Long-Range MD Electrostatics Force Computation on FPGAs

Sahan Bandara, Anthony Ducimo, Chunshu Wu, and Martin Herbordt

***Abstract*—Strong scaling of long-range electrostatic force computation, which is a central concern of long timescale molecular dynamics simulations, is challenging for CPUs and GPUs due to its complex communication structure and global communication requirements. The scalability challenge is seen especially in small simulations of tens to hundreds of thousands of atoms that are of interest to many important applications such as physics-driven drug discovery. FPGA clusters, with their direct, tightly coupled, low-latency interconnects, are able to address these requirements. For FPGA MD clusters to be effective, however, single device performance must also be competitive. In this work, we leverage the inherent benefits of FPGAs to implement a long-range electrostatic force computation architecture. We present an overall framework with numerous algorithmic, mapping, and architecture innovations, including a unified interleaved memory, a spatial scheduling algorithm, and a design for seamless integration with the larger MD system. We examine a number of alternative configurations based on different resource allocation strategies and user parameters. We show that the best configuration of this architecture, implemented on an Intel Agilex FPGA, can achieve $2124ns$ and $287ns$ of simulated time per day of wall-clock time for the two molecular dynamics benchmarks DHFR and ApoA1; simulating 23K and 92K particles, respectively.**

***Index Terms*—Electrostatics computation, FPGA acceleration, grid mapping, molecular dynamics, particle mesh ewald.**

I. Introduction

ACCELERATION of Molecular Dynamics (MD) simulations is critical: there is a many orders-of-magnitude gap between the largest current simulations and physical systems of interest [1], [2]. There are dozens of MD packages which support GPUs (e.g., [3], [4], [5], [6], [7]). Scalability, however, remains problematic for the small simulations (20K–50K particles) commonly used, e.g., in drug design [8], [9], where long timescales are also extremely beneficial; several studies discuss challenges for CPU, GPU, and CPU+GPU heterogeneous clusters [10], [11], [12]. Simulation of long timescales of small molecules is a motivation for the Anton family of ASIC-based MD engines [13], [14], [15]. Anton addresses scalability, in part, by combining dedicated MD computing with direct communication links–application layer to application layer–within and among the integrated circuits. Since FPGAs also support these capabilities, they represent a comparatively low-cost COTS alternative [16].

The challenge in strong scaling of MD results, especially, from the long-range (LR) electrostatic force computation. In this work, we present an FPGA-based computation unit implementing LR through the Particle Mesh Ewald (PME) method [17]. When using the PME for LR, the three dimensional (3D) Fast Fourier Transforms (FFT), which requires global communication, poses perhaps the greatest hurdle. While clusters of GPUs are efficient for large FFTs, clusters of FPGAs have shown to outperform GPUs in performing 3D FFTs in smaller grid sizes [18]. Overall, this work builds upon work presented in [19], [20], [21], [22]; proposes novel solutions to challenges not fully addressed in prior work, and provides a unified architecture and algorithmic framework. The proposed design can be used either standalone or as part of a fully integrated MD simulation accelerator [23].

Apart from the FFTs, additional complexity arises from the charge and force mapping operations, which have both high fan-in/fan-out and require complex communication structures. FPGAs are particularly well-suited to both mapping and FFT, as well as the data movement and conversion operations, and the integration with the other MD computations. In particular, FPGAs' many thousands of independently addressable on-chip memories (Block RAMs or BRAMs)—together with support for creating application-specific interconnection networks that connect the BRAMs to each other and to the computation units—leads to extremely high efficiency. The main theme of this work is leveraging these FPGA features through a unified architectural and algorithmic framework based on memory organization and memory access patterns. Contributions include:

- Combining many previous partial solutions, on different components of the PME algorithm, to design a complete LR force computation architecture;
- Many fine-grained design innovations for an FPGA-LR force computation unit that can be either a stand-alone accelerator or part of a fully integrated MD accelerator;
- Establishing the requirements for an ideal memory structure for PME and proposing an application-specific interleaved memory architecture that satisfies those requirements;

Manuscript received 24 January 2023; revised 16 June 2024; accepted 22 July 2024. Date of publication 26 July 2024; date of current version 6 August 2024. This work was supported in part by the NSF through awards CCF-1919130 and CCF 2151021, in part by the NIH through award R44GM128533, and in part by AMD and Intel both through donated FPGAs, tools, and IP. Recommended for acceptance by M. Becchi. *(Corresponding author: Sahan Bandara.)*

Sahan Bandara and Martin Herbordt are with ECE Department, Boston University, Boston, MA 02215 USA (e-mail: sahanb@bu.edu).

Anthony Ducimo was with ECE Department, Boston University, Boston, MA 02215 USA. He is now with Intrinsix Corp, Marlborough, MA 01752 USA.

Chunshu Wu was with ECE Department, Boston University, Boston, MA 02215 USA. He is now with the University of Rochester, Rochester, NY 14627 USA.

Digital Object Identifier 10.1109/TPDS.2024.3434347

- Proposing new algorithmic improvements for increasing the efficiency of charge mapping pipelines and data movement between RL and LR units, i.e., an on-chip spatial scheduling algorithm that results in 12× improved performance;
- A design for efficiently integrating LR with the Range-Limited (RL) force compute units;
- Establishing resource usage values and performance for a baseline configuration of the LR accelerators;
- Analysis of different resource allocation strategies and extrapolation of resource usage and performance for alternative configurations that, e.g., trade-off accuracy and performance;
- Experiments with a production MD package that compares the performance of FPGA-LR to that of a high-end GPU.

High-level significance of this work therefore includes the demonstration that chip-level FPGA processing of LR is a benefit, rather than a drawback, in using FPGA clusters for long timescale MD. The remainder of this paper is organized as follows. In Section II we describe MD simulations and long-range electrostatic force computation and discuss FPGAs as acceleration platforms for these. In Section III we present the overall LR force computation architecture and provide detailed descriptions of the components. Resource usage and performance baselines are established in Section IV. Section V presents alternative design configurations and their impact on performance and resource usage. We present prior work on FPGA- and ASIC-based MD accelerators in Section VI; and conclude the paper in Section VII.

II. MD Preliminaries

A. Overview

Molecular Dynamics uses an iterative application of Newtonian mechanics on ensembles of atoms and molecules. MD simulations alternate between force computation and motion integration. The types of forces computed depend on the system being simulated and may include: bonded terms – pairwise, angle, and dihedral; and non-bonded terms – Van der Waals, and Coulomb. A collection of functions and corresponding parameters computing these force components is often referred to as a force field, e.g.,

$$F^{total} = F^{bond} + F^{angle} + F^{dihedral} + F^{non-bonded} \quad (1)$$

The non-bonded force is comprised of Lennard-Jones (LJ) and Coulombic forces. For a particle i in an ensemble of particles, these forces can be calculated as:

$$F_i^{LJ} = \sum_{j \neq i} \frac{\epsilon_{ab}}{\sigma_{ab}^2} \left\{ 48 \left(\frac{\sigma_{ab}}{|r_{ji}|} \right)^{14} - 24 \left(\frac{\sigma_{ab}}{|r_{ji}|} \right)^{8} \right\} \mathbf{r_{ji}} \quad (2)$$

$$F_i^{C} = \frac{q_i}{4\pi} \sum_{j \neq i} \frac{1}{\epsilon_{ab}} \left\{ \frac{1}{|r_{ji}|} \right\}^3 \mathbf{r_{ji}} \quad (3)$$

where ϵ_{ab} (unit: kJ or kcal) and σ_{ab} (unit: meters) are parameters related to the types of particles.

To reduce the complexity of the non-bonded force computation from $O(N^2)$, it is generally split into two components: range-limited (RL), which is $O(N)$ (since each particle interacts with only a much smaller number of neighbors); and long-range (LR), which is often $O(N \log N)$ as will be described. The Coulombic force is itself split into two parts, fast decaying and slow decaying. The slow decaying part is the Long-Range (LR) force.

B. Long Range Electrostatics Computation

The Particle Mesh Ewald (PME) method is a grid-based method widely used to calculate LR electrostatic forces. It reduces the asymptotic complexity from $O(N_p^2)$ to $O(N_g log(N_g))$, where N_p is the number of particles and N_g is the number of grid points. While there are other methods, such as k-space summation [24], μ-series [25], and multi-grid [26], [27], the widespread support for PME in production-grade MD packages, such as OpenMM [6], GROMACS [5], and many others, makes it an important target for acceleration.

PME involves three main phases: (i) mapping particle charges to a discrete grid to create the charge distribution; (ii) deriving the potential field caused by the charge distribution; and (iii) calculating the force exerted on each particle by the potential field. In the first (mapping) and third (force computation) phases, particle ↔grid interpolations are used to derive charge densities at grid points from particle charges and to calculate electric field vectors at particle positions using the potential grid. These mappings are typically done using tricubic interpolation.

The method used here applies a third order basis function in distributing particle charges to grid points, and the gradient of the same basis function for electric field vector calculations. Equation (4) is used to calculate charge densities ρ_g at grid points from particle charges. Equations (5), (6), and (7) are used to calculate force vectors along each dimension at particle positions using the potentials φ_g at grid points. ϕ represents the basis function. Indices p and g denote particle and grid points respectively.

$$\rho_g = \sum_p Q_p \phi\left(|x_g - x_p|\right) \phi\left(|y_g - y_p|\right) \phi\left(|z_g - z_p|\right) \quad (4)$$

$$\vec{F_{p,x}} = \sum_g \varphi_g \partial\phi\left(|x_g - x_p|\right) \phi\left(|y_g - y_p|\right) \phi\left(|z_g - z_p|\right) \quad (5)$$

$$\vec{F_{p,y}} = \sum_g \varphi_g \phi\left(|x_g - x_p|\right) \partial\phi\left(|y_g - y_p|\right) \phi\left(|z_g - z_p|\right) \quad (6)$$

$$\vec{F_{p,z}} = \sum_g \varphi_g \phi\left(|x_g - x_p|\right) \phi\left(|y_g - y_p|\right) \partial\phi\left(|z_g - z_p|\right) \quad (7)$$

$$\phi(\xi) = \begin{cases} (1-|\xi|)(1+|\xi|-\frac{3}{2}\xi^2) & |\xi| \leq 1 \\ -\frac{1}{2}(|\xi|-1)(2-|\xi|)^2 & 1 \leq |\xi| \leq 2 \\ 0 & 2 \leq |\xi| \end{cases} \quad (8)$$

Equation (8) shows the third order basis function where ξ is the distance between the particle and any grid point. Following [20], the basis function is modified to be a set of polynomials of o_i. By substituting ξ with o_i+1, o_i, $1-o_i$, and $2-o_i$, four polynomials, corresponding to the four neighboring grid points shown in (9),

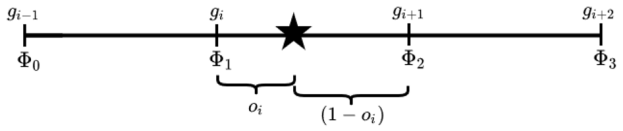

Fig. 1. Grid index and offset.

are derived. Fig. 1 depicts how the term o_i relates to the particle position and neighbor grid points.

$$\begin{cases} \phi_0(o_i) = -\frac{1}{2}o_i^3 + o_i^2 - \frac{1}{2}o_i \\ \phi_1(o_i) = \frac{3}{2}o_i^3 + \frac{5}{2}o_i^2 + 1 \\ \phi_2(o_i) = -\frac{3}{2}o_i^3 + 2o_i^2 + \frac{1}{2}o_i \\ \phi_3(o_i) = \frac{1}{2}o_i^3 - o_i^2 \end{cases} \tag{9}$$

The second phase involves computing the electric field generated by the charge grid. The solution to the Poisson equation is the electric potential generated by a given charge distribution. In three-dimensional Cartesian coordinates, the Poisson equation takes the form

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}\right)\varphi(x,y,z) = f(x,y,z) \tag{10}$$

Usually, f is given and φ is sought. In electrostatics, φ and f represent the electric potential and the charge density distribution, respectively. The Poisson equation can be solved by computing the convolution between the charge distribution and a Green's function.

PME uses FFTs to replace convolution in real space with multiplication in Fourier space. First, a 3D FFT is performed to transform the charge grid into the Fourier domain. Next, the resulting grid is multiplied by the Fourier transform of the Green's function. Finally, an inverse 3D FFT is performed to transform the result back into real space. The final phase of PME is calculating the force exerted on each particle by the potential field. The derivatives of the basis functions used for particle to grid charge mapping are used to compute the electric field vectors at particle positions. Force vectors along X, Y, and Z dimensions are calculated as the output of the LR force computation.

