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

Studienarbeit Nr. 2358

GPU-based Assembly of Stiffness Matrices in the Parallel Multilevel Partition of Unity Method

Sebastian Kanis

Course of Study:	Computer Science
Examiner:	Prof. Dr. rer. nat. Marc Alexander Schweitzer
Supervisor:	Dr. rer. nat. Stefan Zimmer
Commenced:	November 15, 2011
Completed:	May 15, 2012
CR-Classification:	G.4, J.2

Contents

List of Figures

List of Tables

List of Abbreviations

CPU Central Processing Unit. 13, 23, 25, 28, 29, 31

CUDA Compute Unified Device Architecture. 13, 17, 20, 31

DOF Degree Of Freedom. 24, 25

FLOP Floating Point Operation. 25, 28, 29

FLOPS Floating Point Operations per Second. 23

GPGPU General-purpose computing on graphics processing units. 7, 13, 23, 24, 29, 31

GPU Graphics Processing Unit. 7, 13, 15, 18, 20, 23, 25, 28, 29, 31

MPI Message Passing Interface. 23

PDE Partial Differential Equation. 7, 9

PMPUM Parallel Multilevel Partition of Unity Method. 7, 9, 13, 16–19, 24, 26, 28, 29, 31

SIMT Single Instruction Multiple Threads. 13

List of Algorithms

1 Introduction

Many real world problems can be modeled with Partial Differential Equations (PDEs). For example a wave traveling through a medium can be modeled by a PDE. An application of this is the modeling of a vibrating membrane. When looking at the time-independent form of this PDE an example of an elliptic PDE, the Helmholtz equation, arises.

Since for many PDEs no exact solution can be found, there exists a variety of methods which give an approximate solution to those PDEs. One method which can be applied to find an approximate solution for elliptic PDEs is the Parallel Multilevel Partition of Unity Method (PMPUM) which is introduced in [Sch03]. This method uses a Galerkin approach to discretize the PDE. The major computational effort in this method is needed for the discretization of the differential operator.[Sch03, p. 153]

General-purpose computing on graphics processing units (GPGPU) could be a way to improve the performance of the implementation of the PMPUM. In this work we focus on the applicability of GPGPU on the major computational task of the PMPUM. An implementation of the discretization of the differential equation, which is the assembly of the stiffness matrix in the PMPUM, on a Graphics Processing Unit (GPU) is presented in this work.

In this thesis we start by presenting the needed parts of the theoretical background of the PMPUM. Here we give a PDE as a model problem and present the discretization used. After that the implementation is described and properties of the same are listed. The comparison of the performance of the GPGPU implementation and the implementation presented in [Sch03] is given afterward. Finally we summarize the results of this thesis and give an outlook on further improvements.

2 Discretization of elliptic partial differential equations

In the PMPUM the discretization of the elliptic PDE is a major computational task. In this chapter a short introduction to the PMPUM is given. We will be focusing on the discretization of the differential equation. For a detailed description of the whole method see [Sch03], where all the theoretical concepts used in this work are taken from.

2.1 Model Problem

The PMPUM is a mesh-free method for the approximation of the solution of elliptic PDEs. For the better understanding of the concepts used we introduce the following model problem of Helmholtz type:

$$\begin{aligned} \Delta u &= f \text{ in } \Omega \subset \mathbb{R}^d \\ u &= g \text{ on } \Gamma_D \\ u_\nu &= g \text{ on } \Gamma_N = \partial\Omega \setminus \Gamma_D \end{aligned} \tag{2.1}$$

Here u is the solution we search for and f is the given second derivative of u. Γ_D is the part of the boundary with Dirichlet boundary condition and Γ_N the part of the boundary with Neumann boundary conditions, where n_u is the normal derivative of u on the boundary. The two main tasks are the discretization of the differential operator and the treatment of the boundary conditions. Before however presenting the discretization, we introduce the partition of unity.

2.2 Partition of Unity Method

The Partition of Unity Method gives us two necessary properties for the Galerkin method, the local approximability and the inter-element continuity.[Sch03, p. 13] How a Shepard partition of unity can be employed to assure inter-element continuity is described in the following subsection. After that, the construction of the local approximation space is presented.

2.2.1 Shepard partition of Unity

The domain Ω of the problem can be covered by overlapping patches ω_i which are based on an arbitrary set of points.[Sch03, p. 14 et seq.] The construction of such a cover is out of the scope of this work and can be found in [Sch03, p. 98 et seqq.]. To construct the shape functions φ_i, needed in a scattered data approximation, an inverse distance weighting is used in the Shepard's method. Since global weight functions would result in a dense matrix, a localized version is used:

$$\varphi_i(x) = \frac{W_i(x)}{\sum_{\omega_k \in C_i} W_k(x)} \tag{2.2}$$

Here W are piecewise linear splines in our case and C_i are neighbor patches of patch ω_i. This local version results in a complexity of $O(1)$ for the local evaluation and a sparse stiffness matrix. The weight functions $\varphi_i(x)$ form a partition of unity.[Sch03, p. 16 et seq.]

2.2.2 Approximation Space

On a patch ω_i the Shepard functions φ_i are multiplied with a function from the local approximation space. This results in the global approximation space:

$$V^{PU} = \sum_i \varphi_i V_i^{p_i} = \sum_i \varphi_i \langle \{\psi_i^n\} \rangle = span\langle \{\varphi_i \psi_i^n\} \rangle \tag{2.3}$$

Here ψ_i^n are the basis functions for the local approximation space $V_i^{P_i}$. Since the inter-element continuity is provided by Shepard's partition of unity an arbitrary function space can be used for approximation on the patch.[Sch03, p. 13 et seq.]

2.3 Galerkin Discretization

The discretization of the differential equation (2.1) via the Galerkin approach employs the weak form $a(u, v) = l(v)$ which is defined by:

$$\begin{aligned} a(u,v) &= \int_\Omega \nabla u \nabla v + \int_\Omega uv + \int_{\Gamma_D} u(\beta v - v_\nu) - u_\nu v \\ l(v) &= \int_\Omega fv + \int_{\Gamma_D} g_D(\beta v - v_\nu) + \int_{\Gamma_N} g_N v \end{aligned} \tag{2.4}$$

Here v is an arbitrary function from test space V: $v \in V \subset H_g^1(\Omega)$. A regularization parameter β is introduced to assure that the resulting matrix is positive definite. The boundary conditions are generally subdivided into Dirichlet and Neumann boundary conditions which is denoted by the index D and N respectively, where ν is the normal derivative. For more details on the weak formulation see [Sch03, p. 32 et seqq.] As shown in [Sch03, p. 32] the functions $\varphi_i\psi_i^n$ may be used as trial and test function in a Galerkin method without any modification.

