ISC 2020 DIGITAL

JUNE 22-25

#ISC20

Italian National Agency for New Technologies, Energy and Sustainable Economic Development

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

Alessandro Marongiu, Paolo Palazzari, ENEA, ICT-HPC division

In memory of Alessandro Marongiu

- He was a reference point in all the workplaces where he worked, both in the public research (ENEA) and in the private (the Ylichron spin-off, PLDA and Accelize).
- He was one of the main architects of the QuickPlay HLS flow.
- He worked for more than 20 years on parallel computing. His main interest has been the automation of the process to translate a high-level description of an algorithm into an equivalent, parallel, lower level description.

Outline of the presentation

- Some preliminary considerations on how to use an HLS flow
- The problem to be solved
- Exploitation of spatial and pipeline parallelism at the different granularities
- Few details on the implementation through the QuickPlay HLS flow
- Performance evaluation
- Conclusions

Outline of the presentation

- Some preliminary considerations on how to use an HLS flow
- The problem to be solved
- Exploitation of spatial and pipeline parallelism at the different granularities
- Few details on the implementation through the QuickPlay HLS flow
- Performance evaluation
- Conclusions

High Level Synthesis

- A common misunderstanding:

«as we use some C-like language, developing on FPGA is like developing on CPU»

High Level Synthesis

- A common misunderstanding:

«as we use some C-like language, developing on FPGA is like developing on CPU»

This is quite **true** on the **functional** side (once you have a flow which transparently instantiates all the necessary IPs – memory controller, clock and reset generator, PCIe communication, FIFO and streamed channels, …)

High Level Synthesis

- A common misunderstanding:

«as we use some C-like language, developing on FPGA is like developing on CPU»

This is quite **true** on the **functional** side (once you have a flow which transparently instantiates all the necessary IPs – memory controller, clock and reset generator, PCIe communication, FIFO and streamed channels, …)

```
float ScalarProduct(float a[N], float b[N]) {
float sum = 0;
for (i=0; i<N; i++)
    sum += a[i]*b[i];
return sum;}
```

High Level Synthesis

- A common misunderstanding:

«as we use some C-like language, developing on FPGA is like developing on CPU»

This is quite **true** on the **functional** side (once you have a flow which transparently instantiates all the necessary IPs – memory controller, clock and reset generator, PCIe communication, FIFO and streamed channels, …)

```
float ScalarProduct(float a[N], float b[N]) {
float sum = 0;
for (i=0; i<N; i++)
    sum += a[i]*b[i];
return sum;}
```

In 5 min (plus ~ 1 hour of compile time) we obtain a working design which computes the scalar product

High Level Synthesis

- A common misunderstanding:

«as we use some C-like language, developing on FPGA is like developing on CPU»

This is quite **true** on the **functional** side
but it is **false** when we refer to **performance**

Previous code would require (at least) **N* Latency**$_{\mathbf{Add}}$ cycles to be executed

High Level Synthesis

- A common misunderstanding:

«as we use some C-like language, developing on FPGA is like developing on CPU»

This is quite **true** on the functional side
but it is **false** when we refer to **performance**

Previous code would require (at least) **N*Latency$_{\mathbf{Add}}$** cycles to be executed

- each add is dependent on the result of the previous add;
- we suppose that compiler will be able to overlap the
  - reads from the two memory banks (a[i] and b[i])
  - the a[i-1]*b[i-1] multiply
  - and the sum = sum + result of a[i-2]*b[i-2]

High Level Synthesis

- A common misunderstanding:

«as we use some C-like language, developing on FPGA is like developing on CPU»

This is quite **true** on the functional side
but it is **false** when we refer to **performance**

Previous code would require (at least) **N* $\mathbf{Latency_{Add}}$** cycles to be executed

2N-1 operations => ≈ **2/ $\mathbf{Latency_{Add}}$** operations/cycle ≈ 0.5 op/cycle => **75 Mflop/s** (using a clock frequency of 150 MHz)

High Level Synthesis

- A common misunderstanding:

«as we use some C-like language, developing on FPGA is similar to developing on CPU»

This is quite **true** on the **functional** side

but it is **false** when we refer to **performance**

At this point users say that FPGA is not a good solution to efficiently solve their problem