C. Problem and Grid Sizes of Interest

Various molecular modeling techniques, e.g., quantum mechanics methods, hybrid quantum/molecular mechanics, MD, and coarse-grained MD, are best suited for certain problem sizes. MD is typically used to target systems ranging from thousands to many millions of atoms [28]. Table I provides an overview of system sizes, corresponding PME grid sizes, typical use cases, and matching benchmarks. The grid sizes are computed assuming an Ewald error tolerance of 0.0005 following the GPU benchmark setup used in [29].

D. Mapping MD Computation to Hardware

An MD/LR hardware accelerator implementing the methods in Section II-B must have the following components: particle memory (positions, velocities, and forces) organized into cells

TABLE I
OVERVIEW OF PARTICLE NUMBERS AND GRID SIZES FOR MD SIMULATIONS

Atoms	Grid size	Domain	Benchmark
1,000s	$\geq 20^3$		
10,000s	$\geq 42^3$	Protein folding [30] Enzyme-inhibitor binding [31] Drug discovery [32]	DHFR (23,558 atoms) ApoA1 (92,224 atoms)
100,000s	$\geq 89^3$		Cellulose (408,609 atoms)
>1,000,000	$\geq 192^3$	Viral capsids [33] Ribosome [34] Photosynthetic membrane [35]	STMV (1,067,095 atoms)

for easy reference during RL; charge grid memory; potential grid memory; 1-64 charge grid creation pipeline(s); 64-1 force application pipeline(s); 1D FFT pipeline(s); and the interconnections among the memories and pipelines. For maximal efficiency, the memories must enable parallel accesses: for the mapping steps, writing/reading 64 grid charges/potentials to/from $4 \times 4 \times 4$ subgrids; for the FFT steps, reading and writing as many 1D vectors as possible.

To integrate MD/LR into a complete MD accelerator, additional data structures and computations are needed. To keep this study focused on LR, we associate motion integration with RL (at no change in hardware cost or performance). Since the RL unit performs motion integration, it also stores the particle positions and calculated force values in caches (*position* and *force*). The RL unit subdivides the simulation space into a grid of cells as the range-limited forces are evaluated only considering particles within a cut-off radius. There are position and force caches corresponding to each cell. The *position caches* store particle positions that are fed to the RL force computation pipelines. After motion integration, the updated particle positions are written back to the position caches. The force caches store force values calculated by the different MD parts until used by motion integration pipelines. The LR unit is responsible for retrieving particle position information from the position caches, computing LR, and transferring the force values to the force caches. Force caches include accumulation logic to accumulate force values computed by the RL and LR units. Details of the RL unit are in [36], [37], [38], [39], [40].

E. LR Computation and Hardware Acceleration

The phases described in Section II-B are computed as shown in Fig. 2 (in 2D). Fig. 2(a) shows particles in a simulation space organized into cells, as needed for the RL computation. Fig. 2(b) shows the first step in PME: distributing the charge of a particle onto 64 nearby points of a charge grid (16 for 2D). Note that the grid geometry is unrelated to that of the cells. The next step is the 3D FFT (2D here), which is computed with 1D FFTs in successive dimensions (Fig. 2(c) and (d)). The next steps are the multiplication and inverse FFTs (Fig. 2(e) and (f)); at this point a potential grid has been created. Fig. 2(g) shows the forces from 64 nearby grid points (16 for 2D) being applied to a particle.

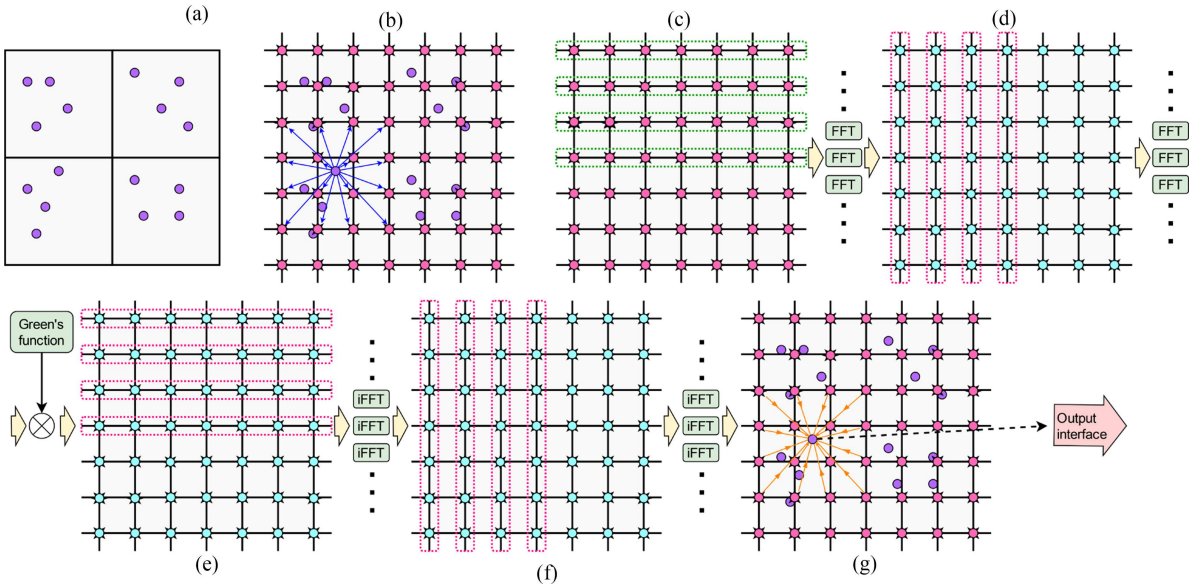

Fig. 2. Long-range force computation overview.

Designing a long-range force computation accelerator requires solving several architectural challenges. While much optimization is possible within the compute pipelines, the biggest challenges are in organizing, accessing, and interconnecting the various memories. The challenge is, therefore a data movement problem given FPGA BRAM architecture: to find the best layouts and indexing schemes to optimize parallel accesses, while minimizing conflicts to ensure efficient data movement, which results in high pipeline utilization. Going beyond the LR force computation, there are further data movement challenges in performing efficient data exchanges between RL and LR units. In this work, we present novel solutions that far outperform prior work in data movement efficiency.

The different phases of the LR force computation involve two distinct memory access patterns, neither suitable for traditional memory architectures. In the first, the charge mapping and force computation phases require accessing potentially overlapping subgrids from a 3D grid that overlays the simulation space. In the second, the FFT phase requires reading 1D vectors from the same grid data structure along each of the three dimensions, preferably without having to transpose data in memory. While prior work has provided solutions for these access patterns individually, a memory architecture that supports both access patterns with high throughput, and without additional stages of memory manipulation, has not previously been presented. In the following sections we show how an efficient "virtual" memory architecture can be built with spatially distributed, individually addressable blocks of memory, such as block RAMs on an FPGA. We also demonstrate address sequences that ensure the memory bandwidth is fully utilized in every phase of the computation.

Part of the challenge is that because the charge mapping stage requires both reading and writing of grid memory, RAW hazards result. As data forwarding is prohibitively expensive, prior work dealt with these through pipeline stalls. Also, complete position information was first cached in the LR unit before feeding the data to the charge mapping pipelines. In a complete MD system, however, this leads to inefficiencies in data movement between RL and LR units.

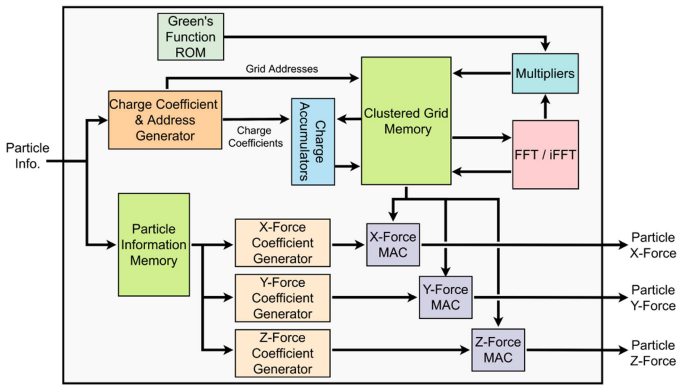

Fig. 3. Long-range force computation architecture.

Two optimizations are proposed to deal with these issues. First, data movement from RL to LR is scheduled: hazards can be avoided by first spatially sorting particles. For simulations of more than 10K particles, nearly 100% pipeline utilization is achieved resulting in a 12× performance improvement. Moreover, these schedules can be computed offline for a given simulation size without considering the actual particle positions. Second, we overlap data transfers with computation so that only a small fraction of particles needs to be buffered between computations.

III. LR Force Computation Architecture

An overview of the LR force computation architecture is presented in Fig. 3. The LR architecture is comprised of three

stages, which roughly correspond to the three phases of the PME algorithm. The charge mapping pipeline accepts particle information from an external interface, interpolates the charge values to the 4x4x4 subgrid surrounding the particle, and updates the charge values in grid memory. Particle information is temporarily stored in *particle information memory* to be used in the force computation.

The *clustered grid memory* connects all three phases of the computation. After receiving charge values for all the particles from the charge mapping pipeline, the grid memory is read and the readouts are sent to the FFT units. The outputs of FFT and inverse FFT computations are written back to the grid memory. After the 3D FFT and inverse FFT (iFFT) computations, the grid memory holds the potential grid representing the electric field caused by the charge grid. In the final phase, the potential values read from the grid memory and the particle information read from particle information memory are sent to the force computation pipeline. LR force values for each of the particles are transmitted through the output interface. The entire LR pipeline uses single precision floating point values.

We examine each of the components in detail. The first subsection covers grid memory. The next subsection describes basic charge mapping, hazards, parallelization, and a performance model. After that we detail hazard avoidance through particle scheduling, which is followed by a subsection covering FFTs and iFFTs. There follows a subsection on force mapping and its parallelization. The final subsection describes features of LR needed to interact with RL.

A. Clustered Grid Memory

The clustered grid memory is the central component of the LR design as it is used in, and affects the performance of, every phase of the computation. First, the number of ports provided by the grid memory controls how fast the charge of a particle can be spread over 64 grid points. Second, in the FFT/iFFT phase, the grid memory's bandwidth determines how many parallel FFT units can be employed. And third, the number of read ports controls how many grid points can be read per cycle and used in force computation. Also, most previously explored FFT architectures, both FPGA- [18], [21] and ASIC-based [41], lend themselves well to performing 3D FFTs, but require a transpose phase between each of the 1D FFT phases where the stored values are rearranged to match the direction for the next 1D FFT. Eliminating the transpose could result in a significant improvement in FFT latency. With the above in mind, we can enumerate the characteristics of an ideal memory structure:

1) Store 64 bits of data per grid point (two 32-bit single precision floating values for real and imaginary parts of grid data).
2) Support reading and writing the 64 nearest neighbor grid points of a particle at a rate of one particle per clock cycle.
3) Provide the capability to perform consecutive 1D FFTs without transposing the grid data in between each FFT phase.

Meeting (1) is trivial. To meet (2), there must be at least 64 read and 64 write ports. (3) requires that the mechanism used to access consecutive grid points placed along any of the three dimensions be agnostic to the particular direction.

1) Memory Interleaving: The grid memory is made of 64 independently accessible and addressable memory banks, each of which is implemented with one or more FPGA block RAMs (BRAMs), depending on the BRAM geometry and the size of the grid memory. The grid points are assigned to the 64 memory banks to ensure that any subgrid of size $4 \times 4 \times 4$ can be read from (or written to) the memory banks in parallel. The subgrid size is selected to match the interpolation scheme used. The memory interleaving scheme is trivially generalizable to support different subgrid sizes and, therefore, different interpolation schemes.

The address of a datum at a grid point is the ordered tuple used to specify the location in 3D Cartesian coordinates. By partitioning the grid memory into $4 \times 4 \times 4$ subgrids, referred to hereafter as *neighborhoods*, the location of a grid point can be given with a *neighborhood ID* and a *neighbor ID*. The neighborhood ID refers to a subgrid in the charge (or potential) grid. The neighbor ID indicates an individual grid point inside a neighborhood. When mapping grid locations to the 64 memory banks, the neighbor ID is used to select the memory bank; the neighborhood ID is used to index into the particular memory bank. The two low-order bits of each of the X, Y, and Z coordinates are concatenated to form the neighbor ID. The remaining (high-order) bits of the three coordinates uniquely identify the grid neighborhood.

