VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

MODERNÍ KNIHOVNY PRO PROGRAMOVÁNÍ GRAFICKÝCH KARET

MODERN LIBRARIES FOR GPGPU PROGRAMMING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **PATRIK ŠUBA**
AUTHOR

VEDOUCÍ PRÁCE **Doc. Ing. JIŘÍ JAROŠ, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Šuba Patrik**
Program: Informační technologie
Název: **Moderní knihovny pro programování grafických karet**
Modern Libraries for GPGPU Programming
Kategorie: Paralelní a distribuované výpočty

Zadání:

1. Osvojte si základy práce s grafickými kartami.
2. Proveďte rešerši v oblasti knihoven pro programování grafických karet.
3. Vytvořte sadu testovacích problémů, na které demonstrujete možnosti jednotlivých knihoven. Uvažujte jednoduché testy i složitější výpočetní problémy.
4. S využitím vybraných knihoven implementujte a optimalizujte algoritmy řešící dané problémy na zvolené grafické kartě.
5. Porovnejte výhody a nevýhody jednotlivých knihoven. Zaměřte se především na výkonnost a náročnost jejich použití.
6. Zhodnoťte dosažené výsledky a diskutujte možnosti pokračování projektu.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Jaroš Jiří, doc. Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 29. října 2021

Abstrakt

Cieľom tejto bakalárskej práce je uskutočniť výskum v oblasti knižníc pre programovanie grafických kariet a vytvoriť sadu testovacích problémov s využitím týchto knižníc. Testovacie problémy pozostávajú z matematických operácií s maticami a vektormi. Pre testovacie problémy boli vytvorené dve aplikácie. Prvá aplikácia bola implementované v jazyku C++ s použitím knižnice OpenMP. Druhá aplikácia bola implementovaná v jazyku C++ s použitím knižnice cuBLAS a CUDA. Implementácia tejto práce umožňuje nahliadnuť do problematiky programovania grafických kariet a ukázať ich praktické využitie. Výsledky práce overujú výkon a priepustnosť poskytnutých grafických kariet od skupiny IT4Innovations. Výsledky aplikácií sú porovnané s referenčnými hodnotami od výrobcu grafických kariet a medzi použitými knižnicami.

Abstract

The main goal of this thesis is to conduct research in the field of graphics card libraries and to use this libraries to create a set of test cases. Test cases consist of mathematical operations with matrices and vectors. Two applications have been created for test cases. The first application was implemented in C ++ using the OpenMP library. The second application was implemented in C ++ using the cuBLAS and CUDA libraries. The implementation part of this work allows reader to look into the problematics of GPGPU programming and shows its practical use. The results of this work is to verify the performance and throughput of the graphics cards provided by the IT4Innovations group. The results of the applications are then compared with the referential values from the graphics card manufacturer and also among the used libraries.

Kľúčové slová

OpenMP, C++, cuBLAS, NVCC, NVC++, GPGPU, CUDA, Thrust

Keywords

OpenMP, C++, cuBLAS, NVCC, NVC++, GPGPU, CUDA, Thrust

Citácia

ŠUBA, Patrik. *Moderní knihovny pro programování grafických karet.* Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Doc. Ing. Jiří Jaroš, Ph.D.

Moderní knihovny pro programování grafických karet

Prehlásenie

Vyhlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Doc. Ing. Jiří Jaroša Ph.D. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Patrik Šuba
10. mája 2022

Poďakovanie

Rád by som venoval poďakovanie môjmu vedúcemu práce Doc. Ing. Jiřímu Jarošovi, Ph.D, za odborné vedenie, konzultácie, trpezlivosť a podnetné návrhy k práci. Táto práca bola podporená Ministerstvom školstva, mládeže a telovýchovy Českej republiky prostredníctvom e-INFRA CZ (ID:90140).

Obsah

Kapitola 1

Úvod

Popularita využitia grafickej karty pre výpočet zložitých výpočtov v posledných rokoch stúpla. Už v tejto dobe pomáha grafická karta v medicíne, astronómií, vytváraní videohier, pri strojovom učení a mnoho ďalších odvetviach. V minulosti slúžila len ako výpočtová jednotka pre zobrazovacie zariadenia. V súčastnosti sa neprestala používať ako zobrazovacie zariadenie, je to práve naopak, stále sa pracuje na kvalitnejšom a rýchlejšom zobrazení [3].

Cieľom tejto práce je vytvoriť sadu testovacích problémov, ktoré budú demonštrovať možnosti jednotlivých knižníc pre prácu s grafickými kartami. Vytvorenie testov zahrňuje preštudovanie aktuálne používaných knižníc a pochopenie použitia jednotlivých knižníc. Testy pozostávajú z operácií nad maticami a vektormi, ktoré majú využitie v porovnávaní dosiahnutých rýchlostí rôznymi grafickými kartami, inými slovami benchmark grafických kariet. Tieto testy majú za úlohu porovnať a poukázať na výhody a efektívnosť jednotlivých použitých knižníc.

Kapitola 2 sa venuje grafickej karte vo všeobecnosti, využitiu a konštrukcii. Následne je opísaná architektúra Volta a Ampere od spoločnosti Nvidia, z ktorých pozostávajú grafické karty, ktoré boli použité pre spúšťanie sady testov. Výsledky spúšťania boli využité ako výsledky tejto práce. Architektúry Volta a Ampere určujú zloženie grafických kariet, ktoré boli vytvorené na základe týchto architektúr.

Kapitola 3 je zameraná na knižnice pre prácu s grafickou kartou. Nachádzajú sa tu moderné knižnice a to konkrétne, OpenMP, CUDA, cuBLAS a Thrust. Pre knižnice cuBLAS a Thrust vysvetľuje aké funkcie sa v nich nachádzajú a k čomu je ich možné využiť. Čo sa týka knižníc OpenMP a CUDA, sú tu vysvetlené jednotlivé konštrukcie knižníc, ktoré sú vytvorené pre paralelné programovanie na grafickej karte. Za pomoci knižnice OpenMP je možné vytvárať paralelný kód aj pre procesor.

Znalosti získané z kapitoly 3 sú využité pri implementácii aplikácií. Implementácia je podrobne rozobratá v kapitole 4. Implementácia pozostáva z dvoch aplikácií. Prvá aplikácia bola vytvorená za použitia knižnice OpenMP. Druhá aplikácia bola vytvorená za použitia knižníc CUDA a cuBLAS. V rámci implementácie sa nachádzajú aj kompilátory, ktoré boli potrebné pre kompilovanie a následné spustenie aplikácií.

Posledná kapitola tejto práce 5 sa venuje výsledkom získaných pomocou aplikácii z kapitoly 4 a ich testovaniu. Tieto výsledky sú vynesené do logaritmických grafov, ktoré zobrazujú dosiahnuté rýchlosti pre celú sadu testov. Testovanie aplikácií pozostáva z overenia vypočítaných výsledkov a sledovania celého procesu počítania. Vytvorené kernely pre CUDA aplikáciu boli profilované pre väčšiu optimalizáciu.

Kapitola 2

Grafické karty a ich architektúra

Grafická karta je druhá najťažšie pracujúca súčasť počítača hneď po procesore. Oproti procesoru obsahuje mnohonásobne viac jadier, ktoré pracujú na nižšej frekvencií. Prvotné využitie bolo iba pre vykreslenie obrazu na zobrazovacom zariadení pomocou aplikačných programovacích rozhraní OpenGL a DirectX. Uplatnenie týchto aplikačných programovacích rozhraní sa nachádza hlavne vo vývoji počítačových hier. Vývojári dokážu vytvárať stále zložitejšiu a zložitejšiu grafiku počítačových hier, ktorá potrebuje vysoko výkonné grafické karty aby dokázali včas vykresliť obraz. Na začiatku 21. storočia začal vývoj aplikačných programovacích rozhraní pre využitie grafickej karty ako akcelerátoru pri počítaní zložitých výpočtov. Uplatnenie môžme nájsť napríklad pri vytváraní umelej inteligencie, ťažení kryptomien, astronomických a molekulárnych výpočtoch. Existuje mnoho aplikačných programovacích rozhraní pre rôzne programovacie jazyky. Kapitola 3 je zameraná na vybrané rozhrania s podrobnejším popisom použitia [1].

2.1 Architektúry jadra grafických kariet spoločnosti Nvidia

Výpočtová schopnosť grafického procesoru od spoločnosti NVIDIA je reprezentovaná číslom verzie, tiež nazývaná SM verzia. Číslo verzie udáva podporované funkcie pre daný hardvér grafickej karty a je použité pri spúšťaní aplikácie aby sa rozhodlo, ktoré hardvérové funkcie a/alebo inštrukcie sú dostupné na prítomnej grafickej karte. Číslo výpočtovej schopnosti sa skladá z hlavného čísla revízie X a vedľajšieho čísla revízie Y, dokopy tvoria číslo X.Y. Hlavné číslo revízie určuje architektúru jadra. Zariadenia s NVIDIA Ampere GPU architektúrou majú hlavné číslo revízie 8, zariadenia s Volta architektúrou majú číslo 7. Tieto dve architektúry budú hlavným predmetom tejto práce, pretože grafické karty, ktoré boli použité pre spúšťanie sady testovacích problémov z kapitoly 4 sú reprezentantmi spomenutých architektúr [10].

2.2 Volta

V roku 2017 bola predstavená grafická karta NVIDIA Tesla V100 s architektúrou VOLTA, ktorá bola prelomová vo vysoko výkonnom počítaní, ktoré je základným pilierom pre modernú vedu. Akcelerátor nachádzajúci sa v NVIDIA Tesla V100 bol svetovo najvýkonnejší paralelný procesor navrhnutý pre vysokovýkonné počítanie, umelú inteligenciu a grafickú zátaž. Grafický procesor karty je vyrobený pomocou TSMC 12 nm FFN vysokovýkonného výrobného procesu prispôsobený pre NVIDIA a obsahuje 21,1 biliónov tranzistorov.

Nová kľúčová vymoženosť architektúry pozostáva z nových streaming multiprocesorov (SM) optimalizovaných pre hlboké strojové učenie. Volta SM sú o 50% efektívnejšie, pri výpočte hodnôt s dátovým typom float 32 a 64, ako SM pri predchádzajúcej architektúre Pascal. SM taktiež podporuje novú kombináciu L1 cache pamäti a zdielanej pamäti podsystému, ktorá značne zvyšuje výpočtový výkon. Ďalšia kľúčová vymoženosť je druhá generácia NVLink-u, ktorá poskytuje väčšiu rýchlosť priepustnosti dát a viacej prepojení pre výpočet vykonávaný na viacerých grafických kartách zároveň alebo vykonávaný na viacerých grafických kartách zároveň za použitia procesoru. Ďalšia kľúčová vymoženosť, ktorá stojí za zmienku je typ pamäti grafickej karty HBM2, ktorá je o veľkosti 16GB a podporuje šírku prenosového pásma až 900 GB/s. Architektúra obsahuje mnoho ďalších vylepšení oproti staršej architektúre Pascal. Na obrázku 2.1 je možné vidieť zloženie jedného Volta streamovacieho multiprocesoru. SM sa skladá zo štyroch procesných blokov, L1 inštrukčnej pamäti, 128 KB L1 cache pamäti a zo štyroch textúrových jednotiek. Jeden procesný blok pozostáva zo 16 FP32 jadier, 16 INT32 jadier, 8 FP64 jadier, 2 nové Tenzorové jadrá, novú L0 inštrukčnú pamäť, warp plánovač, expedičnú jednotku a 64KB registrovú jednotku. Toto zloženie SM zvyšuje využitie a celkový výkon oproti staršej architektúre Pascal [21].

Obr. 2.1: Volta streaming multiprocesor. Prevzaté z: [21].

2.3 Ampere

Ampere architektúra je postavená nad Volta architektúrou. Nová architektúra poskytuje nové funkcie, značne väčší výkon pre vysoko výkonné počítanie, umelú inteligenciu a analýzu

dát počas záťaže. Architektúra poskytuje novú technológiu tenzorového jadra. Podporuje tenzorové operácie nového matematického modelu TensorFloat32 (TF32), ktorý poskytuje zrýchlenie vstupno/výstupných dát dátového typu float32, ktoré sú použité pre hĺbkové učenie a vysokovýkonné počítanie. Je to už tretia generácia tenzorových jadier, ktorá zlepšuje zdieľanie operandov, zvyšuje efektívnosť a pridáva nové silné dátové typy. Ďalej podporuje FP64 a BF16 tenzorové jadro pre prácu s dátovým typom float64 a float16. Grafický procesor založený na tejto architektúre je vyrobený pomocou TSMC 7nm N7 výrobného procesu a obsahuje 54,2 biliónov tranzistorov. Obsahuje nové streaming multiprocesory, ktoré výrazne zvyšujú rýchlosť počítania a pridávajú mnoho nových schopností. Grafická karta NVIDIA A100 je vyrobená na základe tejto architektúry. Grafická karta podporuje zbernicu PCI Express Gen 4, ktorá obsahuje šírku prenosového pásma 64 GB/sec čo je zdvojnásobenie šírky pásma oproti PCIe 3.0/3.1. Ako je možné vidieť z obrázku 2.2 celková stavba SM ostala rovnaká ako pri architektúre Volta. Rovnaká stavba však neznamená, že neboli pridané žiadne vylepšenia. L1 cache pamäť je zväčšená na 192KB a SM podporuje novú rýchlejšiu inštrukčnú sadu. Zloženie jedného procesného bloku sa zmenilo. Jeden procesný blok obsahuje 16 INT32 jadier, 16 FP32 jadier, 8 FP64 jadier a jedno tenzorové jadro. Nové tenzorové jadro podporuje akceleráciu všetkých dátových typov, ktoré zahrňujú FP16, BF16, TF32, FP64, INT8, INT4 a binárne. Taktiež umožňuje dosiahnuť mnohonásobne vyššie rýchlosti s počítaným týchto dátových typov [9].

Obr. 2.2: Ampere streaming multiprocesor. Prevzaté z: [9].

Kapitola 3

Knižnice pre prácu s grafickými kartami

Moderné knižnice pre prácu s grafickými kartami umožňujú paralelné počítanie, ktoré poskytuje značne väčšiu priepustnosť ako tradičné systémy. Masívny paralelizmus je budúcnosťou vysokovýkonného počítania. Bez dobrých nástrojov je vývoj vysokovýkonných aplikácií pomalý a značne komplikovaný. Každý problém pri vývoji aplikácie je špecifický a programátori si "zašpinia ruky"riešením jedinečných problémov napísaním kódu, ktorý bude vykonaný na cieľovom zariadení paralelne. V tejto kapitole sa venujem knižniciam a aplikačným programovacím rozhraniam, ktoré som v rámci rešeršu pre túto prácu spracoval. Vývoj mojej sady testov pre vysokovýkonné paralelné počítanie na grafickej karte využíva tieto knižnice [4].

