Institute of Parallel and Distributed Systems
University of Stuttgart
Universitätsstraße 38
D–7 05 69 Stuttgart

Diplomarbeit Nr. 32 68

Implementierung eines Peridynamik–Verfahrens auf GPU

Patrick Diehl

Course of Study:	Software Engineering
Examiner:	Prof. Dr. Marc Alexander Schweitzer
Supervisor:	M. Sc. Sa Wu
Commenced:	November 14 2011
Completed:	May 15 2012
CR-Classification:	J.2, G.4

A computer will do what you tell it to do, but that may be much different from what you had in mind.

(Joseph Weizenbaum)

Contents

List of Symbols

α

The scaling parameter in the Prototype Microelastic Brittle material.

c

The stiffness constant in the Prototype Microelastic Brittle model in $\frac{\mathrm{kg}}{\mathrm{m}^3}$.

δ

The radius of the horizon of a particle P in m.

η

The relative displacement of two particles P_i and P_j.

$\mathscr{H}_i$

The horizon of a particle P_i.

k

The bulk modulus in the Prototype Microelastic Brittle material in GPa.

Ω

The reference configuration of the body.

$\mathscr{R}$

The body.

ρ

The mass density of the material in the reference configuration Ω in $\frac{kg}{m^3}$.

r_s

The radius of a particle in m.

s_{00}

The critical stretch for bond failure for the Prototype Microelastic Brittle material.

V

The volume of a particle in m^3.

x

The initial position of a particle P relating to the reference configuration Ω.

ξ

The relative position of two particles P_i and P_j in the reference configuration Ω.

List of Figures

List of Tables

List of Listings

List of Acronyms

CUDA	Compute Unified Device Architecture
CPU	Central processing unit
FEM	Finite Element Method
FLOPS	Floating Point Operations Per Second
GPU	Graphics processing unit
HPC	High Performance Computing
MD	Molecular Dynamics
MP	Multiprocessor

MPI	Message Passing Interface
OCL	Open Computing Language
ODE	Ordinary differential equation
PDE	Partial differential equation
PMB	Prototype Microelastic Brittle
PUM	Partition of Unity Method
RAM	Random-Access Memory
SIMT	Single Instruction Multiple Threads
SPH	Smoothed Particle Hydrodynamics

1 Introduction

This thesis examines the use of the Compute Unified Device Architecture (CUDA) the implementation of a Peridynamic technique on a Graphics processing unit (GPU). CUDA is a parallel computing architecture for NVIDIA GPUs. The NVIDIA GPUs are programmable through different industry standard programming languages. In this thesis the programming language "`C` for CUDA" is used to implement the Peridynamic technique. "`C` for CUDA" is similar to the programming language `C` with some restrictions. The software was executed on a *NVIDIA GeForce GTX* 560 *Ti*.

The Peridynamic theory is a non local theory in continuum mechanic, with a focus on discontinuous functions as they arrive in fracture mechanics. The word "**Peridynamic**" is the syncrisis of the two Greek words *πεϱι* (peri = near) and *δύναμη* (dynami = force). The principle of this theory is that particles in a continuum interact with other particles in a finite distance by exchanging forces.

Outline

The chapters cover following topics:

Chapter 2 – Related work: This chapter contains information about other implementations of the Peridynamic technique, which use different kinds of massively parallel architectures.

Chapter 3 – Peridynamics: This chapter contains the basics of the Peridynamic and the discrete Peridynamic equation of motion (equation 3.18).

Chapter 4 – Implementation: This chapter contains a short introduction to Compute Unified Device Architecture (CUDA) and the program flow chart (figure 4.4) of the simulation. For each activity of the program flow chart the used algorithms are introduced.

Chapter 5 – Combination with the Partition of Unity Method: This chapter contains an approach to the combination of the the Peridynamic technique with the Partition of Unity Method (PUM).

Chapter 6 – Results: This chapter contains the results of two simulations: In the 2D case a thick square plate with a initial crack. In the 3D case the impact of a projectile in a cylinder.

Chapter 7 – Conclusion and outlook: This chapter summarizes the results of this thesis and gives an overview of potential improvements and features of the software.

2 Related work

"EMU is the first code based on the Peridynamic theory of solid mechanics"

According to the quote [Sana] *EMU* is the first implementation of the Peridynamic theory, which is implemented in *FORTRAN* 90 and provided from the Sandia National Laboratories. Developers implemented *EMU variants* to add extended features to the standard version. In the aviation industry Dr. Abe Askari works on modeling of fatigue cracks and composite material failure for Boeing Corporation.

In research *Peridynamics PDLAMMPS* and *SIERRA/SM* are the most common tools, which realizing the Peridynamic technique.
Peridynamics PDLAMMPS [PSP$^+$11] is a realization of the Peridynamics in *LAMMPS*. *LAMMPS* [Pli95, Sanb] is a classical molecular dynamics code and the integration of the Peridynamic technique is possible, because the Peridynamic technique is in some sections similar to a molecular dynamics (MD). *Sierra/Solid Mechanics* [SIE11] is based on the Sierra Framework [Edw02] and provides a realization of the Peridynamic theory too. Because it is based on the Sierra Framework a coupling with other SIERRA mechanics codes is possible.

Nowadays most simulations are executed on massively parallel distributed systems. Therefore frameworks like the Message Passing Interface (MPI) to distributed the simulation on different Central processing units exist. It is possible to build PDLAMMPS with MPI support.

Another approach for massively parallel programming is to execute the simulation on GPUs. For this exist Open Computing Language (OCL) and Compute Unified Device Architecture (CUDA). This thesis studies the use of CUDA to realize a massively parallel implementation of the Peridynamic technique.

3 Peridynamics

The Peridynamic theory is a non local theory in continuum mechanics. As described in [ELP] the Partial differential equation (PDE) (3.1) defines linear elastic behavior of solids, according to Newton's second law: $force = mass \times acceleration$.

$$\begin{aligned} \varrho(x)\partial_t^2 u(x,t) &= (Lu)(x,t) + b(x,t), \quad (x,t) \in \Omega \times (0,T) \\ \text{with } (Lu)(x,t) &:= (\lambda + \mu)\, \text{grad div}\, u(x,t) + \mu\, \text{div grad}\, u(x,t) \end{aligned} \tag{3.1}$$

PDE (3.1) contains information about the material with $\varrho(x)$ as the density of the body. The inner tensions and macroscopic forces are described with the Lamé parameters μ and λ. The term $b(x,t)$ defines the extern force at position x at time t. The displacement field is described with $u : \overline{\Omega} \times [0,T] \to \mathbf{R}^d$ with $d \in \{1,2,3\}$.

PDE (3.1) implies that the displacement of the body is twice continuously differentiable. With this assumption it is not possible to model cracks or fracture, because a crack or fracture implies discontinuities in the displacement field u. Discontinuities conflict with the assumption that u is twice continuously differentiable.

An alternative theory to model solid mechanics is the Peridynamic theory, which was introduced by Silling [Sil00] in 2000. The Peridynamic theory formulates the problem with an integro-differential equation. This solves the conflict of the discontinuities and the twice continuously differentiability.

3.1 Peridynamic theory

The word "**Peridynamic**" is the syncrisis of the two Greek words $\pi\epsilon\varrho\iota$ (peri = near) and $\delta\acute{\upsilon}\nu\alpha\mu\eta$ (dynami = force). The principle of this theory is that particles in a continuum interact with other particles in a finite distance by exchanging forces. Some of this concepts are similar to a selection of a Molecular Dynamics (MD).

Figure 3.1 shows the body $\mathscr{R}$ in the reference configuration with particles $P_{1\cdots n}$. Each particle has a initial position x relating to the reference configuration Ω. To define which other particles interact with a particle each particle has a horizon $\mathscr{H}_i$ with the radius δ. All other particles in this horizon $\mathscr{H}_i$ have a bond with the particle P_i and interact with this particle by forces. Metaphorically speaking the particle does not "see" particles outside his own horizon $\mathscr{H}_i$ and is not influenced by them.

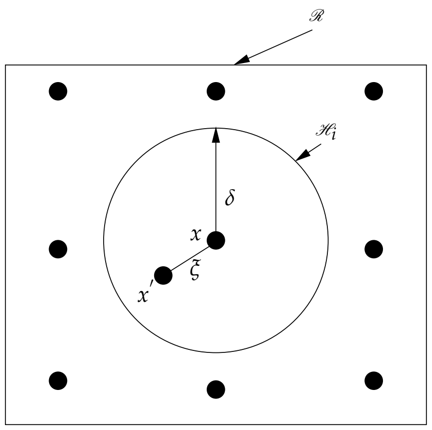


Figure 3.1: The body $\mathscr{R}$ of the continuum with particles $P_{1 \cdots n}$ and the horizon $\mathscr{H}_i$ of particle P_i.

To compute the acceleration of a particle P_i in the reference configuration Ω at time t following definitions are made:

$$\xi(t) = x^{'}(0) - x(0) \tag{3.2}$$

The relative position of two particles P_i and $P^{'}$, with coordinates x and x^i, in the reference configuration Ω is defined as ξ(t).

$$\eta(t) = u(x^{'}, t) - u(x, t) \tag{3.3}$$

The relative displacement of two particles P_i and $P^{'}$ is defined as η(t), with $u(x,t) = x(t) - x(0)$. The current relative position is defined as $\eta + \xi$.

With this two definitions the acceleration of a particle P_i in the reference configuration Ω at time t is defined by the integral equation (3.4).

$$\varrho(x)\ddot{y}(x,t) = \int_{\mathscr{H}_i} f((u(x^{'},t) - u(x,t), x^{'} - x) dV_{x'} + b(x,t) \tag{3.4}$$

Equation (3.4) contains at the left hand side the function $\varrho(x)$. This function $\varrho(x)$ returns the mass density ρ of the material at position x in the reference configuration Ω.

At the right side u is the displacement vector field, $b(x,t)$ returns the external force at position x at time t and f is the pairwise force function. The pairwise force function f returns the force vector (per unit volume squared) which particle $P^{'}$ exerts on the particle P_i.
For the horizon $\mathscr{H}_i$ exists, for a given material, a positive number δ such that:

$$|\xi| > \delta \Rightarrow f(\eta,\xi) = 0 \mid \forall \eta \tag{3.5}$$

This means that an spherical neighborhood $\mathscr{H}_i$ of particle P_i in $\mathscr{R}$ exists and there are particles outside the neighborhood, which have no influence on particle P. To assure the conservation of linear momentum $p = m \cdot v$ and angular momentum $L = r \times p = r \times m \cdot v$ the pairwise force function f is required to have the following properties:

$$\forall \eta, \xi \ : \ f(-\eta,-\xi) = f(\eta,\xi) \tag{3.6}$$

Property (3.6) assures the conservation of linear momentum, which means, that the momentum of the closed system is constant, if no external force acts on the closed system.

$$\forall \eta, \xi \ : \ (\eta+\xi) \times f(\eta,\xi) = 0 \tag{3.7}$$

The property (3.7) assures the conversation of angular momentum. In closed systems the angular momentum is constant. The meaning of the equation (3.7) is, that the force vector between particle P_i and $P^{'}$ is parallel to their current relative position vector $\eta + \xi$.
For a micro elastic material the pairwise force function has to be derivable from a scalar micro potential w:

$$\forall \eta, \xi \ : \ f(\eta,\xi) = \frac{\partial w}{\partial \eta}(\eta,\xi) \tag{3.8}$$

The micro potential has the unit of energy per unit volume and holds the energy for a single bond. A bond between a particle P_i and $P^{'}$ exists, if particle $P^{'}$ is in the neighborhood $\mathscr{H}_i$. The local strain energy density per unit volume in the body $\mathscr{R}$ is defined as:

$$W = \frac{1}{2}\int_{\mathscr{H}_i} w(\eta,\xi) dV_{\xi} \tag{3.9}$$

The factor $1/2$ results because each particle of the bond holds half of the energy of the bond.

