POLITECHNIKA WARSZAWSKA
Wydział Elektroniki i Technik
Informacyjnych
Instytut Informatyki

PRACA DYPLOMOWA MAGISTERSKA

Przemysław Szurmak

Rozwój nowych narzędzi do komputerowego projektowania leków

Opiekun naukowy:
prof. dr hab. inż. Jan Mulawka

Ocena pracy

Data i podpis przewodniczącego
Komisji Egzaminacyjnej

Warszawa, 15 września 2016

Kierunek: Informatyka

Specjalność: Inżynieria Systemów Informacyjnych

Data urodzenia: 09.04.1991r.

Data rozpoczęcia studiów: 24.02.2014r.

nr albumu: 236 605

Życiorys

Urodziłem się 3 stycznia 1991r. w Warszawie. W latach 2004-2007 uczęszczałem do Publicznego Gimnazjum im. Krzysztofa Kamila Baczyńskiego w Warszawie. Następnie, w latach 2007-2010 kontynuowałem naukę w XLI Liceum Ogólnokształcącym im Joachima Lelewela w Warszawie, w klasie o profilu matematyczno-fizycznym. W 2010 roku rozpocząłem studia wyższe na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej na kierunku Elektronika, Informatyka i Telekomunikacja. W marcu 2012 r. rozpocząłem pracę w Instytucie Łączności w Warszawie, przy projekcie LDUiTV (Lokalni Dostawcy Usług w Interaktywnej Telewizji), podczas którego powstała aplikacja będąca tematem mojej pracy inżynierskiej. W roku 2012 wybrałem specjalność Teleinformatyka i Zarządzanie w Telekomunikacji i w tej specjalności zdobyłem tytuł inżyniera w lutym 2014 r.. Od razu po obronie rozpocząłem studia magisterskie na kierunku Informatyka, kontynuując edukację w tej samej placówce. Wybrałem specjalizację Inżynieria Systemów Informacyjnych. W latach 2014-2015 odbyłem jednoroczny wyjazd zagraniczny z programu Erasmus+ do Luksemburga gdzie także studiowałem Informatykę. Aktualnie, jestem w trakcie rozwijania własnej działalności gospodarczej w ramach której stworzyłem aplikację, która wygrała główną nagrodę w organizowanym przez „Europejską Agencję Kosmiczną”.

Przemysław Szurmak

EGZAMIN DYPLOMOWY

Złożył egzamin dyplomowy w dniu .....

z wynikiem .....

Ogólny wynik studiów: .....

Dodatkowe wnioski i uwagi Komisji.....

.....

Streszczenie

Bioinformatyka odgrywa ważną rolę w naukach przyrodniczych. Jedna z jej gałęzi – komputerowo-wspomagane projektowanie leków (ang. skrót CADD) – daje praktyczne wskazania dla projektowania i odkrywania nowych – lepszych i bezpieczniejszych – leków. CADD obejmuje wiele różnych technik, takich jak dokowanie, przesiew wirtualny czy ilościowe zależności struktura-aktywność (QSAR). Ta ostatnia zajmuje się budowanie równań korelujących aktywność leków z ich strukturą przedstawioną w postaci zmiennych zwanych deskryptorami molekularnymi. Naukowcy cały czas poszukują nowych deskryptorów, lepiej modelujących związki chemiczne. Ważnym i obiecującym typem takich deskryptorów są miary chiralności typu Sinister-Rectus Chirality Measures (^{SR}CM). Jakkolwiek, jedyny dostępny program do obliczeń ^{SR}CM jest bardzo wolny, co znacząco ogranicza zastosowanie ^{SR}CM przez społeczność QSAR. Wobec powyższego podjęto próbę znalezienia nowego algorytmu obliczania ^{SR}CM (z użyciem algorytmu genetycznego) oraz implementacji tegoż w postaci wydajnego i nowoczesnego programu komputerowego. Wynikiem tych wysiłków jest napisany w C++ program Chirmes. Przeprowadzone badania wykazały, że Chirmes uzyskuje poprawne (w rozsądnych granicach błędu) wyniki obliczeń miar chiralności, przy czym jest znacznie szybszy niż dotychczasowe oprogramowanie. Co więcej, postanowiono sprawdzić zastosowanie platformy CUDA do akceleracji algorytmów genetycznych – prosta implementacja, którą stworzono nie przyniosła znacznej poprawy wydajności, jednakże stanowi bazę do dalszego rozwoju. Do dodatkowych zalet programu Chirmes należy obsługa wielu formatów plików chemicznych oraz możliwość pracy na różnych systemach operacyjnych. W pracy opisano najpierw chemiczne i informatyczne aspekty podjętego zagadnienia. Następnie przedstawiono szczegóły implementacji wraz z wynikami testów i potencjalnymi dalszymi kierunkami rozwoju.
Słowa kluczowe: projektowanie leków, cadd, qsar, srcm, algorytmy genetyczne, cuda

Abstract

Development of new tools for Computer-Aided Drug Design

Bioinformatics plays an important role in natural sciences. One of its branches – Computer-Aided Drug Design (CADD) – gives practical insights for designing and discovering of novel – better and safer – drugs. The CADD encompasses many different techniques like docking, virtual screening and quantitative structure-activity relationships (QSAR). The latter deals with building equations relating drug activities and their structures represented by variables called molecular descriptors. Scientists have been continuously looking for new descriptors that allow better modeling of chemical compounds. An important and promising type of such descriptors are Sinister-Rectus Chirality Measures (SRCM). However, the only so far available software for ^{SR}CM calculation is very slow, and this impedes wider application of ^{SR}CM by QSAR community. Therefore, an attempt to develop a novel algorithm for calculation of ^{SR}CM (using Genetic Algorithm) and to implement it in an efficient and modern computer program was made. The result of these efforts is Chirmes, written in C++. Performed tests have shown that Chirmes gives correct (within reasonable error) results of ^{SR}CM calculations and performs way faster than the so far available software does. Furthermore, basic implementation of CUDA acceleration for genetic algorithm was created to find out if it provides any significant improvement in performance. Although speed up was not very large it is a good basis for future development. Additional advantages of Chirmes include using many common chemical file formats and the possibility to work under different operating systems. The thesis describes first chemical and computational background behind the tackled problem. Then details of the implementation are presented, along with the test results and future prospects.
Keywords: computer-aided drug design, qsar, srcm, genetic algorithms, cuda

Spis treści

1. Wstęp

1.1. Metody bioinformatyki

Wśród licznych dziedzin aktywności ludzkiej, które wiele zawdzięczają powstaniu i rozwojowi informatyki w XX wieku, są również szeroko pojęte nauki przyrodnicze. Stwierdzenie to nie odnosi się jedynie do korzystania z możliwości komputerów do przetwarzania, przechowywania i udostępniania informacji zdobywanych w badaniach naukowych, ale dotyczy przede wszystkim aktywnego zastosowania mocy obliczeniowych do prowadzenia tych badań.

Początki bioinformatyki - tym terminem zwykło się określać powstałą w ten sposób interdyscyplinarną gałąź wiedzy – związane są bezpośrednio z komputerową analizą sekwencji z jakich zbudowane są kwasy nukleinowe i białka (biologia molekularna), których dostępna obecnie ilość gwałtownie rosła od lat 70 zeszłego stulecia [1]. W dzisiejszych czasach, metody informatyczne znajdują zastosowanie również w wielu innych dziedzinach nauk o życiu, począwszy od ekologii i biologii ewolucyjnej, przez analizę organizmów żywych jako systemów (biologia systemów), aż po badania związane z symulacją poszczególnych procesów biochemicznych zachodzących w organizmach [2, 3]. Te ostatnie są unikalnym miejscem styku nie tylko biologii i informatyki ale także chemii i fizyki, gdyż na najniższym poziomie zjawiska biologiczne są specjalną (szczególnie skomplikowaną) formą zjawisk chemicznych i fizycznych.

Poza znaczeniem poznawczym, badania takie mają bardzo ważną rolę praktyczną.

Współczesna medycyna jest w znacznej mierze oparta na farmakologii, czyli manipulowaniu funkcjami organizmu pacjenta za pomocą cząsteczek chemicznych (leków) w celu osiągnięcia określonego skutku terapeutycznego. Zrozumienie na elementarnym poziomie przyczyn choroby i mechanizmu działania leków umożliwia racjonalne tworzenie nowych substancji aktywnych. Dziedziną bioinformatyki stosowaną w tym celu jest komputerowo-wspomagane projektowanie leków (ang. computer-aided drug design, CADD).

Metody CADD należą do podstawowego warsztatu chemii leków zarówno w środowisku akademickim jak i przemysłowym. Dzięki nim możliwe jest szybsze, skuteczniejsze i tańsze tworzenie nowych substancji leczniczych.

Rozwój narzędzi komputerowo-wspomaganego projektowania leków jest w świetle powyższego zadaniem o dużym potencjale praktycznym. Rozwój ten odbywa się niejako dwutorowo: z jednej strony specjaliści nauk przyrodniczych opracowują teorie i hipotezy, które są podstawą narzędzi CADD, z drugiej zaś informatycy implementują je w postaci poprawnie działających programów komputerowych. W takiej współpracy, zadanie informatyków nierzadko jest czymś więcej niż stworzeniem szablonowego kodu, ale wymaga również pewnej pracy oryginalnej, szczególnie jeśli otrzymane mają być narzędzia jak najszerzej wykorzystujące potężne moce obliczeniowe współczesnych komputerów.

1.2. Cel i struktura pracy

Obliczenia miar ${}^{SR}CM$ są jednym z zagadnień ważnych dla metodologii QSAR, będącej kluczową techniką CADD (szerzej zostaje to wyjaśnione w dalszych rozdziałach). Do tej pory dostępna była tylko jedna aplikacja służąca do liczenia miar ${}^{SR}CM$: program CHIMEA [4] napisany około 2010 roku. Niestety, aplikacja została przygotowana w sposób archaiczny bez uwzględnienia możliwości współczesnych komputerów PC. Przykładowo, szybkość działania na komputerze wyposażonym w najnowszy procesor Intel Core i7 6700k nie różni się od prędkości dla Intel Core2Duo E6400[1] z roku 2007. Dalej, program ten obsługuje tylko jeden format plików chemicznych (format PDB [5]), co znacznie utrudnia korzystanie mniej biegłym komputerowo użytkownikom.

Niniejsza praca dotyczy zagadnienia z pogranicza informatyki, biochemii i komputerowo-wspomaganego projektowania leków. Wobec powyżej opisanych faktów, celem podjętych wysiłków było wynalezienie efektywnego algorytmu liczącego miary chiralności ${}^{SR}CM$ oraz napisanie nowego programu implementującego ten algorytm, który:

a) obsługiwałby powszechnie używane formaty chemiczne

b) potrafił wykorzystać zasoby nowoczesnych komputerów

c) szybciej rozwiązywałby problem optymalizacyjny (minimalizacja odległości, znajdowanie najlepszego nałożenia enancjomerów).

[1] Według serwisu `cpubenchmark.net` wykonującego porównawcze testy wydajności procesor Core2Duo osiąga wynik 1296 punktów przy 10985 dla Core i7

W ramach dyskusji o problemie, autor postanowił wykonać cel pracy przez użycie algorytmów genetycznych, a poza implementacją na klasycznym procesorze CPU, spróbować implementacji na procesorze graficznym GPU.

W związku z interdyscyplinarnym charakterem podjętego tematu, część literaturowa dysertacji podzielona jest na rozdział poświęcony chemicznym aspektom pracy, gdzie opisane zostają najpierw metody komputerowo-wspomaganego projektowania leków i problematyka miar chiralności (Rozdział 2) oraz rozdział poświęcony zagadnieniom z zakresu nauk informatycznych, które leżą u podstaw proponowanej implementacji (Rozdział 3). Dalej następuje opis implementacji (Rozdział 4), w tym na GPU (Rozdział 4.4), a także weryfikacja programu i opis zastosowania praktycznego (Rozdział 5).

2. Aspekty chemiczne zagadnienia

2.1. Komputerowo-wspomagane projektowanie leków

Wspomniane we wstępie, komputerowo-wspomagane projektowanie leków weszło już na trwałe do kanonu metod badawczych współczesnej chemii leków. CADD umożliwia racjonalne projektowanie nowych substancji leczniczych, i choć nie zastępuje prac eksperymentalnych pozwala na znaczne obniżenie kosztów, a także czasu poszukiwania nowych leków. Wśród licznych metodologii CADD najważniejsze znaczenie mają opisane pokrótce w pracy dokowanie molekularne, przesiew wirtualny i techniki QSAR. W niniejszej pracy nacisk położony jest na te ostatnie, a konkretnie na miary chiralności jako obiecujących deskryptorach w modelowaniu QSAR. Miary te, z uwagi na powszechność zjawiska chiralności w molekułach o znaczeniu medycznym, są istotnym kierunkiem badań podstawowych w CADD.

2.1.1. Tradycyjne poszukiwanie leków

Aby zrozumieć doniosłość wkładu bioinformatyki we współczesne poszukiwanie nowych leków należy spojrzeć krótko na tradycyjny sposób prac nad nowymi substancjami aktywnymi. Do XIX wieku lekami były głównie preparaty roślinne, bądź pochodzenia zwierzęcego, których aktywność biologiczną stwierdzano głównie w wyniku przypadkowych obserwacji empirycznych. Wraz z rozwojem chemii organicznej w XIX stuleciu, zaczęto wprowadzać do lecznictwa substancje syntetyczne. Wciąż jednak punktem wyjścia dla prac chemików było przypadkowe spostrzeżenie. Przykładowo, pod koniec XIX wieku zaobserwowano, że kwas salicylowy, stosowany do leczenia duru brzusznego jako substancja przeciwbakteryjna, powoduje także obniżenie temperatury ciała. Modyfikacja struktury chemicznej kwasu salicylowego prowadziła do uzyskania dziesiątek nowych pochodnych, z których jedna – kwas acetylosalicylowy (aspiryna) – stał się częścią podstawowego zestawu leków przeciwgorączkowych.

Tradycyjne poszukiwanie leków opierało się więc na przypadkowych spostrzeżeniach empirycznych, po których następowało systematyczne tworzenie nowych związków

chemicznych, poddanych następnie testom biologicznym. Procesem tym nie kierowała zazwyczaj żadna wiedza teoretyczna, a jedynie wyczucie badacza. W chemii leków podejście typu „trial-and-error" wymaga nie tylko bardzo dużych nakładów czasu i wysiłków badaczy, ale przede wszystkim dużych nakładów finansowych (droga synteza i drogie badania aktywności), a jego wynik nie jest wcale pewny, gdyż większość stworzonych pochodnych jest zazwyczaj pozbawiona pożądanej aktywności biologicznej. Szacuje się, że przy tradycyjnym podejściu na jedną wprowadzoną do lecznictwa substancję trzeba przetestować od kilkuset do kilku tysięcy nieaktywnych pochodnych.

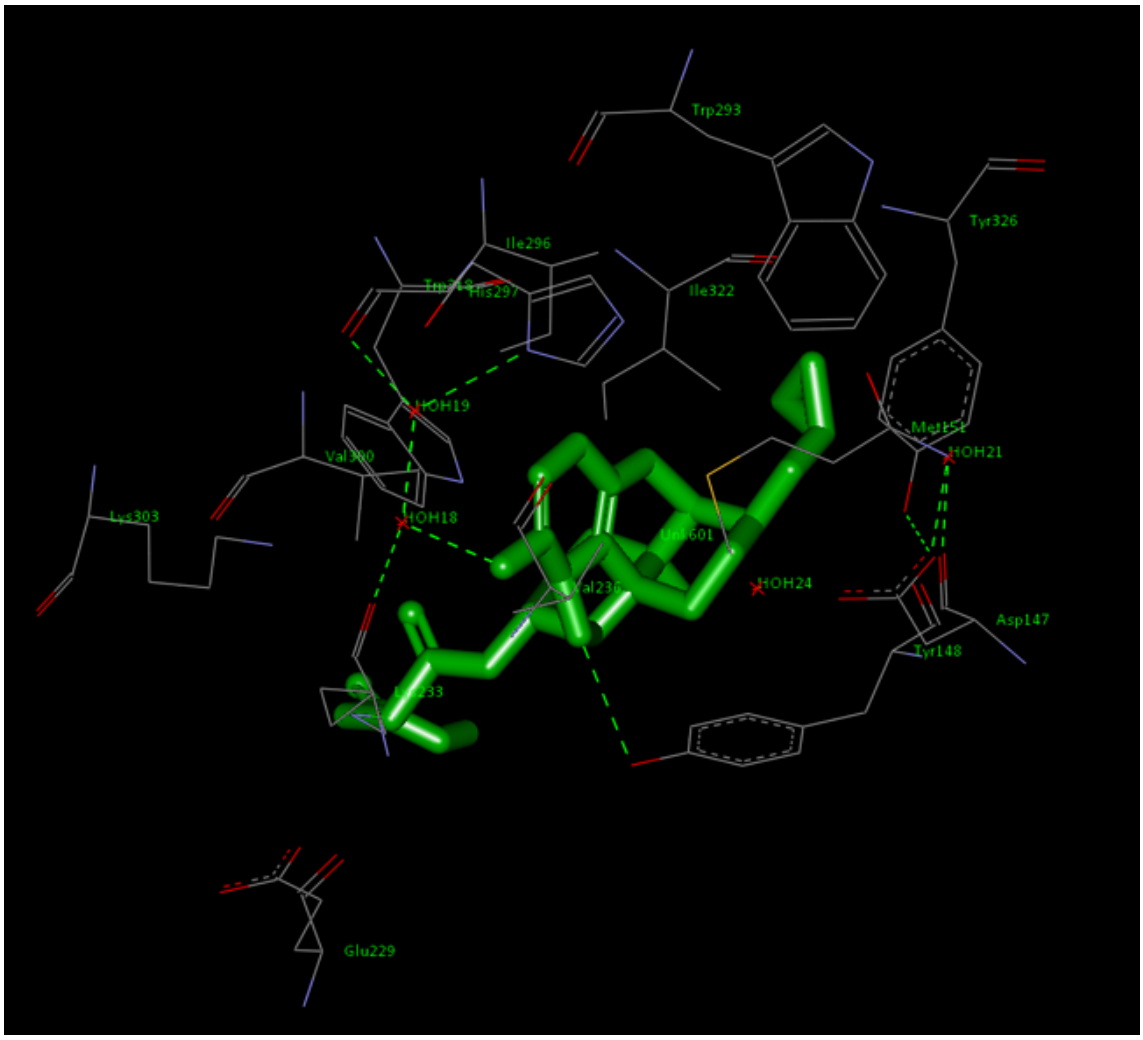
2.1.2. Sposób działania leków

Postęp nauk o życiu w XX wieku pozwolił zrozumieć podstawy molekularne wielu chorób, a także działania leków. Większość substancji aktywnych to małe cząsteczki chemiczne, które podane do organizmu, łączą się z odpowiednim punktem uchwytu (najczęściej białkiem: receptorem lub enzymem) i pośrednio lub bezpośrednio wywołują pożądaną zmianę w działaniu organizmu. Przykładowo, wspominana już aspiryna wiąże się z enzymami z rodziny cyklooksygenaz (COX) i blokuje ich funkcjonowanie, co powoduje spadek produkcji prostaglandyn, czyli cząsteczek uruchamiających w organizmie procesy zapalne [6]. Inhibicja (blokada) COX zmniejsza natężenie zapalenia, bólu, a także obniża temperaturę ciała.

Sam proces wiązania leku z białkiem polega na przestrzennym i elektrostatycznym dopasowaniu obu cząsteczek – mała molekuła leku znajduje wnękę w dużej cząsteczce białka, w którą wchodzi i utrzymuje się w niej dzięki różnym oddziaływaniom fizyko-chemicznym (balans między przyciąganiem i odpychaniem).

2.1.3. Dokowanie molekularne

Dzięki metodom bioinformatyki i chemii obliczeniowej możliwe jest modelowanie tego procesu. Jeśli cząsteczki potraktuje się jako obiekty fizyki klasycznej (mechanika molekularna) – kulki (atomy) o określonym ładunku połączone sprężynami (wiązaniami chemicznymi) – możliwe jest zgrubne przewidywanie ich oddziaływania. W komputerowo-wspomaganym projektowaniu leków (CADD) szczególnie przydatne jest dokowanie molekularne, które przyjmując jako dane wejściowe eksperymentalne struktury trójwymiarowe cząsteczek leku i białek, generuje w dość wiarygodny sposób strukturę



Rysunek 2.1. Wynik dokowania molekularnego – ułożenie cząsteczki leku w centrum aktywnym receptora[1]

kompleksu lek-cel molekularny (Rysunek 2.1). Uważna analiza (która jest ułatwiona dzięki grafice trójwymiarowej) pozwala na racjonalne zaproponowanie modyfikacji leku, tak aby uzyskać silniejsze wiązanie z docelowym białkiem. Dzięki temu zamiast syntezy i badania tysięcy pochodnych (jak w schemacie tradycyjnym), możliwe jest ograniczenie prac eksperymentalnych do racjonalnie wybranej podprzestrzeni chemicznej.

[1] Cząsteczka leku na zielono, grubymi liniami. Elementy cząsteczki białka cienkimi liniami. Przerywane zielone linie to oddziaływania pomiędzy lekiem a białkiem. Analiza sposobu związania leku z białkiem pozwala na racjonalne proponowanie nowych pochodnych.

2.1.4. Przesiew wirtualny

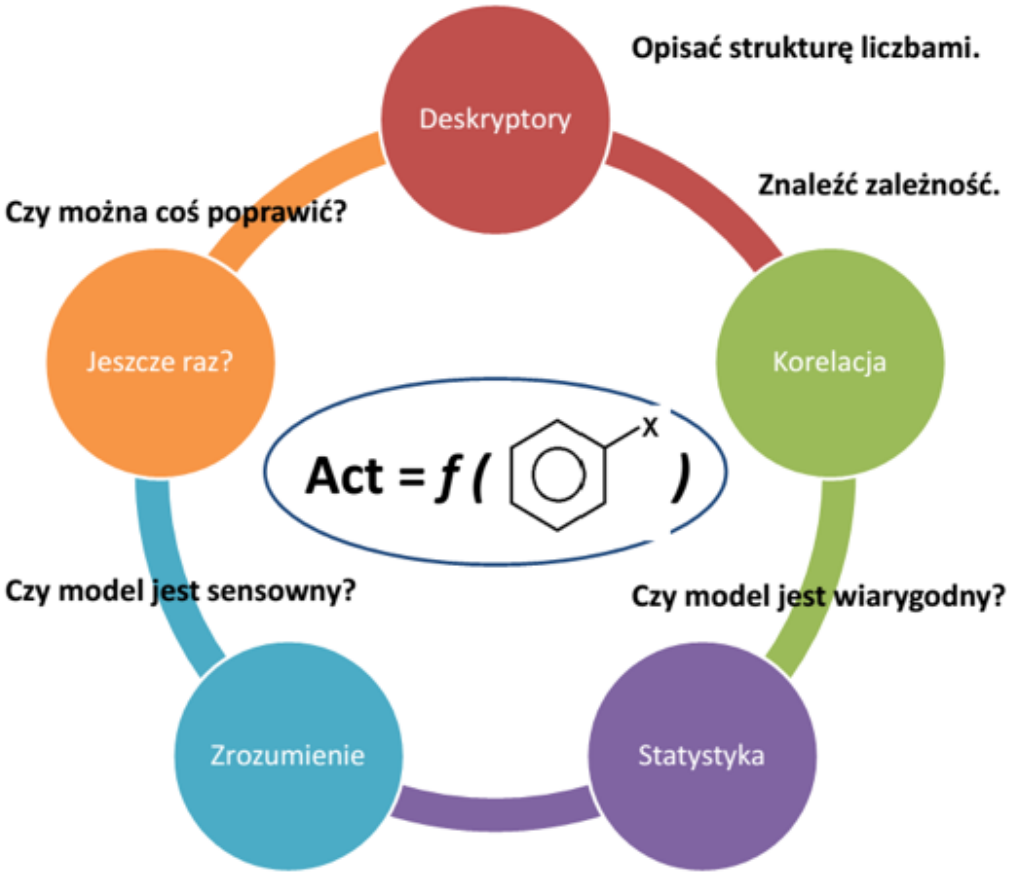
Szczególnym zastosowaniem dokowania molekularnego jest przesiew wirtualny (ang. virtual screening) [7]. Technika ta używana jest na początku prac nad nowym lekiem, zwłaszcza gdy nieznane są żadne substancje aktywne wobec danego celu molekularnego, bądź pożądane jest znalezienie substancji zupełnie niepodobnych do znanych leków. Przesiew wirtualny polega na komputerowym dokowaniu obszernej (o rzędzie wielkości nawet milionów) bazy potencjalnych substancji chemicznych do docelowego punktu uchwytu. Badanie eksperymentalne takiej liczby cząsteczek jest praktycznie niewykonalne, a z racji kosztów również niewskazane. Komputerowy skryning, pomimo dużej zgrubności przewidywań, pozwala wyselekcjonować kilkadziesiąt do kilkuset najbardziej obiecujących molekuł, przeznaczonych do dalszych badań, co znacznie obniża koszty finansowe całej kampanii badawczej.

