Acceleration of ANN based Molecular Dynamics using GPU and FPGA

M.Tech Thesis

By
KISHORE REDDY KURAPATI

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY INDORE
JUNE 2023

Acceleration of ANN based Molecular Dynamics using GPU and FPGA

A Thesis

Submitted in partial fulfillment of the
requirements for the award of the degree
of
Master of Technology

by

KISHORE REDDY KURAPATI

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY INDORE
JUNE 2023

INDIAN INSTITUTE OF TECHNOLOGY INDORE

CANDIDATE'S DECLARATION

I hereby certify that the work which is being presented in the thesis entitled **ACCELERATION OF ANN BASED MOLECULAR DYNAMICS USING GPU AND FPGA** in the partial fulfillment of the requirements for the award of the degree of **MASTER OF TECHNOLOGY** and submitted in the **DEPARTMENT OF ELECTRICAL ENGINEERING, Indian Institute of Technology Indore**, is an authentic record of my own work carried out during the time period from August 2021 to June 2023 under the supervision of **Dr. Srivathsan Vasudevan,** Professor, Department of Electrical Engineering, Indian Institute of Technology, Indore and **Dr. Satya S. Bulusu,** Associate Professor, Department of Chemistry, Indian Institute of Technology, Indore.

The matter presented in this thesis has not been submitted by me for the award of any other degree of this or any other institute.

Kishore.K
Signature of the Student
Date: 05/06/2023
KISHORE REDDY KURAPATI

This is to certify that the above statement made by the candidate is correct to the best of our knowledge.

V. Srivathsan
Signature of the Supervisor
Date: 6/6/2023
DR. SRIVATHSAN VASUDEVAN

S. S. Bulusu
Signature of the Supervisor
Date: 6/6/23
DR. SATYA S. BULUSU

MR. KISHORE REDDY KURAPATI has successfully given his M.Tech. Oral Examination held on **May 11th 2023.**

V. Srivathsan Signature of the Supervisor Date: 6/6/2023 **Dr. Srivathsan Vasudevan**	S. S. Bulusu Signature of the Supervisor Date: 6/6/23 **Dr. Satya S. Bulusu**	Convener, DPGC Date: **Dr. Swaminathan Ramabadran**
Signature of PSPC Member Date: **Dr. Swaminathan Ramabadran**	Signature of PSPC Member Date: **Dr. Eswara Prasad Korimilli**	Signature of PSPC Member Date: **Dr. Trapti Jain**

Acknowledgments

This thesis work would not have been possible without the support of several individuals. I would like to express my heartfelt gratitude to all those who have supported me throughout this journey.

First and foremost, I am immensely grateful to Prof. Srivathsan Vasudevan for providing me with the opportunity to work on this project. Working on this project, I have acquired invaluable knowledge and skills that have greatly enriched my learning experience.

I would also like to extend my thanks to Dr. Satya S Bulusu for his continuous support and guidance throughout the course of the project work. His insights and feedback have been instrumental in shaping the project work.

I am grateful to Dr. Swaminathan Ramabadran, Dr. Eswara Prasad Korimilli, Prof. Trapti Jain, and Prof. Abhinav Kranti for their time and valuable feedback during the assessment of my progress. Their inputs have significantly contributed to the improvement of my thesis.

Special appreciation goes to Mr. Harshit Verma, my senior, for his guidance during the initial stages of this work. His insights and experience have been instrumental in laying a strong foundation for my work.

I would also like to express my gratitude to Mr. Pracheta Chatterjee, Mr. Ankit Patel, and all the other lab members for their support and assistance throughout my project. Their collaboration and shared knowledge have enhanced the quality of my work.

Lastly, I would like to acknowledge my family for their unwavering love, encouragement, and support.

Kishore Reddy Kurapati

This work is dedicated to my parents and my sister.

Abstract

High Performance Computing (HPC) technology uses large number of powerful processors, working in parallel, to process large amounts of data and solve complex problems at very high speeds. Processors used in the HPC systems were primarily Central Processing Units (CPUs). As the data became multidimensional, the use Graphics Processing Units (GPUs) as co-processors for CPUs in HPC systems became popular making the HPC systems heterogeneous. GPUs process data parallelly with higher throughput than the CPUs due to the large number of processing cores. CPUs have higher per-core throughput compared to the GPUs. Keeping this in mind, the workload in a heterogenous computing system is divided between the different processors in such a way to provide maximum throughput. Field Programmable Gate Arrays (FPGAs) provide an architecture with large number of programmable hardware components which facilitates implementing various logic on the hardware.

This project deals with accelerating ANN based Molecular Dynamics calculations by utilizing the hardware accelerators, GPU and FPGA. In the first phase of the project, a heterogeneous computing system with one CPU and one GPU is used to calculate the energy and force on each atom from their positions in a Au nanocluster. The computed forces values are used to calculate the new coordinates of atoms, thereby forming a closed loop. CUDA Programming Model is employed to program the workload onto the Heterogeneous Computing System. The sequential computations of the workload are processed on the CPU and the parallel computations of the workload are offloaded to the GPU for processing. This approach of offloading along with effectively utilizing the GPU cores provides better throughput than computations on CPU system.

In the second phase of the project, the force and energy calculations are computed on Genesys2 Kintex FPGA board. The coordinates are transferred from PC to FPGA board via UART, the FPGA board does the required calculations and sends back the force and energy values to the PC via UART. The required logic to be implemented on FPGA is first packaged as an Intellectual Property (IP) block. Various techniques like pipelining, unrolling etc., are employed to optimize the IP. System Design is done including the IP block which is controlled by a Microblaze processor. The whole hardware design is synthesized and implemented on the FPGA Board using Vivado Tools.

Contents

List of Figures

List of Algorithms

List of Acronyms

HPC	High Performance Computing
CPU	Central Processing Unit
GPU	Graphics Processing Unit
FPGA	Field Programmable Gate Array
MD	Molecular Dynamics
ANN	Artificial Neural Network
ANN-MD	ANN based Molecular Dynamics
IP	Intellectual Property
SIMD	Single Instruction Multiple Data
SPMD	Single Program Multiple Data
CUDA	Compute Unified Device Architecture
SM	Streaming Multiprocessor
CLB	Configurable Logic Block
LUT	Look Up Table
MUX	Multiplexer
HDL	Hardware Design Language
AXI	Advanced eXtensible Interface
HLS	High Level Synthesis
RISC	Reduced Instruction Set Computer

Page Intensionally left blank

Chapter 1

Introduction

This chapter provides an introduction to the various topics which are necessary in understanding the work carried out.

1.1 High Performance Computing

High-performance computing (HPC) refers to the use of powerful computing resources to solve complex problems that would be too difficult or time-consuming to tackle with traditional computing methods. HPC systems typically use parallel processing and distributed computing to execute tasks faster and more efficiently than a single processor could. These systems often employ specialized software and hardware, such as graphics processing units (GPUs) and field-programmable gate arrays (FPGAs), to perform calculations in parallel. HPC has applications in a wide range of fields, from scientific research to business analytics, and is essential for modelling complex phenomena, simulating real-world events, and analysing large amounts of data. HPC is a rapidly evolving field that continues to push the boundaries of computing technology.

In HPC, heterogeneous computing is used to accelerate specific components or aspects of complex simulations or calculations. For example, GPUs may be used to accelerate molecular dynamics simulations or machine learning algorithms, while CPUs may be used for other computational tasks, such as data analysis or visualization. HPC systems may also use specialized hardware accelerators, such as FPGAs or ASICs, to perform specific computations more efficiently. HPC and heterogeneous computing are complementary approaches that can be used to design highly efficient and scalable computing systems for scientific and engineering applications.

1.2 Heterogeneous Computing

Heterogeneous computing refers to the use of different types of computing devices and architectures in a single system to optimize performance and efficiency. Heterogeneous computing is typically used to accelerate specific tasks or computations, by offloading them to specialized devices that are designed to perform these tasks more efficiently. Offloading tasks to specialized processing units can improve overall system performance by reducing the workload on the main processor and leveraging the capabilities of specialized hardware. The main advantage of heterogeneous computing is that it allows for more efficient use of resources and can significantly improve performance. For example, GPUs are well-suited for tasks that require a large amount of parallel processing, while CPUs are better at handling tasks that require a high level of sequential processing. By combining these two types of processors, a system can be designed to handle both types of tasks efficiently. Heterogeneous computing is widely used in scientific computing, data analysis, machine learning, and other fields that require high-performance computing. The use of heterogeneous computing is expected to continue to grow as the demand for more efficient and powerful computing systems increases.

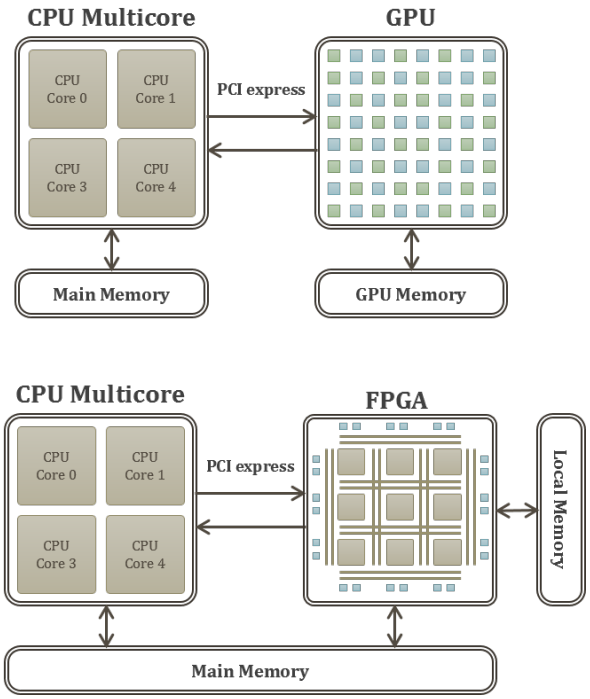


Figure 1.1: Heterogeneous computing

1.3 Molecular Dynamics

Molecular Dynamics (MD) is a computational simulation technique used to study the dynamic behaviour of atoms and molecules over time. It provides insights into the movements, interactions, and properties of these particles by numerically solving their equations of motion. In MD simulations, the positions and velocities of individual atoms or molecules are tracked and updated at discrete time steps. The behaviour of the system is governed by classical mechanics, assuming that the atoms or molecules follow Newton's laws of motion. These laws describe how forces acting on the particles determine their acceleration and subsequent motion.

To perform an MD simulation, the system is typically described using a mathematical model called a force field. The force field includes potential energy functions that represent the interactions between atoms or molecules, accounting for factors such as bond stretching, angle bending, dihedral rotations, and non-bonded interactions (such as van der Waals forces and electrostatic interactions). By integrating the equations of motion over time, MD simulations generate a trajectory of the system's behaviour. This trajectory provides information about various properties, such as particle positions, velocities, energies, and structural changes. Additionally, statistical analysis techniques can be applied to the trajectory to obtain thermodynamic quantities, diffusion coefficients, reaction rates, and other dynamic properties.

MD simulations have broad applications in fields such as chemistry, physics, materials science, and biology. They help researchers understand the behaviour of complex systems at the molecular level, investigate reaction mechanisms, predict thermodynamic properties, and provide insights into the structure-function relationships of molecules and materials.

1.4 ANN-based Molecular Dynamics

ANN-based Molecular Dynamics (ANN-MD) is a computational approach that combines artificial neural networks (ANNs) with traditional molecular dynamics simulations. In this approach, ANNs are used to model the potential energy surface of a system, enabling accelerated simulations while maintaining reasonable accuracy.

Here's an overview of how ANN-MD works:

1. Training the ANN Potential: A large dataset of atomic configurations, along with their corresponding energies and forces obtained from ref-

erence calculations (e.g., DFT), is used to train an ANN. The ANN learns to approximate the potential energy surface based on these input configurations and reference data.

2. Input Representation: Atomic configurations, such as positions, velocities, and atomic species, are encoded as input features for the ANN. Additional features like radial and angular descriptors can also be incorporated to capture the local atomic environment.

3. Architecture and Training: The architecture of the ANN, including the number of layers and neurons, is designed. Training algorithms, such as backpropagation and optimization techniques, are employed to adjust the ANN parameters and minimize the discrepancy between the predicted and reference energies and forces.

4. MD Simulation: In each MD simulation step, the forces on the atoms are calculated by evaluating the gradient of the potential energy surface predicted by the ANN with respect to the atomic positions. Numerical integration methods, such as the Verlet algorithm, are used to update the atomic positions and velocities.

5. Ensemble and Temperature Control: Ensemble methods, such as the NVE, NVT, or NPT ensembles, can be employed to control the system's thermodynamic properties during the MD simulation. These methods ensure that the system reaches the desired temperature, pressure, or energy distribution.

6. Validation and Analysis: The accuracy and reliability of the ANN potential are assessed by comparing simulation results with reference calculations and experimental data. Analysis techniques, such as radial distribution functions, diffusion coefficients, or structural properties, are employed to study the system's behaviour.

ANN-MD offers several advantages over traditional MD simulations. It can significantly accelerate the simulations while maintaining reasonable accuracy, enabling the study of larger and longer timescale systems. However, careful validation of the ANN potential against reference data and experimental observations is essential to ensure its reliability and applicability to the specific system of interest.

1.5 Interatomic Descriptors

Interatomic descriptors are a class of features or properties that describe the interactions between pairs of atoms in a molecular system. These descriptors are used in computational chemistry and material science to analyse the properties and behaviour of molecular systems. They are used in structural analysis of molecular systems. In structural analysis of nanoparticle system using Machine Learning, they are used as inputs to the neural networks to predict energy and forces of atoms in the nanoparticle system. Though interatomic descriptors are very useful, calculating them involves very complex mathematical computations and is very time consuming. Accelerating the calculation of the interatomic descriptors plays a huge role in speeding up the whole process of predicting energy and forces of atoms in nanoparticle system. In this project radial distribution function and power spectrum are used as descriptors to define the local environment of the atoms in a nanoparticle.

1.6 Motivation

ANN-MD calculations involves calculating energy and force on atoms.The calculation of energy and force experienced by atoms in a nanoparticle structure using Interatomic potentials involves repetitive computations and are very time consuming if processed serially on a CPU. Parallelising these computations can hugely reduce the time taken for calculating the energy and force values, thereby improve the performance of MD system. Hardware accelerators like GPU and FPGA provides architecture to improve the throughput of the calculations. GPU provides an architecture with large number of processing cores to run large number of instructions in parallel at the same time. Utilizing this architecture of the GPU can provide better throughput than the CPU for calculating the energy and force values [1], [2], [3]. FPGA provides an architecture with large number of programmable hardware components which facilitates implementing various logic on the hardware which can be used to accelerate the energy and force calculations [4].

1.7 Objective

The objective of the thesis work is to use heterogeneous computing system with a hardware accelerator working along with a CPU to accelerate ANN-MD calculations. The idea is to offload repetitive and time consuming energy and force calculations onto the hardware accelerators while performing serial

calculations on the CPU. Both GPU and FPGA provide different architectures that can be used to accelerate the calculations. While achieving the objective of accelerating the ANN-MD calculations, focus is given to optimizing the acceleration results.

1.8 Organization of the Thesis

This section provides an overview of how the thesis is organized. The thesis consists of five chapters whose contents are as follows:

Chapter 1 provides an introduction to the various topics which are necessary in understanding the need for the thesis work. The chapter also describes the motivation and objective of the thesis work.

Chapter 2 provides a comprehensive exploration of Graphics Processing Units (GPUs) architecture and the memory hierarchy present in the GPUs. It also discusses the CUDA Programming Model and various CUDA APIs used while programming GPUs are also discussed along with examples.

Chapter 3 provides an exploration of Field Programmable Gate Arrays (FPGAs). The FPGA architecture and the different programmable hardware components in an FPGA are discussed in this chapter.

Chapter 4 discusses the methodology used in realizing the project objectives. Methods used to parallelise the calculations to run on GPU and various algorithms used in the project are also discussed in the chapter. The chapter also discusses the steps involved in implementing the calculations on an FPGA.

