# Technical Report: SPIDAL Summer REU 2016: Distance Array and RMSD Speed-Up in MDAnalysis

Robert Delgado[1] Oliver Beckstein[2]

Currently, computation of the distance_array and rmsd for each frame in a molecular dynamics trajectory in MDAnalysis is a computationally intensive process. MDAnalysis (mdanalysis.org) is an open source library that allows for the analysis of molecular dynamics simulations. The goal of this REU was to speed up parts of MDAnalysis. We created a GPU based program through PyCUDA that allows us to compute distance arrays. PyCUDA (https://mathema.tician.de/software/pycuda/) is a package that allows us to access NVIDIA's CUDA API via python. Unfortunately, even though that it was evident that GPU's do compute distance arrays faster than CPU's, the transfer time between the local memory and the GPU is long enough such CPU computation is still faster. We also made slight modifications to the rmsd calculations to improve speeds by up to 30%.

## 1. Introduction

MDAnalysis[3] is a library that allows one to evaluate molecular dynamics trajectories. Just some of the things it can do are read and access atomic coordinates, and the aforementioned trajectories can be written out and manipulated. Additionally, it can also read multiple kinds of trajectories such as ones generated by CHARMM, Gromacs, NAND, LAMPS, AMBER, and DL_POLY.

PyCUDA[4] is a package that allows one to utilize NVIDIA's CUDA API via python. In addition to this, PyCUDA also has several convenient futures, which include automatic error checking in python rather than CUDA, the use of the full CUDA API, convenience due to the pycuda.driver.SourceModule and pycuda.gpuarray.GPUArray modules, and speed because its base layer is written in C++.

Distance_array is a method in MDAnalysis, which is used to compute the distances between each atom of two different molecules. The distance_array between an N x 3 molecule

---

[1] Department of Applied and Engineering Physics, Cornell University, Ithaca NY

[2] Department of Physics and Center for Biological Physics, Arizona State University, Tempe AZ.
E-mail: oliver.beckstein@asu.edu

[3] http://www.mdanalysis.org

[4] https://mathema.tician.de/software/pycuda/

and an M x 3 molecule will have size N x M. This is because we are finding the mutual distances between a set of N atoms and a set of M atoms.

RMSD, which stands for root-mean-square deviation, allows one to calculate the average distance between two superimposed proteins. Additionally, this method also returns a rotation matrix that when applied to a molecule, minimizes the value of RMSD.

## 2. PyCUDA

PyCUDA was used to attempt to speedup the computation times of the distance_array method. Using PyCUDA allows one to program NVIDIA based GPU's through python and be able to use NVIDIA'

To start a python PyCUDA program, one must first have PyCUDA installed and be using a NVIDIA based GPU. To fully understand PyCUDA based programs, one must first be familiar with threads, blocks, and grids.

### 2.1 Threads, Blocks, and Grids

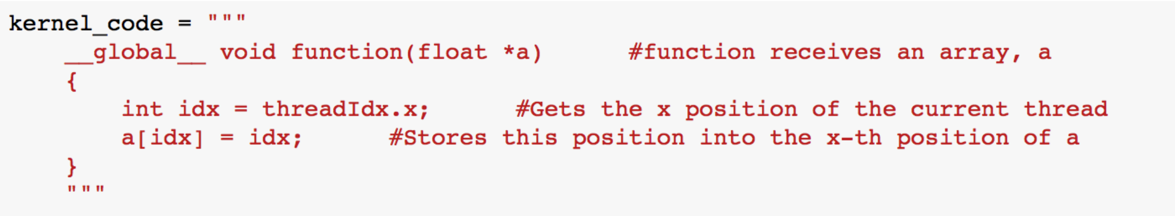
In all GPU programming, there exists threads, blocks, and grids. A thread is simply represents a single computation to be performed by the GPU. A block is a group of threads, usually of size 32 x 32, but some older GPU's are 16 x 16. A grid is a group of blocks, however, its size is determined by the maximum amount of memory that a GPU could hold, divided by the amount of memory each block takes.

When a GPU launches its kernel, which is a set of computations, it first begins to process the threads on a single block. Each thread of this block is computed at the same time. Once all of the threads of this block are executed, another block is executed. This process continues until all of the blocks in the grid have been executed.

