A Fast Sparse Block Circulant Matrix Vector Product

Eloy Romero[1], Andrés Tomás[1], Antonio Soriano[1], and Ignacio Blanquer[1]

Instituto de Instrumentación para Imagen Molecular (I3M),
Centro Mixto CSIC – Universitat Politècnica de València – CIEMAT
Camino de Vera s/n, 46022 Valencia (Spain)
{elroal,antodo,asoriano}@i3m.upv.es, iblanque@dsic.upv.es

Abstract. In the context of computed tomography (CT), iterative image reconstruction techniques are gaining attention because high-quality images are becoming computationally feasible. They involve the solution of large systems of equations, whose cost is dominated by the sparse matrix vector product (SpMV). Our work considers the case of the sparse matrices being block circulant, which arises when taking advantage of the rotational symmetry in the tomographic system. Besides the straightforward storage saving, we exploit the circulant structure to rewrite the poor-performance SpMVs into a high-performance product between sparse and dense matrices. This paper describes the implementations developed for multi-core CPUs and GPUs, and presents experimental results with typical CT matrices. The presented approach is up to ten times faster than without exploiting the circulant structure.

Keywords: circulant matrix, sparse matrix, matrix vector product, GPU, multi-core, computed tomography

1 Introduction

Iterative approaches to image reconstruction have cut down the radiation dose delivered to the patient in computed tomography (CT) explorations [5], because they are less sensitive to noisy acquisitions than filtered backprojection (FBP). Iterative methods consider the reconstruction problem as a system of linear equations $\boldsymbol{y} = A\boldsymbol{x}$. The probability matrix A constitutes a mathematical model of the tomographic system that links the reconstructed attenuation map $\boldsymbol{x}$ with the estimation of the measurement $\boldsymbol{y}$. The large number of projections and the high spatial resolution (≈ 0.1 mm) in CT require the solution of huge systems. This is the reason why CT image reconstruction has been dominated by analytic methods like FBP [3]. However the availability of cheaper and more powerful hardware has favoured a novel interest in the use of iterative methods in CT image reconstruction [1, 18, 15].

Like in many other engineering and scientific applications, A is usually a large, sparse matrix, *i.e.*, with relatively few non-zeros. Operating with sparse matrices is computationally efficient: the storage requirements and the time of a product by a vector are almost linear to the number of non-zeros, instead of quadratic to the matrix dimension for dense matrices. In spite of this good asymptotic behavior, the performance shown by sparse matrices is far from exhausting the computing throughput of modern processors, mainly because of low count of operations per memory transaction. As an example, results testing the matrix-vector (SpMV) product on several multi-core processors in [23] show disparate peak performance between 3% and 40%. This has aimed the development of optimizations techniques and enhanced formats specialized for matrices with dense blocks [7, 22], dense triangles, diagonals, symmetry [12, 11] and general patterns [10].

In the same way, this work addresses block circulant matrices, block matrices where each row of blocks is rotated one element to the right relative to the previous row. Examples of these matrices can be found in applications involving discretization aware of cylindrical or cyclical symmetries in the domain [4, 9, 13, 20]. Particularly in the context of CT scanners, mathematical descriptions based on polar coordinates take advantage of the rotational symmetries in the tomographic system, because it is easy to find an ordering of the unknowns so that projections share the same pattern of weights in the probability matrix, although shifted by a fixed number of columns [19, 16]. The probability matrix A constructed like this corresponds to a block circulant matrix, in which the rows associated to a projection form a row of blocks with the displacement being the number of columns in every block. Implicit representations of A can save storage and speed up its construction by a factor of the number of projections, which is around 100 in practice. Nevertheless, the cost of the associated SpMV product remains the same, in terms of the number of floating-point operations.

In general the SpMV products dominate the time spent on the solution of the system of linear equations by iterative methods such as the maximum likelihood expectation maximization (MLEM) algorithm [17], one of the most used in CT. In this paper, we propose to accelerate these products by rewriting them as sparse matrix-dense matrix (SpMM) products. The results we obtained show a reduction of time by a factor up to ten.

The remainder of this paper is organized as follows. In section 2 it is explained the approach based on the SpMM product and possible implementations. Sections 3 and 4 detail several implementations for multi-core CPUs and GPUs, respectively, and show performance results. And finally section 5 concludes.

