CARNEGIE MELLON UNIVERSITY
COLLEGE OF ENGINEERING

Final Project Report December 14, 2023

Smooth Particle Mesh Ewald in CUDA.jl

by

Ethan Meitz Nick Hattrup

Abstract: Smooth particle mesh Ewald (SPME) is a method of calculating long range interactions like gravity and electrostatics for molecular dynamics (MD) simulations. This calculation is essential to get accurate simulations, but is often the most expensive component and is a natural candidate for parallelization. To accelerate the SPME calculation we implement the energy and force calculation on a GPU using `CUDA.jl`. This requires three kernels: the real space sum, charge interpolation and the reciprocal space sum. In the real space sum we demonstrate linear scaling and a 4x speed-up compared to a commercial MD code by building a spatially sorted neighbor list. Combining all three kernels, we achieve a 6x speed-up for the largest system tested. For even larger systems our code will continue to outperform the CPU implemented and achieve better speed-ups greater than the number of CPU cores.

Keywords: SPME, CUDA, GPU, Ewald Sum, Molecular Dynamics, Julia

Our code can be found at this Github repository: https://github.com/ejmeitz/CUDA$_{P}3M$

Work Distribution: Ethan completed the real space sum and benchmarks, Nick completed the reciprocal space sum and charge interpolation. All other work (writing etc.) was split evenly.

1 Introduction

Molecular dynamics (MD) is a computational method for the study of matter and materials at or below the micro-scale. MD can model a variety of different interactions between matter: for example, short range Van der Waals forces and Coulombic charge interactions. Different interactions have different decay behaviors as a function of distance. Revisiting above, the former and latter interactions have a decay relation of r^{-6} and r^{-1}, respectively. The much slower decay rate of Coulombic interactions makes computing the associated interaction potential and forces precarious. When utilizing periodic images in an MD system, a standard practice, summing total interactions to estimate the energy for instance is conditionally convergent, and can require summing over multiple periodic images to get within an acceptable estimation error.

$$E_{\text{total}} = \frac{1}{2} \sum_{i,j=1}^{N} {\sum_{\mathbf{n} \in \mathbb{Z}^3 \setminus \mathbf{0}}}' \frac{q_i q_j}{|\mathbf{r}_{ij} + \mathbf{n}L|} \tag{1}$$

Where L is the set of unit vectors defining the system. Given N particles, direct summation is $O(N^2)$. To make this summation more efficient, Ewald proposed splitting the summation of interactions into short range "real-space" and long range "reciprocal-space" terms. Noting the following relation:

$$\frac{1}{r} = \frac{f}{r} + \frac{1-f}{r} \tag{2}$$

A hypothetical function $f(r)$ that changes rapidly over short distances but slowly over long distances would allow for different summation approaches over the two terms. The original Ewald summation of the potential is:

$$E_{\text{total}} = E^{\text{r}} + E^{\text{k}} + E^{\text{s}} + E^{\text{d}} \tag{3}$$

For brevity we only highlight E^{r} and E^{k}:

$$E^{\text{r}} = \frac{1}{2} \sum_{i,j=1}^{N} {\sum_{\mathbf{n} \in \mathbb{Z}^3 \setminus \mathbf{0}}}' \frac{q_i q_j \operatorname{erfc}(\beta |\mathbf{r}_{ij} + \mathbf{n}L|)}{|\mathbf{r}_{ij} + \mathbf{n}L|} \tag{4}$$

$$E^{\text{k}} = \frac{1}{2V} \sum_{\mathbf{k} \neq \mathbf{0}} \frac{4\pi}{\mathbf{k}^2} \exp\left(-\frac{\pi^2 \mathbf{m}^2}{\beta^2}\right) |\tilde{\rho}(\mathbf{k})|^2 \tag{5}$$