[illegible] **Latency$_{Add}$** cycles to be executed

2N-1 operations => ≈ **2/ Latency$_{Add}$** operations/cycle ≈ 0.5 op/cycle => **75 Mflop/s** (using a clock frequency of 150 MHz)

High Level Synthesis

- A common misunderstanding:

«as we use some C-like language, developing on FPGA is similar to developing on CPU»

This is quite **true** on the **functional** side

but it is **false** when we refer to **performance**

At this point users say that FPGA is not a good solution to efficiently solve their problem

[illegible] **Latency**$_{Add}$ cycles to be executed

2N-1 operations => ≈ **2/ Latency**$_{Add}$ operations/cycle ≈ 0.5 op/cycle => **100 Mflop/s** (using a typical clock frequency of 200 MHz)

Let’s try to convince them that FPGA can be a good solution once they understand that they must change their mind as they are using a different technology…

Outline of the presentation

- Some preliminary considerations on how to use an HLS flow
- The problem to be solved
- Exploitation of spatial and pipeline parallelism at the different granularities
- Few details on the implementation through the QuickPlay HLS flow
- Performance evaluation
- Conclusions

The Matrix-Vector Multiplication

- In the framework of algorithms for Adaptive Optics (AO) we have been requested to efficiently implement on FPGA technology the multiplication between a large matrix (8K x 8K single precision floating point elements) and several vectors (8K elements)

The Matrix-Vector Multiplication

- In the framework of algorithms for Adaptive Optics (AO) we have been requested to efficiently implement on FPGA technology the multiplication between a large matrix (8K x 8K single precision floating point elements) and several vectors (8K elements)
- Each new vector can be multiplied by the matrix only when the previous matrix vector multiplication is finished

The Matrix-Vector Multiplication

- In the framework of algorithms for Adaptive Optics (AO) we have been requested to efficiently implement on FPGA technology the multiplication between a large matrix (8K x 8K single precision floating point elements) and several vectors (8K elements)
- Each new vector can be multiplied by the matrix only when the previous matrix vector multiplication is finished
- The Matrix-Vector Multiplication (MVM) is the core of the Wavefront Reconstruction control algorithm.

The Matrix-Vector Multiplication

- Because of its size (256 MB), the matrix must be stored in the external memory;

PP1

The Matrix-Vector Multiplication

- Because of its size (256 MB), the matrix must be stored in the external memory;
- It’s well known that MVM is limited by the available memory bandwidth; as discussed in the paper, the computational speed in the MVM cannot be larger than **half of the bandwidth** between the FPGA and the external memory;

Diapositiva 20

PP1 Paolo Palazzari; 06/06/2020

The Matrix-Vector Multiplication

- Because of its size (256 MB), the matrix must be stored in the external memory;
- It's well known that MVM is limited by the available memory bandwidth; as discussed in the paper, the computational speed in the MVM cannot be larger than **half of the bandwidth** between the FPGA and the external memory;
- Computing speed = $\frac{\#\text{Operations}}{\#\text{Cycles to compute MVM}} = \frac{2\text{N}}{\frac{4\text{N}}{\text{BW}}} = \frac{\text{BW}}{2}$

The Matrix-Vector Multiplication

- Because of its size (256 MB), the matrix must be stored in the external memory;
- It's well known that MVM is limited by the available memory bandwidth; as discussed in the paper, the computational speed in the MVM cannot be larger than **half of the bandwidth** between the FPGA and the external memory;
- Our design is targeting a FPGA board with an Intel ARRIA 10 GX1150 FPGA, with 4 HMC memory banks; the BW toward each bank is 17 GB/s so we know that MVM implementation could not sustain more than **34 Gflop/s**

Outline of the presentation

- Some preliminary considerations on how to use an HLS flow
- The problem to be solved
- Exploitation of spatial and pipeline parallelism at the different granularities
- Few details on the implementation through the QuickPlay HLS flow
- Performance evaluation
- Conclusions

