ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY (ROMJIST)
Volume **25**, Numbers 3–4, 2022, 368–380

Improvements in Data Transfer for a MapReduce Accelerator

George-Vlăduț POPESCU

University Politehnica of Bucharest, Romania

Email: george.popescu1012@upb.ro

Abstract. This paper presents a way to improve data transfer for a MapReduce Accelerator designed for parallel data processing and integrated into a heterogeneous computing system. The proposed solution is a new module called the Data Transfer Engine, integrated into the structure of the Accelerator. This module takes over the tasks related to data transfer, separating the data processing and data transfer flows, thus reducing the total execution time of a program. Two algorithms for processing large matrices were proposed to evaluate the improvement in the total execution time brought by the Data Transfer Engine. These algorithms are based on a custom linear algebra library used to run basic matrix operations on the target Accelerator.

Key-words: Data Transfer Engine; heterogeneous computing; improved data transfer; MapReduce Accelerator.

1. Introduction

The increasing need for high computing power, imposed especially by artificial intelligence applications, has led to the development of parallel processing architectures, which try to offer the best possible support in terms of execution time and power consumption.

Several architectures have been proposed over time, each of them trying to improve the *number of operations/consumed power* ratio as much as possible.

The Many Integrated Core (MIC) Architecture is a solution proposed by Intel for highly parallel computation [1]. The Intel Xeon Phi product family, which implements this architecture, are general purpose, many-core coprocessors, having up to 72 cores. Each core has a Vector Processing Unit and can execute both scalar and vector instructions. Each core has two cache levels, with the level-two cache (L2) being shared. The communication inside the coprocessor is ensured by a high-bandwidth ring bus which connects all the cores and the memories [2].

Graphics Processing Units (GPUs) are processors with high number of cores initially designed to accelerate tasks related to computer graphics. Nevertheless, their parallelism makes them suitable also for other applications, such as the artificial intelligence.

The best-known GPUs are those manufactured by NVIDIA. They contain thousands of cores, whose implementation has been adapted over time to better respond to challenges from the field of artificial intelligence. Depending on the architecture, those cores are grouped into processing blocks, each having their own register set and instruction cache. Those blocks are further grouped into Streaming Multiprocessors (SMs), with separate L1 caches for data and instructions. An important observation is the existence of a shared memory on the same level as L1, allowing communication between SMs. Several Streaming Multiprocessors are grouped in a Graphic Processing Cluster, the GPU having several such clusters with a common L2 cache [3–6].

The GPU's processing cores are specialized in integer, single-precision, and double-precision operations. Although the initial NVIDIA GPUs were based only on CUDA cores, in the latest products, new cores, called Tensor cores, were added to each Streaming Multiprocessor. Those cores are execution units specialized in matrix operations, intensively used in deep learning training and inference.

Google's Tensor Processing Unit (TPU) is an Application-Specific Integrated Circuit, designed for machine learning applications. The TPU is based on two-dimensional Matrix-Multiply Units, which are systolic arrays of 256×256 or 128×128 multipliers, depending on the version of the architecture. Although the first version offered support for 8-bit integer operations, the following ones could also perform 16-bit or 32-bit floating point operations [7].

The previously presented solutions have a high theoretical peak performance, but their actual performance for different tasks may be much lower [8]. This performance limitation may be caused by the way the data transfer is organized.

The need of a solution that offers both flexibility and good performance have led to the emergence of heterogeneous systems. Those systems integrate CPUs and other processing elements specialized for certain tasks. To support the development of these systems and to offer very good flexibility, FPGA manufacturers have created systems that combine a CPU and an FPGA on the same chip. Using those platforms, the user can describe his own processing core, which can be configured and adapted to the target application.

In this paper, a MapReduce Accelerator specialized in parallel processing tasks is analyzed and solutions to improve the data transfer are presented. The theoretical performance of such an Accelerator was analyzed in [9] and [10]. The current version of the MapReduce Accelerator is part of a heterogeneous processing system based on a Zynq-7020 SoC from Xilinx, integrated on a PYNQ-Z2 board. User interaction with the Accelerator is ensured by a custom programming environment, developed based on the PYNQ software package [11].

2. The Target MapReduce Accelerator

The target processing element for which the data transfer improvement was investigated is a MapReduce Accelerator, a parallel computing core that can process large amounts of data, making it suitable for the intensive computation part of a program.

The Accelerator is part of a heterogeneous computing system that integrates a general-purpose CPU, which ensures communication with the external world and can run an operating system, and the Accelerator, specialized in parallel computing. A general view of the architecture of such a system is presented in Figure 1.