3.2 Inner forces for the Peridynamic theory

In section 3.1 the theory of Peridynamic is described, but there is no description of the pairwise force function f. The subsection 3.2.1 gives a definition for a pairwise force function f for a Prototype Microelastic Brittle material. In the subsection 3.2.2 is an approach for a pairwise force function for short range forces.

3.2.1 Prototype Microelastic Brittle Model

To model cracks and fractures in a Prototype Microelastic Brittle (PMB) material the assumption, that the pairwise force function for inner forces f depends only on the bond stretch, is made. The bond stretch is defined by:

$$s(\eta, \xi, t) = \frac{\|\eta + \xi\| - \|\xi\|}{\|\xi\|} \tag{3.10}$$

The easiest way to model failure in a model is to let bonds break when they are stretched beyond a predefined constant value. The function $s(\eta, \xi, t)$ (3.10) returns positive values, if the bond is under tension. An isotropic material has the property, that its properties behave the same in all directions. So the bond stretch is independent of the direction of ξ.

The pairwise force function f for a PMB material is defined as

$$f(\eta, \xi, t) = g(\eta, \xi, t) \frac{\eta + \xi}{\|\eta + \xi\|} \tag{3.11}$$

with $g(\eta, \xi, t)$ as a linear scalar valued function, which implements the behavior of the material and the decision if the bond is broken or "alive"

$$g(\eta, \xi, t) = \begin{cases} c \cdot s(\eta, \xi, t) \cdot \mu(\eta, \xi, t), & \|\xi\| \leq \delta \\ 0, & \|\xi\| > \delta. \end{cases} \tag{3.12}$$

with c as the material dependent stiffness constant of the PMB model, $s(\eta, \xi, t)$ as bond stretch (3.10) and $\mu(\eta, \xi, t)$ as history dependent scalar valued function. The function $\mu(\eta, \xi, t)$ is history dependent, because in the PMB model no "healing" of bonds is allowed, and it is defined as:

$$\mu(\eta, \xi, t) = \begin{cases} 1, & \text{if } s(\eta, \xi, t') < s_{00} \;\; \forall\, 0 \leq t' \leq t \\ 0, & \text{otherwise} \end{cases} \tag{3.13}$$

s_{00} as the critical stretch for bond failure for the PMB material. For a PMB material exist only two material constants: the stiffness constant c and the critical stretch for bond failure s_{00}.
Figure 3.2 visualizes the pairwise force function f with $s(\eta, \xi, t)$ as the argument. If the bond between this two particles is not broken, the history dependent scalar valued function returns the constant value 1. After the break the function value jumps to zero and discontinuity exists.

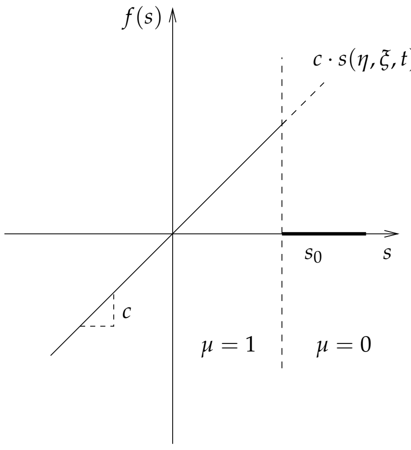


Figure 3.2: Bond force in the PMB model visualized as a function $f(s)$.

In [P+08] an alternative history dependent scalar valued function $\mu(\eta, \xi, t)$ is presented:

$$\mu(\eta, \xi, t) = \begin{cases} 1, & \begin{array}{l} s(\eta, \xi, t') < \min\{s_0(\eta, \xi, t'),\ s_0(\eta', \xi', t')\},\ 0 \leq t' \leq t, \\ s_0(\eta, \xi, t) = s_{00} - \alpha \cdot s_{\min}(\eta, \xi, t),\ s_{\min} = \min_{\eta} s(\eta, \xi, t) \end{array} \\ 0, & \textit{otherwise.} \end{cases} \tag{3.14}$$

With α as a material dependent scaling factor. The stiffness constant c depends on the bulk modulus k. The stiffness constant c is defined in [P+08] as:

$$c = \frac{18 \cdot k}{\pi \cdot \delta^4} \tag{3.15}$$

3.2.2 Short range forces

In the subsection before, particles interact with each other in the horizon $\mathscr{H}_i$, over inner forces, if a bond between this two particles exists. Sometimes it is possible that all bonds of particle P_i in the horizon $\mathscr{H}_i$ are broken. In this case particle P_i is a "free" particle and does not interact with any other particles. In a solid continuum it is not possible that a particle overlaps another particle. To avoid the overlapping of particles a additional chosen pairwise force function f_s can be added to the Peridynamic model:

$$\begin{aligned} f_s(\eta,\xi) &= \frac{\eta+\xi}{\|\eta+\xi\|}\min\left\{0,\frac{c_s}{\delta}(\|\eta+\xi\|-d_s)\right\} \\ \text{with } d_s &= \min\{0,9\|x-x^{'}\|,1,35(r_s+r_s^{'})\} \end{aligned} \tag{3.16}$$

The parameter r_s is defined as the node radius. If the particles lie on a regular grid in the reference configuration Ω the node radius is chosen as half of the lattice constant. The short force ranges are always repulsive and never attractive. According to [P$^+$08] the pairwise force function f_s may also be replaced with a constant potential. A suggestion is the repulsive part of the Lennard-Jones potential, i.e. $\| \eta + \xi \|^{-12}$.

3.3 Discretization

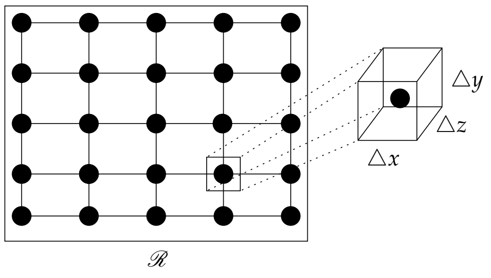


Figure 3.3: Discretization in particles $N = \{n_{i,j} \,|\, 0 < i < n, 0 < j < m\}$ with an surrounding volume $V = \triangle x \cdot \triangle y \cdot \triangle z$ on an equidistant $n \times m$ lattice.

Figure 3.3 shows the discretization of the body $\mathscr{R}$ in N particles on a equidistant $n \times m$ lattice. Each particle has a surrounding volume $V = \triangle x \cdot \triangle y \cdot \triangle z$. In this case a two dimensional lattice is used for the discretization, but it is also possible to use a cubic lattice for the discretization in the three dimensional case. In both cases the surrounding volume of a particle should not intersect with volumes of other particles in the reference configuration Ω.

To discretize the equation of motion (3.4) to get rid of the integral $\int_{\mathscr{H}_i}$ the set $\mathcal{F}_i$ is defined for each particle P_i as:

$$\mathcal{F}_i = \{j \mid \|x_j(0) - x_i(0)\| \leq \delta,\ j \neq i\} \tag{3.17}$$

The set $\mathcal{F}_i$ contains the indices of all particles, which are in the horizon $\mathscr{H}_i$ of particle P_i.

$$(\varrho(x_i)V_i)\ddot{y}_i^t = \sum_{j\in\mathcal{F}_i} f(\underbrace{u(x_j,t)-u(x_i,t)}_{\eta},\underbrace{x_j-x_i}_{\xi})\tilde{V}_jV_i + b(x_i,t)V_i \tag{3.18}$$

The equation (3.18) is the discrete Peridynamic equation of motion. With $\varrho(x_i)$ as the mass density function, V_i as the surrounding volume V of particle P_i, f as the pairwise force function and $b(x_i,t)$ as the function for the extern force at position x at time t at the body $\mathscr{R}$.
The volume $\tilde{V}_j$ is the scaled volume $\nu(x-x^{'})\cdot V_j$ of particle P_j with the linear dimensionless scaling function:

$$\nu(x-x^{'}) = \begin{cases} -\frac{1}{2r_s}\|x_j-x_i\| + (\frac{\delta}{2r_s}+\frac{1}{2}), & \delta - r_s \le \|x_j-x_i\| \le \delta \\ 1, & \|x_j-x_i\| \le \delta - r_s \\ 0, & otherwise. \end{cases} \tag{3.19}$$

The function (3.19) is needed, for particles, which are bond to particle P_i but very close to the horizon $\mathscr{H}_i$. A part of the volume of this particle is outside the sphere around particle P_i. The influence of this particle is not the same, as of a particle, which is with the volume totally inside the sphere of the horizon $\mathscr{H}_i$. If the distance $\| x_j - x_i \| = \delta$ the volume V_j is scaled with the factor 0.5, because nearly one half of the volume is inside and one half of the volume is outside the horizon $\mathscr{H}_i$.

Chapter 3.2.2 introduced the additional optional pairwise force function f_s (3.17). To add the pairwise force function f_s to the discrete Peridynamic equation of motion the following set $\mathcal{F}_i^S$ is defined:

$$\mathcal{F}_i^S = \{j \,|\, \|x_j(t) - x_i(t)\| \le d_s,\ j \ne i\} \tag{3.20}$$

The difference to the set $\mathcal{F}_i$ is that the set $\mathcal{F}_i^S$ uses the actual positions of the particle P_i and P_j at time t. So the set $\mathcal{F}_i^S$ may change every time step. The use of the short range forces depends on the problem and should be chosen wisely. The equation (3.21) is the discrete Peridynamic equation of motion with support of short range forces.

$$\begin{aligned}(\varrho(x_i)V_i)\ddot{y}_i^t &= \sum_{j\in\mathcal{F}_i} f(\underbrace{u(x_j,t)-u(x_i,t)}_{\eta},\underbrace{x_j-x_i}_{\xi})\tilde{V}_jV_i \\ &+ \sum_{j\in\mathcal{F}_i^S} f(\underbrace{u(x_j,t)-u(x_i,t)}_{\eta},\underbrace{x_j-x_i}_{\xi})\tilde{V}_jV_i + b(x,t)V_i\end{aligned} \tag{3.21}$$

3.3.1 Computation of the position at time t

The discrete Peridynamic equation of motion (3.18) returns the acceleration $\ddot{y}_i^t$ of a particle i at the time step t. In most cases of simulation results the position x_i^t of a particle i at the time step t is interesting. To estimate the position x_i^t the Störmer–Verlet method [HLW03] is used. From a historical view this method is interesting, because the description of this method was first given by Isaac Newton's Principia in 1687. To estimate the position x_i^t the two–step formulation is used:

$$x_i^{t+1} = 2 \cdot x_i^t - x_i^{t-1} + \ddot{y}_i^t \cdot \triangle t^2 \tag{3.22}$$

In the first step are $x_i^{t-1} = x_i^t$ and $\ddot{y}_i^t = 0$. The Störmer–Verlet method (3.22) has the error term of $\mathcal{O}(\triangle t^4)$.

4 Implementation

In section 4.1 the Compute Unified Device Architecture (CUDA), developed from NVIDIA, is introduced. Section 4.2 shows the program flow chart (figure 4.4) and describes the used algorithms during the program flow.

4.1 Compute Unified Device Architecture

NVIDIA introduced with the Compute Unified Device Architecture (CUDA) a parallel computing architecture for Graphics processing units (GPU). With CUDA[1] as computing engine for NVIDIA GPUs, the access for developers through standard programming languages is provided.
The computing engine is accessible with the programming language "`C` for CUDA". This programming language is similar to the programming language C, but has some special keywords and certain restrictions [cud11b]. The source files are compiled with a Path Scale Open64[2] C compiler (nvcc) for the execution on the GPU. The utils for developers are provided for Microsoft Windows, Linux and Mac OS X.

Figure 4.1 shows the processing flow on CUDA. At the beginning memory for the data structures has to be allocated on the host and filled with data. Step ① copy the data from the host to the device. For this step the memory on the device has to be allocated from the host side. In step ② the CPU instructs the GPU with the processing of the device code. In step ③ the device code is executed parallel on each core. To use the result of the computation on the host side, the data is copied on back to the host ④.

To describe the processing flow of CUDA it is enough to talk from one kind memory. To understand the Compute Unified Device Architecture it is necessary to introduce more kinds of memory: *global memory*, *constant memory*, *texture memory*, *shared memory* and *per-thread local memory*. To differentiate between the different kinds of memory we need to talk about: *threads*, *blocks* and *grids*.

