1 Definice problému

Mějme čtvercovou matici $A \in \mathbb{R}^{n,n}$ a vektor pravých stran $b \in \mathbb{R}^n$. Cílem je najít vektor $x \in R^n$ takový, že $Ax = b$. Problém lze jednoduše vyřešit přímou metodou pomocí Gaussovy eliminace, nicméně kvůli omezené binární reprezentaci neceločíselných hodnot v paměti, nelze jednoduše algoritmitcky naimplementovat. Proto se používají numerické metody k řešení takových rovnic a jednou z známých metod na řešení lineárních rovnic je metoda konjungovaných gradientů.

Metoda předpokládá, že matice A je symetrická pozitivně definitní, jinak konvergence řešení není zaručené. Implementačně se jedná o velmi jednoduchou metodu, používá jen základní maticové a vektorové operace. Důkaz korektnosti, časové složitosti, ani odvození rovnic nebude součástí zprávy. Popis algoritmu a následně implementace je popsáno v pozdějších kapitolách.

O matici se navíc předpokládá, že je velmi řídká, proto je uložená v formátu vhodná pro takové situace.

1.1 Sekvenční algoritmus

Metoda konjungovaných gradientů je založena na iterativním přiblížení nějaké počátečního vektoru x_0 k optimálnímu výsledku $\overline{x}$. Na začátku se spočítá reziduum $r_0 = b - Ax_0$ a následně pomocí matice se opakovaně spočítá vektor směru s k nejlepšímu výsledku. Nová aproximace řešení se spočítá jako $x_{i+1} = x_i + \alpha_i s_i$ a α_i je koeficient posunutí (o kolik se ve směru s má posunout předchozí řešení).

Algoritmus vypadá následovně.

CG($A \in \mathbb{R}^{n,n}$, $b \in \mathbb{R}^n$)

(1) $x_0 = \{0\}^n$

(2) $r_0 = b - Ax_0$, $s_0 = r_0$

(3) iterace $k = 0, 1, \ldots$

(4) $$a_k = \frac{r_k^T r_k}{s_k^T A s_k}$$

(5) $$x_{k+1} = x_k + \alpha_k s_k$$

(6) $$r_{k+1} = r_k - \alpha_k A s_k$$

(7) pokud $||r_{k+1}|| < \varepsilon$, return x_{k+1}

(8) $$\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$$

(9) $$s_{k+1} = r_{k+1} + \beta_k s_k$$

Na řádku (1) a (6) se opakuje nejnáročnější operace, a to násobení matice s vektorem. Ale pokaždé se násobí stejná matice a stejný vektor s_k v jedné iteraci, proto lze výsledek As_k uložit a znovu využít. Další optimalizace spočívá v ukládání výsledku skalárního součinu

$r_k^T r_k$ který se vyskytuje na řádku (4), (8) a technicky taky (7) a (8) ještě jednou pokud si budeme ukládat výsledek pro 2 po sobě jdoucí iterace. V implementaci také není třeba si ukládat všechny prvky $x_0, \ldots, x_k$, ale vystačíme si s jedním vektorem x který budeme neustále přepisovat. To samé platí pro vektory r a s.

Zbytek operací má lineární složitost vzhledem k n. Sčítání vektorů probíhá po složkách, skalární součin je jen redukce po násobené po složkách. Násobení skalár vektorem je jen přepisování každého prvku jednou.

I když se technicky jedná o přímou metodu, jelikož z teoretického hlediska by algorimus měl dokonvergovat k exaktnímu řešení po nejvýše n krocích, tak implementačně se jedná o iterativní algoritmus. Nepřesnost nastává kvůli omezené reprezentaci datového typu `double`. K zastavení iterace je dán maximální počet iterací, který je nastaven na 2000. Na řádku (7) se testuje jestli je reziduum (chyba $b - Ax_k$) dostatčně malý. V implementaci se vyžaduje chyba menší než 10^{-9}.

