Comparing GPU and CPU energy efficiency in the context of MD simulations

Aniketh Reddy Kanukula, Lucas Marthin Klemensen, Noah Elkhatali,

Solmon Ahmed Alice, S. Stefanovich Silva

Basic Project 1:

Applications of Science in Technology and Society

Supervisor: Thomas Schrøder

Roskilde University

Wednesday 17th December, 2025

Abstract

As GPUs become central to modern scientific computing, questions about their growing energy use are becoming more important. In this project, we compare the energy efficiency of GPUs and CPUs when running molecular dynamics simulations. Using gamdpy (built for GPUs) and LAMMPS (originally CPU-focused), we run the same benchmark on both an RTX 4090 GPU and a Ryzen 9 7900 CPU while measuring power consumption with an external power meter. We also examine how energy use changes with the number of simulated molecules and what kinds of tasks suit GPUs better than CPUs. Our results show that GPUs can be significantly more energy-efficient for large, highly parallel simulations, while CPUs remain a good fit for smaller or less parallel workloads.

Table of Contents

Table of abbreviations

API	Application Program Interface
CPU	Central Proccesing Unit
CUDA	Compute Unified Device Architecture
FLOPS	Floating point Operations Per Second
GAMDPY	GPU Accelerated Molecular Dynamics in Python
GPU	Graphics Proccesing Unit
HPC	High Performance Computing
JIT	Just-In-Time Compilation
LAMMPS	Large-scale Atomic/Molecular Massively Parallel Simulator
LJ	Lennard-Jones
MD	Molecular Dynamics
MPI	Messaging Passing Interface
TDP	Thermal Design Power
SM	Streaming Multiproccessor

1 Introduction and research question

The world of computing has taken a storm in the last 10 years. Rapid advancements in artificial intelligence, rendering, and High Performance Computing (HPC) have led to even more explosive growth in the GPU (Graphics Processing Unit) market. The new paradigm is one where GPUs are the most vital component of any HPC center. Our group is made up of gamers and hardware enthusiasts that have seen this growth with our own eyes.

During our introduction period to RUC, we learned of their in-house HPC and their multi-decade effort to utilize GPUs for scientific computing tasks. One of these efforts has been gamdpy (pronounced gam-dee-pai), a python-based Molecular Dynamics (MD) simulation software package designed around ease of programming and utilizing GPUs for much faster results. We were exposed to examples where "*The duration of the longest simulation was four months, which with traditional central processing unit (CPU) computing would have taken several years.*"[16] thereby greatly aiding in research.

However, the rise of GPUs and HPC have not been without controversy with large media attention towards their ever increasing electricity usage and the negative effects for the local communities around them. Since our semester theme is "Applications of Science in Technology and Society", we wanted to investigate, if efforts like gamdpy sacrifice energy efficiency for computational speed, or if they should be considered the more environmentally friendly option. With the help of our supervisor, we decide to investigate the following question:

In the context of molecular dynamics simulations, is GPU or CPU computation more energy efficient?

We aim to explore the relationship between GPU utilization and power consumption. Specifically, we want to determine whether GPU workload is proportional to energy. To investigate this, we will vary the number of molecules processed and observe the impact on power consumption.

How does the number of molecules affect energy efficiency?

Fundamentally, part of the aim of this project is to learn about the differences of GPU and CPU programming. One could wonder, if GPU programming is much more power efficient, why continue working with CPU architectures at all? In addition to backward compatibility, there must be strengths and weaknesses in each compute platform. This is a question that is further investigated in this report:

What are the limitations of computing on a GPU?

We plan to answer these questions through a simple experiment, where we execute a benchmark on gamdpy with and without its autotuner feature and the same benchmark in LAMMPS, an estab-

lished competitor, with its default CPU-centric configuration and with a GPU-acceleration package extension, while measuring the total power. In addition, the theory section of the report will provide information on specific algorithms used by MD packages, as well as the structure and ideas behind GPU programming in contrast to CPU programming. These will provide a framework for understanding and interpreting our results in relation to the questions.

2 Theory

2.1 Introduction to molecular dynamics

Molecular Dynamics (MD) are software packages developed with the intention to simulate and test the behavior of molecular structures. These could range from analyzing proteins folding for medical research, to understanding the inner workings of melting a crystalline structure. Broadly speaking, simulations are an important tool for science to progress. Researchers can use them to inspect processes at a single-particle level and use the data provided by these models to make conjectures and hypotheses about reality, which can then be later tested with more traditional experiments and thus continue powering the wheel that moves forward mankind's understanding of the world.

Despite its relevance to science, the costs of these efforts and the issue of energy consumption is increasingly important to be questioned by both researchers and members of the general public. Many scientific simulation tasks have a large computational cost associated with them often requiring large HPC centers that siphon energy. Therefore, it is important to design, iterate and benchmark software packages with the aim of optimizing computational use and energy consumption.

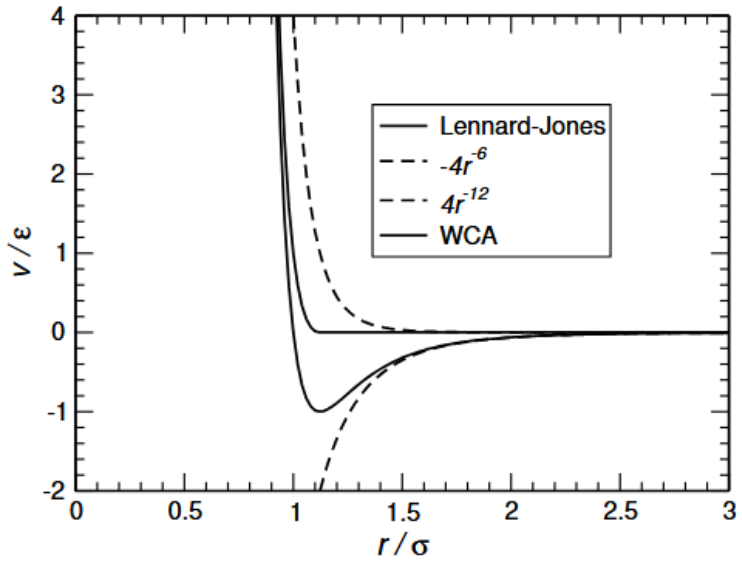

Figure 1: Lennard-Jones pair potential, the primary model used by gamdpy and LAMMPS. Shown in units of energy ϵ[10]

2.2 An overview of MD algorithms

In order to understand the following sections, an overview and explanation of some algorithms and how to implement MD simulations is needed. Consider the case of particle, i, out of a large number, N, in order to model its movement in a small *timestep* forward, Δt, one must calculate the total force acting on this particle, $\vec{F}_i = \sum_{j \neq i} \vec{F}_{ij}$. Relevant forces to consider depend on the aim of the simulation, as it is too computationally taxing to take everything into account. Thus, it is important to only consider the most important forces. [10]

The scale of the simulation affects which forces are more appropriate to use. In smaller scales quantum effects have a high influence over particles, in larger scales a more traditional Newtonian model can be used. We will primarily concern ourselves with MD software modeling classical force fields, mirroring what both gamdpy and LAMMPS do. Once we have chosen a model for the simulation, then Newton's second law can be used to establish systems of differential equations: [10]

$$\vec{F}_i = \sum_{j \neq i}^{N} \vec{F}_{ij}$$

Using Newton's second law:

$$\sum_{i \neq j}^{N} \vec{F}_{ij} = m \cdot \vec{a}_i(t)$$

$$\Updownarrow$$

$$\frac{d\vec{v}_i(t)}{dt} = \vec{a}_i(t) = m \cdot \sum_{j \neq i}^{N} \vec{F}_{ij}$$

$$\frac{d}{dt}\vec{x}(t) = \vec{v}_i(t)$$

Where $\vec{x}(t)$ is a function describing the position of the particle.

Thus, in order to find the position where the particle will be after Δt, one must find a method to numerically integrate. There are a widespread variety of techniques to implement an integrator that gives us the desired result - one such example is the leap-frog algorithm.

The input for the leap-frog algorithm, the force F, is important while optimizing MD simulations. The simplest implementation of an interaction module iterates through the particles, calculates the force on each particle and gives this to the integrator. This is a $O(N^2)$ operation and is therefore undesirable for large systems. The primary model for these interactions, the Lennard-Jones potential, has the property that as distance, r, increases the total potential tends to 0 and therefore the force

Algorithm 1: A basic implementation of the leap-frog algorithm for one particle i

```
t ← 0;
while t < t_max do
    a ← F/m ;                    /* where F is force and m is mass of particle i */
    v ← v + a · Δt;
    x ← x + v · Δt;
    t ← t + Δt;
end
```

tends to 0 (see figure 1). A way of only considering relevant interactions is designing a cutoff range, r_c, and only considering forces from particles that are closer than the cutoff range, $r < r_c$.

