Procedia Computer Science
Volume 66, 2015, Pages 642–650
YSC 2015. 4th International Young Scientists Conference on Computational Science

# Accelerating industrial applications: The development of basic GPU kernels for the new block AMG algorithms for solving SLE with explicitly calculated sparse basis

Ilya Afanasyev[1*], Yury Potapov[1†]
Sergey Sobolev[1‡], Sergey Kharchenko[2§]
[1]*Lomonosov Moscow State University, Moscow, Russia*
[2]*«Tesis»*
*Afanasiev_ilya@icloud.com, potap.yurich@gmail.com*

**Abstract**
Nowadays, GPU computations are playing significant role in supercomputing technologies. This work is a part of a project dealing with solving problems of modeling hydro- and aerodynamics where linear algebra operations are frequently used and occupy most of execution time. In despite of the fact that GPUs are traditionally used for solving high sized problems, in our project we need to solve many tasks of low sizes. Because of this, modern library's solutions such as cuBLAS (1) and cuSPARSE (2) are not suitable enough for that, so we have a task of implementation more efficient functions for concrete linear algebra operations taking into account its specialties.

*Keywords:* CUDA, performance optimizations, linear algebra

## 1 Introduction

This work has been performed in cooperation with Russian company «Tesis» that developed program package FlowVision (3). This package is an integrated multi-purpose solution for three-dimensional modeling of liquid and gas flows. FlowVision is based on the numerical solution of three-dimensional steady and unsteady equations of fluid dynamics and gas. However, nowadays it supports only the calculation on high-performance multicore CPU clusters. The main goal of this work is to

[*] Implemented DAXPY, MVM and LU decomposition operations
[†] Implemented DOT, transposed MVM and solve SLE operations
[‡] Work coordination
[§] Formulation of the problem

doi:10.1016/j.procs.2015.11.073

transfer some computational functions of package FlowVision from CPU to GPU, without whole code refactoring.
For this goal it was necessary to develop for GPU several base linear algebra operations of sparse and dense block matrices. Here is the list of the required operations:

1. **Operation DAXPY** is a common matrix multiplication of a small block by a block of dense vectors. It is also a block generalization of AXPY operation from first level of BLAS package.
   $$Y += X * A \quad (1.1)$$
   X and Y here are the blocks of dense vectors and A is the small block.

2. **Operation DOT** is the dot product of two blocks of dense vectors.
   $$Y += X * Z \; (1.2)$$
   X and Z here are the blocks of dense vectors and Y is the result of their dot product – a small block of fixed size.

3. **Multiplication of a sparse block matrix and transpose to it on a block of dense vectors**[**] is an update of block of dense vectors with different sign using matrix-vector multiplication:
   $$Y += A \times X \text{ или } Y -= A \times X \quad (1.3)$$
   A here is sparse block matrix and X is the blocks of dense vectors.

4. **LU decomposition of sparse block matrix** is the process of finding two matrices, corresponding to one of the following formulas in the case of complete and incomplete decomposition respectively:
   $$A \approx LU \text{ или } A = LU \quad (1.4)$$
   $A, L, U$ here are sparse block matrices with the same sparsity structures.

5. **Solution of block system with finely upper and lower triangular sparse matrix**
   $$AX = B (1.5)$$
   A here is a sparse triangular block matrix, B and X are dense blocks of vectors.

All mentioned operations have several important qualities: the size of sparse matrix block and block of dense vectors in any operation can take one of the discrete values: {4, 8 and 16}. Also, the most interesting cases is then this values are equal (for example DOT and DAXPY will be used in QR factorization with the same number of dense vectors in block). Type of floating-point data also has to be the same in a single operation.