If one has more data than can fit into a single grid on the GPU, one may have to send multiple grids of data to the GPU. This is accomplished by first partitioning the data into groups whose size can fit onto a single grid. Next, one grid would be loaded on to the CPU and then processed. After that, the grid would be removed from the GPU and this process would continue until all of the grids have been processed.

### 2.2 Using PyCUDA

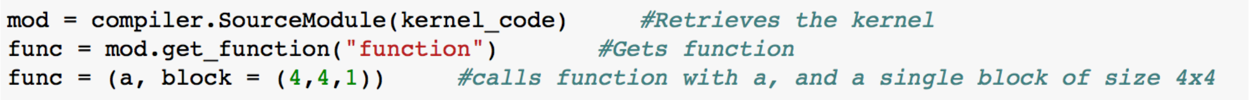
Every PyCUDA program needs a kernel, which is a set of instructions that tells the GPU how to evaluate each thread. If there is no kernel, then the code would not compile. Additionally, kernels are not written using python syntax. All variables are hard-typed, comments can only be done with the '#' character, there must be a ';' after every line, and there must be 3 quotation characters before and after the kernel code. An example of a kernel is found below in Figure 1:



```
kernel_code = """
    __global__ void function(float *a)      #function receives an array, a
    {
        int idx = threadIdx.x;      #Gets the x position of the current thread
        a[idx] = idx;      #Stores this position into the x-th position of a
    }
    """
```

**Figure 1. This is an example of a kernel code. This kernel takes in an array, a, and for every thread it processes, it stores its x value into a local int, idx. Then the idx-th value of our array a would be changed to idx.**

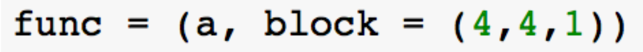
To launch the kernel, PyCUDA allows us to conveniently do it in three lines, as shown in [Fig 2]



```
mod = compiler.SourceModule(kernel_code)      #Retrieves the kernel
func = mod.get_function("function")       #Gets function
func = (a, block = (4,4,1))      #calls function with a, and a single block of size 4x4
```

**Figure 2. These three lines launch the kernel. The first line compiles and retrieves the kernel. The second gets the method 'function' that we included in Figure 1. And the last line sends an array a, and a 4x4x1 block of arrays to the kernel.  If we did not specify the block size, it would be 32x32x1**

Additionally, one can see how the above kernel changes a specific value of 'a' in the diagram below.

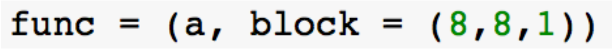


```
func = (a, block = (4,4,1))
```

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \longrightarrow A = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

**Figure 3. Here we have a as a 4 x 4 array filled with zeros, and we are evaluating 16 threads. However, each thread will have an x-value of 0, 1, 2, or 3. Hence, the 0th, 1st, 2nd, and 3rd values of A are set to 0, 1, 2, and 3 respectively. This happens 4 times each because there are 4 threads with an x value of 0, 4 with 1, and so on.**

We can also change the size of our block, as shown below.

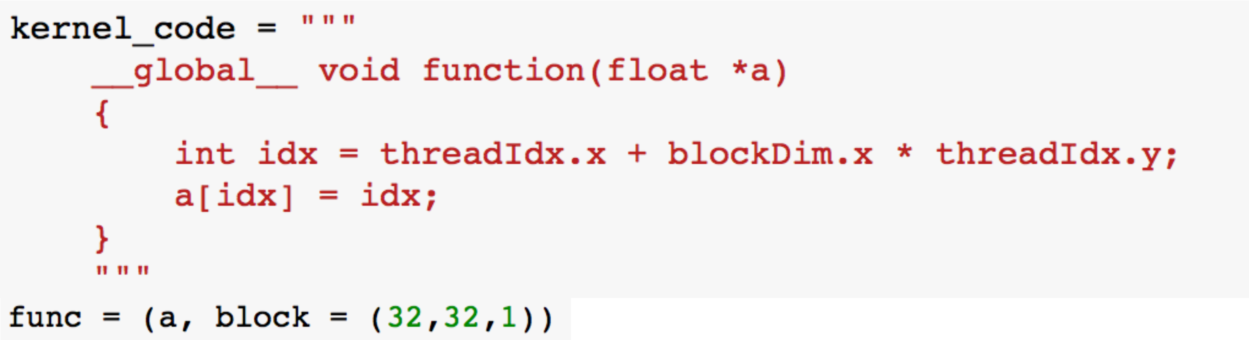


```
func = (a, block = (8,8,1))
```

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \longrightarrow A = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

