Thesis presented to the Faculty of the Department of Graduate Studies of the Aeronautics Institute of Technology, in partial fulfillment of the requirements for the Degree of Master in Science in the Course of Electronic Engineering and Computer Science, Field Computer Science

Daniel Lélis Baggio

GPGPU BASED IMAGE SEGMENTATION LIVEWIRE ALGORITHM IMPLEMENTATION

Thesis approved in its final version by signatories below:

Prof. Dr. Jackson Paul Matsuura
Advisor

Prof. Dr. Homero Santiago Maciel
Head of the Faculty of the Department of Graduate Studies

Campo Montenegro
São José dos Campos, SP - Brazil
2007

Cataloging-in Publication Data

Division of Central Library of ITA/CTA

Baggio, Daniel Lélis

GPGPU Based Image Segmentation Livewire Algorithm Implementation / Daniel Lélis Baggio. São José dos Campos, 2007.

108f.

Thesis of Master in Science – Course of Electronic Engineering and Computer Science. Area of Computer Science – Aeronautical Institute of Technology, 2007. Advisor: Prof. Dr. Jackson Paul Matsuura.

1. GPGPU. 2. Segmentation. 3. Livewire. I. Aerospace Technical Center. Aeronautics Institute of Technology. Division of Computer Science. II. Title.

BIBLIOGRAPHIC REFERENCE

BAGGIO, Daniel Lélis. **GPGPU Based Image Segmentation Livewire Algorithm Implementation**. 2007. 108f. Thesis of Master in Science – Technological Institute of Aeronautics, São José dos Campos.

CESSION OF RIGHTS

Daniel Lélis Baggio

Rua Lima Machado, 239

CEP 11.310-310 – São Vicente–SP

GPGPU BASED IMAGE SEGMENTATION LIVEWIRE ALGORITHM IMPLEMENTATION

Daniel Lélis Baggio

Thesis Committee Composition:

Prof. Dr.	José Maria Parente de Oliveira	Chairperson	-	ITA
Prof. Dr.	Jackson Paul Matsuura	Advisor	-	ITA
Prof. Dr.	Celso Massaki Hirata		-	ITA
Prof. Dr.	Nei Yoshihiro Soma		-	ITA
Prof. Dr.	Sérgio Shiguemi Furuie		-	InCor

ITA

Resumo

Esta tese apresenta a implementação do algoritmo de segmentação de imagens Livewire em uma placa de vídeo, que é uma arquitetura *Single Instruction Multiple Data*(SIMD), ao invés da utilização tradicional da CPU. O algoritmo é dividido em três fases: aplicação do filtro *Sobel* ou *Laplaciano* sobre a imagem, seguido de modelagem da mesma através de grafos do tipo *grid* e posterior resolução do menor caminho a partir de um dado nó. Para tal cálculo uma abordagem paralela é feita através do desenvolvimento de uma versão adaptada do algoritmo Δ-stepping para placas de vídeo. Cada uma das partes do algoritmo foi transformada em um núcleo que é executado e compilado na própria placa de vídeo, utilizando-se a arquitetura *CUDA (Compute Unified Device Architecture)* disponível na série 8 da *NVidia GeForce*. As placas de vídeo são os primeiros dispositivos SIMD amplamente disponíveis em diversos computadores. Embora originalmente desenvolvidos para aplicações de renderização, pesquisadores da área de GPGPU (*General Purpose Computing on Graphic Processing Units*) têm demonstrado que o imenso poder computacional destes dispositivos e a sua recente capacidade de programação fazem deles uma alternativa atrativa como plataforma de alta-performance. A implementação foi focada na arquitetura CUDA, mas diversas outras abordagens são comentadas e referenciadas mostrando grande parte das alternativas disponíveis, como outras plataformas – CPUs com multicore, processador *Cell* –, outras APIs gráficas como Cg e OpenGL, ou mesmo

abordagens que deixam transparente o uso de GPUs, como RapidMind e Brook. A conclusão coloca em evidência o sucesso da implementação do algoritmo para a plataforma de GPUs ressaltando os aspectos positivos e negativos da abordagem utilizada. Uma análise crítica dos resultados demonstra que o processamento de imagens através de filtros tem grande ganho de desempenho com relação a uma CPU, no entanto, devido a muitos acessos à memória do dispositivo, o algoritmo Δ-stepping não mostrou performance superior em tal arquitetura com os tamanhos de grafos testados, apontando uma maior melhora quanto maior o tamanho do grafo. Uma menor demora no acesso à memória local ou mesmo um maior número de threads poderiam aumentar muito a performance do algoritmo. Além da demonstração de viabilidade de implementação do algoritmo, esta tese contribui disponibilizando uma aplicação *open-source* de segmentação de imagens através da GPU (em http://code.google.com/p/gpuwire/), servindo como base para futuras implementações na mesma arquitetura.

Abstract

This thesis presents a GPU implementation of the Livewire *algorithm. Instead of using traditional architectures, like the CPU, this implementation focuses advantages obtained using* Single Instruction Multiple Data *(SIMD) architectures. The algorithm is divided in three phases:* Sobel *or* Laplacian *filter convolution, image modeling as a grid graph and solving the non-negative weighted edges single source shortest path problem. In order to calculate the shortest path, a parallel approach is made through the development of an adapted version of the* Δ*-stepping algorithm for GPUs. Each part of the algorithm was programmed as a single kernel, which is executed and compiled to the GPU, using CUDA (*Compute Unified Device Architecture*), available on NVidia GeForce 8 series. GPUs have been the first SIMD commodity devices widely available on several desktops. Although originally designed for applications highly focused on rendering, GPGPU (*General Purpose Computing on Graphic Processing Units*) researchers have shown that the huge processing power available on these devices as well as the recent advent of a programmable pipeline have made of GPUs an attractive option for low cost high performance platforms. Even though the implementation has used CUDA API, several other approaches are pointed out, showing a wide variety of alternatives such as other platforms – multicore CPUs, Cell processor –, other graphic APIs, such as Cg, and OpenGL, or even different approaches like RapidMind and Brook, which make GPU access transparent. The conclu-*

sion of this thesis highlights a successful implementation of the algorithm using the GPU architecture showing advantages and disadvantages of this approach. An in-depth result analysis shows that intense speedups are seen in image filtering algorithms. On the other hand, the wide use of dependent device memory look-ups has constrained Δ*-stepping algorithm from achieving higher performance than CPU implementation although a better performance is expected for wider graphs. If device memory access latency was decreased or if more threads were available, a huge increase in performance would be expected. Besides showing the viability of the* Livewire *algorithm implementation, this thesis makes available an* open-source *image segmentation GPU based application, which can be used as example for future GPU algorithm implementations at http://code.google.com/p/gpuwire/.*

Contents

List of Figures

List of Tables

List of Symbols

ACML	AMD Core Math Library
API	Application Programming Interface
BRT	Brook Runtime
CAL	Compute Abstraction Layer
CMP	Chip-level Multiprocessing
CPU	Central Processing Unit
CTM	Close To Metal
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
EIB	Element Interconnect Bus
GLSL	OpenGL Shading Language
GPGPU	General Purpose programming for Graphic Processing Units
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
MCSTL	Multi-Core Standard Template Library
MFC	Memory Flow Controller

MIMD	Multiple Instruction, Multiple Data
NSSP	Non-weighted Single-source Shortest Path Problem
PS3	Playstation 3
PPC	PowerPC
PRAM	Parallel Random Access Machine
RAM	Random Access Machine
SDK	Software Development Kit
SIMD	Single Instruction, Multiple Data
SMP	Symmetric Multiprocessing
SPE	Synergistic Processing Elements
STI	Sony Toshiba IBM
STAPL	Standard Template Adaptive Parallel Library
SWAR	SIMD Within A Register
TBB	Threading Building Blocks

1 Introduction

1.1 Motivation

Concurrency is nowadays the new trend in writing software. Over the last 30 years (SUTTER, 2005), CPU chip designers have achieved performance gains focusing clock speed, execution optimization, and cache. Execution flow has been optimized through pipelining, branch prediction, multiple instruction execution and instruction reordering, while new memory layouts have tried to bring memory closer to CPU by putting it in the same chip, through on-die memory such as level 1 and level 2 caches. But the increase in performance through clock speed has been reaching several physic barriers such as heat, power consumption and current leakage problems. In order to overcome these limits, more processing power is going to be retrieved from new architectures that gather multi cores inside chips, which requires higher levels of parallelism in software packages. While programmers have been able to increase their programs performance by increasing CPU frequency until now, future software will need to be rewritten in order to take advantage of these new architectures.

The easiest way for a programmer to exploit the power of concurrency in new CPUs is probably by using threads in a double or quad core processor, such as Intel's or AMD's. Besides using the CPU itself, another common hardware that is able to do parallel processing is the Graphic Processing Unit (GPU) and a new processor from Sony, Toshiba and IBM, known as the Cell processor.

On the other hand, in the field of image segmentation, such an important algorithm for medical image analysis is the *LiveWire* (FALCÃO, 1997). This algorithm is divided in two main steps: image filtering, and finding the shortest path in a graph. Since serial algorithms for solving the single source shortest path problem can be very slow for several

instances of this problem – given that its running time belongs to $O(V \lg V + E)$ (CORMEN *et al.*, 2001), where V is the number of vertices and E is the number of edges in the graph – it would be interesting if a parallel solution could be created exploring this algorithm and new high-performance architectures.

1.2 Objective

Given that such a huge amount of processing power is available on low cost GPUs, as explained in Section 2.1, the main objective of this thesis is to investigate if a parallel GPU implementation of the *LiveWire* image segmentation algorithm (FALCÃO, 1997) is feasible and how it performs compared to traditional serial CPU implementation. The major challenge to accomplish this goal is the requirement of parallel algorithms that outperform serial versions and are able to support a GPU implementation. Besides that, studying the best way to implement such algorithm in the GPU platform is not trivial, since a handful of alternatives are available.

1.3 Thesis Outline

As this thesis intends to study how a parallel implementation of the *LiveWire* algorithm performs in GPUs, the second chapter gives background information about parallel architectures as well as the advent of General Purpose Programming for Graphic Processing Units (GPGPU). Besides that, this chapter gives important information on how to program for GPUs through the study of APIs, platforms, and languages available, such as *CUDA*, *GPUCV*, *MATLAB*, *CTM*, *Brook*, *Sh*, *RapidMind*, *Microsoft Accelerator*, and *Shaders*. More information about GPU Programming Model, and GPU Hardware Model are given as well. Another important background information given in this chapter is the description of the *LiveWire* algorithm, and how a parallel solution for finding the single source shortest path can be made through the Δ-*stepping* algorithm.

The GPU implementation of *LiveWire* algorithm is described in chapter 3. This description shows details of parallel filtering, and CUDA technology. It also shows optimizations applied and the development of a Graphical User Interface (GUI). Chapter 4 is

reserved for the results and analysis, while conclusions and future research are described in chapter 5. The appendixes provide implemented code for reusability.

2 Background

Developing applications for GPUs requires several paradigm shifts. One of them is changing the traditional CPU architecture to a many-core one. As several multi-core architectures have been developed and studied, this chapter gives some background information about the way these architectures deal with memory and managing multiple processors, so that key elements can be identified. Besides this paradigm shift, programming for GPUs requires learning different languages, APIs, and platforms. This chapter shows the most important work developed, such as CUDA, Brook, RapidMind, and other solutions. GPU programming model as well as GPU hardware model are also described. As a GPU implementation of the *LiveWire* algorithm is developed in this thesis, the end of the chapter is dedicated to explain main details of this algorithm. This chapter starts with the description of the General Purpose Programming using Graphic Processing Units (GPGPU) advent.

2.1 General Purpose Programming Using Graphic Processing Units

Recent graphic architectures are probably today's most powerful computational hardware for the dollar (OWENS *et al.*, 2007). For example, NVidia's GeForce 8800 GTS features 64 GB/s memory bandwidth (NVIDIA, 2007c) and it costs $409 (as of October 2007). Besides having high bandwidth, GPUs have been getting faster quickly. GeForce 7800 GTX(165 GFLOPS) more than triples its predecessor GeForce 6800 Ultra(53 GFLOPS)(OWENS *et al.*, 2007). It's clear from Figure 2.1 that GPUs growth has outpaced Moore's Law, which states that the number of transistors that can be inexpensively placed on an integrated circuit is increasing exponentially, doubling approximately

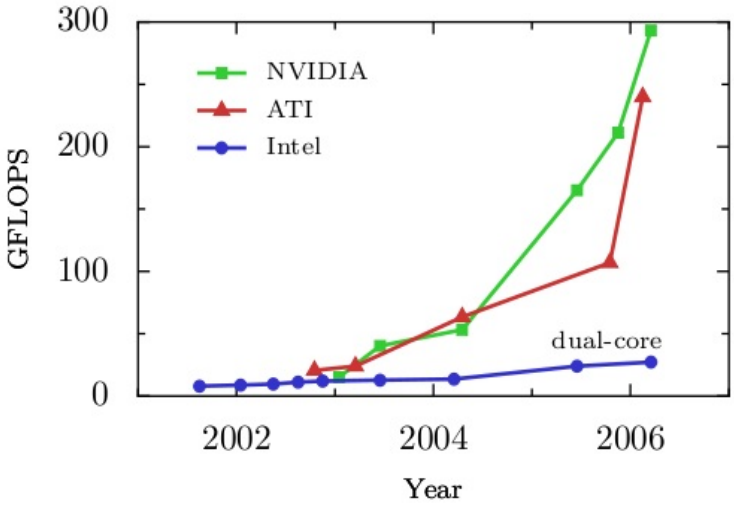


FIGURE 2.1 – GPU GFLOPS growth measured on the multiply-add instruction counting 2 floating-point operations per MAD (OWENS *et al.*, 2007)

every two years. While the average yearly rate of graphic performance has been 1.7 (pixels/second) and 2.3 (vertexes/second), CPU's performance has roughly increased 1.4 times per year in average (EKMAN; WARG; NILSSON, 2005).

Still considering Figure 2.1, semiconductor capability and advances in fabrication process have increased for both platforms at the same rate, but the main growth disparity is due to architectural differences: CPUs have large caches and are optimized for high performance on sequential code, focusing branch prediction and out-of-order execution. On the other hand, GPUs that focus on highly parallel graphic computations achieve higher arithmetic intensity by using roughly the same number of transistors.

Main difficulties and limitations found in GPU programming have been:

- Single precision floats – most of the architectures currently only make available single precision floats, although double-precision can be emulated with some performance penalties;
- Bitwise operations – some GPU models do not support bitwise operations;
- Memory scatter – only new models have support for memory scattering operations;
- Graphic API programming – in order to achieve vendor independence, graphic APIs such as OpenGL or DirectX are usually recommended, although CUDA and CTM make it possible for programmers not to use graphic APIs;

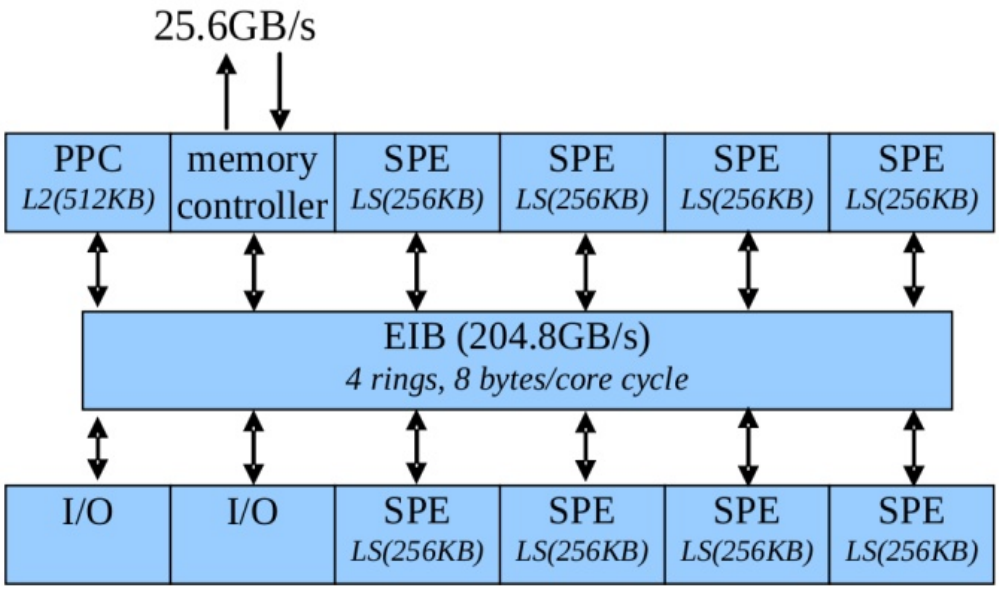


FIGURE 2.2 – Cell Broadband Engine Processor overview

- New programming paradigm – GPU programming is different from simply using a new API. Programs and algorithms need to go through a completely different design process in order to exploit parallel advantages of the architecture.

2.2 Parallel Processor Architectures

2.2.1 Graphic Processors

One of the architectures that has received more focus during the latest years has been GPUs. NVidia(NVIDIA, 2007d) and ATI(AMD, 2006) have been the main GPU manufacturers with a long list of different models and features. Some of the latest technologies make it possible to combine multiple GPUs in a single machine, increasing several levels of parallel processing.

2.2.2 IBM Cell Processor

Cell, which is a shorthand for Cell Broadband Engine Architecture (Cell BE), is a microprocessor jointly designed by Sony, Toshiba and IBM (STI) as the core of PlayStation3 gaming system. The great innovation of Cell is to combine a high performance general

purpose 64-bit POWER Processing Element (PPE) with eight Single Instruction, Multiple Data (SIMD) cores – the Synergistic Processing Elements (SPEs) – instead of using identical cooperating commodity processors. Each SPE has a dedicated local storage of 256KB and a Memory Flow Controller (MFC) (WILLIAMS *et al.*, 2006). A high level view of Cell BE first implementation is seen in Figure 2.2. It must be noted that memory bandwidth with off-chip memory is 25.6 GB/s, while Local Storage (LS) and L2 Cache reach 51.2 GB/s.

The PPE is the core processor of the Cell BE and consists of a POWER Processing Unit connected to a 512KB L2 cache. It is a dual-issue, in-order processor with dual-thread support, designed to maximize the performance/area ratio as well as the performance/power ratio. The general purpose of this processor is to run the operating system and to coordinate the SPEs. It selects which one is supposed to run which task, as well as equally sharing the data for each thread (CHEN *et al.*, 2007).

SPE consists of a Synergistic Processing Unit (SPU) and a Memory Flow Controller (MFC). It also has SIMD support and 256KB of dedicated local storage. Since it is a dual-issue, in-order machine, with an 128 entry, 128-bit register file, it can increase processing power in a wide range of applications. The 128-bit registers can process either integers or floating-point numbers, which must be in its dedicated local store, as well as the instructions it must perform. While SPUs process data, the MFC can be independently accessed and translate addresses for DMA transfers – a technique called double buffering. Since the 128-bit registers can store four 32-bit single-precision floating-point numbers, each SPU is capable to perform 25.6 GFLOPs, at 3.2 Ghz (default Cell BE frequency). Eight SPEs can be accessed in parallel (while they are limited to 6 in Playstation 3(BUTTARI *et al.*, 2007)), this adds up a total of 204.8 GFLOPs for single-precision floating-point numbers (CHEN *et al.*, 2007).

The Element Interconnect Bus (EIB) in the Cell is a high speed bus used for communication among the PPE, the SPEs, off-chip memory and the external I/O. It consists of 16B-wide data rings and one address bus operating at half the speed of the processor. Each unit of the EIB can simultaneously send and receive 16B of data every bus cycle and its theoretical peak data bandwidth, at 3.2GHz is 204.8GB/s, what makes of this bus an excellent option for data streaming(CHEN *et al.*, 2007).

2.2.3 Multi-core CPUs

Most of the multi-core CPUs rely on Symmetric Multiprocessing (SMP) (JONES, 2007), which is an architecture whose processors are identical and connect to one another through a shared memory. Earliest SMP systems were multiple standalone computers connected by a high-speed interconnect, such as Ethernet, Fibre Channel, or Infiniband. As too much space and power were required to build a loosely-coupled multiprocessor architecture, the tightly-coupled architecture became more accepted. The latter is a chip-level multiprocessing (CMP) in which multiple chips, shared memory and an interconnect are in a single integrated circuit. In multi-core architectures, each processor has its own fast memory in a level 1 cache, while a shared memory connects multiple CPUs through level 2 cache.

Although several operating system kernels support multicore CPUs natively – such as Linux 2.6 kernels or Windows – some template libraries have been developed to fully exploit these architectures. One of them, from Intel, *Threading Building Blocks*, makes available a wide range of structures and algorithms such as *parallel_while, parallel_sort, concurrent_queue, concurrent_vector* as well as mutual exclusion operators and atomic operations.

Intel's *Threading Building Blocks (TBB)* (INTEL, 2008) is a runtime-based parallel programming model for C++ code that uses threads. In order to compile and link programs that use *TBB*, header files and libraries can be downloaded from *http://threadingbuild ingblocks.org*. An open source edition is available. Using the library is as straightforward as including header files and linking against the built shared libraries. The tutorial available (INTEL, 2007) explains how to use the following building blocks:

- Parallel Simple Loops: *parallel_for, parallel_reduce*;
- Parallel Complex Loops: *parallel_do, pipeline*;
- Containers: *concurrent_hash_map, concurrent_vector,concurrent_queue*;
- Mutual Exclusion: *mutex, recursive_mutex, spin_mutex, queuing_mutex, spin_rw_mutex,queuing_rw_mutex*;
- Atomic Operations: *fetch_and_store, fetch_and_add, compare_and_swap*;

- Timing: *tick_count* class as a simple interface available for measuring wall clock time;
- Memory Allocation: *scalable_allocator* and *cache_aligned_allocator*, two memory allocator templates available.

Intel's approach makes use of different numbers of multi-core processors by querying the system, without any needs of recompilation.

A related work, from Texas A&M University is the Standard Template Adaptive Parallel Library (STAPL) (AN *et al.*, 2003). It is an adaptive, generic parallel C++ library that is designed to be a superset of the ANSI C++ Standard Template Library(STL). In order to be adaptive STAPL provides several different algorithms for some library routines, and selects among them adaptively at run-time. A preprocessing phase can be used to automatically replace STL for STAPL, making it easier to use STAPL in existing projects. While STL consists of three major components – *containers, algorithms, and iterators* – STAPL's interface layer consists of five major ones: *pContainers, pAlgorithms, pRanges, schedulers/distributors, and executors.* The *pContainers* and *pAlgorithms* are parallel versions of STL containers and algorithms. The *pRange* allows random access to a partition or subrange of elements in a *pContainer* as well as uses a *distributor* for data distribution and a *scheduler* that can increase parallel execution through data dependence graphs (DDGs). *Executors* can execute subranges of *pRange* based on the execution schedule.