For efficient implementation of these operations it is important to take into account the storage features of FlowVision package. First of all, due to mentioned above restrictions on the matrix and dense vectors block sizes it becomes possible to perform some optimizations connected to these fixed block sizes (implement different CUDA kernel for each important size). In the second place, all matrices and vectors, which are used in FlowVision, usually have a relatively small size, that's why it is unprofitable to use the whole GPU for a single operation. Finding optimal solution for this problem is the main idea of current paper and it is described in next paragraph.

[**] Later in this paper we will use the abbreviation MVMT and MVM for transposed or not multiplication respectively

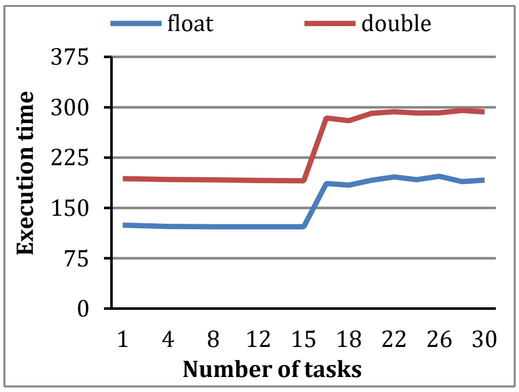
## 2 General result of the research

So, there is a problem of efficient GPU usage due to FlowVision data features, because it is extremely disadvantageous to launch sequentially a number of tasks, placed on the entire graphic card. In this work we use the following approach: execute each task on a fixed part of GPU's cores. This approach results in efficient parallel execution of up to 32 tasks on a single GPU, and moreover it allows efficiently using both CPU and GPU resources.

For implementing this approach it is important to have GPUs with Kepler architecture (or newer), because these GPUs support Hyper-Q technology. This technology is necessary for parallel execution of up to 32 CUDA kernels in different CUDA streams (in previous architectures there was possible to launch only up to 8 parallel kernels). Each kernel is one of the six mentioned above operations. Executed kernel uses GPU in the following way: it is almost always launched in {1 block, 1024 threads} configuration. This makes it possible to execute each operation in its CUDA stream continuously, and after execution this operation is followed by next one (probably with another type).

Due to the fact that each operation is executed with 1024 threads of a single block, it is placed on a single GPU multiprocessor. Modern multiprocessors are able to execute up to 2048 threads, that's why in our approach each multiprocessor executes two operations at the same time. Modern computational GPUs are equipped with up to 16 multiprocessors, so a single GPU can execute up to 32 tasks, all in parallel using Hyper-Q technology.

An experimental research showed that the most efficient way is to execute two operations on each multiprocessor, because this partially compensates memory latency. This research can be illustrated by the following graphs, showing the dependence of each operation's execution time from number of parallel launched tasks (operations of the same type for each test).

1) 

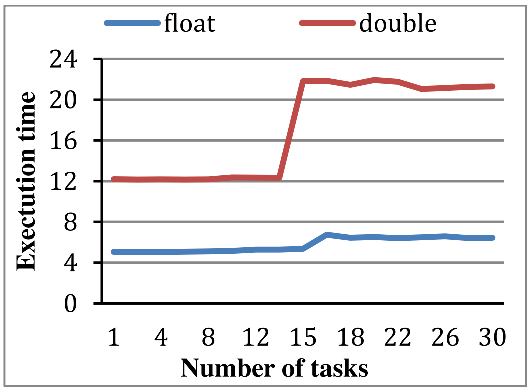
2) 

**Figure 1:** Dependence of execution time (ms) from number of tasks with fixed problem size for operations 1) DOT and 2) Matrix-vector multiplication.

All charts have been obtained on the K40 GPU, containing 15 multiprocessors. Thereby, using current approach up to 30 tasks can be launched in parallel and execution time of 30 tasks is less than two times longer than 15 tasks execution time, which proves the above thesis.
Moreover, this approach resolves the problem of simultaneous efficient usage of CPU and GPU. Indeed, the computations can be performed according to the following scheme: in the beginning, several CPU threads are launched (as an example, using Intel TBB), then some of them are attached to CPU cores and perform computations on CPU, while others control computations on the GPU (mainly launch kernels and copy data from host to device or vice versa).

