	Bandwidth (GB/s)
PC1	Core 2 T7500	2.2	Geforce 8600M GS	16	6.7
PC2	Core 2 T8900	2.4	Quadro FX 1600M	32	11.2
PC3	Core 2 Quad	2.83	Quadro FX 1800	64	38.4
PC4	Core i7 Quad	2.8	Geforce GTX260	216	111.9

Table 1: Different types of CPUs and GPUs for testing

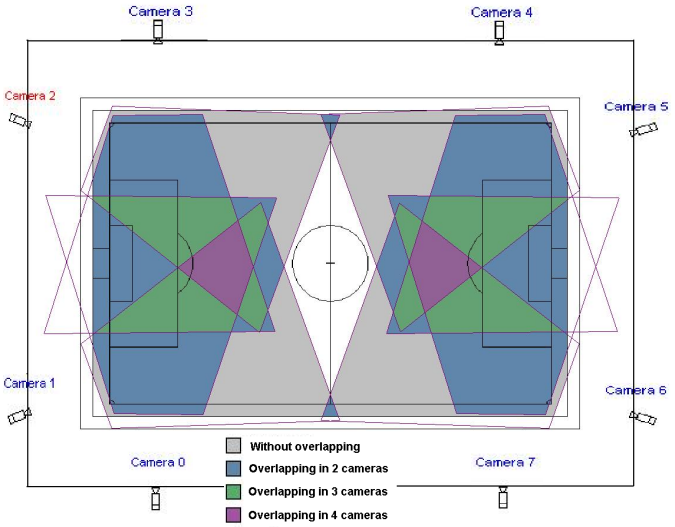


Figure 1: Camera distribution on the roof

3. DESCRIPTION OF APPLICATION

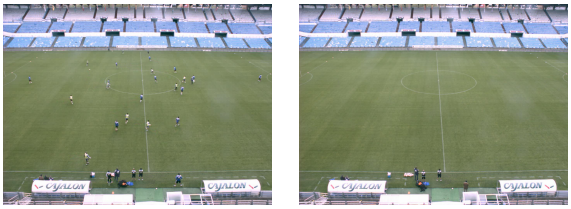
The proposed classification algorithm can be decomposed into a set of steps. Most of them should be done per camera. The steps and input data that they require are described next.

3.1 Independent processing per camera

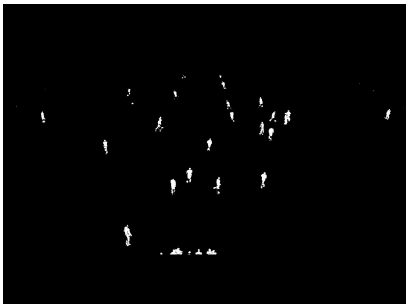
- Image Capture: at this stage, images are retrieved on demand from each camera.
- Color Space Transformation from Bayer to RGB: high-resolution cameras usually provide images in raw format (also called Bayer-type RGGB [2]), i.e. 8 bits per pixels for color codification. To obtain a RGB image, we need an intermediate transformation process

called BayerToRGB. RGB values which match up in the RGGB sequence are mapped directly, while other channels are calculated as an arithmetic mean of all neighbors corresponding to the same channel.

- Motion Detection: it consists in a thresholded subtraction between the current image (fig. 2 a)) of every camera and a pre-generated image of the scenario, called *background* (fig. 2 b)). Process is shown in fig. 2 c). Motion detection image contains the dynamic areas, which will be used for posterior processing like distracter removal.
- Color Space Conversion *RGB* to *HSV*: under variable illumination conditions, better segmentation results can be obtained by applying a transformation in the color space [14]. Instead of RGB, HSV (Huge, Saturation, Value) has shown a better accuracy.
- Blob Labelling: it is the algorithm that seeks connected areas, called *blobs*, in the resulting image of the previous step. By grouping pixels into *blobs* and assigning a common label we simplify the posterior tracking stage.
- Color Segmentation: this procedure tackles the problem of identifying different areas of the image. *GMM (Gaussian Mixture Model)* has been chosen as paradigm, which implies a preliminar training by extracting color features from regions of interest. Thanks to this technique, a distinction into three groups is obtained: *player of team 1, player of team 2 and noise from the background*.



a) Current image b) *Background* image



c) Motion Image, thresholded subtraction between the current image and *background*.

Figure 2: Current image, *background* image and substraction result image.

Fig. 3 details the processing flow per camera. Output generated from previous stages is used as input for a tracking algorithm in order to ensure the temporal coherence. Although it is out of the scope of this paper, a *Multi-Camera Uncensted Kalman Filter (MCUKF)* [8] has been used to

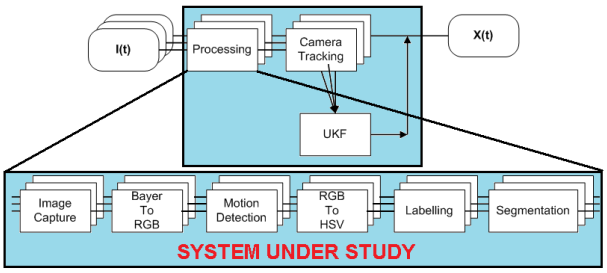


Figure 3: Processing schema

demonstrate the global feasibility. Empirical experiments allow us to conclude that a successful tracking can be obtained with a processing frame rate between 8 and 15 per each camera and to process 8 cameras we need more than 64 images per second for real time.

3.2 Gaussian Mixture Method for Image Segmentation

In collaborative sport applications, it is known a priori that both teams, as well as background, are defined by clear and distinctive color patterns in their clothing. These color patterns can be easily modeled by parametric methods.

GMM is a method that allows a reliable object modeling and image segmentation even in presence of complex targets, which can be composed of multimodal appearance distributions. Since it is a parametric technique, it needs a off-line training phase to calculate those parameters. Training results are used afterwards in classification *On-line* stage.

The simplest technique to model the appearance coefficients consists in assuming the target as a monochrome region and modeling it as a Gaussian using only two parameters: mean μ and covariance σ^2. Although this assumption limits the generality of the methodology, it can be easily extended by dividing the target into a predefined set of monochrome regions [12].

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2\sigma^2} \quad (1)$$

$$p(x) = \sum_{i=1}^{N} w_i \frac{1}{\sqrt{2\pi}\sigma_i} e^{-(x-\mu_i)^2/2\sigma_i^2} \quad (2)$$

Both off-line training and on-line classification are composed of different phases:

1. Off-Line Computing consist of a sample selection in order to supervise sample selection for every group (*team 1, team 2 and background*), parameter tuning to adequate number of Gaussians used to model every group and training.

2. On-Line Computing is composed of a classification step in which every pixel is classified into one of the different groups. For this, it is required to HSV conversion, computation of the distance between the pixel candidate and the different model of every group, final decision based on minimum distance and probability computation to measure the membership degree to every group. This last process generates, as result, probability images [5] that can be used to improve the tracking quality based on stochastic approaches.

As the offline stage is only applied once at the beginning and under supervision, it can be considered out of the real-time system and, therefore, its implementation is not required over a *GPU* platform for our goals. However, this process is also amenable to be implemented using the *GPU*, as it was demonstrated in [9], obtaining excellent results.

4. CPU IMPLEMENTATION

A single thread implementation of the algorithm was made in C++ language running under Windows.

For this optimization process, performance analysis tools, such as *Intel VTune Performance Analyzer* were applied to identify the possible *hotspots*. This tool aimed at increasing performance, in addition to the location of hotspots, allowed us to perform a deep analysis of them. Thus, *Intel VTune Performance Analyzer* let us to detect, to re-code and to optimize our implementation, improving the performance substantially. An important difference must be noticed between those general optimizations and those that act on specific parts of the code.

- General optimizations: they improve the global performance of the application. Our main optimizations consist in:

  1. Usage of a specific optimizing compiler, such as *Intel C++ compiler*. Full optimization and specific architecture compilation flags are both used in this implementation.
  2. Classical Code Optimizations [1]. It is crucial to take into account the memory mapping of data structures. In this way, we ensure a high success rate in the access to cache memories. For that, input data must be stored consecutively in memory as often as possible, i.e. data are stored in memory in raw order. Therefore, in loops, image data have to be accessed by rows. Thus, data access obtains high cache hit. Access by columns fails in cache because data are not consecutively in memory.

- Specific optimizations: they improve the performance of given functions. The most important of them is the use of **Look-up Tables or *LUTs***. Those functions with a clear and repetitive pattern, such as color classification, can be replaced for a storage in memory of all possible result for any input combination. This resulting matrix is called *Look-up Table* (*LUT*). An example is color space conversion. For each RGB value, the classification result is calculated and stored in LUT. After its generation, the expensive calculation is replaced for a memory access to the right memory slot, which implies a substantial boost of the efficiency. The more complex the operation is, the more efficient this technique is. For our particular case, calculations for determining the segmentation of a pixel costs 39.96ns, whereas the memory access to check the value in the LUT is 6.02 ns, which implies a speed-up x6.63. Although generating the LUT implies a fixed cost of 670.48 ms, it can be done off-line since the color model is usually static.

Stages	Time (ms)	% of total
Conversion Bayer to RGB	15.7	10.34
Motion Detection	9.83	6.47
Conversion RGB to HSV	35.61	23.45
Labelling	5.3	3.49
Segmentation	85.39	56.24

Table 2: Time of different stages over optimized Intel C++ compilation [ICC] and percentage time of them over total time.

In table 2 is shown that any implementation using Intel C++ [ICC] achieves good results.

$$Rate_{Intel}(fps) = 6.58 fps$$

In spite of this considerable improvement, *Conversion RGB to HSV* and *Segmentation* stages still remain as critical bottle necks. Therefore, in order to raise the performance and to be able to achieve our goals, a more powerful tool is required. Moreover, other stages that were not so critical *a priori*, like *Conversion RGB to HSV*, have acquired now a more important role. It is because of this reason that stages shown in Table 2 have been implemented on the *GPU* platform, with a special focus on the *Segmentation* stage.

5. GPU IMPLEMENTATION

The hardware architecture of a system with a *GPU* can be seen in fig. 4. A *GPU* is a hardware device connected to the main system through a fast bus, second-generation PCI Express currently. It has some very specific processing features allowing to take advantage over the current CPUs.

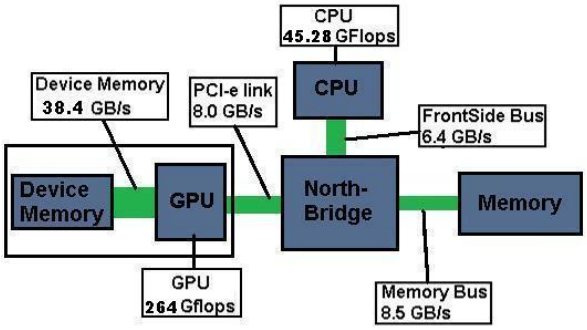


Figure 4: Hardware architecture of a system with *GPU*

Specifically, the features that make *GPUs* specially powerful in massively parallel computing are:

1. Hardware composed of several computing functional units and several multicores.
2. In single precision floating point, a *GPU* can reach up to 500 Gflops owed to the 30-50 Gflops of conventional CPUs.
3. It has a high bandwidth to the internal memory of up to one order of magnitude higher than the bandwidth of a CPU and system memory (about 86.4 GB/s in a *GPU* versus 8.5 GB/s in a CPU).

4. In order to take advantage of such high bandwidth, *GPUs* allow several memory access operations to run simultaneously.

5. Paradigm *Single Instruction Multiple Thread*, SIMT, is used by the *GPU*. This specific execution allows and needs many independent and simultaneous active threads that execute the same instructions over different data. All of them are running into a unique kernel at the same time.

Attending *GPU* characteristics and SIMT paradigm, a preliminary study of our application is needed. Different criteria have been used in this analysis: Computational cost and redesign of several algorithms for massively parallel computing.

5.1 Preliminary Study

In this section, the adequacy of each stage to be implemented on *GPU* has been analyzed. The *CUDA* implementation was tested using PC3 (see Section 2 table 1) obtaining the results shown in fig. 5, where we can conclude:

- **Conversion Bayer to *RGB*:** this stage requires, for every pixel, access to the neighbor pixels in order to calculate the resulting RGB. The processing is made per pixel independently, although the final result also depends on the adjacent input values. Therefore, there is no easily adaptable and massively parallelizable implementation due to a dependency among the instructions data. In spite of pixelwise calculation, *Conversion Bayer a RGB* stage presents several dependencies in its data. *CUDA* implementation has to be carefully studied because time is higher in *CUDA* implementation as is depicted in fig. 5.
- **Motion detection:** Since it is basically a pixelwise subtraction, there is not dependency with the neighbor pixels and a new thread per pixel can be launched independently. *Motion Detection* and *Conversion RGB to HSV* stages prove a good behavior when they are implemented over *CUDA*. This results are obtained because in this phases the computation is realized pixel by pixel and dependency data is very scarce. Time cost is reduced considerably.
- **Conversion *RGB* to *HSV*:** in the same way as the previous stage, processing is pixelwise but there is no data dependency regarding the neighbor pixels.
- **Blob Labelling:** this algorithm searches for connected zones in the image. The nature of the connectivity search produces a strong dependency among neighbors. There is not a simple parallel solution and a new algorithm should be developed to take advantage of the available features. *Labelling* stage is not parallelizable and our designed algorithm for *GPU* has a deficient behavior. Its computation time has increased.
- **Color Segmentation:** it is also a good candidate to be implemented on *GPU* as computation does not have dependency with the neighbors and it implies a substantial part of the total time. It can be decomposed into three substages: resulting image calculation by consulting the corresponging *LUT*, *LUT* update for the next frame and noise filtering by morphological operators. *CUDA* implementation of *Segmentation* stage presents a significative improvement.

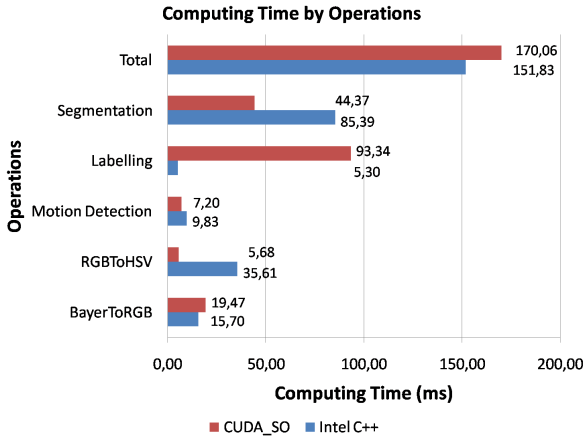


Figure 5: Computational cost for [ICC] and *CUDA* (over PC3) implementations.

Because of data shown in fig. 5, relevant decisions can be taken. Analyzing fig. 5, it can be observed that *segmentation*, for being the most expensive stage, must be analyzed carefully. This stage takes between 58.27% to 57.8% of computing time (without taking into account labelling time) in [ICC] or *CUDA* implementations respectively. Since *Conversion Bayer to RGB* stage takes between 25.37% (in PC3) of the total time in *CUDA* implementation and its data dependencies detected, it needs a special optimization. For *Motion Detection* and *Conversion RGB to HSV* stages, data independence provides margin to get better.

A critical design phase is the labelling computing, since it is not parallelizable. Since labelling becomes a expensive stage in *GPU* as observed in fig. 5, it is worthy to take special care in aspects as kernel context switch or data transfer with CPU, avoiding unnecessary waste of time. Three solutions must be studied:

- **Option 1:** All the stages are run on the *GPU*: Labelling allows identifying active areas in the image, reducing the segmentation to those areas and making unnecessary segmenting the rest of the image. Total computational cost would be $T_{total_1} = T_p + t_{e_{gpu}} + t_{s_{blob}}$, where T_p is the time due to the pre-labelling stages, $t_{e_{gpu}}$ is the labelling cost in *GPU* and $t_{s_{blob}}$ is the segmentation cost on the active areas.
- **Option 2:** Previous stages to labelling are run on *GPU*, results are transferred to the host, which runs the labelling and returns the result to the *GPU*, where the segmentation is done on the active areas. $T_{total_2} = T_p + t_{totaltrans} + t_{e_{cpu}} + t_{totaltrans} + t_{s_{blob}}$, being $t_{totaltrans}$ the transference cost + kernel commutation cost + driver access cost.
- **Option 3:** All the stages are run on *GPU* an the labelling is eliminated. This implies that segmentation is applied to the whole image and not only over active areas. $T_{total_3} = T_p + t_{s_{image}}$.

In each kernel switch or data transference, the *CPU* needs to access the *GPU* driver to complete the operation, which implies an additional time.

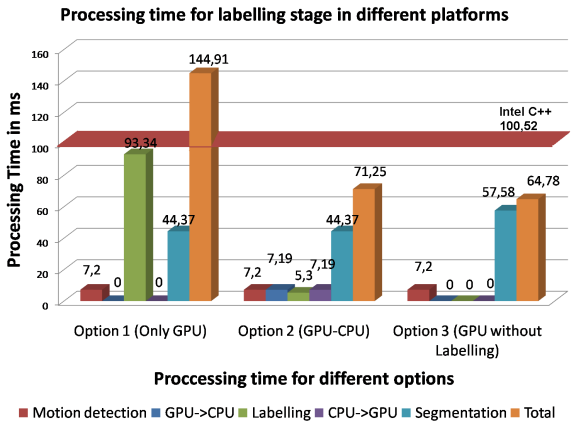


Figure 6: Comparison: First *CUDA* (over PC3) implementation versus optimized C++

1. The computational cost of transferring data $CPU \Rightarrow GPU$ *or* $GPU \Rightarrow CPU$ is around 7.19 ms. By running as many instructions as possible inside the *GPU*, just two transfers should be needed: to introduce input data and to obtain the results.
2. For each kernel switch, the *GPU* requires extra time for changing the context. By grouping different stages in a shared *kernel*, we save this extra time.

Therefore, every single operation of any type that we could run into the *GPU* will avoid to waste time unnecessarily. Thus, it is a good practise to design stages which could run into the same kernel.

Previous options have been tested and results are shown in fig. 6 from PC3. By minimizing the computational cost (T_{total_1}, T_{total_2} and T_{total_3}), the optimum decision can be taken. As fig. 6 shown, option 3 provides the optimum solution (64.78 ms) in comparison with the other alternatives whose costs are 144.91 and 71.25 ms. As fig. 6 shows, option 1 is even more expensive than [ICC] implementation whose processing time is about 100.52 ms. Because the extra data transfers and the kernel context switching, option 2 is worse than option 3 although the whole image is segmented in the last one. In the light of previous results, we can conclude that *Blob Labelling* is not efficient for parallel computing and, in case it would be necessary for the posterior stages such as tracking or distracter removal (football field lines), must be relegated to the CPU. Taking this decision as a new starting point, the next step consists in the optimization of all the stages.Therefore option 3 has been selected, *Labelling* stage is relegated to CPU (if it is needed) and *segmentation* is applied over the whole image.

5.2 Techniques for optimizing GPU code

Several techniques are at our disposal for an optimum use of *GPU* capacities according to recommended methodologies [11]. Across all the stages these techniques have been evaluated. A *GPU* is a device designed for highly parallel computation having a very high number of functional units and a large memory bandwidth. Therefore, the main techniques for increasing performance are based on keeping up the occupation of functional units (known as occupancy) and maximizing the use of effective bandwidth to memory. Next, the most effective ones are described.

5.2.1 *Occupancy*

Occupancy is measured by the number of threads assigned to each processor. Maintaining a high occupancy in the *GPU* is important to performance due to it can be achieved by means of two different ways: through the number of registers and through the amount of *shared* memory employed.

As a general rule, the less the number of register used per kernel, the higher occupancy. However, it is worthy to note that this modification is not always easy since it strongly depends on the algorithm and could imply a deep restructuring.

By analyzing one of the segmentation substages and restructuring an indexing instruction for memory allocation, we were able to save 2 registers per kernel. This complex reduction implies the core occupancy has gone from 66% to 100%

5.2.2 *Coalescence*

Coalescence is a technique for optimizing memory accesses. Memory accesses from different threads can be merged into a single access if the required conditions are fulfilled [4]. This fusion process is known as coalescence. Coalescence is defined as a mean to gather several simultaneous memory accesses in parallel. It is promoting during the global memory accesses.

Coalescence is, without doubt, the most powerful method for optimization in *GPU*. It consists in a mechanism that fuses into a unique operation all read/write accesses from the running threads in the current active block. *GPUs* have specific hardware that detects and makes this fusion, allowing to hide the high latency of threads accessing to local or global memory when cache is not available, and improving the speed-up above two orders of magnitude for these operations.

This technique is specially relevant in the following stages, although it has been applied across the whole system: conversion Bayer to RGB: 100% coalescent on writing and on some reading, conversion RGB to HSV: 100% coalescent on both writing and reading, motion detection: 100% coalescent on both writing and reading and color segmentation: $\approx$ 10% coalescent in substage 1, $\approx$ 30% coalescent in substage 2 and 100% coalescent in substage 3.