For the Galerkin approach we need to compute the entries of the stiffness matrix

$$A = (A_{(i,n),(j,m)}), with A_{(i,j),(j,m)} = a(\varphi_j\psi_j^m, \varphi_i\psi_i^n) \tag{2.5}$$

and the entries of the right-hand side vector:

$$\hat{f} = (\hat{f}_{(i,n)}) \text{ with } \hat{f}_{(i,n)} = l(\varphi_i\psi_i^n) \tag{2.6}$$

To compute the entries of the stiffness matrix we need to compute the integrals for the stiffness matrix

$$\int_\Omega \nabla\varphi_i\psi_i^n\nabla\varphi_j\psi_j^m + \int_\Omega \varphi_i\psi_i^n\varphi_j\psi_j^n + \int_{\Gamma_D} \varphi_i\psi_j^n(\beta\varphi_j\psi_j^m - (\varphi_j\psi_j^m)_\nu) - (\varphi_i\psi_i^n)_\nu\varphi_j\psi_j^m$$

and the right-hand side

$$\int_\Omega f\varphi_i\psi_i^n + \int_{\Gamma_D} g_D(\beta\varphi_i\psi_i^n - ((\varphi_i\psi_i^n)_\nu) + \int_{\Gamma_N} g_N\varphi_i\psi_i^n$$

.

For the implementation we use the model problem defined in equation (2.1). Here we choose $g = 0$. This gives us the task to compute the integral for the operator and the corresponding right-side:

$$\int_\Omega \nabla\varphi_i\psi_i^n\nabla\varphi_j\psi_j^m + \int_\Omega \varphi_i\psi_i^n\varphi_j\psi_j^n \qquad \int_\Omega f\varphi_i\psi_i^n \tag{2.7}$$

The integral on the whole domain needs to be computed. Since the basis functions may differ on the patches, the integral needs to be computed on all intersections of patches. This results in the following integrals for all patches, ω_i, ω_j:

$$\int_{\Omega\cap\omega_i\cap\omega_j} \nabla\varphi_i\psi_i^n\nabla\varphi_j\psi_j^m + \int_{\Omega\omega_i\omega_j} \varphi_i\psi_i^n\varphi_j\psi_j^n \int_{\Omega\omega_i\omega_j} f\varphi_i\psi_i^n \tag{2.8}$$

On these integration cells an integration formula is applied. [1] At the given integration points the functions needed for the evaluation of the bilinear and the linear form need to be evaluated. For the local approximation spaces the polynomial space with different degrees is used. When repeated for all integration cells this results in the task given in equation (2.7). The details of the procedure are explained in context of the implementation in the next chapter.

[1] We use Gauß-Legendre integration formulas in the following. Others may be used as well.

3 Implementation

In this chapter the implementation of the assembly of the stiffness matrices in the PMPUM using GPGPU is described. First an overview over the CUDA-technology is given. After these basics, the data structures needed and the scheduling of the tasks are outlined. The Chapter is completed with the presentation of the kernel and the discussion of its properties.

3.1 CUDA technology

Compute Unified Device Architecture (CUDA) is a technology for GPGPU from *NVIDIA*. It is a general purpose parallel computing architecture, which consists of a new parallel programming model and instruction set architecture and simplifies the usage of *NVIDIA*'s GPUs for GPGPU.[NVI11, p. 3 et seq]. In the following section only the concepts of CUDA used in the implementation for this work are presented. For an overview of the CUDA technology see [NVI11], from which all information in this section is taken. Since *C++* was used for the implementation we present some details of *CUDA C*. A CUDA application consists of host code running on the Central Processing Unit (CPU) and device code running on the GPU. Device code is referred to as the kernel in the following. Before a kernel can be launched the arguments need to be transferred from host to device. And after the kernel is finished the results are transferred from device to host.[NVI11, p. 11 et seq] This basic concept is outlined in figure 3.1. The following two subsections outline the concepts for the execution of a kernel on the device and the memory hierarchy of the device.

3.1.1 Thread Hierarchy

A CUDA kernel execution is organized in a thread hierarchy. The grid consists of all threads running a kernel. The grid is subdivided into thread blocks. A thread block is executed on one multiprocessor of the device[1]. The number of blocks executed on a single multiprocessor in parallel depends on the resource usage of a thread block. Each thread block itself contains a number of threads which are mapped to 1 to 4 so called warps. All threads of a warp are all executed all at the same time. For more details on the Single Instruction Multiple Threads (SIMT) architecture of CUDA see [NVI11, p. 85 et seqq.] Threads of a single thread block can be synchronized efficiently, but syncing over multiple blocks is only possible by using relatively expensive atomic operations on global memory.[NVI11, p. 10] The thread hierarchy is shown in figure 3.2.

[1] A GPU consists of several multiprocessor, the number depends on the device used.

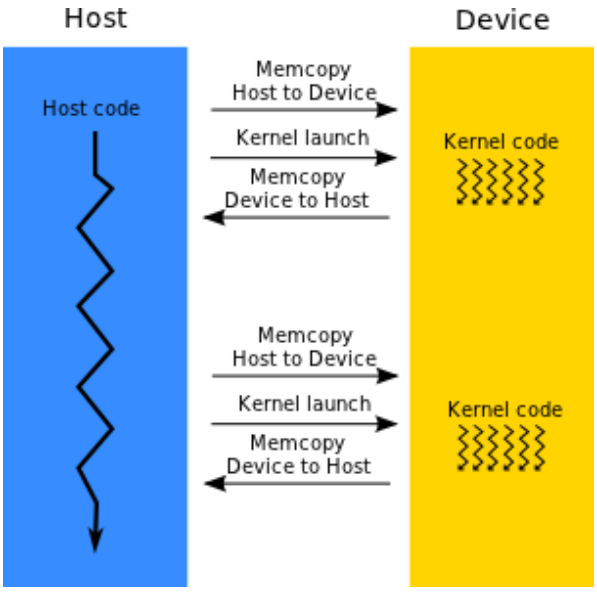

Figure 3.1: The CUDA Execution Model, own graphic based on [NVI11, p. 13]

