Parallel Computing on Boundary Integral Poisson-Boltzmann Solver

Presented by Sharon Yang, Math 6370

Collaborated with: Weihua Geng, Jiahui Chen

Reference

A GPU-accelerated direct-sum boundary integral Poisson–Boltzmann solver

Weihua Geng [a,*], Ferosh Jacob [b]

[a] Department of Mathematics, University of Alabama, Tuscaloosa, AL 35487, USA
[b] Department of Computer Science, University of Alabama, Tuscaloosa, AL 35487, USA

ARTICLE INFO

Article history:
Received 9 December 2012
Received in revised form
19 January 2013
Accepted 23 January 2013
Available online 1 February 2013

Keywords:
Poisson–Boltzmann
Electrostatics
Boundary integral
Parallel computing
Graphic processing units (GPU)

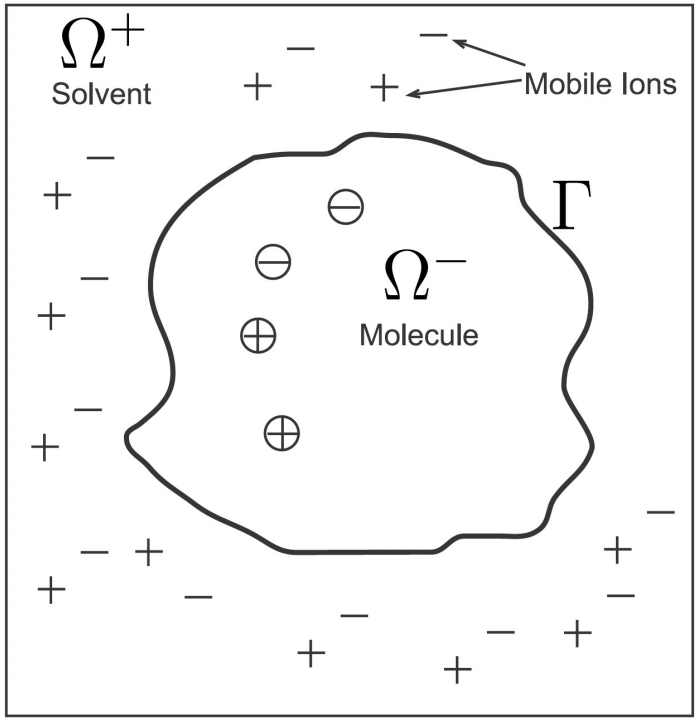
ABSTRACT

In this paper, we present a GPU-accelerated direct-sum boundary integral method to solve the linear Poisson–Boltzmann (PB) equation. In our method, a well-posed boundary integral formulation is used to ensure the fast convergence of Krylov subspace based linear algebraic solver such as the GMRES. The molecular surfaces are discretized with flat triangles and centroid collocation. To speed up our method, we take advantage of the parallel nature of the boundary integral formulation and parallelize the schemes within CUDA shared memory architecture on GPU. The schemes use only $11N+6N_c$ size-of-double device memory for a biomolecule with N triangular surface elements and N_c partial charges. Numerical tests of these schemes show well-maintained accuracy and fast convergence. The GPU implementation using one GPU card (Nvidia Tesla M2070) achieves 120–150X speed-up to the implementation using one CPU (Intel L5640 2.27 GHz). With our approach, solving PB equations on well-discretized molecular surfaces with up to 300,000 boundary elements will take less than about 10 min, hence our approach is particularly suitable for fast electrostatics computations on small to medium biomolecules.

Weihua Geng, Ferosh Jacob. (2013). “A GPU-accelerated direct-sum boundary integral Poisson–Boltzmann solver”. *Computer Physics Communications*, 184(6), 1490-1496.

