Masarykova univerzita
Fakulta informatiky

Bakalářská práce

Porovnávání molekul proteinů za pomoci výpočtů technologie CUDA grafických karet nVidia

Tomáš Došek

2011

Masarykova univerzita
Fakulta informatiky

Zadání Bakalářské práce

Datum: 19.4.2011

Student: Tomáš Došek

Program: FI B-AP Aplikovaná informatika

Obor: FI BcAP Aplikovaná informatika

Vedoucí práce: Ing. Matej Lexa, Ph.D.

Název práce: Porovnání molekul proteinů za pomoci výpočtů technologie CUDA grafických karet nVidia

Zadání: V bioinformatice je často potřeba srovnávat struktury molekul mezi sebou.
Toto je výpočetně náročná operace, kterou by bylo možné urychlit rozložením výpočtu mezi CPU a GPU.
Nastudujte problematiku porovnávání struktur proteinů, prostředí CUDA pro programování GPU
a implementujte vybraný algoritmus pro srovnání proteinů tak,
aby využíval GPU. Korektnost implementace ověřte na vhodné datové sadě. Úspěšnost posuďte
na základě vyhodnocení dosaženého zrychlení ve srovnání s verzí využívající výhradně CPU.

Základní literatura: Blelloc, Guy E.: Vector Models for Data-Parallel Computing, Massachusetts, The MIT Press, 1990.
Chuong, B. Do; Kazutaka, Katoh: Protein Multiple Sequence Alignment, Stanford, Stanford Press, 2008.

Souhlas se zadáním (podpis, datum)

Tomáš Došek, 19.4.2011
student(ka)

vedoucí
bakalářské práce

vedoucí
odpovědného pracoviště

*KTP
KPGD
KPSK
KIT
*Hodící se označte!

Prohlášení

Prohlašuji, že tato práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, kterou jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

_____

Jméno a příjmení, podpis

Poděkování

Děkuji vedoucímu bakalářské práce Ing. Mateji Lexovi, Ph.D. za cenné rady, připomínky a metodické vedení práce. Děkuji také Bc. Tomáši Janouškovi za praktické rady k implementaci zvoleného algoritmu pro výpočet na grafických kartách.

Abstrakt

V této práci se zabývám metodami porovnání proteinových struktur a základním principům paralelizace výpočetních úloh, které mohou být k porovnání takovýchto struktur použity. Dále se v této práci věnuji návrhu algoritmu vhodného pro aplikaci jak na klasickém procesoru, užívaném například ve stolních počítačích, tak i pro aplikaci na výpočetní prostředí moderních grafických karet od výrobce NVIDIA. Součástí této práce je také vyhotovení programu v jazycích Java a C++ CUDA, který je na přiloženém CD nosiči.

Abstract

In this thesis I describe methods of protein structure alignment and basic principles of parallelization of computing taks that can be used for calculation of protein structure alignments.
I also propose an algorithm that can be easily implemented both on classic control processing unit used, for instance, in ordinary desktop computers, but also in the computing environment of modern graphic cards made by manufacturer NVIDIA. As a part of this thesis I also developed a computer programme written in languages Java and C++ CUDA, included on the attached CD medium.

Klíčová slova

protein, zarovnání proteinů, CUDA výpočet, semiglobální zarovnání, globální zarovnání, paralelizace výpočtů, PDB

Keywords

protein, protein alignment, CUDA computing, semiglobal alignment, global alignmnet, parallel computing, PDB

Obsah

1. Úvod

Obor bioinformatiky se v poslední době těší nemalé oblibě, a to jak ze strany médií, tak ze strany široké veřejnosti a zájemců o studium tohoto zaměření. Jeho prvopočátky sahají do dob, kdy potřeba pokročilého zpracování dat nasbíraných chemiky a biology přerostla rámec možností ručních výpočtů a vyhodnocování. K zvýšení podstatnou měrou přispěly projekty jako sekvenování genomů živých organismů a vytvoření a správa databází známých organických molekul a proteinů. Po mnoha vědeckých výzkumech však vyvstala snaha nejenom získávat sekvence a struktury různých molekul, ale tyto výsledky i navzájem porovnávat.

Typickým příkladem jsou odvětví genomiky a proteomiky, která na bázi srovnání exprese genů, proteinů či molekul určitých látek, pomáhají hledat nová léčiva na těžké nemoci [10]. Veškeré tyto informace jsou doposud poměrně zdlouhavě a za pomoci mnohdy rozsáhlých počítačových sítí zpracovávány. K porovnávání rozsáhlých molekul se totiž používají standardní procesory, jaké má nejspíš i každý z nás ve svém stolním počítači. Další možnou variantou, jak takto náročné výpočty provádět, jsou specializované počítače (např. IBM BLUEGENE), které obsahují, od běžně dostupných procesorů, odlišné varianty výpočetních prostředků specializovaných na konkrétní výpočetní operace. Bohužel varianta specializovaného zařízení je také, jak už to bývá, velice nákladnou variantou.

Společnost NVIDIA, výrobce grafických karet, čipových sad a jiných počítačových součástí, vývojem herních komponent vynalezla technologie, které nyní sice slouží náruživým hráčům počítačových her, ale v současné době začínají být čím dál, tím více využity i k složitým výpočtům v trojrozměrném prostoru. Jednou z těchto technologií je NVIDIA Compute Unified Device Architecture (dále pouze CUDA) [2]. CUDA je technologie, která moderním grafickým kartám umožňuje provádět složité výpočty v trojrozměrném prostoru během několika málo milisekund. Rychlost získání výsledku je oproti normálnímu procesoru několikanásobná [3]. Tohoto faktu by bylo možné využít například pro porovnání dvou složitých molekul, protože algoritmy, které takovéto srovnání provádí, využívají buďto terciární struktury molekul, nebo trojrozměrné grafy, do kterých jsou zaneseny různé informace, jako vazebné úhly mezi atomy v molekule.

1.1. Členění práce

Úvodní kapitola stručně představuje bakalářskou práci.

Druhá kapitola seznamuje se základními typy struktur používaných v praxi a teoretickými základy porovnání proteinových molekul. Dále představuje algoritmy k porovnání molekul používané.

Třetí kapitola popisuje základní vlastnosti technologie CUDA, možnosti jejího užití ve vědecké praxi a popis již navrhnutých projektů a použitá řešení na bázi této technologie.

Čtvrtá kapitola obsahuje návrh a implementaci algoritmu pro porovnání molekul na základě metody srovnání grafů ve třech dimenzích prostoru. Obsahuje také modelové příklady, na kterých ukazuji funkčnost veškerých užitých programových částí. Jako základní prvek pro porovnání dvou molekul používám vazební úhly mezi uhlíky na hlavním řetězci.

V páté kapitole popisuji testování korektnosti navrženého algoritmu vzhledem ke vzorovým sekvencím z databáze CATH. Dále zde srovnávám rychlosti dosažení výsledku při použití klasického procesoru, obsaženého ve stolním počítači - a to jak varianty využívající pouze jednoho vlákna, tak více vláken - s rychlostí výpočtu na grafické kartě NVIDIA GTS450, jejíž hlavní předností je 192 procesorů zpracovávajících mimo jiné také instrukce technologie CUDA. Snažím se zde také vysvětlit, co je příčinou rozdílů ve výsledcích. Na konkrétních příkladech ukazuji přednosti výpočtů na bázi technologie CUDA. Porovnávám zde i způsoby, kterými by konkrétní úlohy byly vypočítány na CPU a na procesoru grafické karty.

Závěrečná kapitola obsahuje stručné shrnutí dosažených výsledků.

Po závěrečné kapitole následuje seznam zdrojů a citací použitých v této práci a také seznam použitých zkratek a výrazů.

Přílohy obsahují návod ke spuštění a obsluze programu přiloženého k této práci.

2. Porovnání proteinových molekul

V této kapitole popisuji základní přehled stavby molekul proteinů, metody zarovnání jejich sekvencí a struktur. U každého typu porovnání proteinů vždy následuje několik příkladů již zavedených a v praxi běžně využívaných metod.

2.1. Hierarchie různých typů struktur v proteinech

Informace o proteinech lze zobrazit v dnešní době několika způsoby, z nichž každý má určité opodstatnění v praxi. Z hlediska vyobrazení a zápisu molekul proteinů rozlišujeme čtyři typy struktur a tomu odpovídajících dat.

2.1.1. Primární struktura

Tato struktura je dána pořadím aminokyselin v polypeptidovém řetězci. Byla stanovena v roce 1953 Frederickem Sangerem, kdy tento vědec objevil a prokázal jedinečnou kovalentní strukturu bílkovin.
Jedná se o zápis aminokyselin v pořadí od N-konce po C-konec bílkoviny [4].

2.1.2. Sekundární struktura

Sekundární struktura je uspořádání bílkovinného řetězce v prostoru, které zakládá na vlastnosti „krátké vzdálenosti“ mezi aminokyselinami navzájem sousedními v řetězci bílkoviny. Poprvé byla použita v třicátých až čtyřicátých letech dvacátého století a přinesla několik nových pojmů, jako α-helix, β-sheet, β-hairpin a random-coil a jiné další pojmy, které vesměs popisují strukturu bílkoviny.
α-helix je pojem popisující strukturu bílkoviny na základě dominantního stavebního prvku, tzv. α-šroubovice. Kdy tato pravotočivá šroubovice tvoří kostru bílkoviny.
β-sheet je naopak struktura tzv. skládaného listu. Tato struktura se vyznačuje ne moc vysokou prostorovou kompaktností, jelikož veškeré řetězce v bílkovině jsou při této struktuře uspořádány rovnoběžně v jedné rovině. Zbylé struktury dále v této práci nepopisuji, neboť z hlediska zarovnání struktur nejsou považovány za dominantní [4].

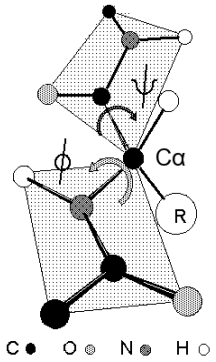
2.1.3. Terciární struktura

Rozdíl mezi terciární a sekundární strukturou proteinových molekul hlavně ve faktu, že v terciární struktuře uvažujeme vzájemnou polohu aminokyselin na větší vzdálenost. V bioinformatice tedy terciární strukturou míníme řetězec bílkoviny jako celek, zatímco sekundární struktura popisuje pouze jeho části [4]. S terciární strukturou pracuje i program, který jsem vytvořil pro účely této bakalářské práce.

Velice důležitým prvkem, z hlediska zpracování mnou navrženými algoritmy, je tzv. C-α uhlík, resp. jejich seskupení v kostře molekuly proteinu. C-α uhlíky totiž určují něco jako "páteř" proteinu (převzato z anglického termínu protein backbone), na kterou jsou připojeny postranní řetězce, které dohromady s touto "páteří" tvoří molekulu proteinu.

Molekula proteinu může být, do jisté míry, dobře popsána ať už pomocí polohy C-α uhlíků, tak i pomocí tzv. torzních úhlů mezi atomy v molekule proteinu. Torzní úhly jsou úhly v prostoru, určující polohu atomů, přičemž poloha každého následujícího atomu je v prostoru popsatelná od předchozího atomu pomocí dvou úhlů - torzních úhlů - ϕ a ψ [čteno *fí* a *psí*]. Soustavu těchto úhlů lze vidět na obrázku 1.

