VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

AKCELERACE ULTRAZVUKOVÉ NEUROSTIMULACE POMOCÍ VYSOKOÚROVŇOVÝCH GPGPU KNIHOVEN

ACCELERATION OF ULTRASOUND NEUROSTIMULATION USING HIGH-LEVEL GPGPU LIBRARIES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. RICHARD MIČKA**
AUTHOR

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

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2020/2021

Zadání diplomové práce

Student: **Mička Richard, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Strojové učení
Název: **Akcelerace ultrazvukové neurostimulace pomocí vysokoúrovňových GPGPU knihoven**
Acceleration of Ultrasound Neurostimulation Using High-Level GPGPU Libraries
Kategorie: Paralelní a distribuované výpočty
Zadání:

1. Seznamte s vysokoúrovňovými knihovnami pro tvorbu programů akcelerovaných pomocí grafických karet. Zaměřte se především na knihovny OpenACC a OpenMP.
2. Prostudujte implementace šíření ultrazvuku napsané v jazycích Matlab, C++ a CUDA z balíku k-Wave.
3. Navrhněte postup transformace C++ kódu určeného pro procesor do podoby vhodné pro grafické karty s využitím některé z vysokoúrovňových knihoven.
4. Navržený postup realizujte tak, aby výsledné zdrojové kódy podporovaly jak procesor, tak grafickou kartu.
5. Vyhodnoťte přesnost výpočtu a porovnejte ji s výsledky přesností implementací.
6. Proveďte detailní měření výkonnosti s rozborem limitujících faktorů. Porovnejte výsledky s původními implementacemi.
7. Srovnejte pracnost jednotlivých implementací.
8. Diskutujte vhodnost vybraných pro využití v dalších modulech balíku k-Wave.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 4 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 2020
Datum odevzdání: 19. května 2021
Datum schválení: 30. října 2020

Abstrakt

Táto práca sa zaoberá akceleráciou výpočtu simulácie šírenia akustických vĺn z balíku k-Wave pomocou GPGPU knižníc. Ako prvé, sú v práci preskúmané a ohodnotené dostupné vysokoúrovňové GPGPU knižnice. Následne je, po oboznámení sa so súčasným stavom riešenia akcelerácie simulácie v k-Wave, navrhnutý spôsob, ktorým je možné transformovať kód určený pre procesor, do podoby spustiteľnej aj na grafickej karte. Výsledkom tejto práce je aplikácia schopná akcelerovať výpočet simulácie na grafickej karte. V prípade neprítomnosti grafickej karty, je schopná bežať na procesore, s využitím vláknového a SIMD paralelizmu. Táto implementácia je následne ohodnotená z hľadiska výkonnosti, náročnosti a užitočnosti.

Abstract

This thesis explores potential use of GPGPU libraries to accelerate k-Wave toolkit's acoustic wave propagation simulation. Firstly, the thesis researches and assesses available high level GPGPU libraries. Afterwards, an insight into k-Wave toolkit's current state of simulation acceleration is provided. Based on that, an approach to enahnce currently available code for processors into a heterogeneous application, that is capable of being run on graphics card, is proposed. The outcome of this thesis is an applicaton that can utilize graphics card. If graphics card is unavailable, a fallback into thread and SIMD based acceleration for processor is executed. The product of this thesis is then evaluated based on its performance, maintenance difficulty and usability.

Kľúčové slová

akcelerácia, CUDA, GPGPU, GPU, k-Wave, neurostimulácia, OpenACC, OpenMP, simulácia, ultrazvuk

Keywords

acceleration, CUDA, GPGPU, GPU, k-Wave, neurostimulation, OpenACC, OpenMP, simulation, ultrasound

Citácia

MIČKA, Richard. *Akcelerace ultrazvukové neurostimulace pomocí vysokoúrovňových GPGPU knihoven.* Brno, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. Jiří Jaroš, Ph.D.

Akcelerace ultrazvukové neurostimulace pomocí vysokoúrovňových GPGPU knihoven

Prehlásenie

Prehlasujem, že som túto diplomovú 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.

.....
Richard Mička
18. mája 2021

Poďakovanie

Moje poďakovanie patrí vedúcemu práce, doc. Ing. Jiří Jarošovi, Ph.D., za odborné vedenie tejto práce, cenné rady a ochotu poskytnutú pri konzultáciách.

Obsah

Kapitola 1

Úvod

Grafické karty ponúkajú rádovo vyšší teoretický výkon ako procesory. Rozdiel v teoretickom výkone sa pohybuje v rozsahu od 2 do 20 násobného, v niektorých prípadoch ešte vyšších hodnotách [9]. Grafické karty ponúkajú svoj výpočetný výkon aj mimo vykresľovacie úlohy pomocou tzv. GPGPU rozhraní (GPGPU z angl. general-purpose GPU programming – všeobecné programovanie na GPU). Tieto rozhrania častokrát poskytujú abstrahovanú architektúru zariadenia, ktorá skrýva implementačné detaily funkcionality GPU a taktiež zvyšuje medzigeneračnú prenositelnosť aplikácií pre GPU. Medzi príklady takýchto rozhraní patrí napr. Nvidia CUDA, či AMD ROCm.

S pokrokom hardwaru nastáva čoraz väčšia divergencia medzi používanými architektúrami. V sfére superpočítania (ďalej ako HPC z angl. High Performance Computing) to má ako následok časté zmeny architektúr vrcholných superpočítačov. Nežiadúci dopad častých zmien v architektúre sa prejavuje predovšetkým v zvyšujúcej sa náročnosti a nákladnosti udržovania softvéru cieleného pre väčšie spektrum cieľových počítačov.

Preto sa čoraz viac rozrastá sektor tzv. heterogénneho programovania. Tento termín sa používa na systémy, kedy je program určený pre beh na rôznych typoch zariadení bez ohľadu na konkrétnu architektúru systému. Základným a populárnym heterogénnym systémom je systém pozostávajúci z procesora (ďalej ako CPU z angl. Central Processing Unit) a grafického akcelerátora (ďalej ako GPU z angl. Graphics Processing Unit).

Potenciál grafických kariet na akceleráciu výpočtov je taktiež využitý v balíku k-Wave na akceleráciu riešení simulácie šírenia ultrazvuku v parametricky popísanom prostredí. V rámci tejto práce sú preskúmané spôsoby, ktorými sú výpočty v balíku k-Wave akcelerované na procesoroch alebo grafických kartách. V súčasnosti sú tieto implementácie spravované ako dve samostatné implementácie, obsahujúce veľké množstvo spoločného kódu. Správa takto divergujúceho kódu si vyžaduje zbytočné úsilie pri jeho rozširovaní, kedy treba podobné kroky realizovať v oboch podobných projektoch. Preto je v rámci tejto práce preskúmaná možnosť vytvorenia jednej spoločnej implementácie, ktorá by zastrešovala možnosť výpočtu na grafickej karte, ale aj na procesore, v prípade, že grafický akcelerátor nie je k dispozícii. V rámci práce je navrhnutý a implementovaný postup transformácie systému pôvodne zameraného pre procesor pomocou knižnice OpenMP a FFTW na systém podporujúci grafické karty a procesory zároveň, pomocou dostupných nástrojov na heterogénne programovanie.

Práca začína tematikou GPGPU a popisom knižníc na programovanie GPGPU v kapitole 2. Následne sú v tejto kapitole popísané vysokoúrovňové knižnice poskytujúce možnosti na heterogénne programovanie pre systémy pozostávajúce z CPU a GPU. Tieto knižnice sú posúdené z hľadiska výkonu a vhodnosti na riešenie tejto diplomovej práce.

V nasledujúcej kapitole 3, je priblížený balík k-Wave a jeho moduly na akceleráciu výpočtu. Popísaná je funkcionalita týchto rozširujúcich modulov, ktoré umožňujú urýchlenie výpočtu o niekoľko rádov. Keďže ide o viacero samostatne vyvíjaných aplikácií, je diskutovaná ich príbuznosť, z ktorej sa následne vychádza v rámci implementácie tejto práce. Taktiež je dostatočne popísaná funkcionalita testovacieho prostredia, ktoré je využité pri tvorbe implementácie. Následne je predstavený vhodný a fungujúci postup transformácie aplikácie pre procesor s OpenMP na heterogénnu aplikáciu podporujúcu CPU aj GPU. V kapitole 5 sú popísané dielčie úlohy, z ktorých pozostáva implementácia tejto práce.

Následne sú, v kapitole 6, vyhodnotené dosiahnuté výsledky v rámci tejto práce. Výsledná implementácia je ohodnotená z hľadiska udržania numerickej presnosti a aj výkonnosti. V tejto kapitole sa taktiež pojednáva o pracnosti tohto riešenia a jeho využiteľnosti v ostatných moduloch balíka k-Wave.

Kapitola 2

Technológie na GPGPU výpočty

V tejto kapitole sú uvedené hlavné dostupné technológie používané na akceleráciu výpočtov, ktoré boli v rámci tejto diplomovej práce použité, alebo sú pre túto prácu relevantné. Špeciálna pozornosť je venovaná práve vysokoúrovňovým rozhraniam OpenMP a OpenACC, ktoré poskytujú rozhrania na akceleráciu výpočtov na grafickej karte. Jednotlivé systémy sú popísané a sú vysvetlené princípy ich fungovania. Následne sú tieto nástroje porovnané z hľadiska viacerých vhodne zvolených faktorov relevantných v rámci tejto práce. Na základe tohto porovnania je zvolený vhodný nástroj, s použitím ktorého bola táto práca realizovaná.

2.1 Nvidia CUDA

Rozhranie Nvidia CUDA je rozhranie poskytujúce GPGPU (angl. General-purpose GPU programming) funkcionalitu na grafických kartách spoločnosti Nvidia. Pod pojmom GPGPU sa rozumie akékoľvek programovanie na GPU za iným účelom ako vykresľovanie grafiky. Tento spôsob sa najprv realizoval s využitím grafických API ako OpenGL. S postupom výkonu grafických kariet a so zväčšujúcim sa potenciálom GPGPU využitia grafických kariet, bola zmenená samotná architektúra hardware grafických kariet a zároveň bolo vytvorené rozhranie poskytujúce GPGPU funkcionalitu pre grafické karty Nvidia [16].

Toto rozhranie abstrahuje hardware grafických kariet do jednoduchšieho modelu spoločného pre viacero generácií grafických kariet. Programuje sa v jazyku založenom na C/C++, ktorý sa kompiluje do jazyku pre virtuálny stroj operujúci nad inštrukčnou sadou Parallel Thread Execution (PTX).

Ekosystém CUDA sa skladá z viacerých vrstiev [16]. Na najnižšej úrovni je poskytovaná základná funkcionality v podobe *Driver API*. Na ňom je postavená behová knižnica *CUDA Runtime*, nad ktorou sú poskytované vysokoúrovňové knižnice *cuBLAS* – poskytujúca knižnicu pre podporu lineárnej algebry, *cuFFT* – knižnica implementujúca diskrétnu Fourierovu transformáciu (ďalej ako DFT) pomocou Fast Fourier Transform algoritmu (ďalej ako FFT), a mnoho ďalších.

Nevýhodou CUDA knižnice je predovšetkým stále celkom nízka úroveň programovania. Je potrebné explicitne a pracne spravovať pamäť na grafickej karte, tzn. duplikujú sa ukazovatele, s ktorými je nutné pracovať. Vplyv tejto nevýhody je možné do značnej miery znížiť využitím CUDA Unified Memory, kedy je spojený adresný priestor procesora s adresným priestorom grafickej karty za cenu pridanej réžie pri prenosoch. Ďalšou nevýhodou knižnice CUDA je naviazanosť na platformu grafických kariet Nvidia spôsobujúca nulovú prenositeľnosť kódu na karty iného výrobcu ako Nvidia.

2.1.1 Parallel Thread Execution

Táto inštrukčná sada má za úlohu reprezentovať grafickú kartu ako paralelný výpočetný stroj, ktorý je stabilnejší ako samotný hardware implementujúci jeho funkcionalitu, ktorý sa podstatne mení každú generáciu. Zároveň však poskytuje ľahko preložitelnú reprezentáciu, ktorá je schopná dosiahnuť výsledky porovnateľné s natívnym kódom GPU. Tento virtuálny stroj a jeho inštrukčná sada poskytuje spoločný cieľový jazyk, pre kompilátory založené nielen na jazyku C/C++. Do instrukčnej sady pre PTX taktiež prekladajú, v prípade využitia grafickej karty Nvidia, v ďalších častiach uvedené vysokoúrovňové knižnice OpenMP a OpenACC. Kód pre PTX býva súčasťou spustiteľného binárneho súboru.

2.2 OpenCL

Alternatívou Nvidia CUDA knižnice je knižnica OpenCL skupiny Khronos Group, Inc. Táto knižnica poskytuje GPGPU funkcionalitu postavenú na instrukčnej sade SPIR-V (angl. Standard, Portable Intermediate Representation - V), ktorá unifikuje rozhranie pre grafické karty pre knižnice skupiny Khronos. Okrem OpenCL na GPGPU sú na SPIR-V založené aj knižnice OpenGL a Vulkan primárne určené na grafické vykresľovanie s čiastočnou podporou pre GPGPU v podobe *compute shaderov*, ktoré do SPIR-V prekladajú programy v jazyku pre shadery GLSL [11].

2.2.1 SPIR-V

SPIR-V predstavuje medzijazyk na paralelné výpočty, vyvinutý prvotne pre implementáciu OpenCL funkcionality, ktorý bol neskôr rozšírený o podporu grafických rozhraní. Zabezpečuje jednotný formát kódu spustiteľného na rôznych architektúrach grafických kariet (aj medzi výrobcami). Zároveň tak umožňuje zdieľanie nástrojov na generáciu či preklad do iných jazykov.

Táto inštrukčná sada je portabilná medzi viacerými architektúrami grafických kariet. Podporujú ju grafické karty spoločností AMD, Nvidia aj Intel.

2.3 OpenMP

OpenMP je rozhranie na programovanie aplikácií (ďalej ako API, z angl. *Application Programming Interface*), ktoré špecifikuje prostriedky na štandardizované a prenositeľné paralelizovanie algoritmov v programoch. Toto rozhranie je špecifikované na vysokej úrovni štandardom, ktorý popisuje potrebné prostriedky, ktoré by mal poskytovať kompilátor poskytujúci podporu OpenMP. Hlavným rozhraním, ktoré OpenMP poskytuje sú tzv. direktívy kompilátoru, ktoré sa vkladajú medzi bežný zdrojový kód programu. Tieto direktívy sú definované tak, aby mohli byť prípadne chýbajúcej podpory kompilátorom vynechané a funkcionalita kódu sa nezmenila. Realizácie kódu na akcelerátore je možná v rámci oblastí označených príslušnou direktívou ako `target` oblasť. Pri kompilácii na akceleráciu na platforme grafických kariet Nvidia, sú regióny akcelerované na grafickej karte prekladané do inštrukcií pre PTX.

Okrem samotných direktív, reprezentujúcich programovací model, štandard OpenMP vyžaduje aj jednotný model vykonávania kódu, spresňuje požiadavky na nízkoúrovňové rutiny, do volania ktorých sa transformujú jednotlivé direktívy kompilátoru. Taktiež špecifikuje udalosti a premenné prostredia OpenMP, ktorými sa ovláda jeho chovanie, ako napr.

počet používaných vlákien. Udalosti sú vyvolávané pri rôznych krokoch, vďaka čomu je možné profilovať či ladiť aplikáciu napísanú v OpenMP.

OpenMP bolo v minulosti sústredené výhradne na akceleráciu paralelizovním kódu pre procesor. Od verzie 4.5, vydanej v roku 2015, do jeho špecifikácie pribudla základná podpora prostriedkov poskytujúcich akceleráciu výpočtu aj na grafických kartách alebo na externých akcelerátorch ako napr. Intel Xeon Phi [13]. Novšia špecifikácia OpenMP 5.0 vydaná v 2018 túto podporu ďalej rozšírila.

Voľne dostupné kompilátory s podporou OpenMP sú Clang [1] a GNU Compiler Collection[2] (ďalej ako GCC). U týchto kompilátorov síce existuje podpora pre najnovšie direktívy na akceleráciu výpočtu na grafickej karte, avšak táto podpora je momentálne stále pod aktívnym vývojom. Ďalším nedostatkom je, ako je ukázané neskôr v časti 2.6.1, že výkonnosť tejto akceleračnej funkcionality pri výpočetne náročných úlohách je nepostačujúca. Podobné zistenia boli získané aj v práci [2]. Medzi ďalšie kompilátory podporujúce OpenMP s podporou akcelerácie, avšak pre túto prácu nevhodnej architektúre, patria napr. riešenia od AMD podporujúce grafické karty AMD, XL od IBM pre platformy IBM Power. Aktuálny prehľad kompilátorov s popisom štádia podpory funkcionalít OpenMP, je vedený na stránke organizácie OpenMP ARB, ktorá zodpovedá za publikáciu štandardu OpenMP[3].

2.4 OpenACC

OpenACC je rozhranie zamerané na paralelizáciu výpočtu na akcelerátoroch. Vzniklo odvetvením z OpenMP a jeho prvá verzia vznikla už v roku 2011 [7]. Aj vďaka tomu je v súčasnej dobe táto knižnica dostatočne vyvinutá a overená. Podobne ako OpenMP, je OpenACC štandard pozostávajúci z viacerých častí. Jeho hlavnou časťou je programovací model, ktorý abstrahuje architektúry akcelerátorov [3]. Ten je reprezentovaný kernelmi, ktoré sú vykonávané gangami, workermi, ktoré sa delia na vektory. Paralelizáciu a distribúciu práce v tomto modeli zabezpečujú direktívy kompilátoru v oblastiach `parallel`. Následne je tento model transformovaný do spustiteľného kódu architektúry akcelerátora. Úlohou kompilátora je správne priradiť modelu paralelizácie OpenACC reprezentáciu v cieľovej architektúre. V prípade využitia na akceleráciu na grafickej karte Nvidia sú oblasti kernelov (tj. výpočtových rutín, ktoré bežia na GPU) prekladané do jazyku PTX.

Direktívy kompilátoru sú prekladané do volaní vnútorných funkcií OpenACC, ktoré špecifikuje štandard. Okrem direktív a vnútorných nízkoúrovňových rutín taktiež špecifikuje model pamäte, ktorá je oddelená od pamäte procera. Taktiež špecifikuje udalosti, ktoré sú použiteľné pri profilovaní či ladení.

Hlavnými kompilátormi podporujúcimi OpenACC je kompilátor od The Portland Group (ďalej ako PGI) a kompilátor Cray od Cray Inc. Podpora OpenACC však už v najnovších kompilátoroch od spoločnosti Cray bola upustená v prospech OpenMP. Kompilátor PGI v súčasnosti spadá pod spoločnosť Nvidia, kde je stále vyvíjaný a podporovaný ako súčasť Nvidia HPC SDK[4] – NVC++.