Here are two figures: the first one illustrates the above scheme, the second is a profiling trace of DAXPY operation execution, which illustrates that the current scheme is working properly.

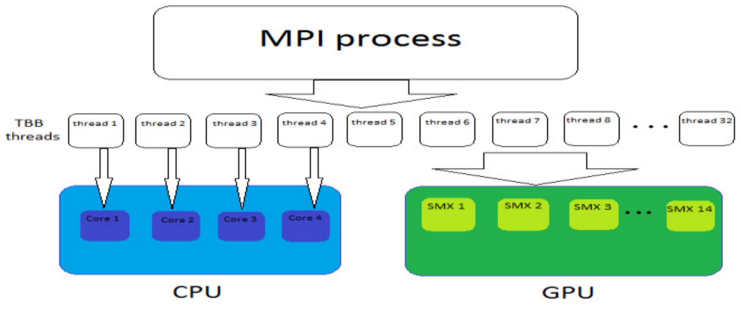


**Figure 2:** The interaction between CPU and GPU tasks using Intel TBB.

So, for example, for quad-core CPU and K40 GPU with 15 multiprocessors 3 + 30 CPU threads will be launched. Three of them will perform computations on CPU cores, while others on GPU.

## 3 Results for different operations

### 3.1 DOT operation

To reach the highest level of memory bandwidth dual caching was used. Threads are divided to groups. Each group has allocated memory for two blocks in shared memory and for three blocks on registers. We copy data from global memory to registers at the same time while computations are performing with preloaded data from registers to shared memory. Result C is placed on registers distributed and is gathered only after all computations.

```
void __global__ DOT(float *A, float *B, float *C, int N)
        Store B block on registers
        Load $A_0$ block from device to shared memory
        Load $B_0$ block from device to shared memory
        __syncthreads();
        FOR i = 0 to N – 2 // i is number of block, being currently handled
                Load $A_{i+1}$ block from device memory to registers
                Load $B_{i+1}$ block from device memory to registers
                Perform block multiplication $A_i$ (stored in shared memory) by B (stored in shared memory): $C_i += A_i \times B$
                (C is stored on registers)
                __syncthreads();
                Load $A_{i+1}$block from registers to shared memory
                Load $B_{i+1}$block from registers to shared memory
                __syncthreads();
        END FOR
        Multiple last blocks: $C += A_{N-1} \times B_{N-1}$
        Gathering C results
END DOT
```

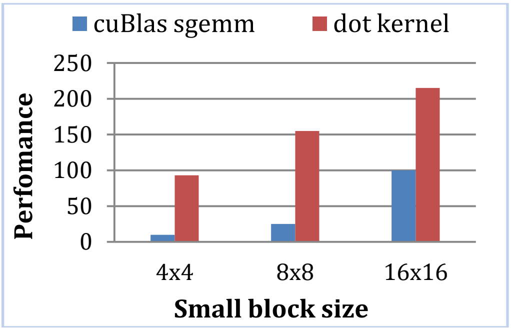


**Figure 3:** Performance comparison (in GFlops) between cuBLAS functions and DOT kernels. 30 tasks, $10^5$ elements in each vector in block, different block sizes**, float** accuracy.

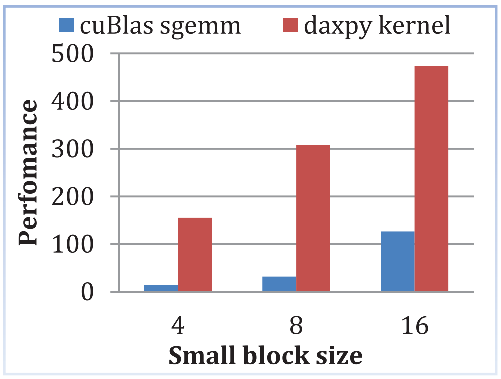
Implemented DOT kernels again outperform library analogues from 2 to 9 times.