Notation. We denote matrices with uppercase letters (A, B...) and vectors with bold lowercase letters ($\boldsymbol{x}$, $\boldsymbol{y}$...). Indices in vectors and matrices start by zero. $X_{i,j}$ or $\mathsf{X}[i,j]$ refer to the element on row i and column j of the matrix X. We refer to the BLAS-1 functions, AXPY as $\boldsymbol{y} \leftarrow \boldsymbol{y} + \alpha \cdot \boldsymbol{x}$, and MAXPY with cardinality k as $\boldsymbol{y} \leftarrow \boldsymbol{y} + \alpha_0 \cdot \boldsymbol{x}_0 + \alpha_1 \cdot \boldsymbol{x}_1 + \cdots + \alpha_{k-1} \cdot \boldsymbol{x}_{k-1}$.

2 Circulant Matrix Product Approach

Let C being a block circulant matrix of size $m_C \times n_C$, made by $k \times k$ matrix blocks A_i of size $m_B \times n_B$. Then $m_C = k \cdot m_B$ and $n_C = k \cdot n_B$. The matrix-vector product $\boldsymbol{y} \leftarrow C\boldsymbol{x}$ takes the form

$$\begin{pmatrix} \boldsymbol{y}_0 \\ \boldsymbol{y}_1 \\ \vdots \\ \boldsymbol{y}_{k-2} \\ \boldsymbol{y}_{k-1} \end{pmatrix} \leftarrow \begin{pmatrix} A_0 & A_1 & \cdots & A_{k-2} & A_{k-1} \\ A_{k-1} & A_0 & \cdots & A_{k-3} & A_{k-2} \\ \vdots & \vdots & & \vdots & \vdots \\ A_2 & A_3 & \cdots & A_0 & A_1 \\ A_1 & A_2 & \cdots & A_{k-1} & A_0 \end{pmatrix} \begin{pmatrix} \boldsymbol{x}_0 \\ \boldsymbol{x}_1 \\ \vdots \\ \boldsymbol{x}_{k-2} \\ \boldsymbol{x}_{k-1} \end{pmatrix}, \tag{1}$$

where $\boldsymbol{x}_i$ are subvectors of length n_B and $\boldsymbol{y}_i$ are of length m_B. The block circulant matrix C is fully specified by the first block of rows, which we named $A = (A_0\ A_1\ \ldots\ A_{k-1})$. Basic implementations avoid storing explicitly the matrix C, for instance by rewriting the whole product as products by the blocks A_i,

$$\boldsymbol{y}_i \leftarrow \sum_{j=0}^{k-1} A_{(j-i) \bmod k}\, \boldsymbol{x}_j, \qquad \text{for } i \text{ from } 0 \text{ to } k-1. \tag{2}$$

This approach employs the SpMV product, which in practice has a performance mostly limited by the memory bandwidth. Instead, we propose to rewrite the block circulant matrix-vector product into a matrix-matrix product, which offers better performance even in simple implementations, as we show further. Then the product in (1) can be reformulated as $Y \leftarrow A\mathring{X}$, which is in detail

$$\begin{pmatrix} \boldsymbol{y}_0\ \boldsymbol{y}_1\ \ldots\ \boldsymbol{y}_{k-1} \end{pmatrix} \leftarrow \begin{pmatrix} A_0\ A_1\ \ldots\ A_{k-1} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}_0 & \boldsymbol{x}_1 & \cdots & \boldsymbol{x}_{k-2} & \boldsymbol{x}_{k-1} \\ \boldsymbol{x}_1 & \boldsymbol{x}_2 & \cdots & \boldsymbol{x}_{k-1} & \boldsymbol{x}_0 \\ \vdots & \vdots & & \vdots & \vdots \\ \boldsymbol{x}_{k-2} & \boldsymbol{x}_{k-1} & \cdots & \boldsymbol{x}_{k-4} & \boldsymbol{x}_{k-3} \\ \boldsymbol{x}_{k-1} & \boldsymbol{x}_0 & \cdots & \boldsymbol{x}_{k-3} & \boldsymbol{x}_{k-2} \end{pmatrix}. \tag{3}$$

Therefore, the circulant property has transfered from C to the vector $\boldsymbol{x}$, converting the latter in a kind of *anti-circulant* block matrix $\mathring{X}$, where the rows of blocks rotate to the left instead. The matrix $\mathring{X}$ is of size $n_C \times k$ and, as earlier, it is fully specified by the first row of blocks, which we named $X = (\boldsymbol{x}_0\ \boldsymbol{x}_1\ ...\ \boldsymbol{x}_{k-1})$. In the process, also the output vector $\boldsymbol{y}$ is transformed into the matrix form $Y = (\boldsymbol{y}_0\ \boldsymbol{y}_1\ ...\ \boldsymbol{y}_{k-1})$.

