U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

May 2016

Implementation and analysis of Matrices Multiplication algorithms

Parallel Computing - 2nd assignment

Daniel Gomes and Diogo Gomes
{up201306839, up201106586}@fe.up.pt

Problem description

One simple way to make an algorithm execution faster, is to run it on a CPU – central processing unit – with a higher clock speed. Yet, this solution shows up to be highly limited as the heat dissipation problem and the consequently highly energy efficiency loss, makes it quickly impractical.

Another approach is to find on the algorithm partitions that are independent from the remaining and execute them simultaneously on multiple execution threads. Modern computers are already equipped with multi-core processors, which allow for this multi-thread programming, but this is often not enough, as the number of cores are limited. In order to surpass this, multiple computers, possibly with multi-core processors each, are used in order to achieve a scalable solution.

Having multiple threads performing a common operation requires that each one has access to the input data and the exchange of intermediate results is possible. When these are running on different machines, this is not a trivial task as a global shared memory is inexistent and so the exchange of data communication methods is required.

Another idea for the gathering of multiple cores is to replace CPUs by GPUs – Graphics Processing Units. With a different architecture GPUs can execute the same algorithm on thousands of simultaneous threads.

In this project, the product of matrices will be analysed using both of these approaches. Firstly using NVIDIA CUDA® – Compute Unified Device Architecture – a framework to execute programs on NVIDIA GPUs. After, the distributed multiple computer/CPU manner will be also analysed, through the implementation of SUMMA – scalable universal matrix multiplication algorithm – using the Open Message Passing Interface (OpenMPI) to facilitate data exchange.

The basic matrix multiplication operation

The matrix multiplication is a mathematical operation that takes as the input two matrices, let's say ***A*** and ***B***, and outputs a another matrix, ***C***. The formation of the third matrix is achieved by:

$$C(i,j) = \sum_{k=1}^{A_{cols}} A(i,k) * B(k,j)$$

As discussed in our previous work – Performance evaluation of a single core – this could be performed in a efficient way by traveling matrices line by line performing multiple rank-1 updates on each cell. This algorithm has an complexity of $\mathbf{3 \times O(n)^3}$.

```
for(i = 0; i < A_rows; i++)
        for(k = 0; k < A_cols; k++)
                for(j = 0; j < B_cols; j++)
                        C[i][j] += A[i][k] * B[k][j];
```

Fig$_1$ - pseudo-code of sequential matrix multiplication algorithm

When analysing the algorithm, something to notice is that from matrices ***A*** and ***B*** are only performed reads and so, the calculation of each ***C(i, j)*** can be realized independently from all others. Also, for the calculation of each ***C(i,j)***, only values of the row ***i*** of ***A*** and of column ***j*** of ***B*** are required.

These characteristics makes of this algorithm an embarrassingly parallel problem which can be easily solved by split into smaller tasks through data parallelism. This will be explored later in this document.

Matrix multiplication on the GPU

NVIDIA CUDA® platform

The programming model implemented by NVIDIA CUDA® platform is substantially different from the one used on mainstream CPUs. While on regular CPUs a program runs only on one processor, on CUDA, as it is built on SPMD – Single Program Multiple Data – model, the same program runs simultaneously on multiple threads. This is ideal for the data parallelism that it is intended to be implemented.

On CUDA model, threads are organized in blocks. Each thread exist in a block and has its own local memory. Each block has its own memory which can be accessed by threads in it. Also three different types of global memory are available. The local memory is the fastest one, while the global is the slowest. The shared memory and the constant global memory, are significantly faster than global memory but much smaller in size.

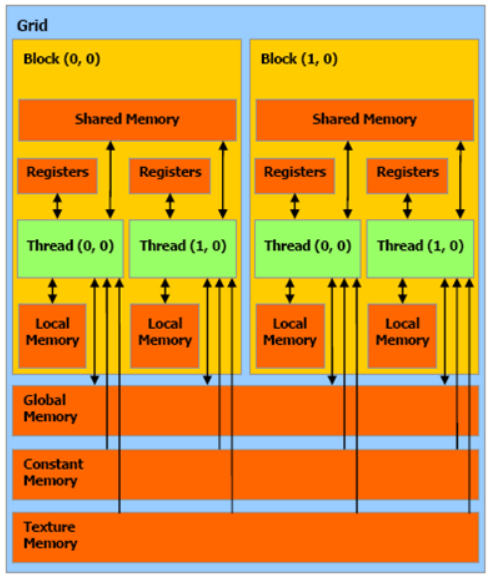


Fig_2 - CUDA's memory and block's grid model

Further, CUDA platform provides the necessary API – Application Programming Interface – when launching the program on the GPU to define the configuration of blocks and threads. It also provides methods for copying data between CPU memory to GPU global memory and for each running thread its block and thread identificators.

