

MAPPING MOLECULAR DYNAMICS TO  
RECONFIGURABLE HARDWARE

René Gabriëls

November 20, 2009

MASTER'S THESIS

TECHNISCHE UNIVERSITEIT EINDHOVEN  
Department of Mathematics and Computer Science

---

**Graduation supervisors:** Prof. dr. ir. C. H. van Berkel  
Dr. R. H. Mak

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Background . . . . .	3
1.2	Thesis outline . . . . .	4
<b>2</b>	<b>Molecular Dynamics</b>	<b>5</b>
2.1	Physical model . . . . .	5
2.2	Numerical integration . . . . .	8
2.3	Simulation environment . . . . .	10
2.4	Implementation . . . . .	11
2.5	Applications . . . . .	14
<b>3</b>	<b>Field Programmable Gate Arrays</b>	<b>16</b>
3.1	Rationale for the use of FPGAs . . . . .	16
3.2	Technology . . . . .	17
3.3	Programming . . . . .	20
3.4	FPGA machines . . . . .	21
<b>4</b>	<b>Problem Statement</b>	<b>24</b>
4.1	Related Work . . . . .	25
<b>5</b>	<b>Architecture</b>	<b>28</b>
5.1	Problem Analysis . . . . .	28
5.2	Storing the state . . . . .	31
5.3	Numerical representation & arithmetic . . . . .	34
5.4	Hierarchy of machines . . . . .	36
<b>6</b>	<b>Hardware simulator</b>	<b>43</b>
6.1	Architecture . . . . .	43
6.2	Uniform particle systems . . . . .	48
6.3	Non-uniform particle systems . . . . .	50
<b>7</b>	<b>Hardware implementation</b>	<b>55</b>
7.1	Particle machine . . . . .	55
7.2	Cell machine . . . . .	60
7.3	MD machine . . . . .	63

<b>8 Conclusion</b>	<b>65</b>
8.1 Further work . . . . .	66
<b>A FPGA arithmetical operations</b>	<b>67</b>

# Chapter 1

## Introduction

Recent advancements in the capacity of FPGAs have enabled running complex algorithms on them. By exploiting the inherent parallelism of these devices, large potential speedups as compared to CPUs can be obtained. This report describes an FPGA implementation of a molecular dynamics algorithm and compares its performance to traditional CPU based implementations.

### 1.1 Background

Scientific simulations are computationally expensive algorithms, not in their inherent algorithmic complexity, but in their very large inputs and long simulation times. As soon as more processing power is made available, scientists simply start doing more elaborate simulations: more runs, more accurate models, more time steps, or larger input systems. Therefore, more processing power will always be put to some use in this field.

Molecular dynamics (MD) simulations is one of the most popular and demanding types of scientific simulations. It computes the trajectory of a set of particles through time by integrating the laws of motion as described by Newton. MD simulations are used in many fields, including material sciences and molecular biology. Especially in molecular biology, larger and larger simulations are ran to understand the processes going on inside living organisms.

Molecular dynamics, as almost all scientific simulations, have traditionally been run on the fastest available computers, which nowadays are largely networked PCs with commodity CPUs. However, CPU performance hasn't kept pace with the advances in semiconductor manufacturing, and is now hitting a performance wall [9]. Making a single CPU faster has now largely been abandoned in favor of providing more CPUs in the same package (multicore CPUs).

Field Programmable Gate Arrays (FPGAs) on the other hand are a class of reconfigurable hardware now challenging the CPU in several fields. FPGAs have been growing rapidly in capacity and speed, and do not face any performance walls yet. They are now big enough to do complex operations involving floating point computations that were previously possible only on CPUs and dedicated hardware (ASICs). Therefore, they have recently been under investigation to perform (parts of) these scientific simulations [11].

## 1.2 Thesis outline

This thesis is divided into chapters as follows:

- Chapter 2 gives a brief introduction to MD simulations: the physics behind it, how the algorithm works, how the results of a simulation must be interpreted, and some state of the art applications.
- Chapter 3 gives a brief introduction to the internals of FPGAs, how it compares to other technologies, how it can be programmed, and what type of FPGA systems can be used.
- Chapter 4 defines the problem that is being solved and gives a summary of the work that has already been done in this field.
- Chapter 5 presents a performance analysis of an MD algorithm, investigates the requirements for numerical precision, and proposes a hardware architecture that maps an MD algorithm to a network of FPGAs.
- Chapter 6 describes a software simulator that has been build to establish the performance of the architecture given a set of parameters, and to make it possible to quickly prototype extensions. Furthermore, it describes the results obtained using this simulator.
- Chapter 7 describes an actual hardware implementation that was build for the Digilent Virtex 2 Pro XUP board and some estimates for larger FPGA based systems.
- Chapter 8 wraps up this report with a conclusion and further work that can be done in this area.

## Chapter 2

# Molecular Dynamics

Physics tells us that everything in nature, even the most complex macroscopic object, is made out of interacting elementary particles. The laws that describe these particle systems are relatively simple. However, in general, it is impossible to predict from these simple laws how any non-trivial particle system is going to evolve. Furthermore, we are typically interested in macroscopic properties of a system, not in what each and every particle is going to do.

By doing an experiment and measuring the proper macroscopic quantities, we can circumvent this problem. However, experiments are often hard to perform when scales of interest (both space and time) are very small, such as nano-meters and nano-seconds.

Computer simulations try to solve this problem, by simulating a particle system's behavior over time. This is done by numerical integration of the physical laws. Macroscopic properties can then be extracted using statistical physics or visualization software. The problem with this approach is that computers inevitably make discretization errors, so one should always carefully check that a simulation makes physical sense.

### 2.1 Physical model

On an elementary level, matter is comprised of particles such as electrons and protons which are governed by the laws of quantum mechanics. However, simulation on this level is computationally expensive for large particle systems. Molecular Dynamics simulations make the simplification that everything is made out of point-mass atoms that are governed by Newton's laws of motion. This makes simulations computationally tractable, but we lose the ability to simulate chemical reactions.

In a particle system of size  $N$ , each particle  $i : 1 \leq i \leq N$  has a position  $\mathbf{r}_i(t)$ , a velocity  $\mathbf{v}_i(t)$ , and a mass  $m_i$ . The Newtonian equations that govern such a particle system are:

$$\frac{d\mathbf{r}_i(t)}{dt} = \mathbf{v}_i(t) \tag{2.1}$$

$$\frac{d\mathbf{v}_i(t)}{dt} = \frac{\mathbf{F}_i(\mathbf{r}_1(t), \dots, \mathbf{r}_n(t))}{m_i} \tag{2.2}$$

where  $\mathbf{F}_i$  is the force exerted on particle  $i$  and can be any function of the particle positions, depending on the type of simulation. Because we have  $N$  particles, and each equation above consists of 3 independent equations for the  $x$ ,  $y$ , and  $z$  coordinates, we have in total a system of  $6N$  differential equations. To find the behavior of a particle system over time (meaning the functions  $\mathbf{r}_i(t)$  and  $\mathbf{v}_i(t)$  for all  $i$ ), the equations have to be solved with a given initial state at  $t = t_0$ . The result is a *trajectory* through  $6N$  dimensional *phase space*, which is the space of all possible states of the particle system.

In general, the force function  $\mathbf{F}_i$  depends on the position of all particles in the system. In molecular dynamics, the following restrictions are placed on the forms  $\mathbf{F}_i$  can take:

1. Forces must be conservative, meaning that the work resulting from a force acting over a distance does not depend on the actual path taken. This allows us to define a potential function  $U(\mathbf{r}_1(t), \dots, \mathbf{r}_n(t))$  with regard to some arbitrary reference point, representing the total potential energy embodied by the particle system. The force on any particle  $i$  can then be rewritten in terms of the potential function:

$$\mathbf{F}_i(\mathbf{r}_1(t), \dots, \mathbf{r}_n(t)) = -\nabla_i U(\mathbf{r}_1(t), \dots, \mathbf{r}_n(t)) \quad (2.3)$$

Furthermore, this restriction results in energy conservation within the system, meaning the sum of the potential and kinetic energy is constant for all  $t$ :

$$\frac{d(U(\mathbf{r}_1(t), \dots, \mathbf{r}_n(t)) + \sum_{i=1}^n \frac{1}{2} m_i \mathbf{v}_i(t)^2)}{dt} = 0 \quad (2.4)$$

2. The potential function  $U$  must be the sum of potentials  $U'$  between particle pairs only, no higher order interactions, such as 3-particle interactions, are taken into account:

$$U(\mathbf{r}_1(t), \dots, \mathbf{r}_n(t)) = \sum_{i=1}^n \sum_{j=i+1}^n U'(\mathbf{r}_i(t), \mathbf{r}_j(t)) \quad (2.5)$$

3. The force must be central, meaning that the potential between a particle pair  $\mathbf{r}_i(t)$  and  $\mathbf{r}_j(t)$  depends only on their distance  $r_{ij}(t) = \|\mathbf{r}_{ij}(t)\| = \|\mathbf{r}_i(t) - \mathbf{r}_j(t)\|$ :

$$U(\mathbf{r}_1(t), \dots, \mathbf{r}_n(t)) = \sum_{i=1}^n \sum_{j=i+1}^n U'(r_{ij}(t)) \quad (2.6)$$

If we now define  $F(r_{ij})$  as  $-\frac{dU'(r_{ij})}{dr_{ij}}$ , then the force on particle  $i$  can be expressed as:

$$\mathbf{F}_i(\mathbf{r}_1(t), \dots, \mathbf{r}_n(t)) = -\nabla_i \sum_{j=1}^n \sum_{k=j+1}^n U'(r_{jk}(t)) = \sum_{j=1, j \neq i}^n F(r_{ij}(t)) \frac{\mathbf{r}_{ij}(t)}{r_{ij}(t)} \quad (2.7)$$

In molecular dynamics simulations, 3 kinds of forces can be taken into account (see figure 2.1 for the corresponding graphs):

**Intra-molecular force** that acts only between atoms that are covalently bound together in an atom. These forces arise because bonds are flexible: the distance between the atoms in the bond can vary as well as the orientation of the atoms with respect to each other. The distance between atoms in a bond can be modeled by a simple harmonic potential function, where  $r_0$  and  $k_{ij}$  are parameters:

$$U^{bond}(r_{ij}(t)) = \frac{1}{2}k_{ij}(r_{ij}(t) - r_0)^2 \quad (2.8)$$

$$F^{bond}(r_{ij}(t)) = -\frac{dU^{bond}(r_{ij}(t))}{dr_{ij}(t)} = -k_{ij}(r_{ij}(t) - r_0) \quad (2.9)$$

When 3 or 4 atoms are connected in a chain, forces resulting from bond angles (3 atoms) and dihedrals (4 atoms) can also be taken into account.

**Van der Waals force** that acts on all particles. This force is a combination of a repulsive force when atoms are too close together due to overlapping electron clouds, and an attractive force when atoms are further away due to dispersion forces resulting from uneven electron distribution. The Van der Waals force can be modeled using the Lennard-Jones potential, where  $\sigma_{ij}$  and  $\epsilon_{ij}$  depend in the type of particles  $i$  and  $j$ :

$$U^{LJ}(r_{ij}(t)) = 4\epsilon_{ij} \left( \left( \frac{\sigma_{ij}}{r_{ij}(t)} \right)^{12} - \left( \frac{\sigma_{ij}}{r_{ij}(t)} \right)^6 \right) \quad (2.10)$$

$$F^{LJ}(r_{ij}(t)) = -\frac{dU^{LJ}(r_{ij}(t))}{dr_{ij}(t)} = \frac{48\epsilon_{ij}}{r_{ij}(t)} \left( \left( \frac{\sigma_{ij}}{r_{ij}(t)} \right)^{12} - \frac{1}{2} \left( \frac{\sigma_{ij}}{r_{ij}(t)} \right)^6 \right) \quad (2.11)$$

The force has 2 components: a repulsive part (the term raised to the power 12) that is dominant when particles are close together, and an attractive part (the term raised to the power 6) that is dominant when particles are far apart.

**Electrostatic force** or Coulomb force that acts on all charged particles, typically ions. This force can be either attractive or repulsive, depending on the signs of the charges. It has a relatively long range (it falls off as  $\frac{1}{r^2}$ ). It is described as follows, where  $q_i$  is the charge of particle  $i$ ,  $q_j$  is the charge of particle  $j$  and  $k_e$  is Coulomb's constant:

$$U^C(r_{ij}(t)) = k_e \frac{q_i q_j}{r_{ij}(t)} \quad (2.12)$$

$$F^C(r_{ij}(t)) = -\frac{dU^C(r_{ij}(t))}{dr_{ij}(t)} = k_e \frac{q_i q_j}{r_{ij}(t)^2} \quad (2.13)$$

The total force exerted on a particle  $i$  by another particle  $j$  is then simply the sum of these individual force components:

$$F(r_{ij}(t)) = F^{bond}(r_{ij}(t)) + F^{LJ}(r_{ij}(t)) + F^C(r_{ij}(t)) \quad (2.14)$$

Note that none of the forces above depends only on the distance of the particles involved in an interaction. The bonded interaction and the Van Der Waals interaction both depend on the types of the particles involved, while the electrostatic force depends on the charge of the

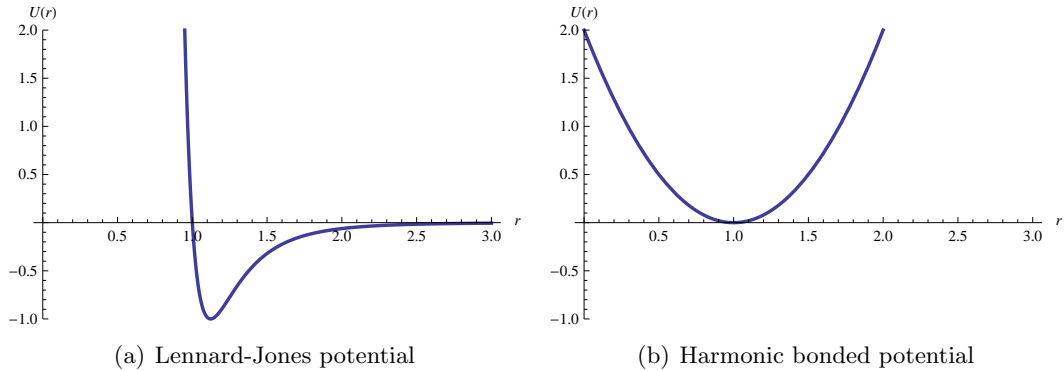


Figure 2.1: Potential between 2 particles at distance  $r$  in reduced units (see section 2.2.2).

particles involved. However, these particle properties don't change, hence it is possible to define a group of force functions, each for a specific pair of particles, each obeying the rules posed above. In this report, we make the restriction that particles are of 1 type only, meaning that in the Lennard-Jones terms, the  $\sigma_{ij}$  and  $\epsilon_{ij}$  terms will be constants, i.e:

$$F_{ij}^{LJ}(r_{ij}(t)) = \frac{48\epsilon}{r_{ij}(t)} \left( \left( \frac{\sigma}{r_{ij}(t)} \right)^{12} - \frac{1}{2} \left( \frac{\sigma}{r_{ij}(t)} \right)^6 \right) \quad (2.15)$$

## 2.2 Numerical integration

To compute a trajectory in phase space, the following set of  $6N$  differential equations needs to be integrated numerically:

$$\frac{d\mathbf{r}_i(t)}{dt} = \mathbf{v}_i(t) \quad (2.16)$$

$$\frac{d\mathbf{v}_i(t)}{dt} = \frac{\mathbf{F}_i(\mathbf{r}_1(t), \dots, \mathbf{r}_n(t))}{m_i} \quad (2.17)$$

Doing this will inevitably result in computational errors. By carefully choosing an integration scheme and a sufficiently small time step  $\Delta t$ , this error can be kept in check. This results in a trajectory through phase space that “shadows” parts of real trajectories and is energetically stable. The longer real trajectories are followed for given  $\Delta t$ , the more stable the integration scheme is. However, there is a trade-off here: taking smaller time steps or choosing a better integration scheme results in more computing steps for the same simulated system. These factors can also be traded for one another: complex integration schemes and large time steps can be as good as simple integration schemes with small time steps.

### 2.2.1 Verlet scheme

The Verlet integration method is a popular method, due to its stability and simplicity. It discretizes the above system of equations as follows:

$$\mathbf{r}_i(t + \Delta t) = 2\mathbf{r}_i(t) - \mathbf{r}_i(t - \Delta t) + \Delta t^2 \frac{\mathbf{F}_i(\mathbf{r}_1(t), \dots, \mathbf{r}_n(t))}{m_i} \quad (2.18)$$

$$\mathbf{v}_i(t + \Delta t) = \mathbf{v}_i(t) + \Delta t \frac{\mathbf{F}_i(\mathbf{r}_1(t + \Delta t), \dots, \mathbf{r}_n(t + \Delta t))}{m_i} \quad (2.19)$$

It uses the current positions  $\mathbf{r}_i(t)$  and past positions  $\mathbf{r}_i(t - \Delta t)$  to compute the next positions  $\mathbf{r}_i(t + \Delta t)$ . Then it uses the next positions to compute the next velocities  $\mathbf{v}_i(t + \Delta t)$ .

An algebraically equivalent but numerically better algorithm is the leap-frog algorithm. It computes velocities on half time steps instead of full steps:

$$\mathbf{r}_i(t + \Delta t) = \mathbf{r}_i(t) + \Delta t \mathbf{v}_i(t + \frac{\Delta t}{2}) \quad (2.20)$$

$$\mathbf{v}_i(t + \frac{\Delta t}{2}) = \mathbf{v}_i(t - \frac{\Delta t}{2}) + \Delta t \frac{\mathbf{F}_i(\mathbf{r}_1(t), \dots, \mathbf{r}_n(t))}{m_i} \quad (2.21)$$

It uses the previous velocities  $\mathbf{v}_i(t - \frac{\Delta t}{2})$  and current positions  $\mathbf{r}_i(t)$  to compute the next velocities  $\mathbf{v}_i(t + \frac{\Delta t}{2})$ . It then uses the next velocities to compute the next positions  $\mathbf{r}_i(t + \Delta t)$ . This is exactly the opposite of the Verlet scheme, but equivalent nonetheless.

In this report we will use the leap-frog scheme. Furthermore, because we only consider particles of 1 type only, the mass of the particles  $m$  is a constant.

### 2.2.2 Reduced Units

In MD simulations, the values of quantities are typically very small if expressed in conventional S.I. units such as meters and seconds. These small numbers are inconvenient to work with and might result in computational errors due to limited numerical representation. Reduced units are chosen such that they are in the order of 1 for MD simulations. They can be chosen freely, but are typically taken out of the Lennard-Jones and Electrostatic function for a certain type of particle. The units are:

Quantity	S.I unit	Reduced unit
length	meter (m)	$\sigma^*$
mass	kilogram (kg)	$m^*$
charge	Coulomb (C)	$q^*$
energy	Joule (J)	$\epsilon^*$

All other units such as time and density can be derived from these units. The exact choice of these units determines their size relationship to the S.I. units.

## 2.3 Simulation environment

Computer experiments, unlike real experiments, are always done in complete isolation. This must be handled correctly for the simulation to give meaningful results. Furthermore, the results of a simulation cannot be measured with an instrument, but rather have to be calculated using some function of the state.

### 2.3.1 Boundary conditions

In the real world, an object under study always interacts with the environment it is situated in via its boundaries. In the simulation there is no environment, so care has to be taken to manage the boundaries correctly. This is typically handled by defining a simulation box and tiling the entire 3-dimensional space with identical images of that box. This technique is known as *periodic boundary conditions*. Because all images are the same, only 1 has to be simulated. If a particle leaves a side of the box, it will enter the box at the opposite side. An alternative is *harmonic boundary conditions*, where particles bounce back into the system with no loss of energy when they bump into the boundary of the simulation box.

With periodic boundary conditions, a particle can interact with images of itself. Therefore, care must be taken to prevent the system to behave in a physically unrealistic way, e.g. when large molecules interact with themselves. In most force computations, only the closest image of a particle is considered, which is referred to as the *minimum image convention*.

### 2.3.2 Ensembles

Real molecular systems typically have interaction with the environment: through heat exchange, through mechanical work and through exchange of particles. The simulated system can be seen on a macroscopic scale as a thermodynamic system that keeps certain values constant, thereby defining the accessible states in phase space. The most common ensembles are:

**Micro-canonical ensemble** The number of particles  $N$ , the volume  $V$  and the total energy  $E$  are kept constant. This means that the temperature and the pressure are allowed to vary. In essence, this is a system in complete isolation.

**Canonical ensemble** The number of particles  $N$ , the volume  $V$  and the temperature  $T$  are kept constant. This is done by putting the system in an infinitely large heat bath, that adds kinetic energy if the system falls below a reference temperature and removes kinetic energy when the system goes above it. Because the heat bath is infinitely large, the simulated system cannot influence its temperature.

**Isothermal-isobaric ensemble** The number of particles  $N$ , the temperature  $T$  and the pressure  $P$  are kept constant. This is done by putting the system in a heat bath and being able to grow or shrink the simulation box to keep the pressure constant. If the pressure is above the reference pressure, the box will expand, if its below the reference pressure, the box will shrink.

**Algorithm** MD( $T, N, \mathbf{L}, \Delta t, \epsilon, \sigma, m, s$ )

```

1. for  $t \leftarrow 1$  to  $T$  do
2.   for  $i \leftarrow 1$  to  $N$  do
3.      $s_i \leftarrow s[t-1][i]$ 
4.      $\mathbf{f}_i \leftarrow \mathbf{0}$ 
5.     for  $j \leftarrow 1$  to  $N$  do
6.       if  $i \neq j$  then
7.          $s_j \leftarrow s[t-1][j]$ 
8.          $\mathbf{r}_{ij} \leftarrow \text{DIST}(s_i.\mathbf{r}, s_j.\mathbf{r}, \mathbf{L})$ 
9.          $r_{ij} \leftarrow \|\mathbf{r}_{ij}\|$ 
10.         $f_{ij} \leftarrow \text{FORCE}(r_{ij}, \epsilon, \sigma)$ 
11.         $\mathbf{f}_{ij} \leftarrow f_{ij} * \mathbf{r}_{ij}/r_{ij}$ 
12.         $\mathbf{f}_i \leftarrow \mathbf{f}_i + \mathbf{f}_{ij}$ 
13.       $s[t][i] \leftarrow \text{INTEGRATE}(s_i, \mathbf{f}_i, m, \Delta t)$ 
14. return  $s$ 

```

Figure 2.2: Brute force  $\mathcal{O}(T * N^2)$  molecular dynamics algorithm.