An efficient (at least, in memory) implementation of the product needs to maintain $\mathring{X}$ implicit. In a SpMM product code, a way to do so is by replacing the $\mathring{X}$ accesses by accesses to X like this,

$$\mathring{X}_{i,j} = X_{i',j'}, \text{ where } \quad i' = i \bmod n_B \quad \text{and} \quad j' = (\lfloor i/n_B \rfloor + j) \bmod k. \tag{4}$$

This solution can be useful if either the SpMM product routine allows to reimplement the behaviour of the operators (for instance, because matrices are implemented as classes in an object oriented language like C++), or the source code is available.

Nevertheless, we propose an alternative when it is not possible, for instance in the case of using a commercial numerical library. If the routine requires the dense

```
Data: $A : \mathbb{R}^{m \times n}$ sparse matrix; $\mathsf{X} : \mathbb{R}^{n \times k}$, dense matrix
Result: $\mathsf{Y} : \mathbb{R}^{m \times k}$, dense matrix with the product $AX$
$Y \leftarrow 0$
for $i \leftarrow 0$ to $m - 1$ do (in parallel)
    foreach nonzero with column index $j$ and value $v$ in row $i$ of $A$ do
        for $p \leftarrow 0$ to $k - 1$ do // Done as an AXPY
            $\mathsf{Y}[i,p] \leftarrow \mathsf{Y}[i,p] + v \cdot \mathsf{X}[j,p]$
```

Fig. 1. Generic sparse-dense matrix product, $Y \leftarrow A\,X$

matrix to be stored in column-major (*i.e.*, elements in consecutive rows and in the same column are contiguous in memory), then the next relation between $\mathring{X}$ and a vector $\hat{\boldsymbol{x}}$ that contains two contiguous copies of $\boldsymbol{x}$ can be useful,

$$\mathring{X}_{i,j} = j'\text{-th element in } \hat{\boldsymbol{x}}, \qquad \text{where} \quad j' = i + j \cdot n_B \quad \text{and} \quad \hat{\boldsymbol{x}} = \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{x} \end{pmatrix}. \tag{5}$$

Then the routine is passed the sparse matrix A, and $\hat{\boldsymbol{x}}$ as the dense matrix, indicating the leading dimension n_B (the number of elements in between two elements with consecutive indices in the dimension that is not contiguous in memory).

Otherwise, if the dense matrix has to be stored in row-major (*i.e.*, elements in the same row are contiguous in memory) instead, column indices of the non-zeros in the sparse matrix has to be updated in the next way,

$$A_{i,j} = \hat{A}_{i,j'}, \text{ where } j' = 2 \cdot k \cdot (j \bmod n_B) + (\lfloor j/n_B \rfloor) \bmod k. \tag{6}$$

Then the routine is passed the modified sparse matrix $\hat{A}$ (with size $m_B \times 2 \cdot n_C$) and the dense matrix $\hat{X} = (X\ X)$ in row-major, indicating the leading dimension 1.

3 Multi-Core CPU Implementation

Earlier we discussed how to perform the block circulant SpMV product by using a SpMM product. Although the SpMM product is algorithmically simple, it admits several implementations. One of them consists on multiple calls to the SpMV product kernel, but in the case of implementing (3) it is equivalent to do the product in the original way of (1).

From the rest of implementations, we conveniently choose the one that for every nonzero value in a sparse matrix A with row i and column j, an AXPY operation is done involving the j-th row of the input matrix X and the i-th row of the output matrix Y. The algorithm is illustrated in Fig. 1.

The product is not computationally heavy, then the performance is conditioned to the capability of the machine's cache system to take advantage of the reference locality of the implementation. Considering the spatial locality (*i.e.*, the use of data elements within relatively close address locations), the memory

Table 1. Description of tested matrices.

Matrix	Rows	Columns	Blocks	Nnz	A Nnz/row	A^t Nnz/row
CT small	19,600	1,284,000	150	7,029,618	358.7	821.2
CT medium	19,600	5,583,600	150	18,845,735	961.5	506.3
CT big	19,600	15,767,700	150	40,601,519	2,071.5	386.2
CT large	78,400	29,764,800	150	120,506,745	1,537.1	607.3
CT huge	78,400	116,660,700	150	304,228,353	3,880.5	391.2

Table 2. Description of the test machines.

CPU Name	Freq.	PUs	Cores	L1	L2	L3	Mem.	Bandwidth
Intel Xeon X3450	2.7 GHz	8	4	32 KiB	256 KiB	8 MiB	4 GiB	21 GB/s
Intel i7 3930K	3.2 GHz	12	6	32 KiB	256 KiB	12 MiB	32 GiB	51 GB/s
NVIDIA GTX680	1.1 GHz	8	192	48 KiB	512 KiB	–	4 GiB	192 GB/s

access pattern is efficient if the vectors involved in the AXPYs are contiguous. This is the case of the input X and output Y dense matrices stored in row-major. Considering the temporal locality (*i.e.*, the use of the same data elements within a relatively small time duration), per nonzero value in A read it is done k read accesses of X, and k read and write accesses of Y. Along a row in the sparse matrix A, all the accesses go to the same row of Y. Then it seems an optimal strategy to visit the non-zeros on the sparse matrix by rows. A similar conclusion is found on [7].