5.2.3 *Others Techniques*

Other techniques to achieve improvement in *GPU* are:

- Masking of high latency memory accesses: this can be achieved by sending non data-dependant instructions to the processing units during the transference cycles.
- Avoiding branch divergence: when several threads should take different paths, it is called divergence and the execution times of all the branches become serialized, increasing the cost for every divergent thread.

	CPU	GPU	Speedup
PC1	4.11	8.66	2.11
PC2	5.33	12.37	2.32
PC3	6.58	22.45	3.41
PC4	5.94	63.38	10.67

Table 3: Final results (in number of frames)

6. RESULTS AND SCALABILITY TEST

As our system is composed of identical high definition cameras (1388x1036), we will only analyze the processing time for one of them. Later, we could extrapolate results to work out the scalability of our processing kernel.

A scalability study aims to assess the performance of our algorithm as a function of the number of images, the number of cameras or the computational power. To this end, we have processed the algorithms on several computers that have been selected on the basis of different criteria:

1. The CPUs will be of mid-high range because it seeks a significant increase in computational power.
2. The first 3 GPUs have been chosen with the criteria of having a number of cores that is a multiple of the number of cores of the previous GPU. The aim is to study the evolution of the cost of processing each of the phases and the global system.
3. The fourth *GPU* is chosen to confirm the tendency showed in the previous tests as this section describes.

Thus, the chosen configuration for each experiment is as is shown in Section 2 table 1.

Analyzing results from the application point of view (see Table 3), a considerable speed increase has been obtained (10.67x), being possible to process 63.38 frames per second with a Geforce GTX 260 versus the 5.94 that CPU4 could. A comparison GPU - CPU in PC1 shows that achieved improvement is around 2.11x, since this CPU processed 4.11 fps and its GPU processed 8.66 fps. Values in comparison GPU - CPU in PC2 achieve 5.33 fps in CPU2 and 12.37 fps in GPU2 resulting in a speed up of 2.32. Values in comparison GPU - CPU in PC3 achieve 6.59 fps in CPU3 and 22.45 fps in GPU3 resulting in a speed up of 3.41.

In the same manner, a comparison among the time cost evolution of different stages and process ratio in fps over different equipments has been extracted (see fig. 7 and 8).

In these figures, results using the 4 *GPUs* with 16, 32, 64 and 216 cores are depicted. Two comparative analysis can be done: evaluating the time cost for every stage for each GPU or comparing the global performance of the application using the 4 different CPUs against the GPUs measured in frames per second, fps.

Analyzing in the stage level (fig. 7), it is important to note that improvement increase with *GPU* performance, almost always proportional to the number of cores. The only exceptions are the conversion Bayer to RGB and segmentation stages, where input data dependence produces a slightly lower rate (see fig. 7). Global improvement has an almost

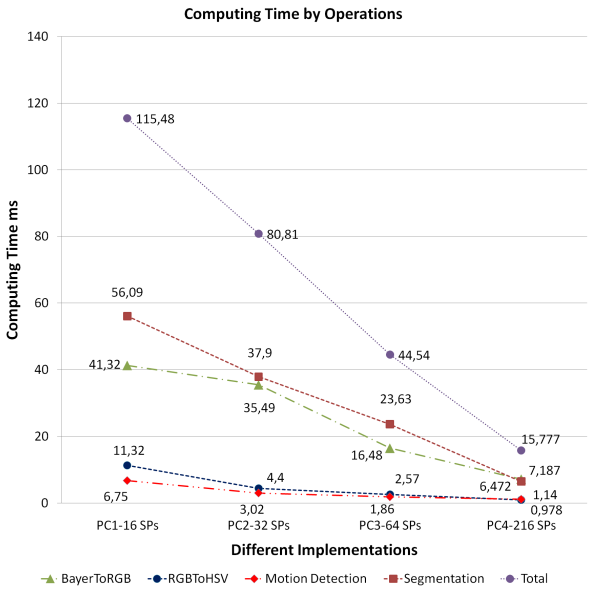


Figure 7: Stage computing time using different *GPU* models.

linear tendency achieving an execution code 7.32 time faster in GPU4 than in GPU1.

In fig. 8, results are compared in the application level between three GPU-CPU configurations, and the same tendency can be appreciated. A very low-cost laptop equipped with GPU1 is able to obtain enough processing ratio in fps to connect a tracking stage (8 fps or more). Nevertheless a highly optimized implementation in a medium PC as PC4 is not able to do that without the *GPU*.

It is also worthy to note some characteristics of the three different equipments under test. Despite the fact that the pair CPU-GPU are contemporary, the evolution of both architectures are not equal over time. CPU power increase in the last two years is really smaller in comparison with GPUs in the same period. This can be explained due to the maturity of both technologies and the improvement margin.

It has to be noticed how a low-cost *GPU* as Geforce 8600M GS with only 16 cores takes advantage over a medium-high CPU as Core i7 (8.66 fps versus 5.94 fps, respectively), being 1.46 times faster in global processing.

Finally, note that the speed-up increase with the number of cores is constant although not in the same proportion. This difference is mainly due to the overhead of the transference time CPU $\Leftrightarrow$ *GPU*.

Given that a minimum processing rate of 8 fps is required for a posterior tracking stage and that we need to process 8 cameras, it is necessary a minimum processing rate of about $8 * 8 = 64 fps$. Thus, the scalability can be obtained as:

- PC type 3 processes 6.58 fps per camera using CPU3, so we need 10 medium-high PC.
- A *GPU Quadro FX 1800* (*GPU3*) processes 22.45 fps per camera, so we need 3 low-cost *GPUs*.

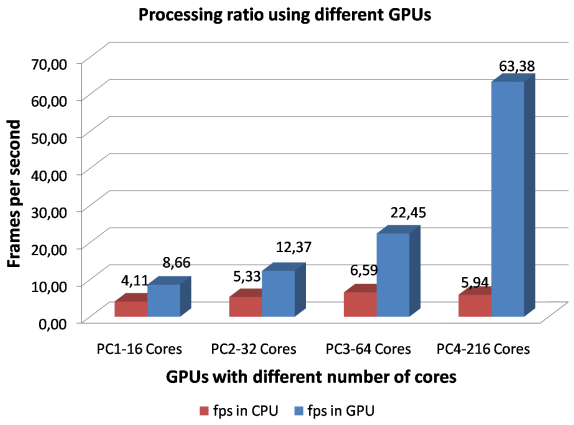


Figure 8: Ratio in frames per second for different *GPUs* in comparison with the three available CPUs.

In addition, since the performance increase is ∼1.4x when the number of cores doubles, we could extrapolate that a machine with a *GPU* with a triple number of cores, could process the 64 fps needed over only one equipment.

This extrapolation has been confirmed in a experiment over a Geforce GTX 260 while price is around 150 dollars. Results show a processing ratio around 64 fps proving that our scalability study is correct.

7. CONCLUSIONS

In the light of these results, we can assert a set of interesting conclusions:

- Usage of high-capability computing devices, such as *GPUs*, have a potential for this kind of applications. It has been possible to segment football players in real time by making an efficient use of these platforms. We are able to improve all the processing stages, with the exception of labelling, with speed-ups up to 40x and using medium-cost hardware. The global performance improvement is x10.67 making possible a processing rate of 63.38 fps instead of the 4.11 fps in low-medium PC (PC1), 5.33 fps in medium PC (PC2), 6.58 fps in medium-high PC (PC3) or 5.94 fps in medium-high PC (PC4).
- We have been able to make the functionality independent of the scalability. Therefore, we have proved that a single but more powerful card would be able to process our 8 cameras.
- Even optimizing the *GPU* occupancy and the effective memory bandwidth using coalescence, scalability is affected by the data dependencies.

8. REFERENCES

[1] D. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26:345–420, 1993.

[2] B. E. Bayer. Bayer. United States Patent, 1975. http://en.wikipedia.org/wiki/Bayer_filter.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using cuda. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008.

[4] N. Corp. *CUDA 2.0 Programming Guide*. NVIDIA, 2008.

[5] J. M. del Rincón and C. O. Uruñuela. *Feature-based human tracking: from coarse to fine*. PhD thesis, Zaragoza, University of Zaragoza, Zaragoza, Dic 2008. Presented: December 2008.

[6] J. Fung and S. Mann. Using multiple graphics cards as a general purpose parallel computer: Applications to computer vision. In *ICPR '04: Proceedings of the Pattern Recognition, 17th International Conference on (ICPR'04) Volume 1*, pages 805–808, Washington, DC, USA, 2004. IEEE Computer Society.

[7] M. Garland, S. L. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel computing experiences with cuda. *Micro, IEEE*, 28(4):13–27, September 2008.

[8] J. R. Gómez, J. E. Herrero, C. Medrano, and C. Orrite. Multi-sensor system based on unscented kalman filter. In *IASTED*, pages 13–18. In Proc. Image Processing (VIIP), 2006.

[9] N. S. L. P. Kumar, S. Satoor, and I. Buck. Fast parallel expectation maximization for gaussian mixture models on gpus using cuda. *High Performance Computing and Communications, 10th IEEE International Conference on*, 0:103–109, 2009.

[10] P. Lu, H. Oki, C. Frey, G. Chamitoff, L. Chiao, E. Fincke, C. Foale, S. Magnus, W. McArthur, D. Tani, P. Whitson, J. Williams, W. Meyer, R. Sicker, B. Au, M. Christiansen, A. Schofield, and D.Weitz. Orders-of-magnitude performance increases in gpu-accelerated correlation of images from the international space station. *Journal of Real-Time Image Processing*, 2009.

[11] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[12] P. Pérez, C. Hue, J. Vermaak, and M. Gangnet. Color-based probabilistic tracking. In *ECCV '02*, pages 661–675, London, UK, 2002. Springer-Verlag.

[13] S. N. Sinha, J. Frahm, M. Pollefeys, and Y. Genc. Gpu-based video feature tracking and matching. Technical report, In Workshop on Edge Computing Using New Commodity Architectures, 2006.

[14] A. R. Smith. Color gamut transform pairs. In *SIGGRAPH '78: Proc. of the 5th annual conference on Computer graphics and interactive techniques*, pages 12–19, New York, NY, USA, 1978. ACM.

[15] T. Tuytelaars and K. Mikolajczyk. Local invariant feature detectors: a survey. *Found. Trends. Comput. Graph. Vis.*, 3(3):177–280, 2008.

Fluid Simulation With CUDA Using the Lattice Boltzmann Method

Andreas Monitzer
University of Applied Sciences Technikum Wien
monitzer@technikum-wien.at

ABSTRACT

Recent developments in programmability and performance of graphics processing units have enabled GPU programmers to simulate various physical phenomena in real-time that are calculated in weeks on classic CPU-based architectures. One of these phenomena are fluid simulation, which allow simulation of water, wind and weather among many others. The Lattice Boltzmann method fits very well into the GPU architecture as exposed by NVidia's CUDA, which will be shown in this paper. An adaptation to CUDA using the methods for handling complex obstacles presented here has not yet been demonstrated to the best of our knowledge, and so this paper serves as a demonstration that the Lattice Boltzmann method can be executed efficiently on a GPU, and outlines the specific adaptations required for a CUDA-based implementation.

1. INTRODUCTION

Graphics processing units (GPUs) on current graphics cards are generic programmable stream processors. This class of processors is designed for parallelizable algorithms that do not make heavy use of branching. Algorithms having these properties can be accelerated significantly compared to implementations on current central processing units (CPUs). Additionally, GPUs do not suffer from caching issues, since they have very closely defined input and output streams and are optimized at the hardware level for this configuration.

Grid-based fluid simulations are an obvious choice for GPU-based calculation, since operating on the cells of a grid is easily parallelizable. However, care has to be taken to avoid slowdowns caused by making inefficient use of the card's features.

Applications for real-time fluid simulations include metrology, games and medical purposes. It especially enables real-timer interaction of the user with the simulation, for example using a stick to stir water.

Section 1.1 introduces the traditional method of simulating fluids, while section 2 outlines another method better suited for the task at hand. Section 3 explains how to use graphics cards as general purpose processing units and how to apply this knowledge to fluid simulations. Section 4 enhances the fluid simulation by adding obstacles, and Section 5 evaluates the implementation. Section 6 concludes this paper.

1.1 Previous Work

The first attempts at simulating fluids in computer graphics were using wave-based approximations that do not allow interactivity with the fluid [5], but give a very realistic impression used in many computer games for panorama views. Wejchert and Haumann [20] implemented more complex two dimensional flows by assembling them from well-known primitives like vortices and sinks.

Chen and da Vitoria Lobo [3] introduced the Navier-Stokes equations (NS) to the graphics community, which are directly derived from Newton's second law, as a method for simulating flows even at interactive rates. They allow calculating the fluid movements at arbitrary detail, and are suitable for describing many different phenomena, like water, clouds, smoke, foams, and even motion of stars inside a galaxy.

The basic formulation of the NS for incompressible fluids is [3, 15]:

$$\nabla \cdot u = 0 \tag{1}$$

$$\frac{\partial u}{\partial t} = -(u \cdot \nabla)u - \frac{1}{\rho}\nabla p + \nu \nabla^2 u + f \tag{2}$$

where $\cdot$ denotes a dot product between vectors, ∇ is the vector of spatial partial derivatives, u and p are the velocity and pressure field of the fluid respectively, ρ is the density and ν is the kinematic viscosity. f is a vector representing external forces.

The NS is inherently dimensionless. In practice, this means that both two dimensional and three dimensional solutions are possible. Since calculating the equations for three dimensions is computationally more expensive, Chen and da Vitoria Lobo [3] solved them for two dimensions only, and then used the pressure field p as a height map (higher pressure results on more displacement of the mesh from the ground at

that point). The justification given is that higher pressure at the base of a fluid results in taller columns of the surface above, due to the incompressibility of the fluid. Krüger and Westermann [7] propose using multiple layers of two dimensional fluids and interpolate between them to get a more realistic look.

The difficulty imposed by Equation (2) is the part on the left side of the equal sign, $\frac{\partial u}{\partial t}$. This is non-linear, and thus can not easily be solved. Various solutions have been proposed, which is outlined below.

Chen and da Vitoria Lobo [3] used a finite-difference solution to create an iterative solver for the NS.

Stam [15] emphasizes the importance of stable calculations. When the time steps used for calculating the NS are too large (also limited by other factors like the size of the domain or the viscosity), the simulation "blows up". This effect is non-linear and causes small errors in the simulation to amplify due to numerical reasons. To avoid this, Stam used a method called *method of characteristics* using a semi-Lagrangian solver, which is unconditionally stable. However, the simulation suffers from too much numerical dissipation [14], which means that this method is only suitable for situations where the fluid simulation is only used as a visual effect.

Liu et al. [10] implemented the solver introduced by Stam [15] on the GPU in three dimensions. It slices the third dimension of the domain into multiple planes, which are then tiled into a two dimensional texture. Scheidegger et al. [14] used a different method called "Simplified Marker and Cell", which uses an explicit solver, that means it is subject to certain time step limitations to maintain a stable simulation.

The first attempt at mapping the Lattice Boltzmann Method to the GPU was documented by Li et al. [9], but uses the shader programming language, which was not designed for tasks like this. Thus, some additional steps have to be taken, like using multiple shaders for a single simulation step, which reduces the maximum possible throughput due to the higher amount of memory access.

2. FLUID SIMULATION USING THE LATTICE BOLTZMANN METHOD

In 1872, the Austrian physicist Ludwig Boltzmann developed the Boltzmann equation, which is a mathematical model to describe the dynamics of an ideal gas at microscopic scale.

If this equation would be applied directly, every single molecule of the gas would have to be stored and simulated (using its position and direction). Calculating these would be unrealistic today, due to the limitation posed by processors and memory. Thus, simplifications were developed.

2.1 The Lattice Boltzmann Method (LBM)

In this method, the simulation domain is split into discrete cells, forming a lattice. Every cell stores a molecule distribution function by a varying number of values f_i, which denote the amount of fluid molecules traveling in this cell in a certain direction e_i. The velocity u and density value ρ of a cell as known from the NS can be calculated by [8]:

$$\rho = \sum_i f_i \qquad u = \frac{1}{\rho} \sum_i f_i e_i \tag{3}$$

Calculating the simulation is split into two phases: Streaming and collision. In the streaming phase, the molecule distribution f_i is copied one cell into the direction e_i. In the collision phase, a new distribution is calculated based on the information available solely in this cell. The formulas used for this are outlined here:

After discretizing and simplifying the Boltzmann equation, the following equation can be derived:

$$f_i(x, t + \Delta t) - f_i(x, t) = \Omega_i \tag{4}$$

where Ω_i is a fluid collision operator to be determined.

In 1992, a simplified collision operator called the *Bhatnagar-Gross-Krook approximation* was introduced to the LBM [17]. It uses a single relaxation time approximation to reduce the operator to operations suitable for computers. It is based on the idea that the main effect of the collision operator is to bring the molecule distribution closer to the equilibrium distribution, which is defined as

$$f_i^{eq} = \omega_i \rho \left(1 - \frac{3}{2}u^2 + 3(e_i \cdot u) + \frac{9}{2}(e_i \cdot u)^2\right) \tag{5}$$

where ω_i is a constant that depends on the lattice geometry (more on that later, see Table 1). The collision operator itself is defined as

$$\Omega_i = -\frac{\Delta t}{\tau}(f_i(x, t) - f_i^{eq}(\rho, u)) \tag{6}$$

where τ is a constant that represents the viscosity ν of the fluid, given by $\tau = \frac{1}{2}(1 + 6\nu)$ [19].

Unlike most methods based on the NS, the LBM is unconditionally stable, while still demonstrating fluid behavior. The only limitation is that information in the grid cannot travel faster than one cell distance per streaming phase (usually called c_s, speed of sound or "Mach number").

2.1.1 Lattice Geometry

A LBM lattice has to be symmetrical to satisfy the isotropic requirement of fluid properties [19], which means that it has to be an equally-spaced grid.

Since the selection of the geometry depends on the application and dimension, a nomenclature has been developed for easy identification. The format is "DnQm", where n is the number of dimensions (usually 2 or 3), and m is the number of distinct lattice velocities. A common two dimensional geometry used is D2Q9, shown in Figure 1.

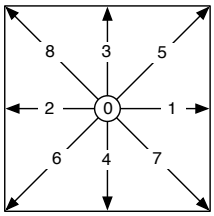


Figure 1: The D2Q9 geometry. The zero-velocity vector is visualized by a small circle in the center.

In three dimensions, three options are widely used [19], as can be seen in Figure 2. They are:

1. D3Q15: Zero velocity, faces, corners
2. D3Q19: Zero velocity, faces, edges
3. D3Q27: Zero velocity, faces, edges, corners

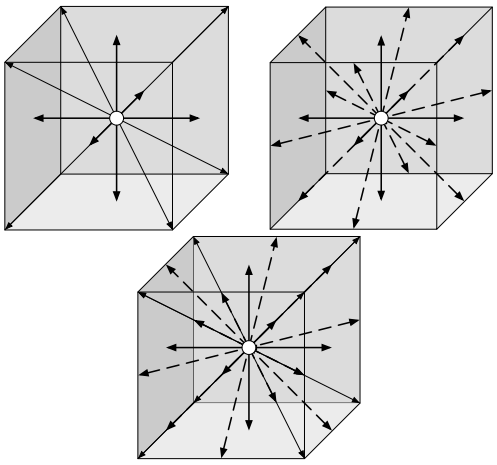


Figure 2: From left to right: D3Q15, D3Q19, D3Q27.

The number of velocity vectors has a direct impact on the performance of the simulation. D3Q15 is prone to numerical instability and other visual artifacts, while D3Q27 requires 27 copy operations per streaming phase, which is expensive. D3Q19 is a good tradeoff between those two extremes, and thus is used in most papers [19, 8, 17, 18].

Table 1 lists the constant weight ω_i for the D3Q19 lattice geometry used in this paper.

i	$\rightarrow\omega_i$
$i = 0$	$\rightarrow\omega_i = 1/3$
$1 \leq i \leq 6$	$\rightarrow\omega_i = 1/18$
$7 \leq i$	$\rightarrow\omega_i = 1/36$

Table 1: The weight ω_i for the D3Q19 lattice geometry.

2.1.2 Gravity

In its original form, the LBM does not account for external forces acting on the fluid like gravity. Buick and Greated [2] outline several methods of varying complexity and compare them using test cases.

The most accurate extends the BGK collision operator by another factor:

$$\Omega_i = -\frac{1}{\tau}(f_i(x,t) - f_i^{eq}(x,t)) + \frac{2\tau - 1}{2\tau}\frac{3}{\omega_i}F \cdot e_i \qquad (7)$$

where F is the force to be applied. This force can not only be gravity, but also others like solid macroparticles acting on the fluid.

2.1.3 Initial Conditions

Since the fluid simulation tends towards the equilibrium distribution, the rest position can be determined by using f_i^{eq} with an arbitrary ρ and u, which can serve as the initial conditions.

Note however, that in a system using gravity, this configuration does not result in a rest configuration, since this force causes the rest configuration to have an uneven density distribution. One possibility for working around this problem is to run the simulation until it comes to rest before introducing any other forces and presenting the interactive display to the user.