Poisson-Boltzmann model

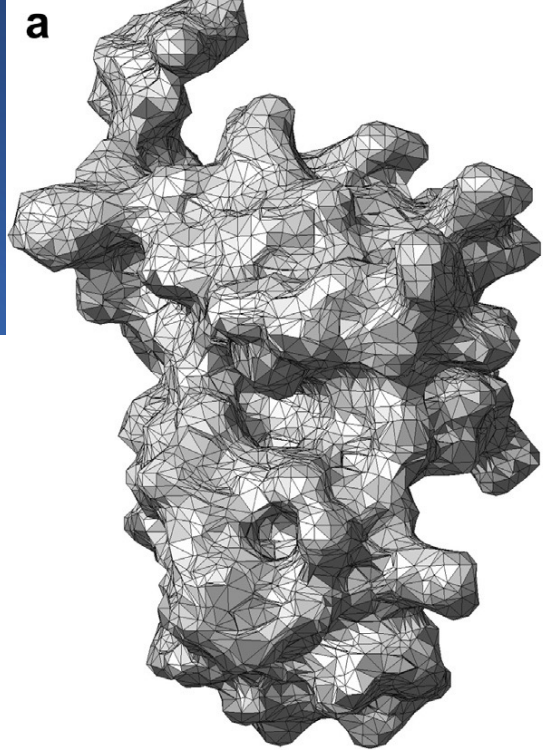
- $\nabla \cdot (\varepsilon_1(\mathbf{x})\nabla\phi_1(\mathbf{x})) = -\sum_{i=1}^{N_c} q_i \delta(\mathbf{x}-\mathbf{x}_i)$ in Ω_1 (1)
- $\nabla \cdot (\varepsilon_2(\mathbf{x})\nabla\phi_2(\mathbf{x})) - \kappa^2\phi_2(\mathbf{x}) = 0$ in Ω_2 (2)
- $\phi_1(\mathbf{x}) = \phi_2(\mathbf{x})$
- $\varepsilon_1 \frac{\partial\phi_1(\mathbf{x})}{\partial v} = \varepsilon_2 \frac{\partial\phi_2(\mathbf{x})}{\partial v}$ on $\Gamma = \partial\Omega_1 = \partial\Omega_2$ (3)
- $\lim_{|\mathbf{x}|\to\infty} \phi_2(\mathbf{x}) = 0$ (4)



Well-posed boundary integral formulation

- Coulomb and screened Coulomb potentials: $\mathrm{G}_0(\mathbf{x},\mathbf{y}) = \frac{1}{4\pi|\mathbf{x}-\mathbf{y}|}$, $\mathrm{G}_\kappa(\mathbf{x},\mathbf{y}) = \frac{e^{-\kappa|\mathbf{x}-\mathbf{y}|}}{4\pi|\mathbf{x}-\mathbf{y}|}$ (5)
- $\frac{1}{2}(1+\varepsilon)\phi_1(\mathbf{x}) = \int_\Gamma \left[\mathrm{K}_1(\mathbf{x},\mathbf{y})\frac{\partial\phi_1(\mathbf{y})}{\partial v_\mathbf{y}} + K_2(\mathbf{x},\mathbf{y})\phi_1(\mathbf{y})\right] dS_\mathbf{y} + S_1(\mathbf{x})$ (6)
- $\frac{1}{2}\left(1+\frac{1}{\varepsilon}\right)\frac{\partial\phi_1(\mathbf{x})}{\partial v_\mathbf{x}} = \int_\Gamma \left[K_3(\mathbf{x},\mathbf{y})\frac{\partial\phi_1(\mathbf{y})}{\partial v_\mathbf{y}} + K_4(\mathbf{x},\mathbf{y})\phi_1(\mathbf{y})\right] dS_\mathbf{y} + S_2(\mathbf{x})$ (7)
- $K_1(\mathbf{x},\mathbf{y}) = G_0(\mathbf{x},\mathbf{y}) - G_\kappa(\mathbf{x},\mathbf{y}), \quad K_2(\mathbf{x},\mathbf{y}) = \varepsilon\frac{\partial G_\kappa(\mathbf{x},\mathbf{y})}{\partial v_\mathbf{y}} - \frac{\partial G_0(\mathbf{x},\mathbf{y})}{\partial v_\mathbf{y}}$
- $K_3(\mathbf{x},\mathbf{y}) = \frac{\partial G_0(\mathbf{x},\mathbf{y})}{\partial v_\mathbf{x}} - \frac{1}{\varepsilon}\frac{\partial G_K(\mathbf{x},\mathbf{y})}{\partial v_\mathbf{x}}, \quad K_4(\mathbf{x},\mathbf{y}) = \frac{\partial^2 G_\kappa(\mathbf{x},\mathbf{y})}{\partial v_\mathbf{x}\partial v_\mathbf{y}} - \frac{\partial^2 G_0(\mathbf{x},\mathbf{y})}{\partial v_\mathbf{x}\partial v_\mathbf{y}}$ (8)
- $S_1(\mathbf{x}) = \sum_{k=1}^{N_c} q_k G_0(\mathbf{x},\mathbf{y}_k), \qquad S_2(\mathbf{x}) = \sum_{k=1}^{N_c} q_k \frac{\partial G_0(\mathbf{x},\mathbf{y}_k)}{\partial v_\mathbf{x}}$ (9)

