VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

SIMULACE ŠÍŘENÍ TEPLA V MOZKU S VYUŽITÍM VYSOKOÚROVŇOVÝCH GPGPU TECHNIK

SIMULATION OF HEAT DIFFUSION IN THE BRAIN USING HIGH-LEVEL GPGPU TECHNIQUES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. MARTIN KRBILA**
AUTHOR

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

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2021/2022

Zadání diplomové práce

Student: **Krbila Martin, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Počítačová grafika a interakce
Název: **Simulace šíření tepla v mozku s využitím vysokoúrovňových GPGPU technik**
Simulation of Heat Diffusion in the Brain Using High-Level GPGPU Techniques
Kategorie: Paralelní a distribuované výpočty
Zadání:

1. Seznamte se s knihovnou OpenMP 5.0 / Python pro paralelní výpočty na procesorech a grafických kartách.
2. Prostudujte implementaci simulace šíření tepla v balíku k-Wave.
3. Transformujte tuto implementaci z jazyka Matlab do jazyka C++.
4. Implementujte akcelerovanou verzi simulace pomocí knihovny OpenMP 5.0. Zaměřte se na přenositelnost mezi grafickými kartami různých výrobců.
5. Implementujte akcelerovanou verzi simulace pomocí knihovny CUDA.
6. Na reprezentativních datech a vzorku grafických karet porovnejte výkonnost jednotlivých implementací.
7. Srovnejte náročnost jednotlivých implementací a jejich pracnost.
8. Diskutujte vhodnost knihovny OpenMP 5.0 pro využití v dalších modulech balíku k-Wave.
9. Zhodnoťte dosažené výsledky.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/
Vedoucí práce: **Jaroš Jiří, doc. Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 18. května 2022
Datum schválení: 29. října 2021

Abstrakt

Tato diplomová práce se zabývá akcelerací simulace šíření tepla na grafické kartě. Je zde popsán postup akcelerace existující implementace v *Matlabu*, která je součástí balíku *k-Wave*. V práci jsou popsány různé vysokoúrovňové i nízkoúrovňové knihovny pro programovaní na grafických kartách a shrnuty jejich silné a slabé stránky. Byla vytvořena implementace simulace na *GPU* kompletně pokrývající funkcionalitu původní verze, která dosahuje přibližně stonásobného zrychlení oproti procesorové implementaci v *Matlabu*. Jako součást této práce byl také vytvořen modul umožňující výpočet diskrétních trigonometrických transformací na grafické kartě, který dosahuje přibližně desetinásobného zrychlení oproti nejlepší procesorové variantě a umožňuje akceleraci simulace s různými okrajovými podmínkami. Výstupem práce je také srovnání výkonu několika verzí základní simulace při využití různých *GPGPU* technik.

Abstract

This master's thesis deals with acceleration of heat diffusion simulation using graphics cards. It describes an approach to acceleration of an existing implementation in *Matlab*, which is a part of *k-Wave* package. Various high-level as well as low-level libraries for *GPU* programming are introduced here and their strenghts and weaknesses compared. A complete implementation of the simulation on *GPU* was created as a part of this work. This implementation achieves around hundredfold speedup over the existing *CPU* solution in *Matlab*. A module for computation of discrete trigonometric transformations on graphics card was created to accelerate simulation with various boundary conditions. This module achieves around ten times speedup over the best *CPU* implementation. Another output of this thesis is a performance comparison of several implementations of basic diffusion simulation each using a different *GPGPU* technique.

Klíčová slova

GPGPU, Simulace šíření tepla, GPU, OpenMP, OpenACC, CUDA, CuPy, OpenCL, Matlab, DTT

Keywords

GPGPU, Heat diffusion simulation, GPU, OpenMP, OpenACC, CUDA, CuPy, OpenCL, Matlab, DTT

Citace

KRBILA, Martin. *Simulace šíření tepla v mozku s využitím vysokoúrovňových GPGPU technik*. Brno, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. Jiří Jaroš, Ph.D.

Simulace šíření tepla v mozku s využitím vysokoúrovňových GPGPU technik

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.

.....
Martin Krbila
14. května 2022

Poděkování

Děkuji svému vedoucímu panu docentovi Jiřímu Jarošovi za pomoc při vypracování této diplomové práce. Tato práce byla podpořena Ministerstvem školství, mládeže a tělovýchovy České republiky prostřednictvím e-INFRA CZ (ID:90140).

Obsah

Kapitola 1

Úvod

Tato diplomová práce se zabývá akcelerací simulace šíření tepla ve tkáních na grafických kartách. Studium šíření tepla ve tkáních má význam v mnoha oblastech medicíny jako například kryochirurgie, radio-frekvenční a laserové termální ablace, termální resekce a ultrazvuk [3], kde je výhodné pochopit mechaniky přenosu tepla předtím, než je aplikován jakýkoliv reálný zákrok.

Konkrétně jde o akceleraci simulace šíření tepla podle *Pennes' bioheat equation*[10] aktuálně implementované v jazyce *Matlab* na procesoru jako součást balíku *k-Wave*. Aktuální řešení není optimální z hlediska výkonnosti. Jeho optimalizace na procesoru, ale především jeho akcelerace na grafické kartě může přinést významné zrychlení.

Obsahem této práce je kromě implementace samotné také srovnání několika dostupných nízko i vysokoúrovňových knihoven a jazyků pro práci s *GPU*. Grafické karty obvykle umožňují zrychlení oproti procesoru přibližně o jeden řád za podmínky, že úloha dokáže využít model *SIMD* (Single Instruction, Multiple Data). Programování grafických karet je ale v mnoha případech náročné. Zavedené knihovny jako *CUDA* a *OpenCL* jsou sice výkonné, ale vývoj v těchto technologiích trvá dlouho a vytvořený kód často obsahuje části, které přímo nesouvisí s řešenou úlohou ale spíše s hardwarovými záležitostmi, jako je využití sdílené paměti, volání kernelů a synchronizace front, vláken a warpů atd. Takový kód pak může být hůře čitelný a jeho údržba bude náročnější. Z tohoto důvodu se objevuje množství vysokoúrovňových knihoven, které se snaží programátora od těchto implementačních detailů odstínit. Otázka, kterou si tato práce klade je, zda využití těchto knihoven může přinést zrychlení blížící se využití nízkoúrovňových knihoven s mnohem nižším úsilím programátora a se zachováním vysoké čitelnosti kódu. Pozornost bude věnována především vysokoúrovňovým knihovnám jako jsou například *OpenMP*, *Thrust* nebo *OpenACC* v jazyce *C++* a *CuPy* v jazyce *Python*. Při porovnání bude zohledněn dosažený výkon a paměťová náročnost, ale také jejich spolehlivost, možnosti nasazení na různých systémech, ale především náročnost na implementaci.

Následující kapitola představuje knihovnu *k-Wave* a již existující implementaci simulace šíření tepla v jazyce *MATLAB*.

V kapitole 4 – Implementace bude popsána architektura vzniklých řešení v *Pythonu* a *C++* a také modul pro výpočet diskrétních trigonometrických transformací na GPU. Jsou zde popsány i další technologie využité pro tvorbu aplikace.

V předposlední kapitole lze nalézt srovnání jednotlivých implementací a porovnání zrychlení a paměťové náročnosti vůči původní verzi řešení na procesoru. Bude zde vyhodnocena i výkonnost vytvořeného *DTT* modulu.

Kapitola 2

Knihovna k-Wave

K-Wave je knihovna pro *Matlab* a *C++* s otevřeným zdrojovým kódem[1] distribuovaná pod licencí *LGPL*. Tato knihovna primárně slouží pro simulaci šíření akustických a ultrazvukových vln ve složitých prostředích, poskytuje ale také mnoho dalších funkcí. Jednou z nich je simulace šíření tepla v tkáni, na kterou se zaměřuje tato práce. Balík obsahuje sadu testovacích úloh (`testing/unit/kWaveDiffusion*.m`), které budou využity pro ověření správnosti mého řešení vůči původní implementaci.

2.1 Stávající implementace

Existující verze v *Matlabu* obsahuje řešení „Pennes' bioheat equation"[10] v čase pro $1D$, $2D$ a $3D$ problémy. Aktuální implementace se nachází v souboru `kWaveDiffusion.m`. Cílem mé práce je převést funkcionalitu tohoto skriptu do *C++/Python* pro vykonání na *GPU*. Nejdůležitější funkcionalitou tohoto skriptu je:

- Simulace v $1D$, $2D$ a $3D$
- Homogenní a heterogenní médium
- Okrajové podmínky (periodické, vodivé a izolující médium)
- *k-Space* korekce
- Perfuze
- Výpočet integrálu poškození tkáně
- Zdroje tepla
- Senzory

Tento skript (respektive třída *kWaveDiffusion*) se stará o inicializaci dat potřebných pro výpočet, samotné vykonání kroků simulace a také o výstup ze senzorů a vizualizaci výsledků simulace. Hlavní smyčku simulace lze vidět na Algoritmu 2.1. Pro ilustraci je uvedena pouze zjednodušená $2D$ verze bez průběžného ukládání dat ze senzorů a grafického výstupu. Ukázkový výstup simulace je vidět na Obrázku 2.1. Hlavní smyčka obsahuje čtyři části:

1. Výpočet perfuze (`p_term`)

[1] Knihovna *k-Wave*: `http://www.k-wave.org/`

2. Výpočet difuze (`d_term`)
3. Aktualizace teploty (`T`)
4. Výpočet integrálu poškození tkáně (`cem43`)

```
% loop through time steps
for t_index = 1:Nt

  % compute perfusion update term
  if use_perfusion
    p_term = -perfusion_coeff .*
             IT(kappa .* FT(T - blood_ambient_temperature));
  end

  % compute diffusion update term
  if flag_homogeneous
    d_term = diffusion_p1 .*
             diffusion_p2 .*
             IT(-k.^2 .* kappa .* FT(T));
  else
    T_FT = FT(T);
    d_term = diffusion_p1 .*
               (IT(deriv_x .* FT(diffusion_p2 .* IT(deriv_x .* T_FT))) +
               IT(deriv_y .* FT(diffusion_p2 .* IT(deriv_y .* T_FT))));
  end

  % update temperature using finite difference time step
  T = T + dt .* (d_term + p_term + q_term);

  % update the damage integral
  cem43 = cem43 +
          dt ./ 60 .*
          (0.5 .* (T >= 43) + 0.25 .* (T >= 37 & T < 43)).^(43 - T);
end

```

Algoritmus 2.1: Původní implementace hlavní smyčky v balíku *k-Wave* (zjednodušená a pouze pro $2D$)

Proměnná `perfusion_coeff` udává míru perfuze a `blood_ambient_temperature` teplotu krve, která je přiváděna do tkáně. Proměnná `kappa` značí *k-Space* korekci.

Výpočet difuze probíhá odlišně pro homogenní a heterogenní médium. V případě homogenního média lze výpočet zjednodušit a není nutné počítat gradienty pro každý rozměr matice (`deriv_x` a `deriv_y`) a lze je sjednotit do matice `k`.

