Parallel Implementation of Classical Gram-Schmidt Orthogonalization on CUDA Graphics Cards

Benjamin Milde and Michael Schneider

Technische Universität Darmstadt
Department of Computer Science
milde@stud.tu-darmstadt.de

1 Introduction

The Gram-Schmidt orthogonalization process is a fundamental algorithm in linear algebra [TB97]. It is a method for orthogonalizing a set of vectors in the Euclidean space. It can be used to do a QR factorization [DGKS76] efficiently, that is a factorization of a matrix into an orthogonal and a triangular matrix. Two main computational variants exist, the classical Gram-Schmidt (CGS) and the modified Gram-Schmidt (MGS) [Lon81]. In practical applications MGS is often chosen, because it is numerically more stable than CGS. However, the latter algorithm is well suited for a parallel version [YTBS06]. Our results show that an efficient parallel implementation of CGS is also possible on current Graphics Cards.

2 QR-Factorization with classical Gram-Schmidt

Let $\mathbf{V}$ be a square matrix of linear $n \times n$ vectors $\vec{v_1}, \dots, \vec{v_n} \in \mathbb{R}^n$, $\mathbf{U}$ a square matrix with $\vec{u_1}, \dots, \vec{u_n} \in \mathbb{R}^n$ and $\mu \in \mathbb{R}^{n \times n}$ a triangular matrix. The CGS algorithm, presented as Algorithm 1, uses $\mathbf{V}$ to compute $\mathbf{U}$ and μ so that $\mathbf{V} = \mathbf{U} \cdot \mu$ and $\langle \vec{u_j}, \vec{u_k} \rangle = 0 \; \forall 1 \leq j, k \leq n; j \neq k$.

Algorithm 1: Algorithm for QR-Factorization with classical Gram-Schmidt

```
  Input: $\vec{v_1}, \dots, \vec{v_n} \in \mathbb{R}^{n \times n}$
1 $\mu \leftarrow I_n$
2 $\vec{u_1} \leftarrow \vec{v_1}$
3 for $m := 2$ to $n$ do
5     $\vec{u_m} \leftarrow \vec{v_m} - \sum_{i=1}^{m-1} \mu_{m,i} \cdot \vec{u_i}$,
6     where $\mu_{m,i} \leftarrow \frac{\langle \vec{u_i}, \vec{v_m} \rangle}{\langle \vec{u_i}, \vec{u_i} \rangle}$
7 end
  Output: $(\vec{u_1}, \dots, \vec{u_n}) \in \mathbb{R}^{n \times n}$, matrix $\mu \in \mathbb{R}^{n \times n}$
```

3 Parallel QR-Factorization with Classic Gram-Schmidt on Graphics Cards

The algorithm can be parallelized in two ways. First of all, the inner product of two vectors is computed very frequently. According to the NVIDIA reduction paper [Har07], this can be best parallelized in a tree like manner using shared memory. A good starting point is to use $\frac{n}{2}$ threads that compute the product in $\log(n)$ parallel steps. See the NVIDIA reduction paper [Har07] for further details. When computing sums with the parallel reduction algorithm in floating point, the accuracy rises compared to the normal single algorithm. This is because on average more numbers of the same magnitude are summed up.

The crucial idea for the parallel CGS is to transform it into a more suitable version (see Algorithm 2). Instead of calculating every vector $\vec{u}$ one after the other, one can subtract one term of the sum in Algorithm 1 line 5 for all vectors at once. This yields one new completed vector $\vec{u}$ after one such step, that is needed for the next iteration. It also means that the degree of parallelization degrades after each step, with an average of $\frac{n}{2}$ parallel operations at once. Global barrier synchronization must be used to guarantee that each new vector $\vec{u}$ is available in the next iteration. Both methods combined allow a parallel version with a parallelization degree of approximately $\frac{n^2}{4}$. This is sufficient to guarantee a good utilization on current graphics cards.

