Interpolation with Radial Basis Functions on GPGPUs using CUDA

D. Martin G. Haase

SFB-Report No. 2014-004 March 2014

Supported by the
Austrian Science Fund (FWF)

SFB sponsors:

- **Austrian Science Fund** (FWF)
- **University of Graz**
- **Graz University of Technology**
- **Medical University of Graz**
- **Government of Styria**
- **City of Graz**

Interpolation with Radial Basis Functions on GPGPUs using CUDA

D. Martin[1,2] and G. Haase[3]

[1]VRVis Research Center, Donau-City-Strasse 1, Vienna
[2]AVL List GmbH, Hans-List-Platz 1, Graz
[3]Institute for Mathematics and Scientific Computing
University of Graz

Abstract

This report gives a brief introduction to the interpolation with radial basis functions and it's application to the deformation of computational grids. The FGP algorithm is quoted as an iterative method for the calculation of the interpolation coefficients. A multipole method is described for the efficient approximation of the required matrix-vector product. Results are presented for different test systems and the results are compared with respect to the theoretical peak performances of the systems. The report concludes with a summary and an outlook to open issues.

1 Introduction

Although graphics processing units (GPUs) were historically developed to perform tasks at manipulating computer graphics efficiently they have evolved to more generalized computing devices during the last years. Today GPUs are a well established platform in many areas of high performance computing. The usage of GPUs for computing tasks other than computer graphics is often referred to as general-purpose computing on graphics processing units (GPGPU). In the same matter a GPU that is capable of general-purpose computing is called general-purpose graphics processing unit (GPGPU). The Compute Unified Device Architecture (CUDA[1]) is a proprietary computing platform and programming model provided by the NVIDIA corporation.
Interpolation with radial basis functions (RBF interpolation) is a widely used and well established tool in modern approximation theory to interpolate scattered data in several dimensions. In numerical simulation RBF interpolation can be used to interpolate data between different discretisations as well as a mighty, mesh independent tool for the deformation of computational grids.
Depending on the choice of the basis function the calculation of an interpoland leads to a system of linear equations with a dense system matrix. The problem sizes in industrial

[1]`http://www.nvidia.com/object/cuda_home_new.html`

applications often prohibit the direct solution of the linear system, therefore iterative methods with suitable preconditioning have to be used. In the application of Krylov-subspace methods the highest computational cost originates in the calculation of matrix-vector products with the dense system matrix.
In this report we will investigate the efficient usage of GPGPUs for the computation of a RBF interpolation. The remaining report is organized as follows. Section 2 will give a short introduction to RBF interpolation and the deformation of computational grids using RBF interpolation. An efficient method for the calculation of an interpoland given a certain basis function will be presented in section 3. Some results will be presented in section 4 and the results are compared with respect to the theoretical peak performances of the test systems in section 5. The report ends with a conclusion in section 6.

2 RBF interpolation

This section gives a short introduction to RBF interpolation, closely following [2, chapters 1,2,5], [9, chapters 6,7,8] and [3, chapter 1].

2.1 Radial basis functions

Definition 1. If a univariate real-valued function $\phi : [0, \infty) \to \mathbb{R}$ is used as a symmetric multivariate function $\Phi : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ via

$$\Phi(x, y) = \phi(\|x - y\|_2) \text{ for all } x, y \in \mathbb{R}^d,$$

then ϕ is called a **radial basis function (RBF)** and Φ is called the associated **kernel**.

The argument of a basis function is often written as $r := \|x - y\|_2$.

Definition 2. The **support** of a function u defined on $\Omega \subseteq \mathbb{R}^d$ is defined as

$$\text{supp } u := \overline{\{x \in \Omega : u(x) \neq 0\}}.$$

We distinguish between basis functions with local and global support. Basis functions with local support fulfill w.l.o.g. the condition

$$\phi(r) = 0, \text{ for } r > 1.$$

In application, basis functions are often scaled with a real factor $c > 0$, thus $\phi_c(r) = \phi(r/c)$. In the following only basis functions with global support are considered.

2.2 RBF Interpolation

Given is a set of points $\mathcal{X} = \{x_i\}_{i=1}^N$ in a domain $\Omega \subseteq \mathbb{R}^d$. A set of associated real function values $f_i = f(x_i)$ is assigned to these points. The function f is usually unknown, but it's existence is postulated for the reasonableness of the interpolation task. Sought is an approximating function $s : \Omega \to \mathbb{R}$ by interpolation. Restricted on the set $\mathcal{X}$ we request the interpolation condition

$$s|_{\mathcal{X}} = f|_{\mathcal{X}}. \tag{1}$$

In the context of RBF interpolation we seek for an interpoland of the form

$$s(x) = \sum_{i=1}^{N} \lambda_i \phi(\|x - x_i\|) + p(x), \quad \lambda_i \in \mathbb{R}, p \in \mathbb{P}^M. \tag{2}$$

The polynomial term p is required for the existence and uniqueness of a solution. A detailed analysis can be found in [2, chapter 5]. The required degree M for the existence of a solution depends in the choice of the basis function. Some popular choices of basis functions ϕ with global support and the required degrees M of the polynomial terms are listed in table 1. A degree -1 indicates that no polynomial term is required.

Name	$\phi(r)$		M
Gaussian	e^{-r^2}		-1
Inverse multiquadric biharmonics	$\frac{1}{\sqrt{r^2+c^2}}$	$c \in \mathbb{R}$	-1
Multiquadric biharmonics	$\sqrt{r^2+c^2}$	$c \in \mathbb{R}$	0
Linear	r		0
Thin-plate spline	$r^2 \log r$		1
Cubic	r^3		1

Table 1: Basis functions with global support

In the case that the choice of the basis function requires a polynomial term the given set of points $\mathcal{X}$ has to be unisolvent with respect to polynomials of the corresponding degree.

Definition 3. A set $\mathcal{X} \subset \mathbb{R}^d$ is called **unisolvent** for $\mathbb{P}_d^{k-1}$, if, for polynomials $p \in \mathbb{P}_d^{k-1}$

$$p|_{\mathcal{X}} = 0 \Rightarrow p \equiv 0.$$

Requiring the interpolation condition (1) in all given points and demanding a side condition on the coefficients of the polynomial term leads to a system of linear equations for the determination of the coefficients $\underline{\lambda}$ and $\underline{\pi}$:

$$\begin{aligned} \sum_{i=1}^{N} \lambda_i \phi(\|x_i - x_k\|) + \sum_{j=1}^{M} \pi_j p_j(x_k) &= f(x_k), & 1 \le k \le N, \\ \sum_{i=1}^{N} \lambda_i p_l(x_i) &= 0, & 1 \le l \le M, \end{aligned} \tag{3}$$

or, in short notation

$$\begin{pmatrix} \Phi & \Pi \\ \Pi^\top & 0 \end{pmatrix} \begin{pmatrix} \underline{\lambda} \\ \underline{\pi} \end{pmatrix} = \begin{pmatrix} \underline{f} \\ \underline{0} \end{pmatrix}. \tag{4}$$

2.2.1 Convergence analysis

The following short convergence analysis closely follows [2, chapter 5]. A detailed analysis on the underlying functional spaces can be found in [9, chapter 10].

Let X denote the functional space spanned by functions of the form (2), and let

$$D^{-k}L_2(\mathbb{R}^d)$$

denote the linear function space of all $f : \mathbb{R}^d \to \mathbb{R}$ whose kth total degree distributional partial derivatives are in $L_2(\mathbb{R}^d)$. Accordingly $D^{-k}L_2(\Omega)$ denotes the superspace of $D^{-k}L_2(\mathbb{R}^d)$ that contains all f with a finite semi-norm when restricted to the domain Ω.
A semi-inner product over this functional space is given by

$$(f, g)_* := \int_\Omega \sum_{|\alpha|=k} \frac{k!}{\alpha!} D^\alpha f(x) \overline{D^\alpha g(x)} dx, \tag{5}$$

where $\alpha \in \mathbb{N}_0^d$ and $k \in \mathbb{N}$. This semi-inner product has by design the non-trivial kernel $\mathbb{P}_d^{k-1}$ and induces the semi-norm

$$|f|_{k,\Omega}^2 : \int_\Omega \sum_{|\alpha|=k} \frac{k!}{\alpha!} |D^\alpha f(x)|^2 dx.$$

The following proposition states a key result on the uniqueness and the quality of an interpolating function, given that distinct assumptions ([2, chapter 5]) on ϕ and Ω hold.

Proposition 2.1. *[2, Prop. 5.2] Suppose a finite set $\mathcal{X} = \{x_i\}_{i=1}^N$ of distinct centres and the f_i are given. Let K be the k-dimensional kernel of the above semi-inner product. Then, if $\mathcal{X}$ contains a unisolvent subset with respect to the kernel K of the semi-norm, there is a unique s of the form*

$$s(x) = \sum_{i=1}^N \lambda_i \phi(x - x_i) + p(x), \quad x \in \mathbb{R}^d,$$

where $p \in K$ and $\sum_{i=1}^N \lambda_i q(x_i) = 0$ for all $q \in K$, such that it is the minimum norm interpolant giving $s(x_i) = f_i$, $i = 1, \dots, N$, and $|s|_{k,\Omega} \le |g|_{k,\Omega}$ for any other such $g \in X$ which interpolates the prescribed data.

