Unifying Stream Based and Reconfigurable Computing to Design Application Accelerators

MSc Article
Bruno Miguel de Oliveira Francisco
bsmash@inesc-id.pt

Presented at the IEEE/IFIP International Conference on VLSI and System-on-Chip (VLSI-SoC) 2010, Madrid, Spain

Abstract—To facilitate the design of hardware accelerators we have proposed the adoption of the stream-based computing model and the usage of Graphics Processing Units (GPUs) as prototyping platforms. This model exposes the maximum data parallelism available in the applications and decouples computation from memory accesses. In this paper we go a step further in showing how to use the proposed methodology to accelerate a widely used MrBayes bioinformatics application. In particular, we provide design and implementation procedures and details. We analyze problems faced during the implementation such as the connectivity between the CPU and the FPGA and we provide possible solutions. Experimental results show that our mapping of the stream-based program for the GPU into hardware structures leads to real improvements in performance, scalability and cost. The hardware accelerator allows to reduce the respective processing time up to more that two hundred times while the whole bioinformatics application can run 1.44 faster than by using only the host.

I. INTRODUCTION

Reconfigurable hardware can be used as a very efficient co-processing solution to accelerate certain types of applications. Both algorithmic and hardware latest evolution trends are turning FPGAs and reconfigurable hardware in general more attractive co-processing solutions. However, although reconfigurable hardware is particularly suitable to accelerate small but computationally intensive kernels, mapping algorithms into hardware is still an open research topic [1]. Frequently, original algorithms have to be modified to achieve efficient hardware implementations and the burden to map and design efficient hardware implementations becomes a limiting factor.

In [2] we have proposed a new hardware methodology based on the stream programming model, which uses GPUs at an intermediate phase of the design to simplify the algorithm mapping and achieve efficient hardware implementations. Actually, efficient streaming program is able to exploit most of the existent concurrency, namely by dividing the work into several primitive kernels that can run in parallel on the GPUs numerous simple processing elements. Since this approach reveals most of the concurrency existent in the application, it naturally leads to scalable hardware architectures.

To illustrate the proposed approach for designing hardware accelerators we use MrBayes [3], a bioinformatics program that performs the Bayesian inference of evolutionary (phylogenetic) trees, as a case-study. The field of phylogenetic inference deals with the reconstruction of the evolutionary history for a set of organisms based on a multiple sequence alignment of molecular sequence data. The scoring function used in MrBayes is also adopted in other popular programs for phylogenetic inference such as RAxML [4]. Namely, there have been also recent efforts to port RAxML to FPGAs [5], which corroborates the fact that FPGAs have the potential needed for this types of applications. We believe that our methodology can be beneficial in such works, reducing the designer’s effort, while obtaining good performance results.

In [2] we only show a proof of concept implementation of MrBayes kernel and we provide estimated results for different FPGA platforms. However, one can argue that other factors must be taken into account when facing real implementations, such as communication of the hardware accelerator with the CPU and other FPGA's physical limitations. These factors can have a great impact in performance and scalability of the system. Thus, in this work we go a step further into considering a real implementation of the MrBayes application in an FPGA having as a baseline the previously proposed methodology. Moreover, we describe problems faced during this implementation and real experimental results. The contributions of this paper can be summarized as follows:

- a complete working solution was designed to use a reconfigurable device as a co-processor for a real application using as baseline the methodology proposed in [2],
- real details not considered in the model were analyzed and taken into account, namely we describe the main problems faced during the implementation and possible general solutions,
- the complete system solution is evaluated on a real platform, and we investigate other possible configurations to analyze the global scalability of the system for different FPGAs and understand what are the advantages and disadvantages of this type of solution for scientific applications.

Overall, our results show that efficient hardware accelerators can be easily designed for reconfigurable devices by using the stream-based programming models. These accelerators have better performance and cost than the programmable approaches with a low Non-recurring Engineering (NRE).

