Sparse-Matrix-CG-Solver in CUDA

Dominik Michels[*]
Supervised by: Stefan Hartmann[†]

Institute of Computer Science II
Rheinische Friedrich-Wilhelms-Universität Bonn
Bonn / Germany

Abstract

This paper describes the implementation of a parallelized conjugate gradient solver for linear equation systems using CUDA-C. Given a real, symmetric and positive definite coefficient matrix and a right-hand side, the parallized cg-solver is able to find a solution for that system by exploiting the massive compute power of todays GPUs. Comparing sequential CPU implementations and that algorithm we achieve a speed up from 4 to 7 depending on the dimension of the coefficient matrix. Additionally the concept of preconditioners to decrease the time to find a solution is evaluated using the SSOR method. In the end additional suggestions are provided to further increase the speed of the presented CUDA cg-solver.

Keywords: parallized GPU solver, sparse matrix solver, conjugate gradient, ELLPACK-R, NVIDIA CUDA, SSOR precondition, 2D heat equation

1 Introduction

In several applications one has to solve linear equation systems with real, symmetric and positive definite coefficient matrices. Examples for such systems are broadly available e.g. physical deformation (cf. [6]), implicit mesh smoothing, mesh parameterization (cf. [9]), diffusion equation for terrain generation (cf. [8]). Usually linear equation systems are derived from dicretizing a continous problem resulting in very sparse coefficient matrices because only a small number of neighboring elements take influence on a specfic element. Therefore coffecient matrices can be stored very efficiently using sparse matrix formats e.g. the ELLPACK-R format which is used here. Such linear systems can become very large regarding to the given problem and one would like to find a time efficient solution for such systems. In literature the conjugate gradient algorithm is suggested to solve such linear, symmetric and positive definite systems. Specific operations of that algorithm can be parallelized e.g. scaled vector addition, dot product and matrix-vector multiplication. In this work a parallel implementation of the conjugate gradient algorithm using the NVIDIA CUDA architecture is presented to exploit the massive compute power of todays GPUs. Additionally the performance of the algorithm is compared to sequential and parallel implementations. Finally the 2D heat equation is solved using the parallized cg-solver.

[*]michels@uni-bonn.de
[†]hartmans@cs.uni-bonn.de

2 Related Work

The conjugate gradient algorithm was introduced in [7] as an efficient method to solve linear equation systems with real, symmetric and positive definite coefficient matrices. Additional details regarding the conjugate gradient algorithm can be found in [1], [11] and [12]. With the appearance of programmable graphics hardware a cheap way for getting massive parallel processors to the masses became possible. Therefore, general people can also take advantages from parallel algorithms. When analysing the conjugate gradient algorithm one will recognize that serveral operations within one iteration of the algorithm can be computed in parallel. In [5] a parallel implementation was introduced using GLSL shader programs. The necessary data for the computation was stored in textures and the algorithm was implemented in a pixel shader. Due to the general purpose GPU paradigm encoding of data in textures got superfluous because new technologies, like the NVIDIA CUDA architecture allow programming the parallel processors using an extended version of the C programming language (see [10]). An exemplary implementation of the conjugate gradient algorithm using CUDA is shown in [2]. If one wants to reach a time efficient implementation of the conjugate gradient algorithm it is essential to have an efficient way to compute sparse matrix vector products. In [4] an efficient storage for sparse matrices the ELLPACK format was suggested, which was extended to the ELLPACK-R format in [13] (explained later) especially for the use on GPUs. For time efficient solution of linear systems it is not enough to have efficient data structures and a lot of compute power: There exist extensions to the conjugate gradient algorithm which use a preconditioning matrix which is applied to the system matrix to decrease its condition number. In this paper a sequential version of such preconditioner is evaluated using the SSOR method described in [3].

3 Conjugate Gradient Algorithm

The primary introduction of the conjugate gradient algorithm from 1952 can be found in [7]. This method is used to solve linear systems $Ax = b$ with $A \in \mathbb{R}^{n \times n}$ (symmetric, positive definite) and $b \in \mathbb{R}^n$.
In this case the solution of the linear system is equivalent to the minimum of the function

$$E : \mathbb{R}^n \to \mathbb{R}, x \mapsto \frac{1}{2} \langle Ax, x \rangle - \langle b, x \rangle,$$

meaning x solves $Ax = b$ if and only if E has a global minimum at x.
To proof the equivalence calculate $\nabla E(x) = Ax - b$ and $\nabla^2 E(x) = A$. Therefore, $\nabla E(x_0) \overset{!}{=} 0 \Leftrightarrow Ax_0 = b$ and $\nabla^2 E(x_0)$ is positive definite, wherefore x_0 is a local minimum. This is the only extremum, so x_0 is also a global minimum. This equivalence is the basic idea of the conjugate gradient algorithm. Instead of solving a linear system in a typical way, we search the minimum of the function E. Let $x = x^{(0)} \in \mathbb{R}^n$ be an arbitrary start vector. We search the miminum of E on the line

