Analysis and Performance Estimation of the Conjugate Gradient Method on Multiple GPUs

Mickeal Verschoor, Andrei C. Jalba

Abstract

The Conjugate Gradient (CG) method is a widely-used iterative method for solving linear systems described by a (sparse) matrix. The method requires a large amount of Sparse-Matrix Vector (SpMV) multiplications, vector reductions and other vector operations to be performed. We present a number of mappings for the SpMV operation on modern programmable GPUs using the Block Compressed Sparse Row (BCSR) format. Further, we show that reordering matrix blocks substantially improves the performance of the SpMV operation, especially when small blocks are used, so that our method outperforms existing ones, in most cases. Due to this, the performance of our CG method is in most cases better compared to other state-of-the-art approaches. Finally, a thorough analysis of the performance of both SpMV and CG methods is performed, which allows us to model and estimate the expected maximum performance for a given (unseen) problem.

Keywords: Conjugate Gradient method, Sparse-Matrix Vector multiplication, Block Compressed Sparse Row format, performance analysis, performance estimation, multiple GPUs.

1. Introduction

The *Conjugate Gradient* (CG) method [21] is a widely-used iterative approach for solving linear systems. At each iteration the method performs a *Sparse-Matrix Vector* multiplication (SpMV). For an $M \times M$ *Symmetric Positive Definite* (SPD) matrix, it provably converges in M iterations [21]. For non-symmetric matrices, more general iterative methods have to be used, such as the *BiConjugate Gradient* (BiCG), *BiConjugate Gradient Stabilized* (BICGSTAB) or other related approaches [2, 36, 34]. In practice, the number of iterations required by the CG method is smaller than M, but the speed of convergence depends on the conditioning of the matrix [35]. However, even if the method converges in fewer than M iterations, typically a large number of iterations is still needed. Therefore, improving the speed of the SpMV operation is an important task, and one way of achieving this is through parallelization.

Modern programmable Graphics Processing Units (GPUs) have a highly-parallel architecture that provides a vast amount of computing power. This, together with the large memory bandwidth of these devices, made GPUs an important means for accelerating certain scientific computations [14, 6, 7, 24]. However, mapping a specific algorithm on such a parallel architecture is in general non-trivial.

The CG and the related numerical methods mentioned above are in fact notoriously difficult to parallelize efficiently because of their low arithmetic intensity and high memory-bandwidth demands. Since these methods perform a large number of SpMVs, it is of paramount importance to achieve the best possible performance for such operations. Depending on the underlying problem, different representations can be used for efficiently storing a sparse matrix and multiplying it by a vector. Additionally, PDE discretizations based on structured grids (e.g. typically used with the Finite Difference Method) result in a structured matrix, for which specialized storage representations can be used, whereas discretizations based on unstructured grids (e.g. Finite Element-based on tetrahedral grids) yield unstructured matrices. Throughout this paper we mainly focus on problems yielding *sparse, unstructured matrices*. General formats for representing such matrices are the *Compressed Sparse Row* (CSR) and its extension – the *Block Compressed Sparse Row* (BCSR), see Section 2.

In this paper, first we propose fast BCSR-based GPU mappings for the SpMV operation using CUDA, see Section 3. As suggested elsewhere [4, 8] and confirmed by the results presented here, a block-based layout for SpMV fits very well with the computational model of a GPU. Therefore, we start with a basic mapping which is

subsequently transformed and optimized in a step-by-step fashion, so that in the end, we obtain an SpMV method that operates close to the limits of the hardware. Then, in Section 4, we propose an efficient GPU mapping of the CG method, based on our SpMV operation, accelerated on single- and dual-GPU setups. Further, in Section 5, we introduce a framework for analyzing the performance of SpMVs and various vector operations performed on GPUs. Since most of these operations are bandwidth limited, we investigate the behavior of the memory throughput with respect to the problem size. Furthermore, we expect the performance of the CG method to be mainly driven by that of the SpMV operation, while the scalability among multiple GPUs should be greatly influenced by the bandwidth difference between the GPU memory and the inter-GPU throughput. Indeed, since the communication bandwidth between multiple GPUs is an order of magnitude smaller compared to the memory bandwidth on the GPUs itself, not all problems scale well across multiple GPUs [17]. Thus, estimating the performance of such parallel systems gives insight into the scalability issue, and enables one to answer questions like 'How many GPUs can be used to solve the problem efficiently ?' or 'What performance can be expected for a given problem ?'.

Since all operations appearing in the CG method are bandwidth limited (low arithmetic intensity), the behavior of the data throughput is studied and modeled using a mathematical model, see Section 5. First, the performance estimates through the proposed model are compared and checked against actual (measured) performance figures, on a number of linear systems. Comparisons are performed for different settings of the SpMV operation, using one or two GPUs. Then, a number of maps are computed (through extrapolation), which allows one to estimate the maximum and average performance of the CG method, given some parameters of the matrix describing an (unseen) problem, see Section 6. Such performance estimates can be used to quickly check if the method performs well on the given hardware, thus allowing the user the possibility of considering a different hardware setup. For instance, the user can choose to increase/decrease the number of GPUs used, or he/she can decide to even use the CPU, for better performance. In Section 7 we discuss additional aspects influencing the overall performance of the CG method, such as scalability for future devices, matrix reordering schemes and performance figures for double-precision computations.

Please note that throughout this paper we focus on the CG method, but our model can also be used for the analysis of related Krylov-subspace methods, like the *BiConjugate Gradient* (BiCG) or *BiConjugate Gradient Stabilized* (BICGSTAB) method [2, 36, 34], or other numerical algorithms which perform SpMV and vector operations.

1.1. Previous and Related Work

The CG method and the SpMV operation have been implemented on various SIMD (multi-core) platforms. In [38] an overview is given on the performance of the CSR-based SpMV operation for a number of modern CPU architectures. A similar comparison is made in [37], where the CG method is implemented on Woodcrest CPUs and NVidia 8800GTX GPUs. The authors report a speedup of about 3 times when using the CSR format on the 8800GTX GPU.

GPU implementations of the CG and multigrid sparse solvers were presented by Bolz et al. [5]. Their methods rely on the programmable graphics pipeline of modern GPUs and were implemented using fragment shaders. Sparse matrices are stored in the CSR format, enhanced by an additional array for storing the main-diagonal elements. The work of [8] is closely related to ours, in that they present CUDA-based GPU mappings of the CG and SpMV operation using the BCSR format. However, since their methods are not optimized, e.g., by using coalesced memory accesses, the peak performance of the underlying hardware was not reached, see Figure 9.

Bell and Garland [4] propose several methods for efficient sparse matrix-vector multiplication, which take into account the structure of the input matrices. They implemented efficient multiplication routines for various sparse matrix representations, such as the Diagonal format (DIA), Row-packed format (ELLPACK/ITPACK), Coordinate list (COO), Compressed sparse row (CSR), Packet format (PKT) and a *hybrid format*. Their hybrid layout is most suitable for unstructured matrices and delivers in general the best performance for such matrices. This approach stores a part of the matrix using ELLPACK and the remaining elements using the COO format. It is known that ELLPACK becomes inefficient if the numbers of elements per row varies greatly [4, 25, 12]. Although extensive results were provided, a complete analysis of their methods was not performed. However, they did suggest that the use of blocks may potentially improve the performance even further, but this was left yet to be explored. This extension was first presented by Monakov and Avetisyan [25]. These authors introduced a hybrid SpMV operation as a combination of the BCSR and BCOO format. Splitting the matrix and choosing the representation format is done using dynamic

programming, but this approach has large memory requirements. According to their performance evaluation, 4×4 blocks seemed to yield the best efficiency, but no thorough analysis was performed to support this finding.

Cevahir et al. [11] propose an enhanced *Jagged Diagonals* (JDS) format, which reorders the matrix according to the number of non-zeros per row, and stores it similar to the CSR method. In [10] a parallel implementation of the CG method on a GPU cluster is presented. For the SpMV operation, their enhanced JDS format [11] along with the other formats from [4] were considered. For a given problem, the best layout for the SpMV operation is found by first benchmarking the performance of each individual format, and then selecting the one which delivered the fastest run. This is clearly disadvantageous, since first the input sparse matrix has to be off-line converted to a number of different layouts, which are then used to perform the SpMV operation.

In [12], the so-called *BELLPACK* method for SpMV multiplication is introduced. Although this method is similar to ours, there are also some important differences. In BELLPACK, first matrix blocks are identified, created and ordered, similar to [11]. After that, the ELLPACK method is employed on the ordered blocks. However, BELLPACK does not initially use coalesced memory transactions when loading the blocks. To improve on that the blocks are stored interleaved in memory, such that the memory transactions become coalesced, see [12]. As mentioned above, ELLPACK performs poorly when the numbers of elements per row varies greatly. This problem is addressed by sorting block rows, prior to storing them in the ELLPACK format. Since BELLPACK uses one CUDA thread to process one row, the number of registers used varies with the block size, making an analysis of the method more difficult. Within our method the number of registers used does not depend on the actual block size, which makes it easier to analyze its performance.

2. Background

2.1. The Conjugate Gradient Method

The CG method [21] is used to solve linear systems of the form

$$\mathbf{b} = \mathbf{A}\mathbf{x}, \tag{1}$$

with $\mathbf{A}$ a symmetric ($\mathbf{A}^T = \mathbf{A}$) and positive-definite matrix ($\mathbf{y}^T\mathbf{A}\mathbf{y} > 0$, with $\mathbf{y} \neq \mathbf{0}$), and $\mathbf{x}$ the vector of unknowns. In various textbooks (e.g. [33, 9, 2]), it is shown that the CG method minimizes the quadratic function

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{x}^T\mathbf{b}, \tag{2}$$

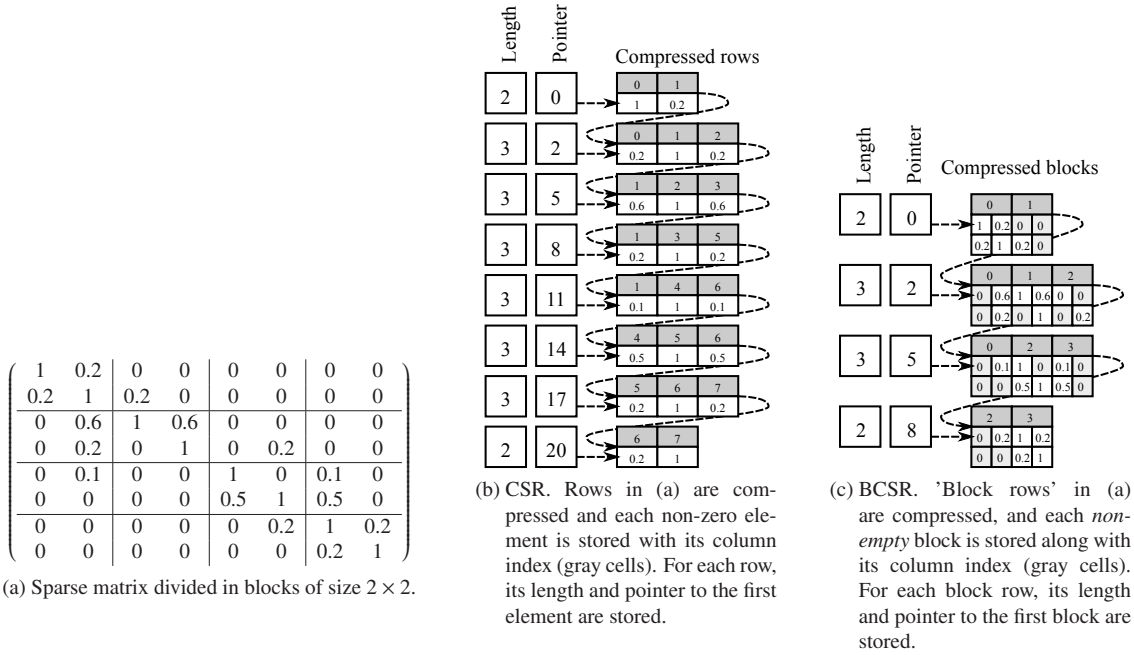
thus solving for $\mathbf{x}$ in Equation 1.

Algorithm 1 in Section 4 shows a parallel realization of the CG method, using a Jacobi preconditioner. Please note that it is straightforward to extend Algorithm 1 to other related methods, such as the Biconjugate gradient or the Biconjugate gradient stabilized methods. Further, the algorithm can easily be modified to accept other preconditioners, such as the Incomplete Choleski [18], although such preconditioners are very difficult to parallelize efficiently, because of data dependencies.

Due to the iterative nature of the method (see Algorithm 1), a large number of SpMV multiplications have to be performed when solving the linear system. Therefore, it is essential to optimize the SpMV operation as much as possible. Depending on the structure of the sparse matrix, different approaches exist to represent the matrix and to perform SpMVs on various types of hardware. Below, we briefly describe only the *Compressed Sparse Row* (CSR) and *Block Compressed Sparse Row* (BCSR) formats; see e.g. [4] for full details and other sparse-matrix storage schemes.

2.2. Compressed Sparse Row (CSR)

The CSR format is a well-known, general storage scheme suitable for *unstructured* matrices. Each non-zero element in a row and its column index are stored in two continuous arrays. Because of this, each row needs a *pointer* to the first element in the array of data elements and indices. The number of non-zero elements in a particular row can be determined by computing the difference between the pointer of the current and the next row. Figure 1b illustrates the CSR storage scheme.



(a) Sparse matrix divided in blocks of size 2×2.

(b) CSR. Rows in (a) are compressed and each non-zero element is stored with its column index (gray cells). For each row, its length and pointer to the first element are stored.

(c) BCSR. 'Block rows' in (a) are compressed, and each *non-empty* block is stored along with its column index (gray cells). For each block row, its length and pointer to the first block are stored.

Figure 1: Compressed Sparse Row (CSR) and Block Compressed Sparse Row (BCSR) storage schemes.

2.3. *Block Compressed Sparse Row (BCSR)*

BCSR constitutes a generalization of the CSR format. This scheme divides the input matrix of size $M \times M$ in blocks of $P \times Q$ elements, and stores each non-empty block similar to the CSR method, see Figures 1a and 1c. Each block row contains a number of non-empty blocks, and each block contains a number of non-zero elements. Note that zero elements inside a block are stored explicitly. For each block, the column index is stored, and for each block row, its length and pointer to the first block in the block row are represented, see Figure 1c.

