Raport: Badanie przemian fazowych w argonie

Heorhii Lopatin (456366)

12 lipca 2023

Wstęp

Celem tego zadania jest zbadanie zjawiska przejść fazowych argonu. W tym celu zaimplementowano program na CPU i GPU, symulujący nagrzewanie argonu za pomocą dynamiki molekularnej.

1 Badanie stabilności algorytmu CPU

Stabilność energii całkowitej

Wyznaczmy wartości odchylenia bezwzględnego całkowitej energii końcowej od początkowej, oraz odchylenia standardowego energii całkowitej, kinetycznej i potencjalnej.

Wartość kroku,ps	ΔE	σ^2_E	$\sigma^2_{E_p}$	$\sigma^2_{E_k}$
0.001	1.78	0.52	33.57	33.51
0.002	1.71	0.54	23.55	23.51
0.005	1.92	0.57	17.10	17.08
0.01	2.10	0.43	13.93	13.91
0.02	3.46	0.52	11.10	11.14
0.05	12.1	0.75	9.79	10.17

Promień odcięcia

Zbadamy także wpływ wprowadzenia promienia odcięcia na szybkość i dokladność obliczeń.

Promień,nm	Wartość kroku,ps	ΔE	Czas wykonania,s	Przyspieszenie
-	0.001	1.72	81.72	-
-	0.005	1.77	81.77	1.00
1.0	0.001	4.50	27.25	3.00
1.0	0.002	4.88	27.21	3.00
1.0	0.010	4.05	27.02	3.02
1.3	0.001	4.75	33.17	2.46
1.5	0.001	1.50	37.98	2.15
2.0	0.001	1.79	52.85	1.55

2 Przeniesienie algorytmu na GPU

2.1 Przeniesienie algorytmu w najprostszej wersji

Przenosimy na GPU obliczenia sił i energii, reszta symulacji odbywa się na CPU (kod nie zmienił się znacząco w porównaniu do przykładowego):

```
1
2  __global__ void calc_forces_v1(atoms_state *state, atom *atoms, struct block_results *help) {
3    real mfx, mfy, mfz, mx, my, mz;
4    real dx, dy, dz, df_by_r2, r2, r4, r6, r12, delj6, delj12, elj6, elj12, sig2, sig4, sig6, sig12;
5    int k;
6    int natoms = state->natoms;
7    int rc2 = state->cutoff * state->cutoff;
8    int i =  blockIdx.y*blockDim.y+threadIdx.y;
9
10   sig2 = sigma*sigma;
11   sig4 = sig2*sig2;
12   sig6 = sig2*sig4;
13   sig12 = sig6*sig6;
14   if(i < natoms) {
15     mx = atoms[i].x;
16     my = atoms[i].y;
17     mz = atoms[i].z;
18   } else return;
19   elj6 = elj12 = mfx = mfy = mfz =  0;
20   if(state->cutoff != 0) {
21     for (int j=0; j<natoms; j++) {
22       dx = mx - atoms[j].x;
23       dy = my - atoms[j].y;
24       dz = mz - atoms[j].z;
25       r2 = dx*dx + dy*dy + dz*dz;
26       if(j != i && r2 < rc2) {
27         r6 = r2*r2*r2;
28         r12 = r6*r6;
29         delj6 =  sig6/r6;
30         delj12 =  sig12/r12;
31         elj6 -=  delj6;
32         elj12 += delj12;
33         df_by_r2 = epsilon * 12 * (delj12 - delj6) / r2;
34         mfx += df_by_r2 * dx ;
35         mfy += df_by_r2 * dy ;
36         mfz += df_by_r2 * dz ;
37       }
38     }
39   }else {
40     for (int j=0; j<natoms; j++) {
41       dx = mx - atoms[j].x;
42       dy = my - atoms[j].y;
43       dz = mz - atoms[j].z;
44       r2 = dx*dx + dy*dy + dz*dz;
45       if(j != i) {
46         r6 = r2*r2*r2;
47         r12 = r6*r6;
48         delj6 =  sig6/r6;
49         delj12 =  sig12/r12;
50         elj6 -=  delj6;
51         elj12 += delj12;
```

```
        df_by_r2 = epsilon * 12 * (delj12 - delj6) / r2;
        mfx += df_by_r2 * dx ;
        mfy += df_by_r2 * dy ;
        mfz += df_by_r2 * dz ;
      }
    }
  }
  atoms[i].fx = mfx;
  atoms[i].fy = mfy;
  atoms[i].fz = mfz;
  atoms[i].elj_6 = epsilon * elj6;
  atoms[i].elj_12 = 0.5 * epsilon * elj12;
}
```

