EPFL

Parallel and High Performance Computing

Dr. Pablo Antolín

Solution Series 8

May 16 2025

CUDA - Graded exercise

1 Profiling and parallelization

Performance profiling is similar to the one in Series 5: the most time consuming part of the algorithm is the matrix-vector product. However, in this case it corresponds to an even larger percentage of the total time as we are using dense matrices instead of sparse ones. Therefore, our fist target is to parallelize the matrix vector product. Even if of minor importance, we will also parallelize the dot product and the vector addition in the CG algorithm.

The solution code can be found in the repository `math454-phpc/exercises-2025`, inside the folder `lecture_8/solution`. There you will find a new solver class named `CGSolverGPU` that derives from the original serial `CGSolver` class. Beyond some extra auxiliary functionalities, this new class just overrides the `solve` virtual method. This reimplementation is based on the original serial code, but it uses the GPU to accelerate the computationally demanding parts. Namely, the matrix vector product (implemented in the new method `dgemv`), the vector dot products (implemented in the method `ddot`) and the vector addition (implemented in the method `daxpy`). Therefore, the actual GPU acceleration is achieved by implementing the three methods `dgemv`, `ddot` and `daxpy` that are called in the `solve` method, and mimic the behavior of the corresponding BLAS[1] routines.

In order to offload these operations to the GPU, we need to allocate the needed memory on the GPU side and copy the data from the CPU to the GPU. Namely, the matrix A, the right-hand-side vector b and the solution vector x, but also some auxiliary vectors (r, p and Ap). The memory allocation and data transfer is performed once, before the iterations start, and the memory is freed after the iterations finish (and the solution vector is copied back to the CPU). In this way, we avoid the overhead of allocating and copying the data in each iteration.

2 Performance results

In this exercise we propose three different solutions (1, 2 and 3). They differ in the way in which the matrix vector product function `dgemv` is implemented, but they all use the same implementation for `daxpy` and `ddot` (except for Solution 1, that uses a naïve implementation of `ddot`). The key idea behind all the proposed GPU implementations is to fix the number of threads per 1D block

[1] https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms

and compute the 1D grid size (the number of blocks to launch) as the number of rows/columns in the matrix divided by the number of threads per block[2].
While the discussion of the different algorithms is deferred to Sections 3.1, 3.2, and 3.3, respectively, in what follows we will comment on the common performance aspects of the three solutions. The obtained results (as a function of the number of threads per block) are shown in Table 1 and Figure 1 for all the three solutions and are compared with cuBLAS[3], the NVIDIA CUDA Basic Linear Algebra Subroutines library, as a baseline reference. Some important remarks that can be made from these results are:

- The time per iteration is shorter for smaller matrices, what, even if obvious, it is worth mentioning.
- The cuBLAS performance is superior to the one of our implementations. This is not surprising as cuBLAS is a highly optimized library that uses the best available algorithms and imple-

[2]Note that some of the threads of the last block will be iddle as, in principle, the matrix size is not a multiple of the number of threads per block.
[3]https://developer.nvidia.com/cublas