Discretization



- Use triangulation program MSMS to discretize the molecular surface
- Discretize (6) and (7) with the flat triangles and the centroid collocation
- Converted into a linear system $\mathbf{Au} = \mathbf{b}$
- $\{\mathbf{Au}\}_i = \frac{1}{2}(1+\varepsilon)\phi_1(\mathbf{x}_i) - \sum_{j=1,j\neq i}^{N} W_j \left[K_1(\mathbf{x}_i,\mathbf{x}_j)\frac{\partial\phi_1(\mathbf{x}_j)}{\partial v_{\mathbf{x}_j}} + K_2(\mathbf{x}_i,\mathbf{x}_j)\phi_1(\mathbf{x}_j)\right]$ (10)
- $\{\mathbf{Au}\}_{i+N} = \frac{1}{2}\left(1+\frac{1}{\varepsilon}\right)\frac{\partial\phi_1(\mathbf{x}_i)}{\partial v_{\mathbf{x}_i}} - \sum_{j=1,j\neq i}^{N} W_j \left[K_3(\mathbf{x}_i,\mathbf{x}_j)\frac{\partial\phi_1(\mathbf{x}_j)}{\partial v_{\mathbf{x}_j}} + K_4(\mathbf{x}_i,\mathbf{x}_j)\phi_1(\mathbf{x}_j)\right]$ (11)
- The expressions of $\mathbf{b}_i$ and $\mathbf{b}_{i+N}$ are directly obtained from the source terms
- Note, we do not explicitly express $\mathbf{A}$ as we only concern $\mathbf{Au}$ on each iteration.

Implementation details

- Use GMRES iterative template routine developed by University of Tennessee and Oak Ridge National Laboratory in 1993
- int gmres_(long int *n, double *b, double *x, long int *restrt, double *work, long int *ldw, double *h, long int *ldh, long int *iter, double *resid, int (*matvec) (), int (*psolve) (), long int *info)
- readin.c: read protein files and call MSMS
- matvec.c: comp_source(), comp_pot(), matvecmul()
- main.c: readin(), comp_source(), gmres_(), comp_pot()
- $E_{\text{sol}} = \frac{1}{2}\sum_{k=1}^{N_c} q_k \phi_{\text{reac}}(\mathbf{x}_k) = \frac{1}{2}\sum_{k=1}^{N_c} q_k \int_{\Gamma} \left[K_1(\mathbf{x}_k, \mathbf{y}) \frac{\partial \phi_1(\mathbf{y})}{\partial v_{\mathbf{y}}} + K_2(\mathbf{x}_k, \mathbf{y}) \phi_1(\mathbf{y}) \right] dS_{\mathbf{y}}$ (12) where $\phi_{\text{reac}}(\mathbf{x}_k) = \phi_1(\mathbf{x}_k) - S_1(\mathbf{x}_k)$.

Sequential results

Table 3
Numerical results for computing the electrostatic solvation energy of 24 proteins: N is the number of elements, N_c is the number of atoms, N_{it} is the number of iterations, area is the molecular surface area, E_{sol} is the electrostatic solvation energy, T_1 is the running time on one CPU, and T_p is the running time with GPU acceleration.

ID	PDB	N	N_c	Area (Å^2)	N_{it}	E_{sol} (kcal/mol)	CPU T_1 (s)	GPU T_p (s)	T_1/T_p
1	1ajj	40 496	519	2176	8	−1145.76	1 323	11	125
23	1a63	132 134	2065	7003	11	−2404.07	19 804	139	143

Serial computing results on 1ajj and 1a63 for den=10								
den	protein	nspt	nface	natm	area	N_{it}	E	Serial
10	1ajj	20259	40514	519	2166.67	8	-1145.76	472.79
10	1a63	66100	132196	2065	6973.64	11	-2404.07	9605.85

Sequential results

Table 4
Case 2 (protein 1A63). TABI and APBS results; PB and Poisson equations; showing electrostatic solvation energy E_{sol}, and error, CPU time, memory usage; TABI columns show MSMS density, E_{sol} values computed by direct sum (*ds*) and treecode (*tc*); discretization error e^{ds}_{sol}, treecode approximation error e^{tc}_{sol}; treecode order $p = 3$, MAC parameter $\theta = 0.8$ (PB), $\theta = 0.5$ (P); APBS columns show maximum grid spacing h_{max}, grid dimensions N_g.

