Finding Enclosures for Linear Systems using Interval Matrix Multiplication in CUDA

Alexander Dallmann, Philip-Daniel Beck, and Jürgen Wolff von Gudenberg

University of Würzburg, Chair of Computer Science II
Am Hubland, D 97074 Würzburg, Germany
alexander.dallmann@uni-wuerzburg.de

Abstract. In this paper we present CUDA kernels that compute an interval matrix product. Starting from a naive implementation we investigate possible speedups using commonly known techniques from standard matrix multiplication. We also evaluate the achieved speedup when our kernels are used to accelerate a variant of an existing algorithm that finds an enclosure for the solution of a linear system. Moreover the quality of our enclosure is discussed.

Keywords: GPGPU, Interval arithmetic, Linear algebra, Parallel computing

1 Introduction

Today graphics cards like the NVIDIA Tesla series are used in workstations as well as supercomputers to speed up computations using the highly parallel execution model of the GPU architecture. Especially linear algebra routines like matrix computations can be accelerated by outsourcing the computation to the GPU.

In this paper we present GPU routines to carry out interval matrix computations, as well as routines that perform a real matrix product with directed rounding. Also a routine for matrix computation in two-fold working precision is implemented using error-free transformations [1]. We then show how those routines can be applied to speed up a variant of an existing algorithm [2] for computing an enclosure for the solution of a linear system.

2 Related Work

In [2] the implementation of an algorithm that finds an enclosure for the solution of a linear system is described. The computation of a matrix-matrix product on CUDA in two-fold working precision is discussed in [1]. General optimizations for CUDA Kernels with matrix multiplication as an example are formulated in [3]. [4] discusses a parallel variant of the Interval Newton Method on CUDA.

3 Preliminaries

3.1 Notation

Throughout the paper we will denote intervals by [], e.g. $[x]$. Interval vectors and matrices will be written with a bold letter, e.g. $[\boldsymbol{x}]$ or $[\boldsymbol{A}]$. For an indepth covering of interval arithmetic we refer to [5] and [6]

3.2 Computing a Verified Enclosure of a Linear System

Finding the solution of a real linear system $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$, with $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ and $\boldsymbol{b}, \boldsymbol{x} \in \mathbb{R}^n$, is a common numerical problem used in various applications. The result $\tilde{\boldsymbol{x}}$ of a numerical algorithm is usually some approximation of the real solution, having some unknown error term $\boldsymbol{e}$, so that $\boldsymbol{x} = \tilde{\boldsymbol{x}} + \boldsymbol{e}$. Using interval arithmetic an algorithm that finds verified enclosures of the solution $\hat{\boldsymbol{x}}$ can be implemented. The verified enclosure itself is obtained by applying Brouwer's fixed-point theorem.

The method we adapted is described in [2], where more details are given. It uses a Newton-like method to find an enclosure $[\boldsymbol{y}]$ for the residual of $\boldsymbol{Ax}$. To do so, the iteration scheme

$$\boldsymbol{y^{k+1}} = \underbrace{\boldsymbol{Rd}}_{\boldsymbol{z}} + \underbrace{(\boldsymbol{I} - \boldsymbol{RA})}_{\boldsymbol{C}} \boldsymbol{y^k}, k = 0, 1, ...$$

is used, with $\boldsymbol{R}$ being an approximate inverse of $\boldsymbol{A}$. By replacing all iterates with interval vectors, a result from [7] can be applied for this equation, that says that if $[\boldsymbol{y}]^{\boldsymbol{k}} \overset{\circ}{\subset} [\boldsymbol{y}]^{\boldsymbol{k+1}}$ holds for any index k, $\boldsymbol{R}$ and $\boldsymbol{A}$ are regular and there exists a unique solution $\boldsymbol{y} \in [\boldsymbol{y}]^{\boldsymbol{k}}$, where $\overset{\circ}{\subset}$ means contained in the interior.

