Using High-Level Synthesis to Implement the Matrix-Vector Multiplication on FPGA

Alessandro Marongiu, Paolo Palazzari

ENEA - Italian National Agency for New Technologies, Energy and Sustainable Economic Development
C.R. ENEA Casaccia, Rome – Italy

Abstract. This work presents how to implement the Matrix-Vector Multiplication (MVM) onto FPGA through the QuickPlay High-Level Synthesis flow. The motivations arise from the Adaptive Optics field, where the MVM is the core of the real-time control algorithm which controls the mirrors of a telescope to compensate for the effects of the atmospheric turbulence. The proposed implementation of the MVM exploits four different levels of parallelism: spatial and pipeline parallelism are used both at the fine (scalar instructions) and at the coarse (vector instructions) levels. To characterize the architecture being developed, a performance model has been developed and validated through the actual results obtained from runs on a prototype board based on the Intel ARRIA10 FPGA. Some details are given to describe how the algorithm has been implemented using the QuickPlay HLS flow. Performance results are presented, in terms of sustained computational speed and resources used in the hardware implementation.

1 Introduction

In the framework of the research project Green Flash [1], we developed the work presented in this paper, aimed at efficiently implementing the Matrix-Vector Multiplication (MVM) on the FPGA technology. As discussed in [2], [3], [4] and [5], in Adaptive Optics (AO) the effect of the atmospheric turbulence is compensated using the mobile mirrors in the telescope, which are moved according to a given real-time control algorithm. The dominating part of such an algorithm, see technical annex of the project [1], is the execution of two MVMs, namely $s_k^*=Mv_k$ and $w_k=Rs_k^{pol}$.

In this paper we illustrate how we used QuickPlay [6], a High-Level Synthesis (HLS) design flow, to efficiently implement the MVM on FPGA. Representing one of the Level-2 BLAS functions [7], MVM is the basis for many algebraic computations and it is fundamental in many application domains. We underline that we see the presented work as a template of the methodology to be adopted when using HLS.

We start describing the problem to be solved, together with the constraints imposed by the challenges on the architecture to be implemented. Next, we present the formulation of the solution, explaining how parallelism should be exploited to obtain an efficient implementation. The implementation we propose in this paper uses four

levels of parallelism: as the MVM is a collection of many independent scalar products, we introduce pipeline and spatial parallelism both at the coarse level (parallelization among different scalar products) and at the fine level (parallelization within the computation of one scalar product). A performance model is derived to quantify the performance achievable through the proposed implementation: this phase is crucial to validate the performance of the HLS. When using HLS, it is crucial the preliminary determination of what can be achieved, checking after the synthesis that the results produced by the automated synthesis process comply with expectations: in lack of this modeling phase, we should rely only on comparisons with other implementations to (indirectly) evaluate the implementation produced by HLS. In this paper, the emphasis is put mainly on the evaluation of the quality of the implementation derived from the HLS flow, as we are not trying to assess the superiority of a given technology against another: discussing FPGA vs GPU is not the aim of this paper. For this reason, we put much effort into the modeling of the performance which can be theoretically achieved, to have an absolute criterion to evaluate the quality of the FPGA implementation: the closer is the performance to the theoretical forecast, the better it is.

The document is concluded with the presentation of the results, in terms of performance achieved in actual runs (GFlop/s) and resource used (LUT, memory blocks, DSP).

2 Related Work

Due to its relevance in many domains, the implementation of the MVM has been widely investigated; in particular, how to efficiently implement the operation on the FPGA technology has been investigated. In [8] the authors present a comparison of the implementation of the MVM, the gaxpy operation, on FPGA, GPU and CPU. They describe the FPGA implementation, organizing the internal dual-ported memories as V row banks which store the rows of the matrix; each of these banks is composed by B banks which store in an interleaved way the rows mapped into the row bank; thanks to this organization, at each clock cycle VxB elements can be read and written from and to the memory. These elements can feed Q≤V pipelined modules, each one computing a B-size scalar product. The work is further improved in [9], where the management of large external memory is added. In [10], [11] the FPGA implementation of the BLAS operations is discussed, with a special focus on the implementation of the reduction circuit needed in the accumulation involved in each BLAS operation. The authors in [12] report the FPGA implementation of the MVM and matrix-matrix product with a detailed analysis of the error propagation in the accumulation phase. Considering that the MVM problem is I/O bound and there is no benefit in increasing the parallelism beyond the I/O saturation, the authors propose to use some logic to implement the group-alignment based floating-point summation [13], which increases the numerical accuracy of the computation. The FPGA implementation of the BLAS is reported in [14]. In this work, while relying on the OpenCL framework [15] for the actual FPGA implementation, the authors give a detailed performance model to drive the selection of the parameters determining the tradeoff be-

tween speed and resource performance. Using the selected parameters, some code generators are activated to generate the OpenCL description of the optimized BLAS routine. The reader interested in the implementation of the MVM on GPU technology can refer to [16], which presents an analysis of the MVM implementation on GPU, together with a detailed performance model.

3 Problem Definition

The MVM is the basic operation to perform the Wavefront Reconstruction control algorithm; its usage is well known in the Adaptive Optics community and dates back to the late '80s [17] and has been successively improved many times [2]. In our implementation, using single-precision floating-point arithmetic, we have to multiply two matrices $\mathbf{M}[N_{means},N_{rec}]$ and $\mathbf{R}[N_{rec},N_{means}]$ with the vectors $\mathbf{v_k}[N_{rec}]$, $\mathbf{s_k}[N_{means}]$, being N_{means}=9232 and N_{rec}=6316.