**Figure 4. Sending a 8 x 8 x 1 a block now means there are 8 different x values. This means 8 different values of A would be changed, the ones in positions 0 through 7. They are changed 8 times each because there are 8 threads with the same x position.**

Now, we will show a slightly different kernel that deals with arrays that are equal to the maximum block size, and larger than it.



```
kernel_code = """
    __global__ void function(float *a)
    {
        int idx = threadIdx.x + blockDim.x * threadIdx.y;
        a[idx] = idx;
    }
    """
func = (a, block = (32,32,1))
```

**Figure 5. Sets the i-th position to i, and sends ‘a’ and a block of 32 x 32 x 1 threads to the kernel.**

32 x 32

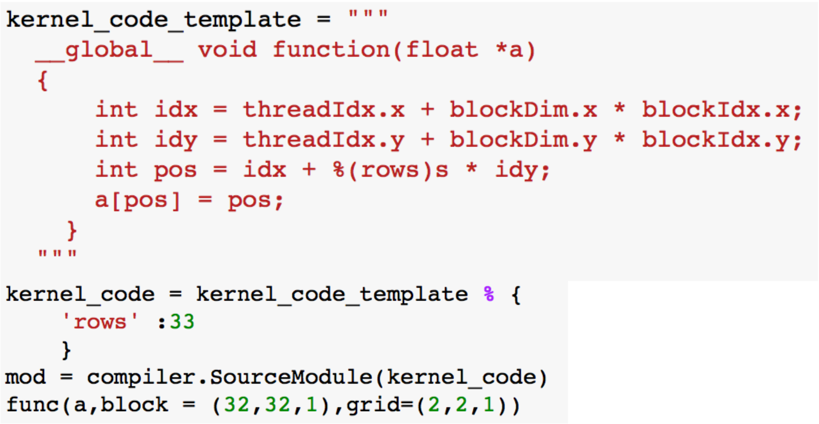
$$A = \begin{bmatrix} 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 \end{bmatrix} \quad \longrightarrow \quad A = \begin{bmatrix} 0 & 1 & \cdots & 30 & 31 \\ 32 & 33 & & 62 & 63 \\ \vdots & & \ddots & & \vdots \\ 960 & 961 & & 990 & 991 \\ 992 & 993 & \cdots & 1022 & 1023 \end{bmatrix}$$

33 x 33

$$A = \begin{bmatrix} 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 \end{bmatrix} \quad \longrightarrow \quad A = \begin{bmatrix} 0 & 1 & \cdots & 31 & 32 \\ 33 & 34 & & 64 & 65 \\ \vdots & & \ddots & & \vdots \\ 1023 & 0 & & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 \end{bmatrix}$$

**Figure 6. When A is a 32 x 32 array the above kernel sets the i-th value to i. This happens one time for each value. But when A is a 33 x 33 array, it sets the i-th value, up to 1023, to i. This is because our block has only 1024 values so any value above that would not be reached.**

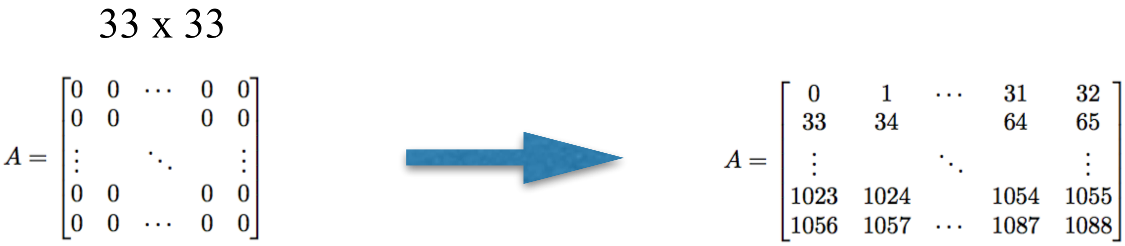
To fix this problem we will have to send multiple blocks to the GPU. This is shown below.



```
kernel_code_template = """
  __global__ void function(float *a)
  {
      int idx = threadIdx.x + blockDim.x * blockIdx.x;
      int idy = threadIdx.y + blockDim.y * blockIdx.y;
      int pos = idx + %(rows)s * idy;
      a[pos] = pos;
    }
  """
kernel_code = kernel_code_template % {
    'rows' :33
    }
mod = compiler.SourceModule(kernel_code)
func(a,block = (32,32,1),grid=(2,2,1))
```

