ADVANCED HIGH PERFORMANCE COMPUTING FINAL ASSIGNMENT

Msc in Data Science and Scientific Computing
University of Trieste

Author
Tommaso Tarchi
Trieste
7-15-2024

Contents

1 CUDA assignment

1.1 Introduction

In this assignment we want to compare performance and scaling of three MPI-parallelized programs for floating point matrix-matrix multiplication:

1. "naive" implementation (i.e. writing the function for multiplication from scratch);
2. using `dgemm` function from a BLAS library;
3. using `dgemm` function from cuBLAS library (i.e. the GPU-accelerated "version" of point 2.).

1.2 Parallel algorithm

The algorithm used to distribute matrix-matrix multiplication is similar for all three programs, differences are in the way multiplication is performed on the single MPI process and in data transfer for GPU-accelerated version.

Our goal is to perform:

$$C = A \times B,$$

where A, B and C are three square matrices of double precision floating point numbers.

We call n_{procs} the number of MPI processes.

To make communications efficient by performing the smallest number of data transfers possible, the following algorithm was adopted:

1. A, B and C are allocated as blocks of n_{rows} rows across processes (called A_{loc}, B_{loc} and C_{loc}).
2. Each process initializes its A_{loc} and B_{loc} randomly;
3. If using cuBLAS, A_{loc} is moved to device.
4. Considering each C_{loc} as divided into n_{procs} adjacent blocks of $n_{rows} \times n_{cols}$ elements, the following points are iterated n_{procs} times:
   - 4a. each process communicates to all other processes only B_{loc}'s data needed to compute the i-th block of C_{loc};
   - 4b. each process gathers data coming from all other processes into a single matrix, called B_{col};
   - 4c. if using cuBLAS, B_{col} is moved to device;
   - 4d. each process computes the i-th block of C_{loc} (if using cuBLAS, C_{loc} is computed directly on device).
5. If using cuBLAS, C_{loc} computed on device is moved back to host.

1.3 Implementation notes

Here we point out some details about the implementation:

- As BLAS library we chose Intel MKL 2023.2.0.
- Matrices distribution was handled by evenly distributing rows among processes. If the number of rows could not be divided exactly by n_{procs}, the remainder rows were assigned to first processes. In any case all rows assigned to a process were adjacent in the global matrices.
- For random initialization of A and B a unique seed was computed randomly by process 0 and broadcasted to all other processes, which then set different seeds for each one of their threads (openMP was used to speed up the initialization).
- Given the nature of communications between processes, they were implemented using the collective function `MPI_Allgatherv`.
- When using BLAS and cuBLAS, we accumulated the computed entries of C_{loc} directly on the memory allocated for C_{loc} (without using an auxiliary matrix). To do that, at each iteration of the algorithm we passed to `dgemm` the pointer to the first element of the block of C_{loc} to be computed as pointer to the result matrix and the number of columns of C as the stride argument.
- Considering that cuBLAS `dgemm` function assumes column-major order, we decided to avoid transpositions by computing on each process $B^T \times A^T$.

1.4 Profiling

All scaling results were obtained on Leonardo's partitions BOOST_USR_PRODUCT for GPU-acelerated programs and DCGP_USR_PROD for others.

MPI processes were distributed one per socket for non GPU-accelerated programs (i.e two processes per node) and one per device for GPU-accelerated programs (i.e. four processes per node).

We measured time for initialization of matrices, communications, computation and total data transfer between host and device. Times were averaged over all processes.

2 OpenACC assignment

2.1 Introduction

In this second assignment we had to take a serial code implementing Laplace equation by Jacobi method, make it parallel with MPI, accelerate it using openACC, and study its scalability. The Jacobi evolution is performed on a square grid of cells with static boundary conditions.

We implemented both a CUDA-aware MPI and a CUDA-anaware MPI version of the program.

2.2 Parallel algorithm

CUDA-unaware

1. Each process allocates its own portion of the grid (divided as blocks of rows), with an additional contour of cells as boundaries. Also, an identical auxiliary grid to store updated cells state is allocated.

2. Each process initializes its own cells, lateral boundaries and, only for first and last processes, upper and lower boundaries.

3. Both main and auxiliary grids are moved to device.

4. For the required number of iterations, repeat the following:

   4a. each process communicates to the bordering processes the corresponding neighbor rows (i.e. the first row to the upper process and the last one to the lower process) from host to host;

   4b. received boundaries are updated on device;

   4c. each process evolves its own cells on device on the auxiliary grid and then swaps pointers on both host and device to update the main grid;

   4d. updated border cells are copied back to host.

