Sprawozdanie zadania Argon

Bartosz Kucypera, bk439964

20 września 2023

1 Wstęp

Zadanie składało się z trzech części. Pierwsza część polegała na symulacji klastra atomów argonu metodą dynamiki molekularne wykożystując CPU. W części drugiej należało poprawić wydajność rozwiązania, przenosząc część obliczeń na GPU jak i stosując szereg innych optymalizacji. W ostatniej, trzeciej części należało zbadać przeminay fazowe argonu.

2 Implementacja części wspólnej

Poszczególne wersje rozwiązania różnią się tylko obliczaniem sił działających na atomy, poniżej opiszę część roziwązania która jest wspólna dla wszystkich wersji.

2.1 Konfiguracja

Konfiguracja symulacji odbywa się poprzez stałe globlane.

```
/* stałe z treści */
constexpr real_t sig =  0.369;      /* nm */
constexpr real_t eps =  1.19;       /* kJ/mol */
constexpr real_t a   =  0.260922;   /* sig = sqrt(2)*a */
constexpr real_t ts  =  0.001;      /* time stamp w symulcji */
constexpr real_t B   =  1;          /* siła balonu */
constexpr real_t mas = 39.95;       /* masa atomu */
constexpr real_t kB  = 0.00831;     /* stała Boltzmana */

/* wymiary siatki z atomami */
constexpr int siat_X = 6;
constexpr int siat_Y = 6;
constexpr int siat_Z = 6;
constexpr int N = siat_X*siat_Y*siat_Z*4; /*ilość atomów */

/* wymagana konfiguracja */
constexpr int DEBUG         = 1;        /* 0-nic, 1-najważniesze, 2-rozszerzone, 3-wszystko */
constexpr bool VERBOSE      = true;     /* czy wypisywać najważniejsze logi również na stdout */
constexpr bool STATS        = false;    /* czy obliczac statystyki energi */
constexpr int Nterm         = 10000;    /* ilość kroków fazy termalizacji */
constexpr int Ngrz          = 5000;     /* ilość kroków fazy grzania */
constexpr int Nch           = 5000;     /* ilość kroków fazy chłodzenia */
constexpr int ENERG         = 100;      /* częstotliowść zapisu anergi */
constexpr int TRAJE         = 100;      /* częstotliwość zapisu współrzędnych */
constexpr real_t rB         = 20;       /* r balonu w nm */
constexpr real_t rOd        = 1.3;      /* promień odcięcia */
constexpr real_t rBuff      = 0.3;      /* bufor */
constexpr real_t start_T    = 70;       /* temperatura początkowa */
constexpr real_t cool_T     = 20;       /* temperatura do której chłodzimy */
constexpr real_t heat_T     = 120;      /* temperatura do której ogrzewamy */

/* max długość listy sąsiadów */
constexpr int MAX_LI = def_min(N, max_zageszczenie(rOd + rBuff));
```

2.2 Stan symulacji

Informacje o stanie symulacji przechowywane sa w nastepujacych tablicach globalnych.

```
/* tablice na hosice */
real_t cords[3*N];      /* gdzie jest atom */
real_t V[3*N];          /* prędkości atomu względem osi */
real_t F[3*N];          /* siła względem osi */
real_t Epot[2*N];       /* energia potencjalna Lennarda Jonesa, [0] - r12, [1] - r6 */
real_t Ebal[N];         /* enrgia potencjalna - człon balonu */
real_t Ekin[N];         /* energia kinetyczna atomu */

int G[N][MAX_LI];       /* lista sąsiedztwa atomów */
real_t poz0[N][3];      /* pozycja atomu w momencie budowy listy sąsiedztwa */

/* gpu I i gpu II */
real_t* F_gpu;          /* siła, gpu */
real_t* Epot_gpu;       /* energia potencjalna, gpu */
real_t* cords_gpu;      /* kordynaty, gpu */

/* wersja gpu 3, podział na atomy w roli hosta i goscia */
real_t* F_gpu_host;
real_t* F_gpu_guest;
real_t* Epot_gpu_host;
real_t* Epot_gpu_guest;

/* wersja gpu 4, listy sasiedztwa */
int* G_gpu;
real_t poz0_cpu[N*3];
```

2.3 Krok symulacji

Do integracji równań ruchu używamy algorytmu leap-frog w postaci kick-drift-kick.

$$v_{i+1/2} = v_i + a_i + \frac{\Delta t}{2},$$

$$x_{i+1} = x_i + v_{i+1/2} \cdot \Delta t,$$

$$v_{i+1} = v_{i+1/2} + a_{i+1} \cdot \frac{\Delta t}{2}$$

Pętla główna symulacji wygląda następująco.

```
init_atomow();

for (int nr = 0; nr < Nterm+Ngrz+Nch; nr++) {
    up_V();
    up_cords();

    up_forces(config);
    up_ballon();

    up_V();

    up_kin();
    up_temp(nr);

    up_logi(nr);
    up_stats(nr);
}
up_logi(0);
stats_out();
```

Po kolei:

(up_V)	Aktualizujemy predkość (krok pierwszy leap-frog).
(up_cords)	Aktualizujemy położenie atomow (krok drugi leap-frog).
(up_forces)	Aktualizujemy siły działajace na atomy (sposobem zależnym od config).
(up_ballon)	Aktualizujemy człon siły balonu.
(up_V)	Aktualizujemy predkość (krok trzeci leap-frog).
(up_kin)	Aktualizujemy energie kinetyczna.
(up_temp)	Skalujemy predkość zeby wpłynac na temperaturę.
(up_logi)	Aktualizujemy pliki logow.
(up_stats)	Aktualizujemy statystyki (badanie poprawności przy promieniu odcięcia).

