A GPU Implementation of a Jacobi Method for Lattice Basis Reduction

Filip Jeremic and Sanzheng Qiao

Department of Computing and Software
McMaster University
Hamilton, Ontario, Canada

June 25, 2013

Abstract

This paper describes a parallel Jacobi method for lattice basis reduction and a GPU implementation using CUDA. Our experiments have shown that the parallel implementation is more than fifty times as fast as the serial counterpart, which is about twice as fast as the well-known LLL lattice reduction algorithm.

Keywords Lattice basis reduction, parallel computing, GPU, CUDA

1 Introduction

Lattice basis reduction has been successfully used for many problems in integer programming, cryptography, number theory, and information theory [1]. In this paper we discuss a parallel version of the lattice basis reduction algorithm called the Jacobi Method. The Jacobi Method is very attractive as it is inherently parallel. We take advantage of this by utilizing the graphics processing unit (GPU) to capitalize on the algorithm's parallel nature. This paper will first introduce a serial version of the Jacobi Method, and later explore its parallel nature. Moreover, we will discuss the tools and techniques used to achieve high runtime performance and finally we will present experimental results of our parallel implementation of the Jacobi Method.

In this section we cover some basic notations used throughout the paper. Given a subspace W of $\mathbb{R}^n$ and a basis $\mathcal{B} = \{b_1, b_2, \ldots, b_m\}$ of n-dimensional vectors which span W, we define a *lattice* $\mathcal{L}$ of W generated by the basis $\mathcal{B}$ as the set of vectors:

$$\mathcal{L}(\mathcal{B}) = \left\{ \sum_{i=1}^{m} a_i b_i \;\middle|\; a_i \in \mathbb{Z} \right\}$$

Typically, we view a lattice basis $\mathcal{B}$ in matrix form, where the vectors in the basis form the columns of the matrix. In this context we say that the respective matrix $\mathcal{B}$ is a *generator* of the lattice $\mathcal{L}$. The value m in the above definition of a lattice is called the lattice *dimension*, or *rank*. A given lattice basis may have fewer vectors than the space it resides in. In such a case the generator matrix is rectangular with $m < n$. If on the other hand $m = n$, we say that the lattice is of *full rank*, and consequently the generator matrix will be an invertible square matrix.

When the lattice dimension $m \geq 2$, the lattice can have infinitely many distinct basis matrices. This is not surprising as the underlying vector space can also have infinitely many bases. The question arises as to how can we transform one basis matrix into another, and more importantly what makes one basis "better" than another? To answer the former question we introduce the notion of a lattice *determinant*, which is defined as the absolute value of the determinant of the respective generator matrix. The lattice determinant is an important numerical invariant as it is independent of the chosen lattice basis. Therefore, two generator matrices $\mathcal{B}$ and $\mathcal{B}'$ generate the same lattice $\mathcal{L}$ if and only if $\mathcal{B}' = \mathcal{B}Z$, where Z, called a unimodular matrix, is an integer matrix with $|\det Z| = 1$. Because the determinant of a unimodular matrix is of unit size, the inverse of a unimodular matrix is also an integer matrix.

The answer to the latter question we posed is relative to the application problem at hand, however for many such problems a desirable property of a lattice basis is that it consists of relatively short and more orthogonal vectors. In this context, we say that such a basis is *reduced*. Thus given a lattice basis matrix $\mathcal{B}$, a lattice basis reduction algorithm produces a unimodular matrix Z, such that the basis $\mathcal{B}Z$ is reduced.