[1] https://clang.llvm.org/
[2] https://gcc.gnu.org
[3] https://www.openmp.org/resources/openmp-compilers-tools/
[4] https://developer.nvidia.com/hpc-sdk

2.5 SYCL

Odlišný prístup poskytuje technológia SYCL od skupiny Khronos Group. Pôvodný úmysel fungovania SYCL bol čisto nad knižnicou OpenCL a SPIR-V. Avšak stále rastie podpora a adopcia SYCL štandardu od iných vývojarov kompilátorov[5], ako napríklad hipSYCL, ktorý implementuje SYCL nad NVIDIA CUDA and AMD HIP[12], či adopcia v systéme Intel SYCL oneAPI DPC++, ktorý podporuje kompiláciu do OpenCL a Nvidia-PTX.

Na rozdiel od OpenMP a OpenACC je jeho programovací model pre heterogénne programovanie založený na štandardnom C++ s využitím moderných prostriedkov jazyka ako sú templates (šablony), či lambda funkcie, bez nutnosti špecifikácie výpočetných oblastí pomocou direktív kompilátoru. Negatívom tohto riešenia je opäť jeho nevyvinutosť a stále aktívny vývoj.

2.6 Vzájomné porovnanie

V predchádzajúcej časti boli predstavené knižnice OpenMP a OpenACC, ktoré umožňujú programovať GPGPU výpočty na vyššej úrovni. S ich použitím je možné získať spoločný kód spustiteľný či už na procesore alebo grafickej karte. V tejto časti sú tieto knižnice vzájomne porovnané z hľadiska výkonu a vhodnosti na použitie pri riešení tejto semestrálnej práce.

2.6.1 Porovnanie výkonu

Súčasťou analýzy dostupných technológií bolo aj porovnanie výkonnosti knižníc OpenMP a OpenACC na sade jednoduchých testovacích programov. Tieto programy boli zvolené tak, aby reprezentovali možné typy úloh, s ktorými je možné sa stretnúť v rámci akcelerácie simulácie. Výkon týchto knižníc je porovnaný s výkonom dosiahnuteľným s použitím nízkoúrovňovej knižnice CUDA. Porovnané boli riešenia OpenMP kompilátorov GCC a Clang, a OpenACC od kompilátoru PGI.

Zvolené úlohy na test výkonnosti kódu boli:

- saxpy kernel,
- násobenie matíc,
- kernel kopírujúci dáta.

Saxpy kernel reprezentuje typ úlohy, kedy sa načítava a ukladá veľké množstvo dát pri nízkej aritmetickej intenzite. Výsledky meraní sú zobrazené v tabuľke 2.1. Výsledok ukazuje výrazne nedostatočnú výkonnosť implementácie OpenMP akcelerácie kompilátoru GCC. Analýza príčiny slabého výkonu kernelu kompilovaného GCC poukazovala značne väčšie množstvo prenosu dát medzi systémovou pamäťou a pamäťou grafickej karty, ako u ostatných implementácií. Bez podrobnej analýzy výsledného PTX kódu prípadne analýzy volaní do OpenMP runtime knižnice však nie je možné s istotou určiť konkrétny problém, čo spôsobuje tieto transakcie medzi systémom a grafickou kartou.

Keďže riešenie poskytované kompilátorom GCC sa ukázalo ako nedostatočné, v ďalších testoch už nebude uvádzaný.

[5] https://www.khronos.org/sycl/

Riešenia OpenACC kompilátoru PGI a OpenMP kompilátoru Clang ukazujú výsledky porovnateľné v rámci odchýlky pri meraní s vlastnou implementáciou kernelu v CUDA. Taktiež je vidieť, že použitie dátového typu nemá negatívny dopad na rýchlosť kernelu.

Tabuľka 2.1: Porovnanie časov dĺžky behu saxpy kernelu na 83886080 prvkoch rôzneho dátoveho typu, CPU Ryzen 5 3600XT, GPU Nvidia GTX 1080 Ti 11GB

	Čas [μ s]	Čas [μ s]	Čas [μ s]	Čas [μ s]	Čas [μ s]
Dátový typ	float	double	std::uint32	std::uint16	std::uint8
CPU OpenMP variant	30025	59705	29552	19149	7192
OpenMP GCC	53425	63076	52754	53420	54447
OpenMP Clang	2945	5769	2891	1897	1665
OpenACC PGI	3180	5702	2882	1655	1780
CUDA	3277	5717	2870	1519	1576

Ako ďalšie meranie bolo uskutočnené porovnanie výkonu pri násobení matíc. Výkon bol na vhodnejšie porovnanie vyhodnotený v GFLOP/s (1 GFLOP/s je jedna miliarda operácií nad číslami s pohyblivou desatinnou čiarkou za sekundu). Získané výsledky merania sú uvedené v tabuľke 2.2. Ako referenčná hodnota je uvedený výkon cuBLAS kernelu `cublasSgemm` predstavujúci teoretický maximálny dosiahnuteľný výkon v danom prípade. Vo výsledkoch je výrazny rozdiel medzi OpenACC implementáciou a OpenMP Clang implementáciou. Samozrejme rozdiel medzi cuBLAS implementáciou a porovnávanými knižnicami je ešte výraznejší. Je potrebná brať ohľad na to, že implementácie v OpenMP a OpenACC sú naívne implementácie algoritmu, nevyužívajú zdieľanú pamäť a nie je v nich využité žiadne znovupoužitie dát. V prípadoch, kedy je žiadaný vysoký výpočetný výkon kernelu, je možné využiť knižnice CUDA, prípadne iných tretích strán, na akceleráciu výpočtu mimo kernel sekciu OpenACC či OpenMP. Obe knižnice poskytujú možnosť na extrahovanie ukazovateľa na dáta v pamäti grafickej karty, čím je možné zavolať CUDA rutinu ako bežne pri použití CUDA.

Tabuľka 2.2: Dosiahnutý výkon pri násobení matíc floating point 4096×4096 (CPU Ryzen 5 3600XT, GPU Nvidia GTX 1080 Ti 11GB)

	Výkon [GFLOP/s]
Triviálna CPU OpenMP implementácia	20
Clang OpenMP	192
PGI OpenACC	639
CUDA cuBLAS	9086

Posledný test bol test zameraný na porovnanie maximálnej priepustnosti pamäte pri presune device–device. Keďže úloha riešená v rámci tohto semestrálneho projektu je, mimo kernel počítajúci FFT, brzdená priepustnosťou pamäte kvôli nízkej intenzite, je vhodné overiť, či sú knižnice schopné využiť dostupnú priepustnosť pamäte. Výsledky merania sú v tabuľke 2.3. Výsledky poukazujú, že obe implementácie sú schopné využiť priepustnosť pamäte na približne rovnakú úroveň ako kernel v CUDA. Toto meranie teda nepoukázalo na nedostatok, ktorý by mohol obmedzovať akceleráciu úlohy riešenej v zvyšku tejto práce.

Tabuľka 2.3: Dosiahnutá priepustnosť globálnej pamäte pri kopírovaní 83886080 prvkov float datatypu (CPU Ryzen 5 3600XT, GPU Nvidia GTX 1080 Ti 11GB)

	Priepustnosť pamäte [GB/s]
CUDA	167
OpenMP Clang	160
OpenACC PGI	164

Výsledkom týchto meraní je teda zistenie, že v prípade jednoduchých pamäťou obmedzených úloh, ktoré nie sú výpočetne obmedzené, OpenMP kompilátoru Clang konkuruje OpenACC implementáci kompilátoru PGI. V prípade úlohy brzdenej výpočtovou rýchlosťou, OpenACC generuje výkonnejší kernel. Avšak v prípade, kedy by bola požadovaná čo najvyššia rýchlosť, je možné využiť knižnice nižšej úrovňe, ako napríklad cuFFT na rýchlu implementáciu Fourierovej transformácie ako volanie do akejkoľvek inej knižnice importovanej do programu.

2.6.2 Zhrnutie a vyhodnotenie

Štandard OpenMP má silnejšiu podporu medzi výrobcami hardwaru. Naopak u štandardu OpenACC chýba podpora pre dôležité architektúry od firiem ARM, AMD, či Intel. Jediným poskytovateľom stále aktívne udržovaného kompilátoru podľa štandardu OpenACC je firma PGI s ich PGI kompilátorom. Dôkazom upadajúcej podpory OpenACC je aj napríklad nie príliš dávny prechod firmy Cray, ktorá od verzie kompilátoru CCE 9.0 ukončila podporu OpenACC v prospech OpenMP [1]. Firma Cray pri tom bol jedným zo zakladajúcich členov konzorcia, ktoré vytvorilo OpenACC. Avšak, v súčasnej situácii, výhody vyspelosti OpenACC kompilátora PGI pre Nvidia platformu a jeho výkon hlboko prevyšujú technológie OpenMP, ktoré sa preukázali doposiaľ ešte v začiatočnom štádiu vývoja.

Kompilátor PGI okrem lepšieho výkonu poskytuje lepší ekosystém v podobe fóra, kde je možné nájsť riešenia pre časté, v minulosti vyriešené problémy. Ďalšou veľkou výhodou PGI kompilátora je detailný manuál pre samotný kompilátor, a príručka [6].

Na protipríklad, kompilátory Clang a GCC vo východzej konfigurácii nepodporujú akceleráciu na grafických kartách pomocou OpenMP. Kompilátory je potrebné kompilovať zo zdrojového kódu. Pri kompilácii je potrebné poskytnúť viacero dôležitých prepínačov, ktoré je potrebné nastavovať manuálne. Tento proces je čiastočne zdokumentovaný pre GCC kompilátor[6], pre Clang oficiálna dokumentácia neexistuje a je potrebné čerpať a kombinovať znalosti zo stránok tretích strán.

GCC kompilátor podporujúci akceleráciu na grafickej karte musí byť nainštalovaný samostatne v systéme popri základnom kompilátore GCC, ktorý si v prípade vyžiadania offloadingu paralelizácie zavolá druhú verziu nastavenú na kompiláciu akcelerovaného kódu.

V prípade Clang kompilátoru, je najprv vyžadované skompilovať kompilátor nastavený pre podporu offloading OpenMP akcelerovaných direktív. Následne je však nutné skompilovať kompilátor znovu na vytvorenie dynamicky linkovateľných knižníc poskytujúcich podporu volaní použiteľných v rámci regiónov, ktoré sú implementované v natívnom jazyku akcelerátora. Tieto dynamicky linkovné knižnice musia byť prítomné a dostupné na

[6] https://gcc.gnu.org/wiki/Offloading

zariadení, ktoré chce spúšťať kód akcelerovaný na grafickej karte skompilovaný pomocou Clang kompilátora.

Je však opodstatnené očakávať, že v blízkej budúcnosti budú riešenia založené na štandarde OpenMP konkurencieschopnejšie, či už s ohľadom na výkonnosť kódu akcelerátora, ale aj priaznivejšie z hľadiska užívateľskej podpory a použiteľnosti. Nárast popularity a použiteľnosti sa dá taktiež očakávať u štandardu SYCL, ktorý je postupne viac adoptovaný rôznymi výrobcami akceleračného hardware.

Kapitola 3

Balík k-Wave

Balík k-Wave je rozširujúci balík do prostredia MATLAB. Balík ponúka rozhranie poskytujúce jednoduchú a rýchlu prácu so simuláciami šírenia akustického a ultrazvukového vlnenia v médiu. Jadro jeho funkcionality predstavuje modul na výpočet riešení diferenciálnych rovníc šírenia akustického vlnenia v médiu popísaného pomocou rôznych parametrov. Hlavnými parametrami média je jeho homogénnosť či heterogénnosť, následne heterogénnosť jeho hustoty a vlastnosti ovplyvňujúcich šírenie vlnenia, ako napr. spôsob simulácie absorpcie.

Úloha šírenia vlnenia akustického vlnenia v kvapalnom médiu je riešená v rámci modulu `kSpaceFirstOrder`. Tento názov bude používaný spoločne pre všetky alternatívy tohto modulu v balíku k-Wave, ako sú `kSpaceFirstOrder1D`, `kSpaceFirstOrder2D`, `kSpaceFirstOrder3D`, `kSpaceFirstOrderAS`. Tento modul rieši parciálnu diferenciálnu rovnicu prvého rádu pomocou k-priestorovej pseudo-spektrálnej metódy, popísanej v [15]. Bežné postupy riešenia diferenciálnych rovníc, ako napr. metóda konečných diferencií alebo metóda konečných prvkov, sú pri simulácii šírenia akustického vlnenia menej užitočné, pretože vyžadujú príliš hustú mriežku na dosiahnutie akceptovateľnej presnosti, čo má za príčinu vysokú výpočtovú náročnosť [14]. Metóda použitá v k-Wave redukuje požiadavok na hustotu mriežky tým, že aproximuje hodnoty gradientov sledovaných veličín nie na základe susedných bodov mriežky, ale na základe priestorovej frekvencie, ktorá je aproximovaná pomocou Fourierovej transformácie celej domény.

Jadro balíku k-Wave je celé vytvorené v jazyku MATLAB. Avšak existujú k nemu dodatočné balíčky, rozšírenia, ponúkajúce alternatívne implementácie niektorých jeho modulov s vyšším výkonom. Pre vyššie spomínaný modul `kSpaceFirstOrder` existujú viaceré implementácie.

Prvou z týchto implementácií je verzia pre počítače so zdieľanou pamäťou, napísaná v jazyku C++ s využitím knižnice OpenMP. Použitím tejto knižnice je možné využiť vláknový paralelizmus a taktiež dátový paralelizmus s využitím SIMD inštrukcií (z angl. *single instruction multiple data*), ktoré poskytujú mnohé procesory súčasnosti.

Druhou implementáciou je verzia pre systémy s grafickým akcelerátorom. Je napísaná v jazyku C++ a CUDA, ktorý umožňuje využitie vláknového a dátového paralelizmu SIMT (z angl. *single instruction multiple thread*) grafických kariet Nvidia. Táto implementácia poskytuje rádovo vyšší výkon ako predchádzajúca pre procesory, avšak je obmedzená veľkosťou pamäte grafickej karty, ktorá je v súčasnosti obmedzená na desiatky gigabajtov.

Pre väčšie domény existuje implementácia v C++ s využitím knižnice OpenMPI. Pomocou tejto knižnice je možné distribuovať výpočet problému väčšej domény po častiach na viacero uzlov superpočítača. Tým sa zníži pamäťová náročnosť na jeden uzol, za cenu kom-

plikovanejšieho výpočtu po častiach a cenu vyššej réžie pri komunikácii medzi susednými uzlami.

Okrem vyššie opísaných rozšírení pre akceleráciu modulu `kSpaceFirstOrder`, existuje ešte rozšírenie modulu `acousticFieldPropagator`, ktorý slúži pre simuláciu akustických polí generovaných parametricky popísanými zdrojmi akustického vlnenia. Tento rozširujúci modul je implementovaný v jazyku C++ s využitím knižnice OpenMP pre paralelizáciu na počítačoch so zdieľanou pamäťou.

Popri moduloch na simuláciu akustických vlnení, balík k-Wave taktiež obsahuje modul na testovanie jeho výkonnosti `benchmark`. Tento modul generuje postupne ťažšie zadania úloh na simuláciu zväčšovaním domény. Následne meria dĺžky behu simulácie. Tento modul bol využitý pri vyhodnocovaní výkonu implementácie tvorenej v tejto práci a jej porovnaní s ostatnými implementáciami v časti 6.2.

Za zmienku taktiež stojí testovací modul `kWaveTester`, ktorý ponúka možnosť porovnať výsledky simulácie, realizovanej rozširujúcou implementáciou `kSpaceFirstOrder` v C++ alebo CUDA, so simuláciou realizovanou v prostredí MATLAB. Tento modul generuje rôzne kombinácie parametrov simulácie, čím sa snaží o čo najlepšie pokrytie možných vstupných parametrov a výstupných veličín. Túto funkcionalitu ponúka pre simulácie v dvojrozmernom a trojrozmernom priestore.

3.1 Akcelerácia výpočtov v k-Wave

Základná implementácia výpočtov v balíku k-Wave je realizovaná pomocou modulov v jazyku MATLAB. Tento jazyk je interpretovaný za behu, a preto sú možnosti jeho optimalizácie obmedzené. Implementácia výpočtov v jazyku MATLAB postačuje na malé domény. Jeho výpočtovú výkonnosť je možné rozšíriť pomocou ďalšieho rozširujúceho balíku prostredia MATLAB, Parallel Computing Toolbox [5]. Tento balík je schopný realizovať výpočty definované v jazyku MATLAB na GPU akcelerátore. Výkonnosť takto akcelerovaného výpočtu je v porovnaní s pôvodným algoritmom v MATLAB pre CPU značne vyššia, viď tab. A.1 v porovnaní s tab. A.2 a [14]. Avšak aj táto implementácia trpí rádovo horšou výkonnosťou, a aj simulácie domény o veľkosti $512 \times 512 \times 256$ trvajú v rádoch minút.

Problém s prudko rastúcou výpočtovou náročnosťou je v balíku k-Wave riešený pomocou rozšírení, ktoré implementujú funkcionalitu výpočtovo náročného modulu pomocou natívneho kódu pre CPU alebo GPU. Tieto rozširujúce aplikácie sú distribuované ako spustiteľné súbory či zdrojové súbory programov, dostupné na oficiálnom portáli k-Wave[1]. V prípade špeciálnej potreby je tak možné tieto rozšírenia skompilovať s odlišnými parametrami, ako tými, s ktorými bol skompilovaný distribuovaný spustiteľný súbor.

Hlavnou možnosťou pri vlastnej kompilácii je možnosť voľby kompilátora a knižnice, ktorá poskytuje výpočet diskrétnej Fourierovej transformácie (ďalej len ako DFT). Výber je medzi kombináciami kompilátoru Intel s knižnicou Intel MKL[2] na výpočet DFT a kompilátoru GCC s knižnicou FFTW[3] na výpočet DFT. Ďalej je možné voliť spôsob linkovania s knižnicami.

Implementácia poskytujúca akceleráciu na GPU pomocou jazyka C++ a CUDA taktiež poskytuje okrem možností spôsobu linkovania, možnosť kompilátora, ktorý je vnútorne použitý pri kompilácii pomocou kompilátora CUDA `nvcc`.

[1] http://www.k-wave.org/download.php
[2] https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html
[3] http://www.fftw.org

3.1.1 Rozhranie k akcelerácii pomocou OpenMP/CUDA z prostredia MATLAB

Stiahnuté alebo vlastnoručne skompilované spustiteľné súbory je potrebné umiestniť do zložky `binaries` v adresárovej štruktúre balíka k-Wave. Následne je možné využiť akceleráciu, ponúkanú implementáciami v C++ s využitím OpenMP či CUDA, zavolaním príslušného modulu, napr. miesto `kSpaceFirstOrder3D` by šlo o `kSpaceFirstOrder3DC` pre verziu s OpenMP, `kSpaceFirstOrder3DG` pre verziu s CUDA. Tieto moduly obaľujú funkcionalitu týchto rozširujúcich implementácií do rozhrania v jazyku MATLAB rovnakého ako modul `kSpaceFirstOrder`.

