View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Online Publikationen der Universität Stuttgart

Institute of Parallel and Distributed Systems
Department Simulation of Large Systems
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3405

GPU-based Numerical Integration in the Partition of Unity Method

Sebastian Kanis

Course of Study: Computer Science

Examiner: Prof. Dr. rer. nat. Marc Alexander Schweitzer

Supervisor: Dipl. Inf. Patrick Diehl

Commenced: Oktober 16, 2012

Completed: April 17, 2013

CR-Classification: G.4, J.2

Abstract

In this thesis, we present a CUDA-implementation of two sub-steps of the Parallel Multilevel Partition of Unity Method (PMPUM). The PMPUM is a method for the approximation of partial differential equations (PDEs) whose main computational effort is caused by the integration of the weak formulation. Therefore, an efficient CUDA-implementation of the required steps could speed up a given PMPUM-implementation. The core of this thesis is the analysis of the applicability of CUDA in the PMPUM. To this end the required steps, the decomposition of the domain and the integration, were implemented using CUDA. The analysis showed, that the usage of CUDA can speed up the implementation and identified the limitations of the implementation. We give recommendations how to improve these limitations and expect the performance to increase further with these recommendations applied.

Contents

1 Introduction 15

2 The Parallel Multilevel Partition of Unity Method 17
- 2.1 Model Problem . . . 17
- 2.2 Partition of Unity Space . . . 18
- 2.3 Galerkin Discretization . . . 19
- 2.4 Decomposition of the Domain . . . 20
  - 2.4.1 Decomposition Task . . . 20
  - 2.4.2 Decomposers . . . 23
  - 2.4.3 Comparison of the Approaches . . . 28
- 2.5 Complexity . . . 30
  - 2.5.1 Integration . . . 30
  - 2.5.2 Decomposition . . . 34

3 Implementation 35
- 3.1 CUDA . . . 35
  - 3.1.1 Execution Model . . . 36
  - 3.1.2 Thread Hierarchy . . . 36
  - 3.1.3 Memory Hierarchy . . . 38
- 3.2 Framework Integration . . . 40
- 3.3 Memory Layout . . . 42
- 3.4 Kernel Design . . . 44
- 3.5 Implementation Properties and Limitations . . . 45

4 Results 47
- 4.1 Hardware and Metrics . . . 47
  - 4.1.1 Hardware . . . 47
  - 4.1.2 Metric . . . 48
- 4.2 Experiment: Overall Performance . . . 49
- 4.3 Experiment: Performance depending on Problem Parameters . . . 51
- 4.4 Experiment: Performance of different Steps of the GPU-Implementation . . . 52
- 4.5 Experiment: Single precision Performance . . . 54
- 4.6 Experiment: Profiler Data Analysis . . . 56
- 4.7 Experiment: 3D Performance . . . 59
- 4.8 Experiment: Different Hardware . . . 60
- 4.9 Summary of the Results . . . 61
- 4.10 Further Improvements . . . 62

4.10.1 GPU-based Accumulation . . . 62
4.10.2 Improved Memory Access . . . 64
4.10.3 Multi Kernel Version . . . 64
4.10.4 Level of Parallelization . . . 65

5 Conclusion 67

A Memory Layout of the Result Data 69

B Raw Result Data 71

Bibliography 83

List of Figures

2.1 The weight function W_i of adjacent patches ω_i (left) and the resulting partition of unity φ_i (right) in one space dimension . . . 19
2.2 The inner and boundary patches generated by an uniform refinement strategy . 21
2.3 The decomposition of a patch ω_i and its neighborhood C_i for integration 22
2.4 Comparison of the decomposition of one patch ω_i and two of its neighbors C_i, generated by two approaches. . . . 29
2.5 The cells generated by the Tree-Based Approach and Tensor-Product Approach from an uniformly refined domain. . . . 29
2.6 The decomposition of a patch ω_i and its neighborhood C_i for integration with patches on different levels . . . 31
2.7 The cells generated by the Tree-Based Approach and the Tensor-Product Approach in a nonuniform case . . . 31

3.1 The hardware components of a CUDA capable GPU from NVIDIA™ with Compute Capability 2.x . . . 36
3.2 The CUDA Execution Model . . . 37
3.3 The CUDA Thread Hierarchy . . . 38
3.4 The CUDA Memory Hierarchy . . . 39
3.5 Coalesced vs. non coalesced memory access . . . 40
3.6 The phases to solve a given BVP by the framework . . . 40
3.7 The phases to solve a given BVP by the framework, with the CUDA extension 41
3.8 Two different memory layouts for the patches $\mathcal{W}_i$ (2D) . . . 43

4.1 The performance for the integration on Ω_I of the CPU-implementation $\mathcal{P}_{CPU}$. 49
4.2 The performance of the integration on Ω_I of the GPU-implementation $\mathcal{P}_{CPU}$. 50
4.3 The relative performance of the integration on Ω_I of the CPU and GPU-implementations $\mathcal{P}_{CPU}/\mathcal{P}_{GPU}$. . . . 50
4.4 The performance of the integration on Ω_I of the CPU-implementation $\mathcal{P}_{CPU}$ in the 3D case. . . . 59
4.5 The performance of the integration on Ω_I of the GPU-implementation $\mathcal{P}_{GPU}$ in the 3D case. . . . 59
4.6 The relative performance of the integration on Ω_I of the CPU and GPU-implementation $\mathcal{P}_{CPU}/\mathcal{P}_{GPU}$ in the 3D case. . . . 60
4.7 A possible indexing scheme for 2 adjacent neighborhoods . . . 63
4.8 Alternative memory layout for the parameters . . . 63

A.1 The indexing scheme used for the results . . . 70

List of Tables

2.1 The number of cells generated by both decomposer approaches, for all inner patches of a cover of a square domain (2D) using uniform refinement 29
2.2 The number of cells generated by both decomposer approaches, for all inner patches of a cover of a square domain (2D) using h-refinement at the center of the domain 30

3.1 The memory requirement of the base functions ψ_i^n for all patches with support on a cell 46

4.1 Reference system configuration 47
4.2 Performance of different architectures 48
4.3 The ratio of the duration of the GPU-implementation between consecutive levels. 51
4.4 The ratio of the duration of the GPU-implementation between consecutive polynomial degrees. 52
4.5 The ratio of the duration of the preprocessing and the whole GPU-implementation 53
4.6 The ratio of the duration of the kernel and the whole GPU-implementation. . . 53
4.7 The ratio of the duration of the postprocessing and the whole GPU-implementation. 54
4.8 The ratio of the duration of the kernel for SP and double precision floating point (DP) 55
4.9 The occupancy of the SMs when executing the kernel for the decomposition and integration of Ω_I. 56
4.10 The ratio of the issued instructions of the kernel between consecutive levels . . 57
4.11 The ratio of the issued instructions of the kernel between consecutive polynomial degrees 57
4.12 The ratio of instructions issued due to waits for local memory of the total instructions issued. 58
4.13 The ratio of non divergent branches and all branches in the kernel for the decomposition and integration of Ω_I. 58
4.14 The relative performance of the implementation on the NVIDIA™ K20 and the Geforce GTX 560 Ti 61

B.1 The duration required by the CPU-implementation for the integration on Ω . . 71
B.2 The duration required by the GPU-implementation for the integration on Ω . . 71
B.3 The relative duration required by the GPU-implementation and the CPU-implementation for the integration on Ω 72
B.4 The duration required by the CPU implementation for decomposition and integration on Ω_I 72

B.5 The duration required by the GPU implementation for decomposition and integration on Ω_I . . . 72
B.6 The relative duration required by the GPU-implementation and the CPU-implementation for decomposition and integration on Ω_I . . . 73
B.7 The duration required by the preprocessing of the GPU-implementation for the decomposition and integration on Ω_I . . . 73
B.8 The duration required by the kernel of the GPU-implementation for the decomposition and integration on Ω_I . . . 73
B.9 The duration required by the postprocessing of the GPU-implementation for the decomposition and integration on Ω_I . . . 74
B.10 The duration required by the GPU-implementation when using SP for the integration on Ω_I . . . 74
B.11 The duration required by the GPU-implementation for the preprocessing when using single precision floating point (SP) for the integration on Ω_I . . . 74
B.12 The duration required by the GPU-implementation for the kernel when using SP for the integration on Ω_I . . . 75
B.13 The duration required by the GPU-implementation for the postprocessing when using SP for the integration on Ω_I . . . 75
B.14 The number of instructions issued during the kernel execution for the integration on Ω_I . . . 75
B.15 The number of L1 load misses for local memory requests in the kernel for the integration on Ω_I . . . 76
B.16 The number of L1 store misses for local memory requests in the kernel for the integration on Ω_I . . . 76
B.17 The number of branches in the kernel for the integration on Ω_I . . . 76
B.18 The number of divergent branches in the kernel for the integration on Ω_I . . . 77
B.19 The duration required by the CPU-implementation for the integration on Ω_I in 3D . . . 77
B.20 The duration required by the GPU-implementation for the integration on Ω_I in 3D . . . 77
B.21 The ratio of the duration required by the GPU-implementation and the CPU-implementation for decomposition and integration on Ω_I in 3D. . . . 77
B.22 The relative performance of the implementation on the NVIDIA™ K20 and the Geforce GTX 560 Ti using SP . . . 78
B.23 The relative performance of the implementation on the NVIDIA™ K20 using SP and DP . . . 78
B.24 The duration required by the kernel for the integration on Ω_I on the NVIDIA™ K20 . . . 78
B.25 The duration required by the kernel for the integration on Ω_I on the NVIDIA™ K20 using SP . . . 79

List of Algorithms

2.1 Decomposition using the Tree-Based Approach in pseudo code 24
2.2 Decomposition using the Tensor-Product Approach in pseudo code 26

3.1 Scheduling of neighborhoods for computation on the GPU in pseudo code . . . 42
3.2 Kernel workflow 44

List of Abbreviations

ALU arithmetic and logic unit. 55, 56, 81
API application programming interface. 35, 36, 48

BVP boundary value problem. 7, 15, 17, 40, 41

CPU central processing unit. 7–9, 14, 16, 25, 28, 35–37, 39, 42, 43, 47, 49, 50, 52, 54, 58–60, 62, 64, 67, 68, 71–73, 77
CUDA Compute Unified Device Architecture. 3, 7, 15, 16, 35–39, 41, 45, 46, 48, 52, 54, 55, 58, 61, 67, 68, 81

DP double precision floating point. 8, 9, 47–49, 51, 54–56, 61, 71, 78
DRAM dynamic random-access memory. 35, 39

FMA fused-multiply-add instruction. 48

gflops giga floating point operations per second. 47, 48
GPGPU general-purpose computing on graphics processing units. 15, 20, 23, 35, 40
GPU graphics processing unit. 7–10, 14–16, 20, 21, 25, 28, 30, 35, 36, 39–44, 47, 49–54, 57, 59, 60, 62–64, 67–69, 72–75, 77, 81

MPI Message Passing Interface. 41, 47

PDE partial differential equation. 12, 13, 15, 17, 19, 40, 41, 67, 68
PMPUM Parallel Multilevel Partition of Unity Method. 3, 12, 13, 15–17, 20, 30, 40, 41, 45, 48, 67, 68

SIMT Single Instruction Multiple Threads. 37, 81
SM Streaming Multiprocessors. 8, 35, 37–39, 49, 55, 56, 60, 81
SP single precision floating point. 8, 9, 47, 48, 54–56, 61, 74, 75, 78, 79

List of Symbols

Symbol	Description
A	stiffness matrix in the Galerkin approach. 12, 19, 20, 41, 43, 69
$A_{(i,n),(j,m)}$	block entry of the stiffness matrix A. 20, 43
B	boundary conditions of a given PDE. 17
C_i	neighbor patches of patch ω_i. 7, 19, 20, 22, 23, 25, 29, 31, 32, 34, 43–45, 69
$H^1(\Omega)$	Sobolev space on Ω. 19, 20
L	second order symmetric elliptic partial differential operator. 17, 20
M_c	the patches with support on an integration cell. 14, 32, 33, 41
N	number of points $card(P)$ in the initial point set P. 18, 21, 30, 32, 34, 48
O	upper bound for the complexity of an algorithm in the Big O notation. 30, 32–34
P	initial set of points in the PMPUM. 12–14, 18, 19
V^{PU}	function space used in the PMPUM. 18, 20
$V_i^{p_i}$	function space defined on patch ω_i. 13, 14, 18, 21, 32–34, 41, 44, 45, 69, 81
W_i	weight function defined on patch ω_i. 7, 18–20, 22, 23, 32
Δ	Laplace operator. 19, 20
Γ_D	part of the domain's boundary $\partial\Omega$ where Dirichlet boundary conditions are given $\Gamma_D = \partial\Omega \setminus \Gamma_N$. 12, 13, 17
Γ_N	part of the domain's boundary $\partial\Omega$ where Neumann boundary conditions are given $\Gamma_N = \partial\Omega \setminus \Gamma_D$. 12, 13, 17
Ω	domain on which the given PDE is defined. 7–9, 12–14, 17–22, 30, 44, 49, 50, 52, 56, 58–61, 71–79
α_i	The stretch factor used for ω_i in the cover generation of the PMPUM. 20–22, 31
$\hat{f}$	discrete right hand side of a given PDE. 13, 19, 20, 41, 69
$\hat{u}$	coefficient vector of the discrete solution of a given PDE. 19

κ	diffusion coefficient in the reaction diffusion equation. 17, 19, 20, 22
ω_i	patch affiliated with point $x_i \in P$. 7, 12, 13, 18–20, 22, 23, 25, 29–32, 34, 42–44, 63, 65, 69, 81
$\vec{n}$	normal vector on the $\partial\Omega$. 17, 19, 20
$\partial\Omega$	boundary of the domain Ω. 12, 13, 17, 19–21
ψ_i^n	n-th base function of $V_i^{p_i}$ defined on patch ω_i. 8, 13, 18, 20, 22, 32, 33, 41, 43, 45, 46
φ_i	partition of unity function defined on patch ω_i. 7, 18–20, 22, 41, 43
$a(\cdot, \cdot)$	bilinear form. 14, 20, 33, 41–43, 45, 52
b_i	number of base functions $card(\psi_i^n)$ in $V_i^{p_i}$. 33, 42, 44, 45, 48, 69
$card()$	the cardinality of a set. 12, 13, 32, 34
d	space dimension. 18, 21, 30, 32–34, 42, 48, 51, 52, 57
dof	number of degrees of freedom. 48–50, 59, 60
f	continuous right hand side of a given PDE. 17, 19, 22
$f_{(i,n)}$	block entry of the discrete right hand side $\hat{f}$. 20
g	boundary values of a given PDE. 17, 20
g_D	Dirichlet boundary values of a given PDE on Γ_D. 17
g_N	Neumann boundary values of a given PDE on Γ_N. 17
h_i	stretch of patch ω_i. 18, 21, 30, 42, 81
l	level used in PMPUM. 21, 29, 30, 33, 34, 48–52, 57, 60
$l(\cdot)$	linear form. 14, 20, 33, 41, 45, 52
n_{CI}	number of cells on a patch ω_i. 32, 34
n_{NI}	number of nodes used by a quadrature rule. 32–34, 51, 52, 57
$n_{\omega C}$	number of patches on an integration cell. 32–34, 45
p	polynomial degree. 21, 30, 32–34, 42, 45, 46, 48, 51–54, 57, 60–62, 67
r	reaction coefficient in the reaction diffusion equation. 17, 19, 20, 22
u	continuous solution of a given PDE. 17–19
v	continuous test function used in the variational formulation. 19
x_i	i-th point in P. 13, 18, 42
$\mathcal{C}_{EI}$	costs for the evaluation of the integrand at an integration node. 32, 34
$\mathcal{C}_{NI}$	costs for numerical integration in the PMPUM. 32, 34

$\mathcal{C}_{\mathrm{B}M_c}$ costs for the evaluation of the base functions of the function space $V_i^{p_i}$ on each of the patches the patches with support on an integration cell (M_c) on an integration cell. 32, 33

$\mathcal{C}_{\mathrm{W}M_c}$ costs for the evaluation of the weight functions of the patches M_c on an integration cell. 32

$\mathcal{C}_{bf}$ costs for the evaluation of the bi-linear form $a(\cdot,\cdot)$. 32, 33

$\mathcal{C}_{lf}$ costs for the evaluation of the linear form $l(\cdot)$. 32, 33

$\mathcal{P}_{CPU}$ performance of a CPU-implementation. 7, 49, 50, 59, 60

$\mathcal{P}_{GPU}$ performance of a GPU-implementation. 7, 49, 50, 59, 60

$\mathcal{W}$ the set of patches generated from P. 18, 41, 43

$\mathcal{W}_i$ the patches evaluated in the i-the GPU schedule. 7, 41, 43, 62–64

t time. 48

Ω_I domain of the inner patches. 7–9, 21, 22, 44, 49, 50, 52, 56, 58–61, 72–79

1 Introduction

In today's society, simulation technology has not only become an integral part of society – it has become almost indispensable [2]. Simulation can be used in situations where physical, ethical or financial reasons impede experiments. Thus, simulation technology can be utilized to gain knowledge about actions or natural phenomenons that we are unable or unwilling to observe. These situations range from weather forecast, where a prediction of future events can not be reached through observation, to the design and development of automobiles, where crash tests are economized to shrink costs.

Simulation is based on domain specific models, consisting of representations of the subjects of study and the physical or other relations between them. Different methods are then applied to reproduce the results of actions or phenomenons. These methods, however, require a large amount of computations to calculate a detailed solution. Therefore, in almost every case, computers are utilized to implement these methods.

In many cases, the models for simulation employ partial differential equations (PDEs) to describe the real world [8, p. 1]. A PDE models the alteration of a given quantity in time and/or space. Examples for that are the stress of materials during crash tests or the diffusion of one chemical substance into another. When additional information on the initial situation and/or the behavior of the phenomenon at the boundary of the domain inspected is given, a so called initial or boundary value problem (BVP) results from the PDE.

For most BVPs arising from practical situations, no definite solution can be calculated with today's methods [8, p. 1]. Therefore, manifold methods exist to compute approximate solutions. Methods for the simulation of physical phenomenons can be classified into mesh-based methods and particle schemes. Both categories exhibit different strengths and weaknesses. Mesh-free methods try to merge the best of both categories. The Parallel Multilevel Partition of Unity Method (PMPUM) [8, p. 13] is a mesh-free method that is especially suitable in situations where some properties of solutions are known a priori.

The implementation of the Parallel Multilevel Partition of Unity Method (PMPUM) can be parallelized to decrease the duration for the generation of a solution. Multiple technologies can be used for that. Graphics processing units (GPUs), initially used for the display of 2D and 3D data in computer graphics, provide great parallel processing power [19]. This capability can be utilized for more general problems by technologies called general-purpose computing on graphics processing units (GPGPU). Compute Unified Device Architecture (CUDA) [19] is a technology of NVIDIA™, that enables the usage of NVIDIA™'s GPUs for GPGPU. However, not for all methods an increase of the performance can be reached when using CUDA.

In this thesis we present an implementation of the steps in PMPUM that account for most of the computational effort. The goal of this thesis is to analyze the potential of CUDA to increase the performance of a given CPU-based[1] implementation of the PMPUM.

This thesis is organized as follows:

Chapter 2 - The Parallel Multilevel Partition of Unity Method: Provides an introduction to the PMPUM. Afterwards, it focuses on the algorithm used in the GPU-implementation. Finally, we give estimates for the costs of these same.

Chapter 3 – Implementation: Introduces CUDA and based on that, gives an outline of the implementation. The explanation of properties and limitations of the implementation closes the chapter.

Chapter 4 – Results: Presents experiments performed to analyze the implementation. Therefore it provides information on the hardware and metrics used. Based on the results of the experiments, we provide recommendations for further improvements.

Chapter 5 – Conclusion: Summarizes the results of this thesis and provides an outlook to further development.

[1]central processing unit (CPU)

2 The Parallel Multilevel Partition of Unity Method

Partial differential equations (PDEs) can be used to model problems in a wide range of fields. Since for many PDEs no exact solutions can be found, there exist manifold methods to generate approximate solutions. See [3] or [1] for an introduction to PDEs and approximation methods. The PMPUM is one method to approximate solutions. In this chapter we give an overview about the method, for a detailed introduction see [8]. In the following we present how a PDE is discretized in the PMPUM. The remarks on the discretization are influenced by the introduction to the PMPUM in [20, p. 3 ff.]. Then we focus on the computational task for which an implementation will be presented in the following chapter.