3. GENERAL PURPOSE-PROGRAMMING ON GRAPHICS HARDWARE USING CUDA

Programming a parallel streaming processor requires a vastly different approach to solving problems than in common single-threaded multi-purpose processing. For example, while the heapsort and quicksort algorithms are considered to be very efficient, they do not easily support parallel processing. When it comes to implementing sorting on the GPU, the NVidia^{TM} developers recommend the bitonic sort introduced by Batcher [1] or radix sort.

The classical approach to General Purpose-Programming on Graphics Hardware (GPGPU) is using the programmable shader graphics pipeline to implement arbitrary parallel algorithms [9]. However, this approach requires intimate knowledge of one of the graphics application programming interfaces, which creates a barrier-to-entry for scientific developers. NVidia^{TM} tries to remove this barrier by implementing a new approach, a general purpose C++-derived compiled language, which is run directly on the GPU. This also allows greater control of the processors at the expense of simplicity.

3.1 Architecture

Since CUDA approaches the GPGPU topic at a lower level than shading languages, a deeper understanding of the underlying architecture of the NVidia graphics cards is required.

The main focus of GPUs is on data processing. In order to optimize for this type of operation, its architecture devotes more transistors to arithmetic operations than regular

CPUs, sacrificing flow control sophistication (branch prediction for example). Memory latency is hidden by interleaved arithmetical operations instead of data caches. They also employ a data-parallel programming model, meaning that the same operation is applied to multiple input data sets.

The GPU maintains its own memory separate from the host system, but copies between them via a direct memory access controller are possible. Its hardware design is following a layer-approach for memory access and thread scheduling, the reader is referred to the CUDA documentation for an overview.

3.2 Adapting Fluid Simulations to the GPU Using CUDA

Fluid simulation using the Lattice Boltzmann method is well suited for being adapted to CUDA. Since there are no global operations, every cell can be mapped to exactly one thread, and all threads can be executed independently of each other, increasing the flexibility for the thread scheduler. The shared memory feature of CUDA is not required, thus the block and thread size can be chosen in any desired way for attaining optimal performance. This is the main contribution of this paper and it demonstrates how an efficient implementation can be achieved, based on the knowledge about the CUDA architecture.

The input data required for LBM can be stored in arrays residing in global memory. Li et al. [9] propose a certain texture memory layout for shader-based LBM D3Q19 calculations that exploits the possibility to retain data locality even when using bounce back-boundaries. This layout can be used for CUDA for the same reasons and is explained in Table 2. This allows fetching all required data by using a single 128-bit fetch instruction per array. Additionally, the "structure of array" concept is applied instead of the usual "array of structures" preferred on CPUs, as recommended by NVidia^{TM}.

Array	X	Y	Z	W
u	u_x	u_y	u_z	ρ
f_0	$f(1,0,0)$	$f(-1,\ 0,\ 0)$	$f(0,\ 1,\ 0)$	$f(\ 0,-1,0)$
f_1	$f(1,1,0)$	$f(-1,-1,\ 0)$	$f(1,-1,\ 0)$	$f(-1,\ 1,0)$
f_2	$f(1,0,1)$	$f(-1,\ 0,-1)$	$f(1,\ 0,-1)$	$f(-1,\ 0,1)$
f_3	$f(0,1,1)$	$f(\ 0,-1,-1)$	$f(0,\ 1,-1)$	$f(\ 0,-1,1)$
f_4	$f(0,0,1)$	$f(\ 0,\ 0,-1)$	$f(0,\ 0,\ 0)$	unused

Table 2: Distributing the D3Q19 variables in a way to collect values that are required at the same time in the same vector.

Since the streaming operation requires reading in adjacent cell values and global synchronization is not possible, the same arrays can not be used for both reading and writing. Thus, the commonly-used flip flop technique is applied, where all arrays are created twice, and on every frame, the input and output arrays are exchanged. This doubles the memory required for the LBM values, but for real-time simulation, the bandwidth is a greater limiting factor to the fluid grid size than the memory available on the current GPUs anyways.

Li et al. [9] use separate shaders for the stream and collide phases. This would be counter-productive for CUDA-based implementations, since the kernel instruction count and the number of read/write operations are not as limited, and reading/writing from the global memory space is expensive compared to using local registers.

Both scatter and gather approaches are possible on CUDA and the LBM. A scatter operation was chosen, allowing a single kernel to calculate the whole simulation using the following steps:

- Read all values f_i from global memory (by utilizing the texture units via tex1Dfetch).
- Calculate u and ρ.
- Calculate the collisions.
- Write the result to the next cell in direction e_i, implementing the streaming operation.

Since visualization requires the macroscopic fluid information u and ρ, this information has to be written to a separate array in the process. A single slice of the domain might suffice for this, though, depending on the type of visualization.

3.3 Visualizing the Flow

Two ways of visualizing the fluid simulation were implemented: Advecting particle objects and mapping the velocities to a texture, which then is used for drawing an OpenGL quad. Both of these require the OpenGL interoperability feature of the CUDA API, which allows accessing a vertex buffer object or pixel buffer object from a kernel.

For the velocity texture, the resulting pixel buffer object has to be copied to a texture, since accessing an OpenGL texture from CUDA is not possible until the version 3.0, which was not yet available during development.

For the particles, a vertex buffer object can be used directly as the input for glDrawArrays using GL_POINTS as the drawing mode. Since OpenGL allows mapping textures on points, a particle system-based gas visualization can be rendered using the fixed-function pipeline without having to copy the particle positions. The particles' position was calculated using Euler integration.

3.3.1 Geometry Shaders

Using geometry shaders available on CUDA-enabled graphics cards as an OpenGL extension, arbitrary glyphs (like spheres) can be used for rendering the points generated by the particle visualization, too. Since point particles cannot visualize the velocity and velocity textures can only visualize a single slice of the domain, a particle glyph that can represent direction can be used to visualize the flow velocity in three dimensions. Using a geometry shader, displaying a field using pyramids used as arrows like in Figure 3 can be achieved.

A geometry shader uses a primitive as its input (a point in this case) and emits any number of primitives (usually triangles), which are then sent to the rendering pipeline.

Using CUDA, the movement direction of a particle can be stored in the texture coordinates of a point in the VBO. This

information can be used in the geometry shader to rotate the glyph to point to the direction the fluid is moving.

4. BOUNDARY CONDITIONS IN FLUID SIMULATIONS

Since the fluid domain has to be finite, the borders have to represent one of these types of boundaries:

1. Closed boundaries: The fluid is enclosed by walls which can not be passed.
2. Free-flow boundaries: The fluid is not enclosed in any way, but molecules exiting the domain are discarded.
3. Periodic boundary: Molecules exiting the domain on one side enter the domain on the opposing side. For two dimensions, this can be thought of like wrapping the fluid around a three dimensional torus. This concept does not exist in nature, but it can be helpful for programatically generating tiling textures [15].

The LBM handles physical interactions at a local level, so it can be enhanced to support domain boundaries and different fluid (like water and air, or water and oil) and solid interactions with complex boundaries with minimal change. This has been documented by Li [8], Wei et al. [19], Thürey et al. [18].

When considering fluids interacting with solids, there are three different types possible:

1. Fluid-solid: The fluid affects the solid, but the solids are treated like having no mass. One application for this are tin cans floating in water.
2. Solid-fluid: The solid affects the fluid. For example, this can be used for simulating water in a non-changing environment. Wei et al. [19] demonstrate this interaction type in combination with LBM.
3. Two-way interaction: Both are affected by the other. A LBM-based implementation is described by Thürey et al. [18]. This is the most reality-like simulation, but can be quite challenging due to the combination of two different kinds of physics (fluid and rigid). GPU-based implementations face an additional challenge here, since the rigid body simulation has also to be implemented on the GPU for optimal performance (NVidia^{TM} has implemented this into their physics simulation library PhysX, but no direct integration is possible for applications due to the API not exposing the GPU data).

4.1 Integrating a Physics Engine into a Fluid Simulation

The GPU can be used for rigid body physics calculations. However, no physics simulation library using CUDA for acceleration exploses their internal data structures used for this to the outside, and so, collaboration between a CPU-based physics engine API and a GPU-based fluid engine is required.

Any physics engine that allows to get the rigid body's current velocity in a point of its surface and then applying force on that point can be used for the integration.

4.2 Solid-Fluid Coupling

Only closed boundaries are applicable for objects immersed in the fluid. They can be taken into account by implementing a bounce-back rule (inverting the streaming direction) instead of the regular streaming step. For the domain edges, a check based on the coordinates of the current cell has to be used to check for these special cases. For other cases, a flag in every cell defines whether there's a solid at that point.

This technique is limited in two ways: First, moving boundaries do not accelerate the fluid, and second, the accuracy is limited to the grid spacing.

Mei et al. [11] developed a refinement that fixes both problems. However, this technique does not lend itself well to CUDA-based implementations, because it requires branching, and so edge cells would cause divergent threads. This would have a noticeable impact on performance.

Noble and Torczynski [13] use a different approach, which is still based on bounce back. Every cell of the fluid lattice contains a certain fraction ε of solid material to fluid material, which is zero for fully fluid and one for fully solid cells, but can be anything between these values. This value can be determined by using any voxelization algorithm that lends itself to a GPU-based implementation, like inside-outside voxelization [4].

A function B is defined as

$$B(\varepsilon, \tau) = \frac{\varepsilon(\tau - \frac{1}{2})}{(1-\varepsilon) + (\tau - \frac{1}{2})} \tag{8}$$

The Equation 4 is modified to include a second collision operator for collisions with the solid object:

$$f_i^{new}(x,t) - f_i(x,t) = (1-B)\Omega_i + B\Omega_i^s \tag{9}$$

Two different options for Ω_i^s are offered. However, Strack and Cook [16] demonstrate that Holdych [6] offers a more stable equation:

$$\Omega_i^s = f_{-i}(x,t) - f_{-i}^{eq}(\rho, u_s) + f_i^{eq}(\rho, u_s) - f_i(x,t) \tag{10}$$

where u_s is the velocity of the solid object at the cell's point.

In addition to being more stable, since the equilibrium equation contains the same parameters except for the inverted e_i, considering $e_{-i} = -e_i$, this equation can be simplified to

$$\Omega_i^s = f_{-i}(x,t) - f_i(x,t) + 6\omega_i\rho(e_i \cdot u_s) \tag{11}$$

This simplification significantly reduces the amount of in-

structions required. Note that the method presented by Noble and Torczynski [13] has three significant advantages over the one developed by Mei et al. [11] for CUDA-based implementations:

1. Since ε is just a multiplication parameter, no branches are required, which avoids divergent threads for any type of obstacle.
2. No ray casting is required for determining the obstacle border. Using the method presented in Crane et al. [4], fractional ε can be derived by increasing the voxelization grid's resolution with respect to the fluid grid resolution, and then counting the number of solid voxelization cells in a fluid cell.
3. Due to the limited resolution of the fluid grid, porous media like sand cannot be represented with solid obstacle particles. By multiplying the results of the voxelization process by some factor between 0 and 1, a solid obstacle can be made arbitrarily porous.

When the simulated rigid bodies are not porous, the fluid simulation suffers when objects move at a faster pace through the domain, since the density of the solid in a cell fluctuates, also causing a fluctuating fluid density in the cell. This can be partly remedied by making all objects slightly porous. Experimentation with a concrete simulation is required for getting a good balance.

The result of this technique can be seen in Figure 4.

4.3 Fluid-Solid Coupling

In rigid physics simulations, forces applied to rigid objects are stored as only two values with respect to the center of mass: impulse and torque. Since an external physics engine is used, only these values have to be determined by the fluid simulation to be able to let the fluid domain apply any force to the rigid object.

The values calculated by the boundary conditions described by Noble and Torczynski [13] can be used to determine these two values, too, as explained by Strack and Cook [16].

$$F = \frac{\Delta x \Delta y \Delta z}{\Delta t} \sum_n B_n \sum_i \Omega_i^s e_i \tag{12}$$

for the impulse and

$$T = \frac{\Delta x \Delta y \Delta z}{\Delta t} \sum_n (x_n - x_s) \times \left(B_n \sum_i \Omega_i^s e_i \right) \tag{13}$$

for the torque, where n is the iterator over the cells, x_s is the center of the solid's mass and x_n is the position of the nth cell. Note that $B_n \sum_i \Omega_i^s e_i$ is the same in both equations and has only to be determined once. In addition, Ω_i^s is already required for the collision step for the method of Noble and Torczynski [13] and can be re-used.

When implementing this operation in CUDA, the sum has to be implemented as a reduce operation. Since the GPU

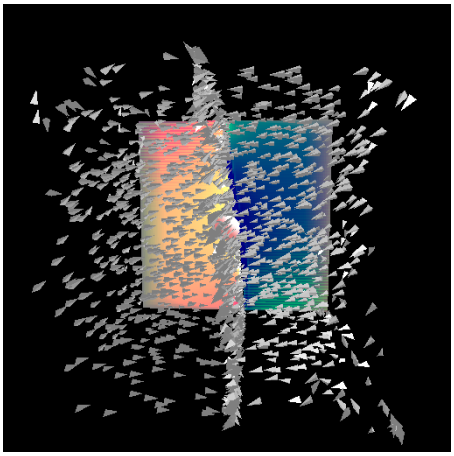


Figure 3: Visualizing the fluid domain's velocity using pyramid glyphs used as arrows.

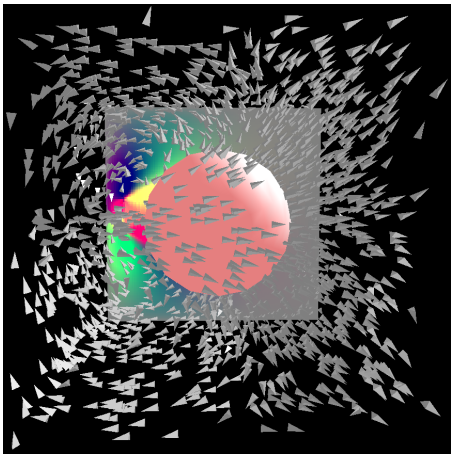


Figure 4: A sphere submersed in the fluid.

is a parallel processing unit, summing is non-trivial. However, NVidia^{TM} provides example code called "reduce" to explain on how to implement this without sacrificing the performance advantage of the GPU.

Since the fluid-solid coupling uses the same parameters as the solid-fluid coupling, it is preferable to combine them into the same CUDA kernel to provide optimal performance.

4.4 Two-Way Coupling

In theory, a combination of the techniques presented in the previous two sections would result in full two-way coupling. However, there are certain issues that have to be kept in mind.

Solids moving faster than the speed of sound in the fluid domain would cause a breakdown of the simulation. This can be avoided by limiting the maximum speed of moving boundaries. However, in two-way coupling, this causes the fluid-solid-coupling to generate incorrect results, too.

Since a physics engine and a fluid simulation have to communicate with each other, the physical units have to be kept in sync. For example, a 1m metal sphere with a mass of 1kg might look realistic in a purely rigid simulation, but would generate unexpected behavior when immersed in a fluid.

Further, when a rigid object moves faster than it would be possible in the given fluid due to aerodynamic resistance, the fluid's recoil into the opposite direction has a greater force than the object's force into its current direction. This is due to skipping the step of applying forces to the fluid and directly using the current speed of the boundary.

5. EVALUATION

In previous experiments [12], the voxelization technique for solids was determined to be the major deciding factor of the runtime performance. Thus, the integration between solid and fluid lends itself well to using inherently voxelized data structures like octtrees. Further, primitive objects like cubes and spheres can be voxelized using an algorithm optimized for these object types.

However, since voxelization is not a focus of this paper, only the performance of the fluid simulation itself was evaluated, as can be seen in Figure 5 (note that performance is independent of the simulation results). The testing equipment used was an Intel Core2 Quad running at 2.83GHz on Ubuntu Linux with a NVidia GeForce GTX 295 (also running the display) with 896MB RAM. The application was compiled in 64bit mode for CUDA 2.3.

Due to RAM limitations on the GPU, a maximum of $128 \times 128 \times 128$ cells could be simulated. The results are shown in Table 3. The memory requirements for a single cell are 5 four-component vectors of floats (16 bytes each), plus another four-component vector of floats if a visualization is desired (containing the velocity and pressure of each cell). Note that the visualization was not part of this performance measurement.

Using a $128 \times 128 \times 128$ domain, an average performance of 83.4 simulation steps per second could be achieved. One simulation step involves one collision and one streaming step. This amounts to over 174.9 million cells per second.

On a $64 \times 64 \times 64$ domain, an average performance of 615.7 simulation steps per second was observed. This amounts to over 161.4 million cells per second. This number being lower stems from the fact that a higher overhead has to paid.

Also of note is that the typical behavior of CPU-based timing, that performance increases over time when an algorithm is executed over and over again, can not be experienced on a GPU, since there are no caches to "warm up".

6. CONCLUSION

Due to the advancements in graphics processing hardware, grid-based fluid simulations have moved into the radar of real-time applications.

Graphics processors have become stream-based processing units capable of processing tasks with large amounts of data and a high arithmetic density in a time frame where real-time simulations are possible, even when rendering is also taken into account.

It has been demonstrated that the Lattice Boltzmann method is ideally suited for GPU-based implementations due to its simplicity and accuracy. Further, integrating complex objects using two-way coupling is possible with minimal modifications to the original equations and any voxelization technique that utilizes the GPU.

The main limitation of simulations is the RAM requirement, not the calculation performance. This will be rectified over time with newer GPU generations (as well as increasing the performance). Scaling the simulation to any number of parallel threads is not a problem, it even reduces the calculation

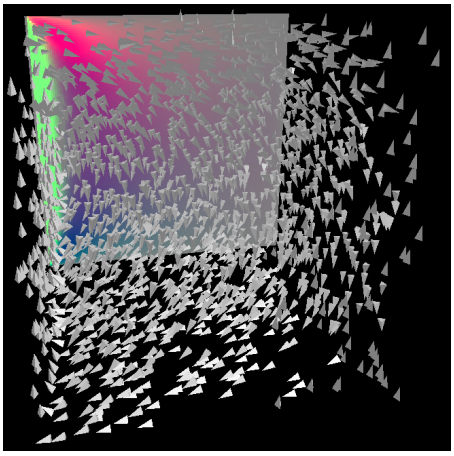


Figure 5: A lid-driven cavity flow.

Resolution	Steps Per Second	Cells per Second
$64 \times 64 \times 64$	615.7	161.4 million
$128 \times 128 \times 128$	83.4	174.9 million

Table 3: The evaluation results of the implementation demonstrated in this paper.

time per cell.

The next step to optimize the simulation process would be to offload the rigid body physics to the GPU, in order to decrease the two-way communication required between the two processing units.

References

[1] K. E. Batcher. Sorting networks and their applications. In *Spring Joint Computer Conference*, Akron, Ohio, USA, 1968.

[2] J. M. Buick and C. A. Greated. Gravity in a lattice Boltzmann model. *Physical Review E*, 61(5):5307–5320, 2000.

[3] Jim X. Chen and Niels da Vitoria Lobo. Toward interactive-rate simulation of fluids with moving obstacles using Navier-Stokes equations. *Graph. Models Image Process.*, 57(2):107–116, 1995. ISSN 1077-3169. doi: http://dx.doi.org/10.1006/gmip.1995.1012.

[4] Keenan Crane, Ignacio Llamas, and Sarah Tariq. *GPU GEMS 3 Chapter 30, Real-Time Simulation and Rendering of 3D Fluids*. Addison Wesley, 2007.

[5] Mark Finch. *GPU GEMS Chapter 1, Effective Water simulation from Physical Models*. Addison Wesley, 2004.

[6] David J. Holdych. *Lattice Boltzmann methods for diffuse and mobile interfaces*. PhD thesis, University of Illinois at Urbana, Champaign, USA, 2003. Ph.D. Thesis.

[7] Jens Krüger and Rüdiger Westermann. GPU simulation and rendering of volumetric effects for computer games and virtual environments. *Computer Graphics Forum*, 24(3), 2005.

[8] Wei Li. *Accelerating Simulation and Visualization on Graphics Hardware*. PhD thesis, Computer Science Department, Stony Brook University, 2004.

[9] Wei Li, Zhe Fan, Xiaoming Wei, and Arie Kaufman. *GPU GEMS 2 Chapter 47, Flow Simulation with Complex Boundaries*. Addison Wesley, 2004.

[10] Youquan Liu, Xuehui Liu, and Enhua Wu. Real-time 3d fluid simulation on GPU with complex obstacles. In *PG '04: Proceedings of the Computer Graphics and Applications, 12th Pacific Conference on (PG'04)*, pages 247–256. IEEE Computer Society, 2004.

[11] Renwei Mei, Wei Shyy, Dazhi Yu, and Li-Shi Luo. Lattice Boltzmann method for 3-d flows with curved boundary. *Journal of Computational Physics*, 161:680–699, 2000.

[12] Andreas Monitzer. Fluid rendering on the gpu with complex obstacles using the lattice boltzmann method. Diploma Thesis, August 2008.

[13] D. R. Noble and J. R. Torczynski. A lattice-Boltzmann method for partially saturated computational cells. In *7th Int. Conf. on the Discrete Simulation of Fluids*, 1998.