Having found an approximate solution $\tilde{\boldsymbol{x}}$ for the original linear system, the enclosure of the residual can be used to give a verified enclosure of the solution by $\hat{\boldsymbol{x}} \in \tilde{\boldsymbol{x}} + [\boldsymbol{y}]^{\boldsymbol{k+1}}$

4 Implementation

All routines are implemented in C++/CUDA using version 5.0 of the CUDA SDK and use double-precision to carry out floating-point computations.

For interval computations directed rounding must be available. In CUDA the rounding-mode can be specified on an instruction level [8] using intrinsic functions [9]. Thus fine grained control over the rounding-mode is possible.

In order to speed up computation we use a tiled matrix multiplication as shown in Figure 1. The result matrix is split into rectangular tiles and each tile is computed by its own thread block. Due to hardware limitations, in our case, the number of threads in a thread block is smaller than the number of cells in a tile. It follows that every thread needs to compute multiple cells of the tile.

In [3] a scheme where every thread computes one or more rows of the tile is described. The shared memory is used to reduce global memory access while computing the rows. We adopted this approach for all our routines.

All routines have been tested with different tile and thread block sizes to determine the fastest combination. The tile and thread block sizes are varied between 256×16 and 64×8 and between 32×4 and 16×4 respectively.

We use the kernel template shown in Algorithm 1.1 for all our kernels. Only the computation of the scalar product is varied according to the specific case. Also kernels that don't use tiling were developed to demonstrate the achieved speed up.

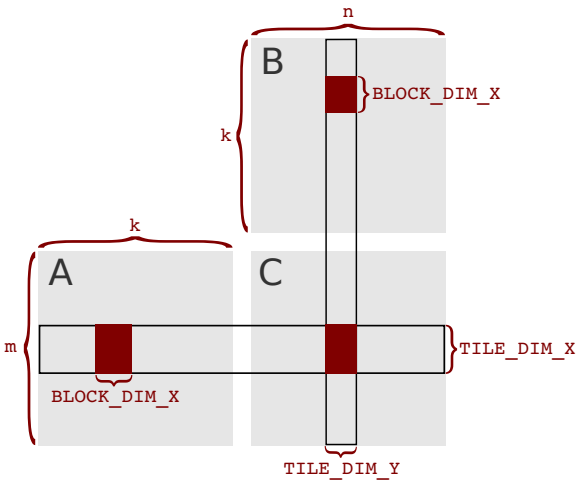

Fig. 1. Model for multiplication of two matrics $C^{m \times n} = A^{m \times k} \cdot B^{k \times n}$.

Whenever appropriate the intrinsic FMA function [9] is used to speed up computation and avoid additional round-off errors. All kernels are implemented as C++ function templates and the decision for a concrete rounding-mode is made at compile-time to reduce runtime overhead.

4.1 Interval Matrix-matrix Product

An interval matrix-matrix product was implemented using the tiling scheme described before. Since an interval consists of two floating-point numbers for the lower and upper bound, more shared memory and registers are used by the kernel compared to a kernel that executes floating-point matrix-matrix product. This results in smaller possible thread block sizes.

Algorithm 1.1 Basic kernel template for matrix-matrix multiplication that implements configuration in Figure 1

Input: m, n, k, A, B, C

a_tile, b_tile, $c\_tile \leftarrow$ positions of tiles.
$results[TILE\_DIM\_Y] \leftarrow 0$ // initialize dot-product results with 0.
$shared\_cache[BLOCK\_DIM\_X][TILE\_DIM\_Y] \leftarrow 0$ // shared memory cache
$steps \leftarrow k/BLOCK\_DIM\_X$
$step \leftarrow 0$
while $step < steps$ **do**
  $shared\_cache \leftarrow$ load part of current sub-tile from b_tile
  $results\_pos \leftarrow 0$
  while $results\_pos < TILE\_DIM\_Y$ **do**
    // Compute next term of scalar product for every row element
    $results\_pos \leftarrow results\_pos + 1$
  end while
  $step \leftarrow step + 1$
end while
$c\_tile \leftarrow results$ // store row of result back

4.2 Real Matrix-matrix Product with Directed Rounding

As of version 5.0, CUBLAS routines do not support directed rounding. We implemented a matrix-matrix multiplication routine that makes use of intrinsic functions to support all in IEEE-754 [10] specified rounding modes. As mentioned before the decision for a specific rounding-mode is made at compile-time to ensure that no overhead occurrs. The routines were implemented using the same kernel template shown in Algorithm 1.1 to achieve a good performance.