2.1 Model Problem

BVPs, for which the PMPUM can be used for the approximation of the solution, are in general of the form:

$$\begin{aligned} Lu &= f \text{ in } \Omega \subset \mathbb{R}^d \\ Bu &= g \text{ in } \partial\Omega \end{aligned}$$

Here L is a second order symmetric elliptic partial differential operator and B are suitable boundary conditions. The task is to find the solution u that solves the equation. We choose the reaction-diffusion equation to illustrate the discretization of a PDE in the PMPUM:

$$\begin{aligned} -\nabla \cdot (\kappa \nabla u) + r \cdot u &= f \text{ in } \Omega \subset \mathbb{R}^d \\ u &= g_D \text{ in } \Gamma_D \subset \partial\Omega \\ \kappa \nabla u \cdot \overrightarrow{n} &= g_N \text{ in } \Gamma_N = \partial\Omega \backslash \Gamma_D \end{aligned} \tag{2.1}$$

Here κ is the diffusion coefficient, while r is the reaction coefficient. The boundary $\partial\Omega$ is split into Γ_D on which Dirichlet boundary conditions are given and Γ_N on which Neumann boundary conditions are given. An approximate solution should have the following two properties [8, p. 13]:

- local approximability
- inter-element continuity

Local approximability is the capability to approximate the solution u at a given point as close as possible. Inter-element continuity means that an approximate solution should be continuous in some sense.

2.2 Partition of Unity Space

We introduce the partition of unity space V^{PU} that is used to find an approximate solution. We start with an arbitrarily chosen point set P with points from the domain Ω:

$$P = \{x_i \in \mathbb{R}^d | x_i \in \Omega, i \in \{1, .., N\}\}$$

For each point x_i we define a patch ω_i by a d-rectangular:

$$\omega_i = \bigotimes_{l=1}^{d} \{x_i{}^l - h_i{}^l, x_i{}^l + h_i{}^l\}$$

Here x_i is the center of the patch and h_i is the stretch in each space dimension. On each patch we define a local approximation space $V_i^{p_i} = span\langle\psi_i^n\rangle$. This local approximation space $V_i^{p_i}$ is used for the local approximability. We define $\mathcal{W}$ as the set of patches generated from the initial set of points P. The global function space V^{PU}, required by the Galerkin approach, is generated by the product of the local approximation space $V_i^{p_i}$ with suitable partition of unity functions φ_i:

$$V^{PU} = \sum_i \varphi_i V_i^{p_i} = \sum_i \varphi_i \langle\{\psi_i^n\}\rangle = span\langle\{\varphi_i \psi_i^n\}\rangle$$

The local approximation space $V_i^{p_i}$ can be chosen independently from all other local approximation spaces $V_j^{p_j}$. For smooth functions polynomials exhibit good approximation properties, for more irregular functions enrichment functions can be added to the base ψ_i^n of the local approximation space $V_i^{p_i}$. Using a partition of unity ensures that arbitrary functions can be added to the local approximation space without violating the inter-element continuity property. The required partition of unity functions φ_i are defined patch-wise. On each patch ω_i a weight function W_i is defined, where:

$$\begin{aligned} &W_i(x) \neq 0 \text{ for } x \in \omega_i \\ &W_i(x) = 0 \text{ else} \end{aligned}$$

Based on these weight functions W_i we construct the partition of unity function φ_i:

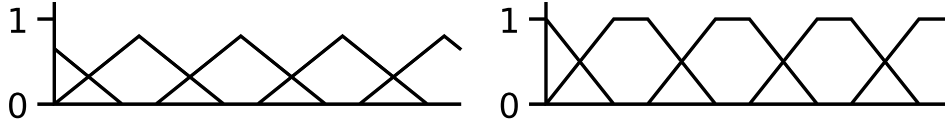

Figure 2.1: The weight function W_i of adjacent patches ω_i (left) and the resulting partition of unity φ_i (right) in one space dimension

$$\varphi_i(x) = \frac{W_i(x)}{\sum_{k \in C_i} W_k(x)}$$

Here the neighborhood C_i consists of all patches ω_j whose weight function W_j have overlapping support with the weight function W_i of ω_i.

Figure 2.1 illustrates the weight function W_i and the resulting partition functions φ_i of some adjacent patches. The cover generation, presented in [8, p. 98 ff.] creates a cover of the domain from the initial point set P, so that the union $\bigcup \omega_i$ of the patches covers the domain Ω. The choice of the weight functions W_i and thus the overlap of the patches determines the sparsity pattern of the stiffness matrix A generated by the Galerkin approach presented in the following.

2.3 Galerkin Discretization

The Galerkin approach can be used to discretize a PDE like the one given in equation 2.1. It utilizes the weak formulation of the problem to redefine the problem in a function space in which a solution can be found. See [3, p. 133 ff.] for an introduction to the weak formulation that is also called the variational formulation. In this section we only give a very short introduction into the concepts required in following. First, we define an approximate solution of a PDE:

$$A\hat{u} = \hat{f}$$

Here A is the stiffness matrix that needs to be generated, $\hat{f}$ is the discretized right-hand side and $\hat{u}$ is the vector of coefficients of the solution. Using a constant diffusion coefficient κ and reaction coefficient r in equation 2.1 results in:

$$-\kappa \Delta u + ru = f$$

Here Δ is the Laplace operator. We generate the variational formulation by multiplying the equation with a test function v from the trial and test space $H^1(\Omega)$ and integrating it. Afterwards we apply Green's first identity and obtain:

$$\int_\Omega -v(\Delta\kappa \cdot u + r \cdot u) = \int_\Omega \kappa \nabla u \nabla v + ruv + \int_{\partial\Omega} \kappa v(\nabla u \cdot \overrightarrow{n}) = \int_\Omega f \cdot v$$

Here $\vec{n}$ is the outer normal on Ω. Let $a(\cdot,\cdot)$ be a continuous elliptic bilinear form induced by $L = -\kappa\Delta + r$ on the Sobolev space $H^1(\Omega)$ and $l(\cdot)$ be a continuous linear form. Applying the Galerkin approach using the base and test functions from V^{PU} results in:

$$A = (A_{(i,n),(j,m)}), \text{ with } A_{(i,n),(j,m)} = a(\varphi_i\psi_i^n, \varphi_j\psi_j^m)$$
$$\hat{f} = (f_{(i,n)}), \text{ with} f_{(i,n)} = l(\varphi_i\psi_i^n)$$

Since the partition of unity-functions φ_i only have local support, the integrals on Ω only must be evaluated for all $\omega_i \cap \omega_j \cap \Omega$ and $\omega_i \cap \omega_j \cap \partial\Omega$ respectively. We only tackle Ω using the GPU and thus can assume $g = 0$ without restriction. When applying the Galerkin approach to the model problem given in equation 2.1 the resulting computational task is given by:

$$\int_\Omega \kappa\nabla(\varphi_i\psi_i^n)\nabla(\varphi_j\psi_j^m) + r(\varphi_i\psi_i^n)(\varphi_j\psi_j^m) \tag{2.2}$$
$$\int_\Omega (\varphi_i\psi_i^n)$$

We need to compute the integrals on all $\omega_i \cap \omega_j \cap \Omega$. Therefore, first the domain must be decomposed into integration domains on which the set of patches, whose weight function W_i have support, is determined. After that, the integrals must be computed for all integration domains. In the following we focus on these 2 steps since they account for a major part of the computational effort of the whole PMPUM [8, p. 153].

2.4 Decomposition of the Domain

In this section we present the decomposition task described in section 2.3 from an implementation point of view. After that different approaches to solve this task are presented and discussed in terms of suitability for GPGPU.

2.4.1 Decomposition Task

The Galerkin discretization requires the computation of the integrals given in formula 2.2. We assume that a cover of patches for the domain has already been generated. The algorithm used for the cover construction can be found in [8, p. 98 ff.]. The cover generation provides covers on different levels. The generated cover does not only cover the whole domain $\Omega \subset \bigcup \omega_i$, but assures the overlap of a patch ω_i with all its neighbors C_i. This is reached by stretching the extent of each patch with a factor $\alpha_i > 1$. This is required to reach the inter-element continuity property given in section 2.1.

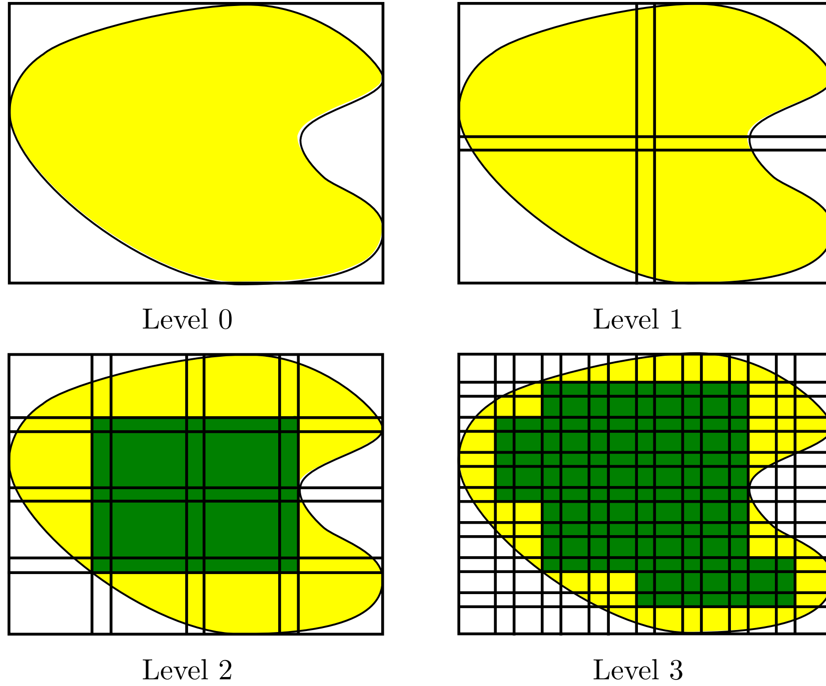

Figure 2.2: The inner (green) and boundary (yellow) patches generated by an uniform refinement strategy from level 0 to 3. The overlap of the patches (caused by α_i) was chosen to be equal on all levels for illustration. Usually it is chosen proportionately to the stretch h_i of a patch.

Figure 2.2 shows the cover generated for a domain on the levels $0-3$ for an uniform-h-refinement strategy.[1] [2] For uniform refinement the number of patches is $(2^d)^l$ for d space dimensions on Level l. Patches with an overlap with the boundary of the domain $\partial\Omega$ are called “boundary patches”. The patches with no overlap are called “inner patches” and the domain they cover is denoted Ω_I in the following. Since the boundary patches intersect the domain boundary an approximation of the domain needs to be calculated. This results in irregular patterns and hence is not suitable for computation on GPUs. Thus we focus in the following on the tasks given for the computation of the integral on the inner patches:

[1] H-refinement is a refinement strategy which increases the number of points N on each level, in contrast to that p-refinement is the increase of the polynomial degree p, used in all $V_i^{p_i}$ on each level.

[2] Uniform in this context means that all patches on a given level are subdivided in every space dimension to form the next level. In contrast to that, adaptive refinement strategies only subdivide a part of the given patches.

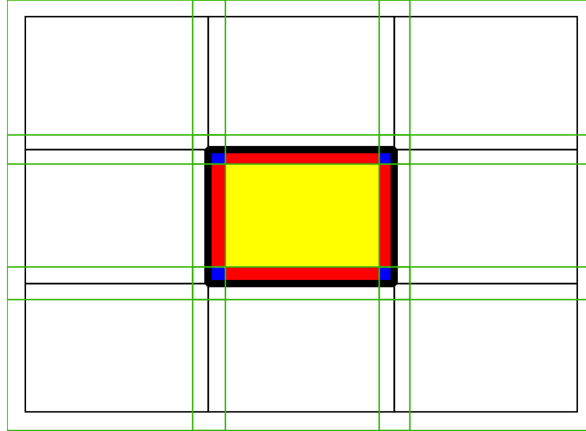

Figure 2.3: The decomposition of a patch ω_i and its neighborhood C_i for integration. The basic extent of the patch (bold) and its neighbors is given by the black rectangles. The green rectangles are the patches stretched by α_i. The needed decomposition is given by all small rectangles in the black bold rectangle. The fill color of the rectangle shows the number of patches on the cells. Yellow stand for 1, red for 2 and blue for 4 patches with support on the cell.

$$\int_{\Omega_I} \kappa\nabla(\varphi_i\psi_i^n)\nabla(\varphi_j\psi_j^m) + r(\varphi_i\psi_i^n)(\varphi_j\psi_j^m) \qquad (2.3)$$

$$\int_{\Omega_I} f(\varphi_i\psi_i^n)$$

Calculating these integrals requires the decomposition of the overlapping patches. This task is illustrated in figure 2.3 for a patch and its neighbors in 2 space dimensions for a cover generation which uses uniform refinement.

When using linear splines as weight functions additional cells are generated. The integration domain must be split at the patch center where the derivative of the weight function is discontinuous when more than one patch has support on a cell. In figure 2.5, which will be discussed in detail when comparing different approaches for the decomposition, these additional splits can be seen. Thus, the resulting decomposition generates at least 13 cells in 2 space dimensions and 57 cells in 3 space dimensions.

Different decomposers approaches can be benchmarked using the following metrics:

1. The number of cells should be minimal. Additional cells do not falsify the results, but lead to unnecessary computations.
2. The number of patches on the cells should be minimal. Additional patches on a cell don't influence the result since their weight function $W_i = 0$ on the cell, but they introduce unnecessary computations.

3. The complexity in time and space should be minimal. For GPGPU the resource data structures as well should be taken into account.

2.4.2 Decomposers

The requirements given in the previous section need to be met by all considered approaches for the decomposition. In the following two approaches are presented and their suitability for an implementation using GPGPU is discussed.

Tree-Based Approach

The first approach uses a tree to generate the required cells for integration. The basic ideas to generate a tree with the following properties:

1. Each node stores an geometric extent and a list of patches and references to two children.
2. A leaf represents a resulting cell.
3. The patches with support on a cell, represented by the leaf, are those in the lists of patches of all nodes on the path from the root to the leaf.
4. The extent stored for non-leaf nodes is the union of the extents of their children.

The algorithm 2.1 starts by adding the patch ω_i itself to the tree. This is achieved by storing its nonstretched extent and a reference to it as the root node of the tree. After that all adjacent patches C_i are added subsequently. We start by comparing the stretched extent to that of the root node. If it is equal, we add the patch to the patch list of the node. If not, the comparison is done for all children whose extents have an intersection with the stretched extent of the patch. This is done until a leaf is reached. When the extent of the leaf is not equal to the stretched extent of the patch two children are added to the leaf, one with an intersection of the stretched extent of the patch and the node and one without. The patch is added to the new node with an intersection with the stretched extent of the patch. Figure 2.4 shows this process for the first two patches of a neighborhood. To get the cells we traverse the tree and store the patches on the path from the root to each leaf. This results in a list of cells containing the patches with support on each cell. The algorithm is adapted to the usage of linear splines as weight functions W_i. Therefore, the cells, on which more than one patch has support and the extent has an intersection with the patch center, are split at the patch center. The resulting patches are illustrated in figure 2.5. Because of the usage of the tree we call this approach Tree-based Decomposition from here on.

The key benefits of this approach are:

- It produces the minimal number of cells.
- It allows fast and random access of all cells.

The disadvantages of this approach are:

Algorithm 2.1 Decomposition using the Tree-Based Approach in pseudo code

```
Tree                  // each node contains a box, a list of patches and a list of children
procedure DECOMPOSE(patch)
    Tree.box = patch.box
    TREE.PATCHLIST.APPEND(patch)
    for all neighbor ∈ neighborhood of patch do
        ADDNEIGHBOR(Tree, neighbor)
    end for
    return GETCELLS( )
end procedure
procedure ADDNEIGHBOR(node, patch)
    if ISEQUAL(patch.box, node.box) then
        NODE.PATCHLIST.APPEND(patch)
    else if ISLEAF(node) then
        overlap = INTERSECTION(patch.box, node.box)
        nonOverlap = patch.box - overlap
        NODE.CHILDREN.APPEND(overlap, patch)
        NODE.CHILDREN.APPEND(nonOverlap)
    else
        for all child ∈ node.children do
            if INTERSECTION(patch.box, child.box) then // only descend to children having
                                                       // an overlap with the patch
                ADDNEIGHBOR(child, patch)
            end if
        end for
    end if
end procedure
procedure GETCELLS( )
    cells                                              // the resulting list of cells
    for all leaf ∈ TREE.GETLEAFS( )do      // For all leafs follow the path to the leaf and
                                           // collect the patches with support on the cell
        cellPatches = LEAF.GETPATCHESONPATH( )
        cell = (leaf.box, cellPatches)
                    // the cells with more than one patch and intersecting the patch's center
                      // must be split due to the usage of linear splines as weight functions
        if INTERSECTS(cell.box, patchCenter) && CELLPATCHES.LENGTH( ) > 1 then
            splittedCells = CELL.SPLIT(patchCenter)
            CELLS.APPEND(splittedCells)
        else
            CELLS.APPEND(cell)
        end if
    end for
    return cells
end procedure
```

- Under the assumption that the order of the patches in the input is unknown, the generated tree is of a priori unknown structure.
- A tree implementation requires dynamic or complex manual memory handling.

On central processing units (CPUs) these restrictions are no drawback since the loss of performance due to lot of small memory allocations is insignificant. The number of cells however and the number of patches per cell is optimal leading to a minimum number of evaluation in the following integration. Since the situation is quite different for a GPU, dynamic memory allocation is expensive and more evaluations are cheap one should consider other approaches.

Tensor-Product Approach

An other approach utilizes the concept of tensor products. The basic idea of this approach is to store the boundary of each intersection of patches in a list per dimension. Afterwards these points per dimension can be used to generate a grid of cells. The information stored at one point can be altered. That leads to different complexity of the algorithm and quality of the results, according to the metrics presented in section 2.4.1.

Algorithm 2.2 shows the approach, called Tensor-Product Approach in the following. In the shown variation the algorithm stores a list of points per space dimension, we will call these points “split points” in the following. Each split point consists of the coordinate and a list of splits. A split stores a reference to a patch and whether the point is the minimum, center or maximum of that patch in the given dimension. For the patch ω_i itself we store the minimum, center and maximum to the split points in each dimension. When adding a patch of the neighborhood C_i we add the minimum, center and maximum to the split points in each dimension. If the point is outside the domain of the patch, whose neighborhood is currently decomposed, the information on minimum and maximum are stored at the boundary of that patch. The generated split points for two example patches are shown in figure 2.4 on the right.

These split points then can be used to retrieve the cells. Therefore, we start at the lowest split point for each dimension. We add the patches with a minimum at this point to a list of active patches which is stored per dimension. The generated cell has the extent from the current to the next split point in all dimensions. The patches with support on the cell are those which are currently active in all dimensions. Iterating over all split points in all dimensions generates all cells. At a split point the patches with minimum are added to the list of active patches in the current dimension. For a maximum the patch is removed, the center has no influence on the active patches and can thus be omitted for the calculation of active patches. We therefore only store the patches whose support begins or ends at a split point. Before we discuss variations of this approach the key benefits and disadvantages are listed.

The key benefit of this approach is:

Algorithm 2.2 Decomposition using the Tensor-Product Approach in pseudo code

```
SplitPoints[dim]    // a list of split-points for each dimension, each split-point stores the
                    // coordinate and the patches which start or end at the point
localDomain                                                    // the box of the center patch
selectedPatches[dim]        // list of Booleans for each dimension used in getNextCell
index[dim] // multi dimensional index with number of split-points entries in each component
procedure decompose(patch)
    localDomain = patch.box
    for all d ∈ dim do                         // Add the split points of the center patch
        index[d] = 0
        addSplitPoint(d, patch.box.min[d], patch, min)
        addSplitPoint(d,patch.box.center[d], patch, mid)
        addSplitPoint(d,patch.box.max[d], patch, max)
    end for
    for all neighbor ∈ neighborhood of patch do// Add the split points of all other patches
        for all d ∈ dim do
            // if a patch boundary is intersecting the localdomain add it to the split points,
              // else add the patch to the list of splits of the min/max of the localdomain
            if neighbor.box.min[d] > localDomain.min[d] then
                addSplitPoint(d, neighbor.box.min[d], patch, min)
            else
                SplitPoints[d][0].splits.append(patch, min)
            end if
            if neighbor.box.center[d] > localDomain.min[d] & neighbor.box.center[d] < local-
Domain.max[d] then
                addSplitPoint(d, neighbor.box.center[d], patch, mid)
            end if
            if neighbor.box.max[d] < localDomain.max[d] then
                addSplitPoint(d, neighbor.box.max[d], patch, max)
            else
                SplitPoints[d][SplitPoints[d].length].splits.append(patch, max)
            end if
        end for
    end for
end procedure
procedure addSplitPoint(dim,coord, patch, type)
    for all SplitPoint ∈ SplitPoints[dim] do
        if SplitPoint.coord == coord then
            SplitPoint.splits.append(patch,type)
        else if SplitPoint.coord > coord then
            newSplitPoint(coord, (patch,type))
            SplitPoints[dim].insert(newSplitPoint, currentPosition)
        end if
    end for
    newSplitPoint(coord, (patch,type))
    SplitPoints[dim].append(newSplitPoint)
end procedure
```

Continuation of Algorithm 2.2: Decomposition using the Tensor-Product Approach in pseudo code

```
procedure getNumberOfCells
    result = 1
    for all d in dim do
        result *= SplitPoints[d].length
    end for
    return result
end procedure
procedure getNextCell
    result                                              // the resulting integration cell
    for all d in dim do       // initialize the selected flag (non-selected) for all patches
        if index[d] == 0 then
            for all neighbor in neighborhood do
                selectedPatches[d][neighbor] = False
            end for
        end if
    end for
    for all d in dim do                                      // set integration cell box
        result.box.lower[dim] = splitPoints[d][index[d]].coord
        result.box.upper[dim] = splitPoints[d][index[d]+1].coord
                                                            // update the selected patches
        for all split in SplitPoints[d][index[d]].splits do
            if split.type == min then
                selectedPatches[d][split.patch] == True
            else if split.type == max then
                selectedPatches[d][split.patch] == False
            end if
        end for
    end for
    for all patch in patches do              // add all patches selected in all dimensions
        selected = True
        for all d in dim do
            if  not selectedPatches[d][patch] then
                selected = False
            end if
        end for
        if selected then
            result.patches.append(patch)
        end if
    end for
    index++
    return result
end procedure
```

- the number of split points and maximum number of splits at a split point are known for a cover which uses uniform refinement strategy. Therefore, no dynamic allocation is needed.

