Received March 12, 2022, accepted April 3, 2022, date of publication April 7, 2022, date of current version April 20, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3165650

FPGA Accelerator for Machine Learning Interatomic Potential-Based Molecular Dynamics of Gold Nanoparticles

SATYA S. BULUSU[1] AND SRIVATHSAN VASUDEVAN[2]
[1]Department of Chemistry, Indian Institute of Technology Indore, Simrol, Indore 453552, India
[2]Department of Electrical Engineering, Indian Institute of Technology Indore, Simrol, Indore 453552, India

Corresponding author: Srivathsan Vasudevan (svasudevan@iiti.ac.in)

This work was supported in part by Department of Science and Technology (DST) Science and Engineering Research Board (SERB), India, under Grant EMR / 2016 / 006579.

ABSTRACT Molecular dynamics (MD) simulations involve computations of forces between atoms and the total energy of the chemical systems. The scientific community is dependent on high-end servers for such computations that are generally sequential and highly power hungry, thereby restricting these computations in reaching experimentally relevant large systems. This work explores the concept of parallelization of the code and accelerating them by exploring the usage of high level synthesis (HLS) based Field Programmable Gate Array (FPGA). This work proposes a hardware and software based interface to implement parallel algorithms in an FPGA framework and communication between the software and hardware interface is implemented. The forces of Au 147 obtained through the ANN based interatomic potentials in the proposed model shows an acceleration of 1.5 times compared with an expensive server with several nodes. Taking this work forward can result in a lab-on-a-chip application and this would potentially be applied onto several large experimentally relevant chemical systems.

INDEX TERMS FPGA accelerator, high performance computing, molecular dynamics.

I. INTRODUCTION

With several advancements in the electronic hardware industry, High Performance Computing (HPC) applications utilize HPC servers. These servers use several conventional processors and graphical processing units (GPUs) [1]–[3]. These advancements play a significant role in increasing the computational power which is very essential for many critical HPC applications. Field Programmable Gate Arrays (FPGAs) have been explored as a possible alternative to HPC servers for quite some time and has been successful in many attempts [4]–[9]. With the recent developments, currently available FPGAs would allow usage of floating-point operators with abundant on-chip resources and hence been introduced as a hardware element in HPC servers [10]–[12].

Despite these advancements, FPGAs were not utilized to their full potential for HPC applications. One reason is that HPC developers usually think that FPGAs are hardware devices that require specific hardware development tools. Even otherwise, HPC developers try to simply run their algorithms on the FPGA framework to parallelize the code using the readily available tools. Although this results in rapid development, it does not provide the complete acceleration one can get from FPGAs [13].

It is well-known that FPGAs can provide upto 100 times acceleration compared to a conventional processor [14], [15]. However, directly using tools to parallelize the code would result in inordinate resource requirement, thereby failing to compete with other HPC alternatives. This work aims at taking up a high performance computational algorithm to determine the structural and dynamical properties of gold nanoparticles using Artificial Neural Network (ANN) based molecular dynamic (MD) simulations. ANN, a machine learning technique, is applied to significantly reduce the computation. Then a hardware-software co-design architecture is proposed to implement the computation through FPGA [16]–[18].

ANN MD simulations involve computations of forces between the atoms and the total energy of the chemical system using ANN based Inter Atomic Potential (IAP). Using the

The associate editor coordinating the review of this manuscript and approving it for publication was Liang-Bi Chen.

knowledge of force acting on atoms, the time trajectories of atoms can be determined by numerically solving Newton's equations of motion. In the last decade, ANN MD simulations were applied to study properties of variety of chemical systems [19], [20]. Recently, it has been shown by our group that structure and dynamics of Au_{147} nanoparticles [21] can be accurately predicted by ANN MD simulations. This approach minimizes the computation so that application of MD can reach real-world systems. Despite these advancements, the ANN based approach still requires HPC servers to run these computations a million times to obtain the properties of materials that are of experimental relevance. Hence, there is a dire need for enhancing the computational power from the hardware perspective.

The objectives of the proposed work are:

- Investigating a hardware-software co-design to perform ANN based MD simulations
- Leveraging FPGA to parallelize the code to reduce computation time
- Optimizing the resources in the FPGA to maximize the performance

In this proposed work, ANN based MD program is divided into two parts, a) calculations of IAP based Intellectual Property (IP) block for energy and atomic forces that will be implemented on FPGA and b) integration of Newton's equations of motion that will be executed on a computer (CPU). Besides, there exist several dependencies in the algorithm which restrict concurrent computation. Identifying each and every step of such processes and paving way for efficient parallelization involves detailed analysis of timing diagrams and proposing alternative programming options without compromising precision and resource utilization. A deep understanding of the code and breaking down into different steps for concurrent computation are novel elements of this work.

The structure of the paper is as follows: Section 2 provides the hardware description followed by the explanation of the HPC algorithm in Sec. 3. Section 4 details the optimizations implemented to get latency improvement. Section 5 shows the optimum usage of resources through algorithm restructuring. Finally, the hardware architecture is explained along with the results obtained in Sec. 6.

II. ACCELERATOR: A HARDWARE DESCRIPTION

In this Section, FPGA based reconfigurable hardware is introduced for high performance computation. Generally, high performance computation algorithm contains several loops. In a conventional processor, the entire computation would be performed through loops in a sequential manner. Since FPGA is a reconfigurable device, these computations can be completely parallelized. This is achieved through multiple replica of the hardware required for performing the computations. Each and every computing cell (CC) would utilize several multiplier adder/accumulator (MAC) blocks, designed in the FPGA. This would provide a very good improvement in the latency. The biggest challenge is that variables stored

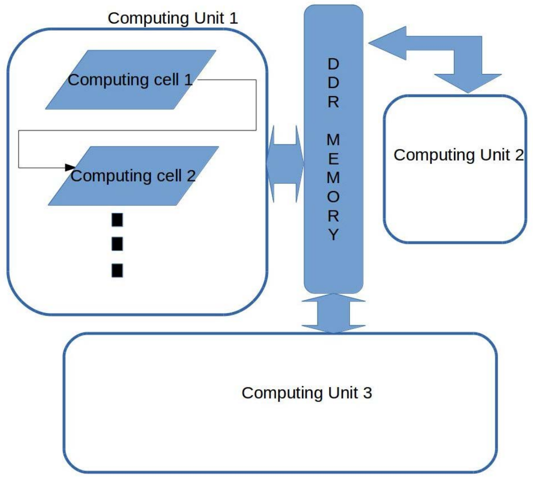

FIGURE 1. Proposed hardware-software co-design using FPGA accelerators.

in arrays would be multi-dimensional, thereby increasing memory requirements. Since FPGAs contain limited memory, this would restrict full use of FPGA accelerators. The second drawback is that often HPC algorithms would have high data dependency. Our proposed approach is to have a hardware-software co-design where the HPC algorithm would be divided into several parts. Each part would use multiple computing cells termed as computing units as shown in Fig. 1. In this way, the frequently used output variables would be stored in DDR memory. Similarly, multi-dimensional arrays are broken down to smaller elements to provide easy access.

