AKADEMIA GÓRNICZO-HUTNICZA

IM. STANISŁAWA STASZICA W KRAKOWIE



Wydział Inżynierii Metali i Informatyki Przemysłowej



Praca Magisterska

pt.

„Implementacja metody dynamiki molekularnej dla heterogenicznych platform sprzętowych”

Imię i nazwisko dyplomanta: **Tomasz Nowak**

Kierunek studiów: **Informatyka Stosowana**

Profil dyplomowania:  **Modelowanie i Technologie Informacyjne**

Nr albumu: **232187**

Promotor: **dr inż. Łukasz Rauch**

Podpis dyplomanta: Podpis promotora:

Kraków 2014

***Oświadczam, świadomy(-a) odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszy projekt inżynierski wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.***

Kraków, dnia ………….… Podpisdyplomanta…………….

**Spis treści**

[1. Wstęp 4](#_Toc405034099)

[2. Dynamika molekularna 5](#_Toc405034100)

[2.1. Przegląd artykułów 6](#_Toc405034101)

[2.2. Przegląd algorytmów 6](#_Toc405034102)

[3. Realizacja oprogramowania 8](#_Toc405034103)

[3.1. GPU 8](#_Toc405034104)

[3.2. Nvidia CUDA 8](#_Toc405034105)

[3.3. Pthreads 9](#_Toc405034106)

[3.4. OpenGL 10](#_Toc405034107)

[3.5. Projekt aplikacji 10](#_Toc405034108)

[3.6. Interfejs użytkownika s 13](#_Toc405034109)

[3.7. Implementacja 15](#_Toc405034110)

[3.8. Optymalizacja 21](#_Toc405034111)

[4. Testy wydajności 25](#_Toc405034112)

[4.1. Pomiar czasu 27](#_Toc405034113)

[4.2. Testy sekwencyjne 29](#_Toc405034114)

[4.3. Testy zrównoleglenia 29](#_Toc405034115)

[4.4. Testy optymalizacji 31](#_Toc405034116)

[4.5. Przyspieszenie 36](#_Toc405034117)

[5. Model algorytm – urządzenie z wykorzystaniem wielu procesorów GPU 37](#_Toc405034118)

[5.1. Podział zadania 37](#_Toc405034119)

[5.2. Skalowalność 39](#_Toc405034120)

[6. Modelowanie defektów 43](#_Toc405034121)

[7. Podsumowanie 56](#_Toc405034122)

[8. Bibliografia 57](#_Toc405034123)

# Wstęp

Rosnąca moc obliczeniowa współczesnego sprzętu dostarcza nowe możliwości rozwiązywania coraz bardziej skomplikowanych zadań. Zagadnieniem bardzo wymagającym obliczeniowo jest dynamika molekularna. Jest jednocześnie ciekawym obszarem badań, który łączy w sobie wiele dziedzin nauki. Zastosowanie dynamiki molekularnej można znaleźć w badaniach nad własnościami fizycznymi materiałów. Symulacje z wykorzystaniem nanostruktur potrafią usprawnić badania, ograniczając czasochłonne wykonywanie doświadczeń z wykorzystaniem rzeczywistych próbek. Skrajnie różnym, ale bazującym na zbliżonej symulacji, doświadczeniem jest analiza wytrzymałości wiązań chemicznych. Domena tego typu symulacji jest dużo mniejsza, ale opiera się wciąż na działaniu sił zewnętrznych na nanostrukturę. Można szukać podobnych zagadnień również w astronomii, biologii, fizyce lub najprościej rzecz ujmując wszędzie, ponieważ wszystko jest zbudowane z atomów. Dlatego właśnie dynamika molekularna opisująca interakcję sił zewnętrznych z atomami, może zostać wykorzystana do dowolnych badań w skali nano. Aby w pełni korzystać z możliwości, jakie daje ta metoda niezbędny jest odpowiedni sprzęt umożliwiający przeprowadzenie symulacji. Wykorzystanie akceleratorów graficznych GPU jest odpowiednim rozwiązaniem, ze względu na łatwą dekompozycję danych oraz możliwość efektywnego wykorzystania oferowanej przez sprzęt masowej wielowątkowości. Odpowiednia implementacja zagadnienia umożliwi wielokrotne wykorzystanie powstałej aplikacji do rozwiązywania problemów z różnych dziedzin przy niewielkim nakładzie pracy włożonym w modyfikację i przystosowanie do aktualnie analizowanej sytuacji.

Celem pracy jest stworzenie jak najbardziej uniwersalnej, otwartej i efektywnej implementacji algorytmów dynamiki molekularnej przy wykorzystaniu akceleratorów GPU. Implementacja zostanie wykorzystana do analizy problemów odkształceń materiałów przy różnorodnej definicji działających sił zewnętrznych.

# Dynamika molekularna

Dynamika molekularna jest komputerową symulacją fizycznych ruchów atomów wewnątrz struktury. Atomy mogą wchodzić w interakcję między sobą w kolejnych krokach czasowych, dając wrażenie poruszania się. Siły pomiędzy atomami oraz potencjał energetyczny definiowane są przez mechanikę molekularną pól siłowych.[9]

(1)

N – ilość atomów w strukturze

mi– masa atomu

ai – przyspieszenie

Fi – działająca siła

V(r*1* …. rN) – funkcja pozycji atomów; reprezentowana jest przez potencjał energetyczny

W praktyce potencjał jest zapisywany, jako suma interakcji pomiędzy parami atomów [8]:

(2)

Najczęściej używanym praktycznie potencjałem jest potencjał Lenarda-Jonesa (LJ). Jest najczęściej wykorzystywany w modelowaniu interakcji pomiędzy atomami. Definiuje go poniższe równanie [8]:

(3)

r – dystans pomiędzy atomami będącymi w interakcji

δ – średnica

ε – minimum lokalne potencjału

ε i δ są stałymi wyznaczanymi tak, aby definiowały właściwości fizyczne materiału [8].

Najbardziej czasochłonną częścią algorytmu jest obliczanie sił interakcji pomiędzy atomami. Zajmuje to ok. 90% czasu całej symulacji. Analizując przebieg symulacji widać, że obliczenie siły występuje pomiędzy każdym atomem struktury z każdym z pozostałych. Daje to złożoność obliczeniową rzędu O(N2). [8]

## Przegląd artykułów

W trakcie analizowania problemu dynamiki molekularnej, można spotkać się z wieloma sposobami optymalizacji algorytmu tak, aby znacząco zmniejszyć złożoność obliczeniową przy jednoczesnym zachowaniu jak największej dokładności.

Jednym ze sposobów jest zastosowanie tzw. promienia odcięcia. Metoda ta redukuje ilość obliczeń tylko do tych niezbędnych, ponieważ interakcja z bardziej oddalonymi atomami jest na tyle znikoma, że może zostać pominięta.[8]

Problem zdefiniowany globalnie tj. przechowywanie całej struktury w pamięci urządzenia nie jest optymalny i ciężki do równoległego wykonywania na więcej niż i układzie GPU jednocześnie. Aby usprawnić obliczenia i przygotować dane wejściowe na platformę równoległą można zastosować listę sąsiadów. Dla każdego atomu, na podstawie promienia odcięcia definiuje się listę atomów, które spełniają dany warunek i tylko one trzymane są w pamięci podczas obliczeń interakcji z wybranym atomem. Daje to możliwość podzielenia danych niezależnie od wielkości całej struktury i rozesłania mniejszych list na kilka niezależnych urządzeń. Rozwiązuje również problem z pamięcią, gdy struktura jest na tyle duża, że nie zmieści się w pamięci pojedynczego urządzenia. Lista sąsiadów jest aktualizowana, co pewien określony przedział czasowy. [8]

Tworzenie listy sąsiadów jest skuteczne podczas symulacji bazujących na gęsto upakowanych atomach (ciała stałe, gęste ciecze), ponieważ atomy są ułożone bardzo blisko siebie i nie mają możliwości szybkiego przemieszczania się. Atomy gazów, mają dużo większe odległości i są niejednorodnie rozłożone w strukturze. Wymagałoby to zastosowanie dużo większego promienia odcięcia i znacznie częstszych aktualizacji list sąsiadów, aby wynik symulacji był wciąż miarodajny i poprawny.

## Przegląd algorytmów

Na przestrzeni lat powstało wiele implementacji dynamiki molekularnej. Algorytmy różniły się od siebie ze względu na zależności związane z architekturą sprzętu, na którym powstawały lub zależnościami samej aplikacji. Z punktu widzenia dekompozycji danych można podzielić je następująco [8]:

* Dekompozycja atomów

Każdy z procesorów ma przydzieloną podzbiór danych wielkości N/P (N jest liczbą atomów; P jest liczbą procesorów) na początku symulacji. Każdy z procesorów musi przechowywać identyczną kopię informacji o atomach (metoda replikacji danych). Metoda ta znalazła zastosowanie wśród architektur z pamięcią współdzieloną. [8]

* Dekompozycja siły

Podzbiór danych pętli sił jest przydzielany każdemu procesorowi. Redukuje to kosztowną komunikację oraz koszt dostępu do pamięci do (PIERWIASTEK Z P) w porównaniu z dekompozycją atomów. Problemem jest odpowiednie zbalansowanie rozkładu danych pomiędzy procesory. [8]

* Dekompozycja przestrzeni

Polega na dekompozycji geometrycznej fizycznej struktury atomów. Każdy procesor oblicza siły działające na atomy tylko na fragmencie całej struktury. W trakcie symulacji procesory wymieniają ze sobą tylko atomy, które przekroczyły granice swojej części struktury i przemieściły się do sąsiedniej. Metoda ta wspiera wielkoskalowe symulacje. Osiąga optymalną skalowalność O(N/P). [8]

Biorąc pod uwagę architekturę, jaka będzie wykorzystywana do wykonywania symulacji, najlepszą metodą dekompozycji danych jest dekompozycja atomów. Główną zaletą tego podejścia jest bardzo dobre zrównoważenie obciążenia pomiędzy urządzeniami oraz łatwa do osiągnięcia skalowalność. Model współdzielonej pamięci GPU pomiędzy procesorami umożliwia osiągnięcie wysokiej wydajności w trakcie obliczeń.

# Realizacja oprogramowania

## GPU

Karty graficzne stają się coraz bardziej zaawansowane, przez co zaczynają być wykorzystywane również do ogólnych zastosowań. Superkomputery rozwiązują wiele skomplikowanych zadań obliczeniowych na liczbach zmiennoprzecinkowych. Procesory GPU dzięki swojej równoległej architekturze, są idealnym wsparciem w obliczeniach. Aby mogły zostać zastosowane w tym celu, potrzebny jest zestaw instrukcji, który umożliwi pisanie aplikacji w sposób zrozumiały zarówno dla urządzenia jak i programisty. [7]

Nvidia jest wiodącym producentem procesorów GPU. Jest również twórcą CUDA - biblioteki i środowiska umożliwiającego wykorzystanie kart graficznych do ogólnych zastosowań.