**Figure 7. Modifies the previous kernel such that it now sends 4 blocks to change the i-th values of a to i.**

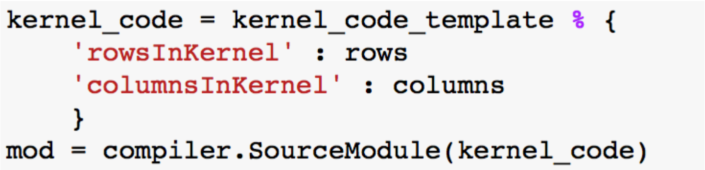
The change in the array can now be seen in [Fig 8].



$$A = \begin{bmatrix} 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 \end{bmatrix} \longrightarrow A = \begin{bmatrix} 0 & 1 & \cdots & 31 & 32 \\ 33 & 34 & & 64 & 65 \\ \vdots & & \ddots & & \vdots \\ 1023 & 1024 & & 1054 & 1055 \\ 1056 & 1057 & \cdots & 1087 & 1088 \end{bmatrix}$$

**Figure 8: Our updated kernel now takes all of the i-th values of the array and sets them to i.**

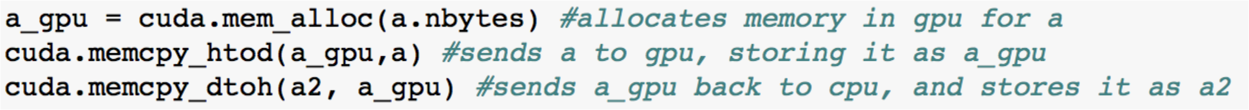
Another important command in PyCUDA is to send variables to the kernel. Normally, the kernel cannot access any of the global variables and they must have to be explicitly sent to the GPU. This is shown in [Fig 9].



```
kernel_code = kernel_code_template % {
    'rowsInKernel' : rows
    'columnsInKernel' : columns
    }
mod = compiler.SourceModule(kernel_code)
```

**Figure 9: This sends local variables to the kernel. The variables rows and columns can now be accessed as rowsInKernel and columnsInKernel in the kernel.**

To get the updated result from the kernel, we only need to use three lines in [Fig. 10]



```
a_gpu = cuda.mem_alloc(a.nbytes) #allocates memory in gpu for a
cuda.memcpy_htod(a_gpu,a) #sends a to gpu, storing it as a_gpu
cuda.memcpy_dtoh(a2, a_gpu) #sends a_gpu back to cpu, and stores it as a2
```

**Figure 10. Memory transfer between local memory and the GPU. The first line allocates memory for a in GPU. The second line sends a to GPU and stores it as a_gpu. After the kernel is done evaluating the threads, the third line is called. It stores the updated a_gpu into a new value, a2 .**