Since all values of a block are consecutively stored in memory, the use of blocks reduces cache misses [19], when employed on CPUs, and improves the efficiency of memory transactions on GPUs. Therefore, larger block sizes should in principle lead to better performances, but they may introduce many zero elements. This usually results in wasted computations and memory space. We choose to use square blocks with size $N \times N$, with N a power of two. This fits best the architecture of many SIMD CPUs, as well as the architecture of modern GPUs.

The best block size with respect to performance depends on many aspects. Apart from the block density, also the problem size, hardware architecture and the runtime GPU configuration determine which block size gives the best performance; in Section 7.3, we shall further discuss this issue.

2.4. *CUDA Overview*

With the release of NVidia's G80 [29] and Tesla [23] series GPUs, general purpose computing was truly enabled via the so-called *Compute Unified Device Architecture* (CUDA). In this section we give a short overview of the architecture and its constraints, as exposed through CUDA; detailed information can be found in [26].

A typical modern GPU consists of a large number of *unified shaders* which can be used as either *vertex-, pixel- or geometry-shaders* in graphics applications. Within the context of general-purpose computing, a group of unified shaders is called a *multiprocessor* [26, 29, 31]. On a global scale, the multiprocessors are connected with the *global memory* of the device through a number of memory controllers. On a local scale, each multiprocessor contains a relatively small amount of memory that is shared among the scalar processors of the multiprocessor (*shared memory*).

The communication between the global memory and the individual scalar processors has a relatively high latency. Between a memory request and the instance when data is available, each processor has to wait between 400 and 800 clock cycles. However, each multiprocessor is able to execute up to 1024 threads, so that this latency can be hidden as follows. When a thread requests data from (non-cached) global memory, the scheduler activates another thread,

whereas the thread requesting data is put to sleep. Once data becomes available, the idle thread is activated and allowed to continue its execution. By scheduling a large amount of threads on a small number of scalar processors, most of the memory latency can be hidden.

Within CUDA, thread scheduling is done automatically. Launching a program on the GPU (*kernel*) creates a *grid* of *thread blocks*, the threads of which all execute the kernel function. Within each thread block, threads can communicate with each other through shared memory. However, threads belonging to different thread blocks cannot directly communicate, since thread blocks can be executed on different multiprocessors and at different time intervals. Each thread block is divided in smaller *warps* of 32 threads which are executed on the scalar processors. For multiprocessors having 8 scalar processors, each instruction is executed 4 times with different sets of threads. Depending on the shared memory and register requirements of a kernel, a multiprocessor can run concurrently a given number of thread blocks. The ratio between the number of active threads and the maximum number of threads per thread block (1024) is called the *occupancy* of the kernel [13]. For a typical problem solved using CUDA, a large number of thread blocks are created which are distributed over all available multiprocessors, so as to occupy all available resources. Once a thread block has finished its computations, a new thread block is started. This process is repeated until all thread blocks have performed their task.

Threads that belong to the same warp, execute the same instruction. If a thread within a warp follows a different branch of a conditional statement than the other threads, *thread divergence* occurs, and each thread follows both branches of the statement. If a particular thread does not need the results obtained by following one of the branches, these results are simply discarded. Although thread divergence reduces the overall performance of a kernel, in some cases it is unavoidable.

The architecture has several limitations also with respect to the pattern of global memory accesses. The global memory is divided in a large number of segments with a particular size. A typical memory segment is 128 bytes wide, depending on the version of the architecture. If all threads of a half-warp access data stored in the same segment of the global memory, only one combined memory transaction is initiated. This is known as a *coalesced memory access*. If threads access different segments, separate memory transaction must be initiated, which increases the total latency and should be avoided if possible. With the release of the Fermi architecture [31] global memory accesses are cached, which increases the data throughput.

Modern GPUs have a large amount of computing power compared to a single CPU. For example, a GTX570 has 15 multiprocessors each containing 32 scalar processors, which makes for a total of 480 scalar processors running at 1464 MHz. The theoretical peak performance of such a device is about 1405 GFlops, if only floating-point multiply and add instructions are performed. The theoretical memory bandwidth is about 152 GB/sec. In order to reach the best performance for a specific problem, some guidelines must be followed, see [26]. First, the memory transactions should be coalesced. Second, thread divergence should be avoided. Finally, the utilization level should be maximized, i.e., for a specific task, as much as possible computational resources should be used. In practice this means that a problem should be solved using a large amount of threads.

3. Proposed SpMV method using CUDA

The SpMV operation can be mapped to a GPU using different strategies, each with its own advantages and disadvantages. In general, mapping a certain problem to a GPU starts with identifying those parts of the algorithm that can run independently from other parts. Within the SpMV operation this is clearly the computation of a single element in the result vector. Throughout this section we shall seek for a GPU mapping of the SpMV operation which gives the best overall performance.

Thread blocks offer the lowest level of parallelism on GPUs, so they are used to compute one or multiple elements of the result vector. Figure 3 shows how a sparse matrix, stored using the BCSR format with $N \times N$ blocks, can be mapped to a GPU using CUDA. The actual mapping depends on the number B_y of *block rows* processed by each thread block, and the number B_x of *matrix blocks* processed collectively per block row, see Table 1. The total amount of thread blocks executed concurrently on a GPU depends on the number of multiprocessors and the number of active thread blocks, and represents the *occupancy* of a kernel [13]. Table 1 defines and describes each parameter appearing in Figure 3 for each different strategy. Since the *warp size* is 32, we use square blocks of dimension N, where N is a power of 2. This choice results in easy-to-implement kernels and reduces the amount of inactive threads. For each

different mapping, each matrix block is processed by threads with consecutive thread indices. For an $N \times N$ matrix block with index i, threads $i \times N \times N$ till $(i+1) \times N \times N - 1$ perform the computations and data lookup.

The following subsections describe the three proposed basic strategies presented in Table 1: *block row mapping*, *warp mapping* and *multiple block-row mapping*. Each strategy maps the computations differently among the available threads of one thread block. On top of the best basic strategy, we apply a few optimizations which further increase the performance, see Section 3.4, Section 3.5 and Section 3.6.

To estimate the efficiency of each mapping with varying block sizes, we compute what we call the 'raw-performance', defined as the number of GFlops achieved if each matrix block had a *density* of 100% , i.e., each block contained $N \times N$ non-zero elements. Figure 2 shows the raw-performance of each mapping for the set of matrices in Table 4. We prefer to use raw-performances instead of actual (measured) performances, because they better reflect the differences between each mapping.

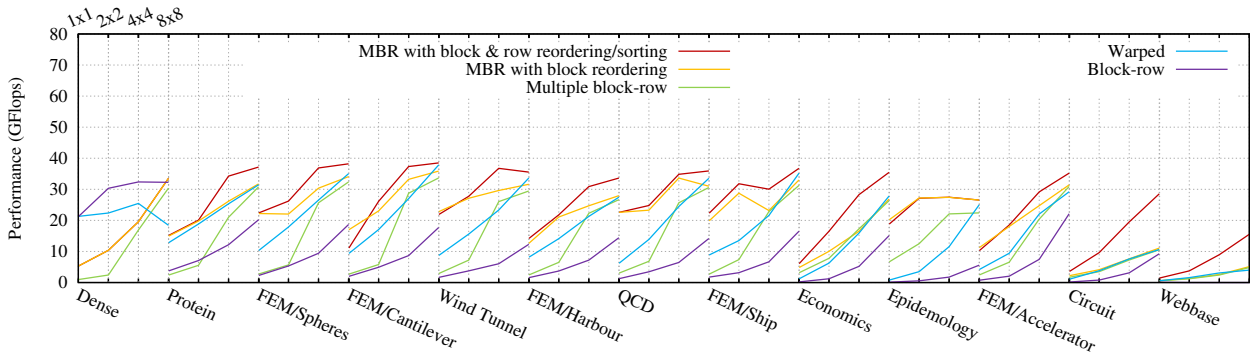


Figure 2: Raw performances for the SpMV operation by our mappings, with varying block sizes, performed using a GTX280 GPU.

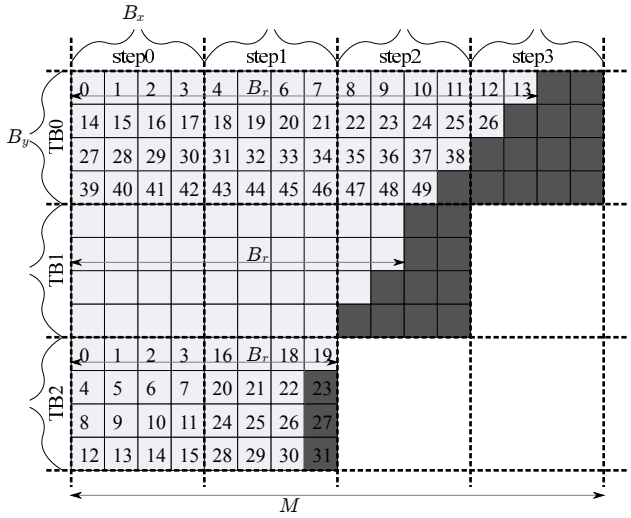


Figure 3: Mapping the sparse matrix on thread blocks. Depending on strategy, B_x and B_y are computed as in Table 1, given the number of threads T and block dimension N. The dark-gray regions indicate non-existing (empty) blocks used for padding. Each individual cell represents one matrix block with size $N \times N$. B_r indicates the number of blocks in the largest block row assigned to a particular thread block. Each thread block (TBi) performs a number of steps (proportional to B_r/B_x), to cover all assigned matrix blocks. During each step, each thread block processes collectively $B_x \times B_y$ matrix blocks containing $N \times N$ elements. Block rows were first *sorted* by length as described in Section 3.5. For thread block $TB0$, the initial indices of the matrix blocks, stored in the BCSR format, are shown. The matrix blocks assigned to thread block $TB2$ are reordered as described in Section 3.4.

3.1. Block-row Mapping

'Block-row mapping' assigns one block row to one thread block. In each *step*, each thread within a thread block loads one matrix element and its corresponding *vector element* from *global memory*. Each vector element is loaded by first computing its index using the *column index* of the matrix block and the *thread index*. Once both matrix- and vector-elements are loaded, they are multiplied and added to a per-thread intermediate value. When all matrix blocks of the current block row are processed, the intermediate values are *reduced* (addition operation) to a column vector of

Table 1: Parameter definitions for each mapping strategy, see Figure 3.

Parameter	Description	Block row		Warp		Multiple block-rows
B_x	Number of collectively processed *matrix blocks* per *block row*.	$\frac{T}{N \times N}$	$>_*$	$\frac{W}{N \times N}$	$>_{**}$	1
B_y	Number of collectively processed *block rows* per *thread block*.	1	$<_*$	$\frac{T}{W}$	$<_{**}$	$\frac{T}{N \times N}$
$B_T = \frac{M}{B_y \times N}$	Total number of *thread blocks* needed to cover the sparse matrix.	$\frac{M}{N}$	$>_*$	$\frac{M \times W}{N \times T}$	$>_{**}$	$\frac{M \times N}{T}$
Steps $= \frac{B_r}{B_x}$	Total number of steps needed to process all *matrix blocks* per *block row*.	$\frac{B_r \times N \times N}{T}$	$<_*$	$\frac{B_r \times N \times N}{W}$	$<_{**}$	B_r

N	Dimension of one *matrix block*, which is a power of 2. Usually $1, 2, 4$ or 8.
T	Number of threads per *thread block*, usually 128 or 256.
W	*Warp size* = 32.
M	Matrix dimension.
B_r	Largest number of *matrix blocks* per *block row* per *thread block*.

Remarks

$*$ Holds if $T > W$.

$**$ Holds if $N < 4$.

size N. Finally, after reducing N row vectors, the result is stored in global memory. Note that $B_y = 1$ for this strategy, i.e., one block row is processed by one thread block.

The number of collectively-processed matrix blocks per block row, B_x, is computed using the formulas in Table 1. Since each thread block processes exactly one block row, a large number of thread blocks (B_T) are needed to cover the whole matrix. Since in each step a large number of matrix blocks are processed together, the number of *steps* performed by each thread block is small. Thus, computations can be wasted if B_r is not a multiple of B_x. For example, if $N = 1$, then $B_x = T$, which implies that a number T of 1×1 matrix blocks are processed together. In general T equals 128 or 256, which means that B_r must be a multiple of 128 or 256 in order to minimize the number of wasted computations. Clearly, this is impractical in most situations. Note that increasing N gives in general better performances, since the amount of wasted computations is reduced, and the memory transactions for vector elements become more efficient.

3.2. Warp mapping

'Warp mapping' assigns one block row to one warp of threads. The computation of the individual elements is similar to the block-row mapping. The difference between both methods lies in the configuration parameters, see Table 1.

Since each block row is mapped to one warp, the number of collectively-processed matrix blocks per block row is smaller than with the block-row mapping strategy. This implies that the number B_y of block rows processed per thread block is larger. Since B_x is smaller, the amount of wasted computation is in general smaller compared to block-row mapping. Furthermore, the number of steps required to process one block row is increased, which results in less additional overhead and thus a higher *memory throughput*. Figure 2 clearly shows the performance improvement of this mapping compared to the single block-row case, except for matrix 'Dense'.

3.3. Multiple block-row mapping

'Multiple block-row' mapping (*MBR*) is the opposite of the block-row strategy. Instead of processing multiple blocks belonging to one block row, the mapping is transposed such that in each i-th step, each i-th block of the block rows is processed together. For each block row, exactly one block is processed, together with blocks belonging to other block rows. The actual computation of the result vector is similar to the previously described mappings, while the layout is different.

Within the MBR mapping, the number B_x of matrix blocks processed together per block row is exactly one. This implies that B_y should be as large as possible. Because $B_x = 1$, the number of steps required becomes exactly B_r. Furthermore, the number of necessary thread blocks decreases. Since the number of steps is maximized and the number of thread blocks is minimized (while the total amount of work remains constant), the work per thread is maximized. The main advantages of this approach are that the additional overhead is reduced and less space and computations are wasted. However, if the variation of row lengths of block rows (assigned to the same thread block) is large, a large number of threads *can* become idle. Another potential drawback is that matrix elements are loaded in a different order than they are initially stored, which results in un-coalesced memory accesses for specific block

sizes. Figure 2 shows that (for all matrices) the performance when $N = 1, 2$ is smaller compared to warp-mapping, due to non-coalesced memory transfers in these cases. However when $N > 2$, the performance is similar to that of the warp-mapping.

To overcome these problems when $N = 1, 2$, we use two reordering steps described below.

