LSH-SMILE: Locality Sensitive Hashing Accelerated Simulation and Learning

Anonymous Author(s)
Affiliation
Address
email

Abstract

The advancement of deep neural networks over the last decade has enabled progress in scientific knowledge discovery in the form of learning Partial Differential Equations (PDEs) directly from experiment data. Nevertheless, forward simulation and backward learning of large-scale dynamic systems requires handling billions mutually interacting elements, the scale of which overwhelms current computing architectures. We propose Locality Sensitive Hashing Accelerated Simulation and Learning (LSH-SMILE), a unified framework to scale up both forward simulation and backward learning of physics systems. LSH-SMILE takes advantages of (i) the locality of PDE updates, (ii) similar temporal dynamics shared by multiple elements. LSH-SMILE hashes elements with similar dynamics into a single hash bucket and handles their updates at once. This allows LSH-SMILE to scale with respect to the number of non-empty hash buckets, a drastic improvement over conventional approaches. Theoretically, we prove novel bound on the errors introduced by LSH-SMILE. Experimentally, we demonstrate that LSH-SMILE simulates physics systems at comparable quality with exact approaches, but with way less time and space complexity. Such savings also translate to better learning performance due to LSH-SMILE's ability to propagate gradients over a long duration.

1 Introduction

Learning-driven scientific discovery has enjoyed rapid progress thanks to the advancement of deep neural networks over the last decade. Since Partial Differential Equations (PDEs) are widely used to model physics systems, a fruitful line of research has been developed focusing on learning PDEs from experimental data, including Finzi et al. [2020], Greydanus et al. [2019], Takashi Matsubara and Yaguchi [2020].

Nevertheless, successful applications of machine learning for scientific discovery still face multiple challenges, many of which are computational. Both the forward simulation and backward learning of large-scale dynamic systems requires handling billions mutually interacting elements, the scale of which overwhelms current computing architectures. Forward simulation involves simulating the trajectory of a PDE from a given starting state. Recent work of Greydanus et al. [2019] demonstrates such process can be represented using a neural network similar to ResNet. Backward learning is to discover (the parameters of) the physics model automatically from experimental data, which also attracts recent attention in Niu et al. [2020] and Xue et al. [2021]. Backward learning can be achieved by embedding a neural network modeling the forward simulation into the overall architecture and minimizing a loss function which penalizes the difference between the simulated result and the observed data via back-propagation. In both forward simulation and backward learning, billions of mutually interacting elements resulted from applying the finite difference or finite element approaches

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

have to be handled efficiently. Current brute-force approaches which treat each element as one matrix element do not scale to meet the computational need.

To tackle the computational bottleneck, we propose **L**ocality **S**ensitive **H**ashing Accelerated **Si**m**u**lat**i**on and **Le**arning (LSH-SMILE), a unified framework to scale up both forward simulation and backward learning of physics systems represented in a set of PDEs. The method is based on three key observations. The first is that only a small fraction of elements change while the majority of elements remain the same in one step forward simulation. The second is that the one-step update of one element depends on the values of a small set of neighboring elements. The third is that many elements share similar temporal dynamics. Motivated by these observations, LSH-SMILE harnesses Locality Sensitive Hashing to boost computational throughput. We hash elements into hash buckets via LSH, where elements with similar neighboring element values are grouped into the same bucket. Then the forward simulation and backward learning are carried out for all the elements hashed into the same bucket at once. In this way, the time and space complexities of LSH-SMILE are reduced to be propotional to the number of non-empty hash buckets, a drastic improvement compared to the number of distinct elements in the brute-force algorithm. We also prove a novel bound on the quality of the approximation of LSH-SMILE and provide an interface to control the error rate.

Experimentally, LSH-SMILE is able to simulate and learn physics systems represented in PDEs at a comparable quality with the exact methods while saving drastically in computation. We focus our attention on systems in nano-physics, where we simulate and learn the grain growth in materials (Fan and Chen [1997]) and model the spatial temporal dynamics of void shaped defects, namely nanovoids, in materials under high temperature and irradiation (Millett et al. [2011]). The proposed LSH-SMILE reduces the overall simulation time compared to an exact Torch implementation by 70% in the grain growth simulation, and by 95% in the nanovoid simulation. In terms of precision, LSH-SMILE matches the ground truth results to the scale of 10^{-5} in both the nanovoid and the grain growth simulations after 100 simulation steps. Such savings directly translate to better learning performance. In our experiment which uses LSH-SMILE to learn the physics parameters governing a grain growth process, LSH-SMILE-based learning approach can successfully identify the correct model parameters while brute-force baseline approaches cannot, The success is due to a longer forward simulation (30 time steps) embedded in the backward learning process enabled by the computational savings of LSH-SMILE. Baseline approaches can only embed 10 forward simulation steps with similar computational overhead. Overall, the computational innovation of LSH-SMILE opens the door for faster and better scientific discoveries in the future.

2 Preliminaries

Energy-based systems represented in Partial Differential Equations (PDEs). The dynamics of many physics systems can be described with first or second-order Partial Differential Equations (PDEs) modeling the system energy. Given the state $\vec{u}(\vec{p},t) = (u_1(\vec{p},t),\ldots,u_m(\vec{p},t))^T$, where $\vec{p} = (x_1,\ldots,x_d)^T$ denotes the spatial coordinates and t denotes time, such PDEs are in the form:

$$\frac{\partial \vec{u}(\vec{p},t)}{\partial t} = D(\vec{u}) + G(\vec{u})\nabla F(\vec{u}) + I(\vec{u})\nabla^2 H(\vec{u}). \tag{1}$$

Here, F and H are functions that map $\mathcal{R}^m$ to $\mathcal{R}$. $\nabla = (\frac{\partial}{\partial x_1},\ldots,\frac{\partial}{\partial x_d})^T$ represents the gradient operator, $\nabla^2 = \nabla\cdot\nabla$ is the Laplace operator. D is a function that maps $\mathcal{R}^m$ to $\mathcal{R}^m$. $G(\vec{u})$ is an m-by-d matrix and $I(\vec{u})$ is a m-by-1 vector.