## Nvidia CUDA

Nvidia CUDA jest platformą służącą do obliczeń równoległych oraz modelem programistycznym umożliwiającym uzyskać znacząco lepszą wydajność obliczeniowa wykorzystując moc układu graficznego GPU.

Środowisko daje programiście bezpośredni dostęp do zestawu wirtualnych instrukcji i pamięci na równoległych jednostkach procesorów. Można dzięki temu wykorzystywać GPU do ogólnych zastosowań nie tylko do przetwarzania grafiki. Biblioteki CUDA wspierają popularne języki programowania: C, C++, Fortran.

Aplikacje wspierające akceleratory graficzne GPU oraz biblioteki CUDA wykorzystywane są między innymi w:

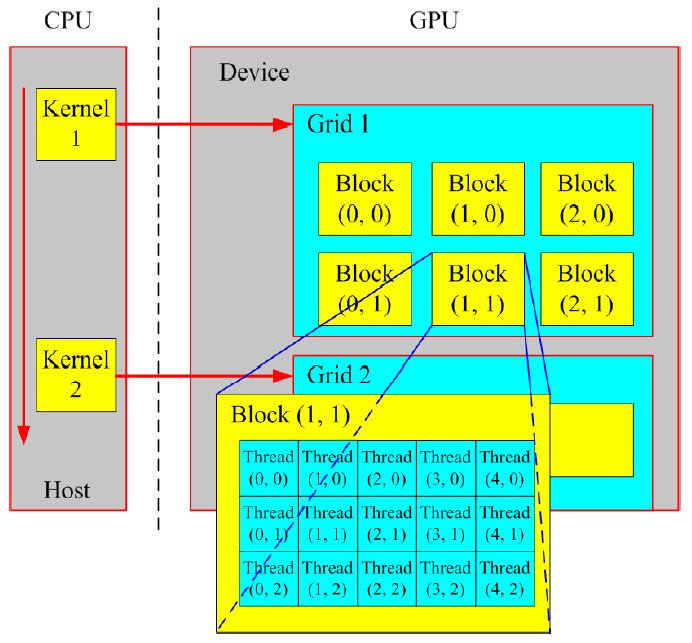
* astronomii,
* biologii,
* chemii,
* fizyce,
* finansach,
* procesach przemysłowych.

Model programowania CUDA definiowany jest przez kilka podstawowych założeń prawdziwych dla programów pisanych pod karty Nvidii:

* kod równoległy tzw. *kernel* jest uruchamiany na urządzeniu (karcie graficznej)   
  i wykonywany przez wiele wątków,
* wątki są grupowane w bloki wątków,
* kod równoległy jest pisany dla pojedynczego wątku
  + każdy wątek wykonuje unikalną część kodu
* bloki są grupowane w siatki

Na potrzeby implementacji zagadnienia będącego tematem pracy wykorzystane zostały[[1]](#footnote-1):

* CUDA Toolkit 5.5
* CUDA Samples – biblioteki wspomagające oraz funkcje pomocnicze
* CUDA Tools – narzędzia do profilowania i badania pamięci



Rys 3.1. Model programowania CUDA

## Pthreads

POSIX Threads często nazywany również Pthreads jest standardem POSIX dla wątków. Standard ten definiuje API[[2]](#footnote-2) do tworzenia i manipulowania wątkami. Implementacja ta jest dostępna dla wielu systemów operacyjnych opartych na systemie UNIX takich jak: FreeBSD, NetBSD, OpenBSD, GNU/Linux, Mac OS X, Solaris. Istnieje również implementacja przeznaczona dla systemu Microsoft Windows zawarta w WinAPI[[3]](#footnote-3). [6]

Pthreads definiuje zestaw typy, funkcje, stałe dla języka C. Implementacja znajduje się w pliku *pthreads.h* biblioteki zarządzającej wątkami. Zawiera ponad 100 funkcji, które można podzielić na grupy: [6]

* zarządzanie wątkami,
* muteksy,
* zmienne warunkowe,
* synchronizacja g

Podstawowym celem Pthreads jest przyspieszenie realizacji zadań. Porównując koszty tworzenia i zarządzania procesem, wątek może być utworzony przy minimalnym udziale zasobów systemu.

W powyższym projekcie technologia Pthreads została wykorzystana w celu zarządzania akceleratorami obliczeń, jakimi są karty graficzne. Bierze udział w procesie podziału danych i przygotowaniu ich do przesłania na zewnętrzne urządzenia. Pomaga również w końcowym procesie zarządzania wynikami obliczeń dostarczonymi do maszyny gospodarza (eng. host).

## OpenGL

OpenGL jest wieloplatformową technologią używaną przez różnorodne języki programowania umożliwiającą przetwarzanie obrazu dwu- i trójwymiarowego. Wykorzystywany jest do interakcji z procesorami graficznymi GPU, aby uzyskać sprzętową akcelerację przetwarzanego obrazu. OpenGL jest stale rozwijaną technologią, która wspiera pojawiające się na rynku najnowsze karty graficzne wiodących producentów.

## Projekt aplikacji

Jednym z celów projektu było stworzenie aplikacji, która będzie nie tylko implementacją mechaniki dynamiki molekularnej, ale również otwartym oprogramowaniem łatwym w użytkowaniu i rozbudowie. Istotną cechą programu jest możliwość wizualizacji wyników, jak również śledzenia zmian zachodzących w strukturze w czasie rzeczywistym. W przypadku uruchamiania testów na klastrach obliczeniowych jest możliwość uruchomienia aplikacji wyłącznie w trybie tekstowym.

Aplikacja wykorzystuje funkcje pomocnicze oraz obsługę błędów pochodzące z biblioteki CUDA. Każdy z plików będący częścią tej biblioteki jest oznaczony na początku specjalnym tekstem. Wspomnienie źródła pochodzenia takiego kodu umożliwia jego wykorzystanie w dowolnym celu edukacyjnym nie łamiąc jednocześnie praw jego stosowania. Przykładowe oznaczenie:

/\*\*

\* Copyright 1993-2013 NVIDIA Corporation. All rights reserved.

\*

\* Please refer to the NVIDIA end user license agreement (EULA) associated

\* with this source code for terms and conditions that govern your use of

\* this software. Any use, reproduction, disclosure, or distribution of

\* this software and related documentation outside the terms of the EULA

\* is strictly prohibited.

\*

\*/

////////////////////////////////////////////////////////////////////////////////

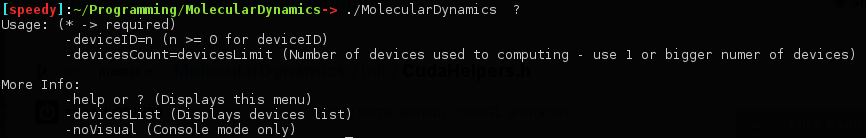
// These are CUDA Helper functions for initialization and error checking

Projekt zakładał stworzenie spójnej aplikacji składającej się z jak najbardziej niezależnych od siebie elementów współpracujących ze sobą. Takie założenie umożliwiło realizację oprogramowania, składa się z poniższych komponentów:

* MolecularDynamics  
  Jest to główny komponent programu. Struktura wejściowe jest tutaj tworzona i inicjalizowana. Odpowiada również za wczytanie parametrów i uruchomienie symulacji.
* Simulation  
  Jest komponentem zajmujący się przetwarzaniem danych wejściowych oraz sterowaniem przebiegu symulacji na GPU. Przetwarzane są tutaj wszelkie parametry dodatkowe podawane przez użytkownika w trakcie uruchamiania programu. Mają one wpływ na typ symulacji oraz urządzenia, które zostaną wykorzystane do jej przeprowadzenia.
* Atom  
  Jest podstawowym elementem wykorzystywanym w dynamice molekularnej. Przechowuje szczegółowe informacje o położeniu aktualnym oraz początkowym, typie czy sile działającej na pojedynczy atom w strukturze molekularnej.
* Structure  
  Jest kombinacją setek, tysięcy lub nawet milionów atomów, tworzących całość, jaką jest struktura molekularna. Przechowuje informacje o typie zastosowanego potencjału, sposobie przyłożenia siły zewnętrznej oraz wartości tej siły. Jest głównym parametrem wejściowym do symulacji.
* GpuHandler  
  Jest komponentem ułatwiającym zarządzanie urządzeniami GPU. Przechowuje identyfikatory dostępnych kart graficznych oraz ich ilość. Umożliwia ich inicjalizowanie i przygotowanie do obliczeń.
* GpuDisplay  
  Jest częścią aplikacji zarządzającą wizualizacją danych. Zajmuje się inicjalizacją środowiska OpenGL, umożliwia konwersję danych wykorzystywanych w obliczeniach do postaci umożliwiającej ich wyświetlenie oraz obsługą myszki i klawiatury. Zapewnia możliwość interakcji ze strukturą w czasie rzeczywistym oraz podgląd każdej części struktury w dowolnym powiększeniu.
* GpuKernel  
  Jest to komponent zajmujący się wykonywaniem obliczeń przy pomocy GPU. Zarządza pamięcią oraz danymi przesyłanymi na urządzenie. Uruchamia funkcje przeznaczone na urządzenie (eng. kernel). Umożliwia wykonywanie obliczeń w trybie wizualizacji lub trybie tekstowym.
* CudaHelpers  
  Jest miejscem do przechowywania elementów aplikacji wykorzystywanych przez wszystkie komponenty wchodzące w interakcję z GPU. Przechowuje dostępne kernele oraz funkcje zajmujące się wielowątkowym przetwarzaniem na kilku urządzeniach GPU.
* Log  
  Jest użytecznym komponentem, umożliwiającym tworzenie logów na standardowym wyjściu jak i w pliku tekstowym. Ułatwia śledzenie błędów, gdy aplikacja niespodziewanie wyrzuci błąd lub przestanie działać.

## Interfejs użytkownika

Program zaprojektowany jest tak, aby był jak najprostszy w obsłudze. Jest uruchamiany z linii poleceń z kilkoma, najczęściej używanymi i najistotniejszymi, wymaganymi parametrami. Pozostałe ustawienia symulacji wczytywane są z plików konfiguracyjnych. Takie podejście nie wymaga ponownej kompilacji przy zmianie rozwiązywanego problemu lub przy modyfikacji struktury atomów. Dostępne opcje uruchomienia mogą zostać wyświetlone po wpisaniu parametrów ‘help’ lub ‘?’.



Rys 3.2. Opcje interfejsu linii poleceń podczas uruchamiania programu.

W celu poprawnego uruchomienia programu wymagane jest podanie jednego z dwóch parametrów: ‘deviceID’ lub ‘devicesCount’. Pozwoli to wykonać obliczenia na konkretnie wybranym urządzeniu GPU lub na losowo wybranym z dostępnych. Aby wskazać konkretną kartę należy podać jej ID. Aby określić ID urządzenia, które ma być użyte można skorzystać z opcji wyświetlenia wszystkich dostępnych urządzeń Nvidii. Służy do tego parametr ‘devicecsList’.