[14] Carlos E. Scheidegger, Joao L. D. Comba, and Rudnei D. da Cunha. Navier-stokes on programmable graphics hardware using smac. In IEEE Press, editor, *Proceedings of XVII SIBGRAPI - II SIACG 2004*, ISBN 0-7695-2227-0, pages 300–307, 2004.

[15] Jos Stam. Stable fluids. In *Siggraph 1999, Computer Graphics Proceedings*, pages 121–128, Los Angeles, 1999. Addison Wesley Longman.

[16] O. Erik Strack and Benjamin K. Cook. Three-dimensional immersed boundary conditions for moving solids in the lattice-Boltzmann method. *International Journal for Numerical Methods in Fluids*, 55:103–125, 2007.

[17] Nils Thürey. A single-phase free-surface lattice-Boltzmann method. Master's thesis, University of Erlangen-Nuremberg, 2003.

[18] Nils Thürey, Klaus Iglberger, and Ulrich Rüde. Free surface flows with moving and deforming objects for LBM. In *Proceedings of Vision, Modeling and Visualization 2006*, pages 193–200. IOS Press, 2006.

[19] Xiaoming Wei, Wei Li, Klaus Mueller, and Arie E. Kaufman. The lattice-Boltzmann method for simulating gaseous phenomena. *IEEE Transactions on Visualization and Computer Graphics*, 10(2):164–176, 2004. ISSN 1077-2626. doi: http://dx.doi.org/10.1109/TVCG.2004.1260768.

[20] Jakub Wejchert and David Haumann. Animation aerodynamics. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 19–22, New York, NY, USA, 1991. ACM Press. ISBN 0-89791-436-8. doi: http://doi.acm.org/10.1145/122718.122719.

A Framework for GPU Accelerated Deformable Object Modeling

Aria Shahingohar
The University of Western Ontario
1151 Richmond Street
London, Ontario, Canada
ashahing@uwo.ca

Roy Eagleson
The University of Western Ontario
1151 Richmond Street
London, Ontario, Canada
eagleson@uwo.ca

ABSTRACT

We have developed a framework that uses multicore CPUs and GPUs found on personal computers to accelerate the computations needed for a class of deformable object modeling algorithms. In recent years there has been a growing interest in using deformable objects in computer applications such as animation, video games, garment CAD, and surgical simulation. Deformable object modeling is quite computationally expensive. However, since most of the related calculations can be parallelized, we have developed a framework that utilizes Nvidia's CUDA technology to accelerate a set of deformable object modeling algorithms by transferring their core computations to the GPU. Our results show that frame rates can be improved more than 20 times using GPU compared with using a multi-core CPU. In addition, we have developed a method called Local Shape Matching which is an extension to Shape Matching method. Using this new method we have achieved fast and robust simulations which are demonstrated in the presentation.

1. INTRODUCTION

GPUs are becoming a natural platform for computationally demanding tasks in a wide variety of application non-graphical, Scientific Computation domains. This is due to the increased performance of graphics hardware, and to recent improvements in their programmability.

Even though CPUs have evolved so much and their price has declined significantly, commodity GPUs are delivering better performance with respect to cost for parallelizable computations.

For example, the NVIDIA GeForce GTX 280 ($450 as of June 2010) can achieve a sustained 141.7 GB/sec of memory bandwidth with a computing performance around 933 GFLOPS in single precision calculations. As of 2010, the fastest PC processor has a theoretical peak performance of 79.9 GFLOPS (Intel Core i7 980X) in double precision calculations with 25.6 GB/sec memory bandwidth. Not only is current graphics hardware fast, but it is growing faster than for CPUs as well. Semiconductor technology, driven by advances in fabrication technology, is increasing at the same rate for both CPU and GPU. The reason that the performance of graphics hardware increasing faster than that of CPUs is due to the scaled enhancement given by the higher parallelism. CPUs are optimized for high performance for sequential computing; therefore, many of their transistors are dedicated to supporting non-computational tasks like branch prediction and caching. On the other hand, the highly parallel nature of graphics computations enables GPUs to use additional transistors for computation, achieving higher arithmetic intensity with the same transistor count [16].

Because of the high parallelism that exists in physical simulation of deformable bodies, we can use the GPU to reduce the calculation time. GPUs have been used for general purpose computing (GPGPU) for several years. However, since they previously were designed specially for rendering and rasterizing, programmers had to use complicated tricks in order to take advantage of their stream processing architecture. However, efforts has been made to perform the computations of deformable object modeling on GPUs using shaders. Georgii and Westermann implemented mass-spring system on GPU [7]. Nonlinear FEM has even been implemented on GPU [21]. However, since GPU hardware and shaders were not designed for this kind of operation, coding was complicated. NVIDIA's CUDA[1] GPGPU technology is a fundamentally different computing architecture for solving complex computational problems. CUDA (Compute Unified Device Architecture) supports development using high-level language, further enhancing its popularity.

In the next section we will provide background on deformable object modeling and review the literature on their GPU implementation. In section 3 we explain four methods that are implemented in our framework and introduce Local Shape Matching Method. In section 4 we will explain our GPU based framework for deformable object modeling. In section 5 we have shown our results and we have concluded in the last section.

2. BACKGROUND

We begin by reviewing the literature and motivating our selection of the set of deformable object modeling algorithms that will be implemented comparatively in this paper.

Mass-Spring models have been popular in Computer Animation for over 20 years. In SIGGRAPH 87 *John Lasseter* shocked the computer graphics industry by presenting *Luxo Jr.* his first 3D animation which was produced at *Pixar* [11]. His ideas allowed animators to extend traditional 2D

storyboarding techniques, keyframe animation, "inbetweening", and scan/print to the 3D realm. At the same venue Terzopouos et. al [23] presented the first paper on physically based animation. They employed elasticity theory to construct differential equations to model the behavior of non-rigid curves, surfaces, and solids as a function of time. Since then, many researchers have taken advantage of various scientific concepts, in animation of deformable objects, hair, cloth and fluids [8, 15].

2.1 Physically Based Methods

Most physically based methods for modeling deformable objects are based on continuum elasticity. Under the assumptions of continuum mechanics, the behaviour of a deformable object can be expressed as follows. Suppose the rest shape of a continuous deformable object is a subset of Ω of $\mathbf{R}^3$. Ω consists of all the particles with positions $\mathbf{x}_0 = [x, y, z]^\tau$, where $\mathbf{x}_0 \in \Omega$. $\mathbf{x}_0$ denotes the *material coordinate* of a point in rest position. When a force is applied, the shape deforms. We suppose that a point at location $\mathbf{x}_0$ moves to a new location with a displacement denoted by $\mathbf{u} = [u, v, w]^\tau$ such that the new location of particle $\mathbf{x}_0$ is $\mathbf{x} = \mathbf{u}+\mathbf{x}_0$. Usually a measure for deformation is defined (Strain) and at each iteration of simulation internal forces are calculated such that linear and angular momentum is preserved or strain energy stored in the object is minimized. Once these forces are calculated, displacements are calculated by integration.

2.2 Non-Physically Based Methods

Although physically based deformable object modeling methods are generally based on simple Continuum elasticity, precise solutions to these methods cannot be implemented in real time. In addition, approximations can make the simulations either very inaccurate, or unstable numerically. Therefore, non-physically based methods that are based on simplifying smoothness constraints are still an attractive choice. Mass-Spring method and shape matching technique[13] are two examples of non-physically based methods. For a complete review of deformable object modeling you can refer to[15].

2.3 NVIDIA CUDA and AMD FireStream

In 2006, ATI launched FireStream as the industry's first commercially available hardware stream processing solution. At first it was designed as a virtual machine abstraction for GPUs that provided policy-free, low-level access to the hardware designed for high-performance, data-parallel applications [17]. In the same year AMD acquired ATI and re-branded the API to the AMD Stream Processor, but it was changed to AMD FireStream in 2007. NVIDIA lunched their own GPU parallel computing architecture CUDA in 2007. CUDA gives developers access to the native instruction set and memory of the parallel computational elements in CUDA GPUs. Using CUDA, the latest NVIDIA GPUs effectively become open architectures like CPUs. CUDA and FireStream both provide similar functionality, however, CUDA seems to have been absorbed by the scientific community. It is shown that typical applications such as Traffic Simulation, Thermal Simulation, and K-Means can be accelerated using the GPU, demonstrating as high as 40 times speedup when compared with a CPU implementation [4]. Hence researchers have used CUDA stream processing for a variety of applications.

CUDA has been used for accelerating some existing deformable object modeling techniques as well. Rasmusson et. al. [20] investigated multiple implementations of volumetric Mass-Spring-Damper systems in CUDA. They compared the performance of CUDA to previous implementations utilizing the GPU through the OpenGL graphics API and showed that performance and optimization strategies differ widely between the OpenGL and CUDA implementations [20] The non-linear FEM method mentioned in [21] has been accelerated with CUDA in [5] by Comas et. al. within their Sofa Framework.

3. IMPLEMENTED METHODS

In this section, we provide an overview on the four methods that are implemented in our framework.

3.1 Weighted Mass-Spring Method

Mass-spring Method (MSM) is the simplest, most intuitive and most common method used in deformable object modeling. Since the early works by Terzopoulos et.al. [22, 23] mass-sprig method has been used for modeling of various deformable objects such as cloth simulation [2, 19], face animation [10] and soft tissue [8, 24].

In this system, the deformable object is considered in a discrete space. This model consist of point masses connected together with mass-less springs and dampers.

In the original mass-spring system no weight function is used but we have added weight so nearest neighbours will have a greater effect. By adding weights and considering constant stiffness coefficient k_s the elastic force is obtained by:

$$\mathbf{f}_i^s = \sum_{j \in N(\mathbf{x}_i)} k_s w_{ij}(\mathbf{x}_{ij}) \left(1 - \frac{l_{ij}^0}{\|\mathbf{x}_{ij}\|}\right) \quad (1)$$

Where $\mathbf{x}_{ij} = \mathbf{x}_j - \mathbf{x}_i$ in which $\mathbf{x}_i$ is the position of node i. l_{ij}^0 is the initial length of spring between node i and node j, $N(\mathbf{x}_i)$ is the set of neighbours of node i and w_{ij} is the weight of jth neighbour of node i. By considering weights and constant damping coefficient k_d, the damping is approximated by:

$$\mathbf{f}_i^d = \sum_{j \in N(\mathbf{x}_i)} k_d w_{ij} \frac{\mathbf{v}_{ij}^\tau \mathbf{x}_{ij}}{\|\mathbf{x}_{ij}\|} \mathbf{x}_{ij} \quad (2)$$

Where $\mathbf{v}_i$ is the velocity of node i and $\mathbf{v}_{ij} = \mathbf{v}_j - \mathbf{v}_i$.

3.2 Local Shape Matching (LSHM)

A special meshless non-physically based model for deformable objects was developed by Müller er al. [13] that is able to provide a robust simulation. We have developed a method for modeling deformable objects based on the "Shape Matching" method. We have extended the concept of clusters in Shape Matching, such that a cluster is defined for each point. An overview of this approach is shown in Figure 1. At the beginning of the simulation for each node i the center of mass $\mathbf{C}_i^0$ is calculated for that node and its neighbours; also the vector ν_i from the center of mass to each node is stored. At each iteration of the simulation, rotation should be extracted from the deformation. The rotation is approximated using the least square optimization explained in [13]. As shown in Figure 1(c), extracting the rotation is equivalent to rotating

the coordinate system in reverse $\mathbf{x}^R = \mathbf{R}^{-1}\mathbf{x}$. For each node i in the new rotated coordinate system, the goal position is located at $\mathbf{g}_i^R = \nu_i + \mathbf{C}_i^R$ where $\mathbf{C}_i^R$ is the new center of mass of node i and its neighbours in the rotated coordinate system. Once the goal position are found in the new coordinate system, the goal positions are transformed to the original coordinate system $\mathbf{g}_i = \mathbf{R}\mathbf{g}_i^R$. Instead of using the integration schema proposed by Müller et.al., we introduce a restoring force from the current position to the calculated goal position:

$$\mathbf{f}_i^s = k_s(\mathbf{g}_i - \mathbf{x}_i) \tag{3}$$

While rotation can be approximated for each node, we approximate the rotation for the whole shape and use it for all nodes for simplicity. On the other hand, it turns out that we don't have to transform all the nodes to the rotated coordinate system. The same results can be achieved if we only rotate ν_i. An overview of our algorithm is given in Algorithm 1.

Algorithm 1 Our Local Shape Matching algorithm

```
{Initialization}
for all nodes i do
  C_i^0 = 1/||N(x_i)|| Σ_{j∈N(x_i)} x_j^0
  ν_i = x_i^0 − C_i^0
end for
{At each iteration}
Approximate the rotation → R.
for all nodes i do
  C_i = 1/||N(x_i)|| Σ_{j∈N(x_i)} x_j
  g_i = C_i + Rν_i
  f_i = k_s(g_i − x_i)
end for
{Explicit integration}
for all nodes i do
  ẋ_i ← ẋ_i + △t f_i/m_i
  x_i ← x_i + △t ẋ_i
end for
```

3.3 Meshless Finite Element method

If the finite element method is applied to a linear system it produces a linear system of algebraic equations. Assuming linear stain and assuming that our material is isotropic the governing equation of the material ($\rho\ddot{\mathbf{x}} = \nabla \cdot \sigma + \mathbf{f}$) can be simplified as follows:

$$\rho\ddot{\mathbf{x}} = \mu\triangle\mathbf{u} + (\lambda + \mu)\nabla(\nabla.\mathbf{u}) \tag{4}$$

Where $\mathbf{u}_i = \mathbf{x}_i - \mathbf{x}_i^0$ is the displacement of each node, ρ is the density of the material and λ and μ are Lamé's coefficients. Debunne et.al. [6] used a mesh free method to solve this equation. Their method is based on calculating Laplacian($\bigtriangledown = \triangle^2$) and gradient of divergence ($\bigtriangledown(\bigtriangledown.)$) in a discrete fashion. Using their approach these geometric operators are defined based on the neighboring nodes only as follows:

$$\triangle\mathbf{u}_i = \frac{2}{\sum_{j\in\mathcal{N}(\mathbf{x}_i)} l_{ij}} \sum_{j\in\mathcal{N}(\mathbf{x}_i)} \frac{\mathbf{u}_j - \mathbf{u}_i}{l_{ij}} \tag{5}$$

$$\nabla(\nabla.\mathbf{u}) = \frac{2}{\sum_{j\in\mathcal{N}(\mathbf{x}_i)} l_{ij}} \sum_{j\in\mathcal{N}(\mathbf{x}_i)} \frac{\left[(\mathbf{u}_j - \mathbf{u}_i).\frac{\mathbf{l}_{ij}}{l_{ij}}\right]\frac{\mathbf{l}_{ij}}{l_{ij}}}{l_{ij}} \tag{6}$$

where $\mathbf{l}_{ij} = \mathbf{u}_j - \mathbf{u}_i$ and $l_{ij} = \|\mathbf{u}_j - \mathbf{u}_i\|$ is the distance between sample points i and j.

In the original method mentioned all neighbours have the same weight. We have modified such that closer neighbours will have a greater effect as expected. On the other hand, the original method does not handle significant rotations. In order to fix this, we perform the calculations on the objects coordinate system as we did in the previous section. Therefore, at each iteration we approximate the global rotation of the object. To calculate Laplacian (Equ. 5) and Gradient of divergence (Equ. 6) we need to calculate the displacements ($\mathbf{u}_i = \mathbf{x}_i - \mathbf{x}_i^0$). To compensate for the rotation, before calculating the displacements we rotate the original points with the approximated rotation of the object:

$$\tilde{\mathbf{x}}_i^0 = R\mathbf{x}_i^0 \tag{7}$$

$$\mathbf{u}_i = \mathbf{x}_i - \tilde{\mathbf{x}}_i^0 \tag{8}$$

3.4 Point Based Animation

Point based graphics is an active research area in computer graphics in which the surface is rendered as point sampled surfaces(surfles) rather than polygonal surfaces [18]. There has been growing interest in combining meshfree methods and point based graphics. Müller et. al. proposed Point Based Animation (PBA) a mesh free continuum mechanics method for animation of elastic, plastic and melting objects [14]. In their approach the geometrical strain is approximated around each node based on the deformation of neighbours nodes. Then the derivation of elastic energy is approximated in the area around each point(phyxel) to calculate the resulting elastic force. They use moving least square optimization to approximate strain, stress and strain energy. To calculate the elastic forces they use the same approach to calculate the derivation of strain energy with respect to displacement. The following is an overview of their method:

$$\begin{array}{ccccccc} \mathbf{u}_t & \rightarrow & \nabla\mathbf{u}_t & \longrightarrow & \varepsilon_t & \rightarrow & \sigma_t \\ & & & & \searrow & & \swarrow \\ \mathbf{u}_{t+\triangle t} & \longleftarrow & \mathbf{f}_t = \nabla_{\mathbf{u}}\mathbf{U}_t & \leftarrow & \mathbf{U}_t & & \end{array} \tag{9}$$

4. OUR GPU BASED FRAMEWORK

We have developed a framework for animating deformable objects based on particle-based methods. Our framework was specifically designed for efficient implementation on CUDA GPU architectures, and can be mapped easily onto other multicore CPUs. An overview of our algorithm is shown in Fig. 2. We made use of the *OpenMP*[3] library to map the kernel loop onto all available CPU cores. Numerically, we made use of explicit integration and restrict each step to 10 iterations. That means we enforce the boundary conditions only once per 10 iterations which reduces the number of transfers between host and device.

Ideally we want to perform all the tasks on the GPU however; some tasks can not be parallelized, and consequently they would be even slower on the GPU. Our goal is to utilize the GPU parallel computing capabilities fully by transferring the heavy computation of internal interaction of nodes and numerical integration to the GPU. By doing so, the CPU is reserved for performing sequential tasks such as collision detection.

Transferring data (such as positions of the nodes) between *host* (CPU side) and *device* (GPU side) is slow and can easily become a bottleneck for processing. Therefore, we limit these transfers as much as possible.

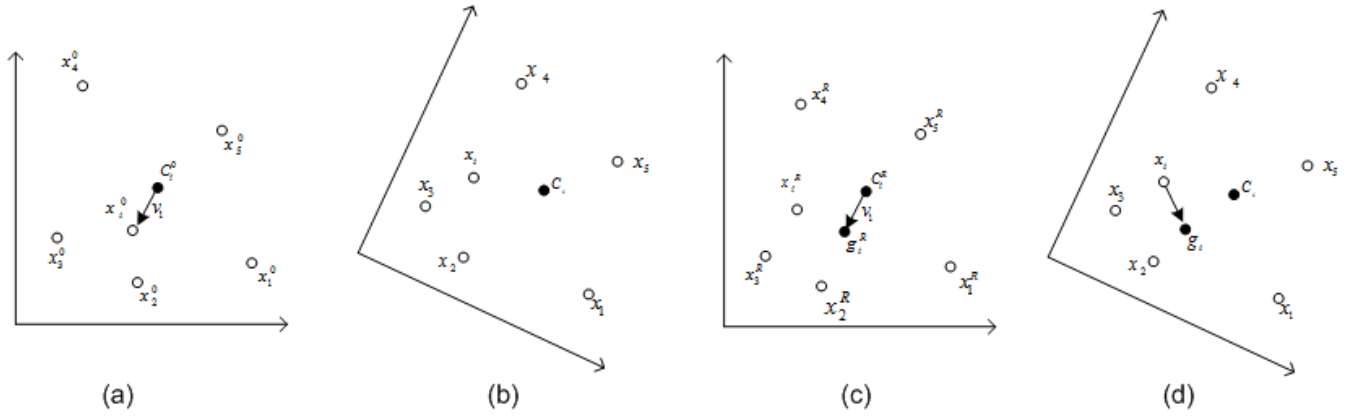


Figure 1: Local Shape Matching. (a) Node $\mathbf{x}_i$ and its neighbours at the beginning of the simulation. $\mathbf{C}_i^0$ is the center of mass node i and its neighbours, $\nu_i = \mathbf{x}_i^0 - \mathbf{C}_i^0$ (b) Node $\mathbf{x}_i$ and its neighbours after deformation. (c) The rotation is extracted by rotating the coordinate system. Then the goal position is calculated at the rotated coordinate system; it is located at the same vector position from the center of the mass $\mathbf{g}_i^R = \nu_i + \mathbf{C}_i^R$. (d) Goal positions are rotated back to the original coordinate system and a force is applied in the direction of $\mathbf{x}_i - \mathbf{g}_i^R$.

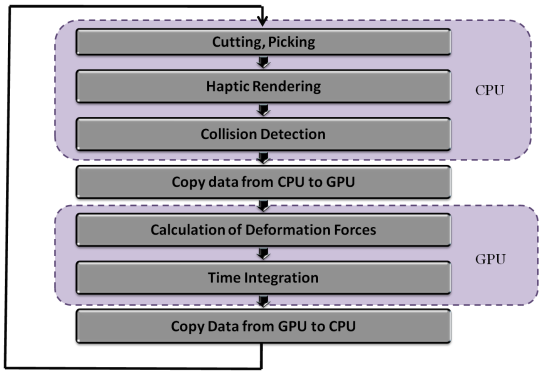


Figure 2: Overview of our simulation framework.