2 Jacobi Method

In this section, we present a serial version of the Jacobi method for lattice basis reduction, but before doing so we describe the Lagrange's algorithm for computing reduced bases for lattices of dimension two [2, 3]. A lattice $\mathcal{L}(\mathcal{B})$ generated by the matrix $\mathcal{B} = \begin{bmatrix} b_1 & b_2 \end{bmatrix}$ is said to be *Lagrange-reduced* if

$$\|b_1\|_2 \leq \|b_2\|_2 \quad \text{and} \quad \left|b_1^{\mathrm{T}} b_2\right| \leq \frac{\|b_1\|_2^2}{2} \tag{1}$$

Intuitively, if θ denotes the angle between the two basis vectors b_1 and b_2, then condition (1) implies that $\frac{\pi}{2} \leq \theta \leq \frac{2\pi}{3}$ since

$$|\cos\theta| = \frac{\left|b_1^{\mathrm{T}} b_2\right|}{\|b_1\|_2 \|b_2\|_2} \leq \frac{\left|b_1^{\mathrm{T}} b_2\right|}{\|b_1\|_2^2} \leq \frac{1}{2}$$

The existence of a Lagrange-reduced basis for any two-dimensional lattice is guaranteed and is optimal in the sense that it consists of the shortest possible basis vectors [4]. The algorithm itself can be viewed as a generalization of Euclid's algorithm for computing the greatest common divisor of a pair of integers.

Algorithm 1 (Lagrange): Given $G = \mathcal{B}^{\mathrm{T}}\mathcal{B}$, where $\mathcal{B}$ is a two-dimensional lattice generator matrix, this algorithm computes a 2×2 unimodular matrix Z such that the generator matrix $\mathcal{B}Z$ is Lagrange-reduced and G is updated accordingly.

```
1   Z = I_2
2
3   if G(1,1) < G(2,2)
4       swap G(:,1) and G(:,2)
5       swap G(1,:) and G(2,:)
6       swap Z(:,1) and Z(:,2)
7   end
8
9   while G(1,1) > G(2,2)
10      q = ⌊G(1,2)/G(2,2)⌉
11      G(:,2) = G(:,2) − q × G(:,1)
12      G(2,:) = G(2,:) − q × G(1,:)
13      Z(:,2) = Z(:,2) − q × Z(:,1)
14  end
15
```

```
16 || return Z
```

Analogous to Euclid's algorithm the computed matrix Z can be viewed as the product of permutations and a Gauss transformations [5]:

$$\begin{bmatrix} 0 & 1 \\ 1 & -q \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & -q \\ 0 & 1 \end{bmatrix}$$

Furthermore, the Gram matrix $G = [g_{ij}]$ is a symmetric and positive definite matrix as an input to Algorithm 1 containing noteworthy information. Namely, the diagonal elements $g_{ii} = \|b_i\|_2^2$ and the off-diagonal elements $g_{ij} = b_i^{\mathrm{T}} b_j$, both of which appear in the condition (1).

For an n-dimensional lattice generated by a matrix $\mathcal{B}$ we can apply Algorithm 1 to every two-dimensional sublattice. The resulting algorithm, called the Jacobi method, Lagrange-reduces all possible pairs of the columns of $\mathcal{B}$ in a row-by-row fashion. We present a serial cyclic-by-row version of the Jacobi method for lattice basis reduction.

Algorithm 2 (Jacobi): Given an n-dimensional lattice generator matrix $\mathcal{B}$, this algorithm computes a unimodular matrix Z such that the columns of the generator matrix $\mathcal{B}Z$ form a reduced basis.

```
 1 || G = B^T B
 2 || Z = I_n
 3 ||
 4 || while not all pairs (b_i, b_j) satisfy (1)
 5 ||     for i = 1 to n-1
 6 ||         for j = i+1 to n
 7 ||             q = G(i,j)/G(j,j)
 8 ||             if |q| > 1/2
 9 ||                 G(:,j) = G(:,j) - ⌊q⌉ × G(:,i)
10 ||                 G(j,:) = G(j,:) - ⌊q⌉ × G(i,:)
11 ||                 Z(:,j) = Z(:,j) - ⌊q⌉ × Z(:,i)
12 ||             end
13 ||             if G(i,i) > G(j,j)
14 ||                 swap G(:,i) and G(:,j)
15 ||                 swap G(i,:) and G(j,:)
16 ||                 swap Z(:,i) and Z(:,j)
17 ||             end
18 ||         end
19 ||     end
20 || end
```

```
21 ||
22 ||return Z
```

Algorithm 2 implicitly applies the Lagrange's algorithm to every two-dimensional sublattice. Lines 7 to 17 carry out the reduction operations presented in Algorithm 1. Some optimizations have also been made. For example the while loop in Lagrange's algorithm was removed and replaced by the while loop on line 4 of Algorithm 2.

