CME253 Introduction to GPU Computing

Protein-Ligand Contact Distances for Many Ligand Poses

February 17, 2017

Jared Dunnmon
jdunnmon@stanford.edu

Brooke Husic
bhusic@stanford.edu

Overview: Application. Drug screening requires assessing the (energetic) relationship of many ligands to a specific protein binding pocket. Here, we create a simplified model for assessing this relationship by calculating concatenated vectors that represent drug-ligand binding by enumerating the distances between every atom in the protein and every atom in the ligand.
Inputs. All versions of the code require two input arguments: the file location of a protein and the file location of a series of ligands. We have provided input files of the proper format. For benchmarking we use a subunit of the NMDA receptor protein, which is found in neurons and is important for memory function, and a glutamic acid ligand, which is a known NMDAR agonist. The ligand poses are generally nonphysical and the Cartesian coordinates of the atoms were created using random perturbations from a physical pose.
Approach. We first create our list of protein and ligand atom locations, with ligand series of several different lengths (3, 10, 100, 1000 poses in the series, 17 atoms for each pose) in order to assess CPU vs. GPU performance at various levels of computational intensity. Given that the operation we choose to perform is relatively simple, it will be interesting to see if the GPU actually provides improved performance. We implement the distance function as:

$$d = \sqrt{(x_L - x_P)^2 + (y_L - y_P)^2 + (z_L - z_P)^2}/10,$$

where the last operation is required in order to retrieve values with appropriate units in angstroms. There exist several ways that one could propose to perform this type of a function. Each thread could, for instance, compute distance between a single ligand location and a single protein location, thereby distributing the computation amongst many different threads in a relatively efficient manner. Similarly, one could imagine a situation wherein a series of four threads performs each distance computation – three threads are used to perform the subtraction and squares for the x, y, and z coordinates while a fourth is used to perform the square root and division after agglomeration (i.e. reduction). In the work below, we investigate the ability of GPU-based hardware parallelism to compute these distances faster than a naive CPU-based implementation.
Benchmark C implementation: Approach. To compute every distance between every atom in the protein and every atom in every ligand, given the concatenated input of the ligand poses we require two for loops, one through the atoms in the protein and one through the atoms in the file of ligand poses. **Performance and execution.** We expect the distance computation to be slow because the distances are computed sequentially. However, there is no memory transfer since the computation takes place on the CPU.
CUDA implementation 1: Approach. We first linearize the Cartesian coordinates of the protein and ligand series and then use a 2-dimensional thread to compute the distances for the appropriate indices of the threads. One thread moves along the linearized protein coordinates and one thread moves along the linearized ligand series coordinates. The kernel uses as many blocks as necessary to accommodate (32 × 32 × 1) threads per block. **Performance and execution.** As expected, the GPU/CUDA implementation can compute the set of contact distances approximately 100 times faster than the CPP implementation for a variety of different problem sizes; see Table 1.

CUDA implementation 2: Shared Memory: Approach. In this implementation, we attempt to ensure that writing coalescence to global memory is not causing a substantial slowdown in our computation. Specifically, we define a shared memory array to store the distances computed by each block. After synchronizing threads to ensure that all distances have been calculated in a thread-by-thread fashion, the entire shared memory array is written back to the global distances vector. **Performance and execution.** While theoretically sound, this approach did not provide additional speedup over the original thread-by-thread implementation.
CUDA implementation 3: Reduction with SMEM: Approach. Since the distance calculation requires the independent summation and squaring of distances in three dimensions, we decided to reduce our problem into different parts and then sum them together afterward. For this we used three threads. As in the original CUDA implementation, one thread moves along the linearized protein coordinates and one thread moves along the linearized ligand series coordinates. Then we have a third thread that indicates whether the x, y, or z distances are computed. Since the kernel can only accommodate 1024 threads across three dimensions, we initially used $(16 \times 16 \times 4)$ threads. However, in this case, the 4th in each z block is unused because there are only three tasks to complete. To address this, we used $(18 \times 18 \times 3)$ threads, which is the largest number attainable less than 1024 if we are only using 3 threads in the z direction. This implementation also included use of shared memory. **Performance and execution.** This reduced implementation performed worse than the original CUDA implementation with $(32 \times 32 \times 1)$ threads (see Table 1). In neither reduction case could the design use all 1024 threads available per block, and this diminished the performance. The discrepancy in performance scales as the size of the input files increases, because the total unused resources increase for when more blocks are needed.
Discussion and relation to course concepts. The process of implementing this relatively simple featurizer in a variety of different ways has allowed for several useful conclusions, and enabled the exploration of a variety of class concepts. Firstly, as shown in Fig. 1, GPU-based computation yields substantial speedups across implementations, ranging from 38x-95x over the CPU implementation. The differences in these implementations, and the reasons that they were achieved, are worth noting. The thread-by-thread implementation of the distance calculation, for instance, yields faster computation by nearly a factor of three over the reduction method, likely as a result of fewer syncthread requirements and the ability to efficiently use every thread for a useful purpose. Further, the output of the Nvidia visual profiler, which was run on every implementation demonstrated here, yields a variety of useful conclusions and opportunities for future improvement.