2.3 Grid deformation using RBF Interpolation

If the definition of the deformation of a computational grid is known on a discrete set of points (not necessarily nodes of the grid), RBF interpolation can be used to determine the displacement of all nodes of the grid in a straightforward manner [5].
The given displacements $\{\underline{d}_i\}_{i=1}^N \subset \mathbb{R}^d$ in N points $\{y_i\}_{i=1}^N \subset \mathbb{R}^d$ are treated component wise. Thus for the deformation of a d-dimensional computational grid we seek for d approximating functions to describe the deformation of the discretisation of the domain Ω, the given computational grid.
For each spacial dimension δ an interpoland of the form

$$s_\delta(x) = \sum_{i=1}^N \phi(\|x - y_i\|) + p(x)$$

has to be determined. The evaluation of the interpoland in the nodes of the computational grid serves as the deformation rule for the grid in the corresponding dimension.

3 Efficient calculation of an interpoland

Analysis of the properties of the resulting system of linear equations (4) (e.g. [2, chapter 5]) yields that a direct solution is inhibited if the number of interpolation points exceeds certain limits. The following section presents an iterative method to solve the resulting system of linear equations [8, 7, 6]. The remaining computational costs lie in the calculation of matrix-vector products with dense matrices. Exploiting knowledge of the generating kernel allows the usage of a multipole-method [1]. The last subsection presents the far field series expansion for the multiquadric basis function [4].

3.1 The FGP algorithm

The FGP algorithm is a Krylow-subspace method to solve the system of linear equations (4). The implemented version of the algorithm can be applied to interpolations with basis functions where constant polynomial terms are sufficient. The following description of the algorithm is suitable for the basis function $\sqrt{r^2 + c^2}$. Other basis functions may require a different sign at the definition of the semi-inner product.

Let X denote the functional space spanned by functions of the form (2). The basis for the algorithm is the semi-inner product

$$\langle s, t \rangle_\phi = -\underline{\lambda}^\top \Phi \underline{\mu}$$

for $s, t \in X$ with $s(x) = \sum_{i=1}^N \lambda_i \phi(\|x - x_i\|) + \alpha$ and $t(x) = \sum_{i=1}^N \mu_i \phi(\|x - x_i\|) + \beta$, induced by the radial basis function ϕ. Φ denotes the associated kernel to ϕ as in definition 1. This semi-inner product is equivalent to the semi-inner product (5) ([9, chapter 10]), again with the non-trivial kernel $\mathbb{P}_d^{k-1}$. The semi-inner product $\langle \cdot, \cdot \rangle_\phi$ induces a semi-norm

$$|s|_\phi = \langle s, s \rangle^{1/2} = \left(-\underline{\lambda}^\top \Phi \underline{\lambda} \right)^{1/2}.$$

Considering the relations

$$\begin{aligned}
\langle s, t \rangle_\phi &= -\underline{\lambda}^\top \Phi \underline{\mu} \\
&= -\sum_{i=1}^N \lambda_i \left\{ \sum_{j=1}^N \phi(\|x_i - x_j\|) \mu_j \right\} \\
&\overset{(3)}{=} -\sum_{i=1}^N \lambda_i \left\{ \sum_{j=1}^N \phi(\|x_i - x_j\|) \mu_j \right\} - \beta \sum_{i=1}^N \lambda_i \\
&= -\sum_{i=1}^N \lambda_i \left\{ \sum_{j=1}^N \mu_j \phi(\|x_i - x_j\|) + \beta \right\} \\
&= -\sum_{i=1}^N \lambda_i t(x_i),
\end{aligned}$$

it is apparent that if the function values of s or t are known in the interpolation centers, the evaluation of the semi-inner product does not require the calculation of a matrix-vector product. The symmetry $\langle s,t\rangle_\phi = \langle t,s\rangle_\phi$ yields

$$-\sum_{i=1}^N \lambda_i t(x_i) = -\sum_{i=1}^N \mu_i s(x_i).$$

The FGP algorithm uses a linear operator $A : X \to X$. A is chosen such, that for all iterations k the resulting searching direction lies in the subspace of X that is spanned by the functions $A^l s^\star$, $l = 1,\ldots,k$, where $s^\star$ denotes the sought interpoland. In addition to this A performs some preconditioning. A is an approximation of the optimal preconditioning operator A^{opt}. The operator A^{opt} can be derived from the interpolation tasks

$$\hat{u}_j(x) = \sum_{i=1}^N \zeta_{j,i}\phi(x-x_i) + \beta, \quad \text{für } x \in \Omega, j = 1,\ldots,N,$$

due to the Lagrange conditions

$$\hat{u}_j(x_i) = \delta_{ij}, \quad i,j = 1,\ldots,N,$$

where δ_{ij} denotes the Kronecker-delta

$$\delta_{ij} = \begin{cases} 1, & i = j \\ 0, & i \neq j. \end{cases}$$

For the construction of A the functions $\hat{u}_j$, $j = 1,\ldots,N$ are determined by solving the interpolation tasks on subsets of $\mathcal{X}$, that contain not more than q interpolation centers. Generally the relation $q \ll N$ holds. These subsets are called $\mathcal{L}$-sets.

3.1.1 Description of the algorithm

Let $s^\star$ denote the sought interpoland and k the current iteration. The interpoland at the begin of the k-th iteration is

$$s^{(k)}(x) = \sum_{i=1}^N \lambda_i^{(k)} \phi(\|x-x_i\|) + \alpha^{(k)},$$

the residual at the beginning of the k-th iteration is

$$r_i^{(k)} = f_i - s^{(k)}(x_i), \quad i = 1,\ldots,N. \tag{6}$$

The algorithm terminates when the criterion

$$\left\| r^{(k)} \right\|_\infty \leq \varepsilon \tag{7}$$

is satisfied, where $\varepsilon \geq 0$ denotes the required accuracy of the solution.

STEP 1: The interpolation centers x_i, $i = 1,\ldots,N$ and the associated function values f_i are given. An integer $q \geq 2$ is chosen to construct the linear Krylov-subspace operator. Typical values for q range from 30 to 50.

STEP 2: The approximand $s^{(1)}$ is initialized with the parameters

$$\begin{aligned}\lambda_j^{(1)} &= 0, \quad j = 1, \dots, N, \\ \alpha^{(1)} &= \frac{1}{2} \left(\min(f_1, \dots, f_N) + \max(f_1, \dots, f_N)\right).\end{aligned}$$

The initial residual is determined by

$$r_i^{(1)} = f_i - s^{(1)}(x_i) = f_i - \alpha^{(1)}, \quad i = 1, \dots, N.$$

The algorithm terminates if (7) is satisfied. Otherwise, steps 3 to 5 prepare the iterative process.
STEP 3: A random permutation $\sigma(i)$,$i = 1, \dots, N$ of the indices $\{1, \dots, N\}$ is generated, the counting variable m is initialized with $m = 1$.

STEP 4: The variable l is set to $l = \sigma(m)$. Let Σ_l denote the set of indices $\{\sigma(m), \dots, \sigma(N)\}$. If $N - m + 1 > q$, the point set $\mathcal{L}_l$ of q is chosen such that $\|x_l - x_i\|_{i \in \mathcal{L}_l} \leq \|x_l - x_j\|_{j \in \Sigma_l \setminus \mathcal{L}_l}$ is satisfied. Otherwise, if $n - m + 1 \leq q$, $\mathcal{L}_l = \{\sigma(m), \dots, \sigma(n)\}$.

STEP 5: The coefficients ζ_{lj}, $j \in \mathcal{L}_j$ and ψ_l of the functions

$$\hat{z}_l(x) = \sum_{j \in \mathcal{L}_l} \zeta_{lj} \phi(\|x - x_j\|) + \psi_l$$

are determined. The required equations result from the Lagrange conditions

$$\hat{z}_l(x_j) = \delta_{lj}, \quad l \in \mathcal{L}_l$$

and the side condition $\sum_{j \in \mathcal{L}_l} \zeta_{lj} = 0$, where δ_{lj} denotes the Kronecker-delta.
While $m \leq N - 2$ is satisfied, the counter m is increased by 1 and the algorithm continues at step 4. Otherwise the iteration counter k is set to 1 and the iteration process starts with step 6.

STEP 6: The coefficients of the function

$$t^{(k)}(x) = \sum_{j=1}^{N} \tau_j^{(k)} \Phi(x, x_j)$$

are determined. Therefore $\underline{\tau}$ is initialized $\underline{\tau} = \underline{0}$. Then l loops over the indices $\{\sigma(1), \dots, \sigma(N-1)\}$. For each l

$$\mu_l^{(k)} = \sum_{i \in \mathcal{L}_l} \zeta_{li} r_i^{(k)} / \zeta_{ll}$$

with the current residual $\underline{r}^{(k)}$ and the previously calculated ζ_{ij}. For all $j \in \mathcal{L}_j$ the resulting $\mu_l^{(k)}$ is added to $\tau_j^{(k)}$.