3.4. Block reordering

For each mapping strategy, each matrix block is processed by consecutive threads. Furthermore, blocks are stored at memory locations given by their indices. (The top-most thread block of Figure 3 shows these block indices.) Because of this, memory transactions are coalesced (when loading the matrix blocks) if $B_x \times N^2 \geq 16$. For the MBR mapping, coalescing becomes problematic if $N < 4$, since $B_x = 1$. Note that for other strategies this can never happen, because $T \geq W$. Figure 2 shows this effect happening when $N < 4$. For example, if $N = 2$ and $B_x = 1$, each half-warp loads data from 4 different blocks that are clearly not stored in the same memory segment. The first half-warp of the top-most thread block in Figure 3 loads blocks 0, 14, 27 and 39, where indices represent memory locations.

To overcome this problem, the order of the matrix blocks in memory must be changed, such that each thread within a half-warp reads from the same memory segment. In Figure 3 matrix block with index 14 of thread block $TB0$ is moved to position 4, block 4 to position 16, so that the final configuration is similar to that shown for thread block $TB2$. By reordering matrix blocks such that those blocks processed within the *same step* of the *same thread block* are consecutive in memory, all memory transactions (required for loading matrix blocks) become coalesced. In Figure 3, all blocks within $TB2$ are reordered such that all threads read from consecutive memory locations, while all threads assigned to $TB0$ read from non-consecutive memory locations. However, if $N = 4$, this problem is solved, since each half-warp reads exactly one matrix block from memory. Since blocks now are stored at different memory locations, all blocks that are loaded within each step are consecutive in memory, for any size of N. This reordering strategy boosts significantly the performance of the SpMV operation for matrix blocks with $N < 4$, as shown in Figure 2.

If block rows within one thread block have different lengths, or if B_r is not a multiple of B_x, empty matrix blocks must be added until each block row assigned to the same thread block has the same amount of matrix blocks. This ensures that each block row, processed by the same thread block, has a length which is a multiple of B_x (the dark-gray blocks in Figure 3). Further, since after padding, each block row processed by a thread block has the same length, we do not have to check the length of each individual block row. This eliminates the possibility that thread divergence occurs at this stage.

If the block rows are sorted by their length, described in Section 3.5, the number of additional empty blocks is reduced since block rows with similar lengths are processed together, see Figure 3. Furthermore, reordering the matrix blocks does not affect the structure of the matrix. Blocks are only stored at different memory locations, but the structure of the matrix remains untouched.

For looking-up vector elements used to multiply with one matrix block, the column index of the matrix block is required, see Figure 1c. These indices are stored in the same order as described previously. All threads that are used to load one particular matrix block, also load the column index for that block. Since each thread accesses the same memory location, and because column indices are stored consecutively in memory, this transfer is always coalesced. Using the column index, block-size and thread index, the index of the vector element is computed and used for looking-up the vector element. This lookup is only coalesced if $N \geq 4$. For smaller block-sizes, threads within a half-warp *can* read data from multiple memory segments. In this case, multiple transactions (one per memory segment) are needed.

3.5. Block-row sorting

Sorting block rows by their lengths results in an ordering so that block rows with similar lengths are spatially close to each other. Thus, sorting reduces the variation of block-row lengths for block rows assigned to the same thread block, as shown in Figure 3. This in turn results in a more balanced computation within one thread block, leading to an improved performance. Furthermore, the amount of empty matrix blocks used for padding (required after reordering the matrix blocks) is reduced. As shown in Figure 2, sorting improves the performance in most cases. Additionally, for matrices with a large variation in row-lengths (e.g. 'Circuit' and 'Webbase'), an even larger improvement is obtained, since the amount of added empty blocks is significantly reduced. Note that sorting block-rows does not influence the performance of matrix 'Dense'. This matrix has homogeneous row lengths, so sorting does not change the order of the rows.

Since in general, the order of the block rows is changed, one extra index per block row has to be used, such that the result is stored at the right position in the result vector. This ensures that sorting the block rows does not change the result of an SpMV operation. Since only one extra value is transferred per block row, the added overhead is negligible while the improvement can be significant. Note that sorting block rows improves the performance only if $B_y > 1$. If $B_y = 1$, no sorting is required since only one block row is processed per thread block.

3.6. Fine tuning

The MBR mapping yields in most cases the best performance. One drawback of this mapping is that the amount of thread blocks is relatively low compared to the block-row mapping strategy, while the number of computations per thread is high. For example, using the MBR mapping, matrix 'Dense' with $M = 2000$, 256 threads per thread block and $N = 1$, each thread block maps to 256 rows in the matrix. So, in total 8 thread blocks are used for the multiplication operation. Clearly, this number is too low to reach a good utilization of the GPU. However, by setting $B_x = 2$, twice the amount of thread blocks are created because B_y is halved, leading to better performance. Note that another way of increasing the amount of thread blocks is by increasing N, see Table 1. Additionally, for matrices having a high variation in row length, increasing B_x further reduces the number of empty blocks required for block-row padding, which also results in better performance figures, see Section 6.1.

3.7. Final SpMV mapping

Throughout this paper we use the MBR mapping combined with block-row sorting and block reordering. Because the number of thread blocks is the smallest, and the number of steps is the largest (Table 1), the amount of work per thread is the largest among our mapping strategies. In general, this results in a higher memory throughput, as shown in Figure 2. Finally, in a few cases it is worthwhile to increase B_x (see 'Best' performance in Figure 9), however for our performance analysis from Section 5 we have used $B_x = 1$.

Further, *textures* are used to enable cached memory accesses, and thus to improve the memory throughput when fetching vector values, similar to [4]. The effects of cached memory accesses is significant if $N < 4$. For $N \geq 4$, the improvements are minimal, since the memory transactions are already close to optimal. Finally, the results in Section 6 show that this method delivers the best performance in most of the test cases. However, if M is small, also B_T will be small. In these cases B_x should be increased, see above.

4. Parallel (Multi-GPU) Conjugate Gradient

Algorithm 1 shows the pseudo-code of our parallel (multi-GPU) Jacobi-preconditioned CG method, the steps of which will be explained throughout this section.

4.1. Parallel SpMV

Since the computation of one element in the result vector is independent from the computation of other elements in the result vector, the SpMV operation is parallelized by distributing the computation of the elements in the result vector over the available GPUs.

Because the BCSR format represents a sparse matrix by a collection of block rows, the matrix is divided in a certain number of segments of consecutive block rows, where each segment has a similar amount of matrix blocks. Each segment is then mapped to one GPU. Sorting the block rows as described in Section 3.5 is preferably done after the segments are created. If sorting is performed prior to segmentation, this results in an unbalanced GPU load, i.e., the block rows assigned to the first GPU will generally be longer than the block rows assigned to the other GPUs. Note that in order to perform an SpMV operation, vector $\mathbf{x}$ must be available to each individual GPU. Accordingly, sub-matrix $\mathbf{A}_i$ of (global) matrix $\mathbf{A}$, stored on GPU i, is multiplied by $\mathbf{x}$, see Figure 4. The result of the SpMV operation on GPU i, is $\mathbf{b}_i = \mathbf{A}_i\mathbf{x}$, where $\mathbf{b}_i$ is a vector which size corresponds with the number of rows of sub-matrix $\mathbf{A}_i$, lines 1 and 14 of Algorithm 1.

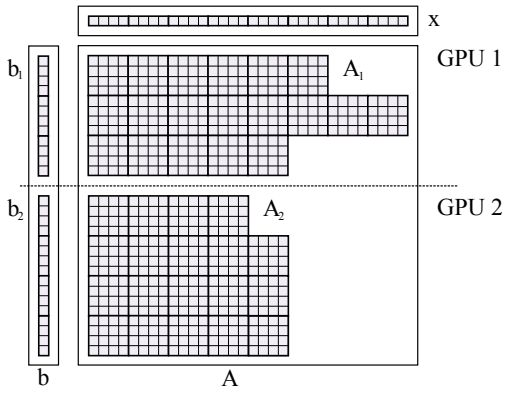


Figure 4: Block distribution over two GPUs. Each GPU is assigned a similar amount of matrix blocks: here, each GPU processes 19, 4×4 matrix blocks, distributed over a (different) number of block-rows.

4.2. Vector operations

Standard vector operations, like addition, subtraction, scalar multiplication or element-wise multiplication, can easily be parallelized. Figure 4 shows the distribution of vector $\mathbf{b}$ over two GPUs, where each part, $\mathbf{b}_i$, matches the number of rows of the sub-matrix assigned to GPU i, $i = 1, ..., n$, with n the number of GPUs. Other vectors appearing within the CG method are distributed similarly among the available GPUs, see Algorithm 1.

Since standard vector operations need to access vector elements with the same indices, no dependencies exist between the data segment of GPU i and the data of other GPUs, i.e., all required data is stored in the memory of GPU i. This means that it is possible to group such vector operations in one CUDA kernel, and thus perform a larger number of vector operations sequentially, without the need to synchronize the GPUs. For example, in Algorithm 1, all vector operations in lines 2 to 6 are performed by one CUDA kernel – CG1. This is advantageous, as shown in Section 5.2.

4.3. Vector reductions

A parallel vector reduction [20] using multiple GPUs requires more effort. Since each GPU contains only a part of a complete vector to be reduced, a final reduction needs to be performed among these parts to yield the final result, see function `parallel_reduction`. Before the final reduction can be computed (line 3), each GPU must have finished computing its (partial) reduction and must have stored its result in the host memory, line 1. By synchronizing among the GPUs, one can assure that each GPU has finished its work. Finally, each host thread (initiating computations on one GPU) computes the final reduction result, by collecting and reducing the results of all GPUs, which were previously stored in the host memory.

In order to prevent *race conditions*, a synchronization barrier among GPUs would also be required, after computing the final reduction per host thread. If this synchronization is omitted, a thread can for example perform a subsequent reduction and overwrite the previous reduction result. Preventing overwriting previously stored values, without using a synchronization point, can only be guaranteed if two successive parallel reductions use different storage locations. In Algorithm 1, function `parallel_reduction` uses the latter approach, which explains the need for three temporary vectors $\mathbf{tj}_i$, with $j = 1, 2, 3$. Although additional storage is required, the reduction in the number of synchronization points among GPUs results in improved overall performance.

4.4. Parallel CG

The CG method is parallelized by replacing each vector operation, reduction and SpMV operation, by their parallel equivalent, see Algorithm 1. Since the parallel SpMV operation requires a complete vector $\mathbf{v}$, and each GPU stores just a part $\mathbf{v}_i$ of the complete vector, that vector has to be reconstructed and updated at each iteration on each GPU. First, each vector part, $\mathbf{v}_i$, is copied to the host memory by host thread i; this is denoted by $\mathbf{v}_{\mathbf{host,i}} \leftarrow \mathbf{v}_i$ in line 11. Synchronizing among the GPUs (line 12) ensures that after the synchronization point, each individual part of $\mathbf{v}$ is available in the host memory. Then, each host thread copies the other parts of $\mathbf{v}$ to their assigned GPU, denoted by $\mathbf{v} \leftarrow \mathbf{v}_{\mathbf{host,j}}$ in line 13. Once $\mathbf{v}$ is reconstructed on a GPU, that GPU can start immediately performing the SpMV (line 14), followed by an element wise multiplication in line 15, which is denoted by '$\cdot*$'.

When vector $\mathbf{v}_i$ is computed on a GPU, the corresponding part of the complete vector $\mathbf{v}$, stored on the same GPU, is also updated. This is denoted by $\mathbf{v} \leftarrow \mathbf{v}_i$ in line 29, and is combined with other vector operations in order to increase the memory throughput.

Algorithm 1: Parallel CG on multiple GPUs.

Input : Matrix $\mathbf{A}$ and vector $\mathbf{b}$, $\mathbf{c}$: preconditioner, n: number of GPUs, i: index of GPU i, TOL: tolerance, and n_{iter}: maximum number of iterations.

Output: Vector $\mathbf{x}$, with $\mathbf{Ax} = \mathbf{b}$.

1 $\mathbf{r}_i = \mathbf{A}_i\mathbf{x}$ /* SpMV */;
2 $\mathbf{r}_i = \mathbf{b}_i - \mathbf{r}_i$ /* CG1 */;
3 $\mathbf{w}_i = \mathbf{c}_i \cdot *\mathbf{r}_i$ /* CG1 */;
4 $\mathbf{v}_i = \mathbf{c}_i \cdot *\mathbf{w}_i$ /* CG1 */;
5 $\mathbf{t1}_i = \mathbf{w}_i \cdot *\mathbf{w}_i$ /* CG1 */;
6 $\mathbf{t2}_i = \mathbf{v}_i \cdot *\mathbf{v}_i$ /* CG1 */;
7 $\mathbf{v} \leftarrow \mathbf{v}_i$ /* CG1 */;
8 α = parallel_reduction ($\mathbf{t1}_i, n, i$);
9 r = sqrt (parallel_reduction ($\mathbf{t2}_i, n, i$));
10 **for** $k_i \leftarrow 0$ **to** $n_{iter} \wedge r < TOL$ **do**
11 $\quad \mathbf{v}_{\mathbf{host,i}} \leftarrow \mathbf{v}_i$;
12 $\quad$ SyncGPUs();
13 $\quad$ **foreach** $0 \leq j < n \wedge j \neq i$ **do** $\mathbf{v} \leftarrow \mathbf{v}_{\mathbf{host,j}}$;
14 $\quad \mathbf{u}_i = \mathbf{A}_i\mathbf{v}$ /* SpMV */;
15 $\quad \mathbf{t1}_i = \mathbf{u}_i \cdot *\mathbf{v}_i$ /* CG3 */;
16 $\quad t = \alpha /$ parallel_reduction ($\mathbf{t1}_i, n, i$);
17 $\quad \mathbf{x}_i = \mathbf{x}_i + t\mathbf{v}_i$ /* CG4 */;
18 $\quad \mathbf{r}_i = \mathbf{r}_i - t\mathbf{u}_i$ /* CG4 */;
19 $\quad \mathbf{w}_i = \mathbf{c}_i \cdot *\mathbf{r}_i$ /* CG4 */;
20 $\quad \mathbf{t2}_i = \mathbf{w}_i \cdot *\mathbf{w}_i$ /* CG4 */;
21 $\quad \beta$ = parallel_reduction ($\mathbf{t2}_i, n, i$);
22 $\quad s = \beta/\alpha; \alpha = \beta$;
23 $\quad$ **if** $\beta < TOL$ **then**
24 $\quad\quad \mathbf{t1}_i = \mathbf{r}_i \cdot *\mathbf{r}_i$ /* CG2 */;
25 $\quad\quad$ **if** parallel_reduction ($\mathbf{t1}_i, n, i$) $< TOL$ **then**
26 $\quad\quad\quad$ **return** $\mathbf{x}$;
27 $\quad \mathbf{v}_i = \mathbf{c}_i \cdot *\mathbf{w}_i + s\mathbf{v}_i$ /* CG5 */;
28 $\quad \mathbf{t3}_i = \mathbf{v}_i \cdot *\mathbf{v}_i$ /* CG5 */;
29 $\quad \mathbf{v} \leftarrow \mathbf{v}_i$ /* CG5 */;
30 $\quad r$ = sqrt (parallel_reduction ($\mathbf{t3}_i, n, i$));
31 $\quad k_i = k_i + 1$;

Algorithm 2: parallel_reduction($\mathbf{x}_i, n, i$)

Input : Vector $\mathbf{x}_i$, n: number of GPUs, and i: index of current GPU.