1.2 Datové typy

V sekvenčním algoritmu se vše počítá pomocí typu `double`. Vektory x, r, s jsou uloženy jako `std::vector<double>`. Nejzajímavější je způsob ukládání matice A, která je řídká.

Matice A je uložená v formátu compressed sparse row (CSR). Matice se skládá celkem z 3 polí `std::vector<double> data`, `std::vector<int> columnPosition` a `rowPosition`.

Prvky matice jsou po řádcích zasebou uloženy v `data` a ukládají se jen nenulové prvky. K každému řádku je pak vyhrazena souvislá část `columnPosition`, kde na každém příslušném pozici je uloženo, v jakém sloupci se nachází prvek. Vektor `rowPosition` pak jen dodává, odkud se má začít číst v vektoru `columnPosition` a `data`. Pole `rowPosition` má fixní rozměr n a velikosti vektorů `data` a `columnPosition` závisí na počtu nenulových prvků.

Ze zadání se také předpokládá, že matice je symetrická, proto jsou v matici uloženy jen prvky nad diagonálou, tím si ušetříme dalších cca. 50% paměti.

1.3 Sekvenční maticové násobení

Díky formátu CSR je potřeba jen linárně projít vektor `data` k načtení prvků. Definice násobení matice a vektoru z prava pro výsledný prvek j je

$$(Ax)_j = \sum_{i=0}^{n-1} A_{j,i} x_i$$

Pokud $A_{j,i} = 0$, tak v součtu nepřispívá. V algoritmu se prvek nevyskytne a nepřičte se k výslednému $A_{j,i}x_i$. Protože je matice uložena symetricky, tak je potřeba k prveku $(Ax)_i$ ještě přičíst $A_{i,j}x_j$. K vyřešení dvojího počátání na pozicích $A_{j,j}$, tak lze využít trik a ukádat si $A'_{j,j} = \frac{1}{2} A_{j,j}$.

2 CUDA implementace

Algoritmus je potřeba nyní poupravit k přípravě na implementaci na grafkou kartu. Celkové řízení programu probíhá na CPU a na GPU se provádějí jen náročné operace jako jsou násobení matice vektorem, sčítání 2 vektorů nebo vektorové násobení dvou vektorů. Kvůli efektivitě jsou některé kroky původního algoritmu konjungovaných gradientů sloučeny do 1 cuda kernelu, ale celkový postup se nijak nemění.

2.1 Práce s pamětí a využití threadů

První úprava je v změně kontejneru kvůli kompatibilě ukazatelů. Všude kde se využil `std::vector`, tak je třeba změnit na pamět v globální paměti a alokovat pomocí `cudaMalloc`. Jelikož si všude vystačíme jen s jednoduchými typy a vektorem, tak byl využit `thrust::device_vector` pro správu paměti a v algoritmech se už jen použil ukazatel k uložené paměti.

Aby se minimalizovalo počet kopírování mezi hostem a device, tak jsou všechny proměnné a pole uloženy na grafické kartě a kopíruje se jen norma vektoru $||r_{k+1}||$ k zjištení konce iterace. Zbytek proměnných není třeba kopírovat k CPU a host jen řídí spuštění operací v každé operaci.

Z hlediska počtu threadů, jelikož se očekává že matice je velmi řídká, tak každému cudaBlocku je přiřazeno 1 řádek matice. Každý cudaBlock obsahuje 256 vláken.

V rámci kernelů se také používá sdílená paměť pro efektivní využití paralelní redukce při násobení matice vektorem, nebo násobení vektor vektorem.

2.2 Paralelní vektorové sčítání a násobení skalárem

Jedná se o nejjednodušší operace. Každému vláknu je přiřazen 1 index. Pomocí globálního indexu (`blockIdx.x * blockDim.x + threadIdx.x`) se načte 1 prvek z vektoru `x` a 1 prvek z vektoru `y`, sečtou se a výsledek se uloží do výstupního pole na stejném indexu. Při násobení skalárem mají všechny vlákna navíc přistup k koeficientu α, který je dostupný v globální paměti.