**Grand canonical ensemble** The volume  $V$ , the temperature  $T$ , and the chemical potential  $\mu$  are kept constant. A heat bath is applied to keep  $T$  constant, and a particle exchange mechanism is applied to keep  $\mu$  constant.

In this report, we will only consider micro-canonical ensembles to keep the algorithm simple.

## 2.4 Implementation

Given a number of time steps  $T$ , a number of particles  $N$ , and an initial state  $s[t]$  at time  $t = 0$  for all particle  $i$ , the algorithm needs to compute the state for all time steps  $t, 1 \leq t \leq T$  and store them in  $s$ . Figure 2.2 shows a very basic implementation. The outer loop iterates over all time steps  $t$ , while the inner 2 loops loop over all pairs of particles  $i$  and  $j$ . For each particle  $i$  the force  $f_i$  acting on it is initialized to zero, then the force between it and every other particle  $j$  is computed and accumulated, and finally this force is used to compute the new position and velocity for particle  $i$ .

Typically, most states  $s[t]$  aren't used after computing state  $s[t+1]$  and can be discarded immediately. If only the final state is of interest, only a current state  $s[t]$  and next state  $s[t+1]$  have to be kept. If some intermediate states are required, e.g. to have a series of snapshots of the system, they can always be saved to external storage. In this report, we will assume only the final state is used.

The algorithm above doesn't specify the DIST, FORCE and INTEGRATE functions. Given the choices we made before, they can be implemented as shown in figure 2.3. Note that the functions INTEGRATE and DIST contain code to compensate for periodic boundary conditions: INTEGRATE keeps particles inside the simulation boundary box defined by the vector  $\mathbf{L}$ , and DIST uses  $\mathbf{L}$  to obtain the minimum image of the particle pair considered.

**Algorithm** DIST( $\mathbf{r}_1, \mathbf{r}_2, \mathbf{L}$ )

1.  $\mathbf{d} \leftarrow \mathbf{r}_1 - \mathbf{r}_2$
2. **for**  $i \in x, y, z$  **do**
3.     **if**  $\mathbf{d}.i > \mathbf{L}.i/2$  **then**
4.          $\mathbf{d}.i \leftarrow \mathbf{d}.i - \mathbf{L}.i$
5.     **else if**  $\mathbf{d}.i < -\mathbf{L}.i/2$  **then**
6.          $\mathbf{d}.i \leftarrow \mathbf{d}.i + \mathbf{L}.i$
7. **return**  $\mathbf{d}$

**Algorithm** FORCE( $r, \epsilon, \sigma$ )

1. **return**  $48 * \epsilon / r * ((\sigma / r)^{12} - 0.5 * (\sigma / r)^6)$

**Algorithm** INTEGRATE( $s, \mathbf{f}, m, \Delta t, \mathbf{L}$ )

1.  $s.\mathbf{v} \leftarrow s.\mathbf{v} + \Delta t * \mathbf{f} / m$
2.  $s.\mathbf{r} \leftarrow (s.\mathbf{r} + \Delta t * s.\mathbf{v}) \bmod \mathbf{L}$
3. **return**  $s$

Figure 2.3: Functions for distance, Lennard-Jones and leap-frog computation.

The complexity of this algorithm is  $\mathcal{O}(T * N^2)$ . Because we want to simulate as fast as possible, we want to reduce complexity as far as possible. Although there are some simple optimizations, such as noting that Newton's third law applies:  $f_{pq} = -f_{qp}$ , these only help by reducing the complexity by a constant factor.

### 2.4.1 Truncated potentials

One of the most popular ways to reduce the complexity of the algorithm is to truncate the potential (setting it to zero) when the distance between 2 particles is larger than some fixed cut-off distance  $r_c$ , meaning that only particles that are close together interact. This truncation results in a slightly lower potential energy of the system. This can be repaired by shifting the potential function  $U$  up by the potential at the cut-off distance. Because the force is the negative derivative of the potential, this constant shift of the potential doesn't influence it, and can thus be ignored in force computation. In case of a uniform particle distribution, the complexity is reduced to  $\mathcal{O}(T * N)$ .

To exploit the fact that particles outside of a certain range don't interact, we need a more elaborate data structure. One of the simplest such data structures is a cubic spatial subdivision, where a number of identical cubic cells partition the simulation box. Each cell has the same size, so we know how many cells in each direction contain particles that can possibly interact with particles of the center cell.

Because particles can interact only with particles falling within a sphere of radius  $r_c$ , the cell based subdivision will consider particles for interaction that turn out to be outside this sphere. The more cells are used, the lower this number of non-interacting particles will be (i.e. approximating a sphere with increasingly smaller cubic boxes), but the higher the bookkeeping costs. For example, when we pick the side of a cubic cell to be  $r_c$ , then only

particles in the same or in neighboring cells (in total that is 27 cells) can possibly interact. In this configuration, the ratio of particles that interact with a given particle to the number of particles that are considered for interacting with is:

$$\frac{\frac{4}{3}\pi r_c^3}{(3r_c)^3} = \frac{\frac{4}{3}\pi}{27} = \frac{4\pi}{81} \approx 0.16 \quad (2.22)$$

Besides making the cells smaller, there are other techniques of bringing this ratio closer to 1. First, a different spatial subdivision can be used that is better able to approximate a sphere, such as a dodecahedron. Another approach is building Verlet lists, which keep track of all particle pairs that are within a range  $r_l$  slightly larger than  $r_c$ . Building the list however is an expensive operation (spatial subdivision can be used to speed it up), so the trick is to choose  $r_l$  sufficiently larger than  $r_c$  such that multiple time steps can be handled with the same Verlet list. The choice of  $r_l$  depends heavily on the speed of the particles and the size of the time step.

In this report, we will only consider the cubic cell based subdivision where the cell edges are at least as large as the cut-off distance. Verlet lists will not be used. Figure 2.4 lists an algorithm that uses a cell based cubic subdivision to speed up computation. It is assumed that  $C$  is the number of cells and that the state of the system  $s$  is now further indexed by cell in between indexation by time step and particle index. Furthermore, the cut-off distance  $r_c$  has been added as a parameter. The function `NEIGHBORHOOD( $a$ )` returns the cells that contain particles that can possibly interact with particles in cell  $a$ , including  $a$  itself. The function `PARTICLES( $a$ )` gives the number of particles in cell  $a$ . The function `CELL( $\mathbf{r}$ )` gives the cell in which position  $r$  falls. For now, we leave them unspecified.

## 2.4.2 Complex algorithms

The Lennard-Jones force, which falls off by  $r^{-7}$  is much more amenable to a cut-off approach than the electrostatic force, which falls off by only  $r^{-2}$ . To compute the electrostatic force and other long range forces efficiently, 3 methods have been invented: Ewald sums, multi-grid methods, and multipole methods. All of these methods result in a running time of less than  $\mathcal{O}(N^2)$ , but have very large constants hidden inside the  $\mathcal{O}$ -notation. Each of these algorithms is quite complex: multigrid methods have a recursive structure, Ewald sums require an FFT implementation to be efficient, and multipole methods require recursive tree structures. Because of the complexity of these algorithms, we will not implement any of them.

On current simulations, computing the Lennard Jones force using a cut-off distance dominates the computation time [28]. This is because the electrostatic force is not computed on every time step (e.g. only every fourth time step), and because the simulations aren't big enough yet to make the larger asymptotic term of the electrostatic force computation dominant. So restricting ourselves to the Lennard-Jones force only doesn't make current simulations significantly faster.

**Algorithm** MD( $T, C, \mathbf{L}, \Delta t, \epsilon, \sigma, r_c, m, s$ )

1. **for**  $t \leftarrow 1$  **to**  $T$  **do**
2.     **for**  $a \leftarrow 1$  **to**  $C$  **do**
3.         **for**  $i \leftarrow 1$  **to** PARTICLES( $s[t-1][a]$ ) **do**
4.              $s_i \leftarrow s[t-1][a][i]$
5.              $\mathbf{f}_i \leftarrow 0$
6.         **for**  $b \in$  NEIGHBORHOOD( $s[t-1][a]$ ) **do**
7.             **for**  $j \leftarrow 1$  **to** PARTICLES( $s[t-1][b]$ ) **do**
8.                  $s_j \leftarrow s[t-1][b][j]$
9.                 **if**  $\neg(a = b \wedge i = j)$  **then**
10.                      $\mathbf{r}_{ij} \leftarrow$  DIST( $s_i.\mathbf{r}, s_j.\mathbf{r}, \mathbf{L}$ )
11.                      $r_{ij} \leftarrow \|\mathbf{r}_{ij}\|$
12.                     **if**  $r_{ij} < r_c$  **then**
13.                          $\mathbf{f}_{ij} \leftarrow$  FORCE( $r_{ij}, \epsilon, \sigma$ )
14.                          $\mathbf{f}_{ij} \leftarrow \mathbf{f}_{ij} * \mathbf{r}_{ij}/r_{ij}$
15.                          $\mathbf{f}_i \leftarrow \mathbf{f}_i + \mathbf{f}_{ij}$
16.              $s'_i \leftarrow$  INTEGRATE( $s_i, \mathbf{f}_i, m, \Delta t, \mathbf{L}$ )
17.              $a' \leftarrow$  CELL( $s'_i.\mathbf{r}$ )
18.              $i' \leftarrow$  PARTICLES( $s[t][a']$ ) + 1
19.              $s[t][a'][i'] \leftarrow s'_i$
20. **return**  $s$

Figure 2.4: Cubic cell based spatial subdivision  $\mathcal{O}(T * N)$  molecular dynamics algorithm

## 2.5 Applications

There are many potential applications of molecular dynamics. Arguably the most challenging and interesting areas are found in molecular biology, which is full of complex and poorly understood systems. The holy grail in this field is to simulate a complete cell in all its facets. The computational power required for this is far outside the reach of current technology. However, building blocks of cells can be simulated nowadays. The following paragraphs will list some state of the art results in this field.

### 2.5.1 Satellite Tobacco Mosaic Virus

An all atom STMV (Satellite Tobacco Mosaic Virus) simulation has been performed on a large cluster using the NAMD molecular dynamics package by Peter L. Freddolino et al. in 2006 [10]. This virus consists of just 2 parts: an RNA molecule with an enclosing capsid around it. The goal was to find out the structural stability of the capsid and RNA alone as compared to the complete virus. It turned out that the capsid collapsed if there was no RNA molecule inside.

The simulation box is a 220x220x220 Å cube with a periodic boundary, in which about 1 million particles are placed. As in many simulations of this kind, the simulation has to take place inside water, so the majority of the atoms in the simulation belong to water molecules. The simulation takes 50 million time steps of 1 fs, meaning a 50 ns simulation.

The simulation is taking all 3 forces into account: bonded interactions, van der Waals interactions with a cut-off distance of 12 Å and Coulombic interactions. An interesting technique that is used is to compute van der Waals forces only every second time step and Coulombic forces every fourth time step. The temperature and pressure are kept constant at respectively 298 K and 1 ATM during the simulation.

Using a cell based subdivision with the side of 1 cell being 12 Å, this results in a total of  $18^3 = 5832$  cells. With 1 million particles in the simulation, this means an average of  $\frac{1,000,000}{5832} = 171.47$  particles per cell.

### 2.5.2 Vesicle fusion & fission

Coarse grained simulations of vesicle fusion and fission have been performed on a cluster using the PumMa molecular dynamics package by A. F. Smeijers, A. J. Markvoort et al. [30] [21]. The goal is to find the mechanisms by which vesicles fuse together and split.

The simulation box that is used is a cube with sides of at least 200 Å, in which about 300.000 particles were placed, mostly water. An interesting property is that this simulation doesn't take atoms as primitives. Instead it takes larger structures such as groups of atoms or entire molecules as primitives. The simulation consists of 5 million time steps, of 24 fs each, meaning a 120 ns simulation.

Only the bonded and van der Waals forces are included in the simulation. The van der Waals force uses a cut-off distance of 11.25 Å. The temperature and pressure is kept constant at respectively 307 K and 1 ATM.

Using a cell based subdivision, this results in  $20^3 = 8000$  cells. With 300,000 particles in the simulation, this means an average of  $\frac{300,000}{8000} = 37.5$  particles per cell.

### 2.5.3 Protein folding

Upon creation, proteins fold from a long string of amino acids into a 3 dimensional structure. The structure is essential for the task the protein has to perform. If the protein misfolds, the protein doesn't work or can even be harmful: many diseases are the result of malformed proteins. The folding of proteins is poorly understood, so molecular dynamics simulations are used.

An interesting project in the field of protein folding is folding@home, where many ordinary computers form a distributed supercomputer, with a claimed performance of over 5 petaflops, which makes it the most powerful computer in the world. To perform the folding simulations, the project uses the GROMACS molecular dynamics package. Another interesting thing to note about this project is that they can exploit the GPU as well as the CPU to perform the simulations.

## Chapter 3

# Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are a class of chips that allow the user to specify the behavior of the chip (i.e. *program* it) after purchasing it, hence “field programmable”. In principle, an FPGA can emulate any digital circuit, as long as it has enough hardware resources available and can meet the timing constraints.

Some FPGAs use fuses to store the program, and can thus be programmed only once, but most chips use SRAM to store the program, and can thus be reprogrammed as many times are desired. In this report, we will focus on the SRAM based FPGAs only, because they are the dominant technology. Xilinx [5] and Altera [1] are the 2 biggest SRAM-based FPGA providers.

### 3.1 Rationale for the use of FPGAs

There are many ways in which a computational problem can be implemented in hardware, each having their own advantages and disadvantages:

**CPU** Write a software program that runs on a general purpose CPU. The advantage of this approach is that writing and modifying the software is easy compared to designing hardware. Furthermore, a CPU can be used to run any kind of application, so it provides maximum flexibility. The disadvantage is the performance penalty that has to be paid for this generality.

**Special purpose CPU** Write a software application for a CPU that has been optimized for the problem at hand. For example DSPs are optimized for signal processing, while GPUs are optimized for rendering 3d graphics. However, these special purpose CPUs exist only for some problem domains. GPUs are now flexible enough to execute more general purpose applications. However, the performance depends on how well the algorithm maps to the architecture of the graphics optimized GPU.

**ASIC** Build an application specific integrated circuit (ASIC), for the problem at hand. This will yield maximum performance for that problem, but it is a completely inflexible

solution. It is also much harder to design and manufacture an ASIC than to write software for a CPU.

**FPGA** Make an FPGA implementation. These chips have very generic resources that can be programmed by the designer to emulate almost any digital circuit. Because of the programmability, it is typically slower than an ASIC by an order of magnitude, but it is much easier and cheaper to design for an FPGA. Compared to a CPU, an FPGA is harder to design for, but can be up to 2 orders of magnitude faster, depending on the problem. In this sense, it sits in between the CPU and the ASIC in both speed and flexibility. This makes them attractive for problems that require more performance than a CPU can provide, but need more flexibility than an ASIC can provide.

**Structured ASIC** Build on top of a partially designed and manufactured chip. These chips provide predefined functionality that anyone needs: NAND-gates, NOR-gates, SRAM, etc. The connectivity between the components has to be specified by the user, after which the chip can be finished. This approach makes ASIC manufacturing easier and cheaper, while sacrificing a bit of performance. It is often possible to migrate an FPGA design to a structured ASIC that contains the same components as the FPGA, but removes most of the FPGA's programmability.

Problems are sometimes decomposed into smaller parts, where each part is mapped to another implementation technology. For example in molecular dynamics, CPUs and FPGAs are often combined, where FPGAs perform the particle interactions, and the CPU does the rest.

## 3.2 Technology

FPGAs obtain their flexibility by making all elements of a regular chip completely programmable: gates, registers, I/O pins, interconnect and clock generation. Many also provide dedicated hardware (e.g. adders or multipliers) to speedup certain computations. The structure of all FPGAs is very similar. The major difference is in the number of resources available and their speed.

### 3.2.1 Logic blocks

In an FPGA, Boolean functions are implemented by encoding the truth table in a small memory called a look-up table (LUT). For example a 2-input Boolean function can be implemented with a 4 entry table. The input variables are used as the address for the memory. Because the LUT in an FPGA can be written to, the function it implements can change at any point in time. In most chips, these functions are implemented using dedicated gates, which cannot be changed after manufacturing.

Modern FPGAs provide either 4-input LUTs (a 16 bit table) or 6-input LUTs (a 64 bit table). An  $n$ -input LUT can implement any function of  $n$  inputs or less, but it can implement only 1 function. So there is a trade-off: bigger LUTs allow for more complex functions (meaning a function with more inputs) to be implemented in 1 LUT, but smaller functions will only partially use the resources of a bigger LUT.

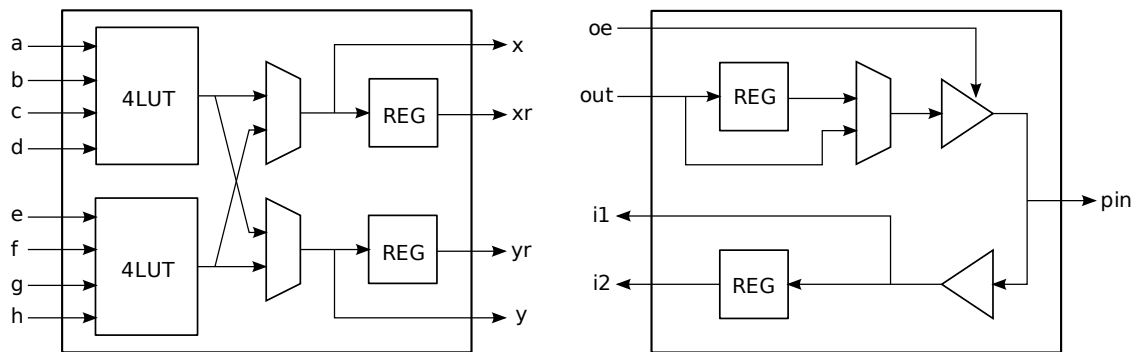


Figure 3.1: Block diagram of a logic block (left) and I/O block (right)

Storage is implemented by using registers in the form of flip-flops or latches. Often the registers can be configured to behave as either one. Typically, each LUT is paired with a register, so that its output can be captured immediately. The combination of a small number of LUTs and registers (about 1 to 8) is called a *logic block*. Figure 3.1 shows a simplified block diagram of a logic block consisting of 2 4-input LUTs and 2 registers.

Logic blocks typically contain extra logic to speedup certain computations. For example, the output of 2 4-input LUTs can be combined into a multiplexer to obtain a 5-input LUT (where the fifth input is the multiplexer input). Even some functions of up to 9 variables are possible. This way, LUTs don't have to be cascaded to implement large functions. Another popular extension is a dedicated carry-logic chain, that can quickly compute carry signals and forward them to the next logic block. This way, large adders can be made efficient without occupying too many resources.

Another direction that is taken is to make the LUT's memory writable by the application running on the FPGA itself. In case of a 4-input LUT, this gives a small 16-bit memory, that can be addressed using the 4-inputs. Many of these small memories can be tied together to form a big *distributed memory*. An extension of this is a shift register, where values are written into the shift register using a dedicated port, and where the 4 inputs specify the element to read.

All this extra functionality requires additional logic resources, which might go unused, and might have been used more effectively at providing more logic blocks instead of more complex ones. There is no consensus on what is the best logic block design.

### 3.2.2 I/O blocks

In an FPGA the input and output pins of the chip can be programmed to conform to many I/O signaling standards. This is accomplished by a programmable I/O block, that is associated with each pin of the chip. For example, the output voltage can be selected, as well as the input threshold. In many FPGAs, 2 of these I/O blocks can be combined into one differential pair to support differential signaling standards. Other resources often found in I/O blocks are registers to store inputs and outputs and double data rate facilities to interface to DDR DRAM for example. Figure 3.1 shows a simplified I/O block diagram.

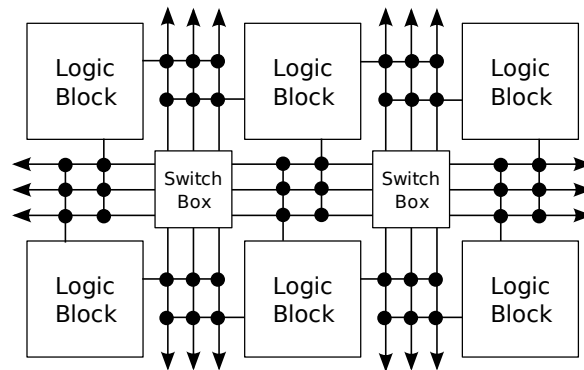


Figure 3.2: Simplified view of the routing of an FPGA.

### 3.2.3 Routing

In an FPGA signals can be routed from any block to any other block on the chip using a programmable routing interconnect. This routing structure consists of a big number of wire segments that can be connected together in *switch boxes*. Inputs to a block can select from a number of wire segments originating from this interconnect. Similarly, outputs of a block can drive a number of wire segments of the interconnect. If an output of a block needs to be connected to an input of another block, a sequence of wire segments must be found for which it holds that each segment can connect to the next segment, and that the 2 blocks can be connected to the end-points of the path.

Figure 3.2 shows a simple routing interconnect structure. The black dots are programmable switches, where the block inputs and outputs can be connected to wire segments of the interconnect. The switch boxes are also full of these programmable switches. The structure is very regular: an FPGA is typically constructed by repeating a small tile consisting of a logic block and some routing resources. At the edge of the tiling, the I/O blocks are placed.

The routing infrastructure of a modern FPGA is far more complex than this. First, there often are dedicated lines outside the generic routing infrastructure, most notable dedicated chip-wide networks to distribute signals with low latency, and dedicated wires between neighboring logic blocks. Second the generic routing infrastructure can have segments that span multiple switching boxes. This allows for faster signal propagation, but also requires more resources.

### 3.2.4 Clock generation

To synchronize operation on a chip, a clock signal is most often used. FPGAs provide programmable clock generators, that can generate many different clock signals based on a reference clock from a crystal or from a data stream. In combination with the chip-wide global routing resources, this is essential to high speed operation.

### 3.2.5 Dedicated blocks

Modern FPGAs provide many resources dedicated to a specific task, such as memory, multipliers and processors. In principle, these resources can be implemented completely using the programmable resources, but because they are needed so often, FPGA vendors decided to make dedicated blocks out of them. Some examples include:

**Fixed point multipliers** these are expensive to implement in programmable resources, and are used in many applications, especially DSP. They can also be used as a building block to construct floating point multipliers, which are used a lot in scientific computations.

**Blocks of SRAM memory** again expensive to implement using LUTs, even more so using registers, and is useful in virtually all applications. These SRAM blocks come in fixed size, but can always be grouped together in bigger blocks.