[1]CUDA is a unregistered trade mark of NVIDIA
[2]An open source compiler with optimization for Itanium and x86-64 microprocessor architectures.

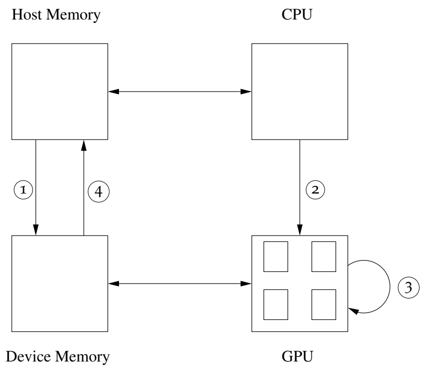


Figure 4.1: Processing flow on CUDA

The figure 4.2 shows the layers of CUDA. The atomic unit in CUDA is a single *thread*. Each thread has its own *per-thread local memory*. Not visualized in this figure are the registers of the threads. In the next layer *blocks* of *threads* are defined by the developer.

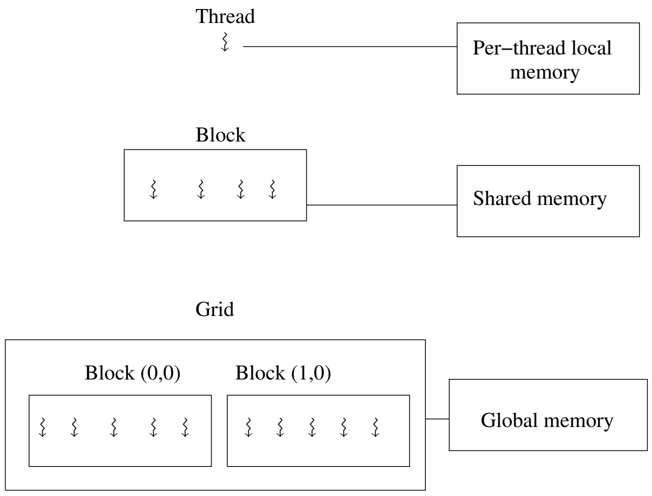


Figure 4.2: Layers of CUDA

Therefore the attribute *blockSize* as an extension of the "C for CUDA" programming language exists. All *threads* in a *block* are executed together with the Single Instruction Multiple Threads (SIMT) architecture. The *blocks* are distributed to different scalar processors within the same multiprocessor (MP). All *threads* in one *block* have access to the same *shared memory*.
In the last layer several blocks are combined as a *grid*. To define the amount of *blocks*

in all grids the attribute *gridSize* exists. One *grid* of *blocks* is executed independently in serial or parallel and is distributed to different multiprocessors. All grids share the same *global memory*.

The attributes *blockSize* and *gridSize* are organized in one, two or three dimensions. The *blockSize* and *gridSize* are defined before the execution of the kernel and so the *blockSize* and *gridSize* are the same for each multiprocessor in this execution. The size of each kind of memory is limited by the specification of the NVIDIA device. To get the values the NVIDIA CUDA LIBRARY provides the struct *cudaDeviceProp*. All significant attributes are listed in the CUDA API REFERENCE MANUAL [cud11a].

Lastly there are *constant memory* and *texture memory*, which not exist as physical memory on the device. These kinds of memory reside in the *global memory* and their maximal size is restricted by the device attributes. The *constant memory* is read-only and not dynamically allocable. This kind of memory can be accessed directly from the device. From the host side it has to be copied as an symbol and is only indirectly accessible. The *texture memory* is also read-only, but it is possible to bind data from the host to a texture. Applications for this kind of memory is heavily read-only data with irregular access patterns. A disadvantage of the *texture memory* is that no double precision support is provided. NVIDIA devices with support of CUDA $\geq$ 3.1 provide additional *surface memory*, which allows read and write access.

The figure 4.3 shows an overview of the different layers: *threads*, *blocks* and *grids* and the different kinds of memory: *global memory*, *constant memory*, *texture memory*, *shared memory* and *per-thread local memory*, which are accessible from the different layers.

An important extension of the "C for CUDA" programming language are the function type qualifiers. The function type qualifiers specify where the function is executed and where the function is callable. For the execution on the device the qualifiers *__global__* and *__device__* exist.

A kernel is the entry point to start an execution on the CUDA device. A kernel is the only possibility to start an execution on the device and it is callable only from the host. A CUDA kernel has to be declared with the function type qualifier *__global__*. A restriction for the return type is, that a kernel allows no return type and must have a **void** return type. A kernel call is asynchronous and returns before the device has completed the execution.

To provide functions with a non *void* return type on the device, the function type qualifier *__device__* is provided.

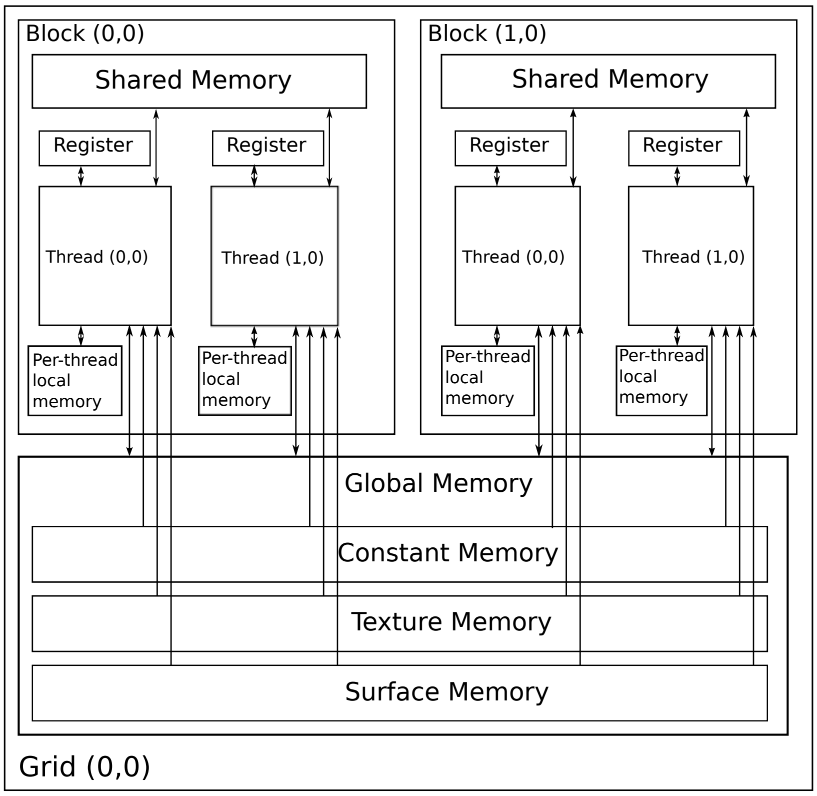


Figure 4.3: Memory access model of CUDA

The function is executed on the device and only callable from a kernel or a other device function. To call a function from the host the function type qualifier __*host*__ is provided.

4.2 Program flow

Figure 4.4 shows the program flow chart of the simulation. The state "Update positions" is executed on the CUDA device. The other states are executed on the CPU. In the first state the particles in the reference configuration Ω are generated. After the generation of the particles in the reference configuration Ω the set $\mathcal{F}_i$ (3.17) for all particles are engendered. The state "Search the neighborhood of each particle" is divided in two sub states. These sub states are described in subsection 4.2.1. In the last state the main

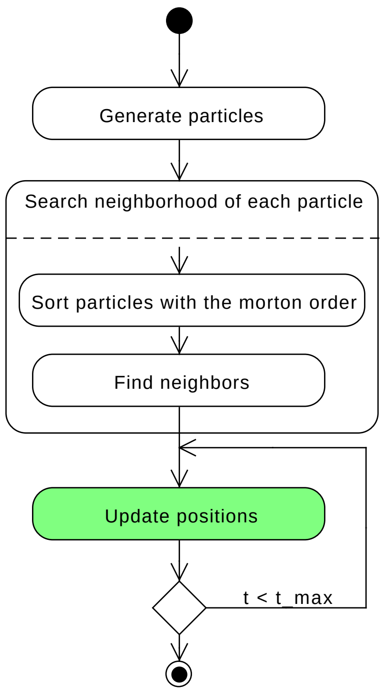

Generate particles
Search neighborhood of each particle
Sort particles with the morton order
Find neighbors
Update positions
t < t_max

Figure 4.4: Program flow chart of the simulation. The green state of the diagram is executed on the CUDA device.

part of the Peridynamic is implemented. These states are executed on the NVIDIA device in parallel with CUDA. To boost up the computation time, the other states of the simulation should be ported to CUDA, but this is not part of this thesis.

4.2.1 Neighborhood search

For the discrete Peridynamic equation of motion the set $\mathcal{F}_i$ (3.17) has to be computed for all particles. The naïve algorithm constructs the sets $\mathcal{F}_i$ in $\mathcal{O}(n^2)$. A simple optimization for a equidistant lattice is to stop if the next length to the next particle is $\geq \delta$. In this special case the algorithm is in $\mathcal{O}(n)$.
To use a non equidistant lattice for the reference configuration Ω of the particles another algorithm is needed. In [Mic09] Connor and Kumar introduced an algorithm for the "construction of k-Nearest Neighbor Graphs for Point Clouds" in $\mathcal{O}(\lceil \frac{n}{p} \rceil \cdot k \cdot log(k))$. The advantage of this algorithm is that randomly generated point

clouds are supported. This means there is no restriction of the order of the particles in the reference configuration Ω.

The algorithm [Mic09] uses the merge sort [Knu73] in combination with the morton order [TH81] to prepare the point cloud for an efficient search of the neighbors. The morton order or z order is a method to map multilevel data to one dimension. This method originally invented for databases and the approach is the concatenation of keys to generate a one dimensional code.

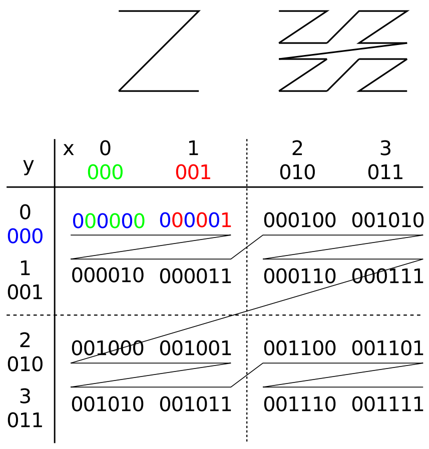


Figure 4.5: This figure illustrates the space filling curve for the morton order / z order in the upper part. The lower part shows an example for the z values for $0 \leq x, y \leq 3$.

The figure 4.5 illustrates the space filling curve for the z order in the upper part. In the upper part the one dimensional code (z values) for the two dimensional point cloud $0 \leq x, y \leq 3$ is computed. The curve between the z values is named z curve.
The library STANN [STA] is the `C++` implementation of the algorithm "Fast construction of k-Nearest Neighbor Graphs for Point Clouds" described in [Mic09]. This library is used for the neighbor search in the simulation. To verify the scaling of the run time of the STANN library the run time with a equidistant lattice was measured and plotted in figure 4.6.

4.2.2 Update Positions

The last state of the program flow chart (4.4) is implemented with the "`C` for CUDA" programming language. In the first part of this section the CUDA kernel is described.