3 A Parallel Algorithm

A closer inspection of the Jacobi method presented in Algorithm 2 reveals further optimizations. Most notably that the algorithm can be parallelized by carrying out the Lagrange's algorithm on two-dimensional sublattices simultaneously. However, as with most parallel algorithms, we must be careful to avoid data hazards.

The two for-loops on lines 5 and 6 generate all column pair combinations (i, j) up to ordering, which makes sense as reducing columns i and j is equivalent to reducing columns j and i. The parallel version of Algorithm 2 must emulate such an ordering to ensure condition (1) is met by all column pairs of the input lattice generator matrix. Evidently, we must figure out the maximum number of parallel reductions we can carry out. Clearly we cannot reduce all column pairs simultaneously. To see why, consider two threads simultaneously reducing columns pairs (i, j) and (j, k) with $i < j < k$. On line 9 of the algorithm, the first thread reduces column j by an integer multiple of column i. Similarly, the second thread reduces column k by an integer multiple of column j. This poses a data hazard as there is a race condition on the value of $G(i, j)$ since the thread reducing (i, j) could update $G(i, j)$ before the thread reducing (j, k) uses it to update $G(j, k)$.

Alternatively we can follow the ordering presented in Algorithm 2 and reduce column pairs $(i, i{+}1), (i, i{+}2), \ldots, (i, n)$ in parallel, followed by the reduction of column pairs $(i{+}1, i{+}2), (i{+}1, i{+}3), \ldots, (i{+}1, n)$, and so forth. However, this ordering also presents data hazards from the swaps on lines 14-16. Even if we can ensure that the swaps never happen (i.e. the if statement on line 13 is never true), this ordering is suboptimal in the sense that at each iteration we decrease the number of threads performing reductions. The extreme

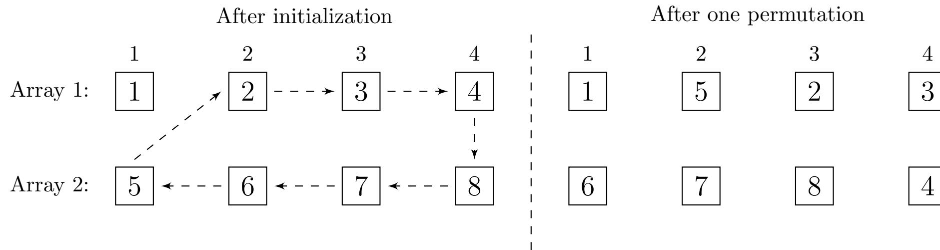

Figure 1: Chess tournament ordering with $n = 8$

here is that on the last iteration only one thread is performing a reduction, namely on the column pair $(n-1, n)$, while other threads are idle.

The solution is to use an ordering which maximizes concurrency while avoiding data hazards and race conditions. One such ordering is called the *chess tournament ordering* and is described in [6]. For a given n-dimensional input generator matrix, the chess tournament ordering is a mechanism of generating all $n(n-1)/2$ combinations of column pairs over $n-1$ iterations generating $n/2$ distinct column pairs.

The chess tournament ordering is best described through an example. Without loss of generality we assume that n, the number of columns of the input generator matrix, is even. For the sake of example we further assume that $n = 8$. The mechanism is implemented by two arrays of size $n/2 = 4$ in our case. Figure 1 depicts the initialization as well as one permutation of the chess tournament ordering, where the dashed arrows represent the transition between the two states. The column pairs are selected based on array indices which are labeled above each box. In our example, after initialization the mechanism generates column pairs $(1, 5), (2, 6), (3, 7)$ and $(4, 8)$. After one permutation, it produces pairs $(1, 6), (5, 7), (2, 8) and (3, 4)$. Clearly, it takes $n - 1 = 7$ permutations to generate all column pairs. This ordering mechanism is employed in the following parallel version of Algorithm 2:

Algorithm 3 (Parallel Jacobi): Given an n-dimensional lattice generator matrix $\mathcal{B}$, this algorithm computes a unimodular matrix Z such that the columns of the generator matrix $\mathcal{B}Z$ form a reduced basis.

