Politecnico di Torino

Corso di Laurea Magistrale in Ingegneria Elettronica

Tesi di Laurea Magistrale

FPGA-based acceleration of a particle simulation High Performance Computing application

Relatore:
Prof. Luciano Lavagno

Candidato:
Aldo Conte

Tutor aziendale:
Dott. Giuseppe Piero Brandino

ANNO ACCADEMICO 2018-2019

FPGA based acceleration of a particle simulation High performance computing application

Aldo Conte

Acknowledgements

Now that this work is completed it is the time to reflect and express my appreciation to those who supported me along the way. I would like to thank Prof. Luciano Lavagno for allowing me have this great opportunity. I'm pleased to be your student. I also want to thank my tutor Giuseppe Brandino to have me patiently helped along the whole Master Thesis's period.

The last six months have been a very special opportunity for my own personal growth and consequently for my life. "[...] therefore whosoaver heareth these sayings of mine, and doeth them, I will liken him unto a wise man, which built his house upon a rock [...]" Matthew 7, 24. All of this has been possible thanks to me but especially to the whole Background that has shown love for me. Thank you Mum and Dad if it weren't for your continuous support and all the opportunities you gave me, i could have never gotten this far.Thank you Valeria for giving me the great opportunity to really be myself, thank you for your loving patience that you have shown me. I want to thank you Friar Francesco for the "strong hand" with which you guided me.

I would like to thank my dear friends Adriano and their brothers Vittorio, Eleonora and Carmine. Thank you Francesco for your sincere friendship and support. I have to thank my electronic classmates,Deborah, Mattia and all my friends and everyone who has allowed me accomplishing such huge goal.

Last, but certainly not least, a special thanks to all of you of the choir that you have, recently given me several exciting moments and that you made me rediscover how living together is more beautiful. Thank you all!!

Abstract

This thesis studies the possibility to use FPGAs in the world of High Performance Computing (HPC) systems. Such systems currently use hybrid platforms that exploit the huge parallel computing power of GPUs in order to reach very high performances. Nevertheless, GPU-based systems are power-hungry and require a power consumption so large, that running and maintaining such systems could be technologically and economically infeasible. This thesis is in the context of ExaNeSt H2020 project which has the purpose to prototype energy efficient solutions to produce exascale-level supercomputers. Low power consumption requirement will be achieved using a Multiprocessor System-on-Chip, namely an Integrated Circuit including both a an ARM Cortex multi-processor and an Ultrascale+ FPGA: the whole module has been specifically designed with special attention to power consumption.

High performance computer systems are very important in the field of computational science; this thesis investigated the possibility of using FPGA accelerators to offload the compute intensive parts of a Molecular Dynamic simulator. The miniMD is a simple, parallel molecular dynamics (MD) code composed of five different OpenCL kernels (neighbor_ bin, neighbor_ build, force_ compute, integrate_ initial, integrate_ final) designed for studying the physical movements of particles such as atoms or molecules. In the first part of this thesis, each kernel has been studied in order to understand how it could be accelerated using the FPGA portion of the Multiprocessor SoC architecture. After a profiling of the full miniMD application, we showed that the task that identifies neighbour particles for each particle of the system (neighbor_ build kernel) and the one related to the force computation (force_ compute kernel) are the most compute intensive ones and have a prominent role in the total execution time of the application.

Moreover,the most important optimizations were related to the reading and writing of the off-chip DRAM memory by the kernels, moving this data in and out of the on-chip SRAM memory of the FPGA. These optimizations were meant to exploit the burst memory transactions and to improve the access bandwidth between the External DDR memory and the Programmable Logic (PL) therefore reducing the access time of the external memory. It is important to notice that these optimizations were not trivial due to two main aspects: (1) not all DRAM arrays were accessed sequentially by the kernels, and (2) not all the loops that accessed arrays were perfectly nested, thus preventing some burst inference by the HLS tool. In the end all the kernels have been implemented and a single Vivado design of the miniMD application has been obtained for execution on the Zynq Ultrascale+ device.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The world of HPC systems has always had the goal to improve the performance various applications, in numerous application areas. For long time, the increment in performance was obtained by relying solely on the increase clock of newer CPUs. When the clock reached a sort of plateau, the increment in performance was obtained by exploiting the multicore scaling, topically using several homogeneous cores. However, the slow down of Moorse's law required new approaches to be explored. So it was used the idea of offloading some of the workload from the main processor to a co-processor which saw in Graphics Processing Units (GPUs) the optimal candidate, due to their increasing programmable capabilities. Still today they are excellent solutions due to the very high floating point throughput, a favourable architecture for data parallelism and higher memory bandwidth with respect to processors [20].Thanks to the aforementioned skills they find very large usage in the world of High-Performance-Computing (HPC). Such systems are nowadays hybrid platforms that exploit the parallel architecture of GPUs in order to reach very high performances. Nevertheless, GPU-based systems are power-hungry and require a power consumption so large, that running and maintaining such systems could be technologically and economically too expensive.

Moreover, reaching exaflop performance with state of the art technology will have energy and cost requirements so high that would make an exascale system unfeasible. The Exanest project [16] tries to explore the idea to greatly lower the power comsumption while increasing the overall performance. These designs are essentially based on the combination of multi-core processors along with FPGA fabric in the form of accelerator and the whole module has been specifically designed with special attention to power consumption.

In the present thesis, it has been studied the possibility to exploit FPGAs in the world of High Performance Computing (HPC) systems. In particular it has been

investigated the possibility of using FPGA accelerators to offload the compute intensive parts of a Molecular Dynamic code.The miniMD is a simple, parallel molecular dynamics (MD) code composed of five different OpenCL kernels (neighbor_bin, neighbor_build, force_compute, integrate_initial, integrate_final) designed for studying the physical movements of atoms and molecules. In the first part of this thesis work, each kernel has been studied in order to understand which of the kernels could be accelerated into the FPGA of the Multiprocessor SoC architecture. After a profiling of the full miniMD application, it has been demonstrated that the task related to the building of the neighbour particles for each molecule of the system (neighbor_build kernel) and the one related to the force computation (force_compute kernel) are the most compute intensive ones and have a prominent role into the total execution time of the application.

As the following chapter 4 will present, the optimization process required at first, a deep understanding of Vivado_HLS logs and reports, in order to ensure the correct match between the optimizations actually performed by the compiler and the corresponding lines in the code. Since code is not well supported by Vivado_hls, it is possible to run into interpretation problems in analysing logs and reports, in particular regarding which lines of code have been optimized and which not. Once it has been sorted out, it has been possible to notice that the most important optimizations,for what concern OpenCL miniMD kernels, can be be performed in the downloading and uploading processes of the data handled by the kernels, directly into the memory of the FPGA. These optimizations were meant to promote the burst memory transactions and to exploit efficiently the low bandwidth between the External DDR memory and the Programmable Logic (PL) therefore reducing the access time of the external memory. It is important to notice that in the aforementioned optimizations, loops have not been so easy to be speeded up due to their not perfect bounded nature, in fact Viavo_HLS is able to perform the best optimizations on loops that have particular requirements like a fixed upper-bound. If these requirements are not met, the obtained increased performance will not be the best ones. At the end, all the kernels have been merged and a single Vivado design of the miniMD application has been obtained to be later executed onto the Zynq Ultrascale+ device.

Chapter 2

FPGAs in HPC systems

In this chapter it will be analysed the main motivations that support the idea of using FPGAs as good candidates for HPC systems. It will be analysed the potentials and drawbacks of these low power devices with respect to GPUs, going from the performance point of view up to the critical issues about *design flow* that has to be used to create applications with it. Furthermore it will be presented the ExaNeSt European project which aims to practically research solutions for the feasibility of low power HPC systems. Going on, the OpenCL (Open Computing Language) framework will be briefly described in order to clarify its Execution and Memory model. Subsequently it will be discussed the main differences between the SoC platforms versus the PCI-e ones, in order to make clear their different field of usage and the stages in which they have been used during this work. In the rest of the chapter it will be clarified the meaning of High Level Synthesis (HLS) as the technique that makes possible the synthesis of OpenCL Kernels in RTL (VHDL or Verilog) models suitable to program FPGAs. Finally, a very brief focus will be devoted to the software workflow followed to compile and run applications with the Xilinx SDAccel tool.

2.1 FPGA Use Models in HPC and the ExaNeSt Project

FPGAs have different use models in the world of high performance computing thanks to their re-configurable nature:

- **Connectivity use model**: In this mode it is used the connectivity and integration qualities of FPGAs to connect different logic components of a system, that may use unusual logic and connectivity standards and protocols. This is possible thanks to the I/O features available in IOBs of the

FPGAs, designed to support multi-voltage, multi-standard parallel processing connectivity technologies. The supported protocols are different: PCIe (Gen 1/Gen2/Gen3), PCI, Intel's Front Side Bus, Serial Rapid I/O, XAUI, and Intel's Quick Path Interconnect (QPI). Moreover, FPGA I/O blocks are suitable to interface with a variety of Memory types like DRAM and SRAM implementing multiple memory controllers that adjust their data rates and width in dependence of the required performance [2]. A possible example of application of this use model in the field os sensors network is presented in fig.2.1. Here can be noticed as FPGA is able to interface to both sensor

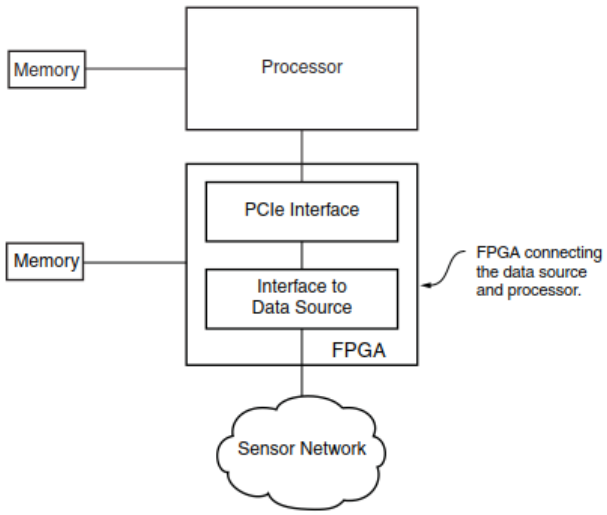


Figure 2.1: Block Diagram of Example os sensors network

Network (thanks to the Interface to Data Source) processor and memory at the same time thanks to the usage of multi-gigabit rate transceivers or high speed interconnects.

- **Fixed function hardware acceleration use model**: In many application specific architectures, it is very common to see large usage of ASICs or other dedicated platforms that are able to ensure an high performance specialized work. In this field FPGAs are used as well and are an highly flexible alternative to application-specific integrated circuits. In this use model the feature of low-latency and high-throughput data processing is useful to accelerate certain *fixed functions* of a predefined workflow of computations like FFT, DCT and so on [2].

- **Software acceleration use model**: This is the second most famous use mode of FPGA in the world of the HPC. The goal is to offload compute

intensive portions of code that natively run on CPU to FPGAs (typically, as described in the following pages, the compute intensive parts of code are specific C/C++ functions that are suitable to be accelerated on parallel or semi-parallel architectures). This is very useful for all that customers that want to accelerate their simulation analysis codes into high performance servers. There are different kinds of field of application like weapon simulations, nuclear waste simulations or molecular dynamic simulations like in this thesis. Following this use model in the following section 2.2 will be discussed as the FPGA is becoming a very good competitor in this application field with respect to GPUs (that instead, are natively thought to work with this way).

2.1.1 ExaNeSt european project

Figure 2.2: Logo of the ExaNeSt project

The building block of the ExaNeSt system the so called Quad-FPGA Daughter Board (QFDB) showed in Fig.2.3. It is composed of four FPGAs Zynq Ultrascale+ MPSoCs, 64 GB of DRAM and 512 GB SSD storage. Each Zynq Ultrascale+ integrates four 64-bit ARMv8 Cortex-A53 cores running at 1.5 GHz and an FPGA module. The QFDBs are then attached to the so called mezzaine boards, which can host 16 of QFDBs. A blade of the system is then composed of 16 QFDBs plus the mezzanine. The configurable aspect of FPGA is used at the blade level to delegate the networking task of transferring data among all QFDBs. As can be appreciated from Fig.2.3 one of the four FPGA is indicated as Network, and has

the role to manage with very high performances the intra-QFDB data transferring and the communication within the Mezzanine among all four QFDBs. At this level, different protocols have been used: low-latency exchanges with Low-Voltage Differential Signaling (LVDS) channels, AXI protocols and high-throughput transmissions with High Speed Serial links (HSS). Another important application of the reconfigurable property of FPGAs has been used to offload the operations of downloading/uploading processed data from NVMe m2 local SSD storage to a further FPGA among the four. This is indicated as "Storage" FPGA in Fig.2.3.

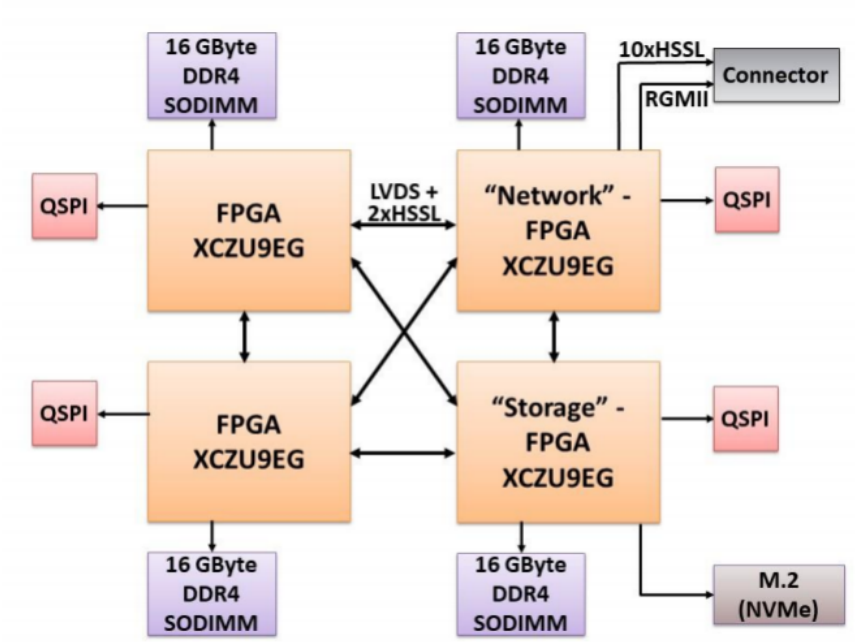


Figure 2.3: The Quad FPGA daughterboard

Finally, 12 blades will be accommodated per row in the rack, and up to four can be fit in the rack itself, for a grand total of 48 blades, 768 QFDBs and 3072 cores maximum per rack. Moreover, ExaNeSt created its own Top-of-Rack switch, to handle communication between different racks. Regarding the scientific software, the ExaNeSt project wants also to provide real HPC applications to validate the system. Among the application to tested on such a system, the Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) is the representative for the field of material science application. Being an application with good scaling on GPUs, it has been considered for the execution of its compute-intensive parts onto FPGA accelerators. In this particular goal can be inserted this master thesis work, with the study of the feasibility of porting the OpenCL kernels of a simplification of LAMMPS application called miniMD onto the last two FPGAs of the QFDB.

In this regard,it will be discussed in next section, how FPGAs are increasing their prominent role in the world of accelerators as good alternatives to the so common GPUs.

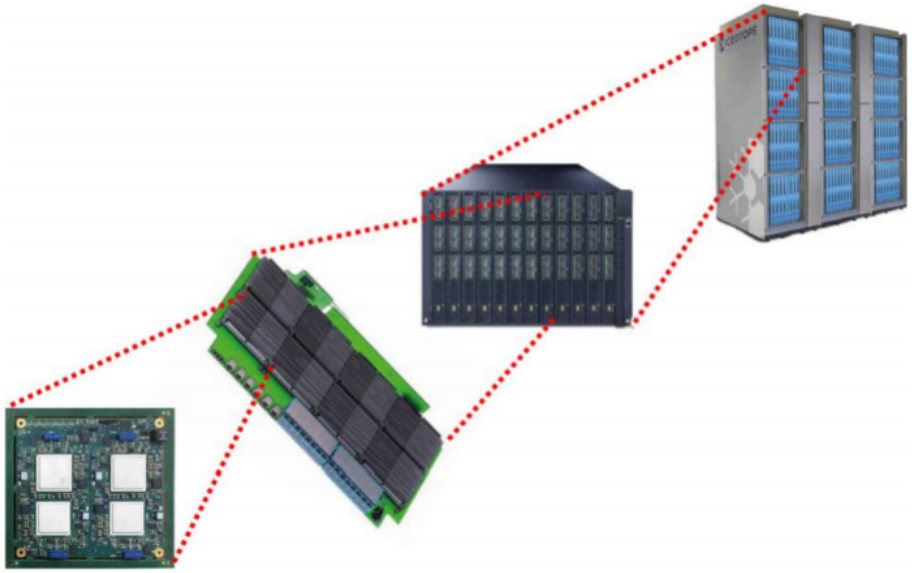


Figure 2.4: The building blocks of the ExaNeSt interconnect and the different Tiers

2.2 FPGA: very good competitor of GPUs in HPC systems

At the beginning the HPC applications were run on processing systems based on general purpose single-core CPU-based systems. Until the early of 2000s ,performance scaled with frequency in accordance with Moore's Law. But the technique of simply scaling a single-core processor's frequency to improve performance has soon met his end. This because if the frequency increases the corresponding power consumption of the device grows quadratically, reaching impractical levels. So the HPC industry has decided to change completely the method to increase performance relying on **multicore** architectures. These last ones have forced developers to adopt a parallel programming model to exploit the multi-core CPU performances. Then, the *Von Neumann bottleneck* and the *memory wall* [17] problems along with the doubt that the performance growth expected by the HPC end user would be delivered, have forced the industry to another big change in order to solve the multicore scaling: the development of CPU-based systems augmented with hardware accelerators as co-processors.

These are heterogeneous architectures that generally consist of a combination of multi-core processors and a variety of hardware accelerators to speed up the execution of data- and compute- intensive applications [2]. In the large world of accelerators, Graphical processing units (GPUs) occupy a very important place in the classification of most powerful accelerators due to their very higher floating point throughput and favourable architecture for data parallelism with respect to processors (namely they have an higher memory bandwidth than processors).

The HPC systems using GPU-based accelerators however, are inefficient in terms of power consumption [13]. So, a good solution can be the replacement of GPU with the modern field programmable gate array (FPGA) devices that, in comparison to GPUs, can reach good processing speed with only a fraction of their operating power consumption [18]. In order to better understand the actual motivations for which FPGAs can be a good solution in the world of HPC it can be useful to briefly remember its structure. The general architectural layout of an FPGA is showed in Fig.2.5.

As can bee seen it is composed of a large array of configurable logic blocks (CLBs), digital signal processing blocks (DSPs), block RAM, and input/output blocks (IOBs). CLBs and DSPs can be programmed to perform arithmetic and logic operations personalized for the specific application to be run. This results in an higher compute efficiency than normal general purpose ALUs in CPUs. In general CLBs and DSPs can perform integer, floating point, and bitwise operations whose results are stored in block RAMs (BRAMs) or in the registers present in CLBs and DSPs.

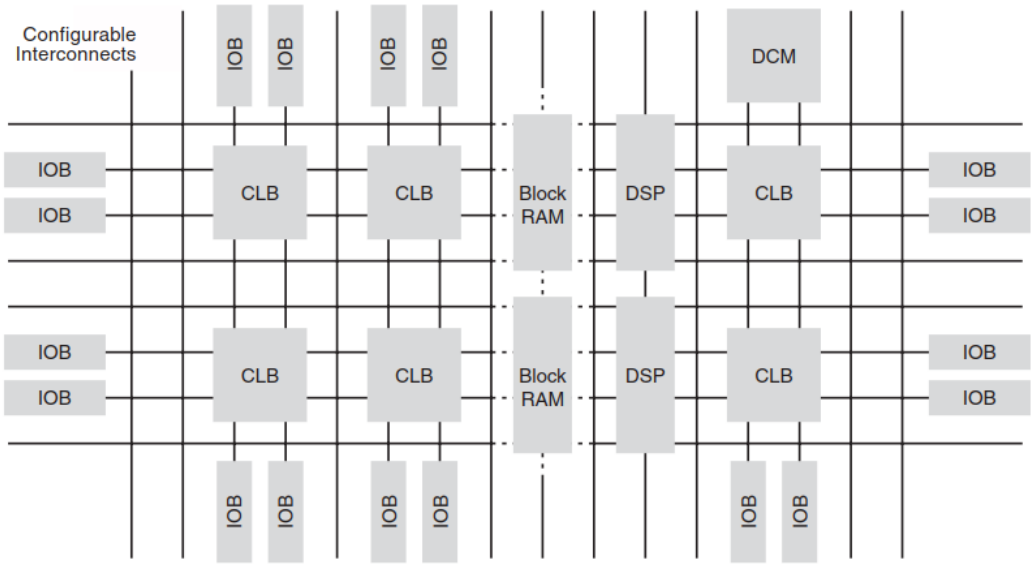


Figure 2.5: High-Level Block Diagram of FPGAs

Since the output of one operator can directly feed the next one it can be noticed as the FPGA architecture give us the possibility to increase both instruction- and data- level parallelism creating arrays of application specific ALUs. Thanks to this reconfigurable property, data can be streamed between operators which can be, in turn, pipelined, increasing performances with respect to normal multicore-systems. Moreover if data can be streamed between operators, this implies the exclusion of problems like processor cache misses. In addition, the *control* logic of the FPGA is configured into the logic itself, thus leading *fetch* and *decode* instructions to be unnecessary. It is important to note that in all of these architectures, not all the components of the device are used, but only ones needed for the application specific operation run. All the other components are off and consequently have negligible power consumptions.