The following is an example of how the ordered tuple of a grid point in a $32 \times 32 \times 32$ grid is decomposed into its neighborhood and neighbor IDs.

$$\begin{aligned}(X, Y, Z) &= (11, 23, 8)\\ &= (01011_2, 10111_2, 01000_2)\\ &= (010_2, 101_2, 010_2) \cdot (11_2, 11_2, 00_2)\\ &= (2, 5, 2) \cdot (3, 3, 0)\end{aligned}$$

In a $32 \times 32 \times 32$ grid, grid point (11, 23, 8) is the neighbor (3, 3, 0) inside the neighborhood (2, 5, 2). The concatenated high-order bits $(010_2, 101_2, 010_2)$ are used to address memory bank (3, 3, 0). These translate to index 170 (010101010_2).

The core grid memory capability for the mapping phases is, for all $4 \times 4 \times 4$ subgrids in a grid, the simultaneous access of those subgrid points. This access mechanism can be visualized as a cluster of 64 read (or write) ports that moves through the grid. Fig. 4 depicts how the access cluster moves about to access neighbors from neighborhoods, thereby accessing grid points from the grid. Neighbors accessed by the access cluster are shown in white (or orange for the 0 point). From (A) to (B) to (C), the access cluster moves along the Z dimension. From (C) to (D) the access cluster moves along the Y dimension.

Note that each neighbor ID appears at most once inside an access cluster. This is necessary to ensure that the 64 ports of the access cluster access the 64 memory banks without overlaps, enabling all points of a subgrid to be read (or written) in parallel. Also, the access cluster is not required to be neighborhood-aligned: any neighbor ID can appear on any of the ports in

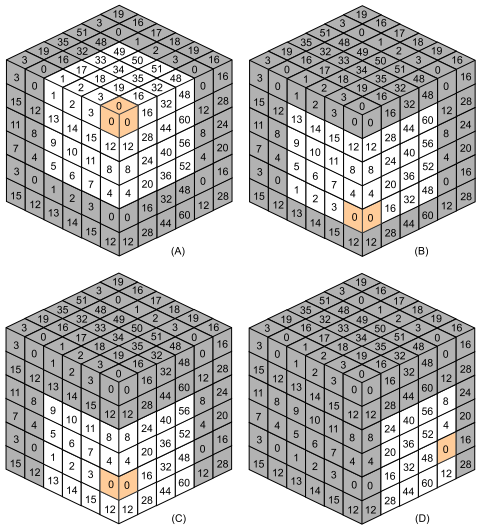

Fig. 4. Grid memory access patterns.

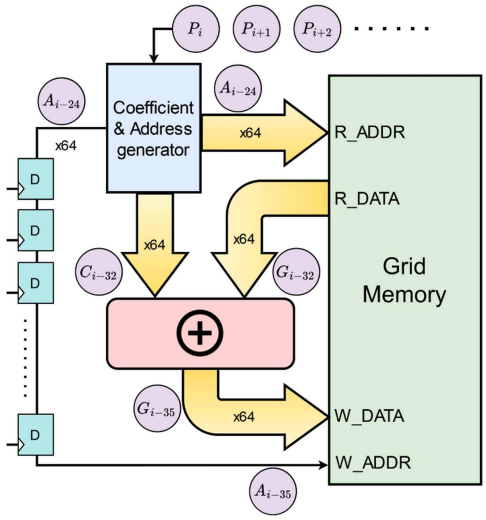

Fig. 6. Charge mapping pipeline.

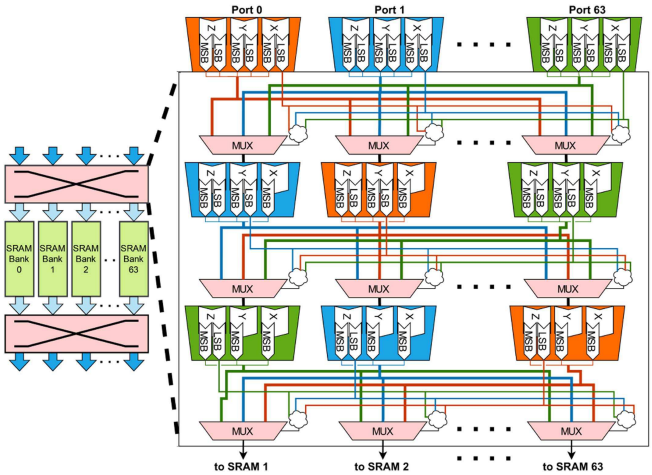

Fig. 5. Grid memory and crossbar architecture.

the access cluster. This is evidenced by the varying location of neighbor 0 (orange) relative to ports within the access cluster.

2) Grid Memory Implementation: Achieving the flexibility described in Section III-A1 requires that all cluster ports be connected to all memory banks. This all-to-all connectivity is achieved by the input and output crossbars using three-stage toroidal shift multiplexers. The neighbor indices requested at access cluster ports are shifted and realigned to match the alignment of the memory banks. At each stage, the access cluster is aligned along one dimension. The alignment control information is sent to each memory bank. The readouts from the memory banks go through the output crossbar, which routes the values to the correct output ports. Section III-D details how the access cluster uses different addressing sequences to support the access patterns required for FFT computations.

Fig. 5 shows how the crossbars and BRAMs are organized and provides a detailed view of a crossbar. When accessing the grid memory, each port is presented with the X, Y, and Z coordinates of one of the 64 grid points to be accessed. Routing requests from input ports to memory banks and readouts from memory banks to output ports are performed by the two crossbars. Both the input and output crossbars are implemented as multi-stage pipelines to improve operating frequency and routability. At each stage, a set of low-order bits from the requested coordinates is used to control the multiplexers. The first stage aligns the request along the X dimension using the low-order bits from the 64 X coordinates. Similarly, the second and third stages align requests along Y and Z dimensions, using the Y and Z coordinates respectively. Recall that the low-order bits correspond to the neighbor ID within the access cluster. Therefore, this essentially realigns the access cluster according to the neighbor IDs. After the low-order bits are used for controlling an alignment stage, they are captured at the input ports, sent through a separate set of pipeline registers, and used to control the output crossbar.

B. Charge Mapping Pipeline

This section describes an extension to previous work [20], [22] through parallelization and combining with the clustered grid memory. In charge mapping via tricubic interpolation, each charge is spread over 64 neighboring grid points. The charge mapping pipeline is 64-way vectorized to read grid memory, compute grid charge values, perform charge accumulation, and update grid memory for all 64 neighbor grid points in parallel. Fig. 6 provides a detailed view of the charge mapping pipeline and how it interacts with the grid memory. The *charge coefficient and address generator* receives particle position and charge information and generates the 64 coefficients corresponding to the 64 neighbor grid points and 64 grid addresses to access the grid memory. Next, the subgrid of 64 grid points is read from the grid memory, accumulated with the output of the charge coefficient generator, and written back to grid memory. The 64 addresses are sent through the pipeline to be used as the write addresses for the grid memory.

1) Handling Hazards: A critical complication arises as follows. Accumulating charge involves performing read-modify-write operations on the clustered grid memory, 64 grid points

per particle. When two particles have overlapping neighbor subgrids, this can lead to read-after-write (RAW) pipeline hazards. Referring to Fig. 6, $P_{<>}$ represents particle data fed into the pipeline; $A_{<>}$, and $C_{<>}$ represent a set of 64 grid addresses (or coefficients) generated by the coefficient and address generator; $G_{<>}$ corresponds to the values of 64 grid points read from (or written back to) the grid memory. The subscript indicates how far down the pipeline each of these values are relative to a particle entering the pipeline. For instance, the read addresses issued in the current clock cycle corresponds to a particle 24 pipeline stages removed from P_i. The issuing of the read addresses and the writing back of the updated grid point data are separated by 11 pipeline stages. Between these two steps, the grid memory contains stale data for the 64 grid points in question. A RAW hazard arises if a new particle has a grid neighborhood which overlaps that of the first. Out of the 11 pipeline stages, 9 are spent accessing the grid memory. It is possible to implement forwarding logic among the 3 grid memory write pipeline stages and 6 read stages and thereby limit pipeline stalls to the 2 stages where the actual accumulator problem occurs. However, this introduces unacceptable routing overhead as each pipeline stage handles 64 64-bit values.

Two other approaches to addressing RAW hazards are to (i) Implement hazard detection and pipeline stall logic or (ii) Schedule particle processing to avoid hazards. Since (i) results in unacceptably poor performance as stalls dominate performance, we have opted for (ii). Performance is discussed in Section III-C.

2) Parallel Charge Mapping: Charge mapping for particles that do not have overlapping neighbor subgrids can be done in parallel. This requires using multiple pipelines as well as increasing the read/write bandwidth of the grid memory. However, adding read/write ports to the grid memory drastically increases resource use and routing complexity. A practical approach is to instantiate multiple smaller grid memory modules and implement logic to have the smaller units mimic the behavior of one larger memory unit. Each of the smaller grid memory units still provides 64 ports so that each charge mapping pipeline can operate on one of the smaller grid memories. Implementation fits directly into the design since the simulation space is already partitioned into non-overlapping sub-regions (RL cells).

Parallelization of charge mapping is also limited by two other factors. One is that particle processing schedules have constraints (see Section III-C), e.g., simulation size. Another is that particles that are positioned close to region boundaries map to grid points in more than one grid memory unit and must be processed by more than one charge mapping pipeline; this is to ensure that all the relevant grid points are updated, and necessitates additional cycles.

3) Particles Near Region Boundaries: Particles near the boundaries of simulation space sub-regions can map to grid points that belong to different regions. This means the grid neighborhoods of these particles could be spread over multiple grid memory units. In the case of such particles, multiple charge mapping (force) computation pipelines work collaboratively to distribute charge values (compute the force vectors). Particles in boundary regions are therefore fed into more than one pipeline; each pipeline has to process more particles than the ones in the sub-region mapped to it.

4) Charge Mapping Latency Model: The latency of the charge mapping phase is determined by the number of particles N_p and the number of charge mapping pipelines P_{cmap}. The calculation is complicated, however, as pipelines may need to process particles from neighboring regions. The number of additional particles to be processed by each pipeline depends on how the simulation space is subdivided and assigned to different pipelines. The number of boundary particles per pipeline ($N_{B\,cmap}$) is specific to each configuration. The number of particles in the shadow regions near the boundaries can be specified in terms of N_p for a given configuration. For simplicity, we denote it as $N_{B\,cmap}$. After also taking into account the charge mapping pipeline depth (D_{cmap}), pipeline utilization (U_{cmap}), and read and write memory access latencies (M_R and M_W), the charge mapping latency is given by:

$$\frac{1}{U_{cmap}}\left(\frac{N_p}{P_{cmap}} + N_{B\,cmap}\right) + D_{cmap} + M_R + M_W \quad (11)$$

The D_{cmap} and M_R terms account for the number of cycles to fill the pipeline. The M_W term is necessary because the FFT phase cannot begin until the charge mapping pipeline outputs are written back to the grid memory.

C. Scheduling Particle Processing

This section describes an extension to previous work [20], [22] through scheduling particle computations. As described in Section III-B1, RAW hazards are removed by guaranteeing data independence by scheduling particles so that they do not have overlapping grid neighborhoods with some number of the preceding or following particles. This number is currently 11, which we use in this section to give a concrete, but generalizable, design. When this condition cannot be satisfied, one or more bubbles is inserted.

1) Grids, Subgrids, Padded Subgrids, and Cells: For example, consider a $4 \times 4 \times 4$ subgrid embedded in a $32 \times 32 \times 32$ grid. After a particle begins processing, not only is its subgrid voided, but also all of the surrounding subgrids within the $8 \times 8 \times 8$ *padded* subgrid. A schedule could therefore rotate among 64 disjoint regions (number of disjoint padded subgrids within the full grid) and thus result in no stalls. With a $16 \times 16 \times 16$ grid, however, some stalls would be necessary; utilization as a function of simulation size is described further below. Note that this calculation is not affected by the number of charge processing units.