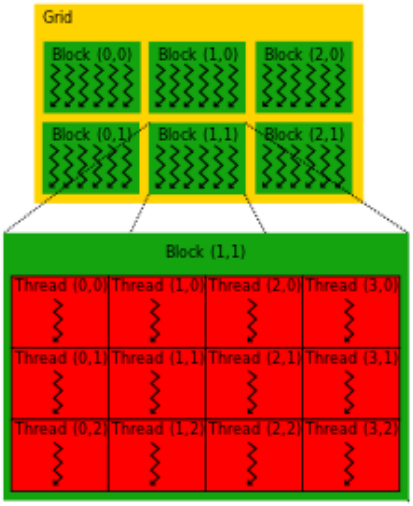

Figure 3.2: The CUDA Thread Hierarchy, own graphic based on [NVI11, p. 9]

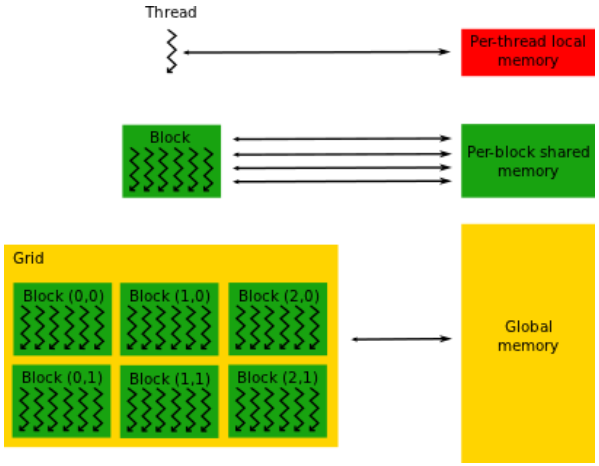

Figure 3.3: The CUDA Memory Hierarchy, own graphic based on [NVI11, p. 11]

When parallelizing a program with CUDA, the question is how to map the tasks to thread blocks and threads. The mapping however is constrained by the structure of the memory of the device.

3.1.2 Memory Hierarchy

The memory on a GPU is structured as a hierarchy as well. The main memory of the device is also called global memory since it is accessible from all threads. All transfers from host to device or vice versa use the global memory as destination or source respectively.[NVI11, p. 20 et seqq][2] Additionally a constant memory, which can be used for constants for the kernel which can also be accessed by all threads in a read-only fashion, exists.[NVI11, p. 96 et seq] Each multiprocessor has a fast memory which is shared by all threads running on this multiprocessor. This memory is referred to as shared memory.[NVI11, p. 96]. Each thread uses registers. The number of registers and the amount of shared memory used determines the maximum thread block size. This is caused by the fact that the registers of a multiprocessor are divided by all threads which are executed in parallel. If a thread needs more memory than registers are available then local memory is used. The local memory is a part of global memory but is organized in a more efficient way to improve access patterns but is constrained in size for each thread.[NVI11, p. 95 et seq] This hierarchy is shown in figure 3.3.

[2]Texture and surface memory is not covered in this summary, but can be destination of transfers from host to device too. See [NVI11, p. 39 et seqq.] and [NVI11, p. 45 et seq.] for details.

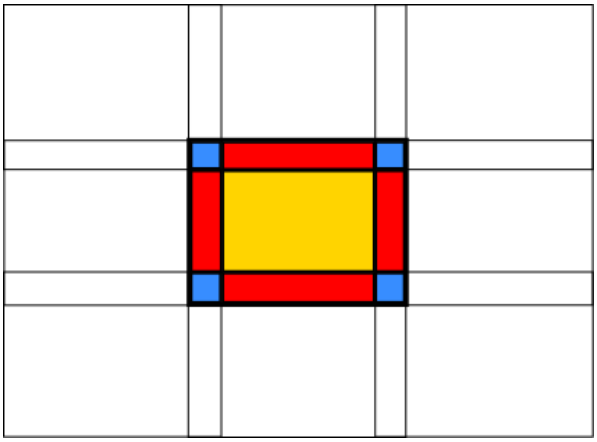

Figure 3.4: The integration cells $\Omega \cap \omega_k \cap \omega_l$ on one patch ω_k, graphic based on [Sch03, p. 25]

3.2 Computational task

To present the implementation of the assembly of the stiffness matrices in the PMPUM we start with recalling the computational task given in section 2.3:

$$\int_\Omega \nabla\varphi_i\psi_i^n\nabla\varphi_j\psi_j^m + \int_\Omega \varphi_i\psi_i^n\varphi_j\psi_j^n \qquad \int_\Omega f\varphi_i\psi_i^n \tag{3.1}$$

As described previously for the integration the domain needs to be split into the intersections of the patches. This is caused by the fact, that on any two patches different local approximation spaces $V_i^{P_i}$ may be used. Figure 3.4 shows the intersections of one patch for an uniform grid in 2D.[3]

We can see that the patch ω_k, marked with a bold line, has intersections with all its neighbors, called C_k in the following. These intersections form the integration cells $\omega_k \cap \omega_l : \omega_l \in C_k$. For each integration cell for all pairs of patches (ω_i, ω_j) with support on the cell, the following integrals need to be computed:

$$\int_{\Omega\cap\omega_i\cap\omega_j} \nabla\varphi_i\psi_i^n\nabla\varphi_j\psi_j^m + \int_{\Omega\cap\omega_i\cap\omega_j} \varphi_i\psi_i^n\varphi_j\psi_j^m \qquad \int_{\Omega\cap\omega_i\cap\omega_j} f\varphi_i\psi_i^n \tag{3.2}$$

[3]The alignment of the patches is not necessary but employed in many cases.