$$g : \mathbb{R} \to \mathbb{R}^n, \alpha \mapsto x + \alpha p.$$

The search direction p is arbitrary for now. Let $r := b - Ax$ be the residual. With $A = A^t$ we get

$$\frac{dE\left(g(\alpha)\right)}{d\alpha} = -\langle r, p \rangle + \alpha \langle Ap, p \rangle$$

and with $dE\left(g(\alpha)\right)/d\alpha \overset{!}{=} 0$

$$\alpha = \frac{\langle r, p \rangle}{\langle Ap, p \rangle}. \tag{1}$$

That is a minimum, because A is positive definite and we get

$$\frac{d^2 E\left(g(\alpha)\right)}{d\alpha^2} = p^t A p > 0.$$

To obtain the minimum approximately we use an iterative search with different search directions. For that purpose set an arbitrary start vector x and calculate a more precise approximation of the minimum in every iteration

$$x^{(m)} \leftarrow x^{(m-1)} + \alpha^{(m)} p^{(m-1)}.$$

To calculate α we need r and p. The residuals $r^{(m)} = b - Ax^{(m)}$ can be computed iteratively using

$$r^{(m)} \leftarrow r^{(m-1)} - \alpha^{(m)} A p^{(m-1)},$$

because

$$\begin{aligned} & r^{(m-1)} - \alpha^{(m)} A p^{(m-1)} \\ = \ & b - A\left(x^{(m-1)} + \alpha^{(m)} p^{(m-1)}\right) \\ = \ & b - Ax^{(m)} \\ = \ & r^{(m)}. \end{aligned}$$

We get the gradient descent algorithm (see Algorithm 1), if we choose the direction of the steepest descent $p = -\nabla E(x)$ as our search direction (cf. [1]). We set $x^{(0)} = 0$ and get $r^{(0)} = b - Ax^{(0)} = b$.

```
$x^{(0)} \leftarrow 0$
$r^{(0)} \leftarrow b$
$p^{(0)} \leftarrow r^{(0)}$

for $m \leftarrow 1$ to $m_{max}$ do
    $\alpha^{(m)} \leftarrow \left\langle r^{(m-1)}, p^{(m-1)} \right\rangle / \left\langle Ap^{(m-1)}, p^{(m-1)} \right\rangle$
    $x^{(m)} \leftarrow x^{(m-1)} + \alpha^{(m)} p^{(m-1)}$
    $r^{(m)} \leftarrow r^{(m-1)} - \alpha^{(m)} A p^{(m-1)}$
    $p^{(m)} \leftarrow -\nabla E(x)$
return $x^{(m_{max})}$
```

Algorithm 1: Gradient descent.

The convergence of this algorithm is a problem, because this method uses search directions, which are similar to each other. Resulting we get an increased number of iterations to reach sufficient accuracy. So we use a set of linearly independent search directions (A-conjugated directions). This seems to be a good approach, because the algorithm is able to minimize in every direction of the space $\mathbb{R}^n$ in n steps.
Two vectors $x_i, x_j \in \mathbb{R}^n$ are called A-conjugated ($x_i \perp_A x_j$) to a symmetric, positive definite matrix $A \in \mathbb{R}^{n \times n}$,
if $\left\langle x_i, Ax_j \right\rangle = 0$ (cf. [1]). A set $\{r_1, r_2, ..., r_k\}$ with $r_1, r_2, ..., r_k \in \mathbb{R}^n$ and $x_i \perp_A x_j$ for all $i \neq j$ is linearly independent.
The proof is easy. For all $l \in \{1, 2, ..., n\}$ with $\forall i \neq l : r_l^t A r_i = 0$ is

$$\begin{aligned} & 0 \overset{!}{=} \sum_{i=1}^{k} \alpha_i r_i \\ \Rightarrow \quad & 0 \overset{!}{=} r_l^t A \sum_{i=1}^{k} \alpha_i r_i = r_l^t A \alpha_l r_l = \alpha_l (r_l^t A r_l) = 0 \\ \Rightarrow \quad & \alpha_l = 0, \end{aligned}$$

because A is positive definite and therefore $r_l^t A r_l \neq 0$.
The search directions can be created iteratively. With the approach

$$p^{(m+1)} = r^{(m+1)} + \beta^{(m+1)} p^{(m)} \qquad \text{and} \qquad p^{(0)} = r^{(0)}$$

and the request $p^{(m+1)} \perp_A p^{(m)}$ we get

$$\left\langle r^{(m+1)}, Ap^{(m)} \right\rangle + \beta^{(m+1)} \left\langle p^{(m)}, Ap^{(m)} \right\rangle \overset{!}{=} 0$$

and therefore

$$\beta^{(m+1)} = -\frac{\left\langle r^{(m+1)}, Ap^{(m)} \right\rangle}{\left\langle p^{(m)}, Ap^{(m)} \right\rangle}. \tag{2}$$