2.4 Aktualizacja predkości

Podstawiamy $a = \frac{f}{m}$ do leap-frog, (ts to Δt).

```
inline void up_V() {
    for (int i = 0; i < N; i++)
        for (int k = 0; k < 3; k++)
            V[i*3 + k] += F[i*3 + k]/mas*ts/2.0;
}
```

2.5 Aktualizacja położenia atomow

Wzór na drogę w ruchu jednostajnym prostoliniowym $s = v \cdot t$.

```
inline void up_cords() {
    for (int i = 0; i < N; i++)
        for (int k = 0; k < 3; k++)
            cords[i*3 + k] += V[i*3 + k]*ts;
}
```

2.6 Aktualizacja energi kinetycznej

Korzystamy ze wzoru $E_k = \frac{1}{2}m|v|^2$.

```
inline void up_kin() {
    for (size_t i = 0; i < N; i++)
        Ekin[i] = kinetyczna(i);
}
```

```
inline real_t kinetyczna(int i) {
    real_t res = 0;
    for (int k = 0; k < 3; k++)
        res += sq(V[i*3 + k]);
    return res*mas/2.0;
}
```

Gdzie *sq* to podniesienie do kwadratu.

2.7 Uwzględnienie działania balonu

Aktualizujemy wektory sił działajacych na atomy, oraz człon energi wynikajacej z działania balonu.
B to siła sprężystości balonu, rB to promień balonu.

```
inline void up_ballon() {
    for (int i = 0; i < N; i++) {
        real_t r = 0;
        for (int k = 0; k < 3; k++)
            r += sq(cords[i*3 + k]);
        r = sqrt(r);

        if (r <= rB)
            return;

        real_t mno = B*(r-rB)/r;
        for (int k = 0; k < 3; k++)
            F[i*3 + k] -= mno*cords[i*3 + k];
        Ebal[i] += B*sq(r-rB)/2.0;
    }
}
```

2.8 Zmiana temperatury

Temperaturę zmieniamy skalujac predkości atomow.

```
inline void up_temp(int nr) {
    if (nr < Nterm)
        return;

    real_t mno;

    /* chłodzenie */
    if (nr >= Nterm and nr < Nterm+Nch) {
        mno = scale_T(cur_temperatura(), cool_T, Nch-nr+Nterm);
    }
    /* grzanie */
    else {
        mno = scale_T(cur_temperatura(), heat_T, Ngrz-nr+Nterm+Nch);
    }

    for (int i = 0; i < N; i++)
        for (int k = 0; k < 3; k++)
            V[i*3 + k] *= mno;
}
```

Gdzie *scale_T* oblicza przez co trzeba przemnażać predkości atomow by z danej temperatury początkowej dojść do danej temperatury końcowej w danej liczbie kroków,

```
constexpr inline real_t scale_T(real_t pocz_T, real_t konc_T, real_t kroki = 1) {
    return sqrt(1.0 + (konc_T-pocz_T)/pocz_T/kroki);
}
```

cur_temperatura oblicza temperaturę układu,

```
inline real_t cur_temperatura() {
    real_t sum = 0;
    for (size_t i = 0; i < N; i++)
        sum += Ekin[i];
    return temperatura(sum);
}
```

temperatura przyjmuje całkowitą energię kinetyczną układu i zwraca średnią temperaturę atomu.

```
inline constexpr real_t temperatura(real_t kin) { return kin/N*2.0/3.0/kB; }
```

3 Badanie poprawnosci

Wszystkie pomiary były wykonywane na symulacji o 10'000 krokach fazy termalizacji (0 fazy chłodzenia i 0 fazy ogrzewania).

3.1 Poprawność wersji bez promienia odcięcia

$\Delta t = 0.001ps$			
rodzaj energi	średnia	wariancja	odchylenie std.
kinetyczna	415.353793	13295.608949	115.306587
potencjalna	-6565.019095	13296.003279	115.308297
całkowita	-6149.665302	0.000005	0.002140
Bezwzględna różnica całkowitej energi początkowej i końcowej = 0.003746			

$\Delta t = 0.002ps$			
rodzaj energi	średnia	wariancja	odchylenie std.
kinetyczna	414.784172	6998.312010	83.655914
potencjalna	-6564.438732	6999.164543	83.661010
całkowita	-6149.654560	0.000041	0.006431
Bezwzględna różnica całkowitej energi początkowej i końcowej = 0.013607			