STEP 7: The searching direction

$$d^{(k)}(x) = \sum_{j=1}^{N} \delta_j^{(k)} \Phi(x, x_j)$$

is determined. In the first iteration ($k = 1$), $\underline{\delta}^{(1)} = \underline{\tau}^{(1)}$. Otherwise ($k > 1$),

$$\delta_j^{(k)} = \tau_j^{(k)} - \beta_k \delta_j^{(k-1)}, \quad j = 1, \ldots, N,$$

with

$$\beta_k = \sum_{i=1}^{N} \tau_i^{(k)} d^{(k-1)}(x_i) \Bigg/ \sum_{i=1}^{N} \delta_i^{(k-1)} d^{(k-1)}(x_i)$$

Step 8: The product $\Phi\underline{\delta}$ is calculated to determine the function values $d^{(k)}(x_i)$, $i = 1, \ldots, N$.

Step 9: The step with of the current iteration is determined by

$$\gamma^{(k)} = \sum_{i=1}^{N} \delta_i^{(k)} r_i^{(k)} \Bigg/ \sum_{i=1}^{N} \delta_i^{(k)} d^{(k)}(x_i) .$$

The new approximand $s^{(k+1)}$ is set to

$$s^{(k+1)}(x) = s^{(k)}(x) + \gamma^{(k)} d^{(k)}(x) + \omega^{(k)},$$

where $\omega^{(k)} \in \mathbb{R}$ is yet to be determined. Equation (6) yields, that the new residual vector has the form

$$r_i^{(k+1)} = r_i^{(k)} - \gamma^{(k)} d^{(k)}(x_i) - \omega^{(k)}, \quad i = 1, \ldots, N,$$

and thus $\omega^{(k)}$ is chosen such that $\max\left\{|r_i^{(k+1)}|, i = 1, \ldots, N\right\}$ is minimized. The coefficients of the interpoland $s^{(k+1)}$ are determined by

$$\begin{aligned} \lambda_i^{(k+1)} &= \lambda_i^{(k)} + \gamma^{(k)} \delta_i^{(k)}, \quad i = 1, \ldots, N \\ \alpha^{(k+1)} &= \alpha^{(k)} + \omega^{(k)}. \end{aligned}$$

Step 10: The iteration counter k is increased by 1. If the termination condition (7) is satisfied, the algorithm terminates with the solution coefficients $\underline{\lambda}^{(k)}$ and $\alpha^{(k)}$, Otherwise a new iteration starts at step 6.

3.2 Multipole methods

Analysis of the FGP algorithm shows that there are two parts with non linear runtime with respect to the given number of interpolation centers N if they are implemented directly.
The first is the construction of the $\mathcal{L}$-sets. A brute force construction of these sets shows a runtime behaviour of $O(N^2)$. The usage of a spatial hierarchy (e.g. an octree) can reduce the required runtime to $O(N \log N)$.
The second is the evaluation of the matrix-vector product in step 8. Since the matrix Φ is dense, the evaluation of the matrix-vector product has a runtime of $O(N^2)$ and a memory consumption of $O(N^2)$. Calculating the entries of the matrix as needed omits the memory consumption but increases the runtime by some constant factor.

Multipole methods can be used to reduce the evaluation time of sums of the form

$$u(x) = \sum_{i=1}^{N} w_i K(x, y_i) \tag{8}$$

for points $x \in \Omega$ and sources $\{y_i\}_{i=1,\dots,N} \subset \Omega$ with respect to the kernel $K : \Omega \times \Omega \to \mathbb{R}$ with real weights w_i over the domain $\Omega \subseteq \mathbb{R}^d$. The goal of multipole methods is the reduction of the evaluation costs from $O(N^2)$ to $O(N \log N)$ or $O(N)$.

The basis for the multipole method are sums of the form (8) where the kernel can be expressed as a finite sum

$$K(x, y) = \sum_{k=1}^{p} \phi_k(x)\psi_k(y). \tag{9}$$

In this case the moments

$$A_k = \sum_{i=1}^{N} w_i \psi_k(y_i)$$

can be calculated independently from any evaluation point x. The evaluation of $u(x)$ equals the evaluation of the sum

$$u(x) = \sum_{k=1}^{p} A_k \phi_k(x).$$

This leads to overall computational costs of $O(Np)$ for the evaluation in N points. The computational costs are significantly reduced whenever $p \ll N$.

Many kernels $K(x, y)$ can not be expressed as a finite sum (9) but as an infinite series expansion

$$K(x, y) = \sum_{k=1}^{\infty} \phi_k(x)\psi_k(y).$$

If an approximate evaluation of the sum (8) is acceptable, the kernel can be approximated with a truncated series expansion.

The implementation of this project closely follows the multipole method for $\mathbb{R}^3$ as it is introduced in [1].

3.3 Series expansion for the multiquadric basis function

This section provides a series expansion for the kernel $\phi(r) = \sqrt{r^2 + c^2}$ as in [4].

The goal is the determination of a far field series expansion for the efficient evaluation of sums of the form

$$u(x) = \sum_{i=1}^{N} w_i \, \phi_i \left(x\right),$$

where $\phi : \mathbb{R}^d \to \mathbb{R}$ is the basis function (or kernel)

$$\phi_i(x) = \phi_i(x, c) = \left((x - x_i)^2 + c^2\right)^{1/2} \tag{10}$$

with $c \geq 0$ and $x \in \mathbb{R}^d$.
The following series expansions use Polynomials P_l of the form

$$P_l(\alpha, \beta, \gamma) = \sum_{j=\lfloor \frac{l+1}{2} \rfloor}^{l} \binom{1/2}{j} \binom{j}{l-j} \beta^{2j-l} (\alpha\gamma)^{l-j}, \tag{11}$$

where $l \geq 0$ and $\alpha, \beta, \gamma \in \mathbb{R}^d$. For negative l we identify $P_l \equiv 0$.
The following lemma states a far field series expansion for the kernel (10) and a bound for the error for the truncated expansion of degree $p+1$.

Lemma 3.1. *[4, Lemma 3.1] Let $t \in \mathbb{R}^d$ and $c \geq 0$. For all $x \in \mathbb{R}^d$ where $\|x\| > \sqrt{\|t\|^2 + c^2}$*

$$\phi(x-t) = \left(\|x-t\|^2 + c^2\right)^{1/2} = \sum_{l=0}^{\infty} P_l(\|t\|^2 + c^2, -2\langle t, x\rangle, \|x\|^2) \Big/ \|x\|^{2l-1}$$

using polynomials P_l as in equation (11).Further

$$\left|\phi(x-t) - \sum_{l=0}^{p+1} P_l(\|t\|^2 + c^2, -2\langle t, x\rangle, \|x\|^2) \Big/ \|x\|^{2l-1}\right|$$
$$\leq (2\sqrt{\|t\|^2 + c^2}) \left(\frac{\sqrt{\|t\|^2 + c^2}}{\|x\|}\right)^{p+1} \frac{\|x\|}{\|x\| - \sqrt{\|t\|^2 + c^2}}$$

holds for $\|x\| > \sqrt{\|t\|^2 + c^2}$ and $p+1 > 0$

The following lemma can be used to derive an efficient calculation routine for the coefficients of the far field series expansion.

Lemma 3.2. *[4, Lemma 2.1] The polynomials as in (11) satisfy the recursion*

$$(l+1)P_{l+1}(\alpha, \beta, \gamma) = \left(\frac{1}{2}\right) \beta P_l(\alpha, \beta, \gamma) + (2-l)\, \alpha\gamma P_{l-1}(\alpha, \beta, \gamma).$$

for all $\alpha, \beta, \gamma \in \mathbb{R}^d$ und $l \in \mathbb{N}$.

As a direct implication of lemma 3.2 one can formulate the following recursion for the calculation of the series coefficients.
Writing

$$G_l(x) = P_l(\|t\|^2 + c^2, -2\langle t, x\rangle, \|x\|^2),$$

G_l is a homogeneous polynomial of degree l in x, depending in c and t. The far field expansion for a center with weight w is

$$\sum_{l=0}^{p+1} w G_l(x) / \|x\|^{2l-1}. \tag{12}$$

The polynomials G_l follow the recursion

$$G_l(x) = \begin{cases} 1, & l = 0, \\ -\langle x, t\rangle, & l = 1, \\ A_l \langle x, t\rangle G_{l-1}(x) + B_l \|x\|^2 (\|t\|^2 + c^2) G_{l-2}(x), & l \geq 2, \end{cases} \tag{13}$$

where

$$A_l = -\frac{2l-3}{l}, \quad B_l = \frac{3-l}{l}.$$

4 Results

All results were achieved using the GNU C++ compiler `gcc` version 4.6.1. unless specified otherwise below. The code running on the CPU was parallelized using OpenMP, with the number of threads matching the number of available cores. The used compiler options are:

```
-ffast-math -O3 -funroll-all-loops -ftree-vectorizer-verbose=2 -fopenmp
```

The GPU code was compiled using Nvidia Cuda release 5.0 and the contained compiler `nvcc` version V0.2.1221. The compiler options used on all systems were

```
-use_fast_math -O3 -arch sm_30 -ptxas-options=-v,
```

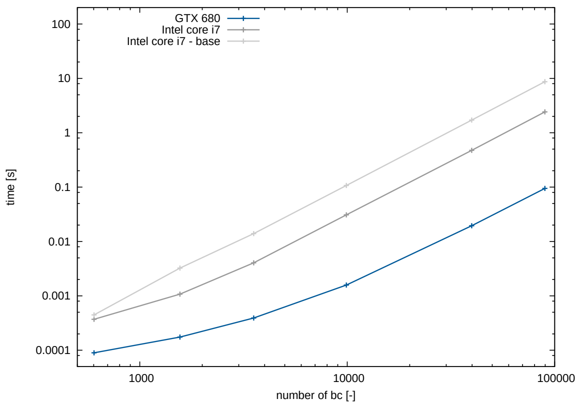
where the statement specifying the target architecture `-arch sm_XX` was adjusted to the test system.