2.3 Paralelní vektorové násobení

Vektorové násobení není nic jiného, než sčítací redukce nad součinem. Každé vlákno si načte své dva prvky pomocí globálního indexu a lokálně pronásobí hodnoty. Poté je potřeba provést paralelní redukci. Redukce je nejdříve provedena v rámci cuda bloku pomocí warpových operací a sdílené paměti. Následně nulté vlákno v rámci bloku provede atomické přičtení k výsledku jelikož kvůli efektivitě je pole rozdělené mezi více cudaBlocků.

2.4 Paralelní násobení matice vektor

Předpokládáme, že matice je velmi řídká a lze namapovat každý řádek k jednomu cuda bloku. Každý cuda blok si pomocí `blockIdx.x` proměnné zjistí v `rowPosition` poli kde začínají data pro daný řádek v `data`. Následně pomocí indexování v slouci `columnPosition` spočítá obě souřadnice a provede násobení s vektorem x. V rámci bloku lze operaci nahlížet jako násobení vektor vektorem, proto si stačí aby každé vlákno drželo dočasný součet násobení. Na konci se pak v rámci cuda bloku provede ještě paralelní redukce a následně vlákno 0 provede atomické přičtení do výsledku. Navíc, matice je uložena symetricky, proto když si vlákno načte prvek matice $A_{i,j}$, tak musí ještě spočítat výsledek pro $A_{j,i}$.

3 Měření

	Testovací prostředí
CPU	AMD Ryzen 5 4600H @ 3.00GHz
RAM	24 GB
GPU	NVIDIA GTX 1650
OS	Windows 10 s WSL2 (Ubuntu-20.04)
g++	(GCC)9.3.0
nvcc	V10.1.243
flags	-O2

3.1 Popis instancí

Celkem bylo použito 12 instancí staženo z https://sparse.tamu.edu/. Jednají se převážně o instance popisující problém z reálného života, proto mohou obsahovat nepřesnosti z měření. Matice tedy nemusí být pozitivně definitní a může dojít k výchylkám.

3.2 Způsob měření

U sekvenční implementace se měří jen čas výpočtu samotné metody konjungovaných gradientů, bez načítání vstupu a kontrola výstupu. K měření času byla využita knihovna `std::chrono` od GNU. CUDA implementace byla také naměřena bez načítání vstupu a bez převodu dat z CPU na GPU, předpokládá se implicitně, že data už od začátku jsou uložené v globální paměti GPU. Narozdíl od CPU ale se zde navíc měří i s časem alokace všech potřebných proměnných.

instance	n	počet nenul	CPU [ms]	CUDA [ms]	zrychlení
Trefethen_20b	19	147	0.554	3.706	0.15
662_bus	662	2447	3.499	47.669	0.07
1138_bus	1138	4054	19.303	219.442	0.07
Trefethen_500	500	8478	3.915	39.311	0.1
msc00726	726	34518	24.223	72.6	0.33
thermomech_TK	102158	711558	101.706	94.192	1.07
apache2	715176	4817870	2383.66	1644.45	1.449
pdb1HYS	36417	4344765	667.927	144.908	4.609
x104	108384	8713602	30287.4	5861.84	5.166

Tabulka 1: Přehled naměřených instancí.

3.3 Výsledky

Jako základ byla použita CUDA implementace s 256 vlákny na 1 cuda bloku. Dle naměřených výsledků je zřejmé, že paralelní implementace se naprosto nehodí pro řešení malých instancí. Zrychlení ale dosáhneme u velmi velkých matic, až řádově $7 \cdot 10^6$ nenulových prvků. Čím je větší uloha, tím je zrychlení výraznější. U poslední matice x104 dosahuje CUDA implementace až 5-ti násobné zrychlení oproti sekvenčnímu řešení, i přes cenu velké režie za alokaci a dealokaci pomocné paměti.