Another programming approach is MCSTL: The Multi-Core Standard Template Library(PUTZE; SANDERS; SINGLER, 2007), whose main idea is to provide multi-core implementations of STL algorithms that can obtain speedup from a small number of cores such as two, four, eight or sixteen. MCSTL is intended to take advantage of multi-core architectures with minimal changes in programs. Developers who have already been using STL in their programs must only add the include directive

```
#include <mcstl.h>
```

and link against OpenMP – the Open Multi-Processing API – through a simple *-fopenmp* flag. Implemented functions for MCSTL comprise the following algorithm classes:

- Embarrassingly Parallel: functions that process n independent atomic jobs in parallel. STL function calls: for_each, $generate$, $fill$, $count(\_if)$, $transform$, $unique\_copy$, $min/max\ element$, $replace$;

- Find: functions that find the first element in a sequence which satisfies a certain predicate. Implemented functions: $find$, $mismatch$, $equal$, $adjacent\_find$, $lexicographical\_compare$, $search$;

- Partial Sum: functions that compute and synchronize partial sums over sequences. Implemented functions: $accumulate$, $partial\_sum$, $inner\_product$, $adjacent\_diff$;

- Sorting and its Kindred: sorting, merging and partitioning functions. Implemented STL functions: $partition$,$merge$, $multiway\_merge$, $inplace\_merge$, $nth\_element$, $partial\_sort$,$sort$,$stable\_sort$;

- Random Shuffle: cache-efficient random permutation algorithm. Implemented function: $random\_shuffle$.

Experimental results from MCSTL (PUTZE; SANDERS; SINGLER, 2007) show that most STL algorithms can be efficiently parallelized on multi-core processors. Because of such results MCSTL has still been under development, so that more functions can be implemented.

2.2.4 Misc architectures

A different architecture has been built by *Cray Inc.* called MTA-2. It is a shared memory MIMD computer with an unusual design based on the Tera computer (ALVERSON *et al.*, 1990). The latter computer architecture was designed to be suitable for very high speed implementations, *i.e.*, high frequency, and it aimed to have 256 processors, 512 memory units, 256 I/O cache units, 256 I/O processors, and 4096 interconnection network nodes. It focused on streams, so that each processor could execute multiple streams simultaneously. Each processor could have from one up to 128 program counters active at once. This amount of streams executing in different clock ticks was used to hide the expected instruction latency.

In a study about porting scientific programs to MTA-2 (ANDERSON *et al.*, 2003), researchers describe the four state bits associated with each memory word: a forwarding bit, a full-empty bit, and two data-trap bits. Besides these bits, the architecture also provides an atomic *fetch-and-add* operation, which is available to programmers through intrinsics.

Another study (MADDURI *et al.*, 2006) relies on the ability to exploit fine-grained parallelism and low-overhead synchronization primitives to implement a parallel algorithm for the single source shortest path problem with non-negative edge weights, successfully achieving a relative speedup of close to 30 by using 40 processors.

2.3 Programming Graphics Hardware

As graphic processing units have become more powerful, each generation of graphics hardware has focused on more realism of rendered images. While offline rendering such as Pixar's RenderMan (UPSTILL, 1990) demonstrated the benefits of a more flexible pipeline, particularly in the areas of lighting and shading, GPUs fixed-function pipeline has become obsolete. The efforts to enhance fixed-function pipeline have been primarily concentrated on two stages: the vertex stage and the fragment stage. A user-defined *vertex program* and a *fragment program* could be sent to the GPU programmable pipeline (OWENS *et al.*, 2007) so that vertice transformations, lighting calculations, and fragment colors could be programmatically defined. In order to exploit the new programmable pipeline, several approaches have been developed. The most important ones are described in this section. Before introducing these approaches, a better explanation of the GPU Programming model is given.

2.3.1 GPU Programming model

In order to fully exploit GPU potential, a different programming paradigm must be used, which is stream programming (OWENS *et al.*, 2007). Data is packed into streams and computations are arithmetic kernels that operate on streams. The stream programming model is discussed by Owens (OWENS, 2005) and Lefohn et al. (LEFOHN; KNISS; OWENS, 2005). The fragment shader is the programmable stage with the highest arith-

metic rates, this way, a typical GPGPU program uses this processor as the computation engine. GPGPU programs are structured in the following manner (HARRIS, 2005):

1. The data-parallel sections of the program are identified and implemented as a *kernel*, which is a fragment program running on the fragment shader. The input and output of the kernel are data arrays stored as textures in GPU memory which are also referred as *streams*. The kernels are invoked in parallel on each *stream* element.

2. In order to invoke a kernel, the range of the computation is passed to the GPU through a polygon draw function call. A quadrilateral that is oriented parallel to the image plane is typically chosen, so that its size cover a rectangular region of pixels matching the desired output array size.

3. When the image is rasterized, a fragment is produced for every pixel location in the quad, producing thousands to millions of fragments.

4. The active kernel fragment program processes each of the generated fragment. It must be noted that every fragment is processed by the same fragment program. Although the fragment program can read from several locations it can only write to memory locations corresponding to the location of the fragment in the frame buffer.

5. The output of the fragment program is a vector of values per fragment, which may be used again for additional computations. Some complex applications may require several passes through the pipeline.

After describing the GPU programming model, one can notice that some programs may fit the model perfectly and might as well benefit from increased performance. On the other hand, there are programs which might not fit the aforementioned programming model and may not present speedup running on the GPU.

2.3.2 CUDA

Since using graphic libraries API is not a trivial task for general programmers and this part of the process can be wrapped up in libraries, some alternatives have been proposed. An attempt to make GPGPU programming more straightforward was made by NVidia, in

a technology known as CUDA (Compute Unified Device Architecture) (NVIDIA, 2007d). This technology allows programmers to use an extension of the C language for a minimum learning curve and it also provides on-chip shared memory with faster access than DRAM, making applications less dependent on DRAM memory bandwidth. CUDA is compatible with NVidia's 8-Series, Quadro FX 5600/4600, and Tesla solutions, while its SDK was made available in February 15th, 2007.

CUDA's software stack is composed by a hardware driver, an application programming interface (API) and its runtime, and two higher-level mathematical libraries, CUFFT (NVIDIA, 2007b) – CUDA Fast Fourier Transform library – and CUBLAS (NVIDIA, 2007a) – an implementation of Basic Linear Algebra Subprograms (BLAS) on top of CUDA. One of the major features from CUDA is the fact that it provides general DRAM memory access. This way, both scatter and gather memory operations are available. In order to achieve fast thread communication, CUDA features a parallel data cache or on-chip shared memory with very fast read and write support. Unfortunately, the size of shared memory can be too small for certain applications.

2.3.3 GPUCV

An important library focused in computer vision and image processing ported to GPUs is GPUCV (FARRUGIA *et al.*, 2006). It is an open source library which uses OpenGL and GLSL (as explained in Section 2.3.10), so, any system supporting OpenGL 2.0 also supports GPUCV. As this library is an extension to Intel's Open CV (BRADSKI; KAEHLER; PISAREVSKY, 2005), it is meant to add GPU acceleration for applications written with Open CV without code modification. Features included in this library are color conversion, histogram, and morphological operations.

Previous benchmarks with GPUCV showed speed gains of up to 18 times, in GeForce 7800 GTX (24 fragment processors) during a 3x3 erosion operation (FARRUGIA *et al.*, 2006). Besides that result, RGB to HSV conversion showed a speedup of more than 10 times. However, histogram computing with GPUCV is 10 times slower than the CPU optimized Open CV version (BRADSKI; KAEHLER; PISAREVSKY, 2005). Hough transforms on GPUCV are also slower than CPU optimized version. Conclusively, GPUCV hides graphic programming from developers while making available several computer vision

algorithms accelerated by the GPU.

2.3.4 MATLAB

An acronym for matrix laboratory, MATLAB is a numerical computing environment and programming language used by a wide range of researchers. Even though MATLAB uses the highly efficient ATLAS (WHALEY; DONGARRA, 1998), LAPACK (DONGARRA; WASŚNIEWSKI, 1998), and BLAS (LAWSON *et al.*, 1979) libraries for numerical linear algebra algorithms, higher performance can still be obtained by using GPU acceleration.

In order to interface with code written in C and FORTRAN, MATLAB uses .MEX[1] files. The main reason MEX files have been created is for calling existing C and FORTRAN code without having to rewrite them. Another useful feature is that .MEX files are compiled code, so better performance than interpreted MATLAB commands is expected.

In (BRODTKORB, 2007) it is explained how to develop a MATLAB interface for solving numerical linear algebra on the graphics processing unit. That thesis focuses on fragment shader implementation, so that it can be used in a wide range of GPUs. Besides that, algorithms are shown to be faster than highly efficient ATLAS MATLAB implementations.

Another way to exploit GPU capabilities in MATLAB is through the use of CUDA (SMITH, 2007). That white paper describes the steps required to integrate CUDA files with MATLAB. Since these files are not supported by MATLAB's native mex script, NVidia has developed *nvmex*, a script that calls *nvcc* compiler to generate CUDA MEX files. This script and available example code can be downloaded from *http://developer.nvidia.com/object/matlab_cuda.html*. The same white paper reports up to 15.7 times speed-up over standard MATLAB – running on Opteron 2210 – using a standard Quadro FX5600 GPU solving 400 Runge-Kutta steps on a 1024 x 1024 mesh for 2D isotropic turbulence simulation on Windows.

The main drawback of using GPU acceleration in MATLAB is that only single precision float formats are accepted, because of GPU hardware limitations. A workaround could emulate more precision, like 44-bit floating point number format presented in (Da Gracca; DEFOUR, 2006).

[1] MATLAB Executable

2.3.5 CTM

AMD's competing GPGPU technology for ATI Radeon based series is called "Close to the Metal" (CTM) and it provides a lower level API for better hardware usage (AMD, 2006). CTM is designed to open up the high-performance, floating-point, parallel processor array found in ATI graphics hardware. It is composed by:

- *ATI Data Parallel Processor Array*: comprises one or more processors that can execute a series of instructions. The processors are directed by the Processor Execution Unit. When processors are idle, the Processor Execution Unit may request that they execute a program. All the data for a program, as well as the instructions, are stored in memory and are read or written through the Memory Controller Unit. While all processors refer to the same instructions and constants, they may index different input, output and conditional data, exposing a SIMD programming model.

- *Processor Execution Unit*: the Processor Execution Unit interprets and forwards commands from a command buffer. Besides that, it is also responsible for distributing work to the processors in the processor array. In order to analyze CTM's performance, the Processor Execution Unit also maintains counters that store the total clocks since last reset and total clocks during which at least one processor was active since last reset.

- *Conditional Operation Unit*: the Conditional Operation Unit performs conditional tests for the Processor Execution Unit and for the Data Parallel Processor Array. Tests can always pass, always fail or may be a comparison from conditional value v from the client to a value b from a conditional output buffer.

- *Memory Controller Unit*: the Memory Controller Unit translates addresses to satisfy read and write memory requests for CTM's clients. It can access two kinds of memory: private memory accessible only by the memory controller and a system memory that is accessible not only by the memory controller but also by a host processor.

CTM's SDK aims to give application developers low-level access to the GPU for those that want it while providing high-level implementations to those that do not want low-level

access (HENSLEY, 2007). CTM's API has evolved into two pieces: *Hardware Abstraction Layer* (HAL) – which is a device specific, driver like interface – and *Compute Abstraction Layer* (CAL), whose core API is device independent, provides heterogeneous computing as well as optimized multi-core and GPU implementations.

Programmers developing for CTM can use AMD Core Math Library (ACML) and other tools like RapidMind (described in Section 2.3.8) or Peakstream, which is a spinout of the Brook project (described in Section 2.3.6). Besides these approaches there are also compiled high level languages and device independent assembly. Device specific Instruction Set Architecture (ISA) specifications are also provided through device specific extensions to CAL and HAL.

CTM's SDK is available on request for researchers but an open source version of CTM is available through SourceForge.net on the address *http://sourceforge.net/projects/amdctm/*.

2.3.6 Brook

Another attempt, developed by the Stanford University Graphics Lab, is BrookGPU, a system for general-purpose computation on programmable graphics hardware (BUCK *et al.*, 2004). The Brook compiler is called *brcc* and it supports an extended C language which includes data-parallel constructs that enable the GPU to work as a streaming co-processor. One of the most interesting features of Brook is its portability, since it can use not only different GPU models but also the CPU as a back end.

An important abstraction brought by Brook is that memory management is made via streams which are named and typed data objects consisting of collections of records. In order to execute data-parallel operations, parallel functions called kernels are invoked. A common paradigm in older GPU APIs, the fact that the output texture size must match the input texture size is avoided through the use of reduction functions.

The collection of data which can be operated in parallel, called streams, is declared through angle-bracket syntax, as in **float myStream<50,10>**, which denotes a two-dimensional stream of floats. The main difference between streams and arrays is that access to the former is restricted only to kernels and to *streamRead* and *streamWrite*

instructions – these instructions transfer data between memory and streams. Another feature of Brook is that streams may contain structures composed of *float, float2, float3 and float4* native types. An example of a stream of rays from (BUCK *et al.*, 2004) can be described like:

```
typedef struct ray_t {
  float3 o;
  float3 d;
  float tmax;
} Ray;
Ray r<100>;
```

In order to act over streams, special functions called *kernels* are used. These functions are defined with the keyword *kernel* and their body is executed over each element of the stream. An example kernel also taken from (BUCK *et al.*, 2004) is shown below.

```
kernel void saxpy (float a, float4 x<>, float4 y<>, out float4 result<>) {
  result = a*x + y;
}
```

All input streams are read only while output streams, specified by the *out* keyword, can be written. If streams of different dimensionality are used as parameters for a kernel, each input array is resized to match the output by either repeating or striding elements in each dimension.

Besides using input and output streams, Brook supports gather streams, which are specified by the C array syntax, like in *array[]*. These streams permit arbitrary indexing and a Brook programmer is forced to distinguish them between data streamed to a kernel. This way, input stream data cannot pollute a traditional cache and locality is not penalized in gather operations.

An important operation over streams is *reduction*, a data parallel method for calculating a single value from a set of records such as an arithmetic sum, computing a maximum or matrix product. Besides the requirement that these operations need to be associative, so that the system can evaluate operations in whichever order it needs, reduce streams are specified with the *reduce* keyword.

Brook is implemented as a compilation and runtime system that maps Brook language to existing GPU APIs through the use of the *brcc* compiler and a runtime library. *brcc* is a source-to-source compiler and it maps Brook kernels to Cg shaders (as described in Section 2.3.10). Cg shaders are translated to GPU assembly through vendor-provided shader compilers.

The Brook Runtime (BRT) is a library that provides runtime support for kernel execution. It offers three back ends: OpenGL, DirectX and a reference CPU implementation (BUCK *et al.*, 2004). This portability is important for choosing the best back end for the underlying architecture as well as for API-specific feature optimization.

Several details of graphic APIs require Brook to create solutions for a transparent streaming programming language. For example, NVIDIA imposes a maximum size of 4096 by 4096 floating-point two-dimensional textures. If streams are mapped directly to textures, 1D streams will have no more than 4096 elements and Brook programs will not be able to create streams of more than two dimensions. A compiler option wraps stream data across multiple rows of textures to overcome this limitation. If more output streams than are supported by the hardware are specified to a kernel, *brcc* will split the kernel into multiple passes so that it can compute all of the outputs. Results show that this workaround yields slowdowns of 47% and 53% (BUCK *et al.*, 2004).

Important applications have been implemented in Brook and compared with GPU reference implementations (BUCK *et al.*, 2004) such as image segmentation, fast Fourier transform, ray tracer and some basic linear algebra routines. Benchmarks of these applications showed that using Brook is, in average, less than 10% slower than using optimized GPU API code, therefore being an interesting option for portable GPU coding.

2.3.7 Sh

Another project that started academically is the Sh embedded metaprogramming language for programmable GPUs, originally developed by the University of Waterloo Computer Graphics Lab (MCCOOL; QIN; POPA, 2002).

Although high level compiler for shading languages were available when Sh was created, the use of an external string specification was inconvenient, since no advantages

from a language like C++ could be used. This way, a high level shading language defined directly in the graphic API, using standard C++ code, would be able to support specialization of shader programs, modularity, and scoping constructs without any additional implementation effort (MCCOOL; QIN; POPA, 2002).

A Sh example that uses a class to control access to parameters, as well as a template argument to specify the number of light sources, is seen in Listing 2.1. Two shaders are defined in this code: a vertex shader – through the *SH_BEGIN_PROGRAM("gpu:vertex")* macro – and a fragment shader – through the *SH_BEGIN_PROGRAM("gpu:fragment")* macro. Some of the declaration parameters are marked with *Input* and *Output* qualifiers. These *attributes* are bound to the input and output data channels of the shaders using a set of rules that depend on their order of declaration as well as their type (MCCOOL *et al.*, 2004). Another advantage of using C++ modularity is that shaders can be constructed algorithmically. Besides that, no glue code is required to bind shaders to the host application. More details on the Blinn-Phong lighting model can be found in (BLINN, 1977).

In Sh, stream objects are represented by the *ShChannel* and *ShStream* class. While channels are sequence of elements of the type given as its template argument, streams are containers for several channels of data. If the *"gpu:stream"* or *"cpu:stream"* profile is chosen to compile an *ShProgram*, this program can be parallelly applied to streams. In order to apply a kernel program to a stream, the *connect* operator is used. An example command of applying the program p to an input stream a so that b receives the output is defined as follows:

```
b = p << a;
```

This way, the use of *ShChannel* and *ShStream* class enable a general-purpose stream processing computational model through the Sh language. Documentation and the library itself can be found at *http://libsh.org/*.

2.3.8 RapidMind

RapidMind is a development and runtime platform that aims to enable applications to take full advantage of multi-core processors. This platform enables programmers to keep

Listing 2.1 – Encapsulated Blinn-Phong lighting model

```
template <int NLIGHTS>
class BlinnPhong {
  public:
    ShTexture2D<ShColor3f> kd;
    ShTexture2D<ShColor3f> ks;
    ShAttrib1f spec_exp;
    ShPoint3f light_position[NLIGHTS];
    ShColor3f light_color[NLIGHTS];
    template <ShVariableKind IO> struct VertFrag {
      ShPoint<4,IO,float> pv;      // position (VCS)
      ShTexCoord<2,IO,float> u;    // texture coordinate
      ShNormal<3,IO,float> nv;     // normal (VCS)
      ShColor<3,IO,float> ec;      // total irradiance
    };
    ShProgram vert, frag;
    BlinnPhong (int res) : kd(res,res), ks(res,res) {
      vert = SH_BEGIN_PROGRAM("gpu:vertex") {
        ShInputNormal3f nm;        // normal vector (MCS)
        ShInputTexCoord2f u;       // texture coordinate
        ShInputPosition3f pm;      // position (MCS)
        VertFrag<SH_OUTPUT> vf;
        ShOutputPosition4f pd;     // position (HDCS)
        vf.pv = modelview | pm;
        vf.u = u;
        vf.nv = normalize(modelview | nm );
        pd = perspective | vf.pv;
        for (int i=0; i<NLIGHTS; i++) {
          ShVector3f lv =
            normalize(light_position[i] - vf.pv(0,1,2));
          vf.ec += light_color[i] * pos(vf.nv|lv);
        }
      } SH_END;
      frag = SH_BEGIN_PROGRAM("gpu:fragment") {
        VertFrag<SH_INPUT> vf;
        ShOutputColor3f fc;          // fragment color
        ShVector3f vv = normalize(-vf.pv(0,1,2));
        ShNormal3f nv = normalize(vf.nv);
        fc = kd(vf.u) * vf.ec;
        ShColor3f kst = ks(vf.u);
        for (int i=0; i<NLIGHTS; i++) {
          ShVector3f lv =
            normalize(light_position[i] - vf.pv(0,1,2));
          ShVector3f hv = normalize(lv + vv);
          fc += kst * pow(pos(hv|nv),spec_exp) * light_color[i];
        }
      } SH_END;
    }
};
```

Listing 2.2 – RapidMind's definition of a reflection function

```
Value3f
reflect (Value3f v, Value3f n) {
  return Value3f(2.0f*dot(n,v)*n - v);
}
```

writing code in standard C++ while RapidMind's tools and processes parallelize code across multiple cores. A fundamental advantage of this platform is that besides compiling code to GPUs it also creates code for Cell architecture.

The RapidMind platform acts as an embedded programming language inside C++. It is built around a small set of types that can be used to specify arbitrary computations. These are done through the parallel execution of functions over arrays. Scatter, gather and programmable reduction are also supported (MCCOOL, 2006).

Another unique feature of RapidMind is that it includes an extensive runtime component as well as interface and dynamic compilation components, thus, automating common tasks such as task queuing, data streaming, data transfer, synchronization, and load-balancing (MCCOOL, 2006). According to the author, the programming model is safe and programs written to the platform cannot suffer from deadlock, read-write hazards, or synchronization errors.

The interface for programming RapidMind is based on three main C++ types: *Value*, *Array*, and *Program*. The first two types are containers for data, while *Program* is a container for operations. Parallel computations are invoked by applying programs to arrays to create new arrays, or by reduction operations (MCCOOL, 2006).

The *Value* type is declared as *Value<N,T>* and it holds N values of type T. For instance, *Value4f* is a 4-tuple of single precision floating point numbers while *Value3bool* is 3-tuple of Booleans. Computations on RapidMind are expressed using tuples. An interesting feature is that operations on value types directly express SWAR (SIMD within a register) parallelism. An example of a function that calculates the reflection of a vector from a 3D plane given by a normal, taken from (MCCOOL, 2006) is as in Listing 2.2

Arrays are also used as data containers and are declared in the *Array<D,T>* form, in which the integer D is the dimensionality and may be 1, 2, or 3 while T is the type of the element, restricted to instances of the *Value* type. Besides storing values, arrays can also

Listing 2.3 – Example of RapidMind's program object and how it is invoked

```
Value3f n;
Program p = BEGIN {
    In<Value3f> v;
    Out<Value3f> r;
    Value3f nn = normalize(n);
    r = reflect(v,nn);
} END;
Array<2,Value3f> V(1000,1000);
// initialize V with data here
Array<2,Value3f> R;
R = p(V);
// read back the result from R here
```

be randomly accessed using the "[ ]" and "( )" operators. It must be noted that subarrays can be accessed through **slice**, **offset**, and **stride** functions (MCCOOL, 2006).

Another important type of RapidMind's platform is the *Program*. A program object stores a sequence of operations through the *retained mode* which is different from the normal *immediate mode* that the platform executes. That mode is enabled by the use of *BEGIN* and *END* keyword macros. In this mode, operations on tuples are not executed but instead symbolically evaluated and stored in the program object. These program objects are compiled using a dynamic compiler that takes advantage of the low-level features of the target machine (MCCOOL, 2006). An example from the same author can be seen in Listing 2.3.

