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 MULTI-GPU SYSTÉMŮ

ACCELERATION OF ULTRASOUND NEUROSTIMULATION USING MULTI-GPU SYSTEMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. DAVID BAYER**
AUTHOR

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

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

144974

Ústav:	Ústav počítačových systémů (UPSY)
Student:	**Bayer David, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Superpočítání
Název:	**Akcelerace ultrazvukové neurostimulace pomocí multi-GPU systémů**
Kategorie:	Paralelní a distribuované výpočty
Akademický rok:	2022/23

Zadání:

1. Seznamte se s architekturou výpočetních systémů založených na několika kartách Nvidia.
2. Prostudujte softwarové prostředky pro akceleraci náročných výpočtů na těchto systémech, zaměřte se především na distribuovanou Fourierovu transformaci.
3. Prostudujte současnou GPU implementaci balíku k-Wave.
4. Navrhněte distribuovanou implementaci k-Wave s využitím unifikované paměti a distribuované Fourierovy transformace.
5. Navržené řešení implementujte.
6. Na sadě testovacích úloh z oblasti ultrazvukové neurostimulace ověřte dosažené zrychlení, škálování a přesnost výpočtu.
7. Zhodnoťte dosažené výsledky a diskutujte přínos práce pro další směrování vývoje balíku k-Wave.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 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:	1.11.2022
Termín pro odevzdání:	17.5.2023
Datum schválení:	9.5.2023

Abstrakt

Tato diplomová práce se věnuje rozšíření akcelerované implementace simulace šíření akustických vln v médiu balíku *k-Wave* o možnost využití více GPU pro výpočet. Nejprve popisuje multi-GPU systémy a nástroje, pomocí kterých je s nimi možné pracovat. Pokračuje popisem balíku *k-Wave* a analýzou existujících akcelerovaných implementací. Dále testuje vybrané technologie na simulaci šíření tepla v médiu a na základě zjištěných výsledků vybírá nástroje pro návrh výsledné implementace. Nakonec shrnuje dosažené výsledky.

Abstract

This theses is focused on extending the accelerated implementation of propagating acoustic waves in a medium simulation of *k-Wave* toolbox by the possibility of using multiple GPUs for the computation. It first describes multi-GPU systems in general and the tools that can be used to work with them. It continues with a description of the *k-Wave* toolbox and an analysis of existing accelerated implementations. Selected technologies are then tested on a heat diffusion in a medium simulation and the results are used to select tools for the design a resulting implementation. Finally, it summarizes the results obtained.

Klíčová slova

HPC, k-Wave, multi-GPU, paralelní FFT, CUDA/C++, MPI, OpenMP

Keywords

HPC, k-Wave, multi-GPU, parallel FFT, CUDA/C++, MPI, OpenMP

Citace

BAYER, David. *Akcelerace ultrazvukové neurostimulace pomocí multi-GPU systémů*. Brno, 2023. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. Jiří Jaroš, Ph.D.

Akcelerace ultrazvukové neurostimulace pomocí multi-GPU systémů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana docenta Jiřího Jaroše. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
David Bayer
23. května 2023

Poděkování

Rád bych tímto poděkoval panu docentu Jarošovi za vedení, ochotu a čas, který mi při psaní této práce věnoval. Rovněž bych také poděkoval i své rodině a kamarádům za jejich podporu. V neposlední řadě patří dík také IT4Innovations za poskytnutí přístupu k harware pro realizaci této práce.

Tato práce byla podpořena Ministerstvem školství, mládeže a tělovýchovy České republiky prostřednictvím e-INFRA CZ (ID:90254).

Obsah

Kapitola 1

Úvod

Využití GPU pro všeobecné použití se v dnešní době stává stále populárnější. Nabízí vysoký výpočetní výkon při nízké spotřebě elektrické energie a ceně výpočtu. Proto se stále častěji vývojáři snaží přesunout výpočet úloh vhodných pro paralelní zpracování právě na GPU.

Problém ovšem nastává v případech, v kterých má úloha příliš vysoké paměťové nároky. Aktuálně se velikosti pamětí u běžných GPU pohybují kolem 10-16 GB, u profesionálních GPU se velikost dostává až na 80 GB. Pro mnoho úloh je to dostačující, nicméně stejně tak, jako se v průběhu času velikosti pamětí zvětšují, zvyšují se rovněž požadavky na aplikace, například na rozměry simulací. Tato práce se zabývá jedním z takovýchto případů.

Balík *k-Wave* je nástrojem pro simulaci šíření akustických vln v médiu určeným pro simulace z oblasti průchodu ultrazvukových vln komplexním prostředí, jako jsou živé tkáně. Původní implementace byla vytvořena pro MATLAB, nicméně pro vyšší výpočetní výkon byly doc. Jiřím Jarošem vytvořeny vysokovýkonnostní implementace v jazyce C++ pro výpočet na jednom nebo více CPU a pro výpočet na jednom NVIDIA GPU. Hlavní výhodou přesunutí výpočtu na GPU je snížení výpočetního času a ceny výpočtu, avšak jejím úskalím je omezená velikost simulační domény. Například pro výpočet složitější 3D simulace na GPU s 40 GB paměti by byla velikost hrany simulační domény omezena na 700 bodů. Existuje však několik způsobů, jak provádět i větší simulace s využitím GPU. Jsou jimi

- ukládání hodnot s nižší přesností — tím se ale výrazně sníží přesnost výpočtu,
- ukládání hodnot, se kterými se právě nepočítá, do systémové paměti — to ale extrémně sníží výkon kvůli nižší propustnosti mezi systémovou pamětí a GPU a
- distribuování výpočtu mezi více GPU.

Tato práce se zabývá akcelerací výpočtu simulace šíření akutstických vln v médiu s využitím multi-GPU systémů založených na několika NVIDIA GPU. Nejprve v kapitole 2 popisuje architekturu multi-GPU systémů a představuje konkrétní systémy, které budou použity pro měření a testování. Kapitola 3 je věnována způsobu programování multi-GPU systémů. Popisuje základní programovácí techniky a představuje řadu knihoven pro práci s nimi. V kapitole 4 je detailněji představen balík *k-Wave*. Popisuje matematické principy, na kterých simulace funguje, a analyzuje existující C++ implementace. Kapitola 5 testuje vybrané technologie a principy na jednodušší simulaci z *k-Wave*. V kapitole 6 je představen návrh a multi-GPU implementace výsledného programu. Nakonec jsou v kapitole 7 popsány dosažené výsledky.

Kapitola 2

Multi-GPU systém

V posledních letech se stále více rozvíjí úloha GPU jako obecného akcelerátoru (GPGPU) a to hlavně díky jeho vysokému výpočetnímu výkonu a energetické úspornosti. [13] Rovněž se stávají čím dál běžnějšími systémy s více GPU, jelikož se zlepšuje jejich škálovatelnost. Tato kapitola se zabývá multi-GPU systémy s grafickými kartami NVIDIA. Popisuje jejich strukturu a způsoby komunikace.

Multi-GPU systém je počítač nebo více počítačů osazený dvěma a více grafickými kartami. Ty jsou mezi sebou projeny pomocí systémové sběrnice, síťového rozhraní nebo jiných propojení. Takovýto systém umožňuje rozdělit úlohu mezi více GPU s cílem dosáhnout vyššího zrychlení nebo většího množství dostupné paměti pro výpočet.

Ideální úlohy pro GPU jsou masivně paralelizovatelné s vysokou aritmetickou intenzitou a opakovaným využitím dat. Vyššího výkonu je také dosaženo pro pravidelné úlohy, u nichž lze lépe předvídat, jakým způsobem program poběží a podle toho je také optimalizovat. Příkladem může být vektorové násobení nebo transpozice matice. Nepravidelné algoritmy jsou na paralelních architekturách náročnější na implementaci a často způsobují nízkou účinnost využití zdrojů. Takovýmito algoritmy mohou být například vytváření nebo průchody stromovými a grafovými datovými strukturami.

Nepravidelnost algoritmu může být popsána kombinací *Control Flow Irregularity* (CFI) a *Memory-Access Irregularity* (MAI). CFI je dáno divergencí toku programu, která je závislá na datech, jenž nejsou známa v době kompilace. Mohou to být např. podmíněné příkazy nebo smyčky. MAI popisuje nepravidelnost přístupu do paměti na základě jiných dat. Obě tyto neregularity zvyšují datové závislosti a je třeba jim věnovat zvýšenou pozornost při implementaci algoritmu.

Přímý přístup do paměti (*direct memory access*, DMA) je jedním z třech hlavních způsobů, jak provádět přenos dat mezi dvěma zařízeními — ostatními jsou *polling* (aktivní dotazování) a přerušení. [21] Jeho hlavní výhodou je, že přenos nikterak nezatěžuje procesor, protože o přenosy dat se stará specializovaný hardware. Také snižuje latenci přístupu k datům a počet přístupů do paměti. Rovněž není nutné vytvářet dočasné buffery pro odesílaná nebo přijímaná data. DMA se používá hlavně pro vstup/výstupní operace a pro vysokorychlostní komunikaci např. s GPU.

Vzdálený přímý přístup do paměti (*remote direct memory access*, RDMA) je rozšíření DMA o vzdálený přístup přes síťové rozhraní. [23] Používá se v případech, kdy je třeba přenášet velké množství dat po síti s nízkou latencí. Typicky bývá zakomponován v architektuře výpočetních nebo datových centrech.

2.1 Technologie propojení

Nejslabším místem GPU a multi GPU systémů je komunikace mezi GPU a CPU nebo mezi dvěma GPU. [22] Technologie propojení se snaží o co největší propustnost a nejnižší latenci komunikace mezi dvěma koncovými zařízeními.

Systémová sběrnice

V průběhu vývoje grafických karet existovala řada sběrnic pro jejich připojení. V roce 2003 vzniklo rozhraní PCI Express (PCIe), založené na svých předchůdcích PCI a PCI-X, které se používá dodnes pro vysokorychlostní připojení rozšiřujících karet. Je definováno již 6 verzí a sedmá je aktuálně ve vývoji. Každá nová verze (s výjimkou verze 3) zdvojnásobuje šířku pásma (tabulka 2.1). Všechny verze jsou zpětně i dopředně kompatibilní, což umožňuje připojovat nový hardware do starších systémů i starší hardware do nových.

PCIe je sériovou sběrnicí pro *point to point* komunikaci skládající se z linek, které jsou exkluzivně vyhrazeny pro komunikaci mezi *root complex* a koncovým zařízením. Linka může obsahovat 1, 2, 4, 8, 12, 16 nebo 32 *full-duplex* spojení (označované jako x1 až x32). V praxi se však x12 a x32 nepoužívají. Celkový počet linek pak závisí na modelu CPU a čipové sadě základní desky počítače.

Topologie PCIe má jednoduchou stromovou strukturu. Neumožňuje smyčky ani jiné komplexní topologie. Kořenem topologie je *root complex*, kterým se označuje skupina komponent mezi CPU a PCIe rozhraním. Může obsahovat několik čipů, rozhraní procesoru nebo rozhraní systémové paměti. Topologie se dále může skládat z přepínačů, mostů nebo koncových zařízení. Přepínače poskytují *fanout* nebo agregační schopnosti. Umožňují připojit více zařízení k jednomu PCIe portu a směrují přeposílané pakety. Mosty poskytují rozhraní ostatním sběrnicím, jako jsou PCI nebo PCI-X.

Verze a rok	Šířka pásma jednoho spojení
1.0 (2003)	0,250 GB/s
2.0 (2007)	0,500 GB/s
3.0 (2010)	0,985 GB/s
4.0 (2017)	1,969 GB/s
5.0 (2019)	3,938 GB/s
6.0 (2022)	7,563 GB/s
7.0 (2025, plánováno)	15,125 GB/s

Tabulka 2.1: Srovnání šířky pásma jednotlivých generací PCI Express.

NVLink

NVLink je prvním vysokorychlostím propojení GPU. [26] Umožňuje obousměrnou *point to point* komunikaci mezi GPU i CPU. Poskytuje přímý přístup do jejich pamětí pro čtení, zápis i atomické operace. Aby bylo možné tuto technologii využívat, je třeba, aby byly

NVLink sběrnice grafických karet mezi sebou fyzicky propojeny NVLink můstkem, nebo aby GPU byly připojeny k základní desce pomocí SXM soketů.

NVLink sběrnice je rozdělena na několik linek. [17] Každá z nich obsahuje dvě podlinky, každou pro jeden směr komunikace. Podlinka se dále skládá z 8 (architektury *Pascal* a *Volta*) nebo 4 vodičů (architektury *Ampere* a *Hopper*). Jednotlivé linky si lze představit jako kabely propojující dvě GPU. Čím více linek povede mezi dvěma GPU, tím větší bude šířka pásma mezi nimi. Pro dosažení maximální šířky pásma mezi dvěma GPU je tedy nutné propojit všechny jejich linky. Reálná propustnost linky je silně ovlivněna velikostí zasílaných paketů. Jeden paket může mít velikost až 256 B. Čím větší pakety jsou posílány, tím vyšší je výsledná propustnost. Tabulka 2.2 ukazuje rozdíly ve vlastnostech a šířkách pásem mezi jednotlivými generacemi NVLink.

	1. generace (*Pascal*, 2016)	2. generace (*Volta*, 2017)	3. generace (*Ampere*, 2020)	4. generace (*Hopper*, 2022)
Počet linek	4	6	12	18
Počet vodičů podlinky	8	8	4	4
Šířka pásma vodiče	20 Gb/s	25 Gb/s	50 Gb/s	50 Gb/s
Šířka pásma linky	40 GB/s	50 GB/s	50 GB/s	50 GB/s
Celková šířka pásma	160 GB/s	300 GB/s	600 GB/s	900 GB/s

Tabulka 2.2: Srovnání jednotlivých generací NVLink. Šířka pásma linky a celková šířka pásma je maximální teoretická hodnota oboustranné komunikace.

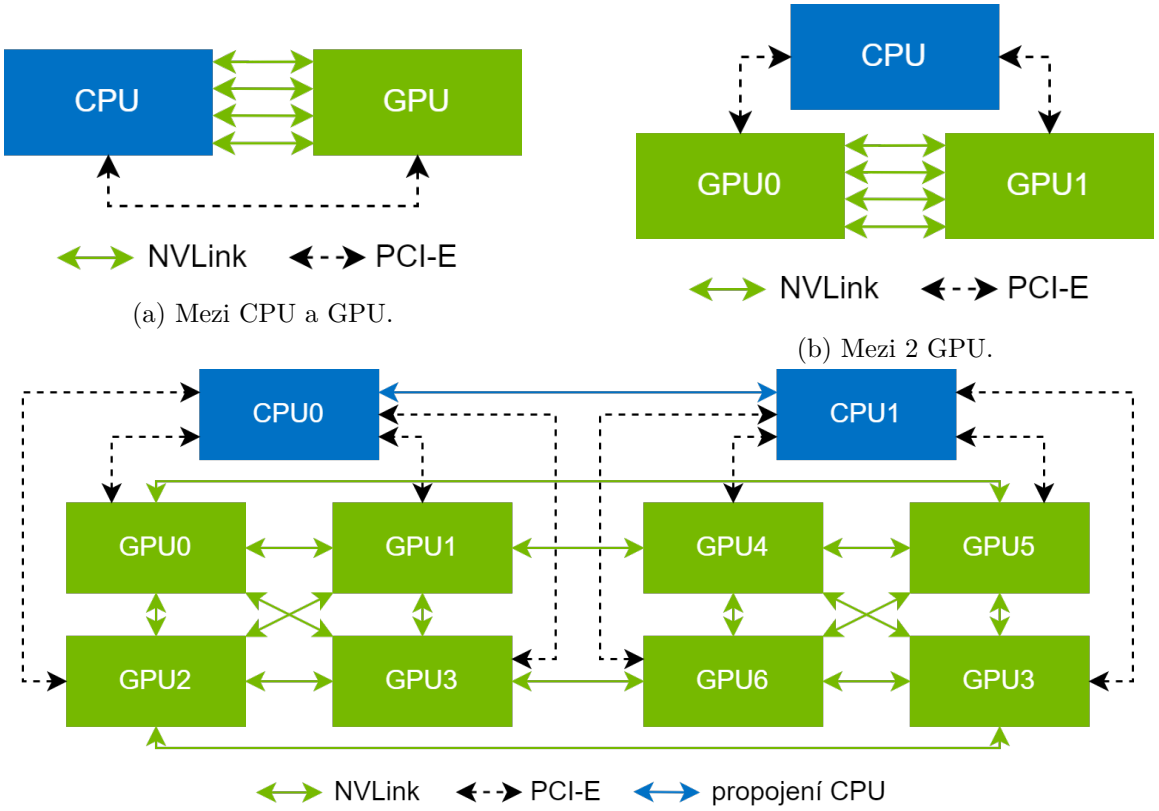
Existuje několik způsobů propojení GPU pomocí NVLink. Nejjednodušší je již zmíněné plné propojení 2 GPU (obrázek 2.1b), které maximalizuje šířku pásma mezi nimi. Lze také plně propojit 4 GPU, 8 GPU na dvousoketových systémech do tzv. *hyperkostky* (obrázek 2.1c) nebo např. plně propojit GPU se systémovou pamětí (obrázek 2.1a), což oproti propojení pomocí PCIe znekoliknásobuje maximální šířku pásma. [18][19] Takového zapojení je použito v serverech DGX-1 od NVIDIA.

NVSwitch je přepínač pro NVLink, který umožňuje propojit 8 nebo 16 GPU na jednom výpočetním uzlu. [16][28] V případě 8 GPU mohou mezi sebou komunikovat 4 páry GPU s maximální šířkou pásma 900 GB/s (NVLink verze 4). NVSwitch také akceleruje multicast a redukční operace a umožňuje komunikaci s jiným NVSwitch. Celkově tak lze propojit až 256 GPU (NVLink verze 4, pouze na NVIDIA DGX H100 serverech).

InfiniBand

InfiniBand je síťová technologie založená na přeposílání přepínaných packetů, která se využívá zpravidla k propojení uzlů ve výpočetních clusterech nebo datových centrech. [27] Jejími přednostmi jsou velice krátké zpoždění, velká šířka přenosového pásma a nenáročnost na CPU koncových zařízení. Využívá konceptu *zero copy*, což je přímý přenos dat mezi pamětmi koncových zařízení bez dalšího bufferování nebo kopírování prostřednictvím software na cílovém zařízení. Umožňuje také uživateli přímý přístup k *Host Channel Adapter* (HCA) bez nutnosti interakce s jádrem operačního systému (*kernel bypass*).

Topologie se skládá z koncových zařízení připojených do sítě prostřednictvím HCA, přepínačů nebo směrovačů. Přesná architektura jednotlivých prvků není ve většině případů známá, jelikož specifikace InfiniBand architektury definuje pouze jejich základní principy, a konkrétní řešení jsou proprietární. Každá InfiniBand síť musí obsahovat jedenoho hlavního *subnet managera*, který ji spravuje — inicializuje, sleduje její změny a komunikuje



(a) Mezi CPU a GPU.

(b) Mezi 2 GPU.

(c) Mezi 8 GPU na dvousoketovém uzlu.

Obrázek 2.1: Příklady propojení GPU a CPU pomocí NVLink a PCI Express. Uvažovány jsou GPU s 4 linkami.

s *subnet management agenty*. Ti běží na všech koncových zařízeních a přepínačích. *Subnet Manager* může být spuštěn na vyhrazeném uzlu nebo na některém z přepínačů. Jedná se tedy o centralizovanou architekturou.