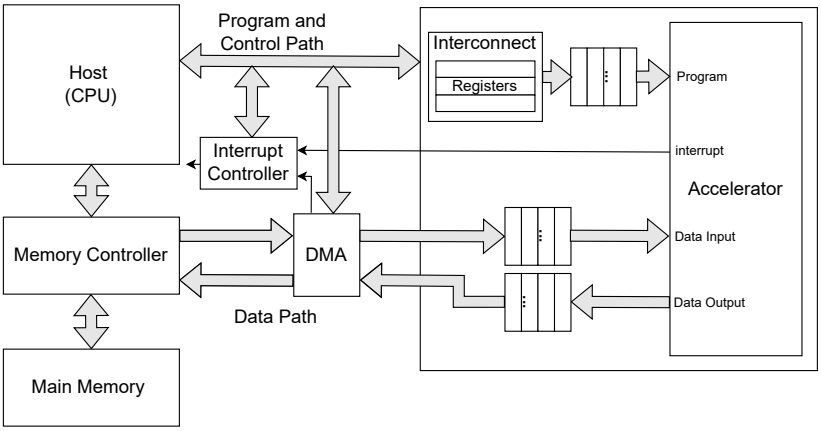

Fig. 1. The architecture of the heterogeneous computing system

The Accelerator communicates with the rest of the system that integrates it using three main channels: the Program and Control Path, used to receive the instructions from the Host and to send debug and status data, the Data Input Path, used to read from the main memory the data to be processed, and the Data Output Path, used to send the result to the main memory. The Program and Control Path is also used by the Host for configuring other elements in the system, such as the Interrupt Controller and the DMA used for efficient data transfer between the main memory and Accelerator.

The main components of the Accelerator are the Controller and the Parallel Processing Unit. The Controller is mainly used for coordinating the activity of the Accelerator and for performing operations on scalar data. The Parallel Processing Unit is responsible for processing vector-type data.

The structure of the Accelerator and its detailed interface are presented in Figure 2.

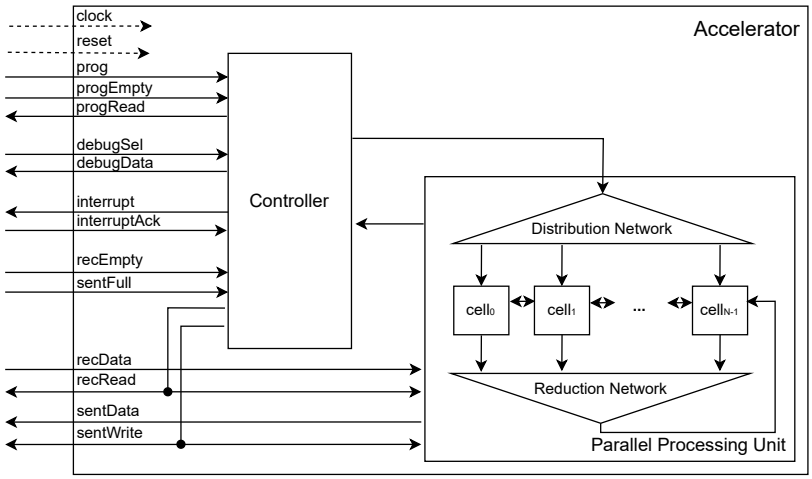

Fig. 2. The structure of the Accelerator

The main role of the Controller is to coordinate the execution process inside the Accelerator. It has a program section, responsible for the execution control, a data section, used to perform operations on scalars, and a connector section, used to prepare and send the commands to the Parallel Processing Unit. It also contains a *cycle counter* used for performance evaluation.

The Parallel Processing Unit is responsible for the processing of data organized as vectors or matrices. Its main components are the Distribution Network, the Array of processing cells, and the Reduction Network.

The Distribution Network is a pipe-lined structure used to send instructions, addresses, memory commands, or data from the Controller to the Array of processing cells. Its dimensions vary with the number of cells in the Array (N), having a depth of $log_2 N$. An important observation is that the information structure sent by the Controller through the Distribution Network has separate fields for instruction, value, and addresses and commands used in memory accesses.

The central element in the Parallel Processing Unit is the Array of processing cells. Using this, a specific instruction can be executed on different data at the same time. Each cell in the Array can process a data word and has its own execution unit and data memory.

In the current implementation, the cells are organized into groups of two, called double-cells. The cells in a double-cell share a decode unit, a register used for data transfer called the I/O register, and a propagation finite state machine.

The I/O registers and the propagation finite state machine are used to transfer data to and from the system that integrates the Accelerator. The I/O registers are organized as a chain of registers through which the data can be shifted in only one direction. Each I/O register stores the data needed for two cells. The propagation finite state machine associates to each double-cell a propagation state: *full* if the double-cell cannot accept data and *empty*, otherwise. The propagation state of each double-cell is computed based on the propagation state of the next and the previous ones.

An overview of the propagation system as well as the memory organization inside the cells and the shift register involved in the data transfer between cells are presented in Figure 3.