Chapter 5 discusses the results obtained from ANN-MD calculations on CPU + GPU system for Au_{147} and Au_{309} systems.

Chapter 2

Graphics Processing Units

Previously, Graphics Processing Units (GPUs) were primarily utilized for rendering abstract geometric objects generated by various programs running on the CPU. The GPU featured programmable shader units, namely the vertex shader, responsible for processing the vertices of three-dimensional objects provided by the user, and the fragment or pixel shader, used to apply colors from textures and implement complex visual effects such as scene lighting for each fragment or pixel. Both shader units were programmable and capable of parallel processing.

As the semiconductor industry advanced, highly parallel processing units were integrated into a unified central unit known as the Unified Shading Unit. This higher-level abstraction combined the functionalities of the vertex and fragment shaders. The Unified Shading Unit comprises a group of parallel processing units, often referred to as multiprocessors. Each multiprocessor consists of multiple stream processors, as well as special function units dedicated to memory access, instruction fetching, and scheduling. While the initial shader unit operated in a single-instruction multiple data (SIMD) fashion, subsequent developments introduced support for branching, resulting in the new streaming processors functioning as single-program multiple data (SPMD) units [5].

Overall, the evolution of GPUs has seen a transition from separate vertex and fragment shaders to a unified architecture, consolidating parallel processing capabilities within multiprocessors and stream processors. These advancements have led to enhanced programmability and parallelism, enabling GPUs to handle a wide range of computationally intensive tasks beyond traditional graphics rendering.

2.1 GPU Architecture

GPU architecture is fundamentally different from that of traditional Central Processing Units (CPUs). While CPUs are optimized for executing a few tasks sequentially, GPUs are designed to handle a massive number of parallel tasks simultaneously. This parallelism is achieved through the use of multiple processing units and a specialized memory hierarchy.

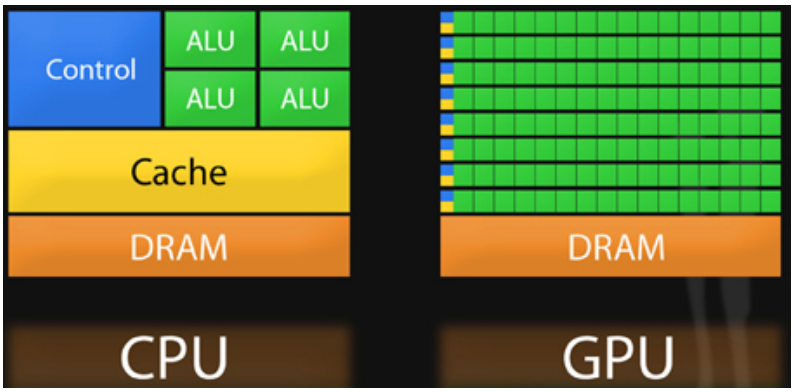


Figure 2.1: Tesla T4 Architecture

2.1.1 Streaming Multiprocessors (SMs)

The primary processing units in a GPU are the Streaming Multiprocessors (SMs). Each SM contains multiple CUDA (Compute Unified Device Architecture) cores, which are responsible for executing individual threads. The number of SMs varies depending on the GPU model, and each SM can handle a large number of threads concurrently.

2.1.2 CUDA Cores

CUDA Cores, also known as shaders or stream processors, are the basic processing units within an SM. These cores are responsible for executing instructions in parallel and performing arithmetic and logic operations on data. Modern GPUs can have thousands of CUDA cores, enabling massive parallelism and high computational throughput.

2.2 Memory Hierarchy in GPUs

GPUs have a complex memory hierarchy that includes various types of memory with different access speeds and capacities. The primary memory components in a GPU are:

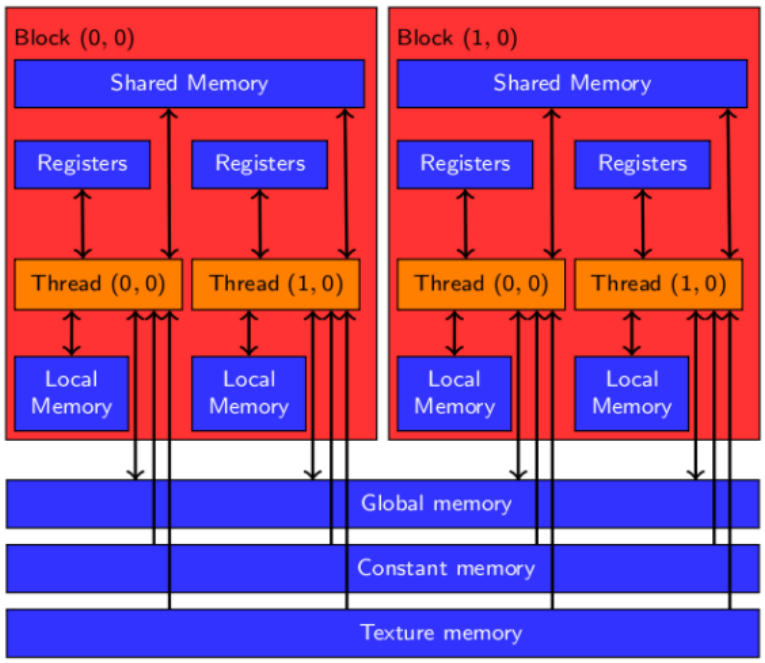


Figure 2.2: CUDA enabled Graphics Processing Unit's memory model

Global Memory

Global memory is the largest memory space in a GPU and is accessible by all threads. It has high capacity but relatively high latency.

Shared Memory

Shared memory is a small, fast, and programmable memory space that is shared among threads within an SM. It allows for efficient data sharing and communication between threads.

Texture memory

Texture memory is a specialized read-only memory optimized for 2D and 3D texture accesses. It provides high bandwidth and caching capabilities for texture data.

Constant Memory

Constant memory is another read-only memory optimized for storing constants and configuration data. It offers low latency and high bandwidth access for frequently used data.

Register File

Each CUDA core has its own set of registers for storing intermediate data during computations. Registers provide extremely fast access but have limited capacity.

2.3 Parallel Processing in GPUs

Parallel processing is the cornerstone of GPU architecture and is achieved through the execution of multiple threads in parallel. GPUs employ two primary types of parallelism:

2.3.1 Task-Level Parallelism

Task-level parallelism, also known as coarse-grained parallelism, involves executing multiple independent tasks concurrently. Each task can be assigned to a separate thread, and the GPU can switch between threads to maximize resource utilization. This type of parallelism is well-suited for applications with independent computations, such as graphics rendering or data parallel algorithms.

2.3.2 Data-Level Parallelism

Data-level parallelism, also known as fine-grained parallelism, involves performing the same operation on multiple data elements simultaneously. GPUs excel at data-level parallelism through the use of SIMD (Single Instruction, Multiple Data) execution. SIMD allows a single instruction to be applied to multiple data elements, leveraging the parallel execution capabilities of

CUDA cores. This type of parallelism is particularly beneficial for algorithms that process large datasets in parallel, such as matrix operations or image processing.

2.4 Programming GPUs

To harness the power of GPUs, developers use programming frameworks such as CUDA or OpenCL. These frameworks provide a software interface to access and utilize the parallel computing capabilities of GPUs. They allow developers to write code that can be executed in parallel on the GPU, distributing tasks across multiple threads and managing data transfers between the CPU and GPU memory. In this project, CUDA framework is used to program an Nvidia Tesla T4 GPU.

2.5 Advantages of GPU Architecture

- **Parallelism and Performance:** GPUs excel at executing a large number of parallel tasks simultaneously, leading to significantly higher computational performance compared to CPUs for parallelizable workloads.
- **Scalability:** GPUs are highly scalable, allowing for the addition of more GPUs to increase computational power. This scalability is essential for large-scale simulations and data-intensive applications.
- **General-Purpose Computing:** With the advent of frameworks like CUDA, GPUs have become increasingly versatile, enabling general-purpose computing beyond graphics rendering. They are widely used for scientific simulations, deep learning, data analytics, and more.

2.6 CUDA Programming Model

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA for utilizing the computational power of GPUs. It allows developers to write code that can be executed on GPUs to accelerate parallel computations. This section provides an overview of the CUDA programming model, along with code snippets and examples to illustrate its usage.

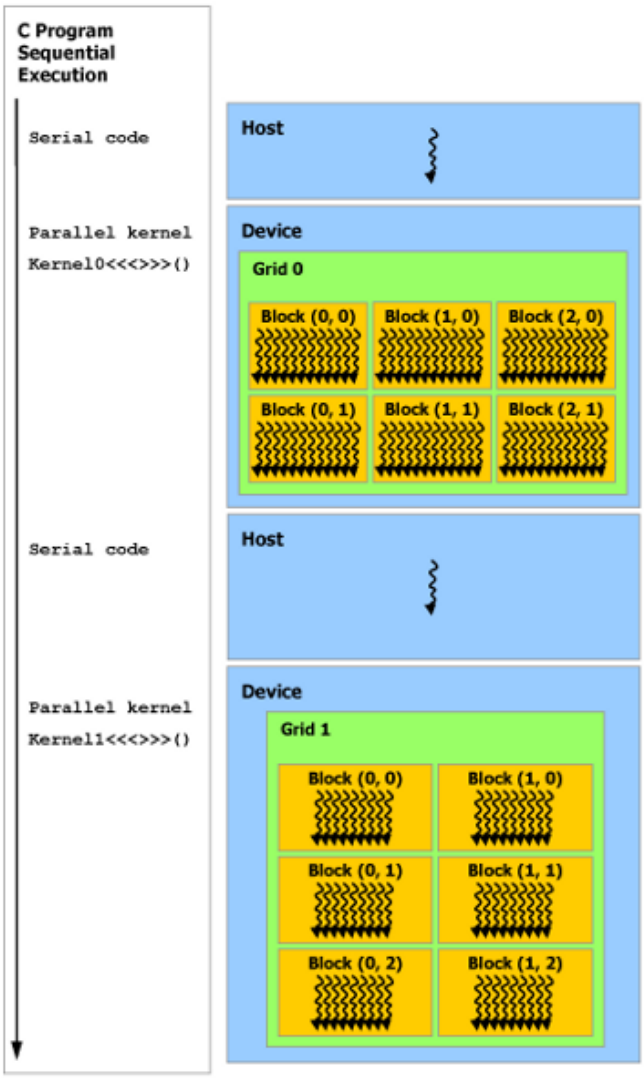


Figure 2.3: CUDA Programming Model

CUDA Programming Model Overview

The CUDA programming model consists of two main components: host code (executed on the CPU) and device code (executed on the GPU). The host code manages data transfers between the CPU and GPU, while the device code performs computations in parallel on the GPU.

In CUDA, the grid, blocks, and threads are key components that define the organization and execution of parallel computations on a GPU. They are closely related to the hardware components of the GPU and play a crucial

role in achieving efficient parallel processing. Let's delve into each of these components and their relationship with the GPU hardware.

2.6.1 Grid

The grid is the highest level of organization in CUDA and represents the overall execution configuration of a kernel. It consists of multiple blocks, and the total number of blocks in a grid is specified during kernel invocation. The grid provides a way to distribute computations across the available hardware resources of the GPU.

In terms of GPU hardware, the grid aligns with the streaming multiprocessors (SMs). Each block within the grid can be assigned to a different SM, allowing for concurrent execution of multiple blocks.

2.6.2 Blocks

A block is a logical unit of parallel computation within the grid. It represents a group of threads that can be scheduled and executed together within an SM. The number of threads per block is determined during kernel invocation.

Blocks are essential for achieving efficient memory access and synchronization. They can be thought of as a mechanism for partitioning the workload into smaller manageable units, allowing for better resource utilization and coordination within the GPU hardware.

2.6.3 Threads

Threads are the smallest units of execution in CUDA. They are individual instances that perform computations in parallel. Threads within a block are organized into a one-, two-, or three-dimensional structure, depending on the problem's nature.

Threads are executed in parallel within an SM, and each thread is assigned a unique index. These indices are used to access data and perform computations in parallel.

In terms of GPU hardware, each thread is executed by a CUDA core (also known as a shader or stream processor). Modern GPUs have a vast number of CUDA cores, enabling massive parallelism and high computational throughput.

Relationship with GPU Hardware

The organization of grids, blocks, and threads in CUDA is closely related to the underlying GPU hardware components, such as streaming multiprocessors (SMs), CUDA cores, and memory hierarchy.

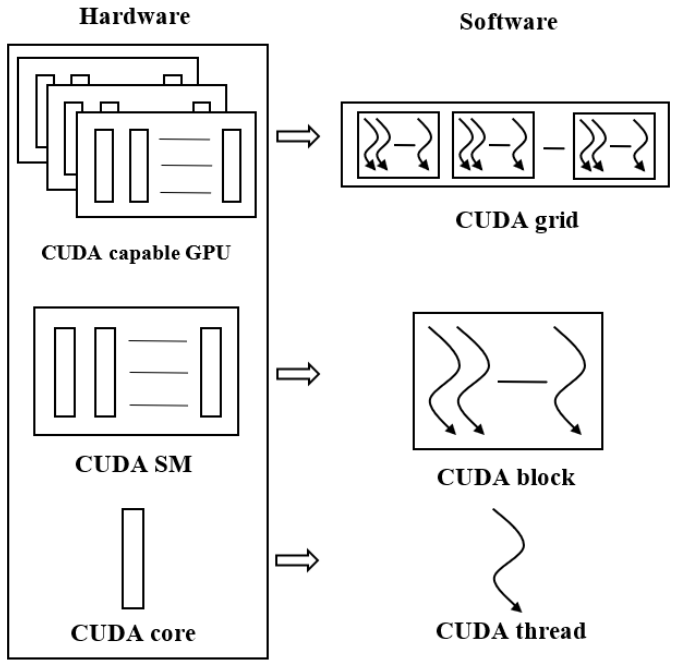


Figure 2.4: Relation between CUDA grid, block and thread to GPU Hardware

- Grids are mapped to the streaming multiprocessors (SMs) of the GPU. Each block within the grid can be assigned to a separate SM, allowing for concurrent execution of multiple blocks, leveraging the parallel processing capabilities of the GPU.
- Blocks are scheduled and executed within the SMs. The SMs manage the execution of threads within a block, assigning resources and coordinating their execution.
- Threads are executed by individual CUDA cores within the SMs. Each CUDA core performs computations independently on a single thread, leveraging the parallelism provided by multiple CUDA cores.

The memory hierarchy of the GPU, including global memory, shared memory, and registers, is also closely related to the organization of grids,

blocks, and threads. Efficient memory access and synchronization within blocks and threads are crucial for achieving high-performance parallel computations on the GPU.

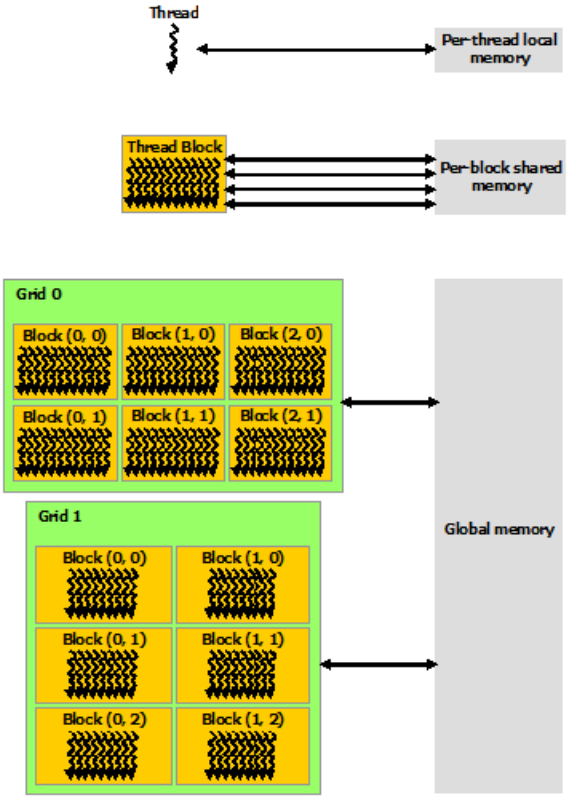


Figure 2.5: Relation between CUDA grid, block and thread to GPU Memory

2.6.4 CUDA Program Example

To write CUDA code, you need to define functions that will be executed on the GPU, known as kernels. Kernels are invoked by the host code and run in parallel by multiple threads on the GPU. These threads are organized into blocks, and blocks are further grouped into a grid. A typical CUDA program consists of the following steps:

1. Device Selection and Memory Management
2. Kernel Definition and Invocation
3. Memory Transfer and Synchronization

Device Selection and Memory Management

- Select the GPU device(s) to use.
- Allocate and manage memory on the GPU.

```
cudaSetDevice(deviceId); //Device Selection
cudaMalloc((void**)&deviceInput, size); //Device Allocation

cudaMemcpy(deviceInput, hostInput, size, cudaMemcpyHostToDevice);
//Memory Copy from Host to Device

cudaMemcpy(hostInput, deviceInput, size, cudaMemcpyDeviceToHost);
//Memory Copy from Device to Host
```

The Host and Device memory allocation and de-allocation are controlled by the host code (CPU Code). In previous versions of CUDA, separate host variables and device variables were allocated memory using ***cudaHostMalloc*** and ***cudaMalloc*** which are CUDA runtime APIs.