Output: $\sum \mathbf{x}_i$

1 $r_i = \sum_{j=1}^{\#\mathbf{x}_i} \mathbf{x}_{i,j}$; /* Parallel reduction */;
2 SyncGPUs();
3 **return** $\sum_{j=0}^{n} r_j$; /* Collect and return */;

5. Performance analysis

In this section we analyze the performance of the CG method described in Section 4. Since the operations appearing in the CG method are bandwidth limited, the best performance is reached when the memory throughput is maximized. Thus, here we focus on analyzing the memory throughput of our method. First, the pure memory throughput is obtained for different kernel configurations. Next, the actual performance and scalability of the operations are estimated, which leads to the (*maximum* and *average*) performance estimate of the complete CG method. When multiple GPUs are used, also the memory throughput between the devices plays an important role. All observations are combined into a model, which is then used to estimate the theoretical maximum performance and the average performance of the CG method, given the properties of both the matrix and hardware. Furthermore, this model is also used to determine the scalability of the CG method, given an unseen linear system.

The analysis and performance estimations are performed on a machine equipped with an Intel Q6600 quad-core CPU and two NVidia GTX280 GPUs managed by an NVidia nForce 790i SLI chipset.

5.1. Memory throughput estimation

The CG method consists of two different types of operations: vector operations and the SpMV multiplication. Each kernel implementing these operations requires a suitable run-time configuration, in which the thread block and *grid* dimensions are specified. The thread block dimensions are in general fixed, while the grid dimensions can either be *fixed* or *variable*. To reveal the behavior of each configuration, we have performed a simple benchmark, in which a number of n vectors of a given size m are loaded from (global) memory. The results presented in Figure 5 were obtained by repeating this test for different vector sizes m and different numbers n of vectors. For each configuration the same amount of data is transferred.

For *fixed grids*, the minimum number of thread blocks B_T needed to fully occupy the device is given by $B_T = 1024\,P/T$, with 1024 the maximum number of threads per multiprocessor, P the number of multiprocessors and T the number of threads per thread block. In this case, each thread executes a loop, and in each step n vector elements are loaded.

With *variable grids*, $B_T = \lceil m/T \rceil$, so that the number of thread blocks directly depends on the problem size and the number T of threads per block. In this case, each thread loads exactly n vector elements. Therefore, the total number of thread blocks needed to cover the complete computation is much larger than with fixed grids. Since the amount of work per thread block is independent of m, the kernel start-up cost (overhead) per thread block is relatively high compared to the total time used by one thread block, which results in a lower memory throughput.

The variation in throughput with the number of memory transactions can be approximated using a *sigmoid function*, see Figure 5. For vector operations, fixed grids deliver in general the best performance, while variable grids also give satisfactory results, if each thread loads more than one vector, i.e., $n \geq 2$. Therefore, all vector operations are implemented using a fixed-grid approach.

If a fixed-grid approach is used for the SpMV operation, an extra loop would be needed in the CUDA kernel, which would result in a larger amount of registers being used. This would negatively affect the *occupancy* of the SpMV kernel, and so its performance. Thus, the SpMV operation (Section 3) uses a variable-grid approach. Since the SpMV kernel transfers a large amount of data in a loop, a high throughput is obtained, thus justifying the choice of a variable-grid.

Figure 5 shows the graph of the memory throughput versus the total number of transferred elements. We use the following scaled and shifted sigmoid function to model the memory throughput

$$B(x, \mu, \sigma, \nu) = \nu \left(1 + e^{-\left(\frac{\log_2 x - \mu}{\sigma}\right)}\right)^{-1}, \tag{3}$$

with x the total number of transferred elements, and μ, ν and σ, model parameters. Further, the Levenberg-Marquardt algorithm [33] was used to fit the sigmoid curve, leading to the (hardware-specific) parameters given in Figure 5. Note that all graphs below, in which Equation 3 is used, are plotted using a logarithmic scale.

In order to reach the best memory throughput, each multiprocessor must be fully occupied, i.e., each multiprocessor should have at least the maximum number of threads running. Also, each multiprocessor must initiate as many as possible memory transactions. For example, the NVidia GTX280 GPU can handle up to 1024 threads per multiprocessor and contains 30 multiprocessors, hence a minimum number of 30, 720 threads must be active to fully occupy

the GPU. If each thread initiates exactly one memory transaction, each kernel launch introduces a relatively large amount of overhead compared to the memory latency, hence the relatively low throughput in this case, see Figure 5. Further, to saturate the memory bus, each thread needs to initiate a large number of memory transactions, such that latencies can be hidden. If each multiprocessor is fully occupied, and each thread performs approximately 8 memory transactions, about 50% of the maximum throughput ν is used. According to Equation 3, this happens if $\log_2 x - \mu = 0$. Therefore,

$$\mu \approx \log_2(P \times 1024 \times 8). \tag{4}$$

Multiplying the total number of memory transactions again by 8, corresponds to 83% of maximum throughput ν. The maximum throughput, ν, is reached when the total number of memory transactions is multiplied again by 8, which corresponds to a total of 16×10^6 memory transactions.

Parameter σ represents the 'width' of the range in Figure 5, where the throughput increases the most. In Figure 5 this range starts where the GPU is fully occupied, with one memory transaction per thread, and ends where 83% of the maximum throughput is reached. The center of this range is clearly 2^μ, so that σ can be approximated by

$$\sigma \approx \frac{\mu - \log_2(P \times 1024)}{2} \approx \frac{\log_2(P \times 1024 \times 64) - \mu}{2}, \tag{5}$$

and will be about 1.5 for the GTX280 GPU. On other GPUs similar figures are expected, with the exception that the number of multiprocessors does affect parameters μ, σ and ν.

Equation 3 approximates the memory throughput for a specific GPU. If a particular kernel is bandwidth limited (similar to those implementing the SpMV and vector operations), one can use this approximation to estimate the total execution time of the kernel, given the problem size. However, the computations performed by a kernel do change slightly the estimation parameters.

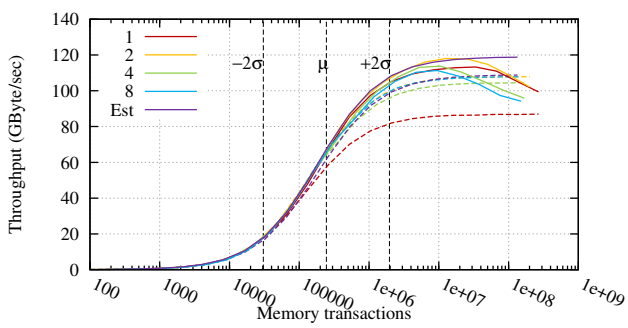


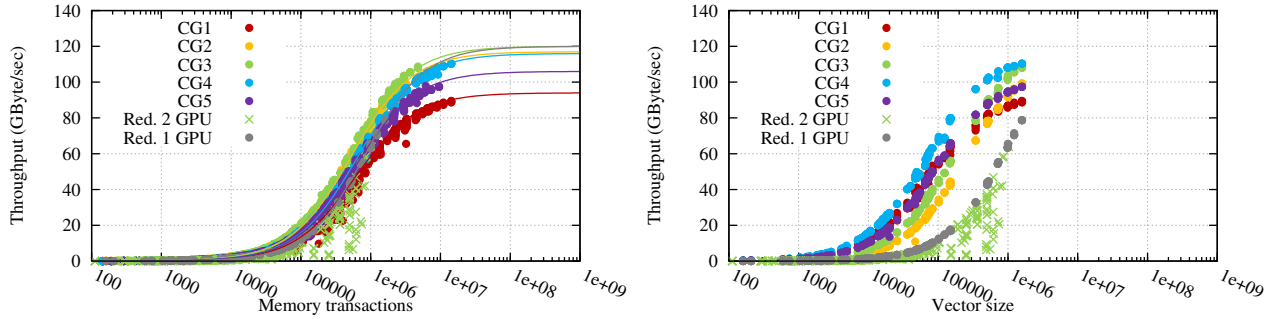
Figure 5: Measured and estimated ('Est') memory throughput for fixed grids (solid), with $\mu = 17.9, \sigma = 1.5, \nu = 119$, and variable grids (dashed), $\mu = 17.9, \sigma = 1.5, \nu = 109$, using an increasing amount $n = 1, 2, 4, 8$ of loaded vectors.

5.2. *Analysis of vector operations*

A number of vector operations, within the CG algorithm, can be *combined* into a few larger kernels. This allows better hiding of memory latencies, so that the performance is improved. Figure 6a shows the measured throughput for each combined vector operation and vector reduction ('Red.'), whereas Table 2 shows the number of floating point operations and transferred elements for each kernel. Further, Algorithm 1 shows which operations are executed by which kernel listed in Table 2. The data is obtained from both single and dual GPU benchmarks of the CG method, using a large set of test matrices[16]. The numbers of transferred elements and flops in Table 2, are obtained by counting the number of vectors loaded or saved by the kernel and by counting the number of floating point operations performed by that kernel. This information is obtained via Algorithm algorithm 1. For example, kernel CG1 load vectors $\mathbf{r}_i, \mathbf{b}_i$ and $\mathbf{c}_i$, and writes vectors $\mathbf{r}_i, \mathbf{w}_i, \mathbf{v}_i, \mathbf{v}, \mathbf{t1}_i$ and $\mathbf{t2}_i$, which makes for a total of $x \times 9$ memory transactions, with x the vector size. Furthermore, 5 floating point operations can be identified for kernel CG1. The other numbers in Algorithm algorithm 1 are derived similarly.

These results show trends similar to those in Figure 5. Note that these kernels actually perform a number of computations, so that slightly more time is consumed, which affects the maximum throughput. Furthermore, one must be aware that Figure 6a shows the performance versus the number of transferred elements. Figure 6b shows the performance as a function of the vector size x, in which large differences are visible between each kernel. A kernel

processing multiple vectors of size x, initiates more memory transactions. Such kernels will reach the maximum performance for smaller vector sizes, while kernels processing only one vector, reach the maximum performance for larger vectors. Therefore we have decided to combine as much as possible vector operations, such that the performance of these kernels is increased.



(a) Throughput versus memory transactions

(b) Throughput versus vector size

Figure 6: Measured throughput per combined CG vector operation as a function of the number of memory transactions (a) and as a function of the vector size (b). Lines represent performance estimates using the parameters in Table 2; the dots are the actual measurements. The graph contains results from both single and dual GPU setups. In the dual-GPU case, individual results of both GPUs are plotted.

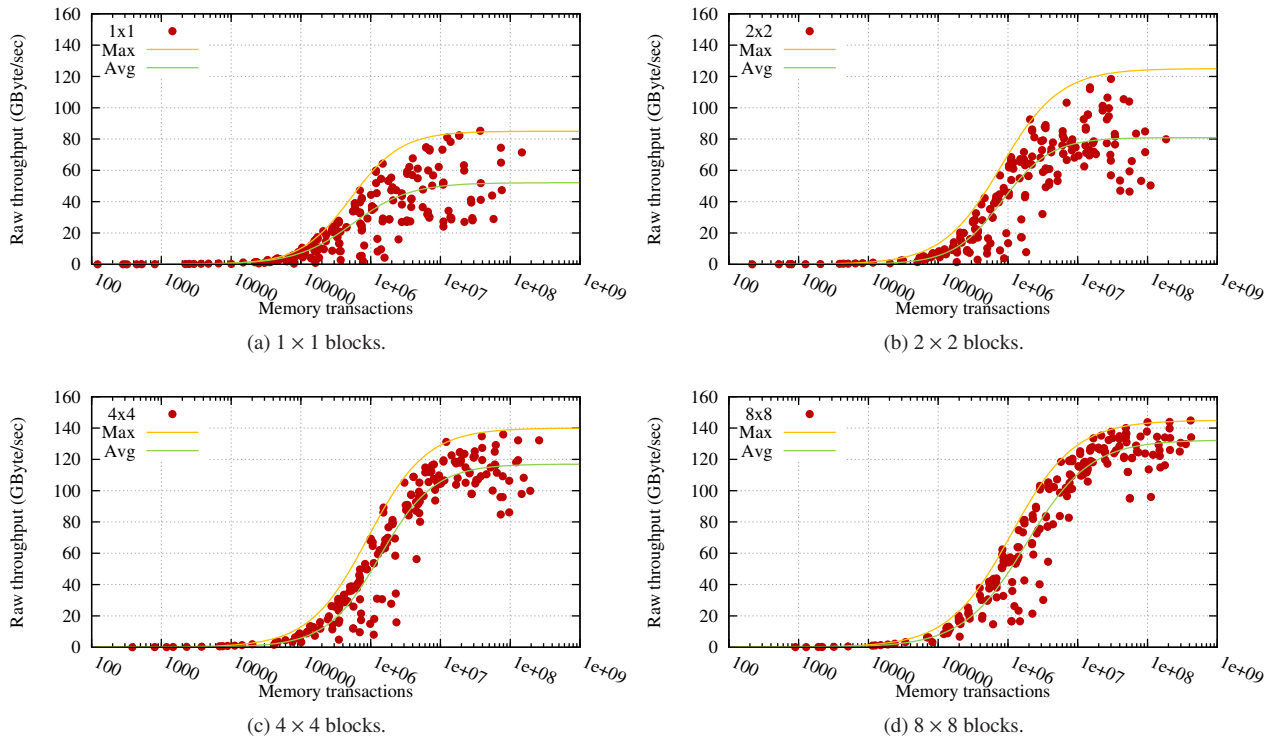
Kernel	transactions m	FLOPS	μ	σ	ν
CG1	$x \times 9$	$x \times 5$	19.2	1.45	94
CG2	$x \times 2$	$x \times 1$	19	1.45	117
CG3	$x \times 3$	$x \times 1$	19	1.45	120
CG4	$x \times 9$	$x \times 6$	19.3	1.45	116
CG5	$x \times 6$	$x \times 4$	19.2	1.45	106
Red	$x \times 1$	$x \times 1$	19.7	1.4	120

Table 2: Number of memory transactions and FLOPS for each kernel used in Algorithm 1. Here x is the vector size, and parameters μ, σ and ν are used to approximate the memory throughput of that specific kernel.

