DOI:10.15774/PPKE.ITK.2014.005

Implementation of Medical Imaging Algorithms on Kiloprocessor Architectures

Gábor János Tornai

Faculty of Information Technology and Bionics

Pázmány Péter Catholic University

Scientific Advisor:
György Cserey Ph.D.
Consulent:
Tamás Roska DSc.

A thesis submitted for the degree of

Ph.D.

Budapest, 2014

DOI:10.15774/PPKE.ITK.2014.005

DOI:10.15774/PPKE.ITK.2014.005

I would like to dedicate this thesis to my loving wife, my children and parents ...

Love never fails. But where there are prophecies, they will cease; where there are tongues, they will be stilled; where there is knowledge, it will pass away. For we know in part and we prophesy in part, but when completeness comes, what is in part disappears. When I was a child, I talked like a child, I thought like a child, I reasoned like a child. When I became a man, I put the ways of childhood behind me. For now we see only a reflection as in a mirror; then we shall see face to face. Now I know in part; then I shall know fully, even as I am fully known. 1 Cor 13,8-12

DOI:10.15774/PPKE.ITK.2014.005

Acknowledgements

I deeply thank *Tamás Roska* and *Péter Szolgay* the former and the present head of the doctoral school for their advice and encouragement giving me the strength to go on.

It is hard to precisely articulate how I thank *György Cserey* my advisor the fascinating work lasting more than seven years. Without his excellent ideas and inspiration I would not have been successful in my doctoral studies. We managed to find a way to discuss the real life problems and joys as well.

I thank *Ion Pappas* his support, encouragement, advices, and inspiration. During our collaboration I learned how industry and academy can benefit from each other.

I thank Barnabás Garay for guiding my first steps in the world of partial differential equations. Additionally, he showed me how mathematicians treat problems. Thanks for the great impression.

To the other doctoral students: *András Horváth, Attila Stubendek, Domonkos Gergely, Miska Radványi, Tamás Fülöp, Ádám Rák, Tam Zsedrovits, Miki Koller, Csaba Józsa, Csaba Nemes, Bence Borbély, Zoli Tuza, Jani Rudan, Dóri Bihari, Anna Horváth, András and Zsolt Gelencsér*. Thanks for the chat near the lunch, the help, the seminars on Fridays, and the other activities.

I thank *Vida Tivadarné Katinka* her eternal service and patience to make the administrative side of life much easier, the work of the dean's office, the administrative and financial personnel.

I had some really great teachers who should be mentioned here as well. I am grateful to Zsuzsanna Vágó, Miklós Rásonyi, János Lev-

DOI:10.15774/PPKE.ITK.2014.005

endovszky and Árpád Csurgay from the Faculty, Mária Bertalan and Katalin Szabó from Reformed Collage Secondary School of Debrecen.

The Öveges scholarship of General Electric Healthcare and the datasets provided by the company are gratefully acknowledged.

The support of grants Nos. TÁMOP-4.2.1.B-11/2/KMR-2011-0002, and TÁMOP-4.2.2/B-10/1-2010-0014 are gratefully acknowledged.

Thank you *Father, Mother, Piri, Tamás, Ildi* and the whole family. Everything.

Niki, Magdi and Blanka. You are worth fighting for, and it is a great joy to return back home to you.

DOI:10.15774/PPKE.ITK.2014.005

Abstract

This dissertation presents two specific fields of medical imaging: (i) fast digitally reconstructed radiograph (DRR) generation on Graphical Processing Units (GPUs) allowing two dimensional to three dimensional (2D to 3D) registration to be performed in real-time and (ii) several sides of the level-set (LS) based methods: theoretical aspects such as initial condition dependence and practical aspects like mapping these algorithms on many core systems.

The generation of DRRs is the most time consuming step in intensity based 2D x-ray to 3D (computed tomography (CT) or 3D rotational x-ray) medical image registration, which has application in several image guided interventions. This work presents optimized DRR rendering on graphical processor units (GPUs) with optimization rules and compares performance achievable on several commercially available devices. The presented results outperform other results from the literature. This shows that automatic 2D to 3D registration, which typically requires a couple of hundred DRR renderings to converge, can be performed on-line, with the speed of 0.5-10 frames per second (fps). Accordingly, a whole new field of applications is opened for image guided interventions, where the registration is continuously performed to match the real-time x-ray.

I investigated the effect of adding more small curves to the initial condition which determines the required number of iterations of an LS evolution. As a result, I discovered two new theorems and developed a proof on the worst case of the required number of iterations. Furthermore, I found that these kinds of initial conditions fit well to many-core architectures. I have shown this with two case studies, which are presented on different platforms. One runs on a GPU

DOI:10.15774/PPKE.ITK.2014.005

and the other is executed on a Cellular Nonlinear Network-Universal Machine (CNN-UM). Additionally, segmentation examples verify the applicability of the proposed method.

DOI:10.15774/PPKE.ITK.2014.005

Contents

DOI:10.15774/PPKE.ITK.2014.005

DOI:10.15774/PPKE.ITK.2014.005

DOI:10.15774/PPKE.ITK.2014.005

Nomenclature

Roman Symbols

B	Diameter of an object
$\mathbf{D}$	Domain of ϕ
F	Driving force, also known as speedfield of the LS equation
F	driving force of LS equation
$N(.)$	Neighborhood set of the argument

Greek Symbols

ϕ	LS function
Γ	Background region
γ	interface (curve, surface) to be tracked
Ω	Object region
τ	time constant of the CNN cell

Acronyms

AP	Anterior-Posterior
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit

DOI:10.15774/PPKE.ITK.2014.005

CBCT	Cone Beam Computed Tomography
CNN	Cellular Neural Network
CNN-UM	CNN-Universal Machine
CPU	CentralProcessing Unit
CT	Computed Tomography
CUDA	Computing Unified Device Architecture
DRR	Digitally Reconstructed Radiograph
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
GFLOPS	Giga Floating Point Operations per Second
GPU	Graphical processing unit
ISA	Instruction Set Architecture
LAT	(Medio-)Lateral
LS	Level-Set
MIP	Maximum Intensity Projection
MR	Magnetic Resonance
N_{it}	Number of iterations
ODE	Ordinary Differential Equation
PDE	Partial Differential Equation
PTX	Parallel Thread Execution
SIMT	Single Instruction Multiple Threads
SM	Streaming Multiprocessor

DOI:10.15774/PPKE.ITK.2014.005

SoC	System on a Chip
TPC	Texture Processing Cluster
UMF	Universal Machine on Flows
US	Ultrasound
VSoC	Visual System on a Chip

DOI:10.15774/PPKE.ITK.2014.005

List of Figures

DOI:10.15774/PPKE.ITK.2014.005

DOI:10.15774/PPKE.ITK.2014.005

DOI:10.15774/PPKE.ITK.2014.005

List of Tables

DOI:10.15774/PPKE.ITK.2014.005

Chapter 1

Introduction

Medical imaging analysis helps doctors in a wide spectrum starting from diagnosis formation to therapy monitoring. It is possible to discover an abnormal change in a given region, to quantify the possibility of different cancer types in specific tissues. These methods work correctly and precisely thanks to the sophisticated algorithms and the development of the semiconductor devices and technology making them realizable in feasible time. In this way, a lot of healthcare procedures and protocols become fast and reliable, for example, minimally invasive operations are available thanks to the advanced imaging systems and analysis.

The mentioned new devices are the many-core architectures spreading and developing in the last few years. The reason of this technology trend is the scaling down of the technological feature size. The former classical architecture with one processor core and increasing clock frequency was blocked by the dissipative power wall and the wiring delay since the secondary parasitic effects can not be neglected any more. The solution to this problem was a slight decrease in the clock frequency and the placement of more processor cores on the same chip. These cores are organized according to a given topology. This is true for central processing units (CPU), graphical processing units (GPU), some application specific integrated circuits (ASIC), and field programmable gate arrays (FPGA) as well.

As an example, one can think of the SUN SPARC [1] and the IBM POWER [2] for CPUs with several cores, Fermi and Kepler architecture of Nvidia and AMD Southern Islands for GPUs. The Q-Eye focal plane processor and the Xenon

DOI:10.15774/PPKE.ITK.2014.005

architecture are good examples for ASICs. All previous examples are homogen architectures. However, there are examples for inhomogen ones as well. Xillinx and Altera FPGAs have naturally a throng of programmable logic but incorporate other elements like digital signal processor (DSP) slices, classical CPU cores (POWER, ARM), etc. Both Intel and AMD manufactures such chips that have a full functional GPU next to the processor cores with a commonly managed cache.

Each of these devices have several great properties of the following like huge performance, bandwidth, low power dissipation, or small chip area. However to be able to benefit from these advantages, new algorithms, and new optimization methods shall be considered for the set of tasks to be solved. These tasks may include completely new ones or older ones considered unfeasible. In my dissertation I present two tasks and their solutions mapped on given many-core devices.

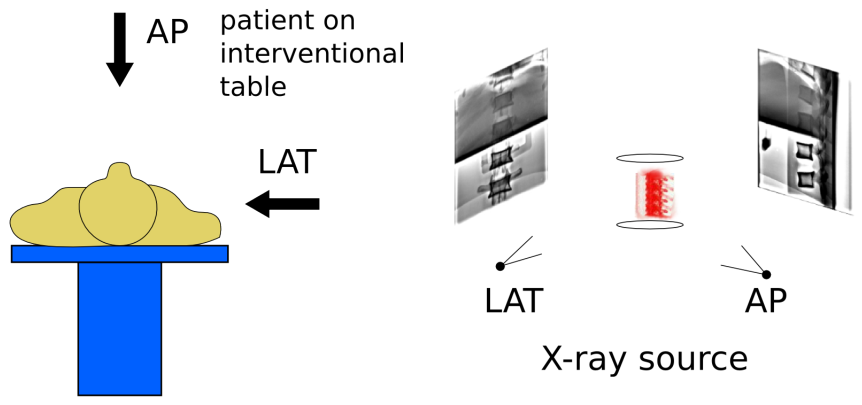
The first task is the rendering of digitally reconstructed radiographs (DRR) that is a key step to several image guided therapy (IGT) applications. I am focusing on the alignment of 3D images taken before the intervention and 2D images taken during an intervention. This alignment procedure called is 2D to 3D registration.

The second task is to achieve as much speedup as possible on a level set (LS) method. The LS methods in general have vast applications from computational geometry through crystal growth modeling to segmentation. I have chosen a method ideal for fast (medical image) segmentation among others.

During my graduate work firstly I wanted to find a solution to be able to realize 2D to 3D registration in real-time in interventional context. The motivation came from the problem assignment of GE Healthcare. It was known both from the literature and from various measurements on CPU that the DRR generation is the most time consuming step. More than the 95% of the execution time is spent with this task if it is done on the CPU. There is the question how can the DRR generation be faster with orders of magnitude without quality degradation.

DRRs are simulated X-ray images since, the attenuation of a virtual X-ray beam is calculated by an algorithm by projecting 3D computed tomography (CT) images or 3D reconstructed rotational X-ray images. In particular, DRRs are used in patient position monitoring during radiotherapy or image guided surgery for automatic 2D to 3D image alignment [3, 4, 5], that is called 2D to 3D registration,

DOI:10.15774/PPKE.ITK.2014.005



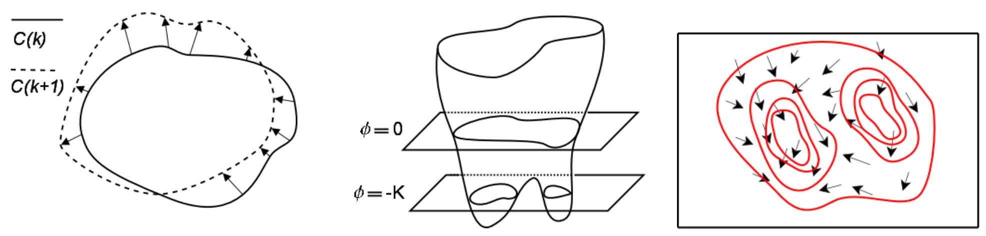
(a) reality (b) virtual

Figure 1.1: 2D to 3D registration. (a) The intervention is monitored from several directions for example with a frontal view (anterior-posterior, AP) and a side view (lateral, LAT). These 2D projection images are aligned with a preoperative CT scan from the same region: (b) the proper pose of the CT is searched until it matches with the pose of the patient's actual one.

and overlay illustrated in Figure 1.1. In my thesis I considered the fast calculation of the DRR rendering that is usually the most time-consuming step and performance bottleneck in these applications. I present a solution to accelerate the 2D to 3D registration performance by properly mapping, implementing and optimizing the DRR execution on state of the art GPUs. So, during an intervention the 2D to 3D registration can be done in real-time. This makes possible for example the on-line tracking and data fusion of interventional devices in the pre-operative CT volume, or the precise dose delivery in an oncological radiation therapy application.

The use of LS based curve evolution has become an interesting research topic due to its versatility and accuracy. These flows are widely used in various fields like computational geometry, fluid mechanics, image processing, computer vision, and materials science [6, 7, 8]. In general, the method entails that one evolves a curve, surface, manifold, or image with a partial differential equation (PDE) and obtains the result at a point in the evolution (see Figure 1.2). The solution can be a steady state, (locally) stable or transient. There is a subset of problems where only the steady state of the LS evolution is of practical interest like segmentation,

DOI:10.15774/PPKE.ITK.2014.005



(a) change of the curve (b) level set function (c) driven by a force field

Figure 1.2: Illustration of LS based methods. (a) Given a curve whose shape is driven by forces acting on its perimeter. (b) Various considerations (automatic handling of topological changes, numerical stability, etc.) suggest to embed the curve in a higher dimensional function as its LS. (c) From now on, the evolution of the higher dimensional function is considered and the desired curve is represented implicitly as the zero LS of this function.

(shape) modeling and detection. Only this subset is considered in this work. In addition, I do not form, design any operator or driving force/speed-field for driving the evolution dynamics of the LSs.

The initial direction in this field was the mapping and analysis of the previously described subset of LS to many-core architectures. The experience gathered during the experiments made me recognize and propose answers to the following questions. Is there a family of initial conditions that was not considered on conventional CPUs since it was not logical? What kind of initial condition can be mapped optimally to many-core architectures? Is there a worst case bound on the required number of iterations of a given LS evolving method?

The structure of this dissertation is as follows. Chapter 2 presents the work on fast DRR rendering on GPU for 2D to 3D registration. Furthermore, a specific aspect of optimization rules and block size dependence of GPUs are revealed. In Chapter 3 the initial condition family for efficient mapping of LS evolution to many-core architectures is proposed giving both the theoretical and the practical side of the material. Chapter 4 gives the conclusions of the dissertation and summarizes the main results. Appendix A describes GPU architecture and GPU computing. Appendix B gives an introduction to CNN computing together with some template definitions. Appendix C shows further measurement data connected to the DRR generation on GPUs that was left out from the main text.

DOI:10.15774/PPKE.ITK.2014.005

Chapter 2

Fast DRR generation for 2D to 3D registration on GPU

In this Chapter I present my work related to 2D to 3D registration and fast DRR rendering. Firstly, the background and the context of my work is described in Section 2.1. The review of the related work is presented in Section 2.2. Then the hardwares, the algorithm, the datasets and measurement setup are described in Section 2.3. It is followed by the results in Section 2.4. Section 2.5 discusses my results and Section 2.6 gives the conclusions.

2.1 Background

Motion and exact position in general is a major source of uncertainty during several kinds of intervention like radiotherapy, radiosurgery, endoscopy, interventional radiology, and image guided minimally invasive surgery. Patient position and motion monitoring plays an essential role in this scenario if it meets the (critical) time requirements of the given application. However, to solve this task, registration has to be applied and if the most recent results are not counted it is performed usually in 20-1000 seconds depending on the size of the data, the exact algorithm and the hardware. This procedure can estimate the six degrees of freedom rigid transformation connecting the 3D (pre-interventional) data to the (intra-interventional) monitoring data. Registration brings the pre-

DOI:10.15774/PPKE.ITK.2014.005

operative data and interventional data together in the same co-ordinate frame. Pre-interventional data are 3D computed tomography (CT), and magnetic resonance (MR) images while intra-interventional data are 2D ultrasound (US), X-ray fluoroscopy, and optical images, or 3D images like cone beam CT (CBCT), US, or 3D digitized points or parametrized surfaces.

In minimally invasive surgery, registration offers the surgeon accurate position of the instruments relative to the planned trajectory, the vulnerable regions and the target volume. In radiation therapy and radiosurgery, it adjusts the radiation precisely to the actual position of the target volume compensating both inter- and intrafractional motion that is of great importance for exact dose delivery. In endoscopy, it provides augmented reality by rendering virtual images of the anatomy and pathology and uncovering structures that are hidden from the actual view.

Registration can be 2D to 3D or 3D to 3D. In the former case pre-interventional CT or MR is registered to intra interventional 2D X-ray projection images. In the latter case the intra interventional image is a CBCT, CT, MR, or US image. There are methods like 2D to 3D slice to volume and video to volume well [9, 10, 11].

The different datasets are represented in different coordinate systems. The pre-interventional data, the intra-interventional data, and the treatment (intervention room, patient) itself define their own coordinate system. So depending on the type one can differentiate between 3D to 3D, image to patient; 3D to 3D, image to image; and 2D to 3D, image to image registration. Registration finds the transformation $\mathbf{T}$ that links the different coordinate systems. In the first case no intra-interventional image is taken but some points or landmarks are determined and aligned on the patient and the image. So there is no direct correspondence with all points of the pre-interventional data. In the second case $\mathbf{T}$ maps all points that appear on both images. This case incorporates image resampling and interpolation. The third case is 2D to 3D image to image registration that may refer to volume to slice or volume to projected image registration. This dissertation is connected to the latter one.

2D projected image to 3D image registration can be semi-intramodal [3, 4, 12, 13, 14, 15, 16, 17] where the 2D image is some kind of X-ray (fluoroscopy,

DOI:10.15774/PPKE.ITK.2014.005

subtraction angiography) and the 3D image is a CT, or multi-modal [18, 19, 20, 21, 22, 23, 24] where the 3D image is usually an MR image. Semi-intramodal means the intensity of both images describes similar characteristic namely, the attenuation of the X-ray beam however, the beam used for the CT has different frequency characteristics than those used for 2D X-ray image. Multi-modal means the quality of the tissue measured in the case of MR is independent from the X-ray beam attenuation of the same tissue. The registration algorithms have very similar basic steps. The first step is the initialization of the transformation **T** and preprocessing of each data if needed. The second step is to make the 2D and the 3D image comparable. This can be intensity based, gradient based, and geometry based.

Intensity based methods are the most intuitive and among the most successful ones. In this case, the 3D volume is projected with a given camera geometry (see Figure 2.1). The projection can be ray-cast type (DRR image), or maximum intensity projection type (MIP image). And this image (DRR, MIP image) is compared to the interventional image. Geometry based methods create correspondence between points, surfaces, or landmarks from the images and use only these features to optimize **T**. Obviously, these processes require less data. The gradient based method lies somewhere in the middle [25]. It creates the gradient map first from the 3D image and projects only this map. The next step is the calculation of the similarity measure like information theoretic type (mutual information, Kullback-Lieber divergence), norm type ($\|.\|_p, p = 1, 2$), or correlation type metrics to be optimized. Then an optimizer (Powell's method, Downhill simplex, Levenberg-Marquardt, sequential Monte Carlo, gradient descent, simulated annealing) modifies **T** and it starts again from creating the comparable images. There was an evaluation of several different optimizers in radiotherapy [26] and were found to have equal performance. The workflow of intensity based 2D projected image to 3D image registration can be seen in Figure 2.2.

DRRs are simulated X-ray images generated by projecting 3D computed tomography (CT) images or 3D reconstructed rotational X-ray images. The DRR rendering is usually the most time-consuming step and performance bottleneck in these applications. In this Chapter, I present a solution to accelerate the 2D to 3D registration performance by implementing and optimizing the DRR execu-

DOI:10.15774/PPKE.ITK.2014.005

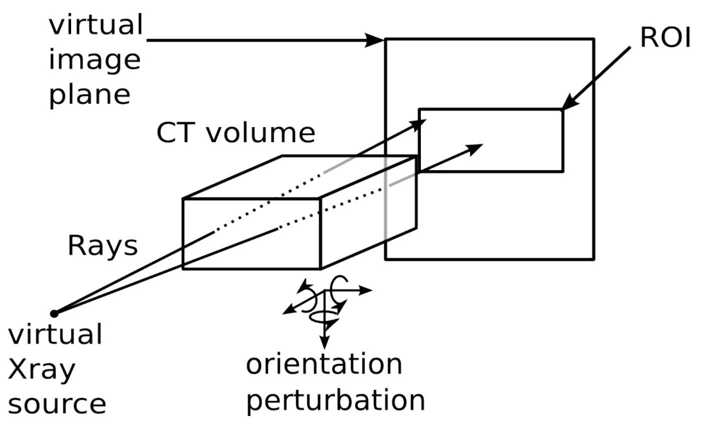


Figure 2.1: DRR rendering geometry. Rays are determined by virtual X-ray source and pixel locations on the virtual image plane, inside the ROI.

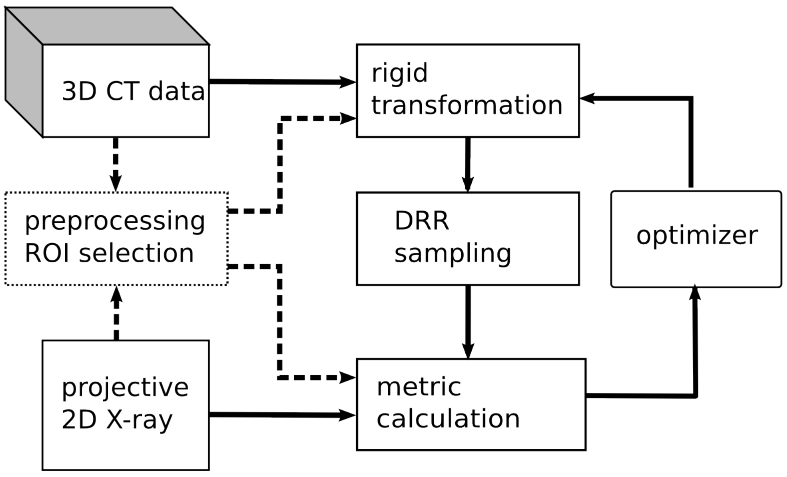


Figure 2.2: 2D projected image to 3D image registration. 2D to 3D intensity based image registration is an iterative process. From the 3D image (sampled) DRRs are rendered according to the camera geometry of the 2D projected image and the actual state of the transformation. This sampled DRR is compared to the original projective X-ray with a similarity measure (mutual information, L_1, L_2 norms, correlation type ones, etc.), and this evaluation is fed to the optimizer which makes a better approximation of the rigid transformation connecting the two datasets. This procedure is iterated until a desired confidence is reached.

DOI:10.15774/PPKE.ITK.2014.005

tion on state of the art Graphical Processing Units (GPUs). GPUs have become efficient accelerators of several computing tasks. They are kilo-core devices with high computing capacity and bandwidth.

2.2 Related Work

Although DRR rendering based registration is far the most reported method, it has some drawbacks. First of all, DRR rendering requires high computational complexity. Additionally, its application together with 3D MR images is limited since there is no physical correspondence between MR and X-ray attenuation of any matter except if special contrast agents are present. Another problem is that by the projection of the 3D image, valuable spatial information is lost. There was a study where a probabilistic extension was introduced to the computation of the DRRs that preserves the spatial information (separability of tissues along the rays) and the resulting non scalar data is handled via an entropy based similarity measure [27]. Unfortunately, the computational burden is even higher in this case.

There are numerous papers [3] presenting a wide spectrum of results connected to the acceleration of DRR generation or reducing the required number of renderings [33, 13, 34, 32]. These results can be divided into three classes: results relying only on algorithmic improvements implemented on CPU, others based on GPU computing that includes algorithmic innovations as well, and methods reducing DRR generation to a subspace that can be a segmented volume, back-projected ROI, or a statistical model. The third class (reduction to a subspace) is a hybrid method since it incorporates some features of the geometric and gradient based methods as well. Table 2.1 presents a condensed summary of the reported results in acceleration of DRR rendering without degrading the volume into a subspace. In addition, it presents specific results in 2D to 3D registration that are straightforward in the way of DRR rendering, as well.

The algorithmic approaches include shear warp factorization [35], transgraph [34], attenuation fields [16], progressive attenuation fields [14] and wobbled splatting [12]. The first three approaches require considerable pre-computation (up to 6 hours) the last two do not. The hardware based improvements use mainly

DOI:10.15774/PPKE.ITK.2014.005

Table 2.1: Summary of published results connected to fast DRR rendering and 2D to 3D registration.

Hardware	Fab	TDP	Price	Year	ROI size	t_{DRR}	t_{Regist}	Comments	Group
Intel Xeon (2.2 GHz)				2005	256×256	50 ms	100 s	SW cache, 0.4-6h preproc	[16]
Intel Pentium 4 (2.4 GHz)				2005	200×200	-	2-17 min	SW cache, no preproc	[14]
Intel Xeon				2009	256×256	-	30-120 s	ROI rand. samp. (5%)	[12]
Intel Core (1.8 GHz octal)				2009	256×172	50 ms	-	multithreaded ray cast	[28]
GeForce 7600 GS (12 SM)	90nm	32W	140$	2007	512×512	54-74 ms	-	voxel based, $(6 \cdot 10^6$ voxel)	[29]
GeForce 7800 GS (16 SM)	110nm	75W	200$	2008	256×256	73 ms	-	X-ray postproc. imitation	[15]
GeForce 8800 GTS (16 SM)	65nm	140W	300$	2012	task dep.	-	0.4-0.7 s	special ROI selection	[30]
GeForce 8800 GTX (16 SM)	90nm	160W	600$	2012	512×267	51 ms	-	100%	[31]
GeForce 580 GTX (16 SM)	40nm	244W	520$	2012	512×267	184 ms	-	100%	[31]
GeForce 8800 GTX (16 SM)	90nm	160W	600$	2012	512×267	27 ms	-	ROI and lineintegral samp.	[31]
GeForce 580 GTX (16 SM)	40nm	244W	520$	2012	512×267	8 ms	-	ROI and lineintegral samp.	[31]
GeForce 8800 GT (14 SM)	65nm	105W	280$	2012	400×225	1.2-7.3 ms	-	rand. samp. (1.1-9.1, 100%)	[32]
GeForce 280 GTX (30 SM)	65nm	236W	460$	2012	400×225	0.5-4.5 ms	-	rand. samp. (1.1-9.1, 100%)	[32]
Tesla C 2050 (14 SM)	40nm	238W	2700$	2012	400×225	0.4-3.9 ms	-	rand. samp. (1.1-9.1, 100%)	[32]
GeForce 580 GTX (16 SM)	40nm	244W	520$	2012	400×225	0.3-2.5 ms	-	rand. samp. (1.1-9.1, 100%)	[32]
Tesla C 2050 (14 SM)	40nm	238W	2700$	2014	400×225	0.23-2.7 ms	-	rand. samp. (1.1-9.1, 100%)	own
GeForce 570 GTX (15 SM)	40nm	219W	330$	2014	400×225	0.18-2.2 ms	-	rand. samp. (1.1-9.1, 100%)	own

DOI:10.15774/PPKE.ITK.2014.005

GPUs [29, 15, 36, 17, 37, 38, 30, 32, 31] but not exclusively [28]. The accurate ROI selection based on a planned target volume together with the DRR rendering performed on the GPU in a recent work [30] indicates the viability of quasi real time operability. Additionally, stochastic techniques are also applied by generating randomly sampled DRRs, or sampling the rendered full DRRs [13, 18, 12, 32]. There are two advantages generating and using randomly sampled DRRs. The first reason is the DRR generating time, it is nearly always smaller compared to the full DRR. The second reason is the drop in the metric calculation time. It must be emphasized that the precision of the registration does not degrade provided the sampling density does not drop below $5-3\%$ [13, 18, 12]. Therefore, the evaluation presented here is essential to show how fast the DRR rendering can be done on the GPU for registration purposes.

2.3 Materials and Methods

During the work connected to the DRR rendering the following materials and methods were used. First, I give a condensed overview of GPUs. After the algorithm is specified together with four optimization rules for the realization, the datasets are itemized, and the measurement setup is depicted.

2.3.1 GPU

Recent GPU models are capable of non-graphic operations and are programmable through general purpose application programming interfaces (APIs) like C for CUDA [39] or OpenCL [40]. In this Chapter, C for CUDA nomenclature is used. The description below is a brief overview of GPUs, and only those notations are summarized that have an impact on the performance of the DRR rendering. An extensive description of GPUs is found in Appendix A. A schematic block diagram of a GPU can be seen in Figure 2.3(a).

A function that can be executed on the GPU is called a kernel. Any call to a kernel must specify an execution configuration for that call. This defines not just the number of threads to be launched but their logical arrangement as well.