Sprawdzenie poprawności:

Wartość kroku,ps	ΔE	σ_E^2	Czas wykonania,s
0.001	0.12	9.56	52.69
0.002	0.27	13.64	53.01
0.005	0.06	9.98	52.66
0.010	0.13	9.49	52.71
0.020	0.94	11.03	52.79
0.050	2.66	10.51	52.76

2.2&2.3 Optymalizacja algorytmu

Zaimplementowano dwa kernela:

```

__global__ void calc_forces(atoms_state *state, atom *atoms, struct block_results *help) {
   if(blockIdx.x < blockIdx.y) return;

  __shared__ coords guests[DIM+1];
  __shared__ forces fguests[DIM+1];

    ... //variable declarations


  if(offset + threadIdx.y < natoms) {
    auto import = atoms[offset + threadIdx.y];
    guests[threadIdx.y].x = import.x;
    guests[threadIdx.y].y = import.y;
    guests[threadIdx.y].z = import.z;
    fguests[threadIdx.y].fx = 0;
    fguests[threadIdx.y].fy = 0;
    fguests[threadIdx.y].fz = 0;
  }

  __syncthreads();
  if(i < natoms) {mycords = atoms[i];}
  elj6 = elj12 = 0;

  if(state->cutoff != 0) {
    for (int j=0; j<DIM; j++) {
      k = (i+j) % DIM;
      dx = mycords.x - guests[k].x;
      dy = mycords.y - guests[k].y;
      dz = mycords.z - guests[k].z;
      r2 = dx*dx + dy*dy + dz*dz;
```

```
      if( k + offset < natoms && k + offset > i && (r2 < rc2 )) {
        r6 = r2*r2*r2;
        r12 = r6*r6;
        delj6 = sig6/r6;
        delj12 = sig12/r12;
        elj6 -= delj6;
        elj12 += delj12;
        df_by_r2 = (delj12 - delj6) / r2;
        myself.fx += df_by_r2 * dx ;
        myself.fy += df_by_r2 * dy ;
        myself.fz += df_by_r2 * dz ;
        fguests[k].fx -= df_by_r2 * dx ;
        fguests[k].fy -= df_by_r2 * dy ;
        fguests[k].fz -= df_by_r2 * dz ;
      }
    __syncthreads();
    }
  }else {
        ...
    }
  }

  if(threadIdx.y + offset < natoms) {
    block->guests[threadIdx.y]= fguests[threadIdx.y];
  }
  if(i < natoms ) {
        block->hosts[threadIdx.y] = myself;
        block->elj_6[threadIdx.y] = elj6;
        block->elj_12[threadIdx.y] = elj12;
      }

}
```

A także

```
 #define MAXDIM  1000
__global__ void sum_forces(atoms_state *state, atom *atoms, struct block_results *help) {
  __shared__ real  buffer6[MAXDIM];
  __shared__ real  buffer12[MAXDIM];

  int in_block = threadIdx.y;
  int y =  blockIdx.y;
  int natoms = state->natoms;
  int step = ((natoms - 1) / DIM + 1);
  int num_atom = in_block + blockIdx.y * blockDim.y;
  real sum = 0;

  switch(blockIdx.x) {
    case 0:
      for(int x = 0; x < step; x++) {
        if(y <= x) {
          sum += help[y * step + x].hosts[in_block].fx;
        }
        if(y >= x) {
          sum += help[x * step + y].guests[in_block].fx;
        }

```

```
      }
      if(num_atom < natoms) {
        atoms[num_atom].fx = 12 * epsilon * sum;
      }
    break;
    case 1:
    //same for fy
    case 2:
    //same for fz
    break;
    case 3:
    //same for elj6

    break;
    case 4:
    //same for elj12
  }
}