Pro InfiniBand je definováno několik verzí rychlostí spojení. Původní *single data rate* dosahuje rychlosti 2,5 Gb/s pro jednu cestu. Později byly přidány definice pro *double a quad data rate*, které zdvojnásobují a zečtyřnásobují šířku pásma. Každá linka obsahuje 1, 4, 8 nebo 12 cest. Nejčastěji využívaná je linka se 4 cestami. Pro přenos dat jsou využívány měděné nebo optické kabely. InfiniBand se vyhýbá zahazování packetů z důvodu přeplněných bufferů na přepínačích, jako je tomu například u Ethernetu. Místo toho jsou zasílány tzv. *pause* rámce. Je využívána *link-by-link* kontrola toku dat.

Transportní vrstva poskytuje 5 typů služeb — spolehlivé spojení, spolehlivý datagram (volitelné), nespolehlivý datagram, nespolehlivé spojení a rozšířené spolehlivé spojení (volitelné). Spolehlivé služby zaručují přenos v pořadí bez duplicitních paketů a opětovné zaslání ztracených nebo chybných paketů. Nespolehlivé služby žádné záruky neposkytují. Obě skupiny služeb využívají sekvenční čísla paketů a je tak možné detekovat i ztacené pakety nespolehlivých služeb, nicméně standard nedefinuje chování v těchto situacích.

InfiniBand definuje vlastní vysokoúrovňové síťové rozhraní verb API. Stejně jako socket API používá IPv4 a IPv6 adresování, *full-duplex* spojení a rozhraní pro přijímání a odesílání dat na uživatelské úrovni. Avšak na rozdíl od socket API umožňuje zasílat pouze zprávy

(nepodporuje proud bytů), využívá *zero copy* a *kernel bypass*, vyžaduje registraci paměti pro transfer a pracuje asynchronně.

NVIDIA GPUDirect

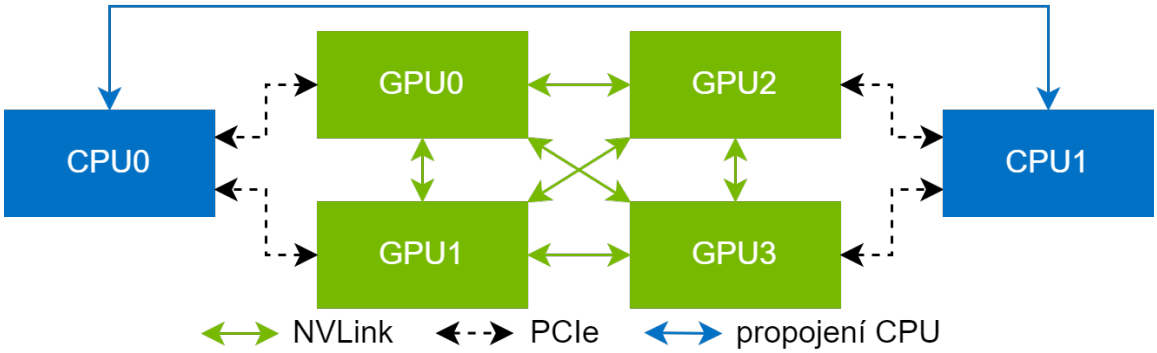
GPUDirect je kolekce technologií firmy NVIDIA pro vysokorychlostní přenosy dat s nízkou latencí bez nutnosti kopírovat data do systémové paměti a nezatěžuje CPU. [24][33] Umožňuje P2P komunikaci GPU, RDMA pro vzdálený přístup nebo přímou komunikaci s diskovým úložištěm. Vždy vybírá nejrychlejší možnou variantu spojení (PCIe, NVLink, InfiniBand a další).

2.2 Hardware

Tato podkapitola se věnuje konkréním multi-GPU systémům. Specifikuje jejich komponenty a popisuje, které technologie jsou použity k propojení jejich komponent a uzlů. Uvedené systémy budou dále použity pro testování technologií a vyhodnocení výsledků.

Superpočítač Barbora

Superpočítač *Barbora* obsahuje 8 výpočetních uzlů s GPU akcelerátory. Každý uzel je osazen dvěma 12 jádrovými procesory Intel Skylake Gold 6126 s frekvencí 2,6 GHz, 192 GB DDR4 paměti a čtyřmi GPU NVIDIA Tesla V100-SXM2 architektury *Volta* (2017). [9] Každé GPU obsahuje 5120 CUDA jader a má k dispozici 16 GB HBM2 paměti o šířce pásma 900 GB/s. [3] GPU jsou připojena k základní desce pomocí SXM2 soketu. Ten propojuje GPU s CPU pomocí PCIe 3.0 a jednotlivá GPU mezi sebou pomocí NVLink 2. generace s podporou GPUDirect (obrázek 2.2). Maximální potenciální výkon na všech GPU je tedy 4×15.7 TFLOPS pro výpočty s jednoduchou přesností. K propojení s dalšími uzly slouží 2×100 Gb/s InfiniBand.

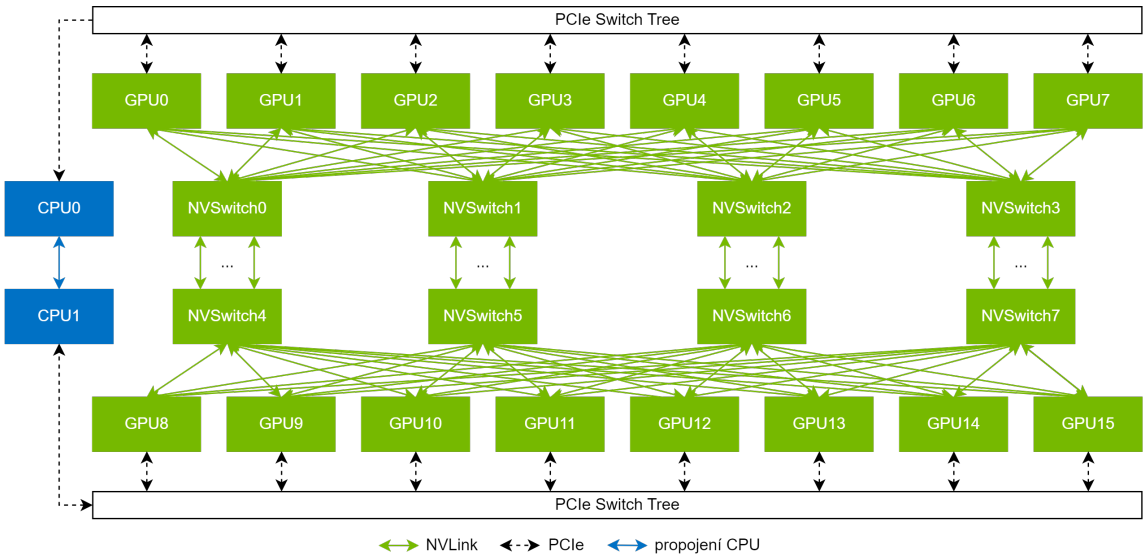


Obrázek 2.2: Schéma propojení komponent na akcelerovaných výpočetních uzlech superpočítače Barbora.

DGX-2

DGX-2 je vysoce výkonný výpočetní uzel od firmy NVIDIA. [12] [7] V jeho jádru se nacházejí 2 procesory Intel Xeon Platinum 8168 s 24 jádry pracující na frekvencí 2.7 GHz s přístupem k 1,5 TB DDR4 operační paměti. Hlavní výpočetní výkon s celkovou teoretickou hodnotou

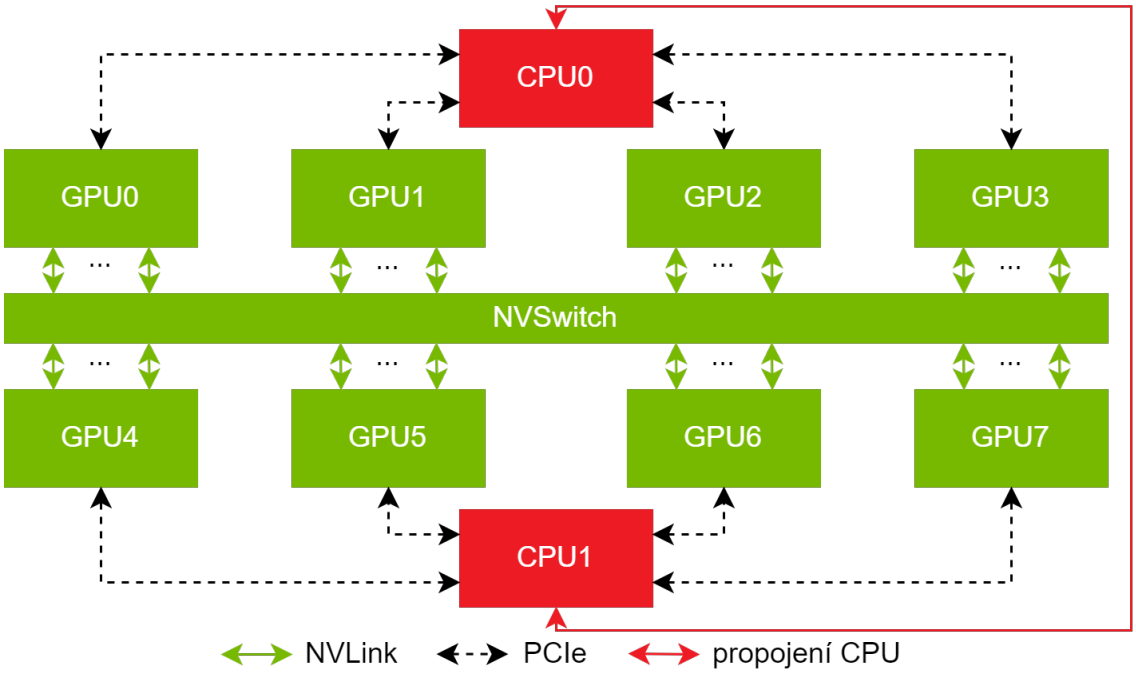
2 PFLOPS pro FP32 poskytuje 16× NVIDIA V100 GPU s 32 GB HBM2 paměti. GPU jsou mezi sebou propojena pomocí NVLink 2. generace skrze 12 NVSwitch do stromové struktury (obrázek 2.3) s celkovou kapacitou paměti 512 GB a propustností 2,4 TB/s. Ke komunikaci s jinými uzly obsahuje 8×100 Gb/s InfiniBand nebo Ethernet.



Obrázek 2.3: Schéma propojení GPU na systému NVIDIA DGX-2 pomocí NVLink s využitím několika NVSwitch.

Superpočítač Karolina

Superpočítač Karolina poskytuje přístup k 829 výpočetním uzlům, z toho 72 výpočetních uzlů je vybaveno GPU akcelerátory. [10] Každý akcelerovaný uzel je osazen dvěma 64 jádrovými procesory AMD EPYC 7763 s frekvencí 2,45 GHz, 1024 GB DDR4 systémové paměti a 8 GPU NVDIA A100 architektury *Ampere* (2020). [6] Ty jsou propojeny s CPU přes PCIe 4.0 a s ostatními GPU pomocí NVLink 3. generace přes NVSwitch skrze SXM4 soket (obrázek 2.4). Všechna GPU tak jsou plně propojena 12 linkami o maximální propustnosti 600 GB/s. V každém GPU se nachází 6912 CUDA jader, která mají přístup k 40 GB HBM2 paměti s šířkou pásma 1555 GB/s. Jejich teoretický výkon je 8×19.5 TFLOPS pro FP32. Pro vysokorychlostní komunikaci s ostatními uzly slouží 4×200 Gb/s InfiniBand rozhraní.



Obrázek 2.4: Schéma propojení propojení komponent na výpočetních uzlech superpočítače Karolina s GPU akcelerátory.

Kapitola 3

Programování multi-GPU systémů

I přes stále zvětšující se možnosti rychlejšího propojení více GPU a CPU zůstává jejich komunikace nejslabším článkem multi-GPU systémů. Některé komunikaci se nelze vyhnout — GPU je stále jenom koprocesor a musí být tedy řízeno z CPU. Ve většině případů má však programátor při implementaci algoritmu kontrolu nad tím, jak komunikace probíhá, a často může výměnu dat podstatně snížit, překrýt nebo dokonce zcela eliminovat.

Tato kapitola se zaměřuje na programování multi GPU systémů. Popisuje práci s těmito systémy pomocí CUDA a dalších knihoven jako jsou NCCL a cuFFT.

3.1 CUDA

CUDA je platforma pro paralelní výpočty. Poskytuje své rozhraní pro mnohé programovací jazyky jako jsou C/C++, Fortran, Python a další. [15] V této práci je výhradně pracováno s rozhraním pro C/C++.

Prvním krokem na multi GPU systémech je zjištění, kolik GPU se v systému nachází. K tomu slouží funkce `cudaGetDeviceCount`. Následně je možné přes jejich *id* (od 0 po zjištěný počet zařízení) iterovat a zjistit informace o nich funkcí `cudaGetDeviceProperties`. Ve vrácené struktuře lze zjistit výpočetní schopnost, velikosti pamětí a další užitečné informace.

Volané CUDA funkce, kernely a streamy jsou vždy vztaženy k aktuálnímu kontextu, to znamená pro zvolené zařízení. Právě používané zařízení je možné zjistit s pomocí funkce `cudaGetDevice`. Zvolit jiné GPU lze pomocí funkce `cudaSetDevice`. Důležitou vlastností změny kontextu je její asynchronnost a nízký overhead.

Unifikovaný virtuální adresový prostor

Unifikovaný virtuální adresový prostor (UVA) vkládá všechny procesory (CPU i GPU) do jednotného paměťového prostoru. [14] Je tak umožněno z hodnoty ukazatele zjistit, na kterém zařízení se data nacházejí. Při kopírování paměti pak není třeba určovat směr kopírování. Před začátkem používání UVA je třeba na všech zařízeních zkontrolovat, zda UVA podporují a to v nastavení příznaku `unifiedAddressing` ve vlastnostech zařízení. UVA je podporován pouze na 64-bitových architekturách.

Peer-to-peer komunikace

Peer-to-peer (P2P) přístup umožňuje jednomu GPU přímý přístup do globální paměti jiného GPU. [15] Všechna GPU musí být připojena ke stejnému PCIe *root complex* nebo propojena přes NVLink. Existují dva módy P2P komunikace:

- **Peer-to-peer Access** — umožňuje načítat a ukládat adresy ukazatelů do paměti jiného GPU,
- **Peer-to-peer Transfer** — dovoluje přímé kopírování dat mezi GPU.

Pro povolení P2P mezi dvěma GPU je třeba nejprve zkontrolovat, jestli je možné navázat spojení, pomocí funkce `cudaDeviceCanAccessPeer`. Pokud ano, lze P2P komunikaci povolit funkcí `cudaDeviceEnablePeerAccess`. Zpřístupněné spojení je pouze jednosměrné, pro povolení opačného spojení je třeba postup opakovat se zaměněnými *id* GPU.

Kopírovat data mezi GPU lze pomocí funkcí `cudaMemcpyPeer` (pokud není podporováno UVA), `cudaMemcpy` a jejich asynchronních verze.

Unifikovaná paměť

Unifikovaná paměť je rozšíření konceptu UVA o automatickou migraci alokovaných stránek do paměti procesoru, který s nimi pracuje. [20] V praxi to znamená, že stačí nahradit volání funkce `malloc` nebo operátoru `new` funkcí `cudaMallocManaged`. Je tak umožné psát podstatně jednodušší kód, jak lze vidět na ukázce kódu 3.1.

Jak už bylo zmíněno, pokud procesor přistupuje ke stránce, která se aktuálně nenachází v jeho paměti, musí čekat, než bude přesunuta. Pro předcházení těmto případům existuje tzv. *prefetching* funkcí `cudaMemPrefetchAsync`, která asynchronně přesune potřebné stránky do paměti specifikovaného procesoru.

Podpora skutečné unifikované paměti s hardwarovou podporou výpadků stránek se objevuje až na architekturách *Pascal (2016)* a později. Do té doby existovala pouze softwarová podpora výpadků stránek, což znamenalo značný overhead v jejich přesunu.

3.2 NCCL

NCCL (NVIDIA Collective Communications Library) je knihovna poskytující rozhraní ke komunikačním primitivům mezi GPU. [2] Umožňuje pracovat s grafickými akcelerátory v kombinaci více GPU na vlákno nebo proces, více GPU na více vláken nebo více GPU na více procesů. Při komunikaci vybírá nejrychlejší cesty podle topologie zapojení. Umožňuje úzkou spolupráci s MPI. Lze tak například vytvářet komunikátory z existujících MPI komunikátorů.

Knihovna zprostředkovává *point to point* send a receive komunikace a 6 komplexních kolektivních operací, kterými jsou:

- **AllReduce** — provede redukci nad daty na všech zařízeních a výsledek rozhlásí mezi všechny ranky,
- **Broadcast** — rozhlásí hodnotu ranku n mezi všechny ranky,
- **Reduce** — provede redukci nad daty všech zařízení a výsledek uloží na ranku n,

```
size_t n;
dim3 grid, blocks;

int* hostData;
int* devData;

hostData = new int[n];
cudaMalloc(&devData, n * sizeof(int));

initDataOnCPU(hostData, n);

cudaMemcpy(devData, hostData, n * sizeof(int),
           cudaMemcpyHostToDevice);

kernelOnGPU<<<grid, blocks>>>(devData, n);

cudaMemcpy(hostData, devData, n * sizeof(int),
           cudaMemcpyDeviceToHost);

useDataOnCPU(hostData, n);
```

(a) Bez unifikované paměti

```
size_t n;
dim3 grid, blocks;

int* data;

cudaMallocManaged(&data, n * sizeof(int));

initDataOnCPU(data, n);

/* Possible prefetch */

kernelOnGPU<<<grid, blocks>>>(data, n);

/* Possible prefetch */

useDataOnCPU(data, n);
```

(b) S unifikovanou pamětí

Obrázek 3.1: Srovnání implementací jednoduché alokace a postupné práce na CPU i GPU bez unifikované paměti a s využitím unifikované paměti.

- **AllGather** — seskládá hodnoty z n ranků do jedné dlaždice, kterou distribuuje mezi všechny ranky,
- **ReduceScatter** — provede redukci nad daty a výsledek rovnoměrně rozmístí mezi všechny ranky.

Výhodou NCCL knihovny oproti jiným implementacím je provádění komunikačních i výpočetních operací v jednom kernelu. Je tak dosaženo vyšší rychlosti synchronizace a vyšší propustnost.

3.3 CUDA-Aware MPI

Message Passing Interface (MPI) je standard definující rozhraní pro meziprocesorovou komunikaci pomocí zasílání zpráv. [24] Využívá se hlavně v paralelních *high performance computing* (HPC) aplikací. Zahrnuje kolektivní i *point to point* komunikace. Standard neurčuje, jakým způsobem mají být komunikace prováděny. Existují tak různé implementace využívajících rozdílných technologií.

V klasické MPI aplikaci akcelerované pomocí GPU by tedy bylo třeba např. při distribuci dat mezi GPU jednotlivých procesů nejprve rozdistribuovat data do systémové paměti procesů a následně je nakopírovat do pamětí GPU.

CUDA-Aware MPI umožňuje přenos dat přímo do paměti GPU. Využívá k tomu UVA, kdy je z adresy zjištěna lokace bufferu. Ubývá tak nutnost kopírování dat do systémové paměti a zátěž procesoru. Implementace navíc mohou použít i GPUDirect RDMA.