Vnútorne tieto moduly generujú vstupný súbor vo formáte HDF5. Generácia vstupného súboru je realizovaná samotným modulom zodpovedným za simuláciu v prostredí MATLAB. Pomocou vstupného argumentu `'SaveToDisk'` je samotný výpočet vynechaný, namiesto toho sa vyexportuje vstupný súbor pre implementáciu pomocou OpenMP alebo CUDA.

Následne je spustený podproces, ktorý spustí príslušný spustiteľný súbor so vstupným súborom, vygenerovaným v predošlom kroku. Po dobehnutí procesu je tento súbor načítaný a navrátený v rovnakej podobe, akoby bol výpočet realizovaný pôvodnou implementáciou v jazyku MATLAB.

Podprogramy akcelerujúce výpočet funkcií `kspaceFirstOrder` pomocou OpenMP či CUDA sú vytvorené tak, aby boli schopné fungovať nezávisle od prostredia MATLAB, a mohli tak počítať aj na systéme, kde MATLAB nie je k dispozícii. Ich hlavným rozhraním sú vstupné súbory, ktoré prenášajú parametre simulácie, ktorá bude vykonávaná. Ďalšie rozhranie sú argumenty príkazového riadka poskytnuté pri spustení programu. Pomocou týchto parametrov je možné určovať veličiny, ktoré budú obsiahnuté vo výstupnom súbore a v akej podobe. Je takto možné zvoliť rôzne redukčné operácie aplikované na sledované veličiny [14]. Tieto súbory využívajú formát HDF5, umožňujúci organizovaný a zároveň efektívny spôsob ukladania, ktorý je popísaný v nasledujúcej časti.

Hierarchical Data Format

The Hierarchical Data Format version 5 (HDF5) je voľne dostupný dátový formát poskytujúci rozhranie pre veľké, komplexné a rôznorodé dáta, zachovávajúc organizovanú štruktúru [10]. Štruktúra je v rámci súborov tohto typu tvorená *grupami* (groups) a *datasetmi* (datasets). Tieto objekty môžu mať priradené metadáta vo forme *atribútov* (attributes). Tento dátový formát umožňuje organizovane spravovať veľké kvantá rôznorodých dát, čím sú vhodnou voľbou na reprezentáciu parametrov a vstupných hodnôt simulácie. Pri výstupe zasa do nich môže byť priebežne zapisované, do viacdimenzionálnych datasetov, kde jedna z dimenzií môže byť časový krok. Použitie tohto štandardného formátu zároveň zaisťuje úplnú izoláciu MATLAB kódu od C++ kódu využívajúceho OpenMP, resp. CUDA. Tým je zabezpečená spustiteľnosť a možnosť realizovať simulácie na odlišnom systéme od systému, na ktorom bola vstupná úloha vygenerovaná.

3.1.2 Analýza modulov na akceleráciu simulácie pomocou OpenMP a CUDA

V tejto časti sú rozobrané potrebné podrobnosti vnútornej funkcionality spustiteľných súborov `kspaceFirstOrder-OMP` a `kspaceFirstOrder-CUDA`, ktoré poskytujú akceleráciu na procesore s využitím OpenMP, resp. na grafickej karte s využitím rozhrania CUDA. Sú prezentované ich zdieľané časti a porovnané časti, v ktorých sa ich implementácia líši. Na

základe analýzy jednotlivých implementácií bude následne navrhnutý spôsob, ktorým môžu byť tieto dve implementácie využité pri tvorbe heterogénnej implementácie, pomocou C++ a OpenACC. Táto implementácia by mala byť schopná využiť akceleráciu GPU, či CPU, v prípade nedostupnosti GPU alebo vynútením behu na CPU používateľom.

Architektúra modulov

Implementácie modulov slúžiacich na akceleráciu pomocou OpenMP a CUDA zdieľajú viacero častí architektúry. Simulácia je spravovaná triedou `KSpaceFirstOrderSolver`, ktorá obaľuje alokáciu prostriedkov, jednotlivé kroky výpočtu a aj jeho výstupy. Pri štarte oboch implementácií simulácií sa ako prvé načítajú vstupné parametre, predané pomocou argumentov príkazového riadka pri vytvorení procesu. Tieto parametre spracováva trieda `CommandLineParameters`, ktorá je súčasťou triedy zastrešujúcej všetky globálne parametre počas behu programu, triedy `Parameters`. Táto trieda je dostupná pod návrhovým vzorom singleton (angl. singleton – jedináčik), čím je zaručená existencia jednej unikátnej inštancie tejto triedy po celú dĺžku behu programu.

Po spracovaní parametrov simulácie, inštancia alokuje všetky potrebné matice, používané pri výpočte. Matice používané pri výpočte spravuje inštancia triedy `MatrixContainer`. Počas alokácie pamäte je pamäť alokovaná aj pre výstupné prúdy dát simulácie. Výstupné prúdy spravuje inštancia triedy `OutputStreamContainer`, ktorá alokuje pamäť pre vzorkovanie alebo ukladanie celej domény pre sledované veličiny. Veličiny, ktoré majú byť sledované sú volené pomocou vstupných parametrov predávaných príkazovým riadkom.

Po alokovaní pamäte potrebnej pre simuláciu, nasleduje načítanie vstupných dát simulácie do príslušných matíc. Pred spustením simulácie sú ešte prevedené prípravné kroky potrebné pre niektoré veličiny a zároveň v tomto kroku sú inicializované knižnice zodpovedné za výpočty DFT.

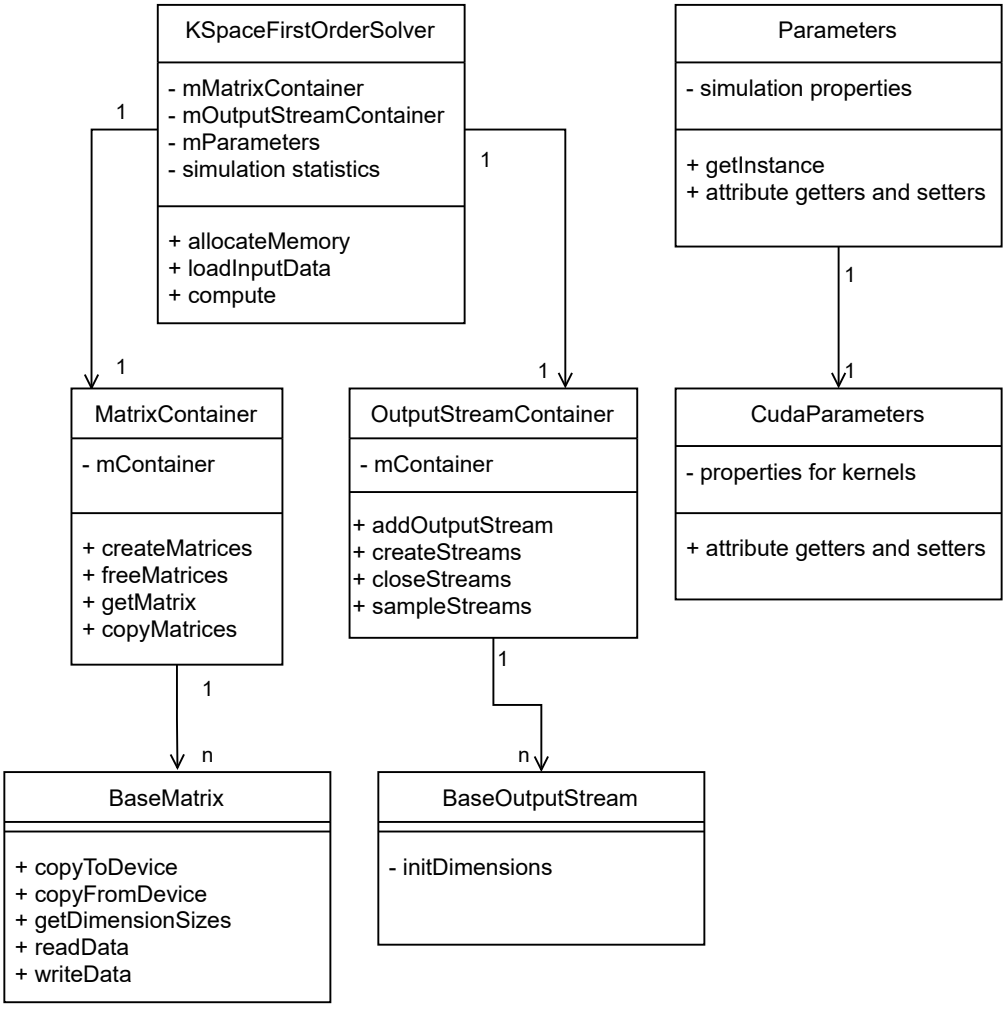
Následne je spustená príslušná implementácia simulácie pomocou metódy `KSpaceFirstOrderSolver::computeMainLoop`. Táto metóda využíva prostriedky šablón (angl. templates) jazyka C++, ktoré umožňujú definovať telo tejto funkcie pre rôzne kombinácie parametrov. Síce sa jednotlivé verzie tejto funkcie líšia podľa zvolených parametrov simulácie, je možné určiť dôležité časti tejto funkcie. V každom kroku slučky, ktorá sa nachádza v tejto funkcii sú realizované potrebné výpočty veličín (podľa zadaných parametrov sú niektoré kroky vynechávané). V závere slučky dochádza k uloženiu výstupných hodnôt v danom kroku do výstupných prúdov pomocou metódy `KSpaceFirstOrderSolver::storeSensorData`.

V tejto metóde sú navzorkované alebo skopírované aktuálne hodnoty tých veličín, ktoré sa nachádzajú zaregistrované v kontajnery `OutputStreamContainer`.

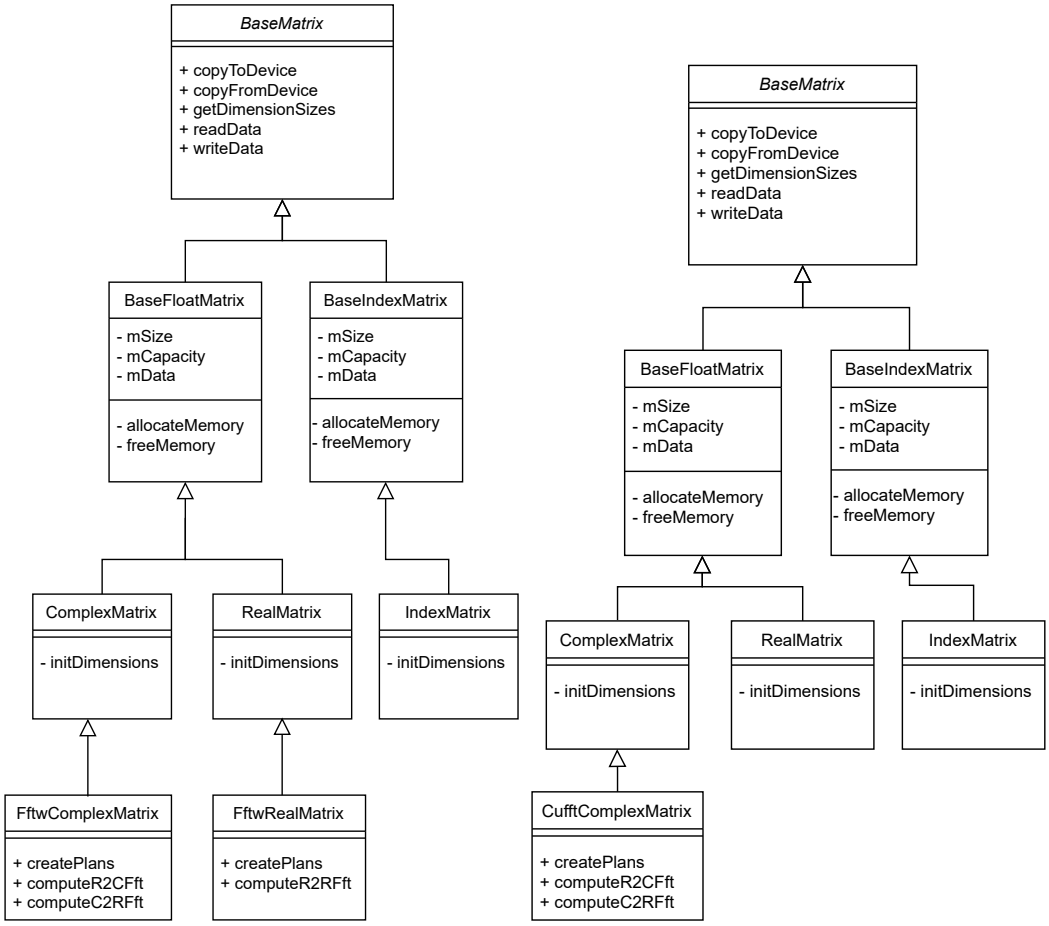
Architektúra matíc

Matice sú v spravujúcom kontajnery `MatrixContainer` alokované podľa potreby, podľa parametrov simulácie. Tento kontajner ich udržiava v polymorfnej štruktúre `std::map` pomocou objektov typu `MatrixRecord`, ktoré obsahujú ukazovateľ na samotnú maticu uchovávajúcu dáta. Tento ukazovateľ je typu `BaseMatrix*`, čím môže reprezentovať všetky typy matíc, ktoré aplikácia podporuje.

Hierarchia jednotlivých typov matíc je zobrazená na obr. 3.2. Na tomto obrázku je vidno rozdiel medzi implementáciami s OpenMP a CUDA, kde miesto `FftwComplexMatrix` vo verzii s OpenMP je na rovnakej pozícii typ `CufftComplexMatrix` vo verzii CUDA. V tejto verzii taktiež nie je prítomná ekvivalentná trieda `FftwRealMatrix`, ktorá je v OpenMP



Obr. 3.1: Architektúra hlavných tried v implementáciách šírenia vlnenia pomocou C++ s OpenMP a C++ s CUDA



Obr. 3.2: Hierarchia tried matíc v implementácii pomocou OpenMP (vľavo) a CUDA (vpravo)

verzii využívaná pri axisymetrických simuláciách. Tieto simulácie implementácia CUDA nepodporuje. Je to z toho dôvodu, že počas axisymetrickej simulácie sú využívané diskrétne kosínusové a diskrétne kosínusové transformácie, ktoré knižnica cuFFT, použitá v implementácii CUDA, neposkytuje.

Architektúra výstupných prúdov

Výstupné prúdy spravuje objekt triedy `OutputStreamContainer`, ktorý uchováva jednotlivé výstupné prúdy v štruktúre `std::map`, pomocou ukazovateľov na objekty typu `BaseOutputStream`. V rámci modulov implementovaných v OpenMP a CUDA existujú tri typy výstupných prúdov, ktoré dedia zo spoločného rodičovského typu `BaseOutputStream`:

- IndexOutputStream,
- CuboidOutputStream,
- WholeDomainOutputStream.

Každý z nich definuje spôsob, ktorým sú vzorkované veličiny, ktoré ukladajú a prípadný možný redukčný operátor, ktorý je aplikovaný pri ich vzorkovaní. Vzorkovanie prebieha vo

funkcii `OutputStreamContainer::sampleStreams`, ktorá volá rôzne implementácie vzorkovania jednotlivých metódy `BaseOutputStream::sample`.

To boli časti architektúry, ktoré sú identické medzi oboma existujúcimi implementáciami pomocou OpenMP alebo CUDA. V následujúcej časti sú rozobrané rozdiely medzi týmito implementáciami. Práve tieto rozdiely bude potrebné zahrnúť v riešení tejto práce na dosiahnutie schopnosti fungovania na CPU alebo GPU pomocou jedného zdrojového kódu. Dobrou vlastnosťou týchto implementácií je, že časti, v ktorých sa prekrývajú, tvoria veľkú časť zdrojového kódu.

Odlišnosti v architektúrach

Hlavné rozdiely medzi implementáciami v OpenMP a CUDA sú v implementáciách matíc, ktoré realizujú DFT. Tieto rozdiely sú však zjednotitelné pod jedno spoločné rozhranie, ktoré by zastrešovalo funkcionalitu DFT nad maticou komplexných dát.

CUDA implementácia používa vo výpočtových kerneloch, ktoré sú definované v rámci triedy `SolverCudaKernels`, reprezentáciu kontajnera matíc v pamäti GPU. Prítomnosť kontajnera matíc v pamäti GPU v inštancii `CudaMatrixContainer` zjednodušuje obsah kernelov. Tento kontajner obsahuje ukazovatele na prítomné matice alebo hodnotu `nullptr` v prípade, že daná matica nie je v danej konfigurácii simulácie potrebná.

V implementácii pre CPU sú výpočty vykonávané v paralelných oblastiach OpenMP, kedy v niektorých vhodných prípadoch je vnútorná slučka paralelizovaná pomocou paralelizmu SIMD. Paralelné oblasti používajú direktívy kompilátora `#pragma omp parallel for` na distribúciu iterácií medzi vlákna, na paralelizáciu pomocou SIMD sú používané direktívy `#pragma omp simd`.

Rozdiel v spôsobe výpočtu medzi CUDA kernelmi implementácie pre GPU a výpočtovými oblasťami implementácie pre CPU nie je veľký. Najväčšou prekážkou, kvôli ktorej sú CUDA kernely oddelene je ten, že výpočtové kernely CUDA vyžadujú kompiláciu v zdrojovom súbore `cu`, kompilátorom `nvcc`. Kvôli tomu sú oddelené a volané pomocou C++ funkcií, ktoré obaľujú volania kernelov. Ich obsah je veľmi podobný obsahu výpočtových oblastí používaných v OpenMP implementácii. Preto by malo byť možné použiť výpočtové oblasti z implementácie OpenMP na implementáciu výpočtových kernelov pre GPU pomocou OpenACC. K tomu napomáha aj fakt, že design programovania pomocou direktív OpenACC je príbuzný programovaniu pomocou direktív OpenMP.

Implementácie sa taktiež líšia v spôsobe, v akom je realizované vzorkovanie výstupných veličín do výstupných prúdov na GPU. CUDA implementácia využíva, kvôli zaisteniu optimálneho výkonu, tzv. zero-copy pamäť na prenos dát z pamäti GPU do pamäte hosťovského systému. Taktiež sú výstupné prúdy, ktoré neaplikujú na výstupné dáta niektorú z redukčných operácií, ukladané na disk asynchrónne, počas výpočtu ďalšieho časového kroku simulácie. Bližšie je tento spôsob vysvetlený v časti 5.7.3, popisujúcej implementáciu tejto funkcionality v rámci implementácie a optimalizácie výstupu tejto práce.

Triedy spravujúce pamäť na grafickej karte majú spravidla ukazovatele na dáta platné v rozdielnych adresných priestoroch, na „host“ a „device“ ukazovateľ, platný v pamäti hosťovského systému, resp. pamäti GPU. Prenosy medzi týmito oblasťami pamäte poskytujú funkcie `copyToDevice` a `copyFromDevice` ktoré implementujú všetky typy objektov, ktoré môžu vlastniť dáta na grafickej karte.