- ***cudaHostMalloc:*** This function is used to allocate pageable (host) memory. Pageable memory resides in the host (CPU) memory space and can be accessed by both the host and the device (GPU). However, accessing pageable memory from the device may incur additional latency due to page faults.
- ***cudaMalloc:*** This function is used to allocate device memory. Device memory resides in the memory space of the GPU and is accessible only by the device. Accessing device memory is typically faster than accessing pageable memory.

```
//Host and Device Memory Management
void* hostPtr, devicePtr;
cudaHostMalloc(&hostPtr, size);
cudaMalloc(&devicePtr, size);
...
cudaHostFree(hostPtr);
cudaFree(devicePtr);
```

To copy data between the memory spaces,host memory (CPU) and device memory (GPU) ***cudaMemcpy*** function is used. It provides a way to transfer data back and forth between the host and the device. The syntax for the function is as shown.

```
cudaMemcpy(destination, source, size, cudaMemcpyKind);
```

- *destination and source:* Pointers to the destination memory and source memory.
- *size:* Size in bytes of the data to be copied.
- *cudaMemcpyKind:* A flag that specifies the direction of the copy and the memory spaces involved. The possible values for cudaMemcpyKind are:
  - *cudaMemcpyHostToHost:* Copy data from host memory to host memory.
  - *cudaMemcpyHostToDevice:* Copy data from host memory to device memory.
  - *cudaMemcpyDeviceToHost:* Copy data from device memory to host memory.
  - *cudaMemcpyDeviceToDevice:* Copy data from device memory to device memory.

An example to summarize the Memory Management is given below.

```
int* hostArray, deviceArray;
int size = 100 * sizeof(int);

// Allocate host and device memory
cudaHostMalloc(&hostArray, size);
cudaMalloc(&deviceArray, size);
// Initialize hostArray with some data

// Copy hostArray to deviceArray
cudaMemcpy(deviceArray, hostArray, size, cudaMemcpyHostToDevice);
// Perform computations on deviceArray

// Copy deviceArray back to hostArray
cudaMemcpy(hostArray, deviceArray, size, cudaMemcpyDeviceToHost);

// Free allocated memory
cudaHostfree(hostArray);
cudaFree(deviceArray);
```

Kernel Definition and Invocation

Define the kernel function using the **__global__** qualifier. Specify the grid and block dimensions for thread organization. Invoke the kernel from the host code.

```
//Kernel Definition
__global__ void myKernel(int* input, int* output)
{
int tid = blockIdx.x * blockDim.x + threadIdx.x;
output[tid] = input[tid] * input[tid];
}

    //Kernel Invocation
int numBlocks = 256;
int blockSize = 256;
myKernel <<< numBlocks,blockSize >>>(deviceInput, deviceOutput);
```

On kernel invocation, multiple instances of kernel code are created and assigned to a particular thread in the grid. This assigning is done by making use of thread organization and thread indexing. CUDA uses a hierarchical organization of threads, blocks, and grids. Each thread executes the kernel code independently and is identified by its unique global index. Each thread has the following indexing parameters associated with it.

- **'threadIdx.x'** represents the thread index within a block or local index.
- **'blockIdx.x'** represents the block index within a grid.
- **'blockDim.x'** represents the block size.
- **'gridDim.x'** represents the grid size.

Based on the above parameters, the global thread index is calculated as shown below.

```
int tid = blockIdx.x * blockDim.x + threadIdx.x;
```

These indexing variables allow threads to access different data elements and perform computations in parallel.

Memory Transfer and Synchronization

- Transfer data between the CPU and GPU memory.
- Synchronize to ensure all GPU computations are completed before accessing results on the CPU.

```
//Memory Transfer
cudaMemcpy(hostOutput, deviceOutput, size, cudaMemcpyDeviceToHost);

//Device Synchronization
cudaDeviceSynchronize();
```

2.6.5 Unified Memory

In the latest versions of CUDA, Unified Memory model was introduced. Unified Memory in CUDA refers to a memory management model that provides a unified address space for both the CPU (host) and the GPU (device) in a CUDA-enabled system. It allows seamless access to memory by both sides without explicit data transfers, simplifying memory management and improving programming productivity.

The **‘cudaMallocManaged‘** API in CUDA is used for allocating managed memory that can be accessed by both the CPU and the GPU in a unified memory model. It was introduced in CUDA 6.0 as part of the Unified Memory feature. The **‘cudaMallocManaged‘** function allocates a region of managed memory and returns a pointer to it. The allocated memory can be accessed by both the CPU and the GPU without explicitly copying data between them. The CUDA runtime automatically manages the data migration between the CPU and the GPU based on the access patterns.

Here is the basic syntax of the **‘cudaMallocManaged‘** function:

```
cudaMallocManaged(void** devPtr, size_t size);
```

The ‘devPtr‘ parameter is a pointer to the pointer that will receive the allocated memory. ‘size‘ specifies the size of the memory to be allocated in bytes.

Here’s an example usage of ‘cudaMallocManaged‘:

```
#include <cuda_runtime.h>
    #include <iostream>
    __global__ void incrementData(int* data, int n)
    {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < n)
    {
    data[tid] += 1;
    }
    }
    int main()
    {
    int n = 100;
    int* data;
    cudaMallocManaged(&data, n * sizeof(int));

    // Access the allocated memory from CPU
    for (int i = 0; i < n; i++)
    {
    data[i] = i;
    }

    // Perform GPU computations using the allocated memory
    int blockSize = 256;
    int numBlocks = (n + blockSize - 1) / blockSize;
    incrementData<<<numBlocks, blockSize>>>(data, n);
    cudaDeviceSynchronize();

    // Access the modified memory from CPU
    for (int i = 0; i < n; i++)
    {
    printf("%d",data[i]);
    }
    cudaFree(data);
    return 0;
    }
```

In this example, we allocate managed memory using **cudaMallocManaged** for an array of n integers. Then, we access the allocated memory from the CPU and initialize it with values from 0 to n-1. After that, we define a GPU kernel **incrementData** that increments the values in the data array.

We launch the kernel with an appropriate number of blocks and threads and synchronize the device using **cudaDeviceSynchronize()** to ensure all GPU computations are completed.

Finally, we access the modified memory from the CPU and print the values to verify that the GPU computations have taken effect. We release the managed memory using **cudaFree** before exiting the program.

Managed memory provides the following features and benefits:

1. **Unified Memory Space:** Managed memory combines the CPU and GPU memory spaces into a single, unified memory space. This eliminates the need for explicit data transfers between the host and the device, as data is automatically migrated between them as needed.

2. **Automatic Data Migration:** The CUDA runtime automatically handles the migration of data between the CPU and the GPU. It determines when data needs to be moved based on access patterns, ensuring that frequently accessed data remains in the appropriate memory space to minimize data transfer overhead.

3. **Dynamic Memory Migration:** The CUDA runtime dynamically migrates data between the CPU and the GPU at runtime. It takes into account factors such as data access patterns, available GPU memory, and system-wide memory pressure. This dynamic migration optimizes memory usage and improves overall performance.

4. **Page Faulting:** Managed memory supports transparent page faulting, which means that memory pages are faulted into the GPU's memory space on-demand. This allows the GPU to access data directly from system memory without requiring explicit data transfers, reducing the need for manual data staging.

5. **Portable Code:** Managed memory simplifies code portability across different CUDA-capable devices. The unified memory model abstracts away the specifics of memory management, making it easier to write code that can run on various GPUs with different memory configurations.

One important thing to note is that the memory allocated with **'cudaMallocManaged'** has a unified memory space, meaning that it is subject to the limitations of the available GPU memory. Therefore, if the memory requirements exceed the available GPU memory, the performance may be impacted due to frequent data transfers between the CPU and the GPU.

2.7 Nvidia CUDA Compiler

The NVIDIA CUDA compiler, commonly referred to as **'nvcc'**, is a key component of the CUDA development toolkit. It translates CUDA source code, which contains both host (CPU) and device (GPU) code, into executable machine code that can be executed on NVIDIA GPUs.

'nvcc' is the NVIDIA CUDA Compiler, a crucial component of the CUDA development toolkit. It is a command-line compiler that translates CUDA source code into executable machine code that can run on NVIDIA GPUs. Here's an overview of how **'nvcc'** works:

1. **Source Code Parsing: 'nvcc'** parses the input CUDA source code files (typically with a '.cu' extension) and identifies sections containing host (CPU) code and device (GPU) code. Host code sections are typically written in standard C/C++, while device code sections use the CUDA C/C++ language extensions.

2. **Host Code Compilation:** The host code sections are handed over to the host C/C++ compiler (e.g., GCC, MSVC) for compilation. The host compiler treats the host code as regular C/C++ code and generates object files or host-side object code.

3. **Device Code Compilation:** The device code sections, written in CUDA C/C++ language, are processed by **'nvcc'** itself. It performs the following steps to generate GPU-specific code:

   (a) Preprocessing: **'nvcc'** preprocesses the device code sections, handling directives specific to CUDA, such as '#include <cuda_runtime.h>' and CUDA-specific macro expansions.

   (b) Device Code Translation: The CUDA device code is translated into an intermediate representation called PTX (Parallel Thread Execution). PTX is an architecture-independent assembly-like code representation that provides flexibility for optimization.

   (c) Optimization: **'nvcc'** applies various optimizations to the PTX code, including instruction reordering, loop unrolling, memory access optimizations, and more. The goal is to improve the performance and efficiency of the generated GPU code.

   (d) PTX to GPU-specific Code: The PTX code is further compiled into GPU-specific machine code, known as SASS (CUDA assembly language). SASS code is specific to a particular NVIDIA GPU architecture and provides low-level instructions that can be executed directly on the GPU.

4. **Linking:** Once the host and device code sections are compiled, ‘nvcc‘ performs the linking step to combine the host object code and the GPU-specific SASS code. This step produces the final executable or shared library that can be run on the target system.

‘nvcc‘ also provides additional features, such as the ability to specify GPU architecture targets, control code generation options, link against CUDA libraries, and more. It offers a range of compiler flags and options to fine-tune the compilation process and optimize performance.

It’s important to note that the **‘nvcc‘** compiler takes care of integrating host and device code seamlessly and handles the necessary steps to compile and optimize code for both the CPU and the GPU. This allows developers to write CUDA programs with a mix of host and device code, leveraging the parallel processing capabilities of NVIDIA GPUs.

2.8 Nvidia CUDA Profiler

‘nvprof‘ is a command-line profiling tool provided by NVIDIA that allows developers to profile CUDA applications and analyze their performance. It provides detailed information about GPU activities, kernel execution times, memory transfers, CUDA API calls, and more. Here’s an overview of how **‘nvprof‘** works:

1. Instrumentation: When you run a CUDA application with **‘nvprof‘**, it automatically instruments the application to collect profiling data. The instrumentation process inserts additional code into the application, which captures information about GPU activities during execution.

2. Application Execution: The instrumented CUDA application is executed, and **‘nvprof‘** collects profiling data as the application runs. It captures information about GPU activities, such as kernel launches, memory transfers, CUDA API calls, and synchronization events.

3. Profiling Data Collection: **‘nvprof‘** collects a wide range of profiling data during the execution of the instrumented CUDA application. This data includes GPU utilization, kernel execution times, memory bandwidth, memory access patterns, and more. **‘nvprof‘** also captures CUDA API call traces, which provide a detailed record of the CUDA function calls made by the application.

4. Profiling Output: After the instrumented CUDA application completes execution, **'nvprof'** generates a profiling report that summarizes the collected data. The report provides insights into the application's performance characteristics and helps identify areas for optimization.

5. Profiling Analysis: The **'nvprof'** tool offers various options and visualizations to analyze the profiling data. It provides a command-line interface with different commands and options to extract specific information from the profiling report. Additionally, **'nvprof'** can generate timeline views, kernel-level analysis, memory access patterns, and other visualizations to aid in performance analysis.

6. Optimization: Based on the profiling results, developers can identify performance bottlenecks and areas for improvement in their CUDA application. They can then make modifications to the code, memory access patterns, or GPU configurations to optimize performance.

'nvprof' is a powerful tool for profiling and analyzing the performance of CUDA applications. It provides detailed information about GPU activities and helps developers understand how their application utilizes GPU resources. By leveraging the insights gained from **'nvprof'**, developers can optimize their CUDA code, memory usage, and GPU configurations to improve the performance and efficiency of their applications.

The CUDA programming model provides a powerful and flexible framework for harnessing the computational power of GPUs. By leveraging parallelism and the CUDA programming model, various computationally intensive tasks can be accelerated and significant performance improvements can be achieved. More information regarding CUDA programming model can be found at [6]. [7] provides details of the various functions available in the CUDA Library.

Chapter 3

Field Programmable Gate Arrays

3.1 Introduction

Field-Programmable Gate Arrays (FPGAs) have emerged as powerful and versatile hardware platforms for implementing digital systems. With their reconfigurable nature, FPGAs offer a flexible solution that can be customized for a wide range of applications, from digital signal processing to embedded systems and high-performance computing. This chapter provides an in-depth exploration of FPGA technology, focusing on the architecture, programming model, and key design considerations, with a specific emphasis on the configurable logic blocks (CLBs) within an FPGA.

3.2 FPGA Architecture

An FPGA consists of a large array of configurable logic blocks (CLBs) interconnected by programmable routing resources. These components, along with other supporting structures, form the overall architecture of the FPGA. Understanding the CLBs and their functionality is crucial for comprehending the inner workings of an FPGA.