3.4 NVSHMEM

NVSHMEM je knihovna pro paralelní jednostranné komunikace implementující standard OpenSHMEM verze 1.3 pro clustery s NVIDIA grafickými kartami propojenými pomocí NVLinku, PCIe nebo InfiniBand a doplňuje jej o vlastní funkcionalitu. [8] Podporuje

- symetrickou alokaci GPU paměti,
- komunikaci iniciovanou GPU s podporou CUDA datových typů,
- rozhraní pro kolektivní spuštění CUDA kernelů na GPU,
- rozhraní pro iniciování přenosu dat z CPU a vykonání přenechávající na GPU s využitím CUDA streamů,
- kolektivní komunikace CUDA warpů nebo bloků a
- blokující i neblokující operace.

NVSHMEM úloha se skládá z několika procesů nazývaných *processing element* (PE), které provádějí stejný program na GPU akcelerovaných uzlech. Všechny PE mají svoje jedinečné ID, jenž je využíváno jako identifikace zdrojového a cílového procesu při komunikaci. PE mohou provádět jednostranné komnunikace nebo kolektivní operace v rámci týmu. Tým může být předdefinován knihovnou:

- `SHMEM_TEAM_WORLD` — všechny PE (výchozí tým),
- `SHMEM_TEAM_SHARED` — všechna PE sdílející paměťový prostor a
- `SHMEM_TEAM_NODE` — všechna PE na jednom uzlu,

nebo může být vytvořen jako podmnožina z předdefinovaných týmů jednou z rodiny funkcí `nvshmem_team_split_*`.

Paměť každého GPU je rozdělena na privátní a sdílenou. Privátní paměť lze alokovat běžnými prostředky pro alokaci a není přístupná z žádného jiného PE. Sdílená paměť, neboli *symetric memory*, naproti je naproti tomu přístupná všem ostatním PE. Dohromady tvoří tzv. *Partitioned Global Address Space* (PGAS). Její alokace je prováděna kolektivně všemi PE funkcí `nvshmem_malloc`, která alokuje paměť z *symetric heap* (SH). Důležitou vlastností alokace je, že na všech PE je alokován stejně velký blok paměti. Velikost SH je dána proměnnou prostředí `SHMEM_SYMMETRIC_SIZE` a musí být nastavena před během programu.

Hlavní výhodou knihovny je možnost provádět datové přenosy, sychronizace a další vzdálené operace přímo z CUDA kernelů. Je tak snížena nutnost další synchronizace mezi GPU a CPU a také lze provádět delší kernely. Nejzřetelnější rozdíl je možné sledovat u malých aplikací, u kterých má velký vliv na jejich výkon množství komunikace.

3.5 HDF5

HDF5 je knihovna, datový model a formát pro ukládání a nakládání s daty. [11] Knihovna poskytuje multiplatformní I/O rozhraní s vysokým výkonem a efektivitou. Je používána v nejrůznějších odvětvích od vědeckých výpočtů po ukládání dat na mobilních zařízeních.

Umožňuje práci s velkým množstvím datových typů, které ukládá do datových modelů popsanými metadaty.

Knihovna je rozdělena do několika vrstev. [32] Nejvyšší vrstva poskytuje vysokoúrovňové rozhraní pro přístup k infrastruktuře (datovým typům, datovým prostorům apod.), objektům datových modelů a nastavitelným vlastnostem knihovny. V interní části se nacházejí komponenty pro správu paměti, konverzi datových typů, I/O filtry a další. Nejnižší vrstva *virtual file layer* se skládá z implementací *virtual file driver* (VFD) mapující HDF5 adresový prostor na adresy úložiště. Uživatel má možnost zvolit, který VFD má knihovna použít.

Výchozím VFD je SEC2. Ten pracuje s pomocí POSIXových funkcí pro práci se soubory. MPI-IO VFD se používá v paralelních aplikacích využívajících implementovaných pomocí MPI. Direct VFD pracuje podobně jako SEC2, ale na rozdíl od něj vynucuje přímý zápis dat do souborového systému. Zabraňuje tak bufferování v operačním systému. GDS VFS využívá technologii GPUDirect Storage a umožňuje kopírovat data přímo mezi GPU a úložištěm při vyšší rychlosti. Podobně lze také využít MPI-IO VFD v případě, že byla knihovna zkompilování s MPI knihovnou podporující CUDA-Aware MPI.

3.6 Paralelní FFT

Rychlá Fourierova transformace (FFT) je jedním z nejdůležitějších a také časově a výpočetně náročných výpočtů užívaných v simulacích. [29] [30] Jeho hlavní výhodou oproti standardní implementaci Fourierovy transformace je jeho efektivita. Jeho časová složitost je totiž $O(n \cdot \log(n))$ oproti původní $O(n^2)$. S rostoucími požadavky na velikosti simulací a rychlost výpočtu je nezbytné mít možnost provádět tento výpočet paralelně. Typická aplikace využívající FFT má schéma:

1. dopředné FFT,
2. výpočet hodnot ve frekvenční doméně,
3. zpětné FFT a
4. využití vypočítaných hodnot.

Pro práci se spočítanými hodnotami v reálné i frekvenční doméně je nezbytné znát jejich způsob distribuce mezi procesory.

Dekompozice domény

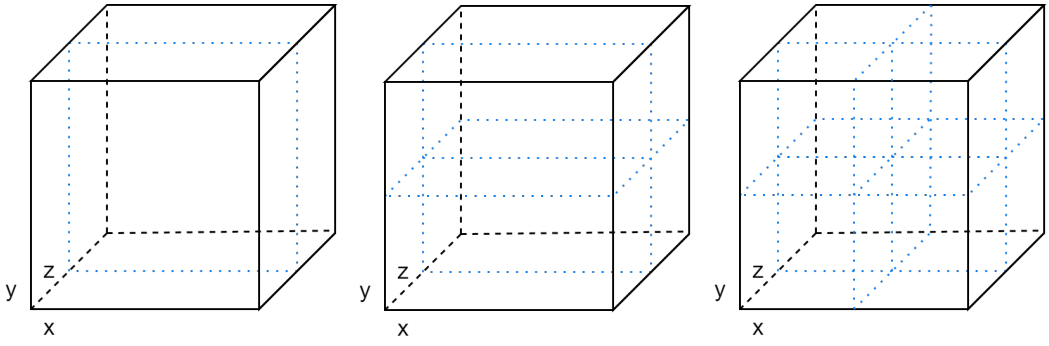
Při paralelním výpočtu FFT je nezbytné určit, jakým způsobem bude doména dekomponována. Pro 3D FFT se používá *slab* (1D dekompozice), *pencils* (2D dekompozice) nebo *blocks* (3D dekompozice).

Slab dekompozice rozděluje doménu v jedné dimenzi rovnoměrně mezi procesory (obrázek 3.2a). Ty tak mohou vypočítat FFT pro ostatní dimenze, u kterých jsou přítomny všechny prvky. Při výpočtu 3D FFT používající dekompozici v ose z by tedy bylo možné provést FFT v osách x a y. Pro výpočet zbývající dimenze je potřeba získat chybějící hodnoty, které jsou uloženy v pamětech ostatních procesorů. Existuje několik přístupů, jak provést výměnu dat. Nejjednodušším je zaslat potřebná data ostatním procesorům, tzv. distribuovaná metoda. To však vede k příliš objemné komunikaci. Proto se častěji používá metoda, kdy je celá

doména transponována a lze tak provést FFT v nejvyšší dimenzi, tedy u 3D FFT v ose z. Pro obnovení původního pořadí je třeba znova provést transpozici.

Pencils dekompozice rozděluje doménu ve 2 dimenzích (obrázek 3.2b) a vzniklé hranoly rozděluje mezi procesory. Rozdělení může být provedeno kolem libovolné osy. Při výpočtu je nejprve vypočteno FFT v ose, ve které jsou přítomny všechny prvky, poté je provedena transpozice a postup se opakuje. Opět je pro uvedení dat do původního pořadí provést nakonec ještě jednu transpozici.

Blocks dekompozice rozbíjí doménu na 3D bločky (obrázek 3.2c), které jsou rovnoměrně rozděleny mezi procesory. V každém kroku je komunikováno s procesory v právě prováděné dimenzi a následně provedena transpozice. Tento přístup se v praxi příliš nepoužívá.



(a) *Slab*, 1D dekompozice v z

(b) *Pencils*, 2D dekompozice v y a z

(c) *Blocks*, 3D dekompozice v x, y a z

Obrázek 3.2: Ukázka dekompozice 3D domény podle dimenzionality

Pro maximální výkon je vhodné u všech postupů dekompozice po provedení transformace nevracet hodnoty do původního pořadí. Místo toho je lepší, pokud je to možné, přizpůsobit distribuci ostatních dat simulace transponovanému pořadí.

Knihovny

Existuje řada knihoven implementující paralelní FFT. Většina z nich implementuje výpočet na CPU s využitím MPI a to hlavně pro *in-place* transformace s 1D nebo 2D dekompozicí. Knihovny většinou implementují FFT pomocí objektu plánu, který nese informaci o tom, kterým algoritmem se bude výpočet provádět na základě rozměrů domény a dostupných výpočetních prostředků, a další pomocná data. Plán rovněž může alokovat pomocné pole. Životní cyklus plánu je tedy:

1. vytvoření objektu plánu,
2. přizpůsobení plánu,
3. inicializace plánu,
4. využití plánu (i opakované) a
5. deinicializace plánu.

V této práci jsou popsány knihovny FFTW, cuFFT a cuFFTMp. Knihovna FFTW byla vybrána, protože se jedná o jednu z nejpopulárnějších knihoven pro paralelní výpočet FFT na CPU. Knihovna cuFFT byla vybrána z důvodu její snadné dostupnosti a vysokému výkonu. Většina ostatních implementací paralelního FFT nstejně knihovnu cuFFT využívá pro interní výpočet FFT a implementují pouze výměnu dat mezi jednotlivými GPU. Knihovna cuFFTMp je představena, jelikož je rozšířením cuFFT, ale na rozdíl od něj používá novější technologie a víceprocesové zpracování.

FFTW

Knihovna *Fastest Fourier Transform in the West* (FFTW) je jednou z nejpopulárnějších knihoven pro výpočet FFT na CPU. [1] Umožňuje provádět *real to complex* (R2C), *complex to real* (C2R), *complex to complex* (C2C) a *real to real* (R2R) s jednoduchou, dvojitou nebo rozšířenou přesností. Transformace může být provedena na jednom uzlu s podporou *multithreadingu* nebo více uzlech s využitím MPI. Knihovna dovoluje také oba přístupy kombinovat, avšak tato kombinace nemusí přinést vyšší výkon, jelikož transpozice matic probíhají pouze pomocí jednoho vlákna.

Před vytvořením plánů pro více vláken nebo procesů je nutné knihovnu inicializovat funkcemi `fftw_init_threads` nebo `fftw_init_mpi`. Při kombinaci obou možností je třeba zachovat pořadí — nejprve inicializovat vlákna a až poté procesy. Počet vláken pro výpočet je možné nastavit funkcí `fftw_plan_with_nthreads`.

Při vytváření plánu transformace je knihovnou vybírán vhodný algoritmus pro výpočet. Jeho výběr závisí na dostupném hardware a rozměrech simulace. Knihovna nabízí několik způsobů, jak k výběru přistoupit, a to na základě plánovacích příznaků:

- `FFTW_ESTIMATE` — k výběru je použita heuristická funkce, pravděpodobně nebude vybrán nejefektivnější algoritmus.
- `FFTW_MEASURE` — knihovna provede měření několika druhů FFT algoritmů a z nich vybere ten nejlepší. To může zabrat i několik sekund.
- `FFTW_PATIENT` — stejný jako `FFTW_MEASURE`, ale je brána v potaz větší škála algoritmů. Většinou je tak dosaženo nalezení ještě optimálnějšího algoritmu. Vyplatí se hlavně pro větší simulace.
- `FFTW_EXHAUSTIVE` — rozšiřuje `FFTW_PATIENT` ještě o další algoritmy.
- `FFTW_WISDOM_ONLY` — vytvoří plán z *wisdom* souboru. Je možné jej kombinovat s ostatními příznaky a tím specifikovat typ wisdom souboru.

Při výběru plánovacího příznaku s měřením je nutné alokovat pomocné pole. Čas výběru plánovacího algoritmu je možné omezi funkcí `fftw_set_timelimit`.

Knihovna dále poskytuje možnost u vícedimenzionálních MPI plánů neprovádět poslední traspozici a ponechat data v transponovaném formátu. To může podstatně snížit čas transformace, je však třeba brát v potaz rozložení dat i v dalších výpočtech. Pro neprovedení poslední transpozice je nutné plán vytvářet s příznakem `FFTW_MPI_TRANSPOSED_OUT` a plán, který očekává transponovaný vstup, s příznakem `FFTW_MPI_TRANSPOSED_IN`.

K provádění plánů slouží rodina funkcí `fftw_execute_*`. Pro MPI plány je třeba použít rozhraní `fftw_mpi_execute_*`. Výpočtu paralelního FFT používá 1D dekompozici v nejvyšší dimenzi. Knihovna snaží co nejrovnoměrněji rozdělit data mezi procesy. K zjištění

přesného rozložení dat poskytuje rodinu funkcí `fftw_mpi_local_size_*` a to i pro transponovanou distribuci.

S výjimkou R2R transformací jsou podporovány všechny typy transformací *in-place* i *out-of-place*. R2R transformace lze provádět pouze *in-place*. Při použití *in-place* transformace je nutné, aby každý řádek transformované matice byl zarovnán na velikost $(\frac{n}{2} - 1)$ komplexních prvků. Pro *out-of-place* transforamce lze navíc při plánování specifikovat, zda mají být zachována vstupní data transformace, pomocí příznaků `FFTW_DESTROY_INPUT`, nebo `FFTW_PRESERVE_INPUT`. Zachování vstupních dat je výchozí možností, avšak může vést k výběru méně efektivního algoritmu.

Pro destrukci plánů slouží funkce `fftw_destroy_plan`. Ta ovšem uvolní pouze alokovanou paměť daného plánu. Knihovna uchovává další perzistentní data, která je nutné uvolnit funkcí `fftw_cleanup` před ukončením programu. Při využití více vláken nebo procesů je třeba ještě navíc volat funkce `fftw_cleanup_threads` a `fftw_cleanup_mpi`.

Pro vyšší výkon jsou použity pro výpočet SIMD jednotky. Proto obsahuje vlastní funkce pro alokaci paměti `fftw_malloc` a `fftw_alloc_*`, které alokují paměť s požadovaným zarovnáním. V případě, že paměť dat není zarovnána, je třeba specifikovat příznak `FFTW_UNALIGNED` při vytváření plánu transformace.

Aby bylo možné znovuvyužít FFTW plány a neprovádět opětovný výběr algoritmů na stejných systémech, umožňuje knihovna exportovat jejich data do tzv. *Wisdom* souborů. Knihovna poskytuje také utilitu pro vytvoření tzv. globálního wisdom souboru, který pak lze načíst z jakéhokoliv programu.

Mimo vlastní implementaci implementují FFTW rozhraní, i když třeba jenom částečně, Intel MKL, AMD clFFT, NVIDIA cuFFT a další.

cuFFT

Knihovna *CUDA Fast Fourier Transform library* (cuFFT) je knihovna pro výpočet rychlé Fourierovy transformace na NVIDIA GPU. [5] Je součástí kolekce GPU akcelerovaných knihoven *CUDA-X*, který náleží balíkům NVIDIA CUDA Toolkit a NVIDIA HPC SDK. Umožňuje výpočet FFT s jednoduchou, dvojitou a sníženou přesností. Implementuje R2C, C2R a C2C transformace. Součástí knihovny je také knihovna cuFFTW, která implementuje část FFTW rozhraní na GPU a tím umožňuje jednodušší přechod aplikací z výpočtu na CPU k využití GPU.

Knihovna používá rozhraní velice podobné FFTW API. Základní cuFFT rozhraní umožňuje provádět 1D, 2D a 3D transformace libovolných rozměrů na 1 GPU. Rozšířené rozhraní, označené jako cuFFTXt, přidává možnost provádět paralelní FFT a také transformace se sníženou přesností. Maximálně lze použít pro výpočet 16 GPU. Multi-GPU transformace využívá 1D dekompozici v nejvyšší dimenzi a komunikace probíhá P2P. Pro použití multi-GPU plánu je třeba provést operace v tomto pořadí:

1. vytvořit objekt plánu funkcí `cufftCreate`,
2. vybrat, která GPU se mají pro výpočet použít, funkcí `cufftXtSetGPUs`,
3. inicalizovat plán jednou z rodiny funkcí `cufftMakePlan*` (případně `cufftXtMakePlan*`),
4. provádět transformace jednou z rodiny funkcí `cufftXtExecDescriptor` a
5. uvolnit zdroje plánu funkcí `cufftDestroy`.

Při výběru GPU je třeba zadat ID alespoň 2 GPU. Je důležité brát v potaz, že plán bude pracovat s GPU v takovém pořadí, jaké bylo určeno při jejich výběru. Teoreticky je možné vybrat GPU i opakovaně, nemá to žádný pozitivní vliv na výkon. Může to však být užitečné v situaci, kdy je třeba spustit multi-GPU implementaci na 1 GPU.

Pro alokaci a distribuci dat na více GPU knihovna využívá deskriptor paměti `cudaLibXtDesc`. Ten uchovává informace o své verzi, knihovně, pro kterou byl vytvořen, počtu použitích GPU, jejich ID, ukazatele na alokovanou paměť a její velikosti a formátu. Formát definuje, jakým způsobem jsou data distribuovány mezi jednotlivá GPU. Může být jedním z hodnot s prefixem `CUFFT_XT_FORMAT_`:

- `INPUT` — lineární distribuce v nejvyšší dimenzi, slouží jako vstup C2C transformace,
- `OUTPUT` — lineární distribuce v druhé nejvyšší dimenzi, jedná se o formát výstupu C2C tranformace,
- `INPLACE` — lineární distribuce v nejvyšší dimenzi v odsazeném formátu, určený pro vstup R2C transformace,
- `INPLACE_SHUFFLED` — lineární distribuce v druhé nejvyšší dimenzi, slouží jako výstupní formát C2R tranformace.
- `1D_INPUT_SHUFFLED` — speciální formát vstupních data jednodimenzionální transformace, pro které dosahuje lepších výsledků než běžný vstupní formát.

Formát desktriptoru je vybrán při jeho vytvoření funkcí `cufftXtMalloc` a dále je měněn funkcí provádějící FFT nebo funkcí pro kopírování dat `cufftXtMemcpy`. Lze jej také měnit manuálně na riziko programátora. Funkce `cufftXtMemcpy` umožňuje rozdistribuovat data mezi GPU, kopírovat je zpět do systémové paměti, nebo provést finální transpozici dat po transformaci.

Při vytváření plánu jsou v závislosti na rozměrech transformace a dalších faktorů alokována pomocná pole. Jejich velikost je možné zjistit ještě před jejich vytvořením s pomocí rodiny funkcí `cufftEstimate`. Knihovna nabízí možnost jejich velikost omezit, minimalizovat nebo optimalizovat pro nejvyšší výkon funkcí `cufftXtSetWorkAreaPolicy`. Za předpokladu, že není více transformací prováděno najednou, lze také pracovní prostory sdílet mezi plány. Je však nutné před inicializováním plánu vypnout automatickou alokaci funkcí `cufftSetAutoAllocation` s parametrem 0, provést manuální alokaci na každém GPU příslušné velikosti a asociovat pracovní prostor s plánem funkcí `cufftSetWorkArea` (`cufftXtSetWorkArea` pro multi-GPU).