The disadvantage of this approach are:

- The cells cannot be requested in arbitrary sequence.
- Only the split points for each dimension are stored and the generation of the cells is basically a tensor-product of the split points. Thus more cells are generated if a split point divides the patch completely but the neighbor, which introduced the split point, only intersects a part of the patch extent. For an example of the additional generated cells see figure 2.5.

The following modifications might be considered:

1. The information which patch has support on which cell can be omitted without difficulty. Hence all patches need to be evaluated on all generated cells. This leads to a remarkable simplification of the algorithm but on the other hand to a considerable amount of unnecessary evaluations.[3]
2. Storing not only the difference between 2 cells but the whole list of active patches at a split point enables random access of cells. On the other hand, the list of patches at a split point needs to contain all patches with support at a split point.

2.4.3 Comparison of the Approaches

To decide which approach is most appropriate for usage on GPUs we compare their properties. For the comparison we use the variation described in algorithm 2.2 of the Tensor-Product Approach.

In terms of the data structures the tensor-product approach doesn't require dynamic memory allocations in contrast to the Tree-Based Approach. Therefore, it seem to be the better approach for the GPU-implementation. The number of patches is larger than for the Tree-Based Approach.[4] Since only the center cell of the Tree-Based Approach is split unnecessarily in the Tensor-Product Approach, the number of resulting cells in the case of uniform refinement is not expected to be significantly higher.

See tables 2.1 and 2.2 for a comparison of the number of integration cells generated by the 2 approaches.[5]. As we see for the uniform case the ratio for the number of cells is constant at 1.23. Since the center cell is split into 4 cells, for each patch instead of 13 cells, 16 are generated. For the adaptive case the explanation is little more complex. When a patch is refined adaptively it is split into four patches on the next level. Thus additional split points

[3]This variation was tested but resulted in a worse performance.
[4]Compare the resulting decompositions in figure 2.5.
[5]Tested with a CPU-implementation of both approaches.

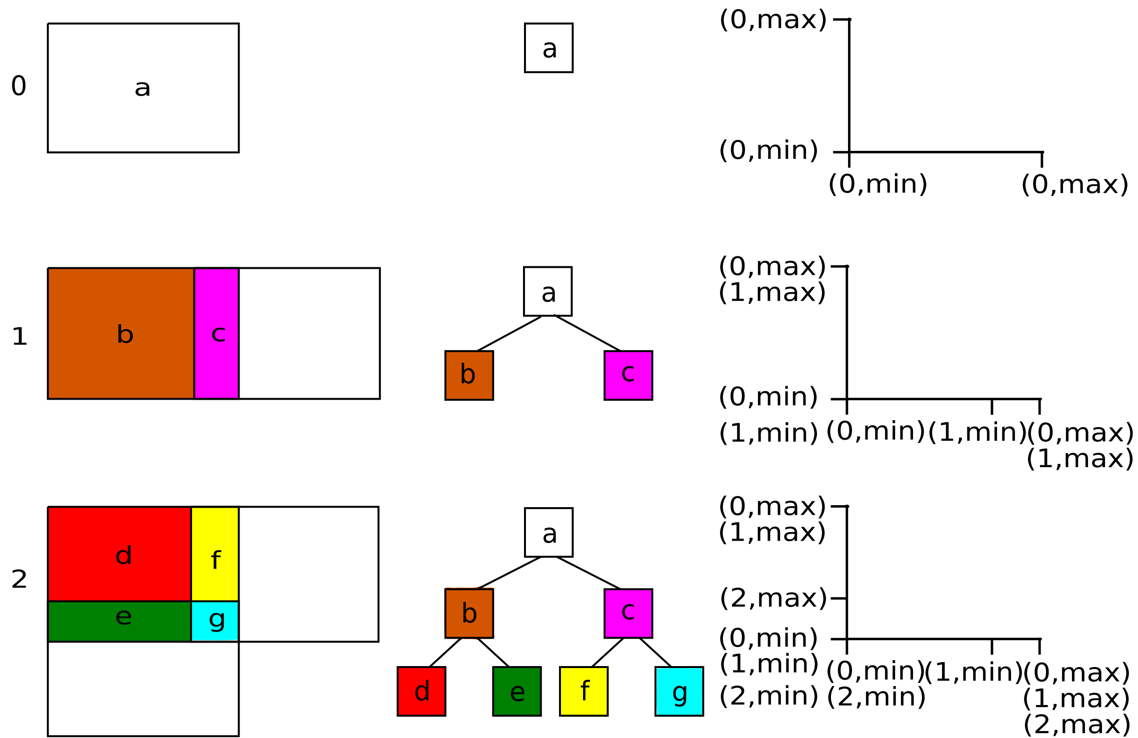

Figure 2.4: Comparison of the decomposition of one patch ω_i and two of its neighbors C_i, generated by two approaches. On the left are the patch (0) and the neighbors (1) and (2), in the middle the tree generated by the Tree-Based Approach and on the right the split points of the Tensor-Product Approach

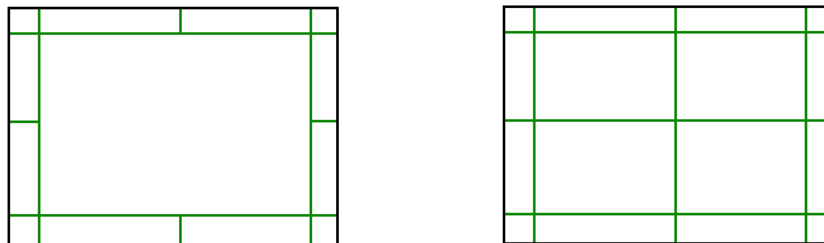

Figure 2.5: The cells generated by the Tree-Based Approach (left) and Tensor-Product Approach (right) from an uniformly refined domain as shown in figure 2.3.

Table 2.1: The number of cells generated by both decomposer approaches, for all inner patches of a cover of a square domain (2D) using uniform refinement

Level (l)	Tree-Based Approach	Tensor-Product Approach	Ratio
2	52	64	1.23
4	2548	3136	1.23
6	49972	61504	1.23
8	838708	1032256	1.23

Table 2.2: The number of cells generated by both decomposer approaches, for all inner patches of a cover of a square domain (2D) using h-refinement at the center of the domain

Level (l)	Tree-Based Approach	Tensor-Product Approach	Ratio
2	52	64	1.23
4	465	796	1.71
6	949	1798	1.89
8	1433	2798	1.95

are introduced. The number of additional split points depends on the refinement pattern. See figure 2.6 for an illustration of this situation.

The figure focuses on the center patch, but, as can be seen in the figure as well, all patches around a patch that is adaptively refined are affected. For nested adaptive refinements, the effect spreads for all patches that are adjacent to a refined patch. Figure 2.7 shows the generated cells from the situation shown in figure 2.6. Table 2.2 shows the case for repeated adaptive refinement steps at the center of the domain Ω. This generates more cells from a patch, because at each refinement level 4 patches are adaptively refined. We note however that this example case the progression seems to be limited by 2. We conclude that the number of generated cells by the Tensor-Product Approach heavily depends on the refinement strategy used.

An factor of 1.23 for the number of integration cells, as given in the uniform case, will not prevent a GPU-implementation from good performance. We chose the Tensor-Product Approach for the GPU-implementation since heavy pointer usage, in the Tree-Based Approach even when implementing the tree without dynamic memory, is expected to decrease performance.

2.5 Complexity

According to [9, p. 228] the optimal time and space complexity for the assembly is in $O(Np^{2d})$. In the following we focus on the metrics, which are relevant for the GPU-implementation. For a derivation of the complexity see [9, p. 228ff.]. Since numerical integration is applied, the optimal bound can hardly realized [9, p. 228]. First estimates for the complexity of the integration are given and afterwards the complexity of the decomposition is discussed.

2.5.1 Integration

Since the PMPUM is a multilevel method, solutions on different levels are computed. We recall that for h-refinement[6] the number of points N is given by $(2^d)^l$.

[6] H-refinement is a refinement strategy in multi level methods. It increases the number of points or elements with increasing level. This results in a decrease of h_i for each patch ω_i.

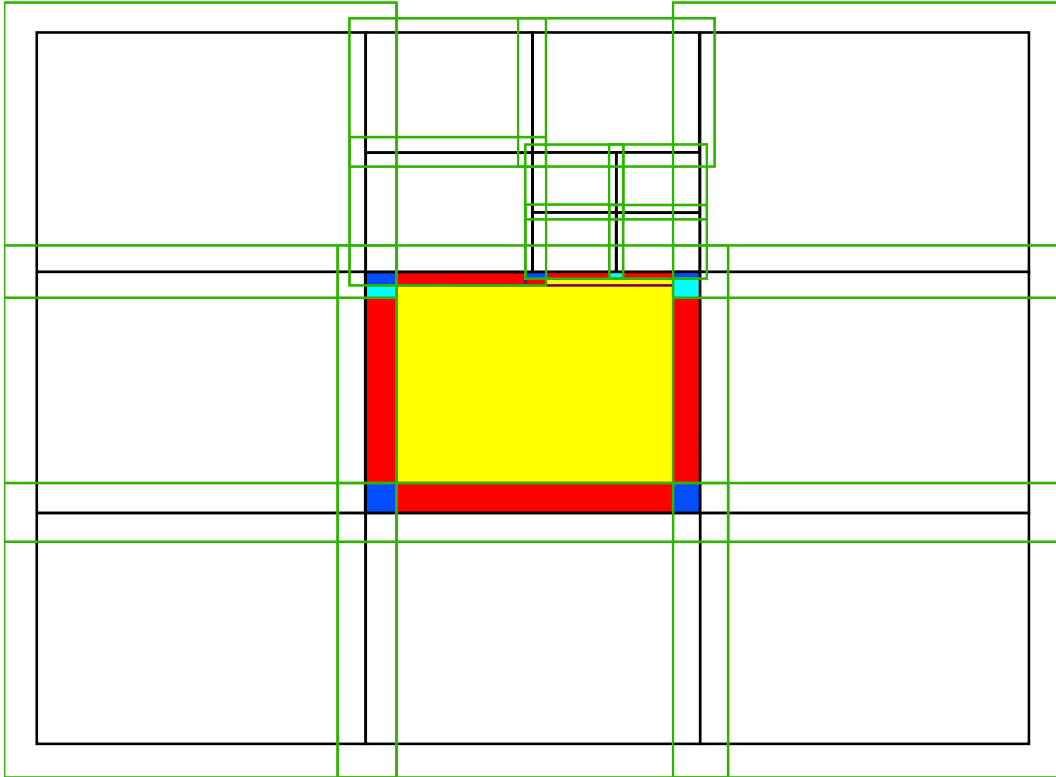

Figure 2.6: The decomposition of a patch ω_i and its neighborhood C_i for integration, with adaptive refinement. The basic extent of the patch (bold) and its neighbors is given by the black rectangles. The green rectangles are the patches stretched by α_i. The needed decomposition is given by all small rectangles in the black bold rectangle. The fill color of the rectangle shows the number of patches on the cells. Yellow stand for 1, red for 2, turquoise for 3 and blue for 4 patches with support on the cell. We note that additional cells are introduced (2 small purple lines) to form rectangular integration cells. See figure 2.3 for the uniform case.

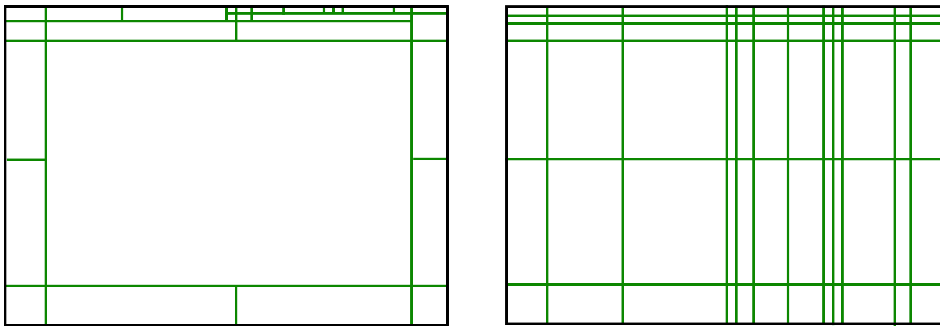

Figure 2.7: The cells generated from the situation given in figure 2.6 by the Tree-Based Approach (left) and the Tensor-Product Approach (right)

The total cost $\mathcal{C}_{NI}$ for the integration is:

$$\mathcal{C}_{NI} = O(n_{CI} \cdot n_{NI} \cdot \mathcal{C}_{EI}) \tag{2.4}$$

Here n_{CI} is the number of cells used for integration. The number of integration points on one integration cell n_{NI} depends on the polynomial degree p and the space dimension d. $\mathcal{C}_{EI}$ are the costs for an evaluation of the integrand at an integration point. The number of integration cells n_{CI} depends on the number of patches N and the decomposition of one patch ω_i and its neighbors C_i.

The number of cells n_{CI} generated by the Tensor-Product Approach presented in section 2.4.2 from one neighborhood C_i depends on the space dimension d and the number of neighbors per dimension. When using uniform h-refinement, assuming 2 neighbors per space dimension and linear splines as weight function W_i the number of cells is given by:

$$n_{CI} = (3+2)^d$$

The required number of integration points n_{NI} on one cell depends on the space dimension d and the polynomial degree p is:

$$n_{NI} = \left(\frac{p+1}{2}\right)^d$$

In the framework on which the implementation is based, see [20] and [24], the number of integration points used is:

$$n_{NI} = (p+1)^d$$

The costs of the evaluation at an integration point $\mathcal{C}_{EI}$ are given by:

$$\mathcal{C}_{EI} = \mathcal{C}_{\mathrm{W}M_c} + \mathcal{C}_{\mathrm{B}M_c} + \mathcal{C}_{lf} + \mathcal{C}_{bf}$$

Here $\mathcal{C}_{\mathrm{W}M_c}$ denotes the costs for the evaluation of the weight functions of all patches M_c on the integration cell. The costs for the evaluation of the base functions of all local function spaces $V_i^{p_i}$ of all patches M_c. And finally $\mathcal{C}_{lf}$ and $\mathcal{C}_{bf}$ are the costs for the evaluation of the linear form and bilinear form respectively. All these costs depend on the number of patches ω_i on an integration cell, denoted $n_{\omega C}$. The number of patches per cell depends on the decomposer used. In general $n_{\omega C}$ is limited by the number of neighbors $card(C_i)$. Applying the Tensor-Product Approach given in algorithm 2.2 the upper bound is given by 4 in the 2D case and 8 in the 3D case.

For each patch ω_i on an integration cell the weight functions of all neighbor patches C_i need to be evaluated. Therefore, the costs for the evaluation of the weight functions at an integration point are given by: $\mathcal{C}_{\mathrm{W}M_c} = O(M_c)$. For the computation of the linear and bilinear form all base function ψ_i^n of the patches M_c need to be evaluated, which accounts for most of the effort.

The costs for evaluation is given by $\mathcal{C}_{\mathrm{BM}_c} = O(M_c \cdot b_i)$, where b_i the number of base functions ψ_i^n in $V_i^{p_i}$ is given by:

$$b_i = \sum_{i=0}^{p} \binom{d+i-1}{d-1}$$

The proof for this formula uses a combinatorial argument. The number of base functions for a polynomial space with degree p without the base function of the polynomial space with degree $p-1$ is equal to the number of possibilities to draw p elements from a set of d elements with replacements.[7]Thus adding the number of possibilities for $0 \leq i \leq p$ leads to the number of base functions of the polynomial space of degree p.
For the sake of completeness we note the complexity for the evaluation of the linear and bilinear forms. For the evaluation of the linear form $l(\cdot)$ for the coefficients for all base functions b_i of all patches on the integration cell M_c need to be computed:

$$\mathcal{C}_{lf} = O(n_{\omega C} \cdot b_i)$$

For the evaluation of the bilinear form for all combinations of patches, with support on the integration cell M_c, we need to compute the product of the base functions b_i. This results in the following costs:

$$\mathcal{C}_{bf} = O({n_{\omega C}}^2 \cdot {b_i}^2)$$

It is especially interesting, for the discussion of the results in chapter 4, to note the factors by which the costs grow for increased level l and polynomial degree p.

For an increase of the level l by one we can see from formula 2.4 that the effort increases by the number of patches resulting from the increased level. The factor by which the effort increases is therefore given by 2^d.

For an increase of the polynomial degree p, the number of required integration points n_{NI} increases. For an increase from $p-1$ to p the number of integration points n_{NI} increases by the factor $\left(\frac{\lceil \frac{p+1}{2} \rceil}{\lceil \frac{p}{2} \rceil}\right)^d$.
When using $n_{NI} = (p+1)^d$ integration points the resulting factor is: $\left(\frac{p+1}{p}\right)^d$

For the same increase, some calculus leads to the factor of the number of base function b_i, that need to be evaluated: $\frac{d+p}{p}$

The factor for the evaluation of the linear form $l(\cdot)$ is the same. Finally, for the evaluation of the bilinear form $a(\cdot,\cdot)$ results in the factor : $\frac{(d+p)^2}{p^2}$.

Since the constant for the assembly of the base function ψ_i^n is the highest, it dominates the effort of the integration for relevant polynomial degrees.

[7]The formula for drawing an unordered sample from cover with replacement can be found in [22, p. 18f.].

We summarize, that for the integration, we expect a factor 2^d for an increase of the level l by one for the complexity. For an increase of the polynomial degree p we expect a factor $\left(\frac{\lceil\frac{p+1}{2}\rceil}{\lceil\frac{p}{2}\rceil}\right)^d \cdot \frac{d+p}{p}$. When using $n_{NI} = (p+1)^d$ integration points the resulting factor is $\left(\frac{p+1}{p}\right)^d \cdot \frac{d+p}{p}$.

2.5.2 Decomposition

The complexity is influenced by the algorithm used for the decomposition in two ways:

1. the complexity of the algorithm itself
2. the influence of the number of cells n_{CI} and the number of patches on a cell $n_{\omega C}$ generated by the decomposer, discussed in the section 2.5.1.

The algorithms described in section 2.4.2 operate on all patches ω_i and their neighborhoods C_i. The number of neighbors of a patch $card(C_i)$ is constant in case of uniform p-refinement.[8] Since the complexity of the decomposers is independent from the local approximation spaces $V_i^{p_i}$, the decomposition is in $O(N)$. For the discussion of the results in chapter 4 we should note, that it may contain a large prefactor.

The number of generated cells n_{CI} and the number of patches on an integration cell $n_{\omega C}$ correlate with the costs for the integration $\mathcal{C}_{NI}$.

Both approaches presented in section 2.4.2 generate the minimal number of patches per cell. And therefore the costs for the evaluation at an integration point $\mathcal{C}_{EI}$ are minimal in both cases. For the number of integration cells n_{CI} we recall from 2.4.3, that for the Tensor-Product Approach n_{CI} is only a constant factor[9] larger than that of the Tree-Based Approach.

The first modification for the Tensor-Product Approach proposed in section 2.4.2 would lead to a massive increase of the costs for the evaluations $\mathcal{C}_{EI}$ since it requires the evaluation of the weight and base functions of all neighbors of a patch. The complexity of the decomposer would not be affected.

We summarize, that the decomposers have a complexity of $O(N)$, but the prefactor may be large.

[8]In case of adaptive refinement the number of neighbors of a patch $card(C_i)$ may be larger but in all cases the number of neighbors is a lot smaller than the number of patches $card(C_i) \ll N$.

[9]Only for uniform h-refinement, for adaptive h-refinement the factor seems to converge to 2, see tables 2.1 and 2.2.

3 Implementation

In this chapter we present the implementation of the decomposition and the integration described in the previous chapter. Therefore, we give an overview about Compute Unified Device Architecture (CUDA) which is used for the GPGPU-implementation and the technologies used in the framework on which the implementation is based. After that we give insight into some details of the implementation. Therefore, the general workflow is presented first. Then the memory layout as well as the implementation of the algorithms presented in the previous chapter are discussed. Finally we depict the properties and limitations of the implementation.