Coarse-grained spatial parallelism

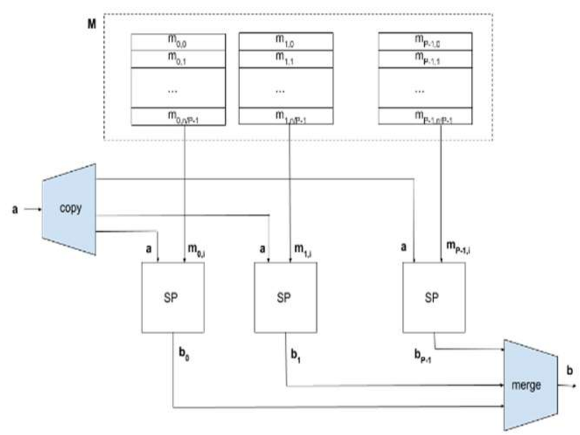
- As we have 4 external memory banks, it is immediate to think to an implementation where 4 equal kernels compute N/4 times the scalar product between a copy of the input vector and a row of the matrix;

Coarse-grained spatial parallelism

- As we have 4 external memory banks, it is immediate to think to an implementation where 4 equal kernels compute N/4 times the scalar product between a copy of the input vector and a row of the matrix;
- the matrix is equally split among the 4 banks, the vector is replicated in each kernel; in this way, each kernel computes in parallel the N/4 elements of the result vector;

Coarse-grained spatial parallelism

- As we have 4 external memory banks, it is immediate to think to an implementation where 4 equal kernels compute N/4 times the scalar product between a copy of the input vector and a row of the matrix;
- the matrix is equally split among the 4 banks, the vector is replicated in each kernel; in this way, each kernel computes in parallel the N/4 elements of the result vector;
- The sketch of the architecture to be implemented is the following



Pipelined implementation of the scalar product

- The scalar product can be implemented with one pipelined MADD (one multiplier and one adder) which iteratively computes the recurrence

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

Pipelined implementation of the scalar product

- 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 $s_0=0$, $a_i \in \mathbf{a}$, $b_i \in \mathbf{b}$.

- As the computation of the next MADD operation is dependent on the completion of the previous operation, a new MADD cannot start until the previous has finished

Pipelined implementation of the scalar product

- The scalar product can be implemented with one pipelined MADD (one multiplier and one adder) which iteratively computes the recurrence

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

- As the computation of the next MADD operation is dependent on the completion of the previous operation, a new MADD cannot start until the previous has finished
- Each time we must wait L cycles (the latency of the MADD operator) before starting a new MADD operation

Pipelined implementation of the scalar product

- Let's partition the **a** and **b** vectors into L equally sized sub-vectors $\mathbf{sa_i}$ and $\mathbf{sb_i}$ (i=1,2,…,L).

Pipelined implementation of the scalar product

- Let's partition the **a** and **b** vectors into L equally sized sub-vectors $\mathbf{sa_i}$ and $\mathbf{sb_i}$ (i=1,2,…,L).
- Thanks to the commutativity and associativity of the ADD operation, the scalar product can be written as the sum of the results of L partial scalar products

$$s = \mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{L-1} ps_i = \sum_{i=0}^{L-1} (\mathbf{sa_i} \cdot \mathbf{sb_i})$$

Pipelined implementation of the scalar product

- Let's partition the **a** and **b** vectors into L equally sized sub-vectors $\mathbf{sa_i}$ and $\mathbf{sb_i}$ (i=1,2,…,L).
- Thanks to the commutativity and associativity of the ADD operation, the scalar product can be written as the sum of the results of L partial scalar products

$$s = \mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{L-1} ps_i = \sum_{i=0}^{L-1} (\mathbf{sa_i} \cdot \mathbf{sb_i})$$

- The computation of each partial scalar product ps_i is independent on the computation of any other partial scalar product ps_j, so we can feed the L computations into the same pipelined MADD component

Pipelined implementation of the scalar product

- Let's partition the **a** and **b** vectors into L equally sized sub-vectors $\mathbf{sa_i}$ and $\mathbf{sb_i}$ (i=1,2,…,L).
- Thanks to the commutativity and associativity of the ADD operation, the scalar product can be written as the sum of the results of L partial scalar products

  $$s = \mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{L-1} ps_i = \sum_{i=0}^{L-1} (\mathbf{sa_i} \cdot \mathbf{sb_i})$$