It is also important to mentioned that a block runs on an independent core and although there are only a few hundred cores, it is possible to launch programs in much larger block grids. In this scenario CUDA divides core processing times among multiple blocks.

This whole environment allows for a completely different approach in comparison to the sequential CPU approach. Instead of using chained loops to travel through each cell of **C**, one thread can be assigned to each.

Implemented algorithm

In order to achieve maximum performance of CUDA model, all blocks should be used. This way, matrix **C** is split into tiles which are mapped to blocks. Each block's cell is then assigned to a thread of the respective block.

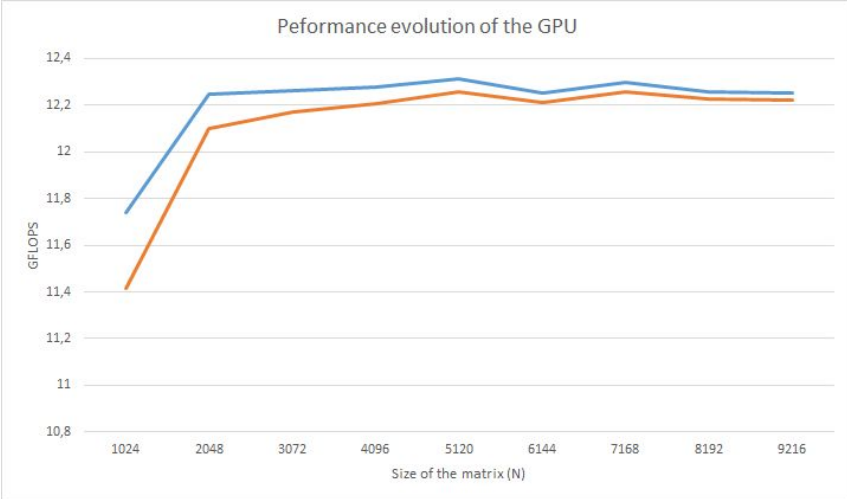
Now, the simple approach would be to make each thread loop through each line of **A** and column of **B** and sum all multiplied pairs and finally set the value of **C** cell. Yet, it should be taken into consideration that global memory accesses are much slower compared to block shared memory access. Given the fact that multiple threads on the same block require the same cells of **A** and **B,** a much efficient solution can be achieved by dividing the two in tiles. All blocks threads loop through each |N|/Tw (where Tw is the tile's width) of the copying the corresponding tile accomplishing together the storage of whole tile into shared memory. Then, each thread can use the needed row of **A** or column of **B**, 'cached' on shared memory.

```
For m = 0 until m < |N|/Tw
        A_shared(i,j) = A(m*Tw + Thread_x ,Tw*Block_y + Thread_y)
        B_shared(i,j) = A(Tw*Block_x + Thread_x, m*Tw + Thread_y)
        For k = 0 until k < Tw
                Tmp += A_shared(Thread_x,k) * B_shared(k, Thread_y)
C(Thread_x,Thread_y) = Tmp
```

Fig$_3$ - pseudo-code of the matrix multiplication with CUDA

Performance measures

In order to test the algorithm performance several runs were executed with matrices sizes varying from 1024x1024 to 9216 in intervals of 1024. The elapsed time was recorded both including the necessary time to copy the matrices from CPU to GPU and without it. Then the values performance were calculated taking into account the theoretical number of operations on the sequential algorithm: $3 \times O(n)^3$.



Fig$_4$ - GPU performance test results. In blue the elapsed performance considering the elapsed time to copy data between CPU and GPU with memory

The tests were were executed 5 times each on a NVIDIA GeForce GT 650M with 2GB DDR3 memory. The size of threads per block was the maximum possible, 32 by 32 meaning 1024 per block. This GPU has 384 cores with an warp size of 32. This means that of N^2 threads only 384*32 = 12288 were actually running in parallel.