4.1 Data Structure

In our framework, for each node a set of arrays are generated to store variables associated with the simulation. These variable are stored for all nodes and include position, velocity, elastic force, list of neighbours, and weight of each neighbour. In addition to these variables, there are others that are specific to each of the methods that we compare, and there fore are declared separately. In the Mass-Spring method the original length of each spring is stored. One of the features of our Local Shape Matching Method is that, instead, the vector from center of mass of neighbourhood to each node is stored. In the Debunne method only the original positions are saved. In Point Based Animation the matrix $\mathbf{M}^{-1} = \left(\sum_j \mathbf{x}_{ij}\mathbf{x}_{ij}^T w_{ij}\right)^{-1}$ is retained. All variable arrays described previously are created and initialized at the beginning of simulation on the host(CPU) and copied into the device(GPU). However, during simulation, only the positions and velocities are sent back and forth between host and device. Positions and velocities are transferred to host to perform collision detection, and then the corrected values are sent back to GPU for each next iteration. CUDA has provided the OpenGL inter-operability which enables the user to map a CUDA buffer to OpenGL buffer and vise versa. Unfortunately since this operation is not implemented efficiently, it turns out to be much faster for us to send us the position to the GPU twice, once for rendering, and another time for the simulation. The calculated forces are also transferred to host at each iteration if haptic rendering is required.

Table 1 shows the proprieties of each variable that is created on the GPU. Velocity, position and force are of float size 4 instead of 3. The reason is that in our CUDA kernel we use float4 since the platform is not optimized for accessing float3.[1] The Neighbours array keeps the index of $n\_Neighbours$ closest nodes to each node and effect of each neighbour is weighted according a function. We have used two kernel functions in our simulation. The first kernel calculates the forces(or accelerations) and the second kernel is used for explicit integration. For each node, a thread is generated; therefore, each thread gathers the required data from the memory updates the force, velocity and the position of its corresponding node. Some of the arrays don't change during the simulation, and therefore we have used the read-only cached texture memory for them. The rest of the arrays are allocated to the global memory. The Global memory has high latency compared to the shared memory. Unfortunately we can not use the shared memory for those arrays since the shared memory is only accessible within threads of the same thread block. The size of the thread block is limited and it is chosen based on shared memory and register requirements. For a typical mesh the threads are scattered on several thread blocks; Therefore, neighbours of a node might reside on another thread block.

4.2 Weight Function

[1]Global memory resides in device memory and device memory is accessed via 32, 64, or 128 byte memory transactions. These memory transactions must be naturally aligned: Only the 32, 64, or 128 byte segments of device memory that are aligned to their size (i.e. whose first address is a multiple of their size) can be read or written by memory transactions.

Table 1: Data arrays and their sizes

Array Name	Type	Size	GPU Memory	Transfer
Position	float4	$4 \times n\_Nodes$	Global	Each Iteration
Velocity	float4	$4 \times n\_Nodes$	Global	At Initialization
Force	float4	$4 \times n\_Nodes$	Global	Each Iteration
Neighbours	int	$n\_Neighbours \times n\_Nodes$	Texture	At Initialization
Nei_Weights	float	$n\_Neighbours \times n\_Nodes$	Texture	At Initialization
Method Specific				
node distances (MSM)	float	$n\_Neighbours \times n\_Nodes$	Texture	At Initialization
ν_i (LSHM)	float4	$4 \times n\_Nodes$	Texture	At Initialization
Original Positions (DEB,PBA)	float4	$4 \times n\_Nodes$	Texture	At Initialization
$\mathbf{M^{-1}}$ (PBA)	float	$9 \times n\_Nodes$	Texture	At Initialization

Table 2: Weight functions used in mesh free methods [12].

Linear	$w_{ij} = \begin{cases} 1-q & \text{if } q < h \\ 0 & \text{otherwise} \end{cases}$
Gaussian Function	$w_{ij} = \begin{cases} e^{\frac{-q^2}{\sigma^2}} & \text{if } q < 1 \\ 0 & \text{otherwise} \end{cases}$
Cubic Spline Function	$w_{ij} = \begin{cases} 1 - 3/2q^2 + 3/4q^3 & \text{if } 0 \le q \le 1 \\ 1/4(2-q)^3 & \text{if } 1 \le q \le 2 \\ 0 & \text{otherwise} \end{cases}$
Quartic Spline Function	$w_{ij} = \begin{cases} 1 - 6q^2 + 8q^3 - 3q^4 & \text{if } q < 1 \\ 0 & \text{otherwise} \end{cases}$
Point Based Animation [14]	$w_{ij} = \begin{cases} (1-q^2)^3 & \text{if } q < 1 \\ 0 & \text{otherwise} \end{cases}$

where h is a threshold and $q = |\mathbf{x}_{ij}|/h$

Different formulas have been used for the weights of the neighbours. Some of these functions are shown in Table 2. The weight function should be symmetric ($w_{ij} = w_{ji}$) and it should be smooth, positive and monotonically decreasing. All the functions in Table 2 satisfy these conditions. We have chosen Cubic Spline Function as our weight function. On the other hand we have to make sure that neighborhood relationship is mutual to ensure the third law of Newton(action equals reaction) is preserved. Therefore, if for example j is in the list of neighbours of i but i is not in the list of neighbours of j we make the weight of j in the list of weight of i to be zero.

5. RESULTS

In order to compare the accuracy of different methods quantitatively, we have analyzed our results with the *Truth Cube* [9] experiment. Kerdoka et. al developed a physical standard to validate soft tissue deformation models. They took CT images of a cube of silicone rubber with a set of embedded Teflon spheres that underwent uniaxial indentation tests. They used silicone rubber (RTV6166, General Electric Co.) which exhibits linear behavior to at least 30% strain. They scanned experimental setup when the cube was unloaded. Then plate was held level as it was lowered onto the oiled top surface of the cube to a set displacement. Three loading conditions were scanned: 4mm, 10mm, and 14.6mm displacements producing 5.0%, 12.5%, and 18.25% nominal strain respectively.

In order to make sure that we track the right positions, we add 343 nodes to each reference mesh. The positions of these additional mesh nodes were initialized based on the position of measured Truth Cube data with zero displacement. In the Truth Cube experiments the cube is in equilibrium state with presence of gravity force. In other words the stress and strain is not equal to zero from the beginning. For our simulation, we assume that gravity is zero, since if we don't neglect the gravity, our model will deform as soon as the simulation starts, while in Truth Cube experiment there is no deformation if there is no contact. For the used silicon rubber the Young's modulus is equal to 15kPa and an Poisson's ratio is close to .5. In our simulation we have assumed that Poisson's ratio is equal to 0.499. Mass-Spring system and Local Shape Matching are not based on continuum elasticity, therefore they can not be expressed by Young's modules of elasticity. For these methods the spring coefficient is manually adjusted to achieve reasonable results for those comparisons.

We then simulated the truth cube experiment by lowering a plate for to simulate the uniaxial compression test. The plate was moved manually and the position of nodes corresponding to truth cube experiment were saved on a file to be processed later. By comparing the result of our simulation to truth cube we can have a performance measure for linear elastic state. We define a relative error measure for each point as follows:

$$\mathbf{e}_i = 100 \frac{\|\mathbf{x}_i^s - \mathbf{x}_i^{tc}\|}{80} \tag{10}$$

where $\mathbf{x}_i^s$ is the simulation result for node i and $\mathbf{x}_i^{tc}$ is the position of corresponding node in truth cube experiment. We have divided the error by 80mm to normalize the error with respect to the dimension of the cube. The average is then defined as follows:

$$E = \frac{1}{343} \sum_{i=1}^{343} e_i \tag{11}$$

Figure 3 shows some screen shots of simulation of cube deformation. The cube is composed of 2207 nodes with 16 neighbours for each method. In the shown simulation the bottom of the cube is fixed to the ground and the cube is pressed from the top. The simulation is repeated using discussed modeling methods. The screen shots are taken at rest state, 5.0%. 12.5% and 18.25% strain levels. As it is shown in the screen shots, Mass-Spring method had failed in 18.25% strain. Our Local Shape Matching Method is more robust than all other methods; however, it does not conserve the volume of the object. Debunne method and PBA are more accurate since they are physically based, they also conserve the volume better. On the other hand the Debunne method has failed in 18.25% strain.

In Figure 4 we have compared the accuracy of the simulations by comparing the errors found according to Equ.11.

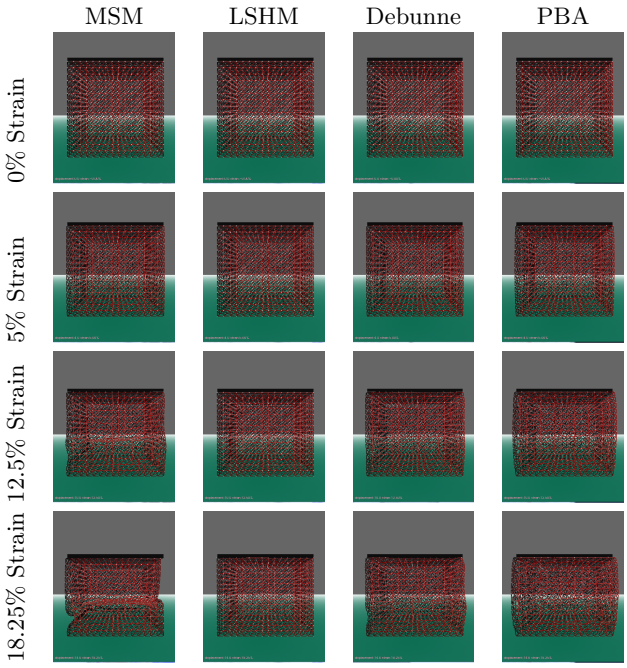


Figure 3: Simulation of uniaxial compression of a cube with four methods in our framework. Number of nodes:2207, number of neighbours: 16.

The results are given for 5 different mesh sizes and are compared for three stages of deformation. We had to increase the number of neighbours from 16 to 32 since simulations were very inaccurate and unstable for larger mesh sizes.
As we expect the error is higher where there is greater deformation(strain). By manually adjusting the stiffness with trial and error we were able to achieve good results, however we had to use different values for different meshes. For example for the mesh with 3232 nodes and 32 neighbours although the selected stiffness coefficient has resulted in low error for 5% and 12.5% strain it has resulted in a total collapse in the cube for 18.25% strain. The Local Shape Matching method has resulted in lower accuracy compared to MSM since it does not conserve the volume. The error for LSHM is in the same range for all different mesh sizes. This is a nice feature since it ensures we get the same behavior when the resolutions is increased or decreased. The Debunne method has resulted in lower errors in general however it shows sensitivity to the mesh size. Although PBA is the most sophisticated method in our framework it has not resulted in less errors. This method might not be suitable for materials with chosen physical properties. It is to be noted that the mentioned comparison only provides a measure for linear elastic simulation. Different results might be achieved in the dynamic simulation.

In order to compare the performance of different methods together and effect of CUDA we run the simulation on a cube meshes with different resolutions. We ran the simulation on a PC with Intel(R) Core 2 Quad 2.4Ghz CPU with 4GB of ram and graphics processing unit of NVIDIA GTX 8800 with 128 CUDA cores and 768MB of memory. We measured the calculation time for single core CPU, 4 core CPU and CUDA.
In Figure 5 the calculation time is compared for different implementations. We have used logarithmic scale to better observe the differences. The simulation is repeated for all four methods for different mesh sizes (639, 2207, 3232, 5567 and 10932 nodes) and the average calculation time is given in milliseconds when running the algorithm on a single core of CPU, on 4 cores of CPU and on the GPU. Our Local Shape Matching method is faster than any other method since there is no square root operation in its calculations. On a typical processor square root requires 18 cycles while Addition Subtraction and Multiplication require 2 cycles and Division requires 12 cycles.
For all methods the calculation time grows linearly as the number of node increases. Using all 4 cores of the CPU we were able to accelerate the simulation almost three times but it will not be fast enough for real time applications if the number of node is too high. On the other hand by using CUDA we have an acceptable frame rate for all methods even when there are 10932 nodes. As it is shown the slope of the curves are almost the same for single core and multi core cpu but CUDA has resulted in a much lower slope for all methods. There is a sudden increase in CUDA calculation time for all methods around 3000. We can conclude that at lower sizes the speed is bounded by the calculation while at higher mesh sizes it is bounded by the bandwidth of the host-device transfer.

6. CONCLUSION

In this paper we have introduced an efficient framework to implement mesh-free deformable object modeling methods on CUDA. We have shown how deformable object modeling methods can be implemented in this framework. Our framework is more suitable for meshfree methods however simple mesh based methods such as Mass-Spring method can be implemented with minor adjustments. Four different methods for modeling deformable objects including the new Local Shape Matching method where included in the framework to take advantage of GPU parallel processing. We showed that while using multi-core CPUs the calculation can be accelerated, but it grows with the same rate as a single core CPU. However, we where able to achieve up to 20 times faster simulations when the number of nodes was more than tens thousands. The reason CUDA is faster than CPU is not just having more processing cores, it is also related to calculation method. When a CPU processing core is waiting for data from the memory it tries to keep itself busy by running awaiting non-dependent instructions out of order, GPU on the other hand hides the memory loading time by performing the same instruction on other threads.
We have compared the accuracy of implemented methods in linear static state by comparing the simulation results to the *Truth Cube* [9] experiment results. The Debunne method is promising since it results in low errors while it is reasonably fast. LSHM is very fast and robust however it results in high errors since it does not conserve the volume. In the future we will add volume conserving force to this method, to converge towards the desired property of volume preservation in tissue.

7. REFERENCES

[1] Nvidia cuda (compute unified device architecture) programming guide. 2008.

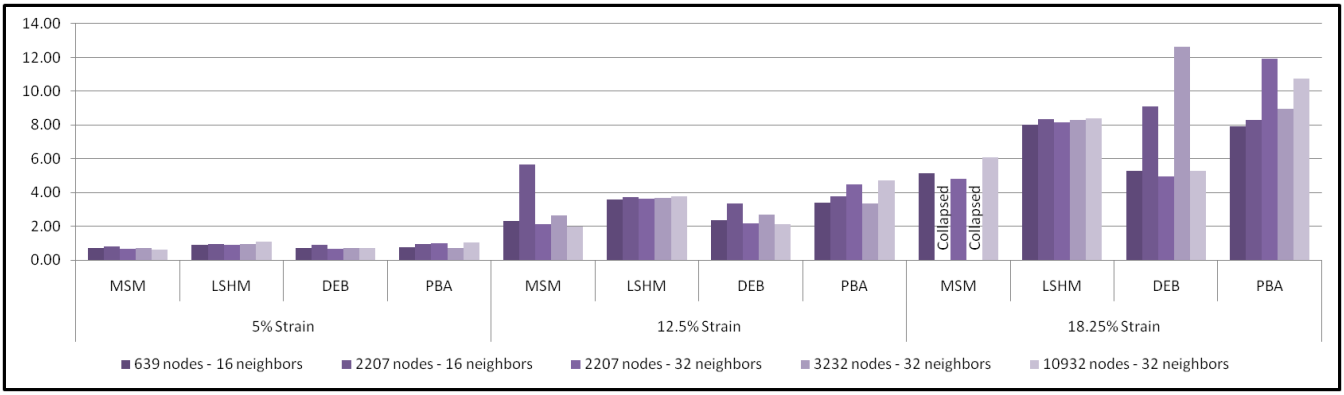


Figure 4: Accuracy of the simulation

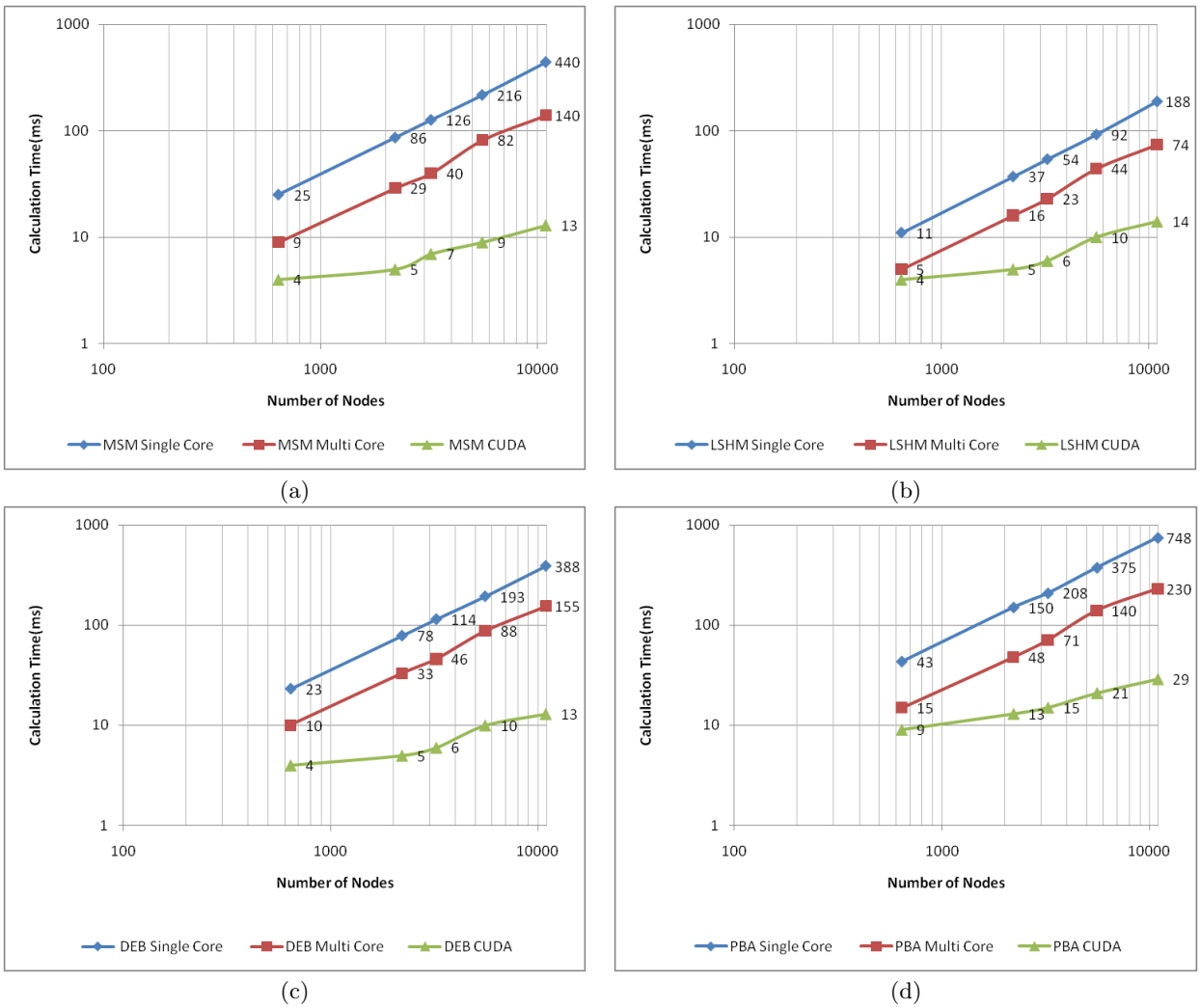


Figure 5: Comparison of calculation time for different methods when 16 neighbours were considered. (a) Mass-Spring Method. (b) Local Shape Matching. (c) Discretized Finite Element method (Debunne). (d) Point Based Animation.

[2] D. Baraff and A. Witkin. Large steps in cloth simulation. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 43–54, New York, NY, USA, 1998. ACM.

[3] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP. Portable Shared Memory Parallel Programming*. MIT Press, Massachusetts Institute of Technology, Cambridge, MA, 2007.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using cuda. *J. Parallel Distrib. Comput.*, 68(10):1370–1380, 2008.

[5] O. Comas, Z. A. Taylor, J. Allard, S. Ourselin, S. Cotin, and J. Passenger. Efficient nonlinear fem for soft tissue modelling and its gpu implementation within the open source framework sofa. In *ISBMS '08: Proceedings of the 4th international symposium on Biomedical Simulation*, pages 28–39, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] G. Debunne, M. Desbrun, A. H. Barr, and M.-P. Cani. Interactive multiresolution animation of deformable models. In N. Magnenat-Thalmann and D. Thalmann, editors, *Eurographics Workshop on Computer Animation and Simulation'99, September, 1999*, Computer Science, pages 133–144, Milan, Italie, Sept. 1999. Springer.

[7] J. Georgii and R. Westermann. Mass-spring systems on the gpu. *Elsevier Science*, July 2005.

[8] S. F. F. Gibson and B. Mirtich. A survey of deformable modeling in computer graphics. Technical report, Mitsubishi Electric Research Laboratories, 1997.

[9] A. E. Kerdok, S. M. Cotin, M. P. Ottensmeyer, A. M. Galea, R. D. Howe, and S. L. Dawson. Truth cube: Establishing physical standards for soft tissue simulation. *Medical Image Analysis*, 7(3):283 – 291, 2003. Functional Imaging and Modeling of the Heart.