4.3 Matrix-vector Product as in Two-fold Working Precision

A matrix-vector product in higher precision is needed by the algorithm implemented to demonstrate our routines. This is realized using error-free transformations to compute the dot product as in twice the working precision [11]. In [1] an implementation of a matrix-matrix product using error-free transformations is shown. We adapted this approach for our matrix-vector multiplication that evaluates the dot-product in twice the working precision.

5 Verification Algorithm

In Section 3.2, we presented the basics for implementing an iterative a posteriori method for calculating an enclosure of a solution $\hat{\boldsymbol{x}}$ of a linear system $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$. The start interval $[\boldsymbol{x}]^{\mathbf{0}}$ of an iterative a posteriori interval method does not neccessarily contain the correct solution $\hat{\boldsymbol{x}}$, but aims to find an enclosure after some iteration steps. In this section we describe some details of our implementation.

Algorithm 1.2 gives an overview of all the neccessary steps.

Algorithm 1.2 LinSolve($\boldsymbol{A}, \boldsymbol{b}, [\boldsymbol{x}]$) [2]

1. Calculation of an approximate solution
2. Real residual iteration to improve approximate solution
3. Computation of enclosures $[\boldsymbol{C}]$ and $[\boldsymbol{z}]$ for $\boldsymbol{C} = \boldsymbol{I} - \boldsymbol{RA}$ and $\boldsymbol{z} = \boldsymbol{R}(\boldsymbol{b} - \boldsymbol{A}\tilde{\boldsymbol{x}})$
4. Finding a verified enclosure of solution $\hat{\boldsymbol{x}}$

In step one, an approximate solution of $\tilde{\boldsymbol{x}}$ is calculated, using an approximation of the inverse matrix $\boldsymbol{R}$ of $\boldsymbol{A}$. We use existing routines from the MAGMA library for calculating the inverse. Therefore matrix $\boldsymbol{A}$ is LU-factorized by using Magama's *getrf* routine. With Magma's *getri* routine the inverse is subsequently determined and $\tilde{\boldsymbol{x}}^{(\boldsymbol{0})}$ is calculated as $\tilde{\boldsymbol{x}}^{(\boldsymbol{0})} = \boldsymbol{R} \cdot \boldsymbol{b}$.

After that the approximate solution $\tilde{\boldsymbol{x}}^{(\boldsymbol{0})}$ is refined using real residual iteration. In every iteration step the scalar products are evaluated in two-fold working precision to reduce rounding errors. The iteration is stopped after a fixed number of iterations or if the desired accuracy is reached.

The symbol $\boxminus$ is used to indicate that scalar products are evaluated in two-fold working precision.

In step three, the verification step is prepared by calculating enclosures of $[\boldsymbol{C}]$ and $[\boldsymbol{z}]$. This step can be seen in Algorithm 1.3. Symbol $\lozenge$ means, that an interval enclosure of the real result is calculated using directed rounding.

Algorithm 1.3 Computation of enclosures $[\boldsymbol{C}]$ and $[\boldsymbol{z}]$.