With this selection of $\beta^{(m+1)}$ we get a new search direction $p^{(m+1)}$, which is A-conjugated to the old direction $p^{(m)}$. Terms (1) and (2) can be written in the advantageous form

$$\alpha^{(m+1)} = \frac{\left\langle r^{(m)}, r^{(m)} \right\rangle}{\left\langle p^{(m)}, Ap^{(m)} \right\rangle} \text{ and } \beta^{(m+1)} = \frac{\left\langle r^{(m+1)}, r^{(m+1)} \right\rangle}{\left\langle r^{(m)}, r^{(m)} \right\rangle}$$

(cf. [11]). So we save the computation of a dot product. Using A-conjugated search directions we get the conjugate gradient method (see Algorithm 2 cf. [12]). The output of the algorithm is shown for $n = 2$ in Figure 1.

```
$x^{(0)} \leftarrow 0$
$r^{(0)} \leftarrow b$
$p^{(0)} \leftarrow r^{(0)}$

for $m \leftarrow 1$ to $n$ do
    $\alpha^{(m)} \leftarrow \left\langle r^{(m-1)}, r^{(m-1)} \right\rangle / \left\langle p^{(m-1)}, Ap^{(m-1)} \right\rangle$
    $x^{(m)} \leftarrow x^{(m-1)} + \alpha^{(m)} p^{(m-1)}$
    $r^{(m)} \leftarrow r^{(m-1)} - \alpha^{(m)} Ap^{(m-1)}$
    $\beta^{(m)} \leftarrow \left\langle r^{(m)}, r^{(m)} \right\rangle / \left\langle r^{(m-1)}, r^{(m-1)} \right\rangle$
    $p^{(m)} \leftarrow r^{(m)} + \beta^{(m)} p^{(m-1)}$
return $x^{(n)}$
```

Algorithm 2: Conjugate gradient (cg).

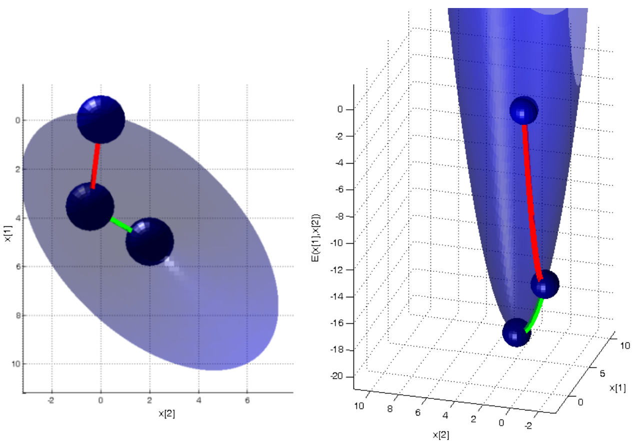

Figure 1: Cg-demo with $n = 2$.

$$A = \begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix}, b = \begin{pmatrix} 8 \\ -1 \end{pmatrix}, x^{(0)} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, x^{(1)} = \begin{pmatrix} 3.5616 \\ -0.4452 \end{pmatrix}, x^{(2)} = \begin{pmatrix} 5 \\ 2 \end{pmatrix}$$

Using induction shows, that these n search directions are pairwise A-conjugated and therefore pairwise linearly independent.
By using this result it is possible to show that with exact arithmetic the method finds the solution x of the linear system $Ax = b$ after at most n steps (cf. [1] and [12]).
The worst case runtime is $\mathcal{O}\left(n^3\right)$ FLOPS, because we need the matrix-vector multiplication $Ap^{(m-1)}$ with runtime $\mathcal{O}\left(n^2\right)$ in every iteration. The other operations can be realized in linear time. It is recommended to calculate only one matrix-vector multiplication per iteration and store the result.

For sparse matrices exist adapted data structures to accelerate the matrix-vector multiplication. With parallelization it is possible to decrease the runtime.

In some cases we need less than n iterations to get a precise approximation of the solution. We use the 2-norm of the residual after every iteration to decide if more iterations are necessary. The 2-norm can be derived from $\langle r, r\rangle$ which is already calculated to determine β.
There again it makes sense to run more than n iterations to minimize the numerical error in some cases.

4 Compute Unified Device Architecture (CUDA)

The parallelization of the algorithm is done using the NVIDIA CUDA technology. This technology makes it possible to exploit the massive compute power of todays GPUs for general purpose computing by creating kernel programs which execute in parallel.

4.1 CUDA-Programing Paradigm

Next the CUDA programming paradigm is reviewed, for additional details the reader is referred to [4] and [10]. In this paper the CUDA Toolkit 3.1 is used. As mentioned above using the CUDA technology we gain the possibility to exploit the massive compute power of modern GPUs by writing kernel programs e.g. in an extended version of C and execute these programs N times in parallel on the available hardware. The program is executed by N different threads while no assumption can be made in which order the threads are executed. Started from the CPU a kernel is attached to a compute grid, which is separated into a number of blocks. In each block a specific amount of threads is running (see Figure 2). The threads of a

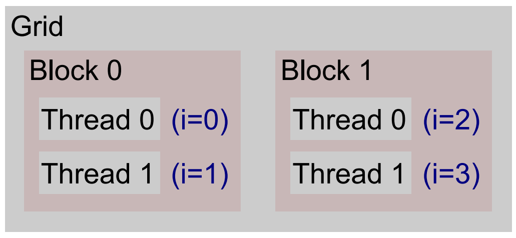

Figure 2: CUDA-programming model.