**Processors** a processor is complex and requires quite a lot of programmable resources to implement. It is used in many application that require a lot of logic that isn't performance sensitive, such as user interfaces and control.

**Communication** Often, people need access to DRAM memory, PCI express, Ethernet, or any other popular I/O standard. Some FPGAs provide dedicated blocks to handle these I/O interfaces.

Vendors often have different types of FPGAs, where the difference is mainly in the type of dedicated blocks that is provided by the FPGA.

### 3.2.6 Some numbers

Table 3.1 shows the resources available in the largest available Xilinx FPGAs over the years. Figure 3.3 shows the same numbers on a logarithmic plot. Not only the number of resources, but also their capabilities and performance have increased over the years. For example, LUTs have switched from being 4-input to being 6-input with the Virtex 5, which results in more resources per LUT, but not a large increase in LUT count. Focus on dedicated logic also has slowed the growth of generic resources (LUTs and registers) somewhat.

## 3.3 Programming

Upon initialization, the FPGA will load it's configuration through its pins. This configuration is a binary file that specifies the value for each SRAM bit on the chip: the LUTs, the routing controls, etc. Because each FPGA is different, it requires a different configuration file.

Designing the configuration is typically done with a hardware description language (HDL) such as VHDL or Verilog. The resulting design is then translated to a configuration for a specific FPGA, similar to how a compiler maps programming language source code to machine instructions. However, the process of generating a configuration for an FPGA is more involved. It typically consists of the following steps:

Device	Year	LUTs	Registers	Multipliers	BRAM (kib)
XC2018	1985	100 (4 input)	100	-	-
XC3090	1987	640 (4 input)	640	-	-
XC4025	1991	2,048 (4 input)	2,048	-	-
XC4085XL	1996	7,448 (4 input)	7,448	-	-
Virtex 1000	1998	24,576 (4 input)	24,576	-	128
Virtex 2 8000	2000	93,184 (4 input)	93,184	168 (18x18)	3,014
Virtex 2 Pro 100	2002	99,216 (4 input)	99,216	444 (18x18)	7,992
Virtex 4 LX 200	2004	178,176 (4 input)	178,176	96 (18x18)	6,048
Virtex 5 LXT 330	2006	207,360 (6 input)	207,360	198 (25x18)	11,664
Virtex 6 LXT 760	2009	474,240 (6 input)	948,480	864 (25x18)	25,920

Table 3.1: Largest available Xilinx FPGAs over the years

1. Synthesis: the HDL design is converted to a netlist representation, which is essentially a graph of the described circuit. This graph can also be used for simulating the design.
2. Mapping: the netlist is mapped into a new netlist, where the nodes are resources of the FPGA. For example, an adder must be broken up into separate logic blocks if the FPGA doesn't provide dedicated adders. Similarly small logic can be combined into 1 logic block.
3. Placing: the nodes of the mapped netlist are assigned to a physical FPGA resource. Any placement will do, but for the resulting speed of the design, it is essential that related logic is placed close together. Depending on the routing infrastructure, a bad placement could even make the design unroutable.
4. Routing: the connections of the placement are routed through the routing resources. Again, any routing will work, but for the speed of the design it is essential that a good routing is chosen.
5. Bit file generation: the resulting design is mapped into a bit file for a specific FPGA.

One of the weak points of FPGAs is that it is much harder to use an HDL than to use a programming language. People are now trying to map a simple programming language to an HDL automatically. There is no consensus yet on how this should be done.

### 3.4 FPGA machines

FPGA systems to implement scientific simulations can be grouped in roughly 2 categories:

- The co-processor model, where the FPGA works along-side a CPU, performing the computationally intensive parts of the computation, while the CPU performs lighter tasks and bookkeeping. This is usually accomplished by fitting a number of FPGAs on a board that hooks into an ordinary computer via e.g. the PCI bus. The main advantage of this approach is that it provides an easy way to accelerate an existing MD

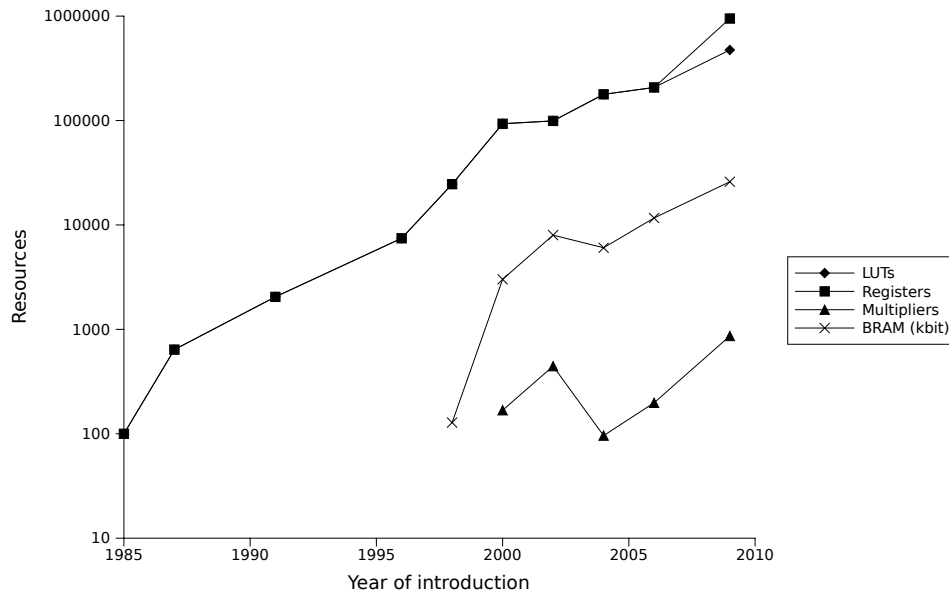


Figure 3.3: Largest available Xilinx FPGAs over the years.

program: only a small part has to be modified to run on the FPGA. An example of such a systems are the Annapolis Wildstar PCI boards that house up to 4 state of the art FPGAs [2].

- The stand-alone model, where the FPGA works alone, executing the complete algorithm. The FPGAs have their own memory and interconnect network through which they can communicate. The main advantage of such a system is system cost: no PC hardware is required. If a small CPU is necessary, FPGAs can implement them. An examples of such systems are the Digilent Virtex 2 Pro XUP boards that hold only an FPGA, a DRAM DIMM and some support chips [4]. Larger examples are the Berkeley BEE machines that house many FPGAs and many gigabytes of memory [3].

In this report we will focus on the stand-alone model, because it provided the biggest potential pay-off and hasn't been studied well. In particular we will focus on the Digilent XUP board for the actual implementation. This board contains the following hardware:

- Xilinx XC2VP30 FPGA.
  - Logic: 3,424 CLBs containing 13,696 slices containing 27,392 4-LUTs and regs.
  - I/O: 644 IOBs
  - DCMs: 8.
  - Multipliers: 136 (18x18 bit)
  - BRAMs: 136 (18 kbit each), 2,448 kbit total
  - Embedded PowerPC CPUs: 2
- Memory: 256 MB DDR DRAM, platform flash, compact flash slot

- Specific I/O: VGA video, AC'97 audio, 10/100 Mbit ethernet, UART
- Custom I/O: PS/2, Hirose, SATA connectors
- USB port to load configurations.

## Chapter 4

# Problem Statement

The problem is to design an architecture for executing MD simulations on a network of FPGAs and build an implementation to show that it can be done. The architecture should be scalable in the size of the simulations it can handle, and it should be scalable in the number of FPGAs that can be applied. The goal is, given enough hardware resources, to be able to run a simulation that has roughly 100x the current state of the art in the same amount of time it takes to simulate state of the art simulations on a supercomputer. Table 4.1 shows the current state of the art and the future system that is 10 times as big in both the number of time steps and the number of particles. The current state of the art is based on simulations described in [10]. Their simulations took about 2 weeks to simulate on a 256 CPU 1.66 Ghz Itanium2 system at the NCSA in 2006.

Metric	Variable	Current	Future
Number of particles	$N$	1,000,000	10,000,000
Number of time steps	$T$	50,000,000	500,000,000
Cell per dimension	$C_x, C_y, C_z$	20	43
Number of cells	$C$	8,000	80,000
Particles/cell (average)	$P$	125	125
Storage (Bytes)	$M$	$10^8$	$10^9$
Work (Operations)	$W$	$10^{18}$	$10^{20}$
Supercomputer time	$t_{sc}$	weeks	years

Table 4.1: Simulation sizes of current and future state of the art MD simulations.

As already stated in chapters 2, we will not implement a full MD simulation, but focus only on the Lennard-Jones force. This is enough to simulate noble gases for example, but not enough for biological simulations, which require bonded and electrostatic forces. The running time of a typical simulation is dominated by the Lennard-Jones computation, so the running time of the simulation shouldn't differ too much [28]. We will also restrict ourselves to cubic simulation boxes with periodic boundary conditions containing only 1 type of particle. The type of ensemble is restricted to micro-canonical.

The starting point of the hardware implementation is the LJPaVe molecular dynamics simulation program, that does exactly what was just described [8]. We will not map it 1-to-1 to

hardware, but the goal is to make the hardware implementation functionally equivalent with the software implementation. The hardware implementation should demonstrate that architecture works as advertised on a small simulation system simulated on 1 FPGA. The system will be implemented on a Digilent XUP board containing a Xilinx Virtex 2 Pro FPGA[4].

## 4.1 Related Work

Because molecular dynamics requires so much processing power, there have been quite some investigations into how to map it to a variety of alternative computation technologies. These can be grouped roughly into the following 4 categories: general purpose processors (CPUs), special purpose processors (GPUs), reconfigurable hardware (FPGAs) and special purpose hardware (ASICs). The CPU based implementation is always used as a reference point to compute speedups.

### 4.1.1 MD on general purpose processors

There are many software programs that can do molecular dynamics simulations of some kind. The 2 most well known that support a wide range of simulation types are NAMD [23] and GROMACS [19]. Both have been optimized to take maximum advantage of modern CPUs and to be scalable to large computer clusters. There are also many simpler applications such as LJPave [8], which is used as a reference throughout this report.

### 4.1.2 MD on special purpose processors

There have been some investigations in porting molecular dynamics code to special purpose processors such as the GPU and the Cell Broadband Engine. These processors have many parallel arithmetic units organized as clusters of SIMD units. They both have the property that there is very little local memory available, but a lot of bandwidth to external memory. These processors can give a big speedup, provided the arithmetic units can be kept busy. Studies in [22] and [29] report a speedup between 5x and 10x for these type of processors over ordinary CPUs. However, these special purpose processors do not always support double precision floating point, because they were designed for problem areas where precision is not as important as in scientific computations.

### 4.1.3 MD on special purpose hardware

One of the most successful projects that uses custom hardware for scientific computations is the Japanese GRAPE (GRAVity PipE) project. The problem they faced was astronomical N-body simulations, which is similar to molecular dynamics, except that molecular forces have been replaced by gravity. Because the gravitational force falls off slowly (just like the electrostatic force), applying a cut-off is not an option. Starting in 1989, the Japanese designed a series of 6 increasingly more powerful dedicated hardware units, named GRAPE-1 through GRAPE-6, to perform the force computation [24]. These units work as a co-processor

board that can be connected to a regular computer. This can be scaled up by connecting more of these boards to the computer or networking more computers together. The fastest GRAPE machine that has been build contains 2048 GRAPE-6 chips and has a claimed peak performance of 64 Teraflops.

Based on the same concept, they have also made a series of dedicated hardware units for molecular dynamics. These were build around chips that compute the Lennard-Jones force and the real part of the Ewald sum (named MD-GRAPE1, MD-GRAPE2, MD-GRAPE3) and chips that compute the reciprocal part of the Ewald sum (called WINE-1 and WINE-2) [20]. The fastest machine constructed so far with these dedicated parts is a 4800 chip machine build by RIKEN that has a claimed peak performance of 1 Petaflops.

Because of increasing cost of building these dedicated chips, they have switched to a more generic approach using very wide SIMD units called GRAPE-DR [15]. These are essentially special purpose processors, much like a modern GPU or the Cell Broadband Engine.

There have been some earlier projects which had similar goals, but their results were far more modest and have subsequently been abandoned.

#### 4.1.4 MD on configurable hardware

Starting in the year 2000, the GRAPE project also produced a number of systems named PROGRAPE-1 through PROGRAPE-4, where they replaced the dedicated chips with FPGAs [14]. The overall architecture of these systems is very similar to the architecture of the GRAPE systems, except that the PROGRAPE systems have a completely programmable pipeline. Molecular dynamics is mentioned as one of the possible applications, but they focus mostly on gravitational interactions. Performance numbers aren't given.

In 2002, Lienhart et al. targeted smoothed particle hydrodynamics, also an N-body problem [18]. They design and implement a pipeline that computes the kernel of this problem using 24-bits floating point. A library of parametrized precision floating point units developed by the authors provides the operators. The implemented pipeline has a theoretical performance of 3.9 Gflops, an estimated speedup of a factor 40 over a single CPU. No working system is demonstrated.

The first design of molecular dynamics on FPGAs is described by Scrofano et al. in 2004 [27]. They develop a pipeline that computes the Lennard-Jones force. They report estimated speedups of up to 20x of the FPGA hardware over a software implementation on commodity CPUs. When only force computation routines are compared, the speedup of the FPGA implementation drops to 2.5x. No working system is build.

Further studies are done in 2006 by Scrofano et al. [26] and Kindratenko et al. [16] to provide working implementations of the Lennard-Jones force computation. [26] also maps the Coulombic force computation to the FPGA using a cut-off radius, in the same way as is done with the Lennard-Jones force. They both use a direct implementation of the Lennard-Jones expression as a very deep pipeline, and both use single precision rather than double precision to fit the pipeline into a real FPGA. Both designs run on the SRC MAP station, which is restricted to running at 100 Mhz, and both use the Carte C to HDL compiler to generate a

hardware implementations. The reported speedups are relatively small: between 2x and 3x when 2 FPGAs are used.

Others studies map a complete molecular dynamics algorithm to FPGAs. The first fully working system is reported in 2004 by Azizi et al. in [7]. They implement the Lennard-Jones force and a Verlet integration scheme. The most interesting things about this design is that it uses a table look-up scheme to implement the Lennard-Jones force computation, and that they use fixed point representation everywhere, based on an analysis of the required precision. However, they use outdated FPGA technology with very little memory bandwidth, which results in speed-down of 3.5x. They estimate that a better FPGA and a better memory hierarchy would result in a speedup of 21x.

Later work by Gu et al. in 2005, who use a similar design to Azizi et al. confirms this claim [12]. Their design reaches a speedup between 41.5x and 88.5x depending on the precision required. They not only implement the Lennard-Jones force but also the Coulombic force. However, as for [7], the designs can only handle very small particle systems, and are therefore only proof of concepts of what can be achieved with an architecture designed specifically for FPGAs.

In Scrofano et al. [28] the first attempt is made to map the smoothed particle mesh Ewald (SPME) method to an FPGA. They map only the real space part of the computation to the FPGA, because it is the most expensive step and also relatively straightforward. Single precision floating point is used throughout the design and table based look-up is applied to evaluate the  $e$  and  $erfc$  functions. The reported speedup is 2.9x.

Around the same time, Lee et al. [17] map the reciprocal space part of the SPME computation to an FPGA. The rest of the algorithm was executed on the host microprocessor. Fixed point numbers are used throughout the design. Precision is determined based on an energy fluctuation requirement and experiments with varying precision. The implementation consists of 5 functional units that operate in sequence. A working system is build, but the speedup is not reported. However, they estimate that an improved implementation would be able to achieve a speedup of 3x to 14x.

In [13], Gu et al. extend their original Lennard Jones implementation with a multigrid implementation for the electrostatic force. They use semi floating point as the numbering scheme, which is floating point with a limited set of supported exponents. 35 bit precision is used throughout the design. The implementation consists of 2 functional units: a particle to grid converter and a grid convolver. A control unit provides proper input to these blocks. The reported speedup is 7.3x over ProtoMol.

# Chapter 5

## Architecture

### 5.1 Problem Analysis

The primary constituents of any computer are the processing elements (e.g CPUs or FPGAs), the memory elements and their interconnections. The processing elements are characterized by their speed in terms of the maximum number of operations that can be performed per unit time, for example in floating point operations (*flops*) per second. The memory elements are characterized by their capacity in bytes. Links of the interconnect network are characterized by their bandwidth in bytes per unit time and their latency in unit times. The latency and bandwidth constraints of the memory itself is aggregated in the link connected to it. In this section, we will analyze the requirements an MD algorithm places on these elements. It is assumed that the particles are stored in memory, and that the processing elements have very little local memory.

#### 5.1.1 Memory requirements

To run an MD simulation, we need to store at least 2 states: the current one  $s[t]$  and the next one  $s[t + 1]$  that is being computed from  $s[t]$ . A state is simply a set of particles, each having a position and a velocity. Because both position and velocity are 3 dimensional vectors, each particle is defined by 6 coordinates. If we assume the coordinates are stored as  $B$  bytes values each, we need  $6 * B * N$  bytes to store a state of  $N$  particles. Because we need to store 2 states, the memory requirement is  $12 * B * N$ . Table 5.1 shows the memory requirements for current and future MD simulations resulting from this formula.

Coordinate size	Current	Future
4 bytes (single precision)	48 MB	480 MB
8 bytes (double precision)	96 MB	960 MB

Table 5.1: Memory requirements for current and future MD simulations.

Even for future simulations with 64 bit precision per coordinate, only about 1 gigabyte of memory is required to store the entire simulation state. This is very modest given current memory technology.

### 5.1.2 Processor requirements

Processing requirements are much larger than memory requirements for MD simulations. For the brute force  $\mathcal{O}(T*N^2)$  algorithm, we compute  $T*N^2$  interaction steps and  $T*N$  integration steps. Based on the algorithm listed in figure 2.2, we estimate that computing 1 interaction takes 37 flops and that computing 1 integration takes 24 flops. Furthermore, we assume that any management overhead is negligible. The total number of flops can now be estimated as  $T * (N^2 * 37 + N * 24)$ . If we know the performance of a machine in flops per second, we can estimate the running time of a given simulation. Table 5.2 shows the estimated running time on a desktop machine (1 GFlops/sec) and on a super computer (10 TFlops/sec) for the reference simulations.

For the cell based  $\mathcal{O}(T * N)$  algorithm, the number of integrations is identical, but the interactions now splits into 2: distance computation and cut-off check on the one hand and force computation and accumulation on the other hand. Assuming a uniform density, the number of distance computations is  $T * C * 27 * P^2 = T * 27 * \frac{N^2}{C}$ : each time step  $T$ , all cells  $C$  (containing  $P = \frac{N}{C}$  particles) are checked for interaction candidates against 27 other cells (all containing  $P$  particles). The number of interactions and hence force computations and accumulations is only 0.16 this number:  $4.32 * T * \frac{N^2}{C}$ . Note that  $C = \mathcal{O}(N)$ , so the running time is not quadratic in  $N$ , as the formulas suggests. Based on the algorithm listed in figure 2.4, we estimate that computing the distance and checking for cut-off takes 19 flops, that computing the actual force and accumulating it takes 19 flops, and that integrating the force takes 27 flops. Table 5.2 shows the estimated running times.

Algorithm	Simulation size	Operations	Desktop Computer	Supercomputer
Brute force	Current	$1.85 * 10^{21}$ flops	58.7 k-years	5.87 years
	Future	$1.85 * 10^{24}$ flops	58.7 M-year	5.87 k-years
Cell based	Current	$3.72 * 10^{18}$ flops	118 years	4.31 days
	Future	$3.72 * 10^{20}$ flops	11.8 k-years	431 days

Table 5.2: Processing requirements for current and future MD simulations.

From table 5.2 it is clear why the brute force algorithm is almost never used: it would take 6 years of super-computer time to finish just 1 state of the art simulation! On the other hand, simulating that system using a cell based method would take less than a week, which is reasonable. It is also clear why future simulations aren't done yet: they currently take more than a year on a supercomputer.

To compare machines, it is more interesting to know the number of particle-particle evaluations per second or the number of interactions per second they are capable of for a given algorithm, rather than the running times for a certain simulation. Table 5.3 lists the estimated performance of desktop computer and supercomputer we just introduced,

Algorithm	Desktop computer		Supercomputer	
Brute force	27.0 M evs/sec	27.0 M ias/sec	270 G evs/sec	270 G ias/sec
Cell based	45.4 M evs/sec	7.3 M ias/sec	454 G evs/sec	73 G ias/sec

Table 5.3: Particle-particle evaluations (evs) and interactions (ias) per second.

### 5.1.3 Memory bandwidth requirements

The bandwidth to and from memory needs to be sufficient to keep the computation going. Each particle (6 coordinates) must be loaded and stored once for each time step when acting as reference particle. Furthermore, each particle-particle evaluation requires the loading of 1 particle position (3 coordinates). For the brute force algorithm, the total number of bytes to be loaded and stored then equals  $T * (12 * B * N + 3 * B * N^2)$ . For the cell based algorithm, this is again a bit more complex:  $T * (12 * B * N + 81 * B * \frac{N^2}{C})$ . Table 5.4 lists the total data transfers required for the reference simulations.

Algorithm	Current		Future	
	single precision	double precision	single precision	double precision
Brute force	$6.00 * 10^{20}$ bytes	$1.20 * 10^{21}$ bytes	$6.00 * 10^{23}$ bytes	$1.20 * 10^{24}$ bytes
Cell based	$2.03 * 10^{18}$ bytes	$4.06 * 10^{18}$ bytes	$2.03 * 10^{20}$ bytes	$4.06 * 10^{20}$ bytes

Table 5.4: Data transfers in bytes from and to memory.

Given the numbers in 5.4 and the running times of the previous section, we can compute the bandwidth required to be able to sustain that performance. The resulting bandwidth requirements for the desktop computer and super computer we introduced before are shown in table 5.5. Note that the cell based algorithm requires a larger bandwidth than the brute force algorithm because it performs more particle-particle evaluations per second.

Algorithm	Desktop bandwidth		Supercomputer bandwidth	
	single precision	double precision	single precision	double precision
Brute force	324 MB/sec	648 MB/sec	3.24 TB/sec	6.48 TB/sec
Cell based	545 MB/sec	1090 MB/sec	5.45 TB/sec	10.90 TB/sec

Table 5.5: Bandwidth requirements for current and future MD simulations.

From table 5.5 we can conclude that for the cell based algorithm we need about 0.5 bytes/flop for single precision coordinates and about 1 bytes/flop for double precision coordinates.