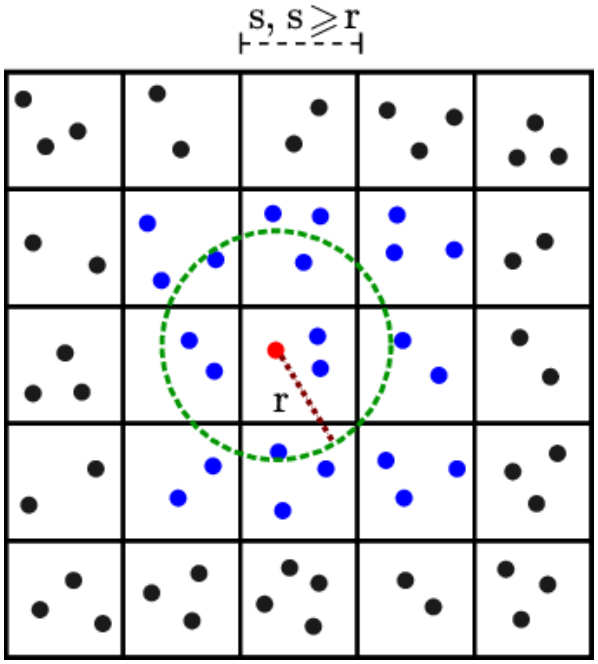

Figure 2: Neighbor list creation example using linked lists, in this case the red particle only if the blue particles are neighbors[9]

The solution used by gamdpy and LAMMPS is to create the neighbor list. The neighbor list is a data structure that keeps track of which particles are close to each other, thus when computing the total force on a particle, the calculation will only include the particles' neighbors[19]. The particle pairs with distance $r < r_c$ are neighbors. The most basic implementation to create a neighbor list iterates through all N particles and calculates the distances between them, before then adding the ones that

fit the criterion onto a neighbor list. This method has a running time complexity of $O(N^2)$. There is a problem however, as particles moving between timesteps might put some beyond the cutoff point, and others can move into the cutoff point, a small skin distance can be added Δ_s to keep the neighbor list valid for longer. In addition, a $O(N)$ check can be implemented to check if the particles have moved half the skin and when this is the case a new neighbor list is created. This method as a whole has a run time complexity of $O(N^2)$, similar to the method above, albeit it is still better as the neighbor-list is used for multiple timesteps.[19]

Algorithm 2: $O(N^2)$ Neighbor list

```
for i<N do
    for j < N ∧ j ≠ i do
        r² ← (x_i − x_j)² + (y_i − y_j)² + (z_i − z_j)²;
        if r² < (r_c + Δ_s)² then
            add j to neighbor list of i.
        end
    end
end
```

The other approach to creating and maintaining a neighbor list is to use a linked list. A linked list, or cell list, is a list-like data structure where each element has a pointer to the next element. In a double linked list, elements also have pointers for the previous element[3], see 3. Like previously, we must find all particles that are $r < r_c + \Delta_s$ away. This is done by splitting the simulation space into cells of size $\Delta x > r_c + \Delta_s$, meaning all the neighboring particles are never further than one cell away, see Algorithm 2. Our linked list connects these neighboring cells and reduces the number of computations used to find the neighbors. This method has a run time complexity of $O(N \cdot N_{nb})$, vastly outperforming the earlier method $O(N^2)$8.

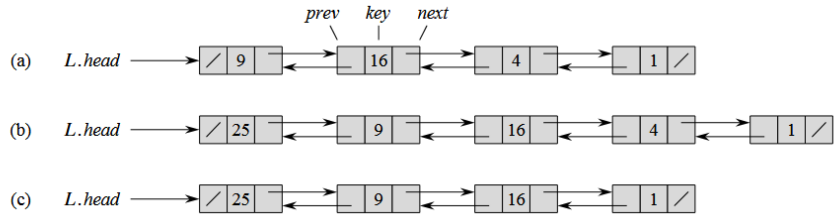

Figure 3: A double linked list, adding and removing elements are shown in (b) and (c)[3]

Algorithm 3: $O(N \cdot N_{nb})$ Neighbor list

The cell list has already been populated.

```
for i<N do
    cellIndex ← getCellIndex(x_i, y_i, z_i);
    neighborCells ← getNeighbors(cellIndex);
    N_nb ← Length of neighborCells;
    for j < N_nb ∧ j ≠ i do
        r² ← (x_i − x_j)² + (y_i − y_j)² + (z_i − z_j)²;
        if r² < (r_c + Δ_s)² then
            add j to neighbor list of i.
        end
    end
end
```

In conclusion that leaves us with the following knowledge: MD software packages have the following components: an interactions module that calculates the total force on a particle, an integrator that calculates movement from the total force, and a neighbor list that efficiently stores which particles are neighbors, and therefore which particles to calculate forces from, which can be created and maintained with either a $O(N^2)$ approach or a $O(N \cdot N_{nb})$ approach.

2.3 CPU vs GPU computation contrast

In order to gain a better understanding of the myriad of optimization and design choices employed by MD packages like gamdpy and LAMMPS, it is important to have an overview of the fundamental differences in their design and their consequences, as well as a complete understanding of the strengths and weaknesses of CPU and GPU programming.

The physical differences between CPUs and GPUs inform their strengths and weaknesses while executing instructions. A Central Processing Unit (CPU) is the main processor of a given computer, the job of the processor is to decode and execute instructions from memory. These instructions do tasks such as reading from memory, executing arithmetic and storing values. These are traditionally executed sequentially. However, modern CPU designs contain multiple cores that can execute instructions in parallel. Protocols such as Message Passing Interface (MPI) allow the transfer of data between CPUs, enabling systems like supercomputers to have thousands of CPU cores working in parallel on a common task.[4]

Graphics Processing Units (GPU) are dedicated hardware accelerators originally designed, as the name

suggests, to process and output computer graphics. They achieve this through a tightly integrated array of many programmable processors (see figure 4). The nature of their design, having up to thousands of cores, compared to a CPUs dozens, allows them to perform parallelized tasks much more efficiently. However, they are reliant on the integrated memory as transferring data between system and on-board memory is both slow and instruction heavy. If workloads constantly move data between system memory and VRAM, the GPU spends more time waiting than computing. GPUs are often programmed through APIs such as OpenGL, Vulkan and DirectX3D for graphics and CUDA for a more general C-like programming language.

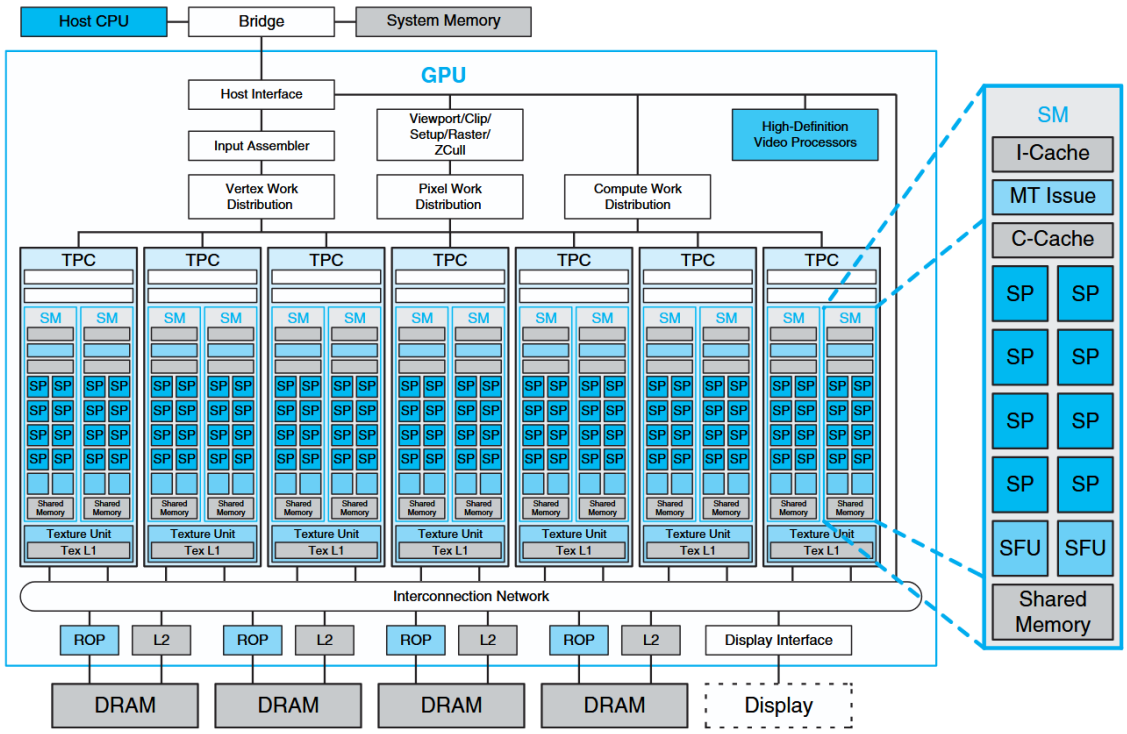

Figure 4: Example of GPU architecture

[14]