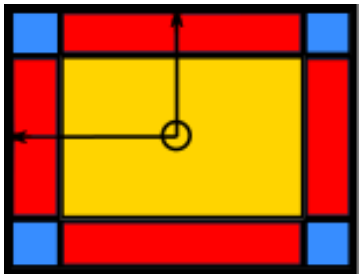

Figure 3.5: The extent of a patch is given by its center and its radius

3.3 Data structures

To solve the computational task (3.2) presented in the previous section we need to compute the following. The weight function φ_i, gradients of the weight function $\nabla\varphi_i$, the basis functions ψ_i^n and the gradients of the basis functions $\nabla\psi_i^n$ need to be computed on each cell to evaluate the problem on the cell. To do so the following information is needed.

- geometric extent of the cell
- for all patches ω_i with support on this cell
  - geometric extent of the patch
  - the type of the weight functions used to define the patch
  - the type of the function space $V_i^{P_i}$ used for approximation on the patch ω_i

The extent of the cell is given by 2 points, which define the opposing corners. The extent of the patches is represented as a center point and the radius in all directions. This is used in the PMPUM framework to ease the change of extent of a patch. The storage of the extent of the patch is shown in figure 3.5 The types of the weight functions and the function spaces used, are given as template parameters but internally represented as enumeration types since CUDA doesn't support the implementation of templates defined on the host. So for a given cell the following data described above needs to be transferred to the device. After the evaluation the resulting matrices and vector need to be transferred back to the host and added to the global data structures. Since memory on the device is more efficient when using static memory allocation, the maximal values for dimension, polynomial degree and number of patches with support on one cell are set at time of compilation.

3.4 Scheduling

In CUDA it is expensive when the threads of a thread block diverge. In the evaluation of the cells, in many cases one needs to loop over all patches with support on a cell. To reduce branch divergence only cells with the same number of patches should be evaluated in parallel

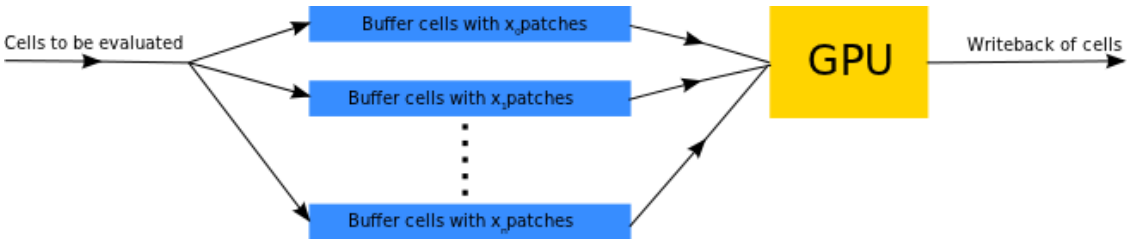

Figure 3.6: The scheduling for evaluation of the cells

in one thread block. Since in the implementation the relationship of the cell to its neighbors is not exploited, the ordering of the evaluations can be changed.[4] This is done to execute only those cells in parallel on which the same number of patches have support. On the host the stream of cells generated by the PMPUM Framework is buffered for each number of patches with support on the cells. When the maximum number of cells executable on the device are buffered, the data is transferred to the device. Then the kernel which is described in the next section is executed and the results are transferred back to the host and added to the global data structures. This process is shown in figure 3.6.

3.5 Kernel

When the schedule of cells to evaluate is transferred to the device, the kernel is launched. As described in section 3.1 the Kernel is executed in parallel on the GPU. The number of threads per block is fixed to 128. The number of blocks launched b is given by $\min b \in \mathbb{N} : 128b >= c$, where c is the number of cells in the schedule. Each thread operates on exactly one cell. Since no synchronization between threads is used in the implementation, in the following only the work flow for one thread will be presented. After this some properties of the kernel are discussed.

3.5.1 Kernel Work Flow

The Kernel evaluates the differential operator, the mass operator and the right side linear form in the weak form (2.5) on the cell. Therefore it integrates them on the cell. For the evaluation the values ψ_i^n and the gradient values $\nabla\psi_i^n$ of the basis functions on all patches with support on the cell are needed. These again require the weight function φ_i and the gradients of the weight function $\nabla\varphi_i$ of the patches. The *Kernel* procedure in algorithm 3.1 outlines the work flow of one thread of the kernel. The *Host Function* represents the procedure on the host which creates the schedules, transfers the data and launches the kernel.

[4] A patch has support on a number of cells, which could be exploited in evaluating all functions on a patch only once instead for every cell on which it has support.

Algorithm 3.1 The kernel and the calling host function in pseudo code

```
procedure KERNEL(cells*,result*)                          // execute in parallel on device
    INITRESULT(result[threadIdx.x]) // initializes the sizes of the result matrices and vector
and sets the entries to zero
    for all intpoint ∈ integrationpoints do     // loop over all integration points of the cell
        EVALUATEWEIGHTSANDWEIGHTGRADIENTS(intpoint, patchoncell)
            // evaluate the weights and the weight gradients for the integration point on all
patches with support on the cell
        ASSEMBLEBASEFUNCTIONS(intpoint, weights, weightgradients, patchoncell)
                // assemble the basis functions for all patches with support on the cell at the
integration point
        EVALUATEOPERATORS(intnode, intweight, values, gradientvalues)
                        // evaluate the bilinear form, the mass bilinear form and linear form at
the integration point, weight them with the integration weight and add them to the result
matrices and vector of the cell
    end for
end procedure
procedure HOST FUNCTION                                                  // create the schedules
    KERNEL(cells*,results*)                                  // execute in parallel on device
end procedure
```

The integration level depends on the polynomial degree of the local approximation space of the patches with support on the cell. The number of integration points therefore depends on the polynomial degree p used.[5] Since the degree of the polynomial space used has significant influence on the performance, in the following procedures depending on the polynomial degree are marked.

The procedure *EvaluateWeightsAndWeightGradients* evaluates the weights $\varphi_i(x) = \frac{W_i(x)}{\sum_{\omega_k} W_k(x)}$ and the gradients of the weight $\nabla\varphi_i$ for the integration point on all patches with support on the cell. Here W_i is the weight function on patch i which is chosen to be a B-spline[6].

In *AssembleBaseFunctions* the product $\varphi_i\psi_i^n$ is generated. Here ψ_i^n is are the basis functions of the local approximation space of patch i. Therefore ψ_i^n needs to be generated for the appropriate polynomial degree and stretched to the extent of the patch i. The complexity of this procedure depends on the polynomial degree used.