II. APPLICATION DESCRIPTION

MrBayes is based on the Maximum Likelihood (ML) model [6] that represents a broadly accepted criterion to score phylogenetic trees. Profiling has shown that the most computationally intensive core of the application consists in two Phylogenetic Likelihood Function (PLF): CondLikeDown, and CondLikeRoot, which account for more than 85% of the total execution. All current PLF-based programs spend the largest part of execution time, typically around 85-95%, for computing the PLF [7]. Thus, the PLF function represents a
typical example of a candidate for parallelization at a fine level of granularity.

The PLFs are computed on a fixed tree, with the estimation of the branch lengths and parameters of the statistical model of nucleotide substitution. For DNA data sequences, a model of nucleotide substitution is provided by a 4x4 matrix (denoted as \( Q \) and shown in Figure 1). This matrix contains the substitution probabilities along the time for a certain DNA nucleotide (\( A \) - Adenine, \( C \) - Cytosine, \( G \) - Guanine or \( T \) - Thymine) to mutate into a nucleotide \( A, C, G \) or \( T \).

In this application, the extended \( I \) model \([8]\) is adopted as usual, with 4 discrete rates, \( r_0, ..., r_3 \) (see Figures 1 and 2). To compute the likelihood of a fixed unrooted tree topology with given branch lengths and model parameters, one initially needs to compute the entries for all internal likelihood vectors. They contain the probabilities of observing an \( A, C, G \) or \( T \) for each column of the input alignment. Hence, the conditional likelihood vectors “cl” have the same length “m” as the sequences in the input alignment. The PLFs mainly consist of independent \texttt{foreach} loops with a computational load that depends on the sequence length (m) and the number of discrete rates (r), as described in Figure 3(a). In each iteration, the functions \texttt{CondLikeDown} and \texttt{CondLikeRoot} multiply the likelihood vector elements by the substitution matrix for each of the defined discrete rates. Thus, considering 4 discrete rates, the computation of a likelihood element requires 4 matrix-vector multiplications, or in other words 16 inner products. The inner product can be seen as a reduction, namely the multiply and accumulate operations over 2 vectors of 4 floating-point numbers as depicted in Figure 3(b). PLF implementation in MrBayes uses single-precision floating-point arithmetic. For more details about MrBayes parallelization we refer readers to \([2]\).

III. Designed Architecture

As stated in Section I, the work presented in this manuscript proposes a complete hardware design solution to accelerate the PLF functions of the MrBayes algorithm on FPGAs. The accelerator is designed to be implemented on an FPGA which is connected to a host processor (CPU) as shown in Figure 4. The general control flow of the application is executed on the CPU and the PLF function calls are offloaded to the FPGA.

According to the solution proposed in \([2]\), to implement a stream-based solution we need a buffer to store data transferred between the CPU and the FPGA. In this case two independent buffers are required, one for the input data transferred to the FPGA and another one to store the processed data to be transferred back to the CPU.

The overall block diagram of the proposed accelerator is presented in Figure 4. This processor is composed by two types of modules: a General Control Unit (GCU) responsible for the overall system control, including the required iterative processing due to the large amounts of processed data; and N Processing Unit’s (PU) that store and compute the data. The basic operation implemented in each PU is the inner product, according to the PLF algorithm presented in Section II. Because our solution is fully modular, it is easy to scale the number of PUs implemented in the FPGA according to the available resources.

A. Processing Unit

The PU is composed by 3 distinct units: an Arithmetic and Logic Unit (ALU) to compute the inner products; a Storage Unit (SU) to store the input and output data; and a Datapath Control Unit (DCU) to control the pipelined execution inside the ALU. The overview of the PU is presented in the diagram block shown in Figure 5.