4.1 Test system I

The test system is equipped with one Intel Core i7 2600K processor with 4 cores. The OpenMP parallelized CPU code used 8 threads due to active hyper threading. No significant performance difference was measured for 4 used threads. The GPU used for the computations is a Nvidia GTX 680 with compute capability 3.0.

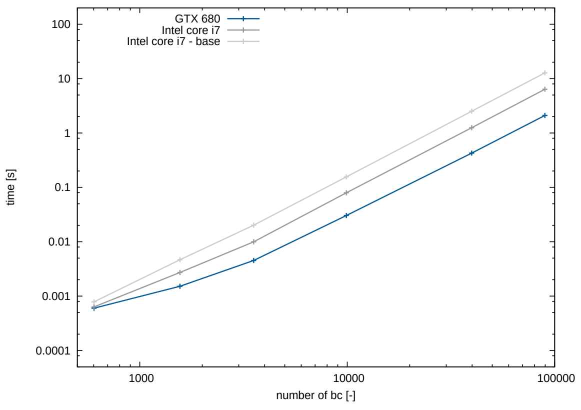
Figure 1 shows the calculation time for one directly calculated matrix-vector product in single precision over the number of boundary condition nodes. The performed calculations and the calculation times are listed in the table anlongside. The column 'CPU original' shows the results for the original code containing a conditional for the diagonal entries of the matrix. The column 'CPU no cond.' holds the results for the code with these conditionals eliminated. Column 'GPU' shows the results on the GPU.

The calculation times for the directly computed matrix-vector product in double precision are shown in figure 2.



# bc	CPU original [s]	CPU no cond. [s]	GPU [s]
602	4.47E-04	3.71E-04	8.96E-05
1562	3.25E-03	1.08E-03	1.75E-04
3542	1.40E-02	4.05E-03	3.92E-04
9902	1.07E-01	3.09E-02	1.58E-03
39802	1.71E+00	4.74E-01	1.95E-02
89702	8.68E+00	2.42E+00	9.48E-02

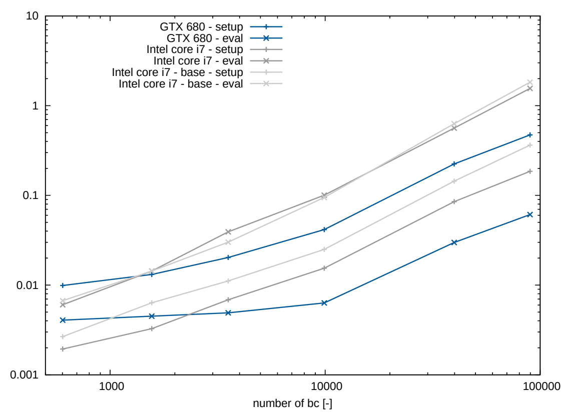
Figure 1: Brute force multiplication with single precision



# bc	CPU original [s]	CPU no cond. [s]	GPU [s]
602	7.84E-04	6.34E-04	6.00E-04
1562	4.67E-03	2.72E-03	1.52E-03
3542	2.02E-02	9.98E-03	4.52E-03
9902	1.56E-01	7.93E-02	3.04E-02
39802	2.51E+00	1.25E+00	4.26E-01
89702	1.28E+01	6.37E+00	2.10E+00

Figure 2: Brute force multiplication with double precision

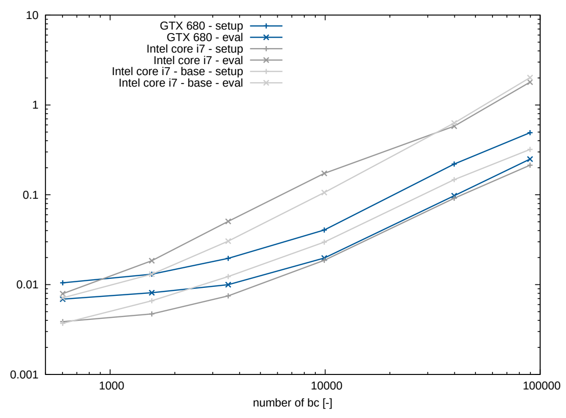
Figure 3 shows the calculation times for a matrix-vector product approximated with the multipole method (see section 3) in single precision over the number of boundary condition nodes. The performed calculations and the calculation times are listed in the table anlongside. The columns 'CPU setup' and 'CPU eval' show the measured times for the setup stage and the evaluation stage of the original implementation of the multipole method. Columns 'CPU setup mod.' and 'CPU eval mod.' show the measured times for a code that implements the modifications for the GPU . Columns 'GPU setup' and 'GPU eval' show the measured times for the calculations on the GPU.



# bc	CPU setup	CPU eval	CPU setup mod.	CPU eval mod.	GPU setup	GPU eval
602	2.67E-03	6.70E-03	1.94E-03	6.04E-03	9.91E-03	4.07E-03
1562	6.36E-03	1.43E-02	3.27E-03	1.43E-02	1.32E-02	4.51E-03
3542	1.11E-02	3.01E-02	6.87E-03	3.92E-02	2.03E-02	4.91E-03
9902	2.50E-02	9.48E-02	1.54E-02	1.01E-01	4.15E-02	6.33E-03
39802	1.44E-01	6.30E-01	8.51E-02	5.63E-01	2.24E-01	2.98E-02
89702	3.63E-01	1.83E+00	1.85E-01	1.56E+00	4.72E-01	6.11E-02

Figure 3: Multipole approximated multiplication with single precision

The calculation times for the multipole approximated matrix-vector product in double precision are shown in figure 4 and the table anlongside.



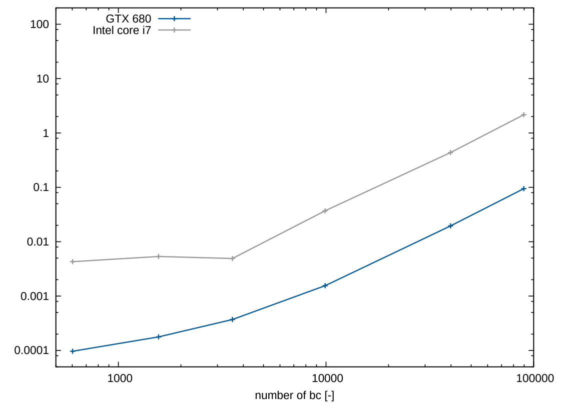
# bc	CPU setup	CPU eval	CPU setup mod.	CPU eval mod.	GPU setup	GPU eval
602	3.71E-03	7.12E-03	3.87E-03	7.92E-03	1.05E-02	6.89E-03
1562	6.61E-03	1.31E-02	4.72E-03	1.85E-02	1.31E-02	8.12E-03
3542	1.23E-02	3.05E-02	7.50E-03	5.06E-02	1.96E-02	9.98E-03
9902	2.97E-02	1.06E-01	1.86E-02	1.73E-01	4.05E-02	1.98E-02
39802	1.47E-01	6.30E-01	9.16E-02	5.81E-01	2.20E-01	9.77E-02
89702	3.20E-01	2.02E+00	2.14E-01	1.80E+00	4.91E-01	2.50E-01

Figure 4: Multipole approximated multiplication with double precision

The remainder of this subsection states the results achieved using the Intel C++ compiler `icpc` version 12.0.4. The used compiler options were

`-w -O3 -static -funroll-all-loops -openmp`.

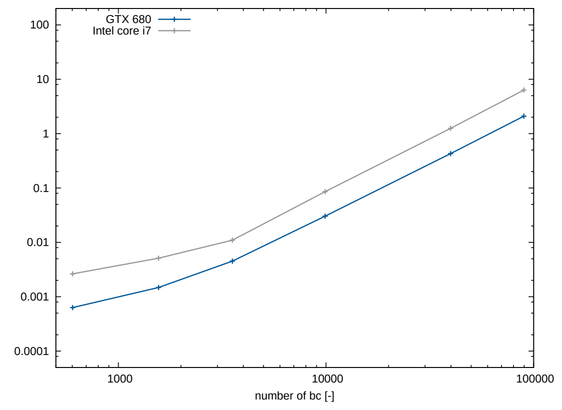
Figure 5 shows the calculation time for one directly calculated matrix-vector product in single precision over the number of boundary condition nodes. The performed calculations and the calculation times are listed in the table anlongside.