Finally in *EvaluateOperators* the differential operators are evaluated to generate the stiffness matrices $\nabla\varphi_i\psi_i^n\nabla\varphi_j\psi_j^m$ and the right-side $f\varphi_i\psi_i^n$. The resulting matrix for the cell depends on the polynomial degree used.

[5] In the PMPUM enrichment functions can be added to the local approximation space to improve the approximation at singularities. Since these functions can be much more complex a higher level or integration is needed in such a case

[6] Only piecewise linear splines where implemented for this work.

The integration over the cell solves the computation tasks stated in (3.2). Executed for all cells in Ω the Kernel generates the stiffness matrices which can be used to discretize a differential equation as stated in 2.3.

3.5.2 Kernel Properties

The kernel computes a lot of complex functions. This results in the usage of many registers. The achieved occupancy of the multiprocessors is therefore limited and leads to a performance penalty. The suboptimal resource usage is caused by the complexity of the functionality of the sub functions. All complex functionality however is in the loop over the integration points. Therefore a multi-kernel version could split the functionality inside the loop into different kernels. This would require the temporary results to be stored in global memory. The critical factor is expected to be the number of kernel launches and too less work for one kernel launch. A large global memory could reduce this effect, but is not expected to be significant for available GPUs.

A reduced number of registers used by one thread could improve the performance significantly since more thread blocks could be executed on one multiprocessor in parallel. This would not only improve the occupancy of the multiprocessor but also improve latency hiding. [7]

The kernel doesn't use any shared memory. This is suboptimal because a relative fast memory in the hierarchy is not used. But the sizes of most data items depends on the degree of the polynomials used for local approximation and the maximal number of patches per cell. Since the shared memory is limited by 48KB, it could only be used for small degrees of the polynomials. To avoid this it isn't used. It isn't totally wasted since it is partly used as cache for the global memory by the CUDA compiler instead.

Due to the scheduling described in section 3.4 the branch divergence could be reduced to a very good level for such a complex kernel. It should be noted that this could also be achieved by the synchronization between threads, but the kernel doesn't use any synchronization.

3.5.3 Limitations

The implementation of the kernel is limited in some aspects:

1. The function spaces used for local approximation is limited to the monomial space.
2. The maximal degree of the polynomials used in the monomial space is set at compile time.[8]
3. For the weight functions of the patches φ_i only piecewise linear splines are implemented.

[7] Latency hiding describes the situation, that when one thread is waiting for memory there are enough others to utilize the GPU in the meantime.

[8] Implied by the usage of static memory allocation on the device.

4. The maximal number of patches which have support on a cell is set at compile time.[8]
5. The dimension of problems is set at compile time[8] and currently limited to 2.

4 Results

In this chapter the performance of the GPGPU implementation is compared to a CPU implementation presented in [Sch03]. First the hardware, the compilers and the metrics that were used are described. After this, results of different experiments are explained and finally some hints are given how to continue improving the performance using GPGPU.

4.1 Hardware and Metrics

The system on which all measurements were taken has the configuration shown in table 4.1.

Operating System	Scientific Linux (6.1) with Kernel 2.6.32.1
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 (NullMPI[a] 0.7) (-fvisibility-inlines-hidden -fstrict-aliasing -O3 -foptimize-sibling-calls -fstrength-recude -march=native)
CUDA compiler (device code)	nvcc 4.0 V.2.1221 (-arch=sm_2)

Table 4.1: System configuration

[a]For details of NullMPI see [Ins12]

The processing power of the NVIDIA GeForce GTX 560 Ti is 115 Floating Point Operations per Second (FLOPS) in double precision.[Har12] Since the CPU implementation uses only one core[1] of the Intel i7-2600 @ 3.4GHz the processing power of the CPU sinks from 83.24 FLOPS[Bü11] to 20.1 FLOPS in double precision. That theoretically results in a performance advantage of factor ≈ 5.5 for the GPU. This theoretical advantage can only be fully achieved in relatively few cases since the flexibility of the execution on the GPU is limited.

[1]The implementation presented in [Sch03] uses Message Passing Interface (MPI), but it was compiled to use one thread.

Next we present the metric for the performance measurements. The performance of the implementation is measured in terms of Degree Of Freedom (DOF):

$$\frac{t/\text{DOF}}{\text{DOF}}$$

Here t is the time in seconds and the DOF value is given by:

$$N * \frac{(p+1)(p+2)}{2}$$

Here in our case $N = n^2$ where n is the number of cells in one dimension, using an uniform cover. The number of cells in one dimensions is calculated by $n = 2^l$. Here l is the level on which the problem is evaluated. In this context the level used in the PMPUM is meant and not the level of the integration formula. See [Sch03, p. 97 et seqq.] for details on the construction of the levels in the PMPUM. And p is the degree of the polynomials used in the local approximation spaces.

4.2 Experiments

In all experiments only those parts of the implementation concerning the assembly of the stiffness matrices are measured. These are:

- the scheduling in the GPGPU version
- the evaluation of the cells
- the insertion of the results into the global stiffness matrix

4.2.1 Experiment 1

In this first experiment we want to get a general impression of the performance of the implementation. When using polynomial spaces of degree p for local approximation the level of the integration l on the cells is sufficient when chosen to be $l = p$. We expect the performance of the GPGPU implementation to be higher for higher degree of the polynomials used. This is expected due to the higher number of operations per global memory access. Figure 4.1 shows the comparison of the performance for the polynomial degrees 1 and 2.