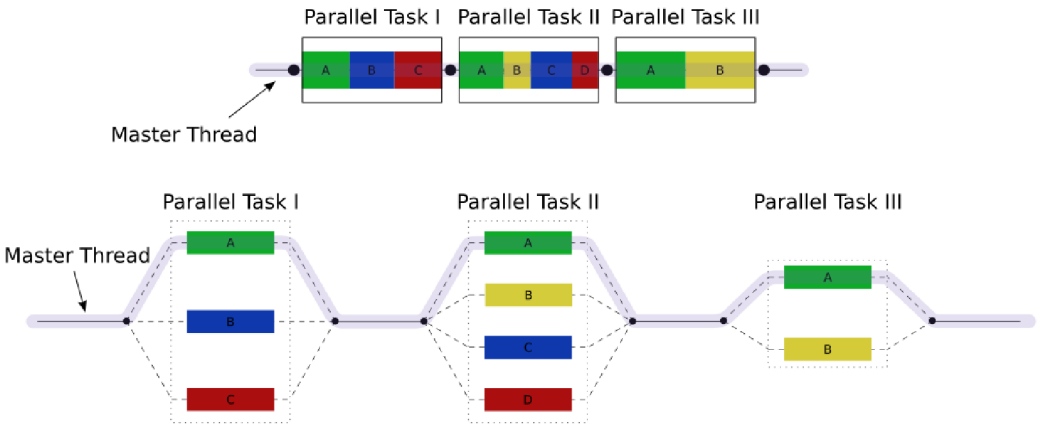
3.1 OpenMP

OpenMP je aplikačné programové rozhranie pre paralelné výpočty so zdieľanou pamäťou. OpenMP je možné používať s programovacími jazykmi C, C++ a Fortran. Konštrukcie z knižnice OpenMP pracujú s vláknami na pozadí a užívateľ teda nemusí priamo vytvárať a usmerňovať vlákna. Na druhú stranu, musí vedieť ako použiť dané konštrukcie knižnice aby bol výpočet dostatočne rýchly a nenastala žiadna chyba pri behu programu. Medzi najčastejšie chyby patrí napr. segmentačná chyba alebo nesprávne vypočítaný výsledok operácie. Podporuje všetky operačné systémy v hlavných verziách založené na Unix, Linux a taktiež podporuje operačný systém Windows. Jeho využitie je teda nezávislé na operačnom systéme a hardvéry, čo je veľká výhoda pre univerzálne použitie programu a jeho jednoduchú distribúciu. Aktuálna oficiálna verzia je 5.0, ktorá bola vydaná v roku 2018. Vo verzií 4.0 začala podpora programovania akcelerátorov, ktoré pozostávajú aj z grafických kariet. Vo fáze príprav sa nachádzajú nové aktualizácie 5.1 a 5.2, ktoré ešte nie sú oficiálne vydané a preto sa v tejto práci venujem hlavne verzii 5.0.

OpenMP pozostáva z troch častí[8]:

- direktívy a konštrukcie pre C, C++,
- premenné prostredia,
- knižničné rutiny za behu programu pre C, C++.

Nasledujúce odstavce sa venujú jednotlivým častiam.



Obr. 3.1: Vytvorenie vlákien OpenMP konštrukciou parallel. Prevzaté z [7].

3.1.1 Direktívy a konštrukcie pre C, C++

Spustenie OpenMP direktív platí pre daný štrukturovaný blok alebo OpenMP konštrukciu. Každá direktíva začína frázou **#pragma omp**. Štrukturovaný blok je definovaný ako jeden výraz alebo skupina výrazov, ktorá obsahuje minimálne jeden začiatočný výraz pred jedným, nie u všetkých použiteľným, ukončujúcim výrazom. Konštrukcie OpenMP sú používané pre špecifikovanie pridelenia nezávislej práce jednému alebo viacerým vláknam. Základná konštrukcia parallel definuje paralelnú oblasť, ktorá sa nachádza pod direktívou v zložených zátvorkách. Hlavné vlákno vytvorí tím vedľajších vlákien a zároveň patrí k tomuto tímu ako je možné vidieť na obrázku 3.1. Veľkosť tímu je definovaná premennou prostredia OMP_NUM_THREADS, jej implicitná hodnota je nastavená podľa počtu vlákien procesoru. Na konci paralelnej oblasti sa nachádza automatická bariéra, kde sa všetky vlákna čakajú a po dokončení práce pokračuje ďalej už len hlavné vlákno, vedľajšie sa ukončia. Konštrukcia target definuje, že za ňou nasledujúci štrukturovaný blok bude vykonaný cieľovým zariadením, ktoré je v mojom prípade grafická karta. Ku každej direktíve sa vzťahujú klauzule, ktoré môžu byť použité.

Klauzule u direktívy ovplyvňujú vykonávanie danej direktívy. Zapisujú sa na koniec direktívy a musia byť oddelené medzerami. Nie všetky klauzule môžu byť použité pri každej direktíve. Možné použitie klauzúl v direktíve sa nachádza v definícií direktív od OpenMP. Pre už zmienenú konštrukciu parallel je možné použiť klauzulu, if(výraz), ktorá vytvára podmienené spustenie daného bloku. Pomocou num_threads() klauzuli sa určí veľkosť vytvoreného tímu pre daný blok. Dôležité pre prácu s premennými v bloku je využitie klauzúl *private()*, *firstprivate()* a *shared()*, s ktorými sa dá odstrániť závislosť pri prístupu k premenným. Východzia hodnota použitej premennej v rámci bloku je *shared*, ktorá určuje že premenná bude zdielaná medzi všetkými vláknami. Klauzula *private* určuje že každé vlákno si vytvorí vlastnú neinicializovanú premennú s ktorou bude pracovať a klauzula *firstprivate* určuje že každé vlákno si vytvorí vlastnú premennú, ktorá bude mať začiatočnú hodnotu definovanú v premennej. Prípadne existuje mnoho ďalších pre rôzne konštrukcie[5]. Niektoré ďalšie sú spomenuté v mojej implementácií 4.

3.1.2 Premenné prostredia

Hostiteľské a cieľové zariadenie, už pred začatím programu má vytvorené a inicializované premenné prostredia. Pri spustení programu sú načítané a pridelené všetky premenné prostredia hostiteľského zariadenia, ktoré boli nastavené používateľom. V rámci programu je možné načítať a modifikovať niektoré premenné prostredia cieľového zariadenia pomocou vytvorených rutín prístupu. Spomenutú premennú prostredia OMP_NUM_THREADS je možné načítať pomocou rutiny omp_get_max_threads() a modifikovať pomocou omp_set_num_threads(). Typ premennej je číslo. Všetky premenné prostredia a k ním príslušné rutiny je možné nájsť v referenčnej príručke pre OpenMP [5].

3.1.3 Knižničné rutiny za behu programu pre C, C++

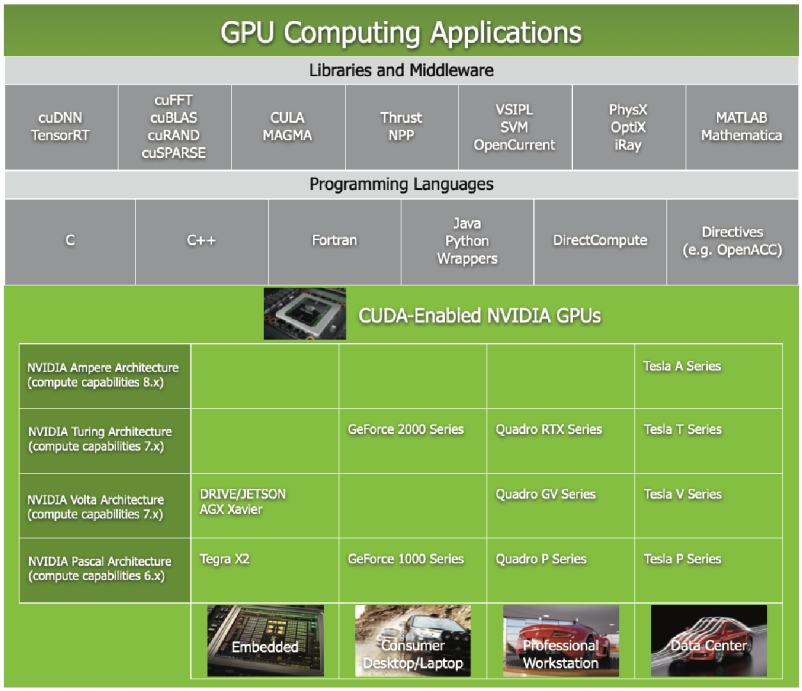
Spustením rutín prostredia je možné modifikovať a monitorovať vlákna, procesory a paralelné prostredie zariadení. Prvá skupina rutín sú spúšťacie rutiny pre paralelné prostredie, ktoré pracujú s premennými prostredia a upravujú spúšťanie aplikácie. Spomenuté rutiny pre premennú OMP_NUM_THREADS spadajú do tejto skupiny. Nachádzajú sa tu rutiny, ktoré pracujú s veľkosťou tímu, počtom vlákien a procesorov, miestom vykonávania vlákien, harmonogramom vykonávania, zariadeniami a paralelizmom. Druhá skupina sú zamykacie rutiny pracujúce so zamykaním častí kódu, nad ktorým musí pracovať iba jedno vlákno aby boli dáta konzistentné. Pri používaní zámkov musí používateľ predchádzať deadlock-u. Tretia skupina rutín načasovania podporuje hodinový časovač, ktorý prislúcha špecifickému vláknu a udáva čas jeho vykonávania. Posledná skupina správy pamäti cieľového zariadenia podporuje pridelenie a spravovanie ukazateľov pre prostredie dát cieľového zariadenia. Dokáže vytvárať, mazať a kopírovať dáta pre cieľové zariadenie, čo umožňuje cieľovému zariadeniu rýchlejší prístup k dátam.

3.2 CUDA

CUDA bola vytvorená spoločnosťou NVIDIA v novembri roku 2006 ako všeobecne účelová paralelne výpočtová platforma a programovací model. CUDA prepája paralelný výpočtový procesor v NVIDIA grafických kartách, ktorý sa používa pre riešenie mnoho zložitých výpočtových problémov, ktoré vypočíta mnohonásobne efektívnejšie ako procesor. CUDA prišla so softvérovým prostredím, ktoré dovoľuje vývojárom využiť C++ ako nízko úrovňový programovací jazyk. Podporuje aj ostatné jazyky ako C, Fortran, Java, Python a ďalšie. Obsahuje aj viacero knižníc a middleware, medzi ktoré patrí cuBLAS, Thrust, TensorRT a mnoho ďalších, ktoré je možné vidieť na obrázku 3.2. Najnovšie vydaná verzia je 11.6, ktorá podpore zariadenia s ovládačmi verzie väčšej ako 450.80.02* pre Linux x86_64 a väčšiu ako 452.39* pre Windows [13].

3.2.1 Funkcie pre grafické karty, KERNEL

CUDA umožňuje užívateľovi vytvárať funkcie, nazývané „kernel“, ktoré budu vykonané na grafickej karte N krát paralelne za použitia N rôznych CUDA vlákien. Definícia kernelu sa od obyčajných funkcií odlišujú použitím kľúčového slova ___global___ pred dátovým typom funkcie. Pri volaní kernelu sa medzi názov funkcie a argumentovú časť vkladá nová spúšťacia konfiguračná syntax <<<X,Y,Z>>>, v ktorej sa na prvom mieste nachádza počet blokov, na druhom veľkosť jedného bloku pre CUDA vlákno a na poslednom treťom mieste veľkosť zdieľanej pamäti v bytoch, ktorá je dynamicky alokovaná pre každý blok. Celkový



Obr. 3.2: Výpočtové aplikácie pre grafické karty. Prevzaté z [13].

počet vlákien je teda definovaný vynásobením počtu blokov a veľkosťou jedného bloku. Ku každému vláknu je možné pristúpiť cez premennú threadIdx, ktorá v sebe uchováva číslo vlákna. Aby bolo možné rozlíšiť, ktoré vlákna patria do akého bloku existuje premenná gridDim, ktorá v sebe uchováva číslo bloku pre dané vlákno.

CUDA poskytuje dve aplikačné programovacie rozhrania a pozostáva z troch častí:

- CUDA sada nástrojov,
- vysoká úroveň - CUDA Runtime API,
- nízka úroveň - CUDA Driver API.

Nasledujúce odstavce sa venujú jednotlivým častiam.

3.2.2 Runtime API

CUDA Runtime je implementovaná v cudart knižnici, ktorá je prepojená s aplikáciou buď staticky cez cudart.lib alebo libcudart.a alebo dynamicky cez cudart.ddl alebo libcudart.so. Aplikácie, ktoré vyžadujú dynamické prepojenie sú väčšinou zahrnuté ako časť inštalačného balíka aplikácie. Runtime API uľahčuje správu kódu na cieľovom zariadení poskytnutím implicitnej inicializácie správy kontextu a správy modulov. Tento prístup vedie k jednoduchšiemu kódu za cenu nedostatočnej prístupovej úrovne, ktorú poskytuje Driver API. Programovací model CUDA predpokladá, že systém pozostáva z hostiteľského a cieľového zariadenia a oba zariadenia majú svoju vlastnú oddelenú pamäť. Runtime vytvorí CUDA kontext pre všetky zariadenia v systéme. Tento kontext je hlavným kontextom pre zariadenie a je inicializovaný v prvej Runtime funkcií, ktorá vyžaduje už aktívny kontext

na zariadení. Kontext je zdieľaný medzi všetkými vláknami na hostiteľskom zariadení. Pri vytváraní kontextu môže byť kód pre cieľové zariadenie preložený práve v čas, čo sa stane jedine ak pri preklade nebolo dostupné cieľové zariadenie, a následne je zavedený do pamäte cieľového zariadenia. K hlavnému kontextu je možné pristúpiť cez Driver API. Kontext je možné synchronizovať zavolaním funkcie cudaDeviceSynchronize(), ktorá čaká kým sa ukončia všetky bežiace procesy na grafickej karte. Zavolaním funkcie cudaDeviceReset() na hostiteľskom zariadení je možné odstrániť hlavný kontext, ktorý je používaný na cieľovom zariadení. Nasledujúca volaná funkcia z Runtime knižnice, vytvorí nový hlavný kontext pre cieľové zariadenie.[15]

Práca s pamäťou

KERNEL pracuje s pamäťou na cieľovom zariadení, preto sa v Runtime knižnici nachádzajú funkcie pre pridelenie a zánik pamäti, cudaMalloc() a cudaFree() a pre presun dát medzi hostiteľskou a cieľovou pamäťou, cudaMemcpy(). Pamäť cieľového zariadenia môže byť pridelená ako lineárna pamäť alebo CUDA polia. Lineárna pamäť je pridelená ako zjednotený blok v adresnom priestore, ku ktorej je možné pristúpiť pomocou ukazateľa. Ukazatele je možné použiť v abstraktných dátových štruktúrach. Veľkosť adresného priestoru závisí na hostiteľskom systéme (procesore) a výpočtovej schopnosti použitej grafickej karty. Je možné vytvoriť tri typy zdielanej pamäti. Prvý typ je špecifikovaný použitím špecifikátoru pamäťového priestoru ___shared___. Zdieľaná pamäť je zdieľaná medzi CUDA vláknami v rámci bloku, ktoré vykonávajú špecifický KERNEL, čo znamená, že zmena pamäti vykonaná jedným vláknom je viditeľná pre všetky vlákna v bloku. Druhý typ je špecifikovaný použitím špecifikátoru pamäťového priestoru ___device___. Tento typ zdielanej pamäti umožňuje vytvoriť premennú, ktorá je viditeľná medzi všetkými blokmi a vláknami, ktoré vykonávajú špecifický KERNEL. Pomocou špecifikátoru ___constant___ je možné vytvoriť zdieľanú premennú tretieho typu, ktorá je viditeľná len pre čítanie všetkými blokmi a vláknami. Klasická deklarácia premennej vytvorí pamäť v registri čipu [20].