### 5.1.4 Memory latency requirements

Memory latency is a more subtle issue than memory bandwidth. For a read operation, the latency is the time it takes for 1 data item to be retrieved after the read command is issued. This highly depends on the type of memory that is used and the access pattern. DRAM for example has very high latency for the first memory read, but reads that are close to it (they are in the same DRAM row, but a different column) can go very fast. This means that it is typically a good idea to organize things in large contiguous arrays and do linear access only. On the other hand, linked lists with small elements are a relatively bad idea, because the high start-up latency has to be paid for each memory access. Allocating an array for each cell is one option to exploit this property.

### 5.1.5 Interconnect requirements

The analysis up till now works for uniprocessor systems, or shared memory multiprocessors. For machines with distributed memory, there needs to be communication between the processing nodes. Each processing node holds its own piece of the simulation box (its own collection of cells), and whenever a particle moves out of it, it has to be send to the node that holds the part of the simulation box that the particle now belongs to. How much has to be communicated depends on how the state is distributed, the topology of the interconnect network, and the simulation state.

Distributing the cells over the processing nodes should be done such that a minimum of particles “leak” out of any node. This is achieved by maximizing the volume to surface area ratio for the group of cells that is assigned to each processing node. For a cubic cell partitioning of the simulation box, a cube of cells maximizes this ratio. If the system is scaled, these cubes of cells also grow. The important observation is that if the linear dimension of the cubes is  $D$ , then the work is in the order of the volume, namely  $O(D^3)$  and the communication in the order of the surface area, namely  $O(D^2)$ . In other words, computation dominates communication in the limit.

The interconnect network should be such that particles have to traverse a minimum number of interconnect links. If the interconnect network is a fully connected graph, it doesn't matter how the cubes of cells are distributed over the processing nodes. A fully connected network is not required however, a 3 dimensional torus that mimics the simulation box with periodic boundary conditions is just as good. In both networks, there is only communication between any 2 adjacent nodes. If a system has a 2 dimensional torus interconnect, the cubic patches of cells are not a very good match. An alternative would then be to subdivide the base x-y-plane of the simulation box into squares and give each node a square, meaning it gets the column of cells that is above that square. This subdivision is used by the LJPave code to map the algorithm to a transputer network.

The simulation state determines how many movement there is per unit area. If the temperature is higher, there is in general more movement. So to simulate a hotter system would require more communication than a colder system. Unfortunately, this factor is inherent in the simulation, so we cannot change it in any way.

### 5.1.6 Conclusion

We can conclude that MD is mostly limited by the processor performance and the memory bandwidth. The memory required is relatively small, and latency shouldn't be a big issue if arrays are used. For a distributed machine, cubic patches of cells and a 3 dimensional torus network are optimal.

## 5.2 Storing the state

The simulation's state needs to be stored somewhere during the simulation. In FPGA systems, we have 2 places where we can store the state: on the chip itself in its embedded BRAM, or

in external DRAM. External memory can have much larger capacity than internal memory, but also has a much lower bandwidth and higher latency. A smart caching system is a way to combine the advantages of both memories into 1 virtual memory: keep the complete state in external DRAM, but keep often used data in the chip's BRAM.

Because we are concerned with a network of FPGAs, the second question is whether we store the state in a central memory, or spread it over a distributed memory. Distributed memory results in a larger aggregate bandwidth and capacity, but requires a scheme to communicate particles between separate FPGAs and to synchronize their operation.

These 2 choices: BRAM versus DRAM and central versus distributed storage give rise to 4 possible designs (see also figure 5.1):

**Central BRAM** This architecture severely restricts the size of the state. Even current state of the art is almost 15 times larger than even the largest FPGA can provide. For this reason this architecture is not a valid option for our system. This is illustrated in the table below for both single precision (SP) and double precision (DP) coordinates:

Chip	Capacity	Particles (SP)	Particles (DP)
Virtex 6 LXT 760	3,317,760 bytes	138,240	69,120
Virtex 2 Pro 30	313,344 bytes	13,056	6,528

**Distributed BRAM** This architecture distributes the state among each FPGA's BRAMs. This gives a larger aggregate bandwidth than storing the state in a central BRAM, but more importantly, it gives larger capacity. For single precision coordinates, the following table lists the number of FPGAs required to store the state:

Chip	Current (SP)	Current (DP)	Future (SP)	Future (DP)
Virtex 6 LXT 760	15 FPGAs	30 FPGAs	145 FPGAs	290 FPGAs
Virtex 2 Pro 30	153 FPGAs	306 FPGAs	1532 FPGAs	3064 FPGAs

**Central DRAM** This architecture solves the storage capacity problem, because external memory can be arbitrarily large. However, it is has a limited bandwidth, so the scalability could be a problem. To solve this, we can make use of a smart caching mechanism that uses the BRAMs to buffer portions of the state on the FPGA.

**Distributed DRAM** This architecture solves the storage problem as well as the scalability problem. However, as for the other distributed approach, it will require a distributed synchronization scheme.

Only the DRAM options satisfy the requirements for a scalable architecture. In this report we focus on the central DRAM architecture for its synchronization simplicity and see how far it can be pushed using smart memory access and caching schemes. If this system has been pushed to its limits, multiple of these systems can be put in parallel in a torus network to further speed up the computation. The resulting system would then be a hybrid between the last 2 options: distributed DRAM on the highest level and central DRAM one level below that.

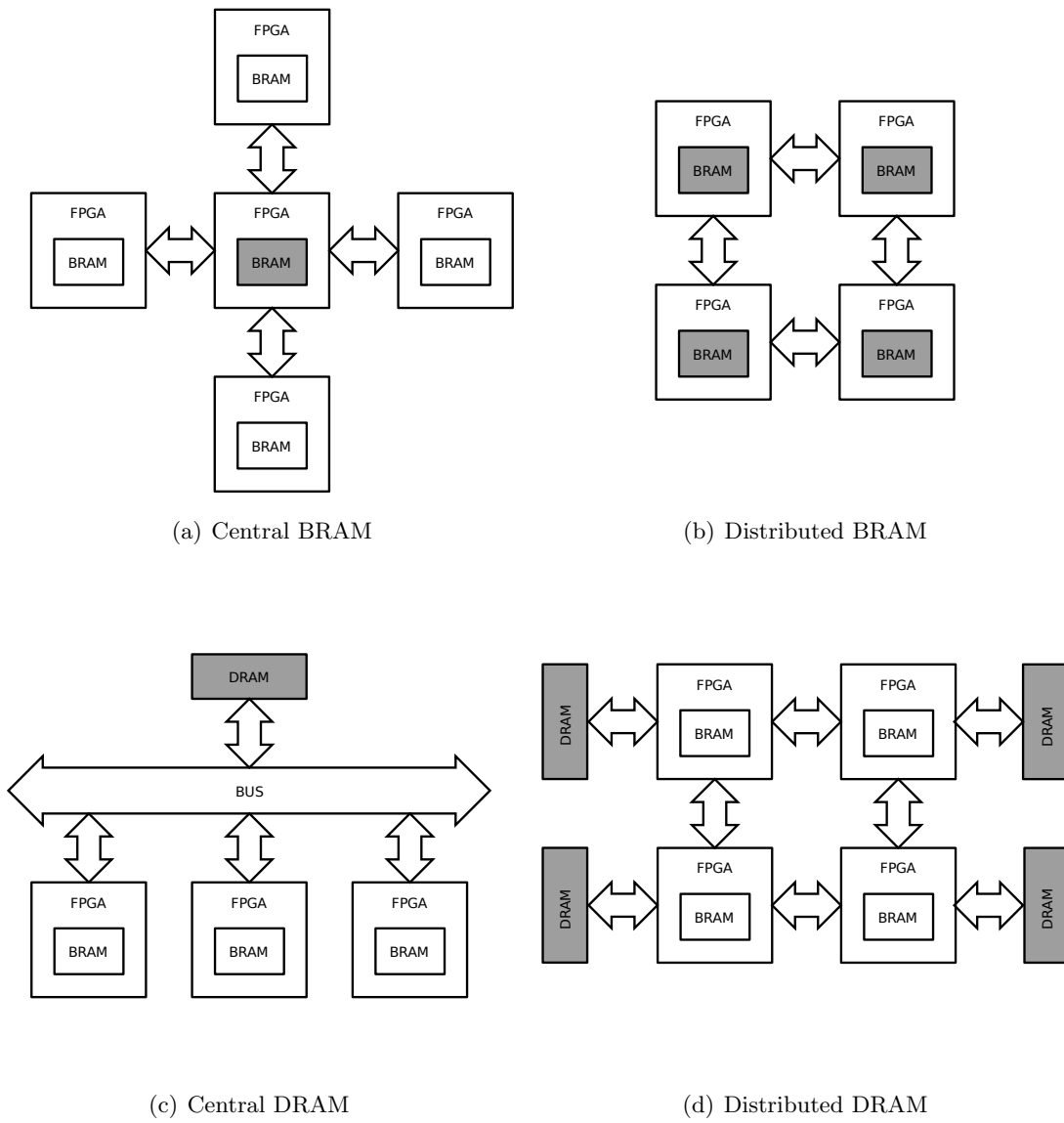


Figure 5.1: 4 possible architectures based on where the state is stored (gray).

## 5.3 Numerical representation & arithmetic

As was discussed in section 2.2, numerical integration schemes introduce errors. In actual implementations, finite representation also introduces errors. The goal is to choose the representation such that the errors introduced by the representation are smaller than the errors introduced by the integration scheme and time step [6]. Another goal is to choose the representation that is most efficiently implemented in the target platform, in our case FPGAs.

### 5.3.1 Representation

There are 2 numerical representations that can be used: fixed point and floating point. In both representations, the precision can be improved by adding more bits. The main difference is that floating point can more easily represent a wider range of numbers than fixed point. However, if the range of numbers that has to be represented is small, fixed point is a better choice because it is much cheaper to implement in hardware. Note that almost all software implementations use double precision floating point, simply because modern CPUs have floating point performance that is very close to (or equal to) that of fixed point performance.

For a floating point representation with a large enough range, Amisaki et al. demonstrate that for the Verlet integration scheme and a time step in the order of 1 fs, 25 bits precision is sufficient for positions, 29 for force computation, and 48 bits for force accumulation [6]. More precision can be used, but energy fluctuations will remain the same. On the other hand, if less precision is used, energy fluctuations will be larger. This shows that double precision floating point (which has a precision of 53 bits) is sufficient, but in most cases not necessary at all. Single precision (24 bits precision) should be used only if larger energy fluctuations are acceptable. Later work by [12] shows similar results: using more than 40 bits precision is useless for time steps of 1 fs.

For fixed point representation, more bits have to be used to be able to capture the whole range of possible numbers. Azizi et al [7] simply look at a demo run of the algorithm at full precision, and then determine the required precision and ranges based on the smallest and largest numbers that have been used during the simulation. They derive fixed point representations for various stages of the computation up to 78 bits, with typical values of around 50 bits.

### 5.3.2 Arithmetic on FPGAs

Floating point is more expensive than fixed point operations of the same bit width, because they need pre- and post-processing steps, and handle the exponent in a different way than the mantissa. For complex operations such as dividing and square root, this is not a big issue, but for simple operations such as addition, this can be expensive. Table 5.6 illustrates the performance, area and efficiency for the most common arithmetical operations. Both the combinational operations and a version with optimal pipelined depth are listed. For details regarding these operations and their performance, see appendix A.

As can be seen from this table, pipelining can be used to increase the performance and efficiency of the operators. However, pipelining has its limits: at a certain point, adding

Operation	Type	Width	Depth	Speed	Slices	Mults	Efficiency
Compare	fixed	32 bit	0	298 Mhz	16	0	18.63 Mhz/slice
Compare	float	32 bit	0	195 Mhz	41	0	4.76 Mhz/slice
Compare	float	32 bit	3	302 Mhz	46	0	6.57 Mhz/slice
Add/Sub	fixed	32 bit	0	289 Mhz	54	0	5.35 Mhz/slice
Add/Sub	float	32 bit	0	46 Mhz	342	0	0.13 Mhz/slice
Add/Sub	float	32 bit	13	294 Mhz	409	0	0.72 Mhz/slice
Multiply	fixed	32 bit	0	78 Mhz	55	4	1.41 Mhz/slice
Multiply	fixed	32 bit	4	247 Mhz	129	0	1.91 Mhz/slice
Multiply	float	32 bit	0	85 Mhz	103	4	0.83 Mhz/slice
Multiply	float	32 bit	2	186 Mhz	110	4	1.69 Mhz/slice
Divide	fixed	32 bit	0	208 Mhz	1307	0	0.16 Mhz/slice
Divide	float	32 bit	0	12 Mhz	405	0	0.03 Mhz/slice
Divide	float	32 bit	27	285 Mhz	739	0	0.39 Mhz/slice
Square root	fixed	32 bit	0	18 Mhz	227	0	0.08 Mhz/slice
Square root	fixed	32 bit	10	138 Mhz	222	0	0.62 Mhz/slice
Square root	float	32 bit	0	15 Mhz	276	0	0.05 Mhz/slice
Square root	float	32 bit	27	277 Mhz	475	0	0.58 Mhz/slice

Table 5.6: Performance, area and efficiency of common 32 bit operators

Operation	Width	Efficiency Fixed	Efficiency Float	Efficiency Ratio
Compare	32 bit	18.63 Mhz/slice	6.57 Mhz/slice	2.84
Add/Sub	32 bit	5.35 Mhz/slice	0.72 Mhz/slice	7.43
Multiply	32 bit	1.91 Mhz/slice	1.69 Mhz/slice	1.13
Divide	32 bit	0.16 Mhz/slice	0.39 Mhz/slice	0.41
Square root	32 bit	0.62 Mhz/slice	0.58 Mhz/slice	1.07

Table 5.7: Efficiency of 32 bit floating point and fixed point operators

more stages will not give a speedup that is justified by the extra resources that are required. Therefore, there will be an optimum pipeline depth for each floating point operator, at which they give the most performance per unit resources consumed. The numbers are summarized in table 5.7. Note that the best floating point divider is actually faster than the best fixed point divider. This is because the tools used to generate the operators (Xilinx Coregen) do not support pipelined fixed-point dividers.

Even though floating point is expensive in FPGAs, large FPGAs can put many of these operators in parallel to beat the peak performance of CPUs. Furthermore, the FPGA implementations can have a real performance that is much closer to their peak performance than a CPU implementation. Underwood [31] shows that although FPGAs do not systematically outperform CPUs in peak double precision floating point performance, their performance grows much faster than that of CPUs. He therefore extrapolates that FPGAs will be much faster in the future than CPUs at floating point computations.

## 5.4 Hierarchy of machines

We take the cell-based algorithm as listed in figure 2.4 as a starting point. In order to gain any speedup over a software implementation, we need to exploit the available parallelism in the algorithm and implement it using efficient hardware primitives.

The algorithm in figure 2.4 contains 5 loops. The outer loop iterates over all time steps, where each iteration depends on the complete output  $s[t - 1]$  of the previous iteration. So this loop cannot be parallelized<sup>1</sup>. The 2 loops inside the outer loop of the algorithm are embarrassingly parallel: they have no dependencies, other than that memory access needs to be arbitrated. The inner 2 loops do have dependencies: their output needs to be summed up to obtain the force  $f_i$ . In principle, all these 4 inner loop bodies can be executed in parallel, but for a scalable architecture, it is necessary to also be able to execute them sequentially.

The body of the inner loop is a very straightforward computation that compute a force vector from a pair of particle positions. The same holds for the integration step. This simplicity allows a direct mapping to hardware that can easily be pipelined. To keep these pipelines busy, we can keep inputting different  $j$  particles for a given reference particle  $i$ . This means that the inner 2 loops are executed sequentially using a very efficient pipeline. The outer loops can both be executed sequentially or parallel, depending on the desired degree of parallelism. This means that a parallelism factor of  $C * P = C * \frac{N}{C} = N$  can be achieved, which is 1M and 10M for our 2 reference systems.

If we assume that this pipeline can operate at 100 Mhz, we can compute a lower limit to the time it would take to run future simulations. In the best case, each particle has its own pipeline that performs  $27 * 125 = 3375$  particle-particle comparisons. Because future simulations are assumed to do  $5 * 10^8$  time steps, the running time is at least  $\frac{3375 * 5 * 10^8}{10^8} = 4:41$  hours. This is well within the limit of a week that we stated within the problem statement. On the other hand, if we need execution time of at most a week for future simulations, this requires at least 279,018 of these pipelines.

The described architecture is realized as a 3 level hierarchical machine: an *md machine* that computes new states for all cells, which is composed of a number of *cell machines* that compute the new state for a single cell, which itself is composed of a number of *particle machines* that compute the new state of a particle using an efficient pipeline. By varying the number of cell machines and particle machines, the degree of parallelism can be varied.

### 5.4.1 Particle Machine

The particle machine implements an efficient pipeline that computes the force  $f_i$  on a *reference particle*  $i$  given all neighboring particles  $j$  in sequence. It then integrates  $f_i$  to obtain the new position  $r_i$  and velocity  $v_i$  for particle  $i$ . The block diagram for this machine is shown in figure 5.2. Note that the stages of this pipeline have different data rates, because computing the force has to be done only once every 6 particle comparisons on average (i.e. out of every 7 particles, only 1 will pass the cut-off test). Integration is even slower: it has to be executed only once for every reference particle. This is inefficient in area. A better approach would

<sup>1</sup>Some parallelism is allowed if particles have a bounded speed

be to share force computation units and integration units among multiple particle machines. For the performance analysis, we stick with this somewhat inefficient design.

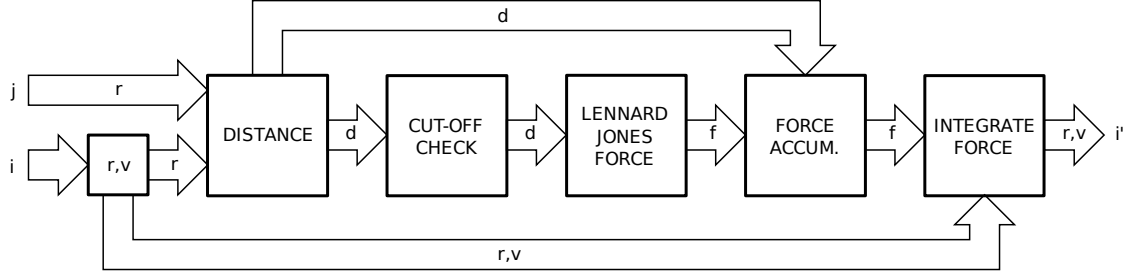


Figure 5.2: Block diagram of the particle machine

To make this pipeline efficient, meaning that it has a very low cycle time (and hence high clock frequency), many pipeline stages have to be added. This results in a rather high latency of the pipeline. Assuming that the work to be done for 1 reference particle is  $w_{pm} = 27 * P$  particle-particle evaluations and that the pipeline latency is  $l_{pm}$  cycles, the total execution time for 1 reference particle in clock cycles is given as:

$$t_{pm,1} = l_{pm} + w_{pm} \quad (5.1)$$

The pipeline latency  $l_{pm}$  could be as high as 200 cycles, so for the reference simulations where  $P = 125$ , the execution time is estimated to be 3575 cycles. If the particle machine cannot handle multiple reference particles simultaneously, the pipeline has to be drained (flushed) for every new reference particle. The time required to flush the pipeline is equal to its latency  $l_{pm}$ . This means that to execute  $n$  particles would take  $n$  times the execution of 1 particle:

$$t_{pm,n} = n * (l_{pm} + w_{pm}) \quad (5.2)$$

The overhead of this flush for the reference simulations is  $\frac{l_{pm}}{t_{pm,1}} \approx 5.6\%$ . This overhead can be reduced by allowing overlapped computation of multiple reference particles. Let's assume that the particle machine is constructed such that it can operate on  $o_{pm}$  reference particles in parallel, and that it can accept a new reference particle during any clock cycle. To completely eliminate all the overhead except for the start-up latency, the factor  $o_{pm}$  should be at least as big as the maximum number of reference particles that could occupy the pipeline during any clock cycle, which is  $1 + \lceil \frac{l_{pm}}{w_{pm}} \rceil$ . This means that in this scenario the pipeline can always accept a new reference particle. The execution time would then be:

$$t_{pm,n} = l_{pm} + n * w_{pm} \quad (5.3)$$

If we assume that this pipeline can run at 100 Mhz, it can effectively do 100 million particle-particle evaluations per second, about twice as fast as the 45 million evaluations per second that the desktop CPU of section 5.1 is able to perform. If this pipeline can run at 450 Mhz (about the maximum speed modern FPGAs support), it would be a factor of 10 faster than the CPU. By putting  $N$  of these pipelines in parallel (for maximum possible performance), the system can perform a theoretical 45 trillion interactions per second for current simulations (1 Petaflops/sec equivalent) and 450 trillion interactions per second for future simulations (10 Petaflops/sec equivalent). The running time for future simulations would then be about 1 hour.

### 5.4.2 Cell machine

The cell machine computes the new state for a certain cell given its environment (itself plus 26 neighboring cells). The block diagram for this machine is shown in figure 5.3. To save memory bandwidth, the cell machine buffers the environment in local memory (the FPGAs BRAM). This local memory then feeds  $p_{pm}$  parallel particle machines. All particle machines work with a different reference particle  $i$ , but all require the same neighboring particles  $j$ , because all  $i$  particles are from the same cell. This can be exploited by broadcasting the  $j$  particles over the input bus. The output of the particle machines is written to an output memory which has room for 1 cell.

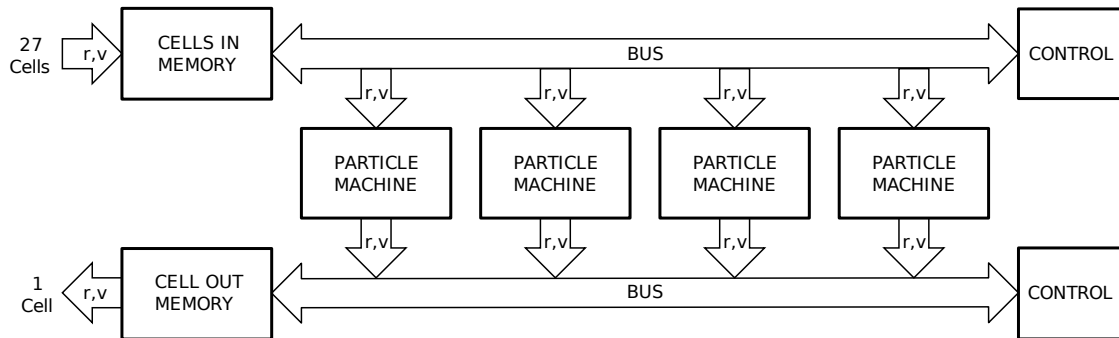


Figure 5.3: Block diagram of the cell machine

Note that particles can move from cell to cell, so the output of a cell machine is not necessarily the new state for the cell: it may contain particles that now belong to a different cell. The cell machine isn't concerned with this, it is handled when the particles are written back to memory in the MD machine.

