MN1. Dynamika molekularna na GPU

Stanisław Sołtan

2 lutego 2015

Streszczenie

Przygotowano symulacje dynamiki molekularnej na CPU i GPU; zwrócono przy tym szczególną uwagę na możliwe do wprowadzenia optymalizacje i wprowadzono w obu przypadkach w najprostszy sposób listę Verleta. Zbadano zgodność obliczeń na CPU i GPU jak również ogólne własności symulacji. Porównano szybkości obliczeń na CPU i GPU przy wykorzystaniu listy Verleta i bez niej. Stwierdzono, że przyspieszenie wynikające z zastosowania listy Verleta zależy zapewne przede wszystkim od fizycznych parametrów symulacji, zaś przyspieszenie wynikające ze zmiany jednostki obliczeniowej na GPU zależy nietrywialnie od różnic między tymi dwoma jednostkami. By jednak to potwierdzić i zbadać dokładniej, potrzebaby wykonać więcej precyzyjnych testów.

Dynamika molekularna (MD, *molecular dynamics*) jest jedną z technik pozwalających przewidywać własności makroskopowe materii z ich własności mikroskopowych; opiera się na założeniu, że wartości mierzone pewnych obserwabli badanego układu odpowiadają średnim z odpowiednich funkcji parametrów mikroskopowych po dostatecznie długim czasie. Zgodnie z hipotezą ergodyczną, to podejście jest równoważne fizyce statystycznej opartej na pojęciu zespołu statystycznego. MD umożliwia jednak przeprowadzanie symulacji numerycznych i pozwala uniknąć konieczności liczenia skomplikowanych sum statystycznych [1, 2]. Ze względu na liczbę elementów symulowanych układów fizycznych, MD wymaga długich obliczeń na standardowych komputerach; możliwe jest jednak przyspieszenie poprzez wprowadzenie równoległych obliczeń. Narzędziem do obliczeń równoległych zyskującym ostatnio na popularności są karty graficzne i związane z nimi środowiska pracy - na przykład udostępniane przez firmę NVIDIA zestaw narzędzi programistycznych CUDA [3].

Celem ćwiczenia było porównanie szybkości obliczeń symulacji 400-500 cząstek na procesorze komputera (CPU) i karty graficznej (GPU).

1 Realizacja MD

1.1 Potencjał oddziaływania

Przyjęto, że cząstki oddziałują ze sobą, potencjałem Lienarda-Jonesa, ale tylko do zadanej krytycznej odległości pomiędzy cząstkami R_C:

$$V_{LJ} = \begin{cases} 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] + const, & r < R_C \\ 0 \qquad\qquad\qquad\qquad\quad , & r > R_C \end{cases},$$

gdzie ϵ to głębokość studni potencjału opisanej tym parametrem, zaś $\sigma < R_c$ to charakterystyczna, skończona odległość na której $V_{LJ} = 0$ [2, 4]. Przedstawione „cięcie" potencjału wywołuje nagły skok energii potencjalnej przy przekraczaniu granicznej odległości R_c. By go zniwelować, za stałą w potencjale wystarcza przyjąć $4\epsilon\left[\left(\frac{\sigma}{R_C}\right)^{12} - \left(\frac{\sigma}{R_C}\right)^{6}\right]$ [4], co, rzecz jasna, nie zmieniało symulowanego ruchu cząstek. Przyjęto $R_C = 2.5\sigma$ Zastosowano periodyczne warunki brzegowe [5].

1.2 Optymalizacja

Cięcie potencjału wprowadza również okazję do optymalizacji - wszak wiele z par cząstek ze sobą przez