$\Delta t = 0.005ps$			
rodzaj energi	średnia	wariancja	odchylenie std.
kinetyczna	413.626971	3395.617382	58.271926
potencjalna	-6563.206356	3398.238114	58.294409
całkowita	-6149.579385	0.000851	0.029179
Bezwzględna różnica całkowitej energi początkowej i końcowej = 0.089974			

$\Delta t = 0.010ps$			
rodzaj energi	średnia	wariancja	odchylenie std.
kinetyczna	414.513856	2131.072219	46.163538
potencjalna	-6563.825136	2137.813790	46.236498
całkowita	-6149.311280	0.009664	0.098306
Bezwzględna różnica całkowitej energi początkowej i końcowej = 0.341988			

$\Delta t = 0.020ps$			
rodzaj energi	średnia	wariancja	odchylenie std.
kinetyczna	415.123586	1482.735668	38.506307
potencjalna	-6563.357016	1502.039471	38.756154
całkowita	-6148.233430	0.119225	0.345290
Bezwzględna różnica całkowitej energi początkowej i końcowej = 1.425067			

$\Delta t = 0.050ps$			
rodzaj energi	średnia	wariancja	odchylenie std.
kinetyczna	410.270381	1051.504381	32.426908
potencjalna	-6548.686890	1133.001339	33.660085
całkowita	-6138.416509	8.139457	2.852973
Bezwzględna różnica całkowitej energi początkowej i końcowej = 9.441853			

Symulacja spełnia warunek poprawności, energia całkowita jest prawie stała.
Gdy zwiekszamy Δt dokładność maleje, ale nawet dla $\Delta t = 0.050ps$ błąd jest mały.

3.2 Wpływ promienia odciecia na dokładność obliczeń

Wszystkie pomiary wykonane z $\Delta t = 0.001ps$, oraz rozmiarem bufora $0.2nm$.

promień odcięcia = $1nm$			
rodzaj energi	średnia	wariancja	odchylenie std.
kinetyczna	405.969270	12883.481542	113.505425
potencjalna	-6434.775838	13911.779320	117.948206
całkowita	-6028.806568	45.052419	6.712110
Bezwzględna różnica całkowitej energi początkowej i końcowej = 6.537997			

promień odcięcia = $1.3nm$			
rodzaj energi	średnia	wariancja	odchylenie std.
kinetyczna	411.735610	13157.873468	114.707774
potencjalna	-6515.878892	13614.238444	116.680069
całkowita	-6104.143282	40.556854	6.368426
Bezwzględna różnica całkowitej energi początkowej i końcowej = 3.108414			

promień odcięcia = $1.5nm$			
rodzaj energi	średnia	wariancja	odchylenie std.
kinetyczna	413.685796	13313.460769	115.383971
potencjalna	-6539.407757	13455.558777	115.998098
całkowita	-6125.721961	4.910431	2.215949
Bezwzględna różnica całkowitej energi początkowej i końcowej = 1.427013			

brak promienia odcięcia			
rodzaj energi	średnia	wariancja	odchylenie std.
kinetyczna	415.353793	13295.608949	115.306587
potencjalna	-6565.019095	13296.003279	115.308297
całkowita	-6149.665302	0.000005	0.002140
Bezwzględna różnica całkowitej energi początkowej i końcowej = 0.003746			

Zgodnie z oczekiwaniami, im mniejszy promień odcięcia tym mniej dokładne obliczenia.
Błąd jest jednak stosunkowo mały.

4 Liczenie sił działajacych na atomy i badanie wydajności

W tej części przedstawię po kolei coraz lepsze wersje symulacji i zbadam ich wydajność.

Szczególnie w wersjach w całości opartych na CPU bardzo ważne jest używanie flag optymalizacyjnych. Najlepsza algorytmicznie wersja CPU (CPU III) bez flagi -O3 jest wolniejsza od najgorszej algorytmicznie wersji (CPU I) z tą flagą.

CPU I z -O3 : 42.254 sekund.

CPU I bez -O3 : 175.996 sekund.

CPU III z -O3 : 18.735 sekund.

CPU III bez -O3 : 54.859 sekund.

W wersjach cpu korzystamy z następujących funkcji pomocniczych.

```
inline static void clear_cpu() {
    for (int i = 0; i < N; i++) {
        for (int k = 0; k < 3; k++)
            F[i*3 + k] = 0;
        Epot[i*2] = 0;
        Epot[i*2 + 1] = 0;
    }
}

inline static void post_up_cpu() {
    for (int i = 0; i < N; i++) {
        for (int k = 0; k < 3; k++)
            F[i*3 + k] *= 12.0*eps;

        Epot[i*2] *= eps/2.0;
        Epot[i*2 + 1] *= eps;
    }
}
```

clear_cpu czyści tablice sił i energi potencjalnej.
post_up_cpu domnaża do sił i energi stałe (robimy to na samym końcu by zminimalizować ilość operacji).

