Halide in Molecular Dynamics

by

Ricardo Gayle Jr.

B.S. of Electrical Engineering and Computer Science

Massachusetts Institute of Technology, 2022

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

Authored by: Ricardo Gayle Jr.
Department of Electrical Engineering and Computer Science
May 12, 2023

Certified by: Saman Amarasinghe
Professor, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Halide in Molecular Dynamics

by

Ricardo Gayle Jr.

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In many fields, especially biology and chemistry, it is important to understand how a collection of particles will interact with each other over some period of time. If only managing a system of two particles, it is simple enough to calculate the final positions of the atoms given their properties and the outside forces placed upon them. However, it is often the case that the system's size is several magnitudes larger; therefore, the task is handed off to computers and simulators.

Molecular dynamics, or MD, simulations tend to be extremely expensive, taking several weeks to compute less than a second's worth of real time. Two significant reasons MD simulations are time intensive are due to the complex loop structures and math required to observe each time step. More tools and research are constantly being developed to increase performance of these simulations.

In this thesis we introduce a tool from the image processing domain, Halide, and argue that Halide is a qualified candidate to efficiently implement MD simulations in the future. We rewrote a potential into Halide to achieve only a 20% slow down serially, which we are certain can reach parity with minimal changes to the code, and over 300% speed up when running in parallel. While it was challenging beginning to work with Halide and its limitations, we are still able to accomplish this performance and versatility writing 47% less code. Halide also makes the transformation to parallel scheduling trivial, whereas this is not the case in the original implementation. Halide was not able to represent all of the loop structures we wanted; however, we also suggest several additions and changes to Halide to make it more suitable to MD.

.

Thesis supervisor: Saman Amarasinghe
Title: Professor

Acknowledgments

I would like to express my sincere gratitude to the following individuals without whose support and guidance, this Master's thesis would not have been possible.

Firstly, I would like to thank my thesis supervisor, Saman Amarasinghe, and Teodoro Collins for their unwavering support, guidance, and encouragement throughout the past year. Their valuable feedback has shaped and strengthened the quality of this thesis.

I also want to thank Ngoc Cuong Nguyen for his invaluable support in understanding fast POD. His expertise and willingness to share his knowledge played a critical role in the success of this thesis.

I would like to extend my heartfelt thanks to my family, friends and loved ones for their support and encouragement throughout my academic journey. Their love provided me with the power to pursue this degree and achieve more than I expected. My friends helped me maintain a balance between my academic and personal life.

Thank you all for your valuable contributions, and for making this Master's thesis possible.

Contents

Chapter 1

Introduction and Background

1.1 Introduction

Molecular dynamics simulations are used to model how different atoms interact with each other in a particular system. This is especially important in biology and chemistry when trying to study the properties of some organism or material. MD can be used to determine how a system will react to some change or environment and even to design new materials with desired properties of conductivity, flexibility, strength, etc. Molecular dynamics simulations are an integral tool for many important chemical and biological problems; however, there is currently no efficient way to run molecular dynamics simulations naively as they follow the classic *n-body* problem, growing in complexity with *n*. Consequently, scientists implement molecular dynamics energy potentials of a specified body order, a finite n, to determine an estimate of the changes, with higher bodies providing more accurate results but being more compute and time intensive. Even with these approximations, molecular dynamics simulations are still time intensive. While it is often necessary to view several milliseconds worth of time steps, traditional technologies are only capable of computing time steps 9 to 12 orders of magnitude smaller. For example, a material might need to withstand heat over the course of seconds or minutes, but a molecular dynamics simulation will simulate a nanosecond in a day of real time.

Currently, the open-source C++ repository, LAMMPS, is widely used for molecular dynamics simulations [9]. LAMMPS is used because it is highly portable, free, easily modifiable and extendable, and already comes loaded with a wide variety of different potentials. Within the pipeline of LAMMPS there are several compute-heavy tasks which may be possible to optimize– most notably, computing the energies of the particle system and computing the derivatives of those energies needed to evolve the state. That is, computing the potential of the system and its derivative.

Halide is an open-source language created to easily manipulate image processing algorithms by order in which a computation occurs [7]. In this work, we explore the possibility of moving Halide to a different domain; we believe Halide can be used to rewrite many of the potentials in LAMMPS to achieve higher performance. So far, we have implemented a potential currently in LAMMPS called Fast POD (Principal Orthogonal Description) in Halide with only a 20% slowdown serially, but with many optimizations still on the table and within easy reach. When run in parallel over the atoms, we see a greater than 350% speed up. We expect to reach parity serially and improve upon the original fast POD while writing 47% less code that is more understandable. Our parallel implementation requires less than five extra lines of code to transform from serial to parallel, whereas the original C implementation would require non-trivial transformations, including the careful application of atomics and the reorganization of memory management. In fact, an author of the original code remarked at the elegance of our implementation!

To showcase Halide's possible success in this new domain, in this paper we will (1) discuss the similarities and differences between image processing and molecular dynamics and why Halide is a reasonable candidate to be applied to the MD domain, (2) tackle the rewriting of Fast POD into Halide, (3) review the various scheduling done to mimick the original implementation, and (4) review the results in comparison to the current Fast POD implementation and an optimized Halide implementation.

1.2 Background

1.2.1 Molecular Dynamics

Molecular dynamics simulations are prominent tools used to examine how a set group of particles or atoms will interact with each other over a set period of time given some model of governing physics, typically encoded in the form of one or more potentials. Each potential describes how particles will interact with each other by assigning an energy to a particle based on its neighbors. These potentials can be defined on n particles; however, the ones used most in molecular dynamics are defined on pairs, triplets, and quadruplets. The potential used in this thesis is a up to 7 body potential called Fast Proper Orthogonal Descriptors (POD), further described in [5].

Molecular dynamics simulations consider the forces that each particle exerts on one another using potentials to continuously find each particle's new location and velocity. This proves to be one of the most compute and time intensive processes. To find the updated energy and force per atom, it is often necessary to perform expensive computations consisting of trigonometry, linear algebra and stencils within even more complicated loops with possible symmetries. A general potential follows the structure shown in Figure 1-1.

```
energies[atoms] = [...]
forces[atoms][dim] = [...]
for atom in atoms:
    for pair in pairs[atom]:
        energies[atom] = (() => expensive_computation(atom, pair):
            for x in non-rectangular loop:
                return trig_linear_stencils(x, y)
        forces[atoms][dim] =
            more_expensive_computation(atom, pair)
```

Figure 1-1: MD General Algorithm

The different forces and networks between particles can be so complex that neural networks have recently been used to represent their states [3]. On top of their complexity, the computational costs and time required to run these simulations have made it common for supercomputers to perform these simulations [2, 6]. Recent developments in GPUs, however, allow for powerful simulations to be run more modestly by offloading compute-intensive work to the GPU; this has not solved the issue of how long these simulations take, though [8], meaning even small performance improvements are still valuable.

Other than physical/architectural improvements to reduce the time of running molecular dynamics simulations, improvements have also arisen from creating new and improving old potentials. It is possible to implement an optimized pipeline, rather than a naive one, which can result in a difference of days to weeks for some simulations.

1.2.2 LAMMPS

The most popular tool that performs these computations is Large-scale Atomic/-Molecular Massively Parallel Simulator (LAMMPS), an open source molecular dynamics code base with a focus on materials modeling [9]. It has become a popular tool for molecular dynamics as it provides a variety of interaction models for a multitude of materials.

One reason it is popular is because LAAMPS has pre-stored potentials for solid-state materials, soft matter, and coarse-grained systems to simulate interactions at the atomic scale. These potentials include multiple kinds of pairwise, machine learning and bond potentials. There are many other reasons for its popularity though; users are able to control simulations with details ranging from using a single CPU vs. a supercomputer, adding new custom potentials via input script, and more. The code's general ease to modify or extend is just another reason it continues to be such a prominent tool. However, it still stands to question whether or not it is possible to simply rewrite the potentials in LAMMPS more optimally. We argue that rewriting these potentials with Halide can lead to simpler code that is more readily changeable for optimizations.

1.2.3 Halide

Many tools and frameworks exist to help build optimized image processing pipelines; one of the most popular tools is Halide. Halide is an open-source domain-specific language created to handle the compute-heavy processing pipelines used in today's image processing applications by keeping a separation between algorithm and scheduling. Halide was created with the trade-offs between locality, parallelism and redundant computation in mind; therefore, the language was made to allow for different schedule representations to easily be compared to better understand the trade-offs. While values do need to be computed before being used, Halide allows the user to determine when and where the values of a coordinate are computed, where they are stored, and how long a value is cached and reused vs being recomputed. Changing where, when, and how these values are stored will not change the result of the algorithm; however, it can drastically change the performance [7]. As long as Halide's internal checks assert the result of the algorithm does not change given the scheduling, Halide takes the specified algorithm and schedule to generate LLVM assembly.

