UNIVERSIDAD DE CÓRDOBA

PhD Programme:

COMPUTACIÓN AVANZADA, ENERGÍA Y PLASMAS

HETEROGENEOUS PARALLEL COMPUTING FOR IMAGE REGISTRATION AND LINEAR ALGEBRA APPLICATIONS

COMPUTACIÓN PARALELA HETEROGÉNEA EN REGISTRO DE IMÁGENES Y APLICACIONES DE ÁLGEBRA LINEAL

Directors:

DR. JOAQUÍN OLIVARES BUENO
DR. JUAN GÓMEZ-LUNA

Author

ORESTIS ZACHARIADIS

Thesis submission date: 2 JUNE 2020

REPORT OF THE SUPERVISORS

THESIS TITLE: Heterogeneous parallel computing for image registration and linear algebra applications

PhD CANDIDATE: Orestis Zachariadis

DETAILED REPORT OF THE THESIS SUPERVISORS:

This PhD thesis stands as proof of high-quality research and, thus, qualifies the PhD candidate, Mr. Orestis Zachariadis, for the title of Doctor.

The contributions of this thesis encompass novel GPU programming and optimization techniques that can benefit a wide range of applications. First, this thesis takes advantage of in-depth architecture details of GPUs to significantly improve the execution time of medical image registration. The work of the PhD candidate provides clinical validation of the proposed registration method on a scenario of liver deformation due to inflation of the stomach during laparoscopic surgery. Second, this thesis repurposes the Tensor Core Units (TCUs), novel processing elements intended to accelerate deep-learning, which equip current GPU architectures, to accelerate the performance of Sparse Matrix-Matrix Multiplication (SpMM). SpMM is an operation widely-used in linear algebra, graph processing, etc.

During the development of this thesis, the PhD candidate has benefited from the collaboration with prestigious international institutions through the H2020 HiperNav project. A 3-month secondment at SINTEF (Norway) facilitated the induction to industrial research.

This thesis resulted in the following research papers:

Accelerating B-spline interpolation on GPUs: Application to medical image registration. O. Zachariadis, A. Teatini, N. Satpute, J. Gómez-Luna, O. Mutlu, O. J. Elle and J. Olivares, Computer Methods and Programs in Biomedicine, vol. 193, p. 105431, Sep. 2020

Accelerating Sparse Matrix-Matrix Multiplication with GPU Tensor Cores. O. Zachariadis, N. Satpute, J. Gómez-Luna, Joaquín Olivares, Computer and Electrical Engineering, 2020. [Revision submitted].

The PhD candidate showed his strong commitment to open research by publishing all source code and datasets.

For the above reasons, we authorize the presentation of this PhD dissertation.

Córdoba, 2 June 2020.

The supervisors of the doctoral thesis:

[illegible] [illegible]

Joaquín Olivares Bueno Juan Gómez-Luna

To everyone who supported me.
This work would not be possible without them.

ABSTRACT

This doctoral thesis focuses on GPU acceleration of medical image registration and sparse general matrix-matrix multiplication (SpGEMM). The comprehensive work presented here aims to enable new possibilities in Image Guided Surgery (IGS). IGS provides the surgeon with advanced navigation tools during surgery. Image registration, which is a part of IGS, is computationally demanding, therefore GPU acceleration is greatly desirable. spGEMM, which is an essential part in many scientific and data analytics applications, e.g., graph applications, is also a useful tool in biomechanical modeling and sparse vessel network registration. We present this work in two parts.

The first part of this thesis describes the optimization of the most demanding part of non-rigid Free Form Deformation registration, i.e., B-spline interpolation. Our novel optimization technique minimizes the data movement between processing cores and memory and maximizes the utilization of the very fast register file. In addition, our approach re-formulates B-spline interpolation to fully utilize Fused Multiply Accumulation instructions for additional benefits in performance and accuracy. Our optimized B-spline interpolation provides significant speedup to image registration.

The second part describes the optimization of spGEMM. Hardware manufacturers, with the aim of increasing the performance of deep-learning, created specialized dense matrix multiplication units, called Tensor Core Units (TCUs). However, until now, no work takes advantage of TCUs for sparse matrix multiplication. With this work we provide the first TCU implementation of spGEMM and prove its benefits over conventional GPU spGEMM.

RESUMEN

Esta tesis doctoral se centra en la aceleración por GPU del registro de imágenes médicas y la multiplicación de matrices dispersas (SpGEMM). El exhaustivo trabajo presentado aquí tiene como objetivo permitir nuevas posibilidades en la cirugía guiada por imagen (IGS). IGS proporciona al cirujano herramientas de navegación avanzadas durante la cirugía. El registro de imágenes, parte de IGS computacionalmente exigente, por lo tanto, la aceleración en GPU es muy deseable. spGEMM, la cual es una parte esencial en muchas aplicaciones científicas y de análisis de datos, por ejemplo, aplicaciones de gráficos, también es una herramienta útil en el modelado biomecánico y el registro de redes de vasos dispersos. Presentamos este trabajo en dos partes.

La primera parte de esta tesis describe la optimización de la parte más exigente del registro de deformación de forma libre no rígida, es decir, la interpolación B-spline. Nuestra novedosa técnica de optimización minimiza el movimiento de datos entre los núcleos de procesamiento y la memoria y maximiza la utilización del archivo de registro rápido. Además, nuestro enfoque reformula la interpolación B-spline para utilizar completamente las instrucciones de multiplicación-acumulación fusionada (FMAC) para obtener beneficios adicionales en rendimiento y precisión. Nuestra interpolación B-spline optimizada proporciona una aceleración significativa en el registro de imágenes.

La segunda parte describe la optimización de spGEMM. Los fabricantes de hardware, con el objetivo de aumentar el rendimiento del aprendizaje profundo, crearon unidades especializadas de multiplicación de matrices densas, llamadas Tensor Core Units (TCU). Sin embargo, hasta ahora, no se ha encontrado ningún trabajo aprovecha las TCU para la multiplicación de matrices dispersas. Con este trabajo, proporcionamos la primera implementación TCU de spGEMM y demostramos sus beneficios sobre la spGEMM convencional operada sobre dispositivos GPU.

SCIENTIFIC PUBLICATIONS

The ideas, images and data that support this doctoral thesis have been published or are under review in the following works:

- Orestis Zachariadis, Andrea Teatini, Nitin Satpute, Juan Gómez-Luna, Onur Mutlu, Ole Jakob Elle, Joaquín Olivares *Accelerating B-spline Interpolation on GPUs: Application to Medical Image Registration*. Computer Methods and Programs in Biomedicine, 2020.
DOI: 10.1016/j.cmpb.2020.105431. Impact factor: 3.424

- [Revision submitted] Orestis Zachariadis, Nitin Satpute, Juan Gómez-Luna, Joaquín Olivares *Accelerating Sparse Matrix-Matrix Multiplication with GPU Tensor Cores*. Computer and Electrical Engineering, 2020.
Impact factor: 2.189

Research data and source code

- Ole Jakob Elle, Andrea Teatini, Orestis Zachariadis *Data for: Accelerating B-spline Interpolation on GPUs: Application to Medical Image Registration*. Mendeley Data, 2020, v1.
DOI: 10.17632/kj3xcd776k.1.

- Orestis Zachariadis *Source code for: Accelerating B-spline Interpolation on GPUs: Application to Medical Image Registration*. Github, 2019.
URL: `https://github.com/oresths/niftyreg_bsi`.

- Orestis Zachariadis *Source code: tSparse: A GPU algorithm for sparse matrix-matrix multiplication*. Github, 2020.
URL: `https://github.com/oresths/tSparse`.

Other works and collaborations

- Nitin Satpute, Rabia Naseem, Rafael Palomar, Orestis Zachariadis, Juan Gómez-Luna, Faouzi Alaya Cheikh, Joaquín Olivares *Fast Parallel Vessel Segmentation*.Computer Methods and Programs in Biomedicine, 2020.
DOI: 10.1016/j.cmpb.2020.105431. Impact factor: 3.424

- Orestis Zachariadis, Juan Gómez-Luna, Joaquín Olivares *Adaptive Threshold Acceleration on GPU: Application on Marker Detection*. Jornadas Andaluzas de Informática, Málaga (Spain), 2017.

Be kind whenever possible. It is always possible.

— Dalai Lama

GRATITUDE

In this section I would like to thank everyone that supported me during this work.

A special thank you to my parents and family for their unconditional support.

I would also like to thank my supervisors Joaquin and Juan who trusted me with this work. With their continuous support and encouragement they got me through my most difficult times.

Finally, I would like to thank all members of the lab, and especially Fran, Fernando and José Manuel who helped me with my adventure in a new country.

Thank You !!!

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

1D 1-Dimensional

2D	2-Dimensional
3D	3-Dimensional
AMG	Algebraic Multigrid
API	Application Programming Interface
AR	Augmented Reality
ASIC	Application Specific Integrated Circuit
AVX	Advanced Vector Extensions
BFS	Bread-First-Search
CC	Compute Capabilities (of CUDA)
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CR	Correlation Ratio
CT	Computed Tomography
CUDA	Compute Unified Device Architecture
ESC	Expand-Sort-Compress
FEM	Finite Element Method
FLOPS	Floating Point Operations per Second
FMA	Fused Multiply-Add
FFD	Free Form Deformation
GEMM	General Matrix Multiplication
GPGPU	General-Purpose Computing on Graphics Processing Units
GPU	Graphical Processing Unit
HCC	Hepatocellular Carcinoma
HPC	High Performance Computing
IGS	Image Guided Surgery
ILP	Instruction Level Parallelism
ITK	Insight Toolkit

LU	Lower-Upper (decomposition)
LUT	Look-Up-Table
MAC	Multiplication-Accumulation
MAE	Mean Absolute Error
MM	Matrix Multiplication
MP	MultiProcessor
MRI	Magnetic Resonance Imaging
ms	milliseconds
NCC	Normalized Cross Correlation
NMI	Normalized Mutual Information
nnz	number of non-zero
nz	non-zero
OpenCL	pen Computing Language
PET	Positron-emission Tomography
PSM	Patient Specific Model
SAE	Sum of Absolute Errors
SDK	Software Development Kit
SIMD	Single Instruction, Multiple Data
SM	Symmetric Multiprocessor, equivalent to MultiProcessor
SpGEMM	Sparse General Matrix Multiplication
SSE	Streaming SIMD Extensions
SSE	Sum of Squared Errors
SSIM	Structured Similarity Index
TCU	Tensor Core Unit
TH	Texture Hardware (implementation)
TLP	Thread Level Parallelism
TT	Thread per Tile (implementation)

TTLI	Thread per Tile with Linear Interpolations (implementation)
TV	Thread per Voxel (implementation)
US	Ultrasound
VCL	C++ vector class library
VT	Vector per Tile (implementation)
VV	Vector per Voxel (implementation)
w.r.t	with regards to

Part I

INTRODUCTION

1

INTRODUCTION

Contents

1.1 A need for fast medical image regisration

Registration applies a geometric transformation to an object to transform it in a way that maximizes the similarity with a different view of the same object or with a different object entirely.

Objects are entities in 3D and 2D images. We obtain 3D images from tomographic modalities such as: CT, MRI, PET etc. 2D images are obtained from projections such as X-rays or individual cuts such as mode-B ultrasound. In medical applications each image depicts an anatomical region. The two views can come from the same patient which constitutes a problem of intra-patient registration, or they can come from different patients (inter-patient registration).

The inputs of a registration algorithm are the two images that you want to register, and the output is a geometric transformation, which maps the points in one image to the corresponding points in the other. The determination of an appropriate similarity criterion is specific to the domain of the images, in our case the human anatomy. For the registration to be useful, the mapping it produces must be able to assist medical doctors in the diagnosis and clinical treatment.

Registration plays a fundamental role in computer-assisted surgery in which preoperative medical images and graphic models are registered with the patient's anatomy during surgery. Registration techniques demand a high amount of computation on large amounts of data, i.e., execution time can be quite long extending the surgery time and increasing the risk for complications for the patient.

Therefore, image requires heterogeneous parallel computing. Thanks to the use of massively parallel structures, not only the execution times are significantly decreased but also the resolution of the images can be increased.

1.1.1 *High performance soft tissue navigation for liver cancer intervention*

Primary liver cancer, typically caused by Hepatocellular Carcinoma (HCC), is the fifth most frequent cancer and the third most frequent reason of mortality by cancer [33]. In addition, the liver is a common target of metastases from other cancers, e.g., colorectal metastasis, with more than 100000 liver metastases in Europe [35]. The most common treatment for HCC is hepatic resection, which removes the tumor plus a safety margin from the liver, while saving as much of the healthy tissue as possible. However, not many patients are eligible for a resection. Therefore, it is vital to progress towards minimally-invasive methods that will increase the eligibility of patients and improve the survival prognosis [45].

HiperNav (High performance soft tissue navigation) project [45] aims to provide medical doctors with new tools for image-guided minimally invasive surgery. This thesis is encompassed within the HiperNav project and focuses on the high performance computing (HPC) for image registration. As the main HPC platform we use GPUs.

1.2 Sparse Matrix-Matrix Multiplication

In Image Guide Surgery we usually have a Patient Specific Model (PSM) which is created by CT, MR and other image modalities captured before the surgery. During the surgery the position of the patient on the operation table can be different than the one used when capturing the images. Furthermore, cutting, contact, inflation of the abdominal area during minimally invasive procedures like laparascopic surgery and ablation can deform the liver significantly. For this reason, we need to fuse the previously created PSM with live data that we capture during the surgery.

VESSEL REGISTRATION CT and MR scanning requires very specific conditions and facilities and usually is impractical during a surgery, unlike US which only requires a probe. Using the doppler mode of US, we can capture the shape of the vessels and subsequently use registration to find the location of the US in the PSM. However, vessels occupy only a small number of the image pixels, and therefore we have to deal with sparse images (i.e., images in which only a very small percentage of pixels contains valuable information). To that end, we are working on Sparse General Matrix Multiplication (SpGEMM). SpGEMM can facilitate the convolution stage of CNNs with sparse vessel images as inputs or graph matching operations.

1.3 General-Purpose GPU computing

Graphics Processing Units (GPUs) typically process computer graphics. The use of a GPU for generic computational workloads is called General-Purpose GPU computing. CUDA [80] is a parallel computing framework and an API that provides easy access to the general purpose graphics processing unit (GPGPU) functionality of NVIDIA GPUs. In this section, we give a short introduction to the concepts of CUDA architecture that we use in this work.

1.3.1 *Processing hierarchy*

A CUDA GPU is a multicore system, with a few tens of processing cores, which are called *MultiProcessors (MPs)* or Symmetric Multiprocessors (SMs). GPUs before with Compute Capabilities (CC) less than 7.x include cores for fp32 arithmetic, whereas the recent Volta and Turing architectures additionally include cores for integer arithmetic and tensor cores for matrix operations [82]. Figure 1.1 illustrates a part of the architecture of the SM on Turing GPUs. We can see the distribution of the processing cores.

Each SM deploys multiple *threads* which operate on data, similar to CPU threads. These threads are organized in groups of 32 threads, called *warps*. All threads that belong to the same warp execute the same instruction in lock-

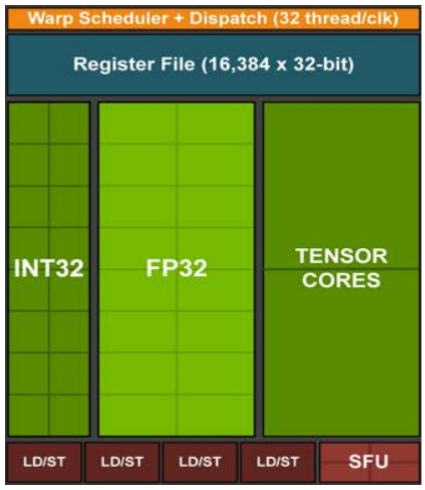

Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
TENSOR CORES
LD/ST
LD/ST
LD/ST
LD/ST
SFU

Figure 1.1
A part of the SM of Turing architecture that shows the distribution of int32, fp32 and tensor cores, as depicted in Turing whitepaper [82].

step. There are three characteristics that make GPU threads different from CPU threads: a) They work in lower frequencies, b) they are many more in number (> 1000), and c) they follow the SIMD (Single instruction, multiple data) model of the Flynn taxonomy [80].

1.3.2 *Memory hierarchy*

CUDA GPUs deploy a multi-level memory hierarchy (Figure 1.2). Input and output data stay to the larger but slower off-chip memory, whereas frequently used data remain to the faster but smaller on-chip memory. Threads can access data from off-chip memory using three different memory spaces, optimized for different types of access. First, *global memory* stores the input and output data. Second, *texture memory* is optimized for texture processing. Three characteristics of texture memory are: a) it is read-only, b) it is optimized for spatial accesses, and c) dedicated hardware units for linear interpolation use the texture memory. Third, *constant memory* is a read-only memory which we use to store constants [80].

CACHES A CUDA GPU also has small, fast on-chip caches to reduce the cost of repeating data movements from off-chip memory. Four notable caches are: a) *shared memory*, a software-managed cache, b) *L1 cache*, a hardware-managed cache for global memory, c) *texture cache* for texture memory and d) *constant cache* for constant memory [80].

REGISTERS Each thread has access to its own *registers*, which are faster than shared memory and other caches. For any instruction (arithmetic, logical, etc) the input and output data *must* be in registers [80].

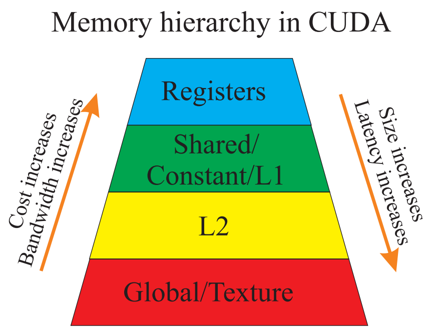


Figure 1.2
Memory hierarchy in CUDA. As we go up the pyramid, memory becomes more expensive and the bandwidth increases. As we go down the pyramid, memory becomes bigger and latency increases.

1.3.3 *Programming model*

CUDA groups threads in *blocks* and each CUDA program can have many blocks of threads. Blocks can contain up to 1024 threads. The GPU schedules blocks to SMs, where the blocks stay until completion [80]. Constant and global memory are common for all blocks, whereas shared memory / L1 cache is private to each block [80].

1.4 Thesis overview

This dissertation is organized in five chapters. The first chapter introduces the necessity for fast medical image registration and the utility of sparse matrix multiplication in medical computing. The second and third chapters present the two main parts of this dissertation. The first part focuses on our work on GPU acceleration of B-spline interpolation in the context of medical image registration. The second part focuses on our work on GPU acceleration of spGEMM. Both parts are presented in a similar way. We start by presenting the motivation and objectives. We continue with background and related work. Then we demonstrate our methods, followed by comprehensive results sections. Finally, we conclude each part and give some insights for future work. The last two chapters discuss our work and outline the conclusions and contributions of this work.

Part II

B-SPLINE INTERPOLATION FOR IMAGE REGISTRATION

2 ACCELERATING B-SPLINE INTERPOLATION AND IMAGE REGISTRATION

Contents

2.1 Motivation

Image Guided Surgery (IGS) aims to provide surgeons with navigation capabilities for better, safer procedures and improved surgical outcome through better visualization [10]. Surgical navigation can guide surgeons through the use of volumetric images, such as Computed Tomography (CT), Magnetic Resonance Imaging (MRI) or UltraSonography (US) [12], and instrument tracking technologies [108]. As an example, during minimally invasive surgery, the laparoscope camera video feedback can be augmented with reprojected CT or MRI scans, a process known as Augmented Reality (AR) [68, 94]. However, the accuracy of image guided surgery is often undermined by movements of the organs, for example, manipulations performed by the surgeon such as resection (cutting) of ligaments during mobilization. Hence, due to the non-linear behaviour of soft tissue deformation, rigid or affine transformations are not sufficient to correctly reproduce the movements of the organs. Therefore, non-rigid registration [106] is a more accurate way to model more complex deformations and is necessary to model soft tissue deformations.

Non-rigid registration through Free Form Deformation (FFD) [96], based on cubic B-spline interpolation, can provide a suitable solution for non-rigid image registration tasks. By manipulating a grid of control points, the shape of the underlying 3D object (e.g., an organ captured by the laparoscope camera) can be changed using a smooth and C^2 continuous (i.e., continuous up to second order derivatives) transform. One important property of B-splines is that they deploy local support (i.e., each control point affects only its neighborhood), and therefore, the workload can be balanced among the execution threads running on multi-core processors.

Cubic B-spline interpolation or, simply, B-spline interpolation is a form of interpolation that is more accurate than the conventional linear interpolation, as B-spline interpolation better approaches the ideal sinc function interpolation [98, 113]. Many image processing (e.g., non-rigid registration) and visualization tasks [98] benefit from the additional accuracy. In this work we focus on the 3D version of B-spline interpolation, also called tricubic interpolation. B-spline interpolation is useful for handling 3D medical images and consequently IGS.

Registering pre-operative (before the surgery) models to intra-operative (during the surgery) reconstructed surfaces or to US images during IGS is particularly demanding. Graphics Processing Units (GPUs) can help achieve real-time requirements of IGS, as they offer massive computation performance in comparison to CPUs. A GPU deploys thousands of execution threads, which operate on large batches of data, and provide high throughput. GPU's multi-threaded architecture makes it much more power-efficient than a CPU on multithreaded workloads [80]. As a result, GPUs can improve the performance of B-spline interpolation significantly [98, 104].

2.1.1 *B-spline interpolation and previous work*

The intensive *data movement* of a large number of input samples between the memory and the GPU cores is the main bottleneck of cubic B-spline interpolation implementations on a GPU [104]. Sigg et al. in [104] and Ruijters et al. in [98] achieve a substantial reduction in the number of input samples by representing the weighted sums as trilinear interpolation and utilizing the interpolation unit of the GPU texture unit. Later, Ellingwood et al. in [28] and Du et al. in [26] use GPU implementations of B-spline interpolation to improve the performance of registration. They improve input sample loading by aligning the control grid with the voxel grid of the volume [26, 28, 100]. However, even including the most recent improvements, the performance of B-spline interpolation is still limited by memory bandwidth, as indicated by our preliminary profiling results.

2.1.2 *Pneumoperitoneum compensation*

This works focuses on image guide liver surgery. As a test case for image registration we use pneumoperitoneum, i.e., the inflation of the patient's abdomen during laparoscopic surgery. Pneumoperitoneum, along with the new position of the patient on the surgical table, deforms the shape of the liver [44, 54]. Consequently, on the day of the surgery, the liver has different shape than that of the pre-operative CT or MRI scans. The deformed shape causes great inaccuracy within IGS systems for laparoscopic liver resection surgery, because the plan of the surgery is based on the initial (pre-operative) CT or MRI scan. In order to correct the inaccuracies, we have to capture new images during the surgery and to match them to pre-operative scans with the help of non-rigid registration [89]. The drawback is that registration is computationally demanding, therefore it would benefit from an improved GPU approach. In Figure 2.14 we will show the role of image registration in matching the images before and after pneumonoperitoneum. We will also show the main components of registration. *Deformation*, which is the target of our optimizations, is highlighted.

2.2 Objectives

The primary objective of this work is to accelerate medical image registration using General-Purpose computing on Graphics Processing Units (GPGPU). To that end, we form the following secondary objectives:

- Optimize one of the most computationally demanding parts of FFD non-rigid registration, i.e., B-spline interpolation.
- Test the performance of B-spline registration in regards to execution time and accuracy in a realistic medical scenario. In order to achieve this, we need to:
  - Create a dataset of CT/MR images to test registration.
  - Integrate our B-spline interpolation methodology in an existing registration library, NiftyReg [74]. NiftyReg is used as reference in recent works [89].

2.3 Background

In this section, we first give an overview of medical image registration and deformation fields. Second, we introduce B-spline interpolation and the properties that play an essential role in our proposed optimizations.

2.3.1 *Medical image registration*

The task of medical image registration is to find the correspondences between two medical images and bring them to the same coordinate system. Two common medical scenarios that we need registration for are: a) to find the correspondences between images that are used during planning of the surgery (i.e., diagnostic images) and images acquired during the surgery, and b) to find correspondences between images of different modalities that we acquire during the surgery (e.g., among US and stereoscopic camera feedback [68], [107]). Generally, image registration finds application in planning, navigation, data fusion, visualization, segmentation based on an atlas, automatic tissue recognition etc [102, 103].

REGISTRATION INPUTS Image registration requires a pair of images as input, the *reference* (also called *static*) image and the *target* (also called *floating* or *moving*) image. The target image needs to be transformed to match the shape and structures of the reference image.

PNEUMONOPERITONEUM COMPENSATION FOR LIVER REGISTRATION Our goal in this work is to use registration to match the shape of the liver between scans that are acquired before and after pneumonoperitoneum, as in Figure 2.1. In this work, the targeted application is non-rigid registration, due to the viscoelastic nature of the liver [89].

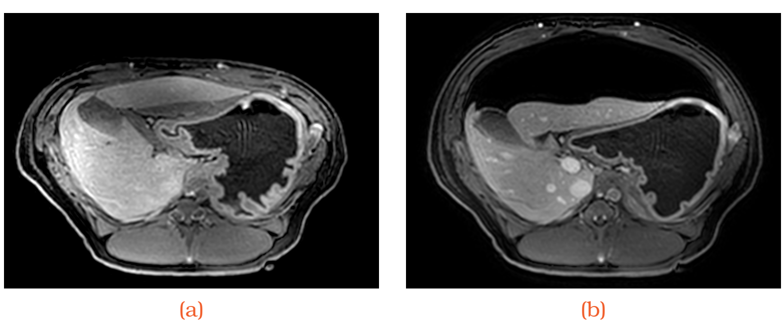


Figure 2.1
MRI scans, (Intra-modality) of porcine liver without (left) and with (right) pneumoperitoneum in preparation for minimally invasive surgery

Deformation fields

Usually, we represent non-rigid transformation as a *deformation* or *displacement* vector field, which connects the corresponding points of the reference and the floating image space.

Figure 2.2 shows an example of a displacement field created after executing registration on the pig liver images described in Section 2.8.2. Figure 2.2a illustrates the floating image, whereas Figure 2.2b illustrates the displacement field overlaid over the reference image. The arrows connect the voxels of the reference image with the correspondent voxels of the floating image. Typically, medical images are 3D. Therefore, the deformation field consists of 3D vectors in the 3D space.

