Portierung von deal.II auf CUDA

Praktikum HPC-GPU Computing

Mohammed Ibrahim
Betreuer: Dipl.-Phys. Dipl.-Math. S. Kramer

Einleitung

Um deal.II Solver Klassen mit CUDA zu verwenden, werden Vektoren, die speziell für CUDA implementiert sind, benötigt. Die CUDA Vektoren müssen die Vektor-Schnittstelle von deal.II erfüllen, also alle benötigten Methoden implementieren.
Die deal.II Vektorschnittstelle sieht wie folgt aus:

```
class VECTOR {
        public:

        void reinit (const VECTOR&, bool leave_elements_uninitialized = false);

        double operator * (const VECTOR &v) const;

        void add (const VECTOR &x);

        void add (const double a, const VECTOR &x);

        void sadd (const double a, const double b, const VECTOR &x);

        void equ (const double a, const VECTOR &x);

        VECTOR & operator *= (const double a);

        double l2_norm () const;
 };
```

Des Weiteren müssen die Speicherformatklassen die Matrix-Schnittstelle aus deal.II erfüllen:

```
class Matrix {

        public:
        void vmult (VECTOR &dst, const VECTOR &src) const;

        void Tvmult (VECTOR &dst, const VECTOR &src) const;

};
```

Ziel

Ziel von Step-6 war die Implementierung einer Vektorklasse, die die Vektoroperationen mit Hilfe von CUDA auf der GPU ausführt.
Dabei sollten nur die von den deal.II Solver-Klassen benötigten Methoden implementiert werden.
Die Hauptschwierigkeit bestand in der korrekten Allokation auf dem Grafikkartenspeicher.

Implementierung

Bei der Implementierung werden die Vektorwerte, seien dies Standard- oder deal.II Vektoren, die auf dem Host liegen, auf die Grafikkarte gespiegelt, indem Zeiger auf diese Daten der GPU übergeben werden und damit auch Speicher auf der GPU allokiert wird.

Des Weiteren werden Kernel für die elementaren Vektoroperationen implementiert, die dies auf der GPU ausführen. Die Ergebnisse werden dann auf den Host zurück kopiert, um die Weiterbearbeitung bzw. Ausgabe zu ermöglichen.

Beispiel Kernel: Addition

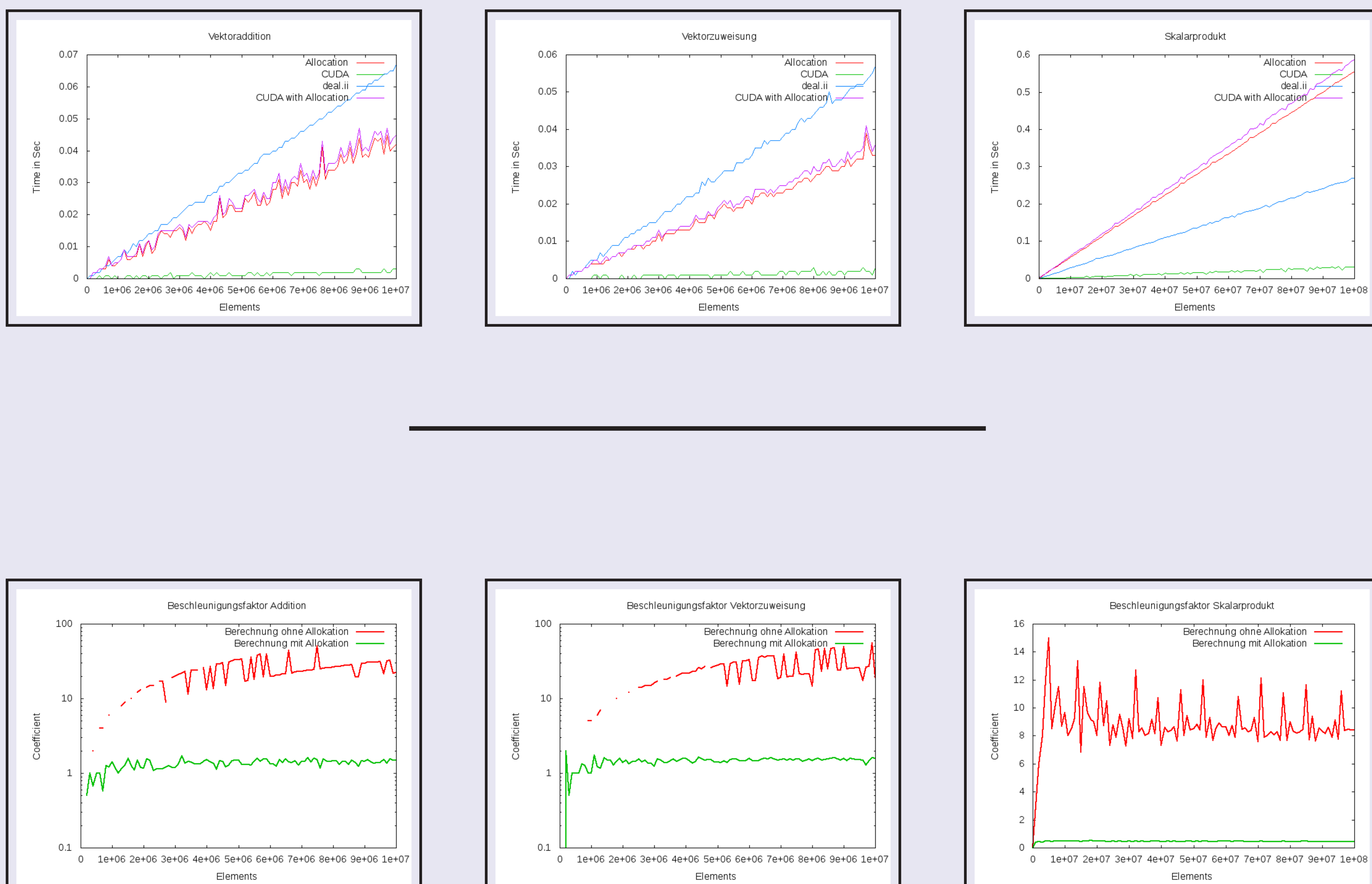
```
template <typename T>
__global__ void VecAdd(T* A, T* B, int N)
{
    int threadId = threadIdx.x + blockDim.x * blockIdx.x;
    int gridSize = gridDim.x * blockDim.x;

    for(int i = threadId; i<N; i += gridSize){
    A[i] += B[i];
    }
}

// gekapselter Kernelaufruf
void call_kernel_VecAdd(int __NUM_BLOCKS, int BLOCK_SIZE, double* A, double* B,
                        int N){
    VecAdd<<<__NUM_BLOCKS, BLOCK_SIZE>>>(A, B, N);
    cudaThreadSynchronize();
}
```

