Detection of defected pills during manufacturing: optimization and implementational aspects on GPUs

G. Haase A. Kucher

SFB-Report No. 2010–028 August 2010

Supported by the
Austrian Science Fund (FWF)

Der Wissenschaftsfonds.

SFB sponsors:

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

Noname manuscript No.
(will be inserted by the editor)

Detection of defected pills during manufacturing: optimization and implementational aspects on GPUs

Gundolf Haase · Andreas Kucher

Sept. 13, 2010

Abstract This paper describes the fast minimization of a functional for a best curve approximation of data resulting from a pill identification problem. The curve approximation in $\mathbb{R}^N$ uses polynomial curves as well as circular splines. Functional evaluations have been faster on the CPU with circular splines but the situation changes on the GPU in favor of polynomial curves. We used a line search with a fixed step size and a fixed reduction factor of it in the Quasi–Newton iteration. This naive step size control reduces code branches in the implemented optimization algorithm and yields a performance gain between 65 and 194 on the GPU in comparison to the CPU implementation. The MPI–parallelization of the code shows additional good speedup results.

1 Introduction

The testing of pills with respect to the amount of contained active ingredients is usually performed stochastically in the production process, i.e., a representative number of pills is examined. Nevertheless, some uncertainty remains. A non-destructive and fast testing of all pills during the production process would be preferable and such a method has been described in [2–4]. This new approach uses spectral analysis to identify certain active pharmaceutical ingredients and their concentration and it offers many advantages compared to testing a few pills only when stochastic method are used. However, since it is not possible to test against the whole frequency spectrum, the N frequencies, which can identify an ingredient best, have to be found. Finding these N frequencies bases on criteria whose characterize *good* frequencies. This characterization will be done by a special curve approximation of the response intensities of the different frequencies with respect to the given probes. The special curve approximation

This publication is based on work supported in part by the Austrian Science Fund FWF under grant SFB F032

G. Haase and A. Kucher
Institute for Mathematics and Scientific Computing, University of Graz
Heinrichstr. 36, 8061 Graz, Austria
Tel.: +43-316-380-5178
Fax: +43-316-380-9815

will be done by minimizing a functional for each data set. Although numerous data sets have to be investigated, these calculations have to be performed only once for one type of pill. The big advantage of this approach is that for each ingredient, the best N frequencies have to be found only once and the actual testing of pills in production itself can be performed in real time.

The increasing success of general purpose GPUs in general computing and also in high performance computing is unquestioned. In this technical report we will examine the suitability of GPUs in the pill identification problem above, which might be considered as a representative example for parallelization of many similar sequential optimization problems on many–core processors. For this purpose we will examine a subproblem on a GPU and then compare the performance of the parallel pill identification algorithm implemented as C++ CPU-only version using openMPI[1] and as C++ CPU/GPU version using openMPI and CUDA[2]. It turned out, that the CPU/GPU approach, under some restrictions, is much faster than the CPU-only approach. The remaining paper is organized as follows. Notations, functional and basic relations are introduced in §2 followed by implementational aspects of them in §3 and §4. Specifications of the optimization algorithm can be found in §5 while §6 focuses on data and code layout for the GPU implementation. The paper finishes with run time behavior of the algorithms and conclusions.

2 Mathematical description

2.1 The general algorithm

If we want to use N frequencies for pill identification then we have to find the best frequency N–tuple. In order to do this we first have to be able to determine the "identification ability" of a tuple. For this take an N–tuple of frequencies $\{v_i\}_{i=1}^N \in \mathbb{R}^N$ and test them with m strictly increasing concentrations $\{c_i\}_{i=1}^m \in \mathbb{R}^m$ and get m response vectors $\{f_i\}_{i=1}^m$, with $f_i \in \mathbb{R}^N$.

Now a sufficiently smooth parameterized curve $q(\cdot)$, $q : \mathbb{R} \to \mathbb{R}^N$ approximates the data points (respectively the response vectors $\{f_i\}_{i=1}^m$) and projects each f_i, $\forall i = 1, \ldots, m$ orthogonally onto $q(\cdot)$ at parameter t_i with the restriction that $t_i < t_{i+1}$ and determines the arc length s_i along the curve from t_1 to t_i afterwards.

The ideal curve $q(\cdot)$ would result in an affine linear relation between arc lengths s_i and concentrations c_i, $\forall i = 1, \ldots, m$. The distances of points (s_i, c_i) from the regression line of these points determines the quality of the approximation. A pseudo algorithm for this quality criterion looks like:

1. for $i = 1, \ldots, m$: project f_i onto $q(t)$, such that for t_i the value $q(t_i)$ is the closest orthogonal projection point of f_i on the curve $q(t)$
2. for $i = 1, \ldots, m$: calculate the arc length s_i along the curve $q(\cdot)$ from t_1 to t_i
3. determine the parameters a, b of the regression line
$$g(s) = as + b \tag{1}$$
from the data pairs $\{(s_i, c_i)\}_{i=1}^m$

[1] Open source MPI-2 implementation, `http://www.open-mpi.org`

[2] C programming environment for general purpose GPUs, `http://www.nvidia.com`

4. evaluate the quality of the curve $q(\cdot)$ according to the functional

$$F(q,f) := \omega_1 \sum_{i=1}^{m} (g(s_i) - c_i)^2 + \omega_2 \sum_{i=1}^{m} \|q(t_i) - f_i\|^2 + \omega_3 \sum_{i=1}^{m-1} \max(0, t_i - t_{i+1})^2 \quad (2)$$

   where ω_1, ω_2 and ω_3 are freely chosen parameters with the restriction that $\omega_3 \gg \omega_1, \omega_2$. The idea of this restriction is that the case $t_i > t_{i+1}$, $i = 1, \dots, m-1$ is to be penalized.

Since F is a convex functional and sufficiently regular, we can find (one of) the best curves for the particular set (v, f, c) in a class (e.g. polynomials of degree $p \in \mathbb{N}$ or circular splines [10]), by optimizing the curves parameters with respect to F.

2.2 Projection

Because $q(t) \in C^1(\mathbb{R}^N)$ the tangential vector of $q(t)$ at parameter t^* is $q\prime(t^*) = \left(q^{(1)\prime}(t^*), \dots, q^{(N)\prime}(t^*)\right)^T$ wherein $q^{(k)\prime}$ denotes the first derivative of $q^{(k)}$ with respect to t. Therefore the projection of the point $\underline{f} \in \mathbb{R}^N$ onto the curve $q(t)$ can be expressed in terms of the inner product $\langle \cdot, \cdot \rangle$ in the Hilbert space $\mathbb{R}^N$

$$\text{Find } t^* \in \mathbb{R} \text{ such that } p(t) := \left\langle q\prime(t^*) \, , \, \underline{f} - q(t^*) \right\rangle = 0 \ . \quad (3)$$

Providing the first derivative of $p(t)$

$$p\prime(t) = \left\langle \underline{f} - q(t) \, , \, q\prime\prime(t) \right\rangle \ - \ \langle q\prime(t) \, , \, q\prime(t) \rangle \quad (4)$$

allows to solve the non–linear equation $p(t) = 0$ via the Newton iteration

$$t^{(l+1)} := t^{(l)} - \frac{p(t)}{p\prime(t)} \ . \quad (5)$$

with an appropriate initial guess $t^{(0)}$. We have to be aware that (3) may have non-unique solutions. Therefore this initial guess for the non–linear solution procedure is of great importance.

Solving (3) for all $\underline{f}_i$ will determine the t_i, $i = 1, \dots m$ from step 1 in the algorithm from §2.1.

2.3 Arc length

The formula for the arc length is simply

$$s(t^*) \ := \ \int_{t_1}^{t^*} \left(\sum_{k=1}^{N} [q^{(k)\prime}(t)]^2 \right)^{0.5} dt \ . \quad (6)$$

A arc–length parameterized curve, i.e. using s instead of t as parameter, is rather hard to achieve and can be done only numerically for the general case. An approach for spline curves is described in [11].