3.2.3 CUDA sada nástrojov

CUDA sada nástrojov poskytuje vývojové prostredie pre vytváranie vysoko výkonných aplikácií za použitia grafickej karty ako akcelerátoru. S touto sadou je možné vyvíjať, optimalizovať a vydávať aplikácie pre systémy založené na použití grafickej karty ako akcelerátora, pracovné stanice, data centrá, cloudové platformy a vysoko výkonné superpočítače. Sada nástrojov obsahuje knižnice pre prácu s grafickou kartou, ladiace a optimalizačné nástroje, kompilátor pre C a C++ a Runtime API pre vybudovanie a vydanie aplikácií na hlavných architektúrach ako x86, Arm a POWER [10]. Sada nástrojov obsahuje viacero knižníc medzi, ktoré patrí aj štandardná knižnica CUDA C++, ktorá môže byť použitá pre grafickú kartu alebo procesor a pre kominikáciu medzi nimi. Spomenuté CUDA Driver API taktiež patrí do tejto sady. Nad CUDA Driver API je postavená CUDA Runtime API, keďže Driver API pracuje ešte na nižšej úrovni ako Runtime API. Preto pre vývoj aplikácií stačí použiť CUDA Runtime API. Ako ďalšia knižnica z CUDA sady nástrojov, ktorá stojí za zmienku je CUDA Math API. CUDA Math API podporuje dva vlastné dátové typy a to dátový typ half, half2, bfloat16 a bfloat162. Ďalej obsahuje všetky základné matematické funkcie, ktoré sú navrhnuté pre vykonanie na grafickej karte. V CUDA sade nástrojov sa nachádza nástroj Nsight Compute, ktorý som využil pre profilovanie mojej CUDA aplikácie. Všetky spomenuté nástroje sú len malou časťou CUDA sady nástrojov. Všetky dostupné nástroje a k nim veľmi detailná dokumentácia sa nachádza na stránke [16]. Najnovšia verzia CUDA

nástrojov je 11.6. V tejto verzií pribudol nový dátový typ ___int128, pre ktorý vznikla podpora v kompilátoroch a vývojárskych nástrojoch.
Nasledujúce knižnice taktiež patria do tejto sady nástrojov.

3.3 cuBLAS - CUDA Basic Linear Algebra Subrutines

CuBLAS knižnica je implementácie knižnice BLAS postavená nad CUDA Runtime. Pre použitie cuBLAS API aplikácia musí prideliť potrebné matice a vektory do pamäťového priestoru grafickej karty, naplniť ich dátami, zavolať postupne požadované cuBLAS funkcie a následne skopírovať výsledok z pamäťového priestoru grafickej karty do hostiteľského pamäťového priestoru [12]. Taktiež poskytuje pomocné funkcie pre zápis a načítanie dát z grafickej karty. Všetky knižničné funkcie vracajú chybový stav dátového typu cublasStatus_t. Pre použitie výpočtových funkcií cuBLAS je potrebné vytvoriť rukoväť ku kontextu zavolaním funkcie cublasCreate(), ktorá je predávaná pri každom zavolaní funkcie a má pridelený dátový typ cublasHandle_t. Pred ukončením aplikácie musí byť zavolaná funkcia cublasDestroy() pre uvoľnenie zdrojov spojených s kontextom. Tento prístup umožňuje kontrolovať nastavenia knižnice keď hostiteľský program vykonáva viac vlákien a viacero grafických kariet. V aplikácií je možné použiť funkciu cudaSetDevice() aby spojila rôzne zariadenia s rôznymi hostiteľskými vláknami. Pre každé vlákno môže byť inicializovaná jedinečná rukoväť, ktorú vlákno použije pre príslušné zariadenie.
CuBLAS knižnica, rovnako ako BLAS, pozostáva z troch úrovní výpočtových funkcií:

- úroveň prvá - obsahuje funkcie, ktoré vykonávajú základné skalárne a vektorové operácie,
- úroveň druhá - obsahuje funkcie, ktoré vykonávajú maticovo-vektorové operácie,
- úroveň tretia - obsahuje funkcie, ktoré vykonávajú maticovo-maticové operácie.

Ďalej knižnica obsahuje cuBLAS Pomocné funkcie, ktoré pracujú s kontextom a BLAS-podobné rozšírenie s doplňujúcimi výpočtovými funkciami.

3.4 Thrust

Thrust je C++ knižnica šablón, založená na štandardnej knižnici šablón (STL), pre CUDA aplikácie. Knižnica poskytuje bohatú kolekciu dátových paralelných primitív, ktoré dokážu skenovať, triediť a redukovať vektory. Thrust poskytuje dva vektorové kontajnery a to kontajner *host_vector* a kontajner *device_vector*. Kontajner host vytvára vektor v pamäťovom priestore hostiteľského zariadenia (procesora) a kontajner device vytvára vektor v pamäťovom priestore cieľového zariadenia (grafickej karty). Tieto kontajnery majú rovnaké vlastnosti ako kontajner *std::vector* zo štandardnej knižnici šablón. Thrust obsahuje menný priestor *thrust::*, s ktorým je možné vytvárať kontajnery a volať vnútorné funkcie. Pre prácu s vektormi knižnica poskytuje funkcie fill, sequence a copy pre naplnenie a kopírovanie vektorov. Ďalej knižnica obsahuje prepychové iterátory, ktoré dokážu nadobúdať rôzne hodnoty vo viacerých postupnostiach, ako napríklad konštantný iterátor má konštantné hodnoty, ku ktorým je možné pristupovať ako k poliam. Thrust poskytuje algoritmy pre prácu s iterátormi a vektormi. Funkcia redukcie potrebuje špecifikovať binárnu operáciu. Za pomoci tejto operácie zredukuje prijatý vektor alebo iterátor na jednu výslednú hodnotu, ktorú funkcia vráti. Nachádzajú sa tu aj funkcia transform, ktorá dokáže vykonávať základné vektorové

operácie a uložiť výsledný vektor. Funkcia príma funktory, ktoré sú prístupné cez Menný priestor knižnice. Funktory pozostávajú zo základných binárnych operácií ako sú sčítanie a násobenie a sú volané ako *thrust::plus<T>* a *thrust::multiplies<T>*, kde T označuje dátový typ hodnôt. Za pomoci knižnice je možné implementovať komplexné algoritmy so stručným zdrojovým kódom [2].

Kapitola 4

Implementácia

Pre túto prácu som vytvoril dve aplikácie, ktoré budú spúšťať vybrané operácie nad maticami a vektormi. Aplikácie majú za úlohu spustiť sadu testov na grafickej karte, odstopovať čas vykonávania a vypočítať dosiahnutý výkon pre jednotlivé operácie. Jednu aplikáciu som vytvoril za použitia jazyka C++ a knižnice OpenMP. Druhú aplikáciu som vytvoril za použitia jazyka C++, CUDA Runtime API a knižnice cuBLAS. V tejto kapitole sa nachádza detailný popis vytvorených aplikácií.

4.1 Trieda Test pre OpenMP aplikáciu

Trieda *Test* sa nachádza v súbore test.cpp. V triede sa nachádzajú nasledujúce privátne premenné, pole s názvom matrixes, štrukturovaného dátového typu Matrixes, pole s názvom vectors, štrukturovaného dátového typu Vectors a ukazateľ s názvom repetition, dátového typu integer. Veľkosť týchto polí je definovaná makrom s názvom numSizes. Štrukturovaný dátový typ Matrixes obsahuje premenné size, dátového typu integer, pre definovanie rozmerov matíc a tri dvojité ukazatele s názvami A, B, C, dátového typu float, pre vytvorenie dvojrozmerných polí - matíc. Štrukturovaný dátový typ Vectors obsahuje podobne ako štrukturovaný dátový typ Matrixes premennú size, dátového typu integer a tri tentokrát obyčajné ukazatele s názvami vA, vB, vC, dátového typu float, pre vytvorenie jednorozmerných polí - vektorov. Tieto štrukturované dátové typy slúžia pre lepší a kompaktnejší prístup k maticiam a vektorom s rovnakou veľkosťou, keďže sú výpočty spúšťané sekvenčne po veľkostiach. Hodnoty z poľa repetition sú použité pri volaní funkcií pre viacnásobný výpočet v slučke. Tento viacnásobný výpočet je použitý kvôli meniacemu sa času výpočtu pri zavolaní rovnakej funkcie viackrát za použitia rovnakých parametrov. Zmena času výpočtu môže byť zavinená nestálou teplotou zariadenia a vyťažením zariadenia. Preto som tieto negatívne vplyvy odstránil použitím viacnásobného výpočtu v slučke a spriemerovaním výsledného času výpočtu. Pre najmenšie veľkosti je časovo prijateľné spúšťať výpočet s väčším počtom opakovaní, čím bude dosiahnutá aj väčšia presnosť, a zase naopak väčšie veľkosti spúšťať s menším počtom opakovaní.

Konštruktor triedy očakáva ako prvú premennú pole s názvom sizes, v ktorom sa nachádzajú požadované veľkosti pre pridelenie pamäti a druhú premennú pole s názvom repet, v ktorom sa nachádzajú požadovaný počet opakovaní pre viacnásobný výpočet v slučke. Tieto polia spolu súvisia a to tak, že volanie funkcií pre veľkosť nachádzajúcu sa v pamäti premennej sizes na indexe 0 budú zavolané s počtom opakovaní nachádzajúcim sa v pamäti premennej repet na indexe 0 atď. Konštruktor uloží premennú repet do svojej privátnej

premennej repetition, uloží sekvenčne do premenných matrixes a vectors veľkosti do premennej size, pridelí požadovanú veľkosť pamäti pre odpovedajúce ukazatele a naplní ich testovacími dátami. K prideleniu a naplneniu pamäti matíc využíva privátne metódy AllocMemoryMat() a FillMemoryMat(), ktoré očakávajú argument dátového typu Matrixes a k prideleniu a naplneniu pamäti vektorov využíva privátne metódy AllocMemoryVect() a FillMemoryVect(), ktoré očakávajú argument dátového typu Vectors.

Trieda obsahuje funkcie s názvom Calculate 4.1 pre každú funkciu zo súboru microBenchmarks.cpp, ktorému sa venujem v nasledujúcej sekcií. Všetky funkcie Calculate sa starajú o presun matíc a vektorov do pamäťového adresného priestoru grafickej karty za použitia OpenMP konštrukcie „#pragma omp target enter data map(to:)“, slučky, ktorá v sebe volá príslušnú funkciu zo súboru microBenchmarks.cpp. Ďalej už nasleduje uloženie štartovacieho času pred začatím výpočtu, za pomoci knižnice chrono, z ktorej využívam triedu high_resolution_clock, ktorá poskytuje uloženie presného času pomocou metódy now(). Pred spustením funkcie jacobi a matMul sa nachádza OpenMP konštrukcia „#pragma omp target update to()“, ktorá vždy pred volaním funkcie nastaví hodnoty potrebných premenných v pamäti grafickej karty na počiatočné aby funkcia v slučke počítala vždy s totožnými hodnotami a aby bol výsledok operácií po ukončení slučky správny. Po ukončení slučky sa uloží ukončujúci čas výpočtu. Ďalej nasleduje OpenMP konštrukcia „#pragma omp target exit data“ pre uvoľnenie pridelenej pamäti v adresnom priestore grafickej karty, pri väčšine funkcií sa ešte za direktívou nachádza klauzula map(from:), kde sa určí, ktorá premenná má byť stiahnutá z pamäťového adresného priestoru cieľového zariadenia do pamäťového adresného priestoru hostiteľského zariadenia aby výsledok výpočtu nebol stratený. Následne sa vypočíta celkový strávený čas výpočtom a to odčítaním štartovacieho času od ukončujúceho času a vydelením výsledného času počtom opakovaní pre získanie priemerného času. Priemerný čas strávený výpočtom je vypísaný na štandardný výstup programu. Z priemerného času je vypočítaný a vypísaný na štandardný výstup počet operácií s pohyblivou rádovou čiarkou za sekundu – FLOPS, táto metrika sa využíva pre meranie výkonu počítača. Výpočet FLOPS je daný vydelením počtu potrebných operácií dosiahnutým výpočtovým časom. Pri funkcií kopírovania matíc a transponovaní matice bude z dosiahnutého výpočtového času vypočítaná a vypísaná na štandardný výstup, priepustnosť pamäti grafickej karty. Priepustnosť pamäti grafickej karty je udávaná v jednotke GB za sekundu. K výpočtu počtu operácií sa ešte dostaneme v ďalších odstavcoch a k porovnaniu dosiahnutých FLOPS a priepustnosti v kapitole 5.

```
void CalculateMatrixMul(int N, float** A, float** B, float** C, int loop){
  #pragma omp target enter data map(to:A[0:N][0:N], B[0:N][0:N], \
  C[0:N][0:N])
  auto timeStart = chrono::high_resolution_clock::now();

  for (int i = 0; i < loop; ++i){
    #pragma omp target update to(C[0:N][0:N])
    matMul(N, A, B, C);
  }

  auto timeEnd = chrono::high_resolution_clock::now();
  #pragma omp target exit data map(from:C[0:N][0:N])

  auto duration = \
  chrono::duration_cast<chrono::nanoseconds>(timeEnd - timeStart);
  double realDuration = duration.count()/loop;
  cout << "Spend in multiplication of " << N*N*sizeof(float) << \
  "B arrays(" << N << "): " << realDuration << "ns\n";
  cout << "Flops of multiplication of " << N*N*sizeof(float) << \
  "B arrays(" << N << "): " << (2*pow(N, 3))/realDuration << " GFlops\n";
  MatrixCheck(N, C);
}
```

Výpis 4.1: Funkcia Calculate pre násobenie matíc

Trieda test obsahuje metódu „CallOmp“, ktorá v slučke volá všetky metódy calculate, sekvenčne pre zadané veľkosti. Pred zavolaním prvej metódy vypíše na štandardný výstup veľkosť, s ktorou sa ide aktuálne pracovať a počet opakovaní pre viacnásobné volanie funkcií pre výpočet na grafickej karte. Ďalej už volá funkcie calculate s príslušnými maticami a vektormi, veľkosťami matíc, z privátnej premennej matrixes a vectors a príslušným počtom opakovaní. Metóda „CallOmp“ očakáva dva argumenty a to argument iterations, dátového typu integer, ktorý určuje maximálny počet iterácií a argument precision, dátového typu float, ktorý určuje minimálnu presnosť pre ukončenie výpočtu. Oba argumenty sú predané funkcií pre výpočet Jacobiho iteračnej metódy.

Ďalej trieda obsahuje privátne metódy pre testovanie výpočtov, ku ktorým sa dostaneme v kapitole výsledky.

4.2 Sada testovacích problémov pre OpenMP aplikáciu

Sada testovacích problémov sa nachádza v súbore *microBenchmarks.cpp*, v ktorom sa nachádzajú implementácie funkcií za použitia knižnice OpenMP pre výpočet na grafickej karte. Dokument [6] obsahuje vysvetlenie a príklady použitia všetkých OpenMP konštrukcií, premenných prostredia a rutín pre verziu 5.0.1, ktorý som využil pri vytváraní mojich vlastných funkcií. Vytvoril som nasledovné funkcie.