block are executed as packages of 16 threads which is also called halfwarp. The threads inside a halfwarp are generally running in parallel. The blocks of a compute grid are executed sequentially but in case of available hardware resources blocks are dispatched and executed in parallel to other blocks. Every thread has two information: first in which block it is running and secound by which block internal number it is identified. Hence the numbering of the threads can be done by

```
int i = blockIdx.x * blockDim.x + threadId.x.
```

This is also illustrated in Figure 2. The next example demonstrates the addition of two vectors implemented in C for CUDA (see Algorithm 3).

```
__global__ void VecAdd(float* A, float* B, float* C, int n){
    int i = blockIdx.x * blockDim.x + threadId.x;
    if (i < n) C[i] = A[i] + B[i];
}
```

Algorithm 3: CUDA-vector addition $C = A + B$ mit $A, B, C \in \text{float}^{256}$.

The data transfer between the CPU and GPU is done by *cudaMemcpy* which allows the transfer of data either from the CPU to the GPU or vice versa (*cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost*). Within a kernel program one can only operate on data available in the GPUs memory. Normally the data is first stored in the global persistent GPU memory. This memory can be allocated by *cudaMalloc* and deallocated by *cudaFree* similar to the C programming language. The global thread identifier is stored in the local thread memory which is only available during the life time of a thread. Additionally the hardware provides shared memory which can be accessed very fast compared to the data access in the global memory. A specific compute kernel is initiated by the CPU with the information how many blocks inside the compute grid should be allocated and how many threads per block shall be spawned (see an example for the provided vector addition kernel below).

```
VecAdd <<< blocksPerGrid, threadsPerBlock >>>
        (A, B, C, 256);
```

A synchronization of the GPU with the CPU is possible by using *cudaThreadSynchronize*.

5 Parallelized Conjugate Gradients

In every iteration of the cg-algorithm we have to compute several scaled vector additions, dot products and a matrix-vector multiplication (see Algorithm 2).

5.1 Parallel scaled Vector Addition

The λ-scaled vector addition

$$sum = x + \lambda y$$

of two vectors each with n components can be realized with n threads, which execute in parallel. Every thread calculates one component of the result. This method equates to Algorithm 3 with additional λ-scaling. If t threads run in parallel the runtime can be decreased from $\mathcal{O}(n)$ to $\mathcal{O}(n/t)$.

5.2 Parallel Dot Product

The calculation of the dot product

$$dot = \langle x, y \rangle$$

of two vectors each with n components is realized in two steps. First we have to calculate a vector

$$help = x. * y,$$

which contains the pointwise product of the vectors. This procedure is similar to Algorithm 3, but we have to replace the addition by a multiplication. The acceleration is analog a decrement from $\mathcal{O}(n)$ to $\mathcal{O}(n/t)$.
In the second step we have to sum up all components of the vector $help$. The complexity of the sequential method is in $\mathcal{O}(n)$. To parallelize the calculation we have to sum up all neighboring components first ($\approx n/2$ FLOPS) and repeat this kind of summation with the resulting sums. This procedure can be realized iteratively ($\approx \log_2 n + 1$ iterations) and is pictured in Figure 3. Let t be the number of

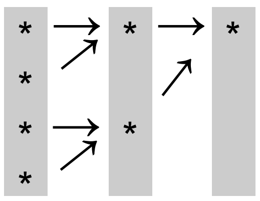

Figure 3: Parallel summation.

threads which run in parallel. Every thread calculates one sum of neighboring components. So the complexity for the second step and the whole dot product calculation is in $\mathcal{O}(n \log n/t)$.
If the number of threads is significantly lower it is more efficient to sum up not only two neighboring components. One option is that every thread has to add n/t elements first. So we get a new vector with t components, which has to be summed up. This can be done in $\mathcal{O}(\log t)$ and we get a complexity in $\mathcal{O}(n/t + \log t)$ for the second step and the dot product calculation. In this paper the first method is used.

5.3 Parallel Matrix-Vector Multiplication

The product of a $n \times m$-matrix and a vector with m components can be realized with n threads, where every thread calculates a component of the result.
This strategy is not efficient, if we want to calculate the product with a sparse matrix, because we execute many unessential operations with zeros. An approach to accelerate the calculation is using a special data structure to store the matrix.