3.2.1 Configurable Logic Blocks (CLBs)

Configurable Logic Blocks (CLBs) are the fundamental building blocks of an FPGA. They are responsible for implementing the desired logic functions and data storage within the device. Each CLB typically consists of the following key components:

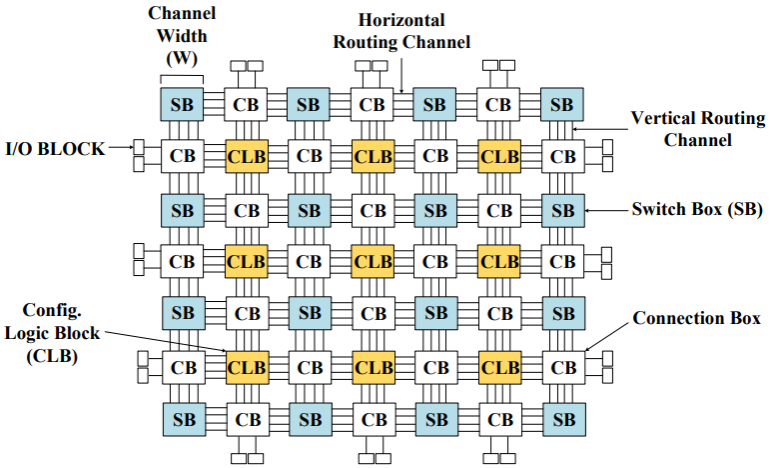


Figure 3.1: FPGA Architecture

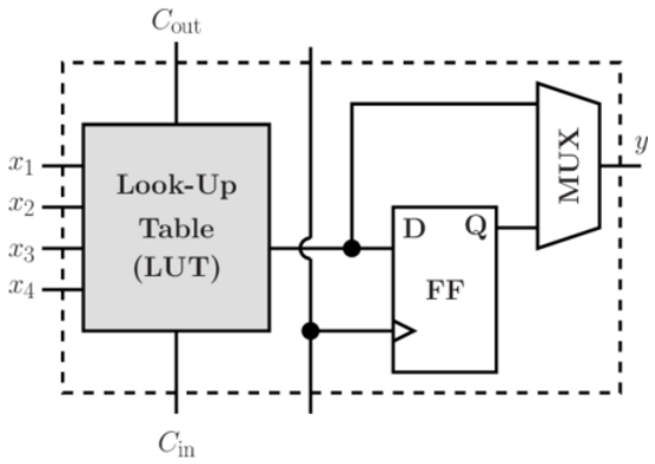


Figure 3.2: Configurable Logic Block

Lookup Table (LUT)

The Lookup Table (LUT) is a key component of the CLB and serves as a programmable logic element. It functions as a truth table, mapping input combinations to corresponding output values. LUTs are typically programmable and can implement any combinational logic function. They provide flexibility in creating custom digital circuits by enabling the implementation of various logic functions, from simple gates to complex functions like adders and multiplexers.

Flip-Flops or Latches

Flip-flops or latches within a CLB provide the ability to store and synchronize data. They act as memory elements and allow the retention of state information within the FPGA. Flip-flops or latches can be used to store intermediate results, implement sequential logic, or create memory elements within the digital circuit.

Multiplexer (MUX)

The multiplexer (MUX) within a CLB enables the selection of different inputs based on control signals. It allows for the routing and connection of signals from different sources to various elements within the CLB. The multiplexer provides flexibility in choosing the appropriate input signals for specific operations or conditions.

3.2.2 Interconnect Resources

Interconnect resources in an FPGA facilitate the routing and connection of signals between different CLBs and other components. These resources consist of a network of programmable routing switches, wires, and channels. They enable the establishment of desired data paths, ensuring proper communication between CLBs and forming the overall circuit functionality. The interconnect resources play a crucial role in determining the performance and flexibility of an FPGA design.

3.2.3 Input/Output Blocks (IOBs)

Input/Output Blocks (IOBs) provide the interface between the FPGA and external devices or systems. They are responsible for handling the input and output signals of the FPGA. IOBs typically consist of input buffers, output drivers, and programmable I/O standards. They allow the FPGA to communicate with other digital circuits or systems by providing configurable I/O interfaces.

3.2.4 Configuration Memory

Configuration memory stores the design information and determines the functionality of the FPGA. It typically comprises non-volatile memory, such as flash memory. The configuration memory stores the configuration bitstream, which is a binary file specifying the interconnections and functionality of

the FPGA. During the configuration process, the bitstream is loaded into the FPGA to configure the CLBs and establish the desired routing connections.

3.2.5 Clock Distribution Network

The clock distribution network in an FPGA ensures the proper distribution and synchronization of clock signals throughout the device. It consists of programmable clock routing resources that enable the precise control and distribution of clock signals to various components within the FPGA. The clock distribution network plays a critical role in achieving reliable and synchronized operation of the digital circuit implemented on the FPGA.

3.2.6 Embedded Memory

Many modern FPGAs also include embedded memory blocks, such as block RAM (BRAM) or distributed RAM. These embedded memory blocks provide additional storage capacity within the FPGA and can be used for various purposes, such as data storage, lookup tables, or FIFO buffers. The presence of embedded memory enhances the FPGA's ability to implement complex and data-intensive designs efficiently.

3.3 Programming Model for FPGA

The programming model for Field-Programmable Gate Arrays (FPGAs) involves several steps to configure the device and implement the desired digital circuit. Here is a detailed explanation of the FPGA programming model:

Hardware Description Languages (HDLs)

The FPGA programming process typically starts with the use of Hardware Description Languages (HDLs) such as VHDL (VHSIC Hardware Description Language) or Verilog. These languages allow designers to describe the behavior and structure of the digital circuit they want to implement. HDLs provide a high-level abstraction that captures the functionality of the circuit, including its inputs, outputs, internal components, and their interconnections.

Synthesis

Once the digital circuit is described in an HDL, the next step is synthesis. Synthesis is the process of converting the high-level HDL code into a gate-

level representation called a netlist. The synthesis tool analyzes the HDL code and generates a netlist that represents the logical structure of the circuit in terms of gates, flip-flops, and other standard digital components.

Place-and-Route

After synthesis, the netlist is fed into the place-and-route tool. The place-and-route tool determines the physical placement of the circuit components on the FPGA device and establishes the routing connections between them. It maps the logical design onto the actual resources available on the FPGA, such as CLBs, interconnects, and I/O blocks. The place-and-route tool aims to optimize various factors, including performance, power consumption, and resource utilization.

Bitstream Generation

Once the place-and-route process is completed, the next step is bitstream generation. The bitstream is a binary file that contains the configuration data necessary to program the FPGA. It specifies the interconnections, functionality, and settings of the FPGA device based on the synthesized and placed design. The bitstream generation tool takes the output of the place-and-route process and generates the bitstream file.

Configuration

The final step in the FPGA programming model is the configuration of the FPGA device. The bitstream file generated in the previous step is loaded onto the FPGA, either through a configuration cable or via a configuration interface on the target board. The bitstream configures the CLBs, interconnects, and other components of the FPGA based on the desired design. Once the configuration is complete, the FPGA is ready to execute the implemented digital circuit.

3.4 Design Considerations

Designing for Field-Programmable Gate Arrays (FPGAs) involves several considerations to ensure optimal performance, resource utilization, power consumption, and reliability. Here is a detailed explanation of the design considerations for FPGA-based designs:

Timing Constraints

Timing constraints play a critical role in FPGA designs. It involves specifying the desired clock frequency, setup and hold times, maximum propagation delays, and other timing requirements. Meeting these constraints ensures that the circuit operates reliably and within the desired performance specifications. Designers must carefully analyze and optimize the critical paths in the design to meet the timing requirements.

Resource Allocation

FPGAs have limited resources such as CLBs, memory blocks, and I/O pins. Efficiently allocating these resources is crucial to ensure optimal utilization and to avoid resource constraints. Careful consideration should be given to mapping the design onto the available resources, avoiding unnecessary duplication, and efficiently utilizing the available resources to achieve the desired functionality.

Power Consumption

Power consumption is an important consideration in FPGA designs, especially for portable or low-power applications. Designers should optimize power usage by employing power-efficient techniques such as clock gating, power gating, and using low-power components when possible. Power analysis tools can be utilized to estimate power consumption and guide power optimization efforts.

3.5 Advantages and Limitations of FPGAs

FPGAs offer several advantages over traditional application-specific integrated circuits (ASICs) and general-purpose processors (GPPs). These advantages include reconfigurability, high performance, parallelism, and flexibility. However, FPGAs also have some limitations, such as higher cost compared to ASICs, limited resources, and increased design complexity.

Chapter 4

Methodology

This chapter presents the methodology used to achieve the project objectives. It provides a clear picture on the various steps taken to achieve the results.

4.1 Problem Statement

The ANN based Molecular Dynamics mainly involves two major stages. First one involves taking the position of atoms in the atomic system and calculating the energy and force values experienced by each atom due to the remaining atoms in the atomic system. This calculation involves calculating the inter atomic descriptors of each atom and passing them as inputs to an ANN which was already designed to predict the energy values of each atom. After calculating the energies of all the atoms in the atomic system, the energy values are further processed to get the force values acting on each atom.

The second stage of the ANN based Molecular Dynamics involves calculating the new position of atoms in the next time step from the force values obtained in the first stage. This stage involves employing a numerical method to integrate the Newton's Law of Motions to obtain the new position of atoms from the respective force values. In our project Verlet Integration Algorithm is employed for calculating the new position of atoms in the atomic system.

In the first stage of ANN-MD, calculation of energy of atoms from inter atomic descriptors and thereby calculating the forces on atoms involves complex mathematical calculations and is very time intensive. Accelerating this stage of the ANN-MD will largely improve the performance of the ANN-MD computations.This is the main focus of the project.

In this project Cartesian coordinates (x,y,z) are used to represent the position of atoms in the atomic system. Calculating forces on each atom from the Cartesian coordinates and calculating new set of Cartesian coordinates of

atoms from the forces is described as one cycle of MD. Carrying out the above computations for large number of cycles gives an understanding of the atomic system and helps in structural analysis of the atomic system. In this project, the computations are carried out on two atomic systems, namely Au_{147} and Au_{309}.

4.2 Need For NN

If not for NN, the energy of the atoms in the nanocluster has to calculated using Density Function Theory (DFT). DFT involves very complex mathematical computations and takes a very large amount of time for calculations. With the advent of NNs, it is now possible to predict the energy of atom in a nanocluster from its local descriptors comparatively faster and very accurately [8]. For this we need to fix a suitable NN architecture and then train it with the DFT data. Keep changing the NN architecture until the RMS error between predicted energy values and energy values obtained from DFT is very minimum. Once the architecture is finalized, the NN is tested with test data to validate the predictions.

4.3 ANN Architecture

In general, NN consists of an input layer, hidden layers and an output layer with each layer consisting of a certain number of nodes. Each node is connected to all the other nodes in the next layer by means of branch. Each branch has a quantity called bias weight associated to it. The number of nodes in each layer and the number of hidden layers in the NN is specific to the application that we what to predict. In our project, NN is used to predict the energy of an atom by taking into account the inter atomic descriptors of that particular atom. In order to take into account, the relative arrangement of atoms in the atomic system along with the inter atomic distances of the atoms, two hidden layers are used as proposed in [9]. For the pre-trained NN that is used, energy of the nanocluster is fitted using Spherical harmonics based atomic environment descriptors [8].The number of nodes in the input layer and the hidden layers are finalised by comparing the RMS error of the predictions with the DFT results for different combinations of nodes for each layer. *In this project a pre trained NN is used for predicting energy values from atomic descriptors.* The NN has 59 input nodes, two hidden layers with 30 nodes each and an output layer with a single node. Of the 59 inputs, first 9 are the radial descriptor functions and the remaining 50 are the power spectrum of the atom.

Sigmoid function defined in 4.1 is used as the activation function for the NN. [10] provides a clear understanding on the implementation of neural network and the various mathematical functions involved in neural networks.

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{4.1}$$

4.4 Energy Calculation

As described in the above section, inter atomic descriptors are calculated for each atom from the Cartesian coordinates (x,y,z) of all atoms in the atomic system. These inter atomic descriptors are fed as inputs to an ANN to predict the energy of that particular atom. This process is continued to calculate the energy of all the atoms in the atomic system.

Let look at the steps involved in calculating inter atomic descriptors from the Cartesian coordinates.

Step 1: Calculating sperical coordinates (r,θ,ϕ) from Cartesian coordinates (x,y,z).

$$r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2} \tag{4.2}$$

$$\theta_{ij} = cos^{-1}\frac{z_i - z_j}{r_{ij}} \tag{4.3}$$

$$\phi_{ij} = tan^{-1}\frac{y_i - y_j}{x_i - x_j} \tag{4.4}$$

Step 2: Calculating Associated Legendre Polynomials $P_l^m(cos\theta)$ and Cut-off function $f_c(r_{ij})$.

$$f_c(r_{ij}) = \frac{1}{2}\left[\cos\left(\frac{\pi r_{ij}}{r_c}\right) + 1\right] \tag{4.5}$$

Step 3: Calculating the Radial Distribution function R_f^i using Cut-off function and different falling rate η.

$$R_f^i = \sum_{i \neq j}^{atoms} e^{-\eta r_{ij}^2} f_c(r_{ij}) \tag{4.6}$$

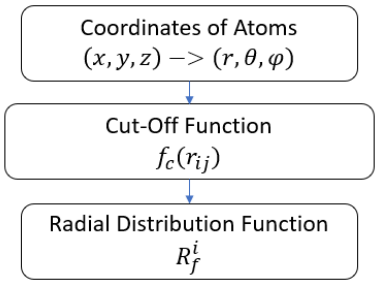


Figure 4.1: Radial Distribution Function Calculation

Step 4: Calculating Spherical Harmonics function $Y_l^m(\theta,\phi)$.

$$Y_l^m(\theta,\phi) = \begin{cases} N_{lm}P_l^m(cos\theta)e^{-im\phi} & when\ \ m<0 \\ (-1^m)N_{lm}P_l^m(cos\theta)e^{im\phi} & when\ \ m\geq 0 \end{cases} \tag{4.7}$$

$$N_{lm} = \sqrt{\frac{2l+1}{4\pi}\frac{(l-|m|)!}{(l+|m|)!}} \qquad -l \leq m \leq l \tag{4.8}$$

Step 5: Calculating Spherical Harmonics Coefficients c_{nlm} using Cut-off function and different Gaussian parameter ζ.

$$c_{nlm} = \sum_{i\neq j} e^{-\zeta r_{ij}^2} f_c(r_{ij}) Y_l^{m*}(\theta,\phi) \tag{4.9}$$

Step 6: Calculating Power Spectrum P_{nl}.

$$P_{nl} = \frac{4\pi}{2l+1}\sum_{m=-l}^{l} c_{nlm}^* c_{nlm} \tag{4.10}$$

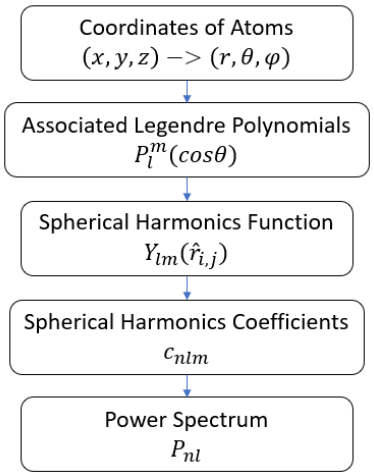


Figure 4.2: Power Spectrum Calculation

The Radial Distribution functions R_f^i and Power Spectrum P_{nl} are used as radial and angular descriptors of atoms. The number of radial and angular descriptors depends on the distinct values of falling rate η, Gaussian parameter ζ and frequency l which are determined while designing and fitting the NN using DFT data. For the NN being used, η takes 9 values giving rise to 9 distinct radial distribution functions and ζ takes 5 values, l takes 10 values giving rise to 50 distinct Power Spectrum. This results in a total of 59 descriptors for each atom in the nanocluster. These 59 descriptors are normalized and then fed to the ANN to predict the energy of the particular atom which is calculated as in 4.11. This process is followed to calculate the energy of remaining atoms in the nanocluster.

$$E_i = \sum_{a=1}^{N_{nh}} W_{a1}^{23}.f_a^2 \left[Wb_a^2 + \sum_{b=1}^{N_{nh}} W_{ba}^{12}.f_b^1 \left(Wb_b^1 + \sum_{s=1}^{N_{inp}} W_{sb}^{01}.D_{is} \right) \right] \tag{4.11}$$

where, E_i is energy of i^{th} atom and N_{nh} is number of hidden nodes. D_{is} is normalized input descriptor functions of length N_{inp} for i^{th} atom. W_{sb}^{01}, W_{ba}^{12} and W_{a1}^{23} are weights connecting from input layer to hidden layer one, hidden layer one to hidden layer 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. The total energy (E) of a nanoparticle is computed by summation all the atomic energies.

$$E = \sum_i E_i \tag{4.12}$$

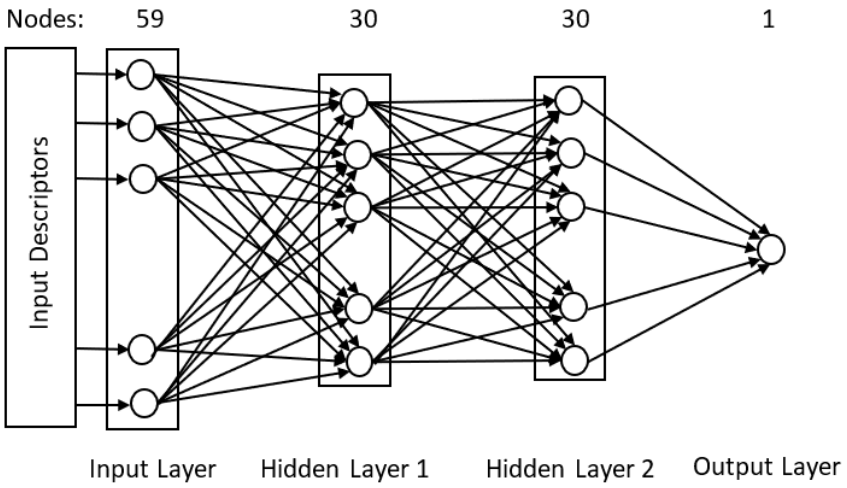


Figure 4.3: NN Architecture

4.5 Force Calculation

The force acting on each atom in the nanocluster depends on the energy of the atoms in the nanocluster and is defined as in 4.13 where $k \in (x,y,z)$.

$$F_k = -\frac{\partial E_{nanoparticle}}{\partial k} = -\sum_{i=1}^{atoms} \frac{\partial E_i}{\partial k} \tag{4.13}$$

$$= -\sum_{i=1}^{atoms} \sum_{s=1}^{inputs} \frac{\partial E_i}{\partial D_{is}} \frac{\partial D_{is}}{\partial k} \tag{4.14}$$

In 4.14, the first gradient term represents the gradient of energy with respect to the input descriptors. This is obtained from the NN in the following steps:

1. Calculate gradient of Sigmoid of each node value in HD1 and HD2 while calculating the energy.

2. The gradient of node values calculated are fed forward to the NN to predict the gradient of energy at the output node.