Algorithm 2: Algorithm for parallel Gram-Schmidt

```
   Input: $\vec{v_1}, \ldots, \vec{v_n} \in \mathbb{R}^{n \times n}$
1  $\vec{u_i} \leftarrow \vec{v_i} \; \forall i = 1 \ldots n$
2  for $m := 2$ to $n$ do
3      invoke $n - m + 1$ blocks with $2^c$ threads
4      for each thread do
5          calculate offsets for the vectors $\vec{u}$ and $\vec{v}$ according to block number and m
6          $sharedmemory[threadnumber] \leftarrow u[threadnumber]^2$
7          synchronize all threads in a block (barrier synchronization)
8          do sum reduction in shared memory
9          $usquare \leftarrow sharedmemory[0]$
10         $sharedmemory[threadnumber] \leftarrow u[threadnumber] \cdot v[threadnumber]$
11         synchronize all threads in a block
12         if $threadnumber = 0$ then
13             $sharedmemory[0] \leftarrow sharedmemory[0]/usquare$
14             $\mu$(offset of $\vec{v}$, m) $\leftarrow sharedmemory[0]$
15         end
16         synchronize all threads in a block
17         $u[threadnumber] \leftarrow sharedmemory[0] \cdot u[threadnumber]$
18     end
19 end
   Output: $(\vec{u_1}, \ldots, \vec{u_n}) \in \mathbb{R}^{n \times n}$, matrix $\mu \in \mathbb{R}^{n \times n}$
```

4 Results

We implemented Algorithm 2 with CUDA in C. The algorithm given resembles very closely the structure and synchronization methods that must be used in CUDA. It should be noted that CUDA can only use barrier synchronization in a block, not for the whole card. In order to achieve a global synchronization, one must invoke a new set of blocks and threads each time, called kernel.

Also one can see that in line 3 only 2^c threads, where c is some constant, can be invoked at a time. This is because of the tree like nature of the parallel sum reduction part. This restricts the dimension of the input matrix to certain sizes. One can extend an arbitrary sized matrix to the next n that matches 2^c and this can be relaxed further to $c_1 \cdot 2^{c_2}$ by doing c_1 multiplications for one shared memory entry.

The implementation was tested on a workstation with a 3,0 GHz Intel CPU, and a Geforce 295. The Geforce 295 is actually two cards in one, each comparable to a Geforce 275 in terms of speed. Algorithm 2 can only be applied to one card. A multi-card approach would produce a lot of communication between the cards, because the input matrix cannot be divided into independent parts. This would outweigh any speed benefits.

For testing purposes we used random integer matrices with entry size up to 10^6. We converted them to double precision floating points and measured the time it takes for the whole computation (including memory transfers for the Graphics Card). A speed-up of approximately 15x to an unoptimized single core reference implementation and a speed up of 3x to the Gram-Schmidt code in the library fpLLL [CPS] can be observed at dimension $n = 2048$. It should be noted that fpLLL only generates one output-matrix and does not save the values for μ, which are needed for a QR-factorization. That and the fact that it was only a reference implementation for comparing the results from the graphics card contribute to the problem that it is many times slower than the fpLLL code. Figure 1 and 2 present the results.