Due to their size, **M** and **R** are stored in external memory. **M** and **R** do not change for a quite long time and must be multiplied many times by vectors $\mathbf{v_k}$ and $\mathbf{s_k}$; processing step (k+1) can start only when the k^{th} step has finished.

Once the bandwidth BW to access the external memory is fixed, an upper bound for the speed of the computation is determined. To perform the MVM, the matrix must be read from the memory; when we refer to a generic matrix M[n,m] and we indicate with D the floating-point data size expressed in bytes (in single-precision D=4, in double-precision D=8), the matrix size is M_s=nmD [Bytes] and the time to read the matrix from external memory is

$$t_R = nmD/BW. \quad (1)$$

As the number of operations performed in the MVM is n_{ops}=2nm and the overall computing time cannot be smaller than t_R, the computing speed S_C cannot be larger than n_{ops}/t_R i.e.,

$$S_C \leq \frac{n_{ops}}{t_R} = \frac{2nm}{\frac{nmD}{BW}} = \frac{2BW}{D}. \quad (2)$$

Using single-precision floating-point, D=4, the speed can never be greater than half of the available memory BW.

In the following sections, we will analyze how the MVM should be implemented to be as close as possible to the previous limit.

4 Guidelines for Implementation: Exploiting Coarse-Grained Parallelism

The MVM **b** = **M** x **a** (**M**[n,m], **a**[m], **b**[n]) is the collection of n independent scalar products between m-sized vectors i.e.,

$$b_i = \mathbf{m_i} \cdot \mathbf{a} \; i=0,1,\ldots,n-1; \; b_i \in \mathbb{R}; \; \mathbf{m_i} \in \mathbb{R}^m; \; \mathbf{a} \in \mathbb{R}^m. \quad (3)$$

Let's implement, in an optimized way, a kernel SP which performs a certain number of scalar products between one vector **a** and several vectors read from the external memory; if we have p external memory banks, we can partition[1] **M** in p equal parts $\mathbf{M_p}$, each containing n/p different matrix lines $\mathbf{m_{p,i}}$ with p = 0,1,..., p-1 and i=0,1,...,n/p-1 (each line is an m-sized vector), storing each $\mathbf{M_p}$ into a different memory bank. We instantiate p replicas of the SP scalar product kernel and we distribute a copy of the **a** vector, to be read once, to all the SP kernels. Each SP kernel computes a portion $\mathbf{b_p}$ of the **b** result vector. The final vector is obtained properly merging (i.e., concatenating) all the $\mathbf{b_p}$ sub-vectors.

The degree of parallelism p is selected to make (nearly) equal the BW requirement with the BW available toward the external memory (BW_{ExtMem}); let's indicate with BW_{req} the memory bandwidth requested by the SP kernel (BW_{req} will be quantified in the following).

The memory bandwidth required by the p SP kernels is $p \times BW_{req}$ and must be large enough to saturate BW_{ExtMem} i.e., $BW_{ExtMem} \approx p \times BW_{req}$ which gives

$$p \approx BW_{ExtMem}/BW_{req}. \quad (4)$$

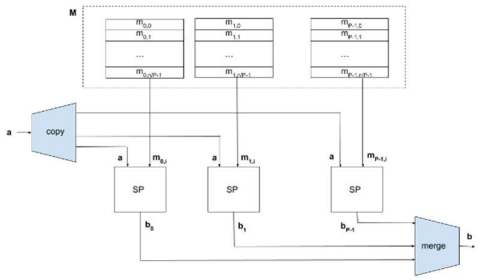

Fig. 1. coarse-grained parallel architecture to implement the MVM

In the following, when giving numerical examples, we use the parameters characterizing the μXComp board, developed by MicroGate and equipped with an Intel ARRIA 10 GX1150 FPGA [18]. Referring to the previous example and to the four Hyper Memory Cube (HMC) banks present in the μXComp board (each HMC bank has a peak BW of 17 GB/s), BW_{ExtMem}=68 GB/s. In our implementation of the SP kernel BW_{req} = 19.2 GB/s, so the degree of parallelism that can be efficiently supported is given by eq. (4) which yields $p \approx 4$. Therefore, four SP kernels can be instantiated, each one accessing a different HMC bank.

[1] Let's assume n to be multiple of p; should this not being the case, (n%p) sets would have $\lfloor n/p \rfloor + 1$ lines and the remaining sets would contain $\lfloor n/p \rfloor$ lines

5 The Scalar Product: Basic Pipelined Implementation

As a consequence of the discussion of the previous section, we recognize the scalar product as our coarse grain unit of parallelism. The scalar product can be implemented with one pipelined MADD (one multiplier and one adder) which iteratively computes the recurrence

$s_{i+1} = a_i \times b_i + s_i \qquad i=0, ..., n-1$ with $s_0=0$, $a_i \in \mathbf{a}$, $b_i \in \mathbf{b}$.

The computation of the next MADD operation is dependent on the completion of the previous operation, so a new MADD cannot start until the previous has finished, thus waiting for the latency L of the MADD.

To avoid paying this penalty, we can exploit the commutativity and associativity of the ADD operation (let us neglect the effects of the limited precision). Under the commutative and associative hypothesis for the ADD and assuming m to be an integer multiple of L, we can rewrite the scalar product as in the following

$$s = \sum_{i=0}^{m-1}(a_i \cdot b_i) = \sum_{i=0}^{L-1}\left(\sum_{j=0}^{\frac{m}{L}-1} a_{jL+i} \cdot b_{jL+i}\right) \tag{5}$$

where

- vectors **a** and **b** have been partitioned into L sub-vectors $\mathbf{a_i}$ and $\mathbf{b_i}$,
- L partial scalar products are computed (expression in brackets) and finally
- the result is derived by summing the L partial scalar products (external sum).