V další fázi jsem pak zjišťoval, která konfigurace spuštění kernelu přináší nelepší výsledky. Byly použity stejné instance, jen se pokaždé naměřil celkový čas s konfigurací 128, 256 nebo 512 vláken na 1 cuda blok. Dle výsledků je opět zřejmé, že s 128 vlákny je implementace nejrychlejší. Bylo by zajímavé tento poznatek pořádněji změřit pomocí profileru.

instance	128	256	512
Trefethen_20b	3.497	3.706	3.514
662_bus	38.239	47.669	49.867
1138_bus	203.212	219.442	374.709
Trefethen_500	34.522	39.311	32.793
msc00726	74.381	72.6	82.291
thermomech_TK	67.373	94.192	174.723
apache2	924.34	1644.45	3098.53
pdb1HYS	82.346	144.908	244.087
x104	3288.59	5861.84	10870.2

Tabulka 2: Časy CUDA implementace s různými počty vláken per CUDA blok.

Poslední měření bylo pak provedeno na základě otázky cvičícího. Konkrétně jde o efekt změny použití datového typu `float` za `double` v programu. Jelikož tento test se provedl předtím, než se zjistilo, že konfigurace s 128 vlákny je nejlepší, tak všechny následující hodnoty jsou

instance	čas	čas	iterace	iterace	chyba	chyba
Trefethen_20b	3.706	3.598	20	18	6.1568e-08	2.39649e-12
662_bus	47.669	38.299	291	239	8.9234e-07	9.14534e-07
1138_bus	219.442	239.177	1455	1217	8.07739e-07	9.31279e-07
Trefethen_500	39.311	37.865	237	236	9.25301e-07	9.32095e-07
msc00726	72.6	90.881	563	485	9.43577e-07	9.3856e-07
thermomech_TK	94.192	227.824	35	35	8.01171e-07	8.01172e-07
apache2	1644.45	3157.5	91	90	8.43131e-07	7.26973e-07
pdb1HYS	144.908	249.327	103	103	9.81847e-07	9.51486e-07
x104	5861.84	11396.4	2000	2000	6.83446	1.27161

Tabulka 3: Porovnání float vs double na CUDA implementaci, levý sloupec je float, pravý je double.

změřeny s konfigurací 256 vláken per CUDA block.

V tabulce lze nejen nahlédnou na celkový čas, ale také počet iterací a konečná velikost rezidua. Z časového hlediska je hned vidět, že implementace se typem `float` je až 2-krát rychlejší. Je to dáno velikostí daných typů. Datový typ `double` je 2-krát větší než `float` a musí se pokaždé provést 2-krát větší transakce přes globální pamět.

Naopak počet iterací se příliš nemění a relativní chyba se také výrazně nezlepší. Z naměřených dat lze vyvodit, že je optimální počítat trochu nepřesněji pomocí `float` ale za to získat lepší časy.

Na závěr byl ještě program spuštěn na serveru STAR FIT ČVUT s grafickou kartou 2080Ti.

instance	1650	2080Ti
Trefethen_20b	3.497	3.994
662_bus	38.239	21.257
1138_bus	203.212	94.006
Trefethen_500	34.522	18.006
msc00726	74.381	38.782
thermomech_TK	67.373	28.975
apache2	924.34	415.189
pdb1HYS	82.346	37.265
x104	3288.59	1314.81

Tabulka 4: Porovnání výkonu 1650 vs 2080Ti.

4 Závěr

Dle zadání byla naimplementovaná metoda konjungovaných gradientů pro GPU. Implementace podporuje formát řídkych matic a ukládá je symetricky. Dle naměřených hodnot je paralelní verze na větších instancích až 5-krát rychlejší oproti sekvenčnímu řešení a lze rozumně využít jako součást většího řešiče, kde se již očekává, že data jsou již v globální paměti GPU.