długi czas nie oddziałuje, można by więc pomijać obliczenia dla nich. Standardową metodą tego pomijania jest wprowadzenie tzw. listy Verleta: konkretnie - przypisanych do każdej cząstki list cząstek, które są lub w najbliższym czasie mogą być w zasięgu wzajemnego oddziaływania. Przez większość czasu, program wykonuje obliczenia tylko dla oddziaływań z tej listy; co pewną liczbę kroków, dokonuje aktualizacji listy dla każdej cząstki poprzez spisanie jej siąsiadów w pewnej zadanej odległości $R_M > R_C$. Liczbę kroków i pomiędzy aktualizacjami przyjmuje się zwykle za zależny parametr symulacji, obliczany z pozostałych parametrów [5, 6]. Tu przyjęto jednak bardziej „paranoiczną" wersję listy Verleta, obliczającą i na podstawie maksymalnej prędkości v_{max} zarejestrowanej od ostatniej aktualizacji:

$$R_M - R_C \simeq i \cdot 2v_{max} \cdot h.$$

Przyjęto $R_M = 3.3\sigma$.

1.3 Algorytm

Do obliczania zmian współrzędnych cząstek w każdym kroku czasowym, zastosowano algorytm Verleta dla prędkości [1]:

$$r(t+h) = r(t) + v(t)h + \tfrac{1}{2}a(t)h^2$$
$$v(t+h) = v(t) + \tfrac{1}{2}[a(t) + a(t+h)]h$$

2 Programowanie GPU w CUDA

Symulacje przeprowadzano z jednostkami obliczeniowymi:

- CPU: Intel®Pentium(R) 3.00GHz x 2, pamięć RAM: 1 GB
- GPU: GeForce GT 430, z dwoma multiprocesorami (co stanowi ograniczenie dla liczby bloków pracujących równolegle), zdolność obliczeniowa 2.1 [3].

Napisanie w języku C++ programu na CPU nie wykraczało ponad podstawowe umiejętności i nie będzie szerzej omawiane. Przy pisaniu programu na GPU poświęcono sporo uwagi synchronizacji bloków oraz implementacji optymalizacji.

2.1 Synchronizacja bloków

By na GPU móc wykonać wszystkie iteracje na karcie graficznej, potrzebna jest generalnie jakaś metoda synchronizacji nie tylko wątków wewnątrz bloku, ale także bloków między sobą. Przegląd technik synchronizacyjnych dostarcza [7]. Skorzystano z zaproponowanej tam metody wolnej od sekwencyjnych operacji atomowych (*GPU lock-free synchronisation*) z dwoma modyfikacjami: po pierwsze, by zapobiec złej optymalizacji kompilatora, trzeba było zadeklarować globalne zmienne odpowiadające za synchronizację jako ulotne, po drugie, wykorzystano proces synchronizacji także w procesie znajdowania obserwabli oraz do kontroli jednoczesnego aktualizowania list Verleta. Stwierdzono, że naiwny program na GPU z taką synchronizacją działa minimalnie szybciej w porównaniu ze swoimi wcześniejszymi roboczymi wersjami.

Synchronizacja tego typu wykorzystuje zajęte czekanie (*busy-waiting*) wykonywane przez wszystkie wątki równolegle, co każe przypuszczać, że dostępna liczba mikroprocesorów na karcie może ograniczać stosowalność tej techniki. Rzeczywiście, stwierdzono, że dla programu wykonującego wiele operacji w ciągu jednej iteracji synchronizacja „zacina" go na wykorzystywanej karcie graficznej dla liczby bloków większej niż 2. Potwierdza to, że w takim przypadku program nie był wykonywany w całości równolegle. Z drugiej strony, przetestowano także to rozwiązanie osobnym programem, nie wykonującym poza synchronizacją i wyświetlaniem komunikatów żadnej innej pracy. Tu można było przekroczyć liczbę dwóch bloków bez zawieszania programu. Dowodzi to, że przy niskim obciążeniu procesory GPU wykonują przemiennie polecenia dla nadmiarowych bloków.

2.2 Implementacja listy Verleta

Jeśli przez N oznaczyć liczbę cząstek, liczba procesów do wykonania przez wersję niezoptymalizowaną rośnie jak N^2; ściśle rzecz biorąc, należy wykonać $\frac{N(N-1)}{2}$ operacji, po jednej na parę, ponieważ oddziaływania są wzajemne i, oczywiście, brak oddziaływań cząstki samej ze sobą. Wprowadzenie listy Verleta redukuje

liczbę operacji do wielkości w przybliżeniu proporcjonalnej do N.