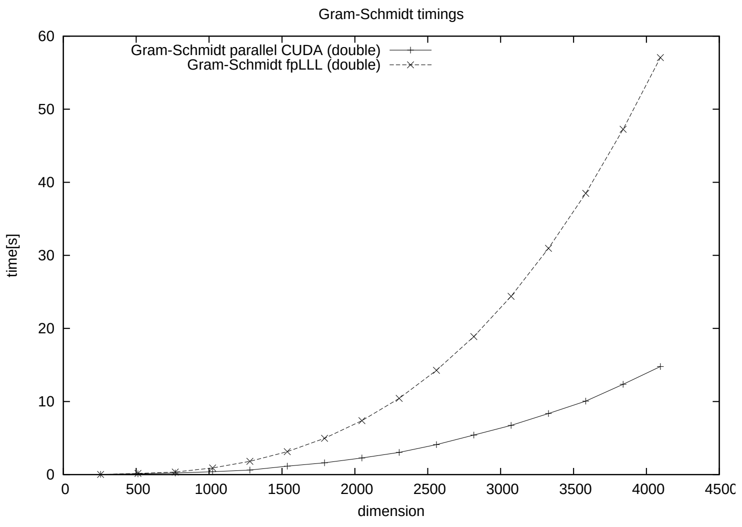

Fig. 1. cudaGramschmidt compared to fpLLL.

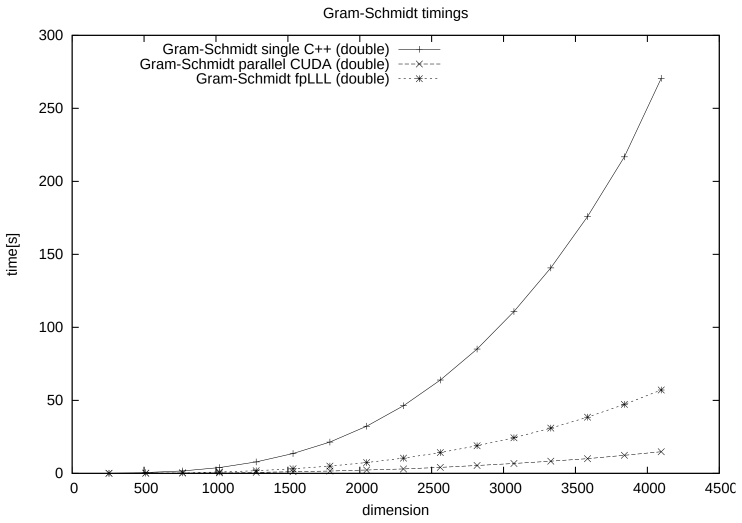

Fig. 2. cudaGramschmidt compared to fpLLL and the own reference implementation.

Floating points are used to approximate the values in the output matrices. On all three implementations double precision (64-bit) was chosen. Current graphics cards are capable of calculating doubles natively, but at a slower speed than floats. On the other hand the precision of the standard float (32-bit) does not suffice for the big dimensions needed to exploit parallelism on the graphics card. Even on smaller dimensions (for example $n = 128$) 32-bit floating point is not enough to compensate the numerical problems of the Classical Gram-Schmidt. However, newer NVIDIA graphics cards will have a better double precision support. A big speed-up with our implementation on them, since it relies heavily on the double performance, will be noticed.

References

[CPS] David Cadé, Xavier Pujol, and Damien Stehlé. fpLLL - a floating point LLL implementation. Available at Damien Stehlé's homepage at école normale supérieure de Lyon, `http://perso.ens-lyon.fr/damien.stehle/english.html`.

[DGKS76] J. Daniel, W. B. Gragg, L. Kaufman, and G. W. Stewart. Reorthogonalization and stable algorithms for updating the Gram–Schmidt QR factorization. *Mathematics of Computation*, 30:772–795, 1976.

[Har07] Mark Harris. Optimizing parallel reduction in CUDA, 2007. `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf`.

[Lon81] J. W. Longley. Modified Gram-Schmidt process vs. classical Gram-Schmidt. *Comm. Statist. Simulation Comput.*, B10, 5:517–527, 1981.

[TB97] Lloyd N. Trefethen and David Bau. *Numerical linear algebra.* Society for Industrial and Applied Mathematics, 1997.

[YTBS06] Takuya Yokozawa, Daisuke Takahashi, Taisuke Boku, and Mitsuhisa Sato. Efficient parallel implementation of classical Gram-Schmidt orthogonalization using matrix multiplication. In *Parallel Matrix Algorithms and Applications — PMAA 2006*, pages 37–38, 2006.