The different lengths of the plots result from the implementation of the PMPUM framework. The storage used depends on the number of cells and the polynomial degree. The framework however takes the next higher power of 2 as the number of cells in one direction for memory allocation. Due to that, only experiments with the number of cell equal to a power of 2 were

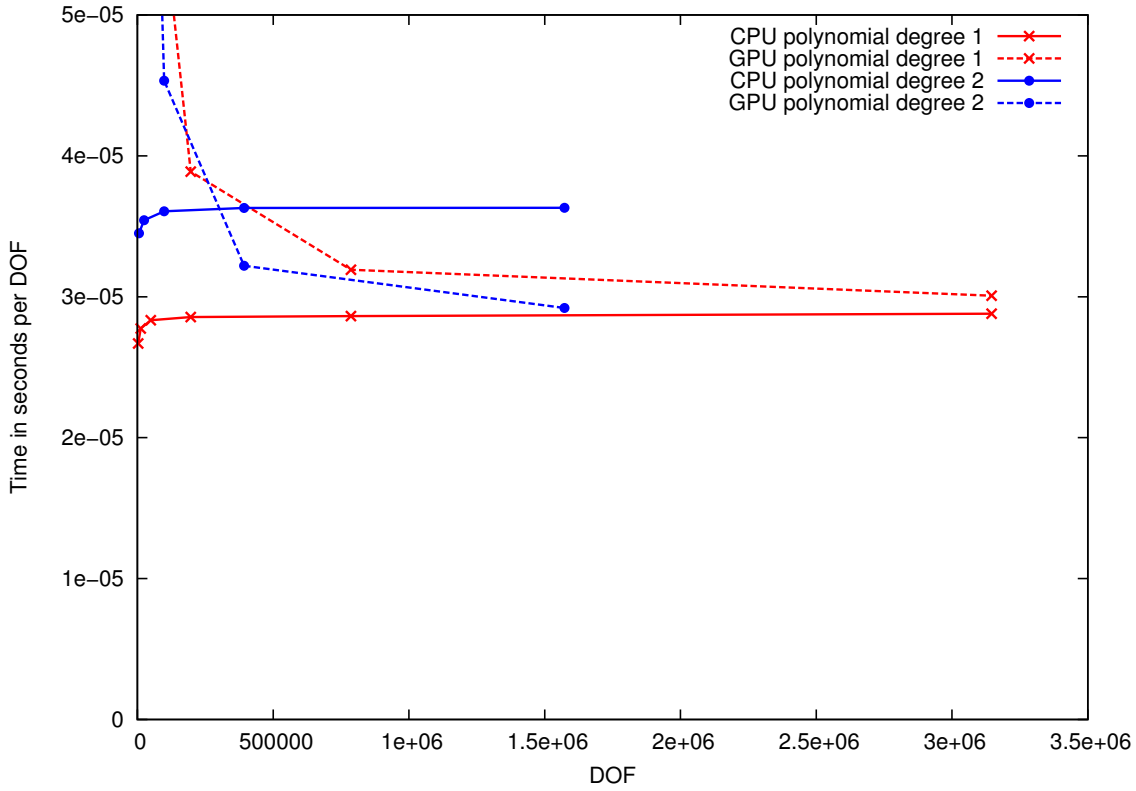

Figure 4.1: Comparison of (t/DOF)/DOF for polynomial degrees 1 and 2

carried out. Since the memory of the host is limited (16 GiB) only experiments up to a certain number of cells per direction for different polynomial degree were carried out. This causes the plots to be of different length.

The figure shows, that for polynomial degree 1 the performance of the CPU implementation is higher. For polynomial degree 2 the GPU implementation performs better for cases of the problem with DOF greater than about $3 * 10^5$. To get a better impression of the values the results are shown in the tables 4.2 and 4.3.

The ratio of operations to memory access increases with growing polynomial degree. Due to that we expect this effect to increase with growing polynomial degree. Figure 4.2 shows the same experiment for polynomial degrees 3,4 and 5.

As can be seen in the figure the performance for an increased polynomial degree increases as expected. As can be seen also in the tables 4.4, 4.5 and 4.6, the performance advantage increases nearly to factor 2. This means that for higher polynomial degree, polynomials with a degree increased by one can be used resulting in the same performance. The dependency on the polynomial degree is based on the number of Floating Point Operation (FLOP)'s executed for one memory access. Since the number of evaluations is based on the quadrature formulas

DOF	seconds/DOF		Ratio GPU/CPU
	CPU	GPU	
$3.07 * 10^3$	$2.67 * 10^{-5}$	$5.96 * 10^{-4}$	22.32
$1.23 * 10^4$	$2.77 * 10^{-5}$	$1.70 * 10^{-4}$	6.14
$4.92 * 10^4$	$2.83 * 10^{-5}$	$6.52 * 10^{-5}$	2.30
$1.97 * 10^5$	$2.86 * 10^{-5}$	$3.89 * 10^{-5}$	1.36
$7.86 * 10^5$	$2.86 * 10^{-5}$	$3.19 * 10^{-5}$	1.12
$3.15 * 10^6$	$2.88 * 10^{-5}$	$3.01 * 10^{-5}$	1.04

Table 4.2: Comparison of the performance for polynomial degree 1

DOF	seconds/DOF		Ratio GPU/CPU
	CPU	GPU	
$6.14 * 10^3$	$3.45 * 10^{-5}$	$2.98 * 10^{-4}$	8.65
$2.46 * 10^4$	$3.54 * 10^{-5}$	$9.61 * 10^{-5}$	2.71
$9.83 * 10^4$	$3.61 * 10^{-5}$	$4.53 * 10^{-5}$	1.26
$3.93 * 10^5$	$3.63 * 10^{-5}$	$3.22 * 10^{-5}$	0.89
$1.57 * 10^6$	$3.63 * 10^{-5}$	$2.92 * 10^{-5}$	0.80

Table 4.3: Comparison of the performance for polynomial degree 2

which depend on the polynomial degree we assume that the integration level is the critical factor for the performance ratio. In the next experiment we check if this assumption holds.

4.2.2 Experiment 2

As stated in the section before, we expect the performance to depend on the level of integration. So in this experiment the level of the quadrature is set to a fixed higher level, independent of the degree of the polynomials used in the approximation spaces. This is especially useful in the context of the PMPUM in which enrichment functions may be added to the local

DOF	seconds/DOF		Ratio GPU/CPU
	CPU	GPU	
$1.02 * 10^4$	$6.26 * 10^{-5}$	$2.03 * 10^{-4}$	3.25
$4.10 * 10^4$	$6.47 * 10^{-5}$	$8.14 * 10^{-5}$	1.26
$1.64 * 10^5$	$6.55 * 10^{-5}$	$5.02 * 10^{-5}$	0.77
$6.55 * 10^5$	$6.59 * 10^{-5}$	$4.15 * 10^{-5}$	0.63
$2.62 * 10^6$	$6.63 * 10^{-5}$	$3.95 * 10^{-5}$	0.60