[10] K. Kähler, J. Haber, and H. peter Seidel. Geometry-based muscle modeling for facial animation. In *In Proc. Graphics Interface 2001*, pages 37–46, 2001.

[11] J. Lasseter. Principles of traditional animation applied to 3d computer animation. *SIGGRAPH Comput. Graph.*, 21(4):35–44, 1987.

[12] S. Li and W. K. Liu. *Meshfree Particle Methods*. Springer Publishing Company, Incorporated, 2007.

[13] M. Müller, B. Heidelberger, M. Teschner, and M. Gross. Meshless deformations based on shape matching. *ACM Trans. Graph.*, 24(3):471–478, 2005.

[14] M. Müller, R. Keiser, A. Nealen, M. Pauly, M. Gross, and M. Alexa. Point based animation of elastic, plastic and melting objects. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 141–151, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association.

[15] A. Nealen, M. Muller, R. Keiser, E. Boxerman, and M. Carlson. Physically based deformable models in computer graphics. *Computer Graphics Forum*, 25:809–836(28), December 2006.

[16] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[17] M. Peercy, M. Segal, and D. Gerstmann. A performance-oriented data parallel virtual machine for gpus. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, page 184, New York, NY, USA, 2006. ACM.

[18] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: surface elements as rendering primitives. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 335–342, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[19] X. Provot. Deformation constraints in a spring-mass model to describe rigid cloth behavior. In *Proceedings of Graphics Interface*, pages 147–154, 1995.

[20] A. Rasmusson, J. Mosegaard, and T. S. Sørensen. Exploring parallel algorithms for volumetric mass-spring-damper models in cuda. In *ISBMS '08: Proceedings of the 4th international symposium on Biomedical Simulation*, pages 49–58, Berlin, Heidelberg, 2008. Springer-Verlag.

[21] Z. Taylor, M. Cheng, and S. Ourselin. High-speed nonlinear finite element analysis for surgical simulation using graphics processing units. *Medical Imaging, IEEE Transactions on*, 27(5):650–663, May 2008.

[22] D. Terzopoulos and K. Fleischer. Modeling inelastic deformation: viscolelasticity, plasticity, fracture. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 269–278, New York, NY, USA, 1988. ACM.

[23] D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer. Elastically deformable models. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 205–214, 1987.

[24] S. Zhang, L. Gu1, P. Huang, and J. Xu. Real-time simulation of deformable soft tissue based on mass-spring and medial representation. In *Computer Vision for Biomedical Image Applications: First International Workshop, CVBIA 2005*, pages 419–426. Springer-Verlag, 2005.

ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs

Karl Rupp
CD Laboratory for Reliability Issues in Microelectronics
IuE, TU Wien, A-1040 Wien
rupp@iue.tuwien.ac.at

Florian Rudolf
Institute for Microelectronics
Gußhausstraße 27-29/E360
TU Wien, A-1040 Wien
rudolf@iue.tuwien.ac.at

Josef Weinbub
Institute for Microelectronics
Gußhausstraße 27-29/E360
TU Wien, A-1040 Wien
weinbub@iue.tuwien.ac.at

ABSTRACT

The vast computing resources in graphics processing units (GPUs) have become very attractive for general purpose scientific computing over the past years. Moreover, central processing units (CPUs) consist of an increasing number of individual cores. Most applications today still make use of a single core only, because standard data types and algorithms in wide-spread procedural languages such as C++ make use of a single core only. A customized adaption of existing algorithms to parallel architecture requires a considerable amount of effort both from algorithmic and programming point of view. Taking this additional amount of work hours required for an adaption to GPUs starting from scratch into account, the use of GPUs may not pay off on the overall.

The Vienna Computing Library (ViennaCL), which is presented in this work, aims at providing standard data types for linear algebra operations on GPUs and multi-core CPUs. It is based on OpenCL, which provides unified access to both GPUs and multi-core CPUs. The ViennaCL API following existing programming and interface conventions established with uBLAS, which is part of the peer-reviewed Boost library. Thus, the open source library can be easily integrated into existing C++ implementations and therefore reduces the necessary code changes in existing software to a minimum. In addition, algorithms provided with ViennaCL can directly be used with uBLAS types due to the common interface.

The algorithmic focus of ViennaCL is on iterative solvers, which are often used for the solution of large systems of linear equations typically encountered in the discretization of partial differential equations using e.g. finite element methods. Benchmark results given in this work show that the performance gain of ViennaCL over uBLAS is on both GPUs and multi-core CPUs up up to an order of magnitude. For small amounts of data, the use of ViennaCL may not pay off due to an OpenCL management overhead associated with the launch of compute kernels.

1. INTRODUCTION

General purpose scientific computing on GPUs has become very attractive over the past years [11–13, 18]. In the early days of such computations, the lack of double precision arithmetic was often considered a major drawback. However, recent GPUs such as a NVIDIA Geforce GTX 470 or an ATI Radeon HD 5850 used for the benchmarks in this work do not suffer from this restriction any longer, thus they push into the field of high performance computing (HPC). Simultaneously, CPUs consist of an increasing number of cores, for which many serial algorithms become less and less attractive.

Considerable performance gains have been reported [11–13, 18], but the adaption of existing algorithms to GPUs starting from scratch requires a considerable amount of change in existing codes to account for the highly parallel architecture of GPUs. Consequently, the effort required for porting an existing code to GPUs was often considered to be too large to have a considerable benefit on the overall. In particular, programmers are required to learn specialized programming languages like CUDA [14] or OpenCL [20], even if only standard linear algebra algorithms such as defined by the basic linear algebra subprograms (BLAS) [2] are to be executed on the GPU. It is thus desirable to have data types that provide parallel standard operations on the target machine, utilizing the available hardware in the best possible way.

There is a number of linear algebra libraries for GPUs available, for example ACML-GPU [1], CULA [4], MAGMA [6] or CUBLAS [14], focusing on computationally demanding operations such as dense matrix-matrix multiplications. However, sparse matrix arithmetic and iterative solvers are much less pronounced or not provided at all, even though this type of matrices is common for the discretization of partial differential equations. Cusp [5] provides two iterative solvers, but only matrix-vector products are computed on the GPU, leading to considerable memory transfer overhead. The CNC plugin in OpenNL [7] provides only a single iterative solver. The functionality provided by these libraries is available through function calls, which provide a certain set of basic operations. Thus, appropriate data setup and initialization is typically left to the user. The approach of the Vienna Computing Library (ViennaCL) [9] presented in this work is to wrap GPU data in high level C++ datatypes and provide an interface that adheres to established conventions. In the following, we refer to version 1.0.5 of the library.

This paper is organized as follows: First, the design of ViennaCL is discussed in Sec. 2. Sec. 3 presents the library interface for linear algebra operations on BLAS levels 1 and 2. The iterative solvers provided with ViennaCL are explained in Sec. 4. The inclusion of custom compute kernels is discussed in Sec. 5 and benchmark results are given in Sec. 6. Finally, an outlook to future work is given in Sec. 7 and a conclusion is drawn in Sec. 8.

2. DESIGN OF VIENNACL

The roots of ViennaCL are in the need for fast iterative solvers for the solution of large sparse systems arising from the discretization of partial differential equations (PDEs) for use in our in-house simulators. To allow other researchers and engineers to benefit from our effort, ViennaCL is designed to be used with other modern software packages that serve a similar purpose, e.g. deal.ii [15], Getfem++ [17] or Sundance [25], which are all implemented in C++. Consequently, C++ is chosen for the implementation of ViennaCL.

For accessing GPUs, the two main branches are CUDA [14] and OpenCL [20]. While CUDA is tailored to the specific architecture of NVIDIA GPUs, the first royalty-free standard for cross-platform parallel programming, OpenCL, provides much higher flexibility with respect to the underlying hardware. Thus, OpenCL supports a superset of the hardware supported by CUDA and is not limited to GPUs. Moreover, CUDA kernels need to be precompiled by a separate compiler, while OpenCL allows just-in-time compilation of the source code on the target machine. The latter approach is especially attractive for developers, because this allows to create header-only libraries. For these reasons, OpenCL is chosen for low level hardware programming.

The major design goal of ViennaCL is to be convenient and easy to use. For simple integration into existing projects, ViennaCL is a header-only library, which simplifies the build process considerably. On the other hand, initialization and management of OpenCL is done completely in the background and is discussed in the following subsections.

2.1 Hardware Initialization

A common approach in parallel software such as PETSc [10] is to rely on dedicated initialization routines that have to be called by the library user prior to any use of other functionality. In ViennaCL, hardware initialization is automatically triggered when the first object of a type provided by ViennaCL such as `scalar` or `vector` is created. In the background, available devices are queried. If a suitable GPU is available, it is then used for all calculations, otherwise ViennaCL searches for a CPU supported by the OpenCL implementation. The simultaneous use of multiple devices is not included in version 1.0.5 of ViennaCL, because multi-device support was added to OpenCL only recently [20].

2.2 Source Code Compilation

The compilation of OpenCL source code at each run of ViennaCL leads to additional setup costs during the automatically triggered initialization phase. A full compilation of all OpenCL sources included in ViennaCL takes several seconds and may be too long for certain applications. Therefore, ViennaCL groups sources into smaller compilation units associated with the basic types and the underlying floating point precision. This allows a on-demand compilation: The first time an object of a particular type is created, all OpenCL kernels associated with that particular type are compiled. A more fine-grained compilation on a per kernel basis, which compiles a kernel at the first invocation, turned out to have larger overall setup costs in most cases. This just-in-time compilation reduces setup times to a bare minimum.

2.3 Transfer between Host and Device

Prior to any calculations on GPUs, the data needs to be transferred from the host memory to the OpenCL device memory (e.g. GPU RAM). Even if ViennaCL is used on multi-core CPUs, data also needs to be set up accordingly in the OpenCL layer.

Since every data transfer from host memory to device memory and back from the device memory to host memory can be seen as a copy operation, ViennaCL reuses the conventions introduces with the Standard Template Library (STL) (see e.g. [24]). In order to copy all entries of a vector `cpu_vec` from the host to a vector `gpu_vec` in the GPU memory, the call

```
1 copy(cpu_vec.begin(),
2      cpu_vec.end(),
3      gpu_vec.begin());
```

is sufficient. The member functions `begin()` and `end()` return iterators pointing to the beginning and the end of the vector respectively. Thus, programmers acquainted with the iterator concept and the STL can reuse their knowledge. Moreover, parts of a vector can be manipulated easily and also plain pointers to CPU data can be supplied. A shorthand notation for the above code line is

```
1 copy(cpu_vec, gpu_vec);
```

which only requires that the `begin()` and `end()` member functions are available for the respective type of `cpu_vec`.

For dense matrix types, the iterator concept could also be used in principal, but matrix dimensions would have to be supplied in addition. Instead, data transfer from a matrix `cpu_matrix` on the host, no matter if dense or sparse, to a matrix `gpu_matrix` on the device is accomplished with

```
1 copy(cpu_matrix, gpu_matrix);
```

For this generic interface a number of type requirements needs to be imposed on the type of the dense `cpu_matrix`, which are as follows:

- A member function `size1()` provides the number of rows
- A member function `size2()` provides the number of columns
- Entries are accessed using the parenthesis operator with index range starting at zero.

These conventions are fulfilled by uBLAS types, so data set up in a dense uBLAS matrix can exchanged with ViennaCL with a single line of code. Library users willing to use a dense matrix type not fulfilling these requirements have to provide a wrapper class.

For sparse matrix types, instead of overloaded parenthesis operators, data must be accessible via iterators as in uBLAS [8]. As an alternative using only STL types, a sparse matrix can also be supplied in a vector of maps, i.e.

```
1 vector< map<unsigned int, NumericT> >
```

where `NumericT` is either `float` or `double`.

To modify individual entries of a vector or a dense matrix located on the OpenCL device, overloaded operators are provided. Sparse matrix types cannot be manipulated directly in OpenCL memory in ViennaCL 1.0.5. For example, setting the fifth element of a vector `gpu_vec` to seven, the line

```
1 gpu_vec(4) = 7;
```

is sufficient. Note that the indices start with zero. Under the hood, the parenthesis operator in `gpu_vec(4)` returns a proxy class, for which the assignment operator is overloaded and the transfer from host to device is initiated. However, direct initialization of all entries on the GPU as in

```
1 // one possible initialization of device
2 // memory (not recommended!)
3 for (int i=0; i<100000; ++i)
4   gpu_vec(i) = i;
```

is not recommended, because each update initiates a separate transfer with a significant overhead. Thus, the loop above takes four to five orders of magnitude longer than for pure host types. A much faster alternative is

```
1 std::vector<NumericT> cpu_vec(100000);
2 for (int i=0; i<100000; ++i)
3   cpu_vec(i) = i;
4 copy(cpu_vec, gpu_vec);
```

which has only small overhead due to creation of the temporary vector `cpu_vec` and the copy operation at the end of the for-loop. Consequently, it is recommended to fully set up the data (i.e. vectors, matrices) on the CPU host, then copy to the device and start processing the data with ViennaCL.

2.4 Kernel Execution

To start an OpenCL kernel, arguments need to be set and several parameters need to be supplied using the C interface. In ViennaCL, however, operator overloads and other abstraction mechanisms in C++ allow an encapsulation of all these details. For example, the addition of two vectors `vec2` and `vec3`, typically written in C++ using operator overloads as

```
1 vec1 = vec2 + vec3;
```

requires the launch of the appropriate OpenCL kernel with the memory locations and the vector lengths as kernel arguments. All these details are encapsulated, so that users of ViennaCL do not have to deal with OpenCL internals.

3. BASIC LINEAR ALGEBRA

There are many linear algebra libraries available in C++, one of the most commonly used is uBLAS [8] included in the peer-reviewed Boost libraries [3]. In contrast to early implementations of BLAS functionality in FORTRAN, overloaded operators are used in uBLAS whenever appropriate. ViennaCL accounts for the broad acceptance of the approach by uBLAS and provides an interface that is to a large extent a subset of that of uBLAS. More precisely, any code for algorithms using linear algebra operations from ViennaCL is required to be also valid when using uBLAS objects. This simplifies testing and verification on the one hand and is a benefit for uBLAS library users due to reusable algorithms on the other hand.

The basic types used for linear algebra operations on BLAS level 1 and 2 are the following:

```
1 scalar<NumericT> s; //scalar
2 vector<NumericT> v; //vector
3 matrix<NumericT> m; //dense matrix
4 compressed_matrix<NumericT> c1; //CSR
5 coordinate_matrix<NumericT> c2; //(i,j,aij)
```

Here, `NumericT` denotes the underlying floating point type (either `float` or `double`). The `compressed_matrix` type stores a sparse matrix in a compressed sparse rows format (see e.g. [21]), while `coordinate_matrix` stores all matrix entries as triplets (i, j, a_{ij}), where i is the row index, j is the column index and a_{ij} is the corresponding entry.

BLAS functionality in ViennaCL can be invoked similarly to uBLAS using overloaded operators:

```
1 // BLAS level 1
2 // x, y and z are vectors
3 y = 2.0 * x;
4 z = x + y;
5 x += 3.1415 * z;
6 NumericT n1 = norm_1(x);
7 NumericT n2 = norm_2(y);
8 NumericT ninf = norm_inf(z);
9 plane_rotation(x, y, n1, n2);
```

The first three code lines manipulate vectors using overloaded operators. Unlike in naive C++, where expressions like `x += 3.1415 * z;` would lead to a temporary object for `3.1415 * z`, none of the expressions above leads to a temporary object due to the use of expression templates [27, 28]. Internally, only a single multiply-add kernel is called for this example with vectors arguments `x`, `z` and scalar argument `3.1415`. Temporary objects on GPUs are much more detrimental for performance and should thus be avoided, since allocation has to be done via the OpenCL layer. Lines 6 to 8 in the above snippet compute the l^1-, l^2- and l^∞-norm of the respective function argument. The last line performs a plane rotation of the argument vectors as required by BLAS level 1.

On BLAS level 2, ViennaCL and uBLAS are also fully compatible:

```
1 // BLAS level 2
2 // x, y are vectors, A is a matrix
3 y = prod(A, x);       //matrix-vector product
4 x = prod(trans(A), x);  //transposed product
5 y = alpha * prod(A, x) + beta * y
6 y = solve(A, x, tag);   //triangular solver
7 inplace_solve(A, x, tag);
8 A += alpha * outer_prod(x,y); //rank1 update
```

Lines 3 to 5 show matrix vector products are handled. Lines 6 and 7 call a triangular solver for dense matrices, where the variable `tag` is either `upper_tag`, `lower_tag`, `unit_upper_tag` or `unit_lower_tag` and is used to choose the dense linear solver.

4. ITERATIVE SOLVERS

In many applications such as the discretization of partial differential equations using finite element or finite difference methods, large sparse systems of linear equations need to be solved. While direct methods can be used for moderate problem sizes, iterative solvers are necessary for large systems of equations. The BLAS levels defined for sparse matrices [16] are not fully implemented in ViennaCL 1.0.5 yet, but the most important sparse operation, namely sparse matrix vector products, is provided and serves as a building block for iterative solvers.

The choice of a suitable iterative solver strongly depends on the properties of the system of linear equations. ViennaCL 1.0.5 provides the following three iterative solvers, which cover most application areas:

- Conjugate Gradient (CG) [19] for the solution of symmetric, positive definite systems.
- Stabilized Bi-Conjugate Gradient (BiCGStab) [26] for positive definite systems.
- Generalized Minimum Residual (GMRES) [22, 29] for general systems.

Since no iterative solvers are provided by uBLAS, the interface for the iterative solvers was designed such that it naturally extends the existing solver interface for the triangular solvers. In ViennaCL, the BLAS level 2 call for dense matrices

```
y = solve(A, x, tag);
```

is extended to support the additional tags `cg_tag`, `bicgstab_tag` and `gmres_tag`, hence the solvers can be called using

```
// CG solver:
result = solve(matrix, rhs, cg_tag());
// BiCGStab solver:
result = solve(matrix, rhs, bicgstab_tag());
// GMRES solver:
result = solve(matrix, rhs, gmres_tag());
```

Additional solver parameters can be passed to the constructors of these tags to specify tolerances and maximum iteration counts. For example, a relative tolerance of 10^{-8} and at most 200 iterations for a CG solver can be set with the line

```
result=solve(matrix,rhs, cg_tag(1e-8,200) );
```

Since uBLAS and ViennaCL are mostly interface compatible, the generic implementation of the iterative solvers allows to directly reuse them with uBLAS types. Thus, the same iterative solver code allows to run the iterative solver either on GPUs or multi-core CPUs using ViennaCL or on a single CPU core using uBLAS. For other matrix and vector types, a wrapper facility allows library users to customize free functions such as `prod()` for matrix-vector products, `norm_2()` for computing the l^2-norm or `inner_prod()` for computing inner products to fit other matrix and vector types from external libraries.

The convergence of iterative solvers can be greatly improved by the use of preconditioners. ViennaCL 1.0.5 provides an optional incomplete LU factorization (ILUT) preconditioner with threshold [21], other preconditioners are in preparation. The ILUT preconditioner is due to its inherent serial structure always computed and applied on the CPU, thus the preconditioner is likely to serve as a bottleneck for an otherwise GPU accelerated iterative solver.

Preconditioners are supplied as an optional fourth argument to the function `solve()`. For example, an ILUT preconditioner can be used within a conjugate gradient solver by writing

```
// Set up ILUT
ilut_precond< compressed_matrix<NumericT> >
  ilut(matrix, ilut_tag());

// CG solver with ILUT preconditioner:
result = solve(matrix, rhs, cg_tag(), ilut);
```

Additional parameters for ILUT can be provided to the constructor of `ilut_tag` similar to the specification of parameters in solver tags. Again, the preconditioner can be used both for uBLAS types and for ViennaCL types. The generic solver interface also allows to provide custom preconditioners, the only requirement is that the parenthesis operator is defined for a vector argument.

5. CUSTOM COMPUTE KERNELS

Unlike other libraries, ViennaCL directly supports user-defined compute kernels written in OpenCL. The user can fully focus on the kernel, since details of the underlying OpenCL implementation are handled internally by ViennaCL.

For example, a kernel for elementwise products of two vectors is the following:

```
__kernel void elementwise_prod(
          __global const float * vec1,
          __global const float * vec2,
          __global float * result,
          unsigned int size)
{
  for (int i = get_global_id(0);
           i < size;
           i += get_global_size(0))
    result[i] = vec1[i] * vec2[i];
}
```

`vec1` and `vec2` denote the operands, `result` is the result vector and `size` the length of the vectors. Details on the OpenCL programming language, which is a subset of C with some extensions for parallelism, can be found in the specification [20], where in particular the keywords `__kernel`, `__global` and the functions `get_global_id()` and `get_global_size()` are explained. With a few additional code lines, the above kernel can be launched for three ViennaCL vectors of type `vector<float>`.