Threads are organized into blocks and blocks build up a grid. An illustration

DOI:10.15774/PPKE.ITK.2014.005

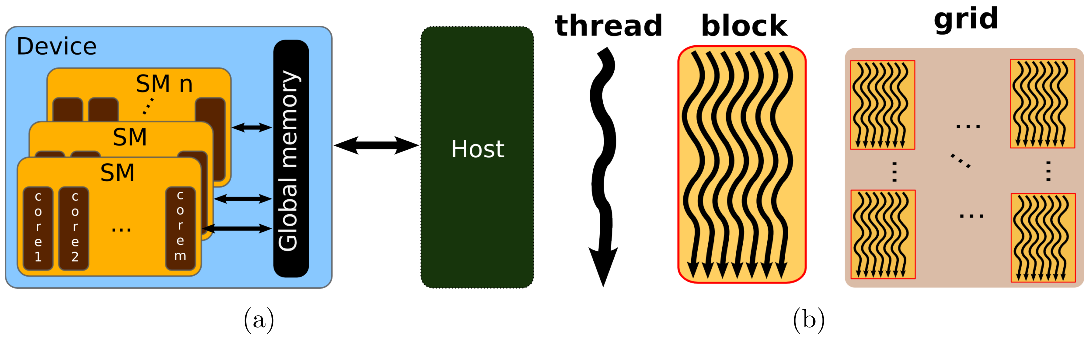


Figure 2.3: (a)Schematic block diagram of a GPU. (b) Illustration of logical hierarchy of execution. Threads are organized into blocks and blocks are groupped to form a grid.

can be seen in Figure 2.3(b). The dimensionality of a block or a grid can be one, two or three. The number of threads in a block is referred to as block size. This is a three tuple of positive integers but in this work only one-dimensional blocks and grids are considered. For a given (total) number of threads, the block size has a great impact on the performance and there have been no explicit rules to find its optimal value for an algorithm implementation. Physically, the scheduling of threads within a block on a streaming multiprocessor (SM) is done in fixed units. This fixed unit is called 'warp' and comprises of 32 threads. The warp size is 32 on all GPUs that are employed in this work. As a consequence, the vendor advises block sizes that are multiples of 32, at least 64 to avoid underutilization as a rule of thumb. The parallel thread execution (PTX) [41] is an intermediate, device independent GPU language above architecture specific instruction set. During the compilation, the kernel is translated first to PTX and then compiled to device dependent code.

Texture memory is a special kind of memory space hidden from direct access from a kernel function. It is a read only, cached memory optimized for spatially coherent local access. Its caching is managed via texture processing clusters (TPC) that is a group of SMs. Each TPC has one channel to access its cache space. The texture memory is read through a texture reference specifying the reading and interpolation mode.

DOI:10.15774/PPKE.ITK.2014.005

Table 2.2: GPUs used in this work. Streaming Multiprocessors (SMs) are a compact group of cores. Texture Processing Clusters (TPCs) are responsible for texture memory that is a cached read-only memory employed in this work. Compute capability is composed of a major and a minor version number used by the vendor denoting architectural versions. See Appendix A.4 for details.

	8800 GT	280 GTX	C2050	570 GTX	580 GTX
cores	112	240	448	480	512
SMs	14	30	14	15	16
TPCs	7	10	14	15	16
compute capability	1.1	1.3	2.0	2.0	2.0
CLK$_{proc}$ (GHz)	1.5	1.3	1.15	1.5	1.5
CLK$_{mem}$ (GHz)	0.9	1.1	1.5	2	2
bus width (bit)	256	512	384	384	384
released in	2007	2008	2009	2010	2010

2.3.2 Algorithm and realization

A ray is a line segment determined by the virtual X-ray source position and a pixel location on the virtual image plane in the 3D scene (Figure 2.4). The logarithm of a pixel intensity is the line integral of the ray segment inside the volume.

There are two basically different ways to map the task on a many core hardware. The first possible approach is volume based. The contribution of voxels of a volume tile is calculated for each pixel in the image, and then iterated to the next tile inside the volume. The second one is ray based, namely, each thread follows a ray and approximates the line integral along the ray known as DRR pixel intensity or sampling value. The first method was rejected because random sampling is applied on the DRRs to reduce the computational burden of calculating the objective function for a given point in the 6 dimensional parameter space. Furthermore, one voxel can contribute to many pixels locally on the image plane. So the internal bandwidth of the GPU would have been wasted by very inefficient global memory reads and by repetitive uncoalesced writes.

The algorithm has two main parts. The goal of the first part is to obtain a

DOI:10.15774/PPKE.ITK.2014.005

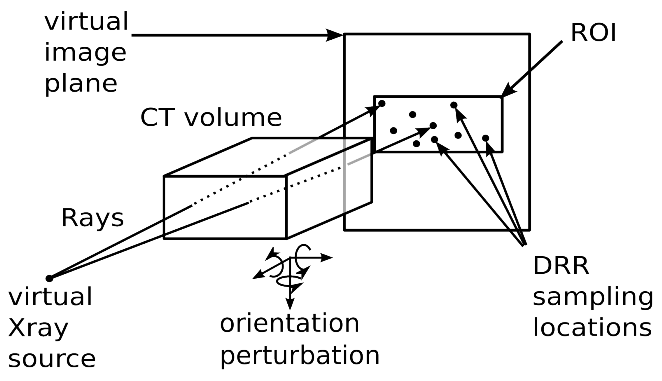


Figure 2.4: DRR rendering. Pixel intensities are approximated line integrals along the dashed line segments (volume interior). ROI is resampled for each DRR rendering. Similarly, the CT position and orientation are varied by uniform distribution.

normalized direction vector corresponding to the ray calculated by the thread, an entry and an exit point on the volume of the given ray. This goal is reached through several steps. First the 2D position of the pixel (within the virtual image plane) corresponding to the ray is transformed into the coordinate system of the CT data. This is followed by the calculation of the entry and exit points of the ray on the CT volume. Then the normalization follows that resizes the direction vector to be equal to a side of a voxel in the volume. The second main part is the main loop that approximates the line integral (see Listing 2.1).

The following optimization rules were applied to maximize the efficiency of the implementation of the algorithm:

1. Slow ‘if else’ branches shall be replaced with ternary expressions that are compiled to ‘selection’ PTX instructions that are faster than any kind of branching PTX instructions.

2. Data that is read locally and in an uncoalesced way shall be placed in texture memory provided it is not written.

3. Avoid division if possible and use the less precise, faster type (div.approx, dif.full instead of div.rnd).

4. If the denominator is used multiple times calculate inverse value and mul-

DOI:10.15774/PPKE.ITK.2014.005

tiply with it.

Rules 1, 3 and 4 have impact on the first part of the algorithm and the Rule 2 applies to the second part. A conditional statement can be realized with two different assembly constructs. The more general is the conditional jump. All if-else statements are compiled to conditional jumps. This instruction causes first a 12 CLK waiting delay and then the different code paths are serialized out. However, the other possibility is less flexible, the select (ternary) instruction is executed within a single CLK.

Although divisions are unavoidable, their optimized usage have major impact on the performance of the first part especially on the two newer GPUs. The two older devices have two different divisions, a basic and a faster one. The basic has 60 CLK while the fast has a 40 CLK delay. The two newer devices are based on the Fermi architecture that is capable of IEEE compliant floating point operations (addition, multiplication, division, rounding modes, etc.) as well. The delay of an IEEE compliant division is several hundred CLKs. The second rule effects the main loop, more precisely its efficiency.

Listing 2.1: Main loop inside the kernel: line integral approximation along a ray. 'integ' is the integral on the voxel intensities traversed, 'pos' is the actual position inside the volume, its initial value is on the volume surface, 'dir' is a voxel sized direction vector. 'Image3D' is the 3D texture with linear interpolation and 'tex3D' is a built in texture reading function. 'C' and 'K' are scaling constants corresponding to the scanning protocol. They give a linear approximation of the mapping between Hounsfield Unit and attenuation coefficient of the given voxel on the X-ray hardness defined by the scanning protocol

```
float integ = 0.0f;
for(int j = 0; j < int(ray_length); ++j)
{
 integ += tex3D(Image3D, pos.x, pos.y, pos.z);
 pos.x += dir.x;
 pos.y += dir.y;
 pos.z += dir.z;
}
```

DOI:10.15774/PPKE.ITK.2014.005

```
PixelValue[threadID] = exp(−C∗integ + K);
```

2.3.3 Data and measurement

The rendering plane is chosen to be 300×300 mm^2 with a resolution of 750×750. An ROI size of 160×90 mm^2 (400 × 225 pixels) is selected within it. The ROI is sampled randomly: the locations of the pixels are chosen by a 2D uniform distribution. Several sampling ratios ($1.1 - 9.1, 11 - 44\%$) and full sampling are investigated: rendering of 1024, 1536, 2048, 3072, 4096, 6144, 8192, 10240, 20480, 30720, 40960 and 90000 pixels (full sampling, 400 × 225 pixels). This last case is referred to as full ROI DRR. Each pixel intensity is calculated by one thread on the GPU. So the number of pixels are equal to the number of threads launched on the device.

The measurements can be divided into two sets. The first set is done on four GPUs (8800 GT, 280 GTX, Tesla C2050, 580 GTX), the used GPU compiler and driver version was 3.2 and 260.16.19, respectively. The hosting PC contained an Intel Core2 Quad CPU, 4GB of system memory running Debian with Linux kernel 2.6.32. In this case two datasets were used a CT scan (manufactured by GE Healthcare, CT model Light Speed 16 see Figure 2.5(a)) of a radiological torso phantom (manufactured by Radiology Support Devices, Newport Beach, CA, model RS-330 see Figure 2.5(b)) and a scan from an annotated data set [42]. The former is referred to as phantom dataset and the latter is referred to as pig dataset. The resolution of the reconstructed image was 512 × 512 × 72 with data spacing (0.521 mm, 0.521 mm, 1.25 mm) in the case of the phantom dataset. Its dimensions are regular for spine surgery aided with 2D to 3D image registration. The reconstructed image of the pig dataset has the following dimensions: 512 × 512 × 825 with data spacing (0.566 mm, 0.566 mm, 0.4 mm). In the first set of measurements only the block size dependence of the optimized kernel using all rules was measured. If the sampling ratio is below 10% the phantom dataset is used since this scenario is relevant for 2D to 3D registration. If the sampling ratio was above 10% the pig dataset was used. These measurements show clearly the block size characteristics of the GPUs.

The second set of the measurements is done on two GPUs (Tesla C2050, 570

DOI:10.15774/PPKE.ITK.2014.005

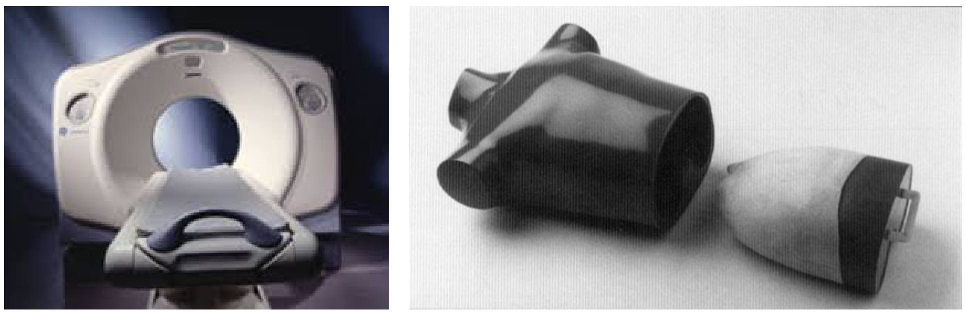

(a) (b)

Figure 2.5: Illustration of (a) CT scanner [43] and (b) radiological phantom [44]

GTX), the used compiler and driver was 5.5 and 331.67, respectively. The hosting PC contained an Intel Core i7 CPU, 8GB of system memory running Debian with Linux kernel 3.12. In this case only the phantom dataset was used. This set of measurements highlights the impact of the rules presented in Section 2.3.2 on the DRR rendering kernel performance.

The pixel locations were resampled for each kernel execution. Similarly, for each kernel execution the initial reference pose of the CT volume was varied (perturbed) in the range of ± 20 mm and ± 15 deg by uniform distribution. The perturbation of the volume pose and the resampling of the pixel locations mimic the repetitive DRR rendering need of a 2D to 3D registration process. It shall be noted that other results [13, 18, 12] showed that 2D to 3D image registration algorithms can robustly converge with good accuracy even if only a few percent of the pixels are sampled randomly.

2.4 Results

First, I demonstrate the performance gain caused by the rules described in subsection 2.3.2 in consecutive order. This is presented together with the results from the optimization of the block size on a recent version of the GPU compiler and driver. Then the block size dependence is demonstrated on an old version of the GPU driver showing that this characteristic appears regardless of the driver or the GPU.

DOI:10.15774/PPKE.ITK.2014.005

Table 2.3: Impact on kernel performance using rules 1 and 2 described in subsection 2.3.2. Kernel execution times are average of 100 executions and the unit μs. Columns t_{opt} present execution times of kernels using all rules. Columns t_{branch} present execution times of kernels using all rules but rule 1. Columns t_{linear} present execution times of kernels using all rules but rule 2.

	Tesla c2050			570 GTX		
# of pixels	t_{opt}	t_{branch}	t_{linear}	t_{opt}	t_{branch}	t_{linear}
1024	234	258	553	181	206	408
1536	319	339	639	263	295	462
2048	466	490	1094	358	403	656
3072	648	689	1275	572	617	1101
4096	969	1112	1935	693	722	1310
full DRR	2666	2763	5278	2259	2375	5221
10240	2307	2560	4591	1739	1843	3738
20480	4469	4539	8623	3359	3728	6012
30720	6515	6971	13766	4961	5357	9912
40960	8808	9249	19465	6571	7359	13026

The results of the first two rules are presented in Table 2.3. The missing branching optimization resulted in a 6-13% performance decrease, 8% in average if the optimized version is considered 100%. The linear memory caused a 1.75-2.4 times slowdown consequently on both GPUs based on the Fermi architecture nearly independent from the block size and number of threads.

Rules 3 and 4 can be measured effectively only together so, in the following these rules are covered together with the block size dependence of the execution time. In Figures 2.6-2.13 the execution time dependence of division pattern and block size on Tesla C2050 GPU and GTX 570 GPU. Both GPUs show similar characteristics. First of all, the gain is 2.3 if the best result of the optimal kernel is compared to the best result of the kernel with bad division pattern in the case of 1024 threads. This ratio is in the range of 1.92-2.36 on the Tesla C2050 GPU

DOI:10.15774/PPKE.ITK.2014.005

and in the range of 1.75-2.25 on 570 GTX GPU.

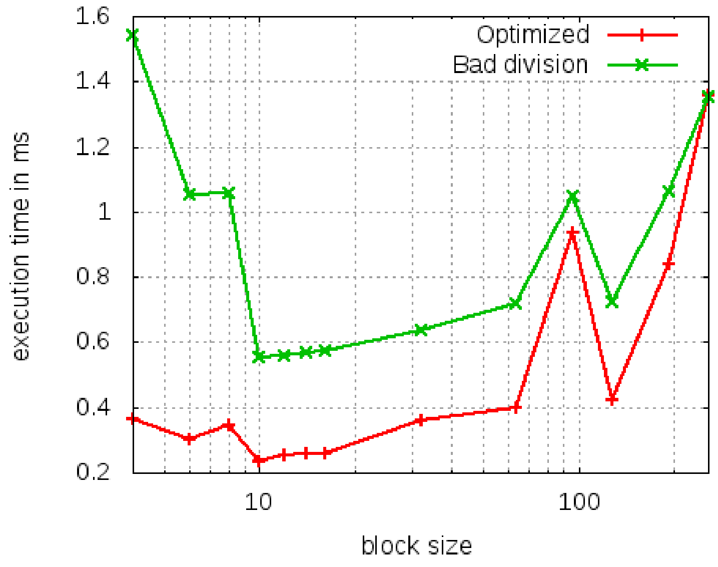
From 1024 to 3072 threads (see Figures 2.6-2.9) the performance of the kernel using the unoptimized division pattern is lower or equal to the optimized kernel independently from the block size. These are the cases when the SMs of the GPUs are not filled completely. However, these are the cases that are used in 2D-3D registration in most cases.

In the case of larger number of threads there is a range where the bad division pattern performs better than the optimized one provided the same block size is used (see Figures 2.10-2.13). The reason is as follows. In all cases the execution time is built up of two dominating components and the block size has completely the opposite effect on them. The first component is the pack of division operations in the first part of the algorithm and the second one is the repetitive texture fetch operation in the main loop. If the block size increases the effectiveness of the divisions increases as well (see the identical nature of the first part of the green lines in Figures 2.10-2.13). In case the block size decreases to the optimum the effectiveness of the texture fetch with this reading pattern increases. The weight of the two component is different in the case of the optimized division pattern compared to the unoptimized division pattern, the execution time curve is shifted as well. Since the weight of the divisions decreased in the case of the optimized pattern the direction of the shift is towards the smaller number of threads in a block.

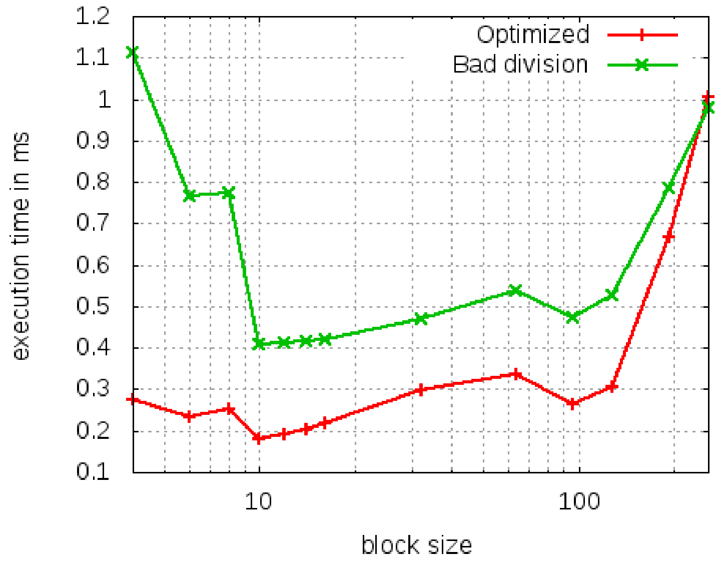
My previous results showed similar characteristics [32] on two additional hardwares with older compiler and driver. These results are presented in Table 2.4 and referenced in Table 2.1.

On 8800 GT GPU optimal block size is 8 in all cases. The increase of the execution time with respect to the optimal execution time is in the range of $57,3 - 116,8\%$ the mean of the increase is 82%. On 280 GTX GPU optimal block size is 8 in all cases. The increase of the execution time with respect to the optimal execution time is in the range of $8,3 - 27\%$ the mean of the increase is 18.7%. On Tesla C2050 GPU optimal block size varies from 10 to 16. The increase of the execution time with respect to the optimal execution time is in the range of $5 - 23\%$, the mean of the increase is 9.3%. On 580 GTX GPU optimal block size varies from 8 to 16. The increase of the execution time with

DOI:10.15774/PPKE.ITK.2014.005



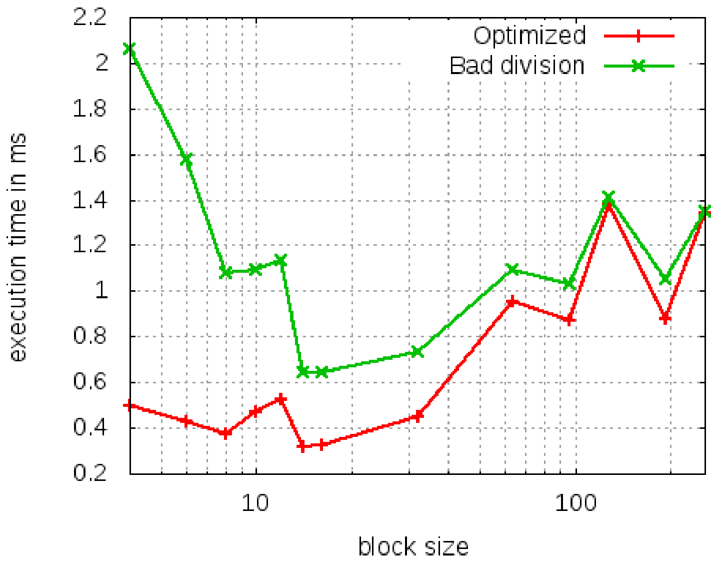
(a)



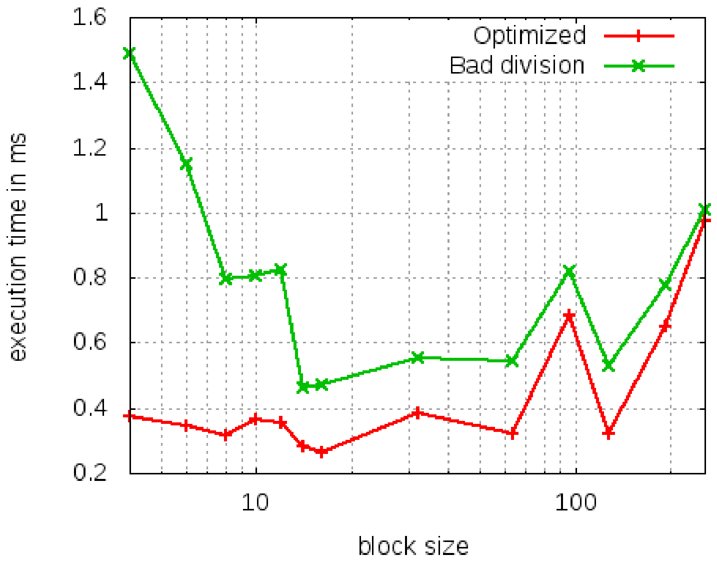
(b)

Figure 2.6: Execution time dependence on division pattern and block size n the case of 1024 threads. On the x axes there is the block size and the y axes is the execution time in ms. Red curve corresponds to mean of measurements applying all rules, while green curve corresponds to measurements applying only rules 1 and 2. (a) Shows characteristics in the case of Tesla C2050 GPU. (b) Shows characteristics in the case of 570 GTX GPU.

DOI:10.15774/PPKE.ITK.2014.005



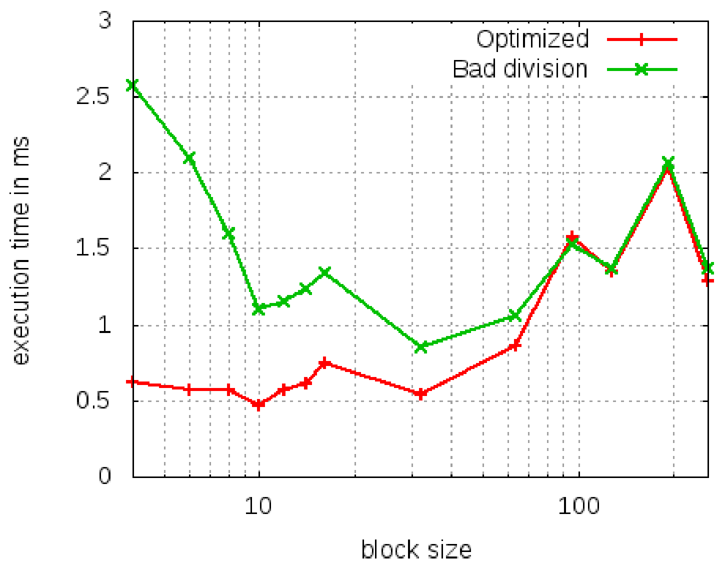
(a)



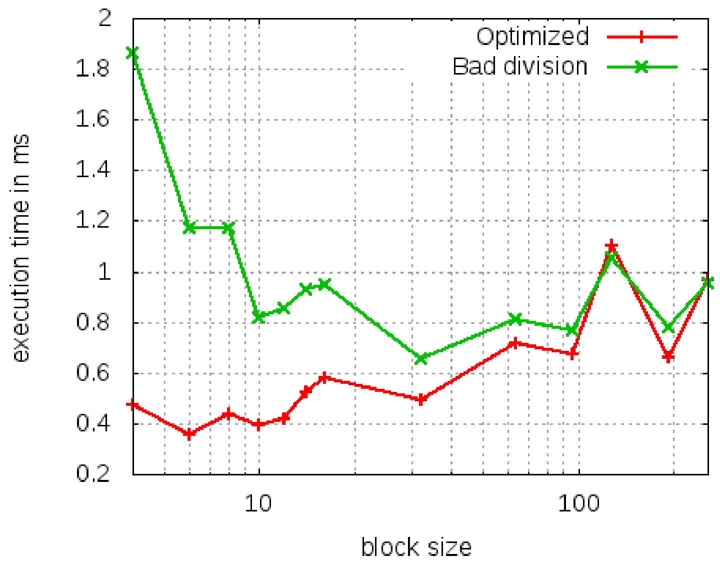
(b)

Figure 2.7: Execution time dependence on division pattern and block size n the case of 1536 threads. On the x axes there is the block size and the y axes is the execution time in *ms*. Red curve corresponds to mean of measurements applying all rules, while green curve corresponds to measurements applying only rules 1 and 2. (a) Shows characteristics in the case of Tesla C2050 GPU. (b) Shows characteristics in the case of 570 GTX GPU.

DOI:10.15774/PPKE.ITK.2014.005



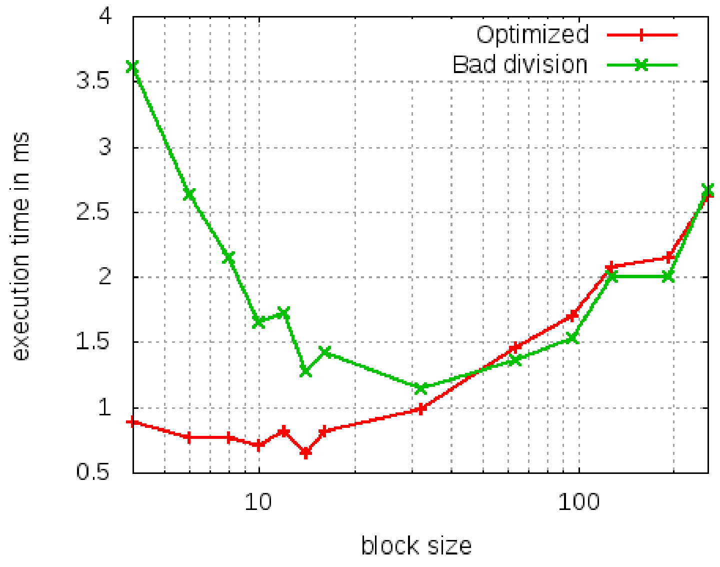
(a)



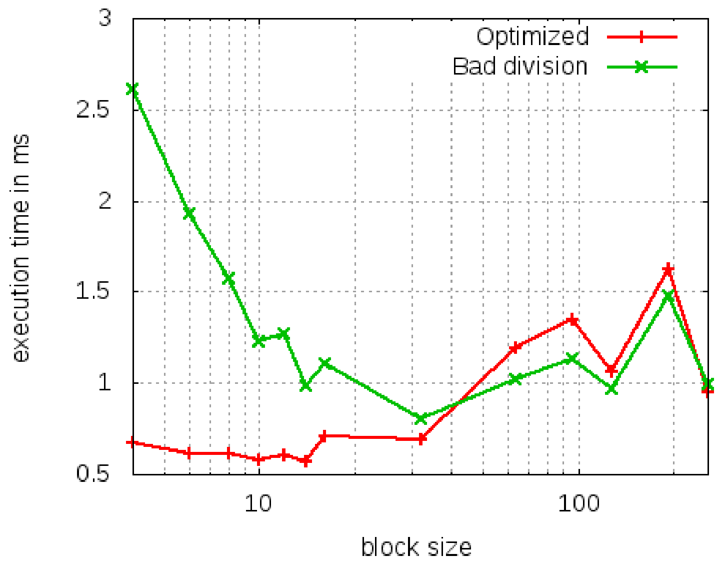
(b)

Figure 2.8: Execution time dependence on division pattern and block size n the case of 2048 threads. On the x axes there is the block size and the y axes is the execution time in ms. Red curve corresponds to mean of measurements applying all rules, while green curve corresponds to measurements applying only rules 1 and 2. (a) Shows characteristics in the case of Tesla C2050 GPU. (b) Shows characteristics in the case of 570 GTX GPU.

DOI:10.15774/PPKE.ITK.2014.005



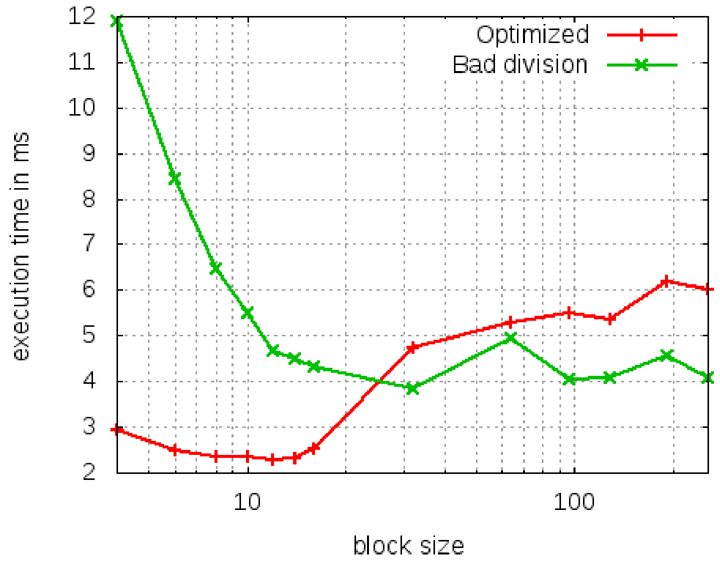
(a)



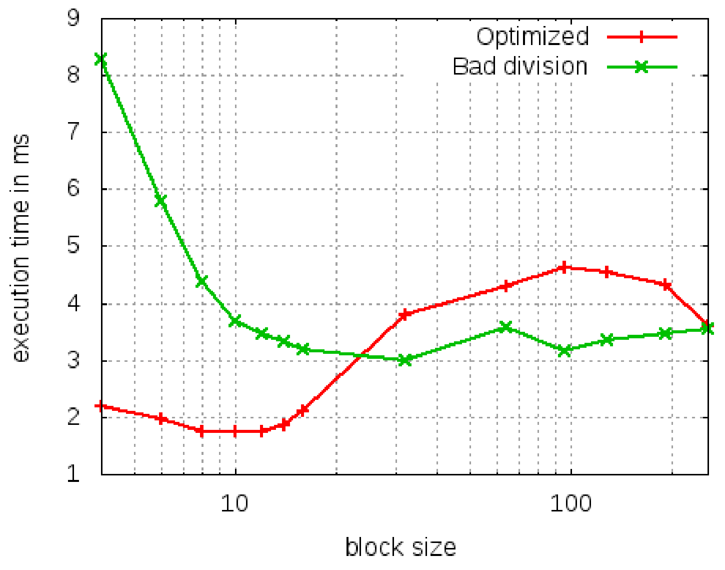
(b)

Figure 2.9: Execution time dependence on division pattern and block size n the case of 3072 threads. On the x axes there is the block size and the y axes is the execution time in ms. Red curve corresponds to mean of measurements applying all rules, while green curve corresponds to measurements applying only rules 1 and 2. (a) Shows characteristics in the case of Tesla C2050 GPU. (b) Shows characteristics in the case of 570 GTX GPU.

