Iterative Sparse Matrix-Vector Multiplication for Integer Factorization on GPUs

Bertil Schmidt[1,2], Hans Aribowo[1], and Hoang-Vu Dang[1]

[1] School of Computer Engineering, Nanyang Technological University, Singapore
{asbschmidt,hans.aribowo,hvdang}@ntu.edu.sg
[2] Institut für Informatik, Johannes Gutenberg University Mainz, Germany
bertil.schmidt@uni-mainz.de

Abstract. The Block Wiedemann (BW) and the Block Lanczos (BL) algorithms are frequently used to solve sparse linear systems over GF(2). Iterative sparse matrix-vector multiplication is the most time consuming operation of these approaches. The necessity to accelerate this step is motivated by the application of these algorithms to very large matrices used in the linear algebra step of the Number Field Sieve (NFS) for integer factorization. In this paper we derive an efficient CUDA implementation of this operation using a newly designed hybrid sparse matrix format. This leads to speedups between 4 and 8 on a single GPU for a number of tested NFS matrices compared to an optimized multi-core implementation.

Keywords: SpMV, CUDA, Block Wiedemann, RSA, Number Field Sieve, Factorization.

1 Introduction

The Number Field Sieve (NFS) is the current state-of-the-art integer factorization method. It requires the solution of a large sparse linear system over GF(2) (called the linear algebra step). Presently there are two efficient algorithms to solve such a large sparse linear system, namely Block Wiedemann (BW) [8] and Block Lanczos (BL) [15]. Both algorithms have a common time consuming operation: iterative sparse matrix vector multiplication (SpMV).

Recent integer factorization efforts have been using CPU clusters to solve the large sparse linear system [1,13]. The RSA-768 factorization [13], for example, reported a runtime of 3 months for the linear algebra step on a cluster with 48 AMD dual hex-core CPUs. Previous works on parallelizing the linear algebra step focused on using CPU clusters and grids [2,10,11,12]. In this paper, we investigate how a Fermi GPU [17] and the CUDA programming model [16] can be used to accelerate the costly iterative SpMV for matrices derived from NFS.

The memory access pattern in the SpMV operation generally consists of regular access patterns over the matrix and irregular access patterns over the vector. The irregular access pattern over the vector is a challenge that is pronounced more on the GPU than on the CPU, because of the smaller cache and the restrictive memory access pattern requirement to achieve maximum performance.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6853, Part II, pp. 413–424, 2011.

However, a high-end GPU has an order-of-magnitude higher bandwidth than a high-end CPU; e.g. a GeForce GTX 580 has 192.4 GB/s memory bandwidth, while an Intel Core-i7 has a maximum of 25.6 GB/s memory bandwidth.

SpMV on the GPU has been explored previously in several papers [3,6,7,14] for matrices derived from scientific computing applications. However, sparse matrices derived from NFS have generally different properties, i.e. they are larger, have a few dense rows and have many extremely sparse rows. The large size of the matrix causes the BL and BW algorithms to require a large number of SpMV iterations. This means that the time spent for matrix preprocessing and matrix data transfer to the GPU memory are negligible compared to the total runtime. Thus, approaches to the SpMV on GPUs for NFS matrices may be different from previously published GPU SpMV approaches.

This paper is organized as follows. Section 2 describes several published sparse matrix formats and their GPU performance when used with NFS matrices. Section 3 presents our new formats specifically designed for NFS matrices and their CUDA implementation for the Fermi GPU architecture. We compare our result to an Intel Nehalem CPU with the publicly available CADO-NFS [9] software in Section 4. Finally, Section 5 concludes the paper.

2 SpMV on GF(2) for NFS Matrices Using Existing Formats on GPUs

In this section, we review a few relevant previously published sparse matrix formats on GPUs and study their performance when applied to sparse matrices over GF(2) derived from integer factorization with NFS.