While Halide was designed with image processing in mind, many of its tools make it possibly viable for other domains. Image processing tends to involve many structures somewhat similar to those in MD: there are many loops involving linear algebra, stencils, and even a few irregular loops with much more complex input/output patterns. Despite the big irregularity in MD computations (the loop over neighbors), MD and image processing computations might sometimes have a lot in common.

Halide is somewhat capable of managing some variation of complex computations over non-rectangular loop domains, loop domains with sizes that vary with some variable, and loop domains determined by another function. The tool was not made with MD in mind; therefore, it can not represent everything possibly wanted. We still believe Halide can be used to rewrite potentials in molecular dynamics simulations for significant improvements to run time in the near future.

Chapter 2

Implementing Fast POD

In this chapter, we introduce fast POD as an example of how complex and compute intensive potentials can be. Despite the complexities, we are able to rewrite fast POD in Halide due to its various tools which manage complex loops and computations relatively well.

2.1 Introduction to the Fast POD Potential

The Fast POD potential is a $n \leq 7$-body potential, which would imply a loop over heptatuples, yet "scales linearly in complexity with the number of neighbors irrespective of the body orders" [5]. To achieve this, Fast POD first calculates one, two, and three body potentials, and then cleverly combines these using the properties of certain classes of polynomials to produce the higher body potentials. We will only lightly reference this math below, as motivation, but focus on understanding the organization of the implementation to better understand the transformation into Halide.

Like any potential, Fast POD finds the energy and force of every atom in the simulation per atom, as shown in Figure 1-1. At a high level, the work done per atom, a, is as follows (pseudo code for fast POD is referenced and can be found at the end of this section).

Bessel parameters, bessel degrees, and inverse degrees are given as inputs which help model the physical two body interactions between any two atoms. For every neighbor in a's region and for every set of specified bessel parameters, bessel degrees, and inverse bessel degrees, a set of coefficients is created, `rbft` in Figure 2-5, using multiple trig computations on the radial distances from a to the neighbor. With these coefficients, we can perform a matrix multiplication to obtain the radial basis functions, as shown in Figure 2-1, where K_r is the specified number of radial basis functions, j is the neighbor of a, Q is some pre-computed fitted process result, and R_k is the radial basis function. `rbf` and its derivative is then used to initialize the energy

$$R_k(r_{aj}) = \sum_{l=1}^{L} Q_{lk} rbft_l(r_{aj}), k = 1, ..., K_r \tag{2.1}$$

Figure 2-1: Obtaining radial basis as stated in [5]

and force of a, as shown in Figure 2-6. As this is the energy and force calculated given the distance between pairs of atoms, we consider this the 2-body update (Figure 2-2).

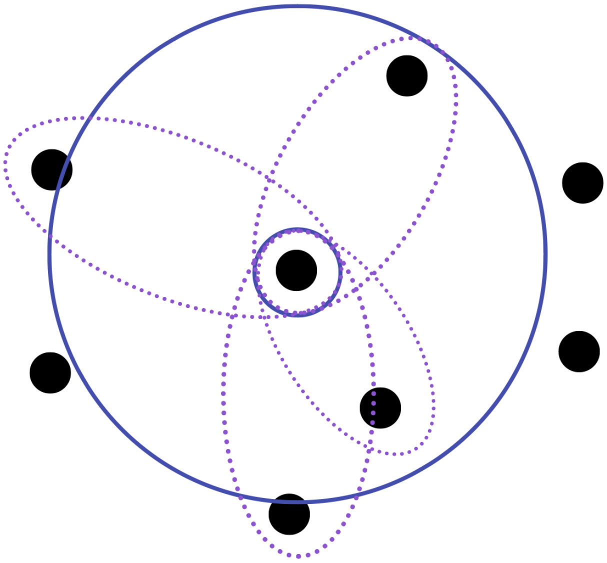

Figure 2-2: In this example, the center atom has 4 neighbors in its region.

Next, we derive an updated energy and force with respect to the 3-body instead (Figure 2-3). Similarly to before, some set of polynomials, `abf`, will be created given the angle between a and any other 2 atoms in its defined neighborhood. While this would naively be done in polynomial time with respect to number of neighbors, fast POD recursively builds a set of polynomials from a set of pre-computed monomials

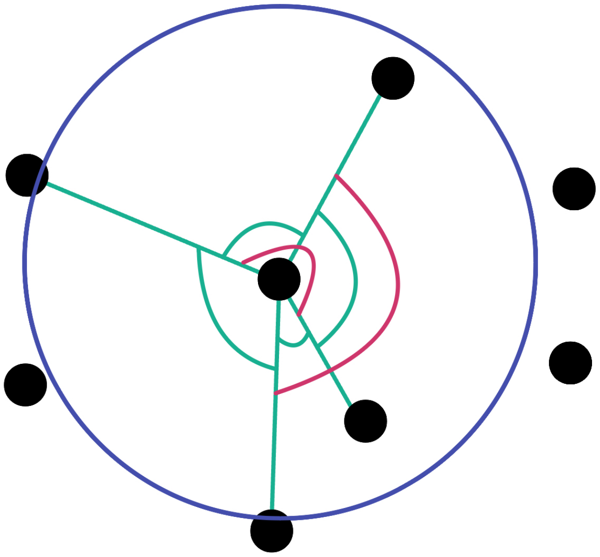

Figure 2-3: 3-body considers the angle between every unique triple with a at the vertex.

(Figure 2-4), allowing `abf` to be computed in linear time with respect to number of neighbors, as seen in Figure 2-16. Rather than updating energy and force with

0	1
1	$\hat{x}_{ij},\ \hat{y}_{ij},\ \hat{z}_{ij}$
2	$\hat{x}^2_{ij},\ \hat{y}^2_{ij},\ \hat{z}^2_{ij},\ \hat{x}_{ij}\hat{y}_{ij},\ \hat{x}_{ij}\hat{z}_{ij},\ \hat{y}_{ij}\hat{z}_{ij}$
3	$\hat{x}^3_{ij},\ \hat{y}^3_{ij},\ \hat{z}^3_{ij},\ \hat{x}^2_{ij}\hat{y}_{ij},\ \hat{x}^2_{ij}\hat{z}_{ij},\ \hat{y}^2_{ij}\hat{x}_{ij},\ \hat{y}^2_{ij}\hat{z}_{ij},$ $\hat{z}^2_{ij}\hat{x}_{ij},\ \hat{z}^2_{ij}\hat{y}_{ij},\ \hat{x}_{ij}\hat{y}_{ij}\hat{z}_{ij}$
4	$\hat{x}^4_{ij},\ \hat{y}^4_{ij},\ \hat{z}^4_{ij},\ \hat{x}^3_{ij}\hat{y}_{ij},\ \hat{x}^3_{ij}\hat{z}_{ij},\ \hat{y}^3_{ij}\hat{x}_{ij},\ \hat{y}^3_{ij}\hat{z}_{ij},\ \hat{z}^3_{ij}\hat{x}_{ij}$ $\hat{z}^3_{ij}\hat{y}_{ij},\ \hat{x}^2_{ij}\hat{y}^2_{ij},\ \hat{x}^2_{ij}\hat{z}^2_{ij},\ \hat{y}^2_{ij}\hat{z}^2_{ij},\ \hat{x}^2_{ij}\hat{y}_{ij}\hat{z}_{ij}\ \ \hat{x}_{ij}\hat{y}^2_{ij}\hat{z}_{ij}$ $\hat{x}_{ij}\hat{y}_{ij}\hat{z}^2_{ij}$

Figure 2-4: The basis set of angular monomials up to 4-body, as shown in [5]

`abf`, a tensor product of `rbf` and `abf` are stored, along with its summation, to be cleverly used later for higher-body updates (Figure 2-7). Next, the 3-body energy is finally updated, but not the force; instead, some set of coefficients are created from the tensor products of `abf` and `rbf` (Figure 2-8) which are then used to update the force (Figure 2-9).

If fast POD is being run as a 3-body potential, it ends here. However, if going up to $3 < n \leq 7$-body, then, we use the results of the 2 and 3 body potentials. Fast

POD creates polynomial descriptors `da` and `dda` for all $1 < a \leq n$ which are used with the tensor products previously stored to update the energy and force for higher-body updates. More specifically, `d2` is created from summations of `rbf` while `d3` is created from some summations of the tensor products. The difference in complexity from finding the two-body descriptors (Figure 2-10) vs the three-body descriptors (Figure 2-11) is large; however, to reduce further growing complexity and work in calculating $n > 3$-body energy and force, finding the remaining polynomial descriptors can be organized into products and computed hierarchically with `dn = di * dj`, where $i + j = n + 1$. Given that we have `d2` and `d3`, `d4 = d2 * d3, d5 = d3 * d3`, and we can now find `d6 = d4 * d3` and `d7 = d4 * d4`. The energy and force are then updated by using relevant descriptors (Figure 2-12) to compute the energy and coefficients needed to compute the force. Though this sounds simple, each of these products is typically four or five loop levels, involving indirect offsets and symmetric matrices or tensors due to the structures of the monomials used to compute `abf`.