This example uses the above defined *reflect* function as well as a *normalize* library function. Besides that, the keywords *In* and *Out* are used to specify inputs and outputs of the program. Before a program is executed, the input values will be initialized with input data and when it is finished, the variables marked as outputs are taken as the output of the program. A program is executed when it is applied to either tuples or Arrays, as through the *R = p(V);* command. This expression applies the program p to all elements of the V array. It must be also noted that programs accept control flow instructions like *IF/ELSEIF/ELSE/ENDIF*, *FOR/ENDFOR*, *WHILE/ENDWHILE* and *DO/UNTIL* (MCCOOL, 2006).

RapidMind's implementation on the GPU uses OpenGL as a back end, which works on both NVIDIA and ATI. As the GPU memory model is quite different from that of Cell BE, memory management has been considered specifically for GPUs. In particular, video

memory is most efficiently accessed by DMA transfers to and from the host memory than as shared memory. An interesting point of the GPU back end is that it can also be used to specify shaders (MCCOOL, 2006).

The performance of RapidMind platform has shown results equivalent to the best available GPU implementations made directly through OpenGL. However, it is reported (MCCOOL, 2006) that more direct access to the hardware would provide opportunities for even higher performance. This way, RapidMind seems to be a good choice for portable applications. Documentations, tutorials and sample applications as well as the download of the platform itself can be found at *http://developer.rapidmind.net/*.

2.3.9 Microsoft Accelerator

Microsoft's answer to GPGPU is Accelerator (TARDITI; PURI; OGLESBY, 2006), which combines an imperative language with a library that provides only high-level data-parallel operations without exposing any aspects of GPU to programmers. Data parallel instructions are compiled on the fly to optimized GPU Pixel shader code and API calls.

In order to use GPU's parallelism, Accelerator provides an abstract data type called parallel array whose elements must be of the same type. The implementation on C#, which is a Java-like language from Microsoft, features 4 subclasses derived from the abstract class ParallelArray: FloatParallelArray, IntParallelArray, BoolParallelArray and Float4ParallelArray – a vector of 4 floating point values. Six classes of operations are available for these types of arrays: construction, conversion to ordinary arrays – arrays that do not support parallel operations –, element-wise operations, reductions, transformation on arrays and linear algebra.

Element wise operations comprise *add, subtract, multiply, divide, min, max, and, or* as well as *compare* operations over each of the elements of two arrays. *Reductions* are operations across a particular dimension, for instance, the element sum of the first dimension of a two-dimensional array is a *row sum reduction*. Other reductions are *product, max* and *min*. *Rotations, shifts* and *replications* of arrays are included in array transformations. Linear algebra instructions are standard *inner* and *outer products* over two arrays.

A simple example of a program that uses Accelerator taken from (TARDITI; PURI;

Listing 2.4 – Simple example of C# program which uses accelerator

```
1  using Microsoft.Research.DataParallelArrays;
2  static float[,] Blur(float[,] array, float[] kernel)
3  {
4      float[,] result;
5      DFPA parallelArray = new DFPA(array);
6      FPA resultX = new FPA(0f, parallelArray.Shape);
7      for (int i = 0; i < kernel.Length; i++) {
8          int[] shiftDir = new int[] { 0, i};
9          resultX += PA.Shift(parallelArray, shiftDir) * kernel[i];
10     }
11     FPA resultY = new FPA(0f, parallelArray.Shape);
12     for (int i = 0; i < kernel.Length; i++) {
13         int[] shiftDir = new int[] { i, 0 };
14         resultY += PA.Shift(resultX, shiftDir) * kernel[i];
15     }
16     PA.ToArray(resultY, out result);
17     parallelArray.Dispose();
18     return result;
19 }
```

OGLESBY, 2006) is shown in Listing 2.4. This example shows how to calculate a 2-D convolution in an image passed as parameter called *array* with a given *kernel*. Each pixel value is retrieved from the multiplication by the weight given in the kernel and summing this value. A parallel array is converted from a C# array through the *DFPA* constructor. Firstly, values in the x direction are calculated by repeatedly shifting the original image by i pixels and multiplying the shifted image by its appropriate weight with the overloaded * operator. After that, the same procedure is repeated in the y direction.

The translation of parallel arrays into GPU code is made through pixel shaders, as seen in Section 2.3.10. In order to optimize this translation, Accelerator tries to decrease the number of generated shaders by using a Directed Acyclic Graph (DAG). Afterwards, the target graphics API is used to compile and run each pixel shader in supported DirectX Pixel Shader in versions 2 and 3.

Benchmarks of Accelerator against hand-written pixel shader(TARDITI; PURI; OGLESBY, 2006) show that the speed of the former are typically within 50% of the speed of hand-written pixel shader code, while benchmarks against C versions on CPU have shown results up to 18 times faster.

FIGURE 2.3 – Example of real-time water reflection image generation using shaders

2.3.10 Shaders

A shader is a set of software instructions compiled to the graphic processing units in order to perform rendering effects, such as the one seen in Figure 2.3. According to (OWENS *et al.*, 2007), the modern graphics pipeline resembles the one seen in Figure 2.4. All geometric primitives are processed in each of these stages: vertex operations (scaling, rotating), primitive assembly, rasterization, fragment operations, and final image composition. These pipelines are generally implemented as separate pieces of hardware, which parallelly process vertices. With the advent of shaders, the vertex and the fragment processors stages have become programmable (KESSENICH, 2006).

According to (KESSENICH, 2006), the vertex processor is a programmable unit that operates on incoming vertices and their associated data. The *vertex shaders* are compilation units that may be written in the OpenGL Shading Language to run on this processor. The *vertex shader executable* is a complete set of vertex shaders that are compiled and linked. It must be noted that the vertex processor operates on one vertex at a time and that it does not replace graphic operations that require the knowledge of several vertices at a time.

The *fragment processor* is another programmable stage whose compilation units, that can also be written in the OpenGL Shading Language, are called *fragment shaders*. These shaders cannot change a fragment's x or y position nor have access to its neighboring fragments. The values computed by the fragment shader are ultimately used to update frame-buffer memory or texture memory (KESSENICH, 2006).

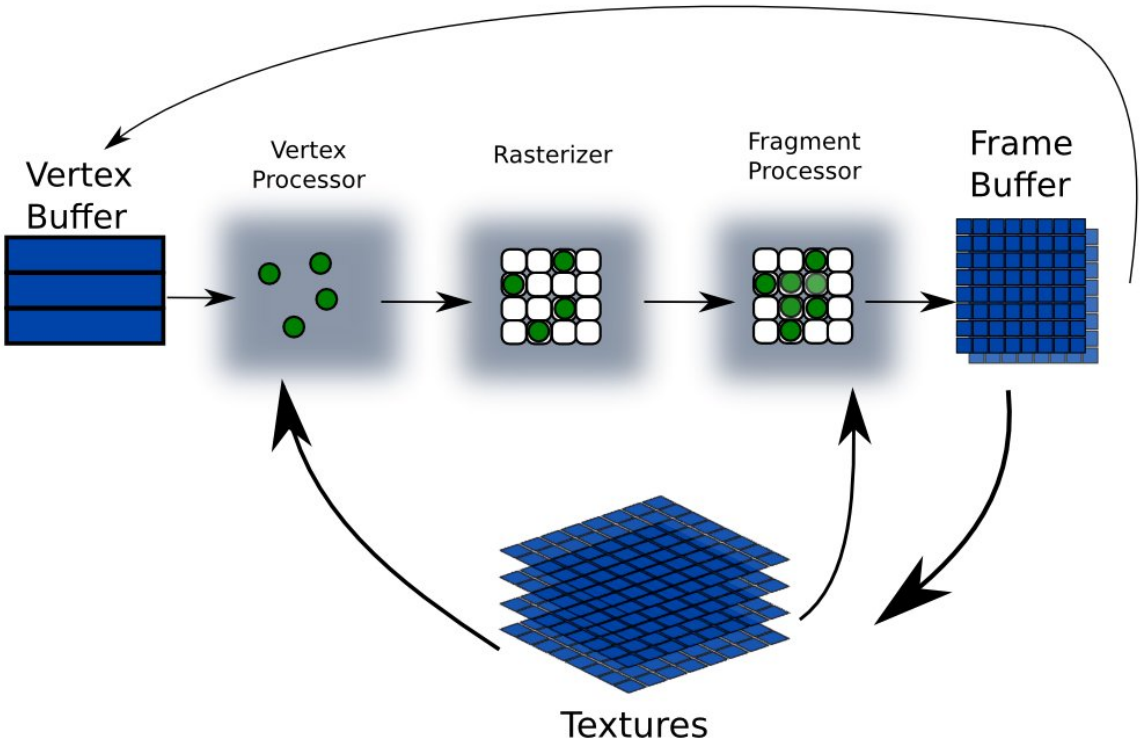


FIGURE 2.4 – GPU graphics pipeline

Besides programming shaders in the OpenGL Shading Language, another interesting shader programming language is Cg, which stands for "C for graphics" (NVIDIA, 2007). It was developed by NVIDIA in close collaboration with Microsoft Corporation and is compatible with both OpenGL and DirectX.

Main types supported by the OpenGL Shading Language are *bool, int, float, void* and its two, three and four component vectors, like *ivec2, ivec3*, and *ivec4* integer vectors. The language also defines square matrices types like *mat2, mat3,* and *mat4* as well as rectangular ones. Another interesting type are the samplers, which are handlers for accessing textures (KESSENICH, 2006).

A simple vertex shader example from Lighthouse 3D website (*www.lighthouse3d.com*) is the flatten shader. It is listed in Listing 2.5.

This shader takes a vertex as input through the vertex shader built-in attribute *gl_Vertex* (KESSENICH, 2006). As it is a *read-only* attribute, it must me copied to a local variable v in order to be set to zero. This way the appearence of the object becomes flat in Z direction. The *gl_Position* built-in variable is available in the vertex shader and it is used for writing the homogeneous vertex position. It is used to update the former value

Listing 2.5 – OpenGL Shading Language flatten shader

```
void main(void)
{
  vec4 v = vec4(gl_Vertex);
  v.z = 0.0;
  gl_Position = gl_ModelViewProjectionMatrix * v;
}
```

of *gl_Vertex* through the use of a *ModelView* projection matrix and the flattened vertex so that the overall appearence of the object becomes flat.

2.4 GPU Hardware Model

In order to parallelly process a great amount of vertices and fragments at the same time, GPUs rely on a set of multiprocessors. It's clear from (NVIDIA, 2007d) that *"the device is implemented as a set of multiprocessors as illustrated in Figure 2.5. Each multiprocessor has a Single Instruction, Multiple Data architecture (SIMD): At any given clock cycle, each processor of the multiprocessor executes the same instruction, but operates on different data"*. This way, GPUs can be seen as a set of SIMD Multiprocessors with on-chip shared memory, which is rather different from PRAM (Parallel Random Access Machine) models (MILLER; BOXER, 2000). The latter models were created in order to be widely used as a parallel model of computation, so that they would do for parallel computing what the RAM[2] model did for sequential computing. The PRAM model consists of a set of processors (P_1, P_2, ..., P_n) each of which identical to a RAM processor, a memory element – this is a common global memory, which is seen by all processors – and a memory access unit, which is supposed to access every memory location in $\Theta(1)$ time, for every processor. It is important to note that processors will need to communicate to each other through common shared memory if they wish to cooperate.

Several facts refrain GPUs from being classified as general PRAMs. One of them is that, depending on the instructions issued, different classifications would apply. From (NVIDIA, 2007d), *"If the instruction executed by a warp[3] writes to the same location in*

[2]The Random Access Machine (RAM) consists of a single processor and memory. This is considered the traditional sequential model of computation capable of accessing any location of memory in $\Theta(1)$.

[3]Each SIMD group of threads

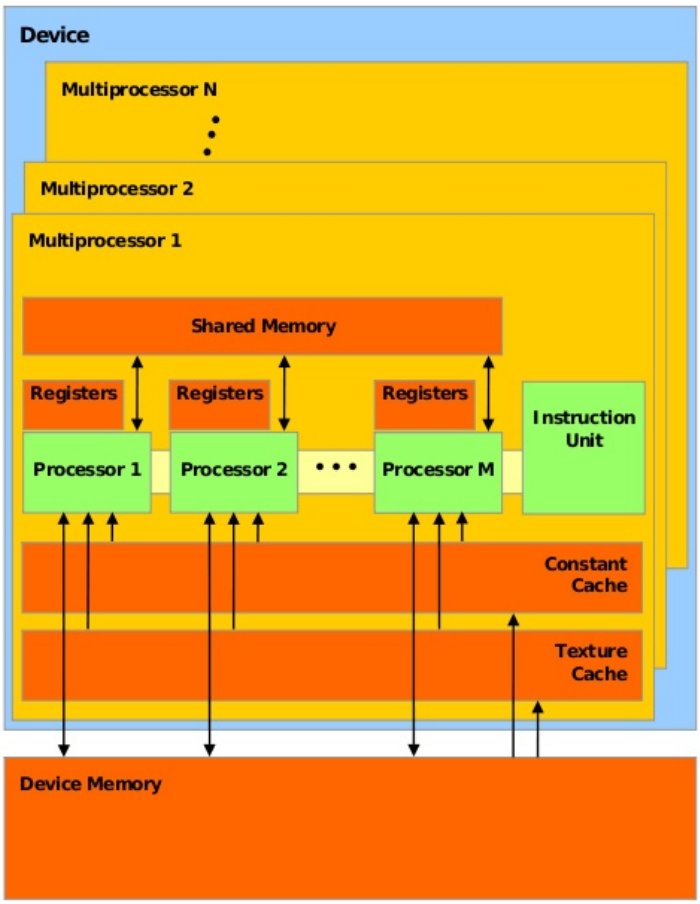


FIGURE 2.5 – Hardware model for NVIDIA's G80 architecture (NVIDIA, 2007d)

global or shared memory for more than one of the threads of the warp, how many writes occur to that location and the order in which they occur is undefined, but one of the writes is guaranteed to succeed". This would classify G80 architecture as an arbitrary concurrent-read, concurrent-write PRAM. But atomic instructions can be used, *"If an atomic instruction executed by a warp reads, modifies, and writes to the same location in global memory for more than one of the threads of the warp, each read, modify, write to that location occurs and they are all serialized, but the order in which they occur is undefined"*, so all processors are granted to read and write. Atomic instructions do not fit in any of PRAM classifications. Another reason for not modeling GPUs as PRAMS is that it considers all memory access times equal to every processor. Since memory accesses are important parts of algorithm implementations on GPUs, this modeling must be very careful.

2.5 Livewire algorithm

Livewire (FALCÃO, 1997) or Intelligent Scissors (MORTENSEN; BARRETT, 1998) is a semi-automatic image segmentation technique that allows a user to easily and quickly select regions of interest on an image using mouse clicks to delineate the edges. When the user starts the selection of the region with a mouse click, a virtual wire is created linking the first clicked point – referred as anchor – to the point where the mouse is over, following a path that is as close as possible from image features detected as edges. Figure 2.6 shows the result of a user segmentation using such tool.

This algorithm is widely used in the field of user steered image segmentation, as seen in Figure 2.7. This picture shows a longitudinal slice of coronary intra-vascular ultra-sound exam and how livewire is used to delineate a region between both edges of the coronary, known as lumen. The algorithm can be easily adapted to measure the area of regions of interest as well as the length of 1-D segmentations.

In order to implement livewire algorithm, several steps are required. Firstly, the image goes through a convolution with a Sobel filter – more details in Section 2.5.1 – so that edge features can be extracted. From the filtered image a graph is built, considering its pixels as nodes. Edges are created from every pixel to its up, down, left and right neighbor

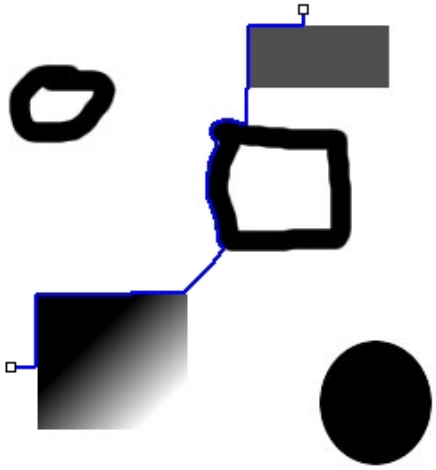


FIGURE 2.6 – Result of shortest path following edges in livewire segmentation after the user has positioned two points

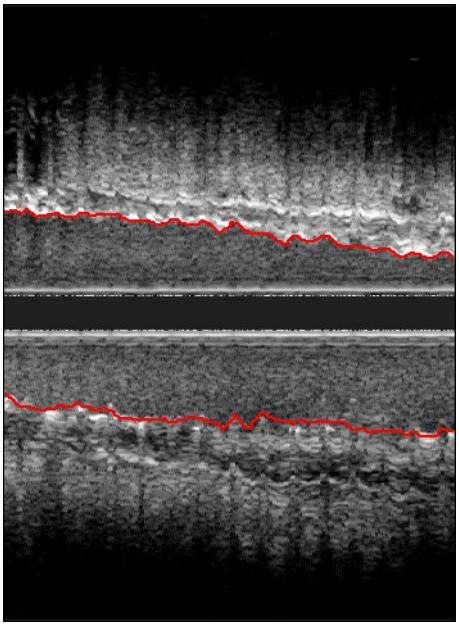


FIGURE 2.7 – Segmentation of North-South longitudinal slice of coronary ivus image using IVUS and LiveWire plugins

pixels. These edges are weighted with features gathered from the sobel filter convolution, so that pixels that stay on the edge are lighter and the ones that go outside the edge are heavier. Several costs are described in (BAGGIO, 2006), but the most important is the gradient magnitude, and this is the one used in this thesis.

2.5.1 Sobel filter

Internally, a digital image is represented as a matrix of pixels. Each pixel can have one or more channels. For gray scale images, each pixel is stored in a single channel, whose values typically range from 0 to 255. Values nearer zero are displayed darker and values nearer 255 are displayed closer to white. Colored images have three channels, which can be represented in several different color spaces (FAIRCHILD, 1998). Most frequently used are RGB (red, green and blue), HSV (hue, saturation and value) and HSL (hue, saturation and luminance). Another useful channel is the alpha one, often used along with RGB, in a color space called RGBA. Alpha is used to create the appearance of partial transparency when one image is over another. Two simple equations that show how alpha channel can be used to combine two colors, when *color A* is over *color B* are the following (PORTER; DUFF, 1984):

$$C_o = C_a\alpha_a + C_b\alpha_b(1-\alpha_a) \tag{2.1}$$

$$\alpha_o = \alpha_a + \alpha_b(1-\alpha_a) \tag{2.2}$$

where C_o is the output color of the operation, C_a is the color of pixel in region A, C_b is the pixel color in region B and α_a and α_b are the respective alpha channel values of region A and region B. α_o is the resulting alpha channel value. The output color of this equation can be seen in Figure 2.8.

Another parameter used in image representation is the color depth, which is the number of bits used to represent the color of a single pixel in a bitmapped image. A low level bit depth, used to represent 256 different colors, is achieved by encoding the 3 bits for the red channel, 3 bits for the green one and 2 bits for the blue channel. This way, there are $8 \times 8 \times 4 = 256$ different colors. For life like colors, the *TrueColor* encoding is used,

FIGURE 2.8 – Using alpha channel to combine blue and green colors. The intersection area is drawn with values obtained from equation 2.1

producing 16.8 million colors. This system uses $2^8 = 256$ levels for each channel.

The first step in livewire algorithm is an image pre-processing, that uses filters. In the case of images with noise, a Gaussian filter may be used to remove it.

After the noise has been decreased, an edge detection filter must be applied. As described in (MORTENSEN; BARRETT, 1995), an useful way to find edges in an image, is using the gradient operator. As known from vector calculus, the gradient of a *scalar field* is a *vector field* which points in the direction of the greatest rate of increase of that field and whose magnitude is the greatest rate of change. It is mathematically represented by a column vector whose components are the partial derivatives of f (where f is a scalar field) as in equation 2.3, which is the specific equation for a two-dimensional scalar field.

$$\bigtriangledown f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2} \right)^T \tag{2.3}$$

In order to retrieve the scalar field from the image, it is enough to get each of the channels from a colored image, which would create three different vector fields. Another possibility is to convert the colored image into a gray scale one and use the converted values for a single scalar field. A simple way to convert a colored image to a gray scale one is using equation 2.4. Since the human eye is more sensitive to the green channel and

less sensitive to the blue one, a higher weight is given to green.

$$Value = 0.3 \times Red + 0.59 \times Green + 0.11 \times Blue \tag{2.4}$$

The scalar field derived from an image is discrete, and no analytic expression can be obtained to calculate the gradient. A good operator to do so is described in (ABDOU; PRATT, 1979), the Sobel filter. It is technically a discrete differentiation operator, which computes the gradient's approximation of the image's intensity function. It can be implemented by two 3×3 kernels convolved with an image, one for the horizontal direction and another for the vertical one. Considering an image T, the horizontal and vertical derivative approximations, G_x and G_y respectively are defined as the following convolutions with T:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * T \qquad \text{and} \qquad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * T \tag{2.5}$$

The following expressions are used to obtain the gradient magnitude and direction:

$$G = \sqrt{G_x^2 + G_y^2} \tag{2.6}$$

$$\theta = \arctan\left(\frac{G_y}{G_x}\right) \tag{2.7}$$

After the Sobel filter has been applied to an image, the next step, the graph building may be taken.

2.5.2 Laplacian of Gaussian

In order to calculate the Laplacian and Gaussian in a single convolution, the function Laplacian of Gaussian (LoG)(WHITAKER, 1996) was used:

$$LoG(x, y) = -\frac{1}{\pi\sigma^4}\left[1 - \frac{x^2 + y^2}{2\sigma^2}\right] e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.8}$$

where σ is the standard deviation of the Gaussian filter.

2.5.3 Graph building

As stated before, the main importance of creating the Sobel image is to build a graph using pixels as nodes. In order to reflect the best edge detection, several different edge costs might be chosen to each of the edges. The procedure chosen for this thesis was to use the gradient magnitude as defined in equation 2.6. One of the problems of this approach is that values found in an image are not upper bounded, so they might need to be normalized. A simple way to normalize the value at point **P** is to divide its value by the maximum gradient value found in the image:

$$G_n = \frac{G_{\mathbf{P}}}{G_{max}} \tag{2.9}$$

where G_n is the normalized value of the gradient magnitude at the point **P** G_P. Since points with high magnitude show strong edge features, the cost assigned to a path that goes through these points should be very low. Besides that, neighbor pixels that are located in the diagonal should have higher cost than the ones that are not, because the distance is higher. An equation that considers all these properties is 2.10, which shows the cost of going from pixel **p** to pixel **q** as a normalized inverse linear ramp of the gradient magnitude G (MORTENSEN; BARRETT, 1998).

$$f_G(\mathbf{p}, \mathbf{q}) = \left(1 - \left(\frac{G(\mathbf{q}) - G_{min}}{G_{max} - G_{min}}\right)\right) . \frac{\| \mathbf{p} - \mathbf{q} \|}{\sqrt{2}} \tag{2.10}$$

This equation is useful for making the shortest path in the graph follow edges because going from one pixel to another that is not in the edge will have a cost value that is near one, while staying in the edge has cost near zero.