3.2 Testovacia sada balíka k-Wave

Sada testov obsiahnutá v balíčku k-Wave výrazne uľahčuje vývoj. Pozostáva z regresných a jednotkových testov pre implementáciu funkcionality balíčku v jazyku MATLAB. Ďalej obsahuje `kWaveTester`, tj. sadu testov na testovanie rozširujúcich implementácií pomocou C++ a OpenMP resp. CUDA. Funkčnosť a presnosť implementácií simulácie v C++ s OpenMP či C++ s CUDA je vyhodnocovaná testovacími skriptom `kwt_cpp_run_all_tests`, prípadne je možné volať podskripty na OpenMP variantu alebo CUDA variant zvlášť.

Tento modul generuje vstupné súbory HDF5 pre testovanie jednotlivých implementácií na základe pevne danej matice kombinácií parametrov. Táto matica nepokrýva všetky možné kombinácie vstupných parametrov, pretože takéto množstvo kombinácií nie je reálne otestovať. Napriek tomu poskytuje pokrytie funkcionality jednotlivých parametrov a niektorých z ich možných kombinácií, čím je minimalizovaná šanca, že prípadná chyba vo výpočte by ostala neobjavená.

Takto vygenerovaný vstupný súbor je uložený na disk a je naň zavolaný program implementujúci simuláciu pomocou OpenMP alebo CUDA. Po dobehnutí simulácie, `kWaveTester` načíta výstup simulácie, ktorý je uložený v HDF5 formáte. Následne sú vyhodnotené odchýlky výstupných veličín z výstupu alternatívnej implementácie simulácie s hodnotami simulácie v prostredí MATLAB. V rámci tohto modulu je aj určená maximálna veľkosť prípustnej chyby. Určitá chyba je prítomná vždy, pretože výpočet prebieha v desatinných číslach s pohyblivou desatinnou čiarkou. Táto aritmetika nie je asociatívna, čo znamená, že výsledky operácií sú závislé od poradia, v ktorom boli realizované. Implementácia v jazyku MATLAB prirodzene počíta v 64-bitových reálnych číslach, implementácie v C++ a OpenMP, či CUDA, využívajú 32-bitové reálne čísla reprezentované pomocou pohyblivej desatinnej čiarky. Výsledky simulácií v MATLAB sú preto presnejšie. Tieto výsledky sú aj spoľahlivejšie, pretože sú pokryté regresnými a unit testami.

V prípade, že je maximálna chyba prekročená, je daný test označený ako neúspešný v zhrnutí behu testu. Je tak možné spätne zistiť, v ktorých veličinách bola prekročená tolerovaná chyba.

Táto časť balíku k-Wave bola v rámci tejto práce rozšírená, pre podporu implementácie v jazyku C++ s využitím knižnice OpenACC. Keďže táto implementácia podporuje výpočet na procesore či grafickej karte, boli pridané testovacie skripty nie len pre dve či tri dimenzie, ale aj pre oba možné spôsoby výpočtu, CPU aj GPU. Testovací skript pre test výpočtu na CPU využíva vstupný parameter na vynútenie výpočtu na CPU, `--force-host`.

Takto rozšírená testovacia sada bola nevyhnutnou súčasťou tvorby implementácie pomocou C++ a OpenACC. Pomocou tejto sady testov bolo možné kontrolovať presnosť výpočtu po každom upravenom a pridanom kroku výpočtu.

Kapitola 4

Návrh transformácie pomocou OpenACC

Na základe znalostí získaných analýzou balíka k-Wave v kapitole 3, je v tejto kapitole predstavený, podľa mojich znalostí o balíku k-Wave, najvhodnejší spôsob transformácie kódu určeného pre procesor na kód, ktorý bude možné vykonávať aj na grafickej karte. Podrobne sú rozobrané problémy, ktoré je potrebné vyriešiť na to, aby bol kód pôvodne určený pre procesory, napísaný v C++ s využitím OpenMP na paralelizáciu, podporovaný na grafickej karte za účelom akcelerácie.

Kvôli náklonnosti numerických problémov divergovať do nezmyselných hodnôt pri čo i len malej chybe v kóde, bolo potrebné vymyslieť postup, počas ktorého bude zachovaná možnosť priebežne kontrolovať presnosť a funkčnosť algoritmu. V prípade, že vo výpočte vznikne nová chyba je tak možné nájsť jej pôvod aspoň s použitím vyraďovacej techniky. Aj preto bol zvolený značne konzervatívny prístup transformácie pôvodného kódu pre CPU, ktorý je možné zjednodušiť. Aj keď sa následovný postup osvedčil ako postup, ktorý je najviac pomalý a pracný, umožňuje priebežnú kontrolu správnosti a presnosti výpočtu. V prípade nezrovnalostí taktiež umožňuje ľahšie identifikovať miesto, kde chyba vzniká. Pri takejto transformácii je obzvlášť potrebné dbať na správnosť výpočtu minimálne medzi každým z prezentovaných krokov.

Hlavné kroky transformácie z kódu pre procesor na heterogénny kód

1. Ako počiatočný bod je vhodné začať so súčasnou verziou kódu pre procesor s použitím C++ a OpenMP.
2. Z kódu pre OpenMP je potrebné odstrániť všetky direktívy OpenMP, ktoré spôsobujú paralelný výpočet.
3. Systém je potrebné zmigrovať na nový kompilátor podporujúci heterogénne programovanie. V rámci tejto práce bol na základe kritérií v časti 2.6.2 zvolený kompilátor PGI 19.10, ktorý umožňuje akceleráciu výpočtu na GPU Nvidia.
4. Alokácie pamäti je potrebné upraviť tak, aby sa zároveň s pamäťou hosťa alokovala pamäť aj na grafickej karte. Objekty spravujúce dáta je potrebné doplniť o funkcie zabezpečujúce prenos medzi hosťom a grafickou kartou.
5. Momentálne sú dáta dostupné primárne v pamäti hosťovského systému, avšak už by malo byť možné s nimi narábať aj na grafickej karte. Je teda možné transformovať

oblasti, ktoré boli v minulosti paralelizované na procesore, na oblasti vykonávané ako výpočtové jadrá na grafickej karte (ďalej aj ako kernel z angl. kernel – jadro). Aby bola možná kontrola správneho výpočtu každého kernelu samostatne, je dobré pred takýto kernel vložiť synchronizáciu dát na grafickú kartu, resp. synchronizáciu do pamäte hosťovského systému za takýto kernel. Týmto je možné overovať funkcionalitu jednotlivých kernelov oddelene od zvyšku celého systému. V prípade, že pri transformácii nastala chyba, prejaví sa zvýšením chyby vo výstupe simulácie v porovnaní s implementáciou v MATLAB.

6. Ak je v rámci kódu použité volanie externej knižnice, je potrebné toto volanie rozvetviť na dve cesty. V prípade, že aktuálne program nebeží na akcelerátore, môže sa naďalej volať pôvodná knižnica (na príklad v rámci tejto práce sa volá procesorová verzia FFTW). V prípade, že program beží s povolenou akceleráciou na grafickej karte, je pôvodné volanie knižnice pre procesor nutné nahradiť ekvivalentným volaním pre grafickú kartu (v prípade tejto práce, volanie kernelu z knižnice cuFFT).

7. V tomto bode by mali byť všetky kroky počítané na grafickej karte, pričom sa po každom takomto kerneli synchronizujú dáta medzi pamäťou systému a grafickej karty. Správne výsledky by sa tak mali nachádzať po každom kroku aj v pamäti GPU. Je tak možné prejsť k implementácii vzorkovania a kopírovania výstupných dát do výstupných prúdov na GPU. V rámci implementácie tohto kroku je potrebné správne umiestniť prenos dát z pamäte GPU do pamäte hosťovského systému.

8. V tomto okamžiku by malo byť možné redukovať prenosy z pamäte hosťovského systému do pamäte GPU len na začiatok simulácie.

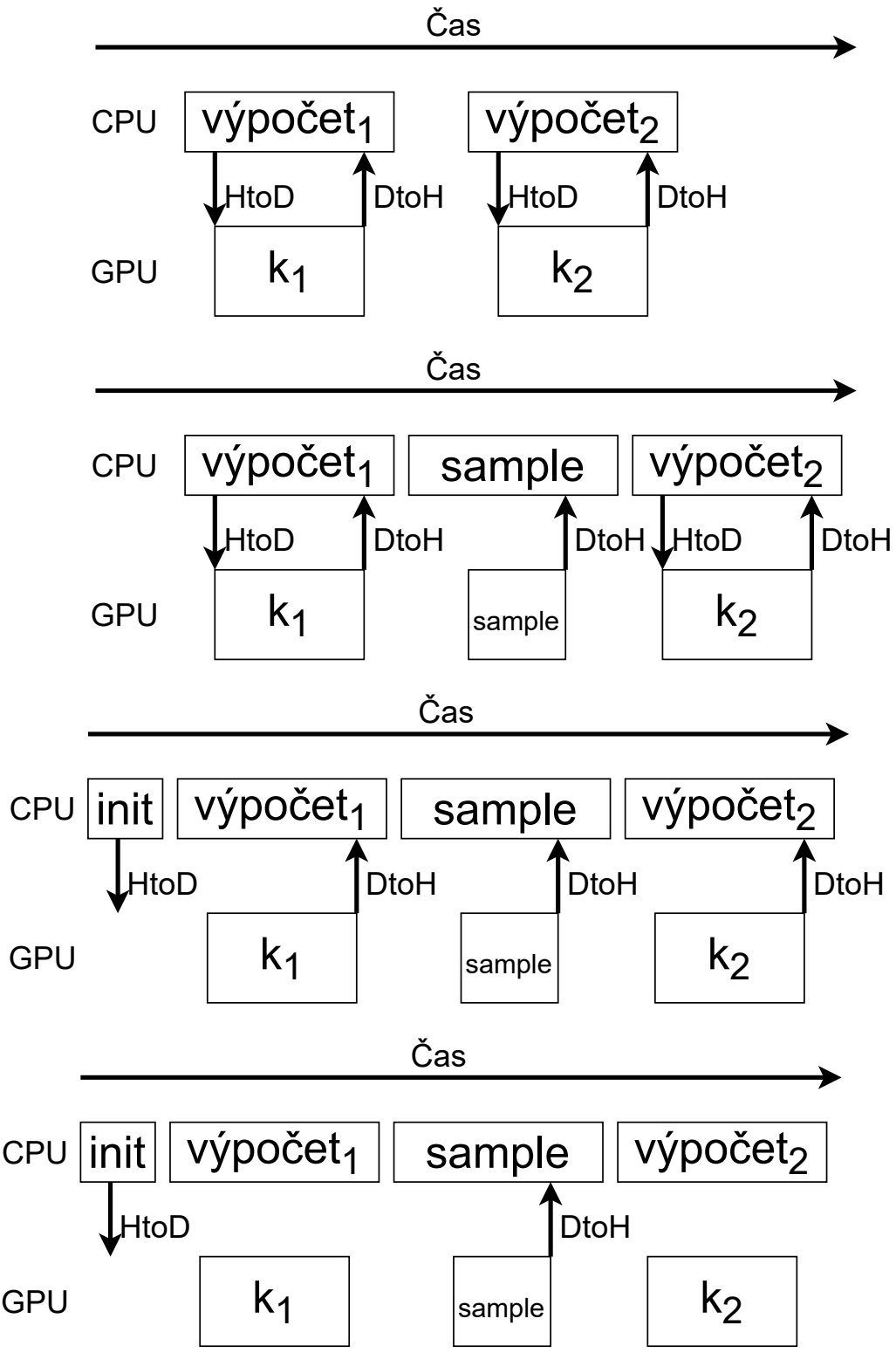
9. Zostávajúce prenosy dát z pamäte GPU do pamäte hosťovského systému po kerneloch, môžu byť taktiež odstránené.

10. Reaktivácia akcelerovaného počítania na CPU. Na začiatku transformácie boli deaktivované paralelné oblasti určené pre CPU. Tie boli neskôr prepísané do výpočtových kernelov pre GPU. V tomto kroku je potrebné znova umožniť vláknový a SIMD paralelizmus spracovania výpočtových oblastí v prípade, že výpočet neprebieha na GPU.

11. Výsledné riešenie je možné ladiť a profilovať na zaistenie optimálneho výkonu. Tento krok je potrebné opakovať, dokiaľ je po implementácii jednotlivých optimalizácii, stále dostupný priestor na vylepšenie výkonu.

12. Kozmetické úpravy transformovaného kódu.

Bod 6 je možné vhodne implementovať s využitím polymorfizmu a dedičnosti v jazyku C++, ktorá umožňuje pod jednotným rozhraním vykonávať rôzne operácie. Tento prístup bol vybraný v rámci tejto práce na zjednotenie použitia knižníc FFTW a cuFFT cez matice typov `FftwComplexMatrix` a `CufftComplexMatrix`. Tieto matice boli zjednotené pod spoločným abstraktným rozhraním `FftComplexMatrix`, ktoré zastrešuje inicializáciu vnútorne použitej knižnice a vykonávanie jednotlivých operácií DFT. Počas vývoja však bolo výhodné používať triedu `CufftComplexMatrix` ako potomka triedy `FftwComplexMatrix`. Tým bola zabezpečená funkčná implementácia zatiaľ neimplementovaných operácií DFT pomocou implementácie v rodičovskej triede. Z hľadiska architektúry však takáto hierarchia nemá opodstatnenie, preto tento vzťah bol v závere vývoja eliminovaný a nahradený riadnym dedením triedy `CufftComplexMatrix` z rozhrania `FftComplexMatrix`.



Obr. 4.1: Postupná eliminácia redundantných dátových prenosov v bodoch 7, 8 a 9.

Kapitola 5

Implementácia

Nakoľko, ako je uvedené v časti 3.1.2, sú implementácie pomocou knižnice OpenMP a CUDA celkom príbuzné, bolo možné ako počiatočný bod transformácie zvoliť poslednú verziu implementácie v OpenMP. Východzím bodom transformácie bola verzia obsiahnutá vo verzii 1.3 balíka k-Wave.

5.1 Migrácia pod PGI kompilátor

Ako prvý krok, boli odstránené všetky paralelné oblasti OpenMP, aby sa uľahčila zmena kompilátora. Tento krok sa ukázal povinný kvôli tomu, že kompilátor PGI produkoval nesprávne funkčné paralelné oblasti OpenMP. Bližšie je tento problém popísany v sekcii 5.6. Postup kompilácie definovaný v `Makefiles/Release/Makefile` bol upravený tak, aby sa objekty binárneho kódu kompilovali pomocou kompilátora PGI `pgc++`. Na prilinkovanie knižníc CUDA je použitý kompilátor CUDA `nvcc`.

Tento kompilátor kompiluje statické knižnice cuFFT a behovú knižnicu CUDA do objektu, ktorý je možné linkovať s objektami vytvorenými kompilátorom PGI, ktorý kompiluje zvyšné zdrojové súbory `cpp`. Kompilátor CUDA je tiež použitý na kompilovanie zdrojových súborov `cu`, ktoré sú použité predovšetkým pre nastavenie konštánt pre zdrojový súbor `TransposeCudaKernels.cu` a samotnú jeho kompiláciu. Tento súbor realizuje transpozície potrebné pre 1D DFT pozdĺž ôs Y a Z, ktoré sú implementované pomocou transpozície a následne realizáciou DFT pozdĺž osi X.

5.2 Zabezpečenie alokácie dát na GPU

Dáta sú dynamicky alokované v rámci troch tried: `BaseFloatMatrix`, `BaseIndexMatrix`, `BaseOutputStream`. V každej z týchto tried je potrebné vytvoriť reprezentáciu objektu v pamäti GPU. To je možné dosiahnuť pridaním direktívy OpenACC kompilátoru. Príklad pre vytvorenie reprezentácie objektu triedy `BaseFloatMatrix` v konštruktore a jeho uvoľnenie v deštruktore je vo výpise 5.1. Analogicky je realizovaná reprezentácia objektov `BaseIndexMatrix`, `BaseOutputStream`.

```
/**
 * Default constructor.
 */
BaseFloatMatrix::BaseFloatMatrix()
  : BaseMatrix(),
```

```
    mDimensionSizes(),
    mSize(0),
    mCapacity(0),
    mData(nullptr)
{
  #pragma acc enter data copyin(this)
}// end of BaseFloatMatrix
//------------------------------------------------------------------

/**
 * @brief Destructor deletes the object representation on device
 */
BaseFloatMatrix::~BaseFloatMatrix()
{
  #pragma acc exit data delete(this)
}// end of ~BaseFloatMatrix
//------------------------------------------------------------------
```

Výpis 5.1: Alokácia objektu triedy `BaseFloatMatrix`

Alokácia a uvoľnenie bloku pamäte, ktorý je použitý na uchovávanie dát je realizovaný v odlišných metódach:

- `BaseFloatMatrix::allocateMemory` a `BaseFloatMatrix::freeMemory`,
- `BaseIndexMatrix::allocateMemory` a `BaseIndexMatrix::freeMemory`,
- `BaseOutputStream::allocateMemory` a `BaseOutputStream::freeMemory`.

V týchto metódach bola preto, za alokáciou bloku pamäti na hosťovskom systéme, vložená direktíva na alokáciu rovnako veľkého bloku dát v pamäti GPU. Príklad pre triedu `BaseFloatMatrix` je uvedený vo výpise 5.2. Pre ostatné triedy je alokácia bloku pamäte realizovaná obdobne.

```
/**
 * Memory allocation based on the capacity and aligned at kDataAlignment
 */
void BaseFloatMatrix::allocateMemory()
{
  // No memory allocated before this function
  assert(mData == nullptr);

  mData = (float*) _mm_malloc(mCapacity * sizeof(float), kDataAlignment);

  if (!mData)
  {
    throw std::bad_alloc();
  }

  #pragma acc update device(this->mCapacity)
  #pragma acc enter data create(this->mData[0:this->mCapacity])
```

```
  zeroMatrix();
}// end of allocateMemory
//------------------------------------------------------------------

/**
 * Free memory.
 */
void BaseFloatMatrix::freeMemory()
{
  if (mData)
  {
    #pragma acc exit data delete(this->mData[0:mCapacity])
    _mm_free(mData);
  }

  mData = nullptr;
}// end of freeMemory
//------------------------------------------------------------------
```

Výpis 5.2: Alokácia bloku pamäte pre `BaseFloatMatrix`

Tieto objekty boli taktiež rozšírené o metódy na prenos dát medzi pamäťou hosťovského systému a pamäťou GPU. Do vyššie zmieňovaných tried boli pridané metódy `copyToDevice` a `copyFromDevice`. Príklad jednej z nich je vo výpise 5.3. Metóda `copyFromDevice` je implementovaná obdobne, s tým rozdielom, že v pragma direktíve je miesto `device` použité slovo `host` alebo `self` (ktoré sú synonymami podľa špecifikácie OpenACC [8]).

```
/**
 * Copy data from host -> device (CPU -> GPU).
 * The transfer is synchronous there is nothing to overlap with in the code
 */
void BaseFloatMatrix::copyToDevice()
{
  #pragma acc update device(this->mData[0:this->mCapacity])
}// end of copyToDevice
```

Výpis 5.3: Metóda obaľujúca prenos dát na GPU v triede `BaseFloatMatrix`

5.3 Prenos výpočtu na GPU

Keďže už sú potrebné matice alokované aj v pamäti GPU, je možné postupne začať implementovať výpočtové kernely na GPU. Problémom v tomto bode transformácie je však, že dáta na GPU nie sú aktuálne. Preto som sa rozhodol postupne implementovať výpočet jednotlivých kernelov postupne s tým, že každý výpočet bude obalený prenosom dát na GPU a následným prenosom dát výsledku z GPU. Príklad tak implementovaného kernelu je vo výpise 5.4.

```
getRealMatrix(MI::kRhoX).copyToDevice();
getRealMatrix(MI::kRhoY).copyToDevice();
```

```
if (simulationDimension == SD::k3D)
{
getRealMatrix(MI::kRhoZ).copyToDevice();
}
if (!c0ScalarFlag)
{
getRealMatrix(MI::kC2).copyToDevice();
}
#pragma acc parallel loop present(p, rhoX, rhoY, rhoZ, c2Matrix)
for (size_t i = 0; i < nElements; i++)
{
const float c2 = (c0ScalarFlag) ? c2Scalar : c2Matrix[i];

const float sumRhos = (simulationDimension == SD::k3D) ?
  (rhoX[i] + rhoY[i] + rhoZ[i])
  : (rhoX[i] + rhoY[i]);

p[i] = c2 * sumRhos;
}
getRealMatrix(MI::kP).copyFromDevice();
```

Výpis 5.4: Príklad implementovaného kernelu pre GPU v štádiu transformácie kedy väčšina výpočtov prebieha na CPU.