2.1.1 Fast POD pseudo code

```
for each neighbor in neighbors[atom]:
    for each besselparam in besselparams:
        for each degree in besseldegrees:
            rbft = expensive_trig and spherical harmonics
            drbft = derivatives of rbft
    for each degree in inversedegree:
        rbft = expensive_trig and spherical harmonics
        drbft = derivates of rbft
```

Figure 2-5: radialbasis

```
e = 0
for each m in rbf:
    for each n in neighbors[atom]:
        e += coeff * rbf(n, m)
        f += coeff * drbf(n, m)
```

Figure 2-6: tallytwobodylocalforce

```
for each m in rbf:
    for each k in abf:
        for each neighbor in neighbors[atom]:
            U = some tensor product of rbf and abf
            dU = derivates of U
            sumU += some summation of the tensor products
```

Figure 2-7: radialangularbasis

```
for each m in rbf:
    for each p in abf:
        for ... :
            for type in element types:
                for type in element types:
                    coeff = some function of sumU
                    e += some function of coeff
                    coeff += some function of coeff
return e
```

Figure 2-8: threebodycoeff

```
for each m in rbf:
    for each k in abf:
        for each neighbor in neighbors[atom]:
            fij += coeff * dU
```

Figure 2-9: tallylocalforce

```
for each m in rbf:
    for each neighbor in neighbors[atom]:
        d2 += rbf
        dd2 += drbf
```

Figure 2-10: two-body descriptors

```
for each m in rbf:
    for each p in abf:
        ...
            for each type in element types:
                for each neighbor in neighbors[atom]:
                    d3 += some function of sumU
                    dd3 += some function of U
```

Figure 2-11: three-body descriptors

```
dn = di * dj     // where i + j = n + 1
e += dotproduct(&coeffs, dn, nl23)
f += some function of coeffs, di, and dj
```

Figure 2-12: $n > 3$-body updates

2.2 Rewriting in Halide

In this section we will introduce the different components of Halide needed to write different loop structures, especially those found in molecular dynamics. We also introduce several methods and strategies to transforming C code to Halide.

2.2.1 General Rewrite

In rewriting any algorithm in Halide, there are several classes in Halide to become familiar with. The four classes we will use the most are `Funcs`[1], `Vars`[2], `RDoms`[3] and `Exprs`[4]. `Funcs` are the units that the pipelines in Halide are scheduled by, representing a collection of regular loop nests to compute one output array. `Vars` are used to help define `Funcs` by representing a domain to iterate over. By default, `Funcs` are expected to be pure functions; however, `RDoms` are used to specify multi-dimensional domains to iterate over. They are used to give `Funcs` update definitions, allow recursion, and scattering. The dimensions to iterate over are defined by `RVars`[5]. Finally, `Exprs` are immutable expressions that can be used to help define a `Func` with intermediate computation. We will also note an essential function of `RDoms`. Halide's `RDom` class has a `where` method which attaches a predicate to a reduction domain in the form of an `Expr`. Given a predicate, the computation in an RDom only occurs when the predicate is true. It is able to handle `Exprs` containing the `RDom`'s own `RVars`, the `Func`'s pure `Vars`, another `Func`, or a recursive call to the same `Func` [6].

We will explain how these classes are used together for a general rewrite of any function.

1. First, we determine the loop structure needed to perform the stores/updates and any possible reads. We create `Vars` for each level of the loop structure. If there are non-rectangular loops, we also define the `RDom` and `RVars` needed.

2. Second, we determine where stores are happening in the algorithm. This can be the intermediate computation that is temporarily stored or the final result. We create `Funcs` for each of the stores. If the store consists of updates, recursion, or

[1] https://halide-lang.org/docs/class_halide_1_1_func.html#details
[2] https://halide-lang.org/docs/class_halide_1_1_var.html
[3] https://halide-lang.org/docs/class_halide_1_1_r_dom.html
[4] https://halide-lang.org/docs/struct_halide_1_1_expr.html
[5] https://halide-lang.org/docs/class_halide_1_1_r_var.html
[6] https://halide-lang.org/docs/class_halide_1_1_r_dom.html

scattering, we must use a `RVars` from an `RDom` for the variables that are reduced, scattered, or reused.

3. Create `Funcs` and/or `Exprs` for any intermediate calculations.

   - If there is an intermediate read from an array, this will have to be a `Func`.

   - Expensive calculations can be manually scheduled if written as a `Func`; otherwise, an `Expr` is sufficient.

4. Rewrite.

While this is a basic algorithm to follow, this is only sufficient for algorithms with very simple loop structures. Also, without proper scheduling, Halide promises to obtain the same result, but may not perform the operations in the same order. For example, any given `Func` may be pre-computed with respect to any of its given `Vars` depending on the supplied schedule. Custom scheduling will be discussed in Chapter 3.

Within the next few sections, we will discuss rewriting specific parts of Fast POD and the different strategies used to successfully do so.

2.3 Per Atom Energy/Force Calculations

In the Fast POD algorithm, there is a point which finds the energy and force in the system per atom per time step. As stated before, this will be one of the most time consuming processes within the MD simulation; therefore, it was the first part of the potential to consider rewriting.

Due to the differing complexities of each function that makes up this part of the potential, we will discuss the functions out of order; rather, we will discuss them in order of complexity to allow for a better understanding of how to transform any potential. We will also provide pseudo code for the c++ implementation to further show Halide's capabilities.

2.3.1 Tallying Two-Body Local Force

Update Definitions and Indirect Read

One of the simpler functions we needed to rewrite was `tallytwobodylocalforce` (Figure 2-6). As the name partially suggests, this function finds the total force and energy for a given atom given each pair of atoms. The function is, roughly, as follows:

```
double e = 0.0;
for (int m=0; m<M; m++)
    for (int n=0; n<N; n++) {
        int nm = n + N*m;
        double c = coeff2[m + nbf*(tj[n]-1)];
        e += c*rbf[nm];
        fij[0 + 3*n] += c*rbfx[nm];
        fij[1 + 3*n] += c*rbfy[nm];
        fij[2 + 3*n] += c*rbfz[nm];
    }
return e;
```

This function has two complexities which makes it different from a general Halide function. Rather than being able to calculate `fij` or `e` all at once, both are repeatedly updated. To accomplish this in Halide, we will use `RDoms` to specify a domain to iterate over. We also will need to use Halide's `clamp`[7] function because of the indirect access `coeff2[m + nbf*(tj[n]-1)]`. Specifically, the "indirect access" refers to indexing into one array, `coeff2`, with an element of another array, `tj`. When using Halide's clamp function, Halide will generate extra assertions to ensure the indirect access is safely within the correct memory bounds (however, if correctness is proven already, `unsafe_promise_clamp` allows for indirect accesses without assertions). Clamps are necessary for Halide to be certain of the bounds of an indirect access. Further, these clamps allow Halide to occasionally pre-compute smaller regions in the case of indirect reads. Despite these complexities, we begin the translation as normal.

First, we identify the loop structure. We will define this loop structure as a 2-level loop over `M` and `N`. This will correspond to `Vars` m and n. Next, we identify the

[7] https://halide-lang.org/docs/namespace_halide.html#a40b1c066344e4816e822a467522bb1f1

important stores that occur: stores to `e` and `fij`. We will define two `Funcs` `fij` and `e`. It is important to notice that m and n are not both necessary for storing neither `fij` nor `e`. `e` is a scalar and does not need any `Vars` for its store. `fij` appears to be a 1D array; however, upon further inspection `fij` is actually a flat 2D array where each row is the vectored force by dimension. Therefore, we can create a new `Var` `dim`. Note, we can apply a similar trick to `rbfx`, `rbfy` and `rbfz`. We will call this combined array `rbfd`. We, then, bound the variables n from [0,`N`) and `dim` from $[0, 3)$. Finally, we create an `Expr` for c. We do not need to create an `Expr` for `nm` because, similarly to `fij`, `coeff2` is a 2D array. As `coeff2` and `tj` are arrays, it is assumed they are both `Funcs`, as well. Fortunately, they are each indexed with `Vars` we have already defined. We will also assume they were already bounded outside of this method's scope.

Given the above structure, we can begin the transformation. First, we define all the `Vars` introduced above. We must also initialize `fij` and `e` to 0 because they are being updated (we must give them an original state!). The updates occur over the entire domain of the nested loops; therefore, we create an `RDom` with two dimensions bounded from $[0, N)$ and $[0, 3)$, similarly to the pure `Vars`. The pure `Vars` will be used in the initialization and the `RDoms` are used in the updates. When accessing `tj`, we apply a `clamp`. Finally, we place explicit bounds on the pure `Vars`.