As can be noticed, FPGAs have their "killer feature" in their power-efficiency with respect to multi-core CPUs and GPU-based HPC systems. Actually, FPGAs have an average power consumption of tens of Watts in comparison to the hundreds of watts of multi-cores and GPUs systems. In different tests performed [19] about the execution of the K-nearest neighbor (KNN) algorithm in both a GPU (AMD Radeon HD7950 graphics card) and FPGA (Stratix IV 4SGX530 FPGA from Altera) has been noticed that GPU consumes about 100 W against the 3 W of the FPGA. This demonstrates the effectively energy saving and the consequential preservation of money in cooling systems due to a very much lower emission of

heat with respect to GPUs [18].
There are different reasons for which this can be possible. One of the primary ones is related to the operation's frequency that, in FPGAs is around 100-300 MHz while in high-performance multicore processors is much more higher: 2-3 GHz.

The other reasons, instead, can be found in the advancements in process technology that have allowed an increased logic cell-count and consequently an increase of FPGA logic compute performance [2]. In fact, if we consider that Xilinx FPGAs double their device density from one generation to the next and if we are able to program them in a very "dedicated way" promoting parallelization of executed codes, it can be enlarge the performance gap with respect to multi-core CPUs and reach the same performances of GPUs or, sometimes, overcome them [18]. Moreover big performances are reached especially for low-precision computations.
In general one of the reason for which a GPU is more powerful than an FPGA resides in its **very-high bandwidth** DRAM interface. Since the access to DRAM is a very expensive operation in terms of power consumption, it can be said that this is one of main moments in which GPUs consumes more power. In order to contrast this energy consumption, in the FPGAs is mandatory to try to use a sort of on-chip resources especially for kernel to kernel communication by means of streaming data with on-chip buffers. Following this approach they reach GPU performances in all codes in which is required a very little external DRAM access [18] because the time spent to access an external memory is avoided and also the energy to compute this action is not present. Regardless newer studies have allowed to show as the IOBs shown in Fig.2.5 can be used to support various memory and processor-interface standards [2] as the support for multiple DDR3 memory controllers which would increase the bandwidth to the external memory also for the FPGA case.

The only limitation to a very large usage of these devices in HPC is related to their *cost* that is not so affordable and has historically limited the usage to a very narrow set of HPC applications. But now, the direction that all the major companies of HPC are undertaking in the use of this device would increase its production and consequently reduce its cost [18]. The overall performances of newest FPGA are drastically grown due to the increase of the logic cell count and speed and thanks to all the architectural enhancements. As can be noticed in fig.2.6, to an average improvement of 25% of clock frequency per year corresponds a Logic compute performance increase (clock frequency increase x logic cell count increase) of 92X on a decade. In the same time the price per logic cell has experienced a decrease

of almost 90% [2].

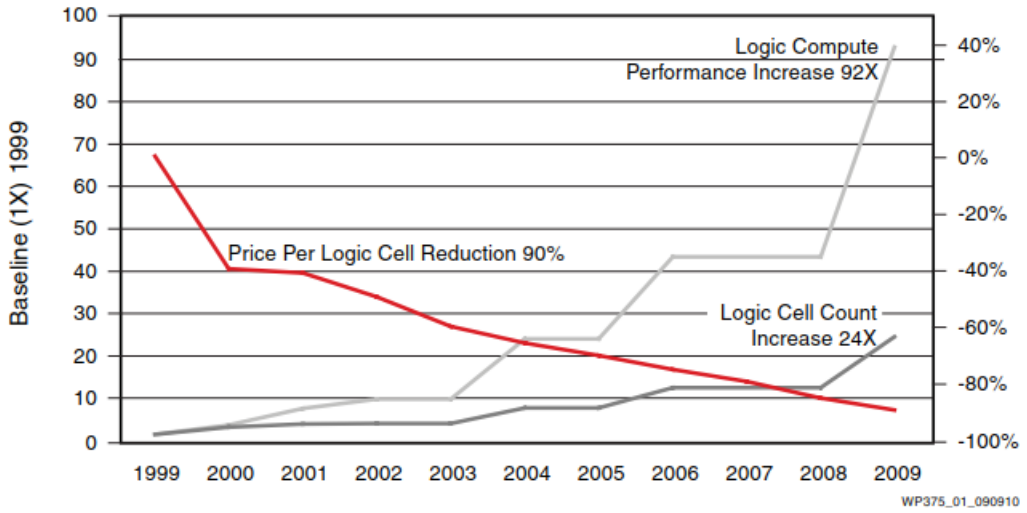


Figure 2.6: Historical Advancement of FPGA Technology

All of these solutions and considerations are not feasible if we don't face with the main obstacle in the utilization of FPGAs for acceleration: the complexity of programming them. In-fact the FPGAs are in general programmed by means of hardware description languages (HDL) such as Verilog or VHDL in the field of hardware Design. If we consider that FPGAs used in HPC would averagely be programmed by Scientists that don't know in very deep way the world of HDL, this becomes a very big problem. This hurdle however can be overcome by a technique called **High-Level-Synthesis** that enable designers to program FPGA using high-level languages like C/C++, System C or, like in our case, in OpenCL. So, in this way accelerate portions of codes for FPGA accelerators becomes much more affordable and not restricted only for hardware engineers.

Before explaining the concepts of High-Level-Synthesis, the Software workflow and the hardware platforms used, it is useful to better introduce the OpenCL framework.

2.3 OpenCL

Open computing language (OpenCL) is a single parallel programming model, based on C (developed by Khronos™ Group). It can be seen as a system-level abstraction

for all hardware platforms that supports this standard. It is suitable for all types of multi-core and heterogeneous parallel compute platforms and the Xilinx company collaborates actively with the Khronos Group in order to allow a better and easier execution of programs into the Xilinx FPGA devices [15]. The OpenCL can be used in different devices and is not restricted to particular platforms as CUDA for Nvidia GPUs and guarantees the functional correctness of its code executed on different platform. Nevertheless the performance portability is not guaranteed and instead is strictly dependent on the specific architecture [15] on which the code is run. So the performance of an OpenCL code will depend on how we are able to adapt the coding style to the architecture of the hardware platform on which it will run.

OpenCl coded-routines, as stated before, are particular portions of a general scientific code (*host-code*) that are actually compute intensive. So these portions of code are functions (kernels) that, in order to increase the performance of code and so reduce its time of execution, are translated from the original C/C++ language into OpenCL in order to allow its execution into accelerators (devices) as GPUs or FPGAs (actually FPGAs can run both C/C++ and OpenCL codes). Basically the OpenCL version of these functions are programmed in order to allow the largest possible parallelization execution. The host code has the role to handle and synchronize the execution of the kernels. Moreover it has to send and retrieve the data processed by the accelerator devices from the memory.
In order to appreciate the programming model of OpenCL codes it is mandatory understand the OpenCL Memory model and the Execution Model. It is very important to make clear this because on these concepts will be given, in the next chapters,the main motivations for which OpenCL codes in FPGA run differently with respect to GPUs.

2.3.1 The OpenCL Execution model

A view of the OpenCL platform is shown in Fig.2.7. We can notice the presence of the CPU that is in general a multi-core processor on which the *host-code* is run. The different kernels are executed, instead, on the device that is composed of different *Work-items* (Processing Elements) grouped into *Work-groups*. Many work-groups can be launched into a single *Compute-Unit*.
The same kernel can be run by all the work-items into a single compute unit and also by other compute-units. In other words: in order to further increase the parallelism of the application the same kernel function can be executed by all the work-items within all the work groups of a compute-unit and also by different compute-units. In order to uniquely identify each work-item from the other ones in the compute unit, each PE has a *global-ID*. Moreover, it is important that each

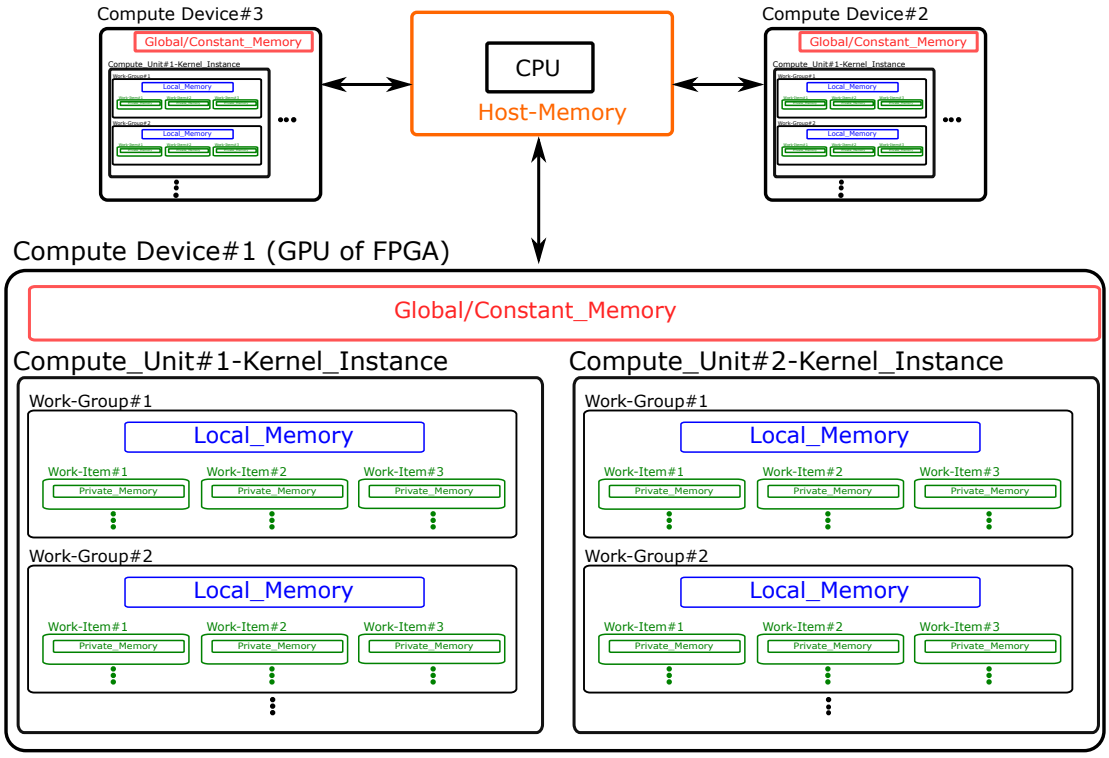


Figure 2.7: openCL platoform and memory model

work-item can be addressable with respect to all the other work-items in the same work-group. so at this scope it has also another identifier :the *local-ID*. Of course we can have also multiple devices (Fig.2.7) that share the same Host. This is another option to further increase the parallelism level of the application or to run different kernels at the same time.

2.3.2 The OpenCL Memory model

As far as memory-model concerned we can notice from Fig.2.7 an Host memory used directly by the CPU to save local variables and then all the other memories in the device. In this last ones it can be seen the global/constant memory used to save data shared between the CPU and the accelerator device. In fact this can be read from and written to by both the host and the device [3].The Access to this memory by means of single Processing element is very slow because is usually an external DRAM and it is the largest into an OpenCL region. So other faster memories are present: the *local-memory* that is shared among all the processing elements (work-items) into the same work-group. It usually resides in on-chip SRAM and can be

accessed 100x faster than global/constant memory. Finally we have the *private-memory* exclusively thought for each work-item to save local variable useful only for it. This is implemented by means of local registers composed of LUT or BRAM and it can be accessed faster than local or global memory.

2.4 System types in High Performance Computing and FPGA platforms descriptions

There are two main different system types in HPC. The first one is related to High-performance servers. These are usually used in big supercomputers suitable to run data intensive applications. For this reason they need of a big amount of power and bandwidth to global/constant memory. In these systems FPGA's architecture provides different advantages from the interface point of view, as the support for PCIe® Gen1/Gen2/Gen3 protocol suitable to communicate with processors through very high bandwidth and speed. An example of such system can be seen in fig.2.8.

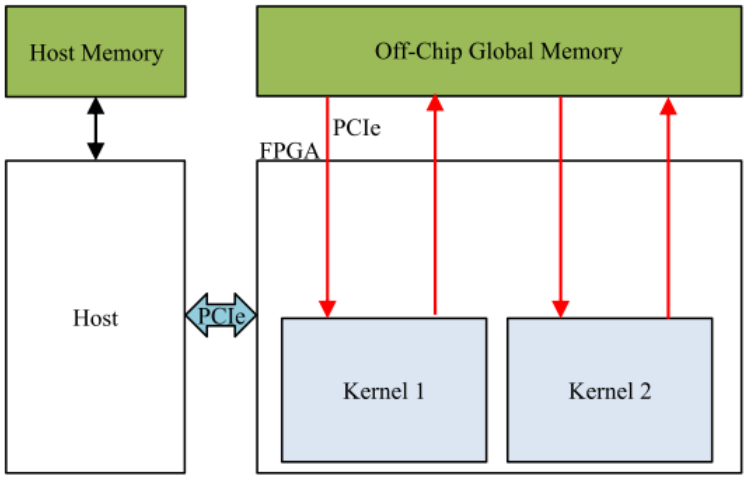


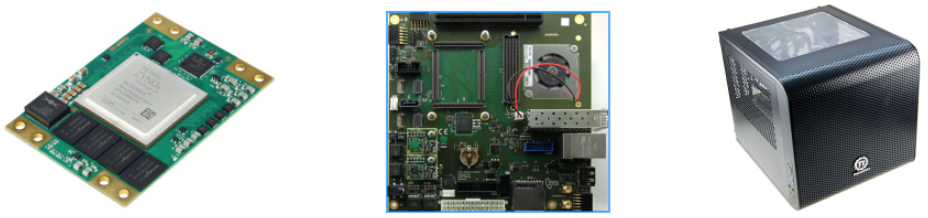
Figure 2.8: Traditional structure of an High-performance server type

As can be noticed the communication between the processors and the FPGA platform occurs across the PCIe® bus.

The second system type, is instead related to Embedded applications. These are the high-performance embedded systems. Such architectures don't rely on PCIe® to perform the communication between a processor and the FPGA because they

are usually based on System-on-chip architectures used for heterogeneous embedded systems applications.
The PCI-express based system FPGA is not soldered on the same chip where are also the other components (CPU, peripherals, etc..) but is externally connected by means of the PCI-express high bandwidth connection. By contrast System on chip houses and integrates microprocessor, graphics,memory, memory interfaces and also advanced peripherals like FPGA accelerators for specific purposes into a single electronic substrate. These are mainly thought for embedded systems improving performance and reducing power consumption [4] and are the ones chosen by the ExaNeSt project to build the aforementioned supercomputer.

During the prototyping phase of the board Tier1, the ExaNeSt project has employed the *Trenz Starter Kit 808* as testbed. This is composed of different components:a Trenz TE0808-03 module equipped theXilinx Zynq Ultrascale+ XCZU9EG MPSoC and " GByte of DDR memory (see Fig.2.9[a]) and the Trenz TEBF0808-04A carrier board, which mounts the Trenz TE0808-03 module shown in Fig.2.9[b]. Then the carrier board is hosted into an ATX case (see Fig.2.9[c]).



(a) TE0808-03 module (b) Carrier board (c) Board case

Figure 2.9: Trenz Starter Kit 80

Nevertheless, during this thesis work, all the tests have been performed on PCIe based FPGA systems. In particular AWS services have been utilized with Virtex Ultrascale+ FPGAs.
As it has been said before, the *unit* of application is a Zynq UltraScale+™ MPSoC (see fig.2.10). The XCZU9EG SoC combines two main parts: a Processing system and the Programmable Logic. The processing system (PS), in general, contains all the fundamental components to run the Host-code of an HPC application and also other peripherals like GPUs, High-Speed Connectivity, Configuration and security unit, Real-Time processing unit. In particular we have:

- An Application Processing Unit *APU* equipped of quad-core ARM Cortex-A53 with 32KB L1 Cache and 1MB L2 Cache.The ARM cores embed the advanced SIMD technology NEON in order to support the Single/Double precision Floating Point computing [5]. This multi-processor is usually used to run general purpose tasks or, like in our specific case, to run an *host code*.This also controls the functional blocks built in the programmable logic by setting/reading specified signals in a reserved memory location.
- *Real Time Processing Unit (RPU)* ARM Cortex-R5 used for all applications that need to respond in deterministic time.
- A GPU ARM Mali-400 MP2 with 64KB L2 Cache working with a frequency of 664 MHz
- *Dynamic memory Controller ()DDRC)* used to allow the communication between the PS and PL.

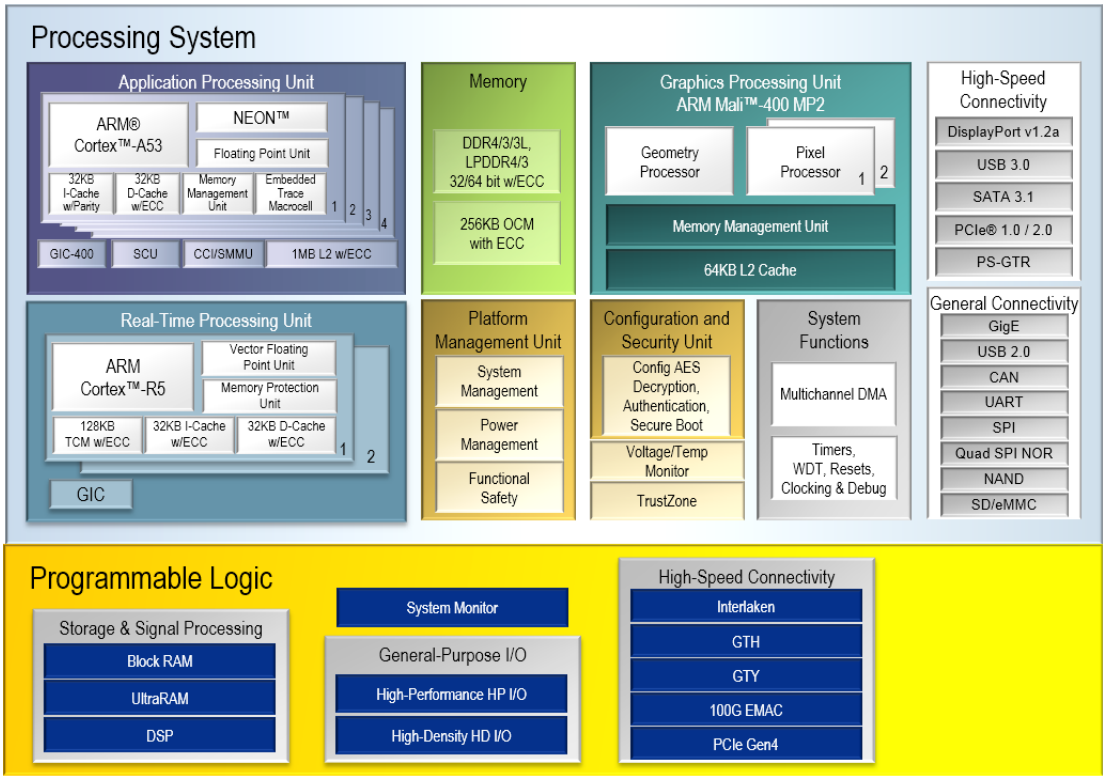


Figure 2.10: Xilinx Zynq ultrascale+ EG

The programmable logic contains the FPGA used in our case to accelerate some kernels of the molecular dynamic (miniMD) code. The connection between the

PL and PS in the Zynq ultrascale+ device counts a total of 12 among Master and Slave interfaces. The master interfaces are the M_AXI_HPM0_LPD and M_AXI_HPM[0-1]_FPD shown in fig.2.11. These are used by the processors in PS to control the PL by writing/reading the control status registers in a reserved memory location for each instance of kernel (functional_block). In other words each functional block has some control registers in memory that are used by the ARM processor to see when the computation is started/finished and so launch the command to transferring data from/to FPGA device (by means of the slave interfaces).

The Slave interfaces are used by the PL to retrieve/send data to PS and in particular to the DDR memory subsystem. These are S AXI_HPC[0-1]_FPD, S_AXI_HP[0-3]_FPD and S_AXI_LPD in Fig.2.11. In this thesis only the **S_AXI_HP[0-3]_FPD 128bit-width** interfaces have been used beacuse are related to the specific function to exchange with high performance large data between DDR memory and FPGA. It is important to note as the bandwidth between external DDR Memory and PL is, in case of SoC, lower than PCIe based systems in which each AXI interface can have up to 512 bit-width buses.