Where charge density $\tilde{\rho}(\mathbf{k})$ is given by a Fourier transform. is referred to as the "Ewald parameter", and optimal tuning gives complexity $O(N^{3/2})$ for total summation. $O(N^{3/2})$ comes from reciprocal space component E^{k}. Building off of Ewald summation, replacing a full Fourier transform with a fast Fourier transform (FFT) brings the total complexity down to $O(N \log N)$. However, FFTs are discrete Fourier transforms, requiring interpolation of the charges in the system onto a discrete grid. SPME is one method of how to interpolate the charges onto a mesh grid. The reciprocal term E^{k} is reexpressed as:

$$E^{\mathsf{k}} = \sum_{\mathbf{m}\neq\mathbf{0}} \frac{\exp\left(-\frac{\pi^2\mathbf{m}^2}{\beta^2}\right)}{\mathbf{m}^2} S(\mathbf{m})S(-\mathbf{m}) \tag{6}$$

Where $S(\mathbf{m})$ is termed the "structure factor" and m are reciprocal lattice vectors. $S(\mathbf{m})$ is defined as:

$$S(\mathbf{m}) = \sum_{j=1}^{N} q_j e^{2\pi i q_j \cdot r_j} \tag{7}$$

The exponential in the sum is then approximated using B-Cardinal splines. As:

$$S(\mathbf{m}) \approx b_1(m_1)b_2(m_2)b_3(m_3)\mathcal{F}(Q) \tag{8}$$

With Q and b defined as:

$$\begin{aligned} Q(k_1,k_2,k_3) = \sum_{i=1}^{N}\sum_{\vec{n}} q_i M_n(u_{1i}-k_1-n_1K_1) \\ * M_n(u_{2i}-k_2-n_2K_2)M_n(u_{3i}-k_3-n_3K_3) \end{aligned} \tag{9}$$

$$b_j(m_j) = \frac{\exp\left(2\pi i\frac{m_j}{K_j}(n-1)\right)}{\sum_{k=0}^{n-2} M_n(k+1)\exp\left(2\pi i\frac{m_j k}{K_j}\right)} \tag{10}$$

These splines are noteworthy in that this interpolation is differentiable, allowing for analytical expressions of the forces for MD. SPME is consequently the standard method for modeling MD systems with Coulombic interactions. Given the importance of this method, we saw it interesting and relevant to implement this algorithm

on a GPU, in particular, with the open source programming language Julia. For comparison, we bench-marked against the MD package LAMMPS, which uses an similar interpolation scheme known as particle-particle-ewald PPE.

2 Approach

To implement SPME on a GPU, we used the `CUDA.jl` wrapper of the CUDA C-API [1]. This warpper provides the conveniences of a high level language (Julia) while retaining all the kernel programming functionality of CUDA. `CUDA.jl` helped us avoid common issues like segmentation faults, as there are no raw pointers in Julia, and accelerated the development process by removing code compilation.

The SPME code contained three CUDA kernels which map to the majors steps of the SPME algorithm: the real space sum, charge interpolation and the reciprocal space sum.

2.1 Real Space Sum

The implementation of the real space sum follows the OpenMM kernel for non-bonded interactions [2]. This kernel is generic and capable of calculating the interaction of any potential, not just electrostatics. Before execution this kernel requires a custom neighbor list that maps well onto GPU hardware. The steps to build this neighbor list are:

1. Divide simulation box into voxels of width *w*, and assign atoms to each voxel like in Fig 1. $\mathcal{O}(N)$

2. Map each voxel onto a Hilbert curve that starts at (0,0,0) as shown in Fig 1, and sort based on the distance along the curve. This spatially sorts the voxels. $\mathcal{O}(N)$