2.4 Regression line

Determining the a, b in the regression line (1) from the given data pairs $\{(s_i, c_i)\}_{i=1}^m \in \mathbb{R}^2$ is equivalent to the minimizing the functional

$$\widetilde{F}(a,b) := \sum_{i=1}^{m} (g(s_i) - c_i)^2 \overset{(1)}{=} \sum_{i=1}^{m} (a \cdot s_i + b - c_i)^2 \tag{7}$$

with respect to the parameters a, b. Note, that the linear functional $\widetilde{F}$ in (7) is similar (not identical) to the non–linear functional F in (2).

Some simple numerical analysis solves (7) as

$$\begin{pmatrix} a \\ b \end{pmatrix} = \frac{1}{m \sum\limits_{i=1}^{m} s_i^2 - \left(\sum\limits_{i=1}^{m} s_i\right)^2} \begin{pmatrix} m \sum\limits_{i=1}^{m} s_i c_i & - \left(\sum\limits_{i=1}^{m} s_i\right)\left(\sum\limits_{i=1}^{m} c_i\right) \\ \left(\sum\limits_{i=1}^{m} s_i^2\right)\left(\sum\limits_{i=1}^{m} c_i\right) & - \left(\sum\limits_{i=1}^{m} s_i c_i\right)\left(\sum\limits_{i=1}^{m} s_i\right) \end{pmatrix} \tag{8}$$

2.5 Parameterization of the polynomial curve

Let us assume that the functions $q^{(k)}(t)$ are global polynomial functions of degree p, i.e.

$$q^{(k)}(t) := \sum_{j=0}^{p} a_{kj} \cdot t^j \qquad \forall k = 1, \ldots, N \ . \tag{9}$$

If we use piecewise polynomial function, e.g., cubic splines, then we take into account a similar representation of the curve in which the coefficients a_{kj} will characterize the curve $q(t) \in \mathbb{R}^N$. These coefficients are functions of the given intensities $\underline{f}_i \in \mathbb{R}^N$. The coefficients a_{kj} are unique under the presupposition $q(t = 0) \equiv \underline{f}_1$.
The first derivative of $q^{(k)}(t)$ is

$$q^{(k)\prime}(t) := \sum_{j=0}^{p-1} a_{k,j+1}(j+1) \cdot t^j \qquad \forall k = 1, \ldots, N \ . \tag{10}$$

2.6 Circular splines

A circular spline $q\,(t)$ is a parametrized spline curve in $\mathbb{R}^N$, which segments consist of n circular arcs y_i, $i = 1, \ldots, n$ [10].

Each circular arc has a rational Bezier-representation

$$y_i\,(u) = \frac{(1-u)^2\, A + 2u\,(1-u)\,\omega B + u^2 C}{(1-u)^2 + 2u\,(1-u)\,\omega + u^2} \qquad u \in [0,1] \ . \tag{11}$$

Hence, $q\,(t) = y_k\,(\tilde{u})$ with $k = \lfloor t \rfloor$ and $\tilde{u} = t - k$.
Figures 1 indicates, that A, B, C and $\omega = \cos\varphi$ can be easily computed from M, the two shape parameters h^2 and k and the rotation of the arc.
In order to achieve C^1-regularity, we first have to fix the arcs y_1 and y_4. This allows to determine y_2 and y_3 such that this piece of the curve is continuously differentiable. Following this procedure iteratively (by fixing y_7, y_{10}, …), one can construct a continuously differentiable circular spline.

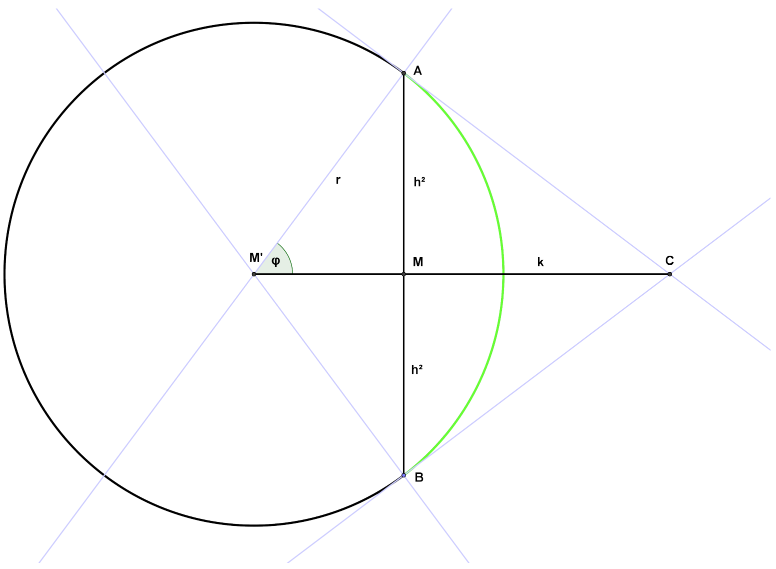

Fig. 1 Geometric description of a circular arc in $\mathbb{R}^2$

2.6.1 Projection

$u^* \in [0,1]$ is the parameter of a projection point of $\underline{f} \in \mathbb{R}^N$ on a circular arc y if the relation in (3) is fulfilled for u^*, i.e.,

$$\left\langle y'\left(u^*\right), \underline{f} - \frac{\left(1-u^*\right)^2 A + 2u^*\left(1-u^*\right)\omega B + {u^*}^2 C}{\left(1-u^*\right)^2 + 2u^*\left(1-u^*\right)\omega + {u^*}^2}\right\rangle = 0 \tag{12}$$

where

$$y'\left(u\right) = -2\frac{u^2\left(\omega-1\right)\left(A-B\right) - 2u\omega\left(A-C\right) + u\left(A-B\right) + \omega\left(A-C\right)}{\left(2u^2\left(\omega-1\right) - 2u\left(\omega-1\right) - 1\right)^2} \tag{13}$$

This means that u^* may be computed directly without using iterative methods, since solving (12) for u^* is equivalent to find a real zero of a polynomial with degree 4 in the interval $[0,1]$.

2.6.2 Arc length

The full arc–length l_y of a circular arc y may be computed by $l_y = 2r\varphi$ where

$$\varphi = \cos^{-1}\left(\frac{h^2}{\sqrt{h^4+k^2}}\right) \text{ and } r = \left|\frac{h^2}{\cos\left(\frac{\pi}{2}-\varphi\right)}\right| \tag{14}$$

Analogously for a parameter u the arc–length $s_y\left(u\right)$ from the point A to $y\left(u\right)$ may be calculated by determining $\psi = \sphericalangle\left(A, M', y\left(u\right)\right)$. Therefore the arc–length $s\left(t^*\right)$ of a circular spline at parameter t^*, with $t = k + u^*$ $(k \in \mathbb{N}, u^* \in [0,1])$ is

$$\sum_{i=1}^{k} l_{y_i} + s_{y_i}\left(u^*\right) \tag{15}$$

2.7 Determining the best curve

Formally, determining the best curve is equivalent to minimizing functional F from (2). The functional F is a function of the regression parameters a, b which are functions of all s_i, c_i. The s_i are determined from $q^{(k)}, t_i$ and the latter is a function of curve $q^{(k)}$ and intensities $\underline{f}_i$. The parametrization of $q^{(k)}$ involves finally the curve parameters a_{kj} from §2.5, i.e., we consider only the polynomial curves approach in the following. The formulas for circular splines will be similar using the appropriate curve parameters for them.

$$\begin{aligned}
F &:= \sum_{i=1}^{m} (g(s_i) - c_i)^2 \\
&\qquad\text{regression line} \\
&= F(a(\{s_i\}, \{c_i\}),\, b(\{s_i\}, \{c_i\})) \\
&\qquad\text{arc length} \\
&= F(a\left(\left\{ s_i \left[\{q^{(k)}\},\, t_i \right] \right\} ,\, \{c_i\} \right), b(\cdots)) \\
&\qquad\text{projection} \\
&= F(a\left(\left\{ s_i \left[\{q^{(k)}\},\, t_i \left(\{q^{(k)}\},\, \underline{f}_i \right) \right] \right\} ,\, \{c_i\} \right), b(\cdots)) \\
&\qquad\text{parameterization} \\
F(\{a_{kj}\}, \{f_i^{(k)}\}, \{c_i\}) &= F(a\left(\left\{ s_i \left[\{a_{kj}\},\, t_i \left(\{a_{kj}\},\, \underline{f}_i \right) \right] \right\} ,\, \{c_i\} \right), b(\cdots))
\end{aligned}$$

Determining the best curve for an N-tuple of frequencies means solving the optimization problem

$$\min_{a_{kj}} F(\{a_{kj}\}, \{\underline{f}_i\}, \{c_i\}) \qquad \forall i = 1, \ldots, m \tag{16}$$

which can be expressed as non–linear equation

$$\nabla_{a_{kj}} F(\{a_{kj}\}, \{\underline{f}_i\}, \{c_i\}) = 0 \tag{17}$$

with respect to the design variables a_{kj}, $k = 1, \ldots, N$, $j = 0, \ldots, p$.