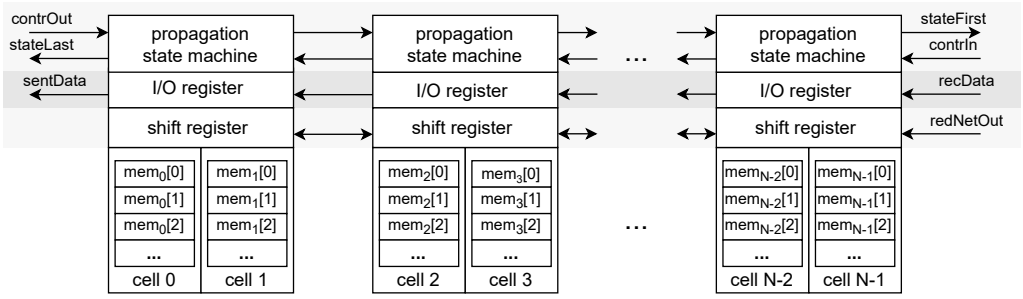

Fig. 3. Data transfer through the cells

The instruction that commands the data propagation through the chain of I/O registers is executed by the Controller. It sends commands on *contrIn* and *contrOut* based on the current states of the first and last double-cells in the propagation chain.

The Array's memory is formed by the data memories in all the cells. The group of data stored at the same address in all the cells in the Array is called a data line. To transfer a data line from the Data Input Path to the Array, the Controller must first shift the data received on *recData* through the chain of I/O registers and, after reaching its final position, send a store command to

the Array. To transfer a data line from the Array to the Data Output Path, the Controller must first send a I/O register load command to the Array, and then shift it out to *sentData*.

The cells in the Array can communicate with each other through a shift register distributed among all the cells.

The Reduction Network processes data organized as a vector and outputs a scalar. It gets the values from the accumulators in all the cells and performs operations such as addition, finding the maximum value, or finding the minimum value. The result of the Reduction Network is then sent back to the Array at the right end of the shift register. In this way, the results from the Reduction Network can reach any cell in the Array.

For running processing tasks on the MapReduce Accelerator, the Host must first load a library of functions into the Controller's program memory. After this initial step, the interaction between the Host and the Accelerator will consist of sending function call commands and their parameters on the Program and Control Path and writing and reading data through the Data Input and Output Paths.

3. Improvements in Data Transfer

Considering that the processes executed on the Accelerator require large amounts of data to be transferred from and to the main memory, the time allocated to the data transfer will represent an important part of the total execution time.

The negative impact of the large time required for data transfer can be mitigated by either increasing the resources allocated to it (e.g., increasing the data interface width), or by transferring new data from the main memory in advance, while the Accelerator is busy processing data already in its internal memory.

For transferring new data while the Accelerator is busy with another processing task, the data transfer and the processing flows must be independent. If this separation is possible, the data can be brought to the Accelerator's memory at the right time, so that the time a processing sequence waits for the input data to be as short as possible.

To achieve the separation of data transfer and processing flows, the two must use different resources with separate control paths.

As mentioned in the previous section, the read and write data transfers imply two main operations: the shift through the chain of I/O registers and the transfer of data between those registers and the Array's memory. Because the data shifting operation uses the chain of I/O registers, which is a storage resource separate from those used for data processing, its independence can be achieved by implementing a module that takes over the shift control from the Controller. Additionally, there must be a direct connection between the I/O registers and the memory in each cell. Since simultaneous requests to access the memory may occur from the data transfer flow through the I/O register and the processing flow through the accumulators in each cell, it is necessary to implement additional arbitration logic.

In the initial design of the MapReduce Accelerator, the data transfer to the Array's memory and the processing are completely sequential: the data that must be processed is transferred, then its processing starts, and only after the current processing is finished can the next data be transferred.

Considering the previous observations, the execution flow in the current design of the MapReduce Accelerator is the one presented in Figure 4. P1, P2, and P3 are processing sequences, and D1, D2, and D3 are the transfer sequences of the corresponding input data. For simplicity, it was

considered that no results are read from the Accelerator and the execution time of each processing sequence is proportional to the amount of input data. As it can be observed, new data can be transferred only after the current processing sequence has finished. If the data transfer and the processing flows become independent, the transfer of new data can start immediately after the previous transfer has finished, regardless of the current processing sequence.