4.1 CPU I

Pierwsza, najprostsza wersja cpu. Dla każdego atomu iterujemy sie po wszystkich innych atomach i wyliczamy dzialajace na niego sily.

```
void up_forces_cpu_1() {
    clear_cpu();

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (i == j)
                continue;

            real_t rij[3];
            for (int k = 0; k < 3; k++)
                rij[k] = cords[i*3 + k] - cords[j*3 + k];

            real_t dl_rij = 0;
            for (int k = 0; k < 3; k++)
                dl_rij += sq(rij[k]);
            if (rOd > 0.0 and dl_rij > sq(rOd))
                continue;

            real_t sig2 = sig*sig/dl_rij;
            real_t sig6 = sig2*sig2*sig2;
            real_t sig12 = sig6*sig6;

            real_t dif = (sig12 - sig6)/dl_rij;

            for (int k = 0; k < 3; k++)
                F[i*3 + k] += dif*rij[k];

            Epot[i*2] += sig12;
            Epot[i*2 + 1] -= sig6;
        }
    }

    post_up_cpu();
}
```

Przeprowadzenie symulacji w tej wersji zajęło 157.92 sekund.

4.2 CPU II

W tej wersji zmiejszymy około o połowę ilość obliczeń, kożystając z III zasady dynamiki Newtona.
Dla każdej pary atomów tylko raz wyliczamy siły z jakimi na siebie działają.

```
void up_forces_cpu_2() {
    clear_cpu();

    for (int i = 0; i < N; i++)
        for (int j = i+1; j < N; j++) {
            real_t rij[3];
            for (int k = 0; k < 3; k++)
                rij[k] = cords[i*3 + k] - cords[j*3 + k];

            real_t dl_rij = 0;
            for (int k = 0; k < 3; k++)
                dl_rij += sq(rij[k]);

            if (rOd > 0.0 and dl_rij > sq(rOd))
                continue;

            real_t sig2 = sig*sig/dl_rij;
            real_t sig6 = sig2*sig2*sig2;
            real_t sig12 = sig6*sig6;

            real_t dif = (sig12 - sig6)/dl_rij;

            for (int k = 0; k < 3; k++) {
                F[i*3 + k] += dif*rij[k];
                F[j*3 + k] -= dif*rij[k];
            }

            Epot[i*2] += sig12;
            Epot[i*2 + 1] -= sig6;
            Epot[j*2] += sig12;
            Epot[j*2 + 1] -= sig6;
        }

    post_up_cpu();
}
```

(na czerwono) Iterujemy się tylko po atomach z większym numerem.
(na zielono) Aktualizujemy siły i energie potencjalną również dla atomu gościa.

Zgodnie z oczekiwaniami, czas wykonania zmniejszył się o około połowę i wyniósł 78.46 sekund.

4.3 CPU III

W tej wersji wprowadzamy listę sąsiadów. Każdy atom pamięta listę jedynie tych atomów które mają szansę wejść z nim w interakcję (jeśli atomy znajdują się od siebie dalej niż promień odcieńcia zakładamy, że nie działają na siebie żadną siłą).

Atomy cały czas się przemieszczają, musimy więc aktualizować listę sąsiedztwa jeśli jakieś dwa atomy przemieszczą się na tyle, że będą miały szansę stać się sąsiadami. By nie przebudowywać listy sąsiedztwa zbyt często wprowadzamy dodatkowo bufor i zakładamy, że atomy są sąsiadami jeśli ich odległość jest $\leq$ promień odcięcia + bufor.

Przebudowa następuje jeśli jakiś atom oddali się od swojej początkowej pozycji o przynajmniej połowę buforu.

Dla dwóch atomów sąsiadów zapisujemy tylko atom o wyższym numerze na liście sąsiedztwa atomu o niższym numerze. Przy wyliczaniu sił na nie działających i tak skorzystamy z III zasady dynamiki Newtona, by zmniejszyć ilość obliczeń.

Wykorzystujemy następujące tablice globalne na listę sąsiadów i początkowe pozycje atomów.

```
int G[N][MAX_LI];       /* lista sąsiedztwa atomów */
real_t poz0[N][3];      /* pozycja atomu w momencie budowy listy sąsiedztwa */
```

Kożystamy z następującej funkcji do sprawdzenia czy należy przebudować listę sąsiedztwa.

```
inline void check_G() {
    static bool first_builded = false;
    if (first_builded == false) {
        rebuild_G_cpu();
        first_builded = true;
        return;
    }

    for (int i = 0; i < N; i++) {
        real_t dl = 0;
        for (int k = 0; k < 3; k++)
            dl += sq(poz0[i][k]-cords[i*3 + k]);
        if (dl*4.0 >= rBuff*rBuff) {
            rebuild_G_cpu();
            return;
        }
    }
}
```

(na zielono) Sprawdzamy czy to nie pierwszy raz kiedy budujemy listę sąsiedztwa.

(na czerwono) Dla każdego atomu sprawdzamy czy nie oddalił się za bardzo od pozycji początkowej. Jeśli tak to przebudowujemy listę sąsiedztwa i kończymy.