We consider a sparse binary matrix A of size $N \times N$ and a dense vector X of size $N \times n$ bit, where n is called the *blocking factor*. Typical blocking factors are of the form of $64 \cdot k,\ k \in \mathbb{N}$. Note that doubling the blocking factor roughly halves the number of SpMV iterations required but doubles the input vector size.

For all $0 \leq i \leq N-1$ let $c\_index[i]$ be a column index of $A[i]$ which contains the indices of the non-zero entries of row i. Then, the following pseudocode shows a single SpMV iteration of A with input vector X and result vector Y.

```
SPMV( Input: c_index,X; Output:Y )
   for (i=0 ; i<N ; i++)
     Y[i] = 0
     for (j=0; j<c_index[i].size(); j++)
       ind = c_index[i,j]
       Y[i] = Y[i] XOR X[ind]
end
```

The costly operations in the SpMV pseudocode are the memory accesses for loading $c\_index[i,j]$, $X[ind]$, $Y[i]$ and storing $Y[i]$. To speed up those operations on any architecture a common approach is to design a cache-friendly order of

accessing the memory. The order is especially important for the vectors X and Y, since their memory locations might be accessed multiple times.

CUDA implementations generally store both matrix and vectors in high latency global memory. Memory accesses to A and Y can usually be coalesced. Memory accesses to X are random and non-coalesced, but texture memory can be used to take advantage of texture cache. We now briefly review the CUDA implementation of a number of SpMV formats published in previous papers [3,14].

Coordinate list (COO). For each non-zero, both its column and row indices are explicitly stored. The Cusp implementation [4] stores elements in sorted order of row indices ensuring that entries with the same row index are stored contiguously. This format is well suited with respect to storage space for very sparse matrices with many empty rows, since the storage size is strictly proportional to the number of non-zero elements. Implementing SpMV on CUDA with this storage format requires doing atomic updates to the Y vector from parallel threads, which leads to a low performance. The Cusp implementation attempts to solve this problem by using parallel segmented reduction on shared memory within a warp and block before writing to Y. However, because shared memories are only visible to threads within the same block, results from different blocks still need to be combined in the global memory.

Compressed Sparse Row (CSR). Non-zeros are sorted by the row index, and only their column indices are explicitly stored in a column array. Additionally, the vector *row_start* stores indices of the first non-zero element of each row in the column array. The CSR Cusp implementation assigns one warp to each matrix row. Each thread in a warp computes the result of one non-zero at a time and then the warp moves to the next 32 elements. A parallel reduction operation is performed within a warp to get the final result of the row.

ELLPACK (ELL). Let K be the maximum number of non-zero elements in any row of the matrix. Then, for each row, ELL stores exactly K elements (extra padding is required for rows that contain less than K non-zero elements). As in the CSR format, elements are sorted by row index and only column indices are explicitly stored. In this format, the column array is stored in transposed manner allowing coalesced memory access. The storage size of the ELL format is proportional to $K \times N$. The Cusp ELL implementation assigns one thread per row. Each thread iterates K times accumulating the sum of the respective elements into a register. This format outperforms CSR if the number of non-zeros per row is relatively even. When the number of non-zero elements per row is uneven, overhead from extra padding elements increases the memory usage and decreases performance.

Sliced ELLPACK (SLE). This format partitions the matrix into horizontal slices of S adjacent rows [14]. Each slice is stored in ELLPACK format. The