Provádění transformací je implicitně synchronní, lze je však provádět také asynchronně. Pro plán je třeba definovat stream, ve kterém se provádí, pomocí funkce `cufftSetStream`. Plán může být vždy asociován maximálně s jedním streamem. Při multi-GPU transformacích, to ale může způsobit potřebu složitější synchronizace, protože CUDA stream je vždy aktuální pouze pro GPU, pro které byl vytvořen.

cuFFTMp

Knihovna *cuFFT Multi Process library* (cuFFTMp) vychází z cuFFT knihovny, je ale určena pro víceprocesové aplikace využívající více GPU s využitím MPI bez omezení počtu použitých GPU. [4] Je součástí NVIDIA HPC SDK balíku. Na rozdíl od multi-GPU cuFFT používá ke komunikaci NVSHMEM knihovnu a umožňuje využití více uzlů pro výpočet.

Používá cuFFT rozhraní obohacené o vlastní funkcionalitu. Program využívající knihovnu by měl provést tyto kroky:

1. na začátku programu inicializovat MPI funkcí `MPI_Init`,
2. vybrat GPU, se kterým proces pracuje funkcí `cudaSetDevice`,
3. vytvořit objekt plánu funkcí `cufftCreate`,
4. připojit MPI komunikátor k plánu pomocí funkce `cufftMpAttachComm`,
5. inicalizovat plán jednou z rodiny funkcí `cufftMakePlan*` (případně `cufftXtMakePlan*`),
6. provádět transformace jednou z rodiny funkcí `cufftXtExecDescriptor`,
7. uvolnit zdroje plánu funkcí `cufftDestroy` a
8. na konci programu deinicializovat MPI funkcí `MPI_Finalize`.

Stejně jako u knihovny cuFFT může být pro alokaci a distribuci dat deskriptor paměti `cudaLibXtDesc`. Ten ale obsahuje ukazatel na data a jejich velikost pouze pro GPU využívané procesem. Paměť desktriptoru je symetrickou pamětí alokovanou pomocí NVSHMEM z SH. Knihovna podporuje 1D a 2D distribuci dat. 2D distribuci lze nastavit přesně podle potřeb pomocí funkce `cufftXtSetDistribution` a zavádí pro ni další formáty deskriptoru `DISTRIBUTED_INPUT` a `DISTRIBUTED_OUTPUT` se stejným prefixem jako u cuFFT. Funkce `cufftXtMemcpy` umožňuje kopírovat již distribuovaná data na nebo z GPU nebo provést finální traspozici po transformaci.

FFT lze také počítat bez využití desktriptoru z dat alokovaných z SH. V takovém případě je třeba nastavit, jaký vstupní a výstupní formát má plán použít funkcí `cufftXtSetSubformatDefault` před provedením transformace.

Knihovna umožňuje také umožňuje sdílení pracovního prostoru mezi plány za stejných podmínek, avšak pracovní prostor musí být stejně jako paměť deskriptoru alokována z SH. Plán lze asociovat s CUDA streamem stejným způsobem jako u cuFFT. Pro změnu distribuce dat bez cuFFT plánu obsahuje knihovna navíc *Reshape* rozhraní.

Kapitola 4

k-Wave

k-Wave je open source projekt University College London a Vysokého učení technického v Brně pro simulaci šíření vln v čase v 1D, 2D a 3D prostoru. [34] Tato kapitola popisuje principy, podle kterých je simulace počítána, a analyzuje aktuální akcelerované implementace simulátoru v jazyce C++.

4.1 Matematický model

Při průchodu akustické vlny stlačitelným médiem vznikají fluktuace. Tyto fluktuace lze popsat několika diferenciálními rovnicemi prvního řádu. Jsou to zákon zachování hybnosti 4.1, zákon zachování hmotnosti 4.2 a závislost tlaku na hustotě 4.3, kde $\mathbf{u}$ je akustická rychlost částic, p je akustický tlak, ρ je akustická hustota, ρ_0 je hustota prostředí a c_0 je izentropická rychlost zvuku.

$$\frac{\partial \mathbf{u}}{\partial t} = -\frac{1}{\rho_0}\nabla p \tag{4.1}$$

$$\frac{\partial \rho}{\partial t} = -\rho_0 \nabla \cdot \mathbf{u} \tag{4.2}$$

$$p = c_0^2 \rho \tag{4.3}$$

V akustice jsou tyto rovnice většinou spojeny do jedné diferenciální rovnice druhého řádu. V *k-Wave* je ale simulace počítána pomocí soustavy diferenciálních rovnic prvního řádu. Má to několik výhod. Lze tak například do rovnic jednoduše zakomponovat zdroje hmotnosti a síly, provádět anisotropní filtrování vln, které byly absorbovány okrajem simulované domény, nebo vypočítat akustickou intenzitu a jiné.

Pro řešení parciální diferenciálních rovnic je využívána *k-space* pseudospektrální metoda, která kombinuje spektrální výpočet prostorových derivací s propagací času vyjádřené v prostorové fekvenční doméně. K výpočtu prostorových derivací je využita Fourierova kolokační spektrální metoda, která k výpočtu derivace využívá všechna data a dosahuje tak vyšší přesnosti než běžné metody počítající gradient z okolí bodu.

Jelikož výpočet pro výpočet prostorových gradientů je použito FFT, je spektrum periodické. To způsobuje, že vlny opouštějící doménu na jedné straně, se objeví na druhé. Pro eliminaci tohoto jevu používá *k-Wave* tzv. *perfectly matched layer* (PML), což je tenká vrstva okolo simulované domény s vysokou absorpcí a minimálním odrazem.

Simulace v *k-Wave* umožňuje měnit celou řadu parametrů. Podle nich se pak náležitě mění diferenciální rovnice prvního řádu. Těmito parametry jsou:

- **Vlastosti média** — médium je specifikováno svou hustotou, rychlostí šíření zvuku, absorpcí a linearitou propagace vln. Prostředí může být pro každý z těchto parametrů homogenní nebo heterogenní.
- **Zdroje** vln — simulace umožňuje definovat lineární zdroje akustických vln.

Složitost počítané simulace se tedy mění s danými parametry. Nejsložitější případ nastane při simulování v heterogenním prostředí při nelineární propagaci vln s zdroji v osách x, y a z. Výpočet se bude skládat z výpočtu prostorového gradientu na základě *k-space* kolokační metody (rovnice 4.4 a 4.6), výpočtu nové hodnoty rychlosti, hustoty a tlaku v kroce (rovnice 4.5, 4.7 a 4.9) s absorpčním výrazem založeným na zlomkovém Laplaceovém operátoru (rovnice 4.8). Z největší části se ve výpočtu objevují základní aritmetické operace, několikrát se ale objevuje také výpočet Fourierovy tranformace.

$$\frac{\partial}{\partial \xi} p^n = \mathcal{F}^{-1}\{ik_\xi \kappa e^{ik_\xi \Delta\xi/2} \mathcal{F}\{p^n\}\} \tag{4.4}$$

$$u_\xi^{n+\frac{1}{2}} = u_\xi^{n-\frac{1}{2}} - \frac{\Delta t}{\rho_0} \frac{\partial}{\partial \xi} p^n + \Delta t \mathrm{S}_{F_\xi}^n \tag{4.5}$$

$$\frac{\partial}{\partial \xi} u_\xi^{n+\frac{1}{2}} = \mathcal{F}^{-1}\{ik_\xi \kappa e^{-ik_\xi \Delta\xi/2} \mathcal{F}\{u_\xi^{n+\frac{1}{2}}\}\} \tag{4.6}$$

$$\rho_\xi^{n+1} = \frac{\rho_\xi^n - \Delta t \rho_0 \frac{\partial}{\partial \xi} u_\xi^{n+\frac{1}{2}} + \Delta t \mathrm{S}_{\mathrm{M}_\xi}^{n+\frac{1}{2}}}{1 + 2\Delta t \frac{\partial}{\partial \xi} u_\xi^{n+1/2}} \tag{4.7}$$

$$\mathrm{L_d} = -\tau \mathcal{F}^{-1}\{k^{y-2} \mathcal{F}\{\rho_0 \Sigma_\xi \frac{\partial}{\partial \xi} u^{n+\frac{1}{2}}\}\} + \eta \mathcal{F}^{-1}\{k^{y-1} \mathcal{F}\{\rho^{n+1}\}\} \tag{4.8}$$

$$p^{n+1} = c_0^2 (\rho^{n+1} + \frac{B}{2A} \frac{1}{\rho_0} (\rho^{n+1})^2 - \mathrm{L_d}) \tag{4.9}$$

4.2 Formát vstupního souboru

Vstupní HDF5 soubor se skládá z hlavičky a dat v kořenové skupině. V hlavičce jsou uloženy informace o datu vytvoření souboru, jeho popisu, typu a verzi. Kořenová skupina obsahuje podskupiny a datasety konkrétních matic. Pro uložení dat jsou používány 3 druhy datových typů — reálný a komplexní jako `H5T_IEEE_F32LE` a indexovací jako `H5T_STD_U64LE`.

Rozměry simulace jsou definovány v celočíselných proměnných `Nx`, `Ny`, `Nz` a `Nt`, velikosti kroků pak v reálných proměnných `dx`, `dy`, `dz` a `dt`. Vlastnosti simulace jsou uloženy jako příznaky v celočíselných proměnných `u*_source_flag`, `p_source_flag`, `p0_source_flag`, `transducer_source_flag`, `nonuniform_grid_flag`, `nonlinear_grid_flag` a `absorbing_flag`.

Médium

Pro určení vlastností média je možné definovat několik parametrů. V závislosti na homogenitě média mají tyto parametry velikost jedné hodnoty, nebo celé simulované domény.

1. **Rychlost** šíření zvuku médiem — je dána reálnou maticí `c0`.
2. **Hustota** média — určena reálnou maticí `rho0` a dále reálnými maticemi `rho0_sg*` pro každou dimenzi simulace.

3. **Nelinearita** média — uložena v reálné matici `BonA`.

4. **Absorpce** média — definována pomocí absorpčního prefaktoru v reálné matici `alpha_coeff` a absorpčního exponentu `alpha_power`.

PML

Vlastnosti PML je možné definovat pro každou dimenzi samostatně. Tloušťka PML je definována celočíselnou hodnotou `pml_*_size` a její absorpce jako reálná hodnota `pml_*_alpha`. Konkrétní hodnoty PML jsou uchovány v reálné matici `pml_*` a *staggered* hodnoty v reálné matici `pml_*_sg*`. Jejich velikost je dána velikostí dané dimenze.

Akustické zdroje

V *k-Wave* je možné použít 3 typy akustických zdrojů v médiu:

1. **Počáteční tlak** — je dán reálnou maticí `p0_source_input` stejných rozměrů jako simulace.

2. **Časově proměnlivý zdroj tlaku** — body zdroje tlaku jsou definovány 1D maticí `p_source_index` s indexy bodů zdrojů a jejich hodnoty 2D reálnou maticí `p_source_input`, kde každý řádek matice obsahuje hodnotu zdroje pro aktuální časový krok t.

3. **Časově proměnlivý zdroj rychlosti** — je obdobně definován jako časově proměnnlivý zdroj tlaku, indexy bodů zdrojů se ale nacházejí v matici `u_source_index` a obsahuje vlastní 2D matici hodnot pro každou dimenzi v `u*_source_input`.

Senzory

Simulátor umožňuje vzorkování hodnot pomocí senzorů, které jsou reprezentovány jako body v doméně, jejichž hodnota je s danou frekvencí snímána. Senzory jsou definovány ve vstupním souboru jako

- 1D indexy do simulované domény v matici `sensor_mask_index`, nebo jako tzv.

- *cuboidy*, což jsou trojdimenzionální hranoly definované dvěma protilehlými body, jejichž celý objem je snímán.

Je umožněno vzorkovat rychlost částic v osách nebo akustický tlak buď jako *raw* sérii, nebo s využitím redukčních operátorů pro zjištění minimální hodnoty, maximální hodnoty nebo směrodatné odchylky.

4.3 Stávající implementace

Implementace *k-Wave* je určena pro MATLAB. Nicméně kvůli vysokým nárokům na výpočetní výkon byla hlavní část simulátoru přepsána také do jazyka C++. Aktuálně existují 3 verze implementace, které se liší ve využitých akceleračních technologiích. Jsou to:

- `kSpaceFirstOrderOMP` — implementace určená pro výpočet simulací ve 2D a 3D na CPU s využitím OpenMP na jednom uzlu,

- `kSpaceFirstOrderMPI` — implementace pro výpočet velkých simulací ve 3D na CPU s využitím standardu MPI umožňující výpočet na jednom nebo více uzlech a
- `kSpaceFirstOrderCUDA` — implementace pro výpočet malých a středně velkých simulací v 2D a 3D na jednom NVIDIA GPU napsaná v jazyce CUDA/C++.

Všechny implementace dodržují stejnou strukturu modulů, ze kterých se program skládá, liší se v jejich konkrétní implementaci.

Parameters

Modul `Parameters` se skládá z tříd `Parameters` a `CommandLineParameters`. Třída `CommandLineParameters` slouží ke zpracování argumentů programu. K parsování využívá POSIX funkci `getopt_long`. Argumenty mohou určit vstupní a výstupní soubor, soubor kontrolního bodu, vzorkování a další. OpenMP a CUDA implementace používají dodatečný příznak `-t` pro specifikaci počtu vláken, CUDA verze ještě navíc příznak `-g` umožňující vybrat GPU, na kterém výpočet poběží. Třída `Parameters` slouží k uchování nejen spouštěcích parametrů programu, ale také konkrétních parametrů simulace načtených ze vstupního HDF5 souboru.

V CUDA implementaci se navíc nacházejí ještě třídy `CudaParameters` a `CudaDeviceConstants`. První z nich obsahuje konfiguraci GPU a metody pro zjištění verze použitého kódu a ovladače. Druhá je pak třídou určenou pro konfiguraci konstantní paměti GPU. Obsahuje vybrané konstanty a parametry simulace.

Logger

Modul `Logger` obsahuje třídu `Logger`, která má na starosti logování běhu programu. Obsahuje tak metody pro tisk logů, chybových hlášení, nástroje pro formátování textu a všechny zprávy použité v programu. Množství tisknutých informací zobrazuje podle nastavení statické proměnné `sLogLevel`. MPI implementace navíc obsahuje třídy `ErrorChecker` a `DistException` pro kontrolu a emitování chyb v MPI prostředí a CUDA implementace zase makra pro kontrolu chyb CUDA funkcí.

Hdf5

Modul `Hdf5` je složen z tříd `Hdf5FileHeader` a `Hdf5File`. Třída `Hdf5FileHeader` spravuje všechny informace ukládané v HDF5 souborech, se kterými program pracuje. Jsou to například verze programu, datum vytvoření, informace o času zpracování, počet jader apod. Třída `Hdf5File` poskytuje přístup k datům HDF5 souborů. Umožňuje vytvářet nové nebo upravovat stávají datasety, číst jejich data atd. V OMP a CUDA verzích programu je využit výchozí VFD, v MPI verzi je využit MPI-IO VFD a k tomu přizpůsobeny metody pro čtení a zápis.

MatrixClasses

Modul `MatrixClasses` obsahuje implementace všech matic využitých v simulaci. Jejich bázovou třídou je ryze virtuální třída `BaseMatrix` definující jejich společné rozhraní, které umožňuje získat rozměry matice a načíst nebo zapsat data z HDF5 souboru. Další vrstou tříd jsou třídy `BaseFloatMatrix` a `BaseIndexMatrix`. Tyto třídy implementují alokace a

dealokace paměti, nulování a rozměry matic založených na datových typech *float* a *size_t*. Z těchto tříd již dědí konečné implementace tříd.

Třída `IndexMatrix` je určená pro bezznaménkové celočíselné hodnoty. V *k-Wave* obsahuje buď hodnoty indexů prvků jiných matic nebo tzv. *cuboidy* (kvádry). Pro cuboidy obsahuje navíc metody pro získání levého horního rohu a pravého spodního rohu cuboidu. Implementace dále umožňuje přepočet hodnot z MATLAB do C++ a naopak, protože MATLAB používá indexování od 1, C++ narozdíl od něj od 0.

Třída `RealMatrix` implementuje načítání a zápis reálných matic v jednoduché přesnosti. V OMP implementaci se navíc nachází její rozšíření třídou `FftwRealMatrix` o možnost provádět *real to real* FFT transformace v ose y. Toho je využito při osově souměrných simulacích ve 2D.

Pro implementaci komplexních matich slouží třída `ComplexMatrix`. Stejně jako `RealMatrix` ukládá komplexní čísla v jednoduché přesnosti. Jejím rozšířením jsou třídy `FftwComplexMatrix` (OMP a MPI implementace) a třída `CufftComplexMatrix`, které umožňují provádět n-dimenzionální (v závislosti na dimenzionalitě simulace) a 1D (v každé ose) *real to complex* a *complex to real* FFT nad jejími daty. `FftwComplexMatrix` implementuje FFT s pomocí *fftw3* knihovny, `CufftComplexMatrix` pak pomocí knihovny *cuFFT*. Třídy také obsahují FFT plány jako statické proměnné a rozhraní pro jejich konstrukci a destrukci.

MPI implementace ještě navíc obsahuje *Scatter* (rozptýlené) a *Broadcast* (rozkopírované) verze matic. Rozptýlené matice jsou rozdistribuovány mezi ranky v nejvyšší dimenzi simulace. CUDA implementace používá k transpozici matic speciální kernely z jmenného prostoru `TransposeCudaKernels`.

OutputStreams

V modulu `OutputStreams` je implementováno vzorkování domény na pozicích senzorů. Všechny třídy pro vzorkování jsou založeny na bázové třídě `BaseOutputStream`, která definuje společné rozhraní výstupních streamů, nese jejich společná data a definuje redukční operátor třídy `ReduceOperator`. Implementované redukční operátory jsou kvadratický průměr, maximum a minimum. Umožněno je také vzorkování *raw* dat, která jsou ukládána jako časová posloupnost.

Třída `IndexOutputStream` implementuje vzorkování na pozicích senzorů definovaných binární maskou, která je reprezentována indexy v doméně. Maska senzorů definována cuboidy je implementována třídou `CuboidOutputStream`. Poslední implementací výstupních streamů je třída `WholeDomainOutputStream` vzorkuje celou doménu.

V případě všech implementací je snaha o minimalizici nadbytečných alokací a znovuvyužívání alokované paměti. V MPI implementaci jsou cuboidy rozděleny na lokální podcuboidy tak, aby byla vzorkována pouze lokální data. CUDA implementace obsahuje zvlášť kernely pro vzorkování a postprocessing v jmenném prostoru `OutputStreamsCudaKernels`.

InputStreams

Modul `InputStreams` se nachází pouze v MPI implementaci. Obsahuje bázovou třídu `BaseInputStream` a implementaci `IndexInputStream`. Ta slouží k průběžnému načítání indexů zdrojů.

Containers

Modul `Containers` obsahuje kontejnery pro všechny matice a streamy potřebné pro běh simulace. Skládá se z tříd `MatrixContainer` a `OutputStreamContainer`.

Třída `MatrixContainer` obsahuje matice simulace, které ukládá v pomocné struktuře `MatrixRecord` do kontejnetru `std::map`. Ta uchovává dodatečná data k matici jako jsou např. jméno matice v HDF5 souboru, její datový typ, rozměry nebo zda je načtena na začátku simulace. V MPI implementaci je navíc specifikováno, jak má být matice distribuována mezi ranky. Kontejner umožňuje hromadnou práci s maticemi — umožňuje jejich vytvoření na základě vstupních parametrů, destrukci, načítání dat z HDF5 souboru apod. CUDA implementace má navíc kontejner `CudaMatrixContainer`, který obsahuje ukazatele na data matic v GPU, určený pro uložení do konstantní paměti GPU. Je tak snížen počet předávaných argumentů při volání kernelů.