Density ($Å^{-2}$)		N^a	E_{sol} (kcal/mol)		Error (%)		CPU (s)		Iters[b]		Memory (MB)	
			ds	*tc*	e^{ds}_{sol}	e^{tc}_{sol}	*ds*	*tc*	*ds*	*tc*	*ds*	*tc*
TABI	nface	ltzmann		E				time				
1	20264	20,264	−2913.46	-2913.45	22.691	0.0260	535	476.24		25	10	23
2	30358	30,358	−2531.67	-2531.64	6.613	0.0385	1059	898.92		22	14	36
5	70018	70,018	−2440.05	-2440.19	2.755	0.0717	3371	2959.73		13	31	82
10	132196	132,196	−2404.07	-2404.07	1.239	0.0433	10,768	9605.85		11	57	149

Weihua Geng, Robert Krasny. (2013). “A treecode-accelerated boundary integral Poisson-Boltzmann solver for electrostatics of solvated biomolecules.” *Journal of Computational Physics, 247*(2), 62-78.

OpenMP

- #pragma omp parallel for outer loops iterating nface times: comp_source(), matvecmul(), comp_pot()

OMP results for 1ajj												
den	nface	E	Serial	N=1	N=2	N=4	N=8	N=16	N=32	N=64	N=128	N=256
1	6036	-1274.12	17.06	17.06	8.57	4.33	2.21	1.30	0.93	0.80	0.77	0.80
2	9204	-1183.33	35.63	35.61	17.90	9.01	4.58	2.51	1.86	1.54	1.48	1.53
5	21558	-1155.82	173.14	172.95	86.64	43.53	21.97	11.19	8.51	7.03	6.74	6.68
10	40514	-1145.76	610.26	609.45	305.37	153.20	77.29	39.22	27.92	24.08	23.41	23.22

OpenMP parallel speedup

OMP parallel speedup for 1ajj								
den/N	2	4	8	16	32	64	128	256
1	1.99	3.94	7.71	13.09	18.42	21.44	22.05	21.25
2	1.99	3.95	7.78	14.20	19.18	23.19	24.10	23.30
5	2.00	3.97	7.87	15.45	20.33	24.61	25.65	25.88
10	2.00	3.98	7.89	15.54	21.83	25.31	26.03	26.24

OpenMP parallel efficiency

OMP parallel efficiency for 1ajj								
den/N	2	4	8	16	32	64	128	256
1	99.51%	98.47%	96.33%	81.83%	57.56%	33.50%	17.23%	8.30%
2	99.47%	98.86%	97.30%	88.73%	59.95%	36.23%	18.83%	9.10%
5	99.81%	99.32%	98.42%	96.57%	63.55%	38.45%	20.04%	10.11%
10	99.79%	99.46%	98.56%	97.13%	68.22%	39.54%	20.33%	10.25%

MPI approach

- Root processor called readin()
- MPI_Bcast() to other processes
- Chunked the for loops in comp_source(), comp_pot(), matvecmul()
- MPI_Allgatherv() to share data between every processes for gmres_()

Table 1
Pseudocode for GABI-PB solver using GPU.

1	On host (CPU)
2	read biomolecule data (charge and structure)
3	call MSMS to generate triangulation
4	copy biomolecule data and triangulation to device
5	On device (GPU)
6	each thread concurrently computes and stores source terms for assigned triangles
7	copy source terms on device to host
8	On host
9	set initial guess $\mathbf{x}_0$ for GMRES iteration and copy it to device
10	On device
11	each thread concurrently computes assigned segment of matrix–vector product $\mathbf{y} = \mathbf{Ax}$
12	copy the computed matrix–vector $\mathbf{y}$ to host memory
13	On host
14	test for GMRES convergence
15	if no, generate new $\mathbf{x}$ and copy it to device, go to step 10 for the next iteration
16	if yes, generate and copy the final solution to device and go to step 17
17	On device
18	compute assigned segment of electrostatic solvation energy
19	copy results in step 18 to host
20	On host
21	add segments of electrostatic solvation energy and output result

MPI results

- MPI: salloc -p standard-mem-s -N8 -n256 --x11=first

MPI results for 1ajj										
den	Serial	N=1	N=2	N=4	N=8	N=16	N=32	N=64	N=128	N=256
1	13.16	13.65	6.99	3.60	2.02	1.07	0.59	0.35	0.24	0.52
2	27.37	28.38	14.74	7.31	3.98	2.18	1.16	0.78	0.75	1.08
5	132.72	125.34	64.12	32.38	17.88	10.17	5.22	2.99	1.77	1.13
10	472.79	441.62	226.14	114.56	62.38	35.36	18.11	9.76	5.41	3.22
20	1874.79	1798.12	924.94	464.44	253.66	143.83	72.64	36.71	18.77	13.41
30	4561.48	4647.39	2360.91	1198.33	634.91	335.47	169.73	85.97	44.45	23.34
40	7443.10	7622.13	3868.88	1959.97	1056.69	587.75	295.80	150.41	77.08	40.38