Main Thread:

```
1 || G = B^T B
2 || Z = I_n
3 ||
```

```
 4   for i = 1 to n/2
 5       arr1(i) = i
 6       arr2(i) = i + n/2
 7   end
 8
 9   launch n/2 child threads
10
11   return Z
```

Child Thread:

```
 1   while not all pairs (b_i, b_j) satisfy (1)
 2       i = arr1(tid)
 3       j = arr2(tid)
 4
 5       if i > j
 6           swap i and j
 7       end
 8
 9       q = G(i,j)/G(j,j)
10
11       if |q| > 1/2
12           G(:,j) = G(:,j) − ⌊q⌉ × G(:,i)
13           Z(:,j) = Z(:,j) − ⌊q⌉ × Z(:,i)
14       end
15
16       thread barrier
17
18       if |q| > 1/2
19           G(j,:) = G(j,:) − ⌊q⌉ × G(i,:)
20       end
21
22       thread barrier
23
24       if G(i,i) > G(j,j)
25           swap G(:,i) and G(:,j)
26           swap Z(:,i) and Z(:,j)
27       end
28
29       thread barrier
30
31       if G(i,i) > G(j,j)
32           swap G(i,:) and G(j,:)
33       end
```

```
34
35      permute arr1 and arr2
36
37      thread barrier
38  end
```

Algorithm 3 consists of two parts; the main thread which initializes the data and the child threads which carry out the reduction using the said data. The first thing to notice about the child threads is the use of the special keyword *tid* which stands for the thread identification number. We assume that the our environment generates a unique incremental *tid* (starting at 1) for every child thread. The *tid* is used to extract column pair that a specific thread will reduce.

The next thing to note is the use of thread barriers. A thread barrier (or thread fence) forces the current thread to wait until all other threads have also reached the barrier. It is a synchronization technique used to avoid race conditions. As an example, the thread barrier on line 16 is used to avoid the race condition between the row and column updates of the matrix G. The assignments on lines 12 and 19 interfere with each other as they both overwrite the value of $G(j, j)$. The thread barrier must be placed outside of the branching if-statement to avoid the case in which one thread branches away while another does not. In this case the former thread will encounter a thread barrier, but the latter thread will never reach the barrier as it branched away, hence the program enters a deadlock.

4 GPU Implementation

To achieve high performance, Algorithm 3 requires that multiple threads are reducing a given basis simultaneously. The current models of multi-core CPU's do not offer such functionality as they are typically limited to four to eight threads running concurrently. Thus, we chose to implement Algorithm 3 on the GPU using the CUDA parallel computing platform [7].

Unlike the CPU (referred to as the *host*), the GPU (referred to as the *device*) is not optimized to achieve performance through fast serial program execution, but rather it exhibits high performance through massive parallelization. Therefore, only problems which are parallel in nature and can be recursively decomposed into similar subproblems will benefit from the

massive parallelization offered by the GPU. The maximum number of parallel threads executing on a device supporting the CUDA parallel computing platform exclusively depends on the underlying architecture's computing capability [7].

The host and device must work in unison to coordinate a task. This relationship starts out by transferring data from the system memory (RAM) to the device memory. The host then invokes *kernels*, which are the programs executing on the device in parallel, on the device to compute on the said data. The host then transfers the data back from the device and continues execution. CUDA programs are heterogeneous in the sense that both the host and the device can be executing programs at the same time, however synchronization between host programs and kernels is often necessary and is provided by the CUDA framework.