Given that charge mapping is likely to be part of a larger MD computation that includes the range-limited (RL) force computation, particle scheduling can be greatly simplified by using the spatial sorting of particles already integral to RL. In particular, the RL force computation unit assigns particles to *cells* (typically cubic subdivisions of the simulation space of dimension similar to the cutoff radius) according to their position and stores the particle information in *position caches*. Position caches are independent blocks of memory, each corresponding

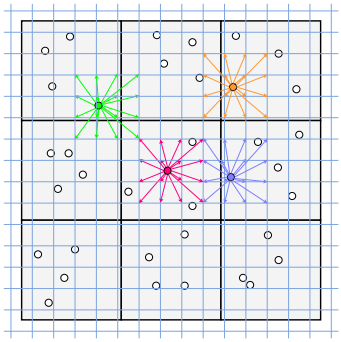

Fig. 7. Grids, cells, and particles visualized.

to a cell, that store particle information used by the RL force computation pipelines.

Essential to the scheduling algorithm is the relationship between cell and grid spaces with respect to simulations of interest for FPGA clusters. The latter are characterized by uniform distributions of particles (e.g., biomolecules in water), and by sizes in the 10s of thousands of particles, as are needed in drug design and protein folding. MD simulation software typically selects the grid size based on the size of the simulation space, cutoff radius, and the Ewald error tolerance specified by the user, unless the grid dimensions are explicitly specified by the user (e.g., [42]).

Concretely, the number of grid points is often similar to the number of particles. In the case of FPGA implementations, cell size is typically set to the cutoff radius r_c [36], [43]; cells often contain 50-100 particles. The key point is that, while there are many parameters, the ratio of cell size to grid size is such that a cell generally contains an entire $4 \times 4 \times 4$ subgrid. For instance, for a $64 \times 64 \times 64$ charge grid simulating water with a cutoff radius of 0.9 nm; a ratio of 4 between the number of grid points and RL cells corresponds to a simulation size of around 300K atoms and an Ewald tolerance of 5.5×10^{-3}. For comparison, the OpenMM default settings choose $56 \times 56 \times 56$, and $97 \times 97 \times 97$ charge grids for DHFR (23K atoms) and ApoA1 (92K atoms) benchmarks, respectively, for the same cutoff radius, because it targets much lower tolerances [42]. While the 4:1 assumption is conservative, if necessary, r_c can be increased to guarantee this relationship with little loss in RL performance.

2) Scheduling Algorithms: The spatial sorting of particles into cells performed by the RL unit can be leveraged to simplify the particle scheduling in the LR unit. However, it increases the padding from the surrounding subgrids ($8 \times 8 \times 8$ grid points) to the surrounding cells ($3 \times 3 \times 3$ cells or, with one subgrid per cell, $12 \times 12 \times 12$ grid points). Fig. 7 illustrates the relationship between some particles distributed in a cell, their subgrids, and how the subgrids correspond to cells.

Fig. 8 illustrates spatial scheduling. For simplicity, it is in 2D. In Fig. 8(a) we pick a particle from the cell labeled as '1'. This means that we have to avoid its neighbor cells when picking the next k particles, where k is the minimum number of clock cycles required between two particles with overlapping grid neighborhoods such as the *red* and *blue* particles in Fig. 7.

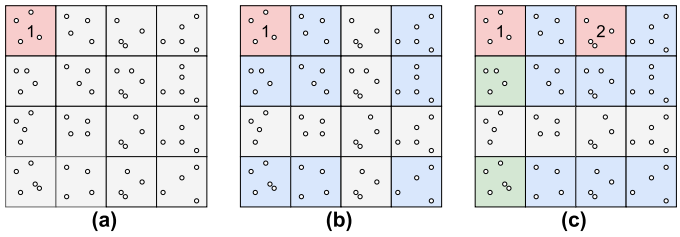

Fig. 8. Scheduling based on cells in RL unit.

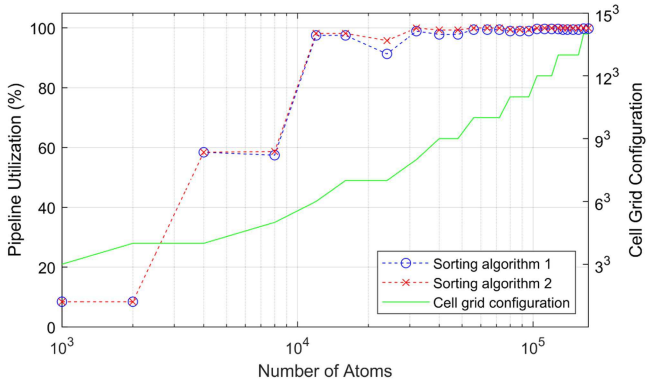

Fig. 9. Pipeline utilization for different scheduling algorithms.

Fig. 8(b) shows the neighbor cells of cell 1 shaded in blue. These cells are blocked out for the next k clock cycles. This also depicts how the periodic boundary conditions are applied. Fig. 8(c) shows the selection of the next particle from cell labeled '2'. This reduces the number of cells whence we can potentially pick the next particle to four. The blue cells are voided for the next k cycles, and the green cells for the next $(k-1)$ cycles.

Note that while particles from cells 1 and 2 can both update grid points within the cell in between, they never update overlapping grid points. Therefore, picking particles from cells 1 and 2 in consecutive cycles does not result in a RAW hazard. The *green* and *orange* particles in Fig. 7 represent such a scenario. If all cells are blocked out, bubbles are inserted until a cell becomes available. This procedure is repeated until all particles are sent to the charge mapping pipeline. It is always possible to pre-compute the selection sequence as it is independent of the particle positions.

This scheduling procedure allows for various algorithms. Two are as follows. Algorithm 1 converts the coordinates of the current cell into a linear address and increments that address to select the next target cell. If the target cell or its neighbor cells have not been selected in the last k iterations, a particle can be picked from that cell. Algorithm 2 selects the cell closest to the current cell as the target for the next iteration. This is done by iterating over the cell grid along X, Y, and Z dimensions (with wrap-around) starting from the current cell until a cell with a neighborhood not selected in the last k iterations is visited.

3) Effectiveness of Scheduling Algorithms: The effectiveness of the algorithms is given by the pipeline utilization and shown in Fig. 9. Pipeline utilization plotted against the number of atoms is a step function because the cell grid grows in

discrete steps. The first transition is the cell grid growing from $3 \times 3 \times 3$ to $4 \times 4 \times 4$. Both sorting algorithms show marginal or no improvement going from $4 \times 4 \times 4$ to $5 \times 5 \times 5$. From $6 \times 6 \times 6$ grid, there is nearly full utilization. Both algorithms are similarly effective, although Algorithm 2 provides slightly higher pipeline utilization. We have noticed, however, that Algorithm 1 results in higher pipeline utilization for some other cuboid-shaped simulation spaces, which are encountered when subdividing the simulation space into sub-regions for parallel charge mapping.

D. FFT and Inverse FFT

This section describes combining previous work [21] with clustered grid memory. The PME method requires a 3D FFT and a 3D inverse FFT to be performed over the charge grid. Before the iFFT phase, the result of the FFT is multiplied by the Green's function values stored in the *Green's ROM*. The forward and inverse Fourier transforms are built up from vendor-provided FFT IP cores. To fully utilize the read/write capability of the grid memory, 64 FFT cores are used by default. While this number can be reduced for small simulations, the FFT cores do not use many resources so the benefit is marginal.

Higher order FFTs can be decomposed into lower order ones (e.g., [41]). An $N \times N \times N$ charge grid requires N^2 N-point 1D FFTs per dimension, and each of the FFT cores perform $N^2/64$ N-point FFTs per dimension. For each 1D FFT, grid values are streamed out of the grid memory and outputs from the FFT units are streamed back to the grid memory. When performing the 1D FFT along the Z dimension, results are also sent through a set of multipliers to be multiplied by the Green's function values. Green's function values depend only on the location of the corresponding grid point, the true volume of the grid, the order of the basis function used to map the particle charges, and a convergence parameter [42]. Thus, the values of the Green's function at each grid point can be computed offline and stored in a ROM. The 3D iFFT is performed similarly.

The clustered grid memory provides 64 read ports and 64 write ports. Therefore, at full utilization, it should be able to service read and write requests from 64 FFT pipelines. However, the grid memory is designed to support accessing 64 grid points arranged in a $4 \times 4 \times 4$ layout, where each port is required to access a unique neighbor in the $4 \times 4 \times 4$ neighborhood. This is not conducive to the FFT calculation as the grid points are required to be sent to the FFT cores in a specific order. We decompose the 3D FFT into three sets of 1D FFTs. When performing 1D FFTs along a particular dimension, 1D arrays of grid points along that dimension need to be sent to the FFT cores. This is achieved by staggering 2D slices of the grid memory access cluster and assigning these slices to neighborhoods from different *districts*, where a district is an array of neighborhoods along a given dimension.

Fig. 10 illustrates the access cluster mapping using a $4 \times 4 \times 4$ grid memory and a $2 \times 2 \times 2$ access cluster. The two 2D slices of the access cluster are shown in green and purple. The same strategy is valid for larger grid memories and access clusters. For this simplified example, the $2 \times 2 \times 2$ access cluster can support

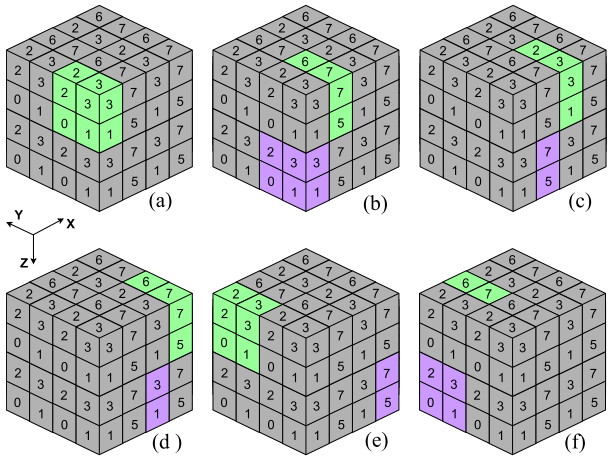

Fig. 10. Grid memory access cluster staggering for the FFT phase.

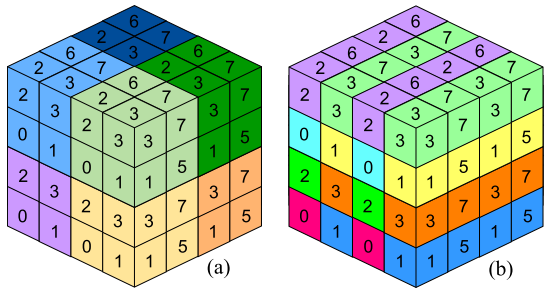

Fig. 11. Mapping of grid points to FFT pipelines.

8 FFT pipelines. Fig. 11 details how the grid points are mapped to FFT pipelines. In Fig. 11(a) colors represent grid neighborhoods. Different *shades* of the same color represent neighborhoods that belong to the same district and are accessed using a 2D slice of the access cluster. For this example configuration, the green and blue districts are accessed using one 2D slice of the access cluster, while the orange and purple districts are accessed using the other slice. Different colors in Fig. 11(b) depict which grid points are sent to each of the 8 FFT units when performing the 1D FFTs along X dimension. In this configuration, each of the FFT pipelines performs two 1D FFTs per dimension.

Latency of 3D FFT calculations is deterministic. For simplicity, we assume the simulated space to be a perfect cube. If we consider an $N \times N \times N$ charge grid, the 3D FFT is decomposed into three sets of 1D FFTs. Each set comprises N^2 N-point FFTs. We denote the number of FFT pipelines by P_{fft}, the pipeline depth by D_{fft}, and access latencies for read and write grid memory accesses by M_R and M_W, respectively. Latency for one set of 1D FFTs is then:

$$\frac{N^3}{P_{fft}} + D_{fft} + M_R + M_W \tag{12}$$

For a cubic charge grid, N^3 is the number of grid points which we denote as N_g. The M_R term is for reading the first grid point from the grid memory and the M_W term is writing back the last term of the FFT output to the grid memory. It is necessary to

ensure that the results of one FFT phase is completely written back to the grid memory before starting the next FFT phase. The total latency for the 3D FFT or inverse FFT is three times the latency of a single phase. In this implementation, the third phase of the forward FFT takes three additional cycles since the outputs of the FFT pipelines are multiplied by the Green's function values before they are written back to the grid memory.