W najprostszej implementacji na GPU, w danym kroku czasowym wykonuje się obliczenia dla każdej cząstki z osobna, uzyskując liczbę operacji rzędu N; można ją dalej zredukować wprowadzając listę Verleta. Istnieje możliwość większej redukcji przez wykonywanie większej liczby operacji równolegle. Wydaje się, że algorytm z listami Verleta nie może zostać „sparalelizowany" tak, by ruch jednej cząstki mógł być obliczany przez wiele równoległych procesów, lecz i to jest możliwe, przez wykorzystanie algorytmów sortujących [8]. Jednakże, z braku zarówno umiejętności jak i technicznych (mała liczba bloków) nie optymalizowano już bardziej programu na GPU.

3 Wykonanie ćwiczenia

Dla wygody, wykonano symulacje gazu 512 cząstek. Przygotowano cztery wersje programu, po dwie na CPU i GPU

- wersję „naiwną", w każdej iteracji przeprowadzającą obliczenia dla każdej pary na CPU i dla każdej cząstki na GPU,
- wersję zoptymalizowaną, korzystającą z listy Verleta, przeprowadzającą obliczenia dla każdej pary na CPU i każdej cząstki na GPU

Obliczenia były prowadzone w podwójnej precyzji. Programy zaczynały symulację od stanu początkowego, w którym cząstki były rozmieszczone w węzłach sieci sześciennej, wypełniającej całą dostępną dla cząstek przestrzeń. Współrzędne prędkości początkowych były losowane w zakresie $[-0.5, 0.5]$, przy czym przechodzono do układu środka masy (średnia prędkość równa była zero). Parametrami symulacji były: stała b tejże sieci (odległość od węzłu do najbliższego węzłu w kierunku x, y, z), krok czasowy h i liczba iteracji l.

3.1 Testy

Przeprowadzono parę testów zgodności wyników pomiędzy poszczególnymi programami. Już podczas tych testów ujawniła się różnica w szybkości wykonywania obliczeń na GPU i PCU. Testy przeprowadzano przy wartościach b: 1.0, 2.5 i 4.0. Przez „wyniki symulacji" rozumie się wartości energii kinetycznej i energii całkowitej w każdym kroku czasowym.

1. Na przestrzeni 10 000 kroków, nie stwierdzono rozbieżności między wersją naiwną i optymalną na danej jednostce obliczeniowej (CPU lub GPU). Z kolei, między parami programów na różnych jednostkach obliczeniowych, stwierdzono rozbieżności co do dokładnych wyników pojawiające się po liczbie kroków rzędu 5 000 - 8 000, zależnie od warunków początkowych; nie zmieniała się jednak jakościowa (makroskopowa) natura wyników. Powodem rozbieżności mogła być tylko różnica w sposobie zaokrąglania przeprowadzanym przez CPU a GPU. Jednak rozbieżności te uznano za wystarczająco małe, by dla liczby kroków poniżej 5 000 przeprowadzić pozostałe testy tylko na GPU, co pozwoliło zaoszczędzić czas.

2. Zmiana kroku czasowego potrafiła istotnie zmienić wyniki mikroskopowe; za podstawowe kryterium doboru kroku czasowego przyjęto więc stabilność energii całkowitej. Testy pozwoliły wybrać $h = 0.01$ jako wartość właściwą do dalszych badań; drobne zmiany h rzędu 10% wokół tej wartości powodowały rozbieżności w wartościach energii kinetycznej po około 2 000 krokach.

3. Sprawdzono różne losowe rozkłady prędkości. Rozkłady te charakteryzowały się różną energią całkowitą, gdyż przy ich przygotowaniu nie nakładano żadnego warunku na nią. Jednak ponieważ wartości energii były stosunkowo podobne, możliwa była przynajmniej jakościowa ocena podobieństwa wyników dla energii kinetycznej. Istotnie, widoczne było podobieństwo - szczególnie dla początkowych czasów i niskich gęstości, dało się obserwować charakterystyczny okres, kiedy cząstki nie zaczęły jeszcze ze sobą oddziaływać.