Přírůstek ze zdrojů tepla je před-počítán v matici `q_term`. Funkce `FT()` a `IT()` značí dopředné a zpětné transformace podle okrajových podmínek zvolených uživatelem:

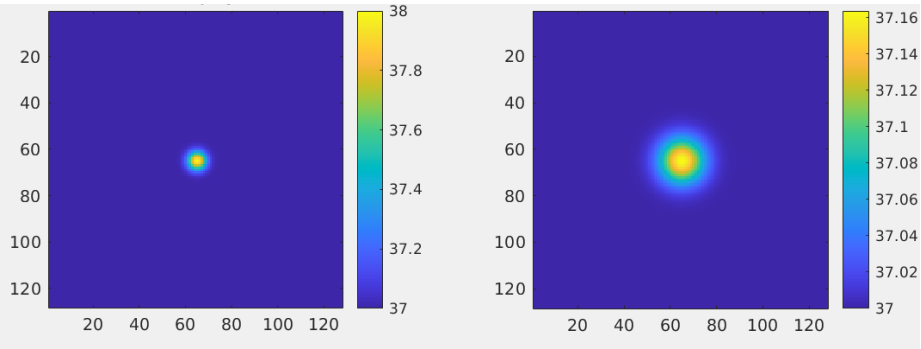
- Diskrétní Fourierova transformace pro periodické médium
- Diskrétní cosinová transformace typu 1 pro izolující okraj domény
- Diskrétní sinová transformace typu 1 pro vodivý okraj domény

Kromě této funkcionality obsahuje hlavní smyčka také výstup dat ze senzorů, přičemž se ukládají hodnoty teploty a poškození tkáně ze zvolených bodů domény. Hlavní smyčka také obsahuje vizualizaci simulace (pro jednoduchost neuvádím). Z implementace je zřejmé, že

všechny operace až na transformace probíhají nezávisle pro každý element matice a jejich paralelizace na *GPU* bude velmi přímočará. Pro výpočet Fourierových a trigonometrických transformací bude potřeba využít některou z dostupných vysoce optimalizovaných knihoven pro *GPU*.

Třída *kWaveDiffusion* závisí na třídě *kWaveGrid* (ze souboru `kWaveGrid.m`), která je využívána napříč celou knihovnou *k-Wave* a zprostředkovává inicializaci některých matic před začátkem simulace. Výsledné řešení proto musí obsahovat i část funkcionality tohoto skriptu.

Stávající implementace běží na procesoru, ale je možné využít *Parallel Computing Toolbox*[2] pro výpočet na *GPU*. Při použití tohoto nástroje není možné použít jiné než periodické okraje domény, neboť potřebné transformace jsou implementovány pouze pro *CPU* variantu. Pro spuštění simulace v *Matlabu* na *GPU* stačí aby některá matice použitá při řešení (např. vstupní teplota) byla inicializována jako `gpuArray()`.



Obrázek 2.1: Výstup simulace šíření tepla ve $2D$ homogenním médiu. Na obrázu vlevo je počáteční rozložení tepla, na obrázku vpravo je rozložení tepla v doméně po 300 krocích délky $0.5s$.

[2] *Matlab parallel computing toolbox*: `https://www.mathworks.com/products/parallel-computing.html`

Kapitola 3

Knihovny pro programování GPU

V této kapitole jsou popsány vybrané knihovny a jazyky používané pro výpočty na grafických kartách, z nichž některé budou použity pro výslednou implementaci a porovnání výkonu a paměťové náročnosti. Budou shrnuty silné a slabé stránky jednotlivých technologií a také náročnost jejich použití a nasazení.

3.1 CuPy

CuPy je knihovna jazyka *Python* pro práci s poli na grafické kartě. Silnou stránkou této knihovny je především její kompatibilita s rozhraním knihoven *NumPy* a *SciPy*, které jsou hojně používané pro výpočty v jazyce *Python* na procesoru. To v mnoha případech umožňuje jednoduchou výměnu importované knihovny *NumPy* za *CuPy* a existující kód může být spuštěn na grafické kartě. Operace s poli jsou převedeny na jednotlivé kernely běžící na grafické kartě, čímž lze dosáhnout násobného zrychlení s velmi malým úsilím. Každé n-dimenzionální pole je automaticky uloženo na grafické kartě. Hlavní rozdíl oproti použití knihovny *NumPy* je tedy nutnost volat metodu pro kopii dat zpět na *CPU* pro získání výsledků.

CuPy poskytuje ale také mnoho dalších možností zvyšování výkonu. První z nich je možnost tvorby uživatelsky definovaných kernelů, a tím spojení více elementárních operací do jednoho kernelu. *CuPy* rozlišuje několik typů uživatelských kernelů, které se převážně liší v úrovni abstrakce nad rozhraním *CUDA*. Všechny uživatelské kernely v *CuPy* jsou definovány za běhu. Kernely jsou kompilovány až při volání, v závislosti na dodaných vstupních hodnotách, jejich typech a rozměrech.

Prvním typem uživatelského kernelu je *ElementwiseKernel*, který umožňuje nezávislé zpracování prvků v poli. Uživatel u tohoto typu kernelu pouze specifikuje vstupní parametry kterými jsou jednoduché hodnoty základních datových typů a stejně tak výstupní hodnoty. Výsledný kernel se následně při volání stará o distribuci vstupních parametrů jednotlivým vláknům, takže uživatel nemusí specifikovat, jestli je daný parametr kernelu pole, matice, nebo jen jedna hodnota aplikovaná pro všechny volání kernelu. *ElementwiseKernel* obsahuje také možnost specifikovat tzv. raw parametr, který je možné indexovat libovolně. Také podporuje šablony kernelů, kde se konkrétní typ parametru nebo návratové hodnoty dosadí až při volání, v závislosti na typech předaných vstupních parametrů.

Druhým typem uživatelského kernelu v *CuPy* je *ReductionKernel*, který provádí libovolnou redukci a mapování funkce na zadané vstupy. Jeho definice je podobná jako v případě

ElementwiseKernelu, s tím rozdílem, že uživatel navíc definuje operaci pro redukci, počáteční hodnotu redukce a dále funkce aplikované na každý element před i po redukci.

Poslední podporovaný typ kernelu je *RawKernel*, který umožňuje zápis kernelu přímo v rozhraní *CUDA* a při jeho volání je nutné specifikovat velikost a počet bloků. Tento typ kernelu nabízí největší kontrolu nad vykonávanými operacemi ovšem za cenu nutnosti zapsat vše explicitně (stejně jako v *C/C++* se standardním rozhraním *CUDA*).

Kromě uživatelských kernelů je možné také použít některé dekorátory jako například `@cupy.fuse()`, který lze aplikovat na existující pythonovou funkci, čímž dojde k automatickému spojení všech kroků této funkce do jednoho kernelu. V rámci mého experimentování jsem však narazil na funkce, které takto spojit nelze a způsobují výjimky při spuštění, nebo je jejich výkon výrazně nižší než u explicitně zadaného kernelu.

CuPy také obsahuje *Just-In-Time* kompilátor, který dokáže funkce napsané čistě v pythonu převést na raw *CUDA* kernel. *CuPy* pro generování kernelu analyzuje abstraktní syntaktický strom dané funkce v jazyce *Python*[1].

CuPy obsahuje také základní knihovny *CUDA* jako *cuBLAS*, *cuFFT*, *cuRand*, knihovnu pro transpozice apod. Použití těchto knihoven je velmi jednoduché obzvláště ve srovnání s implementací v *C++*, neboť *CuPy* se stará o správné nastavení parametrů operací nebo vytvoření plánů automaticky, stejně jak tomu je i v knihovně *NumPy*.

Vysokoúrovňovost *CuPy* ale přináší i některé nevýhody. Především se jedná o alokaci paměti, která probíhá dynamicky podle potřeby během výpočtu, přičemž *CuPy* si alokuje vlastní vyrovnávací paměť. Toto chování může mít za následek vyšší spotřebu paměti, protože uživatel nemá nad alokacemi plnou kontrolu. Podobně jako u správy paměti není ani spouštění kernelů v režii uživatele, což mělo ve sledovaných příkladech za následek existenci několika zbytečných kernelů a kopií paměti, kterých není vždy snadné nebo možné se jednoduše zbavit. Řešením těchto problémů je samozřejmě explicitní alokace paměti a využívání raw kernelů v *CuPy*, ovšem tento přístup přináší jen minimální výhody oproti přímé implementaci v jazyce *C++*.

CuPy je multiplatformní a podporuje jak *Windows* tak i *Linux*. Přestože cílí na grafické karty *nvidia*, obsahuje experimentální podporu grafických karet *AMD* pomocí backendu implementovaném v *ROCm* (ten je však zatím dostupný výhradně v operačním systému *Linux*). Navíc v této verzi *CuPy* chybí některá funkcionalita oproti *CUDA* backendu[2] jako například *cuFFT* pro více grafických karet, *cuTENSOR*, *cuDNN* a další.

Další problémy jsou svázány s funkčností knihovny *ROCm* jako takové. Stále například chybí podpora pro novější karty řady *AMD RX řada 6000*[3], což limituje použitelnost tohoto backendu. Dlouhou dobu byly vývojáři *CuPy* podporovány pouze verze *ROCm 4.0, 4.2 a 4.3*. Podpora pro *ROCm 5.0* se objevuje až ve verzi *CuPy* 11.4 vydané 27. dubna 2022.

3.2 OpenMP

OpenMP je rozhraní sloužící k paralelizaci výpočtů na více-jádrových procesorech. Rozhraní obsahuje direktivy překladače a knihovny, které umožňují popsat způsob paralelizace programu. Od *OpenMP* verze 4 přibyla podpora grafických akcelerátorů pomocí konstrukce `target`[4].

[1] Python JIT: `https://docs.cupy.dev/en/stable/user_guide/kernel.html#basic-design`

[2] Omezení *ROCm* backendu: `https://docs.cupy.dev/en/stable/install.html?highlight=ROCM#limitations`

[3] Karty podporované *ROCm*: `https://github.com/ROCm/ROCm.github.io/blob/master/hardware.md`

[4] Podpora *GPU* v *OpenMP*: `https://www.openmp.org/updates/openmp-accelerator-support-gpus/`

Hlavním přínosem *OpenMP* je možnost snadno akcelerovat existující *C/C++* kód pomocí několika direktiv, přičemž tento kód je možné přeložit jak pro použití na procesoru, tak na grafické kartě a mimo jiné je zachována jeho čitelnost a přímočarost (v porovnání s explicitně paralelizovanou a vektorizovanou implementací). V případě využitého kompilátoru (*nvc++*) je možné cílové zařízení pro překlad *OpenMP* kódu specifikovat pomocí argumentu příkazové řádky `-mp` s možností `multicore` nebo `gpu`. Velkou výhodou na rozdíl od *CuPy* je, že *OpenMP* umí automaticky využít sdílené paměti na grafické kartě, bez nutnosti toto explicitně programovat (pro řešenou úlohu je toto ovšem irelevantní). V rámci této práce vzniklo několik demonstračních programů, ukazujících fungování (nebo naopak nefunkčnost) některých konceptů v rámci *OpenMP*, jako např. interoperabilita s jinými *GPGPU* knihovnami, atomické operace, redukce, podpora *STL* knihovny jazyka *C++* (datové kontejnery, chytré ukazatele a iterátory).