4.2.1 Maticové násobenie

Cieľom benchmarku je otestovať hrubý výkon grafickej karty na maticiach. Funkcia očakáva tri následovné argumenty, argument s názvom N, dátového typu integer, určujúci veľkosť

matíc, argumenty s názvom A a B, dátového typu dvojitý ukazateľ float, obsahujú vstupné matice, argument s názvom C, dátového typu dvojitý ukazateľ float, obsahuje výstupnú maticu a dátový typ návratovej hodnoty je void. Funkcia očakáva, už prítomné matice v pamäťovom adresnom priestore grafickej karty s rovnakými názvami premenných, o čo sa stará funkcia CalculateMatrixMul() v triede Test. V implementácií 4.2 je použitá zložená OpenMP konštrukcia „#pragma omp target teams distribute parallel for collapse(2)“. Direktíva target určuje, že výpočet bude vykonaný na grafickej karte. Direktíva teams vytvára spolok tímov podľa počtu SM (Streaming Multiprocessor), ktoré sú nezávisle spúšťané v štrukturovaných blokoch. Direktíva distribute špecifikuje, že iterácie dvoch najbližších slučiek budú spustené a rozdistribuované medzi vytvorené tímy vlákien. Direktíva parralel for znova paralelizuje a rozdistribuuje už vytvorené vlákna. Klauzula collapse(2) určuje, že rozdelenie iterácií bude rozdelené nie len pre prvú najbližšiu slučku ale práve medzi dve najbližšie slučky. Za prvými dvomi slučkami nasleduje kód, ktorý bude vykonávaný každým jedným vláknom paralelne a to následovne, vytvorí si premennú s názvom sum, dátového typu float, a nastaví jej hodnotu 0.0, nasleduje slučka, v ktorej sa vykonáva násobenie riadku so stĺpcom matice a pričíta výsledok každej iterácie do premennej sum. Použitú konštrukciu som vybral po otestovaní viacerých konštrukcií, ku ktorému sa dostaneme v kapitole 5. Keďže operácia maticového násobenia je zo všetkých naimplementovaných operácií výpočtovo najnáročnejšia vykonával som na nej najviac testov. Touto implementáciou som dosiahol najnižší výpočtový čas. Počet operácií je daný výrazom $2 * N^3$, pretože vykoná 2 operácie pre matice o rozmere N*N Nkrát, čo pri veľkosti dimenzií matice 1000 robí 2,000,000,000 operácií.

```
void matMul(int N, float** A, float** B, float** C){
    #pragma omp target teams distribute parallel for collapse(2)
    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++){
            float sum = 0.0;
            for (int k = 0; k < N; k++){
                sum += A[i][k] * B[k][j];
            }
        C[i][j] = sum;
        }
    }
}
```

Výpis 4.2: Násobenie matíc

4.2.2 Skalárny súčin

Cieľom tohoto benchmarku je otestovať grafickú kartu pri práci s vektormi. Funkcia očakáva tri nasledovné argumenty, argument s názvom N, dátového typu integer, určujúci veľkosť vektorov, argumenty s názvom vA a vB, dátového typu ukazateľ float, obsahujúce vstupné vektory a dátový typ návratovej hodnoty funkcie je float, v ktorej vráti výsledok skalárneho súčinu. Funkcia očakáva, už prítomné vektory v pamäťovom adresnom priestore grafickej karty s rovnakými názvami premenných, o čo sa stará funkcia CalculateScalarMul() v triede Test. V implementácií 4.3 je použitá OpenMP konštrukcia „#pragma omp target teams distribute parallel for shared(vA, vB, N) default(none) reduction(+:sum)“. Spojená

konštrukcia target teams distribute parallel for je vysvetlená v predchádzajúcej sekcií a je použitá aj tu, pretože dosahuje najnižší výpočtový čas. Direktíva obsahuje klauzulu shared(vA, vB, N) ktorá určuje, že premenné vA, vB a N budú zdielané medzi vláknami, čo znamená že všetky vlákna budú pristupovať do tej istej premennej v pamäti, východzia hodnota všetkých použitých premenných v rámci bloku je zdielaná ak nie je určené inak. Klauzula default(none) zaručuje, že všetky použité premenné v rámci bloku sú priradené aspoň v jednej klauzule shared, private, firstprivate alebo lastprivate, ak nie sú, preklad bude hlásiť chybu. Ako ďalšiu obsahuje klauzulu reduction(+:sum), ktorá určuje premennú sum, používanú pri sčítaní, ako kritickú, čo znamená že k nej môže pristúpiť maximálne jedno vlákno v špecifickom čase. Do premennej sum je pričítaný výsledok násobenia vektorov v každej iterácií slučky a po ukončení výpočtu je premenná sum vložená do návratovej hodnoty funkcie. Počet operácií je daný výrazom $2 * N$, pretože sú vykonávané 2 operácie pre vektory o veľkosti N.

```
float scalarMul(int N, float* vA, float* vB){
    float sum = 0.0;
    #pragma omp target teams distribute parallel for shared(vA, vB, N) \
    default(none) reduction(+:sum)
    for (int i = 0; i < N; i++)
        sum += vA[i] * vB[i];
    return sum;
}
```

Výpis 4.3: Skalárny súčin vektorov

4.2.3 Súčet matíc a vektorov

Cieľom týchto benchmarkov je otestovať operáciu sčítania pre vektory aj matice. V implementacií 4.4 je použitá OpenMP konštrukcia, ktorá maximálne paralelizuje výpočet. Pri maticovom sčítaní je použitá klauzula collapse(2) aby boli paralelizované obe slučky.

```
void matAdd(int N, float** A, float** B, float** C){
    #pragma omp target teams distribute parallel for collapse(2)
  for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
        C[i][j] = A[i][j] + B[i][j];
}

void vectAdd(int N, float* vA, float* vB, float* vC){
  #pragma omp target teams distribute parallel for
    for (int i = 0; i < N; ++i)
        vC[i] = vA[i] + vB[i];
}
```

Výpis 4.4: Sčítanie matíc a vektorov

4.2.4 Kopírovanie Pamäti

Cieľom tohoto benchmarku je otestovať rýchlosť presunu troch matíc do pamäti grafickej karty. Vo funkcií memCpy 4.5 sa nachádza jediná OpenMP konštrukcia „#pragma omp target map(to:A[0:N][0:N], B[0:N][0:N], C[0:N][0:N])“, ktorá poukazuje na jednoduchosť pridelenia a nastavenia hodnôt v pamäťovom priestore grafickej karty pre dvojdimenzionálne polia. Prenášané polia sú prímané cez argumenty funkcie a ako ďalší argument s názvom N je očakávaná veľkosť dimenzie, aby bolo možné presne definovať aká veľkosť dvojdimenzionálneho poľa má byť skopírovaná. Táto konštrukcia vytvára pamäť na grafickej karte, ktorá je prístupná len vrámci jej bloku, po dokončení bloku je táto pamäť uvoľnená. Dôvod výberu nelineárnej pamäti sa nachádza v ďalšej kapitole 5. Pre výpočet priepustnosti bude prenášaná veľkosť dát daná výrazom $3 * N^2$ v bajtoch.

```
void memCpy(int N, float** A, float** B, float** C){
  #pragma omp target map(to:A[0:N][0:N], B[0:N][0:N], C[0:N][0:N])
  {
  }
}
```

Výpis 4.5: Kopírovanie pamäti

4.2.5 Transponovanie matice

Pri transponovaní matíc nedokážeme namerať počet operácií pretože čítanie a zápis do pamäťového priestoru nepatria medzi operácie s pohyblivou rádovou čiarkou. Preto tento benchmark bude testovať priepustnosť pamäti. Veľkosť prenášaných dát bude o veľkosti matice. Funkcia CalculateTranspose() pridelí potrebnú pamäť na cieľovom zariadení, odstopuje a vypíše strávený čas výpočtom na štandardný výstup programu. Samotná implementácia 4.6 je veľmi jednoduchá a je paralelizovaná pre získanie najvyššieho výkonu OpenMP konštrukciou „#pragma omp target teams distribute parallel for collapse(2)“.

```
void transpose(int N, float** A, float** C){
  #pragma omp target teams distribute parallel for collapse(2)
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j){
      C[j][i] = A[i][j];
    }
}
```

Výpis 4.6: Transponovanie matice

4.2.6 Jacobiho iteračná metóda

Tento benchmark má za úlohu spustiť tri výpočty na grafickej karte za sebou. Prvý výpočet konvergencie počíta nezávisle od ostatných. Výpočet neznámych v iterácií je potrebný pre výpočet odchýlky v nasledujúcom volaní kernelu. Preto je tretí výpočet závislý na druhom a musí čakať na jeho ukončenie. Cieľom benchmarku je otestovať viacnásobné volanie troch kernelov. Funkcia jacobi B.1 očakáva sedem nasledujúcich argumentov:

- argument s názvom N, dátového typu integer, definujúci veľkosť dimenzií vektorov a matíc
- argument s názvom A, dátového typu dvojitý ukazateľ float, obsahuje symetrickú maticu s ľavou stranou matice
- argument s názvom D, dátového typu ukazateľ float, obsahuje vektor s pravou stranou matice
- argument s názvom X, dátového typu ukazateľ float, obsahuje vektor s odhadnutými hodnotami, ktoré budú použité ako aproximatívne riešenie pri prvej iterácií
- argument s názvom Xo, dátového typu ukazateľ float, obsahuje vektor, ktorý bude použitý pri výpočte pre uschovanie hodnôt z predchádzajúcej iterácie
- argument s názvom iterations, dátového typu integer, definujúci maximálny počet iterácií, ktoré majú byť vykonané
- argument s názvom precision, dátového typu float, definujúci minimálnu hodnotu odchýlky pre ukončenie výpočtu

Návratová hodnota funkcie, dátového typu integer, vracia celkový počet vykonaných iterácií. Výpočet funkcií pozostáva z troch volaní na grafickú kartu. Prvé volanie je vytvorené OpenMP konštrukciou „#pragma omp target distribute parallel (&&:converg)“, ktorá overuje konvergenciu matice pomocou výpočtu, ktorý overí či je matica ostro riadkovo diagonálne dominantná. Výpočet spočíva v súčte absolútnych hodnôt nediagonálnych prvkov v riadkoch a v následnom porovnaní absolútnej hodnoty na diagonále s daným súčtom riadku. Ak je táto hodnota pre každý riadok väčšia matica je ostro riadkovo diagonálne dominantná. OpenMP konštrukcia obsahuje klauzulu reduction(&&:converg), ktorá označuje premennú converg, dátového typu boolean, ako kritickú, pretože k nej môže pristúpiť viacero vlákien naraz. Bez tejto klauzule by výsledok logického násobenia nebol správny. Počas výpočtu je paralelizovaný aj súčet hodnôt v riadku za pomoci OpenMP konštrukcie „#pragma omp parallel for reduction(+:sum)“. Všetky vlákna pre riadok sčítavajú premennú sum, ktorá sa musí nachádzať v redukcií inak by výsledok nebol správny. Po ukončení výpočtu v sebe premenná converg obsahuje hodnotu True, ak je matica ostro riadkovo diagonálne dominantná alebo hodnotu False, ak nie je. Pre získanie absolútnych hodnôt je vo výpočte použitá funkcia fabs() z knižnice math pre programovací jazyk C. Nasleduje výmena ukazateľov Xo a X, pretože následujúci výpočet potrebuje v premennej Xo hodnoty, z ktorými bude pracovať a výsledok bude ukladať do vektoru X. Nasleduje slučka, ktorá vykonáva jednotlivé iterácie Jacobiho iteračnej metódy. Slučku nie je možné paralelizovať, pretože nasledujúca iterácia potrebuje výsledok predošlej iterácie. V slučke sa nachádzajú tri OpenMP konštrukcie. Konštrukcie „#pragma omp target teams distribute“ a „#pragma omp parallel for reduction(-:sum)“, hľadajú riešenie sústavy N lineárnych rovníc o N neznámych následovne. Pre každú rovnicu vyjadríme neznámu hodnotu X na diagonále. Neznáma X na prvom riadku je vyjadrená výrazom $X_1 = \frac{1}{A_{1,1}}(D_1 - A_{1,2}X_2 - A_{1,3}X_3 - ... - A_{1,N}X_N)$ takto postupne vyjadrujeme hodnotu pre každý riadok až po riadok N, ktorý je daný výrazom $X_N = \frac{1}{A_{N,N}}(D_N - A_{N,1}X_1 - A_{N,2}X_2 - ... - A_{N,N-1}X_N - 1)$. Výsledok výpočtu je uložený vo vektore X. Redukcia pre premennú sum je použitá rovnako ako pri výpočte konvergencie ale tentokrát pre odčítanie. Po ukončení výpočtu nasleduje tretia OpenMP konštrukcia „#pragma omp target teams distribute parallel reduction(&&:prec)“, ktorá počíta dosiahnutú presnosť odčítaním výsledku predošlej iterácie, uloženej vo vektore Xo, od výsledku

aktuálnej iterácie, uloženej vo vektore X. OpenMP klauzula reduction(&&:prec), spravuje prístup k premennej prec, dátového typu boolean, pretože k nej pristupuje viacej vlákien zároveň, podobne ako u premennej converg. Ak sú absolútne hodnoty rozdielov neznámych, zo všetkých riadkov, menšie ako minimálna zadaná presnosť výpočtu, tak sa v premennej prec nachádza hodnota True, čo znamená že sa funkcia ukončí a vráti počet vykonaných iterácií. Ak ešte nebola dosiahnutá potrebná presnosť výpočtu vykoná sa výmena ukazateľov a program pokračuje v ďalšej iterácií, pokiaľ nedosiahne zadaný počet iterácií. Výmenu ukazateľov vykonáva funkcia swap() z C++ štandardnej knižnice algorithm. Výsledok poslednej iterácií sa nachádza vo vektore X. O stiahnutie výsledku z pamäťového priestoru cieľového zariadenia a overenie správnosti výpočtu sa stará funkcia CalculateJacobi() v triede Test. Približný počet vykonaných operácií je definovaný výrazom $N^2 + 2 * N^2 * result$. Počet operácií ovplyvňuje počet vykonaných iterácií, ktorý obsahuje premenná result.

4.3 Kompilátor pre OpenMP aplikáciu

Pri výbere kompilátoru jazyka C++ s podporou OpenMP som narazil na mnoho problémov. Skúšal som použiť kompilátory GCC a Clang, pri ktorých sa mi nepodarilo úspešne preložiť program. Na oboch prekladačoch prebieha implementácia OpenMP verizie 5.0. Podpora stále rastie a väčšina základných konštrukcií už je podporovaná. Podporujú presun programu na grafickú kartu, avšak stále nie je úplná a kompletná. Pre preklad programu je potrebné spúšťať kompilátory s prepínačom *-fopenmp* a v implementačnom súbore importovať knižnicu *omp.h*. Ako ďalší som použil kompilátor **nvc++** z NVIDIA HPC software developer kit-u. Podporuje štandard C++17, C++ Standard Parallelism knižnicu, OpenACC a OpenMP pre NVIDIA grafické karty a multijadrové procesory.[18] Kompilátor dokáže automaticky predať paralelné algoritmy z jazyka C++ na NVIDIA karty založené na Volta, Turing alebo Amprere architektúrach. Tieto grafické karty obsahujú vlastnosti, ktoré boli špeciálne vyvinuté pre podporu vysokovýkonných, všeobecne účelových paralelných programovacích modelov ako je C++ Parallel Algorithms či OpenMP. Kompilátor dokáže automaticky zistiť a vygenerovať kód pre typ grafickej karty, ktorý sa nachádza v systéme. Podporuje aj definovanie presnej architektúry pomocou prepínača -gpu=ccXX. Minimálna podporovaná CUDA verzia je 10.1. Pre preklad nie je potrebné mať nainštalovanú CUDA sadu nástrojov. Môj program som prekladal pomocou príkazu 4.7. Prepínač -mp=gpu určuje, že kompilátor bude paralelizovať OpenMP regióny, ktoré budú presunuté na NVIDIA grafickú kartu. Prepínač -Minfo=mp umožní kompilátoru pri preklade vypisovať informácie o preklade na štandardný chybový výstup. Prepínač -O3 zapne mnoho optimalizačných vlajok, ktoré zvýšia čas kompilácie a zároveň výkon generovaného kódu.

```
nvc++ test.cpp -o test -mp=gpu -Minfo=mp -O3
```

Výpis 4.7: Preklad OpenMP aplikácie

4.4 Trieda CuTest pre CUDA aplikáciu

Trieda CuTest sa nachádza v súbore cutest.cpp. Trieda pracuje s rovnakými štrukturovanými dátovými typmi Matrixes a Vectors ako trieda Test. Konštruktor pridelí a nastaví potrebnú pamäť pre všetky zadané veľkosti a taktiež očakáva pole repet pre viacnásobné volanie funkcií. Trieda CuTest nepracuje s dvojdimenzionálnymi poliami ako pri Triede Test

ale pracuje s lineárnou pamäťou o veľkosti dvojdimenzionálneho poľa a prístup k určitému prvku matice je vyjadrený pomocou výrazu $riadok * velkostmatice + stlpec$. Lineárna pamäť je použitá kvôli cuBLAS funkciám, ktoré očakávajú v argumentoch matice s lineárnou pamäťou pre lepšiu kompatibilitu s jazykmi C a C++. Funkcie cuBLAS používajú Fortran prostredie, ktoré ukladá dvojdimenzionálne matice podľa stĺpca a používa indexovanie od 1, narozdiel od jazyka C a C++, ktoré ukladajú dvojdimenzionálne matice podľa riadku a používajú indexovanie od 0. Indexovanie pre C a C++ je možné upraviť pomocou makier z dokumentácie [12]. Ďalej Trieda CuTest obsahuje metódy Calculate pre rovnaké operácie ako boli použité v súbore microBenchmark.cpp, avšak tentokrát za použitia cuBLAS funkcií. Pre Jacobiho iteračnú metódu som vytvoril CUDA KERNEL-i za použitia CUDA Runtime API. Použité funkcie a CUDA KERNEL-i sú vysvetlené v sekcií Sada testovacích problémov pre CUDA aplikáciu. Metódy Calculate sa starajú o pridelenie a presun hodnôt do pamäťového priestoru grafickej karty za použitia funkcií z knižnice Runtime. Metódy Calculate sa tiež starajú o prípravu potrebných premenných pre zavolanie funkcií pre výpočet operácií z knižnice cublas_v2 a cublas. Pre kompaktnejšie volanie týchto funkcií som vytvoril súbor cublasOperations.cpp, ktorý je vysvetlený v nasledujúcej sekcií. Trieda Cutest obsahuje metódu CallCuda(), ktorá sa stará o sekvenčné vykonávanie funkcií pre všetky definované veľkosti a o výpis veľkosti a počtu opakovaní pre aktuálne počítanú veľkosť.