Třída `OutputStreamContainer` ukládá vzorkovací streamy stejně jako kontejner matic v kontejneru `std::map`. Na základě vstupních parametrů vytváří požadované streamy a poskytuje rozhraní pro práci s nimi.

MPI implementace navíc obsahuje ještě třídu `InputStreamContainer` implementovanou obdobně jako kontejner vzorkovacích streamů pro vstupní streamy.

KSpaceSolver

Modul `KSpaceSolver` obsahuje v třídě `KSpaceFirstOrderSolver` implementaci simulátoru šíření ultrazvukových vln v médiu podle rovnic podle postupu popsaném v kapitole 4.1. Ke své činnosti využívá všech ostatních modulů. Jeho vnější rozhraní je však vcelku jednoduché.

1. Vytvořením objektu se spustí časovač celkového času simulace.

2. Voláním metody `allocateMemory` jsou jsou v kontejneru matic nejprve vybrány matice potřebné pro výpočet a poté je alokována jejich paměť. Stejně tak je inicializován také objekt vstupních (v MPI implementaci také výstupních) streamů.

3. Metoda `loadInputData` nejprve prostřednictvím objektu kontejneru matic načte všechny potřebné matice. Poté jsou v závislosti na tom, zda je pokračováno z bodu obnovy, načtena data matic, výstupních streamů a dalších parametrů z souboru bodu obnovy.

4. Hlavní výpočetní smyčka je spuštěna metodou `compute`. Na počátku jsou inicializovány FFT plány (FFTW pro OMP a MPI, cuFFT pro CUDA) a následně probíhá preprocessing. V rámci něj jsou nejprve přepočítány indexy z MATLABu do C++ a poté jsou generovány a vypočítány konstanty, operátory a dané matice. Pak je spuštěna hlavní výpočetní smyčka. Ta je z důvodu optimalizace implementována jako šablona, a tak musí být na základě vstupních parametrů vybrána z kontejneru implementací a poté spuštěna. V rámci ní probíhají výpočty jednotlivých kroků simulace a na konci každé iterace probíhá vzorkování na pozicích senzorů. Po dokončení simulace jsou v případě cílového kontrolního bodu uložena data kontrolního bodu, jinak je spuštěn postprocessing, během kterého jsou aplikovány postprocessingové operace na výstupní streamy, následně uložen jejich konečný stav a finální data do výstupního souboru.

5. Destrukcí objektu je uvolněna naalokovaná paměť.

CUDA implementace využívá k výpočtu simulace kernely prostřednictvím třídy `SolverCudaKernels`.

Utils

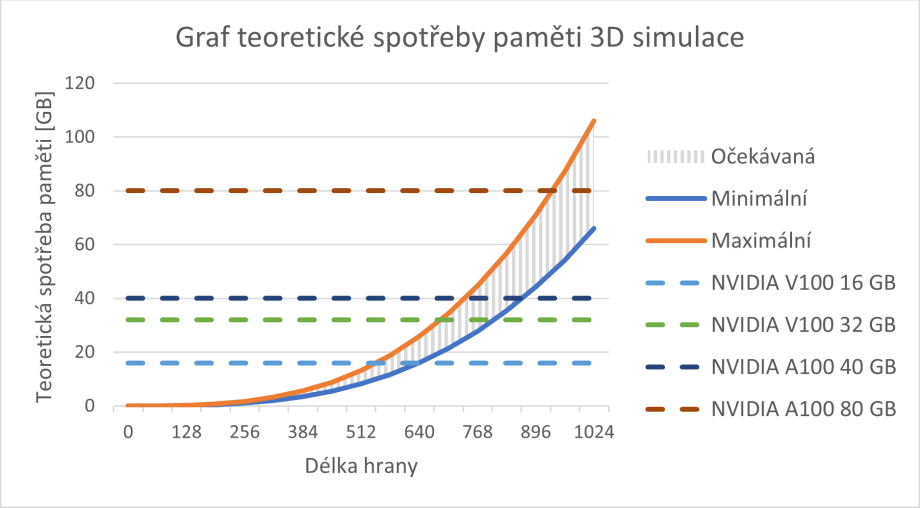
Modul `Utils` obsahuje jednoduché utility. Třída `DimensionSizes` implementuje 4D rozměry a je využívána k nesení dat o rozměrech matic a pro čtení a zápis dat do HDF5 souborů. Třída `TimeMeasure` umožňuje měřit čas simulace a její podčástí. MPI implementace dále obsahuje utility pro práci s MPI a CUDA implementace utility pro CUDA kernely jako jsou funkce pro výpočet indexu vlákna nebo přetížení operátorů.

4.4 Analýza výkonu

V závislosti na parametrech simulované domény se může v simulaci vyskytovat až 22 matic o plných rozměrech domény, 9 matic o redukovaných (komplexních) rozměrech domény a několik dalších 1D matic. Celková spotřeba paměti se pak dá aproximovat pomocí vzorce 4.10, kde $A \in \langle 0, 9 \rangle$, $B \in \langle 0, 2 \rangle$, vstupy reprezentují vstupní data zdrojů a výstupy reprezentují data ukládaná senzory.

$$\text{spotřeba paměti [GB]} \approx \frac{(13+A)\mathrm{N}_x\mathrm{N}_y\mathrm{N}_z + (7+B)\frac{\mathrm{N}_x}{2}\mathrm{N}_y\mathrm{N}_z}{1024^3/4} + \text{vstup} + \text{výstup} \tag{4.10}$$

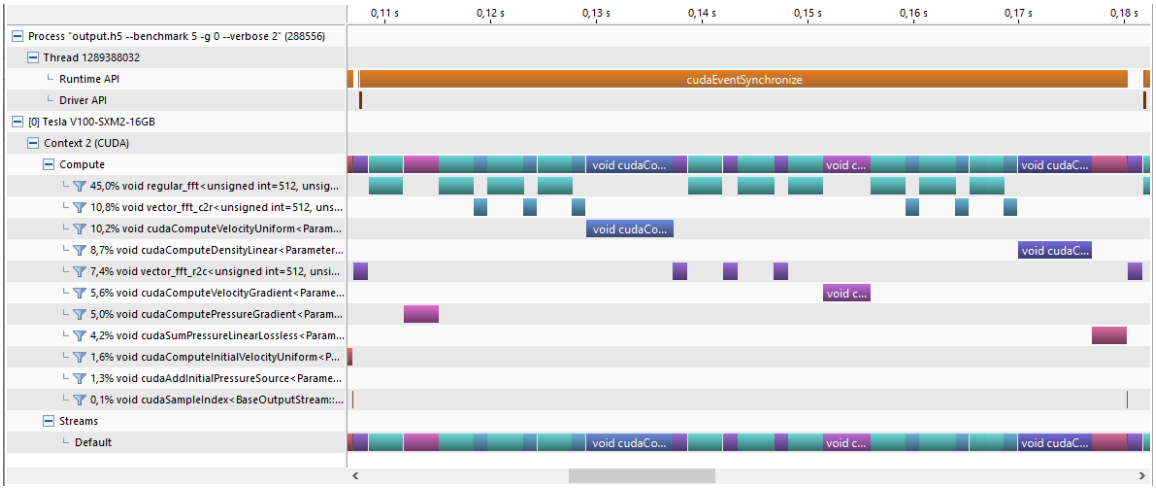
Na jeho základě je možné sestrojit graf 4.1 aproximující spotřebu paměti v závislosti na velikosti simulace a nastavených parametrech. Na grafu lze pozorovat, že výběr parametrů má velký vliv na spotřebu paměti a pro velké simulace může být rozdíl i v desítkách GB. Také je možné vidět, že ani na většinu profesionálních GPU s velkými paměťmi se nevejdou o moc větší simulace než 700^3.



Obrázek 4.1: Graf závislosti teoretické spotřeby paměti na délce hrany 3D simulace a jejích parametrech.

Aby bylo zjištěno, které části výpočtu jsou nejnáročnější, bylo provedeno na implementaci pro GPU provedeno měření profilerem *nvprof* pro simulaci o rozměrech 512^3 v nehomogenním médiu. Z výsledku profilingu na obrázku 4.2 lze vidět, že téměř 64 % času

simulace zabírá výpočet FFT, pouhých asi 36 % času zabírají ostatní výpočty. To znamená, že výsledný výkon simulátoru silně závisí na rychlosti knihovny použité pro výpočet FFT.



Obrázek 4.2: Výstup profileru pro běh simulace a ukázka jednoho simulačního kroku.

Kapitola 5

Testování technologií

Tato kapitola se věnuje testování technologií pro programování více GPU popsané v kapitole 3. Jejím cílem je rozšířit práci Martina Krbily [25], která implementovala simulátor šíření tepla v médiu pomocí vysokoúrovňových jazyků C++ a Python3 pomocí různých technologií na CPU a GPU, o možnost výpočtu na více GPU pomocí knihovny cuFFT, naměřit časy provední simulace a srovnat je s implementací pro jedno GPU.

Simulace v C++ je založena na *Pennes' bioheat equation* [31] implementované v *k-Wave*. Jedná se pouze o zjednodušenou variantu podporující homogenní médium s *k-space* korekcí bez perfuze a tepelných zdrojů. Simulace probíhá výpočtem rovnice změny tepla 5.1 a integrálu poškození tkáně 5.2, kde T je teplota, Δt je časový krok, p hodnota perfuze, q teplo přidané zdrojem, d_1 a d_2 koeficienty difúze, N je celkový počet bodů, k je vlnové číslo, κ je *k-space* korelační koeficient, cem43 je suma integrálu poškození tkáně a f je funkce poškození tkáně.

$$T_{i+1} = \Delta t(p + q + \frac{d_1 d_2}{N}\mathcal{F}^{-1}\{-k^2\kappa\mathcal{F}\{T_i\}\}) \tag{5.1}$$

$$\text{cem43}_{i+1} = \text{cem43}_i + \frac{\Delta t}{60} f(T_{i+1}), f(x) = \begin{cases} 0, & x \leq 37 \\ \frac{1}{4}^{43-x}, & 37 < x < 43 \\ \frac{1}{2}^{43-x}, & 43 \leq x \end{cases} \tag{5.2}$$

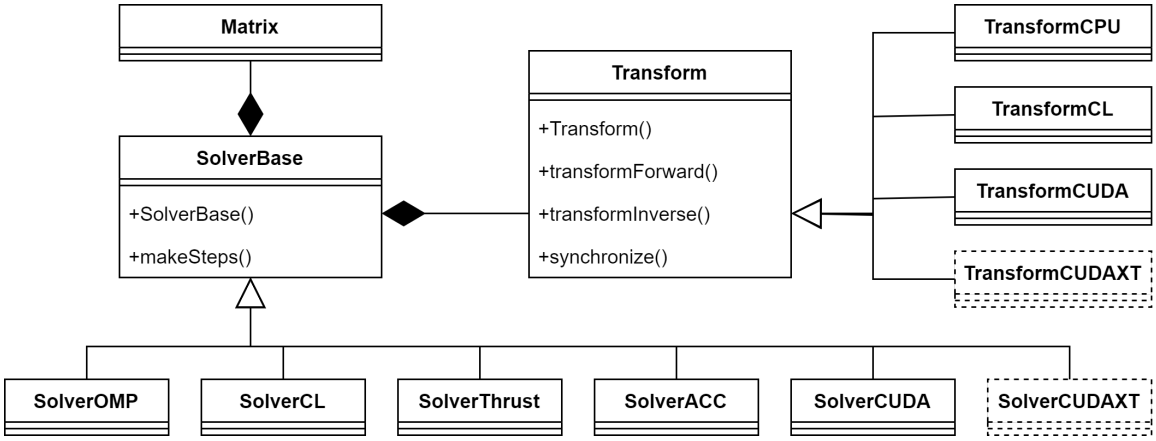
Výpočet má tedy podobnou povahu jako simulace šíření vln v médiu. Lze jej tedy využít k otestování popsaných technologií a to tak, že bude implementována multi-GPU varianta výpočtu s využitím paralelního FFT.

5.1 Stávající implementace

Implementovány jsou výpočty simulace na GPU pomocí několika technologií — *OpenMP*, *CUDA*, *openCL*, *Thrust* a *openACC*. Samotný program má jednoduchou strukturu. Nejprve jsou zpracovány vstupní argumenty, ze kterých je zjištěn vstupní HDF5 soubor a dále která technologie se má k výpočtu použít.

Každá technologie je implementována jako podtřída bázové třídy `SolverBase` (obrázek 5.1). Ta poskytuje rozhraní solveru a implementuje jejich společné metody. Při vytvoření objektu se stará o načítání vstupního souboru a generování κ a k matic. Prostřednitvím abstraktní metody `makeSteps` umožní spustit konkrétní implementaci simulace. Po jejím skončení umožňuje kontrolu správnosti spočítaného výsledku metodou `checkResult`.

Výpočet FFT je implementován pomocí tříd založených na ryze virtuální třídě `Transform`. Ta deklaruje rozhraní pro dopředné a zpěné FFT a synchronizaci. Implementovány jsou transformace na GPU s pomocí OpenCL a cuFFT a na CPU s pomocí FFTW3 knihovny s použitím vláken. Všechny implementace využívají *out-of-place* transformace.



Obrázek 5.1: Diagram tříd C++ verze simulátoru šíření tepla v médiu. Převzato a doplněno z [25].

5.2 Návrh a implementace

Tato podkapitola se věnuje návrhu a implementaci rozšíření stávající implementace o

1. multi-GPU implementaci pro jeden proces s využitím CUDA a knihovny cuFFT,
2. víceprocesovou implementaci na CPU s využitím MPI a knihovny FFTW3 s podporou MPI a
3. víceprocesovou implementaci pro multi-GPU systémy využívající CUDA, CUDA-Aware MPI a knihovny NVSHMEM a cuFFTMp.

Jednoprocesová implementace pro multi-GPU

Multi-GPU implementace pro jeden proces vyžaduje nejmenší zásahy do struktury programu. Jednoduše je přidána další podtřída třídy `SolverBase` a to třída `SolverCUDAXT` využívající vlastní implementaci paralelního FFT v třídě `TransformCUDAXT` (obrázek 5.1). Implementace vychází z CUDA implementace pro 1 GPU, je však podstatně komplikovanější.

Pro jednodušší práci s více GPU byla vytvořena třída `CudaDevices`. Třída je inicializována statickou metodou `initialize`, která nastaví počet GPU, jenž se mají použít k výpočtu, do statické proměnné `sDeviceCount`. Po inicializaci lze používat ostatní statické metody `forEach` provádějící zadanou funkci ve *for* smyčce přes všechna GPU se změnou CUDA kontextu, `getCount` vracející počet vybraných GPU a `synchronize` synchronizující dané streamy přes všechna GPU.

Třída `TransformCUDAXT` implementuje paralelní FFT pomocí knihovny cuFFT. Při vytvoření objektu jsou vytvořeny FFT plány podle aktuálního počtu GPU získaných metodou `CudaDevices::getCount` pro R2C a C2R transformace v *in-place* formátu. Pro ušet-

ření paměti je využito sdílení pracovního prostoru mezi plány. Při provádění FFT v metodách `transformForward` a `transformInverse` třída vybírá, zda-li spustit mutli-GPU nebo single-GPU transformaci. Jelikož multi-GPU transformace využívají speciálního `cudaLibXtDesc` deskriptoru, ale transformace na 1 GPU nikoliv, byla vytvořena pomocná třída `CudaXtDesc`, která poskytuje stejné rozhraní pro oba případy. Implementuje metody pro alokaci `malloc`, destrukci `free` a získání ukazatele na data `getData`.

V třídě `SolverCUDAXT` je implementován výpočet simulace. Při konstrukci objektu je nejprve vytvořen CUDA stream pro každé GPU. Následně je vytvořen objekt pro FFT a jsou inicializovány lokální rozměry a offsety dat na každém GPU podle distribuce dané 1D dekompozicí cuFFT knihovny. Následně je alokována paměť na každém GPU a jsou rozdistribuována příslušná data na každé GPU. V metodě `makeSteps` jsou nejprve spočítány příslušné rozměry mřížky a velikosti bloků CUDA kernelů a následně je spuštěna výpočetní smyčka. Kernely pro výpočty bylo nutné upravit tak, aby pracovaly s *in-place* FFT. Po dokončení výpočtu jsou sesbírány rozdělené výsledné matice na GPU zpět do systémové paměti. V implementaci byly využity asynchronní operace a spouštění kernelů tak, aby mohly probíhát simultárně na všech GPU a byl maximalizován jejich výkon.

V hlavní části programu byla nakonec přidána možnost spustit multi-GPU implementaci jako solver `CUDAXTn`, kde n specifikuje počet GPU, na kterých se má výpočet provést.

Víceprocesová implementace pro CPU

Při přechodu z jednoprocesového a víceprocesový program se podstatně mění celková struktura programu. Aby nedocházelo ke kolizím s ostatními implementacemi, byl pro MPI implementaci vytvořen samostatný program, který ale využívá některé části a logiku jednoprocesové implementace.

V hlavní části programu jsou přidány volání `MPI_Init` a `MPI_Finalize` pro inicializaci a deinicializaci MPI. Textové výpisy, čtení vstupního souboru a porovnávání správnosti řešení jsou prováděny pouze *root* procesem.

Pro provádění paralelního FFT je vytvořena třída `TransformFFTWMp`. Ta při vytvoření objektu inicializuje knihovnu FFTW pomocí `fftw_mpi_init` a následně inicializuje proměnné lokálních rozměrů a offsetů. Jelikož třídá používá pro výběr plánů příznak `FFTW_MEASURE`, jsou před vytvořením plánů alokována pomocná pole pro měření. Pro zvýšení výkonu je přeskočena poslední transpozice u R2C plánu. Dále jsou vytvořeny plány pro transpozici dat, které budou využity pro transpozici ostatních dat v přirozeném formátu do transponovaného.

Simulace probíhá stejně jako v ostatních případech v implementaci třídy solveru `SolverMPI`. Při vytvoření objektu jsou nejprve rozkopírovány skalární hodnoty mezi procesy a následně je inicializován objekt pro FFT. Z něj je získána distribuce dat a jsou inicializovány lokální rozměry a offsety každého procesu. Jelikož se vstupní data nacházejí pouze v *root* procesu, je nutné je rozdistribuovat mezi ostatní procesy. Proto, aby nebyla zasílána nadbytečná data, je vytvořeno několik MPI datových typů:

- `GlobalN0Slab` — *slab* původní domény v nejvyšší dimenzi,
- `GlobalN1Slab` — *slab* původní domény v druhé nejvyšší dimenzi,
- `LocalN0Slab` — *slab* lokální domény v nejvyšší dimenzi,
- `LocalN0SlabPadded` — *slab* lokální domény v nejvyšší dimenzi v odsazeném formátu a

- `LocalN1Slab` — *slab* lokální domény v druhé nejvyšší dimenzi.

Globální *slaby* jsou vytvářeny pouze v *root* procesu, u ostaních by neměly význam.

Po jejich vytvoření je provedena alokace paměti a distribuce dat mezi procesy, která probíhá tak, že je každému procesu rozeslán příslušný počet *slabů*. U matic distribuovaných v druhé nejvyšší dimenzi je nutné provést jejich transpozici. Po dokončení inicializace solveru je možné spustit simulaci. Její struktura je prakticky identická s OpenMP verzí v jednoprocesové implementaci, pouze jsou doplněny bariéry před začátkem a dokončením výpočetní smyčky pro přesné měření času. Poté, co je výpočet dokončen, jsou zpět sesbírány matice na *root* proces. Matice rozdistribuované v druhé nejvyšší dimenzi je třeba opět transponovat zpět do jejich původního tvaru.