An FPGA should be able to harbor at least 1 cell machine, so we need enough BRAM to store 28 cells (27 inputs and 1 output). For the reference simulations, the average cell sizes  $M_C$  are  $125 * 6 * B = 3$  kB for single precision and 6kB for double precision. Table 5.8 lists the number of cells that can be put on a number of FPGAs. As can be seen, even moderate FPGAs can house at least 1 cell machine. Of course, some headroom must be taken into account because the number of particles per cell can vary.

FPGA	BRAM	Cells (SP)	Cells (DP)	CMs (SP)	CMs (DP)
Virtex 2 Pro 30	2,448 kbit	102	51	3	1
Virtex 6 LXT 760	25,920 kbit	1080	540	38	19

Table 5.8: FPGAs BRAM capacity in terms of cells and cell machines (CMs).

A cell machine goes through 3 stages: load (loading all required cells in input memory), execute (computing new state for all particles in the center cell), and store (writing the output cell back to memory). We can estimate the times spend in these operations for the reference simulations, by assuming a cooperative environment with bandwidth  $b_m$  and latency  $l_m$ . We need to transfer a total of 28 cells (27 loads and 1 store), so the number of clock

cycles spend in loading and storing can be expressed as:

$$t_{l+s} = 28 * (l_m + \frac{M_C}{b_m}) \quad (5.4)$$

Assuming the bandwidth  $b_m = 16$  bytes/cycle and the latency  $l_m = 10$  cycles, this works out to be 5,530 cycles for single precision coordinates for the reference simulations. For double precision coordinates, this numbers doubles to 10,780 cycles.

Estimating the time spend in execution is a bit harder. We already know that the execution time of the particle machine  $t_{pm}$ . We assume  $p_{pm}$  parallel particle machines within the cell machine, where  $1 \leq p_{pm} \leq P$ . Furthermore, we assume that loading and storing takes 1 clock cycle, so for  $P$  particles, this works out to be  $2 * P$  clock cycles. There are  $P$  reference particles, and they have to be distributed over  $p_{pm}$  particle machines, so this results in  $\lceil \frac{P}{p_{pm}} \rceil$  reference particles for at least 1 particle machine. The execution time  $t_e$  is then estimated as:

$$t_e = 2 * P + t_{pm, \lceil \frac{P}{p_{pm}} \rceil} = 2 * P + \left\lceil \frac{P}{p_{pm}} \right\rceil * (l_{pm} + 27 * P) \quad (5.5)$$

For just one particle machine per cell machine ( $p_{pm} = 1$ ), the execution time is 446,877 cycles, about 81 times larger than  $t_{l+s}$  for single precision coordinates. For 125 particle machines,  $t_e$  drops to a mere 3825 cycles; only 70% of  $t_{l+m}$ , meaning that in this scenario the communication of the cells is the most time consuming part. The total time to compute the new state for 1 cell by the cell machine,  $t_{cm}$ , is the sum of the load, store and execution times:

$$t_{cm} = t_{l+s} + t_e = 28 * (l_m + \frac{C_B}{w_m}) + 2 * P + \left\lceil \frac{P}{p_{pm}} \right\rceil * (l_{pm} + 27 * P) \quad (5.6)$$

The loading time can be reduced by a factor of 3, if the next cell to be computed is adjacent to the current one. These cells share 18 of the cells in their environment, so only 9 new cells have to be loaded. The time spend in load and store is then computed as:

$$t_{l+s} = 10 * (l_m + \frac{M_C}{b_m}) \quad (5.7)$$

For single precision, the load and store take a total of 3750 clock cycles for the reference simulations, lower than the lowest possible execution time. So in this scenario, communication will never take more time than computation, no matter how many particle machines are applied.

If we increase the number of cells we can store locally, we can overlap loading and storing with execution, thereby improving the performance of successive invocations. If we assume that available memory for cell storage is doubled, the time it takes to load, execute and store  $n$  successive cells becomes

$$t_{cm,n} = t_{l+s} + t_e + (n - 1) * (t_e \uparrow t_{l+s}) \quad (5.8)$$

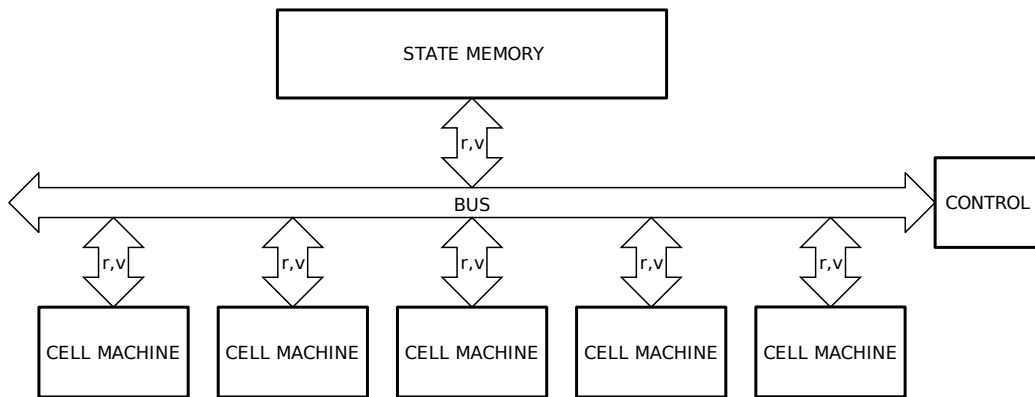


Figure 5.4: Block diagram of the MD machine consisting of 5 parallel cell machines

### 5.4.3 MD machine

The MD machine is the top-level machine, that computes all the successive states of the simulation. It does this by distributing the cells of the current state  $s[t]$  over the available cell machines and writing back the results to the new state  $s[t + 1]$ . The block diagram is shown in figure 5.4. It consists of  $p_{cm}$  cell machines and a central memory accessible through a bus.

This whole system can be implemented in 1 FPGA (except for the state memory), or can be mapped to many FPGAs, where 1 FPGA serves as the controlling FPGA that is connected to the main memory, while the other FPGAs are housing at least 1 cell machine each.

The MD machine hands out cells to the cell machines according to some *scheduling strategy*. As was clear from the previous section, it is a good idea to give cell machines a cell that is adjacent to the cell it computed in the previous invocation, to obtain maximum reuse of the locally buffered cells. Furthermore, there should be some load balancing: all cell machines should get an equal amount of work. There are 4 scheduling strategies that have been considered:

**First come first served** that goes through all cells in a memory in some arbitrary order and hands out each cell to the first cell machine that is available earliest. This strategy completely disregards the cells that are already buffered locally by the cell machines. On the other hand, it is extremely easy to implement. Because for this scheduling strategy,  $t_{l+s}$  is about 81 times smaller than  $t_e$  for the reference simulations, it is expected to be able to keep 81 cell machines busy that each contain 1 particle machine.

**Segmented Hamilton path** that generates a Hamilton path through all the cells, and then divides it into equally sized segments. The Hamilton path goes through all cells exactly once, so that the next cell on the path will always be adjacent to the current cell (see figure 5.5). By subdividing this path into equal sizes and giving each cell machine its own segment to work on, the locally buffered cells are reused as much as possible. For a completely uniformly distributed particle system, this is the optimal scheduling strategy. Because it is a factor 2.7 more efficient than the first come first served, it is

expected that for the reference simulations it is able to keep 218 cell machines busy, each containing 1 particle machine.

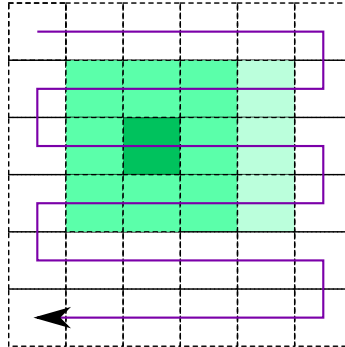


Figure 5.5: A 2 dimensional Hamilton path.

**Balanced Hamilton path** that extends the segmented Hamilton path to have segments of variable length. The length of the segments now depends on the number of particles on it, rather than the number of cells. This should give a better indication of the amount of work that has to be performed for each segment. This results in simple, but suboptimal load balancing for the cell machines, which improves the performance on non-uniformly distributes system.

**Work balanced Hamilton path** that divided the Hamilton path in segments based on the actual work that it takes to compute the new state for that path. This results in improved load balancing, as compared to the balanced Hamilton path strategy. However, the implementation is also more complex.

Note that for non-uniform systems, finding the optimal schedule is very difficult. Although it seems that the work balanced Hamilton path strategy is optimal, it is not. The problem is that the path taken through the simulation box is always the same, only the segmentation is different. All possible Hamilton paths would have to be tried to find the optimal schedule.

In the cell machine we could make use of broadcasting to increase effective memory bandwidth without needing wider memories. This could also work for the MD machine, because each cell is used by 27 computations of a new cell. However, the Hamilton path does not guarantee in any sense that these computations are mapped to 27 different cell machines that happen to require these cells at exactly the same time. A completely different scheduler would be required, that traverses the simulation box as a moving front. Inside the front, there is many overlap in the cells required (each cell is required by 9 cell machines), so broadcasting can be used to load the cells. The practical speed-up for this scheduler would be quite a bit slower, because it cannot traverse a Hamilton path, and therefore will not be able to take as much advantage of already available local cells.

Storing the cells in memory is easy if a uniform distribution is assumed. We can simply allocate an array of fixed size for each cell. Because the worst case number of particles for a cell is very close to the average case, little memory is wasted. However, if we assume that

there could be differences in density of a factor 10, then up to 90 percent of the memory can be wasted if arrays are used. Linked lists of particles would solve this problem completely, but as we mentioned earlier, they are a bad match for DRAM. A solution would be a file system like block structure that is a linked list of particle blocks. Each particle block is large enough to be able to make use of the pipelined nature of DRAM access, but not so big as to waste too much memory.

Estimating the execution time of the MD machine is not so easy, because cell machines may have to idle when they need to transfer data while the bus is locked. Furthermore, when the bus is available, all cell machines might be busy. This was one of the reasons that prompted the design of a high level hardware simulator (see chapter 6) , that could give insight into the performance of an actual MD machine.

If we assume that for each cell that is computed, 18 of the 27 cells in its neighborhood are already buffered in the designated cell machine, we can compute how many particle machines can possibly be served without saturating the memory interface. We assume the memory interface can provide 16 bytes per cycle and has a latency of 10 cycles that has to be paid only once. For the simple case where all cell machines contain 1 particle machine, we can reuse the cell machine analysis of the previous section. The cell load and store time  $t_{l+s}$  is then 1778 clock cycles, while execution time is 421875 cycles. By dividing these 2 numbers, we see that we can keep at most 237 particle machines busy at once. This is close to the number of 218 cell machines that we estimated that could be served simultaneously.

A better memory interface (4 channel DDR DRAM that runs at twice the system) frequency would be able to supply an 8 fold improvement in performance, meaning that at most 1900 particle machines can be kept busy using this memory. This is 2 orders of magnitude short of the target of 270,000 particle machines. If we really want to perform future simulations within a week, we would need at least 142 parallel MD machines, each working on their own piece of the simulation box.

## Chapter 6

# Hardware simulator

A hardware simulator was built to bridge the gap between the software implementation and an actual hardware implementation. Firstly, this proves that the design works as advertised, and secondly, it can be used as a more elaborate performance model. In order to do the former, the simulator must run the actual MD simulation, and to do the latter it must be close to cycle accurate and have many tunable knobs.

To validate the simulator, a reference program was obtained by removing the Verlet lists and the parallel computing facilities from the LJPaVe program. The resulting program implements the algorithm as shown in figure 2.4. The reference program is about 482 lines of C code. It has command line parameters to define the simulation parameters and the file from which to load the initial state.

### 6.1 Architecture

The simulator is implemented in C++ to be able to track the C reference implementation as closely as possible, while at the same time being able to exploit C++'s more expressive language constructs and more comprehensive standard library. The simulator is about 2800 lines of (rather verbose) C++ code.

#### 6.1.1 Static view: machine hierarchy

The simulator is based on the concept of a hardware machine that keeps track of its own state and time. Hardware machines are organized hierarchically to mirror the structure in the actual implementation. The hierarchy of machines synchronize their times so that when the simulation is finished, the time of the top-most machine indicates the actual time elapsed on the hardware.

An abstract hardware machine is modeled by the class `machine`, that only has the means to keep track of time and of the machine hierarchy, but that doesn't implement any hardware functionality. Derived classes of `machine` are specific machines that define a set of operations that implement that machine's functionality. In the simulator there are currently 3

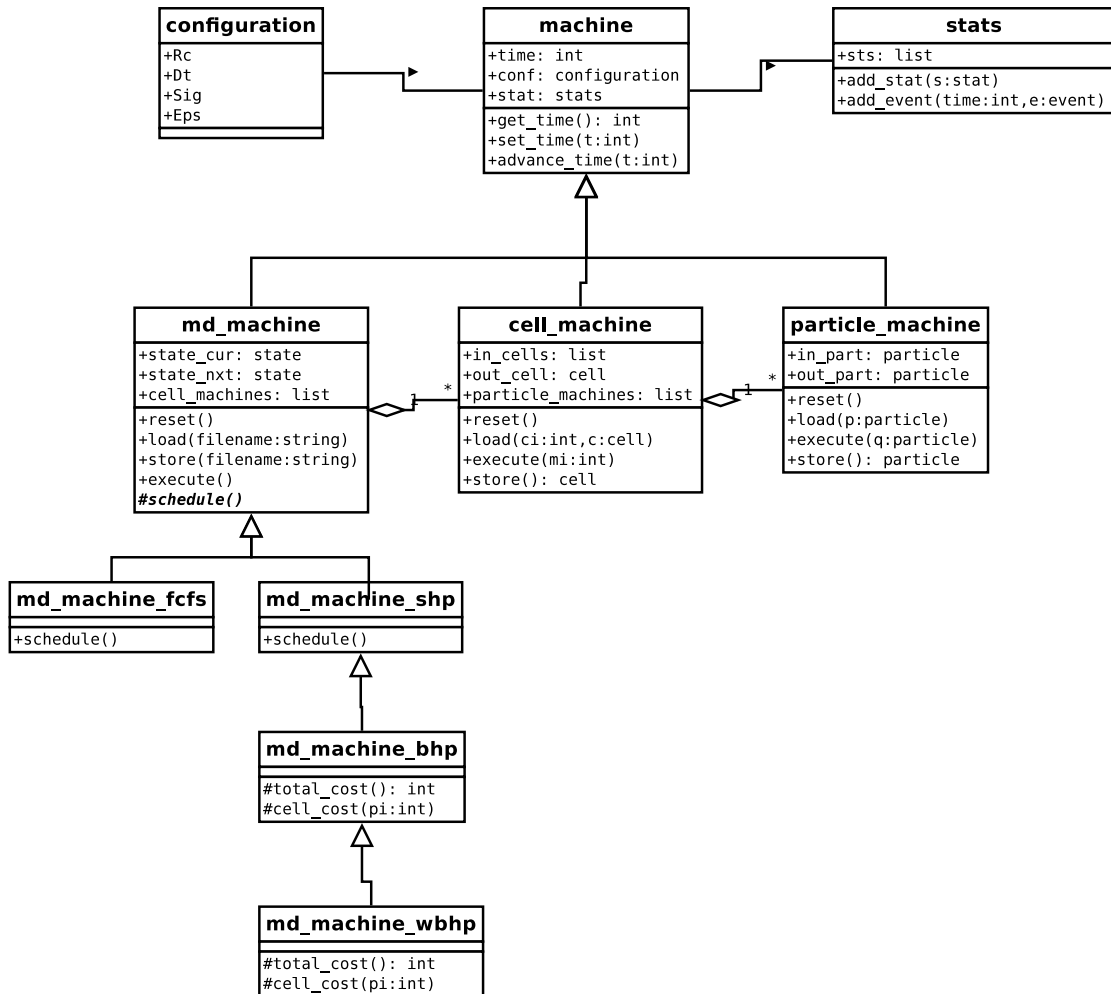


Figure 6.1: Class diagram of the simulator

direct descendants of `machine` that implement the 3 machines in the design: `md_machine`, `cell_machine` and `particle_machine`. See figure 6.1 for the complete class diagram of the simulator.

To obtain the hierarchy of machines, each machine has a back-pointer to the parent machine and can contain instances of other machines (its children). In our case, an `md_machine` instance contains many `cell_machine` instances and a `cell_machine` instance contains many `particle_machine` instances.

Different schedulers for the MD machine are implemented as subclasses of the MD machine. This could also have been achieved by aggregation, where each scheduling algorithm is put into its own class that is then associated with the MD machine. However, because the scheduling algorithm is by far the largest part of the MD machine and is tightly integrated with it, it has been decided that a subclass would be more appropriate. There are currently 4 subclasses of `md_machine` that implement different cell scheduling strategies: `md_machine_fcfs` that implements the first come first served scheduler, `md_machine_shp` that implements the segmented Hamilton path scheduler, `md_machine_bhp` that implements the balanced segmented Hamil-

ton path scheduler, and `md_machine_wbhp` that implements the work balanced Hamilton path scheduler.

### 6.1.2 Dynamic view: discrete event simulator

There are no threads in the design, so the control flow is sequential. Contrary to most discrete event simulators, there are no event queues. Rather, the control is explicit in the code itself. This makes the program a little less flexible, but makes it a lot easier to implement.

Each machine has its own notion of time in the number of clock cycles that have occurred since the start of the simulation. If a machine performs some operation, it advances its time by the number of clock cycles that it would take to do that operation. If the machine has to call some of its child machines to perform an operation, it first updates their time with its time, and then lets them execute. When they are finished, it updates its own time with the new time of its children. When the operation is finished, it returns control to the calling (parent) machine, that in turn updates its time, and so on. The number of clock cycles to perform certain operations is estimated based on a user specifiable set of performance parameters.

As an example, take the MD machine with a first come first served scheduling strategy. Its time  $t$  is initialized to 0. It then starts handing out 27 cells to the first cell machine, which takes  $t_l$  cycles. Its internal time as well as the time of the cell machine is then advanced by  $t_l$  cycles. Then the cell machine gets the execution command from the md machine, after which it computes the new particle positions in  $t_e$  cycles. It then advances its local time by  $t_e$  cycles and returns control to the MD machine. The md machine reads out the cell machines time and uses that as its own new time. If  $m$  cell machines run in parallel and the MD machine has to wait for all to finish, the MD machine updates its time to the maximum of the times of all  $m$  cell machines. More elaborate time update schemes can be used for components that can accept multiple inputs before returning the first output, such as the particle machines.

The accuracy of the time bookkeeping depends on the detail of implementation of the machines. In principle, to compute the running time on a specific problem, each machine can be replaced by a stub that simply estimates the running time and then return immediately. In the other opposite, each machine can be a detailed hardware implementation that is split recursively into smaller machines until circuit primitives are reached.

### 6.1.3 Configuration

The kind of simulation that has to be run and the machine parameters for the run are stored in a configuration object that is associated with each machine. Upon initialization, each machine configures itself to comply with the parameters in this configuration object. The configuration object is initialized from the command line and from the input file that describes the simulated system and its size. The following parameters, that are also accepted by the reference program, define the molecular dynamics problem that has to be performed:

**File** (default = `LJ.uni`) : the name of the file that contains the particle system to be simulated. The format of this file is assumed to be the same as what the `LJPaVe` and reference codes expect. This means a text file where the first line lists the universe sizes

$(L_x, L_y, L_z)$  and the number of particles ( $N$ ) in that order. The remaining lines list the particles, one per line, where each particle has 6 coordinates that are listed in order  $r_x, r_y, r_z, v_x, v_y, v_z$ .

**Xmul, Ymul, Zmul** (default = 1): multiplication factors that indicate how many times the particle system must be replicated in each dimension. The number of particles goes up by a factor of  $Xmul * Ymul * Zmul$ . This makes it easy to run the simulator on a variety of input sizes by just varying these parameters.

**Max\_IT** (default = 80): the number of time steps for which the simulation is run.

**Sig, Eps** (default = 1.0): the  $\sigma$  and  $\epsilon' = \epsilon/\sigma^2$  parameters of the Lennard-Jones potential.

**Rc** (default = 2.5  $\sigma^*$ ): the cut-off distance  $r_c$  for the Lennard-Jones potential in normalized length units  $\sigma^*$ .

**Dt** (default = 0.001  $t^*$ ): the time step  $\Delta t$  of the integration scheme in normalized time units.

The following parameters define the hardware system that must be simulated:

**cm\_count** (default = 1): the number of cell machines per md machine.

**pm\_count** (default = 1): the number of particle machines per cell machine.

**dram\_latency** (default = 10): the number of clock cycles between start and finish of a random DRAM read or write operation.

**dram\_ops\_per\_cycle** (default = 2) : the number of operations supported per clock cycle (1 for ordinary DRAM, 2 for DDR DRAM, etc.)

**dram\_bytes\_per\_op** (default = 8): the width of the DRAM interface in bytes. The bandwidth can be obtained by multiplying this with `dram_ops_per_cycle`.

**dram\_ops\_per\_row** (default = 128): the width of a DRAM row in terms of the DRAM interface width. The model assumes that consecutive reads from the same DRAM row can be pipelined, so that the latency only has to be paid for the first row access. The default value of 128 means that the DRAM has rows of size 8192 bits.

**pm\_flush** (default = false): whether the particle machine must be flushed upon switching reference particle, or that it can support multiple reference particles.

**pm\_latency** (default = 202): the latency in clock cycles of the particle machine pipeline.

**pm\_cycle\_time** (default = 1): the number of cycles the particle machine requires to accept a new data item.

**cs\_strat** (default = fcs): the cell scheduling strategy of the md machine.

### 6.1.4 Gathering results

This design separates the functionality of the machine from gathering data about its operation. To get information out of the simulator, each of the machines generates events that are passed to the `stat` object associated with each machine. The `stat` object distributes the events over all registered statistics generating objects. These objects store some internal state that is updated each time a relevant event is processed. Irrelevant events are simply ignored. An interesting feature is that statistics can depend on other statistics, so that derived statistics can be easily computed.

Before the simulation is started, a `stat` object is created that is populated with the desired statistics generating objects. Currently, this is not configurable at run time, but only at build time. At the end of each time step the gathered statistics are reported to the user.