4.5 Cublas Operácie

Všetky funkcie z cuBLAS a Runtime knižnice vracajú návratovú hodnotu, dátového typu cudaError_t, ktorá v sebe obsahuje status vykonania. Pri úspešnom vykonaní funkcia vráti status CUBLAS_STATUS_SUCCESS. Funkcie tiež môžu vrátiť chybový kód z predchádzajúceho asynchrónneho spustenia. Možné chybové kódy je možné nájsť v referenčnej príručke ku funkciám.[15][12] Všetky mnou vytvorené funkcie v súbore cublasOperations.cpp obsahujú kontrolu tohoto statusu a pri neúspechu vypíšu na štandardný chybový výstup príslušnú chybovú hlášku, resetujú doposiaľ vytvorený kontext pomocou funkcie cudaDeviceReset() a vrátia hodnotu -1. Pri úspešnom vykonaní vrátia hodnotu 0. Pre zavolanie funkcie pre výpočet operácie je potrebné:

- Definovať, na ktorom zariadení bude výpočet prebiehať. Funkcia setDevice() sa stará o zavolanie funckcie cudaSetDevice(), z knižnice cuBLAS, ktorá definuje grafickú kartu podľa predaného čísla do funkcie. Grafické karty sú očíslované v systéme od nuly.

- Vytvoriť rukoväť pre kontext, o čo sa stará funkcia createHandle(), ktorá očakáva argument, dátového typu cublasHandle_t, ktorý bude obsahovať vytvorenú rukoväť.

- Prideliť potrebnú pamäť vektorov a/alebo matíc v pamäťovom priestore grafickej karty. funkcia cudaMalloc(), z Runtime API, očakáva ako prvý argument ukazateľ, ktorý bude obsahovať pridelenú pamäť a druhý argument, dátového typu size_t, ktorý bude obsahovať veľkosť pamäti pre pridelenie. Funkcia gpuAllocMatrix() sa stará o vykonanie cudaMalloc() funkcie pre vytvorenie jednej matice. Ukážka funkcie gpuAllocMatrix() sa nachádza vo výpise 4.8. Pri zlyhaní funkcií cudaMalloc je vypísaná na štandardný chybový výstup chybová hláška s názvom premennej, pre ktorú mala byť pamäť pridelená. Ďalej súbor obsahuje funkcie gpuAllocB() a gpuAllocF(), ktoré fungujú rovnako ako funkcia gpuAllocMatrix(), avšak pre pridelenie pamäti vektoru s dátovým typom boolean a vektoru s dátovým typom float. Pre

- Nastaviť hodnoty vektorov a/alebo matíc pre výpočet. Funkcia gpuCpyToDevice() sa stará o zavolanie funkcie cudaMemcpy(), z Runtime API, ukážka funkcie sa nachádza vo výpise 4.8. Funkcia gpuCpyToDevice() očakáva prvý argument, dátového typu integer, obsahujúci veľkosť pamäti, ktorá má byť prekopírovaná. Druhý argument, dátového typu dereferencia ukazateľa float, obsahuje pridelenú pamäť v grafickej karte. Tretí argument, dátového typu dereferencia ukazateľa float, obsahuje hodnoty, ktoré majú byť byť skopírované do pamäti grafickej karty a posledný štvrtý argument, dátového typu string z knižnice string, obsahuje názov premennej pre chybovú hlášku.

```
int gpuAllocMatrix(int N, float*& gpuA, std::string message){
  cudaError_t cudaStatus = cudaMalloc((void**)&gpuA, N * N * sizeof(float));
  if (cudaStatus != cudaSuccess) {
    std::cerr << "Matrix " << message << " alloc on GPU failed.\n";
    cudaDeviceReset();
    return -1;
  }
  return 0;
}

int gpuCpyToDevice(int size, float*& gpuA, float*& A, std::string message){
  cudaError_t cudaStatus= cudaMemcpy(gpuA, A, size, cudaMemcpyHostToDevice);
  if (cudaStatus != cudaSuccess){
    std::cerr << "Copy to GPU of " << message << " failed.\n";
    cudaDeviceReset();
    return -1;
  }
  return 0;
}
```

Výpis 4.8: Vytvorenie a skopírovanie pamäti

Z vyššie spomenutých funkcií som vytvoril funkcie, ktoré pripravia príslušnú grafickú kartu pre výpočet jednotlivých operácií. Jednotlivé funkcie sú vysvetlené v nasledujúcej sekcií.
Pre nastavenie pamäti matíc obsahuje knižnica cuBLAS funkciu cublasSetMatrix(). Pre nastavenie pamäti vektorov obsahuje funkciu cublasSetVector(). Funkcie sú optimalizované pre najrýchlepšie kopírovanie pamäti. Moje funkcie som vytvoril pre lepšie pochopenie procesu kopírovania pamäti do pamäťového priestoru grafickej karty. Taktiež služia ako ukážka ako jednotlivé knižnice pre prácu s grafickými kartami využívajú Runtime API, ku komunikácií s grafickou kartou. Po ukončení výpočtu je potrebné presunúť výsledok operácie z pamäťového priestoru grafickej karty do pamäťového priestoru procesoru, aby bolo možné pristúpiť k výsledku za pomoci procesora. V knižnici cuBLAS sa pre tento účel nachádzajú funkcie cublasGetMatrix() pre maticu a cublasGetVector() pre vektor. Moja implementácia sa nachádza vo funkcií copyResult(), ktorá zavolá funkciu cudaMemcpy() s argumentom druhu prenosu - cudaMemcpyDeviceToHost, ktorý určí smer prenosu z grafickej karty na procesor. Argumenty očakáva rovnaké ako pri funkcií gpuCpyToDevice() s tým rozdielom, že na druhej pozícií sa nachádza ukazateľ do pamäti procesoru a na tretej pozícií ukazateľ do pamäti grafickej karty z kadiaľ majú byť dáta skopírované. Pre vyčistenie kontextu pre grafickú kartu existuje funkcia z knižnice cuBLAS cublasDestroy(), ktorej je potrebné

pridať vytvorenú rukoväť. Taktiež pre tento účel som použil funkciu cudaDeviceReset() z Runtime API, ktorá resetuje vytvorený kontext. Pre ukončenie výpočtu som vytvoril funkciu exitGPU, ktorá zavolá synchronizáciu, ktorá čaká kým sa dokončia všetky predom vyžiadané úlohy za pomoci funkcie cudaDeviceSynchronize(), z Runtime API. O vykonanie funkcie sa stará mnou vytvorená funkcia synchronize(), ak jedna bežiaca úloha zlyhá tak je na štandardný chybový výstup vypísaná chybová hláška. Nakoniec funkcia exitGPU resetuje vytvorený kontext na grafickej karte. Pre ukončenie výpočtu s výsledkom som vytvoril funkciu exitGPUResult(), ktorá očakáva štyri argumenty, ktoré sú predané funkcií copyResult(). Argumenty očakáva v rovnakom poradí ako funkcia copyResult() a rozdiel medzi kopírovaním vektoru a matice závisí od zadanej veľkosti, ktorá má byť skopírovaná. Pred skopírovaním pamäti zavolá funkciu synchronize() a po úspešnom dokončení kopírovania zavolá funkciu cudaDeviceReset().

4.6 Sada testovacích problémov pre CUDA aplikáciu

V príslušných metódach Calculate, v súbore cutest.cpp, sú volané jednotlivé funkcie výpočtu operácie z knižnice cuBLAS. Dokumentácia ku cuBLAS funkciám sa nachádza v dokumente[12]. Metóda CalculateJacobi volá mnou vytvorenú funkciu cuJacobi, ktorá sa nachádza v súbore jacobi.cu. Súbor musí byť prekladaný samostatne a preto je v súbore cutest.cpp funkcia cuJacobi importovaná ako externý prototyp. Za použitia tohoto prístupu nie je potrebné vyrábať hlavičkový súbor. Výpočtové operácie sú rovnaké ako pri sade testovacích problémov pre OpenMP aplikáciu.

```
extern int cuJacobi(int N, float*& gpuA, float*& gpuD, float*& gpuXo, \
float*& gpuX, int iterations, float precision, bool*& prec, \
bool*& converg, bool*& gpuOUT);
```

Výpis 4.9: Externý prototyp

4.6.1 Maticové násobenie

Pre maticové násobenie som použil funkciu cublasSgemm(). Písmenko S v názve funkcie určuje, že funkcia očakáva argumenty s dátovým typom float. Funkcia gemm je vytvorená pre viacero dátových typov. Funkcia cublasSgemm počíta maticové násobenie podľa nasledujúceho výrazu $C = \alpha op(A) op(B) + \beta C$. Ukážka spúšťania funkcie sa nachádza vo výpise 4.10. Funkciu som nastavil tak aby pracovala podla výrazu $C = A * B$. O nastavenie rukoväti, pridelenie a nastavenie matíc sa stará funkcia allocMatrixes(), ukážka implemntácie tejto funkcie sa nachádza vo výpise 4.8. Ak nastane chyba pri vykonávaní funkcie, funkcia CalculateMatrixMul sa ukončí. Kedže v tomto bode sú pripravené všetky potrebné premenné začína stopovanie času a viacnásobné volanie funkcie. Po ukončení výpočtu je zavolaná funkcia exitGPUResult(), ktorá zosynchronizuje čas aplikácie, skopísuje výsledok do pamäťového pristoru procesoru a zresetuje kontext. Nasleduje zastavenie stopovania času a na štandardný výstup je vypísaný strávený čas výpočtom a počet dosiahnutých FLOPS.

```
cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, N, N, &alpha, gpuA, N, \
gpuB, N, &beta, gpuC, N);
```

Výpis 4.10: Volanie funkcie cublasSgemm

4.6.2 Skalárny súčin

Pre skalárny súčiň som použil funkciu cublasSdot(). Funkcia cublas<t>dot môže byť zavolaná pre dátové typy float, double, cuComplex a cuDoubleComplex. Funkcica vykonáva výpočet podľa výrazu $result = \sum_{i=0}^{N}(x[k] * y[j])$ kde k je definované ako $k = 1+(i-1)*incx$ a j definované ako $j = 1+(i-1)*incy$. Výpočet hodnôt k a j je vytvorený pre kompatibilitu s Fortran indexovaním od čísla jeden. Ukážka volania funkcie sa nachádza vo výpise 4.11. O vytvorenie potrebných premenných sa stará funkcia CalculateScalarMul. O nastavenie rukoväti a vektorov v pamäti grafickej karty sa stará funkcia allocScalarMul() zo súboru cublasOperations. Po ukončení výpočtu je zavolaná funkcia exitGPU(), ktorá zosynchronizuje zariadenia a zresetuje kontex.

```
cublasSdot(handle, N, gpuvA, 1, gpuvB, 1, &result);
```

Výpis 4.11: Volanie funkcie cublasSdot

4.6.3 Súčet matíc a vektorov

Pre sčítanie matíc som využil funkciu cublasSgeam(). Funkcia vykonáva výpočet podľa výrazu $C = \alpha op(A) + \beta op(B)$. Funkcia očakáva rovnaké argumenty ako u funkcií cublasSgemm(), s tým rozdielom, že namiesto troch veľkosti dimenzií počítania sú očakávané len dve veľkosti. Vo funckcií CalculateMatrixMul sú volané funkcie allocMatrixes() a exitGpuResult() zo súboru cublasOperations.cpp.

Pre sčítanie vektorov som použil funkciu cublasSaxpy(). Funkcia vykonáva výpočet podla výrazu $y[j] = \alpha * x[k] + y[j]$ pre veľkosti $i = 1, ..., N; k = 1 + (i-1) * incx$ a $j = 1 + (i-1) * incy$. Funkcia očakáva podobné argumenty ako funkcia cublasSdot(). Pri funkcií cublasSaxpy() neočakáva premennú pre uschovanie výsledku nakoľko sa výsledok nachádza vo vektore y. Funkcia očakáva konštantnú premennú alfa, ktorá môže byť použitá pred sčítaným na vynásobenie vektora s skalárom. V mojom volaní som jej nastavil hodnotu jeden. Vo funkcií CalculateVectorAdd sú volané funkcie allocVectorAdd(), pre vytvorenie rukoväti a skopírovanie vektorov do grafickej karty, a exitGPUResult().

4.6.4 Kopírovanie Pamäti

Vo funkcií CalculateCopyMemory je odstopované zavolanie funkcie cuMemCpy 4.12. Na štandardný výstup je vypísaná priepustnosť a čas strávený vytvorením a skopírovaním matíc do pamäťového priestoru grafickej karty, o čo sa starajú funkcie gpuAllocMatrix() a gpuCpyToDevice(). Implementáciu týchto funkcií je možné vidieť vo výpise 4.8. Po dokončení je zavolaná funkcia exitGPU(), ktorá zosynchronizuje procesor s grafickou kartou a resetuje kontext.

```
int cuMemCpy(int N, float* A, float* B, float* C){
  float* gpuA;
  float* gpuB;
  float* gpuC;

  if(setDevice()) return -1;
  if(gpuAllocMatrix(N, gpuA, "A")) return -1;
  if(gpuAllocMatrix(N, gpuB, "B")) return -1;
  if(gpuAllocMatrix(N, gpuC, "C")) return -1;

  if(gpuCpyToDevice(N * N * sizeof(float), gpuA, A, "matrix A")) return -1;
  if(gpuCpyToDevice(N * N * sizeof(float), gpuB, B, "matrix B")) return -1;
  if(gpuCpyToDevice(N * N * sizeof(float), gpuC, C, "matrix C")) return -1;

  if(exitGPU()) return -1;
  return 0;
}
```

Výpis 4.12: Kopírovanie Pamäti CUDA

4.6.5 Transponovanie matice

Pre transponovanie matice som využil funkciu cublasSgeam, ktorú som už použil pre sčítanie matíc. Premennú beta som nastavil na hodnotu 0 aby nepričítavalo maticu B a premennú alfa som nastavil na hodnotu 1 aby matica A ostala v pôvodnom stave. Druhý argument, ktorý definuje operáciu nad maticou A som nastavil na hodnotu CUBLAS_OP_T, ktorá definuje že matica A bude transponovaná. Výsledok transponovania matice A bude uložený v matici C. Po ukončení výpočtu je zavolaná funkcia exitGPUResult a na štandardný výstup je vypísaný strávený čas výpočtom a priepustnosť. Keďže funkcia vykonáva veľa zbytočných krokov predpokladám že transponovanie matice za použitia knižnice OpenMP bude rýchlejšie.