III. MATHEMATICAL FORMULATION OF ANN BASED IAP

Here, we will describe the algorithm for constructing the ANN based IAP for gold nanoparticles along with details of functional forms used to derive the ANN based IAP energy and calculations of force acting on atoms.

In Fig. 2, the flowchart describes the algorithm, starting from getting Cartesian coordinates of all the atoms in the nanoparticle and converting them to generalized coordinates.

A set of generalized coordinates or descriptors (D_{ij}) for i^{th} atom in the molecules forms input for ANN. Two layer feed-forward ANN architecture will be used to construct ANN potentials. Energy of individual atom is evaluated as shown in equation below.

$$E_i = \sum_{a=1}^{N_{nh}} W^{23}_{a1}.f^2_a \left[Wb^2_a + \sum_{b=1}^{N_{nh}} W^{12}_{ba}.f^1_b \left(Wb^1_b + \sum_{s=1}^{N_{inp}} W^{01}_{sb}.D_{is} \right) \right] \tag{1}$$

where, E_i is energy of i^{th} atom and N_{nh} is number of hidden nodes. D_{is} is input descriptor functions of length N_{inp} for i^{th} atom. W^{01}_{sb}, W^{12}_{ba} and W^{23}_{a1} are weights connecting from input layer to hidden layer one, hidden layer one to hidden layer

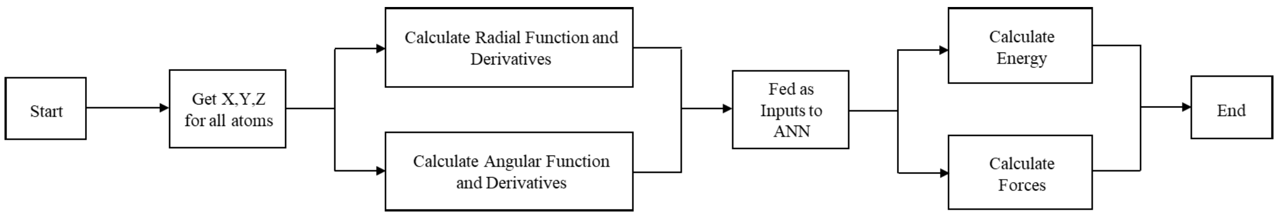
Start
Get X,Y,Z for all atoms
Calculate Radial Function and Derivatives
Calculate Angular Function and Derivatives
Fed as Inputs to ANN
Calculate Energy
Calculate Forces
End

FIGURE 2. Flowchart for serial processing.

two and hidden layer two to output layer respectively. Wb_a^2, Wb_b^1 are bias weights in hidden layer one and hidden layer two respectively. f_a^2 and f_b^1 represents sigmoid function for activation of network. If we use 30 nodes in each hidden layer(this can vary) and 59 input functions (this can vary) then the network comprises of 2520 weights(N_w). The total energy (E) of a nanoparticle is computed by summation all the atomic energies.

$$E = \sum_i E_i \tag{2}$$

For correct representation of forces, along with energy, we also fit $3N$ components of forces using Eq. (3) below.

$$F_k = -\frac{\partial E_{nanoparticle}}{\partial k} = -\sum_{i=1}^{atoms} \frac{\partial E_i}{\partial k} \tag{3}$$

$$= -\sum_{i=1}^{atoms}\sum_{s=1}^{input} \frac{\partial E_i}{\partial D_{is}}\frac{\partial D_{is}}{\partial k} \tag{4}$$

F_k is the force of an atom at coordinate k, where $k \in \{x, y, z\}$.

The generalized coordinates(D_{is}) or descriptors in (1) consists of angular or power spectrum (P_{nl}) parameters and radial parameters. We can obtain P_{nl}, a rotationally and permutationally invariant descriptor, from the coefficients of the basis expansion as given below.

$$P_{nl} = \frac{4\pi}{2l+1}\sum_{m=-l}^{l} c^*_{nlm}c_{nlm} \tag{5}$$

The coefficients c_{nlm} in Eq. (5) are obtained as

$$c_{nlm} = \sum_{i\neq j} e^{-\xi r_{ij}^2} f_c(r_{ij}) Y^*_{lm}(\hat{r}_{ij}) \tag{6}$$

where $Y^*_{lm}(\hat{r}_{ij})$ are the spherical harmonics and $f_c(r_{ij})$ is the cut off function defined as

$$f_c(r_{ij}) = \frac{1}{2}\left[\cos\left(\frac{\pi r_{ij}}{r_c}\right) + 1\right] \tag{7}$$

where, r_c is the cut off radius. Along with the descriptor P_{nl} for the angular information of an atom, radial environment of the atom is required for binding the system such that correct representation of energy and forces is done. Radial functions of an atom i consists of a two body interaction term summed over all the possible neighbors (j).

$$F_R^i = \sum_{i\neq j} e^{-\eta r_{ij}^2} f_c(r_{ij}) \tag{8}$$

Flowchart in Fig. 3 shows a parallel version of the above described algorithm that allows us to run the code in multiple CPU's.

IV. HPC ALGORITHM OPTIMIZATION

A. HARDWARE ANALYSIS: CONVENTIONAL PROCESSOR VS. FPGA

It is important to understand the difference between the hardware working principles of a conventional processor and an FPGA. A small code comprising ANN for calculating the energy of the system, is taken as an example to explain the hardware analysis.