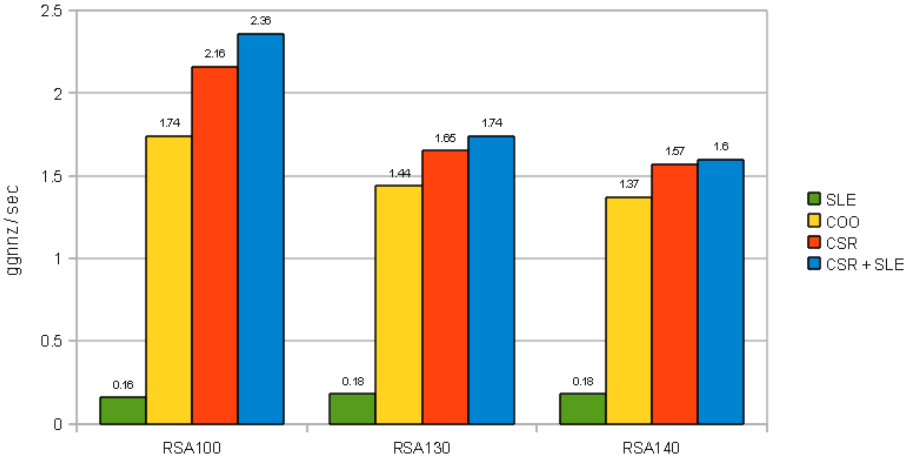


Fig. 1. SpMV performance comparison between sparse matrix formats for various NFS matrices in terms of giga non-zeros per second on a Tesla C2070 GPU with 64 bit blocking factor

Table 1. Properties of utilized NFS matrices resulting from factorizing integers with 100, 130, 140 and 170 digits, respectively

(a) Small Matrices

	RSA-100	RSA-130
Size	$284,836 \times 284,996$	$1,698,881 \times 1,699,041$
Non-zeros	26,274,784	192,416,939
Max row weight	118,252	731,247
Min row weight	11	11
Average row weight	92.24	113.2

(b) Large Matrices

	RSA-140	RSA-170
Size	$3,576,848 \times 3,577,008$	$10,463,019 \times 10,463,197$
Non-zeros	347,915,287	994,785,014
Max row weight	1,327,624	5,582,861
Min row weight	11	3
Average row weight	97.26	95.08

maximum number of non-zeros may be different for each slice. An additional array *slice_start* is used to index the first element in each slice. The matrix rows are usually sorted by the number of non-zeros per row in order to move rows with similar number of non-zeros together. Since there is to our knowledge no open-source SLE CUDA implementation, we have developed our own code. Our SLE CUDA implementation is similar to ELLPACK i.e. 1 thread per row. A requirement is that the height of each slice has to be divisible by the warp size

(32). This format adapts well to many sparse matrix types, and improves the memory usage compared to ELLPACK. However, there is still some overhead due to padding. The Variable-Height SLE [14] format can be used to reduce the overhead further.

We have modified the NVIDIA Cusp library to adapt with GF(2) operation and performed the comparison between the above formats on matrices from RSA-100, RSA-130 and RSA-140 factorization. A summary of these matrices is shown in Table 1. The result of our comparison in Figure 1 shows that CSR outperforms other formats. SLE is slower because of the uneven number of non-zeros per row in the dense part of the matrix. However, SLE performs better than CSR on the sparse part of the matrix. Thus, using CSR for the dense part and SLE for the sparse part improves the performance.

3 New Formats for SpMV on GPUs for NFS Matrices

As a preprocessing step, we reorder the rows of the matrix by their *row weight*, in non-increasing order. The row weight of row j of A is defined as the total number of non-zero elements in row j. We then partition the sorted matrix rows into at most four consecutive parts. Each part uses a different format. The different formats are optimized for the sparseness properties of each partition as shown in Figure 2. For the densest part, we use a dense format. When the matrix gets less dense, we switch to another format which we call Sliced COO. Sliced COO has three variants, small, medium, and large. Our formats are now described in more detail.

3.1 Dense Format

The dense format is used for the dense part of the matrix. This format uses 1 bit per matrix entry. Within a column, 32 matrix entries are stored as a 32 bit integer. Thus, 32 rows are stored as N consecutive integers.