2.1.5. Metody QSAR

Jeszcze inną metodologią CADD jest QSAR, czyli ilościowe zależności struktura-aktywność (ang. quantitative structure–activity relationships) [8]. W badaniach QSAR poszukuje się równań, które korelują aktywność biologiczną serii substancji z ich strukturą, co ideowo przedstawia wzór 2.1:

$$aktywnosc = f(struktury) \tag{2.1}$$

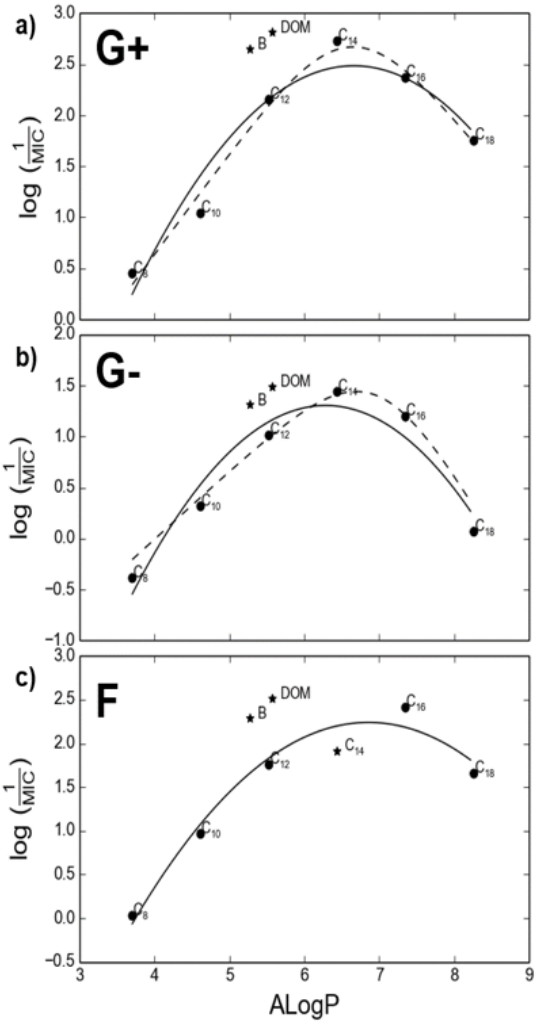
Struktura cząsteczek opisywana jest liczbowo przez deskryptory molekularne. Współczesna chemia zna tysiące różnych deskryptorów opartych na wielu podstawach teoretycznych. Niektóre wyprowadzane są ze wzorów cząsteczkowych (deskryptory 1D, np. liczba atomów tlenu w cząsteczce etc.), wzorów strukturalnych (deskryptory 2D, np. ilość wiązań pomiędzy pewnymi atomami tlenu w cząsteczce etc.), albo teoretycznych lub eksperymentalnych struktur trójwymiarowych (deskryptory 3D, np. objętość cząsteczki, powierzchnia etc.). Inna grupa deskryptorów wynika z traktowania molekuł jako grafów i związana jest z operacjami matematycznymi na odpowiednich macierzach opisujących grafy (deskryptory topologiczne) [9]. W badaniach QSAR dużą rolę odgrywają deskryptory opisujące właściwości całych cząsteczek (np. eksperymentalnie wyznaczana lipofilowość, czyli powinowactwo cząsteczki do fazy olejowej), albo poszczególnych atomów (np. ładunki atomowe z obliczeń kwantowo-chemicznych).



Rysunek 2.2. Schemat postępowania w modelowaniu QSAR

Deskryptory molekularne obliczone za pomocą odpowiedniego oprogramowania (Dragon [10], Accelrys Discovery Studio [11] etc.) lub uzyskane z badań eksperymentalnych tworzą pulę deskryptorów, która jest następnie wykorzystywana do trenowania modeli (tworzenia równań) za pomocą metod statystycznych (głównie regresja liniowa sprzężona z różnego rodzaju metodami wyboru zmiennych [12]). Otrzymane równania poddawane są ocenie statystycznej (współczynnik korelacji, odchylenie standardowe etc.) [13], a także – o ile to możliwe – interpretacji fizycznej. Pożądane są modele, które posiadają nie tylko doskonałe parametry statystyczne, ale również jasny sens fizyczny.

Równania QSAR mogą być dalej zastosowane do przewidywania aktywności cząsteczek jeszcze niezsyntezowanych i w ten sposób służyć do określenia, który obszar przestrzeni chemicznej wart jest czasochłonnych i drogich badań eksperymentalnych (Rysunek 2.2). Ostatecznym miernikiem jakości modeli QSAR jest właśnie ich moc predykcyjna, określana na zbiorze cząsteczek, które nie były użyte do trenowania modeli.



Rysunek 2.3. Wykresy korelujące aktywność migdalanów benzalkoniowych (1/MIC) z lipofilowością cząsteczek (w praktyce długością łańcucha węglowego) [2]

Jednym z podstawowych kierunków rozwoju metod QSAR opisanych w poprzednim rozdziale jest poszukiwanie nowych deskryptorów molekularnych. Jedną ze szczególnych klas takich deskryptorów są miary chiralności.

2.2. Chiralność

Chiralność jest kluczowym pojęciem współczesnej chemii leków. Według definicji Międzynarodowej Unii Chemii Czystej i Stosowanej (IUPAC), cząsteczka jest chiralna wtedy i tylko wtedy, gdy nie da się nałożyć na nią jej lustrzanego odbicia [14]. Operacyjnie można też powiedzieć, że cząsteczka jest chiralna, jeśli w jej budowie występują elementy

[2] Uzyskane modele QSAR są jasną wskazówką dla projektowania nowych związków. Parabola wskazuje na istnienie pewnego optymalnego dla aktywności zakresu długości łańcucha węglowego.

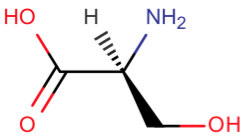
chiralności: centrum chiralne (najczęściej asymetryczny atom węgla), oś chiralności albo płaszczyzna chiralności (Rysunek 2.4). Konsekwencją chiralności, jest występowanie w przyrodzie enancjomerów, czyli par cząsteczek (wzajemnych lustrzanych odbić) o takiej samej liczbie takich samych atomów, połączonych ze sobą tymi samymi wiązaniami, ale inaczej ułożonych przestrzennie. Z punktu widzenia właściwości chemicznych i fizycznych enancjomery są praktycznie identyczne i nierozróżnialne – mają takie same masy cząsteczkowe, temperatury topnienia, dają identyczne widma cząsteczkowe i atomowe etc. Jedynym wyjątkiem jest kierunek skręcania światła spolaryzowanego, dlatego w parze enancjomerów wyróżnia się molekuły lewo- i prawoskrętne, oznaczane +/-, R/S bądź D/L. Enancjomery mogą też różnić się właściwościami biologicznymi [15]. Przykładowo, (+)-asparagina wywołuje odczucie smaku słodkiego, podczas gdy (-)-asparagina pozbawiona jest smaku; (-)-morfina jest bardzo silnym lekiem przeciwbólowym, a jej (+)-enancjomer nie powstrzymuje bólu; (R)-talidomid ma działanie uspokajające, (S)-talidomid jest bardzo silną trucizną, powodującą uszkodzenia komórek [16][3]. Jako że bardzo duża część współcześnie stosowanych leków jest chiralna, konieczne są skuteczne metody otrzymywania, oczyszczania i analityki pojedynczych enancjomerów.

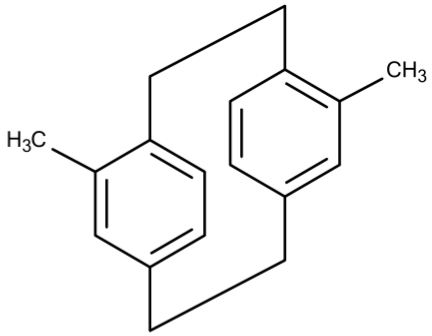
2.3. Miary chiralności

Oprócz tego, zwrócono uwagę na jeszcze inny aspekt powszechności tej właściwości wśród molekuł o znaczeniu leczniczym. Chiralność jest cechą trójwymiarowego kształtu cząsteczki. A jeżeli, jak wspomniano wcześniej, oddziaływanie leku z celem molekularnym związane jest między innymi z przestrzennym dopasowaniem cząsteczki leku i celu molekularnego, można wykorzystać chiralność do opisu struktury cząsteczki. Służą do tego miary chiralności, czyli deskryptory mówiące, jak bardzo chiralna jest cząsteczka, albo inaczej (zgodnie z definicją IUPAC[14]): jak bardzo nienakładalna na swoje lustrzane odbicie.

Wśród kilku podejść do tego zagadnienia, dalej omówiono krótko miary chiralności typu Sinister-Rectus (^{SR}CM), które są przedmiotem niniejszej pracy [17, 18]. W procedurze obliczania ^{SR}CM, generowane jest lustrzane odbicie analizowanej cząsteczki, które

[3]Przypadek talidomidu jest prawdopodobnie najsłynniejszą i najtragiczniejszą w historii chemii leków ilustracją znaczenia chiralności [16]. Substancja ta w latach 50 i 60 XX wieku stosowana była w Europie jako łagodny i bezpieczny lek przeciwko nudnościom odczuwanym przez kobiety ciężarne. Podawany był w postaci racematu – czyli jako mieszanina obu enancjomerów. Wiedza o szkodliwym działaniu (S)-talidomidu nie była wtedy znana. Cząsteczka ta spowodowała poważne uszkodzenia u dzieci, w tym brak kończyn i nienaturalne proporcje ciała. Odnotowano ok. 15 000 przypadków ciężkich wad rozwojowych wywołanych przez talidomid. Obecnie lek ten jest stosowany jako trucizna komórkowa w leczeniu niektórych nowotworów.



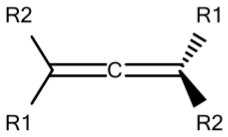


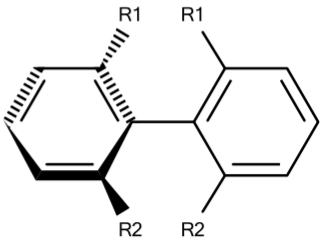
asymetryczny węgiel

seryna

płaszczyzna chiralności

[2.2]-paracyklofan





oś chiralności

podstawione alleny **orto-podstawione bifenyle**

Rysunek 2.4. Przykłady cząsteczek chiralnych z różnymi elementami chiralności[4]

następnie jest nakładane w najlepszy możliwy sposób na oryginał. Miarą nienałożenia jest znormalizowana suma kartezjańskich odległości pomiędzy atomami w cząsteczce i jej lustrzanym odbiciu, ważonych zależnie od zastosowanej przestrzeni właściwości:

$$^{SR}CM(A) = \frac{1}{a} min(\sum_{i=1}^{n} w_i d_i) \tag{2.2}$$

gdzie a to współczynnik normalizujący (masa cząsteczkowa), d_i – odległość, w_i – waga (najczęściej właściwość przypisana danemu atomowi, np. masa atomowa, ładunek cząsteczkowy etc.), n – liczba atomów w cząsteczce.

Poważnym problemem jest znalezienie optymalnego nałożenia, dla którego ^{SR}CM będzie najmniejsze. Lustrzane odbicie cząsteczki achiralnej zostanie po prostu idealnie dopasowane do oryginalnej molekuły (wtedy $^{SR}CM = 0$), natomiast w przypadku

[4]Pogrubione linie oznaczają wiązania skierowane ponad płaszczyznę, linie przerywane oznaczają wiązania skierowane pod płaszczyznę.

cząsteczek achiralnych możliwych jest wiele niedoskonałych nałożeń, z których konieczne jest znalezienie takiego, gdzie ^{SR}CM będzie najmniejsze. Ponieważ molekuły o znaczeniu biologicznym składają się z kilkudziesięciu do kilkuset atomów, przestrzeń rozwiązań tego problemu jest bardzo duża i dlatego istotne jest tutaj opracowanie oraz implementacja wydajnych algorytmów poszukujących optymalnego nałożenia (najmniejszego nienałożenia).

Miary typu ^{SR}CM zastosowane zostały już w badaniach chiralnych heterofullerenów [17], modelowaniu aktywności steroidów działających na globulinę wiążąca hormony płciowe [18] oraz w analizie widm oscylacyjnego dichroizmu kołowego[19]. Inne miary chiralności wykorzystywano także do modelowania aktywności związków przeciwbólowych, inhibitorów acetylocholinoesterazy, zachowania aminokwasów w chromatografii płytkowej, a także do wyjaśniania działania katalizatorów. Pokazuje to potencjał oraz konieczność opracowywania efektywnego oprogramowania do obliczania tych deskryptorów. Dzięki temu możliwe będzie rozszerzenie stosowania miar chiralności w bioinformatyce (ściślej w komputerowo-wspomaganym projektowaniu leków), a także szeroko pojętej chemii obliczeniowej.

3. Metody obliczeniowe wykorzystywane w tworzonej aplikacji

3.1. Algorytmy genetyczne

Wiele z problemów komputerowo-wspomaganego projektowania leków, albo szerzej chemii obliczeniowej, sprowadza się do zadania optymalizacyjnego. W niniejszej pracy również pojawia się problem optymalizacyjny: odnalezienie najlepszego nałożenia cząsteczki i lustrzanego odbicia, który autor postanowił rozwiązać za pomocą algorytmu genetycznego.

Algorytmy genetyczne (AG) to metody znajdowania rozwiązań należące do klasy przeszukiwania heurystycznego, czyli takiego które nie gwarantuje znalezienia optymalnego rozwiązania. Znane już od wczesnych lat siedemdziesiątych zeszłego stulecia AG zdążyły zdobyć uznanie jako sprawdzona metoda poszukiwania rozwiązań dla problemów nieliniowych, które są słabo opisane.

AG wzorowane na naturalnych mechanizmach ewolucyjnych zachodzących w przyrodzie. W uproszczeniu algorytm genetyczny symuluje życie osobników w losowej populacji, w której zachodzą zjawiska selekcji, mutacji oraz krzyżowania[20].

Typowy przebieg algorytmu genetycznego przedstawia Rysunek 3.1. Na początku generowana jest losowa populacja osobników, gdzie każdy osobnik posiada zestaw cech tj. zestaw genów zwany genotypem. Najczęściej geny kodowane są binarnie albo w postaci liczb rzeczywistych. Dla każdego osobnika, na podstawie jego genotypu, liczona jest funkcja przystosowania, zwana także funkcją celu, której wartość chcemy zmaksymalizować (lub zminimalizować) dzięki działaniu algorytmu genetycznego. W kolejnym kroku sprawdzamy wcześniej założone warunki końca – może to być m. in. brak poprawy rozwiązań od kilku iteracji, maksymalna liczba iteracji, pewna progowa wartość funkcji celu - kryteria końca należy dobrać do konkretnego zastosowania algorytmu genetycznego. Jeśli nie są one spełnione, przeprowadzany jest proces selekcji osobników nadających się do bycia rodzicami dla nowego pokolenia.

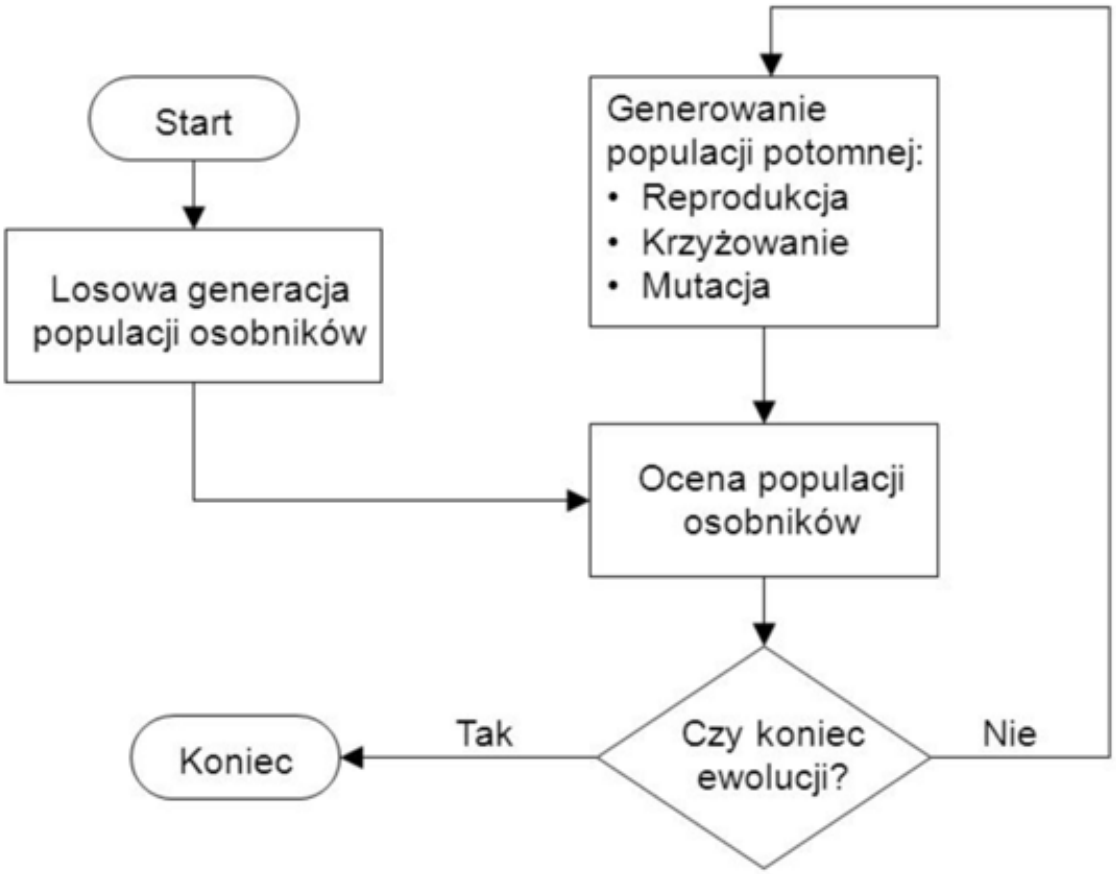
Podstawowymi metodami selekcji są: metoda ruletkowa, metoda rankingowa oraz metoda turniejowa. Poza nimi występuje też wiele innych metod, często będących odmianami

wymienionych trzech podstawowych[21]. Gdy rodzice zostaną wybrani, zwyczajowo następują 2 operacje genetyczne, krzyżowanie oraz mutacja. W zależności od wybranego sposobu krzyżowania oraz zadanego prawdopodobieństwa, część genów od jednego i część od drugiego (istnieją algorytmu genetyczne, w których do reprodukcji używa się więcej niż dwóch rodziców) rodzica trafia do potomka. W trakcie mutacji, pod pewnym prawdopodobieństwem (zazwyczaj bardzo niskim) następuje zmiana wartości wybranego genu na wartość losową co pozwala algorytmowi „skakać" po całej dostępnej przestrzeni rozwiązań. W tym momencie następuje powrót algorytmu do etapu oceny, tym razem nowej populacji oraz powtórzenia wszystkich pozostałych kroków w przypadku niespełnienia kryteriów końca.

Niestety, elastyczność, która z jednej strony jest dużą zaletą algorytmów genetycznych, jest też ich znaczącą wadą. Nie ma jedynego prawidłowego modelu pasującego do wszystkich klas problemów, więc przed zastosowaniem AG w każdym konkretnym przypadku trzeba przeprowadzić wstępną analizę oraz podjąć decyzje odnośnie poniższych punktów:

- jakie cechy danego problemu przedstawić w postaci genów,
- jakie kodowanie wybrać dla genów (binarne lub w postaci liczb rzeczywistych),
- jaką przyjąć funkcję przystosowania,
- dobrać operatory genetyczne (selekcja, krzyżowanie, mutacja, inwersja, permutacja),
- dobrać najlepszą metodę selekcji z odpowiednią presją selekcji,
- wybrać sposób krzyżowania,
- dobrać wartości prawdopodobieństw operatorów genetycznych,
- ustalić warunki przerwania algorytmu.

Poza trudnością doboru wszystkich wymienionych wyżej parametrów algorytmy genetyczne mają jeszcze jedną dosyć istotną wadę - bardzo dużą złożoność obliczeniową (w teorii algorytmy genetyczne pracują przy użyciu nieskończonych zasobów, co oczywiście w świecie rzeczywistym jest niemożliwe). Jednym ze sposobów optymalizacji wydajności jest tu zastosowanie równoległych wersji algorytmów genetycznych.