- The computation of each partial scalar product ps_i is independent on the computation of any other partial scalar product ps_j, so we can feed the L computations into the same pipelined MADD component
- At each cycle, a different partial scalar product enters the pipeline; when, after L cycles, the result exits from the pipeline it is ready to be used for the next MADD

Pipelined implementation of the scalar product

- Let's partition the **a** and **b** vectors into L equally sized sub-vectors $\mathbf{sa_i}$ and $\mathbf{sb_i}$ (i=1,2,…,L).
- Thanks to the commutativity and associativity of the ADD operation, the scalar product can be written as the sum of the results of L partial scalar products

  $\mathrm{s} = \mathbf{a} \cdot \mathbf{b} = \sum_{\mathrm{i}=0}^{\mathrm{L}-1} \mathrm{ps_i} = \sum_{\mathrm{i}=0}^{\mathrm{L}-1} (\mathbf{sa_i} \cdot \mathbf{sb_i})$
- The computation of each partial scalar product ps_i is independent on the computation of any other partial scalar product ps_j, so we can feed the L computations into the same pipelined MADD component
- At each cycle, a different partial scalar product enters the pipeline; when, after L cycles, the result exits from the pipeline it is ready to be used for the next MADD
- After N+L-1 cycles the L ps_i values have been computed (full utilization of the pipeline)

Pipelined implementation of the scalar product

- Let's partition the **a** and **b** vectors into L equally sized sub-vectors $\mathbf{sa_i}$ and $\mathbf{sb_i}$ (i=1,2,…,L).
- Thanks to the commutativity and associativity of the ADD operation, the scalar product can be written as the sum of the results of L partial scalar products

  $s = \mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{L-1} ps_i = \sum_{i=0}^{L-1} (\mathbf{sa_i} \cdot \mathbf{sb_i})$
- The computation of each partial scalar product ps_i is independent on the computation of any other partial scalar product ps_j, so we can feed the L computations into the same pipelined MADD component
- At each cycle, a different partial scalar product enters the pipeline; when, after L cycles, the result exits from the pipeline it is ready to be used for the next MADD
- After N+L-1 cycles the L ps_i values have been computed (full utilization of the pipeline)
- The final result is computed summing the L ps_i values. This additional sum requires O(log(L)) cycles and is negligible when N >> L

Fine-grained spatial parallelism

- With the fully pipelined computation of the scalar product and the coarse-grained parallelism, we can read from the external memory 4 floats at each cycle i.e., when f_{ck}=150 MHz, we read **2.4 GB/s**

Fine-grained spatial parallelism

- With the fully pipelined computation of the scalar product and the coarse-grained parallelism, we can read from the external memory 4 floats at each cycle i.e., when f_{ck}=150 MHz, we read **2.4 GB/s**
- In this way we waste a lot of the available memory BW, which is 68 GB/s, and we limit the computing performance because it must be less than half of the used memory BW. With the fine-grained pipelined scheme and using the coarse-grained spatial parallelism, the performance is less than **1.2 Gflop/s**

Fine-grained spatial parallelism

- With the fully pipelined computation of the scalar product and the coarse-grained parallelism, we can read from the external memory 4 floats at each cycle i.e., when f_{ck}=150 MHz, we read **2.4 GB/s**
- In this way we waste a lot of the available memory BW, which is 68 GB/s, and we limit the computing performance because it must be less than half of the used memory BW. With the fine-grained pipelined scheme and using the coarse-grained spatial parallelism, the performance is less than **1.2 Gflop/s**
- To increase the used memory BW we partition each of the L sub-vectors into P smaller sub-vectors $\mathbf{ssa_{ij}}$ and $\mathbf{ssb_{ij}}$

$$\mathrm{s} = \mathbf{a} \cdot \mathbf{b} = \sum_{\mathrm{i}=0}^{\mathrm{L}-1} (\mathbf{sa_i} \cdot \mathbf{sb_i}) = \sum_{i=0}^{L-1} \sum_{\mathrm{j}=0}^{\mathrm{P}-1} \left(\mathbf{ssa_{ij}} \cdot \mathbf{ssb_{ij}}\right)$$

