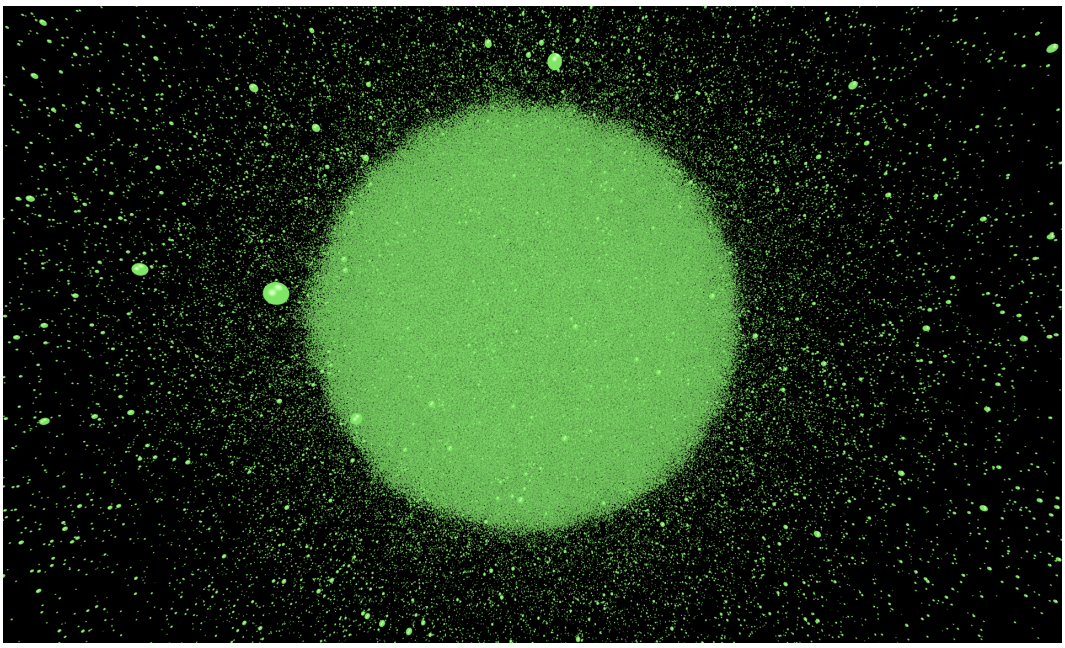
Introduction to GPU and accelerator programming for scientific computing
Fast prefix sum on the GPU, for molecular dynamics

Daniel Hedman[1, *]

[1] *Applied Physics, Division of Materials Science, Department of Engineering Sciences and Mathematics, Luleå University of Technology, SE-971 87 Luleå, Sweden*

(Dated: January 29, 2016)



Keywords: GPU, CUDA, PyCUMD

I. INTRODUCTION

In order to create a high performance molecular dynamics (or other multibody simulation) code on the GPU it's key that the particles are sorted to be inline in memory (for coalesced reads of neighboring particle coordinates from memory). This can be achieved by first dividing up the simulation volume into bins and then filling these bins with the particles. In order to achieve coalesced reads these particle coordinates needs to be sorted so that the particles that are in the same bin are also close in memory.

This sorting can be done by a simple counting sort algorithm[1], a key part of this algorithm is preforming a prefix sum[2] over the array containing the particle counts (the number of particles in each bin). The focus of this project is to create and optimize a prefix sum algorithm on the GPU using the CUDA programing language and the apply this to a molecular dynamics code written for the GPU using CUDA and Python.

II. METHODS

For the GPU programing part of this project the CUDA programing language was used. On the host side to interfacing with the GPU, Python together with the PyCUDA module was used. For development and testing of performance two different systems were available, these two systems consists of a custom desktop computer, with specifications as in Table I running Windows 10. The other system is one GPU compute node on the Tegner cluster running Linux, specifications for this systems can be found in Table II.

Table I: System 1, a custom desktop computer.

Type	Id	Core clock (MHz)	Mem. clock (MHz)
CPU	Intel i7-4770K	4000	2400
GPU	Nvidia GTX 760	1124	3005

Table II: System 2, one GPU node on the Tegner cluster.

Type	Id	Core clock (MHz)	Mem. clock (MHz)
CPU	2x Intel E5-2690 v3	3100	?
GPU	Nvidia Tesla K80	875	2500

As reference for the GPU prefix sum the Python `numpy.cumsum` function was used, this is a highly optimized work-efficient (work complexity, $\mathcal{O}(n)$) prefix sum running on the CPU. The finalized GPU prefix sum will the be used in creating a fast GPU lennard jones molecular dynamics (MD) program.

This new CUDA/Python (PyCUMD) MD program will be based on an older OpenCL/Python (PyOCLMD) MD implementation. Improvements will be made to the new PyCUMD program (such as periodic boundary conditions) so that the accuracy of the PyCUMD program can be verified against the popular and highly optimized open source MD program LAMMPS[3].

For performance testing and verification of PyCUMD against LAMMPS a microcanonical (NVE) Argon (Ar) atom system was chosen, the simulations of the Ar system where performed using System 2 (see Table II). This system consists of an Argon sphere with a diameter of 66.2256 nm (1047293 atoms) contained in a periodic box of size $(105.120 \times 105.120 \times 105.120)$ nm^3 (5359375 bins). The Argon sphere initially has an simple cubic structure with a lattice spacing of 5.256 Å making it usable and hence collapsing in on itself, other simulation parameters is found in Table III.

Table III: Simulation parameters for the Argon system.

Parameter	Symbol	Value
Epsilon	ϵ	0.0104 eV
Sigma	σ	3.40 Å
Mass	m	39.948 u
Force cutoff	r_{cf}	12 Å
Timestep	dt	2 fs

III. RESULTS AND DISCUSSION

As stated in Section II the CPU prefix sum used as reference has a work complexity of $\mathcal{O}(n)$, ideally the GPU prefix sum should also have a work complexity of $\mathcal{O}(n)$. However the first implementation of the GPU prefix sum was a naive implementation with a work complexity of $\mathcal{O}(n^2)$ (not shown here).

The performance of this naive implementation was on par with the performance of the CPU version for large arrays (because of the poor scaling of GPU version), a better approach was needed. A decision was made to implement a work efficient prefix sum on the GPU using balanced trees as described in[4], the code for the final implementation of the balanced trees GPU prefix sum CUDA kernel is shown below.

```
inline __device__ unsigned int ceilInt(unsigned int x, unsigned int y)
{
        return (x + y - 1u)y;
}

__global__ void prefixSum(unsigned int input, unsigned int output, unsigned int blockSum,
                unsigned int arraySize)
{
        unsigned int idx = threadIdx.x;
        unsigned int dix = blockDim.x;
        unsigned int idg = 2ublockIdx.xdix + idx;

        const unsigned int nrMB = 32u;
        unsigned int idxDouble = 2uidx;
        unsigned int idgL, idgH, idxL, idxH;
        unsigned int blockOffsetL, blockOffsetH;

        __shared__ unsigned int buffer[1024u + 2unrMB];

        idgL = idg;
        idgH = idg + dix;

        idxL = idx;
        idxH = idx + dix;

        blockOffsetL = idxLnrMB;
        blockOffsetH = idxHnrMB;

        if (idgL  arraySize)
        {
                buffer[idxL + blockOffsetL] = input[idgL];

                if (idgH  arraySize)
                {
                        buffer[idxH + blockOffsetH] = input[idgH];
                }
        }
        __syncthreads();

        for (unsigned int offset = 1u; offset  2udix; offset = 2u)
        {
                idxH = offset(idxDouble + 2u) - 1u;

                if (idxH  2udix)
                {
                        idxL = offset(idxDouble + 1u) - 1u;

                        idxH += idxHnrMB;
                        idxL += idxLnrMB;

                        buffer[idxH] += buffer[idxL];
                }
                __syncthreads();
        }

        for (unsigned int offset = powf(2.0f, ceilf(log2f(ceilInt(dix, 2u))));
                        offset = 1u;
                        offset = 2u)
        {
```

```
                idxH = offset(idxDouble + 3u) - 1u;

                if (idxH  2udix)
                {
                        idxL = offset(idxDouble + 2u) - 1u;

                        idxH += idxHnrMB;
                        idxL += idxLnrMB;

                        buffer[idxH] += buffer[idxL];
                }
                __syncthreads();
        }

        idxL = idx;
        idxH = idx + dix;

        if (idgL  arraySize)
        {
                output[idgL] = buffer[idxL + blockOffsetL];

                if (idgH  arraySize)
                {
                        output[idgH] = buffer[idxH + blockOffsetH];
                }
        }

        if (idx == dix - 1u && blockIdx.x  gridDim.x - 1u)
        {
                blockSum[blockIdx.x] = buffer[idxH + blockOffsetH];
        }
        else if (idgH == arraySize - 1u)
        {
                blockSum[blockIdx.x] = buffer[idxH + blockOffsetH];
        }
        else if (idgL == arraySize - 1u)
        {
                blockSum[blockIdx.x] = buffer[idxL + blockOffsetL];
        }
}
```

For curiosity an alternative GPU prefix sum was implemented, this one is build around a double buffer concept. Although much easier to implement this algorithm is not work efficient but has a work complexity of $\mathcal{O}(n \log(n))$ according to[4], which should make it slower for large arrays. The code for the final implementation of the double buffered GPU prefix sum CUDA kernel is shown below.

```
__global__ void prefixSum(unsigned int *input, unsigned int *output, unsigned int *blockSum,
                unsigned int arraySize)
{
        unsigned int idx = threadIdx.x;
        unsigned int dix = blockDim.x;
        unsigned int idg = blockIdx.x*dix + idx;

        if (idg < arraySize)
        {
                unsigned int in = 1u, out = 0u;
                __shared__ unsigned int buffer[2u][512u];

                buffer[out][idx] = input[idg];
                __syncthreads();

                for (unsigned int offset = 1u; offset < dix; offset *= 2u)
                {
                        out = 1u - out;
                        in = 1u - out;

                        if (idx >= offset)
                                buffer[out][idx] = buffer[in][idx] + buffer[in][idx - offset];
                        else
                                buffer[out][idx] = buffer[in][idx];
```

```
                __syncthreads();
        }

        output[idg] = buffer[out][idx];

        if (idx == dix - 1u || idg == arraySize - 1u)
        {
                blockSum[blockIdx.x] = buffer[out][idx];
        }
    }
}
```

For both the balanced trees and the double buffered GPU prefix sum shared memory was used in order to improve performance, however it was found that in early implementations of the balanced trees algorithm the performance was poor owing to shared memory bank conflicts[4]. This was (partly) solved by adding an offset to the indexing of the shared memory buffer `blockOffsetL, blockOffsetH` in the code of size 32.

Both of the GPU prefix sum algorithms work on segments of the input array (one segment for each block) of size `2*threadIdx.x` for the balanced trees and size `threadIdx.x` for the double buffered algorithm. This leads to the problem that the input array will be summed in parts, this is solved by having an extra output array `blockSum` which contains the last element of each partial prefix sum. This `blockSum` array is in turn prefix summed until it's size is smaller then `2*threadIdx.x` or `threadIdx.x`. The prefix sum of the last `blockSum` array will then be added to back to the prefix sum of the previous `blockSum` array and so forth until all of the `blockSum` arrays have been added back up to the original partly prefix summed array completing the prefix sum.

Performance tests of both GPU prefix sum algorithms were done using System 1 (see Table I) and their performance was compared to the `numpy.cumsum` CPU prefix sum. A span of array sizes was used from 2^9 to 2^{27} elements, the performance is measured in millions of elements proceed per second (Me/s).

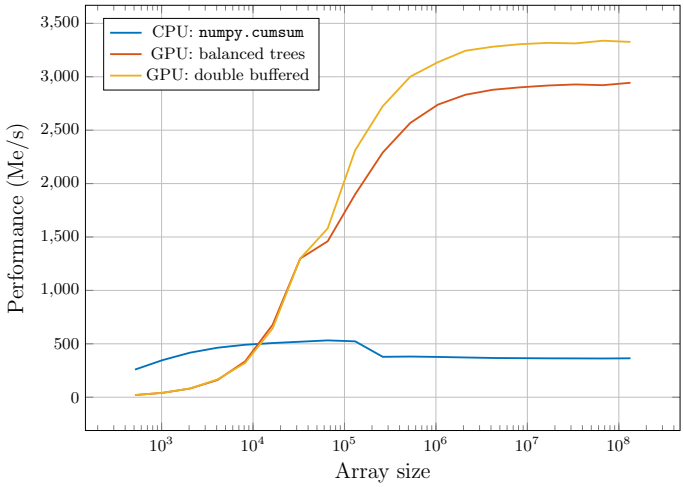
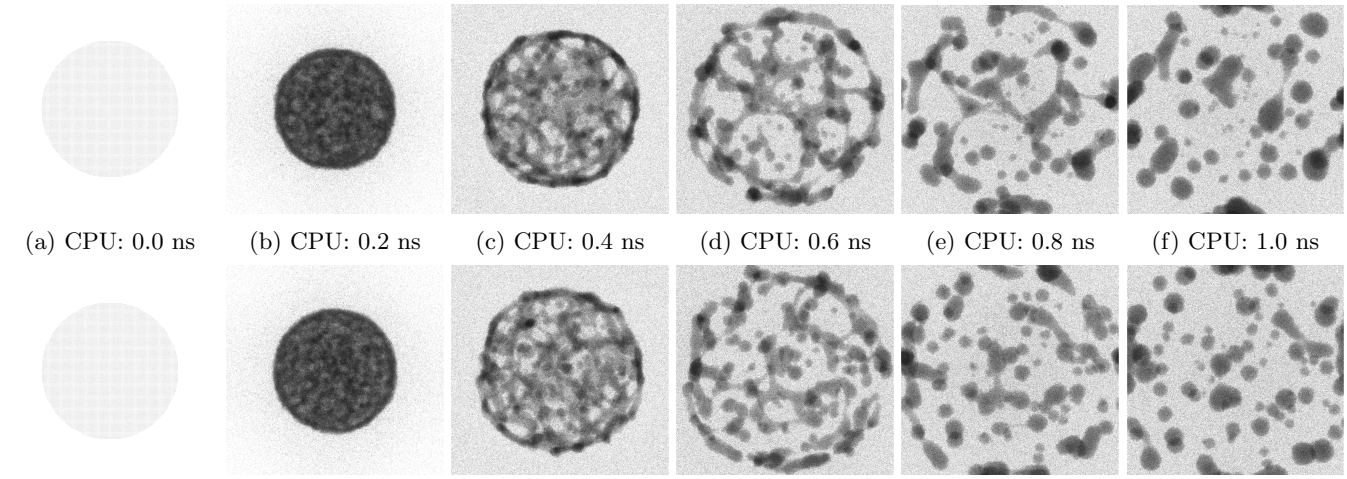


Figure 1: Performance comparison between the balanced trees and double buffered GPU prefix sum algorithms, also included as reference is the `numpy.cumsum` CPU prefix sum.

As seen in Figure 1 the CPU prefix sum is faster for array sizes smaller then $\sim 10^4$ elements, this is attribute to the extra overhead that is created by calling the GPU prefix sum algorithms via PyCUDA (performing several kernel calls and executing Python code). Up to an array size of $\sim 10^5$ elements the difference in performance between the two GPU prefix sum algorithms are negligible, this also points to other performance blockers. For array sizes larger then $\sim 10^5$ elements the double buffered algorithm clearly outperforms the balanced trees one by about 14%.

Form Figure 1 it's clear that the double buffered prefix sum algorithm is the winner when it comes to performance. The next step is to apply this algorithm to a real world application, molecular dynamics as described in Section II. The code for both the PyCUMD program and the two prefix sum algorithms are available on GitHub.

Figure 2 shows a couple of snapshots from the Ar simulations described in Section II, Figure 2a-f corresponds to the CPU simulation using LAMMPS and Figure 2g-l are the corresponding frames from the GPU simulation using PyCUMD. From Figure 2 it's clear that the two simulations show similar behaviors, although an exact comparison (e.g the position of the Ar droplets) can not be made because of the chaotic nature of a multibody system.

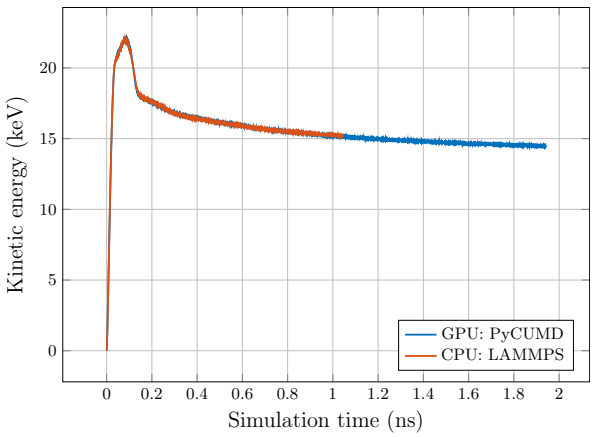


(a) CPU: 0.0 ns (b) CPU: 0.2 ns (c) CPU: 0.4 ns (d) CPU: 0.6 ns (e) CPU: 0.8 ns (f) CPU: 1.0 ns

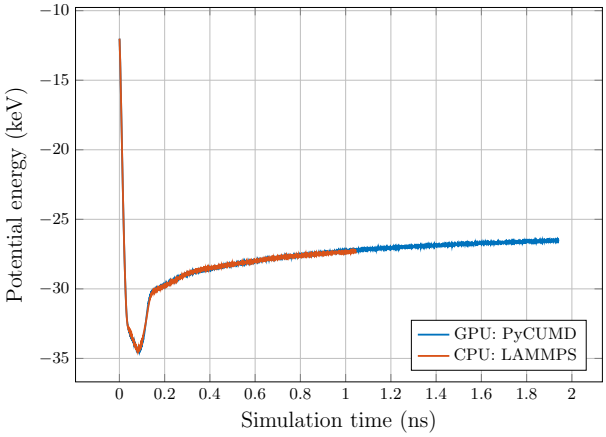
(g) GPU: 0.0 ns (h) GPU: 0.2 ns (i) GPU: 0.4 ns (j) GPU: 0.6 ns (k) GPU: 0.8 ns (l) GPU: 1.0 ns

Figure 2: Snapshots of the atom positions during the simulation, a-f for the CPU simulation and g-l for the GPU simulation. Here one can see the general evolution of the system, starting as an unstable Ar sphere that collapses in on itself creating a Ar gas with suspended liquid Ar droplets. Video available at https://youtu.be/_Tioow9XWPk

A more appropriate test of the accuracy for a molecular dynamics code is to look at properties of the system as a whole, one can for example look at the systems total kinetic and potential energy. Figure 3 shows the systems total kinetic and potential energy plotted over time, here it's clear that the PyCUMD program gives the same results as LAMMPS.



(a) The total kinetic energy of the system.



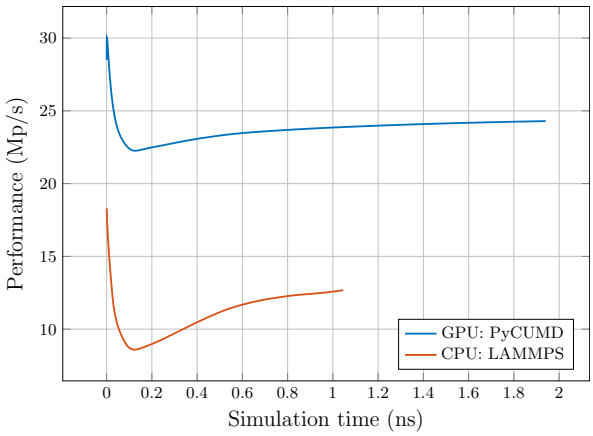
(b) The total potential energy of the system.

Figure 3: Characteristic properties of the Ar system, the total kinetic energy and the total potential energy of the system.

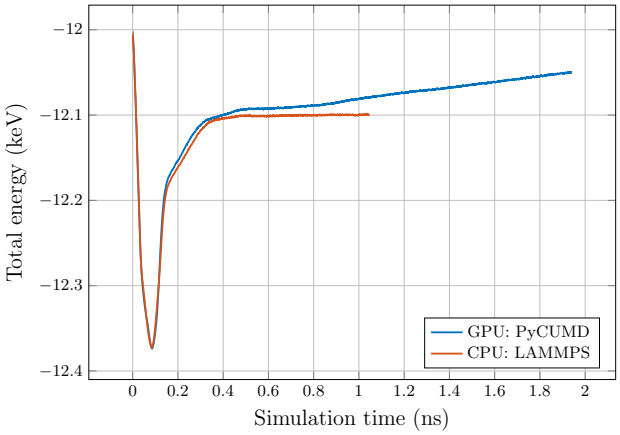
Another important question is of course the performance of the PyCUMD program as compared to LAMMPS, as well as the numerical stability of PyCUMD, Figure 4 answers these two questions. Worth noting is that the Nvidia K80 has two GPU chips on the same board and currently PyCUMD only uses one of them, for a fair comparison only one of the two E5-2690 v3 (16 cores) on System 2 was used for the LAMMPS simulation. Figure 4a shows the performance of the PyCUMD program as compared to LAMMPS, the performance is measured in millions of particles processed per second (Mp/s). It's clear that the GPU MD implementation is faster than the CPU one, with an average performance increase of 85.55% and a peak performance increase of 159.2% over LAMMPS.

For the numerical accuracy of an MD simulation one can look at the total energy of the system, in an NVE simulation this energy should be constant (e.g no energy drifts due to numerical errors). Figure 4b shows the total energy for the PyCUMD and LAMMPS simulation respectively, it's clear here that both simulations drift in their total energy. This energy drift is small though $< 5\%$ and similar for both the LAMMPS and PyCUMD simulations,

the total energy for the PyCUMD simulation appears to diverge at the later stages of the simulation probably due to floating point rounding errors.



(a) Performance comparison, blue curve is GPU performance and red CPU.



(b) Total energy for the system, used to test accuracy, for an NVE simulation this should be constant.

Figure 4: Performance and numerical accuracy of the PyCUMD program as compared to LAMMPS.

Some final thoughts about the double buffered GPU prefix sum and PyCUMD. The GPU prefix sum has a clear advantage over the CPU version, on System 1 which has a moderately powerful GPU and a very powerful CPU the performance increase is still about 10 times (2.5-3 times more powerful if one would consider a parallel CPU prefix sum). The fact that the double buffered algorithm is faster is surprising, pointing to the fact that the balanced trees algorithm still has room for improvements. Both of the prefix sum algorithm are memory limited, hence the shared memory alignment and access problems that balanced trees algorithm has coupled with a 12.5% branch divergence probably accounts for the lower performance.

For PyCUMD 98.7% of the compute time is spent in the force calculation kernel (`ljForce`), this is were performance can be gained. Due to the limited time of this project not as much optimizations could be made for this kernel, currently the performance seems to be limited by the register count (39) that gives an achieved vs theoretical occupancy of 64.4% vs 75%. An other big chunk of performance is probably lost due to branch divergence where the biggest divergence 40.6% comes from the handling of periodic boundary conditions. Even though the `ljForce` kernel clearly has lots of room for improvements PyCUMD is still about twice as fast as the highly optimized open source MD program LAMMPS and still shows similar accuracy which is impressive.

---

* Daniel.Hedman@ltu.se

[1] Counting sort, "Counting sort — Wikipedia, the free encyclopedia," (2015), [Online; accessed 20-January-2016].
[2] Prefix sum, "Prefix sum — Wikipedia, the free encyclopedia," (2015), [Online; accessed 20-January-2016].
[3] LAMMPS, "LAMMPS — molecular dynamics simulator," (2015), [Online; accessed 23-January-2016].
[4] M. Harris, "Parallel prefix sum (scan) with cuda — NVIDIA DEVELOPER ZONE, everything you need to develop with nvidia products," (2007), [Online; accessed 20-January-2016].