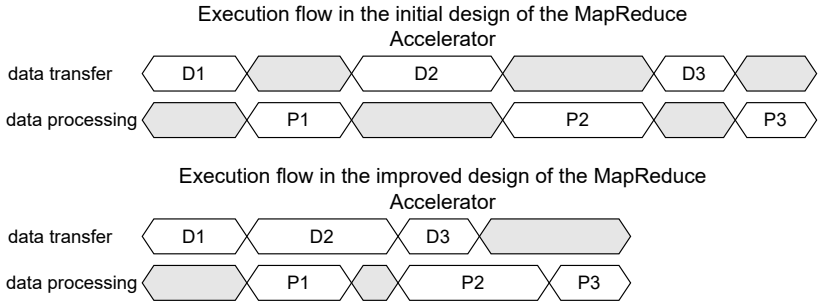

Fig. 4. Execution flows in the initial and improved designs of the MapReduce Accelerator

To achieve the requirements outlined above, a new module called the Data Transfer Engine was implemented. It has the role of coordinating the data transfers between the main memory of the system and the memory of the Accelerator, having a high degree of independence from the current states of the Controller and Parallel Processing Unit. In this way, the two modules can continue the processing of data already present in the internal data memory while data that will be used in subsequent processing is transferred. Considering that the memory accesses initiated by the execution units in each cell have a higher priority, the Data Transfer Engine can access the Array's memory for read and write transfers if the current running instructions do not initiate similar requests. Otherwise, it will wait for the higher priority access to finish.

Two very important observations are that the processing of a data group cannot start until it is completely transferred into the memory and the result cannot be read from the Accelerator until the processing sequence that generated it is finished. This means that, even though they are separate, the data transfer and the processing flows must be synchronized when needed. Moreover, simultaneous requests to access the Array's memory may occur from both the data transfer and processing flows. Because of this, the Accelerator must be able to arbitrate between them, considering the requests from the processing flow to have a higher priority.

The structure of the Accelerator that integrates the Data Transfer Engine is presented in Figure 5. Through its interface, the Data Transfer Engine establishes connections with the other components of the Accelerator and with the modules that ensure the connection with the rest of the heterogeneous system.

As it can be observed in Figure 5, the signals involved in data transfers no longer reach the Controller. The Data Transfer Engine is now communicating directly with the Parallel Processing Unit and the two Data FIFOs.

To start a data transfer, specific commands must be sent by the Host on the Program and Control Path. When such a command is detected by the Data Transfer Engine at the output of the Program FIFO, it is read and saved in its internal command FIFO. In this way, the Program Path will not be blocked if the transfer module is busy.

The entire operation of the Data Transfer Engine is coordinated by a finite state machine. When a data input transfer is performed, a data line is first shifted from the Data Input FIFO

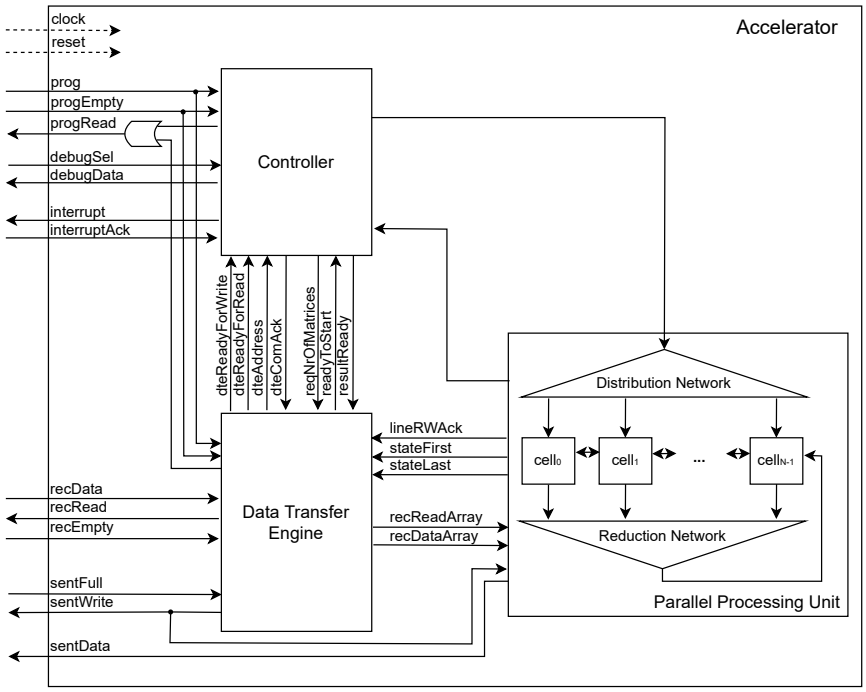

Fig. 5. The structure of the Accelerator integrating the Data Transfer Engine

through the chain of I/O registers. When it reaches its final position, a write request is sent to the Controller on *dteReadyForWrite*, together with the address where the data must be written. An arbitration logic checks if the current information packet prepared to be sent through the Distribution Network to the Array contains memory commands. If not, the request from the Data Transfer Engine is included in the packet and an acknowledgement is sent on *dteComAck*.

The Data Transfer Engine then waits for the commands to pass through the Distribution Network and reach the Array. When the acknowledgement is received on *lineRWAck*, the transfer of the line of data is considered to be complete. The algorithm is then repeated until all the lines of the matrix are transferred.