DOI:10.15774/PPKE.ITK.2014.005



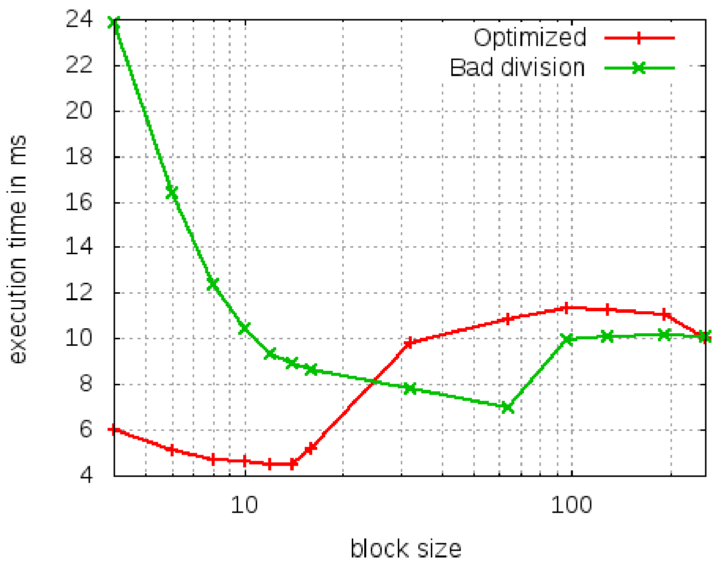
(a)



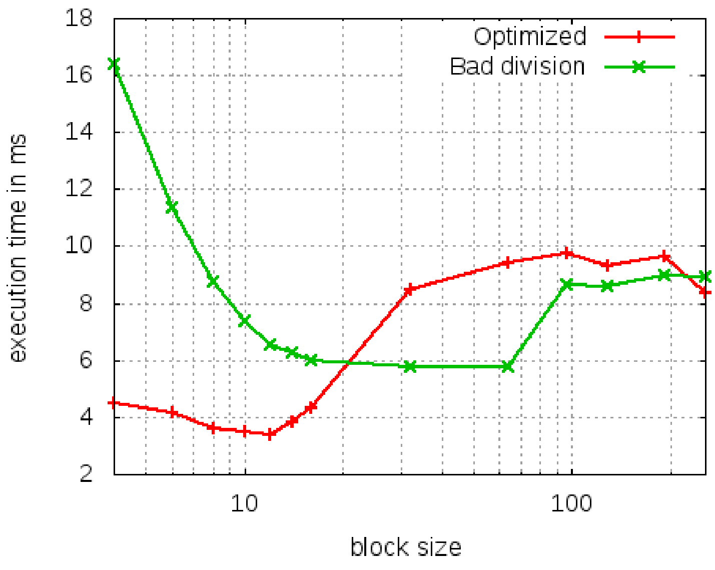
(b)

Figure 2.10: Execution time dependence on division pattern and block size in the case of 10240 threads. On the x axes there is the block size and the y axes is the execution time in ms. Red curve corresponds to mean of measurements applying all rules, while green curve corresponds to measurements applying only rules 1 and 2. (a) Shows characteristics in the case of Tesla C2050 GPU. (b) Shows characteristics in the case of 570 GTX GPU

DOI:10.15774/PPKE.ITK.2014.005



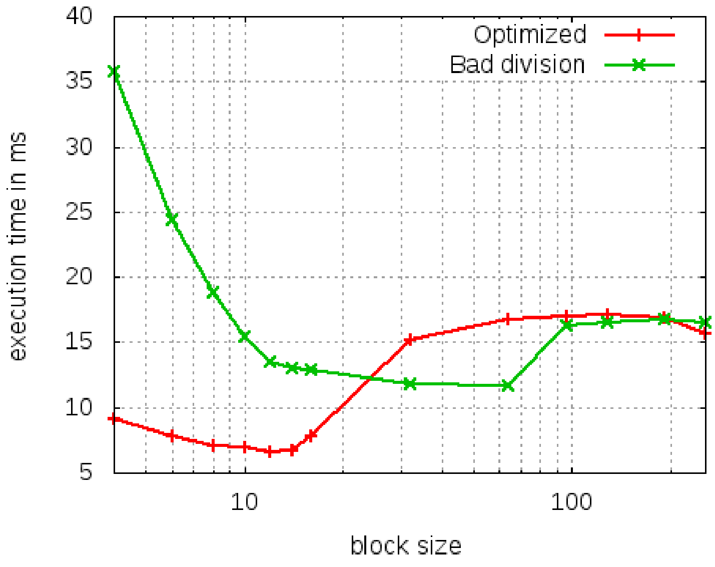
(a)



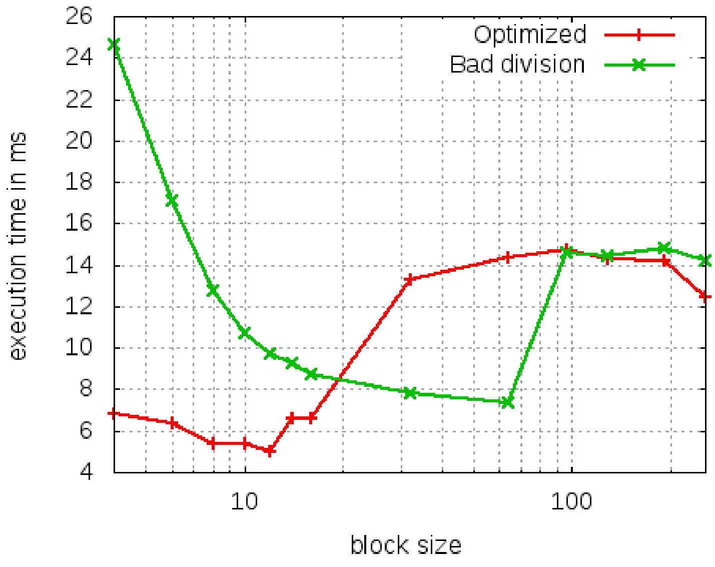
(b)

Figure 2.11: Execution time dependence on division pattern and block size in the case of 20480 threads. On the x axes there is the block size and the y axes is the execution time in ms. Red curve corresponds to mean of measurements applying all rules, while green curve corresponds to measurements applying only rules 1 and 2. (a) Shows characteristics in the case of Tesla C2050 GPU. (b) Shows characteristics in the case of 570 GTX GPU

DOI:10.15774/PPKE.ITK.2014.005



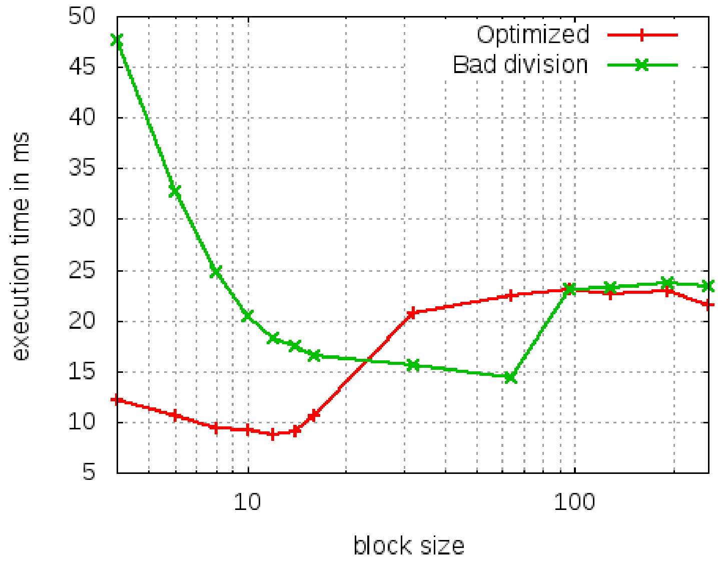
(a)



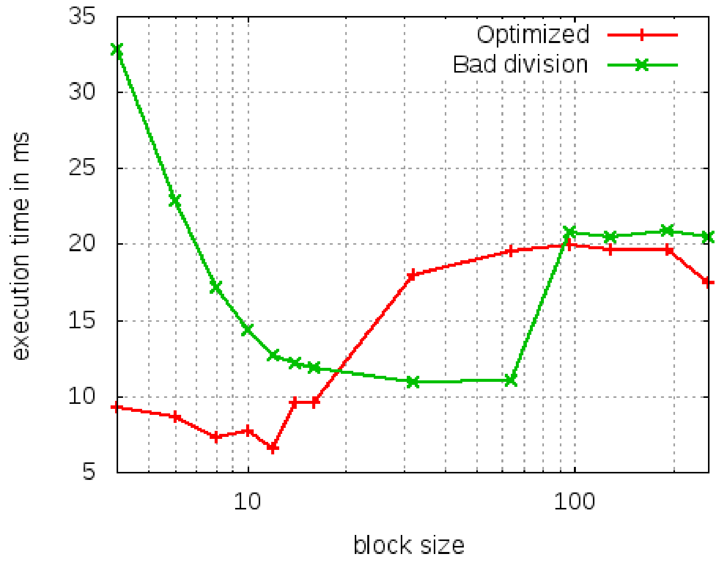
(b)

Figure 2.12: Execution time dependence on division pattern and block size in the case of 30720 threads. On the x axes there is the block size and the y axes is the execution time in ms. Red curve corresponds to mean of measurements applying all rules, while green curve corresponds to measurements applying only rules 1 and 2. (a) Shows characteristics in the case of Tesla C2050 GPU. (b) Shows characteristics in the case of 570 GTX GPU

DOI:10.15774/PPKE.ITK.2014.005



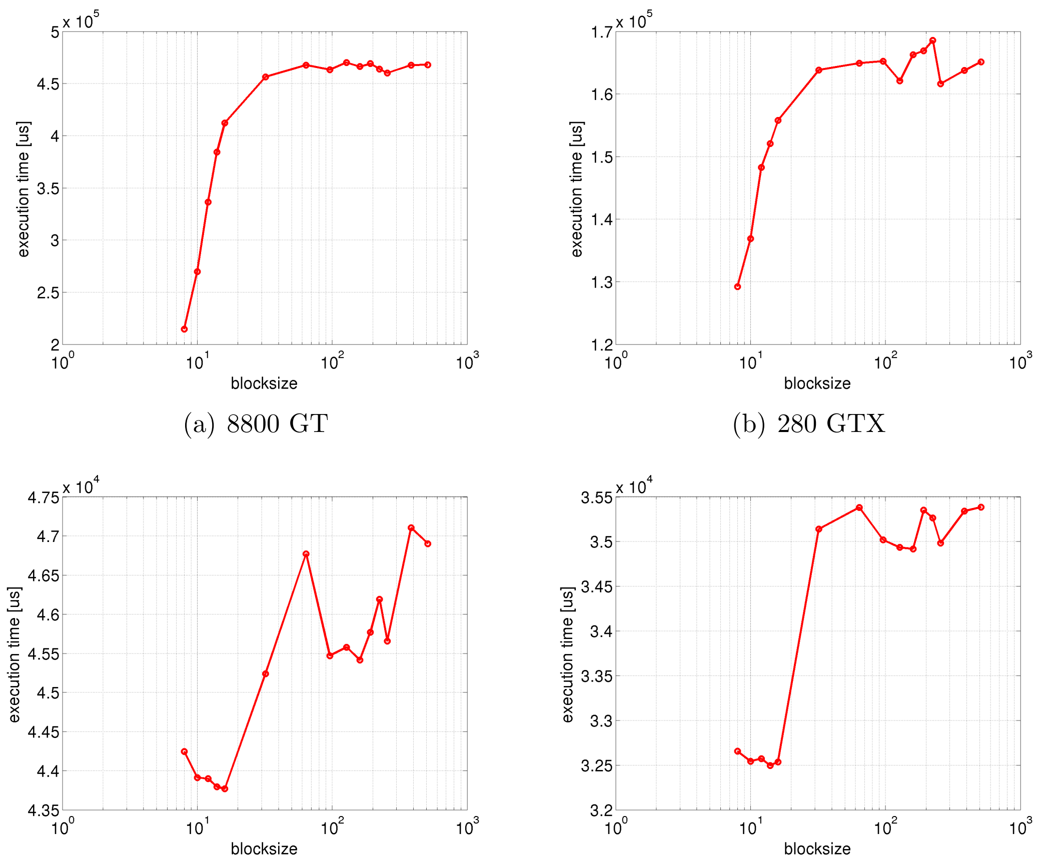
(a)



(b)

Figure 2.13: Execution time dependence on division pattern and block size in the case of 40960 threads. On the x axes there is the block size and the y axes is the execution time in ms. Red curve corresponds to mean of measurements applying all rules, while green curve corresponds to measurements applying only rules 1 and 2. (a) Shows characteristics in the case of Tesla C2050 GPU. (b) Shows characteristics in the case of 570 GTX GPU

DOI:10.15774/PPKE.ITK.2014.005



(a) 8800 GT

(b) 280 GTX

(c) Tesla C2050

(d) 580 GTX

Figure 2.14: Large thread numbers (40960) on 8800 GT, 280 GTX, Tesla C2050, and 580 GTX in the case of the Pig dataset. The execution time of the optimized kernel is depicted as a function of the block size. The results of the four GPUs can be seen. It is clear that the best block size is in the range of 8-16. This is an unexpected result since the physical scheduling of threads is made in warps (32 threads).

DOI:10.15774/PPKE.ITK.2014.005

Table 2.4: Optimized execution characteristics on old compiler and driver [32]. Columns 't_o' represent the means of optimized execution times of DRR computing kernel in μs. Columns 'bs' show optimized block sizes for device and thread number pairs. Columns 'SU' present speedup of execution times compared to naive block size of 256.

	8800 GT			280 GTX			Tesla c2050			580 GTX		
# of pixels	t_o	bs	SU	t_o	bs	SU	t_o	bs	SU	t_o	bs	SU
1024	1257	160	1.12	547	96	1.61	417	10	2.44	297	8	2.44
1536	1588	10	1.25	826	160	1.3	510	14	1.97	342	16	2.07
2048	2128	14	1.37	1144	12	1.1	690	32	1.45	391	16	1.78
3072	3016	8	1.34	1480	8	1.3	886	32	1.92	550	192	1.21
4096	3922	10	1.56	1922	10	1.32	1159	64	1.57	700	128	1
6144	5815	16	1.59	2641	10	1.51	1508	64	1.64	1006	192	1.3
8192	7300	32	1.09	3424	10	1.36	2222	128	1.31	1290	256	1
full ROI	5269	128	1.34	4545	32	1.31	3989	128	1.1	2666	128	1.17

respect to the optimal execution time is in the range of $8.2 - 14.8\%$ the mean of the increase is 11.1%.

2.5 Discussion

In this Chapter, 4 rules are presented that proved to be an essential aid to optimize a fast DRR rendering algorithm implemented in C for CUDA on more contemporary Nvidia GPUs. Furthermore, a significant new optimization parameter is introduced together with an optimal parameter range in the presented case. The presented rules include arithmetic, instruction and memory access optimization rules as well. The performance gain is presented corresponding to the 4 rules as well as to the block size. For thread numbers required to the registration (1024-3072) all rules caused performance gain independently from the block size. However, outside this interval, gain from the division pattern vanishes and changes to loss if the block size is not taken into account together with the division pattern. Namely, for the same block size the execution time is better in the case of unoptimized division pattern than in the case of optimized division

DOI:10.15774/PPKE.ITK.2014.005

pattern (see rules 3-4 in Section 2.3.2). It is illustrated in Figures 2.10-2.13. A likely explanation is given for this phenomenon in Section 2.4.

I have showed through several experiments that the block size is an important optimization factor and I have given the interval (between 8 and 16) where it resides in nearly all cases for randomly sampled DRR rendering independently from the hardware. These optimized values differ from values suggested by the vendor in nearly all cases. Additionally, the same interval has been determined with similar characteristics on an old compiler and driver combination on two other older GPUs as well. This indicates that this characteristic is independent from the compiler and the driver and it is an intrinsic property of the NVIDIA GPUs. Further measurement data is provided in the Appendix C.

The ray cast algorithm is embarrassingly parallel: the pixels are independent from each other and similarly, integrals of all disjoint segments of a ray are independent too. Another advantage of the algorithm is its independence from the pixel and virtual X-ray source locations. The performance bottleneck of the algorithm is its bandwidth limited nature. For each voxel read instruction there are only four floating point additions. There are possibilities to improve even further the execution time. Line integral of disjoint segments can be computed independently. This enables the complete integral of one pixel to be calculated by one or more blocks. A block works on a segment that can be either the complete line inside the volume or a fraction of it. In the case of full ROI DRRs the block and grid size can be chosen to be 2D, so a block renders a small rectangle of the ROI. This arrangement may be more effective, since the locality is better than in the 1D case, which was used in this work. Completely different approaches can not be much faster in the random case because rendering is bandwidth limited.

The presented results outperform a similar attempt from the literature [31, 30]. The comparison is easier to the work of Dorgham et al. [31]. In one case nearly the same (8800 GT and 8800 GTX) and in another the same (580 GTX) GPUs were used. Both the 3D data and the number of rendered pixels are in the same range (512×512×267 vs 512×512×72 in the case of 3D data and 512×267 vs 400×225 in the case of number of pixels). Furthermore, the GPU compiler and driver is assumed to be the same because of the date of the publication. After normalizing by the ratio between the 3D data and the number of pixels a 5.1

DOI:10.15774/PPKE.ITK.2014.005

times speedup appears in the case of 8800 GT GPU and 1.81 times speedup in the case of 580 GTX GPU [32]. If different compiler and driver is allowed than the result of Dorgham et al. on 580 GTX can be compared to the the result of 570 GTX GPU (see Table 2.3) normalized with the number of SM-s. The speedup is 2.33.

The comparison to the work of Gendrin et al. [30] is hard since, we have only implicit information on the speed of the DRR rendering. It presents an on-line registration at the speed of 0.4-0.7s. However, the 3D CT volume is preprocessed by (a) intensity windowing (b) and the unnecessary voxels are cut out. The windowing eliminates pixels under and above proper thresholds and maps the voxel value to a 8 bit range. Furthermore, not only the ROI is remarkably smaller than in our case but the projected volume as well. Unfortunately, there is no precise information about the reduced volume size making the exact comparison hardly possible.

2.6 Conclusions

Execution time optimization is in the heart of real time applications. Finding optimization rules and optimal parameters is a non-trivial task. I showed that the rules I defined are indeed effective optimization rules in several important and relevant cases on more GPU hardware. I emphasized the effect of block size on the performance. Furthermore, I determined its optimal range for the DRR rendering. Of course, these results should help in any other cases when the task contains calculation of random projection.

To automatically register the content of an X-ray projection to a 3D CT, 20-50 iteration steps are required. For each iteration, 10-20 DRRs are computed depending on the registration procedure. On the whole this amounts to 200-700 DRRs to be rendered for a registration to converge. DRR rendering is the most time consuming part of the 2D to 3D image registration. Following the presented implementation rules, the time requirements of a registration process can be decreased to 0.6-1 s if full-ROI DRRs are applied. If random sampling is used the time requirement of registration can be further reduced to *0.07-0.5 second* resulting in quasi real time operability. This achievement allows new services and

DOI:10.15774/PPKE.ITK.2014.005

protocols spread in the practice in the fields where real time 2D to 3D registration is required like patient position monitoring during radiotherapy, device position and trajectory monitoring and correction during minimally invasive interventions. The code-base was integrated into a prototyping framework of GE. As for my last information the company considered the possibility to use the module in later upcoming softwares.

DOI:10.15774/PPKE.ITK.2014.005

Chapter 3

Initial condition for efficient mapping of level set algorithms on many-core architectures

This Chapter is organized as follows. Section 3.1 summarizes the related work. Section 3.2 describes the interface propagation in an introductory level and presents the fast LS method of Shi. Section 3.3 gives the necessary definitions and tools to handle rigorously the theoretical results presented in Section 3.4. It is followed by Section 3.5 presenting the proofs of the Theorems. This part of the dissertation focuses on the initial condition and its impact on the evolution in both theoretical and practical ways. The theoretical part is covered in Sections 3.4 and 3.5. The context of the practical side is laid down in Section 3.7 namely it shows the conventional and the proposed initial condition families. This Section helps to understand the significance of the results. Section 3.6 describes the two hardware platforms namely, a mixed mode CNN-UM implementation and a GPU, that executed the two case studies described in Section 3.8. This Section also presents some examples of the Theorems and demonstrates a segmentation example. It is followed by Section 3.9 comparing the Shi LS evolution against a numerical PDE approximation in three cases using different force fields. Section 3.10 gives the discussion and Section 3.11 concludes this Chapter.

The use of Level Set (LS) based curve evolution has become an interesting

DOI:10.15774/PPKE.ITK.2014.005

research topic due to its versatility and accuracy. These flows are widely used in various fields like computational geometry, fluid mechanics, image processing, computer vision and material science [6]. In general, the method entails that one evolves a curve, surface or image with a partial differential equation (PDE) and obtains the result at a point in the evolution.

There is a subset of problems where only the steady state of the LS evolution is of practical interest like segmentation and detection. In this Chapter, only this subset is considered. In addition, I do not form any operator or force field (F) for driving the evolution of the LSs. However, two theoretically worst case bounds of the required number of iterations are proposed to reach the steady state for a well defined class of LS based evolution. These bounds depend only on the initial condition. Furthermore, the bounds only allow an extremely small number of iterations if the evolution is calculated with a properly chosen initial condition. These kinds of evolutions are calculated very quickly on many-core devices.

The subject of this Chapter is both theoretical and practical. The theoretical side is clearly two new Theorems in the worst case of the required number of iterations of the LS evolution of [45]. This evolution omits the numerical solution of the underlying PDE and successfully approximates it with a rule based evolution. It is based on the sign of the force fields (F) normal to the curves to be to change. Theorem 1 gives a general bound and Theorem 2 assumes a special kind of discrete convexity defined in Section 3.3.

The practical side is presented through two case studies, namely, the LS evolution of Shi can be mapped in a straightforward way on two completely different many-core architectures. With a lot of small curves in the initial condition, which would be unfeasible on a conventional single core processor, the proposed Theorems ensure small number of iterations. Additionally, with the change in the initial condition (instead of one curve, a lot of small curves are used) the computing width of the many-core platform is utilized.

3.1 Related Work

The first successful method to speed up the LS evolution was introduced by [46]. It introduces the narrow band technique. The original LS method required cal-

DOI:10.15774/PPKE.ITK.2014.005

culations over the entire domain, while the narrow band method constructs a narrow band (also called tube) around the zero LS of ϕ and restricts the numerical solution of the LS PDE to this band. If the zero LS reaches the boundary of the narrow band a new tube is constructed. A local method was proposed [47] with better big O characteristics. Both methods are labeled as narrow banding methods.

However, I am not presenting any PDE operators and do not design any force field. Instead, I direct the reader to the classical book of Sapiro [7] who gives a detailed picture from the art of PDE operator design for a given purpose. Furthermore, a short summary is given here which gives a picture of this field. In general two approaches are possible; (i) an energy functional is constructed and this functional is minimized; or (ii) the equation is formed following certain physical rules. The two approaches can replace each other if some conditions are met. There are several results [48, 49, 50] regarding edge, region and model based evolutions. Edge based methods use implicitly the gradient to drive the evolution of the curve [51, 52]. Unfortunately, the capture range of these methods are rather small and require close initialization. Region based evolutions are driven not by the gradient but by the intensity and this is corrected by regularization terms [48]. Model-based approaches have an initial a priori information on the object and incorporate this to the energy functional [53]. It can be seen that energy functions evolved from the simplest gradient to the more complicated quantities. This has two reasons: (i) more and more complicated images are processed and segmented and (ii) the evolution may be trapped in a local minima. There are efforts to maintain the second reason. It has been shown that the evolution can depend on the chosen metric [54] and that the classical scalar product based L_2 space is unsuitable for shape analysis. In [55] Sobolev norm was used instead of the unsuitable L_2 norm and showed that this norm allows new energies to implement otherwise considered unfeasible due to convergence or other problems.

There are multiple results reporting successful mapping of various curve evolution methods to many-core platforms. The first attempt to map an LS evolution to graphics hardware was presented in [56]. In this work the full LS model in 2D was solved without regularization term and the operations had to be cast to graphics rendering pipeline primitives. In [57] the authors presented an inter-

DOI:10.15774/PPKE.ITK.2014.005

active 3D sparse solver for GPU. This realization uses again the graphics API (OpenGL) and the rendering pipeline. The specialty of this work is the possibility to interactively tune many parameters of the evolution. Two later works [58, 59], applied the computing unified device architecture (CUDA) of NVIDIA. Both papers worked with 3D volumes. The work in [58] mapped a sparse solver while others [59] used a higher order scheme to evolve the LSs. Cellular Neural Networks (CNN) [60] proved to be an inspiring construct. There have been results regarding the mapping of LS like evolutions to CNN [61, 62, 63]. In [61] the authors successfully mapped a nonlinear, global histogram modification operator to local nonlinear CNN dynamics. The PDE was discretized in space and was converted to coupled nonlinear ODEs. The histogram modification was combined with embedded morphological processing to get a smooth result. Later, [63] realized an on-line boundary detection algorithms, called topographic cellular active contours based on curve evolution to extract the volume of the right atrium. More specifically, three types of evolutions are realized on a CNN-UM ASIC implementation (ACE-16k). These methods are partially based on the fundamental work of Kass et al. [64] and on the LS evolution [65, 66]. These papers and results indicate that various LS evolutions can be mapped and used on different many-core platforms. In this Chapter, I'm focusing on a given type of evolution and for this evolution I give two Theorems upper bounding the required number of iterations of the evolution process.

3.2 Background theory of LS

I present here the formulation of boundary value and initial value PDE which describe the interface motion. These formulations could lead to two efficient schemes, to the Fast Marching Method and to the Narrow Band Level Set Method. However, I focus here on the theoretical aspects and only the LS based formulation is discussed in details. Additionally, some computational advantages are summarized. Later the evolution method of Shi [45] is described that omits the numerical approximation of the underlying PDE and uses a rule based approach. Since my work is based on his result, this method is covered in more detail.

DOI:10.15774/PPKE.ITK.2014.005

3.2.1 Formulation of interface propagation

Consider a boundary, a curve in two dimensions or a surface in three dimensions separating two regions. Imagine that this curve is modified by a force field F. The goal is to track the motion of the interface during the evolution. If F has both tangential and normal component then only the normal component plays role in deforming the shape of the interface. It shall be noted that F changes the curve meaning its parametrization and shape as well, but the tangential component changes only the parametrization and the normal only its shape. The proof is simple and based on the chain rule, for details see chapter 2 of [7]. The force field F, may depend on many factors, can be written as:

$$F = F(L, G, I) \tag{3.1}$$

where L, G and I stand for local, global and independent properties. Local geometric features are curvature, normal direction, etc. Global properties are those that depend on the shape and position of the front. For example it may incorporate terms with integrals along the front and associated equations. Independent properties are those that are independent of the space of the front such as underlying fluid velocity that passively convects, transports the front.

A large part of the challenge in these problems is to construct an adequate force field F or energy function E to be minimized. This is a separate problem that will not be discussed in this dissertation. I direct the interested reader to [6, 7] and other works of Osher, Malladi, Mumford, Sethian and Sapiro.

Let us fix for a moment $F > 0$. Than the front moves always outward. A possible way to characterize the position of the interface is to extract it from the arrival time of each position. Since the sign of F is fixed, the arrival time is unique and it is a function. Using the simple fact that $distance = rate * time$, I have got:

$$dx = F dT, \qquad 1 = F\frac{dT}{dx} \tag{3.2}$$

In multiple dimensions ∇T is orthogonal to the LSs of T, so:

$$|\nabla T|F = 1, \qquad\qquad T = 0 \text{ on } \gamma_0 \tag{3.3}$$

DOI:10.15774/PPKE.ITK.2014.005

where γ_0 is the initial location of the boundary. Thus, the motion is characterized as the solution of a boundary value problem. From the boundary value scenario the fast marching methods emerged as effective schemes, but these methods are not used in this thesis.

Suppose now that the front moves in both directions because there are no assumptions on the sign of the force field. So it can move over a point several times and the arrival time will not be unique and it is not a single valued function. The interface can be embedded into a higher dimensional function ϕ as its zero level set. Now the evolution of the interface is linked to the evolution of the LS function ϕ through a time dependent problem that is of initial value type. Now I have:

$$\phi(\gamma(t), t) = 0. \tag{3.4}$$

From the chain rule,

$$\phi_t + \nabla\phi(\gamma(t), t) \cdot \gamma'(t) = 0. \tag{3.5}$$

Since F is responsible for the speed in the normal direction, than $\gamma'(t) \cdot n = F$, where $n = \nabla\phi/|\nabla\phi|$. This yields to the classical LS evolution equation:

$$\phi_t + F|\nabla\phi| = 0, \tag{3.6}$$

$$\text{given } \phi(x, t = 0).$$

There are several advantages of the formulation described above. It is unchanged in higher dimensions. Topological changes in the evolving front are handled naturally since it is a LS of ϕ. This formulation relies on viscosity solutions of the associated PDE in order to guarantee the unique and entropy-satisfying weak solution. These analytical weak solutions can be approximated by computational schemes that were developed to handle hyperbolic conservation laws. The interested reader is directed to chapters 2-6 of [6] where the material summarized here is discussed in a wider extent with a mathematically rigorous way.