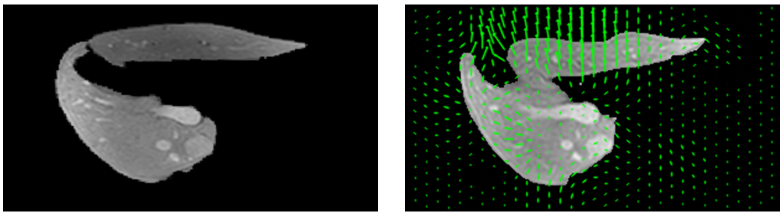

(a) (b)

Figure 2.2
(a) Liver of a pig before registration, (b) The generated deformation field overlaid on top of the reference image

2.3.2 *B-spline interpolation*

In non-rigid registration, the interpolation of a deformation field describes the transformation. We prefer B-spline interpolation because it requires data only in a small neighborhood around the point of interest, thus reducing the computational complexity for large medical images [102]. B-spline interpolation is one of the most time consuming steps of the registration [74]. Therefore, by optimizing B-spline interpolation, we increase the total performance of image registration. Although the application in this work is image registration, B-spline interpolation can also be used for general image processing, like zooming and geometrical image transformation [48, 113]. In this section we provide the theoretical background of B-spline interpolation and we define the tiles, groups of elements with common properties.

B-spline interpolation theory

It is easier to understand B-spline *interpolation* by comparison to linear interpolation. With conventional linear interpolation we have two known points and we want to estimate the *interpolant*, a third point between the other two. The value of the interpolant is the result of a first order function of the two known points (i.e., two arguments). Similarly, in B-spline interpolation, we have a number of known points and we would like to estimate the interpolant. B-spline based interpolation requires 4^N (where N is the image dimension) neighboring known points as function arguments (owing to the limited support of B-splines). In comparison to linear interpolation, the function of the four known points is of the third order (cubic). This function is called the B-spline basis function and it is where the name B-spline interpolation comes from [98]. The known points are called *control points*, and they are arranged in a *control point grid* over the entire image.

In this work we work with 3D images in the context of medical imaging, therefore we focus on the theoretical background required for 3D images. We denote the elements of a $n_x \times n_y \times n_z$ control point grid, with uniform spacing $\delta_x \times \delta_y \times \delta_z$, as $\phi_{i,j,k}$. The domain of the image volume is in the x, y, z coordinate space. We denote the function that calculates the interpolants as $T(x, y, z)$ [96]:

$$T(x,y,z) = \sum_{l=0}^{3} \sum_{m=0}^{3} \sum_{n=0}^{3} B_l(u) B_m(v) B_n(w) \phi_{i+l,j+m,k+n} \tag{2.1}$$

where $i = \lfloor x/\delta_x \rfloor - 1, j = \lfloor y/\delta_y \rfloor - 1, k = \lfloor z/\delta_z \rfloor - 1, u = x/\delta_x - \lfloor x/\delta_x \rfloor, v = y/\delta_y - \lfloor y/\delta_y \rfloor, w = z/\delta_z - \lfloor z/\delta_z \rfloor$. B are the B-spline basis function derived weights and ϕ are the control points.

Tiles

Tiles are logical groups of voxels (voxel is the analogue of pixel in 3D space) that share common properties. Let us denote as $\delta_\alpha \in \mathbb{Z}$ the spacing in voxels, where a is one of x, y, z (for each of the three dimensions). Then, from Equation (2.1), we make two observations. First, $\lfloor \alpha/\delta_\alpha \rfloor - 1$ increases by one every δ_α elements in the direction of α. Second, $\alpha/\delta_\alpha - \lfloor \alpha/\delta_\alpha \rfloor$ is periodic with period δ_α. Based on these two observations, we form tiles of $\delta_x \times \delta_y \times \delta_z$ dimensions and we derive the following two important properties:

1. The *same* set of control points affects all voxels inside the tile. In the case of B-splines, there are four neighboring control points in each direction that affect the voxels inside the tile.

2. B-spline weights depend only on the relative distance from the coordinate origin of the tile. We emphasize that when the voxel and control point grids are aligned, there is always the same number of equally spaced voxels between two control points. As a result, when the grids are aligned, all tiles in this voxel grid use the *same* set of B-spline weights.

From Equation (2.1), it follows that $4 \times 4 \times 4$ control points, forming a cube (Figure 2.3), affect each voxel (and consequently tile). In general, in N-dimensional images, 4^N control points affect each voxel. Figure 2.4 highlights the tile properties of a 2D example. We present the 2-dimensional case because it is easier to understand. In this figure, the gray-colored area highlights the pixels that form a tile. The filled black circles delineate the 4×4 control points that affect this tile.

We utilize tiles and their properties in our proposed optimization scheme to significantly reduce memory traffic between off- and on-chip memory.

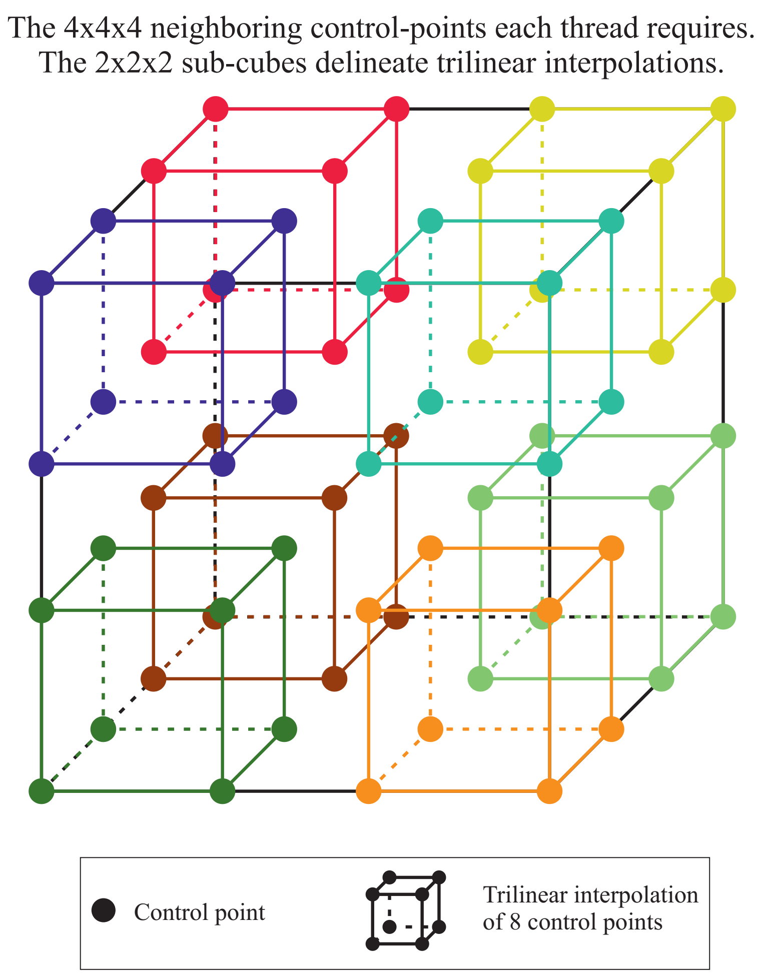

The 4x4x4 neighboring control-points each thread requires.
The 2x2x2 sub-cubes delineate trilinear interpolations.
Control point
Trilinear interpolation of 8 control points

Figure 2.3
Cubes depicting the grouping in trilinear interpolations for a 3D control point grid

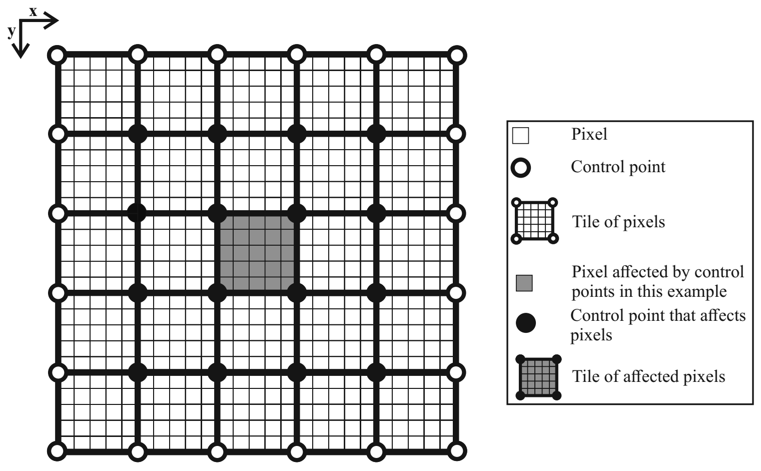

x
y
Pixel
Control point
Tile of pixels
Pixel affected by control points in this example
Control point that affects pixels
Tile of affected pixels

Figure 2.4
Tile properties of a 2D image

2.4 Related Work

In this section we provide an overview of GPU-based medical image registration.

2.4.1 *Medical image registration on GPU*

In the following paragraphs we give a short review of medical image registration on GPU.

Basic components of registration

The registration procedure can be divided into four basic components. These components are: 1) transformation, 2) interpolation, 3) measurement, and 4) optimization (Figure 2.5).

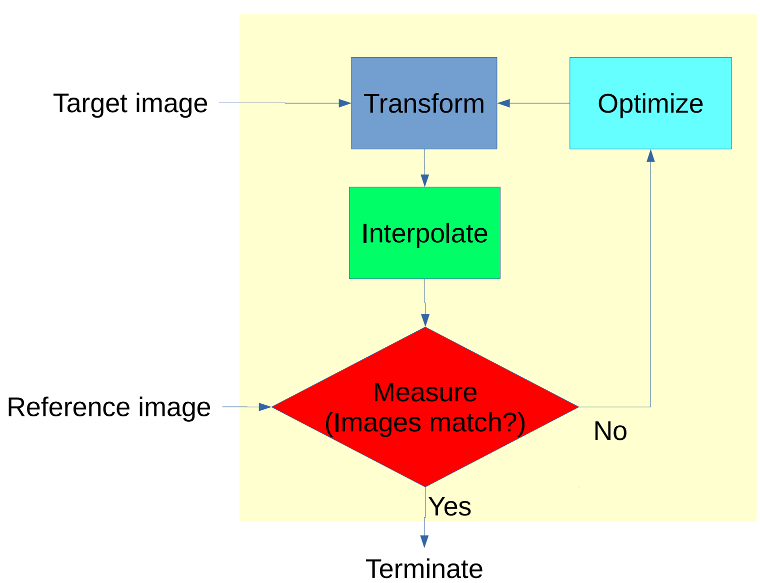


Figure 2.5
The building blocks of registration.

Transformations

The transformation component produces a transformation function that converts the target image based on parameters that the optimization component sets. Transformations can be classified as linear or non-linear (deformable). For linear transformations we use rigid and affine registration, whereas for non-linear transformations we use non-rigid registration [102]. Traditionally, we first apply rigid and affine transformations to capture the global deformations and, then, non-rigid transformation to capture the more fine-grained local transformations.

RIGID TRANSFORMATION Rigid transformation rotates and translates the target to best match the reference. The target object maintains its size and shape. These methods apply rotation and translation matrices to all pixels (or block of pixels) in a way that maximizes a similarity criterion [30, 90, 101].

AFFINE TRANSFORMATION Affine transformation extends rigid transformation by adding additional transformations, like scaling, shearing etc. Points and lines from the target match points and lines of the reference, respectively, and parallel lines remain parallel. Similarly to rigid, these methods apply small transformation matrices to the images [65, 74, 86].

DEFORMABLE TRANSFORMATION Rigid and affine transformations are fast and reliable for rigid or relatively rigid tissue (e.g. brain which has limited movement in the cranium, bones etc). Nevertheless, they do not adapt well to elastic deformations of more fluid tissues, e.g., breast [96, 102] or liver [45, 122].

A popular category of deformable registration is Demons, which is based on optical flow [103, 112] and is accelerated by CUDA [50, 75]. Another popular category is the one that is based on the parameterization of the displacement field by a number of control points [74, 96, 122]. Parametric methods often achieve better results than other methods when registering images of multiple modalities [103].

Interpolation

After the transformation component defines the desired transformation of the pixels, an interpolation (or resampling) stage is necessary to determine the actual pixel intensities of the to-be-transformed image. Nearest neighbor, linear interpolation, B-spline interpolation, quadratic interpolation, Gaussian interpolation are possible interpolation methods [63]. Nearest neighbor is the fastest but gives the greatest error. The most commonly used is linear interpolation, with a good balance between computation time and error. B-spline interpolation has excellent performance in keeping the characteristics of the original image with a slight decrease in performance [63, 113].

Measurement

After transformation and resampling we have to measure the similarity of the transformed image to the reference image. Some of the common similarity measures include Sum of Absolute Errors (SAE), Normalized Cross Correlation (NCC), Correlation Ratio (CR) [95] and Normalized Mutual Information (NMI) [101]. Although SAE and NCC are relatively trivial to implement on GPU [38], this is not the case for NMI and CR.

NORMALIZED MUTUAL INFORMATION Normalized mutual information is one of the most popular methods for registration with multiple modalities [91] and a performance critical component of the registration workflow [27]. NMI requires the computation of a joint image histogram. Histograms are difficult to implement on GPUs because the access to the bins requires synchronization. Shams et al. avoid synchronization and atomic operations using sorting [101], whereas Gómez-Luna et al. employ many subhistograms to minimize conflicts [36].

CORRELATION RATIO Correlation ratio offers comparable results to NMI but, unlike NMI, needs only a 1D histogram, which makes CR more suitable for GPU implementations [64].

Optimization

If the measurement component indicates that the matching between target and reference images is not sufficient according to the selected convergence criterion, we call the optimization component. Optimization searches the transformation parameter space to find a transformation that will improve the previous result. According to [102] the optimization can be categorized to gradient-based and gradient-free.

GRADIENT-BASED AND GRADIENT-FREE Gradient-based methods calculate the partial derivatives of a cost function in order to find the minimum of the cost function. A typical gradient-based method is gradient descent [93]. An example of gradient-free GPU approach is [43], where they use a symmetric explicit search at each pixel to find the neighbor that maximizes a point-wise mutual information metric. Typically, gradient-based methods converge in fewer iterations [102].

PARALLELIZATION The optimization component itself usually is inherently serial (e.g., a partial derivative that has to "wait" for the previous partial derivative). Moreover, in comparison to other steps of registration, optimization does not affect the execution time a lot, and therefore parallelizing the optimization is usually not practical [102, 103]. What is important with regards to improving the execution time of registration is the number of iterations until convergence.

2.4.2 *B-spline interpolation in medical image registration*

Since one of the most suitable applications of B-spline interpolation is non-rigid registration for medical images, several attempts happened to accelerate B-spline interpolation for this context. Modat et al. [74], in NiftyReg library, optimize for GPU the FFD registration that Rueckert et al. [96] present. They observe that the most computationally demanding step is B-spline interpolation, which is used for the calculation of the deformation field.

Alignment of the control point grid with the voxel grid of the volumetric scan significantly improves the computational efficiency of B-spline interpolation. By aligning the grids, voxels are organized in tiles, which present beneficial properties that reduce redundant operations. To our knowledge, Shackleford et al. in [100], in their work on B-spline registration methods, are the first that use tile properties to reduce redundant operations and memory transfers.

Later, tile properties were also used by Ellingwood et al. in [28] and by Du et al. in [26] to improve the performance of their non-rigid registration algorithm.

PREFILTERING A drawback of acquiring a reconstructed function with cubic B-spline interpolation is that the function may not pass through the sample points, and thus the output may be more smoothed than necessary. One way to avoid over-smoothing is to add a prefiltering step before the B-spline interpolation step. For this reason, Ruijters et al. in [97], additionally to B-spline interpolation, they implemented on GPU the prefiltering step presented by Thévenaz et al. in [111]. Shortly after, Champagnat et al. [15] replaced the IIR filter filter of [97] with a FIR filter to improve the GPU performance of smaller image outputs. Ruijters et al. provide online as a library their GPU implementation of the prefiltering step along with cubic B-spline interpolation. This library is still actively used in recent literature [5, 14] etc. We will not use prefiltering in our implementations because it is not used in the calculation of the deformation field. Nevertheless, our approach can use the same prefiltering step from the literature. However, our approach is a suitable replacement for the interpolation step of the approaches of the previous paragraph.

In the following two sections we introduce two state-of-the-art GPU B-spline interpolation methodologies and their respective implementations.

Texture Hardware (TH)

The work by Sigg et al. [104] is one of the first attempts on acceleration of B-spline interpolation using interpolation hardware of the GPU. Their approach utilizes the texture unit of the GPU, a special unit usually used for computer graphics, that is accessed through easy to use Application Programming Interfaces (APIs). They use the texture hardware unit to reduce the number of memory transfers by loading only the result of the trilinear interpolation, instead of loading the eight control points that the trilinear interpolation needs. Later, Ruijters et al. [98] implemented this method in CUDA. This method reduces memory transfers from off-chip memory significantly [98, 104].

If we define the dimensionality of the input image as N, each voxel is affected by the 4^N control points surrounding it. However, if we use the texture unit to calculate trilinear interpolation, only 2^N loads are required [104]. This is particularly useful for higher dimensional inputs, such as 3D medical images, where the required number of memory transfers from global memory is reduced from 64 loads to 8 loads of trilinear interpolation results. Figure 2.3 illustrates the control points of 3D input. Without using the trilinear interpolation hardware, we have to load all $4 \times 4 \times 4$ control points in order to calculate each voxel. With trilinear interpolation hardware, we have to load only the results of trilinear interpolation of $2 \times 2 \times 2$ sub-cubes. Each such sub-cube has a different color in the figure.

Hardware interpolation is fast but it brings two problems. First, its accuracy is very low, with only eight bits holding the interpolation result [80]. Second, although the number of coefficient loads are drastically reduced, these loads are dependent on the absolute position of each voxel. Therefore, there is no way to cache them for reuse by nearby voxels because the fetched values are unique. Texture Hardware B-spline interpolation is included in an easy-to-use library by Ruijters et al. [97] and is used by recent works [14].

By doing the trilinear interpolation in software, instead of hardware, we can create different thread assignment schemes that can increase the overlap between neighboring threads significantly, achieving even fewer memory transfers than TH. Moreover, we can take advantage of FMA instructions to decrease computational complexity and avoid the inaccuracies caused by the inherently low precision of trilinear interpolation of the texture unit of the GPU [80].

Thread per Voxel (TV)

The basic mechanism of this approach is that each CUDA thread is assigned to a single element in a straightforward way, e.g., a thread for each voxel in the case of 3D images. Specifically, for each voxel position of the reference image one thread calculates the deformation vector using Equation (2.1). NiftyReg [74] uses this straightforward parallelization. This method can be further enhanced by utilizing the tile properties of Section 2.3.2. In this case, one or more blocks are assigned to each tile, with one thread for each voxel of the tile, as Ellingwood et al. [28] do. Since all voxels inside a tile need the same control points, these control points are loaded from global memory only once and are stored in shared memory for faster access.

The thread assignment scheme of this approach is more straightforward, and therefore easier to implement. However, for maximum performance, the tile size should be a multiple of the warp size (32 threads for CUDA). Otherwise, the rest of threads of each warp remain inactive due to hardware limitation. Essentially, this means that the users of this approach are not free to select the tile size that best fits their needs, but they have to choose between a small number of presets.

NIFTYREG LIBRARY NiftyReg [74] is a lightweight open-source medical image registration library, which contains optimized implementations of B-spline interpolation, both for CPUs and GPUs. It is open-source and well-maintained, with competitive performance against other state-of-the-art implementations [62]. For this reason, it is commonly used as the reference state-of-the-art non-rigid registration (e.g., [89]). The GPU implementation uses the simple, straightforward thread-per-voxel scheme, i.e., no tile properties, LUTs, etc. The CPU implementation uses multi-core and vectorization optimizations and does minor utilization of tile properties. We started this work by profiling NiftyReg to find the impact of B-spline interpolation on the entire registration process. The profiling results show that B-spline interpolation takes 30% of the total execution time. The main bottleneck is the memory latency due to saturation of cache bandwidth when loading the control points. Therefore, our initial goal is to minimize the memory transfers from global memory to on-chip caches.

2.5 Optimizing B-spline interpolation

This section describes our GPU implementation of B-spline interpolation and explains the logic behind assigning one GPU thread per tile. We present the most important GPU optimizations in detail, along with the computation complexity analysis that leads to our fastest GPU method. Furthermore, we apply our methodology to CPU to show that it is applicable to other platforms.

Finally, we define the number of required memory transfers for the various B-spline interpolation implementations.

Within this study, the implementation of GPU parallelization is specific to 3D medical images (CT, MRI or US volumes). Therefore, analysis and implementations focus on the 3D case, unless otherwise specified. To facilitate understanding of this section, we emphasize that, in the case of 3D images, each voxel is affected by the control points in its local $4 \times 4 \times 4$ cubic neighborhood. Consequently, all voxels that belong to a tile are affected by the same $4 \times 4 \times 4$ neighborhood.

2.5.1 *Overview*

The key optimizations of our GPU implementation of B-spline interpolation are two. First, an entire tile of voxels is assigned to a single GPU thread, in contrast to the one-thread, one-voxel paradigm. Figure 2.6b compares the thread assignment between TV and the proposed approach. With this assignment, we minimize: a) the reads from off-chip memory by maximizing the overlap of the input control points and, b) the cache accesses by keeping the input points in registers and reusing them by many voxels. Although registers are very fast, which significantly benefits the performance of our implementation, 3D medical images require a large amount of them, which limits the number of active GPU threads. Nevertheless, the reduced number of required memory transfers enables us to hide their latencies efficiently by overlapping them with independent arithmetic operations, a method commonly known as Instruction Level Parallelism (ILP). Second, we replace the weighted sum of the basic formula of B-spline interpolation with independent trilinear interpolations. We calculate these trilinear interpolations with FMA instruction of the GPU. FMA increases both accuracy and speed in regards to typical multiplication and addition.

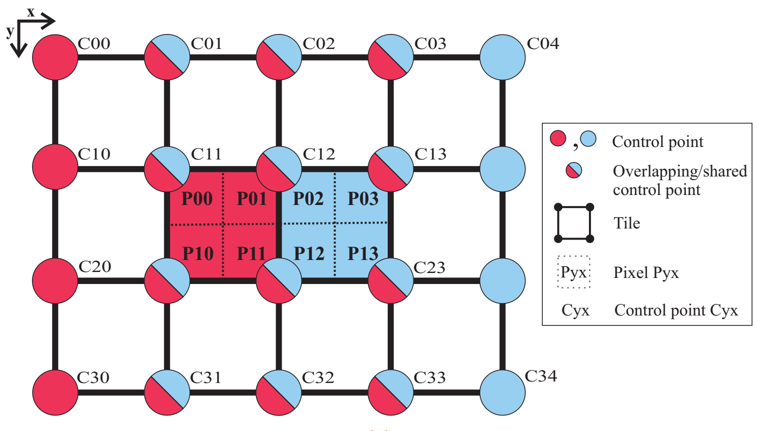
2.5.2 *Thread per Tile (TT)*

In the following paragraphs we describe the optimizations utilized in the proposed implementations. We show how our input loading and register optimizations reduce memory accesses. To better display how are work improves over the state-of-the-art, we describe our optimizations by comparing to the methods of TV.

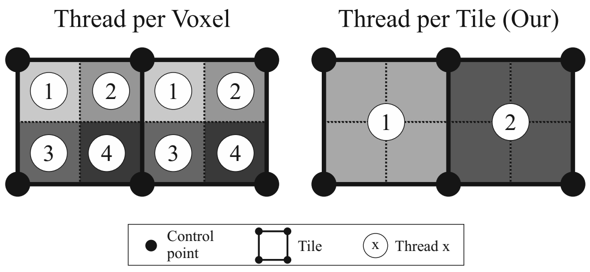
Input loading optimization

The main idea of this optimization is to reduce loads from global memory by taking advantage of the overlap of neighboring tiles.

In a TV implementation, each block of threads works on a unique tile of voxels. Each block stores the required 4^N control points (Section 2.3.2) in a unique shared memory area. Therefore, for each tile we need to move 4^N control points from global memory to shared memory. Step 1, of the left part of Figure 2.7, illustrates the required transfers from global memory to shared memory for a 2D example. In this example, we have two tiles and each tile is assigned to



(a)



(b)

Figure 2.6
a) An example 2D grid with two tiles, b) Thread assignment for Thread per Voxel (left) and Thread per Tile (right) approaches. The different threads are represented by different shades of gray, which correspond to the numbers

one block. The figure shows that the amount of transfers from global memory to shared memory, that the two tiles require, is $4 \times 4 + 4 \times 4$ control points each.

We want to reduce the required amount of transfers from global memory to shared memory. Two important observations derive from Equation (2.1). First, only the four neighboring control points in each direction of a Cartesian coordinate system affect each tile. Second, tiles that are consecutive in a direction require control points that are consecutive in the same direction. Therefore, there is overlap among neighboring tiles. Figure 2.6a illustrates two tiles that are consecutive in the x-direction. The first tile has its pixels and necessary control points highlighted with red color and the second with blue. The first tile requires the first 4×4 area of pixels (the square defined by vertices C00 and C33), whereas the second requires a 4×4 area that is shifted by one in the x-direction (the square defined by vertices C01 and C34). In total, the two tiles require $4 \times (4 + 1)$ pixels. Step 1, of the right part of Figure 2.7, illustrates the reduction in transfers to shared memory, with overlap in the x-direction. Two tiles require only 4×5 control points. In a real case scenario, with overlap in all directions, the benefits are more pronounced.

In the general 3D case, the cubic $4 \times 4 \times 4$ control point neighborhood of each of the consecutive tiles overlaps with a stride of one for each direction. In the general 3D case, a group of tiles of size $l \times m \times n$ with overlap in the x, y, z -directions, needs $(4 + l - 1) \times (4 + m - 1) \times (4 + n - 1)$ control points in total.

We observe that there is significant overlap of control points for neighboring tiles. Our thread per tile assignment scheme takes full advantage of the overlap to reduce the ratio of data movements per voxel. Our approach requires fewer transfers from global memory than both TH and TV methods, as we detail in Section 2.5.6.

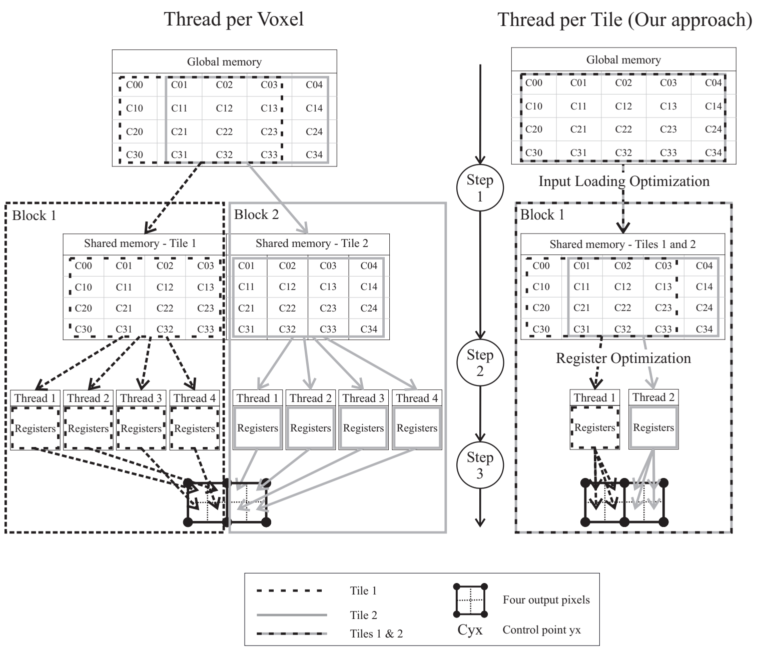


Figure 2.7
Comparison of input loading and register optimization for Thread per Voxel (left) and Thread per Tile (right) for the two neighboring tiles of Figure 2.6

Register optimization

The two main ideas of this optimization are: a) to load the control points for all voxels of the tile from shared memory only once, and b) to keep the loaded control points to registers, which are the fastest on-chip memory, until the thread exits.