Aufruf aus C++ Programm :

```
// Treiber fuer den GPU-Teil
#include "cuda_driver.h"
#include <cutil_inline.h>
#include "cuda_driver.cu.hh"
...

// L&ouml;ser aus deal.II mit GPU-Vektor initialisieren
SolverCG<CUDAVectorView<double> > cg (solver_control);

 // ... um damit die GPU-seitigen Vektoren zu initialisieren
CUDAVectorView<double> solview(solution);
CUDAVectorView<double> rhsview(system_rhs);

// ... um unsere GPU-seitige Matrix damit zu initialisieren
CudaEllItMatrixView<double> sm_cuda_view(system_matrix);

// ... PC initialisieren (geht genauso)

// Problem l&ouml;sen
cg.solve (sm_cuda_view, solview, rhsview, preconditioner_cuda);

solview.copyToHost();
```

Ergebnisse

Im Vergleich zur CPU Lösung mit deal.II, können folgende Resultate für elementare Vektoroperationen betrachtet werden:



Beobachtungen

Da die Klasse CUDAVectorView als Grundlage für folgenden Klassen aus step-7 und step-8 dient, können die Resultate bei den Klassen CudaCSRMatrixView, CudaEllItMatrixView und CudaEllRMatrixView betrachtet werden.

- Anhand der Abbildungen erkennt man, dass die Rechenzeit linear zur ProblemgrÃ§e wächst.
- Im Falle der Vektoraddition und der Zuweisung wird der Vorteil von CUDA recht deutlich im vergleich zur CPU-Lösung. So erhält man doppelt so schnell eine Lösung, betrachtet man die allokation jedoch nicht so hat man einen perfromance Vorteil von faktor 6.
- Im Falle der Berechnung des Skalarprodukts wird deutlich, dass die Rechenzeit mit CUDAVectorView Vektoren viel niedriger ist als mit deal.II, beachtet man aber die Allokationzeit, so ist CUDAVectorView weniger effizient im Vergleich zur deal.II.
  Da die Vektoren nur einmal allokiert werden, kann man davon ausgehen, dass bei Berechnungen, die immer wieder die selben Vektoren benutzen, mit CUDA die schnellere Rechenzeit erhält.

Literatur

- Yousef Saad - Iterative methods for sparse linear systems (2nd edition).
- Numerische Mathematik, interaktiv Uni-Wien (http://www.dorn.org/uni/sls/index.html).
- deal.II Dokumentation (http://www.dealii.org/6.2.1/doxygen/tutorial/index.html).