In addition, the straightforward parallelization is that every task carries on the AXPYs of several rows of the sparse matrix A, which corresponds to distribute the iterations of the loop at line 2 in Fig. 1. This strategy prevents two tasks accessing the same row of Y. The distribution of work will be balanced if every task performs almost the same number of AXPYs, *i.e.*, every task processes almost the same number of non-zeros from A.

3.1 Custom Product for Circulant Block Sparse Matrices

We developed several kernels that implement the product for sparse circulant matrices in CSR format. They are written in C++ and parallelized using threads by means of OpenMP directives. We present performance results of the kernels compiled with GNU GCC 4.8 and the options `-Ofast -march=native -mtune=native`, running on two Intel multi-core processors detailed on Table 2. The test set comprises five matrices from a CT scanner, whose characteristics are detailed on Table 1. They come from reconstructions with different spatial resolutions. Their patterns are quite similar and, as an example, Fig. 2.a shows the pattern of the first rows for some blocks of the smaller matrix on the set.

Figure 3 summarizes the performance of the kernels grouped by processor and matrix. In order to emulate the behavior of an iterative solver (like MLEM), it is performed 20 products alternating the direct and the transposed matrix. The results correspond to the shorter time of three attempts.

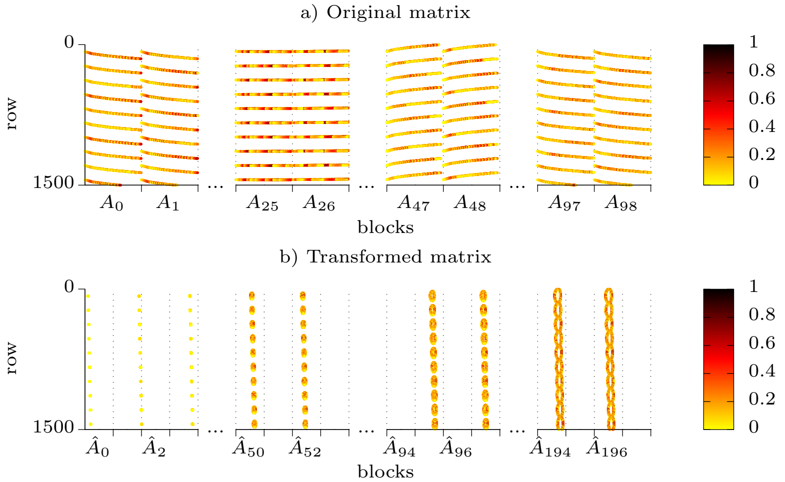

Fig. 2. Fragments of nonzero pattern in a) the matrix CT small and b) after applying the transformation of (6).

The first kernel, tagged *basic MV*, uses the matrix-vector product approach and it is parallelized by every thread computing a $\boldsymbol{y}_i$ as (2) indicates.

The second kernel, tagged *MM*, uses the matrix-matrix product approach (indicated in (3)) and the implicit circulant matrix formulas for $\mathring{X}$ (indicated in (4)) over the input vector, stored as a row-major matrix. It corresponds to the algorithm in Fig. 5 without the code under the *while* loop at line 4. The *for* loop at line 9 is implemented as two calls to an AXPY kernel: one from $p \leftarrow 0$ to $p_0 - 1$ and other from $p \leftarrow p_0$ to $k - 1$, where p_0 is $k - \lfloor \mathsf{Aj}[j]/n_B \rfloor \bmod k$. Otherwise GNU GCC fails to vectorize the loop. The results suggest a gain of this kernel up to four times with respect to *basic MV*.

If the kernel is passed the matrix $\hat{X} = (X\ X)$ instead of X, then the *for* loop at line 9 can be implemented as a single call to an AXPY kernel with k-length vectors. This variant, tagged *MM-2*, obtains an extra performance of up to 70% respect to *MM*.

Finally, some memory transactions from the output matrix Y can be saved by merging s AXPYs originated from the same row, in a single MAXPY operation, as shown on Fig. 5. Although it can make worse the temporal locality of X (because s different parts of X are being visited at a time), simple tests shown in Fig. 4 suggest that in general the performance is improved with larger s, while the compiler is able to vectorize the innermost loop of MAXPY, which corresponds to *for* at line 5 in Fig. 5. Codes using MAXPYs with $s > 8$ vectors fail to be vectorized by the tested compiler. The performance of this optimization for $s = 8$, tagged as *MM-2-8*, is about 25% better than *MM-2*, and about 10 times better than the *basic MV* kernel.