In a TV implementation, threads belonging to the same block work on individual voxels of the same tile. For every voxel belonging to the tile, the corresponding threads need to access the exactly same area of shared memory as the other threads of the block, in order to load the same set of control points. Step 2, of the left part of Figure 2.7, illustrates the required transfers from shared memory to registers for a 2D example. In this example, each tile comprises four pixels and each pixel is assigned to one thread. The figure shows that four pixels require four transfers (from shared memory to registers) of sixteen control points each.

Shared memory is faster than global memory, but before the GPU can execute any instruction on the control points, each thread has to move the entire set of control points to its registers first (Section 1.3). Thus, we want to minimize transfers from shared memory to registers. We make two improvements. First, we assign one thread for the entire tile. For every voxel belonging to the tile, the corresponding thread needs to access the same shared memory area exactly once, in order to load the control points. Second, we utilize register tiling. The thread keeps the control points to (the faster than shared memory) registers [115] until it processes every voxel of the tile. Step 2, of the right part of Figure 2.7, illustrates a drastic reduction in memory transfers. A tile of four pixels requires only one transfer of sixteen control points.

Other optimizations

In this section we present other optimizations, that are not part of our key optimizations.

LOOK-UP-TABLES (LUTS) LUTs are tables that hold pre-calculated values, so that we do not have to calculate them during execution of the program, saving computational resources for other tasks. Based on the second property of the tiles (Section 2.3.2), the B-spline basis functions weights remain the same among tiles when having uniform (aligned) grids. Therefore, they can be stored in LUTs in constant memory.

SKIPPING SHARED MEMORY According to Perrot et al. [88], at least for the case that the input is overlapping, it may be beneficial to load to registers directly, instead of first loading to shared memory and then from shared memory to registers. The control points needed by each group of tiles can be stored to registers directly (using instead only the hardware-managed caches). With this approach, we avoid pointer arithmetics and synchronization that the management of shared memory requires.

2.5.3 *Thread per Tile with Linear Interpolations (TTLI)*

We extend TT by reformulating the triple sum of Equation (2.1) to *trilinear* interpolations. The basic idea is that a linear interpolation can replace an addition of two weighted summands. We can extend this to three dimensions, where we combine eight summands to a trilinear interpolation [104].

We calculate trilinear interpolation as a combination of seven linear interpolations (we do not use the hardware interpolation unit). Linear interpolations have the form $a + w * (b - a)$, so they can take advantage of FMA instructions. FMA instructions combine extended precision multiplication with addition. The benefits of FMA are two: First, owing to the increased precision of the intermediate product, the result is more accurate. Second, owing to the combination of multiplication with addition in the same instruction, the computational complexity is lower.

Figure 2.3 illustrates the $4 \times 4 \times 4$ neighborhood of control points each voxel requires. Each one of the $2 \times 2 \times 2$ colored sub-cubes of control points corresponds to one trilinear interpolation. For each voxel in the tile, the respective thread first calculates each one of the eight trilinear interpolations corresponding to each one of the colored cubes. Then, the thread calculates one last trilinear interpolation with the eight results of the eight trilinear interpolations that are corresponding to each one of the colored cubes.

The arithmetic operations that are needed by each trilinear interpolation (i.e., colored sub-cube) are independent, therefore we have independent instructions to increase ILP [115]. However, by creating several independent instructions and using loop unrolling, the required register count increases, leading to register spills (Section 2.5.4). Although registers spills are cached in L1/L2, cache blocks are regularly evicted. Instead, to avoid spills, we store one quarter of the total number of required control points in the shared memory.

2.5.4 *Implementation details of TT and TTLI*

Register tiling, which we employ in our approach, requires a careful management of the registers. We explain these difficulties and how we manage them in the following paragraphs. We also provide an analysis of the computational complexity of our approach.

Registers and performance with low occupancy

THREAD LEVEL PARALLELISM (TLP) The traditional way of hiding memory *latencies* is to do other, independent, work while waiting. In CUDA, thread blocks typically are independent work units and they consist of warps. We hide latencies by deploying many blocks/warps, many more than the amount the GPU can execute simultaneously. When one warp stalls, the GPU uses another warp. This method is called TLP [29].

INSTRUCTION LEVEL PARALLELISM (ILP) Instead of using independent blocks, as in TLP, to hide latencies, we can create independent instructions inside the same block. We achieve that by assigning to each thread multiple independent instructions (i.e., the inputs of the second instruction do not depend on the output of the first instruction). A basic way to increase ILP is loop unrolling. The shortcoming of loop unrolling is that more registers are required.

OCCUPANCY Depending on the amount of resources that are available to each MP the number of warps that can be active per MP is determined. The ratio of the active warps to the maximum number of warps that each MP supports is called *occupancy* [78]. Higher occupancy indicates that the GPU has more warps available when it tries to hide latencies with TLP.

Registers are one of the resources that can limit occupancy, therefore CUDA compiler, nvcc [79], applies various methods to reduce their number, for example:

- Removing regularly from registers the values that were loaded from shared or constant memory and reloading again later from the same source.
- Temporarily saving some of them in L1 / L2 cache, which is known as *register spilling* [73].

Cache is slower than registers, therefore we want to keep data in registers and to avoid register spills. In order to limit cache usage, we need to carefully place data in registers and make it stay there. When we force data to stay in registers, threads require more registers. More registers per thread reduces the amount of active warps per MP (because registers are a limited resource) and consequently occupancy becomes low. Low occupancy gives less opportunity for TLP. Therefore we must rely on ILP to hide latencies when we have low occupancy. Nevertheless, by utilizing the fast registers, the performance benefits greatly [115].

According to CUDA Programming Guide [80], most of the current GPUs have 65536 registers per MP and each thread can use up to 255 registers. The deformation field of a 3D image requires 64 control points and each control points comprises three values, one for each of the three coordinates (x, y, z). Therefore, we need $3 \cdot 64 = 192$ registers to store the control points only. Program execution requires additional registers. According to the compiler statistics, TT requires 235 registers in total, whereas TTLI requires all 255 registers (TTLI requires more due to additional loop unrolling). In both cases, due to the amount of required resources, each MP can run at maximum 256 threads, decreasing occupancy significantly. 256 threads correspond to eight warps - enough to occupy the four warp schedulers that NVIDIA's GPUs commonly have [80]. We group threads to four blocks of $4 \times 4 \times 4$ threads. We arrange threads in this configuration for two reasons. First, a cube is the geometrical structure that maximizes overlap and consequently minimizes memory transfers (i.e., minimizes Equation 2.5 in Section 2.5.6). Second, by keeping more and smaller blocks we follow the trend of future NVIDIA GPU generations, which have less resources per MP but more MPs [81].

Computational complexity

In order to evaluate the arithmetic performance of TTLI and TT, we perform the computational analysis of both implementations in this section.

TT For every voxel of the output image, we need to calculate the triple sum in Equation (2.1). Each operand of the summation requires the multiplication of one control point (ϕ) with three weights (B). Thus, each voxel requires $(64\ summands) * (3\ multiplications + 1\ accumulation) - 1 = 255$ vector (ϕ is a 3D vector in deformation fields) arithmetic operations. The calculation of Equation 2.1 requires $4+4+4 = 12$ scalar loads for the weights and 64 vector loads for the control points. If we use one weight for the $B_l(u) \cdot B_m(v) \cdot B_n(w)$ product, instead of three individual weights, the required operations decrease to $(64\ summands) * (1\ multiplications + 1\ accumulation) - 1 = 127$ (same as a parallel reduction) and the weights to be loaded increase to $4 * 4 * 4 = 64$. This is not suitable for our register-only implementations, because there are not enough registers to store the 64 weights and the use of one of the caches would impact the performance substantially (Section 2.5.4).

TTLI For every voxel of the output image, we reformulate the summation of the $4 \times 4 \times 4$ weighted control points to trilinear interpolations. We divide the $4 \times 4 \times 4$ cubic neighborhood to eight $2 \times 2 \times 2$ sub-cubes, as in Figure 2.3. Each sub-cube corresponds to a trilinear interpolation. A trilinear interpolation requires seven linear interpolations for its calculation. A linear interpolation has the form $a + w * (b - a)$, which equals to a subtraction and a fused multiply-accumulate (FMA) operation. Thus, for the eight sub-cubes and the ninth final sub-cube that is formed by the eight results of the eight trilinear interpolations, we have $(9 \text{ cubes}) \times (7 \text{ linear interpolations.}) \times (2 \text{ operations}) = 126$ operations for each voxel.

CONCLUSION TT requires 255 arithmetic operations to calculate the value of one voxel, whereas TTLI requires only 126 operations. TTLI has half the computational complexity of TT.

2.5.5 *Application of our methodology on CPUs*

The techniques that we use for the GPU implementation of B-spline interpolation can be also applied to the CPU implementations. CPUs have SIMD units (controlled by SSE/AVX instruction sets) to execute the same instruction on multiple input data. Normal non-SIMD units of the CPU operate on single values, commonly of 8 to 64 bits in length. SIMD units pack many single values in a special register. We call the special register as *vector* and each single value it can hold as *element*. Vectors of common CPUs (i.e., Intel, AMD) are 128, 256 or 512 bits in length [32, 52]. We denote as *slots* the places in a vector that can hold a single element. For example a 256-bit vector has 8 slots for 32-bit elements. Furthermore, each core of a multicore CPU contains one SIMD unit, thus we use multi-threading to process a significant amount of data simultaneously.

We employ the two main optimizations that derive from tile properties (Section 2.3.2) in the CPU implementations. First, by grouping in tiles, we can use the same set of control points for all voxels of the tile. Second, voxel and control points grids are aligned and therefore we can use LUTs for B-spline weights.

In addition, we employ the following two optimizations of our GPU approach. First, we use input loading optimization through L1 cache. Consecutive tiles overlap in the x-direction, as each thread in our algorithm iterates through all tiles in the x-direction first (i.e., row-wise). Second, we calculate the triple sum of Equation (2.1) as trilinear interpolations. One point worth noting in regards to FMA instructions is that they are available only in newer CPUs and only through vector instruction sets [13, 46]. Nevertheless, instead of explicitly using FMA instructions, we provide simple multiplication and addition operations. This way the compiler, depending on the CPU architecture, can choose to replace multiplication and addition operations with FMA operations. Thus, we maintain compatibility with older CPUs.

Our CPU implementations require minimal changes to the source code of the GPU. The main differences among our GPU and CPU implementations are two: a) how we load the input, and b) what we assign the vector elements for. We avoid using vector instructions explicitly because they are architecture specific and thus not compatible among different CPU models. To keep compatibility with different architectures, we use "C++ vector class library" (VCL), which uses heuristics to choose the appropriate version of instructions [31].

In the following sections we describe two CPU implementations, one using coarse-grained parallelization and one using fine-grained parallelization. We store control points and B-spline weights as 32-bit elements. In our descriptions we consider 3D tiles which contain $l \times m \times n$ voxels, i.e., l voxels in the z-direction, m voxels in the y-direction and n voxels in the x-direction.

Vector per Tile (VT)

In this method, we parallelize by processing many voxels of the tile at the same time. We assign each of the n voxels of a row of a $l \times m \times n$ tile to one slot of a vector. Each thread of the CPU calculates n voxels at a time, as we iterate through y and z directions. Let us consider the 2D example of Figure 2.6a, where $l = 0, m = 2, n = 2$. Let us suppose we have two threads, one for each tile. In first iteration, first thread will calculate pixels P00 and P01, whereas second thread will calculate pixels P02 and P03. In second iteration (we move one step farther in y-direction), first thread will calculate pixels P10 and P11, whereas second thread will calculate pixels P12 and P13.

The drawback of this approach is that while n can be of any size, vectors are fixed size. Therefore some slots of the vector may remain unused. The fewer the slots that remain unassigned are, the more efficient the implementation is.

Vector per Voxel (VV)

In this method, we parallelize by processing all trilinear interpolations that a single voxel requires at the same time. Specifically, each one of the eight elements of a 256-bit vector is assigned to one $2 \times 2 \times 2$ sub-cube of control points (Figure 2.3). Each element of the 256-bit vector holds the current state of the trilinear interpolation of the respective sub-cube. We can imagine the eight elements as eight "threads". The eight threads calculate the eight trilinear interpolations in Figure 2.3 concurrently.

Unlike VT, no vectors slots remain unused. Nevertheless, with this approach, loading the input control points from memory is more demanding because the respective vertices of sub-cubes are not consecutive in memory.

To increase the performance of this approach we take advantage of the fact that each control point of the deformation field contains three values (control points are 3D vectors in our case). We process all three values in the same iteration for the following three benefits. First, we simplify the addressing arithmetics for loading the input control points from memory because we need to calculate the memory address for all three values only once. Second, we increase ILP. This is necessary because this approach processes only one voxel and therefore there are more dependency chains (i.e., instructions that depend on the output of previous instructions) than with VT. Third, we create more elements that fill vector slots. This is useful because, after the calculation of the eight trilinear interpolations, we interpolate the eight results of the eight trilinear interpolations. But unlike the first part of the algorithm, where we have available eight trilinear interpolations to calculate simultaneously, one trilinear interpolation is not enough to fill a vector.

2.5.6 *Off-chip memory to on-chip memory data movement*

We use the external memory model [58] to describe the data movement from off-chip memory to on-chip memory. We consider a 3D image. Let us define M as the total number of voxels, $N = 64$ as the number of control points, T as the number of voxels inside each tile, and L as the size, in words (words are 32-bits long, a common size for storing integer and real numbers), of transactions into the cache (i.e., transactions between off- and on- chip memory). The L sized memory transfers of the three cases we are interested in are:

a) *No tiles*: When we do not have tiles, for each of the M voxels, we need to transfer N control points from global memory to shared memory. Transfers happen in L sized chunks. Hence, the total number of transfers required is

$$\frac{N \times M}{L} \tag{2.2}$$

b) *Hardware trilinear interpolation*: When we utilize the texture hardware for loading the input, for each of the M voxels, we need to transfer 2^3 (Section 2.4.2) control points from global memory to shared memory. Transfers happen in L sized chunks. Hence, the total number of transfers required is

$$\frac{2^3 \times M}{L} \tag{2.3}$$

c) *A block per tile*: When we use a block for each tile, for each tile we need to transfer N control points from global memory to shared memory. Each tile contains T voxels, thus the total number of tiles is M/T. Transfers happen in L sized chunks. Hence, the total number of transfers required is

$$\frac{N \times M}{T \times L} \tag{2.4}$$

d) *Blocks of tiles*: When we have 3D blocks of tiles, and each block contains $l \times m \times n$ tiles, for each block we need to transfer $(4 + l - 1) \times (4 + m - 1) \times (4 + n - 1)$ (Section 2.5.2) control points from global memory to shared memory. Each block contains $l \times m \times n$ tiles and each tile contains T voxels, thus the total number of blocks is $M/(l \times m \times n \times T)$. Transfers happen in L sized chunks. Hence, the total number of transfers required is

$$\frac{(4 + l - 1) \times (4 + m - 1) \times (4 + n - 1) \times M}{l \times m \times n \times T \times L} \tag{2.5}$$

Observations

We make the following four observations.

1. A hardware trilinear interpolation implementation requires fewer memory transfers than a no tiles implementation because $2^3 < N$ in all cases.

2. A block per tile implementation requires fewer memory transfers than a hardware trilinear interpolation implementation because $N/T < 2^3$ when $T > 8$. $T > 8$ is a rare case (T is 125 by default in NiftyReg).

3. A blocks of tiles implementation requires fewer memory transfers than a block per tile implementation because $\frac{(4+l-1)\times(4+m-1)\times(4+n-1)}{l\times m\times n} < N$ as long as a block contains more than one tile.

4. We divide (2.4) by (2.5) to find the improvement of our approach over the state-of-the-art. In our implementations we use blocks of $4 \times 4 \times 4$ tiles, and therefore $l = 4, m = 4, n = 4$. The result of the division shows that we reduce the data movement $11.9\times$.

5. The CPU implementations are a special case of Equation (2.5), in which $l = m = 1$, i.e., each thread processes contiguous tiles in the x-axis direction.

2.6 Evaluation Methodology

In this section we describe the infrastructure of our experiments.

2.6.1 *Dataset*

We performed a clinical study, through which we acquired two types of medical images. First, we acquired pre- and intra-operative MRI scans of the liver of a porcine model. Second, we acquired non-deformed and deformed Dyn-aCT scans of a patient-specific liver phantom. The detailed description of the acquired images is in Section 2.8.2 and in Section 2.8.1 respectively. From these images, we selected five pairs (we create pairs by corresponding either pre- and intra-operative images in case of the porcine model, or non-deformed and deformed images in case of the phantom). We pre-processed these pairs and we prepared them for non-rigid registration. Table 2.1 lists the properties of the acquired images. We utilize the five registration pairs for the analysis of both performance and accuracy.

Table 2.1
Image characteristics.

Registration pair	Resolution	Voxel count (millions)	Voxel Spacing
Phantom1	512x228x385	44.94	0.49x0.49x0.49
Phantom2	294x130x208	7.95	0.90x0.90x0.90
Phantom3	294x130x208	7.95	0.90x0.90x0.90
Pig1	303x167x212	10.73	0.94x0.94x1.00
Pig2	267x169x237	10.70	0.94x0.94x1.00

2.6.2 *Accuracy*

TEST PLATFORM For the experiments of accuracy we use one CPU and one GPU. The CPU is an Intel i7-7700HQ. The GPU is an NVIDIA GeForce GTX 1050. We use CUDA SDK v9.2 for the GPU.

METRIC In our implementations (for both CPU and GPU) we use single precision (32-bit) representations for real numbers [1]. In order to measure the accuracy, we create a double precision (64-bit) CPU version, which uses double precision operands. We calculate accuracy in three steps. First, we calculate the deformation field with both the implementation we are testing and the double precision CPU version. Second, we measure the accuracy in terms of average absolute difference between respective voxels of the deformation field. Third, we find the average of our five registration pairs for all tile size configurations.

2.6.3 *Performance*

TEST PLATFORM For the experiments of performance we use one CPU and two GPUs. The CPU is an Intel i7-7700HQ. The first GPU is an NVIDIA GeForce GTX 1050 (Pascal architecture [80]). We use CUDA SDK v9.2 for the first GPU. The second GPU is an NVIDIA GeForce RTX 2070 (Turing architecture [81]). We use CUDA SDK v10.1 for the second GPU. For both GPUs, we use CUDA event API to acquire the timing results.

METRICS We use two metrics for measuring the performance, time per voxel and speedup. First, we measure the *time per voxel*, i.e., the amount of time B-spline interpolation needs to calculate a single voxel. This happens in two steps: a) for each image, we divide the total B-spline interpolation time with the total number of voxels of the image to find the *average time* a single voxel requires, and b) we find the mean and standard deviation of average times per voxel of all images of the dataset. Second, we measure the *speedup* of each implementation over an optimized reference implementation on the respective platform. We use the optimized GPU implementation of NiftyReg library [74] as the GPU reference and the optimized CPU implementation of NiftyReg [74] as the CPU reference. For speedup, we also find the mean and standard deviation of all images of the dataset.

2.6.4 *Parameters*

We select five different tile sizes to evaluate the performance of the algorithms under different arrangements, namely $3 \times 3 \times 3$, $4 \times 4 \times 4$, $5 \times 5 \times 5$, $6 \times 6 \times 6$, $7 \times 7 \times 7$. We select these tile sizes because they are centered around $5 \times 5 \times 5$, which is the default of non-rigid registration in NiftyReg.

2.6.5 *Implementations we compare with*

We compare our approaches with other approaches from the state-of-the-art. In addition to NiftyReg [74] that we use as the reference for testing the performance, we compare with TH and TV implementations. As TH, we use the library from Ruijters et al. [97]. As TV, we created an implementation that uses tile properties and is based on the recent literature. It is a highly optimized version with both literature's observations and our own, using shared memory, LUTs for B-spline basis functions and ILP.

2.7 Results and Analysis

2.7.1 *Accuracy*

One of the benefits of FMA instruction in our implementations is that the result is more accurate due to the increased precision of the intermediate product of this instruction.

To show the effect of FMA in accuracy, we compare each approach with a high precision CPU implementation. Table 2.2 shows the average absolute difference between each *GPU* approach and the high precision CPU implementation. Similarly, Table 2.3 shows the average absolute difference between each *CPU* approach and the high precision CPU implementation.

Table 2.2
Average absolute difference of GPU approaches from high precision CPU implementation.

Implementation	Error (e^{-6})
Nifty GPU	5.3
Texture Hardware	9245
Thread per Voxel	5.5
Thread per Tile	5.6
Thread per Tile (Interp.)	2.8

Table 2.3
Average absolute difference of CPU approaches from high precision CPU implementation.

Implementation	Error (e^{-6})
NiftyReg CPU	6.0
Vector per Tile	3.0
Vector per Voxel	3.0

We make three observations. First, our implementations that use FMA instructions (TTLI for GPU, VT and VV for CPU) are almost two times more accurate than the implementations that do not use FMA instructions. Second, TH approach is significantly less accurate than the rest of the implementations, as we expect from the low accuracy of interpolation hardware [80]. Third, GPU is competing well with CPU in regards to accuracy.

CONCLUSION In general, accuracy of software-only implementations is significantly better than the hardware one. TTLI and FMA-based approaches are over 3000× more accurate than TH. We refer the reader to [116] for more information on floating point accuracy for GPUs.

2.7.2 *Performance on GPU*

To study the effect of our optimizations, we measure average time per voxel (Section 2.7.2) and speedups (Section 2.7.2). To show the performance and stability among different GPU generations, we use two GPUs of different generations.

Time per voxel

In this section, we analyze the mean and standard deviation of the average time per voxel. Figure 2.8 and Figure 2.9 show the average time per voxel for GTX 1050 and RTX 2070 GPUs respectively. We group times per voxel by implementation and for each group we vary the tile size. With this grouping scheme, we show how each implementation behaves when we vary the tile size. We draw four major conclusions.

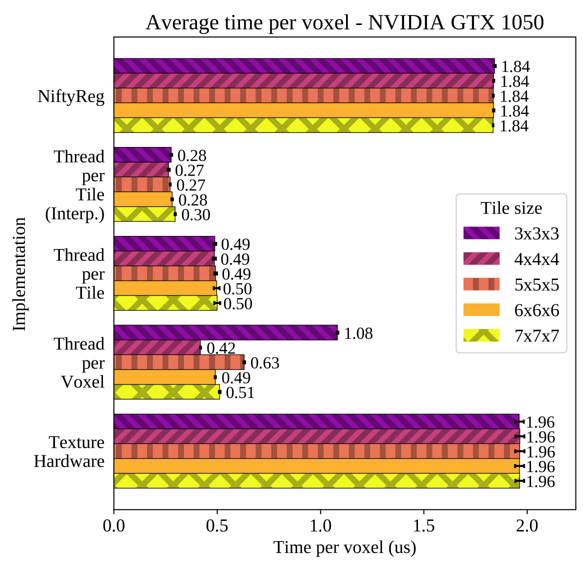


Figure 2.8
Average time per voxel of the five registration pairs for various tile sizes on GTX 1050 GPU

First, our TTLI approach is faster in all cases, for both GPUs.

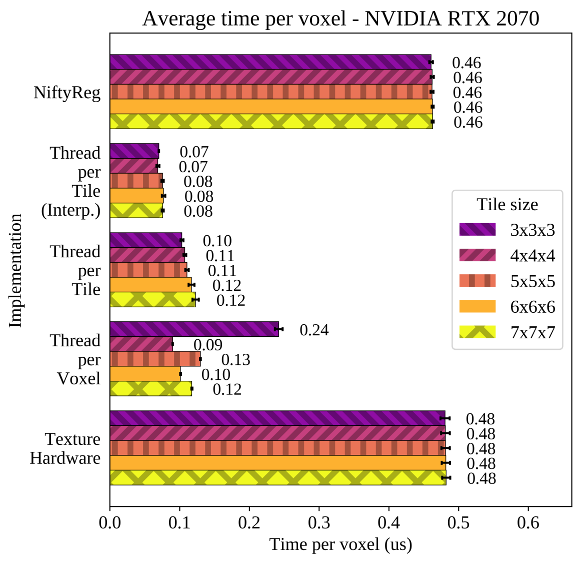


Figure 2.9
Average time per voxel of the five registration pairs for various tile sizes on RTX 2070 GPU

Second, we would expect the performance of Thread per Tile approaches to improve with bigger tile sizes, as they require fewer memory transfers from off- and on- chip memories. However, this is not the case. The reasons are three. First, the thread blocks that calculate the voxels at the borders of the image have inactive threads in our current implementations. The issue with the borders occurs when the total amount of voxels the blocks process does not divide exactly the dimensions of the image. Then, the threads corresponding to the remainder of voxels stay inactive. In our approach, each thread block works on many tiles. The bigger the tiles, the more voxels each block processes and the bigger the remainder can be. For example, in our implementation we use blocks with $4 \times 4 \times 4$ threads. In the worst case of our experiments, i.e., a tile of size $7 \times 7 \times 7$, each block processes $28 \times 28 \times 28$ voxels. This amount of voxels is considerable in comparison to the size of the images in our dataset (Table 2.1) and leads to many inactive threads. Second, tiles decrease the coalescence of GPU memory accesses. In CUDA, in order for the memory access to be optimal, consecutive threads must access consecutive memory locations [80]. In our approach, a single thread processes the entire tile and subsequently stores in memory the result of the entire tile. The bigger the tiles, the more consecutive voxels a single thread has to store. Figure 2.7, Step 3, shows the uncoalesced memory store of Thread per Tile approaches. Third, some resources of the GPU may remain underutilized. In order to submit work to the GPU we have to divide the workload into thread blocks. The amount of the blocks depends on the dimensions of the images. The GPU scheduler distributes blocks to MPs. If the number of MPs does not divide the amount of blocks exactly, some SMs may remain idle (tail effect). As the tile size gets bigger and the image size becomes smaller, the number of created blocks becomes smaller. The fewer the created blocks and the larger the number of SMs, the more significant the impact of the remainder of the division. GTX 1050 has 5 SMs and RTX 2070 has 36 SMs, therefore the impact of the tail effect is more pronounced on RTX 2070. In conclusion, the performance of our approach in regards to different tile sizes, is a balance between the acceleration that the reduction of memory transfers produces and the deceleration that border effects and memory uncoalescence cause.