The possibility to easily include custom compute kernels in ViennaCL allows to run a long chain of possibly custom operations on the GPU without the overhead of copying data between host and device. For example, a custom matrix-vector multiplication kernel could be required for a specialized matrix of type, say, `A`. After writing the custom OpenCL kernel and overloading

Compute Device	float	double
Intel i7 960, single core	0.33	0.32
Intel i7 960, ViennaCL	**1.98**	0.85
NVIDIA Geforce GTX 470	1.88	**1.66**
ATI Radeon HD 5850	0.86	0.89

Table 1: Computational speed (in GFLOPs) for inner products of vectors with 3 000 000 entries. Multiply-add operations are counted as single floating point operations.

Compute Device	float	double
Intel i7 960, single core	0.17	0.16
Intel i7 960, ViennaCL	1.06	0.81
NVIDIA Geforce GTX 470	**1.71**	**1.10**
ATI Radeon HD 5850	1.30	0.93

Table 2: Computational speed (in GFLOPs) for sparse matrix-vector multiplication using `compressed_matrix`. The 65 025 matrix rows have seven nonzero entries on average. Multiply-add operations are counted as single floating point operations.

```
1 prod(A & a, vector<T> & b);
```

for matrix-vector products, objects of type `A` can directly be passed to the iterative solvers provided. Thus, the possibility to provide custom compute kernels and the generic implementation of the algorithms in ViennaCL result in high flexibility for the library user.

6. PERFORMANCE

The performance of ViennaCL, version 1.0.5, is compared on GPUs from ATI and NVIDIA and a CPU from Intel. uBLAS is used to measure the performance on a single CPU core. The test platform was a Intel Core i7 960 with 4 physical cores, 8 logical cores, and 6 Gigabytes of random access memory, running a 64-bit Linux kernel. Stream SDK 2.2 was used with kernel of version 2.6.33 and GPU driver version 10.6. The Stream SDK was also used for running ViennaCL in parallel on the CPU. We observed that benchmark results for ViennaCL using Stream SDK under Windows 7 are by up to 30 percent better, especially when using double precision, hence the performance of ViennaCL is likely to improve with better OpenCL support in the future. For NVIDIA GPUs, a kernel with version 2.6.34 and a GPU driver, version 195.36.24, was used. When evaluating the following benchmark results in computational speed per money, it has to be considered that the CPU is by a factor of around two more expensive than each of the GPUs. All compute kernels are launched with the default settings in ViennaCL, namely 128 work groups with 128 work items each.

In Tab. 1 benchmark results for inner products are shown. Performance gains on GPUs and a fully loaded multi-core CPU of a factor of up to six compared to a single CPU core are observed. In double precision, the parallel execution on the CPU still results in a performance gain of a factor 2.6. A curiosity is that the OpenCL implementation of the Stream SDK provides better performance in double precision than in single precision on the GPU. We assume that this is due to the early stage of OpenCL support by ATI.

Execution times for matrix-vector products in Tab. 2 depict that the performance benefit over a single CPU core is around a factor of ten in single precision and about a factor of seven in double precision. Running ViennaCL on the CPU results in about 60 to 90 percent of the performance of the GPUs. We note that additional notable performance gains on GPUs can be obtained by the use of hybrid formats [11, 12], which are not included in ViennaCL yet. Additionally, we observed that the use of vector data types in the OpenCL kernels doubles performance on the GTX 470 in this case.

Tab. 3 lists the execution times for two iterative solvers. The CG solver is accelerated by a factor of five on the NVIDIA GPU, and only slightly on the ATI GPU. The performance gain for BiCGStab is comparable to that of CG. Using the ATI Stream SDK, a significant overhead of OpenCL kernel launches becomes apparent: While the performance of sparse matrix vector products, inner products and vector additions is comparable on the two GPUs, a call of several different kernels has a much larger overhead using the Stream SDK than for the NVIDIA implementation of OpenCL.

The observed performance gains of GPUs over CPUs for iterative solvers are essentially determined by the available memory bandwidth, because the iterative solvers use BLAS level 1 and 2 functions only. Further speedups can possibly be obtained if parts of the assembly algorithm for the linear system of equations are also ported to OpenCL. Higher performance gaps are usually observed for BLAS level 3 functions, e.g. matrix-matrix products, which are computationally more demanding than lower BLAS levels. However, no BLAS level 3 functionality is provided in version 1.0.5 of ViennaCL, but planned for future versions.

7. OUTLOOK

With the possibility of using ViennaCL on many different platforms, a global number of work groups and work items is not sufficient to yield reasonable performance on all target devices. While the choice is easier on CPUs due to the smaller number of cores on a die, it has a much stronger influence on GPUs. In particular, a higher number of work items or work groups does not necessarily result in better performance due to synchronization overhead. Thus, work on an automated tuning environment is in progress, which aims at finding the best set of parameters for each compute kernel. Performance gains of about 25 percent have already been observed for the operations compared in Sec. 6.

Having fast sparse matrix vector product available, an implementation of eigenvalue computations using either Lanczos' or Arnoldi's method is in progress. Simple implementations often suffer from severe round-off errors that introduce so-called ghost eigenvalues, therefore orthogonality of the Krylov basis has to be ensured by e.g. partial reorthogonalization [23]. The time consuming matrix-vector and inner products can then be carried out on the GPU or the CPU in parallel.

Compute Device	CG, float	CG, double	BiCGStab, float	BiCGStab, double
Intel i7 960, single core	0.23	0.21	0.25	0.22
Intel i7 960, ViennaCL	0.73	0.44	0.52	0.33
NVIDIA Geforce GTX 470	**1.15**	**0.87**	**1.16**	**0.73**
ATI Radeon HD 5850	0.40	0.35	0.20	0.22

Table 3: Computational speed (in GFLOPs) for the CG and BiCGStab solvers without preconditioner. The 65 025 matrix rows have seven nonzero entries on average. Multiply-add operations are counted as single floating point operations.

8. CONCLUSIONS

The newly released open source library ViennaCL is presented in this work. It allows to use the huge computational resources of both GPUs and multi-core CPUs without going into the details of the underlying hardware. Thanks to a common programming interface with uBLAS, ViennaCL library users benefit on the one hand from the reuse of a widely accepted programming interface and on the other hand from the implementation of the three iterative solvers CG, BiCGStab and GMRES provided by ViennaCL, which can also directly be used with uBLAS types as well as with linear algebra types from other libraries using the generic wrappers provided. Benchmarks show that the library provides good performance on both GPUs and multi-core CPUs for large amounts of data. Performance gains of up to a factor of ten compared to a single CPU core can be observed for common linear algebra operations. Due to the use of OpenCL, ViennaCL can be run on many different parallel architectures.

9. ACKNOWLEDGEMENTS

Karl Rupp gratefully acknowledges support by the Graduate School PDEtech at the Vienna University of Technology. The authors wish to thank Prof. Siegfried Selberherr for providing a test platform for benchmarking and regression tests. This work has been supported by the European Research Council through the grant #247056 MOSILSPIN.

10. REFERENCES

[1] AMD Core Math Library for GPUs. http://developer.amd.com/gpu/acmlgpu/pages/default.aspx.
[2] BLAS homepage. http://www.netlib.org/blas/.
[3] Boost C++ Libraries. http://www.boost.org/.
[4] CULA. http://www.culatools.com/.
[5] Cusp. http://code.google.com/p/cusp-library/.
[6] MAGMA - Matrix Algebra on GPU and Multicore Architectures. http://icl.cs.utk.edu/magma/.
[7] Open Numerical Library. alice.loria.fr/index.php/software/4-library/23-opennl.html.
[8] uBLAS Library. http://www.boost.org/doc/libs/release/libs/numeric/ublas/.
[9] ViennaCL. http://viennacl.sourceforge.net/.
[10] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. Curfman McInnes, B. F. Smith, and H. Zhang. PETSc Web Page. http://www.mcs.anl.gov/petsc/.
[11] M. M. Baskaran and R. Bordawekar. Optimizing Sparse Matrix-Vector Multiplication on GPUs. *IBM RC24704*, 2008.
[12] N. Bell and M. Garland. Efficient Sparse Matrix-Vector Multiplication on CUDA. *NVIDIA Technical Report NVR-2008-004*, 12, 2008.
[13] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Trans. Graph.*, 22:917–924, July 2003.
[14] NVIDIA CUDA. http://www.nvidia.com/object/cuda_home_new.html.
[15] deal.II . http://www.dealii.org/.
[16] I. S. Duff, M. A. Heroux, and R. Pozo. The Sparse BLAS. *Technical Report TR/PA/01/24*, Sept. 2001.
[17] Getfem++. http://home.gna.org/getfem/.
[18] D. Göddeke, R. Strzodka, and S. Turek. Accelerating Double Precision FEM Simulations with GPUs. *Proceedings of ASIM 2005 - 18th Symposium on Simulation Technique*, 2005.
[19] M. R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 49, 1952.
[20] Khronos OpenCL. http://www.khronos.org/opencl/.
[21] Y. Saad. *Iterative Methods for Sparse Linear Systems, Second Edition*. Society for Industrial and Applied Mathematics, April 2003.
[22] Y. Saad and M. H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, 1986.
[23] H. D. Simon. The Lanczos Algorithm with Partial Reorthogonalization. *Mathematics of Computation*, 42(165):115–142, January 1984.
[24] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
[25] Sundance. http://www.math.ttu.edu/ klong/Sundance/html/.
[26] H. A. van der Vorst. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Non-Symmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, 12:631–644, 1992.
[27] D. Vandevoorde and N. M. Josuttis. *C++ Templates*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
[28] T. Veldhuizen. Expression Templates. *C++ Report*, 7(5):26–31, June 1995.
[29] H. F. Walker and L. Zhou. A Simpler GMRES. *Numer. Linear Algebra Appl.*, 1(6):571–581, 1994.

Dynamic Work Scheduling for GPU Systems

Miguel Angel Lastras-Montaño
Universidad Autónoma de San Luis Potosí
Álvaro Obregón 64, SLP, México
mlastras@gmail.com

Maged M. Michael
IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598, USA
magedm@us.ibm.com

J. Alan Bivens
IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598, USA
jbivens@us.ibm.com

ABSTRACT

Graphics processing units (GPUs) lack the capability to migrate active threads from busy processors to idle processors. This makes irregular applications, where the amount of work per thread is unpredictable, vulnerable to severe load imbalance, if work assignment to threads is determined statically.

In this paper we present a framework for efficient dynamic work scheduling in GPUs, that enables general purpose irregular applications to balance the load among threads. We exploit the performance characteristics of the GPU memory hierarchy as well as synchronization operations and utilize a combination of private and shared work stealing structures at different memory levels. To complement our dynamic work scheduling framework, we propose and study thread termination and "victim" selection policies.

To evaluate the performance of our framework, a benchmark that solves the shortest-path problem was built. A performance comparison between our dynamic work scheduling framework and a version of the benchmark that assigns work to threads statically is done for a variety of parameter values. Our results indicate that our dynamic work scheduling framework outperforms static scheduling by a factor of up to 2× using graphs with a uniform random distribution and up to 3× utilizing graphs with a geometric distribution.

1. INTRODUCTION

Modern Graphics Processing Units (GPUs) have evolved in such a way that they achieve their maximum computational power in applications that are compute-intensive and highly parallel [9]. In these specialized many-core processors, it is vital to keep all the processing cores busy, since GPUs hide their memory access latency maintaining active threads performing high arithmetic intensity calculations [1, 6, 10].

In the current GPUs' model, a static low-overhead hardware scheduler is responsible for scheduling threads on processors [7, 8]. However, the model lacks the capability to migrate active threads from busy to idle processors. This creates a vulnerability for applications (see for instance Ref. [5]) where the amount of work per thread –or processing unit– is unpredictable and variable as it may lead to severe load imbalance.

To enable GPUs for general-purpose computing (known as GPGPU), GPU manufacturers have given hardware support for atomic operations. With the use of atomic operations such as atomicCAS (Compare-and-Swap) and atomicExch (Exchange), it is possible to implement, among other things, mutual exclusion locks that could be used to enable multiple threads to access shared data consistently.

In this work we present a framework for efficient dynamic work scheduling in GPUs that overcomes the load imbalance problem of irregular applications. In this framework we utilize mutual exclusion locks to protect shared work stealing structures in the larger but slower GPU memory and we also take advantage of small private work structures in the low-latency GPU memory to minimize the overhead of work scheduling. Despite the additional overhead generated by the use of mutual exclusion locks, our results show that the migration of work effectively balances the load among threads with significant performance gains.

Since the execution of a function on a GPU terminates only when the last thread ends its execution, the policies for threads to decide whether to end their execution or attempt to steal work form unfinished threads, as well as which *victim* thread to steal from, have deep impact on the performance of an application. We compare two stealing policies; the first chooses victims in a consecutive order while the second approach selects victims randomly.

In order to test the performance of our framework and of our different termination and victim selection policies, we built a benchmark that solves the shortest-path problem.

This paper is structured as follows: in Section 2 we describe the GPU's hardware and software organization and their implications. In Section 3 we present our framework for dynamic work scheduling as well as our thread termination and victim selection policies. Details and results of our benchmark can be found in Section 4 where we show the capabilities of our dynamic work scheduling framework compared to static scheduling. Conclusions can be found in Section 5.

2. BACKGROUND

2.1 Hardware organization

The NVIDIA GTX 280 graphics processing unit used in this work has a total of 240 scalar processor cores organized in 30 multiprocessor units with 8 scalar processors each. Broadly, it has two types of memory: a local low-latency memory called *shared memory* of 16 KB inside each multiprocessor unit and a larger but slower memory called *global memory* accessible by all the active threads.

2.2 Software Organization

Functions that run on the GPU side are called *kernels*. To invoke a kernel we must define the total number of threads that we want to use in the kernel. Essentially, we have to define the number of equal-sized *blocks of threads* that we want to use in the kernel, and naturally, we need to indicate the size of the blocks, i.e., the number of threads per block.

2.3 Hardware/Software Implications

The maximum number of threads per block is 512 and the maximum number of active blocks –concurrently active in the GPU– is limited by the number of available multiprocessors. A complete block must reside in the same multiprocessor and up to 8 blocks may be running concurrently per multiprocessor; that is the case when there is one in each scalar processor; however, the maximum number of blocks that could be launched in a single kernel may exceed the maximum number of active blocks.

The GPU's shared memory only allows data sharing between threads in the same block, threads of different blocks cannot communicate through shared memory regardless of whether they are on the same multiprocessor or not. If an inter-block thread-cooperation is required, a higher-latency global memory is available and visible to all the active threads in the GPU.

Each block of threads is automatically split by the multiprocessor into groups of 32 threads called *warps*. When a warp is available, the multiprocessor executes in lockstep one common instruction in the warp in a *single-instruction multiple-data* fashion called *SIMT* (single-instruction, multiple-thread). When threads in the same warp have different instructions, the multiprocessor SIMT unit executes each branch separately, and therefore, the full efficiency is a-chieved when all the threads in the warp share the same instruction.

2.4 Atomic Operations

Some high-end NVIDIA GPU devices (with compute capability 1.1 and above) are capable of performing atomic operations. Among other operations, NVIDIA gives support for the atomic Compare-and-Swap (`atomicCAS`), atomic Exchange (`atomicExch`) and `atomicMin` operations defined as follows [1]:

- `atomicCAS(int* addr, int comp, int val)`: Atomically reads `old` (located in `addr`), computes `(old==comp ? val : old)`, stores the result in `addr` and returns `old`.
- `atomicExch(int* addr, int val)`: Atomically reads `old` (located in `addr`), stores `val` back to `addr` and returns `old`.
- `atomicMin(int* addr, int val)`: Atomically reads `old` (located in `addr`), computes the minimum of `old` and `val`, stores the result back to `addr` and returns `old`.

2.5 Barrier Synchronizations

Due to the lockstep execution of instructions in the warps, threads within warps are implicitly synchronized. A block-level synchronization is available with the intrinsic function `__syncthreads()`; it acts as a barrier that all the threads within a block have to reach to continue with their execution. An inter-block synchronization is also possible using the global memory, but should be handled with extra care to avoid deadlocks.

The self-controlled hardware scheduler launches blocks on available multiprocessors, these blocks cannot migrate to other multiprocessors and they will not free up resources until they complete their execution. A potential deadlock condition may occur if we force a synchronization between all the blocks, specially if we use more blocks than the capacity of the available multiprocessors, which is a normal operating condition for GPU devices.

2.6 Memory Fences

When a thread performs a series of memory accesses, other threads may observe an order for these accesses different from the original. To avoid this, the fence function `__threadfence()` ensures that an access to global and shared memory will be visible to all the threads, in program order. For block-level-only scope, the function `__threadfence_block()` can be used instead.

3. GPU DYNAMIC WORK SCHEDULING

The existence of two different levels of memory suggests us to have a series of private work queues (WQ) and work stealing queues (WSQ) in global memory and in shared memory. For work stealing queues we utilize the Blufome and Leiserson's work stealing algorithm [3].

3.1 Hierarchy of structures

We propose a general *queue structure* (q-structure) that consists of two small-sized queues in shared memory, a private work queue, $\mathrm{WQ_{Sh}}$, and a shared work stealing queue, $\mathrm{WSQ_{Sh}}$. We also have the corresponding larger work queue and work stealing queue in global memory which we will represent with $\mathrm{WQ_{Gl}}$ and $\mathrm{WSQ_{Gl}}$, respectively. Additionally, we included in our designs a centralized shared queue whose size needs to be large enough to accommodate the maximum number of tasks. A q-structure may be constructed with any combination of these individual queues.

3.1.1 Notation and Definitions:

For the ith thread T_i, we will represent each of the constituent queues of a q-structure as $q_0^i, q_1^i, q_2^i, \cdots q_L^i$, where q_0^i is the lowest queue, i.e., the first queue we try to access, and q_L^i the highest queue, i.e., the last queue we try to access. We define a potential *victim* of thread T_i as another thread T_j such that $i \neq j$.

3.2 Policies for Using Hierarchy Layers

Given a q-structure configuration, we assign an instance of the q-structure to each one of the threads or processing units that participate in the execution. Assuming initially that the component queues are fully or partly populated with tasks, we define the following policies to manage the different layers for any thread T_i:

1. If a task t taken from queue q_j^i generates a new task t'; we try to put the new task t' in the first queue, q_0^i.
2. If we try to put a task t in queue q_j^i and find that q_j^i is full; we try to put t in queue q_{j+1}^i.
3. If we try to take a task from queue $q_j^i \neq q_L^i$ and find that q_j^i is empty; we try to take a task from queue q_{j+1}^i.
4. If we try to take a task from the last queue q_L^i and find that q_L^i is empty; we have two options:
   (a) we end the execution of thread T_i (static approach)
   (b) we try to steal a task from a victim's q-structure (dynamic approach)
5. If a task t stolen from a victim generates a new task t'; we put the new task t' in the first own local queue, q_0^i.

Once a task is successfully stolen, the task is handled in the same way as if it was taken from the local q-structure. It should be noted that the public centralized queue guarantees that all the generated tasks will find a position in at least one queue.

3.3 Mutual Exclusion Locks

In Figure 1 we show two implementations of a test-and-set lock. Figure 1(a) corresponds to a structure with the `atomicExch` operation while Figure 1(b) shows the implementation with the `atomicCAS` operation. We make use of the memory fence function `__threadfence()` to guarantee that the lock acquisition and critical section manipulation will be visible by all the other threads in the correct order.

3.4 Stealing Modes

3.4.1 Consecutive Order Mode:

The first and simplest mode is to steal tasks from victims in consecutive order: thread T_i tries to steal from thread T_{i+1} and if no task is found then it tries thread T_{i+2}, if not, then thread T_{i+3} and so on.

3.4.2 Random Order Mode:

Our second approach is to select victims randomly. For this, we require a lightweight *pseudo-random number generator* to produce a pseudo-random sequence of numbers that let us to explore all the possible q-structures.

To implement this, we utilize a *linear feedback shift register* (LFSR) which is a shift-register-based counter that is commonly used to produce pseudo-random sequences and that could be easily implemented with a couple of bitwise operations [2], as depicted in the 14-bit LFSR of Figure 2. The maximum number of states given a n-bit LFSR counter is $2^n - 1$.

```
while( atomicExch(&lock, 1) );
__threadfence();
  // critical section
__threadfence();
lock=0;
```

(a) Lock with atomicExch

```
while( atomicCAS(&lock, 0, 1) );
__threadfence();
  // critical section
__threadfence();
lock=0;
```

(b) Lock with atomicCAS

Figure 1: Test-and-set locks utilizing the atomic operations atomicExch and atomicCAS. The memory fences around the critical section guarantee that the memory accesses will be seen in the correct order by all the threads.

```
__device__ uint rng14bits(uint reg){
  uint bit;
  bit = ((reg>>0)^(reg>>1)^(reg>>2)^(reg>>12)) &1;
  return (reg>>1) | (bit<<13);
}
```

Figure 2: 14-bit LFSR implementation.

3.5 Termination Conditions

In an environment without dynamic work scheduling, if a thread or processing unit exhausts its own q-structure, i.e., there are not more tasks to carry out, such thread terminates its execution –this is the case 4(a) in the policies to manage the layers of Section 3.2–, however, the kernel execution terminates only when all the launched threads reach the same condition.

On the other hand, when threads have the capability to steal from other q-structures –case 4(b)–, a maximum number of steals can be attempted by the thread before terminating its execution.