5.4 Počítanie DFT pomocou FFTW alebo cuFFT

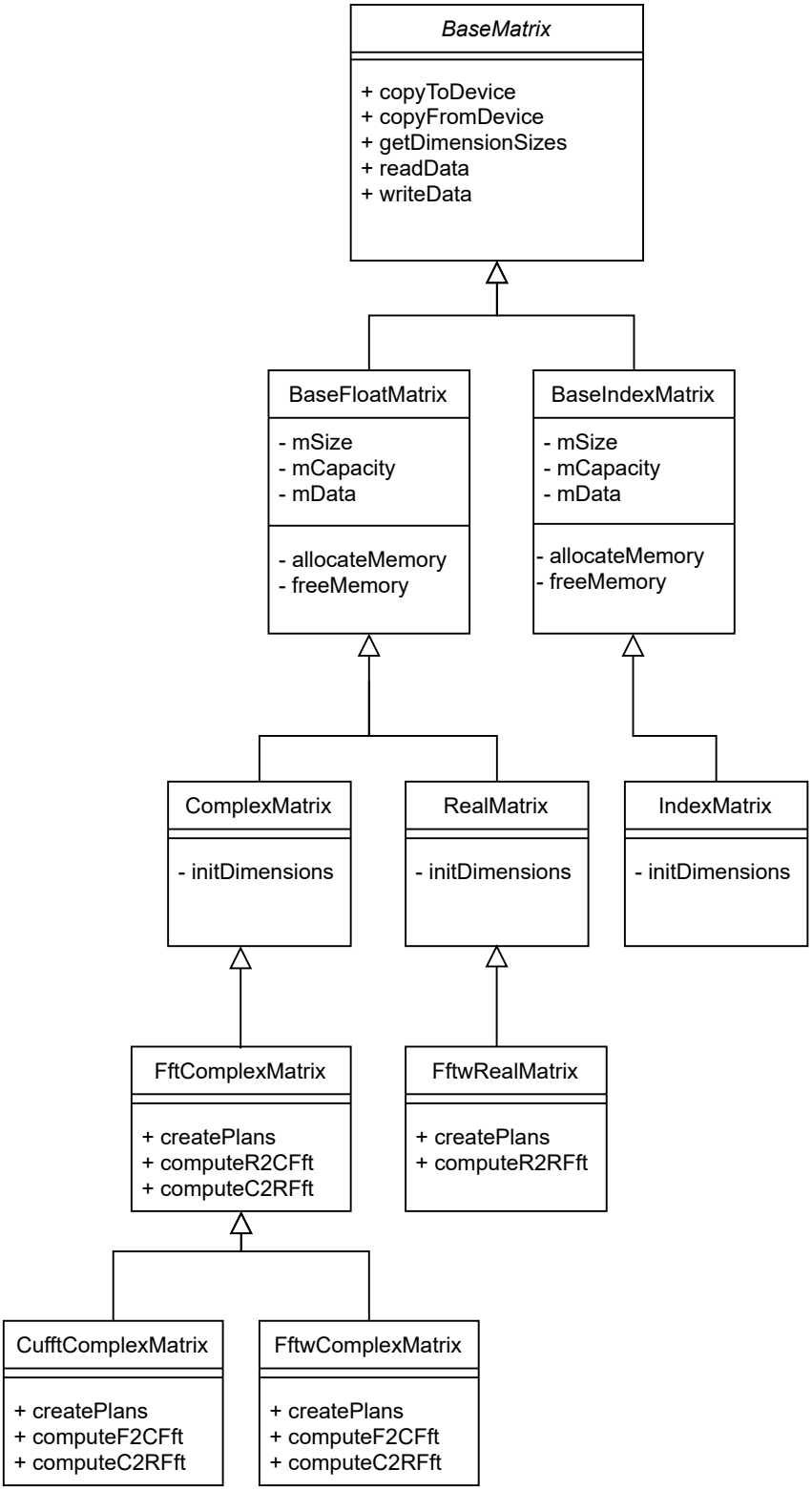
Problematika počítania DFT pomocou FFTW alebo cuFFT bola riešená s využitím dedičnosti. Bola vytvorená abstraktná trieda `FftComplexMatrix`, ktorá poskytuje zjednotené rozhranie triedy matice poskytujúcej implementáciu výpočtu DFT. Triedy `CufftComplexMatrix` a `FftwComplexMatrix` následne implementujú metódy tohto rozhrania a výber implementácie je zabezpečený automaticky za behu programu, podľa toho, aké matice sú prítomné v kontajnery matíc `MatrixContainer`. Typ matice uloženej do kontajnera je určený práve pri inicializácii kontajnera matíc.

Výber typu matice je založený na hodnote v singleton objekte parametrov `Parameters::mAccDeviceType`. V prípade, že aplikácia **nepobeží** na GPU, je hodnota tohto parametru `acc_device_host`. V tomto prípade je vytvorená matica typu `FftwComplexMatrix`. Hodnota parametru `mAccDeviceType` môže nadobudnúť hodnotu `acc_device_host` vo viacerých prípadoch:

- nie je dostupná grafická karta kompatibilná so systémom OpenACC,
- je použitý parameter `--force-host` pri spustení aplikácie,
- je požadovaná axisymetrická simulácia, ktorá nie je podporovaná na GPU.

Nová hierarchia tried matíc je ukázaná v obr. 5.1.

V prípade, že bola volaná verzia implementovaná v rámci triedy `CufftComplexMatrix`, je zavolaný kernel na dáta matice pomocou oblasti OpenACC `host_data`, ktorá sprístupní ukazovatele platné v rámci adresného priestoru pamäte GPU. Tie môžu byť poskytnuté volaniu rutiny cuFFT alebo transpozičného kernelu. Problém, s ktorým je možné sa v tomto



Obr. 5.1: Nová hierarchia tried matíc

kroku stretnúť je, že v prípade neuvedenia parametru `-Mcuda` pri kompilácii, implicitné prúdy úloh na GPU sa líšia pre CUDA kernely a kernely, ktoré sú výsledkom oblasti konštrukcie `kernels` alebo `parallel` z OpenACC. (`-Mcuda` parameter slúži pre podporu CUDA Fortran, a preto mi nepríde správne, spoliehať na tento nezdokumentovaný vedľajší efekt[1].)

Na vysporiadanie sa s týmto problémom bol pridaný parameter `mAccDefaultStream` do inštancie parameterov pre CUDA kernely `CudaParameters`, ktorý reprezentuje CUDA stream (prúd úloh, ktorý je možné si predstaviť ako FIFO front), v ktorom sú vykonávané oblasti `kernels` alebo `parallel` z OpenACC. Spustením CUDA kernelov v tomto CUDA streame je vynútená sériová exekúcia kernelov OpenACC a CUDA, pod ktoré spadajú aj kernely cuFFT. Zmeniť cieľový CUDA stream kernelu cuFFT je možné zavolaním `cufftSetStream` na stream, ktorý je možné získať zavolaním rutiny poskytovanej kompilátorom PGI, na získanie CUDA streamu z fronty OpenACC, `acc_get_cuda_stream(<acc_queue>)`.

5.5 Vzorkovanie a prenos dát z pamäte GPU

V rámci CUDA implementácie je kvôli optimalizácii výkonu vzorkovanie a ukladanie dát realizované asynchrónne. V tomto kroku je však túto funkcionalitu možné vynechať, pretože všetok výpočet je vykonávaný synchrónne. Táto optimalizácia bude vykonaná vrámci ladenia výkonnosti v rámci implementácie bodu 11.

5.6 Reaktivácia akcelerovaného výpočtu na CPU

Pôvodná implementácia pomocou OpenMP využíva dva prostriedky akcelerácie výpočtu. Vláknovy paralelizmus zavedený pomocou direktívy OpenMP `parallel` a SIMD paralelizmus zavedený pomocou direktívy OpenMP `simd`. PGI kompilátor uvádza podporu pre generovanie akcelerovaného kódu pre viacjadrové procesory v prípade nastavenia parametru `-ta=multicore`, avšak tento mód nie je možné spúšťať zároveň s generovaním akcelerovaných oblastí na GPU s nastavením `-ta=tesla`. S parametrom pre GPU je možné kombinovať len parameter `-ta=host`, ktorý generuje záložný kód pre výpočtové kernely sériovo pre CPU.

Keďže OpenMP poskytuje detailnejšiu kontrolu nad spôsobom, ktorým sú akcelerované slučky vykonávané, a je možné ich kompilovať aj pri nastavení `-ta=tesla,host`, tj. kompilácii jedného spustiteľného súboru, schopného bežať buď na CPU, alebo GPU, rozhodol som sa využiť pôvodné akcelerované oblasti pomocou OpenMP aj v implementácii pomocou OpenACC.

V počiatku popisu implementácie bol spomenutý problém s funkcionalitou OpenMP v rámci kompilátoru PGI. Tento kompilátor je primárne vyvíjaný s podporou OpenACC. Prostriedky OpenMP sú podporované, avšak len s čiastočnou podporou. Niektoré klauzule v direktívach sú napríklad ignorované. Takto je tomu napríklad v prípade direktív `parallel` a ich klauzule `if`, ktorá ma podmieňovať ich paralelné vykonávanie.

Ako je možné sa dozvedieť v užívateľskej príručke kompilátoru PGI 19.10 [6], niektoré klauzule OpenMP pragiem nie sú podporované. Klauzula `if` síce nie je spomenutá, ale jej prítomnosť vôbec nemení chovanie kódu v porovnaní s chovaním, v prípade, že je táto klauzula vynechaná. Ako alternatíva konštrukcie vo výpise 5.5, je možné podobnú slučku paralelizovať aj využitím klauzule `collapse`, ktorá spôsobí spojenie dvoch vnorených slu-

[1] https://forums.developer.nvidia.com/t/acc-wait-ignoring-cuda-default-stream/134452/2

čiek. V takom prípade, ukázanom vo výpise 5.6, nenastáva problém so súbehom a nevhodne vytvorenými vláknami kompilátorom.

```
#pragma omp parallel for schedule(static) \
  if (simulationDimension == SD::k3D)
for (size_t z = 0; z < dimensionSizes.nz; z++)
{
  #pragma omp parallel for schedule(static) \
    if (simulationDimension == SD::k2D)
  for (size_t y = 0; y < dimensionSizes.ny; y++)
  {
```

Výpis 5.5: Príklad validnej OpenMP paralelnej oblasti, pre ktorú kompilátor PGI generuje nesprávny kód

```
#pragma omp parallel for schedule(static) collapse(2)
for (size_t z = 0; z < dimensionSizes.nz; z++)
{
  for (size_t y = 0; y < dimensionSizes.ny; y++)
  {
```

Výpis 5.6: Príklad nahradých klauzúl `if` za jednu klauzulu `collapse(2)`, ktorá vedie k správnemu výsledku

Niektoré takto implementované kernely boli použité po boku transformovaných kernelov pomocou OpenACC, oddelených vetvením, podľa zariadenia, na ktorom daná simulácia beží (podľa `Parameters::mAccDeviceType`). V rámci tejto práce bol vykonaný pokus o jednotné kernely, ktoré budú v kóde umiestnené len raz a ich OpenMP či OpenACC podoba bude vytvorená preprocessorom. Bolo použité preprocesorové makro umožňujúce aplikáciu dvoch rôznych pragma direktív na jedne blok kódu. Ukážka je uvedená vo výpise 5.7. Niektoré takto realizované kernely poskytujú dostatučujúci výkon a boli ponechané. V niektorých kerneloch, najmä tých, kde je prechádzaná trojrozmerná slučka, bolo potrebné zduplikovať kernel do OpenMP a OpenACC podoby samostatne, pretože v nich bolo potrebné aplikovať explicitne SIMD paralelizmus na vnútornú slučku a vláknový paralelizmus na vonkajšiu slučku.

```
#define OMP_ACC_KERNEL(ompCond, ompPragma, accPragma, kernel) \
 if (ompCond) { _Pragma(ompPragma) kernel } \
 else { _Pragma(accPragma) kernel }

OMP_ACC_KERNEL(
  isHost,
  "omp parallel for",
  "acc parallel loop present(pVelocityMatrix, sourceInput, sourceIndex)",
  for (size_t i = 0; i < sourceSize; i++)
  {
    const size_t signalIndex = (isManyFlag) ? index2D + i : index2D;
    pVelocityMatrix[sourceIndex[i]] = sourceInput[signalIndex];
  }
)
```

Výpis 5.7: Makro usilujúce sa o zjednodušenie definícií kernelov v `KSpaceFirstOrderSolver`

5.7 Ladenie výkonu GPU verzie

Keďže v tomto štádiu ide o samostatnú verziu, ktorá realizuje celý výpočet na grafickej karte, je možné pri ladení výkonu postupovať obdobne ako pri ladení bežnej CUDA aplikácie určenej pre grafickú kartu. Hlavným nástrojom užitočným pri tomto ladení je Nvidia Visual Profiler (NVVP), ktorý poskytuje možnosť analyzovať aktivitu GPU na časovej osi. Na časovej osi sú za sebou zoradené jednotlivé kernely, kde je možné sa dozvedieť ich konkrétne názvy a základné parametre, s ktorými bežali. Kernely, ktoré využívajú najviac času sú pod časovou osou výpočtovej fronty zoradené podľa množstva času, ktoré strávili behom na GPU. Konkrétne volania kernelu je možné analyzovať hlbšie pomocou ďalších metrík, ktoré je možné odsledovať pri opätovnom behu aplikácie. Je tak možné získať prehľad o tom, čo najviac spomaľuje beh kernelu.

5.7.1 Implicitne sekvenčný výpočet

Hlavným úzkym hrdlom výpočtu bol fakt, že všetky oblasti výpočtu OpenACC `kernels` alebo `parallel` implicitne využívajú radu `acc_async_sync`, ktorý znamená, že po spustení kernelu sa beh vlákna zastaví, dokedy GPU nepošle signál o skončení behu kernelu. Až potom by bol spustený ďalší kernel. Takto medzi kernelmi vzniká nevyužitý čas, ktorý je ešte amplifikovaný latenciou grafickej karty. Lepšou alternatívou je niečo, čo robí CUDA natívne, tj. implicitný front, na ktorý sa implicitne nečaká. Na takýto účel je v OpenACC definovaná rada `acc_async_noval`, ktorá je ekvivalentná s použitím klauzule `async` bez ďalšieho parametru v direktíve OpenACC.

V tomto kroku však stále nie je kopírovanie výsledkov do výstupných prúdov implementované asynchrónne, takže síce nastalo vyššie chvíľkové vyťaženie, stále sa v prúde úloh pre GPU objavujú medzery spôsobené synchronizáciou s procesorom. Bolo však dosiahnuté asynchrónne spúšťanie kernelov, ktoré tak na seba naväzujú plynulejšie bez časových prestávok, kedy by GPU strácala čas nevykonávaním žiadnej práce medzi kernelmi. Ukážka, ako taký priebeh vyzerá na časovej osi NVVP, je na obr. 5.4.

Tým, že sa výpočet OpenACC kernelov presunul z implicitnej synchrónnej fronty `acc_async_sync` do asynchrónnej fronty `acc_async_noval`, bolo potrebné aj zmeniť frontu, do ktorej sú volané CUDA kernely na výpočet FFT v triede `CufftComplexMatrix`. Táto hodnotu je taktiež dôležité dostať do hodnoty `CudaParameters::mAccDefaultStream`, ktorá je použitá na volanie transpozičných kernelov pri výpočte jednorozmernej FFT pozdĺž ôs Y a Z.

5.7.2 Generovanie neoptimálneho kódu pre GPU od PGI

Druhým najväčším úzkym hrdlom výpočtu na GPU sa ukázali kernely využívajúce dátový typ `FloatComplex`, ktorý je aliasom pre dátový typ zo základnej knižnice C++, `std::complex<float>`. Tento dátový typ obaľuje dvojicu hodnôt type `float` a umožňuje nad nimi priamo prevádzať operácie pomocou preťažovania operátorov `operator+` a `operator+=`. Detailnou analýzou týchto kernelov a taktiež pri nahliadnutí do medziprodukčných PTX zdrojových súborov, ktoré je možné získať pri použití parametra `-Mcuda=ptxinfo`, bolo možné vyvodiť vinu nízkej výkonnosti kernelu.

Použitie tohto dátového typu v kerneli spôsobuje prudký nárast v množstve komunikácie s lokálnou pamäťou CUDA. Táto pamäť sa nachádza mimo čip GPU a zdieľa dátový prietok s globálnou pamäťou CUDA. Pre porovnanie, je v obr. 5.2 ukázané množstvo komunikácie s lokálnou pamäťou pri použití dátového typu `FloatComplex` ako vo výpise 5.8. V obr. 5.3 a výpise 5.9, je naopak ukážka behu rovnakého kernelu, v ktorom je miesto aritmetiky `FloatComplex`, použitá aritmetika základného dátoveho typu jazyka, `float`. Množstvo inštrukcií operujúcich s lokálnou pamäťou prudko stúpa a rýchlo prekonáva množstvo inštrukcií prevedených s globálnou pamäťou, ktorá je hlavnou príčinou rýchlosti kernelov počítaných v simuláciách riešených v tejto práci.