The statistics that can be generated for each iteration are:

- The total number of simulated clock cycles spend.
- The number of cell load, execute and store events and the cycles spend in those operations.
- The utilization of the memory, i.e. the ratio of total number of cycles to the cycles spend in load and store.
- The number of particles processed by the particle machines.
- The number of particle-particle evaluations and interactions done by particle machines.
- The particle machine utilization, i.e. the ratio of cycles they have work to do to the total number of cycles elapsed.
- The computational efficiency, i.e. the ratio of particles that are considered for interactions that turn out to interact.
- The kinetic, potential and total energy in the system and the same values per particle.

For the performance, the most important factors are the total number of clock cycles spend, the memory utilization factor and the particle machine utilization factor. The others are mainly for validating the correctness of the computation.

The following listing demonstrates a call to the simulator (called `archsim`), and the resulting output for the first and only time step that is simulated:

```
$ ./arch-sim File sim.1k.uni \  
    max_IT 1 \  
    sched fcfs \  
    cm_count 1 \  
    pm_count 1 \  
    pm_flush 1 \  
    pm_latency 100 \  
    dram_latency 10
```

```

Report at time 1150632
Time spend:                1150632
Physical time steps:       1
Cell load events:          729
Cell execute events:       27
Cell store events:         27
Time spend in cell loading: 49788
Time spend in cell storing: 1844
Time spend in cell execution: 1099000
Time spend memory operations: 51632
Memory utilization:        0.044873
Particle interactions:     80000
Particles processed:       1000
Particles compared:        999000
Computational efficiency:  0.080080
Total potential energy:    -3330.890625
Total kinetic energy:      125.925858
Total energy:              -3204.964844
Potential energy per particle: -3.330891
Kinetic energy per particle: 0.125926
Total energy per particle: -3.204965
Particle machine utilization: 0.868218

```

## 6.2 Uniform particle systems

We start with simulating uniformly distributed particle systems. To prevent having to run systems of many million particles, we have restricted ourselves to 2 systems that are significantly smaller, but are sufficient to demonstrate the validity and scalability of the architecture and to show the effect of different schedulers. These systems are shown in table 6.1

Metric	Small system	Medium system
Simulation box ( $\sigma^*$ )	10x10x10	40x40x40
Particles	1000	64,000
Cells per dimension	3	14
Cells in total	27	2,744
Particles per cell (average)	37	23
Evaluations per time step (average)	998,001	42,674,688
Interactions per time step (average)	159,680	6,827,950

Table 6.1: Uniform particle systems that were used for simulation.

Both systems have particles allocated to grid points of a cubic 3 dimensional lattice. The velocities are initialized to random values within the range -1 to 1. The goal is not to have a realistic simulation, but to have some test benches to validate the design and the performance estimates.

### 6.2.1 The first come first served scheduler

Firstly, the performance of the MD machine with the “first come first served” (fcfs) scheduler is examined for the 2 particle systems described above. We have assumed that the particle machine pipelines do not have to be flushed when switching reference particles and have a latency of 202 cycles. Furthermore, we have assumed that the memory is 64-bit DDR DRAM with 1024 bytes per row and a latency of 10 clock cycles. The remaining degrees of freedom are the particle and cell machine count. Both parameters have been varied individually (see figure 6.2 and 6.4).

For the small particle system, increasing the number of cell machines is improving performance until 27 cell machines are reached, after which there are more cell machines available than cells to be processed. Putting more particle machines in 1 cell machine is initially giving less benefit than simply increasing the cell machines, because memory transfers and execution cannot be interleaved if only 1 cell machine is used. However, the maximum speedup that is obtainable with 1 cell machine is larger, because there are simply more particles per cell than cells in the simulation box.

For the medium system, the same analysis holds. The main difference is that for this system, there is less work to do per cell, so the memory interface becomes a bottleneck even sooner. If there are on average  $P$  particles in a cell, the number of particles that need to be transported to compute the new state for that cell is  $27 * P$ , while the work is in the order of  $27 * P^2$ . At 30 cell machines, the execution time is completely dominated by the memory operations. This can be demonstrated by taking 30 cell machines and then either decreasing the memory’s latency or increasing its bandwidth. This is demonstrated in figure 6.5. Note that increasing the bandwidth is more effective than decreasing the latency, which is good, because increasing bandwidth in an actual design is far easier than decreasing latency.

### 6.2.2 The segmented Hamilton path scheduler

To test the segmented Hamilton path scheduler, the exact same parameters (except the scheduling strategy) have been used as for the first come first served scheduler. See figure 6.3 and 6.4 for a visualization of some of the results. For the 1k system, the segmented Hamilton path scheduler is slightly faster for a small number of cell machines, because less cell load operations have to be performed. Then the performance of both schedulers starts to converge, and becomes equal when 27 cell machines are reached. At this point, the segmented Hamilton path scheduler is assigning segments of length 1 to each cell machine, meaning it has become functionally identical to the first come first served scheduler. Adding more particle machines per cell machine helps quite a bit more than adding new cell machines, because this reuses more of the locally buffered cells. This is exactly the opposite situation as compared to the first come first served scheduler.

For the medium size 64,000 particle system, it is far better to add more cell machines than to add more particle machines per cell machine. As can be observed from the third graph, near perfect scaling is obtained with this. This works because there are plenty of cells to keep tens of cell machines busy. However above 30 cell machines, memory bandwidth becomes a major bottleneck.

The balanced and work balanced Hamilton path scheduler perform identical to the segmented Hamilton path scheduler for a completely uniform particle system, because the generated segments are identical for all 3 schedulers (the work in each segment is directly proportional to the segments length).

### 6.3 Non-uniform particle systems

Particle systems with a non-uniform distribution of particles require load balancing to distribute the work equally over the available cell machines. The first come first server scheduler will automatically perform load balancing, but as we saw earlier, makes inefficient use of the memory bandwidth. The segmented Hamilton path scheduler on the other hand does no load balancing at all. The balanced Hamilton path scheduler and the work balanced Hamilton path scheduler were introduced to add load balancing capabilities to the Hamilton path schedulers.

To test load balancing, a particle system was created containing 64,000 particles, where 90 percent of the particles was allocated to 1 side of the simulation box. To demonstrate the performance of these schedulers, they were tested on a variety of hardware configurations. The table below lists the performance of the 4 schedulers on a system comprised of 10 cell machines, each containing 2 particle machines.

Scheduler	Cycles (total)	Cycles (memory)
first come first served (fcfs)	4,348,189	3,636,948
segmented Hamilton path (shp)	7,213,150	1,307,258
balanced Hamilton path (bhp)	3,947,209	1,311,297
work balanced Hamilton path (wbhp)	3,653,706	1,312,272

Table 6.2: Performance of the 4 schedulers on a non-uniformly distributed particle system.

From this table it is clear that the balanced Hamilton path scheduler is a vast improvement over the ordinary segmented Hamilton path scheduler. This advantage increases as the non-uniformity of the particle system increases.

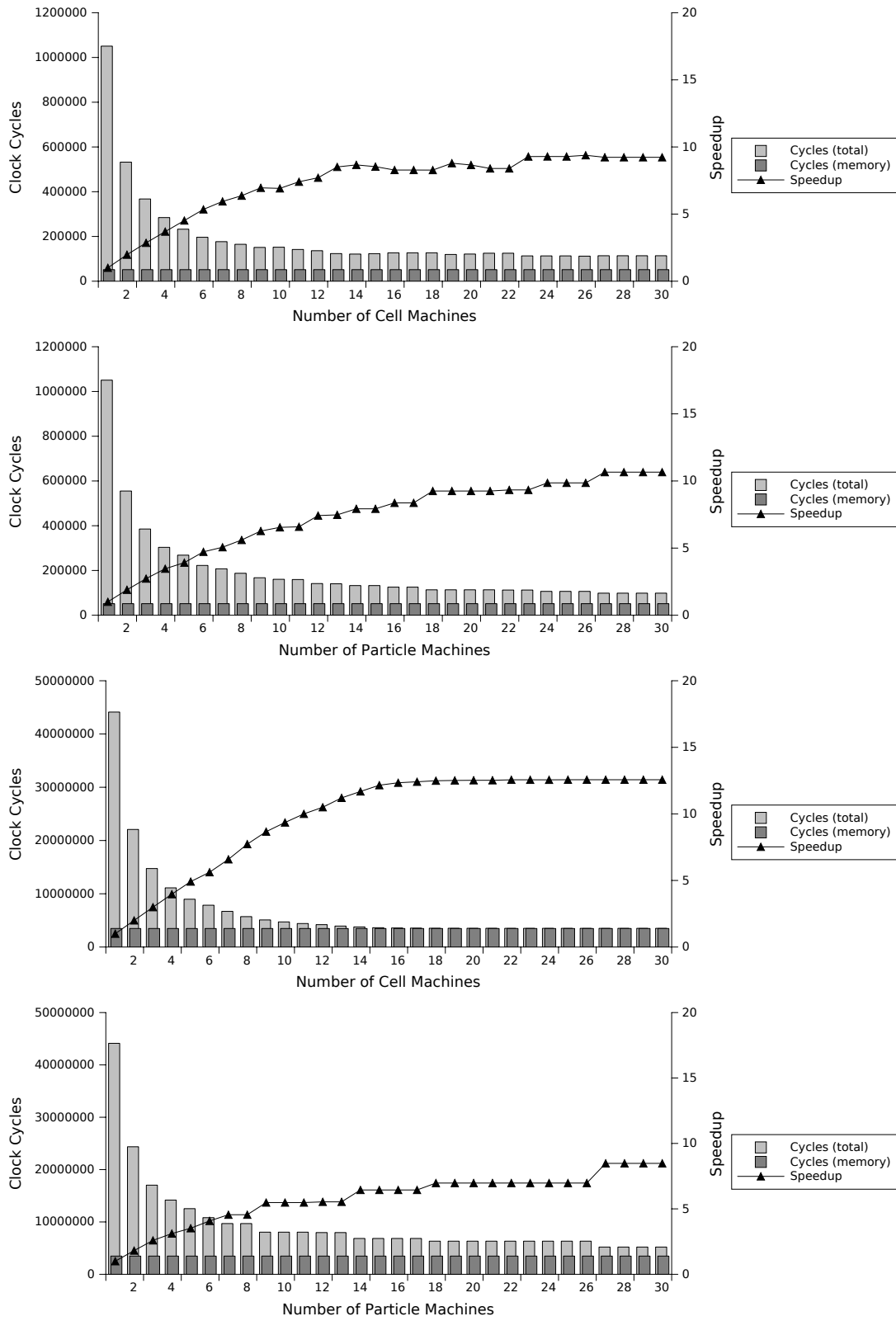


Figure 6.2: Simulation results for the first come first served scheduler: 2 graphs for the 1,000 particle system (above) and 2 graphs for the 64,000 particle system (below)

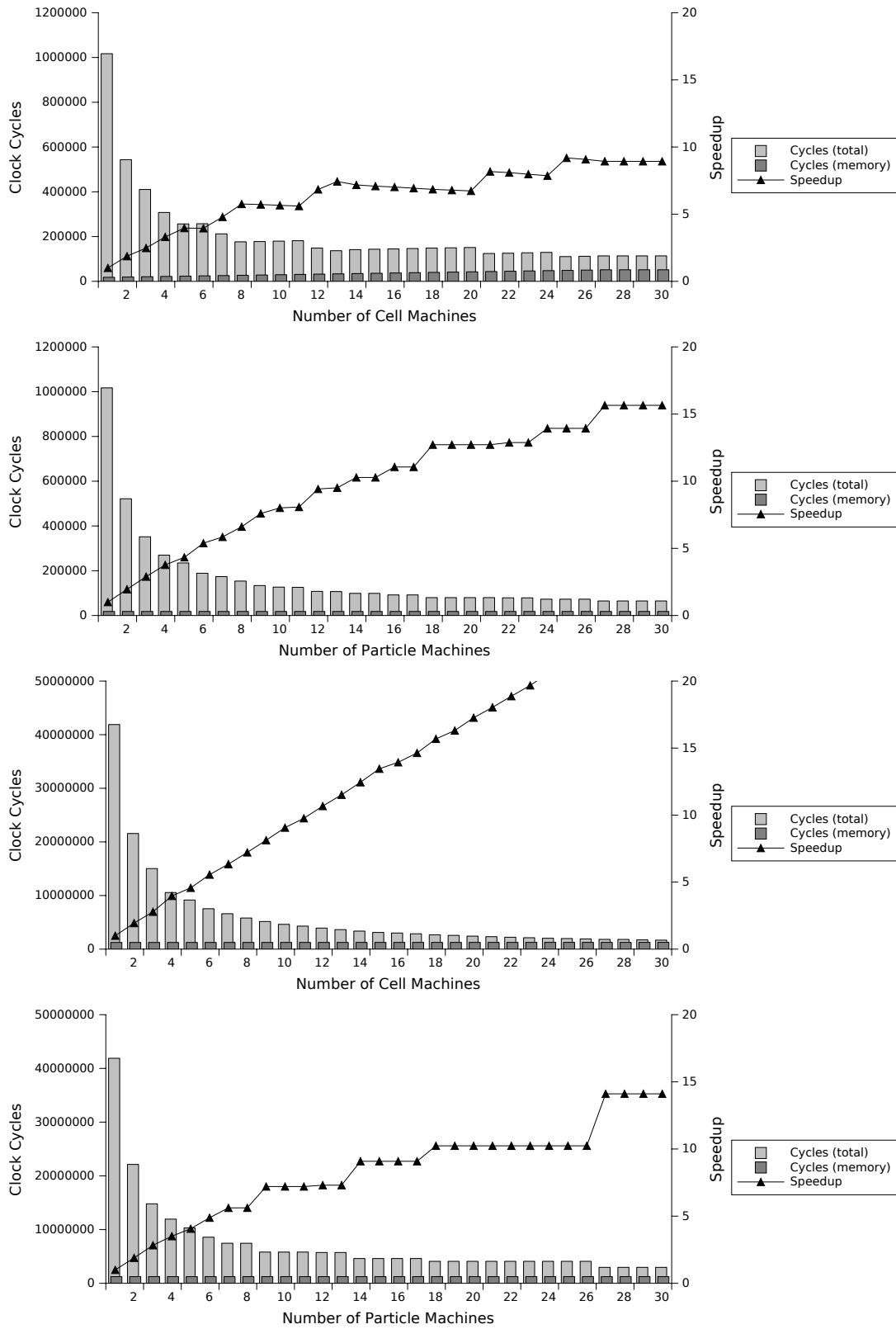


Figure 6.3: Simulation results for the segmented Hamilton path scheduler: 2 graphs for the 1,000 particle system (above) and 2 graphs for the 64,000 particle system (below)

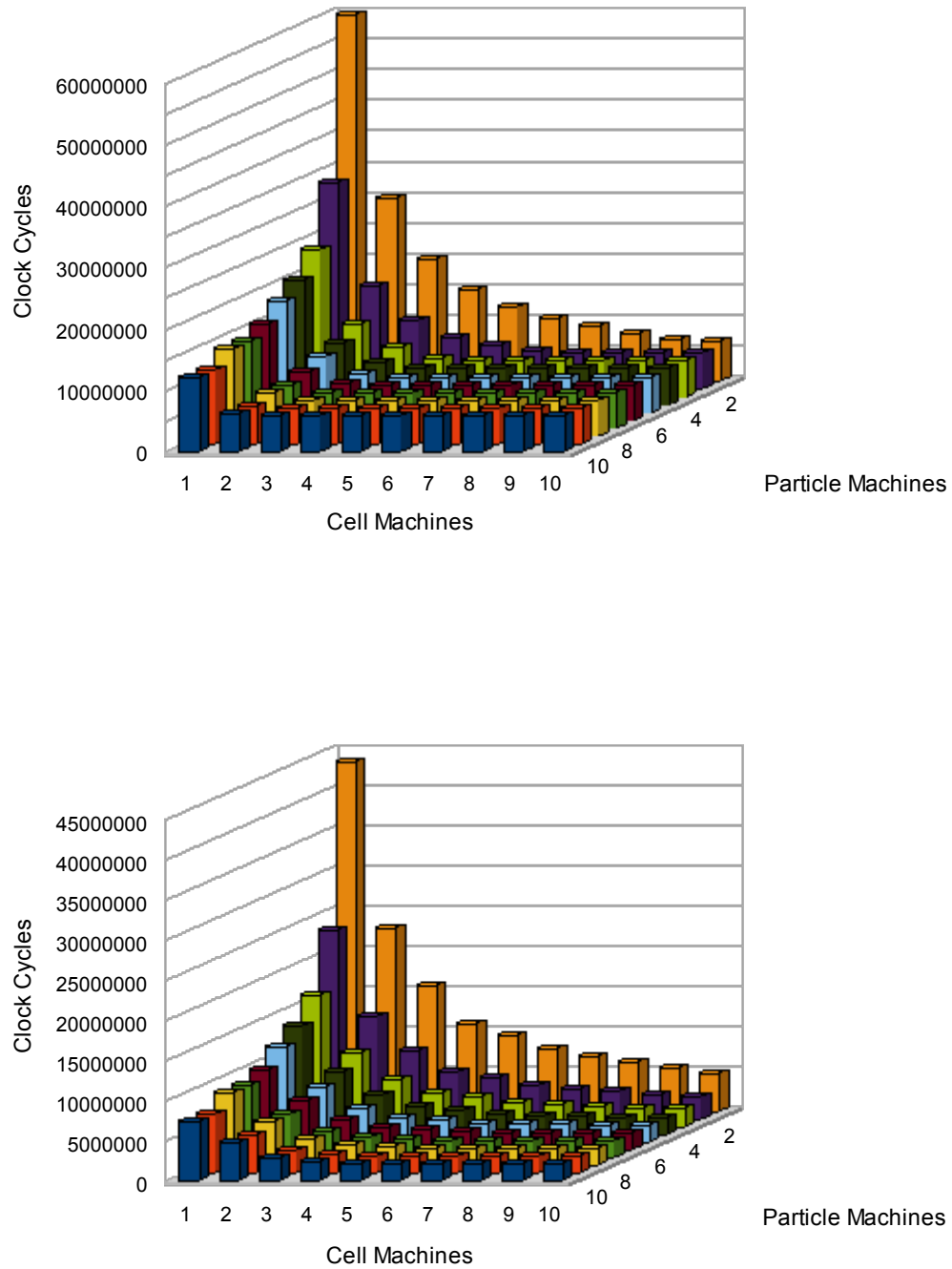


Figure 6.4: Running times when both the cell machines and particle machines are varied for the first come first served scheduler (above) and the segmented Hamilton path scheduler (below).

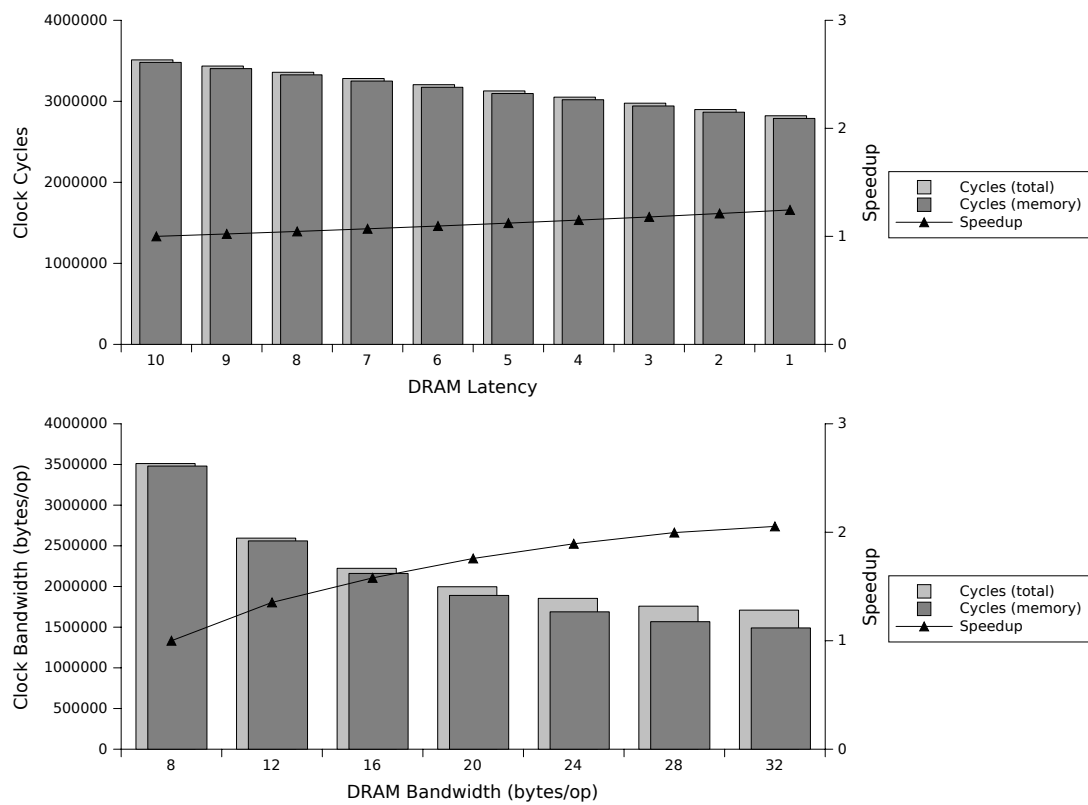


Figure 6.5: Simulation results for the first come first served scheduler, where the bandwidth and latency were varied.

## Chapter 7

# Hardware implementation

This chapter describes an implementation of the architecture presented in chapter 5 on the Digilent XUP board containing 1 Xilinx Virtex 2 Pro 30 FPGA. The goal of the implementation is to show that the architecture can be build, and that it can performs as advertised.

The implementation was written in 1800 lines of Verilog 2001 and mapped to hardware using the ISE 8.1 and EDK 8.1 tool-chains. At each level in the hierarchy, the design is validated against a reference code running on a CPU to show that the results are identical, or that they differ only by a very small factor. Validation was done with custom test benches in the Xilinx ISE 10.1 simulator.

All the arithmetic was implemented using the Xilinx floating point core library. This library can generate floating point operators (adders, subtracters, multipliers, dividers, square rooters, comparators) with arbitrary mantissa and exponent widths. We have chosen to do every computation in single precision floating point (meaning a 24 bit mantissa and 8 bit exponent), to be able to validate the results against a software implementation. The generator floating point cores comply with the IEEE 754 standard, except that denormalized numbers are not supported. These are the smallest numbers that are representable in floating point and we do not expect to encounter them during an MD simulation with normalized units.