Third, we make three observations for NiftyReg, our reference. First, NiftyReg is considerably slower than the rest of the implementations, similarly to TH. Second, tile size does not affect the performance of NiftyReg because NiftyReg is not utilizing tile properties. Third, the timings of NiftyReg are constant for all tile sizes. As we use NiftyReg, which is constant, as reference, the speedup graph in the next section is also a scaled time graph.

Fourth, the small standard deviation of the images of our dataset shows that the performance is not affected by the image. The reason is that B-spline interpolation acts on all voxels and therefore the image structure does not affect B-spline interpolation. As we show in Section 2.5.4, the amount of work is directly proportional to the number of voxels.

Speedup

In this section, we analyze the mean and standard deviation of speedups. Figure 2.10 and Figure 2.11 show the average speedup for GTX 1050 and RTX 2070 GPUs respectively. We group speedup by tile size and for each group we vary the implementation. With this grouping scheme, we show how each implementation behaves in comparison to the other implementations. We draw four major conclusions.

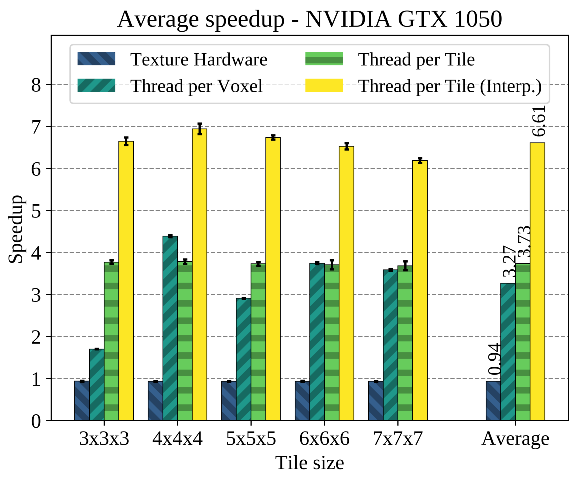


Figure 2.10
Average speedup of B-spline interpolation of the five registration pairs for various tile sizes on GTX 1050 GPU. The B-spline interpolation of NiftyReg is the reference of the speedup

First, our TTLI approach is outperforming the rest in all cases and is 6.5× faster than NiftyReg , on average. TTLI outperforms all other implementations by at least 1.56× on GTX 1050 and 1.3× on RTX 2070 (the worst case is TV for $4 \times 4 \times 4$ tile size).

Second, despite our optimizations of memory accesses (Section 2.5.2) , TT does not improve a lot over TV. The reason is that, after we eliminate the memory latencies, arithmetic computations become the bottleneck of the implementation. We use NVIDIA's Visual Profiler [78] to solve the problem in two steps. First, we confirm that our GPU kernel is compute bound (the profiler collects statistics of utilization of memory and computation units). Second, we use the Program Counter sampling of the same tool. The profiler collects, at periodic

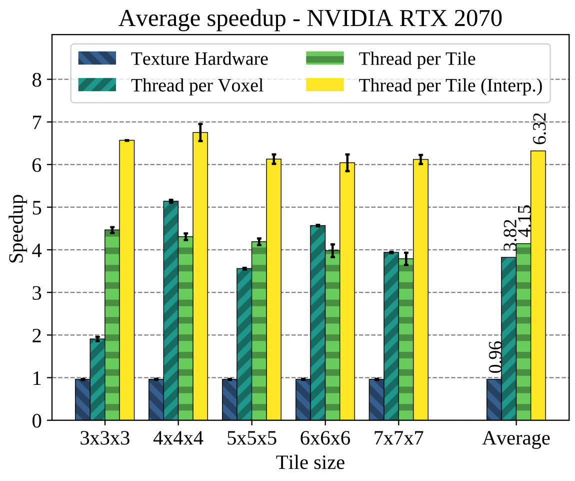


Figure 2.11
Average speedup of B-spline interpolation of the five registration pairs for various tile sizes on RTX 2070 GPU. The B-spline interpolation of NiftyReg is the reference of the speedup

intervals, concurrent samples of the state of two hardware units of the GPU: a) the warp scheduler, which is the unit responsible for scheduling the GPU instructions, and b) the warp Program Counter, which is a counter that points to the instruction that is currently executing [78]. Almost 80% of the acquired samples indicate that the warps could issue (submit for execution) their instructions without waiting. This result implies that the algorithm is well optimized and further improvement can only be achieved with structural changes to the algorithm. Reformulating the summation of basic Thread per Tile approach to trilinear interpolations (Section 2.5.3) solves this issue. According to Section 2.5.4, computational complexity of TTLI is half that of TT, which is reflected to the results. TTLI is 50% − 80% faster than TT.

Third, TH has almost the same performance as NiftyReg and is the slowest approach in our experiments. The performance does not vary with the different tile sizes as no tile properties are used, similarly to NiftyReg.

Fourth, our approach works well on both Pascal architecture (GTX 1050) and Turing architecture (RTX 2070) GPUs. Figure 2.10 and Figure 2.11 show mostly similar trends of the data.

GPU conclusion

We compare the performance of our approach in terms of time and speedup over a range of images. Our approach outperforms all other approaches and improves performance by at least 1.3×. We verify the benefits in performance on two GPU architectures, including Turing architecture (RTX 2070), the latest architecture at the time of writing. The speedup of the reference, NiftyReg, is 6.5× on average and therefore our approach is a suitable alternative for B-spline interpolation in non-rigid registration.

2.7.3 *Performance on CPU*

We apply the methodology of our GPU approach to CPU (Section 2.5.5). To study the effect of our methodology on CPU, we measure average time per voxel (Section 2.7.3) and speedups (Section 2.7.3).

Time per voxel

In this section, we analyze the mean and standard deviation of the average time per voxel of the CPU approaches. Figure 2.12 shows the average time per voxel for an i7-7700HQ CPU. We group times per voxel by implementation and for each group we vary the tile size. With this grouping scheme, we show how each implementation behaves when we vary the tile size. We draw four major conclusions.

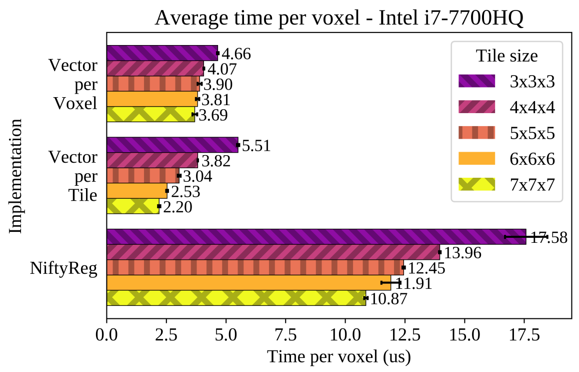


Figure 2.12
Average time per voxel of the five registration pairs for various tile sizes on i7-7700HQ CPU

First, our implementations outperform the reference in all cases.

Second, in VV, time decreases with increasing tile size. The reason is that the bigger the tile is, the more the voxels that use the same control points are (Section 2.5.5). Thus, we have fewer memory transfers with bigger tiles.

Third, in VT, time decreases with increasing tile size for two reasons. First, bigger tiles fill more slots of the SIMD vectors (Section 2.5.5) and therefore each CPU thread calculates more voxel simultaneously. Second, bigger tiles require fewer transfers from memory (Section 2.5.6).

Fourth, in NiftyReg, time decreases with increasing tile size. The CPU implementation of NiftyReg uses tile properties to avoid loading control points that overlap in the x-axis. The bigger the tile is, the more the overlapping voxels are.

Speedup

In this section, we analyze the mean and standard deviation of speedups of the CPU approaches. Figure 2.13 shows the average speedup for an i7-7700HQ

CPU. We group speedup by tile size and for each group we vary the implementation. With this grouping scheme, we show how each implementation behaves in comparison to the other implementations. We draw three major conclusions.

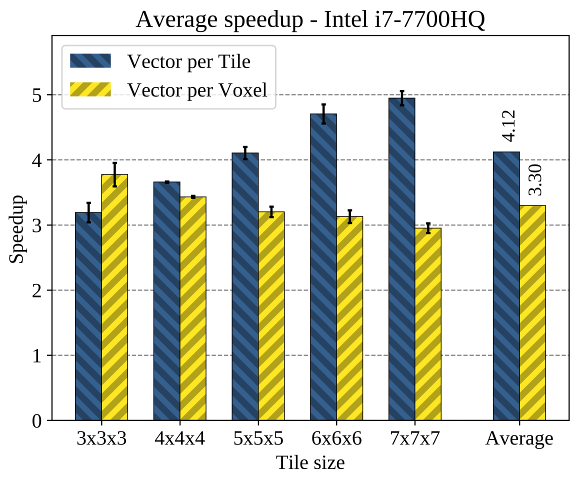


Figure 2.13
Average speedup of B-spline interpolation of the five registration pairs for various tile sizes on i7-7700HQ CPU. The B-spline interpolation of NiftyReg is the reference of the speedup

First, our CPU implementations outperform the reference in all cases by at least 3× and they are up to 5× faster.

Second, VT generally outperforms VV except in the $3 \times 3 \times 3$ case, where more slots of the vector remain empty (Section 2.5.5). It is $3\times$ to $5\times$ faster than the reference. The speedup of VT increases as the tiles get bigger.

Third, VV behaves better than VT when the tile size is small, because VV uses all vectors slots and therefore tile size does not affect the performance as much as VT (Section 2.5.5). On average, it is $3\times$ to $4\times$ faster than the reference. The speedup of VV decreases as the tiles get bigger, although VV becomes faster with increasing tile sizes, as we show in Section 2.7.3. The reason is NiftyReg also becomes faster.

CPU conclusion

We compared the performance of our CPU implementations in terms of time and speedup over a range of images. We observe that all implementations become faster as the tile size increases because they need to read fewer input data. Our CPU implementations outperform the reference in all cases by 3× to 5×. We conclude that our methodology can be beneficial even for CPU.

2.8 Clinical validation of image registration implementation

In order to test our novel implementation of parallelization of B-spline interpolation on GPU, we performed a pre-clinical study to solve a clinical application scenario. In the medical field, image registration aims to establish spatial correspondences between volumetric datasets [110]. The clinical applications include techniques such as information fusion (inter-modality: combining MRI, CT and US data) [72], patient motion (intra-modality: aligning of temporal series) [77] and patient changes (intra and inter-modality: pre-post treatment scans or volume changes over time) [61]. Our work on B spline interpolation is focused mainly on the deformation application of image registration. Figure 2.14 shows the placement of our method within image registration algorithms.

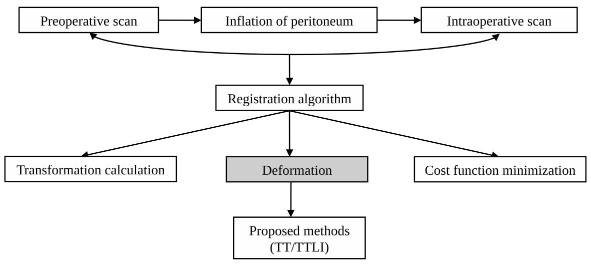


Figure 2.14
The association of our method in the pneumoperitoneum compensation workflow

Minimally invasive procedures require pneumoperitoneum (practice to inflate the patient's abdomen to allow space for insertion of surgical tools). As can be seen in Figure 2.1, the effect of inflating the peritoneum causes a large deformation of the liver surface [8, 44, 114]. This deformation causes large errors for image guided surgery applications because the intra-operative deformation is not corrected. A procedure to compensate for this deformation, in case of availability of intra-operative imaging, is to use image registration.

Non-rigid registration using FFD, as presented by Rueckert et al. in [96] and optimized by Modat et al. in NiftyReg [74], provides a solution. NiftyReg is capable of all phases of registration (rigid, affine, and non-rigid) and therefore we chose it as a suitable platform to apply our method and evaluate the performance improvement. Moreover, as Modat et al. mentions in [74], B-spline interpolation is one of the most significant bottlenecks in terms of performance. However, the FFD implemented in NiftyReg through Normalized Mutual Information (NMI), a measure of similarity of the images, is not a registration algorithm specific to compensation of liver deformations due to pneumoperitoneum. Hence, testing was performed throughout two different experiments with different subjects and imaging modalities: *Experiment 1* makes use of a patient-specific liver phantom [87] and DynaCT scanning, whereas *Experiment 2* was performed by the authors through the use of a porcine model and an MRI, to validate the registration process in vivo.

2.8.1 *Experiment 1: Validation in liver patient-specific phantom.*

The patient-specific liver phantom aforementioned presents a total of 5 tumors and a blood vessel tree. We used these structures to evaluate the registration process (Section 2.8.3). The liver phantom used throughout the experiments was produced by the ARTORG centre and Cascination® [87] and has been used also by Teatini et al. in [109] for registration studies. We performed a series of intra-operative CT (Artis Zeego, Siemens®) scans (DynaCT) of the liver phantom the phantom positioned on the surgical table. We executed of a total of 5 liver phantom scans, between which we applied various deformations to the liver phantom.

We performed segmentation of the liver phantom DynaCT scans to remove artifacts, and applied the segmentation to the images to perform image registration on parts of the image including only the liver phantom. Resulting output images show that the shape of the output images of the registration match rather precisely those of the reference deformed images. An example of the output of the registration process is shown in Figure 2.15.

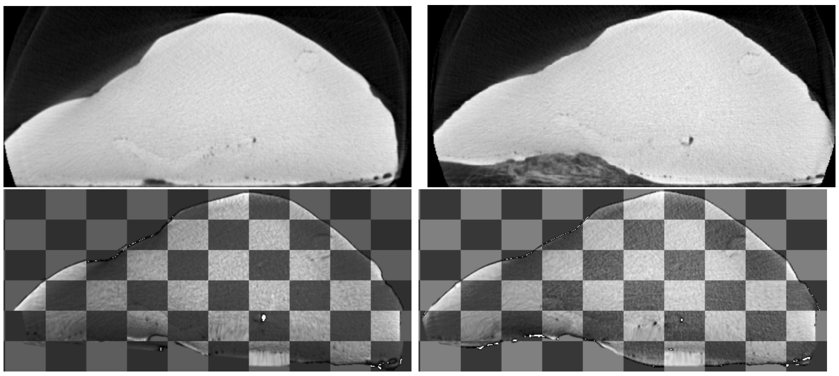


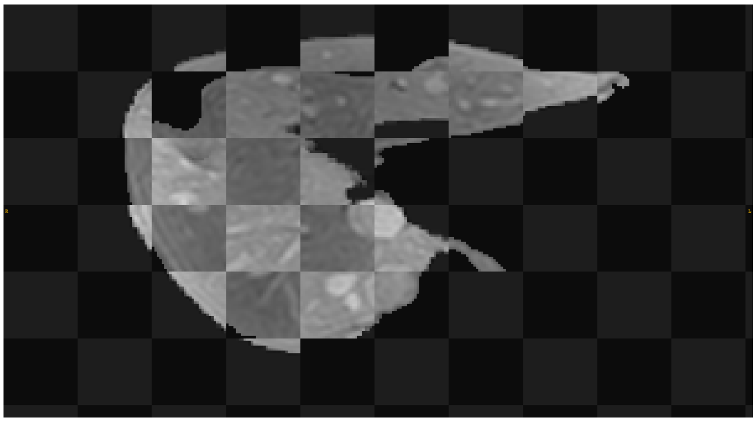
Figure 2.15
DynaCT scans of the liver before deformation and after deformation. The result of the registration shows a correctly performed registration

2.8.2 *Experiment 2: Validation in porcine study.*

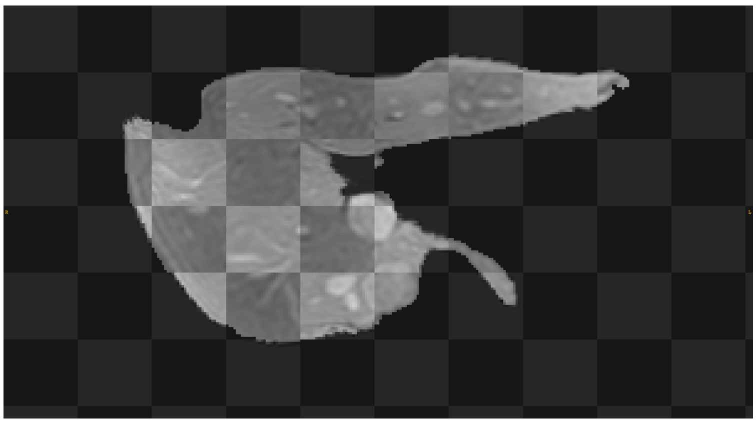
We performed a porcine study to acquire pre-operative and intra-operative (post pneumoperitoneum) MRI scans in order to acquire data regarding the deformation that the liver undergoes due to pneumoperitoneum. We performed this study at Oslo University Hospital through the use of a 3T Siemens MRI scanner, model Ingenia Philips ®. The surgeon applied pneumoperitoneum through the use of a Verress needle. We applied a pneumoperitoneum at 14 mmHg pressure, although some pressure was lost throughout the duration of the MRI scans. Both MRI scans were performed with injection of contrast to improve imaging of the liver parenchyma and blood vessels (Flow rate 5.0 and Volume 11.0, based on the weight of the animal at 55kg).The MRI scans are thin sliced (1 mm and 1.5 mm) enhanced-T1 high-resolution isotropic volume examination (e-THRIVE) scans.

2.8.3 *Qualitative assessment of the registration procedure*

We used our method to perform the registration between the MR images, and we used qualitative assessment of the registration procedure using a checkerboard validation procedure [92]. In order to decrease the required amount of GPU memory, we cropped the images to fit the volume of the segmented liver. We then compared our results to the results from an affine (rigid) registration procedure (visible in Figure 2.15). The affine transformation presents several mismatches (as could be expected, and clearly visible in the outer borders of the liver), whereas the non-rigid registration method provides very accurate (only based on a qualitative assessment) registration for both the parenchyma (the outer shape of the liver is preserved correctly), and the tumor and vessel structures present in the porcine model are also quite consistent between images (Figure 2.16).



(a)



(b)

Figure 2.16
Comparison of registration through qualitative checkerboard assessment on porcine model. a) Affine registration, b) Non-rigid FFD

2.8.4 *Quantitative assessment of the registration procedure*

With respects to the original FFD algorithm [74], we did not apply changes (only the deformation section, as shown in Figure 2.14). Hence, to confirm that nothing was modified with respect to the results obtainable through the GPU implementation in the NiftyReg, we performed a quantitative analysis using difference images. The image differences displayed no real differences between various implementations, which verifies that quantitatively the method does not introduce changes. Validation of accuracy of the registration method can be inferred from the original studies performed by Modat et al. in [74].

Table 2.4
Mean absolute error (and standard deviation) for our dataset. Columns 2, 3, 4 compare the the deformed intra-operative image with the output of the registration for 3 methods: 1) affine, 2) non-rigid with NiftyReg, 3) non-rigid with our approach. The last column shows the mean absolute error between registration with NiftyReg and our approach.

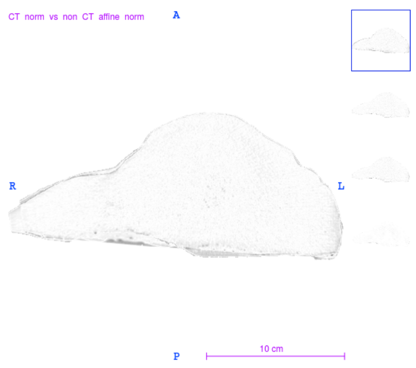
Registration pair	Mean absolute error			
	Affine	NiftyReg	Ours	NiftyReg vs Ours
Phantom 1	0.2288 ±0.5403	0.1308 ±0.2822	0.1301 ±0.2835	0.0377 ±0.11
Phantom 2	0.2338 ±0.4971	0.179 ±0.311	0.1718 ±0.3141	0.0243 ±0.0768
Phantom 3	0.256 ±0.5181	0.1718 ±0.3013	0.1739 ±0.3109	0.0198 ±0.102
Porcine 1	0.2007 ±0.551	0.0724 ±0.2388	0.0723 ±0.2392	0.0088 ±0.053
Porcine 2	0.1623 ±0.477	0.0708 ±0.1932	0.0723 ±0.2007	0.0066 ±0.0456
Average	0.2163 ±0.5167	0.125 ±0.2653	0.1241 ±0.2697	0.0194 ±0.0774

Difference images

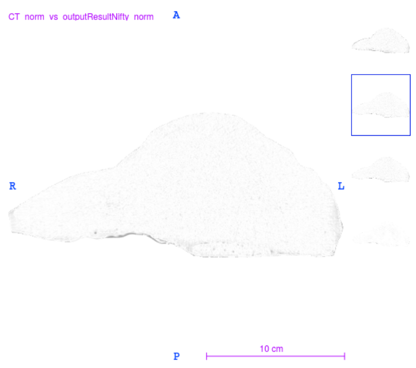
Figure 2.17 shows the normalized (z-score) difference images for the liver phantom. Figure 2.17a compares the affine transformed non-inflated (pre-operative) liver to the inflated (intra-operative) liver. Figures 2.17b and 2.17c compare the inflated liver to the non-rigid transformed non-inflated liver with NiftyReg and our approach, respectively. We draw two observations. First, affine transform performs worse than non-rigid registration. Second, both approaches of non-rigid registration give almost the same result. In order to facilitate comparison between the two non-rigid registration approaches we create a difference image of the two non-rigid registration images in Figure 2.17d. We observe that the difference image of them is almost completely black which indicates that the two registrations are very similar. High similarity means that the difference will not be easily visible to the human eye.

Figure 2.18 shows the difference images for the liver phantom. Figure 2.18a compares the affine transformed non-inflated (pre-operative) liver to the inflated (intra-operative) liver. Figures 2.18b and 2.18c compare the inflated liver to the non-rigid transformed non-inflated liver with NiftyReg and our approach, respectively. Similar to the phantom, affine transform performs worse than non-rigid registration and both approaches of non-rigid registration give almost the same result. The difference image of the two non-rigid registration images in Figure 2.18d again indicates that the two registrations are very similar.

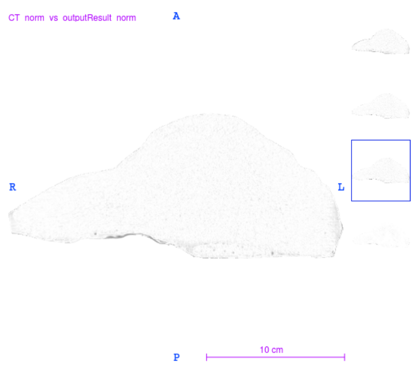
To facilitate understanding of the differences we also use standard metrics to quantify them, i.e., we use Mean Absolute Error (MAE), Sum of Squared Errors (SSE) and Structured Similarity Index (SSIM). We show the MAE in Table 2.4. The figures confirm our observations on the difference images, i.e., non-rigid registration performs much better than affine registration. Moreover, the MAE between our approach and original NiftyReg is minimal, with our approach having the edge slightly. SSE and SSIM in Table 2.5 confirm the same observations.



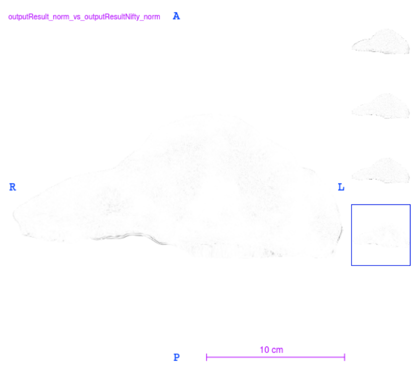
(a) Difference of affine registration output and deformed liver



(b) Difference of NiftyReg registration output and deformed liver



(c) Difference of registration output with our approach and deformed liver



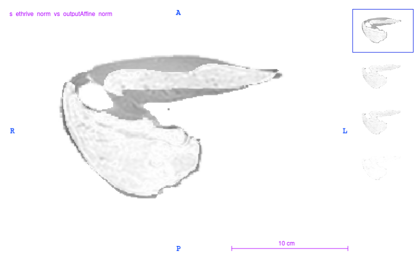
(d) Difference of NiftyReg registration and our approach

Figure 2.17
Comparison of registration through difference images on liver phantom.

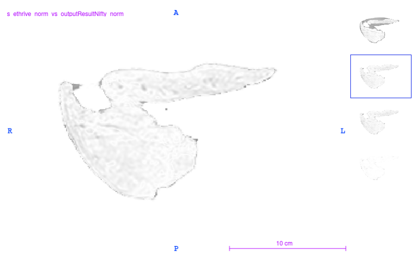
Table 2.5
Sum of squared errors (Left) on normalised outputs of affine registration and non-rigid registration with our approach and original NiftyReg, using the intra-operative image as reference. Structured Similarity Index Metric (Right) of the registration output, using the intra-operative image as reference).

Registration pair	SSE			SSIM		
	Affine	Ours	NiftyReg	Affine	Ours	NiftyReg
Phantom 1	322951	171241	170725	0.865	0.929	0.934
Phantom 2	223659	150963	152037	0.916	0.952	0.946
Phantom 3	248370	156682	153190	0.889	0.952	0.95
Porcine 1	8439875	3994665	3997671	0.797	0.912	0.911
Porcine 2	12152982	12028758	12027581	0.716	0.737	0.737
Average	4219435	0.1240	0.1249	0.8368	0.8963	0.8956

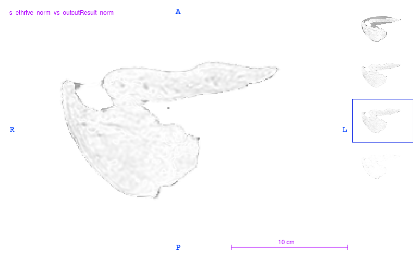
2.8.5 *Performance evaluation on clinical data*



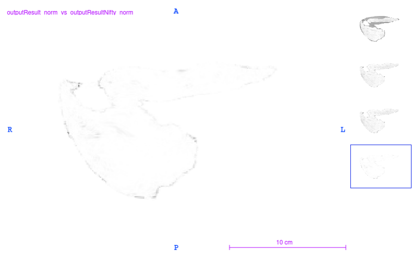
(a) Difference of affine registration output and deformed liver



(b) Difference of NiftyReg registration output and deformed liver



(c) Difference of registration output with our approach and deformed liver



(d) Difference of NiftyReg registration and our approach

Figure 2.18
Comparison of registration through difference images on porcine liver.

Evaluation methodology

We test the performance of non-rigid registration on two platforms: a) Intel i7-7700HQ CPU and GTX 1050 GPU, and b) Intel i7-8700 CPU and RTX 2070 GPU. We use five image pairs from the data in Sections 2.8.1 and 2.8.2. The tile size is set to $5 \times 5 \times 5$, the default setting of NiftyReg. As timing results we use the registration time that NiftyReg application reports at the end of the registration. These timing results also include the time for loading and storing images, allocating memory, initialization etc.

To test the contribution of the accelerated B-spline interpolation to the total time required for the registration of medical images, we integrate our approach (TTLI) to NiftyReg (Section 2.4.2) and compare it with the original. There are two important parameters to consider before describing our methodology for collecting the timing results.