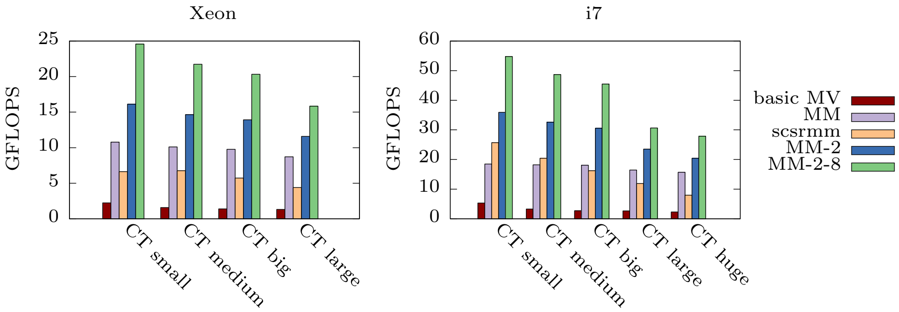

Fig. 3. Performance on CPU of several kernels computing the SpMV product with circulant block sparse matrices of different sizes. (CT huge matrix cannot be tested on Xeon machine due to memory limitations.)

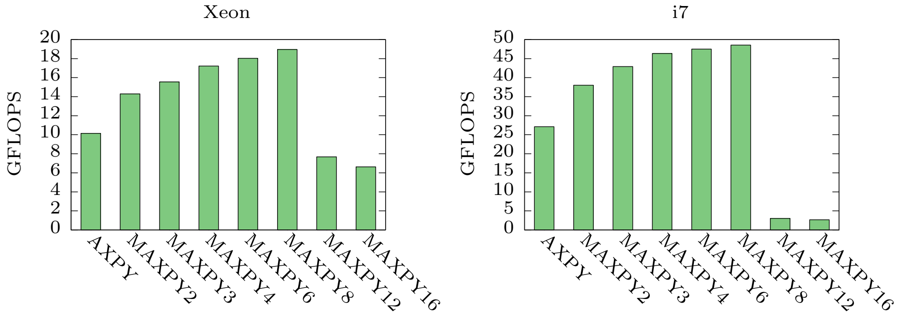

Fig. 4. Performance on both processors of AXPY and MAXPY (with different cardinalities) adding 50 vectors with 100 elements. GCC reports that it failed to vectorize the loops in the routines MAXPY12 and MAXPY16.

3.2 Using a SpMM kernel in a numerical library software

We found two open source implementations of the SpMM product. One is in the Scipy library (the core library of SciPy [8]), in which the input and output matrices are stored in row-major. And other is in Epetra (a core linear algebra package in Trilinos [6]), which merges multiple AXPYs in MAXPYs, but the dense matrices are stored in column-major. We do not show results with any of these libraries because none of them include both optimizations at the same time. Other popular libraries, like OSKI [21] or CUSP [2], offer an interface to perform the SpMM product but the implementation relies on multiple calls to the SpMV product kernel.

Regarding commercial high-performance numerical libraries, only Intel® Math Kernel Library (Intel® MKL) offers several routines for SpMM product. Concretely we tested the function `*csrmm`, in which the sparse matrix is introduced in CSR format and the input and the output dense matrices are stored in row-major. For that, the original sparse matrix A has to be modified as indicated in (6) (the new nonzero pattern is shown in Fig. 2.b), and the input dense matrix

```
Data: $\mathsf{Ai} : \mathbb{N}^{m_B+1}$, $\mathsf{Aj} : \mathbb{N}^{nnz}$, $\mathsf{Av} : \mathbb{R}^{nnz}$, CSR vectors of a $m_B \times (n_B \cdot k)$, $nnz$
      non-zeros sparse matrix representing the first block row of a $k \times k$ block
      circulant matrix; $\mathsf{X} : \mathbb{R}^{n_B \times k}$, dense matrix.
Result: $\mathsf{Y} : \mathbb{R}^{m_B \times k}$ dense matrix with the product of the sparse matrix by $\mathsf{X}$.
$\mathsf{Y} \leftarrow 0$
for $i \leftarrow 0$ to $m_B - 1$ do (in parallel)
    $j \leftarrow \mathsf{Ai}[i]$
    while $j + s \leq \mathsf{Ai}[i+1]$ do // Optional: take $s$ vector at a time
        for $p \leftarrow 0$ to $k - 1$ do // MAXPY
            $\mathsf{Y}[i,p] \leftarrow \mathsf{Y}[i,p] + \sum_{l=j}^{j+s-1} \mathsf{Av}[l] \cdot \mathsf{X}[\mathsf{Aj}[l] \bmod n_B, (\lfloor \mathsf{Aj}[l]/n_B \rfloor + p) \bmod k]$
        $j \leftarrow j + s$
    while $j \leq \mathsf{Ai}[i+1]$ do // Take the last vectors
        for $p \leftarrow 0$ to $k - 1$ do
            $\mathsf{Y}[i,p] \leftarrow \mathsf{Y}[i,p] + \mathsf{Av}[j] \cdot \mathsf{X}[\mathsf{Aj}[j] \bmod n_B, (\lfloor \mathsf{Aj}[j]/n_B \rfloor + p) \bmod k]$
```