Rysunek 3.1. Przebieg algorytmu genetycznego

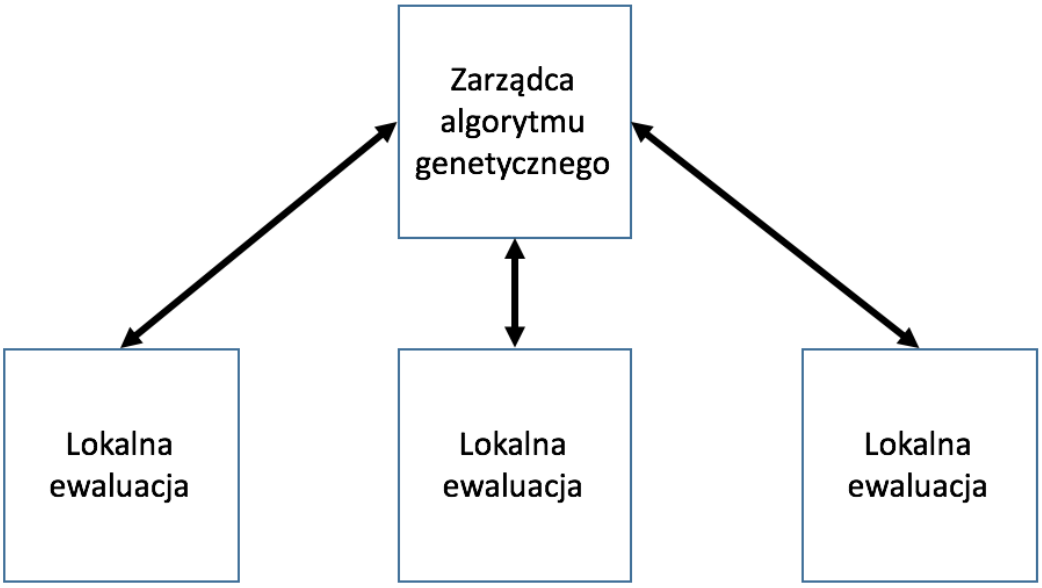
3.2. Równoległe algorytmy genetyczne

Wstępna analiza algorytmów genetycznych z punktu widzenia programowania równoległego pozwala wyodrębnić w prosty sposób trzy poziomy niezależności danych:

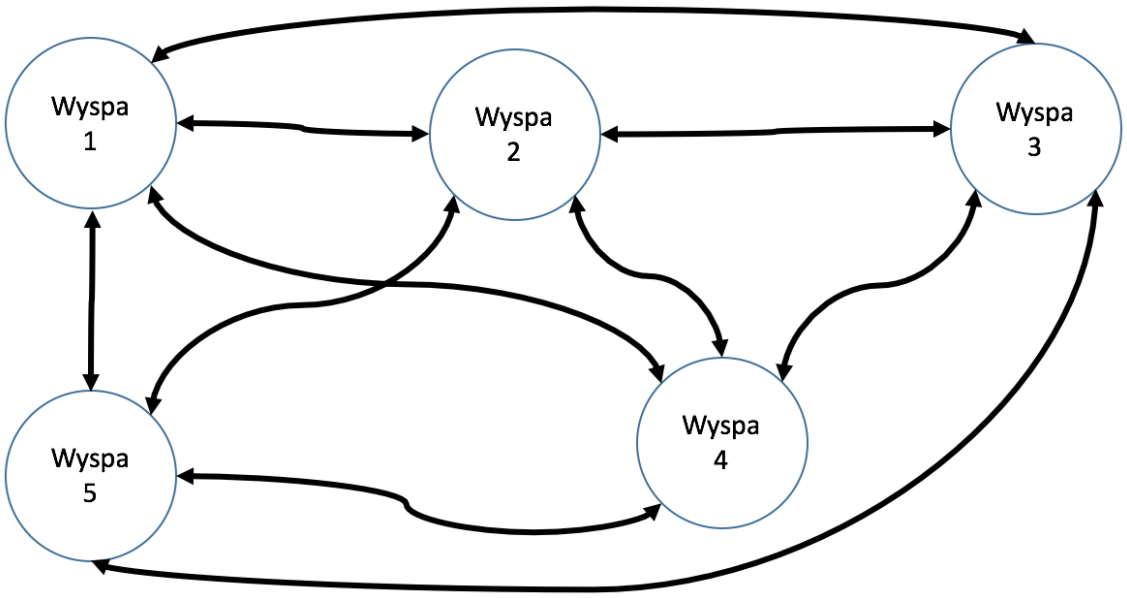
- wartości poszczególnych genów w genotypie zazwyczaj nie zależą od siebie,
- każdy osobnik w populacji jest zupełnie niezależny od innych osobników (do momentu fazy krzyżowania i mutacji),
- z punktu drugiego wynika też, że populację można podzielić na mniejsze populacje, które mogą żyć zupełnie niezależnie.

Dzięki wyodrębnieniu tych trzech możliwych miejsc problem zrównoleglenia algorytmu wydaje się względnie prosty. Wyróżnia się 3 główne klasy równoległych algorytmów genetycznych[22, 23]

a) Master Slave Model,



Rysunek 3.2. Master Slave Model



Rysunek 3.3. Model wyspowy (Coarse Grained)

b) Coarse Grained,

c) Fine Grained.

W modelu Master Slave Model (MSM) istnieje element centralny, który dzieli pracę pomiędzy różne jednostki obliczeniowe (węzły), np. oddzielne węzły liczą funkcję celu dla oddzielnych części populacji, co przedstawia Rysunek 3.2. W wersji synchronicznej węzeł nadrzędny czeka na wyniki obliczeń węzłów podrzędnych za każdym razem, w wersji asynchronicznej, w miarę możliwości kontynuuje obliczenia bez czekania na wyniki.

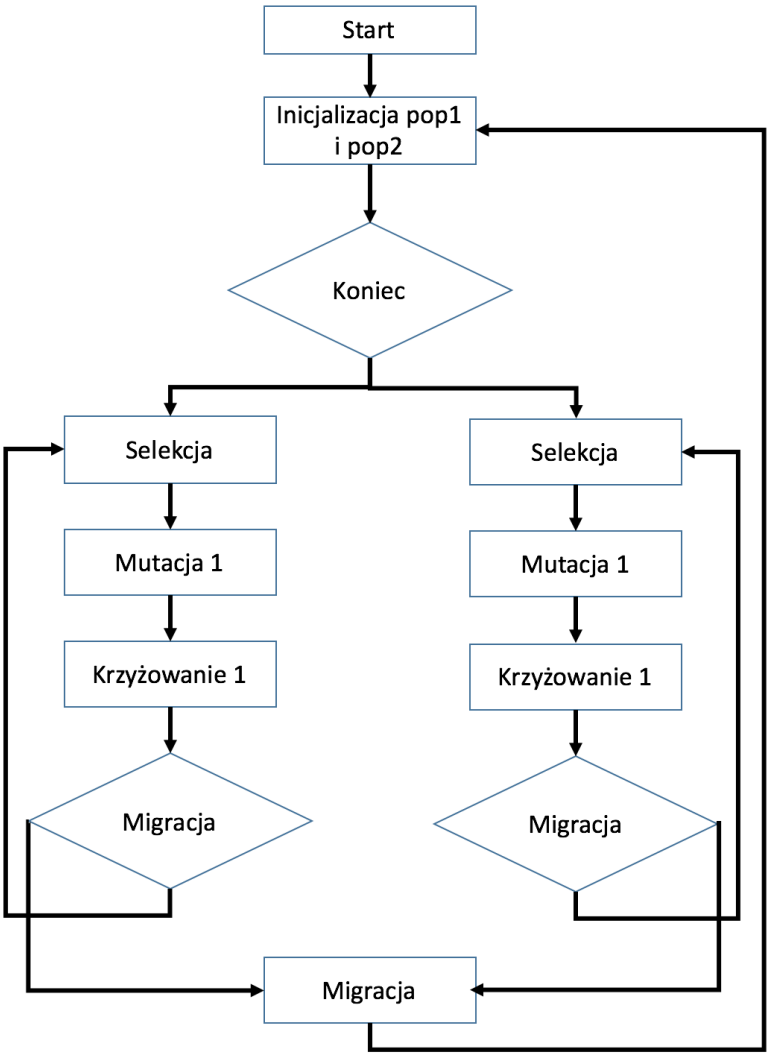
Model Coarse Grained (CG, zwany też wyspowym) przedstawiony na Rysunek 3.3 dzieli całą populację na kilka mniejszych, tzw. „wysp". Każda z nich zachowuje się samodzielnie, czyli wszystkie kroki algorytmu wykonywane są niezależnie dla każdej wyspy. W pewnych zdefiniowanych momentach następuje wymiana części osobników pomiędzy populacjami. Podejście to ma problem ze zbyt wczesną dominacją lokalnie silnych osobników, czemu zapobiegać można odpowiednim dobraniem częstotliwości migracji oraz rozmiaru migracji. Warto wspomnieć też o odmianie „Dual Species" zaprezentowanej na Rysunek 3.4. Tak samo jak w Coarse Grained populacja dzielona jest na grupy, jednak każda grupa ma inne parametry algorytmu genetycznego, np. w grupie pierwszej częściej krzyżują się osobniki, które są podobne do siebie, w drugiej grupie z większym prawdopodobieństwem krzyżowanie zachodzi w przypadku osobników mocno różniących się od siebie. Identycznie jak w modelu CG, co kilka pokoleń następuje migracja. Różnicowanie parametrów dla poszczególnych grup ma także chronić algorytm przed przedwczesnym zejściem do lokalnego minimum.

W przypadku modelu Fine Grained jest jedna duża populacja, a zrównoleglenie następuje na poziomie poszczególnych osobników - nie ma globalnej selekcji, wszystkie operatory genetyczne działają lokalnie, tj. osobniki współpracują z najbliższymi sąsiadami, np. tak jak na Rysunek 3.5. Wadą tego podejścia jest zauważalny spadek wydajności w przypadku wzrostu wielkości populacji.

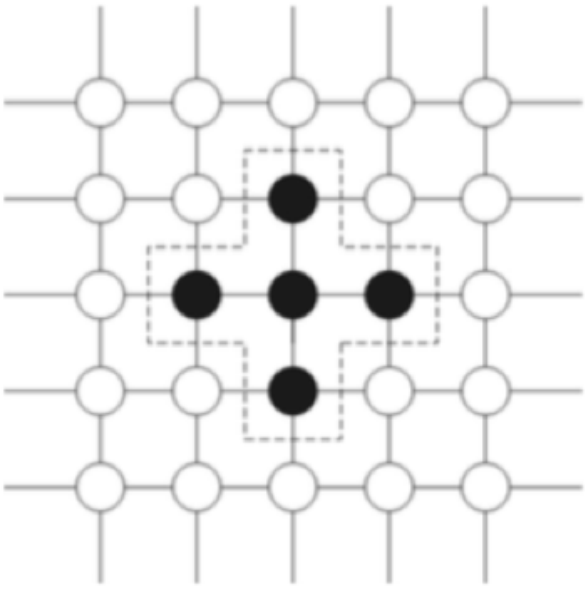
3.3. GPGPU

GPGPU (General-Purpose Computing on Graphics Processing Units) jest to technika programowania aplikacji polegająca na wykorzystaniu procesorów kart graficznych (GPU) do obliczeń związanych nie tylko z grafiką komputerową, w stylu przypominającym programowanie tradycyjnych procesorów CPU (ang. Central Processing Unit).

Geneza GPGPU wiąże się z rozwojem hardware'u do wyświetlania grafiki trójwymiarowej na potrzeby gier komputerowych. Karty graficzne zostały wyspecjalizowane do równoległego przetwarzania bardzo dużej liczby danych zmiennoprzecinkowych (zwykle scena w grze komputerowej składa się z dziesiątek albo setek tysięcy wierzchołków których pozycje opisane są liczbami zmiennoprzecinkowymi). W końcu dostrzeżono możliwość wykorzystania tego faktu do zastosowań innych niż gry komputerowe. Za początek GPGPU przyjmuje się pojawienie się możliwości programowania potoku pixel i vertex shader (czyli



Rysunek 3.4. Schemat modelu Dual Species



Rysunek 3.5. Model Fine Grained

krótkich programów opisujących właściwości wyświetlanych pikseli oraz wierzchołków) co miało miejsce około roku 2001. Językiem wykorzystywanym do oprogramowania potoku wierzchołków i pikseli była odmiana assemblera. Następnie w 2002 roku pojawił się język Cg wzorowany na C. Jednak prawdziwym przełomem była tu unifikacja jednostek odpowiedzialnych za pixel/vertex shading i stworzenie przez firmę NVIDIA w 2006 roku środowiska CUDA.

Pojawienie się GPGPU spowodowało niespotykany dotąd spadek cen mocy obliczeniowej w przeliczeniu na jeden GigaFlops (miliard operacji zmiennoprzecinkowych na sekundę). Niecałe 10 lat temu najtańszy 1 GFlop kosztował ok. 48 dolarów zaś w roku 2015 jego cena wynosiła już tylko 0.08 dolara, co oznacza 600-krotny spadek cen w czasie krótszym niż dekada[24]. Niedawno zaprezentowana najnowsza seria produktów firmy NVIDIA oznaczona numerem 10xx w topowym modelu GTX 1080 osiąga teoretyczną wydajność 9 TFlops (9 bilionów obliczeń zmiennoprzecinkowych na sekundę). Dla porównania, najwydajniejszy procesor firmy Intel osiąga 113 GFlops.

Duża wydajność obliczeniowa rozwiazań GPGPU powoduje duże zainteresowanie nimi w zastosowaniach fizyki i chemii obliczeniowej.

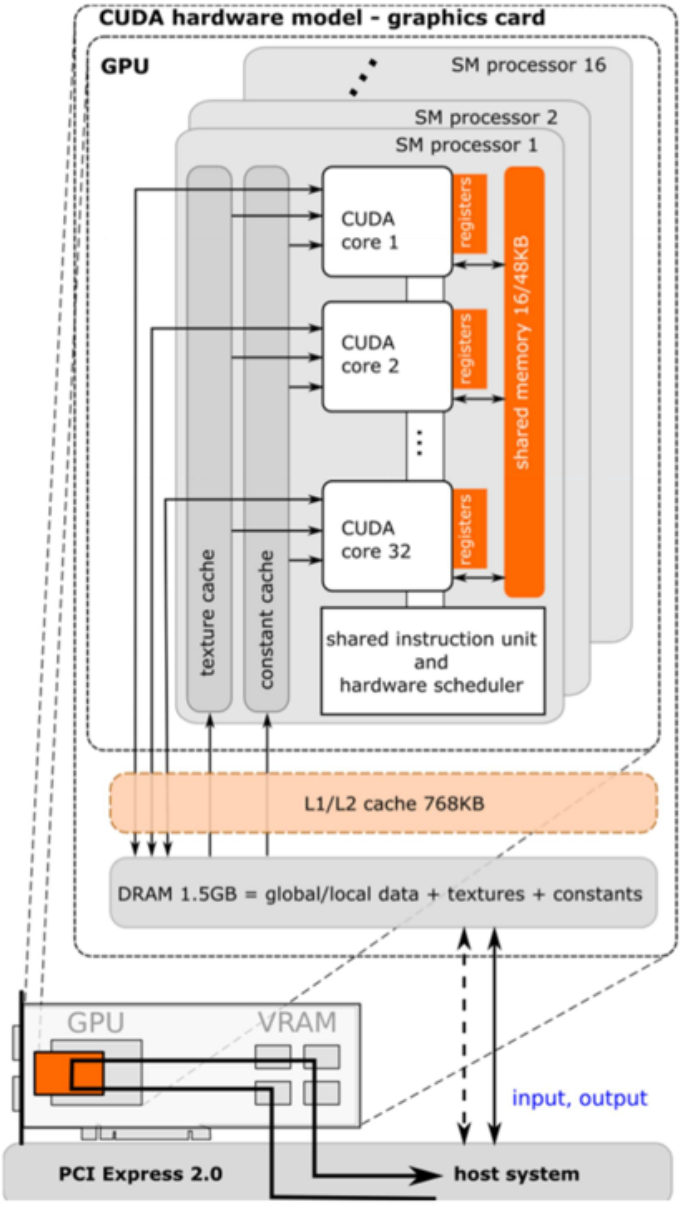
3.4. Technologia CUDA

Pojawienie się architektury CUDA (Compute Unified Device Architecture)[1] zdecydowanie przyspieszyło rozwój GPGPU. Dzięki tej platformie sprzętowo-programowej możliwe stało się wykonywanie kodu napisanego w języku wysokiego poziomu bezpośrednio na karcie graficznej.

CUDA składa się z następujących elementów:

- host (gospodarz)
  - biblioteki ukrywające wykorzystanie CUDA jak np. Thrust, PhysX,
  - kod napisany w języku wysokiego poziomu takim jak C, C++ (od wersji 7 pełne wsparcie C++11, szablony, funkcje lambda itp.),
  - kompilator nvcc, który kod wykonywany na CPU kompiluje za pomocą kompilatora dostępnego w systemie (MSVC, GCC, CLANG).

[1] Obecnie, firma NVIDIA zaprzestała korzysta z pełnej nazwy, w oficjalnej dokumentacji wstępuje tylko CUDA.



Rysunek 3.6. Budowa procesora karty graficznej firmy NVIDIA

- device (urządzenie)
  - urządzenie firmy NVIDIA obsługujące technologię CUDA, z „compute capability”[2] wyższym lub równym temu, które zostało wykorzystane do zaprogramowania aplikacji,
  - urządzeń tych może być więcej niż jedno.

Wizualizację budowy GPU przedstawia Rysunek 3.6. Procesor GPU podzielony jest na przeciętnie do kilkunastu Streaming Multiprocessor (SM), z których każdy zawiera wiele

[2]Compute capability – określa możliwości sprzętowe do zastosowań GPGPU danej karty

rdzeni CUDA. Rdzenie wewnątrz jednego SM posiadają bardzo szybką współdzieloną pamięć, dzięki której możliwa jest łatwa i szybka wymiana danych pomiędzy poszczególnymi rdzeniami wewnątrz SM a także synchronizacja. Dodatkowo, na karcie graficznej znajduje się także pamięć główna wykorzystywana do komunikacji między kartą graficzną a komputerem, a także do wymiany danych między SM. Przy czym, pamięć główna jest stosunkowo wolna. Cały GPU zbudowany jest w oparciu o model SIMT (ang. Single Instruction, Multiple Threads) czyli pojedyncza instrukcja wykonywana jest przez wiele wątków [25].

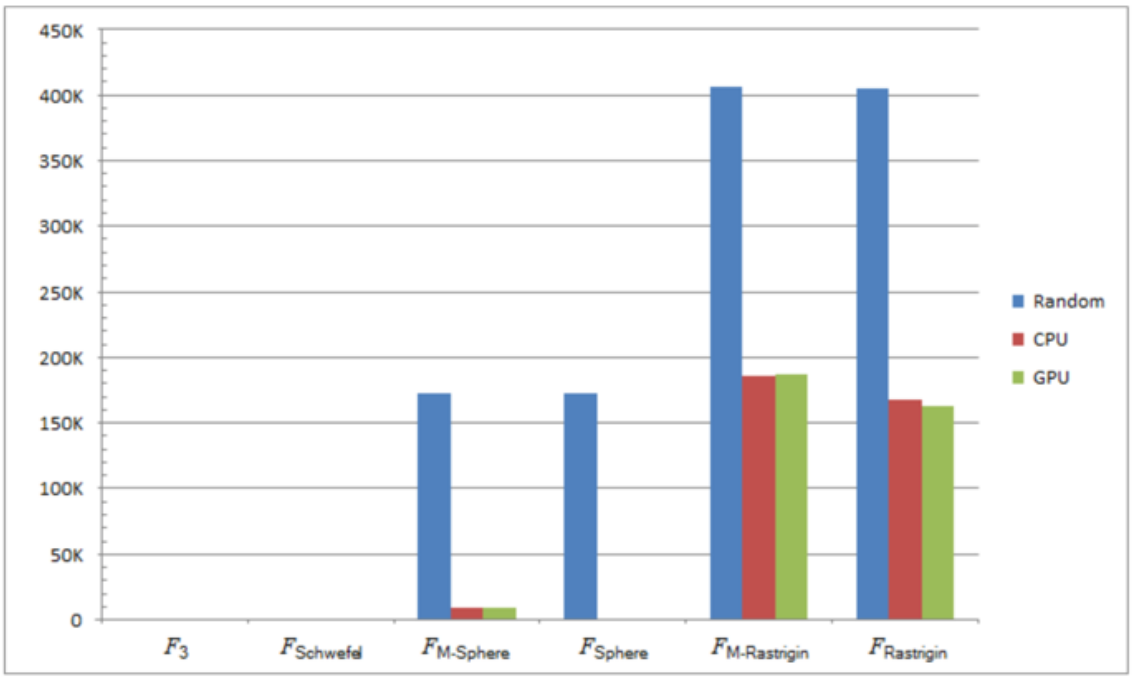
GPU ma w porównaniu do CPU wiele zalet:

- teoretyczna moc obliczeniowa pojedynczego układu jest bardzo duża, dla przykładu GEFORCE GTX 1080 ma 9TFlops,
- cena jednego układu jest stosunkowo niska, podawany wyżej GTX 1080 kosztuje ok. 600 dolarów,
- stosunek zużycia energii do mocy i ceny takiego układu wypada zdecydowanie lepiej niż dla CPU,
- z racji modelu SIMT koszt utworzenia i obsługi pojedynczego wątku jest bardzo mały,
- pamięć RAM na karcie graficznej jest dużo szybsza niż ta stosowana dla CPU.

Z drugiej strony, inne aspekty porównania wypadają korzystniej dla CPU:

- rozgałęzianie kodu w model SIMT jest problematyczne, dlatego też należy zmodyfikować styl programowania tak aby ich unikać,
- obliczenia podwójnej precyzji (typ *double*) są w teorii dwa razy wolniejsze niż pojedynczej precyzji(typ *float*), w praktyce różnice są jeszcze większe,
- wydajność pojedynczego wątku jest bardzo mała,
- aby osiągnąć największy wzrost wydajności, należy dostarczyć karcie graficznej możliwie dużo zadań tak, aby nie było momentów, kiedy poszczególne rdzenie nie pracują.

Warto też pamiętać, że wąskim gardłem w aplikacjach realizowanych z użyciem GPU jest szyna łącząca kartę graficzną i CPU – PCI-Express. Niska przepływność tego połączenia może w drastyczny sposób ograniczyć efektywność GPGPU. W skrajnym przypadku dochodzi nawet do spowolnienia względem programów działających wyłącznie na CPU. Stąd też należy unikać transferu danych do i z karty graficznej.



Rysunek 3.7. Porównanie czasu obliczeń CPU/GPU[3]

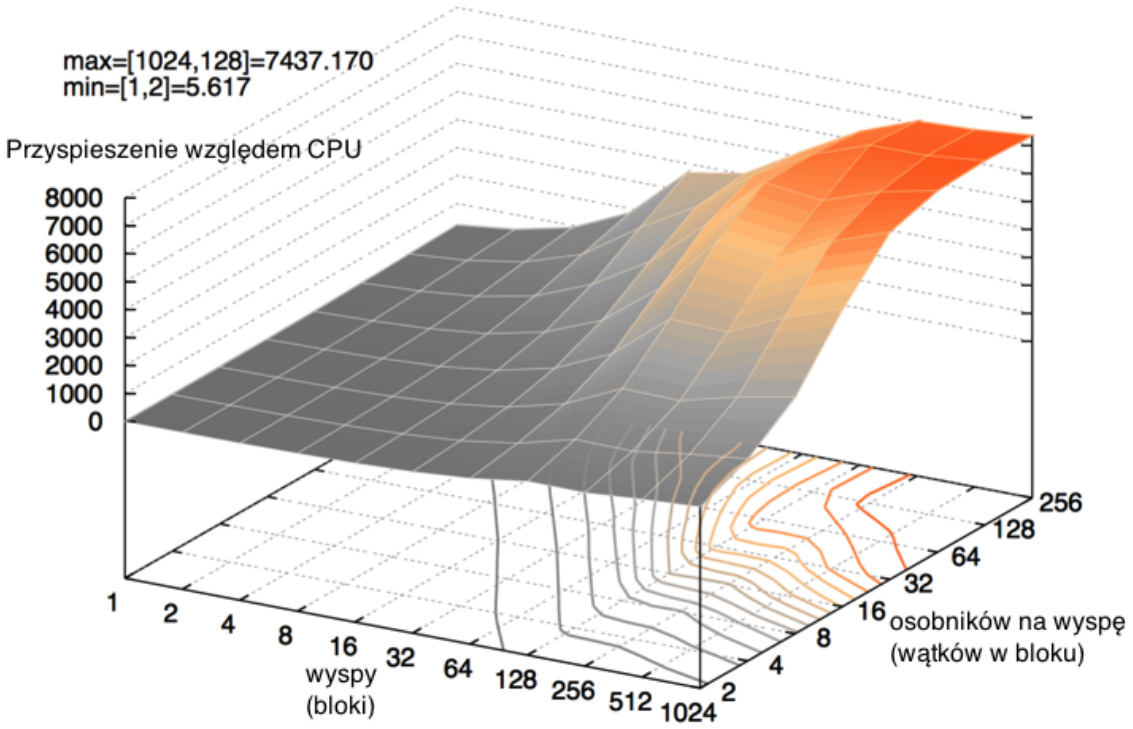
3.4.1. Równoległe algorytmy genetyczne na CUDA

W literaturze znanych jest kilka przykładów algorytmów genetycznych na GPU.