Although this GPU possesses a theoretical maximum performance 30,4 GFLOPs on double precision operations the same were never nearly achieved, having the performance stabilized around 12 GFLOPS given the number of global memory accesses still performed. The usage of constant memory for the storage of **A** and **B** was thought but, given the size of these matrices constant memory was not enough. Another possibility could be use CUDA **__strict__** flags to declare these as constant, yet this possibility is only available from NVIDIA GPUs with 3.5+ compute capabilities, while 650M supports only 3.0.

Matrix multiplication with multiple CPUs

Message Passing Interface (MPI)

With the multiple CPU approach, there is a need to establish communication channels between the involved nodes, since there isn't a Global Shared Memory. OpenMPI solves this problem by implementing an API defined by the MPI standard. It provides an easy way to exchange messages in a parallel system because of some of its characteristics, for example, point-to-point and collective communication, synchronization and easily definable subsets and topologies of processes.

Each one of the processes associated have a distinct global rank that allows the process to accomplish different tasks if needed. This rank is associated with the default communicator group, MPI_COMM_WORLD, but OpenMPI can have more than 1 communicator, and inside each one the processes have an associated rank.

Multiplication of matrices with SUMMA

Extracting from "SUMMA: scalable universal matrix multiplication algorithm" by R. A. Van De Geijn and J. Watts, two slightly different versions of summa where implemented. Both follow the same principles and only differ in the way matrices **A** and **B** are shared between processes.

In SUMMA, matrices **A, B** and **C** are split in equal **P** tile blocks. Computer processes are arranged in a square matrix manner, being mapped each tile of **C** mapped to the corresponding block ***P(i, j)***. The algorithm starts with each process having in its possession the corresponding tile **(i, j)** of matrices **A** and **B**. SUMMA takes also advantage from the fact that for its calculations each processor **P** only requires tiles of its corresponding line o **A** and column of **B**. Also, the calculation of tile cells can be performed by multiplying all the pairs of tiles of a given row and column and adding them.

For simplification purposes, square matrices were used and limited the number of processes to number whose square roots are integers.

Broadcast based communication

In this version on each iteration **k** iteration, the process that has in its possession the correspondent **A** tile broadcasts it across the entire processes row, the one that has the **B** tile broadcasts it across the processes column.

```
for k = 0 until k < n / p
        If P_col == k
                Broadcast A(I,k)_tile to the entire row
        If P_row == k
```

```
                Broadcast A(k,J)_tile to the entire column
        Receive A(I,k)_block as A_tile
        Receive B(k,J)_block as B_tile
        C(I,J) = C(I,J) + A_tile * B_tile
```

Fig$_5$ - pseudo-code of the SUMMA algorithm using tiles instead of single rows/columns. The 'k' represents the current line/row block, the 'P' the current process row or column and 'p' the number of processes

Ring based communication

In this second version on all iterations, each node sends passes the its current tile of **A** to the next process in its row, and the current tile of **B** to the next process of its column. This ensures that after **k** iterations all processes had access to all of its row and column.

```
start_k = (I+J) % √|P|
nodes_size = √|P|
k = start_k
Do
        A_current = I == k ? A_local : A_saved
        A_current = J == k ? B_local : B_saved
        If I+J % 2 == 0
                Send A_current to next node in row ring
                Recv A_saved from next previous row ring node
                Send B_current to next column node ring
                Recv B_saved from prev column node ring
        Else
                Recv A_saved from next previous row ring node
                Send A_current to next node in row ring
                Recv B_saved from prev column node ring
                Send B_current to next column node ring
        C_local = A_current × B_current
        k = next(k, nodes_size)
While k != start_k
C(I,J) = C_local
```

Fig$_6$ - pseudo-code of the SUMMA algorithm with ring-based communication.

Performance evaluation and scalability analysis

Both versions of the SUMMA were run on 16 and 64 processors distributed through 4 and 16 Intel® Core™ i7-4790 CPUs with matrices from 1024 width to 9216 with steps of 1024. Also for the broadcast version tests were run from 1080 to 7560 with steps of 1080 with 36 processes.