E. Force Computation Pipeline

Force computation can be broken into two stages: coefficient generation and multiply-accumulate (MAC). Particle positions are read from particle information memory and sent to three force coefficient generator units. Force coefficient generators apply the partial derivatives of the basis function along each dimension to a particle location to generate 64 coefficients corresponding to the 64 neighbor grid points.

In the MAC portion of the force computation pipeline, the 64-point neighbor subgrid is read from grid memory and potential values at grid points are multiplied by the coefficients calculated by the coefficient generator module. Up to this point, the force pipeline is also a 64-way vectorized implementation. Next, the multiplication results are summed over all 64 points. Three independent adder trees are used to reduce the 64 values to a single force value along each of the X, Y, and Z dimensions.

1) Parallel Force Computation: The force computation phase could be fully parallelized since computing the force exerted on one particle by the potential field does not depend on other particles. The limitation is reading the grid points from the clustered grid memory. The baseline implementation allows 64 grid points to be read from the grid memory per cycle, which translates to a force computation throughput of one particle per cycle. However, when we use the alternative implementation of the grid memory to enable parallel charge mapping, we also enable parallel force computation. Each of the subgrid memories, which map to different regions of the simulation space, provides 64 ports allowing multiple force computation pipelines to operate on particles from different regions. Similar to parallel charge mapping, boundary region particles are sent to multiple pipelines during force computation.

2) Particle Information Memory: The baseline architecture uses a single memory unit to cache the particle information until used in force computation. However, parallel force computation requires each of the force pipelines to have access to the particles from different regions in parallel. This requirement is satisfied by using multiple particle information memories, each assigned to a different region in the simulation space. Since the particles from different regions are handled by separate pipelines at the charge mapping stage, caching the particle information in separate memory units is trivial.

3) Force Computation Latency Model: We can derive an analytical formula for the force computation latency similar to charge mapping. The effect of boundary region particles is the same. We denote the boundary region particles per pipeline as $N_{B\ fcalc}$. The relationship between $N_{B\ fcalc}$ and N_p is specific to a given configuration. The force computation latency is given by:

$$\frac{N_p}{P_{fcalc}} + N_{B\ fcalc} + D_{fcalc} + M_R \tag{13}$$

where P_{fcalc} is the number of force pipelines and D_{fcalc} is the pipeline depth.

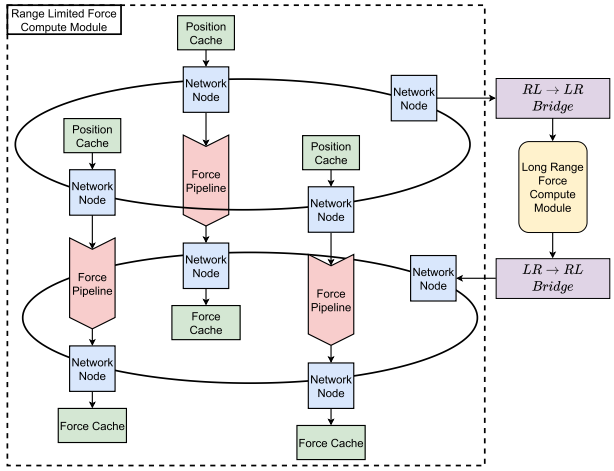

Fig. 12. Connectivity between LR and RL units.

F. Connecting Long-Range and Range-Limited Units

This section extends previous work [23] with multiple optimizations. Since then the Range-Limited (RL) design has gone through a major update [36]. Due to these changes, and to support the particle scheduling algorithm described in Section III-C, the logic connecting the LR and RL components was also redesigned.

The RL unit stores particle information in *position caches* and uses ring interconnects to transfer particle information. The *input ring* transfers data from position caches to the RL pipelines, while the *output ring* transfers the outputs of the RL pipelines to the force caches, which store the computed force values until used for motion integration. LR needs to capture particle information from the input ring and inject the calculated long-range force values to the output ring to be written to the force caches. Fig. 12 shows how the LR unit is connected to the RL ring interconnects via a pair of bridge modules. The bridge module connected to the input ring also implements the particle scheduling algorithm described in Section III-C.

1) Input Bridge: Fig. 13 illustrates the internal architecture of the input bridge module. It uses double buffering to schedule a set of particles while capturing the next set of particles from the input ring. The bridge module and the RL unit's control logic orchestrate particle data movement between RL and LR units. During each data movement phase, one particle from each position cache in the RL unit is read and injected into the input ring. The bridge module captures the data and stores it in one of the buffers. It then starts scheduling the particles in the first buffer (active) while storing the next set of particle data, received over the input ring, in the second buffer (inactive). The roles of the two buffers are reversed in each phase. To write data to the

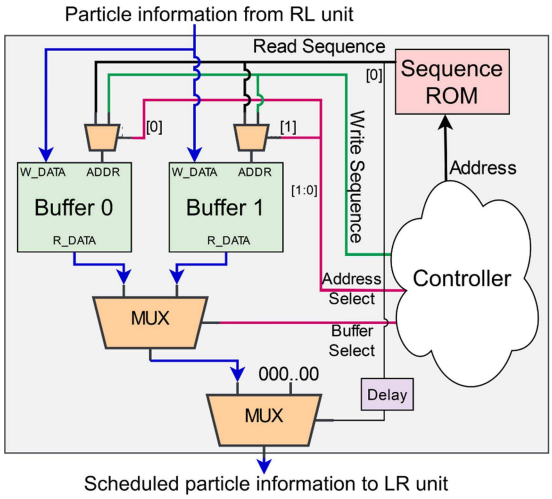

Fig. 13. RL to LR bridge.

inactive buffer, the controller asserts the corresponding *address select* bit. The same signal is used as the write enable for that buffer. The controller also issues the write address sequence.

To schedule particles in the active buffer, the controller first issues the read addresses for the *sequence ROM*. The address sequence to read particles without violating the scheduling constraints is pre-calculated and stored in the *Sequence ROM*; it indicates the cell from which the next particle should be chosen. Readout from the ROM is used as the read address for the active buffer. Each entry in the ROM has an extra bit to indicate whether it is a valid address or a pipeline bubble. This bit is routed to the select input of the output multiplexer.

2) Output Bridge: The output bridge is responsible for sending force values corresponding to each of the particles to the correct force caches, which are connected to the RL output ring, and so is itself connected (see Fig. 12). The LR unit outputs the force values in the same order as it receives the particle information. Therefore the output bridge uses the same sequence ROM as the input bridge to determine the destination addresses for the packets carrying force information to the force caches. But since LR only outputs valid force values, the ROM entries here are always valid.

IV. Baseline Results

In this section, we establish resource usage and performance baselines for the FPGA-LR design without parallel charge mapping or parallel force computation. The baseline design consists of a charge mapping pipeline, sixty-four FFT pipelines, and a force computation pipeline.

The LR force computation unit was validated and performance and resources were measured with implementations on an Intel Stratix 10 1SX280HN2F43E2VG FPGA [44] (Intel D5005 programmable acceleration card) and an Agilex AGFB027R31C3I3V device [45]. The FPGA-LR design was implemented using SystemVerilog. The Intel Open Programmable Acceleration Engine flow was used for the D5005 implementation, while the standard Quartus synthesis and Place & Route flows were used for the Agilex implementation. Intel Quartus Prime Pro - Version 21.2.0 was used. The results reported are for the Agilex. The LR unit was implemented as a standalone accelerator. An application running on the host processor was used to send particle information to the FPGA, receive computed force values, and validate the results from the FPGA against force values calculated on the host processor.

TABLE II
Resource Usage on an Agilex AGFB027R31C3I3V Device

Grid Size	ALM	DSP	BRAM
$16\times16\times16$	102,178 (11%)	1,869 (22%)	148 (1%)
$32\times32\times32$	128,926 (14%)	2,253 (26%)	352 (3%)
$64\times64\times64$	152,661 (17%)	2,381 (28%)	1,728 (13%)

A. Resource Usage

Table II presents the absolute and percent usage of Adaptive Logic Modules (ALM), Digital Signal Processing (DSP) blocks, and block RAMs (BRAMs) for the different design sizes. For the first two grid sizes, the particle information memory was set to 4096 and 32768 respectively which is the same as the number of grid points in the grid memory. For the $64 \times 64 \times 64$ grid size, which better represents the problem sizes of interest, particle memory size was set to 65536.

ALM, DSP, and BRAM usage increases with the design size. Since the baseline configuration of the LR architecture is used, the numbers of charge mapping and force computation pipelines do not change with the size of the charge grid. Therefore any change in DSP usage is attributable to increased usage by the FFT IP cores. For ALMs the larger grid memory also contributes to the increased usage. However, the majority of the additional ALMs are still used by the larger FFT cores.

The Green's ROM might use ALMs or BRAMs depending on the design size. The FPGA tools opt to implement the 16^3 Green's ROM using ALMs, but implement the other two sizes using BRAMs. The size of the clustered grid memory increases by eight times going from one design size to the next. However, the BRAM usage increases by less than that. This can be explained by the fact that although the grid size increases, the memory interleaving remains at 64 and only the size of individual memory banks increases. Therefore, even if the total BRAM bits used increases by $8\times$, the number of BRAMs grows at a lower rate.

Fig. 14 shows the breakdown of ALM usage by different parts of the design. Usage is reported for the three phases of the PME algorithm: charge mapping, FFT computations, and force computation. Components that are used in more than one phase, such as particle memory and clustered grid memory, as well as any logic that cannot be assigned to a particular PME phase (e.g., the finite state machine (FSM) controlling the whole design), are reported separately. Similarly, Figs. 15 and 16 similarly present the DSP slice and RAM block usage. The clustered bars represent the resources as absolute values while the pie charts present the resource usage as a fraction of the total resources. Note that Figs. 14 and 16 use log scales on the X-axis.

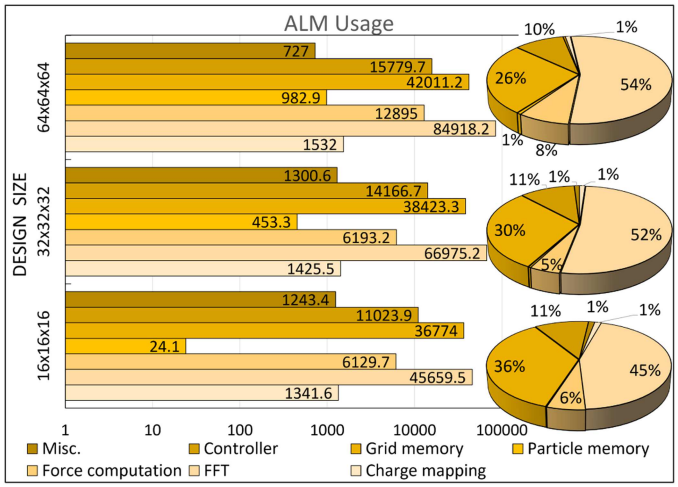

Fig. 14. ALM usage.

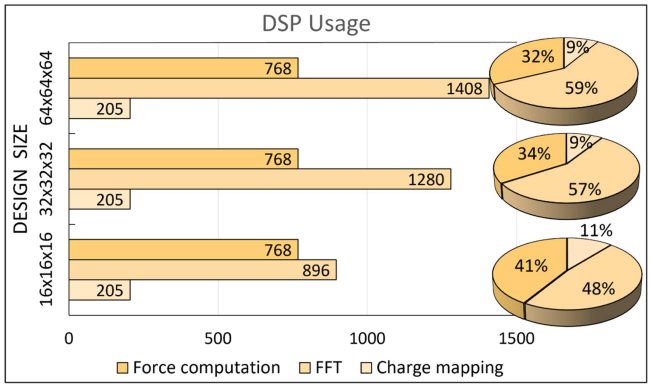

Fig. 15. DSP usage.

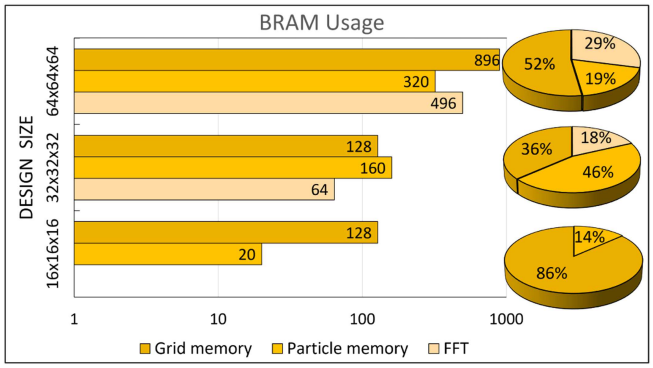

Fig. 16. BRAM usage.