The feed forward ANN, shown in Fig. 4 consists of all the input variables taken from the input buffer. The input values will be multiplied,"$\times$", to first set of weights. The summations,"$+$", of all multiplications give values that will go through a sigmoid function,"$\wr$", to check whether the value is activated or deactivated. The value is then sent to the next layer as an input variable. This process will be repeated, each time using different sets of weights, until we obtain the number of inputs required (a predetermined number) in the next layer. In the present work, we used a 59-30-30-1 setup of feed forward ANN for calculation of energy. It means that the network consists of 59 input values for every atom, which are mapped to energy of the atom, through two hidden layers each of size 30. In this way, the energy of each atom in the molecule is computed and this is repeated for all the 147 atoms sequentially.

In order to convert these computation into the reconfigurable hardware of the FPGA, the algorithm will be converted into multipliers and adders that go along several loops. In a conventional processor, the algorithm would use multipliers and the results will be stored in First in First Out (FIFO) memory that would later be used for the next round of summation.

Since this computation involves several loops, several nodes are required in a HPC server for the same. In contrast, in a FPGA fabric, hardware elements can be re-configured to compute many of these loops concurrently through approaches such as unrolling, pipelining etc. This work explores utilizing the FPGA fabric to perform the HPC algorithm and improving the throughput of the computation (in terms of clock cycles). This is performed through the Vivado HLS tool [22].

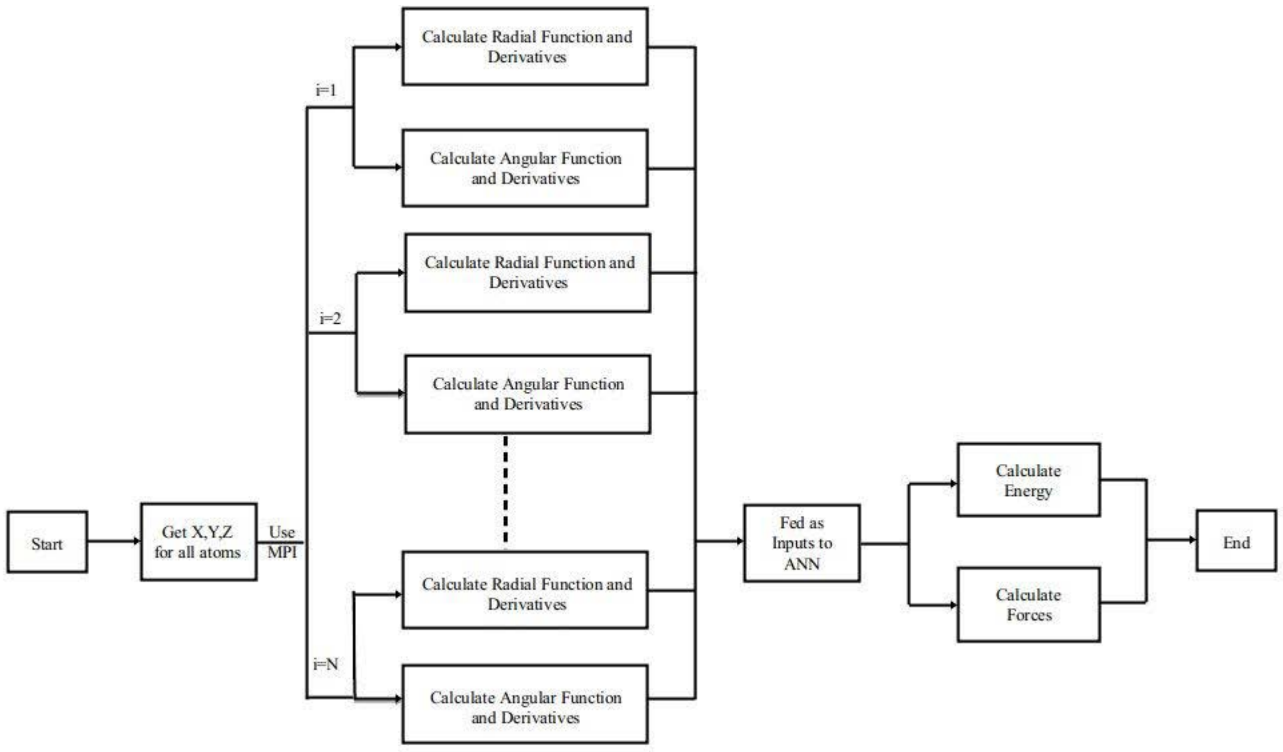
Start
Get X,Y,Z for all atoms
Use MPI
i=1
i=2
i=N
Calculate Radial Function and Derivatives
Calculate Angular Function and Derivatives
Calculate Radial Function and Derivatives
Calculate Angular Function and Derivatives
Calculate Radial Function and Derivatives
Calculate Angular Function and Derivatives
Fed as Inputs to ANN
Calculate Energy
Calculate Forces
End

FIGURE 3. Flowchart for parallel processing using MPI.

B. OPTIMIZATION USING VIVADO HLS

Hardware acceleration is the key in an FPGA fabric and that is essential for improvement in the HPC application in hand. We chose Vivado HLS tool as a solution to our problem. HLS is a tool that takes up algorithmic description written in C-based design flow into RTL schematic, finally exports into an IP block. In order to leverage the use of FPGA's concurrent programming, several optimization tools can be applied. These tools are called directives.

To optimize latency and throughput of the HPC algorithm, unrolling, pipelining and array partitioning directives were applied. Unrolling any loop unrolls the loop thereby allowing concurrent computation, by increasing the hardware resources. Pipelining is a technique where multiple instructions are overlapped during execution. The third directive is array partioning which enables multiple memory access to enable data flow operations. One such snippet of code with the applied directives is provided below.

```
float A[147] [30];
float B[31] [30];
#pragma HLS ARRAY_PARTITION variable=A
block factor=15 dim=2
#pragma HLS ARRAY_PARTITION variable=B
block factor=16 dim=1
for(jj=0;  jj<natoms;  jj++)
{
   for(i=0;  i<30;  i++)
   {
#pragma HLS PIPELINE
     result = 0.0f;
     for(p=0;  p<30;  p++)
     {
       term = A[jj] [p]*B[p] [i];
       result += term;
     }
   }
}
```

In above code, i^{th} loop is to be pipelined and p^{th} loop is to be unrolled. Unrolling p^{th} loop will create 30 parallel copies of it and will require all the 30 values of p^{th} dimension of arrays A and B in single clock cycle. p^{th} dimension of arrays A and B are partitioned so that it is possible to retrieve all the 30 values of both the arrays in single clock cycle. We have partitioned the second dimension of array A[147][30] by a factor of 15, so that it will result in 15 separate arrays. These 15 smaller arrays can give all the 30 elements of second dimension of array IO1 in single clock cycle, as maximum of two elements can be read from a single block RAM. Likewise, first dimension of array B[31][30] is partitioned by a factor of 16 so as to get all the 30 elements in parallel.