Many notable physics systems can be represented using PDEs in Equation 1. The widely used Allen-Cahn and Cahn Hilliard Equations in phase field modeling are good examples. The Cahn–Hilliard equation and Allen-Cahn equation are used to describes the physical changing process of field variables u in nano-scale phase-field physics model. The field variables are representation of micro-structures in such a model. The Cahn–Hilliard equation has a general form:

$$\frac{\partial u}{\partial t} = \nabla\cdot\left(M\nabla\frac{1}{N}\frac{\delta F}{\delta u}\right). \tag{2}$$

And the Allen-Cahn equation has a general form:

$$\frac{\partial v}{\partial t} = -L\,\frac{\delta F}{\delta v}. \tag{3}$$

Here u and v are the field variables that we concern, and it is assumed to be continuous and changing rapidly across the inter-facial regions. t is time. M, N, L are physics parameters that are related to the changing process. And F is the energy function which has different form case by case. More detail is shown in appendix.

Discretization with finite difference. Finite difference methods allow us to simulate and learn PDEs using the forward and backward pass of a neural network. It has attracted recent attentions in work Wang et al. [2020], Dong Ming [2017] and Shojaei et al. [2019]. More specifically, let $\{x_{i,1}, x_{i,2}, \ldots, x_{i,K}\}$ be a finite discretization of the domain of x_i, where each $x_{i,k}$ is called a discretization point. $\{t_1, t_2, \ldots, t_L\}$ is the discretization of the time. Let $\vec{i} = (i_1, i_2, \ldots, i_d)$. We use $\vec{u}(\vec{i}, j)$ as a shorthand for $\vec{u}$ evaluated at position $\vec{p} = (x_{1,i_1}, \ldots, x_{d,i_d})^T$ and time t_j, i.e., $\vec{u}(\vec{i}, j) = \vec{u}(x_{1,i_1}, \ldots, x_{d,i_d}, t_j)$. Each discretized position $\vec{i}$ is also called an *element*. The left-hand side of Equation 1 can be discretized using the finite difference: $\frac{\partial \vec{u}(\vec{i},j)}{\partial t} \approx \frac{\vec{u}(\vec{i},j+1)-\vec{u}(\vec{i},j)}{t_{j+1}-t_j}$. Similarly, notice $\nabla F(\vec{u}) = (\frac{\partial F(\vec{u})}{\partial x_1}, \ldots, \frac{\partial F(\vec{u})}{\partial x_d})^T$, where $\frac{\partial F(\vec{u})}{\partial x_l} = \sum_{k=1}^{m} \frac{\partial F(\vec{u})}{\partial u_k} \frac{\partial u_k}{\partial x_l}$. We need $\frac{\partial u_k(\vec{i},j)}{\partial x_l}$ to compute ∇F and $\frac{\partial u_k(\vec{i},j)}{\partial x_l}$ can be approximated by $\frac{u_k(\vec{i}+\mathbf{1}_l,j)-u_k(\vec{i},j)}{x_{l,i_l+1}-x_{l,i_l}}$. Here, $\vec{i} + \mathbf{1}_l$ means to move the l-th coordinate of $\vec{i}$, namely i_l, to the next discretization point i_{l+1}. After repeating this type of calculations for the second-order derivatives $\nabla^2 H(\vec{u})$, we can approximate the PDE in Equation 1 as:

$$\vec{u}(\vec{i}, j+1) = \vec{u}(\vec{i}, j) + \delta_t Q(\{\vec{u}(\vec{i}', j), \vec{i}' \in N(\vec{i})\}). \tag{4}$$

We use function Q to abstract out the functional dependence, since the actual form is not important for the purpose of this paper. Q depends on all $\vec{u}(\vec{i}', j)$, in which $\vec{i}'$ is a neighboring element of $\vec{i}$. Two elements neighbor each other if and only if they differ in one dimension with at most two discretization points, or they differ in two dimensions, each dimension with a difference of one discretization point. We use $N(\vec{i})$ to represent the set of neighboring elements of $\vec{i}$.

Discrete simulation and learning. By discretizing PDEs with finite difference approach(Equation 4), we could express the physics simulation process by encoding physics variables into vectors and thus perform calculation on them. We here give a definition of the discrete simulation and learning task. We are given a series of observations of a physics system, and we know the rules that dominate such a system. But we don't know the specific parameters in these rules. We want to express the physics rules in a form of Equation 4 and perform learning on it so that we could learn the closing value of the unknown parameters in these physics rules. In terms of expressing such physics rules in a form of Equation 4, we instantiate the abstract function Q with these rules and for the unknown parameters, we initialize them randomly like what we did to initialize a neural network parameters. We denote these unknown parameters as P. In terms of learning, we first introduce the dataset. A data consists of two states $\vec{u}(\vec{p}, t)$ and $\vec{u}(\vec{p}, t+T)$, where $\vec{u}(\vec{p}, t)$ is the state of physics system at time t and $\vec{u}(\vec{p}, t+T)$ is the state at time $t+T$. Here T is a constant that we set manually, meaning how many time steps do we simulate. Then we put $\vec{u}(\vec{p}, t)$ into Equation 4 and iterate it for T times and thus we get the simulated results $\vec{u}(\vec{p'}, t+T)$. After this, we use a loss function L to measure the difference between simulated result $\vec{u}(\vec{p'}, t+T)$ and ground truth result $\vec{u}(\vec{p}, t+T)$. And then we use stochastic gradient descent (Robbins [2007]) and back propagation method (Rumelhart et al. [1986]) to calculate the gradient and update the parameters P in the model Q. We can approximate the backward computation as:

$$\vec{u}(\vec{i}, j-1) = \vec{u}(\vec{i}, j) + \delta_t Q'(\{\vec{u}(\vec{i}', j), \vec{i}' \in N(\vec{i})\}). \tag{5}$$

$$\frac{\partial Loss(\vec{u}(\vec{i}, j-1), \vec{u}(\vec{p}, t+T), \theta)}{\partial \theta} = \frac{\partial Loss(\vec{u}(\vec{i}, j), \vec{u}(\vec{p}, t+T), \theta)}{\partial \vec{u}(\vec{i}, j-1)} * \frac{\partial \vec{u}(\vec{i}, j-1)}{\partial \theta} \tag{6}$$

Here we abstract out the backward calculation as Q' and the parameter in Q is denoted as θ. If we could successfully learn those P in Q, we could see that by using these parameter, the model Q can simulate the physics process with a high precision in matching the ground truth observation.

Locality sensitive hashing. For a domain S of the points set with distance measure D, an LSH family is defined as

Definition 1. *A family $\mathcal{H} = \{h : S \to U\}$ is called (r_1, r_2, p, q)-sensitive for D if for any two points $x, y \in S$ and h chosen uniformly from $\mathcal{H}$ satisfies the properties:*

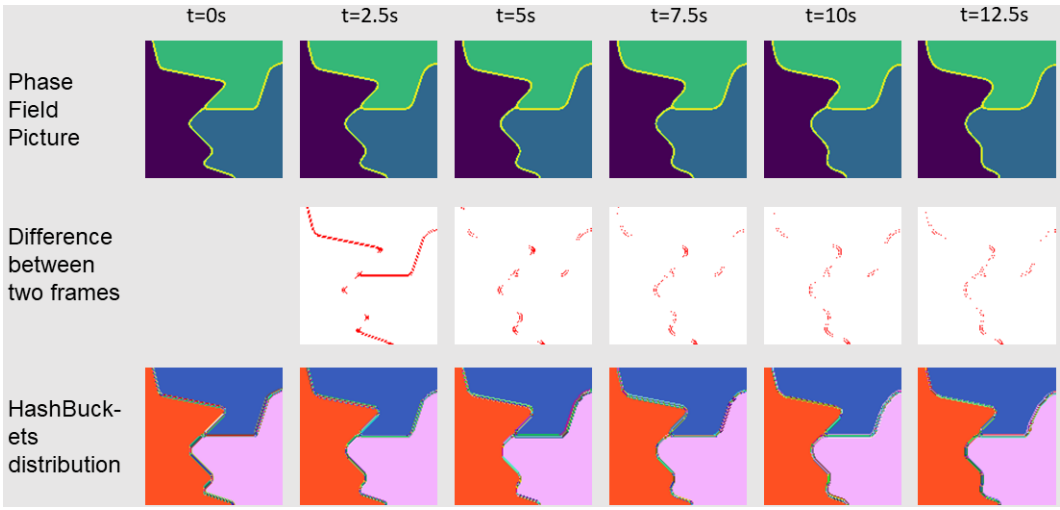

Figure 1: Intuitions that motivates LSH-SMILE. The updates of grain growth (difference between two consecutive frames, shown in the second row) only happens on the boundary of each grain. In the third row, elements with similar neighboring values share the same dynamics, and therefore can be hashed into the same hash bucket using LSH. The colors represent the hash buckets of the elements.

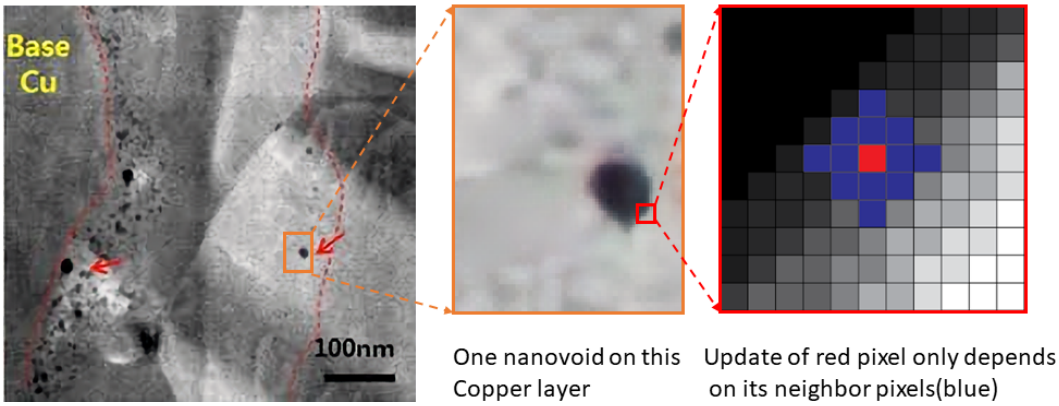

Figure 2: Left picture (from Stefanie Bremmert [2020]) shows a nanovoid, enlarged in the middle. The update of one pixel of the nanovoid (shown in the red box) only depends on the values of itself and its neighboring pixels (shown in blue).