# bc	CPU no cond. [s]	GPU [s]
602	4.29E-03	9.68E-05
1562	5.36E-03	1.77E-04
3542	4.91E-03	3.71E-04
9902	3.70E-02	1.55E-03
39802	4.37E-01	1.95E-02
89702	2.17E+00	9.48E-02

Figure 5: Brute force multiplication with single precision - using Intel icpc

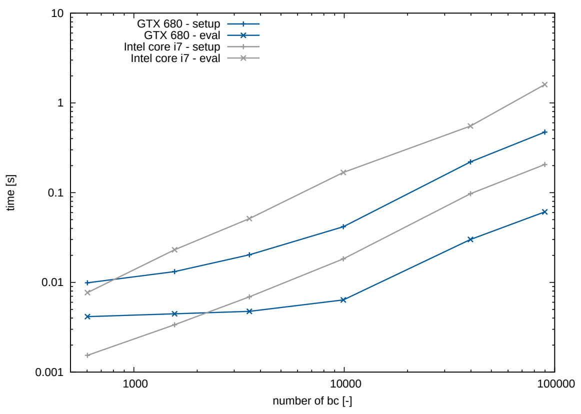
The calculation times for the directly computed matrix-vector product in double precision are shown in figure 6 and the table anlongside.



# bc	CPU no cond. [s]	GPU [s]
602	2.63E-03	6.31E-04
1562	5.12E-03	1.48E-03
3542	1.10E-02	4.52E-03
9902	8.58E-02	3.03E-02
39802	1.25E+00	4.29E-01
89702	6.34E+00	2.09E+00

Figure 6: Brute force multiplication with double precision - using Intel icpc

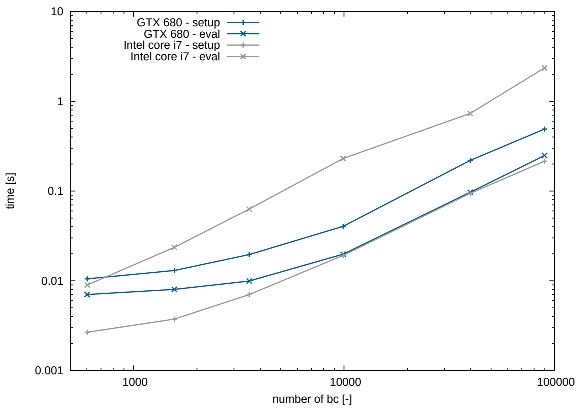
Figure 7 shows the calculation times for a matrix-vector product approximated with the multipole method in single precision over the number of boundary condition nodes. The performed calculations and the calculation times are listed in the table anlongside.



# bc	CPU setup mod.	CPU eval mod.	GPU setup	GPU eval
602	1.54E-03	7.70E-03	9.90E-03	4.15E-03
1562	3.37E-03	2.31E-02	1.32E-02	4.47E-03
3542	6.90E-03	5.13E-02	2.03E-02	4.76E-03
9902	1.83E-02	1.68E-01	4.17E-02	6.38E-03
39802	9.74E-02	5.52E-01	2.20E-01	3.01E-02
89702	2.06E-01	1.60E+00	4.72E-01	6.10E-02

Figure 7: Multipole approximated multiplication with single precision - using Intel icpc

The calculation times for the multipole approximated matrix-vector product in double precision are shown in figure 8 and the table anlongside.

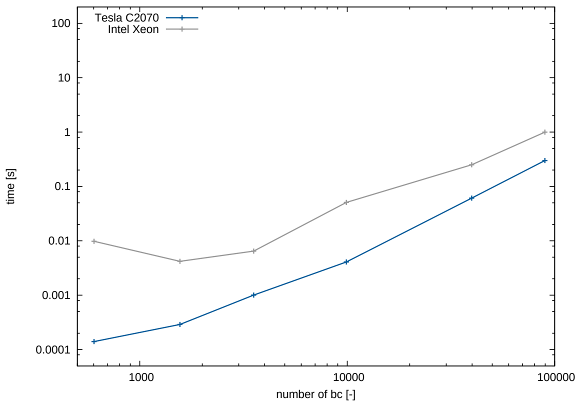


# bc	CPU setup mod.	CPU eval mod.	GPU setup	GPU eval
602	2.67E-03	8.98E-03	1.05E-02	7.03E-03
1562	3.74E-03	2.36E-02	1.30E-02	8.01E-03
3542	7.02E-03	6.29E-02	1.96E-02	9.94E-03
9902	1.92E-02	2.31E-01	4.04E-02	1.98E-02
39802	9.44E-02	7.33E-01	2.20E-01	9.69E-02
89702	2.15E-01	2.36E+00	4.91E-01	2.49E-01

Figure 8: Multipole approximated multiplication with double precision - using Intel icpc

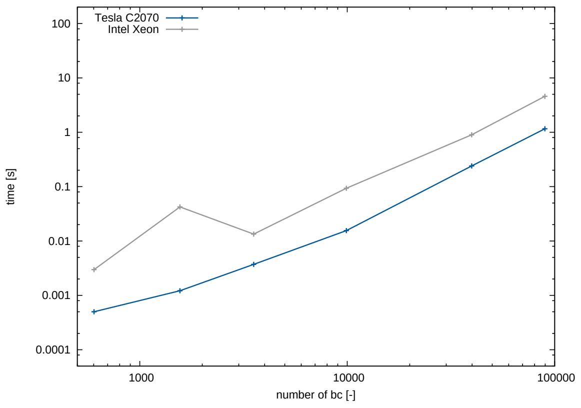
4.2 Test system II

The test system is compute node 4 of the 'Mephisto'-GPU-cluster. It is equipped with two Intel Xeon X5650 processors with 6 cores each. The OpenMP parallelized CPU code used 24 threads due to active hyper threading. The GPU used for the computations is a Nvidia Tesla C2070 with compute capability 2.0. The used C++ compiler was Intel `icpc` version 12.0.4. The used compiler options were as in subsection 4.1, the target architecture for the GPU code was adjusted to `-arch sm_20`.
Figure 9 shows the calculation time for one directly calculated matrix-vector product in single precision over the number of boundary condition nodes. The performed calculations and the calculation times are listed in the table anlongside. The calculation times for the directly computed matrix-vector product in double precision are shown in figure 10 and the table anlongside.



# bc	CPU no cond. [s]	GPU [s]
602	9.82E-03	1.40E-04
1562	4.20E-03	2.89E-04
3542	6.46E-03	1.00E-03
9902	5.08E-02	4.08E-03
39802	2.50E-01	6.10E-02
89702	9.97E-01	3.00E-01

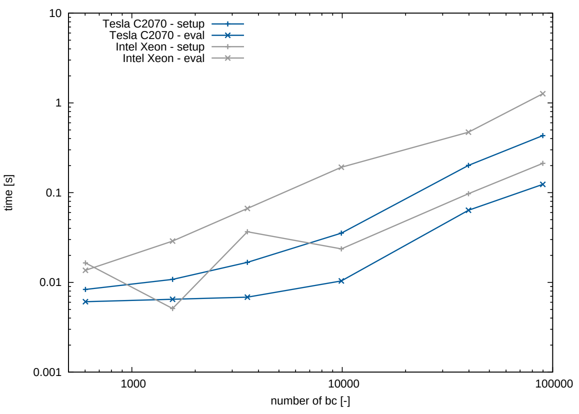
Figure 9: Brute force multiplication with single precision - using Intel icpc



# bc	CPU no cond. [s]	GPU [s]
602	2.97E-03	4.99E-04
1562	4.24E-02	1.22E-03
3542	1.34E-02	3.72E-03
9902	9.33E-02	1.55E-02
39802	8.99E-01	2.40E-01
89702	4.58E+00	1.16E+00

Figure 10: Brute force multiplication with double precision - using Intel icpc

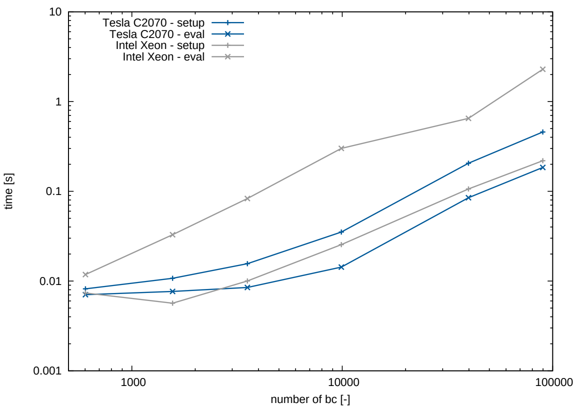
Figure 11 shows the calculation times for a matrix-vector product approximated with the multipole method in single precision over the number of boundary condition nodes. The performed calculations and the calculation times are listed in the table anlongside.



# bc	CPU setup mod.	CPU eval mod.	GPU setup	GPU eval
602	1.65E-02	1.36E-02	8.35E-03	6.09E-03
1562	5.12E-03	2.88E-02	1.08E-02	6.49E-03
3542	3.66E-02	6.67E-02	1.67E-02	6.85E-03
9902	2.36E-02	1.92E-01	3.54E-02	1.04E-02
39802	9.73E-02	4.72E-01	2.01E-01	6.37E-02
89702	2.12E-01	1.27E+00	4.32E-01	1.24E-01

Figure 11: Multipole approximated multiplication with single precision - using Intel icpc

The calculation times for the multipole approximated matrix-vector product in double precision are shown in figure 12 and the table anlongside.



# bc	CPU setup mod.	CPU eval mod.	GPU setup	GPU eval
602	7.36E-03	1.18E-02	8.18E-03	7.05E-03
1562	5.66E-03	3.28E-02	1.07E-02	7.67E-03
3542	1.00E-02	8.30E-02	1.56E-02	8.49E-03
9902	2.55E-02	3.01E-01	3.52E-02	1.43E-02
39802	1.06E-01	6.49E-01	2.05E-01	8.48E-02
89702	2.19E-01	2.29E+00	4.58E-01	1.84E-01

Figure 12: Multipole approximated multiplication with double precision - using Intel icpc

4.3 Test system III

The test system is equipped with one Intel Core2 6600 processor with 2 cores. The OpenMP parallelized CPU code used 2 threads. The GPU used for the computations is a Nvidia GTX 650Ti with compute capability 3.0.