Najprostszym podejściem do wykorzystania GPU w algorytmie genetycznym wydaje się być liczenie wartości funkcji celu dla każdego osobnika z populacji w oddzielnym wątku na GPU, która zazwyczaj w algorytmie genetycznym jest najtrudniejsza obliczeniowo[26]. Przykładowe porównanie wykonania zadań benchmarkowych przedstawiono na Rysunek 3.7. W tym przypadku mamy do czynienia ze statystycznie nieistotnymi różnicami wydajności między CPU i GPU. Brak wzrostu efektywności związany jest z wymienionymi już w poprzedniej sekcji problemami. Po pierwsze, aplikacja czeka bezczynnie na obliczenie funkcji celu. Analogiczna sytuacja ma miejsce kiedy funkcja celu zostanie policzona, a GPU czeka bezproduktywnie, aż CPU zleci nowe zadanie. Kolejnym problemem w tym podejściu jest wspominane wąskie gardło w postaci szyny PCI-Express. A jako że funkcja przystosowania liczona jest w każdej iteracji algorytmu i iteracji wykonywa się bardzo dużo, sumaryczne spowolnienie związane z transferem danych zwielokratnia się.

Innym podejściem [28], które w założeniu próbuje lepiej wykorzystać równoległą architekturę GPU jest model wyspowy (albo inaczej model Coarse Grained) zaprezentowany

[3]Porównanie przy użyciu funkcji benchmarkowych stosowanych do testowania algorytmów genetycznych, opis wykorzystanych funkcji w [27].



Rysunek 3.8. Przyrost wydajności GPU względem CPU w modelu wyspowym dla karty GeForce GTX 285[4]

na Rysunku 3.3. Całościowa populacja wszystkich osobników podzielona jest na kilka mniejszych „sub-populacji". Co kilka generacji następuje migracja pewnej grupy osobników pomiędzy „wyspami" dzięki czemu materiał genetyczny może być skutecznie wymieniany dla populacji jako całości. Częstotliwość i rozmiar wymiany są kluczowymi parametrami, które należy dobrać tak aby zapewnić odpowiednią wydajność (zbyt częste migracje spowodują utratę „zrównoleglenia" algorytmu) oraz jakość(zbyt małe i rzadkie migracje spowodują szybkie zbieganie do lokalnych maksimów) rozwiązania.

Bardzo duża zaletą takiego modelu jest jego łatwość w implementacji na architekturę CUDA. Odpowiednio dobierając rozmiar wysp można umieścić wszystkie osobniki danej populacji w pamięci współdzielonej przez jedno SM, tak aby pojedynczym osobnikiem zajmował się oddzielny wątek z danego SM. Do migracji pomiędzy populacjami wykorzystana jest główna pamięć karty graficznej. Dzięki takiemu podejściu kosztowna wydajnościowo synchronizacja pomiędzy SM odbywa się tylko w przypadku migracji pomiędzy populacjami (synchronizacje wątków w obrębie jednego SM są mniej kosztowne), nie licząc tych momentów cały algorytm odbywa się asynchronicznie na karcie graficznej, bez potrzeby transferu danych do CPU w trakcie działania.

[4]W oparciu o [26].

Porównując wydajność takiej implementacji z najprostszą, jednowątkową implementacją algorytmu genetycznego na CPU można uzyskać wzrost wydajności nawet do ośmiu tysięcy razy, Rysunek 3.8. W tym miejscu należy jednak zwrócić uwagę, iż wszystkie obecnie sprzedawane CPU występują jako procesory co najmniej dwu rdzeniowe, dlatego też porównanie wydajności powinno zostać wykonane dla odpowiednio zoptymalizowanej implementacji na CPU. Według autorów pracy [29] implementacja wyżej przedstawionego podejścia z wykorzystaniem tylu wątków, ile rdzeni ma procesor, oraz optymalizacja dostępu do cache L2 procesora pozwala zmniejszyć przewagę wydajności GPU z kilkuset do kilku razy. Dodatkowym parametrem, który może mieć decydujące znaczenie dla pewnych klas problemów, jest jakość znalezionego rozwiązania - zwykle implementacje na CPU pozwalają znaleźć osobniki o wyższej wartości funkcji przystosowania niż implementacje na GPU.

Innym możliwym sposobem zwiększenia poziomu zrównoleglenia jest wykorzystanie oddzielnego SM dla każdego osobnika[29]. Wtedy każdy gen genotypu obsługiwany jest przez oddzielny wątek. Niestety, takie podejście jest skuteczne tylko w przypadku posiadania długich genotypów, w innym przypadku zbyt dużo zasobów GPU jest bezproduktywne. Podejście to pozwoliło na uzyskanie przyspieszenia rzędu 18 razy, niestety, autorzy nie podają porównania z podejściem wyspowym.

4. Implementacja aplikacji

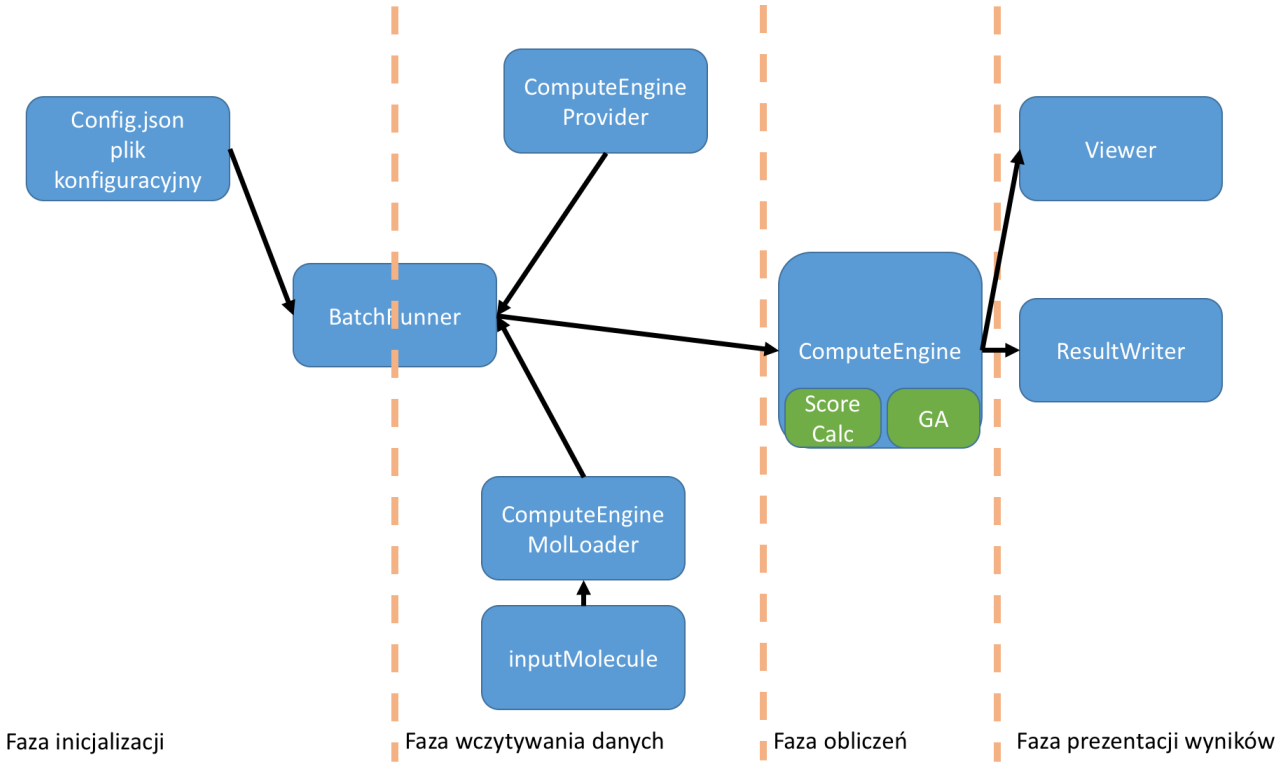
4.1. Rozważania projektowe

Jak zaznaczono w podpunkcie 1.2, zamiarem autora jest opracowanie efektywnego algorytmu obliczającego miary chiralności typu ^{SR}CM oraz implementacja tego algorytmu w postaci programu komputerowego. Ściślej rzecz ujmując, problem algorytmiczny związany jest tutaj z zadaniem optymalizacyjnym napotykanym w trakcie obliczeń ^{SR}CM. Po wygenerowaniu lustrzanego odbicia cząsteczki, konieczne jest odnalezienie najlepszego nałożenia obu obiektów, tj. takiego, dla którego wartość równania (2.2) definujiącego miarę ^{SR}CM jest najmniejsza.

W trakcie dyskusji zagadnienia postanowiono zastosować tutaj algorytm genetyczny. Przemawiała za tym przede wszystkim potencjalna łatwość implementacji oraz fakt, iż algorytmy genetyczne są metodą uniwersalną, stosowaną do wielu różnorakich problemów, także w zakresie metod komputerowo-wspomaganego projektowania leków. Przyjęty algorytm opisany jest dalej w podpunktach 4.3.2 oraz 4.3.3.

Jako że jednym z głównych wymagań sformułowanych w celu pracy była wydajność, należało rozważyć też, w jakim języku programowania należy napisać kod. Ostatecznie, autor zdecydował o zastosowaniu języka C++. Przeciwko temu rozwiązaniu przemawiała powszechnie znana czasochłonność pisania w C++. Z drugiej strony, w ostatnich latach język ten przeszedł znaczną modernizację przynoszącą nowe standardy: C++11 oraz C++14 (obecnie przygotowywana jest wersja C++17 [30]). Wniosły one liczne rozwiązania usprawniające rozwój aplikacji, m. in. funkcje lambda [31], automatyczną dedukcję typów, „sprytne” wskaźniki, nowy rodzaj pętli for, listy inicjalizacyjne oraz wiele innych [32, 33]. Ogólnie, nowe funkcjonalności nie przyniosły spadku wydajności języka jako takiego.

Z kolei kwestia szybkości działania spowodowała zainteresowanie możliwością implementacji również na procesory kart graficznych. W związku z tym zastosowano budowę modularną aplikacji. Każdy moduł wydzielony został za pomocą *namespace* [34] gdzie dwa główne to *Chirmes* oraz *Chirmes::GPU*. Budowa modularna uwzględniała też założenie działania aplikacji na różnych platformach: Apple OS X, Microsoft Windows, a perspektywicznie także Linux.



Rysunek 4.1. Blokowy schemat działania aplikacji

Powstały program komputerowy realizujący cel tej pracy został nazwany Chirmes od słów **Chir**ality **Me**asure**s**. Schemat przedstawiający poszczególne etapy działania aplikacji wraz z podziałem modułowym znajduje się na Rysunek 4.1. Dokładny opis modułów znajduje się w dalszych sekcjach 4.3.1, 4.3.2, 4.3.3, 4.3.4, 4.3.5.

W przebiegu programu, pierwszym krokiem jest załadowanie oraz interpretacja pliku konfiguracyjnego main_cfg.json znajdującego się w głównym katalogu programu. Plik ten zawiera informacje na temat wartości opcji aplikacji oraz ścieżki do struktur obliczanych cząsteczek. Dla każdego związku chemicznego istnieje możliwość ustawienia oddzielnych parametrów algorytmu genetycznego. Dzięki zastosowaniu pliku konfiguracyjnego, użytkownik może w wygodny sposób ustawiać bardzo rozbudowane kolejki obliczeń, także ze zróżnicowanymi parametrami. Dokładna lista opcji wraz z opisami i akceptowalnymi wartościami znajduje się w dodatku A.

Konfiguracja opcji zostaje dalej przekazana do modułu *BatchRunner*. Na podstawie ustawień zdefiniowanych w parametrze *mode* za pomocą pomocniczego obiektu *ComputeEngineProvider* tworzona jest odpowiednia instancja klasy *ComputeEngine*. Klasa *ComputeEngineProvider* realizuje wzorzec „Fabryki” tworząc odpowiednią instancję implementującą interfejs *IComputeEngine*. Dzięki wykorzystaniu takiego mechanizmu kod zwracający

instancje *ComputeEngine* może być wspólny dla różnych platform. Dodatkowo, jest on niezbędny w przypadku podziału modułowego na poziomie projektów (gdzie implementacja *CudaComputeEngine* znajduje się w oddzielnym projekcie i jest dołączana do projektu jako statycznie linkowana biblioteka, dokładniejszy opis w sekcji 4.4.1). Po utworzeniu odpowiedniej implementacji następuje iteracja po wszystkich zleconych zadaniach (jedno zadanie równa się jednemu plikowi wejściowemu), *BatchRunner* zleca przetworzenie pliku wejściowego (z opisem geometrycznym związku chemicznego) pomocniczej klasie *ComputeEngineMolLoader* (więcej w 4.3.1) do postaci klasy *StrippedMol* wraz z jej lustrzanym odbiciem.

Tak przygotowane dane, wraz z parametrami algorytmu genetycznego dla aktualnego pliku wejściowego przekazywane są do instancji *ComputeEngine*, który to za pomocą algorytmu genetycznego poszukuje optymalnego nałożenia cząsteczki chemicznej i jej lustrzanego odbicia. Sposób działania algorytmu genetycznego wraz z funkcją liczącą dopasowanie (czyli wyznaczającą miarę chiralności) przedstawiony został na Rysunek 4.2 oraz Rysunek 4.11, a ich opis znajduje się w podpunktach 4.3.2 i 4.3.3.

Po przeprowadzeniu obliczeń dla danego pliku wejściowego wyniki przekazywane są do obiektu klasy *ResultWriter*(sekcja 4.3.5). Po przetworzeniu każdego zadania z pliku konfiguracyjnego następuje wypisanie wyników, oraz, w razie potrzeby, wizualizacja wyników za pomocą modułu *Viewer* (podpunkt 4.3.4).

4.2. Środowisko programistyczne

Aplikacja Chirmes została przygotowana z wykorzystaniem opisanych dalej konfiguracji sprzętowych oraz narzędzi programistycznych. Odpowiedni dobór narzędzi, zwłaszcza dodatkowych bibliotek programistycznych pozwala w znaczący sposób przyspieszyć proces tworzenia aplikacji, a także zmniejszyć nakłady czasowe potrzebne na testowanie poszczególnych funkcjonalności aplikacji (zwłaszcza przy korzystaniu z popularnych bibliotek open-source).

4.2.1. Konfiguracje sprzętowe

W związku z projektowanym działaniem na różnych platformach, prace nad aplikacją zostały wykonane na dwóch różnych komputerach. Ich konfiguracje przedstawione są w Tablicy 4.1.

Tablica 4.1. Wykorzystana konfiguracje sprzętowe

	Komputer 1	Komputer 2
Typ	MacBook Pro 2013	PC
Procesor	Intel Core i5 2.4 Ghz	Intel Core i7 4.1 Ghz
Pamięć RAM	8 GB	32 GB
Karta graficzna	Intel Iris	NVIDIA GeForce 660 Ti 2GB
System operacyjny	OS X 10.11.6	Windows 10
Kompilator	clang-703.0.31	MSVCP 15
Środowisko deweloperskie	Apple Xcode 7.3.1	Microsoft Visual Studio 15 Update 1
CUDA SDK	brak	CUDA SDK 8.0 RC1

4.2.2. Wykorzystane narzędzia

- **Apple Xcode** – zintegrowane środowisko programistyczne (*Integrated Development Environment* – IDE), wykorzystywane do pracy w systemie OS X. Głównie stosowane do tworzenia aplikacji działających na urządzeniach iPhone/iPad (w języku Objective-C, Swift), posiada także tryb tworzenia aplikacji w języku C++. Zawiera wbudowane narzędzie do dynamicznej analizy kodu (debugger),
- **Microsoft Visual Studio 2015** – zintegrowane środowisko programistyczne, wykorzystywane do pracy w systemie Windows,
- **Git** – system kontroli wersji plików. Dla każdego pliku zapisywana jest historia, dzięki czemu, w razie potrzeby, można wracać do poprzednich wersji. Umożliwia także jednoczesną pracę wielu osób nad jednym plikiem dzięki możliwości synchronizacji ze zdalnym repozytorium,
- **Bitbucket** – serwis internetowy służący jako zdalne repozytorium Git,
- **SourceTree** – graficzny klient systemu Git,
- **TeamViewer** – aplikacja umożliwiająca pełny, graficzny zdalny dostęp do innego komputera. Z racji fizycznej lokalizacji komputera wyposażonego w kartę NVIDIA praca wykonywania z jego użyciem była prowadzona poprzez zdalne sesje za pomocą programu TeamViewer,
- **NVIDIA Profiler** – narzędzie firmy NVIDIA zbierające statystyki dotyczące działania kerneli na platformie CUDA. Posiada funkcjonalność wykrywania najbardziej

obciążających części kodu, jak i sposobu dostępu do pamięci wraz z sugestiami na temat poprawy wydajności. Dołączona standardowo do pakietu CUDA SDK,

- **Visual Studio Profiler** – narzędzie firmy Microsoft zbierające statystyki dotyczące działania poszczególnych części programu w systemie Windows,
- **Apple Instruments** – narzędzie firmy Apple zbierające statystyki dotyczące działania poszczególnych części programu w systemie OS X,
- **CUDA memcheck** – narzędzie firmy NVIDIA szukające problemów dotyczących zarządzania pamięcią w aplikacjach działających na platformie CUDA. Wykrywa także próby dostępu do nieprzydzielonych obszarów pamięci. Dołączona standardowo do pakietu CUDA SDK.

4.2.3. Wykorzystane zewnętrzne biblioteki programistyczne

- **Eigen** – zestaw wysokopoziomowych bibliotek dla języka C++ ułatwiający pracę na macierzach, wykorzystujący rozbudowany system szablonów, w tym m.in. metaprogramowanie. Domyślnie wspiera wektoryzację, wykorzystując dodatkowe instrukcje procesora jak SSE 2/3/4 na platformie x86 oraz NEON w architekturze ARM (zarówno 32-bit jak i 64-bit). Źródła tej biblioteki dostępne są publicznie,
- **Open Babel** – zestaw aplikacji oraz bibliotek programistycznych służących do interakcji z danymi chemicznymi. Obsługuje ponad 110 różnych formatów plików chemicznych, m. in PDF, SDF, XYZ. Źródła tej biblioteki dostępne są publicznie,
- **SDL** (Simple DirectMedia Layer) – biblioteka umożliwiająca tworzenie gier i aplikacji multimedialnych. Dzięki wieloplatformowści wykorzystywana jest do integracji z OpenGL. Źródła tej biblioteki dostępne są publicznie,
- **OpenGL** (Open Graphics Library) – standard uniwersalnego interfejsu programistycznego służącego do budowy grafiki dwu- i trój-wymiarowej. Za pomocą tego interfejsu możliwe jest bezpośrednie operowanie na poziomie karty graficznej,
- **jsoncpp** – biblioteka służąca do manipulacji danymi zapisanymi w formacie JSON (zapis, odczyt). Zbudowana w oparciu o C++11. Źródła tej biblioteki dostępne są publicznie,

- **spdlog** – biblioteka umożliwiająca logowanie danych na konsoli z poziomu kodu. Do jej zalet należy zaliczyć wieloplatformowość, wsparcie logowania asynchronicznego, szybkość działania, łatwość integracji (jeden plik nagłówkowy bez konieczności kompilacji). Dodatkowo, umożliwia jednoczesny zapis do kilku miejsc jednocześnie. Zbudowana w oparciu o C++11. Źródła tej biblioteki dostępne są publicznie,

- **CUDA SDK** – zestaw bibliotek programistycznych umożliwiających programowanie na platformę CUDA z wykorzystaniem wysokopoziomowych języków programowania jak C++ czy Python. W projekcie użyta została wersja 8.0 RC1, która jest jedynie wersją testową nadchodzącej wersji 8.0.

- **Thrust** – biblioteka wspomagająca tworzenie aplikacji działających równolegle, dystrybułowana razem z CUDA SDK. Wzorowana na STL z biblioteki standardowej C++ pozwala na konstruowanie algorytmów działających na CUDA bez konieczności pisania bezpośrednio kodu specyficznie dla CUDA. Ponadto zapewnia automatyczne zarządzanie pamięcią globalną znajdującą się na karcie graficznej, upraszcza także wymianę danych pomiędzy pamięcią komputera, a pamięcią karty.

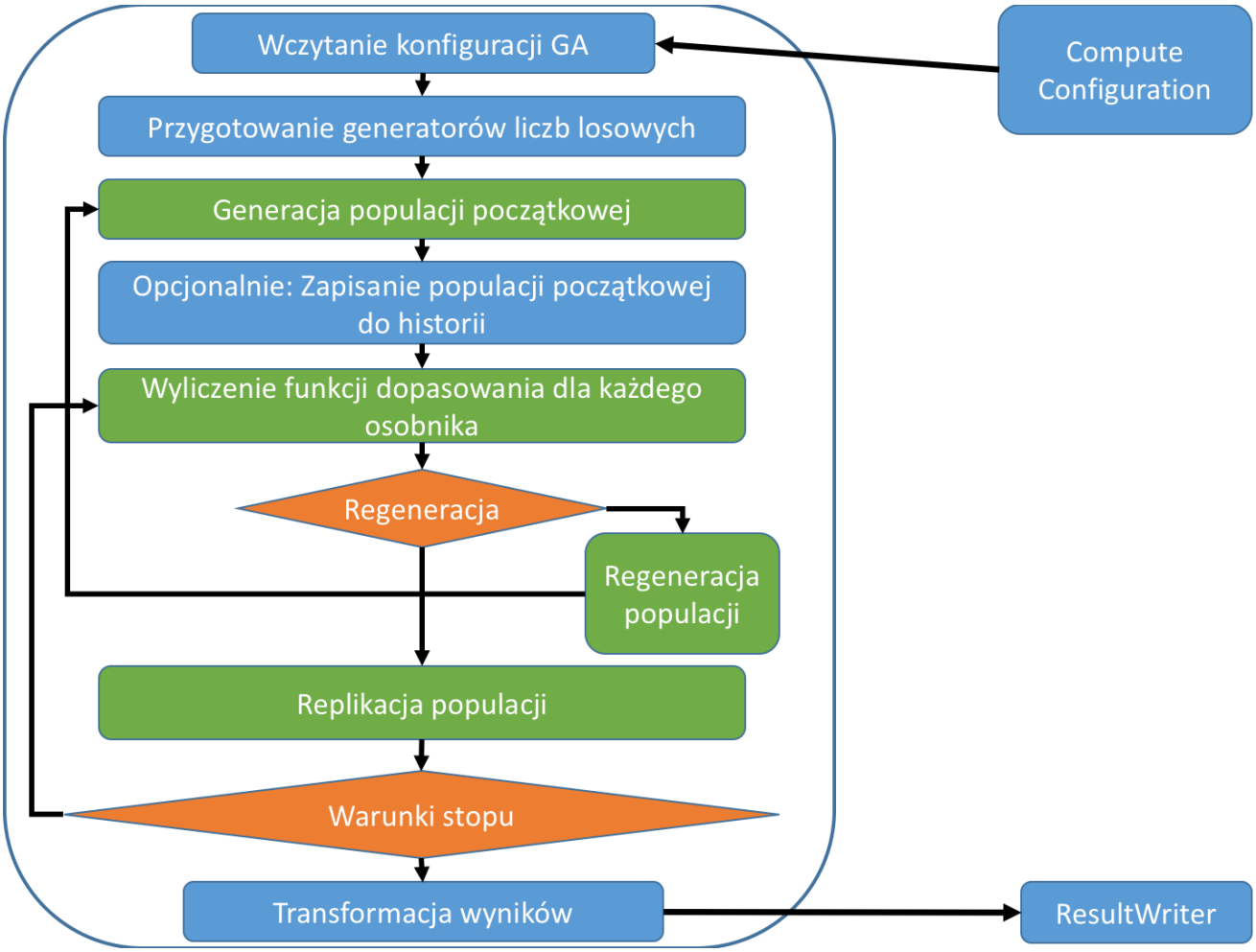
4.3. Realizacja poszczególnych modułów

4.3.1. Moduł wczytywania danych wejściowych

Moduł wczytywania danych wejściowych dostępny jest w namespace *Chirmes::input*. Składa się z dwóch podstawowych części - klasy *Config*, służącej do odczytywania plików konfiguracyjnych oraz pomocniczej struktury *ComputeEngineMolLoader*, za pomocą której następuje załadowanie struktury wejściowej cząsteczki.