Interoperabilita s jinými knihovnami je v současnosti problematická. *OpenMP* podporuje direktivu `use_device_ptr`, která dává přístup k paměti na GPU nebo naopak `is_device_ptr`, která umožňuje použití *CUDA* paměti v rámci *OpenMP* [5]. Nelze ovšem určit například *stream* ve kterém se bude výpočet vykonávat. Tato situace vede k nutnosti použití `cudaDeviceSynchronize()` po každém takovém volání, což vede ke zpomalení v důsledku zbytečného čekání mezi jednotlivými kernely. V dalších verzích *OpenMP* byla uvedena konstrukce `interop`, která umožňuje přístup k nízkoúrovňovým informacím jako jsou například streamy [1], tato funkcionalita ale zatím není dobře podporována v překladači *nvc++*, který jsem využil. Interoperabilita s *OpenCL* v tomto překladači nefunguje a bude muset být otestována v jiném překladači s podporou karet *AMD*.

OpenMP dále podporuje běžné konstrukce jako atomické operace a redukce, které jsou realizovány stejně jako v případě CPU verze (konstrukce `#pragma omp atomic` a `#reduction`). V použitém překladači (*nvc++*) nejsou podporovány zámky a kritické sekce v *GPU* regionu. Asynchronní vykonávání je řešeno pomocí konstrukce `depend`, která umožňuje specifikovat datové závislosti mezi jednotlivými kroky výpočtu a spouštění kernelů v odpovídajících streamech je řešeno automaticky [6], což na jednu stranu přináší zjednodušení v rámci *OpenMP* kódu, ale jak bylo zmíněno, limituje možnosti současného použití s jinými knihovnami.

Paměť je v *OpenMP* spravována pomocí konstrukcí `target data`, `target enter data` a `target exit data`, které provádějí alokace, ale nespouštějí kód na akcelerátoru. Samotné spuštění kódu na akcelerátoru probíhá pomocí direktivy `target` [5].

V oblasti platnosti direktivy `target` lze paralelizovat smyčky `for` několika způsoby. Direktiva `parallel` funguje stejně jako na *CPU* a vytváří pouze jednu úroveň paralelismu. To znamená, že spustí pouze jedno vlákno v každém bloku grafické karty [5]. K lepšímu využití zdrojů grafické karty je nutné použít direktivu `teams`, která vytvoří několik „týmů“ vláken. Vykonávání kódu probíhá redundantně pro každý tým [5]. Pro rozdělení práce mezi týmy je nutné použít direktivu `distribute`, která staticky rozdělí iterace paralelizované smyčky mezi vzniklými týmy a jejich vlákny [5].

Další užitečná direktiva je například `collapse`, která dokáže spojit několik po sobě následujících smyček a zvýšit množství paralelismu [5].

Nevýhodou *OpenMP* pro použití na grafických kartách je především fakt, že se jedná o poměrně novou, stále se rozvíjející technologii. Podpora překladačů zatím není stoprocentní a je často specifická pro různé výrobce grafických karet a operační systémy. To také omezuje multiplatformní nasazení a je nutné využít nástrojů jako např. *Docker*. V mé diplomové práci jsem pro překlad kódu s *OpenMP 5* využíval překladač *nvc++* od společnosti

nvidia, dodávaný v rámci *nvidia HPC SDK*[5], který je aktuálně dostupný výhradně pro systém *Linux* a grafické karty *nvidia*. Alternativou pro grafické karty *AMD* je překladač *AOMP*[6] a *AOCC*.

3.3 CUDA

CUDA (Compute Unified Device Architecture) je hardwarová a softwarová architektura umožňující výpočty na grafických kartách v jazyce *C/C++* od společnosti *nvidia*[13].

CUDA je na rozdíl od předchozích zmíněných nízkoúrovňová knihovna. Uživatel se musí starat o veškeré alokace i dealokace paměti, přesuny dat mezi *CPU* a *GPU*, spouštění kernelů, práci se sdílenou pamětí apod. Na druhou stranu by použití této knihovny mělo přinést největší zrychlení (oproti vysokoúrovňovým knihovnám) za cenu vyššího úsilí uživatele.

Mezi výhody knihovny *CUDA* patří zejména její rozšířenost jak na *Windows*, tak i na *Linuxu* a její stabilita. *CUDA* také poskytuje množství vysoce optimalizovaných knihoven pro některé typické operace. Mezi tyto patří *cuBLAS* a *cuTENSOR* pro lineární algebru, *cuFFT* pro výpočet rychlé diskrétní Fourierovy transformace, *cuRAND* pro generování pseudonáhodných čísel na *GPU*, *cuSPARSE* pro lineární algebru v řídkých maticích a mnohé další[7].

Zásadní nevýhodou je nemožnost použití této knihovnu na jiných grafických akcelerátorech než od společnosti *nvidia*. Jako alternativa se nabízí nástroj *hipify*[8], umožňující konverzi volání knihoven *CUDA* v *C/C++* kódu za ekvivalentní volání knihoven *ROCm HIP*.

3.4 OpenCL

OpenCL (Open Computing Language) je standard pro programování heterogenních výpočetních systémů[9], spravovaný společností *Khronos group*.

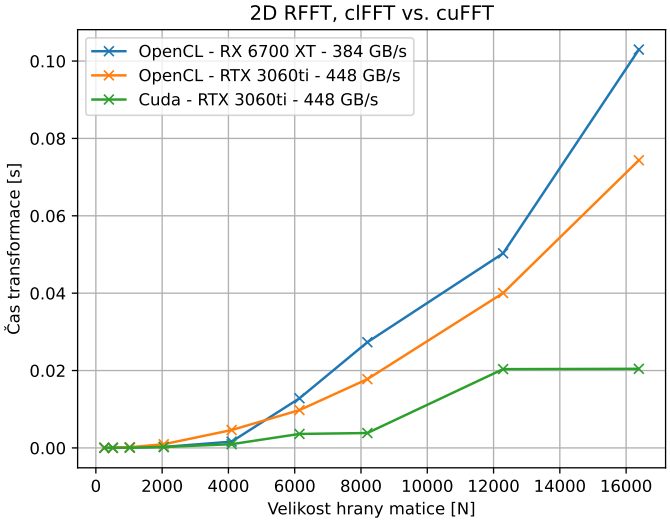
Podobně jako u frameworku *CUDA* je i *OpenCL* poměrně nízkoúrovňové.

Mezi hlavní výhody *OpenCL* patří jeho rozšířenost a stabilita, ale také podpora pro grafické karty *nvidia* a *AMD*, ale i jiné akcelerátory jako např *FPGA*. Další výhodou je také fakt, že jde o knihovnu s otevřeným zdrojovým kódem.

Mezi nevýhody patří slabší knihovny, které nedosahují takového výkonu jako v případě ekvivalentních knihoven *CUDA* (na grafické kartě *nvidia*) jak bude ukázáno v Kapitole 5 nebo na Obrázku 3.1, který zobrazuje výkonnost knihovny *clFFT* v porovnání s *cuFFT*. Na grafu lze vidět, že *clFFT* na kartě *AMD* se do velikosti domény 4096 blíží k výkonu *cuFFT* na kartě *nvidia* a je zde jen asi o 50 % pomalejší, přesto že *clFFT* na kartě *nvidia* dosahovalo až 5× horších časů. Při zvětšování domény se situace mění a *clFFT* na kartě *nvidia* je opět rychlejší. U měření *cuFFT* lze vidět anomálii v bodě 12288, kde transformace této velikosti trvá stejně dlouho jako větší transformace s velikostí 16384. Myslím si, že je to způsobeno tím, že se nejedná o matici s velikostí mocniny 2 a použitý algoritmus transformace není tak efektivní. Vzhledem k tomu, že výpočetně nejnáročnější část simulace řešené v této práci je výpočet diskrétní Fourierovy transformace, je využití *OpenCL* k tomuto účelu (alespoň na kartách *nvidia*) nevhodné.

[5] *nvidia* HPC SDK: `https://developer.nvidia.com/hpc-sdk`
[6] *AOMP*: `https://github.com/ROCm-Developer-Tools/aomp`
[7] *CUDA* knihovny: `https://developer.nvidia.com/gpu-accelerated-libraries`
[8] Nástroj hipify: `https://github.com/ROCm-Developer-Tools/HIPIFY`
[9] OpenCL: `https://www.khronos.org/opencl/`

Z méně důležitých a subjektivních nevýhod lze uvést poměrně nepohodlné rozhraní této knihovny (např. předávání argumentů kernelům).



Obrázek 3.1: Měření doby běhu *RFFT* (Real–to–Complex *FFT*) v *cuFFT – CUDA* a *OpenCL – clFFT* na $2D$ matici s velikostí hrany N. Měřeno na kartách *nvidia RTX 3060ti* a *AMD RX 6700 XT*. V pozorovaném případě je na kartě *nvidia clFFT* přibližně $2 - 5\times$ pomalejší než *cuFTT*.

3.5 OpenACC

OpenACC (Open Accelerators) je rozhraní velmi podobné *OpenMP*. Je určené pro paralelizaci programů v *C/C++* a *Fortran* [9]. Podobně jako v případě *OpenMP* uživatel specifikuje pomocí `#pragma acc` oblasti kódu určené k paralelizaci. Kód může být přeložen pro vykonávání buď na *CPU*, nebo na *GPU* (V překladači *nvc++* přepínač `-acc`). *OpenACC* se také stará o správu paměti na *GPU* v závislosti na direktivách dodaných uživatelem [9]. Výhodou *OpenACC* oproti *OpenMP* je lepší interoperabilita s rozhraním *CUDA*. Například umožňuje nízkoúrovňový přístup ke *CUDA* streamům přes volání `acc_set_cuda_stream()`, což eliminuje problémy se synchronizací, na které jsem narazil při použití *OpenMP*. Nevýhoda *OpenACC* je například nemožnost vytvořit binární zámek.

3.6 Thrust

Thrust je šablonová knihovna jazyka *C++* založená na STL[10]. Jedná se o vysokoúrovňové rozhraní pro tvorbu paralelních aplikací. Rozhraní poskytuje základní konstrukce jako redukce, sumy prefixů, řazení a iterátory. *Thrust* také umožňuje definovat vlastní funktory (transformace) pro jiné specifické operace. Tento přístup umožňuje spojit více operací do jednoho kernelu, a tím omezit počet čtení a zápisů do globální paměti *GPU*. Základní transformace v knihovně *Thrust* bohužel podporují jen unární a binární operace. Pokud

[10]Thrust: `https://docs.nvidia.com/cuda/thrust/index.html#abstract`

chce uživatel vytvořit obecnou transformaci s libovolným množstvím vstupních argumentů, je nutné použít tzv. *zip* iterátory a spojovat takto vstupní data dohromady před voláním transformace. Tento postup je z mého hlediska poměrně neintuitivní a zbytečně složitý v porovnání s ostatními zmiňovanými knihovnami.