1) Arithmetic and Logic Unit: As presented in Section I, the ALU design was based in the methodology proposed in our previous work \([2]\). This implementation explores the maximum data parallelism available in the application and it allows the GPU to be used as a prototyping platform for the custom design presented herein. The computation of each likelihood entry, as described in Section II, is summarized as a multiplication and a sum of the inputs for each child and a final product between left and right children as showed in Figure 6(a). As a result of applying the mentioned methodology to the algorithm we obtain the Final Collapsed Archi-
conditional likelihood discrete array. In Figure 6(b) including the ALU is able to compute one likelihood vector with maximum efficiency. Namely, it is possible to reuse the adder arithmetic unit very efficiently by introducing a feedback loop (represented as S2 and S3 in Figure 6(a)) and interleaving the input data of each inner product. Moreover, the computation of one likelihood discrete rate array requires the use of 4 FCA units that can be collapsed into the same hardware by using floating-point units with 8 pipeline stages \(2(Left/Right) \times 4(FCA\ units)\). As explained, the input streams arriving from the substitution matrix must be interleaved, i.e., each nucleotide \((A, C, G, \text{ or } T)\) has to be synchronized with the correspondent line of the substitution matrix. The final input data stream in Figure 6(a), can be easily obtained by properly controlling the multiplexer (“Mux”) depicted in Figure 6(b). The final design represented in Figure 6(b) including the ALU is able to compute one conditional likelihood discrete array.

2) Storage Unit: Each storage unit is composed by 3 memory modules of 512 words used to store the input vectors for the left and right children, and the results vector (clL, clR, and clP, respectively); and 2 memory modules of 16 words to store the left and right substitution matrices (QL and QR, respectively). The vectors and the matrices are represented in Figure 6(b) and stored in the memories showed in Figure 5. All memories used are true dual port RAMs, one port is connected to the ALU and the other is accessible directly from the exterior. This fact makes the data processing and its transfer independent, which is important since it is an iterative process, due to memory limitations, and it is possible to overlap the computation and transfer times, for example by implementing a double-buffering mechanism.

3) Datapath Control Unit: The DCU implements a finite state machine responsible for generating all the signals required to control the ALU, namely enabling pipelining registers and controlling multiplexers, which guarantee the correct input order and propagation of the data through the pipeline stages. This control is critical to ensure the correct interleaving of the data.

B. General Control Unit

In general this unit is responsible for controlling the execution flow. It controls the beginning and finalization of an PLF call, by using two system signals, namely start and reset. These signals affect all PU’s and are generated according to the data present on the memories. The GCU monitors them to detect when they are filled with data and transmits the start signal to the PUs. At the other end, it verifies when the data is completely processed and stored at the internal memories and triggers the transference of the results back to the main memory. It is also responsible for the management of the memory addresses. The main purpose of this control unit is to correctly synchronize the inputed, processed and outputted data between iterations.

IV. IMPLEMENTATIONS DETAILS

As mentioned previously the proposed design is general enough and can be customized according to the target FPGA device, for example by defining the number of PUs according to the requirements and the available hardware resources in the FPGA. It is also possible to reuse this architecture for other streaming algorithms by only modifying the ALU and the DCU, and resizing the SU to accommodate the required data. In this work we have used Virtex 4/5 platforms. The only restriction applied to the design is the fact that the number of PU’s must be a multiple of the discrete rates adopted (herein 4). This restriction appears as a result of the implementation since each discrete rate is associated to a single PU. Moreover the number of PUs is also limited by area, the number of DSP’s and BRAMs available. This critical resources vary according to the FPGA device used.

In this section we describe relevant implementation details as well as the main problems faced and the solutions to overcome or mitigate their effects.

A. Arithmetic and Logic Unit

As described in Section III the ALU is composed by three floating-point pipelined units: two multipliers and an adder as represented in Figure 6(a). The unit used to multiply the cl vector elements by the lines of the substitution matrix has 8 pipeline stages. The other one, used to multiply the left and right children has only 4 stages. The adder is used to accumulate the subsequent multiplications and has 8 pipeline stages. To implement these floating-point units the FPGA technologies provide DSP’s which are optimized to perform arithmetic and logic operations increasing the frequency and improving the efficiency. In our case the number of pipeline stages used maps well to the DSPs. It would be possible in other FPGA architectures to use a different number of pipeline stages by modifying the interleaving pattern of the inputs.
B. Storage Unit

The FPGA technologies provide optimized memory structures, namely blocks of RAM, that can be used to increase the amount of stored data while efficiently exploring the reconfigurable capabilities of the design. Since in modern FPGA technologies the RAMs usually have true dual ports, both access ports can be addressed independently. This allows to implement the stream-based processing solution directly on the hardware. Thus, in this implementation one port is directly connected to the ALU and the other is connected to the interface used to transfer the data between the CPU and the FPGA device. The latter port is directly mapped to the address bus of the interface, and consequently to the CPU, while the one accessed by the ALU is mapped via an internal address counter. In the input memories the port accessed by the interface is used only in writing mode and the enable is generated by the interface. The second is used as a reading port by the ALU. Relatively to the output memories the usage is the opposite and the enable is generated by the DCU.