Solving (17) directly cannot be done analytically because of the non–linear equation (3) the gradient cannot be provided analytically. Instead, we tackle the optimization problem (16) with standard SQP methods [6,8]. The optimizer is based on a Quasi–Newton approximation of the Hessian using a modified BFGS update formula following [9] in order to avoid the need for Hessian information of the objective, see §5 for implementational details.

Figures 2 and 3 illustrate the optimization process with a polynomial of degree 3 from §2.5 and a circular spline from §2.6, respectively. The initial curves, generated by heuristic approximation with respect to $\{f_i\}_{i=1}^{m}$, have a poor quality. The other curves are generated by optimizing the curves parameters with respect to the functional F.

3 General implementational aspects

3.1 The parallel algorithm

Since there is a huge amount of test data, i.e. many frequencies in the spectrum that can be evaluated for their suitability to identify ingredients and only the best ones should

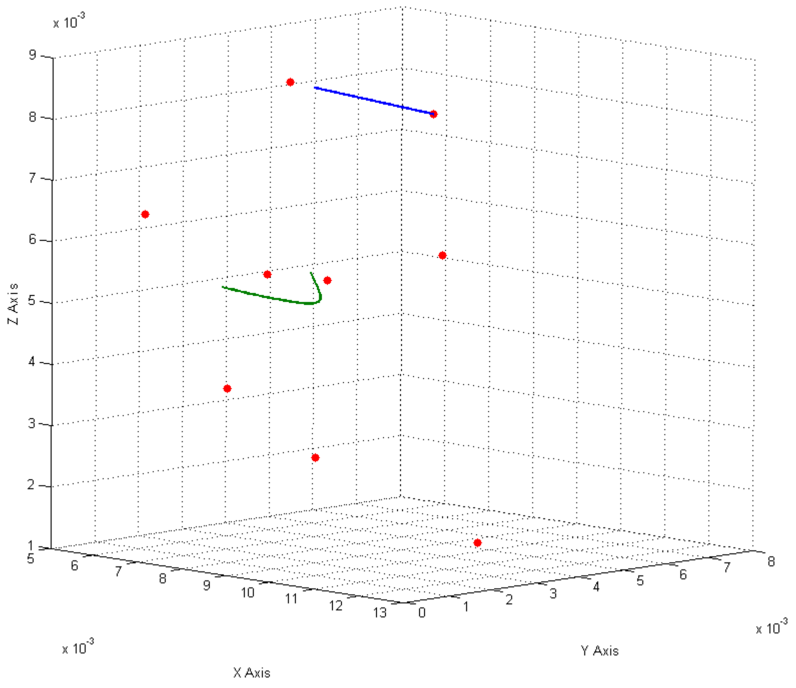

Fig. 2 Polynomial curve before (blue) and after (green) optimization with respect to the given datapoints (red)

be used, it is necessary to execute the algorithm above with each set of available test data. Basically, finding the best curve for a frequency tuple (and hence evaluate the quality of this tuple) is a sequential problem for one CPU, respectively for one CPU core. Therefore the parallelization approach is to have a distinguished master process and many slave processes.
The master process takes care of the data handling (supplies all other processes with data sets) and also keeps track of the quality of each dataset, while the slave processes receive one or multiple data set from the master, find the best curve for the data sets (by performing the algorithm above), send the result back to the master process and evaluate the new data sets received from the master.

3.2 Implementation steps

First the functional evaluation was implemented for two classes of curves on a CPU with double precision. The first class were polynomials of degree n, where polynomials of degree higher than 3 prove to be numerically unstable at the projection process. Since there is neither a closed formula for either the projection of the intensity points, nor the calculation of arc length in general but only for special cases [1], iterative approximation was used. To avoid these very computational tasks, we also implemented circular splines with double precision for comparison, because closed formulas for projection and arc

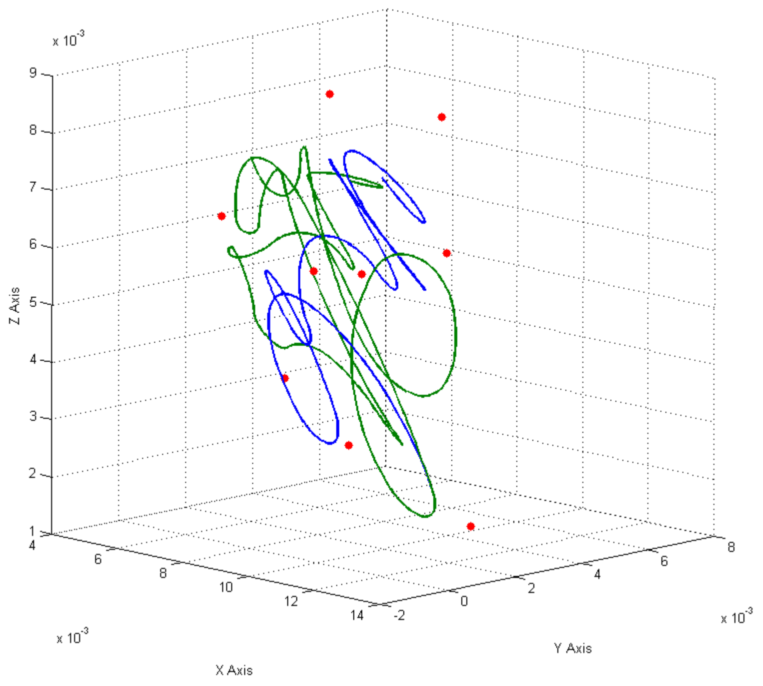

Fig. 3 Circular spline curve before (blue) and after (green) optimization with respect to the given datapoints (red)

length computation can be derived easily. Then the functional evaluation was ported on GPU using CUDA and single precision, where special care for GPU characteristic programming and optimization strategies was taken.

The next step was to implement the optimization for the class of polynomials. For this an intelligent double precision implementation of a Quasi–Newton [8], [5, pp.138] optimization algorithm with BFGS-Update [9] and common line search strategies implemented by Ewald Lindner[3] was used [7]. Due to the enormous amount of divergent branches and a poor single precision behavior of the algorithm, and hence the lack of suitability for a GPU implementation we had to use a simpler algorithm. Several more primitive optimization approaches as the method of steepest descent or conjugate gradients for non–quadratic problems with and without "proper" step size control were tested and finally a primitive Quasi–Newton algorithm with an Inverse-BFGS-Update and backtracking in the line search was first implemented for CPU and then ported for execution on a GPU. For backtracking a fixed step length α in the descent direction d is reduced by a fixed factor, until any descent is achieved. In the step size control any step length is valid, as long as there is any descent with respect to a step length of 0. This approach does not guarantee any convergence rates, but prove to be suitable for F.