Výhodou knihovny *Thrust* je především interoperabilita s rozhraním *CUDA* a dobrá podpora v překladači *nvcc*. Nevýhodou je chybějící podpora karet *AMD*, pro které ale existuje alternativní knihovna *Bolt* s podobnými funkcemi[11].

3.7 ROCm HIP

AMD ROCm je platforma s otevřeným zdrojovým kódem pro vysoce náročné výpočty na *GPU*[12].

Jeho součástí je i *HIP* (Heterogeneous-Computing Interface for Portability) – *C/C++* rozhraní navržené pro snadnou konverzi *CUDA* kódu. K tomuto účelu vznikl nástroj *HIPify*, umožňující automatickou konverzi *CUDA* kódu do *HIP*.

Oproti *CUDA* rozhraní, ze kterého vychází, nabízí *HIP* navíc přenositelnost mezi kartami *AMD* a *nvidia*. Nevýhodou je opět fakt, že se jedná o relativně nové, stále se rozvíjející rozhraní, nebo také úplná absence podpory operačního systému *Windows*.

3.8 SYCL

SYCL je multiplatformní *C++* rozhraní pro výpočty na heterogenních systémech [2]. Rozhraní využívá standardní konstrukce moderního *C++* (např. šablony), ale umožňuje také interoperabilitu s nízkoúrovňovou implementací v *OpenCL* [2].

Výhodou tohoto rozhraní je jednoduší použití oproti *OpenCL* a také možnost spustit vytvořený kód jak na procesoru, tak na akcelerátoru. Nevýhodou tohoto rozhraní je opět slabší podpora v kompilátorech. *Nvidia* sama o sobě *SYCL* nepodporuje a podpora v běžných kompilátorech je rovněž omezená. V současnosti ho podporují tyto kompilátory: *hipSycl, triSycl, dpc++* a některé další [13].

3.9 Julia

Julia je poměrně nový a populární jazyk určený pro vysoce náročné výpočty, vizualizace dat, strojové učení ale i běžné použití[14]. Jedná se o multiparadigmatický rychlý kompilovaný jazyk (*Just-in-time*) s dynamickým typováním a otevřeným zdrojovým kódem.

Julia umožňuje práci s *GPU* pomocí knihovny *CuArrays*, které podobně jako v případě knihovny *CuPy* (sekce 3.1) reprezentují n-dimenzionální pole na grafické kartě.

Výhodou tohoto jazyka je jednoduchost použití a vysoký výkon, kterého je možné dosáhnout. Jako nevýhody tohoto jazyka bych uvedl jeho novost a slabší podporu nebo nekompletní podporu všech knihoven *CUDA*. Subjektivní nevýhodou může být i fakt, že jde zatím o poměrně neznámý jazyk, který se ne každý bude chtít učit, zvláště když známější jazyky jako např. *Python* poskytují knihovny s obdobnou funkcionalitou. Jedinou výhodu přináší fakt, že jde o jazyk kompilovaný do strojového kódu. Z funkčního hlediska mi také

[11] Bolt C++: `https://github.com/HSA-Libraries/Bolt`
[12] *AMD ROCm*: `https://rocmdocs.amd.com/en/latest/`
[13] SYCL kompilátory: `https://github.com/illuhad/hipSYCL`
[14] Julia: `https://julialang.org/`

nevyhovovaly pomalé časy spouštění programu způsobené překladem a načítáním modulů *CUDA*, které navíc mohou být za běhu stahovány.

Kapitola 4

Implementace

V rámci mé diplomové práce vznikly dvě oddělené implementace simulace šíření tepla. První z nich v jazyce *Python* a druhá v jazyce *C/C++*. Další experimenty probíhaly i s jazykem *Julia*. Tato kapitola také popisuje vytvořenou knihovnu pro trigonometrické transformace na *GPU* a technologie použité pro nasazení programu.

4.1 Použité technologie

První z implementací vznikla v jazyce *Python 3* jako prototyp pro implementaci v *C/C++*. Toto řešení využívá již zmíněný modul *CuPy* (sekce 3.1) pro výpočty na *GPU*, dále knihovny *h5py* umožňující volání knihovny *HDF5* v jazyce *Python*, *numpy* a pro účely vizualizace také *matplotlib* (knihovna pro kreslení grafů v jazyce *Python*) a *opencv-python*, která poskytuje volání knihovny *OpenCV* v jazyce *Python*.

Druhá implementace je realizována v jazyce *C++*. Zde je využit překladač *nvc++* pro *C++* kód, společně s překladačem *nvcc* pro *CUDA* moduly. Z knihoven jsou použity *FFTW 3* pro Fourierovu transformaci na *CPU*, *HDF 5* pro načítání souborů `*.h5` a *cuFFT* společně s *clFFT* pro Fourierovu transformaci na *GPU* pro *CUDA* a *OpenCL*. Součástí této implementace je řešení v *OpenMP* (jak na procesoru, tak na grafické kartě), *CUDA* a *OpenCL*, *Thrust* a *OpenACC*, přičemž *OpenMP* a *OpenACC* verze pro *GPU* využívá knihovnu *cuFFT*, kterou volá v rámci interoperability s knihovnou *CUDA*.

K překladu *C++* verze byl využit nástroj *Cmake*. Překlad byl testován na operačním systému *Ubuntu 20.04 LTS* a *Ubuntu 18.04 LTS*.

4.2 Docker

Vzhledem k velkému množství požadovaných knihoven a nástrojů je program špatně portovatelný na různé systémy. K řešení tohoto problému jsem využil nástroj *Docker*, který umožňuje zabalit program do tzv. kontejneru, který je odlehčeným obrazem operačního systému. Takový kontejner potom může být spuštěn v jiném operačním systému. Oproti klasické virtualizaci je spuštění kontejnerů téměř okamžité a proto je možné *Docker* použít nejen pro usnadnění vývoje, ale i pro nasazení aplikace.

Obrazy kontejneru v *Dockeru* jsou popsány speciálním souborem `Dockerfile`, který obsahuje sekvenci příkazů pro vytvoření obrazu. Typicky se jedná o kopii souborů potřebných k běhu vlastní aplikace dovnitř obrazu, dále pak příkazy pro nastavení prostředí pro

překlad (instalace knihoven, nástrojů, utilit, překlad potřebných nástrojů apod.) a nakonec obvykle příkazy pro překlad aplikace samotné.

Kontejnery lze spustit v interaktivním režimu pomocí příkazu `docker run -it`, což je užitečné pro vývoj aplikace uvnitř kontejneru. V „ne-interaktivním“ režimu pak kontejner může sloužit jako aplikace samotná. Kontejnery také umožňují namapovat část hostitelského souborového systému do svého vlastního souborového systému (přepínač `-v`). To umožňuje pracovat se zdrojovými kódy mimo kontejner a tedy pouze využít jeho prostředí pro překlad. Podobně je možné do kontejneru dostat vstupní soubory aplikace.

Kontejneru lze při spuštění přiřadit grafické karty pomocí argumentu `-gpus all`. Na operačním systému *Linux* je navíc nutné (v případě grafických karet *nvidia*) nainstalovat *nvidia-container-toolkit* poskytující nástroj *nvidia-docker*. V systému *Windows* k tomuto účelu stačí výchozí *Docker*.

Kvůli uvedeným výhodám kontejnerizace oproti nativním aplikacím a plnohodnotným virtuálním strojům byl vytvořen obraz v nástroji *Docker* (`nvidia-docker`), založený na obrazu `nvidia/opencl:devel-ubuntu18.04`. Tento obraz byl vybrán, protože zabírá nejméně místa na disku z testovaných alternativ. Vytvořený *Dockerfile* obsahuje kompletní vývojové prostředí jak pro *C++*, tak i *Python* verzi. Obraz pro vývoj je poměrně velký a zabírá asi 10 GB. Z převážné části se jedná o vývojový balík *nvidia HPC toolkit*. Obraz určený pouze pro běh aplikace je mnohem menší a s využitím experimentálního přepínače `--squash`, který dokáže sloučit několik vrstev vytvořeného obrazu do jediné vrstvy a automatickým smazáním nepotřebných knihoven, se podařilo dosáhnout velikosti obrazu pod 1 GB. Při tvorbě obrazu se nejprve instalují potřebné knihovny a nástroje přes `apt`, následně se stahuje *NVHPC Toolkit* a *FFTW 3*. Vytvoření obrazu proto může trvat přibližně půl hodiny.

Výhodou použití nástroje *Docker* je také možnost spuštění kontejneru na *Windows 10/11* v subsystému *WSL 2*, který má podporu pro grafické karty *nvidia*. Podpora *WSL 2* pro *OpenCL* však zatím chybí, a proto běh této verze programu končí chybou. Docker také umožňuje běh na *MacOS*, tato možnost ale nebyla testována.

4.3 Vstupní formát

Aplikace očekává vstup ve formátu *HDF5* (Hierarchical Data Format). Jak název napovídá jedná se o hierarchický formát pro ukládání velkých objemů dat. V jazycích *Matlab*, *C/C++* a *Python* existují rozhraní pro práci s tímto formátem. Jeho výhodou je možnost komprese ukládaných dat. Data jsou zde uložena v jakési adresářové struktuře, kde výchozí adresář/skupina je označena „/“. V této skupině je možné vytvářet podskupiny, ukládat datasety (n-rozměrné matice různých typů) a atributy (jednoduché hodnoty, čísla a textové řetězce).

Přesně je vstupní formát popsán v dokumentaci *Python* verze, v hlavičce souboru `kwave_diffusion.py` nebo v Příloze C.

Formát je rozdělen do několika skupin. První skupina `simulation_flags` specifikuje, zda má být při simulaci využita perfuze, zda je povolena *k-Space* korekce a také udává typ okrajové podmínky domény (periodická, izolující a vodivá).

Skupina `grid_properties` popisuje velikost domény v dimenzích X, Y a Z, přičemž X je dimenze s nejrychleji se měnícím indexem (kontinuální v paměti) a Z je dimenze s nejpomaleji se měnícím indexem (uložena s rozestupem v paměti). V případě $2D$ domény je velikost v $Z = 1$. Je zde uložen také počet kroků simulace a časový rozdíl mezi dvěma kroky simulace. Nakonec jsou zde uloženy rozestupy mezi jednotlivými body simulované domény v metrech.