First, the B-spline interpolation implementations affect the number of iterations until convergence of the cost function (Figure 2.14). However, the differences between the accuracy of different B-spline implementations do not significantly impact the quality of the registration result (Sections 2.8.3 and 2.8.4). To our knowledge, the difference in number of iterations is because the difference in accuracy is enough to change the local optima of the optimization function. Therefore, the registration completion time may differ not because of the time required for the B-spline interpolation, but because one of the implementations happens to find different local optima and converges faster, in fewer iterations.

Second, NiftyReg uses a pyramidal approach, i.e., NiftyReg resamples the input images to different sizes. The bottom level of the pyramid is the original large image, and as we go up the pyramid, NiftyReg subsamples the previous level (i.e., halves the resolution in every dimension). The bottom level requires significantly more computation time than the upper levels (and we emphasize there are many iterations per each level until convergence).

In order to have a fair comparison between approaches, we minimize the impact of number of iterations to the total time of registration, with two methods. First, we use a three level pyramid. We limit the number of iterations so that the cost function iterates on the bottom level for the same number of iterations for both the implementations. Second, we use a one level pyramid. We, again, limit the number of iterations so that the iterations are the same for both implementations. The first method gives better quality of registration but the upper levels (i.e., smaller images) affect (although not much) the total timing. The second method gives slightly worse registration result, but there are no other pyramid levels to affect the total registration time.

Results and analysis

Tables 2.6 and 2.7 show total registration time, speedup of our approach and allowed iterations of the cost function on GTX 1050, for one and three pyramid levels respectively. Tables 2.8 and 2.9 show total registration time, speedup of our approach and allowed iterations of the cost function on RTX 2070, for one and three pyramid levels respectively.

Table 2.6
1 pyramid level registration on GTX 1050. The table shows the speedup of registration with our improved B-spline interpolation GPU approach. The iteration column shows the number of iterations of the cost function.

Registration pair	Speedup	Original (s)	Our approach (s)	Iterations
Phantom1	1.34×	70.6	52.8	150
Phantom2	1.29×	13.3	10.3	150
Phantom3	1.26×	11.1	8.8	120
Pig1	1.32×	19.9	15.1	120
Pig2	1.30×	24.7	19.1	250
Average	1.30×			

We draw two major conclusions.

First, registration with our B-spline interpolation approach is faster in all images, on both GPUs. The speedup of registration is 1.29× on average on GTX 1050, whereas the speedup on RTX 2070 is 1.13× on average.

Second, the speedup of registration depends on what portion of the total registration time the GPU approach takes. According to Amdahl's law [3], the smaller the time portion of B-spline interpolation in the original registration routine is, the smaller the impact of accelerating B-spline interpolation is. For

Table 2.7
3 pyramid levels registration on GTX 1050. The table shows the speedup of registration with our improved B-spline interpolation GPU approach. The iteration column shows the number of iterations of the cost function.

Registration pair	Speedup	Original (s)	Our approach (s)	Iterations
Phantom1	1.28×	54.2	42.3	150
Phantom2	1.24×	10.5	8.5	150
Phantom3	1.31×	8.7	6.6	120
Pig1	1.24×	11.1	9.0	120
Pig2	1.32×	20.1	15.1	250
Average	1.28×			

Table 2.8
1 pyramid level registration on RTX 2070. The table shows the speedup of registration with our improved B-spline interpolation GPU approach. The iteration column shows the number of iterations of the cost function.

Registration pair	Speedup	Original (s)	Our approach (s)	Iterations
Phantom1	1.17×	29.3	25.2	150
Phantom2	1.15×	5.9	5.2	150
Phantom3	1.09×	4.9	4.5	120
Pig1	1.16×	8.7	7.5	120
Pig2	1.14×	10.5	9.2	250
Average	1.14×			

Table 2.9
3 pyramid levels registration on RTX 2070. The table shows the speedup of registration with our improved B-spline interpolation GPU approach. The iteration column shows the number of iterations of the cost function.

Registration pair	Speedup	Original (s)	Our approach (s)	Iterations
Phantom1	1.15×	23.0	20.0	150
Phantom2	1.12×	4.8	4.3	150
Phantom3	1.06×	4.0	3.8	120
Pig1	1.08×	5.1	4.7	120
Pig2	1.14×	8.6	7.6	250
Average	1.11×			

example, RTX 2070 is much faster than GTX 1050, and the code that runs on RTX 2070 completes its execution in much shorter time than on GTX 1050. Therefore, GPU code running on RTX 2070 takes a smaller portion of total registration time, and the effect on speedup is smaller. This is the reason that RTX 2070 has smaller average speedups.

CONCLUSION We apply registration to the images of our dataset (Section 2.6.1) and we compensate for pneumoperitoneum. We obtain a speedup of non-rigid registration of 29% and 13% on average on GTX 1050 and RTX 2070 respectively. Therefore, we believe our approach is beneficial to the performance non-rigid registration

2.8.6 *Memory requirements*

The memory requirements remain the same as the original implementation, except allocations for a) LUTs in constant memory (340 bytes), and b) control points in shared memory (12288 bytes). The memory space we use is much less than the maximum available, and therefore it is not a limitation of our implementations.

2.9 Conclusion

In this work, we present a new thread assignment scheme that reduces memory traffic by at least $11.9\times$ compared to other state-of-the-art B-spline interpolation approaches. We achieve that by assigning one GPU thread per tile. This assignment has two key advantages. First, the loading of the input demonstrates significant overlap. Second, the control points, that each group of tiles needs, stay permanently in registers. To further enhance the performance of our implementation, we rearrange the weighted summation of control points to trilinear interpolations. This rearrangement has two key advantages. First, it reduces the computational load. Second, it increases accuracy. We apply our optimized approach to non-rigid registration of medical images to enhance the performance of registration in time critical applications, like IGS.

The results confirm that by restructuring the B-spline interpolation algorithm to reduce the number of memory transfers is indeed beneficial to the performance. In addition, the use of trilinear interpolation proves helpful not only for performance, but also for the total accuracy. The proposed method, TTLI, improves accuracy by up to $3300\times$ and performs up to $7\times$ faster in comparison to the other GPU implementations. We integrate the optimized B-spline algorithm into NiftyReg medical image registration library, and improve the performance of non-rigid registration by 29% and 13% on our two systems.

We conclude that our methodology improves the performance of B-spline interpolation, in terms of both speed and accuracy. Our approach is an efficient replacement of B-spline interpolation in non-rigid registration. We publish our work ([122]) and make our source code publicly available at [119].

2.10 Future work

This work is focused on medical image registration, but with small changes it can be applied to more general scenarios. For example, although for the field of medical imaging grids with uniform spacing are usually enough, with minimal changes the implementations can be extended to support non-uniform grids. The most significant change needed is to calculate LUTs for B-spline basis functions weights on-the-fly, which has a slight impact in performance. Moreover, the same methods can also be applied to generic image interpolation. In this case, image pixels are used as control points and only image zooming is supported, which is a requirement for tiles to be created (the code needs only to be adapted for different handling of image border). Image size can be increased by integer multiples or by real number multiples with a slight performance hit for the latter (due to non-uniform grid).

Regarding registration, it would prove beneficial to merge multiple steps with B-spline interpolation (careful arranging of registers will be required). Furthermore, by merging other steps, uncoalescence of the output could possibly be avoided. In addition, by optimizing the rest of the registration process the time necessary for the registration diminishes, possibly allowing fast intra-operative updates without intra-operative CT acquisitions, for example through liver models reconstructed with US or through stereo video reconstructions [109].

The importance of speedup for image registration through FFD is not only important for the application to pneumoperitoneum compensation, but also for compensating several other deformations that the liver commonly undergoes throughout surgery. If the registration procedure could allow for real-time registration, FFD could be used to compensate deformations caused by a surgeon when lifting the liver with surgical instrument or by liver mobilization (resection of liver ligaments).

Part III

SPARSE MATRIX-MATRIX MULTIPLICATION

3 ACCELERATING SPARSE MATRIX-MATRIX MULTIPLICATION USING TENSOR CORES

Contents

3.1 Motivation

Sparse general matrix-matrix multiplication, similar to its dense counterpart, performs the Matrix Multiplication (MM) of two matrices. The main difference between sparse and dense matrix multiplication is that we have to account for sparse matrices, which contain mostly zero elements. spGEMM is an important component in scientific and data analytics applications. Graph analytics [22, 57], Bread-First-Search (BFS) [34], Algebraic Multigrid (AMG) [11], Schur complement [118], etc. use sparse matrices. Sparse matrices, which often contain million of elements, require matrix multiplication methods that do not waste computing resources on elements that are zero. The diverse structure and density of sparse matrices poses difficulties in regards to memory management and load balancing in parallel systems.

The re-emergence of deep learning motivated the creation of application specific integrated circuits (ASIC) that specialize in MM, providing significant performance boost over standard multiplication units. Such ASICs are Tensor Core Units (TCUs) from NVIDIA [83] and TPUs from Google [37]. We want to utilize tensor units to accelerate spGEMM. TCUs from NVIDIA provide an attractive target for two reasons. First, accessibility. They are widely available as they are included in the new generation of GPUs from NVIDIA. Second, mixed precision. Typically, in regards to deep learning 16-bits of precision (or less) are sufficient for training purposes and therefore tensor unit manufacturers opt for lower precisions. Mixed precision widens the application field to scientific problems which are more demanding w.r.t. precision. Mixed precision achieves this by mixing 16-bit inputs and high precision multiplication and accumulation.

Our work is motivated by three key observations. First, blocking sparse matrix storage formats [123], which group the elements of the matrix into rectangular *tiles*, are a good fit for TCUs which expect rectangular matrices as input. Second, TCUs are very efficient even when they are not fully occupied [19]. Third, even though TCUs support only low precision inputs, they can operate in mixed precision mode to perform operations that require higher precision, like GEMM [42, 70]. Therefore the key idea is to partition the input in tiles and multiply tiles with TCUs. Tiles are sparse, but TCUs perform MM efficiently even when not fully occupied. Mixed precision mode is necessary in order to keep sufficient accuracy when multiplying large matrices.

3.2 Objectives

The primary objective of the 2nd part of this work is to accelerate spGEMM by taking advantage of TCUs. To the best of our knowledge, this is the first proposal of using TCUs in the context of spGEMM. Our methodology has two advantages. First, it takes advantage of fast MM of TCUs. Second, by utilizing TCUs that would otherwise be idle, we can use the normal processing elements for non-canonical workloads. To that end, we form the following secondary objectives:

- Implement an SpGEMM implementation that works with rectangular blocks. Blocks are necessary to utilize TCUs which work with square matrices.
- Test the performance of SpGEMM. In order to achieve this, we need to:
  - Collect a dataset of sparse matrices.
  - Compare with State-of-the-Art SpGEMM GPU implementations.

3.3 Background

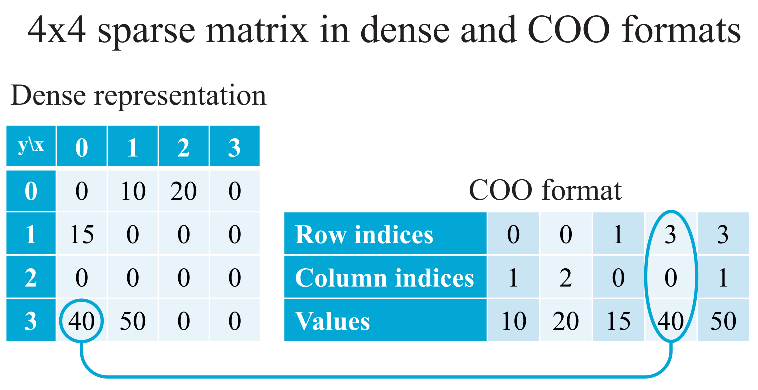
In this section we present the theoretical background of our work.

3.3.1 *Storage format*

In sparse matrices, typically, the number of non-zero (nnz) elements is much smaller than the number of zero elements. In order to save memory, we need an efficient way to store only non-zero (nz) elements.

COO format

COO format stores each nz value along with its coordinates. If the sparse matrix is sufficiently sparse, the storage cost of saving the coordinates is much smaller than the dense representation. This storage format is formally known as COO format (Coordinate Format), one of the simplest and most used storage formats [9]. In COO format, we have three arrays, 1) for row indices, 2) for column indices, 3) for values. The same position in all three matrices corresponds to the same element. In Figure 3.1, we show an the same 4×4 matrix in dense representation and in COO format.



4x4 sparse matrix in dense and COO formats

Dense representation

y\x	0	1	2	3
0	0	10	20	0
1	15	0	0	0
2	0	0	0	0
3	40	50	0	0

COO format

Row indices	0	0	1	3	3
Column indices	1	2	0	0	1
Values	10	20	15	40	50

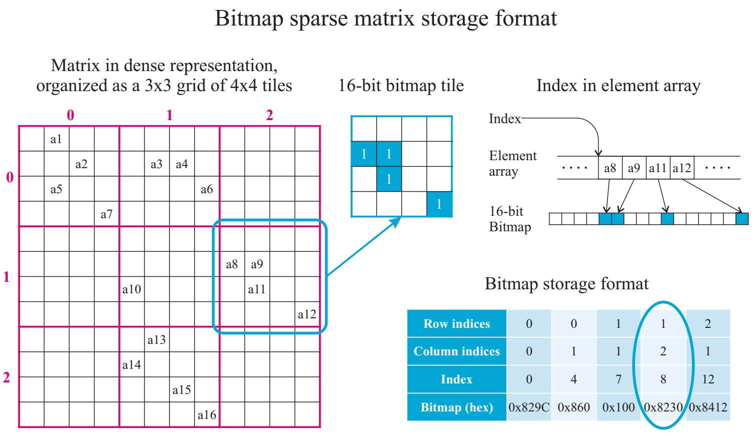
Figure 3.1
A 4×4 matrix in dense and COO formats. Element “40”, which is located at position [3, 0] of the dense matrix, corresponds to [3, 0, 40] in COO format

Bitmap format

TCUs simultaneously process multiple elements in rectangular structures. COO stores only single elements and has no concept of rectangular structures, therefore it is not sufficient by itself as a storage format for our approach. In this work, we use a bitmap-based block shaped storage format to store sparse matrices. The basic idea is to group elements to 8×8 square blocks, which we call *tiles*. Elements have the same placement in the tile as they have in the dense representation of the matrix. Each element in the tile can be either zero or non-zero (nz). To keep track which elements are nz we use a bitmap, a binary number of which each slot corresponds to one slot of the tile. If a slot contains a nz we set the respective bit of the bitmap to “1”, otherwise to “0”. Then, tiles are stored in COO format. The difference with the standard COO format is that, instead of using elements as values, now we use a tuple of two values: 1) an *index* to the *element array*, which holds the elements of the tile (elements of the same tile are in consecutive positions of the array), and 2) the *bitmap* of the tile.

The authors of [7, 51, 124] propose various blocking storage formats, whereas the authors of [56, 59, 66] propose various bitmap formats. In our work, we use a format similar to [123] for three reasons: 1) it is simple and straightforward, 2) square tiles of fixed size fit well to TCUs, and 3) the performance of the format has been evaluated in [123].

Figure 3.2 shows how we convert a dense matrix to bitmap format. We partition the dense matrix in partitions of tile size. We represent the positions of nz elements as “1”s in the bitmap. We store four values for each tile that has at least one nz element: 1,2) row and column indices like in COO format, 3) index into the element array, and 4) bitmap with the location of nz elements in the tile.



Row indices	0	0	1	1	2
Column indices	0	1	1	2	1
Index	0	4	7	8	12
Bitmap (hex)	0x829C	0x860	0x100	0x8230	0x8412

Figure 3.2
A 12 × 12 matrix in dense (left) and bitmap formats (bottom right). Tiles have a size of 4 × 4 and partition the 12 × 12 matrix in a 3 × 3 grid of tiles. Nonzero elements $a8, a9, a11, a12$ of the circled tile are represented as “1” in the *bitmap*. We store the nz elements of the tile in consecutive locations in the element array. *Index* points to the first element of the tile. On the bottom right of the figure, we circle the representation of the selected tile in bitmap format

3.3.2 *Sparse matrix-matrix multiplication*

The general matrix multiplication (GEMM) has the form:

$$D = A * B + C \tag{3.1}$$

where A, B, C are the input matrices and D is the output. Figure 3.3 shows how we perform the matrix-matrix multiplication of the sparse matrices A and B with dimensions $M \times K$ and $K \times N$ respectively. Similarly to dense matrices, to get one element of the output, we need to multiply the nz elements of one row of A with the corresponding nz elements of one column of B and then accumulate the intermediate products (inner product). The difference in spGEMM

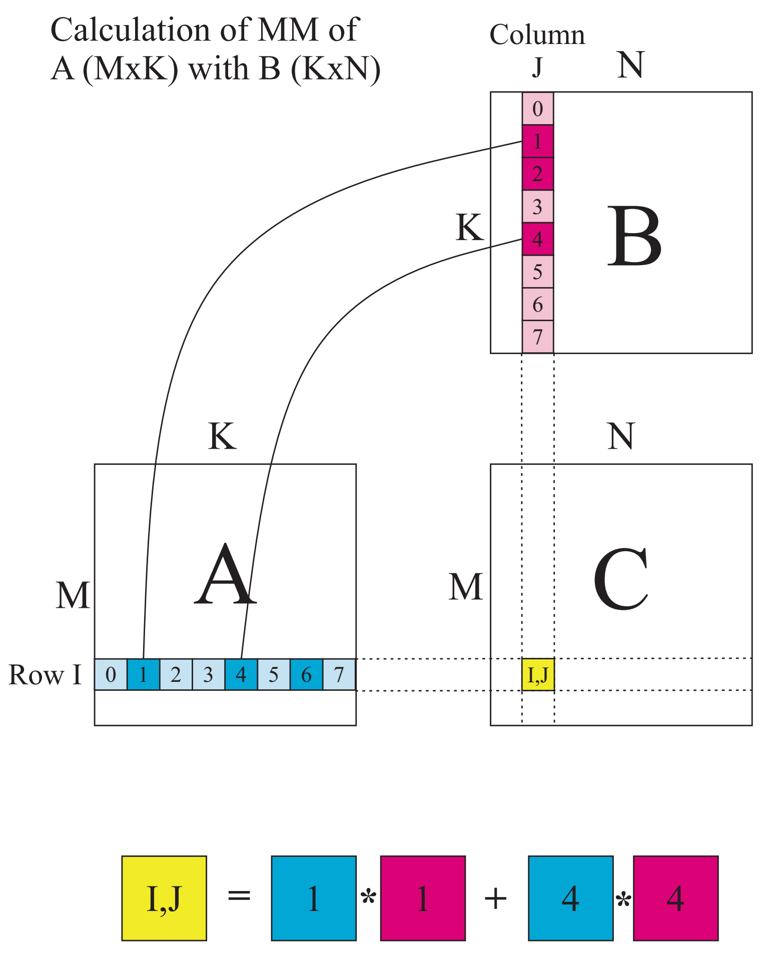


Figure 3.3
Sparse matrix multiplication

is that we multiply the corresponding elements only if the elements in the corresponding positions of the row of A and the column of B are both nz and we accumulate only nz products. The various ways to access the elements of A and B are listed in [71]. Figure 3.3 applies even if instead of elements we use tiles. In this case the product of two corresponding tiles is their outer product (or equally matrix multiplication). Algorithm 1 shows how to obtain the tile of the output which has coordinates [I, J] in a simple spGEMM formulation.

We make two important observations. First, Equation 3.1 takes the form

$$C = A * B + C \tag{3.2}$$

when we accumulate the tiles. Second, the matrix Multiplication-Accumulation (MAC) of small tiles is exactly what the TCUs were designed to do.

3.3.3 *CUSP*

CUSP [20] is a library that specializes in sparse matrix operations. It is open-source and easily accessible on github. It is written in Thrust which makes it

Algorithm 1 Calculate [I, J] tile of C

```
$C\_tile \leftarrow 0$
$i \leftarrow 0$
**while** $i < K$ **do**
    **if** $A\_tile[I, i] \neq empty$ AND $B\_tile[i, J] \neq empty$ **then**
        $C\_tile[I, J] \leftarrow A\_tile[I, i] * B\_tile[i, J] + C\_tile[I, J]$
    **end if**
    $i \leftarrow i + 1$
**end while**
```

easy to read and port to other platforms. Therefore, it provides a good "boilerplate" to test our approach.

CUSP uses Expand-Sort-Compress (ESC) method. According to ESC, there are three main steps in spGEMM [11, 20]. First, Expand. We multiply each nz of a row $A(i,:)$ with all nz of the corresponding row of B (no additions i this stage) [41, 71]. Second, Sort. We sort the products of the previous step so that products that correspond to the same value of C are in consecutive positions. Third, Compress. We calculate each value of C by summing all respective products, which are in consecutive positions, thanks to the previous step.

3.3.4 *Real numbers in digital computer systems*

Computer systems have to store real numbers in bit representation. Floating point numbers are a common representation. The location of the decimal point and the number of bits determines the precision and range of the represented numbers. We denote the 32-bit representation as fp32, whereas the 16-bit as fp16. In contemporary systems, typically, we use floating point numbers as defined in IEEE 754 technical standard [17]. Usually, the fewer the number of bits, the faster the processing of the numbers is. The arithmetic range of fp16 is approximately $6 \times 10^{-8} \ldots 6.55 \times 10^{4}$, whereas for fp32 is $1.4 \times 10^{-45} \ldots 3.4 \times 10^{38}$. Luszczek et al. in [69] show the viability of fp16 arithmetic. They use 16-bit LU (lower-upper) decomposition and iterative precision refinement with mixed 16/64-bit precision to solve an $A \cdot x = b$ system of linear equations.

3.3.5 *Matrix multiplication with CUDA*

NVIDIA, with the latest generation of Graphical Processing Units (GPUs) (Turing architecture [82]), brought Tensor Cores to the mainstream market. Nvidia Tensor Cores or Tensor Core Units (TCUs), are ASICs that have the purpose of accelerating MM. Therefore, our work on spGEMM has significant benefits by properly adapting spGEMM to TCUs. CUDA SDK from NVIDIA provides the necessary Application Programming Interface (API) for TCUs.

Tensor Core Units

TCUs execute MM on the data we provide. The input matrices have two restrictions. First, a matrix can only contain elements of specific types, i.e., one of the following: fp16, 8-bit integers, 4-bit integers and bits. Second, TCUs support only specific configurations of matrix dimensions, which the programming guide defines [83]. We prefer fp16, which has the most bits, because sparse matrices are usually large and physical problems are sensitive to arithmetic precision. We prefer 16×16 matrices because one TCU can fit two 8×8 tiles, as we describe in Section 3.6.4. The programming structure that holds the matrices that TCUs operate on is called fragment (there is no relation with fragments shaders from computer graphics) [85].

TCUs are mainly targeted to deep learning, which is not very demanding precision-wise. Therefore, manufacturers opt for smaller representations (16-bit or less) of numbers to get faster output. fp16 or lower precision is detrimental to the output when dealing with physical problems because precision and range of numbers can be insufficient. To rectify this problem NVIDIA provides mixed precision functionality. Mixed precision allows TCUs to work with numbers of different precision. The defining characteristics of the mixed precision implementation of NVIDIA are two. First, although inputs A and B are in fp16 precision, their multiplication happens in full precision. Second, the product is stored as fp32 to accumulators C and D [85]. Figure 3.4 shows how the different precisions are mixed during MM.

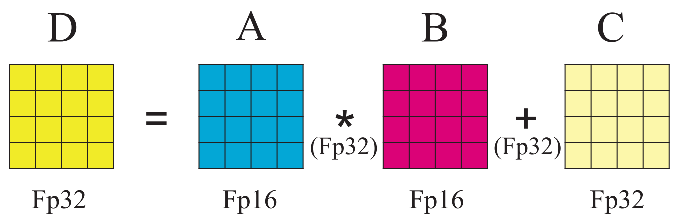


Figure 3.4
Mixed precision with CUDA TCUs. Inputs are stored in fp16, whereas the output and addend are stored in fp32. The multiplication and addition are performed in full precision

Mixed precision fits well to our spGEMM approach. spGEMM requires the calculation of many products, therefore simple fp16 precision is not sufficient for two reasons. First, a product which has a lot of multiplicands in fp16 precision can easily exceed the range of fp16. Second, the precision error accumulates with each successive multiplication. [42, 70] evaluate the performance and precision of GEMM and linear equation solving using the mixed precision mode of TCUs. They show that TCUs can be used in other physical problems, outside deep learning.

3.3.6 *Challenges of spGEMM*

Unlike spGEMV, in spGEMM both inputs and output are sparse. Therefore, it is very difficult to utilize the knowledge we infer from the sparsity structure of the input matrices to make arithmetic and memory optimizations. The reasons that make spGEMM more challenging than spGEMV are three [21, 39, 40, 67, 117]:

First, data access is highly irregular because it depends on the sparsity structure of both matrices and their interaction. Mapping the multiplication of elements of A with elements of B and the accumulation to an element of C is not trivial for two reasons: 1) it requires access of possibly distant memory locations to load the inputs, and 2) it requires inserting data to the output with irregular access patterns. We try to access data row-wise to increase cache locality.

Second, it is very difficult or impossible to know the size of the output before the actual calculation. According to Liu et al. [67], there are four methods to estimate how much memory we need to allocate for the output:

1. Precise. We make an estimate that is very close the actual size of the output, typically by partial execution of the algorithm.

2. Upper bound method. Usually, as upper bound we use the amount of intermediate products (before we accumulate them to their respective values of the output).

3. By using probabilities theory [4].

4. By increasing the size of memory progressively, i.e., allocating more memory if the previously allocated overflows. At the end of spGEMM, we remove empty / unused entries from the allocated memory as necessary.

Third, load balancing. The sparsity structure of both matrices and their interaction determine the distribution of workload. Nz elements in each row may vary significantly, which makes it difficult to partition the workload among threads. We partition workload based on the row size of the input or estimated row size of the output.

3.4 Related work

In this section we review the state-of-the-art GPU spGEMM approaches.

3.4.1 *Expansion Sorting Compression*

Expansion Sorting Compression methodology [11] has three benefits: 1) it exposes fine-grained parallelism, 2) by implementing the algorithm with simple parallel primitives it lessens the effects of load imbalance, and 3) the performance is predictable and the computational complexity is equal to the size of the intermediate matrix. One drawback of ESC is that the intermediate matrix may be significantly larger than the output. Sorting the intermediate matrix is very costly.

Dalton et al. [21] improve ESC. They perform the Expansion step using breadth-first search on a bipartite graph to avoid the deficiencies of loading disparate rows of various sizes from the input. They create thread and warp variants for sorting, the most time consuming step, and use a highly optimized thread block-level radix sort. By using shared memory to localize processing they further increase the performance. However, they did not release these changes to CUSP [20].

Kunchum et al. [60], in HybridSparse, implement variants of ESC in combination with their own scatter vector approach, as we detail in the following section about hybrid spGEMM approaches.