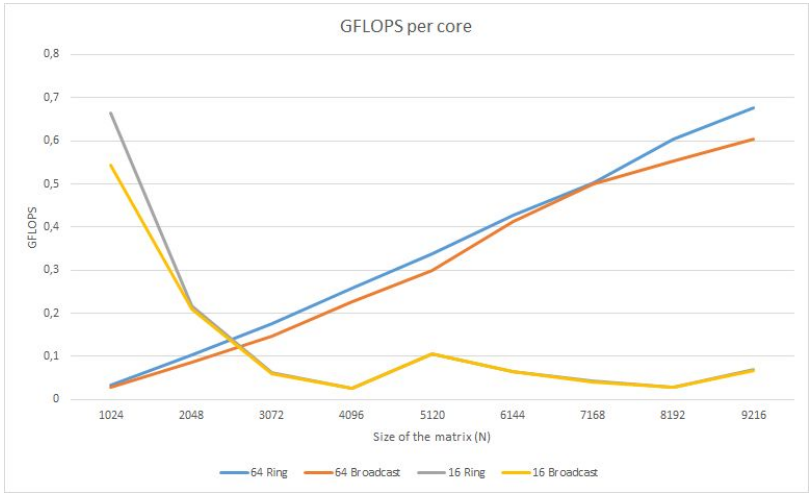


Fig_7 - evolution of the performance of each core in function of the input's matrices size

As it may notice be noticed with the tests ran with 16 cores, after N=3072 the performance per core tends to stabilize and to around 0,05 GFLOPs which evidences the the scalability of the algorithm. When looking to 64 test curves it is noticeable that these matrices sizes are too small for this architecture as performance values keep increasing. Although there isn't an significant distinction, is still possible to notice that GFLOPS registered on version with ring are always equal or superior than with broadcast version.

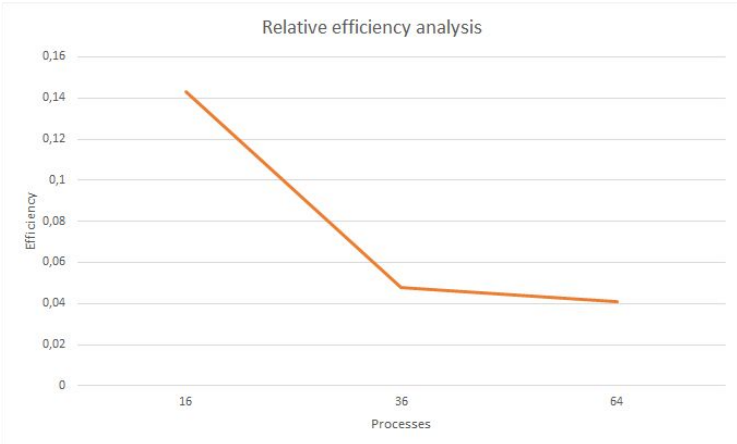


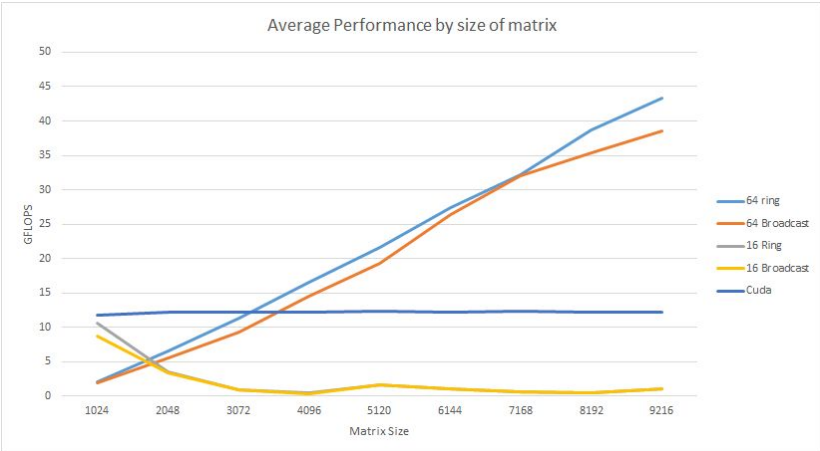
Fig_8 - relative efficiency of broadcast algorithm for N=7168 having the CUDA version has the sequential reference value. $E=(T_{cuda}/T_{CPUs})/N_{processors}$

Here the resolution is not the best but it possible to notice that from 36 to 64 processors the curve seems to start converge an equilibrium value of 0,03 to 0,04 efficiency value.