Przykładowe wyniki z testu 3 przedstawiono na rys. 1. Widać tam szczególnie charakterystyczny okres,

kiedy układ wychodzi z pierwotnego, wysoce uporządkowanego stanu. Dla dostatecznie małych gęstości, widać też że cząstki przez pewien czas nie oddziałują ze sobą.

3.2 Porównanie czasów obliczeń

Przeprowadzono pomiary czasu obliczeń wszystkich czterech programów dla wartości parametru $b = 1.0$, 2.5, 4.0 i 5.5, dla 500, 1000, 1500, ..., 4000 kroków. Konkretnie, mierzono cały czas działania programu bez ładowania wartości prędkości z przygotowanego pliku. Czas obliczeń był

- rzędu dziesiątek sekund dla naiwnego programu na CPU i przy $b = 1,0$ dla programu zoptymalizowanego;
- rzędu sekund w pozostałych przypadkach dla programu zoptymalizowanego na CPU, dla programu optymalnego na GPU dla $b = 1,0$ i we wszystkich przypadkach dla programu naiwnego na GPU; w tych pozostałych przypadkach, zoptymalizowany program na CPU okazał się szybszy od naiwnego na GPU;
- rzędu dziesiątek, setek milisekund dla zoptymalizowanego programu na GPU.

Poza wspomnianym wyżej wyjątkiem, GPU przeprowadzała obliczenia szybciej niż CPU. Oznaczmy przez współczynnik przyspieszenia K stosunek czasów obliczeń jednej wersji programu do drugiej przy zadanej liczbie kroków i współczynniku b. Na rys. 2 przedstawiono K w czterech kategoriach:

1. program naiwny *versus* zoptymalizowany na CPU,
2. naiwny *versus* zoptymalizowany na GPU,
3. naiwny program na CPU *versus* naiwny na GPU i
4. zoptymalizowany na CPU *versus* zoptymalizowany na GPU.

Dla kategorii 1-3 K generalnie wzrastało, gdy gęstość malała. Fakt ten da się łatwo wytłumaczyć dla kategorii 1 i 2, kiedy wraz z malejącą gęstością lista Verleta była rzadziej rekonstruowana i zawierała mniej cząstek. Wyniki dla kategorii 3, z naiwnymi programami, także tłumaczyć można liczbą oddziaływań: program musiał dokonać więcej operacji gdy dane dwie cząstki oddziaływały w danym momencie niż w przeciwnym. Lecz wzrost K znacznie przychamował dla rzadszych gazów, co sugeruje, że zapewne inny czynnik jest tu naprawdę dominujący.

W przeciwieństwie do pozostałych, K dla kategorii 4 maleje wraz ze spadkiem gęstości. Nie da się tego efektu wytłumaczyć czysto fizycznymi własnościami symulowanych gazów; jest to przejawem jakichś technicznych własności symulacji.