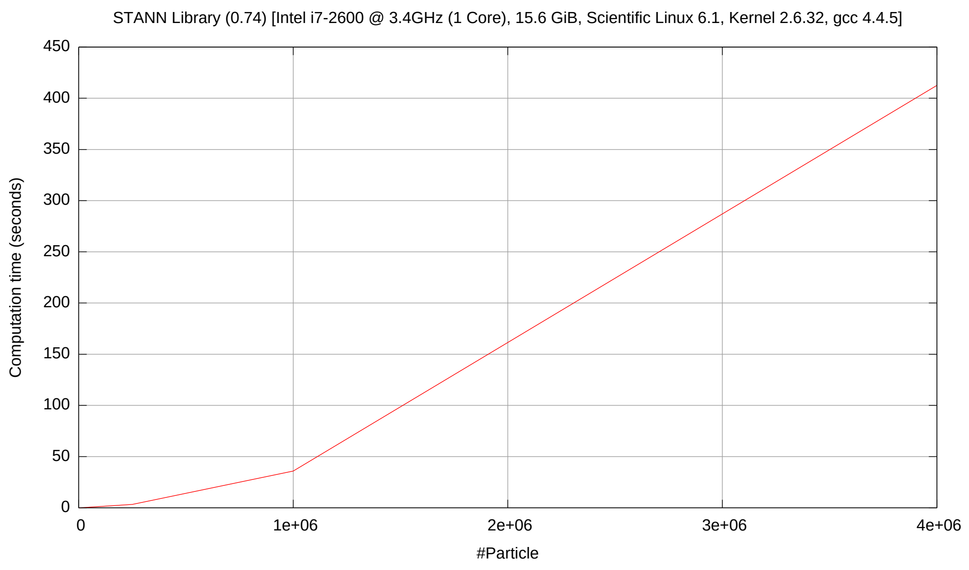


Figure 4.6: Run time of the STANN library on a equidistant two dimensional lattice.

The part of the code of the "C for CUDA" programming language, which is executed on the CUDA device, is called a CUDA kernel.
The second part of this section contains the restrictions for the theoretical maximal amount of particles. The theoretical maximal amount of particles depends on the specification of the CUDA device. An equation (4.2) for the theoretical maximal amount and some examples for the *NVIDIA GeForce GTX* 560 *Ti*, the used CUDA device in this thesis, are provided.

4.2.2.1 Parallelization of the discrete Peridynamic equation of motion

The implementation of the discrete Peridynamic equation of motion (3.18) with the "C for CUDA" programming language is shown in the listing 4.1. The kernel contains four steps, but the last one is optional for improved visualization of the simulation results.
The step $\textcircled{0}$ initialize the vectors for the inner force f_i in the first time step. In all other time steps the vectors have to need to be reset, because the inner force f_i is not history dependent. In $\textcircled{1}$ the discrete Peridynamic equation of motion (3.18) is computed for each particle. So the information about the acceleration $\ddot{y}_i^t$ on each particle P_i at time t is there. To compute the position of each particle P_i at time t the Störmer–Verlet method (3.22) is used in $\textcircled{2}$.

The step $\textcircled{3}$ contains not really to the Peridynamic, but is needed to produce the figures in section 6.2. To measure the computation time of the CUDA kernel the computation time with step $\textcircled{3}$ and without step $\textcircled{3}$ is behold.

4.2.2.2 Maximal amount of particles

The figure 4.7 shows the data structures on the CUDA device. The lower data structure has the size of $n \cdot k$, with n as the amount of particles and k as the maximal amount of neighbors.

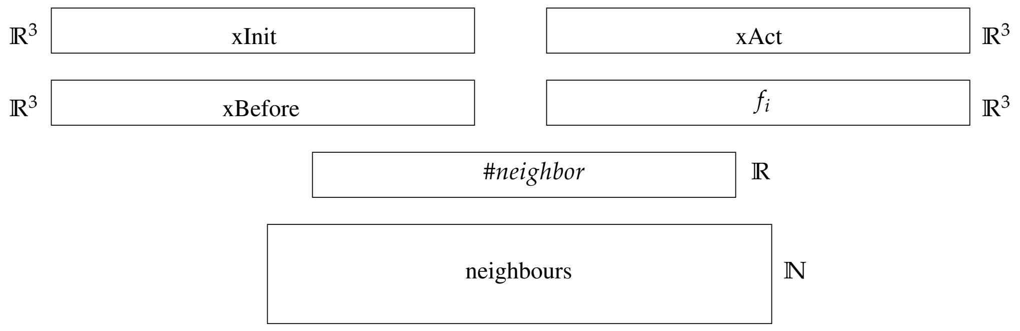


Figure 4.7: This figure shows the data structures on the CUDA device. The upper data structures are linear in the size of n. The lower data structure has the size of $n \cdot k$, with n as the amount of particles and k as the maximal amount of neighbors.

The equation 4.1 shows the usage of memory on the CUDA device. The number n is the amount of particles and k the maximal number of neighbors per particle.

$$[4 \cdot 3 \cdot sizeof(double) + sizeof(int) + (k+1) \cdot sizeof(int)] \cdot n \tag{4.1}$$

The size of the global memory on a device restricts the maximal number of particles per computation. In `C++` and in "`C` for CUDA" the size of a double is 8 bytes and the size of an integer is 4 Bytes. We assume that N is the size of the global memory on our CUDA device. Equation 4.2 defines the maximal amount of particle on the CUDA device.

$$n = \left\lfloor \frac{N}{4 \cdot 3 \cdot 8 + 4 + (k+1) \cdot 4} \right\rfloor = \left\lfloor \frac{N}{104 + 4 \cdot k} \right\rfloor \tag{4.2}$$

The *NVIDIA GeForce GTX* 560 *Ti*, the used CUDA device in this thesis, provides 2.146.631.680 bytes as global memory.

Listing 4.1 The parallel computation of the equation of motion and the estimation of the new positions on the CUDA device

```
__global__ UpdatePositions( ... )
{
        for all time steps
        {

                // ⓪ Initialize the inner forces

                for all particles
                {
                  $f_{i\,index} = \{0,0,0\};$
                }
                __syncthreads();

                // ① Compute the acceleration for each particle
                for all particles
                {
                  $\ddot{y}_i^n = \frac{\sum_{j\in\mathcal{F}_i} f(u_j^n - u_i^n, x_j - x_i)\tilde{V}_j V_i}{p(x_i)V_i}$
                }
                __syncthreads();

                // ② Compute the new position of each particle
                for all particles
                {
                  $x_i^{n+1} = 2 \cdot x_i^n - x_i^{n-1} + \ddot{y}_i^n \cdot \triangle t^2$
                }
                __syncthreads();

                // ③ Compute the updated amount of neighbors for each particle
                for all particles
                {
                        for all neighbors
                        {
                          //Update the amount of neighbors for each particle
                        }

                }
                __syncthreads();
        }
}
```

4.3 Measurement of the computation time

There exist several implementations of the Peridynamic technique [Sana, PSP+11, SIE11], but none of them has no native support of CUDA. All of them are implemented in FORTRAN or C++ and executed on single core CPU or multi core CPU's.
Another possibility is to use a GPU to implement the Peridynamic technique. An important attribute for a simulation is the computation time, especially for large entities. So the computation time for following two implementations of the Peridynamic technique are compared:

1. CUDA (NVIDIA GeForce GTX 560 Ti)
   - Host: Scientific Linux (6.1) with Kernel 2.6.32.1
   - gcc 4.4.5 (-*O*3 -*m*64)
   - nvcc 4.0 *V*.2.1221 (-*m*64 -*arch*=*sm*_21 -*use_fast_math*)
2. Single core (Intel i7-2600 @ 3.4GHz)
   - Host: Scientific Linux (6.1) with Kernel 2.6.32.1 and 16 GiB RAM
   - gcc 4.4.5 $-O3$

To compare different implementations or algorithms a measurement for the comparative property is needed:

> "A software metric is any measurement which relates to a software system, process or related documentation." [Iva89]

The software engineering provides software metrics for objective, reproducible and quantifiable measurements . Software metrics are classified in two classes: control metrics and predictor metrics. Predictor metrics are measurements to compare the quality of to software products. To compare the two implementations the following metric (4.3) is defined:

$$m_1 = \frac{\text{computation time}}{\#\text{particles}} \tag{4.3}$$

Figure 4.8 shows the computation time of the Peridynamic technique for on particle per time step. The first aspect is that the computation time on the GPU is articulately faster per particle and the difference is medial 3.0.
This value is feasible, because the processing power for *NVIDIA GeForce GTX 560 Ti* [Har] in double precision is 105 GFLOPS and the processing power for the *Intel i*7-2600 @ 3.4*GHz* [Mar] is 83 GFLOPS. For a single thread the processing power for the *Intel*

*i*7-2600 @ 3.4*GHz* sinks to ≈ 21 GFLOPS. The theoretical value for the difference in the computation time between the GPU and CPU is $\frac{105\,\text{GFLOPS}}{21\,\text{GFLOPS}} = 5$.

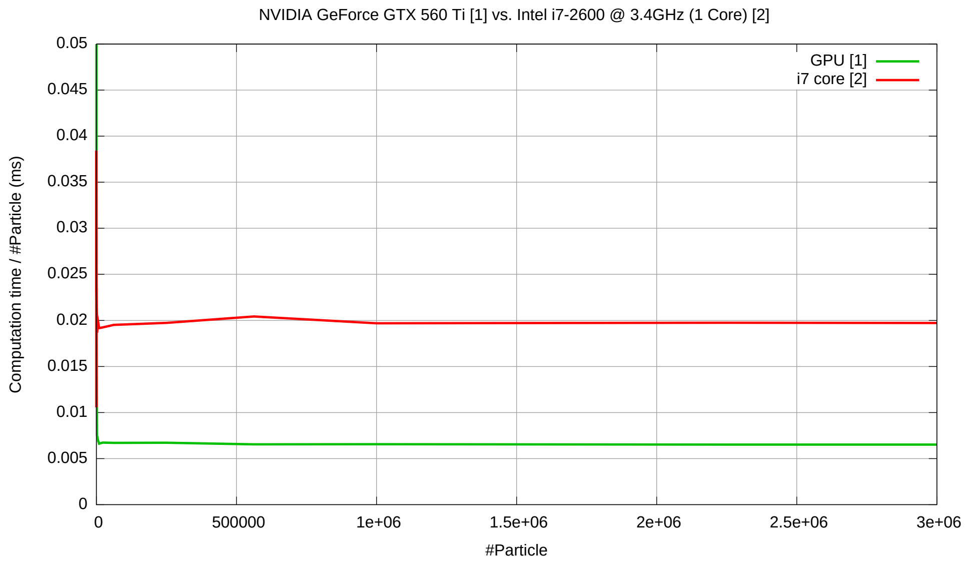


Figure 4.8: Run time of the different implementations for one particle per time step.

Figure 4.9 shows the run time of the CUDA kernel (4.1) with step $\textcircled{3}$ and without $\textcircled{3}$. The step $\textcircled{3}$ is not part of the Peridynamic, but delivers additional information for the visualization.

4.4 Challenges with the Compute Unified Device Architecture

This section contains a summary of challenges with the Compute Unified Device Architecture. The test [Har] of the processing power for *NVIDIA GeForce GTX* 560 *Ti* results 1.263 GFLOPS for single precision. In the Peridynamic theory small time steps in the range of $10^{-9} - 10^{-6}$ are the standard for most simulations. This causes small changes in the values during the simulation and it is necessary to use double recession. For double precision the processing power *NVIDIA GeForce GTX 560 Ti* sinks to 105 GFLOPS. Compared to the test [Mar] of the CPU *Intel Core i*7 2600*K* 3.40 *GHz* with 83 GFLOPS the benefit of the GPU shrinks.

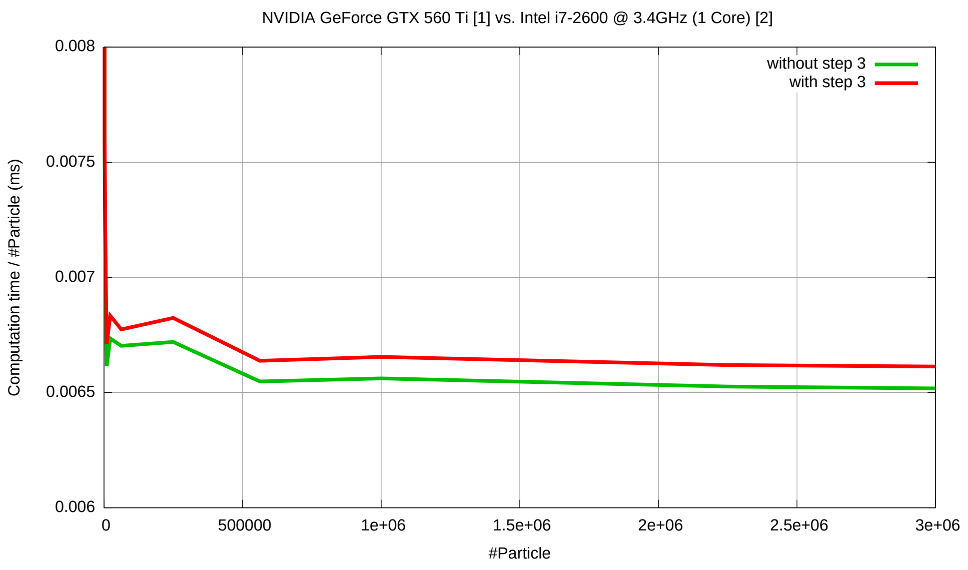


Figure 4.9: Run time with and without the update of neighbors step.