- *if* $D(x, y) \leq r_1$, *then* $P_{h \in \mathcal{H}}[h(x)=h(y)] \geq p$.
- *if* $D(x, y) \geq r_2$, *then* $P_{h \in \mathcal{H}}[h(x)=h(y)] \leq q$

In our scheme of simulation, elements that share similar neighbor elements' value will be hashed into the same bucket and thus perform calculation simultaneously. We use Euclidean distance to be D in the definition. So, the LSH hash function has a this form $h_{\mathbf{a},b}(\mathbf{v}) = \lfloor \frac{\mathbf{a} \cdot \mathbf{v} + b}{r} \rfloor$, where **a** is a d-$dimensional$ vector with entries chosen independently from a Gaussian distribution and b is a real number chosen uniformly from the range $[0, r]$. Here r is the r_1 in the definition.

3 Intuition

By applying finite difference, the simulation process of phase-field model can be viewed as updating elements' value on an 2-dimensional matrix. The intuition of this work is inspired by three findings in the computation of phase-field simulation.

Waste computation on large area The first observation is that most of update of elements are concentrated in the boundary area. We express this in a grain growth model(Fan and Chen [1997]).

As shown in Figure 1, the boundary area (shown in yellow) is rather small compared to the entire image, making it wasteful to spend computation resources on area other than the boundary area. Inside the grains(shown in green, purple, blue), there's no update happening. This empirical observation gets supported in real data, as boundary area are usually under 1% of the entire image.

Spatial-locality dependency The second observation is that updating one element only depends on its neighbor elements, which we call spatial-locality dependency. This theoretic observation comes from the nature of taking the partial derivative of a physics variable and its approximation in finite difference methods Forsythe et al. [1961]. A more intuitive illustration is shown in Figure 2. This figure shows that the Q term in Equation 4 makes the update of $\vec{u}(\vec{i}, j)$ (shown in red in Figure 2) depends on its neighbor elemtns $N(\vec{i})$ (shown in blue in Figure 2).

Elements sharing similar dynamics The third observation is that many elements in the matrix to be udpated share similar dynamics (neighbor elements with similar value). We show this in the third row of Figure 1. Here we use locality sensitive hashing technique to hash every element on the matrix into a hash table. And those elements that belong to the same hash buckets will be printed in the same color. We can see that in this way many elements inside the grains (shown in red, blue and pink) are hashed into their own hash buckets, while elements on the boundary are hashed into different hash buckets.

4 The LSH-SMILE Algorithm

The LSH-SMILE algorithm handles the forward simulation and the backward learning problems for all elements that are hashed to a single bucket *at once*. Therefore, the updates of LSH-SMILE are conducted on hash buckets, rather than on individual elements. In this section, we first describe LSH-SMILE algorithm for forward simulation, then discuss LSH-SMILE for backward learning. The pseudo code of LSH-SMILE forward simulation is introduced in Figure 3.

Notation We first introduce the notation used in the algorithm 3. $\mathcal{B}$ is a hash table. I is input matrix. r, K, L are LSH related parameters. N is how many steps we want to simulate the physics process. We will introduce them in details in the following part.

Data structure We first introduce the data structure used in the algorithms. $\mathcal{B}$ is a hash table, in which each entry is a hash bucket. In a hash bucket, there's a representative value rep, a list of elements' coordinates $list_p$ and a list of neighbor elements' coordinates $list_n$. Here we use union-find-delete data structure(Ben-amram and Yoffe [2010]) to organize $list_p$ and $list_n$, to ensure that we can perform list operation in nearly constant time. The idea of such a hash bucket is that, according to Equation 4, every element $\vec{u}(\vec{i}, j)$ that shares similar $N(\vec{i})$ shall be hash into the same hash bucket with very high probability. And these elements share the same value, which is represented by rep. We use $list_p$ to track all these elements and $list_n$ to track the $N(\vec{i})$ of all these elements. By doing this, we can calculate the function Q in Equation 4 only one time for all the elements in this bucket, thus saving a lot of unnecessary computation time.

Encode initial state The first step of LSH-SMILE is to encode the input data into the data structure described above. At the beginning, since we don't have any prior information of the input data, we use brute-force to iterate any element in the input data and construct a hash table where each hash bucket only contain one element. Then we perform $MergeBucket$ to merge those buckets who share the same LSH hash code.

Forward function After encoding, LSH-SMILE will iterate every hash bucket to perform update given Equation 4. The update function is denoted as $Eval(\cdot)$. After update, it will update its LSH hash code to see if the update is bigger than r. If so, it means it need to be re-hash, and thus be put into $ActiveL$. After iteration on all hash buckets, LSH-SMILE will handle $ActiveL$, rehashing them one by one to make sure that every element in a hash bucket is similar to each other. Finally, it merges all the buckets with the same LSH hash code to avoid duplicate hash buckets.

Invariant In this algorithm, we want to maintain an invariant that every elements in the hash bucket share very similar $N(\vec{i})$ and its own value, and elements that belong to different hash buckets should have very different $N(\vec{i})$ and its own value. Here similarity is measured by LSH. To ensure this, we have the following features: 1)AND of OR LSH, 2)$MergeBuckets()$ and 3)$ActiveL$.

```
Func Forward(N, I, r, K, L):
    B_0 ← EncodeInitState(I, K, L);
    for t = 1, ..., N do
        B_t ← OneStep(B_{t-1}, r, K, L);
    end
    return B_N;
end
Func OneStep(B_t, r, K, L):
    ActiveL ← ∅;
    for B_{t,i} ∈ B_t do
        B_{t,i}.v_old ← B_{t,i}.v;
        B_{t,i}.v ←
        Eval(B_{t,i}.v, NeighborV(i, B_t));
        Update B_{t,i}.LSH_1, ..., .LSH_L;
        if |B_{t,i}.v − B_{t,i}.v_old| > r then
            ActiveL ← ActiveL ∪ B_{t,i}.N
        end
    end
    B_{t+1} ← B_t;
    for a ∈ ActiveL do
        Remove a from the original bucket
          that contains a in B_{t+1};
        B_a ←
          CreateBucketElem(a, K, L);
        B_{t+1} ← B_{t+1} ∪ {B_a};
    end
    B_{t+1} ← MergeBuckets(B_{t+1});
    return B_{t+1};
end
```

```
Func MergeBuckets(B, K, L):
    B' = ∅;
    for B ∈ B do
        for l ∈ {1, ..., L} do
            if there exists B' ∈ B', such that
              B.LSH_l ∈ B'.LSH_l then
                B' ← B' \ {B'};
                B ← Merge(B, B');
            end
        end
        B' ← B' ∪ {B};
    end
    for B ∈ B' do
        for l ∈ {1, ..., L} do
            B.LSH_l ←
              LSH_l(B.v, NeighborV(i, B'))
        end
    end
    return B';
end
Func EncodeInitState(I, K, L):
    B_0 = ∅;
    for a ∈ I do
        B_a ←
          CreateBucketElem(a, K, L);
        B_0 ← B_0 ∪ {B_a};
    end
    return MergeBuckets(B_0);
end
```