Table (1) shows the latency and resource utilization of the above code snippet with and without array partitioning directive. It is clear from this Table that using directives would improve latency and this comes with the increase in hardware. The attempt would be to maximize the

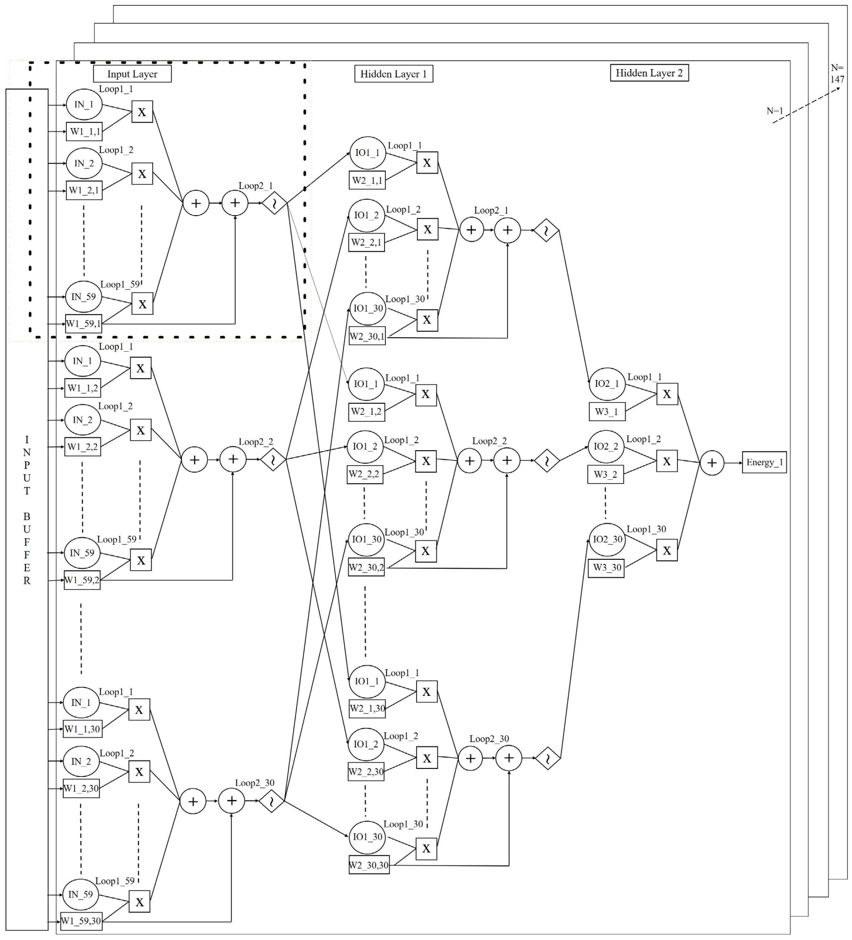

FIGURE 4. Design of ANN algorithm for the calculation of energy of systems.

TABLE 1. Latency/resources with and without applying array partitioning directive (Kintex FPGA family is used for this simulation.

Latency/Resources	Without Array Partitioning	With Array Partitioning
Latency (clock cycles)	7.7×10^4	2.1×10^4
Block RAM	2%	6%
DSP Slices	1%	17%
Flip Flops (FFs)	1%	3%
Look Up Tables (LUTs)	1%	6%

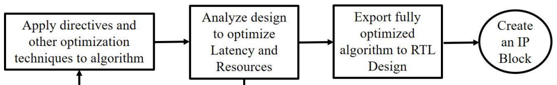
Apply directives and other optimization techniques to algorithm
Analyze design to optimize Latency and Resources
Export fully optimized algorithm to RTL Design
Create an IP Block

FIGURE 5. Complete design flow for IP Block creation.

resources available in the FPGA to decrease the computational time of the algorithm. This is the subject of the next section.

The excercise would be to take up the entire algorithm in the HLS tool, apply directives and optimize them. The final solution would be made as an IP as illustrated in Fig. 5.

V. PERFORMANCE OF THE ACCELERATOR AND TRADEOFFS ADDRESSED DURING OPTIMIZATION

A hardware-software co-design architecture has been chosen and the necessary accelerator IP blocks have been implemented in the FPGA board through the VHDL hardware implementation. As discussed earlier, the best optimization algorithm that we implemented would not fit into the resources of the board. Hence a detailed analysis of the algorithm has been performed to provide appropriate acceleration for the board used. This section explains the different attempts made before the results of the accelerator are presented.

A. COMPLETE ACCELERATION OF THE ALGORITHM

As a first step in any design, different directives were implemented to obtain the maximum acceleration possible. After applying unrolling, pipelining and array partioning directives under the HLS tool, there is an improvement in the latency. However, this comes with a huge increase in the FPGA resources. This summary was obtained from HLS summary report as illustrated in Table (2). It is worthwhile to note that BRAM usage has shot up to 4401% which is extremely high and it is very difficult to look for a FPGA board that

TABLE 2. **Resources on pre-optimizing of algorithm simulated on a Kintex KC705 FPGA platform.**

Resources	Pre-optimization of algorithm
Block RAM	4401%
DSP Slices	40%
Flip Flops (FFs)	11%
Look Up Tables (LUTs)	29%

can contain such large capacity memory elements. Hence, it is very important to optimize the code to reduce the consumption of these resources for effective implementation on a FPGA board. The following subsections address how to optimize the algorithm to suit to the limited resources.