In the previous formulation, each partial scalar product has to be updated every L clock cycles; during its processing (requiring L cycles), the other L-1 partial scalar products will be processed, each one being at a different stage of the pipeline. Only the final (i.e., the external) sum requires the accumulation of values where the dependence cannot be completely hidden, thus imposing the payment of some pipeline penalty.

Following the previous approach, we can compute the scalar product in N_{clk} clock cycles, as follows

$$N_{clk} = (m-1) + L + O(L_A*\log(L)) \tag{6}$$

where (m-1)+L, are the cycles needed to compute the m MADD operations and $O(L_A*\log(L))$ are the cycles needed to perform the final sum of the L partial scalar products (L_A is the latency of the pipelined add operator) using L/2 adders; if m>>L, $N_{clk} \approx m$. In our case m>>L, so we compute the 2m operations required by the scalar product in $N_{clk} \approx m$ clock cycles, thus sustaining 2 FP operations per cycle. The sustained speed of the computation is $S_C=2f_{ck}=300$ MFlop/s for $f_{ck}=150$ MHz.

As seen in the previous section, to sustain the speed of the computation S_C we must have a BW toward the memory which is at least twice the numerical value of S_C (eq.2)). In this case, the memory BW required by the kernel would be $BW_{req} = 2*S_C = 2*300 = 600$ MB/s. Referring to the BW of the HMC memory we are using (≈68 GB/s), to saturate the memory BW we should put p=68/0.6=112 kernels in parallel, which would require 112 ports to access the external memory module: this huge num-

ber of ports is not realistic, so we have to find a way to increase the computational speed of the kernel which performs the basic scalar product, in order to use, with the BW_{req} of a single kernel, a significant portion of the available memory BW.

6 The Scalar Product: Exploiting Spatial Parallelism

To increase the computational speed and the BW_{req} of the kernel which computes the scalar product, we could further partition each of the L sub-vectors into P sub-vectors so that, at each cycle, we can start computing P independent partial scalar products.

Let's rewrite the equation (5) as in the following

$$s = \sum_{i=0}^{m-1}(a_i \cdot b_i) = \sum_{i=0}^{L-1}\sum_{j=0}^{P-1}\left(\sum_{k=0}^{\frac{m}{LP}-1} a_{iP+j+kLP} \cdot b_{iP+j+kLP}\right) \quad (7)$$

where vectors **a** and **b** have been partitioned into LP sub-vectors, each with m/(LP) elements; the generic sub-vector $\mathbf{v_{ij}}$ is defined as

$$\mathbf{v_{ij}} = \{v_{iP+j+kLP} \mid k = 0,1,...,m/LP\} \quad i=0,1,...,L-1 \quad j = 0,1,...,P-1.$$

Once partitioned **a** and **b** into the LP sub-vectors $\mathbf{a_{ij}}$ and $\mathbf{b_{ij}}$, we compute the LP partial scalar products s_{ij} (expression in brackets in (7)), then we sum all the LP partial values to obtain the final result.

Using P MADDs, if we can read 2P floating-point values per cycle, the number of cycles to determine the LP partial scalar products is given by

$$N_{comp} = \left[\left(\frac{m}{P} - 1\right) + L\right]. \quad (8)$$

In fact, after L clock cycles, P MADD results are produced; the remaining (m-P) MADD results are produced in the following (m-P)/P cycles, as P new results are produced at every cycle.

Once generated the N = LP s_{ij} values, they must be summed together to obtain the final scalar product.

As already discussed, we can use N/2 adders to perform the sum of N numbers in $\lceil log_2 N \rceil L_A$ clock cycles. If we use P_A <N adders, in each layer we can parallelize the sums among all the P_A adders. It's easy to verify that the number of cycles to compute the sum of N=LP numbers using P_A pipelined adders is given by

$$NCycles_{sum}(P_A) = \sum_{i=1}^{\lceil \log_2(N) \rceil}\left(\left\lceil \frac{N}{2^i}\frac{1}{P_A}\right\rceil + L_A\right) \approx \frac{N}{P_A} + \lceil \log_2(N) \rceil L_A. \quad (9)$$

The number of cycles $NCycles_{SP}$ necessary to compute the scalar product of two vectors of size *m* using P pipelined MADD modules, with latency L, and P_A pipelined adders, with latency L_A, is given by

$$NCycles_{SP} = N_{comp} + NCycles_{sum}(P_A). \quad (10)$$

From (8), (9) and (10) we get

$$\mathrm{NCycles_{SP}} \approx \frac{m}{P} + L + \frac{LP}{P_A} + \lceil \log_2(LP) \rceil L_A. \quad (11)$$

From the previous expression, we can compute the sustained speed of the computation (expressed in operations/cycle) as

$$\mathrm{SustainedSpeed} = \frac{2m}{\frac{m}{P}+L+\frac{LP}{P_A}+\lceil \log_2(LP) \rceil L_A}. \quad (12)$$

In previous equation L and L_A are fixed by the technology (for instance, with the current version of QuickCompiler and for the ARRIA10 FPGA, L=8 and L_A=3), *m* is fixed by the problem, P and P_A are the parameters of the architecture that must be determined to maximize the sustained speed.

P must satisfy the following requirements:

- must be a power of 2, i.e. $P = 2^k$, because it determines the width of the internal memory used by the SP kernel (width of the memory must be a power of 2),
- must be large enough to nearly saturate the memory BW.