The second gradient term represents the gradient of input descriptors with respect to the coordinates x,y and z. Between both the partial derivative terms, calculating the gradient of descriptors with respect to x, y and z requires the spherical harmonics coefficients c_{nlm} calculated during the energy calculations. Due to this, the force calculations require all the energy calculations to be completed. Moreover, calculating the gradient of Power Spectrum is a tedious task as it involves calculating the partial derivative of multiple components as shown in 4.15.

$$\frac{dP^i_{nl}}{dx_i} = \sum_{m=-l}^{l} \left(\frac{dc^i_{nlm}}{dx_i} c^{i*}_{nlm} + c^i_{nlm} \frac{dc^{i*}_{nlm}}{dx_i} \right) \tag{4.15}$$

$$\frac{\mathrm{d}c^i_{nlm}}{\mathrm{d}x_i} = \sum_{i \neq j} \left(\frac{\mathrm{d}(e^{-\eta r^2_{ij}})}{\mathrm{d}x_i} Y^m_l(r_{ij}) f_c(r_{ij}) + \frac{\mathrm{d}Y^m_l(r_{ij})}{\mathrm{d}x_i} e^{-\eta r^2_{ij}} f_c(r_{ij}) + \frac{\mathrm{d}(f_c(r_{ij}))}{\mathrm{d}x_i} Y^m_l(r_{ij}) e^{-\eta r^2_{ij}} \right) \tag{4.16}$$

Note: If coefficient of an atom i is differentiated with respect to coordinate of atom j then, sum over all atoms vanish

$$\frac{\mathrm{d}c^i_{nlm}}{\mathrm{d}x_j} = \left(\frac{\mathrm{d}(e^{-\eta r^2_{ij}})}{\mathrm{d}x_j} Y^m_l(r_{ij}) f_c(r_{ij}) + \frac{\mathrm{d}Y^m_l(r_{ij})}{\mathrm{d}x_j} e^{-\eta r^2_{ij}} f_c(r_{ij}) + \frac{\mathrm{d}(f_c(r_{ij}))}{\mathrm{d}x_j} Y^m_l(r_{ij}) e^{-\eta r^2_{ij}} \right) \tag{4.17}$$

Solving the derivatives in 4.16 results in the following equations.

$$\frac{\mathrm{d}(e^{-\eta r^2_{ij}})}{\mathrm{d}x_i} = e^{-\eta r^2_{ij}} (-\eta 2 r_{ij}) \left(\frac{2x_{ij}}{2r_{ij}} \right) = -2\eta e^{-\eta r^2_{ij}} x_{ij} \tag{4.18}$$

$$\frac{\mathrm{d}(f_c(r_{ij}))}{\mathrm{d}x_i} = -0.5 \left(\frac{\pi x_{ij}}{r_c r_{ij}} \right) sin \frac{\pi r_{ij}}{r_c} \tag{4.19}$$

$$\frac{\mathrm{d}Y^m_l(r_{ij})}{\mathrm{d}x_i} = \frac{\mathrm{d}Y^m_l}{\mathrm{d}\theta(r_{ij})} \frac{\mathrm{d}\theta(r_{ij})}{\mathrm{d}x_i} + \frac{\mathrm{d}Y^m_l}{\mathrm{d}\phi(r_{ij})} \frac{\mathrm{d}\phi(r_{ij})}{\mathrm{d}x_i} \tag{4.20}$$

The above equation is solved using the $\mathrm{Y}^m_l(\theta, \phi)$ definition in 4.7.

After calculating the gradient of energy and gradient of descriptors using the above equations they are multiplied and summation is applied over the entire inputs and the entire atoms as in 4.14 to give the force experinced by a particular atom in the nanocluster. This processed is followed for all the atoms in the nanocluster to get the force on each atom.

4.6 Verlet Algorithm

The Verlet algorithm is a numerical integration method commonly used for simulating the motion of particles. It is particularly useful in molecular dynamics simulations.

The equations for the Verlet algorithm are as follows:

$$\mathbf{r}(t+\Delta t) = 2\mathbf{r}(t) - \mathbf{r}(t-\Delta t) + \mathbf{a}(t)\Delta t^2 \tag{4.21}$$

$$\mathbf{v}(t) = \frac{\mathbf{r}(t+\Delta t) - \mathbf{r}(t-\Delta t)}{2\Delta t} \tag{4.22}$$

In Equation 4.21, $\mathbf{r}(t)$ represents the position of the particle at time t, $\mathbf{a}(t)$ is the acceleration at time t, and Δt is the time step. This equation updates the position of the particle based on the previous two positions and the acceleration.

Equation 4.22 calculates the velocity of the particle at time t based on the updated positions at $t+\Delta t$ and $t-\Delta t$. It uses the central difference approximation to estimate the velocity.

These equations can be iteratively applied to simulate the motion of particles over time using the Verlet algorithm.

The Taylor series expansion of the Verlet algorithm equations can be written as follows:

$$\mathbf{r}(t+\Delta t) = \mathbf{r}(t) + \mathbf{v}(t)\Delta t + \frac{\mathbf{a}(t)\Delta t^2}{2} + \frac{\mathbf{j}(t)\Delta t^3}{6} + \mathscr{O}(\Delta t^4) \tag{4.23}$$

$$\mathbf{v}(t) = \mathbf{v}(t-\Delta t) + \frac{\mathbf{a}(t)\Delta t}{2} + \frac{\mathbf{j}(t)\Delta t^2}{6} + \mathscr{O}(\Delta t^3) \tag{4.24}$$

In Equation 4.23, $\mathbf{r}(t)$ represents the position of the particle at time t, $\mathbf{v}(t)$ is the velocity at time t, $\mathbf{a}(t)$ is the acceleration at time t, and $\mathbf{j}(t)$ is the jerk (rate of change of acceleration) at time t. The terms involving higher-order derivatives represent the additional terms in the Taylor series expansion.

Equation 4.24 shows the Taylor series expansion of the velocity equation, incorporating the acceleration and jerk terms.

The force acting on a particle can often be expressed in terms of the gradient of a scalar potential function. Mathematically, it can be written as:

$$F = -\nabla V \tag{4.25}$$

where F represents the force vector acting on the particle.

∇ (del) is the gradient operator, which operates on the scalar potential function V.

V is the scalar potential function, which depends on the position coordinates.

According to Newton's laws of motion, F is related to mass (m) of the particle and its acceleration $\mathbf{a}(t)$ as in 4.26

$$F = m.\mathbf{a}(t) \tag{4.26}$$

From 4.25 and 4.26, $\mathbf{a}(t)$ can be written in terms of gradient of scalar potential function V from 4.27

$$\mathbf{a}(t) = \frac{-\nabla V}{m} \tag{4.27}$$

In our project, Δt is very small, in order of 0.03 pico second. In eqautions 4.23 and 4.24, the higher-degree terms of Δt can be ignored and the equation is approximated as in 4.29 and 4.28.

$$\mathbf{v}(t) = \mathbf{v}(t - \Delta t) + \frac{\mathbf{a}(t)\Delta t}{2} \tag{4.28}$$

$$\mathbf{r}(t + \Delta t) = \mathbf{r}(t) + \mathbf{v}(t)\Delta t + \frac{\mathbf{a}(t)\Delta t^2}{2} \tag{4.29}$$

The above equations 4.27,4.28 and 4.29 are used to calculated the new set of positions of atoms in the nanocluster. The ∇V is the force obtained from the force calculations discussed in the previous section.

4.7 Algorithms

Associated Legendre Polynomial Algorithm

Associated Legendre polynomials $P_l^m(x)$ are used in Spherical Harmonics calculations where x=cos(θ), l and m are the degree and order of the polynomial respectively. Associated Legendre polynomials $P_l^m(x)$ can be calculated using the following recursive equations.

$$P_{l+1}^{l+1}(x) = -(2l+1)\sqrt{1-x^2}P_l^l(x) \tag{4.30}$$

$$P_l^{m-1}(x) = \frac{1}{(l+m)(l-m+1)}\left(\frac{-2mxP_l^m(x)}{\sqrt{1-x^2}} - P_l^{m+1}(x)\right) \tag{4.31}$$

In 4.31, P_l^{m+1} (x) becomes zero for m>(l-1) and it reduces to 4.32.

$$P_l^{m-1}(x) = \frac{1}{(l+m)(l-m+1)} \frac{-2mxP_l^m(x)}{\sqrt{1-x^2}} \tag{4.32}$$

In our project, the range of l is $0 \leq l \leq 9$ and the range of m is $-l \leq m \leq l$.

Algorithm 1 $P_l^m \cos(\theta)$ Calculation

```
Initialize variables
for each pair of atoms do
  Calculate (r,θ,φ) from (x,y,z)
  for each l ∈ [0,9] do
    Calculate P_l^l cos(θ) as in 4.31
    for each m ∈ [-l,l-1] from l-1 to -l do
      Calculate P_l^m cos(θ) as in 4.32
    end for
  end for
end for
```

Radial Distribution Function Algorithm

The Radial Distribution function R_f^i is used to calculate the distribution of atoms in the nanocluster. R_f^i for an atom i is given as sum of Gaussian functions with different falling rate η with respect to r_{ij} , the inter-atomic distance between atom i and j. The interaction of an atom with other atoms decreases as we go farther away from the atom in consideration. So a cut-off function $f_c(r_{ij})$ with a cut-off radius of 8 Å is chosen and atoms within the cut-off radius *(r_cut)* are only considered for R_f^i calculation. In our project, 9 η values are used for R_f^i calculation.

Algorithm 2 R_f^i Calculation

```
Initialize variables, Sum = 0
for each of 9 η values do
  for each pair of atoms do
    Calculate (r,θ,φ) from (x,y,z)
    if r≤ r_cut then
      Calculate f_c(r_ij) as in ??
    else
      f_c(r_ij) = 0
    end if
    Calculate e^{-η r_ij^2} f_c(r_ij)
    Sum += e^{-η r_ij^2} f_c(r_ij)
  end for
  R_f^i = Sum
end for
```

Power Spectrum Algorithm

The Power Spectrum (P_{nl}) calculations requires the calculation of Spherical Harmonics function (Y_l^m (r_{ij})) and Spherical Harmonics coefficients (c_{nlm}) both of which are complex quantities. The complex term in both the equations, $e^{-im\phi}$ is calculated as sum of real and imaginary parts as shown in 4.33.

$$e^{-im\phi} = cos(-im\phi) + i.sin(-im\phi) \tag{4.33}$$

In our program, the complex term is calculated as separate real term and imaginary term. This type of representation makes it easy while implemented the program on GPU as the use of complex data types is not straight-forward and requires additional libraries.

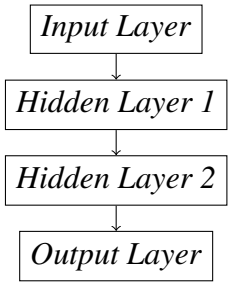
Algorithm 3 P_{nl} Calculation

```
Initialize variables, Sum1 = 0, Sum2 = 0
for each pair of atoms do
  Calculate P_l^m cos(θ)
end for
for each of 5 ζ values do
  for each l ∈ [0,9] do
    for each m ∈ [-l,l] from -l to l do
      for each pair of atoms do
        Calculate (r,θ,φ) from (x,y,z)
        if r≤ r_cut then
          Calculate f_c(r_ij) as in ??
        else
          f_c(r_ij) = 0
        end if
        Calculate e^{-ζ r_ij^2}.f_c(r_ij)
        Calculate Y_lm(r_ij) as in 4.7
        Sum1 += e^{-ζ r_ij^2}.f_c(r_ij).Y*_lm(r_ij)
      end for
      c_nlm = Sum1
      Sum2 += c*_nlm.c_nlm
    end for
  P_nl = [4π/(2l+1)].Sum2
  end for
end for
```

Energy Algorithm

As discussed earlier, the energy values are predicted using ANN by feeding inter-atomic descriptors as inputs to the ANN. The ANN used in our project has an input layer, two hidden layers and one output layer with 59 nodes, 30 nodes, 30 nodes and 1 node respectively. The nodes in a layer are connected to every nodes in the next layer via weights. Each node in both the hidden layers has a bias component linked to it.

Consider $Input_i$, $HD1_j$, $HD2_k$ and *Output* be the nodes in each layer of the ANN. $WeightIHD1_{ij}$, $WeightHD1HD2_{jk}$ and $WeightHD2O_k$ be the weights between the respective layers. The respective nodes in each layer are calculated as shown in figure 4.5 where *Sigmoid* is the activation function.



$$HD1_j = \text{Sigmoid}\left(\sum_i Input_i \cdot WeightIHD1_{ij} + BiasHD1_i\right)$$
$$HD2_k = \text{Sigmoid}\left(\sum_j HD1_j \cdot WeightHD1HD2_{jk} + BiasHD2_j\right)$$
$$Output = \text{Sigmoid}\left(\sum_k HD2_k \cdot WeightHD2O_k\right)$$
$$0 \leq i \leq 59,\ \ 0 \leq j \leq 30,\ \ 0 \leq k \leq 30$$

Figure 4.4: Energy Prediction using ANN

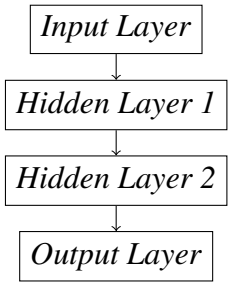
Algorithm 4 Energy Calculation

```
Initialize variables
Calculate R_f^i
Calculate P_nl
for each node in Hidden Layer 1 do
    Calculate HD1_j
end for
for each node in Hidden Layer 2 do
    Calculate HD2_k
end for
Calculate Output
```

The *Output* is the predicted energy of the atom for the set of input descriptors by the ANN.

Force Algorithm

As discussed in the force calculation section, the force values are calculated as the product of gradient of energy with respect to input descriptors and the gradient of input descriptors with respect to x,y and z as shown in 4.14.



$$HD1_j = \text{Sigmoid}\left(\sum_i Input_i \cdot WeightIHD1_{ij} + BiasHD1_i\right)$$
$$grad\_HD1_j = HD1_j * (1 - HD1_j)$$
$$HD2_k = \text{Sigmoid}\left(\sum_j HD1_j \cdot WeightHD1HD2_{jk} + BiasHD2_j\right)$$
$$grad\_HD2_k = HD2_k * (1 - HD2_k)$$

Figure 4.5: Gradient of Energy Prediction using ANN

Algorithm 5 Force Calculation

Initialize variables
Calculate Gradient of R_f^i
Calculate Gradient of P_{nl}
Calculate $grad\_HD1_j$ and $grad\_HD2_k$
for each node in Hidden layer 1 and Hidden layer 2 **do**
  Fed Forward $grad\_HD1_j$ and $grad\_HD2_k$
  Calculate Gradient of Energy
end for
Calculate Force by multiplying gradient of energy and gradient of descriptors.