Daný graf je taktiež izolácia len dvoch takých operácií nad komplexným číslom. V skutočnosti ich v tomto kerneli bolo viac, čím bolo spôsobené obmedzenie priepustnosti globálnej pamäte. Elimináciou tohto problému sa dĺžky trvania jednotlivých kernelov vyrovnali dĺžkam zodpovedajúcim kernelom v CUDA implementácii.

```
const float kappaDivider = divider * kappa[i];
const FloatComplex cpyIfftX = ifftX[i];

const FloatComplex eKappa = cpyIfftX * kappaDivider;
const float eKappaR = eKappa.real();
const float eKappaI = eKappa.imag();
```

Výpis 5.8: Ukážka vnútra vyhodnocovaného kernelu, v tomto obrázku ide o zjednodušený spôsob použitia type `FloatComplex`.

```
const float eKappaR = ifftX[i].real() * kappa[i] * divider;
const float eKappaI = ifftX[i].imag() * kappa[i] * divider;
```

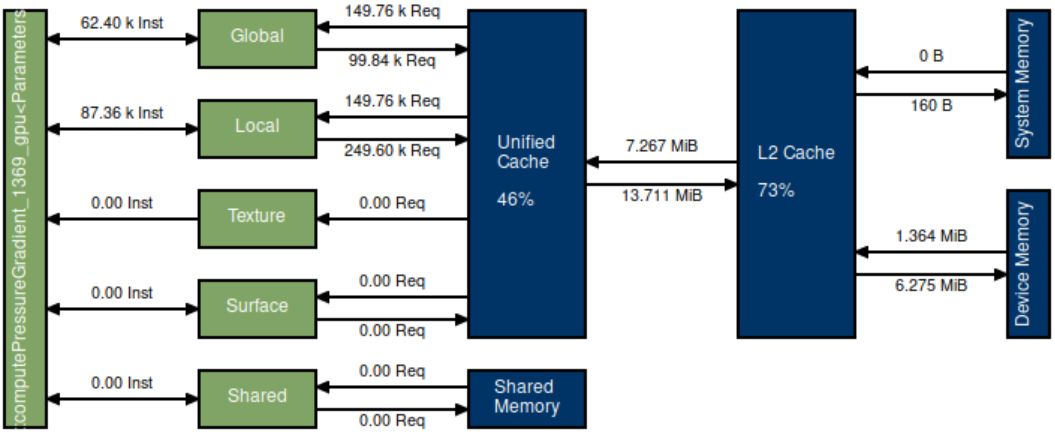
Výpis 5.9: Pojednávaný pomalý kernel napísaný pomocou aritmetiky typu `float` manuálne po zložkách.

5.7.3 Asynchrónne ukladanie výstupných veličín

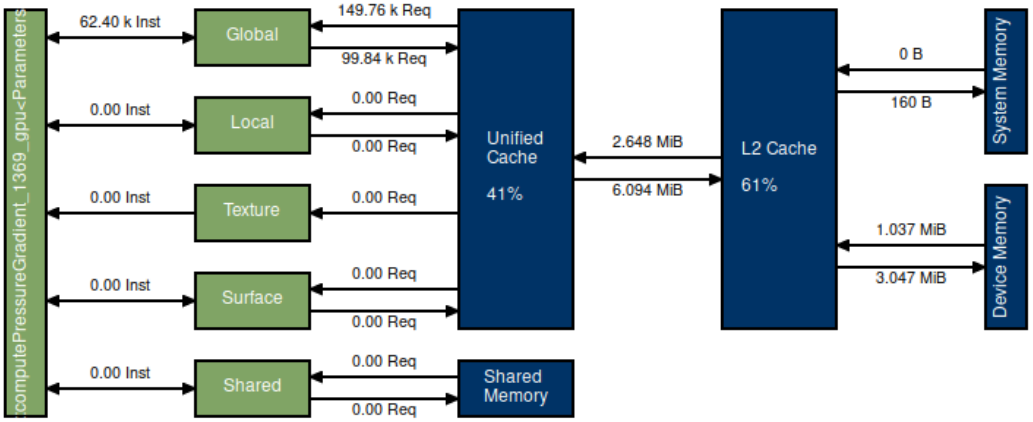
Aj napriek tomu, že jednotlivé kernely trvali podobne dlho ako v referenčnej implementácii pomocou CUDA, simulácia implementovaná pomocou C++ a OpenACC bola stále značne pomalšia. Na obr. 5.4 je možné vidieť okolie dátových prenosov z pamäti GPU do systémovej pamäte, ktoré sú vždy obklopené chvíľou neaktivity. Pre porovnanie, v obr. 5.5 nie sú žiadne zbytočné medzery obklopujúce jednotlivé iterácie simulácie, ktoré by spôsobovali jednoduchú stratu výkonu.

Táto plynulosť využitia dostupných prostriedkov GPU je získaná predovšetkým asynchrónnym ukladaním výstupných dát do súboru, v prípade, že daná doména je ukladaná bez aplikácie redukčnej operácie, v „raw“ forme (z angl. raw - surový). To funguje ako je znázornené na schéme 5.6 tak, že v kroku $t + 1$ je na disk ukladaný výsledok z kroku t. Počas ukladania výsledku kroku t zo systémovej pamäte na disk, na GPU môže prebiehať výpočet hodnôt z kroku $t + 1$.

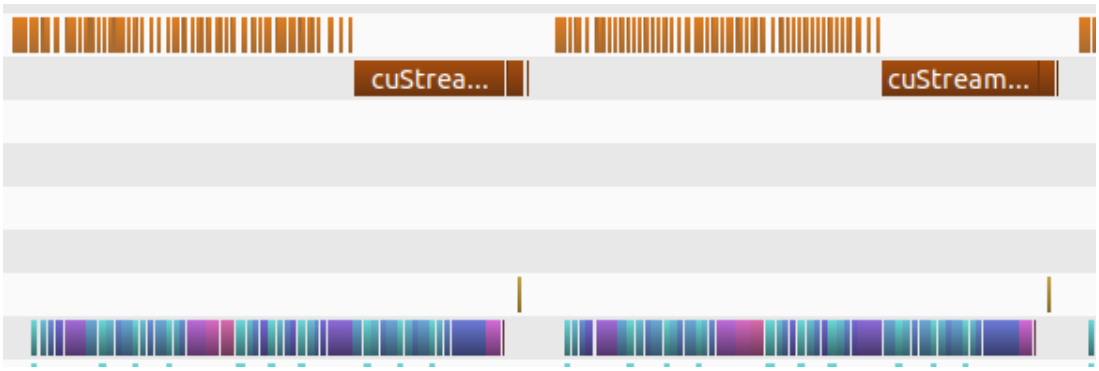
Túto implementáciu som sa snažil implementovať pomocou prostriedkov OpenACC bez nutnosti využitia volaní rutín CUDA. Narazil som však na viacero problémov, či už komplikácie spôsobované optimalizátorom kompilátora PGI 19.10, alebo implementáciou OpenACC rutín v kompilátore PGI 19.10 (ako napríklad `data update async` nie je asynch-



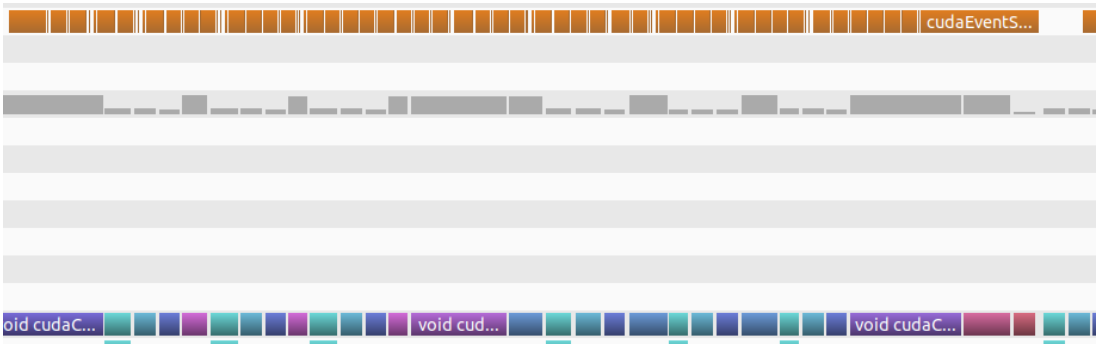
Obr. 5.2: Záznam komunikácie s pamäťou podľa typu, pri použití dátoveho typu `FloatComplex`, ktorý preťažuje operátory.



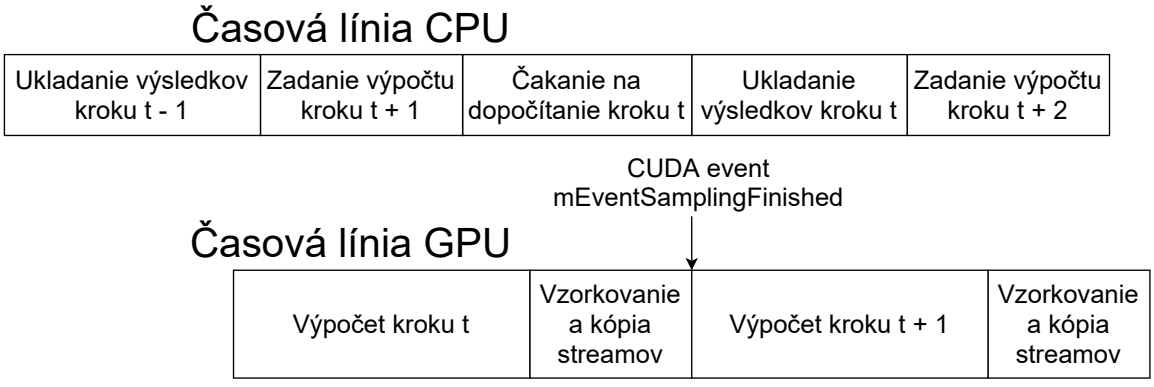
Obr. 5.3: Záznam komunikácie s pamäťou podľa typu, pri použití dátových typov `float` a rozpísaní aritmetiky komplexných čísel po zložkách.



Obr. 5.4: Ukážka neefektívne využívaného času grafickej karty, kvôli synchrónnemu ukladaniu medzivýsledkov simulácie. Na spodku obrázku sa nachádza časová línia kernelov na GPU, hore je časová línia CPU. V okolí prenosu dát neprebieha výpočet, čo má za následok nižší výkon. Tento prestoj existuje kvôli tomu, že sa najprv čaká na skončenie výpočtu, potom sa čaká na skončenie kópie, a až potom sa začína výpočet ďalšieho kroku.



Obr. 5.5: Ukážka efektívnejšie využívaného času implementáciou v CUDA, kedy okamžite po dokončení vzorkovania výstupu nasleduje výpočet FFT bez plýtvaného času.



Obr. 5.6: Schéma vysvetlujúca ukladanie výsledkov výstupných prúdov, ktoré nepoužívajú redukčnú operáciu, v implementácii CUDA.

rónna a má problémy so stránkovateľnou pamäťou[2]). Preto výsledná implementácia využíva udalosti CUDA (ďalej aj ako angl. CUDA Events) a taktiež bolo potrebné použiť CUDA rutinu kópie dát priamo volaním `cudaMemcpyAsync` v oblasti `host_data`, miesto dedikovanej pragma direktívy pre OpenACC `#pragma acc update host` s dovetkom `async`). V prípade, že by implementácia asynchrónnych front a dátových prenosov fungovala podľa špecifikácie OpenACC [8], malo by byť možné realizovať obdobnú funkcionality pomocou front OpenACC a synchronizácie medzi nimi, to však zatiaľ nie je možné.

V rámci tohto kroku bolo taktiež potrebné zaistiť, aby pamäť, určená na ukladanie výsledkov v pamäti hosťovského systému, neležala v stránkovateľnej pamäti. Ak je totiž realizovaný prenos do stránkovanej oblasti, je použitý tzv. double buffering. Vtedy je obsah dočasne skopírovaný do pomocnej pamäte v nestránkovanej pamäte, a až následne do stránkovanej pamäte. Toto chovanie vyplýva z fungovania hardware zodpovedného za prenosy dát medzi hosťom a GPU [4]. OpenACC ponúka možnosť použitia nestránkovanej, tzv. *pinned*, pamäti pomocou prepínača pri kompilácii. Nedostatkom tejto možnosti je, že všetky alokácie pamäte na hosťovskom systéme, sú realizované pomocou nestránkovateľnej/pinned pamäte. Avšak takáto pamäť je potrebná len pre výstupné prúdy. V prípade pokusu o kompiláciu a spustenie implementácie z tejto práce s parametrom `pinned`, nastane problém počas alokácie, kedy podporná knižnica OpenACC, ktorá v sebe obsahuje alokátor spravujúci nestránkovú pamäť, zlyhá, pretože má nedostatok pamäte na uloženie všetkých matíc do pinned pamäte.

Riešením je alokácia nestránkovanej pamäte manuálne, volaním CUDA runtime funkcie `cudaMallocHost`, pri alokácii pamäte pre výstupný prúd, v `BaseOutputStream::allocateMemory` ako je vo výpise 5.10. OpenACC implementácia tak môže požívať výhody rýchlosti prenosu pamäte plynúce s použitia nestránkovanej pamäte bez nutnosti ukladania všetkých dát simulácie v nestránkovanej pamäti.

```
if (Parameters::getInstance().getAccDeviceType() == acc_device_host)
{
  mStoreBuffer = (float*)_mm_malloc(mBufferSize * sizeof(float),
                                    kDataAlignment);
}
else
{
  cudaMallocHost(&mStoreBuffer, mBufferSize * sizeof(float));
}
```

Výpis 5.10: Alokácia nestránkovanej (pinned) pamäte v `BaseOutputStream::allocateMemory`.

5.8 Rozšírenie balíka v jazyku MATLAB

Popri implementácii, ako už bolo popísané v časti 3.2, bol využívaný testovací framework `kWaveTester`, ktorý je súčasťou balíka k-Wave. Preto, aby bol použiteľný na testovanie funkcionality implementácie pomocou OpenACC v tejto práci, bolo potrebné rozšíriť jeho funkcionalitu. Toto rozšírenie využilo ako začiatok už existujúce moduly pre testovanie

[2] https://forums.developer.nvidia.com/t/openacc-directive-update-does-not-work-properly/136488/4

implementácie pre CPU pomocou OpenMP a implementácie pre GPU pomocou CUDA. Pridané boli nasledovné moduly:

- `kSpaceFirstOrder2DA`,
- `kSpaceFirstOrder3DA`,
- `kSpaceFirstOrderASA`.

Tieto súbory obaľujú funkcionalitu implementácie v OpenACC podobne ako príbuzné moduly pre verzie OpenMP či CUDA, poskytujú rozhranie pre simulácie prostrediu v jazyku MATLAB. Ďalej boli pridané moduly:

- `kwt_run_acc_comparison_tests_2D_GPU`,
- `kwt_run_acc_comparison_tests_2D_CPU`,
- `kwt_run_acc_comparison_tests_3D_GPU`,
- `kwt_run_acc_comparison_tests_3D_CPU`.

Tieto moduly sú zodpovedné za prevoľanie modulu `kWaveTester` so správnymi nastaveniami. Tento modul bol rozšírený o dve ďalšie nastavenia, `options.use_acc_code` a `options.force_acc_host`, ktoré slúžia na vyvolanie správneho modulu `kSpaceFirstOrder` a prípadné vynútenie behu OpenACC na procesore, aj napriek dostupnosti grafickej karty. Tento modul bol následne rozšírený o schopnosť volať a vyhodnocovať implementáciu, vytvorenú v rámci tejto práce, podľa poskytnutých parametrov. Ako parameter na vynútenie vykonávania behu na CPU pre rozhrania `kSpaceFirstOrder2DA` a pod. v MATLAB, bol zvolený `'ForceHost'`, ktorý spôsobí spustenie podprocesu implementácie OpenACC s parametrom `--force-host`.

Kapitola 6

Vyhodnotenie výsledkov

V tejto kapitole sú diskutované dosiahnuté výsledky práce. Najprv je riešenie, ktoré bolo vytvorené počas tejto práce, vyhodnotené z hľadiska jeho výpočtovej presnosti. Ako druhé je vyhodnotená jeho výkonnosť, ktorá je porovnaná s referenčnými implementáciami pre CPU s OpenMP a GPU s CUDA. Okrem toho, táto kapitola pojednáva o limitujúcich faktoroch výkonu implementácie vytvorenej v rámci tejto práce. Diskutované sú aspekty limitujúce výkon verzie pre CPU a aspekty obmedzujúce výkon verzie pre beh na GPU. V závere kapitoly je diskutovaná pracnosť takéhoto riešenia, spolu s diskusiou uplatniteľnosti podobného prístupu, ako v tejto práci, na transformáciu ďalších modulov v balíku k-Wave.

6.1 Vyhodnotenie presnosti riešenia

Udržanie dostatočnej presnosti výpočtu bolo zabezpečené použitím testovacej sady obsiahnutej v balíku k-Wave, ktorej funkcionalita je popísaná v časti 3.2. V tejto časti je bližšie rozobraný spôsob, ktorým sa vyhodnocuje presnosť výpočtu v tejto testovacej sade. Následne je prezentovaná presnosť, ktorá je dosahovaná pomocou pôvodnej implementácie v OpenMP, pôvodnej implementácie v CUDA a presnosť implementácie vyvinutej v rámci tejto práce pomocou prostriedkov OpenACC.

Metriky, ktoré sú vyhodnocované počas testov v `kWaveTester` sú metriky L_∞ a $L_{\infty rel}$. L_∞ predstavuje maximálnu absolútnu odchýlku od referenčného výstupu, platí pre ňu: $L_\infty = max|x_{ref} - x_{test}|$, kde x_{ref} je tenzor predstavujúci referenčný výstup spočítaný simuláciou v MATLAB a x_{test} je tenzor reprezentujúci výstup simulácie v alternatívnej implementácii simulácie. Metrika $L_{\infty rel}$ vyjadruje hodnotu L_∞ vydelenú maximálnou hodnotou referenčného výstupu, teda: $L_{\infty rel} = \frac{L_\infty}{max(x_{ref})}$.

Pri testovaní implementácie pomocou OpenACC bola ako medzná hodnota zvolená hodnota $2 \cdot 10^{-5}$, ktorá je používaná aj pri testovaní implementácie pre GPU v CUDA. S touto hodnotou je porovnávaná vždy menšia metrika z dvojice metrík L_∞ a $L_{\infty rel}$. Takýto spôsob zohľadňuje možnú veľkú relatívnu chybu v prípade, že ide o malé signály. V takom prípade je použitá hodnota L_∞. U väčšiny testov a veličín je to však naopak, vyhodnocovaná je predovšetkým hodnota $L_{\infty rel}$.

Výsledné riešenie spoľahlivo dosahuje presnosť v tomto požadovanom rozmedzí. Tabuľka 6.1 zobrazuje namerané hodnoty dosahované v rámci testovania implementácií v `kWaveTester`. Hodnoty implementácie OpenACC bežiacej na CPU sú vyššie ako hodnoty pôvodnej implementácie pomocou OpenMP, ale tento rozdiel je vzhľadom na rád chyby

Implementácia	L_∞	$L_{\infty rel}$
OpenMP	0,375	$2,31 \cdot 10^{-07}$
OpenACC CPU	0,5	$3,08 \cdot 10^{-07}$
CUDA	1,625	$1,00 \cdot 10^{-06}$
OpenACC GPU	1,75	$1,08 \cdot 10^{-06}$

Tabuľka 6.1: Porovnanie chybových metrík alternatívnych implementácií simulácií.

zanedbateľný. Podobne tomu je aj v provnaní pôvodnej implementácie pomocou CUDA a implementácie pomocou OpenACC. Chyba sa prejavila až na ôsomom ráde metriky $L_{\infty rel}$.