To read a data line from the Array, the Data Transfer Engine sends first a read request to the Controller. The arbitration logic checks the current information packet prepared for the Array, and includes in it the received request, when possible. The Data Transfer Engine waits for the acknowledgement on *lineRWAck* and then repeats the algorithm for the rest of the data matrix lines.

As mentioned before, the data and the processing flows must be synchronized when needed. To achieve this, two mechanisms were implemented. To ensure that a processing sequence does not begin until the data is transferred, the Controller sends to the Data Transfer Engine the number of required matrices, and the Data Transfer Engine will acknowledge the Controller on *readyToStart* when those matrices are in the Array's memory. To ensure that a processing sequence has finished and its result is valid, the Controller sends an acknowledgement on *resultReady*. After this, the Data Transfer Engine can start the reading process at anytime.

4. Evaluation of Performance Improvement

A basic linear algebra library was implemented to evaluate the performance improvement brought by the Data Transfer Engine. Linear algebra is suitable for parallel systems, as it uses data organized as vectors and matrices. Furthermore, linear algebra is very important for computer science, being used in applications such as machine learning, video processing, or cryptography.

The library contains the following functions:

- Start cycle counter
  START_CC(): starts the cycle counter in the Controller
- Stop cycle counter
  STOP_CC(): stops the cycle counter in the Controller.
- Send interrupt
  SEND_INT(): sets the Accelerator's interrupt flag. This can be used to signal the end of a task.
- WAIT_RES_READY(): forces the Data Transfer Engine to wait for a processing sequence to finish before starting a new transfer. This is used to synchronize the processing and data transfer flows by not allowing the Data Transfer Engine to read a result from the Array's memory before the Controller marks it as valid.
- Matrix-Matrix element-wise operation
  MM_EWO(*dest, source1, source2, linesNr, operation, waitMatricesNr*):
  performs element-wise operation between two matrices stored in memory starting at addresses *source1* and *source2*. The resulting matrix is stored starting at address *dest*. All the matrices are $linesNr \times N$, where *linesNr* is a parameter of the function and N is the number of cells in the Array. The *operation* parameter can have the following values: ADD, SUB, MULT, AND, OR, and XOR. The execution of this function starts only after *waitMatricesNr* matrices are written in the Array's memory.
- Matrix-Matrix multiplication
  MM_MULT(*dest, source1, source2, linesNr, waitMatricesNr*):
  performs the multiplication between the two matrices stored in memory starting at addresses *source1* and *source2*. The resulting matrix is stored starting at address *dest*. An important observation is that the Accelerator does not perform the transpose operation for the second matrix, considering it as already transposed by the Host. All the matrices are $linesNr \times linesNr$ square matrices. Nevertheless, it is recommended that *linesNr* be equal to N (the number of cells in the Array). The execution of this function starts only after *waitMatricesNr* matrices are written in the Array's memory.
- Matrix-Matrix multiplication and accumulation
  MM_MAC(*dest, source1, source2, linesNr, waitMatricesNr*):
  performs the multiplication between two matrices stored in memory starting at addresses *source1* and *source2* and accumulates the result with the matrix stored in memory starting at address *dest*. The new resulting matrix is stored starting at address *dest*. As in the case of Matrix-Matrix Multiplication, the Accelerator does not perform the transpose operation

for the second matrix, considering it as already transposed by the Host. All the matrices are *linesNr* $\times$ *linesNr* square matrices. Nevertheless, it is recommended that *linesNr* be equal to N (the number of cells in the Array). The execution of this function starts only after *waitMatricesNr* matrices are written in the Array's memory.

In addition to the previously described functions, which will be loaded into the program memory, the following transfer call commands were implemented to allow the matrix transfer between the Array and the two FIFOs on the Data Path: WRITE_MATRIX(*dest, linesNr, columnsNr*), which will be used to transfer a *linesNr* $\times$ *columnsNr* matrix from the Data Input FIFO to the Array's memory, and READ_MATRIX(*source, linesNr, columnsNr*), which will be used to transfer a *linesNr* $\times$ *columnsNr* matrix from the Array's memory to the Data Output FIFO.

If only one call of the above functions is performed, no improvements in the total execution time will be observed, as it must wait for the input matrices to be written in the memory (case D1-P1 in Figure 4). The influence of the Data Transfer Engine is observed only for at least two consecutive calls, and only if the data for those calls is stored in different memory areas.

Considering the previous observations, two test algorithms were prepared: Matrix-Matrix element-wise operation and Matrix-Matrix multiplication for large matrices. A large matrix is considered to be a matrix whose dimensions are larger than the number of cells in the Array. For simplicity, those matrices were considered to be squared, with dimensions equal to $2^x \cdot N$, where N is the number of cells and x is an integer number. Both algorithms are based on the assumption that we can split the large matrices into multiple $N \times N$ matrices, as shown in Figure 6. The smaller matrices will be processed by the Accelerator, obtaining $N \times N$ segments of the result matrix. The large resulting matrix will then be reconstructed based on these segments.