1) BLOCK RAM USAGE

The algorithm implementation in a conventional fashion requires the implementation of several three-dimensional arrays. These arrays would typically use BRAMs and take up all the resources in the FPGA board. One way to optimize is to store all these arrays in the DDR memory which gets transferred from the IP to the DDR through AXI protocol. While this has been implemeted for multi-dimensional arrays that are used repeatedly at different locations of the algorithm, there are other instances where it is not possible to implement 3-D arrays in the DDR (e.g., storing in DDR would compromise heavily on the efficiency of the latency of the algorithm). One such example of a code snippet is as follows: Conventional implementation of the algorithm for variables $diff_x$, $diff_y$, $diff_z$ would result in 3-D arrays obtained from differentiation of c_{nlm}, Fourier coefficients of the algorithm. These arrays would occupy huge memory in the FPGA system as shown in code snippet (a) of table 3. In order to reduce the dimensions of these array variables, a simple interchange of loops was implemented as shown in code snippet (b) of table 3. The resultant algorithm has reduced these variables into one dimensional arrays. Implementation of this strategy at several such locations has reduced the usage of resources to a very significant level (Pre-optimization, the usage of BRAM was 39175 and after this optimization, the number has come down to 266).

$$c_{nlm} = \sum_{i \neq j} G_n(r_{ij}) * Y_{lm}(r_{ij}) \tag{9}$$

where,

$$G_n(r_{ij}) = e^{-\xi r_{ij}^2} f_c(r_{ij}),\ r_{ij} = (x_{ij}, y_{ij}, z_{ij}) \tag{10}$$

$$d_x = \frac{\partial c_{nlm}}{\partial x_{ij}} = \sum_{i \neq j} G_n(r_{ij}) \frac{\partial Y_{lm}(r_{ij})}{\partial x_{ij}} + \frac{\partial G_n(r_{ij})}{\partial x_{ij}} Y_{lm}(r_{ij}) \tag{11}$$

Similarly,

$$d_y = \frac{\partial c_{nlm}}{\partial y_{ij}} = \sum_{i \neq j} G_n(r_{ij}) \frac{\partial Y_{lm}(r_{ij})}{\partial y_{ij}} + \frac{\partial G_n(r_{ij})}{\partial y_{ij}} Y_{lm}(r_{ij}) \tag{12}$$

TABLE 3. **Snippets for multi-dimensional array and it's reduction.**

(a) Using multi -dimensional arrays	(b) Reduction to single variables
for(i=0;i<natoms;	
i++)	for(i=0;i<natoms;
{	i++){
for(j=0;j<natoms;	for(n=0;n<etaang;
j++)	n++){
{	for(l=0;l<lmax;
if(i!=j)	l++){
{	for(m=0;m<2*l+1;
for(l=0;l<lmax;	m++){
l++){	dx=dy=dz=0.0f;
for(m=0;m<2*l+1;	for(j=0; j<natoms;
m++){	j++){
for(n=0;n<etaang;	if(i!=j){
n++){	dCx=G*dYx+dGx*Y;
dCx=G*dYx+dGx*Y;	dCy=G*dYy+dGy*Y;
dCy=G*dYy+dGy*Y;	dCz=G*dYz+dGz*Y;
dCz=G*dYz+dGz*Y;	dx+=dCx;
dx[n][l][m]+=dCx;	dy+=dCy;
dy[n][l][m]+=dCy;	dz+=dCz;
dz[n][l][m]+=dCz;	}}}}}}
}}}}}}	

TABLE 4. **Snippets for variable loop bounds.**

(a) Loop with variable bounds	(b) Loop with fixed bound
for (l=0; l<10;	for (l=0; l<10;
l++)	l++)
{	{
for (m=0; m<2*l+1;	for (m=0; m<19;
m++){	m++)
———	{
}	if(m<2*l+1){
}	———
	}}}

$$d_z = \frac{\partial c_{nlm}}{\partial z_{ij}} = \sum_{i \neq j} G_n(r_{ij}) \frac{\partial Y_{lm}(r_{ij})}{\partial z_{ij}} + \frac{\partial G_n(r_{ij})}{\partial z_{ij}} Y_{lm}(r_{ij}) \tag{13}$$

The multi-dimensional arrays can be replaced with single variables by just interchanging the j^{th} and n^{th} loop as shown in code snippet (b) of table (3). This requires a careful interchange of all the variables of the code in the loops.

2) VARIABLE LOOP BOUNDS AND LOOP DEPENDENCIES

In a nested loop, as shown in code snippet (a) of Table 4, unrolling of the inner loop is not possible as the bounds of the inner loop are dependent on the outer loop. Unrolling would increase the efficiency of the computation. To work around this problem, the dependent variables inside the inner loop were converted into temporary variables. This would acertain the HLS to unroll the inner loops which would reduce the latency of the algorithm.

Similarly, data dependencies would also restrict the efficiency of pipelining. Data dependencies are avoided using temporary variables, rather than using arrays in order to reduce the dependencies. One such example is provided in the following code snippet.

TABLE 5. Snippets for loop dependencies.

(a) Loop dependencies	(b) Reduced loop dependencies

```
P[1]=var1;
P[1−1]= P[1]∗var2;
for(m=7;m>=0;
      m+=−1)
{
if(m<=1−2){
P[m]=P[m+1]∗var3−
      P[m+2]∗var4;
}
}
```
	```
	P[1]=var1;
	P[1−1]=P[1]∗var2;
	tmp1=P[1];
	tmp2=P[1−1];
	for(m=7;m>=0;
	      m+=−1)
	{
	if(m<=1−2)
	{
	trm1=tmp2∗var3;
	trm2=tmp1∗var4;
	tmp3=trm1−trm2;
	P[m]=tmp3;
	tmp1=tmp2;
	tmp2=tmp3;
	}}
	```

TABLE 6. Changes in resources on optimizing loop dependencies.

Resources	Post optimization of algorithm	in %
Block RAM	266	29%
DSP Slices	380	40%
Flip Flops (FFs)	45385	10%
Look Up Tables (LUTs)	59410	28%

In code snippet (a) of Table 5, the result, $P[m]$, is dependent on its previous values $P[m+1]$ and $P[m+2]$. Each iteration of the loop must finish before the next iteration starts. By using temporary variables, access to these registers are available as and when their computation is performed, hence increasing the efficiency of pipelining as shown in the code snippet (b) in Table 5.