The resource usage figures so far do not include the resources for the input and output bridge modules because they are only used in the fully integrated mode and not standalone. Table III presents the resource usage for the bridge modules for different design sizes: these are less than 1% of available resources on a Stratix 10 or Agilex FPGA. The sequence ROMs and buffers are the components most sensitive to the size of the simulation space. A larger simulation space means a higher number of cells in the RL unit and larger buffers to hold particle information until they are scheduled and sent to the LR unit. Higher RL cell count also means a longer address sequence, which increases the size of the sequence ROM. Since the sequence ROM used in the output bridge does not hold invalid entries, the ROM is very small and the synthesis tool opts to implement it using LUTs rather than BRAMs. Other elements of the bridge modules are less sensitive to simulation size and their resource usage does not vary significantly for different design sizes.

TABLE III
RESOURCE USAGE FOR BRIDGE MODULES

Grid Size	Input Bridge		Output Bridge
	ALM	BRAM	ALM
$16\times16\times16$	290	8	62
$32\times32\times32$	309	8	108
$64\times64\times64$	776	26	559

TABLE IV
BASELINE ACCELERATOR PERFORMANCE AND FMAX

Grid Size	Particles	Fmax	Performance
16^3	4096	254MHz	1620 ns/day (213μs per iteration)
32^3	32768	260MHz	1253 ns/day (276μs per iteration)
64^3	65536	251MHz	539 ns/day (642μs per iteration)

B. Accelerator Performance

The aim of offloading MD to FPGAs is enabling longer simulations. A common metric is the time simulated in a day of wall-clock time. For the three design sizes, we set the number of particles to 4096, 32768, and 65536 respectively. We also assume the LR unit is used in the fully integrated mode, where particle information and force values for each iteration are exchanged with the integrated RL unit and not the host processor. The RL unit could be implemented on the same FPGA or on a different FPGA connected directly to ensure that particles are transferred between RL and LR units at a rate as close as possible to one particle per cycle. We have considered a standard iteration time of 4 femtoseconds. Finally, we assume that the LR unit is what bottlenecks the full simulation. Table IV presents the maximum operating frequency (Fmax) and overall performance for LR units with different grid sizes. Each design is implemented using all the optimization modes available in Intel Quartus Prime and the highest Fmax value is reported. Each optimization mode uses different Place & Route strategies, which results in different critical paths and hence different Fmax values.

The effect of the differences in pipeline utilization due to the particle scheduling algorithm is clearly observable in these results. The charge grid for the 32^3 design has eight times more points than the 16^3 design and simulates eight times more particles. Similarly, the charge grid for the 64^3 design is eight times larger than the 32^3 design. However, it only simulates twice as many particles. Therefore, one might expect the performance to decrease at a faster rate between the 16^3 and 32^3 designs than between 32^3 and 64^3 designs; however, the

reverse is true. This is attributable to the low pipeline utilization (8.3%) of the 16^3 design and very high pipeline utilization (95% and 99.4%) of the other two designs. Between the 16^3 and 32^3 designs, the performance loss due to simulating a larger charge grid and a higher number of particles is partially offset by the higher pipeline utilization of the 32^3 design. Pipeline utilizations between the two larger designs are much closer to each other. Therefore, the effect of larger grid size and particle count is more noticeable.

Of note is that the three designs have similar Fmax values despite the different grid sizes. The parts of the design that directly correlate to the charge grid size are the clustered grid memory and FFT pipelines. The crossbars used to route requests to BRAMs within the grid memory contain the longest combinational paths that determine the Fmax. Although the larger grid sizes use more BRAMs, the addressing logic and crossbars (see Section III-A2) remain the same since the memory interleaving is selected based on the interpolation scheme used and not the grid size. While routing signals to the additional BRAMs does incur additional overhead as the design size grows, it does not significantly impact the critical path and therefore Fmax. The smallest 16^3 design has a slightly lower Fmax than the larger 32^3 because the FPGA tools opt to implement some of the smaller memories (the *particle information memory* and the *Green's ROM*) using ALMs instead of BRAMs.

V. Resource and Performance Optimization

Once the LR components are individually optimized (Section III), there remains the task of optimizing them jointly. Each major component can be either replicated or folded as needed to ensure throughput matching between phases. In this section, we present an overall performance model, describe possible resource tradeoffs, and then discuss the performance of alternative configurations, first at a high level and then in depth. Finally, we compare the performance for FPGA-LR to that achievable on high-end GPUs running a production-grade MD simulation package.

A. Modeling LR Accelerator Performance

To create an overall performance model, we combine the formulae derived in Section III-B, III-D, and III-E. Including grid memory initialization, the latency is given by

$$T_{PME} = \frac{1}{U_{cmap}}\left(\frac{N_p}{P_{cmap}} + N_{B\,cmap}\right) + \frac{6N_g}{P_{fft}} + \frac{N_p}{P_{fcalc}} + N_{B\,fcalc} + \frac{N_g}{64} + D_{cmap} + 6D_{fft} + D_{fcalc} + 8M_R + 7M_W \quad (14)$$

For the baseline configuration without parallel charge mapping or force computation, the above equation reduces to

$$T_{PME} = \frac{N_p}{U_{cmap}} + \frac{6N_g}{P_{fft}} + N_p + \frac{N_g}{64} + D_{cmap} + 6D_{fft} + D_{fcalc} + 8M_R + 7M_W \quad (15)$$

Based on this formulation, we can predict the performance for different configurations and determine optimal resource allocation strategies. We can also evaluate the performance of implementations against the predictions. When comparing the compute cycles per iteration for the three design sizes of the baseline configuration reported in Table IV, the difference between the prediction and the actual implementation is less than 0.4%, 0.1%, and 0.01% of the prediction for 16^3, 32^3, and 64^3 designs, respectively.

B. Module-Level Resource Utilization

This subsection has two purposes. First, we refine the component resource usages from Section IV-A to account for the additional resources needed to create a complete MD design from the individual components. These resources include those necessary for parallelization within a phase, for coupling these now parallelized phases to create a complete LR design, and for sharing resources among phases. And second, we find the single resource type that limits the number of compute units that can fit within a given resource budget thereby simplifying further evaluation.

Table V presents resource usage for each type of pipeline and memory unit at different problem sizes, both as absolute values and as a percentage of resources available on an Intel Agilex AGFB027R31C3I3V device. DHFR and ApoA1 benchmarks are chosen to represent the problem sizes of interest for this work. These use 64^3 and 128^3 charge grids, respectively, to achieve Ewald tolerance values comparable with tolerances used to generate GPU benchmark results in [29]. As discussed in Section III-B2, parallel charge mapping and force computation require a grid memory made of multiple smaller grid memory units. Therefore, the resource usage for the grid memory unit is presented at both the lowest and highest levels of parallelization.

The central observation from Table V is that DSPs are the limiting resource; ALM and BRAM usage by the pipelines is negligible. Force computation pipelines are the most resource-hungry at 9% of available DSPs per pipeline. When we consider the highest level of parallelism for charge mapping and FFT stages, as dictated by the particle scheduling efficiency and maximum read bandwidth of the grid memory, each of the charge mapping and FFT stages could also use between 16% and 20% of the available DSP blocks. Because the three PME phases are in series, the charge mapping and force computation units can share the coefficient generation units and so reduce the overall DSP usage.

The memory units show higher ALM and BRAM usage compared to the pipelines. However, even at the larger problem size and at the highest level of parallelism (supporting 8-way parallelism in charge mapping and force computation), the ALM and BRAM usage by the grid memory is at most 43% and 54% of the available resources. The Green's ROM also takes up ~17% of the BRAMs. Still, the total BRAM usage is well within the available resources. Therefore, for the rest of this section, we explore alternative configurations with respect to DSP usage only, as it alone limits the number of parallel pipelines.

TABLE V
RESOURCE USAGE FOR EACH TYPE OF PIPELINE AND MEMORY UNIT ON AGILEX AGFB027R31C3I3V DEVICE

Component	DHFR			ApoA1		
	ALM	DSP	BRAM	ALM	DSP	BRAM
Charge mapping (1 particle, 64 grid points per cycle)	1562 (0.17%)	205 (2.4%)	0	1585.3 (0.17%)	205 (2.4%)	0
FFT (single 1D FFT unit)	1297 (0.14%)	22 (0.26%)	2 (0.02%)	1652.4 (0.18%)	28 (0.33%)	6 (0.05%)
Force computation (1 particle, 64 grid points per cycle)	12895 (1.41%)	768 (9.0%)	0	12626.5 (1.38%)	768 (9.0%)	0
Green's ROM	5965 (0.65%)	0	368 (2.77%)	7516 (0.82%)	0	2208 (16.64%)
Grid memory (64 x 1 grid points per cycle)	42011 (4.6%)	0	896 (6.75%)	46553 (5.1%)	0	7168 (54.0%)
Grid mem.at full parallelization (64 x 8 grid points per cycle)	328372 (35.9%)	0	1024 (7.72%)	396220 (43.41%)	0	7168 (54.0%)

Units omitted from Table V are the particle memory and the control logic responsible for managing the entire design. For simplicity, and because BRAMs are not a limiting resource, we have set the size of the particle memory to 100,000 particles to be able to hold any problem size of interest. This uses less than 5% and 0.3% of available BRAMs and ALMs, respectively. The control logic uses <2% of available ALMs. An observation is how the number of BRAMs used by the two grid memory configurations differs for the smaller problem size, even when the amount of data to be stored is the same. At full parallelization, grid memory has 8 smaller grid memory units, while each grid memory unit has 64 memory banks. The FPGA tools are able to pack the BRAM bits into BRAMs more efficiently when the amount of memory per memory bank is higher. In the case of the smaller problem size, this effect is visible in the differently sized grid memory units.

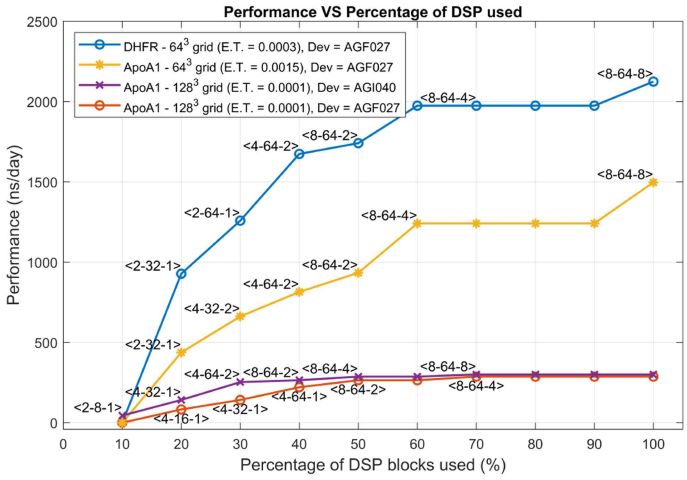

Fig. 17. Performance VS DSP usage (E.T. = Ewald Tolerance, Dev = Target device).

C. LR Configuration Alternatives

In this subsection we find the best LR design alternatives as a function of resource usage. This is a critical measure because it allows extrapolation to variously sized FPGAs as well as LR budgets within larger MD designs. Because of the large design space it is impractical to synthesize all designs of possible interest. We therefore augment the synthesized designs with the model proposed in Section V-A (which was validated to within 0.4%).

The design space consists of integral copies of the three major components up to 100% resource usage. Fig. 17 presents the performance for the two benchmarks at different levels of resource usage ranging from 10% to 100% of the DSPs available. All configurations with 1, 2, 4, or 8 charge mapping and force computation pipelines and 1-64 (powers of two) FFT pipelines are considered. The design space explored is limited by: i) problem sizes of interest, ii) the level of parallelism dictated by the scheduling algorithm efficiency for a given problem size, iii) maximum read bandwidth of the grid memory for the FFT phase, and iv) FPGA resource availability. The best pipeline configuration at each level is shown in the following format: $<Charge\_mapping-FFT-Force\_compute>$.

DHFR shows the highest throughput because of the smaller particle count and the resulting smaller charge grid. The other three lines on Fig. 17 represent three different setups for the ApoA1 benchmark. The configuration corresponding to the *red* line uses the Agilex F-series AGFB027R31C3I3V used in the prior analysis. It uses a 128^3 charge grid and achieves an Ewald tolerance value of 0.0001, which is much lower than the 0.0005 used for GPU benchmark results.