AXI protocol and PS-PL Communication

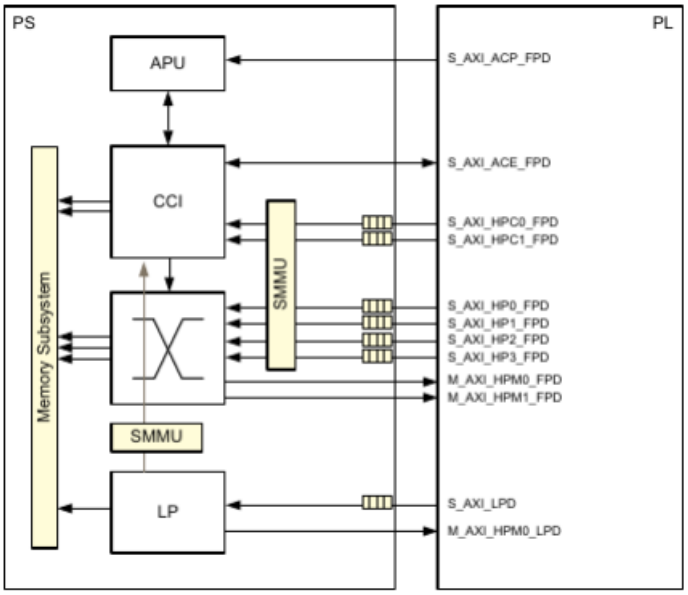


Figure 2.11: AXI Interconect

In the Fig.2.11 above it can be noticed all the connections between PL and PS based on the AXI protocol: an on chip interconnect protocol used to manage and connect different functional units into a System on Chip. There are three types of AXI-4 interfaces [6]:

- *AXI4/AXI4* for high performance memory-mapped transactions.Memory mapped transactions means that data transfers are performed over a memory space mapped by addresses. It is used in our case to allow the transfer of data between all the functional units (implementing the kernels) in the FPGA and the External DDR memory [6].
- *AXI4-Lite* for simple, low-throughput, single memory mapped transactions. It is ised in our case to control and set the status of the functional blocks in the FPGA (PL) [6].
- AXI4-Stream for high performance stream (not mapped) transactions that enable the stream transfer of data. In other words stream transactions do not use the mapped memory space [6].

2.5 Software workflow

As stated before in section 2.2, in order to ensure the translation of OpenCL code into Register-Transfer-Level (RTL), the High-Level-Synthesis technique is largely used. This can be defined as "an automated design process that interprets an algorithmic description of a desired behaviour and creates digital hardware that implements that behavior "[7]. In our case the "algorithmic description" is about the compute-intensive functions (kernels) written in OpenCL that will be directly transformed into a RTL implementation. This RTL version is generally obtained in VHDL or Verilog languages.
The implementation of kernels on FPGA device includes the high-level-synthesis step into a more general workflow performed, in this thesis-work, by means the software's tools of Xilinx. Each step will be described in the following subsections.

2.5.1 Vivado HLS

Vivado HLS is the Xilinx Software related to the synthesis of C/C++/SystemC and OpenCL codes into VHDL- or Verilog-described digital circuits. This automated process consists of three main stages:

- *Scheduling*:During this step are determined which operations have to be performed in each clock cycle considering different factors like the clock frequency and the latency of the operation itself. In dependence of the of the

clock period of the targeted FPGA the operations will be sheduled in one or more clock cycles [8].

- *Binding*: It is used to select which hardware resource has to be implemented for each scheduled operation. in order to perform this operation in an optimized way Vivado HLS needs to know the output target device [8].

- *Control logic extraction*: A finite state machine is implemented in order to generate the control logic used to regulate the operations before scheduled [8].

After these steps Vivado HLS is able to create an optimal implementation of the original code based on the default behaviour and constraints. At the end of this step the user can control if the design generated meets its requirements reviewing the performance metrics reported in the synthesis report generated by high-level-synthesis. This report contains different informations about the Area used by the hardware resources implemented, the Latency of the function to compute all output values, the number of clock cycles brfore the function can accept new input data (initiation interval) and so on. If these metrics are not sufficiently good the user can decide to use **optimization directives** to modify and control the default behaviour of the internal logic and I/O ports of the RTL generated.
This is a very important feature that guarantees to the FPGA acceleration development, a further degree of freedom to ensure a strong increase of performance in the execution time of the kernel. The optimal results obtained by adopting a "personalized programming style" into writing OpenCl codes with the right **optimizations directives** are able to guarantee performances sometimes even gather than that of GPUs. Nevertheless, this topic will be better analyzed in Chapter 3 where will be presented all the possible optimizations available for OpenCl codes and the effort needed to generate optimized compute architecture for FPGA rather than GPUs.[18]

After the execution of Vivado HLS, we are able to extract from the output files a **synthesized functional block** representing the kernel as a black box with its output and input ports in accordance with its hardware description language. Moreover C control driver are emitted and are used by the programmable system to control the functional block in the PL.

An example of a functional block implementing a vector addition kernel can be seen in the following image (Fig.2.12):

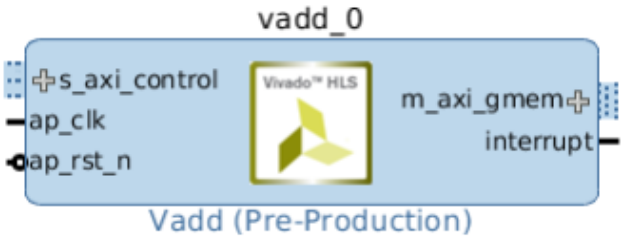


Figure 2.12: Vector addition kernel

2.5.2 Vivado Design Suite

After that the kernel has been synthesized, the design environment Vivado IDE is used to merge the functional block with the rest of greater design including the FPGA. An example of an implemented design is in Fig.2.13

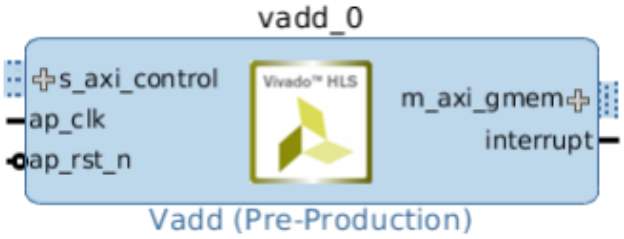


Figure 2.13: Vivado IDE project Design

After that the whole design has been completed, a *logic synthesis* is performed translating the RTL-level design of each functional block into a netlist of primitive FPGA logical elements connected between them. This phase is called *synthesis*. Then *implementation* step places and routes the functional blocks in the FPGA device using the previous generated netlist. During this phase the focus is on meeting the logical, physical and timing constraints given by the user during the set-up of the project. Finally the configuration design of the FPGA resources is transformed into a **bitstream** that is a binary representation of the resources and the way through which they are connected to each other. This binary file along with the functional block driver generated by vivado HLS and other platform files are grouped in an Hardware Platform Specification File (*.hdf file) used flash the FPGA fabric.

Overall, we can say that, the first action that has to be performed when we want to speed up the execution of an application, is to select the functions (kernels)

which need to be accelerated and process each of them by means of Vivvado HLS. Later, when the functional blocks of each kernel has been generated we have to use Vivado Design suite to aggregate them in a toatl final design. This kind of workflow can result a little bit slow in case the number of synthesized kernels becomes larger. So, in these cases the **Xilinx SDAccel** and **SDSoC** tools helps to boost this process. As a matter of fact, these tools do nothing but analyse the Host Code of an application, identify all the C/C++/OpenCL kernels and process each of them with Vivado HLS. At the end the tool is able to merge all the kernels together automatically recalling Vivado Design Suite and to generate the final bitstream useful to flash the FPGA.
During this work has been more convenient to use SDAccel rather than SDSoC. The difference between them is very minamal and essentially consist on the different platform architecture target. In fact, while SDAccel has been thought for FPGAs in High performance server systems (as stated here 2.4) SDSoC, insted, is suitable for FPGAs in High performance embedded systems. Nevertheless, during the whole development phase of the thesis work , it has been used SDAccel but at the end SDSoC has been used to generate the right files for the Zynq Ultrascale+ (this being a platform falling within the category of embedded architectures).
Further details about the functioning of SDAccel will be given in the following subsection.

2.5.3 SDAccel Build process

Xilinx SDAccel software tool is a particular environment offering an optimized compiler to analyze Host Codes and cross-compiler to menage Kernels for FPGA. The SDAccel build process can be viewedd in the following Fig.2.14. The build process is essentially based on a standard compilation and linking processes performed for both Host-Code and FPGA Kernel. As can be seen from the image above, at first, the Host-Code (topically written in C/C++) is compiled and then linked by means of a variant of the GCC compiler called *xcpp*. The same stage in SDSoC is performed by a different variant called *sds++*. At the end of this stage the host application executable (.exe) is generated. Then each FPGA kernel (topically written in C/C++ or OpenCL C as in our case) in dependence of the *build target* selection will be compiled and linked by means of *xocc* compiler (the same compiler is used in SDSoC). During the compile stage Vivado HLS is run generating for each kernel,one object file with **.xo* extension . Follows, the linking stage during which Vivado Design Suite is run. It merges all the *.xo files previously generated with the FPGA platform files, producing at the end, the FPGA binary file with extension **.xclbin* [9]. It is important to note that, while in GPUs, the compiling of kernels is performed at running time, in FPGAs this doesn't happen. FPGA design offers much more customizations with respect to GPU design, but,

by contrast requires synthesis , place and route times much more longer. As a matter of fact the host code has to read the xclbin binary file that has already been produced before during compilation and linking phase. This is called *Offline compilation approach* [18].
There are different types of FPGA binary files generated in dependance of the build target selected. SDAccel provides different build target: **Software Emulation**, **Hardware emulation** and **System**. It is much more useful to analyse their utility and the moment in which they have to be used, by means of the following methodology workflow (see Fig. 2.15).

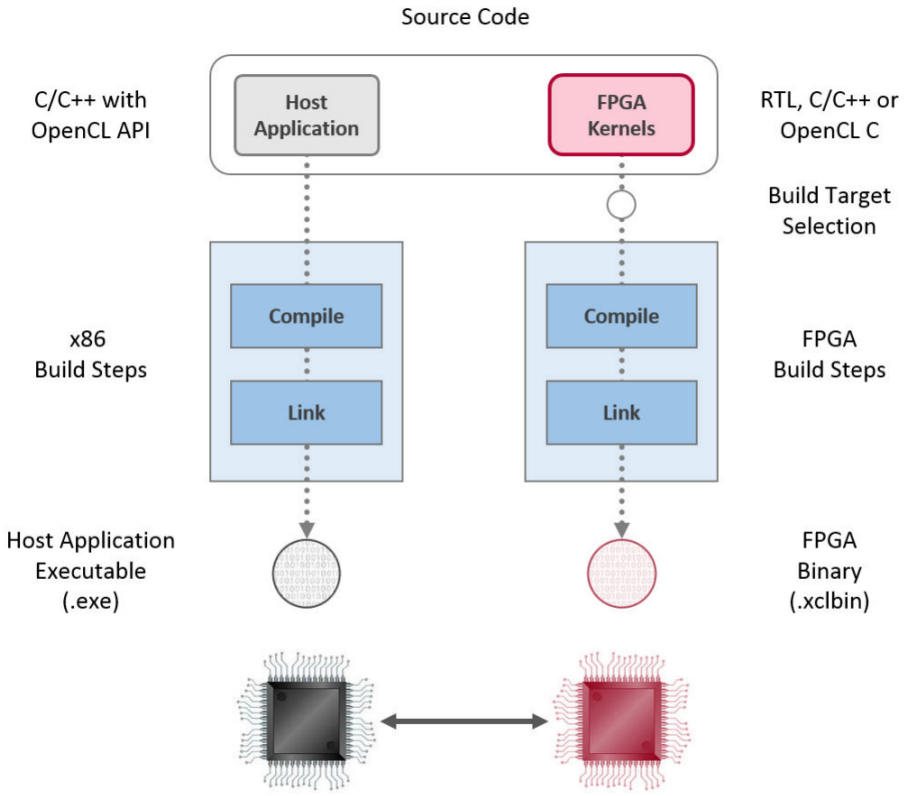


Figure 2.14: Host-Code/kernel build process in SDAccel

The first goal to be reach is to guarantee the functional correctness of the synthetized OpenCL code by means of a CPU or Software emulation. This means that both host-code and kernel code are compiled to run on the x86 processor. This phase is useful to solve syntax issues and perform source-level debugging [9]. In the meanwhile a system estimate report has been generated by Vivado HLS and this document reports an early idea of the performance and resources related to each kernel. In order to obtain much more detailed and cycle-accurate view of the

kernel the hardware emulation has to be executed.During this stage an RTL simulation model of each kernel is generated in order to ensure more accurate results. Of course, if after this stage the performances obtained do not meet the required ones, further optimization directives can be applied to the original OpenCl kernel code. So, all the stages performed up to this point have to be repeated.

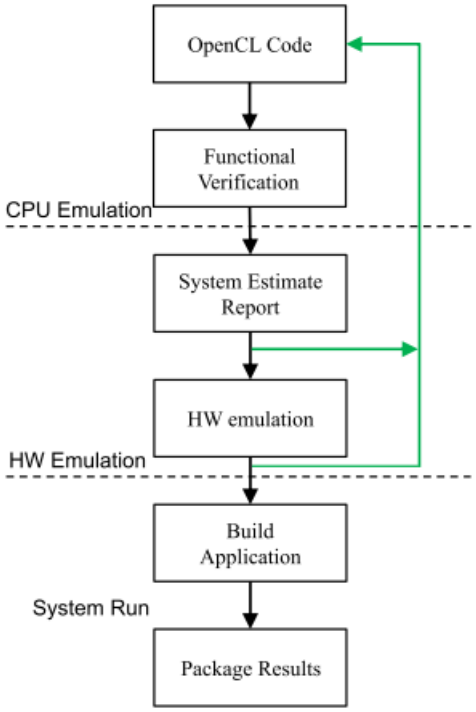


Figure 2.15: SDAccel based FPGA design methodology flow.l

If, instead, the performances are satisfactory, it can proceed to the build system step to link all the generated custom units with the processor and DDR DRAM interfaces related to the target hardware [18]. finally the package to flash the FPGA platform is generated.

Both SDAccel and SDSoC can be executed by means of the GUI Interface or my command-line. During this Thesis work the command-line approach has been preferred.

Chapter 3

MiniMD application and Kernels presentation

In order to give actual benchmark simulations to test the performance of the exascale supercomputer, the ExaNeSt project has decided to use, among other simulation packages, the LAMMPS classical molecular dynamics simulation package. Nevertheless, with the purpose of promoting an easier porting of the OpenCL kernels of LAMMPS to FPGA has been decided to use a simplified mini-app extracted from LAMMPS and called **miniMD**. This, having a limited collection of potentials and minimal post-processing, is suitable to allow to focus onto the possibility on finding optimizations aimed to properly speed up the execution time of the above kernels. Following this solution further advantages can be found in the matching between the kernels' structure and the architecture of the given Zync Ultrascale+ MPSoC.
So this chapter will, at first, analyse the molecular-dynamic (MD) numerical simulation with a very basic explanation of the physical concepts on which it is based. Then, each miniMD kernel will be presented paying attention to explain their behaviours and how they interact with each other.

3.1 Molecular Dynamics

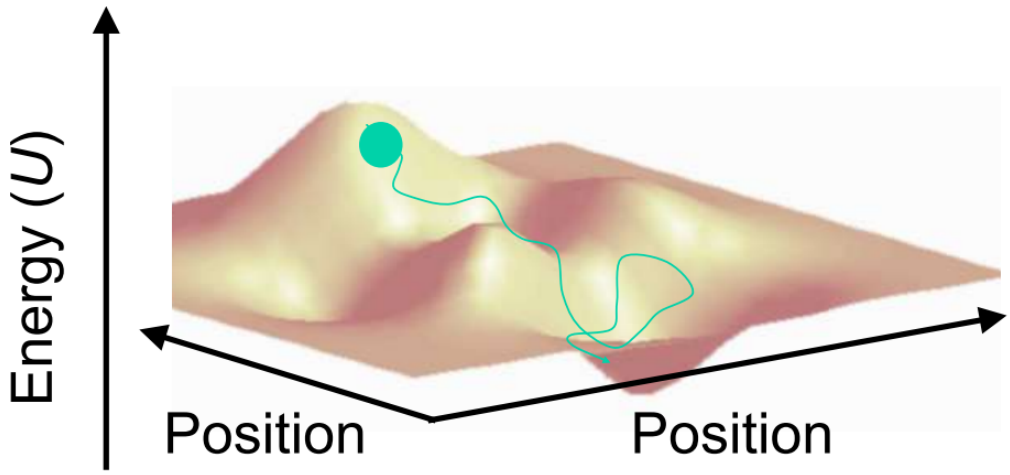


Figure 3.1: Energy function versus space coordinates

Molecular dynamics (MD) is a technique for computer simulation of complex systems, modelled at the atomic level. It is used for studying the movements of particles (atoms or molecules) that can interact between each other for a fixed period of time. Particles interact through forces that can be calculated using interatomic potentials or molecular force fields. In general, assuming a given potential energy function and assuming that every atom has a unique position and velocity (see Fig.3.2), the molecular dynamic simulation mimic what atoms do into the space under the presence of the forces experienced by any other atom in the same space. These forces can be computed thanks to the energy function itself (see Fig.3.1).

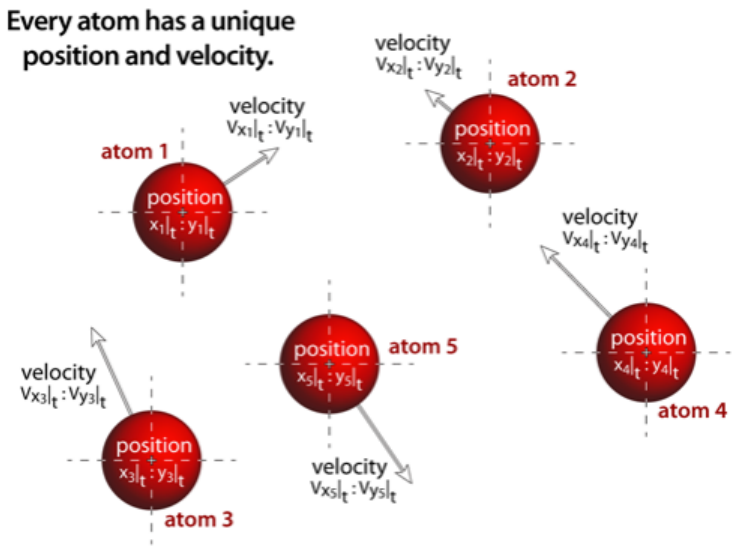


Figure 3.2: Group of atoms with their coordinates and velocities

In such complex system of particles, the interatomic forces and the atom's potential energies lead atoms to move. Since, molecular systems typically consist of huge number of particles, it is impossible to determine such trajectories and the general properties of molecules, analytically. Given the total force acting on each particle, the numerical solution of Newton's equations provides the dynamic evolution of the system. [10].
The potential energies, are often calculated using interatomic potentials or molecular mechanics force fields [10]. In particular it can be computed as a sum of the interactions between the atoms of the system. One of the common choice is the **pair potential** for which the total potential energy is computed as "the sum of energy contributions between pairs of atoms"[10]. A good example of such pair potential is the **Lennard-Jones potential** useful to calculate the **Van-der Waals forces** among all particles.

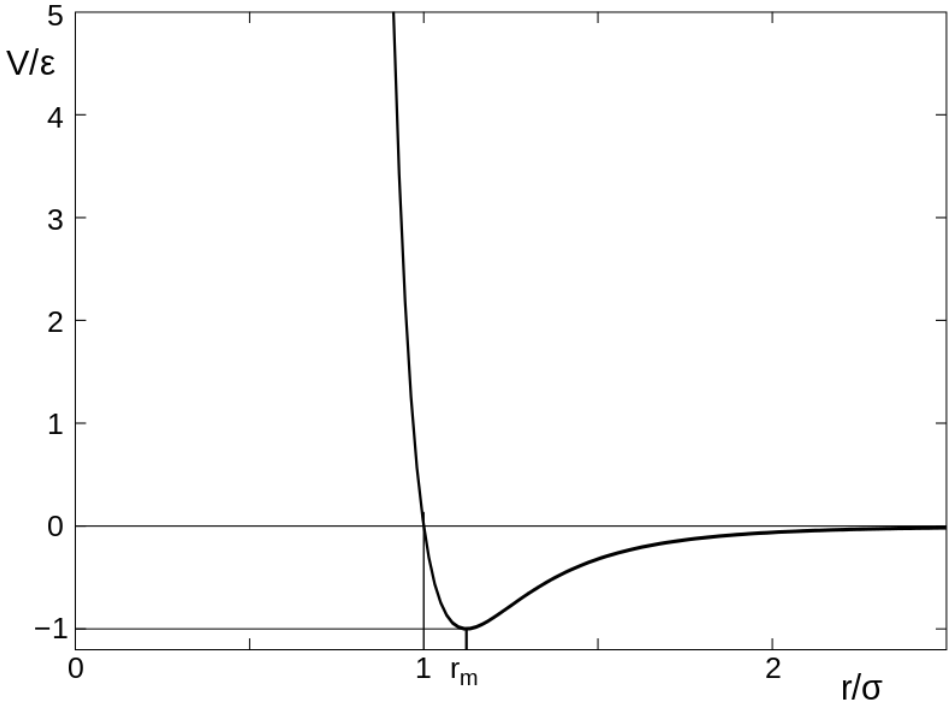


Figure 3.3: Strength versus distance for the Lennard-Jones potential

The Lennard-Jones potential is a mathematical model that approximates the **interaction potential** between a pair of neutral atoms or molecules with given spatial coordinates and velocities that uniquely identify them in the system [10]. The most common expressions of the L-J potential are:

$$V(r) = 4\epsilon\big[(\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6}\big] = \epsilon\big[(r_m/r)^{12} - 2(r_m/r)^{6}\big] \tag{3.1}$$

where $V(r)$ is the potential energy, ϵ is the depth of the potential well, σ is the finite distance at which the inter-potential is zero and r is the distance between two particles. Instead r_m is the distance at which the potential function has the value $-\epsilon$ and is related to σ by means of the following expression : $r_m = 1.122\sigma$. As can be noticed the L-J potential is the result of due main terms. The first fraction is elevated to the power of 12 and indicates the repulsive term describing the Pauli repulsion at short ranges due to the overlapping of electron orbitals. So the inter-molecular force between two particles when r is very small and so at short distances is repulsive. The second term is the one with power of 6 which is the attractive long-range term, describing the attraction at long ranges due to van der Walls forces.
In general the L-J potential is particularly suitable for simulations about the calculation of properties of gases and especially noble gas atoms. The L-J potential

is very simple and for this reason is largely used in computer simulations where is preferred to use this simplified expression:

$$V_{LJ}(r) = (A/r^{12}) - (B/r^{6}) \tag{3.2}$$

where $A = 4\epsilon\sigma^{12}$ and $B = 4\epsilon\sigma^{6}$.