Finally four parallel versions using openMPI were implemented, i.e., a CPU-only double precision version using Ewald Lindner's implementation, a CPU-only version

[3] University of Linz, `http://www.numa.uni-linz.ac.at/~lindner/`

using the own primitive optimization algorithm, both with double precision and finally a CPU/GPU single precision and a CPU/GPU double precision version, also using the own optimization algorithm.

3.3 System overview

Let us introduce the main components of our implementation. Each implemented optimization approach uses the classes described below (except for InterpolatingCurve), although there are slight differences in their implementation.

– *DataDistributor*: The DataDistributor class is meant to be used as singleton, although several instances could be initialized. It manages all data sets using the DataWrapper class, takes care of the data distribution and keeps track of the result by using OptimizedCurve.
– *DataWrapper*: The DataWrapper class is a wrapper class around the available data and provides a convenient interface to it. The data is received from a hard disk file. Since there may be a huge amount of data sets, not all data sets are loaded into system memory at once, but are buffered.
– *DataProcessor*: The DataProcessor class receives data from the DataDistributor class and performs the evaluation of the quality of data sets using either polynomials or circular splines.
  Depending on the implementation, the DataProcessor class uses one of the three optimization algorithms described above.
– *OptimizedCurve and DataSet*: These classes are used for storing the results of an already processed dataset (data number, data quality, ...)
– *InterpolatingCurve and DesignDomain*: These classes are parent to all implementations of curve classes and provide an interface to the optimization class OptiBB and STDE, if one of the two available CPU optimization algorithms is used.
– *OptiBB*: OptiBB is a wrapper around Ewald Lindner's code and performs an optimization with respect to a functional. The optimization can be done for every curve class, which inherits from DesignDomain.
– *STDE*: This class is a wrapper for the own implementation of a Quasi–Newton algorithm, see §5.

The DataDistributor class manages all data sets and first sends the concentration data c_i to all processes in the MPI world and then a data set to each instance of DataProcessor, if there is enough data available. Every instance of DataProcessor that received a data set then determines the quality of the data set and sends the result back to the instance of DataDistributor. If there is still data to be processed, another set of data is sent to the particular instance of DataProcessor that sent the last result about the data quality.
For the GPU version, many data sets are sent in one step and the data management itself is handled by an additional MPI process (i.e. DataWrapper is an additional process). This allows a better scalability. Note, that this data exchange description and the illustration in Fig. 4 present only the idea of the communication process and it differs from the more complex communication protocol and its implementation.

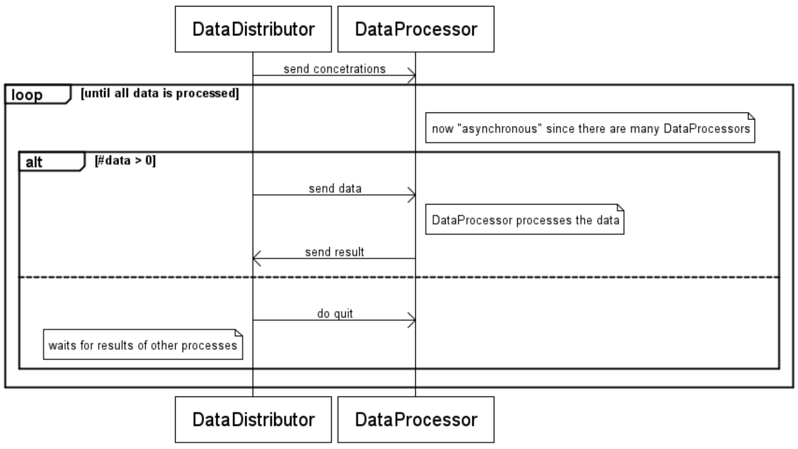
DataDistributor
DataProcessor
send concetrations
loop [until all data is processed]
now "asynchronous" since there are many DataProcessors
alt [#data > 0]
send data
DataProcessor processes the data
send result
do quit
waits for results of other processes
DataDistributor
DataProcessor

Fig. 4 Interaction between the system components

4 Algorithms for the curves

In the following section C++ snippets are shown. To provide a better readability of the snippets, a short list of the most important class members and methods is given now:

m	Number of frequency response vectors
f	Frequency response data
c	Concentration data
a	Polynomial parameter array / Circular spline parameter arrays
b	Circular spline parameter array
cc	Circular spline parameter array
k	Circular spline parameter array
hsquare	Circular spline parameter array
pArcs	Number of independent circular arcs
bArcs	Half the number of dependent circular arcs (C^1–property)
t	Projection points
s	Arc length vector for the projection points
N	Dimension of frequency response vectors
p	Polynomial degree
NJ	Number of interval searches
NN	Number of Newton-steps
NH	Discretization grain size
double inner(double* a, double* b)	Inner product of a and b, induced by the Euclidean norm
void curve(double x, double* fq)	fq is set to the evaluation of the curve at parameter x
void curve_1(double x, double* fq)	fq is set to the evaluation of the curve's first derivative at parameter x
void curve_2(double x, double* fq)	fq is set to the evaluation of the curve's second derivative at parameter x
void curve_dist(double x, double* fq)	Distance between the data point fq and the curve at parameter x

4.1 Polynomial curves

4.1.1 Evaluation

Polynomials are easy to handle since they are C^∞ and rather fast to evaluate using the Horner-scheme. Also they don't have to be rebuilt like splines when a parameter is changed.

Listing 1 CPU polynomial evaluation

```
void InterpolatingPoly::curve(const double x, double val[N]) {
    // Horner scheme evaluation
    for (int k=0;k<N;++k)
        val[k] = a[p][k];
    for (int j=p-1;j>=0;--j) {
        for (int k=0;k<N;++k)
                val[k] = val[k]*x + a[j][k];
    }
}
```

4.1.2 Projection

All m points of $\{f_i\}_{i=1}^m$ have to be orthogonally projected onto the polynomial curve q. The projection is done iteratively by the Newton method, since a direct computation of a projection for a polynomial of degree 3 would already require a closed formula for the zeros of a polynomial of degree 5. For every point (line 5) an interval search is done first (line 15) to find a proper starting point tt, which is iteratively improved using Newton's method (line 26).

Listing 2 CPU polynomial projection

```
1  void InterpolatingCurve::projection(double t[]) {
2      double fq[N], d1[N], d2[N], tt;
3
4      // Perform m projection steps
5      for (int i=0;i<m;++i) {
6          // Interval search for start point
7          double tj, dj;
8          double min_j = 0;
9          double val = curve_dist(tj, f);
10         double h = (t_end-t_beg)/NJ;
11
12         tj = t_beg;
13
14         // Perform interval search
15         for (int j=1;j<=NJ;++j) {
16                 tj += h;
17                 dj = curve_dist(tj, f);
18                 if (dj<val) {
19                     val   = dj;
20                     min_j = j;
21                 }
22         }
23         tt = t_beg+min_j*h;
24
25         // Newton iteration with initial value "tt"
26         for (int l=0;l<NN;++l) {
27             curve(tt, fq);
28             for (int k=0;k<N;++k)
29                 fq[k] = f[i][k]-fq[k];
30
31             curve_1(tt, d1);
32             curve_2(tt, d2);
33             tt -= inner(fq,d1) / (inner(fq,d2)-inner(d1,d1));
34         }
35
36         t[i] = tt;
37     }
38 }
```

4.1.3 Arc length

As mentioned above, a general closed formula for the arc length of a polynomial of degree n does not exist. Therefore the parameter domain is discretized and the polynomial is linearized piecewise (line 12) in order to approximate the arc length of the polynomial. For every projection point $t[i]$, $i = 1, ..., m-1$ the arc length of the curve from $t[0]$ to $t[i]$ is approximated (line 5). To reduce computations, the arc length from $t[0]$ to $t[i-1]$ is used for the computation of the arc length from $t[0]$ to $t[i]$. For this sg was used in line 6, because it is possible that $t[i-1] > t[i]$.