Input: $\boldsymbol{A}$, $\boldsymbol{R}$, $\tilde{\boldsymbol{x}}$
$[\boldsymbol{C}] \leftarrow \lozenge(\boldsymbol{I} - \boldsymbol{R} \cdot \boldsymbol{A})$;
$\boldsymbol{d} \leftarrow \boxminus(\boldsymbol{b} - \boldsymbol{A} \cdot \tilde{\boldsymbol{x}})$;
$[\boldsymbol{d\_err}] \leftarrow \lozenge(\boldsymbol{b} - \boldsymbol{A} \cdot \tilde{\boldsymbol{x}} - \boldsymbol{d})$;
$[\boldsymbol{z}] \leftarrow \lozenge(\boldsymbol{R} \cdot \boldsymbol{d} + \boldsymbol{R} \cdot [\boldsymbol{d\_err}])$;
return $[\boldsymbol{C}]$, $[\boldsymbol{z}]$

During the verification step shown in Algorithm 1.4, the algorithm tries to calculate an enclosure for the real residual $\hat{\boldsymbol{y}}$ using an interval residual iteration. Since we are using an a posteriori method, the starting interval may not contain the searched fixed-point. The iteratees converge towards the fixed-point, but may not contain it. Using ϵ-inflation this problem can be reduced.

After an enclosure for the residual has been computed an interval containing the exact solution $\hat{\boldsymbol{x}}$ can be obtained $[\hat{\boldsymbol{x}}] = \tilde{\boldsymbol{x}} + [\hat{\boldsymbol{y}}]$.

6 Performance Measurements

Our performance tests were executed on a NVIDIA Tesla C2070 GPU with CUDA compute capability 2.0 and Fermi architecture. The host was running a Gentoo Linux 64 Bit system with an Intel Xeon E5504 quad-core CPU with 2

Algorithm 1.4 VerificationStep [2]

Input: $[\hat{\boldsymbol{y}}], [\boldsymbol{z}], [\boldsymbol{C}]$
$\epsilon \leftarrow 1000; p_{max} \leftarrow 10; p \leftarrow 0; [\hat{\boldsymbol{y}}]^{(\boldsymbol{0})} \leftarrow [\boldsymbol{z}];$
repeat
  $[\hat{\boldsymbol{y}}]^{(\boldsymbol{p})} \leftarrow [\hat{\boldsymbol{y}}]^{(\boldsymbol{p})} \cdot \epsilon$; {$\epsilon$-Inflation}
  $[\hat{\boldsymbol{y}}]^{(\boldsymbol{p+1})} \leftarrow \left([\boldsymbol{z}] + [\boldsymbol{C}] \cdot [\hat{\boldsymbol{y}}]^{(\boldsymbol{p})}\right);$
  $IsVerified \leftarrow \left([\hat{\boldsymbol{y}}]^{(\boldsymbol{p+1})} \overset{\circ}{\subset} [\hat{\boldsymbol{y}}]^{(\boldsymbol{p})}\right);$
  $p \leftarrow p + 1;$
until $IsVerified$ **or** $(p \geq p_{max})$
$[\hat{\boldsymbol{y}}] \leftarrow [\hat{\boldsymbol{y}}]^{(\boldsymbol{p})};$
return $[\hat{\boldsymbol{y}}]$, $IsVerified$;

GHz and 8 GB RAM. NVidia Driver version 304.64 and CUDA SDK 5.0 were installed. For comparison with CXSC we used version 2.5.3.

6.1 BLAS Routines

In Figure 2 the performance of our fastest matrix-matrix multiplication kernel is compared to the current CUBLAS *dgemm* operation. As can be seen we reach a peak performance around 207 GFlops while CUBLAS peaks around 310 GFlops. Our kernel reaches roughly 66% of the CUBLAS kernel performance so there is still room left for improvements. We assume that it should be possible to produce still faster versions of our interval routines.

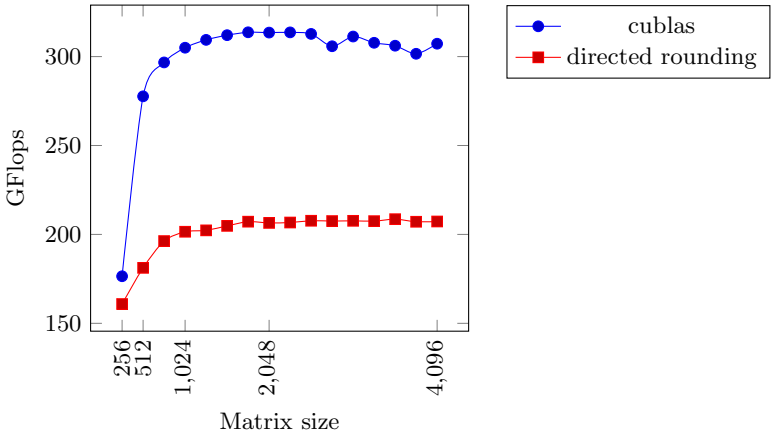

Fig. 2. Performance of our matrix-matrix multiplication implementation that supports directed rounding compared to current CUBLAS.

