VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

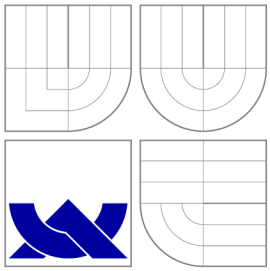
AKCELERACE ALGORITMŮ PRO HLEDÁNÍ TRIPLEXŮ V DNA SEKVENCÍCH

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

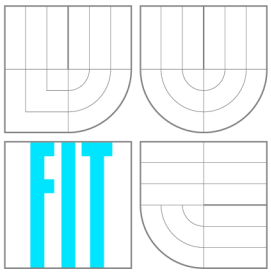
AUTOR PRÁCE
AUTHOR

MICHAŁ WEISER

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

AKCELERACE ALGORITMŮ PRO HLEDÁNÍ TRIPLEXŮ V DNA SEKVENCÍCH

ACCELERATION OF ALGORITHMS FOR TRIPLEX DETECTION IN DNA SEQUENCES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

MICHAŁ WEISER

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MARTÍNEK TOMÁŠ, Ph.D.

BRNO 2012

Abstrakt

Triplexní formy DNA mají vliv na některé základní buněčné funkce. Informace o jejich umístění v genomech a všech vlivech na funkce organismu přesto nejsou známy. Většina algoritmů pro hledání triplexů neuvažuje důležité vlastnosti a možnost výskytu chyb. Komplexní algoritmy jsou velmi výpočetně náročné. Tato práce navrhuje způsob urychlení algoritmu, který při hledání triplexů uvažuje všechny známé informace. Akcelerací algoritmu pomocí paralelního zpracování technologií CUDA bylo dosaženo až 50-ti násobného zrychlení.

Abstract

Triplex forms of DNA act as main factors of some important cell functions. However, their positions within genome and their effect on cell functions are not known well. Triplex search algorithms often don't consider many of triplexs features and the possibility of occurrence of errors. In the other hand the complexity of full featured algorithms is extremely high. This paper shows the way to speed up the algorithm that considers all known triplex features. Parallel aproach allows due to CUDA technology acceleration up to 50.

Klíčová slova

triplex, DNA, grafické karty, CUDA, akcelerace

Keywords

triplex, DNA, graphic cards, CUDA, acceleration

Citace

Michał Weiser: Akcelerace algoritmů pro hledání triplexů v DNA sekvencích, diplomová práce, Brno, FIT VUT v Brně, 2012

Akcelerace algoritmů pro hledání triplexů v DNA sekvencích

Prohlášení

Prohlašuji, že jsem tento semestrální projekt vypracoval samostatně pod vedením pana Ing. Tomáše Martínka, Ph. D., dbal jsem jeho pokynům a snažil se naplnit zadání ve všech jeho bodech.

.....
Michał Weiser
22. května 2012

Poděkování

Děkuji za pomoc a odbornou asistenci vedoucímu práce, panu, Ing. Tomáši Martínkovi, Ph. D. Velice si vážím času, který mi věnoval při práci na tomto projektu.

Obsah

Kapitola 1

Úvod

DNA je nositelkou genetické informace [3] a jsou jí tvořeny geny všech organismů s výjimkou některých nebuněčných. Důležitost DNA a genů obecně je tedy nepopiratelná. Teorie o jejich rolích je dokonce v některých případech upřednostňují před jedinci i celými druhy. Přesto, že jsou tyto prvky naprostými základy pro udržení živého organismu a tedy života, máme o nich málo informací. V roce 1953 byla určena struktura DNA [28] a následně způsob její replikace, přesto dodnes neznáme majoritní část významu informací, které kóduje. Každý postup v této oblasti je oceňován a považován za velmi významný. Proto je důležité pracovat na zrychlení existujících algoritmů pomocí dostupné techniky a urychlit tak poznání zbylých funkcí DNA a jejích různých forem, jako jsou například triplexy. Ve zkoumání triplexních forem existují dva specifické trendy. Jeden se zabývá aplikací genové terapie, tedy ovlivněním vrozených genetických vad. Aplikací třetího vlákna při genové expresi je možné dosáhnout zajímavých výsledků v modifikaci DNA a v budoucnu tak bude možné léčit dnes nevyléčitelné nemoci a dysfunkce. Druhým z případů, ve kterých je triplexy nutné zkoumat, je jejich samovolný vznik v organismech. Každý takový vznik může ovlivnit základní funkce, jakými jsou replikace DNA či exprese genů, které mají vliv na základní funkčnost buňky a tedy celého organismu. Zmíněné vlastnosti této alternativní formy DNA nasvědčují její důležitosti a motivují k jejímu hledání.

Dostupné algoritmy pro hledání triplexních forem ve většině případů zkoumají sekvence heuristickým přístupem. Přesnost výsledků hledání je tedy závislá na konkrétním nastavení. Existují také sekvenční algoritmy, které přesně procházejí vstupní sekvence po nukleotidech a jsou tak schopné uvažovat i nepřesnosti ve stavbě triplexů a jiné vlastnosti přispívající k jejich vzniku. Náročnost těchto algoritmů značně omezuje možnost využití na dlouhých vstupních sekvencích z důvodu doby zpracování.

Paralelizace, je jedním ze způsobů, kterým lze vyhledávání urychlit. Na standardním PC je možné dnes paralelizaci provést pomocí rozdělení úlohy mezi jednotlivá jádra CPU nebo GPU. Proto je vhodné využít tento hardware, který umožňuje několikanásobnou akceleraci běhu programů.

Cílem této práce byl návrh a realizace paralelizace existujícího algoritmu pro hledání triplexů v DNA. Pro výběr vhodného algoritmu a pochopení problematiky bylo nutné nastudovat detailní informace o triplexech a možnostech jejich hledání. Pro výběr způsobu akcelerace algoritmu bylo nutné získat mnoho informací o dostupných paralelních platformách.

Díky získaným znalostem bylo možné v této práci navrhnout převod existujícího algoritmu, který poskytoval nejlepší možnosti a výsledky v sekvenční verzi, do paralelní verze. Výsledkem práce je paralelní verze algoritmu pro hledání triplexů postavená na platformě

CUDA, na které byla také implementována. Výsledné řešení dosahuje až 50-ti násobného zrychlení vyhledávání v porovnání se sekvenční verzí.

Práce je rozdělena do několika hlavních kapitol. Kapitola 2 komplexně shrnuje problematiku triplexů. Nachází se zde biologický pohled na DNA a její specifické formy, kterými jsou právě triplexy. Dále je zde přehled dostupných algoritmů pro hledání triplexů a technologií umožňujících paralelizaci.

Kapitola 3 je vlastním návrhem paralelizace vybraného algoritmu a popisem jeho implementace na grafickém multiprocesorovém systému platformy CUDA. Poslední kapitola 4 obsahuje popis způsobů testování, dosažené výsledky, jejich experimentální ohodnocení a možnosti dalšího vývoje projektu. V rámci semestrálního projektu byla vypracována teoretická část práce zaměřená na informace o triplexech a algoritmech vyhledávání, která byla dále rozvedena v diplomové práci.

Kapitola 2

Triplexy, možnosti hledání a paralelní technologie

Tato kapitola vysvětluje pojem triplexu. K jeho úplnému vysvětlení je nutné zmínit několik základních informací o DNA, na které triplexy vznikají. Proto jsou první odstavce věnovány právě této problematice, následovány důkladným popisem triplexů a jejich vlastností. Další část této kapitoly je věnována existujícím algoritmům sloužícím k vyhledávání triplexních forem s důkladným popisem algoritmu vybraného k paralelizaci. Závěrečná část kapitoly je věnována dostupným technologiím pro paralelní zpracování dat.

2.1 DNA a její specifické formy

Oblast biologie zabývající se genetikou má k dispozici stále více biologických dat, díky zlepšujícím se technologiím sekvenování DNA. Přesto se jedná o oblast, ve které je prozkoumán jen zlomek nashromážděných informací. DNA je nositelkou genetické informace, zaručuje dědičnost a evoluci. Její poškození je přímou příčinou rozličných chorob a její fungování je životně důležité pro organismus. Proto je důležité poznat co největší procento jejích úseků a definovat jejich vliv na jednotlivé funkce a vlastnosti organismu.

DNA

Následující část práce je zaměřena na strukturu DNA a její funkce. Jsou zde popsány důležité vlastnosti, které je nutné zmínit pro úplnost vysvětlení následující části, zabývající se její triplexní formou. Obsah této kapitoly je sestaven z kombinace informací získaných z následujících zdrojů [25, 4, 3].

Struktura

DNA, stejně jako RNA, patří do skupiny nukleových kyselin. Základním stavebním prvkem deoxyribonukleové kyseliny jsou čtyři typy nukleotidů, spojené do lineárního řetězce ve specifickém pořadí. Každý nukleotid se skládá z deoxyribózy, fosfátu a organické báze. DNA je tvořena čtyřmi typy nukleotidů, v závislosti na organické bázi: adenin, cytosin, guanin nebo thymin. Organické báze dělíme na dvě hlavní skupiny:

- Purinové báze - adenin, guanin (deriváty purinu)
- Pyrimidinové báze - cytosin, thymin (deriváty pyrimidinu)

Obrázek 2.1: Purinové báze: adenin (vlevo), guanin (vpravo)

Obrázek 2.2: Pyrimidinové báze: cytosin (vlevo), thymin (vpravo)

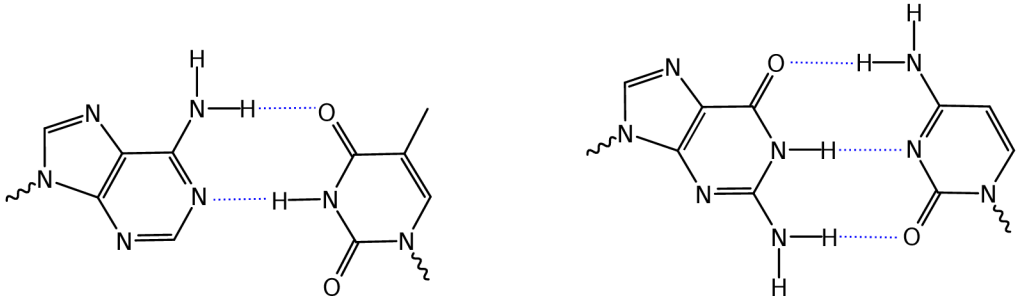
Fosfátová skupina je schopná se pojit s další fosfátovou skupinou na různých pozicích, zejména na 5' a 3' uhlíku. Propojením této skupiny v DNA vzniká lineární řetězec nukleotidů zakončený z jedné strany nukleotidem s volným 5' uhlíkem a z druhé nukleotidem s volným 3' uhlíkem. Takový polynukleotidový řetězec je dle konvence čten ve směru jeho prodlužování, tedy od 5' konce k 3' konci, při čemž jeho zápis v textové podobě znázorňuje souvislý textový řetězec zkratek (prvních písmen) organických bází jednotlivých nukleotidů. Ukázka sekvence DNA lidského genomu GRCh37/hg19:chr21:33,036,341-..., zdroj: [27]:

```
GTGGCGCTGTGCGATCATGGCTGACCTTAGCCTTGACCTCCCAGC...
```

Polynukleotidový řetězec je označován za primární strukturu DNA a tvoří její páteř. Za sekundární strukturu je považována dvoušroubovice, tedy spojení dvou polynukleotidových řetězců pomocí vodíkových můstků. Terciární strukturou je nazývána nadšroubovice DNA neboli *superhelix*. Toto rozpoložení vzniká dalším, kladným či záporným, vinutím dvoušroubovice, díky kterému např. u eukaryotických organismů vznikají útvary jako závit nadšroubovice či solenoidová smyčka. Speciální formou struktury je *relaxovaná DNA*. V této formě se DNA vyskytuje např. v době replikace, jedná se o zrušení nadšroubovicového vinutí. Z pohledu této práce je významná sekundární struktura, ve které mohou vznikat nejen dvoušroubovice, ale také vícenásobná spojení polynukleotidových řetězců a to:

- trojnásobné spojení v rámci jednoho řetězce - *intravláknový triplex*
- trojnásobné spojení v rámci dvou řetězců - *intramolekulární triplex*
- spojení mezi třemi řetězci - *intermolekulární triplex*
- spojení mezi čtyřmi řetězci - *kvadruplex (G-kvartet)*

Tato práce se zabývá pouze druhou variantou, tedy možným trojnásobným spojením polynukleotidových řetězců. Při spojení DNA do formy duplexu se párování bází řídí Watson-Crickovými pravidly komplementarity, tj. adenin se páruje s thyminem (A-T), cytosin se páruje s guaninem (C-G). Jak je vidět na obrázku 2.3, vazba mezi guaninem a cytosinem je



Obrázek 2.3: Watson-Crick párování: adenin - thymin (vlevo), guanin - cytosin (vpravo)

silnější díky přítomnosti třetího vodíkového můstku. U alternativních forem DNA je zapotřebí Hoogsteenova párování (vysvětleno v kapitole 2.1), které umožňuje vznik alternativních struktur, které by s Watson-Crickovým párováním nemohly existovat. Základní jednotkou DNA je dvojice pravotočivě zavinutých řetězců polynukleotidů. Základními vlastnostmi této struktury jsou: stejný počet purinových a pyrimidinových bází - Chargaffovo pravidlo, antiparalelizmus (opačný směr fosfodiesterových vazeb řetězců polynukleotidů: 5'-3' a 3'-5'), vzdálenost 1nm páteře vláken od osy a komplementarita řetězců. Na fyzickou strukturu DNA má vliv také prostředí. Biomakromolekula DNA se v prostředí přizpůsobí podmínkám a nalezne svůj optimální, energeticky nejvhodnější tvar, který je nazýván *konformací*. Vliv na výsledný tvar molekuly má její nukleotidová sekvence, obsah vody v prostředí a iontová síla prostředí (viz [4]).

Triplex

Jak již bylo zmíněno v předchozí kapitole, DNA se může vyskytovat v jiné než běžné dvoušroubovicové formě. Triplexy jsou jednou z alternativních podob deoxyribonukleové kyseliny. Následující kapitola souhrnně vysvětluje jejich stavbu a vlastnosti, tedy informace, které jsou minimem znalostí, nutným k jejich správnému nalezení v nukleotidových sekvencích.

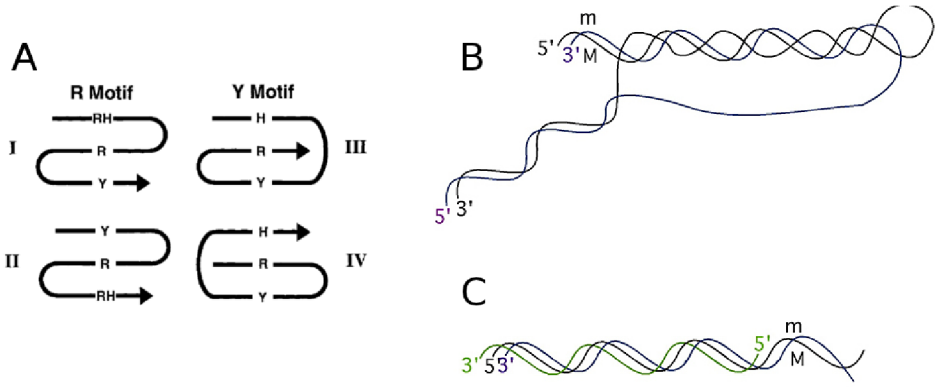
Zájem o jiné, než duplexní, formy DNA vznikl díky poznání, že různé biologické funkce a genetické procesy jsou jimi ve velké míře ovlivněny. Triplexní formy DNA začaly být více zkoumány poté, co byla potvrzena existence podobných struktur v živých organismech. Hlavní funkcí této formy je regulace některých procesů DNA v buňkách, především regulace genové exprese [25]. Vznik těchto struktur může mít velký vliv na fungování celého organismu, proto je důležité tyto struktury zkoumat, poznat co nejlépe situace, ve kterých vznikají a zjistit, jaký vliv mají na své okolí. Důležité je také zmínit, že triplexní formy je schopna dosáhnout i RNA. Stejně tak mohou vznikat kombinované třívláknové struktury skládající se ze dvou vláken DNA a jednoho vlákna RNA [9]. Tato kapitola vychází ze zdrojů [20, 25, 10].

Struktura

Hlavním stavebním elementem triplexní formy DNA je třívláknová spirála skládající se z tripletů, tedy trojic nukleotidů spojených vždy jedním Watson-Crickovým párováním a jedním Hoogsteenovým párováním (více v oddíle 2.1).Triplexy můžeme rozdělit do dvou hlavních skupin v závislosti na složení sekvence a orientaci třetího vlákna:

- pyrimidinový motiv (Y)
- purinový motiv (R)

Pyrimidinový motiv má třetí vlákno složené z pyrimidinových bází. To je paralelně připojeno Hoogsteenovým párováním k duplexu DNA, který je párován Watson-Crickovým párováním. Více na téma párování je zmíněno v samostatném oddílu 2.1. Tato forma triplexu je stabilní v kyselém prostředí a jejím hlavním stavebním pilířem jsou triplety TA*T a CG*C^+. Druhý, tedy purinový, motiv má třetí vlákno homopurinové, ve kterém ve většině případů převládají guaniny. Toto vlákno je připojeno antiparalelně k duplexu pomocí reverzního Hoogsteenova párování. Jeho stabilita není závislá na kyselosti prostředí, ve kterém se vyskytuje.



Obrázek 2.4: Typy triplexů: A) intravláknový, B) intramolekulární, C) intermolekulární, zdroje: A - [10]; B, C - [2]

Rozlišit můžeme dále tři skupiny spojení vláken v závislosti na jejich původu:

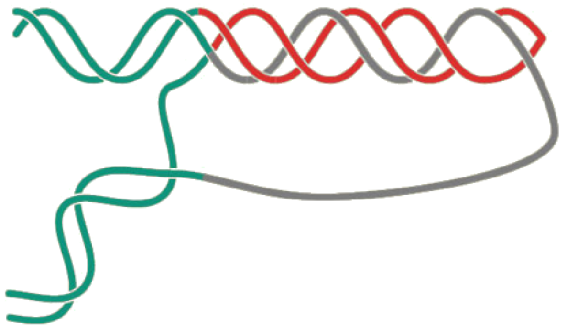
- intramolekulární
- intermolekulární
- intravláknové (intrastrand)

V případě intramolekulárního se jedná o záhyb jednoho z vláken směrem zpět do dvoušroubovice, kde se utvoří třívláknová spirála (triplex), přičemž zbylé komplementární vlákno zůstává volné. Tato forma se vyskytuje v záporném nadšroubovicovém vinutí. Druhá triplexní forma, intermolekulární, předpokládá existenci třetího vlákna pocházejícího z jiného uskupení nukleových kyselin. Obě z těchto variant mohou být jak typu Y, tak typu R. Poslední, třetí z variant, se skládá pouze z jednoho vlákna. Jedná se tedy o vlákno dvojitě složené v rámci samého sebe. Složení se může vyskytovat v několika variantách, jak je vidět na obrázku 2.4A. Byla zkoumána také varianta *in vitro* spojení DNA dvoušroubovice a vlákna RNA, není však známo, zdali toto spojení může figurovat v buňkách a ovlivňovat tak organismus [9]. Bylo potvrzeno, že je možné připojení RNA jako motivu Y při nízkém pH a motiv R je schopná vytvářet pouze DNA [10]. Zkoumané sekvence, které jsou nalezeny, jsou rozdělovány do dvou zásadních skupin:

- formující triplexy
- s potenciálem pro formaci triplexů

Obě z těchto skupin jsou schopny dosáhnout trojšroubovicového uskupení DNA. Přidělení do skupiny je závislé na tripletech, které při formaci triplexu v dané sekvenci vznikají. Některé z tripletů mohou vznikat pouze za specifických chemicko-biologických podmínek. Vliv prostředí je tedy značným faktorem pro stabilitu vzniklých triplexních forem. Za triplexy formující sekvence jsou označovány ty, které mohou samovolně vznikat v buňkách v organismu. Za sekvence s potenciálem pro formaci triplexů jsou označovány ty, které jsou schopné dosáhnout triplexní formy, ale vyžadují pro vznik specifické podmínky, kterých je možné lehce dosáhnout ve zkumavce, ale složitě v buňce organismu.

Tato práce se zabývá intramolekulární formou triplexů, se kterou pracuje většina dostupných algoritmů. Intramolekulární triplexy mohou vznikat dvojí formou. V jednom z případů, který se vyskytuje v buňkách, se jedná o denaturaci homopurinové-homopyrimidinové sekvence, která dovolí ohyb jednoho vlákna směrem zpět do dvoušroubovice. Sekvence, které dovolují tento způsob složení, byly objeveny v savčích genech. Obecně je intramolekulární triplexní forma označována také H-DNA.



Obrázek 2.5: Způsob vinutí vláken H-DNA (intramolekulární triplex), zdroj Mirkin Lab.[1]

Párování

Zásadním faktorem při objevení triplexních forem DNA bylo také zjištění jiných druhů párování bází, Hoogsteenovo párováni a reverzní Hoogsteenovo párováni, které podporují za jistých podmínek připojení třetího vlákna. Toto párování bází nejlépe znázorňuje obrázek 2.6. Jak je z obrázku patrné, existuje 6 základních způsobů napojení třetí báze k existujícímu duplexu. Watson-Crickovo párování není označováno a Hoogstenovo nebo reverzní Hoogsteenovo párování je označováno hvězdičkou. Díky napojení třetího nukleotidu vznikají **triplety** takové jako TA*T, kde TA představuje dvě organické báze duplexu a *T představuje thymin napojený na adenin pomocí Hoogsteenova párování. Stabilita vzniklých forem je závislá na konkrétním tripletu. Některé triplety, jako CG*C^+, vyžadují specifické vlastnosti prostředí. Stabilita spojení jednotlivých nukleotidů má vliv na stabilitu celé triplexní formy DNA. Od párování se také odvíjí směr napojení třetího vlákna (můžeme také ale říci, že se párování odvíjí od směru). V paralelním směru se párování nazývá Hoogsteenovo, v antiparalelním

[1] obrázek přejat z Trufts Mirkin Laboratory - Understanding DNA Structure and Function, [online 30.12.2011]

směru se nazývá reverzní Hoogsteenovo. Dle informací v [14] můžeme dle směru vlákna rozdělit jednotlivé triplety nukleotidů:

- paralelní - *T*AT, T*GC, C*GC, G*GC, G*TA, T*CG*
- antiparalelní - *A*AT, A*GC, T*AT, T*CG, C*AT, G*GC*

Vyhledávání

Způsob hledání sekvencí zodpovědných za formaci triplexních forem je složitý, jelikož, jak vyniká z oddílu 2.1, je formace těchto struktur z velké části závislá na prostředí. Biologické databáze poskytují široké spektrum informací o sekvencích, ale nejsou schopny simulovat podmínky prostředí. Systém takového hledání by navíc byl nesmírně náročný a nejednoznačný. Algoritmy hledající triplexy proto spoléhají na informace o primární struktuře DNA. Zkoumána je délka celé formující sekvence, optimální délka ohybu a také stabilita vzniku. Některé algoritmy spoléhají v prvních krocích pouze na vysoký počet purinů v sekvenci, který je příznakem pro vznik triplexů. Tento přístup se však ukázal být příliš naivním a neefektivním. Pokročilé algoritmy také počítají s nepřesnostmi párování či možností inserce mezery, jelikož bylo potvrzeno, že i neúplné sekvence jsou schopny formovat triplexní struktury. Při testování v laboratorních podmínkách bylo také zjištěno, že neúplné struktury mohou zastávat funkce jejich úplných forem, na jejich funkci tedy nemá případná chyba významný vliv. Z toho vyplývá, že je nutné zaměřit hledání také na struktury, jejichž sekvence může obsahovat chybu, která i tak umožní formaci alternativní struktury.

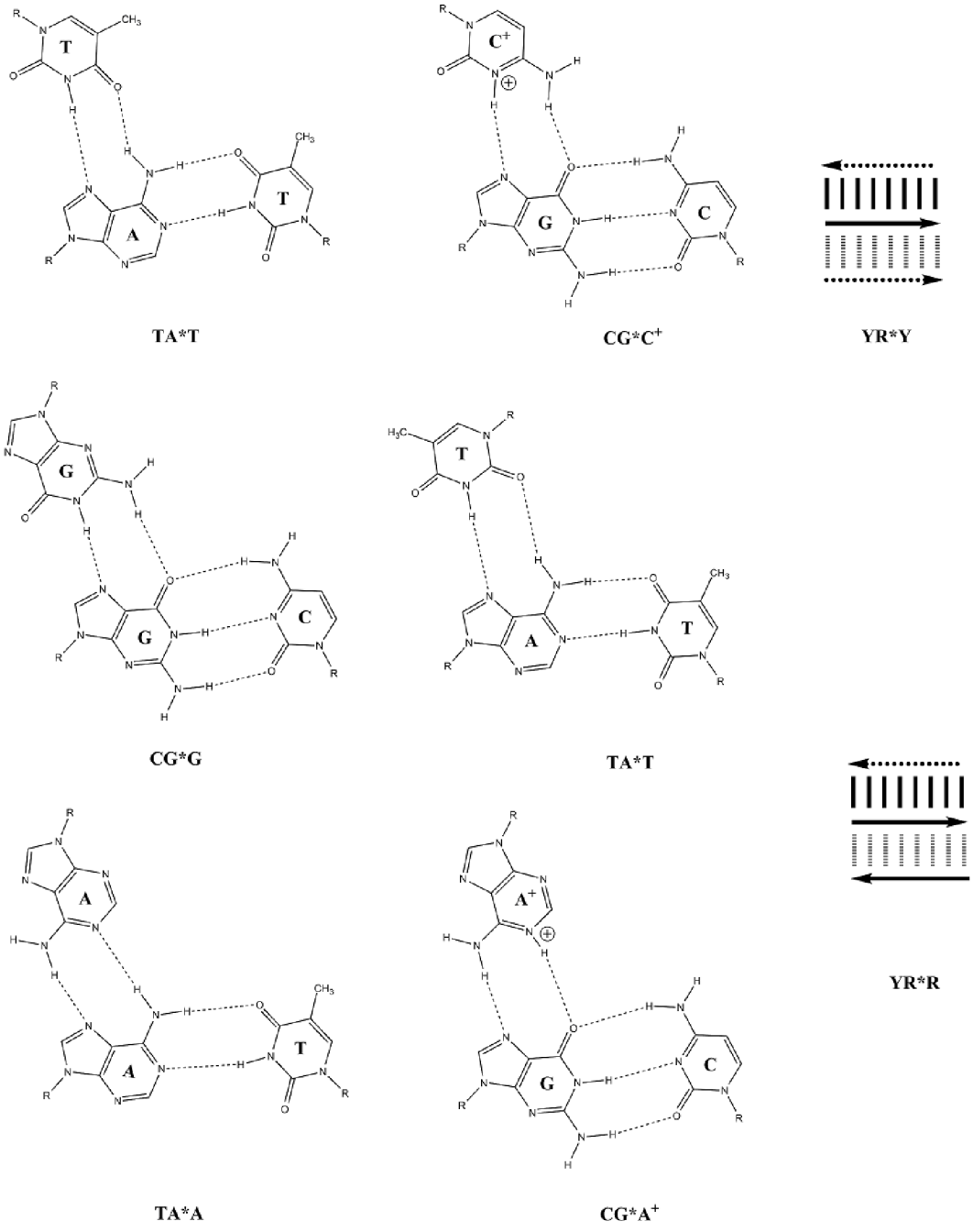
Důvod analýzy triplexů

Jakmile byly objeveny triplexní formy DNA, začal být zkoumán jejich vliv na organismus. V první řadě bylo nutné zjistit podmínky, ve kterých jsou schopny tyto formy vystupovat. Nápomocným v tomto případě bylo zkoumání Hoogsteenových vazeb, které jsou ovlivněny specifickým prostředím. Tím byla funkčnost některých triplexů vyřazena. Pro sestavení následující kapitoly byly použity informace z [10, 5, 11, 2].

Intermolekulární triplexy byly díky zájmu vědců prozkoumány a výsledky jejich zkoumání prokázaly účinnost v oblasti genové terapie (modifikace genů existujících organismů). V rámci této vědy se vyvíjejí léky či malé molekuly, které jsou schopné ovlivnit nežádoucí efekty způsobené vrozenými chybami v DNA. Syntetická DNA v podobě oligonukleotidu se připojí k živé DNA ve specifickém místě, čímž je schopna cíleně ovlivnit genovou expresi. Zkoumání intermolekulárních triplexů by se proto mohlo stát důležitou součástí této vědy budoucnosti.

Zkoumání intramolekulárních forem triplexů se stalo důležitým z důvodu jejich možného výskytu za běžných podmínek v organismech. Ukazatelem funkčnosti se v první řadě stala pozice nalezených sekvencí. Řada z nalezených se nacházela v důležitých místech v genech organismů. Jedním z takových míst jsou promotorové oblasti s vysokým výskytem homopurinových-homopyrimidinových sekvencí. Potencionální formace triplexů v těchto oblastech způsobují citelné následky při expresi genů. Intramolekulární forma je svým vznikem také schopna zabránit vzniku replikační vidličky či způsobit její zánik. Tento jev má za následek ovlivnění replikace DNA, které je základem tvorby nových buněk. U stejného procesu může díky triplexům docházet také k inhibici vlákna, která zastaví vlastní proces replikace.

[2]obrázek přejat z Wikimedia Commons - Hoogsteen, [online 30.12.2011]

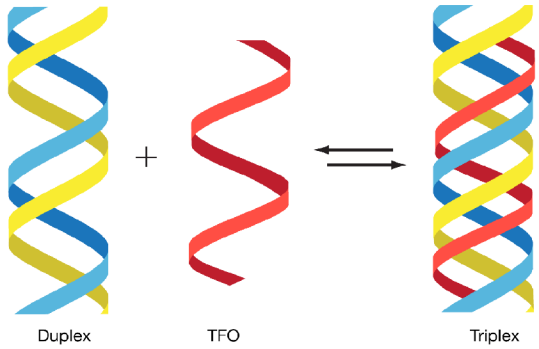


Obrázek 2.6: Hoogsteenovo a reverzní Hoogsteenovo párování bází, zdroj Wikimedia[2]

Intramolekulární DNA může být také využita jako kostra pro opravu proteinů, kdy je využita její funkce ovlivnění genové exprese.

Triplexy mohou být také zodpovědné za vznik mutací. Za specifických podmínek dochází v DNA k rekombinaci, např. při nasvětlování UV zářením. Cíleným nebo náhodným způsobem se mohou při rekombinaci využít oligonukleotidy figurující při vzniku triplexů. Výsledky integrace těchto sekvencí do důležitých úseků DNA mohou být různorodé. V případě cílené rekombinace, která byla zkoumána, se jednalo o léčbu. V opačném případě, tedy náhodném, se může jednat o nevratné poškození struktury DNA a tím i vznik onemocnění jedince.

Některé sekvence jsou odpovědné za tvorbu karcinogenů. Jejich vhodnou modifikací je možné zabránit vzniků těchto nežádoucích buněk. Problémem je přesná modifikace daných genových oblastí. Studium triplexních forem pomáhá při objevování důležitých informací v této oblasti, jelikož umožňuje napojení oligonukleotidů na specifická místa v genech, čímž může způsobit terminaci exprese daného genu. Dalším krokem v tomto vývoji je zkoumání způsobu transportu specifických TFO na přesně určená místa v genech. Zjednodušené schéma aplikace TFO je znázorněno na obrázku 2.7.



Obrázek 2.7: Aplikace TFO - vznik intermolekulárního triplexu, zdroj: [11]

Existuje další spousta potenciálních aplikací triplexů v různých oblastech. Jak vyniká ze článků zmíněných v počáteční části tohoto oddílu je rozdíl ve funkcích intramolekulárních a intermolekulárních forem. V prvním ze zmíněných případů je motivací vyhledávání sekvencí zodpovědných za formace těchto triplexů v genech organismů tak, aby bylo možné předcházet jejich vznikům, a tím i nepříjemným následkům, které z nich vynikají. V případě intermolekulárních triplexů je důležité jejich zkoumání pro potenciální *úpravy* DNA, které mohou přispět k léčbě dnes neléčitelných nemocí. Více na téma aplikací triplexů je možné nalézt v citovaných článcích. Jak je zřejmé z informací v tomto oddíle, je vyhledávání triplexů důležitou úlohou bioinformatiky. Každé zrychlení vyhledávání může přispět v budoucnu k velkým pokrokům v oblasti biologie a medicíny.

2.2 Existující řešení pro hledání triplexů

Tato kapitola popisuje jednotlivé dostupné algoritmy pro vyhledávání a analýzu triplexů. Ke každému algoritmu je uveden stručný popis funkčnosti a uplatnění. Poslední oddíl je věnován algoritmu navrženému v [14], který je zde prozkoumán podrobněji. Tento algoritmus byl zvolen pro paralelizaci, proto je mu v práci věnována pozornost a je zde vysvětlen princip výpočtu do detailů potřebných k popisu způsobu paralelizace, která je navržena v kapitole Návrh paralelizace a implementace na platformě CUDA. Zajisté existuje více algoritmů

zkoumajících danou oblast bioinformatiky. V této kapitole jsou uvedeny ty, které byly podloženy odbornými články z databází dostupných pro Fakultu informačních technologií VUT v Brně[3] a všechny další, které bylo možné nalézt ve volně dostupných databázích odborných článků.

Kategorie algoritmů

Všechny níže popsané algoritmy je možné rozdělit dle specifických vlastností do následujících kategorií:

- dle typu vyhledávaných triplexů:
  - intravláknové
  - intramolekulární
  - intermolekulární
- dle použité techniky hledání
  - heuristické
  - přesné
- dle počtu uvažovaných vlastností
  - striktní
  - uvažující chyby

Tato rozdělení jsou na sebe nezávislá, proto každý algoritmus spadá do jedné kategorie každého typu rozdělení.

Dvouúrovňové vyhledávání triplexů

Algoritmus navržený v bakalářské práci [13] slouží k vyhledávání intramolekulární formy triplexů vniklých zahnutím jednoho vlákna směrem zpět do dvoušroubovice. K analýze vstupní sekvence používá dvouúrovňové prozkoumávání oblasti. V první úrovni kontroly je řetězec prohledáván tak, aby sekvence nukleotidů byla vhodná pro vznik triplexů a současně je kontrolována uživatelem zadaná chyba (počet chybných nukleotidů v načtené sekvenci). Pokud řetězec splní obě ze zadaných podmínek, je nad ním spuštěna plnohodnotná kontrola zkoumající všechny závislosti nutné pro vznik triplexů. Při porušení kteréhokoliv z pravidel je daná sekvence vyloučena. Pokud jsou všechny podmínky splněny, vyhledá se nejdelší možná podsekvence nukleotidů, na které je možný vznik třívláknové intramolekulární DNA. Délku výsledné sekvence je možné parametricky omezit při spuštění prohledávání. Pro urychlení hledání v dlouhých řetězcích jsou zde použita předpočítaná pole hodnot pro komplementaritu prvků, změnu orientace vlákna a pole s počtem chyb. Reprezentace organických bází je vyvedena v binární formě, což umožňuje využití binárních operátorů pro vytvoření komplementárních řetězců a značné zrychlení při veškerých výpočtech v porovnání s počty nad dekadickou číselnou či abecední reprezentací bází. Program nabízí multiplatformní využití díky jazyku Java a multiprocesorové zpracování díky rozdělení úloh kontroly a vyhledávání mezi více vláken. Algoritmus můžeme tedy označit jako: intramolekulární, striktní, přesný.

[3] zdroj: http://www.fit.vutbr.cz/lib/zdroje.php [online 30.12.2011]

Nástroj pro webové vyhledávání oligonukleotidů formujících triplexy

Algoritmus navržený v [7] týmem Anderson Cancer Center je nástrojem pro vyhledávání oligonukleotidů, které mohou být potenciálně součástí triplexní formy DNA. Tyto, několik jednotlivých až několik set nukleotidů, dlouhé sekvence jsou označovány zkratkou TFO[4]. Tato specifická vlákna se skládají z polypurinových sekvencí bohatých na výskyt dG residuí s inzercí několika málo pyrimidinů. I přesto, že jsou známy tyto vlastnosti, nebyla specifikována přesná kritéria, která by byla schopna s jistotou najít oligonukleotidy vhodné pro tvorbu triplexů. Proto řešení vyvinuté týmem A. C. Center poskytuje možnost nastavení všech různých parametrů tak, aby výsledek co nejvíce odpovídal požadavkům uživatele.

Algoritmus je poskytován jako webový nástroj, postavený na skriptu v jazyce Perl, pro hledání sekvencí, odpovídajících vstupním parametrům zadaných uživatelem. Mezi parametry, které je možné zadat, můžeme nalézt: výběr genu, minimální obsah guaninu v procentech, maximální počet vložených pyrimidinů, rozsah délky vyhledávané sekvence a délku domnělého promotoru vyhledávané sekvence. Vyhledávání probíhá v databázi NCBI[5]. Výsledek je vrácen jako tabulka nalezených oligonukleotidů s popisem parametrů zadávaných na vstupu algoritmu. Každý nalezený výsledek je spojen s odpovídajícími identifikátory do jiných databází (Entrez Gene, GenBank,...), takže je možné rychle získat všechny potřebné informace o nalezené TFO sekvenci. Zařazení algoritmu: intermolekulární, striktní, heristický.

TTS Mapping

TTS Mapping, popsaný v [12], je databází, založenou na webových službách, která slouží k vyhledávání regionů v DNA odpovídajících za formace triplexních struktur (TTSs[6]) a jejich následnou důkladnou anotaci. Cílem autorů bylo vytvořit komplexní nástroj pro analýzu sekvencí v lidském genomu, které mohou za určitých podmínek transformovat do stabilní tříšroubovicové formy DNA. Hlavní výhodou tohoto řešení je způsob anotace výsledků, který využívá jiných databází (UCSC genome browser, Quadruplex a dalších) k zobrazení všech relevantních informací o nalezeném úseku nukleotidů. Díky napojení na tyto databáze se ve výsledcích nacházejí informace o možných formacích G4 motivů, CpG ostrůvků, miRNA prekursorů... na zkoumaném úseku lidského genomu. Výsledkem je vizualizace dané sekvence s patřičnou anotací, statické informace o daném úseku a hlavně také informace o tom, se kterými jinými strukturami se může triplexní forma na daném úseku vyskytovat.

Stejně jako u algoritmů uvedených v předchozích kapitolách je vstup do programu parametrizován podstatnými vlastnostmi budoucích výsledků a tak je uživateli umožněno nalezení přesných výsledků dle jeho potřeby. Výchozí nastavení parametrů je odvozeno od předchozích experimentů autorů, je tedy optimální pro obecné vyhledávání. Autorům se během testování tohoto algoritmu na referenčním lidském genomu (hs18) podařilo nalézt několik důležitých TTSs. Jejich výzkum se zaměřil na nalezení delších unikátních sekvencí. Výsledkem bylo nalezení například TTS v promotorových oblastech a oblastech odpovědných za případný vznik rakovinových onemocnění. Autoři tedy vybízejí k dalšímu zkoumání těchto úseků DNA, které mohou svou jinou formací ovlivňovat mnohé základní životně důležité funkce. Algoritmus můžeme zařadit jako: intramolekulární, přesný, heuristický.

[4] z anglického názvu *Triplex Forming Oligonukleotides*, zdroj [7]

[5] National Center for Biotechnology Information, web: http://www.ncbi.nlm.nih.gov/

[6] z angličtiny: *triplex target DNA sites*

TTS v lidském genomu

Stejně jako TTS Mapping, je algoritmus zmíněný v článku [8], zaměřen na hledání úseků potenciálně zodpovědných za formace triplexů v lidském genomu. Způsob, jakým byly triplexy vyhledávany, zde bohužel není zmíněn. Důležité je, že při vyhledávání nebyly uvažovány inzerce a delece. Směrem, kterým se však tato práce ubírala, byl statisticky model výskytů triplexů založený na zkoumání náhodných nukleotidových sekvencí.

Po prozkoumání lidského genomu na výskyt triplexů autoři zjistili, že se v reálných sekvencích vyskytují regiony (TTS) formující triplexy s větší délkou, než tomu bylo v náhodných sekvencích. V náhodných sekvencích byl zjištěn běžný výskyt TTS sekvencí o délce okolo 20 nukleotidů. V sekvencích lidského genomu byly nalezeny úseky o délce 30 nukleotidů formující téměř ideální triplexy. Ojediněle se zde vyskytovaly i sekvence dlouhé 40 a více nukleotidů formující triplexy, které byly autory práce označeny jako extrémně dlouhé. Z minimálních informací, které byly o způsobu vyhledávání zmíněny, můžeme algoritmus kategorizovat jako: intramolekulární, striktní.

Algoritmus dle Mayo Foundation

Výzkumná skupina pod vedením P. R. Hoyne vytvořila algoritmus pro hledání různých možných druhů intramolekulárních triplexních forem DNA. V práci [9] jsou úseky formující tyto alternativní formy označovány jako PIT[7]. Výsledný algoritmus byl využit pro zkoumání tří bakteriálních genomů: E. coli, Synchocysis sp., H. influenzae.

Vlastní prohledávání je realizováno modifikovanou verzí Palingolu, což je program pro rozpoznávání vzorů[8]. Dvouúrovňový průchod v první řadě využívá *Helix Search* k nalezení dvou typů sekvencí dle určených kriterií. První z hledaných sekvenci musí být možné spojit pomocí Watson-Crickových párovacích pravidel. Druhá sekvence, tvořící třetí vlákno triplexu, musí mít rozložení takové, aby se byla schopna připojit k předchozím vláknům pomocí Hoogsteenova nebo reverzního Hoogsteenova párování bází. V prvním kroku jsou kontrolovány potřebné vlastnosti párování, spodní prahy délek sekvencí a specifické vlastnosti odpovídající zařazení do třídy[9]. V druhém kroku algoritmu se kontrolují specifické vlastnosti triplexních forem jako: minimální délka jednotlivých částí vlákna, kontrola purinových a pyrimidinových domén, kontrola délky alfa a beta smyčky...

Rozdělení algoritmu do těchto dvou částí se výrazně projevilo na rychlosti hledání, kdy v prvním kroku je eliminována většina neodpovídajících sekvencí. Díky zkoumání výše jmenovaných tří organismů byly v článku vyvozeny zajímavé závěry, které však přesahují rozsah této práce, proto zde nejsou uvedeny (výsledky je možné nalézt v již odkazovaném článku [9]). Zařazení algoritmu: intramolekulární + intravláknový, přesný, striktní.

Algoritmus inspirovaný vyhledáváním palindromů

Algoritmus ze článku [14] využívá podobnosti triplexů k palindromům. Při hledání intramolekulárních či intravláknových triplexů musí být vždy třetí vlákno párováno pomocí Hoogsteenova či reverzního Hoogsteenova párování, přičemž se jedná o jeden řetězec nukleotidů párovaný spojený sám se sebou. Specifika vyhledávání jsou tedy velice podobná těm u palindromů. Změněn je způsob kontroly komplementarity bází, který odpovídá tripletům

[7] z angličtiny: *Potential Intrastrand Triplex*

[8] více informací na toto téma se nachází v kapitole *Materials and Methods* v článku [9]

[9] více informací o rozdělení nalezených PIT do tříd se nachází v kapitole *Definition of triplex classes* v článku [9]

uvedeným v tabulce 2.1. Díky využití principů dynamického programování je umožněno hledat i triplexy obsahující inzerce a delece. Zpracování pomocí maticového přístupu, možnost inzercí/delecí a dostupnost informací o způsobu zpracování přispěly k **výběru tohoto algoritmu k paralelizaci**, která je v této práci dále popsána. Dle zmíněných vlastností můžeme algoritmus zařadit do následujících kategorií: intramolekulární, přesný, uvažující chyby. Následuje bližší pohled na fungování algoritmu.

Vstupní data

Na vstupu se nachází sekvence nukleotidů zapsaná dle konvence pomocí jednoznakových zkratek organických bází od 5' konce k 3' konci. Sekvence je zasazena do dvourozměrné matice dynamického programování (DP) tak, že jedna strana reprezentuje vstupní sekvenci a na opačné straně se nachází také tato sekvence, ale v opačné formě, tedy zapsaná od posledního znaku k prvnímu. Při tomto zápisu antidiagonála matice DP přestavuje všechna místa začátků možného třetího řetězce triplexu se sudou délkou spojovací smyčky. Sousedící antidiagonála reprezentuje místa začátků možného třetího řetězce triplexu s lichou délkou spojovací smyčky. Diagonály začínající na jedné z těchto antidiagonál a reprezentují samotné triplexy. Výsledkem použití DP je možnost hledat triplexy s inzercí či delecí. Vlastnost uvažování nesouvislých úseků odlišuje tento algoritmus od všech ostatních. Začlenění těchto dvou chyb do procesu hledání triplexů vyniká z různých studií, ve kterých bylo potvrzeno, že i nepřesné sekvence jsou schopny formovat triplex a poskytovat funkce své plnohodnotné triplexní formy. Tato vlastnost má tedy značný vliv na kvalitu výsledků vyhledávání a značně odlišuje tento algoritmus od ostatních.

Způsob výpočtu

Jak bylo zmíněno, vstupní sekvence je zasazena do matice DP. Stejně jako u ostatních algoritmů vycházejících z DP jsou zde zachovány závislosti hodnot. Každá buňka matice je tedy ovlivněna hodnotou diagonální, horní a levé sousední buňky (viz obr. 2.9 A), ale také hodnotou shody/neshody odpovídajících hodnot vstupní sekvence. Jsou-li znaky na souřadnicích [i,j] buňky, která je zpracovávána, schopny vytvořit triplet, je daná buňka matice hodnocena kladně v závislosti na typu daného tripletu dle tabulky 2.1, v opačném případě daná dvojice penalizována.

Výsledná hodnota nové buňky matice DP je maximální hodnotou ze tří možných situací: pokračování triplexu, inzerce, delece. Těmto situacím odpovídají tři směry průchodu maticí: diagonální, horizontální a vertikální. Do hodnoty skóre triplexu se promítá také příslušnost tripletu, na dané pozici, k izomorfní skupině určené v tabulce 2.1. Změna isomorfní skupiny snižuje hodnocení daného tripletu, jelikož tyto změny přispívají k nestabilitě triplexu.

Hodnotu skóre počítané buňky ovlivňuje také nastavení hodnot v tabulce skóre pro jednotlivé triplety. Předdefinované byly semi-empiricky odvozeny od stability jednotlivých triplexů zkoumaných v organismech tak, aby triplexy nevyžadující speciální podmínky prostředí pro svůj vznik, byly hodnoceny vyšším skóre. Pro zohlednění všech možných složení triplexů, zobrazených na 2.8, je nutné algoritmus aplikovat na vstupní sekvenci 8 krát, vždy se specifickou skórovací maticí, odpovídající danému typu složení.

Hlavní antidiagonála a antidiagonála sousedící směrem k levemu hornímu rohu matice, jsou před začátkem výpočtu naplněny nulovým skóre. Vlastní výpočet začíná na druhé antidiagonále, sousedící s hlavní antidiagonálou (směrem k pravému spodnímu rohu). Je vypočteno nové skóre pro každou buňku zpracovávané antidiagonály, dle závislostí zmíněných výše. Výpočet postupně pokračuje pro každou další antidiagonálu směrem k pravému

Tabulka 2.1: skórovací tabulka

typ triplexu	triplet	skóre	isomorfní skupina
paralelní	T*AT	2	a
	T*GC	1	a
	C*GC	2	a
	G*GC	1	b
	G*TA	2	b
	T*CG	1	b
antiparalelní	A*AT	2	c
	A*GC	1	d
	T*AT	2	c
	T*CG	1	e
	C*AT	1	d
	G*GC	2	e

spodnímu rohu, jak je vidět na obrázku 2.9 B. Jak vyniká z obrázku, je možné parametrem nastavit hraniční hodnotu počtu diagonál, které se budou počítat. Tato hodnota vyznačuje délku potencionálního triplexu.

Triplexy jsou detekovány jako úseky dosahující nejvyššího skóre při zpětném průchodu maticí. K nalezení optimálních výsledku HSS[10] je využívána technika podobná technice používané v programu BLAST. Na začátku je nastavena hranice určující, zda je daný úsek zajímavým z pohledu vyhledávání. Když hodnota zkoumaného úseku dosáhne této hranice je úsek označen jako schopný pro formaci triplexů. Úsek je prodlužován do té doby, než jeho hodnota neklesne pod zvolenou hranici. Tento způsob přístupu dovoluje najít s určitou přesností úseky s největší možnou délkou. Nalezení triplexu s jednou inzercí je znázorněno na obrázku 2.10.

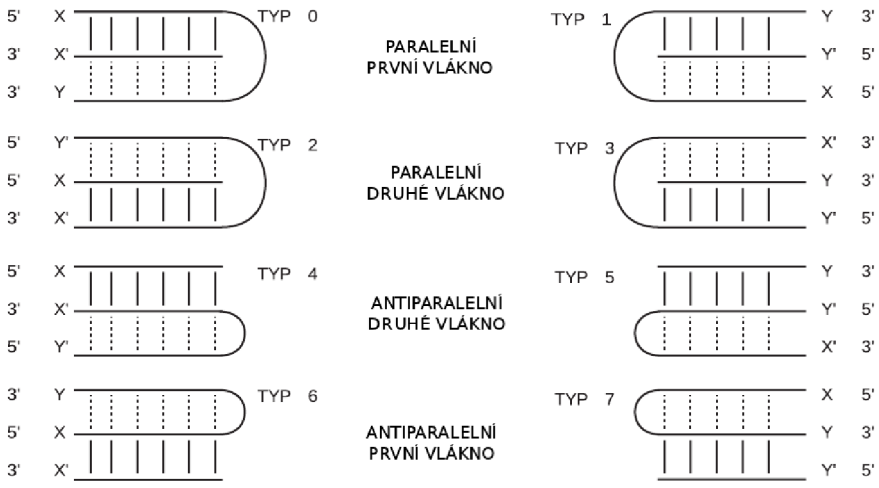
Motivace k převodu do paralelní podoby

Paralelizace algoritmů se stává v posledních letech čím dál, tím důležitějším aspektem návrhu nových systémů. Trend zvyšování výkonu jednotlivých procesorů se ustálil na jisté hranici, která se i tak stále posouvá výše, přičemž ale hlavním cílem vývoje je distribuce výpočtů na více jednotek. Takový postup v mnoha případech zajišťuje nejen nárůst výkonu, ale také snížení energetických nákladů.

Z pohledu návrhu algoritmů však tyto změny znamenají několik zásadních problémů. Časem prověřené postupy algoritmů optimalizovaných pro lineární běh ztrácejí efektivitu. Některé problémy dokonce není možné optimalizovat do paralelní podoby. Někdy je složitost distribuování výpočtů tak náročná, že převyšuje složitost samotného algoritmu. Pokud je možné algoritmus optimálně paralelizovat, jeho efektivita několikanásobně stoupne.

Převod algoritmů do paralelní podoby je úkolem posledních let, jelikož se systémy pro paralelní výpočty staly běžně dostupnými. Běžně dostupné technologie čítají několik stovek výpočetních jader poskytujících výkon na úrovni TFLOP. Tento výkon ještě před pár lety

[10] high score segments - segmenty s potenciálem pro formaci triplexů



Obrázek 2.8: Typy možných intramolekulárních triplexů, zdroj: [14]

poskytovaly pouze superpočítače. Rapidní nárůst dostupného výkonu je přínosem v mnoha oblastech. Jednou z nich je bioinformatika, ve které exponenciálním tempem přibývají biologická data, čehož příkladem jsou biologické databáze jako EMBL, GenBank, DDBJ, PIR, MIPS, UniProt, PDB a další [18].

Paralelní přístup dokáže značně urychlit téměř jakýkoliv výpočet a tím několikanásobně urychlit zkoumání sekvencí nukleotidů či jiných biologických dat. Řetězce mohou čítat až stamilióny prvků. Práce s dlouhými řetězci vyžaduje značného výkonu a paměťových prostředků.

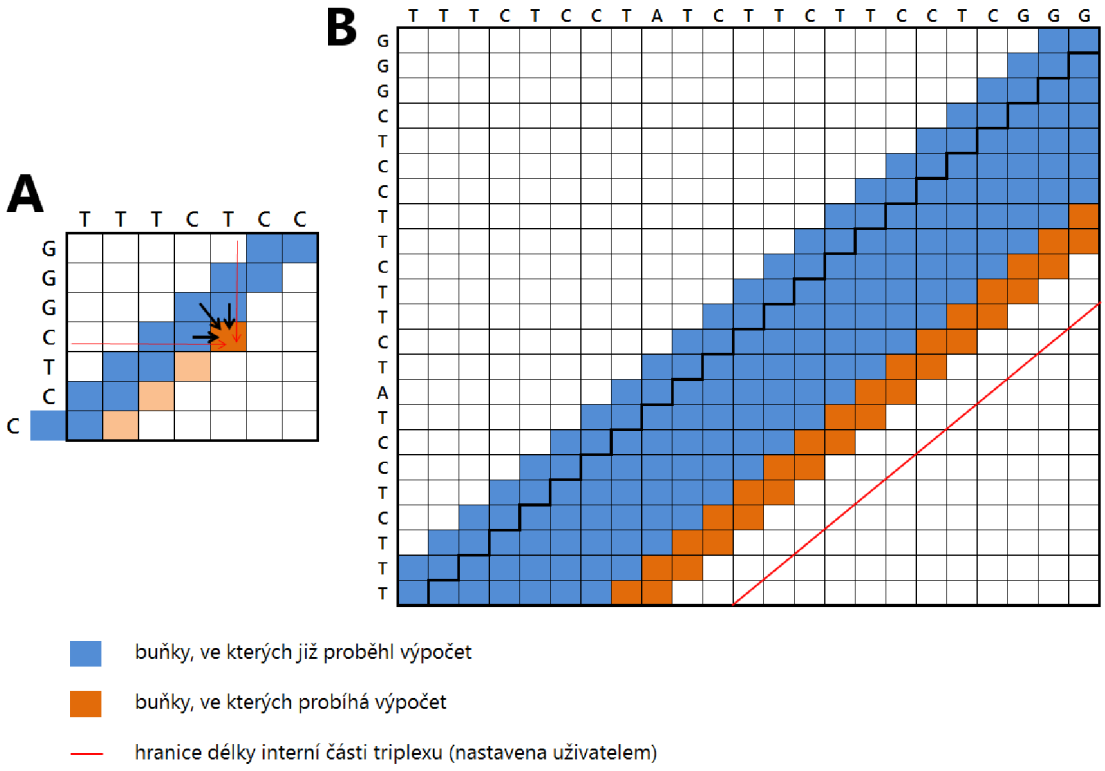
V následující kapitole jsou zmíněny dostupné technologie pro paralelizaci. Veškeré algoritmy zkoumající biologické sekvence jsou nesmírně náročné na výpočetní výkon. Díky využití paralelních systémů je možné zkoumání těchto sekvencí posunout na novou úroveň.

2.3 Technologie paralelního zpracování

V posledních letech se staly více-jádrové procesory dostupnými pro širokou veřejnost. Také v segmentu grafických karet postupně vznikl nový segment technologie pro masivně paralelní výpočty. Tato technologie je výsledkem evoluce původních akcelerátorů grafiky, které potřebovaly stále obecnější výpočty pro fyzikální modely v zobrazovaných efektech. Tato kapitola poskytuje přehled běžně dostupných technologií pro paralelizaci a shrnuje jejich vlastnosti důležité z pohledu návrhu algoritmu.

Více-jádrové procesory - SIMD

Od svého vzniku byly procesory v počítačích typu SISD (Single Instruction Stream, Single Data Stream). Následující odstavec čerpá informace z článku [26]. Optimalizace zpracování instrukcí v logickém sledu vedla k následnému zvyšování taktu procesorů a zefektivnění této činnosti. Díky rychlému vývoji se do popředí výpočetních úkonu dostala multimédia a s nimi spojené výpočty. Úlohy nad multimediálními daty jsou v mnoha případech specifikovány stejnou úlohou nad množstvím dat stejného druhu (např. získání negativu snímku = inverze jednotlivých pixelů). Díky této skutečnosti se do procesorů integrovala podpora technologie SIMD (Single Instruction stream, Multiple Data streams) počínaje technologii

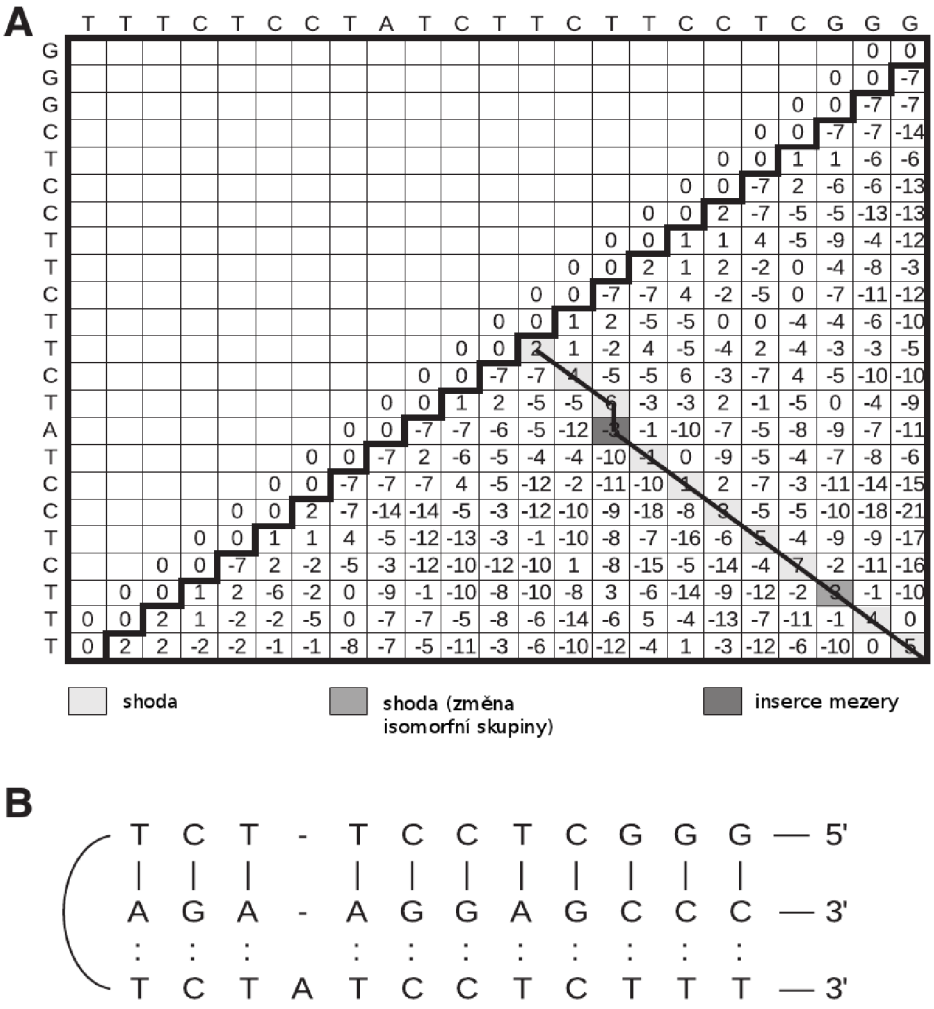


Obrázek 2.9: (A) Závislost buněk na ostatních, (B) Způsob průchodu maticí

MMX a konče dnes dostupnou mnohonásobně výkonnější verzí AVX. Základní jednotkou této technologie, na rozdíl od skaláru pro SISD, je vektor, proto se zpracování dat pomocí instrukcí SIMD označuje za vektorové. Rozdíl mezi SISD a SIMD technologiemi nejlépe vysvětluje obrázek 2.11. Zpracování dat počítá se vstupem hodnot v plovoucí řádové čárce, kde počet zpracovaných instrukcí v jednom cyklu je závislý na zvolené přesnosti těchto dat. Data se *zabalí* do podoby vektoru a jsou zpracována jako jednotka v daném taktu procesoru. Uvážíme-li tedy 128-bitový registr technologie SIMD a přesnost vstupních hodnot 32-bitů, je procesor schopen v jednom taktu zpracovat 4 hodnoty, např. se může jednat o součet hodnot dvou SIMD registrů. Jelikož se jedná o procesorové zpracování, jsou zde dostupné všechny ostatní výhody a optimalizace tohoto řešení. Procesory pro obecné výpočty časem získaly několik úrovní cache (nyní L1 - L3), seřazených dle rychlosti, pro zrychlení přístupů k datům. Optimální je také přístup k pamětem, jelikož se jedná o jednu z nejčastěji využívaných operací při jakýchkoliv výpočtech. K dispozici je pro 64 bitový systém teoreticky až 256 TB virtuálního paměťového prostoru.

Grafické procesory - SIMT

V případě paralelních výpočtů na grafických kartách se hovoří o zkratce GPGPU (General Purpose Graphics Processing Unit). Jedná se o využití výpočetního výkonu grafických procesorů pro zpracování běžných výpočetních úloh s pomocí mnoha jader grafických procesorů v masivně paralelním pojetí. Výhodou toho přístupu je nejen velký počet dostupných jader, ale také vysoká propustnost pamětí. Díky těmto dvěma vlastnostem jsou dnešní nejvýkon-



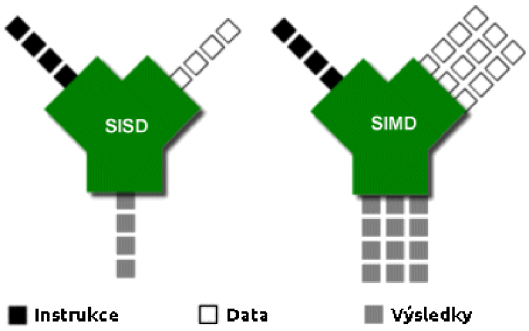
Obrázek 2.10: Ukázka vyhledání triplexů pomocí DP na sekvenci *GGGCTCCTTCTTCTATCCTCTTT*; (A) matice DP, (B) nalezený triplex (dvojtečky představují Hoogsteenovo párování), zdroj: [14]

nější grafické karty schopny poskytnout výkon na úrovni 1TFLOP a jejich vlastnosti se čím dál, tím více optimalizují pro obecné výpočty. Tato skutečnost je také důvodem, proč jsou stále více populární servery založené na výpočetním výkonu několika grafických karet, např. servery společnosti Microway dosahující výkonu okolo 5TFLOP [6]. Jednou z prvních biologických aplikací využívajících GPGPU byla Folding@Home[11]. Výsledky tohoto výzkumu pomáhají při léčbě mnoha kritických chorob, způsobených nesprávným složením proteinů, takových jako rakovina, Alzheimerova choroba, Parkinsonova choroba atp. [24].

Datový paralelizmus

Grafické karty poskytují na rozdíl od běžných procesorů datový paralelizmus. Tento typ paralelizace je velice vhodný pro bioinformatické výpočty, které mohou probíhat nad DNA řetězci dlouhými až několik miliard nukleotidů. Obecně je tento typ výpočtů nazýván SIMT

[11] Folding@Home je sítí počítačů počítajících skládání proteinů podle jejich struktury



Obrázek 2.11: Porovnání SISD a SIMD zpracování dat

(Single Instruction Multiple Threads). Tento termín byl zaveden společností Nvidia u zrodu technologie CUDA, nyní je však obecným termínem pro výpočty na grafických kartách (ve své podstatě se jedná o modifikovaný pohled na technologii SIMD zmíněnou v minulém oddíle 2.3). Někdy jsou tyto výpočty také označovány obecnějším termínem CMT (Coherent multithreading). Následující podkapitoly popisují způsob paralelních výpočtů pomocí SIMT a také hardwarovou implementaci této technologie CUDA. Množství detailů je v této kapitole zmíněno cíleně. Důvodem je způsob návrhu pro GPGPU jednotky, který je velmi závislý na konkrétní architektuře, pro kterou je vyvíjen. Všechny vlastnosti, týkající se technologií SIMT a CUDA, popsané v této kapitole, jsou později zohledněny v kapitole Mapování algoritmu na architekturu CUDA.

Způsob zpracování dat

Běh programu SIMT je rozdělen do vláken. Stejná úloha je spuštěna ve více vláknech tak, že napříč nimi je zpracovávána jedna instrukce, avšak s jinými daty. Vlákna jsou zpracovávána ve warpech, které se přepínají s téměř nulovou latencí. Zkratkou warp se označuje vlákna, která jsou zpracovávána fyzicky v jednom okamžiku. Počet vláken ve warpu je závislý na počtu výpočetních jednotek použité architektury nebo přímo daného modelu grafické karty, běžně je to 32 vláken. Důležitou vlastností je přepínání warpů dle potřeby. Pokud instrukční tok v rámci warpu narazí na instrukci načítání z paměti (časově náročná operace, viz 2.3) je tok přepnut na další z warpů, který může pokračovat. První z warpů je spuštěn zase tehdy, až má připravena data. I když je počet vláken ve warpu nemodifikovatelný a přepínání warpů je řízeno automaticky, je důležité uvažovat při návrhu algoritmu tuto vlastnost architektury.

Technologie SIMT používá streamový programovací model, spuštění kterého na grafické kartě je pojmenováno *stream processing*. *Streamem* jsou datové elementy, které se zpracovávají podobným způsobem. Toto zpracování probíhá ve skupinách výpočetních operací nazývaných **kernel** . Při srovnání se SIMD technologii procesorů CPU, kde je zpracováváno např. 16 hodnot současně, se v případě spuštění kernelu jedná o statisíce až milióny zpracovávaných hodnot. Stream processing může být uniformní, kdy je na kartě spuštěn pouze jeden kernel, či neuniformní, kdy je více spuštěných kernelů, pracujících s různými daty.

Důležitou vlastností je také fakt, že SIMT dokáže pracovat pouze s pamětí grafického adaptéru. Před každým zahájením výpočtů musí být data nakopírována směrem do karty a po skončení výpočtů zase vyčtena zpět. Tato vlastnost je omezující nejen z pohledu časové náročnosti kopírování, ale také z důvodu velikosti dostupné paměti, jelikož se grafické karty

objevují na trhu s maximálně s 2GB dostupné paměti[12]. Oproti zpracování na procesorech je ale výhodou, že se tato paměť používá výhradně pro výpočty.

Pravidla efektivního zpracování

Efektivní fungování algoritmů na SIMT technologiích vyžaduje dodržování několika základních pravidel. I když nové architektury dodávané dnes výrobci potlačují dopady některých omezení, je stále nutné dodržovat jistá pravidla.

Čtení z paměti

Důležitým faktorem efektivity je poměr čtení z globální paměti[13] k výpočetním operacím. V ideálním případě by měl poměr dosahovat hodnoty 50 operací na jedno čtení. Síla výpočtu grafických karet není ve výkonu jednotlivých jader, ale v jejich mnohonásobném zpracování.

Datová lokalita

Čtení dat z paměti probíhá u SIMT po blocích. Je tedy velmi důležité dodržovat ideálně sdružené čtení, kdy jednotlivá vlákna čtou hodnoty uložené v paměti vedle sebe. Pokud za sebou následující vlákna čtou hodnoty uložené vedle sebe v paměti, čtení je sdruženo a efektivita výpočtu balancuje na maximu. Postačí posunutí čtení z paměti o jeden prvek a propustnost čtení klesne o více než polovinu.

Větvení kódu

Jak již bylo zmíněno výše, síla výpočtu GPGPU jednotek je v provádění stejné instrukce na všech vláknech současně. V tomto ohledu jsou problematické pasáže obsahující větvení programu, kde v paralelní podobě je každé zanoření větve prováděno sekvenčně. Znamená to, že pokud je v programu přítomna podmínka typu if–else, která je splněna pouze v některých vláknech, provádění výpočtu proběhne nejprve pro vlákna, která splnila podmínku a až poté pro ostatní. Proto je nutné z programů pro tuto platformu v co největší míře eliminovat větvení kódu.

Další část tohoto oddílu je věnována primárně technologii CUDA, na které je postaveno praktické řešení paralelizace vybraného algoritmu této práce. Na začátku jsou zmíněny technologie OpenCL z důvodu vysvětlení možných rozšíření v kapitole závěru.

OpenCL

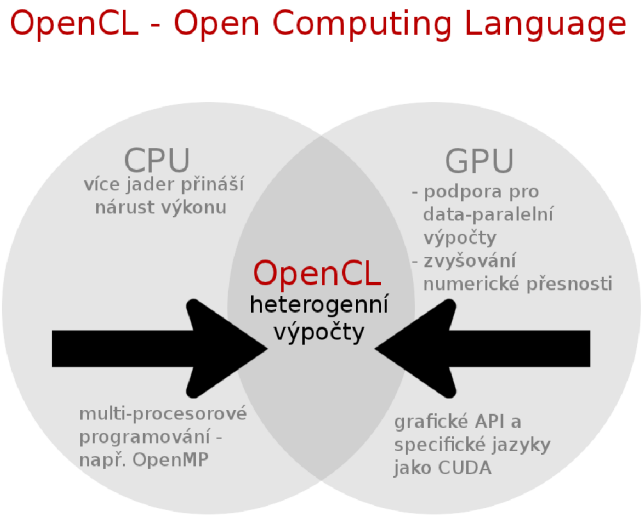
OpenCL (Open Computing Language) je otevřeným frameworkem pro zápis programů běžících na různých více-procesorových platformách, představuje tedy heterogenní způsob přístupu k paralelním výpočtům. Celý projekt je řízen neziskovým konsorciem Khronos Group a je implementován a podporován všemi hlavními hráči na trhu procesorů a grafických karet. V rámci frameworku byl vyvinut univerzální jazyk, založený na C99, sloužící k zápisu kernelů, tedy jader výpočtů. Hlavním cílem OpenCL je zaručit přenositelnost programů mezi platformami a to na úrovni hardwaru i softwaru. Programy psané pro tuto platformu lze tedy bez problémů spouštět na různých typech procesorů a to jak CPU, GPU, tak i např. FPGA.

[12] 4GB pro dvoujádrové modely

[13] viz odstavec Rozdělení pamětí v následující kapitole Nvidia CUDA

Jelikož je OpenCL pouze frameworkem, musejí být při jeho začlenění do programů použity speciální knihovny tzv. wrappery. Aktuálně jsou podporovány všechny z nejčastěji používaných jazyků, jako např. C, C++, Java,... výjimkou není ani JavaScript, pro který je tento standart zapouzdřen pod názvem WebCL. Překlad kernelů může probíhat online[14] i offline způsobem, což umožňuje větší flexibilitu využití.

Program postavený na této technologii je rozdělen do dvou základních částí: výpočetní a ovládací. Výpočetní částí jsou kernely spouštěné přímo na multiprocesorových systémech. Jsou to části, které v původní, neparalelní verzi algoritmu představovaly hlavní[15] smyčky. OpenCL poskytuje nejlepší východisko při programování paralelních algoritmů, jelikož se jedná o univerzální nástroj. Tento fakt je ale také současně nevýhodou z důvodu vysoké složitosti inicializace různorodých zařízení. V každém programu psaném pro OpenCL se tedy nachází mnoho inicializačního kódu, který zvolí HW, na kterém bude realizován vlastní výpočet.



Obrázek 2.12: Schéma OpenCL, kresleno podle informací z Khronos Group

Nvidia CUDA

Základní informace

CUDA (Compute Unified Device Architecture) je řešením technologie SIMT společnosti Nvidia[16]. Jedná se o hardwarové řešení doplněné podporou OpenCL rozhraní a také vlastním CUDA API. Pro výpočetní úlohy jsou nabízeny dvě hardwarové architektury. Nvidia Tesla[17], která je úzce specializovaná pro využití v paralelních výpočtech, je nasazována na serverové úrovni[18]. V článku [6] je vidět trend stavby serverů složených z takových architektur. Proto se Nvidia vydala stejnou cestou a dnes nabízí Tesla GPU SimCluster, který je serverem, určeným pro vysoce náročné datově paralelní výpočty. Výkon těchto stanic

[14] online překlad se provádí za běhu programu, zdrojový kód kernelu je uložen v řetězci ve zvoleném programovacím jazyce

[15] smyčky zabírající nejvíce času za běhu programu

[16] konkurenční společnost AMD nabízí podobný produkt AMD App (zdroj: [1])

[17] v základu Tesla vychází také z architektury Fermi

[18] Nvidia Tesla jsou karty určené speciálně pro obecné výpočty, jejich architektura ale vychází z návrhu původních grafických karet

dosahuje 42TFlopů a jejich cena dosahuje milionů korun. Při testování skládání proteinů (*protein folding*) bylo prokázáno více než desetinásobné zrychlení v porovnání s výpočty na CPU [19].

Architektura Fermi Z pohledu této práce se jeví zajímavějším druhé z hardwarových řešení - Fermi, tedy architektura grafických karet běžně dostupných na trhu počítačů. Jedná se již o pokročilou sérii grafických karet podporujících obecné výpočty. Na Fermi jsou postaveny všechny výrobky série Nvidia GTX 5xx[20] grafických karet této společnosti. V době dokončování této práce však byla nasazena nová platforma Kepler, která měla posunout výpočetní hranice dále, při zachování stejných principů. Jelikož se ale Nvidia u architektury Kepler zaměřila na grafické vlastnosti karet a upustila od výpočetních specifik, karty Fermi v oblasti GPGPU předčí novější Kepler ve všech ohledech. Návrhy pro Fermi by měly fungovat i na novějších architekturách při minimálních úpravách parametrů, proto návrh pro tuto architekturu je smysluplný. Výkon této platformy nedosahuje hodnot na úrovni specializovaných Tesla zařízení, ale je dostupný pro běžné uživatele. V případě portování je výhodou, že algoritmy implementované pro Fermi je možné spouštět také na kartách Tesla, čímž je možné dosáhnout mnohonásobného nárůstu výkonu (vzniká tak možnost nasazení v serverové sféře). Právě dostupnost a rozšířenost této architektury byla důvodem pro její výběr při implementaci výsledného řešení této práce.

Popis architektury

Detailní znalost architektury implementačního prostředí je v případě návrhu pro grafické karty nutností. Každá architektura přináší jistá vylepšení, či omezení. Jak bylo zmíněno v kapitole 2.3, nástupce Fermi - Kepler, je méně přizpůsoben obecným výpočtům, proto by už v tomto případě probíhal návrh ve stejném prostředí mírně odlišným způsobem.

Blokový diagram architektury Fermi je znázorněn na obrázku 2.13. Hlavním prvkem architektury je GigaThread Engine, který plánuje spouštění jednotlivých vláken na procesoru. Celá stavba karet této architektury je rozdělena do čtyř úrovní.

- Graphic processing cluster (GPC)
- Streaming Multiprocessor (SM)
- blok jader[21]
- CUDA jádro

Zatím co první úroveň (GPC) je z pohledu GPGPU nepodstatná, další tři hrají velkou roli. Bližší pohled na stavbu SM se nachází na obr. 2.15 A. Významným prvkem streaming multiprocesorů jsou dvě plánovací jednotky, díky kterým je dosaženo lepšího výkonu při průchodu mnoha podmíněných výrazů (viz 2.3). V případě kdyby pro jeden SP byla dostupná pouze jedna jednotka plánování, by program při podmíněném výrazu využil pouze první řadu CUDA jader a zbylé by zůstaly neaktivní.

Streaming multiprocesor představuje v rámci architektury jednu výpočetní jednotku, v rámci které je omezen počet vláken, které je možné souběžně spustit. Pro architekturu Fermi je limitní hodnotou 1024 vláken na SM. Tato hodnota představuje z pohledu většiny

[19] více informací je možné nalézt na: Nvidia: SimCluster
[20] včetně přidružené série profesionálních karet Quadro a karet pro notebooky
[21] blokem je myšlen počet jader, které sdílí jednu jednotku SFU (Super Function Unit)

Obrázek 2.13: Blokový diagram technologie CUDA na architektuře Nvidia Fermi, zdroj: Benchmark Reviews

navrhovaných algoritmů pouze teoretickou hranici, protože počet vláken je také omezen počtem dostupných registrů a využitím sdílené paměti (viz obr. 2.15 A).

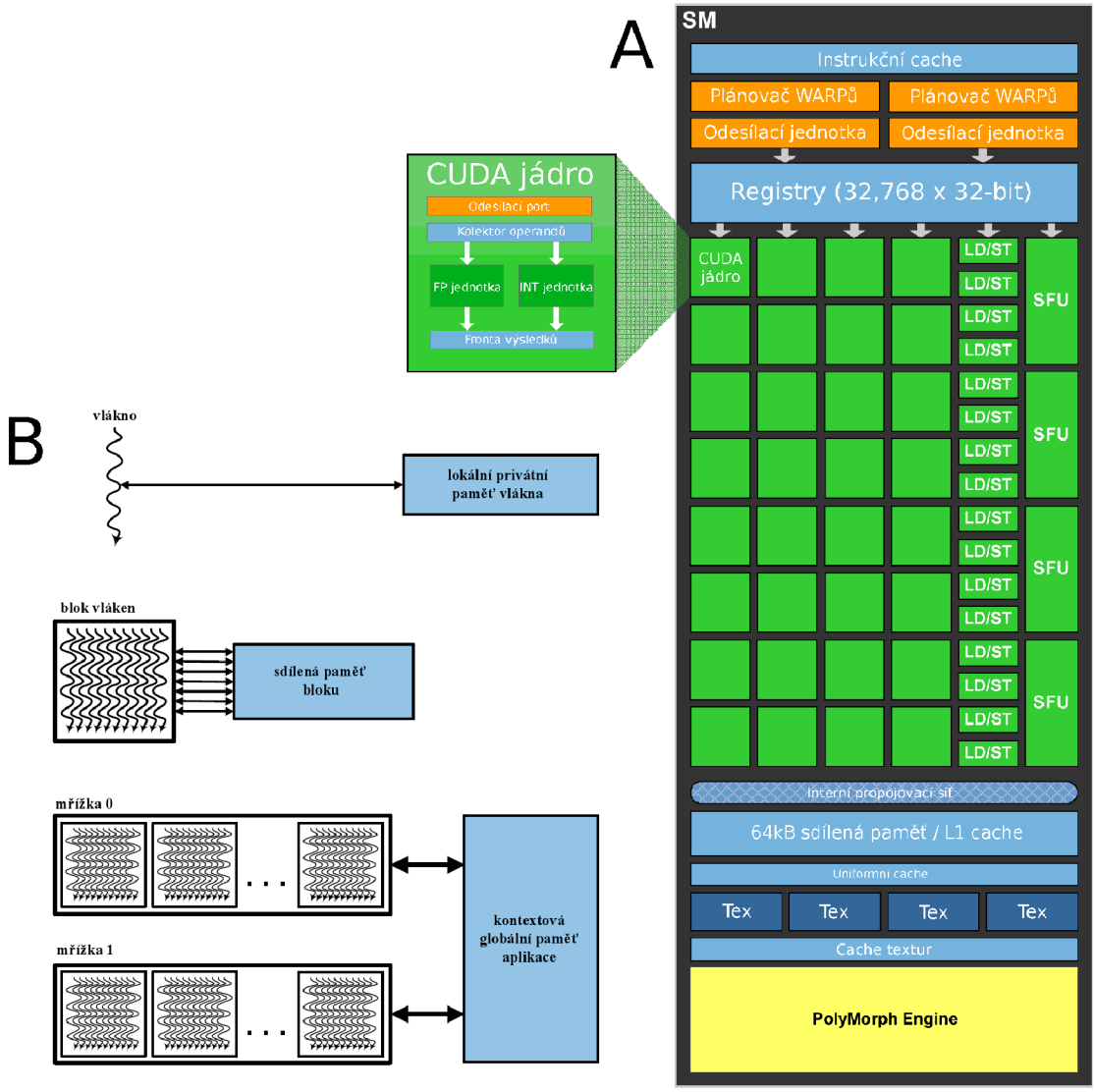
Důležitým aspektem je, že synchronizace vláken je možná pouze na úrovni SM, ne globálně v rámci celé karty (více na toto téma v odstavci 2.3). Tato vlastnost je velice důležitá, jelikož v algoritmech, ve kterých jsou datové závislosti sousedících prvků, lze s jistotou číst výsledky pouze z 1024 hodnot[22]. Jak je patrné z obr. 2.13, na kartě se nachází několik SM které pracují nezávisle na sobě. Počet dostupných SM je ukazatelem výkonnosti daného systému, v případě architektury Fermi je jeho maximální hodnota 16[23].

V každém SM se nachází několik bloků CUDA jader. Rozdělení do bloků je nutné uvažovat z důvodu sdílené SFU jednotky, která provádí matematické operace jako dělení, sinus atd. Pokud některé z CUDA jader provádí některou z těchto funkcí, zbývající jádra jsou pozastavena.

CUDA jádro představuje nejmenší výpočetní jednotku, která slouží k paralelnímu provádění instrukcí. V rámci jednoho SM může být až 48 těchto jader, jejich počet je závislý na konkrétním modelu karty.

[22] 1024 závislých hodnot je myšleno při využití maximálního možného počtu vláken a faktu, že každé vlákno čte jednu hodnotu z globální paměti

[23] jedná se o počet SM příslušných jednomu jádru



Obrázek 2.14: (A) Hierarchie vláken, bloků a mřížek, zdroj: [21]; (B) Diagram streaming multiprocesoru, zdroj: Benchmark Reviews

Rozdělení pamětí

Hierarchii pamětí v rámci architektury znázorňuje obr. 2.15 B. V rámci grafické karty je dostupná dedikovaná paměť (DRAM). Ta je z pohledu GPGPU označována jako globální. Přístup do této paměti může být až 150 krát pomalejší než do ostatních pamětí, proto je vhodné co nejvíce omezit čtení z této paměti. Tato vlastnost se týká všech níže zmíněných pamětí mapovaných v DRAM. Rozdělení:

Samostatné paměti:

- Registry
  - nejrychlejší ze všech pamětí
  - uchovává pouze skalární 32-bitové hodnoty

  - dostupný pouze omezený počet v rámci jednoho SM
  - dostupné pouze v rámci daného vlákna

- Sdílená paměť
  - dostupná všem vláknům v rámci jednoho bloku, tedy SM
  - v případě nekonfliktního přístupu nebo přístupu na stejnou adresu stejně rychlá jako registry
  - sdílí prostor s L1 cache
  - podporuje atomickou inkrementaci proměnných
  - jedná se o cache v režii návrháře algoritmu

Paměti mapované do DRAM:

- Lokální paměť
  - ukládá vektory a struktury vláken
  - dostupná vždy pouze pro konkrétní vlákno
  - ukládána dle potřeby do L2 cache

- Paměť konstant
  - dostupná všem vláknům napříč kartou
  - fixně ukládána do mezipaměti L2
  - omezena na 64kB

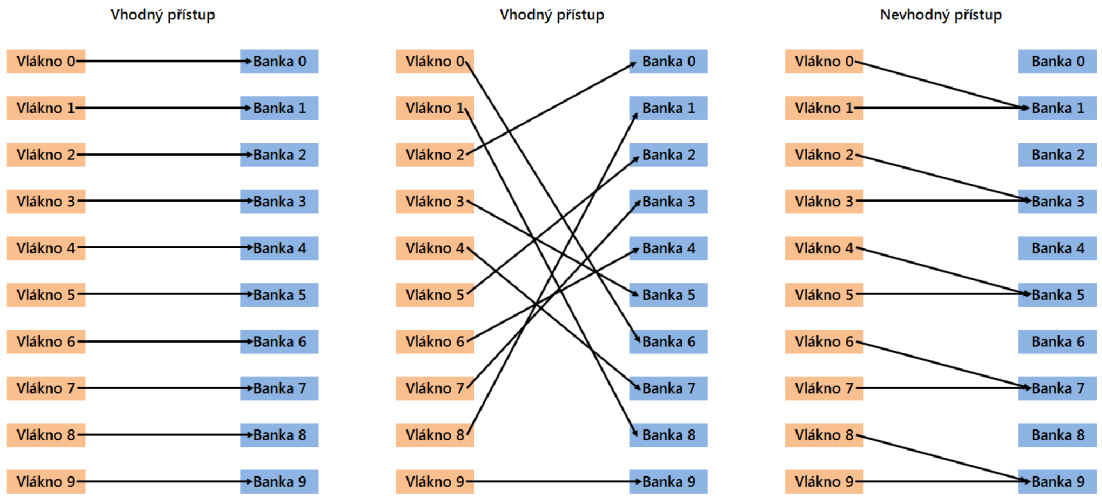
- Globální paměť
  - paměť dostupná všem vláknům napříč kartou
  - umožňuje čtení i zápis
  - ukládána dle potřeby do L2 cache
  - podporuje atomickou inkrementaci proměnných

Důležité jsou také zmíněné mezipaměti L1 a L2. L1 mezipaměť je dostupná pro každý jednotlivý streaming procesor. Její velikost je závislá na tvůrci projektu. Paměťový prostor cache L1 je sdílen se sdílenou pamětí. Celkově je dostupných 64kB přičemž je možné zvolit ze dvou rozdělení: 48kB sdílené paměti / 16kB L1 cache nebo 16kB sdílené paměti / 48kB cache. Volba je závislá na konkrétním algoritmu a jeho paměťových potřebách. L2 cache je globální mezipamětí ve které je fixně udržována konstantní paměť a jsou zde ukládány načítané hodnoty z globální paměti, které jsou odtud dostupné všem vláknům napříč celou kartou.

Dalším velice podstatným faktem, který není na první pohled patrný, je přítomnost bank na vstupu do sdílené paměti každého bloku. Ty jsou zde přítomny, aby mohl být docílen vícenásobný přístup na danou paměť. Počet bank je závislý na kartě, ale zpravidla je násobkem 16-ti[24]. Bitová šířka každé banky je 32-bitů. Data ve sdílené paměti se do bank přiřazuji

[24] pro grafické karty s výpočetní kompatibilitou 2.0 a vyšší je počet bank sdílené paměti shodný s počtem vláken zpracovávaných ve warpu, pro Fermi tedy 32 [22]

automaticky[25]. Přístup k bankám je optimální, pokud vlákna v rámci bloku v jednom čase přistupují každé do jiné banky nebo všechny do jedné banky (využije se funkce broadcast). Nastávají-li sdružené čtení ze stejných bank současně, je k bankám přistupováno sekvenčněa to má za následek snížení propustnosti, a výkonnosti celého programu. Na přístupy k bankám je třeba dbát s ohledem na warpy[26], které se fyzicky vyskytují jako celky na streaming multiprocesorech.



Obrázek 2.15: Přístup vláken k bankám

Nutné je také zmínit, že globální paměťový prostor je možné adresovat pouze lineárně. Při kopírování vícerozměrných polí do paměti grafické karty je vždy nutné provádět mapování do jednorozměrného pole.

Atomické funkce

CUDA API nabízí také využití atomických operací na úrovni sdílené nebo globální paměti. Pomocí těchto operací je možné implementovat zámky, semafory a jiné synchronizační prvky. Na úrovni sdílené paměti je jejich využití možné s minimální ztrátou výkonu. Využití v globální paměti se však silně nedoporučuje [22]. Výkon při použití semaforu na globální úrovni dosáhne stejné hranice výkonnosti, jako program, ve kterém by byl zastaven a znovu spuštěn kernel. Proto se těmto operacím vyhýbá většina programů psaných pro toto prostředí.

Organizace vláken

Organizace vláken v rámci architektury CUDA je hierarchická, několika-vrstevná. Následující odstavce popisují jednotlivé úrovně a jejich vlastnosti.

Vlákno

Kernel [27] je vykonáván pomocí sady souběžných vláken z nichž každé výkonává stejný kód.

[25] každé pole ve sdílené paměti je při přístupu přiřazováno postupně od svého prvního prvku bankám, uvážíme-li 16 bank, pole o velikosti 32 prvků (32-bitů každý), bude přiřazeno: banka[0] = pole[0], banka[1] = pole[], ..., banka[15] = pole[15], banka[0] = pole[16], atd.

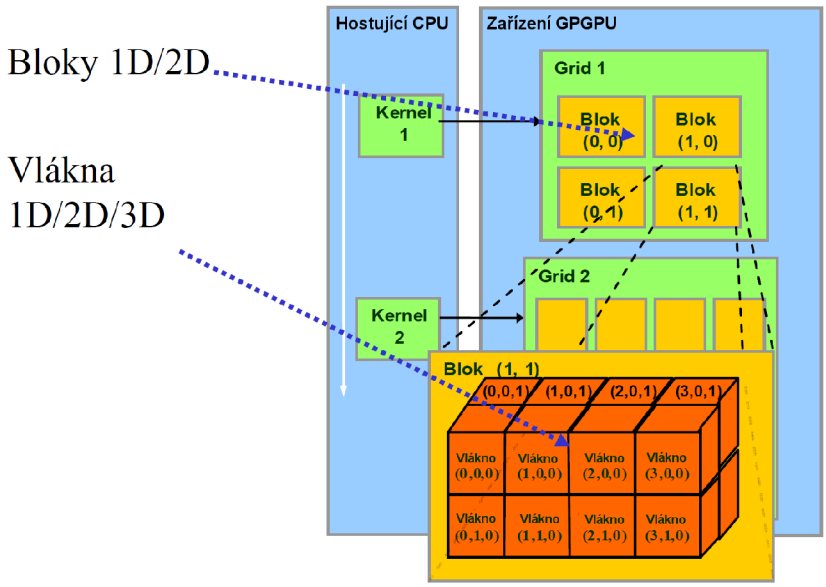
[26] více o warpech v odstavci Warp

[27] viz kapitola Způsob zpracování dat

Každé z těchto vláken je v rámci jedné sady identifikováno specifickým Id (označovaným *threadIdx*), které slouží především k přístupu k datům příslušným danému vláknu.

Blok

Dalším hierarchickým krokem jsou **bloky**, tedy již zmíněné sady vláken. V rámci bloku mohou vlákna spolu komunikovat pomocí sdílené paměti a jejich běh může být synchronizován pomocí bariéry. Každý takový blok dat je na kartě mapován na jeden streaming procesor. V jednom momentě jsou tedy na kartě zpracovávány bloky vláken v **různém** pořadí. Komunikace vláken mezi bloky není možná. Jediným možným způsobem synchronizace všech spuštěných vláken je zastavení kernelu. Organizace vláken v rámci bloku může být jedno-, dvou- či tří-dimenzionální. Id je poté rozšířeno o příslušnou dimenzi (x, y, z), počet indexovaných bloků v jednotlivých dimenzích je 1024, 1024, 64. Je však nutné dodržet omezení pro celkový počet vláken v bloku, který je nastaven na 1024 a dále limitován využíváním sdílených prostředků. Výběr adresování vláken v rámci bloku je závislý na konkrétní úloze a jejím uspořádání dat.



Obrázek 2.16: Organizace vláken v rámci architektury Fermi, zdroj: [23]

Grid

Množina všech bloků se nazývá grid. Grid lze tedy považovat za spuštěný kernel. Kernel je kód, který se zpracovává a grid je množina všech bloků, na kterých jsou vykonávány instrukce kernelu. Bloky v rámci gridu je možné organizovat dle potřeby v jedné až tří dimenzích. Každý z bloků má, stejně jako každé z vláken, svůj jedinečný identifikátor (blockIdx) doplněný příslušnou dimenzí. Počet bloků ve všech třech dimenzích je limitován indexem 65535, přičemž celkový počet bloků je limitován pouze těmito indexy. Velikost indexů je odlišná i na jednotlivých modelech karet. Pravidlem pro Fermi je 65535 v dimenzi x i y.

Warp

Warpy zmíněné v kapitole Způsob zpracování dat nejsou z hlediska organizace nijak ovlivnitelné ani adresovatelné. Je však nutné stále pamatovat o jejich přítomnosti, hlavně při

přistupování do globální a sdílené paměti. Důvodem k jejich uvažováni je fakt, že 32 vláken, které každý warp obsahuje, se zpracovává opravdu souběžně na streaming multiprocesoru. Warp tedy představuje jednotku plánování.

Důsledky rozdělení

Použitá hierarchie představuje výhodu při návrhu algoritmů, ale způsobuje také značné potíže. Na tento fakt je několikrát upozorněno i v doporučeních pro zpracování algoritmů [22]. Jediným a to závažným problémem, který je citelný při návrhu, je indexace. Pokud je v aplikaci nutné přistupovat do několika polí v rámci každého vlákna, je složité dodržet správné indexování. Tato situace se stává složitější, pokud je použito mapování z více rozměrů, do jednoho pole. Přičemž tento způsob mapování je většině aplikací nevyhnutelný z důvodu lineárního paměťového prostoru v globální paměti. Je nutné proto dbát zvýšené opatrnosti při zavádění indexů, neboť přesáhnout hranice polí je velice jednoduché.

Počet zpracovávaných položek

Několikanásobné rozdělení přístupu k vláknům zaručuje možnost specifické indexace mnoha položek. V případě CUDA architektury verze > 2.0 je počet vláken v rámci jednoho bloku omezen teoretickou hodnotou 1024, jak již bylo zmíněno v odstavci Blok
. Počet bloků v rámci gridu je omezen k každé dimenzi hodnotou 65535. Pokud je využito mapování těchto adresací do lineárního prostoru vstupních dat, je možné adresovat až 288217182213504000 prvků. Tato hodnota odpovídá počtu vytvořených vláken. Jelikož je povoleno do globální paměti ukládat pouze jednodimenzionální pole, jeho velikost je omezena touto hodnotou. V případě adresace takového množství prvků vznikají velké problémy se správnou adresací hodnot pro každé vlákno, jak již bylo zmíněno v předchozím odstavci.

Kapitola 3

Návrh paralelizace a implementace na platformě CUDA

3.1 Motivace, výběr algoritmu a platformy

Následující kapitola popisuje návrh paralelního průchodu vybraným algoritmem, motivaci k tomuto návrhu a popis implementace. Cíleně se v teto kapitole prolínají implementační a návrhové prvky. Tento způsob zápisu je zde použit z toho důvodu, že návrh algoritmů pro platformu CUDA je "blízký hardwaru". Návrh probíhá v mnoha iteracích "koncept-zohlednění", kde zohledněním je myšleno přizpůsobení návrhové myšlenky kriteriím platformy. Tato specifická vlastnost návrhu pro platformu CUDA vychází z mnohých omezení oproti zpracování na klasických procesorech, kterou je na této platformě přesnější nazývat vlastnostmi, než omezeními. Proto je nutné dané vlastnosti brát v úvahu přímo při řešení návrhu. Jejich oddělený zápis do více kapitol by vybízel ke zbytečnému častému odkazování v dokumentu a tím k nepotřebnému listování.

Motivace a výběr algoritmu

Z kapitoly Triplexy, možnosti hledání a paralelní technologie vyniká složitost hledání triplexů. Obtížnost tohoto úkolu narůstá s každým dalším uvažovaným parametrem. V případě uvažování delecí/inzercí, což je případ vybraného algoritmu 2.2, složitost dosáhne vyšší úrovně než zarovnání sekvencí. Pro dlouhé nukleotidové sekvence je čas zpracování na běžném stroji dlouhý, již pro sekvence o délce kolem 2000 nukleotidů je roven téměř 20ms. Ve zmíněné kapitole jsou také ale zmíněny běžně dostupné prostředky paralelizace, které pro delší sekvence dokáží být velice účinným urychlením. Při hledání je tedy vhodné využit výkonu dostupného hardwaru. Využití je však vhodné až na dlouhých sekvencích, u kterých je režie spojená s inicializací a nastavováním hardwaru nepatrná.

Jelikož je vybraný algoritmus založen na principu dynamického programování, mělo by se v případě převodu jednat o nárůst výkonu. Proto je zde vysoká motivace pro převod do paralelní podoby. Výsledky převodů jiných biologických algoritmů založených na metodách dynamického programování (Smith–Waterman [16, 17, 15], vyhledávání palindromů[19]) vykazují velký nárůst výkonu, proto byla velká pravděpodobnost podobného výsledku v případě převodu tohoto algoritmu pro vyhledávání triplexů.

3.2 Mapování algoritmu na architekturu CUDA

Následující odstavce popisují mapování algoritmu na platformu CUDA způsobem postupného rozboru algoritmu a jeho přímého převodu do optimální podoby pro implementaci. Důvody k tomuto způsobu zápisu již byly zmíněny v popisu celé kapitoly 3. Pro názornost je dále používána stejná vstupní sekvence do algoritmu, jako v článku 2.2, ve kterém byl algoritmus zveřejněn. Jedná se o sekvenci nukleotidů:

```
TTTCTCCTATCTTCTTCCTCGGG
```

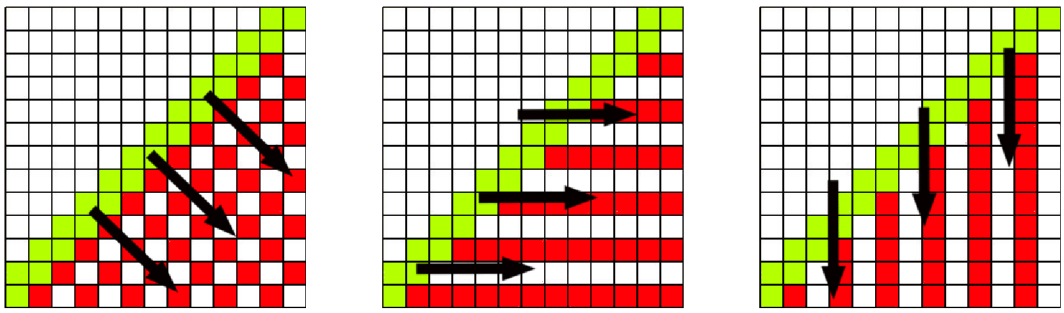
Tato sekvence byla využívána po celou dobu vývoje jako referenční. Aby nedocházelo k nedorozuměním v této kapitole, je algoritmus označován jako:

- sekvenční verze
- paralelní verze

Sekvenčním je označována původní verze algoritmu, navržená v 2.2. Tato verze je v následujících odstavcích z pohledu paralelní verze označována také jako **výpočetní jádro**[1]. Paralelní verzí je algoritmus navržený v této práci.

Způsob paralelního zpracování

Možné směry způsobu paralelizace jsou zobrazeny na obrázku 3.1. Z hlediska implementace bylo již od začátku návrhu pravděpodobné, že některý ze směrů nebude možné realizovat. Každý z určených směrů má své výhody i protiklady. Paralelizace maticových algoritmů je většinou, dle odborných článků, prováděna diagonálně [16, 17]. Z vlastností platformy CUDA však jasně vynikla výhoda pro směr shora dolů (obr. 3.1 vpravo). Na výběru tohoto směru se podílelo několik faktorů spjatých se zvolenou architekturou, které byly omezujícími faktory při dělení běhu algoritmu na části.



Obrázek 3.1: Možné směry paralelizace algoritmu

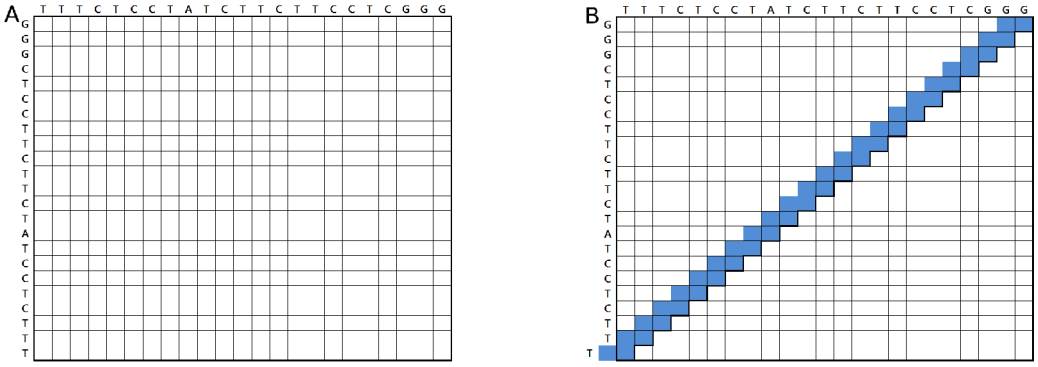
Směr průchodu maticí

Sekvenční verze průchodu

Sekvenční verze algoritmu přistupuje k datům způsobem, jako kdyby vstupní sekvence byla

[1] bude vysvětleno dále

aplikována na hrany matice vodorovně v přímém a svisle v záporném směru, jak je vidět na obrázku 3.2 A. Matice ale v paměti není alokována. Důvodem k tomuto kroku je předpoklad sekvencí o délce několika desítek až stovek milionů nukleotidů. I kdybychom uvažovali uložení pouze jedné 32 bitové hodnoty pro každé pole matice, může velikost matice v paměti dosáhnout řádu několika petabytů[2].



Obrázek 3.2: A: Vstupní sekvence přiřazena do matice, B: Diagonální struktura v rámci matice

Algoritmus proto využívá dvě anti-diagonály, kterými je procházena celá matice. Tento způsob využití je možný díky závislostem, které se ve výpočtu vyskytují. Každá buňka matice je závislá na hodnotě buňky diagonálně vlevo nahoře, buňky vlevo, buňky nad ní a na shodě/neshodě prvků vstupní sekvence na pozicích, na kterých se nachází aktuálně vypočítávané pole matice (Obr. 3.3 A).

Celkový průchod maticí odpovídá diagonálnímu směru směrem k pravému spodnímu rohu matice (obr. 3.2 vlevo). V každém kroku jsou spočítány sekvenčně dvě anti-diagonály.

Implementované řešení závislosti prvků řeší optimálním způsobem tak, jak je znázorněno na obr. 3.3 B. Z paměti je uložena vstupní sekvence pouze jednou. Přístup ve vodorovné pozici je řešen inverzním indexováním do vektoru vstupních dat. Díky tomu je paměťová náročnost algoritmu velice nízká.

Na obr. 3.3 B jsou znázorněny tmavě oranžovou barvou dvě buňky. Z pohledu průchodu matice algoritmem se jedná o stejné buňky v anti-diagonále. Hodnoty, které se v buňce nacházejí se tedy využijí k aktuálnímu výpočtu a poté se přepíši výsledky patřícími buňce, která je aktuálně vypočítávána. Díky tomuto přístupu je možné mít v paměti uloženo pouze *2 * délka vstupu* diagonálních prvků (hodnot buněk).

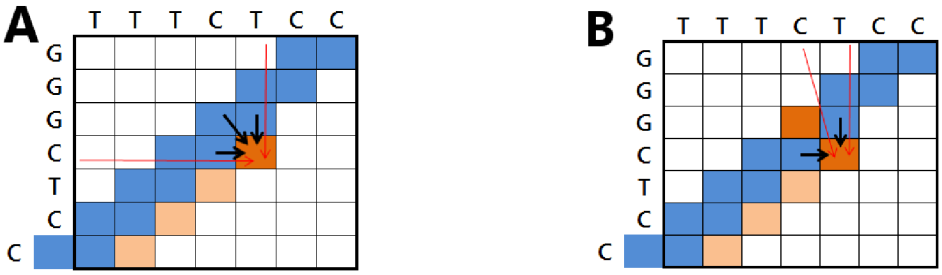
Definice jádra výpočtu

Diagonální prvky jsou nazvány z toho důvodu, že neukládají informace pouze o hodnotě skóre, které získaly, ale také další potřebné hodnoty vlastností nutných pro přesnou detekci triplexů. Důsledkem uložení těchto vlastností velikost diagonálního prvku značně narostla.

Velikost těchto prvků je limitujícím faktorem pro rozdělení algoritmu do několika částí. V případě sekvenčního zpracování je velikost limitována paměťovým limitem, který byl programu přidělen operačním systémem.

V dalších odstavcích bude tato část nazývána **jádrem** výpočtu. Jedná se tedy o část algoritmu, která je schopná bez dalšího rozdělování nalézt triplexy v zadané vstupní sek-

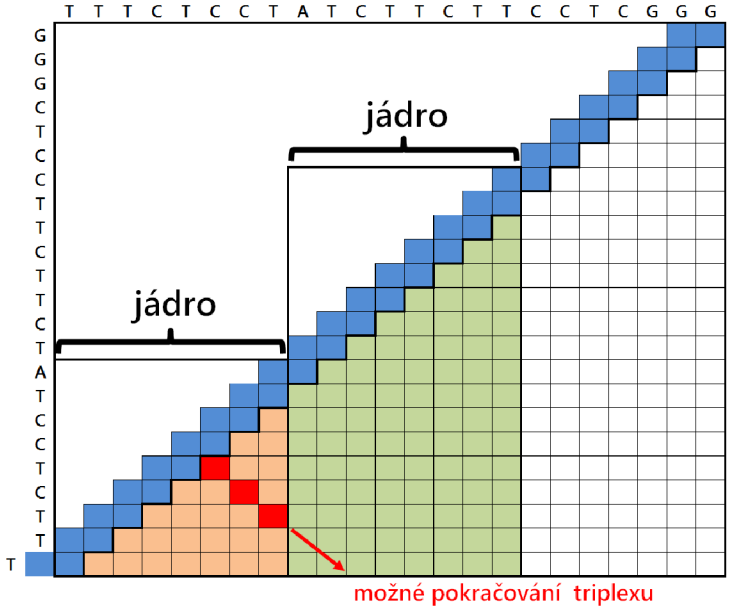
[2]např. pro lidský chromozom 1 hodnota dosahuje 55 petabytů, počítáme-li s přibližnou délkou chromozomu 249 milionů nukleotidů



Obrázek 3.3: A: Závislosti dat, B: Implementační řešení závislostí

venci. Délka sekvence, kterou je **jádro** schopno zpracovat je omezena pamětí dostupnou pro uložení diagonálních prvků. Paměťová náročnost jádra je $2n \times (velikost\ diag.\ prvku) + velik.\ bufferu$[3].

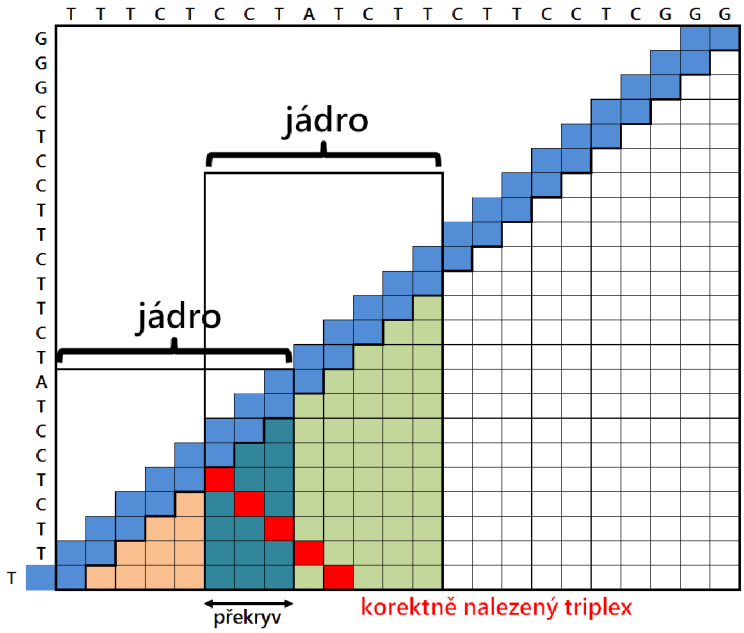
Je-li vstupní sekvence velká tak, že paměťové nároky algoritmu přesahují dostupné zdroje, je nutné běh algoritmu rozdělit na části. Každá část pracuje nezávisle na ostatních na určené části vstupní sekvence. Jelikož jsou hodnoty závislé na sobě, je nutné zaručit, že tripexy, vyskytující se v místech rozdělení jader, budou správně nalezeny. Situaci znázorňuje obrázek 3.4.



Obrázek 3.4: Chybné rozdělení jader výpočtu

Pro správnou detekci všech hledaných triplexů je nutné zavést překrývání výpočtů jednotlivých jader. Velikost překryvu je závislá na délce hledaného triplexu a délce jeho smyčky. Každé následující jádro tedy začíná svůj výpočet na části dat, které již počítalo jádro předchozí z důvodu nalezení triplexů na překryvech. Pokud se v oblasti, kterou počítají dvě jádra, objeví krátký triplex, je identifikován dvakrát a oba výskyty jsou zapsány do výstupních dat. Korektní přístup k použití více jader je znázorněn na obrázku 3.5.

[3]velik. bufferu znamená velikost bufferů pro výpis nalezených triplexů, v případě sekvenční verze je buffer dynamický



Obrázek 3.5: Korektní rozdělení jader výpočtu (překryv je zde pouze názorný, neodpovídá délce hledaného triplexu)

Paralelní verze průchodu

Dle informací z odstavce Organizace vláken kapitoly Nvidia CUDA je zřejmé, že i když architektura CUDA poskytuje možnost spuštění několika milionů vláken, jejich vzájemná komunikace napříč celou architekturou není možná. Synchronizace je možná pouze na úrovni vláken jednoho bloku. Teoreticky je tak možná komunikace maximálně 1024 vláken.

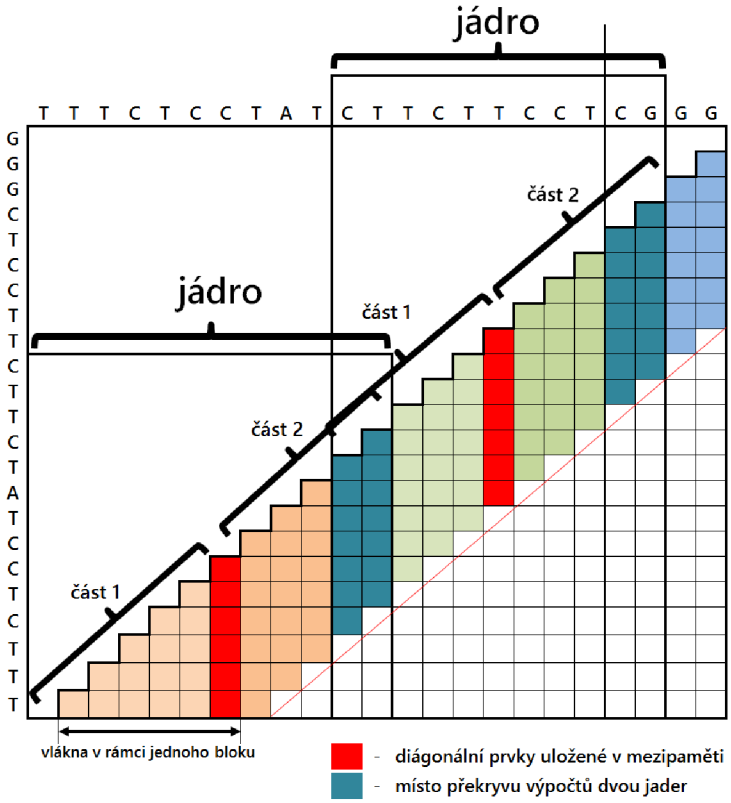
Díky zmíněným závislostem buněk na buňkách okolních je nutné, aby vlákna spolu byla schopna komunikovat z důvodu synchronizace kroků výpočtu. Jedním krokem je v případě sekvenčního algoritmu výpočet hodnot dvou anti-diagonál. V případě paralelní verze se jedná o výpočet jedné anti-diagonály (upřesněno v odstavci Zvolení vertikálního směru průchodu).

Jelikož je počet vláken v bloku omezen dostupnou sdílenou pamětí a počtem dostupných registrů, bylo zřejmé, že při počtu informací nutných k uložení v každé položce diagonálního prvku se počet vláken schopných komunikace sníží minimálně na poloviční hodnotu. V případě 1024 dostupných vláken je počítáno s 8-mi dostupnými registry pro každé vlákno. Složitost algoritmu by ani v případě dostatečné paměti nedovolovala spuštění mnoha vláken z důvodu nedostupných registrů.

Aby výpočet výsledků pro dlouhé sekvence, pro které je tento algoritmus navržen, nebyl zatížen velkým překryvem, bylo nutné zvětšit rozsah hodnot počítaných jedním blokem vláken. Proto vznikla mezipaměť, do které je ukládána hodnota posledního z vláken v bloku. Jedno jádro výpočtu tak je schopno cyklicky spočítat násobky své velikosti. Názorněji je rozdělení zobrazeno na obrázku 3.6.

Adresování bloků a velikost vstupu

Rozdělení na části má také další pozitivní vlastnost, kterou je možnost zpracování většího počtu vstupních dat. Avšak s narůstajícím počtem částí narůstá také režie potřebná k nulování diagonálních prvků před zahájením výpočtu, ale hlavně režie ukládání a načítání diagonálních prvků na hranicích výpočetních částí. Jelikož je operace zápisu a čtení speci-

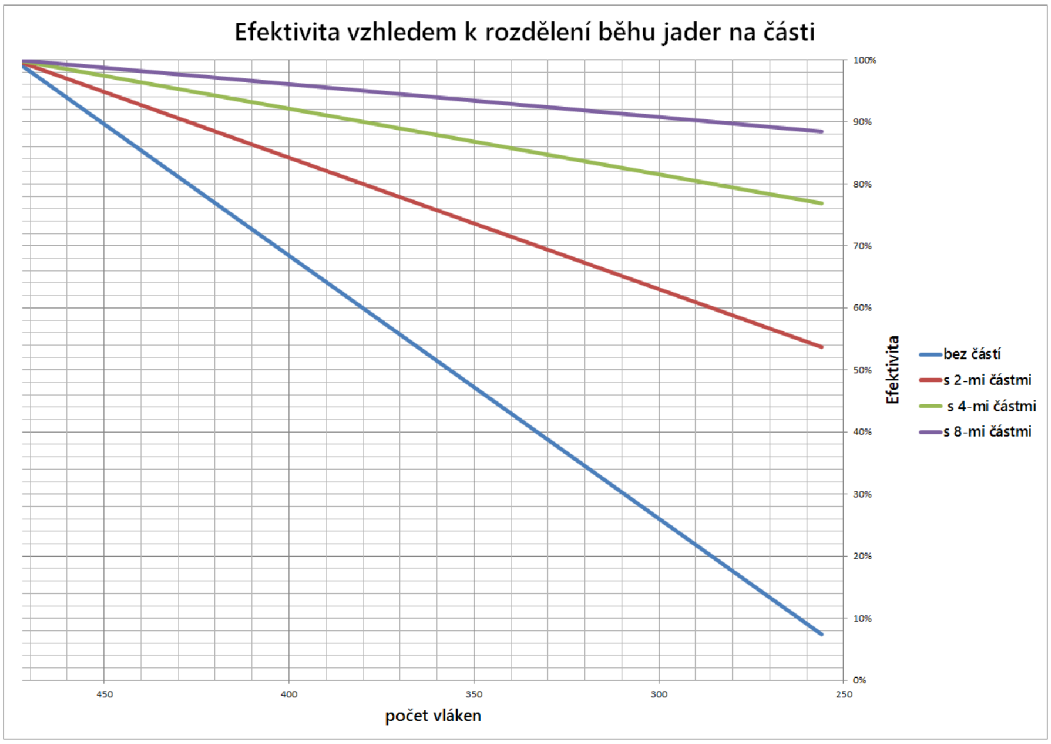


Obrázek 3.6: Rozdělení běhu jader na části

fickou operací pouze pro hraniční vlákna, výpočet na všech ostatních vláknech je v době vybavování operací s mezipamětí hraničního vlákna pozastaven. Důvodem je nutnost synchronizace vláken v každém kroku (výpočet jedné anti-diagonály). Při ukládání diagonální struktury do paměti všechna ostatní vlákna musejí čekat před zahájením dalšího výpočetního cyklu. Na každém stroji, na kterém bude tento algoritmus používán, je vhodné experimentálně nastavit hodnoty počtu částí zpracovávaných jedním blokem v závislosti na délce vyhledávaného triplexu (délka vyhledávaného triplexu ovlivňuje velikost překryvů výpočtů jednotlivých bloků ≈ výpočetních jader).

Algoritmus je na platformě CUDA vždy spouštěn jako třídimenzionální grid bloků. Jak bylo zmíněno v odstavci 2.3 kapitoly Nvidia CUDA, každou dimenzi gridu je možné adresovat do hodnoty 65535. Uvážíme-li 512 vláken v rámci jednoho bloku, je možné bez uvažování překryvů a rozdělení do části zpracovat mírně přes 33 milionů nukleotidů vstupní sekvence. Jelikož je délka chromosomů, které by měly být zpracovávány, v řádech stovek milionů nukleotidů a je nutné uvažovat překryvy, je adresování pomocí jedné dimenze nedostačující.

Pro adresaci gridů je nutné využít mapování třídimenzionálního prostoru bloků do jednorozměrné posloupnosti bloků. Každé vlákno v rámci architektury CUDA má informaci o vlastním *id* v rámci bloku a o počtu vláken v bloku. Stejné informace jsou vláknu dostupné o bloku, ve kterém se nachází, tedy: id bloku v každé dimenzi, velikost dané dimenze bloků v rámci gridu. Adresování tedy využívá systém segment offset, kde segmentem je specifické *id* bloku a offsetem je *id* vlákna v rámci bloku. Specifické id pro segment je možné získat pomocí následujícího vzorce 3.1, kde: *blokId* následované označení dimenze znamená *id* bloku v dané dimenzi, *gridDim* následované označením dimenze znamená počet bloků



Obrázek 3.7: Efektivita výpočtu vzhledem k rozdělení běhu jader na části pro architekturu Fermi

v dané dimenzi gridu.

$$blok\ id = (blokId.z \times gridDim.z \times gridDim.y) + (blokId.y \times gridDim.y) + blokId.x \quad (3.1)$$

Offset, tedy *id* vlákna v rámci bloku, je možné získat přímo. Počet vláken je omezen hodnotou 1024 a adresování poskytuje 3 dimenze, každou o velikosti 1024. Indexovat všechna vlákna v bloku je tedy možné i pouze pomocí jedné dimenze, kterou je v tomto případě dimenze x.

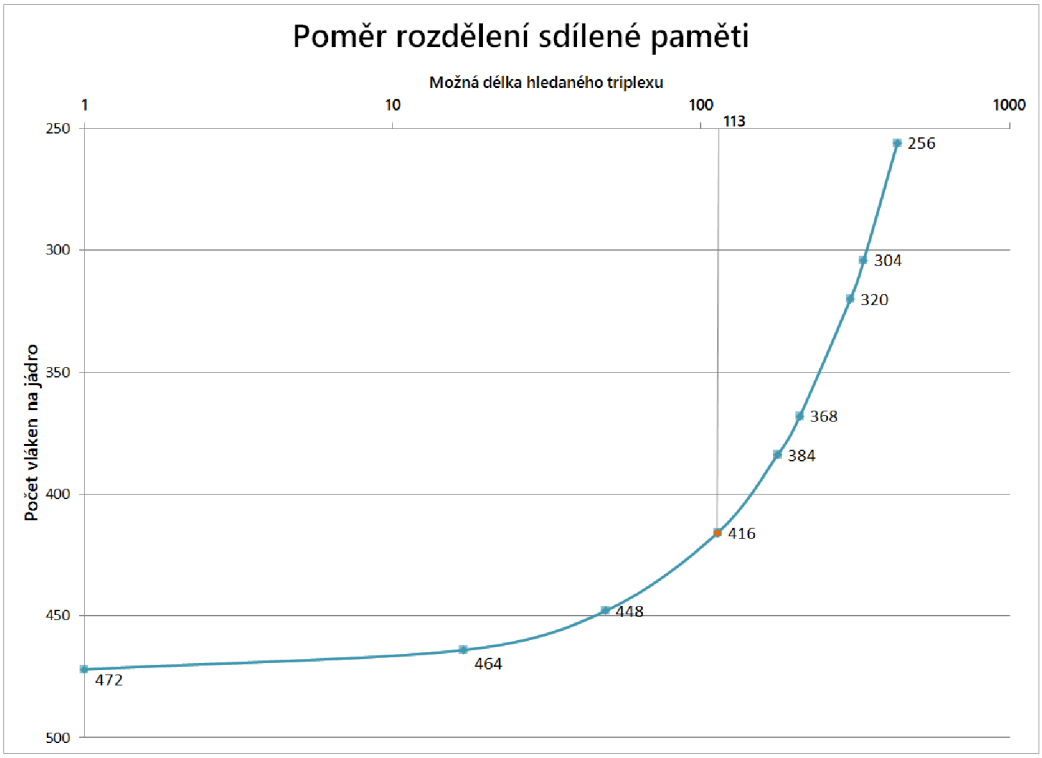
Pro výpočet ideálního počtu bloků v gridu je vhodné využít inteligentní způsob rozložení bloků do dimenzí. Pro tento účel se používá prvočíselný rozklad. Aby efektivita rozkladu nezatěžovala výpočet, je využito Eratosthenovo síto pro získávání prvočísel.

Paralelní jádro výpočtu

Jak vyniká z předchozího odstavce, jádro výpočtu je mapováno na bloky vláken. Faktorem omezujícím velikost jádra je velikost sdílené paměti poskytované na blok. Pro architekturu Fermi je možné zvolit variantu rozdělení interní paměti bloku na 48kB pro sdílenou paměť a 16kB pro L1 cache.

Přístupy do globální paměti jsou velice časově nákladné, proto je vhodné umístit diagonální prvky do sdílené paměti. Velikost diagonálního prvku algoritmu, se zachovanými vlastnostmi uvažovanými v sekvenční verzi algoritmu, je 52 bytů. Pro karty architektury Fermi tedy je možné ve sdílené paměti uložit 495 diagonálních prvků. Tato hodnota je rozdělena v každém jádře mezi prvky diagonál pro výpočet a prvky diagonál pro uložení vypočítaných hodnot v rámci jednotlivých částí.

Velikost paměti potřebná pro uložení je závislá na délce vyhledávaných triplexů[4]. Omezením návrhu je fixní nastavení této paměti. Důvodem k tomuto kroku je náročnost dynamické alokace na platformě CUDA. Experimentální nastavení poměru diagonálních prvků je zvoleno na hodnoty: 416×2 výpočetních prvků, 113 úložných prvků. Nastavení vychází z článku [8], ve kterém byly zkoumány délky nalezených triplexů v lidském genomu. Nejvíce se vyskytovaly triplexy o délce 15. Triplexy o délce větší než 40 nukleotidů byly označeny za extrémně dlouhé. Zvolené hodnoty povolují hledat triplexy do délky 113 nukleotidů bez smyčky. V případě nutnosti hledání delších triplexů je možné parametry upravit. Závislost poměru zvolených vláken na jádro vůči délce hledaného triplexu je znázorněna na obrázku 3.8



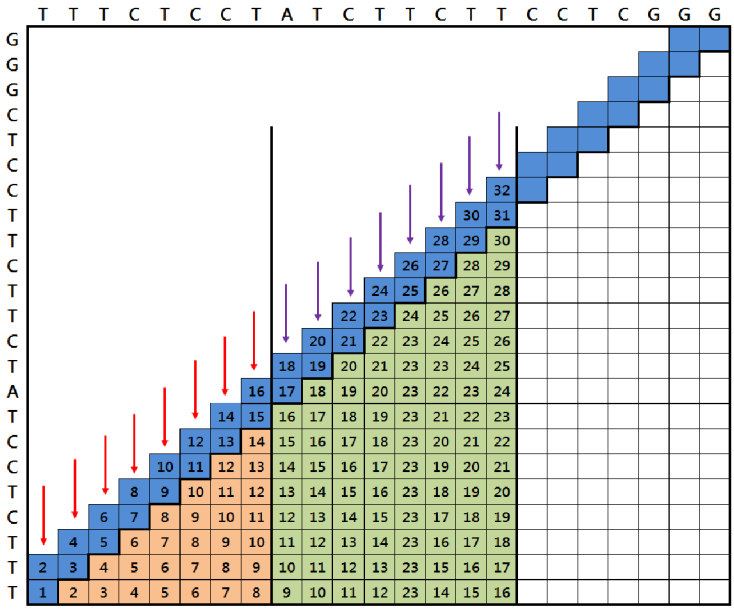
Obrázek 3.8: Poměr rozdělení sdílené paměti pro architekturu Fermi

Zvolení vertikálního směru průchodu Následující odstavec popisuje způsob průchodu dat v rámci jednoho výpočetního jádra. Byl zvolen vertikální směr průchodu, který je znázorněn na obrázku 3.9 (cíleně zde není zaveden překryv pro názornost).

Důvodů pro zvolení tohoto směru vyniklo po prozkoumání architektury hned několik. Hlavním důvodem je možnost ukládání sloupce hodnot při rozdělení výpočtu jádra do části. V případě diagonálního průchodu by bylo nutné zachovávat trojúhelníkovou výseč.

Druhým z důvodů je efektivní přístup k vstupním datům. Každé vlákno si na začátku výpočtu uloží hodnotu vstupních dat na své pozici do registrů. Přístup k datům je v jednom směru omezen pouze na jednu hodnotu. V případě diagonálního přístupu by každé vlákno bylo nuceno přistupovat do globální paměti pro data vícekrát.

[4]bez uvažování délky smyčky



Obrázek 3.9: Způsob průchodu jednotlivých vláken (vlákna prvního bloku jsou znázorněna červenou barvou, vlákna druhého bloku fialovou barvou)

Posledním z důvodů je jednoduchá adresace diagonál v rámci každého vlákna. Každé vlákno přistupuje k diagonálám sestupným způsobem, jak je vidět na indexech diagonál na obrázku 3.9.

Rozdělení vzhledem na vlastnosti klientského spouštění

Výše popisované rozdělení je možné spouštět pouze na serverové straně nebo za speciálních podmínek na osobním počítači[5]. Zde se projevila vlastnost iterativního návrhu, zmíněná v úvodu této kapitoly (Mapování algoritmu na architekturu CUDA). Při testování implementace na jednotce s jednou kartou podporující CUDA se projevila specifická vlastnost ovládače grafické karty, jež ukončí běh jádra po určité době. Doba povoleného běhu není fixní a pohybuje se v rozmezí 3 až 5 vteřin (**časový slot**[6]).

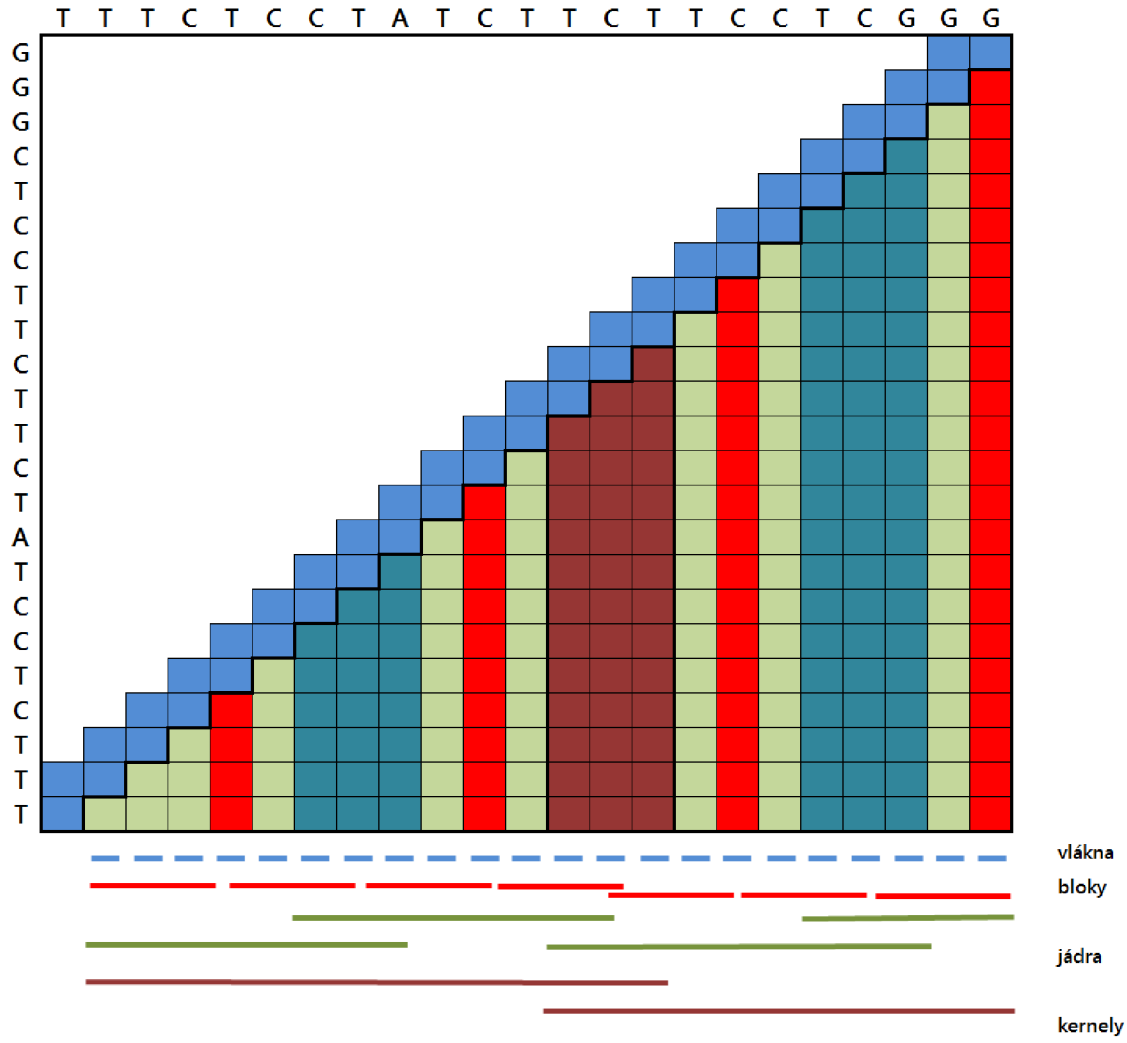
Následkem ukončení běhu je násilné ukončení výpočtů v celém programu, způsobené nutností využívat grafickou kartu také pro zobrazování grafického výstupu operačního systému. Výstup je v době CUDA výpočtu suspendován, což se projevuje jako odpojení výstupu grafické karty do monitoru. V případě, že pomocí karty není nutné zobrazovat grafiku, může výpočet běžet po libovolně dlouhý čas.

Časový slot je přiřazen každému samostatnému kernelu (spuštěnému výpočtu). Způsobem, jak zamezit ukončování výpočtu, je jeho rozdělení do částí, které bude kernel schopen zpracovat v každém časovém slotu. Velikost tohoto bloku dat přitom není možné určit již při návrhu, jelikož je závislá na konkrétním algoritmu a stroji, na kterém je program spuštěn. Velikost je tedy nutné experimentálně nastavit pro každý jednotlivý případ použití, přičemž

[5]nutné podmínky: a) program je spuštěn za vzdáleného sezení tak, že v grafický výstup je interpretován pouze na straně klienta, tedy ne na stanici, na které je spouštěn CUDA výpočet, případně b) systém je vybaven dvěma grafickými kartami, grafická karta CUDA zde má pouze úlohu výpočetní jednotky, druhá grafická karta je používána pro zobrazování grafiky

[6]v rámci CUDA API označován jako *Run time limit on kernels*

je vhodné dosáhnout největšího možného počtu zpracovaných dat jedním kernelem, jelikož každý kernel, stejně jako v případě překrytí bloků, musí počítat s překryvem výpočtu.



Obrázek 3.10: Výsledné rozdělení běhu výpočtu

Výsledné rozdělení běhu výpočtu

Výsledné rozdělení běhu jednoho výpočtu je zobrazeno na obrázku 3.10. Každý *sloupec* matice je zpracováván *jedním vláknem*. Vlákna jsou sdružena do *výpočetního jádra* o fixní velikosti závislé na velikosti dostupné *sdílené paměti*. Každé *jádro* může díky rozdělení na části počítat *násobky své velikosti*. Výpočet jader se *překrývá* o *maximální délku* hledaného triplexu. Počet spuštěných jader v jednom *kernelu* je omezen *časovým slotem* a danou architekturou. Každý spuštěný *kernel* se překrývá výpočtem, stejně jako jádra, o *maximální délku* hledaného triplexu.

Rozdělení dat do pamětí

Jak bylo uvedeno v odstavci Rozdělení pamětí, paměti v rámci karet s podporou CUDA výpočtů jsou rozděleny do několika úrovní. Z pohledu algoritmu rozděleného dle schématu

zmíněného v předchozím odstavci bylo nutné navrhnout správné rozdělení dat do jednotlivých pamětí tak, aby propustnost pamětí nebrzdila algoritmus.

Globální paměť slouží obecně k nahrávání vlastních dat pro zpracování a také pro uložení výsledků výpočtů. Vstupní sekvence nukleotidů je proto před výpočtem nakopírována právě zde. Této paměti je omezena dostupnou pamětí konkrétní karty. V případě karty, na které byl algoritmus vyvíjen, se jednalo o hodnotu 2GB (viz Testovací sestava). Do globální paměti jsou také ukládány vektorové proměnné jednotlivých vláken, buffer printf() funkce kernelů a data konstantní paměti. Jelikož je ale velikost vstupního souboru v případě celého chromosomu pohybuje maximálně do hodnoty 0,5GB, neměl by nastávat problém s nedostatkem paměti.

Načítání dat z paměti je sdružováno do bloků po bytech. Velikost bloku je závislá na datové šířce sběrnice použité karty. Důležité je, že pokud nejsou data čtena postupně, je načten **vždy** celý blok dat a nepotřebná data jsou zahozena. V případě tohoto algoritmu by k dané situaci nemělo docházet, jelikož jsou data načítána postupně vlákny za sebou. Vlákna jsou zpracovávána, jak již bylo zmíněno, po warpech. Pro každý warp je načítání dat pro vlákna sdruženo do jednoho bloku díky navrženému způsobu průchodu algoritmem. Díky tomuto postupu není nutné provádět vícenásobná čtení z paměti.

Výstup algoritmu je vyveden na „standardní výstup“ pomocí funkce *printf()*. Standardní výstup je zapsán v uvozovkách z toho důvodu, že funkce *printf()* v rámci kernelů spuštěných na grafické kartě funguje odlišným způsobem než po překladu pro běžné CPU. Před spuštěním výpočtu je nutné zvolit velikost fixního bufferu[7]. Funkce výpisu poté optimálním způsobem „sbírá“ data ze všech vláken po dobu výpočtu. Po skončení běhu celého kernelu je výstup přenesen na standardní výstup. Tento způsob výpisu je z pohledu návrhu optimální, jelikož se o alokaci prostoru pro výstupní data stará samotné CUDA API. Jelikož délka výstupní sekvence triplexu není v mnoha případech fixní, byla by nutná dynamická alokace prostoru pro výpis. Jak již bylo několikrát zmíněno, uživatelsky řízená alokace v rámci kernelů, je velice neefektivní a je nutné se jí vyvarovat, proto je výhodnější využít vestavěné *printf()* funkce.

Konstantní paměť je omezena na velikost 65536 bytů. Důvodem k tomu to omezení je po spuštění automatické umístění hodnot do L2 cache. Přístupy do této paměti jsou v ideálním případě vybavovány v jednom taktu, proto je zde nutné umístit často potřebná konstantní data, která jsou dostupná všem vláknům. Uložená data v této paměti navíc nevyžadují načítání do registrů vláken, je umožněn přímý přístup. Jelikož se jedná o paměť, která je primárně umístěna v globální paměti, je zde možné ukládat pouze jednorozměrná pole.

V případě algoritmu je vhodné do konstantní paměti umístit pole parametrů, kterými je ovlivněn celý výpočet. Jedná se o pole hodnot parametrů Lambda, Mi a také ohodnocení vlastností triplexů závislých na vybraném typu triplexů: hodnocení skóre, hodnocení členství v izomorfní skupině, hodnocení dodržení pravidel natočení. Dále se v paměti paměti ukládají hodnoty společné pro všechny bloky vláken jako: délka překryvu, velikost vstupních dat, počet opakování bloku a počet vláken na blok (jádro).

Sdílená paměť je dostupná pro každý blok vláken. Jelikož je rychlost sdílené paměti při správném použití mnohonásobně větší, než rychlost globální paměti, je vhodné zde umístit často načítané a ukládané hodnoty. V případě algoritmu se jedná o diagonální prvky. Omezení velikosti paměti nedovoluje libovolné množství prvků, proto bylo nutné svolit fixní velikost dostupných diagonálních prvků.

[7] standardně je výstupní buffer nastaven pouze na 1MB, což je pro většinu vstupních souborů nedostačující

Obrázek 3.11: Rozdělení dat v pamětech

Paměť se dělí mezi hodnoty anti-diagonál a hodnoty hraničních prvků bloku v případě použití rozdělení na části (viz obrázek 3.8, z předchozích kapitol). Velikost dostupné paměti a její uspořádání je možné správně navrhnout pomocí CUDA Occupancy Calculator[8].

V **registrech** se udržují proměnné specifické pro každé vlákno. Díky zvolenému směru průchodu daty je možné také do registrů uložit jednu hodnotu vstupní sekvence, která je používána za celou dobu výpočtu několikrát. Jelikož je přístup do registrů vybavován v jednom taktu, je tato optimalizace velice účinným zrychlením načítání dat.

3.3 Implementace a optimalizace

Následující kapitola je věnována vlastní implementací a optimalizacím, které vycházejí z poznatků architektury CUDA, jež byly shrnuty v podkapitole Grafické procesory - SIMT. Celá kapitola je rozdělená do menších částí. Každá z nich popisuje řešení jednotlivých problémů a optimalizací při implementaci. Jedná se vždy o specifické problémy, proto tok jednotlivých podkapitol na sebe nijak nenavazuje.

Hierarchie zdrojových kódů

Dle doporučení z [22] je vhodné udržovat kód pro výpočet na grafických kartách v jednom souboru. Z tohoto důvodu je celá implementace paralelní podoby algoritmu v jednom zdrojovém souboru *./triplex_hw.cu*. Softwarová implementace původního algoritmu se na-

[8] dostupný z CUDA API na CUDA Occupancy Calculator

chází v souboru *./triplex_hw.cu*. V převzaté softwarové verzi byly provedeny úpravy nutné k možnému jednotnému načítání parametrů programu a testování.

Zdrojové kódy pro platformu CUDA jsou překládány speciálním překladačem, který podporuje podmnožinu jazyka C++ a téměř všechny základní funkce jazyka C. Kód je možné rozdělit do dvou částí:

- kód zařízení (kernel) – **__device__**
- kód hostující stanice – **__host__**

Kód obou zmíněných části je v jednom souboru, přičemž rozdělení funkcí probíhá pomocí prefixů uvedených v seznamu za pomlčkou. Speciálním případem jsou funkce s prefixem **__global__**, které je možné spouštět na obou platformách. Důvodem je možnost testování kódu těchto funkcí.

Hlavní výpočetní funkce, označována v předchozích kapitolách jako **jádro výpočtu**, se ve zdrojových kódech nachází jako funkce:

__global__void triplexKernel(char * data);

Všechny potřebné konstanty a definice používaných datových struktur jsou uloženy v hlavičkovém souboru *./triplex_hw.cuh*. V tomto souboru je vhodné před nasazením programu zkontrolovat a případně změnit nastavení dle použité karty na daném systému.

Rozdělení smyček

Hlavní úlohou paralelního zpracování je rozdělení hlavní smyčky běhu programu tak, aby byla nahrazena paralelním během vláken. V případě vybraného algoritmu vypadají smyčky sekvenční verze následovně:

```
for (x = params->min_loop + 1;
     (x <= 2 * params->steps) && (x < input->size); x++)
{
    for (i = x; i < input->size; i++)
    {
            data_i = input->data[i];
            //výpočet
            d += 2;
    }
}
```

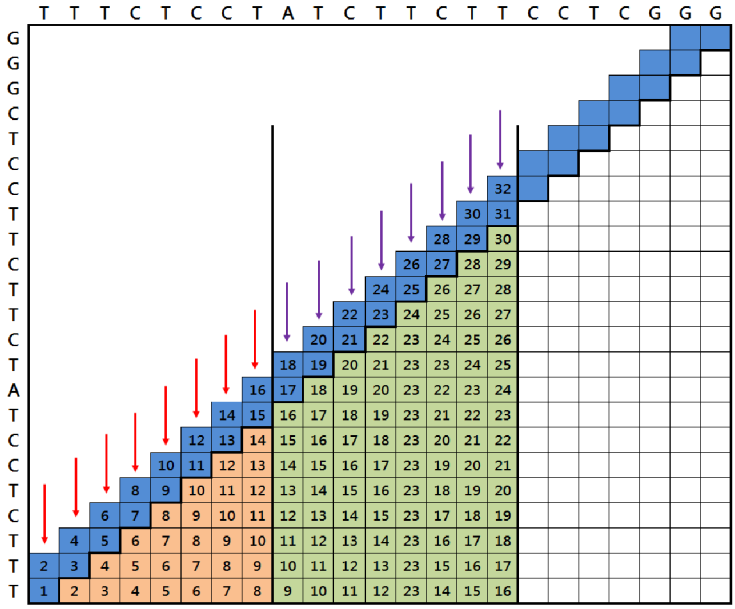
Do paralelní podoby byla převedena, dle návrhu, druhá ze smyček. Běh smyčky je rozdělen mezi jednotlivá vlákna. Výsledné řešení v paralelní podobě:

```
x = (min_loop + 1);

data_i = data[id + min_loop + 1]; // náhrada původní smyčky for(i)

for(d = x + 1 + (2 * threadIdx.x);
    (x <= (unsigned int)(2 * params_HW.steps)) && (x < size_HW) &&
    (d < diag_size-1); ++x)
```

```
{
    //výpočet
    if(d > 0)
        d--;
        __syncthreads();
}
```

Smyčka *for(i = x; ...)* byla nahrazena rozdělením do vláken. Proměnná *data_i* představuje pro vlákno lokální cache pro uložení vstupní hodnoty, která danému vláknu přísluší. Původní smyčka *for(x = params...)* byla nahrazena smyčkou *for(d = x + 1...)*. Původní smyčka *for(x = params...)* iterovala anti-diagonálami od hlavní anti-diagonály směrem k pravému spodnímu rohu, přičemž interní smyčka procházela po diagonálách.

V paralelní verzi je smyčkou, která iteruje anti-diagonálami, smyčka *for(d = x + 1...)*. Každému vláknu je indexem *d* přidělen příslušný diagonální prvek. Index se v rámci smyčky dekrementuje, čímž je zaručen průchod maticí shora dolů. Nejlépe tuto situaci vystihoval již uvedený obrázek 3.9 z kapitoly Mapování algoritmu na architekturu CUDA, který je zde pro úplnost uveden znovu:



Obrázek 3.12: Běh vláken

Jednotlivé šipky znázorňují vlákna, tedy v případě smyček se jedná o *for(d = x + 1...)*. Číselné hodnoty v buňkách matice jsou indexy anti-diagonál *d*. Poslední z označených vláken (fialové nejvíce vpravo) začíná na hlavní anti-diagonále s indexem 31 a postupně počítá hodnoty anti-diagonálních prvků 31, 30, 29, atd. Před každým novým posunem vláken o jednu anti-diagonálu níže je nutná synchronizace pomocí bariéry (*syncthreads()*), jelikož z důvodu závislosti okolních hodnot je nutné mít zaručeno, že všechny hodnoty předchozí anti-diagonály byly již dopočítány.

Plánování bloků

Jak bylo zmíněno v kapitole Mapování algoritmu na architekturu CUDA byl pro vhodné plánování počtu nasazených bloků použit prvočíselný rozklad. Důvodem pro správné plánování počtu bloků je doba zpracování vstupu. Každé jádro, které je inicializováno, navíc musí být na kartě zpracováno a to vyžaduje určitou režii. Rozlišení plánování je na úrovni bloků vláken, tedy v případě Fermi se může v případě jednoho nadměrného bloku zpracovávat již 472 vláken navíc. Jelikož se vlákna zpracovávají po warpech, režie vybavení vláken navíc narůstá rapidním tempem.

Implementace prvočíselného rozkladu je provedena pomocí Eratosthenova síta. To je implementováno jako bitové pole. Vyhledání prvočísel je tedy velice rychlé. Prvočíselný rozklad se snaží zajistit optimální rozložení počtů vláken do dimenzí. Pokud se v rozkladu nachází prvočíslo větší než je povolený počet vláken na jednu dimenzi, je toto číslo děleno dvěma do té doby, než jeho hodnota je menší než hranice dimenze. Tento způsob řešení zajišťuje do jisté míry optimální rozklad bloků do dimenzí.

Optimalizace proměnných a smyček

Počet skalárních informací, které mohou být ve vláknech uloženy, je omezen v rámci bloku, což je jedním z omezujících faktorů maximálního počtu vláken na blok. Proto je vhodné optimalizovat běh programu a eliminovat v něm zbytečné proměnné.

Výpočetní síla grafických karet je ve vykonávání stejných instrukcí současně. Proto je možné si dovolit výpočet některých hodnot proměnných provést vícekrát bez újmy na rychlosti běhu programu a tím ušetřit proměnné.

Optimalizace, která se běžně provádí v kódu pro běžné procesory, tedy pokud je některá hodnota potřebná na více místech v rámci funkce - je přepočítána do proměnné, je inverzní operací k té na grafické kartě. S cílem ušetřit jeden dostupný registr na vlákno a tím 472 registrů na blok, je hodnota počítána na více místech.

Proměnných, které bylo možné daným způsobem změnit, se nacházelo v programu několik, proto tato optimalizace přinesla zisk 4 registrů na vlákno.

Optimalizací za stejným cílem, získání dostupných registrů, bylo využití existujících proměnných stejného typu pro jiné účely v daném kontextu. Jinými slovy, pokud byla proměnná definována na globální úrovni a je využívána pouze v některých podmíněných výrazech, je možné tuto proměnnou využít v jiných místech, pokud to nijak neovlivní původní využití proměnné.

V několika místech v kódu se objevovaly smyčky, které se prováděly pouze dva krát. Pro optimální běh programu byly tyto smyčky rozbaleny, čímž došlo také k získání volného registru (proměnná indexu) a také k úbytku několika komparačních a skokových instrukcí.

Optimalizace bloků

V rámci jednoho streaming procesoru se nachází pouze několik jednotek SFU přidělených na pár CUDA jader. Funkcí SFU jednotky je vybavování speciálních matematických funkcí: dělení, odmocnina atd. V implementaci algoritmu se na několika místech nachází dělení hodnotou 2 a také operace modulo 2.

Obě z těchto operací byly nahrazeny odpovídajícími funkcemi bitových posuvů. Tato optimalizace zajišťuje výpočet dělení a modula v každém CUDA jádře. Díky tomu je zamezeno případné sekvenční zpracování zmíněných matematických instrukcí. Ukázka nahrazení operace modulo 2:

```
//původní kód
if (tri_type % 2 == 0)

//výsledný kód
if (params_HW.tri_type ^ 1 == 0)
```

Uložení konstant

Program pro svůj běh vyžaduje dostupnost vícerozměrných polí. Jak již bylo zmíněno dříve, tato pole jsou nakopírována do konstantní paměti, aby byla zaručena jejich dostupnost.

Konstantní paměť vyžaduje lineární adresování dat. Všechna pole zavedená do konstantní paměti mají proto jednodimenzionální rozměr, ale přístup k datům je mapován do 2D prostoru.

Každé hledání triplexů má předem určený typ hledaného triplexu. Tato skutečnost dovolila optimalizaci přístupu k polím takovým způsobem, že do konstantní paměti jsou nakopírována pouze pole hodnot příslušící danému typu. V důsledku je tak zmenšena velikost požadované konstantní paměti a adresování polí se z původních tří dimenzí[9] zmenší na dvě.

Optimalizace výstupu

Výstupní funkce v původní verzi algoritmu využívala několika smyček pro výpis výsledné sekvence pomocí funkce *printf()* předcházených výpisem parametrů nalezeného triplexu, jak je vidět v následující ukázce kódu:

```
//výpis informací
printf("%s ",info);

//výpis sekvence
for (i = start_ch; i <= start_gap; i++)
    printf("%c", data[i]);
printf("-");
for (i = start_gap + 1; i < end_gap; i++)
    printf("%c", data[i]);
printf("-");
for (i = end_gap; i <= end_ch; i++)
    printf("%c", data[i]);
printf("\n");
```

Pro paralelní výpis tento způsob výpisu nebyl vhodný z důvodu současného výstupu více vláken. Ve výsledném výstupu se v případě výpisu více vláken současně objevila sekvence znaků namíchaných z výstupních funkcí různých vláken.

Pro účely výpisu bylo nutné zavést buffer, ze kterého by bylo možné vypsat celou informaci o triplexu v jednom kroku. Funkce *printf()* je z paralelního pohledu atomickou funkcí, jeden výpis touto funkcí vrátí korektní řetězec na výstupu, který nemůže být ovlivněn jinými vlákny.

Problémem, který doprovází celou architekturu CUDA, je velice nízká rychlost dynamické alokace. Z tohoto důvodu je buffer pro výpis konstantní velikosti, kterou lze určit při překladu programu.

[9] třetí dimenzí poli byl typ triplexu

Buffer se nachází v lokální paměti každého vlákna. Jelikož se jedná o vektor, hodnoty jsou uloženy v globální paměti (viz odstavec Rozdělení pamětí v kapitole Grafické procesory - SIMT). V rámci výpisu nalezeného triplexu je nutné několikrát přistupovat do globální paměti. Při výpisu nukleotidové sekvence jsou data kopírována v rámci globální paměti do bufferu, který se také nachází v globální paměti. Velikost bufferu je nastavena na fixní velikost 1024 znaků. Toto místo je vyhrazeno pouze pro nukleotidovou sekvenci. V případě, že by buffer byl nedostačující nebo příliš velký, je možné jeho velikost modifikovat v hlavičkovém souboru.

Z důvodu těchto přístupů na globální paměť je funkce výpisu triplexů velice časově náročnou. Doba běhu algoritmu je tedy velice zatížena počtem nalezených triplexů. Optimální přístup k paměti by bylo možné vyřešit buffery v rámci sdílené paměti. Problémem však je velikost dostupné sdílené paměti, ve které se nacházejí prvky diagonál. S využitím bufferu by musel být počet diagonálních prvků omezen, a tím by klesl výkon celé aplikace. V případě přístupu do globální paměti lze počítat, že jelikož sekvence byla procházena vlákny při výpočtu diagonál, bude většina čtených hodnot stále dostupná v L1 cache.

Zjednodušený kód výsledné optimalizované funkce:

```
#define TRIPLEXBUFFERSIZE 400

//načtení sekvence do bufferu
buf_i = 0;
    for (i = start_ch; i <= end_ch; i++)
    {
        if(i == end_gap) buffer[buf_i++] = '-';

        if(i+offset < size_HW) buffer[buf_i++] = data[i]);

        if(i == start_gap) buffer[buf_i++] = '-';
    }
buffer[buf_i] = '\0';

//výpis informací i sekvence souběžně
printf("%s %s\n", info, buffer);
```

Kapitola 4

Testování, ověřování, ohodnocení a možnosti rozšíření

4.1 Ověřování korektnosti výsledků

Korektnost

Při převodu algoritmu bylo velmi důležité dbát na korektnost výsledků každé nové verze. U paralelního zpracování pomocí technologie CUDA je nutné klást důraz na správnost přiřazování indexů jednotlivým vláknům. Ale také je nutno dbát na synchronizaci vláken a synchronizované zapisování hodnot do paměti. Každou úpravu kódu proto bylo nutné testovat na korektnost. V případě provedení více úprav bez testování by mohla nastat chyba ve výsledcích, která je téměř nedohledatelná, zvláště v případě, pokud se projevuje jenom na některých spuštěných vláknech.

Při testování korektnosti bylo vycházeno z výsledků původního algoritmu, který byl modifikován pouze do takové podoby, aby bylo možné výsledky kontrolovat. Při vývoji bylo nutné udržovat i sekvenční (původní) verzi algoritmu, která byla dle potřeby modifikována pro produkci kontrolních výpisů. Za účelem jednoduchosti byl do programu implementován jednotný vstup, kterým je možné zvolit spuštění paralelní či sekvenční verze algoritmu. Načítání parametrů je jednotné pro obě verze. Rozdělení jednotlivých parametrů probíhá až před samotným spuštěním vybrané verze algoritmu.

Kontrola pomocí debuggeru v tomto případě nepřipadá v úvahu, jelikož je třeba kontrolovat velké množství hodnot najednou, v ideálním případě pro několik cyklů běhu algoritmu. Za účelem testování korektnosti vznikly automatizované testy pro příkazový řádek konzole MS Windows.

Hlavní test je zaměřen nejen na korektnost hodnot, ale také na správnou synchronizaci zápisu. V obou verzích programu byly doplněny pomocné výpisy kontrolující obsah diagonálních prvků v každém kroku. Tento způsob kontroly vede u sekvenční verze ke zjištění postupně přibývajících hodnot v jednotlivých proměnných. V případě paralelní verze je složité sledovat výstup, jelikož výpis z jednotlivých vláken probíhá paralelně. Ve výstupním souboru se tedy prolínají výpisy ze všech spuštěných vláken. Výstup obou verzí je proto setříděn a porovnán. K porovnání je využíván program WinMerge, který dokáže zarovnat dva podobné soubory v přehledném rozhraní a zvýraznit jejich rozdílnost.

Všechny testy jsou přiloženy na dodaném médiu ve složce *./tests*. Je zde doložen také instalační soubor programu WinMergre a stručný návod ke zprovoznění a použití automatizovaných testů.

Kontrola indexace

Jak bylo zmíněno v odstavci Důsledky rozdělení
kapitoly Architektura Fermi, správná indexace je v programech pro platformu CUDA obtížným problémem. Z toho důvodů je jeden z automatizovaných testů zaměřen na tuto úlohu. Simuluje sekvenční zapojení vláken do struktury stejné, která je používána v CUDA implementaci.

Úkolem je zjistit všechny indexy používané jednotlivými vlákny. Kontrola je nutná z důvodu překrývajících se přístupů k datům. Pro každé vlákno jsou vypsány indexy používané za běhu programu. Kontrola probíhá stejným způsobem jako v předchozím případě, tedy seřazením a diferencí softwarové simulace a hardwarové implementace. Ke správnému spuštění je nutné mít dostupné prostředí Python verze 3.0 a novější.

Ladění

Laděním lze v případě vývoje pro platformu CUDA nazývat zkoumání hodnot proměnných a částí paměti jednotlivých vláken pomocí debuggeru. Na první pohled není zřejmá složitost této úlohy. Čtyři pětiny doby práce na tomto projektu nebyl dostupný debugger pro jednu grafickou kartu. Pokud byla grafická karta samostatně osazena v jednom stroji, bylo možné spuštění buďto výpočtu nebo grafického výstupu. Důsledkem tohoto faktu je nemožnost ladění kódu na jednom stroji.

Za tímto účelem bylo nutné použít dva PC propojené místní sítí. Jednotka s CUDA kompatibilní kartou sloužila v daném okamžiku jako výpočetní, na druhé jednotce bylo spuštěno vývojové prostředí se spuštěným debuggerem, který je v případě CUDA platformy nazýván Nsight Monitor. Spuštění a ladění běhu programu tedy probíhalo vzdáleně.

Při finálních fázích práce byl společností Nvidia vydán nový balík ovládačů a debuggeru umožňující ladění kódu na jedné kartě na jednom PC. Jednalo se však o beta verzi, která v mnoha případech vypisovala spousty varovných či chybových hlášení.

Implementace paralelní verze algoritmu tedy vznikala pomalým tempem z důvodu výše zmíněných faktů. Zvláště při převodu algoritmu z existující verze je důležité dbát na správnost postupu, což v tomto případě vyžadovalo nadměrnou spoustu času díky komplikovanému postupu.

V kódu se na několika místech nacházejí zakomentované ladící výpisy. Jak bývá zvykem v programech pro CPU jsou tyto části zasazeny do podmíněných výrazů omezených jednou z definovaných konstant. Jelikož se jedná o paralelní systém, výpis ladících informací byl nutný často pouze pro jednotlivá vlákna. Omezení globální proměnnou by v tomto případě nemělo smysl. V komentáři se tedy vždy nachází *výpisová* část ladící informace, kterou je vždy nutné doplnit vhodnou podmínkou, aby byla vypsána pouze v konkrétním případě. Cílem tohoto systému výpisů je získat informace pouze o konkrétním vlákně v konkrétní situaci a zbavit se tak nutnosti hledat tuto informaci ve výpisu všech vláken.

4.2 Testování účinnosti nového algoritmu

Sestava pro testování

V tabulce, umístěné v příloze B: Testovací sestava, jsou uvedeny podstatné hardwarové a softwarové parametry sestavy, na které byla implementace testována. V případě implementace pro CUDA kompatibilní zařízení jsou tyto parametry důležité, neboť sestavení programu na jiné konfiguraci může způsobovat značné problémy, zvláště v případě odlišného

CUDA SDK. Důvodem jsou rychlé a významné změny v SDK promítající se do implementací. Příkladem může být podpora využití funkce *printf()*, která je dostupná pouze pod SDK verze > 3 a vyžaduje podporu výpočetní kompatibility > 2.0, kterou je nutné aktivovat explicitně u překladu.

Způsob testování

Testování účinnosti implementace algoritmu probíhalo v několika iteracích. V každé iteraci byl měřen běh programu v softwarové a hardwarové verzi pro zadaný vstup. Vstupní sekvence byly rozděleny do následujících kategorií:

- null - sekvence neobsahující žádné triplexy
- full - sekvence plné možných triplexů
- random - sekvence obsahující náhodně generované nukleotidy
- real - existující nasekvenované řetězce nukleotidů

Pro každou z těchto kategorií byl proveden test na 5-ti připravených vstupních souborech, obsahujících různě dlouhé sekvence nukleotidů dané kategorie. Pro každý vstupní soubor proběhlo 5 měření rychlosti běhu programu s přesností na milisekundy.

Aby bylo měření objektivní, začíná měření času běhu programu v softwarové verzi až přímo před smyčkou hlavního výpočtu. V případě hardwarové (CUDA) implementace je také do výsledného času zahrnut časové úseky, ve kterých jsou data kopírována směrem do karty a zpět. Proces kopírování je nedílnou a nutnou součástí výpočtu, proto je zahrnut do celkového času běhu programu.

V případě obou algoritmů při měření zůstal aktivovaný výpis nalezených výsledku na standardní výstup, který byl následně přesměrován do souboru. Důvodem k jeho ponechání byl fakt, že algoritmus je vždy v běžném režimu spouštěn s aktivovaným výstupem. Proto je vhodné zakomponovat tuto časovou hodnotu do celku. V případě implementace na CUDA platformě může mít výpis, díky zmíněným obtížnostem funkce *printf()*, značný vliv na výsledný čas běhu.

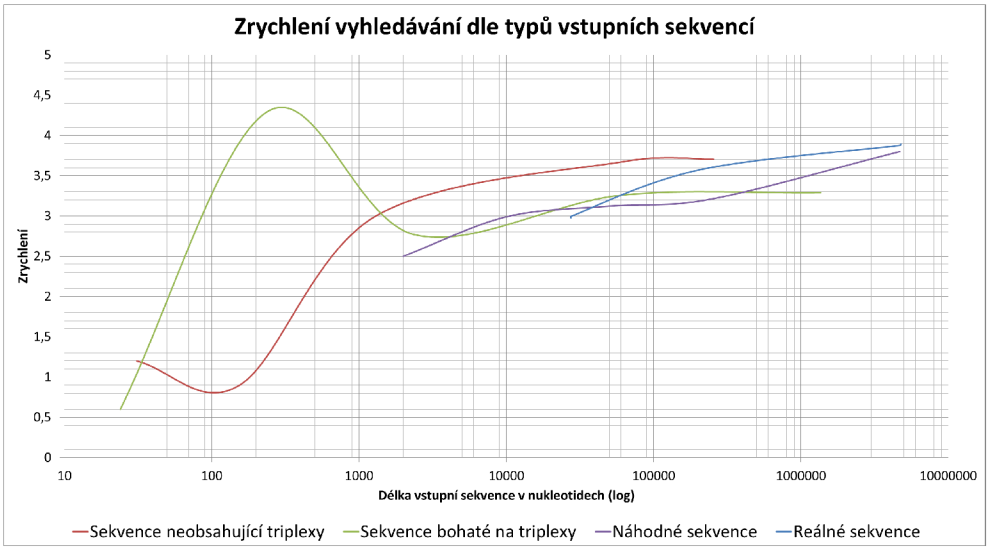
Měření bylo prováděno pomocí vzdáleného připojení k PC s CUDA kartou, aby nebyl výsledek ovlivněn zatížením grafických výpočtů karty. Byla použita verze bez podpory rozdělení na části, jelikož se tato funkce projevila jako velice neefektivní při minoritních testech. Funkce rozdělení do části byla tedy pro získání optimálního výsledku deaktivována pomocí komentářů ve zdrojových kódech, aby byly šetřeny prostředky grafické karty[1]. Finální stav zdrojového kódu odpovídá konfiguraci používané při testování.

Pro testování byly zvoleny fixní parametry spuštění. Cílem bylo získat statisticky směrodatný výsledek, proto byly porovnávány výstupy běhů se stále stejnými parametry.

Výsledky

Hodnoty všech naměřených časů je možné nalézt v souboru *./test/experiments.xlsx* na přiloženém CD. Použité vstupní sekvence rozdělené do zmíněných kategorií je možné nalézt ve složce *./input/* na stejném médiu.

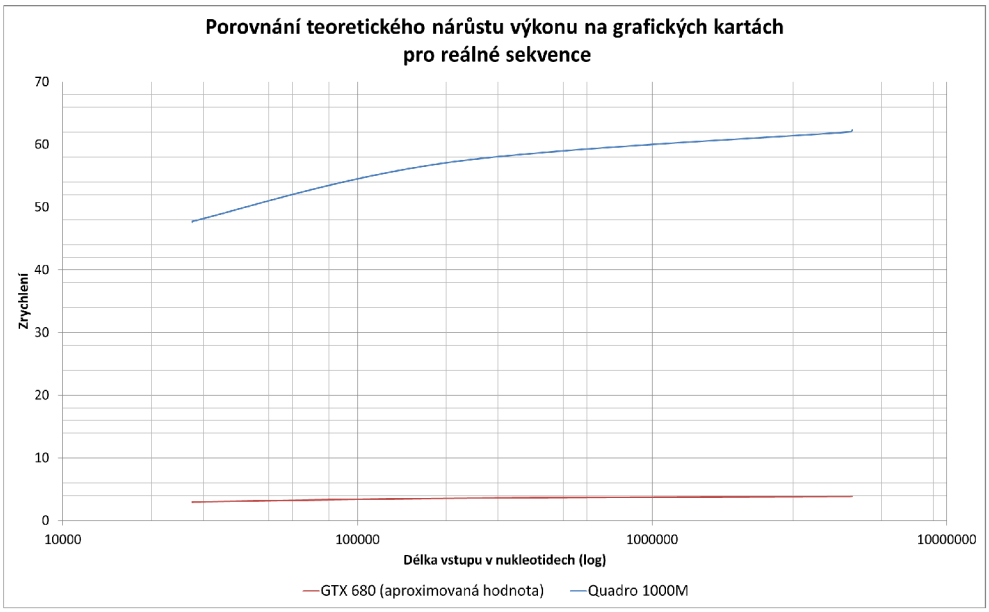
[1] v případě použití podmíněného výrazu by byl výpočet zatížen zbytečnými komparacemi, proto byl zvolen komentář



Obrázek 4.1: Výsledné zrychlení pro různé typy vstupních sekvencí

Výsledné zrychlení na grafické kartě je vyneseno do grafu zobrazeného na obrázku 4.1. Jak je vidět, zrychlení pro krátké sekvence je závislé na vstupní sekvenci. Pro sekvence o délce větší než 10000 nukleotidů se zákmity způsobené odlišnosti vstupní sekvence minimalizují a křivka se blíží hranici čtyřnásobného zrychlení.

Z grafu je také patrné, že nárůst výkonu pro sekvence bohaté na triplexy není tak patrný, jako u ostatních typů sekvencí. Toto je způsobeno složitostí exportní funkce. V případě běhu na grafické kartě je každý export triplexu do na výstup časově náročnou operací, jelikož ji provádí pouze jedno z vláken. Všechna další vlákna jsou v ten moment zastavena na bariéře a čekají na dokončení exportu.



Obrázek 4.2: Teoreticky dosažitelné zrychlení při vyhledávání triplexů v reálných sekvencích

Grafická karta, na které bylo prováděno testování patří v segmentu grafických karet do nejnižší třídy. Jelikož se jedná o grafickou kartu notebooku, jsou výpočetní vlastnosti karty omezeny na tolik, aby bylo možné kartu v malém šasi uchladit. Výsledné zrychlení je tedy odrazem výkonu grafické karty, které disponuje pouze 2-mi streaming multiprocesory a 96 CUDA jádry (viz Testovací sestava).

Dnes dostupné karty disponují až 1536-ti CUDA jádry, vyšší propustností pamětí a mnohonásobně vyšším taktem jádra a paměťových modulů[2]. S ohledem na všechny vlastnosti výkonnějších karet byly výsledky testování aproximovány příslušný koeficienty a vyneseny do grafu. Výsledné porovnání hodnot použité testovací karty a aproximovaně získaných hodnot výkonné moderní karty jsou zobrazeny v grafu na obrázku 4.2. Teoretické zrychlení pro výkonnější kartu přesahuje hodnotu 60.

4.3 Experimentální ohodnocení a budoucí práce

Experimentální ohodnocení

Jak je patrné z kapitoly Testování účinnosti nového algoritmu, návrhem a následnou implementací paralelní podoby algoritmu bylo získáno značné zrychlení. V případě karty, na které byla implementace testována byla hodnota zrychlení pro reálné sekvence v průměru **3,46**. Bylo odvozeno odpovídající zrychlení pro výkonnější variantu grafické karty, které dosáhlo hodnoty **55,38**.

Značného zrychlení bylo dosaženo díky správnému využití hierarchie pamětí architektury Fermi. Všechny získané poznatky o výpočtech SIMT a stavbě karet přizpůsobených pro CUDA výpočty byly aplikovány do návrhu s cílem dosáhnout jak nejlepšího výsledku.

Na grafických kartách je možné dosáhnout v případě některých algoritmů až několika set násobného zrychlení. Toto však neplatí pro paralelizovaný algoritmus, jelikož závislosti hodnot v matici výpočtů nedovolují samostatný běh každého vlákna. Výpočet je tedy zatížen synchronizací vláken v rámci bloků. Ze stejného důvodu bylo nutné zavést překryvy v místech přechodů mezi výpočty jednotlivých vláken, což mělo za následek další omezení možnosti zrychlení.

Posledním omezujícím faktorem je množství podmíněných výrazů, které se v algoritmu vyskytují. Množství větvení programu je výsledkem uvažování mnoha vlastností triplexů, které se využívají. Eliminace většiny podmíněných výrazů proto není možná. Každý z těchto výrazů představuje problematické místo paralelizace, proto je téměř nemožné dosáhnout úplně optimálního způsobu paralelizace.

I přes všechny složitosti, které byly při převodu algoritmu zjištěny, je výsledek uspokojující, přičemž dosažitelné zrychlení pro běžně dostupné grafické karty se pohybuje řádech desítek.

Budoucí práce

Tato podkapitola v několika odstavcích shrnuje možné směry dalšího vývoje této aplikace. Každý odstavec představuje jednu možnost pokračování.

[2] jedná se o vlastnosti modelu NVIDIA GTX 680

Rozšíření o další možné typy triplexů

Původní algoritmus je navržen pro hledání intramolekulárních triplexů. Modifikací způsobu hledání by bylo vhodné algoritmus rozšířit o možnost hledání intravláknových triplexů. Touto modifikací by se značně rozšířila množina nalezených triplexů.

Zapouzdření do knihovny

Výsledný projekt je koncipován jako program pro hledání triplexů s možností výběru verze, která má být použita. Do budoucna by bylo vhodné převést program do formy knihovny, kterou bude možné nalinkovat do jiných programů. V případě převodu do knihovny by bylo vhodné také zavést možnost načítání různých formátů vstupních dat. Momentálně program pracuje se vstupy v čistě textové podobě, tedy se sekvencemi nukleotidů.

Rozšíření o heuristickou nástavbu

Během iterací návrhu se objevila také myšlenka modifikace algoritmu do heuristické podoby. Výsledný návrh, který je v této práci popsán, by se v případě heuristického řešení prezentoval jako přesné hledání.

Běh hledání by byl rozdělen do několika dílčích kroků. Dle uživatelem nastavených parametrů by se heuristicky nastavila funkce hledání částí triplexů, které neobsahují inzerce a delece. Funkce by prohledala paralelním způsobem celé vlákno na výskyty dílčích triplexních úseků. Výsledkem by byla vstupní sekvence označená v místech potencionálních výskytů triplexů.

Následující krok by shlukovou analýzou nalezl potenciálně zajímavé úseky v stupních dat. Každý jednotlivý úsek by byl následně předán podrobné analýze pomocí přesného hledání. Díky tomuto postupu by se eliminovala značná část vstupních dat, která je v případě aktuálního přístupu teoretický zpracovávána zbytečně. Cenou za toto řešení by byla ztráta informací o některých triplexech, který by nebyly nalezeny díky nepřesnosti heuristické funkce či shlukové analýzy.

Původně během návrhu bylo uvažováno, že toto rozšíření bude v konečném řešení také implementováno. Bylo však zjištěno, že složitost shlukové analýzy na CUDA platformě by díky neefektivitě dynamické alokace a atomických funkcí, byla nadměrná a vyžadovala by mnoho času, který by mohl převyšovat i čas dostupný k vypracování této práce. Proto bylo od tohoto záměru upuštěno.

Převod pro platformu OpenCL

Programy psané pro CUDA platformu jsou velice podobné programům pro platformu OpenCL (viz informace v odstavci OpenCL). Do budoucna by bylo vhodné převést implementaci také pro tuto platformu. Jelikož se jedná o univerzální platformu pro paralelní zpracování, bylo by možné takto zaručit běh programů na grafických kartách jiných značek, případně na kartách FPGA.

Integrace do webového rozhraní

Souběžně s touto prací vznikalo, jako další diplomová práce, uživatelské webové rozhraní pro vyhledávání triplexů s možností grafického výstupu nalezených dat. Základem pro stavbu tohoto rozhraní byl původní sekvenční algoritmus, ze kterého vychází i tato práce. Aplikace poskytuje přehledný přístup k vyhledávání a pokročilé možnosti filtrování, proto by bylo

vhodné nahradit sekvenční algoritmus paralelní verzí navrženou a implementovanou v této práci. Mohl by tak vzniknout komplexní portál pro rychlé akcelerované hledání triplexů s uživatelsky přívětivým rozhraním.

Kapitola 5

Závěr

Účelem této práce bylo navrhnout paralelní způsob vyhledávání triplexů v DNA, který vychází ze známého řešení a je možné jej implementovat na dostupných architekturách. Ve výsledku vznikl návrh řešení i s odpovídající implementací, který je schopen urychlit vyhledávání triplexů až **50-ti násobně** oproti existujícímu sekvenčnímu řešení a to při zachování veškerých uvažovaných vlastností triplexů.

První část práce, kapitola 2, byla věnována detailnímu pohledu na triplexy. Triplexy zde byly rozděleny do několika kategorií dle specifických vlastností. Nutnost znalostí informací o triplexech byla promítnuta do návrhu, jelikož zpracovávaný algoritmus při výpočtu uvažuje téměř všechny doposud známé vlastnosti triplexů. Další část kapitoly 2 byla věnována přehledu dostupných technologií se zaměřením na platformu Nvidia CUDA. Detailní přehled vlastností SIMT výpočtů a vlastností architektury CUDA Fermi byl zde umístěn za účelem úplného vysvětlení všech optimalizací, které byly provedeny ve výsledném řešení.

Kapitola 3 představuje vlastní návrh převodu sekvenční verze algoritmu do paralelní podoby pro platformu CUDA Fermi. Veškeré zde uvedené postupy se odvolávají na vlastnosti CUDA platformy a vycházejí z předem získaných znalostí o této platformě. Popis implementace v další části kapitoly popisuje detaily mapování algoritmu a optimalizace provedené za účelem šetření prostředků, či za účelem dosažení většího zrychlení.

Poslední z kapitol, kapitola 4, popisuje dosažené výsledky a způsoby jejich získání. Druhá polovina kapitoly je věnována možným pokračování této práce.

Návrh řešení byl pojat komplexně, proto by další práce na jeho zlepšení spočívaly v optimalizacích malých detailů souvisejících s implementační platformou. Aktuální stav implementace je na úrovni beta verze programu. Pro nasazení do praxe by zajisté bylo nutné provést větší množství komplexních testů a výsledné řešení začlenit do uživatelsky přívětivé formy, jak bylo naznačeno v podkapitole 4.3.

V kombinaci se současně vyvíjeným webovým rozhraním by se mohla implementace stát komplexním nástrojem pro přesné vyhledávání triplexů v DNA a uživatelům tak poskytnout rychlý nástroj pro jejich analýzu.

Literatura

[1] ATI: A History of GPGPU. [online], 2011 [cit. 2011-12-22], A Brief History of General Purpose (GPGPU) Computing.
URL http://www.amd.com/us/products/technologies/stream-technology/opencl/pages/gpgpu-history.aspx

[2] BISSLER, J. J.: Triplex DNA and human disease. *Front. Biosci.*, ročník 12, 2007: s. 4536–4546.

[3] BRUCE, A.; BRAY, D.; JOHNSON, A.; aj.: *Základy buněčné biologie*. 2. vydání, Ústí nad Labem, CZ: Espero Publishing, 2005, ISBN 80-902906-2-0.

[4] DOŠKAŘ, J.: Molekulkární genetika I. 2009 [cit. 2011-12-27], přednášky 2009.

[5] DUCA, M.; VERKHOFF, P.; OUSSEDIK, K.; aj.: The triple helix: 50 years later, the outcome. *Nucleic Acids Res.*, ročník 36, září 2008: s. 5123–5138.

[6] FRIED, M.: GPGPU Architecture Comparison of ATI and NVIDIA GPUs. červen 2010, Microway, Inc.

[7] GADDIS, S. S.; WU, Q.; THAMES, H. D.; aj.: A Web-Based Search Engine for Triplex-forming Oligonucleotides Target sequences. *Oligonucleotides 16*, 2006: s. 196–201.

[8] Goni, J. R.: Triplex-forming oligonucleotide target sequences in the human genome. *Nucleic Acids Research*, ročník 32, 2004: s. 354–360, doi:10.1093/nar/gkh188, iSSN 1362-4962.
URL http://www.nar.oupjournals.org/cgi/doi/10.1093/nar/gkh188

[9] HOYNE, P. R.; EDWARDS, L. M.; VIARI, A.; aj.: Searching Genomes for Sequences with the Potential to Form Intrastrand Triple Helices. *J. Mol. Biol. (2000) 302*, 2000, 797-809, doi:10.1006/jmbi.2000.4502.

[10] HOYNE, P. R.; GACY, A. M.; MCMURRAY, C. T.; aj.: Stabilities of intrastrand pyrimidine motif DNA and RNA triple helices. *Nucleic Acids Res.*, ročník 28, únor 2000: s. 770–775.

[11] JAIN, A.; WANG, G.; VASQUEZ, K. M.: DNA triple helices: biological consequences and therapeutic potential. *Biochimie*, ročník 90, srpen 2008: s. 1117–1130.

[12] JENJAROENPUN, P.; A KUZNETSOV, V.: TTS Mapping: integrative WEB tool for analysis of triplex formation target DNA Sequences, G-qaudruplets and non-protein coding regulatory DNA element in the human genome. *BMC Genomics 2009, 10(Suppl 3):59*, 2009, doi: 10.1186/1471-2164/10/S3/S9.

[13] JURČO, J.: Program pre analýzu DNA sekvencií z hľadiska výskytu triplexov, Bakalářská práce, Masarykova Univerzita, Fakulta informatiky. jaro 2008, Brno.

[14] LEXA, M.; MARTÍNEK, T.; BURGETOVÁ, I.; aj.: A dynamic programming algorithm for identification of triplex-forming sequences. *Bioinformatics, Oxford, Oxford University Press, Great Britain*, 2011-07-26, ISSN 1367-4803, 2011, vol. 27, no. 17, 8 pp.

[15] LIGOWSKI, L.; RUDNICKI, W.: An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. *Parallel & Distributed Processing*, květen 2009: s. 1–8.

[16] LIU, Y.; HUANG, W.; JOHNSON, J.; aj.: GPU Accelerated Smith-Waterman. In *Computational Science – ICCS 2006*, *Lecture Notes in Computer Science*, ročník 3994, editace V. Alexandrov; G. van Albada; P. Sloot; J. Dongarra, Springer Berlin / Heidelberg, 2006, ISBN 978-3-540-34385-1, s. 188–195, 10.1007/11758549_29.
URL http://dx.doi.org/10.1007/11758549_29

[17] MANAVSKI, S. A.; VALLE, G.: CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, ročník 9 Suppl 2, 2008: str. S10.

[18] MARTÍNEK, T.; BURGETOVÁ, I.: Bioinformatika. 2010 [cit. 2011-12-28], přednášky 2010.

[19] MARTÍNEK, T.; LEXA, M.: Hardware Acceleration of Approximate Palindromes Searching. *ICECE Technology*, prosinec 2008: s. 65–72.

[20] MIRKIN, S. M.: Discovery of alternative DNA structures: a heroic decade (1979-1989). *Frontiers in Bioscience 13*, 2008-01-01, 1064-1071.

[21] NVIDIA: NVIDIA's Next Generation CUDA Compute Architecture: Fermi. [online], 2011 [cit. 2012-01-09], Whitepaper. V1.1.
URL http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[22] NVIDIA: CUDA C Best Practices - Design Guide. [online], 2012-01-11 [cit. 2012-05-15], DG-05603-001_v4.1 | January 2012.
URL http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf

[23] NVIDIA: NVIDIA CUDA C Programming Guide. [online], 2012-04-16 [cit. 2012-05-12], Version 4.2.
URL http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

[24] PANDE, V.; Stanford University: Folding@Home. [online], 2000 [cit. 2011-12-22], Folding@Home distributed computing.
URL http://folding.stanford.edu/

[25] SOIFER, V.; POTAMAN, V.: *Triple-helical nucleic acids*. Spinger, 1996, ISBN 9780387944951.

[26] STOKES, J.: SIMD architectures. [online], 2001 [cit. 2012-01-06].
URL http://arstechnica.com/old/content/2000/03/simd.ars/

[27] UCSC Genome Bioinformatics Group: UCSC Genome Bioinformatics. [online], 2012 [cit. 2011-12-27].
URL http://genome.ucsc.edu

[28] WATSON, J. D.; CRICK, F. H.: Molecular structure of nucleic acids; a structure for deoxyribose nucleic acid. *Nature*, ročník 171, duben 1953: s. 737–738.

Příloha A

Obsah CD

./doc	-	programová dokumentace
./info	-	informace o používaných zařízeních
./input	-	ukázkové vstupní soubory
./src	-	zdrojové soubory projektu
./test	-	automatizované testy programu
./text	-	zdrojové soubory textové verze práce

Příloha B

Testovací sestava

Informace o sestavě

hardware	
typ:	HP EliteBook 8560w
procesor:	Intel Core i7 2670QM, 2.2Ghz, 4 jádra/8 virtuálně
paměť:	10GB DDR3, 1800Mhz, dual channel

GPU	
typ:	Nvidia Quadro 1000M
jádra:	96 CUDA jader
procesory:	2 streaming multiprocessory
paměťová sběrnice:	128-bitová
paměť:	2048MB DDR3, dedikovaná pamět
propustnost paměti:	28.8GB/sec
výpočetní kompatibiita:	2.1

software	
operační systém:	MS Windows 7 Professional, 64bit
SDK:	CUDA SDK 4.10.124.1345
ovládač GPU:	Nvidia Quadro 301.24 Beta
debugger:	Nvidia Parallel Nsight 2.2.0.12097 Beta

Detailní informace o grafické kartě

```
Device: "Quadro 1000M"
  CUDA Driver Version / Runtime Version          4.1 / 4.1
  CUDA Capability Major/Minor version number:    2.1
  Total amount of global memory:                 2048 MBytes (2147483648 bytes)
  ( 2) Multiprocessors x (48) CUDA Cores/MP:     96 CUDA Cores
  GPU Clock Speed:                               1.40 GHz
  Memory Clock rate:                             900.00 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 131072 bytes
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Concurrent copy and execution:                 Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Concurrent kernel execution:                   Yes
  Alignment requirement for Surfaces:            Yes

  CUDA Driver = CUDART, CUDA Driver Version = 4.1,
  CUDA Runtime Version = 4.1, NumDevs = 1, Device = Quadro 1000M
```

Příloha C

Manuál

Požadavky a překlad

Soubor projektu je možné nalézt ve složce *./src*. Projekt je konfigurován pro prostředí Microsoft Visual Studio 2010. Ke korektnímu překladu je nutné vlastnit grafickou kartu Nvidia podporující s architekturou Fermi či novější, nainstalované CUDA SDK 4.1 nebo novější a vývojářskou verzi ovládačů grafické karty kompatibilní s nainstalovaným SDK a dostupnou kartou. Vyžadován je překlad na 64 bitové architektuře z důvodu korektní adresace všech spouštěných výpočetních bloků.

Vstup

Program ve vstupním souboru očekává textový soubor se sekvencí nukleotidů zapsaných dle zvyklosti dle prvního písmene. Velikost písmen nehraje roli.

Ukázka vstupu

```
tttctcctatcttcttcctcggg
```

Parametry vstupu

–a	-	nastavení parametrů Lambda 0,1,4,5
–b	-	nastavení parametrů Lambda 2,3,6,7
–c	-	nastavení parametrů Mi 0,1,4,5
–d	-	nastavení parametrů Mi 2,3,6,7
–e	-	minimální hodnota skóre
–f	-	název vstupního souboru
–i	-	minimální délka smyčky
–j	-	maximální délka smyčky
–l	-	minimální délka triplexu
–p	-	hodnota *p value*
–s	-	maximální počet kroků diagonálně
–t	-	typ triplexu
–v	-	výpis hlášení
–x	-	verze spuštění (h – hardware, s – software, hd – hw debug, sd – software debug)

Výstup

Program vypisuje výsledky na standardní výstup. Každý nalezený triplex se nachází na jednom řádku. Jsou postupně vypsány jeho vlastnosti a následně sekvence nukleotidů.

Ukázka výstupu

```
E    1   12   13   23   5   11   5.706780e-001    1 tttctcctatct--tcttcctcggg
```

Vlastnosti výstupu

E	-	"exported", znamená triplex nalezený ve standardním běhu algoritmu
1	-	startovní pozice triplexu
12	-	pozice začátku mezery
13	-	pozice konce mezery
23	-	koncová pozice triplexu
5	-	získané skóre
11	-	délka triplexu
5.70...	-	p-hodnota
1	-	počet insercí/delecí

Upozornění ke korektnosti výstupu

Při výstupu z programu je vždy vhodné nasměrovat výsledek do souboru. Výpisy přímo do konzole bývají problematické z důvodu paralelního přístupu. Na výstupu tak vznikají oblasti prázdných znaků, na místo exportovaných triplexů. Tato vlastnost je způsobena problémovou implementací[1] funkce *printf()* v CUDA API. Výstup do souboru by měl vždy poskytovat korektní výstup. Je vhodné jej seřadit dle zvolených parametrů jelikož při paralelním zpracování se výsledky zapisují dle aktuálně zpracovávaných vláken. Z pohledu výstupu tedy náhodně v porovnání se sekvenční verzí programu.

[1] tato funkce byla zpřístupněna až v nových verzích API (> 2.0), důvodem je složitost dynamické alokace na streaming multiprocessorech

Příloha D

Ukázková konfigurace a její výstup

Vstup

example.txt:

```
tttctcctatcttcttcctcggg
```

Spuštění

```
DIP.EXE -f example.txt -s 20 -e 4 -l 4 -i 0 -j 0 -t 2 -p 1 -x h
```

Výstup

```
E    1   12   13   23   5   11   5.706780e-001    1 tttctcctatct--tcttcctcggg
E    2    8   10   16   5    7   5.706780e-001    0 ttctcct-a-tcttctt
E    4   12   13   20   7    8   3.046374e-001    1 ctcctatct--tcttcctc
E   11   15   16   20   4    5   6.237916e-001    0 cttct--tcctc
```