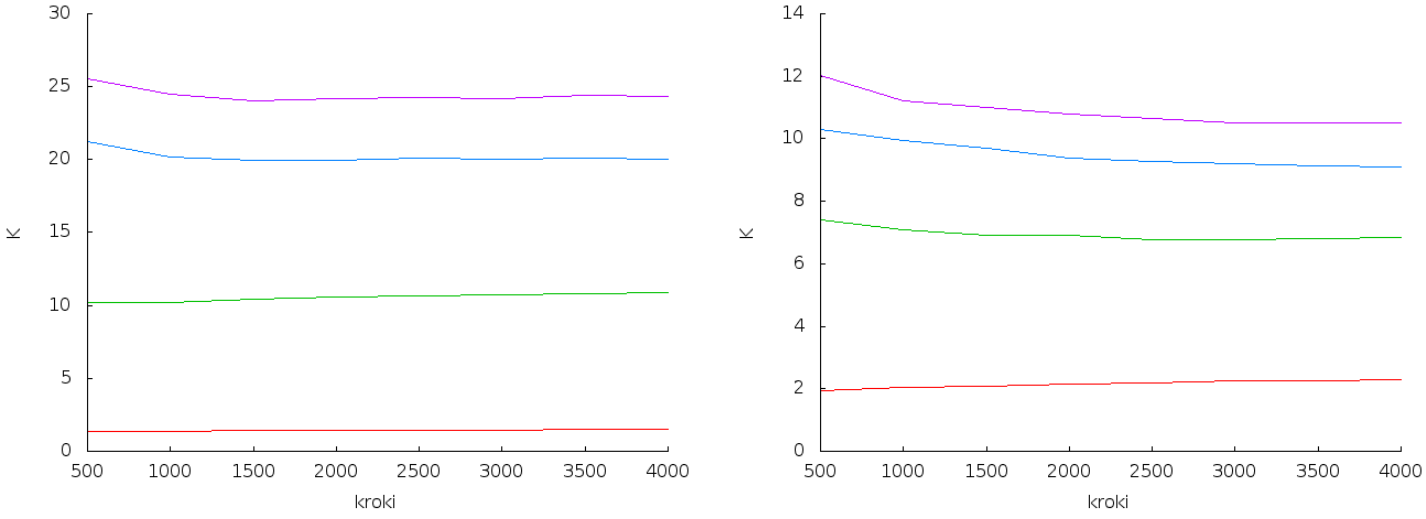
4 Podsumowanie

Jak należało się spodziewać, programy na GPU generalnie liczą szybciej. Można oczywiście napisać program na CPU szybszy od niektórych programów na GPU; ale stosując zasadę, że porównuje się programy na CPU i GPU optymalne dla swoich jednostek obliczeniowych [8] uzyskano szybszy program na GPU. Warto zaznaczyć, że program na CPU wykonywał operacje w liczbie proporcjonalnej do liczby par, czyli mniej niż program na GPU (proporcjonalnej do liczby cząstek); lecz sekwencyjność tych pierwszych obliczeń okazała się tu czynnikiem decydującym.

Zbadano efekt dwóch różnych „efektów przyspieszających" obliczenia: wprowadzenie listy Verleta i przejście z CPU na GPU. Z wyników wnioskuje się, że siła pierwszego efektu jest funkcją parametrów fizycznych konkretnej symulacji i wynika z pierwotnego celu wprowadzenia listy Verleta - to jest redukcji liczby obliczeń dla nieoddziałujących cząstek. Lecz siła drugiego efektu - przejścia z CPU na GPU - zależy zapewne w nietrywialny sposób od różnic w działaniu CPU i GPU.

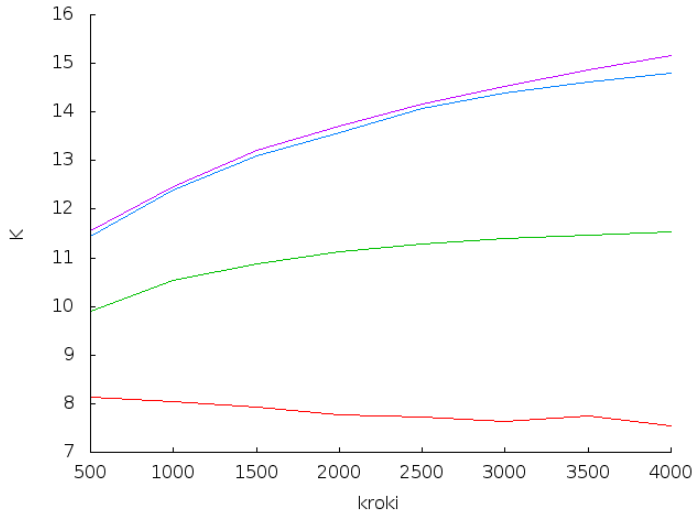
Można zaproponować dalsze testy dla potwierdzenia i badania tych efektów; w szczególności, można by zbadać zależność współczynnika przyspieszenia K od

Rysunek 1: Zachowanie początkowe układu cząstek dla dwóch różnych zestawów początkowej prędkości (po lewej i po prawej) przy $b = 1,0,\ 2,5,\ 4,0$ w kolejności od góry do dołu, przy $h = 0,01$. Czerwoną linią zaznaczono energię całkowitą E, zieloną - kinetyczną T.