4.6.6 Jacobiho iteračná metóda

Výpočet Jacobiho iteračnej metódy je rozdelený na tri kerneli. Prvý kernel s názvom Convergence má za úlohu overiť konvergenciu matice. Kernel pracuje len s jednom blokom vlákien, pretože sa mi nepodarilo vytvoriť správnu implementáciu pre viacero blokov. V kerneli je použité globálne pole s názvom reduc, dátového typu boolean a vytvorené za použitia kľúčového slova ___device___. Do poľa reduc sú uložené hodnoty porovnania. Porovnaná je hodnota na diagonále so súčtom absolútnych hodnôt nediagonálnych prvkov pre všetky riadky. Za dokončením porovnania hodnôt sa nachádza funkcia ___syncthreads(), ktorá podrží vlákna v tomto bode aby vlákna nepokračovali kým ešte nejaké vlákno počíta porovnanie. Na konci výpočtu je pole zredukované na jednu hodnotu, ktorá určuje či je matica ostro riadkovo diagonálne dominantná a je vrátená späť do pamäti procesoru. A.1

Druhý kernel s názvom jacobi vykonáva výpočet nových neznámych v danej iterácií. Kernel pracuje s 672 blokmi o veľkosti 256 vlákien. Tento počet vlákien využije všetky streamovacie multiprocesory, ktoré sú dostupné na grafickej karte. Výsledok je uložený do poľa X kde jedno vlákno počíta jednu hodnotu z poľa X. A.2

Posledný kernel Finish počíta či už bola dosiahnutá požadovaná odchýlka výpočtu. Kernel pracuje s viacerými blokmi vlákien, ktorých veľkosť je rovnaká ako pri kerneli jacobi. Jedno vlákno vypočíta rozdiel hodnoty z predchádzajúcej iterácie a hodnoty z aktuálnej iterácie. Absolútnu hodnotu tohoto rozdielu porovná so zadanou presnosťou a výsledok ukladá do globálnej premennej reduc, rovnako ako pri kerneli Convergence. Použitá funkcia fabs() pre absolútnu hodnotu sa nachádza v knižnici math. Za výpočtom odchýlky sú vlákna zosynchronizované. Ďalej už nasleduje redukcia pola reduc a výsledok redukcie je stiahnutý z grafickej karty. A.3

O volanie kernelov s potrebným počtom vlákien a blokov sa stará funkcia cuJacobi. Funkcia na začiatku zavolá kernel convergence a ak je výsledok false vypíše hlášku, Konvergencia nezaručená, na štandardný výstup. Nasleduje sekvenčné volanie iterácií pre kerneli jacobi a finish. Počet iterácií je definovaný v argumente iterations. Ukončenie výpočtu môže nastať ak kernel finish vráti hodnotu true alebo počet vykonaných iterácií dosiahne maximálnu hodnotu. Funkcia cuJacobi pri ukončení vráti počet vykonaných iterácií a výsledné neznáme sa nachádzajú v poli X. A.4

4.7 Kompilátor pre CUDA aplikáciu

Pre preklad CUDA aplikácie som použil kompilátor NVCC z CUDA sady nástrojov. Kompilátor dokáže spracovať C++ kód s definovanými funkciami pre grafickú kartu. Kompilátor nedokáže samostatne spracovať C++ kód a preto je ešte potrebný kompilátor GCC a G++ pre Linux.[14] Môj program som prekladal v príkazovej riadke pomocou nasledovného príkazu 4.13. Prepínač -lcublas určuje, že v kompilovanom programe je použitá knižnica cuBLAS. Názov súboru jacobi.cu obsahuje špeciálnu príponu .cu, ktorá špecifikuje CUDA C++ súbor. V CUDA súbore je možné importovať a použiť základné knižnice pre C/C++ Kompilátoru musí byť samostatne predaný súbor cutest.cpp, pretože súbor nestačí importovať v súbore cutest.cpp. Z predávaného súboru je potrebné importovať funkciu pre hostiteľské zariadenie za použitia externého prototypu v súbore cutest.cpp. Pri kompilovaní bude kompilátor presne vedieť kde danú funkciu nájsť.

```
nvcc cutest.cpp jacobi.cu -o cutest -lcublas
```

Výpis 4.13: Preklad CUDA aplikácie

Kapitola 5

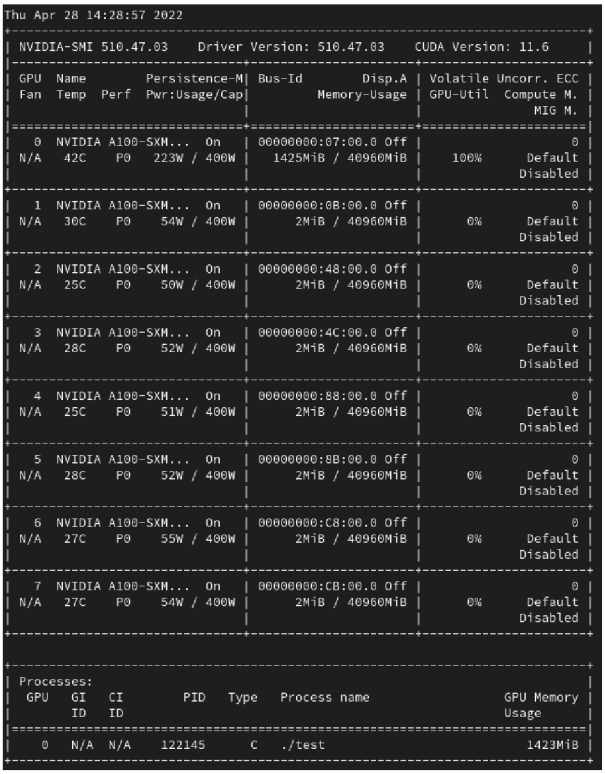
Dosiahnuté výsledky

5.1 Testovanie

Triedy CuTest a Test obsahujú privátne metódy Check pre otestovanie správne vypočítaného výsledku operácie. O volanie týchto metód sa starajú metódy Calculate pre príslušné operácie. Jacobiho iteračnú metódu testuje metóda JacobiCheck, ktorá porovnáva výsledné hodnoty z poľa X vypočítané paralelne grafickou kartou s hodnotami z poľa X vypočítané jedným vláknom na procesore. Keďže sa hodnoty polí nerovnajú metóda vypíše na štandardný výstup počet iterácií vypočítaných grafickou kartou, počet iterácií vypočítaných procesorom a odchýlku nerovnajúcich sa hodnôt. Pred zavolaním funkcií pre výpočet Jacobiho metódy je zavolaná metóda prepareJacobi pre naplnenie matice testovacími dátami, tak aby bola matica ostro riadkovo diagonálne dominantná a výsledok konvergoval. Pri zavolaní konštruktoru triedy sú matice a vektory naplnené testovacími dátami pre otestovanie očakávaného výsledku operácií v metódach Check. Metódy pre výpočet matíc porovnajú každú hodnotu, ktorá sa nachádza v matici, pre operácie sčítania a násobenia matíc. Metóda pre skalárný súčin porovná jednu výslednú hodnotu a metóda pre sčítanie vektorov porovná hodnoty v poli. Funkcia pre transponovanie matice porovná výslednú maticu s pôvodnou, kde všetky hodnoty musia byť na správnom mieste transponované. Ak výsledky operácií nie sú správne, pred ukončením príslušnej funkcie Calculate je vypísaná na štandardný výstup hláška „ZLE“.

5.1.1 NVIDIA System Management Interface Program

Nvidia-smi je nástroj pre príkazový riadok, ktorá je postavená nad knižnicou NVIDIA Management (NVML). Slúži k spravovaniu a monitorovaniu NVIDIA grafických kariet, ktoré sú založené minimálne na Fermi architektúre [11]. Zavolanie príkazu nvidia-smi vypíše na štandardný výstup aktuálne používané verzie nvidia-smi, ovládačov, CUDA knižnice a všetky dostupné NVIDIA grafické karty v systéme a ku každej z nich jednotlivé merané metriky a aktuálne bežiace procesy. Pomocou tohto nástroju som sledoval správnosť vykonávania mojej implementácie. Spomenuté ovplyvnenie výpočtu teplotou som sa snažil odstrániť, tým že som ukladal výsledky až keď bola grafická karta zohriatá. Pomocou tohto nástroju som sledoval aktuálnu teplotu, ktorú som dvíhal spúšťaním zahrievacích testov. Pomocou výpisu aktuálnej veľkosti zabranej pamäti som sledoval pri jednotlivých funkciách veľkosť pamäti, ktorú si zaberajú. Ako je možné vidieť na obrázku 5.1, na grafickej karte s číslom 0 prebieha výpočet programu test. Program má zabraných skoro 1,5 GB pamäti. Používaná grafická karta má zvýšenú teplotu 42 °C a jej aktuálny výkon bol 223 wattov.



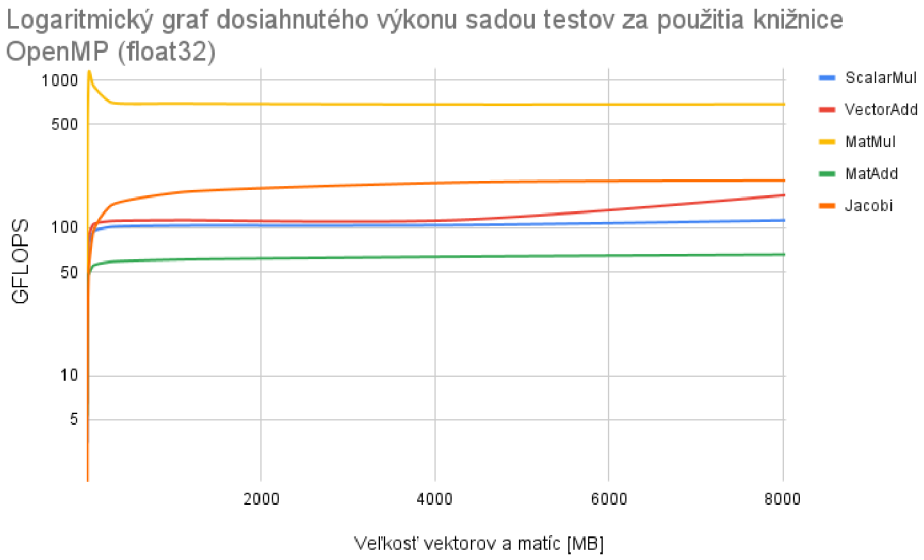
Obr. 5.1: Ukážka nástroju nvidia-smi

5.1.2 NVIDIA Nsight

Pre profilovanie implementácie Jacobiho metódy za použitia Runtime API som použil profilovač kódu Nvidia Nsights. Profilovač som skúšal aj pre implementácie za použitia knižnice OpenMP. Pri výsledku profilovania nebolo možné rozoznať jednotlivé názvy volaných kernelov. Knižnica OpenMP volala jednotný názov kernelu pre všetky vytvorené funkcie. Profilovač vypíše štatistiku pre každý zavolaný kernel. Štatistika sa skladá z troch častí. GPU priepustnosť, štartovacie štatistiky a obsadenosť. Pri každej z týchto štatistík je vypísané upozornenie ako je možné danú štatistiku vylepšiť ak výpočet dosahuje nízke hodnoty alebo je vypísané oznámenie, že funkcia je dobre vyvážená. Pre kernel jacobi sa mi podarilo vytvoriť 25,1% obsadenosť, štartovacie štatistiky sú v poriadku a priepustnosť ukazuje nízke hodnoty. Pre kernel finish sa mi podarilo dosiahnuť 64,5% obsadenosť, štartovacie štatistiky sú v poriadku a priepustnosť je nízka. Pre kernel convergence sa mi podarilo dosiahnuť 12.5% obsadenosť, štartovacie štatistiky odporúčajú pustiť výpočet na viacerých blokoch a priepustnosť je skoro žiadna. Kernel convergence som spúšťal aj s viacerými blokmi ale výpočet sa mi nepodarilo naimplementovať správne. Kernel jacobi a converge som sa pokúsil vytvoriť aj pre výpočet na dvojdimenzionálnych blokoch. Výpočet spočíva v rozdelení všetkých iterácií slučiek medzi jednotlivé vlákna, kde jedno vlákno bude počítať jednu iteráciu. Nanešťastie sa mi nepodarilo vytvoriť implementáciu, ktorá počítala správne.

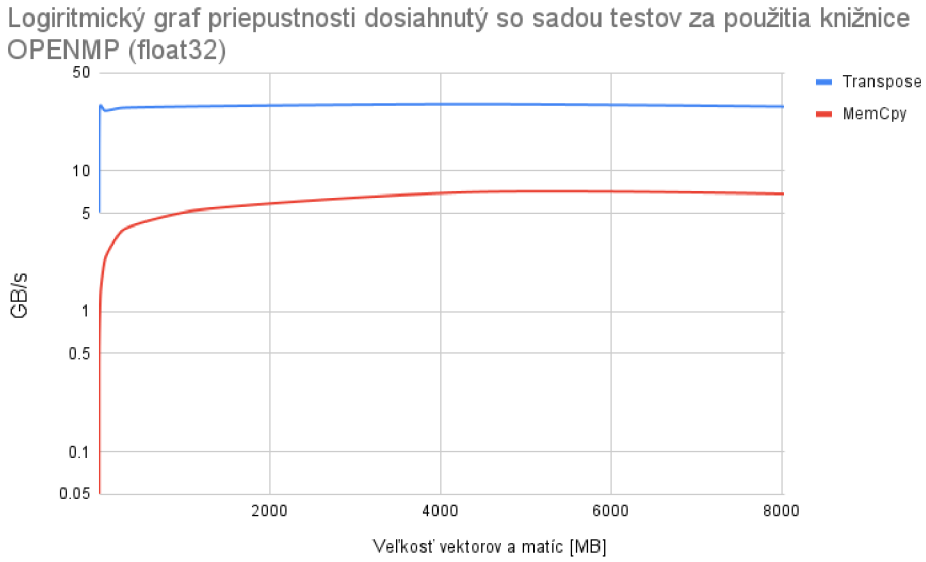
5.2 Grafická Karta NVIDIA A100 40 GB (Karolina)

Výpočty pre grafickú kartu NVIDIA A100 boli uskutočnené na superpočítači Karolina, ktorý poskytuje skupina IT4Innovations z Technickej Univerzity v Ostrave. Na superpočítači sa nachádza 72 výpočtových uzlov, ktoré pozostávajú z 8 grafických kariet NVIDIA A100 40GB. Výpočty boli uskutočnené len na jednej grafickej karte. Moja implementácia pracuje s dátovým typom float o veľkosti 32 bitov a jedna grafická karta NVIDIA A100 40GB môže dosiahnuť výkon až 19.5 TFLOPS pri použití dátového typu float a priepustnosť pamäti až 1555 GB/s. [17] Server obsahuje nainštalovanú najnovšiu verziu CUDA sady nástrojov, z ktorej som využil kompilátor NVC++ pre preklad OpenMP aplikácie. Najvyššie namerané rýchlosti jednotlivých operácií za použitia knižnice OpenMP sa nachádzajú v grafoch 5.2 a 5.3.