## 3.2 DAXPY operation

This operation is implemented by several CUDA kernels for different sizes of small block and number of vectors in block. For equal sizes the highest performance has been achieved, due to efficient register and shared memory usage and high occupancy.
These kernels with the highest performance use the following computation scheme. In the most cases small block B is placed to the registers, while block from vectors A is loaded by threads from global memory using dual caching: in the beginning first portion of data is loaded to the shared memory, after that in parallel multiplication and new data portion loading is performed. High memory throughput is achieved due to a combination of register and shared memory usage. Also some kernels use __shfl instructions to avoid using shared memory at all. The following is pseudo-code of DAXPY kernel:

```
void __global__ DAXPY (Block vector A, Block X, Block vector &Y, int N)
        Load A block to registers
        Load $X_0$ block from global to shared memory
        __syncthreads()
        FOR i = 0 to N – 2 // «i» is number of block, being currently handled
                Load $X_{i+1}$ block from global memory to registers
                Perform block multiplication $X_i$ (stored in shared memory) by block A (stored on registers): $Y_i \pm= X_i \times A$
                __syncthreads()
                Store block result $Y_i$ of multiplication to global memory
                Load $X_{i+1}$block from registers to shared memory
                __syncthreads()
        END FOR
        Multiple last block in vector X by block A: $Y_{N-1} \pm= X_{N-1} \times A$ and store it to global memory
END DAXPY
```

The implemented kernels can be compared with library GEMM functions from cuBLAS, which do exactly the same operation with proper selection of the input parameters. The result of this comparison is shown on the diagrams below:

1) 

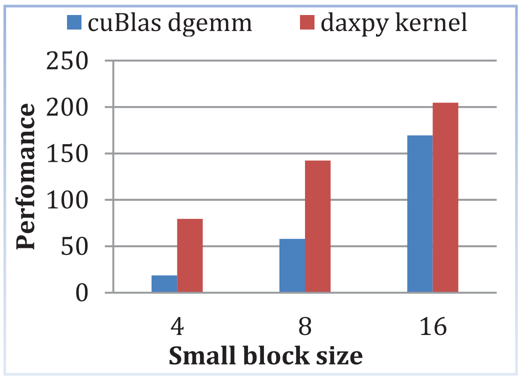
2) 

**Figure 4:** Performance comparison (in GFlops) between functions and kernels. 30 tasks, $10^5$ elements in dense one vector, different block sizes, (1) **float** and (2) **double** accuracy.

From the diagrams above it is clear that implemented DAXPY kernels outperform library analogues from 4 to 20 times.

## 3.3 Multiplication of a sparse block matrix and transposed to it on the block of dense vectors

Implementation of this operation consists from two independent parts: preparations for computations (performed on CPU), and the computations on GPU itself. Before the multiplication reordering of matrix rows is being made, the way that rows with the same number of blocks are places next to each other. As a result, it becomes efficient to call CUDA functions for groups of rows with the same number of blocks in it. It is also important to mention that there is no data moving between rows while reordering; instead of this only extra index array is being reordered. Moreover, such reordering has to be performed for each matrix only once before the computations, so the first CPU part doesn't affect computing time much.
CPU part consists of several steps:

1. Addition to matrix extra data structures: array of structures, describing groups of rows with the same number of blocks and array of references to new rows after the regrouping.
2. Regroup of indexes using standard quick sort on CPU and data copy from host to device if necessary.