Rys 3.3. Lista dostępnych urządzeń GPU

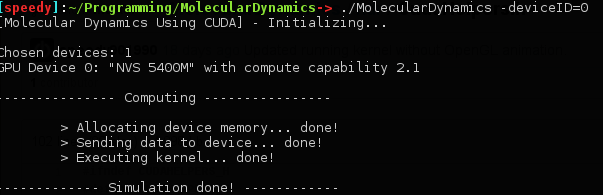
W przypadku pominięcia wyboru urządzenia program zwróci błąd. Jest to celowy zabieg, aby zwiększyć kontrolę i świadomość, z jakich urządzeń oprogramowanie będzie korzystało podczas symulacji.

Pozostała część konfiguracji wczytywana jest z katalogu *config*. Znajdują się tam dwa pliki: *simulation.cfg* oraz *structure.cfg.* Pierwszy z nich zawiera informacje o typie użytego potencjału. Drugi natomiast przechowuje konfigurację struktury wejściowej: jej rozmiar oraz typ i wartość przyłożonej siły.



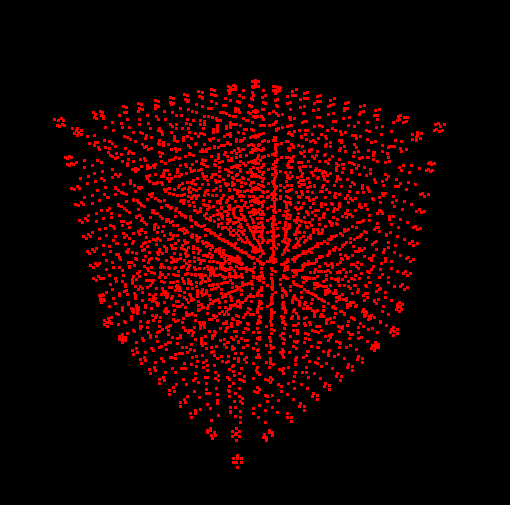
Rys 3.4. Plik konfiguracyjny programu – structure.cfg.

Po dokonaniu poprawnej konfiguracji program może zostać uruchomiony w dwóch trybach: z wizualizacją lub bez. Niezależnie od wybranej opcji wynik wyświetlany przez konsolę będzie taki sam. Pozwala to śledzić aktualny status wykonywania symulacji oraz zaobserwować moment zakończenia. Dane wyświetlane są szczegółowo, przez co widać na którym urządzeniu wykonywane są obliczenia, kiedy dane wysyłane są na GPU, a kiedy uruchomiony zostanie kernel.



Rys 3.5. Przykład poprawnie uruchomionej i wykonanej symulacji.

W przypadku uruchomienia symulacji z trybem wizualizacji, udostępniony zostaje interfejs do sterowania wyświetlaniem struktury w czasie rzeczywistym. Aby obrócić strukturę należy przytrzymać lewy przycisk myszy oraz przesunąć urządzenie wskazujące w kierunku, w którym struktura ma zostać obrócona. Pod prawym przyciskiem myszy zaimplementowana została funkcja przybliżania i oddalania obrazu. Pozwala to na przybliżenie dowolnej części struktury i obserwację bezpośredniego ruch pojedynczych atomów. Klawisz *Escape* służy do zakończenia symulacji i zwolnienia pamięci zaalokowanej w czasie działania programu.



Rys 3.6. Wizualizacja struktury w czasie rzeczywistym.

## Implementacja

Proces implementacji został podzielony na dwa etapy. Pierwszym było przygotowanie danych do obliczeń i zapewnienie ich przepływu przez wszystkie komponenty wyszczególnione w rozdziale 3.5. Na tym etapie celem było uzyskanie niezależności i szybkiego dostępu. Dane przekazywane były bez kopiowania ich, co zaoszczędziło sporo zasobów systemowych. Jedynym elementem gdzie wymagane było kopiowanie, jest moment przesyłania ich do urządzenia GPU. Ten element również został ograniczony do minimum, gdyż do pamięci karty graficznej trafiają tylko dane bezpośrednio wykorzystywane w danej iteracji. Wszystkie z wymienionych operacji wykonywane były po stronie ‘gospodarza’, czyli komputera PC lub klastra. Drugim etapem implementacji było napisanie zarówno funkcji bezpośrednio wykonywanych przez GPU. Zadania te dzieliły się na część obliczeniową oraz graficzną – wizualizację. Celem było zapewnienie płynnego przepływu danych pomiędzy komputerem PC, a urządzeniami GPU.

**Symulacja**

**GpuHandler**

**Wyniki**

**Wizualizacja**

**Brak wizualizacji**

**Kernel**

**Kernel**

Rys 3.7. Schemat przepływu danych w programie

Wyzwaniem, jakiemu należało sprostać w pierwszej kolejności było połączenie wizualizacji danych z jednoczesnym wykonywaniem obliczeń, a to wszystko przy użyciu jednej karty graficznej. Problem tkwił w sposobie wykorzystania GPU w obydwu przypadkach. Przetwarzanie obrazu jest podstawową funkcjonalnością tego typu urządzeń. Dane przeznaczone do wyświetlenia na ekranie są przechowywane w zupełnie innym formacie niż te, przeznaczone do obliczeń. Aby połączyć te dwie funkcjonalności ze sobą wykorzystany został OpenGL Vertex Buffer Object (VBO). Taki obiekt akceptuje dane w formacie *float4,* co znacznie ułatwia implementację. Wektory współrzędnych atomów znajdujące się w trzech osobnych tablicach x, y, z przepisane do zmiennej typu *float4\** bez trudu zostają przesłane i wyświetlone dzięki technologii OpenGL.

|  |
| --- |
| \_\_global\_\_ void update\_display (float4 \*pos, Structure \* input, float time)  {  int atomsCount = input->atomsCount;  int tmpCount = 0;  float u, v, w;  for (int i=0 ; (i<input->dim.x) && (tmpCount < atomsCount) ; i++) {  for (int j=0 ; (j<input->dim.y) && (tmpCount < atomsCount) ; j++) {  for (int k=0 ; (k<input->dim.z) && (tmpCount < atomsCount); k++) {  u = input->atoms[tmpCount].pos.x \* 0.1f;  w = input->atoms[tmpCount].pos.y \* 0.1f;  v = input->atoms[tmpCount].pos.z \* 0.1f;  **pos[tmpCount] = make\_float4(u, w, v, 1.0f);**  tmpCount++;  }  }  }  } |

Kod 3.1. Kernel konwertujący dane obliczeniowe na dane wyświetlane przez OpenGL.

Taka konwersja musi nastąpić po każdej iteracji obliczeniowej. Jest to niezbędne, aby w czasie rzeczywistym móc obserwować zmiany struktury atomów. Można próbować przyspieszyć wydajność przy wyświetlaniu konwertując dane, co kilka iteracji. Zalecane jest to jednak tylko podczas symulacji o małej dynamice lub gdy nie zachodzi potrzeba dogłębnej analizy zmian, a tylko ogólny zarys powstałych zmian.

Aby w płynny i przejrzysty sposób połączyć wyświetlanie z obliczeniami, część obliczeniowa jest częścią głównej pętli OpenGL’a. Zasubskrybowanie na funkcję *display()* sprawia, że przy każdym przebiegu głównej pętli OpenGL wykonuje operacje w niej zawarte. Zazwyczaj są to elementy odpowiedzialne za translację, ustawienia kolorów, kopiowania buforów, zmiany widoku. W tym przypadku wstrzyknięta jest również funkcja odpowiedzialna za wykonywanie obliczeń przy pomocy technologii CUDA. Jej uruchomienie powoduje modyfikację bufora VBO, który niedługo później jest renderowany przez OpenGL.

|  |
| --- |
| void GpuDisplay::display() {  sdkStartTimer(&timer);  // run CUDA kernel to generate vertex positions  **runCuda(&cuda\_vbo\_resource);**  glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);  // set view matrix  glMatrixMode(GL\_MODELVIEW);  glLoadIdentity();  //glTranslatef(translate\_x, translate\_y, translate\_z);  glTranslatef(0, 0, translate\_z);  glRotatef(rotate\_x, 1.0, 0.0, 0.0);  glRotatef(rotate\_y, 0.0, 1.0, 0.0);  // render from the vbo  glBindBuffer(GL\_ARRAY\_BUFFER, vbo);  glVertexPointer(4, GL\_FLOAT, 0, 0);  glEnableClientState(GL\_VERTEX\_ARRAY);  glColor3f(1.0, 0.0, 0.0);  glDrawArrays(GL\_POINTS, 0, mesh\_width \* mesh\_height \* mesh\_depth);  glDisableClientState(GL\_VERTEX\_ARRAY);  glutSwapBuffers();  g\_fAnim += 0.01f;  sdkStopTimer(&timer);  computeFPS();  } |

Kod 3.2. Funkcja display odpowiedzialna za uruchomienie obliczeń i wyświetlenie rezultatów.

Po zapewnieniu poprawnego przetwarzania i konwertowania danych można przejść do przeanalizowania zaimplementowanego algorytmu dynamiki molekularnej. Pierwsza, podstawowa wersja zakłada prostą implementację zachowując ideę i poprawne odwzorowanie fizyki procesu. W kolejnych rozdziałach pokazane będą próby optymalizacji tego rozwiązania oraz inne podejście do rozwiązywanego problemu.

|  |
| --- |
| \_\_global\_\_ void MD\_LJ\_kernel(Structure \*input, Structure \*output, float time) {  int tid = threadIdx.x + blockIdx.x \* blockDim.x;  int atomIndexStart = tid;  int atomIndexEnd = input->atomsCount;  float force[3] = {0.0f, 0.0f, 0.0f};    // COMPUTING  float dX = 0;  float dY = 0;  float dZ = 0;  float x = 0, y = 0, z = 0;  float distance = 0;  float potential = 0;  float deltaTimeSquare = pow(0.05f, 2);  for (int i=atomIndexStart ; i<atomIndexEnd ; i += blockDim.x \* gridDim.x) {  force[0] = 0.0f;  force[1] = 0.0f;  force[2] = 0.0f;  for (int j=0 ; j<input->atomsCount ; j++) {  if (i == j)  continue;    dX = input->atoms[j].pos.x - input->atoms[i].pos.x;  dY = input->atoms[j].pos.y - input->atoms[i].pos.y;  dZ = input->atoms[j].pos.z - input->atoms[i].pos.z;  distance = sqrtf(pow(dX, 2) + pow(dY, 2) + pow(dZ, 2));    if (distance >= 2.5)  continue;    potential = 4 \* (pow((1.0f/distance), 12) - pow((1.0f/distance), 6) );  if (potential > 50)  continue;  force[0] += -(dX / distance) \* potential;  force[1] += -(dY / distance) \* potential;  force[2] += -(dZ / distance) \* potential;  }  output->atoms[i].pos.x = input->atoms[i].pos.x + 0.5 \* force[0] \* deltaTimeSquare;  output->atoms[i].pos.y = input->atoms[i].pos.y + 0.5 \* force[1] \* deltaTimeSquare;  output->atoms[i].pos.z = input->atoms[i].pos.z + 0.5 \* force[2] \* deltaTimeSquare;  }  } |

Kod 3.3. Główny kernel symulacji implementujący dynamikę molekularną.