5.3.1 ELLPACK-R-Data Structure

The classical formats to store a sparse matrix (coordinate storage, compressed column storage, compressed row

storage) are not applicable to parallelize the matrix vector product (e.g. [13]). In [4] an efficient storage for sparse matrices the ELLPACK format was suggested, which was extended to the ELLPACK-R format in [13] especially for the use on GPUs. A matrix $A \in \mathbb{R}^{n\times m}$ is represented by $\mathrm{Nz} \in \mathbb{N}$ the maximal number of elements unequal to zero per row, a representation matrix $\mathrm{A} \in \mathbb{R}^{n\times \mathrm{Nz}}$ for the elements unequal to zero, a representation matrix $\mathrm{j} \in {\mathbb{N}_0}^{n\times \mathrm{Nz}}$ for the indices of the elements and an information vector $\mathrm{rl} \in {\mathbb{N}_0}^n$ containing the number of the elements per row.
For

$$A = \begin{pmatrix} 1 & 3 & 0 \\ 0 & 1 & 1 \\ 4 & 0 & 0 \\ 0 & 0 & 2 \end{pmatrix} \in \mathbb{R}^{4\times 3}$$

with $\mathrm{Nz} = 2$ we have the following ELLPACK-R representation:

$$\mathrm{A} = \begin{pmatrix} 1 & 3 \\ 1 & 1 \\ 4 & * \\ 2 & * \end{pmatrix} \in \mathbb{R}^{4\times 2}, \mathrm{j} = \begin{pmatrix} 0 & 1 \\ 1 & 2 \\ 0 & * \\ 2 & * \end{pmatrix} \in {\mathbb{N}_0}^{4\times 2}, \mathrm{rl} = \begin{pmatrix} 2 \\ 2 \\ 1 \\ 1 \end{pmatrix} \in {\mathbb{N}_0}^4,$$

which has to be saved in column-major order. The $(*)$-elements are replaced with zeros. This representation is dense in many applications (e.g. discretized surfaces).

5.3.2 Matrix-Vector Multiplication

Let A be a matrix in ELLPACK-R representation. We realize the matrix-vector multiplication with n threads as seen in Algorithm 4.

```
**Input:** $A \in \mathbb{R}^{n\times m}$, $v \in \mathbb{R}^m$
**Output:** $u = Av \in \mathbb{R}^n$

for $threadIndex \leftarrow x \leftarrow 0$ to $n-1$ do in parallel
    $svalue \leftarrow 0$
    $max \leftarrow \mathrm{rl}[x]$
    for $i \leftarrow 0$ to $max - 1$ do
        $value \leftarrow \mathrm{A}[x + in]$
        $col \leftarrow \mathrm{j}[x + in]$
        $svalue \leftarrow svalue + value \cdot \mathrm{v}[col]$
    $\mathrm{u}[x] \leftarrow svalue$
```

Algorithm 4: Matrix-vector multiplication in ELLPACK-R format.

Let t be the number of threads which run in parallel. By using the ELLPACK-R format the runtime can be decreased from $\mathcal{O}(nm)$ to $\mathcal{O}(n\mathrm{Nz}/t)$.

6 Running Time

In this paper the cg-algorithm was parallelized with the described methods. It terminates after m_{max} iterations, if the 2-norm of the residual is less than a given upper bound of the error. By using the upper estimates for the parallelized operations we get a runtime in

$$\mathcal{O}\left(\frac{m_{max}}{t} n(Nz + \log n)\right),$$

in which n is the dimension of the coefficient matrix with at most Nz elements unequal to zero per row and t denotes the threads running in parallel. This promises a significant acceleration compared to the runtime of the sequential implementation, which is in $\mathcal{O}\left(m_{max} n^2\right)$.

6.1 Runtimes of the Algorithm

This section provides a comparison of the runtimes of the sequential and the parallel implementation of the conjugate gradient algorithm. As an application the heat equation is solved and different sizes of the $n \times n$-coefficient matrix are chose. When solving a linear system the convergence speed of the conjugate gradient algorithm to compute an acceptably accurate solution strongly depends on the condition number of the system matrix. Therefore, the time per iteration is used for detailed comparison. The table below presents this times (in ms) of different CPU implementations (Armadillo, MATLAB, MKL), the runtimes of the presented GPU version in CUDA 3.1 (highlighted in orange) and a CUDA 3.2 implementation from NVIDIA.

n	CPU	Matl.	MKL	Cu. 3.1	Cu. 3.2
512^2	12.9	13.3	3.4	3.1	3.0
1024^2	56.6	40.6	19.4	7.3	4.6
2048^2	238.6	153.0	79.9	21.5	9.2

The second column (CPU) contains the runtimes of a sequential CPU implementation. This implementation uses the Armadillo C++ Library 1.0.0 (based on LAPACK) for scaled vector additions and dot product calculations. The matrix is stored in the ELLPACK-R format to realize an efficient matrix-vector multiplication.
Intel®MKL 10.3 (Math Kernel Library) is a library of threaded math routines, which contains an implementation of the conjugate gradient algorithm using the compressed row storage (CRS) format to store the matrix. The convergence tolerance from the correct solution is set to 1.0 regarding the 2-norm of the residual. The implementations were tested on a PC with Intel® Core™ i7 860 (2.80 Ghz) and NVIDIA® GeForce® GTX 480 graphics card.
In the CUDA 3.1 implementation the number of threads per block was set to 512, because on the used hardware the configuration was always optimal. This is shown in Figure 4 in case of the $n = 512^2$ matrix.
For the implementation of the conjugate gradient solver, which is described in this paper the CUDA Toolkit 3.1 was used. NVIDIA has now published the version 3.2, which includes a conjugate gradient solver using the CRS format. This solver is in case of large matrices about a factor of two faster then the described implementation.
Additionally it should be mentioned that the data transfer

time to the GPU memory is compensated when using coefficient matrices with dimension $n > 1000$.
Collectively, we can say that in comparison to the sequential CPU implementation (MATLAB) a speed up between 4 to 7 is reached with the presented implementation for the tested coefficient matrices depending on the size. For larger coefficient matrices additional speed up is expected.