```

Podane powyżej kernele wywoluje się następująco:

```
 void simulation_state::forces_gpu() {
    cudaError_t status;
    dim3 threads(1,DIM,1);
    dim3 blocks((this->natoms-1)/DIM+1,(this->natoms-1)/DIM+1,1);
    clock_t tic, tac, toc;
    tic = clock();

    calc_forces<<<blocks, threads>>>(this->gpu_atoms->astate, \
            this->gpu_atoms->atoms, this->gpu_atoms->help);


    status = cudaDeviceSynchronize();
    if (status != cudaSuccess){    ERROR( cudaGetErrorString(status));}
    tac = clock();
    DEBUG_PRINTF("calc_forces took %.6fms\n", ((double)(tac - tic) * 1000 / CLOCKS_PER_SEC));

    blocks = dim3(5, (this->natoms-1)/DIM+1);

    sum_forces<<<blocks, threads>>>(this->gpu_atoms->astate,\
            this->gpu_atoms->atoms, this->gpu_atoms->help);

      status = cudaDeviceSynchronize();
    if (status != cudaSuccess){    ERROR( cudaGetErrorString(status));}

    toc = clock();
    DEBUG_PRINTF("sum_forces took %.6fms\n", ((double)(toc - tac) * 1000 / CLOCKS_PER_SEC));

}


```

Jak widać, uwzględniono (prawie) wszystkie optymalizacje podane w zadaniu:

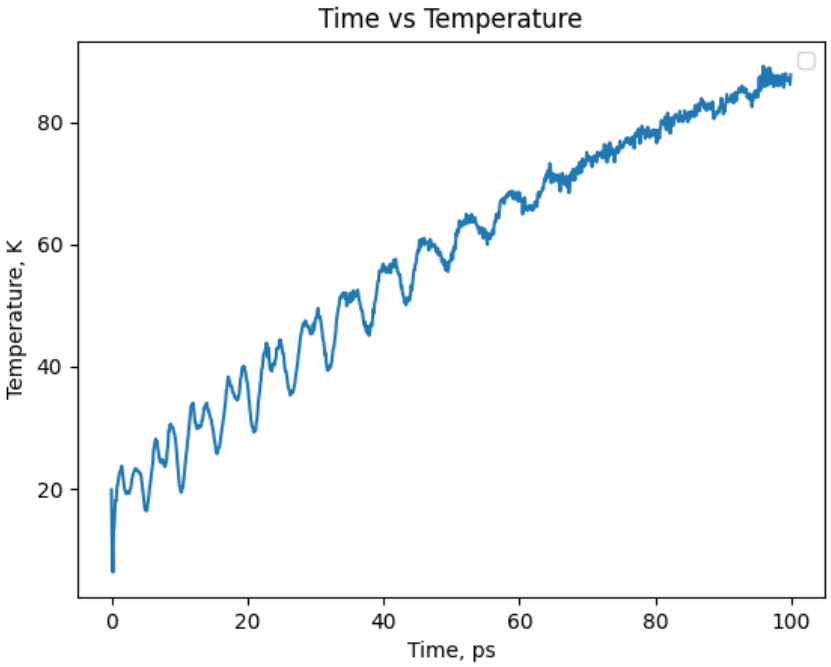
- liczenie oddziaływań dla każdego atomu jest podzielone na wiele bloków
- zoptymalizowano dostęp do współrzędnych atomów gości prze użycie pamięci współdzielonej
- oddziaływanie dla każdej pary atomów jest liczone tylko raz