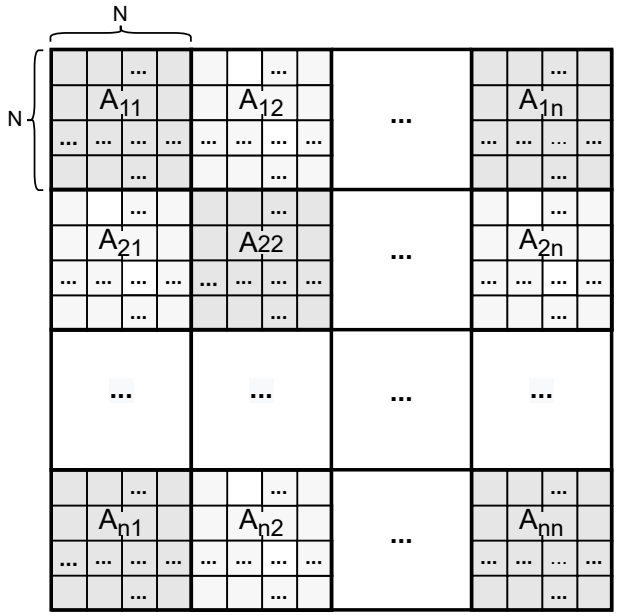

Fig. 6. Dividing large matrices into smaller, $N \times N$ matrices

Considering the previous observations, if A and B are large matrices, the Matrix-Matrix

element-wise operation is defined by (1):

$$\begin{bmatrix} R_{11} & R_{12} & ... & R_{1n} \\ R_{21} & R_{22} & ... & R_{2n} \\ ... & ... & ... & ... \\ R_{n1} & R_{n2} & ... & R_{nn} \end{bmatrix} = \begin{bmatrix} A_{11} \, o \, B_{11} & A_{12} \, o \, B_{12} & ... & A_{1n} \, o \, B_{1n} \\ A_{21} \, o \, B_{21} & A_{22} \, o \, B_{22} & ... & A_{2n} \, o \, B_{2n} \\ ... & ... & ... & ... \\ A_{n1} \, o \, B_{n1} & A_{n2} \, o \, B_{n2} & ... & A_{nn} \, o \, B_{nn} \end{bmatrix}, \quad (1)$$

where R_{ij}, A_{ij}, and B_{ij} are $N \times N$ matrices, components of the large matrices R, A and B.

The Matrix-Matrix multiplication is defined by (2). For multiplication, it is considered that the transpose of matrix B (B^t) has already been computed.

$$\begin{bmatrix} R_{11} & ... \\ R_{21} & ... \\ ... & ... \\ R_{n1} & ... \end{bmatrix} = \begin{bmatrix} A_{11} \times B^t_{11} + A_{12} \times B^t_{12} + ... + A_{1n} \times B^t_{1n} & ... \\ A_{21} \times B^t_{11} + A_{22} \times B^t_{12} + ... + A_{2n} \times B^t_{1n} & ... \\ ... & ... \\ A_{n1} \times B^t_{11} + A_{n2} \times B^t_{12} + ... + A_{nn} \times B^t_{1n} & ... \end{bmatrix}, \quad (2)$$

where R_{ij}, A_{ij}, and B^t_{ij} are $N \times N$ matrices, components of the large matrices R, A and B^t. B^t is the transpose of matrix B.

The two algorithms, as they were implemented for the target heterogeneous computing system using the functions described in the basic linear algebra library, are presented in Figure 7 and Figure 8. The Matrix-Matrix element-wise operation has been particularized for ADD.

```
for(i = 1; i <= n; i = i + 1) {
  for(j = 1; j <= n; j = j + 2) {
    WRITE_MATRIX(A_ij, N, N)
    WRITE_MATRIX(B_ij, N, N)
    WRITE_MATRIX(A_i(j+1), N, N)
    WRITE_MATRIX(B_i(j+1), N, N)
    MM_EWO(R_ij, A_ij, B_ij, N, ADD, 2)
    WAIT_RES_READY()
    READ_MATRIX(R_ij, N, N)
    MM_EWO(R_i(j+1), A_i(j+1), B_i(j+1), N, ADD, 2)
    WAIT_RES_READY()
    READ_MATRIX(R_i(j+1), N, N)
  }
}
```

Fig. 7. The algorithm for large matrices ADD

In both cases, the data memory from each cell of the Array is divided into two separate sections that are used successively. Each section stores the operands and the result of an operation. In this way, two sets of operands can be transferred and processed before reading the results, without data overlapping.