Figure 13 shows the calculation time for one directly calculated matrix-vector product in double precision over the number of boundary condition nodes. The performed calculations and the calculation times using single and double precision are listed in table 2.

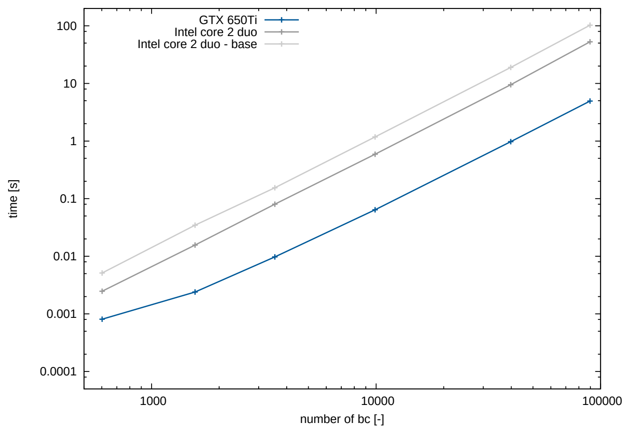


Figure 13: Brute force multiplication with double precision

# bc	CPU original [s]	CPU no cond. [s]	GPU [s]	# bc	CPU original [s]	CPU no cond. [s]	GPU [s]
602	2.92E-03	1.45E-03	2.92E-04	602	5.12E-03	2.48E-03	8.10E-04
1562	1.92E-02	4.68E-03	3.73E-04	1562	3.46E-02	1.56E-02	2.40E-03
3542	7.90E-02	2.01E-02	7.48E-04	3542	1.54E-01	8.01E-02	9.74E-03
9902	5.81E-01	1.47E-01	3.28E-03	9902	1.18E+00	5.91E-01	6.40E-02
39802	9.31E+00	2.33E+00	4.35E-02	39802	1.89E+01	9.48E+00	9.74E-01
89702	5.13E+01	1.20E+01	2.17E-01	89702	1.02E+02	5.28E+01	4.93E+00

Table 2: Brute force multiplication with single and double precision

Figure 14 shows the calculation time for one multipole approximated matrix-vector product in double precision over the number of boundary condition nodes. The performed calculations and the calculation times using single precision are listed in table 3, the calculation times using double precision are listed in table 4.

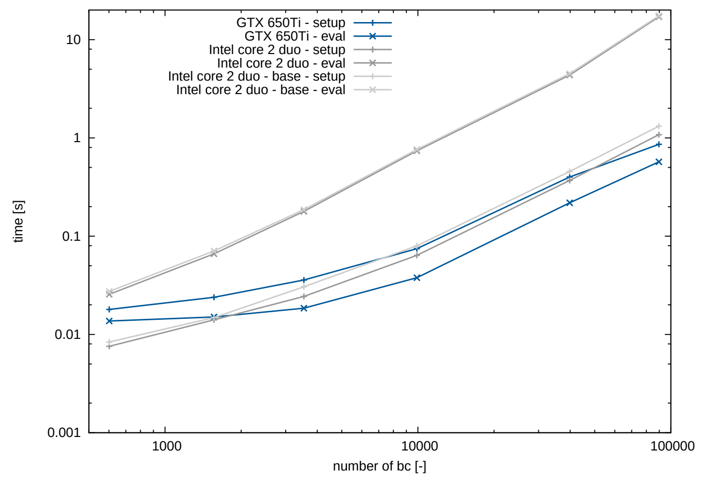


Figure 14: Multipole double precision

# bc	CPU setup	CPU eval	CPU setup mod.	CPU eval mod.	GPU setup	GPU eval
602	8.39E-03	2.43E-02	5.39E-03	2.45E-02	1.20E-02	6.69E-03
1562	1.29E-02	6.61E-02	1.26E-02	6.18E-02	1.59E-02	7.07E-03
3542	3.01E-02	1.78E-01	2.42E-02	1.66E-01	2.61E-02	7.83E-03
9902	7.62E-02	7.58E-01	5.91E-02	6.99E-01	6.27E-02	1.17E-02
39802	4.63E-01	4.71E+00	3.55E-01	4.35E+00	3.76E-01	6.82E-02
89702	1.23E+00	1.79E+01	8.14E-01	1.50E+01	8.25E-01	1.39E-01

Table 3: Setup and evaluation of multipole multiplication with single precision

# bc	CPU setup	CPU eval	CPU setup mod.	CPU eval mod.	GPU setup	GPU eval
602	8.38E-03	2.74E-02	7.57E-03	2.56E-02	1.80E-02	1.37E-02
1562	1.49E-02	7.08E-02	1.41E-02	6.61E-02	2.39E-02	1.51E-02
3542	3.06E-02	1.87E-01	2.43E-02	1.79E-01	3.58E-02	1.85E-02
9902	7.98E-02	7.62E-01	6.38E-02	7.39E-01	7.46E-02	3.78E-02
39802	4.55E-01	4.51E+00	3.69E-01	4.37E+00	4.00E-01	2.18E-01
89702	1.32E+00	1.75E+01	1.08E+00	1.71E+01	8.61E-01	5.71E-01

Table 4: Setup and evaluation of multipole multiplication with double precision

4.4 Test system IV

The test system is equipped with two Intel Xeon X5670 processors with 6 cores each. The OpenMP parallelized CPU code used 12 threads. The GPU used for the computations is a Nvidia GTX 660Ti with compute capability 3.0. The used compiler is `gcc` version 4.1.2., the compiler options were the same as on the other test systems.

Figure 15 shows the calculation time for one directly calculated matrix-vector product in double precision over the number of boundary condition nodes. The performed calculations and the calculation times using single and double precision are listed in table 5.

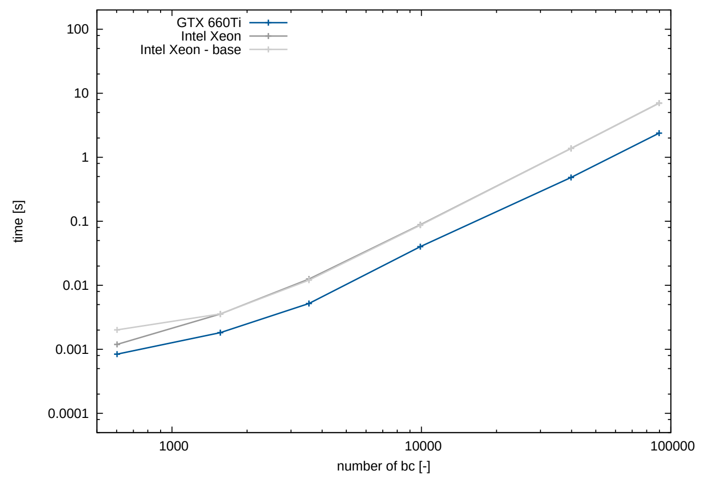


Figure 15: Brute force double precision

# bc	CPU original [s]	CPU no cond. [s]	GPU [s]	# bc	CPU original [s]	CPU no cond. [s]	GPU [s]
602	1.78E-03	1.15E-03	2.90E-04	602	2.01E-03	1.19E-03	8.39E-04
1562	2.01E-03	2.74E-03	4.08E-04	1562	3.58E-03	3.56E-03	1.82E-03
3542	7.25E-03	7.11E-03	6.01E-04	3542	1.20E-02	1.25E-02	5.19E-03
9902	5.13E-02	5.18E-02	2.16E-03	9902	8.67E-02	8.81E-02	4.02E-02
39802	8.04E-01	7.94E-01	2.23E-02	39802	1.37E+00	1.38E+00	4.83E-01
89702	4.13E+00	4.04E+00	1.08E-01	89702	7.01E+00	7.01E+00	2.38E+00

Table 5: Brute force multiplication with single and double precision

Figure 16 shows the calculation time for one multipole approximated matrix-vector product in double precision over the number of boundary condition nodes. The performed calculations and the calculation times using single precision are listed in table 6, the calculation times using double precision are listed in table 7.