W tym podejściu implementacji kernela, poszczególne atomy zostały przydzielone osobnym wątkom na GPU. Indeks atomu, a zarazem wątku, jest obliczana ze wzoru: *tid = threadIdx.x + blockIdx.x \* blockDim.x*. Wykorzystując wbudowane w CUDA zmienne można jednoznacznie określić szukany indeks tak, aby był unikalny. Jest to istotne, aby kilkakrotnie nie modyfikować tych samych danych i nie doprowadzać do sytuacji gdzie mógłby pojawić się wyścig między wątkami. Pozycja atomu jest modyfikowana na podstawie wpływu otaczających go sąsiednich molekuł. Do obliczeń brane są tylko te sąsiadujące atomy, których odległość jest mniejsza bądź równa określonej odległości – w tym przypadku 2.5 jednostki. Jest to zastosowanie tzw. promienia odcięcia. Jeśli odległość mieści się w ograniczeniach, obliczana jest siła oddziaływania na podstawie potencjału energetycznego oraz samej odległości. Ostatnim krokiem, aby można było mówić o dynamice, jest uwzględnienie zewnętrznej siły działającej na strukturę. W całej symulacji masa atomu została ustalona na wartość 1, przez co może ona zostać pominięta w obliczeniach. Jest to istotny zabieg, ponieważ GPU ma w ten sposób kilka operacji zmiennoprzecinkowych do wykonania mniej. W skali tysięcy atomów uzyskuje się większą wydajność. W kolejnych rozdziałach zostaną przedstawione sposoby optymalizacji oraz testy wykonane przy użyciu opisywanego oprogramowania. Istotną częścią funkcji jest zastosowanie warunków brzegowych. Jeden z nich został przytoczony powyżej i jest nim promień odcięcia. Drugi natomiast to kontrola obliczanego potencjału energetycznego. Potencjał LJ charakteryzuje się tym, że w miarę zbliżania się do 0 (oś x) jego wartości drastycznie rosną (rys 3.8). Aby zapobiec niekontrolowanemu zachowaniu atomów przy tak dużych wartościach potencjału został on ograniczony do maksymalnej wartości tak, aby analizowana struktura nie uległa zniszczeniu nawet przy bardzo bliskim kontakcie atomów.

Wyk 3.1. Potencjał Lenard-Jones dla parametrów użytych w symulacji.

## Optymalizacja kodu źródłowego

Pierwszym etapem optymalizacji była redukcja ilości przejść pętli w całej symulacji. Dzięki temu znacznie udało się zredukować liczbę wykonywanych operacji. Redukcja dotyczyła połączenia dwóch etapów: przepisania danych ze struktury wyjściowej na wejściową z przygotowaniem danych do wyświetlenia. (Kod 3.4)

|  |
| --- |
| \_\_global\_\_ void update\_structure\_and\_display(float4 \*pos, Structure \*input, Structure \*output) {  int tid = threadIdx.x + blockIdx.x \* blockDim.x;  float u, v, w;  input->atomsCount = output->atomsCount;  for (int i=tid ; i<input->atomsCount ; i+=blockDim.x \* gridDim.x) {  input->atoms[i].pos.x = output->atoms[i].pos.x;  input->atoms[i].pos.y = output->atoms[i].pos.y;  input->atoms[i].pos.z = output->atoms[i].pos.z;  input->atoms[i].force = output->atoms[i].force;  input->atoms[i].acceleration = output->atoms[i].acceleration;  input->atoms[i].status = output->atoms[i].status;  input->atoms[i].fixed = output->atoms[i].fixed;  u = input->atoms[i].pos.x \* 0.1f;  w = input->atoms[i].pos.y \* 0.1f;  v = input->atoms[i].pos.z \* 0.1f;  pos[i] = make\_float4(u, w, v, 1.0f);  }  } |

Kod 3.4. Optymalizacja aktualizacji danych struktury oraz wyświetlania.

Analizując zmieniony kod 3.4. można zauważyć, że 3 pętle for stosowane do uaktualniania struktury zostały zastąpione pojedynczą. Dodatkowo pętla ta została zrównoleglona podobnie jak w kodzie 3.3. Pomimo, że funkcja *update\_structure\_and\_display* nie zawiera wielu operacji zmiennoprzecinkowych, to powyższe zabiegi znacznie zwiększyły wydajność programu, co zostanie zaprezentowane w rozdziale 4 podczas analizie wydajności. Świadczy to o tym, że czas dostępu do pamięci przy dużej ilości operacji może być istotnym parametrem optymalizacji. Do tej pory wszystkie dane były przechowywane w pamięci RAM karty graficznej. Warto, więc zastanowić się nad wykorzystaniem innych typów dostępnych pamięci takich jak rejestry i pamięć współdzielona, czyli bardzo szybkiej, lecz jednocześnie bardzo małej.

Etap drugi optymalizacji obejmuje ujednolicenie deklaracji i definicji zmiennych oraz wykorzystanie optymalizacji automatycznych kompilatora Nvidii *nvcc.* Przy wykorzystaniu GPU do obliczeń oczywistym jest, że karta graficzna będzie wykonywała wiele operacji matematycznych na liczbach zmiennoprzecinkowych. Przyspieszenie wykonywania każdej, nawet najmniejszej operacji może dać w efekcie widoczny wzrost wydajności. Kompilator *nvcc* posiada parametr, który sprawi, że tego typu operacje będą, teoretycznie, wykonywane ułamki sekundy szybciej.

|  |
| --- |
| NVCCFLAGS = -w -g -**use\_fast\_math** |

Kod 3.5. Optymalizacja kompilatora nvcc

Wykorzystanie opcji zaprezentowanej w kodzie 3.5 powinno dać zamierzony efekt, natomiast, aby w pełni wykorzystać to usprawnienie, warto spojrzeć czy kernel nie zawiera operacji na liczbach podwójnej precyzji. Pomimo, że zmienna jest deklarowana, jako typ *float* to przypisanie do niej wartości 0 jest przypisaniem wartości podwójnej precyzji (*double)*. Nastepuje w tym miejscu automatyczna konwersja na typ pojedynczej precyzji. Aby uniknąć konieczności konwersji, można ręcznie podawać już poprawną i oczekiwaną wartość 0.0f. W tym przypadku konwersja nie następuje, ponieważ jest niepotrzebna.

Istotną techniką optymalizacji kodu jest tzw. rozwijanie pętli. Służy ona od rozpisania przykładowo pętli *for* tak, aby nie było sprawdzanie warunku końca pętli było wykonywane jak najrzadziej. Zaoszczędzi to cenny czas i ułatwi wykonywanie kolejnych kroków obliczeniowych. Nie zawsze jest jednak konieczność wykonywania tych operacji ręcznie. W bibliotece CUDA została zaimplementowana dyrektywa preprocesora (kod 3.6), która takie operacje wykonuje za programistę na poziomie kompilacji kodu. Taka dyrektywa musi zostać umieszczona przed każdą pętlą, która ma być rozwinięta.

|  |
| --- |
| **#pragma unroll**  for (int i=atomIndexStart ; i<atomIndexEnd ; i += blockDim.x \* gridDim.x) {  force[0] = 0.0f;  force[1] = 0.0f;  force[2] = 0.0f;  //…………………………………… |

Kod 3.6. Optymalizacja – rozwijanie pętli

W etapie trzecim celem było umieszczenie wszystkich możliwych danych w pamięci tak, aby dostęp do często używanych zmiennych był szybszy. W tym celu wszystkie zmienne przechowujące odległości pomiędzy atomami, obliczany potencjał, siły, czas zostały umieszczone w zmiennych typu *register*. Odpowiada to przechowywaniu danych w rejestrach, czyli najszybszej dostępnej pamięci GPU (kod 3.7).

|  |
| --- |
| // COMPUTING  register float dX = 0.0f;  register float dY = 0.0f;  register float dZ = 0.0f;  register float x = 0.0f, y = 0.0f, z = 0.0f;  register float distance = 0.0f;  register float potential = 0.0f;  register float deltaTimeSquare = 0.0025;  //……………………………… |

Kod 3.7. Optymalizacja – zastosowanie rejestrów.

Drugim krokiem tego etapu było usunięcie wszystkich możliwych operacji arytmetycznych poza kernel lub zmniejszenie ich ilości w miarę możliwości. Do tej pory kwadrat różnicy czasu był obliczany na bieżąco na początku funkcji. Ponieważ jest to stała wartość dla całej symulacji, takie obliczenie zostało zastąpione stałą. Zredukowana została w ten sposób operacja potęgowania.

W kolejnym rozdziale przedstawione zostaną testy wydajności zaimplementowanego algorytmu oraz analiza wydajności po zastosowaniu powyższych optymalizacji.

# Testy wydajności

Testy wydajności będą wykonywane przy stałej konfiguracji, aby wyniki były miarodajne i porównywalne pomiędzy testami. Analizie poddane zostaną dwie struktury o różnej wielkości. Po uzyskaniu najszybszego i najbardziej wydajnego algorytmu wykonane zostaną testy na klastrze obliczeniowym, gdzie zbadany będzie również temat analizy skalowalności oraz modelu algorytm-urządzenie. Powyższe zagadnienia zawarte będą w rozdziale 5.

Analiza i optymalizacja będzie wykonywane przy wykorzystaniu poniższej konfiguracji sprzętowej:

|  |  |
| --- | --- |
| GPU | **Nvidia Quadro NVS 5400M** |
| CPU | Intel(R) Core(TM) i3-3110M CPU @ 2.40GHz |
| RAM | DDR3 8GB 1333MHz |
| HDD | 500GB 7200 obrotów |

Tab 4.1. Konfiguracja sprzętowa – optymalizacja i analiza wydajności

Do wykonywania testów, budowania kodu i konfiguracji środowiska wykorzystane zostało poniższe oprogramowanie:

|  |  |
| --- | --- |
| System operacyjny | Debian 3.2.57-3+deb7u2 x86\_64 GNU/Linux kernel #3.2.0-4-amd64 |
| CUDA | Nvidia CUDA 5.5 + Nvidia CUDA Toolkit |
| Sterowniki GPU | Nvidia Driver Version 319.60 |
| OpenGL | 1.4 |
| g++ | g++ (Debian 4.7.2-5) 4.7.2 |

Tab 4.2. Konfiguracja oprogramowania – optymalizacja i analiza wydajności

Każda karta graficzna Nvidii posiada charakterystyczne dla siebie parametry. Z punktu widzenia biblioteki CUDA dochodzi jeszcze kilka nowych bardzo istotnych podczas wykonywania obliczeń. Najistotniejszymi są maksymalna ilość wątków przypadających na blok oraz maksymalna ilość bloków przypadających na siatkę. Są one tak istotne, ponieważ to głównie tymi parametrami steruje się przy odpowiednim podziale problemu na wątki. Równie istotne są rozmiary pamięci dostępnych na GPU; szczególnie pamięć *shared.* Jest ona dość mała, lecz odpowiednie jej wykorzystanie może przynieść znaczy wzrost wydajności.

