INTERNATIONAL JOURNAL OF
NUMERICAL ANALYSIS AND MODELING, *SERIES B*
Volume 5, Number 1-2, Pages 1–12

A PARALLEL JACOBI-TYPE LATTICE BASIS REDUCTION ALGORITHM

FILIP JEREMIC AND SANZHENG QIAO

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 twice as fast as the well-known LLL lattice reduction algorithm.

Key words. Lattice basis reduction, Jacobi method, GPU.

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. After introducing notations in Section 2, we will first describe a serial version of the Jacobi method for lattice reduction in Section 3, and later explore its parallel nature in Section 4. Moreover, in Section 5 we will discuss the tools and tricks used in our GPU implementation to achieve high runtime performance. Finally, in Section 6 we will present experimental results of our parallel implementation of the Jacobi method.

2. Preliminaries

In this section we cover some basic notations which we will use 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 generate proper subspace of 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.

Received by the editors January 1, 2014 and, in revised form, March 22, 2014.
2000 *Mathematics Subject Classification.* 65F30, 68W10.

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. For example,

$$\mathcal{B} = \begin{bmatrix} 2.0 & 2.7 \\ 0 & 0.7 \end{bmatrix} \quad \text{and} \quad \mathcal{B}' = \begin{bmatrix} -0.7 & 1.3 \\ -0.7 & -0.7 \end{bmatrix}$$

form two bases for the same lattice. 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 square root of the determinant of $B^{\mathrm{T}}B$, where B is the respective generator matrix, that is,

$$\det(\mathcal{L}(\mathcal{B})) = \sqrt{\det(\mathcal{B}^{\mathrm{T}}\mathcal{B})}.$$

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 length, the inverse of a unimodular matrix is also an integer matrix. In the above example, the two generator matrices $\mathcal{B}$ and $\mathcal{B}'$ are related by

$$\mathcal{B}' = \begin{bmatrix} -0.7 & 1.3 \\ -0.7 & -0.7 \end{bmatrix} = \begin{bmatrix} 2.0 & 2.7 \\ 0 & 0.7 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ -1 & -1 \end{bmatrix} = \mathcal{B}Z$$

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. In the above example, $\mathcal{B}'$ is reduced from $\mathcal{B}$. It consists of shorter and more orthogonal basis vectors than those of $\mathcal{B}$.

There are various notions of a reduced basis. In 1850, Hermite introduced the first notion of reduction for lattices of arbitrary dimensions, proposed an algorithm for computing such reduced bases, and proved its termination [2]. Hermite's algorithm is of theoretical significance, but its complexity is still unknown. Schnorr and Euchner [3] reconsidered this problem and developed a practical algorithm for constructing the Hermite reduced basis. In 1873, Korkine and Zolotareff [4] strengthened the definition of Hermite reduced basis. Their proposed notion of reduction is usually called the *HKZ reduced basis* [5], named after Hermite, Korkine and Zolotareff. In 1983, using induction, Kannan [6] presented the first algorithm for constructing the HKZ reduced bases. Helfrich [7], Kannan [8], and Banihashemi and Khandani [9] further refined Kannan's algorithm and improved the complexity analysis. Note that the methods based on Kannan's strategy are intended as theoretical tools, and the related papers usually focus on asymptotic complexity. Agrell et. al. [10] presented a practical algorithm and used it as a preprocessor for the integer least squares problems. In 1891, Minkowski [11] defined a new notion of reduction, which is stronger than the HKZ reduction. This definition is now known as the *Minkowski reduced basis*. Lenstra, Lenstra, and Lovász [12] developed the first polynomial-time lattice reduction algorithm, known as the LLL algorithm, named after the three authors. Their notion of reduced basis is actually a relaxation of the Hermite reduced basis [2]. The LLL algorithm has become the most important tool in public-key cryptanalysis [13] and integer least squares problems [10, 14]. Further

improvements of the LLL algorithm have been developed. While some [15, 16, 17] improve the quality of the output of the LLL algorithm, others [18, 19] improve the efficiency of the algorithm.

In 1846, Jacobi presented an eigenvalue decomposition algorithm [20]. Its workhorse is the computation of the eigenvalue decomposition of a two-by-two matrix [24]. By using the Lagrange's algorithm for two dimensional lattice reduction [21] as the workhorse, Qiao proposed a Jacobi method for lattice basis reduction [1], to be described in the following section.