Fig. 5. Product of block circulant sparse matrix by dense matrix, taking s by s vectors at a time (CPU).

introduced is the replicated $\hat{X}$ in row-major. As Fig. 3 shows, its performance is superior to the *basic MV* and *MM* kernels in few cases, and *MM-2-8* can double the performance of the MKL kernel.

4 Circulant Sparse Product Implementation on GPU

Current GPU accelerators provide a cost-effective platform for CT applications. These applications require single precision arithmetic only, allowing to use low cost graphics hardware. Furthermore, the sparse matrix vector product performance is limited by memory bandwidth and GPU accelerators provide much higher bandwidth than CPU main memory.

In this paper a NVIDIA Geforce GTX 680 is selected as the hardware platform and CUDA as the software counterpart. The CUDA software package includes an implementation of the SpMM product in the CUSPARSE library [14]. However, this routine checks the input parameters and, unlike the MKL library, forbids to set the leading dimension of the dense matrix to a value smaller than its number of rows. Therefore, an implementation based on the CUSPARSE SpMM routine cannot be tested, and we present only results performing several calls to the SpMV routine from the same library.

We developed a custom CUDA kernel based on the SpMM approach from (3), following many of the considerations detailed earlier on the CPU. In particular, the data layout of the dense matrices X and Y is also row-major. However, these matrices are stored without any redundancy, this is important as GPU memory is not as large as CPU memory.

The work distribution among GPU processors is completely different from the CPU implementation. The key to obtain high performance in a GPU is to

```
Data: $\mathsf{Ai} : \mathbb{N}^{m+1}$, $\mathsf{Aj} : \mathbb{N}^{nnz}$, $\mathsf{Av} : \mathbb{R}^{nnz}$, CSR vectors of a $m \times n$, $nnz$ nonzeros
      sparse matrix representing the first block row of a $k \times k$ block circulant
      matrix; $\mathsf{X} : \mathbb{R}^{n_B \times k}$, dense matrix.
Result: $\mathsf{Y} : \mathbb{R}^{m \times k}$ dense matrix with the product of the sparse matrix by $\mathsf{X}$.
for $i \leftarrow 0$ to $m - 1$ do (in parallel) // block parallelism
    for $p \leftarrow 0$ to $k - 1$ do (in parallel) // thread parallelism
        $w \leftarrow 0$
        for $j \leftarrow \mathsf{Ai}[i]$ to $\mathsf{Ai}[i + 1]$ do
            $c \leftarrow \mathsf{Aj}[j]/n_B + p$
            if $c > n_B$ then
                $c = c - n_B$
            $l \leftarrow \mathsf{Aj}[j] \bmod n_B$
            $w \leftarrow w + \mathsf{Av}[j] \cdot \mathsf{X}[l, c]$;
        $\mathsf{Y}[i, p] \leftarrow w$;
```

Fig. 6. Product of block circulant sparse matrix by dense matrix (GPU).

keep all of the computing elements busy with a minimum of communications among them. Therefore, a straightforward work distribution is to assign to each thread the computation of just one element from the product result vector Y.

Figure 6 contains the GPU implementation pseudocode for the circulant sparse matrix product. The CUDA runtime environment provides blocks of threads, that is, two levels of parallelism. The first level (blocks) is presented in Fig. 6 as the first loop, while the second level (threads) corresponds to the second loop. The actual implementation does not contain these two loops, they are implicitly created by the kernel invocation parameters.

First, to obtain good performance in the GPU a large number of thread blocks must be created, in this case one block per matrix row. The actual amount of work per block is determined by the matrix pattern and could be quite different from row to row. However, this is not an issue because there are enough blocks to keep busy all GPU processors. If the execution time of a block is too short, the hardware scheduler could easily select another block from the execution queue.