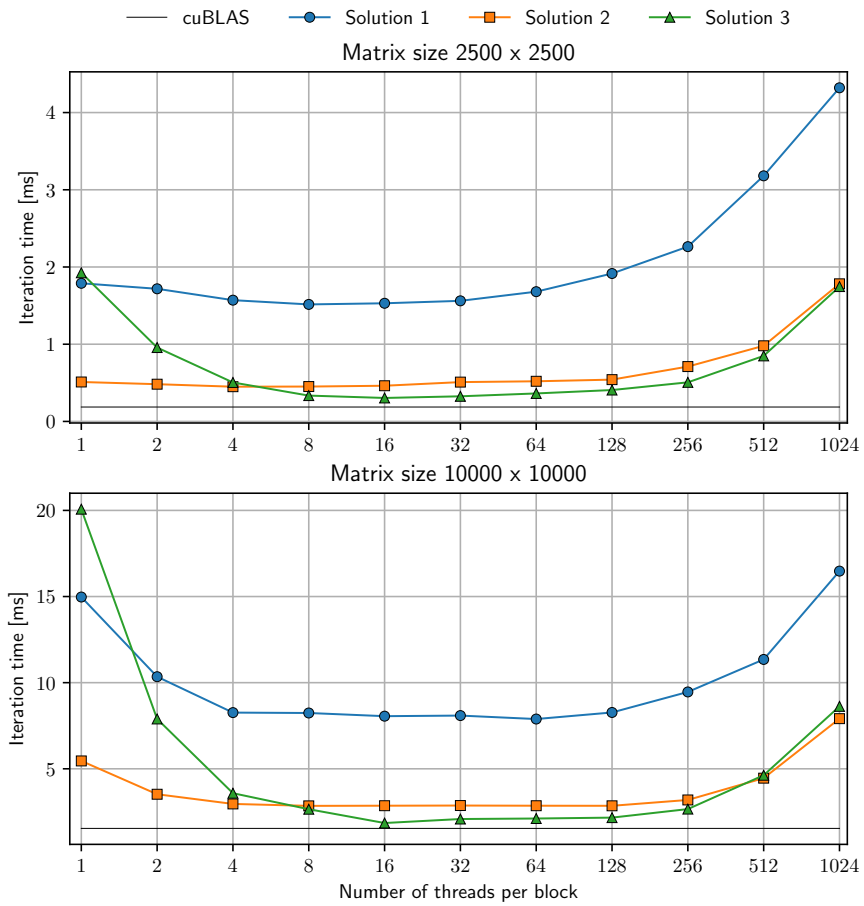

Figure 1: Time per iteration as a function of the block size for two different matrix sizes.

Table 1: Performance results for different thread block sizes

Threads/Block	**Time per iteration (milliseconds)**			
	cuBLAS	**Solution 1**	**Solution 2**	**Solution 3**
Matrix Size: $2,500 \times 2,500$				
1	0.19	1.79	0.51	1.92
2		1.72	0.48	0.96
4		1.57	0.45	0.50
8		1.52	0.45	0.33
16		1.53	0.46	0.30
32		1.56	0.51	0.33
64		1.68	0.52	0.36
128		1.92	0.54	0.41
256		2.26	0.71	0.51
512		3.18	0.98	0.85
1024		4.32	1.78	1.75
Matrix Size: $10,000 \times 10,000$				
1	1.54	14.97	5.46	20.07
2		10.35	3.52	7.89
4		8.26	2.96	3.58
8		8.24	2.85	2.65
16		8.05	2.86	1.85
32		8.09	2.87	2.08
64		7.89	2.86	2.11
128		8.27	2.85	2.17
256		9.46	3.19	2.66
512		11.35	4.46	4.63
1024		16.48	7.91	8.62

mentations for the GPU. However, it offers a good reference to compare our implementations with.

- All three implementations (for both matrix sizes) present a similar qualitative behavior: initially, the time per iteration decreases as the number of threads per block increases; then, there is an intermediate region where performance is approximately constant; finally, at a certain point the time per iteration starts to increase again. The explanation for this behavior is twofold:
  - The behavior for the initial decrease is due to the fact that when using a small number of threads per block thread warps[4] are underutilized.
  - The performance degradation for large number of threads per block is due to the fact that, depending on the matrix size, there is a limited number of blocks that can be launched in parallel. Thus, when the number of blocks is smaller that the number of Streaming Multiprocessors (SMs) in the GPU (80 in the case of the NVIDIA V100 GPU in `izar`), the GPU's occupancy is low and computational resources are not fully utilized. Indeed, this explanation is supported by the fact that the performance degradation for large number of threads per block starts earlier for smaller matrices.

An explanation for the performance of the three solutions is provided in the following sections, where the implementations proposed for `dgemv`, `ddot` and `daxpy` are described.

3 Implementation

3.1 Matrix vector multiplication – `dgemv`

The matrix vector product is the most computationally expensive part of the CG algorithm. For this reason it requires a careful implementation to achieve good performance. The simplified version of the BLAS `dgemv` routine implemented in this work corresponds to the operation $\texttt{y} \leftarrow \texttt{Ax}$, where `A` is a matrix, and `x` and `y` are vectors. Using indices, this operation can be expressed as:

$$\texttt{y}[i] = \sum_{j=0}^{n-1} \texttt{A}[i,j]\,\texttt{x}[j] \tag{1}$$

for $i = 0, \ldots, m-1$, where m and n are the number of rows and columns in the matrix `A`, respectively. The following three implementations are discussed in the following sections:

- **Solution 1** uses a single thread per matrix column.
- **Solution 2** uses a single thread per matrix row.
- **Solution 3** is a refinement of Solution 2 using shared memory to store the vector `x`.