Conclusions

With this project it was possible to understand better two of the most common solutions when scaling an algorithm through parallelization: either by combining multiple CPUs or/and using GPUs.

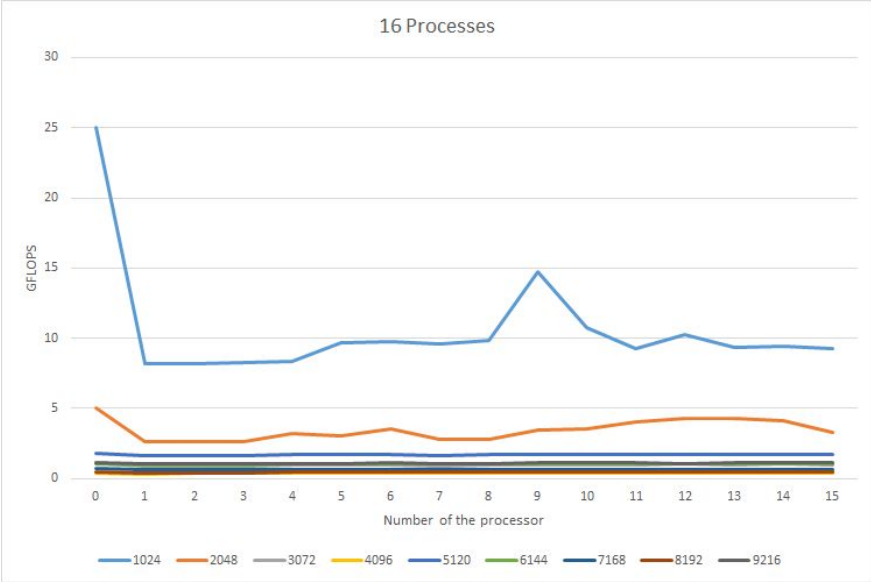
One good surprise was seeing the potential of GPUs, for example, in our tests the laptop graphics card was capable of outperform 4 desktop CPUs together. An interesting study would be joining these two approaches on a single one. While SUMMA would handle the matrix segmentation and communication between multiple processors, the local matrix multiplication would be performed on the GPU.



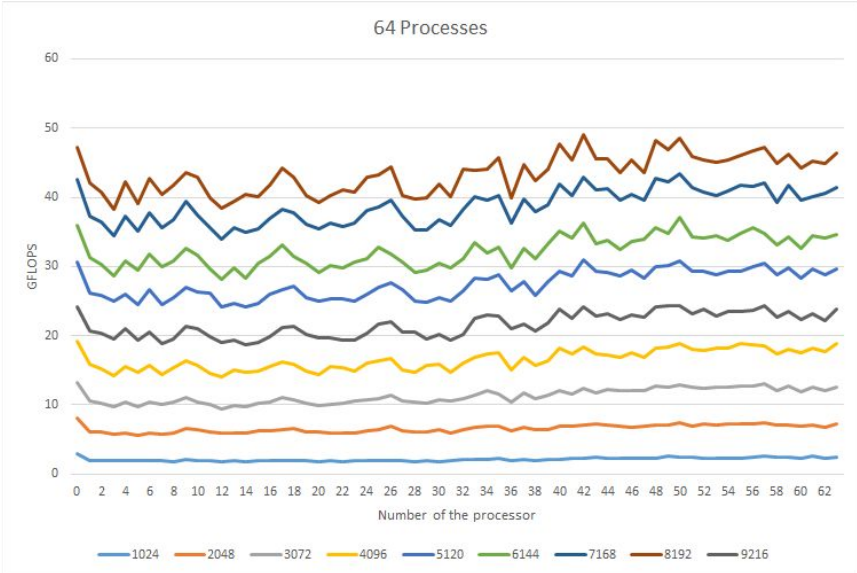
Fig$_9$ - global performance of all analysed versions of matrix multiplication algorithm in GFLOPS

Appendix

Here are included 2 graphs that show performance values on each process.



Fig_{10} - performance of each core in GFLOPS for the ring version running on 16 processes



Fig_{11} - performance of each core in GFLOPS for the ring version running on 64 processes