Second, the number of threads inside a block should be a small multiple of 32 (warp size) to obtain good performance on the GPU. In this implementation there are as many threads as blocks has the circulant matrix. Although this number might not be a multiple of 32 (150 in our tests), it is sufficiently large to occupy several warps.

Last but not least, GPU performance is very dependent on memory access patterns. In this case, all data is stored in device memory with a similar distribution as presented before for the CPU implementation. The vector X is read in coalesced form via a texture cache to further increase the effective bandwidth. Each element of vectors Ai, Aj and Av is read simultaneously by all threads in a block. This memory access is quite efficient and saturate most of the device memory bandwidth. Our tests show no tangible benefits in using shared or constant memory for this access pattern.

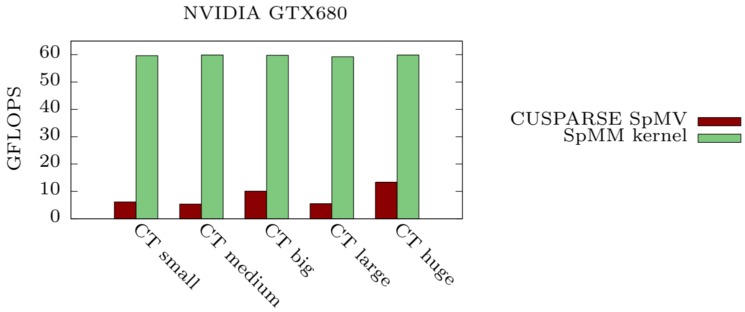

Fig. 7. Performance on GPU of kernels based on the SpMV and SpMM approaches with different matrix sizes.

One small optimization different to the CPU implementation is that Aj (column index) is stored in two separate vectors, one with values $\lfloor \mathsf{Aj}[j]/n_B \rfloor$ and another with $\mathsf{Aj}[j] \bmod n_B$. Both operations have a low throughput on the GPU, and precomputing them on the CPU improves performance of the whole circulant sparse product.

Figure 7 compares the performance of two circulant sparse matrix product implementations on the GTX680 GPU. The first is based on CUSPARSE SpMV routines while the second is our custom SpMM kernel implementation, which is several times faster than the SpMV implementation. The performance of SpMM is almost constant (about 60 GFLOPS) for all matrix sizes, and up to two times faster than our optimized SpMM on the CPU. The main advantage of the GPU over the CPU is that performance does not decrease with large matrices.

5 Conclusions and Future Work

In this paper, we have described optimization techniques to improve the performance of the sparse matrix vector product (SpMV) for block circulant matrices. This matrix structure allows to rewrite the SpMV into a product of two matrices, one sparse and other dense (SpMM). Moreover, we propose to replicate vector data and to join several AXPY products into a MAXPY. Both optimizations simplify data access and improve cache locality.

Our optimized SpMM kernel reduces execution time by 10 times on an Intel i7 CPU compared to a trivial SpMV implementation. We also tested alternative implementations using the SpMV and SpMM products from high-performance libraries (Intel MKL), but all of them obtain worse performance than our kernels.

On GPU, we propose a similar distribution of data which allows to fully exploit the device raw bandwidth via coalesced and textured memory accesses. This SpMM implementation is about 6 times faster than the SpMV from the CUSPARSE library on a NVIDIA GTX680 graphics card. For very large matrices, this GPU halves execution time with respect to our own optimized kernel running on the Intel i7 CPU.

Furthermore the described optimizations are compatible with other enhancements, especially on CPU, such as exploiting patterns in the blocks of the sparse matrix and implementing explicit prefetch to improve performance with large matrices. Beside them, as a future work we intend to develop a competitive kernel for the transposed matrix product without explicitly transposing the matrix. On GPU would be interesting to combine several graphics cards to increase performance and memory capacity.