There are many different types of memory on the device implemented both in hardware and as abstractions. The two most important ones are global memory (analogous to RAM) and shared memory (analogous to L1 cache). Global memory is automatically cached and persists throughout the execution of a kernel and is useful for transferring data from the host to the device and vice versa. In comparison to shared memory, which is a fast user managed memory space local to a block of threads, global memory is quite slow. A common way of increasing performance is to transfer chunks of data from global memory to shared memory in a coalesced manner [8]. Memory coalescing occurs when consecutive threads access consecutive memory locations. Memory coalescing was used in our implementation whenever appropriate. This technique was used to reduce the memory access time of accessing array indices in the two permutation arrays on lines 2 and 3 in the Child Thread part of Algorithm 3. Specifically, the arrays are transferred from the global memory to shared memory at the beginning and transferred back to the global memory at the very end. In theory, we could obtain maximum memory bandwidth by transferring all data from global memory to shared memory and then performing the computations. However, shared memory is limited (48 KB in our case) hence this is not feasible.

Another optimization technique employed in our implementation was to eliminate the swaps on lines 25, 26 and 32 in the Child Thread part of Algorithm 3 by using a permutation array. Meaning that instead of swapping entire rows and columns (very memory intensive) we swap row and column indices in an array and reference this array whenever accessing data from the matrices. For example, a permutation array p is initialized to the identity

array, that is $p(i) = i$ for $i = 1, 2, \ldots, n$. For a given index i, the value of $p(i)$ represents the true location of the column (or row) i in a matrix (G and Z in our case). Whenever we must swap two columns (or rows) i and j, we instead swap the corresponding indices in the permutation array, that is we swap the values $p(i)$ and $p(j)$. Thus after initialization and one swap, the true position of column i can be obtained by referencing $p(i)$ which after the swap would equal to j. The tradeoff here is that since we are using the array p to simulate swapping of columns and rows we must therefore reference p whenever we need to access a particular value of a matrix. Because p is a one dimensional array, we can store it in shared memory hence making this tradeoff worth while.

Unfortunately, the reductions performed on lines 12, 13 and 19 in the Child Thread part of Algorithm 3 cannot take advantage of memory coalescing because of our column pair ordering we cannot ensure that consecutive threads are accessing consecutive memory locations. However, by adjusting various compiler optimization flags loop unrolling was found to be effective at speeding up the reduction of the columns and rows.

5 Experiments

In this section we present benchmarks of our implementation by comparing the performance of our GPU implementation with a serial CPU version. All experiments were performed on an Intel Core i5-2500K and an NVIDIA GTX 660 with CUDA 5.0. This GPU has 960 CUDA cores operating at 980 MHz and 2.0 GB global memory operating at 6008 MHz frequency. Both the CPU and GPU implementations use single precision floating-point arithmetic. The results of the reductions of the CPU and GPU implementations are verified term by term. The effectiveness of the Jacobi method is measured by the *Hadamard ratio* $\delta(\mathcal{B})$, also known as the orthogonality defect or linear independence number [1] and is defined as:

$$\delta^n(\mathcal{B}) = \frac{\prod_{i=1}^{n} \left\| \vec{b_i} \right\|_2}{\sqrt{\det \mathcal{B}^T \mathcal{B}}}$$

From Hadamard's inequality, $\delta(\mathcal{B}) \geq 1$, and the equality holds when the vectors are pairwise orthogonal. This numerical metric describes the relative deviation from a fully orthogonalized basis and can be used to rank different bases of a lattice based on the pairwise orthogonality of the vectors in the

Matrix Size	GPU (ms)	CPU (ms)	δ Original	δ Reduced
10	0.234		1.927	1.200
20	0.374		1.797	1.674
30	0.479		1.664	1.641
40	0.624		1.682	1.677
50	0.882	40		
100	2.481	147		
200	7.733	432		

Table 1: GPU vs. CPU benchmark statistics

basis. Our experiments show that the effectiveness of our GPU parallel implementation is consistent with that of the CPU serial implementation.