Listing 3 CPU polynomial arc length

```
1  void InterpolatingCurve::arc(const double t[], double s[]) {
2      double h, sum, sn, e[N], d[N], sg;
3       s[0] = 0.0;
4
5       for (int i=1;i<m;++i) {
6           sg  = (t[i]>t[i-1]) ? 1.0:-1.0;
7           h   = (t[i]-t[i-1])/NH; // Stepsize in the parameter domain
8           curve(t[i-1], d);
9           sum = 0.0;
10
11          // Arc length approximation
12          for (int l=1;l<=NH;++l) {
13              std::copy(d,d+N,e);
14              curve(t[i-1]+l*h, d);
15              sn = 0.0;
16
17              for (int k=0;k<N;++k)
18                  sn += (d[k]-e[k])*(d[k]-e[k]);
19
20              sum += sqrt(sn);
21          }
22          s[i] = s[i-1]+sum*sg;
23      }
24  }
```

4.2 Circular splines

The use of circular splines [10] has benefits regarding the projection step and the arc length computation. However, the spline has to be rebuilt every time there are changes in its parameters.

4.3 Evaluation

First the parameter x is checked for a valid value, i.e. if the circular spline q is well–defined at this point (line 3). Then a mapping from x to the according spline segment is done (line 9 and 11). The actual evaluation is performed according to equation (11) (line 15).

Listing 4 CPU circular spline evaluation

```
1   void CircularSpline::curve(const double x, double val[N]) {
2       // Check for a valid parameter x
3       if( x < 0 || x > (3*bArcs+1) ) {
4           for(int i=0;i<N;++i)
5               val[i] = std::numeric_limits<double>::max();
6       }
7       else {
8           // Determine the cicular arc of the spline belonging to the spline
9           int pos = static_cast<int>(x);
10          // Determine parameter for the circular arc
11          double u = x - pos;
12          double w = 1 - u;
13
14          // Evaluation of the circular arc at parameter u
15          for(int i=0;i<N;++i)
16              val[i] = (w*w*a[pos][i] + 2*u*w*omega[pos]*cc[pos][i] + u*u*a[pos+1][i])
17                       / (w*w + 2*u*w*omega[pos] + u*u);
18      }
19  }
```

4.3.1 Projection

For every point $\{f_i\}_{i=1}^{m}$ the segments y_k, $k = 1, \ldots, n$ of the circular spline q are checked for possible projection points. Either until one is found, or there is no projection point for f_i on the circular spline (line 50). This is done by solving (12) (line 13 to 24), which is equivalent to solving a fourth order equation with the coefficients $c_0, \ldots, c_4$. If one of the solutions $s_1, \ldots, s_4$ is a real number r in the interval $[0, 1]$, then $q(k + r)$ is a projection point of f_i onto q (line 236 to 45).

Listing 5 CPU circular spline projection

```
void CircularSpline::projection(double t[]) {
    double c0, c1, c2, c3, c4;
    int    seg=0;

    // For every data point f
    for (int j=0;j<m;++j) {
        bool isProjection = false;
        int  iter = 0;

        // Search for a projection point on each segment until one is found
        // or there is none
        while(isProjection == false) {
            c0 = ....
            c1 = ....
            c2 = ....
            c3 = ....
            c4 = ....

            bool   isSolution = false;
            double res;

            std::complex<double> s1, s2, s3, s4;

            solve_quartic_proj(c4, c3, c2, c1, c0, s1, s2, s3, s4);

            if(s1.real() <= 1.0 && s1.real() >= 0.0 && fabs(s1.imag()) == 0.0) {
                res = s1.real();
                isSolution = true;
            }
            else if(s2.real() <= 1.0 && s2.real() >= 0.0 && fabs(s2.imag()) == 0.0) {
                res = s2.real();
                isSolution = true;
            }
            else if(s3.real() <= 1.0 && s3.real() >= 0.0 && fabs(s3.imag()) == 0.0) {
                res = s3.real();
                isSolution = true;
            }
            else if(s4.real() <= 1.0 && s4.real() >= 0.0 && fabs(s4.imag()) == 0.0) {
                res = s4.real();
                isSolution = true;
            }

            if(isSolution) {
                t[j] = seg + res;
                isProjection = true;
            }
            seg++;
            iter++;

            if(iter > pArcs+2*bArcs) {
                t[j] = std::numeric_limits<float>::quiet_NaN();
                isProjection = true;
            }
            if (seg >= pArcs+2*bArcs)
                seg = 0;
        }
    }
}
```

4.3.2 Arc length

Equation (15) allows us the computation of the arc length of every segment of the circular spline q. In a first step the arc lengths of all spline segments with respect to 0 are stored in *seglen* (line 16 to 23). Then the arc lengths $s[i]$ from 0 to $t[i]$ (line 26 to 42) and finally from $t[0]$ to $t[i]$ (line 46) are computed.

Listing 6 CPU circular spline arc length

```
1   void CircularSpline::arc(const double t[], double s[]) {
2       double sn;
3       double alpha, phi;
4
5       double val[N];
6
7       // Arc length of each spline segment
8       double* seglen = new double[pArcs+2*bArcs];
9       // Radius of the circles, belonging to the spline segments
10      double* r      = new double[pArcs+2*bArcs];
11
12      s[0] = 0.0;
13      seglen[0] = 0.0;
14
```

```
    // Computation of the arc length for each spline segment
    for(int i=1;i<pArcs+2*bArcs;++i) {
        phi = acos(omega[i-1]);
        alpha = M_PI/2 - phi;
        r[i-1] = fabs(hsquare[i-1])/cos(alpha);

        sn = r[i-1]*2*phi;
        seglen[i] = seglen[i-1] + sn;
    }

    // Computation of the arc length from 0 to t[i]
    for(int i=0;i<m;++i) {
        int segment = static_cast<int>(t[i]);

        curve(t[i], val);

        for(int j=0;j<N;++j)
            val[j] = val[j]-a[segment][j];

        double dd = norm(val);

        if (fabs(dd/(2*r[segment]))<=1.0)
            sn = 2*r[segment]*asin(dd/(2*r[segment]));
        else
            sn = 2*r[segment]*asin(dd/(2*r[segment]-1.0))+r[segment]*M_PI;

        s[i] = seglen[segment]+sn;
    }

    // Computation of the arc length from t[0] to t[i]
    for(int i=m-1;i>=0;--i)
        s[i] -= s[0];

    delete[] seglen;
    delete[] r;
}
```

5 The optimization algorithm

As mentioned in §3.2 the optimization algorithm in STDE uses a Quasi–Newton method with an Inverse-BFGS-Update. Instead of performing a line search with the commonly used strategies, a very naive approach prove to be much more efficient for our purpose. We start the line search in each Quasi–Newton iteration with a fixed step size α and reduce the step size by multiplication with a fixed factor as long as the functional evaluation $F(x_0)$ of the starting point x_0 is bigger than the evaluation at $F(x_0 + \alpha d)$, where d is the Quasi–Newton descent direction. The reason for this is the reduction of divergent branches in the GPU version of the algorithm on one hand, and the steepness of the F at certain points, which lead to problems with some common line search strategies on the other hand (see §7.2.3), especially when using single precision data types. Also the fact, that we have an extremely large amount of available data to process is important. So, if one data set is not evaluated correctly (i.e. the convergence criterion for an optimization is not fulfilled), the data set gets purged.
As above, a short description of the most important members and methods of the STDE class is given now:

n	Number of the functional parameters, known at compilation time
x0	Current functional parameters
xi	New functional parameters
grad	Gradient of the functional
d	Descent direction for the line search
hessian	Approximation of the hessian, using the BFGS method
dnew	Helper variable
dd	Instance of DesignDomain
s, y	Helper vectors for BFGS-update
BFGS_ITER	Number of optimization steps
LS_ITER_MAX	Maximal number of backtracking steps
LS_ALPHA	Initial value for the line search
LS_STEP_SIZE	Backtracking factor for the line search
double inner(double* a, double* b, int n)	Inner product of a and b, induced by the Euclidean norm
double norm(double* a, int n)	Euclidean norm of a
double Functional(double* x, int n)	Functional evaluation at position x
void vtv(double* a, double* b, double** m, int n)	Vector-vector multiplication with result m of dimension n
void vtm(double* a, double** h, double* b, int n)	Vector-matrix multiplication with result b (length n)
void mtv(double** h, double* a, double b, int n)	Matrix-vector multiplication with result b (length n)