Note that the largest configuration on the *red* line is $<8-64-4>$. The number of DSPs on the Agilex AGFB027R31C3I3V device is not sufficient to implement the <8–64–8> configuration for the larger problem size. This is because of the 128-point FFT units now needed.

The *purple* line shows the performance for the same benchmark at the same tolerance level, but for a larger Agilex I-series FPGA that has 50% more DSPs. While this device is able to accommodate the $<8-64-8>$ pipeline configuration, the incremental benefit is marginal. This is because the simulation time is dominated by the FFT phase due to the larger charge grid. Therefore, the benefit of increasing force compute pipelines is minimal. In contrast, DHFR shows noticeable performance improvements from increasing force pipelines because the force compute phase is more prominent compared to the FFT stage due to the smaller charge grid.

The *yellow* line shows how accuracy can be traded off for higher performance. This setup uses the same device as the *red* line. However, it targets a higher tolerance of 0.0015 and therefore uses a 64^3 charge grid. This shows trends similar to DHFR because the same charge grid size is used. Finally, note that a lower operating frequency of 225 MHz was used with the

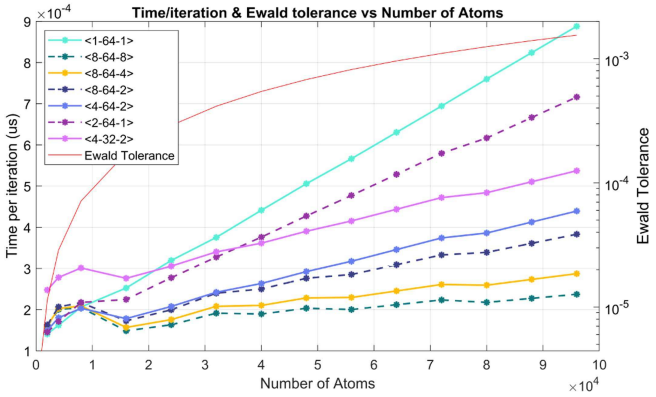

Fig. 18. Performance for alternative configurations.

two configurations using the 128^3 grid memory as opposed to the Fmax value of 251 MHz used for the other two configurations.

Fig. 18 presents performance for the six best configurations from Fig. 17 and the baseline configuration for a range of simulation sizes. A 64^3 charge grid is used for all problem sizes. The second Y-axis shows the Ewald tolerance values corresponding to the problem size and the selected grid size. For a given grid size, the latency for the FFT phase does not change with the number of atoms. The intersection with the Y-axis represents the FFT latency for the different configurations. The $<4-32-4>$ configuration has the highest offset from zero along the Y-axis because it has the fewest FFT pipelines. The latency for the baseline configuration grows linearly with the number of atoms as it has no parallelization in charge mapping and force computation stages. The others grow at a lower rate due to parallel charge mapping and force computation.

The sudden drop in iteration time between ~8K and ~16K atoms corresponds to the particle scheduling algorithm starting to take effect (as shown in Fig. 9). Pipeline utilization increases with simulation size up to around 80K atoms where it reaches close to 100%. From there onward, the iteration time grows linearly with the number of atoms.

D. Discussion

In this subsection, we present five observations about the relationship of performance (for a given problem size and resource budget) with design features.

First, at smaller resource budgets, it is always most beneficial to increase the number of charge mapping and FFT pipelines, rather than force compute pipelines, as more of those will fit within the same resource budget.

Second, between charge mapping and force computation phases, it is easier to achieve higher performance in the force compute phase as the latter do not encounter RAW hazards and so can always operate at 100% efficiency.

Third, unlike the other two phases, for charge mapping the performance benefit diminishes with the the number of pipelines. This is because smaller regions result in less efficient scheduling of particles.

TABLE VI
GPU BENCHMARK RESULTS FROM OUR EXPERIMENTS FOR LR ONLY

GPU	Architecture	DHFR (ns/day)	ApoA1 (ns/day)
RTX 8000	Turing	1150	332

Fourth, the optimal configuration for a given resource budget depends heavily on the problem size. This is especially true for smaller resource budgets. Consider, e.g., the *blue* and *yellow* lines from Fig. 17 at 30% resource usage. Although both configurations use a 64^3 charge grid, the number of FFT units is different. For ApoA1, it is more beneficial to reallocate resources from the FFT phase to the other two phases. This is because the higher particle count of ApoA1 results in longer charge mapping and force compute phases compared to the DHFR benchmark.

And fifth, there is a range of problem sizes that fit a given grid size. As a design guideline, one should allocate more resources to the FFT pipelines at the lower end of the range of problem sizes for a particular grid size, and gradually reallocate resources to charge mapping and force computation when moving through the range of problem sizes. The two lines in Fig. 18, corresponding to two the configurations from above, intersect at around 35K atoms. This demonstrates how the resource allocation strategy changes with the problem size.

E. Performance Comparisons

A likely deployment scenario for FPGA-based LR accelerators is within an FPGA-based MD accelerator, which may itself have scalability advantages (e.g., [39]). Our performance goal for FPGA-based LR accelerators is therefore to demonstrate similarity with other leading accelerators. In this subsection, we compare the performance of the proposed FPGA-based LR accelerator with that of GPUs running OpenMM [42]. We consider two popular MD benchmarks chosen to represent the simulation sizes targeted by this work: Dihydrofolate Reductase (DHFR) and Apolipoprotein A1 (ApoA1).

GPU performance results from our experiments using an Nvidia Quadro RTX 8000 GPU are presented in Table VI. We have modified the OpenMM code to compute only the LR interactions. For the GPU runs, the default Ewald tolerance value of 0.0005 was used. Recall that the best FPGA-LR performance for DHFR is $2124\ ns/day$, with a lower Ewald tolerance value of 0.0003. For ApoA1 the best FPGA-LR performance is 287 and $1496.7\ ns/day$, with Ewald tolerance values of 0.0001 and 0.0015, respectively. Because the FPGA design only uses powers of two charge grid sizes, it cannot exactly match the tolerance value of 0.0005 targeted by the GPU implementations. The FFT IP cores used in this work accept inputs and generate outputs with powers of two points. In order to store the intermediate results of the 3D FFTs, the grid memory also needs to have dimensions which is a power of two. Because the FPGA-LR design reuses the grid memory across all phases of the LR force computation, we cannot save any resources by using smaller non-power of two grid sizes that match the GPU implementations for the charge mapping phase.

TABLE VII
GPU OPENMM BENCHMARK RESULTS (DOWNLOADED 2/28/2024 [29])

GPU	Architecture	DHFR (ns/day)	ApoA1 (ns/day)
RTX 3090	Ampere	1530	425
RTX 4080	Ada Lovelace	2034	607
RTX 4090	Ada Lovelace	2250	785
A100	Ampere	1276	445
H100	Hopper	1502	609

To provide additional context for the performance comparisons, in Table VII we also report benchmark results from a third party [29] for OpenMM on more recent GPU architectures which use more advanced process nodes compared to both the GPU and the FPGA used in our testing. Unlike in our own experiments where LR was isolated, these are for complete MD. Even if we assume a significant performance improvement of $\sim$25% for LR-only execution on more recent GPU architectures, which far exceeds the performance improvement we observed in our testing with an older GPU, FPGA-LR is still competitive with most of the recent GPUs except for few of the highest performing ones.

VI. PRIOR WORK

Previous studies present both FPGA-, and ASIC-based accelerators for parts of, or full MD simulations. FPGA-based FFT accelerators include single FPGA [21], [46], [47], and multi-FPGA [18], [48], [49] implementations. While most of these implementations use custom pipelines for FFT calculations, there are also implementations which make use of soft processors instantiated on FPGAs [50]. Recent work has also focused on improving the usability of FFT accelerators by using OpenCL implementations [51]. [52] provides an OpenCL library for FPGA-based FFT acceleration. OpenCL host code in the form of FFTW-like APIs, which can be used to offload existing FFT routines to FPGAs, and OpenCL kernels that can be synthesized to bitstreams are provided. [53] extends this to provide an OpenCL library for FFT-based 3D convolutions on FPGAs. [54] surveys the design space for offloading 3D FFT calculations to FPGAs. Some studies use coarse-grained reconfigurable array (CGRA) architectures to accelerate FFT computations [55], [56]. These are intended to be used as FPGA or ASIC implementations. There are also non-CGRA FFT architectures intended for ASIC implementations [57].

FPGA-based charge mapping acceleration has been studied in [22]. While not targeting MD applications, there are other works such as [58], [59] focused on FPGA acceleration of cubic interpolation which is a crucial part of the particle-grid mapping.

Going beyond accelerating components of MD calculations, there are accelerators targeting full MD simulations. These accelerators use different hardware configurations and application mapping strategies. Anton [13], [14], [15] is a full ASIC system. FFTs were only used in Anton I as Anton II and III systems used the μ-series method for LR force computation. Anton uses special-purpose datapaths for particle-grid mapping while FFT calculation in Anton I and grid-based convolutions in later Anton systems are mapped to general-purpose processors. Still, due to the custom routing fabric, Anton I was able to provide impressive FFT performance [41].

MDGRAPE-4A [60] uses an ASIC+FPGA approach and implements a novel algorithm named tensor-structured multilevel Ewald summation method (TME). This involves performing 3D FFTs and grid convolutions. The FFTs are performed on FPGAs while the convolutions and grid-particle mapping are performed on custom datapaths implemented on the ASIC portion of the system. The previous iterations of MDGRAPE were ASIC systems [61]. There are also FPGA-based full MD simulation accelerators. [23] is a single FPGA design using custom datapaths for all computations. [16] is a multi-FPGA accelerator for LR force computation designed using OpenCL.

VII. CONCLUSION

In this work, we present an FPGA-based long-range electrostatic force computation architecture for use in Molecular Dynamics simulations, in particular, for the long timescales for which FPGA clusters appear to have advantages. This architecture can either be a standalone accelerator used to offload LR computations of an MD simulation package, or used as part of a fully integrated FPGA-based simulator. We provide detailed architectural descriptions of different components of the accelerator. While doing so, we establish ideal behavior for certain components that optimize the performance and then describe an architecture that satisfies those requirements. We analyze the performance, different resource allocation strategies, and optimal configurations under different resource constraints. Our performance results show that the best configuration of the FPGA-LR design can achieve throughput values of $2124ns/day$ and $287ns/day$ for the DHFR and ApoA1 benchmarks and with low Ewald tolerance values.

For simulation sizes of 20K–50K particles, the FPGA-LR design can provide performance comparable to even the latest generation of GPUs. For larger problem sizes, which correspond to larger charge grids and larger 3D FFTs, the FPGA is somewhat slower. However, the FPGA-LR performance is again comparable for larger problem sizes if higher, but still likely acceptable, tolerance values can be used.

REFERENCES

[1] E. Tajkhorshid et al., "Large scale simulation of protein mechanics and function," *Adv. Protein Chem.*, vol. 66, pp. 195–247, 2003.
[2] D. Shirvanyants, F. Ding, D. Tsao, S. Ramachandran, and N. V. Dokholyan, "Discrete molecular dynamics: An efficient and versatile simulation method for fine protein characterization," *J. Phys. Chem. B*, vol. 116, no. 29, pp. 8375–8382, 2012.
[3] D. A. Case et al., Amber 2020, 2020.
[4] J. Phillips et al., "Scalable molecular dynamics with NAMD," *J. Comput. Chem.*, vol. 26, pp. 1781–1802, 2005.
[5] D. van der Spoel, E. Lindahl, B. Hess, G. Groenhof, A. Mark, and H. Berendsen, "GROMACS: Fast, flexible, and free," *J. Comput. Chem.*, vol. 26, pp. 1701–1718, 2005.
[6] P. Eastman and V. Pande, "OpenMM: A hardware-independent framework for molecular simulations," *Comput. Sci. Eng.*, vol. 4, pp. 34–39, 2010.
[7] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *J. Comput. Phys.*, vol. 117, no. 1, pp. 1–19, 1995.
[8] Z. Cournia, B. Allen, and W. Sherman, "Relative binding free energy calculations in drug discovery: Recent advances and practical considerations," *J. Chem. Inf. Model.*, vol. 57, pp. 2911–2937, 2017.