Fine-grained spatial parallelism

- With the fully pipelined computation of the scalar product and the coarse-grained parallelism, we can read from the external memory 4 floats at each cycle i.e., when f_{ck}=150 MHz, we read **2.4 GB/s**
- In this way we waste a lot of the available memory BW, which is 68 GB/s, and we limit the computing performance because it must be less than half of the used memory BW. With the fine-grained pipelined scheme and using the coarse-grained spatial parallelism, the performance is less than **1.2 Gflop/s**
- To increase the used memory BW we partition each of the L sub-vectors into P smaller sub-vectors $\mathbf{ssa_{ij}}$ and $\mathbf{ssb_{ij}}$

$$s = \mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{L-1} (\mathbf{sa_i} \cdot \mathbf{sb_i}) = \sum_{i=0}^{L-1} \sum_{j=0}^{P-1} (\mathbf{ssa_{ij}} \cdot \mathbf{ssb_{ij}})$$

- The LP partial scalar products are all independent: at each cycle, each scalar product reads P elements from the matrix (and P from the vector which is permanently stored in the local memory)

Fine-grained spatial parallelism

$$s = \mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{L-1} (\mathbf{sa_i} \cdot \mathbf{sb_i}) = \sum_{i=0}^{L-1} \sum_{j=0}^{P-1} \left(\mathbf{ssa_{ij}} \cdot \mathbf{ssb_{ij}}\right)$$

- P is the fine-grained spatial parallelism. The value of P is set to saturate the memory BW, i.e.

$4Pf_{ck} = Mem_{BW} \Rightarrow P = \frac{Mem_{BW}}{4f_{ck}}$ (to be rounded at a power of 2)

Fine-grained spatial parallelism

$$s = \mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{L-1} (\mathbf{sa_i} \cdot \mathbf{sb_i}) = \sum_{i=0}^{L-1} \sum_{j=0}^{P-1} (\mathbf{ssa_{ij}} \cdot \mathbf{ssb_{ij}})$$

- P is the fine-grained spatial parallelism. The value of P is set to saturate the memory BW, i.e.

  $4Pf_{ck} = Mem_{BW}$ => $P = \frac{Mem_{BW}}{4f_{ck}}$ (to be rounded at a power of 2)

- With Mem_{BW} = 17 GB/s and f_{ck}= 150 MHz we get

  P = 28 => round to **32**

- In each kernel we start, at each clock cycle, 32 MADD operations.

Fine-grained spatial parallelism

$$\mathrm{s} = \mathbf{a} \cdot \mathbf{b} = \sum_{\mathrm{i}=0}^{\mathrm{L}-1} (\mathbf{sa_i} \cdot \mathbf{sb_i}) = \sum_{i=0}^{L-1} \sum_{\mathrm{j}=0}^{\mathrm{P}-1} (\mathbf{ssa_{ij}} \cdot \mathbf{ssb_{ij}})$$

- Once the LP partial scalar products have been computed (in N/P + L – 1 clock cycles), all these values must be summed together

Fine-grained spatial parallelism

$$s = \mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{L-1} (\mathbf{sa_i} \cdot \mathbf{sb_i}) = \sum_{i=0}^{L-1} \sum_{j=0}^{P-1} (\mathbf{ssa_{ij}} \cdot \mathbf{ssb_{ij}})$$

- Once the LP partial scalar products have been computed (in N/P + L – 1 clock cycles), all these values must be summed together
- Using P_A adders having latency L_A, the number of cycles to sum n=LP numbers is given by

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

Fine-grained spatial parallelism

$$\mathrm{s} = \mathbf{a} \cdot \mathbf{b} = \sum_{\mathrm{i}=0}^{\mathrm{L}-1} (\mathbf{sa_i} \cdot \mathbf{sb_i}) = \sum_{i=0}^{L-1} \sum_{\mathrm{j}=0}^{\mathrm{P}-1} \left(\mathbf{ssa_{ij}} \cdot \mathbf{ssb_{ij}}\right)$$

- Once the LP partial scalar products have been computed (in N/P + L – 1 clock cycles), all these values must be summed together
- Using P_A adders having latency L_A, the number of cycles to sum n=LP numbers is given by

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

- From previous expression we get the number of cycle to compute a scalar product