Skupina `medium_properties` popisuje vlastnosti materiálu/média. Jsou zde uloženy příznaky o reprezentaci materiálu (zda se jedná o homogenní, nebo heterogenní materiál a jakým způsobem jsou zadány jeho vlastnosti). Dále uchovává některé jednoduché hodnoty jako ambientní teplota krve a referenční koeficienty pro perfuzi a difuzi (obvykle minimum, maximum nebo průměr hodnot matic difuzních/perfuzních koeficientů). Nejdůležitější v této skupině jsou ale matice obsahující samotné vlastnosti materiálu jako je hustota tkáně $[kg/m^3]$, tepelná vodivost $[W/(m \cdot K)]$ a měrná tepelná kapacita $[J/(kg \cdot K)]$. V případě, že je povolena perfuze, očekává se také buď přímo perfuzní koeficient $[s^{-1}]$ nebo matice s hustotou krve $[kg/m^3]$, měrnou tepelnou kapacitou krve $[J/(kg \cdot K)]$ a rychlostí perfuze $[s^{-1}]$. Tyto matice jsou buď jednoprvkové v případě homogenního média, nebo mají stejný tvar jako simulační doména v případě heterogenního média.

Ve skupině `sources` jsou uloženy popisy zdrojů tepla. Zdroje mohou být uloženy dvěma způsoby: jako lineární indexy, nebo vrcholy kvádru. Při použití lineárních indexů se jako maska ukládá pole indexů do simulační domény (jedno-dimenzionální indexy do zploštěné matice simulační domény) a k těmto indexům odpovídající hodnoty tepla dodávaného v každém z těchto bodů $[W/m^3]$. Při použití vrcholů kvádru se ukládá jako maska pouze šestice indexů popisujících vrcholy kvádru a k této masce odpovídající matice hodnot dodávaného tepla $[W/m^3]$ s velikostí kvádru uvedeného v masce. Zdrojů může být více a jejich typ je určen pro každý z nich zvlášť. Formát také umožňuje určit počáteční a koncový časový krok simulace a tím umožnit jejich vypnutí a zapnutí během simulace bez nutnosti spouštět simulaci vícekrát zřetězeně za sebou s mírně změněnými parametry. Kromě těchto zdrojů je v této skupině také uložena výchozí distribuce teplot v doméně při začátku simulace, která má vždy tvar matice o velikosti celé domény.

Poslední skupinu `sensors` tvoří popis senzorů simulace. Senzory mohou být stejně jako zdroje tepla uloženy dvěma způsoby: jako lineární indexy, nebo vrcholy kvádru. Na rozdíl od zdrojů je však uložena pouze maska. Senzory lze nastavit pro ukládání různých výstupů simulace. Je možné ukládat průběžnou teplotu, maximální teplotu, hodnotu poškození tkáně a také finální hodnoty maxima, teploty a poškození tkáně. Rovněž lze určit počátek a konec ukládání dat ze senzorů, stejně jako pod-vzorkování v časové doméně (měření každý n-tý krok.)

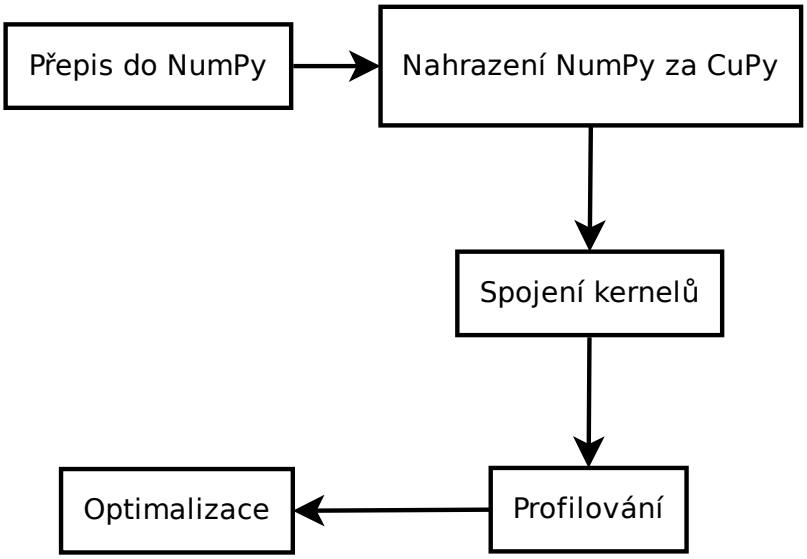
4.4 Výstupní formát

Výstupní formát aplikace je velmi podobný vstupnímu formátu a je také uložen ve formátu *HDF5*. První dvě skupiny (`simulation_flags` a `grid_properties`) jsou identické jako u vstupního formátu. Sekce `medium_properties` obsahuje totožné příznaky jako ve vstupním souboru, ale neukládá matice s vlastnostmi materiálu kvůli velikosti souboru. Podobně tomu je u sekce `sources`, která je rovněž stejná jako ve vstupním souboru, ale chybí výchozí distribuce teploty a příspěvky jednotlivých zdrojů.

Skupina `sensors` je také identická se vstupním formátem, ale navíc je rozšířena o výstupní data senzorů. Finální hodnoty teplot a poškození jsou uloženy jako jednoduché matice s velikostí simulační domény. Výstupy senzorů jsou uloženy jako matice s jedním rozměrem navíc, kterým je časový krok. Vzhledem k tomu, že je možné měřit jen v některých časových krocích a navíc výstup pod-vzorkovat, neodpovídají indexy časovým krokům přímo, ale pouze určují pořadí ve kterém byly hodnoty zapsány. Pro získání skutečné hodnoty časového kroku t_i je nutné index i vynásobit faktorem pod-vzorkování k a přičíst počáteční čas záznamu: $t_i = i \cdot k + t_{start}$.

4.5 Postup při transformaci implementace z jazyka *Matlab*

Postup aplikovaný při transformaci stávajícího kódu z *Matlabu* do *Pythonu* pro výpočet na grafické kartě je zobrazen na Obrázku 4.1.



Obrázek 4.1: Postup transformace *Matlab* kódu do *Pythonu* pro vykonání na *GPU*.

Prvním krokem implementace byla kopie stávající funkčnosti (v omezené podobě, homogenní médium bez perfuze nebo zdroje) z jazyka *Matlab* do jazyka *Python* s využitím procesorové knihovny *NumPy*, což je poměrně přímočarý úkon. Ukázka tohoto postupu je vidět na Algoritmu 4.1.

```
% Matlab
d_term = diffusion_p1 .* diffusion_p2 .* IT( -k.^2 .* kappa .* FT(T) );

# Python
d_term = diffusion_p1 * diffusion_p2 * IT(-k**2 * kappa * FT(T))

```

Algoritmus 4.1: Porovnání výpočtu difuzního termu v *Matlabu* a *Pythonu*

Také byla vypuštěna funkcionalita pro simulaci v $1D$, z důvodů příliš malých řešených úloh. Implementace v této fázi měla násobně horší výkon než původní verze. To bylo způsobeno tím, že *NumPy* je jedno-vláknová knihovna, zatímco *Matlab* dokáže využít všechna jádra procesoru (i když ne zcela efektivně, jak ukazují výsledky varianty s *OpenMP* na *CPU*).

Dalším krokem bylo prosté nahrazení knihovny *NumPy* knihovnou *CuPy*. Toto jednoduché nahrazení vedlo zhruba k šestinásobnému zrychlení oproti *Matlab* implementaci. V této fázi probíhal výpočet kompletně na *GPU*. Toto řešení mělo zásadní nedostatek v podobě extrémního množství samostatných kernelů. Každá jednoduchá operace jako násobení nebo sčítání je realizována jako nezávislý *CUDA* kernel, což má za následek nutnost ukládat každý mezi-výsledek do globální paměti *GPU* a extrémně nízkou aritmetickou intenzitu jednotlivých kernelů v úloze již tak dost omezené propustností paměti.

V další fázi došlo ke spojení více kernelů do jednoho, kde to bylo možné. *CuPy* k tomuto účelu nabízí několik mechanismů (sekce 3.1). Z těchto jsem se rozhodl pro definici uživatelských kernelů, především kvůli jejich spolehlivosti oproti *JIT* a dekorátoru *fuse* a ne o tolik

náročnější implementaci. Spojení všech individuálních kernelů pro každou operaci vedlo k dramatickému zrychlení. Tento proces je vidět na Algoritmu 4.2, kde byla část výpočtu z Algoritmu 4.1 nahrazena uživatelsky definovaným kernelem. Tento kernel spojuje několik operací násobení do jednoho kernelu, a tím eliminuje zbytečné přístupy do globální paměti. Podobně by bylo možné nahradit i násobení po provedení zpětné transformace (`IT()`).

```
# User defined kernel
kernel = cupy.ElementwiseKernel(
    """
        T ft,
        float32 k,
        float32 kappa
    """,
    "T res",
    """
        res = -k * k~* kappa * ft;
    """,
    "dTerm_kernel1",
)

# Kernel used in d_term calculation
d_term = diffusion_p1 * diffusion_p2 * IT(kernel(FT(T), k, kappa))

```

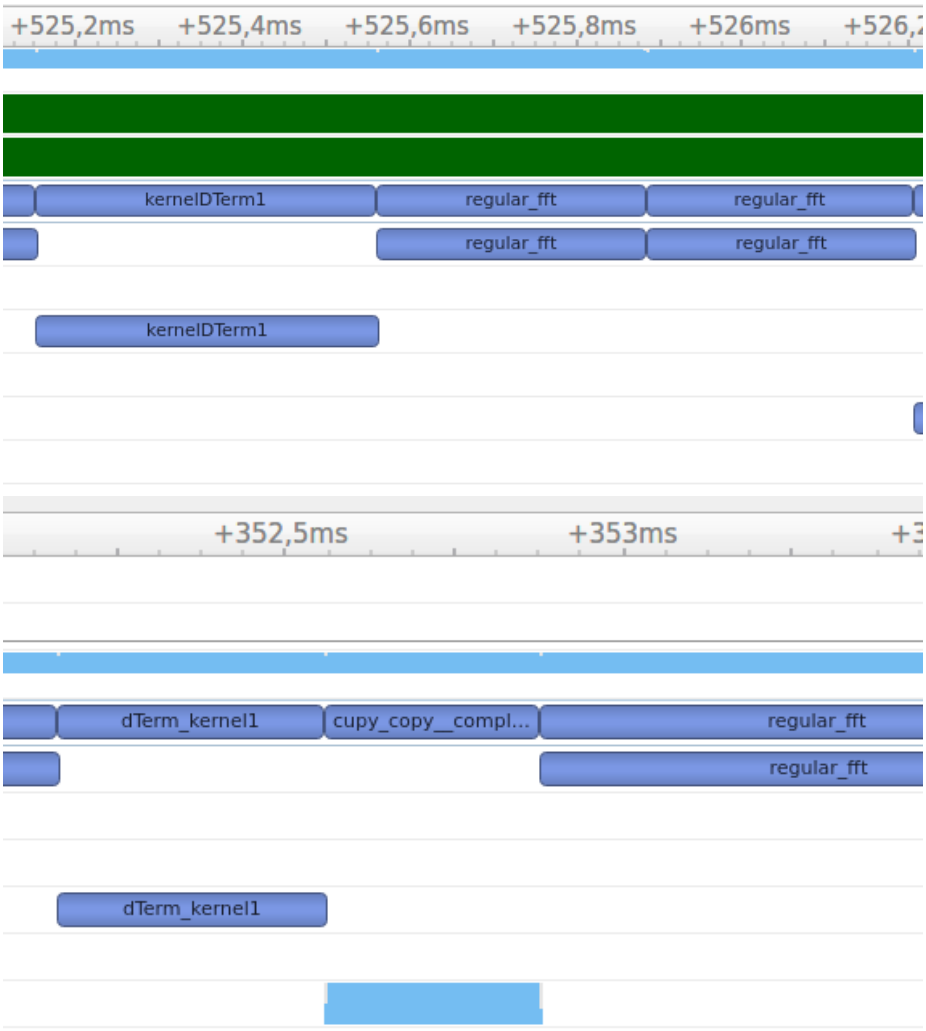
Algoritmus 4.2: Definice uživatelského kernelu a jeho využití při výpočtu difuzního termu.