DOI:10.15774/PPKE.ITK.2014.005

3.2.2 Fast LS without solving PDEs

I have chosen the LS method of Shi [45] because of the following reasons. First, its memory footprint is extremely small compared to other narrow banding like algorithms. The size of the active front recalculated in every iteration cycle is the smallest, only two pixels wide. This decreases the computational pressure as well. Furthermore, the calculation does not contain any data dependent conditional branching. This fact indicates the possible effectiveness of the mapped algorithm to an arbitrary many core device and makes easier to do the *de facto* topological mapping of the algorithm.

Now the LS method of Shi [45] is summarized. This method is based on a key observation made during the analysis of the evolution of LS on regular grid. In the LS method, the curve γ is represented implicitly by the LS function ϕ. Let us assume that ϕ is defined over a domain $\mathbf{D} \subseteq \mathbf{R}^k$, where $(k \geq 2)$ and the domain is discretized into a grid. $\mathbf{D}$ may denote both the domain and the set of points from the grid.

Given the function ϕ, two sets of neighboring grid points can be uniquely defined L_{in} and L_{out} for γ as shown in Figure 3.1(a).

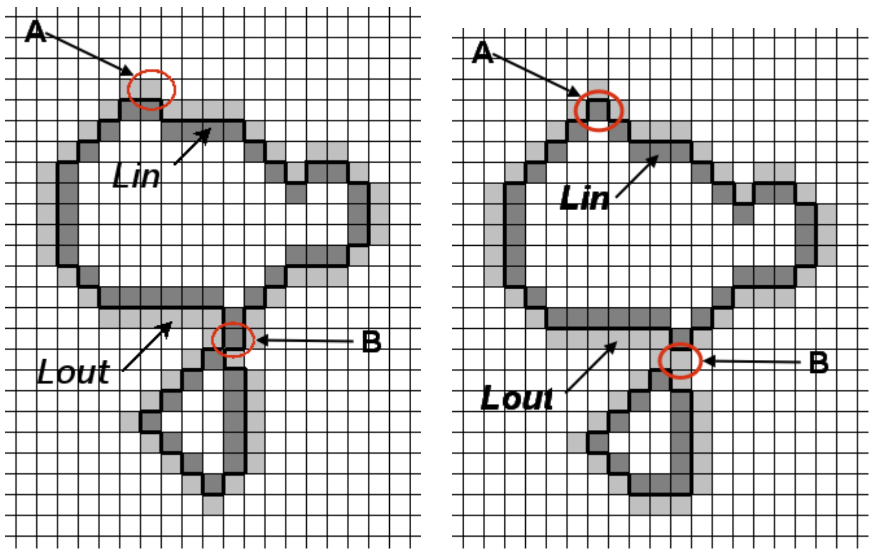
$$L_{in} = \{\mathbf{x}|\phi(\mathbf{x}) < 0 \textit{ and } \exists \mathbf{y} \in N(\mathbf{x}) \textit{ that } \phi(\mathbf{y}) > 0\}, \tag{3.7}$$

$$L_{out} = \{\mathbf{x}|\phi(\mathbf{x}) > 0 \textit{ and } \exists \mathbf{y} \in N(\mathbf{x}) \textit{ that } \phi(\mathbf{y}) < 0\} \tag{3.8}$$

where $N(\mathbf{x})$ is the discrete neighborhood of $\mathbf{x}$. As it can be seen in Figure 3.1, L_{in} is the set of neighboring grid points that are inside γ and L_{out} is the set of neighboring points that are outside. For a given γ, the choice of ϕ can be arbitrary but the two sets are uniquely defined.

To evolve a curve, one must solve numerically the underlying PDE, see Equation (3.6), according to the classical LS methods. As ϕ evolves, so does γ. This is nicely illustrated in Figure 3.1. However, at points A and B the curve moves outward and inward respectively and the corresponding values of ϕ change sign but this is done in a computationally intensive way (the PDE is solved numerically according to a proper numerical scheme and solver). The key observation is as follows. The same motion can be done by simply switching point A from L_{out}

DOI:10.15774/PPKE.ITK.2014.005



(a) before step (b) after step

Figure 3.1: Curve representation and motion by L_{in} and L_{out}. Motion of the curve can be obtained by switching points between L_{in} and L_{out}. This is done according to the sign of F at the points of the sets so the computationally intensive numerical approximation of the LS PDE is omitted.

to L_{in} and switching point B from L_{in} to L_{out} if only the final state of the zero LS is of actual interest. Based on this observation, one shall examine only the sign of the speed field F on the points of L_{in} and L_{out} and if some required conditions (described later) are met, the corresponding point is switched from one set to the other and vica versa.

$$\phi(x) = \begin{cases} -3, & \text{if } \mathbf{x} \in \Omega \text{ and } \mathbf{x} \notin L_{in} \text{ inner points} \\ -1, & \text{if } \mathbf{x} \in L_{in} \\ 1, & \text{if } \mathbf{x} \in L_{out} \\ 3, & \text{if } \mathbf{x} \in \Gamma \text{ and } \mathbf{x} \notin L_{out} \text{ outer points} \end{cases} \tag{3.9}$$

Where Ω is the object region and Γ is the background region.

For faster computation the range of ϕ is limited to $\{-3, -1, 1, 3\}$ as it is presented in Equation (3.9), similarly, the possible values of F are restricted to $\{-1, 0, 1\}$. This restriction to ϕ is a rough approximation of the signed distance

DOI:10.15774/PPKE.ITK.2014.005

function. Additionally, from the value of the LS function ϕ at a given point its location is determined relative to interface γ. Before the algorithm itself is described two procedures are defined.

The first one is called switch_in and is described in Algorithm 3.1. This procedure removes a point from L_{out}, places it in L_{in} and performs some additional operations like updating ϕ and the neighboring pixels. The second one, switch_out is depicted in Algorithm 3.2, and it is quite similar. It takes a point from L_{in}, places it in L_{out} and performs the same required additional operations.

Algorithm 3.1 Switch_in operation

```
Require: $\mathbf{x} \in L_{out}$
 1: function SWITCH_IN($\mathbf{x}$)
 2:     delete($L_{out}$, $\mathbf{x}$)
 3:     add($L_{in}$, $\mathbf{x}$)
 4:     $\phi(\mathbf{x}) \leftarrow -1$
 5:     for $\forall \mathbf{y} \in N(\mathbf{x})$ do
 6:         if $\phi(\mathbf{y}) = 3$ then
 7:             add($L_{out}$, $\mathbf{y}$)
 8:             $\phi(\mathbf{x}) \leftarrow 1$
 9:         end if
10:     end for
11: end function
```

Algorithm 3.2 Switch_out operation

```
Require: $\mathbf{x} \in L_{in}$
 1: function SWITCH_OUT($\mathbf{x}$)
 2:     delete($L_{in}$, $\mathbf{x}$)
 3:     add($L_{out}$, $\mathbf{x}$)
 4:     $\phi(\mathbf{x}) \leftarrow 1$
 5:     for $\forall \mathbf{y} \in N(\mathbf{x})$ do
 6:         if $\phi(\mathbf{y}) = -3$ then
 7:             add($L_{in}$, $\mathbf{y}$)
 8:             $\phi(\mathbf{x}) \leftarrow -1$
 9:         end if
10:     end for
11: end function
```

DOI:10.15774/PPKE.ITK.2014.005

Algorithm 3.3 Shi LS evolution

```
1: procedure EVOLVE($N_a$,$L_{out}$, $L_{in}$)
2:     $i \leftarrow 0$
3:     stopCondition $\leftarrow$ calculateStoppingCondition($L_{out}$, $L_{in}$, $i$, $N_a$)
4:     while stopCondition do
5:         calculateForce($L_{out}$,$L_{in}$)
6:         for $\forall \mathbf{x} \in L_{out}$ do                              ▷ scan $L_{out}$
7:             if $F(x) > 0$ then
8:                 switch_in($\mathbf{x}$)
9:             end if
10:        end for
11:        cleanLin()
12:        for $\forall \mathbf{x} \in L_{in}$ do                               ▷ scan $L_{in}$
13:            if $F(x) < 0$ then
14:                switch_out($\mathbf{x}$)
15:            end if
16:        end for
17:        cleanLout()
18:    end while
19: end procedure
```

Algorithm 3.4 Stopping condition for Shi LS evolution

```
1: function CALCULATESTOPPINGCONDITION($L_{out}$, $L_{in}$, $i$, $N_a$)
2:     if $i \geq N_a$ then
3:         return true
4:     end if
5:     stop $\leftarrow$ true
6:     for $\forall x \in L_{out}$ do
7:         if $F(x) > 0$ then
8:             stop $\leftarrow$ false
9:             return stop
10:        end if
11:    end for
12:    for $\forall x \in L_{in}$ do
13:        if $F(x) < 0$ then
14:            stop $\leftarrow$ false
15:            return stop
16:        end if
17:    end for
18: end function
```

DOI:10.15774/PPKE.ITK.2014.005

The pseudo-code of the main loop of the Shi LS evolution can be seen in Algorithm 3.3. At every iteration the force field for all points within the two sets are computed first. After that, the two lists are scanned sequentially to evolve the curve first outward later inward. After scanning each sets some points become interior or exterior points due to the newly added neighboring points. These points are eliminated from the sets by the two cleaning processes (see lines 11 and 17 in Algorithm 3.3). Scanning L_{out} and applying *switch_in*() moves the curve outward while scanning the other set and applying the other switching operation realizes an inward motion. The stopping condition is as follows either the predefined maximum number of iterations are reached or the curve reached a steady state namely the force field on each pixel within the active front has the correct sign, and no further motion is required. The pseudo-code is available in Algorithm 3.4. A detailed description and analysis of this method can be found in [67].

3.3 Definitions

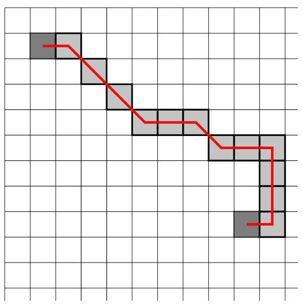
Now the necessary abstract elements are constructed and defined to be able to formulate the theoretical worst case bounds in Section 3.4. Although, the majority of the definitions are straightforward, there are some delicate differences in some of the definitions like minimum and minimal path which have great importance. Furthermore, these constructs and definitions may not completely be the same that are given in discrete topology. The definitions are nicely illustrated in Figure 3.2 and the caption describes some further details. In this Chapter these definitions are used all along.

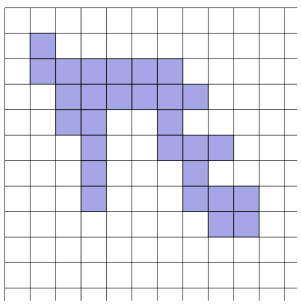
Definition 1 (path)**.** A path p between $\mathbf{x}$ and $\mathbf{y}$ is a sequence of points $\mathbf{x}_l(l = 0, 1, ..., L) \in \mathbf{D}$ subject to $\mathbf{x}_l \in N(\mathbf{x}_{l+1})$ and $\mathbf{x} = \mathbf{x}_0$ and $\mathbf{y} = \mathbf{x}_L$.

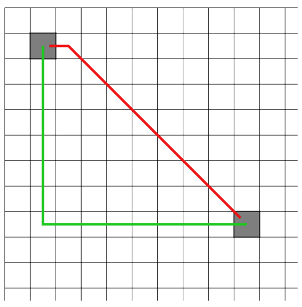
Definition 2 (connected region)**.** A set of points $\mathcal{A}$ forms a connected region if and only if there exists a path p between every $\mathbf{x}, \mathbf{y} \in \mathcal{A}$ subject to $\forall \mathbf{x}_l \in p$ is an element of $\mathcal{A}$.

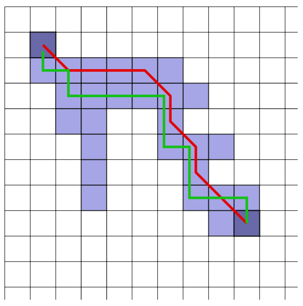
The length of a path is a non-negative integer (L) and $L = |p| - 1$, where $|.|$ denotes the number of points in the path.

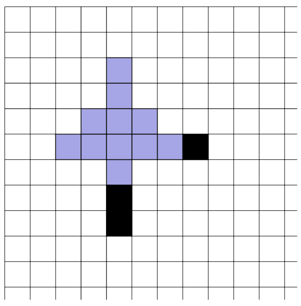
DOI:10.15774/PPKE.ITK.2014.005


(a) path


(b) connected region


(c) minimum path


(d) minimal path, diameter


(e) convex region

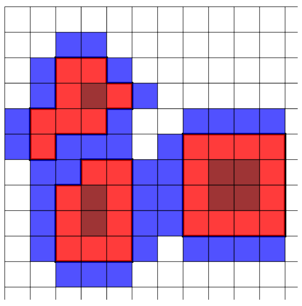

(f) configuration

Figure 3.2: The illustration of definitions: (a) shows an eight connected path (light gray) between the two endpoints (dark gray); (b) shows a connected region in blue, notice that there is at least one path from each point to all the other points; (c) shows a four and an eight connected (green, and red respectively) minimum path between the two endpoints (dark gray); (d) shows two minimal paths, each one is inside the blue connected region, furthermore, the four connected one (green) is the four connected diameter of the connected region as well; (e) shows a convex region, the blue one is an eight connected convex region while adding the black points to the blue ones the region becomes a four connected convex region; (f) shows a configuration, light red represents L_{in} points, dark red points are inner points, blue points are L_{out} points and white ones are outer points. This represents the actual state of ϕ together with Ω, the object region and Γ the background region.

DOI:10.15774/PPKE.ITK.2014.005

Definition 3 (minimum path)**.** A path p_{min} is a minimum path, if $\nexists p' \neq p_{min}$, subject to $L_{p'} < L_{p_{min}}$ and $\mathbf{x}_0^{p'} = \mathbf{x}_0^{p_{min}}, \mathbf{x}_L^{p'} = \mathbf{x}_L^{p_{min}}$.
between $\mathbf{x}$ and $\mathbf{y}$.

Minimum path is usually not unique and can depend on the chosen discrete neighborhood. The distance between $\mathbf{x}$ and $\mathbf{y}$ is a non-negative integer that is exactly the length of a minimum path between the two points. This is a real metric and is going to be referred to as d_d.

Definition 4 (minimal path)**.** Within a connected region $\mathcal{A}$, a path p between $\mathbf{x}$ and $\mathbf{y}$ is minimal if and only if $\mathcal{A} \cap p = p$ and there are no shorter p' paths within $\mathcal{A}$ between $\mathbf{x}$ and $\mathbf{y}$.

Like the minimum path, the minimal path may not be unique and may depend on the chosen neighborhood.

Definition 5. The diameter B of a connected region is the longest minimum path having at least its endpoints within the connected region.

Definition 6 (convex region)**.** A connected region is considered as convex if all minimal paths are minimum paths at the same time.

Definition 7 (configuration)**.** A configuration $C = \{\mathbf{D} \times \phi\}$ is the actual state of the LS function, namely, the shape of the zero LS and the connected regions (Ω_p, Γ_q) composing the object and the background region.

Now I have all the necessary tools to establish proper worst case bounds on the number of iterations required by the Shi LSM to converge.

3.4 Theoretical Results

Theorem 1 (general bound)**.** *Let the true object region be denoted by Ω^* and let it be composed of P connected regions Ω_p^* (where $p = 1...P$). Similarly, let the true background region be denoted by Γ^* and let it be composed of q connected regions Γ_q^* (where $q = 1...Q$). Assume that $F > 0$ in Ω^* and $F < 0$ in Γ^*. At initialization, C is chosen such that $\Omega = \cup_i \Omega_i$, $\Gamma = \cup_j \Gamma_j$ and $\Omega_p^* \cap \Omega \neq \emptyset$, $\forall p =$*

DOI:10.15774/PPKE.ITK.2014.005

1...P and $(D \setminus \Omega) \cap \Gamma^*_q \neq \emptyset$, $\forall q = 1...Q$. Then, the Shi LSM converges to Ω^* in $N_{it} \leq \max(\max_i(|\Omega_i|), \max_j(|\Gamma_j|))$ iterations, where $|.|$ denotes the number of elements in the region.

Theorem 2 (convex bound)**.** *Let the true object region Ω^* be composed of P connected regions Ω^*_p (where $p = 1...P$) and the true background region Γ^* be composed of q connected regions Γ^*_q (where $q = 1...Q$). Assume that $F > 0$ in Ω^* and $F < 0$ in Γ^*. At initialization, C is chosen such that $\Omega = \cup_i \Omega_i$, $\Gamma = \cup_j \Gamma_j$ and $\Omega^*_p \cap \Omega \neq \emptyset$, $\forall p = 1...P$ and $(D \setminus \Omega) \cap \Gamma^*_q \neq \emptyset$, $\forall q = 1...Q$. If either Ω^* or Γ^* is convex than the Shi LSM converges to Ω^* in $N_{it} \leq \max(\max_i(B_{\Omega_i}), \max_j(B_{\Gamma_j}))$ iterations, where B denotes the diameter of the given region.*

Theorem 1 gives a general upper bound on N_{it} and the iteration cycle checking the stopping condition is not necessary if the number of iterations has reached this upper bound. This worst case bound is approached if Ω^* or Γ^* are degenerated in some sense (see Figure 3.6(d) and Table 3.2 for example). However, in many cases the stricter bound can be applied. The proofs are presented in the next Section, namely in Section 3.5.

The possibility of choosing the initial shape of the regions Ω_i and Γ_j is essential to minimize the required number of iterations. It shall be noted that according to the Shi LSM, all calculations are done in the active front that have direct connection with the initial shape of the aforementioned regions. Making both Ω_i and Γ_q smaller, the smaller the worst case bounds become. This statement leads us to Section 3.8, namely, how to construct initial conditions that are minimal or optimal in the sense of worst case bounds. In the same time, evolutions started from the proposed initial conditions are more effective on a many-core architecture than the ones started from conventional initial conditions [68]. It should be noted that the presentation above does not depend on the dimensionality of the data so the Theorems are general from this point of view and the dimension of the region can be arbitrary.

DOI:10.15774/PPKE.ITK.2014.005

3.5 Proofs of the Theorems

Proof of the general bound. Let $\Omega^a = \Omega^* \cap \Omega = \bigcup_{p=1}^{P} \Omega_p^a$. These are fixed sets and will not change during the evolution process. Furthermore, $F(\mathbf{x_k}) > 0$, $\forall \mathbf{x_k} \in \Omega^a$ which ensures that $\Omega^a \subseteq \Omega$ as Ω evolves.

At initialization for each Ω_i two cases are possible. First case: $\Omega_i \cap \Omega^* = \emptyset$. Then $\Omega_i \subseteq \Gamma^*$ so, $F(\mathbf{x}) < 0$. On the boundary of Ω_i, $L_{in,i}$, a *switch_out* operation is applied so the diameter of Ω_i becomes smaller with two in every iteration. Second case: $\Omega_i \cap \Omega^* \neq \emptyset$. Then the longest possible path in Ω_i gives the upper bound of the number of iterations that is obviously upper bounded by the number of points in Ω_i. Following similar arguments, also this can be shown for Γ_j. Taking the maximum of the upper bounds completes the proof of Theorem 1. □

One can argue that this proof gives a stricter bound than it is stated in the corresponding theorem. Even so there is a constant multiplier $C \leq 1$ between the number of pixels in a connected region and the longest possible path. $C = 1$ if the object is a one pixel wide long line, it is asymptotically 0.5 if it is a curved one pixel wide path with one pixel wide separation. So setting the bound to exactly the number of points is reasonable and valid.

Proof of the convex bound. Obviously, the first case of the proof of Theorem 1 obeys the desired bound. The second case is as follows. Since Ω^* is convex the length of the longest path is bounded by the diameter of Ω_i. In worst case $\Omega_i \cap \Omega^*$ is one of the endpoints of the diameter. Following similar arguments, this can also be shown for Γ_j. Taking the maximum of the diameters in each initial and background region completes the proof of Theorem 2. □

3.6 Many-core hardware platforms

In this work two different hardware platforms are used. As it was depicted in the beginning of the first Chapter, the many core architectures have become a must due to physical constraints like power dissipation and wiring delay. In this context the local connections become more and more attractive. This fact appears in both platforms. The very nature of CNN is based on local connections while

DOI:10.15774/PPKE.ITK.2014.005

connection inside the GPU is realized mainly as memory access that have three different types depending on the accessibility level and the access delay of each type nicely illustrates the heavy cost of global communication.

Firstly, the CNN-Universal Machine (CNN-UM) is covered together with the specific hardware implementation that was used during the experiments. Then the necessary notions and details of GPU hardware are summarized. This is not an extensive description. The unfamiliar reader is directed again to Appendix A and B, where the material regarding GPUs and CNN-UM is covered in a wider extent.

3.6.1 CNN-Universal Machine

The experiments were done on an Eye-RIS v1.3 vision system (VS) (Anafocus Ltd., Seville, Spain). It consists of a Q-Eye, Altera NIOS-II 32-bit RISC microprocessor and on chip RAM. The Q-eye is a QCIF (176×144) monochrome image sensor focal plane processor (vision system on a chip, VSoC) with 7-8 bit de facto accuracy. It is a fine grain CNN-UM implementation with nearest neighborhood capable operations. There is one to one correspondence between each sensor/input, CNN cell and output. Additionally, each cell can reach multiple local analog memory and local logic memory elements. These elements are physically next to the CNN cells. The microprocessor handles the memory, the I/O ports and organizes the execution. It can be programmed in C. The consumption of the complete VS is below 750 mW. The VS was programmed using the Eye-RIS Application Development Kit, a complete Eclipse based development environment.

3.6.2 GPU

Recent GPUs are feasible for non-graphic operations as well and programmable through general purpose application programming interfaces (APIs) like C for CUDA [39] or OpenCL [40]. In this Chapter, OpenCL nomenclature is used. The description below is a brief overview of GPUs. For detailed description of GPU architecture and GPU computing see Appendix A. In addition to the basics, it gives only those details that have great influence on the LS evolution.

DOI:10.15774/PPKE.ITK.2014.005

A function that can be executed on the GPU is called a kernel. Any call to a kernel must specify an NDRange for that call. This defines not only the number of work-items to be launched, but also the arrangement of groups of work-items to work-groups and work-groups to the NDRange. The dimensionality of a work-group can be one, two or three.

Physically, the elementary computing element is the computing element. A few computing elements together with a given amount of SDRAM, scheduling unit(s) and special function unit(s) form a computing unit (CU). A device consists of several CUs and a global memory (off-chip).

The experiments were done on an NVIDIA 780 GTX GPU. It has 12 CUs, 192 computing elements and 48KB shared memory in each CU and 3 GB global memory. The hosting PC runs on Intel core i7-2600 CPU @3.4 GHz with 8 GB system memory, the operating system is Debian with Linux kernel the GPU driver version is 325.15.

3.7 Initial conditions

The final state of the LS evolution depends on two factors. The first one is the applied force field. The second one is the initial state of the LS function also referred to as initial condition. Like it was mentioned in the introduction, the question of the properly constructed force field is not discussed here. However, I give an overview of the commonly used initial conditions. This helps to understand the impact of the theorems. Bearing in mind that there are force types that do not or hardly depend on the initial condition, for example, the region active-contour [69], average misclassificational probability functional [70] or the active contour without edges [48] methods. First, this section describes and illustrates the commonly used initial condition types (also referred to as sparse initial condition) and in the second part it presents the proposed initial condition (also referred to as dense initial condition) family keeping the required number of iterations low, fitting more naturally on many core devices [68].

DOI:10.15774/PPKE.ITK.2014.005

3.7.1 Common initial conditions

The most common initial condition is a single curve. The size, shape or placement requires human specification or depends on the available a priori information that is available for the specific application. However, in most publications it is a single square or circle either covering nearly the whole image or just a tiny spot inside the true object region.

Some examples for common initial conditions are illustrated in Figure 3.3(a)-(h). The interested reader is directed to the literature referenced in this chapter to find more examples for the commonly applied initial conditions. There are several advantages of these kind of initial conditions. First. there is full control on the convergence and the selectivity of the evolution. Second, a broader type of force fields can be applied since with properly chosen initialization the resulted local minima can coincide with the desired or true object in more cases (for example see purely edge based forces [51, 52]).

However, there are some drawbacks as well. First, this kind of initial condition may miss some significant parts of the true object region provided it may not contain or intersect with it in every cases. To avoid this problem either a priori information shall be incorporated or human interaction is required to provide a sensible initial curve. There are no bounds on either the required number of iterations or other measures describing the required number of artificial time steps or like. Furthermore the calculations are slow if the initial condition is far (in Hamming, Hausdorff or Wave metric) from the true object. Another characteristic which is neither advantage nor disadvantage that this kind of initial condition fits well to a single CPU core.

3.7.2 Proposed initial condition family

Theorems 1 and 2 gives bounds on the required number of iterations. The value of these bounds depends only on the initial condition. Theoretically the smaller the connected regions in the initial condition the smaller the bounds are. This implies initial conditions with as small connected regions as possible supposing it converges to the desired output. This requires usage of proper force functions being able to handle the initial condition family.

DOI:10.15774/PPKE.ITK.2014.005

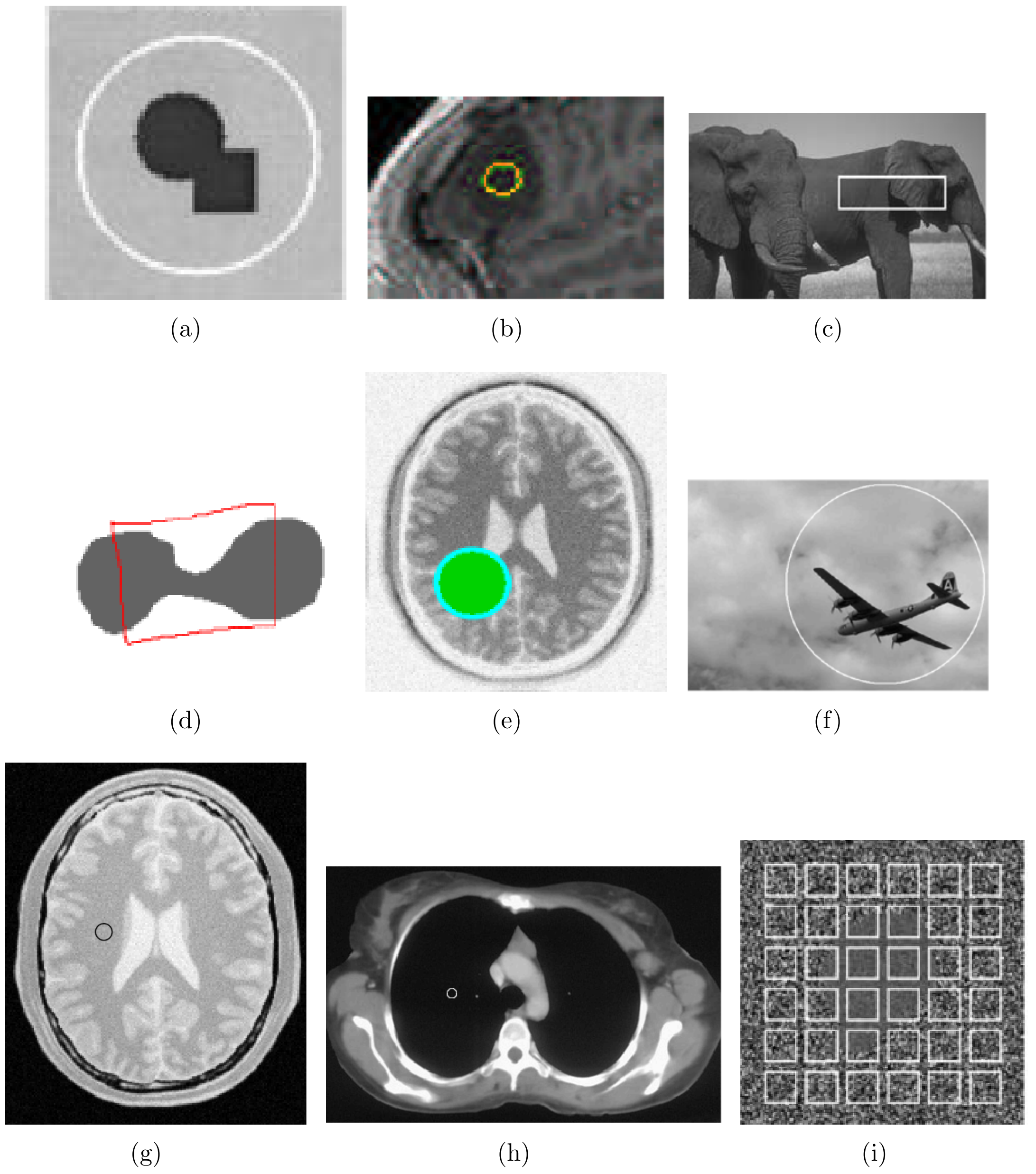


Figure 3.3: Illustrates the commonly applied initial consditions. (a) courtesy of T. Chan and L. Vese. (b) courtesy of A. Lefohn, J. Cates and R. Whitaker, (c) courtesy of N. Joshi and M. Brady, (d) courtesy of G. Sundaramoorthi, A. Yezzi, A. C. Mennucci and G. Sapiro, (e) courtesy of M. Roberts, J. Packer, M. C. Sousa and J. R. Mitchell, (f) courtesy of Y. Shi and W. C. Karl, (g) courtesy of Y. Shi, (h) courtesy of Y. Shi, (i) courtesy of H. Wu, V. Appia and A. Yezzi.