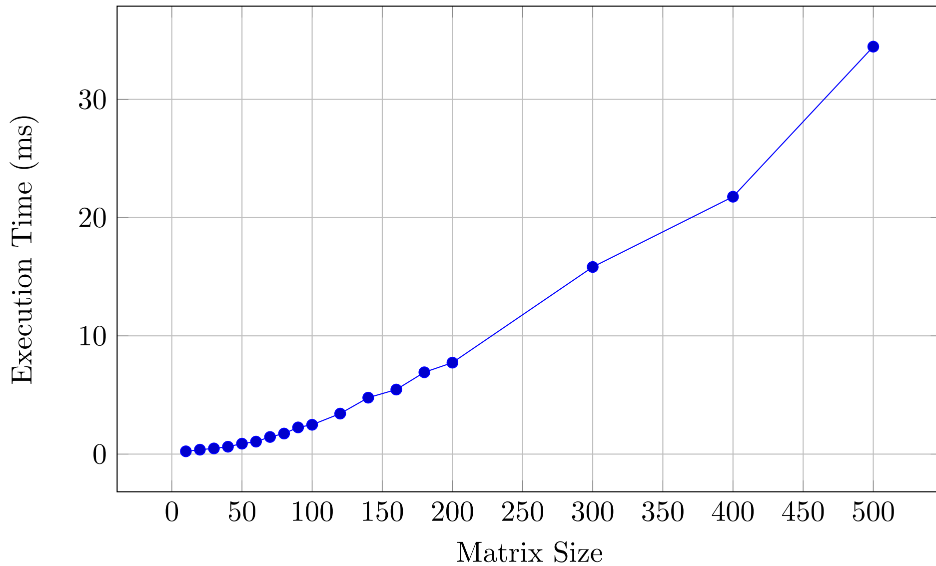

Figure 2: GPU benchmark of Jacobi method.

Table 1 gives benchmark statistics on the GPU implementation of the Jacobi method against the CPU implementation. The timing measurement excludes copying the data to and from the device. Dense matrices with normally distributed random entries were generated and 100 samples for each matrix size were averaged to produce the given statistics.

Figures 2 and 3 depict the results presented in Table 1 with a number of omitted data points. From Figure 3 we can see that the execution time of

the GPU implementation is nearly linear in comparison to the CPU. The GPU implementation achieves an impressive speedup factor of roughly 58 times on average. Further speedups are expected on the newer generations of GPUs. Figure 4 depicts the effectiveness of the reduction method according to the Hadamard ratio. The Hadamard ratio of the reduced basis is always smaller, although for matrices of size greater than 30 the Hadamard ratio of the original basis is not much higher then that of the reduced basis.

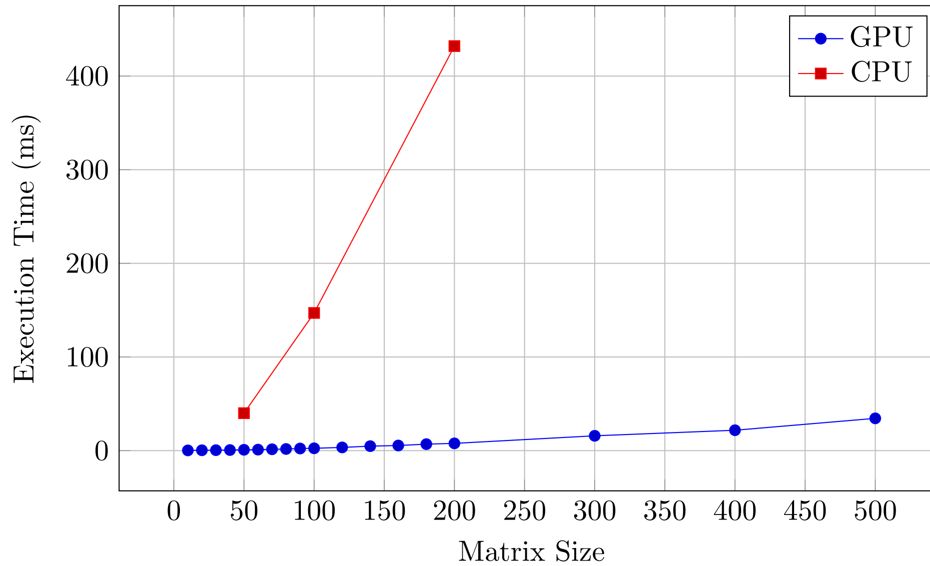

Figure 3: GPU and CPU benchmark of Jacobi method.