In section 4.1 the levels of CUDA were introduced. All threads in a block are executed SIMT. Therefore all threads in a block are grouped to groups of 32 threads. These groups of threads are called *wraps*. Some operations are per *half-warp* and either the first or second 16 threads of the wrap are executed. Control flow divergence is possible, but all other threads of the wrap have to wait, because of the SIMT, until the other threads finished the execution. The developer should try to avoid complex conditional branches for one or only few threads in a wrap. The ideal number of threads in a block is a multiple of 32. The number of registers per multiprocessor is also limited: Register per thread $\cdot$ threads per block $\leq$ registers per block. For the *NVIDIA GeForce GTX* 560 *Ti* 32.768 registers per block are specified.

The global memory resides in the device memory. The device memory is accessed via 32−, 64−, or 128−byte memory transactions. So only 32−, 64−, or 128−byte segments can be read or written by memory transactions. The instruction throughput depends on the amount of memory transactions. The more transactions are necessary, the more unused memory is transferred. Padding unused data and align the size of the data type to 1, 2, 4, 8, or 16 bytes can solve this problem.

Figure 4.10 shows the global memory and the cache line. Data is transferred to/from L1/L2 cache in blocks of 128 bytes. To one cache line belongs 128 bytes in the global

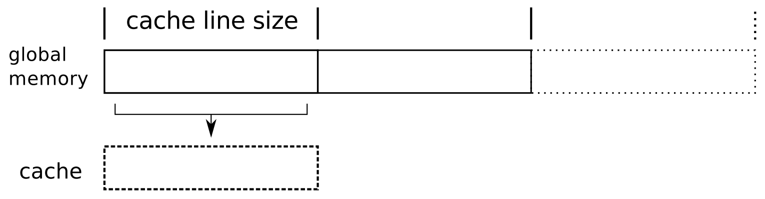


Figure 4.10: Global memory and caching

memory. To improve the performance the access to the global memory should be coalesced. If multiple threads of a wrap access the same cache line the accesses are combined into one memory transfer.

The last advice for improvement of performance is to use fast mathematical operations, e. g *__sprt*(. . .), if it is possible. These functions are less accurate, but executed faster. The compile flags *-use_fast_math* has been set to get correct results of the fast mathematical operations.

5 Combination with the Partition of Unity Method

The Partition of Unity Method (PUM) is introduced in [MB96, pum97]. In [Sch03] the abstract ingredients are:

- A Partition of the unity $\{\varphi_i \mid i = 1,\ldots,N\}$ with $\varphi_i \in \mathcal{C}^r(\mathbb{R}^D, \mathbb{R})$ and patches $\omega_i := supp^\circ(\varphi_i)$,
- a collection of local approximation spaces

  $$V_i(\omega_i, \mathbb{R}) := \text{span}\langle \vartheta_i^n \rangle \tag{5.1}$$

  defined on the patches ω_i , $i = 1,\ldots,N$.

The combination of these two ingredients is the following:

$$V^{PU} := \sum_{i=1}^{N} \varphi_i V_i = \text{span}\langle \varphi_i \vartheta_i^n \rangle \tag{5.2}$$

A property of the PUM is that the approximation spaces V_i can be chosen independent of each other. So it is possible to combine the Peridynamic theory with a Finite Element Method (FEM), e. g. PUM. The figure 5.1 shows an approach to combine these two methods. The body $\mathscr{R}$ in the reference configuration Ω is divided in two areas Ω_{PUM} and $\Omega_{Peridynamic}$. The area $\Omega_{Peridynamic}$ in the reference configuration Ω is placed at the position with an initial crack or at the place where a fracture is presumably. In the figure 5.1 there is a initial crack with the color magenta. The remaining area in the reference configuration Ω is simulated with the PUM.
First several time steps with the PUM are simulated and then the boundary force f_{bound} is exchanged over the boundary of the two areas Ω_{PUM} and $\Omega_{Peridynamic}$. The exchange of the boundary force f_{bound} is done with the discrete Peridynamic equation of motion (3.18, 3.21). The discrete Peridynamic equation of motion has the additive term $b(x_i, t)$ on the right side. This term is used to model extern forces on particle at position x at time t.

Secondly several time steps of the Peridynamic are simulated. After the simulation the displacement field $u = \{X_i \mid i = 1,\ldots,N\}$ with X_i as the position of all

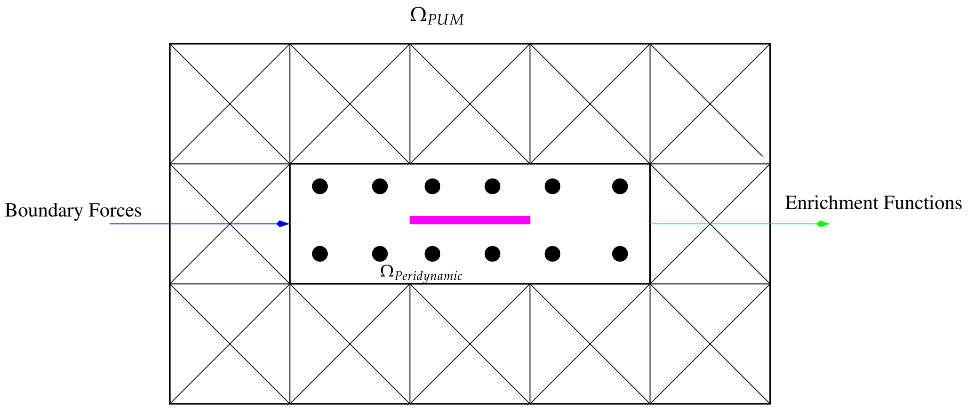


Figure 5.1: Combination with the Partition of Unity Method

particles at time step i exists. With the displacement field u it is possible to generate enrichment functions, which can be used to simulate the next time steps of the PUM.

The generation of the enrichment functions is not part of this thesis. More information about their generation of the enrichment functions can be found in the special research field 716, sub project $D.7$ [Col12].

6 Results

In this Chapter are two different possibilities to use the Peridynamic model on solid mechanics. In the first experiment an thick square plate with an initial crack were used for the geometry. In this scenario the growing of the crack is interesting. The second experiment simulates the impact of an projectile into an cylinder. In this case the interest is in the appearance of cracks.

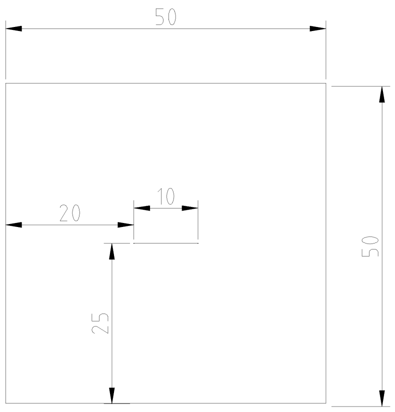
6.1 Experiment 1

This experiment was inspired from the publication "A mesh-free method based on the Peridynamic model of solid mechanics" [SA05]. The experiment in the publication uses velocity boundary conditions. Velocity boundary conditions are not available in the Peridynamic technique introduced in [Sil00]. In [Sil98] in section 13 and section 14 a formulation of the Peridynamic technique with boundary conditions is presented. The figure 6.1 shows the geometry of the thick square plate. In the center of the slim square plate in an horizontal initial crack with the length of 10 mm.

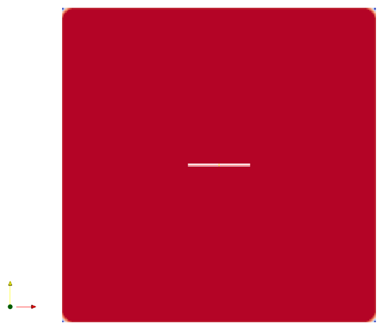
In this experiment the thick square plate receives an first pulse, which generates tensile stress waves. This waves move towards the crack and start the growing of the crack. The second pulse is an compressive pulse and stops the growing of the crack. The figure 6.2 shows the forces on the thick square plate. The forces are applied on each half of the thick square plate.

The table 6.1 contains the simulation parameters for this experiment.

Figure 6.3 shows the growing of the crack and the waves moving to the crack in the first 5 time steps. Figure 6.3 shows the compressive pulse and the delay of the crack in the last 5 time steps. This experiment is not the same as in the publication, because velocity boundary conditions are not available in the Peridynamic technique used in this thesis. This result shows that it is possible to simulate growing and delay of cracks with the Peridynamic technique described in [Sil00].



(a) Blueprint of the thick square plate.



(b) Reference configuration Ω of the body $\mathscr{R}$. Particles are colored by the amount of neighbors.

Figure 6.1: Blueprint and reference configuration Ω of the body $\mathscr{R}$ (experiment 1).

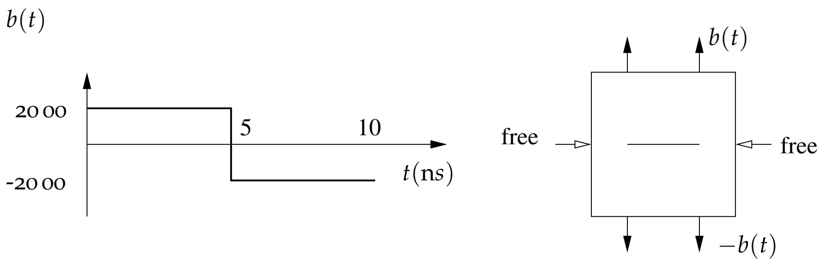
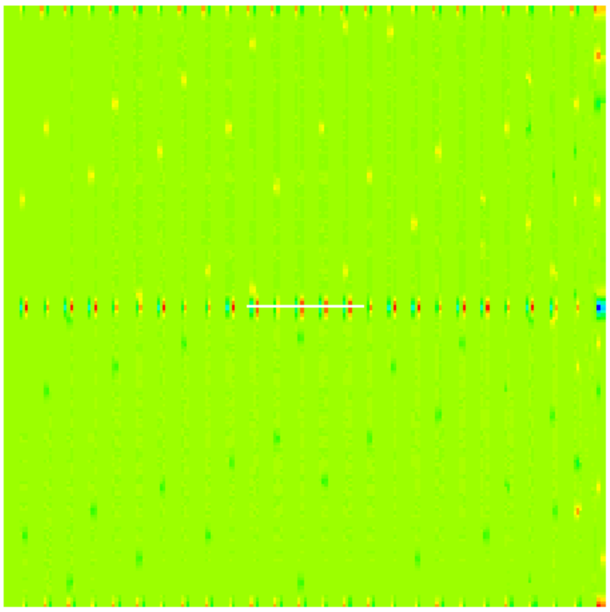


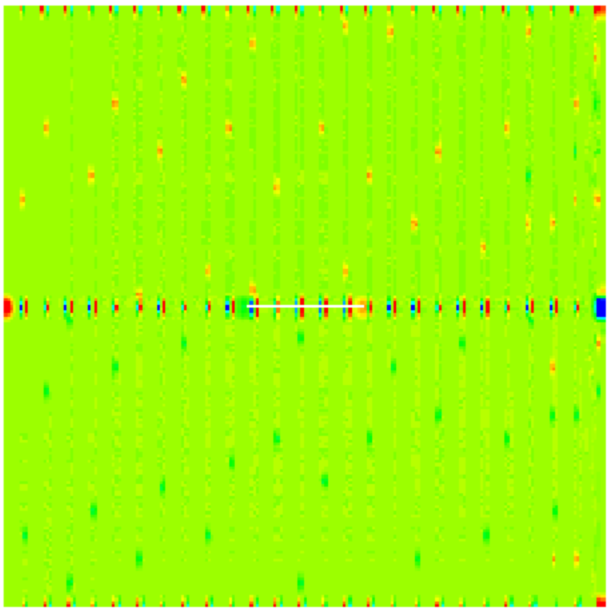
Figure 6.2: Forces on the horizontal edges on the thick square plate.