DOI:10.15774/PPKE.ITK.2014.005

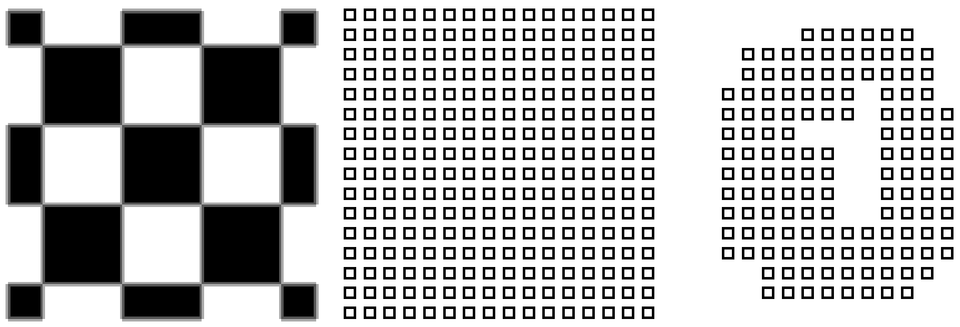

(a) (b) (c)

Figure 3.4: Proposed initial condition family. The whole area of the image is covered with small active fronts. It keeps the required number of iterations under the desired number. The size and shape of the tiny curves can tuned as required and of course a priori information can be incorporated as well. (a) shows an 8×8 pixel block of an initial condition minimizing the bounds. (b) shows an initial condition which theoretically does not minimize the bound due to the large connected region outside the curves. However, practically there is extremely little chance that the true object region lies completely outside the curves. (c) shows an initial condition incorporating a priori information as a form of a spatial mask.

A few illustrations of this initial condition family can be seen in Figure 3.4. The advantages of this family are as follows. Many core implementations are significantly faster if the evolution is started from this kind of initial condition (up to 18× on GPU on 4Mpixel images, see Table 3.1 for measurement data) [68]. However, it must be noted that this kind of initial condition is not completely unknown (see Figure 3.3(i)), it is not widely used according to the literature. Furthermore, there has not been carried out any analysis in this field to the best of my knowledge.

3.8 Experiments

Theorems 1 and 2 give upper bounds on the required number of iterations (N_{it}). A practical proposal of this Chapter is to construct configurations that have as low worst case bounds on N_{it} as feasible and can be computed efficiently on many-core architectures. This scenario is presented and verified through two case studies. The first one is on an Eye-RIS v1.3 VS that contains a hardware

DOI:10.15774/PPKE.ITK.2014.005

implementation of the CNN-UM and the second one is on a GPU [68].

The whole image is covered with many-many small active fronts, and as a consequence, the intersection condition of Theorems 1 and 2($\Omega_p^* \cap \Omega \neq \emptyset$) is automatically fulfilled. Some interesting aspects of this statement will be presented in the discussion.

3.8.1 A case study on CNN-UM

In Appendix B a short overview is given on the CNN-UM. Now the details of the mapped algorithm are described. The perspective in this scenario is the precedence of locality which becomes increasingly important as the technology feature size decreases and delay together with power consumption of global communication increases. As a consequence, the local communication (cellular nature) will become the only viable option.

The mapped algorithm is based on the set theoretic description of the LS function. In addition to L_{in} and L_{out} two other sets are defined representing the inner points of Ω and outer points of Γ

$$F_{in} = \{\mathbf{x} \in D |\, \phi(\mathbf{x}) < 0 \wedge \mathbf{x} \notin L_{in}\} \tag{3.10}$$

$$F_{out} = \{\mathbf{x} \in D |\, \phi(\mathbf{x}) > 0 \wedge \mathbf{x} \notin L_{out}\} \tag{3.11}$$

In other words, the respective value of ϕ of the neighbors of each point in these sets have the same sign as the value of ϕ at the point itself.

Figure 3.5 shows the UMF diagram of the algorithm together with the load and store operations. Templates AND, OR denote elementary logic, ANDNOT performs logic subtraction ($Op1 \wedge \neg Op2$), DIL4 and ERODE4 are the 4 connected dilatation and erosion (spatial logic). All templates are of the nearest neighbor kind and are described in details in Appendix B. In the 'Update Lout' phase, $foutmask$ is computed first. It contains the points that are going to move outward. $foutmask$ is used in three different ways. It is subtracted (ANDNOT) from L_{out}, added (OR) to L_{in} and dilated (DIL4, ANDNOT, AND) to generate its own outer neighbors. This is the new stepped L_{out} part and the unchanged parts are added with an OR operation. The resulting set is finalized as the new

DOI:10.15774/PPKE.ITK.2014.005

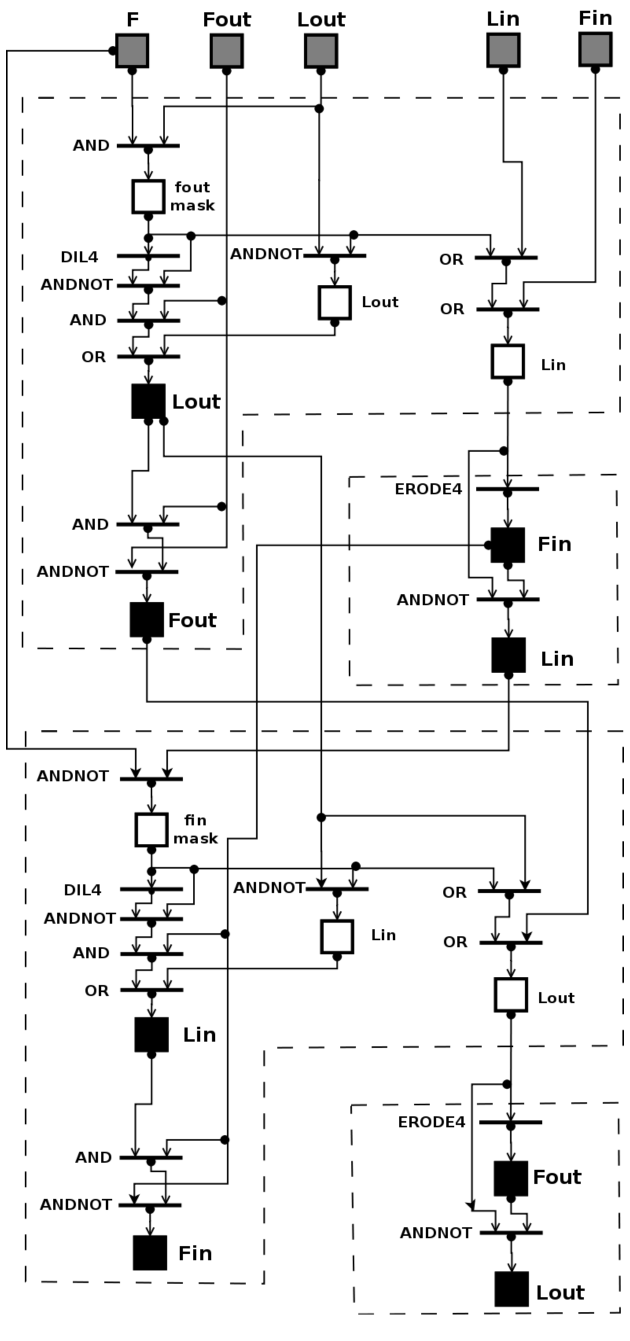


Figure 3.5: UMF diagram of LS evolution. Rectangles denote memories, bold short horizontal lines with capital operator names on the left denote template operations. Dashed lines indicate the phases corresponding to the four cycles of the Shi LSM. Black rectangles denote final forms of sets in that phase. Thin lines ending with arrows denote data-flow from memory to an operation, from an operation to an operation or from an operation to a memory.

DOI:10.15774/PPKE.ITK.2014.005

L_{out} (black rectangle in Figure 3.5). From the old F_{out} the new L_{out} is subtracted (ANDNOT) to get the new F_{out} (again black rectangle in Update L_{out} phase). Finally the modified L_{in} is added to F_{in}. In the 'Clean L_{in}' phase the merged $foutmask$, L_{in}, and F_{in} is the only input. The new L_{in} is the outer pixel layer of this merged input. The new F_{in} is obtained by a simple four connected erosion while L_{in} is the result of a subtraction. 'Update L_{in}' and 'Clean L_{out}' are nearly identical, only the input of the operations are switched, and another mask is used ($finmask$).

In this case study the force field F is assumed to be known and quantized to -1, 0 and 1. Simple templates are used without feed-back dynamic (non central A template elements are zero). This ensures the template operations to be robust and fast. The different types of discrete neighborhoods can be implemented through the type of the dilation and erosion. In this specific case 4 connectedness is used. All template operations reach their stable solution within 2τ.

The algorithm is implemented on the Eye-RIS 1.3 VS. One step of the algorithm is performed in $400 - 440\mu s$ on a QCIF image. It must be noted that the actual computing is finished within $60-70\mu s$ and the remaining time ($340-370\mu s$) is required for the data movement from the main memory of the Eye-RIS (on the Altea NIOS-II microprocessor) to the Q-Eye chip memory.

3.8.2 A case study on GPU

The iteration process is divided into two steps. The first one is the planner step and the second is the evolution step. The elementary block of the image that is minimally processed is a tile. Its size in our case is 16×16 pixels both on the input and ϕ image. The planner creates the so-called plan. It contains the position offsets of the tiles that are calculated actually in the iteration step. The pseudo-codes of both kernels are presented.

Functions have '(...)' after their names. The i^{th} element of an array is accessed by the '[i]' operator. Identifiers starting with capital 'L' denote variables that are shared among the threads of the same work-group and are placed in the local memory. The only exception is the 'LID' variable. The function call 'barrier(Local)' serves as a synchronization point within a work-group namely,

DOI:10.15774/PPKE.ITK.2014.005

all threads of the work-group shall execute this command before any of them can issue a new instruction. This is required to ensure data consistency of the local variables used for local data share. The hardware can execute global atomic operations. These operations are thread safe but the order of the serialization shall assumed to be random. Furthermore, it can return the state of the written variable before the actual, de facto operation (for example addition) takes place.

The pseudo-code of the planner kernel is provided in Algorithm 3.5. The planner works on the indicator image. The indicator is a tiny image and each pixel of the indicator is *true* if the corresponding tile on the input image shall be processed in this iteration and *false* otherwise. This kernel is run in a 2D fashion namely, that each thread corresponds to a single pixel on the indicator image and to a whole tile on ϕ and the input image. A pixel is changed from false to true if any neighboring tile have active front on its connecting side. This functionality is represented by the 'checkNeighborActivity(...)' function. The size of the plan is calculated by local prefix-sum work-group wise, and global atomic addition is used to correctly determine the offset of the work-group within the plan (line 16 in the pseudo-code of the planner kernel).

A prefix-sum operation requires n numbers and n threads/processors. There is a complete ordering defined on the threads. The output of each thread is a number that is the sum of all numbers corresponding to threads not greater than the given thread. This is an optimal way to determine the writing place of each thread within an array provided each thread writes different amount of data. The time complexity of the operation is $O(\log_2 n)$

The pseudo-code of the evolution kernel is provided in two parts: Algorithm 3.6 and 3.7. The evolution kernel processes only those tiles of the LS function that are inserted in the plan. The evolution kernel makes a step either inward or outward direction depending on the sign of the force field on the LS function. This is done simultaneously unlike in the sequential algorithm. Each work-group processes a 16×16 tile provided in the plan and writes the complete tile back to the global memory. First, each work-item calculates force field of the corresponding pixel ('calcForce(...)') then the new value of the pixel of the LS function ('calcNewPhi(...)'). The force can be an arbitrary operator, during the experiments it was a pure region based term. It is beneficial if the force term can

DOI:10.15774/PPKE.ITK.2014.005

Algorithm 3.5 planner kernel

```
1: function PLANNER(Indicator, $\phi$, plan, planSize)
2:     LID ← getLocalID()                                   ▷ ID within the work-group
3:     GID ← getGlobalID()                  ▷ ID within all threads of all work-groups
4:     LSize ← getLocalSize()                                  ▷ size of the work-group
5:     pixel ← readImage(Indicator,GID)
6:     isActive ← pixel = true
7:     for ∀ NGID ∈ neighboring GIDs do
8:         isActive ← checkNeighborActivity(NGID,$\phi$,isActive)           ▷ see text
9:     end for
10:    writeImage(Indicator,GID,isActive)
11:    LPositions[LID] ← isActive               ▷ local array of the writing position
12:    barrier(Local)                           ▷ work-group level synchronization
13:    doLocalPrefixSum(LPositions)
14:    barrier(Local)
15:    if LID = 0 then
16:        LOffset ← atomicAdd(planSize,LPositions[LSize])                 ▷ see text
17:    end if
18:    barrier(Local)
19:    LPositions[LID] ← LPositions[LID]+LOffset
20:    barrier(Local)
21:    if isActive then
22:        plan[LPositions[LID]] ← GID
23:    end if
24: end function
```

DOI:10.15774/PPKE.ITK.2014.005

Table 3.1: Time measurements on NVIDIA GTX 780 GPU compared to Intel core i7 CPU

Data size	**Initial condition**	$\bar{T}_{\text{iteration}}$ **on GPU** (μs)	$\bar{T}_{\text{iteration}}$ **on CPU** (μs)	N_{it}	**Speedup**
256×256	1×1	129	1,610	32	12.5
256×256	2×2	126	2,242	59	17
256×256	8×8	140	3,164	20	22
256×256	32×32	143	8,874	8	62
512×512	1×1	317	3,190	64	10
512×512	4×4	167	8,724	40	52
512×512	16×16	157	12,897	25	82
512×512	64×64	123	16,246	18	132
$1,024 \times 1,024$	1×1	534	6,431	129	12
$1,024 \times 1,024$	8×8	548	27,461	55	50
$1,024 \times 1,024$	32×32	590	43,739	32	74
$1,024 \times 1,024$	128×128	490	84,078	12	171
$2,048 \times 2,048$	1×1	560	14,972	210	26
$2,048 \times 2,048$	16×16	703	79,920	79	113
$2,048 \times 2,048$	64×64	830	198,980	28	239
$2,048 \times 2,048$	256×256	684	327,541	7	478

Presented results are the mean value of 100 runs.

DOI:10.15774/PPKE.ITK.2014.005

be composed only from small radius local operations.

The neighbors of each pixel are updated by a combined Switch operation (pseudo-code available in Algorithm 3.8) as the *switch_out()* and *switch_in()* operations require according to the neighborhood pattern (the pseudo-code shows 4 connectivity). It is followed by the cleaning of the active front (see lines 28-33 in Algorithm 3.6) to maintain the two pixel width. The boundary of the tile requires special care, namely, to properly update the corresponding neighboring pixels of the LS function and the indicator image. The kernel checks whether there was any activity inside the tile. It is done by parallel reduction. It is an optimal operation to sum up n numbers on n processors in $O(\log_2 n)$ time. Finally, the corresponding pixel of the indicator image is set to *false* if there is no activity within the tile.

Table 3.1 shows execution time measurements of the work-efficient parallel algorithm on NVIDIA 780 GTX GPU compared to a baseline single-threaded implementation on Intel core i7-2600 CPU. The execution time was measured by the `gettimeofday()` C-function which has microsecond resolution. The table specifies the image resolution, the initial condition configuration, and presents the mean of the execution time of an iteration on GPU, on CPU and the speedup. The iteration time on the GPU contains the execution time of both kernel functions (planner, iteration). The two kernels evenly share the execution time in the case of conventional, sparse initial condition; however, in the case of dense iteration steps, the ratio of the evolution kernel can shift to 30:1 with respect to the planner.

3.8.3 Number of iterations

In the experiments more initial configurations were tested. In each configuration, regions of Ω and Γ were placed in a chessboard like pattern as it is showed in Figure 3.6(a) and 3.6(b). Two sample objects are presented in Figure 3.6(c) and 3.6(d) that shall be detected. Additionally, the two objects represent the two object families: the degenerate and convex ones having worst case bounds stated in the Theorem 1 and 2.

Iteration examples are presented in Table 3.2 together with the two different bounds of the given configuration. The number of iterations (N_{it}) was measured

DOI:10.15774/PPKE.ITK.2014.005

Algorithm 3.6 evolution kernel Part 1

```
1: function EVOLVE(Image, Ind, $\phi$, plan, planSize, ... )
2:     LID ← getLocalID()
3:     LSize ← getLocalSize()
4:     GrID ← getGroupID()
5:     if LID.x = 0 and LID.y = 0 then
6:         GPosition ← plan[GrID]                          ▷ offset of the tile
7:     end if
8:     barrier(Local)
9:     GID ← {LID.x + GPosition.x*GrID.x, LID.y + GPosition.y*GrID.y,}
10:    pixel ← readImage($\phi$,GID)
11:    F ← calcForce(pixel,GID,$\phi$, I, ...)
12:    pixel ← calcNewPhi(pixel,GID,$\phi$)          ▷ see rules in switch_{in,out}
13:    LPixels                          ▷ a local array for the tile with borders
14:    LPixels[LID] ← pixel
15:    barrier(Local)
16:    if LID.y = 0 then               ▷ fetch neighboring pixels around the tile
17:        calcTileBorders(LPixels, GID,$\phi$, I, NORTH, ... )
18:        calcTileBorders(LPixels, GID,$\phi$, I, SOUTH, ... )
19:        calcTileBorders(LPixels, GID,$\phi$, I, WEST, ... )
20:        calcTileBorders(LPixels, GID,$\phi$, I, EAST, ... )
21:    end if
22:    barrier(Local)
23:    switch($n_N$, LPixels, NORTH)
24:    switch($n_E$, LPixels, EAST)
25:    switch($n_S$, LPixels, SOUTH)
26:    switch($n_W$, LPixels, WEAST)
27:    pixel ← LPixels[LID]
28:    if pixel = -1 and $n_N < 0$ and $n_E < 0$ and $n_S < 0$ and $n_W < 0$ then
29:        pixel ← -3
30:    end if
31:    if pixel = 1 and $n_N > 0$ and $n_E > 0$ and $n_S > 0$ and $n_W > 0$ then
32:        pixel ← 3
33:    end if
34:    writeImage(pixel, GID, $\phi$)
35:                            ▷ Here ends the first part of the evolution kernel
```

DOI:10.15774/PPKE.ITK.2014.005

Algorithm 3.7 evolution kernel Part 2

```
        LPixels[LID] ← pixel
        writeBorder(LPixels, $\phi$, Ind, NORTH)
        writeBorder(LPixels, $\phi$, Ind, EAST)
        writeBorder(LPixels, $\phi$, Ind, SOUTH)
        writeBorder(LPixels, $\phi$, Ind, WEST)
        isActive                                    ▷ local array of size LSize
        isActive[LID] ← pixel = -1 **or** pixel = 1
        barrier(Local)
        doParalelReduction(isActive)
        barrier(Local)
        **if** LID.x = 0 **and** LID.y = 0 **and** isActive[0] = 0 **then**
            writeImage(false, GPosition, Ind)
        **end if**
    **end function**
```

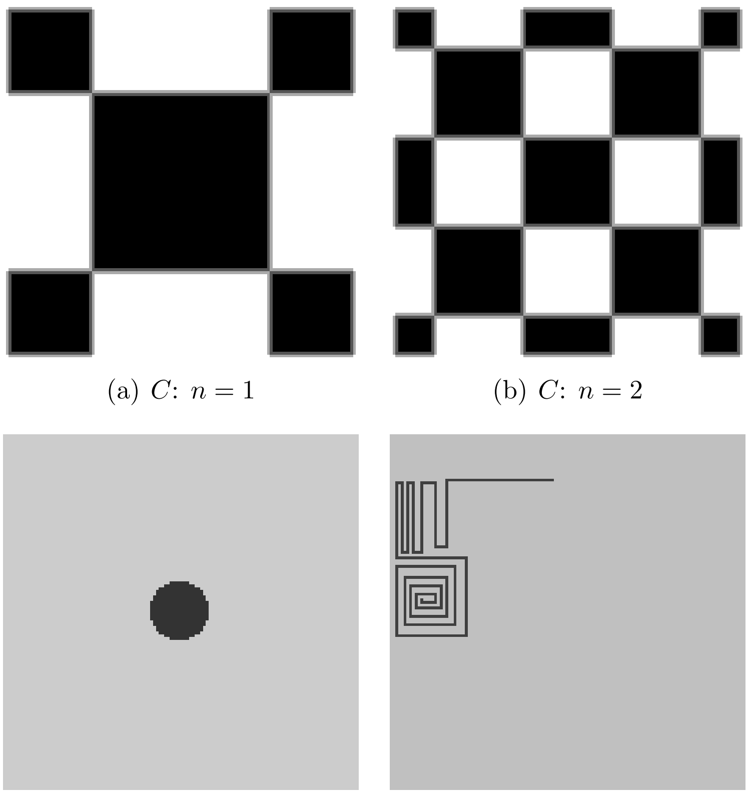
Algorithm 3.8 Switch operation for GPU evolution

```
**function** SWITCH($n_{DIR}$, LPixels, DIR)
    idx ← remap(LID,DIR)                    ▷ connects logical and physical layout
    $n_{DIR}$ ← LPixel[idx]
    LPixel[idx] ← (pixel = -1 and $n_{DIR}$ = 3)?1: $n_{DIR}$
    barrier(Local)
    $n_{DIR}$ ← LPixel[idx]
    LPixel[idx] ← (pixel = 1 and $n_{DIR}$ = -3)?-1: $n_{DIR}$
    barrier(Local)
**end function**
```

DOI:10.15774/PPKE.ITK.2014.005

Table 3.2: Examples of the Theorems. The image is 128^2 pixels. Configuration C was set as squares arranged into n rows and n columns in a chessboard like pattern (see Figure 3.6(a)-(b)). Two different objects were tested: a circle in the center with radius 11 pixels and a snake-like degenerate object. Configuration and objects are presented in Figure 3.6

	number of squares in n rows and n columns							
	1	2^2	4^2	8^2	16^2	24^2	32^2	64^2
bound according to Theorem 1	64^2	32^2	256	64	16	9	4	1
bound according to Theorem 2	127	63	31	15	7	5	3	1
N_{it} for Figure 3.6(c)	26	16	9	6	4	3	3	1
N_{it} for Figure 3.6(d)	145	68	18	7	6	3	3	1


(a) C: $n = 1$ (b) C: $n = 2$

(c) convex object (d) degenerate object

Figure 3.6: This figure presents initial conditions and two test objects representing the two extremities.

DOI:10.15774/PPKE.ITK.2014.005

on the original sequential algorithm of Shi and these values are presented in the Table. It is below or equal to the worst case bounds in every cases.

In the case of CNN-UM, N_{it} coincides with the values presented in the Table, while in the case of GPU implementation, N_{it} is consistently higher with one iteration. This means that it exceeded the bounds in the case of $n = 32$ and $n = 64$. However, the reason is as follows: the boundary pixels of the subregion have one iteration delay in the cleaning process. This causes the additional iteration so it is not a violation of the Theorems.

3.8.4 Segmentation example

In this subsection I present the applicability of the described initial condition on a real task. The selected problem is white matter segmentation from a T1 weighted 3D image. The image originates from the 3T MR scanner of the Semmelweis University (SU) I. Neurological department. The image is taken from a healthy male human who participated in a cognitive experiment done by the Faculty of Information Technology and Bionics and the SU.

The 3D image is processed slice by slice in a sequential manner. In this way the information extracted from the previous slice is available for the actual slice. The force field is a region based one with curvature based regularization. The intensity range of the white matter coincides with intensity range of the bone in T1 weighted images. To eliminate the skull bone from the images a simple wave operation is used. The first object regions appearing on the slices processed sequentially is the skull bone. This is used for the next slice to eliminate the bone parts in a wave like manner. This easy method eliminated completely the boney parts on 90% of the slices and on the remainder a small part containing only few pixels (8×2) remained. The regularization term is the curvature that is handled by one linear heat diffusion operator on the slice. This is equivalent with the Gaussian filtering of the curve or regularizing the curve with directly the curvature through the force field.

This algorithm was implemented in a CNN-UM simulation environment called MatCNN and SimCNN implemented in Matlab and Simulink. In Figure 3.7 the result of the segmentation can be seen. The slices are selected from the region

DOI:10.15774/PPKE.ITK.2014.005

39th and 75th slices. It shall be noted that these results are just demonstrating the applicability of the proposed initial condition.

3.9 Validation

In this section, we compare the result of the exact numerical implementation and the Shi LSM for three different force fields: mean curvature motion, Chan-Vese and geodesic active region (GAR). The quantitative comparison is made by the dice coefficient. Given the state of the two LS functions Ω_1 and Ω_2 of the two different methods, the coefficient is defined as

$$d(\Omega_1, \Omega_2) = \frac{2\mathrm{Area}(\Omega_1 \cap \Omega_2)}{\mathrm{Area}(\Omega_1) + \mathrm{Area}(\Omega_2)} \tag{3.12}$$

Its value is in the range of 0 and 1; 0 means complete difference and 1 means complete agreement. The size of the images is 200×200 pixels in all three cases.

3.9.1 Mean curvature flow

In this case, the force field is defined as

$$F = -\kappa \tag{3.13}$$

where κ is the (Euclidean) curvature of the LS. It is the norm of the second derivative of γ with respect to the (Euclidean) arc length ($\kappa = \|\gamma_{ss}(s)\|$, s is the arc length parametrization of the curve). Another possible, precise and easier way to calculate the curvature of an LS from ϕ is as follows:

$$\kappa = \operatorname{div}\operatorname{grad}\frac{\nabla\phi}{\|\nabla\phi\|} \tag{3.14}$$

This force term appears in almost every LS flow as a smoothing and regularizing term. The steady-state solution is a circle with infinitesimal diameter. In practice, the object region vanishes as the artificial time increases. In this case, not only the steady state but the evolution itself is also investigated. This is an autonomous motion and does not have any control term from an external image.

DOI:10.15774/PPKE.ITK.2014.005

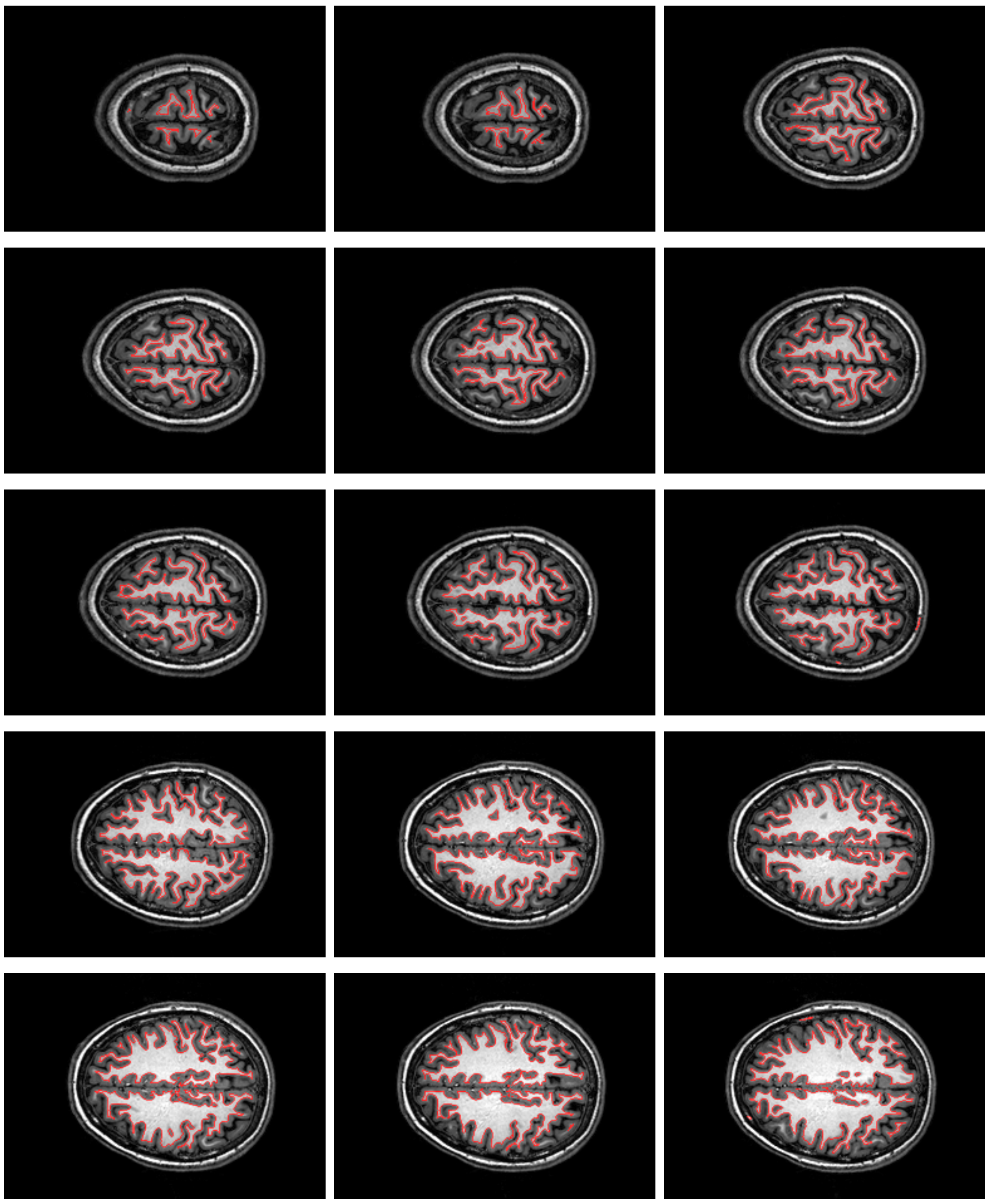


Figure 3.7: White matter segmentation on T1 weighted MR image. Results are only demonstrating the applicability of the proposed initial condition.

DOI:10.15774/PPKE.ITK.2014.005

The details of the numerical approximation are as follows. The LS function ϕ is a signed distance function. It was recalculated after every 30 iterations. The artificial time (T_{maximum}) runs to 800 units. The time step (Δt) size has been set to 0.4. The curvature has been calculated from the LS function from Equation 3.14.

In the case of the fast LS evolution, the curvature was calculated according to the work Merriman, Bence and Osher (MBO) [71, 72], namely, by $G \otimes \phi$, where G is a 2D Gaussian of a given variance.

Figure 3.8 shows the test initial condition for mean curvature motion and the state of the evolution after 20, 40, 60 and 80 iterations of the fast LS evolution.

Figure 3.9 shows the dice coefficient between the first 80 steps of the fast LS evolution and the corresponding state of the numerical approximation.