2.5.4 Parallel single source shortest path algorithms

Several algorithms have been created to find the shortest path in a graph, from a single source. Firstly, the problem will be stated so that a formal analysis of the problem can be conducted.

Let $G = (V, E)$ be a graph with n vertices and m edges. Let $s \in V$ denote the start vertex. For each edge $e \in E$ there's an assigned non-negative real weight by the length

Listing 2.6 – Pseudo-code for Dijkstra's algorithm

```
1  function Dijkstra(Graph, source):
2      for each vertex v in Graph:// Initializations
3          dist[v] := infinity       // Unknown distance function from s to v
4          previous[v] := undefined
5      dist[source] := 0             // Distance from s to s
6      Q := copy(Graph)              // Set of all unvisited vertexes
7      while Q is not empty:         // The main loop
8          u := extract_min(Q)       // Removes best vertex from priority queue;
9          for each neighbor v of u:// returns source on first iteration
10             alt = dist[u] + length(u, v)
11             if alt < dist[v]      // Relax (u,v)
12                 dist[v] := alt
13                 previous[v] := u
```

function $l : E \rightarrow \mathbb{R}$. The *weight of a path* is defined as the sum of the weight of its edges. A solution to the single source shortest path with non-negative edge weights consists of computing $\delta(v)$, the weight of the *shortest* – minimum weighted – path from s to v. $\delta(v)$ is defined as ∞ when v is unreachable from s. Initially, $\delta(s)$ is set to zero.

It must be noted that real world graphs are often very large, with graphs and edges ranging from hundreds of thousands to billions. Besides being very important to the livewire algorithm, finding the shortest path is a classical combinatorial optimization problem and there has been given increasing importance to complex network analysis domain.

One of the most used solutions to the single source shortest path problems is Dijkstra's algorithm (DIJKSTRA, 1959). The original approach is bounded by $O(n^2)$ in time. The original algorithm is described in pseudo-code in Listing 2.6. The main idea is to keep a queue with all unvisited vertices and go visiting one by one, starting at the initial vertex. Each time a vertex with the lowest cost is visited, its neighbors are all updated in a process called relaxation. The process of finding the minimum element in the queue can be very long so an optimized data structure such as a heap can be used to decrease the running time. If a Fibonacci heap is used instead, the algorithm belongs to $O(m+n\log n)$ in time. A different algorithm is proposed in (THORUP, 1999). Instead of visiting vertexes in the order of increasing distance, a *component tree* is traversed. Two linear average time algorithms for uniformly distributed edge weights have been proposed (GOLDBERG, 2001) and (MEYER, 2003).

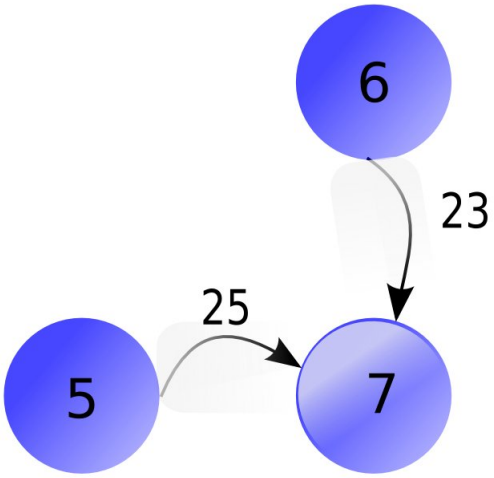


FIGURE 2.9 – Racing condition, where two vertexes in the queue – 5 and 6 – try to update the distance to vertex 7 with different values at the same time

In order to obtain further speedup for solving the single source shortest path, parallel attempts appeared in the literature. A trivial idea (QUINN, 1994) to parallelly optimize code given in Listing 2.6, is visiting each neighbor of u in parallel, in *lines 9 to 13*. Although this strategy would yield some speedup, it is limited by the number of edges that can be expanded in each iteration. For sparse graphs this can present quite limited speedup.

Another attempt to enhance performance is parallelizing the while loop in *line 7*. Two issues can occur. One of them, shown in Figure 2.9, is that while expanding vertexes in the queue, two of them might try to update the same destination vertex at the same time with different values, causing a write after write inconsistency, because a larger value may be updated instead of the smallest one. A locking approach is needed to avoid such hazards, which might decrease performance. Another issue is that the first vertex in the queue might update the distance of the second one, and, in case both of them have been expanded in parallel, a reinsertion will be needed for the second vertex, with smaller value. When it happens, no speedup has been caused by the parallel expansion, since it will need to be updated again. Of course, lines 2 up to 4 can be easily parallelized, but it takes them quite a small time compared to the other lines.

It is clear that several constraints limit Dijkstra's algorithm to be enhanced using the original sequential algorithm. Parallel algorithms for solving the non-negative edge weight single source shortest path are reviewed in detail (MEYER; SANDERS, 2003), (MEYER,

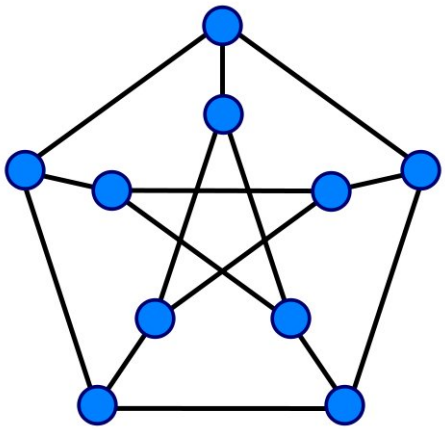


FIGURE 2.10 – The Petersen graph, a notable 3-regular graph

2002). According to them, there are no PRAM algorithms that run in sub-linear time and $O(m + n\log n)$ work. While parallel queues have been proposed (BRODAL; TRAFF; ZAROLIAGIS, 1998), Dijkstra's algorithm using these data structures present a worst-case time bound of $\Omega(n)$, as only their edge relaxations are made in parallel.

If the exact solution is not needed, there are approximate algorithms to solve the non-negative edge weight single source shortest path (KLEIN; SUBRAMANIAN, 1997), (SHI; SPENCER, 1999) that run in sub-linear time based on the randomized Breadth-First search algorithm (ULLMAN; YANNAKAKIS, 1990). However, it is not know if the same approach can be used to determine exact solutions.

An interesting parallel algorithm, called Δ-stepping, for the non-negative edge weight single source shortest path problem is given in (MEYER; SANDERS, 2003), which divides Dijkstra's algorithm in a number of phases that can be run in parallel. This algorithm runs in sub-linear time for random graphs with uniformly distributed edge weights, with linear average case work (MADDURI *et al.*, 2006). Almost linear speedup has been observed in a random d-regular[4] graph instance (2^{19} vertexes and $d = 3$) (MEYER; SANDERS, 2003), running with 9.2 speedup in 16 processors of an Intel Paragon machine with a distributed memory implementation of the Δ-stepping algorithm. Other reports (MADDURI *et al.*, 2006) show 14.82 speedup on 16 processors and 29.75 speedup on 40 processors for the same random d-regular graph family of size 2^{29} vertexes and $d = 3$, on the Cray-MTA architecture (described in Section 2.2.4). Further details about the Δ-stepping algorithm can be found in Section 3.2.1.

[4]A d-regular graph is a graph in which every vertex has the same degree d, as in Figure 2.10.

Experimental studies on parallel solutions for the NSSP (non-weighted single-source shortest path problem) are shown in (HRIBAR; TAYLOR; BOYCE, 1997), (PAPAEFTHYMIOU; RODRIGUE, 1994),(HRIBAR; TAYLOR; BOYCE, 1998). Other implementation results focusing distributed machines, resort to graph partitioning (GREGOR; LUMSDAINE, 2005),(ADAMSOM; TICK, 1991) and running a sequential NSSP algorithm on the generated sub-graph.

A main feature of PRAM implementations of graph algorithms for arbitrary sparse graphs is that they are typically memory intensive, loaded with fine-grained and highly irregular dependent memory accesses. Even cache-based systems yield poor performance due to these features. The best sequential implementations are barely outperformed by parallel graph algorithms because of long memory latencies and high synchronization costs (BADER *et al.*, 2001),(BADER; CONG; FEO, 2005).

No previous study shows implementation of Δ-stepping algorithm on a GPU architecture, although (MADDURI *et al.*, 2006) presents a Cray MTA-2 implementation. Since it is a massively multithreaded parallel machine with high-end shared memory it is a great choice for fine-grained parallelism. The MTA-2 has no data cache and uses hardware multithreading to tolerate the latency. Recent results show that this architecture has presented exceptional performance on key graph and combinatorial optimization problems such as list ranking (BADER; CONG; FEO, 2005), subgraph isomorphism (BERRY *et al.*, 2006) and Breadth-First Search and st-connectivity (BADER; MADDURI, 2006). A better explanation of how Δ-stepping works and the main changes needed to run in a GPU architecture are given in Section 3.2.

2.6 Summary

This chapter showed key details on how to program for a GPU as well as its hardware model. This way, enough background is given to show that parallel algorithms must be developed to take full advantage of the high processing power available in GPUs. Besides that, it is shown that almost no advantage will be obtained if a traditional serial implementation of Dijkstra's algorithm using a heap is applied to the GPU architecture. Given this background, the next chapter gives further details on how a parallel solution to the single source shortest path problem is solved on a GPU architecture, as well as a

parallel solution for filtering the image.

3 GPU Livewire Algorithm Implementation

As all the background information about *LiveWire* algorithm, GPU architecture, and the programming model were explained in the last chapter, this one focuses on how the algorithm was implemented on the GPU, so that it could take advantage of the underlying architecture. The first part of the chapter shows a straightforward implementation of the Sobel filter using standard Graphics API, through the use of a Cg shader. Then, Δ-Stepping algorithm implementation is explained, through the use of NVidia's CUDA API.

3.1 GPU Sobel filter

GPGPU code was implemented with the standard OpenGL API, based on original code *helloGPGPU*, from Mark J. Harris (www.gpgpu.org). Hence the kernel convolution is made through a pixel shader programmed in NVidia's Cg.

Firstly, the loaded bitmap is transferred to an OpenGL Texture structure, and then it is displayed over a rectangular polygon (*GL_QUAD structure*), in a framebuffer object. After that, the resulting drawing is copied to another texture. This texture is the segmentation target and uses the GPGPU concept that an array for GPGPU is a texture. The segmentation kernel is then uploaded to the GPU, in which it is compiled and applied in parallel to many fragments simultaneously, during an orthographic drawing of the image. It shows the concept of binding a computational kernel to a fragment program. The result is then copied back to a texture, what makes feedback possible.

3.2 GPU Δ-stepping implementation

Since it has been shown that a parallel implementation of the Δ-stepping algorithm yields good speedup and scalability (MADDURI *et al.*, 2006) in a massively multithreaded parallel machine with high-end shared memory as the Cray MTA-2, an implementation of the same algorithm in a GPU architecture is shown in this section.

3.2.1 Algorithm Details

Firstly, the pseudo-code in Figure 3.1 needs to be carefully analyzed in order to find details that will need to receive special attention for each of the architectures.

An implementation of the Δ-stepping algorithm is freely available online (MADDURI, 2006). This implementation relies on specific MTA pragma statements for critical section. If one pays attention to line 10 in Figure 3.1, since no duplicates are stored in R, a critical section is established, so that a new position in R can be retrieved, as well as the value of a new vertex distance can be updated. Since two operations need to be done, this issue cannot be solved with atomic operations.

Main differences in implementation are related to GPU memory model as well as different instruction set. Given memory architecture in section 2.3.2 and the hypothesis that the number of node and edges will be around 2^{20}, the Table 3.1 from (NVIDIA, 2007d) must be watched in order to choose the best place for each of the structures. Table 3.2 shows average parameters from NVidia G80 series, which support CUDA. Since shared memory available is not enough to store 2^{20} float values for the smallest distance to each vertex, which would yield 4MB, these values will need to be stored in global memory. Another good option would have been the texture memory, but it is *read-only* and distances need to be updated constantly.

Besides storing the distances to each vertex, the whole graph needs to be stored in device memory as well. Since grid graphs will have approximately $4n$ edges, and we are supposing the number of edges is around 2^{20}, there will be needed $4 \cdot 2^{20} \cdot 4 = 16MB$, since each float requires 4 bytes. For the size, the graph should be surely stored in global memory, but it's better to put it in texture memory, since it is *read-only* and will most probably be highly beneficed by texture memory cache, because the same edges are read

Δ-stepping algorithm:

1 **foreach** $v \in V$ **do**
2 $heavy(v) \leftarrow \{(v,w) \in E : l(v,w) > \Delta\}$;
3 $light(v) \leftarrow \{(v,w) \in E : l(v,w) \leq \Delta\}$;
4 $d(v) \leftarrow \infty$
5 relax(s,0);
6 $i \leftarrow 0$;
7 **while** B *is not empty* **do**
8 $S \leftarrow \emptyset$;
9 **while** $B[i] \neq \emptyset$ **do**
10 $Req \leftarrow \{(w, d(v) + l(v,w)) : v \in B[i] \wedge (v,w) \in light(v)\}$;
11 $S \leftarrow S \cup B[i]$;
12 $B[i] \leftarrow \emptyset$;
13 **foreach** $(v,x) \in Req$ **do**
14 relax(v,x);
15 $Req \leftarrow \{(w, d(v) + l(v,w)) : v \in S \wedge (v,w) \in heavy(v)\}$;
16 **foreach** $(v,x) \in Req$ **do**
17 relax(v,x)
18 $i \leftarrow i + 1$;
19 **foreach** $v \in V$ **do**
20 $\delta(v) \leftarrow d(v)$;

Procedure $relax(v,x)$:

1 **if** $x < d(v)$ **then**
2 $B[\lfloor d(v)/\Delta \rfloor] \leftarrow B[\lfloor d(v)/\Delta \rfloor \backslash \{v\}$;
3 $B[\lfloor x/\Delta \rfloor] \leftarrow B[\lfloor x/\Delta \rfloor] \cup \{v\}$;
4 $d(v) \leftarrow x$;

FIGURE 3.1 – Pseudocode for the Δ-stepping algorithm

TABLE 3.1 – Parameters of NVidia GPUs with compute capability 1.x

Parameter	**Value**
Maximum number of threads per block	512
Maximum size of the x-dimension of a thread block	512
Maximum size of the y-dimension of a thread block	512
Maximum size of the z-dimension of a thread block	64
Maximum size of each dimension of a grid of thread blocks	65535
Warp size	32
Number of registers per multiprocessor	8192
Shared memory available per multiprocessor	16KB
Constant memory available	64KB
Constant memory cache per multiprocessor	8KB
One-dimensional texture cache per multiprocessor	8KB
Maximum number of blocks that can run concurrently on a multiprocessor	8
Maximum number of warps that can run concurrently on a multiprocessor	24
Maximum number of threads that can run concurrently on a multiprocessor	768
Maximum width of texture bound to a 1-D CUDA array	2^{13}
Maximum width of texture bound to a 2-D CUDA array	2^{16}
Maximum height of texture bound to a 2-D CUDA array	2^{15}
Maximum width of texture reference bound to linear memory	2^{27}
Number of processors per multiprocessor	8

when reinsertions are made. As edges in a grid graph can be divided in directions up, down, left and right, it would be useful to divide the graph in 4 textures.

Other essential structures such as buckets and the request set R also need to be put in the device memory. An approximate number of buckets for the given size of nodes and edges, supposing an uniformly distributed edge cost, would be 1024. Measured size of buckets have yielded 32768 as a good number so that no bucket overflow happens. For such a big memory space, buckets will need to be store in global memory.

Since more than 10 thousand vertices have been found in the request set R during each relaxation, the shared memory is not big enough to hold all vertices and its distances, which would require around 100KB, and there's only 16KB, according to Table 3.2. The requested set is one of the most accessed structures during the algorithm. This way, not being able to store it in shared memory can generate a huge bottleneck for the implementation. Another alternative is to copy part of R to shared memory and then process each part separately. Either way, the set would have to be copied from and back to global memory. The best way to hide global memory access latency is executing arithmetic

TABLE 3.2 – Parameters of NVidia GeForce 8600M GT

Parameter	**Value**
Major revision number	1
Minor revision number	1
Total amount of global memory	267,714,560 bytes
Total amount of constant memory	65,536 bytes
Total amount of shared memory per block	16,384 bytes
Total number of registers available per block	8,192
Warp size	32
Maximum number of threads per block	512
Maximum sizes of each dimension of a block	512 x 512 x 64
Maximum sizes of each dimension of a grid	65,535 x 65,535 x 1
Maximum memory pitch	262,144 bytes
Texture alignment	256 bytes
Clock rate	950,000 kilohertz

Listing 3.1 – GPGPU Kernel oriented Δ-stepping code

```
1 memory initialization
2 for ITERATIONS{
3   while (B not empty){
4     labelKernel
5     copyB2SKernel
6     relaxKernel
7   }
8   labelHeavyKernel
9   relaxKernel
10 }
11 copyBackResults
```

operations over received values, but it's not always possible to do it.

The first phase inside *while* loop in line 3 of Listing 3.1 is the *labelKernel*. This kernel is used to decide which will be the unique position of expanded vertices in case of a race condition that two or more vertices want to put the same vertex in the request set R. For instance, if both nodes 2 and 9 are in the current bucket i, and two different threads, respectively with *ids* 2 and 7, try to put the vertex 10 in the request set R they will be labeled differently in the R array. The first phase of the *labelKernel* initializes all R positions to -1 and all R distances to ∞. It must be noted that, from (NVIDIA, 2007d), *"If the instruction executed by a warp writes to the same location in global or shared memory for more than one of the threads of the warp, how many writes occur to that location and the order in which they occur is undefined, but one of the writes is guaranteed to succeed"*.

This way, if both threads try to label vertex 10 with their same tag – which is 4 times *thread id* plus direction, being 0 for down, 1 for right, 2 for up and 3 for left – at least one of them is granted to write their unique tag for the vertex. Supposing the last written value for vertex 10 in Figure 3.2 was from *thread* with $id = 2$, the label for node 10 would be 8. This way, both values would be put in the request set R, parallelly, but each one in the correct slot for each direction. Since node 2 is expanding 10 downwards, its position would be $4 \cdot 8 + 0$ and node 9 would be put in the position $4 \cdot 8 + 1$, since it came from a right edge. As no value was expanded from up and left directions in this instance, they would keep the ∞ value.

Afterwards, another part of *labelKernel* would reduce the four values into a single one, which is the minimum value of them. This way, no duplicates are stored in request set R, which makes it possible to continue the parallel relaxation.

The second kernel *copyB2S* copies all elements from current bucket to the S set. In order to achieve this operation in parallel, atomic instructions will be needed. According to (NVIDIA, 2007d), *"atomicAdd() reads a 32-bit word at some address in global memory, adds an integer to it, and writes the result back to the same address. The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads. In other words, no other thread can access this address until the operation is complete"*. Since some elements from current bucket are void – represented as -1 – the copy kernel needs to check which elements are not void and put them in S array. The position in which each element goes to is defined by an index which is atomically incremented through *atomicAdd()* instruction each time a new value is inserted.

The third kernel is the one which implements the *relaxation* procedure. The first part of the kernel removes vertex in the request set R from their old bucket positions, setting their old value to -1 and atomically decrementing their bucket count. After thread synchronization, values from request set are put in their new bucket positions in the same fashion as they are copied to the S set, which is by an index and *atomicAdd* instruction.

3.2.2 Number of used registers

It must be noted, paying attention to Table 3.1, that there's a limited number of 8192 registers available per multiprocessor. According to (NVIDIA, 2007d), "If the number of

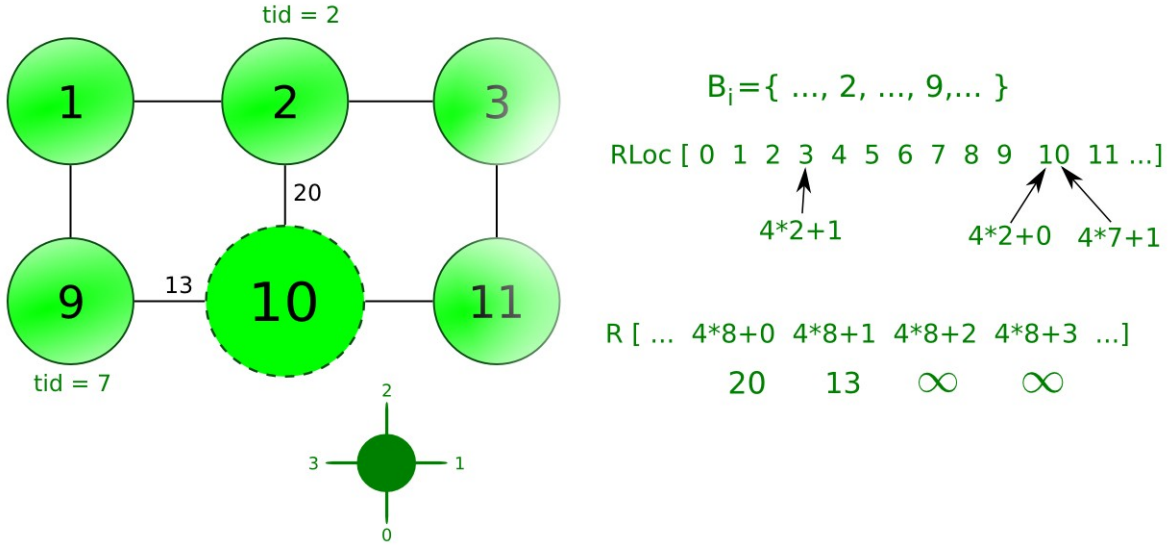


FIGURE 3.2 – Label kernel during a race where vertexes 2 and 9 want to put vertex 10 in the request set R with different distances

registers used per thread multiplied by the number of threads in the block is greater than the total number of registers per multiprocessor, the block cannot be executed and the corresponding kernel will fail to launch". Besides failing, kernels using too much registers cannot be executed by many threads, avoiding memory access latencies being hidden by thread scheduling. This way, programs delay even more. An easy way to find out how many registers are used by each kernel is using the flag *-cubin* with the **nvcc** compiler. This flag, according to **nvcc** online help, *"Compile all .cu/.ptx/.gpu input files to device-only .cubin files"*. These cubin files show important informations about each of the kernels such as their used shared memory, binary code as well as the number of registers used, as seen in Figure 3.3, where *reg* stands for the number of registers, *smem* is the shared memory used and *bincode* is the binary code of the kernel, in gpu assembly. Given the limit of 8192 registers available and that *copyB2SKernel* uses only 9 registers, theoretically 910 threads could be running the same kernel. Nevertheless, there's also a limit of 768 threads that can be run concurrently per multiprocessor, being the upper bound for this kernel. It must be noted that the labeling kernel spends more registers – namely 24 – which constraints it to be run by only 340 threads concurrently.

```
code  {
 name = copyB2SKernel
 lmem = 0
 smem = 40
 reg = 9
 bar = 0
 bincode  {
  0x1000000d 0x0403c780 0x3002c801 0xc4300780
  0x2000cc05 0x04200780 0xd00e0209 0x80c00780
  0xa0000c0d 0x04000780 0x300305fd 0x6400c7c8
  0xa001d003 0000000000 0x1001d003 0x00000280
  0x300fc811 0xc4300780 0x20000611 0x04010780
  0x30020811 0xc4100780 0x2000ca19 0x04210780
  0xa0004211 0x04200780 0x30020815 0xc4100780
  0xd00e0c1d 0x80c00780 0x30800ffd 0x6c4087c8
  0xa0019003 0000000000 0x10019003 0x00000280
  0x1000121d 0x4400c780 0x10018021 0x00000003
  0xd7080e1d 0xe0e00780 0x30020e1d 0xc4100780
  0x2000d021 0x0421c780 0xd00e0c1d 0x80c00780
  0xd00e101d 0xa0c00780 0x20000a19 0x04018782
  0x2000060d 0x04010780 0x300305fd 0x640107c8
  0x1000e003 0x00000280 0xf0000001 0xe0000002
  0x861ffe03 0000000000 0x10008009 0x00000003
  0xd00e0209 0xa0c00780 0x2000ce01 0x04200780
  0xd00e0009 0xa0c00780 0x861ffe03 0000000000
  0xf0000001 0xe0000001
 }
const  {
  segname = const
  segnum = 1
  offset = 0
  bytes = 4
  mem  {
   0xffffffff
  }
 }
}
```

FIGURE 3.3 – Example of a CUDA binary object generated from *copyB2SKernel*

3.2.3 Optimizations

Several attempts can be made to try to decrease execution time. One of them is reading and writing 4 values at the same time, since 128-bit memory transfers are allowed. Another one is using the shared memory more efficiently, since it's almost not being used at all, and accesses are as fast as register accesses.