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

Coarse-grained pipelining

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

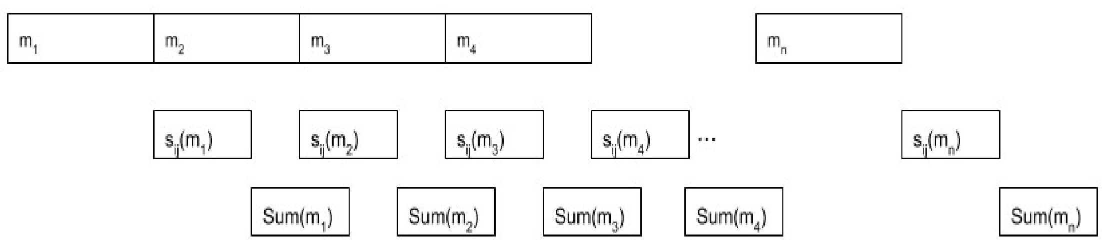
```
for (l=0; l<N; l++){
 load m_l from the external memory
 compute the LP partial scalar products s_ij
 compute the final result b_l = Σ_i,j (s_ij)
}
```

Coarse-grained pipelining

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

```
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)
}
```

- as the loop iterations are independent, they can be pipelined with the following schedule



Outline of the presentation

- Some preliminary considerations on how to use an HLS flow
- The problem to be solved
- Exploitation of spatial and pipeline parallelism at the different granularities
- Few details on the implementation through the QuickPlay HLS flow
- Performance evaluation
- Conclusions

The MADD operator (with fine grained-spatial parallelism)

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

The scalar product (fine-grained pipelined and spatial parallelism)

```
count=0;…,count31=31; //init the 32 count vars
/*#qp unroll 32*/
for (i=0; i<(N)/(L*P); i++){
 // 1st value
 a1 = a[count];   …   a32 = a[count31];
 b1 = b[count];   …   b32 = b[count31];

 MADD(a1, …,a32,b1,…,b32,s0_0,…,s0_31);

 Inc(count,…,count31);
 …

 // Lth value
 a1 = a[count];   …   a32 = a[count31];
 b1 = b[count];   …   b32 = b[count31];

 MADD(a1,…,a32,b1,…,b32,s7_0,…,s7_31);
 Inc(count,…,count31);
}
```

The sum 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;
}
```

MVM with corse-grained pipelining

The preamble