3.9.2 Chan-Vese flow

This method was proposed in [48] and its speed term is defined as

$$F = \mu\kappa - \lambda_1(c_1 - I)^2 + \lambda_2(c_2 - I)^2 \tag{3.15}$$

The parameters are set as follows: $\mu = 1, \lambda_1 = 0.8, \lambda_2 = 0.8$. I represents the input image intensities, the constants $c_1 = 0.5$ and $c_2 = 0$ are simply the means of pixel intensities inside and outside the zero LS. The artificial time parameter runs to 180 units, the time step is 0.5 units. The total number of iterations is 360. The initial condition is 25 circles arranged uniformly in five rows and five columns each with diameter 27 pixels. The LS function (signed distance) is recalculated in every 30 iterations for the numerical solution. The steady states of the two Cahn-Vese evolutions are shown in Figure 3.10(a). The dice index of the two states is 0.998.

3.9.3 Geodesic active regions flow

This method was proposed in [69]. This method combines boundary and region-based information to segment an image. In this method, the pixel intensities are

DOI:10.15774/PPKE.ITK.2014.005

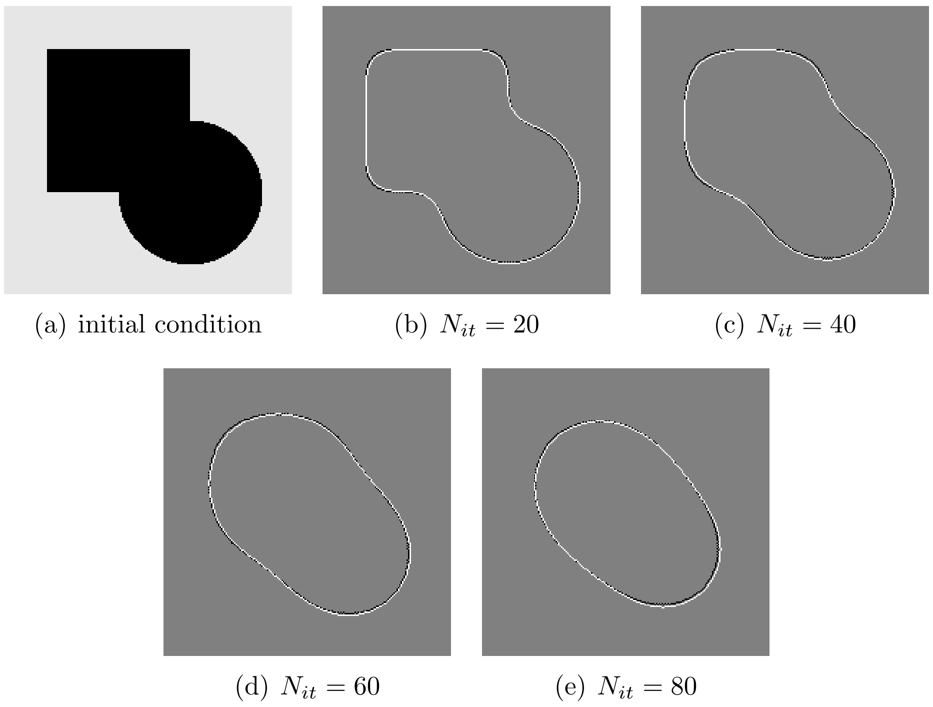


Figure 3.8: Comparison of mean curvature evolution of PDE approximation and fast LS evolution. This shows the initial condition and evolution of fast LS (white line) and numerical PDE approximation (black line). (a) Test initial condition for validation of mean curvature motion: fast LS evolution against numerical PDE approximation. The test region contains positive, negative and zero curvature regions and singularities as well. (b) State of evolution fast LF at $N_{\mathrm{it}} = 20$ and PDE approximation at $T = 56.8$. (c) State of evolution fast LF at $N_{\mathrm{it}} = 40$ and PDE approximation at $T = 190.8$. (d) State of evolution fast LF at $N_{\mathrm{it}} = 60$ and PDE approximation at $T = 405.6$. (e) State of evolution fast LF at $N_{\mathrm{it}} = 80$ and PDE approximation at $T = 706.8$.

DOI:10.15774/PPKE.ITK.2014.005

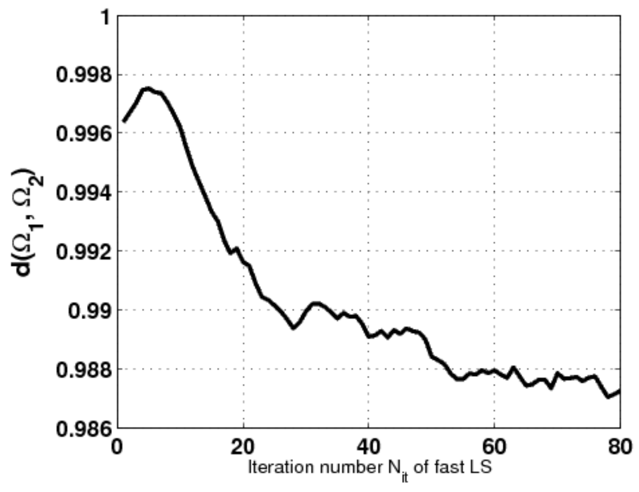
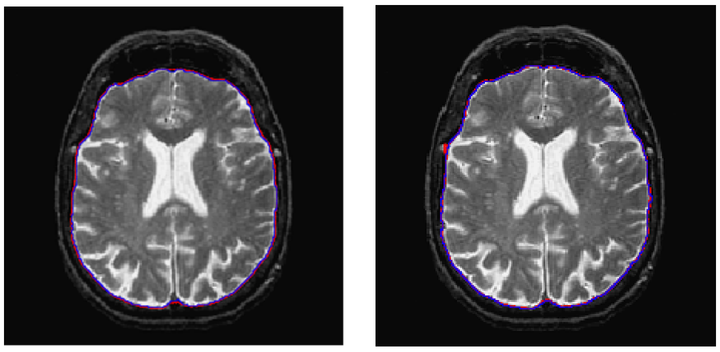


Figure 3.9: Dice index of mean curvature evolution. Ω_1 is the state of the fast LS evolution, and Ω_2 is the state of the numerical solution. The similarity between the two states is very high.



(a) Chan-Vese (b) GAR

Figure 3.10: Validation of fast LS evolution. (a) CV (b) and **(B)** GAR flow. Red corresponds to the numerical PDE solution while blue corresponds to the fast LSM. The two curves are nearly the same and the dice index is 0.998 in both cases.

DOI:10.15774/PPKE.ITK.2014.005

modeled with a Gaussian mixture model (GMM). The force field is as follows:

$$F = -\alpha \log\left(\frac{P(I|R_1)}{P(I|R_2)}\right) + (1-\alpha)\left(b\kappa + \nabla b \frac{\nabla \phi}{|\nabla \phi|}\right) \tag{3.16}$$

where R_1 and R_2 are the regions to be separated, b is a strictly decreasing function of boundary probability, and α is a balancing constant. In our case $\alpha = 0.3$, and b is defined as follows:

$$b = \frac{1}{1 + \|\nabla G \otimes I\|} \tag{3.17}$$

Here G is a 2D Gaussian with $\sigma = 3$. The GMM parameters are calculated from the image histogram with a recursive expectation maximization algorithm. The artificial time runs to 6 units, the time step is 0.02 units. The total number of iterations is 300. The LS function (signed distance) is recalculated in each 30 iterations for the numerical solution. The initial condition is the same as in the case of Chan-Vese evolution, 5×5 circles each with the diameter of 27 pixels. Steady states are shown in Figure 3.10(b). The dice index of the two states is 0.998.

3.10 Discussion

In this chapter, given our investigation of the initial condition and the required number of iterations as a function of it, we presented two bounds on the required number of iterations of LS evolution of Shi. The bounds were proven theoretically and checked experimentally with the original algorithm and also with two different mappings of the algorithm on many-core machines (GPU, CNN-UM). The bounds depend only on the initial configuration of the LS function. The many-core realizations required not only a very small number of iterations less than or equal to the bounds, but the execution of an iteration was also fast (see Table 3.1 for detailed measurement data).

In addition to the drastic decrease of the required number of iterations, the total execution time decreases as well if dense initial condition is used for the evolution. The total execution time on CPU with sparse initial condition is comparable to the total execution time with dense initial condition. For the

DOI:10.15774/PPKE.ITK.2014.005

smaller images, the dense initial condition was less effective by 30% to 15%; but in the case of the biggest image, the dense iteration was the faster by 35%. In the case of the dense initial condition on GPU, there is a significant speedup compared to the sparse initial condition in all cases since our proposed dense initial condition together with the algorithm utilizes the properties of the underlying architecture. Therefore, greater performance gain can be achieved on GPU if dense initial condition is used.

A great property of the results is their scalability. This is true for the performance as a function of cores and for the number of iterations as a function of size of the disjoint active fronts. Considering the chessboard-like initial configuration with increasingly finer regions, the general bound is proportional to the area of the regions and the convex bound is proportional to the half perimeter of the regions. This is changed in three dimensions to the volume of region in the case of general bound and half of the longest perimeter of the volume in the case of a convex bound.

The assumption on F is stronger in Theorem 1 than the one that was given in the convergence analysis in [45]. In the examples presented there, our stronger assumption stands for at least one of the regions Ω^*, Γ^*. However, there may be cases when for a short period of iterations the sign of F changes. Typically, this is the case when inside the true object region, the actual state of the LS function contains a concave background region with high negative curvature. In these cases, the curvature-based term can be greater than the region term (the pixel-intensity-based terms), but this is a temporary effect. As soon as the local concavity is vanished, the region term becomes again greater and the sign of F changes back. Furthermore, as it was declared in the introduction, the construction of the force field and its components is out of the scope of this dissertation. Additionally, the validations indicate that the method converges *de facto* to the same state as the exact numerical solutions.

The fact that the active front of the initial condition covers the whole image has a special consequence, namely, separate, disjoint regions of the same object or multiple target objects can be found automatically without user interaction. For example, the gray matter of the brain on an MRI slice can be disconnected and may be composed of 8 to 20 disjointed regions on the given slice. The

DOI:10.15774/PPKE.ITK.2014.005

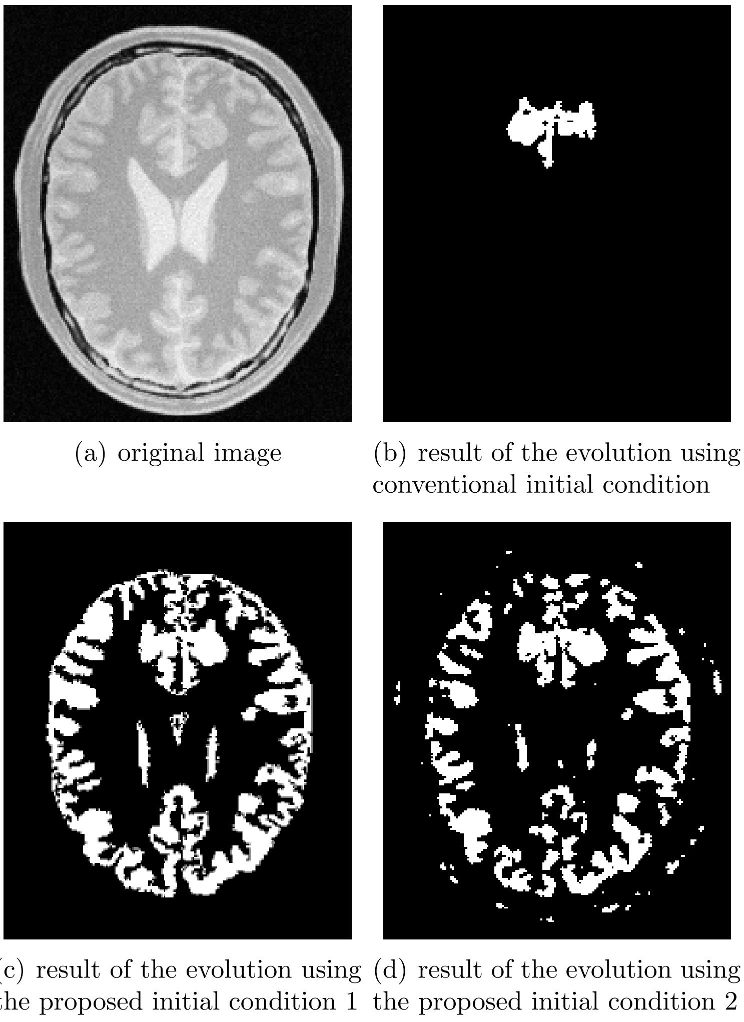
problem of detecting all regions is greatly simplified with the proposed dense initial condition. Similarly, a selected group of cells in a histology image can show this property as well. Additionally, histology images can be extremely large (2 to 30 Mpixel), and the performance gain of our proposed method (initial condition together with the parallel algorithm) becomes more expressed on larger images. A conventional sparse initialization can easily fail this task, with wrongly chosen initial condition, see for instance the initialization and evolution of a gold standard LS implementation of [73], which is a widely used framework for medical image segmentation and analysis.

Figure 3.11 shows an example. The evolution from a single-circle initial condition is presented on Figure 3.11(b), while our result is presented on Figure 3.11(c)-(d). It demonstrates its potential and it may be an initial condition for fine-tuning the segmentation with another method. Of course, the dense iteration may have the drawback of increased false-positive rate, for example see Figure 3.11(d) where the evolution runs with slightly different parameters, but this could be handled with more sophisticated force fields or building *a priori* information into the initial condition.

I have evaluated the precision of the Shi method by three different force fields. The results were compared to the solutions of the numerically approximated PDE evolutions. Since the time steps satisfied the Courant-Friedrich-Levis condition ($\Delta t \cdot F < \Delta x$) these numerical approximations can be viewed as ones extremely near to the exact (analytical) solutions. I have not evaluated other fast LS methods since the Shi method is one of the fastest ones with very small memory footprint that can be transformed into an effective memory access layout on GPU. There are some limitations due to the lack of enough logic memory on the Q-Eye breaking down the performance even so it is a lightweight, fast and low power realization. On CNN-UM there may be further directions to incorporate different wave operators and shift from the fully feed forward approach to include feed back terms as well.

It must be emphasized that the case studies presented here are not necessarily optimal mappings of the Shi LS evolution by any means. The purpose of presenting them is twofold: (1) to highlight the advantage of the proposed initial condition concept especially on those machines and (2) to give a proof of concept

DOI:10.15774/PPKE.ITK.2014.005


(a) original image (b) result of the evolution using conventional initial condition

(c) result of the evolution using the proposed initial condition 1 (d) result of the evolution using the proposed initial condition 2

Figure 3.11: Initial condition dependence of evolution. (a) Shows the original image to be segmented (gray matter of the brain). (Figure 3.11(a) is reproduced from [73]). (b) Shows the reached solution of evolution started from a single circle initial condition. (c) Shows the reached solution with our proposed initial condition (32×24 curves with diameter 3 pixels) with force field containing a priori information. (d) Shows the reached solution of evolution with slightly modified parameters compared to the evolution shown in Figure 3.11(c) without the built-in a priori information.

DOI:10.15774/PPKE.ITK.2014.005

mapping of this fast evolution on two totally differently organized (virtual and physical) many-core machines.

3.11 Conclusions

To automatically detect and segment objects in an image or on a region of it, the LS based algorithms are feasible tools. In this Chapter, it was shown theoretically and experimentally through two case studies that the initial condition plays an essential role in decreasing the execution time. It must be emphasized that this is only validated on many-core architectures where the computations can be distributed among the cores.

Based on the initial condition configuration, two worst case bounds were given on the required number of iterations depending on the convexity of the true object or background region. The bounds are proven theoretically and some example experiment were done. Additionally, the execution time of one iteration was measured on two different architectures showing a very fast total execution time till the convergence.

In the case of the proposed dense initial condition, there is a significant speedup compared to the sparse initial condition in all cases since our dense initial condition together with the algorithm utilizes the properties of the underlying architecture. Therefore, greater performance gain can be achieved (up to 18 times speedup compared to the sparse initial condition on GPU).

The results and tools presented in this Chapter provide a method to efficiently calculate LS algorithms mapped on many-core architectures and ensure bounds on the execution time through the two Theorems.

DOI:10.15774/PPKE.ITK.2014.005

Chapter 4

Conclusions

In principle this dissertation covered two main fields, the DRR generation on GPU, and the initial condition dependence of LS, and one minor field, GPU and block size optimization, connected to the DRR generation. Each field has its added value and has impact on medical imaging either directly or indirectly. The most direct contribution is clearly the DRR generation on GPU. It has many time critical applications in various fields from diagnosis through intervention to therapy. The work itself was motivated from the industry as well. The findings of the optimization were examined in a wider extent and it has become a completely new and surprising result in execution time optimization on GPUs. LS based algorithms and methods have applications in several different fields from mathematics, physics, engineering and computer science. Among the many two fields should be mentioned: computer vision and medical imaging analysis. However, it is even possible that other LS fields may benefit from the proposed initial condition family.

DOI:10.15774/PPKE.ITK.2014.005

Summary

4.1 Materials and methods

DRR rendering is realized in CUDA C of Nvidia. I examined several optimization rules and parameters to be able to fit the task on a given hardware. I have made experiments on GPUs based on different architecture generations (8800 GT, 280 GTX, Tesla C2050, 570 GTX, 580 GTX). Furthermore, two different compiler and driver combinations have been used (3.2 compiler + 260.16.21 driver, and 5.5 compiler + 331.67 driver). Two different datasets have been utilized. The first is a CT scan made from a radiological torso phantom (Radiology Support Devices, Newport Beach, CA, model RS-330) with resolution $512 \times 512 \times 72$, the other is a scan taken from a pig head with resolution $512 \times 512 \times 825$ from an annotated database [42]. The phantom imitates the attenuation of human tissue like lungs, bone, arteries, etc. In the X-ray spectrum I have made measurements on complete DRRs and randomly sampled ones as well. The parameters of the rendering have been set to values that are relevant in the case of minimally invasive surgeries (region of interest, ROI and sampling rate).

During my work connected to LS methods I was required to understand the hyperbolic conservation laws as well as the concept of viscosity solutions from the field of partial differential equations (PDE). The viscosity solution is defined as the solution of the following PDE $G(u)u_x + u_t = \epsilon u_{xx}$, subject to ϵ tends to 0. The theory of LS methods and the underlying equations are essential to understand for specific tasks like segmentation and curve motion. Additionally I required the basic notions of discrete topology and convex sets to be able to construct the proofs of my theorems. Execution time measurements were done on Eye-RIS v1.3 vision system (VS) and on Nvidia 780 GTX GPU.

DOI:10.15774/PPKE.ITK.2014.005

4.2 New scientific results

Thesis 1.

I formed a ruleset (1)-(4) allowing the rendering of DRRs to be performed efficiently on Nvidia GPUs. This step is responsible for the slowness of 2D to 3D registration. I applied the rule-set on the calculation of randomly directed line integrals for DRR rendering and systematically searched the block size parameter in the theoretically possible range. According to my findings the value of block size for efficient rendering is in the range of 8-16 threads in a block unlike the theoretical suggestions. So the 2D to 3D registration can be performed in real time for surgical need depending on the application in 0.5-10 frames per second. I showed that DRR rendering can be performed in 0.2-2.2 ms in the case of a region of interest (ROI) containing fully a lumbar vertebra (16×9 cm^2, 400×225 resolution).

1. *Slow 'if else' branches shall be replaced with ternary expressions if possible that are compiled to selection 'parallel thread execution' (PTX) instructions that are faster than any kind of branching PTX instructions.*

2. *Data that is read locally and in an uncoalesced way shall be placed in texture memory provided it is not written.*

3. *Avoid division if possible and use the less precise, faster type (div.approx, dif.full instead of div.rnd).*

4. *If the denominator is used multiple times calculate inverse value and multiply with it.*

I presented measurements on randomly sampled DRRs executed on GPU first [32]. The effectiveness of the first and second optimization rules are presented in Table 2.3. The cumulative effect of the third and the fourth rules as a function of the block size is presented in Figures 2.6-2.13.

The missing branching optimization resulted in a $6-11\%$ performance decrease, 8% in average on Tesla C2050 GPU while $6-13\%$ decrease on 570 GTX GPU, if the optimized version is considered 100%. The linear memory caused a 1.75-2.4 times slowdown consequently on both GPUs.

DOI:10.15774/PPKE.ITK.2014.005

The optimal block size in the case of the optimized kernel is always in the range of 8-16 threads in a thread block. This property was tested in a former version of compiler and driver as well as on four different top GPUs (8800 GT, 280 GTX, Tesla C2050, 580 GTX). The characteristics of the optimized kernel were similar in this software environment too.

Publications connected to this thesis group: [I]. The thesis claim is specified and paraphrased in details in the second chapter of my dissertation.

Thesis group 2.

I present bounds on the required number of iterations of the LS method of Shi [45] and this bound depends only on the initial condition. I propose an initial condition family that decreases the bound in a flexible and effective way. Additionally, evolutions started from this initial condition family require drastically reduced time to converge.

Thesis 2.1 I discovered two new theorems, one for a general case and another for a convex case to determine the worst case required number of iterations of the Shi LS method to converge to the solution. These bounds depend only on the initial condition. I developed proofs for both cases and supported the bounds with experiments. The results are utilized in thesis claim 2.2.

Let us consider a subset of $\mathbb{Z}^n$, say D. A point $\mathbf{x} \in D$ is characterized by its coordinates ($\mathbf{x} = (x_1, ..x_k)$). A *path* p between $\mathbf{x}$ and $\mathbf{y}$ is a sequence of points $\mathbf{x}_l(l = 0, 1, ..., L) \in D$ subject to $\mathbf{x}_l \in N(\mathbf{x}_{l+1})$ and $\mathbf{x} = \mathbf{x}_0$ and $\mathbf{y} = \mathbf{x}_L$. A set of points $\mathcal{A}$ forms a *connected region* if and only if there exists a path p between every $\mathbf{x}, \mathbf{y} \in \mathcal{A}$ subject to $\forall \mathbf{x}_l \in p$ is an element of $\mathcal{A}$. A *minimum path* p_{min} is the shortest path meaning there are no shorter p' paths between $\mathbf{x}$ and $\mathbf{y}$. Minimum path is usually not unique and can depend on the chosen discrete neighborhood. The diameter B of a connected region is the longest minimum path having at least its endpoints within the connected region. A connected region is considered as convex if all minimal paths are minimum paths at the same time.

Theorem 1 (general bound)**.** *Let the true object region be denoted by Ω^* and let it be composed of P connected regions Ω^*_p (where $p = 1...P$). Similarly the true background region be denoted by Γ^* and let it be composed of q connected*

DOI:10.15774/PPKE.ITK.2014.005

regions Γ^*_q (where $q = 1...Q$). Assume that $F > 0$ in Ω^* and $F < 0$ in Γ^*. At initialization, C is chosen such that $\Omega = \cup_i \Omega_i$, $\Gamma = \cup_j \Gamma_j$ and $\Omega^*_p \cap \Omega \neq \emptyset$, $\forall p = 1...P$ and $(D \setminus \Omega) \cap \Gamma^*_q \neq \emptyset$, $\forall q = 1...Q$. Then, the Shi LSM converges to Ω^* in $N_{it} \leq \max(\max_i(|\Omega_i|), \max_j(|\Gamma_j|))$ iterations, where $|.|$ denotes the number of elements in the region.

Theorem 2 (convex bound)**.** *Let the true object region Ω^* be composed of P connected regions Ω^*_p (where $p = 1...P$) and the true background region Γ^* be composed of q connected regions Γ^*_q (where $q = 1...Q$). Assume that $F > 0$ in Ω^* and $F < 0$ in Γ^*. At initialization, C is chosen such that $\Omega = \cup_i \Omega_i$, $\Gamma = \cup_j \Gamma_j$ and $\Omega^*_p \cap \Omega \neq \emptyset$, $\forall p = 1...P$ and $(D \setminus \Omega) \cap \Gamma^*_q \neq \emptyset$, $\forall q = 1...Q$. If either Ω^* or Γ^* is convex than the Shi LSM converges to Ω^* in $N_{it} \leq \max(\max_i(B_{\Omega_i}), \max_j(B_{\Gamma_j}))$ iterations, where B denotes the diameter of the given region.*

Figure 3.6 shows two sample objects. While Figure 3.6(d) shows a concave object requiring a number of iterations as its number of pixels in the worst case, Figure 3.6(c) shows a convex object requiring a number of iterations upper bounded by its diameter in the worst case.

Table 3.2 explains through an example the effect of initial condition on the bounds. The resolution of the image is 128×128 pixels, the initial condition configuration is a chessboard like pattern. The number of squares was placed in n rows and n columns according to the values of the first row of the Table. Second and third rows show the general and convex bounds corresponding to initial condition configuration. The last two rows contain the number of iterations required to converge to the objects shown in Figure 3.6.

Thesis 2.2 I proved that the evolution of the Shi method can be mapped efficiently to many core architectures provided it is started from an initial condition that minimizes the bounds stated in thesis claim 2.1. I implemented it on two architectures: on CNN-UM and on GPU. The results supported the claims.

The smaller the connected regions in the initial condition, the lesser the required number of iterations to be able to converge. This kind of initial condition is used seldom because the number of processed pixels is $O(N \times M)$ in one iteration in the case of an $N \times M$ image since the small curves fill the whole image. In the case of an evolution starting from an initial condition containing a single

DOI:10.15774/PPKE.ITK.2014.005

curve one iteration processes $O(N + M)$ pixels. It shall be noted is an initial condition is “far” from the true object region then the number of pixels to be processed increases to $O(k(N+M))$ where $k \sim \max(N, M)$ leading to complexity $O(N \times M)$. Since the initial conditions are “far” from the real object in most cases the complexity of the two different evolution is asymptotically the same.

It follows from thesis claim 2.1 that densely placed curves with small diameters keep the worst case bound on the number of iterations according to the theorems low. On the Eye-RIS VS the execution time of one iteration is independent from the type of initial condition while in the case of GPU a mild deviation is experienced together with the drastic decrease of the number of iterations.

The algorithm mapped to CNN-UM is implemented on the Eye-RIS 1.3 VS. The realization uses only simple templates, one step of the algorithm is performed in $400 - 440\mu s$ on a QCIF image. It must be noted that the actual computing is finished within $60 - 70\mu s$ and the remaining time $(340 - 370\mu s)$ is required for the data movement from the main memory of the Eye-RIS (on the Altea NIOS-II microprocessor) to the Q-Eye chip memory.

The execution times of the algorithm mapped to GPU are summarized in Table 3.1. It is clear that evolutions started from the proposed initial condition family perform much better in all cases than the ones started from conventional initial conditions. In an extreme case it caused 24 times speedup ($2,048 \times 2,048$ image resolution, $210 \cdot 560$ vs. $7 \cdot 684$).

It can be seen that both on CNN-UM and GPU a significant speedup can be achieved in the case of the LS evolution of Shi if the proposed initial condition family is used.

Publications connected to this thesis group: [II, III, IV]. The thesis claim is specified and paraphrased in details in the third chapter of my dissertation.

4.3 Application fields

I demonstrated that it is possible to perform 2D to 3D registration during image guided therapy applications at the speed of (0.5-10 fps). This is essential and has great impact on the following applications. Furthermore, the problem was solved with the constant consulting with field experts from GE Healthcare and

DOI:10.15774/PPKE.ITK.2014.005

the technical knowledge and code-base were forwarded to the French research and development team.

The claims of the second thesis group can be utilized for faster segmentation or detection. The application fields of these methods are known. Naturally I emphasize the analysis of medical images. It is straightforward that I managed to utilize an initial condition that was considered unfeasible until now. Additionally the results from thesis claim 2.1 give guarantee which is essential in time critical applications.

DOI:10.15774/PPKE.ITK.2014.005

Appendix A

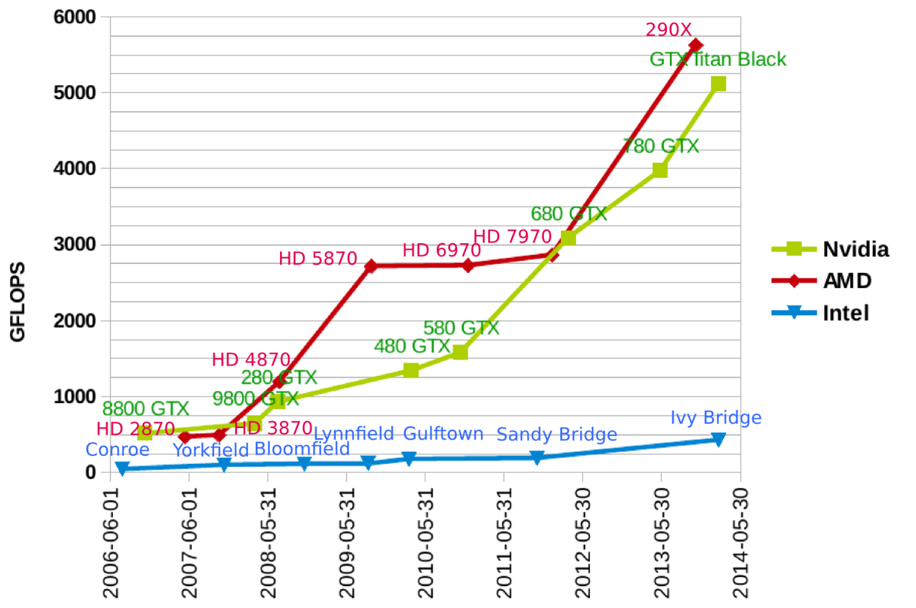
GPU

In the past 6-10 years GPUs have become programmable, massively parallel, manycore devices with very high theoretical computing capacity and bandwidth (see Figure A.1). The reason behind the anomaly of GPU an CPU theoretical floating point capacity is that GPUs are specialized for exactly these kinds of operations (multiplication, addition). On the contrary, CPUs have to handle a lot of other kind of operation types (integer, transcendental operation, division etc.) and a large portion of the chip is filled with cache memories and flow control.