Table 4.4: Comparison of the performance for polynomial degree 3

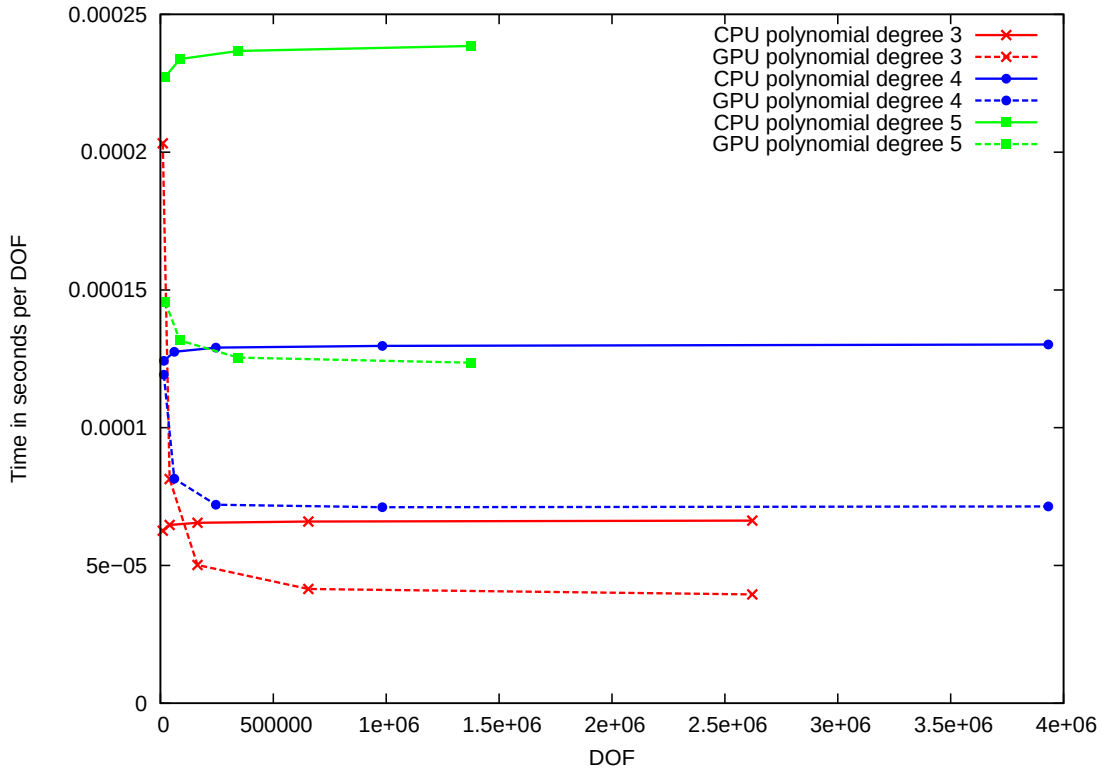

Figure 4.2: Comparison of (t/DOF)/DOF for polynomial degrees 3 to 5

DOF	seconds/DOF		Ratio GPU/CPU
	CPU	GPU	
$1.54 * 10^4$	$1.24 * 10^{-4}$	$1.19 * 10^{-4}$	0.96
$6.14 * 10^4$	$1.28 * 10^{-4}$	$8.15 * 10^{-5}$	0.64
$2.46 * 10^5$	$1.29 * 10^{-4}$	$7.20 * 10^{-5}$	0.56
$9.83 * 10^5$	$1.30 * 10^{-4}$	$7.12 * 10^{-5}$	0.55
$3.93 * 10^6$	$1.30 * 10^{-4}$	$7.14 * 10^{-5}$	0.55

Table 4.5: Comparison of the performance for polynomial degree 4

DOF	seconds/DOF		Ratio GPU/CPU
	CPU	GPU	
$2.15 * 10^4$	$2.27 * 10^{-4}$	$1.46 * 10^{-4}$	0.64
$8.60 * 10^4$	$2.34 * 10^{-4}$	$1.32 * 10^{-4}$	0.56
$3.44 * 10^5$	$2.37 * 10^{-4}$	$1.25 * 10^{-4}$	0.53
$1.38 * 10^6$	$2.39 * 10^{-4}$	$1.24 * 10^{-4}$	0.52