3. Jacobi Method

In this section, we present the serial version of a 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 [21, 22]. 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 $\pi/2 \leq \theta \leq 2\pi/3$ 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 [23]. The algorithm itself can be viewed as a generalization of the 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 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 the Euclid's algorithm the computed matrix Z can be viewed as the product of a permutation and a Gauss transformation [24]:

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

Furthermore, the Gram matrix $G = [g_{ij}]$ is a symmetric matrix which is taken as input to Algorithm 1 carries noteworthy information. Namely that 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 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 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 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.

4. 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

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 n the sense that at each iteration we decrease the number of threads which are performing reductions. The extreme 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 [25]. 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.

Chess tournament ordering is best described through an example. Without loss of generality we assume that n, the number of columns of he 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 $(4, 8)$. 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
```

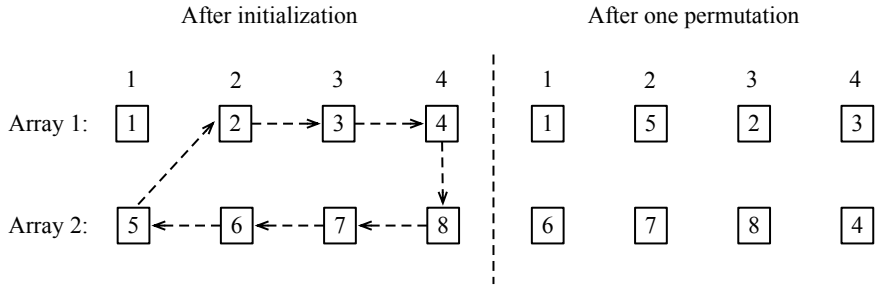

FIGURE 1. Chess tournament ordering with $n = 8$

```
    arr1(i) = i
    arr2(i) = i + n/2
end

launch n/2 child threads

return Z
```

.

Child Thread:

```
while not all pairs (b_i, b_j) satisfy (1)
    i = arr1(tid)
    j = arr2(tid)

    if i > j
        swap i and j
    end

    q = G(i,j)/G(j,j)

    if |q| > 1/2
        G(:,j) = G(:,j) − ⌊q⌋ × G(:,i)
        Z(:,j) = Z(:,j) − ⌊q⌋ × Z(:,i)
    end

    thread barrier

    if |q| > 1/2
        G(j,:) = G(j,:) − ⌊q⌋ × G(i,:)
    end

    thread barrier

    if G(i,i) > G(j,j)
        swap G(:,i) and G(:,j)
        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.

5. GPU Implementation

To achieve high performance, Algorithm 3 requires that many 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 4 or 8 threads running concurrently. This is why for our experiments we chose to implement Algorithm 3 on the GPU, specifically using the CUDA parallel computing platform [26].

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 the GPU 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 [26].

The host and device must work in unison to coordinate a task to perform. This relationship starts out by transferring data from the system memory (RAM) to the device memory. The host then invokes *kernels*, which are programs executing on the device in parallel, on the device to compute on the said data. The host

then transfers the memory 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 desired and is provided by the CUDA framework.

There are many different types of memory on the device both implemented 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 memory from the host to the device and vice versa. Memory coalescing [27] is one way of obtaining optimal memory bandwidth from global memory. Memory coalescing occurs when consecutive threads access consecutive memory locations. Memory coalescing was used in our implementation whenever appropriate.

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. 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 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 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. Because we only need a one dimensional array, we can store it in shared memory for optimal performance.

Unfortunately, the reductions performed on lines 12, 13 and 19 cannot take advantage of memory coalescing because of our column pair ordering we cannot ensure that consecutive threads are accessing consecutive memory locations. However loop unrolling was found to be effective at speeding up the reduction of the rows and columns.

6. 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:

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

From the Hadamard's inequality, $\delta(\mathcal{B}) \geq 1$, and the equality holds if and only if 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 basis.