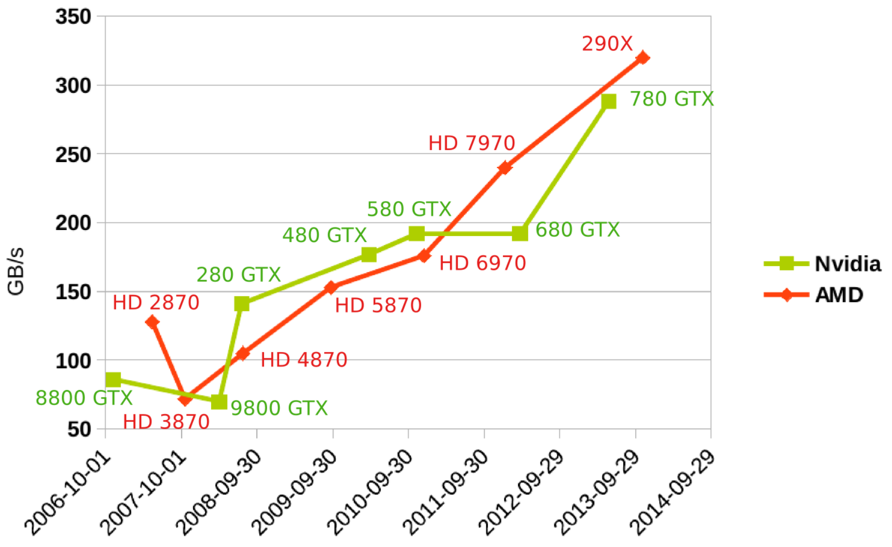
Two vendors provide programmable GPUs, AMD and Nvidia. During my work Nvidia GPUs were used. These devices are programmable through general purpose APIs. One is the CUDA platform [39] and the other is the OpenCL standard [40]. The former one is Nvidia specific, and OpenCL is a royalty-free standard for cross-platform, heterogen parallel programming. It is implemented by many vendors. In this dissertation the CUDA platform an ecosystem is used for measurements in Chapter 2 and OpenCL is used for measurements in Chapter 3.

The CUDA ecosystem consists of several components: the CUDA enabled GPU, the driver, the middleware and libraries, and the selected language. There are a lot of CUDA libraries and middleware available "off the shelf" (Fourier transform, BLAS, performance primitives, LAPACK, sparse solver, random generator, Matlab primitives, Mathematica routines etc.). They are fairly optimized and represent a good trade-off between fast realization and performance. The OpenCL consists of the single header API (free) and its implementation that is provided by the vendors like Intel, IBM, Apple, AMD, Nvidia, QUALCOM,

DOI:10.15774/PPKE.ITK.2014.005



(a) GFLOPS



(b) bandwidth

Figure A.1: Theoretical GFLOPS of GPUs an Intel CPUs and theoretical bandwidth of GPUs. The computing capacity increase shows exponential characteristics while the bandwidth increase is more likely to be linear. It is clear that GPUs have one order of magnitude higher theoretical computing capacity and this seems to be constant over this period not counting small variations. However, there are small fluctuations the two vendor manufacture GPUs having similar theoretical performance.

DOI:10.15774/PPKE.ITK.2014.005

Table A.1: Nomenclature dictionary, giving each concept the corresponding name in CUDA and OpenCL

CUDA name	OpenCL name
thread	work-item
block	work-group
grid of blocks of threads	ND-range
streaming multiprocessor (SM)	compute unit (CU)
cuda core	processing element
local memory	private memory
shared memory (per SM)	local memory (per CU)
texture object	read only image object
surface object	write only image object

ARM, etc. No further tools are provided. Most concept in the two nomenclatures have one to one correspondence as it is shown in Table A.1

A.1 Programming model

A C like function that can be executed on the GPU is called a kernel. This is a CUDA C/OpenCL extension to the C++/C language. During execution, the kernel is executed N times in parallel by N different threads. Any call to a kernel must specify an execution configuration for that call. This defines not just the number of threads to be launched but the arrangement of groups of threads to blocks and blocks to a grid. The dimensionality of a block and a grid can be one, two or three. The number of threads in a block is referred to as block size and limited to 1024.

Thread blocks are required to be independent, namely they have to be able to be executed in any order in parallel or in series. Threads in a thread block can cooperate and share information via on-chip memory space called shared memory. One can place synchronization points and barriers within the source

DOI:10.15774/PPKE.ITK.2014.005

code to regularize the access and to ensure the validity of the memory content.

The programming model assumes that the CUDA threads are executed on a physically separate device and works as an accelerator/coprocessor for given types of operations. The main program is executed on the CPU called host and the kernel is called by the host. Additionally, the device (GPU) and the host (CPU) does not share a common memory space. Therefore, the host program manages the allocation and deallocation of memory spaces in the global memory of the device.

A.2 Memory

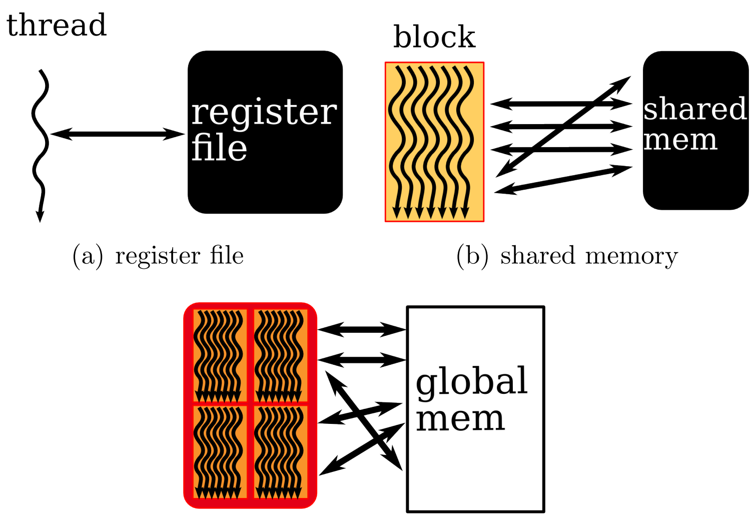
Threads can access data from multiple memory spaces. There are three basic hierarchical levels. Each thread has its own per-thread local/private memory. Threads in a block share a given amount of on-chip shared memory. The third space is called global memory accessible from all threads and can have the lifetime of the application itself. The memory hierarchy can be seen in Figure A.2. There are two additional memory spaces readable by all threads: texture and constant memory spaces. The global, texture and constant memory spaces are optimized for different usage. Texture memory also offers several kind of addressing modes and data filtering for given types.

A.2.1 Texture memory

Texture memory is read from kernels using dedicated functions. This is called texture fetch. Each fetch specifies a parameter called texture reference. The reference specifies:

- The texture and memory space is bound together via the texture reference. A given memory region may bound to several different reference at the same time.
- The dimension of the texture can be one, two or three dimensional. Additionally, the number of texels (texture elements) per dimension are given in the reference.

DOI:10.15774/PPKE.ITK.2014.005



(a) register file (b) shared memory

(c) global memory

Figure A.2: Memory hierarchy of Nvidia GPUs.

- The type of a texel can be a 1, 2 or 4 component vector of primitive data types (float, char, short, int, unsigned types, etc).
- The read mode can be normalized or unnormalized. In the first case the coordinates are normalized into the range [-1.0;1.0] or [0.0;1.0]. In the second case no conversion is performed.
- The addressing mode. This specifies the behavior on the boundaries and in the case of out of range requests: clamped, circular, mirrored, or wrapped.
- The filter mode which specifies the return value based on the input coordinates. Linear filter mode performs linear interpolation: bilinear in 2D and trilinear in 3D. Point filter mode returns the texel nearest to the input reading location.

It is a cached, read only, globally visible space. In graphics rendering textures are used widely and the hardware components are usually optimized for 2D locality. The details of the caching are not revealed by Nvidia. However, a work [74] gave a detailed micro-benchmark allowing to predict some features. In this work the 280 GTX GPU was considered to have compute capability 1.3.

DOI:10.15774/PPKE.ITK.2014.005

This architecture has two levels of texture cache L1 and L2, 5KB and 256 KB respectively. It was proved experimentally that texture reading does not reduce reading latency but reduce DRAM bandwidth demand.

A.2.2 Register file

The number of 32-bit registers per SM is varying from 8K-64K depending the micro architecture. Devices with compute capabilities 1.x have 8K, or 16K 32-bit register file. Fermi type devices have 32K 32-bit, and Kepler devices have 64K 32-bit register file.

A.2.3 Global memory

Global memory is accessible by all threads and physically placed off chip. Accessing it causes a delay of 400-600 clock cycles. On devices with compute capability 1.x it is uncached, later architectures have both L1 and L2 caches. 2.x devices have 16KB or 48KB L1 cache in each SM. It is configurable from the host program. The physical size of the L2 cache is 768 KB. In compile time it can be decided to use the L1 cache or just the L2 cache. The cache line is 128 byte.

A.2.4 Shared memory

Shared memory is a non-cached, per-SM memory space used by threads in a block to share data with other threads from the same block. The amount of shared memory is varying. Devices with compute capability 1.x have 16KB per block. The parameters of the kernel function occupy also shared memory so this reduces slightly its size. On Fermi and Kepler architectures, its amount is 16KB or 48KB depending on the choice of L1 cache size. It is organized into 16 banks on Tesla architecture and 32 on Fermi and Kepler GPUs. The read latency according to [74] is less than 40 clock cycles on devices with 1.x compute capability.

DOI:10.15774/PPKE.ITK.2014.005

A.3 Compilation and execution flow

Kernels can be written for CUDA in CUDA C/C++ or in CUDA instruction set architecture (ISA) called parallel thread execution (PTX). In both cases not counting few minor corner cases the code is device independent and has to be compiled by nvcc to produce executable binary (cubin). In the case of OpenCL, kernels can be implemented in the OpenCL extension of the C language that is vendor independent or in vendor dependent form that is loaded as executable binary.

The compilation can be off-line or just in time (JIT) in the case of CUDA. Off-line compilation separates host and device code first and later from the device code creates PTX code and then a cubin object. Just in time compilation allows the host program to load a PTX for further compilation to cubin code or directly a cubin object. In OpenCL only JIT compilation exists, however the binary object of the compiled kernel function can be queried by the OpenCL API for later use.

In CUDA one can chose the PTX version to be compiled to. In this way, significant differences can be experienced in performance. For example, arithmetic can be IEEE compliant or not. While the IEEE compliant version is complete within 200-250 clock cycles, the fast version is complete within 40-60 cycles in the case of single precision floating point division. Similarly, the presence or complete absence of caching can modify the performance. In OpenCL we have fewer parameters to drive the compilation of the kernel.

In the case of Nvidia devices threads within a block are arranged into consecutive groups called warps during execution. A warp is the group of threads scheduled together physically on the device. Threads within a warp start together but have their own instruction address counter and register state so they can branch, diverge and converge. However, this is very inefficient. The warp is the parameter of the physical device and it is independent from the used API.

A.4 Architecture of GPUs

Nvidia architecture in general Physically a device consists of SMs, cache and memory controllers. An SM contains cuda cores, special function units

DOI:10.15774/PPKE.ITK.2014.005

(SFU), warps schedulers, register files, and shared memory. The SFU is a dedicated floating point unit executing built-in floating point functions like square root, exponential, sin, cos, logarithm, etc. Nvidia refers to this arrangement as single instruction multiple threads (SIMT).

Groups of SMs compose the texture processing clusters (TPCs). These organizations share some additional hardware elements like texture caches, texture fetching units. These elements are invisible to the programmers.

The compute capability of a device is defined by two numbers. The first is a major revision number and the second is a minor revision number. Major revision numbers indicate same core architecture. 1 denotes Tesla architecture, 2 is for Fermi architecture, and 3 is for Kepler architecture. Unfortunately, the word 'Tesla' has two different meanings. It can mean an architecture family with compute capability 1.X. The other meaning is a product line for GPUs made for high performance general purpose computing. Minor revision number indicates fine architectural or capability differences for example 8800GTX: 1.0; 8800GT: 1.1; 280GTX: 1.3; Tesla C2050 and 580 GTX: 2.0.

Nvidia Tesla architecture There are 8 cores, 32 KB SDRAM as shared memory, 2 SFUs and 1 warp scheduler in a SM in this architecture. This family incorporates the 8, 9, 100, 200, 300 and Tesla 870-1070 series of GPUs.

Nvidia Fermi architecture There are 32 or 48 cores, 64 KB SDRAM, 4 SFUs and 2 warp schedulers in an SM in this architecture. This architecture introduces L1, L2 caching. The L2 is 768 KB accessible from all SMs. The L1 is implemented per SM and resides in the SDRAM in a configurable way. The 64 KB is divided into two parts: the L1 and the shared memory. Their size can be 16 KB or 48 KB as they are configured. This family includes the 400, 500 and Tesla C2050-M2090 series of GPUs.

Nvidia Kepler architecture There are 192 cores, 64+48 KB SDRAM, 32 SFUs and 4 warp schedulers in a SM in this architecture. The 64 KB is configurable between the shared memory and the L1 cache as in the Fermi architecture. Additionally the architecture introduces an additional 48 KB read only data cache

DOI:10.15774/PPKE.ITK.2014.005

that is accessible by the developer. This family includes the 600, 700 and Tesla K10-K40 series of GPUs.

AMD Southern Islands architecture The device consists of CUs, L2 cache, global data share memory, and memory controllers. Each CU contains 64 vector processor, a scalar core, 64×256×32 bit vector register file, 512×32 bit scalar register file, 64 KB local data share memory and a texture unit. Unlike Nvidia, AMD has made the complete ISA of the devices open. So, the exact capabilities of the hardware are known.

DOI:10.15774/PPKE.ITK.2014.005

Appendix B

The CNN Computer

Cellular neural networks (CNNs) are regular, single or multi-layer, parallel processing structures with analog nonlinear dynamic units called cells. The state of the cells is continuous in time. Their connectivity is local in space. The program of a CNN is determined by the pattern and strength of the local connectivity, the so-called template. The time-evolution of the cells, “driven” by the template and the cell dynamics, represents the elementary instruction in CNN (both in equilibrium or non-equilibrium states of the network can define results). The standard CNN equation [60] contains first order cells placed on a regular grid of one layer.

The CNN Universal Machine (CNN-UM, [75]) is a cellular wave computer architecture that includes CNN dynamics as its main instruction. To ensure stored programmability, a global programming unit is added to the standard CNN and for reuse of intermediates, each cell contains a few local memories. Additionally, every cell might be equipped with local sensors to provide input (for example optical) and further circuits to perform operations per cell.

Using the CNN-UM, one can design and run analog and logic CNN wave algorithms. It is known that CNN-UM is universal as a Turing Machine [75] in the sense that a CNN-UM can present all the behaviors that a predefined CNN dynamics can show. Furthermore, it is universal as a nonlinear operator as well. Therefore, many problems can be solved by this machine. Its structure fits naturally for image processing. There are lots of methods for solving image processing problems based on partial differential equations [7] which need huge computational power in the most cases. Most of these kind of problems can be

DOI:10.15774/PPKE.ITK.2014.005

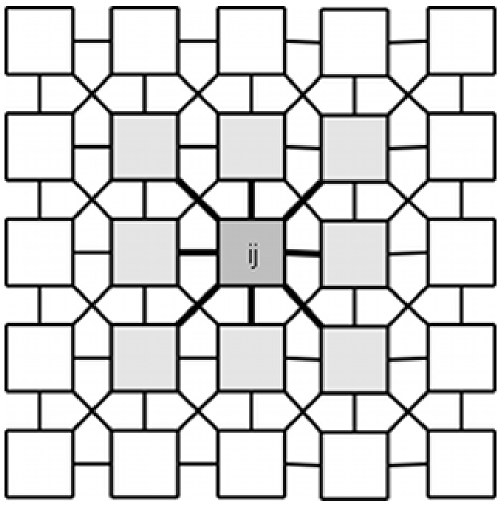


Figure B.1: CNN cells in 2D rectangular grid. Each cell has connections to its neighbors within the neighborhood radius. In this case the radius type is nearest neighbor and the dark gray cell has 8 additional neighbors marked with light grey.

transformed into CNN algorithm too.

B.1 Standard CNN dynamics

The cellular nonlinear network (CNN) is a locally connected, analog dynamical cell network, which has two or more dimensions. The common CNN architecture consists of an M×N rectangular grid of cells $c(i,j)$ with Cartesian coordinate $(i,j) i = 1..M, j = 1..N$

Each cell is connected to its (nearest) neighbors within a given range (N_r). This is nicely illustrated in Figure B.1.

A template has two main parts, feedforward and feedback matrices. These parts are called A and B templates. The z on Equation (B.1) is the offset (bias) term. In the simplest case the template is given by 19 numbers, 9 feedback, 9 feedforward and one bias terms. This 19 number template is an elementary operation of CNN-UM and codes a complex spatial-temporal dynamics. A CNN algorithm might contain templates and logical operations as well. The following

DOI:10.15774/PPKE.ITK.2014.005

differential equation system describes the dynamics of the network:

$$\frac{d}{dt}x_{ij}(t) = -x_{ij}(t) + \sum_{kl\,\in\,N_r} A_{kl,ij}y_{kl}(t) + \sum_{kl\,\in\,N_r} B_{kl,ij}u_{kl}(t) + z_{ij} \tag{B.1}$$

Here x_{ij}, u_{ij}, y_{ij} stand for the state, input, and output of the cell ij.

Each cell has a state, an input, and an output that is a nonlinear function of the state. The nonlinear function can be arbitrary but in the majority of the cases it is the piecewise linear approximation of a sigmoid function defined as:

$$f(x) = \begin{cases} 1 & \text{if } x \geq 1 \\ x & \text{if } -1 \leq x \leq 1 \\ -1 & \text{if } x \leq -1 \end{cases} \tag{B.2}$$

In cases when $A_{kl,ij}$ and $B_{kl,ij}$ does not depend on (i, j) the template is called space invariant. Since the 2D and regular nature of the grid of cells, the state, input and output of the whole CNN can be represented as an image with M×N pixels. The value of a pixel varies between -1 (white) an 1 (black) in the case of input and output, the possible range of the state is unbound theoretically.

The structure of linear templates used in the LS evolution algorithm can be seen in Equation (B.3). Feedback and feed forward templates A and B are central symmetrical. So each template element is described by 3 numbers (single central element, neighbors of the central element according to the 4 connectivity and 4 elements in the corners). Typical parameter values of the used operators are listed in Table B.1.

$$A = \begin{bmatrix} a_2 & a_1 & a_2 \\ a_1 & a_0 & a_1 \\ a_2 & a_1 & a_2 \end{bmatrix}; B = \begin{bmatrix} b_2 & b_1 & b_2 \\ b_1 & b_0 & b_1 \\ b_2 & b_1 & b_2 \end{bmatrix}; z \tag{B.3}$$

DOI:10.15774/PPKE.ITK.2014.005

Table B.1: Template parameter values

	A			B			z	Boundary
Template	a_0	a_1	a_2	b_0	b_1	b_2	z	condition
AND	2	0	0	1	0	0	-1	Dirichlet, 0
ANDNOT	2	0	0	-1	0	0	-1	Dirichlet, 0
OR	2	0	0	1	0	0	1	Dirichlet, 0
DIL4	0	0	0	1	1	0	4	Dirichlet, -1
ERODE4	0	0	0	1	1	0	-4	Dirichlet, -1

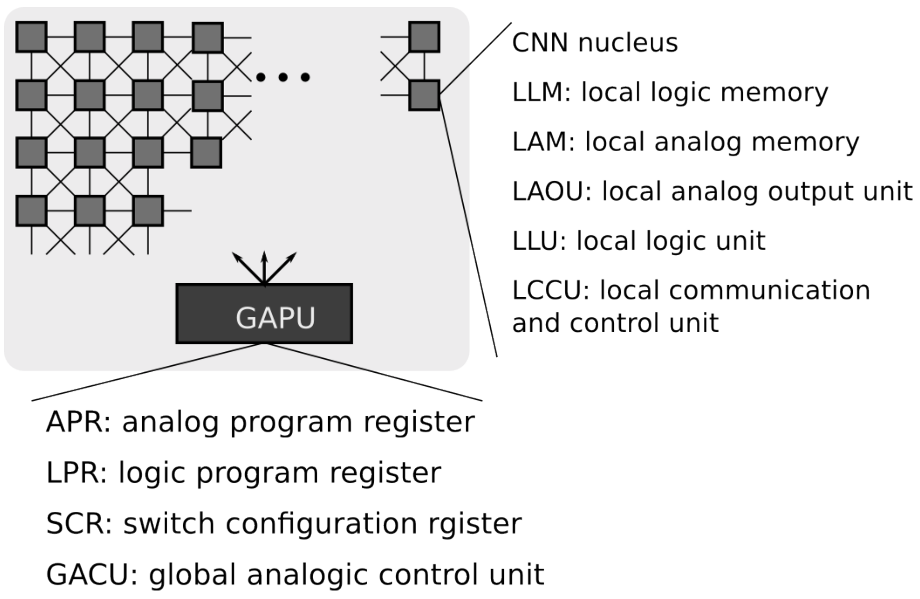

GAPU
CNN nucleus
LLM: local logic memory
LAM: local analog memory
LAOU: local analog output unit
LLU: local logic unit
LCCU: local communication and control unit
APR: analog program register
LPR: logic program register
SCR: switch configuration rgister
GACU: global analogic control unit

Figure B.2: Illustration of a CNN-UM.

DOI:10.15774/PPKE.ITK.2014.005

B.2 CNN Universal Machine

The CNN-UM is based on the standard CNN (see Figure B.2). This is a programmable analogical processor array with own language defined by the template operations and several VLSI implementations. The cells contain local analog and logic memories and may contain sensors, and miscellaneous aiding circuitry. These universal cells are controlled by the global analogic programming unit (GAPU). The GAPU has four main parts. These parts are responsible for the analog program, the logical program, switch configuration and the control flow.

Algorithms designed for CNN-UM can be represented on a universal machine on flows (UMF) diagram. The UMF is a purely continuous computational model capable of describing and characterizing the capabilities of a CNN-UM. From a practical point of view a UMF diagram of a CNN-UM consists of the following primitives: template execution (specifying the template values, input and state images, boundary condition, execution time), branching, looping, and the flow of data. It is a directed graph, which may have cycles in itself.

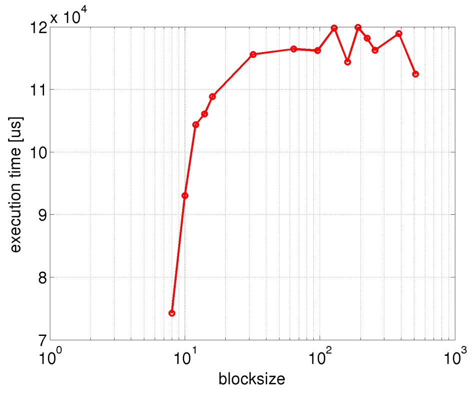
DOI:10.15774/PPKE.ITK.2014.005

Appendix C

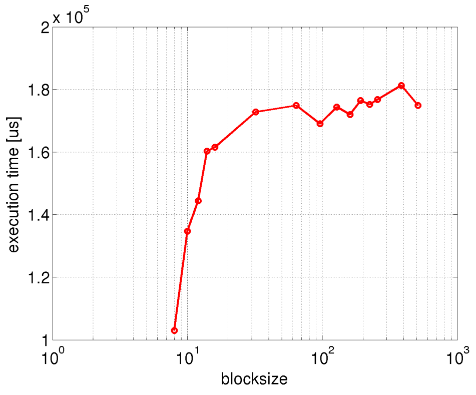
Block size dependence of DRR rendering

In Figures C.1-C.4 the block size dependence of execution time can be seen on the four GPUs: 8800 GT, 280 GTX, Tesla C2050, and 580 GTX. The block size is on the (logarithmic) X axes, the execution time in μs is on the y axes. The measured block sizes are 8, 10, 12, 14, 16, 32, 64, 96, 128, 160, 192, 224, 256, 384, and 512. Each subfigure shows the characteristic of the given GPU with a fixed number of threads as follows: 1024, 15360, 20480, 25600, 30720, and 35840.

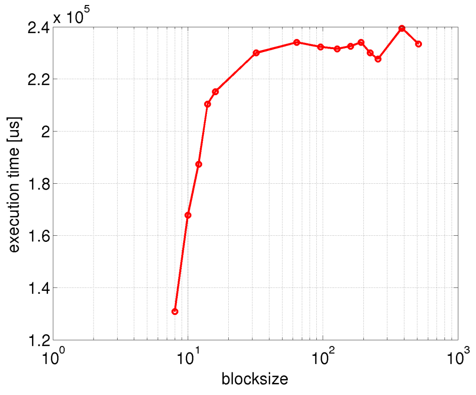
DOI:10.15774/PPKE.ITK.2014.005



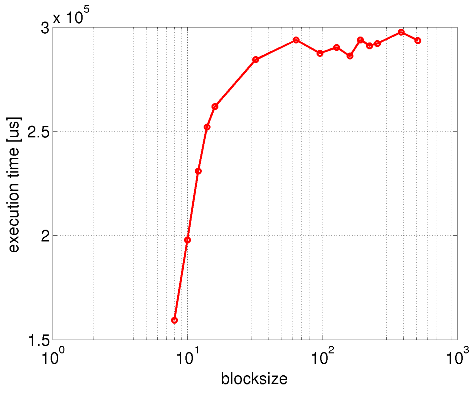
(a) 10240



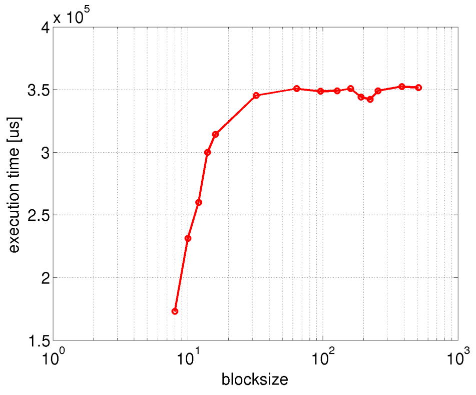
(b) 15360



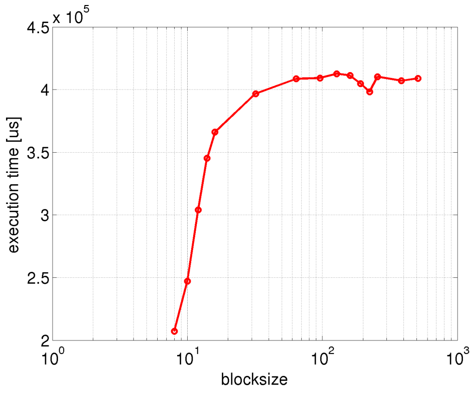
(c) 20480



(d) 25600



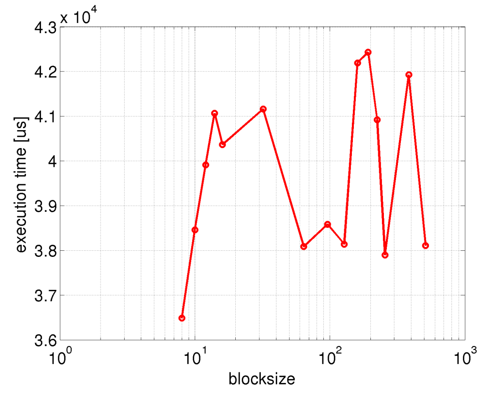
(e) 30720



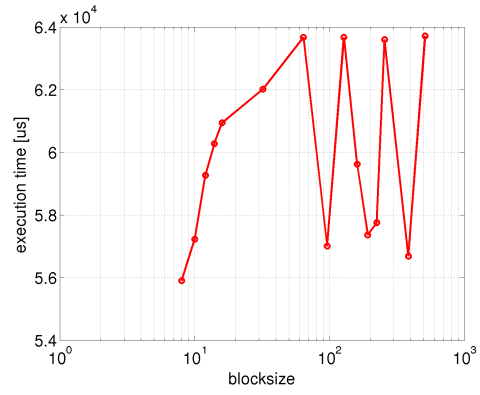
(f) 35840

Figure C.1: 8800 GT–Large thread numbers.