|  |
| --- |
| Device 0: "NVS 5400M"  CUDA Driver Version / Runtime Version 5.5 / 5.5  CUDA Capability Major/Minor version number: 2.1  Total amount of global memory: 1024 MBytes (1073283072 bytes)  ( 2) Multiprocessors, ( 48) CUDA Cores/MP: 96 CUDA Cores  GPU Clock rate: 950 MHz (0.95 GHz)  Memory Clock rate: 900 Mhz  Memory Bus Width: 128-bit  L2 Cache Size: 131072 bytes  Maximum Texture Dimension Size (x,y,z) 1D=(65536), 2D=(65536, 65535), 3D=(2048, 2048, 2048)  Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048 layers  Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048 layers  Total amount of constant memory: 65536 bytes  **Total amount of shared memory per block: 49152 bytes**  Total number of registers available per block: 32768  Warp size: 32  Maximum number of threads per multiprocessor: 1536  Maximum number of threads per block: 1024  **Max dimension size of a thread block (x,y,z): (1024, 1024, 64)**  **Max dimension size of a grid size (x,y,z): (65535, 65535, 65535)**  Maximum memory pitch: 2147483647 bytes  Texture alignment: 512 bytes  Concurrent copy and kernel execution: Yes with 1 copy engine(s)  Run time limit on kernels: Yes  Integrated GPU sharing Host Memory: No  Support host page-locked memory mapping: Yes  Alignment requirement for Surfaces: Yes  Device has ECC support: Disabled  Device supports Unified Addressing (UVA): Yes  Device PCI Bus ID / PCI location ID: 1 / 0 |

Tab 4.3. Parametry CUDA dla NVS 5400M.

Konfiguracje wejściowa programu wykonana na podstawie pliku *structure.cfg*:

|  |  |  |
| --- | --- | --- |
| dimX | 15 | 25 |
| dimY | 15 | 25 |
| dimZ | 15 | 25 |
| force | 1 | |
| forceType | ALL\_AROUND | |

Tab 4.4. Konfiguracja – dane wejściowe struktury.

Każda symulacja zostanie wykonana 100 krotnie. Zostanie wyciągnięty średni wynik spośród wszystkich testów. Pozwoli to na uzyskanie optymalnych wyników.

Jedynym zmiennym parametrem symulacji będzie ilość wątków przydzielonych na blok. Dzięki temu można będzie sprawdzić wpływ ilości wątków na wydajność symulacji. Na podstawie ilości wątków na blok oraz rozmiaru problemu wyznaczana jest ilość potrzebnych bloków wątków w siatce. Pojedynczy blok z wątkami jest nazywany grupą roboczą.

Rozdział 4 prezentuje jedynie testy wydajności. Nie będą poruszane tutaj sprawy modelowania defektów i graficzne przedstawienie rozwiązywanych problemów. Odkształcenia i ich modelowanie szerzej opisane zostało w osobnym rozdziale.

## Pomiar czasu

W celu uzyskania jak najdokładniejszego pomiaru czasu wykorzystany został wbudowany w biblioteki CUDA mechanizm. Opiera się on na zdarzeniach, które przechwytują dokładny czas na starcie i końcu okresu pomiarowego. Wbudowana funkcja na podstawie wyników subskrypcji tych dwóch zdarzeń określa ile upłynęło pomiędzy nimi czasu.

|  |
| --- |
| cudaEvent\_t start;  handleTimerError(**cudaEventCreate**(&start), START\_CREATE);  cudaEvent\_t stop;  handleTimerError(**cudaEventCreate**(&stop), STOP\_CREATE);    handleTimerError(**cudaEventRecord**(start, NULL), START\_RECORD);  /\*  \*\* OBLICZENIA  \*/  cudaDeviceSynchronize();  handleTimerError(**cudaEventRecord**(stop, NULL), STOP\_RECORD);  handleTimerError(**cudaEventSynchronize**(stop), SYNCHRONIZE);  handleTimerError(**cudaEventElapsedTime**(&msecTotal, start, stop), ELAPSED\_TIME); |

Kod 4.1. Pomiar czasu symulacji.

Powyższy przykład przedstawia szablon, jaki jest stosowany do pomiaru czasu. Zawiera wszystkie niezbędne funkcje do przechwytywania, synchronizacji i obliczania czasu wraz z obsługą błędów. Po zebraniu danych w taki sposób można przeprowadzić szacowanie uzyskanej wydajności. Wymaga to, oprócz zmierzonego czasu, również ręcznego obliczenia ilości operacji wykonanych w algorytmie. Dzięki temu możliwe będzie wyliczanie wielkości GFLOPS, czyli ilości operacji na sekundę.

|  |
| --- |
| float msecPerSimulation = msecTotal / nIter;    double flopsPerSimulation = 25.0 \* structure->atomsCount \* structure->atomsCount + 10.0 \* structure->atomsCount + 2 \* (structure->atomsCount + 256 - 1 )/ 256;    double gigaFlops = (flopsPerSimulation \* 1.0e-9f) / (msecPerSimulation / 1000.0f);    printf("\n\t\tPerformance= %.2f GFlop/s, Time= %.3f msec, Size= %.0f Ops, WorkgroupSize= %u threads/block\n",  gigaFlops,  msecPerSimulation,  flopsPerSimulation,  block.x \* block.y);  ///////////////// Wynik przykładowego wykonania powyższego kodu////////////////////  Performance= 14.68 GFlop/s, Time= 42.675 msec, Size= 626514778 Ops, WorkgroupSize= 256 threads/block |

Kod 4.2. Obliczanie wydajności uzyskanej przez algorytm.

Kod 4.2 jest przykładem obliczenia wydajności. Istotnym elementem jest wartość zaznaczona grubą czcionką. Jest to bardzo istotny parametr – średni czas dostępu do pamięci (w tym wypadku RAM) na GPU. Jest to wartość dokładna, określona przez deweloperów Nvidii, która znajduje się między innymi w przykładach dołączonych do CUDA Toolkit. Wynik wykonania kodu przedstawia wartość GFLOPS, czas wykonania pojedynczego przejścia algorytmu, ilość operacji, jakie zostały wykonane w trakcie obliczeń oraz ilość wątków pracujących w ramach jednego bloku.

## Testy sekwencyjne

Testy sekwencyjnego przebiegu algorytmu przy wykorzystaniu procesorów GPU zawsze są dyskusyjne. Na potrzeby analizowanych problemów ‘testy sekwencyjne’ będą uważane za punkt odniesienia, do obliczania przyspieszenia czy skalowalności. Nie jest tu zakładane, że operacja zostanie wykonana przez dokładnie jeden wątek na jednym multiprocesorze. Stosując nomenklaturę CUDA, testy sekwencyjne są uważane za wywołanie kernela z następującymi parametrami:

|  |
| --- |
| Kernel\_GPU<<< 1, 1 >>>(…); |

Kod 4.3. Parametry kerenla dla testów sekwencyjnych.

Takie wykonanie funkcji GPU sprawi, że zostanie ona wykonana przez 1 blok z 1 wątkiem w ramach jednego bloku. W czasie działania programu może jednak dojść do przełączania kontekstu pomiędzy wątkami oraz pomiędzy multiprocesorami. Niestety nie ma możliwości kontrolowania takich sytuacji.

|  |  |
| --- | --- |
| Wydajność [GFLOP/s] | 0.35 |
| Czas [ms] | 1768.511 |

Tab 4.5. Wyniki testów sekwencyjnych

## Testy zrównoleglenia

W tym rozdziale przedstawiony będzie wpływ rozmiaru grupy roboczej na wydajność i czas wykonywania symulacji. Wartością odniesienia będzie wynik uzyskany w rozdziale 4.2 podczas testów sekwencyjnych.

Wyk 4.1. Wykres zależności wydajności GPU od rozmiaru grupy roboczej wątków wewnątrz jednego bloku.

Wyk 4.2. Wykres zależności czasu wykonania na GPU od rozmiaru grupy roboczej wątków wewnątrz jednego bloku.

Zarówno wykres 4.1 jak i 4.2 przedstawiają charakterystykę karty graficznej NVS 5400M. Wbrew pozorom zwiększanie rozmiaru grupy roboczej wątków w pewnym momencie przestało przynosić coraz lepszą wydajność. Przy wartości 128 uzyskana została najlepsza wydajność obliczeniowa. Niewiele gorszy wynik można zaobserwować również w momencie ustawienia wielkości grupy roboczej na maksymalną, jaką obsługuje karta. Dla tych parametrów zostaną wykonane testy podczas optymalizacji. W powyższych testach ujawniona została również ciekawa cecha samego algorytmu. Im większy rozmiar danych wziętych do obliczeń tym lepszą wydajność udaje się osiągnąć. Takie testy wykonane przed rozpoczęciem docelowych obliczeń na nowym sprzęcie pozwolą zaznajomić się z charakterystyką karty i przystosować ustawienia do rozwiązywanego problemu.

Analizując wykres wykonany z wykorzystaniem struktury 25x25x25 można zauważyć, że dla dwóch pierwszych testów wydajność wynosi 0 GFLOP/s. Jest to związane z błędem jaki był wynikiem tych symulacji. Obliczenia zostały przerwane ze względu na zbyt długi czas wykonywania pojedynczego kernela w połączeniu z bardzo dużym obciążeniem karty graficznej. Jest to wbudowany mechanizm kart graficznych Nvidii: *Run time limit on kernels*. Aktualna konfiguracja użytego procesora GPU (Tab 4.3) pokazuje, że podczas symulacji ta funkcja była włączona, przez co wspomniane testy kończyły się błędem.

## Testy optymalizacji

W rozdziale 3.8 opisano metody optymalizacji, jakie zostały po kolei stosowane w programie w celu podniesienia wydajności obliczeń. W tym rozdziale zostanie przedstawiona analiza wpływu zastosowanych optymalizacji na wydajność. Korzystając z wyników otrzymanych w rozdziale 4.3 aktualne testy będą wykonywane przy rozmiarze grupy roboczej 128 oraz 1024. Biorąc pod uwagę charakterystykę GPU oraz rozwiązywanego problemu ustawienie tych wartości parametru skutkuje uzyskaniem najlepszej wydajności obliczeniowej.

Wyniki testów będą porównywane z podstawowym algorytmem równoległym, aby pokazać o ile udało się zwiększyć wydajność stosując kolejne elementy optymalizacji.

Pierwsza część optymalizacji zakładała znaczne usprawnienie pod względem ilości przebiegu pętli i podziale na wątki również pomocnicze funkcje uaktualniające dane.

Wyk 4.3. Wykres wzrostu wydajności po zastosowaniu pierwszego zestawu optymalizacji.

Wyk 4.4. Wykres zmiany czasu po zastosowaniu pierwszego zestawu optymalizacji.