Kożystamy z następującej funkcji do budowy listy sąsiedztwa.

```
void rebuild_G_cpu() {
    int pom[N];
    fill(pom, pom+N, 0);

    for (int i = 0; i < N; i++) {
        for (int j = i+1; j < N; j++) {
            real_t r = 0;
            for (int k = 0; k < 3; k++)
                r += sq(cords[i*3 + k]-cords[j*3 + k]);
            if (r <= sq(rOd + rBuff))
                G[i][pom[i]++] = j;
        }
    }

    for (int i = 0; i < N; i++) {
        G[i][pom[i]] = -1;
        assert(pom[i] < MAX_LI);
        for (int k = 0; k < 3; k++)
            poz0[i][k] = cords[i*3 + k];
    }
}
```

(na zielono) Dla każdego atomu iterujemy się po atomach o wyższym numerze i sprawdzamy czy są sąsiadami.

(na czerwono) Zaznaczamy końce list sąsiedztwa, upewniamy się czy nie przepełniliśmy listy (assert) i zapisujemy pozycję atomu.

Funkcja licząca siły działające na atomy jest prawie identyczna co poprzednia. Jedyna różnica to zmiana wewnętrznej pętli, by iterowała się po liście sąsiedztwa.

```
void up_forces_cpu_3() {
    check_G();
    clear_cpu();

    for (int i = 0; i < N; i++)
        for (int pm = 0, j = G[i][pm]; G[i][pm] != -1; j = G[i][++pm]) {
            real_t rij[3];
            for (int k = 0; k < 3; k++)
                rij[k] = cords[i*3 + k] - cords[j*3 + k];

            real_t dl_rij = 0;
            for (int k = 0; k < 3; k++)
                dl_rij += sq(rij[k]);

            if (rOd > 0.0 and dl_rij > sq(rOd))
                continue;

            real_t sig2 = sig*sig/dl_rij;
            real_t sig6 = sig2*sig2*sig2;
            real_t sig12 = sig6*sig6;

            real_t dif = (sig12 - sig6)/dl_rij;

            for (int k = 0; k < 3; k++) {
                F[i*3 + k] += dif*rij[k];
                F[j*3 + k] -= dif*rij[k];
            }

            Epot[i*2] += sig12;
            Epot[i*2 + 1] -= sig6;
            Epot[j*2] += sig12;
            Epot[j*2 + 1] -= sig6;
        }

    post_up_cpu();
}
```

Czas przeprowadzenia symulacji zmniejszył się do 18.74 sekund.

4.4 GPU I

Będziemy korzystać z zaokrąglenia N w góre do pierwszej wielokrorności 32 (w późniejszych wersjach).

```
/* zaokraglenie N w gore do pierwszej wielokrotnosci 32 */
constexpr int Nr32 = (N+31-(N+31)%32);
```

Przenosimy najprostrzą wersję CPU na GPU. Zewnętrzną pętlę zastępujemy numerem wątka.

```
__global__ void up_forces_gpu_1(real_t* cords_gpu, real_t* F_gpu, real_t* Epot_gpu) {
    int i = blockDim.x*blockIdx.x + threadIdx.x;

    if (i >= N)
        return;

    for (int k = 0; k < 3; k++)
        F_gpu[i*3 + k] = 0;
    Epot_gpu[i*2] = 0;
    Epot_gpu[i*2 + 1] = 0;

    for (size_t j = 0; j < N; j++) {
        if (i == j)
            continue;

        real_t rij[3];
        for (int k = 0; k < 3; k++)
            rij[k] = cords_gpu[i*3 + k] - cords_gpu[j*3 + k];

        real_t dl_rij = 0;
        for (int k = 0; k < 3; k++)
            dl_rij += sq(rij[k]);

        if (rOd > 0.0 and dl_rij > sq(rOd))
            continue;

        real_t sig2 = sig*sig/dl_rij;
        real_t sig6 = sig2*sig2*sig2;
        real_t sig12 = sig6*sig6;

        real_t dif = (sig12 - sig6)/dl_rij;

        for (int k = 0; k < 3; k++)
            F_gpu[i*3 + k] += dif*rij[k];

        Epot_gpu[i*2] += sig12;
        Epot_gpu[i*2 + 1] -= sig6;
    }

    for (int k = 0; k < 3; k++)
        F_gpu[i*3 + k] *= 12.0*eps;
    Epot_gpu[i*2] *= eps/2.0;
    Epot_gpu[i*2 + 1] *= eps;
}
```

(na niebiesko)	Na podstawie numeru bloku i wątku wyliczamy numer atomu.
(na czerwono)	Czyścimy tablice sił i energi potencjalnej.
(na zielono)	Wewnętrzna pętla wersji CPU I.
(na różowo)	Domnażamy brakujące stałe.

Udaje nam się trochę poprawić wydajność. Policzenie symulacji zajmuje nam 14.920 sekund.