3.1.1 Solution 1 – `dgemv_kernel_v1`

Solution 1 is *dummy* implementation that uses a single thread per matrix column. Once fixed the number of threads per block, the number of blocks to launch is computed as the number of columns in the matrix divided by the number of threads per block. Thus, each thread j is in charge of

[4] https://en.wikipedia.org/wiki/Thread_block_(CUDA_programming)#Warps

computing a full vector $\mathtt{y}_j$, the product of the j-th column of the matrix `A` and the j-th entry of the vector `x`, that is added to the final result `y` as $\mathtt{y} = \sum_{j=0}^{n-1} \mathtt{y}_j$. This is done by iterating along the rows and computing:

$$\mathtt{y}[i] \leftarrow \mathtt{y}[i] + \mathtt{A}[i,j]\,\mathtt{x}[j] \tag{2}$$

what requires synchronization when updating the entry $\mathtt{y}[i]$ as other threads may be also reading/writing from/to the same memory location. This synchronization (performed using atomic operations[5]) significantly slows down the performance of the algorithm.

3.1.2 Solution 2 – `dgemv_kernel_v2`

A possible improvement of the previous implementation is to use a single thread per matrix row. In this case, the number of blocks to launch is computed as the number of rows in the matrix divided by the number of threads per block. In this case, each thread i is in charge of computing the scalar product of the i-th row of the matrix `A` and the vector `x`:

$$\mathtt{y}[i] = \sum_{j=0}^{n-1} \mathtt{A}[i,j]\,\mathtt{x}[j] \tag{3}$$

This implementation is much more efficient as it does not require any synchronization as all these operations are embarrassingly parallel. See the results in Table 1 and Figure 1.

3.1.3 Solution 3 – `dgemv_kernel_v3`

In the matrix vector product, every single entry of the matrix `A` is read only once, but the vector `x` is accessed multiple times, what makes it a good candidate for using shared memory.
Thus, the previous (row-wise) implementation can be further improved by using shared memory to store the vector `x`. However, due to the reduced size of the shared memory, in principle it is not possible to store the whole vector `x` in shared memory. Thus, a possible strategy is to divide the vector `x` into chunks and associate a thread block to each chunk. Accordingly, the corresponding matrix columns will be processed by the same thread block.

We selected chunk size to be equal to the number of threads per block[6]. Thus, the number of blocks to launch is computed as the number of columns in the matrix divided by the chunk size. Then, within a block each thread is in charge of processing one (sub-) row of the matrix `A` (restricted to the columns associated to its thread block). As the maximum number of threads per block is 1024, there will be not enough threads within a single block to process all the rows of the matrix `A`, therefore each thread in a block will be in charge of processing multiple rows of the matrix `A`. The implementation looks like:

```
for(int r = threadIdx.x; r < m; r += blockDim.x) {
    double partial_sum {0.0};
    for (int c = c0; c < c1; ++c) {
        partial_sum += A[r * n + c] * shared_x[c-c0];
    }
    atomicAdd(y + r, partial_sum);
}
```

[5] https://docs.nvidia.com/cuda/cuda-c-programming-guide/#atomicadd

[6] This is an arbitrary choice and further options should be explored. In principle it is possible to select bigger chunks, however, selecting the chunk size close to the shared memory maximum size (96 KB in the NVIDIA V100) may be counterproductive as it may force SMs to process one single thread block at a time due to memory scarcity.

where `c0` and `c1` are the first and last column of the chunk associated to the thread block, respectively, and `shared_x` is the shared memory array that stores the previously loaded chunk of the vector `x` corresponding to the columns `c0`–`c1`, and that is reused many times by all the threads in the block.
As shown in Table 1 and Figure 1, this implementation is the most efficient one, as it reduces the number of accesses to the global memory (the vector `x`) and the number of required synchronizations (atomic operations) when compared to previous implementations.