Parameter	Value
Density ϱ	$8000\ \frac{\mathrm{kg}}{\mathrm{m}^2}$
Young's modulus k	192 Pa
Length of horizon δ	1.6 mm
Critical stretch for bound failure s_{00}	0.02
Δx	0.25 mm
Δy	0.25 mm
Δz	0.25 mm
#Particles	39920
time step	1µs
step size	10

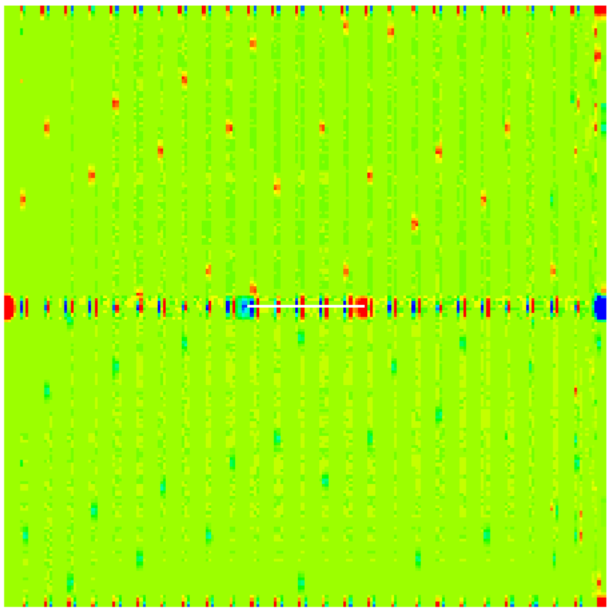
Table 6.1: Simulation parameters for experiment 1



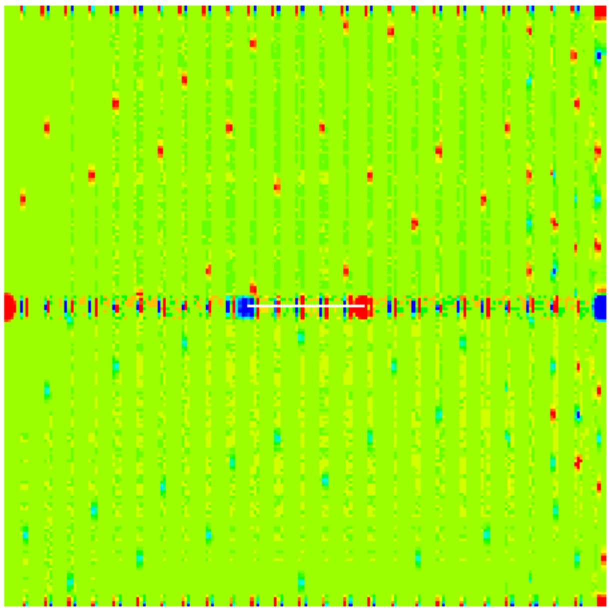
(a) 1µs



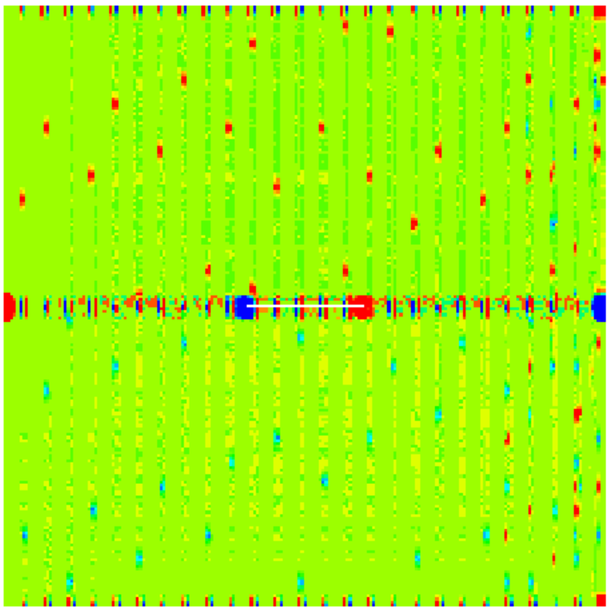
(b) 2µs

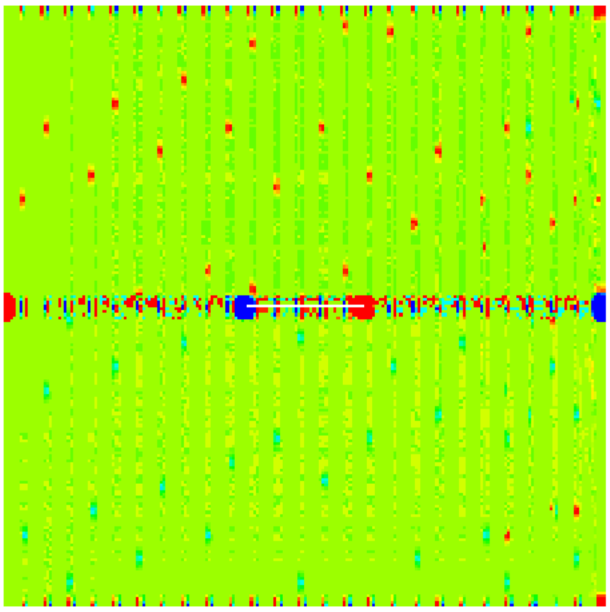


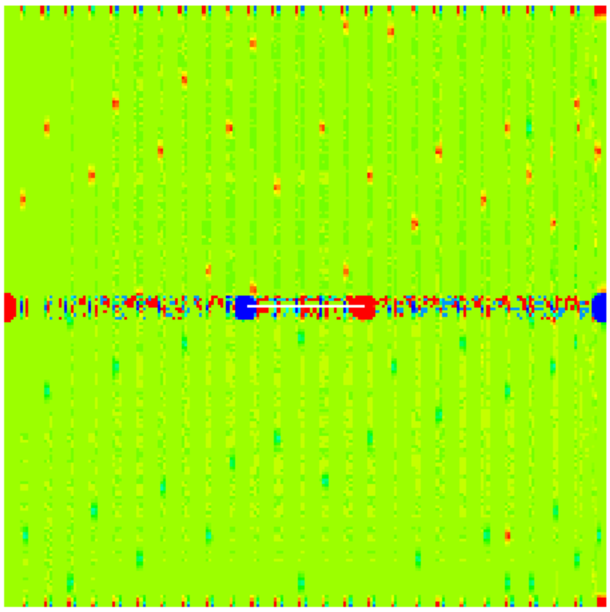
(c) 3µs



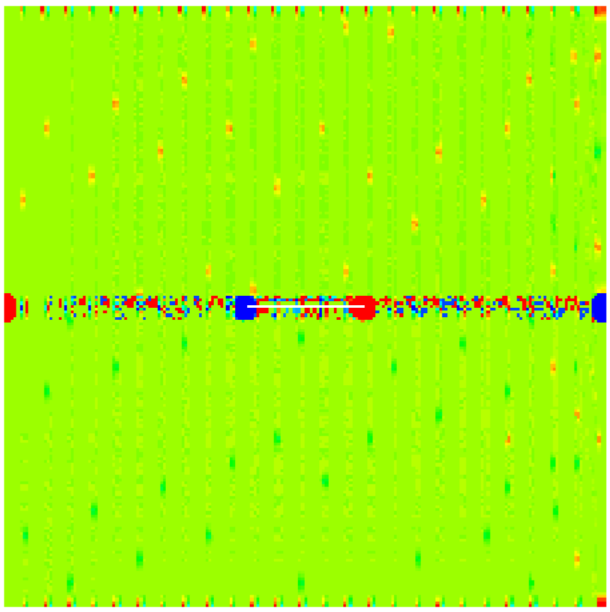
(d) 4µs


(e) 5µs

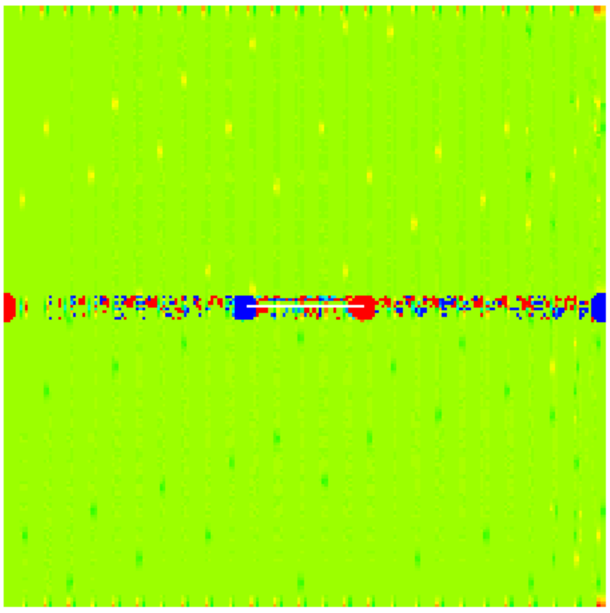

(f) 6µs



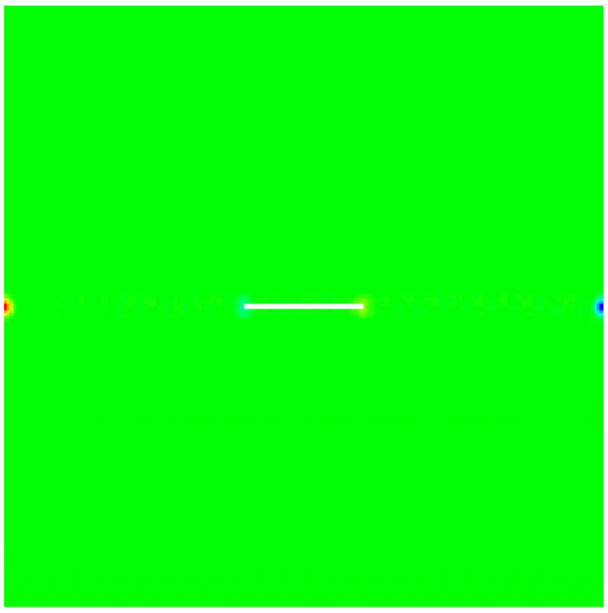
(g) 7µs



(h) 8µs



(i) 9µs



(j) 10µs

Figure 6.3: Growing of the crack (1µs–5µs) and delay of the crack (6µs–10µs). Particles are colored by the acceleration in x–direction.

6.2 Experiment 2

The Experiment 2 was adapted from the publication "Implementing Peridynamic within a molecular dynamics code" [P+08]. The figure 6.4 shows the geometry of the cylinder. In this experiment the impact of an projectile was simulated. The projectile is modeled as an rigid sphere with the diameter 0.01m and an indenter, exerting force $F(r) = -1 \cdot 10^{17}(r-R)^2$. The scenario of the impact is shown in figure 6.5.

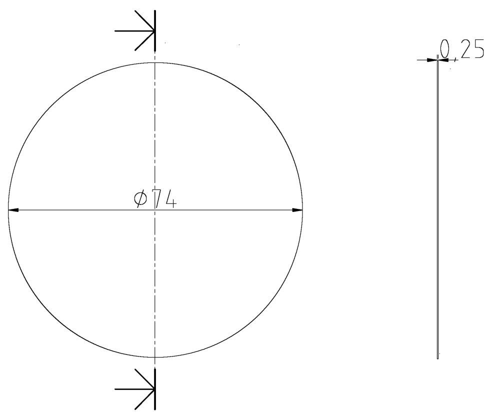


Figure 6.4: Blueprint of the cylinder.