Víceprocesová implementace pro GPU

Hlavním důvodem vytvoření víceprocesové implementace pro CPU bylo otestování správnosti funkcionality distribuce dat mezi procesy, protože s využitím CUDA-Aware MPI stačí nahradit alokaci systémové paměti za alokaci paměti na GPU. Samozřejmě již také není třeba provádět transpozici dat distribuovaných v druhé nejvyšší dimenzi, protože knihovna cuFFTMp provádí poslední transpozici vždy. Implementace je umístěna jako další možnost vedle víceprocesová implementace na CPU.

Třída `TransformCUDAMp` implementuje provádění paralelního FFT pomocí knihovny cuFFTMp. Při vytvoření instance této třídy jsou inicializovány FFT plány. Stejně jako v případě `TransformCUDAXT` je využito sdílení pracovního prostoru mezi plány. Jelikož pracovní prostor musí být alokován pomocí knihovny NVSHMEM a velikost alokace z SH musí být stejná u všech procesů, je všemi procesy provedena operace `AllReduce` pro zjištění jeho maximální velikosti. Provádění FFT je implementováno pomocí `cudaLibXtDesc`, které v tomto případě fungují i pro jedno GPU.

Třída solveru `SolverCUDAMp` má podobnou strukturu jako `SolverMPI`. Nejprve je vybráno GPU, se kterým bude proces pracovat a následně jsou inicializovány lokální rozměry, inicializovány MPI datové typy, alokována paměť na GPU a rozdistribuovány matice mezi GPU pomocí CUDA-Aware MPI. Stejně jako v případě `SolverCUDAXT` je implementována *in-place* varianta výpočtu a na jeho konci jsou sesbírány výsledné matice zpět na *root* rank.

Kvůli použití NVSHMEM knihovny byla přidána v hlavní části programu její inicializace z MPI komunikátoru všech procesů a deinicializace. Rovněž byl přidána možnost specifikovat výpočet této implementace výběrem `CUDAMp` solveru.

5.3 Výsledky měření

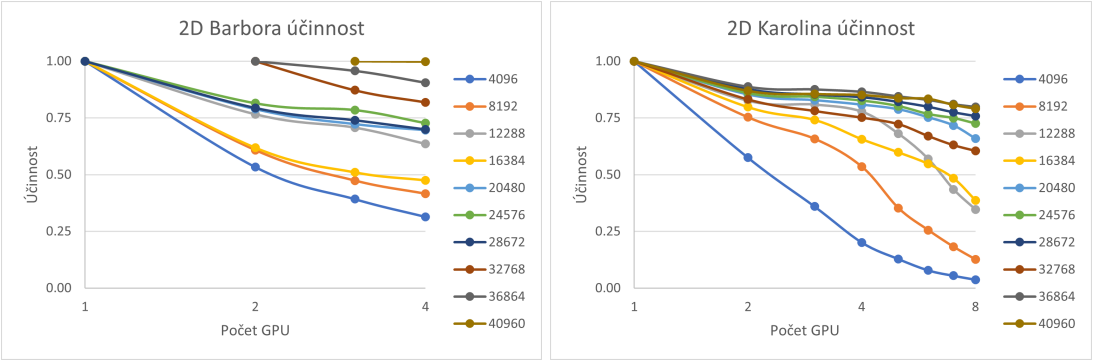
Testování probíhalo na 2D a 3D vstupech o různých rozměrech simulace, jejichž řešení bylo spuštěno se solverem `SolverCUDAXT` na superpočítačích Barbora a Karolina. Měřen byl pouze čas výpočtu bez počátečních transferů dat. Pro každý běh bylo provedeno 300 iterací simulace. Byly vytvořeny grafy silného škálování, zrychlení a účinnosti umístěné v příloze A. Solver `SolverCUDAMp` bohužel otestován nebyl z důvodu nekompatibility knihoven na superpočítačích.

2D simulace

Z naměřených výsledků 2D simulací v 5.2 lze na první pohled konstatovat, že novější hardware na Karolině je na jednom i více GPU rychlejší nezávisle na velikosti simulace. Jelikož

GPU na Karolině mají i větší paměť, bylo na nich možné spustit větší simulace na menším počtu GPU. Obecně lze říci, že čím větší úloha je počítána, tím je vyšší je zrychlení i účinnost výpočtu na více GPU.

Pro menší úlohy spuštěné na 2 GPU na Barboře účinnost rychle klesá na asi 55 %, pro 3 a více GPU pak pod 50 %. Pro větší úlohy se účinnost pohybuje okolo 80 %. Nejvyššího zrychlení 2,9x bylo dosaženo pro simulaci o délce hrany 24576 při použití 4 GPU. Na Karolině se pro většinu simulací na 2-4 GPU pohybuje účinnost mezi 80 a 90 %. Výjimkou jsou malé simulace, pro které účinnost rychle klesá a to ještě rychleji než na Barboře. Pro 5-8 GPU se účinnosti snižují na hodnoty okolo 75 %. Na Karolině bylo dosaženo nejvyššího zrychlení 6,4x pro simulaci o délce hrany 36864.



(a) Barbora

(b) Karolina

Obrázek 5.2: Grafy účinností 2D simulací šíření tepla v médiu. Rozměr značí délku hrany simulace.

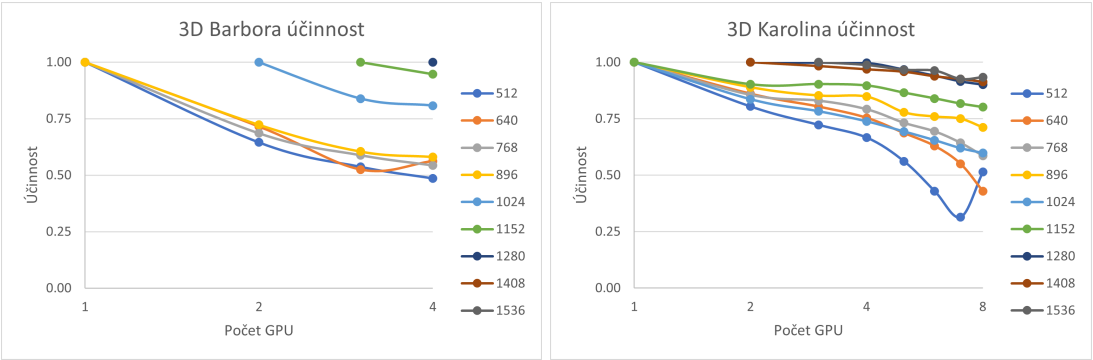
3D simulace

Stejně jako v případě 2D simulace i u výsledků 3D simulace 5.3 jsou hodnoty naměřené na Karolině výrazně lepší. Kvůli menší GPU paměti nebylo možné větší simulace na Barboře vůbec spustit. Většina simulací na Barboře má účinnost na 2 GPU okolo 70 %, pro více GPU se pak snižuje na hodnoty mezi 50 a 60 %. Nejvyššího zrychlení 2,35x dosahuje pro simulaci o délce hrany 896.

Hodnoty účinností pro Karolinu se pro všechny simulace na 2 GPU pohybují kolem 80 % a pro více GPU klesají. Čím větší rozměry simulace jsou, tím pomaleji účinnosti klesají. Nejvyššího zrychlení 6,4x bylo naměřeno pro simulaci o délce hrany 1152.

Závěr

Z naměřených výsledků vyplývá, že je pokud jde uživateli o co nejnižší cenu výpočtu, je nejvýhodnější použít výpočet na 1 GPU. V případě, že se data na GPU nevejdou nebo je cílem získat výsledky co nejdříve, je důležité volit co nejnovější hardware. Jako ideální volbou se jeví 2 GPU, na Karolině i 3 nebo 4 GPU v závislosti na velikosti simulace. Důležité je, pokud je možné se tomu vyhnout, nepoužívat větší množství GPU na malé simulace.



(a) Barbora

(b) Karolina

Obrázek 5.3: Grafy účinností 3D simulací šíření tepla v médiu. Rozměr značí délku hrany simulace.

Kapitola 6

Návrh a implementace

V této kapitole jsou popsány návrh a implementace simulátoru umožňující zapojení více GPU do výpočtu. Nejprve jsou definovány cíle, kterých by implementace měla dosáhnout, po nich následují technologie, jenž budou k řešení využity, a poté jsou popsány jednotlivé moduly programu.

6.1 Definice cílů

Cílem je navrhnout simulátor založený na popsaných implementacích splňující tyto podmínky.

1. **Multi-GPU** — umožnění zapojení několika NVIDIA GPU do výpočtu s podporou stejné funkcionality jako stávající CUDA implementace,
2. **Nízká spotřeba paměti** — jelikož je výpočet náročný na paměť, měly by být omezeny alokace pro nepotřebná data.
3. **Přenositelnost** — výsledný program by měl být spustitelný různých operačních systémech a být přeložitelný různými kompilátory.
4. **Zpětná kompatilibila** — měl by být podporován i starší harware.
5. **Dostupnost** — při výběru technologií je třeba brát i ohled na to, aby byly závislosti běžně dostupné, a případně snadno nainstalovatelné.
6. **Škálovatelnost** — výsledná implementace by měla vykazovat dobré hodnoty škálovatelnosti s přibývajícími GPU. Účinnost pro 4 GPU by neměla být nižší než 50 % na testovaných zařízeních.

Dále jsou definovány volitelné cíle:

- sloučení OMP a CUDA implementací a poskytnutí jedné implementace umožňující provádět výpočet na CPU nebo 1 a více GPU.

6.2 Technologie

Původní implementace byla napsána v jazyce C++11, nicméně v této implementaci bude použit standard C++20, který je nyní již podporován většinou překladačů a také překladačem NVCC v CUDA Toolkit verze 12.0. K načítání a ukládání dat bude použito C a HL

rozhraní knihovny HDF5. Pro provádění FFT v implementaci pro CPU bude použita stejně jako v původní implementaci knihovna FFTW3 s využitím *multithreadingu*. Pro výpočet FFT na GPU bude použita knihovna cuFFT, u které bude kombinováno rozhraní pro 1 a pro více GPU.

6.3 Hlavní principy

Jednou z hlavních myšlenek návrhu by měla být oddělitelnost CPU a GPU implementace. Základem celého simulátoru by měla být implementace pro CPU a možnost provádět výpočet na GPU její nádstavbou. V případě tříd tak bude vytvořena verze pro CPU a verze pro GPU, která z ní bude dědit. Např. pro třídu `Class` a vznikne její CUDA rozšíření `CudaClass`. Implementace pak bude z velké části využívat *polymorfismu*.

Jelikož je výpočet simulace hlavně omezován dostupnou pamětí, bude třeba věnovat zvýšenou pozornost, jakým způsobem budou matice distribuovány mezi jednotlivá GPU. Pokud to není bezpodmínečně nutné, měly by být matice rozdistribuovány v některé z dimenzí, nikoliv celé rozkopírovány. Bude tak omezena spotřeba paměti. Distribuce dat bude záviset hlavně na distribuci dat knihovny cuFFT. Ta distribuuje data buď v nejvyšší dimenzi pro reálnou doménu, nebo v druhé nejvyšší dimenzi pro spektrální doménu. U každé matice pak bude třeba rozhodnout, o jaký případ se jedná, a podle toho s ní pracovat. Speciálním případem budou matice pro zdroje a vzorkování, jejichž distribuce závisí na jejich pozici. Pro ně bude třeba vytvořit jiné distribuovací mechanismy.

Návrh by neměl příliš měnit původní strukturu programu, pokud to nebude nutné. Bude tak usnadněn přenos některých částí původní implementace do nové.

6.4 Překlad programu

K usnadnění překladu programu je vytvořen *cmake* skript, který vyžaduje minimální verzi programu *CMake 3.25.2*. K nalezení potřebných knihoven využívá skript program *pkg-config*, který je nutné mít rovněž mít v systému nainstalován.

Pro překlad CPU verze je nutné mít na zařízení nainstalovaný kompilátor s podporou C++20 a OpenMP 4.5. Dále jsou vyžadovány HDF5 knihovna s C a HL rozhraním, knihovna OpenMP s rozhraním pro C++, knihovna FFTW3 s podporou *multithreadingu* pro OpenMP a knihovny *libz*, *libm* a *limvec*.

K překladu programu s podporou výpočtu na GPU je třeba nastavit příznak `ALLOW_CUDA` a mít nainstalovaný CUDA Toolkit verze 12.0 a vyšší. Je také nutné používat překladač podporovaný CUDA Toolkitem.

6.5 Modul Cuda

Modul `Cuda` obsahuje třídy a další utility usnadňující a zapouzdřující práci s CUDA zařízeními. Toto obalení umožní v případě změny v CUDA rozhraní jednoduše změnit implementaci na jednom místě a také psát bezpečnější kód, jelikož je v implementaci pracováno s více CUDA kontexty. Sníží se tak pravděpodobnost jejich zaměnění. Navíc bylo možné si přizpůsobit rozhraní potřebám implementace.

Hlavní částí modulu je statická třída `CudaDevices`. Ta poskytuje rozhraní pro hromadnou práci s CUDA zařízeními. Před použitím jakéhokoliv rozhraní z modulu je třeba tuto třídu inicializovat metodou `initialize`. Ta na základě poskytnutých *id* vybere zařízení, se

kterými se bude pracovat, a provede jejich inicializaci. Dále se v programu již *id* zařízení nepoužívají, místo toho má každé vybrané zařízení svůj jedinečný index. Po provedení inicializace je již možné používat ostatní metody třídy. Metoda `forEach` provede ve smyčce *for* funkci pro každé zařízení se změnou kontextu. Je to jediné místo v programu, kde je změna kontextu umožněna. Ostatní metody umožňují např. hromadné spuštění CUDA kernelu na všech GPU nebo synchronizaci streamů nebo událostí a jejich vytvoření.

V programu často nastává případ, kdy je třeba uchovat jedinečnou hodnotu pro každé zařízení. Z toho důvodu vznikla šablonovaná třída `CudaDeviceEntry`, která má počet prvků vždy stejný jako je počet zařízení. Před prvním jejím použitím však musí být provedena inicializace CUDA zařízení.

Pro reprezentaci CUDA zařízení byla vytvořena třída `CudaDevice`. Přístup k této třídě je umožněn pouze skrze metodu `forEach` třídy `CudaDevices`. Třída zapouzdřuje práci s GPU, umožňuje spravovat jeho paměť (kontantní i globální), spouštět CUDA kernely nebo vytvářet události a streamy. Události i streamy jsou při vytvoření uloženy do interní struktury a navráceny jsou pouze jejich *id*. Toto opatření bylo implementováno z toho důvodu, že streamy i události jsou aktuální jenom pro CUDA kontext, ve kterém byly vytvořeny. Při jejich použití v jiném kontextu by došlo k chybě. Třída také nese spouštěcí konfigurace kernelů `CudaKernelConfig`.

K zapouzdření práce s CUDA rozhraním pro systémovou paměť bude sloužit třída `CudaHost`. Umožňuje alokaci a dealokovat *pinned* paměť nebo registrovat již dříve alokovanou paměť. Pro utility používané v CUDA kernelech slouží modul `CudaKernelUtils`. Obsahuje přetížení operátorů a metody pro získání 1D indexů a strideů vláken. Ke kontrole návratových kódů volání CUDA a cuFFT rozhraní byla vytvořena třída `CudaError`.

6.6 Modul MatrixClasses

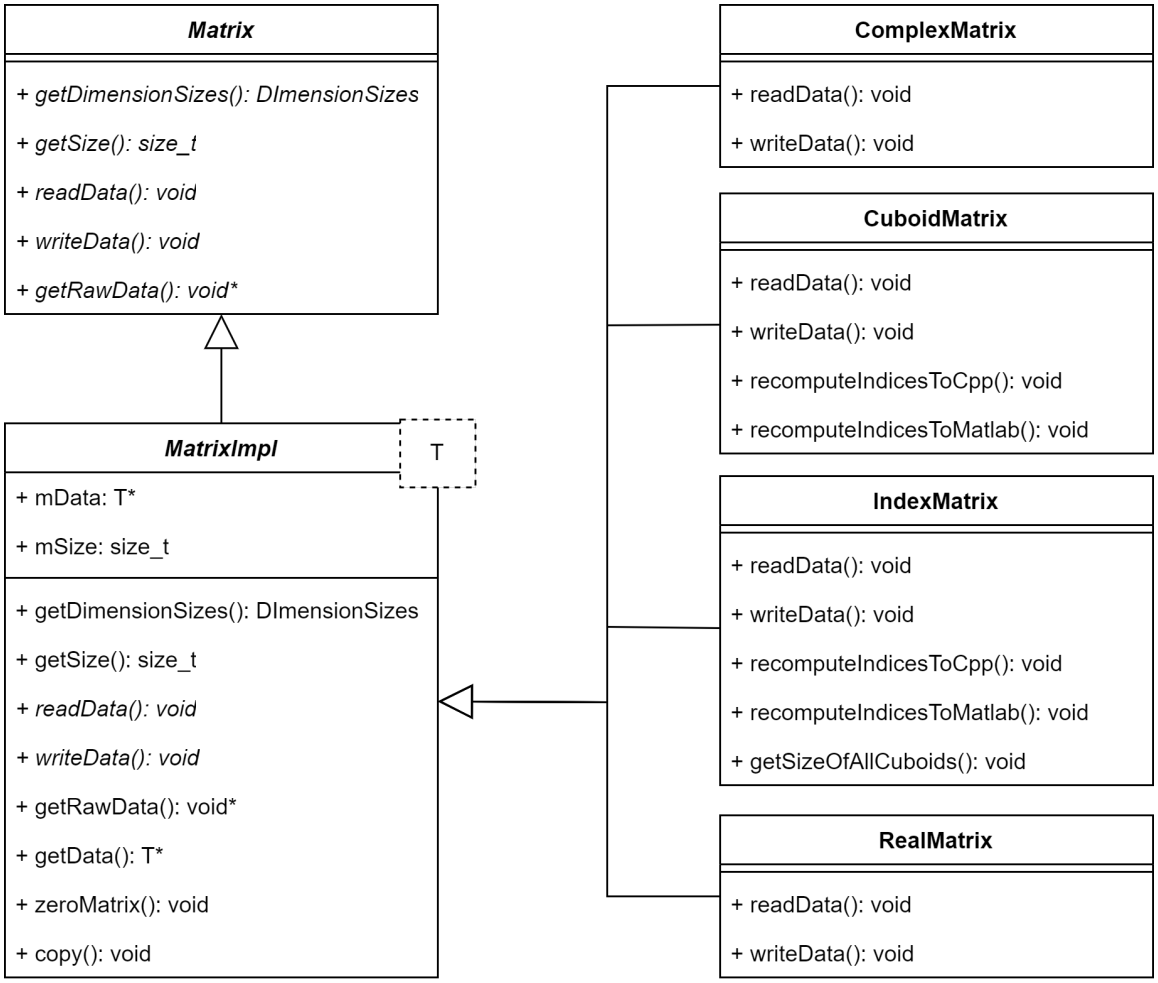
Modul `MatrixClasses` prošel zásadní změnou. Podobně jako v původní implementaci bylo využito vrstvení funkcionality pomocí dědičnosti, avšak oproti ní využívá i šablony. To zamezuje multiplikování stejného kódu a zlepšuje přehlednost kódu.

Bázovou třídou všech matic je abstraktní třída `Matrix` (obrázek 6.1). Definuje jejich společné rozhraní pro získání rozměrů matice, *raw* ukazatele na data, načtení dat ze vstupního souboru a zápis do výstupního souboru.