Figure 3: The forward simulation algorithm of N steps using locality sensitive hashing. The main function is *Forward*, which calls children functions *OneStep*, *MergeBuckets*, and *EncodeInitState*.

AND of OR LSH To ensure that every element $\vec{u}(\vec{i}, j)$ that shares similar $N(\vec{i})$ shall be hash into the same hash bucket with a probability that meets the user's standard, we introduce multi-probe LSH techniques from Lv et al. [2007] to have the control of the probability that two elements within the distance of r will be hashed into the same hash bucket. Consider a series of LSH h_i satisfying Definition 1. Construct $g_i(p) = [h_1(p), \ldots, h_K(p)]$, select L different functions $g_1, \ldots, g_L$. For one point p, hash p into ALL L buckets, denoted by $g_1, \ldots, g_L$. p and q is called to "collide" if they collide under any of the $g_1, \ldots, g_L$ functions. By doing this, we have an AND of OR LSH as follow:

Theorem 1. *Let h_i be a LSH, $g_i(p)$ = $[h_1(p), \ldots, h_K(p)]$, $H(p)$ = $[g_1, \ldots, g_L]$. $H(p)$==$H(q)$ $\iff$ any of the g_i functions has $g_i(p)$==$g_i(q)$.*

- *Under all previous definitions, if $||p - q|| \leq r$, then $Pr(p$ collide with $q) \geq 1 - (1 - p^K)^L$.*
- *Under all previous definitions, if $||p - q|| \geq cr$, then $Pr(p$ collide with $q) \leq 1 - (1 - q^K)^L$.*

Here, to match the Definition 1, r=r_1 and cr=r_2. In this way, we have an AND of OR LSH function $H(\cdot)$, and increase the collision probability from p to $1 - (1 - p^k)^l$. Notice that here r, k and l are two hyperparameters that need to be tuned during simulation.

Merge Bucket with same $H(\cdot)$ During the simulation, in each bucket, the representative value rep and its neighbors' value $N(\vec{i})$ may change, and this leads to some hash buckets have the same LSH hash code. To avoid doing duplicate computation on these buckets, we use $MergeBuckets$ function to merge those buckets with the same LSH code. And this code is given by the AND of OR LSH function $H(\cdot)$.

ActiveL The $ActiveL$ is a list of element p whose representative rep has changed so differently that the LSH hash code $H(p)$ has changed. In order to maintain the invariant, we maintain this list while iterating hash buckets and then process these element one by one.

Backward learning When doing backward propagation, the process is the same as the forward simulation, besides we add the gradient calculation part in Equation 6. And we use stochastic gradient descent to update the parameters when we get the gradient.

5 Analysis

Algorithm Runtime Analysis We here give a runtime analysis on the LSH-SMILE. Let n be the size of input I, b be the size of $\mathcal{B}$. In function $MergeBuckets(\cdot)$, line 3 to 11 iterates on the hash table, line 4 to 9 iterates L times. Every time it calculate the LSH_l hash code, it runs K different LSH. So $MergeBucket$ has time complexity as **O(bKL)**. In function $EncodeInitState(\cdot)$, line 21 to 24 iterates every element in I, line 25 calls $MergeBuckets$, thus this function is having time complexity as **O(n + bKL)**. In function $OneStep(\cdot)$, line 10 to 18 iterates on $\mathcal{B}$, thus here the time complexity is **O(b)**. And line 20 to 24 iterates on $ActiveL$, which has size smaller than b. Line 25 calls $MergeBuckets$. So the time complexity of function $OneStep$ is **O(bKL)**. In function $Forward(\cdot)$, line 2 calls $EncodeInitState(\cdot)$, line 3 to 5 calls $OneStep(\cdot)$ for N step. So the time complexity of $Forward(\cdot)$ is **O(n+N*bKL)** In comparison, a *baseline* method will directly doing brute-force for one step simulation and iterate N steps, thus having time complexity as **O(n*N)**.

According to above analysis, as long as we have $bKL \leq n$, our LSH-SMILE will have running time less than baseline method theoretically. And we show that this is true later in the experiments.

Error Bound introduced by LSH-SMILE We here give a novel proof on the error bound introduced by LSH-SMILE. We first show how to use K and L in the LSH function $H(\cdot)$ to control the collision probability in Definition 1.