3. Iterate through the sorted voxels and use this order to reorganize the atom data to match the spatial sorting.

4. Divide the list of atoms into groups of 32 (matching NVIDIA warp size) and calculate a bounding box for each group.

5. Calculate the distance between each pair of bounding boxes. If the distance

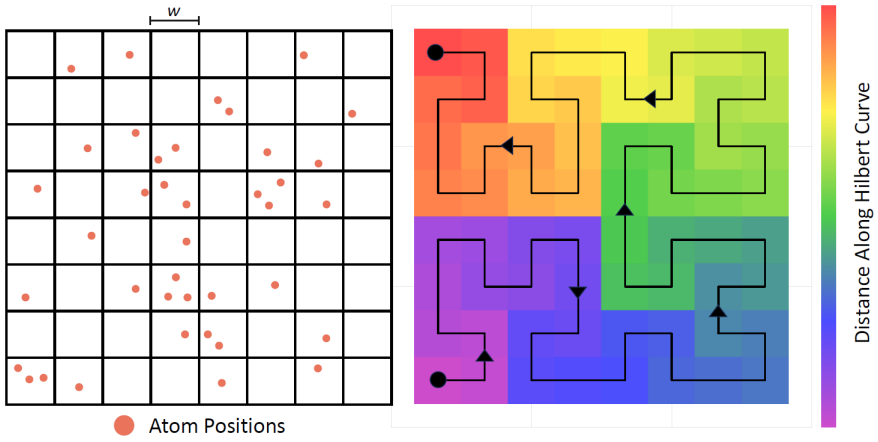


Figure 1: Atoms assigned to voxels in a 2D domain (left) and mapped onto a Hilbert curve that according to the voxels distance along the curve (right).

is larger than a chosen cutoff plus a skin thickness flag this pair as non-interacting.

6. For each pair of bounding boxes calculate the distance between the first box and all of the particles in the second box. If the distance is more than a chosen cutoff radius flag that atom as non-interacting.

The final step to build the neighbor list is $\mathcal{O}(N^2)$; however, the neighbor list only needs to be re-built every 100 time steps. The computational complexity of the real space sum will be dominated by the GPU kernel. Note that the neighbor list is built on CPU, but these kernels could be ported to GPU as well. The CUDA kernel is launched with one thread block for every pair of interacting bounding boxes each with 32 threads. Within each block the positions of the 32 atoms in each bounding box are moved into shared memory (384 bytes). Then one of three kernels is selected to calculate the force and energy. If the pair of bounding boxes contains the same two bounding boxes, then only half of the interactions need to be calculated to avoid double counting (528 interactions). The next scenario checks the flag calculated in step 6. If more than 24 atoms are flagged as non-interacting, the kernel only calculates the non-flagged interactions. This requires a reduction across the warp at each step as it cannot be guaranteed that the warp will not diverge. Otherwise, all 1024 interactions are calculated. Inside of each warp the interactions are calculated in a staggered fashion, like in Fig 2, so that