Listing 7 CPU Optimization

```
void STDE::Optimize(double* normgrad) {
    double fnew = FLT_MAX;

    double hnew[n][n]; // used for hessian update
    double ha[n][n];   // helper variable for hessian update

    // Set the hessian approximation to the identity matrix
    for(int i=0;i<n*n;++i)
        *(hessian[0]+i) = 0.0;

    for(int i=0;i<n;++i)
        hessian[i][i] = 1.0;

    // Functional and gradient evaluation at the starting point
    func_val = Functional(x0, n);
    gradient(x0, grad);

    // Quasi--Netwon iteration
    for(int k=0;k<BFGS_ITER;++k) {
        alpha = LS_ALPHA;

        for(int i=0;i<n;++i)
            dnew[i] = -grad[i];

        // descent direction
        mtv(hessian, dnew , d, n);

        for(int i=0;i<n;++i)
                xi[i] = x0[i] + alpha*d[i];

        func_val = Functional(x0,n);
        int lsiter = 0;

        // Naive line search
        while(((fnew = Functional(xi, n)) > func_val) && lsiter < LS_ITER_MAX) {
            alpha*=LS_STEP_SIZE;

            for(int i=0;i<n;++i)
                xi[i] = x0[i] + alpha*d[i];

        }

        dd->GetDesignVar(x0, n);

        for(int i=0;i<n;++i)  {
            s[i] = alpha*d[i];
            y[i] = -grad[i];
        }

        gradient(x0, grad);

        for(int i=0;i<n;++i)
            y[i] += grad[i];

        memcpy(hnew, hessian, n*n*sizeof(double));

        double a, b, sy;

        sy = inner(s, y, n);

        // If sy > 0 then the hessian update is positive definite
        if(sy > 0.0) {
            mtv(hessian, y, dnew, n);

            a = (sy + inner(y, dnew, n))/(sy*sy);
            vtv(s, s, ha, n);

            for(int i=0;i<n;++i)
                for(int j=0;j<n;++j)
                    hnew[i][j] += a*ha[i][j];

            vtv(dnew, s, ha,n);

            for(int i=0;i<n;++i)
                for(int j=0;j<n;++j)
                    hnew[i][j] -= ha[i][j]/sy;


            vtm(y, hessian, dnew, n);
            vtv(s, dnew, ha, n);

            for(int i=0;i<n;++i)
                for(int j=0;j<n;++j)
                    hessian[i][j] = hnew[i][j] - ha[i][j]/sy;

        }
        //If hessian is not > 0 for sure: Reset to identity matrix
        else {
            for(int i=0;i<n*n;++i)
                *(hessian[0]+i) = 0.0;

            for(int i=0;i<n;++i)
                hessian[i][i] = 1.0;
        }
        // End of hessian update
    }
    // set normgrad to the norm of the gradient
    *normgrad = norm(grad, n);
}
```

6 GPU code

The GPU code was designed in a way that many functional evaluations, respectively optimizations are performed concurrently. One thread on the GPU performs one functional evaluation, respectively one optimization. This approach is possible because only few divergent branches can occur, besides some insignificant exceptions. The problem with divergent branches on GPUs in general consists in the serialization of the according branch executions in the same warp (32 threads on the GPUs used).

6.1 Functional Evaluation

Evaluating the functional F is a sequential problem and not suitable for parallelization. Hence, the mapping "one thread - one functional evaluation" was chosen. The frequency responses f, the concentration data c and the curve's parameters a (9) have to be stored on the GPU. The most successful optimization strategy was to align the parameters a to the GPUs memory layout to avoid unnecessary memory overhead[4]. The resulting code snippet follows.

Listing 8 CPU vs. GPU memory layout for polynomial curves

```
// Comparison of the layout of M 2 dimensional parameters
// M ... Number of threads
// P ... Polynomial degree
// N ... Dimension

// Arrays for CPU (theoretical) and for the GPU (actually used) implementations
float a_cpu[M][P+1][N]; // CPU alignment
float a_gpu[P+1][N][M]; // GPU alignment

// Access of the first parameter belonging to a thread
unsigned int thread_id = BLOCKSIZE * blockIdx.x + threadIdx.x;
a_cpu[thread_id][0][0]; // CPU access
a_gpu[0][0][thread_id]; // GPU access
```

So, in difference to the listings of the CPU code, every time access to the functional parameters is performed, the parameter arrays are accessed as above. As a concrete example the following listing shows how concurrent polynomial evaluations are performed on the GPU (compare Listing 1). The values of $GRIDSIZE$ and $BLOCKSIZE$ control the number of executed threads and their topology.

Listing 9 GPU polynomial curve evaluation for N=3

```
__device__ void curve(float a[][N][GRIDSIZE*BLOCKSIZE], const float x, float val[N]) {
    #pragma unroll
    for(int j=0;j<N;++j)
            val[i] = a[P][i][BLOCKSIZE*blockIdx.x + threadIdx.x];

    #pragma unroll
    for (int j=P-1;j>=0;--j) {
          val[0] = val[0]*x + a[j][0][BLOCKSIZE*blockIdx.x + threadIdx.x];
          val[1] = val[1]*x + a[j][1][BLOCKSIZE*blockIdx.x + threadIdx.x];
          val[2] = val[2]*x + a[j][2][BLOCKSIZE*blockIdx.x + threadIdx.x];
    }
}
```

[4] see the Nvidia CUDA Programming Guide 2.0, `http://developer.download.nvidia.com`

Besides the layout issues and the "#pragma unroll" directive for loop enrolling, another GPU specific programming technique, recalculating instead of storing and fetching, is used. However, whether recalculation pays off depends on several aspects and has to be tested. We avoided to use the GPU shared memory after some measurements. The reason is that we had to use mainly global memory because even for a rather small number of executed threads, the amount of shared memory was not sufficient to store the data. On the other hand, the memory latency of the global GPU memory has been masqueraded quite well by a huge amount of executed threads. The number of available registers is sufficient for storing results of intermediate calculations. A further performance improvement has been achieved by using CUDA specific mathematical routines as $sqrtf()$ which are not IEEE compliant.

6.2 Optimization algorithm

The implementation of the optimization algorithm basically follows the same strategies as the functional evaluation above. Again, the mapping "one thread - one optimization" was chosen. This strategy is suitable because the available GPU memory is sufficiently large. This also allows an easy porting of the CPU code for the GPU. The nvcc compiler apparently does fulfill the IEEE 754 standard therefore operations as $memset(x, 0, sizeof(x))$ are not used. Mainly single precision has been used but for comparison a double precision version of the code was also implemented, see Tab. 5. The functional evaluation returns FLT_MAX in case the curve is too far away from the frequency response vectors during line search. This allows in most cases a successful continuation of the optimization process.

In order to handle the different convergence behavior of the functional when different data sets are used, a relatively small number of Quasi–Newton iterations are performed in a first step. This allows to separate those data sets from all currently processed data sets, that do not fulfill the convergence criterion yet. As a consequence, "sufficiently" evaluated data sets are replaced by new ones and the remaining old data sets remain for further optimization. In this case, performed tests indicated that a change in the optimization parameters does not have a beneficial effect, so only the number of Quasi–Newton iterations increases for those data sets.

7 Results

All data sets consist of 9 frequency response vectors, each one defined in $\mathbb{R}^3$. Therefore, the parametric polynomial curve consists of 3 cubic polynomials having 12 parameters. The circular spline has also a three dimensional parametric representation consisting in our case of 8 primary–arcs and 7 bi–arcs according to the notation in [10]. This means 71 parameters and hence a significantly bigger flexibility and approximation ability for the circular splines than the polynomial curves.

7.1 Functional evaluation

Although the structure of the algorithm for the functional evaluation takes a very similar amount of time for each data set, the same sets were used for CPU and GPU evaluation to guarantee comparable results.