Theorem 2. *For $0 < \epsilon < 0.5$, let $C = \log_\epsilon(1-\epsilon)$, then $0 < C < 1$, pick $K = \max\{1, \lceil \log_{p/q} C \rceil\}$, pick $L = \frac{1}{\log_\epsilon(1-p^k)}$, under the previous definitions, we must have*

- *For all p, q satisfying $||p-q|| \leq r$, $Pr(p$ collide with $q) \geq 1-\epsilon$.*
- *For all p, q satisfying $||p-q|| \geq cr$, $Pr(p$ collide with $q) \leq \epsilon$.*

We will show the proof in the supplementary material. In this theorem, we can see that by setting r ϵ, K and L, we can have control on the distance between two elements within a hash bucket and the probability that two elements within distance r will collide in one hash bucket. So now if we pick up two elements randomly from one hash bucket, the distance between them is less than r with the probability greater than $1-\epsilon$. That's the first error introduced by LSH-SMILE. And according to Equation 4, let's assume the abstract function Q will magnify this difference by magnitude M. So for function $OneStep(\cdot)$, it introduce errors within the range of $(0, r*M)$. And for function $Forward(\cdot)$, since it repeat $OneStep(\cdot)$ for N times, the total error introduced by LSH-SMILE is under **r*M*N**. Usually the magnitude M is around 10^1. So if we set r to be very small, the error introduced by LSH-SMILE is tolerable in simulation. We will show the forward simulation result in experiment to support this.

6 Experiments

Forward simulation We first examined the proposed LSH-SMILE algorithms in forward simulation. We apply LSH-SMILE on two phase field model. One is grain growth(Fan and Chen [1997]), another is nanovoid(Millett et al. [2011]). In each of the experiments, there are two algorithms that simulate the physics process. One is the ground truth simulation coded up using Torch framework(Collobert et al. [2011] under Modified BSD license). The ground truth simulation obeys the physics rules strictly by we directly coding them with the framework. Another is our LSH-SMILE simulation. The physics rules is implemented in the $Eval(\cdot)$ function in the pseudo code(Figure 3).

Grain Growth Simulation We evaluated the forward simulation in grain growth system(Fan and Chen [1997]). After applying finite difference approach, the update function of each grain are expressed as follow. More detail could be found in appendix.

$$\eta_{i,k+1} = \eta_{i,k} + dt * -L_i(-A\eta_{i,k} + B\eta_{i,k}^3 + 2\eta_{i,k}\sum_{i \neq j}^{N} \eta_{j,k}^2 - \kappa_i \nabla^2 \eta_{i,k}) \quad i = 1, 2, ..., N$$

where the subscript $\eta_{i,k}$ denotes the i-th grain at time k in the system. Because of using finite difference approach, the ∇^2 operator is expressed as a convolution operator where the 3x3 kernel is

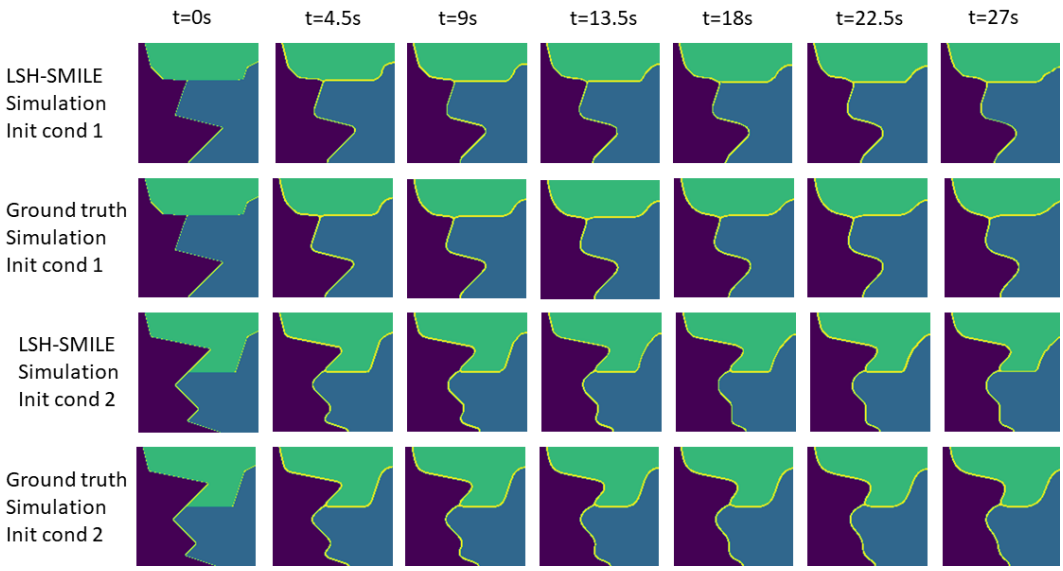

Figure 4: LSH-SMILE simulation results (the first and the third rows) closely match the ground truth (the second and the fourth rows). First two rows are initialized with condition 1. Last two rows are initialized with condition 2.