Na jejím základě je vytvořena šablonovaná třída `MatrixImpl`, která implementuje zděděné abstraktní metody pro získání rozměrů matice a *raw* ukazatele na data. Z důvodu budoucího rozšíření o CUDA verze matic je využita virtuální dědičnost. Třída dále implementuje alokaci a dealokaci paměti, metody pro získání dat, vynulovaní matic, kopírování a pro datové typy v plovoucí řádové čárce také metodu pro podělení hodnoty každého prvku matice inverzní hodnotou skaláru. Třída neimplementuje metody pro čtení ani zápis do souboru kvůli nekonzistenci práce s HDF5 soubory implementovaném v modulu `Hdf5`.

V další vrstvě dědičnosti již vznikají kompletní implementace matic v třídách `RealMatrix`, `ComplexMatrix`, `IndexMatrix` a `CuboidMatrix`. Tyto třídy implementují čtení a zápis dat do HDF5 souborů a své specifické metody. Třídy `CuboidMatrix` a `IndexMatrix` tak implementují metody pro transformaci indexů mezi formáty používanými v MATLABu a C++. `CuboidMatrix` navíc ještě poskytuje metodu pro zjištění sumy objemů všech cuboidů.

Pro třídy podporující FFT jsou navíc vytvořeny třídy `FftRealMatrix` a `FftComplexMatrix` dědící z tříd `RealMatrix` a `ComplexMatrix`. K transformaci používají plány z třídy `FftPlanContainer` (viz dále 6.8). Pro matice závisící na vstupní indexové matici je vytvořena šablonovaná třída `IndexedMatrix`.



Obrázek 6.1: Diagram tříd modulu matic pro CPU implementaci.

CUDA implementace

Třídy matic budou v CUDA implementaci rozšířeny o data GPU. Bázovou třídou CUDA matic je abstraktní třída `CudaMatrix`. Ta virtuálně dědí z třídy `Matrix` a rozšiřuje ji o metody pro získání velikosti dat na konkrétním GPU, ukazatele na *raw* data na GPU, vynulování matic na GPU a kopírování mezi systémovou pamětí a pamětí GPU.

Pro CUDA matice uložené v systémové paměti je vytvořena šablonovaná třída `CudaHostMatrix`. Parametr šablony je pomocí C++20 konceptů omezen na třídy založené na třídě `Matrix`. Třída dědí z `CudaMatrix` a matice poskytnuté šablonovým parametrem. Hlavní smysl této třídy je registrování *pinned* paměti matic v systémové paměti při vytváření objektu.

Poslední vrstvou dědičnosti jsou šablonované třídy implementující distribuování dat mezi jednotlivá GPU. V šablonovém parametru je očekávána implementace třídy `CudaHostMatrix`. Třída `CudaBroadcastMatrix` implementuje matici, která je celá zkopírována na každé GPU. U této třídy není implementování kopírování dat zpět do systémové paměti, jelikož se v implementaci nepoužívá. Nicméně by bylo možné tuto metodu implementovat např. pomocí knihovny NCCL. Třída `CudaNaturalScatterMatrix` implementuje rozdistribuovanou matici mezi GPU v nejvyšší dimenzi odpovídající distribuci dat při použití subformátu

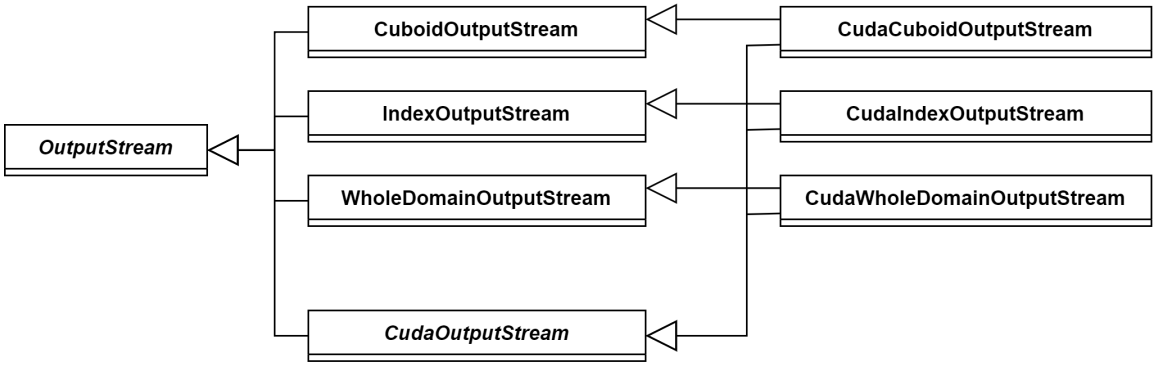
INPLACE knihovny cuFFT. Naproti tomu třída CudaShuffledScatterMatrix implementuje rozdistribuovanou matici v druhé nejvyšší dimenzi odpovídající formátu INPLACE_SHUFFLED. Každá třída implementuje mimo metod pro kopírování také metody pro přístup k datům a jejich lokálním velikostem.

Pro distribuovanou indexovou matici je vytvořena třída CudaScatterIndexMatrix, která nerovnoměrně distribuuje data mezi GPU tak, aby každé GPU obsahovalo jenom své vlastní indexy. Pomocná třída CudaScatterIndexedMatrix pak slouží k distribuci dat závisející na dané indexovací matici.

Pro komplexní matice podporující FFT je vytvořena třída CudaScatterFftComplexMatrix. Ta využívá pro uchování dat instanci třídy CudaFftDescriptor. Je implementována *in-place* transformace.

6.7 Modul OutputStreams

Modul OutputStreams vychází z původního návrhu, neimplementuje však vlastní buffery pro ukládaní dat. Místo toho využívá již implementovaných matic z modulu MatrixClasses. Redukční operátory jsou stále implementovány pomocí výčtového datového typu kvůli problematickému předávání ukazatelů na funkce do CUDA kernelů.



Obrázek 6.2: Zjednodušený diagram tříd modulu OutputStreams znázorňující heirarchii dědičnosti mezi třídami.

Základem všech výstupních streamů je bázová třída OutputStream (obrázek 6.2). Stejně jako její předchůdce BaseOutputStream definuje jednotné rozhraní výstupních streamů, implementuje jejich postprocessing a nese jejich společná data. Rozhraní umožňuje vytvoření, znovuotevření, zavření streamů a spuštění samplování.

Další vrstvou v dědičnosti jsou konkrétní implementace výstupních streamů pro CPU implementaci v třídách CuboidOutputStream, IndexOutputStream a WholeDomainOutputStream. U všech tříd je nutné použít virtuální dědičnost. Každá třída implementuje zděděné virtuální metody pro daný typ streamu.

CUDA implementace

Třída CudaOutputStream je bázovou třídou výstupních streamů určených pro CUDA implementaci. Virtuálně dědí z třídy OutputStream. Definuje nové ryze virtuální metody pro kopírování z a na GPU a implementuje vlastní postprocessing, který probíhá na GPU.

Implementace výstupních streamů se bude nachází v třídách `CudaCuboidOutputStream`, `CudaIndexOutputStream` a `CudaWholeDomainOutputStream`. Všechny dědí ze svých ekvivalentních tříd pro CPU implementaci a třídy `CudaOutputStream`. Implementují vlastní metody pro vytvoření, znovuotevření a uzavření streamů a metodu pro vzorkování. Dědí také z třídy `CudaOutputStream` a implementují zděděné ryze virtuální metody pro kopírování dat z a na GPU.

6.8 Modul Containers

Modul `Containers` implementuje třídy spravující matice a výstupní streamy použité v simulaci. Oproti původní implementaci je hierarchie tříd složitější, ale bylo tak umožněno oddělit implementace pro CPU a pro GPU.

Třída `FftPlanContainer` slouží ke správě FFT plánů knihovny FFTW pro CPU implementaci, které na základě daných rozměrů vytvoří, a implementuje metody pro přístup k nim. Pro GPU implementaci je vytvořena třída `CudaFftPlanContainer`, která třídu `FftPlanContainer` rozšiřuje o plány knihovny cuFFT a mimo ně se stará o alokaci a dealokaci jejich pracovních prostorů.

Pro uchování matic potřebných pro výpočet je vytvořena třída `MatrixContainer`. Obsahuje rozhraní pro hromadné operace jako jsou výběr potřebných matic, vytvoření matic, načtení dat matic ze vstupního souboru a jejich uložení do vstupního nebo kontrolního souboru. Také obsahuje metody pro přímý přístup k jednotlivým maticím na základě jejich *id*. V CUDA implementaci je třída rozšířena třídou `CudaMatrixContainer`, která pracuje s CUDA maticemi. Je doplněna o metody pro kopírování matic z a do pamětí GPU. Pro každý záznam matice také uchovává informaci o tom, jak má být matice distribuována mezi GPU. V třídě `CudaMatrixSizes` jsou pak uchovány rozměry všech matic, a to globální i lokální pro každé GPU. Pro rychlejší přístup k datům jsou na každém GPU uloženy rozměry matic a ukazatele na lokální data matic v konstantní paměti.

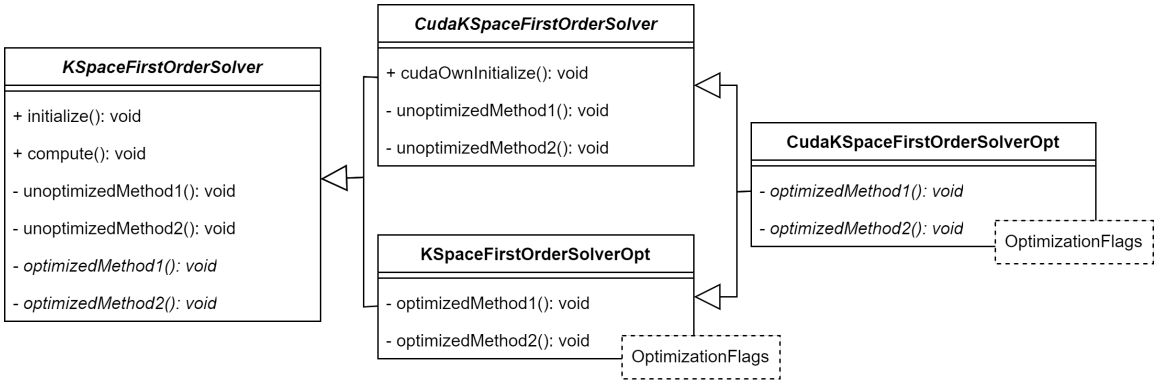
Kontejner pro výstupní streamy je implementován třídou `OutputStreamContainer`. Umožňuje jejich inicializaci a hromadné operace — vytvoření, znovuotevření, uzavení, postprocessing a další. Pro CUDA implementaci je třída rozšířena třídou `CudaOutputStreamContainer`, která vytváří výstupní streamy určené pro GPU a poskytuje jejich rozšířené hromadné rozhraní.

6.9 Modul KSpaceSolver

Modul solveru se je stejně jako všechny ostatní moduly dělí na implementaci pro CPU a její rozšíření pro GPU. Podobně jako v původní implementaci je zachována optimalizace metod výpočtu pomocí šablonovaných tříd, nicméně je provedena trochu odlišně.

Solver pro CPU se skládá z třídy `KSpaceFirstOrderSolver` a šablonované třídy `KSpaceFirstOrderSolverOpt` (obrázek 6.3). Třída `KSpaceFirstOrderSolver` implementuje vnější rozhraní a neoptimalizované metody. Dále virtuálně definuje optimalizované metody solveru, které jsou implementovány třídou `KSpaceFirstOrderSolverOpt`, jenž z ní virutálně dědí.Jejím šablonovým parametrem je struktura `OptimizationFlags`. Vytvoření objektu pak probíhá pomocí statické metody `create`, která na základě parametrů programu vytvoří objekt optimalizovaného solveru a vrátí ukazatel na jeho nešablonovanou verzi. Vnější rozhraní solveru umožňuje jeho inicializaci, načtení vstupních dat, spuštění výpočtu, zjiš-

tění časů výpočtu a tisk informací o něm. Privátní metody mají stejné rozhraní jako u OMP implementace.



Obrázek 6.3: Návrh struktury tříd solverů.

Implementace solveru pro GPU má podobnou strukturu jako solver pro CPU, na kterém je založena. Také se bude skládá z neoptimalizované části v třídě `CudaKSpaceFirstOrderSolver` a optimalizované části implementované v šablonované třídě `CudaKSpaceFirstOrderSolverOpt`. Třída `CudaKSpaceFirstOrderSolver` virtuálně dědí z `KSpaceFirstOrderSolver` a rovněž implementuje statickou metodu `create` s obdobnou funkcionalitou pro CUDA implementaci. Také implementuje vlastní tisk informací o solveru. Oproti implementaci pro CPU navíc obsahuje metody k inicializaci konfigurací kernelů a konstant. Pro rychlejší běh výpočetních kernelů jsou konstanty simulace uloženy v konstantní paměti v třídě `CudaConstants`. Šablonovaná třída `CudaKSpaceFirstOrderSolverOpt` implementuje veškeré výpočty simulace na GPU s výjimkou generování matic, které je prováděno na CPU.

6.10 Ostatní moduly

Tato podkapitola popisuje návrh a implementaci modulů `Hdf5`, `Logger`, `Parameters` a `Utils`, u kterých nejsou změny oproti původní implementaci tak výrazné.

Hdf5

V modulu `Hdf5` nedošlo k žádným změnám v rozhraní jeho tříd. Jedinou změnou je odstranění závislostí na konkrétním operačním systému při zjištění existence souboru a místo toho jsou použity funkce z knihovny *std::filesystem* definované standardem C++17.

Logger

Modul `Logger` také neprošel žádnými většími změnami. Z třídy `Logger` jsou odstraněny metody pro kontrolu návratových hodnot CUDA volání a tato funkcionalita je přesunuta do třídy `CudaError`. Dále jsou upraveny některé logovací výpisy tak, aby odpovídaly multi-GPU implementaci.

Parameters

V porovnání s původní implementací ubyla v modulu `Parameters` funkcionalita a to hlavně z důvodu přesunutí části jeho funkcí do modulů `Containers` a `Cuda`. Nyní se skládá z tříd

`CommandLineParameters` a `Parameters`. V třídě `CommandLineParameters` došlo ke změnám ve zpracování vstupních argumentů tak, aby bylo možné specifikovat více GPU parametru `-g`. Třída `Parameters` pak poskytuje rozhraní pro přístup k příznaku spuštění na GPU a případně také *id* vybraných CUDA zařízení.

Utils

Modul `Utils` obsahuje v původním stavu třídy `DimensionSizes` a `TimeMeasure`. K nim přibyla třída `Tile` reprezentující dlaždici matice. Nese její 3D offset a velikost. Modul dále obsahuje definice datových typů používaných pro výpočet `Real`, `Complex`, `Index` a `Cuboid`. Pro funkce poskytující systémové informace a zarovnanou alokaci paměti je vytvořena třída `System`. Ostatní utility se nacházejí v třídě `Utils`.

Kapitola 7

Dosažené výsledky

Tato kapitola popisuje testování vzniklého programu, které bylo provedeno na superpočítači Karolina. Superpočítač Barbora nemohl být použit z důvodu nekompatibilní verze nainstalovaných CUDA ovladačů. Testování bylo provedeno na 2D a 3D vstupech dvou kombinací parametrů, které se v *k-Wave* používají nejčastěji.

2D simulace byly testovány pro délky hrany 4096 až 40960 s krokem 4096. 3D simulace pro délky hrany 128 až 1152 s krokem 128. Pro každou velikost bylo simulováno 100 simulačních kroků.

Simulace s homogenním médiem

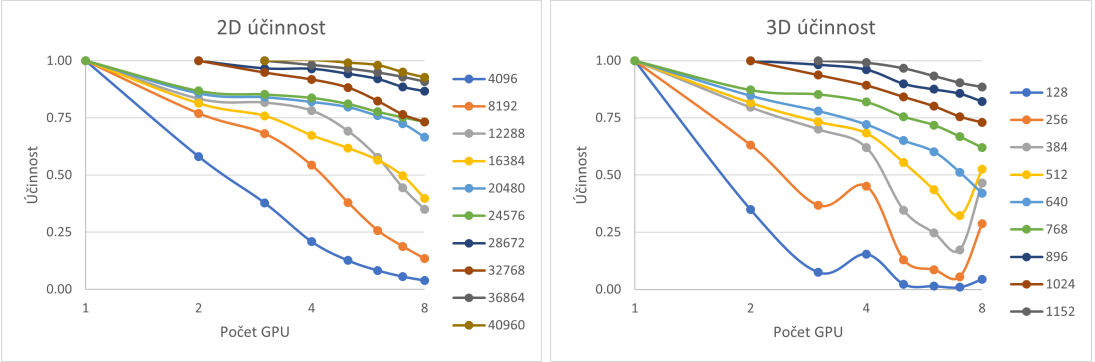
Simulace s homogenním médiem je nejjednodušší simulací v *k-Wave*. Parametry simulace jsou definovány jedinou hodnotou a tak jsou vytvářeny relativně malé vstupní soubory. Rovněž vyžadují menší paměťový prostor na GPU a umožňují provádět větší simulace. Kompletní výsledky lze najít v příloze B.

Na naměřených výsledcích na obrázku 7.1 lze na první pohled vidět, že výsledky jsou obdobné jako u testování termální simulace. Výsledky 2D simulací měřené pro 2 GPU dosahují účinnosti kolem 80 % s výjimkou nejmenší velikosti, jejíž účinnost rychle klesá. Podobných výsledků je dosaženo také pro 3 a 4 GPU. Poté se účinnost pozvolna snižuje. Nejvyššího zrychlení 5,86x bylo naměřeno pro doménu o délce hrany 24576 při využití 8 GPU.

Z výsledků 3D simulací lze vyčíst, že účinnost s přibývajícími GPU klesá rychleji než u 2D simulací. Je patrné, že provádět 3D simulace s menší délkou hrany než 384 bodů na více GPU prakticky nemá smysl. Při použití 2 GPU se účinnost u většiny simulací drží na 80 %, pro 3 GPU už to je asi 75 % a pro 4 GPU něco přes 70 %. Maximálního zrychlení 5x je dosaženo pro simulaci o délce hrany 768.

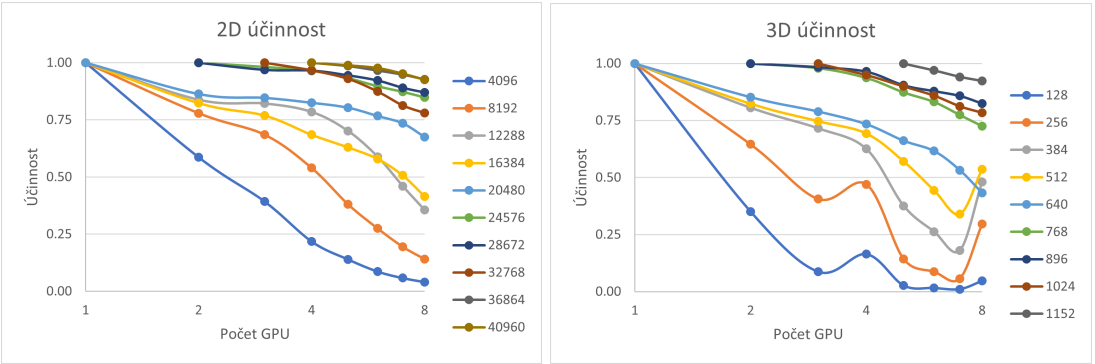
Simulace s heterogenním médiem

Testované médium má heterogenní rychlost šíření zvuku, hustotu, absorpci i nelineární výraz. Jedná se o nejsložitější vstup simulace bez zdrojů. Velikosti vstupních souborů jsou několikanásobně větší než u homogenního média. Také paměť potřebná při výpočtu je o mnoho větší, a tak bylo možné naměřit menší simulace než v případě homogenního média. Kompletní výsledky měření je možné najít v příloze B.2. Výsledky 2D i 3D simulací na obrázku 7.2 jsou prakticky identické s výsledky homogenní simulace.