GPU part consists of the following steps:

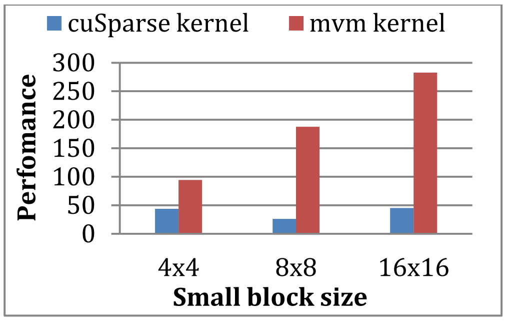
1. GPU launches a cycle by the number of rows groups with the same blocks number in each row.
2. For each group of rows __device__ function is called, performing multiplication of these rows by vector X. Each row is handled by group of threads, which number is equal to block size in square.
3. Blocks from each row are sequentially loaded to shared memory, and then the multiplication is performed. After that the result is placed to global memory and group of threads handle next row.

The following is pseudo-code of MVM kernel:

```
void __global__ MVM (Sparse matrix A, Block vector X, Block vector &Y)
    FOR group = 0 to GROUPS_NUMBER
        first_row = get_first_row_of_group()
        last_row = get_last_row_in_group()
        current_row = 0
        WHILE current_row < ( last row – first row )
            FOR current_block = 0 to BLOCKS _IN_ROW
                Load sparsity data of current block (in CSR format)
                Load $A_{cur\_row,row\_block}$ and $X_{row\_block}$ to shared memory using sparsity data
                __syncthreads()
                Perform block multiplication:
                $Y_{current\_block} \pm= A_{cur\_row,current\_block} \times X_{current\_block}$
                __syncthreads()
            END FOR
            current_row += ROWS_STEP
        END WHILE
    END FOR
END MVM
```

For computational of MVMT (multiplication with transposed matrix) the same actions are performed, but on the first stage the matrix is converted to CSC format and instead of rows extra data structures store information about columns. This approach allows using the same kernels with the minimal modifications.

Due to this approach it became possible to implement operation MVM on GPU even for relatively small matrix sizes (and also highly sparse matrices). Following is a comparison between implement kernels and library functions – cusparseSbsrmv from cuSPARSE library.

1) 

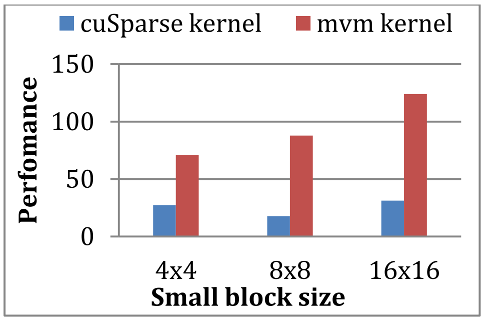
2) 

**Figure 5:** Performance comparison (in GFlops) between cuSPARSE functions and MVM kernel. 30 tasks, **float** and **double** accuracy.

It is clear that the current implementation of MVM outperform library analogues from 2 to 6 times.

## 3.4 LU decomposition of sparse block matrix

Operation of LU decomposition both on CPU and GPU consists of three parts:

1. Preparing data structures about matrix sparseness and supplementing them for the future computations.
2. Data copying from host to device (if we want to perform computations on GPU instead of CPU)
3. Launching computational kernel (or CPU function)

From this list it is clear that the first part is the same for CPU and GPU computations. Moreover, it has already been implemented by «Tesis» company, that's why it will not be described it this paper. Also this part is less computationally expensive, so it can be performed on CPU without significant time losses.

The following is a scheme of GPU kernel execution. The kernel is launched on 512 threads instead of 1024 in other operations, because it uses much more registers, what is limiting the occupancy. For each matrix row 5 actions are performed:

1. Updating current row using previous rows of upper triangular matrix and columns of lower triangular matrix – the most computationally expensive part. To a certain degree it is a sparse generalization of DAXPY operation and therefore will not be described here in more details.
2. Diagonal blocks modification (can be implemented efficiently using __shfl instructions)
3. Factorization and inversion of diagonal block
4. Multiplication rows blocks by the inverted diagonal block – it is sparse DAXPY generalization again.
5. Saving results to lower and upper triangular matrices.