LAMMPS was designed in mid-1990s with the philosophy of scalability. "*The goal was to create a parallel MD code which could leverage the spatial parallelism algorithms [...] to effectively use what were then large supercomputers with 100s to a few 1000 processors (cores today) for either materials or biomolecular modeling*"[19] The code was originally built to be executed by CPU-based systems, and only much later was a GPU module added. Consequentially, this means that the structure is completely different from gamdpy, even when both are utilizing GPU-acceleration. LAMMPS primarily uses a hybrid approach where GPU-acceleration is used for the force calculation, $\vec{F}_i = \sum_{j \neq i} \vec{F}_{ij}$, after which

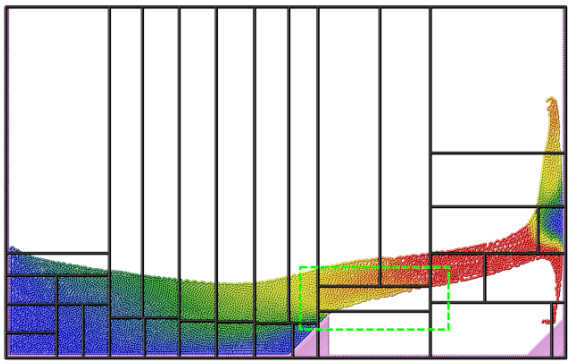

Figure 5: The black lines show the division into subdomains in LAMMPS, and the contained atoms are "owned" by the corresponding CPU thread. [19]

the data is sent back to the CPU for algorithms such as updating and validating the neighbor list. On the other hand, gamdpy is built from the ground up with GPUs in mind meaning all algorithms are executed by the GPU, saving computationally costly transfers. [19]

Let us examine one of the problems that LAMMPS encounters due to its CPU approach to parallelization. Imagine N particles spread across a 2D[1] volume. An easy way to spread the work of the simulation is to partition the volume space into $n \cdot n$ boxes and assigning a thread/processor to each. However, if the distribution is non-uniform, some boxes will contain more particles and the assigned thread will take longer to execute the algorithms. This means that every other thread will have to wait, vastly increasing the elapsed time. On the flip side, some threads will likely have few to no assigned particles in their domain and will be done much quicker. LAMMPS solves this partitioning problem by implementing an algorithm, that partitions the volume into rectangles each containing a roughly similar amount of particles, thus removing the bottleneck. [19]

Programming for GPUs is different than traditional CPU programming. The following section deals with explaining some of the quirks one might encounter.

The basic structure of code being executed in the GPU is the following: A kernel is the lowest level, it is a program or function that is designed to be executed by many threads concurrently. These are then further organized into thread blocks, which run the same kernel and share important resources such as memory. Then there is a grid, which is a set of thread blocks that can execute independently of each other and thus may execute in parallel, see Figure 6. When initializing a program, one decides the amount of threads per blocks and the number of blocks comprising a grid. [14]

[1]The principles used throughout this section are explained in 2-dimensions as it is easier to visualize, but generalize straightforwardly to 3-dimensions as implemented in MD software.

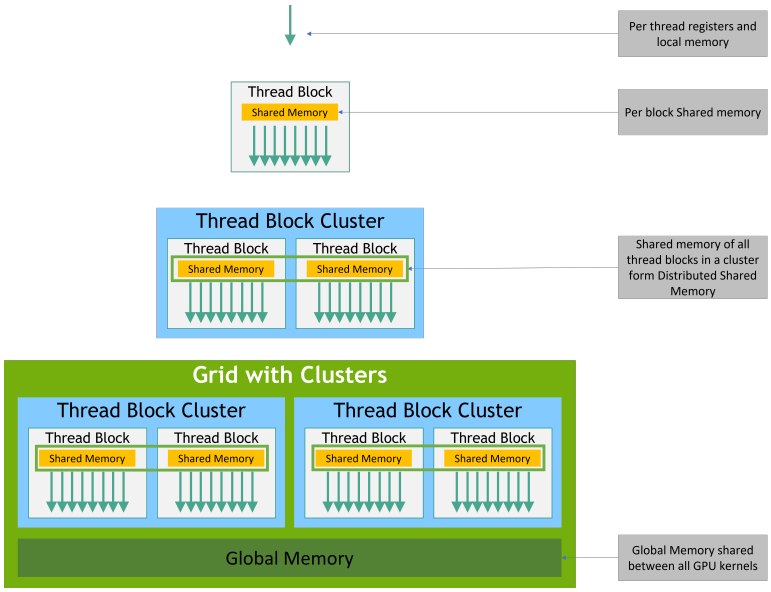

Figure 6: A diagram showing the hierarchy of kernels, thread block, grids and their relation to memory. [1]

An important aspect to consider when working with algorithms being executed on the GPU is the issue of syncing, different threads may be done with their portions of their calculations faster than others, similar to the above domain partitioning problem. Since every thread block grid is working independently, they can finish their kernels at different times, if the results of later threads depend on earlier ones, then the results can be different each time. The solution to this is to synchronize the thread blocks after certain steps are finished. In the case of gamdpy, syncing happens whenever the neighbor list is checked as valid and whenever velocities are updated. There are two primary methods gamdpy uses thread sync and grid sync. Thread sync is the default method that works by waiting at the end of each kernel, e.g. waiting for the neighbor list updates to be finished by every thread. The other one is grid sync, which does syncing at the grid level which is computationally faster, but requires overhead that leads to the method not working at larger system sizes. [1] [2]

Another issue that can plague parallel programs is that of how to handle variables. In order to be able to work parallel multiple threads must be able to work on and refer to certain variables while keeping consistent behavior. CUDA utilizes atomic memory operations, which give memory reads, modify and writes without intervening access at the cost of some computational performance overhead. Gamdpy uses these atomic numbers to utilize Newtons Third law, which states that every Force must have an equal but opposite reaction, thus saving computations under certain circumstances. [19]

[2] CUDA is closed source and there is no way to know how exactly each method works, the drawbacks and strengths of each were determined through testing i.e. using the autotuner.

GPUs exceed at tasks with a lot of kernels, that is, the most trivial is case is one where one can assign one particle to each thread. However, for smaller system sizes there are not enough particles to assign to each thread. The solution done by gamdpy is to assign multiple threads to one particle, that is to let multiple threads handle portions of the calculation. This increases the number of threads used to get more out of the hardware, but it is still underutilized. Gamdpy manages these values primarily in the context of particles, the program when initialized is given a compute plan that it uses to tell the CUDA API how to allocate compute. This compute plan comprises of things like the skin size, which algorithm is used for the neighbor list $O(N^2)11$ or $O(N \cdot N_{nb})$ 8, the number of threads used per particle, the number of particles per block, which syncing method is used and finally if Newtons third law should be used. Most of these values we have already learned about with the last two comprising the incoming paragraphs. In addition, gamdpy has an autotuner functionality that finds the optimal values of the compute plan for different system sizes and graphics card profiles. [2]

To conclude, CPUs and GPUs are structurally different and therefore have different strengths and weaknesses. LAMMPS utilizes certain protocols like MPI to connect with other CPUs and algorithms to divide the domain they work on to maintain computational efficiency. GPUs have a lot of parameters, which the gamdpy autotuner attempts to optimize, such as the particles per thread threads per block, usage of Newtons Third Law, which neighbor list algorithm to use, size of the skin and finally wether to use grid sync or not.

3 Method

3.1 Aim of experiment

To attempt to answer our research questions, we need to evaluate and compare the energy efficiency of CPUs and GPUs when doing identical computations. This should allow identification of which types of workloads benefit the most from GPU and CPU computation. To meaningfully compare the different implementations of MD simulation, we need to carry out an experiment with the following criteria:

- A repeatable MD simulation task which allows configuration of number of particles and simulation timespan. The configuration must be software-agnostic to fairly compare, how the same task is handled by different MD simulation software.
- Measurements of the total power usage of the computer and simulation speed, measured in timesteps per second.[3]
- Collected data stored in a tabulated format. This allows for easy, programmatic data analysis and presentation.