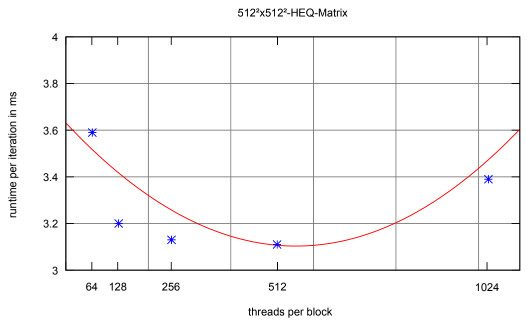

Figure 4: Dependence of the runtime of the number of threads per block.

7 Application: Heat Equation

In this paper the parallel cg-algorithm was tested on solving the heat equation. Let G be a $n \times m$ grid (see Figure 5) and let $u(x,y,t)$ be the temperature at a given time t at the point $(x,y) \in G$. The temperature gradation on the grid is in case of diffusion represented by the differential equation

$$\frac{\partial u(x,y,t)}{\partial t} = \frac{\partial^2 u(x,y,t)}{\partial x^2} + \frac{\partial^2 u(x,y,t)}{\partial y^2}.$$

7.1 Discrete Heat Equation

Using finite differences the discretization of the heat equation is given by

$$\begin{aligned} u(x,y,t) = (1-4\tau)\; & u(x,y,t+\Delta t) \\ +\tau \quad & u(x+\Delta x,y,t+\Delta t) \\ +\tau \quad & u(x-\Delta x,y,t+\Delta t) \\ +\tau \quad & u(x,y+\Delta y,t+\Delta t) \\ +\tau \quad & u(x,y-\Delta y,t+\Delta t) \end{aligned}$$

in the case of $\Delta x = \Delta y$ and $\tau := -\Delta t/\Delta x^2$. A problem occurs if a discretized point (x,y) is located on the boundary of the grid, because such a point is located in a neighborhood consisting of only two or three points. In such a case the term for the missing grid point has to be removed. The equation above can then be written as a linear equation system and is shown in Figure 5 for $n = m = 4$ with $\bar{\tau} := 1 - 4\tau$. The coefficient matrix of the system $A = A(G)$ can be represented as

$$\left|\begin{array}{cccc|cccc|cccc|cccc}
\bar\tau & \tau & & & \tau & & & & & & & & & & & \\
\tau & \bar\tau & \tau & & & \tau & & & & & & & & & & \\
& \tau & \bar\tau & \tau & & & \tau & & & & & & & & & \\
& & \tau & \bar\tau & \tau & & & \tau & & & & & & & & \\
\hline
\tau & & & \tau & \bar\tau & \tau & & & \tau & & & & & & & \\
& \tau & & & \tau & \bar\tau & \tau & & & \tau & & & & & & \\
& & \tau & & & \tau & \bar\tau & \tau & & & \tau & & & & & \\
& & & \tau & & & \tau & \bar\tau & \tau & & & \tau & & & & \\
\hline
& & & & \tau & & & \tau & \bar\tau & \tau & & & \tau & & & \\
& & & & & \tau & & & \tau & \bar\tau & \tau & & & \tau & & \\
& & & & & & \tau & & & \tau & \bar\tau & \tau & & & \tau & \\
& & & & & & & \tau & & & \tau & \bar\tau & \tau & & & \tau \\
\hline
& & & & & & & & \tau & & & \tau & \bar\tau & \tau & & \\
& & & & & & & & & \tau & & & \tau & \bar\tau & \tau & \\
& & & & & & & & & & \tau & & & \tau & \bar\tau & \tau \\
& & & & & & & & & & & \tau & & & \tau & \bar\tau
\end{array}\right|$$

In this case the red elements in the cofficient matrix cause a wrap around from the and the left-hand side and will be removed.

For larger grid sizes the coefficient matrices can be derived analogously. From the previous statements a discretization of the heat equation can be done using finite differences. The computation of the heat dissipation is done by solving a linear system of size $n \cdot m$ (cf. Figure 5) starting from a given initial configuration $u(0) \in \mathbb{R}^{n \cdot m}$. The resulting coefficient matrix is symmetric and positive definite, wherefore the solution can be computed using the conjugate gradient algorithm. From a current heat dissipation the previous dissipation can be computed by multiplying the coefficient matrix with the current result.