At each step of the algorithm, the data transfer commands are placed first to avoid blocking the Program Path. Behind this reasoning is the existence of the FIFO memory in the Data Transfer Engine, which can accept data transfer commands even though the module is currently busy running a transfer task. In this way, the output of the Program FIFO is read and the Program and Control Path outputs the next command. On the other hand, if a command for the Controller is available at the output of the Program FIFO and the Controller is busy running another task, it cannot be accepted, leading to the blocking of the Program and Control Path.

In the algorithm used for adding large matrices described in Figure 7, two pairs of $N \times N$ matrices from each large matrix are transmitted at each step. The matrices in each pair occupy

the same position in the two large source matrices (e.g., A_{11} and B_{11} form a pair), and the result of their processing represents a segment from the resulting large matrix.

The algorithm used for the multiplication of large matrices also uses the division of memory into two distinct areas for storing pairs of operands, but this time the result area is shared. Since the result is not read after each MULT or MAC operation, overwriting the memory area where operands were stored will be done only after the task using the old operands is complete and its result is valid.

```
for(i = 1; i <= n; i = i + 1) {
  for(k = 1; k <= n; k = k + 1) {
    WRITE_MATRIX(A_i1, N, N)
    WRITE_MATRIX(B^t_k1, N, N)
    WRITE_MATRIX(A_i2, N, N)
    WRITE_MATRIX(B^t_k2, N, N)
    MM_MULT(R_ik, A_i1, B^t_k1, N, 2)
    MM_MAC(R_ik, A_i2, B^t_k2, N, 2)
    if(n - 2 >= 2) {
      for(j = 1; j <= n - 2; j = j + 2) {
        WAIT_RES_READY()
        WRITE_MATRIX(A_ij, N, N)
        WRITE_MATRIX(B^t_kj, N, N)
        WAIT_RES_READY()
        WRITE_MATRIX(A_i(j+1), N, N)
        WRITE_MATRIX(B^t_k(j+1), N, N)
        MM_MAC(R_ik, A_ij, B^t_kj, N, 2)
        MM_MAC(R_ik, A_i(j+1), B^t_k(j+1), N, 2)
      }
    }
    WAIT_RES_READY()
    WAIT_RES_READY()
    READ_MATRIX(R_ik, N, N)
  }
}
```

Fig. 8. The algorithm for large matrices multiplication

The two algorithms were executed for a preliminary characterization of the system performance when working with large matrices. A simulation environment was used to interact with the Accelerator by transmitting to its inputs the program and the required data and by reading the results when ready. The performance was measured in *number of clock cycles* necessary to execute the task. In order to have a clearer picture of the Accelerator's performance, the starting point for measuring the number of clock cycles was considered to be the moment when the library is already in the program memory and the first data reaches the input of the Accelerator.

The variation in the number of clock cycles required for adding and multiplying 16 $\times$ 16, 32 $\times$ 32, 64 $\times$ 64, and 128 $\times$ 128 matrices on an Accelerator having 16 processing cells is shown in Figure 9. As expected, the multiplication takes much longer than the addition. Its execution time increases much faster with the increase in the size of the matrices. From the point of view of the improvement in performance brought by the Data Transfer Engine (Improved MRA), a decrease in the total execution time is observed for both algorithms. In all 3 cases of adding matrices larger than the 16-cell Array, an approximate improvement of 35% was obtained. In the case of multiplication, this decrease is smaller (up to 22%) compared to the execution time on

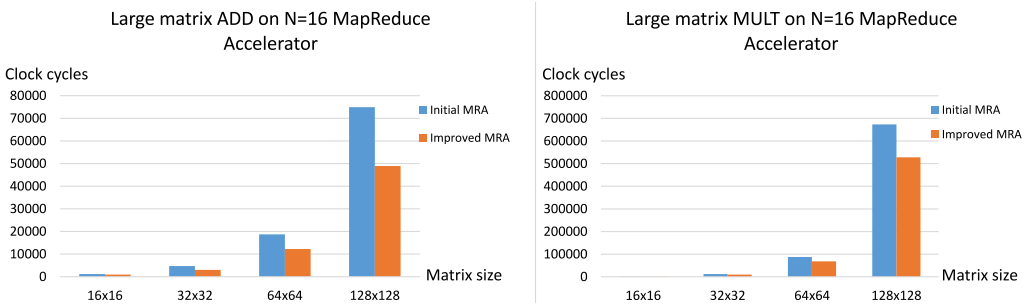

Fig. 9. Large matrix ADD and MULT on N = 16 MapReduce Accelerator

an Accelerator without Data Transfer Engine. This can be explained by the fact that, during the multiplication, the memory is accessed more often and the requests with a lower priority from the Data Transfer Engine are accepted with a delay.