The performance of the vector reduction kernel, Figure 6b, is significantly lower compared to other vector operations. Since the reduction kernel performs only $x \times 1$ memory transactions, with x the vector size, this kernel only performs well for large vectors. Furthermore, parameter μ has a slightly higher value, because a complete vector reduction launches at least two kernels. For the dual-GPU cases, also some time is spent on synchronization among the devices explains why the estimations do not agree with the measurements, see 'Red. 2 GPU' in Figure 6a. Given these observations one can conclude that vector reductions can be problematic for the (parallel) CG method, even though an efficient algorithm [20] was used to implement them.

5.3. Analysis of the SpMV operation

Estimating the performance of the SpMV operation is more difficult because additional aspects of the matrix affect it, like the average density of the matrix blocks. By estimating *raw performances*, the block density is neglected, i.e., we consider that each matrix block contains 100% non-zeros, which artificially increases the number of non-zeros. By doing so, we get more insight in the behavior of the SpMV operation on the used hardware and the different kernel mappings, since the measurements are not affected by the average block density. However, variations between row lengths still influence the distribution of the computations, and thus the efficiency of SpMV. Figure 7 shows the *raw memory throughput* versus the number of memory transactions (per block size), which gives an estimate for the upper limit or maximum throughput for a particular block size and the average throughput. Note that Figure 7 contains measurements from both a single and dual GPU setup for the set of matrices in [16]. For the dual-GPU results, each measurement of each individual GPU is presented. The *upper limit* and average throughput are estimated using Equation 3 and the parameters are given in Table 3. Note that this behavior agrees with the observations in Figures 5 and 6. The 'average' throughput parameters were obtained by fitting the non-linear sigmoidal curve in the measured data using the Levenberg-Marquardt algorithm. The 'maximum' throughput parameters were obtained by adapting the average parameters such that the envelope of the measurements fits the sigmoidal function.



(a) 1×1 blocks.

(b) 2×2 blocks.

(c) 4×4 blocks.

(d) 8×8 blocks.

Figure 7: Measured raw throughput versus total number of memory transactions for the SpMV operation. The graph contains results from both single and dual GPU setups. In the dual-GPU case, individual results of both GPUs are plotted.

Table 3: The number of transactions and FLOPS for our SpMV operation using $N \times N$ blocks, where e is the total number of stored elements. Parameters μ, ν and σ are used to approximate the maximum raw memory throughput (performance) of that specific SpMV operation.

Block-size	transactions m	flops	μ_{max}	σ_{max}	ν_{max}	μ_{avg}	σ_{avg}	ν_{avg}
1×1	$2 \times e$	$2 \times e$	19	1.2	85	19.08	1.35	52.12
2×2	$2 \times e$	$2 \times e$	19.6	1.4	135	19.65	1.15	80.92
4×4	$2 \times e$	$2 \times e$	19.9	1.4	140	20.39	1.28	117.70
8×8	$2 \times e$	$2 \times e$	20.11	1.5	145	20.81	1.51	132.39

The difference between the measured throughput and its estimated upper limit is mainly caused by the variation of the row lengths. Further, if most rows contain less than 16 blocks, the expected throughput and performance is usually lower than the upper limit. 4×4 and 8×8 blocks usually deliver the best raw performances, whereas 2×2 and 1×1 blocks also have less efficient vector lookups. Thus, in general, the larger the blocks, the less the extra overhead, yet larger blocks *can* result in a lower block density, which degrades the overall performance. Furthermore, as noted in Section 3.7, if $N \geq 4$, the lookup of the vector elements are coalesced by default. This is also clearly visible in the results presented in Figure 7. For $N = 1$ and $N = 2$, this effect is visible as the variation of the measurements compared with the average throughput, i.e., the difference between the maximum and the average throughput. The measurements for $N = 4$ and $N = 8$ show that the maximum and average throughput are much more closer to each other compared to the smaller block sizes, as expected.

5.4. Scalability

The scalability of the CG method on a single GPU depends on the scalability of each individual operation. Vector operations scale well if the maximum memory throughput is reached. If the total number of memory transactions is larger than 5 million, these operations scale well, see Figure 6a. By combining multiple vector operations in a single kernel, the total number of memory transactions is increased for that kernel, which eventually increases

the performance, see Figure 6b. Unfortunately, vector reductions are problematic, because the amount of memory transactions is low and the operation itself cannot be combined with other vector operations. This results in a low memory throughput and a poor scalability. Furthermore, a vector reduction using multiple GPUs requires some additional synchronization among the devices, which further degrades the performance and scalability of the method.

The SpMV operation scales also well, if the maximum throughput is reached, i.e., when about 5 million memory transactions are initiated. For each element in the sparse matrix, roughly two values are looked up. One matrix block value, and one corresponding vector value. This implies that sparse matrices with more than 2.5 million elements *can* achieve a good scalability.

When the CG method is executed on multiple GPUs, vectors and matrices are divided in parts (segments), such that each individual GPU processes a part of the vector or matrix, see Section 4. This division also means that the total number of memory transactions per GPU is distributed among all available GPUs. Recalling Figure 5, a performance drop can be expected if the throughput does not reach the maximum. If the number of memory transactions is larger than 5 million, this performance drop is relatively small. Therefore, the individual operations scale well if the problem is large enough, although the communication between the devices does affect the scalability of the CG method significantly. Furthermore, when the CG method is executed using multiple GPUs, extra synchronizations are needed: one for updating vectors on each GPU, and one synchronization within each vector reduction. However, the time spent on each synchronization is constant.

5.5. Inter-device communication

When the CG method, or other similar methods, is executed on multiple GPUs, the GPUs need to communicate with each other in order to update their current result vector. Since on our test system, GPUs cannot directly exchange data with each other, each GPU transfers data via the PCI Express bus and the system memory. Because the bandwidth between the GPU and the system memory is limited, and in general, an order of magnitude lower than the bandwidth of the memory bus on the graphics card, this communication negatively and significantly affects the total performance, see Figure 8. The effective throughput between the GPUs is also estimated using Equation 3. On our test system we found the following parameters, $\mu = 16.7, \sigma = 1.4$ and $\nu = 10$ GByte/sec, indicating that the maximum throughput is reached when more then one million elements are transferred. Given this approximation, the total time for this operation can be estimated for different data sizes.

First generation GPUs were not able to directly communicate with each other. However, by involving the host memory, GPUs could indirectly exchange data, see above. Current generation GPUs (Fermi) and motherboard chipsets support direct communication between GPUs via the PCIe bus. With the release of CUDA 4.0, a large single address space can be created using *Unified Virtual Addressing* (UVA), which encompasses the memory of each individual GPU and the host memory. Therefore, by interchanging GPU pointers, GPU data can directly be exchanged. This simplifies the communication mechanism and improves the memory throughput, thus increasing the performance of the parallel CG method, see [32, 30, 26, 31].

5.6. Performance of the CG method

In order to evaluate the performance of the CG method, as shown in Algorithm algorithm 1, and given the properties of a specific problem, the maximum and average performances are estimated. Since all operations appearing in the CG method are bandwidth limited, the memory throughput is used to estimate the total running time. Given the number of transferred elements and the corresponding estimated memory throughput, the time per iteration $T(x, e)$ is obtained as follows,

$$T(x, e) = T_{spmv}(e) + T_{CG2}(x) + T_{CG3}(x) + T_{CG4}(x) + T_{CG5}(x) + 3T_{red}(x), \tag{6}$$

where T_{spmv} denotes the time spent for the SpMV operation, T_{CGi} on vector operations of kernel CGi, and T_{red} is the vector-reduction timing, e the total number of elements stored in the matrix (including the stored zeros) and x the dimension of the matrix and vectors. Note that kernel CG_1 is executed exactly once and does not contribute to the time per iteration. Each individual timing in Equation 6 is given by

$$T(x) = \frac{4m}{B(m, \mu, \sigma, \nu)}, \tag{7}$$

with m the amount of transferred elements (which is a function of x or e) and μ, ν and σ the fitting parameters; each timing can be estimated using the parameters and functions in Tables 2 and 3. Since Algorithm 1 executes approximately 3 vector reductions per iteration, the corresponding timing T_{red} is multiplied by a factor of 3.

The total number of floating point operations per iteration can be computed by

$$F(x, e) = 2e + 15x, \tag{8}$$

with x the dimension of the problem and e the total number of elements stored in the matrix, *including* the zeros stored in non-empty blocks. The *raw performance* is then given by

$$P(x, e) = \frac{F(x, e)}{T(x, e)}, \tag{9}$$

were T end F are given by Equations 7 and 8.

Figure 8a shows the *estimated raw performance*, $P(x, e)$, of the CG method on one GPU. In order to estimate the *real performance*, the raw performance of the SpMV operation is multiplied by the average density of the matrix blocks, i.e.,

$$F(x, e, d_N) = 2ed_N + 15x, \tag{10}$$

with d_N the average matrix-block density for a $N \times N$ block. Furthermore, if the matrix contains rows with uneven lengths, a large number of computations are not contributing, due to the added empty blocks (Section 3.5). Note that it is difficult to quantify this lost performance, because it requires a lot more information about the input matrix and the used hardware.

5.7. Performance of the parallel CG method

The parallel performance of the CG method is estimated similarly, but two GPUs run in parallel with approximately 50% of the data. In this case, the communication between the devices and the synchronization time also affects the performance of the parallel CG method. In general, the total time per iteration using n GPUs becomes

$$T_p(x, e) = \max\left(T_1\left(\frac{x}{n}, \frac{e}{n}\right), T_2\left(\frac{x}{n}, \frac{e}{n}\right), \ldots, T_n\left(\frac{x}{n}, \frac{e}{n}\right)\right) + T_{idc}(x\,n) + 4T_{sync}, \tag{11}$$

where T_i is the computation time of GPU i, for the sub-matrix (segment) stored on that GPU, T_{idc} is the time spent on communication (Section 5.5) and T_{sync} is the average time spent for synchronization ($35\mu s$ on our test system). Each GPU has to copy an x/n-sized vector to the main memory, then $(n-1)$, x/n-sized vectors are copied to n GPUs, which yields a total data transfer of $x\,n$ elements. If GPUs are able to communicate directly using the *Unified Virtual Addressing*, each GPU transfers x/n elements, hence $T_{idc}(x\,n)$ becomes $T_{idc}(x)$.

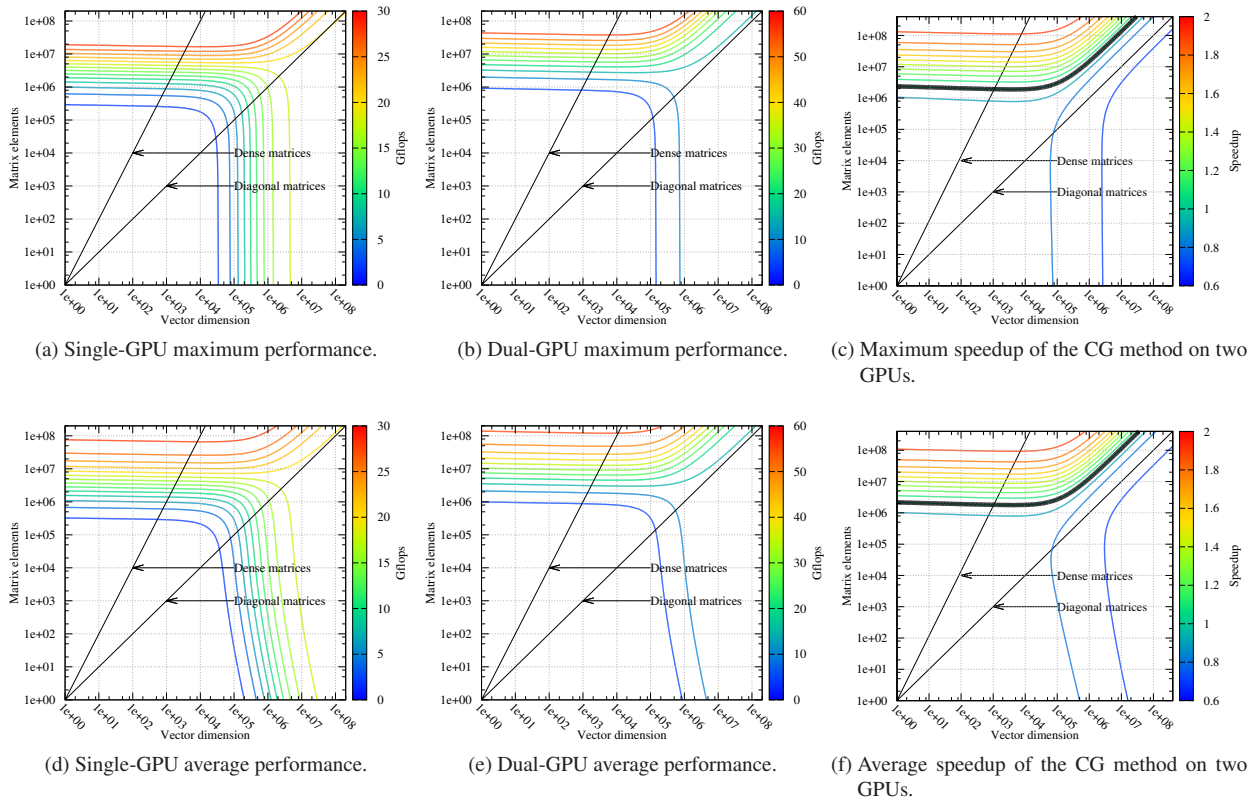
Figure 8a shows the maximum raw performance of the CG method, executed on one GTX280 GPU. In general, the more elements a matrix has and the denser it is, the better the performance becomes. If the density of a sparse matrix is relatively high, the relative time spent on vector operations is quite small, such that the SpMV operation dominates the computation. If the dimension of the matrix is larger, or the matrix is sparser, the relative time of the SpMV operation becomes smaller, so that vector operations influence more the total performance. Further, if the matrix is large and very sparse, also the communication time becomes important when multiple GPUs are used. This behavior is shown in Figure 8b. For example, having a relatively dense matrix containing approximately 10^5 elements, a performance of a few GFlops is to be expected at most. If the dimension increases, while the number of elements remains the same (i.e. the matrix becomes sparser), the performance will increase. In that case the performance of the vector operations dominate the total computation time. Increasing the dimension, results in better performances for the vector kernels. Contrary, having a relatively dense matrix containing 5 million elements, results in a maximum raw performance of about 20 GFlops. Increasing the dimension, thus making the matrix sparser, results in a decrease of the raw performance. In this case the SpMV operation dominates the total computation time. By increasing the dimension, more time is spent on vector operations. But at that scale, they do not yet reach the maximum performance, see Figure 6b. For a matrix containing about 2.5 million elements, the SpMV and vector operations seem to be balanced.