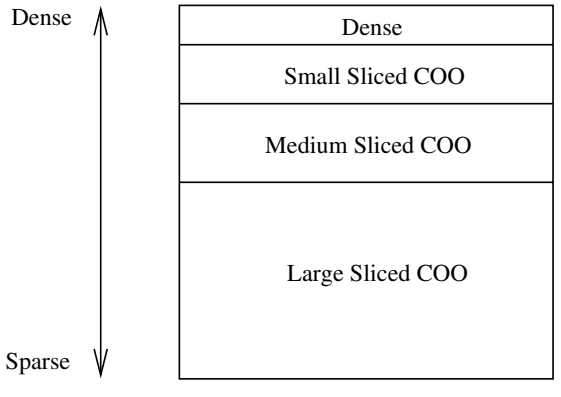


Fig. 2. Partitioning of a row-sorted NFS matrix into four formats

Each CUDA thread works on a column. Each thread fetches one element from the input vector in coalesced fashion. Then, each thread checks the 32 matrix entries one by one. When the matrix entry is a non-zero, the thread performs a XOR operation between the element from the input vector and the partial result for the row. This means each thread only accesses the input vector once to do work on up to 32 non-zeros. The partial result from each thread needs to be stored and combined to get the final result for the 32 rows. These operations are performed in CUDA shared memory.

The 32 threads in a warp share 32 shared memory entries to store the partial results from the 32 rows. Since all threads in a warp execute a common instruction at a time, access to these 32 entries can be made exclusive. The result from each warp in a thread block is combined using reduction on shared memory. The result from each thread block is combined using an atomic XOR operation on global memory.

When the blocking factor is larger than 64, access to the shared memory needs to be reorganized to avoid bank conflicts. Each thread can read/write up to 64 bit data at a time to the shared memory. If a thread is accessing 128 bit data for example, two read/write operations need to be performed. Thus, there will be bank conflicts if we store 128 bit data on contiguous addresses. We can avoid bank conflicts by having the threads in a warp first access consecutive 64 bit elements representing the first halves of the 128 bit elements. Then, the threads again access consecutive 64 bit elements representing the second halves of the 128 bit elements. The same modification can be applied to other formats as well.

The dense format is used for 32 to 64 dense rows of the RSA-170 matrix, which comprises about 15-24% of the total non-zeros in the matrix. This translates to a memory usage of at most 0.36 bytes per non-zero.

3.2 Sliced COO

The Sliced COO format is adapted from the CADO-NFS software for CPUs [9]. The aim is to reduce irregular accesses to the input vector and increase the texture cache hit rate. Sliced COO stores the column index and the row index of each non-zero. A number of consecutive rows form a slice. Non-zeros within a slice are sorted by their column index.

For each non-zero, two bytes are used to store the column index. However, two bytes are not enough for large RSA matrices. Thus, we further divide a slice into groups. Group i contains non-zeros with column index between $i \times 2^{16}$ and $(i+1) \times 2^{16} - 1$. An additional array stores the starting position of each group in the slice.

One thread block works on a slice, one group at a time. Neighboring threads work on neighboring non-zeros in the group. Each thread works on more than one row. Thus, each thread needs some storage to store the partial result and combine them with the result from the other threads. Since neighboring non-zeros may or may not come from the same row, we cannot share the entries in shared memory among the threads in a warp with exclusive access. Thus, shared memory is either partitioned among threads or shared using atomic XOR

operations. Based on the way we allocate the shared memory, we further divide the sliced COO format into three different subformats: small, medium, and large.

Small Sliced (SS) COO. In this subformat, each thread has one exclusive entry in shared memory to store the partial result for each row. The assignment of the shared memory is organized such that each thread in a warp accesses only one bank and there is no bank accessed by more than one thread. Thus, there is no bank conflict. A reduction operation on shared memory is required to combine partial results from each thread.

The maximum number of rows per slice is calculated as *size of shared memory per SM in bits / (number of threads per block * blocking factor)*. In Fermi, the size of shared memory per SM is 48 KB. We use 512 threads per block for 64 bit blocking factor which gives 12 rows, and 256 threads per block for 128 and 256 bit blocking factor, which gives 12 and 6 rows, respectively. Hence, one byte per row index is sufficient for this subformat.