The floating point library also can pipeline the generated operators up to a certain depth (depending on the operator), to decrease the cycle time. Initially we generated only combinational operators, but later we added pipeline stages to some operators to speed-up the design. A complete overview of the speed and area used by each operator can be found in appendix A. The floating point library is a core generator, meaning that it generates an optimized net-list (essentially a black box) that is not meant to be read by the user. The resulting cores can therefore not be modified.

### 7.1 Particle machine

The particle machine is implemented as a pipeline of units as shown in figure 5.2. The units themselves are internally pipelined as well, such that there is a register stage in between any 2 successive arithmetical units. In the particle machine, there is room for 1 reference

particle only, so the pipeline must be drained each time the reference particle is switched. Long delays (such as the  $\mathbf{d}$  vector that comes out of the distance unit and must be used in the force accumulation unit) are saved using shift registers that can be implemented efficiently by exploiting the LUTs as 16 stage shift registers.

The pipeline has 2 control signals (tokens) that are present in each stage. First there is the *valid* signal, which indicates whether the current data is valid. If it is invalid, we proceed with the computation anyway, except that we don't accumulate the resulting force value to the force for the current reference particle. Second there is the *trans* signal, which is high during the computation of the force for 1 reference particle. In between each 2 consecutive reference particles, there should be at least 1 clock cycle during which this signal is low, so that units further down the pipeline know that an output has to be produced. When *trans* is low, *valid* must be low as well. We will now discuss the individual units in more detail.

### 7.1.1 Force computation

The force computation is implemented as a direct mapping of the expression to a data flow graph. The alternative is to make a generic force evaluator that can compute any force that is defined as a piecewise polynomial. Any force can be approximated this way by making the pieces small enough and/or making the polynomial order high enough. The coefficients of all these pieces are then stored in a small memory and read out during computation. This scheme can be very useful if expressions involving e.g.  $\sin$  or  $e$  must be computed. However, it costs extra memory and is more difficult to design. Furthermore we can rewrite the force expression to obtain something that is far easier to implement in hardware (this optimization is also used in the LJPave code):

$$\begin{aligned}
& F^{LJ}(r) \frac{\mathbf{r}}{r} \\
&= \\
& \frac{48\epsilon}{r} \left( \left( \frac{\sigma}{r} \right)^{12} - 0.5 * \left( \frac{\sigma}{r} \right)^6 \right) \frac{\mathbf{r}}{r} \\
&= \\
& \frac{48\epsilon}{\sigma^2} \left( \left( \frac{\sigma}{r} \right)^{14} - 0.5 * \left( \frac{\sigma}{r} \right)^8 \right) \mathbf{r} \\
&= \left\{ \epsilon' = \frac{48\epsilon}{\sigma^2} \right\} \\
& \epsilon' * \left( \frac{\sigma}{r} \right)^2 * \left( \frac{\sigma}{r} \right)^6 * \left( \left( \frac{\sigma}{r} \right)^6 - 0.5 \right) \mathbf{r} \\
&= \left\{ \alpha = \left( \frac{\sigma}{r} \right)^2 \right\} \\
& \epsilon' * \alpha * \alpha^3 * (\alpha^3 - 0.5) * \mathbf{r}
\end{aligned}$$

Because the  $r^2$  term is a natural result of the distance computation, we don't need to do a square root operation either. Furthermore,  $\sigma^2$  can be precomputed, so that only the division

to compute  $\alpha$  remains. A block diagram of the implementation of the resulting expression is shown in figure 7.1 (pipeline registers in between operators have been omitted). The  $r$  factor is not included in this unit; it is part of the force accumulation unit that comes right after force computation.

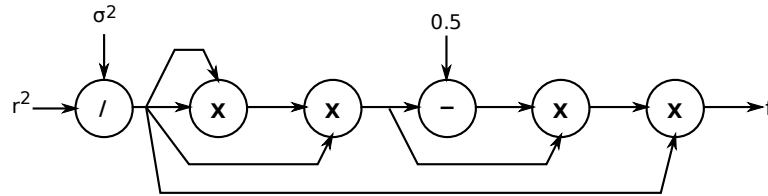


Figure 7.1: Block diagram of the Lennard-Jones force unit.

### 7.1.2 Distance computation

As described above, we can omit the square root operation in the distance computation. The component-wise distances have to be wrapped to comply with the minimum image convention. To facilitate this, we select a correction factor for each distance component, based on 2 guards for that component. If the distance is larger than half the universe size ( $0.5 * L_x$  for the  $x$  dimension), we have to subtract the universe size out, so the correction factor is the negative of the universe size. Similarly if the distance is less than minus half the universe size, the universe size has to be added to it. Otherwise, the correction factor is zero. Figure 7.2 shows parts of the block diagram for this unit. On the left, the part that computes the component-wise distance in the x-dimension  $r_x$  is shown. Identical units are instantiated to compute  $r_y$  and  $r_z$ . The output is then squared and summed up, as shown on the right. The values for  $L_x$ ,  $L_y$ ,  $L_z$  and their halves are provided by the environment. The negatives are obtained by flipping the first bit, which is allowed because IEEE 754 floating point uses a sign and magnitude representation.

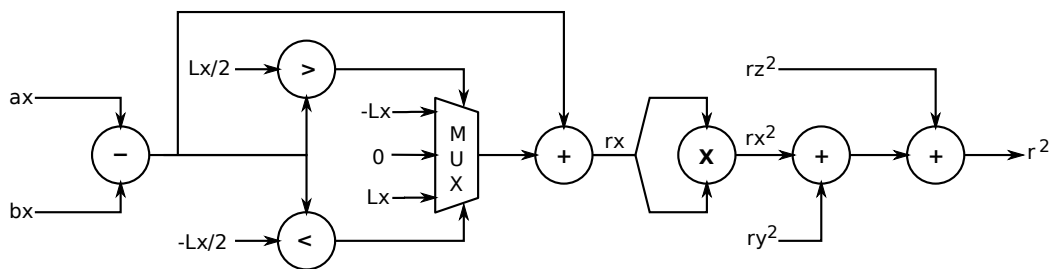


Figure 7.2: Block diagram of the distance unit ( $r_y^2$  and  $r_z^2$  computation omitted).

### 7.1.3 Cut-off check

The cut-off check is trivial: it is just a comparator that checks whether the squared distance  $r^2$  is smaller than the squared cut-off radius  $r_c^2$ . We assume  $r_c^2$  is precomputed and provided

by the environment, to save a multiplier. The output of the comparator is a valid signal that is combined with the valid signal that is input to the component using a logical **and**, to generate the output valid signal. Thus, this component acts as a sieve that sets the valid signal to zero whenever particle pairs are too far apart. Figure 7.3 shows the block diagram of this component.

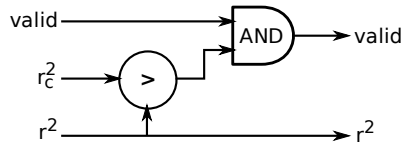


Figure 7.3: Block diagram of the cut-off check unit.

#### 7.1.4 Force accumulation

Force accumulation seems rather simple: just multiply the piecewise distances  $r_x, r_y, r_z$  with the force  $f$  to obtain the force vector, then accumulate it (see figure 7.4). This is indeed simple when the adder is implemented without any internal pipelining: it can then use the output of the previous computation for the next one. However, when the adder is pipelined, it cannot, because the previous computation isn't finished yet when the next one starts. We can solve this by adding more accumulation registers, that each store partial results, as described in [25].

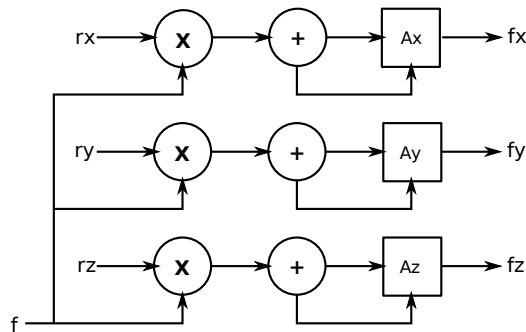
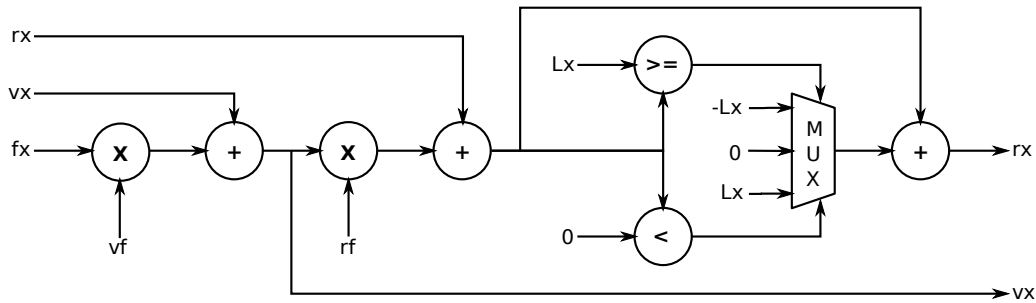


Figure 7.4: Block diagram of the force accumulator unit.

#### 7.1.5 Force integration

The force integration step is implemented as a direct mapping of the expression to a data flow graph as shown in figure 7.5. The  $vf$  and  $rf$  factors are the factors used for integration that are assumed to be precomputed.  $rf = \Delta t$ ,  $vf = 48 * \epsilon / \sigma^2 * \Delta t$ . This system is replicated 3 times for  $x, y$  and  $z$ . The right part takes care of readjusting the positions such that they fall inside the universe. Note that the LJPave reference code doesn't do this. Instead, it lets particles drift outside of the universe, and then compensates for this when computing the distance between 2 particles.

Figure 7.5: Block diagram of the integrator unit for  $rx$  and  $vx$ .

### 7.1.6 Control

The particle machine contains a small control element, that generates the inputs to the pipeline (in particular the `valid` signal), based on the input bus data. Furthermore, it contains the register for the reference particle that is loaded from that same input base. The output of the integrator unit is directly passed into the output bus.

### 7.1.7 Performance

The synthesized design occupies the majority of the Virtex 2 Pro 30's resources (see table 7.1): approximately 76 % (10,426) of the slices are used, primarily their LUTs (60%) and not their registers (11%). This suggests that there is a lot of room for pipelining. The numbers from appendix A support this. Note that there are enough multipliers to fit 2 particle machines in 1 FPGA, but the LUT usage prevents this.

Resource	Usage	Virtex 2 Pro 30	Percentage
Number of slices:	10,426	13,696	76%
Number of registers:	2,940	27,392	11%
Number of 4 input LUTs:	16,560	27,392	60%
Number of MULT18x18s:	64	136	47%

Table 7.1: Resource usage of the particle machine implementation.

The design is limited by the speed of the combinational multiplier, which has a cycle time of 79.367 ns, and thus a maximum operating frequency of 12.60 Mhz. According to appendix A, the divider can be pipelined up to 28 stages to perform at 288 Mhz.

### 7.1.8 Validation

The first 4 stages of the pipeline: distance computation, cut-off check, Lennard-Jones force computation and force accumulation were simulated in unison against a trace from a 1000 particle simulation using a custom test bench and a Verilog simulator. The simulator output was then compared against a software implementation that computes the reference outputs based on the same trace. The results were (bit-wise) identical.

The whole particle machine was validated with a difference test bench, but with much fewer input particles, to prevent excessive simulation times. This time the output was again bit-wise identical to the output of the reference code.

### 7.1.9 Optimization

Because the divider is the bottleneck in the performance, it makes sense to optimize it first. The next slowest operator is the adder, which has a cycle time of 19.814 ns. So we decided to pipeline the divider using 6 internal stages. This reduces its cycle time to 18.09 ns, meaning that the system should now be limited by the speed of the adder. After synthesis of the new design, the stated cycle time is 21.283 ns. This is slightly higher, but is explained by the fact that in several parts of the design, an adder is preceded by a MUX that also adds a little to this number. A cycle time of 21.283 ns means a maximum clock frequency of 46.986 Mhz. This is enough to integrate a particle machine into an EDK design (see section 7.3), which requires a minimum system frequency of 25 Mhz. Other frequencies for certain components are possible, but then we would need to support multiple clock domains, which is error prone.

## 7.2 Cell machine

An implementation of the cell machine was built that supports only 1 particle machine. However, it is relatively straightforward to extend this cell machine to support multiple particle machines. The cell machine has 3 main components. Firstly there is the cell memory to locally buffer a cell, its environment and its new state. Secondly there is a set of registers to set the constants used during computation. And thirdly there is a control unit that communicates with the particle machines over the buses.

The cell machine was implemented as a PLB (processor local bus) device, which is the main high speed system bus for a PowerPC based design. This will make it possible to run a program on the PowerPC processor that tests the design. To make it easier to attach a device to the PLB, the Xilinx PLB IPIF (IP Interface) module was used, that automates certain tasks, such as address decoding. The design is such that the PLB part is separate from the control part, so that it is relatively easy to rewrite the cell machine to support other buses. The PLB is a on-chip bus only, if multiple FPGAs are to be supported, something else has to be used. See figure 7.6 for the cell machine's block diagram.

### 7.2.1 Cell memory

There is room for 27 input cells and 1 output cells in the current implementation of the cell machine. Both memories are implemented as a set of BRAM blocks as generated by the Xilinx dual port block memory generator. Due to limitations in this generator, the sizes of the 2 ports can only differ by a multiple of 2, and due to the 64 bit nature of the PLB, the one of the ports is restricted to 64 bit. It was therefore decided to make the other port 128 bit wide (where  $32*3 = 96$  would have been the ideal choice). On the 128 bit side, particle positions and velocities are then interleaved in memory: positions on the even addresses and velocities

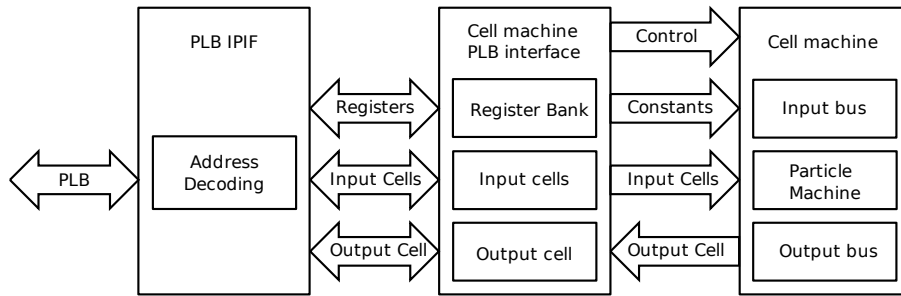


Figure 7.6: Block diagram of the cell machine implementation.

on the odd addresses. 32 bits for each data element are wasted in the current design. In the future, these extra bits can be used to store additional information of the particle such as its type and charge.

Both cell memories are made available to the PLB as 64 bit (8 byte) memories, that are mapped directly into the address space of the system. Extra logic was added to support writing to only a selected number of bytes, so that it is possible for the 32 bit PowerPC processor (or any other 32 bits component) to fill the cell memories.

Cells are stored as arrays inside this memory, with fixed offsets. Each cell can store 128 particles, which is enough to store a target of 125 particles per cell. More particles per cell is simply not possible on the Virtex 2 Pro 30 FPGA.

### 7.2.2 Constant registers

All the constants that are used during the computation by the particle machines should be available to the cell machine. To achieve this, they are stored in a set of registers in the PLB interface component. The alternative is to hard-code them at design time. This means that whenever these constants are changed, a new hardware design must be generated. Although this is possible in theory, it is not well supported in the tools that have been used.

Besides registers for constants, there are also 3 registers to control the cell machine's behavior. There is a write-only register that is used as a start trigger for the cell machine: if a 1 is written to it, the cell machine begins to compute the new state for the input cell. Then there is the read-only register that indicates whether the cell machine is finished with the computation. Finally, there is the finish trigger register, that resets the cell machine to its initial state upon writing a 1 to it.

A third group of registers is provided to store the number of particles contained in each cell. This means 28 registers are required for this task alone. The number of particles in the input cells are written by the environment (the MD machine), and the number of particles in the output cell is written by the component itself.

### 7.2.3 Control

The cell machine's control is relatively simple. It is currently only supporting 1 particle machine, but extending it for multiple particle machines should be relatively easy. It is implemented as a state machine that goes through 7 states: `IDLE`, `LOAD-R`, `LOAD-V`, `EXEC`, `STORE-R`, `STORE-V`, and `DONE`, see also figure 7.7.

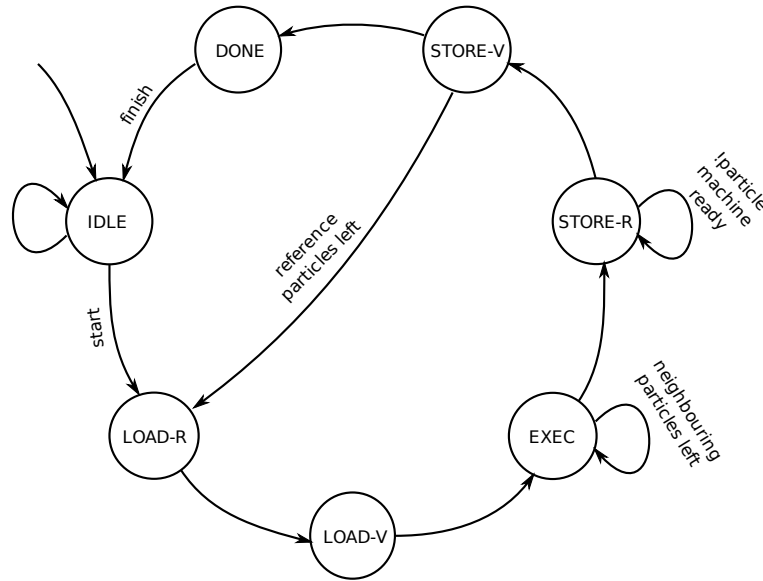


Figure 7.7: The cell machine control state machine.

The cell machine starts in the `IDLE` state, and then waits for the start trigger from the environment (provided by the start register from the PLB interface in the current implementation). Then it starts to load a reference particle in the only particle machine there is. This requires 2 separate states (states `LOAD-R` and `LOAD-V`), because positions and velocities are stored at separate memory addresses. Once a reference particle is loaded, all neighboring particles are send one after the other to the particle machine (state `EXEC`). When all neighboring particles have been send, the cell machine waits for the particle machine to finish (i.e. to be drained and produce an output), upon which it immediately writes the position back into memory (state `STORE-R`) and the state after that it writes back the velocity to the next memory address (state `STORE-V`). It then determines whether there are reference particles left to be processed. If not, it goes to the state done, otherwise it goes to state `LOAD-V` again. Once all reference particles are processed, state `DONE` will be reached, and the new state for the current cell is ready in the output memory. The environment can read this new state, and then reset the cell machine to its initial state by setting the finish trigger. The cell machine is now ready to perform a new computation.

### 7.2.4 Performance

The synthesis report states that 87% of the slices are used (see table 7.2), a mere 11 percent increase as compared to the particles machine alone. In theory the BRAM resources are large

enough to implement 2 cell machines on the FPGA, but other subsystems also require some BRAM (e.g. to store PowerPC code and data), so it might or might not fit depending on the requirements of these other components).

Resource	Usage	Virtex 2 Pro 30	Percentage
Number of slices	11,958	13,696	87%
Number of registers	4,985	27,392	18%
Number of 4 input LUTs	19,240	27,392	70%
Number of BRAMs	54	136	39%
Number of MULT18x18s	64	136	47%

Table 7.2: Resource usage of the cell machine implementation.

The synthesized design has a cycle time of 22.927 ns, which means a maximum clock of 43.617 Mhz. The critical path is reading from the BRAM, and then immediately doing a subtraction for the distance. So to improve the speed of the design, we need to pipeline the adders first.

### 7.2.5 Validation

The cell machine was validated using a custom test bench that computes the new state of 1 cell containing 8 particles and an environment containing roughly 300 particles. The output was then compared against the output of a software program. The results were again bit-wise identical.

## 7.3 MD machine

A full MD machine wasn't constructed. A system was constructed that contains the core of a simple MD machine. It is based on the PLB (processor local bus). In a real MD machine, all cell machines, the main memory and the scheduler should be connected to this bus. In our demo system, there is just 1 cell machine, there is only a small portion of on-chip memory that is available to store cells, and the scheduler is executing on the CPU. The block diagram of this system is shown in figure 7.8.

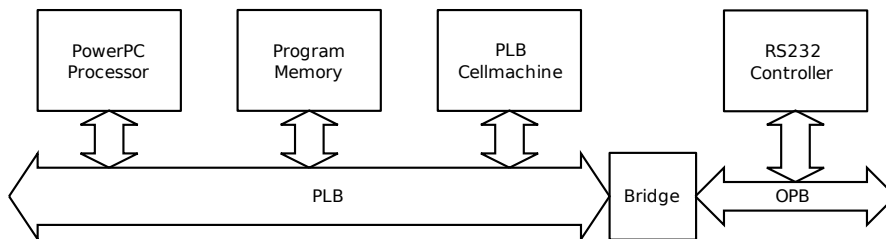


Figure 7.8: Block diagram of a prototype MD machine implementation.

The user interface is provided by a RS232 serial port that exports a simple menu based interface. This interface can be accessed on a regular PC by a terminal program. Currently,

the application only allows the setting of all the constants that are used during computation, and to execute a small test case to verify that the cell machine is indeed working.

### 7.3.1 Performance

The resource usage of the complete system is dominated by the cell machine, see table 7.3. The logic required for the PLB, the OPB, The RS 232 interface and the program memory is relatively small compared to the cell machines arithmetical operators. Note that the number of slices occupied by the complete system is lower than for the cell machine alone. This is probably because ISE 8.1 uses a synthesis and map algorithm that is able to map more LUTs into the same slice. Note also that 8 BRAMs have been allocated for the program memory. This would allow us to add another cell machine, or to extend the memory capacity of the existing cell machine.

Resource	Usage	Virtex 2 Pro 30	Percentage
Number of slices	11,929	13,696	87%
Number of registers	5,591	27,392	20%
Number of 4 input LUTs	19,643	27,392	71%
Number of BRAMs	62	136	45%
Number of MULT18x18s	64	136	47%

Table 7.3: Resource usage of the MD machine implementation.

After place and route, the maximum delay in between 2 consecutive registers is 31.155 ns. This means, the system can run at 32.10 Mhz, which is enough, because the lowest possible system clock is 25 Mhz.

### 7.3.2 Validation

The complete system wasn't simulated in the simulator. Instead, 2 test programs were written to test the design on the FPGA. The first one simply checks whether the cell machine's registers and local memory space work, while the second one checks whether a real simulation test run consisting of a hand full of particles is working.