```
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++;
```

MVM with corse-grained pipelining

The main body

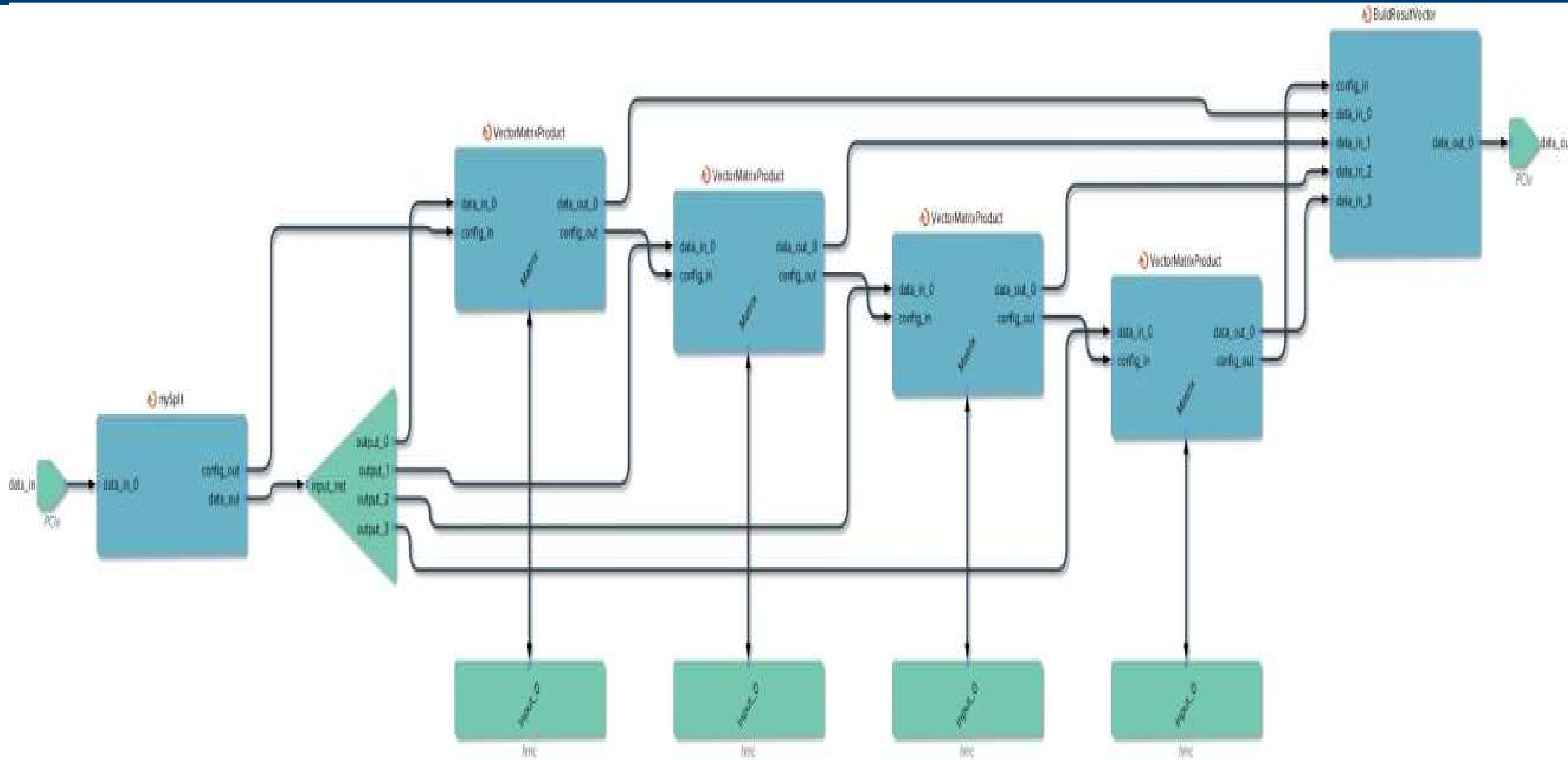
```
for (i=0; i<myNbProducts-6; i+=3)    {
        Write(dout,sum1,false); //send an element of the result vector
        sum2 = Sum(cr0_0,..., cr0_31);
        Write(dout,sum2,false);
        ComputePartialScalarProducts(a1, b3, cr0_0,..., cr0_31);
        sum3 = Sum(cr0_0,..., cr0_31);
        Write(dout,sum3,false);
        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++;}
```

MVM with corse-grained pipelining

The postamble