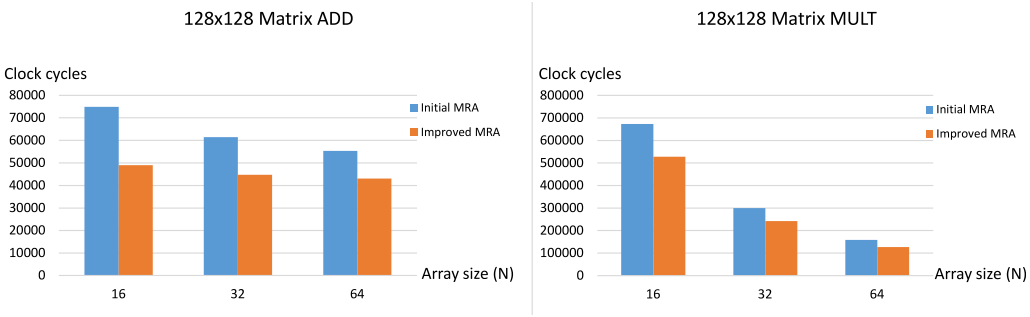

Fig. 10. 128x128 Matrix ADD and MULT on MapReduce Accelerator

Figure 10 shows the execution times of adding and multiplying two 128 $\times$ 128 matrices on Accelerators with 16, 32, and 64 processing cells. It can be seen that the performance improvement becomes more significant as the number of cells decreases, especially in the case of the ADD operation: up to 35% improvement of execution time for the Accelerator with 16 cells. Moreover, it can be observed that in the case of the addition, the Data Transfer Engine (Improved MRA) leads to the mitigation of the increase in the execution time with the decrease in the number of processing cells.

5. Conclusions

The proposed solution for improving the execution time for a MapReduce Accelerator by separating the data transfer and processing flows with the help of a Data Transfer Engine provides good preliminary results.

The proposed module was designed by taking into account the particularities of the target Accelerator, especially those related to data transfer. It has been integrated into the structure of the MapReduce Accelerator, its operation being synchronized with the other modules when necessary.

For performance evaluation, two versions of the Accelerator were used: the initial version and the improved version, which integrates the Data Transfer Engine. On both architectures, two algorithms that process matrices whose dimensions exceed the size of the Array of processing cells were executed: large matrices element-wise addition and large matrices multiplication. In addition to the picture of the influence of the Data Transfer Engine on the execution time, the results also showed how it varies with the number of processing cells in the Array. In all tests, the execution time was improved, its magnitude being influenced both by the resources of the Accelerator and by the frequency of memory accesses requested by the processing flow. The improvement in execution time was up to 35%, obtained for the operation of adding large matrices on an Accelerator with 16 processing cells.

References

[1] A. DURAN and M. KLEMM, *The Intel® Many Integrated Core Architecture*, Proceedings of 2012 International Conference on High Performance Computing & Simulation, Madrid, Spain, pp. 365–366, 2012.

[2] *Intel® Xeon Phi™ Coprocessor System Software Developers Guide*, Intel Corporation, 2014. Available at: https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-coprocessor-system-software-developers-guide.pdf.

[3] *NVIDIA TESLA V100 GPU Architecture*, NVIDIA Corporation, 2017. Available at: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[4] *NVIDIA TURING GPU Architecture*, NVIDIA Corporation, 2018. Available at: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[5] *NVIDIA A100 Tensor Core GPU Architecture*, NVIDIA Corporation, 2020. Available at: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[6] *NVIDIA H100 Tensor Core GPU Architecture*, NVIDIA Corporation, 2022. Available at: https://resources.nvidia.com/en-us-tensor-core.

[7] N. P. JOUPPI, D. H. YOON, G. KURIAN, S. LI, N. PATIL, J. LAUDON, C. YOUNG and D. PATTERSON, *A domain-specific supercomputer for training deep neural networks*, Communications of the ACM, **63**(7), pp. 67—78, 2020.

[8] M. MALIȚA, G. V. POPESCU and G. ȘTEFAN, *Heterogenous computing system for deep learning*, in Deep Learning: Concepts and Architectures, W. Pedrycz and S.-M. Chen, Eds., Springer, Cham, Studies in Computational Intelligence **866**, pp. 287—319, 2019.

[9] M. MALIȚA, G. V. POPESCU and G. M. ȘTEFAN, *Heterogeneous computing for Markov models in Big Data*, Proceedings of 2019 International Conference on Computational Science and Computational Intelligence, Las Vegas, NV, USA, pp. 1500–1505, 2019.

[10] M. MALIȚA, G. V. POPESCU and G. M. ȘTEFAN, *Pseudo-reconfigurable heterogeneous solution for accelerating spectral clustering*, Proceedings of 2020 IEEE International Conference on Big Data, Atlanta, GA, USA, pp. 5138–5145, 2020.

[11] G. V. POPESCU and C. BÎRĂ, *Python-based programming framework for a heterogeneous MapReduce architecture*, Proceedings of 2022 14th International Conference on Communications, Bucharest, Romania, pp. 1-6, 2022.