Algoritmy, které jsem použil v přiloženém programu využívají znalosti vzájemné polohy C-α uhlíků v proteinové molekuly. Pomocí jednoduché vektorové matematiky pak převádím tyto souřadnice v prostoru na jakési improvizované úhly ϕ a ψ, které však ve skutečnosti nemají s torzními úhly téměř nic společného. Bližší informace o způsobu využití znalosti souřadnic C-α uhlíků v prostoru pro určení polohy atomů jsou uvedeny v kapitole 4.



Obrázek 1: Torzní úhly v molekule proteinu

2.1.4. Kvartérní struktura

Tato struktura řeší uspořádání jednotlivých „podjednotek“ do tzv. proteinových aglomerátů, které jako celek tvoří jednu bílkovinu. Takovéto uspořádání však mají pouze hodně složité bílkoviny, jako např. fibrily kolagenu, nebo lidské DNA polymerázy [4].

2.2. Porovnání struktur v prostoru

Poctivému čtenáři předešlých podkapitol neuniklo, že výše uvedené struktury jsou všechny, kromě primární, trojrozměrné objekty. V dnešní době existují různé metody porovnání struktur v trojrozměrném prostoru. Většina z těchto metod se zabývá problematikou minimalizace časových nároků na výpočet tohoto poměrně výpočetně složitého procesu.V jednoduchém přehledu v této práci zachycuji několik metod, které se v posledních letech diskutují nejvíce v odborných článcích a které jsou pokládány za, každá svým způsobem, přelomové v oboru porovnání struktur.

2.2.1. Algoritmus porovnání struktury proteinů na základě kombinatorického rozšíření optimální cesty

Tato metoda se poprvé objevila v hledáčku vědců z oboru bioinformatiky a výpočetní chemie přibližně v roce 1996, kdy byla publikována autory Alexandrovem a Fischerem. Na rozdíl od jiných totiž dokázala bez pomoci algoritmů dynamického programování vyřešit problém podobnosti dvou proteinů v poměrně krátkém čase na základě převedení vazebných vlastností v proteinech do jisté míry značně striktních kombinatorických pravidel pro výpočet podobnosti na základě matic podobnosti. Základním pojmem této metody je takzvaná nejdelší souvislá cesta v proteinu.

Zde tento algoritmus nejdříve vypočte nejdelší souvislé zřetězení uhlíků v proteinu, dále pak v různých informatických databázích vyhledá pro optimalizaci algoritmu pouze proteiny, u nichž délka hlavního řetězce odpovídá právě délce řetězce daného proteinu. Dále pak postupuje tato metoda tak, že na základě vzdáleností mezi podjednotkami proteinů, porovná tyto výsledky a nehodící se proteiny vyřadí. Pomoci postupné aplikace kombinatorických pravidel pak zmenšuje počet atomů v podjednotkách a vyřazuje tak méně vhodné proteiny pro srovnání. Výsledkem je zarovnání proteinu strukturně nejvhodnějšího „příbuzného“ [9]. Pro složitost tohoto algoritmu nebudu jeho postup podrobněji popisovat.

2.2.2. Algoritmus DALI

Algoritmus DALI je jedním z předchůdců výše zmíněného algoritmu pro porovnání struktury proteinů na základě kombinatorického rozšíření optimální cesty. Byl uveden v roce 1992, nicméně vědeckou veřejností byl za poměrně přesný a vhodný jako řešení dané problematiky až v roce 1993. Od té doby ovšem ale všechny nové, přelomové metody porovnávají své výsledky s touto, neboť jak rozsáhlé testy této metody a její aplikace v praxi naznačují, jedná se o velice přesnou metodu. Bohužel na rozdíl od předešlé metody, tato vyžaduje aplikaci dynamických programovacích algoritmů. Pokud bychom chtěli blíže specifikovat, které algoritmy dynamického programování používá tato metoda, pak bychom měli mluvit hlavně o tzv. Monte Carlo proceduře, která zde hraje roli optimalizace výpočtu bodování podobnosti proteinů s ohledem na intramolekulární vzdálenosti [2]. Opět jako u předchozího případu nebudu pro složitost více popisovat tento algoritmus do hloubky.

2.3. Porovnání sekvencí

Porovnání sekvencí obvykle nazýváme také jako zarovnání sekvencí, při čemž v odborných kruzích se spíše hovoří o zarovnání sekvencí – dle doslovného překladu z anglického termínu "alignment" – je poměrně složitým tématem moderní bioinformatiky, neboť se zabývá způsoby, jak v ideálním případě dokonale porovnat sekvence, či fragmenty sekvencí, s již známými sekvenčními řetězci a tímto způsobem zjistit alespoň přibližnou strukturu experimentálně získaných látek (DNA, proteinů, genů, enzymů, atd.).

Při porovnávání sekvencí můžeme narazit na několik problémů. Typickým problémem je např. nízký stupeň homologie a tím podobnosti vzorku s jakýmkoli jiným, již známým vzorkem.

Na základě různých experimentů byly postupem času vyvinuty techniky, které se určitým způsobem snaží problémům porovnávání sekvencí, buď předcházet, nebo je brát v potaz a „bodovat“ jejich výskyt ve vzorku.

Porovnání sekvencí tedy je obvykle prováděno metodou tzv. porovnání na základě tzv. homologie proteinových molekul, které určuje celkovou podobnost srovnávaných proteinů.Tato metoda pracuje na základě předpokladu, který zjednodušeně řečeno říká, že pokud má jedna sekvence určitou pravděpodobnost podobnosti, která je vypočtena např. zarovnáním „písmen“ sekvencí z jednoho zdroje na srovnávaný vzorek, pak tento vzorek pravděpodobně bude vypadat velice podobně i co se týče terciární struktury. Výsledný model tedy bude velice podobný modelu terciární struktury, se kterou porovnáváme.

Existují dva základní typy porovnání sekvencí a jeden typ kombinovaný. Prvním typem je tzv. lokální porovnání sekvencí, tj. porovnání sekvencí hledající ve srovnávaných proteinech nejpodobnější úseky libovolných délek takové, že oblasti od těchto úseků vzdálené zanedbáváme [4]. Dalším typem pak je tzv. globální porovnání sekvencí. Globální porovnání sekvencí je takové, které bere v potaz celkovou strukturu srovnávaných molekul [6]. Jistou kombinací obou předchozích je pak tzv. semiglobální zarovnání, které se snaží najít nejlepší zarovnání, které obsahuje ale i začátek a konec porovnávané struktury [4]. Modifikaci poslední z těchto metod používám v algoritmu, který popisuji ve čtvrté kapitole.

2.3.1. Algoritmus Needleman-Wünsch

Algoritmus Needleman-Wünsch patří do rodiny algoritmů, které vypočítávají globální zarovnání dvou sekvencí. Stejně jako algoritmus Smith-Watterman byl původně vynalezen na srovnání prostých textových sekvencí. Vědecké obci byl představen poprvé v roce 1970 Saulem B. Needlemanem a Christianem D. Wünschem [4].

Postup tohoto algoritmu je následující:

1) Alokuj matici M o rozměrech (n+1)*(m+1), kde m a n jsou délky zarovnávaných sekvencí
2) V prvním řádku a prvním sloupci matice M nastav hodnoty na nula.
3) Pro prvek M[i, j] v matici, kde i leží v intervalu 1, n+1 a j leží v intervalu 1, m+1 včetně, porovnej prvky - sekvence 1 na pozici i-1 a sekvence 2 na pozici j-1 - nastav hodnotu na maximum z M[i, j-1]+s, M[i-1, j]+s, M[i-1, j-1]+s, kde s je tzv. hodnotící funkce. Hodnotící funkce může vypadat například následovně, pokud maximum pochází z pole M[i-1, j-1], pak pokud se srovnávané prvky shodují, pak s = 2, jinak s = -1, pokud maximum pochází z polí M[i-1, j], nebo M[i, j-1], pak při rovnosti prvků vstupních sekvencí s = 0 (takzavaná sankce za vložení mezery), jinak s = -1.
   Nyní jsme ve stavu, kdy jsou všechna pole matice M
4) Z pravého dolního rohu matice, tedy prvku M[n+1, m+1] začneme vyhledávání optimálního zarovnání sekvence tak, že i nastav na pozici n+1, j nastav na pozici m+1. Dokud hodnota i a j není rovna 1, do i a j ulož pozici maxima z prvků M[i-1, j], M[i, j-1], M[i-1, j-1].

2.3.2. Algoritmus Smith-Watterman

Tento algoritmus je typickým zástupcem skupiny algoritmů založených na lokálním porovnávání struktur. Za svůj základ bere algoritmus Needleman-Wünsch, je popsán výše. Tento algoritmus byl vynalezen v roce 1981 pány Temple F. Smithem a Michaelem S. Watermanem a původně měl, stejně jako algoritmus Needleman-Wünsch sloužit k porovnání textových sekvencí [4]. V níže uvedeném postupu algoritmu si všimněme, že dle kroku 3 nepovolujeme v matici M záporné hodnoty, což je jeden ze základních rozdílů oproti algoritmu Needleman-Wünsch. Dalším podstatným rozdílem pak je práce s maximem, neboť algoritmus Smith-Watterman si po dobu výpočtu pamatuje maximální prvek v matici M a jeho souřednice v této matici.

Postup výpočtu algoritmem Smith-Watterman je následující:

1) Alokuj matici M o rozměrech (n+1)*(m+1), kde m a n jsou délky zarovnávaných sekvencí
2) V prvním řádku a prvním sloupci matice M nastav hodnoty na nula, hodnotu maxima nastav na nula, pozici x maxima nastav na nula, pozici y maxima nastav na nula.
3) Pro prvek M[i, j] v matici, kde i leží v intervalu 1, n+1 a j leží v intervalu 1, m+1 včetně, porovnej prvky sekvence 1 na pozici i-1 a sekvence 2 na pozici j-1 nastav hodnotu M[i, j] na maximální hodnotu z nula, M[i, j-1]+s, M[i-1, j]+s, M[i-1, j-1]+s, kde s je tzv. hodnotící funkce. Hodnotící funkce může vypadat například následovně, pokud maximální hodnota pochází z pole M[i-1, j-1], pak pokud se srovnávané prvky shodují, pak s = 2, jinak s = -1, pokud maximum pochází z polí M[i-1, j], nebo M[i, j-1], pak při rovnosti prvků vstupních sekvencí s = 0 (takzavaná sankce za vložení mezery), jinak s = -1.
   Nakonec porovnej maximum s nově vzniklým prvkem, pokud je nový prvek vyšší nastav maximum na nový prvek a pozici x maxima nastav na i, pozici y maxima pak na j.
   Nyní jsme ve stavu, kdy známe maximum z matice a máme vypočítané veškeré její prvky.
4) Ze známé pozice maxima vyhledej ideální zarovnání následovně:
   i nastav na pozici maxima x, j nastav na pozici maxima y. Dokud prvek M[i, j] není roven nule, nebo i, nebo j není rovno 1, do i a j ulož pozici maxima z prvků M[i-1, j], M[i, j-1], M[i-1, j-1].