7.1.1 CPU

The CPU benchmark with 40960 functional evaluations has been performed as single process on one CPU core on a computer with an Intel Core i7–920@2.67GHz CPU and 6GB DDR3 RAM.[5] It took 2.17s, i.e., 50.29μs for each evaluation when the polynomial curves have been used and it took 0.47s, i.e., 11.48μs for each evaluation in case of circular splines, see Tab. 3.

7.1.2 GPU

The GPU/CPU benchmark for functional evaluations has been performed on a computer with an AMD Phenom 9950 Quadcore Processor with 8GB DDR2 RAM and four Nvidia GTX 280 (1GB Graphics RAM) and on an Intel Core i7–920 system @ 2.67GHz with 12GB DDR3 RAM and two Nvidia GTX 480 (1.5GB Graphics RAM). We used only one core of the CPU and one GPU on each system. The time for evaluation of the functional depends on the chosen parameters for grid size and block size of the appropriate CUDA kernel call.

Already for a relatively small number of threads good results could be achieved. Fig. 5 shows the correlation between the number of threads and time needed per evaluation on the GTX 280 GPU. One can notice a slight beak–in with a block size of 160, which shows that a proper choice of the thread topology is of importance. Tab 1 and tab 2 give an overview about the average timings for functional evaluations on the GPUs depending on the number of threads. The best results, respectively best thread topology is marked. The performance gain with respect to the number of threads is in each case mainly saturated at a block size of 64.

Comparing these runtimes with the timings in §7.1.1 the GPU outperforms the CPU tremendously.

Block size	**Num. threads**	**Polynomials [μs]**	**Circular splines [μs]**
8	5 120	0.527	1.605
16	10 256	0.312	0.996
32	20 544	0.211	0.652
64	41 152	0.167	0.496
96	61 440	0.164	0.496
128	82 432	0.154	0.443
160	103 200	0.163	0.510
192	124 032	0.157	0.447
224	144 928	0.154	0.445
256	165 888	0.150	**0.423**
288	184 320	0.149	n/a
320	204 800	**0.149**	n/a

Table 1 Run time for functional evaluations with a grid size of 640 on the GTX 280 GPU

[5] `http://www.kfunigraz.ac.at/~haasegu/Lectures/RO-I/WS08/Projekte/report_compute_pc_ws08.pdf`

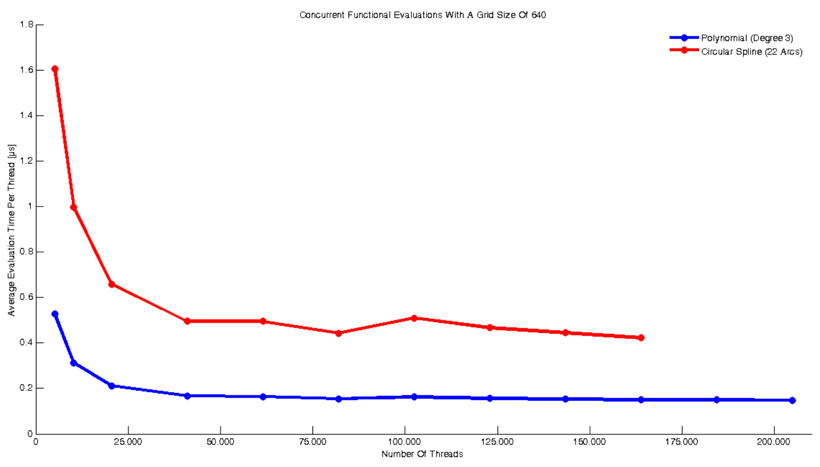

Fig. 5 Run time on the GTX 280 GPU for function evaluations depending on the thread topology

Block size	**Num. threads**	**Polynomials** [μs]	**Circular splines** [μs]
8	5 120	0.338	0.957
16	10 256	0.195	0.569
32	20 544	0.118	0.365
64	41 152	0.082	0.257
96	61 440	0.078	0.247
128	82 432	0.076	0.236
160	103 200	0.075	0.238
192	124 032	0.075	0.256
224	144 928	0.079	0.243
256	165 888	0.076	**0.235**
288	184 320	0.074	0.295
320	204 800	**0.074**	0.281

Table 2 Run time for functional evaluations with a grid size of 640 on the GTX 480 GPU

7.1.3 Timing overview

Tab 3 and tab 4 show the speedup results compared to the CPU evaluation. These results are based on the optimal thread topology. Note, that the functional evaluations on the CPU are done sequentially and the functional evaluations on the GPU are done concurrently.

	CPU [μs]	**GPU** [μs]	**Speedup**
Cubic polynomial	50.29	0.15	337.91
Circular spline	11.47	0.42	27.30

Table 3 Time per functional evaluation with the best thread topology on the GTX 280 GPU

	CPU [μs]	**GPU [μs]**	**Speedup**
Cubic polynomial	50.29	0.07	687.08
Circular spline	11.47	0.24	47.20

Table 4 Time per functional evaluation with the best thread topology on the GTX 480 GPU

One can notice a big difference in the speedup between polynomials and circular splines. Basically there are three different reasons for this: First the number of parameters for the circular spline is significantly higher than the number of parameters for the cubic polynomial, thus copying the spline's parameters to the GPU takes almost as long as the evaluation of the spline. Secondly the maximal number of concurrent spline evaluations is smaller and the memory latency cannot be masqueraded as well as in the polynomial's evaluation. Also the circular spline code was not optimized as well as the code for polynomials, because the data and parameter transfer to the GPU is a big limiting factor anyway.

7.2 Optimization

Test system (A) is a compute cluster owned by the Mathematics Department[6] of the University of Wyoming. The cluster consists of one master node with one Intel Core i7 CPU 965 @ 3.20GHz and 8 Nvidia GeForce GTX 295 GPUs and 7 computing nodes with each having 4 Intel Core i7 CPU 965 @ 3.20GHz and 8 Nvidia GeForce GTX 295 GPUs. Due to problems with the power supply of the computing nodes, only the master node was used.
Test system (B) is an Intel Core i7-920 system @ 2.67GHz, 12GB DDR3 RAM and two GeForce GTX 480 GPUs[7].
For the GPU code on (A) a thread topology with a grid size of 90 and a block size of 256 (96) for single precision (double precision) computations, i.e., 23040 (8640) concurrent optimizations was chosen. The GPU code on (B) was executed with a thread topology with a grid size of 90 and a block size of 256 (192) for single precision (double precision) computations, i.e., 23040 (17280) concurrent optimizations. These numbers were close to the maximal number of concurrent optimizations for both GPUs, the GeForce GTX 280 and the GeForce GTX 480.

7.2.1 Timing

The table below allows the comparison of the different approaches regarding performance. One can see that the CPU/GPU single precision version is about 65 times faster on (A), resp. 194 times faster on (B), than the CPU-only version on (A) of the same optimization algorithm. Comparing the CPU/GPU single precision version with the intelligent optimization algorithm on (A) we even reach an acceleration of 146 on A and 438 on (B), respectively. The single precision CPU/GPU version is 2.5 times faster than the double precision CPU/GPU version on (A). The GPU in system (B) is the best NVidia GPU available (Aug. 2010) whereas system (A) contains one year old

[6] `http://math.uwyo.edu`
[7] `http://www.kfunigraz.ac.at/~haasegu/Lectures/RO-I/WS09/ComputeServer/bericht_gruppe2_compute_Server10.pdf`

	Data sets	Total [s]	Per data set [ms]	Speedup
CPU intelligent (A)	10000	307	30.70	
CPU naive (A)	10000	136	13.60	1
CPU/GPU float (A)	10000000	2128	0.21	65
CPU/GPU float (B)	10000000	732	0.07	194
CPU/GPU double (A)	10000000	5422	0.54	25
CPU/GPU double (B)	10000000	2043	0.20	68

Table 5 Run time and GPU–speedup for optimization of the data sets

GPUs. The redesigned Fermi architecture is responsible for the additional performance factor 3 in comparison to system (A).