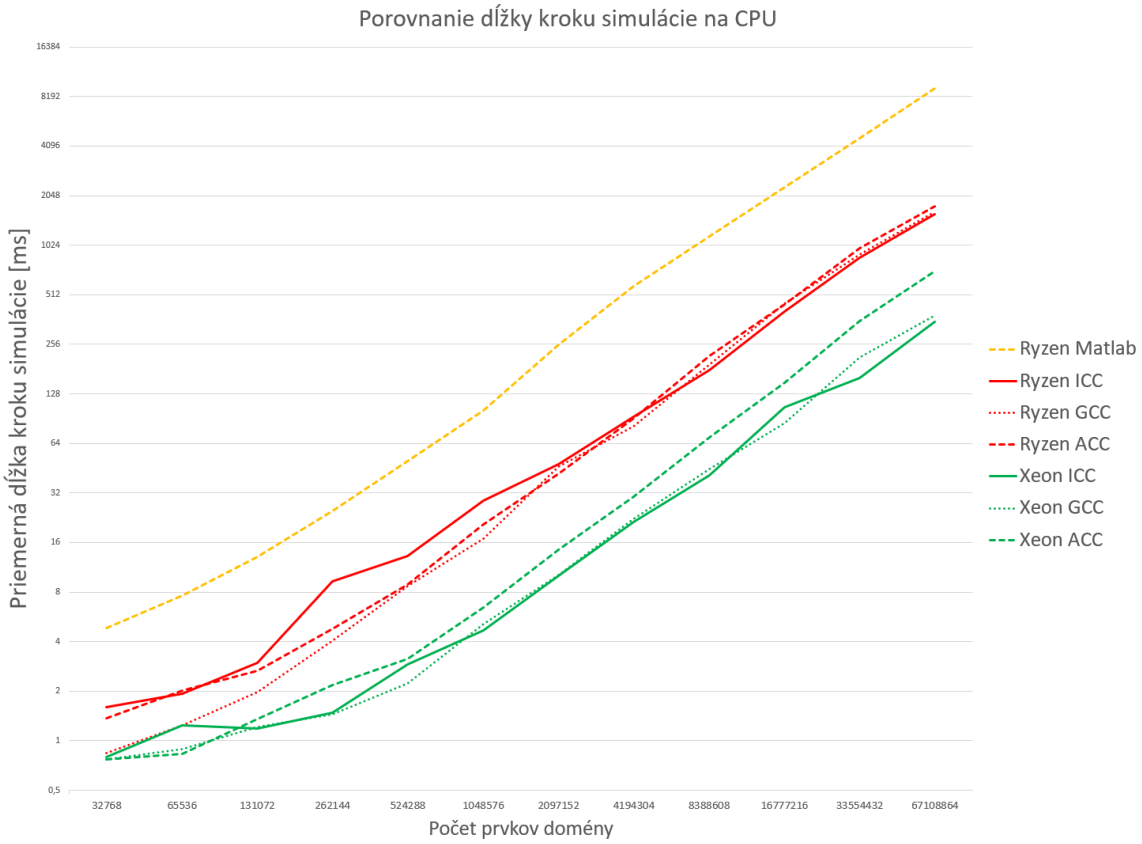
6.2 Vyhodnotenie výkonnosti riešenia

Na meranie výkonnosti bol použitý modul `benchmark` balíka k-Wave, ktorý je určený na meranie výkonnosti simulácie v MATLAB. Tento modul je možné jednoduchým rozšírením volania `kspaceFirstOrder3D` o parameter `'SaveToDisk'`, s cestou k cieľovému súboru, použiť na vygenerovanie vstupných súborov na zmeranie výkonnosti alternatívnych implementácií simulácie. Tento modul vie časovať dĺžku behu simulácie v MATLAB, ktorá je v tabuľkách A.1 a A.2 uvedená pre porovnanie. Ostatné merania alternatívnych implementácií simulácie boli realizované samostatne na vygenerovaných vstupných HDF5 súboroch. Meraná bola dĺžka behu simulačnej slučky, ktorej trvanie je obsiahnuté vo výstupnom výpise simulácie. Pre simulácie s malou doménou bol odmeraný väčší počet krokov, aby bolo možné určiť priemernú dĺžku trvania jedného kroku simulácie. Počet krokov simulácie je možné meniť parametrom `--benchmark <počet krokov>`, ktorý akceptujú všetky implementácie simulácie.

Merania výkonnosti implementácie pre CPU boli realizované na bežnom desktopovom procesore AMD Ryzen 5 3600XT, ktorý má 6 jadier s podporou hyperthreadingu, a na uzle superpočítača Barbora (IT4Innovations), ktorý má procesor Intel Xeon Cascade Lake 6240 s 18 jadrami s podporou hyperthreadingu. Získané hodnoty priemernej dĺžky kroku simulácie pre rôzne veľké domény sú v tabuľke A.1 a grafe 6.1. Použitá simulačná úloha používa heterogénne nelineárne médium s absorpciou.

Výkonnosť OpenACC implementácie pri zväčšujúcich sa doménach zaostáva za alternatívnymi implementáciami OpenMP. V tomto porovnaní sú uvedené implementácie OpenMP dve, jedna verzia pre Intel kompilátor s knižnicou MKL, druhá GCC kompilátor s knižnicou FFTW. OpenACC implementácia zaostáva pri zväčšujúcich sa doménach predovšetkým z dôvodu nevyužitia dostupného výkonu širšími SIMD inštrukciami. Kompilátor pre túto oblasť generuje len inštrukcie SSE, ktoré majú polovičnú šírku ako inštrukcie AVX, resp. štvrtinovú oproti AVX-512, ktoré podporuje procesor na počítači Barbora. Preto je rozdiel v rýchlostiach implementácií OpenACC a OpenMP výraznejší pri meraní na počítači Barbora. Na tomto počítači bol Intel kompilátor schopný vygenerovať inštrukcie AVX-512, kompilátor GCC inštrukcie sady AVX a PGI inštrukcie SSE. Preto sa časy väčších domén rozchádzajú a významne líšia.

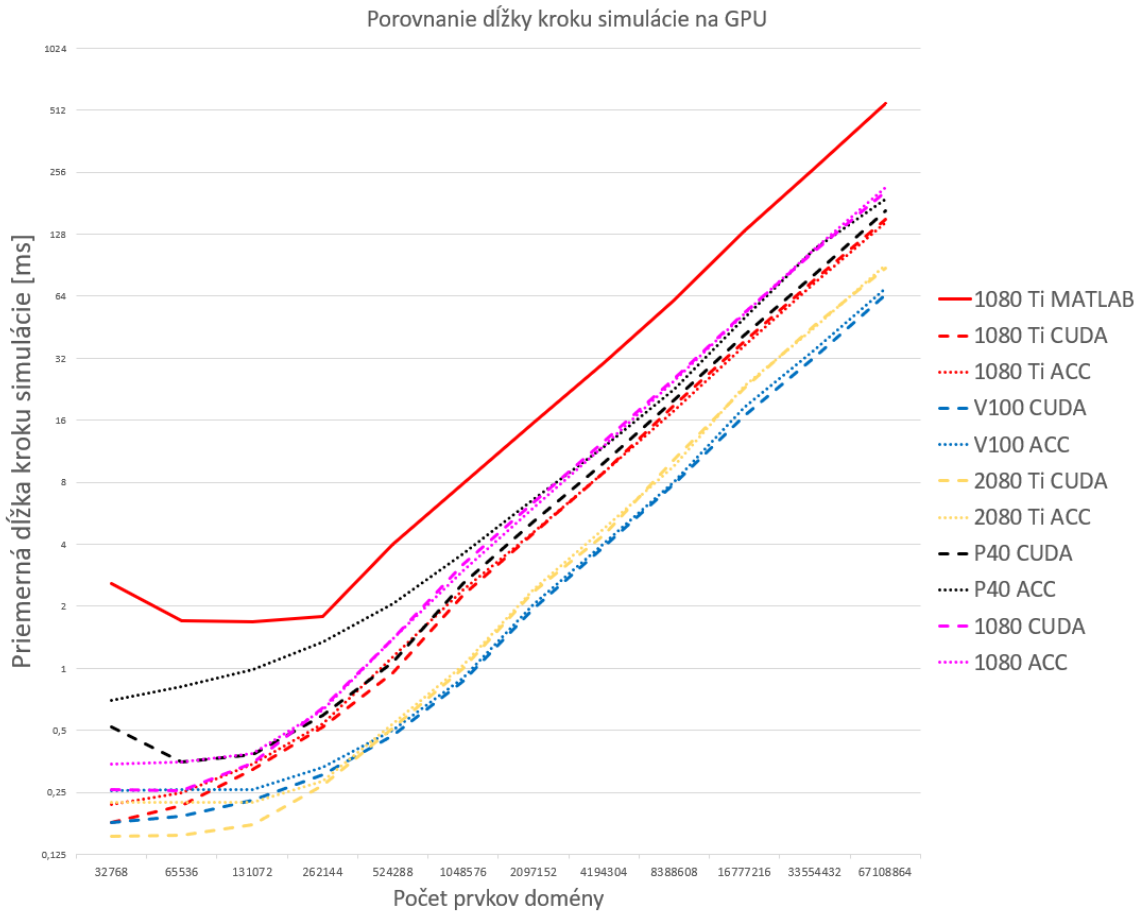
Výkonnosť v prípade behu na GPU bola meraná na piatich grafických kartách. Prvé meranie na grafickej karte pre stolné počítače, NVIDIA GeForce GTX 1080 Ti 11 GB, obsahuje aj namerané referenčné dĺžky jedného kroku výpočtu v implementácii MATLAB akcelerovanej pomocou Parallel Computing Toolbox [5]. Na tomto stroji bola implementá-



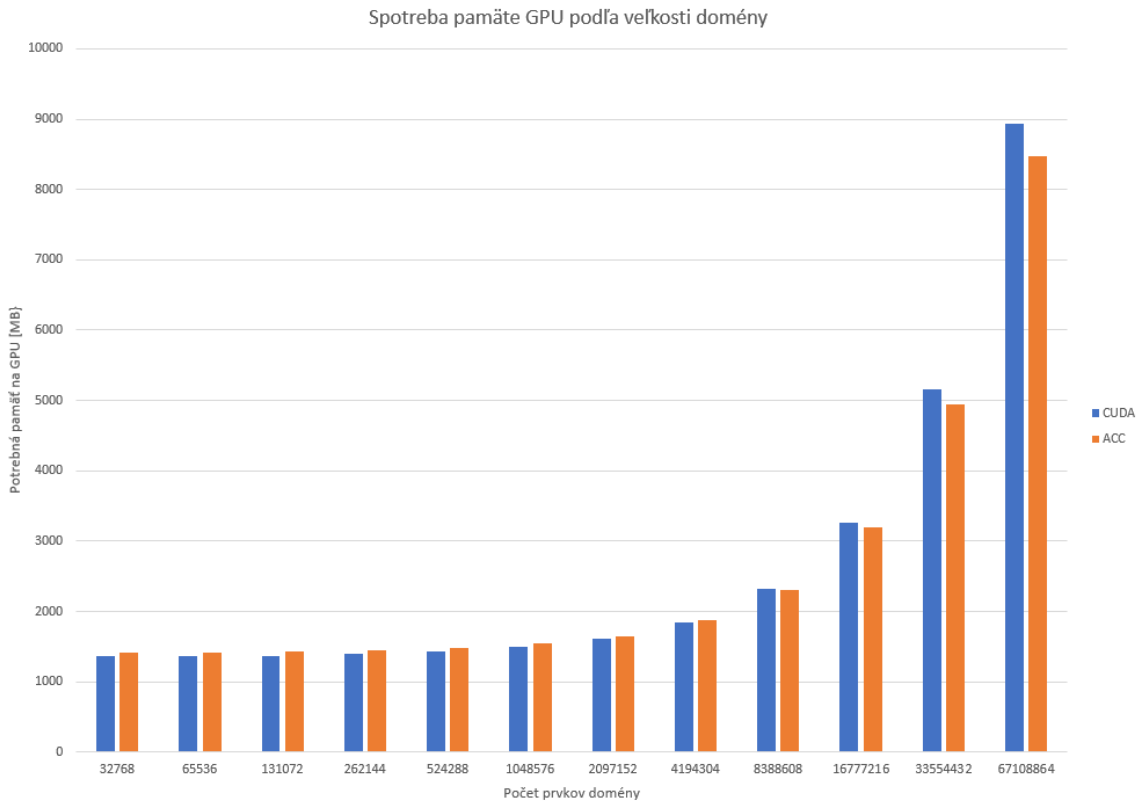
Obr. 6.1: Graf dĺžky trvania jednej iterácie simulácie na CPU. (Údaje z grafu sú v tabuľke A.1.)

cia aj profilovaná za účelom zisku maximálnej výkonnosti. Druhé meranie bolo realizované na uzle s akcelerátorom superpočítača Barbora (IT4Innovations), ktorý obsahuje najvýkonnejšiu testovanú grafickú kartu, NVIDIA Tesla V100 SMX2 16 GB. Ďalej boli merania vykonané na GPU počítači Výskumnej skupiny Superpočítačových technológií SC@FIT, ktorý je vybavený grafickými kartami NVIDIA GeForce GTX 1080 8 GB. Okrem desktopu autora a výpočtových uzlov, bol výkon taktiež odmeraný na kartách NVIDIA GeForce RTX 2080 Ti 11 GB a NVIDIA Tesla P40 24 GB. Výsledky meraní sú uvedené v tabuľkách A.2 a A.3.

Na všetkých z týchto kariet dosiahla implementácia pre GPU pomocou OpenACC a cuFFT uspokojivé výsledky. V niektorých prípadoch je možné pozorovať extrémne prepady výkonu v porovnaní pôvodnou CUDA implementáciou. Počas tejto práce som mal možnosť použiť profilovací nástroj NVIDIA Visual Profiler len na stroji s prvou testovanou kartou, pre ktorú výsledky meraní dopadli skôr pozitívne, teda opačne, ako pre ostatné karty. Nie je preto možné s istotou určiť dôvod, kvôli ktorému je viditeľný niekoľkopercentný horší výkon na grafických kartách, ktoré nie sú architektúry NVIDIA Pascal (GTX 1080 Ti, GTX 1080 a Tesla P40). Tento rozdiel vo výkone je u väčšiny kariet, s výnimkou P40, do 10 % výkonu, čo je možné považovať za úspech. Výkon na karte NVIDIA P40 je príliš kolísajúci, v porovnaní s ostatnými meraniami. Taktiež je v tomto meraní pozorovateľná anomália v časoch pôvodnej implementácie CUDA. Bez ďalších hlbších znalostí nie je možné na základe tohto merania vyvodzovať závery, ktoré by vyvrátili úspešnosť ostatných meraní.



Obr. 6.2: Graf dĺžky trvania jednej iterácie simulácie na GPU. (Údaje z grafu sú v tabuľkách A.2 a A.3.)



Obr. 6.3: Graf porovnávajúci množstvo potrebnej pamäte GPU na simuláciu zväčšujúcej sa domény medzi OpenACC a pôvodnou CUDA implementáciou.

Názov kernelu	OpenACC	OpenACC [%]	CUDA	CUDA [%]
FFT	9,24465	66,03	9,22485	62,36
Compute velocity	0,91957	6,57	1,14219	7,72
Compute density nonlinear	0,77306	5,52	0,97184	6,57
Compute pressure terms nonlinear power law	0,76597	5,47	0,96872	6,55
Sum pressure terms nonlinear power law	0,5376	3,84	0,67818	4,58
HtoD memcpy	0,51971	3,71	0,52204	3,53
Compute velocity gradient	0,50162	3,58	0,51315	3,47
Compute absorbtion term	0,38197	2,73	0,39279	2,66
Compute pressure gradient	0,35611	2,54	0,37973	2,57

Tabuľka 6.2: Porovnanie množstva času stráveného v jednotlivých kerneloch počas simulácie. (Pozn.: Nie sú uvedené všetky kernely, ale len tie, ktoré je možné považovať za podstatné.)

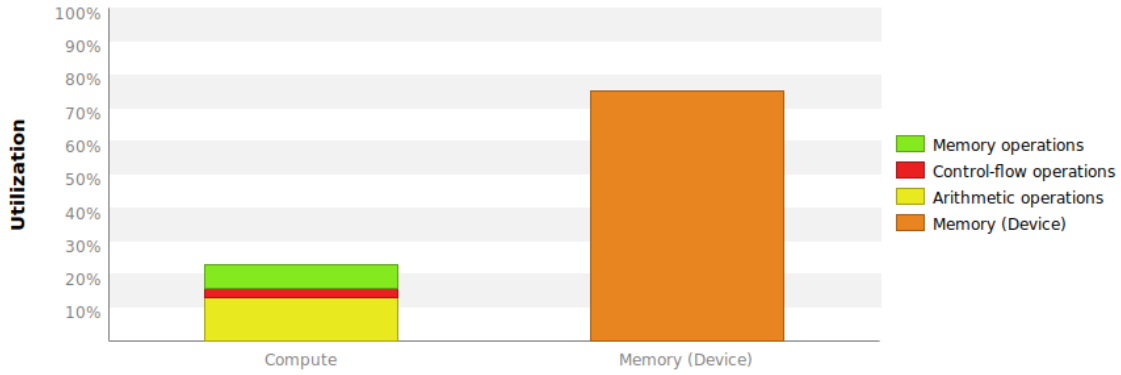
6.2.1 Analýza limitujúcich faktorov

Limitujúcim faktorom verzie pre CPU bola identifikovaná schopnosť kompilátora PGI 19.10 generovať SIMD kód pre novšie SIMD inštrukčné sady ako už staršia sada SSE. Tento záver bol vyvodený z analýzy pomocou nástroja na profilovanie Intel VTune Profiler, ktorý poskytuje možnosť odfiltrovať a identifikovať časti kódu, v ktorých je trávené najväčšie množstvo času. Miesto s vysokým podielom stráveného času bola výpočtová oblasť používajúca paralelizáciu pomocou direktív `#pragma omp parallel for` a `#pragma omp simd`. Po bližšej inšpekcii strojového kódu tejto oblasti a vzájomným porovnaním s ostatnými implementáciami, ktoré sú kompilované rozdielnymi kompilátormi, bolo možné identifikovať ako najväčší problém použitie SSE. Kompilátor Intel bol schopný identické regióny v pôvodnej implementácii OpenMP paralelizovať pomocou inštrukcií AVX-512 a kompilátor GCC pomocou inštrukcií AVX. V manuáli kompilátora je uvedený parameter na určenie veľkosti SIMD registra `-Mvect=simd:[128|256]`, ale tento parameter nemal dopad na problematické časti kódu.

Preto sa javí ako najvhodnejšie riešenie tohto problému kompilácia pomocou iného kompilátora. Má to však za následok rozdielny binárny spustiteľný súbor pre CPU optimalizovanú verziu a nutnosť testovať kompiláciu na viacerých kompilátoroch zároveň, čo zvyšuje pracnosť udržiavania takejto implementácie.