V poslední fázi transformace byly provedeny další menší optimalizace. Nejdůležitější z nich bylo využití diskrétní Fourierovy transformace typu *R2C* (real to complex), která generuje na výstupu matici poloviční velikosti (z důvodu hermitovské symetrie), což ušetří zhruba polovinu výpočtů. Z dalších méně významných optimalizací lze uvést před-počítání některých hodnot a jejich dosazení přímo do textu kernelu jako konstanty, explicitní mazání dočasných objektů v paměti *GPU* (pomocí destruktoru), spojení volání kernelů v jednom řádku (pomáhá *CuPy* s lepším využitím paměti *GPU*) a odstraňování problémů odhalených při profilování aplikace.

Při profilování bylo odhaleno několik problémů s využitím *CuPy*, které je nutno řešit přímo nebo nepřímo. Vzhledem k tomu, že *CuPy* se stará o veškerou alokaci potřebné paměti i pro mezi-výpočty, objevují se často zbytečné kernely pro kopie paměti na *GPU*, kterým nelze vždy předcházet. Profiling odhalil ale i jiné problémy jako například to, že volání zpětné diskrétní Fourierovy transformace s normalizací způsobuje vytvoření velmi neefektivního kernelu pro normalizaci. Proto je nutné vždy ověřit profilováním, že generované kernely dávají smysl. Tyto problémy jsou také hlavním důvodem, proč je implementace v *CuPy* méně efektivní než *C/C++* verze s *CUDA*. Jeden takový případ je demonstrován na obrázku 4.2.

4.6 Diskrétní trigonometrické transformace

Pro výpočet diskrétních trigonometrických transformací na *GPU* byl vytvořen *Python* modul `dtt.py`. Tento modul umožňuje provést $1D$, $2D$ a $3D$ diskrétní sinové a cosinové transformace typu 1 (a také cosinové typu 2) nad poli knihovny *CuPy*. Modul je závislý pouze na knihovně *CuPy*. Transformace lze provést vektorově nebo více-dimenzionálně (funkce `dtt()` a `dttn()`). Modul využívá diskrétní Fourierovu transformaci v knihovně *cuFFT* pro výpočet trigonometrických transformací v jedné dimenzi nad rozšířeným polem ($2(N-1)$ pro *DCT*, resp. $2(N+1)$ pro *DST*) [12][11].



Obrázek 4.2: Příklad vzniku neefektivního kernelu při použití *CuPy* (dole), oproti implementaci v *CUDA* (nahoře). Mezi výpočtem uživatelského kernelu (*d-term*) a Fourierovy transformace dochází ke vzniku kernelu, který pouze kopíruje data, přestože je výsledek kernelu přímo předán funkci pro odpovídající transformaci a žádné explicitní kopírování voláno není. Během práce na optimalizaci se tento kernel nepodařilo odstranit bez použití explicitní alokace paměti. Zůstává tak hlavním důvodem zpomalení této implementace při základní verzi simulace.

Pro výpočet *DCT-I* zůstává prvních N prvků zachováno a zbylé prvky rozšířeného pole (do $2(N-1)$) se doplní symetricky (bez opakování posledního prvku původní posloupnosti) [12]. Například:

$$(a, b, c, d, e) \rightarrow (a, b, c, d, e, d, c, b)$$

Na takto rozšířenou posloupnost lze aplikovat diskrétní Fourierovu transformaci a použít její reálnou složku. Je vhodné použít transformaci typu *R2C*, která generuje jen poloviční matici a reálná složka jejího výstupu přímo odpovídá *DCT-I* původní posloupnosti bez nutnosti dalšího zpracování. Přestože je nutné rozšíření na $2(N-1)$ prvků (např. oproti *DCT-II* [7]), paměťové nároky jsou při použití *DFT* přibližně stejné, protože bez použití

rozšíření (jako u *DCT-II*) je nutné ukládat zbytečnou imaginární část, která je stejně velká jako reálná část.

Stejně jako *DCT-I* lze i *DST-I* vypočítat pomocí diskrétní Fourierovy transformace. K tomu je nutné provést rozšíření původního signálu na $2(N+1)$ prvků [11]. První polovina signálu je nula následovaná původním signálem. Druhá polovina signálu je nula následovaná obráceným původním signálem vynásobeným -1. Například:

$$(a, b, c) \rightarrow (0, a, b, c, 0, -c, -b, -a)$$

Stejně jako u *DCT-I* je nad tímto upraveným vstupem provedena *R2C* Fourierova transformace. Sinovou transformaci původního vstupu lze získat výběrem prvků $\{1, ..., N+1\}$ imaginární části a jejich vynásobením -1.

Pro více-dimenzionální trigonometrické transformace se využívá vlastnosti separability [4]. Odpovídající vektorové $1D$ transformace jsou aplikovány postupně ve všech dimenzích, čímž se získá požadovaná více-dimenzionální transformace [8]. K tomu je nutné po každé transformaci v jedné dimenzi matici transponovat, aby se vystřídaly všechny rozměry. Následně je matice transponována zpět do původního stavu. K transpozici matice se využívá řešení z knihovny *CuPy*. Pro $2D$ transformaci se provedou dvě běžné transpozice. V případě $3D$ transformace se provádí transpozice v tomto pořadí:

$$DTT \rightarrow (2, 1, 0) \rightarrow DTT \rightarrow (0, 2, 1) \rightarrow DTT \rightarrow (1, 2, 0),$$

kde (A, B, C) značí permutaci dimenzí vstupní matice po transpozici a C je dimenze s nejrychleji se měnícími indexy, která je v každém kroku využita k výpočtu vektorové transformace (DTT). Toto pořadí transpozic bylo experimentálně nalezeno jako nejrychlejší možnost.

Toto řešení trigonometrických transformací je ovšem neefektivní jak paměťově, tak časově, neboť vyžaduje D Fourierových transformací nad $\sim 2N$ daty (kde D je počet dimenzí a N je velikost dat). Nepodařilo se ale najít knihovnu, která by tyto transformace prováděla přímo. *cuFFT* ani *clFFT* tuto funkcionalitu na rozdíl od *FFTW* (která je jen pro *CPU*) neposkytují.

4.7 Architektura *Python* verze

Python verze sestává z hlavní třídy `HeatDiffusionSolver`, která se stará o načítání a ukládání dat a spouští hlavní smyčku simulace. Dále tato třída využívá třídy `HeatSensor` a `HeatSource`, které se starají o logiku ukládání dat ze senzorů a výpočet příspěvku zdrojů, v závislosti na způsobu specifikace masky zdrojů/senzorů. Dále využívá modul `dtt.py`, který implementuje více-dimenzionální trigonometrické transformace. O textový výstup aplikace se stará třída `Logger`.

Při inicializaci objektu třídy `HeatDiffusionSolver` dochází k načítání dat ze vstupního souboru a vytvoření objektů kernelů *CuPy*. Také jsou před-počítány některé potřebné hodnoty a matice použité v hlavní simulační smyčce. Hodnoty invariantní pro celý běh simulace jsou dosazeny přímo do textu vytvořených kernelů pomocí formátovaných řetězců v *Pythonu* a není nutné je předávat jako parametry kernelů, což je výhoda dynamické kompilace.

Nejdůležitější metodou tohoto solveru je `takeTimeStep()`. Tato metoda nejprve vypočte některé zbývající pomocné hodnoty jako např. matice pro *k-Space* korekci a otevře požadované výstupní soubory, kterými mohou být buď výstupní soubor `.h5` nebo soubor obsahující video se záznamem simulace. Ve výstupním souboru jsou vytvořeny potřebné

datasety pro uložení záznamu ze senzorů. U video výstupu se specifikuje formát a bitová hloubka.

Následuje spuštění hlavní smyčky simulace. Zde je prvním krokem výpočet příspěvků ze zdrojů. Toto se děje jen v časových krocích, kde byl nějaký zdroj buď aktivován nebo deaktivován. Další krok je výpočet perfuze pokud je povolena. Jedná se o poměrně jednoduchý výpočet, kde mezi transformacemi probíhá maximálně jedno násobení nebo sčítání. Proto zde nebyl vytvořen žádný uživatelský kernel. Hlavní smyčka pokračuje výpočtem difuzního termu. Zde jsou dvě možnosti výpočtu pro homogenní a heterogenní médium. U homogenního média probíhá transformace následovaná uživatelským kernelem s několika násobeními, následovaná zpětnou transformací. U heterogenní varianty je situace složitější a dopřednou a zpětnou transformaci je potřeba provést pro každou dimenzi zvlášť. Toto je realizováno pomocí jednoduchého *lambda* výrazu aplikovaného na všechny dimenze. Kromě toho byly opět sloučeny všechny operace násobení do jednoho kernelu tam, kde to dává smysl (kde se jich vyskytuje více v řadě).

Posledním krokem výpočtu jedné iterace je aktualizace teplot a integrálu poškození. To je realizováno jedním uživatelským kernelem se dvěma výstupy.

Na konci každé iterace se také ukládá maximum teplot (je-li zapotřebí), zapisují se data ze senzorů, zapisuje se jeden snímek do video výstupu a každých 50 iterací se vypisuje postup simulace a ukládají se informace o průběžné spotřebě paměti. Po skončení hlavní smyčky se ještě zapisují výsledky do výstupního souboru a další informace na standardní výstup. Tím simulace končí.

4.8 Architektura *C++* verze

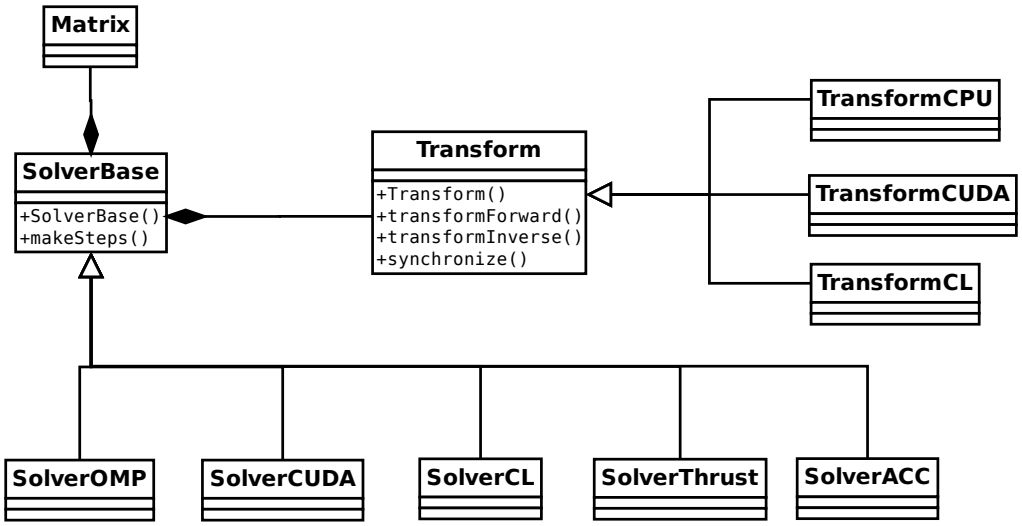
Kromě implementace v *Pythonu* vznikla také *C++* verze obsahující implementaci v *OpenMP*, ale také v *CUDA*, *Thrust*, *OpenACC* a *OpenCL*. Struktura aplikace umožňuje jednoduché přidání dalších implementací pro účely porovnání výkonnosti.