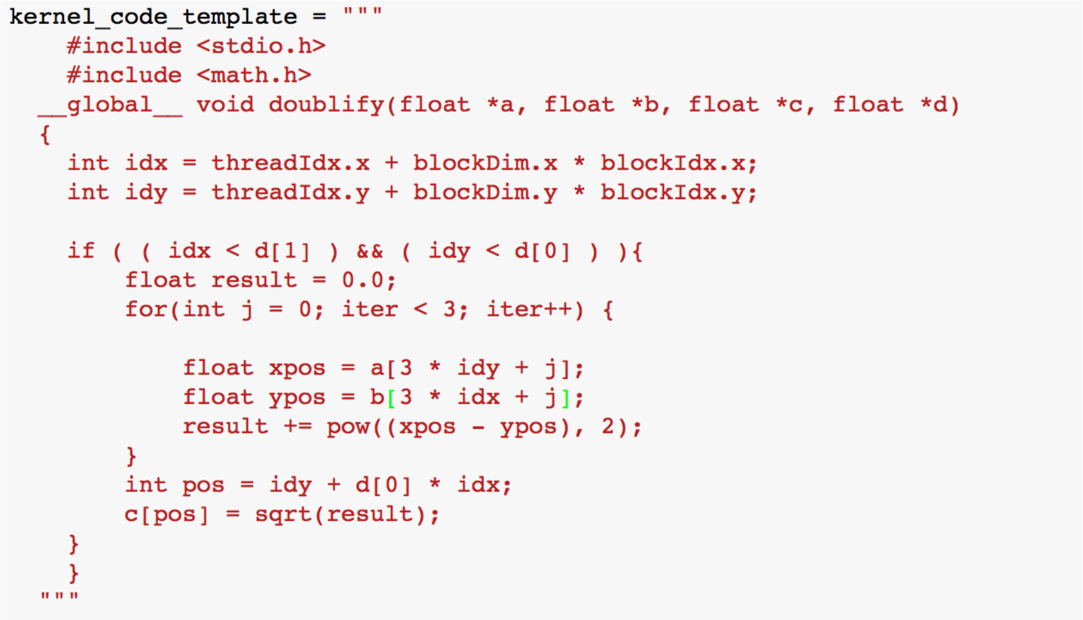
## 2.3 Distance_array

A distance array is defined as a 2 dimensional array that holds the values of the distances between each atom of two molecules. Mathematically, the distance array is defined as:

$$D_{ij} = \sqrt{\sum_{n=1}^{3}(A_{in} - B_{jn})^2}$$

**Figure 11: Distance array, D, between two molecules, A and B. $A_{in}$ stands for coordinate n of atom i in molecule A. $B_{jn}$ stands for the coordinate n of atom j in molecule B.**

Writing a PyCUDA program for distance_array consisted of first finding the grid sizes necessary to have enough threads to complete a whole distance_array. Remember, the amount of threads should be greater than (N*M), where N is the amount of atoms in the first molecule and M is the amount of atoms in the second molecule. Then, each block that is processed would compute 32 x 32, or 1024 entries of the distance_array. The kernel is found in [Fig . 12] and the determination of the grid size is found in [Fig. 13]

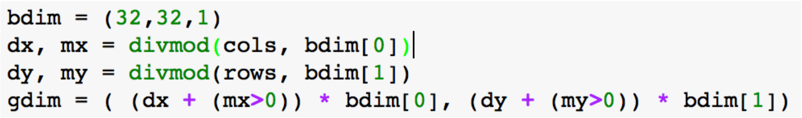


```
kernel_code_template = """
    #include <stdio.h>
    #include <math.h>
  __global__ void doublify(float *a, float *b, float *c, float *d)
  {
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    int idy = threadIdx.y + blockDim.y * blockIdx.y;

    if ( ( idx < d[1] ) && ( idy < d[0] ) ){
        float result = 0.0;
        for(int j = 0; iter < 3; iter++) {

            float xpos = a[3 * idy + j];
            float ypos = b[3 * idx + j];
            result += pow((xpos - ypos), 2);
        }
        int pos = idy + d[0] * idx;
        c[pos] = sqrt(result);
    }
    }
  """
```

**Figure 12: Kernel that computes the distance_array of two molecules.**



```
bdim = (32,32,1)
dx, mx = divmod(cols, bdim[0])
dy, my = divmod(rows, bdim[1])
gdim = ( (dx + (mx>0)) * bdim[0], (dy + (my>0)) * bdim[1])
```

**Fig. 13. Code to get the grid sizes.**

## 2.4 The Downside of using GPU Programming

Initial benchmarks of computing a distance array between two molecules on a CPU showed that it takes a few hundredths of a second to compute, depending on the size of the molecules [Fig. 14]. On the other hand, computing the distance array between those same molecules on a GPU takes a few tenths of a second [Fig 15].

The main reason for this large discrepancy in benchmarks is that it takes roughly .15 seconds to send each grid of data from memory to GPU. This is a property of all GPU's and cannot be worked around. Additionally, despite the speed up from using a GPU rather than a CPU to do the calculations, the gain in speed was not enough to overcome the transfer time. Even if larger molecules were used, there would still be no significant speed up because despite the fact that the GPU would process the calculations quicker, more grids would have to be loaded, which would take an even longer amount of time. Data for transfer time from NVIDIA[5] is included in [Fig. 16].