Wykresy 4.3 oraz 4.4 pokazują, że odpowiednie pogrupowanie wykonywanych operacji może dać znaczące wyniki. Udało się uzyskać przyspieszenie o ok. 10 GFLOP/s. Przy rozmiarze grupy roboczej 128 wątków nadal można zaobserwować najlepsze wyniki wydajności. Czerwone oraz niebieskie słupki przedstawiają wyniki po optymalizacji. Słupki zielone oraz fioletowe prezentują wyniki symulacji równoległych przed optymalizacją. Aby reprezentacja danych była czytelniejsza wykorzystana została skala wykładnicza na osi wartości. Czas uwzględniony na powyższych wykresach pokazuje o ile dłużej trwają symulacje z wykorzystaniem znacznie większych struktur wejściowych. Porównując długość wykonywania obliczeń z osiągniętą wydajnością, można stwierdzić, że czas nie ma tu jak widać tak dużego znaczenia, ponieważ algorytm zachowuje się w miarę stabilnie i nawet przy dużym i długotrwałym obciążeniu karty graficznej osiągnięta wydajność jest porównywalna.

Drugi etap optymalizacji implementuje usprawnienia związane z wykonywaniem działań na liczbach zmiennoprzecinkowych oraz modyfikacje przy pomocy dyrektyw preprocesora, które rozwijają pętle.

Wyk 4.5. Wykres wzrostu wydajności po zastosowaniu drugiego zestawu optymalizacji.

Wyk 4.6. Wykres zmiany czasu po zastosowaniu drugiego zestawu optymalizacji.

Wykresy 4.5 oraz 4.6 tak jak podczas wcześniejszego etapu optymalizacji prezentują wyniki w porównaniu z podstawowym algorytmem. Również tym razem można zaobserwować wzrost wydajności o ponad 10 GFLOP/s. W porównaniu z wynikami osiągniętymi we wcześniejszym teście (Wyk. 4.3, 4.4) ta wydajność wzrosła jedynie o ok 2 GFLOP/s. Część sekwencyjna algorytmu staje się coraz większym procentem całości obliczeń, przez co coraz ciężej jest osiągać wyższe poziomy równoległości i lepszą wydajność .

Etap trzeci optymalizacji implementuje użycie rejestrów do przechowywania często używanych zmiennych w obliczeniach oraz redukcję ilości wykonywanych operacji w kernelu.

Wyk 4.7. Wykres wzrostu wydajności po zastosowaniu trzeciego zestawu optymalizacji.

Wyk 4.8. Wykres zmiany czasu po zastosowaniu trzeciego zestawu optymalizacji.

Wykresy 4.7 oraz 4.8 pokazały, że zastosowanie ostatnich operacji optymalizacji pozwoliło przekroczyć 30 GFLOP/s a co za tym idzie o ok 50% zwiększyć wydajność. Jest to zadowalający wynik biorąc pod uwagę, że przyczyniły się do tego jedynie operacje modyfikujące funkcje wykonywane na GPU, a nie zmiana sprzętu wykorzystywanego do obliczeń. Tak zoptymalizowany algorytm, może posłużyć do dalszych symulacji z wykorzystaniem sprzętu dedykowanego do obliczeń wysokiej wydajności.

## Przyspieszenie

Wielkością dobrze prezentującą uzyskane wyniki jest przyspieszenie. Pokazuje ile razy udało się uzyskać lepszą wydajność względem sekwencyjnego wykonania algorytmu. Jak wspomniano w rozdziale 4.2 przy wykorzystaniu GPU ciężko mówić o algorytmie w pełni sekwencyjnym, to biorąc pod uwagę założenia z tego rozdziału wykres przyspieszenia przedstawiony został na wykresie 4.9.

Wyk 4.9. Wykres przyspieszenia uzyskanego w kolejnych etapach zrównoleglania kerneli.

Wykres 4.9 prezentuje osiągnięte przyspieszenie w kolejnych krokach modyfikacji kerneli. Ostatecznie udało się osiągnąć ponad 86 krotne przyspieszenie względem sekwencyjnego wykonania programu. Ciężko porównywać te wyniki z osiągnięciami, które mogły być uzyskane przez procesory CPU ze względu na zupełnie inną architekturę GPU.

# Model algorytm-urządzenie z wykorzystaniem wielu procesorów GPU

Wykorzystując wiele kart graficznych do rozwiązania problemu największym problemem staje się podział danych pomiędzy urządzenia. Równie ważną operacją w tym wypadku staje się synchronizacja podzielonych danych. W skrajnych przypadkach może znacznie spowolnić obliczenia i sprawić, że pomimo wykorzystania wielu urządzeń, wyniki nie są dużo lepsze.

## Podział zadania

Sposób implementacji opisany w rozdziale 3.7 umożliwił bezproblemowy podział danych poprzez przydzielenie każdemu urządzeniu części struktury atomów. Dzięki takiemu podziałowi nie będzie dochodziło do wyścigu pomiędzy wątkami, ponieważ każdy z nich będzie operował na osobnym, niezależnym zestawie danych. Nie ma również potrzeby stosowania sesji krytycznych, aby aplikacja była wielowątkowo bezpieczna. Zadaniem, jakie należało w tym wypadku rozwiązać to synchronizacja danych. Aby obliczenia były spójne, każdy atom do obliczeń powinien wykorzystać sąsiadujące z nim molekuły. Nie są jednak potrzebne wszystkie, lecz tylko te, które leżą w promieniu odcięcia. Mając na uwadze ten fakt, każda karta graficzna oprócz części przestrzeni wynikającej z podziału, dostawała część przestrzeni o promieniu nieco większym od promienia odcięcia. Sprawiło to, że atomy leżące na krawędziach, miały kontakt z atomami znajdującymi się na urządzeniu obok. Obliczenia natomiast prowadzone były jedynie dla atomów, które przydzielone były przy podziale struktury. Takie rozwiązanie sprawia, że synchronizacja w czasie obliczeń nie jest wymagana w trakcie krótkich symulacji. Atomy brzegowe nie zmienią w tym czasie swojej pozycji na tyle znacząco, aby konieczne było ich synchronizowanie pomiędzy urządzeniami. Taka synchronizacja powinna nastąpić w czasie symulacji, gdzie struktura ulega znacznemu odkształceniu i każda iteracja znacząco wpływa na ułożenie atomów. Nie jest jednak wymagane synchronizowanie wszystkich atomów z każdej karty, a jedynie wspomniane części powiększające strukturę o rozmiar promienia odcięcia tak, aby były spójne pomiędzy wszystkimi urządzeniami. Aby precyzyjnie określić moment synchronizacji, można mierzyć przemieszczenie atomu w czasie. Zbyt duża zmiana położenia w krótkim odcinku czasowym oraz duże przyspieszenie uzyskane przez atom, może wskazywać na konieczność synchronizacji.

Wykres 5.1 przedstawia porównanie czasu wykonania algorytmu na 1 kardzie graficznej oraz na 4 kartach. Wyraźnie widać, że obydwa wykresy są do siebie bardzo zbliżone, przesunięte jedynie o pewnie wektor. Świadczy to o tym, że rozdzielenie zadania skróciło jedynie czas obliczeń, natomiast charakterystyka pozostała bez zmian. Wykres 5.2 pokazuje jak kształtowała się wydajność obliczeń dla różnych rozmiarów struktur wejściowych. Od pewnego progu rozmiaru danych wydajność nieco spadła. Może być to spowodowane koniecznością sięgania do wolniejszej pamięci. Szybka pamięć i rejestry mogły zostać wykorzystane i odczyt danych trwał w tym momencie dłużej. Pomimo tego wydajność utrzymuje się na wysokim, zadowalającym poziomie, a zastosowanie więcej niż jednego urządzenia GPU przyniosło oczekiwane wyniki.

Wyk 5.1. Porównanie czasów wykonania dla zmiennego rozmiaru zadania wykorzystując 1 oraz 4 karty graficzne

Wyk 5.2 Porównanie wydajności dla zmiennego rozmiaru zadania z wykorzystaniem 1 oraz 4 kart graficznych

## Skalowalność

Skalowalność jest zdolnością systemu, sieci lub procesu do radzenia sobie z rosnącą ilością pracy w skończonym czasie lub zdolnością do rozszerzania się, aby poradzić sobie ze wzrostem ilości pracy. Z zagadnieniem skalowalności wiążą się pojęcia takie jak: skalowalność silna lub słaba, przyspieszenie skalowalne, efektywność zrównoleglenia czy wydajność obliczeń. Każdy z wyznaczanych parametrów różni się i pomaga określić specyficzne cechy algorytmu lub też rozwiązywanego problemu.

Istotnymi miarami wydajności obliczeń równoległych są przyspieszenie (4) oraz efektywność zrównoleglenia (5). Aby uzyskać idealne wyniki należałoby osiągnąć linowe przyspieszenie i 100% efektywności. [10]

(4)

(5)

Ts jest czasem rozwiązania zadania najlepszym algorytmem sekwencyjnym na pojedynczym procesorze. W przypadku GPU będzie to rozpatrywane trochę inaczej ze względu na to, że nie ma możliwości wykonania zadania na pojedynczym, wybranym procesorze. Taki podział zadania jest automatyczny. Można jednak manipulować ilością wątków, które biorą udział w obliczeniach. TII(p) jest czasem rozwiązania zadania algorytmem równoległym na *p* procesorach.

Wyk 5.3 Przyspieszenie uzyskane przy użyciu 4 procesorów GPU.

Wyk 5.4 Efektywność zrównoleglenia dla 4 procesorów GPU.

Przyspieszenie uzyskane na 4 kartach graficznych zaprezentowane na wykresie 5.3 ma na początku charakter liniowy. W pewnym momencie wykres zaczyna kierować się bardziej poziomo. Widać wyraźnie, w którym miejscu narzut części sekwencyjnej algorytmu uniemożliwia uzyskanie liniowego przyspieszenia. Jest to zgodne z prawem Amdahla [11], które mówi, że przyspieszenie obliczeń równoległych z wykorzystaniem wielu procesorów jest ograniczone przez czas potrzebny na wykonanie sekwencyjnej części algorytmu. Można stwierdzić, że program wykazuje silną skalowalność, ponieważ przy zwiększającej się liczbie zasobów oraz stałym rozmiarze problemu uzyskiwane były coraz lepsze wyniki. Taka analiza pozwala stwierdzić jak zachowuje się część programu podlegająca zrównolegleniu. [12]

Analiza efektywności przedstawiona na wykresie 5.4 potwierdza wnioski wyciągnięte w rozdziale 4 (wykres 4.1). Efektywność obliczeń uzyskana dla 128 wątków na 1 blok wynosi ok 80%. Daje to najlepszy stosunek uzyskanej mocy obliczeniowej do osiągniętej efektywności. Dalsze zwiększanie ilości wątków rzutuje na znaczy spadek wydajności. Można uznać, że program jest skalowalny równolegle, ponieważ funkcja efektywności obliczeń jest ograniczona z dołu przez prostą leżącą w okolicy 20%. Wyniki nie spadną poniżej tej wartości, ponieważ zasoby karty graficznej nie pozwolą na ustawienie większej liczby wątków na blok.