Winter et al. [117], in AC-SpGEMM, perform ESC locally in shared memory. They use dynamic scheduling to keep data longer in shared memory, for many iterations of ESC. Thus they reduce global memory traffic and the cost of sorting a huge intermediate matrix in global memory. One drawback of this approach is that merging partial results degrades the performance when the nz elements of rows of A increase.

In this work we propose a tiling approach. The intermediate matrix has fewer entries because we are using tiles as values of ESC. The cost of ESC is smaller with fewer entries. Moreover, in order to fully utilize MAC operations of TCUs, we have to merge Expansion and Compression steps. This can be achieved by using a task list.

3.4.2 *Hash tables*

Intermediate products that correspond to the same value of the output can be accumulated using hash tables. Hash tables have two advantages. First, hash tables typically require less memory than the intermediate matrix of the ESC approach. Second, hash tables do not use an intermediate matrix. Therefore, they do not have to sort the huge matrix of the intermediate products (they sort the output instead, which is always smaller). Hash tables have two drawbacks. First, efficient usage of shared memory is necessary to keep global memory traffic low. Typically, hash table approaches use two levels of hash tables, one on on-chip memory and another one on off-chip memory. Second, due to their non-deterministic nature, the floating point result may be different between different runs.

Demouth et al. [24] present one of the first implementations of spGEMM with hash tables. In their implementation, each warp loads a row of A. For each nz column of the loaded row of A, the warp loads the respective row of B. The active nz element of A, which is shared to the rest of the warp by shared memory, is multiplied with all elements of the loaded row of B. The product is put to a warp-local memory area using a hash function. Then they repeat with the next nz column of A. cuSPARSE, which is based on this implementation, is widely available through the CUDA Software Development Kit (SDK) [84]. cuSPARSE has been updated since Demouth et al. presented it, but cuSPARSE is delivered as closed-source binary and we cannot know the changes. cuSPARSE has two drawbacks. First, there is imbalance between threads because different threads of the warp may have to insert a different number of values in the hash table. Second, shared memory is small, which results in frequent data movement to global memory.

Anh et al. [6], in BalancedHash, first calculate the amount of intermediate products corresponding to each row of A. They call this amount *partition*. They assign one thread block to each partition, and each partition has its own hash table. Hash conflicts are sent to a spillover queue, where they are kept until the next round. Partitions keep the hash tables small, so that they can fit in shared memory. The hash table size, which is static, is critical to performance. Therefore BalancedHash may not behave well with highly varying partition sizes.

Nagasaka et al. [76], in Nsparse, solve the small shared memory problem by partitioning the rows of the output. Nsparse improves on BalancedHash in two ways: 1) by using hash tables of variable size in shared memory, less shared memory is required and more blocks can run, and 2) by using fewer auxiliary matrices, it keeps memory traffic low and reduces memory storage requirements. As all hash table approaches, Nsparse is sensitive to larger rows that cause hash conflicts or relocation to global memory.

Deveci et al. [25] use two level hash tables. They create a portable implementation that works on many platforms (GPU, Power8, Knights Landing CPUs etc). They adapt the hash tables to the number of threads of each platform. High portability prevents architecture specific optimizations that could give an additional boost in performance.

In our tiling approach, the storage requirements of the intermediate matrix are also low for two reasons: 1) we operate on tiles and not single elements, and 2) we use precise methods to estimate the size of the output matrix. By keeping the size of the intermediate matrix small, the sorting cost is low as well.

3.4.3 *Hybrid*

Depending on the thread assignment / workload partitioning of each approach, there are up to three reasons for load imbalance: 1) the number of values in each row of A may vary substantially among rows, 2) the number of values in each row of B may vary substantially among rows, and 3) the number of intermediate products for each value of C may vary substantially. Hybrid methods use an analysis phase to determine the number of values in each row of A or C (in case of C we use an estimate because we do not know the actual size until the algorithm completes). Then, hybrid methods select an appropriate method / kernel depending on row size.

Dalton et al. [21] improve the load balancing of ESC methodology [11] by permuting rows of C based on their size. They use this permutation to select the most appropriate sorting method based on row size of C. This way they replace the sorting of million intermediate elements with the sorting of many groups of fewer elements. Therefore, they can select between warp- and block-wide methods for sorting, as well as use shared memory for sorting the shorter rows. Nevertheless, permutation of rows has an additional overhead due to two additional sorting operations: one during pre-processing, and one before storing the final result.

Liu et al. [67], in bhSparse, implement a four stage approach. First, they calculate the upper bound limit of nz elements for each row of C. Second, they arrange the rows of C to 38 bins. They further organize the bins to 5 groups. Based on the bin size, they select the most appropriate memory pre-allocation scheme for the output C. Moreover, based on the bins they select which kernels to launch (warp-wide, block-wide etc). Third, they compute the result using customized merging and sorting routines based on the bin groups. Fourth, they allocate memory for the final output and move the result to the final *C* matrix. Thanks to bhSparse's multi-bin strategy, they achieve good load-balancing with matrices that have varied row sizes. Nevertheless, bhSparse may not be as fast as specialized spGEMM implementations when matrices have short rows.

Kunchum et al. [60], in HybridSparse, evaluate the performance of various spMM implementations using synthetic matrices to vary the workload of each row of A. They use the results of this analysis to built an algorithm that bins rows based on row workload of *A* and selects the appropriate spMM implementation and memory scheme for each bin. For smaller workloads they prefer ESC variants. For larger rows with many nz they propose their own implementation, which is based on scatter vector method from CPU. Their approach achieves good results when the floating point operations required for each element of C are high, i.e., when their own GPU scatter vector implementation is called. Otherwise, the ESC variants do not perform much better than the other ESC approaches.

Gremse et al. [39], in RMerge, assign a subwarp to each whole row of *A* with one thread of the subwarp for each element of this row of *A*. The process is similar to the vector-matrix multiplication with matrix *B* for the correspondent (whole) row of *C*, i.e., each thread is assigned to one row of *B*. Then, warp shuffle reduction operations [83] are used to find the minimum index (of the *B* rows) of the subwarp and all values belonging to this index are accumulated with another shuffle reduction. If the elements per row of *A* are more than subwarp size, then intermediate representations of *A* are created so that each row has subwarp size elements. Later, in RMerge2 [40], they expanded RMerge. First, they assign more rows of *B* to each thread. Second, they sort and group rows of *A* for load balancing purposes, with different kernels serving the various cases. Additionally, block-based row merging was implemented to perform block wide reductions through shared memory. Third, they use concurrent kernel execution, as the kernels for the various cases could block the GPU if launched one by one, underutilizing it. With their updated scheme, they better balance the workload and reduce significantly the global memory accesses. However, their approach seems to perform better when rows have fewer than 32 elements (the size of a CUDA warp)

In our approach we will use: 1) Thrust library for the parallel primitives, and 2) custom GPU kernels for the kernels that do MM with TCUs. Thrust achieves good load balance. To achieve load balancing with our custom kernels, we take into consideration that we will have more blocks than resources available. Therefore, we can let the GPU hardware scheduler load new blocks transparently, depending on the available resources of each SM.

3.5 Overview of our technique

The key idea is to apply the Expand Sort Compress (ESC) methodology of CUSP [20] to tiles instead of single values in order to create a task list, which delivers tiles to TCUs. Our approach works in two steps. First, we follow the workflow of CUSP, except the value is a tuple of values instead of single elements. Using the approach of CUSP, we determine which tiles of A need to be multiplied with which tiles of B in order to get one tile of C. Second, we use the task list from the previous step to deliver the tiles to TCUs. We will show how all parts of spGEMM blend together in Section 3.6.5.

3.5.1 *Key insights*

Our work is motivated by four observations.

First, Dakkak et al. in [19] observe that full utilization of TCUs is not necessary in order to achieve performance benefits. In fact, the high performance of TCUs compensates for wasted resources and memory operations. In spGEMM, most of the entries of the multiplicand matrices are zero, which in turn leads to very sparse tiles. Therefore, the high performance of TCUs compensates for the relatively empty tiles.

Second, Zhang et al. in [123] group nz in 8x8 tiles. Tiles are a type of rectangular matrix, and therefore they are a good target for TCUs.

Third, Haidar et al. and Markidis et al. in [42, 70], respectively, show the benefits of mixed precision in GEMM computation. Similarly, we use mixed precision mode of CUDA TCUs for two reasons: 1) to perform the multiplication of elements in full precision, and 2) keep the intermediate result of accumulation in 32-bit format. Thus, mixed precision increases the application field of TCUs to include: 1) matrices with a wide range of values (fp16 has tight limits), and 2) applications where accuracy is critical.

Fourth, TCUs are specialized multiplication units that perform mixed precision multiplication with $4\times$ the floating point operations per second (flops) of standard fp32 multiplication [82]. spGEMM requires a great amount of multiplications.

3.5.2 *Components*

We modify CUSP to work with a tuple of values instead of single values, where each tuple consists of index in element array and bitmap of the tile. We adapt CUSP to create a task list. Custom GPU kernels consume the entries of the list and accelerate matrix multiplication using TCUs.

Creating the task list and allocating memory for tiles

We use a methodology similar to that of CUSP to accomplish two tasks. First, to determine which tiles of A will be multiplied with which tiles of B in order to get each tile of C. The pairs of tiles of A and B are stored in a task list. Second, to estimate the number of C tiles in the output.

There are two major differences in comparison to ESC from CUSP. First, instead of using a single element as value, we use a tile as value, where each tile is a tuple of the index in the element array and the bitmap. Second, instead of directly multiplying corresponding tiles of A and B in the expansion phase, we create a task list with the pairs. The values of the pair are pointers to tiles of A and B.

Counting kernel

A limitation of GPU kernels is that they cannot reallocate memory during their execution. Therefore, we have to allocate the memory for storing the output before calling the kernel. However, an important problem when multiplying sparse matrices is that we do not know beforehand how many elements the output will have. In order to proceed with the multiplication, first we have to make an estimate of the *count* of elements of the output. We emphasize that the memory we allocate for the elements is different to the memory we allocate for tiles.

In our implementation we use the counting kernel to get an estimate of how many elements the result has. The counting kernel is a partial implementation of the multiplication kernel, that neither loads nor stores any elements. Instead, this kernel uses only zeros and ones as elements according to the bitmap in order to simulate the MAC operations. The estimation of memory requirements is typically more that what is actually required, so after the actual multiplication, we shrink the allocated array accordingly.

Multiplication kernel

Once we know how much memory to allocate for tiles and elements we use the multiplication kernel for the main spGEMM operation.

Putting everything together

In order to perform the matrix multiplication of A with B, we need to 1) determine which products need to be accumulated for each tile of C, and 2) to allocate memory for the C tiles and the element array. Using CUSP and the counting kernel we determine the memory allocation size for C tiles and the element array, respectively. Subsequently, the multiplication kernel has everything it needs to multiply A with B.

We expect three benefits. First, by placing both the multiplication and accumulation steps of matrix multiplication in the same kernel we can use TCUs for MM. By moving MM to TCUs, the computational heavy MM is no longer a bottleneck of the spGEMM algorithm. Second, the use of bitmap format reduces memory consumption [123]. Third, by grouping elements to tiles, we have less values to manipulate and therefore there are additional performance benefits (e.g., less values to sort during sorting phase of ESC).

3.6 Our technique in-depth

3.6.1 *Creating the task list and allocating memory for tiles*

Our implementation is similar to CUSP. It uses parallel primitives from Thrust library to find the correspondence among values of A and values of B. One important difference in comparison to CUSP is that instead of using elements as the values of the input/output COO matrices, we use tuples. Each tuple consists of two values. The index in element array and the bitmap that correspond to each tile. The two main parts of our algorithm are: 1) the part that determines which tiles of A will be multiplied with which tiles of B and creates a task list, and 2) the part that estimates how much memory to allocate for the tiles of the output.

The first part uses CUSP methodology with tiles as values to find corresponding tiles of A and B. One important consideration of our approach is that TCUs perform MAC in the same operation. Therefore, in order to have TCUs perform both multiply and accumulation of tiles in the same operation, we need to further modify the ESC implementation of CUSP. The method from CUSP is not sufficient because it multiplies and accumulates in different steps, i.e., the accumulation step is after the sorting step. In detail the steps in CUSP are three: 1) multiplication of all corresponding values of A and B (but no accumulation yet) to get the intermediate products (Expand), 2) sorting of the products of the previous step so that products that correspond to the same values of C are in consecutive positions (Sort), and 3) accumulation of the products that correspond to each value of C (Compress). In order to put multiplication and accumulation in the same step we sort the locations of the multiplicands, instead of sorting the intermediate products. Effectively, we create a task list, of which each entry holds the pointers to the corresponding tiles of A and B. Then we sort the task list instead of the intermediate products, deferring the multiplication step until the accumulation step.

The second part counts how many of the intermediate products correspond to the same tile of C using a segmented reduce parallel primitive. We create an offset array from the prefix sum of the counted intermediate products.

3.6.2 *Counting kernel*

An important observation is that because tiles of A and B are sparse, the result of their MM will not necessarily have any nz values. The purpose of the counting kernel is to find if there are any nz elements of A that will be multiplied with nz elements of B and consequently what elements of C will become nz (i.e., not accounting for cancellation because of addition of opposite numbers). The counting kernel makes an estimation of the size of memory we need to allocate in order to store the element array of C. The counting kernel has functionality which is very similar to multiplication kernel. The two main differences of the counting kernel are: 1) it does not load the elements to multiply them, and 2) consequently, it does not store any data to the output. The counting kernel returns an array of which each value holds an estimation of how many elements each tile of C has.

The counting kernel works in four steps. First, it reads the bitmaps of A and B. Second, it creates tiles, wherein each value is set to "1" or "0" based on the corresponding position in the bitmap. Third, it multiplies and accumulates the tiles of A and B that correspond to each individual tile of C using TCUs. Fourth, we count how many values of the resulting tile are not zero. To count them, threads of the same warp retrieve values from the fragment of the result. Then threads share the retrieved values that are greater than zero with the `ballot` instruction. Finally, we count how many values are greater than zero.

Our method of estimating the required memory size is the precise one (Section 3.3.6). As each tile stores only "1" or "0" and we only check the output for zero and nonzero values, half precision is enough for executing the MM multiplication.

3.6.3 *Multiplication kernel*

The multiplication kernel performs the actual multiplication and constructs the COO matrix of the output, i.e., it sets the row and column indices, the idx and bitmap tuple and the elements of the element array. Using the memory allocated by the counting kernel to store the elements of C, the multiplication kernel loads the actual elements from A and B.

There are two important considerations when multiplying the elements, that are real numbers.

First, fp16 arithmetic has a very limited representation range of numbers (approximately $6 \times 10^{-8} \ldots 6.55 \times 10^{4}$), which we can easily exceed with multiplication. Therefore, we prefer the mixed precision functionality of TCUs.

Second, unlike the counting kernel where we have only positive numbers, when accumulating real numbers, elements get canceled as a result of addition of opposite numbers. Many tiles may end up empty, something that counting kernel, which acts on boolean values, does not predict. For this reason, our multiplication kernel has the additional task of marking for removal tiles that are completely empty.

Figure 3.5 shows that the counting kernel does not need to load any actual element. It just creates "1"s based on the bitmap. The multiplication kernel, on the other hand, loads the elements and it places them according to the bitmap.

3.6.4 *Other components*

ARRANGEMENT OF TILES IN THE TCUS TCUs execute MM on 256 elements at a time. TCUs only support specific matrix sizes, of which we prefer the square 16×16 configuration [83]. Nevertheless, our tiles have a size of 8×8, which means that a large part of the TCU remains unused. Although a TCU does not have to be fully loaded in order to get performance benefits (Section 3.5.1), we can fit two tiles in a single TCU. To put two tiles in the same TCU two steps are necessary. First, we initialize the fragment to zero. Second, the tiles must be placed in the same diagonal of the fragment. If the tiles were not

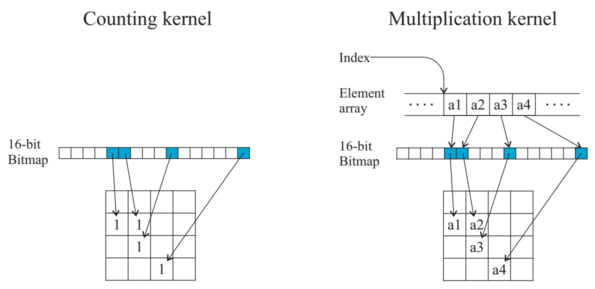


Figure 3.5
Comparison of counting and multiplication kernels. The counting kernel (left) places "1"s at the locations indicated by the bitmap. The multiplication kernel (right) loads the actual elements and places them at the locations indicated by the bitmap

in the diagonal, but instead side-by-side, a row (column) would have elements of two unrelated tiles, which would spoil the inner product of rows of A with columns of B. Figure 3.6 show the placement of tiles in the fragment during MM. Of all supported configurations, only the square 16×16 can fit the two square 8×8 tiles.

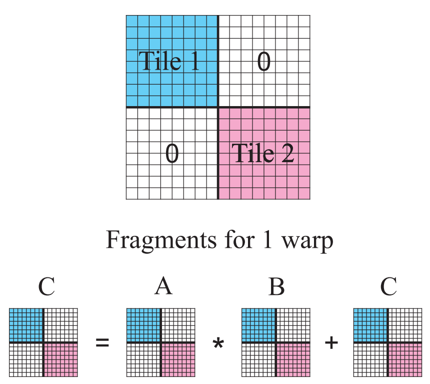


Figure 3.6
(Up) Placement of the two tiles in the fragment, (Down) Tile placement during multiplication

LOAD BALANCING Both counting and multiplication kernels calculate an inner product of tiles. The number of intermediate required for each tile of C is different, because the number depends on the sparsity structure of A and B. Therefore the execution time for the calculation of each tile of C is different. There are two types of load imbalance.

First, imbalance among thread blocks. Blocks with different execution times create load balancing issues among the SMs of the GPU. To tackle this issue, we assign (pairs of) tiles of C to different blocks. When a block finishes and releases the resources, the scheduler of the SM schedules another block to take its place.

Second, inside the TCU. Each TCU multiplies two independent tiles. The two tiles may require the accumulation of a different amount of intermediate products, which leads to load balancing issues. The second type of imbalance is not as important as the first for two reasons: 1) because TCUs are efficient even if they are fully utilized, and 2) the current bottleneck in our implementations is memory latency.

Sorting the tiles by number of intermediate products fixes load balancing for both types of imbalance. Nevertheless, sorting sacrifices data locality, which in our current implementation is more important performance-wise.

COMPACTION OF ZEROS Multiplication and accumulation (MAC) creates zero elements because of cancellation (addition of opposite numbers). To store the output in a strictly sparse format we need to remove all zeros. Therefore, we need a way to detect zeros and remove them or, in other words, *compact* the arrays that hold the elements and the tiles. For the element array, which is just an array, we use a compaction parallel primitive from Thrust. For the array that holds the tiles, we first mark empty tiles during the multiplication phase (see Section 3.6).

3.6.5 *Putting everything together*

Algorithm 2 shows the steps of our approach:

1. We create the task list (lines 1-7).

2. We allocate memory for tuples (lines 8-14).

3. We estimate the count of elements in the output, C, and allocate the respective memory (lines 15-16).

4. We perform the main multiplication (line 17).

5. We remove zero elements and tiles from the output (lines 18 and 19).

Algorithm 2 Pseudocode for tSparse

```
1: for all nnz tiles $A[i,j]$ in $A[:,:]$ do
2:     for all nnz tiles $B[j,k]$ in $B[j,:]$ do
3:         $task\_list \leftarrow \{row\_ptr(A[i,j]), col\_ptr(B[j,k])\}$
4:     end for
5: end for
6: SortByKey($B_{cols}[task\_list], task\_list$)
7: SortByKey($A_{rows}[task\_list], task\_list$)
8: $tile\_count \leftarrow 0$
9: for all $c$ in $task\_list$ do
10:     if $C[A_{rows}[c], B_{cols}[c]]$ is unique then
11:         $tile\_count \leftarrow tile\_count + 1$
12:     end if
13: end for
14: AllocateMemGPU($tile\_count$)
15: $element\_count \leftarrow$ CountingKernel($task\_list$)
16: AllocateMemGPU($element\_count$)
17: $C_{tiles}, C_{elements} \leftarrow$ MultiplicationKernel($task\_list$)
18: CompactZeroElements($C_{elements}$)
19: CompactEmptyTiles($C_{tiles}$)
```

Figure 3.7 shows details of the multiplication kernel and its connection to the task list. For each tile of the the output we have to accumulate a varied number of products of MMs. For example tile $C0 = A0_0 \times B0_0 + A0_1 \times B0_1$ and tile $C1 = A1_0 \times B1_0$. As the amount of addends for *C1* is less, on the 2nd load to TCUs, we load zeros in its place. Finally, we get the elements of *C* and the bitmaps (using `ballot`).

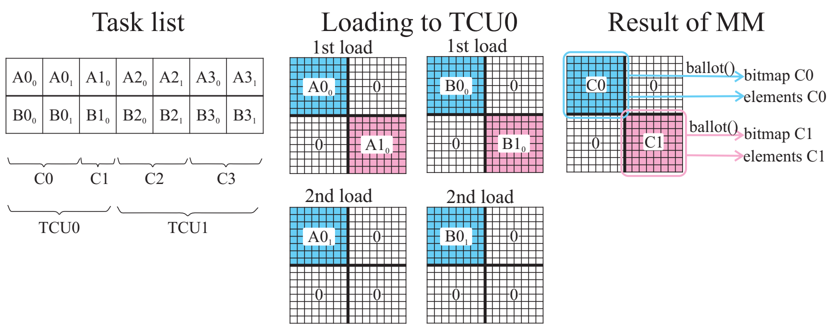


Figure 3.7
The multiplication kernel and its connection to the task list. Each TCU calculates two tiles of the output *C*. If the accumulation of the two output tiles requires a different number of addends, we fill with "0"s as appropriate. As output we get the elements of the two output tiles, *C*, and the respective bitmaps (using ballot operations).

3.7 Evaluation Methodology

We test our approach on a system with an Intel i7-8700 CPU and an NVIDIA RTX 2070 GPU (Turing architecture [82]). We use CUDA SDK v10.1 and the accompanying parallel primitives library, Thrust [84], for our GPU code.

We compare our approach with cuSPARSE from CUDA Toolkit [84] and CUSP [20]. We select the best performing storage format for each approach, i.e, for cuSPARSE we use CSR format, whereas for CUSP we use COO format. In addition, to confirm the benefit of using TCUs, we create one implementation of our approach without TCUs. In this implementation, we use a method similar to [123] to multiply the tiles (Algorithm 3).

Algorithm 3 Matrix multiplication of two 8×8 tiles without TCUs

```
tid: The id of a thread
$i \leftarrow 0$
**while** $i < 8$ **do**
    $C\_tile[tid] = C\_tile[tid] + A\_tile[(tid/8) * dim + i] * B\_tile[i * 8 + mod(tid, 8)]$
**end while**
```

To evaluate the performance of our approach, we perform the A*A MM, which has the benefit that both matrices have the same sparsity structure. We select matrices from SuiteSparse Matrix Collection (formerly known as the University of Florida Sparse Matrix Collection) [23] for our dataset. All selected matrices are square, as our A*A problem dictates. We select matrices which have elements in the fp16 range. A good fit (but not exclusively) to the fp16 range are binary matrices, like certain graphs or structural problems. We divide our dataset into two parts. The first part consists of matrices that other works use

[6, 21, 24, 39, 40, 60, 67, 76, 117, 123]. The second part consists of matrices that we select after taking into consideration the characteristics that define the performance of our approach. Table 3.1 shows the characteristics of our dataset. The upper half corresponds to the first part, whereas the bottom half corresponds to the second part. We denote a matrix stored in bitmap storage format as C_{tiles}. We denote the intermediate matrices as $\overline{C}_{tiles}$ and $\overline{C}$, for bitmap and non-bitmap storage formats respectively.

Table 3.1
Matrix characteristics. We list the size of the matrix (number of rows (columns)), the density of the tiles, and the number of non-zeros of: the input (nnz(A)), the output (nnz(C)), the intermediate matrix (nnz($\overline{C}$), the number of tiles (nnz(C_{tiles})), and the number of tiles of the intermediate matrix (nnz($\overline{C}_{tiles}$)). The upper part corresponds to matrices that are commonly used in the literature, the bottom part to matrices we selected based on their characteristics

Matrix name	Size	nnz(A)	nnz(C)	nnz($\overline{C}$)	nnz(C_{tiles})	nnz($\overline{C}_{tiles}$)	bitmap density (median, mean, std)
mc2depi	525825	2100225	5245952	8391680	718228	1620780	6, 7.3, 6.2
qcd5_4	49152	1916928	10911744	74760192	477184	2004992	16, 22.8, 14.8
rma10	46835	2374001	7900917	156480259	234561	2023575	32, 33.7, 19.8
webbase-1M	1000005	3105536	51111996	69524195	2546355	7540274	8, 20.1, 23
ca-CondMat	23133	186936	2355437	4127524	1787564	13629092	1, 1.31, 0.8
cage12	130228	2032536	15231874	34610826	2945653	15390213	4, 5.2, 4.9
dawson5	51537	1010777	3616737	21284355	219077	2078495	12, 16.5, 14.6
lock1074	1074	51588	134676	2752056	3050	21170	52, 44.2, 20.3
m133-b3	200200	800800	3165861	3203200	1591414	4912984	1, 2, 2.1
p2p-Gnutella31	62586	147892	537601	538318	361039	1501316	1, 1.5, 1.2
patents_main	240547	560943	2281308	2604790	2089143	11688648	1, 1.1, 0.3
wiki-Vote	8297	103689	1831112	4542805	526421	7261770	2, 3.5, 3.3
bcsstk30	28924	2043492	8946070	173481412	252076	1925418	32, 35.5, 20.8
nemeth21	9506	1173746	2578720	146859992	47341	526143	64, 54.5, 17.7
pcrystk03	24696	1751178	7240266	129128312	212471	1876143	31, 34.1, 20.2
pct20stif	52329	2698463	10016951	154237335	323396	2366028	26, 31, 20.5
pkustk06	43164	2571768	10596384	179924544	451380	3641052	16, 23.5, 15.6
pli	22695	1350309	8548665	99698581	292851	2721449	28, 29.2, 19.2
qa8fk	66127	1660579	7351189	42857197	443738	2867240	18, 16.6, 10.9
struct3	53570	1173694	3400384	26704476	146007	583547	29, 23.3, 11.1
web-NotreDame	325729	1497134	16801350	64593748	693759	3216627	8 , 24.2, 25.8

We collect two types of measurements: 1) the absolute times to perform an spGEMM, and 2) speedup. For speedup, we divide each implementation with our approach. We collect all timing data with CUDA event API.

Tests with cuSPARSE, CUSP and no TCU version use fp32 (single precision) for both the input and the output, whereas our TCU approach uses fp16 for the input and fp32 for the output (mixed precision).