However, there is still a subtle pitfall that likely hinders the performance of this implementation: many threads in a block are trying to access the same shared memory location (the same bank!) at the same time, what can lead to bank conflicts[7], so their accesses are serialized, which can significantly slow down the performance. This issues has not been addressed in this work, but fixing it may help to close the performance gap with cuBLAS.

3.2 Vectors addition – `daxpy`

Even of minor importance when compared to `dgemv`, `daxpy` can also benefit from GPU acceleration. It corresponds to the operation $\mathtt{y} \leftarrow \alpha\mathtt{x} + \mathtt{y}$, where α is a scalar and `x` and `y` are vectors.
Its implementation is straightforward: we fix a priori the number of threads per block and compute the number of blocks to launch as the number of elements in the vector divided by the number of threads per block. Then, each thread is responsible for computing the sum of one, and only one, entry from each vector and store the result in y. Notice that this operation is embarrassingly parallel, as each thread can work independently on its own entry, and does not need to synchronize with other threads. In addition, the memory (in vectors `x` and `y`) is accessed in a coalesced way, what is important for performance. This algorithm is implemented in the kernel `daxpy_kernel` and was used in all the three solutions.

3.3 Dot product – `ddot`

Finally, also the dot product operation can be parallelized using the GPU. It corresponds to $\mathtt{z} \leftarrow \mathtt{x}\cdot\mathtt{y}$, where `x` and `y` are vectors and `z` is a scalar.

A simple (naïve) implementation of the dot product, similar to the one proposed for `daxpy`, is to create as many threads as entries in the vector and let each thread compute the product of one, and only one, entry of each vector `x` and `y`. The computed products must be added to the result `z`. However, due to the fact that all the threads are trying to write to the same memory location, it requires to synchronize the access, what can be achieved using atomic operations. However, this is a very inefficient way of implementing the dot product, as it requires synchronization for all the threads that are trying to write to the very same memory location, what leads to a performance bottleneck. This the approach is implemented in the kernel `ddot_kernel_v1` and used in Solution 1.

In order to reduced the number of required synchronizations it is possible to do a partial reduction of the computed products (sum them together) at the level of each thread block (synchronization among the block threads is also required), and then add the block partial result to the final result `z` in the global memory (one thread per block does it and needs to synchronize with other blocks). Even if this approach also requires synchronization it is much more efficient as the number of threads trying to write to the same memory location is much lower (for both synchronizations) when compared to the previous case.

[7] https://developer.nvidia.com/blog/using-shared-memory-cuda-cc

However, in this work we propose a slightly more efficient approach that is based on the use of shared memory. The main idea is to use shared memory in each thread block to store the products of pairs of entries from the vectors `x` and `y`. Then, a reduction is performed over that shared memory to compute the sum of the products. This reduction can be performed (serially) by a single thread in each block or using a (parallel) recursive reduction algorithm (as implemented in the kernel `ddot_kernel_v2`)[8]. Finally, one thread in each block writes the block partial result to a global memory location `z` (what requires synchronization). This implementation proved to be the most efficient one and was used in Solutions 2 and 3.

4 Conclusions

In this exercise we have seen how to use the GPU to accelerate the CG algorithm. The most critical part of the algorithm from the performance point of view is the matrix vector product. Even of minor importance for CG, the dot product and the vector addition can also benefit from GPU acceleration.

The obtained performance strongly depends on the number of threads per block. When too small, threads warps are not fully utilized and the performance is poor. On the other hand, when the number of threads per block is too large (compared to the matrix size), the number of blocks to be launched in parallel is smaller that the number of SMs and GPU's occupancy is low, leading to underutilization of the computational resources. Thus, it is critical to choose the number of threads per block carefully.

Another important aspects to consider are how many times the memory is allocated and transferred from the CPU to the GPU, and vice versa, and how this memory is accessed within the GPU, as well as to consider the use of shared memory whenever possible. It is also important to keep to the minimum the required thread synchronizations as threads stall when waiting for other threads to finish.

Finally, and probably the most important take home message is that available libraries, such as cuBLAS, are highly optimized and should be used whenever possible.

[8] A similar result can be achieved by directly using warp-level primitives.