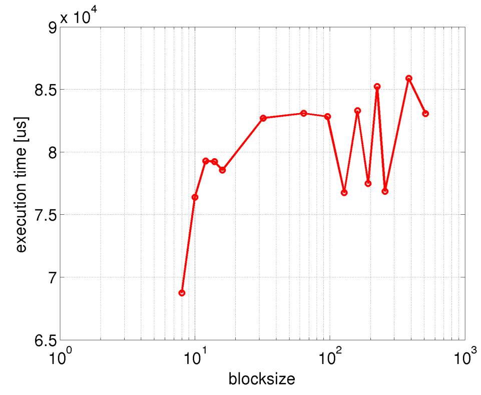
DOI:10.15774/PPKE.ITK.2014.005



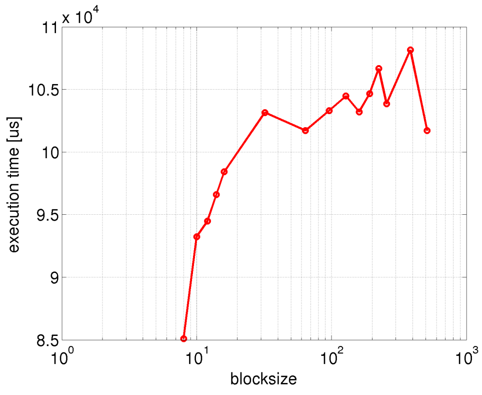
(a) 10240



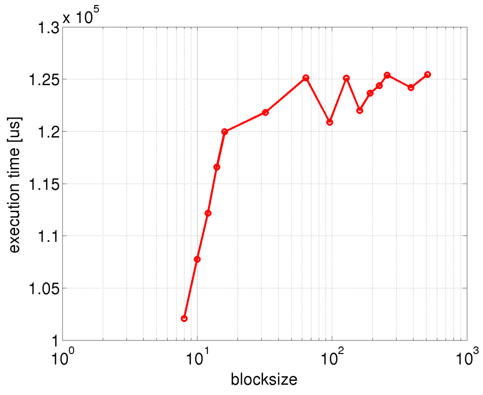
(b) 15360



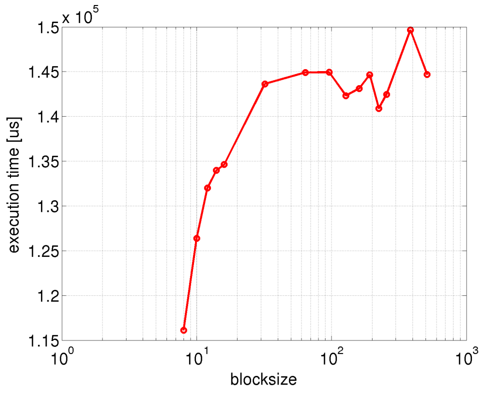
(c) 20480



(d) 25600



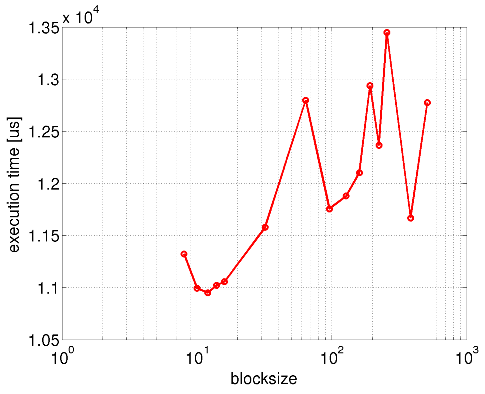
(e) 30720



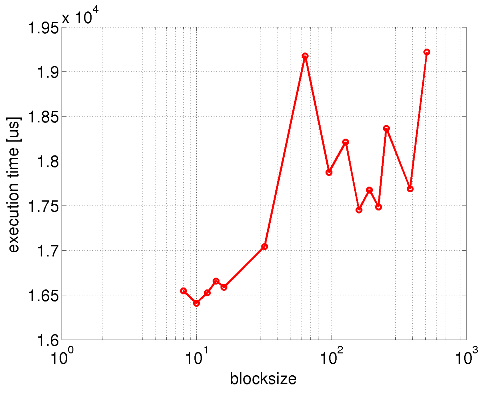
(f) 35840

Figure C.2: 280 GTX–Large thread numbers.

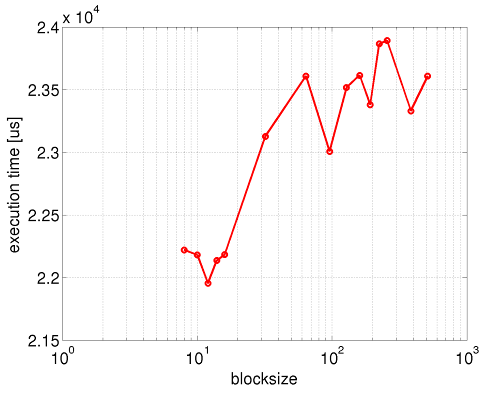
DOI:10.15774/PPKE.ITK.2014.005



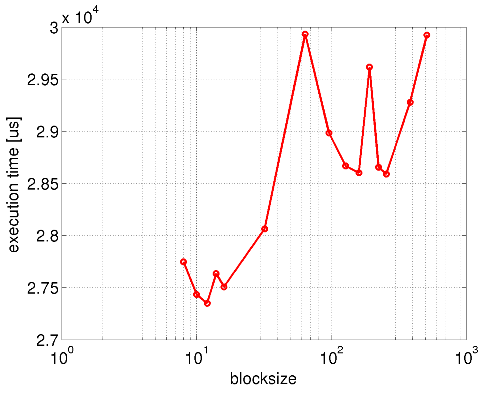
(a) 10240



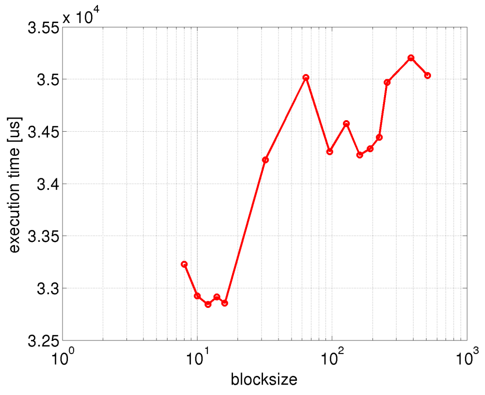
(b) 15360



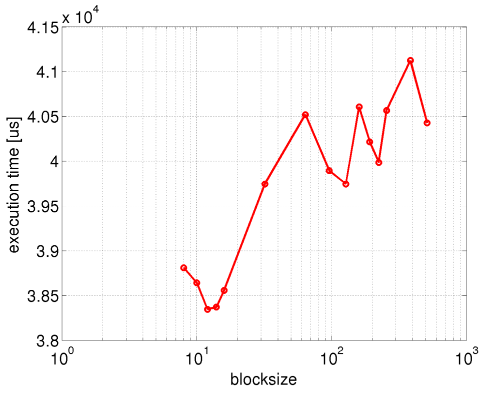
(c) 20480



(d) 25600



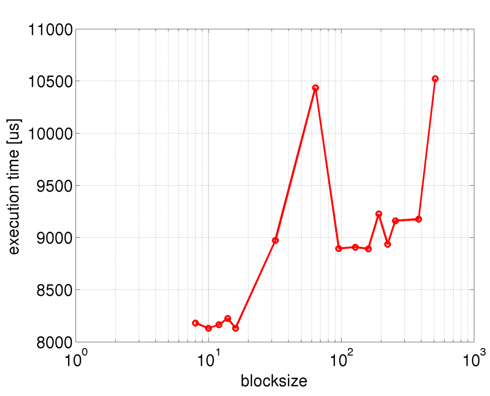
(e) 30720



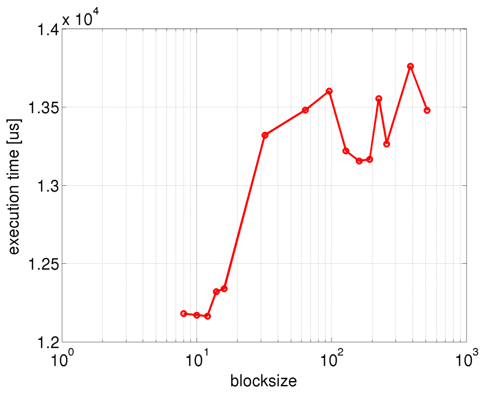
(f) 35840

Figure C.3: Tesla C2050–Large thread numbers.

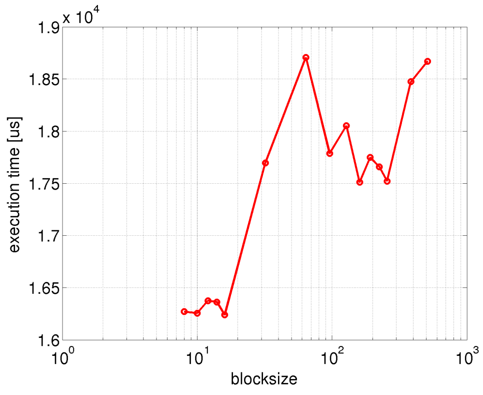
DOI:10.15774/PPKE.ITK.2014.005



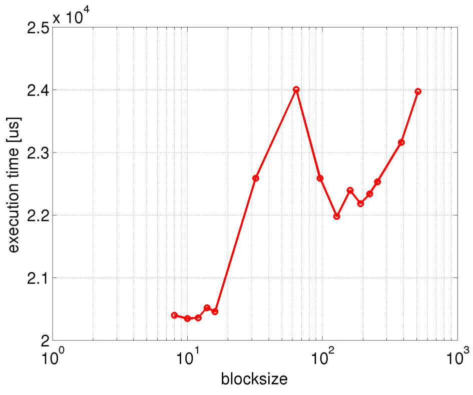
(a) 10240



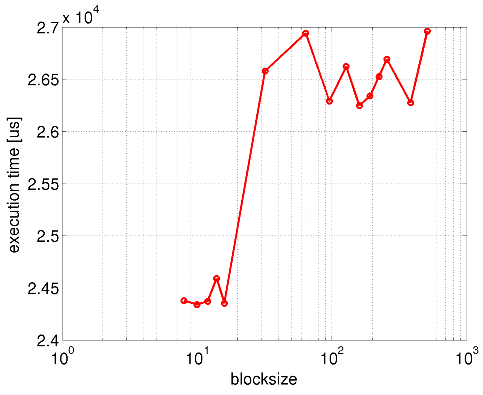
(b) 15360



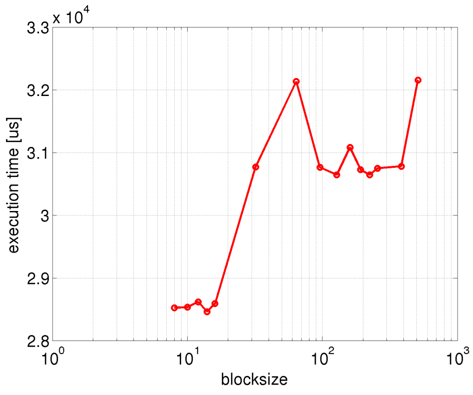
(c) 20480



(d) 25600



(e) 30720



(f) 35840

Figure C.4: 580 GTX–Large thread numbers.

DOI:10.15774/PPKE.ITK.2014.005

The author's publications

[I] **G. J. Tornai**, G. Cserey, and I. Pappas, “Fast DRR generation for 2D to 3D registration on GPUs,” *Medical Physics*, vol. 39, no. 8, pp. 4795–4799, 2012.

[II] **G. J. Tornai** and G. Cserey, “Initial condition for efficient mapping of level set algorithms on many-core architectures,” *EURASIP Journal on Advances in Signal Processing*, 2014:30. doi:10.1186/1687-6180-2014-30

[III] **G. J. Tornai**, G. Cserey, and A. Rák, “Spatial-Temporal level set algorithms on CNN-UM,” in *International Symposium on Nonlinear Theory and its Application, (NOLTA), 2008*, pp. 696–699, 2008.

[IV] **G. J. Tornai** and G. Cserey, “2D and 3D level-set algorithms on GPU,” in *Cellular Nanoscale Networks and Their Applications (CNNA), 2010 12th International Workshop on*, p. 1–5, 2010.

A. Horváth, **G. J. Tornai**, A. Horváth and G. Cserey, “Fast, parallel implementation of particle filter on GPU,” *EURASIP Journal on Advances in Signal Processing*, 2013:148. doi:10.1186/1687-6180-2013-148

DOI:10.15774/PPKE.ITK.2014.005

References

[1] J. L. Shin, K. Tam, D. Huang, B. Petrick, H. Pham, C. Hwang, H. Li, A. Smith, T. Johnson, F. Schumacher, *et al.*, “A 40nm 16-core 128-thread CMT SPARC SoC processor,” in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, p. 98–99, 2010. 1

[2] D. Wendel, R. Kalla, R. Cargoni, J. Clables, J. Friedrich, R. Frech, J. Kahle, B. Sinharoy, W. Starke, S. Taylor, *et al.*, “The implementation of POWER7: a highly parallel and scalable multi-core high-end server processor,” in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, p. 102–103, 2010. 1

[3] P. Markelj, D. Tomazevic, B. Likar, and F. Pernus, “A review of 3D/2D registration methods for image-guided interventions,” *Medical Image Analysis*, Mar. 2010. 2, 6, 9

[4] G. P. Penney, P. G. Batchelor, D. L. Hill, D. J. Hawkes, and J. Weese, “Validation of a two-to three-dimensional registration algorithm for aligning preoperative CT images and intraoperative fluoroscopy images,” *Medical physics*, vol. 28, p. 1024, 2001. 2, 6

[5] J. Wu, M. Kim, J. Peters, H. Chung, and S. S. Samant, “Evaluation of similarity measures for use in the intensity-based rigid 2D-3D registration for patient positioning in radiotherapy,” *Medical Physics*, vol. 36, no. 12, p. 5391, 2009. 2

[6] J. A. Sethian, *Level set methods and fast marching methods: evolving interfaces in computational geometry, fluid mechanics, computer vision, and*

DOI:10.15774/PPKE.ITK.2014.005

materials science. No. 3 in Cambridge monographs on applied and computational mathematics, New York: Cambridge Univ Pr, 2nd edition ed., 1999. 3, 34, 37, 38

[7] G. Sapiro, *Geometric partial differential equations and image analysis*. New York: Cambridge University Press, 2001. 3, 35, 37, 90

[8] V. Caselles, R. Kimmel, and G. Sapiro, "Geodesic active contours," *International Journal of Computer Vision*, vol. 22, pp. 61–79, Feb. 1997. 3

[9] G. P. Penney, D. C. Barratt, C. S. K. Chan, M. Slomczykowski, T. J. Carter, P. J. Edwards, and D. J. Hawkes, "Cadaver validation of intensity-based ultrasound to CT registration," *Medical Image Analysis*, vol. 10, no. 3, p. 385–395, 2006. 6

[10] W. Birkfellner, M. Figl, J. Kettenbach, J. Hummel, P. Homolka, R. Schernthaner, T. Nau, and H. Bergmann, "Rigid 2D/3D slice-to-volume registration and its application on fluoroscopic CT images," *Medical Physics*, vol. 34, no. 1, pp. 246–255, 2007. 6

[11] K. Mori, D. Deguchi, J. Sugiyama, Y. Suenaga, J. ichiro Toriwaki, C. R. M. Jr, H. Takabatake, and H. Natori, "Tracking of a bronchoscope using epipolar geometry analysis and intensity-based image registration of real and virtual endoscopic images.," *Medical Image Analysis*, vol. 6, no. 3, p. 321, 2002. 6

[12] W. Birkfellner, M. Stock, M. Figl, C. Gendrin, J. Hummel, S. Dong, J. Kettenbach, D. Georg, and H. Bergmann, "Stochastic rank correlation: A robust merit function for 2D/3D registration of image data obtained at different energies," *Medical physics*, vol. 36, p. 3420, 2009. 6, 9, 10, 11, 17

[13] L. Zollei, E. Grimson, A. Norbash, and W. Wells, "2D-3D rigid registration of x-ray fluoroscopy and CT images using mutual information and sparsely sampled histogram estimators," in *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, vol. 2, pp. II–696–II–703 vol.2, 2001. 6, 9, 11, 17

DOI:10.15774/PPKE.ITK.2014.005

[14] T. Rohlfing, D. B. Russakoff, J. Denzler, K. Mori, and C. R. Maurer, "Progressive attenuation fields: Fast 2D-3D image registration without precomputation," *Medical Physics*, vol. 32, no. 9, pp. 2870—2880, 2005. 6, 9, 10

[15] A. Kubias, F. Deinzer, T. Feldmann, D. Paulus, B. Schreiber, and T. Brunner, "2D/3D image registration on the GPU," *Pattern Recognition and Image Analysis*, vol. 18, no. 3, p. 381–389, 2008. 6, 10, 11

[16] D. Russakoff, T. Rohlfing, K. Mori, D. Rueckert, A. Ho, J. Adler, and C. Maurer, "Fast generation of digitally reconstructed radiographs using attenuation fields with application to 2D-3D image registration," *Medical Imaging, IEEE Transactions on*, vol. 24, no. 11, pp. 1441–1454, 2005. 6, 9, 10

[17] L. Xu and J. W. Wan, "Real-time intensity-based rigid 2D-3D medical image registration using RapidMind multi-core development platform," in *Engineering in Medicine and Biology Society, 2008. EMBS 2008. 30th Annual International Conference of the IEEE*, pp. 5382–5385, 2008. 6, 11

[18] L. Zollei, J. Fisher, and W. Wells, "A unified statistical and information theoretic framework for multi-modal image registration," in *Information Processing in Medical Imaging* (J. Noble and C. Taylor, eds.), vol. 2732 of *Lecture Notes in Computer Science*, pp. 366–377, Springer Berlin / Heidelberg, 2003. 7, 11, 17

[19] H. M. Chan, A. C. Chung, S. C. Yu, and W. Wells, "2D-3D vascular registration between digital subtraction angiographic (DSA) and magnetic resonance angiographic (MRA) images," in *Biomedical Imaging: Nano to Macro, 2004. IEEE International Symposium on*, p. 708–711, 2004. 7

[20] K. Rhode, D. Hill, P. Edwards, J. Hipwell, D. Rueckert, G. Sanchez-Ortiz, S. Hegde, V. Rahunathan, and R. Razavi, "Registration and tracking to integrate x-ray and MR images in an XMR facility," *IEEE Transactions on Medical Imaging*, vol. 22, pp. 1369–1378, Nov. 2003. 7

DOI:10.15774/PPKE.ITK.2014.005

[21] T. P. L. Roberts, A. Martin, R. L. Arenson, W. P. Dillon, and C. B. Higgins, "Integrating x-ray angiography and MRI for endovascular interventions," *Medicamundi*, vol. 44, p. 2–9, Mar. 2000. 7

[22] E. van de Kraats, T. V. Walsum, J. Verlaan, and W. J. Niessen, "Noninvasive MR to 3D rotational X-Ray registration of vertebral bodies," in *SPIE Medical Imaging*, pp. 1101–1108, 2003. 7

[23] T. Dohi, R. Kikinis, A. Chung, W. Wells, A. Norbash, and W. Grimson, "Multi-modal image registration by minimising Kullback-Leibler distance," in *Medical Image Computing and Computer-Assisted Intervention — MICCAI 2002*, vol. 2489 of *Lecture Notes in Computer Science*, pp. 525–532, Springer Berlin / Heidelberg, 2002. 7

[24] A. Andronache, P. Cattin, and G. Székely, "Local intensity mapping for hierarchical non-rigid registration of multi-modal images using the cross-correlation coefficient," in *Biomedical Image Registration*, p. 26–33, 2006. 7

[25] H. Livyatan, Z. Yaniv, and L. Joskowicz, "Gradient-based 2-D/3-D rigid registration of fluoroscopic x-ray to CT," *Medical Imaging, IEEE Transactions on*, vol. 22, no. 11, pp. 1395–1406, 2003. 7

[26] A. Khamene, P. Bloch, W. Wein, M. Svatos, and F. Sauer, "Automatic registration of portal images and volumetric CT for patient positioning in radiation therapy," *Medical Image Analysis*, vol. 10, no. 1, p. 96–112, 2006. 7

[27] T. Rohlfing, D. B. Russakoff, M. J. Murphy, and C. R. J. Maurer, "An intensity-based registration algorithm for probabilistic images and its application for 2-D to 3-D image registration," in *Proc. SPIE*, vol. 4684, p. 581–591, 2002. 9

[28] M. F. Dorgham and S. Laycock, "Accelerated generation of digitally reconstructed radiographs using parallel processing," in *Proc. Medical Image Understanding and Analysis*, p. 14–15, 2009. 10, 11

DOI:10.15774/PPKE.ITK.2014.005

[29] J. Spoerk, H. Bergmann, F. Wanschitz, S. Dong, and W. Birkfellner, "Fast DRR splat rendering using common consumer graphics hardware," *Medical Physics*, vol. 34, no. 11, p. 4302, 2007. 10, 11

[30] C. Gendrin, H. Furtado, C. Weber, C. Bloch, M. Figl, S. A. Pawiro, H. Bergmann, M. Stock, G. Fichtinger, and D. Georg, "Monitoring tumor motion by real time 2D/3D registration during radiotherapy," *Radiotherapy and Oncology*, vol. 102, pp. 274–280, 2012. 10, 11, 30, 31

[31] O. M. Dorgham, S. D. Laycock, and M. H. Fisher, "GPU accelerated generation of digitally reconstructed radiographs for 2-D/3-D image registration," *Biomedical Engineering, IEEE Transactions on*, vol. 59, no. 9, pp. 2594–2603, 2012. 10, 11, 30

[32] G. J. Tornai, G. Cserey, and I. Pappas, "Fast DRR generation for 2D to 3D registration on GPUs," *Medical Physics*, vol. 39, no. 8, pp. 4795–4799, 2012. 9, 10, 11, 19, 29, 31, 76

[33] B. P. Selby, G. Sakas, S. Walter, W. D. Groch, and U. Stilla, "Selective X-Ray reconstruction and registration for pose estimation in 6 degrees of freedom," in *XXI Congress, Proceedings. International Archives of Photogrammetry, Remote Sensing and Spatial Geoinformation Sciences*, vol. 37-B5, (Beijing, China), pp. 799–804, 2008. 9

[34] D. Knaan and L. Joskowicz, "Effective Intensity-Based 2D/3D rigid registration between fluoroscopic X-Ray and CT," in *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2003* (R. Ellis and T. Peters, eds.), vol. 2878 of *Lecture Notes in Computer Science*, pp. 351–358, Springer Berlin / Heidelberg, 2003. 9

[35] J. Weese, R. Göcke, G. P. Penney, P. Desmedt, T. M. Buzug, and H. Schumann, "Fast voxel-based 2D/3D registration algorithm using a volume rendering method based on the shear-warp factorization," *Medical Imaging 1999: Image Processing*, vol. 3661, p. 802–810, 1999. 9

[36] D. Ruijters, B. M. ter Haar-Romeny, and P. Suetens, "GPU-accelerated digitally reconstructed radiographs," in *Proceedings of the Sixth IASTED*

DOI:10.15774/PPKE.ITK.2014.005

International Conference on Biomedical Engineering, (Innsbruck, Austria), pp. 431–435, ACTA Press, 2008. 11

[37] C. T. Metz, M. Schaap, S. Klein, A. C. Weustink, N. R. Mollet, C. Schultz, R. J. V. Geuns, P. W. Serruys, T. V. Walsum, and W. J. Niessen, "GPU accelerated alignment of 3-D CTA with 2-D x-ray data for improved guidance in coronary interventions," in *Biomedical Imaging: From Nano to Macro, 2009. ISBI'09. IEEE International Symposium on*, p. 959–962, 2009. 11

[38] J. Spoerk, C. Gendrin, C. Weber, M. Figl, S. A. Pawiro, H. Furtado, D. Fabri, C. Bloch, H. Bergmann, E. Gröller, and W. Birkfellner, "High-performance GPU-based rendering for real-time, rigid 2D/3D-image registration and motion prediction in radiation oncology," *Z Med Phys*, vol. 22, no. 1, pp. 13–20, 2012. 11

[39] NVIDIA, "CUDA C Programming Guide." http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf, Sept. 2011. 11, 48, 81

[40] K. Group, "OpenCL." http://www.khronos.org/opencl/, July 2011. 11, 48, 81

[41] NVIDIA, "Parallel thread execution ISA version 3.0," Jan. 2012. 12

[42] S. A. Pawiro, P. Markelj, F. Pernus, C. Gendrin, M. Figl, C. Weber, F. Kainberger, I. Nobauer-Huhmann, H. Bergmeister, M. Stock, D. Georg, H. Bergmann, and W. Birkfellner, "Validation for 2D/3D registration i: A new gold standard data set," *Medical Physics*, vol. 38, no. 3, p. 1481, 2011. 16, 75

[43] G. Healthcare, "GE Healthcare LightSpeed RT 16 CT." http://www3.gehealthcare.com/en/Products/Categories/GoldSeal_-_Refurbished_Systems/GoldSeal_Computed_Tomography/Gold_Seal_LightSpeed_RT_Series, maj. 28. 2014. 17

[44] "Radiology support devices website." http://www.rsdphantoms.com/rd_lung.htm, maj. 28. 2014. 17

DOI:10.15774/PPKE.ITK.2014.005

[45] Y. Shi and W. Karl, "A Real-Time algorithm for the approximation of Level-Set-Based curve evolution," *IEEE Transactions on Image Processing*, vol. 17, no. 5, pp. 645–656, 2008. 34, 36, 39, 70, 77

[46] D. Adalsteinsson and J. A. Sethian, "A fast level set method for propagating interfaces," *Journal of Computational Physics*, vol. 118, no. 2, pp. 269 – 277, 1995. 34

[47] D. Peng, B. Merriman, S. Osher, H. Zhao, and M. Kang, "A PDE-Based fast local level set method," *Journal of Computational Physics*, vol. 155, no. 2, pp. 410 – 438, 1999. 35

[48] T. Chan and L. Vese, "Active contours without edges," *Image Processing, IEEE Transactions on*, vol. 10, no. 2, pp. 266–277, 2001. 35, 49, 66

[49] N. Joshi and M. Brady, "Non-Parametric mixture model based evolution of level sets and application to medical images," *International Journal of Computer Vision*, vol. 88, pp. 52–68, 2010. 35

[50] L. Bertelli, S. Chandrasekaran, F. Gibou, and B. S. Manjunath, "On the length and area regularization for multiphase level set segmentation," *International Journal of Computer Vision*, vol. 90, no. 3, pp. 267–282, 2010. 35

[51] V. Caselles, F. Catté, T. Coll, and F. Dibos, "A geometric model for active contours in image processing," *Numerische mathematik*, vol. 66, no. 1, p. 1–31, 1993. 35, 50

[52] R. Malladi, J. A. Sethian, and B. C. Vemuri, "Shape modeling with front propagation: A level set approach," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 17, no. 2, p. 158–175, 1995. 35, 50

[53] Y. Chen, F. Huang, H. D. Tagare, M. Rao, D. Wilson, and E. A. Geiser, "Using prior shape and intensity profile in medical image segmentation," in *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, p. 1117–1124, 2003. 35

DOI:10.15774/PPKE.ITK.2014.005

[54] A. Yezzi and A. Mennucci, “Conformal metrics and true,” in *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on*, vol. 1, p. 913–919, 2005. 35

[55] G. Sundaramoorthi, A. Yezzi, A. Mennucci, and G. Sapiro, “New possibilities with sobolev active contours,” *International Journal of Computer Vision*, vol. 84, pp. 113–129, 2009. 35

[56] M. Rumpf and R. Strzodka, “Level set segmentation in graphics hardware,” in *Image Processing, 2001. Proceedings. 2001 International Conference on*, vol. 3, p. 1103–1106, 2001. 35

[57] A. Lefohn, J. Cates, and R. Whitaker, “Interactive, GPU-Based level sets for 3D segmentation,” in *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2003* (R. Ellis and T. Peters, eds.), vol. 2878 of *Lecture Notes in Computer Science*, pp. 564–572, Springer Berlin Heidelberg, 2003. 35

[58] M. Roberts, J. Packer, M. C. Sousa, and J. R. Mitchell, “A work-efficient GPU algorithm for level set segmentation,” in *Proceedings of the Conference on High Performance Graphics*, p. 123–132, ACM, 2010. 36

[59] O. Sharma, Q. Zhang, F. Anton, and C. Bajaj, “Multi-domain, higher order level set scheme for 3D image segmentation on the GPU,” in *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, p. 2211–2216, 2010. 36

[60] L. O. Chua and L. Yang, “Cellular neural networks: theory,” *Circuits and Systems, IEEE Transactions on*, vol. 35, no. 10, p. 1257–1272, 1988. 36, 90

[61] G. Cserey, C. Rekeczky, and P. Földesy, “PDE based histogram modification with embedded morphological processing of the Level-Sets,” *Journal of Circuits, Systems, and Computers*, vol. 12, no. 04, p. 519–538, 2003. 36

[62] C. Rekeczky and T. Roska, “Calculating local and global PDEs by analogic diffusion and wave algorithms,” in *Proceedings of the European Conference on Circuit Theory and Design*, vol. 2, p. 17–20, 2001. 36

DOI:10.15774/PPKE.ITK.2014.005

[63] D. Hillier, Z. Czeilinger, A. Vobornik, and C. Rekeczky, “Online 3-D reconstruction of the right atrium from echocardiography data via a topographic cellular contour extraction algorithm,” *Biomedical Engineering, IEEE Transactions on*, vol. 57, pp. 384 –396, Feb. 2010. 36

[64] M. Kass, A. Witkin, and D. Terzopoulos, “Snakes: Active contour models,” *International Journal of Computer Vision*, vol. 1, no. 4, p. 321–331, 1988. 36

[65] S. Osher and J. A. Sethian, “Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations,” *Journal of computational physics*, vol. 79, no. 1, p. 12–49, 1988. 36

[66] D. L. Vilarino and C. Rekeczky, “Pixel-level snakes on the CNN-UM: algorithm design, on-chip implementation and applications,” *International Journal of Circuit Theory and Applications*, vol. 33, no. 1, p. 17–51, 2005. 36

[67] Y. Shi, *Object based dynamic imaging with level set methods.* PhD, Boston Univ. College of Eng., 2005. 43

[68] G. J. Tornai and G. Cserey, “Initial condition for efficient mapping of level set algorithms on many-core architectures,” *EURASIP Journal on Advances in Signal Processing*, vol. 2014, no. 1, p. 30, 2014. 46, 49, 52, 53

[69] N. Paragios and R. Deriche, “Geodesic active regions: A new framework to deal with frame partition problems in computer vision,” *Journal of Visual Communication and Image Representation*, vol. 13, pp. 249–268, Mar. 2002. 49, 66

[70] H. Wu, V. Appia, and A. Yezzi, “Numerical conditioning problems and solutions for nonparametric i.i.d. statistical active contours,” *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 35, no. 6, p. 1298, 2013. 49

[71] B. Merriman, J. Bene, and S. Osher, “Diffusion generated motion by mean curvature,” in *Computational Crystal Growers Workshop* (J. Taylor, ed.), (Providence, RI), pp. 73–83, 1992. 66

DOI:10.15774/PPKE.ITK.2014.005

[72] B. Merriman, J. K. Bence, and S. J. Osher, "Motion of multiple junctions: A level set approach," *Journal of Computational Physics*, vol. 112, pp. 334–363, June 1994. 66

[73] "The insight toolkit." www.itk.org, 2012. 71, 72

[74] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pp. 235–246, 2010. 85, 86

[75] L. O. Chua, T. Roska, and P. L. Venetianer, "The CNN is universal as the turing machine," *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, vol. 40, no. 4, p. 289–291, 1993. 90