(a) 2D (b) 3D

Obrázek 7.1: Grafy účinností simulací šíření akustických vln v homogenním médiu. Rozměr značí délku hrany simulace.



(a) 2D (b) 3D

Obrázek 7.2: Grafy účinností simulací šíření akustických vln v heterogenním médiu. Rozměr značí délku hrany simulace.

Kapitola 8

Závěr

Tato práce se věnovala akceleraci výpočtu simulace šíření akustických vln v médiu na multi-GPU systémech. Popisovala jejich strukturu, představila řadu technologií pro práci s nimi a na základě získaných znalostí navrhla a implementovala akcelerovaný simulátor, který podporuje výpočet na CPU nebo na jednom a více GPU. V rámci testování také vznikla multi-GPU implementace simulace šíření tepla v médiu s využitím knihoven cuFFT a cuFFTMp.

Měření přinesla u obou implementací zajímavé výsledky a to hlavně pro střední a velké simulace, na které tato práce cílila. Bylo zjištěno, že používání hardware novější architektury na superpočítači Karolina přináší o mnoho lepší výsledky než starší hardware na superpočítači Barbora. Nezávisle na testovaném hardware i dimenzionality simulace bylo nejefektivnějších výsledků dosaženo při použití 2 GPU pro výpočet. Dobré výsledky s většími rozměry simulací a novějším hardware přinesly také měření na 3 a 4 GPU. Pro velké simulace se může vyplatit použít i 5 nebo více GPU, avšak je třeba počítat s tím, že hodnoty účinnosti pomalu klesají s každým dalším GPU.

Hlavním přínosem této práce je umožnění provádění velkých simulací na GPU a naměření hodnot silného škálování.

Další práce

Jako další pokračování práce by bylo možné implementovat multi-GPU simulátor šíření vln v médiu s použitím MPI a knihovnou cuFFTMp a sloučit všechny implementace akcelerovaného simulátoru do jedné. Bylo by také zajímavé naměřit výsledky na nejnovějších kartách NVIDIA H100, které používají NVLink 4. generace.

Literatura

[1] *FFTW 3.3.10*. Dostupné z: https://www.fftw.org/fftw3_doc/.

[2] NVIDIA Collective Communication Library Documentation. *NCCL Documentation*. NVIDIA. 2020. Dostupné z: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/index.html.

[3] NVIDIA V100 TENSOR CORE GPU. *NVIDIA V100S Datasheet*. NVIDIA. Jan 2020. Dostupné z: https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf.

[4] NVIDIA, 2022. Dostupné z: https://docs.nvidia.com/hpc-sdk/cufftmp/index.html.

[5] CuFFT API Reference. *CUDA Documentation*. NVIDIA. 2022. Dostupné z: https://docs.nvidia.com/cuda/cufft/.

[6] NVIDIA A100 TENSOR CORE GPU. *NVIDIA A100 | Tensor Core GPU*. NVIDIA. May 2022. Dostupné z: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-nvidia-us-2188504-web.pdf.

[7] NVIDIA DGX-2. *NVIDIA DGX-2*. NVIDIA. May 2022. Dostupné z: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/dgx-1/dgx-2-datasheet-us-nvidia-955420-r2-web-new.pdf.

[8] *NVIDIA OpenSHMEM Library (NVSHMEM) Documentation*. 2022. Dostupné z: https://docs.nvidia.com/nvshmem/api/index.html.

[9] *Compute Nodes*. IT4Innovations, Jan 2023. Dostupné z: https://docs.it4i.cz/barbora/compute-nodes/.

[10] *Compute Nodes*. IT4Innovations, Jan 2023. Dostupné z: https://docs.it4i.cz/karolina/hardware-overview/.

[11] *HDF5 Support*. May 2023. Dostupné z: https://portal.hdfgroup.org/display/HDF5/HDF5.

[12] *NVIDIA DGX-2*. IT4Innovations, Jan 2023. Dostupné z: https://docs.it4i.cz/dgx2/introduction/.

[13] Belcher, K. Multi-GPU Parallelization of Irregular Algorithms. 2016.

[14] Cabrera, F. CUDA Unified Virtual Address Space and Unified Memory. *A tech notebook*. Dec 2019. Dostupné z: https://nichijou.co/cudaRandom-UVA/.

[15] Cheng, J., McKercher, T. a Grossman, M. *Professional CUDA C Programming*. 1st. GBR: Wrox Press Ltd., 2014. ISBN 1118739329.

[16] Eassa, A., Ishii, A. a Wells, R. *Upgrading Multi-GPU interconnectivity with the third-generation Nvidia NVSwitch*. NVIDIA, Aug 2022. Dostupné z: https://developer.nvidia.com/blog/upgrading-multi-gpu-interconnectivity-with-the-third-generation-nvidia-nvswitch/.

[17] Foley, D. a Danskin, J. Ultra-Performance Pascal GPU and NVLink Interconnect. *IEEE Micro*. 2017, sv. 37, č. 2, s. 7–17. DOI: 10.1109/MM.2017.37.

[18] Harris, M. *How NVLink will enable faster, easier multi-gpu computing*. NVIDIA, Nov 2014. Dostupné z: https://developer.nvidia.com/blog/how-nvlink-will-enable-faster-easier-multi-gpu-computing/.

[19] Harris, M. *Nvidia DGX-1: The Fastest Deep Learning System*. Apr 2017. Dostupné z: https://developer.nvidia.com/blog/dgx-1-fastest-deep-learning-system/.

[20] Harris, M. Unified Memory for CUDA Beginners. *NVIDIA Technical Blog*. NVIDIA. Jun 2017. Dostupné z: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[21] Harvey, A. F. DMA Fundamentals on Various PC Platforms. In:. 1994.

[22] Jackson, M., Budruk, R., Winkles, J. a Anderson, D. *PCI Express Technology 3.0*. Mindshare Press, 2012. ISBN 0977087867.

[23] Kalia, A., Kaminsky, M. a Andersen, D. G. Design Guidelines for High Performance RDMA Systems. In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, červen 2016, s. 437–450. ISBN 978-1-931971-30-0. Dostupné z: https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia.

[24] Kraus, J. *An introduction to cuda-aware MPI*. NVIDIA, Mar 2013. Dostupné z: https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/.

[25] Krbila, M. *Simulace šíření tepla v mozku s využitím vysokoúrovňových GPGPU technik*. Brno, CZ, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/24464/.

[26] Li, A., Song, S. L., Chen, J., Li, J., Liu, X. et al. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems*. 2020, sv. 31, č. 1, s. 94–110. DOI: 10.1109/TPDS.2019.2928289.

[27] MacArthur, P., Liu, Q., Russell, R. D., Mizero, F., Veeraraghavan, M. et al. An Integrated Tutorial on InfiniBand, Verbs, and MPI. *IEEE Communications Surveys*. 2017, sv. 19, č. 4, s. 2894–2926. DOI: 10.1109/COMST.2017.2746083.

[28] NVIDIA. *NVSwitch for Advanced Multi-gpu Communication*. Dostupné z: https://www.nvidia.com/en-us/data-center/nvlink/.

[29] Orozco, D., Garcia, E., Pavel, R., Ayala, O., Wang, L.-P. et al. Demystifying Performance Predictions of Distributed FFT3D Implementations. In:. Září 2012, s. 196–207. DOI: 10.1007/978-3-642-35606-3_23. ISBN 978-3-642-35605-6.

[30] Pekurovsky, D. P3DFFT: A Framework for Parallel Computations of Fourier Transforms in Three Dimensions. *SIAM Journal on Scientific Computing*. Society for Industrial And Applied Mathematics (SIAM). jan 2012, sv. 34, č. 4, s. C192–C209. DOI: 10.1137/11082748x. Dostupné z: https://doi.org/10.1137%2F11082748x.

[31] Pennes, H. H. Analysis of Tissue and Arterial Blood Temperatures in the Resting Human Forearm. *Journal of Applied Physiology*. 1948, sv. 1, č. 2, s. 93–122. DOI: 10.1152/jappl.1948.1.2.93. PMID: 18887578. Dostupné z: https://doi.org/10.1152/jappl.1948.1.2.93.

[32] Ravi, J., Byna, S. a Koziol, Q. GPU Direct I/O with HDF5. In: *2020 IEEE/ACM Fifth International Parallel Data Systems Workshop (PDSW)*. 2020, s. 28–33. DOI: 10.1109/PDSW51947.2020.00010.

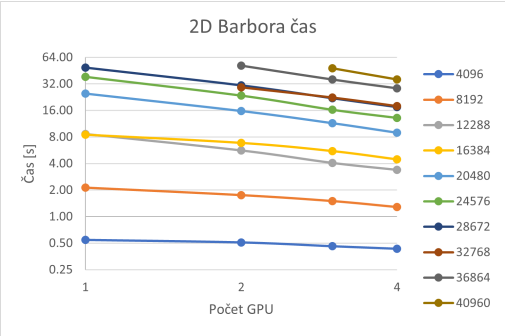
[33] Thompson, A. a Newburn, C. *GPUDirect storage: A direct path between storage and GPU memory*. NVIDIA, Aug 2021. Dostupné z: https://developer.nvidia.com/blog/gpudirect-storage/.

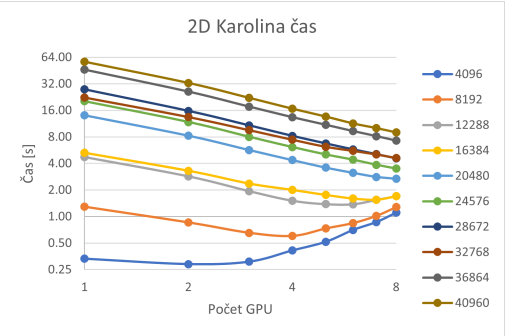
[34] Treeby, B., Cox, B. a Jaros, J. K-Wave User Manual. *K-Wave documentation*. k-Wave. Aug 2016. Dostupné z: http://www.k-wave.org/manual/k-wave_user_manual_1.1.pdf.

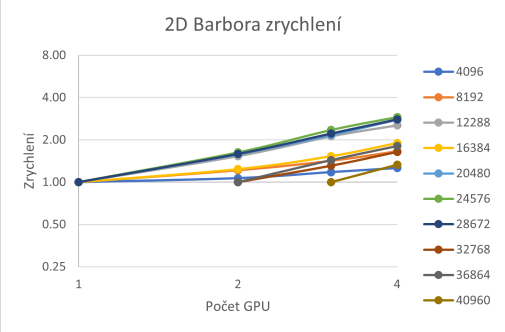
Příloha A

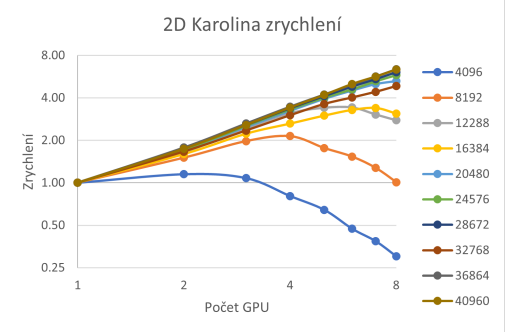
Výsledky měření simulace šíření tepla v médiu

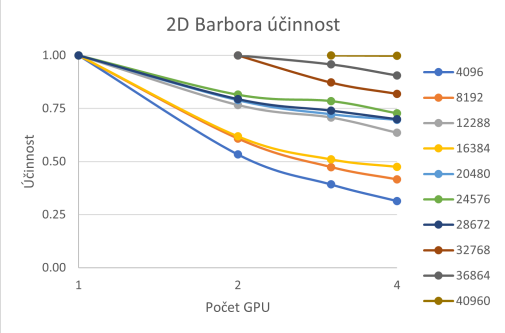
A.1 Výsledky 2D simulace

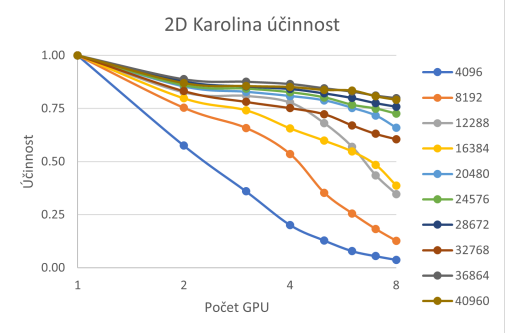




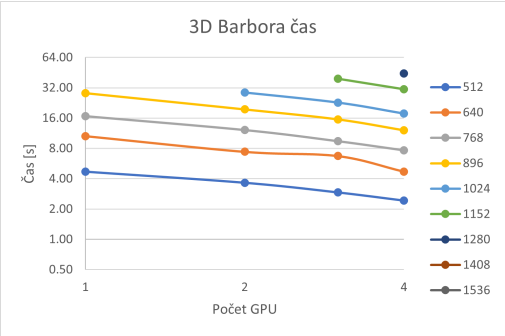


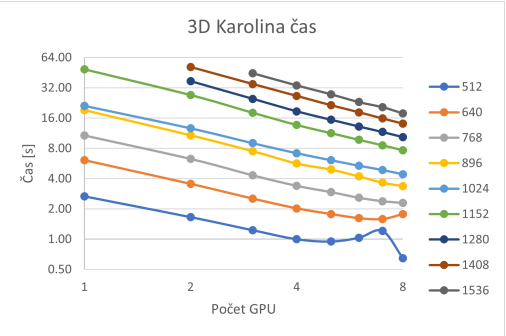


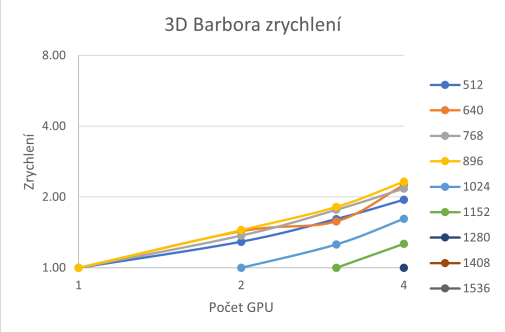


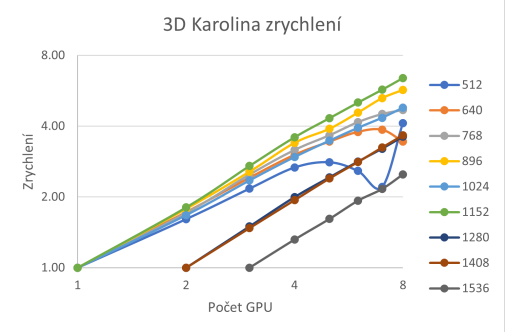


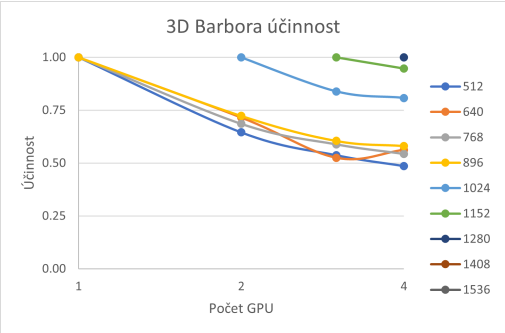
A.2 Výsledky 3D simulace

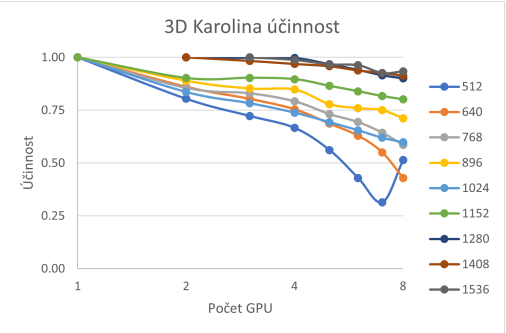








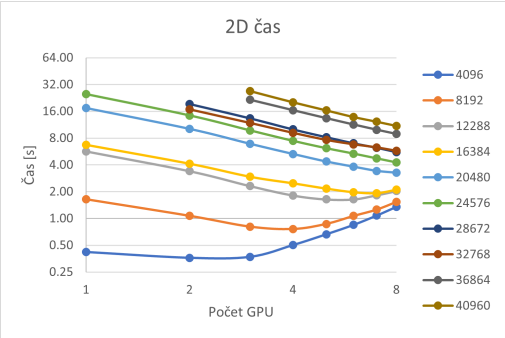


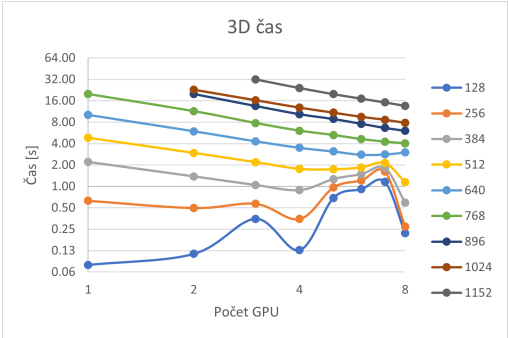


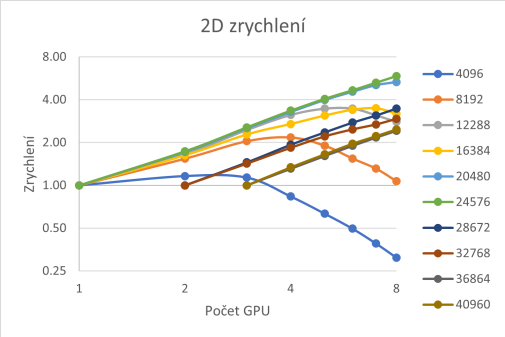
Příloha B

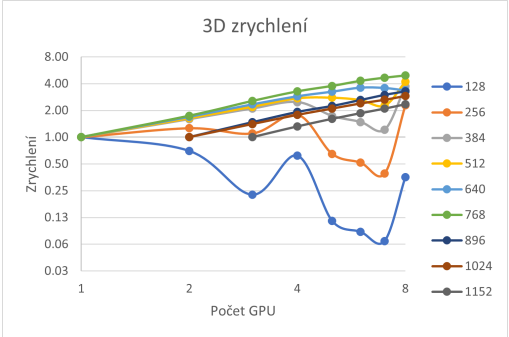
Výsledky měření simulace šíření akustických vln v médiu

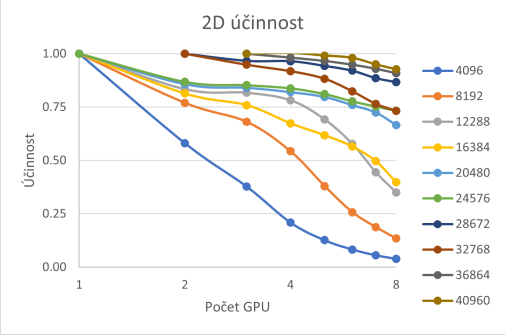
B.1 Výsledky simulace s homogenním médiem

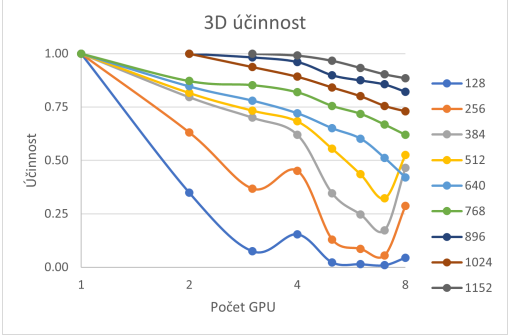












B.2 Výsledky simulace s heteregenním médiem