When computing an interval matrix-matrix product every computation has to be carried out for the upper and lower bound. It follows that such a kernel needs more registers and shared memory as a corresponding floating-point kernel. When reaching the maximum usable registers a CUDA kernel spills over into global memory. Additional store and load instructions will be generated that slow down the computation. In Figure 3 we compare kernels that use different tile and thread-block sizes. Details about the kernels can be found in Table 1.

No.	Block	Tile	Regs	SM	W/SMP	GFlops	Speedup
1	32x4	128x8	57	4096	16	205	3.94
2	32x4	128x16	63 [1]	8192	16	153	2.94
3	16x4	64x8	61	2048	16	209	4.02
4	8x8	8x8	36	0	16	52	1

Table 1. A table showing resource allocation for kernels with different dimensions. Performance and Speedup against the routine without tiling are given for a problem size of 4096×4096. SM = Shared Memory; Regs = Registers; W/SMP = Warps/Shared Multiprocessor

Although all kernels reach the same occupancy of 16 warps per multiprocessor kernel 2 is a lot slower because the maximum register limit is reached and additional store and load instructions to global memory are neccessary to correctly run the kernel. Kernel 1 and 3 are almost equally fast but since in kernel 3 the thread-block consists only of 64 threads it needs less shared memory, 8 blocks instead of 4 can be scheduled which seems to result in slightly better overall latency-hiding.

6.2 Verification Algorithm

For measurement of our verification algorithm implementation, we used routines from LAPACK to create random integer test-matrices. With these test-matrices we measured performance for our optimized CUDA implementation averaged over 10 test runs. For each run the measured time contains data transfer of matrices to the GPU, run time of the solving algorithm as well as copying back results from the GPU. Figure 4 and 5 shows time measurements of our implementation and the reference CXSC implementation for matrix sizes of 256 up to 8192. The maximum matrix size was limited by available memory on the GPU.

Besides the fact that our implementation uses the GPU, the main difference between our CUDA implementation and the compared CXSC implementation is the quality of scalar-product calculation. CXSC uses exact evaluation of dot products which results in tight enclosures of the exact floating-point dot product result. The drawback of this approach is that exact evaluation is computation

[1] Register limit is reached. Additional access to global memory is neccessary.

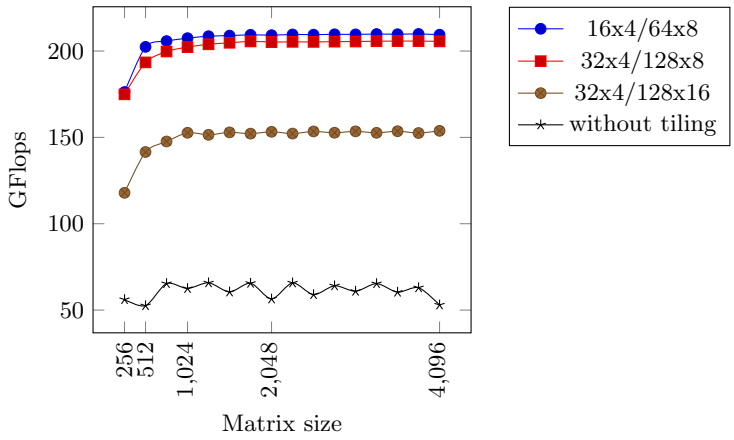

Fig. 3. Performance of interval matrix-matrix multiplication routines with different block and tile dimensions.

intensive. In order to reduce accumulation of errors we use two-fold working precision for dot product calculations where appropriate, still rounding-errors accumulate and therefore, our enclosure is getting wider as matrix dimensions increase. For a random testcase this effect can be seen in Table 2. Since we use random integer test matrices CXSC finds the exact solutions, represented as point-interval vectors while the width of our results increases with the problem size. Overall, our implementation is less accurate, but as speedup also shows, much faster than CXSC.