TABLE 1. GPU vs. CPU benchmark statistics

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		

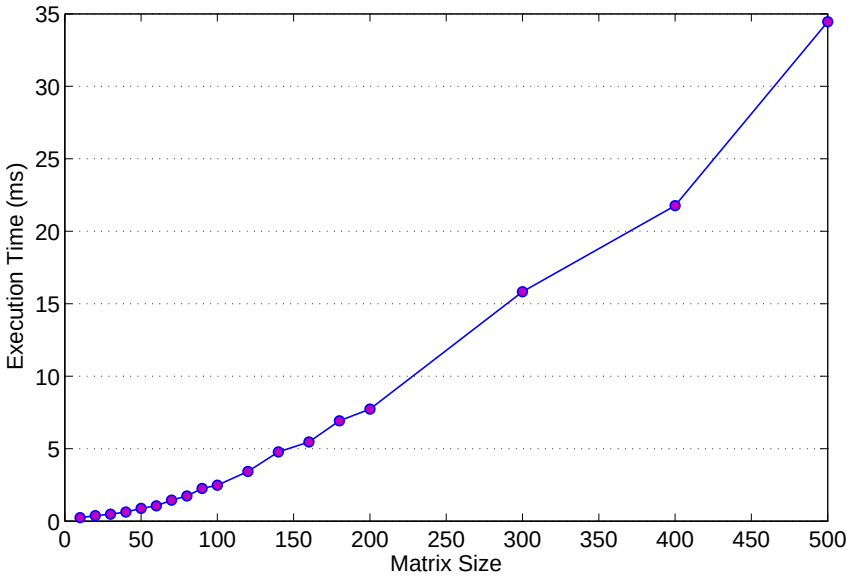

FIGURE 2. GPU benchmark of Jacobi method.

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. Normally distributed dense random matrices were generated and 100 samples for each matrix size are 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 a speedup factor of roughly 58 times on average which is quite impressive. 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 than that of the reduced basis due to the nth root in the definition of the ratio $\delta(\mathcal{B})$. Further speedups are expected on the newest generation of GPUs.

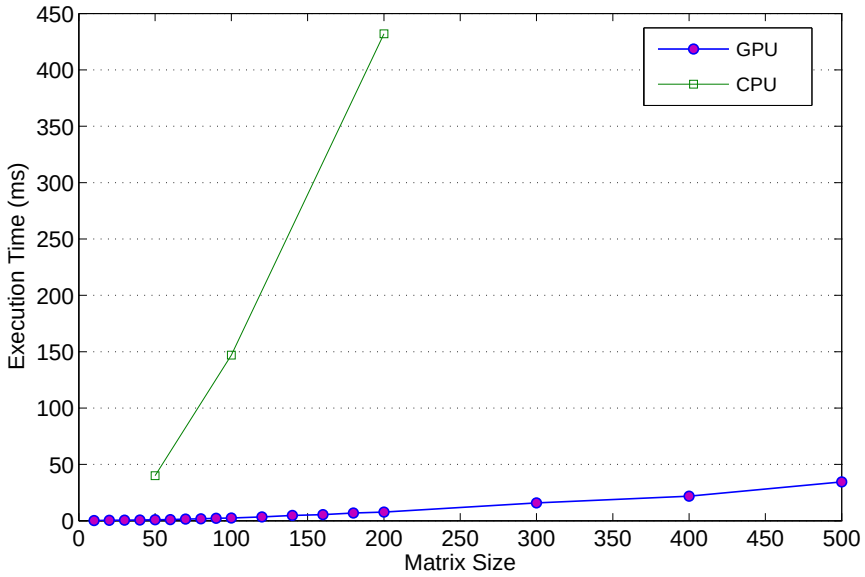

FIGURE 3. GPU and CPU benchmark of Jacobi method.

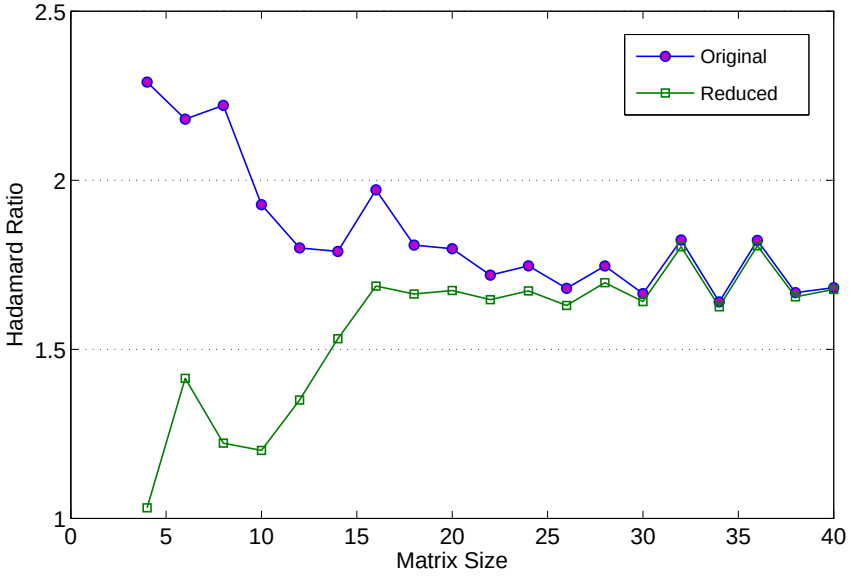

FIGURE 4. Hadamard ratio of original vs. reduced bases.