3.2 Tools used

For the experiment, we are using the following tools:

- A computer with an AMD Ryzen 9 7900 CPU and an NVIDIA GeForce RTX 4090 GPU.
- LAMMPS and gamdpy MD simulation software packages.
- An Elcanic ENG100 power meter for measuring the power draw of the entire computer.
- A command line utility, nvidia-smi, for measuring the power draw of the GPU.
- A custom Python program responsible for running multiple sets of simulation benchmarks with user-defined parameters, measuring power draw, and logging obtained data for further analysis.

Configuration of gamdpy does not leave much room for error in terms of performance optimization - only the compute plan parameters can be controlled, which is done automatically by the autotuner. LAMMPS, on the other hand, has a wide range of optional modules, that fundamentally change which type of hardware is responsible for the different calculations, as well as when specific pieces of data is transferred between hardware. In order to avoid a biased result, we have included two modules for LAMMPS, that greatly increase performance: the MPI library, which allows utilization of multiple

[3] A timestep is the unit of time in the simulation, for which the simulation advances. Simulation variables are updated no more than once per timestep. More in section 2.2.

Figure 7: Setup of the computer with the power meter attached.

CPU-threads, increasing leveraged number of threads from 1 to 12, and the GPU package, which allows some calculations to be done on the GPU.[4] For the default LAMMPS runs, we use all 12 CPU-threads. For the GPU-accelerated LAMMPS runs, we manually hand-picked the optimal number of CPU-threads to maximize simulation speed (TPS).

3.3 Data collection

Our Python-based program (source code available - see appendix A.2) is structured to do the aforementioned tasks (see section 3.2) of simulating, measuring, and logging with an output, which is consistent across MD simulation packages or backends. We have implemented the following four backends:

- Gamdpy with its default computation parameters (see section 2.3)
- Gamdpy with autotuned parameters
- LAMMPS with its default CPU-exclusive implementation running on 12 CPU-threads
- LAMMPS with the GPU package running on hand-picked number of CPU-threads

[4]While this CPU has 24 threads, the HPC administrator has put restrictions on available number of threads per task.

Regardless of implementation, the program will run the same benchmark, a Lennard-Jones crystal melting, repeatedly for approximately 15 seconds, then doing nothing for 15 seconds and then doubling the amount of particles (system size) simulated from 2^9 (512) to 2^{21} (2097152). Upon completion of each simulated system size, the simulation speed (timesteps per second) is appended to the data log. Meanwhile, the power usage of the computer is measured and appended to the data log in intervals of 1 second. A complete data log contains timestamp, power draw, simulation system size, and timesteps per second.

3.4 Limitations

We are aware of several limitations of our experiment of varying severity:

- Power usage of compute-relevant hardware not measured exclusively:
  As the power meter measures the total power draw of the computer, the measurements do not account for power usage of the other computer components such as motherboard, memory, drives, or extra cooling, some or all of which may fluctuate depending on internal factors such as scheduled background processes or external factors such as temperature. We attempt to account for this by measuring pauses between each simulations to establish a baseline idle power draw.
- HPC hardware access:
  Due to the implementation of the HPC access restrictions, the hardware resources are shared with other users on that system. While we are trying to avoid this problem by running our experiments at hours with less traffic, as well as checking for other running processes beforehand, the collected data may still be polluted. To account for this, we are running each variation of the experiment three times, which allows us to spot any outliers by comparison.
- Memory limits:
  The GPU, even with 24GB of VRAM (video random-access memory), will run out of memory when running the LJ benchmark with system sizes of 2^{22} particles or more. We would have liked to analyze larger system sizes (up to 2^{23} particles) to better analyze a notable divergence between gamdpy and LAMMPS happening in that range (see figure 14).

4 Analysis

In this section, we analyze the data collected from the experiments described in Section 3. The goal of the analysis is to quantify and interpret the relationship between power consumption, simulation performance, and energy efficiency across different molecular dynamics (MD) backends. We focus on how these metrics scale with system size and what this reveals about the suitability of CPU and GPU based computation for MD simulations. This section will mainly feature data for one of the backends, gamdpy, but identical data processing has been done for the remaining backends in our discussion section (see Section 5).

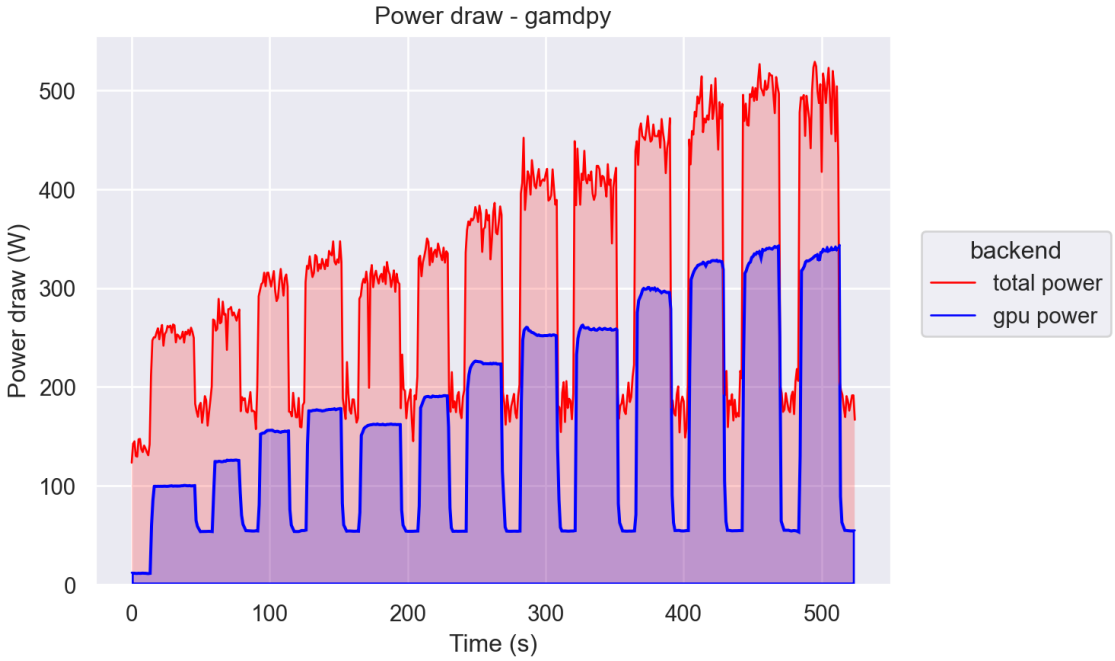

Figure 8: Comparing the total power draw and that of the GPU over time.

Figure 8 shows the raw power measurements recorded during the gamdpy benchmark runs. The red curve represents the total system power draw, while the blue curve shows GPU-specific power consumption. A clear stepwise pattern is observed, corresponding to successive increases in system size during the benchmark. As the number of particles doubles, both GPU and total power draw increase, indicating higher computational load.

Despite the limitations discussed in Section 3.4 most notably that total system power includes non compute components such as cooling and memory the strong correlation between GPU power and total power suggests that the GPU dominates overall energy consumption during simulation. The gradual

upward trend in peak power values reflects increasing GPU utilization as larger particle systems provide more parallel work, allowing the GPU to operate closer to its intended performance envelope.