Now, given the (3.2) expression of the potential, the force between two particles can be computed as

$$\mathbf{F}_r = -\nabla V(r) = -\frac{d}{dr}V(r)\vec{\mathbf{r}} = 4\epsilon\Big(12\frac{\sigma^{12}}{r^{13}} - 6\frac{\sigma^{6}}{r^{7}}\Big)\vec{\mathbf{r}} \tag{3.3}$$

The total force acting on a particle is obtained by the summation (integration) of (3.4) computed for each particle neighbours. The particle neighbours are all molecules within a given (not so high) distance of the considered particle (see Fig.3.4).

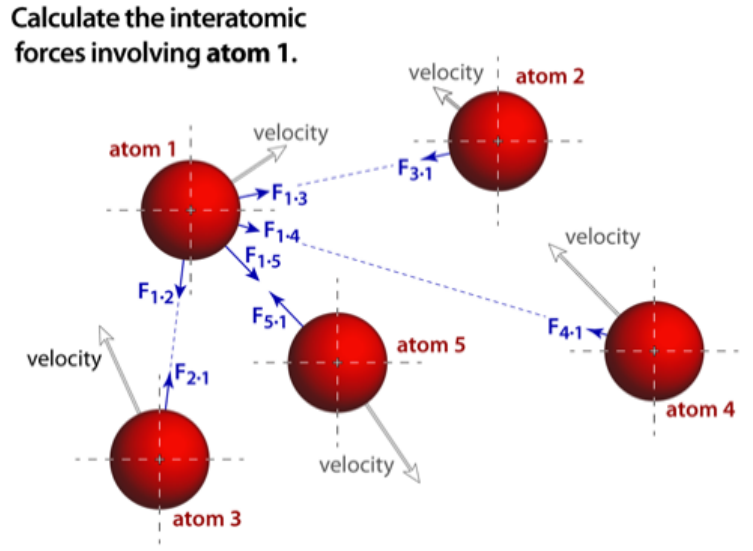


Figure 3.4: Computation of total force acting on atom1 due to the presence of its neighbours

After the computation of the resultant force applied by all the neighbours onto one single particle, this will change its spatial coordinates and its velocity. Itself, with its movement will cause the change of the total force applied to one or more of its neighbours updating also their coordinates and velocities thanks to Newton's laws of motion. In fact, the Newton's second law asserts that the force applied

onto an atom is proportional to its mass m and to the acceleration a consequently imposed on it. Now if we consider that the velocity is the derivative of position ($\mathbf{x}$ indicates the coordinates of the atom) and acceleration is the derivative of velocity we can write the equations of motion as:

$$\frac{\mathbf{dx}}{dt} = v\frac{d\mathbf{v}}{dt} = F(\mathbf{x})/m \tag{3.4}$$

As can be seen this is a system of ordinary differential equations. In order to solve this, numerical solution is used implementing the following molecular dynamic algorithm:

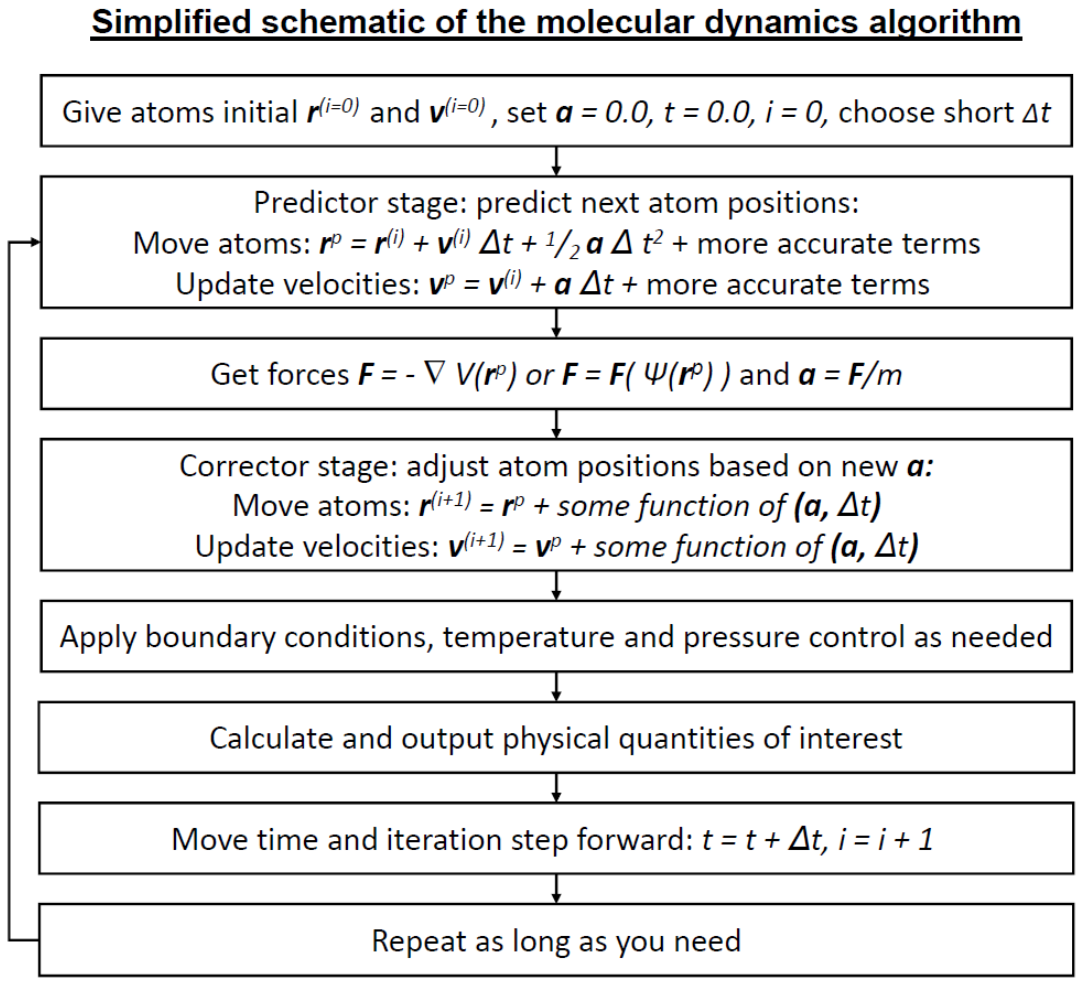


Figure 3.5: A simplified description of the standard molecular dynamics simulation algorithm

3.2 Leapfrog Algorithm and miniMD kernels' description

Actually, the algorithm expressed in Fig.3.5 is nothing more than the application of **Leapfrog integration** algorithm that is a second order method for updating position and velocity following these equations:

$$a_i = F(x_i), \tag{3.5}$$

$$v_{i+\frac{1}{2}} = v_i + a_i \frac{\Delta t}{2}, \tag{3.6}$$

$$x_{i+1} = x_i + v_{i+\frac{1}{2}} \Delta t, \tag{3.7}$$

$$v_{i+1} = v_{i+\frac{1}{2}} + a_{i+1} \frac{\Delta t}{2} \tag{3.8}$$

where x_i is the position at step i, $v_{i+\frac{1}{2}}$ is the velocity, or the first derivative of x, at step $i + \frac{1}{2}$, $a_i = F(x_i)$ is the acceleration, or second derivative of x, at step i, and, finally Δt the size of each time step.

MiniMD is a simple and lightweight C++ code composed of five main kernels:

- **neighbor_bin**
- **neighbor_build**
- **force_compute**
- **integrate_initial**
- **integrate_final**

What is important to understand is the way through which these codes interact among each other into the miniMD simulation code. A summary scheme of such algorithm can be viewed in Fig.3.6

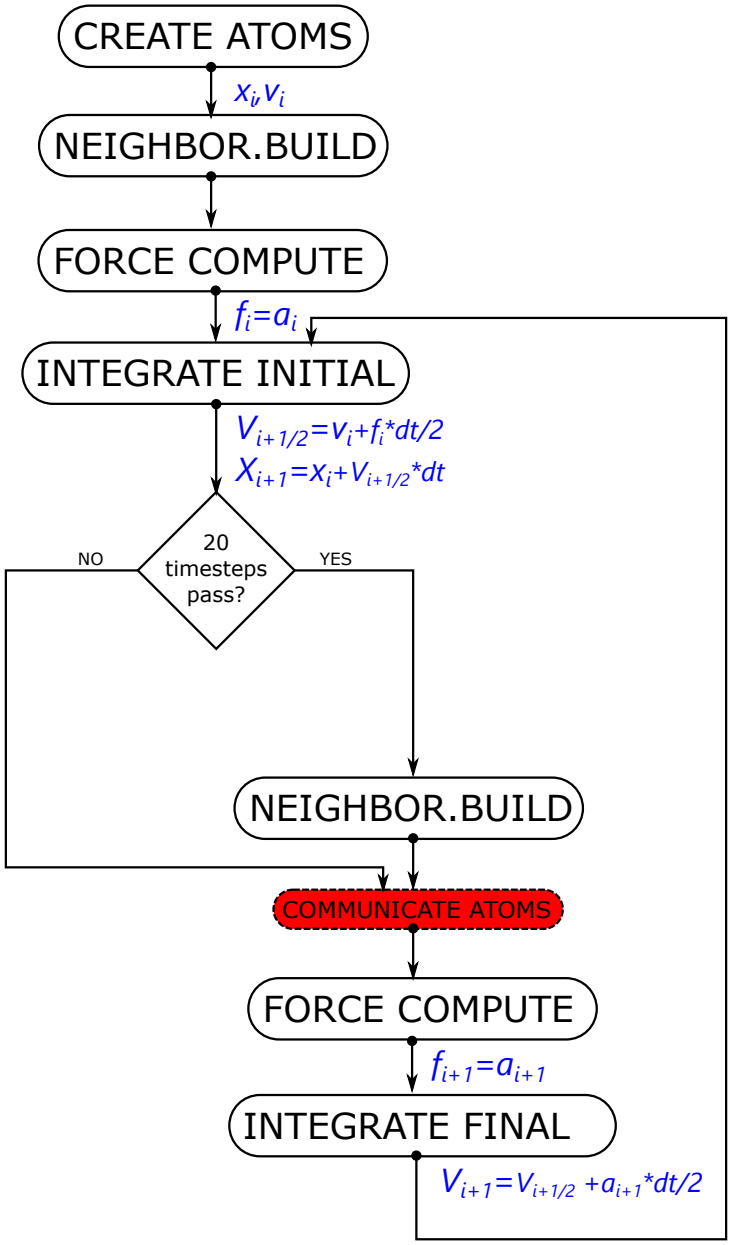


Figure 3.6: Leapfrog Algorithm

As can be seen, the first step of the algorithm deals with the creation of all the atoms that will participate on the simulation, saving for each of them the spatial coordinates and their initial velocities. This creation is performed by allocating the particles into a general *reticle of bin*s. This forms the so called simulation box. After This stage, the neighbour list of atoms for each particle is developed. This means that, if each atom is associated to an integer identifier, all their neighbours

will be indicated as a list of indexes. From a theoretical point of view the miniMD algorithm computes the interactions of each of the N particles in the simulation with all other $N-1$. This leads the computational complexity, for computing the total force acting of a determine dparticle , to become proportional to N^2. But, this can be largely improved considering the actual possible ranges of interactions. In fact for all short range interactions the better approximation wants to neglect all particles that are outside a *cut-off radius*. Now if we call with d the dimension order in which we are performing the simulation (if d=3 we are in the 3Dimensional case) and with r_c the cut-off radious, the computational complexity down to ${nr_c}^d$. Therefore the searching of neighbours is very important and the short-range case is especially good situation that can be used by making sure that interactions are only calculated between those particles having a chance to interact due to their mutual distance. This can be better seen if we imagine to have a computational box (with side L)and divide it into smaller cells with side length l that fill completely the computational box. Each particle is located in exactly one of the smaller cells . If we set the length of l so that it is at least as large as the cut-off , $l \geq r_c$ we can be sure that all the *potential* neighbours of each atom can be found in each small cell and in each of the $3^d - 1$ cells that are adjacent to the primary region. (Each small cell can be called primary cell and all its neighbours cell neighbour cells) (see Fig.3.7).

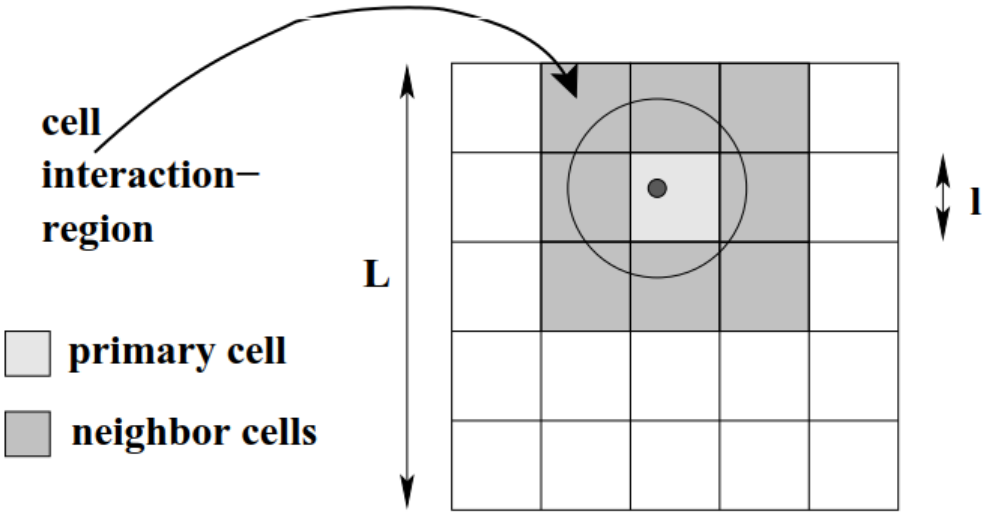


Figure 3.7: The simulation box with side length L and its sub small cells with side length l.

In Fig.3.7 it can be noticed a 2D system with a simulation box of side length L and all its sub small cells of side length l.The light grey coloured cell indicates the

primary region and all the dark-grey cells are the possible regions in which can be possible to find potential neighbours particles. It is important to underline, that the assumption of approximately uniform particle density has been presumed. So in these conditions the computational work for calculating the force on a particle is proportional to the volume that force calculation algorithm searches for possible interactions. Given the aforementioned assumptions, these interactions can be much lower.

At this point all the particles have been indexed and saved in the DDR memory of system. The following step if the total force computation relative to each atom due to the acceleration a_i induced by each one of its neighbours. So the *Integrate initial kernel* will run. Knowing the velocity of the atom at time step i, v_i, it will compute the velocity at time step $i + 1/2$ and so the new position x_{i+1} assumed. This is the moment in which the program controls if 20 time-steps hav been elapsed. If this control returns success, it means that the particles have been moved so much that the list of neighbours for each atom has to be re-populated. Otherwise the algorithm will directly pass to a newer computation of force compute.This because the atoms have experienced a movement with respect to their original positions (after integrate initial) and also would have been completely changed cause of the last execution of neighbour_build kernel. Finally the velocity at time step $i + 1$ will be calculated thanks to Integrate_final and the generated data will be used to execute again the fill Leapfrog algorithm, starting by integrate_initial.
The algorithm will be executed until the required number of time-steps will be elapsed.

Chapter 4

FPGA-based approach for OpenCL kernel acceleration

The OpenCL kernel acceleration methods for the world of FPGAs is different with respect to the one of GPUs because of architectural differences between the two devices. In this chapter it will be explained the OpenCL development from the point of view of FPGA, considering the possible optimizations that can be done by means Vivado HLS, by using both manual standard annotations and Xilinx-specific OpenCL attributes.

4.1 OpenCL development for FPGAs

The first thing to clarify is that the same OpenCL code is executed in different manners in FPGA devices with respect to GPUs. This because of many reasons, mainly dealing with External DRAM memory access bandwidth and execution-modes of each work item within work-groups. Therefore, the goal is to insert into the OpenCL code some directives that will guide the Xilinx OpenCL compiler to create the best mapping between the FPGA architecture and the code itself. Anyway, this effort is largely justified beacuse these optimizations will be able to reach very good performances almost equal to ones of GPUs but saving a lot of energy. In general GPU devices have a very high *bandwidth DRAM interfaces* that overcome those of FPGA. For this reason the first optimization that has to be done for FPGA-based systems, is the one aimed to make more efficient the *movement of data* between external memory and the programmable logic.

4.1.1 Data Movement Optimization

In order to maximize the system-level data throughput it is important initially to isolate, within our OpenCL kernels, the data transfer portions of code from the computational ones because potential inefficiencies in the former one can compromise the good execution of the latter [15]. It is important to remember that in case of Zynq Ultrascale+ architecture the Programmable logic and the Programmable System share the same DDR memory space. This means that the step aimed to transfer data between the host and the device memories by means of OpenCL APIs in not needed. For our special case, the global DDR external memory is a DDR chip soldered onto the Trenz TE0808-09 module outside of the Zynq Ultrascale+ device.

When an OpenCL kernel is synthesized for FPGA, the compiler will create two main conventional memory interfaces: an AXI4-Lite Slave interface in order to manage the functional block control (management of start, end and interrupt signals of each kernel instance) and an AXI4 Master interface (*m_axi_gmem*) for the connection of the functional block to the DDR Memory (see Fig.2.12). While for C/C++ and System C codes the compiler makes available the possibility to choose more different interface type, in case of OpenCL only conventional interfaces are available. By means of the *m_axi_gmem* master interface the kernel instance is able to reach the external DDR memory by accessing the slave interface S_AXI_HP[0-3_FPD of the programmable system (PS). Just to better understand this communication we can take the example 4.1.

```
1 __kernel void dwn_upl(__global int * in, __global int * out) {
2 __local int in_local[128];
3 __local int out_local[128];
4 read(in, in_local);
5 process(in_local, out_local);
6 write(out_local, out);
7 }
```

Listing 4.1: Subdivision of kernel code in the data movement and process parts

As can be noticed the kernel has, as attributes, two global pointers of type integer. It is important to underline the subdivision within the kernel's code between the computational part and the one related to the downloading/uploading of managed data. In fact, a local version of *in* and *out* attributes, are created: *in_local* and *out_local*. After this declaration the read function download all the needed data through the global pointer *in* from the global memory and save them into the local memory version *in_local* of this variable. Then, after that the data are

processed by the *process* function and saved in the local version of *out*, *out_local*, the write function move them in the external DDR memory by means of the *out* global pointer. By performing the separation of the read/write functions from the computational results it has been simplified the following optimization of the computation part of the kernel. Moreover, since the internal variables are mapped to on-chip memories, the retrieving of their values can be performed much more faster then the AXI interfaces.
The global pointers passed as kernel arguments determines the width of the m_axi_gmem interface. In fact its width is equal to the largest type define as global pointer in kernel arguments. This is a very important aspect related to the third further improvement obtained as consequence of the previous computational division: the **burst memory transfers** from external DDR is promoted. Transferring data in bursts, instead of multiple single-memory-transactions, hides the memory access latency because they need to exchange less control-synchronization messages. In general a burst transfer is characterized by the following fields:

- *Burst size*: it describes how many bytes are sent within one transaction
- *Burst Size*: it describes the number of burst-size transactions are performed in the burst.

Therefore,it is useful to use burst in all the cases in which the data to retrieve from the memory are in consecutive address locations and multiple requests of them are performed. Moreover, in order to maximise the data throughput it is raccomended to choose data types that fill the full width of the S_AXI_HP[0-3_FPD interfaces. For example in previous example, the *in* global pointer is an integer pointer of 4 bytes (32 bits). If it had been implemented the *int4* vector-data-type (four integers in only one variable) data type the full bandwidth of 128bit of the interface would be used.
Overall, one of the available methods to infer burst transfer in FPGAs, as also in GPU, is the usage of *async_work_group_copy* function. An example implementing it, is the following Listing 4.2.

```
1 __kernel void dwn_upl(__global int * in, __global int * out) {
2 __local int in_local[128];
3 __local int out_local[128];
4
5 //download phase
6 event_t e_in = async_work_group_copy(in_local, in, 128, 0);
7 wait_group_events(1, &e_in);
8
```

```
9  //process phase
10 process(in_local, out_local);
11
12 //upload phase
13 event_t e_out = async_work_group_copy(out, out_local, 128, 0);
14 wait_group_events(1, &e_out);
15 }
```

Listing 4.2: Optimized code

At line 6, it can be noticed the call to this OpenCL function which has to role to promote the burst downloading of 128 elements of data type int addressed by global pointer *in* and saving them into *in_local*. Then process phase is called. At the end the same function is called to write in burst mode the results from *out_local* to *out*. The *wait_group_events* function is used to wait that the copy operations performed by the previous *async_work_group_copy* is successfully completed.