Another attempt can be merging kernels, so that delays occurred because of kernel calls can be minimized. This is possible if the number of registers of the merged kernel is small enough to yield a large number of concurrent threads. Attempts to decrease the size of the request and bucket sets should also be considered, as long as the time taken can be compensated in future set lookups.

3.2.4 User interface

Main advantages of using GPUs in user interface are that calculation results are already in device's memory, so displaying is faster. Besides that more resources can be used from OpenGL such as better pan and zoom.

As other window objects are also desirable, such as a menu to load the file, and some buttons to adjust parameters or help user interaction, the developed component must be easily extensible with other window objects. A solution for this problem was using Trolltech's Qt Toolkit, since it is a cross-platform application development framework with OpenGL support. Some notable software produced with Qt is Google Earth, which makes heavy use of the OpenGL API as well as Skype, which has the cross-platform feature.

All the OpenGL functionality is available extending the *QGLWidget* class. The main virtual protected function to be reimplemented is *QGLWidget::paintGL()*. In this function, the loaded texture must be bound to a polygon. Besides that, each vertex must map one coordinate of the texture. This is done through code shown in Listing 3.2.

The fact of using Qt makes mouse events easily available, through a simple override. The events *mousePressEvent, mouseMoveEvent,* and *wheelEvent* have been overridden in order to make the loaded image be panned and zoomed. The mouse wheel controls the zoom factor while dragging allows image translation.

Listing 3.2 – OpenGL code to display a texture over a polygon

```
glBegin( GL_QUADS );
glTexCoord2d(0.0,0.0); glVertex2d(0.0,0.0);
glTexCoord2d(1.0,0.0); glVertex2d(1.0,0.0);
glTexCoord2d(1.0,1.0); glVertex2d(1.0,1.0);
glTexCoord2d(0.0,1.0); glVertex2d(0.0,1.0);
glEnd();
```

The effect of zoom is accomplished through the *GL_PROJECTION* matrix mode. A call to the *glOrtho* function with the right parameters is used. In order to pan the image, a translation in the *GL_MODELVIEW* matrix mode is issued. It must be noted that no perspective is applied to the camera in so that the image is not distorted.

In order to compile Qt code with standard *g++* compiler and CUDA code with *nvcc* a workaround was made, since *nvcc* (NVIDIA, 2007d) doesn't support C++. The flag *-c* was added in Qt's Makefile, so that generated code was compiled but not linked. Afterwards, *nvcc* compiler was run over standard CUDA files as well as generated *g++* object files, so that an executable could be created.

3.3 Summary

This chapter dealt with the explanation of key points used in the implementation of *LiveWire* algorithm, as well as the image filtering. Explained points were how to implement Δ-stepping algorithm turning critical sections into atomic operations as well as watching important parameters like the number of used registers. Another point focused in this chapter was the graphical user interface developed with Qt Toolkit. Now that important points of the implementation have been explained, the following chapter shows benchmarked results and comparisons with the CPU.

4 Results and Analysis

As this thesis intends to show that a GPU implementation of the livewire algorithm is possible and also show key features that will enhance and interfere performance, a heavy use of timers and CPU comparisons have been exploited. Main methodology used throughout the thesis has been implementing quite the same algorithm on CPU and comparing performances as well as retrieving important debug information in order to correct GPU programs. Since the algorithm can be divided in several steps, a different approach has been applied to each of the phases. As the algorithm is intended to run on personal computers, the environment for most of the experiments has been a Linux distribution, namely Fedora, which is an open source Linux-based operating system available for download at *http://fedoraproject.org/*.

4.1 Image filtering results

Image filtering is the first step of the algorithm. Figure 4.1 shows the result of a segmentation using the *Sobel* filter. In order to see how filtering works through different architectures and compare them with GPU implementation, the convolution kernel for the Laplacian filter described in Section 2.5.2 was used. Several comparisons have been made with a straightforward CPU implementation, with and without SSE instructions, and a GPU one using Cg language(mentioned in Section 2.3.10). In order to track time in each of the executions, a simple timer like the one shown in Listing 4.1 was used. Another timer readily available on CUDA architecture has also been used, such as *cutCreateTimer* and *cutStartTimer* from *cutil* library.

As lots of details are involved when running the filter, ranging from image loading to memory transfers from CPU to GPU, several parts of the algorithm have been bench-

Listing 4.1 – C timer used for timing filtering executions in each architecture

```
#include <sys/time.h>

static struct timeval start_time;

void tick()
{
  gettimeofday(&start_time, NULL);
}

float tock(int exp)
{
  struct timeval end_time;
  gettimeofday(&end_time, NULL);
  int micros = 1000000 * (end_time.tv_sec - start_time.tv_sec) +
    end_time.tv_usec - start_time.tv_usec;
  int mult = -6 - exp;
  return micros * pow(10, mult);
}
```

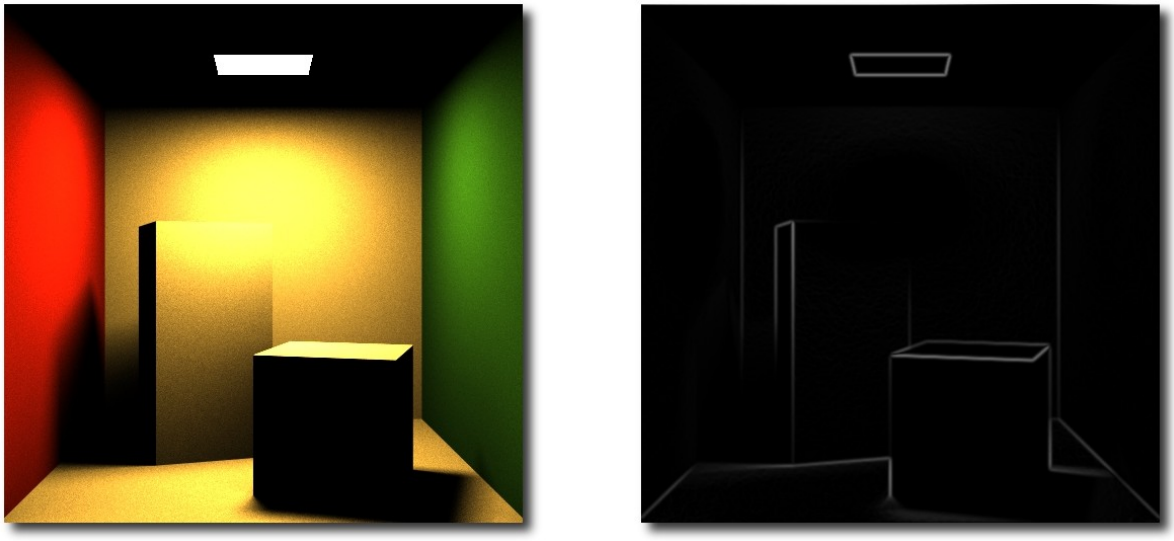
marked separately. Main parts of GPU program that have been timed were: creating a Cg Context, choosing best profile for graphics, loading the fragment program, downloading images, executing the fragment program and uploading the image back to the computer. Other parameters have shown great importance during measurements, such as which hardware has been used – several GeForce models have been tested – as well as the size of the images. Since the same loader has been used in most of the architectures, the time for loading an image to the CPU RAM has not been benchmarked separately.

The first attempt, for implementation comparison, is shown in Table 4.1. The computer's processor used for benchmark was an *Intel Core 2 Duo Centrino T7300*, 2.0 GHz, with 2GB of RAM memory, running Linux Fedora Core 7. In order to count the time elapsed, system function gettimeofday was used (it is declared in sys/time.h). Code was compiled with GNU's *g++* C++ compiler, with optimization flag *-O3*.

TABLE 4.1 – Parameters used in the segmentation of Noiseless images

Processor	**256x256**	**512x512**	**1024x1024**
Centrino Duo T7300	3.3 ms	12.3 ms	51 ms
Centrino Duo T7300 with SSE instructions	2.4 ms	9.6 ms	40 ms

Besides benchmarking default CPU implementation, SSE extensions were also used. This code puts each color channel in the same vector instruction (as well as a spare



(a) Original image (b) Image with *Sobel* filter

FIGURE 4.1 – Comparison of original and filterd images for edge detection

channel, which is not used) making operations over 4 channels per instruction. The theoretical speedup of 3 was not achieved, although the code ran faster. This happens because not all of the operations are arithmetic, as there are several memory lookups and flow control operations. Main advantages of CPU processing is that initialization time is null, since the CPU is already up and the results are stored in main memory.

Table 4.2 shows results for what would be a typical usage of image segmentation and screen displaying for an entire movie – such as a coronary exam in a medical application. The time considered for the application only reads images from CPU memory, transfers them to the GPU and runs the kernel program, displaying results in the framebuffer, without uploading them back to the CPU – what would be expected in case only viewing the results was enough.

TABLE 4.2 – Download and filtering time for 10,000 samples

Processor	**256x256 pixels (ms)**	**512x512 pixels (ms)**	**1024x1024 pixels (ms)**
GeForce 8600M GT	0.40	1.29	5.71
GeForce 6800	0.53	1.6	6.97
GeForce 6600	0.99	3.27	13.07
GeForce 6200	14.7	15.71	65.9

GPGPU code was run over different GPUs in order to watch how much performance

was gained as more pixel shaders were added. Benchmarked GPUs were Nvidia's GeForce 6 series: GeForce 6200 (4 pixel shader processors, 350MHz), GeForce 6600 GT (8 pixel shader processors, 500MHz) and GeForce 6800 Ultra (16 pixel shaders, 400MHz). A mobile version, GeForce8600M GT (32 processors) was also tested. Since image has to be retrieved from computer's main memory, the memory bandwidth should also be observed: GeForce 6200 (5.6 GB/s), GeForce 6600 GT (16.0 GB/s) and GeForce 6800 (35.2GB/s).

It is clear from results in Table 4.2 that the amount of time grows linearly with the number of pixels of the images (since each image is 4 times bigger as well as the processing time).

Running the segmentation thousands of times and taking the mean value may hide some important details about bottlenecks in the program. Separate tests were run for Nvidia GeForce 8600MGT and gave more insight about the application profile, as shown in Table 4.3. Firstly, 95 ms are required for the initialization of the context, through instruction cgCreateContext(). This function creates a Cg context object and returns its handle. According to (NVIDIA, 2007), a Cg context is a container for Cg programs and all Cg programs must be added to a Cg context. Then, the best profile is chosen in 2 ms. Documentation of cgGLGetLatestProfile(NVIDIA, 2007) shows that during this function, extensions are checked to determine the best profile which is supported by the current GPU, driver, and cgGL library combination. Then, it takes around 8 ms to execute function cgGLLoadProgram, which is responsible for preparing the program for binding.

TABLE 4.3 – Major execution delays for GeForce 8600M GT

Task	**Mean time (ms)**
Create Cg Context	95
Choose best profile	2
Load fragment program	8
Download 1024x1024 image	5.6
Execute fragment program	0.06
Upload 1024x1024 image	6.5

Since preparing Cg Context is required only once per application, it is clear from Table 4.3 that the greatest bottlenecks of this program are memory transfers as it takes 5.6 ms for downloading the image to the GPU and 6.5 ms for uploading (although the segmented images are not required to be uploaded in case they are not going to be later

recorded in the hard disk, as it might be enough to only display the results, depending on the application). This way, one can conclude that the speedup caused by using different models in Table 4.2 is due to their different bandwidths. Tables 4.4 and 4.5 show download and upload times for different image sizes. As it can be seen in Table 4.3, this application is extremely light in terms of computation, since the computational kernel only looks up and multiply nine pixel values for each pixel, which is linear for the size of the image.

TABLE 4.4 – Memory transfer delay during download

Image size(pixels)	**Mean time (ms)**
256 x 256	0.3
512 x 512	1.2
1024 x 1024	5.7

In order to download images to GPU, the function glTexImage2D was used, since the image was stored in framebuffer objects. In case it was needed to record the processed images, they would have to be uploaded to the CPU memory. This is accomplished through the function glReadPixels, used to create Table 4.5.

TABLE 4.5 – Memory transfer delay during upload

Image size(pixels)	**Mean time (ms)**
256 x 256	0.5
512 x 512	1.8
1024 x 1024	6.5

4.2 Shortest path algorithm results

The single source shortest path algorithm is the main bottleneck for livewire application so it has received special focus in benchmarking. As Dijkstra's implementation with a binary heap yields great performance results, an STL C++ implementation has been made not only for timing comparisons but also correctness of the other implementations, working as a gold standard.

Besides implementing Dijkstra's algorithm, Δ-stepping has also been implemented on the CPU in a single-threaded and multi-threaded version. The purpose of implementing a multi-threaded version is to use a double core Intel processor so that speed-up and thread

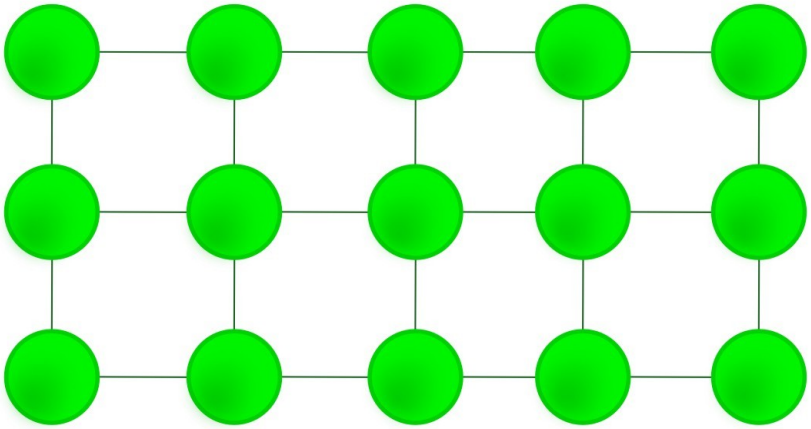


FIGURE 4.2 – A 5 by 3 grid graph

synchronizing issues can be watched. The multi-threaded version works just like the GPU version – simultaneous threads can relax the queue at the same time. Implemented threads were of type POSIX Threads, also known as PThreads which are thread implementations that adhere to IEEE POSIX 1003.1c standard. In order to use such threads, a header include library is necessary (*pthread.h*) as well as a linking flag on *g++* (*-lpthreads*). All this flexibility has given useful tools to debug and profile GPU implementation.

As the 9th Dimacs Implementation Challenge was focused on shortest path problems, a handful of tools have been made available for download through the competition site [1]. In order to generate grid graphs for testing, the Randgraph tool has been used. This is a command line tool to generate graphs. If the following command is given,

```
./Randgraph new.graph 262144 13 1
```

a file *new.graph* is generated, with $512 \cdot 512$ nodes (262144). The parameter 13 tells that the graph is of grid type and the parameter 1 states that it is square. The graph description is given in the format source node, destination node and edge weight. In the generated graph, weights were uniformly distributed. Figure 4.2 shows a rectangular grid graph.

Several parameters can be varied in the Δ-stepping algorithm, such as Δ's value itself, the size of the buckets and the number of iterations. Different types of graphs are also encouraged, although livewire mostly deals with the grid graph.

Other parameters linked to CUDA API can be tested, such as the number of threads

[1] 9th Dimacs Implementation Challenge download site: http://www.dis.uniroma1.it/~challenge9/download.shtml

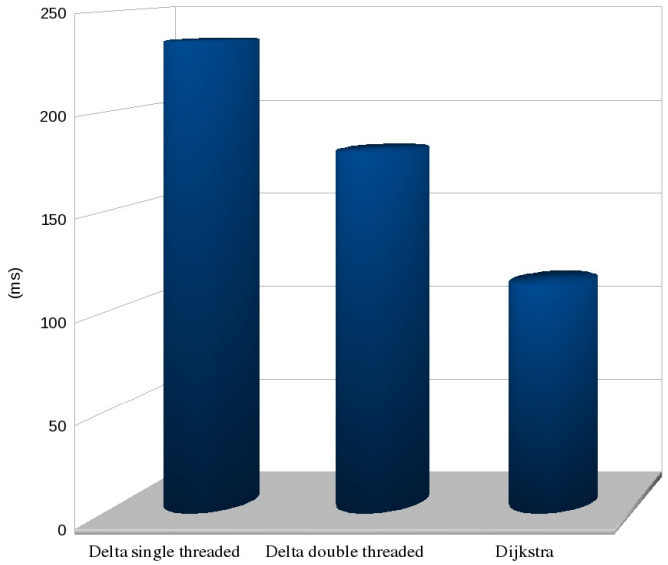


FIGURE 4.3 – Execution times for different shortest path algorithms solving a 512x512 grid graph on the CPU

and the size of grids. Different implementations can also be tested, using shared memory. A detailed comparison and benchmarks of specific parts of the code can give great insight about application bottlenecks as well as points to be optimized.

4.2.1 CPU approach

For the generated algorithms, firstly, a small comparison with Dijkstra, delta-stepping with and without multi-threads can be seen in Figure 4.3. The graph used is a 512x512 grid graph with edges ranging from 0.0 to 1.0 in a uniform distribution. It must be noted that for Δ-stepping, an initialization phase for creating buckets has been executed before, taking around 600 ms. This phase only creates STL vectors separating light and heavy edges, which can be done faster if C arrays are used, as well as adding an *if* statement during edge analysis.

From Figure 4.3 one can conclude that the Δ-stepping algorithm is slower than Dijkstra's but it can be scaled. Using *pthreads* also adds some overhead in thread creation. Measured overhead shows 35 ms for creating threads in order to process the request set R.

4.2.2 GPU initialization

GPU implementation must be analyzed carefully. The first detail is that its initialization takes a while. According to (NVIDIA, 2007d), *"There is no explicit initialization function for the runtime API; it initializes the first time a runtime function is called. One needs to keep this in mind when timing runtime function calls and when interpreting the error code from the first call into the runtime"*. This way, a couple tests have been made to interpret initialization. Figure 4.4 shows a simple program used to benchmark initialization time. The output of the program, after being run in a *GeForce 8600M GT*, under Linux, is seen in Table 4.6. It is clear from the results that the timing of the first call to the *emptyKernel* is also initializing CUDA runtime. Another important fact from the results is that a lower bound for algorithms that need to call any kernel has been found. It will take at least 40 μs per call.

TABLE 4.6 – Initialization benchmark

Call	**Time (ms)**
First	181.980
Second	0.039

4.2.3 GPU thread study

Besides considering initialization times, the number of threads executing the kernels has been analyzed. From Figure 4.5 one can conclude that the more threads are executed the faster the algorithm is. It must be noted, though, that the number of threads is only a parameter passed on kernel invocations and that it does not reflect the fact of having more hardware running more threads. It simply states that the thread scheduler will be able to process more threads during its time slices. The main advantage of increasing the number of threads is that it hides latencies for accessing device memory. As stated in (NVIDIA, 2007d), *"A multiprocessor takes 4 clock cycles to issue one memory instruction for a warp. When accessing global memory, there are, in addition, 400 to 600 clock cycles of memory latency"*. As the multiprocessor can switch warps between memory requests, latency can be hidden to a point.