Wykorzystując bibliotekę *jsoncpp* klasa *Config* ładuje zawartość pliku main_cfg.json. Poprzez publiczny interfejs zapewniona jest możliwość odczytu wszystkich wymaganych ustawień aplikacji oraz listy wejściowych plików z cząsteczkami wraz z parametrami konfigurującymi działanie algorytmu genetycznego dla każdej cząsteczki.

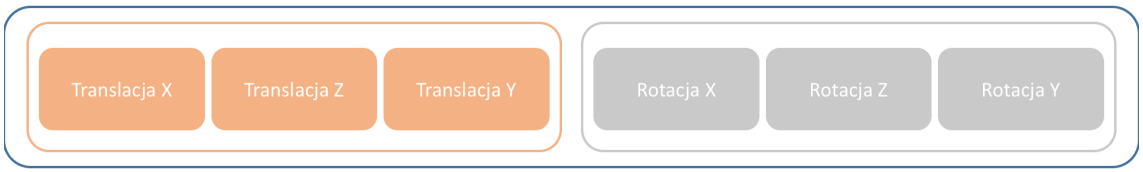
ComputeEngineMolLoader posiada tylko jedną metodę, w której obiekt klasy *MolLoader* z użyciem biblioteki *Open Babel* ładuje do pamięci zawartość pliku wejściowego. Następnie, przepisywane są pozycje wszystkich atomów, wraz z ich łączeniami do obiektu klasy *StrippedMol*. Pozycje atomów przechowywane są w czterorzędowej macierzy w której każda kolumna opisuje jeden atom. Poszczególne rzędy odpowiadają składowym x, y, z, w.



Rysunek 4.2. Schemat zaimplementowanego algorytmu genetycznego

Parametry x, y, z, to współrzędne atomów w kartezjańskim układzie współrzędnych, zaś w to właściwość atomu, stosowana do ważenia w obliczeniu miary chiralności. W ostatnim kroku następuje utworzenie lustrzanego odbicia poprzez wymnożenie macierzy pozycji atomów (oraz łączeń) przez macierz lustrzanego odbicia, domyślnie wykonywane jest odbicie przez płaszczyznę XY. Przykład takiej operacji dla cząsteczki metanu opisuje równanie (4.1)

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} -0.366 & -0.009 & -0.009 & -0.009 & -1.436 \\ -0.88 & -1.889 & -0.376 & -0.376 & -0.88 \\ -2.702 & -2.702 & -1.828 & -3.575 & -2.702 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} \tag{4.1}$$



Rysunek 4.3. Geny zakodowane w genotypie osobnika

4.3.2. Algorytm genetyczny

Algorytm genetyczny jest najważniejszą częścią aplikacji Chirmes, ponieważ rozwiązuje problem optymalizacyjny napotykany w obliczeniu miar chiralności ${}^{SR}CM$, czyli znalezienie optymalnego nałożenia cząsteczki i jej lustrzanego odbicia. Po załadowaniu współrzędnych cząsteczki i wygenerowaniu lustrzanego odbicia, konieczne jest - przez serię rotacji i translacji - znalezienie najlepszego nałożenia obu obiektów, tj. takiego, dla którego wartość miary chiralności jest najmniejsza.

Od początku projektu, zdecydowano o stworzeniu własnej implementacji zamiast korzystania z gotowych bibliotek. Po pierwsze, umożliwiło to lepsze panowanie nad aplikacją, po drugie zaś najpopularniejsza stosowana biblioteka, GAlib [35] od dłuższego czasu nie była aktualizowana (ostatnia wersja pochodzi z 2007 roku), a co za tym idzie mogłaby sprawiać problemy przy integracji z projektem, który wykorzystuje C++14.

W aplikacji zdecydowano się na zastosowanie standardowego algorytmu genetycznego, którego ogólny przebieg został przedstawiony na Rysunek 4.2. Zastosowane zostały jednak pewne modyfikacje dotyczącymi sposobu kodowania genów oraz wprowadzono pewien dodatkowy operator. Wybrano kodowanie genów w postaci liczb zmiennoprzecinkowych (nie zaś binarnie), gdyż te lepiej odwzorowują problem, który aplikacja ma rozwiązywać czyli poszukiwanie najlepszego przesunięcia oraz rotacji, a także zapewniają lepszą precyzję. Ponadto, korzystanie z binarnie zakodowanego genotypu wymuszałoby dużo bardziej skomplikowane operatory mutacji i krzyżowania (które posiadałyby skomplikowane warunki ograniczające), ponieważ nie wszystkie wartości wygenerowane podczas tych operacji miałyby sens fizyczny[1]. Utworzony genotyp jest przedstawiony na Rysunek 4.3. Możemy wyróżnić w nim część dotyczącą translacji oraz rotacji. Każda z nich dzieli się na 3 zmienne, po jednej dla każdej składowej wektora translacji lub rotacji: x,

[1] Inspiracja do zastosowania kodowania w postaci liczb rzeczywistych była [21]

Listing 4.1. Sposób przechowywania informacji o genotypie

```
1 struct Genotype
2 {
3     using ElementType = float;
4     union d {
5         struct s { ElementType x, y, z, rotX, rotY, rotZ; } single;
6         ElementType tab[6];
7     } data;
8 }
```

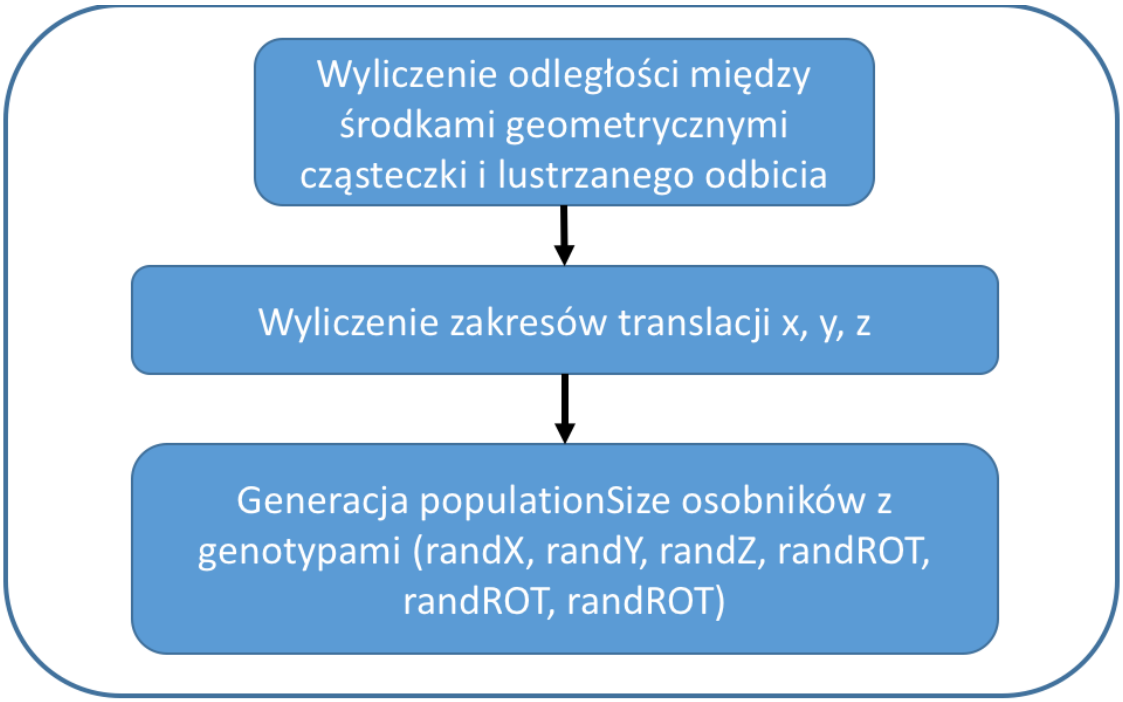
y, z. Każda wartość jest zapisana w 32 bitowej zmiennej typu *float*, który został wybrany zamiast *dobule* z dwóch powodów dotyczących wydajności:

- korzystając z wektoryzacji SSE/AVX (której użycie zapewnia biblioteka *Eigen*), można przeprowadzać dwa razy więcej operacji na liczbach pojedynczej precyzji w tym samym czasie
- *float* jest domyślnym typem na platformie CUDA, jest ona zoptymalizowana do przeprowadzania działań właśnie na tym typie, wykorzystanie *double* w praktyce podwaja czas działania

Warto też wspomnieć o tym, iż genotyp w pamięci komputera przechowywany jest w formie pośredniej, przedstawionej na Listingu 4.1, t.j. w postaci sześciu zmiennych (*x, y, z, rotX, rotY, rotZ*), tworzących unię razem z sześcioelementową tablicą typu *float*. Takie podejście pozwala na dostęp do każdego elementu poprzez nazwę albo indeks tablicy. Dopiero kiedy potrzebne jest wykonanie obliczeń z wykorzystaniem genotypu, zamieniany jest on w macierz transformacji 4x4 [36]. Macierz zostaje przygotowana za pomocą funkcji pomocniczych biblioteki *Eigen* według wzoru (4.2):

$$matrix = (translation * translationFromOrigin * rotation * translationToOrigin) \tag{4.2}$$

Kolejna modyfikacja standardowego algorytmu genetycznego została zastosowana przy losowej generacji populacji. Z definicji problemu znajdywania dopasowania możemy zauważyć, że zakres w którym może następować translacja jest ograniczony, np. jeśli różnica między środkami geometrycznymi dwóch cząsteczek wynosi 10 jednostek dla



Rysunek 4.4. Algorytm generacji populacji losowej wykonywany na CPU

składowej x, nie ma sensu generować osobników dla których przesunięcie x będzie wynosić 100. Stąd też generowane osobniki mają następujące ograniczenia:

- zakres rotacji ograniczony jest pomiędzy wartościami zdefiniowanymi w pliku konfiguracyjnym, *rotationRangeMin rotationRangeMax*,
- w przypadku translacji jej zakres wyliczany jest w następujący sposób:
  - wyznacza się środek geometryczny źródłowej cząsteczki i jej lustrzanego odbicia,
  - jeśli wektor różnicy jest dodatni, wtedy zakres dla każdej składowej wynosi: od 0 do $abs\_diff * rangeMultiplier + rangeExtension$, gdzie *abs_diff* jest absolutną różnicą odległości dla danej składowej, *rangeMultiplier* jest wartością zdefiniowaną w pliku konfiguracyjnym, *rangeExtension* jest wartością zdefiniowaną w pliku konfiguracyjnym,
  - jeśli wektor różnicy jest ujemny, wtedy zakres dla każdej składowej wynosi: od $-abs\_diff * rangeMultiplier - rangeExtension$ do 0, gdzie *abs_diff* jest absolutną różnicą odległości dla danej składowej, *rangeMultiplier* jest wartością zdefiniowaną w pliku konfiguracyjnym, *rangeExtension* jest wartością zdefiniowaną w pliku konfiguracyjnym.

Zastosowanie formuły rozszerzającej górny/dolny zakres ponad odległość między środkami pozwala na większą wariancję możliwych przesunięć. Manipulując parametrami *rangeMultiplier* oraz *rangeExtension* możemy zdefiniować, jak duża jest to wariancja. Należy tutaj wspomnieć, że poprawne określenie warunków początkowych jest istotne jeśli chodzi o szybkość (oraz możliwość) znalezienia poprawnego rozwiązania, zwłaszcza jeśli chodzi o parametry dotyczące rotacji. Schemat ogólny algorytmu generacji populacji losowej został przedstawiony na Rysunek 4.4.

Na tym etapie Chirmes zapisuje do pamięci generatory liczb losowych stworzone dla translacji. Będą one wykorzystane później na potrzeby mutacji, z tymi samymi zakresami. Wszystkie generatory liczb losowych wykorzystane w implementacji działającej na CPU zbudowane są w oparciu o algorytm *Mersenne Twister 19937* [2] udostępniany przez bibliotekę standardową *random* języka C++11 w postaci klasy *std::mt19937* [37]. Dla wszystkich wartości losowych został zastosowany rozkład jednostajny, w zależności od potrzeby zwracający liczby całkowite typu *int* (*std::uniform_int_distribution<int>*) lub rzeczywiste typu *float* (*std::uniform_real_distribution<float>*).

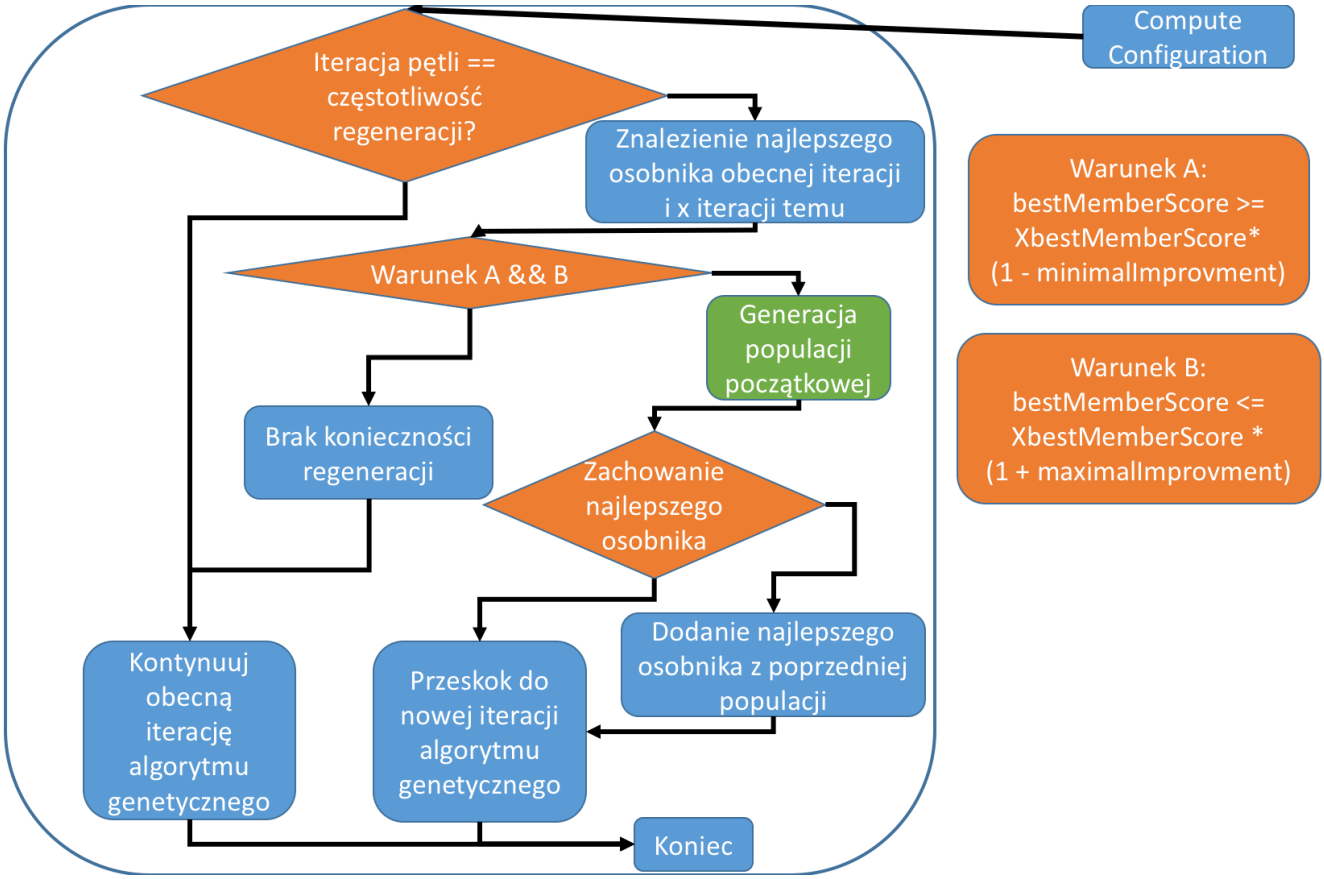
Po utworzeniu pierwszej losowej populacji algorytm jest gotowy do wykonania właściwej pracy, tj. poszukiwania optymalnego nałożenia. Jeśli włączone są opcje diagnostyczne, przed uruchomieniem właściwej części tworzony jest także bufor na historię całej populacji i zapisywana jest pierwsza populacja startowa.

Następnym krokiem jest obliczenie wartości funkcji dopasowania czyli wyznaczenie miary chiralności dla każdego osobnika, schemat działania tego algorytmu przedstawia Rysunek 4.11, jego dokładny opis algorytmu znajduje się w podpunkcie 4.3.3.

W zależności od ustawień diagnostycznych algorytm po wyliczeniu wartości funkcji dopasowania dla całej populacji może wypisać informację o aktualnym numerze iteracji oraz wartość funkcji dopasowania dla pewnej, parametrem *debugPrintoutSize*, liczby pierwszych najlepszych osobników. Częstotliwość wypisywania określa parametr *debugPrintoutFrequncy*.

Kolejnym krokiem algorytmu jest warunkowa regeneracja populacji. Tutaj również mamy do czynienia z modyfikacją względem oryginalnego schematu algorytmu genetycznego. Operator ten został wprowadzony, aby algorytm mógł skuteczniej reagować na brak

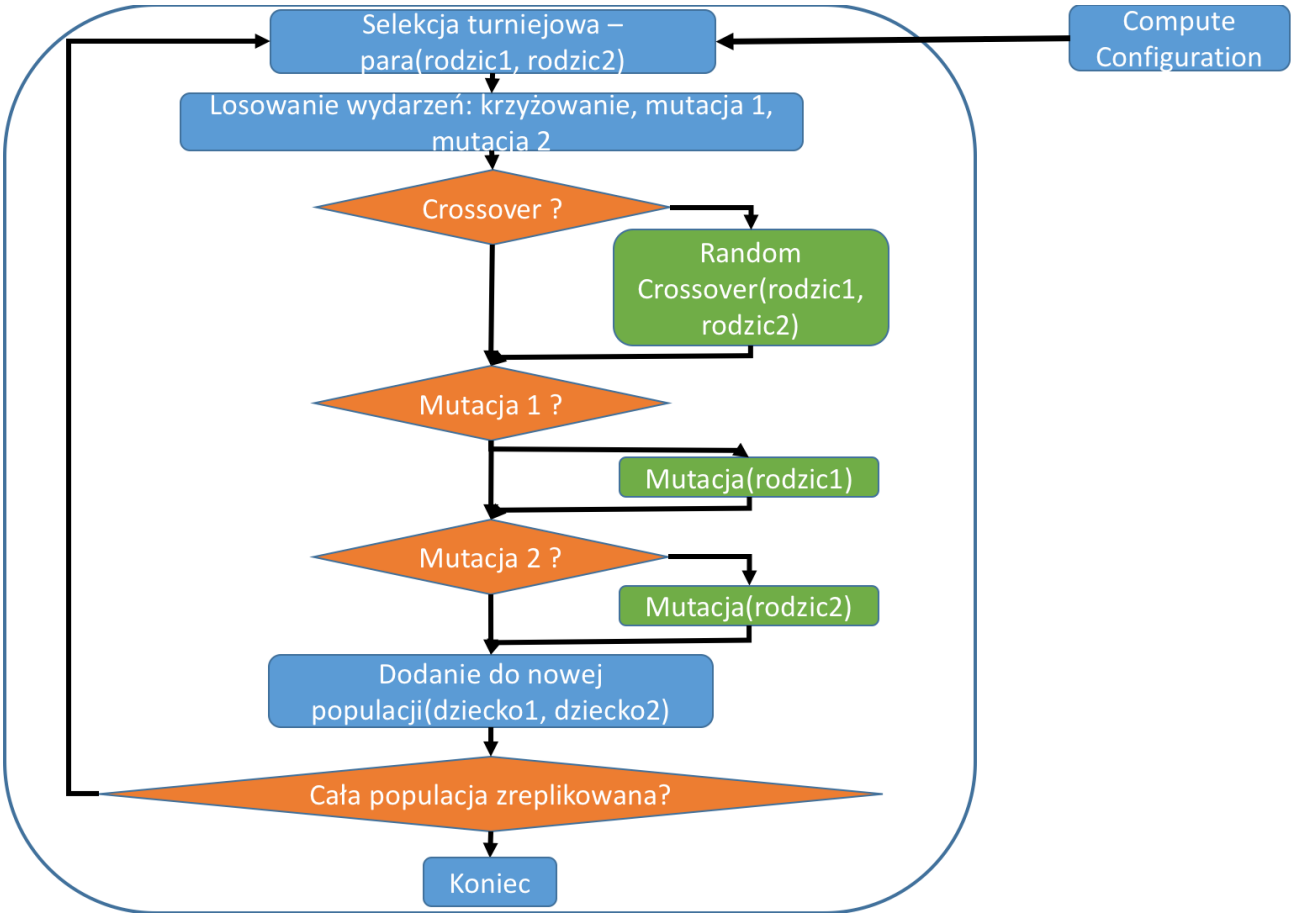
[2] Algorytm ten posiada okres $2^{19937}-1$ w zupełności wystarczający na potrzeby generowania losowych wartości dla osobników. Ponadto, algorytm ten znany jest ze swojej szybkości działania.