The Halide translation is shown in Figure 2-13.

2.3.2 Radial Angular Basis

Indirect Access to Write

The function `radialangularbasis` finds the tensor products needed to find higher body energies and forces (Figure 2-7). `radialangularbasis` is very similar to `tallytwobodylocalforce` structurally with some key differences which adds some further complexity; therefore, we will examine this transformation next. First, let's look at the original code in Figure 2-14.

The key differences to note are:

```
Expr zero = Expr((double) 0.0);
Func e("e"), fij("fij");
Var n("n"), m("m"), dim("dim");
e() = zero;
fij(n, dim) = zero;

RDom r(0, N, 0, nbf);
Expr c = coeff2(clamp(tj(r.x), 1, N - 1) - 1, r.y);
e() += c * rbf(r.x, r.y);
fij(r.x, dim) += c * rbfd(r.x, r.y, dim);

fij.bound(n, 0, N);
fij.bound(dim, 0, 3);
```

Figure 2-13: Halide tallytwobodylocalforce

```
for (int m=0; m<Ne*K*M; m++)
    sumU[m] = 0.0;
for (int m=0; m<M; m++) {
    for (int k=0; k<K; k++) {
        for (int n=0; n<N; n++) {
            int ia = n + N*k;
            int ib = n + N*m;
            int ii = ia + N*K*m;
            double c1 = rbf[ib];
            double c2 = abf[ia];
            U[ii] = c1*c2;
            Ux[ii] = abfx[ia]*c1 + c2*rbfx[ib];
            Uy[ii] = abfy[ia]*c1 + c2*rbfy[ib];
            Uz[ii] = abfz[ia]*c1 + c2*rbfz[ib];
            int in = atomtype[n]-1;
            sumU[in + Ne*k + Ne*K*m] += c1*c2;
        }
    }
}
```

Figure 2-14: radialangularbasis code

- The indirect access now occurs within a store instead of a read.
    - `sumU[atomtype[n]-1 + Ne*k + Ne*K*m]`
- The important stores occur between two different arrays instead of a single array and a scalar.

A big similarity that will track throughout all of the following functions is noticing seemingly flat arrays to actually be ND arrays (`ii` is equivalent to indexing into a 2D array at $[m][n]$). We will also find arrays that represent some data vectored out (`Ux`, `Uy`, `Uz`). From this point forward, we will not be explicitly stating this. Now let's begin the translation.

The loop structure will be represented by three `Vars` m, k, and n. In terms of stores, we have `sumU`, `U` and vectored out arrays we will call `Ud`. We will again add a `Var` `dim` for `Ud`. Also notice, `U` and `Ud` have different dimensions from `sumU`. While this may not always be necessary, we will create a new `Var` for `ne`. Between the four `Vars` we have created, this is sufficient for our stores. We now bound the variables m from [0,`M`), k from [0,`K`), n from [0,`N`), ne from [0,`Ne`) and `dim` from $[0, 3)$. Finally, we create `Exprs` for `c1` `c2` and `in`. Each of our `Exprs` are computed (or just are reads) from arrays; therefore, these are each `Funcs` which we will again assume were bounded outside of this method's scope. We also make sure to `clamp` `in` during the `sumU` store. Finally, place explicit bounds on the pure `Vars`.

The Halide translation is as shown in Figure 2-15.

2.3.3 Angular Basis

Recursive Updates

Next, we will translate a seemingly more complex loop structure, but use Halide's built-in tools to make the code much more readable. The function is `angularbasis` (Figure 2-16). One of the two new complexities applied in this function is dealing with conditionals. Halide comes with a `select`[8] struct which works as Halide's built-in ternary operator. This function does not act exactly like a ternary though, as it evaluates both sides of the expression regardless of the conditional's result. The other complexity is a recursively built array. In other words, an array where the element at index i is computed based on some previous index j that was already computed. At a high level, this might seem to be a problem that Halide can not solve due to Halide's internal scheduling; the order of computation is not guaranteed. However,

[8] https://halide-lang.org/docs/struct_halide_1_1_internal_1_1_select.html

```
Expr zero = Expr((double) 0.0);
Var n("n"), k("k"), m("m"), ne("ne"), dim("dim");

sumU(ne, k, m) = zero;
RDom r(0, M, 0, K, 0, N);
Expr c1 = rbf(r.z, r.x);
Expr c2 = abf(r.z, r.y);
Expr in = atomtype(r.z) - 1;

U(m, k, n) = c1 * c2;
Ud(m, k, n, dim) = abfd(k, n, dim) * c1 + c2 * rbfd(m, n, dim);
sumU(clamp(in, 0, Ne - 1), r.y, r.x) += rbf(r.x, r.z) * abf(r.y, r
    .z);

sumU.bound(m, 0, M);
U.bound(m, 0, M);
Ud.bound(m, 0, M);
...
sumU.bound(ne, 0, Ne);
U.bound(n, 0, N);
Ud.bound(n, 0, N);
Ud.bound(dim, 0, 3);
```

Figure 2-15: Halide radialangularbasis

we have already been bypassing this with Halide's RDom. Because RDoms are used for updates, Halide takes the time to apply these in sequential order rather than in any order. Therefore, it is usually best to use pure definitions when possible, as we are able to schedule "more flexibly"[9]. At a high level, recursion can also be viewed as multiple update passes on the same array, updating later indices with values from earlier ones on each update. Therefore, this can be rewritten in Halide using RDoms in the same fashion as we have shown before.

Let us examine the original code on the following page, Figure 2-16.

We can break down the loop structure to be a 2-level loop across N and K and represent them with Vars n and k. The important stores occur in abf and abfd. We add Var dim to handle the vectored arrays. We bound each variable appropriately; k from [1,K), n from [0,N) and dim from $[0, 3)$. This function requires a lot of Exprs; so much so that we have only included the code relevant to the new tactics discussed

[9] https://halide-lang.org/docs/class_halide_1_1_r_dom.html

```
tm[0] = 1.0;
tmu[0] = 0.0; tmv[0] = 0.0; tmw[0] = 0.0;
for (int j=0; j<N; j++) {
    double x = rij[0+3*j];
    double y = rij[1+3*j];
    double z = rij[2+3*j];
    ...
    double u = x/dij;
    double v = y/dij;
    double w = z/dij;
    ...
    double dudx = (yy+zz)/dij3;
    ...
    double dwdz = (xx+yy)/dij3;

    abf[j] = tm[0];
    abfx[j] = 0.0; abfy[j] = 0.0; abfz[j] = 0.0;
    for (int n=1; n<K; n++) {
      int m = pq[n]-1;
      int d = pq[n + K];
      if (d==1) {
        tm[n] = tm[m]*u;
        tmu[n] = tmu[m]*u + tm[m];
        tmv[n] = tmv[m]*u;
        tmw[n] = tmw[m]*u;
      }
      ...
      else if (d==3) {
        tm[n] = tm[m]*w;
        tmu[n] = tmu[m]*w;
        tmv[n] = tmv[m]*w;
        tmw[n] = tmw[m]*w + tm[m];
      }
      abf[j + N*n] = tm[n];
      abfx[j + N*n] = tmu[n]*dudx + tmv[n]*dvdx + tmw[n]*dwdx;
      abfy[j + N*n] = tmu[n]*dudy + tmv[n]*dvdy + tmw[n]*dwdy;
      abfz[j + N*n] = tmu[n]*dudz + tmv[n]*dvdz + tmw[n]*dwdz;
    }
}
```

Figure 2-16: angularbasis

and any reads. We do not need to create any extra Vars for the Exprs or out-of-scope Funcs.

Now, we can examine how to use select to simplify this code. We can rewrite the updates to tm and tmd by noticing what is different in each of the conditions and finding some symmetry. The difference lies in whether tm[m] and tmd[m][dim] are multiplied by u, v or w. Also, which dimension of tmd[m] will be added with tm[m]. We will then add a new Func to represent the choice between u, v and w, called uvw. Because the value of u, v and w rely on Var n, uvw will take n as an argument, as well as some new Var, selected, to represent which condition we are in. Applying this idea, we can rewrite the updates to tm and tmd using select as shown in Figure 2-17.

```
Var selected("selected");
Func uvw("uvw");
uvw(n, selected) = select(selected == 1, u,
    select(selected==2, v,
    select(selected==3, w, Expr((double) 0.0))));
tm(n, r.x) = tm(n, m) * uvw(pair, d);
tmd(n, r.x, r.y) = tmd(n, m, r.y) * uvw(n, r.y) +
    select(r.y + 1 == d, tm(n, m), Expr((double) 0.0));
```

Figure 2-17: select statement for uvw

In a somewhat similar fashion, we can rewrite the pure definition of abfd. We create a Func that is computed based on nested select statements, as seen below in Figure 2-18.