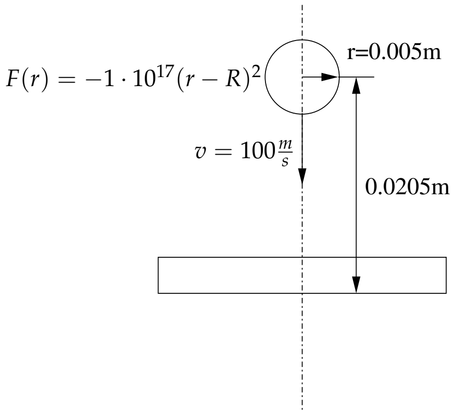
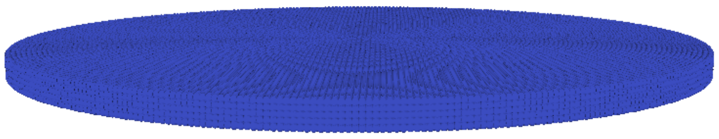


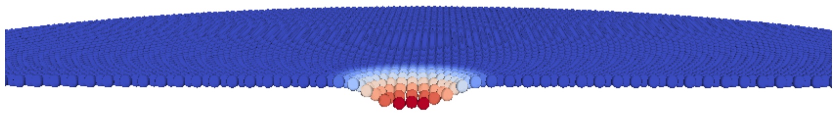
Figure 6.5: Model, measurements and forces of the projectile.

Parameter	Value
Density ϱ	$2200 \ \frac{\mathrm{kg}}{\mathrm{m}^2}$
Young's modulus k	14.9 GPa
Length of horizon δ	0.0005 m
Critical stretch for bound failure s_{00}	0.0005
Δx	0.0005 m
Δy	0.0005 m
Δz	0.0005 m
#Particles	83250
time step	1ns
step size	100000

Table 6.2: Simulation parameters for experiment 2

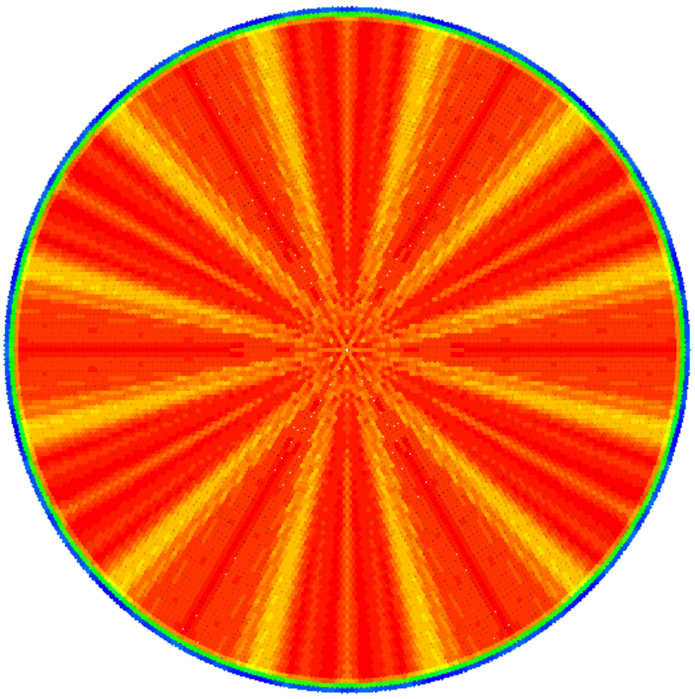


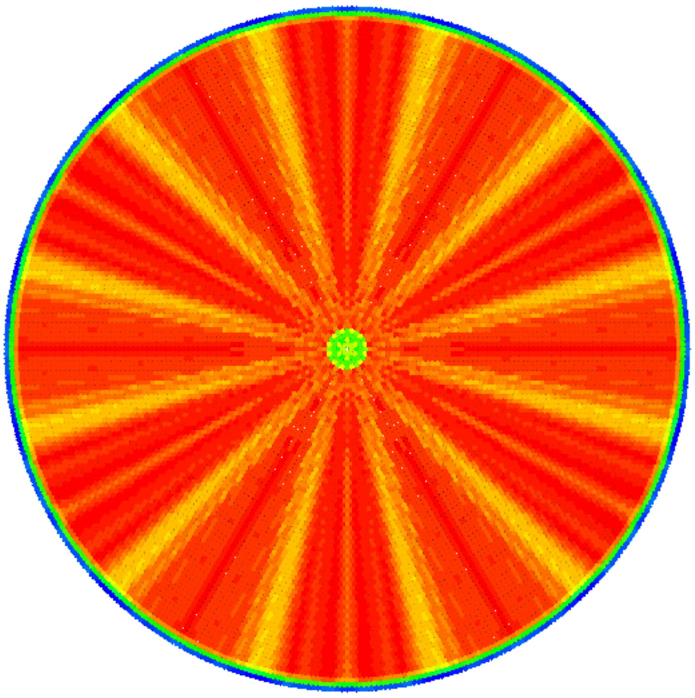
(a) The reference configuration of the body $\mathscr{R}$.

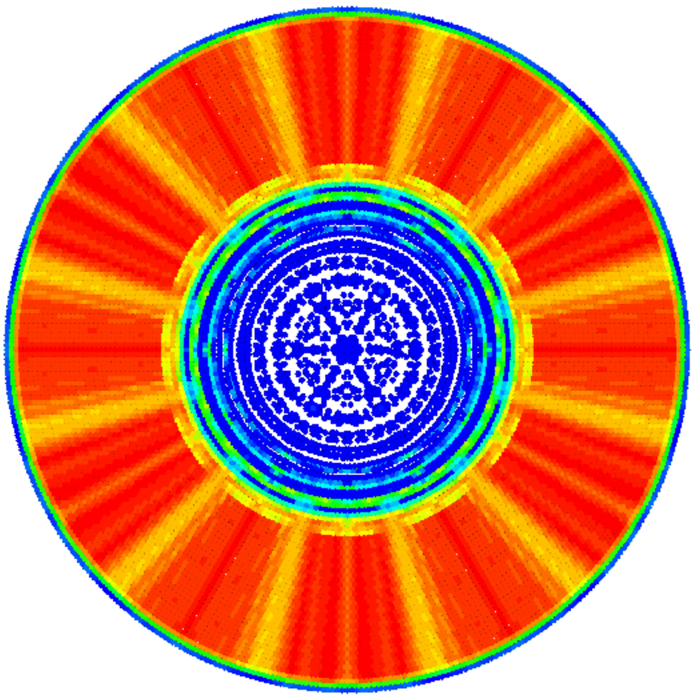


(b) The impact of the projectile into the top layer of body $\mathscr{R}$. Particles are colored with the amount of neighbors.

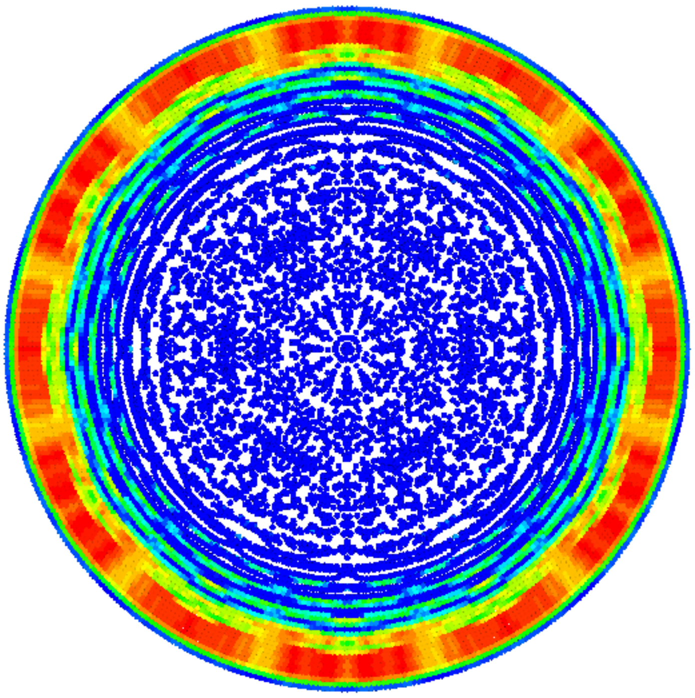
Figure 6.6: This series of picture shows the body $\mathscr{R}$ in the reference configuration Ω and the cross section of the top layer $\mathscr{R}$ during the impact of the projectile.