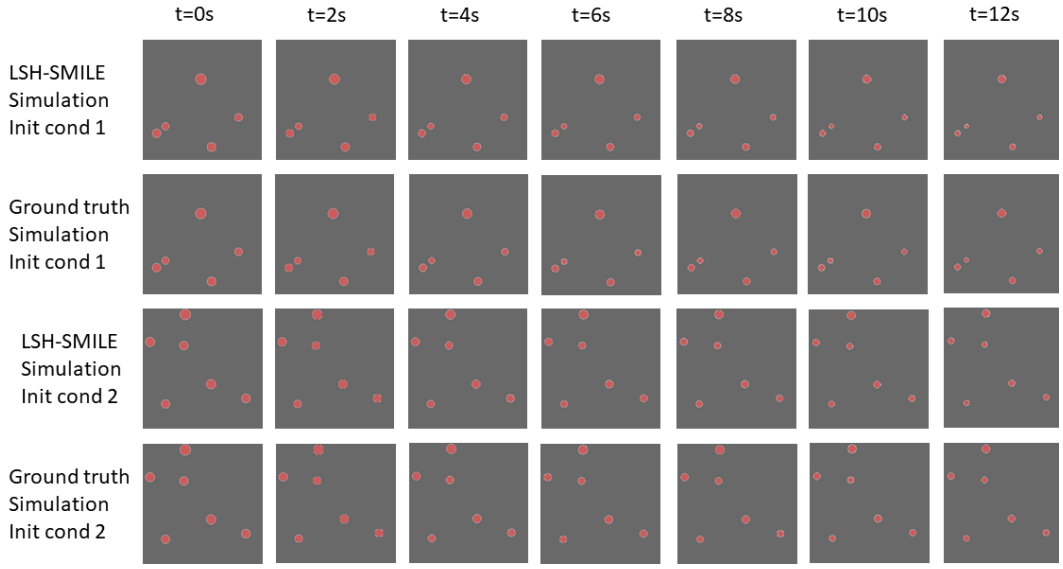

Figure 5: LSH-SMILE simulation results of nanovoids closely match the ground truth. The first two rows are initialized with condition 1. The last two rows are initialized with condition 2.

$[0\ 1\ 0,\ 1\ \text{-}4\ 1,\ 0\ 1\ 0]$. We set dt to 0.05 in this simulation, meaning that after one update, the physics state pass 0.05 second. The image size is 128 by 128. For LSH parameter, r is set to be 0.01, K is 3 and L is 10. The simulation results is in Figure 4. It shows that LSH-SMILE simulate the physics process perfectly compared to the ground truth, supporting our error bound shown above. To measure the difference, we subtract the simulated result with ground truth observation, and the difference is at the level of 10^{-5} after 180 steps.

Nanovoid Simulation We evaluated the forward simulation in nanovoid system(Millett et al. [2011]). After applying finite difference approach, the update function for field variable c_v are expressed as follow. More detail could be found in appendix.

$$c_{vt+1} = c_{vt} + dt * \frac{M_v}{N} \nabla^2 N * \left[h(\eta_t) \frac{df^s(c_{vt}, c_{it})}{dc_{vt}} + j(\eta_t) \frac{df^v(c_{vt}, c_{it})}{dc_{vt}} - \kappa_v * \nabla^2 c_{vt} \right]$$

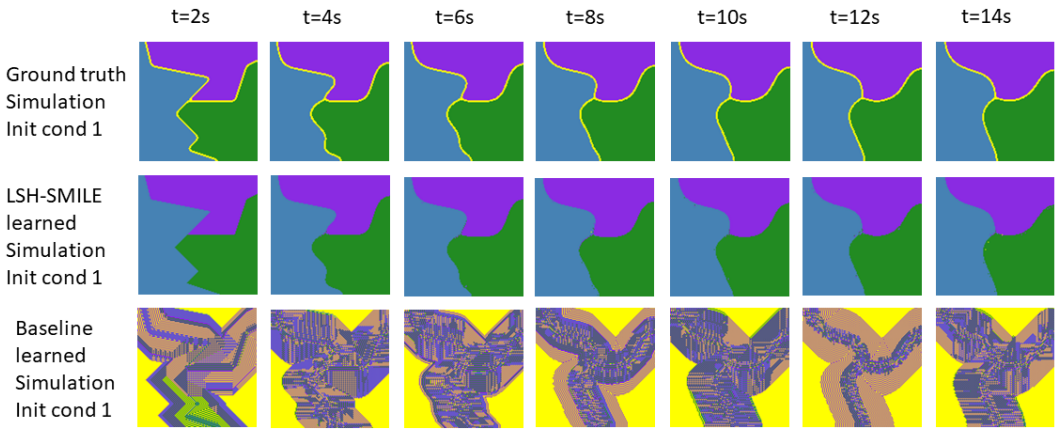

Figure 6: LSH-SMILE is able to learn the dynamics of grain growth, while the brute-force baseline cannot. The first row is the ground truth data. The second row shows the simulated dynamics of the physics model learned by LSH-SMILE, which matches the ground-truth well. The third row uses brute-force approach to learn, resulting in a physics model with incorrect dynamics.

Still, we start the simulation with two different starting frame, shown in the first column of Figure 5. We set dt to 0.1 in this simulation. The image size is 128 by 128. For LSH parameter, r is set to be 0.0001, K is 3 and L is 10. The results is in Figure 5. It shows that LSH-SMILE simulate the physics process perfectly compared to the ground truth in nanovoid as well (difference at the level of 10^{-5} after 100 steps).

Grain Growth Simulation and Learning We examined the proposed LSH-SMILE algorithms in a learning process. The dataset is synthetic and has 1700 frames of grain growth ground truth simulation results.

LSH-SMILE simulates for T=30 steps in training while baseline method simulates for T=10 steps so that both methods will take similar simulating and training time. For LSH-SMILE, it use stochastic gradient descent to update parameters, while baseline use Adam optimizer to update parameters. The ground truth parameter to be learned is L=5.0, A=B=1.0, κ=0.1. After 10 epochs of training, LSH-SMILE learned L=11.6504, A=1.98483, B=2.01454, κ=0.0834962. At the same time, baseline method learned L=9.0462, A=1.9431, B=8.8226, κ=0.8431. We can see that the parameter learned by LSH-SMILE is closer to ground truth, this is also reflected in the simulation result given the learned parameter in Figure 6.

Algorithms	Runtime (hr)	Memory Usage(mb)
GG-Sim LSH-SMILE	**3.86**	**870**
GG-Sim Baseline	16.84	1135
NN-Sim LSH-SMILE	**2.13**	**2735**
NN-Sim Baseline	53.13	3294
GG-Sim Learn LSH-SMILE	**4.5**	**889**
GG-Sim Learn Baseline	17.29	1212

Table 1: Running times of different approaches. LSH-SMILE needs less time and space compared to baseline methods.