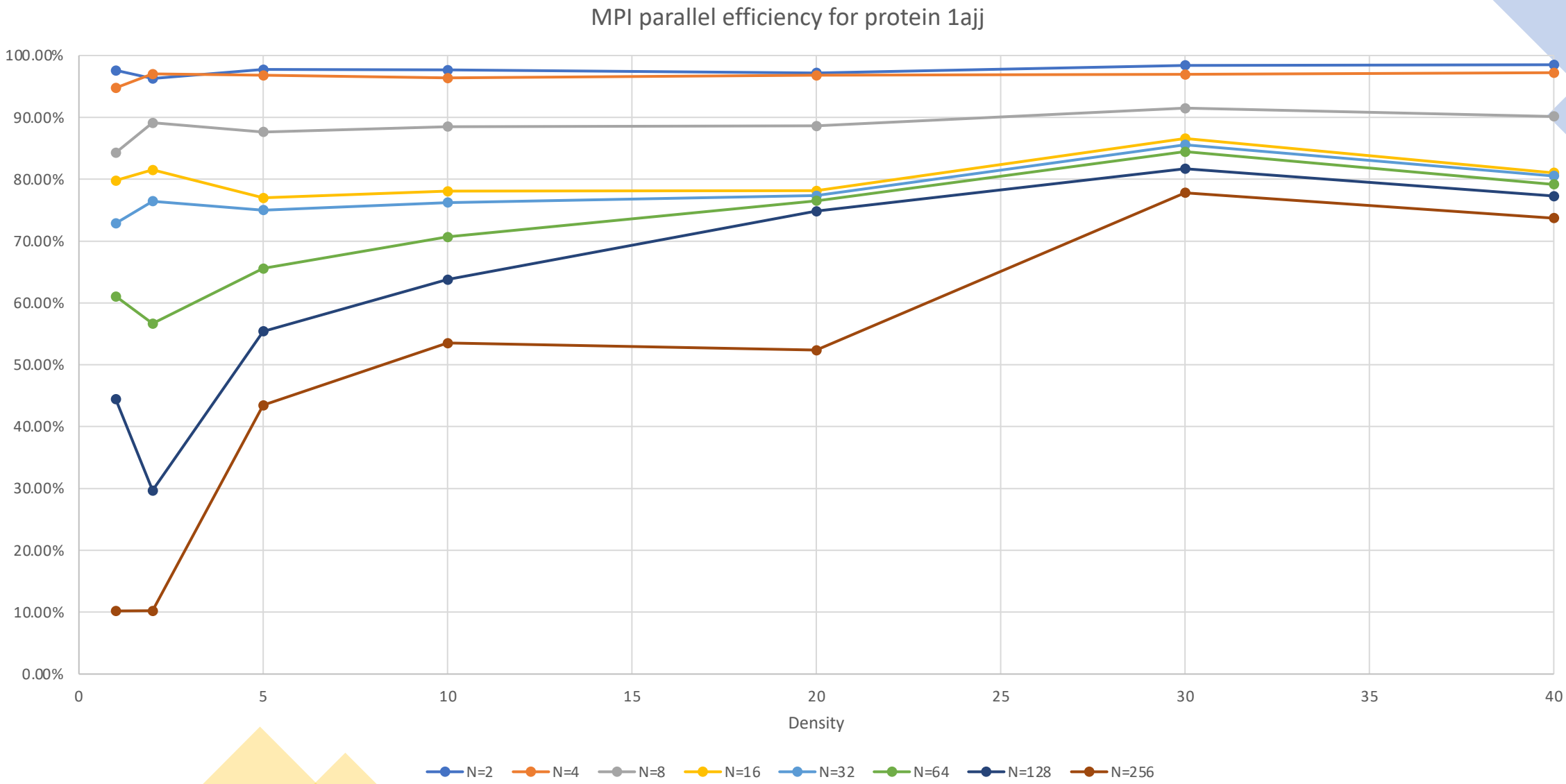
MPI parallel speedup

MPI parallel speedup for 1ajj								
den/N	2	4	8	16	32	64	128	256
1	1.9518	3.7908	6.7437	12.7646	23.3204	39.0586	56.8507	26.1061
2	1.9257	3.8807	7.1300	13.0395	24.4609	36.2681	38.0008	26.1846
5	1.9548	3.8712	7.0106	12.3195	24.0042	41.9540	70.9091	111.2529
10	1.9529	3.8548	7.0800	12.4900	24.3907	45.2309	81.6275	137.0021
20	1.9440	3.8715	7.0888	12.5020	24.7538	48.9791	95.7822	134.0448
30	1.9685	3.8782	7.3198	13.8535	27.3804	54.0560	104.5651	199.1590
40	1.9701	3.8889	7.2132	12.9684	25.7676	50.6749	98.8846	188.7496

MPI parallel efficiency

MPI parallel efficiency for 1ajj								
den/N	2	4	8	16	32	64	128	256
1	97.59%	94.77%	84.30%	79.78%	72.88%	61.03%	44.41%	10.20%
2	96.28%	97.02%	89.13%	81.50%	76.44%	56.67%	29.69%	10.23%
5	97.74%	96.78%	87.63%	77.00%	75.01%	65.55%	55.40%	43.46%
10	97.65%	96.37%	88.50%	78.06%	76.22%	70.67%	63.77%	53.52%
20	97.20%	96.79%	88.61%	78.14%	77.36%	76.53%	74.83%	52.36%
30	98.42%	96.96%	91.50%	86.58%	85.56%	84.46%	81.69%	77.80%
40	98.51%	97.22%	90.17%	81.05%	80.52%	79.18%	77.25%	73.73%


MPI parallel speedup for protein 1ajj
250.0000
200.0000
150.0000
100.0000
50.0000
0.0000
0
5
10
15
20
25
30
35
40
Density
N=2
N=4
N=8
N=16
N=32
N=64
N=128
N=256


MPI parallel efficiency for protein 1ajj
100.00%
90.00%
80.00%
70.00%
60.00%
50.00%
40.00%
30.00%
20.00%
10.00%
0.00%
0
5
10
15
20
25
30
35
40
Density
N=2
N=4
N=8
N=16
N=32
N=64
N=128
N=256

CUDA and Kokkos

- Similar to Lab12 (1), Kokkos::kokkos_malloc(), Kokkos::parallel_for()
- Compiled gmres into static library