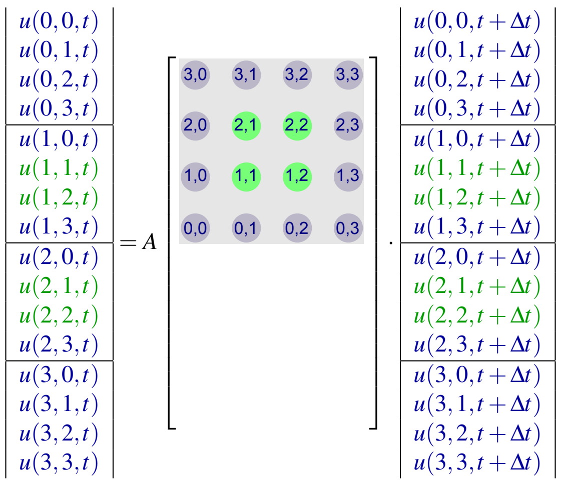

Figure 5: Discretization of a 4×4-grid.

7.2 Implementation

The presented parallel conjugate gradient algorithm was used to solve the 2D heat equation (see Figure 6). The implementation was tested (hardware specification above) with square images with different sizes 256, 512 and 1024

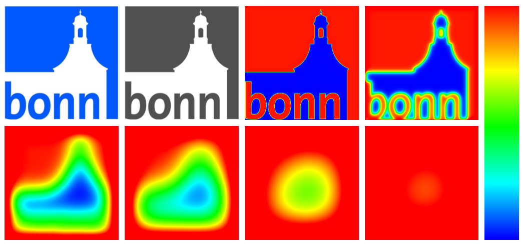

Figure 6: Heat distribution on the 256×256-emblem of the University of Bonn.

resulting in simulating the heat flow in 20, 10 and 7 time steps per second (interactive frame rates).

8 Future Work

In this paper, an accelerated cg-algorithm was implemented. To reduce the processing time the coefficient matrix was stored in ELLPACK-R format and the matrix vector operations were parallelized.
Another approach to advance stability and acceleration is the preconditioning of the matrix. A sequential implementation of the preconditioned conjugate gradient algorithm (pcg) was implemented in MATLAB to evaluate this.
Furthermore, the use of Shared Memory and Texture Cache could accelerate the runtime on older graphics cards.

8.1 Precondition and SSOR

One approach to advance stability and acceleration is the preconditioning of the coefficient matrix A. The cg-algorithm converged much faster for coefficient matrices with smaller condition number. For example this is shown in [12].
We can use this by left multiplication of the matrix $A \in \mathbb{R}^{n\times n}$ with a preconditioning matrix M^{-1}, in which $M \in \mathbb{R}^{n\times n}$ is symmetric and positive definite. We have to choose M in a way, where $\kappa(M^{-1}A) << \kappa(A)$. So we can solve the equivalent linear system $M^{-1}Ax = M^{-1}b$ instead of $Ax = b$. But a problem occurs, if we want to use the cg-algorithm to solve it, because the new coefficient matrix $M^{-1}A$ is in general not symmetric and positive definite. So we have to decompose the matrix with the Cholesky method and get $M = M_1M_2$ called the left and the right preconditioning matrix and execute the cg-algorithm on the linear system

$$\underbrace{M_1^{-1}AM_2^{-1}}_{=:\widetilde{A}}M_2x = \underbrace{M_1^{-1}b}_{\widetilde{b}}.$$

That is possible, because $\widetilde{A}$ is symmetric and positive definite. So we can solve the linear system $Ax = b$ in two steps. First we have to solve $\widetilde{A}y = \widetilde{b}$ with the cg-method. After that we can get the solution x by solving $M_2x = y$.
The second step is realizable with back substitution, because M_2 is an upper triangular matrix.
If we do it that way we have to calculate the matrix product $\widetilde{A} = M_1^{-1}AM_2^{-1}$ first. This would be a kind of a bottleneck. To avoid this we can use the substitution $\hat{x} = M_2x$, $\hat{r} = M_1^{-1}r$ and $\hat{p} = M_2P$ in Algorithm 2. This way we get the simplified preconditioned conjugate gradient algorithm (see Algorithm 5, c.f. [11]).

```
$x^{(0)} \leftarrow 0$
$r^{(0)} \leftarrow b$
$p^{(0)} \leftarrow M^{-1}r^{(0)}$

**for** $m \leftarrow 1$ **to** $n$ **do**
    $\alpha^{(m)} \leftarrow \left\langle r^{(m-1)}, M^{-1}r^{(m-1)}\right\rangle / \left\langle p^{(m-1)}, Ap^{(m-1)}\right\rangle$
    $x^{(m)} \leftarrow x^{(m-1)} + \alpha^{(m)}p^{(m-1)}$
    $r^{(m)} \leftarrow r^{(m-1)} - \alpha^{(m)}Ap^{(m-1)}$
    $\beta^{(m)} \leftarrow \left\langle r^{(m)}, M^{-1}r^{(m)}\right\rangle / \left\langle r^{(m-1)}, M^{-1}r^{(m-1)}\right\rangle$
    $p^{(m)} \leftarrow M^{-1}r^{(m)} + \beta^{(m)}p^{(m-1)}$
**return** $x^{(n)}$
```

Algorithm 5: Preconditioned conjugate gradient (pcg).