Given these rewrites, the rest of the Halide program will follow as shown in the abbreviated version, Figure 2-19.

2.3.4 Three Body Coefficients

Irregular domains and Symmetry

In many potentials it is often that computation is done over some irregular domains. One time this occurs within fast POD is when updating the 3-body energy and creating the coefficients needed to update the force from the tensor products of the

```
Func jacobian("jacobian"), abf("abf"), abfd("abfd");
Var dim("dim"), dim_p("dim_p");
jacobian(n, dim, dim_p) =
    select(dim == 0,
        select(dim_p == 0,
          dudx,
        select(dim_p == 1,
          dvdx,
        select(dim_p == 2,
          dwdx, zero))),
    select(dim == 1,
        select(dim_p == 0,
          dudy,
        select(dim_p == 1,
          dvdy,
        select(dim_p == 2,
          dwdy, zero))),
    select(dim == 2,
        select(dim_p == 0,
          dudz,
        select(dim_p == 1,
          dvdz,
        select(dim_p == 2,
          dwdz, zero))), zero)));
abf(n, k) = tm(n, k);
abfd(n, k, d) = tmd(n, k, 0) * jacobian(n, d, 0) +
    tmd(n, k, 1) * jacobian(n, d, 1) +
    tmd(n, k, 2) * jacobian(n, d, 2);
```

Figure 2-18: jacobian select statement

```
Expr zero = Expr((double) 0.0);
Var n("n"), k("k"), dim("dim"),
    dim_p("dim_p"), selected("selected");
...
tm(n) = 1;
tmd(n, dim) = 0;
...
RDom rn(1, K + 1, 0, 3);
uvw(n, selected) = ...
tm(n, rn.x, rn.y) = ...
...
jacobian(n, dim, dim_p) = ...
abf(n, k) = ...
abfd(n, k, d) = ...
...
```

Figure 2-19: Halide angularbasis

radial and angular basis functions. *pc*3 is the set of multinomials used to compute the coefficients needed to update the force. Given all of this, it is no surprise the function `threebodycoeff` is as shown in Figure 2-8 and in Figure 2-20.

```
for (int m=0; m<nrbf3*K3*nelements; m++)
    cU[m] = 0.0;
    double e = 0.0;
for (int m=0; m<nrbf3; m++) {
    for (int p=0; p<nabf3; p++) {
      int n1 = pn3[p];
      int n2 = pn3[p+1];
      int nn = n2 - n1;
      for (int q=0; q<nn; q++) {
        int k = 0;
        for (int i1=0; i1<nelements; i1++) {
          double t1 = pc3[n1+q] *
            sumU[i1 + nelements*(n1+q) + nelements*K3*m];
          for (int i2=i1; i2<nelements; i2++) {
            double c2 = sumU[i2 + nelements*(n1+q) + nelements*K3*
               m];
            double c3 = coeff3[p+nabf3*m+nabf3*nrbf3*k];
            double t2 = c3*t1;
            e += t2*c2;
            cU[i2 + nelements*(n1+q) + nelements*K3*m] += t2;
            cU[i1 + nelements*(n1+q) + nelements*K3*m] += pc3[n1+q
               ]*c2*c3;
            k += 1;
          } ...
} return e;
```

Figure 2-20: threebodycoeff code summary

We will discuss implementing `for (int q=0; q<nn; q++)`, `for (int i2=i1; i2<nelements; i2++)`, and `k`.

Let us tackle `for (int q=0; q<nn; q++)` first. Notice that `q` is only used within the expression `n1 + q`. Therefore, this loop is equivalent to a loop from `n1` to `n2`. Given some `RDom`, `r` where `r.x` is bounded to the greatest value in `pn3`, we can use the `where` statement to achieve this loop with `r.where((n1 <= r.x) && (r.x < n2))`. It is important to note that Halide may provide a conditional rather than producing a linear looping structure if the bounds inference cannot determine a reasonable conservative interval. Different scheduling commands can change the bounds inference leading to Halide producing a conditional even if it didn't produce one in the default schedule. This will be discussed more in Chapter 3.

As for the next irregular loop, our RDom will need two different RVars both bounded by nelements. If we imagine these are RVars r.z and r.w, our predicate would be r.w >= r.z. This predicate could be added to the previous or added as a new line.

Finally, instead of continuously updating k in such a linear way (forcing Halide's scheduling to run work linearly, as it is dependent on k), we can notice that k is the flat index of a symmetric array, i.e if we order (x, y) such that $x \leq y$, k is the location of (x, y). This follows that we can rewrite k as an Expr where

$$k = (2 * nelements - 3 - r.z) * (r.z/2) + r.w - 1. \tag{2.2}$$

Given this new definition, although it uses RDoms, each definition of k is independent of past iterations; Halide notices this and gives more flexibility in scheduling than if we updated k constantly.

In conclusion, the rest of the transformation occurs similarly as described in previous sections. The abbreviated transformation is shown in Figure 2-21.

```
...
cU(ne, k3, rbf3) = zero;
RDom r(0, K3, 0, nabf3, 0, nelements,
    0, nelements, 0, nrbf3);
Expr n1 = pn3(r.y);
Expr n2 = pn3(r.y + 1);
r.where((n1 <= r.x) && (r.x < n2) && (r.w >= r.z));
Expr k = (2 * nelements - 3 - r.z) * (r.z/ 2) + r[3] - 1;
...
e() += t2 * c2;
cU(r[3], r.x, r[4]) += t2;
cU(r.z, r.x, r[4]) += pc3(r.x) * c2 * c3;
...
```

Figure 2-21: Halide threebodycoeff

2.4 Outer Loop

Given the tools shown above, we were able to recreate the code that finds the energy and force per atom. At this point in time, we have only recreated the work done per atom in Halide. Therefore, our pipeline in LAMMPS consists of transforming all the inputs into Halide's per atom function within the original outer loop over atoms. After completing this, we worked to manage the outer loop over all the atoms in Halide.

To accomplish this, there were two main changes we made. The original implementation uses a neighbor list that is created per atom. We rewrote this to be a CSR matrix of the neighbor lists of all atoms where each row corresponds to an atom and the columns correspond to neighbors[10]. Then, we added a new `Var` to each necessary `Func` to represent the `atom` needed to access the appropriate offset location in the matrix. Then, we wrote every access to input data to be of the form `pair + offsets(atom)` where `offsets` is an input buffer that stores the CSR offsets. Finally, we added a function to aggregate the outputs, which is outside the current fastpod implementation, but a part of LAMMPs. The original implementation is shown in Figure 2-22.

```
for (int n=0; n<N; n++) {
    int im =  3*ai[n];
    int jm =  3*aj[n];
    int nm = 3*n;
    force[0 + im] += fij[0 + nm];
    force[1 + im] += fij[1 + nm];
    force[2 + im] += fij[2 + nm];
    force[0 + jm] -= fij[0 + nm];
    force[1 + jm] -= fij[1 + nm];
    force[2 + jm] -= fij[2 + nm];
}
```

Figure 2-22: tallyforce

We implement a slightly more complex version of this function because it also gathers all the energies together, allowing these two processes to be scheduled to-

[10] This is common in potentials similar to POD so we reused code from other parts of LAMMPs.

gether, as will be discussed in the next section. Our transformation is as follows in Figure 2-23, where `oatom` and `rout`, is the atom on the outer loop. Finally `y` of the `RDom` corresponds to the 6 updates in the original and an energy update.

```
tallyforce(oatom, dim) = Expr((double) 0.0);
    Expr lhs = tallyforce(
    select(rout.y == 6, rout.z,
        select(rout.y > 2 && rout.y < 6, app,
          .    select(rout.y <= 2, rout.z, 0))),
                 select(rout.y == 6, 3, dimp));
    tallyforce(
    select(rout.y == 6, rout.z,
        select(rout.y > 2 && rout.y < 6, app,
                select(rout.y <= 2, rout.z, 0))),
               select(rout.y == 6, 3, dimp)
          ) = select(rout.y == 6, etemp(rout.z),
                      select(rout.y > 2 && rout.y < 6, lhs + -1 *
                         fij(dimp, rout.x, rout.z),
                            select(rout.y <= 2, lhs+fij(dimp,
                               rout.x, rout.z), Expr((double)
                               -1.0))));
```

Figure 2-23: Halide tallyforce

2.5 Future Work in Implementation

Given the tools shown above, we were able to achieve consistency with fast POD's results on sufficiently complex systems.

The implementation, as it is currently stated above, is not the same algorithm as fast POD, as Halide is performing on its default schedule. It is apparent Halide can succeed in achieving correctness of the fast POD algorithm, but we will discuss the possibilities and limitations to scheduling fast POD in Halide, as well as possible improvements to the current schedule, in the next section.