3 Badanie przemian fazowych w argonie

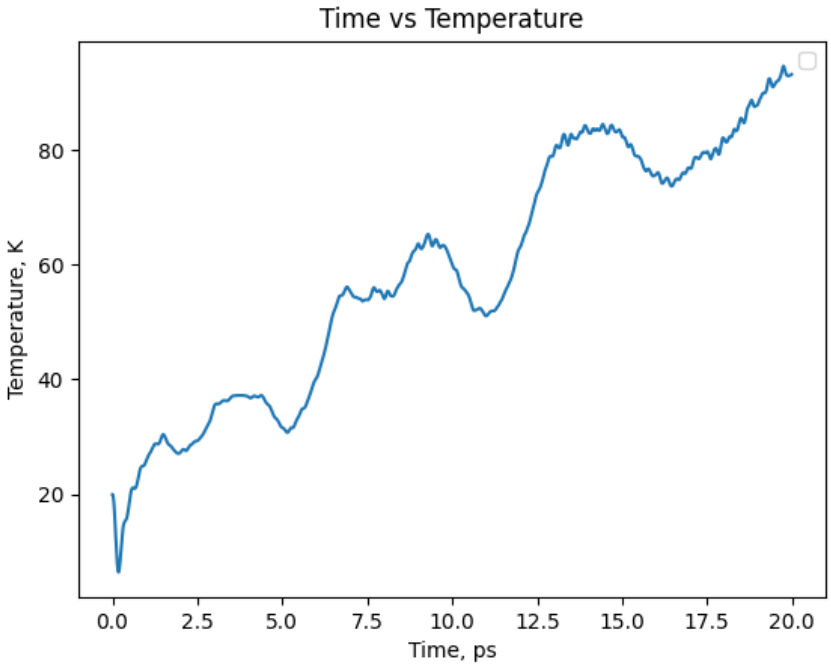
Sprawdźmy stabilność algorytmy dla 10,000 kroków:

Wartość kroku,ps	ΔE	σ_E^2	Czas wykonania,s
0.001	0.15	7.02	15.02
0.002	0.01	8.39	15.28
0.005	0.41	4.29	15.10
0.010	1.76	5.23	14.89
0.020	0.42	4.00	15.26
0.050	3.61	8.02	15.20

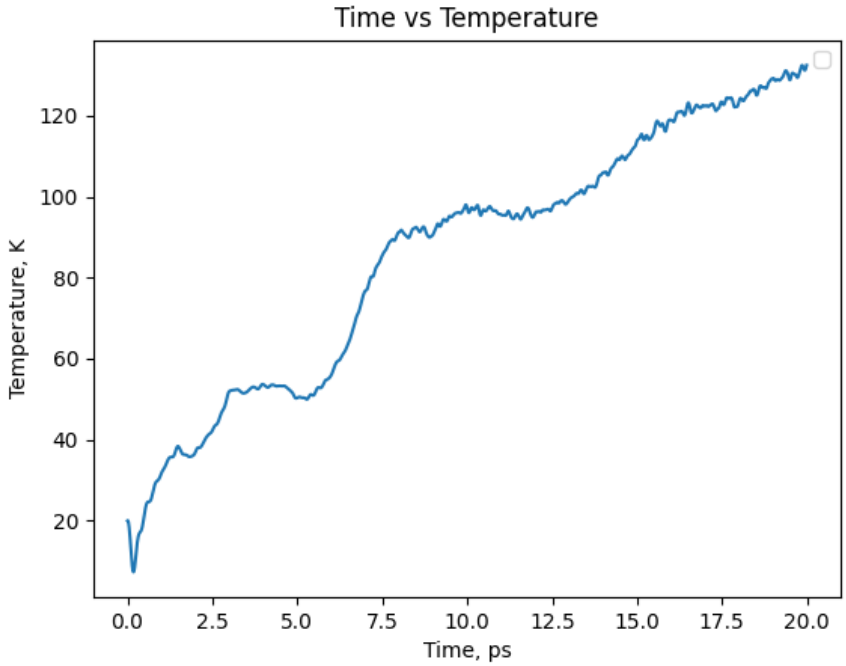
Przeprowadzamy symulację ogrzewania od 20 do 120 K:



Rysunek 1: krok czasowy 0.001 ps, liczbia kroków równa 100,000



Rysunek 2: krok czasowy 0.001 ps, liczbia kroków równa 20,000



Rysunek 3: krok czasowy 0.001 ps, liczbia kroków równa 20,000, delta 200 K

Podsumowanie

Wydajność

Trzecia wersja GPU jest o > 3 razy szybsza niż pierwsza. Tak jest rzeczywiście przez to, że liczba instrukcji na każdy wątek jest równa DIM * k zamiast Natoms * k. Przydało się także połączenie wyników dla gości i gospodarzy oraz użycie pamięci współdzielonej.

Przejścia fazowe

Jak widać na rysunkach, gdy temperatura osiąga 80-85 K, argon staje się bardziej chaotyczny, a wahania temperatury zmniejszają się. Z tego możemy wywnioskować, że argon staje się gazem.