4.5 GPU II

W tej wersji powiększamy równoległość przez zwiększenie liczby bloków. Każdy wątek policzy oddziaływania pomiędzy jednym atomem gospodarzem i K atomami gośćmi (K - rozmiar bloku). K ustawiamy na 32 (rozmiar warpa) dzięki czemu możemy dodatkowo skorzystać z instrukcji shfl. Zminimalizujemy dzięki temu odwołania do pamięci globalnej.

```
__global__ void up_forces_gpu_2(real_t* cords, real_t* F_gpu, real_t* Epot_gpu) {
    int x = blockIdx.x*blockDim.x + threadIdx.x;
    int y = blockIdx.y*blockDim.x + threadIdx.x;

    real_t C_host[3], C_guest[3];
    for (int k = 0; k < 3; k++) {
        C_host[k] = cords[x*3 + k];
        C_guest[k] = cords[y*3 + k];
    }

    int offset = x + blockIdx.y*Nr32;

    real_t F[3] = {0, 0, 0};
    real_t Epot[2] = {0, 0};

    for (int l = 0; l < 32; l++) {
        real_t rij[3];
        int j = blockIdx.y*32 + l;

        for (int k = 0; k < 3; k++)
            rij[k] = C_host[k] - __shfl_sync(static_cast<unsigned>(-1), C_guest[k], l);

        if (x == j or x >= N or j >= N)
            continue;

        real_t dl_rij = 0;
        for (int k = 0; k < 3; k++)
            dl_rij += sq(rij[k]);

        if (r0d > 0.0 and dl_rij > sq(r0d))
            continue;

        real_t sig2 = sig*sig/dl_rij;
        real_t sig6 = sig2*sig2*sig2;
        real_t sig12 = sig6*sig6;

        real_t dif = (sig12 - sig6)/dl_rij;

        for (int k = 0; k < 3; k++)
            F[k] += dif*rij[k];

        Epot[0] += sig12;
        Epot[1] -= sig6;
    }

    for (int k = 0; k < 3; k++)
        F_gpu[offset*3 + k] = F[k];
    Epot_gpu[offset*2] = Epot[0];
    Epot_gpu[offset*2 + 1] = Epot[1];
}
```

(na czerwono)	Kopiujemy do rejestrów współrzędne atomu gospodarza (dla siebie) i jednego atomu gościa (dla siebie i dla innych wątków z naszego warpa).
(na niebiesko)	Liczymy odziaływania pomiędzy atomem gospodarzem i 32-dwoma atomami gośćmi.
(na różowo)	Obliczoną przez nas część odziaływań kopiujemy do pamięci globalnej.

Oddziaływania dla każdego atomu liczymy w porcjach z kolejnymi 32-dwoma atomami gośćmi, dlatego na końcu musimy jeszcze zsumować wyniki.

```
__global__ void summer_gpu_2(real_t* F_gpu, real_t* Epot_gpu) {
    int x = blockIdx.x*blockDim.x + threadIdx.x;

    for (int y = 1; y < Nr32/32; y++) {
        for (int k = 0; k < 3; k++)
            F_gpu[x*3 + k] += F_gpu[(x + y*Nr32)*3 + k];
        for (int k = 0; k < 2; k++)
            Epot_gpu[x*2 + k] += Epot_gpu[(x + y*Nr32)*2 + k];
    }

    for (int k = 0; k < 3; k++)
        F_gpu[x*3 + k] *= 12.0*eps;
    Epot_gpu[x*2] *= eps/2.0;
    Epot_gpu[x*2 + 1] *= eps;
}
```

Nie jest to najlepsza algorytmicznie wersja, ale dzięki jej prostocie możemy ją bardzo dobrze zoptymalizować. Jest to najszybsza wersja jaką udało mi się napisać. Przeprowadza całą symulację w czasie 2.374 sekund.

4.6 GPU III

W tej wersji zastosujemy III zasade dynamiki Newtona. Wyliczamy tablice odziaływań w roli gospodarza i gościa.

```
/* wersja gpu 3, podział na atomy w roli hosta i gościa */
real_t* F_gpu_host;
real_t* F_gpu_guest;
real_t* Epot_gpu_host;
real_t* Epot_gpu_guest;
```

Odziaływania liczone w roli gospodarza trzymamy lokalnie w każdym wątku (na końcu przepisujemy do tablicy globalnej), natomiast odziaływania w roli gościa liczymy w każdym obrocie pętli dla innego atomu, dlatego trzymamy te wyniki w pamięci współdzielonej.

```
__global__ void up_forces_gpu_3(real_t* cords, real_t* F_gpu_host, real_t* Epot_gpu_host,
                                real_t* F_gpu_guest, real_t* Epot_gpu_guest)
{
    int x = blockIdx.x*blockDim.x + threadIdx.x;
    int y = blockIdx.y*blockDim.x + threadIdx.x;
    int offset_host = x + blockIdx.y*Nr32;
    int offset_guest = y + blockIdx.x*Nr32;

    __shared__ real_t F_sh[32][3];
    __shared__ real_t Epot_sh[32][2];

    /* caly blok nie ma nic do liczenia */
    if (blockIdx.y*blockDim.x + 31 <= blockIdx.x*blockDim.x) {
        for (int k = 0; k < 3; k++) {
            F_gpu_host[offset_host*3 + k] = 0;
            F_gpu_guest[offset_guest*3 + k] = 0;
        }
        for (int k = 0; k < 2; k++) {
            Epot_gpu_host[offset_host*2 + k] = 0;
            Epot_gpu_guest[offset_guest*2 + k] = 0;
        }
        return;
    }

    for (int k = 0; k < 3; k++)
        F_sh[threadIdx.x][k] = 0;
    for (int k = 0; k < 2; k++)
        Epot_sh[threadIdx.x][k] = 0;

    real_t C_host[3], C_guest[3];
    for (int k = 0; k < 3; k++) {
        C_host[k] = cords[x*3 + k];
        C_guest[k] = cords[y*3 + k];
    }

    real_t F[3] = {0, 0, 0};
    real_t Epot[2] = {0, 0};

    __syncthreads();
```

(na zielono) Dodajemy dodatkowe tablice pamięci współdzielone na wyliczenie odziaływań w roli gości.