C. Data management

The adopted data management method is mainly influenced by the system bus characteristics due to the huge differences on the throughput and the number of independent channels.

To analyze the performance of different types of buses there are some particular assumptions and definitions that must be considered, namely: i) “serial execution” refers to the sequential transmission, processing and reception of the data for each iteration and its total execution time is represented as $t_{send} + t_{processing} + t_{receive}$, which is used as the reference scenario; ii) “slow bus” refers to a bus where the transmission time of the data required in each iteration is much longer than the required processing time, typically one order of magnitude or higher, and a “fast bus” refers to the situation when it is at the same order of magnitude of the processing time; iii) at least two independent communications are required for the send and receive operations. In this paper we analyze three different scenarios. An example of each, showing the bus occupation and the processing time is showed in Figure 7. It is worth to note that in all scenarios the proposed optimizations require additional control logic, namely multiplexers to switch across the parallel memory blocks, which are controlled by the GCU.

1) Scenario 1: Slow bus with 1 channel: In this scenario we consider the same type of bus used for the serial version but we try to overcome the huge delays associated with that straightforward approach. The first possible optimization consists in transferring as much data as possible in one unique communication avoiding the time to establish new connections, i.e., instead of transmitting $S1, S2, ..., Sn$ sequentially we establish one initial communication to send the data for the N iterations to the FPGA, process it all, and send it back in another communication. In this case we need N memory blocks for each PU to store all the input and output data of the complete operation. Another observation is the fact that after receiving the first blocks of data the system can start to compute them, overlapping the computation time with the transmission time. However, it is clear that this solution is limited by the number of available BRAM’s in the reconfigurable device. Nevertheless, this limitation can be mitigated if a double-buffering technique is used on the input data. Double-buffering consists in having 2 blocks of memory used in parallel as buffers in the system. In practice, the blocks are associated interchangeably with the computation and communication for each iteration N. The memory size limitation is then reduced to the output data size. Overall, in this scenario the total execution time is limited by the total data transmission time, i.e., $t_{send} + t_{receive}$.

2) Scenario 2: Slow bus with 2 channels: In the second case the same kind of bus is considered but with 2 channels, allowing the system to send data without interruption trough one channel and receive trough the other. Thus, the time to re-establish connections is avoided and at the same time the memory limitation of the previous scenario can be overcome by using double-buffering on both the input and output data. Thus, in this case the total execution is limited to the largest transmission time, which in our case is the input transmission. The total time is represented as $t_{send} + constant$, where constant is the time to initialize the buffers.

3) Scenario 3: Fast bus with 1 channel: Finally, the third scenario consists in a fast bus. Again, we implement the double-buffering technique to prevent stalls in the process by maintaining the data always available, and to reduce the memory size requirements. Since the transmission takes approximately the same or less time than the processing, in this scenario the total execution is limited by the maximum of the total transmission time and the processing time ($max\{t_{send} + t_{receive}, t_{processing}\}$).

Scenario 3 falls into Scenario 1 when the bus time is dominant, with the difference that in this case we transmit the data in smaller slots as we are able to re-establish the communication during the processing without having a large impact on the system’s performance and thus reducing memory size limitation. Moreover, we use only one channel, which can be important to reduce the power consumption and the area requirements, e.g., in systems where the bus controller is implemented in the FPGA a second channel requires more circuit area and reduces the number of slices available.