3.1 CUDA

CUDA was introduced by NVIDIA™ in 2006 as a new parallel computing platform and programming model [19]. It aims at simplifying the usage of the performance of GPUs for GPGPU. In the following relevant aspects of CUDA for the implementation are presented. For concepts not covered in this introduction one may start with *CUDA C Programming Guide* [13]. The *CUDA API Reference Manual* [11] gives a complete overview about the CUDA application programming interface (API).
To understand how CUDA is used, some basic understanding of the hardware features of NVIDIA's GPUs is required. Figure 3.1 gives a schematic overview of the components of such a GPU which we in the following refer to as device (compared to the CPU, which we refer to as host). A device consists of multiple Streaming Multiprocessorss (SMs) and multiple memories. The number of multiprocessors varies from type to type, but the features of a single SM is defined by its Compute Capability [13, p. 12f.]. A SM can execute multiple threads in parallel which are organized in warps, that are executed at the same time while other warps on the same SM are inactive. The registers of a SM are split to the number of threads executed on it, but are local to the thread and can not be used for inter thread communication. The shared memory and L1 Cache are used by all threads of a SM where Shared Memory can be used to share data between threads of a thread block[1] and L1 caches access to global memory. Not shown in the figure is the read-only cache which is located in each SM for devices with Compute Capability higher than 3.0. L2 caches all these access for all SMs. The global memory is a dynamic random-access memory (DRAM) which can be accessed by all threads and is used for the communication with the host as well. For detailed specification and limitations for all these components see the *CUDA C Programming Guide* [13, p. 148 ff.].

[1] The concept of thread blocks and this limitation is explained in subsection 3.1.3.

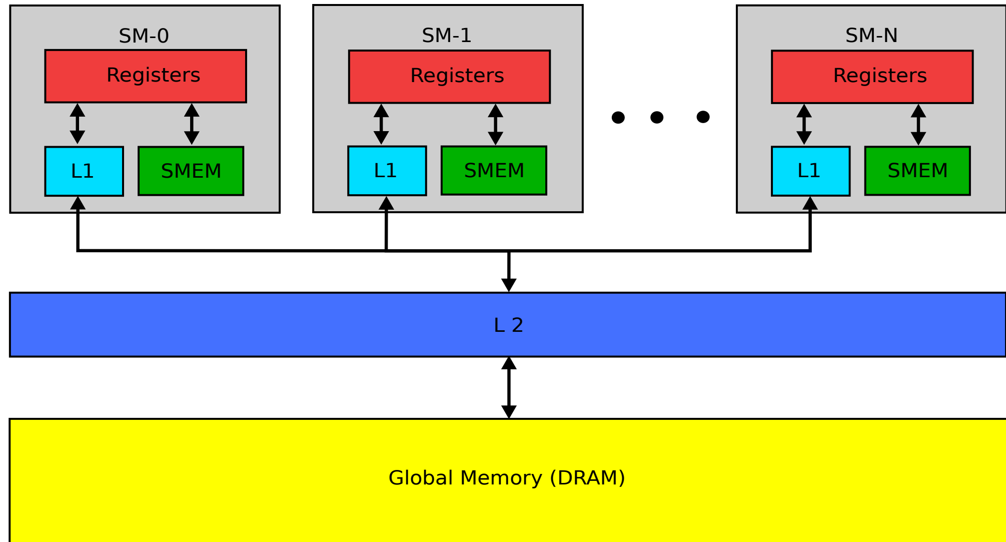

Figure 3.1: The hardware components of a CUDA capable GPU from NVIDIA™ with Compute Capability 2.x, based on [10, p. 26]

In the following paragraphs the usage of these hardware features by the CUDA API is explained. We start with the presentation of the basic execution model. Based on that, we take a look at the thread hierarchy and the memory structure. Then we put the knowledge gained into the context of the infrastructure given by the implementation upon which we built.

3.1.1 Execution Model

In this section we focus on how the GPU can be utilized by the host. The basic process is shown in figure 3.2. At first the parameters for execution are transferred from host to device. This is done by a memory copy from the host's main memory to the device's global memory using API functions. Then, the kernel, a special C function which is executed in parallel by a specified number of CUDA threads [13, p. 7], is launched by the host. The CUDA threads are organized in thread blocks, which are explained in subsection 3.1.2. Synchronization between host and device enables waiting for a kernel to finish. Afterwards the results are fetched from the device's global memory to host's main memory.

3.1.2 Thread Hierarchy

As mentioned in the previous sections, threads[2] are organized in thread blocks. The grid, which holds all the threads executed in one kernel, consists of multiple thread blocks. This

[2] A CUDA thread cannot, in contrast to an ordinary CPU thread, be scheduled independently, but has its own registers and program pointer.

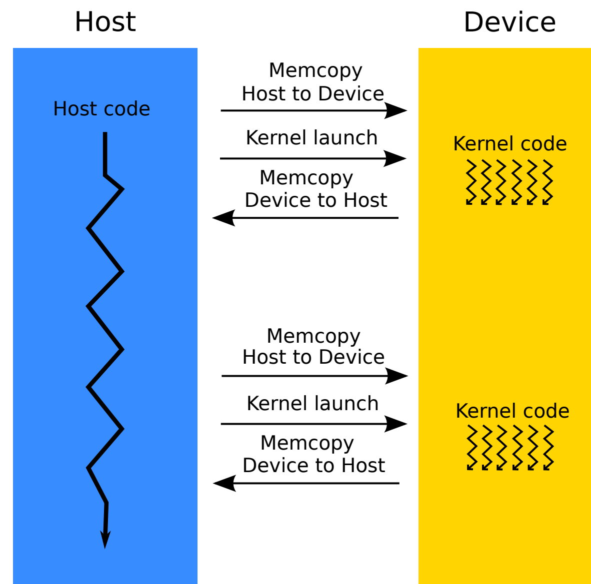

Figure 3.2: The CUDA Execution Model [7, p. 14] based on [13, p. 12]

hierarchy is shown in figure 3.3. The number of threads in a thread block can be specified at kernel launch time and is limited by the Compute Capability. The number of thread blocks, which is limited as well, is normally calculated from the number of threads used in the kernel launch and the number of threads per thread block. The concept of warps is fundamental to understand how the size of a thread block should be chosen.

Multiple thread blocks are assigned to a multiprocessor. These thread blocks are partitioned into groups called warps. These warps are scheduled by the warp scheduler of a SM. They start at the same program address and execute exactly the same instructions. If a condition occurs, in which different branches are chosen, all branches are executed sequentially, which massively decreases performance.[3] Only the threads which do need to execute the instructions in a given branch actually execute them, the rest idles. This technique is referred to as Single Instruction Multiple Threads (SIMT) by NVIDIA [13, p. 63ff.]. To allow as many threads as possible to be scheduled in parallel on one SM the number of divergent branches in a thread block and the number of registers used by one thread should be minimized.

Threads of one thread block can wait for all other threads of a thread block to reach a specific point in the kernel.[4] This is needed when using shared memory which is introduced in section 3.1.3.[5] Synchronization of threads of the same thread block implies additional overhead. Therefore, to maximize the performance synchronization should be used well-considered.

[3] This situation is referred to as branch divergence.
[4] This concept is known form CPUs as “thread fence” or “memory fence”.
[5] Synchronization of threads of different thread blocks isn’t supported. For cuda kernels it is always assumed, that thread blocks can be executed independently.

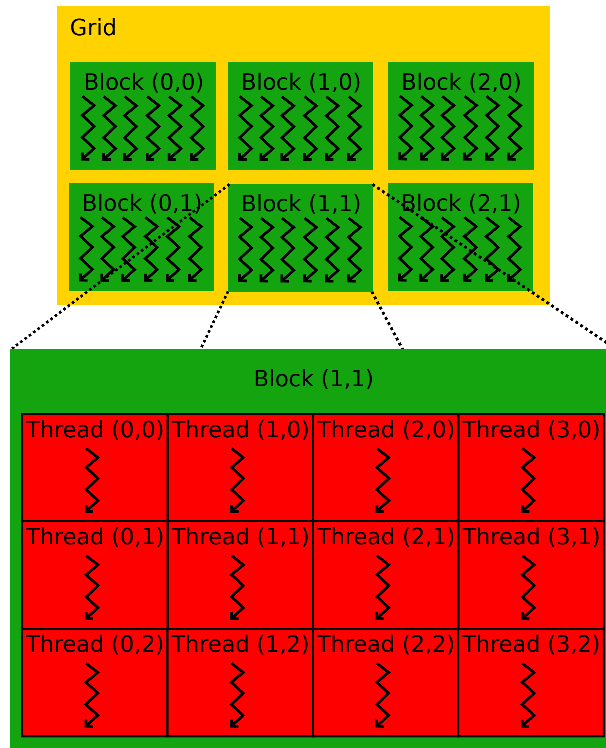

Figure 3.3: The CUDA Thread Hierarchy [7, p. 14] based on [13, p. 9]

3.1.3 Memory Hierarchy

Besides the thread hierarchy the different memories provided by CUDA determine how a task should be parallelized. We present the different memories from local to global scope. See figure 3.4 for an illustration of the memory hierarchy. Each thread has registers which are only accessible by the thread itself. The registers of a SM are distributed among all threads of all active warps on a SM. Because the registers are assigned to a thread for the whole kernel, stopping and restarting a thread[6] can be done with minimum overhead. For devices of Compute Capability between 2.0 and 3.0 the number of registers per thread is limited by 63 and for devices of Compute Capability 3.5 by 255 [13, p. 50]. Thus, the number of registers used by a thread should be minimized to maximize the number of threads which are executed in parallel.[7] The number of registers used should therefore be minimized, as mentioned in

[6]The warp scheduler does this when all threads of a thread block wait for memory and another thread block is executed in the mean while.

[7]In [23] it is shown, that optimizing performance at instruction level can improve the performance even when this reduces the number of threads executed in parallel.

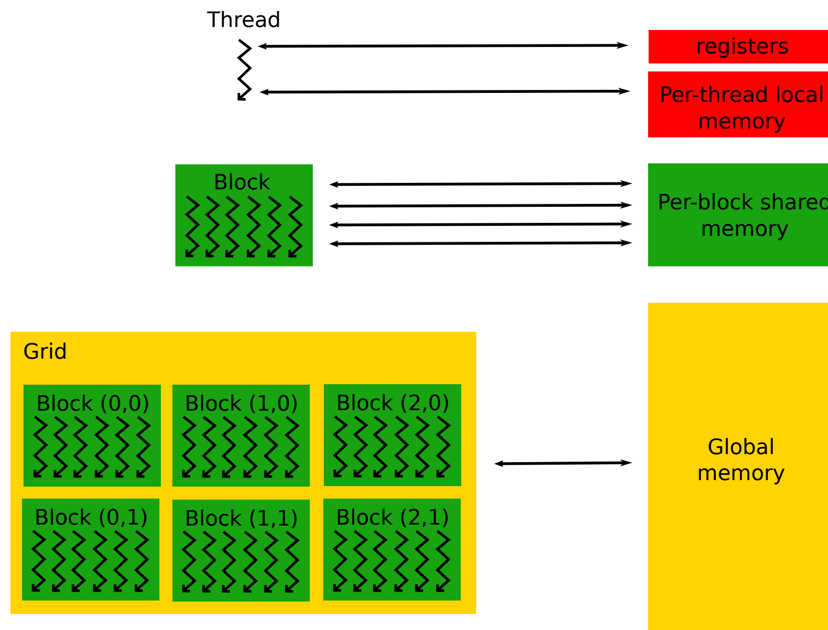

Figure 3.4: The CUDA Memory Hierarchy based on [13, p. 11]

section 3.1.2. If more memory is needed by a thread, than is available in registers, register spilling occurs. This is explained in the introduction to local memory at the end of this section [13, p. 73].

Shared memory is an on-chip memory which is accessible by all threads of a thread block [13, p. 21 ff.]. Since it is on-chip, see figure 3.1, it is very fast if employed optimally, yet it is significantly slower than registers. It has a maximum size of 48KB for devices of Compute Capability ≥ 2.0 and the same physical memory ($64KB$ in total) is used as an L1 cache for global memory access. The division of the memory can be configured at kernel launch time [13, p. 21 ff.].[8]

All threads of a grid can access global memory. This is the GPU's DRAM. It has a high latency and low bandwidth, compared with registers or shared memory [13, p. 73]. As described in [13, p. 71ff.], the memory throughput depends massively on access patterns. If threads of a warp are accessing memory addresses next to each other, the memory requests are merged into one transaction. To achieve access patterns which allow coalesced memory access, the memory layout often differs completely from the one used on a CPU. When adjacent threads access memory addresses which are next to each other the access are coalesced in one memory transaction. If not, multiple memory transactions are required. Figure 3.5 illustrates these two different situations. For details about coalesced and non-coalesced access see [12, p. 24 ff.]. All access from all SMs to global memory are cached by a L2 cache.

[8]There are two possible configuration: 48KB used as cache and 16 as shared memory, or vise versa.

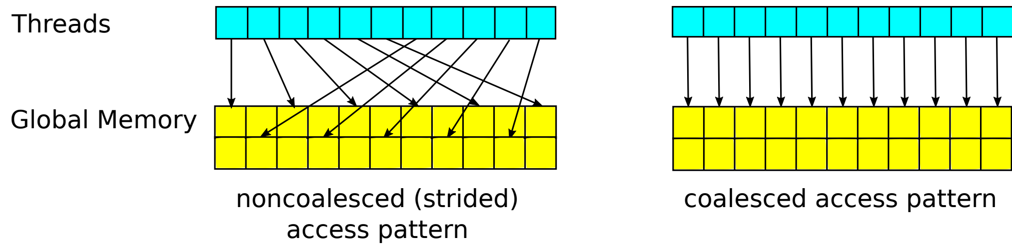

Figure 3.5: Coalesced vs. non coalesced memory access based on [12, p. 28]

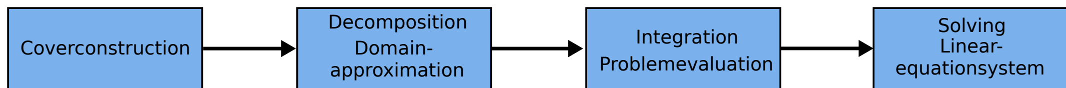

Figure 3.6: The phases to solve a given BVP by the framework

As described before, if a thread uses more variables than registers are available, register spilling occurs. This means, that the variables are stored in local memory. Local memory resides in global memory space which means, that the same latency and throughput deficiency as global memory. The compiler organizes local memory, in a way, that optimizes it for coalesced access [13, p. 74].

In addition to these more or less general memory levels, there is constant memory [13, p. 74 f.] and a texture and surface memory [13, p. 75]. The constant memory has an extra cache, but only can be initialized from host before kernel launch. Therefore, it should be used for constants, as the name suggests.

We can summarize, that the memory hierarchy has significant influence on design of the memory layout used by an application. Algorithms need to be designed in a very specific way to efficiently utilize the given architecture.

3.2 Framework Integration

The implementation presented in the following is based on a PMPUM-implementation, on which [20] and [24] are based. We describe how the framework splits up the approximation of a PDE into steps and which of these steps are implemented using GPGPU. We give a short overview about these steps shown in figure 3.6.

As described in section 2.4.1, a cover needs to be constructed for the given domain. The algorithm for the cover construction can be found in [8, p. 98 ff.]. Afterwards, the generated patches need to be subdivided into inner patches and boundary patches, which have an overlap with the domain boundary. We recall, that we only tackle inner patches using the GPU. Subsequently, the domain needs to be decomposed into cells for which the patches with support on them are known. Different approaches to achieve this decomposition were discussed in

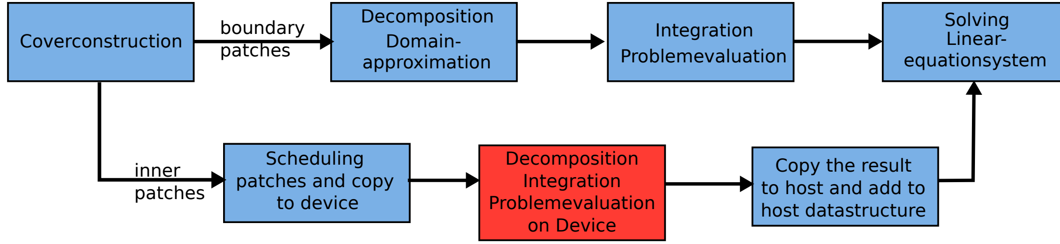

Figure 3.7: The phases to solve a given BVP by the framework, with the CUDA extension

section 2.4. For boundary patches, the intersection with the domain boundary requires the approximation of the domain, which is done using triangulation. Every cell needs to be integrated to calculate the weak form. This is achieved by evaluating the base functions ψ_i^n of the local approximation space $V_i^{p_i}$ and scaling them by the partition of unity φ_i for each integration point[9] on all cells. Afterwards, these base functions are used to evaluate the weak form, given by the PDE to solve. This generates the stiffness matrix A and the right-hand side vector $\hat{f}$ of a system of linear equations. The last step is required to solve this system and write the output.

A parallelization of the given PMPUM-implementation using Message Passing Interface (MPI) is presented in [24]. This parallelization enable the usage of computer clusters for the distributed computation of a solution. Therefore, the patches are distributed among the cluster nodes for decomposition and integration. The intention of the GPU-implementation is, to utilize one or more GPUs to acceleration the computation on a cluster node. The steps accelerated using CUDA are the decomposition of the patches into a integration cells and the integration of the same.

The decomposition and integration is designed as an add-on-library for the existing PM-PUM-implementation. The differences of the workflow for the inner patches in the GPU-implementation and the given PMPUM-implementation are presented in the following. The cover generation provides the neighborhood for each patch. The neighborhoods of all inner patches are transferred to the GPU. Then a CUDA-kernel decomposes the domain and applies the bilinear $a(\cdot,\cdot)$ and linear $l(\cdot)$ form on the generated integration cells. This requires the evaluation of the pum φ_i and base ψ_i^n functions of all patches M_c on the integration cell at all integration points of the integration cell. After the results have been transferred back to the host where they are added to the host's stiffness matrix. This modified workflow for the inner patches is illustrated in figure 3.7.

Since the memory on a GPU is in general much smaller than that of the host not all patches in $\mathcal{W}$ can be decomposed and integrated in one kernel launch. Therefore, a scheduling algorithm, see algorithm 3.1, is employed to be able to compute the task for a larger number of patches. The set of patches decomposed and integrated in the i-th kernel launch is denoted $\mathcal{W}_i$.

[9]The number of the integration points depends on the base functions used.

Algorithm 3.1 Scheduling of neighborhoods for computation on the GPU in pseudo code

```
procedure DISCRETIZE(PointSet)                                   // discretize the pde
    Cover = GENERATECOVER(PointSet)     // generates the cover of patches for the given point set
    for all neighborhood ∈ Cover do              // add the neighborhoods to the schedule
                                  // while it isn't full or all neighborhoods are read
        SCHEDULEAPPEND(neighborhood)
        if SCHEDULEFULLORLASTNEIGHBORHOOD( ) then
            TRANSFERSCHEDULE( )
            RUNKERNEL( )
            TRANSFERRESULT( )
        end if
    end for
end procedure
```

3.3 Memory Layout

As discussed in section 3.1.3 the memory layout has massive influence on the performance. First, we focus on the layout for the parameters and then on the layout for the results.

For each patch ω_i contained in any neighborhood the following information needs to be transferred:

- the geometric extent of the patch given by its center x_i and stretch h_i
- type of the function space used on the patch
- the resolution[10]

There are multiple layouts which should be considered. When using a CPU, data of a patch should be stored in one piece and redundancies should be kept at a minimum. The layout for GPUs however should be quite different as mentioned in section 3.1.3. We need to anticipate from section 3.4, that we use a thread per neighborhood, to understand the choice for the presented memory layout. To allow access from all threads to be coalesced the layout should store data items of all patches in one chunk. When two threads access the same information on different patches at the same time, their memory request can be handled in a single memory transaction.

The affiliation of patches to a neighborhood could be stored separately from the patches, this however would require additional effort in the preprocessing. We will clarify this additional effort. The resulting data structure would store the information for all patches in one location and the neighborhood information in another. A neighborhood, in this case is a list of indices referencing patches. To evaluate the bilinear form $a(\cdot,\cdot)$ on an integration cell the indices of

[10] The resolution is the number of base functions of the function space used. For the polynomial space which is used in following the resolution is given by $b_i = \sum_{i=0}^{p} \binom{d+i-1}{d-1}$.