4.1.2 Kernel process Optimization

Once all the needed data to be processed have been downloaded and saved into local memories of work-groups, we are ready to process them optimizing the computation flow.
The first very big difference between the GPUs and FPGAs, in the guise of OpenCL accelerators, consists on the different manner by means work-items interact to each other during their execution. In fact, in GPUs work-items execute the kernel-code in a pure *parallel* manner, in FPGAs they execute in a *sequential* mode. In other words, it can be said that all the work-items of a work-group will execute the kernel code sequentially as if they are inserted in a loop over their global identifier. The loop we are talking about is the one that have as upper bound the total number of work-items in the work-group. If this upper bound is fixed by means of appropriate directives, Vivado HLS is able to promote many optimizations about the size of internal local memories and about the overall execution time of the kernel. Just to better understand what is happening is useful to see the following example 4.3.

```
1  #define WG_SIZE (64)
2
3  __kernel void dwn_upl(__global int * ina,__global int * inb, __global
       int * out) {
4
5  __local int in_locala[64];
6  __local int in_localb[64];
7  __local int out_local[64];
8
```

```
9  //download phase
10 async_work_group_copy(in_locala, ina, get_group_id(0)*WG_SIZE, 0);
11 event_t e_in = async_work_group_copy(in_localb, inb, get_group_id(0)*
       WG_SIZE, 0);
12 wait_group_events(1, &e_in);
13
14 //process phase
15   LOOP_1: for (int i=0; i < FIXED_UPPERBOUND; i++)
16   {
17       out_local[get_local_id(0)] += in_locala[get_local_id(0)] *
       in_localb[get_local_id(0)]+i*3;
18   }
19
20 //upload phase
21 event_t e_out = async_work_group_copy(out, out_local, get_group_id(0)
       *WG_SIZE, 0);
22 wait_group_events(1, &e_out);
23 }
```

Listing 4.3: example

Let's imagine to have two integer arrays of 128 elements. The first thing to be noticed is that the work-group dimension has been set to 64. This means that each work-group will process only 64 elements of the two 128-width arrays. The task is multiply each element of first array with each element of the second into the LOOP1. This multiplication is then summarized to the value $3 * i$. Then $out_local[get_local_id(0)]$ is updated. If this operation is performed by a GPU, what happens is something like in Fig.4.1.

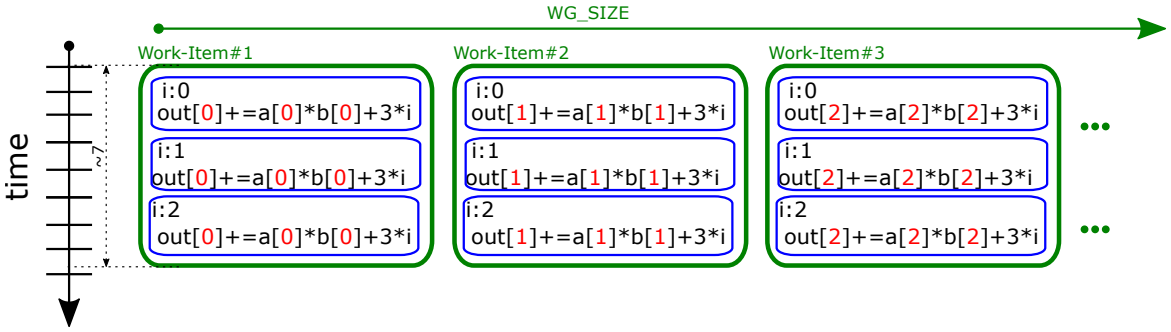


Figure 4.1: Execution of the Listing 4.3 into a GPU

Is important to note that the first action that the work-group will do is to download in local memory 64 elements 128 of each input array. In order to do so, the expression $get_groupid(0) * WGSIZE$ in line 10 is pointing to the first element at which the single work-group can access and from this position it is saying that the amount of data to be transferred consists of WG_SIZE elements of type *int*. In

the meanwhile the $wait_group_events$ waits until the copy operations will be completed. Now, it can start the process phase of the kernel in which, each of the 64 work-items will operate on one of the 64 elements of the two arrays by executing one time the content of the FOR LOOP1. In this operation each work item will retrieve the element to be processed from each local array by means of the get_local_id(0) then will perform the multiplication and the sum. The pure parallel approach of GPUs leads them to execute the code of all the 64 work-items at the same time. Moreover, it can be noticed that the latency about the execution of LOOP1 by each work-item is more or less of seven time-steps.

```
#define WG_SIZE (64)

__kernel __attribute__ ((reqd_work_group_size(WG_SIZE, 1, 1)))

void dwn_upl(__global int * ina,__global int * inb, __global int *
    out) {

__local int in_locala[128];
__local int in_localb[128];
__local int out_local[128];

//download phase
async_work_group_copy(in_locala, ina, get_group_id(0)*WG_SIZE, 0);
event_t e_in = async_work_group_copy(in_localb, inb, get_group_id(0)*
    WG_SIZE, 0);
wait_group_events(1, &e_in);

FOR: 0 -> WG_SIZE
{
  //process phase
  LOOP_1: for (int i=0; i < FIXED_UPPERBOUND; i++)
    {
      out_local[get_local_id(0)] += in_locala[get_local_id(0)] *
    in_localb[get_local_id(0)]+i*3;
    }

}

//upload phase
event_t e_out = async_work_group_copy(out, out_local, 128, 0);
wait_group_events(1, &e_out);
}
```

Listing 4.4: example

As far as FPGA is concerned, as stated before, although the OpenCL compiler

can define the work-group-size, the specification of the *reqd_work_group_size* attribute, at the beginning of the kernel, is able to promote performance optimization during the generation of the custom logic for a kernel [11]. Thanks to this attribute, it can be seen how Vivado HLS pay attention to *loops*. In fact, this attribute is defining the dimension of the work-group in order to clarify to the *xocc* compiler the work-item loop iteration count. By applying this directive, the compiler is transparently transforming the previous kernel code in the Listing 4.4. The same behaviour can be seen from a graphical point of view in Fig.4.2.

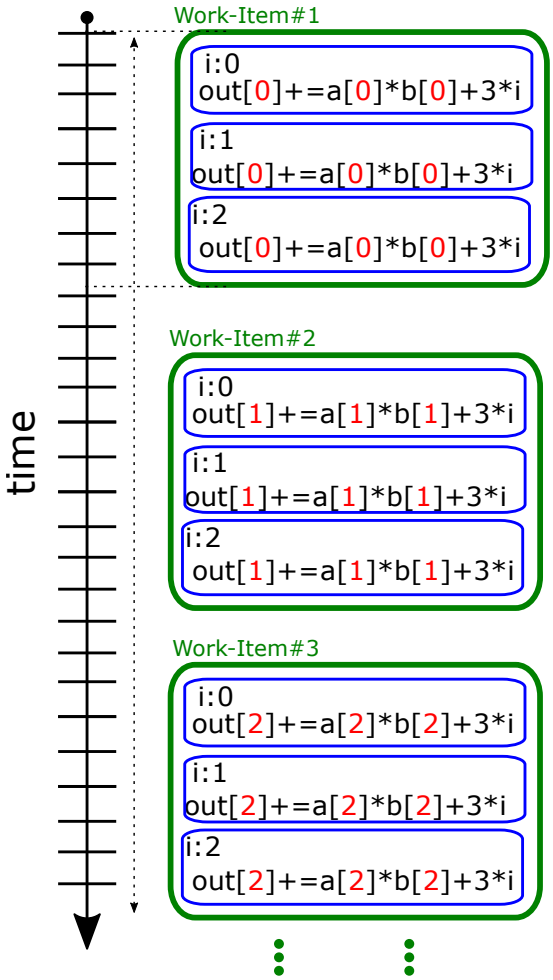


Figure 4.2: Execution of the Listing 4.4 into an FPGA

Nevertheless, the development onto FPGAs by means of High-level-Synthesis allow to use many OpenCL standard and Xilinx-specific OpenCL manual annotations in order to improve the performance. These are essentially based on loop optimization techniques like *pipelining* and *unrolling*.

Before we go into detailed explanation of the examples it is useful to remember some definitions about loops and the parameters that manage their performances. The *Loop iteration latency* is defined as the number of clock cycles the loops takes to complete one iteration of the loop. If this number is multiplied by the number of trip count (th e number of loops that the LOOP has to implement) it can be obtained the total latency of the loop [8]. Furthermore, when pipelining is applied to kernel's loops, Vivado HLS will always try to reduce the *Loop Initiation Interval (II)*. This is the number of clock cycle before the next iteration of the loop starts to process data [8] and the target value that you always want to reach is $II = 1$. In fact in this way, the total latency becomes $TotalLatency \approx IterationLatency + (II * LoopCount)$ instead of $TotalLatency = IterationLatency * LoopCount$.

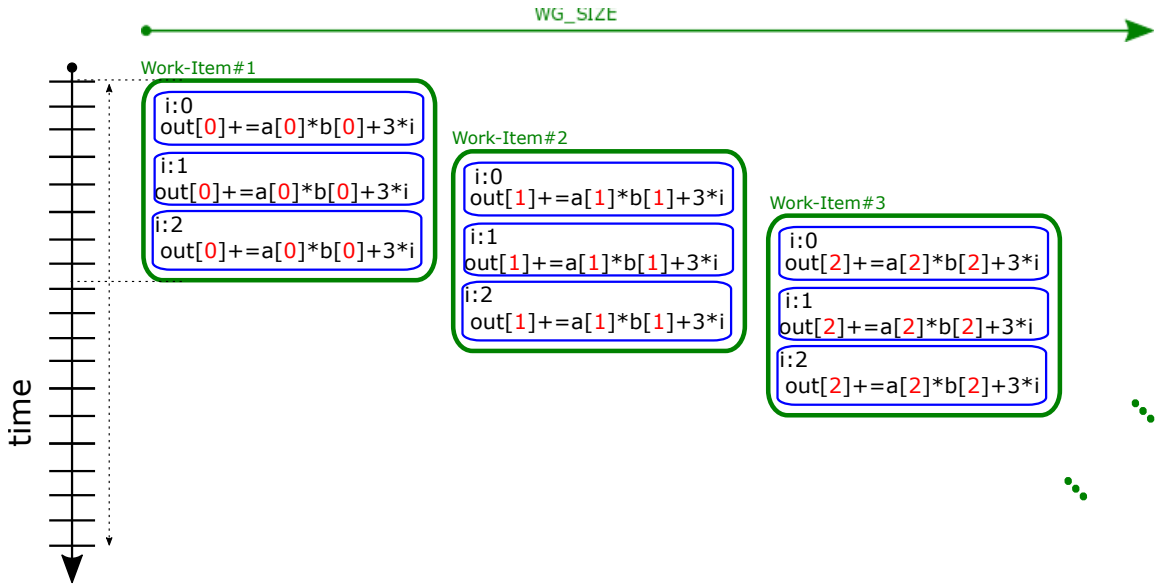


Figure 4.3: Execution of the Listing 4.5 into an FPGA

For what concerns the pipelining directives, there are available two main kind of optimizations.
The first one affects the FOR loop in Listing 4.4 by pipelining the work-item loop iteration count. This means that each work-item will no longer work in a sequential mode but they will be pipelined as in Fig. 4.3. It can be noticed that, by executing this new code, the potential performance improvement has been increased a lot because the effect of pipelining is to reduce the total latency of loops. In fact in this way the FPGA can almost reach the pure parallel approach performances of GPUs but saving a lot of power consumption due to its nature. The Initiation Interval (II) about the execution of one work-item and the following one has been reduce.

The second loop optimization affects, instead, the LOOP1 within each work-item (see Fig.4.4). In this case, each work item will not execute its code in more or less seven time steps like GPUS, but will be able to decrease the initiation interval of LOOP1 targeting the best performances (II = 1).

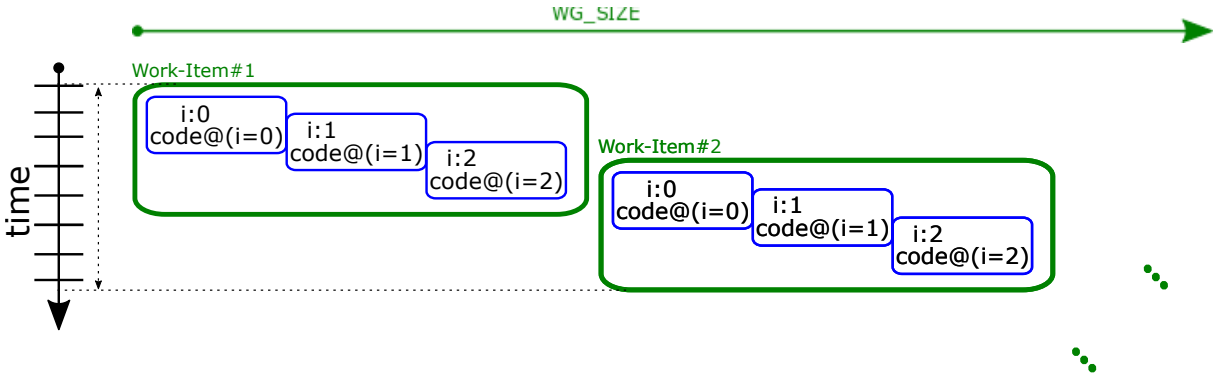


Figure 4.4: Execution of the Listing 4.5 into an FPGA

The final code, after the application of all the aforementioned directives, can be seen in Listing 4.5. Regardless, it is important to note that Vivado HLS will be able to perform pipelining optimizations only if perfect or semi-perfect loops will be available. This means that, the compiler will be able to pipeline a loop in the only case in which this has a *fixed upper bound* .In fact in the following code, the LOOP1 trip count has been fixed to FIXED_UPPERBOUND in order to allow the good work of the annotation. Whenever, the most external loop hosts nested loops that cannot be unrolled or there are some data-dependencies, the pipelining of this one will not success. Actually there is a possibility to apply at the same,the pipelining technique, but the final initiation interval will not be the minimal one but will be greater.
This fixed loop upper bound is very important and the absence of fixed bounded loops drastically decrease the possibility of code's improvement.

```
1 #define WG_SIZE (64)
2 #define FIXED_UPPERBOUND 3
3
4 __kernel __attribute__ ((reqd_work_group_size(WG_SIZE, 1, 1)))
5
6 void dwn_upl(__global int * ina,__global int * inb, __global int *
    out) {
7
8 __local int in_locala[128];
```

```
__local int in_localb[128];
__local int out_local[128];

//download phase
async_work_group_copy(in_locala, ina, get_group_id(0)*WG_SIZE, 0);
event_t e_in = async_work_group_copy(in_localb, inb, get_group_id(0)*
    WG_SIZE, 0);
wait_group_events(1, &e_in);

//process phase

__attribute__((xcl_pipeline_workitems)) {

__attribute__((xcl_pipeline_loop))
LOOP_1: for (int i=0; i < FIXED_UPPERBOUND; i++)
{
out_local[get_local_id(0)] += in_locala[get_local_id(0)] * in_localb[
    get_local_id(0)]+i*3;
}

}


//upload phase
event_t e_out = async_work_group_copy(out, out_local, 128, 0);
wait_group_events(1, &e_out);
}
```

Listing 4.5: example

Another useful technique to optimize performances of kernels is *unrolling* loops. In general the goal of this directive is to perform all the loop iterations of a loop at the same time like the case of GPUs. This will lead to a extreme reduction of latency (the latency of only one iteration loop) but will consume, at the same time, a lot of FPGA resources. Typically pipelining is better then unrolling because returns the best cost/performance trade-off, but sometimes unrolling loops can use more efficiently the resources of the device. Also in this case it can be unrolled both the work-items within a work-group of the loops into loops in each work-item.
The OpenCL attribute that can perform the unrolling of loops is the *opencl_unroll_hint(n)* . The value of n indicates the factor of unrolling and in case it is not present , it means that the loop will be completely unrolled.

```
#define WG_SIZE (64)
#define FIXED_UPPERBOUND 3

```

```
4  __kernel __attribute__ ((reqd_work_group_size(WG_SIZE, 1, 1)))
5
6  void dwn_upl(__global int * ina,__global int * inb, __global int *
       out) {
7
8  __local int in_locala[128];
9  __local int in_localb[128];
10 __local int out_local[128];
11
12 //download phase
13 async_work_group_copy(in_locala, ina, get_group_id(0)*WG_SIZE, 0);
14 event_t e_in = async_work_group_copy(in_localb, inb, get_group_id(0)*
       WG_SIZE, 0);
15 wait_group_events(1, &e_in);
16
17 //process phase
18
19 __attribute__((opencl_unroll_hint))
20 LOOP_1: for (int i=0; i < FIXED_UPPERBOUND; i++)
21 {
22 out_local[get_local_id(0)] += in_locala[get_local_id(0)] * in_localb[
       get_local_id(0)]+i*3;
23 }
24
25 //upload phase
26 event_t e_out = async_work_group_copy(out, out_local, 128, 0);
27 wait_group_events(1, &e_out);
28 }
```

Listing 4.6: example

Just to have an idea about the work performed by unrolling technique, it can be noticed in Listing. 4.6 that the LOOP1 is completely unrolled. This means that, the three loops will be executed at once (see Fig.4.5).

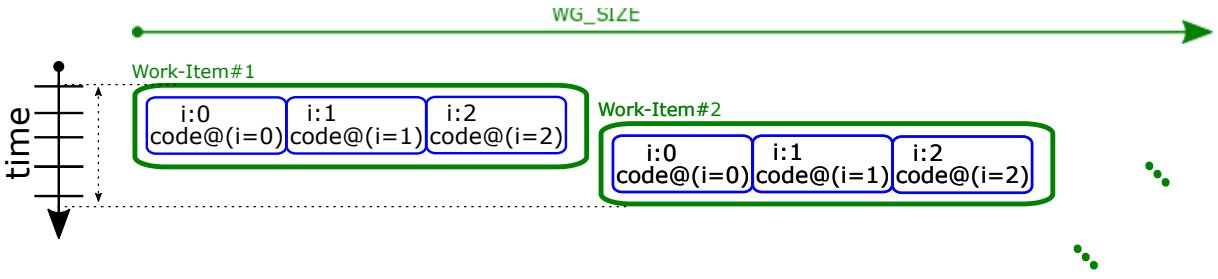


Figure 4.5: Execution of the Listing 4.6 into an FPGA

4.1.3 Intra-FPGA memory optimizations

As stated before, in case of applications dominated by DRAM access FPGAs will loose the challenge with respect to GPUs. This is related to higher bandwidth of GPUs between the device and the global memory. On the other hand, FPGAs have a very high potential in the **internal** memory bandwidth. In fact the *private* and *local memories* in the context of FPGAs are represented by BRAM and LUT inside the programmable logic. Their dimension is largely higher then the internal memories in GPUs and the access to them can be performed with very high speed and bandwidth. As a matter of fact, for this reason, when an application has to be accelerated onto FPGAs is highly recommended to download at first all the needed data in local memories and then access them with a very high throughput. Just to better understand we can see Fig.4.6. In the moment in which the previous code (Listing 4.6) retrieves the internal local version *in_ locala* and *in_ localb* it acces a BRAM with a single port. This means that one work-item at a time will access the local memory because of a single memory port.