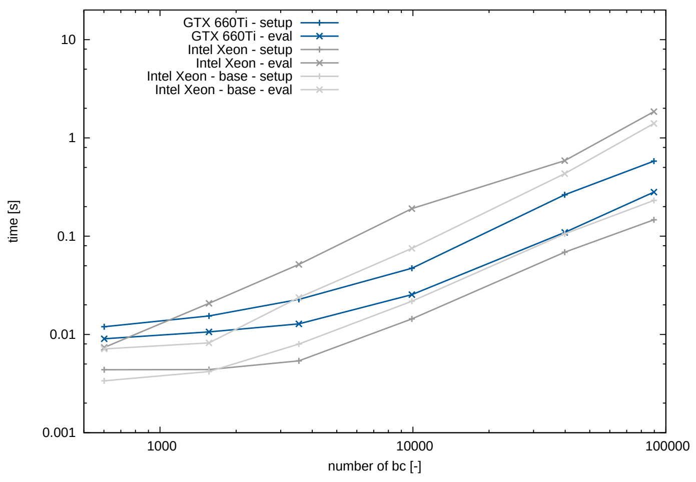


Figure 16: Multipole double precision

# bc	CPU setup	CPU eval	CPU setup mod.	CPU eval mod.	GPU setup	GPU eval
602	1.83E-03	7.01E-03	1.85E-03	8.12E-03	1.09E-02	4.83E-03
1562	2.89E-03	8.83E-03	2.46E-03	2.21E-02	1.44E-02	5.17E-03
3542	6.19E-03	2.68E-02	5.21E-03	5.99E-02	2.18E-02	5.82E-03
9902	1.63E-02	9.16E-02	1.40E-02	2.42E-01	4.46E-02	7.38E-03
39802	8.40E-02	5.39E-01	7.08E-02	7.86E-01	2.28E-01	3.41E-02
89702	1.83E-01	1.85E+00	1.53E-01	2.66E+00	4.92E-01	6.79E-02

Table 6: Setup and evaluation of multipole multiplication with single precision

# bc	CPU setup	CPU eval	CPU setup mod.	CPU eval mod.	GPU setup	GPU eval
602	3.37E-03	7.13E-03	4.38E-03	7.38E-03	1.20E-02	9.04E-03
1562	4.19E-03	8.21E-03	4.39E-03	2.07E-02	1.54E-02	1.06E-02
3542	7.99E-03	2.38E-02	5.38E-03	5.16E-02	2.27E-02	1.28E-02
9902	2.18E-02	7.52E-02	1.44E-02	1.91E-01	4.71E-02	2.54E-02
39802	1.06E-01	4.33E-01	6.86E-02	5.87E-01	2.64E-01	1.09E-01
89702	2.31E-01	1.40E+00	1.47E-01	1.85E+00	5.80E-01	2.81E-01

Table 7: Setup and evaluation of multipole multiplication with double precision

4.5 Test system V

The test system is equipped with one Intel Core i7 920 processor with 4 cores. The OpenMP parallelized CPU code used 8 threads due to active hyper threading. The GPU used for the computations is a Nvidia GTX 480 with compute capability 2.0. The target architecture for the GPU code was adjusted to `-arch sm_20`.

The table 8 shows the measured times for the directly computed matrix-vector products in single and double precision respectively.

# bc	CPU original [s]	CPU no cond. [s]	GPU [s]	# bc	CPU original [s]	CPU no cond. [s]	GPU [s]
602	1.03E-03	2.46E-04	1.12E-04	602	1.20E-03	5.76E-04	5.59E-04
1562	4.29E-03	1.20E-03	2.37E-04	1562	7.30E-03	3.71E-03	1.39E-03
3542	2.14E-02	5.83E-03	7.99E-04	3542	3.71E-02	1.86E-02	3.13E-03
9902	1.66E-01	4.17E-02	3.34E-03	9902	2.89E-01	1.44E-01	2.52E-02
39802	2.68E+00	6.72E-01	4.65E-02	39802	4.66E+00	2.33E+00	3.68E-01
89702	1.36E+01	3.42E+00	2.25E-01	89702	2.37E+01	1.18E+01	1.81E+00

Table 8: Brute force multiplication with single and double precision

The tables 9 and 10 show the measured times for the multipole approximated matrix-vector products in single and double precision respectively.

# bc	CPU setup	CPU eval	CPU setup mod.	CPU eval mod.	GPU setup	GPU eval
602	3.52E-03	9.63E-03	2.52E-03	7.61E-03	6.67E-03	4.44E-03
1562	7.48E-03	1.74E-02	4.40E-03	1.86E-02	8.73E-03	4.63E-03
3542	1.45E-02	4.08E-02	1.02E-02	7.43E-02	1.30E-02	4.83E-03
9902	4.09E-02	1.43E-01	2.38E-02	1.58E-01	2.57E-02	8.26E-03
39802	1.94E-01	9.64E-01	1.23E-01	8.73E-01	1.48E-01	4.87E-02
89702	4.34E-01	2.75E+00	2.88E-01	2.40E+00	3.05E-01	9.50E-02

Table 9: Setup and evaluation of multipole multiplication with single precision

# bc	CPU setup	CPU eval	CPU setup mod.	CPU eval mod.	GPU setup	GPU eval
602	6.23E-03	1.05E-02	4.74E-03	7.34E-03	6.59E-03	5.82E-03
1562	7.46E-03	2.02E-02	5.83E-03	1.91E-02	8.45E-03	6.31E-03
3542	1.69E-02	4.91E-02	9.67E-03	5.87E-02	1.15E-02	7.31E-03
9902	3.95E-02	1.80E-01	2.31E-02	2.14E-01	2.40E-02	1.40E-02
39802	2.01E-01	1.05E+00	1.42E-01	9.57E-01	1.24E-01	7.90E-02
89702	4.48E-01	3.44E+00	3.22E-01	3.12E+00	2.68E-01	1.90E-01

Table 10: Setup and evaluation of multipole multiplication with double precision

4.6 Test system VI

The test system is equipped with one AMD Phenom 9950 processor with 4 cores. The OpenMP parallelized CPU code used 4 threads. The GPU used for the computations is a Nvidia GTX 280 with compute capability 1.3. The target architecture for the GPU code was adjusted to `-arch sm_13`.

The table 11 shows the measured times for the directly computed matrix-vector products in single and double precision respectively.

# bc	CPU original [s]	CPU no cond. [s]	GPU [s]	# bc	CPU original [s]	CPU no cond. [s]	GPU [s]
602	8.96E-04	4.06E-04	1.87E-04	602	1.42E-03	8.24E-04	1.52E-03
1562	6.72E-03	2.49E-03	4.22E-04	1562	9.54E-03	6.27E-03	3.85E-03
3542	2.43E-02	1.38E-02	9.06E-04	3542	3.34E-02	2.35E-02	8.67E-03
9902	1.70E-01	7.83E-02	4.75E-03	9902	2.33E-01	1.56E-01	6.88E-02
39802	2.72E+00	1.23E+00	6.84E-02	39802	3.73E+00	2.42E+00	9.96E-01
89702	1.51E+01	6.21E+00	3.36E-01	89702	3.55E+01	2.09E+01	4.88E+00

Table 11: Brute force multiplication with single and double precision

The tables 12 and 13 show the measured times for the multipole approximated matrix-vector products in single and double precision respectively.

# bc	CPU setup	CPU eval	CPU setup mod.	CPU eval mod.	GPU setup	GPU eval
602	7.94E-03	1.79E-02	4.33E-03	1.77E-02	1.09E-02	9.95E-03
1562	1.01E-02	3.56E-02	8.33E-03	3.46E-02	1.18E-02	9.99E-03
3542	1.75E-02	8.27E-02	1.04E-02	7.99E-02	1.73E-02	1.19E-02
9902	4.76E-02	2.73E-01	2.77E-02	2.46E-01	3.70E-02	1.96E-02
39802	2.88E-01	1.83E+00	1.69E-01	1.71E+00	2.08E-01	1.05E-01
89702	6.42E-01	4.94E+00	3.68E-01	4.44E+00	4.55E-01	2.20E-01

Table 12: Setup and evaluation of multipole multiplication with single precision

# bc	CPU setup	CPU eval	CPU setup mod.	CPU eval mod.	GPU setup	GPU eval
602	7.49E-03	2.09E-02	5.91E-03	1.81E-02	1.12E-02	1.90E-02
1562	1.16E-02	3.63E-02	1.09E-02	3.94E-02	1.30E-02	2.11E-02
3542	2.18E-02	8.79E-02	1.64E-02	8.24E-02	1.93E-02	2.75E-02
9902	5.56E-02	2.95E-01	3.52E-02	2.69E-01	4.39E-02	5.66E-02
39802	2.97E-01	1.89E+00	1.82E-01	1.75E+00	2.77E-01	3.16E-01
89702	6.51E-01	5.55E+00	3.97E-01	4.82E+00	6.01E-01	7.64E-01

Table 13: Setup and evaluation of multipole multiplication with double precision

5 Performance comparison

This section provides a comparison of the test results among the different test systems. The theoretical peak performance is calculated from the number of cores times core frequency times the number of foating point operations per core per cycle based on the hardware manufacturer specifications. For the comparison among the different test systems the theoretical peak performances for single and double precision arithmetics are compared to the theoretical peak performances of test system I featuring a Nvidia GTX 680 graphics card.

$$\text{relative peak performance (sp or dp)} = \frac{\text{theoretical peak performance (sp or dp)}}{\text{theoretical peak performance of test system I (sp or dp)}}$$

In addition to the theoretical peak performances the runtimes of the test case with 89702 boundary condition nodes are compared to the runtime on test system I.

$$\text{relative runtime (sp or dp)} = \frac{\text{runtime on test system I (sp or dp)}}{\text{runtime (sp or dp)}}$$

The comparison graphs also feature the relative memory bandwith of the test system, again compared to the memory bandwith of test system I.

Figures 17 and 18 show the results for the brute force calculation of the matrix-vector product, the evaluation stage and the setup stage of the multipole approximation of the matrix-vector product among the test systems I, III, IV, V and VI. The results for test system II will be listed below.