Figure 4.6 shows each part of kernel execution in detail. It's clear from the figure that

```
#include <stdio.h>
#include <cutil.h>

__global__ void
emptyKernel(){
}

int main( int argc, char** argv)
{
    dim3  grid( 1, 1, 1);
    dim3  threads( 512, 1, 1);

    unsigned int nvtimer = 0;
    cutCreateTimer( &nvtimer);
    cutStartTimer( nvtimer);

    emptyKernel    <<<grid, threads >>> ();
    cudaThreadSynchronize();

    cutStopTimer( nvtimer);
    printf("First  call: %10.5fms\n", cutGetTimerValue(nvtimer));
    cutResetTimer( nvtimer);

    cutStartTimer( nvtimer);
    emptyKernel    <<<grid, threads >>> ();
    cudaThreadSynchronize();

    cutStopTimer( nvtimer);
    printf("Second call: %10.5fms\n", cutGetTimerValue(nvtimer));

    CUT_EXIT(argc, argv);
}
```

FIGURE 4.4 – Simple CUDA code for runtime initialization and benchmark

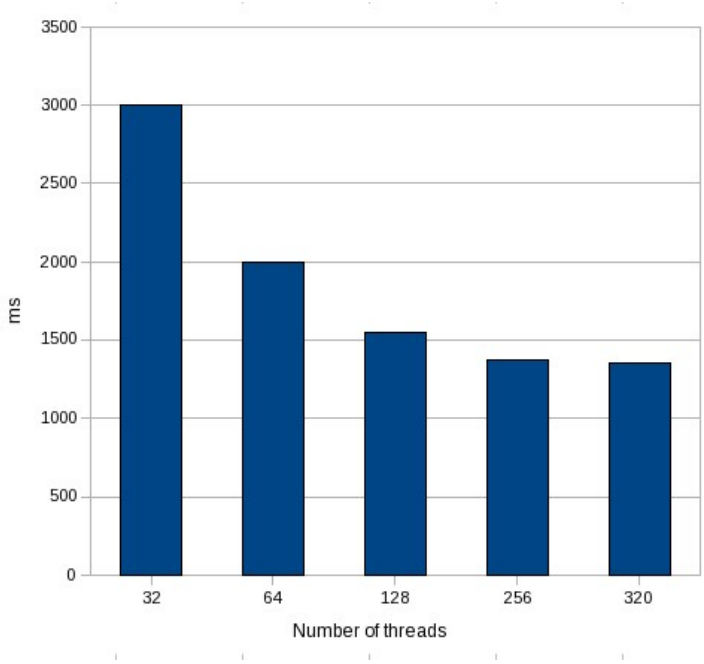


FIGURE 4.5 – Δ-stepping algorithm execution time on GPU with different number of threads for a 512x512 nodes graph

labeling and relaxing kernels scale better than copying kernel. Analyzing copying kernel code, it is heavily dependent on reading and writing to device memory, as well as on the *atomicAdd* instruction. Since almost no processing can be done while waiting for memory accesses, no speedup can be seen while trying to hide memory latency through thread scheduling. On the other hand, the other algorithms have some reasonable speedup up to 128 threads. Associations to the size of nodes in queue can provide better insight why further scaling has not been observed.

4.2.4 Number of light edge requests

Another interesting data is the number of light requests in each phase, which would be the number of nodes that are actually being parallelly relaxed. The pattern shows that, when entering a new bucket, a great number of nodes are in the set and, as the bucket is being relaxed, this number tends to decrease. Figure 4.7 clearly shows this behavior. This graph is related to a shortest path request to the upper left vertex. As the number of graphs in the request set is increasing, more neighbor nodes can be explored, yielding a fast growth up to the phase 1400. After that, the expansion has found the graph edges located at the bottom and at the right side. This figure shows that substantial parallelism

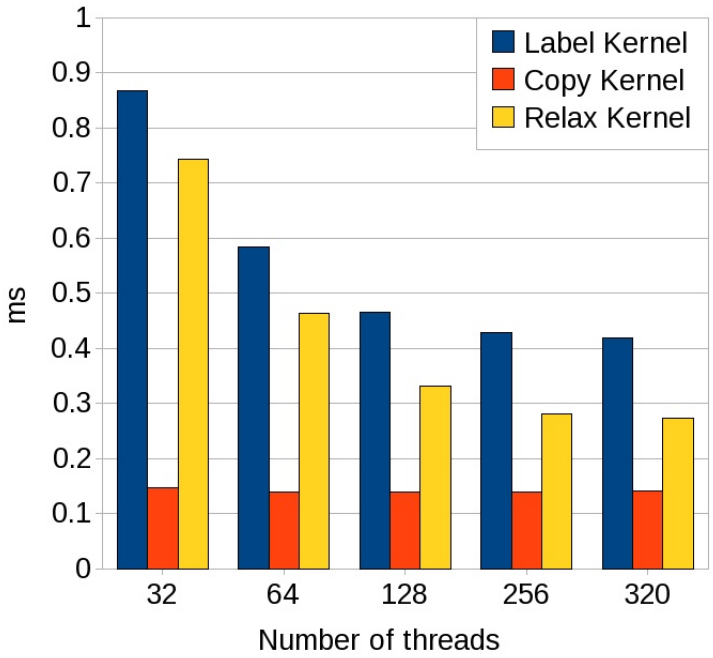


FIGURE 4.6 – Mean execution times for 1966 calls of each kernel running the Δ-stepping algorithm on GPU

can be found in higher phases.

On the other hand, this pattern generates phases with small amounts of nodes – 38.35% of the phases have 32 or less nodes in the request set. Another data that can partially explain why increasing threads above 256 does not produce much speedup is that the average size of the request set is 247.32. More information about frequency distribution can be seen in Figure 4.8, detailing request set sizes up to 512.

4.2.5 Varying starting position

Depending on the starting position, the maximum distance found in the grid graph can be very different. For instance, if a node at the upper left corner is chosen as the starting node, which is, with zero distance, in an uniform distribution of edges, the node that will most likely have the farthest distance will be located at the bottom right of the graph. Notwithstanding the long distance, some starting positions might deliver substantial parallel workload. Some tests have been made changing the starting position of the search from the first node to a node in the middle of the graph. Results can be seen in Table 4.7. It is clear from the table that nodes that have smaller maximum distances present lower processing time. As the expected work is $O(dn)$ – where d is the maximum

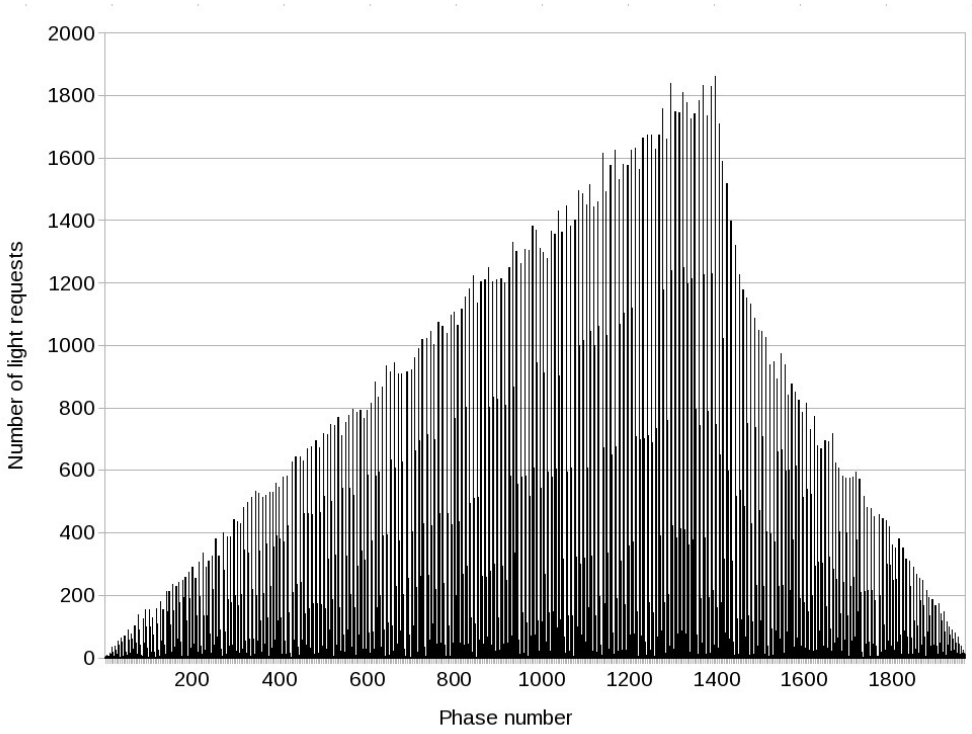


FIGURE 4.7 – Number of nodes in the request set during the light edge relaxation phase in a 512x512 grid graph

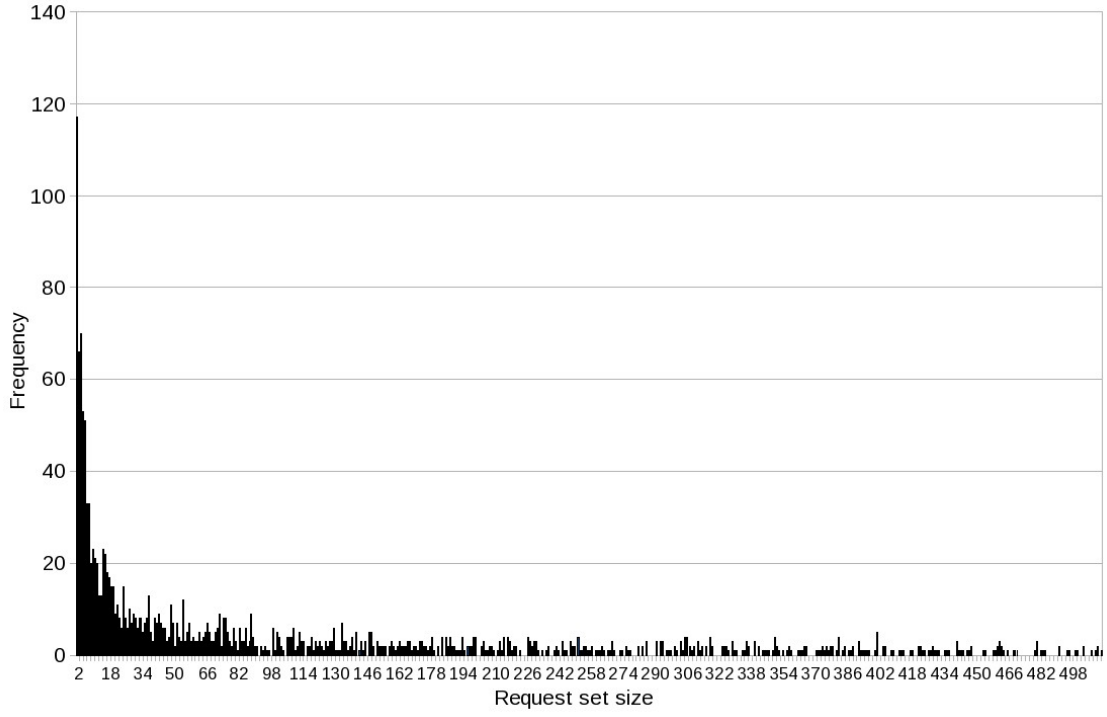


FIGURE 4.8 – Frequency distribution of the request set size for a 512x512 grid graph single source shortest path request of the upper left node

degree of the graph – more parallelism might have been found while expanding nodes from the center of the graph.

TABLE 4.7 – Mean time and maximum distance found while executing the Δ-stepping algorithm on the GPU for different starting positions

Type of graph	Starting position	Mean time (ms)	Maximum distance
256x256	upper left	488.35	119.71
256x256	center	345.16	61.15
512x512	upper left	1425.28	235.4
512x512	center	1089.93	80.94
1024x1024	upper left	4301.37	314.62
1024x1024	center	3629.6	159.36

4.2.6 Different edge distributions

While using 9th DIMACS Challenge application to generate the graph with uniform edges, other options are available, like the normal distribution as well. Another performance test was executed comparing both uniform and normal distributions. The result can be seen in Table 4.8. From this table we can see that both distributions yield near execution time being faster the ones with shortest maximum distances.

TABLE 4.8 – Mean time and maximum distance found while executing the Δ-stepping algorithm on the GPU for different edge weight distributions

Type of graph	Distribution	Mean time (ms)	Maximum distance
256x256	uniform	488.35	119.71
256x256	normal (mean 0.5)	512.8	120.03
512x512	uniform	1425.28	235.4
512x512	normal (mean 0.5)	1337.64	154.98

4.2.7 Comparing with CPU solutions

Throughout all the research for the thesis, CPU solutions have been used to compare performance and debug GPU kernels. One of the intended goals during the research was

to compare how fast could GPU approach be compared to standard CPU solutions. Table 4.9 shows times for each type of graph. Dijkstra's implementation was the one done with STL priority queues, while CPU Δ-stepping is the sequential version. It must be noted that Dijkstra's implementation is supposed to be $O(mlogn)$ – for m edges and n nodes – and Δ-stepping has $O(dn)$ expected work, where d is the graph's maximum degree. Figure 4.9 shows better the relation seen in Table 4.9 with a logarithm scale. From this figure, it's clear that the expected time from Dijkstra is going to be longer for graphs with more nodes than the ones in the figure. Besides that, calculated slope for Δ-stepping running on the GPU is about 74% of Dijkstra's. This shows that expected performance for GPU Δ-stepping is to be faster than Dijkstra's in wider graphs. Although tests with graphs that have 2048x2048 pixels were desired in order to see whether GPU performance could outperform CPU approaches, there was not enough memory on targeted GeForce GPU. This way, it couldn't be tested.

TABLE 4.9 – Comparison of single source shortest path algorithms for CPU and GPU with different graph sizes

Algorithm	**256x256 (ms)**	**512x512 (ms)**	**1024x1024 (ms)**
GPU Δ-stepping	488.35	1425.28	4301.37
CPU Δ-stepping	222.45	843.96	3317.09
Dijkstra	27.39	118.08	515.07

4.3 Graphical user interface

As discussed in 3.2.4, an OpenGL interface has been developed for user interface. This interface is basically used to retrieve points to start the single source shortest path algorithm and to display results to the user. As explained before, the same component used to render the two-dimensional texture can be used to draw tridimensional models, as can be seen in the prototype of Figure 4.10.

Some more functionality can be seen in Figure 4.11, in which zoom and pan have been applied, as well as texture and image loading. The highlighted yellow line shows the result of the segmentation given a starting point and the position the mouse was over during the screen capture.

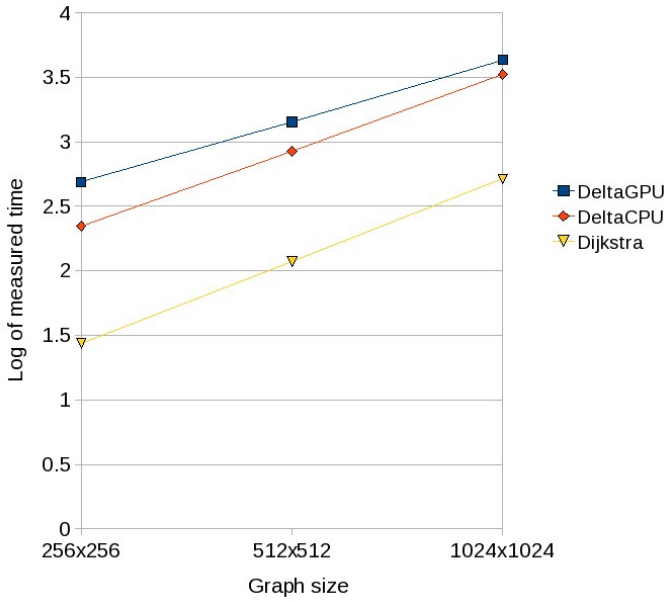


FIGURE 4.9 – Logarithm of measured time displayed in Table 4.9 for CPU and GPU shortest path algorithms

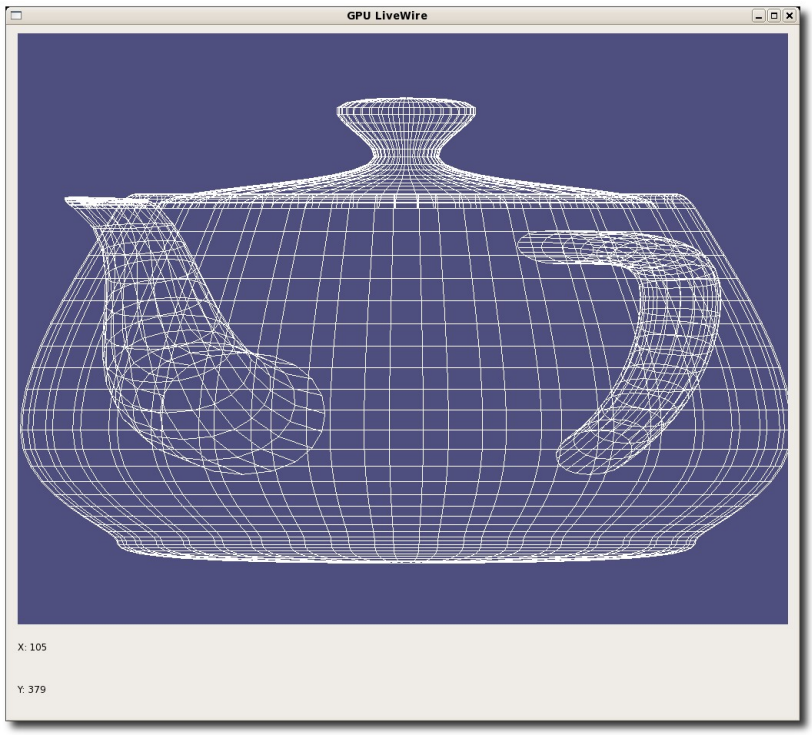


FIGURE 4.10 – 3D model drawn in the same component as 2D images are loaded

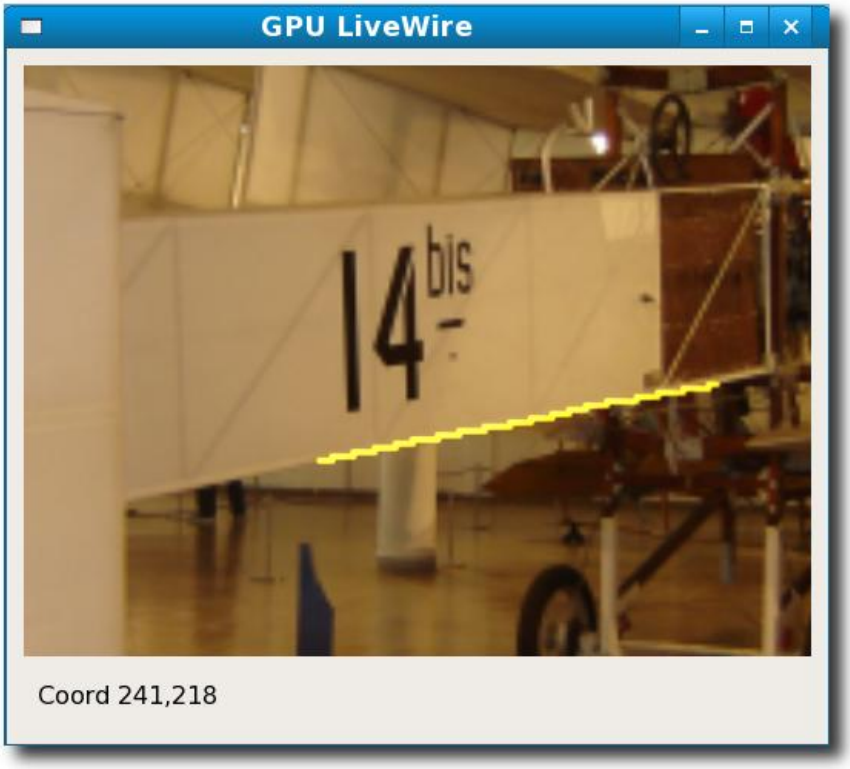


FIGURE 4.11 – Graphical user interface for determining start point for the single source shortest path algorithm. The highlighted yellow line shows the result of the segmentation

Another function that is already available from OpenGL rendering pipeline is image filtering when zooming to pixel levels as can be seen in a comparison with GIMP 2.2 loading the same picture in Figure 4.12.

4.4 Code availability

Open source code has been made available through *Google Code* project hosted at the address: *http://code.google.com/p/gpuwire/*

4.5 Summary

This chapter has given important information about GPU *LiveWire* algorithm performance. The first part has shown that filtering can run extremely fast on GPUs but several steps must be considered carefully, such as the time to upload and download images. Another delay observed in GPU architectures is the initialization time. This way, depending

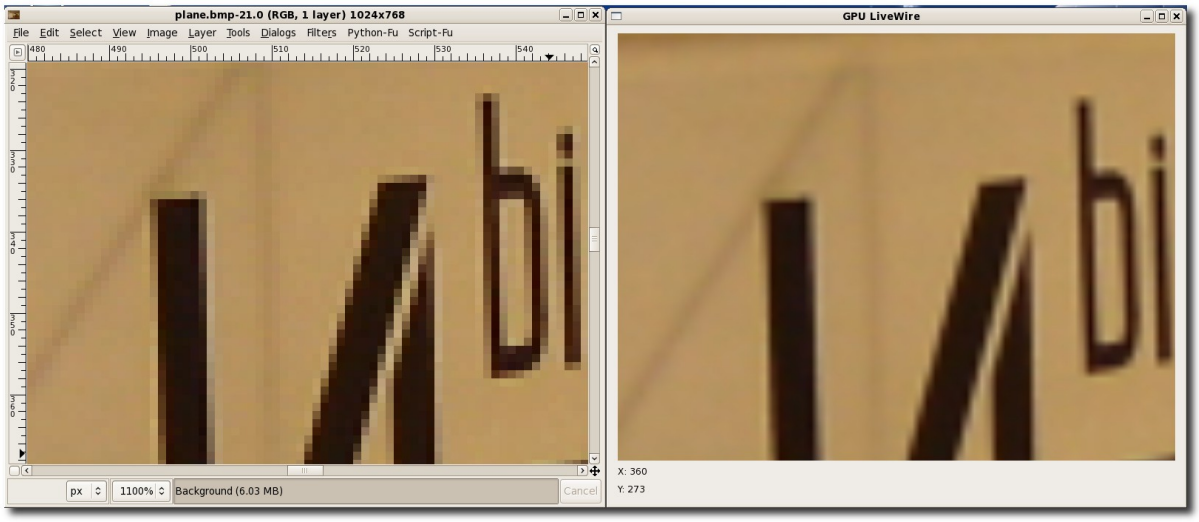


FIGURE 4.12 – Comparison of filtered zoom of the developed software on the right and GIMP's on the left side

on how many times the image is going to be filtered or how large it is, it might not be interesting to wait for GPU initialization, since CPU might already be up and running.

The Δ-stepping part of the algorithm has also some interesting points. Firstly, the amount of parallelism available for the problem increases with the phase number. Initially, very few nodes can be expanded parallelly. Besides that, this algorithm depends heavily on memory lookups, which might stall it in case no processing can be made while waiting for data to be retrieved.

5 Conclusion

This thesis has proved that the livewire algorithm can be implemented in a GPU and several parts of it run faster than a CPU implementation. Throughout the whole thesis, key points of data stream programming have been shown as well as how much benefit can and cannot be retrieved from GPGPU.

Another conclusion from this thesis was that new algorithms must be developed in order to fully exploit the potential of new parallel architectures. It also shows that in the design of new algorithms, several aspects that are not usually considered, must be focused such as memory access pattern and parallelization of data structures.

The main advantages of using *LiveWire* algorithm on GPU are that CPU processing is relieved, a faster filtering time is achieved and the results are already on GPU, which can give the user better interfacing options, such as zooming faster and exploiting other OpenGL benefits. Benchmarked results show that larger images will present more parallelism and extrapolation of Figure 4.9 indicate faster performance in GPUs for these images.

Besides showing an application that works on GPU, giving insights about how processing works better on GPU – such as register counting, key memory access patterns and general data parallel programming – the thesis also shows what types of algorithms might not get full advantages out of the GPU. The application developed has also been able to fully utilize NVidia's G80 CUDA architecture features such as atomic functions as well as memory scattering, justifying the use of such technology.