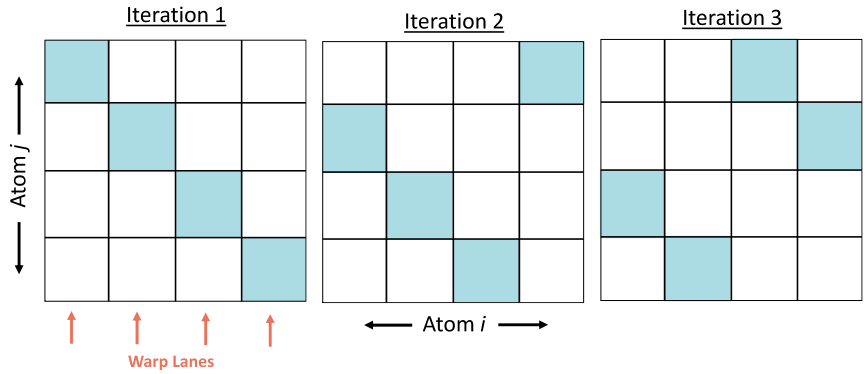


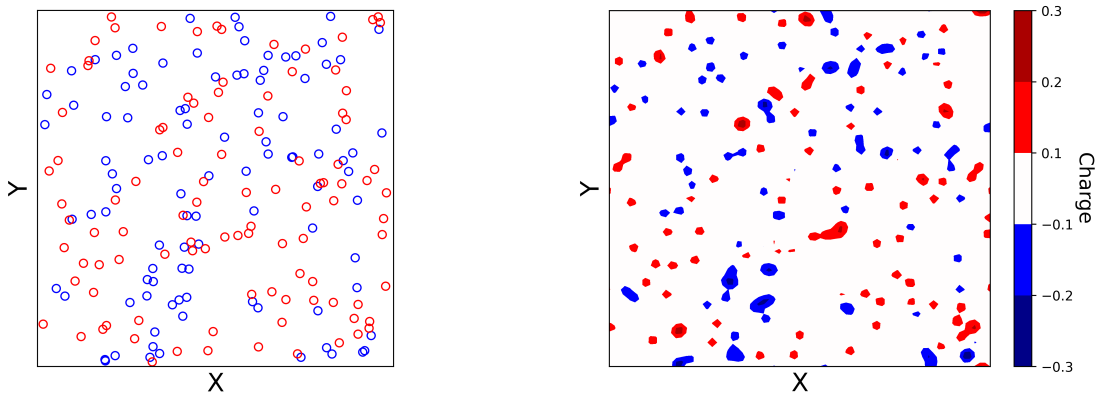
Figure 2: Each warp lane calculates the interaction between atom *i* and atom *j*. The blue blocks represent a energy/force calculation. When the loop advances its wraps around to the beginning if the index overflows the warp size.

there is no warp divergence and each thread calculates interactions independent of other threads.

2.2 Charge Interpolation

Before the reciprocal space sum can be performed the discrete set of point charges must be interpolated onto a regular grid of mesh points using Eqn. 9. This allows us to use a Fast Fourier Transform (FFT) to approximate the structure factor $S(\vec{m})$ in Eqn. 8 and accumulate the reciprocal space energies in $\mathcal{O}(Nlog(N))$. Figure 3 shows the application of Eqn. 9 to spread out a set of randomly chosen point charges. The use of *n*th order B-splines guarantees the interpolated charge field is n-2 times differentiable which means forces can be analytically derived from the energy expressions in Eqns. 5, 6.

To interpolate the charge on GPU we assigned one thread to every atom in the system with a block size of 64. The small block size was chosen as the number of atoms in system was on the order of 1000 and it is better to launch multiple blocks to fully saturate the GPU's compute capabilities. Each thread calculated an atom's contribution to the n^3 neighboring mesh points. Because multiple atoms can contribute to the same cell we used the atomic add operation to avoid data races.



(a) Random point charges in space.

(b) Charges interpolated onto grid.

Figure 3: An example of interpolating a set of discrete charges onto a regular mesh grid. This has the effect of ”spreading” the point charges out.

2.3 Reciprocal Space Sum

The reciprocal space took advantage of the high-level Julia wrapper of the FFTW library to call the CuFFT library [1, 3]. We simply moved the interpolated charge array to the GPU and called the appropriate FFT and IFFT functions. To optimize the calculation, we chose the mesh spacing to be a factor of 2, 3, or 5 as stated by the CUDA documentation.

3 Results

3.1 Real Space Sum

First, we will look at the performance of the real space and reciprocal space kernels separately to understand their scaling. Figure 4 shows the scaling of our GPU real space kernel with the number of atoms compared to LAMMPS on 1 CPU-core. Both codes exhibit linear scaling, but the GPU code has a lower constant factor and becomes a better option than the CPU around 1000 atoms.

For the last data point, which represents a 9x9x9 unit cell salt crystal with 5832 atoms, we also ran several MPI simulations of LAMMPS to determine how many cores it took to match a single GPU. Due the highly optimized nature of LAMMPS, the CPU code scaled almost perfectly with the number of atoms and only required