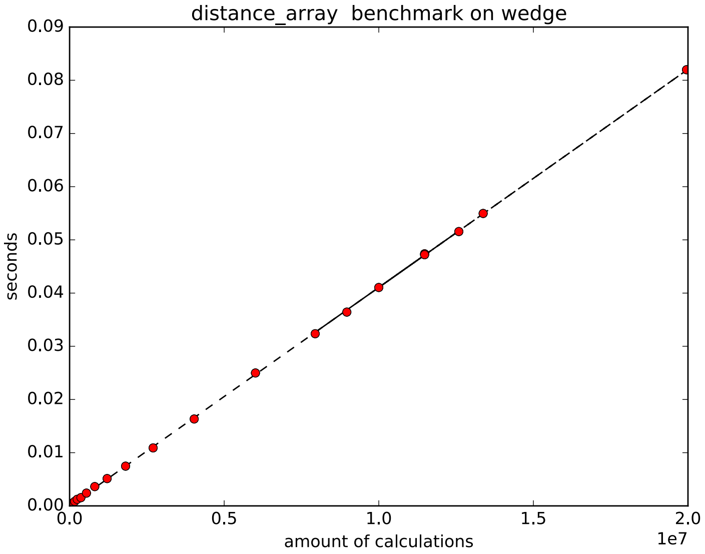


**Figure 14. Benchmarks for distance_array on a CPU. "Amount of Calculations" refers to the number of distances computed. Wedge has a processor with Intel(R) Core(TM) i7-4930K CPU @ 3.40GHz .with NVIDIA Quadro K4000 GPU**

---

[5] https://www.cs.virginia.edu/~mwb7w/cuda_support/memory_transfer_overhead.html

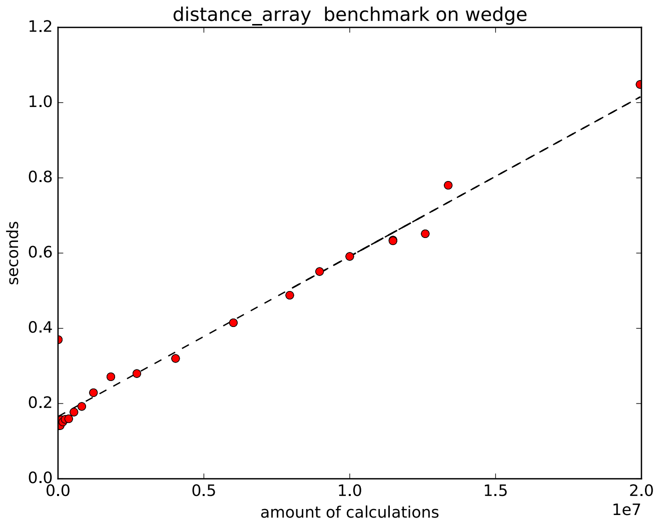


**Figure 15. Benchmarks for distance_array on a GPU. "Amount of Calculations" refers to the number of distances computed.**

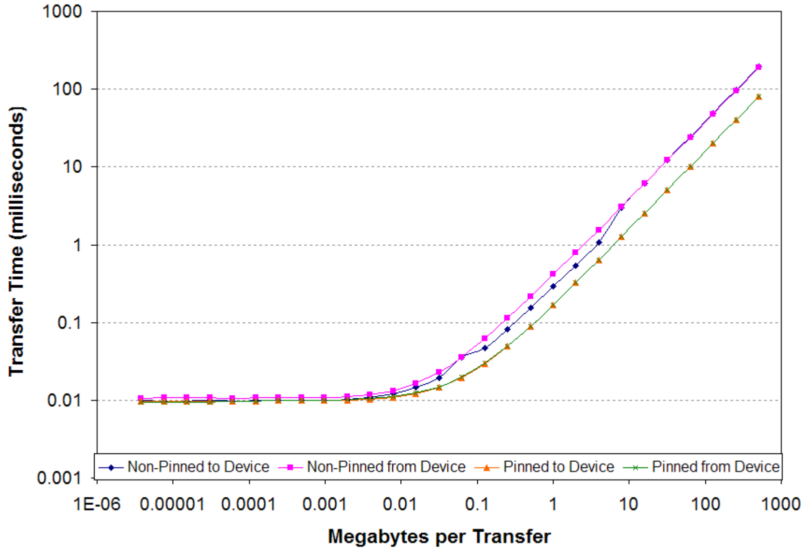


**Figure 16. Transfer time for an NVIDIA GPU, based on the amount of data being transfered.**
https://www.cs.virginia.edu/~mwb7w/cuda_support/memory_transfer_overhead.html