(na niebiesko) Jeśli cały blok nie ma nic do roboty (nie liczymy go, bo jego wyniki dostaniemy z III zasady dynamiki Newtona) to czyścimy przydzielone mu globalne fragmenty tablic odziaływań i wychodzimy.

(na różowo) Czyścimy pamięć współdzieloną, kopjujemy do rejestrów współżędne atomu gospodarza i przydzielonego atomu gościa i tworzymy tablice na wyniki w roli gospodarza.

Synchronizujemy wątki.

Po synchronizacji:

```
    __syncthreads();

    for (int poml = 0; poml < 32; poml++) {
        int l = (poml + threadIdx.x)%32;

        real_t rij[3];
        int j = blockIdx.y*32 + l;

        for (int k = 0; k < 3; k++)
            rij[k] = C_host[k] - __shfl_sync(static_cast<unsigned>(-1), C_guest[k], l);

        if (j <= x or x >= N or j >= N)
            continue;

        real_t dl_rij = 0;
        for (int k = 0; k < 3; k++)
            dl_rij += sq(rij[k]);

        if (rOd > 0.0 and dl_rij > sq(rOd))
            continue;

        real_t sig2 = sig*sig/dl_rij;
        real_t sig6 = sig2*sig2*sig2;
        real_t sig12 = sig6*sig6;

        real_t dif = (sig12 - sig6)/dl_rij;

        for (int k = 0; k < 3; k++) {
            F[k] += dif*rij[k];
            F_sh[l][k] -= dif*rij[k];
        }

        Epot[0] += sig12;
        Epot[1] -= sig6;
        Epot_sh[l][0] += sig12;
        Epot_sh[l][1] -= sig6;
    }

    __syncthreads();
```

Pętla liczenia odziaływań praktycznie niezmieniona. Tak jak w wersji CPU liczmy odziaływania dla obu atomów (gospodarza i gościa).

(na różowo) Zmieniamy kolejność iteracji po atomach gościach dla każdego wątku by uniknąć konfliktów banków pamięci.

(na czerwono) Oddziaływania dla atomu gościa zapisujemy w pamięci współdzielonej.

Synchronizujemy wątki.

Po synchronizacji:

```
    __syncthreads();

    for (int k = 0; k < 3; k++) {
        F_gpu_host[offset_host*3 + k] = F[k];
        F_gpu_guest[offset_guest*3 + k] = F_sh[threadIdx.x][k];
    }
    for (int k = 0; k < 2; k++) {
        Epot_gpu_host[offset_host*2 + k] = Epot[k];
        Epot_gpu_guest[offset_guest*2 + k] = Epot_sh[threadIdx.x][k];
    }
}
```

Przepisujemy policzone wyniki częściowe do tablic globalnych.

Musimy jeszcze uruchomić kernel który zsumuje wszystkie wyniki częściowe (i przy okazji podomnaża stałe).

```
__global__ void summer_gpu_3(real_t* F_gpu_host, real_t* Epot_gpu_host,
                             real_t* F_gpu_guest, real_t* Epot_gpu_guest)
{
    int x = blockIdx.x*blockDim.x + threadIdx.x;

    for (int y = 1; y < Nr32/32; y++) {
        for (int k = 0; k < 3; k++) {
            F_gpu_host[x*3 + k] += F_gpu_host[(x + y*Nr32)*3 + k];
            F_gpu_guest[x*3 + k] += F_gpu_guest[(x + y*Nr32)*3 + k];
        }
        for (int k = 0; k < 2; k++) {
            Epot_gpu_host[x*2 + k] += Epot_gpu_host[(x + y*Nr32)*2 + k];
            Epot_gpu_guest[x*2 + k] += Epot_gpu_guest[(x + y*Nr32)*2 + k];
        }
    }

    for (int k = 0; k < 3; k++) {
        F_gpu_host[x*3 + k] += F_gpu_guest[x*3 + k];
        F_gpu_host[x*3 + k] *= 12.0*eps;
    }

    for (int k = 0; k < 2; k++) {
        Epot_gpu_host[x*2 + k] += Epot_gpu_guest[x*2 + k];
        Epot_gpu_host[x*2 + k] *= eps;
    }

    Epot_gpu_host[x*2] /= 2.0;
}
```

Pomimo zmniejszenia ilości obliczeń pogarsza nam się czas względem ostatniej wersji. Cały czas korzystamy z wolnej (w porównaniu z rejestrami) pamięci współdzielonej.