Medium Sliced (MS) COO. In this subformat, each thread in a warp gets an entry in the shared memory to store the partial result for each row. However, this entry is shared with the threads in other warps. Access to the shared memory uses an atomic XOR operation. Each thread in a warp accesses only one bank, avoiding bank conflicts. A reduction operation on shared memory is required to combine the 32 partial results.

The maximum number of rows per slice is calculated as *size of shared memory per SM in bits / (32 * blocking factor)* where 32 is the number of threads in a warp. This translates to 192, 96, and 48 rows per slice for blocking factor of 64, 128, and 256 bit, respectively. Hence, one byte per row index is sufficient for this format.

Large Sliced (LS) COO. In this subformat, the result for each row gets one entry in shared memory, which is shared among all threads in the thread block. Access to shared memory uses an atomic XOR operation. Thus, there will be bank conflicts. However, this drawback can be compensated by a higher texture cache hit rate.

The maximum number of rows per slice is calculated as *size of shared memory per SM in bits / blocking factor*. This translates to 6144, 3072, and 1536 rows per slice for blocking factor of 64, 128, and 256 bit, respectively. We use two bytes for the row index.

3.3 Determining the Cut-Off Point of Each Format

To determine which format to use, we compare the performance of two consecutive formats in terms of giga non-zeros (*gnnz*) per second, starting with the dense format and the SS-COO format. The two formats start from the same row (starting from the first row) and work on the minimum number of rows possible. For the dense format, the minimum number of rows is 32. For the SS-COO format (and its variants), the minimum number of rows is the number of rows in

a slice times the number of multiprocessors in the GPU, since one thread block works on one slice and one thread block is assigned to one multiprocessor.

The next comparison depends on the result of the current comparison. If the dense format performs better, we decide to use it for rows 1 to 32, and we continue comparing the dense format and the SS-COO format starting from row 33. However, if the SS-COO format performs better, we compare its performance with the next format, MS-COO, starting from the same row, and so on. The idea is to stop considering the denser format once the sparser format outperforms it. Once we get to the comparison between MS-COO and LS-COO, and LS-COO performs better, we don't need to do any further comparisons. LS-COO should be used for the rest of the matrix.

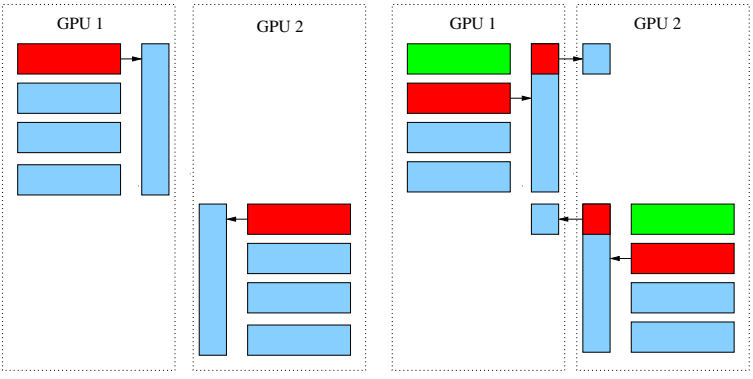
For Sliced COO format, it needs to be noted that when one slice is assigned to each multiprocessor, the load for one multiprocessor may be much higher than the other multiprocessors. This is because the matrix rows have been reordered by their weight in a non-increasing order, so the first slice contains more non-zero entries than the rest. Thus, we need to further reorder the rows such that each multiprocessor gets the same level of load.