3.8 Results and Analysis

In this work, we use a tiling approach to group nz elements. Tiles are suitable for MM with TCUs. To show the performance benefits we compare our approach with other approaches from the state-of-the-art, as well as, with one version of our approach that does not use TCUs. To study the effect of our optimizations, we execute the A*A spGEMM and we measure the absolute times and the speedup of our approach over the other approaches.

3.8.1 *Times*

We measure the execution time of each approach for each matrix in Table 3.1. We compare the timing results to the matrix characteristics in Table 3.1 to make general observations in regards to the performance of each approach. Figures 3.8 and 3.9 show the absolute execution time of the four approaches when multiplying a matrix with itself on an RTX 2070 GPU. We form groups of four bars, one bar for each approach, in order to facilitate the comparison of execution time of spGEMM for each matrix of the dataset. Figure 3.8 corresponds to the first part of our dataset (randomly selected matrices), whereas Figure 3.9 corresponds to the second part of our dataset (matrices selected based on criteria).

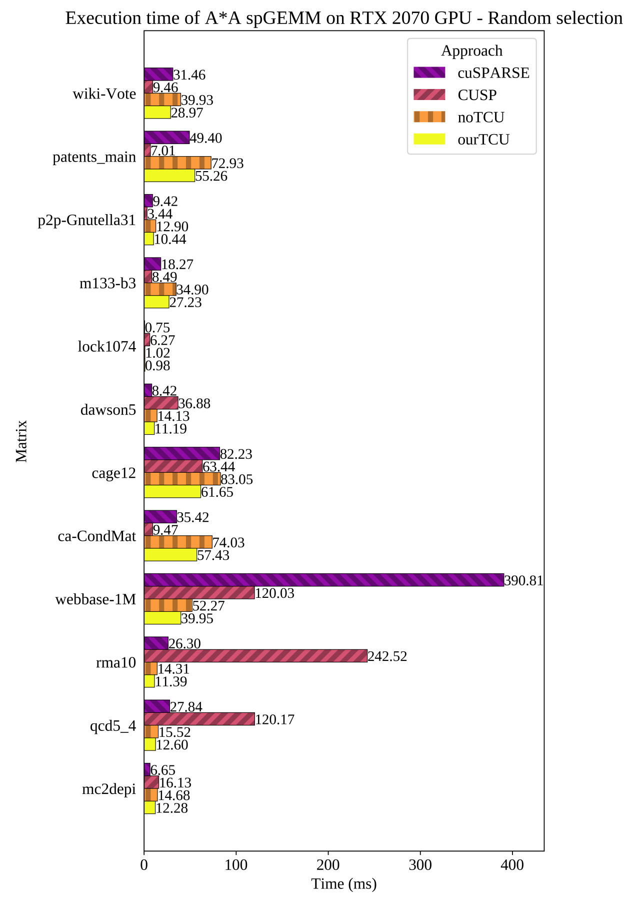


Figure 3.8
Absolute times of our approach on A*A spGEMM using randomly selected matrices

We make three major observations.

First, our approach shows better performance with denser tiles. Grouping nz elements in tiles reduces the number of intermediate products of the expansion

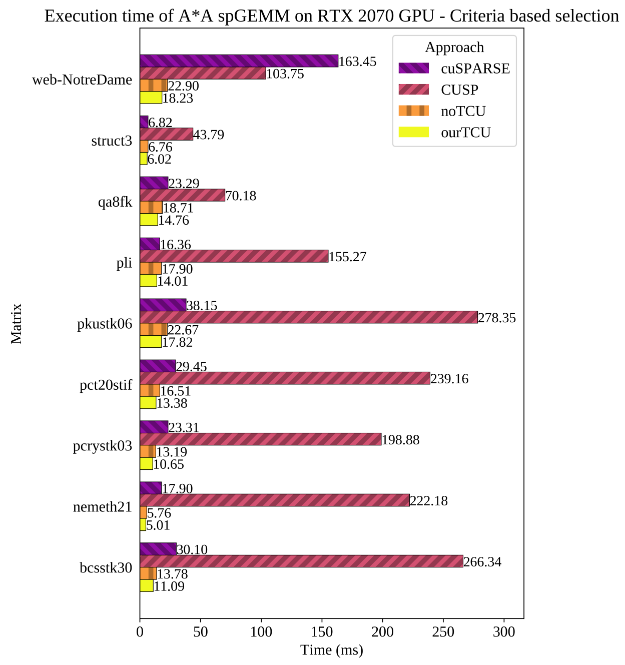


Figure 3.9
Absolute times of our approach on A*A spGEMM using matrices selected based on criteria

phase. According to [21], the main cost of CUSP is sorting. Consequently, fewer entries equals to less time sorting. Generally, our approach performs better for densities greater than five.

Second,our approach performs better than CUSP when the number of intermediate tiles is sufficiently smaller than the number of intermediate elements of the normal CUSP approach, i.e., $\overline{C}_{\text{tiles}} < \overline{C}$. Tiles group elements in a 8×8 area. When the two multiplicand tiles hold only a very small number of elements, it is very likely that when calculating the product all the elements of the tile that holds the product will be zero. The reason is that no elements of the tile of A multiply with elements of the tile of B. If there are many intermediate products like this, the number of intermediate values with our approach will exceed the number of the intermediate values with CUSP approach. In such cases, CUSP, which handles only single elements, is faster, as our approach has an additional overhead for handling tiles.

Third, our approach generally outperforms cuSPARSE in the larger matrices of our dataset ($\sim nnz(A) > 1000000$). This happens probably because shared memory is not sufficient for the hash tables and therefore global memory traffic increases. Nevertheless, the size of the matrix is not the only thing that decides the performance. To the best of our knowledge, performance of cuSPARSE also depends on the sparsity structure of the matrix because the structure also affects the number of hash conflicts.

Based on the observations of this section, we create a dataset of matrices that fit well to our approach (second part of Table 3.1). Specifically, we select matrices that fulfill the following two criteria: 1) $nnz(A) > 1000000$, and 2) $\overline{C}_{\text{tiles}} < \overline{C}$.

3.8.2 *Speedup*

To show the benefits of our approach that uses tiles and TCUs for MM, we find the speedup over cuSPARSE, CUSP and the non-TCU implementation over all matrices of our dataset. Figures 3.10 and 3.11 show the speedup of our approach over the other three approaches when multiplying a matrix with itself on an RTX 2070 GPU. We, again, group bars by matrix, but in this experiment we show the speedup instead. Figure 3.10 corresponds to the first part of our dataset (randomly selected matrices), whereas Figure 3.11 corresponds to the second part of our dataset (matrices selected based on criteria).

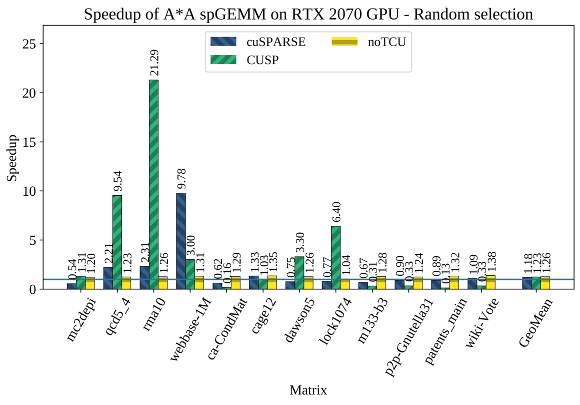


Figure 3.10
Speedup of our approach on A*A spGEMM using randomly selected matrices

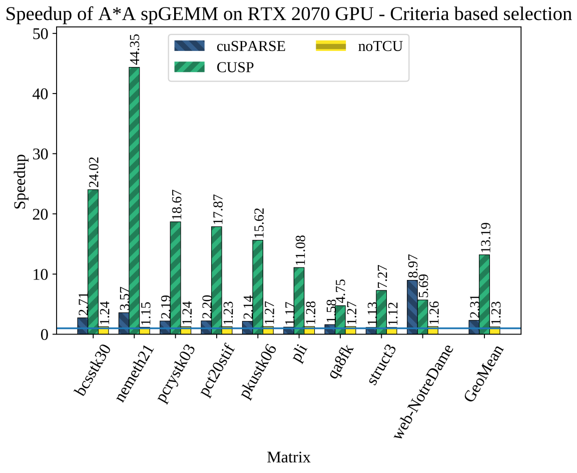


Figure 3.11
Speedup of our approach on A*A spGEMM using matrices selected based on criteria

We make two major observations.

First, our approach that uses TCUs performs 1.25× faster on average. This speedup may seem low considering that TCUs in mixed precision promise 4× more flops than normal fp32 operations. There are two reasons that keep it low: 1) our counting and multiplication kernels, that use TCUs, occupy about 50% of the total execution time, so according to Amdahl's law we do not expect more than 2× speedup, and 2) our approach is memory bound, rather than arithmetic bound, because it loads the input from non-continuous memory locations.

Second, in sixteen and fourteen out of the 21 matrices our approach outperforms CUSP and cuSPARSE, respectively. In comparison to cuSPARSE, CUSP and non-TCU implementation our approach is 1.12×, 1.23×, 1.26× faster in a random selection of matrices. If we select matrices that fit well to our approach, based on the criteria of Section 3.8.1, our approach is 2.31×, 13.19×, 1.23× faster. The total speedups for all the matrices of our dataset are 1.6×, 3.4×, 1.25×.

3.8.3 *Conclusion*

We compare the performance of our approach in A*A, in terms of time and speedup, to CUSP and cuSPARSE over 21 sparse matrices. We draw three important conclusions. First, our approach generally outperforms CUSP with denser tiles. Second, our approach usually performs better than cuSPARSE for the larger matrices of our dataset. Third, TCUs improve performance over non-TCU implementation by 25%.

Our approach outperforms cuSPARSE and CUSP 1.6× and 3.4× respectively. Therefore our approach is a suitable alternative for spGEMM.

3.9 Conclusion

In this work, we present an spGEMM approach which uses a tiling scheme to divide the matrices to blocks of equal size. Tiles are stored as bitmaps. As our spGEMM approach follows a blocking approach, it has to act on fewer values. Therefore, sorting the intermediate matrices of tiles is faster in comparison to sorting millions of intermediate elements in a non-blocking approach.

We perform MM of these tiles using TCUs. To the best of our knowledge, our approach is the first to use TCUs for spGEMM. TCUs perform the MM much faster than normal CUDA cores. Tensor core units perform both matrix multiplication and accumulation. We create a task list of of pairs of tiles of A and B. The task list is responsible for efficiently distributing the workload to thread blocks.

The results confirm that TCUs increase the performance of MM and the combination of our tiling approach with TCUs provides significant benefits to spGEMM. TCUs increase the performance of our approach by 25%. Our approach is, on average, 1.6× and 3.4× faster than state-of-the-art libraries cuSPARSE and CUSP respectively. We conclude that our methodology improves the performance of spGEMM by making efficient use of tiles and TCUs. We publish our work ([121]) and make our source code publicly available at [120].

3.10 Future work

Future work can take two paths, either on the application level or on the spGEMM algorithm level.

3.10.1 *SpGEMM algorithm extensions*

The possible optimizations of the spGEMM algorithm are summarized as:

- Adjust tile size depending on the density of tiles. Determine the density of the tiles by sampling the input. If the tiles do not contain many elements, we can use smaller tiles. Then each TCU can run more tiles simultaneously.

- Attempt to increase the density of the tiles. For example we could rearrange the rows of the input [21] or apply tiling only to areas of the matrix that are dense enough [47].

- Increase data locality. The task list accesses bitmap and the corresponding tiles from disparate memory locations. Matrix storage formats other than COO may help with increasing memory coalescence.

- Optimize sorting of task list. Sorting the task list is one of the most computationally demanding parts of our approach. Possible methods are bipartite graphs [21], hash tables [6, 76] and improved segmented sort algorithms [49].

- Create a hybrid approach that adapts to the input. For example, depending on density of tiles or tiles per row of C we could choose different type of sorting or a different approach all together.

3.10.2 *SpGEMM applications*

Although there are numerous spGEMM application for sparse matrix-matrix multiplication we focus to only a few that are of special interest in this line of work.

System equation solving

A system of equations is usually solved by using iterative methods, e.g., conjugate gradient, as inverting the system matrix is practically impossible for bigger matrices. To attain faster convergence we use preconditioner matrices. One interesting preconditioner is Algebraic Multigrid (AMG). AMGs have a setup phase and a cycle phase. During the setup phase the most time consuming part is a sparse matrix-matrix multiplication called Galerkin product. Galerkin product is a good target for spGEMM [2].

Finite element method (FEM), a method that is used to solve partial differential equations in many engineering problems, can benefit from accelerated spGEMM. We are specifically interested in simulation of biomechanical models of the liver, where FEM is particularly useful. By increasing the performance of simulation we can have more realistic simulation of contacting and cutting a liver at increase resolution [18].

Vessel registration

Vessel registration is an interesting field [99]. However, using deep-learning for vessel registration faces performance problems because using sparse inputs, like vessels, is computationally costly. SpGEMM could help in this case as the convolution of Convolutional Neural Networks (CNNs) is calculated as a matrix-matrix multiplication [16].

Betweenness centrality

Betweenness centrality finds the importance of a vertex based on the number of shortest paths that pass through it. Betweenness centrality is an essential graph operation for analyzing biological networks, transportation networks etc [55, 105]. The multi-source BFS that is used in betweenness centrality can benefit from spGEMM. This accounts for multiplying A with a tall skinny sparse matrix where each column is a BFS frontier.

Part IV

DISCUSSION AND CONCLUSIONS

4 DISCUSSION

Contents

In the recent years great progress has been made towards image guided surgery, enabling safer surgeries and increased survivability rate. GPU image processing plays a critical role at all levels, enabling faster and more accurate image analysis. This chapter gives an overview of the work done as a part of attaining the goal of this thesis.

4.1 Accelerating B-spline interpolation

In this work, we apply registration between pre and intra-operative medical images for both MRI and CT liver scans and we evaluate the performance of B-spline interpolation and its impact on the total registration time using NiftyReg as a registration tool [74]. NiftyReg is a GPU accelerated lightweight medical image registration library, which includes a GPU implementation of B-spline interpolation. Recent works [89] use it as reference for registration.

The key optimizations of our GPU implementation of B-spline interpolation are two. First, we partition the workload carefully. This way, not only we reduce the data movement needs (i.e., need to bring less data from off-chip memory), but also we maximize register reuse. Second, we substitute the weighted sum of the basic formula of B-spline interpolation with linear interpolations. This way, we can use special interpolation instructions, namely Fused Multiply-Add (FMA) instructions [83]. The use of such instructions not only reduces computational complexity, but also increases arithmetic accuracy. Many of our proposed optimizations can also be applied to CPUs.

In order to show how our approach impacts the performance and accuracy of registration in a realistic scenario and to provide clinical validation, we integrate it to the FFD registration of NiftyReg. We use FFD on CT scans of patient-specific liver phantom (artificial liver) and MRI scans of a porcine liver model to compensate for liver deformation due to pneumoperitoneum.

4.2 Accelerating sparse matrix-matrix multiplication: tSparse

In this work, we propose a new methodology of spGEMM that utilizes TCUs to accelerate spGEMM. To the best of our knowledge, this is the first proposal of using TCUs in the context of spGEMM. Our methodology has two advantages. First, it takes advantage of fast MM of TCUs. Second, by utilizing TCUs that would otherwise be idle, we can use the normal processing elements for non-canonical workloads.

We use CUSP [20], an open-source library, as the "boilerplate" of our approach. We make two modifications to ESC methodology [11] of CUSP [20]. First, we make it work with tiles. Second, we form a task list instead of calculating the intermediate products immediately. The tiles and the task list allow our custom kernels to calculate MM of blocks with TCUs.

We compare the performance of our approach in matrix squaring (A^2) on matrices from SuiteSparse (formerly known as University of Florida Sparse Matrix Collection) [23]. We show that by converting ESC method to work with tiles and processing the blocks with TCUs, we perform spGEMM, on average, $1.6\times$ and $3.4\times$ faster than cuSPARSE and CUSP respectively. We show that TCUs improve the performance of the whole spGEMM procedure by 25% on average.

5

CONCLUSIONS

Contents

This thesis covers two topics. The first targets the computationally demanding B-spline interpolation, which is often found as a part of image registration. The second targets sparse matrix-matrix multiplication (spGEMM), a frequent component in linear algebra applications. In both cases, we create efficient GPU implementations that take full advantage of the underlying hardware.

5.1 B-spline interpolation

To our knowledge, our work is the first to use a single GPU thread for an entire tile of voxels in the context of B-spline interpolation. With this method each thread handles a whole tile of voxels instead of a single voxel, which normally occurs with the previously introduced approaches. We show that, with this thread assignment scheme, we reduce memory accesses substantially. Our main contributions are:

- A GPU implementation of B-spline interpolation with three key optimizations: a) new workload partitioning scheme for GPU execution threads that reduces the number of accesses to off-chip memory and caches, b) register-only approach that keeps input data close to the execution units, and c) replacement of weighted summation with linear interpolations that reduces the computational load and increases accuracy.

- A pre-clinical validation of non-rigid registration between pre-operative and intra-operative MRI and DynaCT scans after pneumoperitoneum and repositioning on a porcine model and a patient-specific liver phantom.

- A highly-optimized GPU-based implementation of B-spline interpolation which improves performance by up to $7\times$ and accuracy by up to $3300\times$ in respect to other state-of-the-art B-spline interpolation methods. We integrate the proposed approach to FFD registration algorithm, which improves the total registration time by up to 34%.

- A new dataset for image registration from MRI and CT scans of a porcine model and a liver phantom. This dataset is publicly available [53].

- The source code of our B-spline interpolation, which is publicly available on `https://github.com/oresths/niftyreg_bsi`, as part of the medical image registration library, NiftyReg.

Our acceleration of B-spline interpolation directly improves the execution time and accuracy of image registration as we show by integrating our approach to a popular registration methodology. The benefits of an optimized B-spline interpolation are not limited to image registration. Other fields like image reconstruction, image zooming etc. also benefit from our approach.

5.2 Sparse matrix-matrix multiplication

To our knowledge, our work is the first to use Tensor core Units (TCUs) in the context of spGEMM. With this approach, we group non zero elements into tiles. Unlike previous methods, which use normal CUDA cores, we use tensor cores to multiply the tiles. We show that, with this approach, we can increase the performance of spGEMM. Our main contributions are:

- A fast GPU approach of spGEMM. We modify the Expand-Sort-Compress method to bring both multiplication and accumulation after *Sort*. This change has two advantages. First, we do not have to store in memory a large matrix of intermediate products. Second, we can take full advantage of the combined Multiply-Accumulate operation of TCUs. For this purpose, we create a task list that feeds our highly optimized TCU kernel with tiles to multiply.

- An attestation that algorithms can enjoy performance benefits even if the TCUs are not fully occupied, thanks to the high computational throughput of TCUs.

- A highly-optimized GPU-based implementation of spGEMM, which improves the performance of spGEMM of cuSPARSE and CUSP by 1.6× and 3.4× on average. We performed our tests on SuiteSparse dataset, the most common benchmark for spGEMM, to facilitate comparison with other spGEMM approaches.

- The source code of our spGEMM approach, tSparse, which is publicly available on `https://github.com/oresths/tSparse`.

Our techniques reduce the total execution time of spGEMM. Our approach can be used to accelerate linear algebra applications, e.g., solving of a system of equations, graph applications, e.g. betweenness centrality etc.

BIBLIOGRAPHY

[1] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008. doi: 10.1109/IEEESTD.2008.4610935. (Cited on page 34.)

[2] Jérémie A. Allard, Hadrien Courtecuisse, and François Faure. Implicit fem Solver on GPU for Interactive Deformation Simulation. *GPU Computing Gems Jade Edition*, pages 281–294, 2012. doi: 10.1016/B978-0-12-385963-1.00021-6. ISBN: 9780123859631. (Cited on page 79.)

[3] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967. (Cited on page 48.)

[4] Rasmus Resen Amossen, Andrea Campagna, and Rasmus Pagh. Better size estimation for sparse matrix products. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 406–419. Springer, 2010. (Cited on page 63.)

[5] Fredrik Andersson, Marcus Carlsson, and Viktor V Nikitin. Fast algorithms and efficient gpu implementations for the radon transform and the back-projection operator represented as convolution operators. *SIAM Journal on Imaging Sciences*, 9(2):637–664, 2016. (Cited on page 22.)

[6] Pham Nguyen Quang Anh, Rui Fan, and Yonggang Wen. Balanced Hashing and Efficient GPU Sparse General Matrix-Matrix Multiplication. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 36:1–36:12, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4361-9. doi: 10.1145/2925426.2926273. URL http://doi.acm.org/10.1145/2925426.2926273. event-place: Istanbul, Turkey. (Cited on pages 64, 74, and 79.)

[7] Arash Ashari, Naser Sedaghati, John Eisenlohr, and P Sadayappan. An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on gpus. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 273–282. ACM, 2014. (Cited on page 59.)

[8] J Bano, A Hostettler, S A Nicolau, S Cotin, C Doignon, H S Wu, M H Huang, L Soler, and J Marescaux. Simulation of pneumoperitoneum for laparoscopic surgery planning. *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, (7510):91–98, 2012. ISSN 16113349. URL http://link.springer.com/chapter/10.1007/978-3-642-33415-3_12. (Cited on page 42.)

[9] Richard Barrett, Michael W Berry, Tony F Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods*, volume 43. Siam, 1994. (Cited on page 58.)

[10] Adrien Bartoli, Toby Collins, Nicolas Bourdel, and Michel Canis. Computer assisted Minimally Invasive Surgery: Is medical Computer Vision the answer to improving laparosurgery? *Medical Hypotheses*, 79(6): 858–863, 2012. ISSN 03069877. doi: 10.1016/j.mehy.2012.09.007. URL http://dx.doi.org/10.1016/j.mehy.2012.09.007. (Cited on page 12.)

[11] N. Bell, S. Dalton, and L. Olson. Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods. *SIAM Journal on Scientific Computing*, 34 (4):C123–C152, January 2012. ISSN 1064-8275. doi: 10.1137/110838844. URL https://epubs.siam.org/doi/abs/10.1137/110838844. (Cited on pages 56, 61, 63, 65, and 84.)

[12] Sylvain Bernhardt, Stéphane A. Nicolau, Luc Soler, and Christophe Doignon. The status of augmented reality in laparoscopic surgery as of 2016. *Medical Image Analysis*, 37:66–90, 2017. ISSN 13618423. doi: 10.1016/j.media.2017.01.007. (Cited on page 12.)

[13] Mark Buxton. Haswell new instruction descriptions now available! software.intel.com, retrieved January 17, 2019, 2019. (Cited on page 30.)

[14] Julien Carron and Antony Lewis. Maximum a posteriori cmb lensing reconstruction. *Physical Review D*, 96(6):063510, 2017. (Cited on page 22.)

[15] Frédéric Champagnat and Yves Le Sant. Efficient Cubic B-spline Image Interpolation on a GPU. *Journal of Graphics Tools*, 16(4):218–232, 2012. ISSN 2165-347X. doi: 10.1080/2165347X.2013.824736. (Cited on page 22.)

[16] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. pages 1–9, 2014. URL http://arxiv.org/abs/1410.0759. arXiv: 1410.0759. (Cited on page 80.)

[17] IEEE Standards Committee et al. 754-2008 ieee standard for floating-point arithmetic. *IEEE Computer Society Std*, 2008:517, 2008. (Cited on page 61.)

[18] Hadrien Courtecuisse, Jérémie Allard, Pierre Kerfriden, Stéphane P.A. Bordas, Stéphane Cotin, and Christian Duriez. Real-time simulation of contact and cutting of heterogeneous soft-tissues. *Medical Image Analysis*, 18(2):394–410, 2014. ISSN 13618415. doi: 10.1016/j.media.2013.11.001. URL http://dx.doi.org/10.1016/j.media.2013.11.001. Publisher: Elsevier B.V. ISBN: 1361-8415. (Cited on page 80.)

[19] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. Accelerating reduction and scan using tensor core units. pages 46–57, 2019. (Cited on pages 56 and 67.)

[20] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2015. URL http://cusplibrary.github.io/. Version 0.5.1. (Cited on pages 60, 61, 64, 66, 73, and 84.)

[21] Steven Dalton, Luke Olson, and Nathan Bell. Optimizing Sparse MatrixMatrix Multiplication for the GPU. *ACM Transactions on Mathematical Software*, 41(4):1–20, 2015. ISSN 00983500. doi: 10.1145/2699470.

URL http://dl.acm.org/citation.cfm?doid=2835205.2699470. (Cited on pages 62, 63, 65, 74, 76, and 79.)

[22] Timothy Davis. Algorithm 9xx: Suitesparse: Graphblas: graph algorithms in the language of sparse linear algebra. 2018. (Cited on page 56.)

[23] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011. (Cited on pages 73 and 84.)

[24] Julien Demouth. Sparse Matrix-Matrix Multiplication on the GPU. page 21. Nvidia, 2012. (Cited on pages 64 and 74.)

[25] Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. Multithreaded sparse matrix-matrix multiplication for many-core and gpu architectures. *Parallel Computing*, 78:33–46, 2018. (Cited on page 65.)

[26] Xiaogang Du, Jianwu Dang, Yangping Wang, Song Wang, and Tao Lei. A Parallel Nonrigid Registration Algorithm Based on B-Spline for Medical Images. *Computational and Mathematical Methods in Medicine*, 2016. ISSN 17486718. doi: 10.1155/2016/7419307. (Cited on pages 13 and 22.)

[27] Anders Eklund, Paul Dufort, Daniel Forsberg, and Stephen M. LaConte. Medical image processing on the GPU - Past, present and future. *Medical Image Analysis*, 17(8):1073–1094, 2013. ISSN 13618415. doi: 10.1016/j.media.2013.05.008. URL http://dx.doi.org/10.1016/j.media.2013.05.008. arXiv: 1206.3975 Publisher: Elsevier B.V. ISBN: 1361-8415. (Cited on page 20.)

[28] Nathan D. Ellingwood, Youbing Yin, Matthew Smith, and Ching Long Lin. Efficient methods for implementation of multi-level nonrigid mass-preserving image registration on GPUs and multi-threaded CPUs. *Computer Methods and Programs in Biomedicine*, 127:290–300, apr 2016. ISSN 0169-2607. doi: 10.1016/J.CMPB.2015.12.018. (Cited on pages 13, 22, and 23.)

[29] Rob Farber. *CUDA application design and development*. Elsevier, 2011. (Cited on page 28.)

[30] O Fluck, Christoph Vetter, Wolfgang Wein, Ali Kamen, Bernhard Preim, and Rüdiger Westermann. A survey of medical image registration on graphics hardware. *Computer methods and programs in biomedicine*, 104(3): e45–e57, 2011. (Cited on page 20.)

[31] Agner Fog. *VCL: C++ vector class library v1.30*. www.agner.org/optimize, 2017. (Cited on page 30.)

[32] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*. Technical University of Denmark, 2018-09-15 edition, September 2018. (Cited on page 30.)