Runtime Memory Comparison We examine the running time of both algorithms in simulation and learning. The forward simulation steps is 1700*10*30 = 510000 for both baselien method and LSH-SMILE. For learning task, both algorithms train for 10 epoch with each data point simulating for $T = 30$ steps. The results is shown in Table 1. Here GG stands for grain growth and NN stands for nanovoid. We can see that our LSH-SMILE method drastically reduce both the running time and memory usage in the same setting compared to baseline method implemented by Torch.

7 Conclusion

We propose LSH-SMILE, a unified framework to accelerate the forward simulation and backward learning of physics models, taking advantages of locality sensitive hashing. We show both theoretically and experimentally that LSH-SMILE simulates and learns physics models in a precise fashion and with reduced time and space complexity.

References

Amir Ben-amram and Simon Yoffe. A simple and efficient union-find-delete algorithm, 2010.

R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

Simos Theodore E. Dong Ming. A new high algebraic order efficient finite difference method for the solution of the schrödinger equation, 2017.

D. Fan and L.-Q. Chen. Computer simulation of grain growth using a continuum field model. *Acta Materialia*, 45(2):611–622, 1997. ISSN 1359-6454. doi: https://doi.org/10.1016/S1359-6454(96)00200-5. URL `https://www.sciencedirect.com/science/article/pii/S1359645496002005`.

Marc Finzi, Ke Alexander Wang, and Andrew Gordon Wilson. Simplifying hamiltonian and lagrangian neural networks via explicit constraints. *Advances in Neural Information Processing Systems*, 33, 2020.

G. E. Forsythe, W. Wasow, and W. Nachbar. Finite-difference methods for partial differential equations. 1961.

Samuel Greydanus, Misko Dzamba, and Jason Yosinski. Hamiltonian neural networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL `https://proceedings.neurips.cc/paper/2019/file/26cd8ecadce0d4efd6cc8a8725cbd1f8-Paper.pdf`.

Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, page 950–961. VLDB Endowment, 2007. ISBN 9781595936493.

Paul C Millett, Anter El-Azab, Srujan Rokkam, Michael Tonks, and Dieter Wolf. Phase-field simulation of irradiated metals: Part i: Void kinetics. *Computational materials science*, 50(3): 949–959, 2011.

T. Niu, M. Nasim, R. Annadanam, C. Fan, Jin Li, Zhongxia Shang, Y. Xue, Anter El-Azab, H. Wang, and X. Zhang. Recent studies on void shrinkage in metallic materials subjected to in situ heavy ion irradiations. *JOM*, 72, 09 2020. doi: 10.1007/s11837-020-04358-3.

H. Robbins. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 2007.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, 1986. doi: 10.1038/323533a0. URL `http://www.nature.com/articles/323533a0`.

Arman Shojaei, Ugo Galvanetto, Timon Rabczuk, Ali Jenabi, and Mirco Zaccariotto. A generalized finite difference method based on the peridynamic differential operator for the solution of problems in bounded and unbounded domains. *Computer Methods in Applied Mechanics and Engineering*, 343:100–126, 2019. ISSN 0045-7825. doi: https://doi.org/10.1016/j.cma.2018.08.033. URL `https://www.sciencedirect.com/science/article/pii/S0045782518304316`.

Kay Wurdinger Thomas Vágó Tobias Bernhard Frank Bruning Roger Massey Stefanie Bremmert, Laurence Gregoriades. Developments in electroless copper processes to improve performance in amsap mobile applications. 2020.

Ai Ishikawa Takashi Matsubara and Takaharu Yaguchi. Deep energy-based modeling of discrete-time physics. In *Advances in Neural Information Processing Systems 33 (NeurIPS2020)*, 2020. URL `https://arxiv.org/abs/1905.08604`.

Yuanyuan Wang, Yan Gu, and Jianlin Liu. A domain-decomposition generalized finite difference method for stress analysis in three-dimensional composite materials. *Applied Mathematics Letters*, 104:106226, 2020. ISSN 0893-9659. doi: https://doi.org/10.1016/j.aml.2020.106226. URL `https://www.sciencedirect.com/science/article/pii/S0893965920300197`.

Yexiang Xue, Md. Nasim, Maosen Zhang, Cuncai Fan, Xinghang Zhang, and Anter El-Azab. Physics knowledge discovery via neural differential equation embedding. *Technical Report*, 2021.

Checklist

The checklist follows the references. Please read the checklist guidelines carefully for information on how to answer these questions. For each question, change the default **[TODO]** to [Yes] , [No] , or [N/A] . You are strongly encouraged to include a **justification to your answer**, either by referencing the appropriate section of your paper or providing a brief inline description. For example:

- Did you include the license to the code and datasets? [Yes] See Section **??**.
- Did you include the license to the code and datasets? [No] The code and the data are proprietary.
- Did you include the license to the code and datasets? [N/A]

Please do not modify the questions and only use the provided macros for your answers. Note that the Checklist section does not count towards the page limit. In your paper, please delete this instructions block and only keep the Checklist section heading above along with the questions/answers below.

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] see section 6 and 5
   (b) Did you describe the limitations of your work? [Yes] see section 5
   (c) Did you discuss any potential negative societal impacts of your work? [No]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes] see section 5
   (b) Did you include complete proofs of all theoretical results? [No] they will be uploaded in supplementary material
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] they will be provided in the supplementary material
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] they will be provided in the supplementary material
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] they will be provided in the supplementary material
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] We use Torch as our baselien implementation
   (b) Did you mention the license of the assets? [Yes]
   (c) Did you include any new assets either in the supplemental material or as a URL? [No]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A] We use our own synthetic data
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [No]

5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]