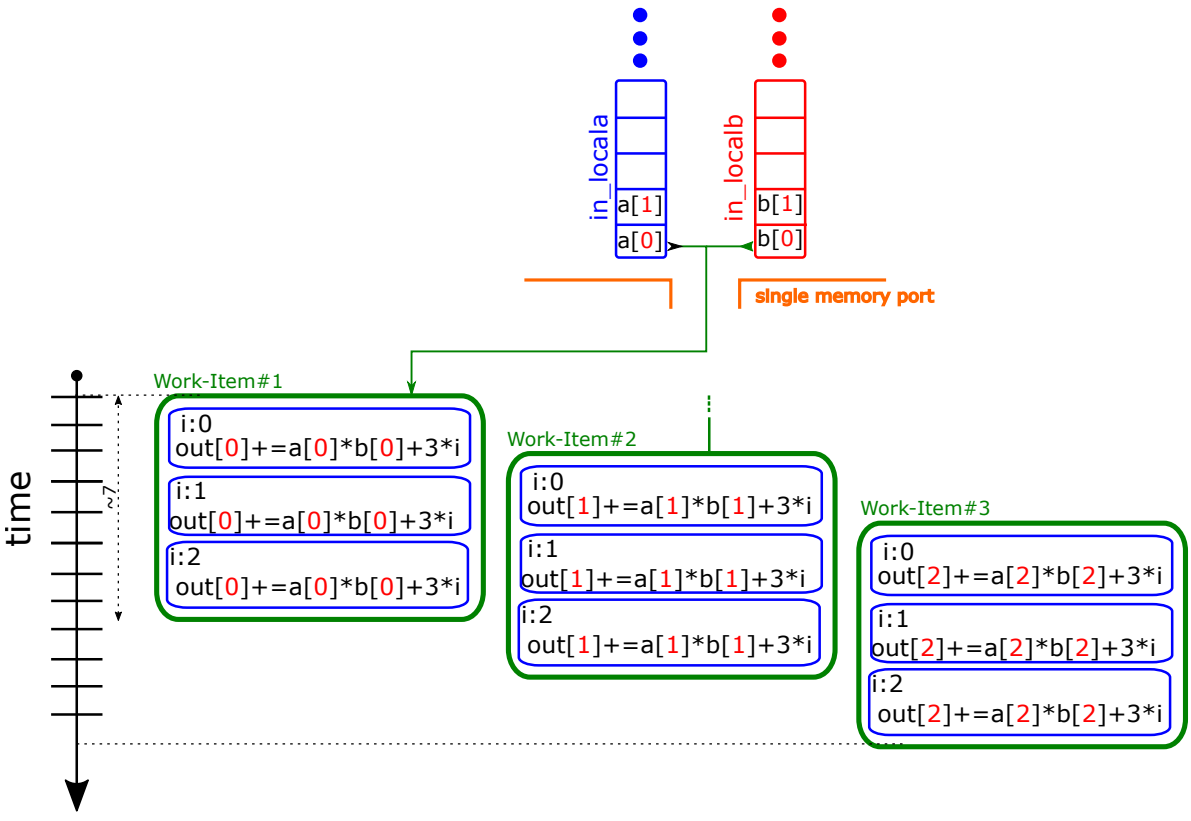


Figure 4.6: Internal memory access

Nonetheless, any from of internal memory like BRAMs can provide two memory access to different locations at the same clock cycle. This is possible by calling a specific *directive*: *xcl_array_partition* . This optimization allows to the code to access more than two times per clock cycle to the same local BRAM thanks to an increase of its number of port. This technique improves data throughput inside the programmable logic because more data can be accessed in each clock cycle. Moreover different types of partitioning exist:

- *cyclic*:"Cyclic partitioning is the implementation of an array as a set of smaller physical memories that can be accessed simultaneously by the logic in the compute unit. The array is partitioned cyclically by putting one element into each memory before coming back to the first memory to repeat the cycle until the array is fully partitioned" [11].

- *block*: "Block partitioning is the physical implementation of an array as a set of smaller memories that can be accessed simultaneously by the logic inside of the compute unit. In this case, each memory block is filled with elements from the array before moving on to the next memory" [11]

- *complete*:"Complete partitioning decomposes the array into individual elements. For a one-dimensional array, this corresponds to resolving a memory into individual registers" [11]

The application of this directive to the previous code can be seen in Listing 4.7. What happens, instead, from the physical point of view can be appreciated in Fig. 4.7.

```
1  #define WG_SIZE (64)
2  #define FIXED_UPPERBOUND 3
3
4  __kernel __attribute__ ((reqd_work_group_size(WG_SIZE, 1, 1)))
5
6  void dwn_upl(__global int * ina,__global int * inb, __global int *
       out) {
7
8  __attribute__((xcl_array_partition(cyclic,2,1)))
9  __local int in_locala[128];
10
11 __attribute__((xcl_array_partition(cyclic,2,1)))
12 __local int in_localb[128];
13
14 __local int out_local[128];
15
16 //download phase
17 async_work_group_copy(in_locala, ina, get_group_id(0)*WG_SIZE, 0);
```

```
18 event_t e_in = async_work_group_copy(in_localb, inb, get_group_id(0)*
       WG_SIZE, 0);
19 wait_group_events(1, &e_in);
20
21 //process phase
22
23 __attribute__((opencl_unroll_hint))
24 LOOP_1: for (int i=0; i < FIXED_UPPERBOUND; i++)
25 {
26 out_local[get_local_id(0)] += in_locala[get_local_id(0)] * in_localb[
       get_local_id(0)]+i*3;
27 }
28
29 //upload phase
30 event_t e_out = async_work_group_copy(out, out_local, 128, 0);
31 wait_group_events(1, &e_out);
32 }
```

Listing 4.7: example

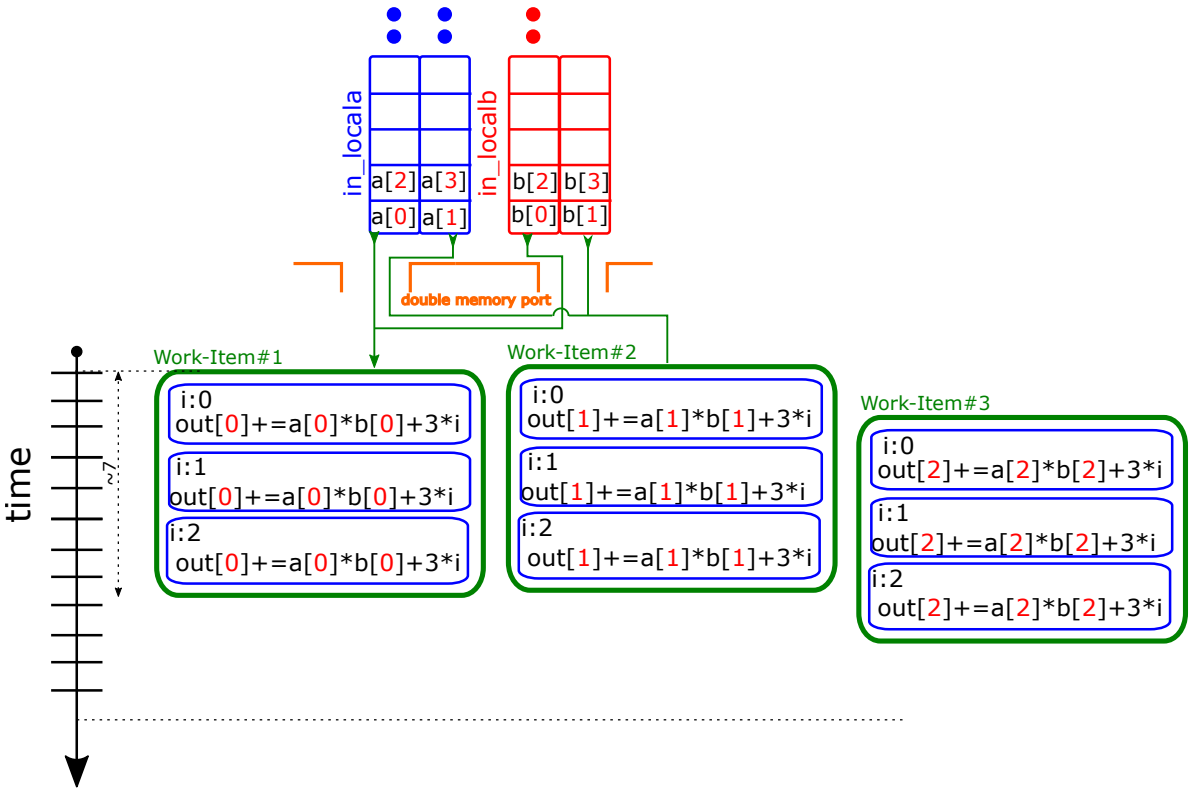


Figure 4.7: Internal memory access optimized

Chapter 5

MiniMD porting on FPGA

this chapter aims to describe the work done for porting the full miniMD application onto FPGA. Particular care has been taken to the analysis of each kernel composing the application and, after a first base-lining, it has been identified which of them could be optimized, in order to speed up the execution time of the whole molecular dynamics code. Then, after the optimization of some of the main five kernels available in the application, a final execution of the code has been performed in order to control the final performances. In the beginning, the focus has been to optimize the performance of each kernel. Later, in order to maximize the usage of the parallelization on the FPGA, some tests were run to understand if using multiple instance of some of the kernel would globally speed up the code.

5.1 Base-lining of the MiniMD Application

As told before, the miniMD application consists of two main parts: the *Host-Code* [1] and *kernels* that will operate within the Leapfrog algorithm.
The first thing that has been performed, in order to better understand the actual relation in terms of execution time among all the kernels, has been the simulation of the full miniMD application onto an aws-FPGA. This is an FPGA platform made available by amazon web services that has made possible a detailed profiling of the code. As stated before all of the following tests and simulation, have been carried out by means of the Xilinx SDAccel software tool in order to manage all the steps about high-level-synthesis, vivado design and writing of the application in an automatic approach.

[1] The Host-Code is indicated in the list of available miniMD files as *lgs.cpp*

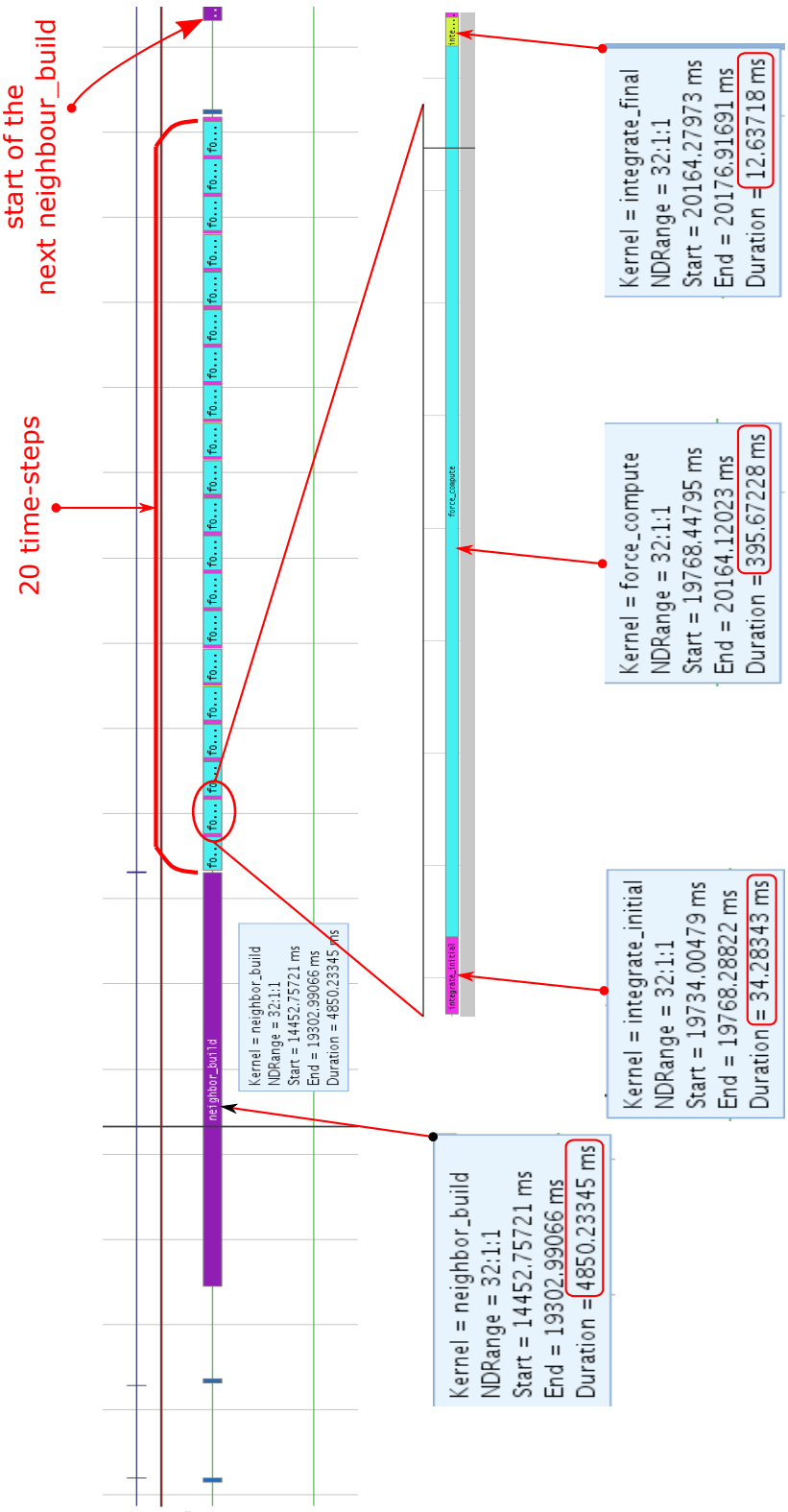


Figure 5.1: Base-lining of MiniMD application

The results about the profiling of the miniMD application can be viewed in Fig.5.1. In accordance with Leapfrog algorithm (see Fig.3.6) the first kernel executed is *Neighbor_build* (first purple block in the figure above), related to the creation of neighbour's list for each atom. Then there is the first execution of the *Integrate_initial* kernel. Then, for 20 time-steps,there will be the loop execution of *integrate_initial* , *force_compute* and *integrate_final*. After that time, there is a need for re-build the neighbours' list for each atom due to a the high quantity of particle's movements. Finally, all of afore-mentioned steps will be executed again till the end of the simulation.
This simulation has been used a total number of atoms equal to 10976 and as can be noticed, the time spent for the execution of *Neighbor_build* is almost $4.850s$, while the one related to *force_compute* is $395.67ms$. These two kernels are the ones that appear to mainly impact on the total execution time of the leapfrog algorithm and so it has been decided to optimize their execution.
Therefore in the following sections we will present a deeper explanation of the logic of *neighbor_build* and *force_compute* and their related optimizations.

5.2 Neighbor_build kernel optimization

If we consider that the execution of *neighbor_build* kernel,it lasts $4.80s$ that is almost the 35% of the total execution of *neighbor_build* plus 20 time-steps of the other kernels. Moreover this kernel is linked to the number of particles used in the application as N^2 and so it would be very useful to increase its performance.

Regardless, before explaining the actual logic of the kernel, we will introduce a very brief clarification of the method through which the general kernel's arguments are prepared and in which way the host code is able to manage to upload/download data from the programmable logic (PL).

5.2.1 Preparation of Kernel Arguments

The host code of the application is the *ljs.cpp* C++ code and has the role to setup all the needed variables required by the OpenCL framework and then all the needed objects after managed by each kernel. As has been told before in Chapter 2, the molecular dynamic simulation has the solve, at the beginning, the simulation-box in that will be filled by all the atoms that will participate to the simulation. As a matter of fact, after the initialisation of some OpenCL variables the code runs the following piece of code (see Listing 5.1):

```
...

create_box(atom, nx, ny, nz, rho);
printf("# ****** CREATE_BOX DONE ****** .... \n");


neighbor.setup(atom);
printf("neighbor.setup(atom) DONE\n");

integrate.setup();
printf("integrate.setup() DONE\n");

force.setup();
printf("force.setup() DONE\n");

create_atoms(atom, nx, ny, nz, rho);
printf("create_atoms(atom, nx, ny, nz, rho) DONE\n");




thermo.setup(rho, integrate, atom, units);
printf("thermo.setup(rho, integrate, atom, units) DONE\n");

create_velocity(t_request, atom, thermo);
printf("create_velocity(t_request, atom, thermo); DONE\n");

atom.d_x->upload();
atom.d_v->upload();
atom.d_vold->upload();
printf("atom.d_x->upload() atom.d_v->upload() atom.d_vold->upload()
    DONE\n");
...
}

//upload phase
event_t e_out = async_work_group_copy(out, out_local, 128, 0);
wait_group_events(1, &e_out);
}
```

Listing 5.1: example

Here, it can be noticed the $create_box$ function aimed to create the simulation-box thanks to the parameters nx,ny,nz set at the beginning of the code. These are very important because are related to total number of atoms ($natoms$) that will be considered in the simulation. In the following simulation $nx = 14$,$ny = 14$,$nz = 14$ with a total of $nx * ny * nz * 4 = 10976$ particles. The number 4 in previous expression stands for the number of particles inside each box of the simulation-box,

while *nx*,*ny*,*nz* are the numbers of boxes along each direction. After that all the boxes have been filled by means of the *create_atom* function and after that all the variables needed by each object related to the kernels have been set the *upload* method of the class atom is called. Since this is the same host-code that can eventually be run for GPU simulations it is curious to analyse what happens when the OpenCL APIs about uploading/writing of data are executed. In this code, the *upload/download* methods will actually call the OpenCL methods *clEnqueueWriteBuffer/clEnqueueReadBuffer* that have the task to enqueue a command to copy the contents of host memory into a buffer pre-allocated region into the device's memory. In the case of the Zynq Ultrascale+ this functions don't work in the same way. The external DDR memory is shared by the x86 CPU and by the programmable Logic. This means that it is like the DDR memory has two address space, one addressable only by CPU and one only by FPGA. As a matter of fact, as will be more clear in the following sections, the clEnqueueWriteBuffer function does nothing but send a copy of the pointer to the data saved in the addressable CPU part of memory, to the FPGA addressable one. At the same time whenever the host-code run the clEnqueueReadBuffer function, what is received by the CPU is the group of pointers to the data in the FPGA addressable part of memory. Just to clarify the functioning see Fig.5.2.

5.2.2 Kernel's optimization

In order to better understand the optimizations performed in the kernel, it is best to explain the logic of its original version (see Listing 5.2).
It can be noticed that all the pointers in the kernel's arguments are preceded by the *__global* identifier. This means that all of these variables are pointers copied by the *clEnqueueWriteBuffer* OpenCL API into the region of external DDR memory addressable by the FPGA. As a matter of fact this is the region recognised by the FPGA device as the OFF-chip global memory (see Fig. 5.2). These are pointers [2] to the standard OpenCL memory-objects created by OpenCL APIs when they prepare the allocation of buffers to transfer onto the device.

[2]These pointers have a weight of 8 bytes that is the dimension of a general OpenCL mem_object

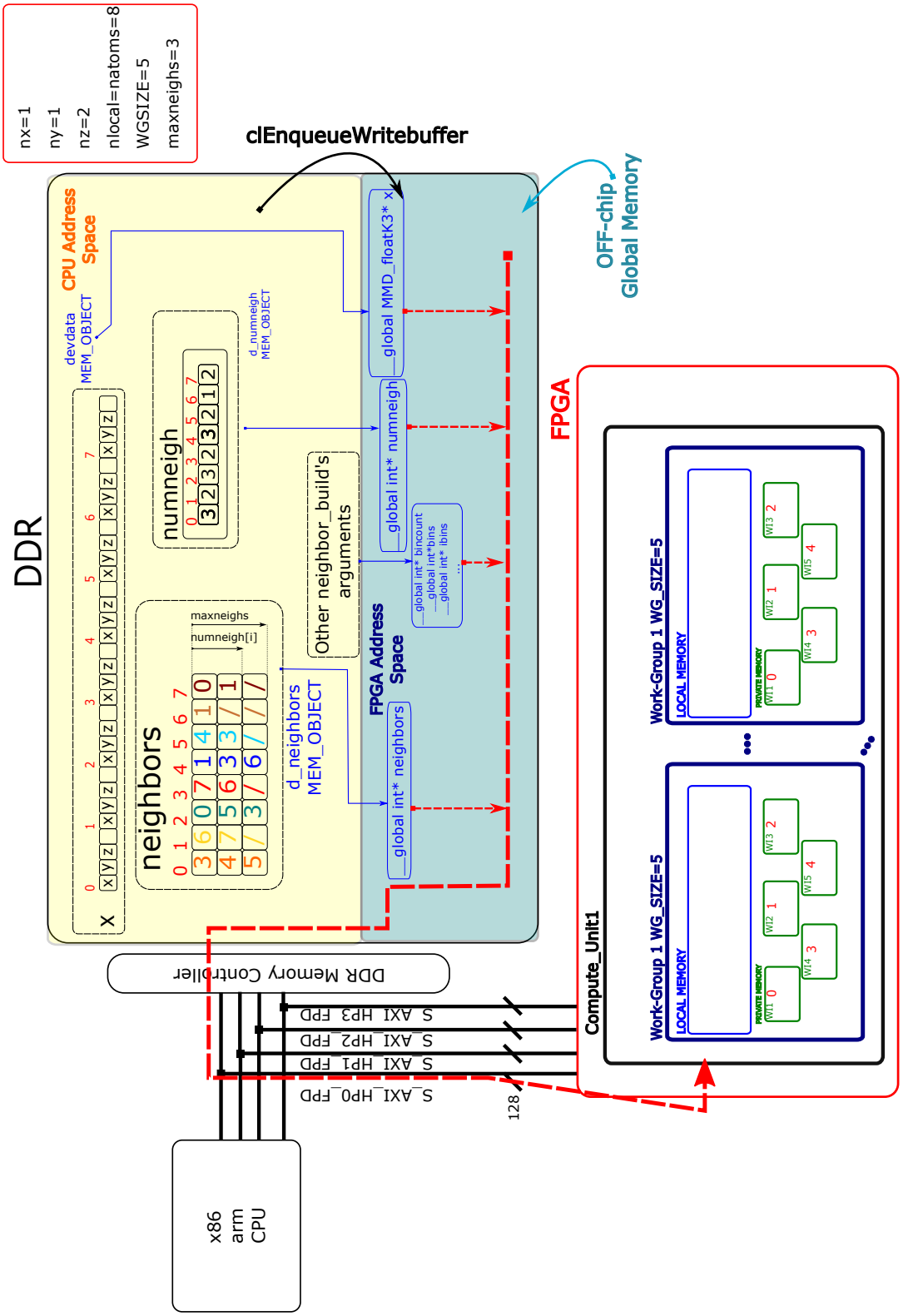


Figure 5.2: System overview and example of a Write operation onto FPGA address memory space

```
__kernel void neighbor_build(__global MMD_floatK3* x, __global int*
    numneigh, __global int* neighbors,
__global int* bincount, __global int* bins, __global int* ibins,
    __global int* flag,
__global int* stencil, int nstencil, MMD_float cutneighsq, int
    atoms_per_bin, int maxneighs, int nlocal)
{
  int i = get_global_id(0);
  if(i>=nlocal) return;
  int ibin = ibins[i];
  MMD_floatK3 xtmp = x[i];
  int n = 0;

  loop1:for(int k = 0; k < nstencil; k++)
  {
    int jbin = ibin + stencil[k];

    loop2:for(int m=0;m<bincount[jbin];m++)
    {
      int j = bins[jbin*atoms_per_bin+m];
      MMD_floatK3 del = xtmp - x[j];
      MMD_float rsq = del.x*del.x + del.y*del.y + del.z*del.z;
      if ((rsq <= cutneighsq)&&(j!=i)) neighbors[i+n++*nlocal] = j;

    }

  }

  numneigh[i] = n;
  if(n>maxneighs)
    flag[0] = 1;
}
```

Listing 5.2: Original version of neighbor_build

Among these arguments we can notice the important variables *__global MMD_floatK3* x*,*__global int* numneigh* and *__global int* neighbors*. The variable *x* is a pointer to an array of *float4* vector data-type containing the spatial coordinates x,y and z of each atom (Despite the float4 can handle 4 different floats values, only 3 of them are occupied for compatibility reasons. The MMD_FFloatK3 data type is a re-definition of float4). Then we have the variable *__global int* numneigh*. this indicates the number of different neighbours particles for each atom. Since each particle is distinguished by the other ones by means of an identifier each neighbour of a given atom will be recognisable by means of its id. The lists of neighbours's ids are saved in the *__global int* neighbors* variable. The actual organization of variables can be clearly viewed in Fig. 5.2. Here, at

th top of the Figure can be observed the array of coordinates x associated to each atom indexed with the red value upon each sub-block. As can be seen the upon example deals with only 8 atoms in order to make simpler the sketch. For each indexed-atom the neighbors data-structure, reports in each column all the neighbours particles of each particle. The total number of neighbours can be different in dependence of the atom and its actual value is reported in the same format in the *numneigh* data-structure. The maximum number of neighbours is *maxneighs*.