The number of steal attempts plays an important role in the performance of the application. If we employ a small number of steals, we run the risk of terminating the execution prematurely and missing the benefits of work stealing queues. However, if a very large number of useless steals are attempted, they will add unnecesary overheads to execution time.

4. EXPERIMENTAL EVALUATION

All the experiments of this work were done in a NVIDIA GTX 280 GPU running at 1.30 GHz with 1 GB of global memory. The host CPU is an AMD Athlon 64 X2 5200+ Dual Core processor with 8 GB in RAM and running under openSUSE 11.1 linux distribution.

4.1 Evaluation of Overheads

To quantify the cost of each one of the layers of our general q-structure we calculated the number of tasks per second that could be done in each queue individually at different number/size of blocks; the size of a block is the number of threads that it contains.

In Figure 3(a) we show a throughput comparison of work queues and work stealing queues in both global and shared memory. We used 30 blocks and made a scan from 1 to 512 threads per block. Each thread had its own queue and for each task we *took* and then *put* an item in their own queues.

We found that the use of work stealing queues reduces throughput by a factor of 3 in shared memory and in the case of global memory we found an even greater reduction, a factor of 15.

In Figure 3(b) we calculated the throughput of a work queue in shared memory with 30, 60, 90 and 120 blocks of threads. As expected, if we use small-sized blocks, e.g., 128 threads per block or less, we must launch enough blocks if we want to saturate the GPU. With bigger blocks, e.g., 256 or more threads per block, we reach the saturation regardless of the number of blocks as long as we launch enough to "fill" the available multiprocessors, 30 in this GPU.

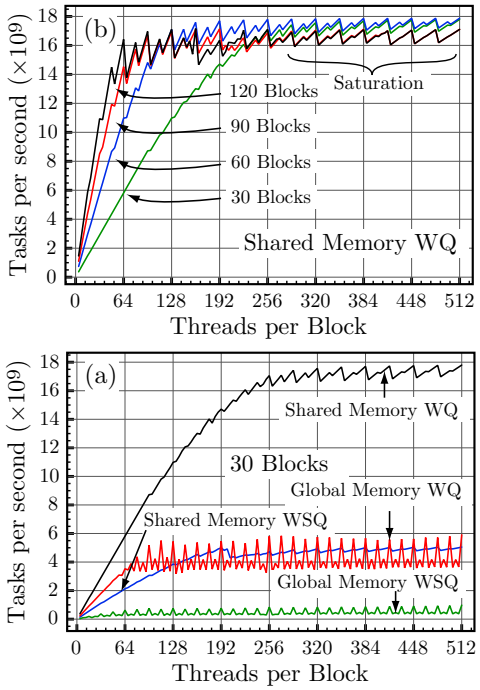


Figure 3: Illustration of the performance of work queues and work stealing queues in shared and global memory.

4.2 Graph Benchmark

We developed a parallel single-source shortest-path benchmark based on the Dijkstra's algorithm [4]. Initially, the N-vertex graph is fully connected with a nonnegative uniform random distribution in path costs and the objective is to find the shortest distance between the source vertex s and all the other vertices.

In our algorithm, each thread or processing unit has its own q-structure with tasks to complete where each task symbolizes a specific vertex that needs to be *explored*. To explore a vertex v means to check if the cumulative cost –coming from the source vertex s– from vertex v to the other vertices is lower or not. If a thread finds a lower cost in, let us say, vertex w, the thread updates the cost of the vertex w, with the lower cost previously found. Finally, if the thread updated the cost, a new task that indicates that vertex w has to be explored, is generated and put in the thread's q-structure. The execution ends when there are no more vertices to explore, i.e., when all the q-structures from all the threads are empty.

Figure 4 shows our algorithm for solving the single-source shortest-path problem. The output is the vector `path` with N elements containing the shortest distance between the source vertex and all the other vertices. The initial cost is stored in the matrix `cost` with size $N \times N$. The function `take()`, takes tasks from the private or shared queues and the function `put(u)`, puts the tasks `u` in the in the per-thread q-structure.

To avoid unnecessary calls to the `atomicMin` function, we first check if the `newpath` cost is lower than the `oldpath` cost, and if so, we atomically check it again and if we find that in fact the cost was lower, we then generate a new task.

Our algorithm allows the same vertex to be put multiple times in different queues and therefore we may end up doing more work than in a sequential implementation. The amount of work per thread is variable in our benchmark and can not be predicted beforehand, which is what we were looking for in our benchmark.

```
while(1){
  v = take();
  if (v != emptyQ){
    for(int u=0;u<N;u++){
      int newpath = path[v] + cost[u + v*N];
      int oldpath = path[u];
      int toPut = oldpath > newpath;
      if(toPut){
        oldpath = atomicMin( &path[u], newpath );
        toPut = oldpath > newpath;
      }
      if (toPut){
        put(u);
      }
    }
  }
  else
    break;
}
```

Figure 4: Parallel single-source shortest-path algorithm. The input consists on a cost matrix and the output is a vector with the shortest distance between the source vertex and all the other vertices.

It is important to note that in the GPU scheme, we assigned one q-structure per warp, so all the threads that constitute

the warp had the same task, i.e., the 32 threads that constitute the warp, explore different neighbors of the same vertex.

4.3 *q*-structure Configurations

Of all the possible configurations of *q*-structures that could be obtained by combining the different layers of Section 3.1, we present results for two that show the highest performance for static and dynamic scheduling, as they allow us to compare and evaluate the performance of the effectiveness of the load balancing techniques. We utilize, as a baseline, a static private *q*-structure which we will represent with *private*Q. For the dynamic case we utilize a partially shared potentially stealable *stealable*Q. Their configurations are the following:

- *private*Q: A private WQ in shared memory (WQ_{Sh}) and a large private WQ in global memory (WQ_{Gl}).
- *stealable*Q: A private WQ in shared memory (WQ_{Sh}) and a large shared WSQ in global memory (WSQ_{Gl}).

where in both cases, we set the queue in global memory to be large enough to prevent the loss of tasks.

The private WQ_{Sh} in shared memory in both *q*-structures are used to reduce the number of accesses to global memory. However, due to memory size restrictions in the shared memory, we limited our *q*-structures to have only one queue in shared memory. A private WQ_{Sh} was chosen against a shared WSQ_{Sh} since the latter has a reduced scope to steal tasks and it is slower than the private WQ_{Sh}.

Figure 5 shows one of the functions used to manipulate the tasks of the *q*-structures. In this case a function to *take* tasks from a shared WSQ in global memory is shown. The additional functions such as *take*/*put* in global and shared memory are straightforward modifications of Figure 5. The `ACQ_LOCK()` and `REL_LOCK()` macros are used to acquire and release the lock, respectively, and are only used in shared WSQ structures.

```
__device__ int takeFromGmemWSQ(volatile int *gWSQ,
                               int id, int qSize,
                               int totNumOfQueues ){
  int currentTask = EMPTY_QUEUE;
  ACQ_LOCK(gWSQ);
  int tail = gWSQ[qSize*totNumOfQueues + id];
  if( tail>0 ){
    tail--;
    currentTask = gWSQ[tail*totNumOfQueues + id];
    gWSQ[qSize*totNumOfQueues + id] = tail;
  }
  REL_LOCK(gWSQ);
  return currentTask;
}
```

Figure 5: Function to take tasks from a WSQ located in global memory.

4.4 Experimental results

To evaluate the performance of the dynamic work scheduling framework utilizing our benchmark, we measured the execution time of each one of the threads, which is equivalent to measuring the execution time of the warps since threads within a warp finish at the same time. Then, we compared the results of the *private*Q and the *stealable*Q. Additionally, for statistical purposes, we included a series of counters to monitor the state of the queues: the number of times they get empty/full and the number of task stolen among others.

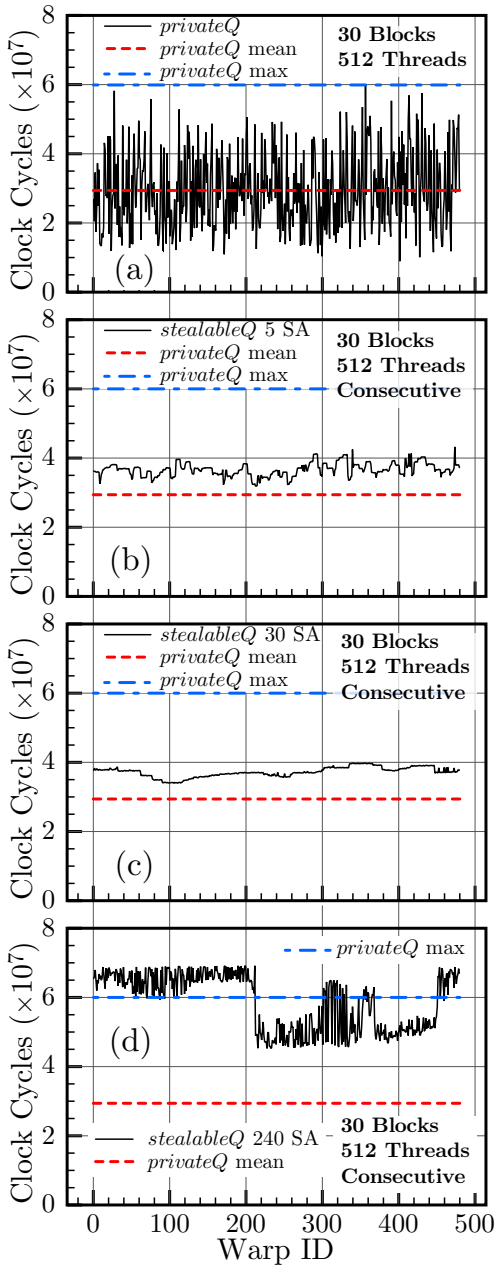


Figure 6: Load balancing experiments using the consecutive stealing mode. The baseline *private*Q performance is shown in (a). The *stealable*Q performance is shown in (b), (c) and (d) with 5, 30 and 240 SA, respectively. We included the *private*Q average time as a reference. The execution time is very sensitive to the number of steal attempts and the optimal point occurs around 30 SA.

4.4.1 Consecutive order:

In Figure 6 we show four plots with execution times per warp of the *private*Q and *stealable*Q using the consecutive order steal method. The experiment was run using 30 blocks with 512 threads each which gives a total of 480 warps. Plot (a) shows our baseline *private*Q experiment while plots (b), (c) and (d) are the experiments with the *stealable*Q utilizing 5, 30 and 240 *steal attempts* (SA) respectively.

We can see from Figure 6(a) that the *private*Q experiment is clearly unbalanced. As a reference we included an horizontal line to indicate the execution time average, we would expect to obtain this line if the work were perfectly balanced. We also included a line to indicate the maximum, i.e., when all warps have finished their execution. In this experiment we found that there are warps that take 3 to 5 times more time to finish than others, so there are warps that become idle quickly and waste resources while there are others with more work to do.

This contrast with the curves obtained when using the *stealable*Q structure where the big variation in the execution time is reduced. However, we can also see that the number of steal attempts has an impact on the shape of the curves. In Figure 6(b) with 5 SA, the overall execution time is lower but we still had warps spending more time that others. The best scenario occurs in Figure 6(c) with 30 SA, we obtained a curve that is practically flat which gives speedups up to 1.5x compared to the *private*Q structure.

In the last Figure, with of 240 SA, we obtained two sections with a width of approximately 240 warps each and whose variations are smaller compared to the *private*Q, however, the overall execution time is worse.

4.4.2 Random order:

In this second approach, we similarly executed several experiments with different number of steal attempts. In Figure 7 we show the baseline *private*Q experiment and the *stealable*Q experiments with 5, 30 and 240 SA.

Compared with the consecutive order method, we note that the warps are uniformly balanced, the random selection of victims helps to distribute tasks evenly, which improves the execution time. We also note that the execution time is more stable at different numbers of steal attempts, it remains flat with 30 and 240 SA. Measurements with 480 attempts (not shown) show a similar behavior as that of Figures 7(c) and 7(d).

4.4.3 Number of steal attempts:

Figure 8 shows the impact of number of steal attempts on execution-time at various parameters. Figure 8(a) is a comparison between the consecutive victim selection mode and three different-sized LFSR random number generators with 8-, 11- and 14-bit sequences, respectively. We note an improvement in the execution time with larger random sequences and in general a better performance compared to the consecutive mode.

In Figures 8(b) and 8(c), we show a comparison utilizing 30 blocks with 128, 256 and 512 threads each and selecting victims with a 14-bit random number generator in (b) and in consecutive order in (c). In both cases the best performance is achieved when the GPU is saturated with 512 threads per block.

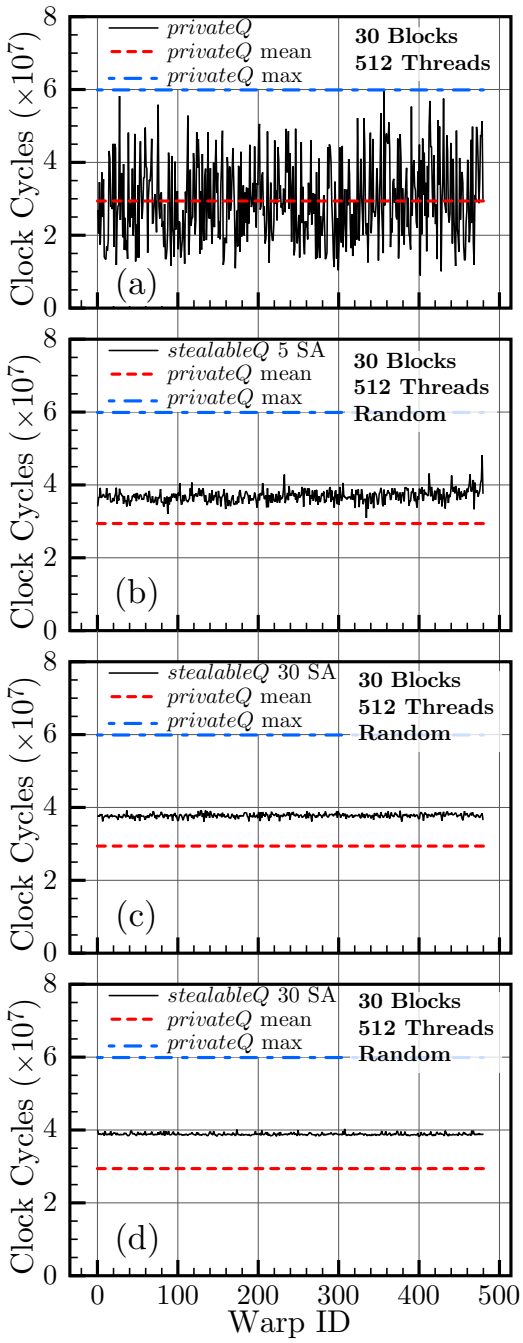


Figure 7: Load balance experiment utilizing the random stealing mode with a 14-bit LFSR random number generator. (a) is the baseline *private*Q and (b), (c) and (d) are the *stealable*Q experiments with 5, 30 and 240 SA respectively. The execution time is more stable with the number of steal attempts and tasks are better distributed.

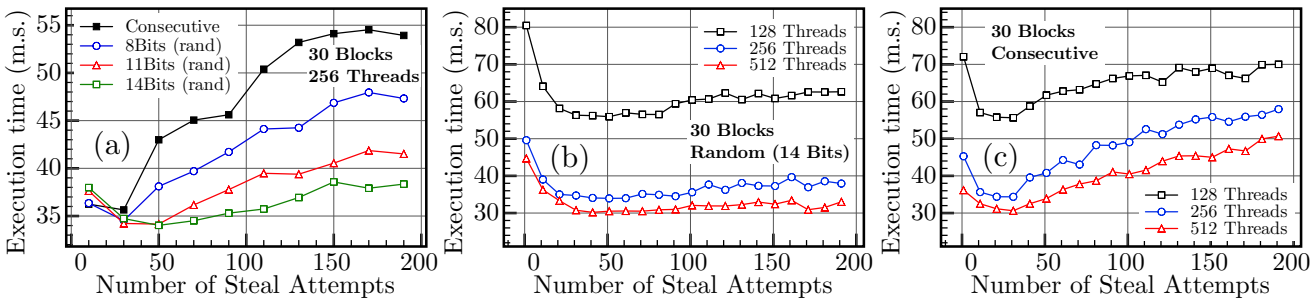


Figure 8: Effect of the number of steal attempts in the execution time of the benchmark. In (a) a comparison is made between the consecutive mode and several random modes. In (b) and (c) are shown the execution times at different number of threads per block utilizing the random and consecutive mode, respectively.

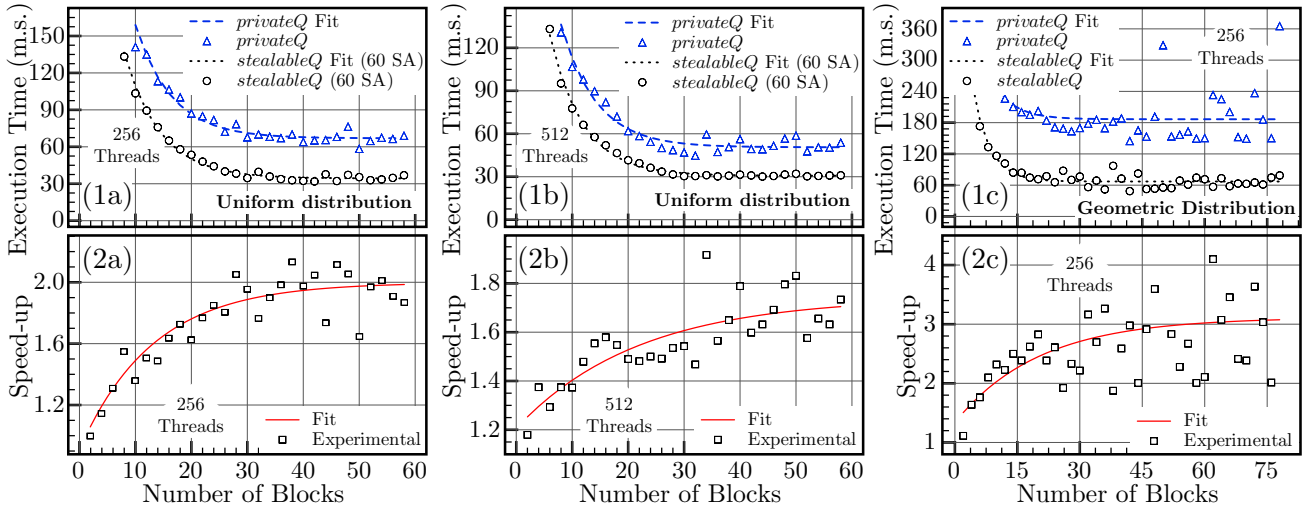


Figure 9: Execution times and speed-ups at different number of blocks. Utilizing a uniform random distribution in costs with 256 threads in (a) and 512 threads in (b). In (c) a geometric distribution was used instead. In all cases 60 steal attempts were used.

4.4.4 *Number of blocks:*

The number of blocks also have an impact on the execution time. We show in Figure 9 that at least 30 blocks are needed to saturate the GPU, this agrees with the number of available multiprocessors. Figure 9(1a) compares the execution time of our baseline, the *private*Q, and the *stealable*Q at different number of blocks with 256 threads each. Figure 9(2a) shows the speed-up achieved, that goes up to 2×. Figure 9(1b) and 9(2b) makes the same comparison but with 512 threads per block. In this case we observed a lower speed-up of ≈ 1.7×.

All the previous experiments were calculated with a benchmark utilizing a uniform random distribution in the cost of the vertex-paths. In Figure 9(1c) and 9(2c), to study the behavior of our dynamic work scheduling framework in a different distribution, we utilized a geometric distribution and found a speed-up of up to 3× with blocks of 256 threads each. We also noted that the results from the dynamic scheduling approach are more predictable and stable compared to the results from the static approach, as shown in Figure 9(1c).

5. CONCLUSIONS

In this paper we presented a framework for efficient dynamic work scheduling in graphics processing units. Our work addressed a limitation of current GPUs where work is assigned statically to available multiprocessors. We built a benchmark that solves the single-source shortest-path problem to expose such limitation and found that our dynamic work scheduling framework outperforms the static scheduler by a factor of up to 2× when using graphs with uniform random distribution in cost and by a factor of up to 3× with a geometric distribution.

We demonstrated that the random selection of victims generates a better distribution of the tasks among threads compared to the consecutive selection and that it is more stable at different number of steal attempts. Our experimental results show that the length of the random sequence also affect the effective selection of tasks, longer sequences generates a better random distribution and therefore a better load balance is achieved.

References

[1] Cuda programming guide 2.3. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf, 2009.

[2] P. H. Bardell, W. H. McAnney, and J. Savir. *Built-In Test for VLSI: Pseudorandom Techniques*. John Wiley & Sons, New York, 1987.

[3] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[4] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[5] T. Foley and J. Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, page 22, 2002.

[6] W. Fung, I. Sham, G. Yuan, and T. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420. *IEEE Computer Society*, 2007.

[7] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, pages 39–55, 2008.

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

[9] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[10] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82. ACM, 2008.

Author Index