2.3.3. Algoritmus pro semiglobální zarovnání

Jak jsem již uvedl výše ve všeobecném přehledu, jedná se o kombinaci dvou předchozích algoritmů.
Postup výpočtu matice M je shodný s jejím vyplněním v algoritmu Needleman-Wünsch, nicméně nemáme pevně stanoveno, kterým prvkem začíná výpočet optimálního zarovnání. Tento prvek nejprve musíme nalézt v posledním sloupci a posledním řádku. Od tohoto vypočteného maxima pak začíná výpočet optimálního zarovnání následovně:

Z pozice maximálního prvku, tedy prvku M[n+1, m+1] začneme vyhledávání optimálního zarovnání sekvence tak, že i nastav na pozici maxima x, j nastav na pozici maxima y. Dokud hodnota i, nebo j není rovna 1, do i a j ulož pozici maxima z prvků
M[i-1, j],M[i, j-1], M[i-1, j-1].

3. Technologie NVIDIA CUDA

Následující kapitola se věnuje technologii NVIDIA CUDA (= Compute Unified Device Architecture). Popisuji zde jak technologii CUDA ve všeobecném kontextu, tak jeden z jejich přenosů na jiný programovací jazyk, než je původní C/C++, a to na jazyk Java.

3.1. NVIDIA CUDA C/C++

Technologie NVIDIA CUDA byla původně navržena pro jazyky C/C++ a Fortran. Jejím primárním cílem bylo zaujmout svým způsobem dominantní postavení na trhu grafických karet, neboť měla vývojářům počítačových her nabídnout prostředky, jak výpočetně složité operace provádět na grafické kartě, bez enormního vytížení standardního procesoru typu CPU. Od jejího uvedení netrvalo dlouho a největší světová studia vyvíjející počítačové hry začala používat tuto technologii např. pro výpočet náročné fyziky ve svých počítačových hrách. V rozvíjejícím se herním průmyslu tento okamžik znamenal obrovský zlom, neboť tato technologie umožnila za využití současných běžných přostředků zpracovávat mnohem náročnější operace, jako je například tříštění objektů, dokonalá simulace různých druhů povrchů od dokonale vypracované tkaniny až po tekoucí vodu [7].

Po začátcích v herním průmyslu se ale ukázalo, že by tuto technologii bylo možné využít nejen k vylepšení dojmů z "dokonalého" světa moderních her, a tak se začaly objevovat její první aplikace do vědeckého prostředí. Jedním z prvních úspěšně aplikovaných pokusů o využití této technologie se pak stal program Matlab se svou nadstavbou Simulink, který umožňoval užití paralelizace úloh pro výpočty, ať už matematického, tak i fyzikálního rázu [5].

3.1.1. Co je CUDA?

NVIDIA CUDA je technologie, jejíž hlavním účelem je zprostředkovat schopnost paralelních výpočtů na GPU jednotkách grafických karet NVIDIA (od série GeForce 8XXX výše) a tímto umožnit řešení rozsáhlých, komplexních výpočetních problémů ve zlomku času, který by tyto operace trvaly, kdybychom použili standardní procesor typu CPU. Aby bylo možné využít předností této technologie, poskytuje společnost NVIDIA tzv. “CUDA engine“, který obsahuje několik částí.

První části CUDA pracovního rozhraní je tzv. CUDA architektura instrukční sady (anglickou zkratkou CUDA ISA) a paralelní výpočetní rozhraní grafického procesoru.

K programování na architektuře CUDA dnes mohou vývojáři využít mnohých programovacích jazyků, z nich nejrozšířenější je však užití jazyka C/C++. V dnešní době se již pracuje na knihovnách této architektury přibližujících tuto moderní cestu řešení složitých výpočtů pro další jazyky, jako je Fortran, OpenCL, Python a Java. Na obrázku níže je ilustrace rozdílu mezi architekturou klasického procesoru (do tohoto výpočetního modelu autoři zahrnují i tzv. operační paměť častěji známou pod anglickou zkratkou RAM) a architekturou grafické karty schopné provádět výpočty na bázi technologie CUDA.

Obrázek 2: Architektury CPU a GPU

Na výše uvedeném obrázku je vidět poměr aritmetickologických jednotek (tzv. ALU), které provádějí

samotný výpočet. Oproti standardním procesorům, které dnes v běžné podobě nabízejí nejvíce šest těchto jednotek, grafická karta jich může poskytnout několik stovek - v případě modelu NVIDIA GTS450, který jsem použil pro tuto práci, je to 192 ALU, extrémním případem je v současné době grafická karta NVIDIA GTX590, která nabízí uživateli 1024 výpočetních jader - čehož lze využít při výpočtu náročných operací, které lze tzv. paralelizovat, tedy lze provádět více úkonů naráz [6].

Dalším rozdílem mezi klasickým procesorem typu CPU a procesorem grafické karty typu GPU je to, jakým způsobem přistupují tyto procesory k paměti, ve které jsou uloženy pro výpočet důležité informace, jako jsou mezivýpočty a proměnné potřebné k běhu výpočetního úkonu. K těmto datům pak procesor přistupuje přes tzv. sběrnici, která má ale svá omezení co se týče objemu přenesených dat za jednotku času. Tato schopnost přenést určitý objem dat za jednotku času je nejčastěji uváděna výrobci jako tzv. šířka sběrnice. Operační paměť, kterou používá běžný procesor má v dnešní době šířku sběrnice 64 bitů, zatímco nejmodernější grafické karty mají šířku sběrnice až 768 bitů, z čehož vyplývá, že grafická karta dokáže nejen provést více výpočtů, ale dokáže zároveň mnohem rychleji přistupovat k datům pro tyto výpočty potřebným.

Velkým úskalím grafických karet je však počet instrukcí, které je každé jádro schopné zpracovat a vyšší náročnost na provedení běžných programátorských úkonů. O nutnosti vyšší náročnosti na sestrojení GPU vhodných algoritmů se zmíním ještě v kapitole č. 4.

3.1.2. Předpoklady programování na architektuře CUDA.

První nezbytnou součástí je tzv. CUDA ovladač, který nám poskytne nejdůležitější hardwarovou spolupráci grafické karty s knihovnami daného programovacího jazyka. Druhou součástí jsou pak CUDA nástroje a poslední částí pak je CUDA SDK (Software Development Kit) a ukázky zdrojového kódu.

CUDA ovladač poskytne vývojáři základní nástroje pro užití této architektury. Těmito nástroji jsou jmenovitě nvcc kompilátor pro jazyk C, CUDA FFT a BLAS knihovny, profiler (pro sledování běhu aplikace a prováděných operací), gdb debugger (pro jednodušší odstraňování chyb) pro GPU, CUDA běhový ovladač a programátorský manuál.

Rozšířením této základní sady je pak CUDA SDK, které obsahuje mnohem více užitečných materiálů ve formě ukázek zdrojového kódu. Mezi těmito ukázkami jsou ukázka paralelního řazení, násobení matic, transpozice matic, ladění výkonu a užívání časovačů, prefixové sčítání velkých polí, obrazová konvoluce a mnohé další [8].

3.2. Knihovny CUDA pro jazyk Java

Knihovny CUDA pro jazyk Java se nazývají jCUDA. jCUDA je jedním z řady projektů, které mají za svůj hlavní cíl zprostředkovat funkcionalitu technologie CUDA co nejširší společnosti vývojářů zabývajících se tématem využití grafických karet k jiným výpočtům, než herně a graficky orientovaným.
Poskytuje programátorům základní funkcionality programování na základě NVIDIA CUDA prostřednictvím knihovních funkcí a metod. Z hlediska mé práce je nejdůležitější, že poskytuje mimo jiné i knihovnu pro import původního C/C++ zdrojového kódu, jeho kompilování "za běhu programu" a následné spuštění tohoto pro grafickou kartu optimalizovaného programu nad daty, které mu předávám ze samotné aplikace napsané v jazyce Java.

3.2.1. Předpoklady programování s využitím jCUDA?

Pro programování s knihovnami jCUDA je třeba již výše zmíněných nástrojů (3.1.2.), jako je CUDA Toolkit, CUDA SDK, atd. Novou komponentou pro takto zaměřené programování je pak samotná sada knihoven jCUDA, která může být stažena ze stránek www.jcuda.org, pro jednoduchost práce s programem, který je přiložen k této práci, lze tyto knihovny pro nejrozšířenější operační systémy nalézt na přiloženém CD nosiči.

4. Návrh a implementace algoritmu

V této kapitole popisuji samotný návrh algoritmu a jeho implementaci postupně na CPU s pomocí výpočtů rozdělených do více vláken, CPU při použití pouze jednoho vlákna a grafické karty prostřednictvím technologií jCUDA a CUDA C/C++.

4.1. Návrh algoritmu

Nejprve se věnujme lehce výběru algoritmu z pohledu celkové složitosti a náročnosti na implementaci v praxi. V kapitole číslo dvě popisuji některé ze základních používaných algoritmů. Jistě neuniklo čtenáři této práce, že algoritmy na porovnání struktur v prostoru jsou popsány značně zevrubně, zatímco příklady algoritmů na porovnání sekvencí jsou popsány poměrně podrobně a je k nim připojen jakýsi návod v "pseudojazyku", který umožní případnému čtenáři této práce náhled na to, jak popsané algoritmy pracují a také poskytne základní informace pro případný pokus o implementaci těchto algoritmů.

Zatímco algoritmy pro porovnání v prostoru využívají různé techniky např. prohledávání grafů, jejich časová složitost je poměrně vysoká a jsou velice náročné na implementaci (nepoužíváme-li již předpřipravené knihovní funkce), časová složitost algoritmů pro porovnání sekvencí je naopak poměrně nízká a ve většině případů je časová náročnost výpočtu rovna násobku délek porovnávaných sekvencí.
Jelikož algoritmy pro porovnání v rovině pracují na bázi výpočtu podobnostní matice, jsou tedy i poměrně snadno implementovatelné jak pro klasické CPU výpočty, tak i pro výpočty na grafických kartách.

4.1.1. Teorie nutná pro návrh algoritmu

Když jsem před začátkem programování algoritmů, které jsou součástí programu přiloženého k této práci, přemýšlel jak algoritmus pro výpočet podobnosti napsat, přirozeně mě tedy napadlo nějakým způsobem vstupní sekvence (v mém případě molekuly proteinů) transformovat tak, aby se daly porovnat jako sekvence a tím snížit jak celkovou časovou složitost algoritmu, tak i složitost implementace. Celková náročnost tohoto úkonu se projevila také na výsledné délce zdrojového kódu programu, který je k této práci přiložen, jehož výsledná délka činí přes 2280 řádků.

Jelikož vstupními daty pro můj program dle zadání měly být soubory typu .pdb, neboli soubory databáze Protein Data Bank, které obsahují poměrně přesné údaje o každém atomu v sekvenci, mimo jiné například pozici atomu v prostoru, zda je příslušníkem nějakého řetězce. Společnou konzultací s vedoucím této práce jsem došel k názoru, že je vhodné porovnávat proteiny pomocí alfa C-α uhlíků srovnávaných proteinových molekul, protože pro většinu proteinových molekul je tento řetězec směrodatný, neboť tvoří základní kostru těchto molekul.