Rysunek 4.5. Algorytm regeneracji populacji wykonywany na CPU

poprawy najlepszego wyniku względem wyników z poprzednich iteracji. W przypadku wykrycia braku poprawy tworzy on na nowo całą populację losowo, według tych samych warunków co generacja pierwszej populacji, ponadto jest możliwość odziedziczenia najlepszego osobnika z istniejące populacji. Schematyczny przebieg procesu regeneracji przedstawia Rysunek 4.5. Pierwszym warunkiem który musi zostać spełniony, aby została podjęta próba regeneracji jest przejście odpowiedniej liczby iteracji, konfigurowalnej parametrem *populationRegenerateTime*. Następnie sprawdzane jest czy nastąpiła poprawa w ciągu tych iteracji. Jeśli wynik najlepszego osobnika zmniejszył się mniej niż wartość zdefiniowana przez *populationRegenerateMinimalImprove* lub wzrósł o mniej niż *populationRegenerateMaximalImprove*, nastąpi wówczas proces regeneracji. Taka konstrukcja warunku wejściowego ma na celu uruchomienie regeneracji w przypadku braku poprawy, jednocześnie chroni przed zbyt częstą regeneracją, jeśli algorytm nie zdążył poprawić wyniku. Jeśli regeneracja miała miejsce to algorytm od razu przechodzi do następnej iteracji bez kroków dotyczących replikacji.

Istotą algorytmu genetycznego są 3 następujące etapy: selekcja, krzyżowanie oraz mutacja wykonywane do momentu stworzenia nowej populacji. To w trakcie tych etapów



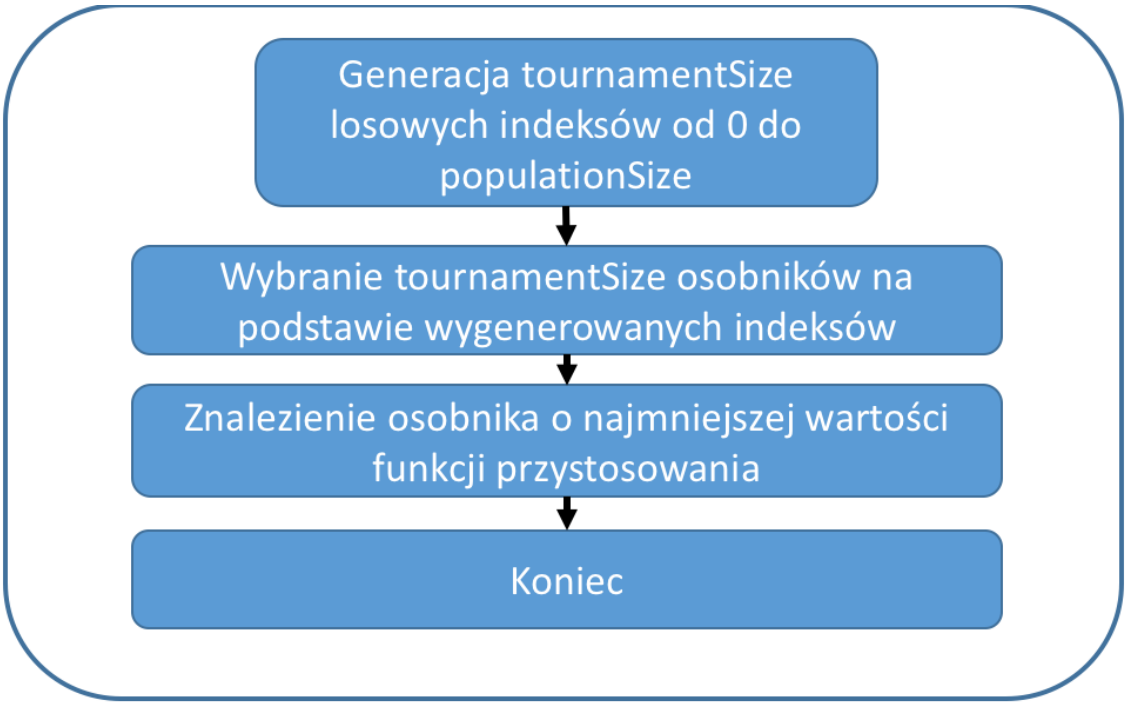
Rysunek 4.6. Algorytm replikacji populacji wykonywany na CPU

następuje wymiana informacji genetycznej między osobnikami, dzięki której algorytm potrafi odnaleźć poprawne rozwiązanie.

Selekcja została tutaj zaimplementowana w postaci metody turniejowej. Algorytm ten losuje spośród całej populacji zdefiniowaną parametrem *tournamentSize* liczbę osobników, spośród których wybiera jednego o najlepszej (najniższej) wartości funkcji dopasowania. Schemat tej operacji przedstawia Rysunek 4.7. Sterując parametrem *tournamentSize*, możemy wpływać na wielkość naporu selekcyjnego. Duża wartość zwiększa napór, co jednocześnie zmniejsza różnorodność populacji.[3].

Po wybraniu dwóch rodziców następują trzy losowania (przedstawione na Listingu 4.2) decydujące o tym, czy kolejne operacje genetyczne zostaną wykonane, czy też dany osobnik trafi do następnej populacji bez jakiejkolwiek zmiany.

[3]Silny napór selekcyjny wspomaga przedwczesną zbieżność algorytmów genetycznych, za to mały napór selekcyjny może spowodować, że przeszukiwanie będzie mało efektywne. Dlatego ważne jest, aby utrzymać równowagę między tymi czynnikami [21]



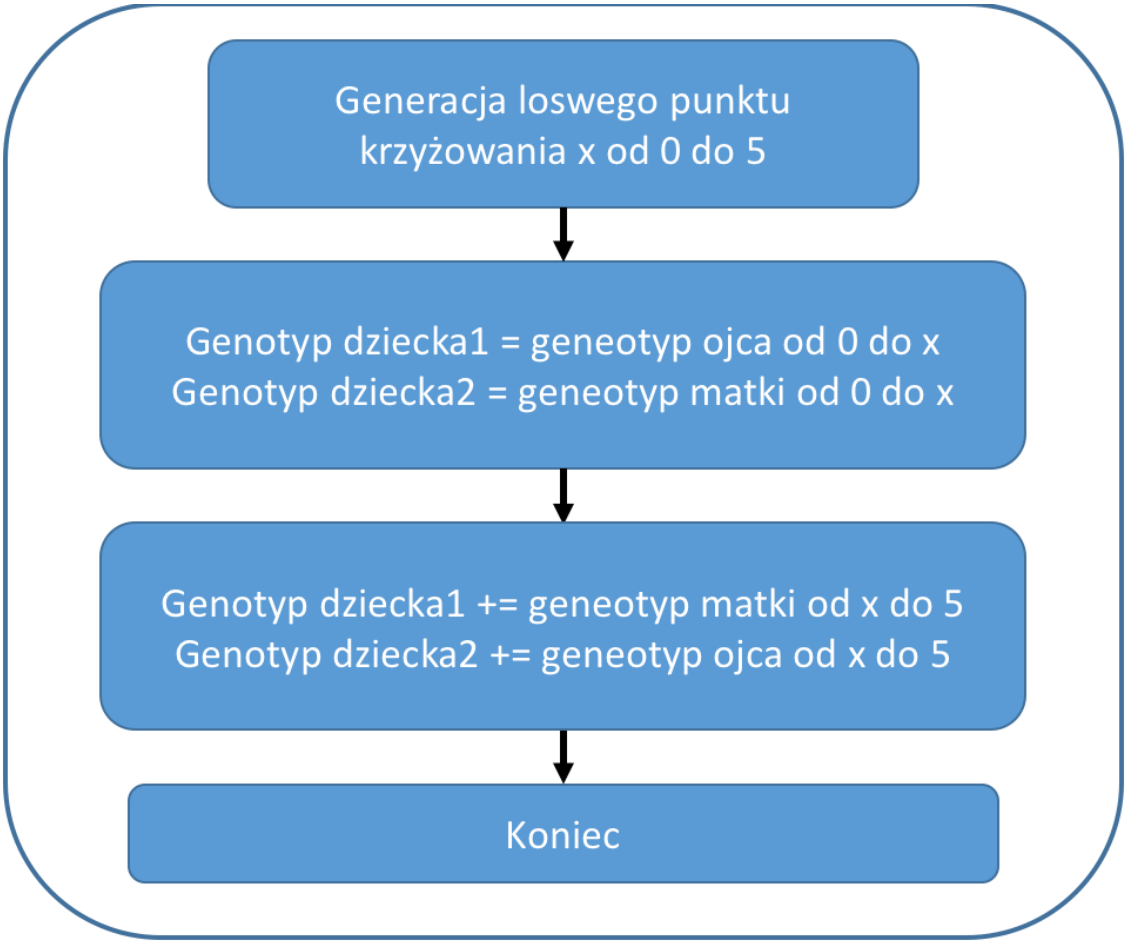
Rysunek 4.7. Algorytm selekcji turniejowej wykonywany na CPU

Listing 4.2. Decyzja o wystąpieniu krzyżowania oraz mutacji

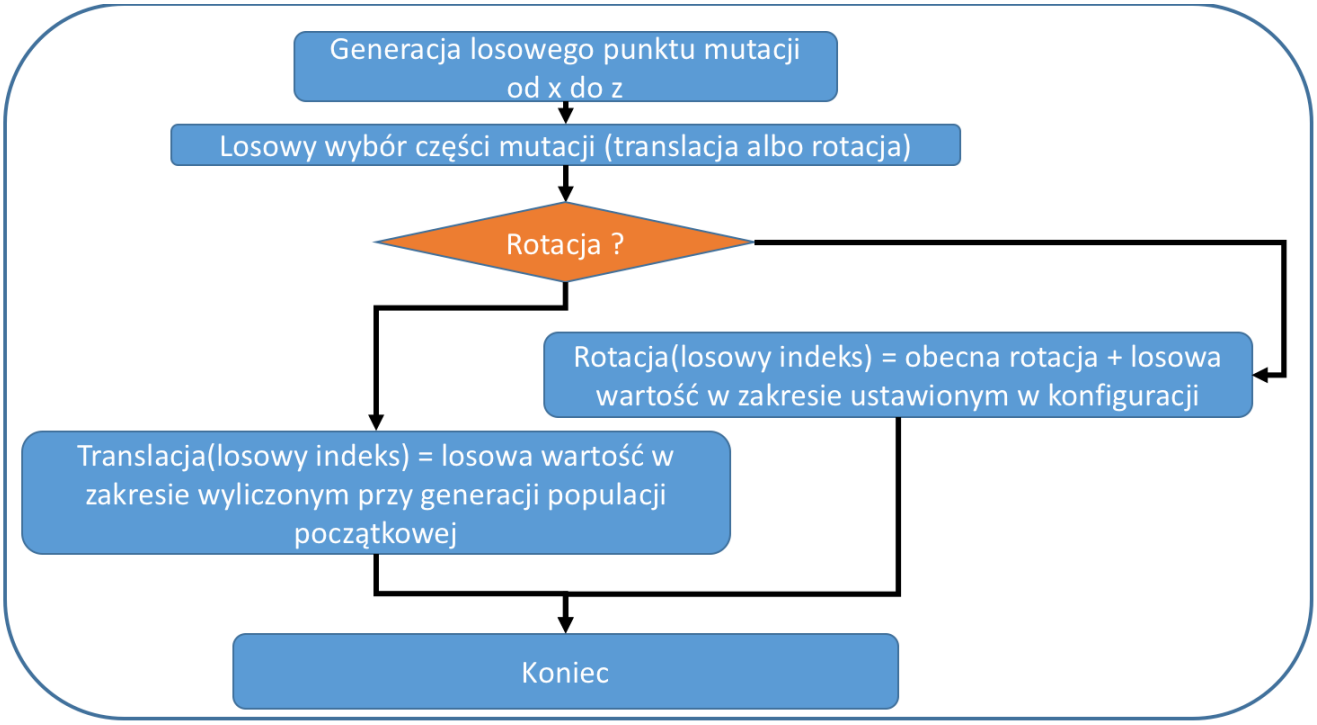
```
1 crossoverEvent = (uniformRandomFloat() < computeConfiguration.crossoverProbability);
2 mutationFirstEvent = (uniformRandomFloat() < ↙
      ↳ computeConfiguration.mutationProbability);
3 mutationSecondEvent = (uniformRandomFloat() < ↙
      ↳ computeConfiguration.mutationProbability);
```

Pierwszą operacją jest krzyżowanie, którego prawdopodobieństwo wykonania sterowane jest parametrem *crossoverProbability* W aplikacji zaimplementowano dwie metody krzyżowania, z ustalonym punktem wymiany informacji oraz z losowym. W obecnej konfiguracji zawsze używany jest ten drugi sposób, przedstawiony na Rysunek 4.8. Po wylosowaniu punktu krzyżowania dla pierwszego dziecka, następuje przepisanie wartości genów do tego punktu od ojca, druga część pochodzi od matki. Dla drugiego dziecka sytuacja jest analogiczna, z tym, że najpierw przepisywane są wartości od matki.

Kolejną operacją jest mutacja, zaprezentowana na Rysunek 4.9, wykonywana niezależnie dla każdego dziecka, w zależności od ustawionego prawdopodobieństwa parametrem *mutationProbability*. W trakcie samego procesu mutacji wykonywane są jeszcze dwa losowania sterujące. Po pierwsze, regulowane parametrem *rotationPartMutationProbability*, losowanie decydujące, czy w trakcie mutacji nastąpi zmiana wartości dotyczącej translacji



Rysunek 4.8. Algorytm losowego krzyżowania wykonywany na CPU



Rysunek 4.9. Algorytm losowej mutacji wykonywany na CPU

Listing 4.3. Tworzenie tymczasowego osobnika

```
1 genotypeTransformationMatrix = individual.getTransformationMatrixFromGenotype()
2 atoms = genotypeTransformationMatrix * mirroredMolAtomsMatrix
3 bonds = genotypeTransformationMatrix * mirroredMolBondsMatrix
4 StrippedMol individual(atoms, bonds);
```

czy też rotacji. Po drugie, losowanie indeksu, która składowa rotacji lub translacji ma zostać zmieniona, x, y, czy z

Po wybraniu genu, który zostanie poddany mutacji, jego nowa wartość ustalana jest według następujących zasad, w zależności od części która ma zostać zmieniona:

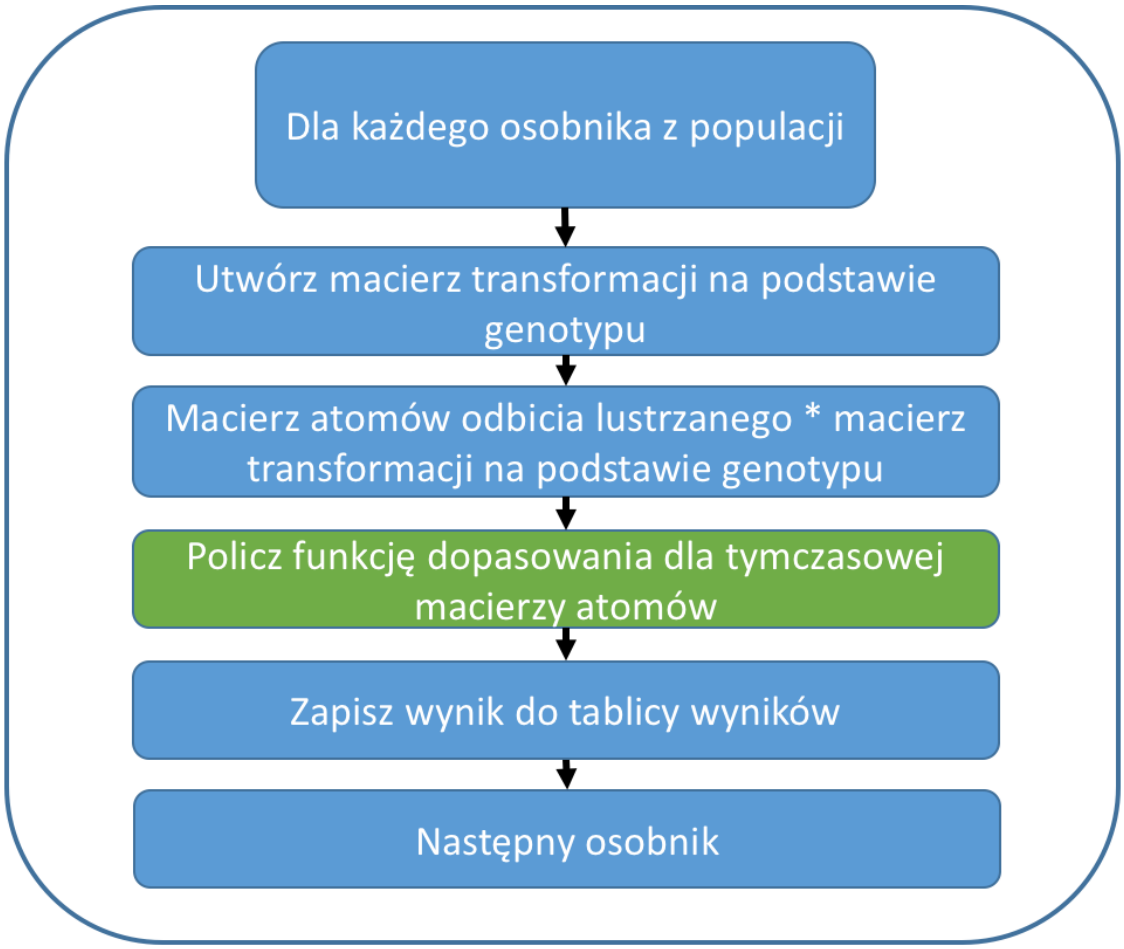
- **rotacja** – do istniejącej wartości dodawana/odejmowana jest liczba wylosowana w zakresie od *mutationRotationRangeMin* do *mutationRotationRangeMax*,
- **translacja** – dla każdej składowej x, y, z wykorzystywany jest oddzielny generator losowy zwracający nową wartość w zakresie wyliczonym przy generacji losowej populacji dla konkretniej składowej.

Po wykonaniu powyższych operacji tyle razy, ile wynosi *populationSize*, nowa populacja jest gotowa, by zastąpić starą. Tym samym, jeden pełen przebieg algorytmu genetycznego został wykonany.

4.3.3. Obliczenia miary chiralności

Funkcja dopasowania (miara chiralności) w Chirmesie określona jest wzorem (2.2). Postępowanie przedstawione jest z kolei na Rysunku 4.10.

Na początku, stworzony zostaje tymczasowy, rzeczywisty osobnik poprzez wymnożenie macierzy transformacji (będącej pochodną genotypu) przez macierz atomów, tak jak w Listingu 4.3 Po utworzeniu osobnika aplikacja wchodzi do najbardziej złożonej obliczeniowo części, czyli wyliczenia miary chiralności. Przedstawia to schemat pokazany na Rysunku 4.11. Na początku wyliczana jest macierz odległości między odpowiadającymi sobie atomami z lustrzanego odbicia i cząsteczki źródłowej. Aktualnie aplikacja liczy tylko odległość w przestrzeni geometrycznej, nie bierze pod uwagę właściwości fizycznych/chemicznych – taka funkcjonalność pojawi się w przyszłych wersjach. Kolejnym krokiem jest poszukiwanie najmniejszej sumy odległości pomiędzy atomami, ten proces powtarzany jest



Rysunek 4.10. Obliczenie funkcji dopasowania populacji, wykonywane na CPU

tyle razy ile wynosi wartość parametru *randomSelectionRounds*. W trakcie każdej iteracji poszukiwania wykonywane są następujące kroki:

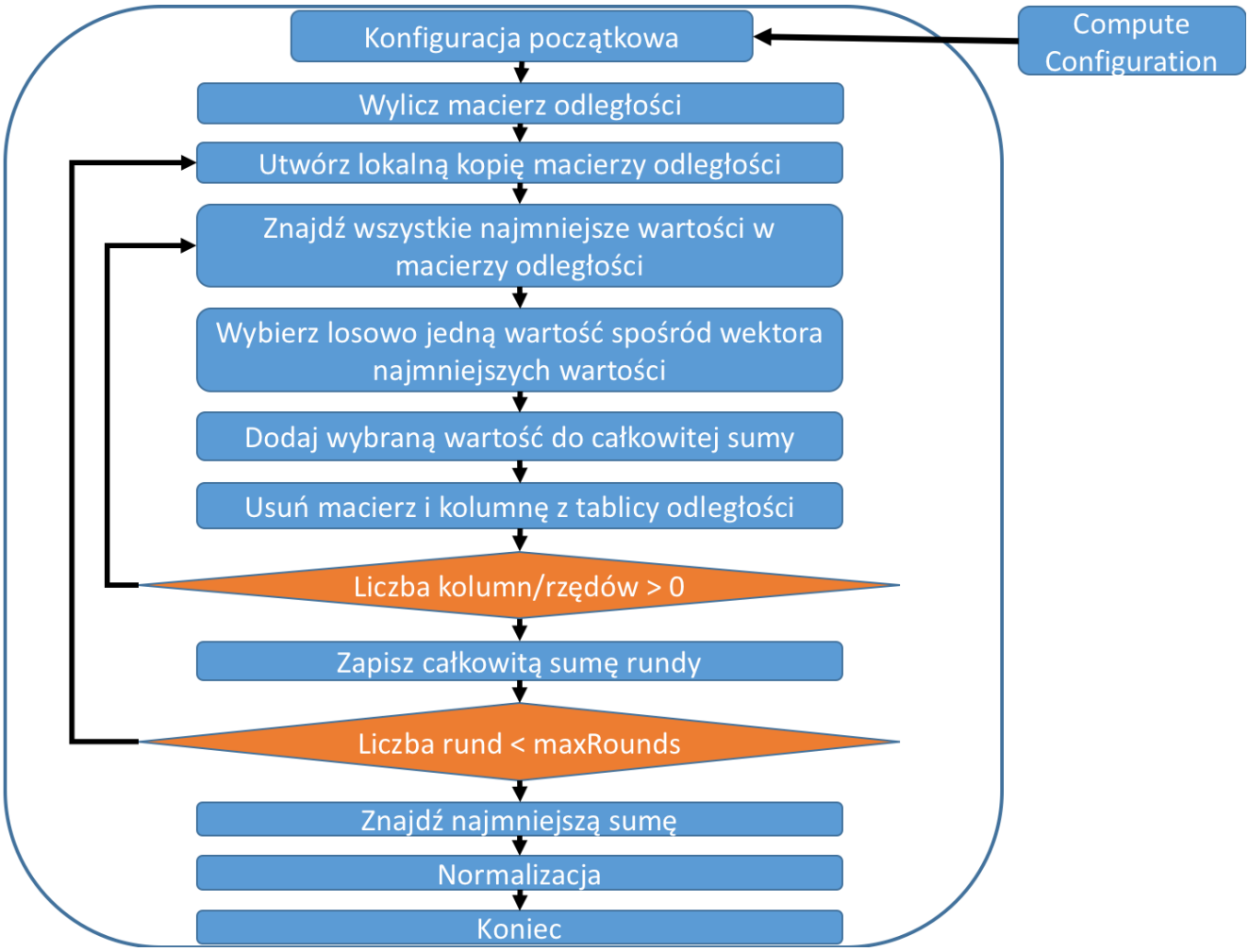
1) znalezienie najmniejszych wartości w lokalnej kopii macierzy odległości,

2) losowy wybór jednej pośród najmniejszych wartości. Ponieważ wartości tych może być więcej niż jedna, pojawia się konieczność przeprowadzenia tego procesu wielokrotnie, z tego powodu przeprowadzanych jest kilka rund losowania,

3) wybrana odległość dodawana jest do zbiorczej odległości, rząd oraz kolumna w której znajdowała się najmniejsza wartość zostają usunięte z lokalnej kopii macierzy odległości,

4) proces jest powtarzany aż do usunięcia wszystkich rzędów/kolumn.

Ostatnim krokiem jest tu znalezienie najmniejszej wartości spośród wszystkich wyliczonych w poszczególnych rundach i jej normalizacja poprzez podzielenie przez liczbę atomów w badanej cząsteczce.