4.8 CPU Implementation

The algorithms discussed above are converted into C code for calculating the energy and force on atoms in Au_{147} and Au_{309} clusters. In our project, previously available code for the above calculations is optimized and modified so that the code can be easily parallelized to be implemented on the GPU later.

The above mentioned C code is tested to check the correctness of the results obtained. On successful testing of energy and forces values, this C code is integrated with a FORTRAN code which implements the Verlet algorithm to generate new set of coordinates for the atoms in Au_{147} (Au_{309}) cluster using the old coordinates and the force values as discussed in 4.6.

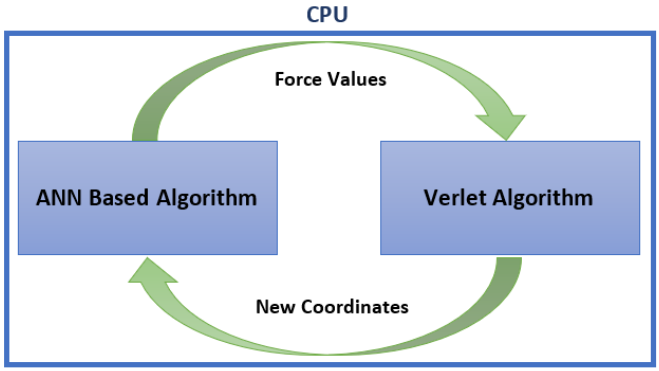


Figure 4.6: CPU Flow

4.9 GPU Implementation

Implementing the ANN based energy and force calculations on GPU by offloading these calculations from CPU to GPU using CUDA programming model provides better throughput because of the highly parallel architecture of the GPU. To implement the energy and force calculations on the GPU, the C code implemented on the CPU is parallelized and written as kernels to be executed on the GPU. In kernel, thread indexing is used to assign a task to a particular thread. The execution of the algorithm begins at CPU and the tasks to be executed in parallel are offloaded onto the GPU. The CPU takes care of the other tasks like memory management, kernel call, threads synchronization etc. In general, the CPU and GPU are connected onboard via PCIe, through which high speed data transfers happen.

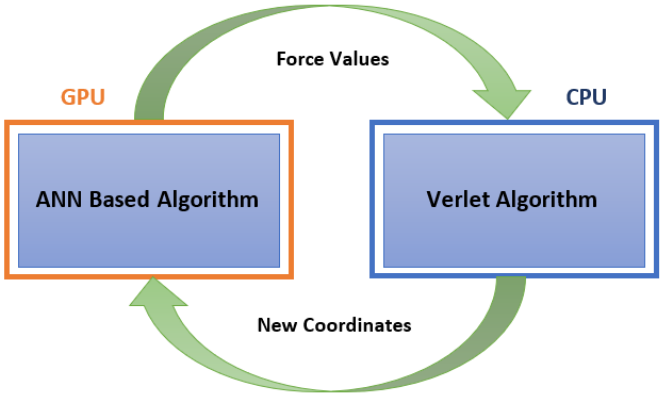


Figure 4.7: GPU Flow

Kernels

As discussed in the force calculations section, to calculate the force on atoms we require the Spherical Harmonics coefficients c_{nlm} of all the atoms in the nanocluster. Due to this it is not possible to include both energy and force calculations into a single kernel. So the following approach is employed:

1. Two Kernels are used in the CUDA C program.
2. One Kernel calculates the energy of each atom in the nanocluster using the coordinates.
3. Another Kernel calculates the force on each atom in the nanocluster using the c_{nlm} data and gradient of energy data calculated by the energy kernel.

4.9.1 Energy Kernel

Number of Au Atoms in the Nanocluster : 147 (309).

- Kernel Input Pointers:
  - Coordinates Pointer - size: 441 (927) float values
  - N_{lm} Pointer - size: 100 float values
  - Normalization Pointers - size: 59 values each
    - *meangtot_f*
    - *gtotmin_f*
    - *gtotmax_f*
  - Neural Network Weight Pointers
    - *a01* - Input Layer to HD1 - size: 1800 float values
    - *a12* - HD1 to HD2 - size: 930 float values
    - *a23* - HD2 to Output Layer - size: 30 float values
- Kernel Output Pointers:
  - Energy Pointer - size: 147 (309) float values
  - Gradient of Energy Pointer - size: 8673 (18,231) float values
  - c_{nlm} Pointer - size: 147000 (309000) float values

4.9.2 Force Kernel

Number of Au Atoms in the Nanocluster : 147 (309).

- Kernel Input Pointers:
  - Coordinates Pointer - size: 441 (927) float values
  - N_{lm} Pointer - size: 100 float values
  - Normalization Pointers - size: 59 values each
    - *meangtot_f*
    - *gtotmin_f*
    - *gtotmax_f*
  - Gradient of Energy Pointer - size: 8673 (18,231) float values
  - c_{nlm} Pointer - size: 147000 (309000) float values
- Kernel Output Pointers:
  - Force Pointer - size: 441 (927) float values

```
__global__ energy(float *inputs, float *outputs)//Energy Kernel
{
// Declare and Initialize variables
...
//Thread Indexing
int h=blockIdx.x*blockDim.x+threadIdx.x;

// N = 147 (309)
if(h<N)
{
// P_l^m(cosθ) calculation
...
// R_f^i calculation
...
// c_nlm,P_nl calculation
...
// Energy and it's gradient calculation
...
} }
```

```
__global__ force(float *inputs, float *outputs) //Force Kernel
{
// Declare and Initialize variables
...
//Thread Indexing
int h=blockIdx.x*blockDim.x+threadIdx.x;

// N = 147 (309)
if(h<N)
{
// P_l^m(cosθ) calculation
...
// Gradient of R_f^i calculation
...
// Gradient of P_nl calculation using c_nlm calculated by energy kernel
...
// Force calculation using gradients of R_f^i, P_nl and energy
...
} }
```

4.9.3 Memory Management

In our project, Unified Memory model is used to allocate and deallocate managed memory for Input/Output pointers. The below code shows the memory allocation and deallocation functions for all the variables used in this project.

```
// Variables Declaration
float* xyz;
float* meangtot, * gtotmin, * gtotmax;
float* nlm, * a01, * a12, * a23;
float* ene, * grad_e;
float* t_cnlm, * force;

// Unified Memory Allocation
cudaMallocManaged(&xyz,size_xyz*sizeof(float));
cudaMallocManaged(&nlm,size_nlm*sizeof(float));
cudaMallocManaged(&meangtot,size_meangtot*sizeof(float));
cudaMallocManaged(&gtotmin,size_gtotmin*sizeof(float));
cudaMallocManaged(&gtotmax,size_gtotmax*sizeof(float));
cudaMallocManaged(&a01,size_a01*sizeof(float));
cudaMallocManaged(&a12,size_a12*sizeof(float));
cudaMallocManaged(&a23,size_a23*sizeof(float));
cudaMallocManaged(&ene,size_ene*sizeof(float));
cudaMallocManaged(&grad_e,size_grad_e*sizeof(float));
cudaMallocManaged(&t_cnlm,size_t_cnlm*sizeof(float));
cudaMallocManaged(&force,size_force*sizeof(float));

//Unified Memory Deallocation
cudaFree(xyz);
cudaFree(nlm);
cudaFree(meangtot);
cudaFree(gtotmin);
cudaFree(gtotmax);
cudaFree(a01);
cudaFree(a12);
cudaFree(a23);
cudaFree(ene);
cudaFree(grad_e);
cudaFree(t_cnlm);
cudaFree(force);
```

The size of each variable used in the CUDA program is listed below. *Number of Au Atoms in the Nanocluster : 147 (309).*

- Size of Input/Output Pointers
  - size_xyz: 441 (927)
  - size_nlm: 100
  - size_meangtot: 59
  - size_gtotmin: 59
  - size_gtotmax: 59
  - size_a01: 1800
  - size_a12: 930
  - size_a23: 30
  - size_ene: 147 (309)
  - size_grad_E: 8673 (18,231)
  - size_t_cnlm: 147000 (309000)
  - size_force: 441 (927)

4.9.4 Kernel Configuration

In our project, each kernel is programmed in such a way that each instance of the kernel calculates energy or force of a single atom in the nanocluster. So the number of instances of kernel required depends on the number of atoms in the nanocluster.Calculations are carried out for two structures of gold nanocluster, namely Au_{147} and Au_{309} each with 147 and 309 atoms respectively. So the kernel configuration ,i.e., the number of Blocks and the number of Threads per Block are chosen in such a way that required number of threads are available with better resource management for optimized performance of the Kernel.

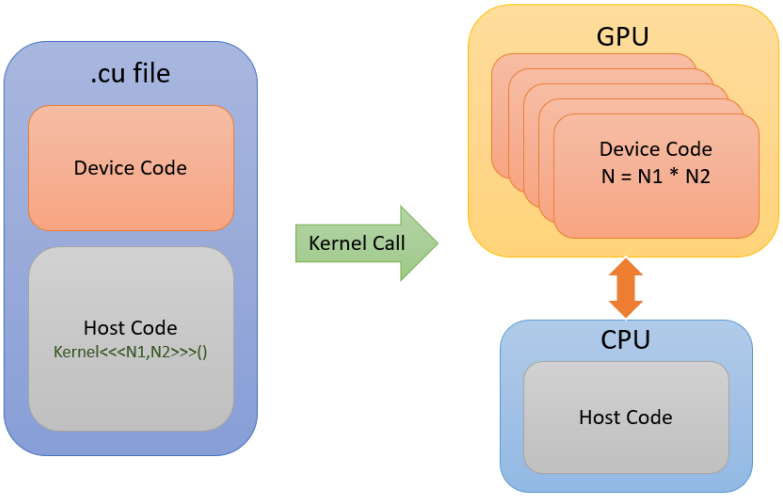


Figure 4.8: Kernel Call

```
// Variables for Kernel Configuration
int n_blocks;
int n_threadsperblock;

// Kernel Call and Synchronization

    // Energy Kernel Call
energy <<< n_blocks,n_threadsperblock >>> (inputs, outputs);
cudaDeviceSynchronize();

// Force Kernel Call
force <<< n_blocks,n_threadsperblock >>> (inputs, outputs);
cudaDeviceSynchronize();
```

4.9.5 Code Compilation

The entire host code and device code are packaged in .cu CUDA file. The host C/C++ compiler cannot compile the entire code in the .cu file. The host C/C++ compiler compiles host code and the device code is compiled by the **'nvcc'** compiler. Steps involved in code compilation by **'nvcc'** Compiler is discussed in detail in 2.7. The **'nvcc'** compiler converts the device code into fat-binary images which are added to object file produced after compiling host code by the host C/C++ compiler. This is packaged into a single object file

that runs on the host (CPU). While executing, whenever the host encounters the fat-binary images the execution is offloaded onto the device (GPU) and the remaining code is executed on the host.

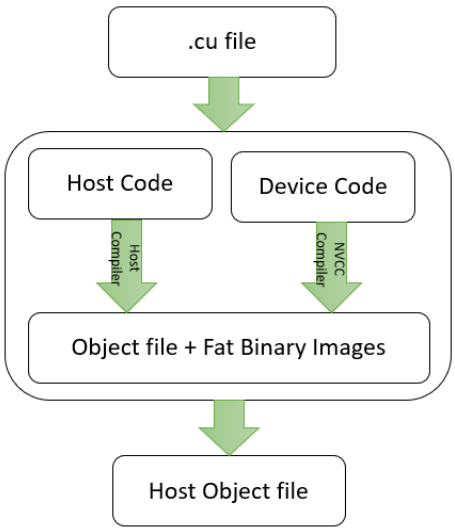


Figure 4.9: CUDA Code Compilation

4.9.6 Code Profiling

Code profiling helps in analysing the code and the performance of various APIs. While .cu file is compiled and executed, a number of CUDA run time APIs are invoked automatically during the code run-time. Profiling the code using **'nvprof'** CUDA profiling tool provides an analysis of all the APIs invoked during the code run-time. The output of **'nvprof'** tool for our force and energy calculation is as shown below. The steps involved in profiling the code using **'nvprof'** tool is discussed in 2.8.

The code for compiling a .cu file using **'nvcc'** compiler and profiling the executable file using **'nvprof'** profiler tool is as shown below.

```
nvcc force.cu -o force
nvprof ./force
```

By running the above commands, the **'nvcc'** tool compiles the *force.cu* file and outputs an executable file *force*. this executable file *force* is profiled by the **'nvprof'** profiler reulting in the following profiling information.

==1601== Profiling application: ./force

==1601== Profiling result:

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	68.76%	312.85ms	1	312.85ms	312.85ms	312.85ms	forcee(float* variables)
	31.24%	142.11ms	1	142.11ms	142.11ms	142.11ms	energy(float* variables)
API calls:	64.85%	454.97ms	2	227.49ms	142.11ms	312.86ms	cudaDeviceSynchronize
	33.99%	238.50ms	12	19.875ms	3.6360us	238.44ms	cudaMallocManaged
	1.11%	7.7930ms	2	3.8965ms	33.971us	7.7590ms	cudaLaunchKernel
	0.02%	161.21us	12	17.912us	9.5790us	64.722us	cudaFree
	0.02%	116.07us	101	1.1490us	136ns	48.313us	cuDeviceGetAttribute
	0.00%	23.940us	1	23.940us	23.940us	23.940us	cuDeviceGetName
	0.00%	5.2640us	1	5.2640us	5.2640us	5.2640us	cuDeviceGetPCIBusId
	0.00%	1.9960us	3	665ns	215ns	1.4330us	cuDeviceGetCount
	0.00%	952ns	2	476ns	239ns	713ns	cuDeviceGet
	0.00%	456ns	1	456ns	456ns	456ns	cuModuleGetLoadingMode
	0.00%	392ns	1	392ns	392ns	392ns	cuDeviceTotalMem
	0.00%	238ns	1	238ns	238ns	238ns	cuDeviceGetUuid

==1601== Unified Memory profiling result:

Device "Tesla T4 (0)"

Count	Avg Size	Min Size	Max Size	Total Size	Total Time	Name
3	21.333KB	4.0000KB	56.000KB	64.00000KB	12.67200us	Host To Device
15	68.267KB	4.0000KB	384.00KB	1.000000MB	99.07200us	Device To Host
7	-	-	-	-	738.8680us	Gpu page fault groups

Total CPU Page faults: 7

4.10 FPGA Implementation

Performing ANN based energy and force calculations on the FPGA involves the following steps:

1. Transfer coordinate values of atoms from host (CPU) to the FPGA evaluation board and store the coordinates data in on-board DDR Memory.
2. Energy and force values on atoms are calculated using the coordinate values of atoms and store back in the on-board DDR Memory.
3. Transfer the energy and force values from FPGA evaluation board to the host (CPU).

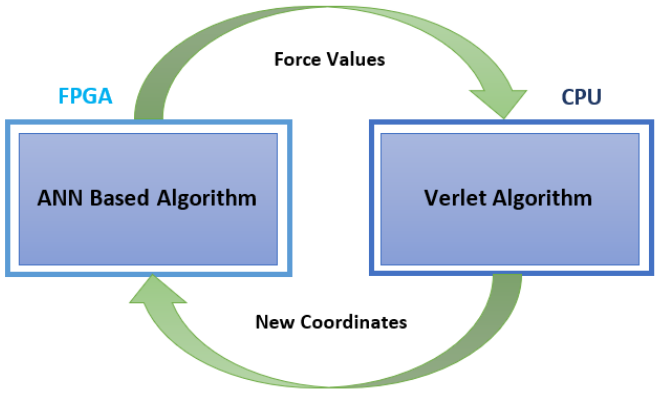


Figure 4.10: FPGA Flow

The force values received from the FPGA evaluation board are processed by the host (CPU) using verlet algorithm to generate new coordinates of atoms. The new coordinates are again transferred to the FPGA board to calculate new energy and force values on atoms. This process is carried out for large number of cycles to study the MD of nanocluster. Various communication methods like UART, Ethernet and PCIe can be used to transfer data between FPGA board and the host. UART communication protocol is used in our project.