(a) 10ns

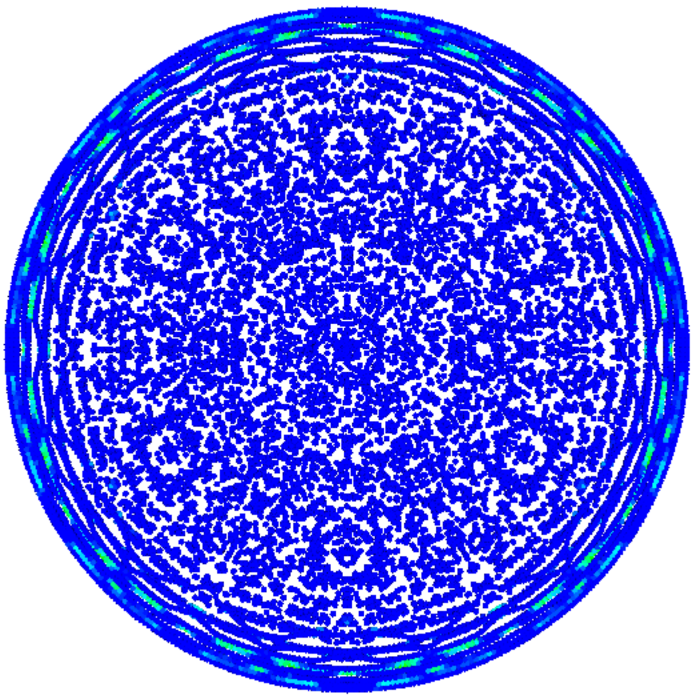

(b) 50ns



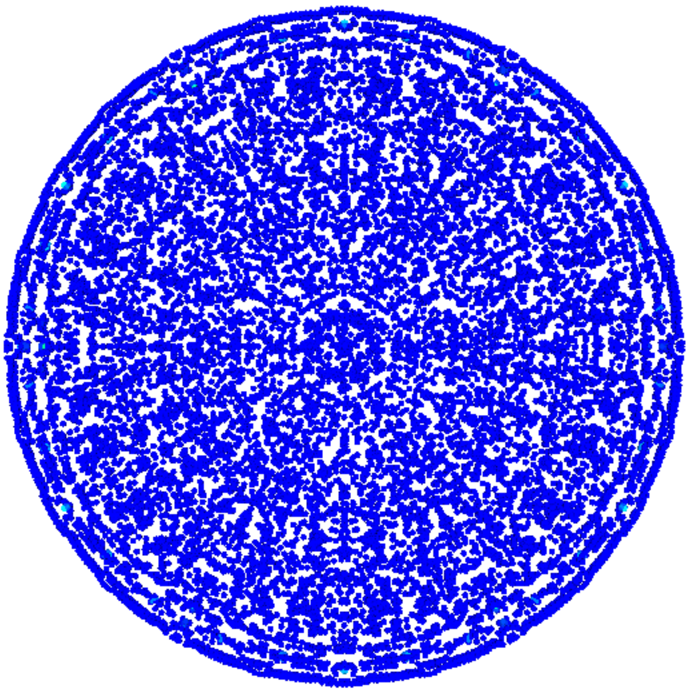
(c) 150ns



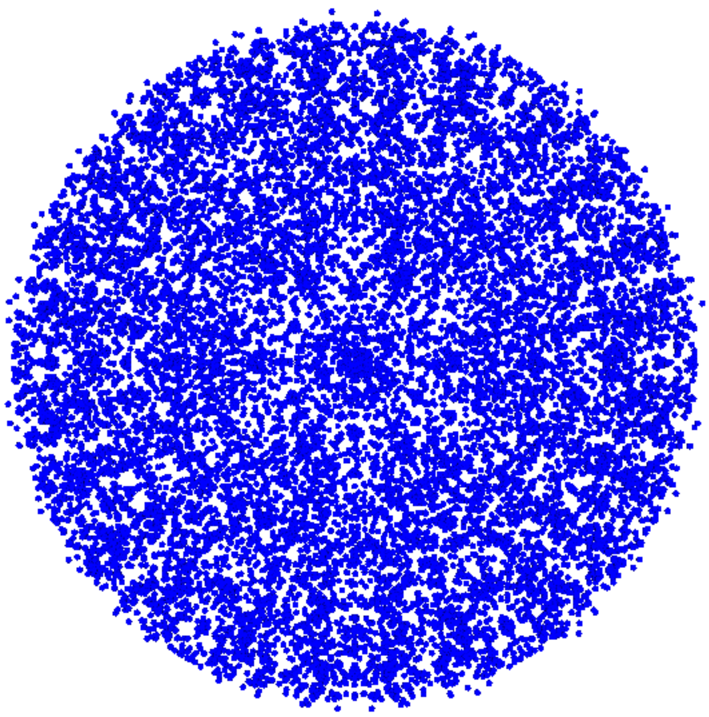
(d) 225ns



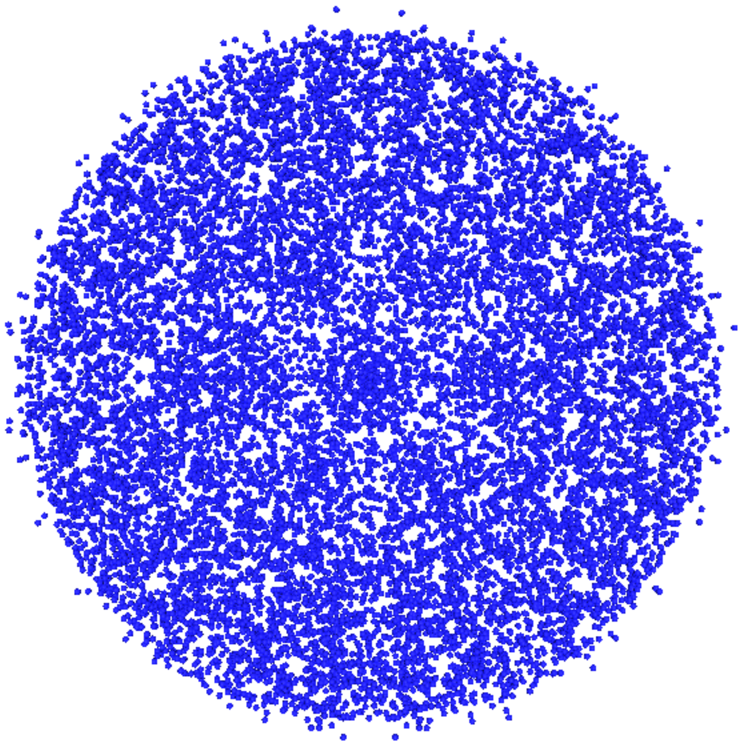
(e) 275ns



(f) 30ns



(g) 375ns



(h) 400ns

Figure 6.7: Top layer of the body $\mathscr{R}$ during the impact. Particles are colored by the amount of neighbors.

7 Conclusion and outlook

This thesis proves that it is possible to use the Compute Unified Device Architecture for the implementation of a Peridynamic technique on a GPU. In chapter 6 the results of the two experiments described in [SA05, P+08] are showing that it was possible to reproduce the two numerical experiments on the GPU with similar results. This proves that the Peridynamic technique was correctly implemented on the GPU with CUDA and it is possible to use the fast mathematical operations which are executed faster, but which are therefore less accurate.

In chapter 4.3 the implementation on the GPU is compared with the CPU implementation. To compare the two implementations a software metrics for objective, reproducible and quantifiable measurements has been defined. The results of the comparison with the metric presented in chapter 4.3 are plotted in figure 4.8. Considering metric presented in chapter 4.3 the GPU implementation is a factor ≈ 3 faster in the computation time for one step of the Peridynamic technique for one particle. Consider the theoretical factor by comparing the double precision floating point performance (PEAK) of the two devices is ≈ 5.0. In this case the measured difference in the computation time for one step of the Peridynamic technique for one particle is appropriate.

Outlook

Following improvements of the software are recommended:

1. Running the software on a High Performance Computing GPU.
2. Porting the other states to CUDA.
3. Boundary conditions as described in [Sil98].

The first improvement for the computation time is to use a High Performance Computing (HPC) GPU for the simulation. NVIDIA provides GPUs for different application areas and the *NVIDIA GeForce GTX 560 Ti* is designed for the gaming application area. For the application area HPC the *NVIDIA Tesla 20 Serie* is provided.

The advantage of *NVIDIA Tesla* 20 *Serie* for simulations is the accession of processing power in double precision. For example, the double precision floating point performance (PEAK) for the *NVIDIA TESLA*[1] *C*2075 is specified with 515 GFLOP's [NVI12]. The double precision floating point performance (PEAK) for the *NVIDIA GeForce GTX* 560 *Ti* is measured with 105 GFLOP's [Har]. It is theoretically possible to reduce the computation time by a factor 5 by using the *NVIDIA TESLA C*2075. The source code in the "C for CUDA" programming language can be compiled for different device architectures, but in most cases some customizing because of other device features has to be done.

The second improvement is to port the other states of the program flow chart (figure 4.4) to CUDA. The first state "Generate particles" can be easily ported to CUDA because the initial position of the particles in the reference configuration Ω has to be generated. In order to port the state "Search neighborhood of each particle" to CUDA, a parallel sorting algorithm with the morton order has to be implemented in CUDA first. In [SHG09] efficient sorting algorithms for many core GPUs are presented. Secondly the neighbor search has to be implemented into CUDA. In [GSSP10] an alternative algorithm for the neighbor search in CUDA is presented.

The third improvement is the implementation of the boundary conditions as described in [Sil98]. Boundary conditions are important in the conventional theory of continuum mechanics for specific solutions in equilibrium problems.

[1]TESLA is a unregistered trade mark of NVIDIA

Bibliography

[Col12] Collaborative Research Center (SFB) 716. Meshfree Multiscale Methods for Solids, 2012. URL `http://www.sfb716.uni-stuttgart.de/forschung/teilprojekte/projektbereich-d/d7.html`. (Quoted on page 28)

[cud11a] *CUDA API REFERENCE MANUAL*. 4.0. NVIDIA, 2011. (Quoted on page 15)

[cud11b] *NVIDIA CUDA C Programming Guide*. 4.0. NVIDIA, 2011. (Quoted on page 13)

[Edw02] H. C. Edwards. SIERRA Framework Version 3: Core Services Theory and Design. Technical report, Sandia National Laboratories, 2002. (Quoted on page 3)

[ELP] E. Emmrich, R. B. Lehoucq, D. Puhst. "Peridynamics: a nonlocal continuum theory". (Quoted on page 5)

[GSSP10] P. Goswami, P. Schlegel, B. Solenthaler, R. Pajarola. Interactive SPH Simulation and Rendering on the GPU. In M. Otaduy, Z. Popovic, editors, *Eurographics/ ACM SIGGRAPH Symposium on Computer Animation*. 2010. (Quoted on page 46)

[Har] HardTecs4U. NVIDIA GeForce GTX 560 TI im Test. URL `http://ht4u.net/reviews/2011/nvidia_geforce_gtx_560_ti_msi_n560ti_twin_frozr_2/index2.php`. (Quoted on the sides 22, 23 and 46)

[HLW03] E. Hairer, C. Lubich, G. Wanner. Geometric numerical integration illustrated by the Störmer/Verlet method. *Acta Numerica*, 12:399–450, 2003. (Quoted on page 12)

[Iva89] Ivan Summerville. *Software Engineering*. 3. Addision Wesley, 1989. (Quoted on page 22)

[Knu73] D. E. Knuth. *THE ART OF COMPUTER PROGRAMMING*, volume 3. Addision Wesley, Stanford University, 1973. (Quoted on page 18)

[Mar] Marc Büchel. Sandy Bridge: Core i7 2600K und Core i5 2500K. URL `http://www.ocaholic.ch/xoops/html/modules/smartsection/item.php?itemid=452&page=6`. (Quoted on the sides 22 and 23)

[MB96] J. Melenk, I. Babuška. The Partition of Unity Finite Element Method: Basictheory and applications. *Computer Methods in Applied Mechanics and Engineering*, 139:289–314, 1996. (Quoted on page 27)

[Mic09] Michael Connor and Piyush Kumar. Fast construction of k-Nearest Neighbor Graphs for Point Clouds. In *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS*. 2009. (Quoted on the sides 17 and 18)

[NVI12] NVIDIA. NVIDIA TESLA C2075 COMPANION PROCESSOR CALCULATE RESULTS EXPONENTIALLY FASTER, 2012. URL `http://www.nvidia.com/docs/IO/43395/NV-DS-Tesla-C2075.pdf`. (Quoted on page 46)

[P$^+$08] M. L. Parks, et al. Implementing peridynamics within a molecular dynamics code. In *Computer Physics Communications*, volume 179, pp. 777–783. ELSEVIER, 2008. (Quoted on the sides 9, 10, 37 and 45)

[Pli95] S. Plimpton. Fast Parallel Algorithms for Short–Range Molecular Dynamics. In *Journal of Computational Physics*, volume 117, pp. 1 – 19. 1995. (Quoted on page 3)

[PSP$^+$11] M. L. Parks, P. Seleson, S. J. Plimpton, S. A. Silling, R. B. Lehoucq. Peridynamics with LAMMPS: A User Guide v0.3 Beta. SANDIA REPORT, Sandia National Laboratories, 2011. (Quoted on the sides 3 and 22)

[pum97] The Partion of Unity Method. *International Journal for Numerical Methods in Engineering*, 40:727–758, 1997. (Quoted on page 27)

[SA05] S. A. Silling, E. Askari. A meshfree method based on the peridynamic model of solid mechanics. In *Computer & Structures*, volume 83, pp. 1526–1535. ELSEVIER, 2005. (Quoted on the sides 29 and 45)

[Sana] Sandia National Laboratories. EMU. URL `http://www.sandia.gov/emu/emu.htm`. (Quoted on the sides 3 and 22)

[Sanb] Sandia National Laboratories. LAMMPS Molecular Dynamics Simulator. URL `http://lammps.sandia.gov/`. (Quoted on page 3)

[Sch03] M. A. Schweitzer. *Parallel Multilevel Partition of Unity Method for Elliptic Partial Differential Equations*, volume 29. Springer, 2003. (Quoted on page 27)

[SHG09] N. Satish, M. Harris, M. Garland. Designing Efficient Sorting Algorithms for Manycore GPUs, 2009. To Appear in Proc. 23rd IEEE International Parallel and Distributed Processing Symposium. (Quoted on page 46)

[SIE11] SIERRA Solid Mechanics Team. Sierra/SolidMechanics 4.22 User's Guide. Technical report, Computational Solid Mechanics and Structural Dynamics Department (Sandia National Laboratories), 2011. (Quoted on the sides 3 and 22)

[Sil98] S. A. Silling. Reformulation of elasticity theory for discontinuities and long-range forces. Technical report, Sandia National Laboratories, 1998. (Quoted on the sides 29, 45 and 46)

[Sil00] S. A. Silling. Reformulation of elasticity theory for discontinuities and long-range forces. In *Journal of the Mechanics and Physics of Solids*, volume 48, pp. 175–209. Pergamon, 2000. (Quoted on the sides 5 and 29)

[STA] STANN. URL `https://sites.google.com/a/compgeom.com/stann/Home`. (Quoted on page 18)

[TH81] H. Tropf, H. Herzog. Multidimensional Range Search in Dynamically Balanced Trees. In *Angewandte Informatik (Applied Informatics)*, pp. 71–77. Vieweg Verlag, 1981. (Quoted on page 18)

All links were last followed on March 12, 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.

(Patrick Diehl)