After performing these optimizations at different parts of the algorithm, the same table for optimization is re-visited. Table 6 shows the resource information. It is clear that the resources have increased considerably and this is the main reason for getting an improvement in latency. Therefore, optimization of the algorithm is necessary for taking this application forward for any lab-on-a-chip application.

VI. RESULTS AND DISCUSSION

The optimized algorithm has been implemented with other IP blocks in Vivado software [23] and the RTL implementation has been obtained and programmed on the Kintex - 7 KC705 FPGA board. [24] The board contains 16 MBits of BRAM and 840 DSP slices to implement several floating point operations and other operations in an efficient manner. The clock frequency of the board is 100 MHz.

To run the HPC algorithm on the FPGA, a hardware-software co-design has been proposed. The hardware architecture of the accelerator is shown in Fig. 6. The entire accelerator is divided into three modules. The first module is a software module, present in the computer which provides the different XYZ coordinates to the FPGA fabric. For the FPGA fabric, Xilinx Kintex-7 FPGA KC705 evaluation kit [24] was

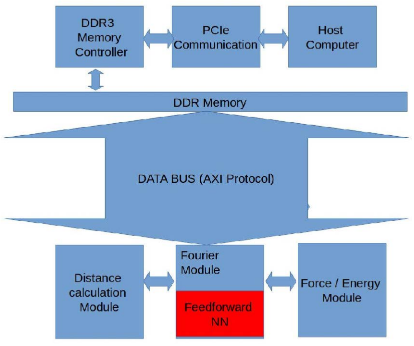

FIGURE 6. Architecture of the accelerator.

used for this application. This FPGA board is used because of its high performance serial transceiver applications, end to end PCIe configuration, advanced memory interface with DDR3 memory and higher number of available block RAM and DSP slices. [24] The communication between the software and hardware accelerator is through PCIe communication. DMA subsystem for PCIe enables host computer to access memory that resides in the FPGA board [25]. While performing the computations in the FPGA, there are certain quantities that require huge floating point arrays. This limits the availability of RAM for concurrent operations. For example, Eq. (9) lists one of the Fourier coefficients, c_{nlm}, as a quantity to be calculated. This quantity, if implemented in FPGA, would result in huge usage of BRAMs. Moreover, these quantities are repeatedly used by several other equations. Care has been taken that these can be pre-calculated (Fourier module) and stored in the DDR memory. Multiple IPs (e.g., shown in red in Fig. 6 can access these quantities directly from the DDR RAM. The communication protocol used for the same is AXI protocol which is a standard high speed communication between different parts of the chip. The Force module would calculate forces and energy and again stores them in the DDR memory. Upon completion of the task, an interrupt would trigger the software module (located in PC) to transfer all these values to the computer through PCIe communication. The software module takes up these forces and calculates the next set of XYZ values which are again fed back to the FPGA fabric. This forms a loop that can run for several thousand times. The entire architecture uses a 32 bit IEEE 754 format [26] floating point numbering system. This format is chosen based on the precision required in the computations and also with the availability of the resources.

For checking, whether the results obtained from FPGA are identical to those obtained from HPC server, we plotted potential energies and norm of gradients of structures along the MD trajectories in Figs. 7 & 8, respectively. From these figures, it is evident that potential energy and norm of gradient for each structure on the MD trajectory obtained from FPGA is identical to that obtained from HPC server. The HPC server(35 nodes), used in this study, is running at 2600 MHz on Genuine Intel processor. The memory in each

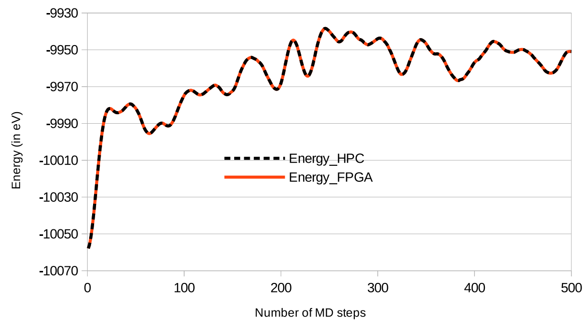

FIGURE 7. Potential energy of Au_{147} vs. number of MD steps calculated on HPC server and FPGA.

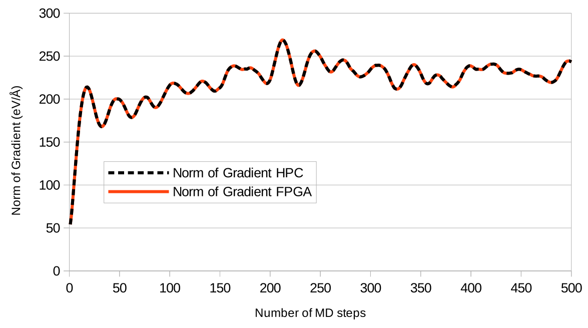

FIGURE 8. Norm of gradient of Au_{147} vs. number of MD steps calculated on HPC server and FPGA.

TABLE 7. Latency/resources utilization for implemented design with and without optimization for Au_{147}.The values in parenthesis are for Au_{309}.

Latency Resources	Without Parallelization	After Parallelization
Latency (clock cycles)	5.9×10^9 (2.6×10^{10})	1.2×10^8 (4.6×10^8)
Block RAM	266 (569)	298 (569)
DSP Slices	380 (380)	512 (470)
Flip Flops (FFs)	45385 (45385)	169931 (163040)
LUTs	59410 (59410)	123382 (123383)

node is 600GB. The computation is performed for both Au_{147} and Au_{309} structures and a performance comparison is done. Latency (shown in terms of number of clock cycles taken for the entire computation) would be a suitable metric for determining the acceleration obtained through the application of directives (parallelization) on FPGA platform. Table 7 provides a comparison between a conventional code and a parallelized code through directives on FPGA platform.

From Table 7, it can be seen that the latency drops to 48 times using optimization directives for Au_{147} and the resources utilization, espesially block RAM's and DSP's, did not increase much. When we increase the size of the system to 309 atoms, the latency drops to 56 times using optimization directives without any big change in block RAM and DSP slices.