3.4 Dual-GPU Implementation

Two GPUs are connected to PCIe slots and communicate to each other directly using the NVIDIA GPUDirect™ v2.0. To balance the workload between the two GPUs, we partition the matrix rows into two smaller matrices so that each has a similar number of non-zeros. Each smaller matrix is assigned to one GPU. Each GPU computes the multiplication with the complete input vector. The result from each GPU is half of the result vector.

As we need to perform multiple SpMV iterations, we have to combine the result from each GPU before moving on to the next multiplication. Our goal is to reduce the overhead of communication by overlapping computation and communication. We also take advantage of the fact that bi-directional PCIe data transfer has a higher bandwidth than uni-directional. Each half-matrix is divided further into several sub-matrices so that the computation and communication can be interleaved. This is illustrated in Figure 3. Note that the number of rows in each sub-matrix doesn't have to be equal to each other. The dense sub-matrix has fewer rows than the sparse sub-matrices.

There are two events where non-overlapping computation and communication occur. The non-overlapping computation occurs in the multiplication of the first sub-matrix. The non-overlapping communication occurs when the GPU sends the result of the last sub-matrix. To reduce the time spent on non-overlapping computation and communication, we do the multiplication in the order of the sparseness of the sub-matrix, sparsest first. The sparse sub-matrix multiplication is fast to compute because it has few non-zeros, so non-overlapping computation is minimized. The dense sub-matrix takes longer to compute, but has fewer rows than the sparse sub-matrices. Thus, the transfer size for the result is smaller, and non-overlapping communication is minimized.



(a) Dual-GPU Step 1 (b) Dual-GPU Step 2

Fig. 3. (a) GPU1 and GPU2 perform the SpMV iteration on the first sub-matrix and store the result in their own device memory. (b) After computing the result of the first sub-matrix, the result is transferred to the other GPU using GPUDirect™ v2.0. At the same time, GPU1 and GPU2 compute the next sub-matrix multiplication. In this step, two operations are executed at the same time. Bi-directional data transfer is utilized on each GPU.

In Figure 3, the boxes on the left and on the right side represent the two GPUs, the horizontal bars at each side represent small sub-matrices of each half-matrix and the vertical bars represent the result vector. The input vector is not shown in the diagram.

The process shown in Figure 3 continues until each GPU finishes every sub-matrix and the last result is transferred to the other GPU. At this point, both GPUs have the same result vector, and the next SpMV iteration can be started.

If GPUDirect™ v2.0 is not available, data transfer between GPUs is performed via the CPU. In this case, we can allocate a CPU vector to receive the result from each GPU before transferring it to the other GPU. The bi-directional data transfer can still be utilized, but the data transfer is less efficient because of the additional communication with the CPU.

4 Results

We have evaluated our implementation on an NVIDIA Tesla C2070 with 6 GB RAM and an NVIDIA GTX580 with 1.5 GB RAM. We have compared the GPU performance with the open-source CADO-NFS [9] program running on Intel Core i7-920 CPU with 12 GB DDR3-1066 memory. The RSA-140 and RSA-170 matrix (see Table 1) are used for performance evaluation. These matrices have been created by CADO-NFS [9] and Msieve [5,18], respectively. The performance is measured in terms of gnnz/s. CADO-NFS contains several CPU optimized SpMV implementations using multi-threading and SSE instructions. In Table 2 and 3, we have included the performance for the basic format (based on CSR)

Table 2. Performance in terms of gnnz per second for a single SpMV iteration with the RSA-140 matrix on one and two GPUs. The speedups compared to the multi-threaded CADO-NFS bucket implementation on an Intel Core i7-920 are given in brackets. The GPU memory required to store the sparse matrix and the corresponding bytes per non-zero (nnz) are also reported.