[9] Molecular Dynamics (MD) on GPUs, "NVIDIA," 2017. [Online]. Available: https://images.nvidia.com/content/tesla/pdf/Molecular-Dynamics-July-2017-MB-slides.pdf

[10] C. Li, W. Chen, Y. Zhang, and Q. Bai, "Analyses on performance of GROMACS in hybrid MPI+OpenMP+CUDA cluster," in *Proc. Int. Conf. High Perform. Comput. Commun.*, 2014, pp. 904–911.

[11] M. Schaffner and L. Benini, "On the feasibility of FPGA acceleration of molecular dynamics simulations," 2018, *arXiv: 1808.04201*.

[12] Amber20: Pmemd.cuda performance information, Sep. 2021. [Online]. Available: https://ambermd.org/GPUPerformance.php

[13] D.E. Shaw et al., "Anton, A special-purpose machine for molecular dynamics simulation," in *Proc. Int. Symp. Comput. Archit.*, 2007, pp. 1–12.

[14] D.E. Shaw et al., "Anton 2: Raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer," in *Proc. Int. Conf. High Perform. Comput. Netw., Storage Anal.*, 2014, pp. 41–53.

[15] D.E. Shaw et al., "Anton 3: Twenty microseconds of molecular dynamics simulation before lunch," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2021, pp. 1–11.

[16] C. Pascoe, L. Stewart, B. Sherman, V. Sachdeva, and M. Herbordt, "Execution of complete molecular dynamics simulations on multiple FPGAs," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2020, pp. 1–2.

[17] U. Essmann, L. Perera, M. L. Berkowitz, T. Darden, H. Lee, and L. G. Pedersen, "A smooth particle mesh Ewald method," *J. Chem. Phys.*, vol. 103, no. 19, pp. 8577–8593, 1995.

[18] J. Sheng, B. Humphries, H. Zhang, and M. Herbordt, "Design of 3D FFTs with FPGA clusters," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2014, pp. 1–6.

[19] T. VanCourt and M. Herbordt, "Application-specific memory interleaving for FPGA-based grid computations: A general design technique," in *Proc. IEEE Conf. Field Programmable Log. Appl.*, 2006, pp. 1–7.

[20] Y. Gu and M. Herbordt, "FPGA-based multigrid computations for molecular dynamics simulations," in *Proc. IEEE Symp. Field-Programmable Custom Comput. Mach.*, 2007, pp. 117–126.

[21] B. Humphries, H. Zhang, J. Sheng, R. Landaverde, and M. Herbordt, "3D FFTs on a single FPGA," in *Proc. 22nd Int. Symp. Field-Programmable Custom Comput. Mach.*, 2014, pp. 68–71.

[22] A. Sanaullah, A. Khoshparvar, and M. Herbordt, "FPGA-accelerated particle-grid mapping," in *Proc. Int. Symp. Field-Programmable Custom Comput. Mach.s*, 2016, pp. 192–195.

[23] C. Yang et al., "Fully integrated FPGA molecular dynamics simulations," in *Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2019, pp. 1–31.

[24] R. Halver, J. H. Meinke, and G. Sutmann, "Kokkos implementation of an Ewald Coulomb solver and analysis of performance portability," *J. Parallel Distrib. Comput.*, vol. 138, pp. 48–54, 2020.

[25] C. Predescu et al., "The u-series: A separable decomposition for electrostatics computation with improved accuracy," *J. Chem. Phys.*, vol. 152, no. 8, 2020, Art. no. 084113.

[26] R. Skeel, I. Tezcan, and D. Hardy, "Multiple grid methods for classical molecular dynamics," *J. Comput. Chem.*, vol. 23, pp. 673–684, 2002.

[27] C. Sagui and T. Darden, "Multigrid methods for classical molecular dynamics simulations of biomolecules," *J. Chem. Phys.*, vol. 114, pp. 6578–6591, 2001.

[28] M. Aminpour, C. Montemagno, and J. A. Tuszynski, "An overview of molecular modeling for drug discovery with specific illustrative examples of applications," *Molecules*, vol. 24, 2019, Art. no. 1693.

[29] OpenMM Benchmarks, 2023. Accessed: Feb. 28, 2024. [Online]. Available: https://openmm.org/benchmarks

[30] P. L. Freddolino and K. Schulten, "Common structural transitions in explicit-solvent simulations of villin headpiece folding," *Biophysical J.*, vol. 97, no. 8, pp. 2338–2347, 2009.

[31] I. Buch, T. Giorgino, and G. De Fabritiis, "Complete reconstruction of an enzyme-inhibitor binding process by molecular dynamics simulations," *Proc. Nat. Acad. Sci. USA*, vol. 108, no. 25, pp. 10 184–10 189, 2011.

[32] M. M. Rahman et al., "Virtual screening, molecular dynamics and structure–activity relationship studies to identify potent approved drugs for Covid-19 treatment," *J. Biomol. Struct. Dyn.*, vol. 39, no. 16, pp. 6231–6241, 2021.

[33] G. Zhao et al., "Mature HIV-1 capsid structure by cryo-electron microscopy and all-atom molecular dynamics," *Nature*, vol. 497, no. 7451, pp. 643–646, 2013.

[34] E. Villa et al., "Ribosome-induced changes in elongation factor Tu conformation control GTP hydrolysis," *Proc. Nat. Acad. Sci. USA*, vol. 106, no. 4, pp. 1063–1068, 2009.

[35] M. L. Cartron et al., "Integration of energy and electron transfer processes in the photosynthetic membrane of Rhodobacter sphaeroides," *Biochimica et Biophysica Acta (BBA)-Bioenergetics*, vol. 1837, no. 10, pp. 1769–1780, 2014.

[36] C. Wu et al., "Upgrade of FPGA range-limited molecular dynamics to handle hundreds of processors," in *Proc. Int. Symp. Field-Programmable Custom Comput. Mach.*, 2021, pp. 142–151.

[37] C. Wu et al., "Optimized mappings for symmetric range-limited molecular force calculations on FPGAs," in *Proc. 32nd Int. Conf. Field-Programmable Log. Appl.*, 2022, pp. 101–108.

[38] C. Wu, T. Geng, V. Sachdeva, W. Sherman, and M. Herbordt, "A communication-efficient multi-chip design for range-limited molecular dynamics," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2020, pp. 1–8.

[39] C. Wu et al., "FASDA: An FPGA-aided, scalable and distributed accelerator for range-limited molecular dynamics," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2023, pp. 1–14.

[40] C. Wu et al., "FPGA-accelerated range-limited molecular dynamics," *IEEE Trans. Comput.*, vol. 73, no. 6, pp. 1544–1558, Jun. 2024.

[41] C. Young, J. Bank, R. Dror, J. Grossman, J. Salmon, and D. Shaw, "A 32x32x32, spatially distributed 3D FFT in four microseconds on anton," in *Proc. Conf. High Perform. Comput. Netw. Storage Anal.*, 2009, pp. 1–11.

[42] OpenMM, "OpenMM: A high performance molecular dynamics library," Aug. 2021. [Online]. Available: https://github.com/openmm/openmm

[43] M. Chiu and M. Herbordt, "Molecular dynamics simulations on high performance reconfigurable computing systems," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 4, pp. 1–37, 2010.

[44] Intel Stratix 10 Device Datasheet, May 2021. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/documentation/mcn1441092958198.html

[45] Intel Agilex Device Data Sheet, Dec. 2021. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/documentation/fno1550626027274.html

[46] R. Chen, N. Park, and V. K. Prasanna, "High throughput energy efficient parallel FFT architecture on FPGAs," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2013, pp. 1–6.

[47] C.-L. Yu, K. Irick, C. Chakrabarti, and V. Narayanan, "Multidimensional DFT IP generator for FPGA platforms," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 58, no. 4, pp. 755–764, Apr. 2011.

[48] J. Sheng, C. Yang, A. Caulfield, M. Papamichael, and M. Herbordt, "HPC on FPGA clouds: 3D FFTs and implications for molecular dynamics," in *Proc. 27th Int. Conf. Field Programmable Log. Appl.*, 2017, pp. 1–4, doi: 10.23919/FPL.2017.8056853.

[49] L. Stewart, C. Pascoe, E. Davis, B. Sherman, M. Herbordt, and V. Sachdeva, "Particle mesh ewald for molecular dynamics in OpenCL on an FPGA cluster," in *Proc. IEEE Symp. Field Programmable Custom Comput. Mach.*, 2021, pp. 270–270.

[50] P. Wang and J. McAllister, "Streaming elements for FPGA signal and image processing accelerators," *IEEE Trans. Very Large Scale Integration Syst.*, vol. 24, no. 6, pp. 2262–2274, Jun. 2016.

[51] A. Sanaullah and M. Herbordt, "OpenCL for HPC/FPGAs: Case study with 3D FFT," in *Proc. 9th Int. Symp. Highly-Efficient Accel. Reconfigurable Technol.*, 2017, pp. 1–6, doi: 10.1145/3241793.3241800.

[52] A. Ramaswami, T. Kenter, T. D. Kühne, and C. Plessl, FFTFPGA, Oct. 2021. Accessed: Feb. 28, 2024. [Online]. Available: https://github.com/pc2/fft3d-fpga

[53] A. Ramaswami, T. Kenter, T. D. Kühne, and C. Plessl, ConvFPGA, Oct. 2021. Accessed: Feb. 28, 2024. [Online]. Available: https://github.com/pc2/ConvFPGA

[54] A. Ramaswami, T. Kenter, T. D. Kühne, and C. Plessl, "Evaluating the design space for offloading 3D FFT calculations to an FPGA for high-performance computing," in *Proc. Int. Symp. Appl. Reconfigurable Comput.*, Springer, 2021, pp. 285–294.

[55] C. Liang and X. Huang, "Mapping parallel FFT algorithm onto smart-cell coarse-grained reconfigurable architecture," *IEICE Trans. Electron.*, vol. 93, no. 3, pp. 407–415, 2010.

[56] A. Xu and Q. Zhang, "A scalable coarse-grained reconfigurable array based FFT hardware accelerator," in *Proc. IEEE 5th Int. Conf. Electron. Technol.*, 2022, pp. 308–311.

[57] A. X. Glittas, M. Sellathurai, and G. Lakshminarayanan, "A normal I/O order radix-2 FFT architecture to process twin data streams for MIMO," *IEEE Trans. Very Large Scale Integration Syst.*, vol. 24, no. 6, pp. 2402–2406, Jun. 2016.

[58] S. Boukhtache, B. Blaysat, M. Grédiac, and F. Berry, "FPGA-based architecture for bi-cubic interpolation: The best trade-off between precision and hardware resource consumption," *J. Real-Time Image Process.*, vol. 18, no. 3, pp. 901–911, 2021.
[59] J. Koljonen, V. A. Bochko, S. J. Lauronen, and J. T. Alander, "Fast fixed-point bicubic interpolation algorithm on FPGA," in *Proc. Nordic Circuits Syst. Conf. Int. Symp. Syst.-on-Chip*, 2019, pp. 1–7.
[60] G. Morimoto et al., "Hardware acceleration of tensor-structured multilevel ewald summation method on MDGRAPE-4A, a special purpose computer system for molecular dynamics simulations," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2021, pp. 1–15.
[61] M. Taiji et al., "Protein explorer: A petaflops special-purpose computer system for molecular dynamics simulations," in *Proc. ACM/IEEE Conf. Supercomput.*, 2003, Art. no. 15.

Sahan Bandara received the bachelor's degree in electronic and telecommunication engineering from the University of Moratuwa, Sri Lanka, in 2015, and the master's degree in computer engineering from Boston University, in 2019. In the interim between academic degrees, Sahan worked as a corporate application engineer for Synopsys. His research interests are computer architecture, hardware security, and hardware operating systems for reconfigurable hardware.

Anthony Ducimo received the BS degree in electrical and computer engineering from Worcester Polytechnic Institute and the MS degree in computer engineering from Boston University in 2021. Currently, he is a principle design engineer with Intrinsix Corp.

Chunshu Wu received the bachelor's and master's degrees in physics from the Dalian University of Technology, Liaoning, China, in 2016, and from Brown University in 2018. After that, he joined ECE department of Boston University as a graduate student. His research was on FPGA-based acceleration for Molecular Dynamics. He is currrently a postdoctoral researcher at the University of Rochester.

Martin Herbordt is a professor with the Department of Electrical and Computer Engineering, Boston University where he directs the Computer Architecture and Automated Design Lab.