D. Software Interface

As already mentioned the FPGA is used to accelerate the PLF calls. However, an important issue is how to handle in a simple way the interface between the MrBayes original program and the FPGA. From the programmers point of view it is crucial to have a transparent way to use the acceleration hardware, namely because, due to the memory limitations of the FPGA, there is some control that must be performed...
TABLE I: Available Resources

<table>
<thead>
<tr>
<th>Device</th>
<th>Type</th>
<th>Board</th>
<th>Bus type</th>
<th>Slices</th>
<th># BRAMs</th>
<th># DSPs</th>
<th>Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtex 4 (XC4VSX35)</td>
<td>Real</td>
<td>Annapolis Wildcard 4</td>
<td>PCMCIA</td>
<td>15,360</td>
<td>192</td>
<td>192</td>
<td>Intel Pentium M @ 1.86 GHz</td>
</tr>
<tr>
<td>Virtex 4 (XC4FX60)</td>
<td>Real</td>
<td>Avnet Xilinx V4FX</td>
<td>PCIe 8x</td>
<td>25,280</td>
<td>232</td>
<td>128</td>
<td>Intel Xeon @ 1.86 GHz</td>
</tr>
<tr>
<td>Virtex 4 (XC4VSX55)</td>
<td>Hypothetic</td>
<td>n/a</td>
<td>PCIe 16x</td>
<td>24,576</td>
<td>320</td>
<td>512</td>
<td>Intel Xeon @ 1.86 GHz</td>
</tr>
<tr>
<td>Virtex 5 (XC5VFX200T)</td>
<td>Hypothetic</td>
<td>n/a</td>
<td>RocketIO</td>
<td>30,720</td>
<td>456</td>
<td>384</td>
<td>Intel Xeon @ 1.86 GHz</td>
</tr>
</tbody>
</table>

![Fig. 8: Input and Output Streams](image)

on the CPU side. Thus, to completely decouple the FPGA implementation details from the end user, we developed a simple interface that can be directly used in the original code to execute the PLF on the FPGA.

The additional procedures required include data reorganization into streams which are split and sent to the FPGA in multiple transfers. According to the architecture description, the streams are mapped into the left and right memories of the PUs as shown in Figure 8. Each transfer fills all the memories implemented in the FPGA device with \((M_{\text{size}} \times \text{discrete rates}) \times 2(\text{Left and Right}) \times N)\) words, where \(N\) is the number of the PUs implemented and \(M_{\text{size}}\) is the number of elements that can fit into the target FPGA memories. The latter number depends on the FPGA BRAM size, in our case it is 128, since the BRAMs used have 16KB, 512 words of 32 bit each.

The preparation of the conditional likelihood vectors to be sent consists in separating the elements by their discrete rates and distributing them by the PUs. As already referred in this section, each PU is associated with one of the discrete rates used and it must be filled with the correspondent conditional likelihood vectors and the substitution matrices. The algorithm used to prepare the data is represented in Figure 9. This procedure consists mainly of independent for loops with a computational load that depends on the number of elements \((m)\) and discrete rates \((r)\). In each iteration the software interface copies the likelihood elements to a stream, organizing them by the corresponding PU for each defined discrete rate. Moreover, the substitution matrices are also copied in a similar way. The reverse procedure is used to collect the results returned by the FPGA.

V. Experimental Setup and Results

The proposed architecture was implemented and tested in two Xilinx Virtex 4 FPGA devices. To analyze the scalability of the system we also extrapolated the obtained real results after Place&Route to other two devices: one larger Virtex 4 device and a newer Virtex 5 device. The characteristics of the four systems, the two real and the two hypothetical, are described in Table I. The FPGA programming file was obtained from a VHDL description of the processor synthesized with Synplify Premier version D-2009.12 tools. The data memories and the floating-point units were configured using the Xilinx Coregen 10.1.03 tool. Place&Route was performed with the ISE 10.1.03 tools. For the hypothetic systems we have used the execution times obtained in the same host processor as the real ones, the Place&Route frequency results, and the bus transmission times were obtained taking into consideration the performance analysis provided in [9]. For comparison purposes we provide results for a GTX285 GPU from NVIDIA, with 240 stream-cores working at 1476MHz. The serial execution of the algorithm on the host processor is used as baseline.