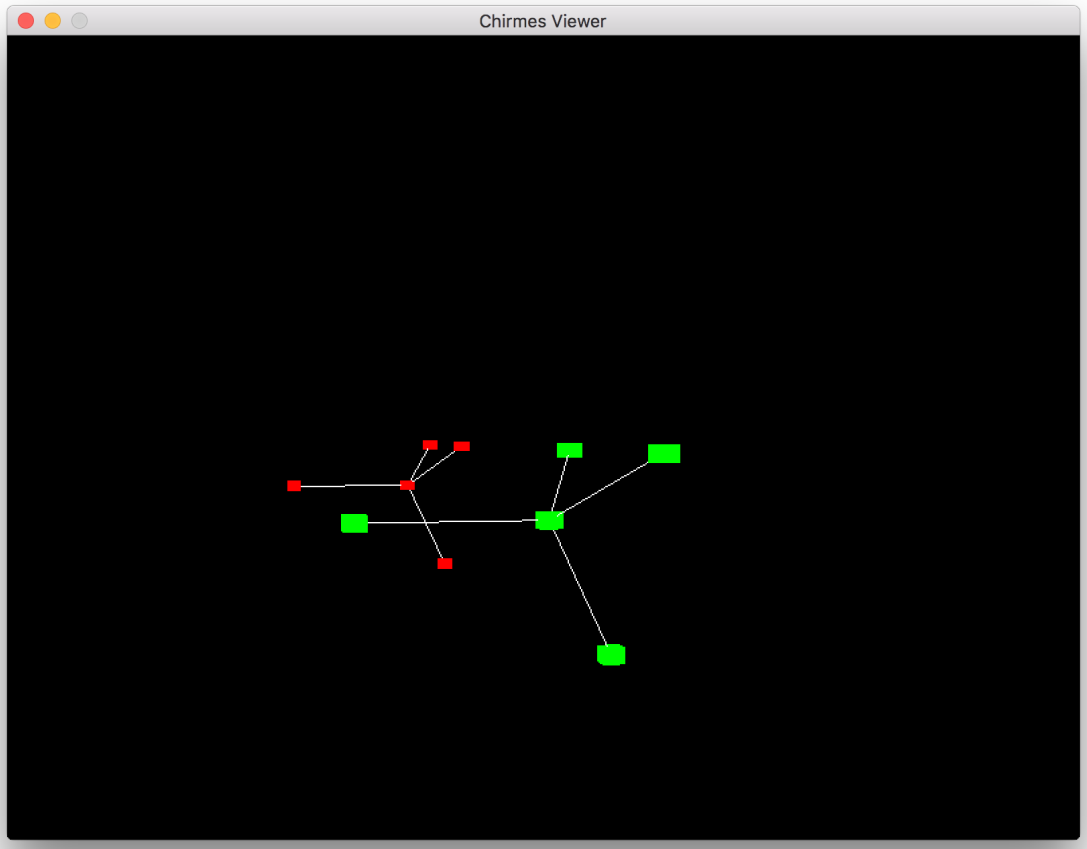


Rysunek 4.11. Algorytm liczący funkcję dopasowania - miara chiralności - wykonywany na CPU

4.3.4. Moduł wizualizacji

Moduł wizualizacji został stworzony jako funkcjonalność pomocnicza, stąd też jej możliwości w obecnej wersji są bardzo podstawowe. Wizualizator umożliwia wyświetlenie dowolnej ilości związków chemicznych opisanych klasą *StrippedMol* wraz z wiązaniami. Posiada także funkcjonalność manipulacji kamerą widoku. Kontrola nad widokiem oraz innymi funkcjami zapewniana jest poprzez klawiaturę. Na Rysunku 4.12 zaprezentowany został sposób działania modułu wizualizacji. W Dodatku D znajdują się domyślne ustawienia obsługiwanych klawiszy. Dodatkowo, przewidziano także możliwość przypisania dodatkowych funkcji pod inne klawisze z poziomu kodu aplikacji.

Moduł ten został zbudowany w oparciu o bibliotekę SDL, która wykorzystywana jest do tworzenia okna oraz obsługi pętli zdarzeń – zaimplementowane w klasie *Viewer*.



Rysunek 4.12. Moduł wizualizacji obrazujący metan wraz z jego lustrzanym odbiciem

Do generowania grafiki użyto biblioteki OpenGL w wersji 3.3. Jako model atomu użyto wygenerowanej wcześniej tablicy z pozycjami wierzchołków, zapisanej w postaci pliku nagłówkowego .h. Jest to jedyny model wykorzystany w aplikacji. Do wyrysowania łączeń między atomami użyto prymitywu wbudowanego w OpenGL – *GL_LINES*. Cząsteczka chemiczna reprezentowana jest przez klasę *RenderableMolecule*, która implementuje interfejs *Renderable*. Wszystkie obiekty ze sceny (obecnie są to tylko obiekty klasy *RenederableMolecule*) przekazywane są do obiektu klasy *Renderer*, która odpowiada za wyczyszczenie ekranu i wyświetlenie obiektów. Do operacji na macierzach wykorzystano również bibliotekę *Eigen*. Ponieważ od wersji 3.0 w OpenGL usunięto wszystkie metody do bezpośredniego umieszczania wierzchołków w buforze obrazu, zaimplementowane zostały także proste programy cieniujące, tj. *vertex shader* oraz *pixel shader*.

4.3.5. Moduł zapisywania wyników

Moduł zapisywania wyników działa w ostatniej fazie aplikacji, zapisując wynik wyliczonej miary chiralności do pliku wyjściowego w formacie *csv*. Dodatkowo, zapisywana jest także macierz transformacji użyta do przekształcenia lustrzanego odbicia do finalnej postaci (w porządku kolumnowym, rząd po rzędzie).

4.4. Implementacja CUDA

4.4.1. Rozważania projektowe

Z racji ograniczeń stawianych przez architekturę CUDA, należało zmodyfikować metodę obliczenia miary chiralności wraz z algorytmem genetycznym. Kod działający z wykorzystaniem technologii CUDA został zaimplementowany tylko na platformie Windows, w związku z czym należało odpowiednio przygotować kod odpowiedzialny za stworzenie instancji *ComputeEngine* realizującej obliczenia na karcie graficznej. Klasą, która bezpośrednio implementuje interfejs *IComputeEngine*, jest *IComputeEngine*. Jednocześnie realizując wzorzec „Fasady" zapewnia ona dostęp do obiektu właściwej klasy *CudaComputeEngine*, która wykonuje wszystkie operacje na GPU.

W celu sprawnego przeprowadzenia procesu implementacji na platformie CUDA, zdecydowano się na skorzystanie z biblioteki *Thrust*, która posiada dużą ilość wbudowanych algorytmów oraz struktur danych działających na GPU w sposób transparentny dla programisty. W aplikacji wykorzystana została ona do zarządzania alokacją pamięci oraz do automatycznego decydowania o tym, jak powinna zostać zorganizowana struktura, na której uruchamia się dany kernel (ile wątków, ile bloków, w jakiej hierarchii).

4.4.2. Struktury danych

W pamięci karty graficznej dane zostały zorganizowane w sposób inny niż w przypadku implementacji tradycyjnej. Podstawowa struktura *Genotype*, która w wersji CPU jest przykładem podejścia *AOS* (array of structures)[4], została przeorganizowana do postaci *SOA* (structure of arrays)[5], która z racji organizacji dostępu do pamięci

[4]Tablica struktur
[5]Struktura tablic

Listing 4.4. Struktura opisująca genotyp w pamięci karty graficznej

```
template <typename T>
struct GenotypesArray
{
    using TupleType = typename T::value_type;
    using GenotypeTuple = thrust::tuple<TupleType, TupleType, TupleType, TupleType, ↙
        ↳ TupleType, TupleType>;

    T x;
    T y;
    T z;
    T rotX;
    T rotY;
    T rotZ;
}
using DeviceGenotypesArray = GPU::GenotypesArray<thrust::device_vector<float>>;
using HostGenotypesArray = GPU::GenotypesArray<thrust::host_vector<float>>;
```

zapewnia znacznie lepszą wydajność. Zamiast przechowywania każdego parametru genotypu oddzielnie, na karcie graficznej zaalokowanych jest 6 oddzielnych tablic typu *float*. Dzięki zastosowaniu najnowszej wersji CUDA SDK możliwe było stworzenie szablonu opisującego taką strukturę, *GenotypesArray*. Dodatkowo, zadaklerowane zostały pomocnicze aliasy domyślnie używanych klas specjalizujących szablon (Listing 4.4). Poprawne skompilowanie biblioteki *Eigen* tak, aby działała na karcie graficznej, okazało się niemożliwe, stąd pojawiła się konieczność stworzenia struktury opisującej macierz transformacji *TransformationMatrix*. W celu zapewnienia wygodnego dostępu do jej elementów, w sposób możliwie podobny do wersji CPU, została ona przygotowana w oparciu o unię o następującej konstrukcji (Listing 4.5). Dodano też pomocnicze funkcje do mnożenia przez inną macierz oraz przez tablicę atomów. Strukturą do przechowywania informacji o pozycjach atomów w cząsteczce jest zwykła tablica typu *float4*[6].

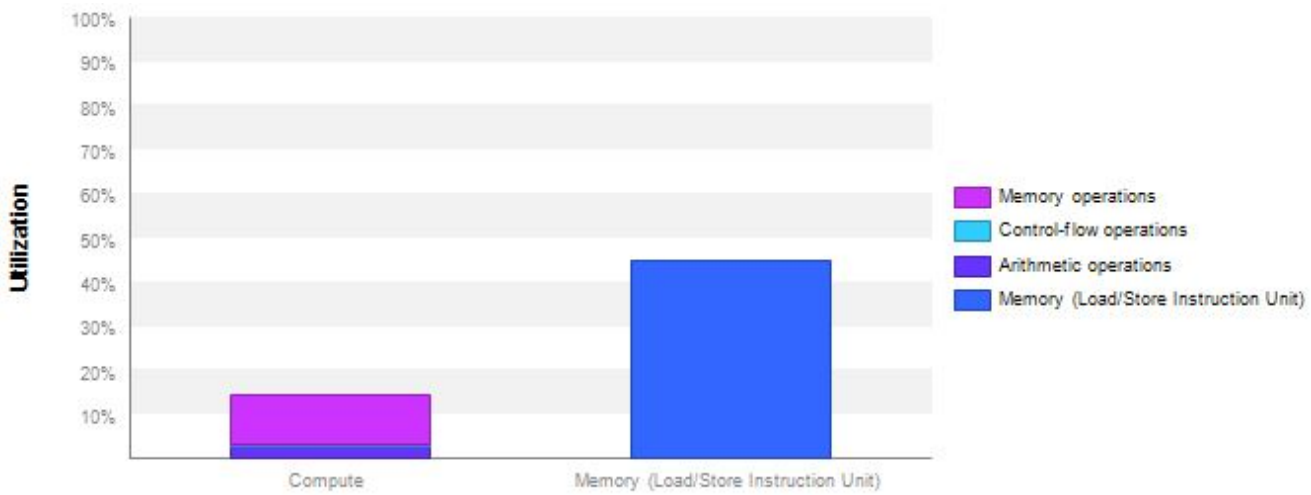
4.4.3. Algorytmy

W zdecydowanej większości algorytmy zastosowane w aplikacji działające na CUDA są identyczne jak te działające na CPU. Różnice głównie wynikają z odmiennych struktur

[6] *float4* – typ wbudowany w CUDA C\C++, zawiera cztery składowe: *x, y, z, w*

Listing 4.5. Struktura opisująca macierz transformacji w pamięci karty graficznej

```
union TransformationMatrix
{
    struct s { float4 col1, col2, col3, col4; } single;
    float tab[16];
    float4 tab4[4];
}
using DeviceTransformationMatrixVector = thrust::device_vector<TransformationMatrix>;
using HostTransformationMatrixVector = thrust::host_vector<TransformationMatrix>;
```



Rysunek 4.13. Analiza kodu wyliczającego wartość funkcji dopasowania działającego na GPU wykonana przez Nvidia Visual Profiler

danych oraz sposobu programowania na procesorze karty graficznej. Do głównych różnic należą:

- generatory losowe – wszystkie liczby losowe generowane są za pomocą wbudowanej w bibliotekę *thrust::random* klasę *thrust::uniform_real_distribution<float>* lub *thrust::uniform_int_distribution<int>* w zależności od wymaganego typu danych. Innym podejściem spotykanym w przypadku korzystania z liczb losowych na GPU jest ich uprzednie wygenerowanie „na zapas” dla całego czasu działania aplikacji – takie podejście nie zostało jednak wykorzystane z racji niewielkiego wzrostu wydajności względem potencjalnych komplikacji,

- generacja populacji losowej – implementowana przez funktor[7] *GenerateRandomPopulation*, zasada działania jest identyczna, główna różnica wynika ze specyfiki CUDA - każdy osobnik generowany jest przez oddzielny wątek, a zakresy generatorów losowych są obliczane wcześniej na CPU,

- obliczanie funkcji dopasowania – algorytm został podzielony na 3 części, za każdą odpowiada oddzielny funktor. Wszystkie z nich działają dla każdego osobnika osobno:

  – konwersja genotypów do macierzy transformacji – implementowana przez klasę *transformationMatrixMaker*. Dzięki zastosowaniu specjalnych iteratorów dostarczanych przez *thrust* do funktora przekazywana jest krotka [38] ze wszystkimi wartościami genotypu dla danego osobnika. Tworzy cztery macierze 4x4, tak jak we wzorze (4.2), odpowiednio mnożąc je przez siebie,

  – stworzenie tymczasowych osobników – implementowana przez klasę *individualMaker*. Odpowiada za wymnożenie macierzy transformacji przez macierz z pozycjami atomów,

  – wyliczenie wartości funkcji dopasowania – implementowana przez klasę *ComplexRandomChiralityCalculator*. Zasada działania jest identyczna jak w wersji działającej na CPU czyli każdy wątek najpierw wylicza macierz odległości, potem znajduje ciąg najmniejszych wartości jednocześnie usuwając rzędy/kolumny je zawierające, ostatecznie normalizując wyliczoną wartość. Po zbadaniu narzędziami profilującymi okazało się, że jest to najbardziej obciążający funktor spośród całego algorytmu zajmujący około 99% czasu wykonania kodu na karcie graficznej. Największym problemem obecnej implementacji są nadmierne operacje na pamięci, w tym dynamiczna jej alokacja i dealokacja. Na Rysunku 4.13 zaprezentowana została analiza wykonana przez *NVIDIA Visual Profiler* wskazująca na problemy związane z nadmierną ilością operacji na pamięci,

- operatory algorytmu genetycznego – implementowana przez klasę *ReplicatePopulation*. Identycznie jak na CPU najpierw następuje selekcja turniejowa, następnie w zależności od zadanego prawdopodobieństwa krzyżowanie oraz mutacja. Oprócz drobnych zmian wynikających z innej obsługi pamięci kod tych operacji jest identyczny jak w wersji CPU,

[7] funktor – w języku C++ klasa\struktura implementująca operator wywołania funkcji

- warunkowa regeneracja populacji – w wersji działającej na CUDA zdecydowano się na pominięcie tej fazy.

5. Weryfikacja eksperymentalna opracowanego oprogramowania

W celu weryfikacji i testowania opracowanego oprogramowania przeprowadzono następujące badania:

- test wartości dla cząsteczek achiralnych,
- test wartości dla cząsteczek chiralnych,
- test wydajności implementacji CPU w porównaniu z GPU,
- zastosowanie w praktycznych badaniach naukowych.

Eksperymenty te wykonano w konfiguracji opisanej w Dodatku B.

5.1. Test wartości dla cząsteczek achiralnych

Fundamentalnym warunkiem poprawności implementacji jest otrzymywanie dla cząsteczek achiralnych wartości zerowej miary chiralności. Cząsteczki takie są identyczne ze swoim lustrzanym odbicie, zatem w najlepszym dopasowaniu atomy związku wyjściowego i jego odbicia są doskonale nałożone, co powoduje zerowanie funkcji oceniającej.

W celu sprawdzenia, czy Chirmes spełnia ten warunek, przeliczono 4 achiralne cząsteczki chemiczne o różnej wielkości. Wyniki podane są w Tablicy 5.1.

Tablica 5.1. Wartości miar ^{SR}CM wyliczone dla cząsteczek achiralnych

lp.	związek	ilość atomów	wartość ^{SR}CM	czas obliczenia [ms]	wartość ^{SR}CM, CHIMEA	czas obliczenia, CHIMEA[s]
1	fulleren	60	0,0067	132884	0,0007	3480
2	tetrafenyl	44	0,0018	49360	0,0018	600
3	butan	14	0,0043	4479	0,0005	120
4	etan	8	0,0020	1819	0,0017	60

Jak widać, Chirmes znajduje wartości zbliżone do idealnego 0,0000. Odchylenie od wartości idealnej jest niewielkie. Z doświadczenia wiadomo, że przeciętnie związki chiralne mają wartości miary ^{SR}CM co najmniej rzędu 0,100 – 0,200. Błąd jest tym

bardziej zaniedbywalny, że wizualnie cząsteczka jest dokładnie nałożona na swoje lustrzane odbicie we wszystkich 4 przypadkach. Co więcej, osiągnięcie podobnej dokładności w programie CHIMEA wymaga ponad godziny, podczas gdy w przypadku Chirmesa całość zajęła około trzech minut.

5.2. Test wartości dla cząsteczek chiralnych

Dalej porównano wartości miar ${}^{SR}CM$. Policzono w tym celu programami CHIMEA i Chirmes wartości dla 11 związków chiralnych. Uzyskane rezultaty podano w Tablicy 5.2.

Tablica 5.2. Wartości dla zbioru testowego uzyskane w programach CHIMEA i Chirmes

Lp.	Testowany związek	CHIMEA	Chirmes	odchylenie w %
1	5-androstenediol	0,6777	0,7051	4
2	3β-androstanediol	0,6962	0,7756	11
3	5,6-didehydroizoandrosteron	0,7281	0,7944	9
4	5α-dihydrotestosteron	0,7512	0,7828	4
5	3α-androstanediol	0,7689	0,8139	6
6	androsteron	0,8154	0,8978	10
7	epitestosteron	0,8188	0,8372	2
8	testosteron	0,8435	0,8727	3
9	4-androstenedion	0,8464	0,9758	15
10	5β-dihydrotestosteron	0,848	1,0453	23
11	4-androstenediol	0,8568	0,8686	1

Procentowe odchylenie wartości obliczonych za pomocą programu Chirmes wynosi od 1 do 23 procent. Wyniki z CHIMEI i z Chirmesa są ze sobą skorelowane, a współczynnik korelacji wynosi $R = 0.86$. Zatem pomimo znaczącego błędu Chirmesa w przypadku niektórych cząsteczek, wynik z tego programu jest, z punktu widzenia zastosowań QSAR, tożsamy z wzorcową miarą ${}^{SR}CM$. W QSAR chodzi bowiem o dobre odwzorowanie wzajemnych zależności w całej serii związków, niekoniecznie zaś o bezwzględne liczby.

Jeszcze raz zwraca uwagę szybkość wykonania obliczeń. CHIMEA do przeliczenia zbioru 11 związków potrzebowała ponad 16 minut, podczas gdy Chirmes jedynie 1.83 minuty, czyli prawie dziesięciokrotnie mniej.

5.3. Test wydajności implementacji CPU w porównaniu z GPU

Aby porównać wydajność implementacji jednowątkowej wykonywanej na CPU oraz wielowątkowej działającej na GPU zbadano wydajność dla trzech różnych cząsteczek

z trzema różnymi konfiguracjami algorytmu genetycznego. Różnica pomiędzy poszczególnymi konfiguracjami dotyczy liczby osobników w populacji oraz liczby powtórzeń algorytmu genetycznego, dokładniejszy opis znajduje się w Dodatku C, w Tablicy 6.3. Ponieważ w tym teście badano tylko kwestie wydajności, ewentualne różnice w poprawności rozwiązania nie były brane pod uwagę. Poniższe testy przeprowadzono na komputerze nr. 2 z konfiguracji przedstawionych w Tablicy 4.1. Wyniki przedstawia Tablica 5.3.

Tablica 5.3. Porównanie czasu obliczeń dla implementacji CPU oraz CUDA

	Konfiguracja 1		Konfiguracja 2		Konfiguracja 3	
nazwa związku testowego	CPU[ms]	GPU[ms]	CPU[ms]	GPU[ms]	CPU[ms]	GPU[ms]
metan	64143	57488	6	146	3504	2104
butan	297110	699892	34	1650	6806	5961
11-deoxy-corticosterone-100	6724821	34163653	743	86025	378859	371893

Wyniki zebrane w powyższej tabeli potwierdzają analizę implementacji CUDA przeprowadzoną w rozdziale 4.4. Na wydajność implementacji CUDA największy wpływ ma ilość atomów w cząsteczce (przedstawione w tabeli 5.4) oraz wielkość populacji. Wzrost liczby atomów dramatycznie zwiększa czas obliczeń, nie tylko dla implementacji CUDA. Z racji architektury, dla zaimplementowanego algorytmu jest to szczególnie widoczne przy implementacji działającej na karcie graficznej (rozmiar dynamicznej alokacji pamięci globalnej, stosowanie zagnieżdżonych pętli w kodzie jednego wątku). Co więcej, brak wystarczającego obciążenia (mała wielkość populacji – konfiguracja 1 i 2) powoduje, że duża część zasobów karty graficznej pozostaje bezczynna – uruchomionych jest zbyt mało wątków. Dopiero dla konfiguracji nr 3 widać przewagę wykonywania wielowątkowego – obciążenie populacją o wielkości 5000 osobników (~5000 wątków) powoduje, że zysk z wykonywania równoległego jest większy niż spowolnienie związane z alokacjami pamięci i zagnieżdżonymi pętlami. Należy jednak podkreślić, że tak ekstremalne wartości jak w konfiguracji 1 i 3 nie są wymagane do tego aby znaleźć poprawne rozwiązanie.

W celu poprawy wydajności funkcja licząca dopasowanie pojedynczego osobnika powinna zostać rozdzielona na dodatkowe funkcje, które realizowałby pętle poprzez uruchamiane na wielu wątkach jednocześnie. Ponad to, należałoby skorzystać z prealokowanej pamięci zamiast dynamicznej alokacji (występujących w dużej ilości w funkcji obliczającej miarę chiralności). Dodatkowo, odpowiednia modyfikacja organizacji wątków w bloki

(obecnie realizowana automatycznie przez *Thrust*) pozwoliłaby na skorzystanie ze znacznie szybszej pamięci współdzielonej zamiast globalnej.

Tablica 5.4. Liczba atomów w badanych związkach

Nazwa	Liczba atomów
metan	5
butan	14
11-deoxycorticosterone-100	54

Przy porównywaniu wyników należy brać pod uwagę, iż posiadana konfiguracja sprzętowa nie jest adekwatna, jeżeli uwzględnimy cenę i datę produkcji zważywszy, że wykorzystywana karta graficzna, GeForce GTX 660 Ti, została wypuszczona na rynek w 2012 roku w cenie około 1000 zł – była więc przedstawicielem średniej klasy wydajnościowo-cenowej wśród kart graficznych, natomiast procesor Intel Core i7 6700K (rok produkcji 2016), jest obecnie najwydajniejszym procesorem dla komputerów domowych, kosztuje około 1600 zł.