Now we are ready to analyse the Listing 5.2. The first thing that each work-items does, is to download its global address (line 5) in order to control if it will process an atom that doesn't actually exist because exceeds the total number of atoms *natoms*. The rest of code is executed only if the atom i exists. A this point the kernel perform the first transfer of data from global memory by downloading the value of *ibin* and *xtmp* processed by each single work-item. It is important to remember that in all these cases the programmable logic will, at first, send the address of data to retrieve and then it will download it. This consumes a considerably amount of time, especially if performed by all the work-items of all the work-groups. The variable *xtmp*, now contains the coordinates of the atom processed by the i-th work-item. Then two for loops will execute with the goal to find the neighbours particles.
In order to better figure out the algorithm see Fig.5.3.

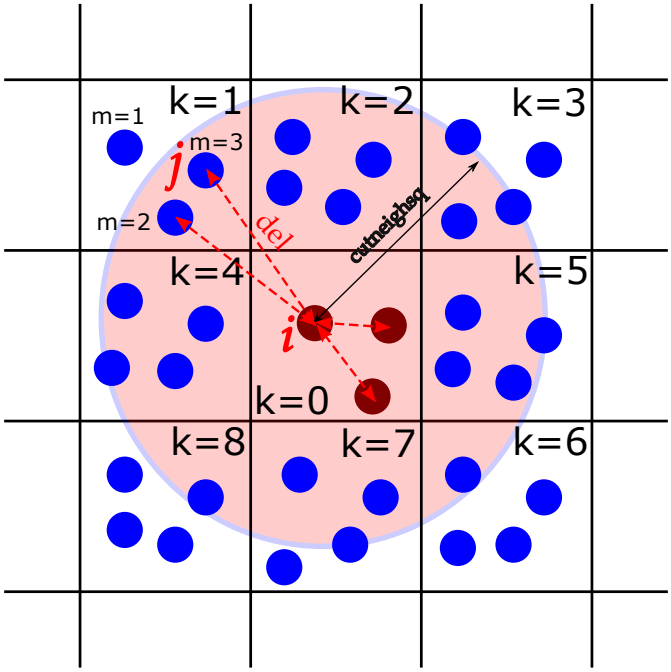


Figure 5.3: Look for neighbour particles

Let's suppose that the atom i at the center of the figure is the one processed by the i-th work-item. What is performed by the loop1 is to look for particle's neighbours in the box where it already is the processed i-th atom and in each of the boxes surrounding it. Once one of the surrounded boxes has been selected, the loop2 starts looking for neighbours by examining the 3D distance *del* between the i-th atom and the j-th. Pay attention that the j index ranges among all atoms of one box. At this point, if the distance *del* doesn't exceed the cut-off ray *cutneighsq* of the i-th atom then the j-th particle will be added to its neighbour's list. At the same time the number of neighbours n is incremented ($n++$). Just to be more clear, if we imagine to process the particle with index 0, the algorithm will add, at first the atom 3 then the 4 and finally 5. At the end the number of total neighbours is 3 and consequently *numneigh* variable will have 3 in the position related to index 0.
After the update of the *numneigh* variable with n, the algorithm will control if total amount of neighbour particles is increased. If this is true a *flag* is set.

As stated in Chapter 3, the methodology to optimize a kernel deals with a first optimization of data transfers and then follows the optimization of computation part. All the optimization can be viewed in Listing 5.5.

```
1  #define WG_SIZE 32
2  #define STENCIL_LENGTH 125
3
4  kernel __attribute__ ((reqd_work_group_size(WG_SIZE, 1, 1)))
5  void neighbor_build(__global float4* x, __global int* numneigh,
       __global int* neighbors,
6  __global int* bincount, __global int* bins, __global int* ibins,
       __global int* flag,
7  __global int* stencil, int nstencil, float cutneighsq, int
       atoms_per_bin, int maxneighs, int nlocal)
8  {
9    int n = 0;
10   __local int ibin_local[WG_SIZE];
11   __local float4 xtmp_local[WG_SIZE];
12   __local int local_stencil[STENCIL_LENGTH];
13   __local int local_numneigh[WG_SIZE];
14   int i = get_global_id(0);
15
16   if(i<nlocal){
17
18     event_t e0 =async_work_group_copy(xtmp_local,x + WG_SIZE*
       get_group_id(0), WG_SIZE, 0);
```

```
19      event_t e1 =async_work_group_copy(ibin_local,ibins + WG_SIZE*
        get_group_id(0), WG_SIZE, 0);
20      event_t e2 = async_work_group_copy(local_stencil,stencil,
        STENCIL_LENGTH, 0);
21      barrier(CLK_LOCAL_MEM_FENCE);
22
23      My_LOOP:for(int k = 0; k < nstencil; k++)
24      {
25        int jbin = ibin_local[get_local_id(0)] + local_stencil[k];
26
27        My_internal_LOOP:for(int m=0;m<bincount[jbin];m++)
28          {
29          int j = bins[jbin*atoms_per_bin+m];
30          float4 del = xtmp_local[get_local_id(0)] - x[j] ;
31          float rsq = del.x*del.x + del.y*del.y + del.z*del.z;
32          if ((rsq <= cutneighsq)&&(j!=i)) neighbors[i+n++*nlocal] = j;
33          }
34
35        }
36
37      local_numneigh[get_local_id(0)] = n;
38
39      if(n>maxneighs)
40        flag[0] = 1;
41
42      barrier(CLK_LOCAL_MEM_FENCE);
43
44      async_work_group_copy(numneigh + WG_SIZE*get_group_id(0),
        local_numneigh, WG_SIZE, 0);
45    }
46  }
```

Listing 5.3: Original version of neighbor_build

For what concerns the data movement, it has been created a local version of all the variables that in the original code had to be downloaded by each work-item of all the work-groups by means of a single transaction memory. This means that all the lines in the original code, in which there was an access to the off-chip global memory by means one pointer available among kernels arguments, are replaced with a local arrays saved in the work-group's local memories (blue regions in Fig.5.2). For example,in the original code (Listing 5.2)at line 8,the single work-item retrieves the coordinates of the processed particle by downloading a float4 variable from global memory. The creation of a local array *xtmp_local* in Listing 5.5 at line 8, allows to download in *burst* all the particle's coordinates belonging to a whole work-group. The download is performed by using the single 128 bit-width axi interface S_AXI_HP0_FPD and the corresponding line in the code is the 15-th. The *async_work_group_copy* function allows the download of WG_SIZE

elements of type float4 from the address *x+ WG_SIZE*get_group_id(0)*.
The same optimization is also carried out for *ibin*, *stencil* and *numneigh* pointers. It is important to note as vivado_hls sees all these *async_work_group_copy* calls as many loops. As a matter of fact, it is referring to the loop over all the work-group's work-item that is optimized by the compiler by means of the *pipelining* technique. In fact from the vivado_hls *log file* it can be noticed the following lines:

```
1 INFO: [SCHED 204-61] Pipelining loop 'Loop 1'.
2 INFO: [SCHED 204-61] Pipelining result : Target II = 1, Final II = 1,
      Depth = 3.
3 INFO: [SCHED 204-61] Pipelining loop 'Loop 2'.
4 INFO: [SCHED 204-61] Pipelining result : Target II = 1, Final II = 1,
      Depth = 136.
5 INFO: [SCHED 204-61] Pipelining loop 'Loop 3'.
6 INFO: [SCHED 204-61] Pipelining result : Target II = 1, Final II = 1,
      Depth = 137.
```

Listing 5.4: Vivado_HLS log about pipelining of *async_work_group_copy*

where 'Loop-1', 'Loop-2' and 'Loop-3' are referring to the first three calls of *async_work_group_copy* functions at lines 18,19 and 20.
Then at line 21, the *barrier(CLK_LOCAL_MEM_FENCE)* function is run in order to wait that all the work-items finish the download of all needed variables. Also this line is viewed by Vivado_hls as a loop and in most of the cases, into the log file, it is signed as *XCL_WG_DIM_X*.
All the needed variables to execute the process part are now available. In this part of the code there are two for loops with a not-fixed upper bound. The first one is *My_LOOP*. Different executions of the code showed that its upper bound *nstencil*, has a minimum of 27 and a maxim of 125. For this reason, it has been allowed the download of 125 elements from the stencil *pointer* in the array *local_stencil* but the actual value used in the code depends on the single execution and ranges between 27 and 125.
The second loop is the most internal one: *My_internal_LOOP*. Also in this case *bincount[jbin]* is not a fixed value because depends on the variability of the *local_stencil* element's values. These are the cases in which vivado_hls is not able to pipeline loops, or in case it is able to do it the Initiation Interval is not the minimum one but is higher.

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	9	9	3	1	1	8	yes
- Loop 2	136	136	136	1	1	2	yes
- Loop 3	260	260	137	1	1	125	yes
- XCL_WG_DIM_X	3	?	3 ~ ?	-	-	1 ~ ?	no
+ My_LOOP	140	?	141 ~ ?	-	-	1 ~ ?	no
++ My_internal_LOOP	447	?	448	137	1	1 ~ ?	yes
- Loop 5	165	165	135	1	1	32	yes

Figure 5.4: Pipelining technique applied to all Loops of neighbor_build kernel

As a gesture of such, in Fig.5.4 can be noticed that while *My_LOOP* has not been pipelined, the *My_Internal_LOOP* yes but with a final Initiation Interval of 137. This happens because of a *carried dependence constraint* between line 29 and 30. Finally, the value of n is saved into the local version of *numneigh*. Then with a burst access (seep Loop5 in Fig. 5.4) *local_numneigh* array is uploaded into the off-chip global memory.

Overall the optimizations done on this code have been deeply limited by the no-bounded loops but in general an increase of performance was detected. Another simulation on 10976 atoms has been performed and the execution time of neighbor_build became of $4.3s$ against the previous $4.856s$. This improvement is mainly due to the burst access memory optimizations. The frequency at which the kernel has been synthesised is *250 MHz* and the resource utilization about the original code and the optimized one can be viewed in Fig.5.5.

original_code (exec_time)	**optimized_code (exec_time)**	**improvement**	**natoms**	**kernel_frequency**
4.856 s	**4.3**	**11%**	10976	250 MHz

Table 5.1: Execution times of the original and optimised version of neighbor_build kernel

The main differences concern the increased usage of internal BLOCK_RAM and Flip-Flops. This is probably due to the enhanced presence of internal arrays created in local memories of work-groups. The increase of FFs is, instead probably due to the pipelining technique applied in the process part and during the application of burst upload/download commands.

original version

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	3835	-
FIFO	-	-	-	-	-
Instance	8	13	2553	3109	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	2594	-
Register	0	-	4119	32	-
Total	8	13	6672	9570	0
Available	4320	6840	2364480	1182240	960
Utilization (%)	~0	~0	~0	~0	0

optimised version

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	19555	-
FIFO	-	-	-	-	-
Instance	30	13	3355	3907	-
Memory	17	-	0	0	-
Multiplexer	-	-	-	2771	-
Register	0	-	8683	544	-
Total	47	13	12038	26777	0
Available	4320	6840	2364480	1182240	960
Utilization (%)	1	~0	~0	2	0

Figure 5.5: Resource utilization of both original and optimised version of neighbor_build kernel

5.3 Force_compute kernel optimization

The force compute kernel has the role to calculate the forces acting on each particle of the stimulation. It is a step executed into every step of the simulation and it is compute intensive from both the computational and memory access points of view. For this reasons it is easy to understand that it needs to be optimized. A first explanation of kernel will be presented along with its original version code.

```
1  #define WG_SIZE 32
2
3  __kernel void force_compute(__global MMD_floatK3* x, __global
       MMD_floatK3* f, __global int* numneigh,
4  __global int* neighbors, int maxneighs, int nlocal, MMD_float
       cutforcesq)
5  {
6  int i = get_global_id(0);
7  if(i<nlocal)
8    {
9    __global int* neighs = neighbors + i;
10   MMD_floatK3 ftmp;
11   MMD_floatK3 xi = x[i];
12   MMD_floatK3 fi = {0.0f,0.0f,0.0f};
13
14   for (int k = 0; k < numneigh[i]; k++) {
15     int j = neighs[k*nlocal];
16     MMD_floatK3 delx = xi - x[j];
17
18     MMD_float rsq = delx.x*delx.x + delx.y*delx.y + delx.z*delx.z;
19     if (rsq < cutforcesq) {
```

```
20          MMD_float sr2 = 1.0f/rsq;
21          MMD_float sr6 = sr2*sr2*sr2;
22          MMD_float force = 48.0f*sr6*(sr6-0.5f)*sr2;
23          fi += force * delx;
24          }
25        }
26    f[i] = fi;
27
28    }
29  }
```

Listing 5.5: Original version of force_compute

The first thing that can be noticed is that also in this case the single work-item will execute the kernel only if its global address is lower than the total number of atoms,with the purpose to avoid ghost particles. After this first control each work-item will download the spatial coordinates relatives to the i-th processed atom (line 11) and will also save (line 9) in the *neighs* variable, the pointer to the array with the indexes of all of its neighbours. The situation is similar to the one presented in Fig.5.2. Subsequently, the actual force computation algorithm starts. In each cycle of the loop,the variable j hosts the index of one of the neighbour particle and after the computation of the distance between them, the algorithm will continue with the computation of the force acting on the i-th particle, only if the previous distance is lower than a threshold *cutforcesq*. The same computation is performed on all the neighbours particle of the i-th atom that is equal to *numneigh[i]*. Of course, at the end, the total force contribution acting on the examined atom will be obtained (line 26).
A simulation of this original version of the kernel leads to an execution time of $395.6ms$ associated to an single istance of the kernel and an IP synthesized at $250MHz$.

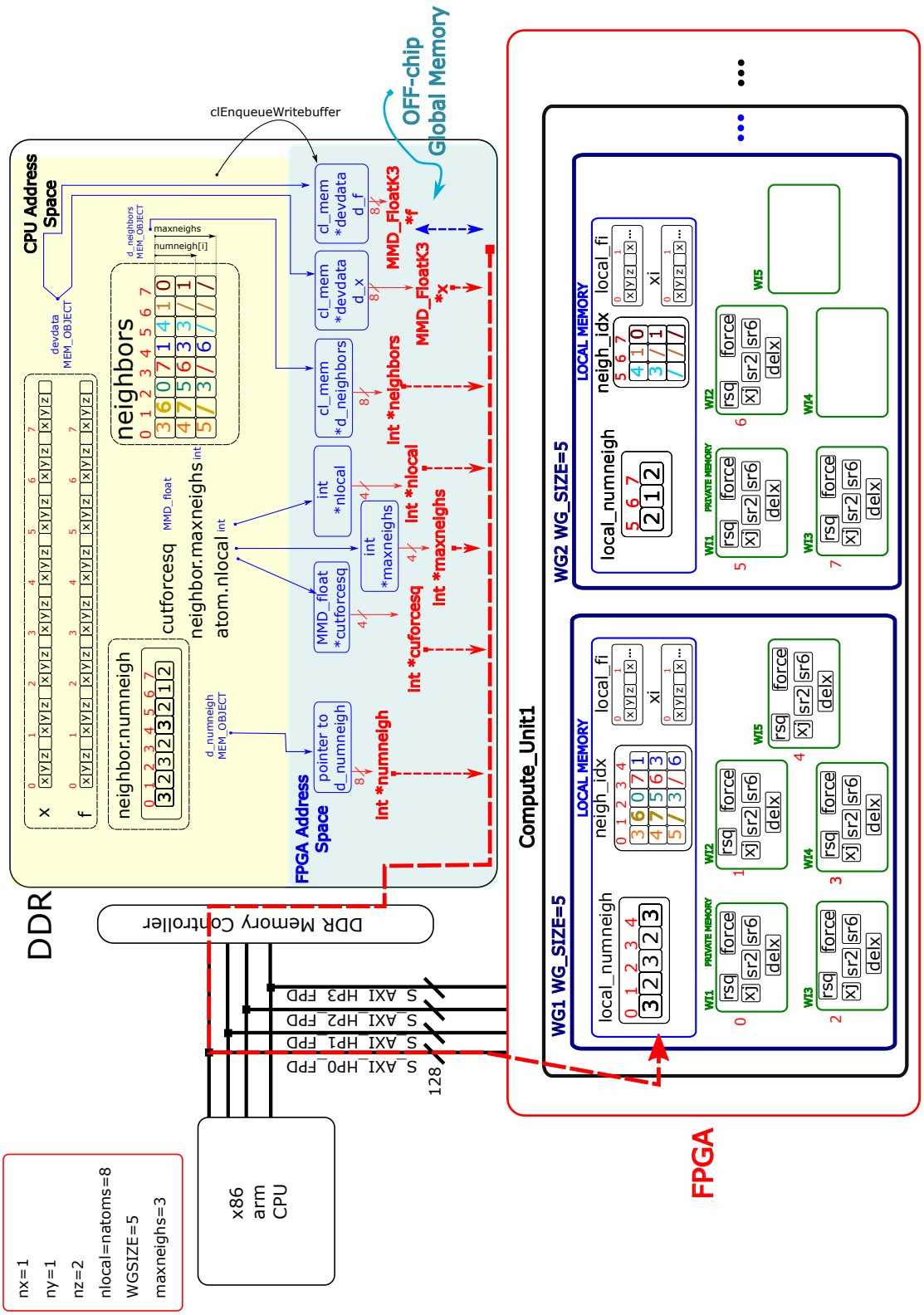


Figure 5.6: force_compute kernel variables' provision

5.3.1 Kernel's optimization

Also for the optimization of the force_compute kernel have taken place two stages. The first one is related to the download by means of burst memory transaction of all the needed variables. Listing 5.8 shows these first optimizations. First of all, it is clarified that the work-group is based on 32 work-items in order to promote the future optimizations in the process part. Then a local version of the spatial coordinates of each atom, processed by each work-item and the respective initial values of forces and their neighbours are created (see line 6,8,11 and 13).

```
1  #define WG_SIZE 32
2  kernel __attribute__ ((reqd_work_group_size(WG_SIZE, 1, 1)))
3  void force_compute( __global float4* x, __global float4* f, __global
       int* numneigh, __global int* neighbors, int maxneighs, int nlocal,
        float cutforcesq )
4  {
5
6    __local float4 xi[WG_SIZE];
7
8    __local float4 fi_local[WG_SIZE];
9    fi_local[get_local_id(0)]=(0.0f,0.0f,0.0f,0.0f);
10
11   __local int local_numneigh[WG_SIZE];
12
13   __local int neigh_idx[WG_SIZE];
14
15   if(get_global_id(0) < nlocal)
16   {
17
18     event_t e3 = async_work_group_copy(xi,x + WG_SIZE*get_group_id(0)
       , WG_SIZE, 0);
19     event_t e0 = async_work_group_copy(local_numneigh,numneigh +
      WG_SIZE*get_group_id(0), WG_SIZE, 0);
20     wait_group_events(1, &e0);
21
22     MY_LOOP_FOR: for (int k = 0; k < maxneighs; k++)
23     {
24
25       async_work_group_copy(neigh_idx,neighbors + WG_SIZE*
      get_group_id(0) + k*nlocal,WG_SIZE, 0 );
26
27       if(k < local_numneigh[get_local_id(0)])
28       {
29
30         float4 delx;
31         float4 xj = x[neigh_idx[get_local_id(0)]];
```

```
32          delx=xi[get_local_id(0)]-xj;
33          float rsq = delx.x*delx.x+ delx.y*delx.y + delx.z*delx.z;
34          if (rsq < cutforcesq)
35          {
36            float sr2 = 1.0f/rsq;
37            float sr6 = sr2*sr2*sr2;
38            float force = 48.0f*sr6*(sr6-0.5f)*sr2;
39            fi_local[get_local_id(0)] += force * delx;
40          }
41        }
42
43      }
44        async_work_group_copy(f + WG_SIZE*get_group_id(0),fi_local,
      WG_SIZE, 0);
45    }
46
47  }
```

Listing 5.6: First optimization of force_compute kernel

Just to better understand the organization of variables in the memory hierarchy it can be possible to analyse Fig. 5.6. Here it can is represented the case of $ntoms = 8$ particles processed by work-groups of size 5 (WG_SIZE=5). In the CPU address space of RAM (yellow region),it can be noticed the presence of arrays of spatial coordinates x and forces f associated to each atom indicated with the red numbers. Moreover, there are the array of neighbours' number for each atom saved in *numneigh* and the the indexes of such neighbours saved in *neighbors* . Apart from these, there are also the other variables that will be requested in the kernel arguments : $cutforcesq$, $maxneighs$, $nlocal$ ($nlocal = natoms$). It is important to remember that every time the OpenCl API clEnqueueWriteBuffer is called the memory objects or the pointer of all the aforementioned variables are transferred in the FPGA address space (blu region). Also in this case the $MMD\_floatK3$ data type is a re-definition of a *float4* data type.
After this first delacation of variables, the algorithm executes the usual control about the out-of-bound particle and then starts to infer burst download of the coordinates x and the number of neighbours for each of the atoms in the work-group (line 18 and 19). At this point, starts MY_LOOP_FOR that is in charge of compute the actual total forces acting of each atom. At line 25 is inferred the burst download of all the neighbour's indexes of all the work-items belonging to the same work-group. A better graphical explanation can be found in Fig.5.7.