Finally, Figure 8c illustrates the speedup $S = T/T_p$ of the CG method on the two GTX280 GPUs, given the parameters of the problem. The thick black line represents the region where the speedup $S \approx 1$. For problems falling

below that line, a slowdown can be expected, while for cases above that line, a speedup should normally be obtained. For different systems and GPUs the exact location of this line may vary. This 'map' quickly shows if it is worthwhile to use multiple GPUs for a particular problem.

Figures 8d - 8f show the average raw performance using one and two GPUs and the speedup when two GPUs are used. For these plots the average parameters in Table 3 were used. Section 6.2 compares the timing results for a large set of test matrices with the average estimated time derived by applying the method described in this section.

The density of the matrix blocks d_N, does not have a significant influence on the speedup, i.e., on the blocks stored on each GPU, the average densities are similar, such that the performances on each GPU are similar. Hence, the speedup will not be affected significantly.



(a) Single-GPU maximum performance.

(b) Dual-GPU maximum performance.

(c) Maximum speedup of the CG method on two GPUs.

(d) Single-GPU average performance.

(e) Dual-GPU average performance.

(f) Average speedup of the CG method on two GPUs.

Figure 8: Estimated maximum (a)-(b) and average (d)-(e) raw-performances and speedup ((c) and (f)) for the CG method using 4×4 matrix blocks. Black lines depict dense and diagonal matrices, whereas the region between them represents sparse matrices. The black curve in (c) and (f) represents the area in which the speedup is approximately one.

6. Results

In this section the results of our SpMV implementation and (parallel) CG method are presented. Both our SpMV and CG methods were benchmarked using different collections of test matrices/problems. The machine used for benchmarking was equipped with an Intel Q6600 quad-core CPU and two NVidia GTX280 GPUs managed by an NVidia nForce 790i SLI chipset. Additionally, Sections 6.1 and 6.4 also present results using an NVidia GTX570 GPU.

First, the results of our SpMV implementation are compared to those in [4, 8, 28]. Next, the actual and estimated performances of our CG method using one and two GPUs are used to verify the model presented in Section 5. Finally, we compare the performance of our CG implementation to a similar implementation using the CUSP library[3], on different GPUs and with different precision settings.

Name	*Dim* ($\times 10^3$)	*nz* ($\times 10^6$)	*max*	*avg*	d_2	d_4	d_8
Dense	2×2	4	2000	2000	1.00	1.00	1.00
Protein	36×36	4.3	204	119	0.86	0.66	0.48
FEM/Spheres	83×83	6	81	72	0.77	0.55	0.34
FEM/Cantilever	62×62	4	78	64	0.75	0.54	0.35
Wind tunnel	217×217	11	180	53	0.82	0.70	0.51
FEM/Harbour	46×46	2.3	145	50	0.77	0.55	0.37
QCD	49×49	1.9	39	39	0.64	0.47	0.28
FEM/Ship	140×140	7.8	102	55	1.00	0.63	0.34
Economics	206×206	1.2	44	6	0.29	0.10	0.04
Epidemology	525×525	2.1	4	4	0.41	0.20	0.05
FEM/Accelerator	121×121	2.6	81	21	0.64	0.23	0.08
Circuit	170×170	0.95	353	6	0.40	0.15	0.06
Webbase	1000×1000	3.1	4700	3	0.47	0.21	0.09
Rail	1092×4	11279	56181	3	0.48	0.20	0.03

Table 4: Properties of the test matrices from [38]. *Dim* is the dimension of the matrix, *nz* the total number of non-zeros, *max* the largest number of non-zeros in a row and *avg* is the average number of non-zeros per row; d_N is the average block density when the matrix is represented using $N \times N$ blocks. $d_1 = 1$ for all matrices.

6.1. SpMV: performance comparison

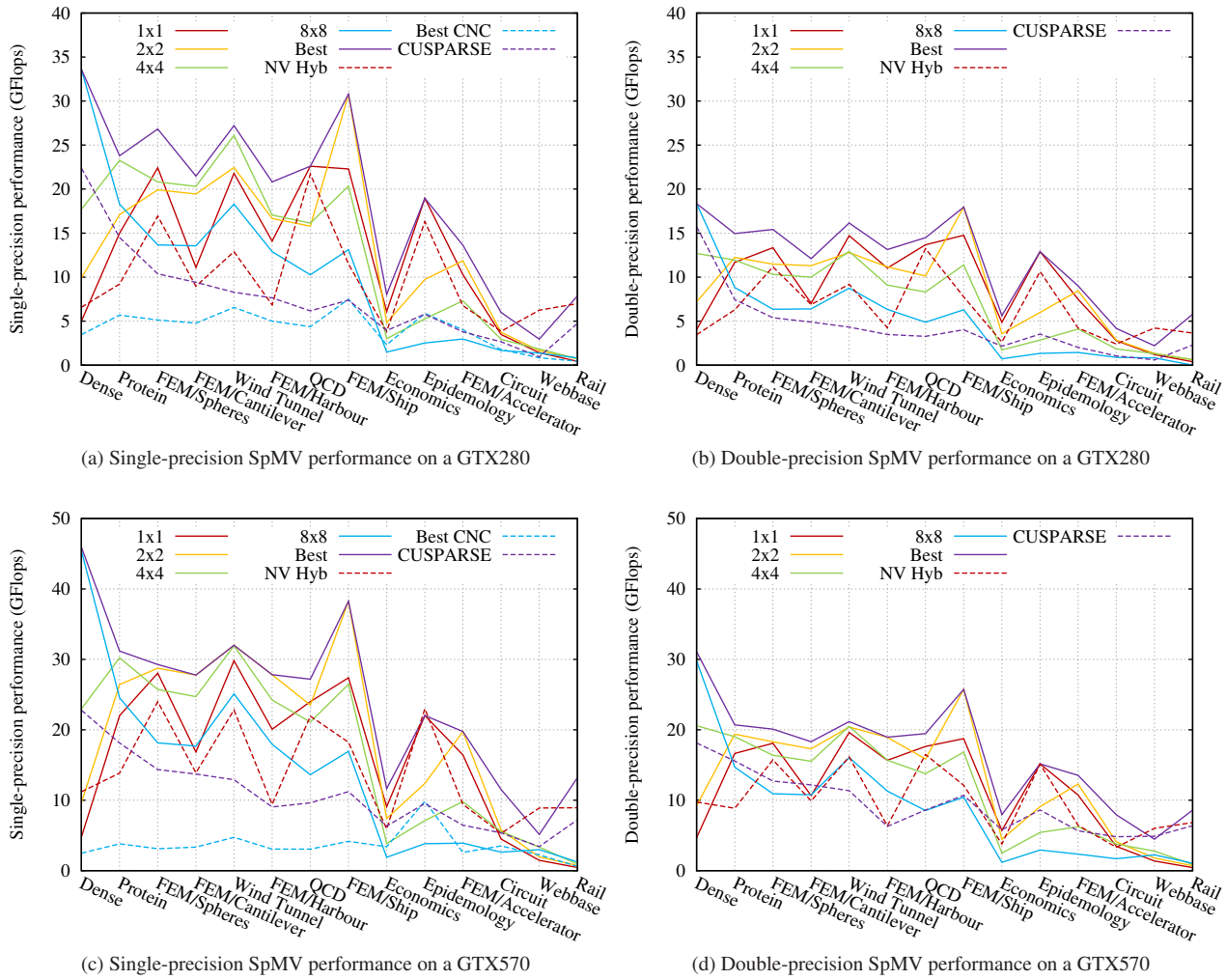
Figure 9 shows the results of our SpMV approach as described in Section 3.7 for varying block sizes. 'Best' denotes our best result after increasing parameter B_x as described in Section 3.6. 'NV Hyb' denotes the hybrid method of Bell and Garland [4] (implemented in CUSP 0.2, [3]), 'Best CNC' shows the best result obtained by Buatois et al. [8] and finally, 'CUSPARSE' denotes the result of the CUSPARSE library [28] from CUDA 4.0 in which the CSR storage was used. The test set used in this benchmark was introduced in [38] to evaluate the performance of the SpMV operations on various multi-core platforms; some of the properties of the test matrices are given in Table 4. Finally, we have performed this benchmark on both a GTX280 and a GTX570 GPUs, in single and double precision.

The results in Figure 9 show that our method gives the best performance in most cases, except for matrices that have very few non-zero elements per row. Since such matrices have in general a low average block density (d_N in Table 4) for larger blocks, they generally cannot benefit from the BCSR storage format. For example, in the worst case scenario, the usage of 2×2 blocks means that 75% of the computations and memory throughput are useless since blocks contain only one non-zero element. In fact, matrix 'Economics' represents just such an example: it has on average a block density of 29% for 2×2 blocks.

Matrices 'Circuit', 'Webbase' and 'Rail' also exhibit poor performance figures. Table 4 shows that the number of non-zeros per row is highly unbalanced. Matrix 'Webbase' has on average 3 elements per row, yet a few rows contain several thousands elements. In such cases, a lot of threads become idle, while others are busy with processing very large block rows. Combining this with the low number of non-zeros per row, makes that these matrices are difficult to process on GPUs using BCSR. By changing the layout of the mapping as discussed in Section 3.6, the performance is improved, see 'Best' performance.

Matrix 'FEM/Ship' shows a large performance boost for 2×2 blocks compared to 1×1 blocks. Table 4 shows that $d_2 = 1$, which means that each block contains 4 non-zero elements. Hence, no computing resources are wasted. This clearly shows the benefit of the BCSR layout compared to the others, if blocks have high densities d_N.

Since the test set in [38] is small, we have also benchmarked the implementations of Buatois et al. [8], Bell and Garland [4] and ours on a substantially larger set of matrices. This large set contains all matrices from Harwell-Boeing, SPARSKIT, the sample collection of The University of Florida [16], Williams et al. [38] and the matrices described in Table 5, making for a total of 486 matrices. We have benchmarked each implementation on a GTX570 GPU. We found that in 95% of the cases our method performs better than the hybrid implementation of Bell and Garland. Further, we measured a median speedup of about 8× and a total speedup, with respect to the total computation time (wall-clock time), of about 1.25×. Because of the large difference between the median and total speedups, after careful inspection of the results, we found that our method performed in 3 cases substantially worse than the method of Bell and Garland. It turned out that these matrices have highly unbalanced row lengths, which made our method to perform poorly, see discussion in Section 3. After neglecting these outliers, the speedup becomes 2.5× in favor of our method. The method of Buatois et al. [8] performed in 33% of the cases better than ours and significantly better than the hybrid method of Bell and Garland [4]. This happened for most of the matrices contained in the Harwell-Boeing set. However, in all these cases it turns out that a parallel CPU implementation of Algorithm 1 (using 4 threads on our quad-core CPU) was still faster than *any* of the GPU methods. Finally, comparing the total computation time for the



(a) Single-precision SpMV performance on a GTX280

(b) Double-precision SpMV performance on a GTX280

(c) Single-precision SpMV performance on a GTX570

(d) Double-precision SpMV performance on a GTX570

Figure 9: Single and double precision performances for the SpMV operation using various storage formats performed on a GTX280 and a GTX570 GPUs. Numbers in the legend indicate block sizes for our method, 'Best' indicates our best results after increasing parameter B_x as in Section 3.6, 'NV Hyb' represents the hybrid format of [4, 3], 'Best CNC' – the best result obtained by the method in [8] and 'CUSPARSE' – [26] the CSR implementation from the CUSPARSE library in CUDA 4.0.

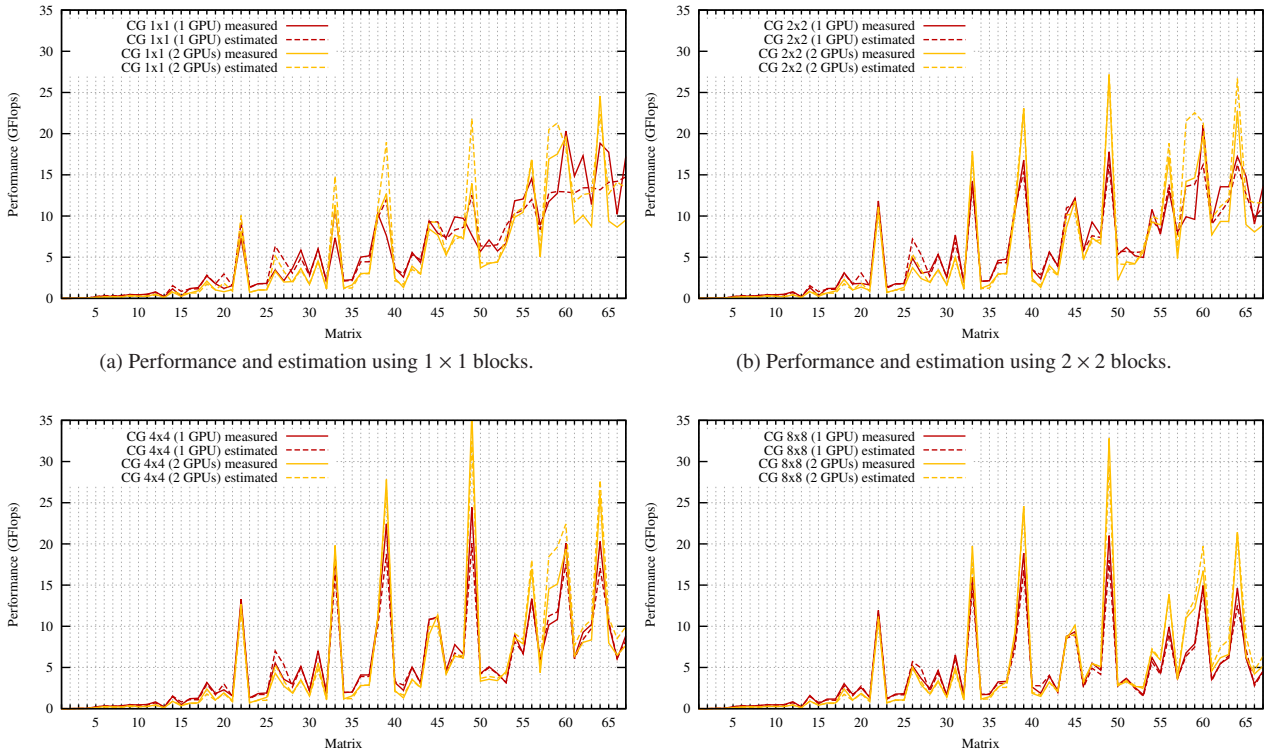
complete set showed that our method was about 3.7× faster than the best implementation of Buatois et al. Note that by neglecting the three outliers mentioned above, our method performed 6.1× better than their method.