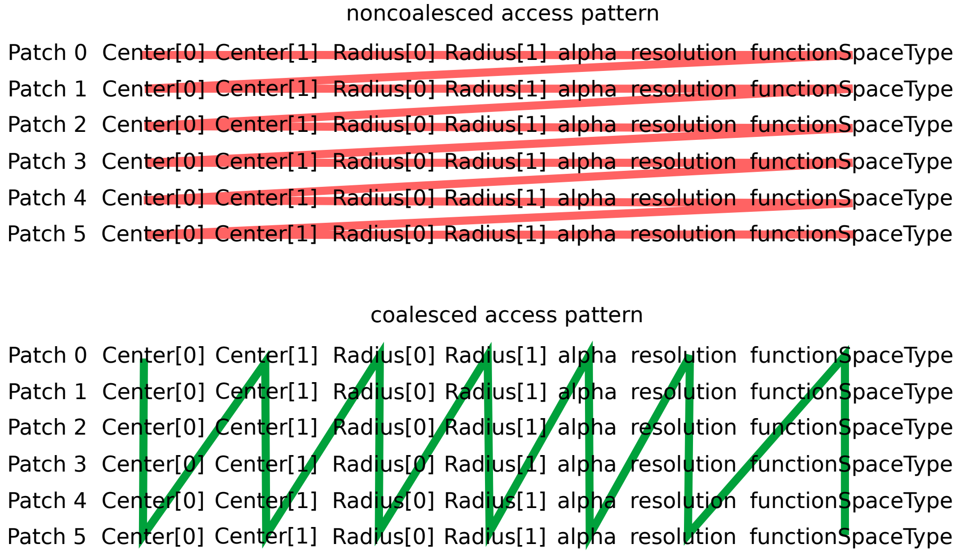

Figure 3.8: Two different memory layouts for the patches $\mathcal{W}_i$ (2D). The red and green lines represent the sorting in the memory. The layout with the red line leads to non coalesced access, while the green access pattern leads to coalesced access

all patches ω_j in the neighborhood of patch ω_i, from which the integration cell was generated, need to be known. Therefore, when scheduling is applied, one schedule computes the integral on the domain of a subset of the patches $\mathcal{W}_i \subset \mathcal{W}$. However, the neighborhood information of the patches, which are referenced in the current schedule, are required. This requires to copy the neighborhood information of theses additional patches. The set of these patches however can only be determined after the set of patches which are decomposed and integrated is known and thus also the set of additional patches can be computed. We will recall this version of the memory layout in section 4.10.1, where an illustration of the information required can be found.

By storing all patches for a neighborhood separately, we avoid this overhead in the preprocessing. This however duplicates the information on the patches which are in multiple neighborhoods of the current schedule. This leads to additional memory requests, but we expect this effect not be dramatic since we ensured coalesced access as described before. Figure 3.8 compares the resulting memory layout to a memory layout typically used on CPUs.

For the results a similar approach is required. First, we present a memory layout in CPU-fashion, used in the framework on which the implementation is build. Based on that, we present the layout used for the GPU-implementation.

Each row of the matrix A, resulting from the evaluation of the bilinear form $a(\cdot,\cdot)$, represents a patch ω_i and its neighborhood C_i. For each patch in the neighborhood ω_j, there is one non-zero entry in the row. An entry for a patch is a matrix $A_{(i,n),(j,m)}$ containing the scalar product of the base function values of the two patches a$(\varphi_i\ \psi_i^n, \varphi_j\ \psi_j^m)$. The number of rows

and columns of this matrix depend on the number of base functions b_i and b_j of the function spaces $V_i^{p_i}$ and $V_j^{p_j}$ defined on the two patches ω_i and ω_j.

The vector, resulting from the evaluation of the linear form, is organized as follows. Each component of the vector belongs to one patch ω_i. Every component itself is a vector, with the size of the number of base function b_i of the function space $V_i^{p_i}$, used on the patch ω_i. For multiple linear forms or bilinear forms multiple vectors respectively matrices would be stored.

Since this memory layout leads to non-coalesced access for a GPU-implementation a different approach is required. Since every thread operates on one neighborhood, we store the result per neighborhood and accumulate the results in a postprocessing step. The data should be organized in a way, that all threads access adjacent memory locations concurrently. Therefore, the innermost index is the index of the neighborhoods. The other indices are organized in the order in which the kernel, see section 3.4, operates on them. For details on this memory layout see chapter A in the appendix. A memory layout which avoids redundancies is discussed in section 4.10.1.

3.4 Kernel Design

The kernel needs to decompose the domain of the inner patches Ω_I and integrate the resulting integration cells. Each thread operates on a patch ω_i and its neighbors C_i. This gives the opportunity to abandon synchronization completely. The Tensor-Product Approach, presented in section 2.4.2, is employed to generate the decomposition of the patch.

Algorithm 3.2 Kernel workflow

```
procedure KERNEL(neighborhood)              // assemble the stiffness matrix on GPU
    decomposition = DECOMPOSE(neighborhood) // decompose the neighborhood employing
algorithm 2.2
    for all IntegrationCell ∈ decomposition do    // get all integration cells in succession
        INTEGRATIONCELL.INTEGRATE( )
    end for
end procedure
procedure INTEGRATIONCELL.INTEGRATE( )
    for all IntegrationPoint ∈ IntegrationCell do
        EVALUATEWEIGHTS( )
        EVALUATEBASEFUNCTIONS( )
        EVALUATEOPERATORS( )
    end for
end procedure
```

As presented in algorithm 3.2, each resulting integration cell needs to be integrated. The number of integration points used for the integration depends on the function space V^{P_i} used on

the patches with support on the cell.[11] The reference integration points, of the Gauss–Legendre quadrature used, are stored in constant memory and transformed to the cell. The first step of the evaluation at a given integration point is the evaluation of the weight functions ω_i for all patches. Afterwards, the base functions ψ_i^n for the function spaces V^{P_i} on all patches are evaluated. Then they are scaled using the partition of unity $\varphi_i(x) = \frac{W_i(x)}{\sum_{\omega_k \in C_i} W_k(x)}$, which is calculated from the weight functions, evaluated in the previous step. The evaluation of the linear $l(u)$ and bilinear $a(u, v)$ forms generate the resulting vectors and matrices.

3.5 Implementation Properties and Limitations

Every implementation requires design decisions which have major impact on the results. In the following the most important properties and limitations of the implementation are presented:

1. The polynomial degree used in V^{PU} is set at compile time. This disqualifies the usage of p-refinement.[12]

2. The space dimension is set at compile time, as well. Since the PMPUM-framework utilizes the same approach, this is no real limitation.

3. Since all threads operate on their own result data, a postprocessing step is required to accumulate the results. Each CUDA thread writes a matrix composed of a row and column for each neighbor. Every entry is a matrix again with the dimensions of the function spaces as number of rows respectively columns. This leads to multiple matrices, one for each neighbor, which need to be accumulated to form the required result where only one matrix represents the bilinear form of two overlapping patches. Strategies how to avoid this are discussed in section 4.10.1.

4. A thread block size of 128 is chosen but changing this to reasonable values has no significant influence on the performance.

5. For the evaluation of the bilinear form $a(\cdot, \cdot)$ and the linear form $l(\cdot)$, all base functions ψ_i^n of all patches with support on a patch C_i need to be evaluated. The memory required for these base functions is given by $b_i \cdot n_{\omega C}$. Here b_i is the number of the base functions and $n_{\omega C}$ is the number of patches with support on an integration cell. As mentioned in section 2.4.1, $n_{\omega C}$ if limited by 4 for uniform h-refinement. The number of base functions b_i, is given in table 3.1, depends on the polynomial degree p of the local approximation spaces $V_i^{p_i}$. As mentioned in section 3.1.3 at most 32 KB of the Shared Memory can be used as L1 cache for devices of Compute Capability ≥ 2.0. Together with at most 63 registers the size of fast memory per thread is limited to $48KB/128 + 4B \cdot 63 = 636B$.[13]

[11] Only the polynomial space is analyzed in the following, note that the PMPUM supports enrichment functions which require additional integration points.

[12] p-refinement is refinement by increasing the polynomial degree used.

[13] This is only the case if no information is shared by the threads of a thread block. If information is shared each thread has $4B \cdot 63 = 252B$ of memory and in addition to that, every thread block has 48KB of memory. For an approach that enables sharing of information between threads see section 4.10.4

Table 3.1: The memory requirement of the base functions ψ_i^n for all patches with support on a cell

p	#scalar (2D)	size in Bytes (double)	size in Bytes (single)
0	4	32	16
1	12	96	48
2	24	192	96
3	40	320	160
4	60	480	240
5	84	672	336

The maximum of the Shared Memory/L1 cache is used as cache and is shared by all threads of a thread block with size 128. Since additional memory is needed, i.e. the decomposer requires to store all split points, the size of the available fast memory may limit the overall performance.

6. The implementation uses up-to-date features such as dynamic memory allocation[14] and separate compilation to decrease build time. Due to the usage of these features CUDA 5.0 and devices with Compute Capability >= 2.0 are required.

[14] Since dynamic memory allocation should be minimized for high performance CUDA code, it's only used for the instantiation of structures which are depending on problem parameters, such as the function space or operators used, except those listed.

4 Results

In this chapter experiments are presented which allow the evaluation of the implementation and discussion of recommendations for further improvements of the implementation. First, we present the hardware and metric used and afterwards we present the different experiments that were conducted.[1] The chapter closes with a summary of the results and the recommendations for further improvements.

4.1 Hardware and Metrics

Multiple hardware configurations were used for the experiments. If not noted otherwise the configuration given in table 4.1 was used. To investigate the performance on other GPUs we consider the GPUs listed in table 4.2.

4.1.1 Hardware

Table 4.1: Reference system configuration

Operating System	Ubuntu Linux (12.04.2 LTS) with kernel 3.2.0-38-generic
CPU	Intel i7-2600K @ 3.4GHz
Main memory	16 GiB
GPU	NVIDIA GeForce GTX 560 Ti with 2 GiB RAM
MPI compiler (host code)	mpicxx (-std=c++0x -O3 -DNDEBUG)
CUDA compiler (device code)	5.0, V0.2.1221 (-Xcompiler ,\"-O3\",\"-DNDEBUG\" -arch=sm_21 -DNDEBUG)[a]

[a] the `--arch` flag is set according to the Compute Capability given in table 4.2 when using other GPUs

The table provides the most relevant characteristics for performance measurements. It shows that the peak DP performance[2] of the Geforce GTX 560 Ti is about 3.4 times higher than that of a single core of the Intel i7 2600k. For SP the peak performance is however about 41.4 times higher. We note, that the NVIDIA™ K20 outruns the Intel i7 2600k (using all 4 cores) in DP performance. The NVIDIA™ K20 is 10.75 times faster than an Intel i7 2600k utilizing

[1]A framework for automated installation, execution and evaluation was written in python.
[2]floating point performance is measured in giga floating point operations per second (gflops).

Table 4.2: Performance of different architectures

Metric	Intel i7 2600k (single core)	Geforce GTX 560 Ti	NVIDIA™ K20
Peak SP performance in gflops	30.5[a]	1263[cd]	3520[f]
Peak DP performance in gflops	30.5[a]	105[e]	1170[f]
Memory bandwidth in GB/s	21[b]	128[c]	208[f]
Streaming Multiprocessors	–	8[g]	13[g]
Compute Capability	–	2.1[h]	3.5[h]

[a] only when using a single core with maximum turbo, otherwise 27.2 gflops per core [5]
[b] according to [6]
[c] according to [17]
[d] the single precision performance can be calculated from the shader clock frequency, given as *Processor Clock*, and the number of cuda cores both given in [17], since FMA-instructions are used the results needs to be multiplied by 2
[e] according to [21] the GTX 560 Ti's double precision performance is 1/12 of the single precision performance
[f] according to [16]
[g] can be retrieved via the CUDA API
[h] according to [15]

all 4 cores. For SP however the performance of the Geforce GTX 560 Ti and the NVIDIA™ K20 is higher than that of the Intel i7 2600k.

4.1.2 Metric

For the experiments presented in the following sections the following metric is used for the performance of an implementation:

$$\frac{t/dof}{dof}$$

Here t is the duration in seconds and dof is the number of degrees of freedom. The number of degrees of freedom for the PMPUM is given by:

$$N \cdot b_i = (2^d)^l \cot \sum_{i=0}^{p} \binom{d+i-1}{d-1}$$

Here N is the number of patches, which depends on the space dimension d and the level l, and b_i is the number of base functions per patch which depends on the polynomial degree p used.[3]

[3]See section 2.5 for a derivation of N and b_i.

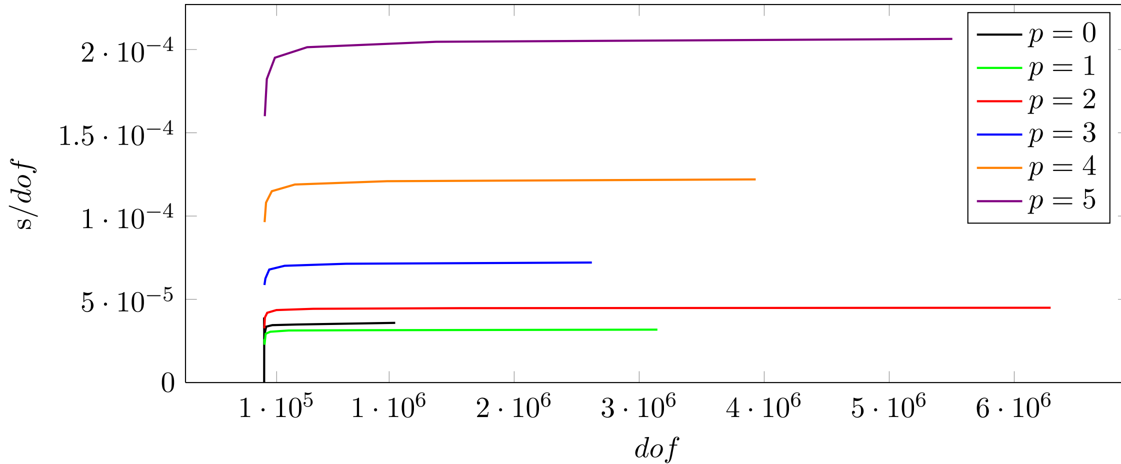

Figure 4.1: The performance of the integration on Ω_I of the CPU-implementation $\mathcal{P}_{CPU}$. See table B.4 in the appendix for the data on which this figure is based.

4.2 Experiment: Overall Performance

The first experiment was conducted to estimate the performance of the implementation. To that end, the performance of the integration of the domain of the inner patches Ω_I for discretization of the model problem (presented in section 2.1) in 2D is examined.[4] In this experiment we use the Geforce GTX 560 Ti and the Intel i7 2600k as reference. Since DP is used we expect a maximum performance boost by a factor of 3.4 because we use a single core of the Intel i7 2600k. Figure 4.1 shows the performance $\mathcal{P}_{CPU}$ of the CPU-implementation.

We compare it to the performance $\mathcal{P}_{GPU}$ achieved on the Geforce GTX 560 Ti, which is shown in figure 4.2. For an easier comparison figure 4.3 shows the quotient $\mathcal{P}_{CPU}/\mathcal{P}_{GPU}$. We see, that the performance of the GPU-implementation is higher than the performance of the CPU-implementation. We note, that for the best case (polynomial degree 1) the performance of the GPU-implementation is about 2.5 times higher than the one of the CPU implementation. Since we expected a maximum of 3.4, see section 4.1.1, we note, that an improvement of the performance of 70% compared to the theoretical maximum is reached for polynomial degree 1. We can however determine more characteristics from the figures.

1. The figures only show data for higher levels l since for lower levels not enough threads can be spawned to utilize GPU.[5] This suboptimal utilization is caused by the fact, that a GPU is organized in SMs which operate on thread blocks. If not enough thread blocks are launched to utilize all SMs the performance drops. Since we launch a thread for each patch and we use thread blocks of size 128 and the Geforce GTX 560 Ti has 8 SMs we

[4] To verify that nothing was omitted during the experiments the overall duration for the discretization was measured. The results of these measurements, which do include the integration of the whole domain Ω instead of only the inner domain Ω_I, are given in the tables B.1, B.2 and B.3 in the appendix.

[5] see the corresponding tables B.4, B.5 and B.6 for data of lower levels.

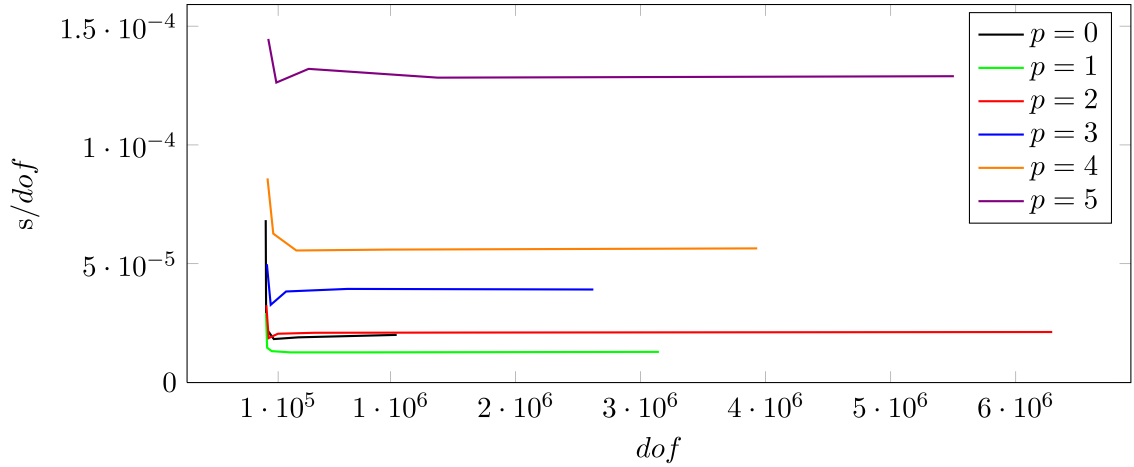

Figure 4.2: The performance of the integration on Ω_I of the GPU-implementation $\mathcal{P}_{CPU}$. See table B.5 in the appendix for the data on which this figure is based.

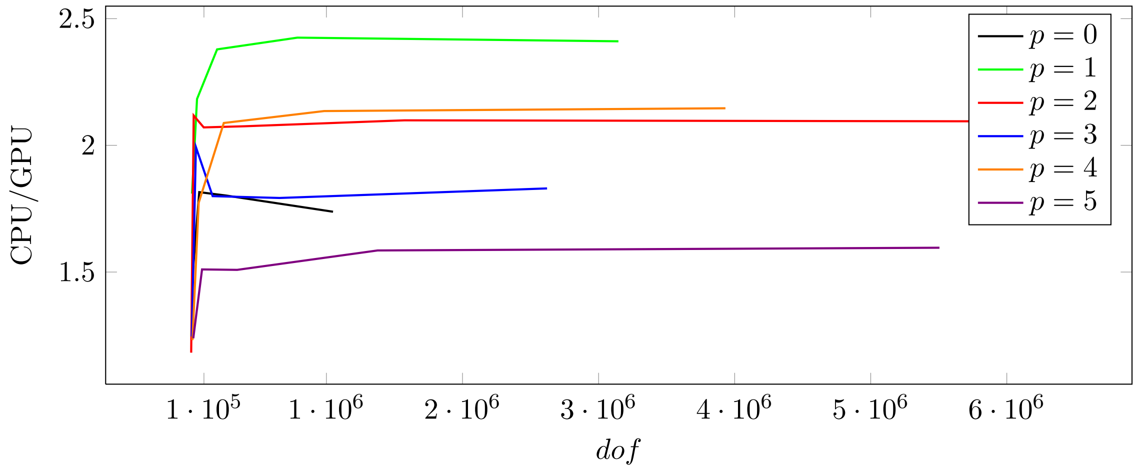

Figure 4.3: The relative performance of the integration on Ω_I of the CPU and GPU-implementations $\mathcal{P}_{CPU}/\mathcal{P}_{GPU}$. See table B.6 in the appendix for the data on which this table is based.

need at least $128 \cdot 8$ patches, to be computed by the GPU. This means, that only for level $l \geq 5$ we reach acceptable utilization of the GPU in the 2D case. Since the duration of the integration on lower levels is irrelevant for the overall performance, in the following, we only examine the performance for level $l \geq 5$.

2. As stated in section 2.5 the complexity of the integration depends on the level l used. It should grow by a constant factor and therefore the metric of the value is expected to be constant. Figure 4.3 confirms this for all polynomial degrees, except for polynomial degree 0, for high levels.

Table 4.3: The ratio of the duration of the GPU-implementation between consecutive levels. See table B.5 for data on which this table is based.

P/L	6/5	7/6	8/7	9/8	reference value
0	1.71	3	3.33	4.16	4
1	2	3.61	3.85	4	4
2	2.3	4.39	4.08	4.02	4
3	2.63	4.69	4.11	3.97	4
4	2.92	3.55	4.03	4.04	4
5	3.49	4.18	3.89	4.02	4

3. As stated in section 2.5 the complexity of the integration depends on the polynomial degree. We expect a growing effort for increasing polynomial degree p. As we see in figure 4.2 the performance doesn't seem to correlate in the expected way. For polynomial degree 1 the performance is better than for polynomial degree 0. Therefore, we need to examine this further.

Overall, the performance reached when using DP is good for the Geforce GTX 560 Ti (Geforce GTX 560 Ti). However, the performance doesn't correlate with the complexity given in section 2.5. In the following experiment, we therefore analyze the performance depending on the problem parameters.

4.3 Experiment: Performance depending on Problem Parameters

In the previous experiment we found, that the performance depending on the polynomial degree doesn't seem to grow according to the values determined in section 2.5. Thus, in this experiment we examine the performance of the implementation depending on these parameters. We choose the same setup and data as in the previous experiment.

We recall from section 2.5, that the complexity grows by a factor of 4 for an increase of the level l by 1 in 2D. Table 4.3 shows these factors for the GPU-implementation.

As we see for high levels, the duration of the GPU-implementation correlates nearly perfectly with the complexity specified.