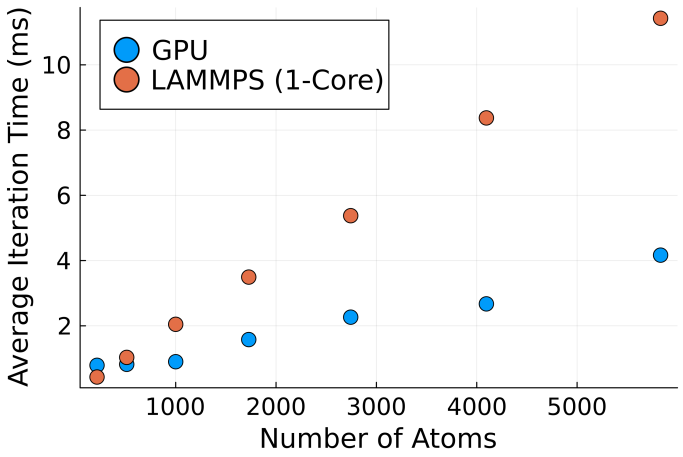


Figure 4: Scaling of LAMMPS on 1 CPU core compared to the real space kernel on an NVIDIA 2080 GPU

4 cores to match the GPU. This may not sound like a big win for the GPU; however, molecular dynamics code does not map well to the GPU and single digit speed-ups are expected [4]. Furthermore, due to the different constant factors the speed-up gained by using a GPU will only continue to grow. When simulating a macro-molecular system, such as a protein with 100,000s of atoms, a GPU is the clear choice.

3.2 Charge Interpolation

The charge interpolation kernel was tested with the same salt crystals and showed the expected linear scaling with number of atoms as shown in Fig 5. The first few data points do not scale well as the kernel launch time is comparable to the compute time and only a few streaming multiprocessors (SMs) are active.

3.3 Reciprocal Space Sum

The FFT is a well known algorithm and scales as Nlog(N), where N is the mesh size, so we chose to look at the scaling of the FFT as the error tolerance changes. In the SPME algorithm one of the inputs is the acceptable level of error. This is typically around 1e-4 but can change depending how accurate the simulation needs to be. Figure 6 shows the scaling as we decrease the error tolerance for a 4 unit cell system. This relationship is complicated as the mesh size is a non-linear

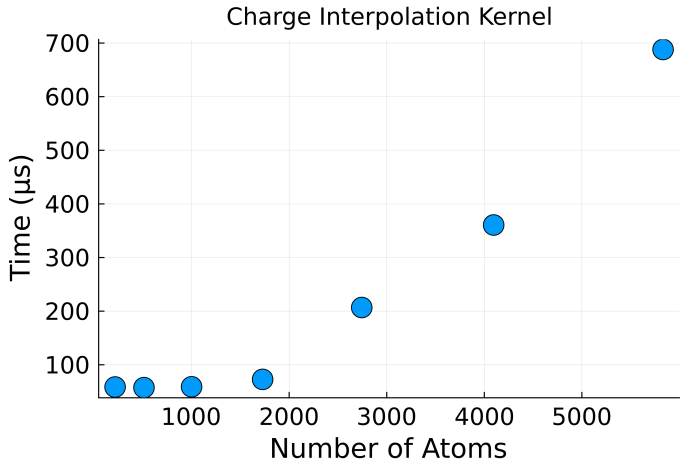


Figure 5: Charge interpolation kernel scaling with number of atoms.

function of the error-tolerance and box size. However, we can clearly see that for the common range of errors the FFT run-time does not appreciably change.