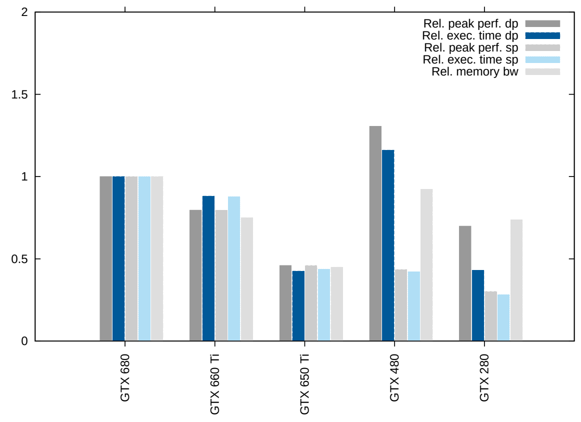
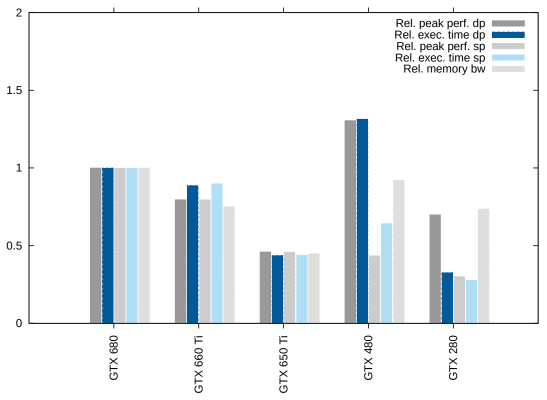


Figure 17: Relative performance - brute force multiplication

The relative performance of the brute force multiplication and the evaluation phase of the multipole approximation conform with the relative theoretical peak performances among the test systems. The relative performance of the setup stage of the multipole approximation does not conform with the relative theoretical peak performances.



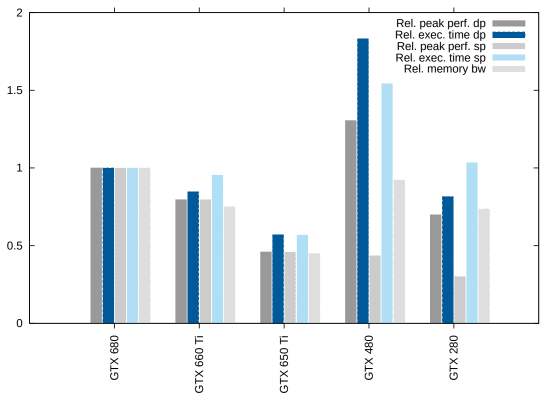


Figure 18: Relative performance - multipole approximated multiplication - evaluation and setup stage

Figures 19 and 20 show the results for the brute force calculation of the matrix-vector product, the evaluation stage and the setup stage of the multipole approximation of the matrix-vector product for the test systems I and II. The GPU of test system II is a product of the high performance computing series of Nvidia and features a high theoretical peak performance for double precision arithmetics.

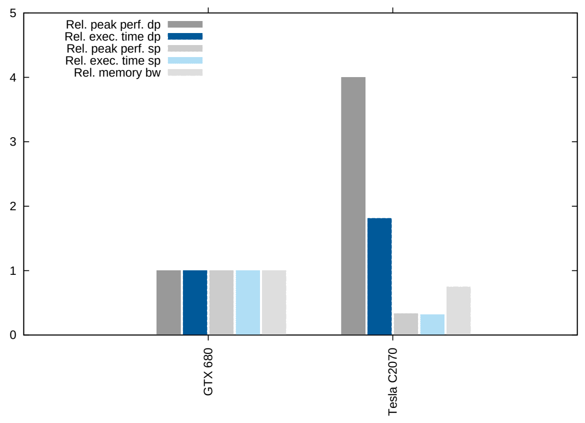


Figure 19: Relative performance - brute force multiplication

The measured relative performance does not conform with the relative theoretical peak performance for double precision arithmetics. The code has not been optimized for test system II, but the same block sizes and number of threads per block as on test system I were used.

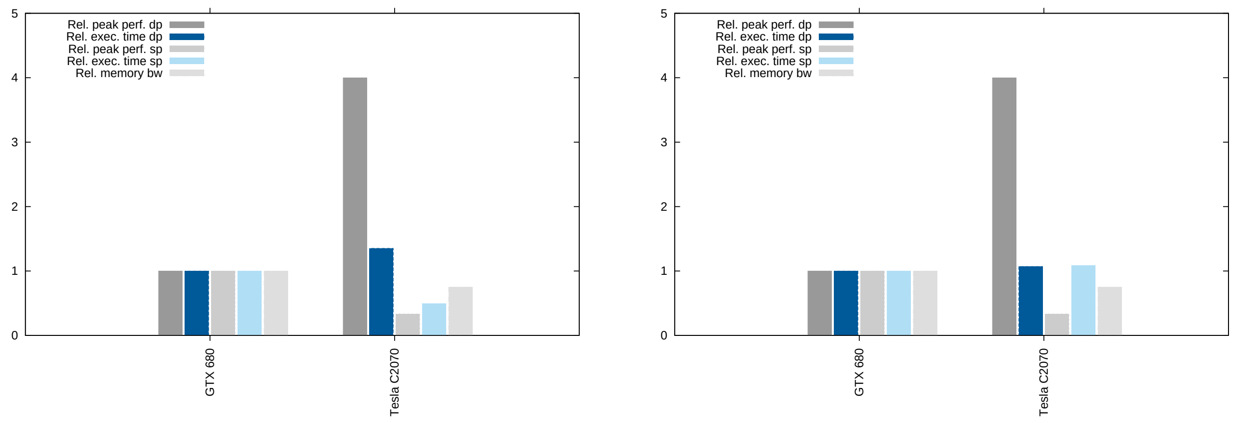


Figure 20: Relative performance - multipole approximated multiplication - evaluation and setup stage

6 Conclusion

6.1 Results

The project shows that the used methods can be successfully transfered on GPGPUs both - if the GPU features the required compute capability - using single and double precision arithmetics. Some parts of the code prove sensitive to cancellation errors if single precision arithmetics are used.
For directly computed matrix-vector products the code shows good performances running on the GPU. The achieved speedups vary due to the different computational powers of the test systems. The memory bandwith can be neglected,i.e. we are limited by the arithmetic capabilities.
The code compiled with the Intel C++ compiler showed no significant deviation in the overall runtime.
The results that were achieved so far indicate that the performance on high performance computing systems can still be improved in case of the application of double precision arithmetics.
The matrix-vector products approximated using a multipole method show different behaviours on the CPU and the GPU. The calculation time spent in the setup phase is smaller than the time spent in the evaluation phase on the CPU. This changes on the GPU. There more time is spent in the setup phase due to the high number of accesses to the global memory. As a result the best performances can be achieved either performing both stages on the GPU or performing the setup phase on the CPU and the evaluation phase on the GPU depending in the computational powers of the host CPU and the used GPU - as well as the used arithmetic precision.

6.2 Open issues

- A value that proves critical for the applicability of the method is the scaling parameter c of the multiquadric basis function. It is to be investigated if a locally adjusted scaling parameter can be used.
- Adapting the number of terms in the Laurent series approximation to the distance of the octree box to the evaluation point including a profound error estimation.
- How the usage of single precision arithmetics can be secured or at least how a failure can be detected reliably.
- Optimization of the code for high performance computing GPU systems.
- The application of the method on even bigger test cases.

Acknowledgements

This work has been partially supported by Austrian Science Fund (FWF): F 3201-N18.
Part of this work was done in the scope of the K1 program at the VRVis Research Center (`http://www.VRVis.at`).

References

[1] R. K. Beatson and L. Greengard. A short course on fast multipole methods. In *Wavelets, Multilevel Methods and Elliptic PDEs*, pages 1–37. Oxford University Press, 1997.

[2] M. D. Buhmann. *Radial basis functions: theory and implementations.* Cambridge monographs on applied and computational mathematics. Cambridge University Press, 2003.

[3] C. S. Chen, Y. C. Hon, and R. A. Schaback. *Scientific computing with radial basis functions.* Georg-August-Universität zu Göttingen, 2003. Unpublished manuscript.

[4] J. B. Cherrie, R. K. Beatson, and G. N. Newsam. Fast evaluation of radial basis functions: Methods for generalized multiquadrics in rn. *SIAM J. Sci. Comput.*, 23(5):1549–1571, 2002.

[5] A. de Boer, M. S. van der Schoot, and H. Bijl. Mesh deformation based on radial basis function interpolation. *Comput. Struct.*, 85:784–795, June 2007.

[6] A. C. Faul, G. Goodsell, and M. J. D. Powell. A Krylov subspace algorithm for multiquadric interpolation in many dimensions. *IMA J. Num. Anal.*, 25(1):1–24, 2005.

[7] A. C. Faul and M. J. D. Powell. Proof of convergence of an iterative technique for thin plate spline interpolation in two dimensions. *Adv. Comp. Math.*, 11:183–192, 1999.

[8] A. C. Faul and M. J. D. Powell. Krylov subspace methods for radial basis function interpolation. In D. F. Griffiths, editor, *Numerical Analysis 1999*, page 115–141. Chapman and Hall, 2000.

[9] H. Wendland. *Scattered Data Approximation.* Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2010.