Chapter 3

Scheduling

Thus far, we have discussed different ways to implement many of the complex computations that are common in MD potentials. However, as previously stated, Halide promises the correct final result, but not a specific order of computation within the generated code. By default, Halide produces each final consumer `Func` independently while in-lining producer `Funcs` and `Exprs`, up to constraints created by reductions, recursion, and scatters. For example, in the subsection Angular Basis, our two `Funcs` `abf` and `abfd` are produced separately with `tm` and `tmd` being in-lined within both, and is thus produced twice! This is costly behavior that results in Halide's default scheduling. There are ways to influence Halide to schedule differently. One example that we have already discussed is `RDoms`, as updates are required to occur in some specified order if changing the order will affect the result (e.g recursive updates). To further influence Halide's schedule, we can write our Halide code differently or explicitly write commands telling Halide what to do. Ideally, we would prefer to solely use the commands, as that is the selling point of Halide: to separate the algorithm from the schedule. First, we will discuss some of the most common commands we used to mimic the current Fast POD implementation, as well as limitations we found. Then, we discuss different steps and ideas we used to further optimize the Halide implementation.

3.1 Scheduling Commands and Tricks

3.1.1 Reorder

One of the first commands to become familiar with is Func.reorder[1]. This function allows us to specify the loop order of the generated code for a given Func by listing the Vars and/or RVars involved from innermost out. For pure definitions, any order of relevant Vars can be used. For update definitions, any order of relevant Vars and/or RVars can be used, as long as it does not change the outcome of the result. It is possible that reordering RDoms where a predicate is used (where command is used) can cause if statements to form where they are not needed, due to an overestimate in bounds inference, leading to a surprising performance problem.

In subsection Angular Basis, we originally had a 2-level loop over k and n, innermost out. However, we also added a new dimension to loop over, dim. If trying to mimic the original algorithm, then the loop structure must be dim, k and n, innermost out. We specify abf.reorder(dim, k, n). To reorder the loop structure of an update, we use reorder on the Func's update stage with Func.update(u).reorder (x, y, z).

Similarly, we can use Halide's Func.reorder_storage[2] function to specify the nesting order that the data is laid out in storage without changing how we refer to the Func. We perform this on every Func to mimic the original implementation as closely as possible. We were not able to perform this transformation on d2 and dd2 in the code shown in Figure 2-10. When the transformation was applied on either of these Funcs, it drastically changed the force and energy outputs. While Halide guarantees that the reorder_storage function should not affect computation, there may be a bug or some structural reason why these two Funcs could not have their storage reordered.

[1] https://halide-lang.org/docs/class_halide_1_1_func.html#a77a2bc9d4bb4dfab342f3f412f8e2927
[2] https://halidelang.org/docs/class_halide_1_1_func.html#ae91507761237f8f477fb190bcba1c779

3.1.2 Specialize

While this was not included in our pseudo code, many of the functions in the original implementation have specialized optimizations for specific scenarios. For example, in `tallyLocalForce`, there is specialized code that reduces computation if there is only one element in the simulation. Fortunately, to mimic this behavior in Halide, it is not necessary to write a specialized block of code. Instead, we can use the command `specialize`, which takes in a conditional in the form of an `Expr` to specialize on.

3.1.3 Computing in Same Loop

In many of the functions described above there are multiple arrays being written to within the same loop structure. As stated before, Halide does not do this by default. However, an intuitive trick to ensure Halide produces two `Funcs` together is to combine them into a single `Func`. We have used this trick in defining the new `Var` `dim` in many of the examples above. Instead of having x vector array computed separately from the y vector array, we combined the vectored arrays into a single multi-dimensional array so Halide would compute them together. We can take this a step further by combining like-`Funcs`, such as `U` with `Ud` and `abf` with `abfd`, all of which referenced in examples above. As neither of these pair of `Funcs` are ever read from without the other, it may naturally make sense to combine them. However, this trick only works well if the structures of the outputs match: if one has three dimensions, but the other two then merging them will be awkward, even more so if the two matching dimensions don't match.

For other `Funcs` that we wish to compute together we use Halide's `compute_with` function[3]. This allows us to specify a loop level of a `Func` or update to compute at the same time as another `Func` or update. However, the function is very limited compared to the more robust scheduling commands that we have used so far. There are several examples throughout where Halide was not able to compute two `Funcs` in the same loop even though they were computed together in the original code.

[3] https://halide-lang.org/docs/class_halide_1_1_func.html#a8f1204939742d77c847c0928e865d318

For example, in radialangularbasis, U and sumU are updated within the same loops. To schedule Halide to produce both sumU and U together, we attempted to add the scheduling command sumU.update(0).compute_with(U, n). The second argument in compute_with tells Halide the inner-most loop to compute both Funcs over. Unfortunately, Halide's restrictions using associativity analysis are too strict and prevent this from occurring, insisting there are cyclic dependencies between the two. Halide's compute_with method also can not be mixed with specializations. It is also limited to the two Funcs sharing one or more Vars even if they are using the same loop structure (e.g. both must use the same RDom so the command doesn't work on two overlapping RDoms). Therefore, we are not able to replicate computing the scalar e and the array fij within the same loop structure as is done in tallytwobodylocalforce and tallylocalforce. A known problem with compute_with is its inability to realize updates can occur at the same time, rather than atomically. For example, in threebodycoeff, two different locations in cU are written to within the same loop structure; however, Halide can not cleanly do this at the moment (although a "hack" is mentioned here[4] but was not used in our implementation).

The compute_with method is comparatively new in respects to other methods we use in Halide and the tool is still prominently used and being improved; many of the issues discussed have current GitHub issues open with responses for "hacks" from Halide's creators to achieve what compute_with can not while they work on the method.

3.1.4 Compute as Needed

One of the more complicated functions to mimic schedule-wise is angularbasis. In the original implementation, the arrays tm, tmu, tmv, and tmw are occupied as the values are needed, rather than all at once beforehand. Assuming we have combined Funcs tm with tmd and abf with abfd similarly to described above in subsection Combining Funcs, we will already produce tm and tmd within the same

[4] https://github.com/halide/Halide/issues/3943

loops; therefore, `tm` is now produced once instead of twice. To further mimic the original implementation, we will use `Func.compute_at` [5] to ensure `tm` is only computed as needed by `abf` within the innermost loop. With the scheduling command `tm.compute_at(abf, k).store_in(MemoryType::Stack)`, we assert that Halide will produce code similar to the code shown below while storing `tm` in the stack.

```
for (int n=0; n<N; n++) {
    for (int k=1; k<K; k++) {
        double tm[x];
        tm[k] = ...;
        ...
        abf[...] = ...;
        ...
}
```

It is important to note that `compute_at` may not utilize predicates (`where` statements) or clamps (`clamp` statements), resulting in an unexpected performance decline due to extra computation occurring.

Similarly, if we wanted to produce all of `tm` before `abf`, we could also call `tm.compute_root()`.

3.2 Mimicking Fast POD's schedule

We use many of the scheduling tricks listed in the previous section to ensure the generated code is similar to the original implementation. As can be seen from the code snippets in Chapter 2, loop orders vary from outermost being the neighbors of atom *a* to being the radial basis functions; therefore, we use Halide's `reorder` function to match the loop structure of the computations. Next, we examine how stores are handled for two reasons. First, we reorder the storage to match the original implementation. Then, we realize throughout virtually all of fast POD's per atom calculations, stores are fully computed and stored to be used for later (with one of the few exceptions being `tm`, Figure 2-16, as described in the previous section). To follow this scheduling, we use Halide's `compute_root` function in most places.

[5] https://halide-lang.org/docs/class_halide_1_1_func.html#af4aca8ca6331e64a6fdfd98fccf1757a

Our original schedule mimicking the original implementation as much as possible with Halide is as shown in Figure 3-1.

We also note that originally Halide was re-allocating space every time Halide is entered; resolving in redundant work. LAMMPS keeps a scratch space of pre-allocated memory, which we can tell Halide to use to avoid constant reallocation.