Table 4.6: Comparison of the performance for polynomial degree 5

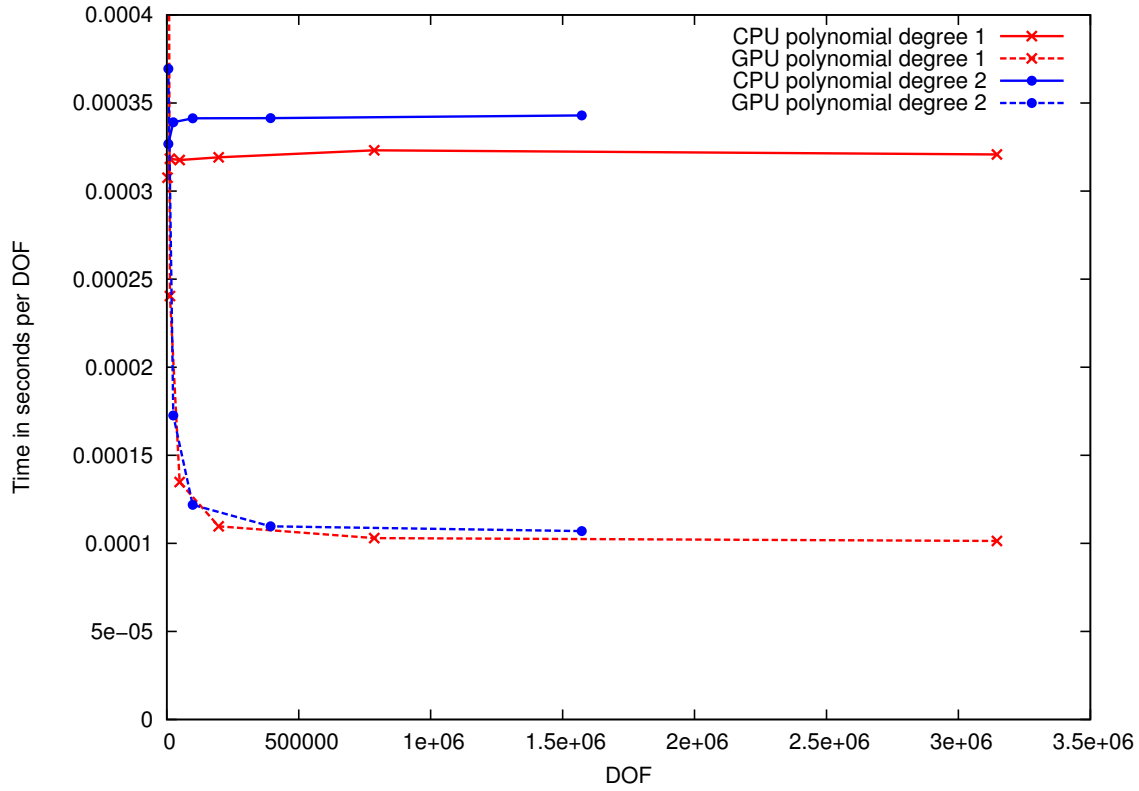

Figure 4.3: Comparison of (t/DOF)/DOF for polynomial degrees 1 and 2 with integration level 10

approximation spaces. These enrichment functions may be of arbitrary type. Due to that, higher level quadrature formulas are needed for the computation of the integrals. Figure 4.3 shows the comparison of the performance in the case of a fix integration level 10.

The figure shows that for a fixed high integration level, the GPU implementation out performs the CPU implementation by about a factor 3. This can be verified by the values in 4.7 and 4.8. This shows, that the ratio of FLOP's to memory accesses is too small for small polynomial degrees when applying no higher level quadrature formula. This should result in a good performance when adding enrichment functions used in the PMPUM, since higher level integration formulas need to be applied in this case.

DOF	seconds/DOF		Ratio GPU/CPU
	CPU	GPU	
$3.07 * 10^3$	$3.08 * 10^{-4}$	$6.68 * 10^{-4}$	2.17
$1.23 * 10^4$	$3.18 * 10^{-4}$	$2.40 * 10^{-4}$	0.76
$4.92 * 10^4$	$3.18 * 10^{-4}$	$1.35 * 10^{-4}$	0.42
$1.97 * 10^5$	$3.19 * 10^{-4}$	$1.10 * 10^{-4}$	0.34
$7.86 * 10^5$	$3.23 * 10^{-4}$	$1.03 * 10^{-4}$	0.32
$3.15 * 10^6$	$3.21 * 10^{-4}$	$1.01 * 10^{-4}$	0.32

Table 4.7: Comparison of the performance for polynomial degree 1 and integration level 10

DOF	seconds/DOF		Ratio GPU/CPU
	CPU	GPU	
$6.14 * 10^3$	$3.27 * 10^{-4}$	$3.69 * 10^{-4}$	1.13
$2.46 * 10^4$	$3.39 * 10^{-4}$	$1.73 * 10^{-4}$	0.51
$9.83 * 10^4$	$3.41 * 10^{-4}$	$1.22 * 10^{-4}$	0.36
$3.93 * 10^5$	$3.41 * 10^{-4}$	$1.10 * 10^{-4}$	0.32
$1.57 * 10^6$	$3.43 * 10^{-4}$	$1.07 * 10^{-4}$	0.31

Table 4.8: Comparison of the performance for polynomial degree 2 and integration level 10

4.3 Further Improvements

An optimization of the kernel could reduce the numbers of registers used by each thread. This would result in a better occupancy which would result in a better performance because, among other reasons, of improved latency hiding. [2]

A completely different approach could exploit the neighborhood relation of the patches, as described in section 3.4, to reduce evaluations required. But since the experiments showed that the evaluations aren't expensive an improvement of the memory access pattern would be more promising.

The performance advantage of the GPU implementation increases with higher level for the integration. This is caused by a higher ratio of FLOP's per memory access. Since in the 3 dimensional case more evaluations are needed, applying it to 3 dimensional problems should result in a better performance for this case too.

The implementation of enrichment functions could show the full potential of the GPGPU implementation in the context of PMPUM. This is claimed because more evaluations are needed in this case. As discussed before a higher number of evaluations lead to a better performance compared to the CPU.

[2] Latency hiding describes the situation, that when one thread is waiting for memory there are enough others to utilize the GPU in the meantime.

5 Conclusion

In this work a GPGPU implementation of the PMPUM was presented. First the theoretical background was outlined to introduce the context of the implementation. The implementation and some of its properties were presented in a second step. Subsequently the results of performance measurements were shown and discussed.

The implementation showed that in many cases arising in PMPUM a GPGPU approach can be applied to improve performance. For polynomial approximation spaces with degree greater 1 the GPU implementation outperformed the CPU implementation. The performance advantage was shown to depend on the level of integration. For higher levels of integration the advantage of the GPU version grows. Since this case occurs frequently in the PMPUM the potential of the GPGPU approach was shown.

When additional functionality for the use of local enrichment functions is implemented using GPGPU the performance might be increased further. Since the performance of the assembly of the stiffness matrix could be improved significantly using CUDA, the performance of the whole PMPUM could be improved by implementing further parts of it using GPGPU.

Bibliography

[Bü11] M. Büchel. Sandy Bridge: Core i7 2600K und Core i5 2500K, 2011. URL http://www.ocaholic.ch/xoops/html/modules/smartsection/item.php?itemid=452&page=6. (Cited on page 23)

[Har12] Hard Tecs 4U. NVIDIA GeForce GTX 560 TI im Test, 2012. URL http://ht4u.net/reviews/2011/nvidia_geforce_gtx_560_ti_msi_n560ti_twin_frozr_2/index2.php. (Cited on page 23)

[Ins12] Institute for Numerical Simulation, Universität Bonn. NullMPI - MPI substitute library, 2012. URL http://wissrech.ins.uni-bonn.de/research/projects/nullmpi/. (Cited on page 23)

[NVI11] NVIDIA. *NVIDIA CUDA C Programming Guide 4.0*. 2011. (Cited on pages 13, 14 and 15)

[Sch03] M. A. Schweitzer. *A Parallel Multilevel Partition of Unity Method for Elliptic Partial Differential Equations*, volume 29 of *Lecture Notes in Computational Science and Engineering*. Springer, 2003. (Cited on pages 7, 9, 10, 11, 16, 23 and 24)

All links were last followed on May 14, 2012.

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)