Obr. 5.2: Graf výkonosti za použitia knižnice OpenMP

Grafy vyplývajú zo spúšťania mojej implementácie, ktoré bolo vykonané pre deväť veľkostí dimenzií. Prvá veľkosť dimenzie začína na hodnote 2^8 a nasledujúce hodnoty sa zvyšujú po násobku dvojky až po hodnotu 2^{15}, ktorá tvorí ôsmu veľkosť dimenzie. Posledná veľkosť dimenzie je 44800. Od najmenších veľkostí boli funkcie viacnásobné spúšťané 100, 80, 60, 40, 30, 20, 15, 10 a 5 ráz za sebou. Veľkosti dimenzií predstavujú v pamäťovom priestore veľkosti pamäti pre vektory a matice o hodnotách 262KB, 1MB, 4MB, 16MB, 67MB, 268MB, 1GB, 4GB a 8GB. Najvyšší počet operácií dosiahla operácia maticového násobenia, čo je možné vidieť pri grafe dosiahnutých GFLOPS, ktorá dosiahla najvyšší výpočtový výkon 1.1TFLOPS. 1 TFLOPS je 5.5% celkového výkonu grafickej karty. Pri prvotnej implementácií maticového násobenia som sa snažil použiť cache blocking pre lepšiu optimalizáciu. Cache bloking spočíva v načítaní dostatočného počtu hodnôt do L1 cache pamäti (Jeden SM má prístupných 192 KB L1 cache a 40MB L2 cache), aby sa výpočet vyhol mnohonásobnému presunu pamäti, ktorý zaberie mnoho času. Nanešťastie sa mi za použitia cache blocking-u nepodarilo funkciu viac optimalizovať. Funkcia pre vytvorenie a kopírovanie pamäti na grafickú kartu dosiahla najvyššiu priepustnosť 7 GB/s pri veľkosti

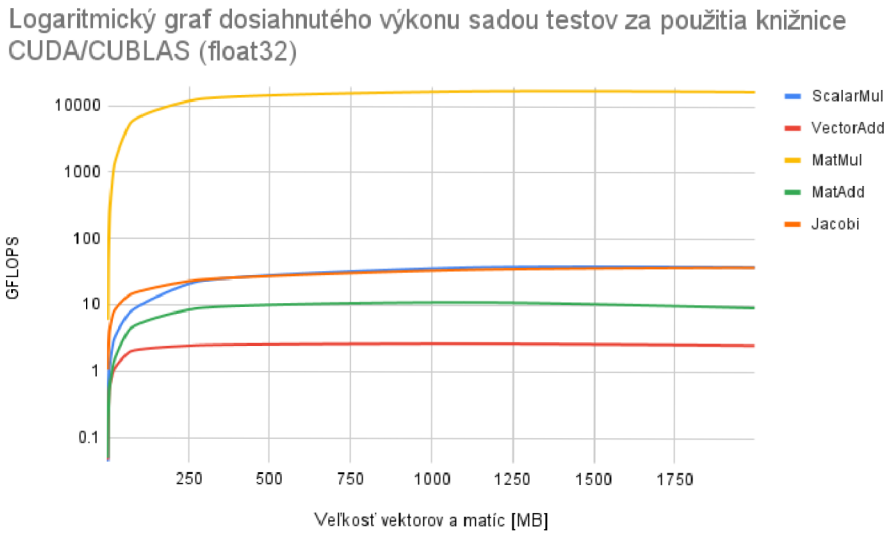


Obr. 5.3: Graf priespustnosti za použitia knižnice OpenMP

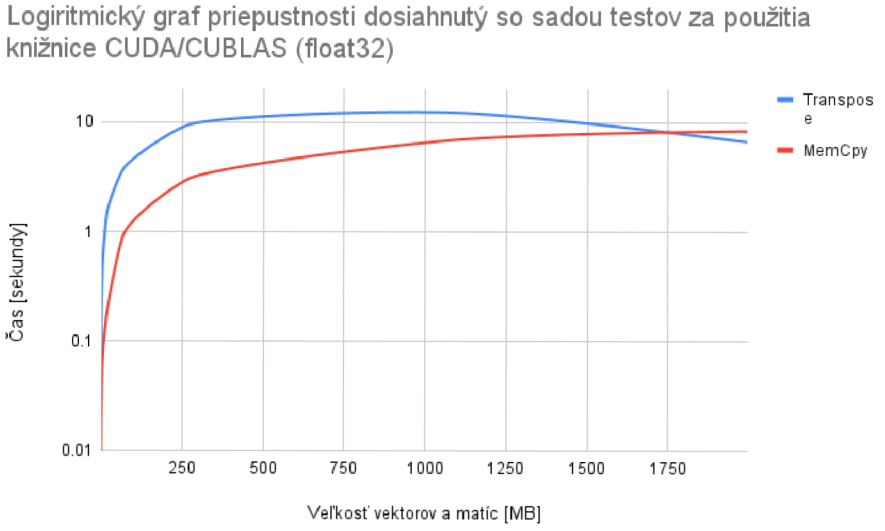
matíc o hodnote 4GB. Maximálna priepustnosť zbernice je 64 GB/s. Dosiahnutá priepustnosť OpenMP kopírovaním dosiahla 11% celkového výkonu zbernice. Pri transponovaní matice bola dosiahnutá najvyššia priepustnosť pamäti 30 GB/s. 30 GB/s tvorí necelé 2% z celkovej priepustnosti vnútornej pamäti, ktorá je 1555 GB/s. Implementácia Jacobiho metódy sa výkonnostne umiestnila na 2 mieste. Pre najväčšiu veľkosť 8GB dosiahla 209 GFLOPS čo je skoro 1.1% celkového výkonu. Operácia vektorového sčítania dosiahla 2 až 3krát väčší výkon ako maticové sčítanie. Vyskúšal som aj rýchlosť použitia lineárnej pamäti, ktorá dosiahla o 2 GFLOPS menší výkon ako dvojdimenzionálna pamäť. Pri implementácií som optimalizoval dosahujúce rýchlosti za použitia viacerých konštrukcií. Miesto konštrukcie „#pragma omp target teams distribute parallel collapse(2) for“ som použil konštrukciu „#pragma omp target teams loop collapse(2)“, ktorá mierne zrýchlila výpočet veľkostí do hodnoty 67MB a mierne spomalila výpočet väčších veľkostí. Paralelizovanie prvých dvoch slučiek, pri operáciach s maticami, za použitia klauzule „collapse(2)“ zrýchlilo výpočet mnohonásobne.

Pre porovnanie dosiahnutých výsledkov som vytvoril CUDA aplikáciu, ktorá volá funkcie pre výpočet z knižnice cuBLAS a Runtime API. Pre preklad CUDA aplikácie som využil kompilátor NVCC z nainštalovanej CUDA sady nástrojov. V grafoch 5.4, 5.5 sa nachádzajú dosiahnuté rýchlosti.

Sada testov pre tieto grafy bola spúšťaná pre veľkosti pamäti o hodnotách 262 KB, 1MB, 4MB, 16MB, 67MB, 1GB a 2 GB. Strop veľkosti 2 GB je daný maximálnou veľkosťou, ktorú dokáže cudaMalloc() vytvoriť. Vytváranie pamäti o väčšej veľkosti už nebolo možné alebo výsledky operácií neboli správne. Operácie boli spúšťané s počtom opakovaní od najmenšej veľkosti o hodnotách 100, 80, 60, 40, 30, 20, 10 a 5 ráz za sebou. Najvyšší výkon dosiahla operácia násobenia matíc. Pri dvoch posledných veľkostiach dosiahla až 16.8 TFLOPS čo je 86% výkonu grafickej karty. Oproti operácií maticového násobenia za použitia knižnice OpenMP je tento výkon 8krát väčší. Jacobiho metóda dosiahla skoro rovnaký výkon ako pri skalárnom súčine. Pri veľkosti 8GB dosiahli 38 GFLOPS čo je 5 násobné spomalenie



Obr. 5.4: Graf výkonnosti za použitia knižnice CUDA/CUBLAS

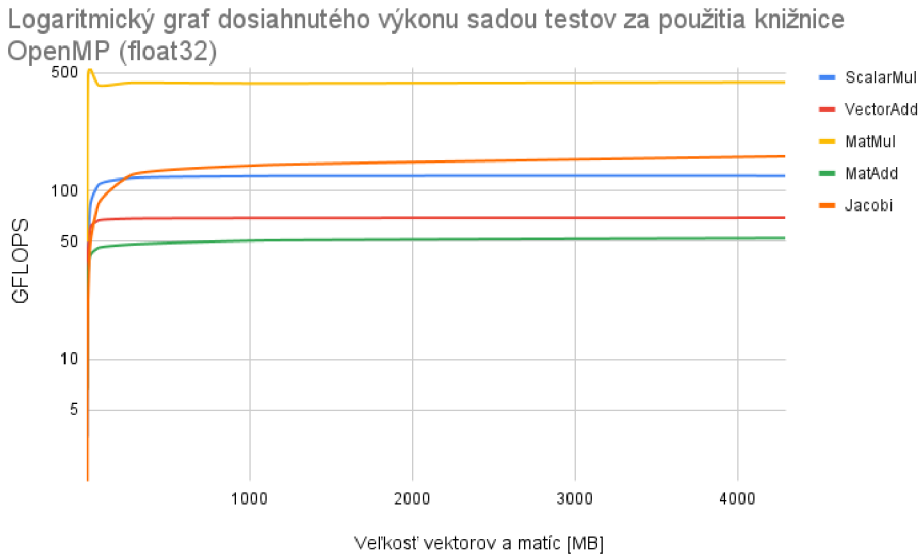


Obr. 5.5: Graf priepustnosti za použitia knižnice CUDA/CUBLAS

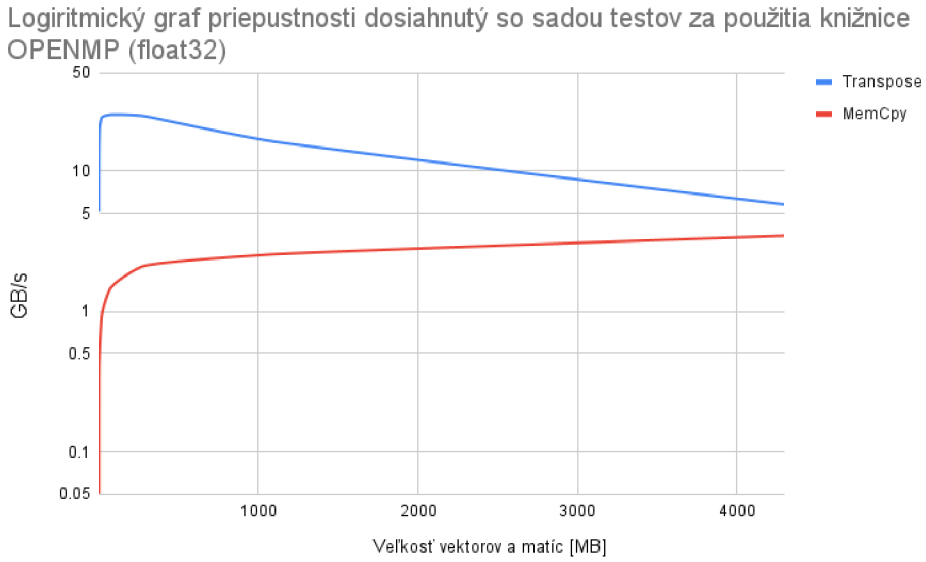
oproti knižnici OpenMP. Operácia vektorového sčítania dosiahla najnižší výkon a to len 2.5 GFLOPS. Operácia kopírovania matíc bola pomalšia pri menších veľkostiach ale pri najväčšej veľkosti 8GB dosiahla priepustnosť 8 GB/s čo je 12.5% z celkovej priepustnosti. Pri operácií transponovania matice bola dosiahnutá priepustnosť 3krát menšia ako pri OpenMP.

5.3 Grafická Karta Tesla V100-SXM2-16GB (Barbora)

Výpočty na grafickej karte Nvidia V100 boli vykonané na superpočítači Barbora, ktorý poskytuje taktiež skupina IT4Innovations. Superpočítač Barbora obsahuje 8 výpočtových uzlov, ktoré pozostávajú zo štyroch grafických kariet Nvidia V100 16GB. Grafická karta dokáže dosiahnúť teoretický výkon až 15.7TFLOPS pri práci s dátovým typom float o veľkosti 32 bitov. Priepustnosť vnútornej pamäti je až 900GB/s. V grafickej karte je použitá zbernica PCIe, ktorá dosahuje priepustnosť 32GB/s [19]. Nameraný výkon sady testov sa nachácha v grafe 5.6 a nameraná priepustnosť sady testov sa nachádza v grafe 5.7.



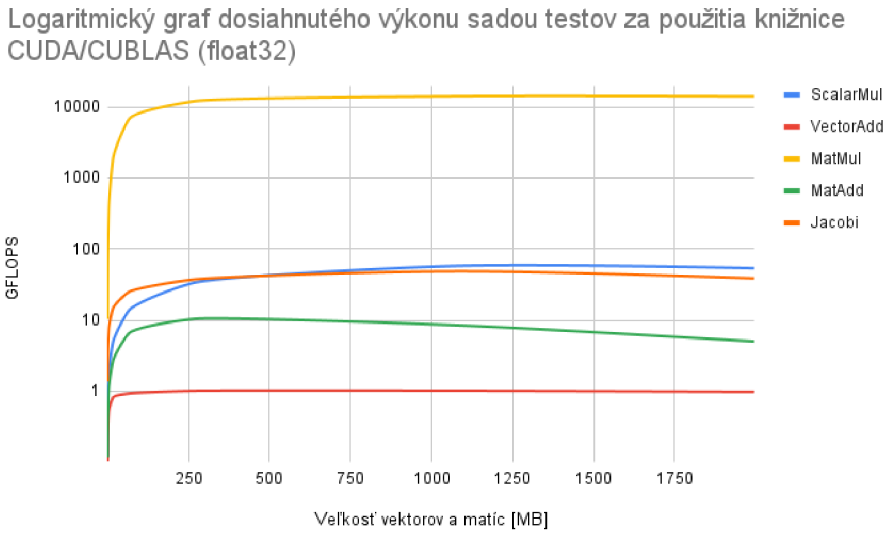
Obr. 5.6: Graf výkonnosti za použitia knižnice OpenMP



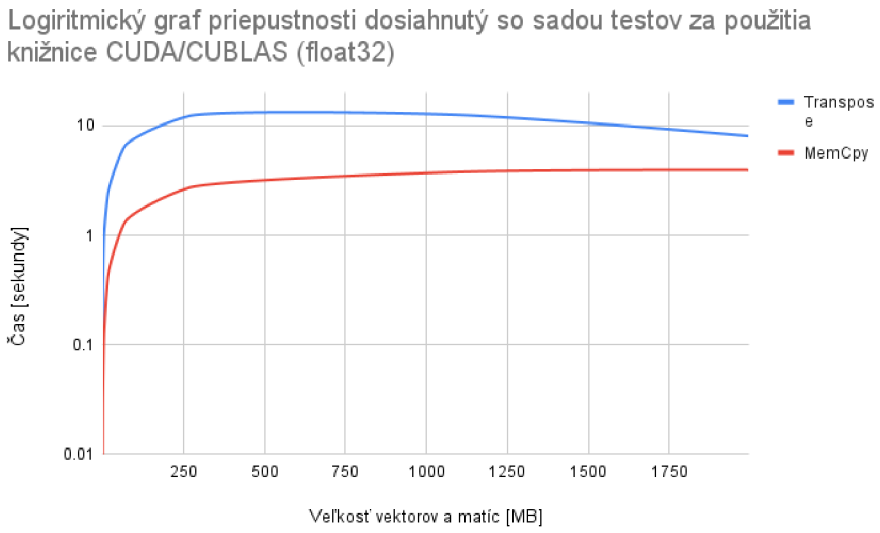
Obr. 5.7: Graf priepustnosti za použitia knižnice OpenMP

Keďže veľkosť pamäti grafickej karty je 16 GB sada testov bola spúšťaná s ôsmymi veľkosťami. Veľkosti v pamäti sú o hodnotách 262KB, 1MB, 4MB, 16MB, 67MB, 268MB, 1GB a 4GB. Veľkosti opakovaní boli rovnaké ako pri grafickej karte Nvidia A100. Najvyšší výkon dosiahla operácia maticového násobenie a to 550 GFLOPS čo je 3.5 % z celkového výkonu grafickej karty. Pri ostatných výkonných operáciach, okrem skalárneho súčinu, je vidieť menší pokles výkonu oproti sade testov vykonanej na grafickej karte A100. Keďže je výkon grafickej karty o 20% menší tieto operácie dosiahli približne o 20% menší výkon. Skalárny súčin dosiahol 122 GFLOPS čo je o skoro 15% väčší výkon ako pri výpočte na grafickej karte A100. Čo sa týka priepustnosti pamäti pre kopírovanie klesla o viac ako 50% ako bolo očakávané keďže priepustnosť pamäti je o 50% menšia ako pri karte A100. Priepustnosť transponovania matice pre malé veľkosti bola o kúsok pomalšia a pre veľké matice značne klesla oproti priepustnosti ako bolo očakávané keďže priepustnosť vnútornej pamäti je o viac ako 40% nižšia.