```
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);
```

The whole design



Outline of the presentation

- Some preliminary considerations on how to use an HLS flow
- The problem to be solved
- Exploitation of spatial and pipeline parallelism at the different granularities
- Few details on the implementation through the QuickPlay HLS flow
- Performance evaluation
- Conclusions

Performance

	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

Performance

- The 21 Gflop/s is below the expected limit, fixed by the available memory BW (34 Gflop/s)
- Going more in depth, we see that the number of cycles needed to transfer data from the external memory to the FPGA internal memory is given by

$$N_{mem} = \frac{N}{P} + L_m$$

where L_m= 200 cycles. As N/P in our case is 256, the latency is comparable with the transfer time.

Therefore we see a memory BW = $f_{ck} \frac{4N}{\frac{N}{P}+L_m} \approx 11 \frac{GB}{s}$ which corresponds to the computing speed of 5.5 Gflop/s, in good agreement with the achieved performance (5.3 Gflop/s with one kernel).

Performance

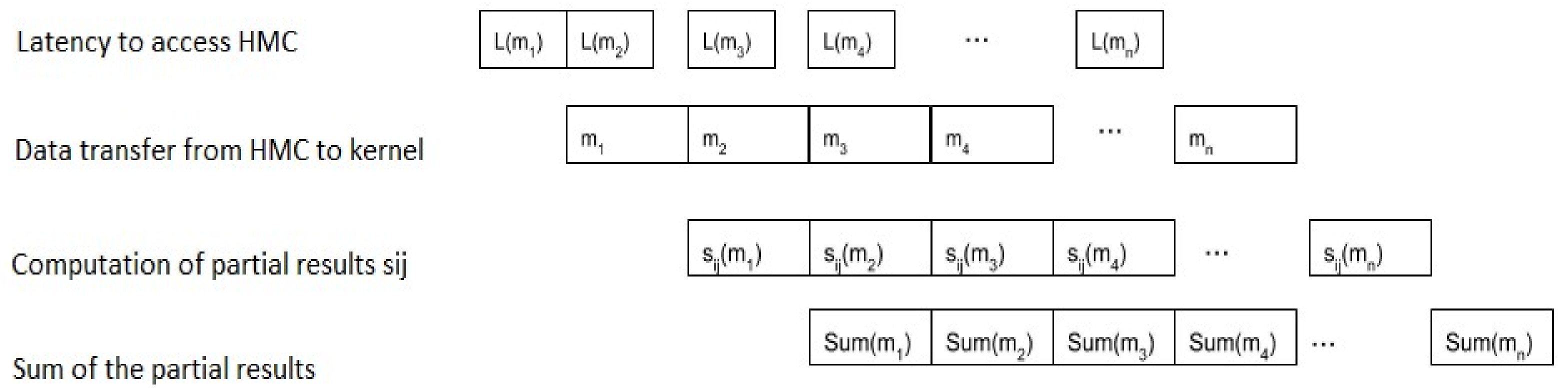
- In order to achieve the expected performance (8.5 Gflop/s for each kernel), we should be able to overlap the latency of the memory read for the transfer of one line of the matrix with the actual transfer of the previous line.

Performance

- In order to achieve the expected performance (8.5 Gflop/s for each kernel), we should be able to overlap the latency of the memory read for the transfer of one line of the matrix with the actual transfer of the previous line.
- In order to do this, the memory controller (and the HLS engine) should be able to support outstanding memory accesses (which was not the case in our environment)

Performance

- In order to achieve the expected performance (8.5 Gflop/s for each kernel), we should be able to overlap the latency of the memory read for the transfer of one line of the matrix with the actual transfer of the previous line.
- In order to do this, the memory controller (and the HLS engine) should be able to support outstanding memory accesses (which was not the case in our environment)



Outline of the presentation

- Some preliminary considerations on how to use an HLS flow
- The problem to be solved
- Exploitation of spatial and pipeline parallelism at the different granularities
- Few details on the implementation through the QuickPlay HLS flow
- Performance evaluation
- Conclusions

Conclusions

- We discussed the use of an HLS tool to implement the MVM algorithm on FPGA

Conclusions

- We discussed the use of an HLS tool to implement the MVM algorithm on FPGA
- We showed the necessity to be aware of the different kind of parallelism in order to efficiently exploit them

Conclusions

- We discussed the use of an HLS tool to implement the MVM algorithm on FPGA
- We showed the necessity to be aware of the different kind of parallelism in order to efficiently exploit them
- In this case, we used both pipeline and spatial parallelism at the fine-grain and at the coarse grain

Conclusions

- We discussed the use of an HLS tool to implement the MVM algorithm on FPGA
- We showed the necessity to be aware of the different kind of parallelism in order to efficiently exploit them
- In this case, we used both pipeline and spatial parallelism at the fine-grain and at the coarse grain
- We exposed our idea that HLS should not abstract us too much from the actual architecture, as we should be able to foresee which should be the performance achievable and the performance of the actual HLS implementation of a given algorithm should be evaluated against this theoretical prediction

Conclusions

- We discussed the use of an HLS tool to implement the MVM algorithm on FPGA
- We showed the necessity to be aware of the different kind of parallelism in order to efficiently exploit them
- In this case, we used both pipeline and spatial parallelism at the fine-grain and at the coarse grain
- We exposed our idea that HLS should not abstract us too much from the actual architecture, as we should be able to foresee which should be the performance achievable and the performance of the actual HLS implementation of a given algorithm should be evaluated against this theoretical prediction
- We discourage as much as possible performance evaluation through comparison with other implementations

- Thank you for your attention

- For any information, feel free to contact me at

  paolo.palazzari@enea.it

SPONSORS

Follow us on Twitter at #ISC20 !

ISC 2020 DIGITAL MEDIA SPONSORS

Follow us on Twitter at #ISC20 !