Limitujúcim faktorom GPU verzie bol rovnaký faktor ako u pôvodnej CUDA implementácie, teda výkonnosť cuFFT knižnice a priepustnosť pamäte pre ostatné operácie vykonávané v simulácii, ktoré majú nízku aritmetickú intenzitu (počet operácií na bajt). Nízka aritmetická intenzita týchto kernelov zapríčiňuje, že výkon týchto kernelov je obmedzený priepustnosťou pamäte. V tabuľke 6.2 je vidno, že až 66 % času simulácie je stráveného v kerneloch počítajúcich DFT knižnicou cuFFT, ktorá je vysoko optimalizovaná na takéto výpočty. V tejto časti nie je priestor na zlepšenie. V ostávajúcej časti je však viditeľné, že kompilátor vygeneroval efektívnejší kód kernelu definovaného pomocou OpenACC, ako pôvodná implementácia, kde je kernel definovaný v jazyku CUDA.

Kvôli tomu, že nie sú k dispozícii takto podrobné dáta pre výsledky meraní realizovaných na ostatných kartách, kvôli nemožnosti profilovať kernely GPU, nie je možné vyvodiť definitívny záver o tom, či je takýto priestor pre zlepšenie aj pri behu na ďalších kartách.



Obr. 6.4: Obrázok zobrazujúci rozdelenie záťaže počas počítania `computeVelocityUniform` kernelu.

Na týchto kartách bol naopak nameraný prírastok v dobe trvania simulácie, čo priamo neguje výsledok nameraný na domácej karte autora. Stále však platí, že výkon simulácie implementovanej v OpenACC je limitovaný predovšetkým výkonom cuFFT a schopnosťou vyťažiť maximálne pamäť v kerneloch s nízkou aritmetickou intenzitou. Toto tvrdenie je možné potvrdiť aj grafom na obr. 6.4, zobrazujúcim rozloženia záťaže počas kernelu `computeVelocityUniform`, ktorý spadá do položky „Compute velocity“ v tabuľke 6.2 (tj. ide teda o časovo najnáročnejší kernel po kerneloch cuFFT).

6.3 Porovnanie pracnosti implementácií

Zatiaľ čo sa pôvodná implementácia pre procesor pomocou OpenMP zmenila len minimálne, vo vytvorenej implementácii pomocou OpenACC sa výrazne znížilo množstvo operácií CUDA, ktoré je treba realizovať pomocou rutín CUDA. Kvôli zachovaniu vysokého výkonu bolo nutné na niektoré miesta vniesť volania CUDA, ale toto množstvo je len ojedinelé, v porovnaní s pôvodnou implementáciou CUDA. Správa parametrov simulácie, alokácia a manipulácia dát je v súčasnej implementácii zjednodušená do jedného objektového systému, ktorý je schopný sa prispôsobiť pre výpočet na GPU.

Taktiež je zjednodušené narábanie s maticami pri výpočte na GPU. V pôvodnej implementácii bolo potrebné mať kontajner matíc aj v pamäti hosťovského systému, aj v pamäti GPU. V implementácii pomocou OpenACC je využitá možnosť zjednotiť ukazovatele na dáta v pamäti hosťovského systému a ukazovatele platné v rámci pamäte GPU do jedného ukazovateľa. Preto postačuje jeden kontajner matíc, ktorý spravuje objekty matíc, ktoré obsahujú jeden ukazovateľ na dáta.

Za nepriaznivý efekt tejto implementácie je možné považovať ojedinelé problémy spôsobované optimalizátorom kompilátora, ktorý v niektorých prípadoch je schopný generovať kód, ktorý pokazí validný a funkčný kód. V iných prípadoch zase negeneruje kód, ktorý by užívateľ očakával podľa špecifikácie OpenACC. Ako príklad je možné uviesť príklady „obchádzok“, ktoré bolo v takých prípadoch nutné pridať do kódu:

- v rutine `BaseFloatMatrix::copyFromDevice` je nutné použiť izolované hodnoty ukazovateľa a veľkosti poľa a taktiež je nutne vynútiť synchronizáciu volaním rutiny OpenACC `acc_wait_all()`, miesto použitia pragmy `#pragma acc wait` (či len klauzule `wait` v direktíve kópie pamäte),

- v metóde `MatrixContainer::getMatrix` je potrebné použiť prístup do mapy pomocou `std::map::find`, miesto pôvodného `std::map::operator[]`, pretože kompilátor túto metódu znefunkčnil na úrovni optimalizácie 2 a vyššie príliš agresívnou medziprocedurálnou optimalizáciou.

Takéto riešenia zhoršujú čitateľnosť kódu a sú preto nežiaduce.

6.4 Použiteľnosť v ďalších moduloch balíku k-Wave

V balíku k-Wave existuje implementácia simulácie akustického poľa z parametricky popísaných zdrojov akustického vlnenia `acousticFieldPropagator`, ktorá by mohla byť podobným postupom rozšírená o možnosť behu na GPU. Keďže pre túto simuláciu existujú implementácie len v jazykoch MATLAB a C++ s pomocou OpenMP, je vhodným kandidátom prerobenie do podoby podporujúcej aj GPU. Je tak relatívne jednoducho možné dosiahnuť zrýchlenie v rádoch desiatok oproti implementácii OpenMP.

Je však diskutabilné, či prínos spôsobený zrýchlením implementácie pre GPU preváži negatívny dopad spomalenia, ktoré spôsobí prechod na kompilátor PGI. Toto spomalenie spôsobené nízkym výkonom CPU verzie však je možné obísť pomocou kompilácie CPU verzie rozdielnym kompilátorom. Taktiež môže byť výkonnosť CPU verzie lepšia pri použití novšej verzie kompilátora z rodiny PGI/NVC++, čo by mohlo byť predmetom ďalšej experimentácie s implementáciou vytvorenou v tejto práci.

Kapitola 7

Záver

V tejto práci bol úspešne overený postup, ktorým je možné transformovať kód určený pre procesory na kód, ktorý bude podporovať akceleráciu aj na grafickej karte. Výkonnosť takejto implementácie sa ukázala ako rovnocenná, z hľadiska výkonu, implementácii tvorenej čisto pre GPU pomocou CUDA. Výskyt CUDA kódu bol zredukovaný na ojedinelé výnimky, potrebné k dosiahnutiu výkonu porovnateľného s implementáciou čisto pomocou CUDA. Bolo dokázané, že aj s takto veľmi zredukovaným množstvom volaných CUDA kernelov a CUDA rutín, je možné sa dostatočne priblížiť k výkonu, ktorý poskytuje čistá CUDA implementácia.

Výhodou implementácie, ktorá bola vytvorená v tejto práci, je jej ľahšia udržateľnosť v synchronizácii s implementáciou pre procesory pomocou OpenMP. Nie je tak nutné udržiavať dva rozdielne projekty, ktoré vykonávajú rovnaký výpočet na rozdielnych výpočtových prostriedkoch. Úsilie potrebné na rozšírenie aktuálnej implementácie o podporu nových výpočtov na CPU i GPU je menšie, ako v prípade, že by bola podobná snaha uskutočnená v dvoch rozvetvených projektoch. Veľké množstvo kódu je predsa len zdieľaného medzi jednotlivými implementáciami a je vhodné ho udržiavať v jednom projekte.

Rozšírenie pôvodnej implementácie pre CPU však vyžadovalo prechod pod nový kompilátor, ktorý sa ukázal ako nedostatočný v schopnosti generovať optimálny kód pre procesor vo výpočtových oblastiach. Sú však dostupné východiská z tejto situácie, ako napríklad použitie iného kompilátora na kompiláciu optimálnej implementácie pre CPU. Takýto krok by vyžadoval ohradiť úseky s OpenACC a CUDA kódom direktívami podmieňujúcimi kompiláciu, ktoré by dané oblasti vylúčili z kompilácie. Taktiež bol pri tejto práci bol použitý už starnúci kompilátor z rodiny PGI verzie 19.10. V súčasnosti pre PGI/NVC++ už existuje verzia 21.3. Je možné, že najväčší problém obmedzujúci výkon na CPU bude čoskoro vyriešený na strane výrobcu kompilátora, prípadne je už vyriešený v novších verziách.

Ďalším faktorom v otázke uplatnitelnosti implementácie tejto práce je otázka podpory OpenACC. V súčasnosti už začína adopcia OpenMP aj v kompilátoroch NVC++, následníkoch kompilátorov PGI. Aktuálne je táto podpora ešte len v štádiu prvotnej beta podpory offloadingu na GPU. V prípade, kedy by v budúcnosti bola ukončená podpora OpenACC, prípadne by sa ukázali alternatívne spôsoby heterogénneho programovania ako SYCL či Intel DPC++ ako účinnejšie, implementácia z tejto práce stále umožňuje jednoduchšiu transformáciu na takýto novší systém, pretože už ide o heterogénnu aplikáciu. Tá poskytuje lepší počiatočný bod k prepisu ako holá OpenMP CPU implementácia alebo CUDA implementácia.

Literatúra

[1] Cray Inc.. *CCE 9.0.0 Compatibilities and Differences* [online]. [cit. 12.1.2021]. Dostupné z: https://pubs.cray.com/bundle/Cray_Compiling_Environment_Release_Overview_90-0619b/page/CCE_9.1_Compatibilities_and_Differences.html.

[2] Davis, J. H., Daley, C., Pophale, S., Huber, T., Chandrasekaran, S. et al. *Performance Assessment of OpenMP Compilers Targeting NVIDIA V100 GPUs.* 2020.

[3] Farber, R. *Parallel Programming with OpenACC.* 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016. ISBN 0124103979.

[4] Harris, M. *How to Optimize Data Transfers in CUDA C/C++* [online]. [cit. 17.05.2021]. Dostupné z: https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/.

[5] Mathworks. *Parallel Computing Toolbox* [online]. [cit. 17.05.2021]. Dostupné z: https://uk.mathworks.com/products/parallel-computing.html.

[6] Nvidia Corp.. *PGI version 19.10 Documentation for x86 and NVIDIA Processors* [online]. [cit. 12.1.2021]. Dostupné z: https://docs.nvidia.com/hpc-sdk/pgi-compilers/19.10/x86/index.htm.

[7] OpenACC organization. *OpenACC 1.0 Specification* [online]. 2011. [cit. 12.1.2021]. Dostupné z: https://www.openacc.org/sites/default/files/inline-files/OpenACC_1_0_specification.pdf.

[8] OpenACC organization. *OpenACC 3.0 Specification* [online]. 2011. [cit. 17.5.2021]. Dostupné z: https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC.3.0.pdf.

[9] Rupp, K. *CPU, GPU and MIC Hardware Characteristics over Time* [online]. [cit. 12.1.2021]. Dostupné z: https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/.

[10] The HDF group. *HDF5 Design specification* [online]. [cit. 17.05.2021]. Dostupné z: https://portal.hdfgroup.org/display/HDF5/Design+Specifications.

[11] The Khronos Group, Inc.. *SPIR-V - OpenGL wiki* [online]. [cit. 12.1.2021]. Dostupné z: https://www.khronos.org/opengl/wiki/SPIR-V.

[12] The Khronos Group, Inc.. *SYCL Resources* [online]. [cit. 12.1.2021]. Dostupné z: https://www.khronos.org/sycl/resources.

[13] The OpenMP ARB. *OpenMP 4.5 Specs Released* [online]. [cit. 12.1.2021]. Dostupné z: https://www.openmp.org/uncategorized/openmp-45-specs-released.

[14] Treeby, B., Cox, B. a Jaros, J. *K-Wave – A MATLAB toolbox for the time domainsimulation of acoustic wave fields – User Manual*. Manual Version 1.1. 2016.

[15] Treeby, B. E. a Cox, B. T. k-Wave: MATLAB toolbox for the simulation and reconstruction of photoacoustic wave fields. *Journal of Biomedical Optics*. SPIE. 2010, zv. 15, č. 2, s. 1 – 12. DOI: 10.1117/1.3360308. Dostupné z: https://doi.org/10.1117/1.3360308.

[16] Wilt, N. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley, 2013. ISBN 9780321809469.

Príloha A

Výsledky meraní dĺžky simulácie

A.1 Merania výkonu behu simulácie na CPU

Dimenzie domény	Počet bodov mriežky	AMD Ryzen 5 3600XT, 12 jadier			
		Matlab	ICC+MKL	GCC+FFTW	OpenACC PGI + FFTW
32 x 32 x 32	32768	4,82	1,60	0,84	1,37
64 x 32 x 32	65536	7,65	1,92	1,24	2,01
64 x 64 x 32	131072	13,06	2,96	1,98	2,66
64 x 64 x 64	262144	24,93	9,26	4,04	4,80
128 x 64 x 64	524288	50,26	13,22	8,66	8,90
128 x 128 x 64	1048576	101,49	28,66	16,88	20,68
128 x 128 x 128	2097152	253,6	47,7	46,4	42,0
256 x 128 x 128	4194304	578,9	92,8	81,8	90,7
256 x 256 x 128	8388608	1164,4	178,3	192,9	216,8
256 x 256 x 256	16777216	2292,8	404,9	451,5	447,4
512 x 256 x 256	33554432	4562,3	855,5	898,0	982,5
512 x 512 x 256	67108864	9163,9	1573,8	1629,1	1755,3

Dimenzie domény	Počet bodov mriežky	Intel Cascade Lake 6240, 36 jadier		
		ICC+MKL	GCC+FFTW	OpenACC PGI + FFTW
32 x 32 x 32	32768	0,80	0,77	0,77
64 x 32 x 32	65536	1,24	0,89	0,83
64 x 64 x 32	131072	1,19	1,21	1,36
64 x 64 x 64	262144	1,48	1,45	2,19
128 x 64 x 64	524288	2,92	2,24	3,14
128 x 128 x 64	1048576	4,68	5,12	6,46
128 x 128 x 128	2097152	10,0	10,1	14,4
256 x 128 x 128	4194304	21,4	22,4	30,3
256 x 256 x 128	8388608	40,8	44,8	69,8
256 x 256 x 256	16777216	106,4	85,4	150,3
512 x 256 x 256	33554432	160,4	212,6	355,5
512 x 512 x 256	67108864	351,5	384,9	710,0

Tabuľka A.1: Porovnanie priemernej dĺžky trvania jedného kroku výpočtu pre rôzne implementácie pre CPU. Simulovaná úloha využíva heterogénne nelineárne médium s absorpciou. Uvedená je priemerná dĺžka jedného kroku simulácie v milisekundách.

A.2 Merania výkonu behu na GPU

Dimenzie domény	Počet bodov mriežky	NVIDIA GeForce GTX 1080 Ti 11 GB		
		Matlab	CUDA	OpenACC
32 x 32 x 32	32768	2,574	0,178	0,218
64 x 32 x 32	65536	1,694	0,217	0,249
64 x 64 x 32	131072	1,680	0,322	0,344
64 x 64 x 64	262144	1,782	0,518	0,539
128 x 64 x 64	524288	3,983	0,954	1,142
128 x 128 x 64	1048576	7,876	2,276	2,396
128 x 128 x 128	2097152	15,50	4,51	4,55
256 x 128 x 128	4194304	30,49	8,96	8,96
256 x 256 x 128	8388608	61,11	18,98	17,82
256 x 256 x 256	16777216	132,58	38,70	37,00
512 x 256 x 256	33554432	267,37	77,10	74,00
512 x 512 x 256	67108864	549,91	152,00	146,70

Dimenzie domény	Počet bodov mriežky	NVIDIA Tesla V100 SMX2 16 GB	
		CUDA	OpenACC
32 x 32 x 32	32768	0,178	0,256
64 x 32 x 32	65536	0,192	0,258
64 x 64 x 32	131072	0,229	0,258
64 x 64 x 64	262144	0,304	0,331
128 x 64 x 64	524288	0,472	0,504
128 x 128 x 64	1048576	0,864	0,898
128 x 128 x 128	2097152	1,94	2,04
256 x 128 x 128	4194304	3,90	4,03
256 x 256 x 128	8388608	7,84	8,10
256 x 256 x 256	16777216	16,80	18,60
512 x 256 x 256	33554432	32,50	35,30
512 x 512 x 256	67108864	65,50	70,10

Dimenzie domény	Počet bodov mriežky	NVIDIA GeForce RTX 2080 Ti 11 GB	
		CUDA	OpenACC
32 x 32 x 32	32768	0,154	0,224
64 x 32 x 32	65536	0,155	0,223
64 x 64 x 32	131072	0,175	0,224
64 x 64 x 64	262144	0,270	0,284
128 x 64 x 64	524288	0,516	0,536
128 x 128 x 64	1048576	0,998	1,030
128 x 128 x 128	2097152	2,35	2,40
256 x 128 x 128	4194304	4,48	4,77
256 x 256 x 128	8388608	10,32	9,60
256 x 256 x 256	16777216	23,00	23,40
512 x 256 x 256	33554432	45,90	45,20
512 x 512 x 256	67108864	87,50	90,30

Tabuľka A.2: Porovnanie priemernej dĺžky trvania jedného kroku výpočtu pre rôzne implementácie pre GPU. Simulovaná úloha využíva heterogénne nelineárne médium s absorpciou. Uvedená je priemerná dĺžka jedného kroku simulácie v milisekundách. Merania 1 až 3.

Dimenzie domény	Počet bodov mriežky	NVIDIA Tesla P40 24 GB	
		CUDA	OpenACC
32 x 32 x 32	32768	0,522	0,702
64 x 32 x 32	65536	0,350	0,815
64 x 64 x 32	131072	0,380	0,991
64 x 64 x 64	262144	0,593	1,343
128 x 64 x 64	524288	1,076	2,064
128 x 128 x 64	1048576	2,570	3,578
128 x 128 x 128	2097152	5,13	6,60
256 x 128 x 128	4194304	10,02	11,93
256 x 256 x 128	8388608	20,08	22,70
256 x 256 x 256	16777216	41,30	50,06
512 x 256 x 256	33554432	81,90	108,90
512 x 512 x 256	67108864	165,60	188,70
Dimenzie domény	Počet bodov mriežky	NVIDIA GeForce GTX 1080	
		CUDA	OpenACC
32 x 32 x 32	32768	0,259	0,342
64 x 32 x 32	65536	0,255	0,350
64 x 64 x 32	131072	0,346	0,385
64 x 64 x 64	262144	0,643	0,631
128 x 64 x 64	524288	1,394	1,394
128 x 128 x 64	1048576	3,198	2,978
128 x 128 x 128	2097152	6,36	5,98
256 x 128 x 128	4194304	12,72	12,04
256 x 256 x 128	8388608	25,64	24,90
256 x 256 x 256	16777216	53,40	52,80
512 x 256 x 256	33554432	105,80	108,40
512 x 512 x 256	67108864	205,70	214,80

Tabuľka A.3: Porovnanie priemernej dĺžky trvania jedného kroku výpočtu pre rôzne implementácie pre GPU. Simulovaná úloha využíva heterogénne nelineárne médium s absorpciou. Uvedená je priemerná dĺžka jedného kroku simulácie v milisekundách. Merania 4 a 5.