Implementation of ANN based energy and force calculations on the FPGA involves the following steps:

1. Package the ANN based energy and force calculations into a custom IP for the particular FPGA family.

2. Design a system containing microblaze IP, the custom IP, UART IP and necessary IP blocks required for FPGA implementation.

3. FPGA is programmed using SDK to control the system for carrying out the necessary tasks.

4.10.1 Custom IP Design

Implementation of ANN based energy and force calculations on the FPGA is carried out by initially packaging the ANN based energy and force calculations into a custom IP for the particular FPGA family. Vivado HLS tool is used for this purpose. Vivado HLS (High-Level Synthesis) is a high-level synthesis tool provided by Xilinx. It allows designers to create hardware designs using high-level programming languages such as C, C++, and SystemC, instead of traditional hardware description languages (HDLs) like VHDL or Verilog. Packaging the necessary logic into an IP using Vivado HLS tool involves the following steps:

1. Use Vivado HLS directives to guide the synthesis process and optimize the generated hardware. Directives include specifying data types, array partitioning, loop *unrolling*, *pipelining*, and *interface* pragmas.

2. Launch the C synthesis process in Vivado HLS. This process converts the C code into RTL (Register Transfer Level) hardware description.

3. Analyze the synthesis report to evaluate the performance, resource utilization, and other metrics of the generated hardware. Identify opportunities for optimization and refine the design directives accordingly.

4. Iterate the process of modifying design directives, running synthesis, and analyzing the results to achieve the desired performance and resource utilization.

5. Once satisfied with the synthesized design, export it as hardware IP. This generates IP files, including the RTL description, testbench, and associated files required for integration into a larger FPGA design.

The necessary methods like pipelining, array partitioning etc. are used to reduce the latency of the IP. Optimizing the latency of the IP depends on the FPGA resources and is particular to the FPGA used. In our project, **Genesys2** FPGA board with **Kintex** FPGA with part number *xc7k325tffg900-2*.

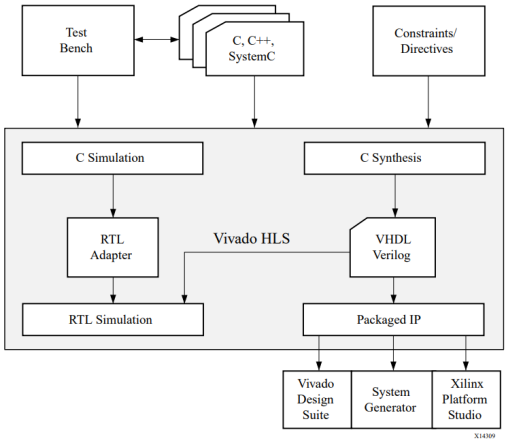


Figure 4.11: Vivado HLS Design Flow

In our IP design, two interface pointers *input* and *output* are used to communicate with other IPs. These interfaces are used to exchange data and control signals. The *input* and *output* interfaces are configured to be memory-mapped AXI4 (*m_axi*) interfaces which can be configured as *direct* or *slave* for assigning the memory offset for the *m_axi* interface. With *direct* the offset is set during the system design and with *slave* the offset can be set while programming using SDK. In our design,*m_axi* interfaces are configured as *slave*. The C code snippet for implementing the energy and force calculations as an IP using HLS directives is shown below.

```
int force_kishore_pipeline(float *input, float *output)
{
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE m_axi depth=in_depth port=output offset=slave
#pragma HLS INTERFACE m_axi depth=out_depth port=input offset=slave
//Copy Coordinates from DDR Memory
memcpy(xyz,(float *)(input),input_size*sizeof(float));

//Energy and Force Calculations

//Copy energy and force values into DDR Memory
memcpy((float *)(output),Force,output_size*sizeof(float));
return 0;
}
```

The *in_depth*, *out_depth*, *input_size* and *output_size* depends on the number of atoms in the nanocluster and are different for Au_{147} and Au_{309}.

The status signals specifying the state of the IP are bundled into a return port. In our IP design, The return port is configured as AXI4-Lite slave (*s_axilite*) interface. When an *s_axilite* interface is implemented in Vivado HLS, a set of C driver files are automatically created. These C driver files provide a set of APIs that can be integrated into any software running on a CPU and used to communicate with the device via the *s_axilite* interface. All the AXI peripheral ports follow the RTL timing as shown below. Detailed information on Vivado HLS can be found at [11].

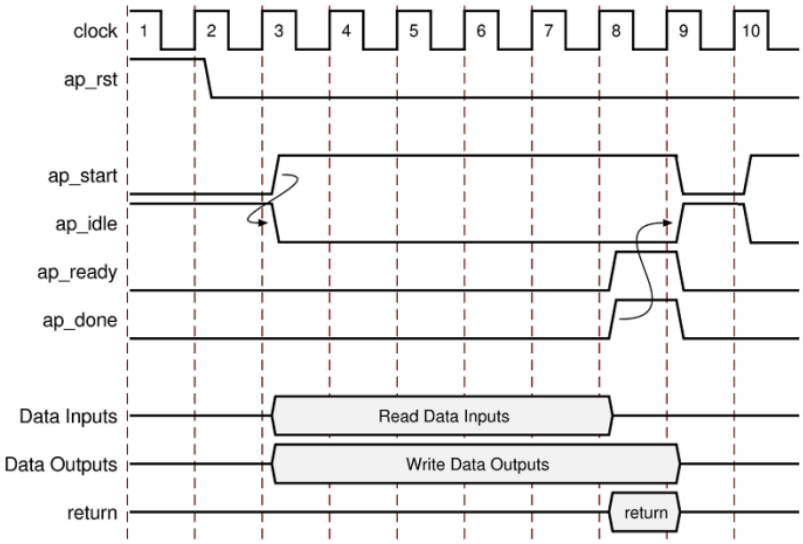


Figure 4.12: RTL Port Timing

Figure 4.13 shows the IP for energy and force calculations packaged using Vivado HLS. The *input* and *output m_axi* interfaces are bundled into a single *m_axi_gmem* interface. The return port is represented as *s_axi_AXILiteS* interface. *ap_clk* and *ap_rst_n* represents the clock and active low reset signal port respectively.

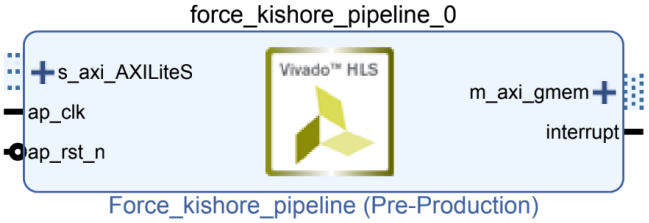


Figure 4.13: IP for Energy and Force Calculations

4.10.2 System-Level Design

System-Level Design refers to the process of designing a system with all the necessary blocks to enable the system to carry out a specific task. System-Level Design allows us to develop complex systems by utilizing pre-designed IP blocks, enabling faster development cycles. Xilinx Vivado 2017.4 is used to design the system as a block design. In our project, the System-Level design includes the following IP blocks:

1. Custom IP: This is the IP designed and packaged using Vivado HLS. This IP takes coordinates data from the DDR Memory, calculates the energy and forces values and stores the values back into the DDR Memory.

2. MIG 7 Series IP: Memory Interface Generator (MIG) IP is used to create a memory interface to the DDR memory on the FPGA board.

3. UARTlite IP: UARTlite IP is used for serial communication with the FPGA board. This IP is used to receive the coordinates values from the host and send the energy and force values to the host.

4. MicroBlaze IP: MicroBlaze is a soft processor core developed by Xilinx. It is a configurable and customizable 32-bit RISC (Reduced Instruction Set Computer) processor that can be implemented in Xilinx FPGAs for use in embedded system design. It is used for initializing and configuring the IPs in the system.

5. Additional IP blocks: Apart from the IP blocks mentioned above, additional blocks such as System Reset IP, Debug IP, AXI SmartConnect IP and AXI Interconnect IP are included in the design.

AXI SmartConnect IP is used to interface the MIG 7 IP with the custom IP and the MicroBlaze IP whereas AXI Interconnect IP is used to interface all the remaining IPs with each other. System Reset IP is used to generate a reset signal that is reset the corresponding IP to which it is connected. Debug IP is used to debug the MicroBlaze IP.

Clock Frequency

All the above mentioned IPs require a clock signal to function. The Genesys2 FPGA board has differential system clock of 100MHz. In our system design the system clock is connected to the MIG 7 series IP. The MIG 7 series IP is

capable of generating multiple clock frequencies. MIG 7 series IP is configured to generate a 100MHz clock signal which is used as clock signal by all the other IPs in the system. DDR Memory works at 200MHz clock frequency.

Once all the IPs are added and necessary connections are made, the design is validated to check for any design violations. On successful validation, the design is synthesized, implemented and bitstream is generated. Once done, the necessary files and information required to program and configure the FPGA hardware are generated by exporting the hardware in Vivado.

4.10.3 FPGA Programming using SDK

Once the system design is completed and the necessary files required to program the FPGA are generated, an application is created to specify the task to the FPGA. This includes initializing and configuring the various IP blocks needed to carry out the necessary task. The task at hand is to receive the coordinates values via UART and store in DDR, run the force IP, send the values from DDR to host via UART. This is to be carried out for multiple cycles.

Before initializing and configuring the various IP blocks, the necessary files are included and necessary variables are declared as shown below.

```
#include "xil_io.h"
#include "xuartlite.h"
#include "xforce_kishore_pipeline.h"
#include "xparameters.h"

XForce_kishore_pipeline force_ip;
XUartLite uart_ip;

#define force_ip_id XPAR_FORCE_KISHORE_PIPELINE_0_DEVICE_ID
#define uart_ip_id XPAR_AXI_UARTLITE_0_DEVICE_ID
#define ddr_base XPAR_MIG_7SERIES_0_BASEADDR
```

The *xparameters.h* file contains parameters related to all IPs in the system. The *xforce_kishore_pipeline.h* and *xuartlite.h* files contain the functions required to use the force IP and UART IP respectively. The *xil_io.h* file contains the functions required for memory I/O operations.

To initialize the UART and force IP, the following functions are used.

```
XUartLite_Initialize(&uart_ip,uart_ip_id);
XForce_kishore_pipeline_Initialize(&force_ip,force_ip_id);
```

To send and receive data over UART, the following functions are used.

```
while(1)
{
RecvCount+=XUartLite_Recv(&uart_ip,(u8 *)input_offset + RecvCount,
1764 - RecvCount);
if(RecvCount == 1764)
break;
}

while(1)
{
SendCount+=XUartLite_Send(&uart_ip,(u8 *)output_offset + SendCount,
1768 - SendCount);
if(SendCount == 1768)
break;
}
```

The numbers 1764 (441*4) and 1768 (442*4) represent number of bytes received and sent. Each floating point value is represented as 4 bytes as per IEEE 754 Standard.

To set the memory offset for *m_axi* interface configured as *slave*, the following functions are used.

```
XForce_kishore_pipeline_Set_input_r(&force_ip,input_offset);
XForce_kishore_pipeline_Set_output_r(&force_ip,output_offset);
```

To start the force IP and wait for the Done signal from the IP, the following functions are used.

```
XForce_kishore_pipeline_Start(&force_ip);
while(1)
{
if(XForce_kishore_pipeline_IsDone(&force_ip)==XST_SUCCESS)
break;
}
```

Communication between Host and FPGA

The communication between the host system (PC) and the FPGA system is acheived on the host side with the help of a python program. The python program does the following tasks:

1. Read floating point coordinates data from .txt file.
2. Convert the floating data into bytes and transmit the data byte by byte over UART using Serial library.
3. Receive the energy and force values, convert the received data in bytes to floating values and save as .txt file.

Chapter 5

Results and Analysis

This chapter presents the results obtained from implementing the ANN-MD System on CPU, GPU and FPGA. The results are analysed and a comparative study is carried out on results obtained from implementing the ANN-MD System on CPU, GPU and FPGA.

5.1 Data Analysis

The ANN-MD calculations are carried out on CPU, GPU and FPGA. The specifications of the CPU, GPU and FPGA used are given below.

CPU		GPU	
CPU Name	Intel Xeon	GPU Name	Nvidia Tesla T4
RAM Size	12GB	RAM Size	16 GB
CPU Cores	1	GPU Cores	2560
Frequency	2.3 GHz	Frequency	585 MHz

FPGA	
Board Name	Genesys2 FPGA Board
FPGA Architecture	Kintex - 7
FPGA Part	xc7k325fttg900 - 2

Hardware Components of FPGA	
CLBs	50,950
DSPs	840
BRAM (18KB)	890
FFs	4,07,600
LUTs	2,03,800

Note:

1. Each CLB consists of four LUTs and eight flipflops.
2. Each DSP consists of a pre-adder, a 25 x 18 multiplier, an adder, and an accumulator.

5.2 GPU Results

Google Colab platform is used to utilize the GPU services. Google Colab provides Nvidia Tesla T4 GPU for research purpose free of cost. The energy and force calculations are performed on the Tesla T4 GPU using Google Colab. Colab is a cloud platform that provides free access to computing resources, including GPUs and TPUs. It's environment is based on Jupyter Notebook and has CUDA preinstalled in it. A couple of steps are required to configure Colab so as to run CUDA programs on GPU using Colab.

1. Change Runtime configuration to use GPU services
    - In the *Runtime* tab, select *Change Runtime Type* and opt for *GPU* in the Hardware accelerator options.
2. Install and load *nvcc* extension for Jupyter Notebook

The following code is used to install and load the *nvcc* extension for Jupyter Notebook

```
!pip install git+https://github.com/andreinechaev/nvcc4jupyter.git
%load_ext nvcc_plugin
```

The first line installs the *NVCCPlugin* from the github repository mentioned. The second line uses *%load_ext* cell magic command to load the installed extension.

To check the version of CUDA available the following command is used.

```
!nvcc –version

Output:
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2022 NVIDIA Corporation
Cuda compilation tools, release 11.8, V11.8.89
```

Kernel Configuration

In CUDA code, the energy and force kernels are written in such a way that each instance of kernel calculates energy and force of one particular atom which is decided by thread indexing. So the total number of kernel instances required for Au_{147} and Au_{309} are 147 and 309 respectively. Different kernel configurations of blocks and threads per block for the required number of kernel instances are used and there performance is studied. The kernel combination which gives better timing results is choosen. Combinations from 1 block with 147 threads (1,147) to 147 blocks with 1 thread each (147,1) are used. All the configurations have shown similar timing results with no larger deviations. Since each block executes on a single streaming multiprocessor (SM), for better SM utilization 147 blocks with 1 thread per block configuartion is choosen for Au_{147} calculations. Similarly, 309 blocks with 1 thread per block configuartion is choosen for Au_{309} calculations.

```
int nB; // Number of blocks
int nTpB; //Number of threads per block

energy <<< nB,nTpB >>>(xyz,nlm,meangtot,gtotmin,gtotmax,
a01,a12,a23,t_cnlm,grad_e,ene);
cudaDeviceSynchronize();

forcee<<< nB,nTpB >>>(xyz,nlm,meangtot,gtotmin,gtotmax,t_cnlm,
grad_e,force);
cudaDeviceSynchronize();

* (nB,nTpB) is choosen as (147,1) and (309,1) for Au147 and Au309 calculations respectively
```

Once the kernel configuration is set, the energy and force calculations and integrated with the verlet algorithm and run for multiple MD steps and the energy values are verified.