Blocking factor	GTX580 (speedup)	C2070 (speedup)	2 x C2070 (speedup)	Core i7-920		GPU memory (bytes/nnz)
				basic 8 threads	bucket 4 threads	
64	7.65 (7.0)	5.20 (4.8)	10.38 (9.5)	0.32	1.09	1177 MB (3.55)
128	4.20 (6.6)	2.83 (4.4)	5.64 (8.8)	0.27	0.64	1205 MB (3.63)
256	2.80 (8.0)	1.85 (5.3)	3.60 (10.3)	0.17	0.35	1207 MB (3.64)

Table 3. Performance in terms of gnnz per second for a single SpMV iteration with the RSA-170 matrix on one and two GPUs. The speedups compared to the multi-threaded CADO-NFS bucket implementation on an Intel Core i7-920 are given in brackets. The GPU memory required to store the sparse matrix and the corresponding bytes per non-zero (nnz) are also reported.

Blocking factor	C2070 (speedup)	2 x C2070 (speedup)	Core i7-920		GPU memory (bytes/nnz)
			basic, 8 threads	bucket, 4 threads	
64	4.28 (4.9)	8.38 (9.5)	0.31	0.88	2748 MB (2.90)
128	2.78 (5.6)	5.37 (10.7)	0.26	0.5	2967 MB (3.13)
256	1.88 (7.0)	3.68 (13.6)	0.16	0.27	3000 MB (3.16)

and the CPU-cache optimized bucket format of CADO-NFS with the number of threads that gives the best performance. Speedups of our GPU implementation are given compared to the faster bucket format.

Table 2 shows the result for the RSA-140 matrix. GTX580 achieves the best performance, with speedups between 6.6 to 8.0 over the Core i7-920 depending on the blocking factor used. C2070 achieves speedups between 4.4 to 5.3. Note that C2070 has the ECC (error correcting codes) memory disabled, since ECC reduces the GPU performance. As described in [11], a checkpointing approach could be used to replace the necessity of ECC memory. Comparing to the performance of existing sparse matrix formats with 64 bit blocking factor in Figure 1, our NFS optimized format is 3.25 times faster than the best performing format at a similar memory consumption on the same hardware. The dual-GPU implementation achieves speedups between 1.94 to 1.99 compared to the single-GPU performance.

Table 3 shows the result for the RSA-170 matrix. C2070 achieves similar performance to the RSA-140 matrix. Since storing the RSA-170 matrix requires almost 3 GB, the GTX580 cannot be used in this case due to lack of RAM. The dual-GPU implementation achieves speedups between 1.93 to 1.96 compared to the single-GPU performance.

Table 4 and 5 show that the performance in terms of gnnz per second decreases when the matrix gets sparser. The dense format is not used in RSA-140 matrix,

Table 4. Performance in terms of gnnz per second for each of the three sub-format partitions of the RSA-140 matrix on a C2070. The percentage of non-zeros (nnz) per partition is given in bracket.

Blocking factor	SS-COO	MS-COO	LS-COO
64	13.73 (20%)	8.30 (26%)	3.72 (54%)
128	7.16 (31%)	3.46 (6%)	2.16 (63%)
256	3.95 (29%)	2.25 (7%)	1.47 (64 %)

Table 5. Performance in terms of gnnz per second for each of the four sub-format partitions of the RSA-170 matrix on a C2070. The percentage of non-zeros (nnz) per partition is given in bracket.

Blocking factor	Dense	SS-COO	MS-COO	LS-COO
64	13.66 (24%)	9.66 (11%)	8.13 (13%)	2.77 (52%)
128	9.66 (15%)	7.53 (24%)	3.23 (6%)	1.86 (55%)
256	7.00 (15%)	4.21 (21%)	2.34 (6%)	1.33 (58 %)

since the dense rows are not dense enough for the dense format to outperform the SS-COO format. The MS-COO and LS-COO performance degrade when the blocking factor is increased from 64 to 128 and 256 bit. This is caused by the increased number of bank conflicts and serialization of atomic XOR operations on larger blocking factors. Thus, the SS-COO format gets a higher percentage of non-zeros with 128 and 256 bit blocking factor.