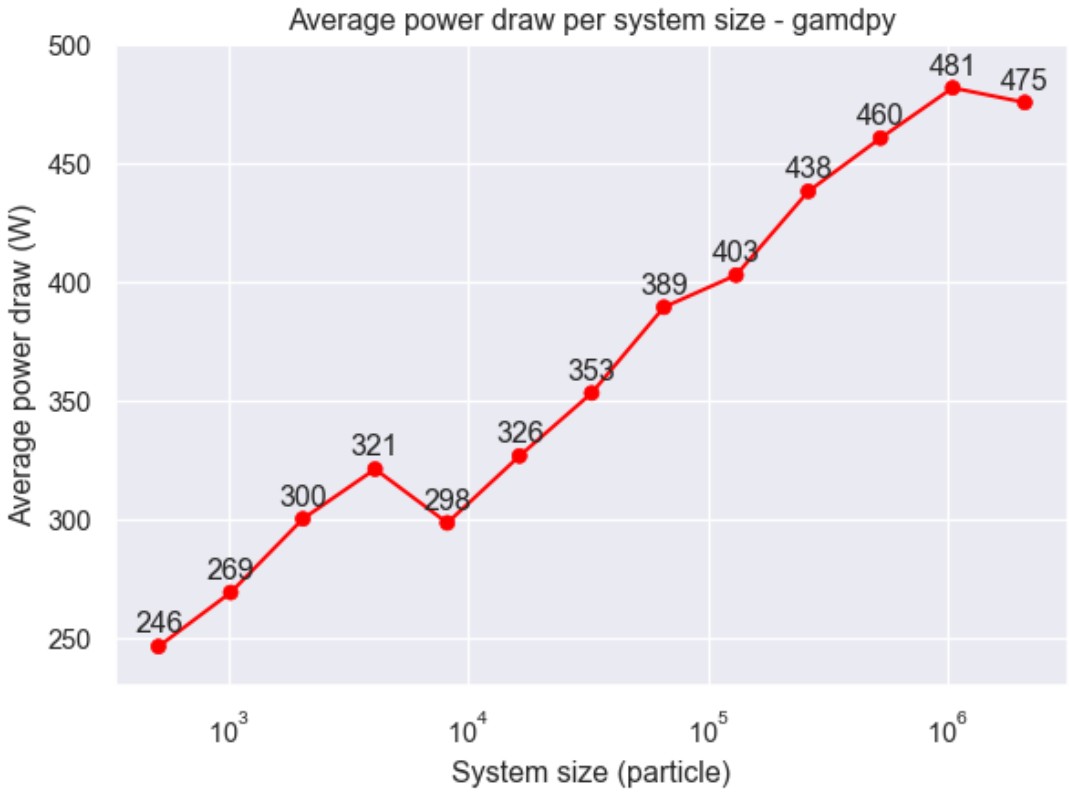

Figure 9: Average power draw by system size.

Calculating the averages of each of those peaks and plotting them with the corresponding system size yields the plot seen in figure 9. This does not show the whole picture, as mentioned in Section 3, the benchmark is running for a fixed amount of time.

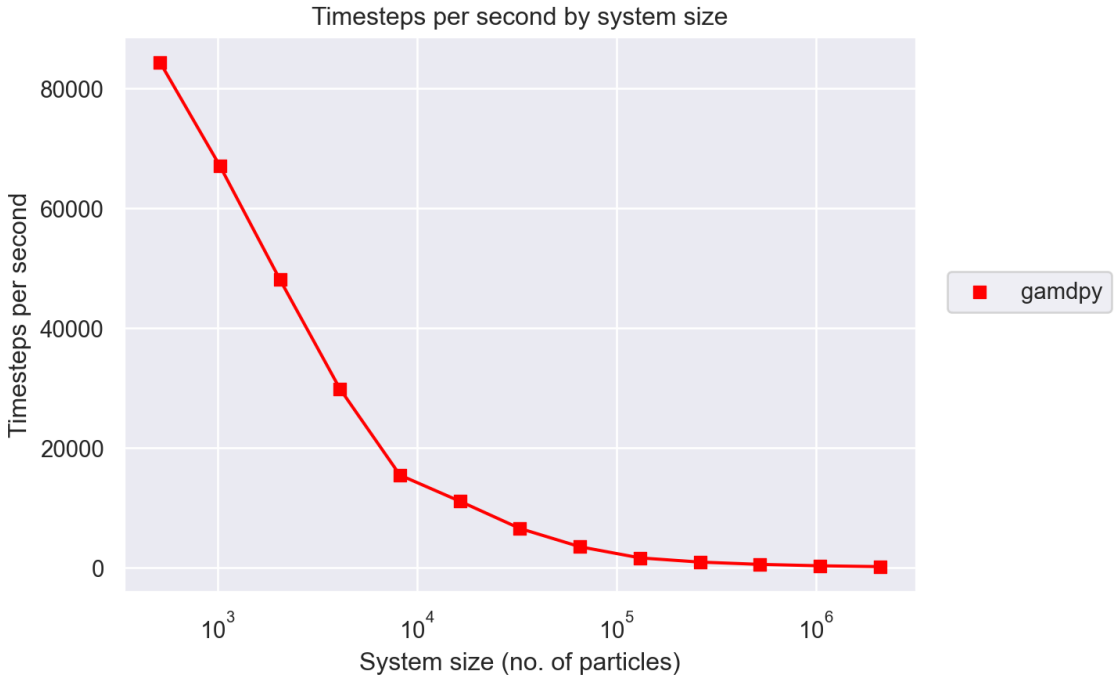

Figure 10: Timesteps per second by system size

In order to quantify the amount of useful work done by the processor we will use the timesteps per second (TPS) metric. The plot in figure 10 uses the same dataset of a gamdpy benchmark run, but compares TPS to system sizes.

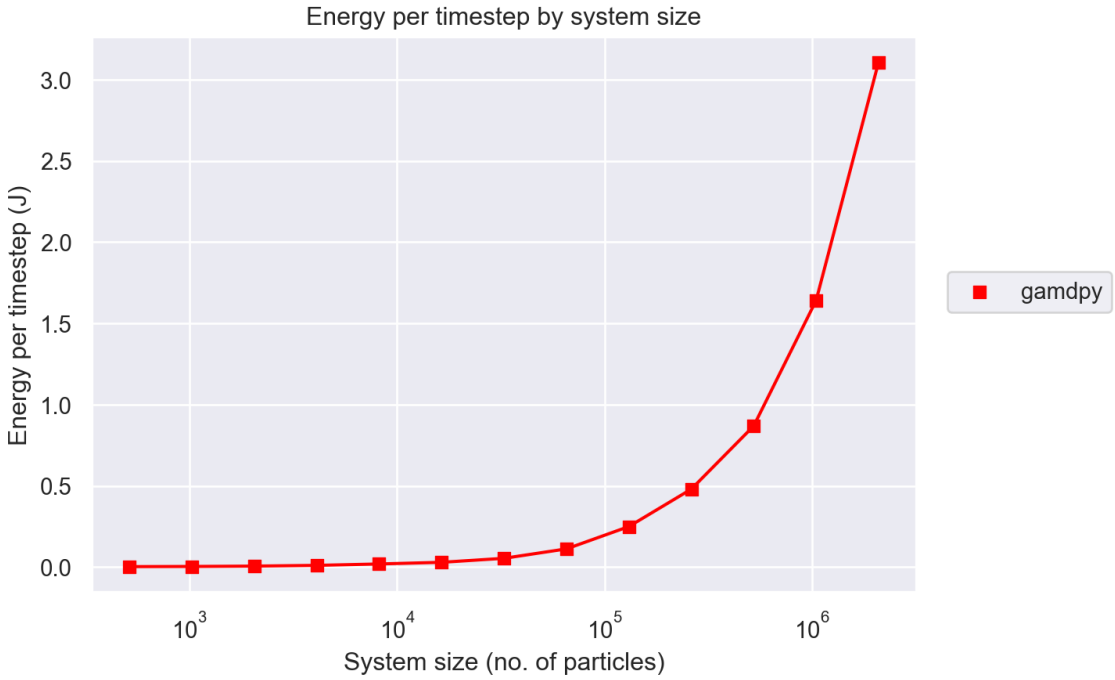

Figure 11: Energy efficiency by system size.

Dividing the power draw by the TPS yields an energy per timestep metric, measured in J. Plotting energy per timestep as a function of system size yields the plot seen in figure 11.

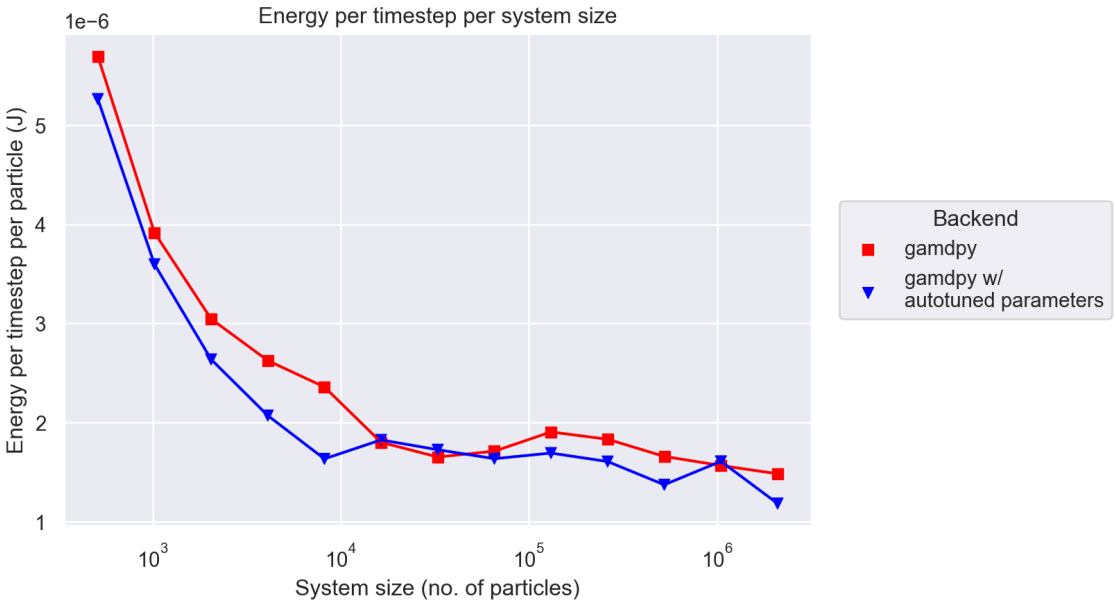

Figure 12: Comparing the energy efficiency per particle by system size of gamdpy with and without autotuned parameters.