In order to get the comparison of performance of MD simulation on FPGA with respect to performance on HPC server, it is important to run the MD code, using optimization techniques discussed above, on FPGA and compare it with running the code that is parallelised using MPI on a HPC server. We used MVAPICH2.2 open source software for implementation of MPI standard on the HPC server. In case of HPC server, the parallel MD code runs at optimal level using 7 CPU's. The time taken to complete 1, 100 and 500 MD steps for Au_{147} are tabulated in Table 8. The values in parenthesis correspond to Au_{309} system. The total time taken to run a complete MD calculations on FPGA includes time taken to run force/energy calculations exclusively on FPGA and the time time taken to integrate Newton's equations of motion on a single CPU. In Table 8, we report the total time taken to run 1, 100 and 500 MD steps on FPGA for Au_{147} and Au_{309}.

TABLE 8. HPC server and FPGA timings for Au_{147} (Au_{309}).

MD Steps	HPC Server Timings	FPGA Timings
1	4.18 s (16.75 s)	2.77 ± 0.02 s (9.65 ± 0.03 s)
100	3.51 min (14.08 min)	140.32 ± 0.33 s (487.55 ± 0.86 s)
500	17.43 min (69.94 min)	694.93 ± 1.95 s (2414.89 ± 0.66 s)

From Table 8, it is clear that for Au_{147}, the acceleration of 1.5 times is achieved using an FPGA in comparision to using 7 CPU's on a HPC server, irrespective of number of MD steps. As the size of the system increases to 309 atoms the acceleration increases to 1.7 times in comparison to HPC server. To check the repeatability of the design, MD simulations for Au_{147} and Au_{309} was performed for 5 times. The standard deviation of the FPGA timings are tabulated in Table 8. The values in parenthesis belong to Au_{309}. We can see that no big deviations are observed in both the cases, confirming that our calculations are repeatable on FPGA board. Figure 9 shows a timing plot in which HPC server and FPGA timings are plotted against number of MD steps for Au_{147}. As the number of MD steps increases, it take more time to run the optimized code on HPC server in comparison to FPGA. Acceleration achieved for running a optimized code using FPGA is 20 mins for 1000 MD steps and there is a 3 fold increase in acceleration when the MD simulations are run for 5000 steps. We can get reasonable increase in acceleration if we run longer MD simulations. In figure (10) we are plotting difference in HPC server and FPGA timings with increase in number of MD steps for Au_{147} and Au_{309}. We can see that as the size of the system increases the acceleration using FPGA increases five times. We can conclude that the accleration using FPGA can be increased for larger systems and for longer MD runs.

For the completeness of this discussion, FPGA based computation is also compared with a very popular alternative, GPU. A snippet of the algorithm, the Fourier module, has been computed in a cloud based Tesla K80 GPU (562 MHz frequency) with a 13 GB memory capacity. It took 416 ms

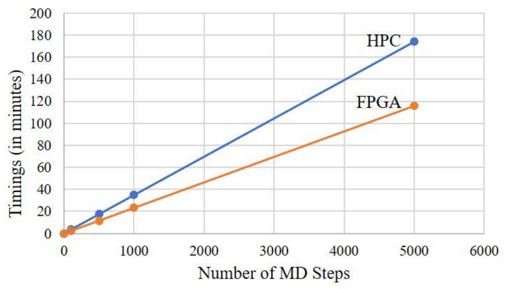

FIGURE 9. Timings of FPGA and HPC server for Au_{147}.

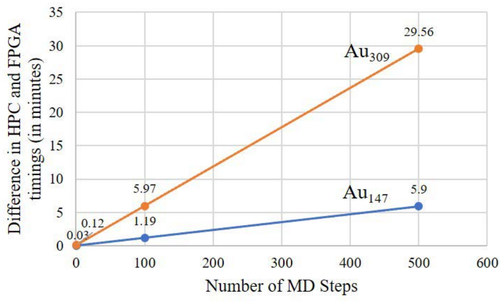

FIGURE 10. Difference in timings of FPGA and HPC server for Au_{147} and Au_{309}.

for the computation. While the same computation was run on the Kintex KC 705 board, the Fourier module can be completed in 375 ms. While there is a very good competition between the two hardware platforms, FPGA would have an advantage in enabling lab-on-a-chip application of several complex computations leading to a product.

Finally, we calculated the speedup ratios for understanding the effect of parallelization on MPI, GPU and FPGA platforms respectively. The speedup ratio using MPI is the ratio of time taken for the optimized algorithm to run on a single processor to the time taken to run on 7 processors. It is 1.76 on a MPI platform. In case of GPU the speedup ratio is 3.13. For FPGA, speedup ratio is defined as the time taken to run the optimized algorithm without using directives to the time taken by using directives. The calculated speedup ratio is approximately 37.84 on a FPGA platform. It is clearly evident that high speedup ratio for FPGA is due to its effective parallelization. It is also important to mention that the speedup ratio for FPGA will increase with increase in size of system.

VII. CONCLUSION

ANN based IAP for Au_{147} and Au_{309} is developed and optimized on Kintex-7 FPGA KC705 board using Vivado HLS. Implementing designs in this way takes advantage of FPGA parallel performance, low power, and low cost. Hardware design running at 100 MHz clock frequency is much faster than its software execution on 7 CPU's each running at 2600 MHz on a HPC server. Au_{147} clearly shows the timing difference between HPC server and FPGA for Au_{147} system. Fully optimized model for Au_{147}(Au_{309}) is about 48(56) times faster in comparison to unoptimized version. Fully optimized 500 steps MD simulations for Au_{147} running on HPC machine takes 17.43 minutes whereas its FPGA equivalent design takes 11.53 minutes.

ACKNOWLEDGMENT

The authors thank Kritika Bhardwaj, M. S. Ansari, Arup Banerjee, and Amod Umarikar, who provided assistance that greatly helped this research. They also thank IIT Indore for providing the server facilities - OMICORN [GenuineIntel 2600.0 MHz].