3.3 Outer Loop

Once we mimicked the original schedule per atom, we worked to recreate the outer loop over all the atoms. To mimic the original schedule with the outer loop implemented, we added the outer atom to the outermost of the loop orders and changed many of the `compute_root` commands to `compute_at` to be computed at the outer loop atom instead (Figure 3-2). Then, unlike the original implementation, we parallelized the atom loop, using splits and atomics, though this required an override of Halide's associativity test.

```
rbft.store_root().compute_root();
rbf.store_root().compute_root();
fij.bound(n, 0, N).bound(dim, 0, 3);
fij.reorder_storage(dim, n).update(0)
    .reorder(dim, r.x, r.y).unroll(dim).compute_root();
e.compute_root();
tm.reorder_storage(c, m, n).store_root().compute_at(abf, m)
    .update(1).reorder(n, r.x, r.y).unroll(r.y);
abf.reorder_storage(c, n, m).store_root().compute_root();
U.reorder_storage(c, n, k, m).reorder(c, n, k, m).store_root()
    .compute_root().unroll(c, 4).specialize(Ne == 1);
sumU.reorder_storage(n, k, m).store_root().compute_root()
    .update(0).reorder(r.z, r.y, r.x).specialize(Ne == 1);
d2.compute_root().update(0).reorder(r.x, r.y);
dd2.compute_root().update(0).reorder(dim, r.x, r.y)
    .unroll(dim).compute_with(d2.update(0), r.y);
    // Note: we could not compute over r.x because of a
       compute_with limitation
d3.reorder_storage(k, m, km).compute_root().update(0)
    .reorder(r.w, r.z, r.y, m).specialize(Ne == 1);
dd3.reorder_storage(dim, n, k, m, km).compute_root()
    .update(0).reorder(dim, r.w, r.z, r.y, r.x, m)
    .unroll(dim).specialize(Ne == 1);
cU.reorder_storage(ne, k, m).update(0)
    .reorder(r.w, r.z, r.y, r[4]).update(1)
    .reorder(r.w, r.z, r.y, r[4]);
e.update().reorder(r.w, r.z, r.y, r[4]).compute_root();
fij.update(1).reorder(dim, r.z, r.y, r.z).unroll(dim)
    .specialize(Ne == 1);
d23.reorder_storage(d23i, d23j).reorder(d23i, d23j)
    .compute_root();
e23.compute_root();
cf1.compute_root().update().reorder(r1.x, j);
fij.compute_root().update(2).reorder(dim, n, r2.x);
cf2.compute_root().update().reorder(r2.x, i);
fij.compute_root().update(3).reorder(dim, n, r1.x);
d33.compute_root().update(0).reorder(r.x, r.y);
e33.compute_root();
cf13.compute_root().update(0).reorder(r.x, r.y);
fij.compute_root().update(4).reorder(dim, n, r1.x);
cf23.compute_root().update(0).reorder(r.x, r.y);
fij.compute_root().update(5).reorder(dim, n, r2.x);
fij.update(6).reorder(dim, r.z, r.y, r.x).unroll(dim)
    .specialize(Ne == 1);
```

Figure 3-1: mimicked schedule

```
...
dd2.reorder(dim, ne, m, n, oatom)
...
tm.compute_at(tallyforce, rout.z);
fij.compute_at(tallyforce, rout.z);
cu4.compute_at(tallyforce, rout.z);
cU.compute_at(tallyforce, rout.z);
sumU.compute_at(tallyforce, rout.z);
d33.compute_at(tallyforce, rout.z);
d23.compute_at(tallyforce, rout.z);
d3.compute_at(fij, oatom);
d3.compute_at(tallyforce, rout.z);
dd3.compute_at(tallyforce, rout.z);
dd2.compute_at(tallyforce, rout.z);
d2.compute_at(tallyforce, rout.z);
...
```

Figure 3-2: Outer loop scheduling changes

Chapter 4

Results

The Halide version of fast POD, as it stands, is currently 20% slower than the original when run serially and over 300% faster when run in parallel over the atoms in the outer loop. The simulation was conducted on an Intel Macbook Pro 2020 with a 2.4 GHz 8-Core Intel Core i9 processor and 32 Gigabytes of RAM. The simulation was conducted on 16000 atoms for 20 time steps. There were an average of 50 neighbors per atom except at the first time step where there was an average of a thousand neighbors per atom. There were also 3 bessel degrees, 6 inverse degrees, 6 two-body radial basis functions, 5 three-body radial basis functions and 5 three-body angular basis functions. In this section, we will show the scheduling commands we used to approach closer to parity in serial, the command needed to run in parallel, examples of Halide's ease of implementation in the molecular dynamics domain, and possible future work.

4.1 Changing the Schedule

The mimicked schedule (Figure 3-1) gave a two times slow down. Profiling revealed that producing `tm` in `angularbasis`, Figure 2-16, was the most expensive computation. We experimented with different schedules and found that the most effective option was to compute all of `tm` at once. The next obvious target was `U`; we found Halide's default schedule, inlining the computation, proved better than computing and storing it, as is done in `radialangularbasis` (Figure 2-7). Combined, these two optimizations allowed for a 100% speed up in performance. These two optimizations required changing two lines of our schedule, as shown in Figure 4-1.

```
tm.reorder_storage(c, m, n).store_root().compute_root()
    .update(1).reorder(n, r.x, r.y).unroll(r.y);
U.reorder_storage(c, n, k, m).reorder(c, n, k, m)
    .store_root().unroll(c, 4).specialize(Ne == 1);
```

Figure 4-1: optimized schedule

As we approached closer to parity, we noticed that we were spending a lot of time allocating memory. Given that we did not have the outer loop implemented at the time, memory was continuously reallocated every time the program entered Halide. When one allocation occurred and the memory was reused per atom, we sped up the computation by 11%, bringing the Halide version to only 20% slower than the original (Figure 4-2). This turns out to be an element of the original schedule in LAMMPs that we missed as it was outside of the code we examined. Although, this can't be accomplished with Halide schedules if we do not represent the outer loop, it can be accomplished via pre-allocating memory for the Halide run time.

After reaching this run time, we implemented the outer loop with the motivation of computing the force and energies of atoms in parallel. While this would be a challenging optimization in the original implementation, due to the tallying of forces and energy between all atoms (Figure 2-22) among other factors, this optimization is trivial once the outer loop is implemented. With the scheduling changes used in Figure 3-2 and the extra command `tallyforce.update(0).atomic(true).parallel(rout.z)`, Halide successfully parallelizes over the outer loop atoms, applying necessary atomic operations where needed. All the combined scheduling changes results in a 350% speedup! It is important to note that we did not replicate the allocation optimization in the parallel implementation, as we would need to implement our own parallel allocator. Despite this, we still manage an impressive speedup.

There are several parts of the implementation that we could not replicate, as discussed in Chapter 3. It is possible these differences are the reason for the slow down in the serial implementations. We have also not used any vectorization beyond the LLVM defaults. Finally, we have not optimized the matrix multiplies in this code where as the original manipulates the data to call BLAS.

Original	Halide Mimick	Halide Optimized	Halide Preallocate	Halide Parallel
.09 ns/day	.03 ns/day	.063 ns/day	.075 ns/day	.34 ns/day

Figure 4-2: Simulation computes x ns/day of Halide solution vs original fast POD. The first step of the simulation featured a much faster original, .114 ns/day whereas the Halide implementations performed consistently across all time steps.

While we have not reached parity serially, we have been able to reach 20% within the goal with minimal scheduling changes and without parallelism– both of which we are able to accomplish and experiment with by changing single commands.

4.2 Simplicity of implementation and code

Using the translation and schedule we have shown above, the Halide code consists of roughly 335 lines of code while the corresponding C code is roughly 526 lines (see table 4-3). To be fair in assessment, we removed white space but included extra lines used for readability. Further, we only compare with the methods of fast POD that were directly translated, the kernels, rather than the wiring and infrastructure around them. On average, enforcing the original schedule consisted of about two to three extra lines per method.

Halide accomplishes this because of its elegance in handling reductions and multiple loop levels. In `angularbasis` a recursive array is built as shown in figure 4-4. In Halide, we accomplished this in 3 lines, figure 4-5. Even an author of the original code remarked at how "clean" and "succint" the recursive implementation was.

In `threebodydesc` there is a loop structure which requires accesses based on an offset, as shown in figure 4-6. We accomplish this by implementing the generalized algorithm using `where` predicates and the `specialize` command, as shown in figure 4-7, where `r.x` is equivalent to `q` in the original.

Halide especially shines in the example of the matrix multiplication that occurs

	Original	Halide	Halide w/Scheduling
rbft	~130	~75	+3
matrix multiply	~30	~3	+2
tally two body local force	~10	~7	+3
angularbasis	~70	~65	+4
radial angular basis	~40	~15	+3
two body desc deriv	~10	~7	+3
three body desc	~30	~20	+3
three body desc deriv	~40	~12	+3
three body coeff	~20	~17	+3
tally local force	~25	~4	+1
four body desc	~3	~2	+1
four body coeff	~45	~32	+3
dot product	~4	~3	+1
four body fij	~20	~15	+5
window accessing	~15	~2	+0
five body desc	~5	~5	+1
five body fij	~25	~15	+3
total:	~526	~292	+43
avg:	~31	~17	+2

Figure 4-3: Line numbers of original fast POD vs Halide fast POD

to obtain `rbf`, as shown in figure 2-1. The original implementation performs four separate matrix multiplications. Instead, in Halide we are able to perform this, and any matrix multiplication, in as little as 4-6 lines, regardless of dimension and schedule it to perform the multiplication in any way we may wish (Figure 4-8).