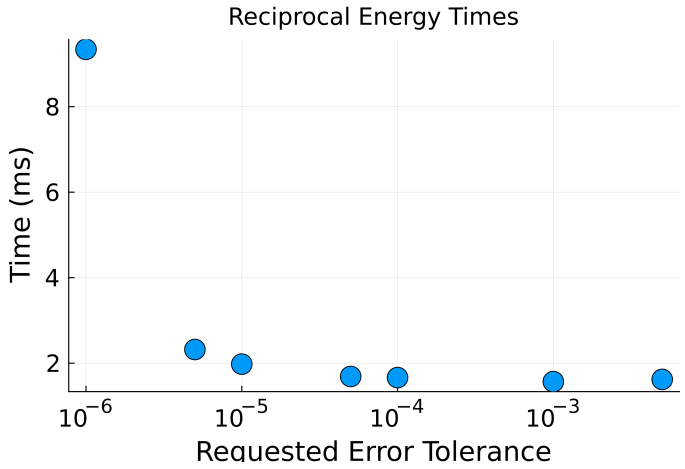


Figure 6: FFT scaling with error tolerance for a system with 4 salt unit cells

3.4 Overall Performance

Combining the real space, charge interpolation and reciprocal space into one code we observe a final scaling that matches LAMMPS with a lower constant factor. Figure 7 shows the comparssion of the energy loop in LAMMPS compared to our energy loop on a GPU. Both show NlogN scaling demonstrating the importance of

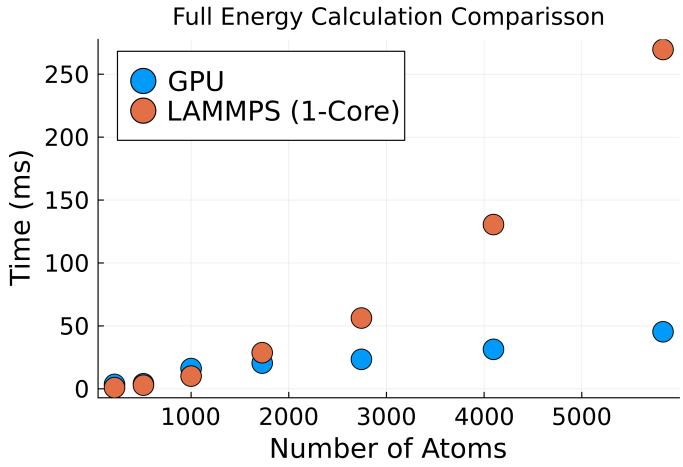


Figure 7: Full simulation benchmark of GPU code compared to LAMMPS on 1 CPU core.

the FFT code. Our GPU code outperforms the LAMMPS simulation with a speed-up of 6x at 9 unit cells. The performance gap will continue to grow as the system size increases demonstrating the importance of using a GPU to run large MD simulations. For the systems studied here the adding extra CPU cores will allow the CPU code to easily match the GPU code; however, in systems with many tens of thousands of atoms the GPU speed-up will be more than the number of cores on a CPU.

4 Conclusions

We successfully implemented SPME on a GPU using the CUDA.jl library. Our code not only demonstrates the superiority of GPU's for large-scale MD simulations, but also the maturity of the Julia ecosystem for GPU programming. In the salt system tested, the GPU routinely outperformed the LAMMPS program when the system had more than 1000 atoms with a maximum speed up of 6x for the 9 unit cell system. In the future, we would like to add support for multiple GPU's and also make it agnostic of which GPU back-end is used (e.g. CUDA vs. AMD). Furthermore, optimizing the real-space kernel would have benefits beyond SPME. This kernel is generally useful for all non-bonded MD interactions and could accelerate other portions of the simulation. With these changes our code could rival the speed and flexibility of other GPU codes like OpenMM and GROMACS.

References

[1] Tim Besard, Christophe Foket, and Bjorn De Sutter. “Effective Extensible Programming: Unleashing Julia on GPUs”. In: *IEEE Transactions on Parallel and Distributed Systems* 30.4 (Apr. 2019), pp. 827–841. ISSN: 1045-9219. DOI: 10.1109/TPDS.2018.2872064.

[2] Eastman Peter and Vijay S. Pande. “Efficient nonbonded interactions for molecular dynamics on a graphics processing unit”. In: *Journal of Computational Chemistry* 31.6 (Apr. 2010), pp. 1268–1272. ISSN: 01928651. DOI: 10.1002/jcc.21413.

[3] M. Frigo and S.G. Johnson. “The Design and Implementation of FFTW3”. In: *Proceedings of the IEEE* 93.2 (Feb. 2005), pp. 216–231. ISSN: 0018-9219. DOI: 10.1109/JPROC.2004.840301.

[4] Sandia National Labs. *LAMMPS Benchmarks*.