- Changed .c files into .cpp files using extern “C”
- CUDA: srun -p v100x8 --gres=gpu:1 ./bimpb_cuda.exe
- Kokkos: srun -p development -c 4 --mem=16G --gres=gpu:volta:1 --pty $SHELL

CUDA and Kokkos results for 1ajj							
den	nspt	nface	E	Total area	Iteration N	Cuda	Kokkos
1	3020	6036	-1274.1225	2075.9982	10	0.2736	0.1195
2	4604	9204	-1183.3262	2127.7655	9	0.2993	0.1478
5	10781	21558	-1155.8154	2155.7153	8	0.4721	0.3100
10	20259	40514	-1145.7591	2166.6733	8	0.6945	0.5934
20	40909	81814	-1142.4895	2171.5429	8	1.8885	1.7797
30	62489	124974	-1141.3645	2173.2231	8	4.7058	4.6559
40	82846	165688	-1139.1506	2174.3510	8	7.3380	7.0144

CUDA and Kokkos results

CUDA and Kokkos results for 1a63							
den	nspt	nface	E	Total area	Iteration N	CUDA	Kokkos
1	10134	20264	-2913.4453	6697.2640	23	0.9242	0.5287
2	15181	30358	-2531.6355	6847.4139	20	1.0514	0.7690
5	35011	70018	-2440.1920	6938.2871	12	2.3392	2.0443
10	66100	132196	-2404.0728	6973.6437	11	6.8728	7.5230
20	132502	265000	-2390.1241	6989.4389	10	21.6191	18.6548
30	202019	404034	-2385.0149	6995.0457	10	40.5651	45.5954
40	268445	536886	-2382.2964	6997.3295	10	74.0344	72.4303

Conclusion

- Both OpenMP and MPI speeds up faster (with less efficiency) as the number of processors and the size of the problem increase
- Parallel efficiency of OpenMP is slightly better than its of MPI for N=2:16, after that, MPI efficiency is better than OpenMP
- CUDA and Kokkos have similar performance, but Kokkos might beat CUDA with better approach
- Overall, GPU parallelization is way faster than CPU parallelization