For the dependency of the duration of the GPU-implementation on the polynomial degree p we recall, that for an increase of the polynomial degree from $p-1$ to p an increase of the duration of $\left(\frac{\lceil\frac{p+1}{2}\rceil}{\lceil\frac{p}{2}\rceil}\right)^2 \cdot \frac{2+p}{p}$ is expected. Table 4.4 shows these factors for the GPU-implementation.

The factors differ from those expected. Especially the factor between polynomial degree 0 and 1 isn't as high as expected. There are multiple reasons for this behavior:

1. The number of integration points in the actual implementation is chosen to be $n_{NI} = (p+1)^d$ instead $n_{NI} = \left(\frac{p+1}{2}\right)^d$ which was derived in section 2.5. This was done to utilize

Table 4.4: The ratio of the duration of the GPU-implementation between consecutive polynomial degrees. See table B.5 for an the data on which this table is based.

P/L	5	6	7	8	9	$n_{NI} = \left(\frac{p+1}{2}\right)^d$	$n_{NI} = (p+1)^d$
1	1.29	1.5	1.81	2.08	2	3	12
2	2.22	2.56	3.11	3.3	3.31	8	4.5
3	2.55	2.91	3.11	3.13	3.1	1.66	2.96
4	2.59	2.87	2.18	2.13	2.16	3.38	2.34
5	2.36	2.82	3.33	3.21	3.2	1.4	2.01

the same number of integration points as the CPU-implementation to reach comparability. The resulting reference values for both formulas for n_{NI} are given. We can inspect, however that there is no obvious correlation with both reference values.

2. The influence of the postprocessing step, described in section 3.5, might decrease with the required total effort. Thus in the experiment presented in section 4.4 we investigate which part of implementation is most accountable for the duration.
3. The mapping of the algorithm to CUDA might limit the performance which can be reached. The amount of fast memory per thread[6] is, however, limited for all possible implementations. Since the base functions need to be available for each pair of patches on an integration cell, which also applies to all possible implementations, it will be hard to design an efficient mapping for higher polynomial degree p.

Altogether, the performance of the implementation correlates with the complexity given in section 2.5 for the level l. For the polynomial degree p, however, we don't see the expected correlation and thus we need to investigate this further. Therefore, in the next experiment we analyze the performance of the different steps of the GPU-implementation.

4.4 Experiment: Performance of different Steps of the GPU-Implementation

In this experiment, we analyze the different steps of the implementation, as presented in section 3.2. This is especially interesting since it might be the reason for the unexpected trend of the performance for lower polynomial degrees. The GPU-based discretization is split, as presented in section 3.2, into the following steps:

- the preparation consisting of the copy of the neighborhoods to the GPU's global memory
- the kernel which decomposes the inner domain Ω_I and performs the integration required to evaluate the bilinear $a(\cdot, \cdot)$ and linear $l(\cdot)$ form.

[6] As described in section 3.5.

Table 4.5: The ratio of the duration of the preprocessing and the whole GPU-implementation. See tables B.5 and B.7 for the data on which this table is based.

P/L	5	6	7	8	9
0	0.14	$8.33 \cdot 10^{-2}$	$2.78 \cdot 10^{-2}$	$6.67 \cdot 10^{-2}$	$5.81 \cdot 10^{-2}$
1	0	0	$3.08 \cdot 10^{-2}$	$1.6 \cdot 10^{-2}$	$3.3 \cdot 10^{-2}$
2	0	$2.17 \cdot 10^{-2}$	0	$7.28 \cdot 10^{-3}$	$9.97 \cdot 10^{-3}$
3	0	$7.46 \cdot 10^{-3}$	$3.18 \cdot 10^{-3}$	$3.1 \cdot 10^{-3}$	$3.02 \cdot 10^{-3}$
4	0	0	$2.2 \cdot 10^{-3}$	$1.82 \cdot 10^{-3}$	$2.03 \cdot 10^{-3}$
5	0	0	$4.4 \cdot 10^{-4}$	$8.49 \cdot 10^{-4}$	$6.34 \cdot 10^{-4}$

Table 4.6: The ratio of the duration of the kernel and the whole GPU-implementation. See tables B.5 and B.8 for the data on which this table is based.

P/L	5	6	7	8	9
0	0.14	0.17	0.39	0.35	0.34
1	0.33	0.44	0.52	0.52	0.52
2	0.65	0.57	0.59	0.61	0.6
3	0.8	0.68	0.72	0.72	0.71
4	0.85	0.77	0.74	0.73	0.73
5	0.89	0.85	0.86	0.85	0.85

- the postprocessing which copies the result back to the host and adds the result to the global data structures. Since for each neighborhood the results are stored separately, this step implies the summation of 9 matrices instead of the copy of 1 matrix for each neighbor in a neighborhood.

Since the preprocessing consists only of copies the duration is expected not to be significant. The postprocessing however might have influence on the performance because of the required accumulation of the results.

Table 4.5 shows the proportion of the duration of the preparation step of the overall duration. Tables 4.6 and 4.7 show the same proportion for the duration of the kernel respectively the postprocessing.

The first table shows that the preprocessing does not account for a significant proportion of the overall duration. As expected the situation is different for the postprocessing. Table 4.7 shows that for smaller polynomial degree p the duration of the postprocessing dominates the duration of the kernel.

Recalling the reason for this experiment from the previous experiment the distorted behavior for lower polynomial degree is explained by the fact, that the duration of the postprocessing dominates, that of the kernel. Since, other strategies which avoid the host-based accumulation of the results affect the performance of the preprocessing and the kernel, it's hard to tell which

Table 4.7: The ratio of the duration of the postprocessing and the whole GPU-implementation. See tables B.5 and B.9 for the data on which this table is based.

P/L	5	6	7	8	9
0	0.71	0.75	0.56	0.57	0.57
1	0.67	0.56	0.45	0.44	0.43
2	0.35	0.41	0.4	0.38	0.39
3	0.2	0.31	0.28	0.28	0.28
4	0.14	0.23	0.26	0.27	0.26
5	0.11	0.15	0.14	0.15	0.15

effect on the behavior would result from applying such a strategy. For details on a strategy which does not require host-based accumulation see section 4.10.1.

Another reason for the unexpected behavior of the duration should be considered for larger polynomial degree. In section 3.5 we discussed, that local resources should be minimized to maximize the performance. For increasing polynomial degree, however, the number of base functions and thus the requirements for local resources increases. If these are not available the performance drops due to many uncached and therefore slow global memory access.

In total the multiplication of the result data structure, which was chosen to avoid atomic memory access on the GPU, shifts a significant amount of computations to the CPU. For smaller polynomial degree p this even dominates the performance. A strategy to improve this is presented in section 4.10.1. In the next experiment, we present results with single precision which decreases the requirements for local resources, which seem to be the performance limiter for larger polynomial degree. In addition to that, we analyze data from the profiler for CUDA in the experiment, presented in section 4.6, to check this assumption.

4.5 Experiment: Single precision Performance

In the previous section we guessed, that excessive usage of local memory could be the reason for the behavior for larger polynomial degree p. The memory requirements depend on the precision used. Thus we expect, that using SP instead of DP should result in an increased performance. This effect should be larger since as shown in table 4.2 using SP instead of DP on the Geforce GTX 560 Ti should result in a much higher performance. In fact it should increase the performance advantage over the CPU(Intel i7 2600k) from factor 3.4 to factor 41.4. In this experiment we thus utilize SP. Table 4.8 shows the ratio of the duration of the CUDA-kernel using SP and DP.

The table shows, that the performance using SP and DP is identical, except from differences, that are caused by measurement errors.[7] Thus in the following we discuss possible reasons for this unexpected result:

[7]See tables B.11, B.12 and B.13 for details on the duration of the different steps when using SP.

Table 4.8: The ratio of the duration of the kernel for SP and DP. See tables B.8 and B.12 for the data on which this table is based.

P/L	5	6	7	8	9
0	0	1.5	1	1	1.02
1	1	0.88	1	0.98	1
2	1	1	1.01	0.99	1.01
3	0.98	1.02	1.04	0.95	1.01
4	1.02	0.98	1	1	1
5	1.06	1	0.99	1	1

1. The occupancy[8] could be low and limit the performance in the SP case. In the DP case this would not affect the performance since only $1/12$ of the floating point units can be utilized and thus for a lower occupancy, the limiter of the performance would still be the number of floating point units of the ALU.
2. The memory bandwidth or more likely the memory access patterns could be the reason why the performance isn't higher for SP than for DP. The memory bandwidth is the same for SP and DP and thus bad access patterns may not affect the DP performance but limit the SP performance. Thus the SP performance is an indicator, that the implementation is memory bound. However the access patterns seem to be correct in theory.[9] Thus accesses to local memory could be the problem since it's physically the same memory.
3. As mention in the previous point, the usage of too many local resources reduces the performance. When more memory is used by a thread than is available in registers, local memory is used. Local memory, however, is, as mentioned in section 3.1.3 only coalesced global memory. Thus using local memory means, that data that we would like to store in very fast registers is swapped out to the very slow global memory. When this situation occurs extensively, is has massive impact on the performance. Thus, reducing the amount of local resources may increase the performance drastically. For a recommendations how to achieve that, see sections 4.10.3 and 4.10.4.

We summarize, that this experiment showed, that the performance is limited by the occupancy and that the implementation is likely to be memory bound. In the following experiments we therefore examine if this statement holds using the profiler for CUDA.

[8] occupancy is the ratio of the active warps on a SM of the maximum number of warps supported by the SM. Therefore, it limits the utilization of the SM's arithmetic and logic unit (ALU) [13, p. 69]. However, the relation between occupancy and performance is complex, for details see [23].

[9] For an improved memory layout for the parameters see section 4.10.1.

Table 4.9: The occupancy of the SMs when executing the kernel for the decomposition and integration of Ω_I.

P/L	5	6	7	8	9
0	$7.68 \cdot 10^{-2}$	0.32	0.32	0.32	0.33
1	$7.81 \cdot 10^{-2}$	0.31	0.32	0.32	0.32
2	$7.85 \cdot 10^{-2}$	0.32	0.29	0.32	0.31
3	$7.8 \cdot 10^{-2}$	0.31	0.28	0.28	0.29
4	$7.77 \cdot 10^{-2}$	0.17	0.22	0.22	0.22
5	$7.65 \cdot 10^{-2}$	0.11	0.12	0.12	0.12

4.6 Experiment: Profiler Data Analysis

The experiment, presented in the previous section, hypothesizes, that the implementation is memory bound. Therefore, in this experiment we analyze data collected using the profiler provided by NVIDIA™ [18]. The first guess in the previous experiment was, that occupancy may limit the performance. This assumption is reasonable since, as discussed in section 3.5, the implementation requires a serious amount of local resources and therefore only less warps can be executed in parallel on a SM in parallel.

Table 4.9 shows the occupancy measured by the profiler for the implementation.

We see, that the assumption was correct and the occupancy for the implementation is only 33% for polynomial degree 0 and decreasing for increasing polynomial degree. The correlation between occupancy and performance is complex, for details see [23, p. 25ff.]. We note however that for an implementation for that threads are likely to be stalled[10] often occupancy has an significant influence. Therefore, the peak performance, presented in section 4.1.1 is likely to be reduced, but the exact factor is unknown. As discussed in the previous experiment the performance isn't influenced in the same way for DP as for SP, because only $1/12$ of the SP units is available in DP in the ALUs of the SMs of the Geforce GTX 560 Ti. However, if only this limitation would apply the performance of the implementation could still be good when using SP.

The profiler can compute additional metrics which can identify bottlenecks in an application. Since we suspect the memory access to be our main bottleneck, we are especially interested in the metrics concerning the efficiency of the global memory access. Unfortunately, the documentation of the profiler isn't consistent with the profiler. Therefore, some metrics could not be computed[11] and we focus on the analysis of the data available. First, we investigate if

[10] A thread is stalled on unsatisfied data dependencies, see [23] for details.

[11] The events described in [18, p. 39ff.] to compute the metrics `gld_efficiency` and `gst_efficiency` for devices of Compute Capability 2.x are not found when executing `nvprof --query-events`. The Visual Profiler [18, p. 4ff.] failed to compute the global store efficiency. The global load efficiency, however, could be computed for the implementation. All metrics could be computed by the Visual Profiler for other tested applications.

Table 4.10: The ratio of the issued instructions of the kernel between consecutive levels. See table B.14 for the data on which this table is based.

P/L	6/5	7/6	8/7	9/8	reference value
1	4.62	4.66	4.04	4.04	4
2	4.45	4.18	4.14	4.04	4
3	4.42	4.15	4.08	4.04	4
4	4.17	4.18	4.07	4.03	4
5	4.22	4.15	4.06	4.03	4

Table 4.11: The ratio of the issued instructions of the kernel between consecutive polynomial degrees. See table B.14 for the data on which this table is based.

P/L	5	6	7	8	9	$n_{NI} = \left(\frac{p+1}{2}\right)^d$	$n_{NI} = (p+1)^d$
1	3.5	3.07	2.21	2.57	2.55	3	12
2	3.54	3.41	3.06	3.13	3.13	8	4.5
3	3.18	3.16	3.14	3.09	3.09	1.66	2.96
4	2.76	2.61	2.62	2.61	2.61	3.38	2.34
5	2.43	2.45	2.44	2.43	2.43	1.4	2.01

the number of instruction issued shows the same dependency on the level l and polynomial degree p as the duration of the GPU-implementation discussed in section 4.3.

From table 4.10 we conclude, that the total number of instructions correlates with the duration, required by the GPU-implementation depending on the level l in the expected way. For the polynomial degree p we see, that the number of instructions, given in table 4.11, correlates with the expected values derived in section 2.5, except for polynomial degree $p = 0$, rather than with the duration of the complete GPU-implementation. This confirms, that the postprocessing has massive influence on the performance, as mentioned in section 4.4.

In section 4.5 we postulated the assumption that the implementation is memory bound. From section 3.5 we recall that this may be caused by register spilling. The profiler provides the ratio of instructions which are executed due to waiting for local memory, which is used in case of register spilling, of the total instruction count. The used metric `local_replay_overhead` is shown in table 4.12.[12]

The ratio is constant for increasing level l and for increasing polynomial degree p. This result seems to be unexpected, but the reason for that seems to be the metric. We assume, that the metric doesn't measure the case when all thread blocks are waiting for memory. Thus, the only metrics to measure this would be the `gld_efficiency` and the `gst_efficiency`, which are unavailable.

[12]Unfortunately `local_replay_overhead` is not defined in the documentation, but seems to be the number of instructions, that are issued by all threads of a warp waiting for other threads of the same warp to finish a memory request.

Table 4.12: The ratio of instructions issued due to waits for local memory of the total instructions issued. See tables B.14, B.15 and B.16 for the tables on which this table is based.

P/L	5	6	7	8	9
0	0.11	0.15	$8.61 \cdot 10^{-2}$	0.1	0.1
1	$7.51 \cdot 10^{-2}$	0.14	0.12	0.11	0.11
2	$4.68 \cdot 10^{-2}$	0.13	0.12	0.12	0.12
3	$3.28 \cdot 10^{-2}$	0.13	0.11	0.12	0.12
4	$3.27 \cdot 10^{-2}$	0.11	0.12	0.12	0.12
5	$3.81 \cdot 10^{-2}$	$9.01 \cdot 10^{-2}$	0.11	0.11	0.11

Table 4.13: The ratio of non divergent branches and all branches in the kernel for the decomposition and integration of Ω_I. See tables B.17 and B.18 for the data on which this table is based.

P/L	5	6	7	8	9
0	1	1	1	1	1
1	1	1	1	1	1
2	1	1	1	1	1
3	1	1	1	1	1
4	1	1	1	1	1
5	1	1	1	1	1

Due to the issue, that `global_store_efficiency` could not be measured at all, we can only speculate about the suitability of the chosen memory layout of the results.[13] Measurements using the Visual Profiler showed, that the `global_load_efficiency` is about 2%.[14] Thus, we note that the `global_load_efficiency` and the high local resource requirements are the main performance limiters. To determine how improvements of these limitations would affect the performance, we inspect another common limiter. An issue reducing the performance for many algorithms when using CUDA is branch divergence, which is explained in section 3.1.2. Table 4.13 shows the ratio of conditions which lead to serialized execution of the branches.

We note, that divergent branches don't limit the performance of the implementation.

We summarize that this experiment shows, that the implementation is memory bound. However, we note, that the limitation may not be accesses to global but to local memory. Other metrics inspected show, that a good performance could be reached by improving the memory layout for the parameters and reduce the requirements of local resources. Improvements of the memory

[13] The switch from a memory layout in CPU-fashion to the memory layout presented in section 3.3, resulted in a boost of the performance of about factor 2.

[14] For the input data structure the boost resulting from changing a CPU-like memory layout to the used layout, described in section 3.3, was nearly negligible. Thus, improving this memory layout, may increase the performance, too. See sections 4.10.1 and 4.10.2 for recommendations.

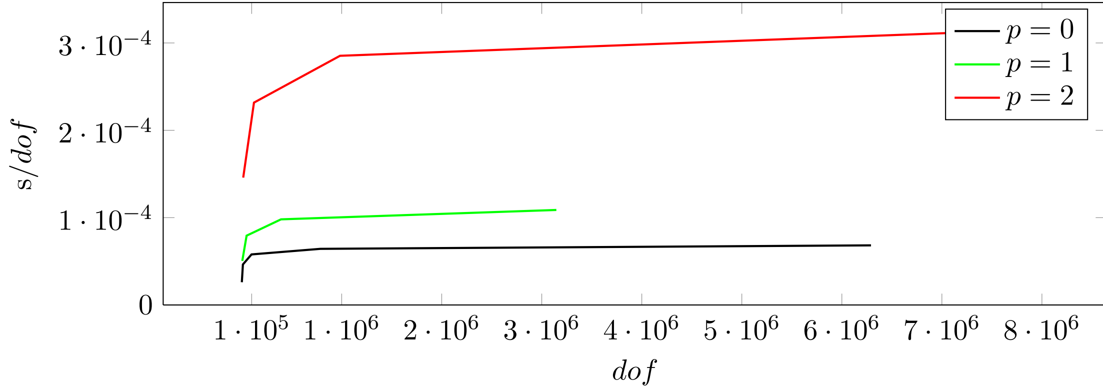

Figure 4.4: The performance of the integration on Ω_I of the CPU-implementation $\mathcal{P}_{CPU}$ in the 3D case. See table B.19 for the data on which the figure is based.

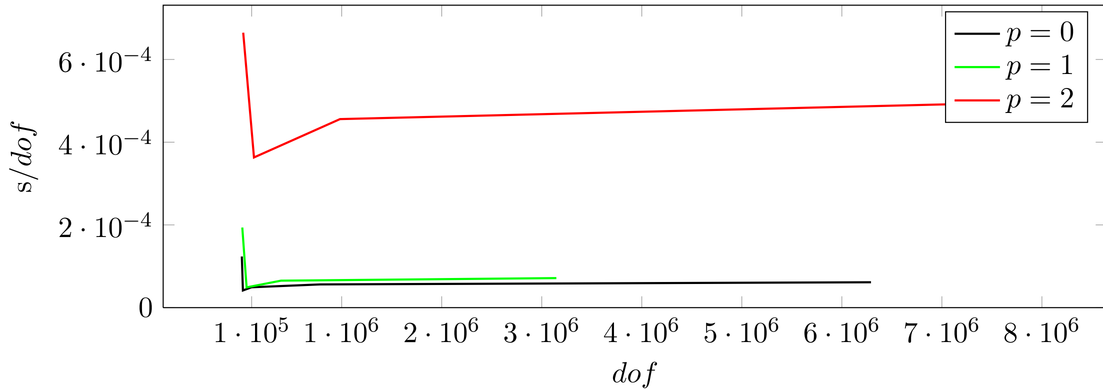

Figure 4.5: The performance of the integration on Ω_I of the GPU-implementation $\mathcal{P}_{GPU}$ in the 3D case. See table B.20 for the data on which the figure is based.

layout are discussed in section 4.10.1. In the sections 4.10.3 and 4.10.4 the reduction of the local resource requirements is discussed.

4.7 Experiment: 3D Performance

From the previous experiments we learned, that accesses to the local and global memory limit the performance. Since for 3D more local resources are required, we expect the performance to be worse. All other effects we observed previously, for example the increase of the cost depending on the level, are expected to be similar for the 3D case. For 3D the computational effort and the memory requirements for the results grow by factor 8 with an increase of the level by 1 and for the polynomial degree the formula given in in section 2.5 applies.

Figure 4.4 and figure 4.5 show the duration of the CPU-implementation and GPU-implementation respectively. The ratio of these durations is given in figure 4.6.