References

1. Bian, J., Siewerdsen, J.H., Han, X., Sidky, E.Y., Prince, J.L., Pelizzari, C.A., Pal, X.: Evaluation of sparse-view reconstruction from flat-panel-detector cone-beam ct. Physics in Medicine and Biology 55, 6575–6599 (2010)
2. Dalton, S., Bell, N.: CUSP: A C++ templated sparse matrix library (2014), `http://cusplibrary.github.com/`, version 0.4.0
3. Feldkamp, L., Davis, L., Kress, J.: Practical cone-beam algorithm. Journal of the Optical Society of America 1, 612–619 (1984)
4. Ganine, V., Legrand, M., Michalska, H., Pierre, C.: A sparse preconditioned iterative method for vibration analysis of geometrically mistuned bladed disks. Computers & Structures 87(5-6), 342–354 (2009)
5. Hara, A.K., Paden, R.G., Silva, A.C., Kujak, J.L., Lawder, H.J., Pavlicek, W.: Iterative reconstruction technique for reducing body radiation dose at CT: Feasibility study. American Journal of Roentgenology 193, 764–771 (Sep 2009)
6. Heroux, M.A., Bartlett, R.A., Howle, V.E., Hoekstra, R.J., Hu, J.J., Kolda, T.G., Lehoucq, R.B., Long, K.R., Pawlowski, R.P., Phipps, E.T., Salinger, A.G., Thornquist, H.K., Tuminaro, R.S., Willenbring, J.M., Williams, A., Stanley, K.S.: An overview of the Trilinos project. ACM Trans. Math. Softw. 31(3), 397–423 (2005)
7. Im, E.J., Yelick, K., Vuduc, R.: Sparsity: Optimization framework for sparse matrix kernels. International Journal of High Performance Computing Applications 18(1), 135–158 (2004)
8. Jones, E., Oliphant, T., Peterson, P., et al.: SciPy: Open source scientific tools for Python (2001–), `http://www.scipy.org/`
9. Kaveh, A., Rahami, H.: Block circulant matrices and applications in free vibration analysis of cyclically repetitive structures. Acta Mechanica 217(1-2), 51–62 (2011)
10. Kourtis, K., Goumas, G., Koziris, N.: Optimizing sparse matrix-vector multiplication using index and value compression. In: Proceedings of the 5th Conference on Computing Frontiers. pp. 87–96. CF '08, ACM, New York, NY, USA (2008)
11. Krotkiewski, M., Dabrowski, M.: Parallel symmetric sparse matrix–vector product on scalar multi-core CPUs. Parallel Computing 36(4), 181–198 (2010)
12. Lee, B., Vuduc, R., Demmel, J., Yelick, K.: Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply. In: International Conference on Parallel Processing 2004 (ICPP 2004). pp. 169–176 vol.1 (2004)
13. Leroux, J.D., Selivanov, V., Fontaine, R., Lecomte, R.: Accelerated iterative image reconstruction methods based on block-circulant system matrix derived from a cylindrical image representation. In: Nuclear Science Symposium Conference Record, 2007. NSS '07. IEEE. vol. 4, pp. 2764–2771 (2007)
14. NVIDIA: CUSPARSE library (2014), `https://developer.nvidia.com/cusparse`

15. Pan, X., Sidky, E.Y., Vannier, M.: Why do commercial CT scanners still employ traditional, filtered back-projection for image reconstruction? Inverse Problems 25, 123009 (2008)
16. Rodríguez-Alvarez, M.J., Soriano, A., Iborra, A., Sánchez, F., González, A.J., Conde, P., Hernández, L., Moliner, L., Orero, A., Vidal, L.F., Benlloch, J.M.: Expectation maximization (EM) algorithms using polar symmetries for computed tomography CT image reconstruction. Computers in Biology and Medicine 43(8), 1053–1061 (2013)
17. Sheep, L., Vardi, Y.: Maximum likelihood reconstruction for emmision tomography. IEEE Transactions on Medical Imaging 1, 113–122 (1982)
18. Sidky, E.Y., Pan, X.: Image reconstruction in circular cone-beam computed tomography by constrained, total-variation minimization. Physics in Medicine and Biology 53, 4777–4807 (2008)
19. Soriano, A., Rodríguez-Alvarez, M.J., Iborra, A., Sánchez, F., Carles, M., Conde, P., González, A.J., Hernández, L., Moliner, L., Orero, A., Vidal, L.F., Benlloch, J.M.: EM tomographic image reconstruction using polar voxels. Journal of Instrumentation 8, C01004 (2013)
20. Thibaudeau, C., Leroux, J.D., Pratte, J.F., Fontaine, R., Lecomte, R.: Cylindrical and spherical ray-tracing for ct iterative reconstruction. In: Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), 2011 IEEE. pp. 4378–4381 (2011)
21. Vuduc, R., Demmel, J.W., Yelick, K.A.: OSKI: A library of automatically tuned sparse matrix kernels. Journal of Physics: Conference Series 16(1), 521 (2005)
22. Vuduc, R.W., Moon, H.J.: Fast sparse matrix-vector multiplication by exploiting variable block structure. In: Yang, L.T., Rana, O.F., Martino, B., Dongarra, J. (eds.) High Performance Computing and Communications, Lecture Notes in Computer Science, vol. 3726, pp. 807–816. Springer Berlin Heidelberg (2005)
23. Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., Demmel, J.: Optimization of sparse matrix-vector multiplication on emerging multicore platforms. Parallel Computing 35(3), 178–194 (2009)