[33] Alejandro Forner, Josep M Llovet, and Jordi Bruix. Hepatocellular carcinoma. *The Lancet*, 379(9822):1245 – 1255, 2012. ISSN 0140-6736. doi: https://doi.org/10.1016/S0140-6736(11)61347-0. URL http://www.

sciencedirect.com/science/article/pii/S0140673611613470. (Cited on page 4.)

[34] John R Gilbert, Steve Reinhardt, and Viral B Shah. High-performance graph algorithms from parallel sparse matrices. In *International Workshop on Applied Parallel Computing*, pages 260–269. Springer, 2006. (Cited on page 56.)

[35] Alice Gillams, Nahum Goldberg, Muneeb Ahmed, Reto Bale, David Breen, Matthew Callstrom, Min Hua Chen, Byung Ihn Choi, Thierry de Baere, Damian Dupuy, et al. Thermal ablation of colorectal liver metastases: a position paper by an international panel of ablation experts, the interventional oncology sans frontières meeting 2013. *European radiology*, 25(12):3438–3454, 2015. (Cited on page 4.)

[36] Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides, and Nicolás Guil. An optimized approach to histogram computation on gpu. *Machine Vision and Applications*, 24(5):899–908, 2013. (Cited on page 21.)

[37] Google. Google cloud TPU, 2019. URL https://cloud.google.com/tpu/. (Cited on page 56.)

[38] Markus Grabner, Thomas Pock, Tobias Gross, and Bernhard Kainz. Automatic differentiation for gpu-accelerated 2d/3d registration. In *Advances in Automatic Differentiation*, pages 259–269. Springer, 2008. (Cited on page 20.)

[39] Felix Gremse, Andreas Höfter, Lars Ole Schwen, Fabian Kiessling, and Uwe Naumann. GPU-Accelerated Sparse Matrix-Matrix Multiplication by Iterative Row Merging. *SIAM Journal on Scientific Computing*, 37(1):C54–C71, January 2015. ISSN 1064-8275, 1095-7197. doi: 10.1137/130948811. URL http://epubs.siam.org/doi/10.1137/130948811. (Cited on pages 62, 66, and 74.)

[40] Felix Gremse, Kerstin Küpper, and Uwe Naumann. Memory-Efficient Sparse Matrix-Matrix Multiplication by Row Merging on Many-Core Architectures. *SIAM Journal on Scientific Computing*, 40(4):C429–C449, January 2018. ISSN 1064-8275, 1095-7197. doi: 10.1137/17M1121378. URL https://epubs.siam.org/doi/10.1137/17M1121378. (Cited on pages 62, 66, and 74.)

[41] Fred G Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)*, 4(3):250–269, 1978. (Cited on page 61.)

[42] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J Higham. Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers. *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18)*, 2018. ISSN 1749-9097. ISBN: 9781538683842. (Cited on pages 56, 62, and 67.)

[43] Xiao Han, Lyndon S Hibbard, and Virgil Willcut. Gpu-accelerated, gradient-free mi deformable registration for atlas-based mr brain image

segmentation. In *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 141–148. IEEE, 2009. (Cited on page 21.)

[44] Jon S Heiselman, Logan W Clements, Jarrod A Collins, Jared A Weis, Amber L Simpson, Sunil K Geevarghese, T Peter Kingham, William R Jarnagin, and Michael I Miga. Characterization and correction of soft tissue deformation in laparoscopic image-guided liver surgery. *Journal of Medical Imaging*, In Press(2), 2018. ISSN 23294310. doi: 10.1117/1.JMI.5.2.021203. (Cited on pages 13 and 42.)

[45] HiperNav. High performance soft tissue navigation. online, 2017. URL https://hipernav.eu. (Cited on pages 4 and 20.)

[46] Brent Hollingsworth. New bulldozer and piledriver instructions: A step forward for high performance software development, 2012. (Cited on page 30.)

[47] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 300–314, 2019. (Cited on page 79.)

[48] Hsieh Hou and H Andrews. Cubic splines for image interpolation and digital filtering. *IEEE Transactions on acoustics, speech, and signal processing*, 26(6):508–517, 1978. (Cited on page 16.)

[49] Kaixi Hou, Weifeng Liu, Hao Wang, and Wu-chun Feng. Fast segmented sort on gpus. In *Proceedings of the International Conference on Supercomputing*, pages 1–10, 2017. (Cited on page 79.)

[50] Yufeng Huang, Tong Tong, Wei Liu, Ya Fan, Huanqing Feng, and Chuanfu Li. Accelerated diffeomorphic non-rigid image registration with cuda based on demons algorithm. In *2010 4th International Conference on Bioinformatics and Biomedical Engineering*, pages 1–4. IEEE, 2010. (Cited on page 20.)

[51] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *The International Journal of High Performance Computing Applications*, 18(1):135–158, 2004. (Cited on page 59.)

[52] Intel. Intel intrinsics guide. software.intel.com, retrieved January 17, 2019, 2019. (Cited on page 30.)

[53] Ole Jakob Elle, Andrea Teatini, and Orestis Zachariadis. Data for: Accelerating B-spline Interpolation on GPUs: Application to Medical Image Registration. *Mendeley Data*, 2019. doi: 10.17632/kj3xcd776k.1. URL http://dx.doi.org/10.17632/kj3xcd776k.1. (Cited on page 86.)

[54] S. F. Johnsen, S. Thompson, M. J. Clarkson, M. Modat, Y. Song, J. Totz, K. Gurusamy, B. Davidson, Z. A. Taylor, D. J. Hawkes, and S. Ourselin. Database-Based Estimation of Liver Deformation under Pneumoperitoneum for Surgical Image-Guidance and Simulation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Arti-*

ficial Intelligence and Lecture Notes in Bioinformatics), 9350:450–458, 2015. ISSN 16113349. URL https://doi.org/10.1007/978-3-319-24571-3_54. (Cited on page 13.)

[55] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 600–614, 2019. (Cited on page 80.)

[56] Ramaseshan Kannan. Efficient sparse matrix multiple-vector multiplication using a bitmapped format. In *20th Annual International Conference on High Performance Computing*, pages 286–294. IEEE, 2013. (Cited on page 59.)

[57] Jeremy Kepner and John Gilbert. Graph algorithms in the language of linear algebra. 2011. (Cited on page 56.)

[58] Hyesoon Kim, Richard Vuduc, Sara Baghsorkhi, Jee Choi, and Wen-mei Hwu. Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU). *Synthesis Lectures on Computer Architecture*, 7:1–96, 2012. ISSN 1935-3235. doi: 10.2200/S00451ED1V01Y201209CAC020. (Cited on page 32.)

[59] Zbigniew Koza, Maciej Matyka, Sebastian Szkoda, and Łukasz Mirosław. Compressed multirow storage format for sparse matrices on graphics processing units. *SIAM Journal on Scientific Computing*, 36(2):C219–C239, 2014. (Cited on page 59.)

[60] Rakshith Kunchum, Ankur Chaudhry, Aravind Sukumaran-Rajam, Qingpeng Niu, Israt Nisa, and P. Sadayappan. On Improving Performance of Sparse Matrix-matrix Multiplication on GPUs. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 14:1–14:11, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5020-4. doi: 10.1145/3079079.3079106. URL http://doi.acm.org/10.1145/3079079.3079106. event-place: Chicago, Illinois. (Cited on pages 64, 66, and 74.)

[61] Thomas Lange, Sebastian Eulenstein, Michael Hünerbein, Hans Lamecker, and Peter-Michael Schlag. Augmenting intraoperative 3d ultrasound with preoperative models for navigation in liver surgery. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 534–541. Springer, 2004. (Cited on page 42.)

[62] Christopher P Lee, Zhoubing Xu, Ryan P Burke, Rebeccah Baucom, Benjamin K Poulose, Richard G Abramson, and Bennett A Landman. Evaluation of five image registration tools for abdominal CT: Pitfalls and opportunities with soft anatomy. In *Medical Imaging 2015: Image Processing*, volume 9413, page 94131N. International Society for Optics and Photonics, 2015. (Cited on page 23.)

[63] Thomas Martin Lehmann, Claudia Gonner, and Klaus Spitzer. Survey: Interpolation methods in medical image processing. *IEEE transactions on medical imaging*, 18(11):1049–1075, 1999. (Cited on page 20.)

[64] Ang Li, Akash Kumar, Yajun Ha, and Henk Corporaal. Correlation ratio based volume image registration on gpus. *Microprocessors and Microsystems*, 39(8):998–1011, 2015. (Cited on page 21.)

[65] Yuping Lin and Gérard Medioni. Mutual information computation and maximization using gpu. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–6. IEEE, 2008. (Cited on page 20.)

[66] Lifeng Liu, Meilin Liu, Chongjun Wang, and Jun Wang. Lsrb-csr: A low overhead storage format for spmv on the gpu systems. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, pages 733–741. IEEE, 2015. (Cited on page 59.)

[67] Weifeng Liu and Brian Vinter. A framework for general sparse matrix-matrix multiplication on GPUs and heterogeneous processors. *Journal of Parallel and Distributed Computing*, 85:47–61, 2015. ISSN 07437315. doi: 10.1016/j.jpdc.2015.06.010. URL `http://dx.doi.org/10.1016/j.jpdc.2015.06.010`. arXiv: 1504.05022 Publisher: Elsevier Inc. (Cited on pages 62, 63, 65, and 74.)

[68] Xinyang Liu, Sukryool Kang, William Plishker, George Zaki, Timothy D. Kane, and Raj Shekhar. Laparoscopic stereoscopic augmented reality: toward a clinically viable electromagnetic tracking solution. *Journal of Medical Imaging*, 3(4):045001, 2016. ISSN 2329-4302. doi: 10.1117/1.JMI.3.4.045001. URL `http://medicalimaging.spiedigitallibrary.org/article.aspx?doi=10.1117/1.JMI.3.4.045001`. (Cited on pages 12 and 15.)

[69] Piotr Luszczek, Jakub Kurzak, Ichitaro Yamazaki, and Jack Dongarra. Towards numerical benchmark for half-precision floating point arithmetic. *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017*, (1), 2017. ISSN 14716372. doi: 10.1109/HPEC.2017.8091031. ISBN: 9781538634721. (Cited on page 61.)

[70] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. NVIDIA tensor core programmability, performance & precision. *Proceedings - 2018 IEEE 32nd International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2018*, pages 522–531, 2018. ISSN 0021-9193. doi: 10.1109/IPDPSW.2018.00091. arXiv: 1803.04014 ISBN: 9781538655559. (Cited on pages 56, 62, and 67.)

[71] K. Matam, S. R. Krishna Bharadwaj Indarapu, and K. Kothapalli. Sparse matrix-matrix multiplication on modern architectures. In *2012 19th International Conference on High Performance Computing*, pages 1–10, December 2012. doi: 10.1109/HiPC.2012.6507483. (Cited on pages 60 and 61.)

[72] Giovanni Mauri, Luca Cova, Stefano De Beni, Tiziana Ierace, Tania Tondolo, Anna Cerri, S Nahum Goldberg, and Luigi Solbiati. Real-time US-CT/MRI image fusion for guidance of thermal ablation of liver tumors undetectable with US: results in 295 cases. *Cardiovascular and interventional radiology*, 38(1):143–151, 2015. (Cited on page 42.)

[73] Paulius Micikevicius and NVIDIA. Local memory and register spilling, 2011. NVIDIA Training. (Cited on page 28.)

[74] Marc Modat, Gerard R. Ridgway, Zeike A. Taylor, Manja Lehmann, Josephine Barnes, David J. Hawkes, Nick C. Fox, and Sébastien Ourselin. Fast free-form deformation using graphics processing units. *Computer Methods and Programs in Biomedicine*, 98(3):278–284, 2010. ISSN 01692607. doi: 10.1016/j.cmpb.2009.09.002. (Cited on pages 14, 16, 20, 21, 23, 34, 35, 42, 44, and 83.)

[75] Pinar Muyan-Ozcelik, John D Owens, Junyi Xia, and Sanjiv S Samant. Fast deformable registration on the gpu: A cuda implementation of demons. In *2008 International Conference on Computational Sciences and Its Applications*, pages 223–233. IEEE, 2008. (Cited on page 20.)

[76] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. High-Performance and Memory-Saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 101–110, Bristol, United Kingdom, August 2017. IEEE. ISBN 978-1-5386-1042-8. doi: 10.1109/ICPP.2017.19. URL http://ieeexplore.ieee.org/document/8025284/. (Cited on pages 65, 74, and 79.)

[77] Felix Nickel, Hannes G. Kenngott, Jochen Neuhaus, Nathanael Andrews, Carly Garrow, Johannes Kast, Christof M. Sommer, Tobias Gehrig, Carsten N. Gutt, Hans Peter Meinzer, and Beat P. Müller-Stich. Computer tomographic analysis of organ motion caused by respiration and intraoperative pneumoperitoneum in a porcine model for navigated minimally invasive esophagectomy. *Surgical Endoscopy and Other Interventional Techniques*, 32(10):1–12, 2018. ISSN 14322218. doi: 10.1007/s00464-018-6168-2. URL http://dx.doi.org/10.1007/s00464-018-6168-2. (Cited on page 42.)

[78] NVIDIA. Profiler User's Guide. (September), 2017. URL http://docs.nvidia.com/cuda/profiler-users-guide/index.html. (Cited on pages 28, 38, and 39.)

[79] NVIDIA. Cuda compiler driver nvcc. 2017. URL https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html. (Cited on page 28.)

[80] NVIDIA. CUDA C Programming Guide 9.2. (November), 2018. URL http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. (Cited on pages 5, 6, 7, 12, 22, 23, 29, 34, 36, and 37.)

[81] NVIDIA. Nvidia Turing Gpu Architecture Whitepaper. 2018. (Cited on pages 29 and 34.)

[82] NVIDIA. Nvidia Turing Gpu Architecture Whitepaper. 2018. (Cited on pages xv, 5, 6, 61, 67, and 73.)

[83] NVIDIA. CUDA C Programming Guide 10.2. (November), 2019. URL http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. (Cited on pages 56, 62, 66, 70, and 83.)

[84] NVIDIA. Cuda toolkit. online, 2019. URL https://docs.nvidia.com/cuda/. (Cited on pages 64 and 73.)

[85] Nvidia Team. Tuning CUDA Applications for Turing . *Nvidia*, (October): 1–8, 2018. (Cited on page 62.)

[86] Sébastien Ourselin, Radu Stefanescu, and Xavier Pennec. Robust registration of multi-modal images: towards real-time clinical applications. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 140–147. Springer, 2002. (Cited on page 20.)

[87] Alessia Pacioni, Marina Carbone, Cinzia Freschi, Rosanna Viglialoro, Vincenzo Ferrari, and Mauro Ferrari. Patient-specific ultrasound liver phantom: materials and fabrication method. *International Journal of Computer Assisted Radiology and Surgery*, 10(7):1065–1075, 2015. ISSN 18616429. doi: 10.1007/s11548-014-1120-y. URL http://dx.doi.org/10.1007/s11548-014-1120-y. (Cited on page 42.)

[88] Gilles Perrot, Stéphane Domas, and Raphaël Couturier. An optimized GPU-based 2D convolution implementation. *Concurrency and Computation: Practice and Experience*, 28(16):4291–4304, nov 2016. ISSN 15320626. doi: 10.1002/cpe.3752. URL http://doi.wiley.com/10.1002/cpe.3752. (Cited on page 27.)

[89] Igor Peterlík, Hadrien Courtecuisse, Robert Rohling, Purang Abolmaesumi, Christopher Nguan, Stéphane Cotin, and Septimiu Salcudean. Fast elastic registration of soft tissues under large deformations. *Medical image analysis*, 45:24–40, 2018. (Cited on pages 13, 14, 15, 23, and 83.)

[90] William Plishker, Omkar Dandekar, Shuvra S Bhattacharyya, and Raj Shekhar. Towards systematic exploration of tradeoffs for medical image registration on heterogeneous platforms. In *2008 IEEE Biomedical Circuits and Systems Conference*, pages 53–56. IEEE, 2008. (Cited on page 20.)

[91] Josien PW Pluim, JB Antoine Maintz, and Max A Viergever. Mutual-information-based registration of medical images: a survey. *IEEE transactions on medical imaging*, 22(8):986–1004, 2003. (Cited on page 20.)

[92] Josien PW Pluim, Sascha EA Muenzing, Koen AJ Eppenhof, and Keelin Murphy. The truth is hard to make: Validation of medical image registration. In *International Conference on Pattern Recognition (ICPR)*, pages 2294–2300. IEEE, 2016. (Cited on page 43.)

[93] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007. (Cited on page 21.)

[94] Maria R. Robu, João Ramalhinho, Stephen Thompson, Kurinchi Gurusamy, Brian Davidson, David Hawkes, Danail Stoyanov, and Matthew J. Clarkson. Global rigid registration of CT to video in laparoscopic liver surgery. *International Journal of Computer Assisted Radiology and Surgery*, 13(6):947–956, 2018. ISSN 18616429. doi: 10.1007/s11548-018-1781-z. URL https://doi.org/10.1007/s11548-018-1781-z. (Cited on page 12.)

[95] Alexis Roche, Grégoire Malandain, Xavier Pennec, and Nicholas Ayache. The correlation ratio as a new similarity measure for multimodal image registration. In *International Conference on Medical Image Computing and*

Computer-Assisted Intervention, pages 1115–1124. Springer, 1998. (Cited on page 20.)

[96] Daniel Rueckert, Luke I Sonoda, Carmel Hayes, Derek LG Hill, Martin O Leach, and David J Hawkes. Nonrigid registration using free-form deformations: application to breast MR images. *IEEE Transactions on Medical Imaging*, 18(8):712–21, 1999. ISSN 0278-0062. doi: 10.1109/42.796284. (Cited on pages 12, 17, 20, 21, and 42.)

[97] Daniel Ruijters and Philippe Thévenaz. GPU prefilter for accurate cubic B-spline interpolation. *Computer Journal*, 55(1):15–20, 2010. ISSN 00104620. doi: 10.1093/comjnl/bxq086. (Cited on pages 22 and 35.)

[98] Daniel Ruijters, Bart M. ter Haar Romeny, and Paul Suetens. Efficient GPU-Based Texture Interpolation using Uniform B-Splines. *Journal of Graphics, GPU, and Game Tools*, 13(4):61–69, 2008. ISSN 2151-237X. doi: 10.1080/2151237X.2008.10129269. URL `http://dx.doi.org/10.1080/2151237X.2008.10129269`. (Cited on pages 12, 13, 17, and 22.)

[99] Nitin Satpute, Rabia Naseem, Rafael Palomar, Orestis Zachariadis, Juan Gómez-Luna, Faouzi Alaya Cheikh, and Joaquín Olivares. Fast parallel vessel segmentation. *Computer Methods and Programs in Biomedicine*, 192:105430, 2020. ISSN 0169-2607. doi: https://doi.org/10.1016/j.cmpb.2020.105430. URL `http://www.sciencedirect.com/science/article/pii/S0169260719323818`. (Cited on page 80.)

[100] J A Shackleford, N Kandasamy, and G C Sharp. On developing B-spline registration algorithms for multi-core processors. *Physics in Medicine and Biology*, 55(21):6329–6351, 2010. ISSN 0031-9155. doi: 10.1088/0031-9155/55/21/001. (Cited on pages 13 and 21.)

[101] Ramtin Shams, Parastoo Sadeghi, Rodney Kennedy, and Richard Hartley. Parallel computation of mutual information on the gpu with application to real-time registration of 3d medical images. *Computer methods and programs in biomedicine*, 99(2):133–146, 2010. (Cited on pages 20 and 21.)

[102] Ramtin Shams, Parastoo Sadeghi, Rodney a Kennedy, and Richard I Hartley. A Survey of Medical Image Registration on Multicore and the GPU. *IEEE signal processing magazine*, 27(March):50 – 60, 2010. ISSN 10535888. doi: 10.1109/MSP.2009.935387. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5438962`. ISBN: 1053-5888. (Cited on pages 15, 16, 19, 20, and 21.)

[103] Lin Shi, Wen Liu, Heye Zhang, Yongming Xie, and Defeng Wang. A survey of GPU-based medical image computing techniques. *Quantitative imaging in medicine and surgery*, 2(3):188–206, 2012. doi: 10.3978/j.issn.2223-4292.2012.08.02. (Cited on pages 15, 20, and 21.)

[104] Christian Sigg and Markus Hadwiger. Fast third-order texture filtering. *GPU gems*, 2:313–329, 2005. (Cited on pages 12, 13, 22, and 27.)

[105] Edgar Solomonik, Maciej Besta, Flavio Vella, and Torsten Hoefler. Scaling betweenness centrality using communication-efficient sparse matrix multiplication. In *Proceedings of the International Conference for High Per-*

formance Computing, Networking, Storage and Analysis, pages 1–14, 2017. (Cited on page 80.)

[106] Aristeidis Sotiras, Christos Davatzikos, and Nikos Paragios. Deformable medical image registration: A survey. *IEEE transactions on medical imaging*, 32(7):1153, 2013. (Cited on page 12.)

[107] Li Ming Su, Balazs P Vagvolgyi, Rahul Agarwal, Carol E Reiley, Russell H Taylor, and Gregory D Hager. Augmented Reality During Robot-assisted Laparoscopic Partial Nephrectomy: Toward Real-Time 3D-CT to Stereoscopic Video Registration. *Urology*, 73(4):896–900, 2009. ISSN 00904295. doi: 10.1016/j.urology.2008.11.040. URL http://dx.doi.org/10.1016/j.urology.2008.11.040. (Cited on page 15.)

[108] A Teatini, T Langø, B Edwin, O Elle, et al. Assessment and comparison of target registration accuracy in surgical instrument tracking technologies. In *2018 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 1845–1848. IEEE, 2018. (Cited on page 12.)

[109] Andrea Teatini, Wang Congcong, Palomar Rafael, Alaya Cheikh Faouzi, Beghdadi Azeddine, Edwin Bjørn, and Elle Ole Jakob. Validation of stereo vision based liver surface reconstruction for image guided surgery. In *Colour and Visual Computing Symposium (CVCS)*, pages 1–6. IEEE, 2018. (Cited on pages 42 and 51.)

[110] Andrea Teatini, Javier Pérez de Frutos, Benjamin Eigl, Egidijus Pelanis, Davit L Aghayan, Marco Lai, Rahul Prasanna Kumar, Rafael Palomar, Bjørn Edwin, and Ole Jakob Elle. Influence of sampling accuracy on augmented reality for laparoscopic image-guided surgery. *Minimally Invasive Therapy & Allied Technologies*, pages 1–10, 2020. (Cited on page 42.)

[111] Philippe Thévenaz, Thierry Blu, and Michael Unser. Interpolation revisited [medical images application]. *IEEE Transactions on medical imaging*, 19(7):739–758, 2000. (Cited on page 22.)

[112] J.-P. Thirion. Image matching as a diffusion process: an analogy with Maxwell's demons. *Medical Image Analysis*, 2(3):243–260, 1998. ISSN 13618415. doi: 10.1016/S1361-8415(98)80022-4. URL http://linkinghub.elsevier.com/retrieve/pii/S1361841598800224. ISBN: 1361-8415. (Cited on page 20.)

[113] Michael Unser. Splines: A perfect fit for signal and image processing. *IEEE Signal processing magazine*, 16(6):22–38, 1999. (Cited on pages 12, 16, and 20.)

[114] Sinara Vijayan, Ingerid Reinertsen, Erlend Fagertun Hofstad, Anna Rethy, Toril A. Nagelhus Hernes, and Thomas Langø. Liver deformation in an animal model due to pneumoperitoneum assessed by a vessel-based deformable registration. *Minimally Invasive Therapy & Allied Technologies*, 23(5):279–286, 2014. ISSN 1364-5706. doi: 10.3109/13645706.2014.914955. URL http://www.tandfonline.com/doi/full/10.3109/13645706.2014.914955. (Cited on page 42.)

[115] Vasily Volkov. Better performance at lower occupancy. *Proceedings of the GPU Technology Conference*, pages 1–75, 2010. (Cited on pages 27, 28, and 29.)

[116] Nathan Whitehead and Alex Fit-Florea. Precision & Performance : Floating Point and IEEE 754 Compliance for NVIDIA GPUs. *NVIDIA white paper*, 21(10):767–75, 2011. doi: 10.1111/j.1468-2982.2005.00972.x. (Cited on page 36.)

[117] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. Adaptive Sparse Matrix-matrix Multiplication on the GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 68–81, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6225-2. doi: 10.1145/3293883.3295701. URL `http://doi.acm.org/10.1145/3293883.3295701`. event-place: Washington, District of Columbia. (Cited on pages 62, 64, and 74.)

[118] Ichitaro Yamazaki and Xiaoye S Li. On techniques to improve robustness and scalability of a parallel hybrid linear solver. In *International Conference on High Performance Computing for Computational Science*, pages 421–434. Springer, 2010. (Cited on page 56.)

[119] Orestis Zachariadis. Accelerating B-spline Interpolation On GPUs: Application To Medical Image Registration, 2020. URL `https://github.com/oresths/niftyreg_bsi/`. (Cited on page 50.)

[120] Orestis Zachariadis. tSparse: Accelerating Sparse Matrix-Matrix Multiplication with GPU Tensor Cores, 2020. URL `https://github.com/oresths/tSparse/`. (Cited on page 78.)

[121] Orestis Zachariadis, Nitin Satpute, Juan Gómez-Luna, and Joaquín Olivares. Accelerating Sparse Matrix-Matrix Multiplication with GPU Tensor Cores. *Computer and Electrical Engineering*, 2020. Revision submitted. (Cited on page 78.)

[122] Orestis Zachariadis, Andrea Teatini, Nitin Satpute, Juan Gómez-Luna, Onur Mutlu, Ole Jakob Elle, and Joaquín Olivares. Accelerating B-spline Interpolation on GPUs: Application to Medical Image Registration. *Computer Methods and Programs in Biomedicine*, 2020. doi: 10.1016/j.cmpb.2020.105431. (Cited on pages 20 and 50.)

[123] Jianting Zhang and Le Gruenwald. Regularizing Irregularity : Bitmap-based and Portable Sparse Matrix Multiplication for Graph Data on GPUs. *GRADES-NDA*, 2018. doi: 10.1145/3210259.3210263. ISBN: 9781450356954. (Cited on pages 56, 59, 67, 68, 73, and 74.)

[124] Jilin Zhang, Jian Wan, Fangfang Li, Jie Mao, Li Zhuang, Junfeng Yuan, Enyi Liu, and Zhuoer Yu. Efficient sparse matrix–vector multiplication using cache oblivious extension quadtree storage format. *Future Generation Computer Systems*, 54:490–500, 2016. (Cited on page 59.)

DECLARATION

I declare that I have developed this Doctoral Thesis, under the supervision of the thesis' directors and, therefore, I assume the authorship of everything described in this document.

I also declare that the work done is not total or partial plagiarism of any other research done by other people.

I affirm that all the data exposed in this investigation have not been falsified and that any error that may exist in the document has not been consciously introduced.

Córdoba, June 2020

Orestis Zachariadis