Istotną miarą wydajności obliczeń równoległych jest również przyspieszenie skalowalne. Uzyskuje się je analizując dla danej liczby *p* procesorów, zadanie p-krotnie większe od zadania dla pojedynczego procesora. [10]

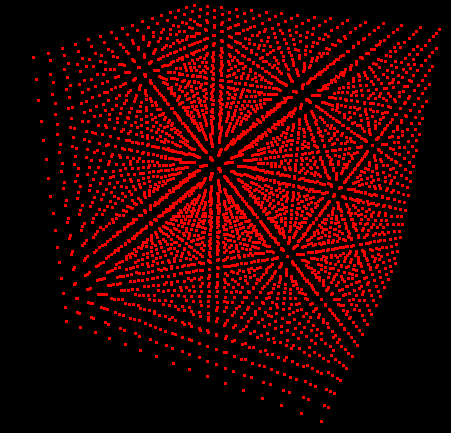
Wyk 5.5. Przyspieszenie skalowalne dla 4 procesorów GPU.

Wykres 5.5 przedstawia przyspieszenie skalowalne uzyskane na 4 procesorach GPU. Przy każdorazowym podwojeniu liczby wątków w obliczeniach, podwojony został również rozmiar zadania. Jak widać powstała funkcja jest zbliżona do liniowej. Odchylenia od początkowej wartości odniesienia są niewielkie. Pokazuje to, że program bardzo dobrze poradził sobie z coraz większym zadaniem wykorzystując więcej zasobów sprzętowych. Na podstawie tego wykresu można wnioskować jak duży rozmiar danych może być z powodzeniem przetworzony przy aktualnej implementacji zrównoleglenia zadania. Jak widać nawet największy rozmiar zadania, jaki był analizowany w trakcie testów, z powodzeniem może być przetworzony bez straty wydajności. Jest to analiza tzw. słabej skalowalności. [12]

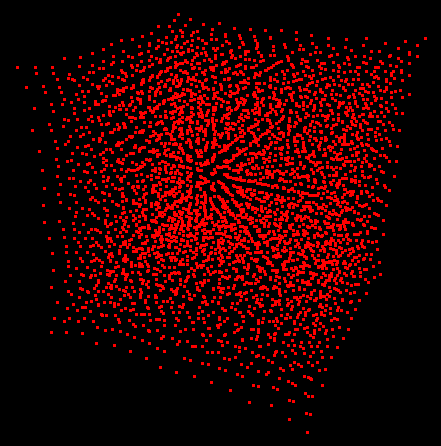
# Modelowanie defektów

Defekty powstałe podczas działania siły z zewnątrz na układ atomów to odkształcenia. Tego typu defekty zostaną przedstawione w rozdziale 6. Struktura, która została poddana działaniu siły, była za każdym razem identyczna. Pozwoliło to otrzymać skalę porównawczą jak różne efekty można uzyskać stosując różne rodzaje siły.

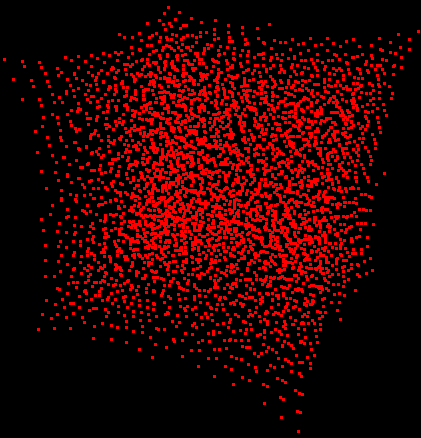
Nieodłączną częścią symulacji jest minimalizacja naprężeń wewnątrz, układu, aby doprowadzić go do stanu równowagi. Struktura początkowa nie jest w stanie równowagi. Poniższe wyniki pokazują jak struktura zachowuje się bez działania siły zewnętrznej.



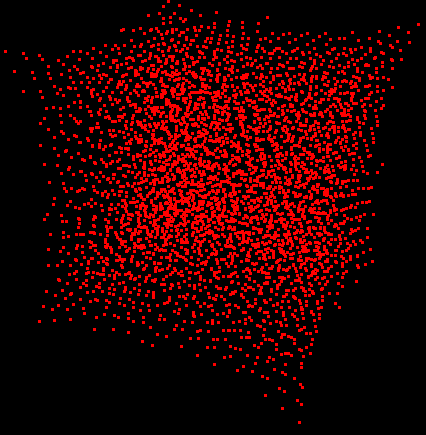
Rys 5.1. Minimalizacja potencjału faza 1



Rys 5.2. Minimalizacja potencjału faza 2

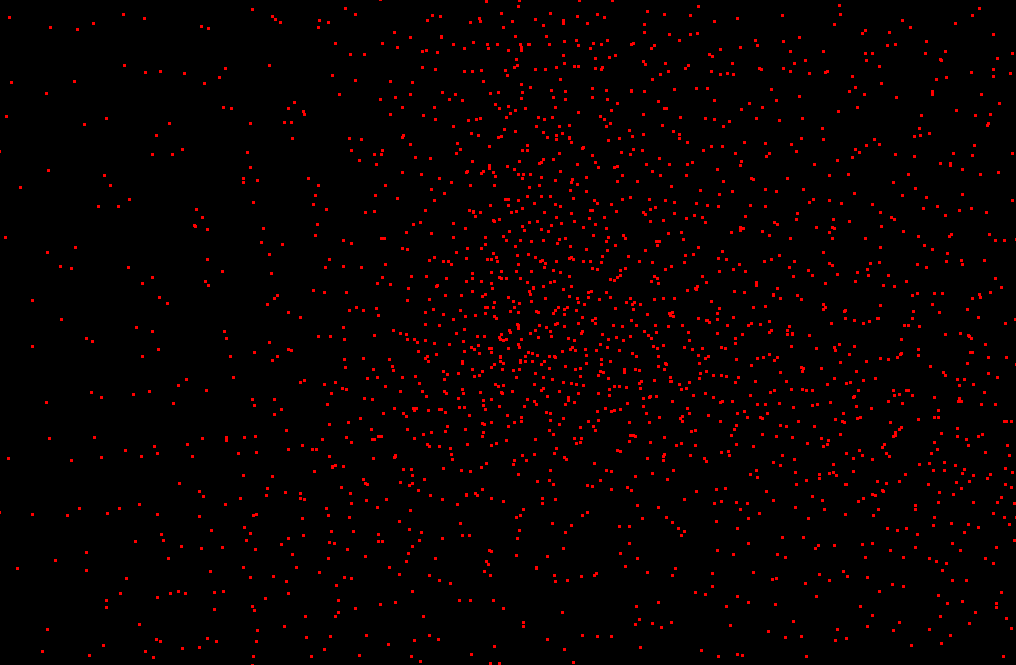


Rys 5.3. Minimalizacja potencjału faza 3



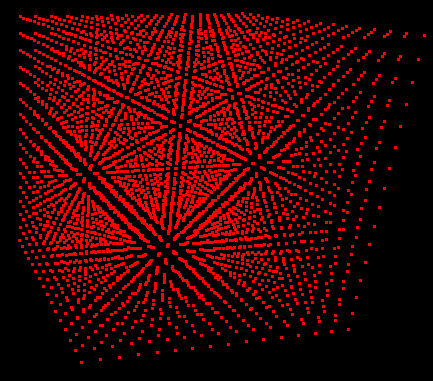
Rys 5.4. Minimalizacja potencjału faza 4

Rysunki od 5.1 do 5.4 przedstawiają wyniki symulacji minimalizacji naprężeń w strukturze bez udziału siły zewnętrznej. Można zauważyć, że największe siły działały w wewnętrznej części struktury, o czym świadczy końcowy kształt na Rys 5.4. Struktura lekko zapadła się do środka. Pomimo, że atomy były ułożone równomiernie w fazie początkowej Rys. 5.1, to ich ułożenie na końcu symulacji jest bardzo nieregularne. Rys 5.5. Można zauważyć skupiska molekuł, gdzie naprężenia były większe jak również bardziej rozproszone w przestrzeni. Warto zauważyć, że atomy były w ciągłym ruchu, nawet już po etapie minimalizacji. Większość atomów w ciąż drgała i przemieszczała się raz w jedną raz w drugą stronę. Daje to poczucie, że ciągłe powiązania pomiędzy atomami utrzymują strukturę w stabilności.

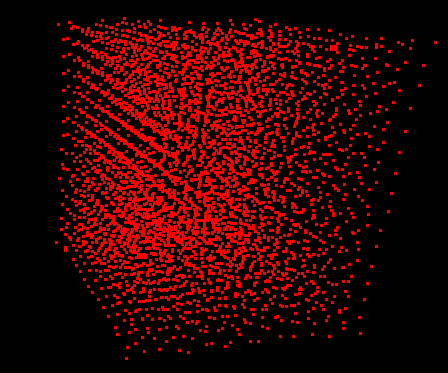


Rys 5.5. Nieregularne ułożenie atomów w strukturze po minimalizacji naprężeń.

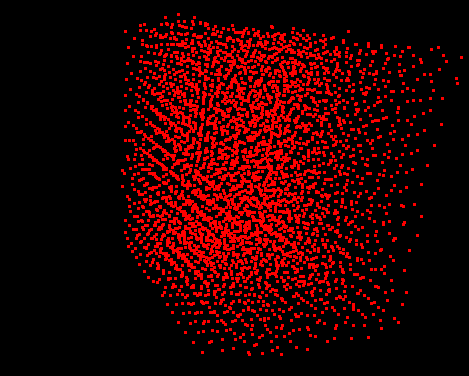
Drugą symulacją, jaka została przeprowadzona było przyłożenie siły z lewej strony struktury tak, aby ze stałą wartością ściskała strukturę. W tym czasie przeciwna strona była nieruchomo zablokowana. Uzyskany został w ten sposób prasy jednostronnej.