In this work we use MrBayes version 3.1.2, and as inputs two simulated DNA test data sets of various sizes, generated with Seq-Gen [10] (v1.3.2), and named as 10_1K and 100_5OK, where the first number corresponds to the leaves and the second number to the columns, which are directly related with the amount of PLF calls and the input stream size, respectively. We have also used a subalignment of a real-world phylogenomic alignment of mammalian sequences with 20 organisms, 28,740 alignment columns, and 8,543 distinct column patterns, which is denoted as 20_8543. MrBayes was executed with a fixed random number of seeds and a fixed number of generations to ensure comparability of results.

The Place&Route results obtained for the different FPGA devices are presented in Table II. For the different devices the table shows that for an average DSP occupancy of 79% the frequency is not significantly affected, which means the throughput will scale with the number of PUs implemented.

The total execution times obtained for the three input tests executed on the different platforms are presented in Table III. It is clear from these results that the most affected device is the xc4vsx35, due to the low bandwidth of the PCMCIA, which is only 20 MB/s. As for the other devices we can see an improvement over the serial execution time (CPU) up to 1.5x with the largest FPGA. The low speedup results obtained are again due to the lack of bandwidth required to exploit the high throughput obtained with the proposed

![Fig. 9: PLF computation on FPGA](image)
architecture. This effect is again shown in the results presented in Figure 10. The results presented in the charts are only relative to one input size because the observed results are similar, thus we opted by using the real data set. We can see in the figure that the processing time is always much smaller than the transmission times. Also, the high PLF speedup obtained when comparing the core execution to the baseline machine is drastically reduced when taking into account the transmission times, e.g., for the xc5vfx200T the PLF speedup is at most 223x, and with communications it drops to 18x.

Another issue investigated was the effect of the different scenarios presented in Section IV. These effects were compared to the serial execution and are presented in Figure 11. As there are no state-of-the-art buses that could overcome the speed of the processing core, the Scenario 3 is reduced to Scenario 1, thus in the figure we present only Scenario 1 and 2 for the different architectures. Scenario 2 does not apply to the PCMCIA system as it has only one channel. For Scenario 1 the effect is not very significative showing a maximum improvement of 10% for XC4FX60. Because the PCMCIA is much slower than the processing, the effect in XC4VSX35 is practically null. As for Scenario 2 the improvements obtained over the serial version are at most 66%.

The significant impact of the low bandwidth communication between the host CPU and the FPGA lead us to conclude that using our proposal in a System-on-Chip (SoC) would lead to a notable speedup. For example by using an on-chip soft core or an embedded hard core and an on-chip interconnect bus such as the AMBA from ARM.

VI. CONCLUSION

In this work we used a previously proposed methodology to design hardware accelerators based on the streaming-based computing model via FPGA prototyping. Since stream-based computing reveals most of the program’s concurrency, in order to parallelize it to the GPU, it naturally leads to scalable hardware architectures.

We adopted MrBayes, a bioinformatics program that performs the Bayesian inference of evolutionary (phylogenetic) trees as a case study and we provide a real implementation and experimental results obtained with the method. We proposed the architecture of a full system and we analyze advantages and disadvantages of certain implementation details. The experimental results show that by mapping the stream based program for the GPU into hardware leads to efficient reconfigurable accelerators both in terms of performance and power consumption (typically FPGAs consume less 10x than GPUs). The processing kernels of the stream based programs are implemented in hardware with improved performance and lower cost. The results also show that the core architecture obtained with the proposed approach is scalable. However, the limited bandwidth of the system bus between the CPU and the hardware accelerators significantly reduces the obtained speedups. The PLF results compared to the serial execution show speedups up to 223x while the overall speedup achieved is up to 1.44x. Although the GPU is about 2x faster than the best FPGA, we can still achieve comparable results using only 27% of PUs at 12% of the frequency.

As future work we plan to analyze the improvements achieved by the proposed design using a SOC solution avoiding the effects of the external communication, for example by using a soft or hard core combined with a on-chip interconnect solution such as the AMBA from ARM.

REFERENCES