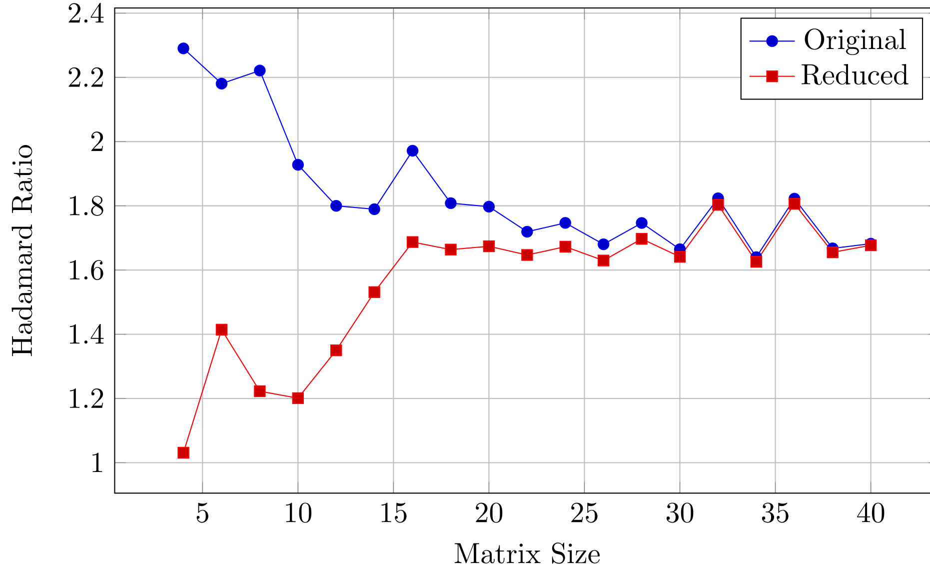

Figure 4: Hadamard ratio of original vs. reduced bases.

References

[1] S. Qiao. *A Jacobi Method for Lattice Basis Reduction*. Proceedings of 2012 Spring World Congress on Engineering and Technology (SCET2012). Vol. 2. IEEE. May 27-30, 2012, Xi'an, China. 649-652.

[2] J.L. Lagrange. *Recherches d'arithmeétique*. Nouveaux Mémoires de l'Académie de Berlin, 1773.

[3] F.T. Luk, S. Qiao, and W. Zhang. *A Lattice Basis Reduction Algorithm*. Technical Report 10-04. Institute for Computational Mathematics, Hong Kong Baptist University, Kowloon, Hong Kong, China, 2010.

[4] The LLL Algorithm: Survey and Applications. *Information Security and Cryptography, Texts and Monographs*. Editors Phong Q. Nguyen and Brigitte Vallée. Springer Heidelberg Dordrecht London New York, 2010.

[5] G.H. Golub and C. F. Van Loan. *Matrix Computations, Third Edition*. The Johns Hopkins University Press, Baltimore, MD, 1996.

[6] X. Wang and S. Qiao. *A Parallel Jacobi Method for the Takagi Factorization.* In Proceedings of the international Conference on Parallel and Distributed Processing Techniques and Applications - Volume 1, 2002.

[7] NVIDIA. *Parallel Programming and Computing Platform.* Online, available at `http://www.nvidia.com/object/cuda_home_new.html`. Accessed June 3, 2013.

[8] G.S. Sachdev et al. *Takagi Factorization on GPU using CUDA*. Technical Report. School of Computing, University of Utah, UT, 2010.