Energy and Force Calculations

The energy and force calculations are performed on single CPU system and a heterogeneous (CPU + GPU) system. The calculations on both the systems are verified by comparing the sum of energies of all the atoms in the nanocluster.

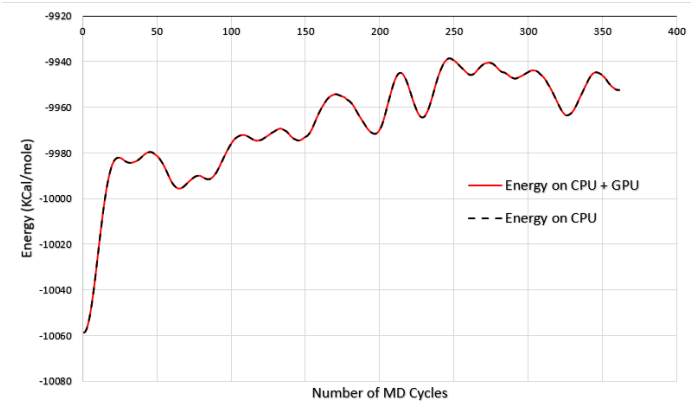


Figure 5.1: Au_{147} Energy Plot

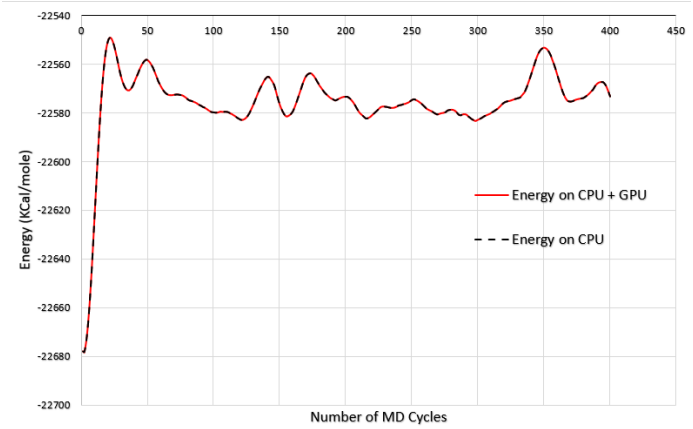


Figure 5.2: Au_{309} Energy Plot

Figures 5.1 and 5.2 show that the energy values for Au_{147} and Au_{309} obtained CPU system and heterogeneous (CPU + GPU) system are same upto two decimal point values. The above graphs indicate the correctness of energy and force calculations on the CPU + GPU system.

After verifying the energy values, the ANN-MD system is run for multiple MD steps (MD Cycles) on the heterogeneous (CPU + GPU) system. The timings results obtained for 1, 100 and 500 MD steps are represented as GPU timings in the table below. The GPU timing results are compared with timing results presented in [12] obtained by running the ANN-MD calculations on a HPC Server. In [12], the calculations are carried out using the HPC Server consisting of 7 CPUs by parallelising the calculations using Message Passing

Interface (MPI).

Before running the ANN-MD system for multiple steps, the MD system has to be initialized. This step add a MD step overhead to the total MD steps. The below timing results includes the mentioned one MD initialization step overhead. In the following table, the timing results of Au_{147} system are represented without parenthesis and the timing results of Au_{309} system are represented in parenthesis for 1, 100 and 500 MD steps.

MD Steps	HPC Server Timings	GPU Timings
1	4.18 sec (16.75 sec)	1.22 sec (3.26 sec)
100	3.51 min (14.08 min)	61.91 sec (164.31 sec)
500	17.43 min (69.94 min)	307.19 sec (817.83 sec)

The above comparative study shows that an acceleration of 3.4 times is achieved by the heterogeneous (CPU + GPU) system in comparison with the HPC Server with 7 CPUs for ANN-MD calculations of Au_{147} system. For Au_{309} system, an acceleration of 5.1 times is achieved by the heterogeneous (CPU + GPU) system in comparison with the HPC Server with 7 CPUs.

5.3 FPGA Results

IP Design for FPGA

The ANN based energy and force calculations are packaged into an IP using Vivado HLS. The C code for ANN based energy and force calculations is synthesized initially without any *directives* and the synthesis report is shown in the table below. The C code is then optimized to reduce the latency by adding required *directives* as discussed in section 4.10.1 and the synthesis report for Au_{147} calculations is shown in the table below.

Without Pipeline Directive - Au_{147}

Latency (Clock Cycles)

Minimum	Maximum
120,589,559	10,137,028,292

Resource Utilization

	BRAM_18KB	DSP	FF	LUT
Used	282	133	25,138	40,763
Available	890	840	407,600	203,800
Utilization (%)	31	15	6	20

With Pipeline Directive - Au_{147}

Latency (Clock Cycles)

Minimum	Maximum
16,236,170	204,895,970

Resource Utilization

	BRAM_18KB	DSP	FF	LUT
Used	287	361	114,619	102,027
Available	890	840	407,600	203,800
Utilization (%)	32	42	28	50

Similar to Au_{147} calculations, Au_{309} calculations are also packaged into an IP and optimized to reduce the latency by adding required *directives*. The synthesis report for Au_{309} calculations is shown in the table below.

Without Pipeline Directive - Au_{309}

Latency (Clock Cycles)

Minimum	Maximum
295,283,201	44,538,963,998

Resource Utilization

	BRAM_18KB	DSP	FF	LUT
Used	562	133	25,306	41,224
Available	890	840	407,600	203,800
Utilization (%)	63	15	6	20

With Pipeline Directive - Au_{309}

Latency (Clock Cycles)

Minimum	Maximum
61,160,390	818,148,590

Resource Utilization

	BRAM_18KB	DSP	FF	LUT
Used	568	361	114,795	102,227
Available	890	840	407,600	203,800
Utilization (%)	64	42	28	50

Note:

1. The C code contains arrays of large dimensions and many nested loops.
2. *ARRAY_PARTITION* directive is used to partition the arrays to implement using the BRAMs.
3. *PIPELINE* directive is used to pipeline the nested loops so as to reduce the latency of the loops.
4. Adding an extra *PIPELINE* directive has resulted in the resource utilization percentage of some FPGA components to go beyond 100%.

Energy and Force Calculations

Using the optimized IPs, a system-level design is implemented on the Genesys2 FPGA board for calculating the energy and force values of Au_{147} and Au_{309} systems. The following system-level configurations are used:

- System Operating Frequency: 100 MHz
- Communication Protocol: Universal Asynchronous Receiver-Transmitter (UART)
- UART Baud Rate: 115200 Bits per second (bps)
- UART Frame:

Start Bit logic 0	D0	D1	D2	D3	D4	D5	D6	D7	Stop Bit logic 1

Using the mentioned configuration, the coordinates are sent to the FPGA system via UART in the form of bytes.The FPGA system receives these values and stores them in the DDR Memory. After all the coordinates are received and stored in DDR, the IP is started to perform the energy and force calculations. The IP copies the coordinates data from the DDR memory, does the required calculations and copies the energy and force values into the DDR Memory. Once the IP has completed its calculations, the FPGA system sends the energy and force values from DDR memory to the host (CPU) via UART.

- Number of Bytes received by the FPGA System:
  441*4=1764 (927*4=3708)
- Number of Bytes sent by the FPGA System:
  442*4=1768 (928*4=3712)

	Bytes Received	Bytes Sent	Time
Au_{147} System	1764	1768	0.32 sec
Au_{309} System	3708	3712	0.66 sec

The received energy and force values matched with the energy and force values obtained from GPU for a single cycle. On verifying the correctness of the data received,the FPGA system for energy and force calculations was integrated with the verlet algorithm running on the CPU to compute ANN-MD calculations for multiple MD steps. Running the system for multiple MD steps resulted in the following:

- Energy values received for different MD steps do not match the energy values obtained using GPU shown in figures 5.1 and 5.2.
- Two consecutive MD steps have the same energies.

Since the correctness of the IP calculations are verified, the above observations should be due to the communication between the CPU and the FPGA system. This arises due to the continues and fast nature of the transmissions between both the CPU and FPGA system. The lack of synchronization between the CPU and FPGA system may also be a reason for such outcomes.

Chapter 6

Conclusion

6.1 Contributions

ANN-MD calculations have been implemented for multiple MD steps on a heterogeneous (CPU + GPU) system for Au_{147} and Au_{309} clusters. This has been achieved by offloading the energy and force calculations onto the GPU for parallel processing. The coordinate values for next time step are calculated on the CPU using the energy and force values calculated by the GPU. The timing results obtained for multiple MD steps have been compared with HPC Server timings in [12]. An acceleration of 3.4 times and 5.1 times compared to the HPC Server, has been achieved for Au_{147} system and Au_{309} system respectively. The energy and force calculations for Au_{147} system and Au_{309} system are also implemented on an FPGA and the timing results for the calculations are presented.

6.2 Future Work

In FPGA implementation, communication between host (CPU) and FPGA board plays a major role in deciding the overall system performance. In this project, while running the ANN-MD calculations for multiple steps on FPGA, the energy values for both Au_{147} and Au_{309} systems do not follow the graphs shown in figure 5.1 and 5.2 respectively. One reason could be the lack of proper data synchronization mechanism between the host and FPGA. Analysis can be done to investigate the reasons behind the deviation of energy values. Adopting an efficient data synchronization mechanism between the host and the FPGA board which can be used with different communication protocols (UART, Ethernet, PCIe etc) can help in rectifying the issue.

Bibliography

[1] M. Zhang, K. Hibi, and J. Inoue, "Gpu-accelerated artificial neural network potential for molecular dynamics simulation," *Comput Phys Commun*, vol. 285, p. 108 655, Apr. 2023. DOI: 10.1016/J.CPC.2022.108655.

[2] M. Szydlarski, P. Esterie, J. Falcou, L. Grigori, and R. Stompor, "Parallel spherical harmonic transforms on heterogeneous architectures (graphics processing units/multi-core CPUs)," *Concurr Comput*, vol. 26, no. 3, pp. 683–711, Mar. 2014. DOI: 10.1002/CPE.3038.

[3] W. Huang, Z. Khalid, and R. A. Kennedy, "Efficient computation of spherical harmonic transform using parallel architecture of CUDA," in *5th International Conference on Signal Processing and Communication Systems, ICSPCS'2011 - Proceedings*, 2011. DOI: 10.1109/ICSPCS.2011.6140886.

[4] Y. Hu, Y. Liu, and Z. Liu, "A survey on convolutional neural network accelerators: GPU, FPGA and ASIC," in *2022 IEEE 14th International Conference on Computer Research and Development, ICCRD 2022*, 2022, pp. 100–107. DOI: 10.1109/ICCRD54409.2022.9730377.

[5] S.-H. Rhee, T. Kim, and H.-S. Kim, "A high performance gpu for 3d graphics and gpgpu computing," *IEEE Transactions on Consumer Electronics*, vol. 58, no. 2, pp. 396–404, 2012. DOI: 10.1109/TCE.2012.6243276.

[6] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*, https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[7] NVIDIA Corporation, *NVIDIA CUDA Library , howpublished = https://developer.download.nvidia.com/compute/devzone/docs/html/c/doc/html/index.html , year =*.

[8] S. Jindal, S. Chiriki, and S. S. Bulusu, "Spherical harmonics based descriptor for neural network potentials: Structure and dynamics of au147 nanocluster," *The Journal of Chemical Physics*, vol. 146, no. 20, p. 204 301, 2017.

[9] J. Behler and M. Parrinello, "Generalized neural-network representation of high-dimensional potential-energy surfaces," *Physical Review Letters*, vol. 98, no. 14, p. 146 401, 2007.

[10] J. Bullinaria, *John bullinaria's step by step guide to implementing a neural network in c*, `https://www.cs.bham.ac.uk/~jxb/INC/nn.html`.

[11] X. Inc., *Vivado high-level synthesis user guide*, UG902, version v2020.1, May 2021. [Online]. Available: `https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2020_2/ug902-vivado-high-level-synthesis.pdf`.

[12] S. S. Bulusu and S. Vasudevan, "Fpga accelerator for machine learning interatomic potential-based molecular dynamics of gold nanoparticles," *IEEE Access*, vol. 10, pp. 40 338–40 347, 2022. DOI: `10.1109/ACCESS.2022.3165650`.

hjvk

ORIGINALITY REPORT

11%	7%	6%	4%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	www.hds.bme.hu Internet Source	<1%
2	Siva Chiriki, Shweta Jindal, Satya S. Bulusu. "Neural network potentials for dynamics and thermodynamics of gold nanoparticles", The Journal of Chemical Physics, 2017 Publication	<1%
3	japan.xilinx.com Internet Source	<1%
4	Submitted to Pace University Student Paper	<1%
5	Satya S. Bulusu, Srivathsan Vasudevan. "FPGA accelerator for machine learning interatomic potential based molecular dynamics of gold nanoparticles", IEEE Access, 2022 Publication	<1%
6	Submitted to University of Glasgow Student Paper	<1%
7	Submitted to University of Wales Swansea Student Paper	<1%

8	Submitted to Edith Cowan University Student Paper	<1 %
9	livrepository.liverpool.ac.uk Internet Source	<1 %
10	lume.ufrgs.br Internet Source	<1 %
11	apps.dtic.mil Internet Source	<1 %
12	ir.lib.uth.gr Internet Source	<1 %
13	papasearch.net Internet Source	<1 %
14	Submitted to Higher Education Commission Pakistan Student Paper	<1 %
15	Submitted to Universiteit van Amsterdam Student Paper	<1 %
16	www.diva-portal.org Internet Source	<1 %
17	Submitted to Great Oak High School Student Paper	<1 %
18	Submitted to Macquarie University Student Paper	<1 %

19	Wu, Jing, and Joseph JaJa. "Optimized FFT computations on heterogeneous platforms with application to the Poisson equation", Journal of Parallel and Distributed Computing, 2014. Publication	<1 %
20	digilib2.unisayogya.ac.id Internet Source	<1 %
21	lup.lub.lu.se Internet Source	<1 %
22	technodocbox.com Internet Source	<1 %
23	Submitted to CSU Northridge Student Paper	<1 %
24	Morris, Gerald R., Antoinette R. Silas, and Khalid H. Abed. "Analytical and measured sustained bandwidth for an FPGA-based processor", 2012 Proceedings of IEEE Southeastcon, 2012. Publication	<1 %
25	Lanjun Wan, Kenli Li, Jing Liu, Keqin Li. " GPU implementation of a parallel algorithm for the subset-sum problem ", Concurrency and Computation: Practice and Experience, 2015 Publication	<1 %
26	riunet.upv.es Internet Source	

		<1%
27	www.inf.ed.ac.uk Internet Source	<1%
28	Submitted to University of Sussex Student Paper	<1%
29	lra.le.ac.uk Internet Source	<1%
30	media.quantum-espresso.org Internet Source	<1%
31	pubsonline.informs.org Internet Source	<1%
32	discuss.pytorch.org Internet Source	<1%
33	www.tdx.cat Internet Source	<1%
34	Gerassimos Barlas. "GPU programming: CUDA", Elsevier BV, 2023 Publication	<1%
35	Seongjae Lee, Taehyoun Kim. "Parallel Dislocation Model Implementation for Earthquake Source Parameter Estimation on Multi-Threaded GPU", Applied Sciences, 2021 Publication	<1%

36	Syed Tahir Hussain Rizvi, Gianpiero Cabodi, Gianluca Francini. "GPU-only unified ConvMM layer for neural classifiers", 2017 4th International Conference on Control, Decision and Information Technologies (CoDIT), 2017 Publication	<1 %
37	Submitted to University of Lincoln Student Paper	<1 %
38	downloads.hindawi.com Internet Source	<1 %
39	Jean Pierre Jessel. "A new method to optimize the force-directed placement for 3D large graph drawing", Proceedings of the 5th international conference on Computer graphics virtual reality visualisation and interaction in Africa - AFRIGRAPH 07 AFRIGRAPH 07, 2007 Publication	<1 %
40	china.xilinx.com Internet Source	<1 %
41	docs.espressif.com Internet Source	<1 %
42	micro.stanford.edu Internet Source	<1 %
43	www.scribd.com Internet Source	<1 %