5.1 Summary of contributions

In this thesis, a complete program has been delivered to the Open Source community, capable of loading an image, applying a filter, generating a graph and calculating the single source shortest path through a GPU approach. In the same software, the user has been able to use fully fledged features of OpenGL to interact with the image and with CUDA.

Although most of the implementation of *LiveWire* in a GPU is resumed to exchange Dijkstra's algorithm for Δ-stepping, this is not a trivial operation, since GPU architecture is very different from CPU. The development of kernels that support the use of critical sections through atomic instructions is also a valuable contribution of this thesis.

Besides the software gain, this thesis contributes with statistical data and benchmark of specific parts of the code, showing which parts can be enhanced in order to get an even faster implementation of such an important algorithm.

5.2 Future research

It has been seen throughout the implementation of the *Livewire* algorithm in this thesis that although CUDA API has lots of features that are required for the application, it lacks a lot of productivity and debugging tools. Although device emulation is provided, it does not fully represent exactly what happens when the device execution itself takes place. Besides that, implementation is not only restricted to GPUs but also only to G80 series. An useful future work would be porting GPU Δ-stepping algorithm to other API's such as Brook or RapidMind, which could maybe show more productivity as well as more portability.

Besides testing other API's it would also be useful to test other approaches for memory accessing during kernel execution, trying to benefit even more from shared memory, decreasing memory access time as well as making attempts using completely different data structures.

Some useful approaches that could be tested in order to increase code efficiency would be:

- Do more computation on the GPU to avoid costly transfers;
- Optimize memory access coherence;
- Optimize for spatial locality in cached texture memory;
- Decrease the number of registers used so that more threads can be run simultaneously

Bibliography

ABDOU, I.; PRATT, W. K. Quantitative design and evaluation of enhancement/thresholding edge detectors. **Proceedings of the IEEE**, p. 753–763, 1979.

ADAMSOM, P.; TICK, E. Greedy partitioned algorithms for the shortest-path problem. **International Journal of Parallel Programming**, v. 20, n. 4, August 1991.

ALVERSON, R.; CALLAHAN, D.; CUMMINGS, D.; KOBLENZ, B.; PORTERFIELD, A.; SMITH, B. The Tera computer system. In: **Proceedings of the1990 International Conference on Supercomputing**. [s.n.], 1990. p. 1–6. Disponível em: <citeseer.ist.psu.edu/alverson90tera.html>.

AMD. **ATI CTM Guide**. [S.l.], 2006.

AN, P.; JULA, A.; RUS, S.; SAUNDERS, S.; SMITH, T.; TANASE, G.; THOMAS, N.; AMATO, N.; RAUCHWERGER, L. Stapl: An adaptive, generic parallel c++ library. In: DIETZ, H. G. (Ed.). **LCPC**. [S.l.]: Springer, 2003. (Lecture Notes in Computer Science, v. 2624), p. 193 – 208.

ANDERSON, W.; BRIGGS, P.; HELLBERG, C. S.; HESS, D. W.; KHOKHLOV, A.; LANZAGORTA, M.; ROSENBERG, R. Early experience with scientific programs on the cray mta-2. In: **SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing**. Washington, DC, USA: IEEE Computer Society, 2003. p. 46. ISBN 1-58113-695-1.

BADER, D. A.; CONG, G.; FEO, J. On the architectural requirements for efficient execution of graph algorithms. **International Conference on Parallel Processing**, p. 547–556, June 2005.

BADER, D. A.; ILLENDULA, A. K.; MORET, B. M. E.; WEISSE-BERNSTEIN, N. R. Using PRAM algorithms on a uniform-memory-access shared-memory architecture. **WAE 2001**, v. 2141, p. 129–144, 2001. Disponível em: <citeseer.ist.psu.edu/article/bader01using.html>.

BADER, D. A.; MADDURI, K. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. In: **ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing**. Washington, DC, USA: IEEE Computer Society, 2006. p. 523–530. ISBN 0-7695-2636-5.

BAGGIO, D. L. **Intravascular Ultra Sound Image Segmentation Algorithms**. 2006.

BERRY, J.; HENDRICKSON, B.; KAHAN, S.; KONECNY., P. Graph software development and performance on the mta-2 and eldorado. **Cray User Group meeting (CUG 2006)**, Lugano, Switzerland, May 2006.

BLINN, J. F. Models of light reflection for computer synthesized pictures. In: GEORGE, J. (Ed.). [S.l.: s.n.], 1977. v. 11, n. 2, p. 192–198.

BRADSKI, G. R.; KAEHLER, A.; PISAREVSKY, V. Learning-based computer vision with intel's open source computer vision library. **Intel Technology Journal**, 2005.

BRODAL, G. S.; TRAFF, J. L.; ZAROLIAGIS, C. D. A parallel priority queue with constant time operations. **Journal of Parallel and Distributed Computing**, v. 49, n. 1, p. 4–21, 1998. Disponível em: <citeseer.ist.psu.edu/article/brodal97parallel.html>.

BRODTKORB, A. R. **A MATLAB interface to the GPU**. Dissertação (Mestrado) — Universitas Osloensis, May 2007.

BUCK, I.; FOLEY, T.; HORN, D.; SUGERMAN, J.; FATAHALIAN, K.; HOUSTON, M.; HANRAHAN, P. Brook for gpus: stream computing on graphics hardware. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 23, n. 3, p. 777–786, 2004. ISSN 0730-0301.

BUTTARI, A.; LUSZCZEK, P.; KURZAK, J.; DONGARRA, J.; BOSILCA, G. Scop3, a rough guide to scientific computin on the playstation3. Innovative Computing Laboratory, University of Tennessee Knoxville. May 2007.

CHEN, T.; RAGHAVAN, R.; DALE, J. N.; IWATA, E. Cell broadband engine architecture and its first implementation – a performance view. **IBM Journal of Research and Development**, v. 51, n. 5, 2007.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to Algorithms**. [S.l.]: MIT Press and McGraw-Hill, 2001.

Da Gracca, G.; DEFOUR, D. **Implementation of float-float operators on graphics hardware**. 2006. Disponível em: <http://www.citebase.org/abstract?id=oai:arXiv.org:cs/0603115>.

DIJKSTRA, E. W. A note on two problems in connexion with graphs. **Numerische Mathematik**, v. 1, p. 269–270, 1959.

DONGARRA, J.; WASŚNIEWSKI, J. High performance linear algebra package lapack90. **Lecture Notes in Computer Science**, Springer, p. 387–391, 1998.

EKMAN, M.; WARG, F.; NILSSON, J. An in-depth look at computer performance growth. **ACM SIGARCH Computer Architecture News**, 2005.

FAIRCHILD, M. D. **Color Appearance Models**. Boston, MA: Addison-Wesley, 1998.

FALCÃO, A. X. **Paradigmas de Segmentação de Imagens Guiada pelo Usuário: Live-Wire, Live-Lane e 3D-Live-Wire**. Tese (Doutorado) — Universidade Estadual de Campinas, 1997.

FARRUGIA, J.-P.; HORAIN, P.; GUEHENNEUX, E.; ALUSSE, Y. Gpucv: A framework for image processing acceleration with graphics processors. In: **ICME**. [S.l.]: IEEE, 2006. p. 585–588.

GOLDBERG, A. V. A simple shortest path algorithm with linear average time. **Lecture Notes in Computer Science**, v. 2161, 2001. Disponível em: <citeseer.ist.psu.edu/640528.html>.

GREGOR, D.; LUMSDAINE, A. Lifting sequential graph algorithms for distributed-memory parallel computation. In: **OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications**. New York, NY, USA: ACM, 2005. p. 423–437. ISBN 1-59593-031-0.

HARRIS, M. Mapping computational concepts to gpus. In: PHARR, M. (Ed.). **GPU Gems 2**. [S.l.]: Addison-Wesley, 2005. cap. 31, p. 493–508. ISBN 0321335597.

HENSLEY, J. Close to the metal. Siggraph 2007 GPGPU Course slides. 2007.

HRIBAR, M.; TAYLOR, V.; BOYCE, D. **Performance study of parallel shortest path algorithms: Characteristics of good decompositions**. 1997. Disponível em: <citeseer.ist.psu.edu/article/hribar97performance.html>.

HRIBAR, M.; TAYLOR, V. E.; BOYCE, D. E. **Reducing the Idle Time of Parallel Shortest Path Algorithms**. 1998. Disponível em: <citeseer.ist.psu.edu/hribar98reducing.html>.

INTEL. **Tutorial**. [S.l.], December 2007. Accessed 2007-01-15. Disponível em: <http://threadingbuildingblocks.org/uploads/81/91/Latest Open Source Documentation/Tutorial (Open Source).pdf>.

INTEL. **Getting Started Guide**. [S.l.], January 2008. Accessed 2007-01-15. Disponível em: <http://threadingbuildingblocks.org/uploads/81/91/Latest Open Source Documentation/Getting Started Guide (Open Source).pdf>.

JONES, M. T. Linux and symmetric multiprocessing. Accessed 2008-01-15. March 2007. Disponível em: <http://www.ibm.com/developerworks/ library/l-linux-smp/>.

KESSENICH, J. **The OpenGL Shading Language**. [S.l.], September 2006.

KLEIN, P. N.; SUBRAMANIAN, S. A randomized parallel algorithm for single-source shortest paths. **J. Algorithms**, v. 25, n. 2, p. 205–220, 1997. Disponível em: <citeseer.ist.psu.edu/article/klein94randomized.html>.

LAWSON, C. L.; HANSON, R. J.; KINCAID, D. R.; KROGH, F. T. Basic linear algebra subprograms for fortran usage. **ACM Trans. Math. Softw.**, ACM, New York, NY, USA, v. 5, n. 3, p. 308–323, 1979. ISSN 0098-3500.

LEFOHN, A.; KNISS, J. M.; OWENS, J. D. Implementing efficient parallel data structures on gpus. In: PHARR, M. (Ed.). **GPU Gems 2**. [S.l.]: Addison-Wesley, 2005. p. 521–545. ISBN 0321335597.

MADDURI, K. **9th DIMACS implementation challenge: Shortest Paths Δ-stepping C/MTA-2 code**. 2006. Disponível em: <http://www.cc.gatech.edu/~kamesh/research/DIMACS-ch9>.

MADDURI, K.; BADER, D.; BERRY, J.; CROBAK, J. Parallel shortest path algorithms for solving large-scale instances. **9th DIMACS Implementation Challenge – The Shortest Path Problem**, 2006.

MCCOOL, M.; TOIT, S. D.; POPA, T.; CHAN, B.; MOULE, K. Shader algebra. In: **SIGGRAPH '04: ACM SIGGRAPH 2004 Papers**. New York, NY, USA: ACM, 2004. p. 787–795.

MCCOOL, M. D. Data-parallel programming on the cell be and the gpu using the rapidmind development platform. In: **GSPx Multicore Applications Conference**. Santa Clara: [s.n.], 2006.

MCCOOL, M. D.; QIN, Z.; POPA, T. S. Shader metaprogramming. In: **HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware**. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2002. p. 57–68. ISBN 1-58113-580-7.

MEYER, U. **Design and Analysis of Sequential and Parallel Single-Source Shortest-Paths Algorithms**. Tese (Doutorado) — Universität des Saarlandes, October 2002.

MEYER, U. Average-case complexity of single-source shortest-paths algorithms: lower and upper bounds. **J. Algorithms**, Academic Press, Inc., Duluth, MN, USA, v. 48, n. 1, p. 91–134, 2003. ISSN 0196-6774.

MEYER, U.; SANDERS, P. δ-steppping: a parallelizable shortest path algorithm. **Journal of Algorithms**, v. 49, n. 1, p. 114–152, October 2003.

MILLER, R.; BOXER, L. **A unified approach to sequential and parallel algorithms**. Upper Saddle River, NJ: Prentice Hall, 2000.

MORTENSEN, E. N.; BARRETT, W. A. Intelligent scissors for image composition. In: **SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques**. New York, NY, USA: ACM Press, 1995. p. 191–198. ISBN 0-89791-701-4.

MORTENSEN, E. N.; BARRETT, W. A. Interactive segmentation with intelligent scissors. **Graph. Models Image Process.**, v. 60, n. 5, p. 349–384, 1998.

NVIDIA. **Cg Documentation**. September 2007.

NVIDIA. Cublas library. NVIDIA Corporation, Santa Clara, CA, 2007.

NVIDIA. Cufft library. NVIDIA Corporation, Santa Clara, CA, 2007.

NVIDIA. **GeForce 8800 specifications**. 2007. Http://www.nvidia.com/page/geforce_8800.html.

NVIDIA. Nvidia cuda programming guide. NVIDIA Corporation, Santa Clara, CA, June 2007.

OWENS, J. Gpu gems 2: Programming techniques for high-performance graphics and general-purpose computation. In: _____. [S.l.]: Addison Wesley, 2005. cap. 29, p. 457–470.

OWENS, J. D.; LUEBKE, D.; GOVINDARAJU, N.; HARRIS, M.; KRÜGER, J.; LEFOHN, A. E.; PURCELL, T. J. A survey of general-purpose computation on graphics hardware. **Computer Graphics Forum**, 2007.

PAPAEFTHYMIOU, M.; RODRIGUE, J. **Implementing parallel shortest-paths algorithms**. 1994. Disponível em: <citeseer.ist.psu.edu/papaefthymiou94implementing.html>.

PORTER, T.; DUFF, T. Compositing digital images. In: **SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques**. New York, NY, USA: ACM, 1984. p. 253–259. ISBN 0-89791-138-5.

PUTZE, F.; SANDERS, P.; SINGLER, J. Mcstl: the multi-core standard template library. In: **PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming**. New York, NY, USA: ACM, 2007. p. 144–145. ISBN 9781595936028. Disponível em: <http://portal.acm.org/citation.cfm?id=1229428.1229458>.

QUINN, M. J. **Parallel Computing: theory and practice**. Columbus, OH: McGraw-Hill, 1994.

SHI, H.; SPENCER, T. H. Time-work tradeoffs of the single-source shortest paths problem. **J. Algorithms**, Academic Press, Inc., Duluth, MN, USA, v. 30, n. 1, p. 19–32, 1999. ISSN 0196-6774.

SMITH, K. V. **Accelerating MATLAB with CUDA using MEX files**. Santa Clara, CA, September 2007.

SUTTER, H. The free lunch is over: A fundamental turn toward concurrency in software. **Dr. Dobb's Journal**, 2005.

TARDITI, D.; PURI, S.; OGLESBY, J. Accelerator: using data parallelism to program gpus for general-purpose uses. **SIGOPS Oper. Syst. Rev.**, ACM, New York, NY, USA, v. 40, n. 5, p. 325–335, 2006. ISSN 0163-5980.

THORUP, M. Undirected single-source shortest paths with positive integer weights in linear time. **Journal of the ACM**, 1999.

ULLMAN, J.; YANNAKAKIS, M. High-probability parallel transitive closure algorithms. In: **SPAA '90: Proceedings of the second annual ACM symposium on Parallel algorithms and architectures**. New York, NY, USA: ACM, 1990. p. 200–209. ISBN 0-89791-370-1.

UPSTILL, S. **The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics**. [S.l.]: Addison-Wesley Professional, 1990.

WHALEY, R. C.; DONGARRA, J. J. Automatically tuned linear algebra software. In: **Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)**. Washington, DC, USA: IEEE Computer Society, 1998. p. 1–27. ISBN 0-89791-984-X.

WHITAKER, J. (Ed.). **The Electronics Handbook**. [S.l.]: CRC-Press, 1996.

WILLIAMS, S.; SHALF, J.; OLIKER, L.; KAMIL, S.; HUSBANDS, P.; YELICK, K. The potential of the cell processor for scientific computing. In: **CF '06: Proceedings of the 3rd conference on Computing frontiers**. New York, NY, USA: ACM, 2006. p. 9–20. ISBN 1-59593-302-6.

Appendix A - GPU Δ-stepping CUDA implementation

A.1 CPU Code

This appendix contains code for the CPU and GPU part of the Δ-stepping algorithm. The first file, *grid.cu* has the CPU code used in benchmarks. It includes the GPU only code in the file *delta_kernel.cu*, available in section A.2.

```
//compile with
//nvcc grid.cu -I .../NVIDIA_CUDA_SDK/common/inc/
// -L .../NVIDIA_CUDA_SDK/lib/ -lcuda -lcudart
// -lcutil -lGL -lGLU -arch sm_11 -O3
// includes, system
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>

// includes, project
#include <cutil.h>
#include <cuda_gl_interop.h>

//#define FINETUNING
// includes, kernels
#include <delta_kernel.cu>
#define NUMTHREADS 320

float* pixels = NULL;
float* dEdges[4];
float maxDistance = 0.0f;
int GN = 262144;
```

```
int maxRCount = 0;
cudaArray* array[4];

struct edge{
  int dNode[4];
  float weight[4];
};

edge* nodes;

void
runTest( int argc, char** argv,int iw, int ih, int startNode);

void printPath(int* source, int destination){
printf("%d <- %d\n",destination,source[destination]);
if(source[destination]==destination) return;
printPath( source, source[destination]);

}

////////////////////////////////////////////////////////////////////////////
// Program main
////////////////////////////////////////////////////////////////////////////
int main( int argc, char** argv)
{
    if(argc <2){
printf("Usage: program_name graph_file graph_width
 graph_height start_node\n");
return 0;
    }
    //printf( default iw and ih is 512
    if(argc == 2){
      runTest( argc, argv,512,512,0);
    }
    else{
      int iw, ih;
      sscanf(argv[2],"%d",&iw);
      sscanf(argv[3],"%d",&ih);
      int startNode = 0;
      if(argc >= 5)
        sscanf(argv[4],"%d",&startNode);
      runTest( argc, argv,iw,ih,startNode);
    }

    //CUT_EXIT(argc, argv);
}

void loadTexture(int iw, int ih, float* data, cudaArray* cArray,
```

```
 texture<float, 2, cudaReadModeElementType>* myTex){
    cudaChannelFormatDesc desc = cudaCreateChannelDesc<float>();
    CUDA_SAFE_CALL(cudaMallocArray(&cArray, &desc, iw, ih));
    CUDA_SAFE_CALL(cudaMemcpyToArray(cArray, 0, 0, data,
sizeof(float)*iw*ih, cudaMemcpyHostToDevice));
    // Bind the array to the texture
    cudaBindTextureToArray( *myTex, cArray, desc);

}

void loadGraphEdges(int iw, int ih,char* myFile){
  int n;
  FILE* in = fopen(myFile,"r");
  fscanf(in,"%d\n",&n);

  for(int i=0;i<4;i++){
    dEdges[i] = (float*) malloc(iw*ih*sizeof(float));
    for(int j=0;j<iw*ih;j++){
      dEdges[i][j]=INF;
    }
  }

  while(1){
    int source, dest;
    double eWeight;
    fscanf(in,"%d",&source);

    if(source==-1) break;
    fscanf(in,"%d%lf\n",&dest,&eWeight);


    //links from graph are either right or down
    if(dest == source + 1){ //it's a RIGHT link
      dEdges[RIGHT][source] = eWeight;
      dEdges[LEFT] [dest  ] = eWeight;
    }
    else if(dest == source + iw){ //it's a DOWN link
      dEdges[DOWN][source] = eWeight;
      dEdges[UP]  [dest  ] = eWeight;
    }

  }

    loadTexture(iw,ih,dEdges[0],array[0],&mytex0);
    loadTexture(iw,ih,dEdges[1],array[1],&mytex1);
    loadTexture(iw,ih,dEdges[2],array[2],&mytex2);
    loadTexture(iw,ih,dEdges[3],array[3],&mytex3);
```

```
}

void printDistances(float* hDist,int iw, int ih){
   for(int i=0;i<iw*ih;i++){
      if(i%iw==0) printf("%d ",i/iw);
      if(hDist[i]<INF){
        printf("%5.1f ",hDist[i]);
if(hDist[i]>maxDistance) maxDistance=hDist[i];
      }
      else
        printf("INFINI ",hDist[i]);
      if(i%iw==iw-1) printf("\n");
    }
    return;
}



void
runTest( int argc, char** argv, int iw, int ih, int startNode)
{

    //initialize the device
    cudaSetDevice(0);
    GN = iw*ih;
    unsigned int num_threads = NUMTHREADS;

    // setup execution parameters
    dim3  grid( 1, 1, 1);
    dim3  threads( num_threads, 1, 1);
    nodes = (edge*) malloc(GN*sizeof(edge));
    loadGraphEdges(iw,ih,argv[1]);

    float* dDist;
    cudaMalloc( (void**) &dDist, GN*sizeof(float));
    float* hDist = (float*) malloc(GN*sizeof(float));
    for(int i=0;i<GN;i++){
      hDist[i]=INF;
    }
    int* dSource;
    cudaMalloc( (void**) &dSource, GN*sizeof(int));
    int* hSource = (int*) malloc(GN*sizeof(float));
    for(int i=0;i<GN;i++){
      hSource[i]=-1;
    }

    int* dBucketMap;
```

```
cudaMalloc( (void**) &dBucketMap, GN*sizeof(int));
int* hBucketMap = (int*) malloc(GN*sizeof(int));

for(int i=0;i<GN;i++){
  hBucketMap[i]=-1;
}

cudaMemcpy( dBucketMap, hBucketMap, GN*sizeof(int),
            cudaMemcpyHostToDevice);

int* dBucketPos;
cudaMalloc( (void**) &dBucketPos, GN*sizeof(int));
int* hBucketPos = (int*) malloc(GN*sizeof(int));

int* dB;
cudaMalloc( (void**) &dB, BUCKETSIZE*NUMBUCKETS*sizeof(int));
int* hB = (int*) malloc(BUCKETSIZE*NUMBUCKETS*sizeof(int));
printf("Bucket size %d\n",BUCKETSIZE);

int* hBi = hB;
for(int i=0;i<BUCKETSIZE*NUMBUCKETS;i++)
  hBi[i]=-1;

//start node
hBi[0] = startNode;
hDist[startNode] = 0.0f;

//set startNode source node in the path as the node itself,
// so that the getPath function works
cudaMemcpy( dSource+startNode, &startNode,
           1*sizeof(int), cudaMemcpyHostToDevice);

// copy host memory to device
cudaMemcpy( dDist, hDist, GN*sizeof(float),
            cudaMemcpyHostToDevice);
cudaMemcpy( dB, hBi, BUCKETSIZE*NUMBUCKETS*sizeof(int),
            cudaMemcpyHostToDevice);

int* BCount;
cudaMalloc( (void**) &BCount, NUMBUCKETS * sizeof(int));
int* hBCount;
hBCount = (int*) malloc(NUMBUCKETS*sizeof(int));
for(int i=0;i<NUMBUCKETS;i++)
  hBCount[i]=0;
hBCount[0]=1;

// copy host memory to device
cudaMemcpy( BCount, hBCount, NUMBUCKETS*sizeof(int),
```

```
            cudaMemcpyHostToDevice);
  int* dBPos;
  cudaMalloc( (void**) &dBPos, NUMBUCKETS * sizeof(int));
  int* hBPos;
  hBPos = (int*) malloc(NUMBUCKETS*sizeof(int));
  for(int i=0;i<NUMBUCKETS;i++)
    hBPos[i]=0;
  hBPos[0]=1;
  cudaMemcpy( dBPos, hBPos, NUMBUCKETS*sizeof(int),
            cudaMemcpyHostToDevice);

  // allocate device memory for result
  int * dRLoc;
  cudaMalloc( (void**) &dRLoc, GN*sizeof(int));
  int * dR;
  cudaMalloc( (void**) &dR, 16*BUCKETSIZE*sizeof(int));
   //R is 4 times bigger than RLoc, because UP,DOWN,
   //LEFT and RIGHT will each have a pos in R

  int* hR = (int*) malloc(16*BUCKETSIZE*sizeof(int));
  int* hRLoc = (int*) malloc(GN*sizeof(int));
  float* hDistR= (float*) malloc(16*BUCKETSIZE*sizeof(float));
  float* dDistR;
  cudaMalloc( (void**) &dDistR  , 16*BUCKETSIZE*sizeof(float));
  int* dSourceR;
  cudaMalloc( (void**) &dSourceR, 16*BUCKETSIZE*sizeof(int  ));

  int * dS;
  cudaMalloc( (void**) &dS, 32*BUCKETSIZE*sizeof(int));
  int * hS= (int*) malloc(32*BUCKETSIZE*sizeof(int));
  int * dSCount;
  cudaMalloc( (void**) &dSCount, 1*sizeof(int));
  int * hSCount= (int*) malloc(1*sizeof(int));

  printf("Starting timer\n");

unsigned int nvtimer = 0;
  cutCreateTimer( &nvtimer);
  cutStartTimer( nvtimer);

  float ktime;
  float* lido;
  cudaMalloc( (void**) &lido, GN*sizeof(float));
   float* Hlido = (float*) malloc(GN*sizeof(float)) ;

    int* RCount;
    RCount = (int*) malloc(1*sizeof(int));
  for(int i=0;i<470;i++){
```

```
    cudaMemcpy( RCount,  &dBPos[i],  1*sizeof(int),
                cudaMemcpyDeviceToHost) ;

    //S <- EMPTY
    hSCount[0]=0;
    cudaMemcpy( dSCount, hSCount,1*sizeof(int),
                cudaMemcpyHostToDevice);

    int sameCount = 0;
    while(RCount[0]!=0){
      if(RCount[0]>maxRCount) maxRCount = RCount[0];
      sameCount++;

      labelKernel    <<<grid, threads >>> ( i, dB, BCount,
      dBPos, iw,ih, dRLoc, dR, dDistR, dSourceR, dDist,dBucketMap);

      cudaThreadSynchronize();

      cudaMemcpy( &hBPos[i],  &dBPos[i],  1*sizeof(int),
                  cudaMemcpyDeviceToHost) ;

      copyB2SKernel  <<<grid, threads >>> ( i, dB, BCount,
                  dBPos, dS, dSCount);
      cudaThreadSynchronize();

      relaxKernelPath    <<<grid, threads >>> ( 4*4*hBPos[i],
                 dB, BCount, dBPos, dRLoc, dR, dDistR, dDist,
                 dBucketPos, dBucketMap,lido, dSourceR, dSource);

      cudaThreadSynchronize();

      cudaMemcpy( RCount,  &BCount[i],  1*sizeof(int),
                  cudaMemcpyDeviceToHost) ;
      cudaMemcpy( hSCount,  dSCount,  1*sizeof(int),
                  cudaMemcpyDeviceToHost);
    }

    labelHeavyKernel    <<<grid, threads >>> ( 0, dS,
         hSCount[0], iw,ih, dRLoc, dR, dDistR, dDist,dBucketMap);
    cudaThreadSynchronize();
    relaxKernel    <<<grid, threads >>> ( 4*4*hSCount[0], dB, BCount,
         dBPos, dRLoc, dR, dDistR, dDist,dBucketPos, dBucketMap,lido);
  }
  // check if kernel execution generated and error

CUT_CHECK_ERROR("Kernel execution failed");

  ktime = cutGetTimerValue( nvtimer );
```

```
    cudaMemcpy( hRLoc,  dRLoc,  1024*sizeof(int),
                cudaMemcpyDeviceToHost) ;
    cudaMemcpy( hRLoc,  dRLoc,  1024*sizeof(int),
                cudaMemcpyDeviceToHost) ;
    cudaMemcpy( hR,     dR,     4*1024*8*sizeof(int),
                cudaMemcpyDeviceToHost) ;
    cudaMemcpy( hDistR, dDistR, 4*1024*8*sizeof(float),
                cudaMemcpyDeviceToHost) ;
    cudaMemcpy( hS,     dS,     1024*8*sizeof(float),
                cudaMemcpyDeviceToHost) ;
    cudaMemcpy( hDist,  dDist,  GN*sizeof(int),
                cudaMemcpyDeviceToHost) ;
    cudaMemcpy( hBCount, BCount,  NUMBUCKETS*sizeof(int),
                cudaMemcpyDeviceToHost) ;
    cudaMemcpy( hB, dB,           2*NUMBUCKETS*sizeof(int),
                cudaMemcpyDeviceToHost) ;

    cudaMemcpy( Hlido, lido, GN*sizeof(float),cudaMemcpyDeviceToHost);
    cudaMemcpy( hR, dR, GN*sizeof(int),cudaMemcpyDeviceToHost);

    cudaMemcpy( hSource, dSource, GN*sizeof(int),  cudaMemcpyDeviceToHost);

    printDistances(hDist,iw,ih);
    printf("It took: %f ms\n", ktime);
    printPath(hSource, 10);

    cudaUnbindTexture(mytex0);
    cudaUnbindTexture(mytex1);
    cudaUnbindTexture(mytex2);
    cudaUnbindTexture(mytex3);

    cudaFreeArray(array[0]);
    cudaFree(dB);
    cudaFree(dRLoc);
    cudaFree(dDist);
    cudaFree(dBucketMap);
    cudaFree(dBucketPos);
    cudaFree(BCount);
    cudaFree(dR);
    cudaFree(dDistR);
    cudaFree(dSource);
    cudaFree(dS);
    cudaFree(dSCount);
}
```