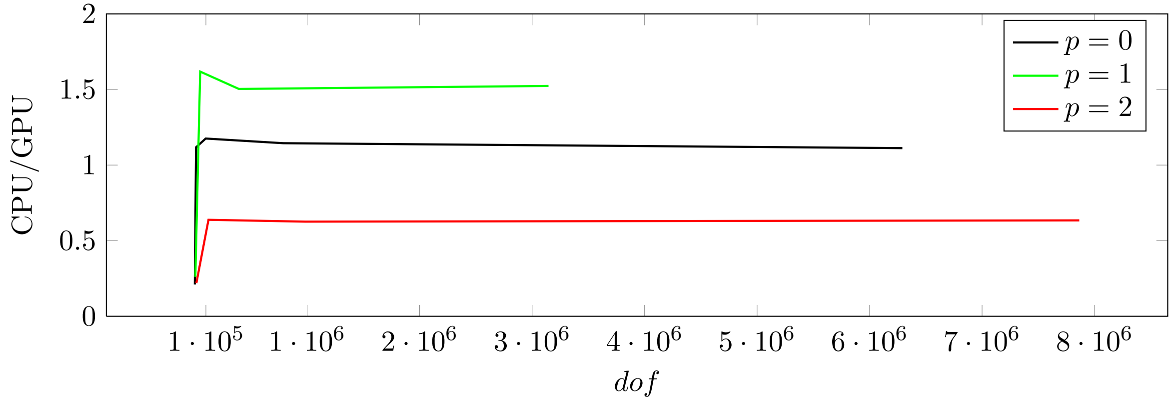

Figure 4.6: The relative performance of the integration on Ω_I of the CPU and GPU-implementation $\mathcal{P}_{CPU}/\mathcal{P}_{GPU}$ in the 3D case. See table B.21 for the data on which the figure is based.

We see, that the performance is worse in the 3D case compared to the 2D case. The other effects are similar to the 2D case. For details see tables B.19, B.20 and B.21. This experiment shows, that in 3D for polynomial degree 1 the GPU-implementation boosts the performance by a factor of about factor 1.5 compared to the CPU-implementation.[15] We conclude, that in 3D the implementation performs worse than in 2D due to higher local resource requirements.

4.8 Experiment: Different Hardware

As mentioned in section 4.6, the implementation is memory bound. Therefore, we expect the performance to be similar for GPUs with higher arithmetical performance, but similar memory configuration. The NVIDIA™ K20 has about 11 times the arithmetical performance of the Geforce GTX 560 Ti. However, the memory bandwidth is about 1.6 times higher. Therefore, we expect the performance on the NVIDIA™ K20 to be about 1.6 times higher than that on the Geforce GTX 560 Ti. The relative performance is given in table 4.14.

We can see from the table that the relative performance depends on the polynomial degree. Since for higher polynomial degree p more local memory is utilized, the performance benefits should enhance for increasing polynomial degree p. For polynomial degree $p = 4$ the performance advantage is unexpected low. For the other polynomial degrees the performance advantage seems to increase for increasing level l and polynomial degree p. The expected boost by a factor of about 1.6, is not reached.[16] One explanation for this behavior is, that the memory latency is the main limiter instead of the memory bandwidth. To examine this further, additional

[15] For other polynomial degrees we see worse performance. For polynomial degree ≥ 2, we note that we cannot launch enough thread blocks to utilize all SMs.

[16] The system, on which this was tested only has 4 GiB of main memory and thus the level l, which could be utilized, was limited.

Table 4.14: The relative performance of the implementation on the NVIDIA™ K20 and the Geforce GTX 560 Ti. See tables B.8 and B.24 for the data on which this table is based.

P/L	5	6	7	8
0	1	0.67	1.4	1.05
1	0.6	0.8	1.13	0.97
2	0.72	0.65	1.15	1.15
3	0.66	0.67	1.27	1.28
4	0.63	0.8	0.98	0.93
5	0.62	0.97	1.48	1.42

experiments would be required. As expected, for SP the performance was equal to that for DP on the NVIDIA™ K20.[17] Thus, this experiment confirms that the implementation is memory bound.

4.9 Summary of the Results

In this section we summarize the results from the experiments, presented in the previous sections.

1. For DP the performance of the integration on Ω_I could be increased by factors between 1.5 and 2.5, depending on the polynomial degree p using a Geforce GTX 560 Ti compared to one core of the Intel i7 2600k, see section 4.2. The highest benefit, of the about 70% of the performance benefit theoretically possible, was reached for polynomial degree 1 on a Geforce GTX 560 Ti.
2. The performance when using SP isn't satisfying on the Geforce GTX 560 Ti, see section 4.5. We concluded that the implementation is memory bound, see section 4.6. This was confirmed by the results from an experiment utilizing a NVIDIA™ K20, which exhibits low performance when using DP, see section 4.8.
3. The analysis of the profiler data in section 4.6 showed that other common issues in CUDA-implementations don't occur. The implementation is memory bound. Two possible reasons were identified:
   - The access patterns to the global data structures
   - The excessive usage of local memory

[17]See tables 4.6 and B.22 in the appendix.

Since the memory patterns for the results seem to be good[18], the main performance limiters are the excessive usage of local memory and the memory layout of the parameters. This permits the assumption, that if the local resource requirements are reduced and the memory layout of the parameters is revised,[19] a satisfying performance can be reached. In the sections 4.10.3 and 4.10.4 we present strategies, how to reduce the local resource requirements and in the sections 4.10.1 and 4.10.2 we give recommendations for the improvement of the memory layout of the parameters.

4. For smaller polynomial degree p the duration of the postprocessing dominates the runtime of the kernel, see section 4.4. Therefore, an improvement of this issue will significantly improve the performance for smaller polynomial degree. In section 4.10.1 we present strategies to avoid this.

4.10 Further Improvements

Based on the results summarized in the previous section, in this section we give some hints how to improve the performance of the implementation. Thus, detailed strategies to address all issues inspected in the experiments are given.

4.10.1 GPU-based Accumulation

As mentioned in section 4.4 for small polynomial degree p, the primary performance limiter of the implementation is the postprocessing. We recall, that the goal of the implementation was to avoid atomic operations, see section 3.3. The experiments showed, that this lead to the mentioned domination of the postprocessing. To solve this we need to accumulate the results on the GPU. We have two options:

1. using atomic operations for the adds to the resulting matrices and vectors
2. using an extra kernel to accumulate the results on the GPU

Both options imply we need for a data structure, that stores the neighbors of all patches, that are in the neighborhood of the patches $\mathcal{W}_i$ processed in one schedule.

We give an example to explain this. Figure 4.7 supposes identifier (numbers) which are used in the example. The patches with the identifiers 0 and 1 are calculated in one schedule. Therefore, all patches illustrated in the figure need to be stored. This is represented by the blocks on the left in figure 4.8. Of course, the neighborhoods of the patches 0 and 1 need to be stored. This information is marked in turquoise. In addition to that, the neighborhoods of all patches that are in the neighborhood of the patches 0 and 1 are required. Especially, only the positions of

[18]As mentioned in section 4.6, the switch from a memory layout in CPU-fashion to the memory layout presented in section 3.3, resulted in a boost of the performance of about factor 2.

[19]As mentioned in section 4.6, for the input data structure the boost resulting from changing a CPU-like memory layout to the used layout, described in section 3.3, was nearly negligible.

6	7	8	11
5	0	1	9
4	3	2	10

Figure 4.7: A possible indexing scheme for 2 adjacent neighborhoods

patch 0
patch 1
patch 2
patch 3
patch 4
patch 5
patch 6
patch 7
patch 8
patch 9
patch 10
patch 11

neighborhood 0	0	1	2	3	4	5	6	7	8
neighborhood 1	1	9	10	2	3	0	7	8	11
neighborhood 2	2	10	..	..	..	3	0	11	9
neighborhood 3	3	2	..	..	..	4	5	0	1
neighborhood 4	4	3	..	..	..	..	..	5	0
neighborhood 5	5	0	3	4	..	..	..	6	7
neighborhood 6	6	7	0	5	..	..	..	..	..
neighborhood 7	7	8	1	0	5	6	..	..	..
neighborhood 8	8	11	9	1	0	7	..	..	..
neighborhood 9	9	..	..	10	2	1	8	11	..
neighborhood 10	10	..	..	..	..	2	1	9	..
neighborhood 11	11	..	..	9	1	8	..	..	..

Figure 4.8: The information required for GPU-based accumulation. The indices used are given in figure 4.7. On the left the list of patches is given and on the right the neighborhood lists are shown. When using GPU based accumulation the yellow marked information is required in addition to the information marked in turquoise. Dots in fields of neighborhood lists represent patches, that are not part of the schedule $\mathcal{W}_i$ of the patches 0 and 1

all patches in one of the neighborhoods of 1 or 2 need to be known in the neighborhood list of all patches. This additional information is marked yellow. At first glance the additional neighborhoods seem to be many. But since the patches in one schedule are adjacent, the number of neighborhoods which need to be stored and are not affiliated to a patch, is small compared to the absolute number of patches. We note that the storage of the patches as well as the neighborhoods should be aligned, as discussed in section 3.3.

With this information at hand, we can determine the position of ω_j in the list of neighbors of every patch ω_i. This information is required, to know into which matrix the results of the integration need to be written.

Recalling the two options, we can use this information to allocate the results per patch and write to the appropriate matrix entry in atomic fashion. Since many atomic writes are required in this case the performance may also be limited but since the host based accumulation is very time consuming the implementation may benefit from that change. Another positive effect would be, that it requires only $1/9$ of the memory compared to the implementation presented. Therefore, more patches can be computed in one schedule which might also lead to a performance improvement.

An other option is, that each patch in $\mathcal{W}_i$ writes into its own result data structure and then an additional kernel is launched to accumulate the results. The task of this second kernel is known as parallel reduction. See [4] for strategies to optimize this reduction. This would even require more memory on the GPU but would avoid atomic additions.

4.10.2 Improved Memory Access

The memory accesses have been identified to be main performance limiter in section 4.6. In this section, therefore, we discuss how the memory accesses can be improved. Two factors should be considered when optimizing memory accesses:

- the number of accesses should be minimized
- the access patterns should lead to coalesced memory accesses

When applying the strategy, given in section 4.10.1, the number of memory requests should also be reduced. The load instructions from different threads to the same memory location can be handled in a single memory transaction. Therefore, the more orderless access to the patches should not compromise the performance. When applying the same memory layout to the neighborhood information it should lead to coalesced accesses since all threads access the same index at a time. Since the metric `gst_efficiency` could not be computed by the profiler, see section 4.6, it's hard to tell if the store instructions are an unexpected performance limiter.[20] Therefore, the attention should be focused on acquiring the missing metric from the profiler.

4.10.3 Multi Kernel Version

The compiler optimizes the usage of local resources. However, this optimization performs better for a kernel with less instructions and less local resources. To support compiler optimization, especially register usage, a multi kernel should be considered. It's possible to split the decomposition from the integration. This can be achieved by storing the result of the decomposer in global memory. A second kernel uses this decomposition to retrieve the cells for integration. This would support the compiler, to free the resources that are temporarily used by the decomposer. Since the implementation uses a single monolithic kernel, a split of the functionality into two kernels may improve the performance.

[20] As before, the switch from a memory layout in CPU-fashion to the memory layout presented in section 3.3, resulted in a boost of the performance of about factor 2.

4.10.4 Level of Parallelization

Another approach that should be considered, is changing the level of parallelization. The level of parallelization defines the granularity of splitting the task to a number of threads. Changing the level of parallelization can have massive impact on the local resource requirements of one thread.

We recall that the requirements for local resources impact the performance in two ways:

- it reduces the occupancy
- it introduces overhead for local memory accesses

When applying the parallelization on a different level this might lead to decreased resource requirements and therefore, to an improved performance. Instead of assigning a patch to each thread for decomposition and integration it should be considered to use a different number of threads for the decomposition and the integration. One thread in a first phase would decompose the domain of a patch ω_i and in a second phase each thread would integrate one integration cell. This could be achieved by three ways:

- A first kernel decomposes the patch and a second operates on the decomposed domain. This is similar to the approach, given in the previous section, but would utilize a thread for the integration of each resulting integration cell.
- A patch is decomposed by one thread of a thread block and each resulting integration cell is integrated by one thread of the thread block afterwards. This could utilize the shared memory in a more efficient way since information on the decomposition and the neighbor patches of the patch, on which the thread block operates, can be shared.
- A patch is decomposed by one thread which uses dynamic parallelism, see [14] for details, to start a second kernel. Each thread of the second kernel integrates one integration cell generated by the calling thread of the first kernel.

All these solutions however involve the problem of writing the results into a global data structure. As discussed in section 4.10.1, the accumulation of the results requires some effort. When using a thread per integration cell instead of a thread per patch it multiplies the expected effect. Therefore, changing the level of parallelization should be reconsidered before applying, but could improve the performance remarkable.

5 Conclusion

In this thesis we presented a GPU-implementation of the decomposition of the domain and the numerical integration which are required in the PMPUM for the discretization of a given PDE. These steps make up a large share of the computational effort in the method. Therefore, first the theoretical background was given and the suitability of different algorithms for the decomposition using CUDA was discussed. An introduction to the major concepts of CUDA laid the foundation for the outline of the implementation. Finally, we analyzed the implementation using different experiments and derived recommendations for further improvements.

The experiments conducted to analyze the implementation showed that the performance of the decomposition and the numerical integration can be increased using CUDA. This increase of about factor 2.5, was achieved for the Geforce GTX 560 Ti GPU compared to the Intel i7 2600k CPU using double precision floating point numbers. The Geforce GTX 560 Ti has a peak performance that is 3.4 times higher than of one core of the Intel i7 2600k CPU. Assuming, that the CPU is maxed out and utilizes only one core, this is about 70% of the performance that can be achieved. For single precision floating point numbers, however, a massive increase was expected, since the processing power for single precision floating point numbers of Geforce GTX 560 Ti is about 41 times higher than that of the Intel i7 2600k. However, in the experiments that could not be verified. We analyzed that the implementation is memory bound, due to an extensive use of local resources, which causes register spilling and a suboptimal memory layout of the parameters. An accumulation step on the CPU accounts for a serious amount of the computations, which constraints the performance when using functions with lower polynomial degree p. Other common performance limiters were eliminated in the implementation, which was verified in the experiments. Therefore, decreasing the usage of local resources and revising the memory layout for the parameters is expected to increase the performance, especially when using single precision floating point numbers. Particularly, when using smaller polynomial degree a GPU-based accumulation will boost the performance. A more technical summary of the results can be found in section 4.9.

Strategies how to tackle these limitations where presented in detail in section 4.10. Which of the suggested strategies will improve the performance most, can only be found out by experiment. However, with these strategies in mind, we expect that a satisfying single precision performance can be reached for all polynomial degrees at least for the 2D case. For devices with a much higher relative double precision floating point number processing performance (compared to single precision floating point), like the NVIDIA™ K20, the double precision floating point performance was low. However, with the presented improvement strategies applied, we expect a good performance for this case, too. In 3D, the task of reducing the local resource requirements is more challenging.

Finally, we put the performance reached into the context of a use case. We recall from the Introduction that the PMPUM can be used to solve PDEs arising from real world problems. In realistic cases, the required detail of the solution is very high and therefore, requires a huge amount of computations. As mentioned in section 3.2 the CPU-implementation can utilize a computer cluster to distribute computations. The GPU-implementation can utilize any CUDA-capable GPUs on a cluster node. Our findings show that utilizing any available GPUs for the computation will boost the performance of the PMPUM-implementation. In the case, however, that the number of CPUs and GPUs can be chosen freely, our findings would propose a computer cluster with more CPUs instead of additional GPUs. With the recommended improvement applied, however, we expect that selecting a computer cluster with one GPU on each node would improve the performance.

A Memory Layout of the Result Data

In section 3.3 we present the memory layout used for the parameters and the results. For the results, however, some further explanations might be interesting, especially for further development.

We recall from section 2.3 that A contains a block row for each patch ω_i. In this row there exists a nonzero block for each neighbor $\omega_j \in C_i$. The size of these nonzero blocks if given by the size b_i of the function space $V_i^{p_i}$ and the size b_j of the function space $V_j^{p_j}$. The situation for the right-hand side is analog. The right-hand side $\hat{f}$ consists of a vector of size b_i per patch ω_i. Since, as discussed in 3.3, the results data is stored per patch ω_i, all rows of the patches ω_j in the neighborhood C_i are store per patch. For uniform h-refinement this means that the result data requires 9 times more memory than would be needed in the case of atomic additions on the GPU, presented in 4.10.1.

Therefore we need to consider the following indices:

- one index for the neighborhoods, neighborhoodIndex in the following
- two indices for the patches of a neighborhood to calculate the scalar product, patchIndexI and patchIndexJ
- two indices for the base functions of the function space on a patch, functionSpaceIndexI and functionSpaceIndexJ

If multiple linear forms and bilinear forms are used for each an additional index is required. The hierarchy of these indices is the following:

- patchIndexI
  - linearIndex
    - functionSpaceIndexI
      - neighborhoodIndex
  - patchIndexJ
    - bilinearIndex
      - functionSpaceIndexI
        - functionSpaceIndexJ
          - neighborhoodIndex

For a illustration of this indexing scheme see figure A.1.

patchIndexI	0..(M-1)																														
patchIndexJ	0	1..(M-1)
linearIndex	0..(L-1)	
biLinearIndex		0..(B-1)	...	0..(B-1)
functionspaceIndexI	0..(F-1)	...	0..(F-1)	0..(F-1)	...	0..(F-1)	...	0..(F-1)	...
functionspaceIndexJ	0	0..(F-1)	...	0..(F-1)	...	0..(F-1)	...	0..(F-1)	...	0..(F-1)	...	0..(F-1)	...
neighborhoodIndex	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...

patchIndexI	0..(M-1)																															
patchIndexJ		...	0	1..(M-1)
linearIndex		...	0..(L-1)	
biLinearIndex	0..(B-1)	...		0..(B-1)	...
functionspaceIndexI	...	0..(F-1)	...	0..(F-1)	...	0..(F-1)	0..(F-1)	...	0..(F-1)	...
functionspaceIndexJ	...	0..(F-1)	...	0..(F-1)	...	0	0..(F-1)	...	0..(F-1)	...	0..(F-1)	...	0..(F-1)	...
neighborhoodIndex	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...

patchIndexI	0..(M-1)															
patchIndexJ	1..(M-1)
linearIndex	
biLinearIndex	...	0..(B-1)
functionspaceIndexI	...	0..(F-1)	...	0..(F-1)
functionspaceIndexJ	...	0..(F-1)	...	0..(F-1)	...	0..(F-1)	...	0..(F-1)
neighborhoodIndex	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)	...	0..(N-1)

Figure A.1: The indexing scheme used for the results

B Raw Result Data

In this chapter we provide the raw data from our experiments, that we used to analyze our implementation in chapter 4. If not noted otherwise the reference system (see table 4.1) is used to solve the model problem given in equation 2.1 using DP. We always reference the experiment in which the data was used.

Durations for the Integration of the complete Domain Ω

Table B.1: The duration required by the CPU-implementation for the integration on Ω. Used in section 4.2.

P/L	3	4	5	6	7	8	9
0	$1 \cdot 10^{-2}$	$2 \cdot 10^{-2}$	$5 \cdot 10^{-2}$	0.15	0.58	2.36	9.55
1	$1 \cdot 10^{-2}$	$3 \cdot 10^{-2}$	$9 \cdot 10^{-2}$	0.38	1.55	6.28	25.22
2	$1 \cdot 10^{-2}$	$6 \cdot 10^{-2}$	0.26	1.08	4.39	17.66	71.02
3	$3 \cdot 10^{-2}$	0.17	0.71	2.91	11.75	47.3	190.3
4	$9 \cdot 10^{-2}$	0.44	1.81	7.36	29.82	120.18	482.43
5	0.23	1.02	4.26	17.46	70.68	284.5	1,142.05

Table B.2: The duration required by the CPU-implementation for the integration on Ω. Used in section 4.2.

P/L	3	4	5	6	7	8	9
0	$7 \cdot 10^{-2}$	$8 \cdot 10^{-2}$	$7 \cdot 10^{-2}$	0.12	0.39	1.3	5.3
1	$9 \cdot 10^{-2}$	$8 \cdot 10^{-2}$	0.1	0.21	0.71	2.64	10.4
2	0.14	0.18	0.22	0.51	2.12	8.51	33.84
3	0.39	0.46	0.57	1.46	6.53	26.39	103.99
4	0.96	1.2	1.47	4.15	14.28	56.28	224.77
5	2.33	2.76	3.44	11.56	46.85	179.47	715.63

Table B.3: The relative duration required by the CPU-implementation and the CPU-implementation for the integration on Ω. Used in section 4.2.