While the previous plot is good, a lot of the increases seen can be attributed to increasing the number of particles the MD package is working with, to further isolate the effects on energy efficiency of changing the system size the Energy per timestep column can be divided with the number of particles, the result of plotting Energy per timestep per particle is shown in figure 12.

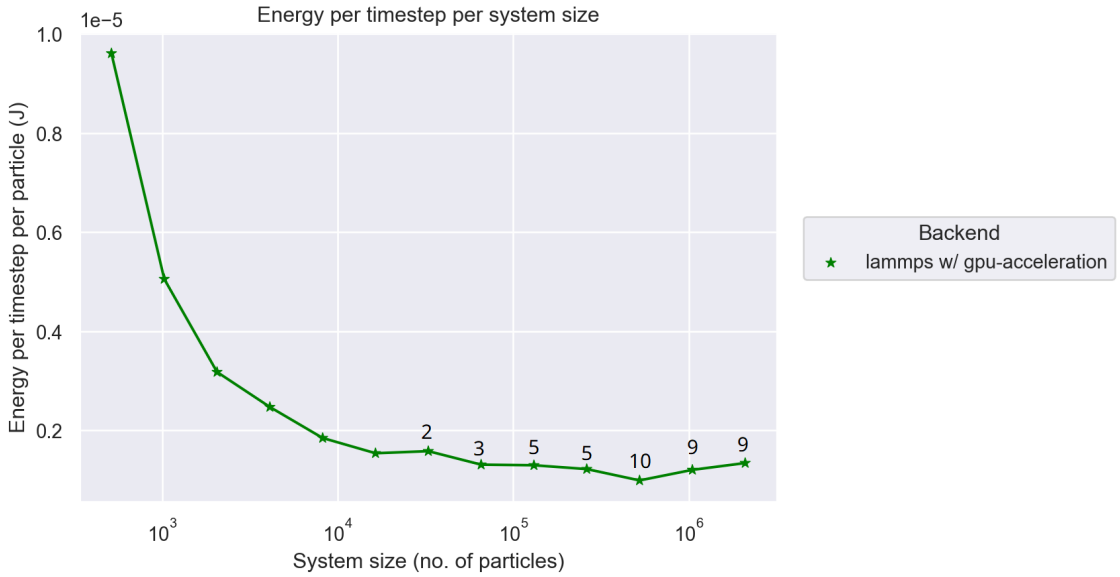

Figure 13: Results for GPU-accelerated LAMMPS with number of CPU-threads assigned (non-numbered means 1 CPU-thread).

For the corresponding plot for GPU-accelerated LAMMPS as the backend, we specified the number of CPU-threads assigned to work alongside the GPU, as mentioned in Section 3. We determined the optimal number of CPU-threads for each system size to maximize performance, as performance decreased greatly when using too many or too few CPU-threads. Figure 13 shows the exact amount of CPU-threads we used per system size - keeping them consistent for every single GPU-accelerated LAMMPS benchmark run. All subsequent plots of GPU-accelerated LAMMPS implies these CPU-thread values.

5 Discussion

At the beginning of the project we set out to find answers for our research questions. This section attempts to interpret the results found in section 4 with the help of the knowledge learned in section 2. Lastly this section will attempt to tie the results of our research to the semester theme, *Applications of Science and Technology*.

5.1 In the context of molecular dynamics simulations, which is more energy efficient, GPU or CPU computation?

Repeating the process done in Section 4 for figure 11 and stacking the results we get figure 14.

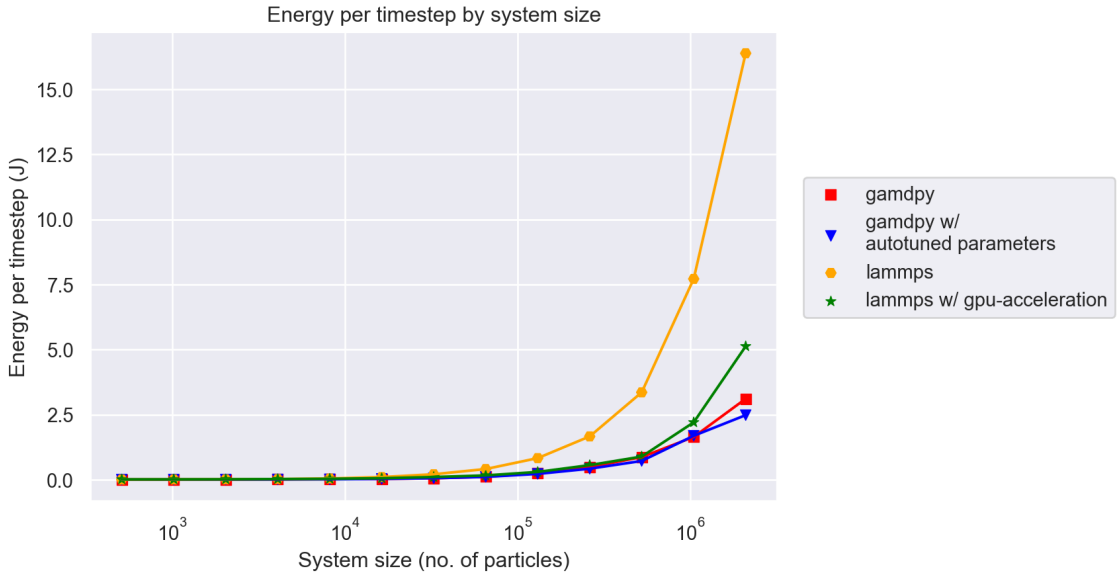

Figure 14: Plot showing the energy per timestep for the four different benchmarks

From this we can see that GPU-based runs (gamdpy without autotuner, gamdpy with autotuner and GPU-accelerated LAMMPS) used significantly less energy per timestep relative to default LAMMPS running on the CPU at large system sizes. Gamdpy is approximately 3 times more energy efficient at the second largest system size tested $N = 1048576$ and 6 times at the highest system tested $N = 2097152$. GPU-based simulations pull increasingly ahead with system size increases as the GPU approaches full utilization.

Using a graph of energy per timestep per particle like 12 and stacking the plots, we get the following plot:

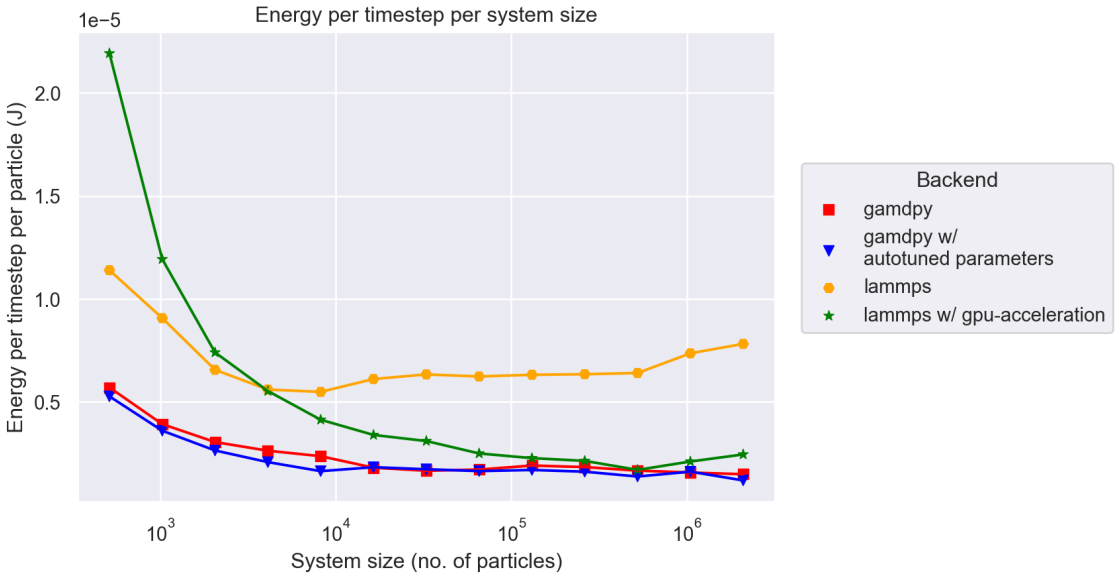

Figure 15: Plot showing the energy per timestep per particle for the four different benchmarks

This plot is useful as it removes the increase from the number of particles. The ideal plot should show a constant low y-value resulting in a completely horizontal line. This would indicate consistent processor utilization across all system sizes and minimal overall energy consumption.