Architektura *C++* verze je znázorněna na diagramu 4.3. Je zde využit návrhový vzor strategie pro samotnou simulaci a Fourierovu transformaci, což umožňuje za běhu zvolit konkrétní implementaci. Verze v *C++* zatím obsahuje pouze implementaci základní simulace s homogenním médiem a *k-Space* korekcí, bez perfuze a bez zdroje tepla.

Pro načítání a uchování dat na procesoru byla vytvořena šablonová třída `Matrix`. Třída `SolverBase` je zodpovědná za načítání a přípravu vstupních dat na procesoru. Třídy `SolverOMP`, `SolverACC`, `SolverThrust`, `SolverCUDA` a `SolverCL`, které jsou jejími potomky, implementují vlastní simulaci a starají se o přesun dat na *GPU* v závislosti na specifické *GPGPU* knihovně, kterou využívají.

Nezávisle na zvolené implementaci třídy `SolverBase` existuje třída `Transform`, reprezentující transformace, které lze realizovat pomocí různých knihoven. Z této třídy dědí třídy `TransformCPU`, `TransformCUDA` a `TransformCL`, které realizují *DFT* postupně pomocí *FFTW*, *cuFFT* a *clFFT*. Tímto způsobem mohou dvě různé implementace šíření tepla sdílet jednu implementaci *DFT* jako například *OpenMP* verze na *GPU* a *CUDA* verze, které obě využívají `TransformCUDA`.

Všechny tyto třídy jsou zkompilovány v jednom spustitelném souboru (a jedné dynamicky linkované knihovně pro *CUDA* moduly). Použitou technologii lze zvolit přepínačem přímo při spuštění, což umožňuje jednoduché porovnání výkonnosti jednotlivých implementací.



Obrázek 4.3: Diagram tříd *C++* verze. Aktuální řešení umožňuje simulaci v *CUDA*, *OpenCL*, *OpenACC*, *Thrust* a *OpenMP* (jak na procesoru, tak na grafické kartě).

4.8.1 *CUDA* implementace

CUDA implementace se nachází v souboru `solverCUDA.cu`. Třída *CUDA* solveru provádí v konstruktoru alokaci paměti na *GPU* a kopii dat z matic zděděných ze základní třídy. Dále se inicializuje objekt třídy `TransformCUDA`, který inicializuje dopředné a zpětné plány transformací v *cuFFT*.

Volání metody `makeSteps()` spouští simulační smyčku. Měření času běhu hlavní smyčky je realizováno pomocí standardní knihovny *chrono*. Hlavní smyčka sestává ze čtyř částí. Nejprve je provedena dopředná transformace teploty s využitím objektu `TransformCUDA`. Následně je volán *CUDA* kernel pro výpočet části difuzního termu mezi transformacemi (řádek 14 Algoritmu 2.1). Poté je obdobně provedena zpětná transformace. Poslední částí smyčky je dokončení výpočtu difuzního termu a přičtení přírůstku teploty a integrálu poškození v posledním kernelu. Po ukončení hlavní smyčky dojde ke kopii dat z matic teploty a poškození zpět na procesor.

4.8.2 *OpenCL* implementace

OpenCL implementace se nachází v souboru `solverCL.cpp`. Tato implementace je velmi podobná *CUDA* verzi. Prvním rozdílem je, že kernely jsou zapsány jako textové řetězce. V konstruktoru probíhá oproti *CUDA* verzi složitější inicializace, protože rozhraní *OpenCL* je více explicitní. Je zde nutné získat *ID* výpočetní platformy, *ID* zařízení, které bude použito pro výpočet (zde *GPU*). Dále je nutné vytvořit kontext a příkazovou frontu, do které se budou vkládat jednotlivé výpočetní kernely. Dalším krokem je kompilace kernelů ze zdrojových textových řetězců. Jakmile je zdrojový kód zkompilován, jsou vytvořeny jednotlivé kernely jako atributy této třídy. Poslední krok je podobný jako v případě *CUDA* verze. Jsou zde inicializovány buffery na *GPU*, které se použijí pro výpočet, a provádí se kopie paměti na *GPU*. Také se zde inicializuje objekt třídy `TransformCL`, který inicializuje dopředné a zpětné plány transformací v *clFFT*.

Hlavní smyčka výpočtu vypadá rovněž velmi podobně jako u *CUDA* verze. Jediným rozdílem je zde spouštění kernelů. Protože bylo využito základní *C* rozhraní *OpenCL*, je

nutné před spuštěním zadat každý parametr spouštěného kernelu v samostatném volání funkce `clSetKernelArg`. Tento přístup by mohl být zjednodušen využitím nějaké *C++* nadstavby nad *OpenCL*[1].

Po ukončení hlavní smyčky opět dojde k výpočtu uplynulého času pomocí knihovny *chrono* a kopii výstupních dat zpět na *CPU*.

4.8.3 *OpenMP* implementace

OpenMP implementace se nachází v souboru `solverOMP.cpp`. Při inicializaci tohoto solveru je možné vybrat libovolný způsob provedení transformace. Při kompilaci s *nvc++* jsou podporovány třídy `TransformCUDA` a `TransformCPU`. V tomto překladači není podpora interoperability s *OpenCL*.

Před spuštěním hlavní smyčky jsou vytvořeny ukazatele na data vstupních a pomocných matic, protože *OpenMP* nemá podporu pro kontejnery *STL*. Alokace paměti a kopie dat na grafickou kartu probíhá pomocí direktivy `omp target data map(tofrom: ...)`. Data jsou před vstupem do tohoto regionu nakopírována na grafickou kartu a při opuštění tohoto bloku kódu probíhá kopie zpět na procesor.

V hlavní smyčce probíhá nejprve dopředná transformace teploty. Zde je využita klauzule `omp target data user_device_ptr()`, která umožní přístup k zadaným datům jako k běžnému ukazateli do paměti grafické karty. Po provedení transformace probíhá synchronizace z důvodů problémů interoperability *OpenMP* a *CUDA* popsaných v sekci 3.2. Kernely jsou zapsány jako obyčejná smyčka `for` a jsou paralelizovány na *GPU* pomocí direktivy `omp target teams distribute parallel for`. U kernelu pro aktualizaci teploty jsem narazil na problém, kde při překladu pro vykonání na grafické kartě nelze použít funkci `powf` pro výpočet integrálu poškození. Proto bylo toto volání nahrazeno voláním `expf` a `logf` takto:

$$a^b = e^{b \cdot \log(a)}$$

Tato změna má mírně negativní vliv na výkon *CPU* i *GPU* verze. U *CPU* verze tato změna není nutná ale byla zachována z důvodů jednotného *CPU*/*GPU* kódu. U *OpenACC* tento problém nenastal.

4.8.4 *OpenACC* implementace

OpenACC implementace se nachází v souboru `solverACC.cpp`. Při inicializaci tohoto solveru je možné vybrat libovolný způsob provedení transformace. Při kompilaci s *nvc++* je však podporován jen `TransformCUDA`.

Při volání metody `makeSteps()` nejprve dojde k uložení ukazatelů na data ze třídy `Matrix` jako ukazatele jazyka *C++*. Následuje alokace paměti a kopie na *GPU*. Toto je realizováno pomocí klauzule `acc enter data copyin`. Hlavní smyčka je umístěna do regionu `acc data present`, který specifikuje, že data jsou přítomna na grafické kartě. Při volání transformace je využita klauzule `use_device()`, která v rozsahu své platnosti umožňuje přístup ke zvoleným ukazatelům jako k běžným ukazatelům v paměti *GPU* a tyto jsou předány do volání metody `transformForward()` a `transformInverse()`. Výpočetní kernely jsou zapsané jako běžné smyčky `for`. Smyčky jsou paralelizovány na grafické kartě pomocí direktivy `acc parallel loop`.

Po skončení hlavní smyčky jsou data zkopírována zpět pomocí direktivy `acc update self` a smazána pomocí direktivy `acc exit data delete`.

[1] *OpenCL C++*: `https://github.khronos.org/OpenCL-CLHPP/index.html`

4.8.5 *Thrust* implementace

Thrust implementace, která se nachází v souboru `solverThrust.cu`, je od ostatních poměrně odlišná. Data jsou zde uložena v `device_vector` objektech, které zapouzdřují paměť *GPU*. Pro alokaci stačí volat metodu `resize`, jak je tomu i u běžného *C++* vektoru. Kopie paměti na *GPU* je rovněž velmi jednoduchá. Stačí přiřadit do `device_vector` libovolný odpovídající *C++* vektor.

Hlavní smyčka opět nejprve volá dopřednou transformaci. Tady je nutné pouze získat „raw“ ukazatel na data z `device_vector`, který je stejně jako u *CUDA* verze předán objektu třídy `TransformCUDA`. Následuje výpočet difuzního termu. K tomu byly využity funktory a *Thrust* transformace. Jednohodnotové parametry jsou předány do konstruktoru funktoru. Matice jsou předány do volání transformace jako iterátory. Tento přístup je výhodný, protože základní transformace může být jen binární a unární, a je tedy možné zadat více parametrů. U heterogenní simulace by ale toto mohl být problém vzhledem k vyššímu počtu matic jako parametrů kernelu a vyžadoval by buď rozdělení do více funktorů, nebo využití *zip* iterátoru. Toto je problém už i při výpočtu integrálu poškození, který využívá stejný vstup jako funktor pro aktualizaci teploty. Funktory v *Thrust* ale umožňují existenci jen jednoho výstupu kernelu. Proto jsou v aktuálním řešení použity dva oddělené kernely pro aktualizaci teploty a výpočet integrálu poškození, což má negativní vliv na výkon.

Zpětná kopie dat na procesor je trochu složitější, neboť *C++* vektor nemá přetížený operátor přiřazení pro *Thrust* vektor. Proto jsem opět využil konverzi na „raw“ ukazatel a volání `cudaMemcpy`.