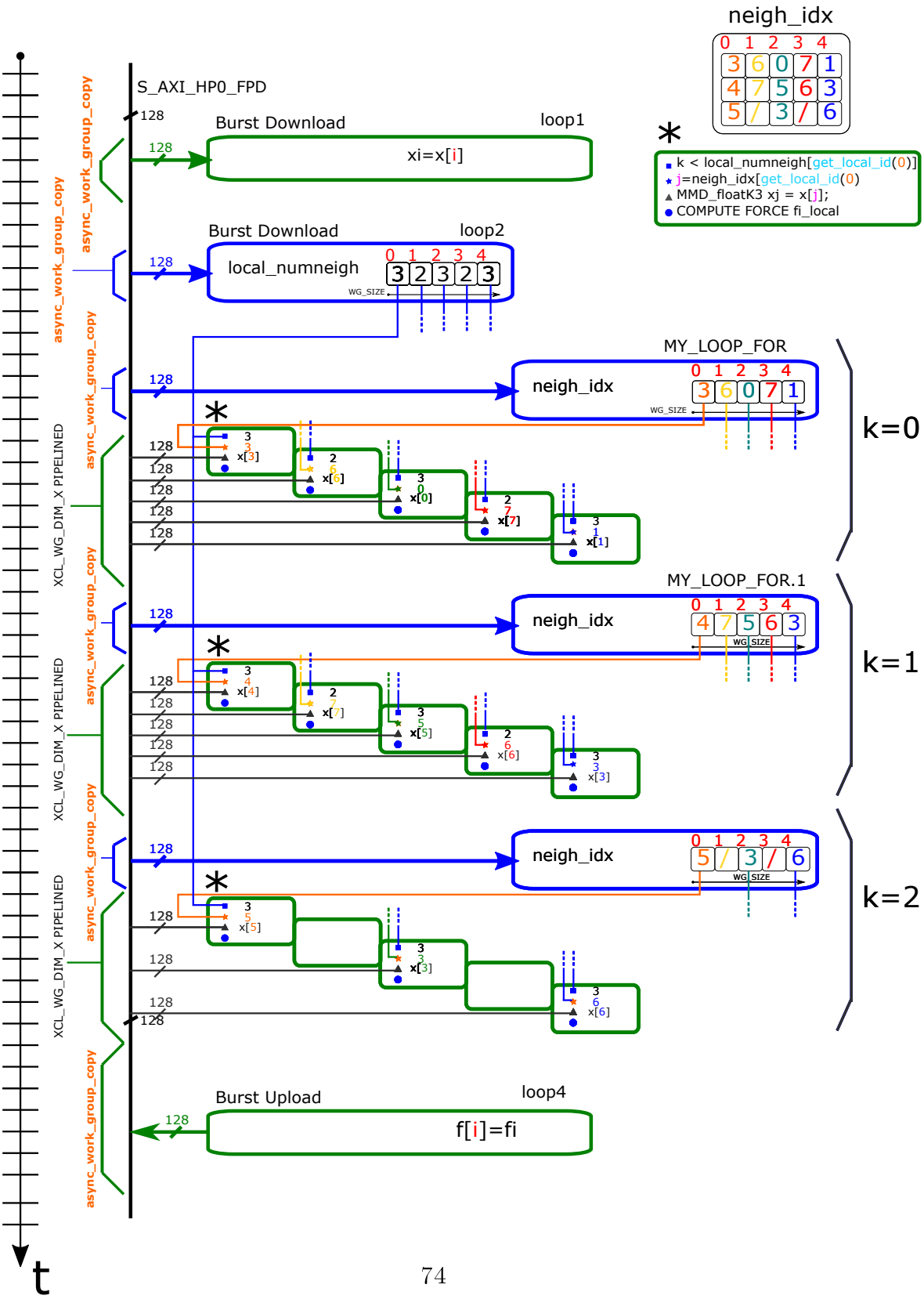


Figure 5.7: Graphical description of the optimised force_compute kernel

There is represented the algorithm executed from the point of view of a single work-item and of the work-group at which it belongs. The first two bursts download are viewed by Vivado_hls's compiler as loop1 and loop2, and for this reason are pipelined. Then If we imagine that we are in the first work-group, the particles processed will be the number 0,1,2,3 and 4. The maximum number of neighbours is *maxneighs* equal to 3 in this example, so the trip count of MY_LOOP_FOR will be 3 (cycle number 0 with k=0 and so on). At k=0,the first neighbours are downloaded in burst by means of an *async_work_group_copy* function so, the first rows of the *neigh_idx* matrix is downloaded. Subsequently, each atom will control if it has finished to control the force contribution from its neighbours atoms (line 27). If not, the partial force computation will be performed only if the distance between the *i-th* atom and its neighbour *j* is less then *cutforcesq*. all these calculations (from line 27 up to 39) are seen from Vivado_hls as something that can be pipelined from the point of view of work-group. In other words, by referring to Fig.5.7 the green boxes represent al the 5 work-items that are executing the previous lines of code in a pipelined fashion. This is indicated in the vivado_hls's report as $XCL\_WG\_DIM\_X$-loop and it is pipelined.
After the execution of all the trips of the most external loop, the situation of handled variables in the FPGA is the one represented in the FPGA region of Fig.5.6. Here, can be appreciated that the local arrays save in local memories of the two work-groups instantiated. Of course, it is interesting to note as, since the second work-group has to process only 3 out of 5 potential atoms, two work items will not do anything due to the control at line 27.

Finally, when the total force acting on each atom has been saved into *fi_local* a burst upload in inferred at line 44, as can be noticed with the green box at end of Fig. 5.7.

A further optimization was possible for what concern the download of spatial coordinates of all the neighbours of each atom, at once at the beginning of code. At the state of art, at every cycle each work-item downloads this coordinates at line 31. This optimization has not been possible because, in order to infer the burst of all these data the exact dimension of the *neigh_idx* matrix was should have been fixed. This means that the total number of rows *maxneighs* and the total number of columns *natoms* of the previous matrix, should have been fixed. This has not been possible because the previous data are variables and so, degrees of freedom of a molecular dynamic simulation.
Also in this case, the single instance of the kernel communicates with only one of the 128 bit axi interfaces with the DDR.

After the synthesis of the previous Listing, Vivado_hls reports the following info

about the optimised loops. It is important to note that MY_LOOP_FOR cannot be pipelined because of its not-fixed upper bound *maxneighs* .

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	9	9	3	1	1	8	yes
- Loop 2	136	136	136	1	1	2	yes
- MY_LOOP_FOR	?	?	?	-	-	?	no
+ MY_LOOP_FOR.1	167	167	137	1	1	32	yes
+ XCL_WG_DIM_X	?	?	255	1	1	?	yes
- Loop 4	17	17	4	2	1	8	yes

Figure 5.8: Optimised loops in force_compute kernel

What can be noticed is that loop4 is the only one that is pipelined with and an achieved II of 2. As a matter of fact, by further analysing the loop by means of the analysis perspective of Vivado_HLS what emerges is in Fig.5.9.

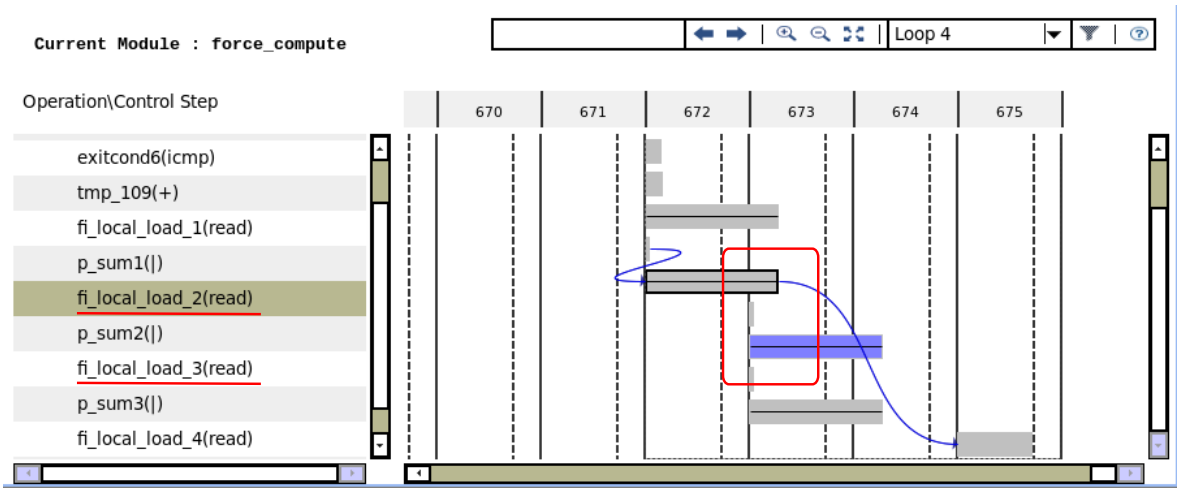


Figure 5.9: Analysis perspective of Vivado_HLS about loop4

In general, read operations take 2 clock cycles: a cycle to generate the address for the block RAM and a cycle to read the data. In the figure above, it can be noticed that the algorithm wants to access to new other values of fi_local (fi_local_load_3 and fi_local_load_4) before that the second cycle about fi_local_load_1 and fi_local_load_2 is finished. This fact is further verified by the log file that notify (see Listing 5.7) that the number of memory ports to the local BRAM has to be increased in order to promote better optimizations.

```
INFO: [SCHED 204-61] Pipelining loop 'Loop 4'.
WARNING: [SCHED 204-69] Unable to schedule 'load' operation ('
    fi_local_load_3') on array 'fi_local' due to limited memory ports.
     Please consider using a memory core with more ports or
    partitioning the array 'fi_local'.
```

Listing 5.7: Portion of log file about force_compute kernel synthesis

Therefore the complete array partitioning of the fi_local array has been performed by adding the following directive:

```
#define WG_SIZE 32
kernel __attribute__ ((reqd_work_group_size(WG_SIZE, 1, 1)))
void force_compute( __global float4* x, __global float4* f, __global
    int* numneigh, __global int* neighbors, int maxneighs, int nlocal,
     float cutforcesq )
{
...

__attribute__((xcl_array_partition(complete,1)))
__local float4 fi_local[WG_SIZE];

...
}
```

Listing 5.8: First optimization of force_compute kernel

After all of these optimizations the kernel has reached a significant increase of performance by pipelining with an II =1 also the loop4. The execution time was reduced from 395.67ms down to 111ms.

original_code (exec_time)	**optimized_code (exec_time)**	**improvement**	**natoms**	**kernel_frequency**
395.67 ms	**111 ms**	**28%**	10976	250 MHz

Table 5.2: Execution times of the original and optimised version of force_compute

As for Resource utilization, in Fig. 5.10 we report a comparison between the original version of the code and the optimised one (@300 MHz).

original version

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	1617	-
FIFO	-	-	-	-	-
Instance	8	5	2681	3183	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	2317	-
Register	0	-	3432	32	-
Total	8	5	6113	7149	0
Available	4320	6840	2364480	1182240	960
Utilization (%)	~0	~0	~0	~0	0

optimised version

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	7758	-
FIFO	-	-	-	-	-
Instance	30	58	9998	7741	-
Memory	15	-	256	16	-
Multiplexer	-	-	-	2362	-
Register	0	-	45431	4082	-
Total	45	58	55685	21959	0
Available	4320	6840	2364480	1182240	960
Utilization (%)	1	~0	2	1	0

Figure 5.10: Comparison about Resource Utilization between original and optimised version of force_compute kernel

It can be noticed an increase of BRAM utilization, but also an increase of DSP48Es and FFs. Also in this case this is probably due to the increased management of local variables handled in work-group's local memories. Instead, the portions of code relative to the computation of total forces are so consistent that allow an considerable rise of logic units like DSPs promoted by a deep utilization of the pipeline technique.

5.4 Further optimizations

Up to this point, every effort has been made to optimise the execution time of the most-compute intensive kernels:*neighbor_build* and *force_compute*. This has consequently reduced the time to complete the full miniMD algorithm.
But there are other solutions to improve the overall performance. The SDx software tool allows to create onto the FPGA a customized area called OpenCL region (OCL region) that allows to implement the OpenCL parallel model [18]. This meeans that multiple compute units (instances) of the same kernel can be created, in order to actually parallelize the work. This technique allows to use much more efficiently the bandwidth between external memory and programmable logic allowing parallelism on a coarse-grained level [10].
As to this particular application, it could be useful to promote the parallel execution of *neighbor_build* and *force_compute*. As a matter of fact, it has been implemented a version of the application with two compute units instantiated for each of these kernels (see Fig.5.11).

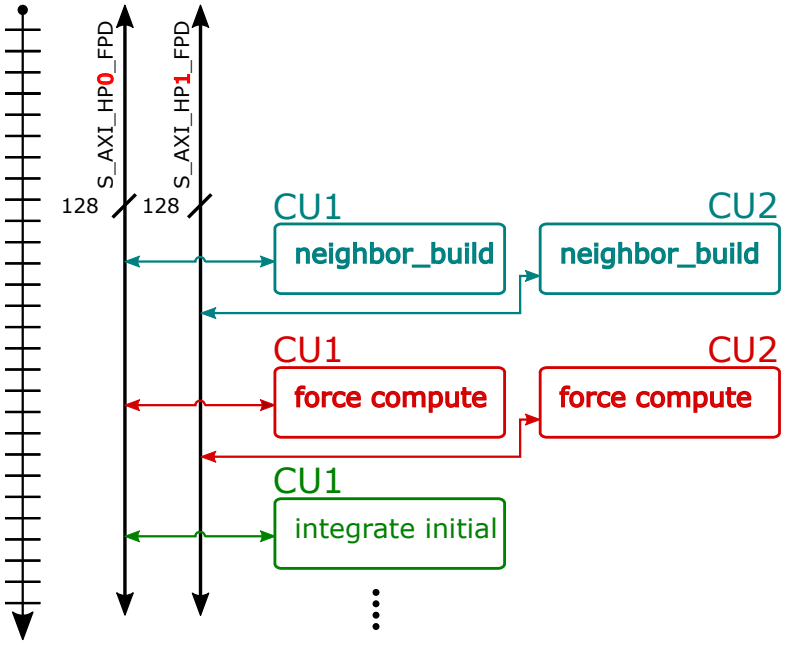


Figure 5.11: Design with two compute-units for neighbor_build and force_compute kernels

As it can be noticed from the figure above, both the instances of the kernels are linked to the external memory by means of two S_AXI_INTERFACES. This implies a doubling of bandwidth to transfer data. Since, most of the time spent by the above kernels is related to the data transfer, this new architecture is able to almost halve their execution times. The performance comparison is presented in table 5.3. While force_compute has reached a final value of $64ms$, neighbor_build a decrease to $2.1s$.

	Single CU	**Two CUs**
force_compute	111 ms	64 ms
neighbor_build	4.3 s	2.1 s

Table 5.3: execution time reduction cause of multiple compute units

Chapter 6

Conclusions and future work

In the present thesis, it has been explored all the main approaches to efficiently perform the porting of native OpenCL kernels onto FPGA. Since the original kernels were intended to execute onto pure-parallel architecture like GPUs, the main effort has mainly affected the research for optimizing the code to the FPGA's architecture and vice versa. As a matter of fact the code optimized to execute on FPGA is significantly different from the one optimized for GPU. This is mainly caused to the different memory bandwidths and the different approach to optimize the process part of the kernel code. While for GPU exits the Single-instruction-Multiple-Data parallelism, for FPGA the goal is reached by completely change the code in order to better guide the Vivado HLS's compiler into optimizations with the insertion of some directives.

The first kind of optimization done, has increased the data transfer between the programmable logic and the external DDR-Memory. This has been mandatory because the main performance limiter in case of Zynq Ultrascale+ is related to the limited bandwidth of the *gmem* AXI interfaces of only 128 bit instead of the common 512-bit of PCI-e based FPGAs. For this reason the some of the variables managed by each of compute intensive kernels (neighbor_build and force_compute) have been re-arranged into local-variables in order to infer as much as possible burst memory-transfers.
The second optimization step has instead affected the re-organization of the "core" kernels' portions, in order to efficiently process all the previously downloaded data. In this case, it has been noticed as the not-fixed upper-bound of loops has limited a lot improvements. In fact Vivado HLS always rely onto fixed-bounded loops in order to promote the pipelining and unrolling techniques. This limitation of the miniMD kernels has not allowed a perfect application of the aforementioned techniques but has equally guaranteed a speed-up of their overall execution. In

this context, it is important to remember that the use of OpenCL language into the high-level-synthesis often leads to less optimizations directives with respect to C codes.

Since the main problem in these application, is related to data transfer, it could probably be a good idea to implement a sort of *On-chip global memories* [18]. This means, to create inside the programmable logic a global memory region with on-chip pipes or buffers in order to speed up the inter-communication of data among all the kernels, and consequently avoid excessive transfer to the external off-chip global memory that requires much more time.
Moreover, in order to obtain further optimizations, it is probably a good solution, to move the focus onto a more general system-level point of view. This implies that there are still global compute unit allocation opportunities that can be exploited in order to use a larger bandwidth with the external memory. That means, direct the optimisation work-flow to a more parallel approach also in case of FPGA world. However, these are beyond the scope of this thesis and are left to future work.

Bibliography

[1] URL: http://www.exanest.eu/.

[2] URL: https://www.xilinx.com/support/documentation/white_papers/wp375_HPC_Using_FPGAs.pdf.

[3] URL: http://www.drdobbs.com/parallel/a-gentle-introduction-to-opencl/231002854?pgno=2.

[4] URL: https://www.xilinx.com/support/documentation/sw_manuals/ug1228-ultrafast-embedded-design-methodology-guide.pdf.

[5] URL: https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf.

[6] URL:https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/v13_4/ug761_axi_reference_guide.pdf.

[7] URL: https://en.wikipedia.org/wiki/High-level_synthesis.

[8] URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf.

[9] URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug1023-sdaccel-user-guide.pdf.

[10] URL: https://en.wikipedia.org/wiki/Molecular_dynamics.

[11] URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug1253-sdx-pragma-reference.pdf.

[12] URL: http://www.hpctoday.com/state-of-the-art/when-are-fpgas-the-right-choice-to-improve-hpc-performance/.

[13] Christian De Schryver et al. "An energy efficient FPGA accelerator for monte carlo option pricing with the heston model". In: *2011 International Conference on Reconfigurable Computing and FPGAs*. IEEE. 2011, pp. 468–474.

[14] Martin C Herbordt et al. "Achieving high performance with FPGA-based computing". In: *Computer* 40.3 (2007).

[15] Xilinx Inc. *SDAccel Development Environment Methodology Guide: Performance Optimization*. 2016.

[16] Manolis Katevenis et al. "Next generation of Exascale-class systems: ExaNeSt project and the status of its interconnect and storage development". In: *Microprocessors and Microsystems* 61 (2018), pp. 58–71.

[17] Rene Mueller, Jens Teubner, and Gustavo Alonso. "Data processing on FPGAs". In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 910–921.

[18] Fahad Bin Muslim et al. "Efficient FPGA implementation of OpenCL high-performance computing applications via high-level synthesis". In: *IEEE Access* 5 (2017), pp. 2747–2762.

[19] Yuliang Pu et al. "An efficient knn algorithm implemented on fpga based heterogeneous computing system using opencl". In: *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*. IEEE. 2015, pp. 167–170.

[20] Rick Weber et al. "Comparing hardware accelerators in scientific applications: A case study". In: *IEEE Transactions on Parallel and Distributed Systems* 22.1 (2011), pp. 58–68.