A.2 GPU kernels

This section shows device code in *delta_kernel.cu* file, which is called from the above program.

```
#ifndef _DELTA_KERNEL_H_
#define _DELTA_KERNEL_H_

#include <stdio.h>
#define DELTA 1.0
#define INF 1e20
#define BUCKETSIZE 4096*8*2
#define NUMBUCKETS 512

#define DOWN  0
#define UP    1
#define RIGHT 2
#define LEFT  3


//#define EMULATION

#ifdef EMULATION
#define DEBUG(x...) printf(x)
#else
#define DEBUG(x...)
#endif

texture<float, 2, cudaReadModeElementType> mytex0;
texture<float, 2, cudaReadModeElementType> mytex1;
texture<float, 2, cudaReadModeElementType> mytex2;
texture<float, 2, cudaReadModeElementType> mytex3;

__device__ void
demptyKernel(){
}

__global__ void
labelKernel (int i, int* B,int* BCount,int* BPos, int tw,int th,
   int* RLoc,int* R, float* dR,int* dSourceR, float* d,int* vBucketMap){

  const unsigned int tid = threadIdx.x;
  const unsigned int num_threads = blockDim.x;

  int BiCount = BPos[i];
```

```
int node, row, col,index;
float cost,f1,f2,fmin;

//cleaning R
//4 times because each node can be reached from up, down, left and right
// directions (and more 4 times because for each node 4 more are open)
for(int k=0;  (num_threads*k + tid) < 16*BiCount;k++){
  R[(num_threads*k + tid)  ]=-1;
  dR[(num_threads*k + tid)  ]=INF;
  dSourceR[(num_threads*k + tid)  ]=-1;
}


__syncthreads();
const int dx[4]={0,0,1,-1};
const int dy[4]={1,-1,0,0};


for(int k=0; num_threads*k + tid < BiCount;k++){
  node = B  [ BUCKETSIZE*i + num_threads*k + tid];
  if(node!=-1){

    for(int j=0;j<4;j++){
      switch(j){
case 0:
         cost = tex2D(mytex0,node%tw,node/tw);
         break;
case 1:
         cost = tex2D(mytex1,node%tw,node/tw);
         break;
case 2:
         cost = tex2D(mytex2,node%tw,node/tw);
         break;
case 3:
         cost = tex2D(mytex3,node%tw,node/tw);
         break;
      }
      row = node/tw + dy[j];
      col = node%tw + dx[j];

      if( (row>=0) && (row < th) && (col >= 0) && (col < tw) ){
        RLoc [ row*tw + col ] = 4*(num_threads*k + tid)+j;
      }

    }

  }
}
```

```
  __syncthreads();

  //copy Edges to R
  for(int k=0; num_threads*k + tid < BiCount;k++){
    node = B  [ BUCKETSIZE*i + num_threads*k + tid];
    if(node!=-1){
      for(int j=0;j<4;j++){
        switch(j){
   case 0:
         cost = tex2D(mytex0,node%tw,node/tw);
         break;
case 1:
         cost = tex2D(mytex1,node%tw,node/tw);
         break;
case 2:
         cost = tex2D(mytex2,node%tw,node/tw);
         break;
case 3:
         cost = tex2D(mytex3,node%tw,node/tw);
         break;
      }

        row = node/tw + dy[j];
        col = node%tw + dx[j];

        if( (row>=0) && (row < th) && (col >= 0) && (col < tw) ){
          if((cost<=DELTA)&&( d[node]+cost < d[row*tw + col])){
              R [4*RLoc[row*tw + col]+j] = row*tw + col;
              dR[4*RLoc[row*tw + col]+j] = d[node]+cost;
      dSourceR[4*RLoc[row*tw + col]+j]=node;
              vBucketMap[node]=-1;
          }
        }
      }
    }
  }
  __syncthreads();

  //gathering data to find the minimum cost way to get to node n
  int smin ;
  float dists[4];
  for(int k=0;  (num_threads*k + tid) < 4*BiCount;k++){
    index    = 4*(num_threads*k + tid);
    dists[0] = dR[ index  ];
    dists[1] = dR[ index+1];
    dists[2] = dR[ index+2];
    dists[3] = dR[ index+3];
```

```
    //finds the node with minimum distance, so that the path can be stored
    if( dists[0] < dists[1]){
if( dists[2] < dists[3]){
  if( dists[0] < dists[2]){
    smin = dSourceR[index];
  }
          else{
            smin = dSourceR[index+2];
  }
}
        else{
  if( dists[0] < dists[3]){
    smin = dSourceR[index];
  }
  else{
    smin = dSourceR[index+3];
  }
}
    }
    else{
if( dists[2] < dists[3]){
  if( dists[1] < dists[2]){
    smin = dSourceR[index+1];
  }
          else{
            smin = dSourceR[index+2];
  }
}
        else{
  if( dists[1] < dists[3]){
    smin = dSourceR[index+1];
  }
  else{
    smin = dSourceR[index+3];
  }
}

    }

    f1 = fminf( dists[0] , dists[1] );
    f2 = fminf( dists[2] , dists[3] );

    fmin = fminf(f1,f2);
    dR[index  ]=fmin;
    dR[index+1]=fmin;
    dR[index+2]=fmin;
    dR[index+3]=fmin;
```

```
    DEBUG("Smin %d\n",smin);
    dSourceR[index  ]=smin;
    dSourceR[index+1]=smin;
    dSourceR[index+2]=smin;
    dSourceR[index+3]=smin;


  }
  __syncthreads();

}

__global__ void
copyB2SKernel(int i, int* B, int* BCount,int* BPos, int* S, int* SCount){

  const unsigned int tid = threadIdx.x;
  const unsigned int num_threads = blockDim.x;

  int pos;

  int BiCount = BCount[i];
  for(int k=0; num_threads*k + tid < BiCount;k++){
    if(B[i*BUCKETSIZE+num_threads*k+tid]!=-1){
      pos = atomicAdd(&SCount[0],1);
      S[pos] = B[i*BUCKETSIZE+num_threads*k+tid];
    }
  }
  __syncthreads();
  BCount[i]=0;
  BPos[i]=0;
  __syncthreads();
//  DEBUG("(tid %d) SCount %d\n",tid,SCount[0]);


}


//Parallel relax edges
__global__ void
relaxKernelPath( int RCount, int* B,int* BCount,int* BPos,
    int* RLoc,int* R,float*  dR,float* d, int* vBucketLoc,
    int* vBucketMap, float* deb, int* dSourceR, int* dSource){

  const unsigned int tid = threadIdx.x;
  const unsigned int num_threads = blockDim.x;
  int v,bn,bn_old, index;
  float x;
```

```
//remove node from old bucket



for(int k=0; num_threads*k + tid < RCount;k++){
  index = num_threads*k + tid;

  if(R[index]!=-1){

    x = dR[index];
    v = R[index];

    if(x<d[v]){

      bn_old = vBucketMap[v];
      if (bn_old != -1) {
        int oldIndex = bn_old*BUCKETSIZE+vBucketLoc[v];

        B[oldIndex] = -1;

atomicSub(&BCount[bn_old],1);

      }
    }
  }
}

__syncthreads();

for(int k=0; num_threads*k + tid < RCount;k++){
  if(R[num_threads*k + tid]!=-1){
    x = dR[num_threads*k + tid];
    v = R[num_threads*k + tid];
    if(x < d[v]){

    bn = (int) (dR[num_threads*k + tid]/DELTA);

    atomicAdd(&BCount[bn],1);
    int pos = atomicAdd(&BPos[bn],1);
    DEBUG("Pos %d BCount[%d] %d node %d (x=%f)\n",pos,bn,BPos[bn],v,x);
    B[bn*BUCKETSIZE+pos] = v;
    d[v] = x;
    dSource[v] = dSourceR[num_threads*k+tid];
    vBucketLoc[v] = pos;
    vBucketMap[v] = bn;
    RLoc[v]=-1;
```

```
      }
    }
  }

  __syncthreads();

}

//Parallel relax edges
__global__ void
relaxKernel( int RCount, int* B,int* BCount,int* BPos,
             int* RLoc,int* R,float*  dR,float* d,
              int* vBucketLoc, int* vBucketMap, float* deb){

  const unsigned int tid = threadIdx.x;
  const unsigned int num_threads = blockDim.x;
  int v,bn,bn_old, index;
  float x;
  deb[20]= (float)RCount;

  for(int k=0; num_threads*k + tid < RCount;k++){
    index = num_threads*k + tid;
    deb[2*index]= R[index];
    deb[2*index+1]= dR[index];
  }

  for(int k=0; num_threads*k + tid < RCount;k++){
    index = num_threads*k + tid;
    if(R[index]!=-1){
      x = dR[index];
      v = R[index];

      if(x<d[v]){
        bn_old = vBucketMap[v];
        if (bn_old != -1) {
          int oldIndex = bn_old*BUCKETSIZE+vBucketLoc[v];
          B[oldIndex] = -1;
  atomicSub(&BCount[bn_old],1);
        }
      }
    }
  }

  __syncthreads();

  for(int k=0; num_threads*k + tid < RCount;k++){
    if(R[num_threads*k + tid]!=-1){
      x = dR[num_threads*k + tid];
```

```
      v = R[num_threads*k + tid];
      if(x < d[v]){

      bn = (int) (dR[num_threads*k + tid]/DELTA);
      atomicAdd(&BCount[bn],1);
      int pos = atomicAdd(&BPos[bn],1);
      B[bn*BUCKETSIZE+pos] = v;
      d[v] = x;
      vBucketLoc[v] = pos;
      vBucketMap[v] = bn;
      RLoc[v]=-1;

      }
    }
  }
  __syncthreads();
}

__global__ void
labelHeavyKernel (int i, int* B,int SCount, int tw,
                  int th, int* RLoc,int* R, float* dR,
                  float* d,int* vBucketMap){

  const unsigned int tid = threadIdx.x;
  const unsigned int num_threads = blockDim.x;

  int BiCount = SCount;
  int node, row, col,index;
  float cost,f1,f2,fmin;

  //cleaning R
  //4 times because each node can be reached from up, down,
  //left and right directions (and more 4 times because
  //for each node 4 more are open)
  for(int k=0;  (num_threads*k + tid) < 16*BiCount;k++){
    index = 4*(num_threads*k + tid);
    R[index  ]=-1;
    R[index+1]=-1;
    R[index+2]=-1;
    R[index+3]=-1;
    dR[index  ]=INF;
    dR[index+1]=INF;
    dR[index+2]=INF;
    dR[index+3]=INF;
  }

  __syncthreads();
  const int dx[4]={0,0,1,-1};
```

```
  const int dy[4]={1,-1,0,0};

  for(int k=0; num_threads*k + tid < BiCount;k++){
    node = B  [ BUCKETSIZE*i + num_threads*k + tid];
    if(node!=-1){
      for(int j=0;j<4;j++){
        switch(j){
  case 0:
           cost = tex2D(mytex0,node%tw,node/tw);
           break;
  case 1:
           cost = tex2D(mytex1,node%tw,node/tw);
           break;
  case 2:
           cost = tex2D(mytex2,node%tw,node/tw);
           break;
  case 3:
           cost = tex2D(mytex3,node%tw,node/tw);
           break;
       }
       row = node/tw + dy[j];
       col = node%tw + dx[j];

       if( (row>=0) && (row < th) && (col >= 0) && (col < tw) ){
         RLoc [ row*tw + col ] = 4*(num_threads*k + tid)+j;
       }
      }
    }
  }

  __syncthreads();

  //copy Edges to R
  for(int k=0; num_threads*k + tid < BiCount;k++){
    node = B  [ BUCKETSIZE*i + num_threads*k + tid];
    if(node!=-1){
      for(int j=0;j<4;j++){
        switch(j){
   case 0:
         cost = tex2D(mytex0,node%tw,node/tw);
         break;
case 1:
         cost = tex2D(mytex1,node%tw,node/tw);
         break;
case 2:
         cost = tex2D(mytex2,node%tw,node/tw);
         break;
case 3:
```

```
          cost = tex2D(mytex3,node%tw,node/tw);
          break;
        }

          row = node/tw + dy[j];
          col = node%tw + dx[j];
          if( (row>=0) && (row < th) && (col >= 0) && (col < tw) ){
            if((cost>DELTA)&&( d[node]+cost < d[row*tw + col])){
                R [4*RLoc[row*tw + col]+j] = row*tw + col;
                dR[4*RLoc[row*tw + col]+j] = d[node]+cost;
                vBucketMap[node]=-1;
            }
          }
        }
      }
    }
    __syncthreads();

    //gathering data to find the minimum cost way to get to node n
    for(int k=0;  (num_threads*k + tid) < 16*BiCount;k++){
      f1 = fminf( dR[4*(num_threads*k + tid)  ], dR[4*(num_threads*k + tid)+1] );
      f2 = fminf( dR[4*(num_threads*k + tid)+2], dR[4*(num_threads*k + tid)+3] );
      fmin = fminf(f1,f2);
      dR[4*(num_threads*k + tid)  ]=fmin;
      dR[4*(num_threads*k + tid)+1]=fmin;
      dR[4*(num_threads*k + tid)+2]=fmin;
      dR[4*(num_threads*k + tid)+3]=fmin;
    }
    __syncthreads();
}

__global__ void
emptyKernel(){
}

#endif // #ifndef _MEMORY_KERNEL_H_
```

FOLHA DE REGISTRO DO DOCUMENTO

1. CLASSIFICAÇÃO/TIPO	2. DATA	3. DOCUMENTO N	4. N DE PÁGINAS
TM	21 de novembro de 2007	CTA/ITA - IEC/TC-009/2006	108

5. TÍTULO E SUBTÍTULO:

GPGPU Based Image Segmentation Livewire Algorithm Implementation

6. AUTOR(ES):

Daniel Lélis Baggio

7. INSTITUIÇÃO(ÕES)/ÓRGÃO(S) INTERNO(S)/DIVISÃO(ÕES):

Instituto Tecnológico de Aeronáutica. Divisão de Engenharia da Computação – ITA/IEC

8. PALAVRAS-CHAVE SUGERIDAS PELO AUTOR:

GPGPU; CUDA; Segmentation; LiveWire

9. PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO:

Image Processing; Algorithms; Delta-stepping; Shortest-path

10. APRESENTAÇÃO: ( ) **Nacional** (**X**) **Internacional**

ITA, São José dos Campos, 2007, 108 páginas

11. RESUMO:

This thesis presents a GPU implementation of the *Livewire* algorithm. Instead of using traditional architectures, like the CPU, this implementation focuses advantages obtained using *Single Instruction Multiple Data* (SIMD) architectures. The algorithm is divided in three phases: *Sobel* or *Laplacian* filter convolution, image modeling as a grid graph and solving the non-negative weighted edges single source shortest path problem. In order to calculate the shortest path, a parallel approach is made through the development of an adapted version of the Δ-stepping algorithm for GPUs. Each part of the algorithm was programmed as a single kernel, which is executed and compiled to the GPU, using CUDA (*Compute Unified Device Architecture*), available on NVidia GeForce 8 series. GPUs have been the first SIMD commodity devices widely available on several desktops. Although originally designed for applications highly focused on rendering, GPGPU (*General Purpose Computing on Graphic Processing Units*) researchers have shown that the huge processing power available on these devices as well as the recent advent of a programmable pipeline have made of GPUs an attractive option for low cost high performance platforms. Even though the implementation has used CUDA API, several other approaches are pointed out, showing a wide variety of alternatives such as other platforms – multicore CPUs, *Cell* processor –, other graphic APIs, such as Cg, and OpenGL, or even different approaches like RapidMind and Brook, which make GPU access transparent. The conclusion of this thesis highlights a successful implementation of the algorithm using the GPU architecture showing advantages and disadvantages of this approach. An in-depth result analysis shows that intense speedups are seen in image filtering algorithms. On the other hand, the wide use of dependent device memory look-ups has constrained Δ-stepping algorithm from achieving higher performance than CPU implementation although a better performance is expected for wider graphs. If device memory access latency was decreased or if more threads were available, a huge increase in performance would be expected. Besides showing the viability of the *Livewire* algorithm implementation, this thesis makes available an *open-source* image segmentation GPU based application, which can be used as example for future GPU algorithm implementations at http://code.google.com/p/gpuwire/.

12. GRAU DE SIGILO:

(**X**) **OSTENSIVO** ( ) **RESERVADO** ( ) **CONFIDENCIAL** ( ) **SECRETO**