In our example, f_{ck}=150 MHz and the BW to one bank of the HMC memory is 17 GB/s. Thus, the width W to saturate the BW is given by

$$W * f_{ck} = BW\ [Byte/s] \Rightarrow \quad W = BW/f_{ck}\ [Byte]$$

which gives W = 17000/150 = 113 [Byte]. As W has to be a power of 2, we can set W=128 [Byte] (the closest to 113), thus fixing the MADD parallelism to 32 (32 MADDs must read 64 floats/cycle; 32 floats come from the buffer memory connected to the HMC and storing a row of the matrix **M** and 32 floats come from the buffer memory connected to the input stream and storing the vector **a**, read only once at the very beginning).

When P = 32 and m=8K elements, the number of cycles necessary to compute the LP partial products s_{ij} is (ref. to eq.(8))

$$\mathrm{NCycles_{comp}} = (m/P) - 1 + L = (8192/32) - 1 + 8 = 263.$$

If we set P_A=4 (adder parallelism), the number of cycles to sum all the partial results is (ref. to eq. (9))

$$\mathrm{NCycles_{sum}}(P_A) \approx \frac{LP}{P_A} + \lceil \log_2(LP) \rceil L_A = \frac{8 \cdot 32}{4} + 8 \cdot 3 = 88.$$

With the previous values, the equation (9) gives a Sustained Speed of 46.7 operations/cycle; as f_{ck}=150MHz, the previous figure corresponds to

$$46.7[ops/cycle] * 150[MHz] = 7.0\ [GFlop/s].$$

7 MVM: Coarse-Grained Pipelining

In the operation **b** = **M** x **a**, the result vector **b** can be computed through the following loop

```
for (l=0; l<n;l++)
  b_l=m_l·a;  // m_l is the l-th row of M
```

whose body can be decomposed in three basic operations:

```
for (l=0; l<n; l++){
  load m_l from the external memory
  compute the LP partial scalar products s_ij
  compute the final result b_l = Σ_i,j (s_ij)
}
```

The loop can be repeated in different kernels when the matrix M is partitioned into *p* submatrices, as depicted in **Fig. 1**.

Regarding the time complexity (expressed in number of clock cycles), we can write the following relations

- moving 4m bytes from the external memory, accessible through a port with W=4P bytes, to the internal multi-ported memory requires the number of cycles

$$N_{mem} = \frac{m}{P} + L_m \tag{13}$$

as the internal memory can accept 4P bytes/cycle; L_m is the latency to access the external memory; if $Wf_{ck}=BW_{req}>BW_{ExtMem}$, the actual number of cycles will be larger than N_{mem} because the required bandwidth Wf_{ck} is larger than the available memory bandwidth;

- the number of cycles required to compute the LP partial scalar products is given by eq. (8):
- the sum of LP values using P_A floating-point adders (with latency L_A) requires the number of clock cycles $N_{Sum}(P_A)$ given by eq. (9).

As the iterations of the loop are independent, the loop can be pipelined, at a coarse grain, with three pipeline stages:

- load vector m_i,
- compute the LP partial scalar products s_{ij},
- sum the LP s_{ij}.

The duration of each stage of this "macro-pipeline" is given by

$$N_{PipeStage} = \max\left(N_{mem}, N_{comp}, N_{SUM}(P_A)\right). \tag{14}$$

Being the loop fully pipelined, n+2 "macro-pipeline" stages are required to process n matrix lines and to compute n scalar products. The number of cycles necessary to compute the whole MVM, using *p* equal SP kernels, is given by

$$N_{Total} = \left(\frac{n}{p} + 2\right) N_{PipeStage}. \quad (15)$$

The sustained speed (operations/cycle) is given by the ratio

$$S = \frac{N_{operations}}{N_{Total}} = \frac{2nm}{\left(\frac{n}{p}+2\right)N_{PipeStage}}. \quad (16)$$

Let's consider the case characterized by the following parameters:

- m=n=8192 (m : size of the vector, n: number of scalar products to be computed)
- L_A=3, L= 8 and L_m= 200 cycles (latencies of FP adder, MADD and HMC)
- P=32 (spatial parallelism, i.e., number of MADD operations performed in parallel)
- P_A=1 (1 adder is used to sum the LP partial scalar products)
- *p*=2 (kernel parallelism, i.e., number of equal kernels, each one performing the scalar product)

Previous values, when inserted in the expressions derived above, give the following values:

- N_{mem}= m/P+L_m = 456 ,
- N_{comp} = m/P+L_{MADD} = 264 ,
- $N_{SUM}(P_A) \approx \frac{LP}{P_A} + \lceil log_2(LP) \rceil L_A = 280.$

So $N_{PipeStage}$=456 and the sustained speed, when f_{ck}=150 MHz, is

$$S = \frac{N_{operations}}{N_{Total}} = \frac{2nm}{\left(\frac{n}{p} + 2\right) N_{PipeStage}} = 71.82 \left[\frac{ops}{cycle}\right] = 10.8 \left[\frac{GFlop}{s}\right].$$

It's worth to be underlined that, when we ran on the μXComp board the test developed using previous values, we measured an overall speed of 10.6 [GFlop/s], in perfect agreement with the performance foreseen by the model (see **Table 1**, reported in the section related to performance).

8 FPGA Implementation of the MVM Through the QuickPlay HLS

In this section, we analyze the actual FPGA implementation of the MVM algorithm, based on the considerations illustrated in the previous sections.

To achieve the FPGA implementation, we use the Accelize HLS framework (QuickPlay with its embedded QuickCompiler HLS engine [6], formerly produced by Accelize and to be shortly released as Open Source SW).

We refer to the architecture depicted in **Fig. 1** and, in the following **Fig. 2**, we report the QuickPlay schematic representing that architecture, in the case of *p*=4 SP kernels.