7.2.2 Parallel speedup

One can see from Tab. 6 that the CPU/GPU approach has a far better parallel speedup than the CPU-only approaches. This is caused by the architecture of the MPI communication process. While the CPU–only versions process one data set after the other, the CPU/GPU approaches process many data sets concurrently. This reduces the amount of time the instances of the DataProcessor classes are idle and also reduces the overhead for the DataDistributor instance.

DataProcessor	CPU intelligent	CPU naive	CPU/GPU float (A)	CPU/GPU double (A)
1	1.00	1.00	1.00	1.00
2	1.99	2.00	1.97	n/a
3	3.01	3.02	2.94	n/a
4	3.74	3.49	3.26	3.03
5	4.32	4.00	3.88	3.91
6	4.95	4.39	4.14	4.94
7	5.58	4.53	5.96	5.75
8	5.69	4.53	7.09	5.86

Table 6 Parallel speedup on CPUs and GPUs for the DataProcessor processes.

The parallel speedup is mainly saturated at 7 processes of the DataProcessor using 7 CPU–cores because one additional CPU–core is needed for the DataDistributor process, see §3.3.

7.2.3 Optimization quality

Depending on the chosen optimization algorithm and the data types (single or double precision) different results occur in the optimization process after a fixed number of iterations. Table 7 presents the achieved results. The convergence criterion is fulfilled if the gradient norm of the functional F is less than 0.1. Comparing the two optimization algorithms with the same test data sets on the CPU, one can see that the intelligent Quasi–Newton algorithm is less successful in the optimization process than the naive Quasi–Newton implementation presented above. The reason for this behavior are the different line search strategies: The line search of the intelligent Quasi-Newton

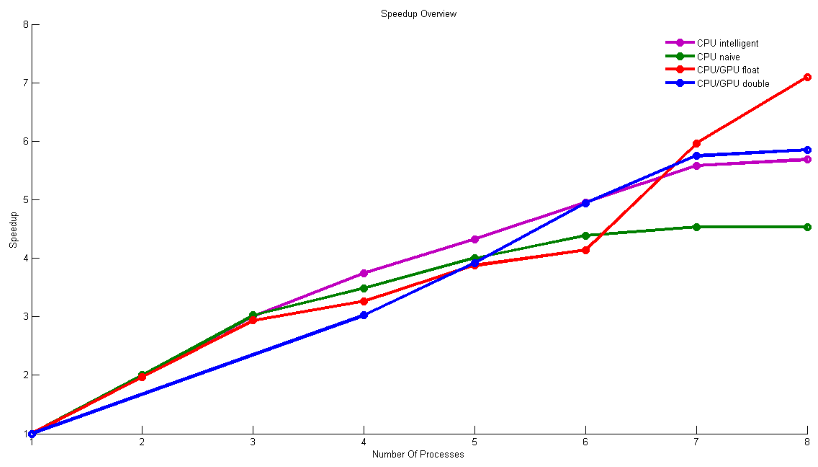

Fig. 6 Speedup of optimization on CPUs and GPUs

	Data sets	**Iter. max.**	**Failed [%]**	**Failed convg. [%]**	**Numerical error [%]**
CPU intelligent	10000	40	13.43	13.42	0.01
CPU naive	10000	40	5.31	5.24	0.07
CPU/GPU float	10000000	40	1.09	1.06	0.02
CPU/GPU double	10000000	40	1.16	1.15	0.01

Table 7 Failure Statistics for the optimization

algorithm iteratively reduces the length of an initial search interval by bisection and the generation of a quadratic model of F for the descent direction until a sufficient descent is achieved. Tests showed, that this line search often failed, i.e. the quadratic model function was too flat. So a quadratic model function is not suitable for approximation. On the other hand, the functional's structure makes it difficult to find a point with sufficient descent. Experience gathered while trying out line search algorithms for F shows that common strategies fail for determining an acceptable step length used in Quasi–Newton methods, like the Goldstein conditions and the Wolfe conditions. These conditions are a set of inequalities, based on ∇F, that have to hold for the current step length in order to be accepted.
Although our line search approach does not guarantee convergence rates, we always find enough "good" frequency tuples because of the amount of available test data. The best e.g. 100 frequency tuples may be re-investigated with possible post-processing.
We noticed in the CPU/GPU approach that the double precision version is more stable against numerical problems but less successful in optimization than the single precision variant. The reason for this is the use of exactly the same optimization parameters in both versions and a few modifications in the functional evaluation for the single precision version, which try to compensate inacurracies due to single precision in the projection step.

One can notice a significant difference between the convergence results of the CPU and CPU/GPU versions. This is caused by the fact that the test data sets are in some way ordered by the value of their norms and the ones for the CPU version are the first 10000 data sets of those of the CPU/GPU version. Apparently smaller norms of the data sets mean poorer convergence behavior.

8 Conclusion

A comparison of CPU-only and CPU/GPU timings was given and one can clearly see, that recent GPUs can be a proper choice in parallel optimization of many sequential problems, not only because of better performance, but also because of good scalability. However, the success of the CPU/GPU approach above is caused by the very small amount of divergent branches in the evaluation of the functional F. One can assume that future GPU and many–core hardware will provide even better double precision performance and better handling of divergent branches, which will most likely allow the evaluation and optimization of more delicate functionals. if some assumptions are fulfilled [8] then a naive approach in nonlinear optimization can be more successful than algorithms which guarantee convergence for very ill–conditioned gradients.

The results above give a perspective that nonlinear optimization in general could be performed on GPUs, if the mapping "one GPU block - one optimization“ or even "one GPU - one optimization“ and not like in our case “one GPU thread - one optimization” is chosen. This approach is currently under investigation.

References

1. Bancsik, Z., Juhasz, I.: On the arc length of parametric cubic curves. Journal for Geometry and Graphics **3**(1), 1–15 (1999)
2. Douglas, C., Haase, G., Hannel, T., Link, D., Lodder, R., L.Deng: A prototype for detecting defective pills during manufacturing. In: Proceedings DCABES 2009 (2010)
3. Douglas, C., L.Deng, Efendiev, Y., Haase, G., Kucher, A., Lodder, R., Qin, G.: Advantages of multiscale detection of defective pills during manufacturing. In: W.Z. et al. (ed.) HPCA 2010, *LNCS*, vol. 5938, pp. 8–16. Springer (2010)
4. Douglas, C.C., Li, D., Haase, G., Lee, H., Loder, R.: Data-driven pill monitoring. Elsevier, Procedia Computer Science (2010). Submitted
5. Fletcher, R.: Practical Methods for Optimization, Vol. 2. John Wiley & Sons, Chichester (1981)
6. Gill, P.E., Murray, W., Wright, M.H.: Practical Optimization. Academic Press, London-San Diego-New York (1981)
7. Haase, G., U.Langer, Lindner, E., Mühlhuber, W.: Various methods for structural optimization problems with industrial applications. In: W. Wall, K.U. Bletzinger, K. Schweizerhof (eds.) Trends in Computational Structural Mechanics, pp. 623–636. CIMNE (2001)
8. Nocedal, J., Wright, S.: Numerical Optimization. Springer Series in Operations Reasearch. Springer, New York (1999)
9. Powell, M.J.D.: A fast algorithm for nonlinear constrained optimization calculations. In: G.A. Watson (ed.) Numerical Analysis, no. 630 in Lecture Notes in Mathematics. Springer, Berlin (1978)
10. Song, X., Aigner, M., Chen, F., Jüttler, B.: Circular spline fitting using an evolution process. Journal of Computational and Applied Mathematics **231**, 423–433 (2009)
11. Wang, H., Kearney, J., Atkinson, K.: Arc-length parameterized spline curves for real-time simulation. In: T. Lyche, M. Mazure, L. Schumaker (eds.) Proceedings of the "5-th International Conference on Curves and Surfaces”, San Malo 2002, pp. 387–396. Nashboro Press, Brentwood, TN (2003)