5 Conclusion and Future Work

We have presented our implementation of iterative SpMV for NFS matrices on GPUs with the CUDA programming language. Our single and dual-GPU implementation have been described in detail and both show promising improvements over an optimized CPU implementation. Our GPU implementation takes advantage of the variety of sparseness in NFS matrices to produce suitable formats for different parts.

Our current dual-GPU implementation on two C2070s is able to deal with NFS matrices containing up to 3 billion non-zeros. To deal with even larger matrices such as the RSA-768 [13] matrix ($\approx$ 28 billion non-zeros) and the SNFS-1024 [1] matrix ($\approx$ 10 billion non-zeros), we are currently working on extending our approach to a GPU-cluster.

Acknowledgements. We would like to thank Martin Krone for making the RSA-170 matrix available to us. We are also grateful to Emmanuel Thomé for his advises that enabled us to obtain performance numbers for CADO-NFS.

References

1. Aoki, K., Franke, J., Kleinjung, T., Lenstra, A.K., Osvik, D.A.: A Kilobit Special Number Field Sieve Factorization.. In: ASIACRYPT (2007)
2. Aoki, K., Shimoyama, T., Ueda, H.: Experiments on the linear algebra step in the number field sieve. In: Miyaji, A., Kikuchi, H., Rannenberg, K. (eds.) IWSEC 2007. LNCS, vol. 4752, pp. 58–73. Springer, Heidelberg (2007)
3. Bell, N., Garland, M.: Efficient Sparse Matrix-Vector Multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation (December 2008)
4. Bell, N., Garland, M.: Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations, version 0.1.0 (2010), `http://cusp-library.googlecode.com`
5. Bonenberger, D., Krone, M.: Factorization of rsa-170. Tech. rep., Ostfalia University of Applied Sciences (2010), `http://public.rz.fh-wolfenbuettel.de/~kronema/pdf/rsa170.pdf`
6. Boyer, B., Dumas, J.G., Giorgi, P.: Exact Sparse Matrix-Vector Multiplication on GPU's and Multicore Architectures. CoRR abs/1004.3719 (2010)
7. Choi, J.W., Singh, A., Vuduc, R.W.: Model-driven autotuning of sparse matrix-vector multiply on GPUs. SIGPLAN Not. 45, 115–126 (2010)
8. Coppersmith, D.: Solving Homogeneous Linear Equations Over GF(2) via Block Wiedemann Algorithm. Mathematics of Computation 62 (1994)
9. Gaudry, P., et al.: CADO-NFS (2010), `http://cado-nfs.gforge.inria.fr/`
10. Hwang, W., Kim, D.: Load Balanced Block Lanczos Algorithm over GF(2) for Factorization of Large Keys. In: HiPC, pp. 375–386 (2006)
11. Kleinjung, T., Nussbaum, L., Thomé, E.: Using a grid platform for solving large sparse linear systems over GF(2). In: 11th ACM/IEEE International Conference on Grid Computing (Grid 2010), Brussels Belgique (October 2010)
12. Kleinjung, T., et al.: A Heterogeneous Computing Environment to Solve the 768-bit RSA Challenge. Cluster Computing (2010)
13. Kleinjung, T., et al.: Factorization of a 768-Bit RSA Modulus. In: International Crytology Conference, pp. 333–350 (2010)
14. Monakov, A., Lokhmotov, A., Avetisyan, A.: Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. In: HiPEAC, pp. 111–125 (2010)
15. Montgomery, P.L.: A Block Lanczos Algorithm for Finding Dependencies Over GF(2). In: Theory and Application of Cryptographic Techniques, pp. 106–120 (1995)
16. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable Parallel Programming with CUDA. Queue 6, 40–53 (2008)
17. Nickolls, J., Dally, W.J.: The GPU Computing Era. IEEE Micro. 30, 56–69 (2010)
18. Papadopoulos, J.: Msieve (2010), `http://sourceforge.net/projects/msieve/`