Unfortunately, the current cell machine implementation contains a bug that always makes the Y component of both the position and the velocity 0 in the output. The problem most likely is occurring while reading the input cell memory. If all y components are set to zero in the reference code, the output that is produced by the cell machine is actually correct.

The clock speed of the actual implementation is only 25 Mhz, meaning it can do at most 25 million particle-particle evaluations per second. This is a speed-down of a factor of 2.25 as compared to the desktop CPU that we introduced in chapter 5.

## Chapter 8

# Conclusion

The goal was to design an FPGA based architecture that was able to run future MD simulations in a week, given enough hardware resources. Another goal was to demonstrate a working implementation of one instance of the architecture on an actual FPGA board.

The machine architecture that was developed and presented in chapter 5 was able to scale up to moderately large particle systems (in the order of tens of thousands of particles), not enough to reach the goal to simulate millions of particles efficiently. Further scalability can be achieved however by replicating the system many times and giving each its own group of cells to work on. The main bottleneck of the current architecture is the memory bus in the MD machine that is shared among all the cell machines. Even for very fast memories, this architecture is only able to keep 2,000 particle machines busy, while almost 300,000 are required.

The hardware simulator presented in chapter 6 was a valuable tool, primarily as a prototype of the hardware implementation. It also was able to show that the architecture was functionally identical to the C reference code. The performance estimates that it gives back have also been very useful in understanding the behavior of the architecture under a variety of conditions.

The demo implementation that is presented in chapter 7 can perform parts of a real MD simulation. There was not enough time to build a fully fledged MD machine, nor was there time to optimize the system to be able to reach the estimated speeds of chapter 5. The current system runs at 25 Mhz, only a quarter of the intended 100 Mhz.

The molecular dynamics algorithm lends itself moderately well to an FPGA mapping. An MD algorithm requires lots of computation and memory bandwidth, which the FPGA has plenty of. On the other hand, floating point is more or less required, which is a bad match for the FPGAs logic and fixed point optimized-structures. The proposed architecture when run on 1 FPGA should be able to outperform a CPU based implementation on contemporary technology by a factor of 2.

If FPGAs are to be successful at accelerating scientific simulations, it should become easier to program them. Currently, this requires extensive knowledge of a hardware description language such as Verilog, and familiarity with the internals of modern FPGAs. A more expressive hardware description languages or a compiler that efficiently maps a programming languages to hardware is required to facilitate this.

## 8.1 Further work

This work can be taken into many different directions. The end goal would be to actually perform future state of the art simulations in at most a week on a sufficiently large network of FPGAs, This report is only the first step towards that goal. The following plan could be followed towards this goal:

1. The main short term goal would be to build an actual MD machine consisting of one cell machine, that in turn consists of 1 particle machine that would be able to perform a simple MD simulation. The current system can almost do this already. What is currently lacking is a DRAM controller to access the on board DRAM to store the simulation state, and a software program that serves as the scheduler. The erroneous behavior that is currently exhibited by the cell machine must be repaired as well.
2. Once a complete MD machine has been built, it can be optimized in many ways. Firstly, the particle machine can be pipelined further to be able to run at 100 Mhz, which is the maximum speed for the PLB, and the requirement that the pipeline must be drained in between each 2 reference particles can be relaxed. Then the software program on the CPU could be replaced by a fast DMA controller that is programmed by the CPU. The resulting system would be a very fast implementation of a single particle machine, that can run at 100 Mhz and is able to perform as predicted in chapters 5 and 6.
3. Once a fast MD machine has been built, the machine could be made generic in the number of cell machines and particle machines required. The scalability of the architecture can then be demonstrated. However, this would certainly require a bigger FPGA than the Virtex 2 Pro 30.
4. At this stage, the architecture could be changed to make more efficient use of the available hardware resources, in particular the implementation of the particle machine. By sharing force computation and integration hardware, extra particle machines can be added for a relatively moderate cost.
5. Then it is time to make the simulation more realistic, starting with supporting multiple particle types in the Lennard-Jones force pipeline. After that, bonded forces and electrostatic forces could be added to the architecture.
6. Finally, to build a large FPGA based system that is able to perform future state of the art simulations within a week. This would certainly require many communicating MD machines. The simulator could be extended to handle such systems to be able to give performance estimates.

# Appendix A

## FPGA arithmetical operations

The Xilinx Floating point generator 3.0 was used to generate the floating point operators for every design in this report. This generator can generate all major floating point operators, and can generate them for any desired mantissa and exponent widths. Furthermore, it can pipeline the operators up to a certain depth. The more pipelining is used, the higher the obtainable clock speed. However, more pipeline stages require more registers, which results in more resource usage.

The following figures show the area (in slices) and the performance (in Mhz) of the various floating point operators as a function of the latency (in number of pipeline stages). The efficiency (in Mhz/slices) is also shown. Note that the performance is a synthesis estimate. The actual implementation could be slower because of additional routing delays.

From this data, we conclude the following:

- Adding more pipeline stages increases the efficiency in most cases. This is because the registers in the slices go unused in combinational design. By enabling them one by one, the performance will increase, while the number of slices will remain constant.
- The embedded multipliers improve efficiency by a factor of about 3. This is not exceptional considering that the efficiency doesn't take the multipliers into account for the occupied area.
- To obtain efficient floating point adders and square root operations, a lot of pipelining must be used. Even then they are not as efficient as the other operators and should therefore be avoided where possible.
- Comparators are very small and efficient. This is due to the fact that floating point comparisons are almost the same operation as for fixed point representation. It is probably a good idea to use them instead of more complex formulas without compare operations that obtain the same result.

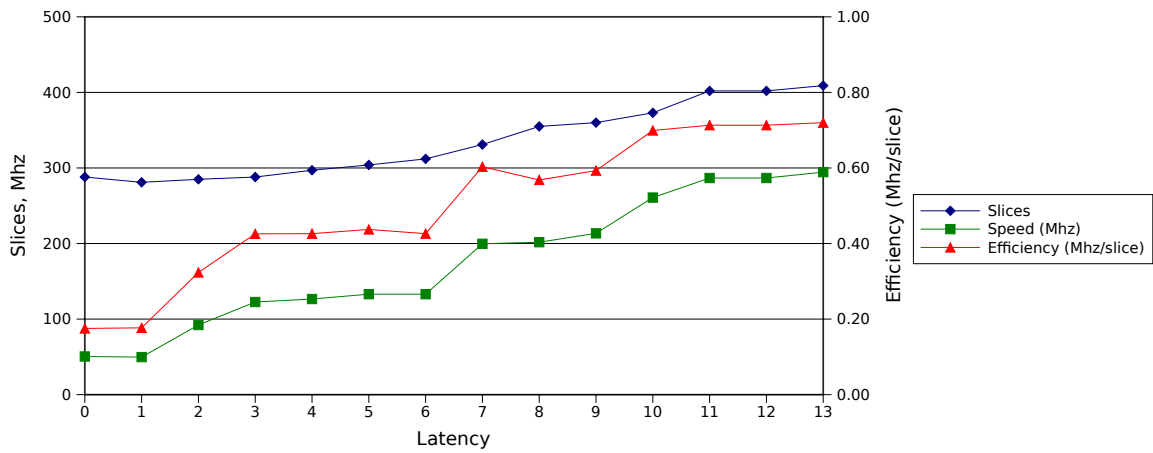


Figure A.1: Efficiency of single precision floating point adder in LUTs

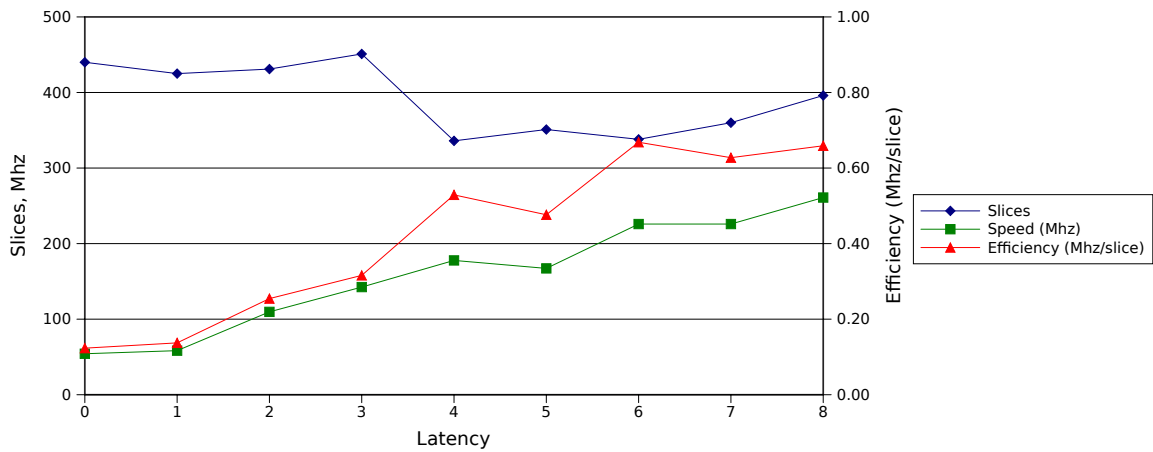


Figure A.2: Efficiency of single precision floating point multiplier in LUTs

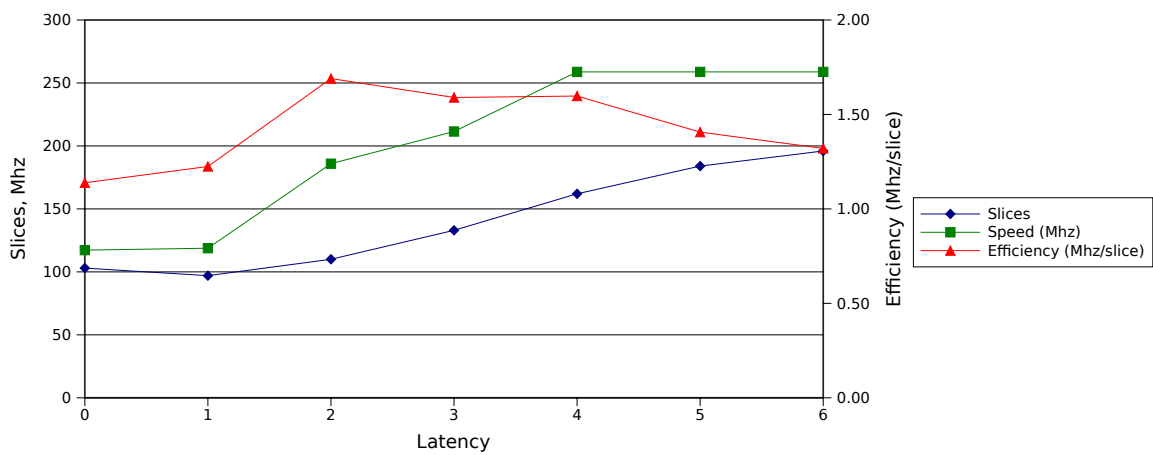


Figure A.3: Efficiency of single precision floating point multiplier in MULTs

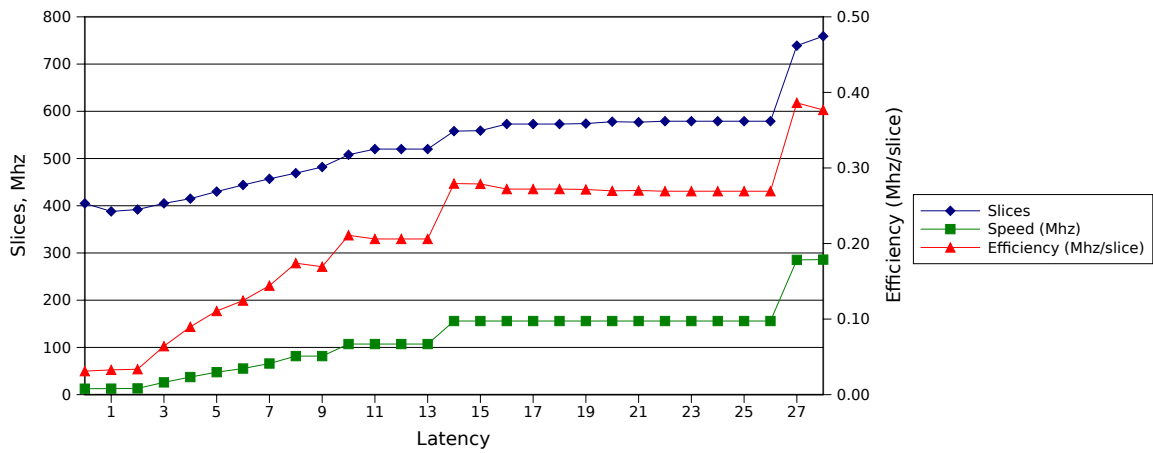


Figure A.4: Efficiency of single precision floating point divider in LUTs

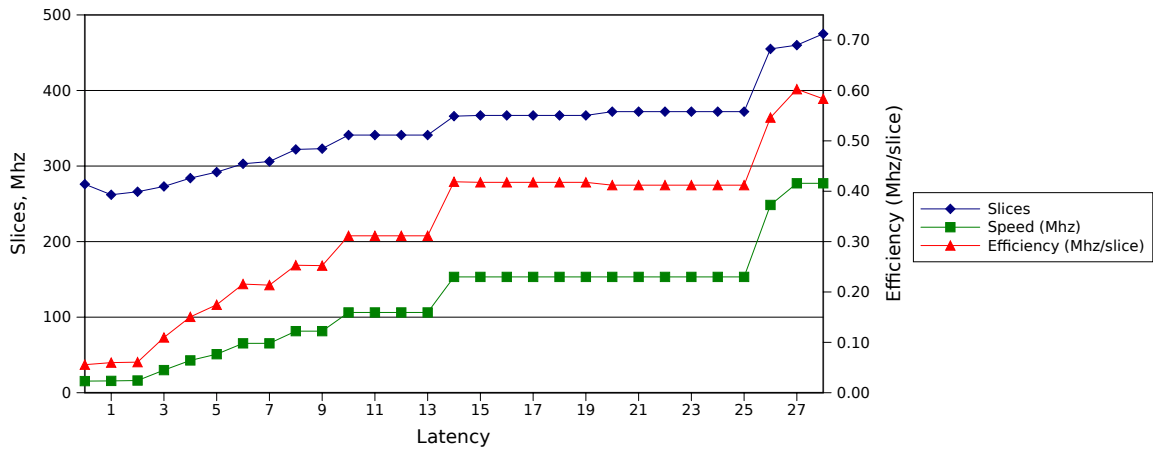


Figure A.5: Efficiency of single precision floating point square root in LUTs

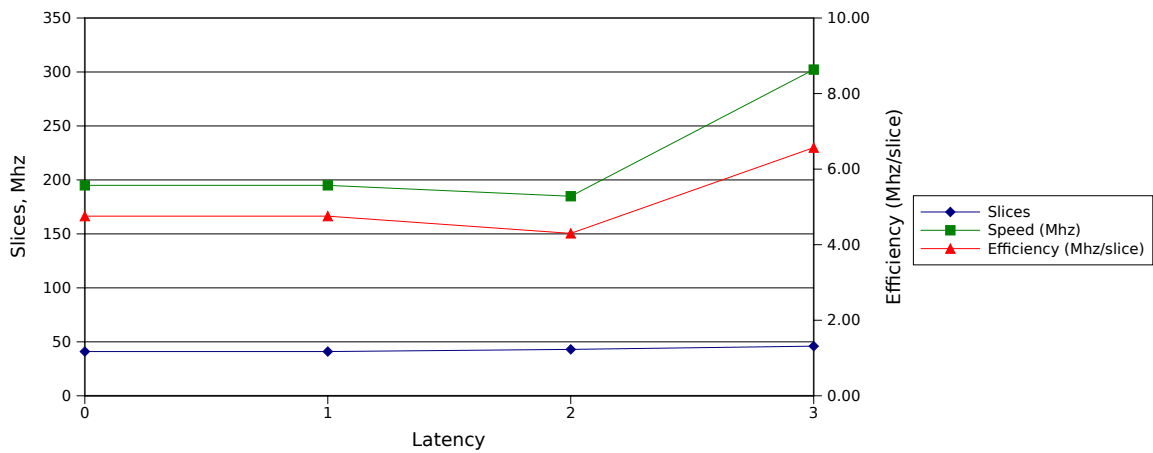


Figure A.6: Efficiency of single precision floating point comparator in LUTs

# Bibliography

- [1] Altera corporation. <http://www.altera.com>.
- [2] Annapolis micro systems inc. <http://www.annapmicro.com>.
- [3] Berkeley BEE. <http://bwrc.eecs.berkeley.edu/Research/BEE/>.
- [4] Digilent inc. <http://www.digilentinc.com>.
- [5] Xilinx inc. <http://www.xilinx.com>.
- [6] Takashi Amisaki, Takaji Fujiwara, Akihiro Kusumi, Hiroo Miyagawa, and Kunihiro Kitamura. Error Evaluation in the Design of a Special-Purpose Processor That Calculates Nonbonded Forces in Molecular Dynamics Simulations. *Journal of Computational Chemistry*, 16(9):1120–1130, 1995.
- [7] Navid Azizi, Ian Kuon, Aaron Egier, Ahmad Darabiha, and Paul Chow. Reconfigurable molecular dynamics simulator. *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2004)*, pages 197–206, 2004.
- [8] Klaas Esselink. LJPaVe, a Molecular Dynamics program for sequential, vector and parallel computers. Technical Report AMER.94.004, Koninklijke/Shell-Laboratorium, May 1994.
- [9] Krste Asanovic et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, University of California, Berkeley, 2006.
- [10] Peter L. Freddolino, Anton S. Arkhipov, Steven B. Larson, Alexander McPherson, and Klaus Schulten. Molecular Dynamics Simulations of the Complete Satellite Tobacco Mosaic Virus. *Structure*, 14(3):437–449, 2006.
- [11] Maya Gokhale and Paul S. Graham. *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*. Springer, December 1995.
- [12] Yongfeng Gu, Tom Van Court, and Martin C. Herbordt. Accelerating Molecular Dynamics Simulations with Configurable Circuits. *IEE Proceedings on Computers and Digital Techniques*, 153(3):189–195, May 2005.
- [13] Yongfeng Gu and Martin C. Herbordt. Fpga-based multigrid computation for molecular dynamics simulations. In *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, pages 117–126, April 2007.

- 
- [14] Tsuyoshi Hamada, Toshiyuki Fukushige, Atsushi Kawai, and Junishiro Makino. PROGRAPE-1: A Programmable, Multi-Purpose Computer for Many-Body Simulations. *Publications of the Astronomical Society of Japan*, 52(5):943–954, 2000.
- [15] Junichiro Makino, Kei Hiraki, and Mary Inaba. GRAPE-DR: 2-Pflops massively-parallel computer with 512-core, 512-gflops processor chips for scientific computing. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing (SC 2007)*, pages 1–11, New York, NY, USA, November 2007. ACM.
- [16] Volodymyr Kindratenko and David Pointer. A case study in porting a production scientific supercomputing application to a reconfigurable computer. In *14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, pages 13–22, April 2006.
- [17] Sam Lee and Paul Chow. An FPGA Implementation of Reciprocal Sums for SPME. In *Proceedings of the 2007 International Conference on Engineering of Reconfigurable Systems & Algorithms, ERSA 2007*, pages 159–165. CSREA Press, June 2007.
- [18] Gerhard Lienhart, Andreas Kugel, and Reinhard Mnnner. Using Floating-Point Arithmetic on FPGAs to Accelerate Scientific N-Body Simulations. In *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2002)*, pages 182–191, 2002.
- [19] Erik Lindahl, Berk Hess, and David van der Spoel. GROMACS 3.0: A package for molecular simulation and trajectory analysis. *Journal of Molecular Modeling*, 7(8):306–317, 2001.
- [20] Makoto Taiji, Tetsu Narumi, Yousuke Ohno, Noriyuki Futatsugi, Atsushi Suenaga, Naoki Takada, and Akihiko Konagaya. Protein Explorer: A Petaflops Special-Purpose Computer System for Molecular Dynamics Simulations. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing (SC 2003)*, page 15, Washington, DC, USA, 2003. IEEE Computer Society.
- [21] A. J. Markvoort, A. F. Smeijers, K. Pieterse, R. A. van Santen, and P. A. J. Hilbers. Lipid-Based Mechanisms for Vesicle Fission. *The Journal of Physical Chemistry B*, 111(20):5719–5725, April 2007.
- [22] Jeremy S. Meredith, Sadaf R. Alam, and Jeffrey S. Vetter. Analysis of a Computational Biology Simulation Technique on Emerging Processing Architectures. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2007)*, pages 1–8. IEEE, March 2007.
- [23] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kal, and Klaus Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005.
- [24] Piet Hut and Jun Makino. Astrophysics on the GRAPE Family of Special-Purpose Computers. *Science*, 283(5401):501–505, 1999.

- 
- [25] Viktor K. Prasanna and Gerald R. Morris. Sparse Matrix Computations on Reconfigurable Hardware. *Computer*, 40(3):58–64, March 2007.
- [26] Ronald Scrofano, Maya Gokhale, Frans Trouw, and Viktor K. Prasanna. Hardware/Software Approach to Molecular Dynamics on Reconfigurable Computers. In *14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, pages 23–34, April 2006.
- [27] Ronald Scrofano and Viktor K. Prasanna. Computing Lennard-Jones Potentials and Forces with Reconfigurable Hardware. In *Proceedings of the International Conference on Engineering Reconfigurable Systems and Algorithms (ERSA 2004)*, pages 284–290, Los Alamitos, CA, USA, 2004. IEEE Computer Society.
- [28] Ronald Scrofano and Viktor K. Prasanna. Preliminary Investigation of Advanced Electrostatics in Molecular Dynamics on Reconfigurable Computers. In *Proceedings of the ACM/IEEE SC 2006 Conference on Supercomputing (SC 2006)*, pages 45–45, November 2006.
- [29] Guochun Shi and Volodymyr Kindratenko. Implementation of namd molecular dynamics non-bonded force-field on the cell broadband engine processor. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008)*, pages 1–8. IEEE, April 2008.
- [30] A. F. Smeijers, A. J. Markvoort, K. Pieterse, and P. A. J. Hilbers. A Detailed Look at Vesicle Fusion. *The Journal of Physical Chemistry B*, 110(26):13212–13219, 2006.
- [31] Keith D. Underwood. FPGAs vs. CPUs: Trends in Peak Floating-Point Performance. In *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays (FPGA 2004)*, pages 171–180. ACM, 2004.