P/L	3	4	5	6	7	8	9
0	7	4	1.4	0.8	0.67	0.55	0.55
1	9	2.67	1.11	0.55	0.46	0.42	0.41
2	14	3	0.85	0.47	0.48	0.48	0.48
3	13	2.71	0.8	0.5	0.56	0.56	0.55
4	10.67	2.73	0.81	0.56	0.48	0.47	0.47
5	10.13	2.71	0.81	0.66	0.66	0.63	0.63

Durations for the Integration of inner Domain Ω_I

Table B.4: The duration required by the CPU implementation for decomposition and integration on Ω_I. Used in section 4.2

P/L	3	4	5	6	7	8	9
0	0	0	$4 \cdot 10^{-2}$	0.12	0.55	2.26	9.14
1	0	$2 \cdot 10^{-2}$	$7 \cdot 10^{-2}$	0.36	1.5	6.15	24.73
2	0	$5 \cdot 10^{-2}$	0.24	1.03	4.28	17.42	70.29
3	$2 \cdot 10^{-2}$	0.15	0.64	2.78	11.49	46.76	188.96
4	$6 \cdot 10^{-2}$	0.37	1.66	7.06	29.23	118.88	479.76
5	0.16	0.86	3.92	16.78	69.29	281.65	1,136.12

Table B.5: The duration required by the GPU implementation for decomposition and integration on Ω_I. Used in sections 4.2, 4.3

P/L	3	4	5	6	7	8	9
0	$6 \cdot 10^{-2}$	$7 \cdot 10^{-2}$	$7 \cdot 10^{-2}$	0.12	0.36	1.2	4.99
1	$8 \cdot 10^{-2}$	$8 \cdot 10^{-2}$	$9 \cdot 10^{-2}$	0.18	0.65	2.5	10
2	0.13	0.15	0.2	0.46	2.02	8.24	33.11
3	0.37	0.42	0.51	1.34	6.28	25.83	102.66
4	0.93	1.13	1.32	3.85	13.66	55.01	222.04
5	2.25	2.6	3.11	10.86	45.42	176.6	709.72

Table B.6: The relative duration required by the GPU-implementation and the CPU-implementation for decomposition and integration on Ω_I. Used in sections 4.2

P/L	3	4	5	6	7	8	9
0	7	4	1.4	0.8	0.67	0.55	0.55
1	9	2.67	1.11	0.55	0.46	0.42	0.41
2	14	3	0.85	0.47	0.48	0.48	0.48
3	13	2.71	0.8	0.5	0.56	0.56	0.55
4	10.67	2.73	0.81	0.56	0.48	0.47	0.47
5	10.13	2.71	0.81	0.66	0.66	0.63	0.63

Durations for the different Steps of the GPU-Implementation

Table B.7: The duration required by the preprocessing of the GPU-implementation for the decomposition and integration on Ω_I. Used in section 4.4.

P/L	5	6	7	8	9
0	$1 \cdot 10^{-2}$	$1 \cdot 10^{-2}$	$1 \cdot 10^{-2}$	$8 \cdot 10^{-2}$	0.29
1	0	0	$2 \cdot 10^{-2}$	$4 \cdot 10^{-2}$	0.33
2	0	$1 \cdot 10^{-2}$	0	$6 \cdot 10^{-2}$	0.33
3	0	$1 \cdot 10^{-2}$	$2 \cdot 10^{-2}$	$8 \cdot 10^{-2}$	0.31
4	0	0	$3 \cdot 10^{-2}$	0.1	0.45
5	0	0	$2 \cdot 10^{-2}$	0.15	0.45

Table B.8: The duration required by the kernel of the GPU-implementation for the decomposition and integration on Ω_I. Used in section 4.4.

P/L	5	6	7	8	9
0	$1 \cdot 10^{-2}$	$2 \cdot 10^{-2}$	0.14	0.42	1.71
1	$3 \cdot 10^{-2}$	$8 \cdot 10^{-2}$	0.34	1.31	5.23
2	0.13	0.26	1.2	5.04	19.85
3	0.41	0.91	4.51	18.6	73.26
4	1.12	2.97	10.06	40.28	162.62
5	2.78	9.26	38.84	149.87	601.49

Table B.9: The duration required by the postprocessing of the GPU-implementation for the decomposition and integration on Ω_I. Used in section 4.4.

P/L	5	6	7	8	9
0	$5 \cdot 10^{-2}$	$9 \cdot 10^{-2}$	0.2	0.68	2.84
1	$6 \cdot 10^{-2}$	0.1	0.29	1.09	4.33
2	$7 \cdot 10^{-2}$	0.19	0.8	3.12	12.81
3	0.1	0.41	1.74	7.13	28.95
4	0.19	0.88	3.56	14.59	58.77
5	0.33	1.6	6.56	26.54	107.55

Durations for the Integration of inner Domain Ω_I using Single Precision

Table B.10: The duration required by the GPU-implementation when using SP for the integration on Ω_I. Used in section 4.5.

P/L	5	6	7	8	9
0	$7 \cdot 10^{-2}$	0.12	0.34	1.23	4.92
1	$9 \cdot 10^{-2}$	0.18	0.64	2.44	10.04
2	0.2	0.46	2	8.24	33.29
3	0.51	1.34	6.44	25.05	103.3
4	1.33	3.81	13.69	55.2	222.13
5	3.29	10.79	44.99	176.97	708.8

Table B.11: The duration required by the GPU-implementation for the preprocessing when using SP for the integration on Ω_I. Used in section 4.5.

P/L	5	6	7	8	9
0	$1 \cdot 10^{-2}$	0	$2 \cdot 10^{-2}$	0.1	0.27
1	0	0	$2 \cdot 10^{-2}$	$9 \cdot 10^{-2}$	0.3
2	0	$1 \cdot 10^{-2}$	$2 \cdot 10^{-2}$	$6 \cdot 10^{-2}$	0.34
3	0	0	$1 \cdot 10^{-2}$	0.14	0.4
4	0	0	$3 \cdot 10^{-2}$	0.1	0.51
5	0	$1 \cdot 10^{-2}$	$2 \cdot 10^{-2}$	0.17	0.53

Table B.12: The duration required by the GPU-implementation for the kernel when using SP for the integration on Ω_I. Used in section 4.5.

P/L	5	6	7	8	9
0	0	$3 \cdot 10^{-2}$	0.14	0.42	1.75
1	$3 \cdot 10^{-2}$	$7 \cdot 10^{-2}$	0.34	1.29	5.22
2	0.13	0.26	1.21	4.99	20.08
3	0.4	0.93	4.68	17.71	74.01
4	1.14	2.91	10.1	40.37	162.68
5	2.95	9.22	38.46	150.26	601.01

Table B.13: The duration required by the GPU-implementation for the postprocessing when using SP for the integration on Ω_I. Used in section 4.5.

P/L	5	6	7	8	9
0	$6 \cdot 10^{-2}$	$9 \cdot 10^{-2}$	0.18	0.69	2.79
1	$6 \cdot 10^{-2}$	0.1	0.28	1.05	4.37
2	$7 \cdot 10^{-2}$	0.19	0.77	3.14	12.78
3	0.11	0.41	1.75	7.18	28.76
4	0.19	0.89	3.56	14.66	58.78
5	0.34	1.56	6.48	26.47	107.05

Profiler data

Table B.14: The number of instructions issued during the kernel execution for the integration on Ω_I. Used in section 4.6.

P/L	5	6	7	8	9
0	$1.42 \cdot 10^{7}$	$7.46 \cdot 10^{7}$	$4.84 \cdot 10^{8}$	$1.68 \cdot 10^{9}$	$6.83 \cdot 10^{9}$
1	$4.96 \cdot 10^{7}$	$2.29 \cdot 10^{8}$	$1.07 \cdot 10^{9}$	$4.32 \cdot 10^{9}$	$1.75 \cdot 10^{10}$
2	$1.76 \cdot 10^{8}$	$7.81 \cdot 10^{8}$	$3.27 \cdot 10^{9}$	$1.35 \cdot 10^{10}$	$5.46 \cdot 10^{10}$
3	$5.58 \cdot 10^{8}$	$2.47 \cdot 10^{9}$	$1.02 \cdot 10^{10}$	$4.18 \cdot 10^{10}$	$1.69 \cdot 10^{11}$
4	$1.54 \cdot 10^{9}$	$6.44 \cdot 10^{9}$	$2.69 \cdot 10^{10}$	$1.09 \cdot 10^{11}$	$4.41 \cdot 10^{11}$
5	$3.74 \cdot 10^{9}$	$1.58 \cdot 10^{10}$	$6.55 \cdot 10^{10}$	$2.66 \cdot 10^{11}$	$1.07 \cdot 10^{12}$

Table B.15: The number of L1 load misses for local memory requests in the kernel for the integration on Ω_I. Used in section 4.6.

P/L	5	6	7	8	9
0	$5.58 \cdot 10^5$	$5.94 \cdot 10^6$	$2.09 \cdot 10^7$	$8.56 \cdot 10^7$	$3.46 \cdot 10^8$
1	$1.6 \cdot 10^6$	$2.04 \cdot 10^7$	$8.06 \cdot 10^7$	$3.1 \cdot 10^8$	$1.27 \cdot 10^9$
2	$4.25 \cdot 10^6$	$7.88 \cdot 10^7$	$3.07 \cdot 10^8$	$1.27 \cdot 10^9$	$5.06 \cdot 10^9$
3	$1.13 \cdot 10^7$	$2.76 \cdot 10^8$	$1.02 \cdot 10^9$	$4.28 \cdot 10^9$	$1.73 \cdot 10^{10}$
4	$3.9 \cdot 10^7$	$6.34 \cdot 10^8$	$3.07 \cdot 10^9$	$1.24 \cdot 10^{10}$	$4.92 \cdot 10^{10}$
5	$1.25 \cdot 10^8$	$1.32 \cdot 10^9$	$6.77 \cdot 10^9$	$2.74 \cdot 10^{10}$	$1.11 \cdot 10^{11}$

Table B.16: The number of L1 store misses for local memory requests in the kernel for the integration on Ω_I. Used in section 4.6.

P/L	5	6	7	8	9
0	$1.03 \cdot 10^6$	$5.33 \cdot 10^6$	$2.07 \cdot 10^7$	$8.48 \cdot 10^7$	$3.39 \cdot 10^8$
1	$2.12 \cdot 10^6$	$1.11 \cdot 10^7$	$4.31 \cdot 10^7$	$1.78 \cdot 10^8$	$7.22 \cdot 10^8$
2	$3.96 \cdot 10^6$	$2.13 \cdot 10^7$	$8.28 \cdot 10^7$	$3.34 \cdot 10^8$	$1.39 \cdot 10^9$
3	$7.02 \cdot 10^6$	$3.86 \cdot 10^7$	$1.5 \cdot 10^8$	$6.79 \cdot 10^8$	$2.59 \cdot 10^9$
4	$1.15 \cdot 10^7$	$6.57 \cdot 10^7$	$2.79 \cdot 10^8$	$1.12 \cdot 10^9$	$4.48 \cdot 10^9$
5	$1.77 \cdot 10^7$	$1.02 \cdot 10^8$	$4.83 \cdot 10^8$	$1.92 \cdot 10^9$	$7.78 \cdot 10^9$

Table B.17: The number of branches in the kernel for the integration on Ω_I. Used in section 4.6.

P/L	5	6	7	8	9
0	$1.11 \cdot 10^6$	$5.49 \cdot 10^6$	$3.62 \cdot 10^7$	$1.4 \cdot 10^8$	$5.78 \cdot 10^8$
1	$2.51 \cdot 10^6$	$1.15 \cdot 10^7$	$5.79 \cdot 10^7$	$2.59 \cdot 10^8$	$1.07 \cdot 10^9$
2	$6.58 \cdot 10^6$	$2.86 \cdot 10^7$	$1.22 \cdot 10^8$	$4.99 \cdot 10^8$	$2.01 \cdot 10^9$
3	$1.75 \cdot 10^7$	$7.47 \cdot 10^7$	$3.08 \cdot 10^8$	$1.25 \cdot 10^9$	$5.05 \cdot 10^9$
4	$4.3 \cdot 10^7$	$1.81 \cdot 10^8$	$7.46 \cdot 10^8$	$3.04 \cdot 10^9$	$1.22 \cdot 10^{10}$
5	$9.77 \cdot 10^7$	$4.11 \cdot 10^8$	$1.71 \cdot 10^9$	$6.77 \cdot 10^9$	$2.79 \cdot 10^{10}$

Table B.18: The number of divergent branches in the kernel for the integration on Ω_I. Used in section 4.6.

P/L	5	6	7	8	9
0	4,178	17,937	81,506	$3.36 \cdot 10^5$	$1.36 \cdot 10^6$
1	4,263	17,902	79,695	$3.21 \cdot 10^5$	$1.3 \cdot 10^6$
2	4,280	17,746	76,881	$3.11 \cdot 10^5$	$1.25 \cdot 10^6$
3	4,293	18,096	74,537	$3.04 \cdot 10^5$	$1.23 \cdot 10^6$
4	4,396	18,496	63,069	$2.82 \cdot 10^5$	$1.24 \cdot 10^6$
5	4,414	18,891	77,694	$3.16 \cdot 10^5$	$1.27 \cdot 10^6$

Durations for the Integration of the inner Domain Ω_I in 3D

Table B.19: The duration required by the CPU-implementation for the integration on Ω_I in 3D. Used in section 4.7.

P/L	2	3	4	5	6
0	0	$5 \cdot 10^{-2}$	0.57	5.69	50.75
1	$1 \cdot 10^{-2}$	0.3	3.93	38.75	344.15
2	$8 \cdot 10^{-2}$	2.23	28.34	278.93	

Table B.20: The duration required by the GPU-implementation for the integration on Ω_I in 3D. Used in section 4.7.

P/L	2	3	4	5	6
0	0.13	0.17	0.52	4.93	45.06
1	0.62	1.16	2.41	25.46	224.08
2	5.52	10.56	45.86	462.29	

Table B.21: The ratio of the duration required by the GPU-implementation and the CPU-implementation for decomposition and integration on Ω_I in 3D. Used in section 4.7.

P/L	2	3	4	5	6
0	0	3.4	0.91	0.87	0.89
1	62	3.87	0.61	0.66	0.65
2	69	4.74	1.62	1.66	

Data for the Integration of the inner Domain Ω_I on different Hardware

Table B.22: The relative performance of the implementation on the NVIDIA™ K20 and the Geforce GTX 560 Ti using SP. See tables B.12 and B.25 for the data, on which this table is based. Referred to in section 4.8.

P/L	5	6	7	8
0	∞	1	0.71	0.95
1	1.33	1.43	0.88	1.06
2	1.38	1.5	0.87	0.88
3	1.55	1.43	0.76	0.82
4	1.56	1.35	1.02	1.07
5	1.52	1.03	0.68	0.7

Table B.23: The relative performance of the implementation on the NVIDIA™ K20 using SP and DP. See tables B.24 and B.25 for the data, on which this table is based. Referred to in section 4.8

P/L	5	6	7	8
0	1	1	1	1
1	1.25	1	1	0.99
2	1	1.03	0.99	1
3	1	1.02	1	1
4	1	0.94	1	1
5	1	1.01	1	1

Table B.24: The duration required by the kernel for the integration on Ω_I on the NVIDIA™ K20. Used in section 4.8.

P/L	5	6	7	8
0	$1 \cdot 10^{-2}$	$3 \cdot 10^{-2}$	0.1	0.4
1	$5 \cdot 10^{-2}$	0.1	0.3	1.35
2	0.18	0.4	1.04	4.39
3	0.62	1.36	3.55	14.57
4	1.78	3.7	10.27	43.24
5	4.49	9.57	26.28	105.76

Table B.25: The duration required by the kernel for the integration on Ω_I on the NVIDIA™ K20 using SP. Used in tables B.22 and B.23.

P/L	5	6	7	8
0	$1 \cdot 10^{-2}$	$3 \cdot 10^{-2}$	0.1	0.4
1	$4 \cdot 10^{-2}$	0.1	0.3	1.37
2	0.18	0.39	1.05	4.37
3	0.62	1.33	3.54	14.57
4	1.78	3.94	10.29	43.16
5	4.49	9.51	26.18	105.74

Glossary

cluster node A cluster node is a computer which is part of a computer cluster. 41, 68, 81

Compute Capability The Compute Capability defines the hardware features available on different GPUs using defined Levels. For details see [13, p.12f.]. 7, 35–39, 45–48, 56

computer cluster A computer cluster is a set of computers (called cluster nodes) connected with an interconnecting network, that is used to distribute computations among all computers. 41, 68, 81

h-refinement H-refinement is a refinement strategy in multi level methods. It increases the number of points or elements with increasing level. This results in a decrease of h_i for each patch ω_i. 21, 30, 32, 34, 45, 69

kernel A CUDA kernel is special C function which is executed in parallel by a specified number of CUDA threads [13, p.7]. 8, 9, 36, 37, 39–41, 44, 47, 52–58, 62, 64, 65, 76, 77

memory bound function A function whose computational effort is dominated by the memory accesses instead of the necessary arithmetical operations. 55, 61

occupancy occupancy is the ratio of the active warps on a SM of the maximum number of warps supported by the SM. It, therefore, limits the utilization of the SM's ALU. [13, p.69] This doesn't apply in all cases. [23, p. 25ff.]. 8, 55, 56, 65

p-refinement P-refinement is a refinement strategy in multi level methods. It increases the polynomial degree used in $V_i^{p_i}$ with increasing level. 21, 34

spline A spline is a continuous, piecewise defined polynomial function. 22–24, 32

warp A warp is a partition of a thread block that is scheduled by the warp scheduler of a Streaming Multiprocessors for parallel execution on it. The threads of a warp are executing all instructions in parallel if a divergent branch occurs, all branches are executed sequentially. This technique is referred to as Single Instruction Multiple Threads (SIMT) by NVIDIA [13, p.63ff.]. 35, 37–39, 55, 57, 81

Bibliography

[1] D. Braess. *Finite Elements: Theory, Fast Solvers, and Applications in Solid Mechanics*. Cambridge University Press, 2007. ISBN: 9780521705189.

[2] Jansen Felix. *From Isolated Numerical Approaches to an Integrative Systems Science*. 2012. URL: http://www.simtech.uni-stuttgart.de/index.en.html.

[3] W. Hackbusch. *Theorie und Numerik elliptischer Differentialgleichungen*. Teubner-Studienbücher. Teubner B.G. GmbH, 1986. ISBN: 3-519-02074-2.

[4] Mark Harris. *Optimizing Parallel Reduction in CUDA*. Sept. 2012. URL: http://vuduc.org/teaching/cse6230-hpcta-fa12/slides/cse6230-fa12--05b-reduction-notes.pdf.

[5] Intel. *Intel® Core i7-2600 Desktop Processor Series*. Feb. 2012. URL: http://download.intel.com/support/processors/corei7/sb/core_i7-2600_d.pdf.

[6] Intel. *Intel® Core™ i7-2600K Processor (8M Cache, up to 3.80 GHz)*. URL: http://ark.intel.com/products/52214.

[7] Sebastian Kanis. "GPU-based Assembly of Stiffness Matrices in the Parallel Multilevel." Studythesis. University of Stuttgart, May 2012.

[8] M. A. Schweitzer. *A Parallel Multilevel Partition of Unity Method for Elliptic Partial Differential Equations*. Vol. 29. Lecture Notes in Computational Science and Engineering. Springer, 2003. ISBN: 3-540-00351-7.

[9] M. A. Schweitzer. "Efficient Implementation and Parallelization of Meshfree and Particle Methods—The Parallel Multilevel Partition of Unity Method." In: *Frontiers of Numerical Analysis*. Ed. by Alan W. Craig James F. Blowey. Springer Berlin Heidelberg, 2005, pp. 195–262. DOI: 10.1007/3-540-28884-8_4.

[10] Paulius Micikevicius. *GPU Performance Analysis and Optimization*. NVIDIA. 2012. URL: http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0514-GTC2012-GPU-Performance-Analysis.pdf.

[11] NVIDIA. *CUDA API REFERENCE MANUAL*. V 5.0. Oct. 2012.

[12] NVIDIA. *CUDA C BEST PRACTICES GUIDE*. Oct. 2012.

[13] NVIDIA. *CUDA C PROGRAMMING GUIDE*. 5.0. NVIDIA, Nov. 2012.

[14] NVIDIA. *CUDA DYNAMIC PARALLELISM PROGRAMMING GUIDE*. Aug. 2012.

[15] NVIDIA. *CUDA GPUs*. 2013. URL: https://developer.nvidia.com/cuda-gpus.

[16] NVIDIA. *Datasheet: NVIDIA® TESLA® KEPLER GPU COMPUTING ACCELERATORS*. May 2012. URL: http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf.

[17] NVIDIA. *GeForce GTX 560 Ti Specifications*. 2013. URL: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-560ti/specifications.

[18] NVIDIA. *PROFILER User's Guide*. Oct. 2012.

[19] NVIDIA. *What Is CUDA*. 2012. URL: http://developer.nvidia.com/cuda/what-cuda.

[20] Klaudius Scheufele. "Robuste Multilevel-Lösung elliptischer partieller Differentialgleichungen mit springenden Koeffizienten." Bachelorthesis. University of Stuttgart, Jan. 2013.

[21] Ryan Smith. *NVIDIA's GeForce GTX 560 Ti w/448 Cores: GTX 570 On A Budget*. Nov. 2011. URL: http://www.anandtech.com/show/5153/nvidias-geforce-gtx-560-ti-w448-cores-gtx570-on-a-budget.html.

[22] Angelika Steger. *Diskrete Strukturen 1.: Kombinatorik, Graphentheorie, Algebra*. 2nd ed. Springer Verlag, 2007. ISBN: 978-3-540-46660-4.

[23] Vasily Volkov. *Better Performance at Lower Occupancy*. Sept. 2010. URL: http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf.

[24] Albert Ziegenhagel. "Parallelisierung des Partition of Unity Codes Crass." Bachelorthesis. University of Stuttgart, Dec. 2012.

All links were last followed on April 16, 2013.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Sebastian Kanis)