Dense matrix-vector multiplication

The very good results obtained by our method for matrix 'Dense' inspired us to use our representations for sparse matrices to perform matrix-vector multiplications for *dense* matrices. We gradually increased the dimensions of a square and dense matrix from 10 to about 8000, and measured the time taken to perform the multiplication by our methods and function `cublasSgemv` from NVidia's CUBLAS library, [27]. Our multiple-block-row mapping using 2×2 blocks yields exactly the same performance as the CUBLAS function, whereas using 1×1 blocks performed poorly compared to CUBLAS. All other combinations, except Single-Block-Row with 1×1 blocks, perform better than CUBLAS: the maximum performance is about 15% higher. More importantly, Single-Block-Row with 8×8 blocks reaches the peak performance at problem sizes 10 times smaller than when using the CUBLAS function. We chose to use the Single-Block-Row mapping since the Multiple-Block-Row mapping is not efficient if the dimension

is too small.

6.2. *Performance of our Conjugate Gradient method*



(a) Performance and estimation using 1×1 blocks.

(b) Performance and estimation using 2×2 blocks.

(c) Performance and estimation using 4×4 blocks.

(d) Performance and estimation using 8×8 blocks.

Figure 10: Performance and estimations for our CG methods using one and two GPUs. The x-axis represents the indices of the used matrices, as found in Table 5. Table 6 shows the average relative-error and variance of the estimation compared with the actual results for each configuration.

Figure 10 shows the performances of our CG method using the SpMV approach from Section 3 (performed using different block sizes), and the corresponding performance estimations, as described in Section 5. Since the estimations are made for a single GTX280 GPU, we use the same GPU for comparing the results with the estimation. The results in Figure 10 are ordered with respect to the dimensions of the matrices. The test set (see Table 5 for some properties) represents a subset of the entire University of Florida sparse matrix collection [16], in which all matrices are Symmetric Positive Definite.

The best performances were obtained for matrices 'Bone010', 'af_shell3', 'nd24k', 'nd12k' and 'nd6k' for each block size. As shown in Table 5, these matrices have several millions of elements and have relatively high block densities. Also the maximum and average numbers of non-zeros per row are close to each other. This implies that each row has a similar amount of elements, and thus, a more balanced computation. Contrary, matrices 'F1' and 'inline_1', which have also a high number of elements, show a large deviation between the maximum and the average number of non-zeros per row. This implies that computations are unbalanced, which is reflected in the performances of these matrices.

In general, poor performances are obtained if (i) there is a large difference between the maximum and average number of non-zeros per row, (ii) the dimension of the problem is too small or (iii) the number of non-zeros is small. Increasing the block size increases the raw performance in general, but also the average block density *can* decrease, which *can* result in a lower actual performance.

Within Figure 10 the dashed lines represent the estimated performance for the corresponding matrices. We have analyzed the average relative error and the corresponding variance of the relative error, as shown in Table 6. From this

Id	Name	*Dim*	*nz*	*max*	*avg*	d_2	d_4	d_8
1	Trefethen_20	20	158	9	4.93	0.65	0.47	0.27
2	mesh1e1	48	306	8	6.37	0.36	0.20	0.17
3	Trefethen_150	150	2040	15	12.75	0.58	0.34	0.21
4	Trefethen_200	200	2890	16	13.89	0.57	0.34	0.21
5	bcsstk34	588	21418	47	36.17	0.76	0.52	0.34
6	msc00726	726	34518	88	46.89	0.55	0.38	0.26
7	msc01050	1050	29156	128	27.60	0.60	0.41	0.23
8	plbuckle	1282	30644	44	23.64	0.54	0.36	0.27
9	msc01440	1440	46270	45	32.13	0.62	0.42	0.24
10	nasa1824	1824	39208	42	21.49	0.52	0.35	0.23
11	Trefethen_2000	2000	41906	22	20.95	0.55	0.30	0.17
12	nasa2146	2146	72250	36	33.44	0.76	0.60	0.44
13	Chem97ZtZ	2541	7361	101	2.89	0.49	0.24	0.12
14	nasa2910	2910	174296	175	59.85	0.70	0.53	0.37
15	sts4098	4098	72356	784	17.59	0.43	0.22	0.13
16	nasa4704	4704	104756	42	22.26	0.53	0.33	0.22
17	crystm01	4875	105339	27	21.58	0.32	0.22	0.14
18	Kuu	7102	340200	98	47.88	0.99	0.71	0.45
19	Muu	7102	170134	49	23.94	0.5	0.35	0.22
20	bcsstk38	8032	355460	614	44.25	0.75	0.53	0.34
21	aft01	8205	125567	21	15.29	0.63	0.35	0.18
22	nd3k	9000	3279690	515	364.08	0.85	0.74	0.60
23	fv1	9604	85264	9	8.86	0.50	0.32	0.16
24	ted_B	10605	144579	49	13.62	0.55	0.40	0.30
25	ted_B_unscaled	10605	144579	49	13.62	0.55	0.40	0.30
26	msc10848	10848	1229778	723	113.36	0.80	0.56	0.34
27	cbuckle	13681	676515	600	49.39	0.94	0.77	0.56
28	crystm02	13965	322905	27	23.11	0.32	0.22	0.12
29	Pres_Poisson	14822	715804	50	48.26	0.73	0.47	0.31
30	Dubcova1	16129	253009	25	15.67	0.37	0.15	0.07
31	olafu	16146	1015156	89	62.81	0.89	0.69	0.50
32	bodyy4	17546	121938	9	6.94	0.49	0.24	0.12
33	nd6k	18000	6897316	514	383.18	0.84	0.73	0.59
34	bodyy5	18589	129281	9	6.95	0.49	0.24	0.12
35	bodyy6	19366	134748	9	6.95	0.49	0.24	0.12
36	Trefethen_20000b	19999	554435	29	27.72	0.53	0.29	0.15
37	Trefethen_20000	20000	554466	29	27.72	0.53	0.29	0.15
38	smt	25710	3753184	414	145.97	0.74	0.45	0.27
39	nd12k	36000	14220946	519	395.02	0.84	0.73	0.59
40	jnlbrng1	40000	199200	5	4.98	0.50	0.25	0.12
41	bcsstm39	46772	46772	1	0.99	0.50	0.25	0.124
42	gridgena	48962	512084	17	10.45	0.75	0.37	0.18
43	cvxbqp1	50000	349968	9	6.99	0.26	0.10	0.04
44	ct20stif	52329	2698463	207	51.56	0.82	0.57	0.36
45	nasasrb	54870	2677324	276	48.78	0.90	0.58	0.39
46	Dubcova2	65025	1030225	25	15.83	0.37	0.15	0.06
47	qa8fm	66127	1660579	27	25.11	0.51	0.29	0.16
48	cfd1	70656	1828364	33	25.87	0.46	0.24	0.13
49	nd24k	72000	28715634	520	398.82	0.84	0.73	0.58
50	finan512	74752	596992	55	7.98	0.40	0.14	0.06
51	apache1	80800	542184	7	6.71	0.39	0.19	0.09
52	thermal1	82654	574458	11	6.94	0.35	0.13	0.05
53	2cubes_sphere	101492	1647264	31	16.22	0.28	0.09	0.03
54	cfd2	123440	3087898	30	25.01	0.55	0.30	0.17
55	Dubcova3	146689	3636649	49	24.78	0.42	0.22	0.11
56	bmwcra_1	148770	10644002	351	71.53	0.75	0.49	0.28
57	G2_circuit	150102	726674	6	4.84	0.45	0.16	0.06
58	F1	343791	26837113	435	78.06	0.68	0.38	0.18
59	inline_1	503712	36816342	843	73.09	0.69	0.39	0.21
60	af_shell3	504855	17588875	40	34.83	0.84	0.65	0.46
61	parabolic_fem	525825	3674625	7	6.98	0.33	0.13	0.05
62	apache2	715176	4817870	8	6.73	0.39	0.19	0.09
63	tmt_sym	726713	5080961	9	6.99	0.50	0.25	0.12
64	bone010	986703	71666325	81	72.63	0.80	0.58	0.37
65	ecology2	999999	4995991	5	4.99	0.49	0.25	0.12
66	thermal2	1228045	8580313	11	6.98	0.34	0.12	0.04
67	G3_circuit	1585478	7660826	6	4.83	0.36	0.15	0.06

Table 5: Properties of the test matrices used for estimating the performance of our GPU mapping of the CG method. *Dim* is the dimension of the matrix, *nz* the total number of non-zeros, *max* the largest number of non-zeros in a row and *avg* is the average number of non-zeros per row; d_N is the average block density when the matrix is represented using $N \times N$ blocks. $d_1 = 1$ for all matrices.

analysis and Figure 10 we can conclude that the estimated performance comes closer to the measured performance, when the block size increases. According to Figure 7, this behavior is to be expected since the variation between the maximum and average performances for the SpMV operation is larger if $N < 4$. Since the estimated performances are closed to the measured ones, our estimation method from Section 5 can be used to compute a first indication of the expected maximum and average raw performances, on unseen problems.

Block size	#GPUs	Average rel. error	Variance
1×1	1	0.18	0.108
2×2	1	0.12	0.037
4×4	1	0.07	0.008
8×8	1	0.06	0.004
1×1	2	0.16	0.042
2×2	2	0.14	0.023
4×4	2	0.11	0.012
8×8	2	0.07	0.006

Table 6: Average relative-error, and the variance, of the estimated performance with respect to the measured performance.

6.3. Performance of our parallel CG method

Figure 10 also shows the performance results and the estimated performances for our CG method accelerated using two GTX280 GPUs. The solid lines show the measured performance, whereas the dashed lines show the estimated performance for the corresponding matrices. Table 6 shows the average relative error of the estimation.

The largest speedup and performance is obtained for matrix 'nd24k'. This matrix is relatively dense, i.e., about 400 elements per row on average, with a dimension of $72,000$ and a total of 28×10^6 non-zeros. According to Figure 8b a maximum raw performance of 42 GFlops/sec can be expected. Given the density $d_2 = 84\%$, the maximum expected performance is about 35 GFlops, which agrees with the results in Figure 10 for 2 GPUs. The measured speedup is about 1.7, which also agrees with the observations in Figure 8c. The dimension of the problem is relatively small, which means that the vector operations do not perform optimal, see Section 5. However, most of the computation time is spent on the SpMV operation, which is much larger than the time spent on the vector operations. This results in a relatively large speedup of about 1.7. Note that according to our analysis, reaching a speedup of 2 is practically impossible, see Section 5.

Figure 10 shows that in most test-cases the parallel performance is similar to the performance of the single GPU case. For these problems it is not worthwhile to use extra computational resources. However, in a small number of cases, the performance is significantly increased when two GPUs are used. Our model also reports similar performances and speedups in those cases. This means that our approach can be used to determine the number of GPUs that would solve a problem efficiently, given some properties of the corresponding matrix.

6.4. Performance comparison for the CG method

In this section we compare our GPU mapping of the CG method with a similar one using the CUSP library [3]. For a fair comparison, we have re-implemented Algorithm 1 (with the same preconditioner and an absolute tolerance of $1e-08$) using CUSP. We performed a large number of benchmarks on both a GTX280 and a GTX570 GPUs using single- and double-precision arithmetic. For each combination we measured the total time to solve a particular linear system. The set of matrices used for benchmarking is a subset of the University of Florida Sparse matrix collection [16], obtained as follows. We selected all matrices that were Symmetric Positive Definite and could fit in the system (and GPU) memory. Furthermore, we selected all matrices that converged using single-precision arithmetic. Finally, we obtained a set of 200 matrices originating from various problems. Table 7 shows a summary of our findings.

As can be seen from Table 7, our CG implementation performs significantly better than the CUSP-based one. For each possible combination of precision and hardware, our implementation performed in about 98 % of the cases better than the one using CUSP. Note that a comparable percentage was also found in Section 6.1, in which we compared our SpMV operation using a larger set of matrices.

Since this test set contains matrices coming from various problems and with very different sizes, the time required to solve a linear system varies a lot. Therefore, the average speedup is not completely reflected in the total speedup, since a number of large matrices dominate the total computation time (wall-clock time).

Method	GTX280						GTX570					
	Single-precision			Double-precision			Single-precision			Double-precision		
	Time	Speedup	Med.	Time	Speedup	Med.	Time	Speedup	Med.	Time	Speedup	Med.
CUSP	3108	1.00	1.00	2500	1.00	1.00	2031	1.00	1.00	1675	1.00	1.00
Ours	1285	2.41	5.73	1388	1.80	5.20	895	2.26	5.95	1058	1.58	5.69

Table 7: Performance comparison for the CG method using our implementation and CUSP [3]. 'Time' represents the total time in seconds needed to solve the complete set of linear systems, 'Speedup' represents the speedup of the total time relative to the CUSP-based implementation, 'Med.' denotes the median speedup of the individual test-cases relative to the CUSP-based implementation.

Switching from single to double precision yields similar average speedups, but the total computation time changes slightly. For our implementation, the total time increases on both GPUs, while the total time for CUSP decreases. Recalling the results presented in Figure 9, one can see that the performance difference between our method and CUSP/Hyb is smaller for double precision than with single precision. However, our double-precision version is not at all optimized.

Finally, we should point out that using double-precision arithmetic usually decreases the number of iterations needed for convergence, which in turn influences the total computation time. In section Section 7.4 we further discuss the differences between single- and double-precision performances.