Dalším důležitým poznatkem je, že vstupem je struktura zadaná v prostoru, kterou lze popsat mimo jiné také vazebnými úhly, které tyto atomy navzájem svírají. V běžné praxi se užívá systému dvou úhlů, které pro určitou část molekuly, běžně se používá např. aminokyselina, v našem případě C-α uhlík, které určí poměrně přesně polohu další části stejného typu. V následujícím textu tyto úhly označuji pomocí řeckých písmen ϕ (čti fí) a druhý úhel pak řeckým písmenem ψ (čti psí).

4.1.2. Příprava vstupních dat

Před samotným popisem funkčnosti algoritmu bych rád dodal, že veškeré mezikroky předcházející výpočtu optimálního zarovnání proteinů jsou vytvořeným programem zapisovány do textových souborů, které lze výsledně nalézt ve stejné složce, v jaké se nachází soubor s optimálním zarovnáním.

Soubor .pdb, který je vstupem pro můj program obsahuje veliké množství informací, které jsou ve skutečnosti pro můj algoritmus nevýznamné, z tohoto důvodu nejdříve program vybere ze vstupních souborů pouze data odpovídající atomům v sekvenci a jejích vlastnostem.

Jelikož můj algoritmus z a vstup používá pouze ty atomy, které náleží do hlavního řetězce, ze souboru obsahujícího pouze údaje o atomech vybírám pouze ty atomy uhlíku, které určují alfa kostru vstupních řetězců, čímž jsem získal "určující" koordináty pro C-α uhlíky zadaných struktur.

Dalším krokem výpočtu je pak vypočítat úhly mezi atomy ze souboru, který byl vypracován v předchozím kroku. Existuje řada metod, které určení úhlů provádí mezi jinými například algoritmus Ye-Janardan-Liu. Ve svém algoritmu jsem se ale rozhodl využít jednoduché vektorové matematiky a na základě vstupních dat (hlavně pak polohy v prostoru) vypočítat jednotlivé úhly pomocí vzorce pro výpočet úhlů mezi vektory. Obecný tvar tohoto vzorce je na obrázku č. 3.

$$\cos\varphi = \frac{\vec{u}\cdot\vec{v}}{|\vec{u}|\cdot|\vec{v}|}$$

Obrázek 1: Výpočet úhlu z vektorů

Pokud chceme tohoto vzorce použít pro výpočet v prostoru, pak jeho znění tak, jak je využito v samotném programu, je zobrazeno na obrázku č. 4.

$$\cos\varphi = \frac{u_1v_1 + u_2v_2 + u_3v_3}{\sqrt{u_1^2 + u_2^2 + u_3^2}\cdot\sqrt{v_1^2 + v_2^2 + v_3^2}}$$

Obrázek 2: Úhel vektorů v prostoru

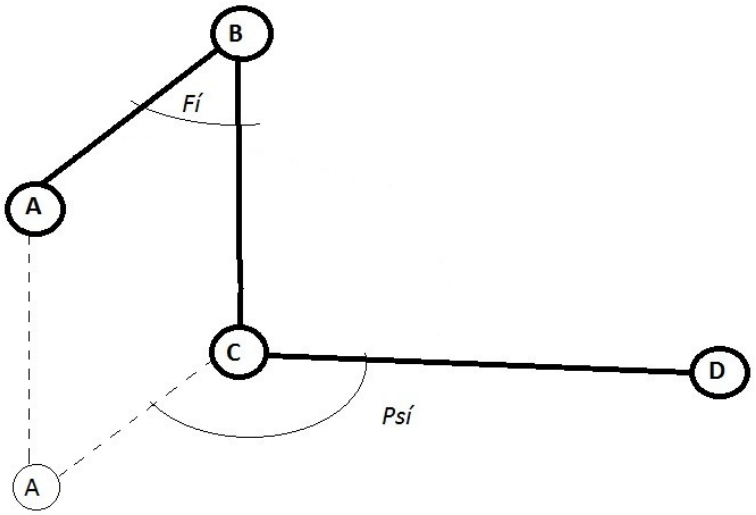
Kde u, v jsou vstupní vektory, u_x, v_x pro x jdoucí od 1 do 3 jsou jednotlivé složky vektorů.

Jak jsem již popsal výše, v této práci nepoužívám torzní úhly určující polohu aminokyselin v prostoru. V této práci je využita pouze jednoduchá vektorová matematika. Označení úhlů řeckými písmeny ϕ a ψ je tedy pouze obecné označení dvou úhlů a s typickým označení torzních úhlů nemá nic společného. Nechť A, B, C, D jsou postupně jdoucí C-α uhlíky v molekule proteinu, pak úhly ϕ a ψ pro atom A lze vypočítat následovně. Výpočet úhlu ϕ :

Nechť souřadnice vektoru *u* jsou postupně $(u_1,u_2,u_3) = (A_x\text{-}B_x, A_y\text{-}B_y, A_z\text{-}B_z)$, nechť vektor *v* je pak roven $(v_1,v_2,v_3) = (C_x\text{-}B_x, C_y\text{-}B_y, C_z\text{-}B_z)$, dosazením do výše uvedeného vzorce (viz Obrázek 4) pak získáme kosinus úhlu ϕ, při čemž aplikací funkce arkuskosinus pak získáváme samotný úhel ϕ.
Výpočet úhlu ψ : nechť souřadnice vektoru *u* jsou postupně $(u_1,u_2,u_3) = (A_x\text{-}B_x, A_y\text{-}B_y, A_z\text{-}B_z)$, nechť vektor *v* je pak roven $(v_1,v_2,v_3) = (D_x\text{-}C_x, D_y\text{-}C_y, D_z\text{-}C_z)$, dosazením do výše uvedeného vzorce (viz Obrázek 4) pak získáme kosinus úhlu ψ, samotný úhel ψ pak získáme opět aplikací funkce arkuskosinus.

Výše uvedeným způsobem jsou vypočteny veškeré vazebné úhly mezi atomy na hlavním řetězci proteinu pro každou vstupní sekvenci. Tyto úhly jsou poté uloženy do prostého textového souboru.

Na následujícím obrázku je ilustrace úhlů ϕ a ψ tak, jak jsou vypočítány výše uvedeným způsobem.



Obrázek 3: Úhly fí a psí

Samotný převod souřadnic v prostoru na dvojici úhlů ϕ a ψ je pak v programu obsluhován metodou *vectorAngle()*, která za své parametry bere souřadnice 4 atomů v prostoru a navrací textový řetězec (tzv. String).

4.2. Návrh implementací pro jednotlivé způsoby výpočtu

Zde se dostáváme k jednomu překvapení, jež jsem si, jakožto autor této práce, pro případného čtenáře/čtenářku připravil. Jak víme, zadáním této práce bylo implementovat jeden vybraný algoritmus pro zarovnání proteinů a to jak ve formě pro CPU, tak ve formě vhodné pro grafickou kartu.
Původně jsem zamýšlel použít pouze algoritmus pro semiglobální zarovnání proteinů, pokud si ale uvědomíme, jak malý rozdíl je v algoritmech pro semiglobální a globální zarovnání, pak se mnou jistě budete souhlasit, že by byla škoda neimplementovat hned oba dva.

Vstupem pro další zpracování je soubor s vypočtenými úhly mezi atomy C-α uhlíků.
Pro všechny metody výpočtu nejdříve z vstupných úhlů mezi C-α uhlíky vypočítám vždy pro každý pár ϕ a ψ vždy jedinečnou celočíselnou hodnotu, která je nezaměnitelná a povoluje vždy odchylku sedmi stupňů. Tento převod provádím v algoritmu z důvodu zjednodušení vstupních struktur na poměrně jednoduché a snadno srovnatelné sekvence, při poměrně malé ztrátě dat popisujících tuto strukturu v prostoru. Jak jsem již zmínil, v programu povoluji odchylku sedmi stupňů (tj. asi 0,125 radiánů), tato konstanta byla ověřena sérií výpočtů, při nichž se ukázala jako nejvhodnější. Vyšší hodnota, zhruba okolo deseti stupňů, již při provedení zarovnání vracela špatné výsledku už pro dvojici naprosto totožných proteinů, nižší pak již v kombinaci s hodnotící funkcí algoritmů, které popisuji níže, již byla velice a výsledné zarovnání také znehodnocovala. Toto nastavení lze změnit ve třídě *GetSequence()*.
Pro libovolnou dvojici úhlů ϕ a ψ je pak výsledná celočíselná hodnota stanovena následujícím výpočtovým vzorcem, který lze slovy zapsat jako:

Pokud úhel ϕ spadá do rozmezí i * 0,125 radiánů a (i+1) * 0,125 radiánů, pak výsledná celočíselná hodnota pro tento atom bude obsahovat k ve své první části, kde i je celočíselná hodnota jdoucí od 1 do 26.
Pokud úhel ψ spadá do rozmezí l * 0,125 radiánů a (l+1) * 0,125 radiánů, pak výsledná celočíselná hodnota bude na své druhé pozici obsahovat l, kde l je opět celočíselná hodnota, v tomto případě však jdoucí od 001 do 0026. V druhém případě je před všechny číslice přidány nuly z důvodu správnosti rozeznání úhlu. Tímto procesem se vyvaruji problému, kdy např. z prvního úhlu by celočíselná hodnota byla 11 a z druhého 1. Kdybych do druhé soustavy nepřidal číslice 0, pak by takováto hodnota byla zaměnitelná za případ, kdy první úhel je nahrazen číslicí 1 a druhý úhel je nahrazen číslicí 11. Pokud např. dvojice úhlů ϕ a ψ by byla 7 stupňů a 14 stupňů, pak celočíselný identifikátor takovéto struktury bude mít hodnotu 1001.

Touto operací se dostávám k diskretizaci samotného řešení, neboť vstupem pro další zpracování pomocí algoritmu pro výpočet optimálního globálního/semiglobálního zarovnání je již pouze posloupnost (ve zdrojovém kódu reprezentováno polem) unikátních celočíselných hodnoty, při čemž každá z nich popisuje určitou strukturu v prostoru. Jelikož ale struktura je popsaná pomocí celočíselných identifikátorů, pak mohu řešení zarovnání struktury v prostoru převést na "jednoduché" zarovnání sekvencí, které lze snadněji implementovat a má celkově nižší výpočetní nároky.

Pro návrh algoritmu je také důležité si uvědomit, že algoritmus pro nalezení globálního/semiglobálního zarovnání je jedním z tzv. záplavových algoritmů tedy, že pro výpočet jednoho prvku je třeba znalost jednoho, nebo více prvků předešlých, toto je fakt, který např. u výpočtu matice více vlákny souběžně, musíme brát v potaz a vhodně navrhnout řešení tohoto problému.

V následujících podkapitolách se budu věnovat postupu vyplnění matice M pomocí jednotlivých prostředků tj. jednoho vlákna CPU, více vláken CPU, grafické karty, neboť tento proces je náročný na výpočetní prostředky, tedy vhodný právě pro ilustraci rozdílných rychlostí výpočtu.

4.2.1. Výpočet pomocí jednoho vlákna na procesoru typu CPU

Zde je situace poměrně jednoduchá, protože při výpočtu jedním vláknem nehrozí chyba výsledku zásahem jiného výpočtu běžícího paralelně s výpočtem současným. Jednotlivé prvky matice M tak, jak je popsána v kapitole 2.3.3., jsou tedy vyplněny tak, že v řádku jsou nejdříve vyplněny veškeré sloupcové hodnoty a pak teprve následuje řešení dalšího řádku. Takto je vyplněna celá matice a po té následuje samotné hledání optimálního zarovnání sekvencí přesně, jak je popsáno v 2.3.3.

Níže uvedená ukázka zdrojového kódu (v jazyce Java) popisuje způsob, kterým je matice M vypočítána. Funkce getMax(i, j), jež za své parametry přijímá pozici zpracovávaného prvku, zde zastupuje funkci, která vyplňuje prvky matice dle schématu uvedeného v 2.3.3., potažmo 2.3.2. Hodnoty *aL* a *bL* zastupují délky zarovnávaných sekvencí. Hodnota *aL* zastupuje délku první sekvence, hodnota *bL* je pak totožná s délkou druhé vstupní sekvence.

```
int i = 1;
int j = 1;
do{
   do{
      getMax(i, j);
      j++;
   }while(j<bL+1);
   j = 1;
   i++;
}while(i<aL+1);
```

4.2.2. Výpočet pomocí více vláken procesoru typu CPU

Při výpočtu pomocí více vláken již nastává výše popsaná situace ohledně problému, který činí vlastnost algoritmu pro semiglobální zarovnání a to "záplavovost" tohoto algoritmu. Aby se zamezilo tomu, že některé z vláken znehodnotí výpočet tím, že by současně přepisovalo hodnotu společně s jiným vláknem, lze předejít pomocí tzv. zamykání prvků. Zjednodušeně zámek je určitý ochranný prvek, který každé vlákno kontroluje před provedením výpočtu a zjišťuje tak, zda již prvek, který se snaží vypočíst není vypočten jiným vláknem.

Abychom předešli problémům při vyplnění, stačí algoritmus z 4.2.1. modifikovat tak, že vložíme obsluhu zámku, který je podobný vypočítávané matici, na rozdíl od ní se však sestává pouze z tzv. pravdivostních hodnot (tzv. datový typ boolean), kde hodnota *true* na určité pozici reprezentuje stav, kdy prvek na této pozici již byl, nebo je zpracován jiným vláknem, hodnota *false* pak značí, že tento prvek ještě vypracován není a vlákno jej tedy může zpracovat.

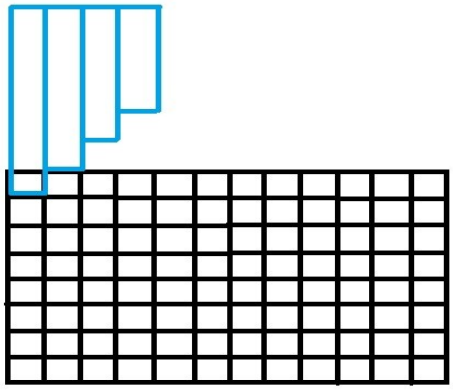
```
int i = 1;
int j = 1;
do{
   do{
      if (!lock[i][j]){
         lock[i][j] = true;
         getMax(i, j);
      }
      j++;
   }while(j<bL+1);
   j = 1;
   i++;
}while(i<aL+1);
```

Na výše uvedeném úryvku zdrojového kódu je vidět, jak vypadá modifikace s využitím zámku. V praxi, když vlákno dojde na prvek matice M na pozici *i*, *j*, tedy M[*i*, *j*], zkontroluje zda již prvek není náhodou vypočítán, pokud je, pak jej přeskakuje a snaží se vypočíst prvek následující. Zde je nutné zmínit, že samotný proces zamykání je z hlediska složitosti poměrně "drahý" proces, což ještě zmíním v kapitole rozebírající naměřené výsledky.

4.2.3. Výpočet pomocí více vláken na grafické kartě

Nyní přichází hlavní implementace algoritmu pro semiglobální zarovnávání a to implementace na grafické kartě. Nejprve je nutné si uvědomit, že grafická karta na rozdíl od procesoru typu CPU poskytuje pouze tzv. striktní paralelizaci, tedy takovou, že všechna vlákna vždy běží bok po boku a nelze u nich využít žádnou instrukci typu *wait()*, která by některému vláknu přikázala vyčkat výsledku vlákna předchozího. Stejně jako absence instrukce *wait()*, není možné na grafické kartě využít zamykání, nebo by jej bylo velmi "drahé" implementovat. Z těchto důvodů tedy nelze použít žádný z předchozích algoritmů, protože sekvenční přístup v nich prováděný nelze na grafické kartě jednoduše reprodukovat.
Dalším problémem je fakt, že samotná technologie CUDA nám neumožňuje užívat jiné, než aritmetické operace násobení, sčítání, odčítání, či dělení pro matice reprezentované pomocí dvourozměrných polí.

Vyhnout se problému s použitím dvourozměrného pole je poměrně jednoduché - z matice vytvoříme jednorozměrné pole tak, že řádky sepíšeme postupně za sebe do pole jednorozměrného, vhodnou prácí s parametry jsme pak s tímto polem pracovat jako s maticí, neboť délka řádku a jejich počet je nám známa - abychom se ale vyhnuli problémům s absencí zamykání, musíme první potenciální problém, tj. striktní paralelizaci, proměnit v naši největší "zbraň". Na následujícím obrázku je znázorněno, jak bude výpočet na grafické kartě probíhat.



Obrázek 4: Výpočet na GPU

Černě vyznačená ve výše uvedeném obrázku je ta část matice M, kterou je nutné vypočítat, modře vyznačeny jsou pak pomyslné pozice vláken při výpočtu. Vlákna tedy provádí výpočet v diagonále, která má předem přesně určenou délku a tou je právě počet spuštěných vláken. Striktní paralelizace nám v tomto zaručuje to, že když vlákno s identifikátorem *n* provádí výpočet pak vlákno *n-1* již ukončilo výpočet všech prvků potřebných k výpočtu prvku vláknem *n*. Ve chvíli, kdy některé vlákno vypočte prvek v posledním řádku matice M, přesouvá se na pomyslný konec diagonály a začíná výpočet sloupce na pozici odpovídající pozici vlákna. Obrázek 6 je pouze ilustrační, protože ve skutečnosti, je algoritmus použitý v přiloženém programu nastaven tak, že používá až 256 vláken současně, také matice, jejíž hodnoty v praxi vyčíslujeme, je mnohem větších rozměrů.

Následující zdrojový kód popisuje, jakým způsobem pracuji na grafické kartě s maticí M a jakým způsobem probíhá vyplnění jednotlivých prvků této matice. Pozn. matice M je ve zdrojovém kódu reprezentována polem dDataOut, dDataInA a dDataInB jsou pak pole obsahující unikátní celočíselné hodnoty reprezentující vstupní proteinové sekvence.

```
__device__ void operator () () {
  int sz = (n2 + BLOCKSIZE - 1) / BLOCKSIZE;
        for ( int i = 0; i < sz; ++i ) {
              int b = TX + 1 + BLOCKSIZE * i;
              for ( int a = 1 - TX; a < n1 + 1 + (BLOCKSIZE - TX - 1); ++a ) {
                    if ( b < n2 + 1 && a >= 1 && a < n1 + 1 ) {
                          getMax( a, b );
                    }
                    __syncthreads();
              }
        }
}
```

Index *a* ve výše uvedeném zdrojovém kódu reprezentuje pozici aktuálního řádku v matici, index *b* pak popisuje pozici sloupce aktuálně zpracovávaného prvku. Funkce getMax(*a,b*) opět plní funkci ohodnocení právě zpracovávaného prvku. Parametr BLOCKSIZE obsahuje velikost zpracovávaného bloku, v našem případě onu pomyslnou "diagonálu" vláken, která je v programu pevně nastavena na 256. Parametry *n1* a *n2* jsou délky zarovnávaných sekvencí. Podmínky typu *if* ve výše uvedeném zdrojovém kódu plní funkci ošetření toho, že vlákna se nesnaží vyplňovat položky, které neexistují (mimo rozsah matice). Toto je nutné provést, neboť v případě programování pro grafické karty přistupujeme do paměti přímo.

5. Rozbor výsledků

Jak jsem již dříve zmiňoval, obsluha zamykání u výpočtu pomocí více vláken na procesoru typu CPU je natolik "drahá" záležitost, že se při praktických měřeních zjistilo, že standardní průběh pomocí jedno vlákna je rychlejší, než při užití více vláken. Příjemně však překvapily hodnoty výpočtu matice M velkých rozměrů pomocí technologie CUDA, která veškeré hodnoty této matice vyčíslila až několikrát rychleji, než konkurenční metody. Na malých vstupních datech (které ale v praxi nemohou nastat), jakými byla pro mě pomyslná matice o rozměrech 6*6 však i výpočet prostřednictvím grafické karty proběhl pomaleji, než výpočet pomocí jednoho vlákna na CPU. Tento fakt přikládám složitosti inicializace samotného grafického výpočtu a nutným operacím kopírováním mezi pamětí počítače a pamětí grafické karty (kopírování paměti do paměti grafické karty, spuštění samotného výpočtu a opětovné kopírování do paměti počítače po vypočtení).

Nejdříve bylo nutné ověřit vnitřní korektnost programu, tj. veškeré tři metody výpočtu vrací shodný výsledek, k tomuto účelu jsem využil linuxového programu *diff*, resp. jeho varianty *vimdiff*, pomocí kterých jsem zkontroloval na testovací sadě shodnost vygenerovaných souborů (tj. souborů obsahující vygenerovanou matici a optimální zarovnání). Ve všech případech byly tyto soubory shodné.

Korektnost programu jsem po té ověřil na triviálních datech, rozumějme první i druhá sekvence jsou totožné, kdy program ukončil výsledek a optimální zarovnání tvořilo přiřazení prvku na pozici *n* v druhé sekvenci na prvek na stejné pozici v sekvenci první. Stejně tak jsem ověřil korektnost programu na dalším triviálním případě a to na případě, kdy druhá sekvence je "podsekvencí" prvního řetězce. V takovémto případě druhá sekvence byla náležitě zahrnuta celá v optimálním zarovnání.

Pro ověření korektnosti programu a jím prováděných algoritmů jsem provedl porovnání různých vzorků z databáze CATH. Tato měření rozepíšu v následujících podkapitolách.

5.1. Databáze CATH

Databáze CATH je databáze proteinů, která seřazuje, v současné době, 104 238 proteinových struktur dle jejich podobnosti dané různými vlivy, hlavně pak jejich evolučním vývojem. Tato databáze klasifikuje proteiny a rozděluje je do čtyř hlavních tříd.

Tzv. třída *C* je určena celkovou kompozicí proteinu a celkovým tvarem struktury proteinu . Třída *A* klasifikuje proteiny na základě struktury proteinu a orientace sekundárních struktur v molekule.
Třída *A* však již nebere v potaz, jakým způsobem jsou k sobě jednotlivé "podstruktury" navázány. Třída *T* je ta část databáze, která kategorizuje proteinové struktury na základě jejich topologického vývoje, jejím základem je vždy tzv. topologické jádro, o kterém se předpokládá, že ostatní proteiny v této kategorii se z něj postupnými evolučními procesy vyvinuly. Poslední základní třídou je tzv. třída *H*, která určuje podobnost proteinů na základě homologické příslušnosti do dané proteinové rodiny. Do této třídy jsou zařazeny takové dvojice proteinů, o nichž víme, že mají společného evolučního předchůdce, vzájemně jsou však prostorově odlišné (např. různá orientace úhlů spojujících hlavní řetězce s řetězci sekundárními).

Pro každou z hlavních určujících vlastností proteinové sekvence je však ještě vypracována další podobnost, pomocí jíž jsou proteiny zařazeny případně ještě do třídy *S*, neboli třídy sekvenčně podobných proteinů. V této třídě jsou obsaženy například i takové dvojice proteinů, o kterých víme, že mají rozdílného evolučního předchůdce, ale postupným vývojem se jejich struktury čím dál, tím více začaly navzájem podobat. V této třídě jsou proteiny řazeny dle tzv. pravidla **S, O, L, I, D** pod kategorii **S** jsou zařazeny takové proteiny, jejichž struktura není více podobná, než 35 %. V kategorii **O** jsou proteiny, které se navzájem strukturně podobají až v šedesáti procentech. Kategorie **L** obsahuje proteinové sekvence, které se navzájem podobají až v devadesáti pěti procentech. V kategorii **I** pak nalezneme proteiny, které jsou navzájem téměř totožné. Poslední kategorie **D** pak obsahuje ty proteiny, které jsou absolutně identické.

U každého proteinu, který si její uživatel zobrazí, je uvedena, pokud existuje, skupina jiných proteinů, které sdílí jistou podobnost s proteinem již vybraným a ke každému z takových "sousedů" je také uvedena percentuální podobnost se zobrazeným proteinem. Z tohoto důvodu je tato databáze vhodná pro srovnání korektnosti algoritmů, které jsem pro účel této práce naprogramoval [1].

5.2. Ověření korektnosti na základě vzorků z podobnostní databáze CATH

Z výše uvedených důvodů (v podkapitole 5.1) jsem tedy přistoupil k ověření korektnosti mnou naimplementovaných algoritmů na základě podobnostních pravidel databáze CATH.

Jako základní vzorek, vůči kterému jsem porovnával sekvenční podobnost jsem pojal protein, který je znám pod kódovým označením 1oaiA00 (struktura je na obrázku č.7), ke kterému jsem poté vybral vždy reprezentanta z některých výše uvedených skupin v databázi CATH (i těch, kde podobnost v procentech byla natolik malá, že výsledné optimální zarovnání tvořilo pouze několik málo atomů).
Pro každou takovouto dvojici proteinů jsem provedl výpočet pomocí přiložené programu a porovnal optimální zarovnání s již známou percentuální podobností dle databáze CATH. Jednoduchým výpočtem jsem pak ověřil, že počet zarovnaných atomů děleno počtem atomů první struktury, odpovídá, s případnou menší tolerovatelnou odchylkou, percentuální podobnosti dle databáze CATH.

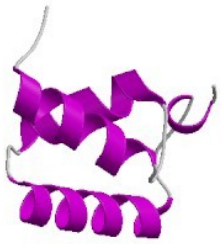
Pro dvojici proteinů s kódovým označením 1oaiA00 a 2chgA02, jejichž podobnost dle již výše zmíněné databáze činí 82.73 % algoritmus globálního zarovnání vrátí následující optimální zarovnání.

(56,63)(55,63)(54,63)(53,63)(52,62)(51,61)(50,60)(49,59)(48,58)(47,57)(46,57)(46,56)(45,56)
(44,56)(43,56)(42,55)(42,54)(42,53)(42,52)(42,51)(42,50)(42,49)(42,48)(42,47)(41,46)(40,46)(40,45)
(40,44)(39,44)(39,43)(39,42)(39,41)(38,40)(37,40)(37,39)(36,38)(36,37)(35,36)(35,35)(34,34)(33,33)
(32,32)(31,31)(30,30)(30,29)(30,28)(29,27)(28,27)(27,26)(26,26)(25,25)(24,24)(23,23)(23,22)(22,21)
(21,20)(20,19)(19,18)(18,17)(17,16)(16,15)(15,15)(14,14)(14,13)(14,12)(14,11)(14,10)(14,9)(13,8)
(12,8)(11,7)(10,6)(9,5)(9,4)(8,3)(7,3)(6,2)(6,1)(5,1)(4,1)(3,1)(2,1)(1,1)

Což odpovídá přibližně 82 % (přesně 82,142 %) podobnosti. Drobná odchylka od procentuální podobnosti z databáze CATH může být způsobena diskretizací úhlů prostřednictvím vektorové matematiky a také způsobem, jakým algoritmus globálního zarovnávání pracuje.

Tímto způsobem jsem tedy ověřil výsledky pro veškeré kategorie podobností, vždy s již výše uvedeným proteinem 1oaiA00 a reprezentantem jiné kategorie abych ověřil správnost výsledků navracených programem. Pro nejbližší podobnostní okolí určitého proteinu, jsem provedl měření a porovnání hodnot známých s hodnotami v databázi CATH - tuto ukázku lze vidět v tabulce 1. Pro vzdálené okolí jsem pak vybral náhodného reprezentanta a ověřil, že podobnost s výše uvedeným proteinem spadá do rozmezí stanoveného databází CATH dle jejich pravidel vypsaných výše. Např. pokud protein A měl nějaký jiný protein zařazen do kategorie S, pak se podobnost měla pohybovat mezi 0 a 35 %.

Během těchto měření se vyskytly odchylky od databáze CATH, které mohou být způsobeny několika faktory. Prvním z těchto důvodů může být způsob porovnávání proteinových molekul tak, jak je použit v mnou sestrojeném programu a jak je použit v algoritmech, které jsou použity pro stanovení výsledku pro účely databáze CATH. Databáze CATH používá pro stanovení podobnosti metodu SIMAX, která je jedním z reprezentantů algoritmů pro porovnání v prostoru, tedy je zde předpoklad, že je mnohem přesnější, naopak je však výpočetně složitější. Dalším faktorem ovlivňujícím přesnost může být povolená odchylka úhlů, která dovoluje odlišnost až 7 % u vzájemně srovnávaných úhlů dvou struktur.



Obrázek 5: Protein 1oaiA00

Následující tabulka obsahuje ukázku vzorků proteinů, které jsem porovnával. V tabulce je pouze několik příkladů z prováděných měření, v druhém sloupci je vždy podobnost (vyjádřená v procentech) tak, jak byla spočtena přiloženým programem, ve třetím pak percentuální podobnost, shodného porovnání, která je zapsaná v databázi CATH.

Tabulka 1: porovnávané proteiny a zjištěné podobnosti

Podobnost/porovnávané proteiny	Vypočteno programem	Z databáze CATH
1oaiA00 - 1dv0A00	89,2 %	85,42 %
1oaiA00 - 1wivA00	85 %	79,80 %
1oaiA00 - 2chgA02	82,142 %	82,73 %
1h12A00 - 1kwfA00	80,55 %	83,94 %
1mz9A00 - 1czqA00	92,85 %	94,18 %
1g6sA01 - 2pqcA01	84,16 %	87,15 %
1g6sA01 - 2pqcA02	80,69 %	78,55 %

5.3. Ověření korektnosti algoritmu na základě jiného nástroje

Dalším způsobem, jak ověřit korektnost přiloženého programu pak bylo porovnání s výsledkem získaným z již zavedených nástrojů, které jsou běžně používány a jsou dostupné na internetu, například Secondary Structure Element Alignment (SSEA), který je dostupný na adrese http://protein.cribi.unipd.it/ssea/. Ve všech případech vracel tento nástroj shodný, nebo s odchylkou pár atomů podobný, výsledek. Tento výsledek byl pro účely této práce směrodatný, neboť, na rozdíl od porovnání s databází CATH, výše uvedený nástroj používá totožný způsob výpočtu optimálního zarovnání. Kdy k výpočtu podobnosti jak programem, který je přiložen, tak nástrojem SSEA bylo použito algoritmu globálního zarovnání proteinů.

5.4. Rozbor rychlostí používaných metod

V této podkapitole se věnuji rozboru a porovnání rychlostí všech tří implementovaných metod výpočtu. Porovnávám zde tedy rychlost výpočtu sekvencí o různých velikostech pomocí jednoho vlákna na CPU, více vláken na CPU a pomocí grafické karty.

Jako testovací sestavu jsem použil stolní počítač současné "střední" kategorie jehož cena se pohybuje okolo dvanácti tisíc korun. Jeho parametry ovlivňující běh výpočtů jsou následující:

CPU procesor:	AMD Phenom X4 840 - 4 jádra, tak každého jádra 3,2 GHz
Operační paměť:	8 GB - DDR3 o frekvenci 1600 MHz
Grafická karta:	NVIDIA GeForce GTS450 - 192 jader, takt každého jádra 783 MHz, frekvence paměti grafické karty 3608 MHz.
Operační systém:	Fedora Linux 13 - 64 bitová architektura

Testy byly prováděny pouze na shodných částech algoritmu, tedy vyplnění matice M. U vláken procesoru byly čas pomyslných stopek spuštěn ve chvíli, těsně před spuštěním prvního vlákna, zastaven pak byl ve chvíli ukončení posledního vlákna provádějícího výpočet. U grafické karty pak byl tento čas měřen od inicializace paměti paměti grafické karty po provedení kopírování výsledku z paměti grafické karty zpět do hlavní paměti.

Během testů za pomoci testovací sady proteinů různých délek algoritmy vyplnění matice pomocí jednoho vlákna, více vláken a pomocí grafické karty dosáhly následujících výsledků (v polích tabulky je znázorněn čas výpočtu v milisekundách):

Tabulka 2: Porovnání rychlosti výpočtu

Rozměr matice/ metoda výpočtu	6*6	37*56	828*324
1 vlákno CPU	3 ms (±1,7 ms)	4 ms (±1,72 ms)	35 ms (±1,76 ms)
4 vlákna CPU	6 ms (±1,7 ms)	7 ms (±1,72 ms)	49 ms (±1,76 ms)
Grafická karta	1 ms (±1,7 ms)	2 ms (±1,72 ms)	22 ms (±1,76 ms)

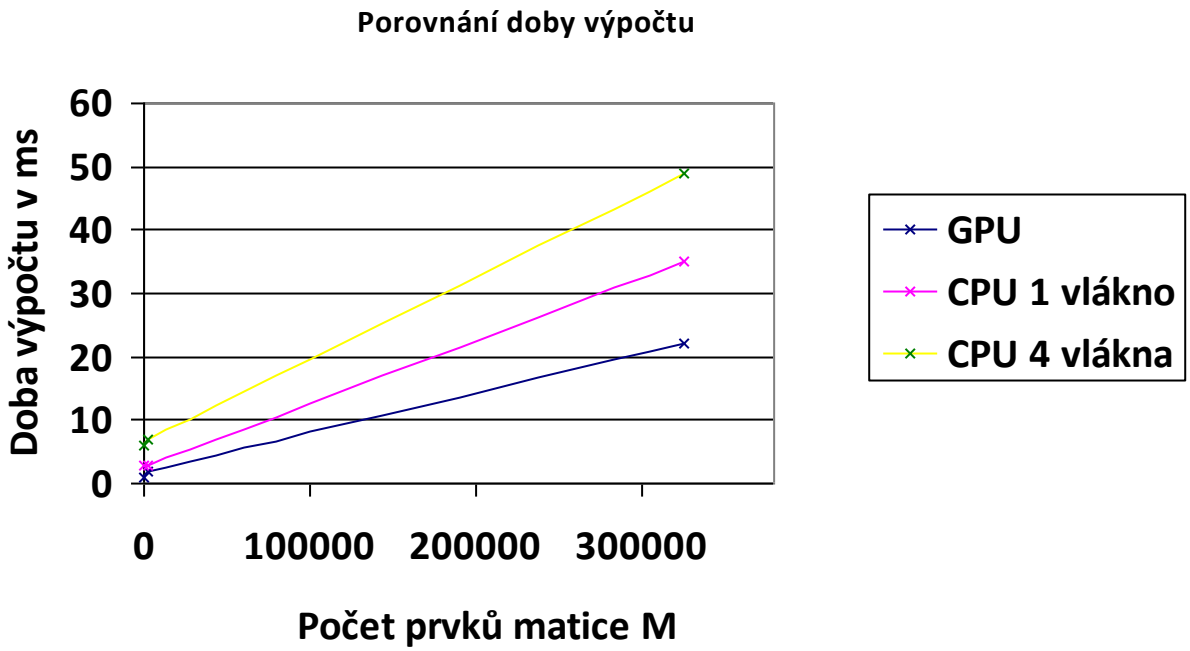
Výše uvedené délky dob výpočtu byly stanoveny, jako průměrný čas z pěti měření (pro dané dva vstupní soubory) zaokrouhlený na celé milisekundy. V závorce je pak uvedena směrodatná odchylka, která u všech měření činila 1,7 až 1,76 ms. Tato odchylka mohla být způsobena například ukládáním údajů do mezipaměti, či aktuálním vytížením procesoru jiným systémovým procesem, což jsou jevy, které nelze ovlivnit pomocí mého programu.

Z výše uvedených výsledků vyplývá několik věcí, v třetím sloupci tabulky 1 (37*56) vidíme, že rychlost výpočtu grafické karty mírně klesla, toto je způsobeno obráceným pořadím sekvencí, kdy jako první sekvence byla zadána sekvence kratší, jako druhá sekvence byla zadána struktura s větším počtem uhlíků na hlavním řetězci. Toto je však dle funkčnosti algoritmu pro výpočet na grafické kartě neoptimální zadání - optimálním zadáním je výpočet, kdy první sekvence je delší, v takovém případě je využit maximální možný počet jader grafické karty - stejný výpočet v obráceném pořadí struktur probíhá opět v čase 1 ms. Je také vidět, že grafická karta provedla výpočet v každém případě rychleji, až několikanásobně, než klasický procesor.

Dalším významným závěrem těchto měření je také fakt, že rychlost výpočtu na CPU pomocí metody výpočtu více vlákny - sekvenčním přístupem k matici a zamykání - v každém případě proběhla pomaleji, než výpočet pomocí jednoho vlákna. Tento fakt je způsobem vysokou časovou náročností obsluhy zamykání a předcházení kolizi vláken, při opravdu velkých rozměrech matice ovšem ale tento přístup rychlost výpočtu jedním vláknem vyrovnává.

Nutno ještě podotknout, že při výpočtu pomocí grafické karty nebyla v grafickém kernelu použita technologie sdílené paměti, která by současné výsledky rychlost výpočtu pomocí grafické karty mohla ještě zlepšit. Vzhledem k navržené metodě výpočtu na grafické kartě by však její případná implementace byla poměrně složitá.

Porovnání délky doby výpočtu vzhledem k zvolené metodě výpočtu je i znázorněno v následujícím grafu.



6. Závěr

Cílem této bakalářské práce bylo nastudování porovnání proteinových sekvencí, implementace vybraného algoritmu jak pro výpočet na klasickém procesoru typu CPU, tak na výpočet optimalizovaný pro běh na grafických kartách.

Během vypracovávání této práce jsem si nejen osvojil a upevnil techniky běžného programování v jazyce Java, ale rozšířil je i o podstatnou znalost programování v jazyce C/C++ CUDA. Praktická část této práce pak dokázala, že předpoklad navýšení rychlosti výkonu výpočetních operací na grafické kartě je oproti standardnímu procesoru typu CPU na rozsáhlých datech na tolik veliký, že případné aplikace různých podobných algoritmů na technologii CUDA mohou značně zrychlit jejich provedení a tím i usnadnit práci lidem, kteří s takto výpočetně složitými operacemi musí pracovat. Při porovnání metody výpočtu pomocí jednoho vlákna a více vláken na klasickém CPU jsem ale zjistil, že pomyslné rčení: "Více může znamenat i méně," má stále i v dnešní době své opodstatnění.

Jako možná rozšíření této práce do budoucna pak vnímám implementaci algoritmu pro lokální zarovnání sekvencí algoritmem Smith-Watterman a také implementaci sdílení paměti mezi vlákny grafické karty, která by současné, již velice dobré, výsledky a rychlosti provedení algoritmů mohla ještě znatelně vylepšit.

7. Použité zdroje

7.1. Základní literatura

1. Blelloc, Guy E. *Vector Models for Data-Parallel Computing.* Massachusets, The MIT Press, 1990.

2. Do, Chuong B. a Katoh, Kazutaka. *Protein Multiple Sequence Alignment*. Stanford, Stanford Press, 2008.

3. NVIDIA Corporation. *NVIDIA CUDA Programming Guide* [online]. Santa Clara, 2010. 3. aktualizované vydání. Dostupné na World Wide Web: http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf

7.2. Citace

[1] CATH. *The CATH database* [online]. Londýn, 2010 [cit. 5. května 2011]. Dostupné na World Wide Web: http://www.cathdb.info/wiki/doku.php?id=about:intro

[2] Holm, L. a Sander, C. *DALI: Distance-matrix ALIgnment* [online]. Chalmers University of Technology, 1996 [cit. 10. Dubna 2011]. Dostupné na World Wide Web: http://www.cse.chalmers.se/~kemp/teaching/TDA506/2010-2011/lecture10.pdf

[3] JAVŮREK, Karel. *Ohlasy ze světa: srovnání výkonu PhysX* [online]. 2008 [cit. 21. Března 2011]. Dostupné na World Wide Web: http://www.zive.cz/Clanky/Ohlasy-ze-sveta-srovnani-vykonu-PhysX/sc-3-a-143159/default.aspx

[4] Mathura, Venkatarajan Subramanian a Kangueane, Pandjassarame. *BIOINFORMATICS: A CONCEPT-BASED INTRODUCTION.* New York: Springer, 2009. ISBN: 978-0-387-84869-3.

[5] *Matlab GPU computing with NVIDIA CUDA-Enabled GPUs. Dostupné na World Wide Web:* http://www.mathworks.com/discovery/matlab-gpu.html

[6] NVIDIA Corporation. *GeForce GTX590* [online]. Santa Clara, 2011 [cit. 10. Dubna 2011]. Dostupné na World Wide Web: http://uk.geforce.com/hardware/desktop-gpus/geforce-gtx-590/specifications

[7] NVIDIA Corporation. *What is CUDA* [online]. Santa Clara, 2009 [cit. 21. Března 2011]. Dostupné na World Wide Web: http://www.nvidia.com/object/cuda_what_is.html

[8] NVIDIA Corporation. *What is CUDA?* [online]. Santa Clara, 2009 [cit. 10. Dubna 2011]. Dostupné na World Wide Web: http://www.nvidia.com/object/what_is_cuda_new.html

[9] Shindyalov, Ilya N. a Bourne, Philip E. *Protein Structure Alignment by Incremental Combinatorial Extension (CE) of the Optimal Path* [online]. San Diego, 1998 [cit. 10. Dubna 2011]. Dostupné na World Wide Web: http://cl.sdsc.edu/ce/ce.pdf

[10] Ústav biologie a lékařské genetiky 1.Lékařské fakulty Univerzity Karlovy. *Aktuální genetika* [online]. Praha, 2005-2006 [cit. 21. Března 2011]. Dostupné na World Wide Web: http://biol.lf1.cuni.cz/ucebnice/proteomika.htm

7.3. Seznam použitých zkratek a výrazů

C - objektově orientovaný programovací jazyk vyvinutý v sedmdesátých letech dvacátého století.

C++ - objektově orientovaný programovací jazyk, který se vyvinul z jazyka C, jedná se o jakousi nadstavbu jazyka C.

CATH - databáze proteinů rozdělených do skupin podle evoluční, strukturní, topologické a architekturní homologie.

CPU - procesorová jednotka provádějící výpočetní operace v počítači. CPU je zkratka z anglického Control Processing Unit.

CUDA - technologie navržená společností NVIDIA, která umožňuje provádět výpočetně náročné operace na grafických kartách s minimálním využitím standardního procesoru typu CPU. CUDA je zkratkou z anglického Compute Unified Device Architecture.

GPU - grafická procesorová jednotka, která stnadarndně vykonává operace spojené s výpočtem a zobrazením grafiky v počítači. GPU je zkratkou z anglického Graphic Processing Unit.

Java - objektově orientovaný programovací jazyk původně vyvinutý firmou Sun a v současné době vyvíjený firmou Oracle.

PDB - proteinová databáze, jejíž oficiální název je Protein Data Bank (odtud také zkratka PDB). Tato databáze obsahuje soubory ve formátu .pdb, které jsou v podstatě textovým zápisem struktury organických sloučenin.

SDK - sada nástrojů pro vývoj počítačových programů. Tato zkratka vznikla z anglického Software Development Kit.

8. Přílohy

8.1. Návod k přiloženému programu

8.1.1. Instalace

Nejdříve je nutné nainstalovat veškeré nástroje popsané v kapitole 3.1.2., tedy jmenovitě CUDA SDK, CUDA Toolkit a NVIDIA CUDA Driver. Pokud tyto nástroje nebudou na počítači nainstalovány, veškeré funkce, které na této technologii závisí budou automaticky vypnuty.

Majitelé operačních 64bitových operačních systémů Windows a Linux budou mít práci snazší, neboť pro ně jsou předpřipraveny již zkompilované formy programu v podsložce ..ProteinAligner/dist. Pokud tedy máte operační systém Windows (XP, Vista,7, 2008 R2) pro architekturu amd64 pak lze využít pro spuštění programu skript, který se nachází v ...ProteinAligner\dist\windows_64\ProteinAligner.bat. Pokud spouštíte program na operačním systému Linux pro architekturu amd64, pak lze spustit program pomocí skriptu ...ProteinAligner/dist/linux_64/ProteinAligner.sh.

Při případném problému s knihovnami jCUDA se majitelé dvou výše vyjmenovaných variant mohou přesunout přímo na návod 8.1.2.

Pokud jste majitelem operačního systému Windows, který obsahuje nástroj pro "Řízení uživatelských účtů" pak je třeba výše uvedené skripty, či program spouštět jako "správce", nebo tento nástroj před použitím programu vypnout, druhá z variant je jistější a vede k zaručenému úspěchu. Pokud tento krok neprovedete, pak program s velkou pravděpodobností během svého chodu nejspíše zahlásí některou z výjimek, např. chyba načtení knihoven jCUDA, nebo neschopnost nalézt a spustit kompilátor nvcc.exe, případně jeho část cl.exe. Operační systém Windows se spuštěným výše zmíněným nástrojem totiž nepovolí programu přístup k souborům, které jsou uloženy v tzv. systémových složkách, což jsou např. i soubory java knihoven, programy a kompilátory ve složce C:\Windows, atd. Tyto součásti jsou ale nezbytně nutné pro běh programu.

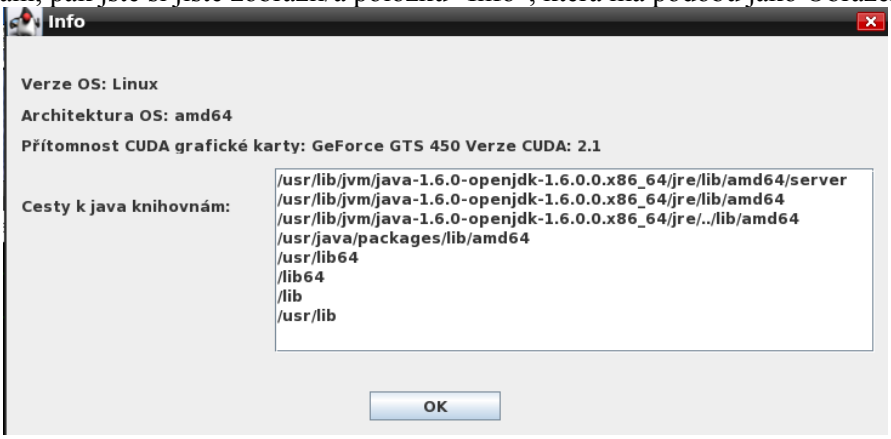
8.1.1.a Kompilace programu

Pro úspěšné spuštění programu je nejdříve nutné program zkompilovat. Za tímto účelem je třeba nainstalovat vývojové prostředí Netbeans, které lze bezplatně stáhnout z webových stránek www.netbeans.org. V tomto prostředí pak klikneme na tlačítko "Open Project" a vyhledáme cestu k samotnému programu. Po otevření projektu klikneme na tlačítko "Compile and run", v této chvíli, pokud na počítači nemáte nainstalovány knihovny jCUDA, pak budete varováni kompilátorem, že skutečně přítomny nejsou. Pokud je to váš případ, pak klikněte na položku "skip errors and compile", program se spustí, v menu "Nápověda" samotného programu pak zvolte položku "Info", která vám zobrazí informace o vašem počítači nutné k úspěšnému pokračování spuštění veškeré funkcionality tohoto programu. Pokračujete dle 6.1.2. Pokud ovšem na svém počítači nemáte grafickou kartu kompatibilní s architekturou CUDA, krok 6.1.2. přeskočte, neboť by to byla zbytečná ztráta času, v takovém případě vám totiž nebude CUDA funkcionality programem zprostředkována.

8.1.1.b Kopírování knihoven jCUDA a řešení referenčních problémů

Jelikož jCuda knihovny nejsou napsány v tzv. "přenositelné" formě, je nutné před samotným spuštěním programu překompilovat a zkopírovat jCUDA knihovny do potřebných systémových složek.

Veškeré jCUDA knihovny jsou přiloženy v podsložce "cuda", která se nachází ve složce "ProteinAlignmer", ve formě zkomprimovaných souborů. Pokud čtete tento oddíl po předchozím zklamání, pak jste si jistě zobrazil/a položku "Info", která má podobu jako Obrázek 8.



Obrázek 6: Info dialog

V textovém poli "Cesty k java knihovnám" jsou uvedeny veškeré cesty, které je možné použít pro následující postup.

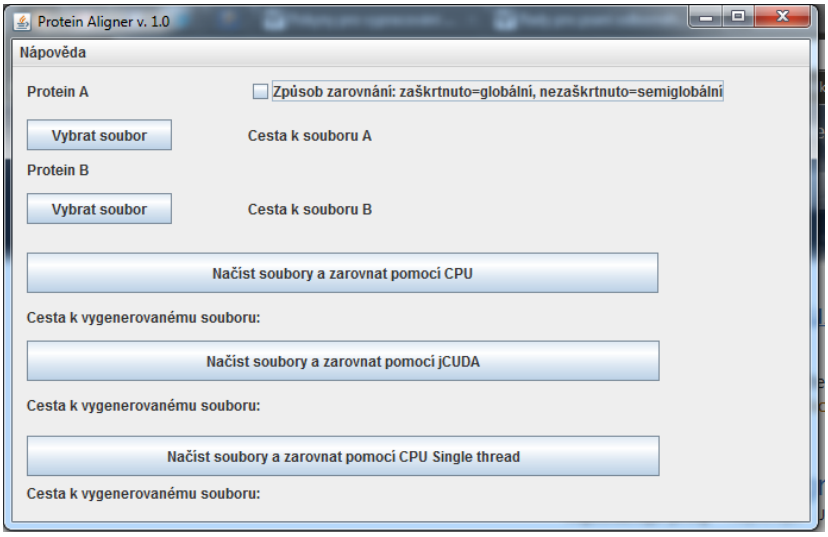
Nejdříve přejděte do výše zmíněné složky "cuda", po té přejděte do podsložky, která označuje vaši architekturu a verzi operačního systému. Rozbalte komprimovaný soubor, který se v ní nachází, pokud jste uživatel/ka operačního systému Microsoft Windows, pak označte veškeré položky končící na .dll, pokud ne, pak označte veškeré položky končící na .so a zkopírujte je do jedné z výše uvedených lokací. Pozn. na obrázku jsou ilustrovány cesty k java knihovnám v operačním systému Fedora Linux.

Dalším krokem je ukončení programu a vyřešení referenčních problémů v prostředí Netbeans. V tomto prostředí klikněte pravým tlačítkem myši na projekt "ProteinAligner" a vyberte položku "Resolve reference problems", objeví se dialogové okno, ve kterém zvolte cesty k jednotlivým komponentám knihoven jCUDA (cesty k vámi rozbaleným souborům).

Nyní můžeme přejít k "čistému" spuštění programu, klikněte tedy na položku "Compile and run".

8.1.2. Uživatelské rozhraní

Tato podkapitola obsahuje informace a jednoduchý návod na obsluhu uživatelského rozhraní. Toto uživatelské rozhraní bylo navrženo tak, aby umožnilo jednoduchou obsluhu algoritmů, které byly pro účely této bakalářské práce vypracovány. Základní rozhraní je vyobrazeno na obrázku č. 9.

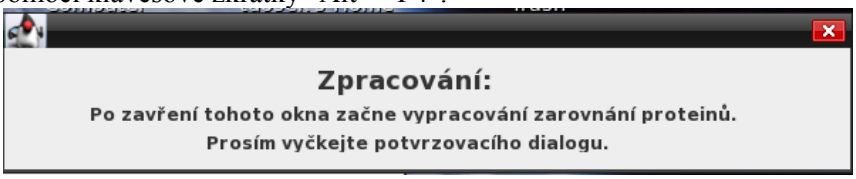


Obrázek 7: Uživatelské rozhraní

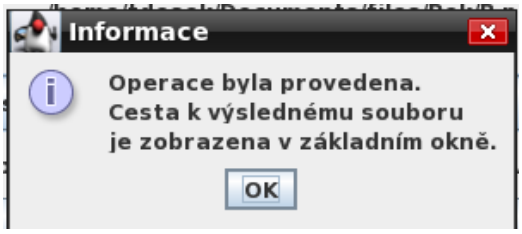
Při vypnutí funkcí spojených s technologií CUDA tlačítko "Načíst soubory a zarovnat pomocí jCUDA" a k němu připojený textový identifikátor cesty nejsou zobrazeny v uživatelském rozhraní.

Toto rozhraní obsahuje tlačítka pro načtení vstupních souborů a dále pro provedení výpočtu pomocí jednoho z prostředků popsaného v této práci. Lze také zvolit způsob zarovnání proteinů. Po kliknutí na jedno z tlačítek "Načíst soubory a..." se zobrazí dialog (Obrázek 10) varující uživatele, že započne samotný výpočet a že má uživatel vyčkat potvrzujícího dialogu (Obrázek 11). Po zobrazení potvrzovacího dialogu je také uživateli zobrazena cesta k souboru s optimálním zarovnáním, který byl programem vytvořen. Po úspěšném provedení výpočtu je také v prostředí Netbeans, případně standardním systémovém výstupu, zobrazen čas trvání výpočtu.

Uživatelské rozhraní může být ukončeno buď kliknutím na "křížek" v pravém horním rohu, nebo pomocí klávesové zkratky "Alt"+"F4".



Obrázek 8: Varovný dialog



Obrázek 9: Potvrzující dialog

8.1.3. Pro programátory

Pro osoby, které by případně mnou vytvořený program nějakým způsobem upravit jsem připravil dokumentaci veškerých mnou naprogramovaných tříd, v anglickém jazyce, vygenerovanou pomocí technologie JavaDoc.
Tato dokumentace je dostupná v ...ProteinAligner/dist/javadoc/index.html.

Veškeré mnou vytvořené třídy jsou popsány tak, aby případný zájemce snadno pochopil již vytvořený zdrojový kód a mohl jej libovolně upravovat. Zdrojový kód také obsahuje zpřesňující poznámky, které nejsou zahrnuty ve výše zmíněné dokumentaci, mouhou být však užitečné pro případné editace.

8.2. Seznam souborů na přiloženém CD médiu

Veškeré součásti použité při programování, tj. zdrojový kód všech použitých tříd, grafický kernel a zkompilované verze programu jsou dostupné ve složce .../**ProteinAligner**. V samotné složce nalezneme soubory pro kompilaci programu ze zdrojového kódu pomocí některého z kompilátorů pro jazyk Java, např. automatizovaný kopilátor **ant**, který jsem použil i já pro kompilaci programu pro operační systémy Windows (architektura 64 bitů) a Linux (taktéž 64 bitové architektury).

Složka **src** obsahuje zdrojový kód samotného programu, tedy jednotlivé třídy použité při vývoji programu.

Složka **dist** obsahuje již zkompilované Java archivy (soubory typu.jar), pro výše uvedené architektury a také dokumentaci programu a všech jeho tříd vygenerovanou technologií JavaDoc.

Složka kernels pak obsahuje finální verzi grafického kernelu, který provádí výpočet na grafické kartě. Tento kernel je ve formátu **.cu**, což je prostý textový soubor, obsahující zdrojový kód napsaný v jazyce CUDA C++. Tento soubor je až za běhu programu automaticky kompilován do souboru **.cubin**, který obsahuje již výše zmíněný zdrojový kód upravený do instrukcí spustitelných na grafické kartě.

Podadresář **files** v adresáři programu obsahuje vzorky používaných proteinů. U těchto vzorků je nutné brát v potaz, že se nejedná o reálné proteiny, ale pouze o automaticky vygenerované .pdb soubory, které obsahují např. podobné vzorce vnitřních struktur, atd. Tyto vzorky byly použity k základnímu ověření korektnosti sestavených algoritmů.

V adresáři **txt**, přiloženého média, pak lze nalézt samotný text této bakalářské práce a to jak ve formátu vhodné pro čtení (**.pdf**), tak i původní zdrojový soubor ve formátu **.doc**.