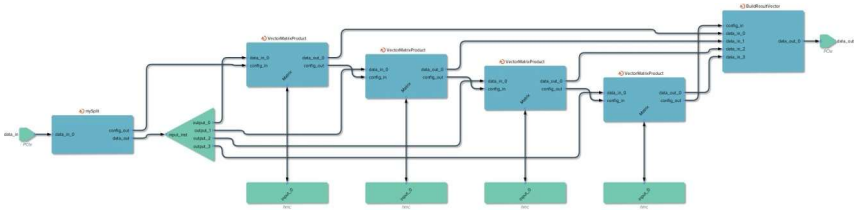

Fig. 2. top level of the design with p=4 SP kernels, as shown in the QuickPlay VisualEditor

In the previous design, we can recognize the four VectorMatrixProduct kernels, each performing n/4 scalar products: they are connected to four different HMC memory banks. The first mySplit kernel is used to divide the input data coming from the input port in

a) the configuration part (8 bytes sent to the config_in - config_out chain to distribute the Id of the computing kernels) and

b) the data part (data are the values of the matrix **M** to be stored in the memory and the vector **a** to be multiplied with the matrix) which is sent to the 4 computing kernels through a streamCopy kernel.

The last BuildResultVector kernel is used to concatenate the results produced by the four VectorMatrixProduct kernels, generating the result vector.

8.1 The Scalar Product

As seen in the §6, the basic step to compute the scalar product between the l^{th} row of the matrix ($\mathbf{m}_l$) and the input vector **a** is the following

```
compute the LP values
```

$$s_{ij} = \mathbf{m}_{\mathbf{l:ij}} \cdot \mathbf{a}_{\mathbf{ij}} = \sum_{k=0}^{\frac{m}{LP}-1} \left(m_{l:ip+j+kLP} \cdot a_{ip+j+kLP}\right) \qquad \begin{matrix} i = 0,1,\dots,L-1 \\ j = 0,1,\dots,P-1 \end{matrix}$$

which requires the computation of LP partial scalar products. The basic operation to implement these scalar products is the vector multiply-and-add pipelined function which takes as input P pairs of single-precision floating-point variables and produces P floating-point values (in our implementation P = 32), performing the computation

$$c_i \mathrel{+}= a_i \times b_i; \quad i=1,2,\dots,P.$$

The sketch of the QuickPlay C code to implement the vector pipelined MADD is the following

```
/*#qp pipeline */
Void MADD(float a1,…a32, float b1,… b32,float &c1,… &c32)
{
  c1 += a1*b1;
  …
  c32 += a32*b32;
}
```

Thanks to the /*#qp pipeline*/ directive the previous function is synthesized as a pipelined function which performs 2P=64 floating-point operations per cycle (P add and P mul).

From the synthesis reports of QuickCompiler we know that previous function requires 7 cycles to produce the output results, so L_{MADD}=7 cycles; we use L=8 to include the cycle needed to read the data from the memory. The MADD is implemented through the instantiation of 32 fp adders and 32 fp multipliers.

The MADD() function computes the LP scalar products s_{ij} i=0,..,L-1 and j=0,1,…,P-1 through the following code:

```
count=0; count1=1; …,count31=31; //init the 32 count vars
/*#qp unroll 32*/
for (i=0; i<(m)/(L*P); i++){
  a1 = a[count];
  …
  a32 = a[count31];
  b1 = b[count];
  …
  b32 = b[count31];
//1st group of 32 MADD scalar operations
  MADD(a1, …,a32,b1,…,b32,s0_0,…,s0_31);
  Inc(count,…,count31);//each count var is incremented by 32
  …
  a1 = a[count];
  …
  a32 = a[count31];
  b1 = b[count];
  …
  b32 = b[count31];
  //8th group of 32 MADD scalar operations
  MADD(a1,…,a32,b1,…,b32,s7_0,…,s7_31);
  Inc(count,…,count31); //each count var is incremented by 32
}
```

In our example the size of the vector m assumes the value m=8192 and L×P = 256. The loop is executed m/(LP) = 8192/256=32 times, so the directive /*#qp unroll 32*/ unrolls completely the loop.

The scalar variables a1, …, b32 are read from the FastMemory a[] and the FastMemory b[] in one clock cycle.

8.2 FastMemory

The FastMemory modules are the memories used by QuickCompiler to map internal arrays. They are implemented on embedded ram and are described by the tuple

FastMemory = <W,G,N,DType, Size>

- W is the width of the wide "external" port.
- G is the number of independent groups, each group being formed by N ports; usually G = 2 (as the embedded Ram modules are dual-ported)
- N is the number of typed ports in each of the G groups. Each port presents a data which has size DType;
- DType is the size (in bytes) of the data type stored in the FastMemory. In QuickCompiler, each array is stored in a different FastMemory.
- Size is the size of the memory, expressed in Bytes.

FastMemory has G×N+1 ports.

The large external port, whose width is W=N*DType, is used to transfer data to/from streams or to/from external memories through the qpReadStream(), qpWriteStream() and memcpy() functions. The bandwidth of read/write through this port is given by BW = W*f_{ck} [Byte/s]; typical value is W=128 [B], f_{ck}=150MHz and BW=19.2 GB/s. The latency to access external memories depends on the available memory controller; the HMC controller in the μXComp board is characterized by a latency L_m=200 cycles.

The G×N "internal" ports, whose size is DType, are accessed by the kernel. The internal BW, between the FastMemory and the computing kernel, is G times the BW of the external port. The latency to read a data from the FastMemory to the kernel is one cycle while writing a data from the kernel to the fast memory is accomplished in the same cycle.

Since W≥ Dtype, each group of ports allows accessing N=W/DType elements of an array at the same clock cycle. As the memory is organized in word of W bytes, when the first port of a group is used it selects the memory word being accessed and it allows the other ports of its group to access the other array elements of the word.