In this paper the preconditioned algorithm was tested with the SSOR method (Symmetric Successive Overrelaxation, cf. [3]) in MATLAB. This method uses the strict lower triangular matrix L and the diagonal matrix D of A.
The SSOR preconditioning matrix

$$M := C^{-1} = (D+L)D^{-1}(D+L)^t$$

is an approximation of the coefficient matrix A.
In every iteration we have to solve the linear system

$$C^{-1}z = r^{m-1}$$

to get $z = Cr^{m-1} = M^{-1}r^{m-1}$.
In the MATLAB implementation the decomposition $C^{-1} = KK^t$ with $K = (D+L)D^{-1/2}$ was used to solve the system with forward and back substitution.
A disadvantage of SSOR is the use of $D^{-1/2}$, which is only available for matrices with a positive main diagonal. All elements of $D+L$ are included in A. So the required memory is less.
The implementation shows a convergence in mind of the number of required iterations to get a precise approximation of the solution, which is significant faster compared to the unpreconditioned method. But the algorithm solves an linear system in every iteration, for which reason the total runtime is only faster for large ($n > 1000$) and bad conditioned matrices. Another aspect is the enhanced stability because of the decreased condition number.
This concept could be another approach to accelerate the parallelized algorithm. For this purpose it is essential to realize the solution of the linear system $Mz = r^{m-1}$ efficiently, for example with parallelization. The parallelization of the back substitution is still an unsolved topic in science.

8.2 Shared Memory and Texture Cache

In the implementation, which is described in this paper the input data were copied into the global memory of the graphics card. During the execution, the threads get the required data only from this memory, which is about 512 to 2048 MB on modern graphics hardware. There exists also a comparatively small shared memory (it can also be configured as a L1-cache) and texture cache, which is only about a few KB. The access time to this kind of memory is much shorter (sometimes about a factor in the dimensions of 100) in comparison to the often uncached global memory (cf. [13]). Because of the small size in the most cases it is not possible to store the whole coefficient matrix inside of it.
Another way to speed up the process is the skillful use of this memory. Therefore, it is necessary to copy parts of data for future calculations from global memory to shared memory and texture cache. In order to achieve an acceleration, this must be constructed so that most of these transfers occur in parallel to the running threads.

9 Conclusion

This work presents a parallel implementation of the conjugate gradient algorithm using the NVIDIA CUDA architecture. The operations which were parallelized are the scaled vector addition, the dot product and the sparse matrix-vector multiplication. As sparse matrix format ELLPACK-R was used which provides an memory efficient storage for large coefficient matrices on the GPU. To compare the presented implementation, a comparison with a sequential CPU implementation using the LAPACK-based Armadillo C++ Library 1.0.0 and different CPU implementations (MATLAB, MKL) was made. In comparison to the sequential CPU implementations the parallel version of the conjugate gradient algorithm is in average 4 to 7 times faster. The speed-up of the algorithm is further increased if larger coefficient matrices are used. The transfer time between the CPU and the GPU will be compensated if the system matrices are acceptably large, in the presented case $n > 1000$. An additional speed up can further be gained if a preconditioner is used (e.g. SSOR). In this case in each iteration an additional equation system must be solved by back subsitution but currently there exists no efficient solution to that problem.

References

[1] G. Alefeld, I. Lenhardt, and H. Obermaier. *Parallele numerische Verfahren*. Springer, New York, Berlin, Heidelberg, 2002.

[2] M. Ament, G. Knittel, D. Weiskopf, and W. Strasser. A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-gpu platform. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0:583–592, 2010.

[3] M. Ament, G. Knittel, D. Weiskopf, and W. Strasser. A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-gpu platform. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 583–592, Washington, DC, USA, 2010. IEEE Computer Society.

[4] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[5] J. Bolz, I. Farmer, E. Grinspun, and P. Schrï¿½der. Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. *ACM TRANSACTIONS ON GRAPHICS*, 22:917–924, 2003.

[6] J. Georgii and R. Westermann. A multigrid framework for real-time simulation of deformable volumes. In *Proceedings of the 2nd Workshop On Virtual Reality Interaction and Physical Simulation*, pages 50–57, 2005.

[7] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49:409–436, 1952.

[8] H. Hnaidi, E. Guérin, S. Akkouche, A. Peytavie, and E. Galin. Feature based terrain generation using diffusion equation. *Computer Graphics Forum*, 29(7):2179–2186, 2010.

[9] L. Liu, L. Zhang, Y. Xu, C. Gotsman, and S. J. Gortler. A local/global approach to mesh parameterization. In *Proceedings of the Symposium on Geometry Processing*, SGP '08, pages 1495–1504, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[10] NVIDIA-Corporation. Nvidia cuda c programming guide, Version 3.1, 2010.

[11] Y. Saad. *Iterative methods for sparse linear systems*. Second edition, 2003.

[12] L. N. Trefethen and D. Bau. *Numerical Linear Algebra*. SIAM: Society for Industrial and Applied Mathematics, 1997.

[13] F. Vazquez, E. M. Garzon, J. Martinez, and J. J. Fernandez. The sparse matrix vector product on gpus. *aceuales*, pages 1–13, 2009.