Pre spúšťanie sady testov CUDA aplikácie bolo použitých osem veľkostí rovnako ako pri spúšťaní na Karolíne. Počet opakovaní bol taktiež rovnaký. Dosiahnuté rýchlosti sa nachádzajú v grafoch 5.8 a 5.9. Operácia maticového násobenia dosiahla najvyšší výkon 14.1 TFLOPS čo je skoro 90% celkového výkonu karty. Pre skalárny súčin a Jacobiho metódu výkon jemne vzrástol oproti výpočte na Karolíne. Pre vektorový a maticový súčet výkon značne klesol oproti výpočte na Karolíne. Priepustnosť pamäti kopírovania taktiež klesla o viac ako 50%. Priepustnosť transponovania matice bola jemne zvýšená oproti výpočte na Karolíne.



Obr. 5.8: Graf priepustnosti za použitia knižnice OpenMP



Obr. 5.9: Graf priepustnosti za použitia knižnice OpenMP

Kapitola 6

Záver

Cieľom práce bolo vytvoriť sadu testovacích problémov, ktoré budú testovať výkonnosť grafickej karty, na ktorej budú spúšťané. Na začiatku práce som spravil rešerš v oblasti knižníc pre prácu na grafickej karte. Na základe rešerše bolo rozhodnuté, ktoré knižnice budú použité pre vytvorenie sady testov. Ako ďalšie bolo potrebné vybrať grafické karty pre otestovanie. Boli mi sprístupnené dva superpočítače od skupiny IT4Innovations. Prvý superpočítač Karolina pozostával z uzlov s grafickými kartami Nvidia A100 a druhý superpočítač pozostával z uzlov s grafickými kartami Nvidia V100. Použil som jednu grafickú kartu Nvidia A100 a jednu grafickú kartu Nvidia V100, ktoré sa stali vedľajšou témou tejto práce.

Práca obsahuje podrobné vysvetlenie vytvorenej sady testov a knižníc, ktoré boli použité pri implementácií. Kľúčová vlastnosť testov spočíva v dosiahnutí čo najvyššieho dostupného výkonu grafickej karty. Pre dosiahnutie tejto vlastnosti boli jednotlivé testy optimalizované za pomoci viacerých techník. Hlavná použitá technika spočíva vo využití čo najväčšieho paralelizovania výpočtu. Medzi ďalšie použité techniky optimalizácie patrí využitie registrov procesných blokov, využitie L1 cache pamäti SM a vyhýbanie sa používaniu redukcií pri viacnásobnom prístupe vlákien ku jednej premennej. Niektoré techniky neboli využité pre všetky testy. Každý test vyžaduje vyriešenie špecifických problémov, ktoré nie sú vždy rovnaké. Preto je potrebné optimalizovať testy po jednom. Pochopenie komplexného problému a následné riešenie dokáže zabrať mnoho času a stále ostane priestor pre lepšiu optimalizáciu.

V praxi je možné pracú využiť pre porovnávanie dosiahnutých rýchlostí grafickými kartami, inými slovami benchmark. Časť testov má za úlohu otestovať hrubý výkon grafických kariet, ktorý je meraný v jednotkách FLOPS. Druhá časť testov testuje priepustnosť vnútornej pamäti a priepustnosť zbernice grafických kariet, ktoré sú merané v jednotkách GB za sekundu.

Práca má stále priestor pre optimalizáciu sady testov. Veľkosť optimalizácie sady testov, ktorú je možné dosiahnuť je možné vidieť na výsledkoch použitých operácií z knižnice cuBLAS alebo z teoretických rýchlostí grafických kariet. Po dosiahnutí týchto limitov by bolo možné rozšíriť sadu testov o test rýchlosti komunikácie viacerých grafických kariet medzi sebou, NVLink a o nové zložitejšie testy hrubého výkonu.

Práca mi dala nové poznatky o štruktúre grafickej karty a jej využitiu. Naučil som sa lepšie pracovať so superpočítačmi a pochopil som potrebné kroky pre vývoj aplikácie, ktorá pracuje s grafickou kartou.

Literatúra

[1] Abi Chahla, F. Nvidia's CUDA: The End of the CPU? *Tom's Hardware*. 2008, s. 15. Dostupné z: https://www.tomshardware.com/reviews/nvidia-cuda-gpu,1954.html.

[2] affiliates, N. C. . *Thrust Quick Start Guide* [online]. 2788 San Tomas Expy, Santa Clara, CA 95051, USA: NVIDIA Corporation & affiliates, marec 2022 [cit. 2022-05-08]. Dostupné z: https://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf.

[3] Alarcon, N. *NVIDIA Parabricks: Accelerating Genomic Analysis from Days to an Hour*. Nvidia, 2020. Dostupné z: https://developer.nvidia.com/blog/advancing-dna-sequencing/.

[4] Baxter, S. *Introduction*. Nvidia, 2013. Dostupné z: https://moderngpu.github.io/intro.html.

[5] Board, O. A. R. *OpenMP 4.5 API C/C++ Syntax Reference Guide* [online]. Kanada, Beaverton: OpenMP Architecture Review Board, november 2015 [cit. 2022-01-19]. Dostupné z: https://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf.

[6] Board, O. A. R. *OpenMP Application Programming Interface Examples* [online]. Kanada, Beaverton: OpenMP Architecture Review Board, jún 2020 [cit. 2022-01-20]. Dostupné z: https://www.openmp.org/wp-content/uploads/openmp-examples-5-0-1.pdf.

[7] Drifter1. *Programming - Parallel Sections in OpenMP: C/Programming & Dev.* 2020. Dostupné z: https://peakd.com/hive-169321/@drifter1/programming-parallel-sections-in-openmp.

[8] Ing. Jaroš Jiří Ph.D. doc. *Programování se sdílenou pamětí Úvod do Openmp a smyčky for* [online]. VUT FIT, 2021 [cit. 2022-01-10]. Dostupné z: https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FACH-IT%2Flectures%2FAVS_07.pdf&cid=14664presentation.

[9] Krashinsky, R., Giroux, O., Jones, S., Stam, N. a Ramaswamy, S. *NVIDIA Ampere Architecture In-Depth*. Nvidia, 2020. Dostupné z: https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/.

[10] NVIDIA. *CUDA TOOLKIT* [online]. [cit. 2022-05-05]. Dostupné z: https://developer.nvidia.com/cuda-toolkit.

[11] NVIDIA. *Nvidia-smi 367.38* [online]. 2788 San Tomas Expy, Santa Clara, CA 95051, USA: NVIDIA Corporation, júl 2016 [cit. 2022-05-08]. Dostupné z: https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf.

[12] NVIDIA. *CuBLAS Library* [online]. 2788 San Tomas Expy, Santa Clara, CA 95051, USA: NVIDIA Corporation & affiliates, marec 2022 [cit. 2022-05-06]. Dostupné z: https://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf.

[13] NVIDIA. *CUDA C++ Programming Guide* [online]. 2788 San Tomas Expy, Santa Clara, CA 95051, USA: NVIDIA Corporation & affiliates, marec 2022 [cit. 2022-05-08]. Dostupné z: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[14] NVIDIA. *CUDA Compiler Driver NVCC* [online]. 2788 San Tomas Expy, Santa Clara, CA 95051, USA: NVIDIA Corporation, marec 2022 [cit. 2022-04-28]. Dostupné z: https://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf.

[15] NVIDIA. *CUDA Runtime API* [online]. 2788 San Tomas Expy, Santa Clara, CA 95051, USA: NVIDIA Corporation, január 2022 [cit. 2022-05-01]. Dostupné z: https://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf.

[16] NVIDIA. *CUDA Toolkit Documentation v11.6.2* [online]. Nvidia, marec 2022 [cit. 2022-05-08]. Dostupné z: https://docs.nvidia.com/cuda/.

[17] NVIDIA. *NVIDIA A100 TENSOR CORE GPU: Unprecedented acceleration at every scale*. Nvidia, 2022. Dostupné z: https://www.nvidia.com/en-us/data-center/a100/.

[18] NVIDIA. *NVIDIA HPC Compilers C++ Parallel Algorithms* [online]. NVIDIA Corporation, marec 2022 [cit. 2022-05-07]. Dostupné z: https://docs.nvidia.com/hpc-sdk/pdf/hpc223c++_par_alg.pdf.

[19] NVIDIA. *NVIDIA V100 TENSOR CORE GPU: The First Tensor Core GPU*. Nvidia, 2022. Dostupné z: https://www.nvidia.com/en-us/data-center/v100/.

[20] Shi, Z. *GPU computing with CUDA*. Ames, IA 50011: Iowa State University of Science and Technology, 2011. Dostupné z: https://wikis.ece.iastate.edu/cpre584/images/3/34/CUDA_Memory.pdf.

[21] Stam, N., Durant, L., Giroux, O. a Harris, M. *Inside Volta: The World's Most Advanced Data Center GPU*. Nvidia, 2017. Dostupné z: https://developer.nvidia.com/blog/inside-volta/.

Prílohy

Príloha A

Jacobi CUDA

```
__global__ void convergence(int N, float* A, bool* converg, bool* out){
  int idx = threadIdx.x;
  float sum = 0.0;

  for (int i = idx; i < N; i += blockDim.x){
    for (int j = 0; j < N; j++){
      if (i != j){
        sum += fabs(A[i * N + j]);
      }
    }
    reduc[i] = fabs(A[i * N + i]) > sum;
    sum = 0.0;
  }
  __syncthreads();
  for (int size = N/2; size>0; size/=2){
    if (idx<size)
      reduc[idx] = reduc[idx] & reduc[idx+size];
    __syncthreads();
  }
  if (idx == 0)
    *out = reduc[0];
}
```

Výpis A.1: KERNEL convergenece

```
__global__ void jacobi(int N, float* A, float* D, float* Xo, float* X){
  int thIdx = threadIdx.x;
  int gthIdx = thIdx + blockIdx.x * blockDim.x;
  const int size = blockDim.x * gridDim.x;

  for (int i = gthIdx; i < N; i += size){
    X[i] = D[i];
    for (int j = 0; j < N; j++){
      if (i != j)
        X[i] = X[i] - A[i * N + j] * Xo[j];
    }
    X[i] = X[i] * (1 / A[i * N + i]);
  }
  __syncthreads();
}
```

Výpis A.2: KERNEL jacobi

```
__global__ void finish(int N, float* X, float* Xo, bool* prec, \
float precision, bool* out){
  int thIdx = threadIdx.x;
  int gthIdx = thIdx + blockIdx.x * blockDim.x;
  const int gsize = blockDim.x * gridDim.x;
  float tmp;

  for (int i = gthIdx; i < N; i += gsize){
    tmp = X[i] - Xo[i];
    reduc[i] = fabs(tmp) < precision;
  }
  __syncthreads();
  for (int size = N/2; size>0; size/=2){
    if(gthIdx < size)
      reduc[gthIdx] = reduc[gthIdx] & reduc[gthIdx+size];
    __syncthreads();
  }
  if (gthIdx == 0)
    *out = reduc[0];
}
```

Výpis A.3: KERNEL finish

```
int cuJacobi(int N, float*& gpuA, float*& gpuD, float*& gpuXo, \
float*& gpuX, int iterations, float precision, bool*& prec, \
bool*& converg, bool*& gpuOUT){
  bool result;

  cudaMemcpyToSymbol(reduc, &converg, sizeof(bool*));
  convergence<<<1, blockSize>>>(N, gpuA, converg, gpuOUT);
  cudaMemcpy(&result, gpuOUT, sizeof(bool), cudaMemcpyDeviceToHost);

  if(!result)
    std::cout << "Convergence not guaranteed.\n";

  std::swap(gpuXo, gpuX);

  int k;
  for (k = 0; k < iterations; ++k){
    jacobi<<<gridSize, blockSize>>>(N, gpuA, gpuD, gpuXo, gpuX);

    cudaMemcpyToSymbol(reduc, &prec, sizeof(bool*));
    finish<<<gridSize, blockSize>>>(N, gpuX, gpuXo, prec, precision,gpuOUT);
    cudaMemcpy(&result, gpuOUT, sizeof(bool), cudaMemcpyDeviceToHost);
    if(result){
      return k + 1;
    }

    if (k != iterations - 1){
      std::swap(gpuXo, gpuX);
    }
  }
  return k;
}
```

Výpis A.4: Funkcia cuJacobi

Príloha B

Jacobi OpenMP

```
int jacobi(int N, float** A, float* D, float* Xo, float* X, \
int iterations, float precision){
  bool converg = true;
  bool prec;
  #pragma omp target teams distribute reduction(&&:converg)
  for (int i = 0; i < N; ++i){
    float sum = 0.0;
    #pragma omp parallel for reduction(+:sum)
    for (int j = 0; j < N; ++j){
      if (i != j)
        sum += fabs(A[i][j]);
    }
    converg = converg && (fabs(A[i][i]) > sum);
  }
  if (!converg)
    std::cout << "Convergence not guaranteed\n";
  std::swap(Xo, X);
  int k;
  for (k = 0; k < iterations; ++k){
    #pragma omp target teams distribute
    for (int i = 0; i < N; ++i){
      float sum = D[i];
      #pragma omp parallel for reduction(-:sum)
      for (int j = 0; j < N; ++j){
        if (i != j)
          sum -= A[i][j] * Xo[j];
      }
      X[i] = sum * (1 / A[i][i]);
    }
    prec = true;
    #pragma omp target teams distribute parallel reduction(&&:prec)
    for (int i = 0; i < N; ++i){
      float tmp = X[i] - Xo[i];
      prec = prec && (fabs(tmp) < precision);
    }
    if (prec)
      return k + 1;
    if (k != iterations - 1)
      std::swap(Xo, X);
  }
  return k;
}
```

Výpis B.1: Jacobiho metóda