The FastMemories a[m] and b[m] have been declared with the directive `/*#qp ports 2 32*/` which specifies that the array, composed by m=8K float elements, is stored in a memory which has G=2 groups of N=32 ports accessible in parallel, every port being four bytes wide (as they are float data type). Both a and b FastMemories are characterized by the tuple

<W=128, G=2, N=32, DType=4, Size=32768 > .

This means that up to 64 floats can be read/written in parallel in one clock cycle.

In one iteration of the loop, the LP s_{ij} values are updated; values s_{ij} are mapped onto the variables si_j (i=0,..,7 and j = 0,..,31).

The previous loop-code is scheduled by the QuickCompiler HLS engine as described in §6, with the performance given by eq. (8).

Looking in the QuickCompiler timing report, we see that the execution of the module implementing the previous code requires 264 clock cycles, in perfect agreement with the formula derived from the analysis $N_{comp}=m/P+L_{MADD}$.

After having computed the LP s_{ij} values, we must sum them together to obtain the result i.e., we must implement the expression

$$b_l = \sum_{i=0}^{L-1} \sum_{j=0}^{P-1} (s_{ij}).$$

The previous formula is very simply computed through the following (not pipelined) function

```
float Sum(float s0_0,..., float s7_31)
{
  float result;
  result =s0_0+s0_1+...+s0_31+s1_0+...+s7_31; //256 operands
  return result;
}
```

which is scheduled by QuickCompiler on one fp adder and requires 263 clock cycles to be executed, slightly better than the simplified model presented in the §6, eq. (9), which was foreseeing 280 clock cycles (in our simplified model we are neglecting the possibility to start the computation of a new layer of sums in the tree adding scheme before terminating the previous layer).

Putting the things together, the number of cycles requested to compute the scalar product of two vectors **a** and **b**, each containing m=8192 floating-point values and stored in two dedicated FastMemory modules, each module having G=2 groups of N=32 ports 4 bytes wide, requires 264 + 263 = 527 clock cycles. This figure corresponds to (nearly) 31 [flop/cycle] which, for a clock frequency f_{ck}=150 [MHz], gives the sustained speed S=31*150= 4650 [MFlop/s].

8.3 MVM with a Coarse-Grained Pipeline

We use the just described scalar product module as a basic block to perform the MVM; the pseudo-code for the MVM is the following:

```
load vector a;
for (i=0; i<n; i++)
{
  load mi, the ith row of M;
  Compute the LP sij values as partial scalar products
  Sum all the sij
}
```

While the loading of the **a** vector is negligible, as it is performed only once, before starting the actual computation loop, the load of the $\mathbf{m}_i$ vector is relevant because it lasts for $N_{mem}=L_{mem}+m*D/W$ cycles, being

- L_{mem} the latency to access the external memory (in our case $L_{mem}\approx 200$)
- W the width of the "external" port of the FastMemory (W=128 [Byte])

In our case (n=m=8192, W = 128, L_{MADD}=8, L_A=3, P_A=1)

- $N_{mem} = 456$
- $N_{comp} = 264$
- $N_{sum} = 263$

and the global number of cycles necessary to compute **b** = **M** x **a** is given by

$$N_{seq} = n*(N_{mem}+N_{comp}+N_{sum}). \quad (17)$$

As the number of floating-point operations to compute the MVM is N_{flop}=2nm, the speed expressed in number of operations per cycle is given by

$$S_{ops/cycle} = \frac{2nm}{n(N_{mem}+N_{comp}+N_{sum})} \approx 16.7 \left[\frac{ops}{cycle}\right].$$

Considering that each iteration of the computing loop is independent on the others, it is immediate to think to a pipelined scheme to overlap the three operations (**Fig. 3**):

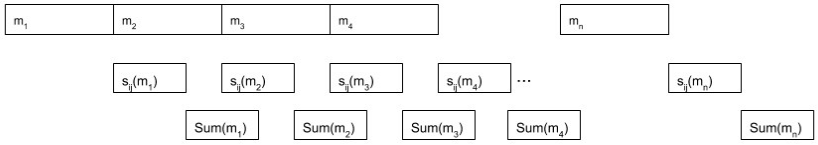

Fig. 3. Gantt for the pipelined execution of the MVM

The computation, arranged according to previous scheduling, can be executed in N_{pipe} cycles

$$N_{pipe} = n*N_{mem}+N_{comp}+N_{sum}. \quad (18)$$

The speed-up of the pipelined implementation, with respect to the not-pipelined implementation, is given by

$$S = \frac{N_{seq}}{N_{pipe}} = \frac{8.1\times10^6}{3.7\times10^6} = 2.2$$

from which we can derive the expected speed for the pipelined implementation, in ops/cycle, through the following expression

$$S_{ops/cycle}(pipe) = S_{ops/cycle}(seq) * S = 16.7 * 2.16 = 36.01 \left[\frac{ops}{cycle}\right].$$

The speed, in flop/s, is obtained as in the following

$$S_{flops/s} = S_{ops/cycle} * f_{ck} = 36.01 * 150 = 5411 \left[\frac{MFlops}{s}\right].$$