5. The final grid is copied back to host.

CUDA-aware

1. Same as in CUDA-unaware.

2. Same as in CUDA-unaware.

3. Same as in CUDA-unaware.

4. For the required number of iterations, repeat the following:

   4a. each process communicates to the bordering processes the corresponding neighbor rows (i.e. the first row to the upper process and the last one to the lower process) directly from device to device;

   4b. each process evolves its own cells on device on the auxiliary grid and then swaps pointers to update main grid.

5. Same as in CUDA-unaware.

2.3 Implementation notes

Here we point out some details about the implementation (we call n_{procs} the number of MPI processes):

- Grid distribution was handled by evenly distributing rows among processes. If the number of rows could not be divided exactly by n_{procs}, the remainder rows were assigned to first processes. In any case all rows assigned to a process were adjacent in the global grid.

- To speed up the grid initialization we used openMP.
- Boundaries communications between processes were performed by "exchanging" bordering rows between neighbor processes. To do that we used `MPI_Sendrecv()`, and `MPI_PROC_NULL` to make code more readable.
- Data movements were managed by placing an openACC data region around the for loop performing the evolution of the grid, and using clauses `copy` for both grids. Using `copyin` for the auxiliary grid was not possible, as pointer swap strategy would have not worked for odd numbers of iterations of the algorithm.
- In the CUDA-unaware version we used the `update` clause to copy borders from host to device and vice versa.
- For CUDA-aware comunications we used an openACC region with clauses `host_data` and `use_device` to "wrap" MPI calls.

2.4 Profiling

All scaling results were obtained on Leonardo's partitions BOOST_USR_PRODUCT for GPU-acelerated programs and DCGP_USR_PROD for others.

MPI processes were distributed one per device (i.e. four processes per node).

We measured time for initialization of grids, communications, computation, one-time host-device data movements and in-loop data movements. Times for communication, computation and in-loop data movements were computed as the average times over all iterations of the algorithm, then averaged over all processes. Other times were only averaged over processes.

3 MPI Remote Memory Access assignment

3.1 Introduction

In this last assignment we took the same serial code as in the second one, and made it parallel using MPI Remote Memory Access paradigm.

The code was further parallelized using openMP.

3.2 Parallel algorithm

The algorithm is the following:

1. Each process allocates its own portion of the grid (divided as blocks of rows), with additional columns as lateral boundaries, and also two additional arrays to store the upper and lower boundaries. Also, an auxiliary grid to store updated cells state is allocated.
2. Each process initializes its own cells, lateral boundaries and, only for first and last processes, upper and lower boundaries.
3. For the required number of iterations, repeat the following:

3a. each process opens two windows on its border arrays to allow (only) neighbor processes to write directly to them;

3b. each process writes directly to the bordering processes' border arrays the corresponding neighbor rows (i.e. the first row to the previous process and the last one to the following process);

3c. each process closes the opened windows;

3d. each process evolves its own cells on the auxiliary grid and then swaps main and auxiliary grids.

Notice that, at each iteration the computation is split between before and after closing the windows, to allow computation and communication to overlap (see section 3.3 for details).

3.3 Implementation notes

Here we point out some details about the implementation (we call n_{procs} the number of MPI processes):

- Grid distribution was handled exactly as in the previous assignment (see section 2.3). The only difference is in the fact that here upper and lower boundaries were separated from the actual grid, to make communications easier.
- To speed up grid and boundaries initialization and grid evolution we used openMP.
- For RMA operations we used the `MPI_Put()` function for direct writing.
- To make communications more efficient we adopted the following strategy:
    1. the boundary arrays were allocated contextually to windows using `MPI_Win_allocate()`;
    2. we created for each process two `MPI Groups` containing one only the previous and one only the following process, and used them to expose the windows exclusively to the interested processes;
    3. we used `MPI_Win_post()`, `MPI_Win_start()`, `MPI_Win_complete()` and `MPI_Win_wait()` to make synchronization less "rigid"; in particular we evolved the internal cells before calling `MPI_Win_wait()`, so the evolution of these cells could be performed even if the communication was not completed on the target.

3.4 Profiling

All scaling results were obtained on Leonardo's partition DCGP_USR_PROD.

MPI processes were distributed one per socket, i.e. two processes per node.

Times were measured in the same way as in the previous assignment (see section 2.4), but obviously without host-data movements times.