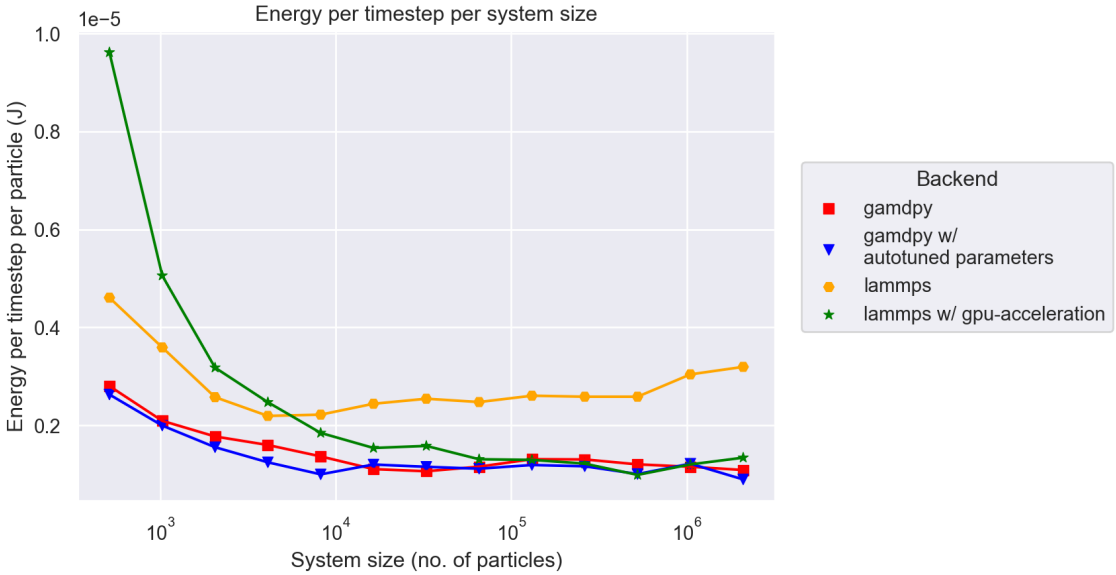

Figure 16: The same data as figure 15, but subtracting the average idle power draw of 125 W.

In figure 16, we subtract the idle power draw of the computer to isolate the power used by the simulation

software exclusively.

We can see that all of the software tested has issues with scaling, as they get increasingly less efficient at smaller system sizes. The amount of energy used per timestep per particle increases at smaller system sizes, default and autotuned gamdpy being affected the least, only getting half the energy per timestep per particle compared to larger system sizes. Default LAMMPS, on the other hand, shows near-constant performance independent of system size albeit with a much higher baseline. All in all GPU implementations are up to 4 times more energy efficient than a CPU approach.

GPU-accelerated LAMMPS suffers a 4 times increase in energy used per timestep per particle at smaller system sizes. This is likely due to the amount of transfers between system memory and GPU memory,that LAMMPS relies on, as mentioned in Section 2.

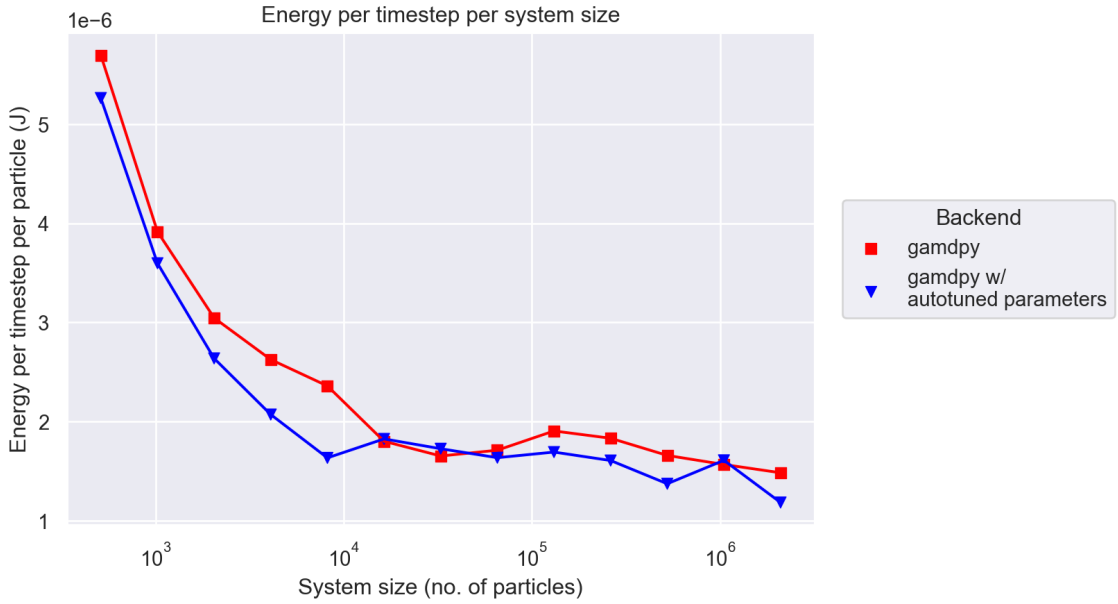

Figure 17: Plot showing the difference for gamdpy with and without autotuner, for energy per timestep per particle

Another takeaway is the difference between default and autotuned gamdpy as seen in 17. The autotuned runs consistently used equal or less power compared to the default runs. This highlights the importance of optimizing all the parameters as discussed in Section 2. It is worth noting that the default gamdpy values were chosen with this class of hardware in mind, using different hardware likely would increase the usefulness of the autotuner.

In short: GPU computation is up to 2 times more efficient for molecular dynamics compared to CPU computation.

5.2 How does number of molecules affect energy efficiency?

From graphs such as 8 we can see an apparent trend of increasing system sizes effecting increases in power usage up until a threshold caused by the GPU being fully utilized (or at least as close to full utilization as possible for the implementation of MD simulation). However, there are certain deviations from this trend - notably between $N = 4096$ and $N = 8192$ as well as $N = 65536$ and $N = 131072$, which corresponds to changes in the parameters, neighbor list method ($O(N^2)$ (see 8) or $O(N \cdot N_{nb})$ (see 11)) and gridsync usage (on or off) respectively.

In addition, we found that the performance gap between GPU-accelerated LAMMPS and gamdpy at low system sizes can be attributed to parameter optimization. The gamdpy autotuner chooses the $O(N^2)$8 time-complexity algorithm for the neighbor list at low system sizes, suggesting it to be more time efficient parameter. This confirms what we read on our original research. [2]

In short: The energy efficiency for GPU-based backends depends heavily on number of particles, with energy per timestep per particle reducing substantially at lower number of particles, while having little effect on the efficiency of CPU-based LAMMPS.

5.3 What are the limitations of computing on a GPU?

GPU computing has a lot of caveats and requires a higher level of expertise to properly utilize. As we learned in section 2.3, GPUs can be difficult to work with due to their structure requiring certain abstractions, that is designing algorithms that can properly parallelize. Even in workloads that are seemingly perfect fits like Molecular Dynamics Simulations, the GPU excels at doing force calculations and maintenance of the neighbor list with the $O(N^2)$8 algorithm, but struggles more in tasks that require higher amounts of synchronization, such as the integrator7 or the linked-list neighbor list11.

Another problem with GPUs is that it is much more difficult to optimize for them. As we learned, initializing gamdpy requires the following parameters: particles per block (pb), threads per block (tp), skin, grid sync, neighbor list algorithm and whether Newtons Third Law is used. As discussed gamdpy uses the autotuner to find good values for these parameters, but this can be a bothersome process that has its own energy and development time cost attached to it.