The scheduling described in **Fig. 3** is enforced by the QuickPlay HLS when compiling the following code:

```
  ...
qpReadStream(d_in_0,a1,NbElem*sizeof(float));//read vect a
ReadVector(b1, Matrix,row);  row++; // read a row of M
ComputePartialScalarProducts(a1, b1, cr0_0,..., cr0_31);
sum1 = Sum(cr0_0,..., cr0_31);
ReadVector(b2, Matrix, row); row++;
ComputePartialScalarProducts(a1, b2, cr0_0,..., cr0_31);
ReadVector(b3, Matrix, row);  row++;
for (i=0; i<myNbProducts-6; i+=3)
{
  Write(dout,sum1,false); //send an element of the result
                                              //vector
  sum2 = Sum(cr0_0,..., cr0_31);
  Write(dout,sum2,false);
  ComputePartialScalarProducts(a1, b3, cr0_0,..., cr0_31);
  sum3 = Sum(cr0_0,..., cr0_31);
  Write(dout,sum3,false);
  ReadVector(b1, Matrix, row); row++;
  ComputePartialScalarProducts(a1, b1, cr0_0,..., cr0_31);
        sum1 = Sum(cr0_0,..., cr0_31);
  ReadVector(b2, Matrix, row);    row++;
  ComputePartialScalarProducts(a1, b2, cr0_0,..., cr0_31);
  ReadVector(b3, Matrix, row); row++;
}
Write(dout,sum1,false);
i++;  //i is the number of written values
sum2 = Sum(cr0_0,..., cr0_31);
Write(dout,sum2,false);
     i++;  //i is the number of written values
ComputePartialScalarProducts(a1, b3, cr0_0,..., cr0_31);
sum3 = Sum(cr0_0,..., cr0_31);
Write(dout,sum3,i==NbProducts-1);
  …
```

preamble

postamble

In previous code we can recognize three sections:

- the preamble, to fill the pipeline modules; in this section, we find the read (once for all) of the a vector, the read of the first three rows of M, two computations of the partial scalar results and one sum operation.
- the loop, which implements the steady-state of the pipelined behavior; in this section we find three reads of rows of M, three computations of partial scalar result, three sum operations and the write to the output of three results i.e., the manual unroll of three complete processing of three rows of M;
- the postamble, which empties the pipeline (no more matrix rows are read). In this phase it is finished the processing of the last three rows. It is the dual of the preamble; we have no read, one computation of the partial scalar products, two sum operations and three write of the results.

To ensure the parallel execution of the different functions accessing the same array, we used three different buffers to store the rows of matrix M.

The QuickPlay project which instantiates all the available 4 HMC memory modules, each connected to one Compute MatrixVectorProduct, is reported in **Fig. 2**.

Both the input and output ports have been mapped onto a PCIe interface.

The PCIe IP, HMC controller IP, clock and reset generator IP, as well as the copy IP and the FIFO IP are all part of the QuickPlay distribution and are instantiated by

the tool in a transparent way (clock & reset generator, FIFO) or based on the configuration derived from the Visual Editor. The computing kernels are generated by QuickCompiler, the HLS engine of QuickPlay.

9 Performance Results

To show the performance achieved, in terms of both speed and resource usage, we report for the different designs developed (with 1, 2, 3 and 4 kernels, each performing the MVM on a portion of the matrix M)

- the sustained speed [GFlop/s] measured on actual runs on the MicroGate board (equipped with one ARRIA 10 FPGA and 4 HMC memory banks),
- the resource used (ALM - Arithmetic Logic Modules, memory modules M20K).

Table 1. Results when implementing the MVM with 1, 2, 3 and 4 SP kernels

	1 Kernel	2 Kernels	3 Kernels	4 Kernels
Speed [GFlop/s]	5.3	10.6	15.9	21.0
ALM	88547	190648	264600	282473
M20K	500	959	1378	2045

The design presents nearly linear scaling for computational performance.

To understand how resources are used, we report, for the largest design using four equal MatrixVectorProduct kernels, the percentage of the resources (ALM, M20K) used to implement

- the PCIe interfacing IP : ALM 2.7%, M2K 0.7%
- the MatrixVectorProduct kernels: ALM 5.3%, M2K 8.1% each kernel
- the HMC memory controllers : ALM 7.0%, M2K 5.7% each controller
- the other auxiliary modules (reset and clock generators, FIFOs, mySplit and BuildResultVector modules, …) : ALM 18.6%, M2K 22.9%

When the FPGA board was configured with the design using four SP kernels, the power consumption of the board was 40 W, resulting in the energy efficiency of 0.53 GFlop/s/W.

Even if we think that comparison with other implementations is a weak way to evaluate an implementation, we report an alternative realization of the MVM to verify that the proposed solution is aligned with what is allowed by the current technology.

The work presented in [14] reports the implementation of several BLAS routines, including the MVM. The performance of this routine is reported in the case of a 1024x1024 matrix stored within the internal RAM, thus not requiring any communication with the DDR banks; in the case of vectorization width set to 64 (i.e., performing in parallel 64 multiply operations) it is reported a computing speed greater than 20 GFlop/s (both in single and in double precision). While giving an idea of the perfor-

mance achievable by the hardware in the FPGA, such a figure would require a significantly large I/O BW to be sustained for larger matrices (as the 8Kx8k matrices used in our case): the proper buffering and macro-pipelining of the computation to sustain the traffic with the DDR memory is not addressed in [14], not being this the core of the FBLAS implementation.