Acknowledgments

This research is partially supported by Natural Science and Engineering Research Council (NSERC) of Canada.

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] C. Hermite. Extraits de lettres de M. Hermite à M. Jacobi sur différents objets de la théorie des nombres. *J. Reine Angew. Math*, **40**, 1850, 279–290.

[3] C. P. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Math. Programming*, **66**, 1994, 181–199.

[4] A. Korkine and G. Zolotareff. Sur les formes quadratiques. *Maht. Ann.*, **6**, 1873, 366–389.

[5] 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.

[6] Ravi Kannan. Improved algorithms for integer programming and related lattice problems. *Proc. ACM Symp. Theory of Computing*, Boston, MA, Apr, 1983, 193–206.

[7] Bettina Helfrich. Algorithms to construct Minkowski reduced and Hermite reduced lattice bases. *Theoretical Computer Science*, vol. 41, no. 2-3, 1985, 125–139.

[8] Ravi Kannan. Minkowski's convex body theorem and integer programming. *Mathematics of Operations Research*, 3(**12**), 1987, 415–440.

[9] A. H. Banihashemi and A. K. Khandani. On the complexity of decoding lattices using the Korking-Zolotarev reduced basis. *IEEE Trans. Inform. Theory*, **44**, 1998, 162–171.

[10] Erik Agrell, Thomas Eriksson, Alexander Vardy, and Kenneth Zeger. Closest point search in lattices. *IEEE Transactions on Information Theory*, vol. 48, no. 8, 2002, 2201–2214.

[11] H. Minkowski. Über die positiven quadratischen Formen und über kettenbruchähnliche Algorithmen. *Journal fur die Reine und Angewandte Mathematik*, vol. 107, 1891, 278–297.

[12] A.K. Lenstra, H.W. Lenstra, Jr. and L. Lovász. Factorizing polynomials with rational coefficients. *Mathematicsche Annalen*, **261**, 1982, 515–534.

[13] D. Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices Amer. Math. Soc*, **46**, 1999, 203–213.

[14] X. W. Chang and G. H. Golub. Solving ellipsoid-constrained integer least squares problems. *SIAM J. Matrix Anal. Appl*, **31**, 3, 2009, 1071–1089.

[15] C. P. Schnorr. A hierarchy of polynomial lattice basis reduction algorithms. *Theo. Comput. Sci*, **53**, 1987, 201–224.

[16] F.T. Luk and D.M. Tracy. An improved LLL algorithm. *Linear Algebra and its Applications*, **428**(2–3), 2008, 441–452.

[17] F.T. Luk and S. Qiao. A Pivoted LLL Algorithm. *Linear Algebra Appl*, 2010.

[18] P. Q. Nguyen and D. Stehlé. An LLL algorithm with quadratic complexity. *SIAM J. Comput*, **39**, 3, 2009, 874–903.

[19] Y. H. Gan, C. Ling, and H. M. Mow. Complex lattice reduction algorithm for low-complexity full-diversity MIMO detection. *IEEE Transactions on Signal Processing*, vol. **57**, no. 7, 2009, 2701-2710.

[20] C.G.J. Jacobi. Uber ein Leichtes Verfahren Die in der Theorie der Sacularstroungen Verkommendern Gleichungen Numerisch Aufzulosen. *Crelle's J.*, vol. **30**, 1846, 51-94.

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

[22] 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.

[23] 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.

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

[25] 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.
[26] NVIDIA. *Parallel Programming and Computing Platform*. Online, available at http://www.nvidia.com/object/cuda_home_new.html. Accessed June 3, 2013.
[27] G. S. Sachdev et al. *Takagi Factorization on GPU using CUDA*. Technical Report. School of Computing, University of Utah, UT, 2010.

Department of Computing and Software, McMaster University, Hamilton, ON, CANADA L8S 4K1

E-mail: jeremif@mcmaster.ca and qiao@mcmaster.ca

URL: http://www.cas.mcmaster.ca/∼qiao/