Overall, potentials in molecular dynamics consists of expensive computations consisting of linear algebra and trigonometry over complex loop structures. Halide is able to represent these computations as either a `Func` or `Expr`, giving us freedom to compute these in-lined, parallel, or any specified schedule. The size of our program also does not grow linearly with number of loop levels or increasing complexity of loop structures as we may only need to create a new `Var`, `RVar`, or predicate with `where`. Finally, as important it is for potentials to be computed efficiently, Halide is a great tool for quickly and safely experimenting with different scheduling, also without increasing complexity of the program.

```
 for (int n=1; n<K; n++) {
  if (d==1) {
    tm[n] = tm[m]*u;
    tmu[n] = tmu[m]*u + tm[m];
    tmv[n] = tmv[m]*u;
    tmw[n] = tmw[m]*u;
  }
  else if (d==2) {
    tm[n] = tm[m]*v;
    tmu[n] = tmu[m]*v;
    tmv[n] = tmv[m]*v + tm[m];
    tmw[n] = tmw[m]*v;
  }
  else if (d==3) {
    tm[n] = tm[m]*w;
    tmu[n] = tmu[m]*w;
    tmv[n] = tmv[m]*w;
    tmw[n] = tmw[m]*w + tm[m];
  }
}
```

Figure 4-4: Recursive array

```
Func uvw("uvw");
uvw(pair, selected) = select(selected == 1, u,
    select(selected==2, v,
    select(selected==3, w, Expr((double) 0.0))));
tm(pair, rn.x, rn.y) = tm(pair, m, rn.y) * uvw(pair, d) + select(d
    == rn.y, tm(pair, m, 0), zero);
```

Figure 4-5: Halide recursive array

4.3 Future Work

Improving HalideFastPod: The next natural step in this work is improving performance. Also, fast POD is a $n \leq 7$-body potential, we only implemented up to 5-body in Halide; however, we can use the same tools and strategies from this thesis to implement the entire fast POD algorithm. There are also different scheduling strategies to be experimented with to reach parity and further increase performance. It may be advantageous to parallelize over a set of atoms. There may also be computation that is repeated often enough that is worth storing between iterations. For

```
if (Ne == 1) { ...similar code to below... } else {
for (int m=0; m<M; m++)
  for (int p=0; p<K; p++) {
    int n1 = pn3[p];
    int n2 = pn3[p+1];
    int nn = n2 - n1;
    for (int q=0; q<nn; q++) {
        k = 0;
        for (int i1=0; i1<Ne; i1++) {
            double t1 = pc3[n1+q]*
                sumU[i1 + Ne*(n1+q) + Ne*K*m];
            for (int i2=i1; i2<Ne; i2++) {
                d3[p + K*m + K*M*k] ... ; k += 1;
} ... }
```

Figure 4-6: Three body descriptors

```
RDom r(0, k, 0, K, 0, Ne, 0, Ne);
Expr n1 = pn3(r.y);
Expr n2 = pn3(r.y + 1);
r.where(n1 <= r.x && r.x < n2 && r.z <= r.w);
t1 = ...; k = ...; t2 = ...; d3(...) = ...;
d3.update()...specialize(Ne == 1);
```

Figure 4-7: Halide three body descriptors

```
Func prod("prod"), rbf("rbf");
prod(c, k, i, j) = Phi(k, i) * rbft(j, k, c);
rbf(j, i, c) = Expr((double) 0.0);
RDom r(0, ns);
rbf(j, i, c) += prod(c, r, i, j);
rbf.{schedule_here}
```

Figure 4-8: Halide Matrix Multiplication

example, U from radialangularbasis (Figure 2-7), is only read from 3 times in the potential; therefore, it is currently advantageous to in-line its computation. However, if up to 7-body is implemented, we would read from U 6 times, possibly making it better to compute_root as we had in the mimicked schedule (Figure 3-1). These are only a few of the possible next steps to improving fast POD, without even considering vectorization[1] and tiling[2]; both of which Halide has methods to easily apply to Funcs when scheduling.

Improving Halide's ability to implement MD potentials: While Halide can implement molecular dynamics codes, there are still limitations as shown in Chapter 3. First, work can be done to improve Halide to be more capable of managing arrays of irregular sizes. Much work can also be done to make `compute_with` more robust, specifically allowing for Funcs sharing loop levels to be computed together, regardless of a specialization or less than exactly shared dimensions. Similarly, Halide could add support for functions that store mixed sized sub-arrays (e.g. an array of 3 by 3 matricies paired with by vectors of size 8; this can be done, but not without the programmer managing it themselves). Second, improvements can be made to `compute_at` to always utilize predicates and `clamps` as the current bounds inference algorithm is not robust enough to using predicates and scheduling commands. Lastly, we can also investigate why reordering the storage of `d2` or `dd2` changed the result of the program. More generally, Halide's ability to implement MD programs and various other domains would improve if it could better manage programs that deal with mostly regular but still technically irregular arrays, such as those with symmetry or irregular sizes defined by some other array or simply a few different sizes. These almost work currently, but can break when a few extra complications are added.

Other DSLs and Molecular Dynamics: Extended future work may consist of rewriting many potentials in Halide, as well as initially writing potentials in Halide. Halide has proven to be a competent tool in accomplishing many of the tasks required in molecular dynamics; however, it is still not perfect and requires additional support to accomplish everything one would expect. It may be possible that another domain specific language, such as Tiramisu [1] and Taichi [4] would be better suited to the task.

[1] https://halide-lang.org/docs/class_halide_1_1_func.html#a13ad9bed80565d85f2cc6d09c607fdfb

[2] https://halide-lang.org/docs/class_halide_1_1_func.html#a5413c606618e7c4257a1129666922fb5

Chapter 5

Conclusion

Although Halide is a tool created to improve image processing pipelines, we believed image processing shared enough of a resemblance to molecular dynamics that Halide might prove capable in both domains. Although we focus on fast POD throughout the thesis, fast POD is only an example of how complex potentials in LAMMPS can be. We showcase Halide's simplistic style in representing complex loop structures and its ability to allow the user to choose and change a program's scheduling. This proves to be useful when optimizing algorithms, as changing a loop order of a computation, the storage of an array, or even how much of an array is computed at a time each can be modified with a single command.

We mimicked the fast POD potential in Halide, achieving the same results but not scheduling; therefore, Halide was not a perfect tool for mimicking the original implementation. With the closest mimick easily possible with Halide's built-in methods, we were able to achieve a performance within two times parity. However, with only three iterations of profiling and another three lines of scheduling changes, we are able to reach within 20% of parity. With a few commands, we are able to compute in parallel over the outer loop atoms, boosting performance to a 350% speedup.

We believe that with more improvements to the schedule, the Halide fast POD can out perform the original implementation serially and we can continue to improve the parallel implementation. Furthermore, we believe that Halide is a great tool that, if improved to better manage common patterns found in MD, can be used to improve many other MD potentials.

Bibliography

[1] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. 2018.

[2] B.O., C.B., and A.B. Supercomputer molecular modeling of thermodynamic equilibrium in gas-metal microsystems. *Numerical Methods and Programming*, (1):123–138, 2015.

[3] Qiangqiang Gu, Linfeng Zhang, and Ji Feng. Neural network representation of electronic structure from ab initio molecular dynamics. *Science Bulletin*, 67(1):29–37, Jan 2022.

[4] Yuanming Hu. The taichi programming language. *ACM SIGGRAPH 2020 Courses*, 2020.

[5] Ngoc-Cuong Nguyen. Fast proper orthogonal descriptors for many-body interatomic potentials. *Physical Review B*, 107(14), 2023.

[6] Shaoliang Peng, Xiaoyu Zhang, Wenhe Su, Dong Dong, Yutong Lu, Xiangke Liao, Kai Lu, Canqun Yang, Jie Liu, Weiliang Zhu, and et al. High-scalable collaborated parallel framework for large-scale molecular dynamic simulation on tianhe-2 supercomputer. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 17(3):804–816, 2020.

[7] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide. *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[8] James C. Sweet, Ronald J. Nowling, Trevor Cickovski, Christopher R. Sweet, Vijay S. Pande, and Jesús A. Izaguirre. Long timestep molecular dynamics on the graphical processing unit. *Journal of Chemical Theory and Computation*, 9(8):3267–3281, 2013.

[9] Aidan P. Thompson, H. Metin Aktulga, Richard Berger, Dan S. Bolintineanu, W. Michael Brown, Paul S. Crozier, Pieter J. in 't Veld, Axel Kohlmeyer, Stan G. Moore, Trung Dac Nguyen, and et al. Lammps - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Computer Physics Communications*, 271:108171, Feb 2022.