Z tego względu, można uznać, że wyniki prezentowane przez implementację CUDA są dobrą bazą do wyżej wymienionych modyfikacji, po wykonaniu których, aplikacja uruchomiona na karcie graficznej obecnej generacji, pozwoli osiągnąć zdecydowany przyrost wydajności.

5.4. Zastosowanie aplikacji Chirmes w badaniach naukowych

Aby przekonać się o praktycznych zaletach utworzonej aplikacji, w Zakładzie Neuropeptydów Instytutu Medycyny Doświadczalnej i Klinicznej PAN przeprowadzono testowe weryfikacje zastosowania tej aplikacji w badaniach naukowych, które skrótowo opisano poniżej.

Wartości miar chiralości dla 11 steroidów przedstawione w podpunkcie 5.2 zostały użyte do budowy modelu QSAR opisującego powinowactwo tych cząsteczek do receptora androgenowego. Należy wyjaśnić, że receptor androgenowy jest białkiem, które łączy się z testosteronem i powoduje wytwarzanie oraz utrzymywanie męskich cech płciowych. Jest również odpowiedzialne za budowę kości i mięśni oraz utrzymywanie siły mięśniowej. Z punktu widzenia chemii leków receptor androgenowy jest ważnym celem dla leków wzmagających odbudowę mięśni w przypadku np. chorób wyniszczających, albo operacji.

Hipoteza, która została zweryfikowana dzięki otrzymanemu modelowi QSAR, mówi, że dla wiązania steroidów z receptorem androgenowym ważna jest obecność i charakter elementów znajdujących się na skraju cząsteczek oraz kształt całej molekuły.

Modelowanie QSAR przyniosło równanie opisujące zależność powinowactwa ($log(RBA)$) od charakteru elementów chemicznych na skraju cząsteczek (wyrażonego jako cząstkowe ładunki elektryczne $q3$ i $q17$) oraz ogólnego kształtu molekuły opisanego przez miarę chiralności ${}^{SR}CM$. Równanie to ma postać

$$log(RBA) = 4.1(\pm 3.1) + 9.2(\pm 2.5) * q3 - 7.0(\pm 2.3) * q17 - 3.3(\pm 1.2) * {}^{SR}CM, \quad r = 0.83, n = 11 \tag{5.1}$$

gdzie $log(RBA)$[1] – powinowactwo, wyrażone jako logarytm względnego powinowactwa; $q3$ i $q17$ – ładunki elektryczne atomów węgla c3 i c17.

Jak można zauważyć, równanie to – stanowiące wstępny model - ma akceptowalną wartość współczynnika korelacji danych eksperymentalnych i teoretycznych ($R = 0.83$). Stanowi zatem dobry punkt wyjściowy do pogłębionej analizy.

Jeszcze raz należy podkreślić fakt, że otrzymanie wartości ${}^{SR}CM$ w programie Chirmes zajęło czas ponad 10 krotnie mniejszy niż w CHIMEI. Będzie mieć to szczególne znaczenie w przypadku przeliczania większych zestawów związków. Tak więc opracowane oprogramowanie może znaleźć istotne znaczenie w prowadzonych obecnie badaniach naukowych w dziedzinie CADD.

[1] RBA – relative binding affinity

6. Zakończenie

6.1. Realizacja celów pracy

Jak wynika z przedstawionego powyżej materiału, postawione cele pracy zostały zrealizowane. Autor opracował nową metodę obliczania miar chiralności ${}^{SR}CM$, z powodzeniem stosując algorytm genetyczny. Po drugie, rozwiązanie to zostało zaimplementowane w postaci aplikacji Chirmes[1].

Przeprowadzone testy wykazały, że Chirmes daje wyniki poprawne, a ponadto wykonuje obliczenia znacznie szybciej niż dotychczas dostępny program CHIMEA. Ewentualne odchylenia znalezione w testach mieszczą się w rozsądnych granicach. Poza tym możliwe jest uzyskanie lepszych wyników przy modyfikacji parametrów algorytmu genetycznego.

Do dodatkowych zalet Chirmesa w porównaniu z CHIMEĄ należy również obsługiwanie 110 powszechnie używanych formatów plików chemicznych, a także możliwość uruchomienia aplikacji na różnych systemach operacyjnych.

Autor podjął również próbę analizy równoległych algorytmów genetycznych działających na platformie CUDA, co zaowocowało stworzeniem wersji aplikacji z akceleracją karty graficznej. Mimo swojej prostoty już teraz pozwala przyspieszyć obliczenia dla pewnej grupy zadań, co stanowi jest dobrą bazą do osiągnięcia przyspieszeń opisywanych w analizowanych pracach.

W pracy przedstawiono też skrótowo zastosowanie Chirmesa w badaniach naukowych.

Innowacyjność nimniejszej pracy polega na tym, że zaproponowano opracowanie komputerowej metody obliczania miar chiralności ${}^{SR}CM$, łącząc różne paradygmaty programowania (w tym genetyczny) z możliwościami obliczeń równoległych w technologii CUDA. Aby to zrealizować wymagane było zastosowanie bardzo zaawansowanych metod programowania.

Do najwiekszych osiągnięć pracy można zaliczyć sukces zastosowania algorytmów genetycznych w tym problemie oraz wynikające z tego wielokrotne przyspieszenie w porównaniu z dostępnym oprogramowaniem.

[1] Cała aplikacja składa się z 71 plików źródłowych zawierających ponad 8600 linii kodu.

Praca wnosi więc nowe spojrzenie oraz istotny wkład w zakresie komputerowo-wspomaganego projektowania leków.

6.2. Potencjalne modyfikacje

Mimo skutecznej realizacji założonych celów, autor zdaje sobie sprawę z możliwości przeprowadzenia potencjalnych ulepszeń, które jeszcze bardziej zwiększyłyby wydajność (a przez to użyteczność) programu. Wśród nich można wymienić:

- zastosowanie bardziej optymalnej struktury do przechowywania informacji o odległościach między atomami, gdyż jak wykazały testy operacje związane z pamięcią są najbardziej czasochłonne w obecnej implementacji algorytmu liczącego funkcję dopasowania,
- programowanie równoległe na CPU (pomocna będzie wykonana już analiza przeprowadzona na potrzeby implementacji CUDA),
- zastosowanie biblioteki MPI (dla zastosowania w klastrach obliczeniowych) [39],
- implementacja bardziej rozbudowanego środowiska graficznego w celu uzyskania interfejsu bardziej przyjaznego dla użytkownika.

W przypadku wersji działającej na platformie CUDA koniecznym byłaby taka modyfikacja funkcji liczącej wartość dopasowania, aby przeprowadzała mniej operacji z wykorzystaniem globalnej pamięci. Ponadto można spodziewać się, że duży wzrost wydajności może dać skorzystanie z nowej funkcjonalności nazwanej „Dynamic Parallelism” [40], która umożliwia uruchamianie dodatkowych wątków z poziomu kodu działającego na karcie graficznej, a tym samym lepsze wykorzystanie zasobów sprzętowych.

6.3. Dalsze zastosowania praktyczne

W tym miejscu warto jeszcze zwrócić uwagę na szereg projektów badawczych, do których zastosowany zostanie Chirmes:

- przeliczenie miar chiralności dla wszystkich ligandów w bazie PDB [41] – jest to ponad 40 000 struktur o różnej wielkości, przeprowadzenie tego badania było niewykonalne w rozsądnej skali czasowej za pomocą dotychczas dostępnego programu CHIMEA,

- próbne obliczenia miar chiralności dla struktur białkowych i DNA – cząsteczki te liczą od kilku do kilkudziesięciu tysięcy atomów, co również stawiało je poza zasięgiem obliczeń w CHIMEI,

- obliczenia miar chiralności dla struktur pochodzących z dynamiki molekularnej – czyli serii danych o zachowaniu molekuł w czasie – przeciętne dynamiki molekularne generują kilkaset tysięcy klatek (geometrii w danym momencie czasu).

Wszystkie powyższe projekty mają istotne znaczenie z punktu widzenia badań podstawowych rozwijających metodologie komputerowo-wspomaganego projektowania leków. Pokazuje to, że Chirmes stanowi dobrą odpowiedź na realne potrzeby naukowe. Poza tym należy na zakończenie wspomnieć, że biochemia stanowi ogromne pole dla różnych obliczeń komputerowych wspomaganych metodami sztucznej inteligencji i nowoczesnych rozwiązań sprzętowych, które są jeszcze słabo opracowane. Dlatego opracowane w tej pracy oprogramowanie otwiera nowe perspektywy na dalsze, intensywne zastosowanie komputerów w chemii.

Bibliografia

[1] H. Stevens. *Life Out of Sequence: A Data-Driven History of Bioinformatics*. University of Chicago Press: Chicago/London, 2013.

[2] A. Da Silva. Current topics in computer-aided drug design. *J. Pharm. Sci*, 3(97) : 1089–1098, 2008.

[3] T. Talele et al. Successful applications of computer aided drug discovery: Moving drugs from concept to the clinic.

[4] M. Jamróz. Chimea. `http://smmg.pl/software/chimea.html`, stan na 12.09.2016.

[5] World Wide Protein Data Bank. Atomic coordinate entry format version 3.3. `http://www.wwpdb.org/documentation/file-format-content/format33/v3.3.html`, stan na 12.09.2016.

[6] J. Vane, R. Botting. The mechanism of action of aspirin. *Thromb. Res.*, 5–6(110) : 255–258, 2003.

[7] W. Walters et al. Virtual screening—an overview. *Drug Discov. Today*, 4(3) : 160–178, 1998.

[8] C. Martin. *Quantitative Drug Design: A Critical Introduction, Second Edition*. CRC Press/Taylor & Francis, 2010.

[9] J. Devillers et al. *Topological Indices and Related Descriptors in QSAR and QSPAR*. CRC Press/Taylor & Francis, 2000.

[10] Talete srl. DRAGON for Windows (Software for Molecular Descriptor Calculations)., 2010.

[11] Accelrys Software Inc. Discovery studio modeling environment. 2014.

[12] M. Shahlaei. Descriptor selection methods in quantitative structure-activity relationship studies: A review study. *Chem. Rev.*, 10(113) : 8093–8103, 2013.

[13] P. Gramatica. Principles of qsar models validation: Internal and external. *QSAR Comb. Sci.*, 5(26) : 694–701, 2007.

[14] Blackwell Scientific Publication. Chirality. IUPAC. Compendium of Chemical Terminology., 1997.

[15] I. Reddy et al. *Chirality in Drug Design and Development.*

[16] T. Eriksson et al. Clinical pharmacology of thalidomide.

[17] M. Jamróz et al. On Stability, Chirality Measures, and Theoretical VCD Spectra of the Chiral C58X2 Fullerenes (X = N, B). *J. Phys. Chem. A*, 1(116) : 631–643, 2012.

[18] Jamróz, M. H.; Rode, J. E.; Ostrowski, S.; Lipiński, P. F. J.; Dobrowolski, J. C. Chirality measures of α-amino acids. *J. Chem. Inf. Model.*, 6(52) : 1462–1479, 2012.

[19] P. Lipiński, J. Dobrowolski. Local chirality measures in qspr: Ir and vcd spectroscopy. *RSC Adv.*, 87(3) : 47047–47055, 2014.

[20] S. Russell,P. Norvig, J. Canny. *Artificial intelligence : A modern approach (3rd ed.).* Pearson Education International., wydanie 3, 2010.

[21] Z. Michalewicz, Z. Nahorski. *Algorytmy genetyczne + struktury danych = programy ewolucyjne.* Wydawnictwa Naukowo-Techniczne, wydanie 3, 2003.

[22] M. Roderick. A generic parallel genetic algorithm., 2003. Master thesis submitted to The University of Dublin, `http://www.maths.tcd.ie/~rmurphy/Project/Report/report.html`, stan na 03.06.2016.

[23] A. Hassani, J. Treijs. An overview of standard and parallel genetic algorithms. Mälardalen's University, Sweden, 2009.

[24] Wikipedia. Flops — Wikipedia, the free encyclopedia, 2016. `https://en.wikipedia.org/wiki/FLOPS`, stan na 03.06.2016.

[25] NVIDIA. Cuda c programming guide, 2016. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz4AToAgq58`, stan na 03.06.2016.

[26] P. Pospichal et al. Parallel genetic algorithm on the cuda architecture. *EvoApplicatons*, 2010.

[27] J. Seo et al. Performance comparison of gpus with a genetic algorithm based on cuda. *Advanced Science and Technology Letters*, (65) : 36–44, 2014. `http://dx.doi.org/10.14257/astl.2014.65.09`, stan na 03.06.2016.

[28] J. Jaros, P. Pospichal. A fair comparison of modern cpus and gpus running the genetic algorithm under the knapsack benchmark. *EvoApplicatons*, 2012.

[29] M. Oiso et al. Implementing genetic algorithms to cuda environment using data parallelization. *Tehnički vjesnik*, 4(18) : 511–517, 2011.

[30] International Organization for Standarization. Recent milestones: C++17 nearly feature-complete, second round of tses now under development. `https://isocpp.org/std/status`, stan na 12.09.2016.

[31] cppreference.com. Lambda functions. `http://en.cppreference.com/w/cpp/language/lambda`, stan na 12.09.2016.

[32] B. Stroustrup. C++14 language extensions. `http://www.stroustrup.com/C++11FAQ.html`, stan na 12.09.2016.

[33] International Organization for Standarization. C++14 language extensions. `https://isocpp.org/wiki/faq/cpp14-language`, stan na 12.09.2016.

[34] cppreference.com. Namespaces. `http://en.cppreference.com/w/cpp/language/namespace`, stan na 12.09.2016.

[35] M. Wall. Galib – a c++ library of genetic algorithm components. `http://lancet.mit.edu/ga/`, stan na 12.09.2016.

[36] Wikipedia. Transformation matrix — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Transformation_matrix`, stan na 12.09.2016.

[37] cppreference.com. std::mersenne_twister_engine. `http://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine`, stan na 12.09.2016.

[38] cppreference.com. Tuple. http://en.cppreference.com/w/cpp/utility/tuple, stan na 12.09.2016.

[39] B. Barney, Lawrence Livermore National Laboratory. Message passing interface (mpi). https://computing.llnl.gov/tutorials/mpi/, stan na 12.09.2016.

[40] A. Adinetz, NVIDIA. Cuda dynamic parallelism api and principles. https://devblogs.nvidia.com/parallelforall/cuda-dynamic-parallelism-api-principles/, stan na 12.09.2016.

[41] Research Collaboratory for Structural Bioinformatics. Rcsb pdb: Rcsb protein data bank. http://www.rcsb.org/, stan na 12.09.2016.

Dodatki

A. Opis pliku konfiguracyjnego

Plik konfiguracyjny wykorzystywany jest do nadania początkowych ustawień aplikacji. Musi się znajdować w głównym katalogu aplikacji oraz być nazwany main_cfg.json Poniżej przedstawiony jest przykładowy plik wraz z opisem poszczególnych pól konfiguracyjnych.

Listing 6.1. Przykładowy plik konfiguracyjny – plik

```
{
    "mode": "CPU",
    "inputDirectoryPath": "molecules/",
    "visualizer": false,
    "input": [
            {
                "fileName": "metan.pdb",
                "debug": {
                    "enabled": true,
                    "debugPrintoutSize": 10,
                    "debugPrintoutFrequncy": 100,
                    "debug_ga": false,
                    "debug_ga_history": true
                },
                "randomSelectionRounds": 50,
                "populationSize": 100,
                "maxIterations": 500,
                "crossoverProbability": 0.6,
                "mutationProbability": 0.55,
                "rotationPartMutationProbability": 0.5,
                "rotationRangeMin": "-PI",
                "rotationRangeMax": "PI",
                "mutationRotationRangeMin": "-PI",
                "mutationRotationRangeMax": "PI",
                "tournamentSize": 4,
                "translationRangeExtension": 0.05,
                "translationRangeMultiplier": 1.05,
                "populationRegenerateTime": 100,
                "shouldKeepBestMemberOnRegenerate": true,
                "populationRegenerateMinimalImprove": 0.1,
                "populationRegenerateMaximalImprove": 0.1
            }
        ]
}
```

Opis pól konfiguracyjncyh:

- **mode** – określa tryb działania aplikacji. Dostępne wartości to:
  - **CPU** – algorytm działający na CPU,
  - **GPU_ON_WINDOWS** – algorytm działający na karcie graficznej pod kontrolą systemu Windows,
  - **GPU_ON_LINUX** – algorytm działający na karcie graficznej pod kontrolą systemu Linux, obecnie nie zaimplementowane,
- **inputDirectoryPath** – katalog w którym będą wejściowe molekuły,
- **visualizer** – wskazuje czy aplikacja ma uruchomić moduł wizualizacji nałożeń. Dostępne wartości to *true/false*,
- **input** – tablica struktur opisująca pliki wejściowe dla który ma zostać uruchomiony algorytm znajdowania najlepszego dopasowania, każda struktura ma identyczne pola konfiguracyjne, które są opisane poniżej:
  - **fileName** – nazwa pliku zawierającego cząsteczkę do zbadania,
  - **debug** – sekcja opisująca zachowania diagnostyczne algorytmu
  - **enabled** – flaga wskazująca czy diagnostyka algorytmu jest włączona. Dostępne wartości to *true/false*,
  - **debugPrintoutSize** – ile najlepszych osobników ma zostać wypisanych na konsolę,
  - **debugPrintoutFrequncy** – jak często ma występować wypis informacji diagnostycznych (co ile generacji),
  - **debug_ga** – flaga określająca czy funkcjonalność zbierania statystyk dotyczących działania algorytmu genetycznego ma być włączona. Dostępne wartości to *true/false*,
  - **debug_ga_history** – flaga określająca czy funkcjonalność zapisywania w pamięci wszystkich osobników na przestrzeni całej historii ma być włączona. Dostępne wartości to *true/false*,

- **randomSelectionRounds** – określa liczbę poszukiwań najmniejszej sumy w trakcie liczenia funkcji dopasowania,
- **populationSize** – określa rozmiar populacji pojedynczej iteracji algorytmu genetycznego,
- **maxIterations** – określa liczbę iteracji algorytmu genetycznego,
- **crossoverProbability** – określa prawdopodobieństwo wystąpienia krzyżowania podczas fazy replikacji,
- **mutationProbability** – określa prawdopodobieństwo wystąpienia mutacji podczas fazy replikacji,
- **rotationPartMutationProbability** – określa prawdopodobieństwo wystąpienia mutacji części genotypu dotyczącego rotacji podczas mutacji,
- **rotationRangeMin** – określa minimum zakresu spośród którego będą generowane wartości rotacji w trakcie generacji populacji. Należy podać kąt w radianach. Można także użyć specjalnej stałej PI/-PI,
- **rotationRangeMax** – określa maksimum zakresu spośród którego będą generowane wartości rotacji w trakcie generacji populacji. Należy podać kąt w radianach. Można także użyć specjalnej stałej PI/-PI.
- **mutationRotationRangeMin** – określa minimum zakresu spośród którego będą generowane wartości o jakie będzie się zmieniać rotacja w trakcie mutacji. Należy podać kąt w radianach. Można także użyć specjalnej stałej PI/-PI.
- **mutationRotationRangeMax** – określa maksimum zakresu spośród którego będą generowane wartości rotacji o jakie będzie się zmieniać rotacja w trakcie mutacji. Należy podać kąt w radianach. Można także użyć specjalnej stałej PI/-PI.
- **tournamentSize** – określa liczbę źawodników"podczas selekcji turniejowej. Parametr ten odpowiada za presję selekcyjną,
- **translationRangeExtension** – używana do powiększenia wyliczonego zakresu translacji o wartość tutaj zadaną,
- **translationRangeMultiplier** – używana do powiększenia wyliczonego zakresu translacji przemnożoną przez wartość tutaj zadaną,

- **populationRegenerateTime** – określa jak często ma następować regeneracja populacji w przypadku braku poprawy wyniku funkcji dopasowania,
- **shouldKeepBestMemberOnRegenerate** – flaga określająca czy najlepszy osobnik ma trafiać do nowo wygenerowanej populacji w trakcie,
- **populationRegenerateMinimalImprove** – określa warunki regeneracji,
- **populationRegenerateMaximalImprove** – określa warunki regeneracji.

B. Bazowa konfiguracja testowa

Testy w rozdziale 5 i 5.4 przeprowadzono na konfiguracji sprzętowej z tabeli 6.1.

Tablica 6.1. Konfiguracja sprzętowa

Typ	Laptop MSI GE 70
Procesor CPU	Intel Core i7-4700MQ 2.4 GHz
Pamięć ram	8 GB
Karta graficzna	NVIDIA GeForce GTX 765M 2GB

Jako ustawienia algorytmu genetycznego wybrani następujące parametry:

Tablica 6.2. Wybrane ustawienia algorytmu genetycznego

Liczba powtórzeń	500
Rozmiar populacji	10
Ilość powtórnych losowań	100
Rozmiar turnieju	4
Prawdopodobieństwo krzyżowania	60%
Prawdopodobieństwo mutacji	55%
Prawdopodobieństwo rotacji w mutacji	50%
Zakres rotacji	$-\pi$ do π

C. Konfiguracja testowa dla testu CPU vs. GPU

Test przeprowadzono z wykorzystaniem konfiguracji podanych w Tablicy 6.3:

Tablica 6.3. Konfiguracje testów wydajnościowych

	Konfiguracja		
Parametr numer	1	2	3
Liczba powtórzeń	1000	10	10
Rozmiar populacji	1000	10	5000
Ilość powtórnych losowań	50		
Rozmiar turnieju	4		
Prawdopodobieństwo krzyżowania	60%		
Prawdopodobieństwo mutacji	55%		
Prawdopodobieństwo rotacji w mutacji	50%		
Zakres rotacji	$-\pi$ do π		

D. Obsługa modułu wizualizacji

Domyślnie, moduł wizualizacji zapewnia następujące funkcje:

- klawisze W, S – obrót kamery przód/tył,
- klawisze A, D – przesuwanie pozycji kamery lewo/prawo,
- strzałki lewa, prawa – obrót kamery lewo/prawo,
- klawisze spacja, C – przesuwanie pozycji kamery góra/dół,
- klawisz Q – przejście do następnej cząsteczki (w przypadku uruchomienia wsadowego),
- klawisz E – zamknięcie aplikacji.

E. Zawartość płyty CD

Dołączona płyta CD zawiera następujące elementy:

- implementację opisanych rozwiązań (sekcja 4) – katalog *Source*, w tym:
  - plik *Chrimes.xcodeproj* zawierający plik projektowy dla programu Xcode,

  - plik *ChirmesWindows.sln* w katalogu *ChirmesWindows* zawierający plik projektowy dla programu Microsoft Visual Studio,
  - katalog *Chirmes* z kodem źródłowym wspólnym dla wszystkich platform,
  - katalog *ChirmesWindows* w którym znajdują się pliki wymagane do kompilacji na platformie Windows,
  - katalog *ChirmesGPU* zawierający kod źródłowy implementacji CUDA,

- elektroniczną wersję pracy dyplomowej – plik *Szurmak_Mgr.pdf* ,
- zestaw przykładowych plików wejściowych (związków chemicznych) – katalog *Molecules*,
- skompilowaną wersję aplikacji dla systemu Windows, wraz ze skompilowanymi dołączanymi dynamicznie bibliotekami *Open Babel* – katalog *BuildWindows*,
- skompilowaną wersję aplikacji dla systemu OS X, wraz ze skompilowanymi dołączanymi dynamicznie bibliotekami *Open Babel* – katalog *BuildOSX*.