REFERENCES

[1] N. Kondratyuk, V. Nikolskiy, D. Pavlov, and V. Stegailov, "GPU-accelerated molecular dynamics: State-of-art software performance and porting from Nvidia CUDA to AMD HIP," *Int. J. High Perform. Comput. Appl.*, vol. 35, no. 4, pp. 312–324, Apr. 2021.
[2] N. DeBardeleben, S. Blanchard, L. Monroe, P. Romero, D. Grunau, C. Idler, and C. Wright, "GPU behavior on a large HPC cluster," in *Proc. Euro-Par Parallel Process. Workshops*. Berlin, Germany: Springer, 2014, pp. 680–689.
[3] L. Shi, H. Chen, J. Sun, and K. Li, "VCUDA: GPU-accelerated high-performance computing in virtual machines," *IEEE Trans. Comput.*, vol. 61, no. 6, pp. 804–816, Jun. 2012.
[4] M. Habboush, A. H. El-Maleh, M. E. S. Elrabaa, and S. AlSaleh, "DE-ZFP: An FPGA implementation of a modified ZFP compression/decompression algorithm," *Microprocessors Microsyst.*, vol. 90, Apr. 2022, Art. no. 104453.
[5] R. Kashino, R. Kobayashi, N. Fujita, and T. Boku, "Multi-hetero acceleration by GPU and FPGA for astrophysics simulation on oneAPI environment," in *Proc. Int. Conf. High Perform. Comput. Asia–Pacific Region*, Jan. 2022, pp. 84–93.
[6] M. C. Herbordt, T. VanCourt, Y. Gu, B. Sukhwani, A. Conti, J. Model, and D. DiSabello, "Achieving high performance with FPGA-based computing," *J. Comput.*, vol. 40, no. 3, pp. 50–57, Mar. 2007.
[7] R. Dimond, S. Racanière, and O. Pell, "Accelerating large-scale HPC applications using FPGAs," in *Proc. IEEE 20th Symp. Comput. Arithmetic*, Jul. 2011, pp. 191–192.
[8] E. Jamro, K. Wiatr, and M. Wielgosz, "FPGA implementation of 64-bit exponential function for HPC," in *Proc. Int. Conf. Field Program. Log. Appl.*, Aug. 2007, pp. 718–721.
[9] J. Sheng, C. Yang, A. Sanaullah, M. Papamichael, A. Caulfield, and M. C. Herbordt, "HPC on FPGA clouds: 3D FFTs and implications for molecular dynamics," in *Proc. 27th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2017, pp. 1–4.
[10] M. Vestias and H. Neto, "Trends of CPU, GPU and FPGA for high-performance computing," in *Proc. 24th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2014, pp. 1–6.
[11] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs," in *Proc. Int. Conf. for High Perform. Comput., Netw., Storage Anal. (SC)*, Nov. 2016, pp. 409–420.
[12] V. Z. Jovanović and V. Milutinović, "FPGA accelerator for floating-point matrix multiplication," *IET Comput. Digit. Techn.*, vol. 6, no. 4, pp. 249–256, Jan. 2012.
[13] B. Sukhwani and M. C. Herbordt, "Fast binding site mapping using GPUs and CUDA," in *Proc. IEEE Int. Symp. Parallel Distrib. Process., Workshops Phd Forum (IPDPSW)*, Apr. 2010, pp. 1–8.
[14] C. Yang, T. Geng, T. Wang, R. Patel, Q. Xiong, A. Sanaullah, C. Wu, J. Sheng, C. Lin, V. Sachdeva, and W. Sherman, "Fully integrated FPGA molecular dynamics simulations," in *Proceedings Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2019, pp. 1–31.
[15] C. Pascoe, L. Stewart, B. W. Sherman, V. Sachdeva, and M. W. Herbordt, "Execution of complete molecular dynamics simulations on multiple FPGAs," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Sep. 2020, pp. 1–2.
[16] Y. Maeda and M. Wakamura, "Simultaneous perturbation learning rule for recurrent neural networks and its FPGA implementation," *IEEE Trans. Neural Netw.*, vol. 16, no. 6, pp. 1664–1672, Nov. 2005.

[17] L. Li, J. Zhou, T. Wei, M. Chen, and X. S. Hu, ''Learning-based modeling and optimization for real-time system availability,'' *IEEE Trans. Comput.*, vol. 70, no. 4, pp. 581–594, Apr. 2021.
[18] H. F. Restrepo, R. Hoffmann, A. Perez-Uribe, C. Teuscher, and E. Sanchez, ''A networked FPGA-based hardware implementation of a neural network application,'' in *Proc. IEEE Symp. Field-Program. Custom Comput. Mach.*, Apr. 2000, pp. 337–338.
[19] J. Behler, ''Perspective: Machine learning potentials for atomistic simulations,'' *The J. Chem. Phys.*, vol. 145, no. 17, pp. 170–901, Nov. 2016.
[20] A. C. Mater and M. L. Coote, ''Deep learning in chemistry,'' *J. Chem. Inf. Model.*, vol. 59, no. 6, pp. 2545–2559, Jun. 2019.
[21] S. Jindal, S. Chiriki, and S. S. Bulusu, ''Spherical harmonics based descriptor for neural network potentials: Structure and dynamics of Au147 nanocluster,'' *J. Chem. Phys.*, vol. 146, no. 20, pp. 204–301, May 2017.
[22] *Xilinx: Vivado Design Suite User Guide High-Level Synthesis UG902*, Xilinx, USA, 2014.
[23] (2019). *Xilinx: Vivado Design Suite and User Guide, Release Notes, Installation, and Licensing, UG973 (v2019.2)*. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html
[24] *Xilinx: KC705 Evaluation Board for the Kintex-7 FPGA User Guide UG810*, Xilinx, USA, 2019.
[25] *Xilinx: DMA/Bridge Subsystem for PCI Express PG195 (v4.0)*, Xilinx, USA, 2017.
[26] *IEEE standard for floating-point arithmetic*, IEEE Standard 754-2008, pp. 1–70, Aug. 29, 2008, doi: 10.1109/IEEESTD.2008.4610935.

SATYA S. BULUSU received the Ph.D. degree from the Department of Chemistry, University of Nebraska–Lincoln, Lincoln, NE, USA. After couple of postdoctoral stints at York University and the University of New Brunswik, Canada, he moved to the Indian Institute of Technology Indore, where he is currently an Associate Professor with the Department of Chemistry. His research interests include developing machine learning-based inter atomic potentials for chemical systems and FPGA-based computing.

SRIVATHSAN VASUDEVAN received the Ph.D. degree from the School of Electrical and Electronics Engineering, Nanyang Technological University, Singapore. After a short stint at Singapore General Hospital, he moved to the Indian Institute of Technology Indore, where he is currently an Associate Professor with the Department of Electrical Engineering. His research interests include FPGA-based computing, biomedical instrumentation, and point of care device development.