Przeprowadzenie symulacji zajmuje 3.199 sekundy.

4.7 GPU IV

W tej wersji zaimplementujemy listę sąsiedztwa na GPU.

Listę sąsiedztwa będziemy trzymać na GPU. Na CPU będziemy sprawdzać czy listę sąsiedztwa trzeba przebudować, więc początkowe pozycje atomów dalej pozostają w pamięci hosta.

```
/* wersja gpu 4, listy sasiedztwa */
int* G_gpu;
real_t poz0_cpu[N*3];
```

Przy pudowie listy sąsiedztwa, jeden wątek liczy listę dla jednego atomu.

```
__global__ void rebuild_G_gpu(real_t* cords_gpu, int* G_gpu) {
    int x = blockIdx.x*blockDim.x + threadIdx.x;
    int pom = 0;

    if (x >= N)
        return;

    for (int j = 0; j < N; j++) {
        if (x == j)
            continue;
        real_t r = 0;
        for (int k = 0; k < 3; k++)
            r += sq(cords_gpu[x*3 + k]-cords_gpu[j*3 + k]);
        if (r <= sq(rOd + rBuff))
            G_gpu[x*MAX_LI + pom++] = j;
    }

    G_gpu[x*MAX_LI + pom] = -1;
}
```

Sprawdzenie czy należy przebudować listę sąsiedztwa jest identyczne jak na CPU, tylko zamiast funckji budującej uruchamiamy kernel (współżędne atomów już są na GPU daltego nie musimy ich kopiować przed uruchomieniem kernela).

```
void check_G_gpu() {
    auto reb = [&](){
        rebuild_G_gpu<<<dim3(Nr32/32), dim3(32)>>>(cords_gpu, G_gpu);
        cudaAssert(cudaDeviceSynchronize());
        memcpy(poz0_cpu, cords, N*3*sizeof(real_t));
    };

    static bool first_builded = false;
    if (first_builded == false) {
        reb();
        first_builded = true;
        return;
    }

    for (int i = 0; i < N; i++) {
        real_t dl = 0;
        for (int k = 0; k < 3; k++)
            dl += sq(poz0_cpu[i*3 + k]-cords[i*3 + k]);
        if (dl*4.0 >= rBuff*rBuff) {
            reb();
            return;
        }
    }
}
```

Liczenie sił, podobnie jak w wersji CPU, od najprostszej wersji różni się pętlą iterującą się po atomach.

```
__global__ void up_forces_gpu_4(real_t* cords_gpu, real_t* F_gpu, real_t* Epot_gpu, int* G_gpu) {
    int x = blockIdx.x*blockDim.x + threadIdx.x;

    if (x >= N)
        return;

    real_t F[3] = {0, 0, 0};
    real_t Epot[2] = {0, 0};
    real_t C[3];

    for (int k = 0; k < 3; k++)
        C[k] = cords_gpu[x*3 + k];

    for (int pm = 0, j = G_gpu[x*MAX_LI + pm]; G_gpu[x*MAX_LI + pm] != -1; j = G_gpu[x*MAX_LI + ++pm]) {
        real_t rij[3];
        for (int k = 0; k < 3; k++)
            rij[k] = C[k] - cords_gpu[j*3 + k];

        real_t dl_rij = 0;
        for (int k = 0; k < 3; k++)
            dl_rij += sq(rij[k]);

        if (rOd > 0.0 and dl_rij > sq(rOd))
            continue;

        real_t sig2 = sig*sig/dl_rij;
        real_t sig6 = sig2*sig2*sig2;
        real_t sig12 = sig6*sig6;

        real_t dif = (sig12 - sig6)/dl_rij;

        for (int k = 0; k < 3; k++)
            F[k] += dif*rij[k];

        Epot[0] += sig12;
        Epot[1] -= sig6;
    }

    for (int k = 0; k < 3; k++)
        F_gpu[x*3 + k] = F[k]*12.0*eps;
    Epot_gpu[x*2] = Epot[0]*eps/2.0;
    Epot_gpu[x*2 + 1] = Epot[1]*eps;
}
```

W zaznaczonych na różowo miejscach odwołujemy się do pamięci globalnej. Nie jesteśmy w stanie tego poprawić, przez co rozwiązanie znowu jest wolniejsze od poprzedniego.

Przeprowadzenie symulacji zajmuje 8.668 sekund.

4.8 Podsumowanie wyników

Kernel	czas (sekundy)
CPU I	42.254
CPU II	24.147
CPU III	18.735
GPU I	14.920
GPU II	2.374
GPU III	3.199
GPU IV	8.668

Najlepszy okazał się kernel GPU II. Pomimo tego, że kolejne kernele zmniejszają kilku krotnie ilość obliczeń to kożystają z wolniejszej pamięci co ostatecznie je pogarsza.

5 Przejścia fazowe argonu

Przeprowadzimy teraz symulację ze zmianą temperatury układu. Ustawiamy 10’000 kroków fazy termalizacji, 5’000 kroków fazy chłodzenia i 5’000 kroków fazy ogrzewania.

Tak zmienia się temperatura: