MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Akcelerace hledání palindromů s využitím GPU

BAKALÁŘSKÁ PRÁCE

Lukáš Skovajsa

Brno, jaro 2009

Prohlášení

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

Vedoucí práce: Ing. Matej Lexa, Ph.D.

Shrnutí

Tato bakalářská práce se zabývá hledáním přibližných palindromů v dlouhých řetězcích pomocí GPU. V první kapitole je popsaný význam palindromů v biologii. Ve druhé kapitole je popsaná CUDA. Ve třetí kapitole je vysvětlený algoritmus na hledání přibližných palindromů, který je vytvořený pomocí metody dynamického programování. Ve čtvrté kapitole je odvozený řadící algoritmus, který je použitý pro seřazení palindromů podle délky. V poslední kapitole je porovnání rychlostí algoritmů na CPU a GPU.

Klíčová slova

CUDA, přibližné palindromy, akcelerace hledání přibližných palindromů, dynamické programování, akcelerace řazení

Obsah

Kapitola 1

Úvod

Nositelem genetické informace v živých buňkách jsou molekuly DNA, které se typicky vyskytují jako šroubovice složená z dvou vláken nukleotidů. V DNA se vyskytují čtyři různé nukleové báze. Tyto čtyři báze se dělí na dvě purinové a dvě pyrimidinové báze. Purinové báze jsou adenin (A) a guanin (G). Pyrimidinové báze jsou thymin (T) a cytosin (C). Vazbu tvoří pouze adenin s thyminem (A–T) a guanin s cytosinem (C–G). Lidská DNA je obsažená v každé buňce v jádru a obsahuje přibližně tři miliardy vazebných párů. DNA slouží k uchování informací o proteinech. [1] Tato informace je soustředěná do genů, kde o podobě proteinu rozhoduje posloupnost nukleotidů. DNA na délku měří přibližně dva metry. Zvláštní kombinace nukleotidů v DNA můžou také vytvářet jiné struktury nebo plnit jiné funkce. Typickým příkladem je poměrně častý výskyt palindromických sekvencí. Palindrom v DNA může být textový, ve kterém jsou báze na jedné šroubovici zrcadlově symetrické kolem nějakého středu, nebo může být komplementární, ve kterém báze na jedné šroubovici jsou komplementární kolem nějakého středu. Textový palindrom je například ACGTAATGCA a komplementární palindrom je například ACGTATACGT.

Na úvod popíšu některé procesy, kterými informace uložená v DNA musí projít předtím, než vznikne protein. Dále uvedu příklady, ve kterých mají palindromy v DNA biologický význam.

K vytváření proteinů slouží RNA, která vznikne z DNA při procesu přepis, při kterém se zkopíruje část DNA (gen) do makro–molekuly RNA. Při tomto procesu musí dojít k rozpojení dvojšroubovice DNA, zkopírování informace a opětovnému spojení dvojšroubovice DNA. RNA obsahuje stejné nukleotidy jako gen v DNA. Výjimka je pouze thymin(T), který je nahrazený uracilem(U). Následně RNA opustí jádro buňky a kolem ní se sestaví biologický stroj (ribozom), který vytvoří protein podle informace, která je obsažená v RNA. Ribozom staví protein z aminokyselin, které přináší molekuly tRNA. Zkratka tRNA znamená *transfer* RNA. Pro každou aminokyselinu existuje více tRNA, ale každá tRNA může přenášet pouze jednu aminokyselinu. Aminokyselin je dvacet. Ribozom čte najednou tři znaky z RNA a podle nich vybere správnou aminokyselinu, kterou připojí do rostoucího proteinového řetězce. Ribozomy vyrábějí všechny proteiny. Záleží pouze na RNA, který protein vznikne. [2] [3]

V prostoru vypadá molekula tRNA jako písmeno L a obvykle ji tvoří 74 až 95 nukleotidů. Možný rozpoznávací znak pro tRNA v DNA je ten, že v DNA po sobě následuje několik komplementárních palindromů s malými mezerami. Při hledání těchto palindromů musíme vzít v úvahu smyčky, které vzniknou díky ohnutím v tRNA. [4]

V každé buňce nemůžou být pořád aktivní všechny geny, které jsou obsažené v DNA. K jejich regulaci slouží určité proteiny, které se vážou na DNA. Regulační proteiny mají schopnost najít určitou část v sekvenci DNA a připojit se k ní. Mnoho přípojných míst má strukturu palindromu, protože se na ně váží proteiny, které jsou symetrické.

DNA může vytvářet i jiné struktury než dvojšroubovice, ve kterých hrají důležitou úlohu palindromy. Tyto struktury jsou *křížová DNA* a *triplex DNA*. [5]

Vodíkové můstky z části dvojšroubovic se musí rozpadnout, aby se mohl vytvořit křížový tvar křížové DNA. Pro toto uspořádání musí být v DNA komplementární palindrom. Ve smyčce vždy budou tři až čtyři nespárované báze, což má za následek menší stabilitu tohoto uspořádání. Prozatím je známých pouze několik biologických významů křížové DNA, ale komplementární palindromy v lineárním uspořádání hrají důležitou roli jako vazebná místa pro dymetrické proteiny. Křížová DNA je spojovaná s místy, ve kterých DNA začíná kopírování sebe sama. [5]

K vytvoření struktury triplex DNA je zapotřebí tří vláken nukleových bází. Frank Kamenetskii a spolupracovníci ukázali, že DNA musí obsahovat komplementární palindrom, aby mohlo dojít k vytvoření triplex DNA. K vytvoření triplex DNA jsou zapotřebí ještě další podmínky. Lidská DNA má potenciál k vytvoření těchto struktur a tyto oblasti jsou běžně spojované s regulačními oblastmi genů. Triplex DNA možná může také pracovat jako ukončovací značka při kopírování DNA. [5]

V buňkách se také vyvinul postup, který ničí invazní viry pomocí enzymů, které rozřežou cizí DNA na určité pozici. Domácí DNA je chráněná jiným enzymem během tohoto procesu. Tyto enzymy jsou nyní používané v laboratořích k práci s DNA. Také toto místo řezu má často strukturu palindromu. [6]

Palindromy tak plní více biologických funkcí a jsou v zájmu molekulárních biologů. V současné době jsou přečtené sekvence nukleotidů v genomech různých organismů a je potřebné vyhledávat palindromy v těchto datech. V době psaní práce neexistovala GPU implementace pro hledání palindromů, ale existovala implementace pro hledání optimálního zarovnání. [12]. Cílem této práce bylo vytvořit takovou implementaci.

Kapitola 2

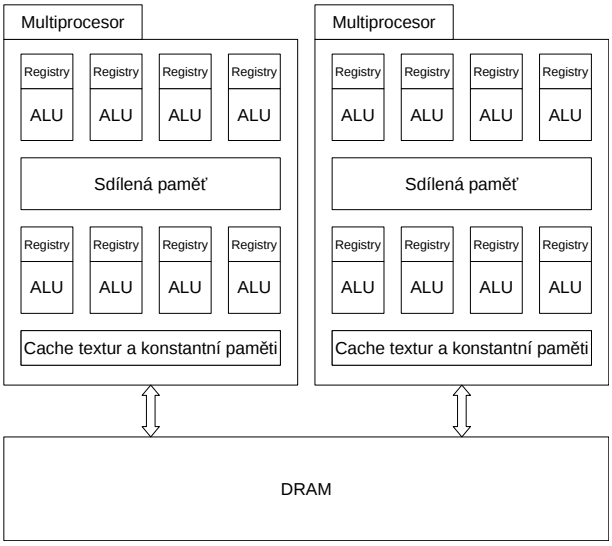
CUDA

CUDA je model paralelního programování, hardwarová architektura a poskytuje knihovny a programovací jazyk pro vývoj aplikací. CUDA je zkratka z *Compute Unified Device Architecture* a byla vyvinutá společností NVIDIA.[1] CUDA byla navržená, aby zjednodušila vytváření programů pro GPU a používá imperativní programovací jazyk C s rozšířeními, která dovolují nastavovat konfiguraci, kopírovat paměť, synchronizovat a spouštět výpočty na GPU. Architektura CUDA bude v budoucnosti podporovat i jiné programovací jazyky. Například C++, OpenCL, Fortran, DirectX Compute, ... Ve vývoji je i neoficiální programovací jazyk Obsidian. Obsidian je rozšíření funkcionálního programovacího jazyka Haskell. Obsidian by měl zjednodušit programátorovi rozhodnutí, do které paměti v paměťovém modelu umístit data. Model rozlišuje dva druhy paralelismů. Jemnozrnný a hrubozrnný. Tento model je vhodný jenom pro problémy, které se dají rozdělit na menší podproblémy, které jsou vzájemně nezávislé. Toto je hrubozrnný paralelismus. Jemnozrnný paralelismus znamená, že části podproblému můžou být vzájemně závislé. Výhodou rozdělení na jemnozrnný a hrubozrnný paralelismus je, že dovoluje vytvářet programy, které dokáží využít prostředky právě instalované GPU. Jednou zkompilovaný program je spustitelný na GPU s libovolným počtem multiprocesorů a části výpočtu se rozloží na všechny přítomné multiprocesory. GPU má mnohem větší výpočetní výkon než CPU z velké části díky tomu, že u zapisovatelných pamětí nemá vyrovnávací paměť. Nepřítomnost vyrovnávací paměti uvolňuje místo pro další výpočetní jednotky, ale zvyšuje nároky na programátora, který musí důkladně promýšlet paměťový model svého programu. Pro výpočet na GPU jsou nejvhodnější problémy, které mají vysoký poměr mezi aritmetickou složitostí a počtem přístupů do zapisovatelné paměti. Doba těchto přístupů je zamaskovaná plánovačem, který může během doby, která je potřebná pro zapsání, nebo přečtení dat ze zapisovatelné paměti, zpracovávat jiný podproblém.

2.1 Hardware

GPU, která podporuje běh CUDA programů, obsahuje sadu SIMT multiprocesorů a různé druhy pamětí. Každý SIMT multiprocesor obsahuje osm výpočetních jednotek, které provádějí v každém kroku stejnou instrukci, sdílenou paměť, sadu registrů, vyrovnávací paměť konstantní paměti, která je umístěná v globální paměti (konstantní paměť) a vyrovnávací paměť texturové paměti, která je také umístěná v globální paměti (texturová). Každý multiprocesor provádí výpočet po dávkách 32 vláken, protože zpracování instrukce je rozdělené do čtyř fází. Dávkám se říká *warps*. Označení *half–warp* je buď vrchní, nebo spodní polovina warpu. Vlákna, která tvoří jeden warp, začínají výpočet v programu na stejném místě, ale můžou se při větvení programu rozejít rozdílnými větvemi. Výpočetní jednotka neprovádí

1. Podrobnější informace lze najít v [9].



Obrázek 2.1: CUDA architektura

žádnou operaci, jestliže právě prováděná instrukce na multiprocesoru neodpovídá instrukci, kterou má vlákno provádět. Nejlépe je multiprocesor využitý, když celý warp provádí po celou dobu běhu stejnou instrukci. Multiprocesor postupně provede všechny větve, jestliže vlákna ve warpu postupují rozdílnými větvemi po nějaké podmínce. Za koncem podmíněným větvením nemusí probíhat synchronizace vláken, protože multiprocesor bude pokračovat společným kódem, až budou všechny rozdílné větve provedené. SIMT architektura je podobá SIMD architektuře v tom, že více výpočetních jednotek provádí jednu instrukci, ale je rozdílná v tom, že u SIMD architektury musí programátor ošetřovat podmíněné větvení. Každé vlákno má tedy přidělený vlastní čítač instrukcí a může být prováděné nezávisle na ostatních vláknech. Při kontrole správnosti navrhovaného algoritmu programátor nemusí brát brát v úvahu architekturu SIMT a postupovat stejně jako při kontrole sériového kódu, ale nevyužije všech osmi výpočetních jednotek na multiprocesoru.

Místo čekání na vystavení hodnoty z globální paměti může multiprocesor přepnout aktuální warp za jiný. Počet aktivních warpů na jednom multiprocesoru závisí na množství sdílené paměti a počtu registrů, které warpy potřebují pro svůj běh. Počet a pořadí zápisů je nedefinovaný, jestliže více neatomických instrukcí zapisuje na stejné paměťové místo ve stejný čas. Jedna instrukce ale vždy uspěje. Pořadí zápisů je nedefinovaný, jestliže více atomických instrukcí zapisuje na stejné místo ve stejný čas, ale zápisy se provedou vždy všechny. Na obrázku 2.1 je znázorněná architektura CUDA grafických karet.

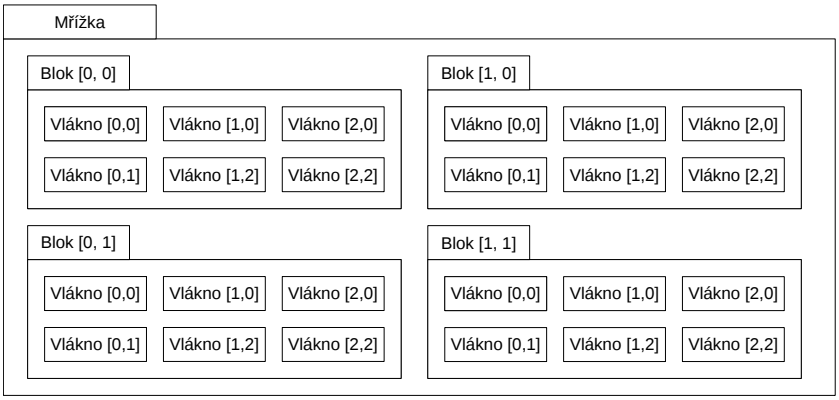
2.2 Programovací model

CUDA rozšíření jazyka C dovoluje spouštět stejnou funkci paralelně. Označení této funkce je *kernel*. Každé vzniklé vlákno může za běhu zjistit svoji identifikaci, aby každé vlákno mohlo vykonávat program na jiných datech. Ve zdrojovém kódu je funkce, která je kernel, definovaná pomocí *__global__* a její návratový typ je vždy *void*. V kernelu je možné volat funkci, která je definovaná pomoci *__device__*, ale tato funkce nesmí být rekurzivní. Počet vláken N, které budou provádět kernel, je určený pomocí špičatých závorek $<<< \ldots >>>$. Na obrázku 2.2 je příklad programu.

```
__global__ void f(void) {
}

int main(void) {
        f<<<1, N>>>();
        return 0;
}
```

Obrázek 2.2: příklad programu v syntaxi CUDA C



Obrázek 2.3: příklad hierarchie vláken s dvojrozměrnou mřížkou i dvojrozměrnými bloky

2.2.1 Hierarchie vláken

Programátor musí rozdělit vlákna do bloků. Tyto bloky musí být navzájem výpočetně nezávislé a musí být možné je zpracovávat v libovolném pořadí, aby byl možný hrubozrnný paralelismus. Rozdělení na bloky neodpovídá počtu multiprocesorů na GPU, ale spíše odpovídá struktuře dat. Počet bloků by měl významně převyšovat počet multiprocesorů, aby každý multiprocesor mohl zpracovávat blok, který je připravený na zpracování. Blok nemusí být připravený na výpočet, jestliže čeká na vystavení hodnoty z globální paměti. Multiprocesor si blok rozdělí na menší části (warpy) a v jeden určitý okamžik multiprocesor paralelně zpracovává pouze jeden warp. Vnitřní logická struktura bloku může být jednorozměrná, dvojrozměrná, nebo trojrozměrná. Struktura bloku odpovídá typu problému. Například u násobení matic je přirozené mít pro každou výslednou hodnotu jedno vlákno a strukturu bloku zvolit dvojrozměrnou. Identifikace vlákna je potom stejná jako identifikace výsledné hodnoty, jejíž pozice v matici je určená číslem řádku a sloupce. Bloky jsou části logické struktury mřížka (*grid*), která může být jednorozměrná, nebo dvojrozměrná. Na obrázku 2.3 je zobrazený příklad hierarchie vláken.

Ke spolupráci vláken je zapotřebí synchronizace. Vlákna v bloku se synchronizují pomocí funkce *__syncthreads()*, která se chová jako bariéra. Všechna vlákna v bloku musí dosáhnout tohoto bodu, než můžou dál pokračovat. Multiprocesor implementuje *__syncthreads()* pomocí jedné instrukce. Synchronizace vláken je velice rychlá, jestliže žádné vlákno nemusí čekat na žádné jiné vlákno. Synchronizace vláken také způsobí, že veškeré zápisy do sdílené paměti a globální paměti budou správně viditelné ostatním vláknům v bloku. Synchronizace vláken je povolená v podmíněném větvení, ale všechna vlákna musí jít stejnou větví, protože jinak

dojde k uváznutí. V první větvi by vlákna čekala na bariéře na vlákna, která čekají na bariéře v druhé větvi na vlákna z první větve.

2.2.2 Typy pamětí

CUDA architektura obsahuje více druhů pamětí, které se liší maximální velikostí, dobou přístupu, zapisovatelností a jestli mají vyrovnávací paměť. Každé vlákno má vlastní lokální paměť, která je ale také fyzicky umístěná v globální paměti. Lokální paměť není lokální ve smyslu, že je umístěná na multiprocesoru, ale že patří jednomu vláknu. Vlákna v jednom bloku mají přístup do sdílené paměti. Sdílená paměť je rozdělená do šestnácti částí (*banks*). Do části i patří adresy ($addr$), pro které platí $\left\lfloor \frac{addr}{4} \right\rfloor \mod 16 = i$. Požadavky do různých části jsou obsloužené paralelně. Přístupy vláken do sdílené paměti musí být obsloužené postupně, jestliže přistupují na adresy, které patří do stejné části. Programátor musí při návrhu kernelu minimalizovat tyto konflikty, protože způsobují ztrátu výkonu. Velikost sdílené paměti je pouze 16KB, což může být velké omezení, protože tato paměť většinou slouží i jako náhrada za vyrovnávací paměť globální paměti. Každé vlákno má přístup do globální paměti, která má velkou odezvu. Pro přístup ke globální paměti musí vlákna dodržovat některá omezení, aby využila dostupnou propustnost sběrnice. Přístup do globální paměti může být společný pouze pro vlákna z jedné poloviny warpu. Přístup do globální paměti je jenom jeden, jestliže všechna vlákna, která patří do jedné poloviny warpu, přistupují na adresy, které patří do stejného segmentu. Tato podmínka je postačující pro GPU s výpočetními schopnostmi od verze 1.2. Velikost segmentu je různá pro proměnné, které mají rozdílný počet bitů. Velikost segmentu je

- 32 Bajtů, jestliže vlákna přistupují k 8-bitové proměnné.
- 64 Bajtů, jestliže vlákna přistupují k 16-bitové proměnné.
- 128 Bajtů, jestliže vlákna přistupuji k 32-bitové, nebo 64-bitové proměnné.

Konstantní paměť má vyrovnávací paměť a čtení z vyrovnávací paměti je přibližně stejné jako čtení z registru, jestliže všechna vlákna z jedné poloviny warpu čtou ze stejné adresy. Podle názvu je zřejmé, že tento typ paměti je určený pouze pro čtení. Texturová paměť má také vyrovnávací paměť. Hodnoty, který jsou uložené ve vyrovnávací paměti, jsou průběžně aktualizované podle dvojrozměrné prostorové lokality. Do vyrovnávací paměti se ukládají hodnoty z adres, které jsou blízké adresám, z kterých byly nedávno vystavené hodnoty.

Registry jsou nejrychlejší paměť, ale je jich málo. Při nedostatku registrů může překladač umístit proměnnou do lokální paměti. V tabulce 2.1 jsou zobrazené hlavní vlastnosti jednotlivých typů pamětí.

Typ paměti:	lokální	sdílená	globální	konstantní	texturová	registry
Rychlost:	pomalá	rychlá	pomalá	pomalá	pomalá	rychlá
Množství:	velké	malé	velké	malé	středně velké	velmi malé
Vyrovnávací p.:				×	×	
Zapisovatelná:	×	×	×			×

Tabulka 2.1: Typy pamětí

2.2.3 Rozhraní

Programátor může pro ovládání GPU použít dvě různá rozhraní. Nízkoúrovňové (*driver API*), nebo vysokoúrovňové (*runtime API*) a musí používat pouze jedno. Vysokoúrovňové rozhraní je postavené na nízkoúrovňovém a je také jednodušší. Jeho hlavní výhoda je, že program napsaný s využitím tohoto rozhraní se může přeložit a spouštět na CPU pomocí emulátoru. Pro každé vlákno v kernelu emulátor vytvoří jedno vlákno, které poběží na CPU. Každé takto vytvořené vlákno dostane automaticky 256KB velký zásobník. Emulátor je vhodný pouze pro testování s malým počtem vláken, protože už 4096 vláken spotřebuje 1GB operační paměti. Kernel může obsahovat funkce pro výpis na obrazovku nebo do souboru, jestliže je kód přeložený tak, aby byl spuštěný pomocí emulátoru. Bezchybný běh programu pomocí emulátoru nezaručuje bezchybný běh na GPU, protože problémy souběhu se častěji projeví pří větším počtu paralelně zpracovávaných vláken, na GPU existuje více typů pamětí, globální paměť GPU je jiná než globální paměť systému a operace s hodnotami, které mají plovoucí desetinnou čárku, mají většinou menší přesnost na GPU. Výhoda nízkoúrovňového rozhraní je, že je nezávislé na programovací jazyce. S jednou GPU může najednou pracovat více systémových procesů, ale jeden proces může najednou pracovat pouze s jednou GPU. Programátor musí vytvořit zvláštní vlákno pro každou GPU, kterou bude používat k výpočtům.

Kapitola 3

Hledání palindromů

Palindrom p je řetězec znaků, který se čte zepředu i zezadu stejně. Platí tedy $p = w \cdot w'$, jestliže řetězec obsahuje sudý počet znaků, nebo $p = w \cdot c \cdot w'$, jestliže řetězec obsahuje lichý počet znaků, kde $w' = x_{n-1} \cdot x_{n-2} \cdot \cdots x_0$ a x_i je i-tý znak řetězce w. Obtížnější je hledat přibližné palindromy ve kterých můžou být chyby. Za chybu se považuje, jestliže znaky na odpovídajících si místech v řetězcích w a w' jsou různé, nebo jeden z nich je v páru s mezerou. Tento problém se může řešit algoritmem dynamického programování.

3.1 Dynamické programování

Pojem dynamické programování poprvé použil v 50. letech 20. století matematik Richard Bellman. Dá se použít pokud řešení problému v sobě obsahuje řešení podproblémů. Dynamické programování se někdy nazývá memorování, protože algoritmus si ukládá jednou vypočítané výsledky, aby je mohl v budoucnu znovu použít a nemusel je znovu počítat. Mnoho problému, které se řeší dynamickým programováním, obsahuje navíc hodnotící funkci. V takovém případě algoritmus hledá maximum, nebo minimum této funkce.

Algoritmus dynamického programování se skládá ze čtyř částí:

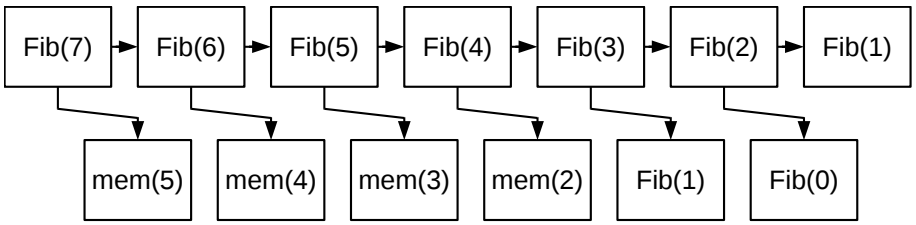
- Rekurzivní definice hodnotící funkce.
- Paměť pro zapamatování skóre již vypočítaných podproblémů.
- Postup vyplňování paměti odspodu nahoru.
- Postup zpětného dohledání cesty, která dává optimální řešení.

Jenom některé hodnotící funkce jsou vhodné k tomuto přístupu. Hodnotící funkce musí dovolovat, aby se řešení dalo rozdělit na nezávisle řešené části a aby optimální řešení problému obsahovalo optimální řešení některého z podproblémů. Jednoduchý příklad využití dynamického programování bez hodnotící funkce a zpětné dohledání cesty je algoritmus na nalezení Fibbonacciho čísla.

3.1.1 Fibonaccioho čísla

Definice Fibonacciho čísla je:

$$\begin{aligned} Fib(0) &= 0 \\ Fib(1) &= 1 \\ Fib(x) &= Fib(x-1) + Fib(x-2) \end{aligned}$$



Obrázek 3.1: postup vylepšeného algoritmu na výpočet Fibbonaciho čísla pomocí dynamického programování

Rekurzivní algoritmus, který je přímým přepisem definice Fibonacciho čísla je krátký a elegantní, ale je neefektivní, protože jeho časová složitost je exponenciální. Rekurzivní algoritmus:

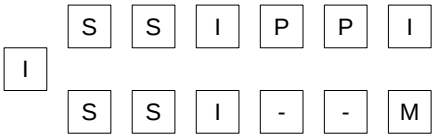
```
1  int Fib(int x) {
2          if (x == 0)
3                  return 0;
4          if (x == 1)
5                  return 1;
6          return Fib(x - 1) + Fib(x - 2);
7  }
```

Z definice plyne, že algoritmu stačí pro výpočet Fibonacciho čísla z čísla x znát všechna Fibonacciho čísla přirozených čísel menších než x. Algoritmus nebude muset přepočítávat jednou vypočítané Fibonacciho čísla, jestliže si je zapamatuje. Následující algoritmus bude vyplňovat pole inicializované na nuly hodnotami, které jsou Fibonacciho čísla indexů prvků pole. Hodnota prvku pole nula znamená, že tato hodnota ještě není vypočítaná. Algoritmus bude postupovat od menších čísel k větším a každou novou vypočítanou hodnotu uloží na příslušné místo do pole *mem*. Vylepšený algoritmus pomocí dynamického programování má lineární časovou složitost. Obrázek 3.1 ilustruje postup algoritmu pro $x = 7$.

```
1  int Fib(int x) {
2          if (mem[x] != 0)
3                  return mem[x];
4          if (x == 0)
5                  return 0;
6          if (x == 1)
7                  return 1;
8          mem[x] = Fib(x - 1) + Fib(x - 2);
9          return mem[x];
10 }
```

3.2 Algoritmus hledání palindromů pomocí dynamického programování

Algoritmus má na vstupu řetězec s a na výstupu minimální skóre pro všechny podřetězce vstupního řetězce, které je potřeba na zarovnání do palindromu a dostatek informací pro konstrukci tohoto zarovnání. Jako první je popsaný čistě rekurzivní algoritmus na hledání



Obrázek 3.2: palindrom z řetězce mississippi

palindromů a následně je rozšířený pomocí dynamického programování, protože má vysokou časovou náročnost a najde pouze skóre a zarovnání pro celý řetězec.

3.2.1 Rekurzivní algoritmus, rekurzivní definice

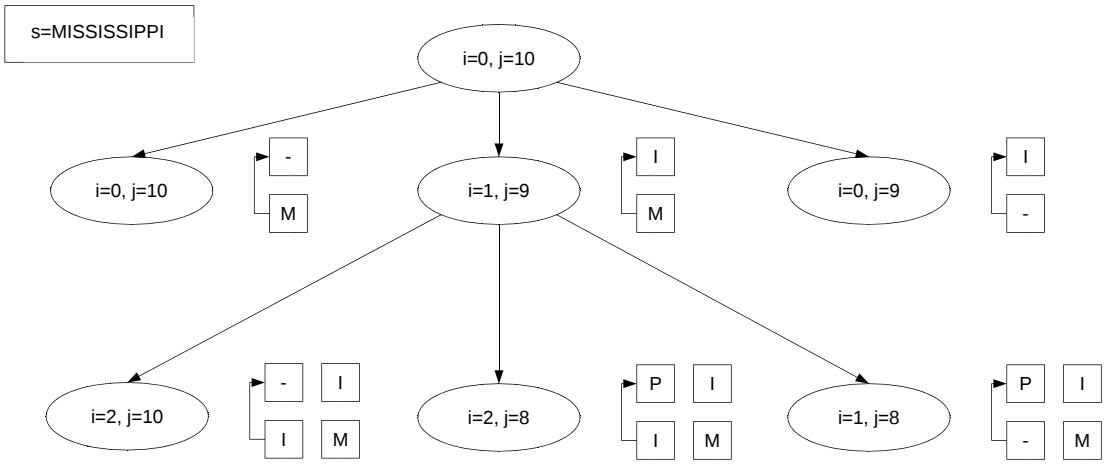
Algoritmus má na vstupu řetězec s a na výstupu nejmenší možné skóre. Palindrom nalezený tímto algoritmem obsahuje všechny znaky, které jsou ve vstupním řetězci. Ve všech možných palindromech vytvořených z řetězce s má každý znak za partnera buď jiný znak řetězce s, nebo speciální znak zastupující mezeru. Znaky se zpracovávají podle pořadí jaké mají v s. Algoritmus postupně ubírá znaky z obou konců řetězce a hledá řešení pro jednodušší podproblémy. Algoritmus má na výběr při zpracování řetězce s ze čtyř možností.

- První a poslední znak v řetězci s budou v páru a dále se bude zpracovávat řetězec o první a poslední znak kratší (nový s).
- První znak v řetězci s bude v páru s mezerou a dále se bude zpracovávat řetězec o první znak kratší (nový s).
- Poslední znak v řetězci s bude v páru s mezerou a dále se bude zpracovávat řetězec o poslední znak kratší (nový s).
- Řetězec s bude prázdný. Nebude už dál zanořovat.

Algoritmus 'hádá' zarovnání řetězce do palindromu, zkouší všechny možnosti. Touto rekurzí vznikne strom s pevným větvením tři. V tomto stromu určitě existuje cesta, která popisuje optimální zarovnání řetězce do palindromu. Skóre palindromu je počet dvojic, které neobsahují stejný znak. Skóre palindromu tedy odpovídá počtu chyb. Algoritmus musí při vynořování zpět ke kořeni mít nějakou hodnotící funkci S, podle které bude rozhodovat, která cesta popisuje řešení s nejmenší chybou. Algoritmus si bude vybírat cestu s nejmenším skóre. Rekurzivní definice funkce, která počítá skóre palindromu:

$$S(i,j) = \min \begin{cases} S(i+1, j-1) + \sigma(x_i, x_j) \\ S(i+1, j) + 1 \\ S(i, j-1) + 1 \end{cases} \qquad \sigma(x_i, x_j) = \begin{cases} 0, & x_i = x_j \\ 1, & x_i \neq x_j \end{cases}$$

Kde i, j značí pozici v řetězci s a x_i, x_j odpovídající znak. $S(i,j)$ označuje skóre podřetězec řetězce s, který začíná na pozici i a končí na pozici j. Skóre $S(i,j)$ je nulové pro podřetězce, které obsahují pouze jeden znak, jestliže se tedy $i = j$. Skóre $S(i,j)$ je také nulové pro prázdné podřetězce, jestliže tedy $i > j$. Algoritmus vypočítá správnou minimální chybu,



Obrázek 3.3: malá část rekurze

která je potřebná pro zarovnání řetězce s do přibližného palindromu, protože výpočet skóre je čistě aditivní. Možná obětʼalgoritmu vybráním neminimálního skóre by se nikdy nevrátila v nižších zanořeních rekurze díky aditivnosti hodnotící funkce. Tento algoritmus je vhodný jenom pro malé řetězce, protože patří do časové složitostní třídy $\theta(2^n)$ [1], kde n je délka řetězce s. Tento algoritmus je exponenciální. Všeobecně se má zato, že prakticky použitelné algoritmy mají polynomickou časovou složitost, proto tento algoritmus není prakticky použitelný. Prostorovou složitostí patří do třídy $\theta(n)$. Algoritmu stačí mít uložené zarovnání do palindromu s nejmenší dosud nalezenou chybou. Při nalezení zarovnání s menší chybou, než měl dosud uloženou, přepíše původní zarovnání novým. Nevýhoda tohoto postupu je, že po skončení algoritmu bude k dispozici pouze nejlepší zarovnání řetězce do palindromu. Výhodnější by bylo znát i ostatní uspořádání řetězce do palindromu, ale to už se v lineárním prostoru nedá udělat.

3.2.2 Matice pro zapamatování skóre již vypočítaných podproblémů

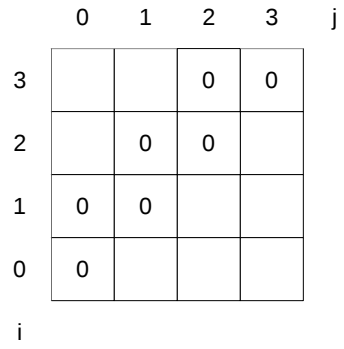
Z definice hodnotící funkce plyne, že počet různých skóre může být maximálně n^2. Z toho počtu navíc přibližně polovina nemá žádný smysl. $S(i, j)$ nemá smysl, jestliže $i > j + 1$. Taková hodnota by znamenala, že začátek podřetězce leží za koncem podřetězce ve vstupním řetězci s.

V exponenciálním rekurzivním algoritmem se mnoho výpočtů skóre provádí zbytečně. Algoritmus by byl o hodně efektivnější, kdyby měl možnost zjisti, jestli už někdy počítal skóre pro podřetězec daný i a j. Vhodná datová struktura pro vyhledávání podle dvou indexů je dvojrozměrná matice. V matici index i označuje řádek a j označuje sloupec. Řádky se číslují od spodního k vrchnímu a sloupce od levého k pravému. Směr číslování je zvolený tak, aby oblast pod antidiagonálou označovala hodnoty $S(i, j)$, které algoritmus bude ukládat a

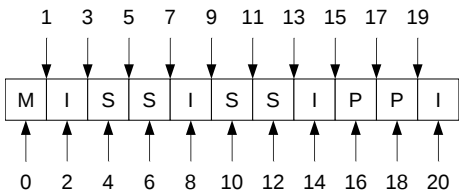
1. základ pro tento algoritmus je dokonce 3

opakovaně používat. Lepší datovou strukturu pro tento problém nelze najít, protože přístup k prvku $S(i, j)$ se znalostí i a j bude trvat jenom konstantní čas. Tato matice je hlavní rozdíl mezi obyčejnou rekurzí a dynamickým programováním.

$S(i, j)$ označuje skóre prázdného řetězce, jestliže $i = j + 1$ a $S(i, j)$ označuje skóre podřetězce, který má jenom jeden znak, jestliže $i = j$. Důsledkem je, že každé políčko na hlavní antidiagonále a antidiagonále nad ní označuje střed nějakého podřetězce vstupního řetězce s a každý střed podřetězce vstupního řetězce s má své odpovídající políčko na hlavní antidiagonále, nebo na antidiagonále nad ní. Diagonálám v této matici lze přiřadit čísla, aby odpovídala středům, které na nich leží. Středy podřetězců budou očíslovány, jak ukazuje obrázek 3.5, jestliže levá spodní diagonála bude mít nejnižší číslo a každá další sousední o jedno větší.



Obrázek 3.4: příklad matice



Obrázek 3.5: čísla středů a diagonál

3.2.3 Postup vyplňování matice odspodu na horu

Algoritmus postupně vyplňuje matici od nejmenších podproblému k těm větším. V inicializační části si vyplní políčka na souřadnicích odpovídajících prázdnému podřetězci a podřetězci tvořenému pouze jedním znakem. Těmto podřetězcům odpovídá skóre nula. Z definice hodnotící funkce plyne, že její hodnota je závislá na třech položkách, které jsou uložené v matici. Algoritmus může postupovat po řádcích od nejvyššího po nejnižší, protože pro vyplnění políčka $S(i, j)$ potřebuje pouze znát políčko nalevo od vyplňovaného ($S(i, j - 1)$), políčko nad vyplňovaným ($S(i - 1, j)$) a políčko s kterým vyplňované sdílí levý horní roh ($S(i - 1, j - 1)$). Dále potřebuje znát znaky na pozici i a j, kvůli funkci σ.

3.2.4 Zpětné dohledání

Každé vyplněné políčko matice $t_{i,j}$ udává minimální počet chyb, který je potřeba na zarovnání podřetězce začínajícího na pozici i a končícího na pozici j. Minimální skóre zarovnání řetězce do palindromu je v matici na průniku řádku 0 a sloupce $n - 1$ (pravý dolní roh matice) , protože algoritmus hledal zarovnání celého řetězce a byl spuštěný s parametrem i nastaveným na 0 a s parametrem j nastaveným na $n - 1$. Algoritmus může z matice sestavit zarovnání řetězce do palindromu tak, že bude zjišťovat z kterého políčka se dostal na aktuální políčko. Tento proces začne v pravém spodním rohu. Algoritmus bude postupovat

	M	I	S	S	I	S	S	I	P	P	I
I										0	0
P									0	0	1
P								0	0	0	1
I							0	0	1	1	0
S						0	0	1	1	2	1
S					0	0	0	1	2	2	2
I				0	0	1	1	0	1	2	2
S			0	0	1	0	1	1	1	2	3
S		0	0	0	1	1	0	1	2	2	3
I	0	0	1	1	0	1	1	0	1	2	2
M	0	1	1	2	1	1	2	1	1	2	3

Obrázek 3.6: vyplněná matice podle vstupního řetězce mississippi

podle definice hodnotící funkce. Pokaždé zjistí, které ze tří políček použil pro výpočet aktuálního a na výstup zapíše odpovídající znaky. Jestliže použil políčko $t_{i+1,j}$ na výstup vypíše uspořádanou dvojici $\binom{-}{x_i}$. Jestliže použil políčko $t_{i,j-1}$ na výstup vypíše uspořádanou dvojici $\binom{x_j}{-}$. Jestliže použil políčko $t_{i+1,j-1}$ na výstup vypíše uspořádanou dvojici $\binom{x_j}{x_i}$. Vypisovanou dvojici vždy vypíše před dvojici, kterou vypsal v předchozím kroku. Palindrom bude lichý, jestliže výpočet skončí na antidiagonále. V takovém případě algoritmus navíc vypíše znak x_i. Palindrom bude sudý, jestliže výpočet skončí za antidiagonálou. Pro představu postupu práce algoritmu je vhodné si nad matici a vedle matice napsat vstupní řetězec. Ke každému řádku i napsat odpovídající znak z řetězce s x_i a ke každému sloupci j napsat odpovídající znak z řetězce s x_j. Díky tomu jde bez odpočítávání pozice ve vstupním řetězci vidět, které znaky algoritmus porovnává, nebo dává na výstup.

3.3 Jednoduchý algoritmus

Předchozí řešení má nevýhodu, že jeho prostorová složitost patří do n^2. Pro $n = 10^6$ a $4B$ políčko by matice zabírala přibližně $3.6TB$ paměti. Tabulka by zabírala prostor v řádu TB i po optimalizaci, ve které by políčka nad antidiagonálou nebyla uložená v paměti. Taková paměťová složitost nedovoluje přímé použití tohoto algoritmu. Dále je popsaný možný způsob, kterým se dá snížit prostorová složitost.

3.3.1 Tabulka pro zapamatování výsledků podproblémů

Předcházející matice popisuje funkci $f(i, j) \rightarrow e$, kde i, j jsou souřadnice v matici a e je počet chyb. Políčko, které je popsané v matici souřadnicemi i, j lze popsat pozicí na diagonále $diag$ a vzdáleností od diagonály k políčku r. Místo funkce $f(i, j) \rightarrow e$ může algoritmus používat funkci $g(diag, r) \rightarrow e$. Funkční hodnota funkce g je minimální počet chyb, který je potřeba na zarovnání řetězce do palindromu. Tento řetězec je popsaný svým středem $diag$ ve vstupním

řetězci s a počtem znaků, který ho tvoří nalevo a napravo od středu r. Tímto převodem se paměťová náročnost algoritmu nesnížila, ale je to výchozí pozice pro závěrečnou úpravu.

Následující převod používá pan Allison ve svém algoritmu. [11] Výhodnější funkce než $g(diag, r) \rightarrow e$ je $h(diag, e) \rightarrow r$. Funkce h je popsaná tabulkou, ve které e určuje řádek a $diag$ sloupec. Funkční hodnota funkce h je maximální poloměr řetězce, který lze zarovnat do palindromu se středem v $diag$ a chybou e. Porovnání prostorové složitosti funkce g a h:

Funkce g zabírá přibližně $\frac{n^2}{2}$ místa v pamětí. Funkce h zabírá přibližně $k(2n)$ místa v paměti, kde k je počet chyb n je délka vstupního řetězce s.

$$\begin{aligned} 2kn &= \frac{n^2}{2} \\ k &= \frac{n}{4} \end{aligned}$$

Z rovnice plyne, že funkce h zabírá méně místa pro $k < \frac{n}{4}$. Ve většině případů je nezajímavé hledat zarovnání do palindromu s chybou větší než k. U funkce h jde také velice jednoduše měnit spotřeba paměti pro výpočet parametrem k. Spotřeba paměti funkce h tedy patří do $\theta(kn)$. Je také přirozenější klást dotaz na maximální poloměr, pro který je chyba rovna e na nějaké pozici ve vstupním řetězci s. Algoritmus může jednoduše odpovídat na dotaz, jaký je nejdelší palindrom a na jeho pozici, jestliže chyba je povolená e. Algoritmu stačí seřadit podle velikosti určený řádek v tabulce a odpovědět na dotaz.

3.3.2 Poloměr podřetězce, vzdálenost od antidiagonály

Nechť p je podřetězec vstupního řetězce s, který je určený dvojicí $(diag, d)$. Tohle určení podřetězce je možné, protože $(diag, d)$ určuje políčko v matici dynamického programování a každé takové políčko určuje nějaký podřetězec. Odůvodnění je u definice rekurze $S(i, j)$. Vzdálenost d od antidiagonály a poloměr r podřetězce p jsou stejné hodnoty, protože postupem po diagonále algoritmus vždy vkládá dva znaky z podřetězce do vytvářeného palindromu a po dosažení políčka $(diag, d)$ jsou všechny znaky podřetězce p použité.

3.3.3 Vyplnění tabulky zdola nahoru

Při inicializaci algoritmus vyplní nultý řádek tabulky, který odpovídá chybě nula. To znamená, že najde největší možný přesný palindrom pro každý střed ve vstupním řetězci s. Z pohledu matice dynamického programováni to znamená, že bude postupovat směrem na jihovýchod z pozice $diag$ dokud v odpovídajícím políčku bude nula. Délku této posloupnosti uloží do tabulky na místo $(0, diag)$.

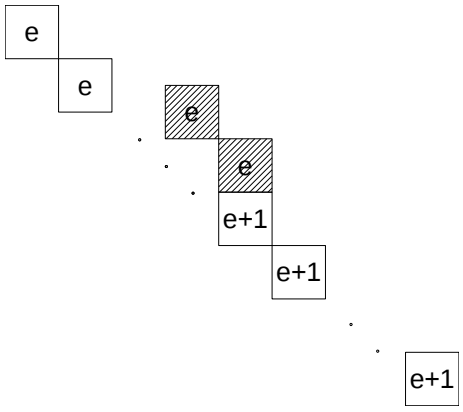
Každá diagonála v matici dynamického programování tvoří neklesající posloupnost protože se počet chyb nesnižuje. V tabulce jsou uložené jenom některé hodnoty. Algoritmus neukládá vzdálenosti, které jsou menší než maximální možné pro určitou chybu. Algoritmus vyplňuje tabulku iterativně podle počtu chyb. Nechť má vyplněnou tabulku pro e chyb a je na řadě vyplnění řádku $e + 1$. Jeho cílem je zjistit největší vzdálenost, na které je v matici dynamického programování uložená hodnota $e + 1$.
Nechť l je největší hodnota z následujících tří:

- Algoritmus zjistí nejdelší vzdálenost od antidiagonály, která lze v sousední diagonále dosáhnout s chybou maximálně e. V této možnosti algoritmus vybere sousední diagonálu, která má číslo $diag + 1$. Algoritmus zvětší tuhle vzdálenost o jedna, jestliže $diag$

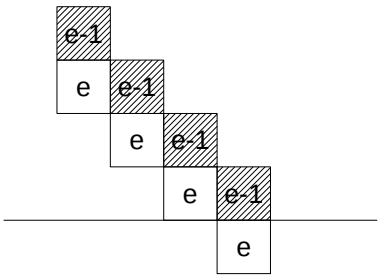
je liché číslo. Přičtení jedničky plyne z obrázku 3.6. Na obrázku 3.7 je příklad použití sousední diagonály.

- Algoritmus přičte k $(e, diag)$ jedničku. Přičtení jedničky odpovídá posunu na diagonále $diag$ o jedno políčko. Algoritmus tímto posunem vždy způsobí přidání chyby, protože v $(e, diag)$ je uložená největší možná hodnota, která se dá pomocí chyby e na diagonále $diag$ dosáhnout.
- Algoritmus zjistí nejdelší vzdálenost od antidiagonály, která lze v sousední diagonále dosáhnout s chybou maximálně e. V této možnosti algoritmus vybere sousední diagonálu, která má číslo $diag - 1$. Algoritmus zvětší tuhle vzdálenost o jedna, jestliže $diag$ je liché číslo.

Algoritmus si ke každé hodnotě poznamená možnost (směr), kterou použil k jejímu získání. Tyto směry jsou NW, jestliže použil diagonálu s o jedna menším číslem, $NORTH$, jestliže použil stejnou diagonálu a NE, jestliže použil diagonálu s o jedna větším číslem. Algoritmus použije tyto pomocné hodnoty při zarovnání řetězce do palindromu. Algoritmus musí zkontrolovat, jestli nepřekročil hranici matice dynamického programování. Zkontrolování a úprava v druhém bodu je jednoduchá. Při překročení hranice stačí odečíst od l jedničku a směr se nemusí měnit. V bodech jedna a tři musí provést složitější operace. Algoritmus také odečte jedničku od l, ale navíc si musí k l poznačit, že použil sousední diagonálu a musel výslednou hodnotu zmenšit o jedna. Tyto dvě značky jsou $NEDel$ a $NWDel$. Algoritmus použije značku $NEdel$, jestliže použil sousední diagonálu s o jedna větším číslem. Značku $NWDel$ použije, jestliže použil sousední diagonálu s o jedna menším číslem. Příklad takové situace je na obrázku 3.8. Algoritmus uloží do políčka $(e + 1, d)$ hodnotu l zvětšenou o počet



Obrázek 3.7: použití diagonály s o jedna větším číslem



Obrázek 3.8: přetažení spodního okraje matice dynamického programování

kroků, které může algoritmus provést na diagonále $diag$ ve vzdálenosti l bez přidání chyby. Přesnější popis tohoto kroku je následující:

- Algoritmus vypočítá pozici v matici dynamického programováni z $diag$ a l.

$$i = \left\lfloor \frac{diag}{2} \right\rfloor + l + 1 \tag{3.1}$$

$$j = \left\lfloor \frac{diag + 1}{2} \right\rfloor - l - 1 \tag{3.2}$$

- Algoritmus skončí, jestliže (i, j) není platná pozice v matici dynamického programování.

- Algoritmus porovná znaky x_i, x_j ve vstupním řetězci s a zvětší l o jedna a bude dál pokračovat prvním krokem, jestliže znaky x_i, x_j jsou stejné.

Z pohledu tabulky algoritmus potřebuje pouze znát hodnoty uložené v $(e, diag-1)$, $(e, diag)$ a $(e, diag+1)$, aby mohl vypočítat a uložit hodnotu do políčka $(e+1, diag)$. Na obrázku 3.9 je vyplněná tabulka podle vstupního řetězce mississippi. Časová složitost tohoto algoritmu patří do $\theta(n^2)$. To je ale nejhorší případ, který je způsobený porovnáváním znaků podél diagonály a inicializací. Inicializace bude trvat přibližně $\frac{n^2}{2}$ kroků, jestliže všechny znaky vstupního řetězce budou stejné. Paměťová složitost tohoto algoritmu patří do $\theta(kn)$.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
e0	0	0	0	0	0	2	0	0	3	0	0	2	0	0	0	0	0	2	0	0	0
e1	0	1	1	1	2	3	2	4	4	4	2	3	2	1	1	1	2	2	1	1	0
e2	0	1	1	2	2	3	3	4	4	5	4	5	3	3	3	3	2	2	1	1	0
e3	0	1	1	2	2	3	3	4	4	5	5	5	4	4	3	3	2	2	1	1	0

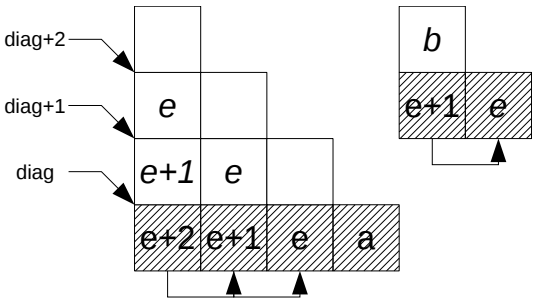
Obrázek 3.9: tabulka vyplněná podle vstupního řetězce mississippi

3.3.4 Zpětné dohledání

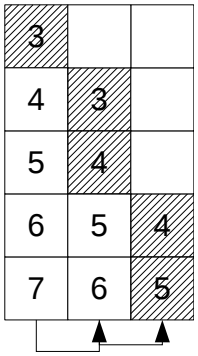
Algoritmus má na vstupu parametry počet chyb e a číslo diagonály $diag$, které určují pozici v tabulce. Algoritmus před vlastním vytvořením palindromu sestaví cestu v zásobníku $path$, která vede do políčka $(diag, e)$. Naplnění zásobníku $path$ je jednoduché, protože každé políčko má uloženou pomocnou hodnotu, která určuje políčko v předchozím řádku. Algoritmus si musí navíc zapamatovat, kolikrát použil $NEDel$, nebo $NWDel$. Nechť si tento počet pamatuje v proměnné del. Algoritmus může narazit pouze na jednu z těchto dvou značek, protože značku $NEDel$ mohl použít pouze při překročení spodního okraje matice DP a značku NWDel mohl použít pouze při překročení pravého okraje matice DP. Nechť $diag$ je diagonála, na které leží pravý dolní roh matice DP. Diagonála $diag$ nemůže obsahovat značku $NEDel$ ani $NWDel$, protože políčko v pravém spodním rohu v matici DP nemá pravého ani spodního souseda. Posuny mezi řádky pomocí $NEDel$ a $NWDel$ algoritmus do zásobníku $path$ neukládá.

Palindrom je lichý, jestliže na vrcholu zásobníku $path$ je uložená diagonála $diag$, která má sudé číslo. Palindrom je naopak sudý, jestliže na vrcholu zásobníku $path$ je uložená diagonála $diag$, která má liché číslo. Algoritmus vypíše do čela palindromu znak $x_{\lfloor \frac{diag}{2} \rfloor}$ ze vstupního řetězce, jestliže je palindrom lichý. Zásobník $path$ popisuje cestu tabulkou. Každý prvek r ze zásobníku $path$ tedy obsahuje číslo diagonály, vzdálenost a směr, ve kterém je další prvek v tabulce. Je to tedy struktura. Algoritmus si musí navíc pamatovat, jakou vzdálenost (pos) už urazil. Algoritmus pro každý prvek ze zásobníku $path$ provede následující operace:

- Posune se po diagonále matice DP ze vzdálenosti, jakou už urazil $pos+1$ až po vzdálenost, která je uložená v prvku r. Tento posun odpovídá vypsání dvojic $\binom{x_j}{x_i}$ do



Obrázek 3.10: překročení spodní hranice matice *DP*



Obrázek 3.11: komplikovaná cesta

palindromu, kde i, j je pozice v matici DP. Dvojice i, j je určená číslem diagonály a vzdáleností. Převodní vztahy jsou v rovnicích 3.1 a 3.2 Výpis probíhá popořadě. To znamená, že druhý prvek je vypsaný za prvním.

- Algoritmus si uloží do proměnné pos vzdálenost ze struktury r a proměnnou pos dále zvětší o jedničku, jestliže smět uložený ve struktuře r je $NORTH$. Do palindromu vypíše dvojici $\binom{x_j}{x_i}$, kde dvojice i, j je určená jako v prvním bodě. Číslo diagonály je uložené ve struktuře r a vzdálenost je nový obsah proměnné pos.

- Algoritmus si uloží do proměnné pos vzdálenost ze struktury r. Proměnnou pos dále zvětší o jedničku, jestliže směr uložený ve struktuře r je NW a číslo diagonály ve struktuře r je sudé. Do palindromu vypíše dvojici $\binom{-}{x_i}$.

- Algoritmus si uloží do proměnné pos vzdálenost ze struktury r. Proměnnou pos dále zvětší o jedničku, jestliže směr uložený ve struktuře r je NE a číslo diagonály ve struktuře r je sudé. Do palindromu vypíše dvojici $\binom{x_j}{-}$.

V předcházejícím výčtu jsou směry popsané ve směru od posledního řádku tabulky k jejímu nultému řádku, i když algoritmus postupuje opačným směrem. Tedy od nultého řádku dolů.

V této fázi algoritmus vygeneroval palindrom pro políčko, na které se dostal po posledním použití směru $NEDel$, nebo $NWDel$. Nyní musí tento palindrom upravit. V následujícím textu je popsaný postup pouze pro směr $NEDel$. Tedy pro překročení spodní hranice matice DP. Pro směr $NWDel$ algoritmus používá obdobný postup.

Na obrázku 3.10 zobrazená část matice, která způsobuje tento problém. Algoritmu stačí pro odvození ostatních potřebných hodnot znát hodnoty ve spodním řádku. Políčko v tabulce $(e + 2, diag)$ bude záviset na políčku $(e + 1, diag + 1)$ a směr bude bít nastavený *NEDel* a políčko v tabulce $(e + 1, diag + 1)$ bude záviset na políčku $(e, diag + 2)$ a směr bude mít také nastavený *NEDel*. Políčko a má v matici DP hodnotu větší rovnu e.

Z těchto hodnot plyne, že hodnota a je e, nebo $e + 1$, protože a může být maximálně $e + 1$. S daným spodním řádkem proměnná b nabývá hodnoty e. Důkaz:

- $b \leq e$ protože diagonála tvoří neklesající posloupnost.

- $b \geq e$ protože při menší hodnotě by spodní soused nemohl mít hodnotu $e + 1$.

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
h	e0	0	0	0	0	0	2	0	0	3	0	0	2	0	0	0	0	0	2	0	0	0
	e1	0	1	1	1	2	3	2	4	4	4	2	3	2	1	1	1	2	2	1	1	0
	e2	0	1	1	2	2	3	3	4	4	5	4	5	3	3	3	3	2	2	1	1	0
	e3	0	1	1	2	2	3	3	4	4	5	5	5	4	4	3	3	2	2	1	1	0

Obrázek 3.12: lichoběžníková část

Z průniku předchozích dvou nerovností plyne, že $b = e$. Toto pravidlo dokazuje, že trojúhelníková část na obrázku 3.10 je vyplněná správně. Rozdíl mezi vygenerovaným palindromem a upraveným palindromem je, že zdrojový postupuje po diagonále, ale upravený po sloupci. Algoritmus musí tedy nahradit prvních *del znaků - ne mezer* ve vygenerovaném palindromu mezerami. Na obrázku 3.11 je příklad složitější cesty. Hodnoty v políčkách na tomto obrázku jsou ve vztazích podle předchozího pravidla. Vygenerovaný palindrom postupuje po diagonále i po sloupci. Algoritmus by správně neupravil tento palindrom, kdyby nahradil prvních *del znaků+mezer* mezerami.

3.4 Implementace CUDA

Prvek v tabulce $(e, diag)$závisí pouze na vrchních sousedech $(e-1, diag-1), (e-1, diag), (e-1, diag+1)$. Každý blok vláken může počítat a ukládat do tabulky pouze lichoběžníkovou část. Vlákna uvnitř bloku spolupracují pomocí sdílené paměti, do které si ukládají právě zpracovávanou a předchozí část řádku, která je v lichoběžníku. Každý sloupec, který je v lichoběžníku, zpracovává jedno vlákno. Program ukládá vstupní řetězec do paměti pro textury, protože dotazy na hodnotu v tomto řetězci mají prostorovou lokalitu. Přístup do globální paměti může maximálně zabrat dvě operace, protože vrchní i spodní polovina warpu přistupují do celých úseků pole, které můžou být maximálně ve dvou segmentech. Kernel musí být spuštěný vícekrát, jestliže výška lichoběžníku h je menší než počet chyb $e + 1$, pro který program vytváří tabulku. Program s velkou hodnotou h je rychlejší, ale spotřebovává víc sdílené paměti, které je jen omezené množství. Program má také potenciál velmi dobře využít libovolný počet multiprocesorů, jestliže tento počet je dostatečně menší než dálka vstupního řetězce. Program nakonec přesune vytvořenou tabulku z globální paměti GPU do operační paměti a odtud sestavuje palindromy podle počtu chyb a pozice v řetězci *(diagonály)*. Na obrázku 3.12 je zobrazená oblast, kterou vyplňuje jeden blok vláken.

Kapitola 4

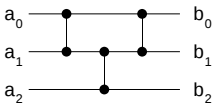
Batcherův mergesort lichá-sudá

Pří hledání palindromů je také důležité seřazení výsledků. Program může seřadit podle velikosti pomocí indexů libovolný řádek v tabulce o odpovídat, kde leží středy palindromů s největším poloměrem. Chyba je daná řazeným řádkem.

Seřazení posloupnosti je klasický problém, ve kterém má algoritmus přetransformovat vstupní posloupnost $A = \{a_0, a_1, \ldots, a_{n-1}\}$ na posloupnost $B = \{b_0, b_1, \ldots, b_{n-1}\}$, ve které pro všechna $i \in \{0, 1, \ldots, n-2\}$ platí $b_i \leq b_{i+1}$. Časová složitost optimálního sekvenčního algoritmu na seřazení posloupnosti patří do $\theta(n \log(n))$. Při použití libovolného počtu procesorů[1] lze sestrojit algoritmus s časovou složitostí, která patří do $\theta(\log(n))$. Tento algoritmu je *Enumeration sort*. Tento algoritmus potřebuje ale n^2 procesorů. Kompromis je Batcherův mergesort lichá–sudá, který potřebuje $n/2$ procesorů a jeho časová složitost patří do $\theta(\log^2(n))$. Při konstrukci řadící sítě by bylo potřeba přibližně $n.\log^2(n)$ komparátorů.

Definice: Neadaptivní řadící algoritmus je algoritmus, ve kterém provedené operace nezávisí na hodnotách řazeného pole.

Modelem pro neadaptivní řadící algoritmy je řadící síť. Příklad řadící sítě je na obrázku 4.1 Řadící síť obsahuje pouze komparátory se dvěma vstupy, které dokáží provádět operace porovnání a záměna dvou prvků. V řadících sítích lze snadno vidět, jestli jdou některé operace provádět paralelně. Algoritmus může paralelně provádět operace, které na sobě nezávisí. Batcherůr mergesort lichá–sudá byl vyvinutý panem Batcherem v roce 1968. Algoritmus



Obrázek 4.1: příklad sítě, která řadí pole o třech prvcích

patří do třídy neadaptivních řadících algoritmů. Základem algoritmu je část, která spojuje dvě seřazené posloupnosti do jedné. Algoritmus řadí jenom posloupnosti délky 2^n, kde n je délka posloupnosti. Algoritmus postupně spojuje prvky uspořádaných dvojic, uspořádané dvojice do čtveřic, uspořádané čtveřice do osmic, atd. Uspořádanost znamená, že prvky v n–ticích jsou seřazené. Důkazy, které jsou uvedené v této kapitole, jsou převzaté a místy upravené z [13] [14].

1. například v modelu PRAM

4.1 Část algoritmu merge

Následující algoritmus spojí dvě seřazené posloupnosti do jedné. První posloupnost je $\{a_0, a_1, \ldots, a_{\lfloor n/2 \rfloor - 1}\}$ a druhá posloupnost je $\{a_{\lfloor n/2 \rfloor}, a_{\lfloor n/2 \rfloor + 1}, \ldots, a_{n-1}\}$. Tyto posloupnosti jsou části posloupnosti A, kterou má algoritmus na vstupu. V následujícím algoritmu není popsané, že algoritmus si musí také pamatovat umístění každé hodnoty v první posloupnosti, aby zapisoval změny na správnou pozici.

- Jestliže $n > 2$ potom:
  - (I) Algoritmus spustí merge lichá–sudá na posloupnosti $A' = \{a_0, a_2, \ldots, a_{n/2-1}\}$ a na posloupnosti $A' = \{a_1, a_3, \ldots, a_{n-1}\}$ s $n = n/2$. Tyto dvě posloupnosti mají také seřazenou vrchní a spodní polovinu.
  - (II) Algoritmus porovná a případně zamění hodnoty v a_i a a_{i+1} pro $i \in \{1, 3, \ldots, n-3$.
- Jestliže $n \leq 2$ potom:
  - Algoritmus porovná a případně zamění hodnoty v a_0 a a_1.

Z tohoto algoritmu není zřejmé, jestli opravdu seřadí každou posloupnost. Proto následuje důkaz správnosti a důkaz jednoho pomocného tvrzení.

4.1.1 Správnost

V důkazu správnosti je použitý princip 0–1. Správnost dokážeme pomocí indukce vzhledem ke k, které je exponent v $n = 2^k$. Báze indukce $k = 1$: Posloupnost je seřazená komparátorem, jestliže $n = 2^1$. Předpokládáme, že algoritmus je správný pro všechna $n \in \{2^1, 2^2, \ldots, n^k\}$ a dokážeme, že algoritmus je správný i pro $n = 2^{k+1}$. Nechť $A = \{a_0, a_1, \ldots, a_n\}$ je vstupní posloupnost. Přepíšeme posloupnost do dvou sloupců. V levém budou a_i se sudým i a v pravém a_i s lichým i. Příklad je na obrázku 4.2a. Na obrázku 4.2b jsou vyplněné možné hodnoty, které a_i můžou nabývat. Oba sloupce budou mít seřazenou vrchní a spodní polovinu. Sloupce budou seřazené v kroku (I) algoritmu merge lichá–sudá díky indukčnímu předpokladu. Seřazené sloupce jsou na obrázku 4.2c. Pravý sloupce může ještě obsahovat maximálně o dvě jedničky víc než levý sloupec, protože přechod $0 \rightarrow 1$ může být ve vstupní posloupnosti maximálně dvakrát. Tento problém řeší krok (II) algoritmu merge lichá–sudá. Příklad kroku (II) algoritmu merge lichá–sudá je na obrázku 4.2d.

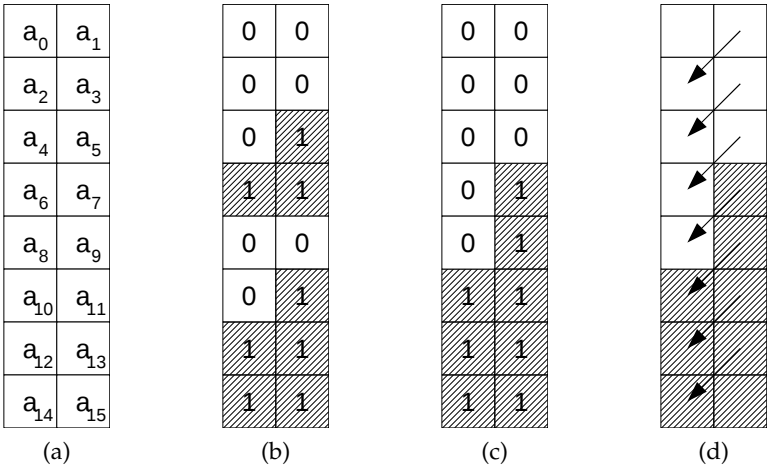
4.1.2 Princip 0–1

Tvrzení: Neadaptivní algoritmus správně řadí všechny posloupnosti libovolných hodnot právě tehdy, když neadaptivní algoritmus správně řadí všechny posloupnosti, ve kterých jsou hodnoty pouze 0 a 1.

K důkazu $\Rightarrow$ stačí nahradit libovolné hodnoty za hodnoty 0 a 1. K důkazu $\Leftarrow$ jsou navíc potřebné následující definice a tvrzení.

Definice: Nechť A, B jsou množiny. Zobrazení $f : A \rightarrow B$ je neklesají právě tehdy, když pro všechny $a_i, a_j \in A$ platí:

$$a_i \leq a_j \Rightarrow f(a_i) \leq f(a_j)$$



Obrázek 4.2: příklad postupu algoritmu merge lichá–sudá

Tvrzení: Nechť $f : A \to B$ je neklesající zobrazeni. Potom pro všechna $a_i, a_j \in A$ platí:

$$f(\min(a_i, a_j)) = \min(f(a_i), f(a_j))$$
$$f(\max(a_i, a_j)) = \max(f(a_i), f(a_j))$$

Důkaz:

- Nechť $a_i \leq a_j$. Potom $f(a_i) \leq f(a_2)$, protože f je neklesající zobrazení.

min			max		
a_i	$=$	$\min(a_i, a_j)$	a_j	$=$	$\max(a_i, a_j)$
$f(a_i)$	$=$	$\min(f(a_i), f(a_j))$	$f(a_j)$	$=$	$\max(f(a_i), f(a_j))$
$f(a_i)$	$=$	$f(\min(a_i, a_j))$	$f(a_j)$	$=$	$f(\max(a_i, a_j))$
$f(\min(a_i, a_j))$	$=$	$\min(f(a_i), f(a_j))$	$f(\max(a_i, a_j))$	$=$	$\max(f(a_i), f(a_j))$

- Nechť $a_i > a_j$. Potom $f(a_i) \geq f(a_j)$, protože f je neklesající zobrazení.

min			max		
a_j	$=$	$\min(a_i, a_j)$	a_i	$=$	$\max(a_i, a_j)$
$f(a_j)$	$=$	$\min(f(a_i), f(a_j))$	$f(a_i)$	$=$	$\max(f(a_i), f(a_j))$
$f(a_j)$	$=$	$f(\min(a_i, a_j))$	$f(a_i)$	$=$	$f(\max(a_i, a_j))$
$f(\min(a_i, a_j))$	$=$	$\min(f(a_i), f(a_j))$	$f(\max(a_i, a_j))$	$=$	$\max(f(a_i), f(a_j))$

Definice: Nechť $f : A \to B$ je zobrazení. Rozšíření zobrazení f na sekvenci $A_{ij} = a_i, a_{i+1}, \ldots, a_j$, kde $a_i, a_j \in A$ je definováno následovně:

$$f(a_i, a_{i+1}, \ldots, a_j) = f(a_i), f(a_{i+1}), \ldots, f(a_j)$$

Označení i-tého členu této posloupnosti je $f_i(a)$.
Definice komparátoru:

$$cmp^{ij}(A) = \begin{cases} b_i = \min(a_i, a_j) \\ b_j = \max(a_i, a_j) \\ b_k = a_k, k \neq i, j \end{cases}$$

Označení k-tého členu výstupní posloupnosti komparátoru je $cmp_k^{ij}(A)$.

Tvrzení: Nechť f je neklesající zobrazení a N je řadící síť. Potom N a f komutují. To znamená, že pro každou sekvenci $A_{ij} = a_i, a_{i+1}, \ldots, a_j$ platí:

$$N(f(A_{ij})) = f(N(A_{ij}))$$

Neklesající zobrazení f tedy může být použito na vstupní posloupnost řadící sítě, nebo na výstupní posloupnost řadící sítě a oba výsledky řadící sítě budou stejné.
Důkaz: Pro jeden komparátor platí následující rovnice:

$$\begin{aligned}
cmp_i^{ij}(f(A)) &= cmp_i^{ij}(f(a_0), f(a_1), \ldots, f(a_{n-1})) \\
&= \min(f(a_i), f(a_j)) \\
&= f(\min(a_i, a_j)) \\
&= f(cmp_i^{ij}(A)) \\
&= f_i(cmp^{ij}(A)) \\
cmp_j^{ij}(f(A)) &= cmp_j^{ij}(f(a_0), f(a_1), \ldots, f(a_{n-1})) \\
&= \max(f(a_i), f(a_j)) \\
&= f(\max(a_i, a_j)) \\
&= f(cmp_j^{ij}(A)) \\
&= f_j(cmp^{ij}(A)) \\
cmp_k^{ij}(f(A)) &= cmp_j^{ij}(f(a_0), f(a_1), \ldots, f(a_{n-1})) \\
&= f(a_k) \\
&= f(cmp_k^{ij}(A)) \\
&= f_k(cmp^{ij}(A))
\end{aligned}$$

Z předchozích rovnic plyne:

$$cmp^{ij} \circ f(A) = f \circ cmp^{ij}(A) \tag{4.1}$$

Tvrzení: Každá řadící síť je kompozice komparátorů a pro každou řadící síť N a neklesající zobrazení f platí:

$$N \circ f(A) = f \circ N(A) \tag{4.2}$$

Důkaz: Indukcí vzhledem k počtu komparátorů n v řadící síti. Báze indukce ($n = 1$) je dokázaná v rovnici 4.1. Předpokládejme, že rovnice platí pro počet komparátorů n a dokažme ji pro počet komparátorů $n + 1$. Levou stranu rovnice 4.2 upravíme tak, že přehodíme funkci f a nejvíce zanořený komparátor. Záměnu můžeme udělat díky rovnici 4.1. Nyní můžeme zanedbat nejvíce zanořené komparátory, protože provádějí stejnou transformaci vstupní posloupnosti A. Dostáváme tak rovnici s počtem komparátorů n, která platí díky indukčnímu předpokladu. Podle principu indukce tedy rovnice 4.2 platí. Příklad rovnice pro $n = 2$:

$$\begin{aligned}
f \circ cmp_1 \circ cmp_0 &= cmp_1 \circ cmp_0 \circ f \\
f \circ cmp_1 \circ cmp_0 &= cmp_1 \circ f \circ cmp_0
\end{aligned}$$

Tvrzení: (princip 0–1) Nechť N je řadící síť. Řadící síť N seřadí správně všechny posloupnosti libovolných hodnot, jestliže seřadí všechny posloupnosti, ve kterých jsou pouze hodnoty 0

a 1.
Důkaz: Nechť A je nějaká posloupnost, která není seřazení řadící sítí N. Ve výsledné posloupnosti jsou tedy nějaké hodnoty, že $b_k > b_{k+1}$. Nechť $f : A \to \{0, 1\}$ je zobrazení, které je neklesající a je definováno:

$$f(x) = \begin{cases} 0, & x < b_k \\ 1, & x \geq b_k \end{cases}$$

Posloupnost $B = N(f(A))$ není seřazená, protože $f(b_k) = 1$ a $f(b_{k+1}) = 0$ Tento postup ukázal, že existuje-li libovolná posloupnost A, která není seřazená sítí N (tvrzení α). Potom existuje posloupnost $f(A)$, která není seřazená sítí N (tvrzení $beta$). Pomocí logických odvozovacích pravidel vznikne tvrzení:

$$\begin{aligned} \alpha &\Rightarrow \beta \\ \neg\alpha &\vee \beta \\ \neg\alpha &\vee \neg\neg\beta \\ \neg\beta &\Rightarrow \neg\alpha \end{aligned}$$

Všechny posloupnosti A jsou seřazené sítí N, jestliže jsou všechny posloupnosti $f(A)$ seřazené sítí N.

4.2 Mergesort lichá–sudá

Mergesort lichá–sudá postupně spojuje seřazené posloupnosti do posloupností delších. Popis algoritmu je následující:

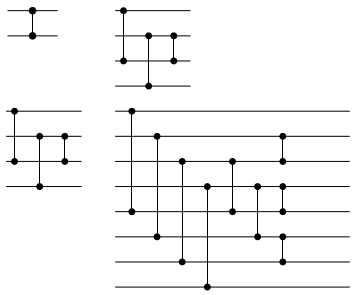
- Jestliže $n > 1$ potom:
  - (I) Algoritmus spustí mergesort lichá–sudá na první $(a_0, a_1, \ldots, a_{n/2-1})$ a druhou $(a_{n/2}, a_{n/2+1}, \ldots, a_{n-1})$ polovinu vstupní posloupnosti s $n = n/2$.
  - (II) Algoritmus spojí výstupní seřazené posloupnosti do jedné seřazené posloupnosti pomocí merge lichá–sudá.

4.3 Vytvoření sítě

Vytvoření řadící sítě je jednodušší odspodu nahoru. Algoritmus při vytvoření sítě velikosti n používá dvě stejné kopie sítě pro merge lichá–sudá, které mají velikost $n/2$. Jednu kopii použije pro sudé linky a druhou kopii pro liché linky. Linka označuje a_i. Výhoda linky oproti a_i je, že zobrazuje časovou posloupnost přístupů k a_i. Nakonec musí algoritmus přidat komparátory, které odpovídají kroku II v algoritmu merge lichá–sudá. Příklad konstrukce je na obrázku 4.3. Algoritmus při konstrukci sítě použije už vytvořenou síť. Sítě může proložit, protože první pracuje pouze s lichými a druhá pouze se sudými linkami.

4.4 Implementace, CUDA

Vstupní posloupnost je uložená v poli p a na vstupu je ještě pole indexů idx. Program přeskládá pole idx tak, aby pro všechna $i \in \{0, 1, \ldots, n-2\}$ platilo $p[idx[i]] \leq p[idx[i+1]]$. Algoritmus je tedy implementovaný jako nepřímá řadící metoda. To znamená, že nepřesouvá



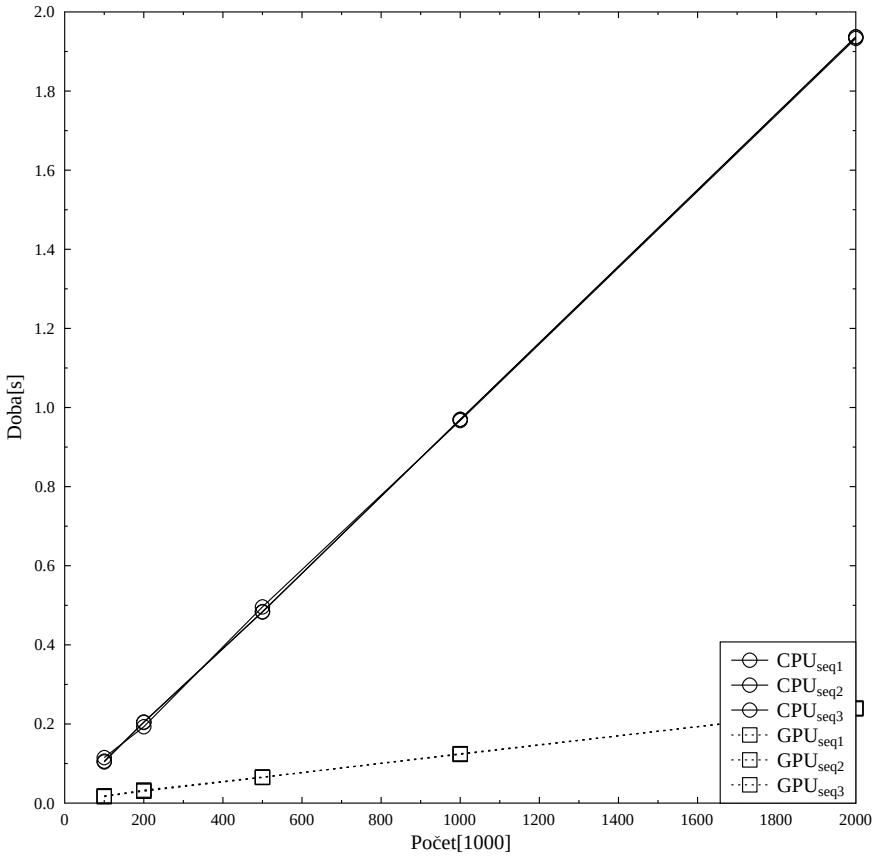
Obrázek 4.3: příklad vytvoření sítě merge lichá–sudá

data, ale indexy, které na data ukazují. Délka vstupní posloupnosti je n. Program rozdělí pole na dvě části. První část bude mít velikost $128 \cdot 2^k$, kde k je největší číslo, pro které platí, že $128 \cdot 2^k \leq n$. Druhá část bude mít tedy velikost $n - 128 \cdot 2^k$. Druhou část program seřadí pomocí obyčejného řazení slučováním (merge sort). Řazení slučováním má výhodu, že je to stabilní řadící algoritmus. Stabilní znamená, že délka výpočtu nezávisí na datech. Program seřadí první část pole pomocí upraveného řazení slučováním. Upravené řazení slučováním se liší od obyčejného v tom, že předpokládá seřazenost 128-tic. Díky zvolené délce první části upravené řazení slučováním bude vždy v nejhlubším zanoření slučovat 128-tice. Program pomocí GPU řadí pouze 128-tice.

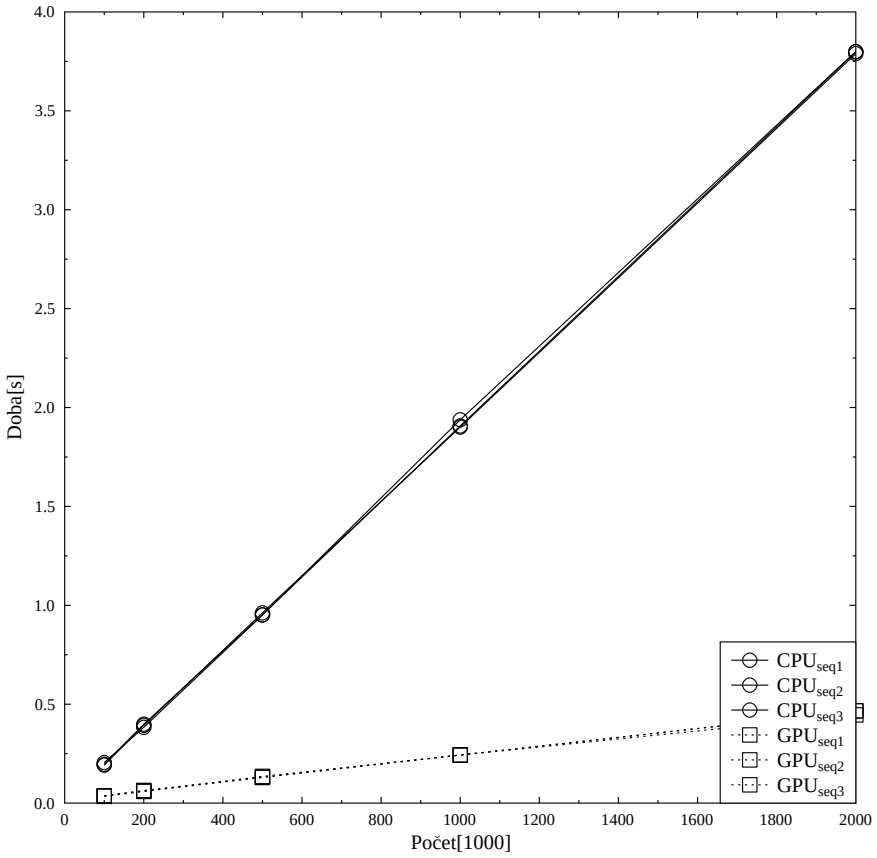
Kapitola 5

Závěrečné testy

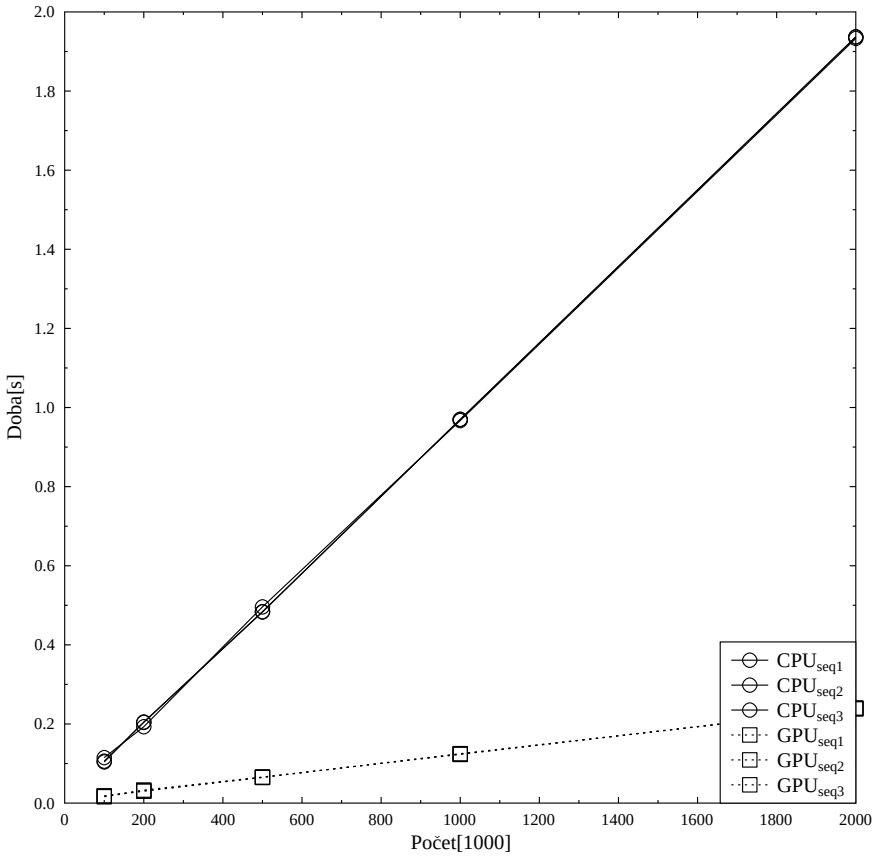
Všechny testy jsem prováděl na fakultním počítači *ops.fi.muni.cz*, který je osazený procesorem Intel Core(TM)2 6400 @ 2.13GHz, má 1GB operační paměti a obsahuje GPU Nvidia GeForce 9800 GX2. Tato GPU je složená ze dvou částí. Každá část obsahuje 16 multiprocesorů a přibližně 512MB globální paměti. Multiprocesory jsou taktované na frekvenci 1.5GHz. Moje implementace využívá pouze jednu část GPU. K testům na hledání palindromů jsem používal sekvence dostupné z *genome.ucsc.edu*. V testech je označená sekvence *chr6_apd_hap1* jako *seq1*, *chr6_cox_hap1* jako *seq2* a *chr6_cox_hap2* jako *seq3*. Při testech jsem používal vždy prvních n znaků posloupnosti bez případných znaků N, které reprezentují nečitelná místa. Při testování řadícího algoritmu jsem používal sekvenci, která byla vygenerovaná generátorem náhodných čísel. Jestliže není uvedeno jinak, tak je v dobách potřebných pro výpočet zahrnutý i čas, který je potřebný pro alokaci místa a přesunu dat na GPU a zpět. Na obrázcích 5.1 a 5.2 jsou zobrazené doby, které potřebují implementace na CPU a GPU vzhledem k délce vstupního řetězce, aby vyplnily tabulku poloměrů. Zde se hledají textové palindromy. Na obrázku 5.3 jsou naopak zobrazené doby při hledání komplementárních palindromů. Implementace algoritmu pro GPU je přibližně $7\times$ rychlejší, než je implementace pro CPU. Na obrázku 5.4 je zobrazený graf, který porovnává rychlost obyčejného mergesortu s mergesortem, který pracuje s posloupností, ve které jsou seřazené 128-tice. Program jsem testoval pouze na posloupnostech, které mají délku 128.2^x. Implementace s předřazováním je přibližně $1.5\times$ rychlejší, než obyčejný mergesort. Na obrázku 5.5 je graf, na kterém je zobrazená doba, po kterou počítá GPU při implementaci s předřazováním. Na tomto grafu není započítaná doba, která je potřebná pro přesuny dat na GPU a zpět. Implementace s předřazováním tráví pouze přibližně $1/100$ vlastním výpočtem na GPU. Zbytek času tráví na přesunech dat a výpočtem na CPU.



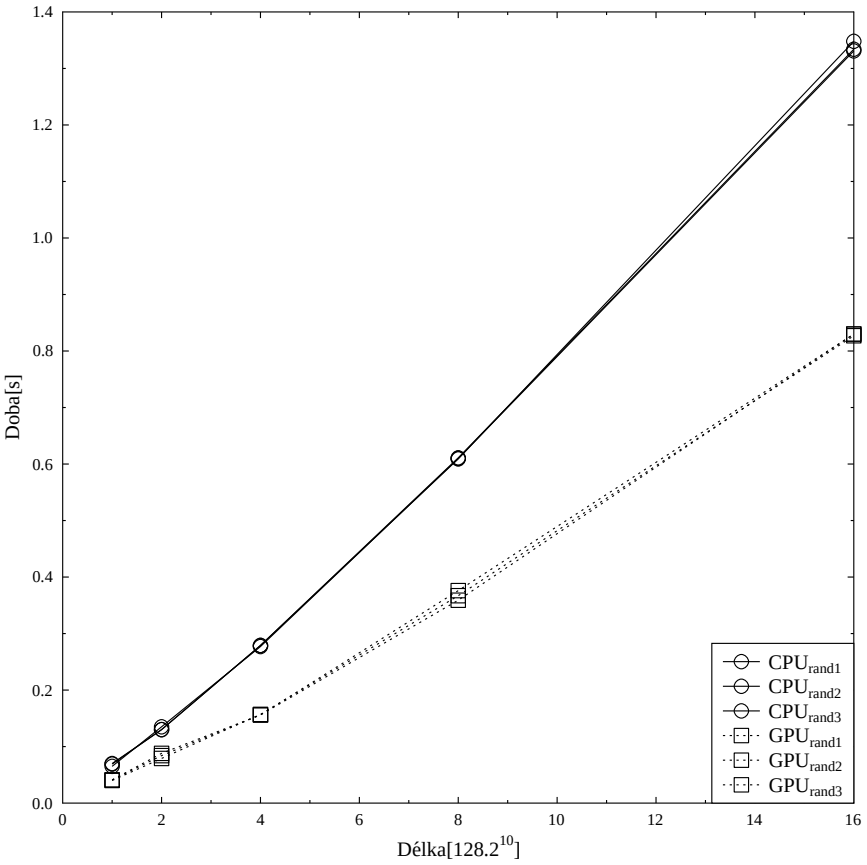
Obrázek 5.1: porovnání rychlostí při hledání textových palindromů s $error = 10$



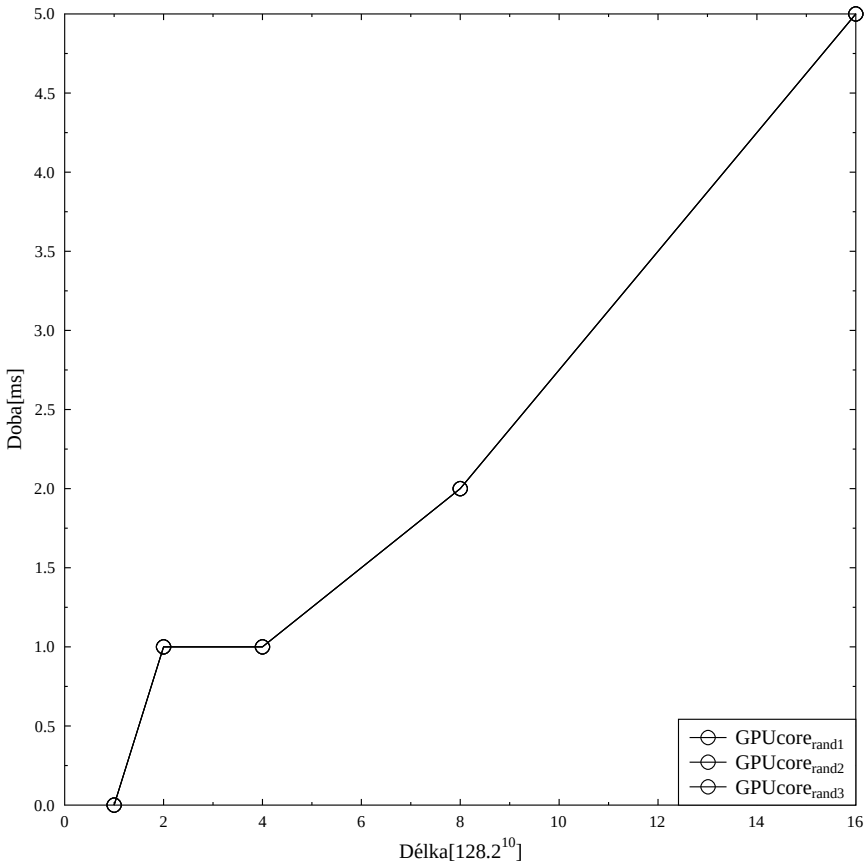
Obrázek 5.2: porovnání rychlostí při hledání textových palindromů s $error = 20$



Obrázek 5.3: porovnání rychlostí při hledání komplementárních palindromů s $error = 10$



Obrázek 5.4: porovnání rychlostí řadícího algoritmu na CPU a GPU



Obrázek 5.5: čas strávený výpočtem na GPU

Literatura

[1] DNA. *Wikipedie: otevřená encyklopedie* [online]. 5.5.2009. Dostupné z WWW: <http://cs.wikipedia.org/wiki/DNA>.

[2] BERRY, Drew. *WEHI-TV DNA Molecular Animation* [online]. 18.5.2009. Dostupné z WWW: <http:// www.wehi.edu.au/wehi-tv/dna/>.

[3] Amino acid. *Wikipedia: The Free Encyclopedia* [online]. 15.5.2009. Dostupné z WWW: <http:// en.wikipedia.org/wiki/Amino_acid>.

[4] tRNA. *Wikipedia: The Free Encyclopedia* [online]. 13.5.2009. Dostupné z WWW: <http:// en.wikipedia.org/wiki/Transfer_RNA>.

[5] SINDER, Richard R.. *DNA Structure and Function*. 1994. ISBN 978-0-12-645750-6.

[6] Restriction enzyme. *Wikipedia : The Free Encyclopedia* [online]. 7.5.2009. Dostupné z WWW: <http:// en.wikipedia.org/wiki/Restriction_enzyme>.

[7] SEDGEWICK, Robert. *Algoritmy v C*. 2003. ISBN 80-864997-56-9.

[8] MARTÍNEK, Tomáš; LEXA, Matej. *Hardware Acceleration of Approximate Palindrome Searching*,In: The International Conference on Field-Programmable Technology, Taipei, TW, IEEE CS. 2008. s. 65-72. ISBN 978-1-4244-2796-3.

[9] *NVIDIA CUDA Compute Unified Device Architecture Programming Guide* [online]. 8.12.2008. Dostupné z WWW: <http://www.nvidia.com>.

[10] EDDY, Sean R.. *What is dynamic programming?*. In: Nature Biotechnology., volume 22. s. 909–910.

[11] ALLISON, Lloyd. *Finding Approximate Palindromes in Strings Quickly and Simply* [online]. 23.11.2004. Dostupné z WWW: <http://www.csse.monash.edu.au/~lloyd/tildeProgLang/Java2/Palindromes/>.

[12] MANAVSKI, Svetlin A.; VALLE, Giorgio. *CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment* [online]. 28.3.2008. Dostupné z WWW: <http://www.biomedcentral.com/>.

[13] LANG, H. W.. *Odd-even mergesort* [online]. 26.01.2008. Dostupné z WWW: <http:// www.iti.fh-flensburg.de/lang/algorithmen/sortieren/networks/oemen.htm>.

[14] LANG, H. W.. *The 0-1-principle* [online]. 26.01.2008. Dostupné z WWW: <http:// www.inf.fh-flensburg.de/lang/algorithmen/sortieren/networks/nulleinsen.htm>.

Příloha A

Dodatky

Na přiloženém disku jsou všechny zdrojové kódy, které jsou zapotřebí k vytvoření programu, LaTeXsoubory, ze kterých byla vysázená tato bakalářská práce, všechny obrázky a manuál.
Struktura disku:

- /source - zdrojové kódy
- /manual - manuál
- /bachelor - zdrojové LaTeXsoubory a obrázky