10 Future Developments

Looking at the Gantt reported in Fig. 3, we see that the transfer of one line of matrix M from memory lasts longer (456 cycles) than the computation of the partial scalar products (264 cycles) and the final sum (263 cycles). This happens because QuickPlay HLS does not support outstanding read operations, which would allow overlapping different memory transfers. Could we use outstanding memory reads, the latency of the next transfer could be overlapped with the actual data transfer of the current, as in the following:

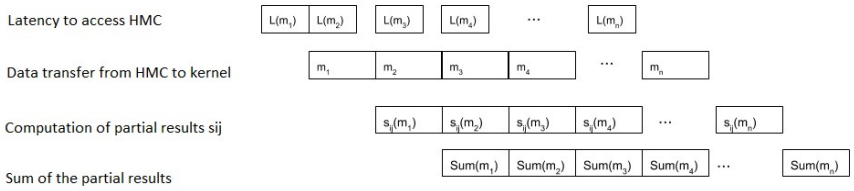

Fig. 4: pipelined implementation with support to outstanding read operations

In the previous figure, we decomposed the time to transfer data from HMC to the kernel into the latency $L(m_i)$ and the actual data transfer. It's easy to verify that the number of cycles needed to perform the computation shown in Fig. 4 is given by

$$N_{pipe} = n*\max(L(m), N_{mem}, N_{comp}, N_{sum})+L(m)+N_{mem}+N_{comp}+N_{sum}.$$

In our case (n=m=8192) the values are L(m)=200, N_{mem}=256, N_{comp}=256 and N_{sum}=263 which yeld

$$N_{pipe} = n*264 + 983$$

Previous value corresponds to

$$S_{ops/cycle}(pipe) = \frac{2nm}{264n + 983} \approx 62[\frac{ops}{cycle}]$$

i.e., 9.3 GFlop/s when f_{ck}=150 MHz, very close to the limit imposed by the memory BW.

11 Conclusions

The activities performed to implement on FPGA the MVM through the QuickPlay HLS flow have been described.

We started formalizing the problem, describing how parallelism is a key factor to achieve the expected performance and we showed how parallelism could be introduced at 4 different levels:

- (spatial) parallelization over the different rows of the matrix, computing in parallel the scalar products between the input vector and p different rows of the matrix M
- parallelization (pipelining) of the basic scalar product, achieved thanks to the introduction of L different independent partial scalar products to break the data dependence characterizing the classical accumulation scheme (L is the latency of the basic Multiply and Add pipelined operation)
- parallelization derived from the iteration of the previous decomposition, dividing each of the L sub-vectors into P smaller sub-vectors, thus performing in parallel P pipelined partial scalar products
- coarse-grained pipelining, overlapping different phases of successive scalar products when multiplying the input vector by different rows of the matrix M (read from external memory, computation of the partial scalar products, sum of the partial results).

Some models to compute the expected performance of the algorithm we have implemented have been presented and discussed. We found a good agreement between the forecasted and the actual performance. This agreement demonstrates the good quality of the hardware generated by the HLS engine.

References

1. «energy efficient high performance computing for real-time science,» [Online]. Available: http://greenflash-h2020.eu/.

2. P. Piatrou e L. Gilles, «Robustness study of the pseudo open-loop controller for multiconjugate adaptive optics,» *APPLIED OPTICS,* vol. 44, n. 6, 2005.

3. E. Gendron e et al., «A novel fast and accurate pseudo-analytical simulation approach for MOAO,» in *Proc. SPIE 9148, Adaptive Optics Systems IV*, 2014.

4. E. Gendron e et al., «High performance pseudo-analytical simulation of multi-object adaptive optics,» in *Euro-Par 2014*.

5. O. Guyon e e. Al., «The compute and control for adaptive optics (CACAO) real-time control software package.,» in *Proc. SPIE 10703, Adaptive Optics Systems VI*, 2018.

6. S. Monboisset, «A Novel Approach to Software-Defined FPGA Computing.,» *XCell Software Journal,* n. 2, 2015.

7. «BLAS (Basic Linear Algebra Subprograms),» [Online]. Available: http://www.netlib.org/blas/.

8. S. Kestur e e. al., «BLAS Comparison on FPGA, CPU and GPU,» in *ISVLSI*, 2010.

9. S. Kestur e e. al., «Towards a Universal FPGA Matrix-Vector Multiplication Architecture,» in *IEEE 20th Int. Symp. on Field-Programmable Custom Computing Machines*, 2012.

10. L. Zhuo e V. Prasanna, «High Performance Linear Algebra Operations on Reconfigurable Systems,» in *SuperComputing SC05*, 2005.

11. L. Zhuo, G. Morris e V. Prasanna, «Designing Scalable FPGA-Based Reduction Circuits Using Pipelined Floating-Point cores,» in *19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

12. C. He, G. Qin e R. Ewing, «High-Precision BLAS on FPGA-enhanced Computers,» in *International Conference on Engineering of Reconfigurable Systems & Algorithms, ERSA 2007*, 2007.

13. C. He, G. Qin, M. Lu e W. Zhao, «Accurate Floating-Point Summation with Group-Alignment Technioque on FPGA,» in *The Int. Conf. on Engineering and Reconfigurable Systems and Algorithms*, 2006.

14. T. De Matteis, J. de Fine Licht e T. Hoefler, «FBLAS: Streaming Linear Algebra on FPGA,» *ArXiv,* vol. abs/1907.07929, 2019.

15. Intel, «Intel FPGA SDK for OpenCL,» 2018. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/products/design-software/embedded-software-developers/opencl/support.html.

16. A. Abdelfattah, D. Keyes e H. Ltaief, «KBLAS: An Optimized Library for Dense Matrix-Vector Multiplication on GPU Accelerators.,» *ACM Trans. Math. Softw.,* vol. 42, n. 3, 2016.

17. C. Boyer, V. Michau e G. Rousset, «Adaptive optics: interaction matrix measurements and real-time control algorithms for the COME-ON project,» in *SPIE Astronomical Telescopes and Instrumentation for the 21st Century,*, Tucson, AZ, United States, 1990.

18. C. Patauner e e. al.D., «FPGA based microserver for high performance real-time computing in Adaptive Optics,» in *Proc. of the "Adaptive Optics for Extremely Large Telescopes" (AO4ELT5)*, 2017.