Following is a table of the implemented kernels performance on complete matrix factorization.

| Matrix block size | **4x4** | **8x8** | **16x16** |
|---|---|---|---|
| Float performance | 135.7 | 221.9 | 229.5 |

**Table 1:** Performance (in GFlops) of LU decomposition with fixed matrix size and for different block sizes, float accuracy.

## 3.5 SLE solution

In this operation it is necessary not to find a single SLE solution with sparse block matrix A, but to find a solution for a block of SLE. Matrices of these systems are grouped into block matrix A and right parts of SLE are grouped into block of vectors B. Because the matrix is already in triangular form (it can be found by using already implemented LU factorization), only the back substitution of Gauss elimination is implemented with the only difference, that instead of matrix element and right part element blocks are used.
Before the elimination matrix is converted from CSR to CSC format for memory access locality.

## 3.6 Conclusion

In the course of this work six basic linear algebra operations from FlowVision package have been efficiently implemented. For each of implemented kernels a lot of different optimizations have been performed, different characteristics and properties have been studied.
From performance analysis using NVIDIA Visual Profiler it was determined that the main limitation on the performance of implemented kernels is shared memory bandwidth, which is used as a fast cache for computational data (usually small block of matrix).

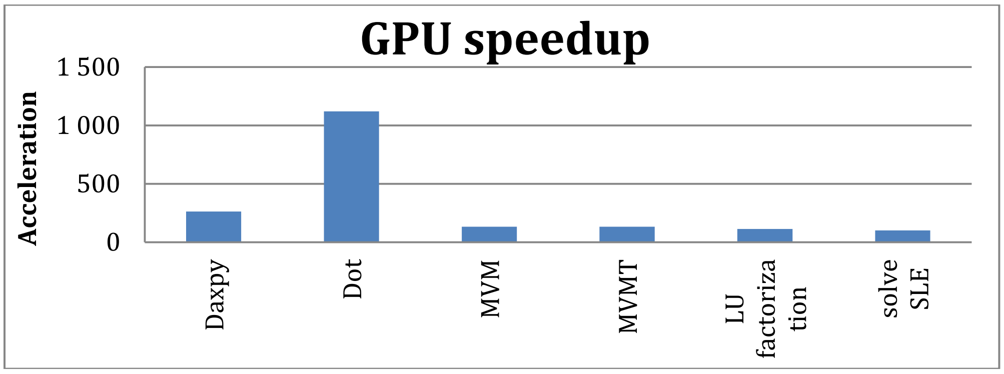


**Figure 6:** Acceleration relative to test one-core similar operations from FlowVision package.

Thus, in the future, these kernels can be successfully used in the software package FlowVision. It is also very important that acceleration in 2-20 times was obtained in comparison with the functions of the software packages and cuBLAS cuSPARSE by company NVIDIA.

Work was funded by the Ministry of Education and Science of the Russian Federation, Agreement N 14.607.21.0006 (unique identifier RFMEFI60714X0006).

## References

1. **Nvidia.** CuBlas. [Online] http://docs.nvidia.com/cuda/cublas
2. **NVidia.** cuSparse. [Online] http://docs.nvidia.com/cuda/cusparse
3. **Tesis.** FlowVision. [Online] http://www.tesis.com.ru/software/flowvision/
4. Villa O. et al. Power/performance trade-offs of small batched LU based solvers on GPUs //Euro-Par 2013 Parallel Processing. – Springer Berlin Heidelberg
5. Bolz J., Farmer I., Grinspun E., Schroder P. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid, Proceeding, 2003, Pages 917-924
6. Tomov S., Dongarra J., Baboulin M. Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems, Parallel Computing Volume 36, Issues 5–6, Pages 232–240
7. Anderson M., Sheffield D., Keutzer K. A Predictive Model for Solving Small Linear Algebra Problems in GPU Registers. Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International