More broadly the strengths of GPUs lie in mass parallelized tasks that require high amounts of number crunching. This explains their recent popularity boom. CPU computing is still overall better for general computing

In short: GPU programming is harder than standard CPU programming, they require tasks that are well fitted to the types of computation that the GPU is strong at.

5.4 On the semester theme

Section 2 greatly aided our understanding of scientific computing, we were able to be learn and understand how Molecular Dynamics packages are created from the ground up and how they can be used to do science. In addition, we were able to get involved with benchmarking two different approaches to the same problem, finding that the environmental impact of this type of scientific research depends on the hardware used for the simulation. We learned that while GPUs use more power, they finish the assigned tasks faster, making them 'greener'. This is directly related to our semester theme on the technology's role on science and society.

We believe that improving on how long it takes to run MD simulations, to the degree that GPUs do compared to CPUs, will help get results for MD simulations faster, while saving on energy. This will mean faster progress for physics and material science, while sparing the planet from needless energy usage.

6 Conclusion

We have found that for MD simulations at least, the potential energy saved by using GPU computation instead of CPU computation, was consistently about 2 times, at and above system sizes of 8192.

We found that LAMMPS with GPU acceleration does poorly at small system sizes, due to back and forth communication between GPU and CPU, it performed as well as pure GPU architecture at larger system sizes.

We found that GPU programming presents a steeper learning curve and is less suitable for workloads that require extensive synchronization, GPU-based computation offers substantially higher performance and greater energy efficiency for tasks that align well with its architecture.

We believe that this improved energy efficiency from CPU to GPU computation, can help make MD simulations faster, and thus, greener as a field of study.

7 Bibliography

References

[1] CUDA C++ programming guide, Oct. 2025. URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[2] N. Bailey, T. Ingebrigtsen, J. S. Hansen, A. Veldhorst, L. Bøhling, C. Lemarchand, A. Olsen, A. Bacher, L. Costigliola, U. Pedersen, H. Larsen, J. Dyre, and T. Schrøder. RUMD: A general purpose molecular dynamics package optimized to utilize GPU hardware down to a few thousand particles. *SciPost Physics*, 3(6):038, Dec. 2017. ISSN 2542-4653. doi: 10.21468/SciPostPhys.3.6.038. URL https://scipost.org/10.21468/SciPostPhys.3.6.038.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to algorithms. In *Introduction to algorithms*, pages 236–237. The MIT Press, Cambridge, Massachusetts London, fourth edition edition, 2022. ISBN 978-0-262-04630-5 978-0-262-36750-9.

[4] Ian Wienand. Computer Science from the Bottom Up. URL https://www.bottomupcs.com/.

[5] M. Kang and M. Park. Power Estimation and Energy Efficiency of AI Accelerators on Embedded Systems. *Energies*, 18(14):3840, July 2025. ISSN 1996-1073. doi: 10.3390/en18143840. URL https://www.mdpi.com/1996-1073/18/14/3840.

[6] T. Kusaba, Y. Awaki, K. Yoshida, S. Miwa, H. Yamaki, T. Hanawa, and H. Honda. Power-Efficiency Variation on A64FX Supercomputers and its Application to System Operation. In *2024 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*, pages 55–65, Kobe, Japan, Sept. 2024. IEEE. ISBN 979-8-3503-8345-4. doi: 10.1109/CLUSTERWorkshops61563.2024.00018. URL https://ieeexplore.ieee.org/document/10740900/.

[7] LANOC.ORG. AMD Ryzen 9 7900X and 7950X, Sept. 2022. URL https://lanoc.org/review/cpus/amd-ryzen-9-7900x-and-7950x?showall=1.

[8] Lee-Ping Wang. Introduction to classical molecular dynamics, Mar. 2014. URL https://youtu.be/_TiQYNWJwYg?si=7lrMxeZq1SVR_U29.

[9] R. R. L. Machado, J. Schmitt, S. Eibl, J. Eitzinger, R. Leißa, S. Hack, A. Pérard-Gayot, R. Membarth, and H. Köstler. tinyMD: A Portable and Scalable Implementation for Pairwise Interactions Simulations, 2020. URL https://arxiv.org/abs/2009.07400. Version Number: 1.

[10] Michael P. Allen. Computational soft matter: from synthetic polymers to proteins. 2: Lecture notes. Number volume 23 in NIC series, pages 1–28. NIC, Jülich, 2004. ISBN 978-3-00-012641-3. Num Pages: 1.

[11] Nick Bailey, Thomas Schrøder, Ulf R. Pedersen, and Lorenzo Costigliola. gamdpy.

[12] Nick Corn. Tutorial: CUDA programming in Python with numba and cupy, July 2021. URL https://youtu.be/9bBsvpg-Xlk?si=L-Adp4WmsAM9xhPd.

[13] NVIDIA. NVIDIA ADA GPU ARCHITECTURE. Technical Report V2.02, NVIDIA. URL https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf.

[14] D. A. Patterson and J. L. Hennessy. *Computer organization and design: the hardware/software interface*. Elsevier Morgan Kaufmann, Amsterdam Heidelberg, 4. ed. edition, 2010. ISBN 978-0-12-374493-7.

[15] R. Prajwal, S. J. Pawan, S. Nazarian, N. Heller, C. J. Weight, V. Duddalwar, and C.-C. J. Kuo. A Study on Energy Consumption in AI-Driven Medical Image Segmentation. *Journal of Imaging*, 11(6):174, May 2025. ISSN 2313-433X. doi: 10.3390/jimaging11060174. URL https://www.mdpi.com/2313-433X/11/6/174.

[16] T. B. Schrøder and J. C. Dyre. Solid-like mean-square displacement in glass-forming liquids. *The Journal of Chemical Physics*, 152(14):141101, Apr. 2020. ISSN 0021-9606, 1089-7690. doi: 10.1063/5.0004093. URL https://pubs.aip.org/jcp/article/152/14/141101/197909/Solid-like-mean-square-displacement-in-glass.

[17] V. Stegailov, G. Smirnov, and V. Vecher. VASP hits the memory wall: Processors efficiency comparison. *Concurrency and Computation: Practice and Experience*, 31(19):e5136, Oct. 2019. ISSN 1532-0626, 1532-0634. doi: 10.1002/cpe.5136. URL https://onlinelibrary.wiley.com/doi/10.1002/cpe.5136.

[18] A. Sánchez-Mompó, I. Mavromatis, P. Li, K. Katsaros, and A. Khan. Green MLOps to Green GenOps: An Empirical Study of Energy Consumption in Discriminative and Generative AI Operations. *Information*, 16(4):281, Mar. 2025. ISSN 2078-2489. doi: 10.3390/info16040281. URL https://www.mdpi.com/2078-2489/16/4/281.

[19] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton. LAMMPS - a flexible simulation tool for particle-based materials

modeling at the atomic, meso, and continuum scales. *Computer Physics Communications*, 271, Feb. 2022. doi: 10.1016/j.cpc.2021.108171. URL https://www.sciencedirect.com/science/article/pii/S0010465521002836.

[20] V. Vecher, V. Nikolskii, and V. Stegailov. GPU-Accelerated Molecular Dynamics: Energy Consumption and Performance. In V. Voevodin and S. Sobolev, editors, *Supercomputing*, pages 78–90, Cham, 2016. Springer International Publishing. ISBN 978-3-319-55669-7.

[21] Z. Yang, K. Adamek, and W. Armour. Part-time Power Measurements: nvidia-smi's Lack of Attention. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–17, Nov. 2024. doi: 10.1109/SC41406.2024.00028. URL http://arxiv.org/abs/2312.02741. arXiv:2312.02741 [cs].

A Appendix

A.1 AI declaration

In this project, we used Copilot during the preparation of this project. The output from these prompts or tools was used to help us get familiarized with python modules like pandas, seaborn, matplotlib, etc. Which we used to assist us during the process of writing scripts for data collecting and data processing. We chose to use these tools because LLMs like GPT-5 are great tools for assisting with coding, particularly in asynchronous environments that we were not familiar with. Only around 15% of our code was AI assisted, as we wanted to be able to understand the codebase and ensure correctness. In addition, it was used to assist in writing by helping to untangle wordy and complicated paragraphs. Less than 5% of the report has been written with the help of AI.

A.2 Source code

The source code for our Python-based program is hosted on GitHub:
https://github.com/seranovic/semester1/tree/73d022ba1586e898fc70516bbbadffdb0d15e6b8

The `main.py` script and `bp1/` Python module contains the main program used for data collection.

The `data/` directory contains the raw output used for all data analysis.

The `bin/` directory contains the custom-built LAMMPS executable with the GPU-package enabled.