7. Discussion

The analysis performed in Section 5 enables us to estimate the *maximum* and *average* performance of the CG method, accelerated using modern GPUs. One can conclude that these estimates are close to the measured performances (Section 6), provided that some conditions are met. First, the number of blocks per block row must not vary greatly. If the variation in row lengths is large, threads may become idle, so that the overall performance drops. Second, if the number of blocks per block row is very low, a larger error between the estimation and the real performance can be expected. If these conditions are met, the average or maximum (raw) performance can properly be estimated by considering the memory throughput. Furthermore, the larger the blocks, the smaller the variance (Figure 7, Table 6), and the better the estimation becomes.

When using two GPUs, a speedup can be expected for matrices with more than 2.5 million elements, see Figure 8. As reported by others [17] and also found by us, a good scalability of the CG method using GPUs can be achieved when the problem size is large enough to fully occupy the GPU. Moreover, the bandwidth between the devices plays an important role, see below.

7.1. Scalability for future devices

The current trend with GPU development is to increase the number of streamprocessors, raster output units and the amount of memory per GPU. Clearly, this increase will lead to a higher total performance. However, in order to reach the peak performance, the problem size should grow accordingly. For example, if the number of streamprocessors is doubled, the maximum performance is reached for problems that are twice as large. If multiple GPUs are used, the bandwidth between the GPUs becomes critical. If the available bandwidth *on* GPUs becomes larger, the bandwidth *between* the devices will represent even a larger bottleneck. This makes it even more difficult, especially for bandwidth-limited problems, to achieve a good scalability using multiple GPUs. Note that such changes can be accommodated by our analysis framework, by adjusting the value of parameter v for the inter-device communication time.

Current-generation GPUs support *Unified Virtual Addressing* [31], such that data stored on the GPUs can be accessed by other GPUs via the PCIe bus. This clearly improves the total memory throughput when, e.g., broadcasting the result vector to all GPUs. Furthermore, the communication time in Equation 11 will be reduced from xn to x, regardless the number n of GPUs connected over the PCIe bus.

7.2. Matrix reordering

The differences between the raw and actual performance are caused by the density of the matrix blocks. If matrix blocks are completely filled with non-zero elements, no computations are wasted. Therefore it is important to

maximize the average density of the matrix blocks. *Reordering* matrices using the (Reversed) Cuthill-McKee [15], Approximate Minimum Degree (AMD) [1] and King [22] matrix reordering schemes, does not usually improve the density of the matrix blocks significantly (results not shown). However, we expect that specific reordering methods, tailored for the BCSR layout, will lead to better performance figures.

7.3. Best block size

To answer the question about which block-size performs the best, we carefully studied the results presented in Section 6. First, we have found that for a large amount of matrices in the Harwell-Boeing set, the dimensions and the number of non-zeros are too small to fully utilize a GPU. The results showed that in 58% of the cases, blocks with $N = 8$ give the best results, even if the average density of the blocks is very low. For the multiple block-row mapping, increasing N automatically increases the amount of thread blocks, see Section 3. This in turn results in a higher utilization of the GPU. For the remaining cases we have found that 23% of the cases reported the best performance for $N = 1$, and in 11% of the cases the best performance was obtained for $N = 4$.

Inspecting the results obtained using the test set of Williams et al. [38] using a single GTX570 showed that in 50% of the cases, $N = 2$ yields the best performance. Contrary, the same test on a GTX280 gives in 35% of the cases the best performance when $N = 4$, while in 25% of the cases the best performance was reported when $N = 1$ and $N = 2$.

In order to find which configuration yields the best performance, we selected all test cases (from all benchmarks performed on a GTX570) in which the GPU was faster than the CPU. We found that in 42% of the cases $N = 1$ yields the best performance, followed by 25% of the cases when $N = 2$.

Since the dimension of the problem, the GPU mapping and the sparsity pattern of the matrices influence the performance, it is difficult in general to indicate which block size leads to the best results. However, one can compute the amount of needed thread blocks, given the dimensions of the problem and the block size. If this number is too small to fully utilize a GPU, increasing N and/or B_x will result in a mapping which has a higher utilization and therefore a better performance.

7.4. Double precision

In this paper we have used both single (32-bit wide) and double (64-bit wide) precision representations within our matrix and vector operations.

If the multiple block-row mapping is used in combination with block reordering and block-row sorting, all matrix blocks are loaded in a coalesced fashion, also if double precision is used, but in this case two memory transactions are needed. Loading the corresponding vector values also results in coalesced memory transfers if $N \geq 4$, similar to the single-precision case, but with twice the amount of transactions. When $N < 4$, memory transactions are not coalesced anymore and require more transactions for loading the corresponding vector values. In the case of double precision, this does not automatically lead to twice the amount of memory transaction, since the second part of the 64-bit value is stored in the same memory segment as the first 32 bits. This gives a small improvement in these cases. Furthermore, depending on the version of the architecture, *bank-conflicts* [26] can occur. For the GTX280 bank-conflicts happen if threads access 64-bit values in shared memory; for the GTX570 (Fermi) such conflicts do not happen.

We have computed the relative slow-down if one uses double precision, see Figure 9. The differences between the block-sizes is especially visible on the GTX280. When $N \geq 4$, the slow-down was approximately 2×, while for $N < 4$ it was between 1.25× and 1.75×. A similar slow-down was measured for the hybrid implementation of [4]. The figures for a GTX570 GPU were slightly different. For any N we have measured a slow-down between 1.4× and 1.6×, which was smaller than on the GTX280. According to the architectural differences between the GPUs, we assume that besides the absence of bank-conflicts, also cached accesses improve the performance when 64-bit data is fetched from the global memory.

Finally, using double precision for solving large linear systems can improve the total computation time, although the operations become roughly two times slower. On both GTX280 and GTX570 we have found that for the CG benchmark described in Section 6.4, about 50% of the cases were faster using double precision. Since the accuracy is higher, the CG method converges faster to the solution, especially for stiff problems. This finding is also reflected in Table 7, since the total computation time for double precision is not the twice the time for single precision.

7.5. Textures

To improve the memory throughput of random memory accesses on the GTX280 and older devices, textures are used frequently. The texture units provide a caching mechanism, which improves the throughput if the memory transactions are not coalesced and highly random.

Current generation GPUs [31, 26] now provide a cache mechanism for global memory. This might imply that the used of textures (for caching) now is deprecated. To test this possibility, the benchmark described in Section 6.1 was performed with and without texture cache on the newer GTX570 GPU. We found that the use of textures still improves the performance significantly on the newer hardware. When $N = 1$ and $N = 2$, the improvement is up to $25 - 50\%$, while for larger blocks it is minimal and sometimes slightly negative. Also the implementation of Bell and Garland [4] benefits from the use of textures. Therefore, it is still worthwhile to use textures if memory accesses are highly random.

8. Conclusions

In this paper we have investigated a number of mappings for block-based SpMV operations on GPUs, using CUDA. *Block row mapping* maps one complete block row (a row containing a number of $N \times N$ matrix blocks) to one thread block. This method is straightforward to implement, but not very efficient, since a lot of computational resources are wasted. Within this mapping one thread block processes a large number of matrix blocks. By transposing the block row mapping, the *multiple block-row mapping* is obtained. This mapping assigns multiple block rows to one thread block. One thread block processes a large number of matrix blocks, which belong to different block rows. This has positive implications on the performance, i.e., less thread blocks are needed and the amount of wasted computational resources is decreased. Furthermore, since each thread block processes a larger number of matrix blocks, better memory throughputs were obtained and thus a better performance. This is in general only the case if $N \geq 4$. If $N < 4$ the data must be reordered to obtain efficient (coalesced) memory transactions for loading the matrix blocks. This *block reordering* significantly improves the performance of the SpMV operation for matrices stored using the BCSR layout with blocks of size $N < 4$, if the MBR mapping is used. Sorting the block rows such that block rows with similar lengths are processed by the same thread block, increases the performance significantly.

By mapping the computations differently on the GPU, and by applying row sorting and block reordering, the best performances for the SpMV operation were obtained. Experimental results showed that our SpMV mapping outperforms existing methods in most cases, and performs close to the limits of the hardware. Our optimized SpMV operation was used to accelerate the CG method, given one or multiple GPUs. Together with the optimized vector operations, this makes (in most cases) our CG mapping about 5 times faster than existing methods.

We have also provided a recipe for estimating the maximum achievable performance and the average performance of a (parallel) CG method, given the properties of the problem. This method can be applied to similar numerical algorithms. Analyzing the memory throughput revealed a clear trend between the number of memory transactions and the performance. This analysis has been done for each kernel performing vector operations, as well as for the SpMV kernel. The resulting trends were modeled by a particular sigmoid function, which was then used to estimate the memory throughput of each individual operation appearing in the CG method. This finally leaded to an approximation of the maximum or average performance of the method. We further extended our performance-estimation framework such that also multiple GPU setups can be modeled. The results showed that our performance estimates were very close to the measured performance, and in general, the estimates became more accurate when larger blocks are used.

In future work, we plan to investigate methods for matrix reordering, suitable for the BCSR format. Existing matrix reordering methods optimize, e.g., the bandwidth of the matrix, which does not necessarily result in increased block densities. Further, our analysis may be improved by taking into account the variations in GPU load, due to block-row padding by empty blocks. Finally, our method performs very well on matrices for which the variation in row lenghts is not too large. To let our implementation also perform well on matrices with highly unbalanced rows, we are planning to extend our method.

References

[1] P. R. Amestoy, Enseeiht-Irit, T. A. Davis, and I. S. Duff. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.*, 30(3), 2004. 25

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994. 1, 2, 3

[3] N. Bell and M. Garland. *CUSP library*. Available at http://code.google.com/p/cusp-library. 18, 19, 20, 23, 24

[4] N. Bell and M. Garland. Efficient Sparse Matrix-Vector Multiplication on CUDA. Technical Report NVR-2008-004, NVidia, 2008. 1, 2, 3, 9, 18, 19, 20, 25, 26

[5] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. In *Proc. SIGGRAPH'03*, 2003. 2

[6] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. State-of-the-art in heterogeneous computing. *Sci. Program.*, 18:1–33, January 2010. 1

[7] Andr R. Brodtkorb, Martin L. Stra, and Mustafa Altinakar. Efficient shallow water simulations on gpus: Implementation, visualization, verification, and validation. *Computers and Fluids*, 55(0):1 – 12, 2012. 1

[8] L. Buatois, G Caumon, and B. Lévy. Concurrent Number Cruncher: An Efficient Sparse Linear Solver on the GPU. In *High Perf. Comp. Conf. (HPCC)*, 2007. 1, 2, 18, 19, 20

[9] R. L. Burden and J. D. Faires. *Numerical analysis*. PWS Publishing Co., Boston, MA, USA, 8th edition, 2005. 3

[10] A. Cevahir, A. Nukada, and S. Matsuoka. High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning. *Computer Science - Research and Development*, 25, 2010. 3

[11] A. Cevahir, A.a Nukada, and S. Matsuoka. Fast Conjugate Gradients with Multiple GPUs. In *Computational Science ICCS 2009*, 2009. 3

[12] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. *SIGPLAN Not.*, 45, 2010. 2, 3

[13] NVidia Corporation. *CUDA Occupancy Calculator*. Available at http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls. 5

[14] A. Corrigan, F.F. Camelli, R. L"ohner, and J. Wallin. Running unstructured grid-based cfd solvers on modern graphics hardware. *International Journal for Numerical Methods in Fluids*, 66(2):221229, 2011. 1

[15] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *ACM '69: Proc. of the 1969 24th nat. conf.*, 1969. 25

[16] T. A. Davis. University of Florida sparse matrix collection., 1997. http://www.cise.ufl.edu/research/sparse. 13, 14, 19, 21, 23

[17] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S.H.M. Buijssen, M. Grajewski, and S. Turek. Exploring weak scalability for FEM calculations on a GPU-enhanced cluster. *Parallel Computing, Special issue: High-performance computing using accelerators*, 33(10–11), 2007. 2, 24

[18] Gene H. Golub and Charles F. Van Loan. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996. 3

[19] Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, and Nectarios Koziris. Understanding the performance of sparse matrix-vector multiplication. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 283–292, Washington, DC, USA, 2008. IEEE Computer Society. 4

[20] M. Harris, S. Sengupta, and J. D. Owens. Parallel Prefix Sum (Scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*. Addison Wesley, August 2007. 10, 14

[21] M.R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 49(6), 1952. 1, 3

[22] I.P. King. An automatic reordering scheme for simultaneous equations derived from network problems. *Int. J. Numer. Meth. in Eng.*, 2, 1970. 25

[23] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, 2008. 4

[24] S.A. Manavski. Cuda compatible gpu as an efficient hardware accelerator for aes cryptography. In *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*, pages 65 –68, nov. 2007. 1

[25] A. Monakov and A. Avetisyan. Implementing Blocked Sparse Matrix-Vector Multiplication on NVIDIA GPUs. In *SAMOS '09: Proc. of the 9th Int. Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2009. 2

[26] NVIDIA Corporation. *Compute Unified Device Architecture programming guide*. Available at http://developer.nvidia.com/cuda. 4, 5, 16, 20, 25, 26

[27] NVIDIA Corporation. *CUBLAS library*. Available at http://developer.download.nvidia.com/compute/cuda/2_0. 20

[28] NVIDIA Corporation. *cuSPARSE library*. Available at http://developer.download.nvidia.com/compute/cuda/2_0. 18, 19

[29] NVIDIA Corporation. *NVidia GeForce 8800 GPU Architecture Overview*. Available at http://www.nvidia.com/page/8800_tech_briefs.html. 4

[30] NVIDIA Corporation. *NVidia GPUDirect*. Available at http://developer.nvidia.com/gpudirect. 16

[31] NVIDIA Corporation. *NVidia's Next Generation CUDA Compute Architecture Fermi*. Available at http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. 4, 5, 16, 24, 26

[32] NVIDIA Corporation. *NVidia nForce 790i SLI Chipsets, Reducing Latencies & Bandwidth Utilizations*, 2008. 16

[33] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 2nd edition, 1992. 3, 12

[34] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003. 1, 2

[35] A. van der Sluis and H. A. van der Vorst. The rate of convergence of conjugate gradients. *Numerische Mathematik*, 48:543–560, 1986. 10.1007/BF01389450. 1

[36] H. A. van der Vorst. Bi-cgstab: a fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13:631–644, March 1992. 1, 2

[37] W.A. Wiggers, V. Bakker, A.B.J. Kokkeler, and G.J.M. Smit. Implementing the conjugate gradient algorithm on multi-core systems. In *Proc. of the Int. Symp. on System-on-Chip (SoC 2007)*, November 2007. 2

[38] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC '07: Proc. of the 2007 ACM/IEEE conf. on Supercomputing*, 2007. 2, 19, 25