In general, compute efficiency for these codes was relatively high (¿75% for thread-by-thread), but memory efficiency could be quite low (¡30 % for reduction). This indicates that next steps should involved detailed consideration of memory access patterns to maximize GPU utilization for larger computations. The low compute/memcpy efficiency can also be rectified by performing more complex calculations – in fact, we designed a pathologically complex scalar potential that requires a large number of operations per thread (not reported for brevity), and were able to substantially improve this ratio. Low overlap between memcpy/compute, low kernel concurrency, and low memcpy parallelization could all be addressed via careful consideration of streaming and concurrency for larger calculations – we have not attempted this as part of this brief project, but are aware that it is a logical next step. Further, the profiler yields useful information in that the cudaMalloc call tends to dominate the required overall computation time. In practice, we could circumvent this problem for repeated calculations by only calling cudaMalloc once, but performing repeated copies and calculations using the same memory addresses if we were to investigate multiple ligand candidates simultaneously for a single protein. We could also improve the effect of share memory, which for our implementation does not help a great deal because we do not commonly access the same values repeatedly, by computing multiple ligand-protein pairs at the same time, using the same protein and multiple ligands. In this case, we could use shared memory to store the protein coordinates and compute all of the relevant ligand distances with respect to that protein. There are also certainly ways to optimize our reduction mechanism, as shown in the course exercises. Ideally, we could efficiently utilize all 1024 threads in 3-dimensional grid structure for better performance.

Table 1: Calc. is the time taken to calculate the distances. Copy is the time taken to copy the distances back to the CPU. Initial memory allocation was not timed. Note that different GPUs were used for different benchmarks.

Distances	C++ (μs)	CUDA (μs)	CUDA+ SMEM (μs)	CUDA+ SMEM+RED (μs)	CUDA+ SMEM+RED (μs)
Threads:	1	$(32 \times 32 \times 1)$	$(32 \times 32 \times 1)$	$(16 \times 16 \times 4)$	$(18 \times 18 \times 3)$
233223	115323	1859	1754	2666	2258
777410	380287	3940	4008	39680	5965
7774100	3787789	35140	34607	68893	52063
77741000	38025128	352768	335852	812300	575773

Appendix

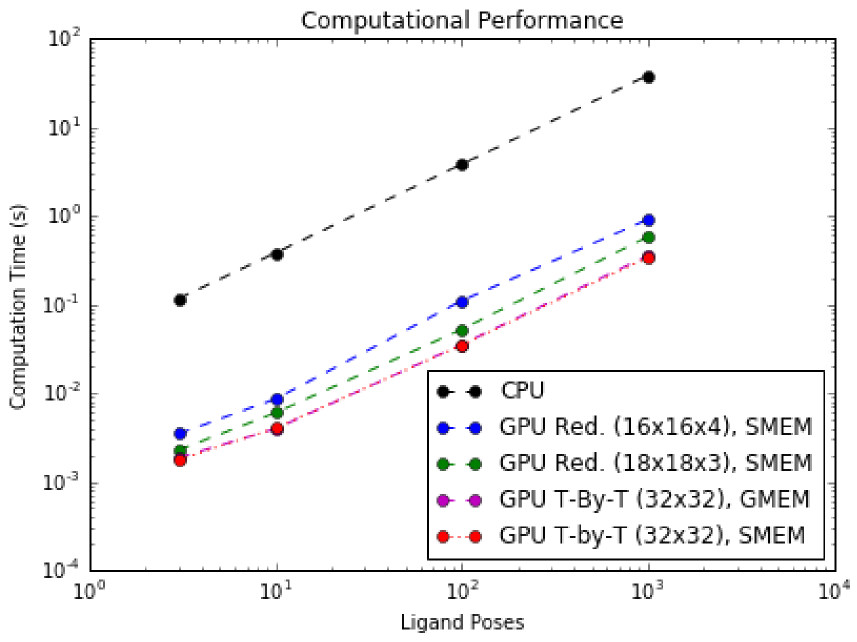

Table 2: Results from the profiler. Two profiler screen captures are included in the project directory.

	CUDA	CUDA+ SMEM	CUDA+ SMEM+RED	CUDA+ SMEM+RED
Threads:	$(32 \times 32 \times 1)$	$(32 \times 32 \times 1)$	$(16 \times 16 \times 4)$	$(18 \times 18 \times 3)$
Memcpy (H→D) (μs)	19	19	19	19 us
Memcpy (H→D) (kB)	119	113	113	113
Memcpy (D→H) (ms)	2.6	2.6	2.4	2.6
Memcpy (H→D) (MB)	6.2	6.2	6.2	6.2
Kernel (ms)	0.555	0.595	7.3	2.7
cudaMalloc (ms)	65	84	97	100
Compute utilization (%)	65	65	75	78
Memory utilization (%)	33	33	35	25
Average speedup vs. C++	95.825	93.534	38.685	63.405