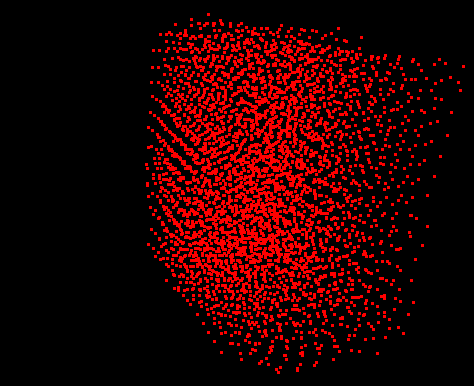
Rys 5.6 Symulacja prasy lewostronnej faza 1



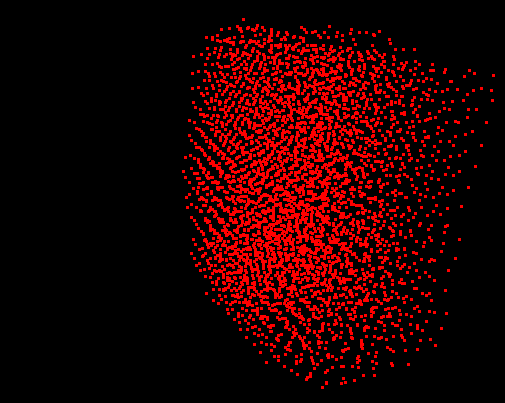
Rys 5.7. Symulacja prasy lewostronnej faza 2



Rys 5.8. Symulacja prasy lewostronnej faza 3



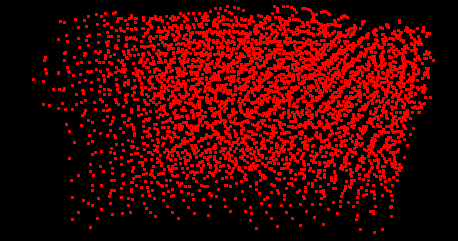
Rys 5.9. Symulacja prasy lewostronnej faza 4



Rys 5.10. Symulacja prasy lewostronnej faza 5

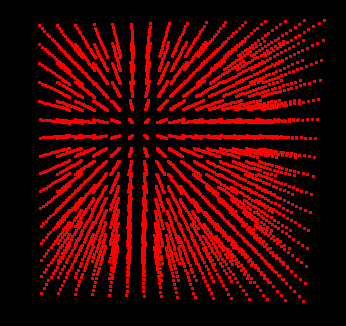
Wynikiem symulacji prasy jest struktura o spłaszczonym kształcie Rys 5.10. Wyraźnie widać zagęszczenie atomów od strony działającej siły. Struktura rozciągnęła się również w górę i w dół. Zbliżające się do siebie atomy nie miały w pewnym momencie już miejsca, więc zaczęły się przemieszczać w prostopadłym kierunku. Bardzo dobrze odzwierciedla to faktyczne zachowanie spłaszczanego materiału. Analizując prawą stronę struktury, która pozostawała nieruchomo, można zaobserwować przemieszczenie się atomów względem początkowego położenia. Świadczy to o ciągłym procesie minimalizacji naprężeń, jaki zachodził w strukturze, oraz wpływ siły zewnętrznej przekazywanej pomiędzy sobą przez oddziałujące na siebie atomy.

Rzut struktury z innej perspektywy potwierdza wyciągnięte powyżej wnioski Rys 5.11.

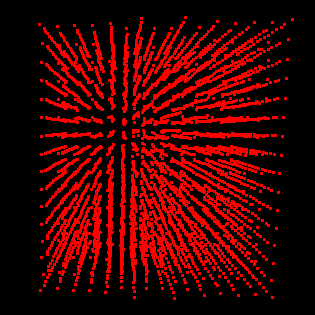


Rys 5.11. Zmiana rzutu struktury z fazy 5 symulacji prasy lewostronnej.

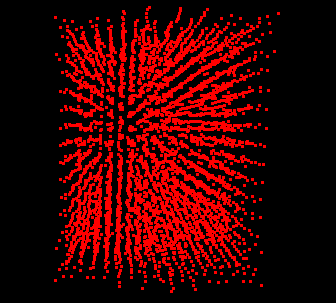
Ostatnią symulacją., jaka została przeprowadzona było przyłożenie siły z obydwu stron struktury. Pozwoliło to na zasymulowanie prasy obustronnej.



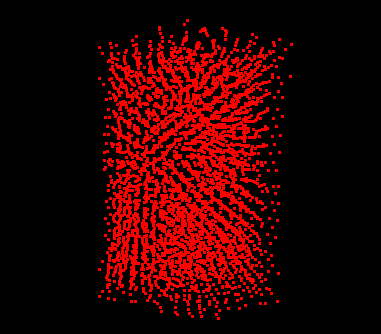
Rys 5.12. Symulacja prasy obustronnej faza 1



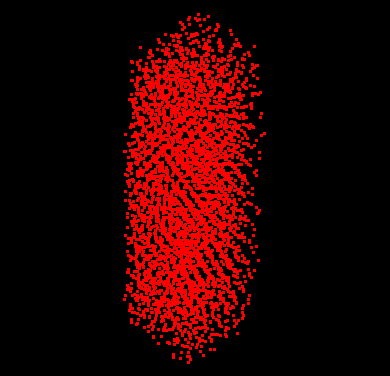
Rys 5.13. Symulacja prasy obustronnej faza 2



Rys 5.14. Symulacja prasy obustronnej faza 3

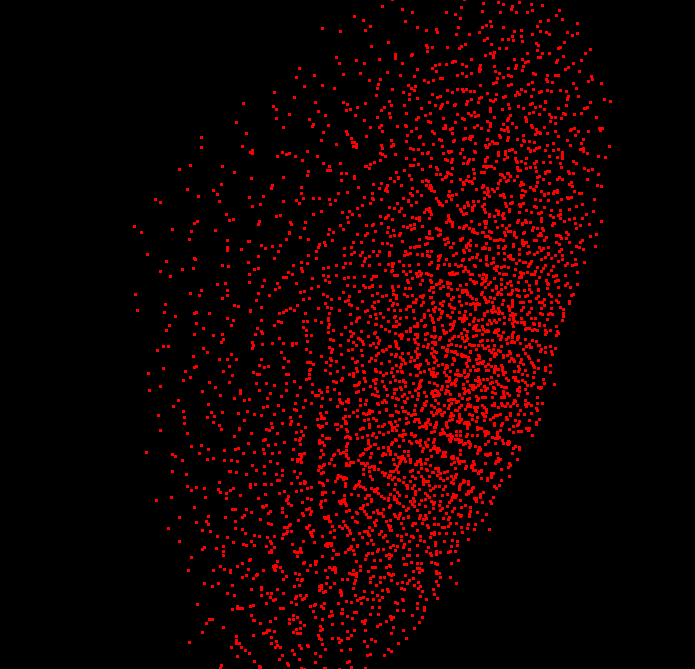


Rys 5.15. Symulacja prasy obustronnej faza 4



Rys. 5.16. Symulacja prasy obustronnej faza 5

Symulacja przebiegła podobnie do lewostronnej prasy z tą różnicą, że struktura jest teraz bardziej symetryczna. Wyraźniej widać miejsce spłaszczenia struktury Rys. 5.16. Po zakończeniu fazy 5 struktura nie ulegała już znaczącym deformacjom. Ułożenie atomów było już na tyle ciasne, że nie można było dostrzec jedynie niewielki ruch w górę oraz w dół. Rzut struktury z innego konta przedstawiono na rysunku poniżej.



Rys 5.17. Rzut z góry na strukturę po symulacji prasy obustronnej.

Podsumowanie wszystkich symulacji zostało zamieszczone na rysunkach 5.18 do 5.20. Zawierają one zestawienie wszystkich powstałych struktur, aby łatwo można było je porównać.

|  |
| --- |
|  |

Rys 5.18. Symulacja 1

|  |
| --- |
|  |

Rys 5.19. Symulacja 2

|  |
| --- |
|  |

Rys 5.20. Symulacja 3

# Podsumowanie

Szczegółowa analiza oprogramowania oraz optymalizacja już istniejącego i działającego kodu jest bardzo istotna. Szczegóły optymalizacji przedstawione w rozdziale 3 oraz testy wykonywane dla poszczególnych jej etapów w rozdziale 4 pokazały, że można uzyskać duży przyrost wydajności dogłębnie analizując problem. Przygotowane oprogramowanie wspierało wiele urządzeń GPU, przez co zwiększało możliwości i zakres wykorzystania go. Analiza skalowalności z rozdziału 5 pokazała, że aplikacja bardzo dobrze radzi sobie ze zwiększającym się rozmiarem zadania, a jeszcze lepiej, gdy dostępne są coraz większe zasoby sprzętowe. Program jest dobrze skalowalny i efektywny. Wykonywanie testów na różnych maszynach i na różnych rodzajach procesorów GPU pokazuje, że aplikacja jest łatwo przenaszalna. Krótka analiza charakterystyki danego procesora GPU pozawala na uzyskanie bardzo wysokiej wydajności bez modyfikowania aplikacji. Porównanie procesorów graficznych używanych w sprzętach przeznaczonych dla użytkowników domowych z procesorami dedykowanymi do obliczeń wysokiej wydajności pokazuje, że technologia CUDA jest bardzo uniwersalna pomimo wymogu wykorzystania procesorów firmy Nvidia.

Aktualny stan implementacji nie pozwala rozwiązywać różnorodnych zadań, natomiast zastosowanie modułowej budowy sprawia, że aplikacja jest łatwa w modyfikacji, co pozwala na dostosowanie jej do własnych potrzeb w stosunkowo łatwy sposób. Daje to możliwość rozwoju i łączenia zadań obliczeniowych, aby możliwe było rozwiązywanie bardziej przekrojowych problemów. Dobra skalowalność i możliwość zintegrowania wielu procesorów GPU odkrywa przed programistą jeszcze większe możliwości.

Wizualizacja wyników oraz śledzenie w czasie rzeczywistym przeprowadzanych symulacji w łatwy sposób pozwala zweryfikować i przeanalizować otrzymywane rezultaty. Umożliwiło to przeprowadzenie symulacji deformacji struktury w rozdziale 6 wraz z graficzną analizą otrzymanych wyników.

# Bibliografia

[1] Bachniak D., Modelowanie defektów strukturalnych w skali nano z wykorzystaniem metody statyki molekularnej w heterogenicznych architekturach sprzętowych, Praca magisterska, Akademia Górniczo-Hutnicza, 2010.

[2] Liu W., Schmidt B., Voss G., Müller-Wittig W., Molecular Dynamics Simulations on Commodity GPUs with CUDA, High Performance Computing – HiPC 2007,Lecture Notes in Computer Science Volume 4873, 2007, pp 185-196.

[3] Nowak T., Modelowanie defektów strukturalnych w skali nano z wykorzystaniem akceleratorów obliczeń GPGPU, Praca inżynierska, Akademia Górniczo-Hutnicza, 2012.

[3] https://developer.nvidia.com (05.04.2014)

[5]https://computing.llnl.gov/ (05.04.2014)

[6] http://en.wikipedia.org/wiki/POSIX\_Threads (05.04.2014)

[7]http://wakespace.lib.wfu.edu/bitstream/handle/10339/38561/Proctor\_wfu\_0248M\_10427.pdf (10.08.2014)

[9] <http://en.wikipedia.org/wiki/Molecular_dynamics>

[10] [www.metal.agh.edu.pl/~banas/OWW/OWW\_W12\_Skalowalnosc.pdf](http://www.metal.agh.edu.pl/~banas/OWW/OWW_W12_Skalowalnosc.pdf) (29.11.2014)

[11] <http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Amdahl_s_law.html> (29.11.2014)

[12] <https://support.scinet.utoronto.ca/wiki/index.php/Introduction_To_Performance> (29.11.2014)

1. http://www.nvidia.com/ (05.04.2014) [↑](#footnote-ref-1)
2. API – eng. application programming interface [↑](#footnote-ref-2)
3. Windows API – zestaw wbudowanych interfejsów programistycznych dostępnych dla systemów Microsoft Windows [↑](#footnote-ref-3)