Size	Width(CUDA)	Width(CXSC)	Speedup
256	$7.92 \cdot 10^{-11}$	0	257
512	$2.9 \cdot 10^{-10}$	0	1,023
1,024	$7.13 \cdot 10^{-9}$	0	3,522
2,048	$1.39 \cdot 10^{-8}$	0	6,900
4,096	$4.53 \cdot 10^{-8}$	0	9,158
8,192	$1.89 \cdot 10^{-7}$	0	11,129

Table 2. Interval vector width for optimized CUDA implementation and CXSC implementation for one test case

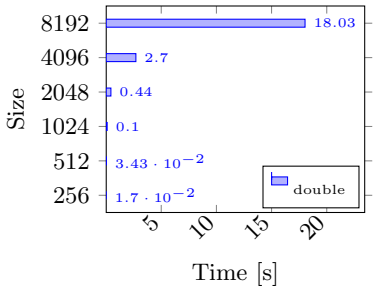

Fig. 4. Performance measurement results for the linear system solver using the optimized CUDA implementation

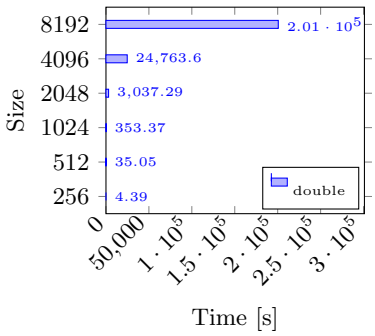

Fig. 5. Performance measurement results for CXSC - C++ Verified Toolbox implementation

7 Conclusion

In this paper we developed interval matrix routines on CUDA and successfully applied them to an existing method for finding enclosures of solutions for linear systems. Applying common optimization techniques from floating-point matrix multiplication improved the performance of those routines. Implementing a known algorithm for finding the solution of a linear system showed that promising speedups can be achieved using the GPU. Since our routines suffer from loosing accuracy compared to exact but more computation intensive evaluation, investigations into exact evalution of scalar products on the GPU are planned in the future.

References

1. Fujimoto, N.: Economical Two-fold Working Precision Matrix Multiplication on Consumer-Level CUDA GPUs. In: Architecture and Multi-Core Applications (WAMCA), 2011 Second Workshop on. (2011) 24–29
2. Hammer, R.: C++ Toolbox for Verified Computing. Springer (1995)
3. Cui, X., Chen, Y., Mei, H.: Improving Performance of Matrix Multiplication and FFT on GPU. In: Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on. (2009) 42–48
4. Beck, P.D., Nehmeier, M.: Parallel Interval Newton Method on CUDA. In: Applied Parallel and Scientific Computing. Volume 7782 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2013) 454–464
5. Jaulin, L., Kieffer, M., Didrit, O., Walter, E.: Applied Interval Analysis. Springer (2001)
6. Alefeld, G., Herzberger, J.: Introduction to interval computations. Computer science and applied mathematics. Academic Press (1983)

7. S. M. Rump: Kleine Fehlerschranken bei Matrixproblemen. (Universität Karlsruhe 1980)
8. NVIDIA Corporation: Parallel Thread Execution ISA (Version 3.1) `http://docs.nvidia.com/cuda/pdf/ptx\_isa\_3.1.pdf`.
9. NVIDIA Corporation: NVIDIA CUDA C Programming Guide (Version 5.0) `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.
10. IEEE 754-2008: IEEE Standard for Floating-Point Arithmetic (2008)
11. Ogita, T., Rump, S., Oishi, S.: Accurate sum and dot product. SIAM Journal on Scientific Computing **26**(6) (2005) 1955–1988