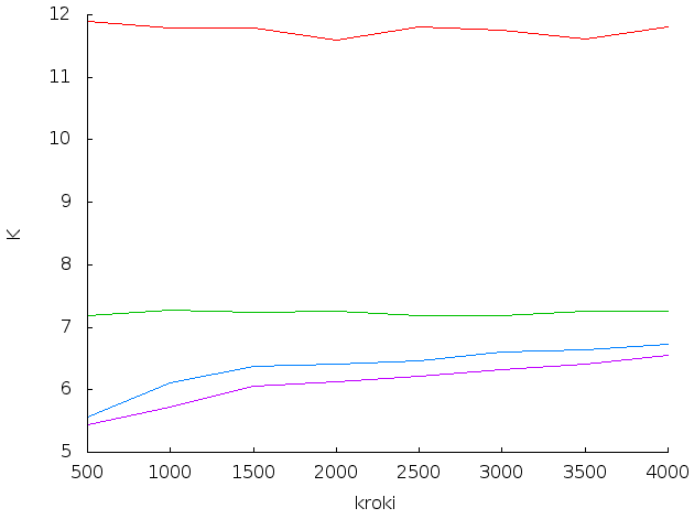


(a)

(b)



(c)



(d)

Rysunek 2: Przyspieszenia optymalizacji na CPU (2(a)) i GPU (2(b)) oraz przy przejściu z CPU na GPU dla naiwnego (2(c)) i zoptymalizowanego (2(d)) podejścia. Linie nie przedstawiają interpolacji i są dodane tylko dla przejrzystości. Gęstości od $b = 1,0$ do $b = 5,5$ oznaczone przez kolory od czerwonego przez zielony i niebieski do fioletowego.

liczby par (cząstek) na liście Verleta, czy też podzielić mierzenie czasu obliczeń na pewne charakterystyczne fragmenty [7]. Można także zawsze wprowadzić kolejne optymalizacje na GPU, jak chociażby równoległe wykonywanie obliczeń i kopiowanie wyników z pamięci GPU na CPU [3] czy wspomnianą już „równoległą" listę Verleta.

Przy pisaniu programów, wykorzystano pewne funkcje wychwytujące błędy w materiałach dydaktycznych do [3]; w strukturze programów, wzorowano się także częściowo na przykładach do tej pozycji. Do diagnostyki błędów wykorzystano narzędzia dostarczane przez NVIDIA wraz z całym oprogramowaniem do CUDA, oraz z programu *Valgrind*. Wspomniano już wcześniej o wykorzystaniu fragmentów kodów z [7].

Literatura

[1] *Theory of Molecular Dynamics Simulations, The Velocity Verlet algorithm*, http://www.ch.embnet.org/MD_tutorial/pages/MD.Part1.htm

[2] Wykład prowadzony przez prof. Marka Napiórkowskiego - fizyka statystyczna B.

[3] J. Sanders, E. Kandrot, *CUDA by Example. An introduction to general-purpose GPU programming*, Addison-Wesley 2011, przykłady do książki dostępne na:

[4] *NSDL Materials Digital Library: Lennard-Jones Potential*, http://www.matdl.org/matdlwiki/index.php/softmatter:Lennard-Jones_Potential

[5] M. P. Allen, *Introduction to Molecular Dynamics Simulation*, rozdział 3

[6] L. Verlet, Phys. Rev. **159**(1), 98-103, 5 lipca 1967

[7] S. Xiao, W. Feng, *Inter-Block GPU Communication via Fast Barrier Synchronization*, dostępne na http://eprints.cs.vt.edu/ (dalej: 00001087/01/TR_GPU_synchronization.pdf)

[8] T. Lipscomb, A. Zou, S. S. Cho, *Parallel Verlet Neighbor List Algorithm for GPU-Optimized MD Simulations*, dostępne na http://users.wfu.edu/choss/ (dalej: docs/papers/21.pdf)