## 3. RMSD speed up

Another problem that was tackled during the REU was to speed up the RMSD method. RMSD, which stands for root mean square deviation, is a method that calculates the average distance between two superimposed coordinate arrays of two molecules. These two coordinate arrays are superimposed such that there centroids overlap. The formula to compute the RMSD is in [Fig.17].

$$\mathrm{RMSD}(A, B) = \sqrt{\frac{1}{N} \sum_{i=1}^{N} \sum_{n=1}^{3} (A_{in} - B_{in})^2}$$

**Figure 17. N is the amount of atoms and δ is the distance between an tom of the two structures. $A_{in}$ stands for coordinate n of atom i in molecule A. $B_{in}$ stands for the coordinate n of atom i in molecule B.**

Computing the rmsd used a method proposed by Theobald (Theobald, 2005), where a quaternion is used to iteratively determine the rmsd value. The rmsd method also supplies a rotation matrix (Pu, 2010) that when applied to one of the arrays, minimizes the rmsd value. But, however, this method was slow in that it did not take in the same type of array used by MDAnalysis. In MDAnalysis, the molecules are of size N x 3 and are of type numpy.float32, yet the implemented method specified that the molecules must be of size 3 x N and be of type numpy.float64. Hence users of MDAnalysis had to transpose the coordinate arrays of the molecules and recast them as numpy.float64 in order to use the rmsd methods. This took an unnecessary amount of time and rewriting the rmsd code to accept N x 3 matrices would speed up computation time. However, we must continue to cast to numpy.float64 because if we kept our precision at numpy.float32, the precision for some methods such as superposition becomes valid up to 1e-4. Keeping it as numpy.float64 retained the precision up to 1e-7. The benchmarks can be seen in [Fig. 18].

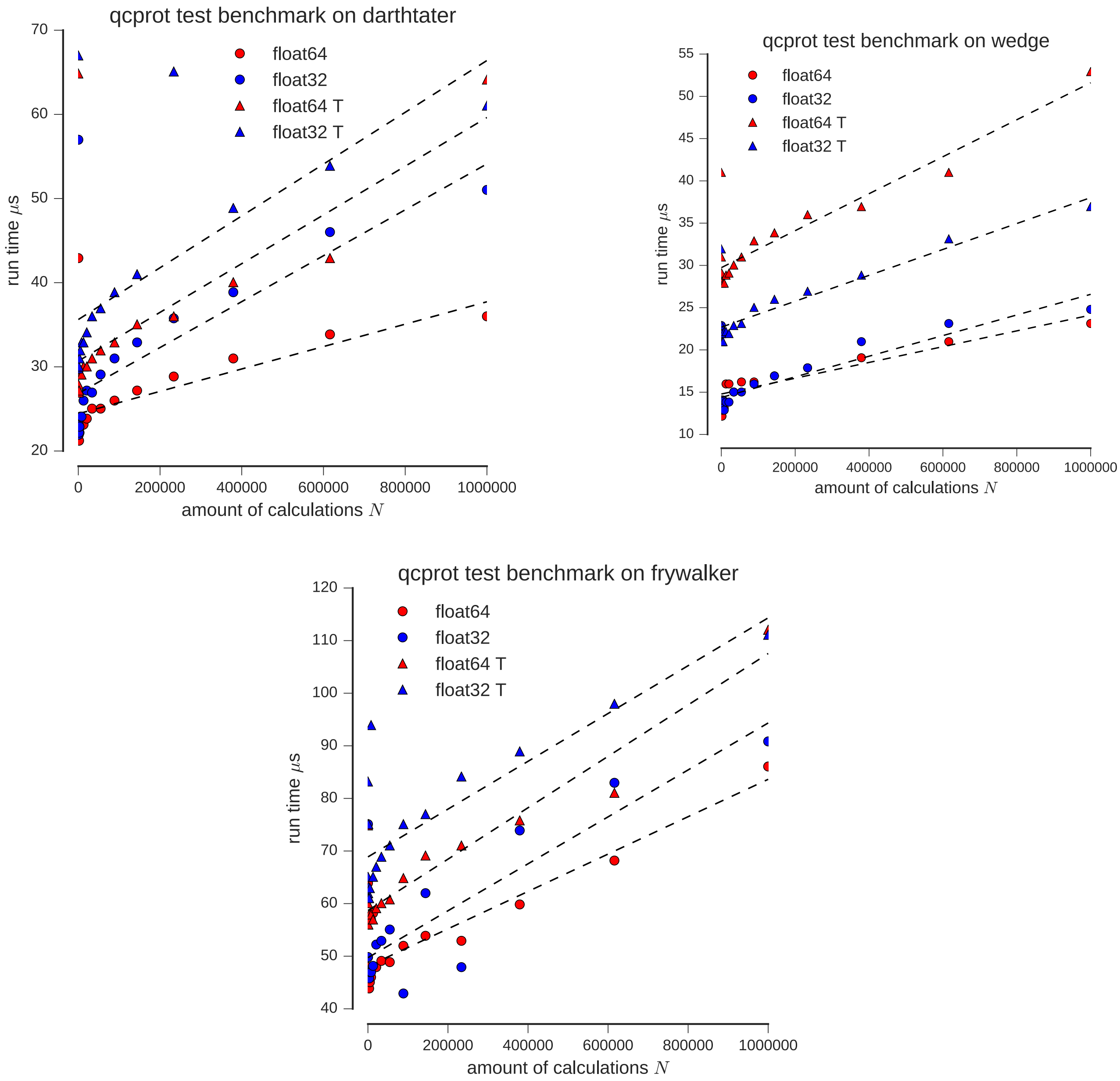


**Figure 18. Benchmarks of running the rmsd calculations. float64 and float 32 means no transpose. float64 T and float32 T mean a transposed had to be performed. The Benchmarks were run on "wedge", a Linux workstation with 32 GB of RAM and a 6 core Intel(R) Core (TM) i7-4930K CPU @ 3.40Ghz, "frywalker", a linux workstation with Intel(R) Xeon(R) CPU E5-2665 0 @ 2.40GHz, and "darthtater", a linux workstation with Intel(R) Xeon(R) CPU E5-2630 0 @ 2.30GHz.**

## 4. Conclusion

The work that was performed on distance_array and rmsd yielded two strong results. First, it was shown that the transfer time was too long to make use of the GPU for computing the distance array. Even though more threads could be executed in parallel on a GPU versus a CPU (1024 compared to 8), the transfer time was too long to make the parallelization worth it.

Second, changing the rmsd code so that we no longer have to transpose the protein molecules saves on average 30%. Additionally, the casting to numpy.float64 was kept to maintain a high degree of precision. If not, the precision for many methods drops by a factor of roughly 1e-3.

## 5. References

Douglas L. Theobald (2005) “Rapid calculation of RMSD using a quaternion-based characteristic polynomial.” Acta Crystallographica A 61(4):478-480.

R. J. Gowers, M. Linke, J. Barnoud, T. J. E. Reddy, M. N. Melo, S. L. Seyler, D. L. Dotson, J. Domański, S. Buchoux, I. M. Kenney, and O. Beckstein. MDAnalysis: A Python package for the rapid analysis of molecular dynamics simulations. In S. Benthall and S. Rostrup, editors, Proceedings of the 15th Python in Science Conference, pages 102 – 109, Austin, TX, 2016. SciPy. URL http://mdanalysis.org.

Pu Liu, Dmitris K. Agrafiotis, and Douglas L. Theobald (2010) “Fast determination of the optimal rotational matrix for macromolecular superpositions.” J. Comput. Chem. 31, 1561-1563.

N. Michaud-Agrawal, E. J. Denning, T. B. Woolf, and O. Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. J Comp Chem, 32:2319–2327, 2011. doi: 10.1002/jcc.21787.

https://www.cs.virginia.edu/~mwb7w/cuda_support/memory_transfer_overhead.html