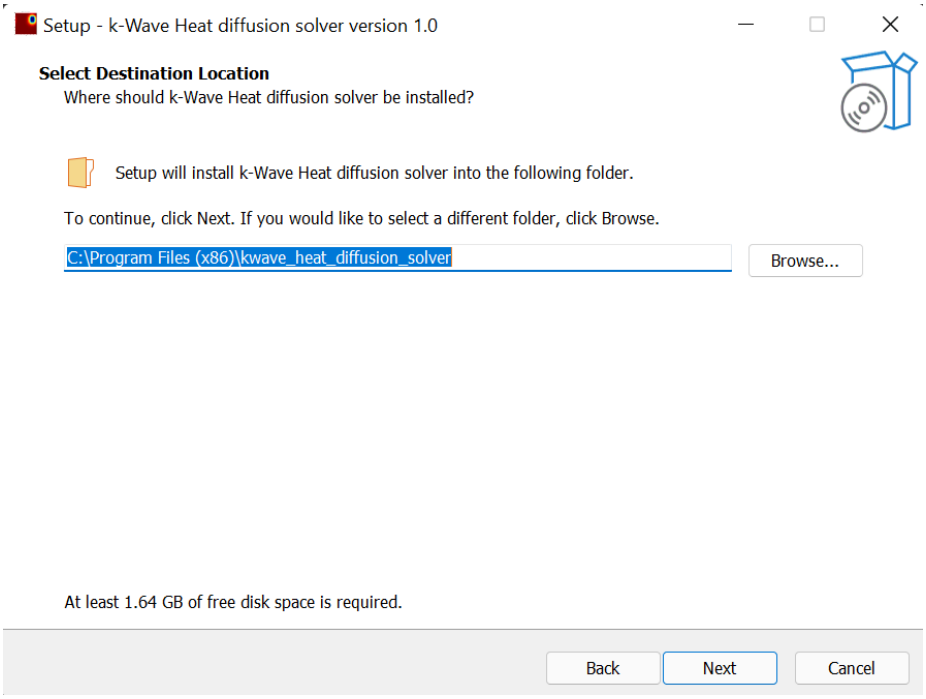
4.9 Vytvoření spustitelných souborů

Vytvořená aplikace byla zabalena do spustitelných souborů tak, aby její běh nevyžadoval žádné externí závislosti nebo nastavení prostředí a tím bylo možné zjednodušit nasazení v praxi. Protože *Python* je primárně interpretovaný jazyk, nejsou při spuštění vytvářeny spustitelné binární soubory. K vytvoření spustitelných souborů z programu v *Pythonu* je nutné použít externí nástroj. Já jsem k tomuto účelu použil nástroj `pyinstaller`[2]. Tomuto nástroji stačí předat název hlavního souboru aplikace a *PyInstaller* z něj automaticky vytvoří spustitelný soubor. Při použití běžných knihoven (*NumPy, SciPy, Matplotlib, ...*) toto obvykle funguje bez jakýchkoli dalších kroků. Při použití méně běžných knihoven mohou nastat problémy. V mém případě toto způsobila knihovna *CuPy*. Proto bylo nutné vytvořit konfigurační soubor pro *PyInstaller* nazvaný `kwave_diffusion.spec`, kde jsou specifikované skryté závislosti balíku *CuPy*. Dále je nutné dodat dynamicky linkované knihovny *CUDA* (`*.dll` nebo `*.so`). Potřebné knihovny se liší pro *Windows* i *Linux*. Po vytvoření nezávislého spustitelného souboru je třeba aplikaci zabalit.

4.9.1 Instalátor

K vytvoření instalátoru pro *Windows* byl použit nástroj *Inno setup*[3]. Spuštěný instalátor lze vidět na obrázku 4.4. K vytvoření instalátoru slouží konfigurační soubor `installer.iss`, který specifikuje soubory, které mají být nainstalovány, licenci, popis, verzi, apod. Tento soubor stačí otevřít v *Inno Setup* a spustit vytvoření balíku. Velikost instalátoru je asi

[2] PyInstaller: `http://www.pyinstaller.org/`
[3] Inno Setup: `http://www.jrsoftware.org/isinfo.php`



Obrázek 4.4: Vzhled instalátoru pro *Windows*.

500 MB. Po instalaci je možné aplikaci volat ze zadaného umístění. Instalátor neupravuje proměnné prostředí.

4.9.2 Balík *.deb*

Kvůli zjednodušení instalace na *Linuxu* byl vytvořen balík `kwave_diffusion.deb`, který je možné nainstalovat jako každý jiný balík přes `apt`. Balík instaluje spustitelné soubory to adresáře `/opt/kwave_diffusion` a rovněž přidá symbolický odkaz na aplikaci do `/usr/local/bin`, aby bylo možné aplikaci odkudkoliv spustit jednoduchým voláním `kwave_diffusion` na příkazové řádce. Pro vytvoření balíku *.deb* je nutné vytvořit adresář s následující strukturou:

```
kwave_diffusion/
├── DEBIAN/
│   └── control
├── opt/
│   └── kwave_diffusion/
│       ├── kwave_diffusion
│       ├── *.so
│       └── ...
└── usr/
    └── local/
        └── bin/
            └── kwave_diffusion
```

Následně se do složky `opt/kwave_diffusion` vloží spustitelné soubory a do složky `usr/local/bin`

symbolický odkaz na aplikaci (s cestou `/opt/kwave_diffusion/...`, nemusí být nutně platný). Ve složce `DEBIAN` je potřeba vytvořit soubor `control` obsahující informace o balíku. Balík lze sestavit pomocí příkazu `dpkg-deb -build`.

4.10 Rozhraní aplikace

Vytvořenou aplikaci lze spouštět dvěma způsoby, buď přímo v příkazové řádce s připravenými vstupními soubory, nebo s využitím stávající *Matlab* implementace.

Spuštění aplikace z příkazové řádky je znázorněno na obrázku 4.5. Aplikace ve výchozím stavu vypisuje vstupní argumenty příkazové řádky, základní parametry simulace, informace o hardware a software a případně porovnání s referenčním výstupem, je-li uložen ve vstupním souboru. Nakonec je vypsán celkový čas hlavní smyčky simulace a také trvání jednoho kroku simulace. Je také možné specifikovat jiné úrovně výpisu na příkazové řádce. V nejnižší úrovni se nevypisuje nic, ve vyšších úrovních se navíc vypisuje průběžný postup simulace včetně odhadovaného času výpočtu. Při nejvyšší úrovni výpisu se také zobrazují kompletní informace o vzniklých výjimkách a záznam o zásobníku volání.

Druhá možnost použití vzniklé aplikace je přímo v Matlabu. Tento způsob lze vidět na Algoritmu 4.3. Jediný rozdíl oproti původnímu způsobu je nutnost specifikovat název vstupního souboru, cestu ke spustitelné aplikaci (případně příkazy k nastavení proměnných prostředí apod. dle potřeby.) a znovu dodat vlastnosti média. To je požadováno, protože původní solver neuchovává všechny potřebné parametry materiálu, ale pouze zpracované koeficienty potřebné ke spuštění hlavní smyčky.

```
% take time steps
kdiff.runExternalSolver("export.h5",
                        Nt,
                        dt,
                        medium,
                        "kwave_diffusion");
% kdiff.takeTimeStep(Nt, dt);
```

Algoritmus 4.3: Příklad volání externího programu z *Matlabu* místo původního volání `takeTimeStep`. Poslední parametr funkce je cesta ke spustitelnému souboru.

4.11 Testování

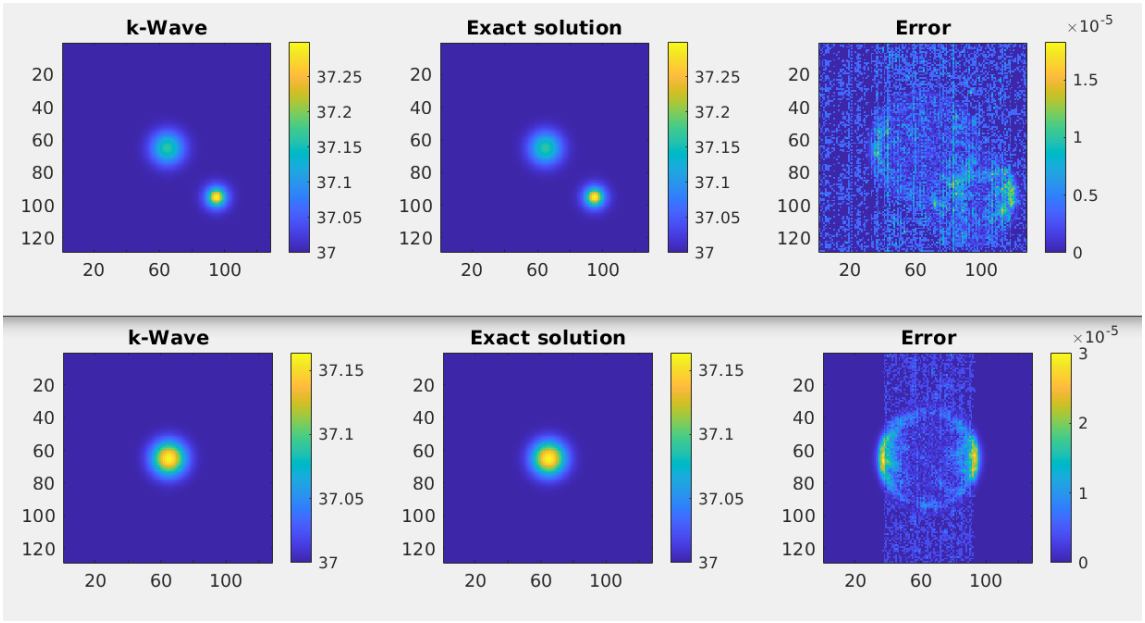
Vytvořené implementace byly testovány pomocí sady regresních testů vzniklých exportováním teplot, hodnot zdrojů, perfuze a vlastností média přímo z implementace v jazyce *Matlab* pro porovnání s původní implementací. Testovací příklady pokrývají různé typy médií (heterogenní i homogenní), různé druhy okrajových podmínek (periodické, izolující i vodivé), velikosti domén a dalších efektů jako je perfuze, *k-Space* korekce a zdroje tepla. Dále byl vytvořen skript (`test.py`), který automaticky a rekurzivně najde všechny *HDF* soubory v zadaném adresáři a ověří absolutní chybu simulace oproti správnému řešení (původní *Matlab* verze). Tímto skriptem je možné testovat jakoukoliv z vytvořených implementací simulace, které disponují jednotným *CLI* rozhraním. Je vyžadováno aby testovaný program načetl vstupní soubor z cesty předané na příkazové řádce pomocí parametru `-i` a následně aby na výstup vypsal hodnotu absolutní chyby oproti referenčním teplotám uvedeným přímo ve vstupním souboru v datasetu `/test/T300`.

```
mk@B450:~/Dokumenty/kwave_diffusion$ ./kwave_diffusion -i ~/Dokumenty/perf/2D_homogeneous/input_128.h5
```

k-Wave - Diffusion solver	
Input file	/home/mk/Dokumenty/perf/2D_homogeneous/input_128.h5
GPU ID	0
Simulation parameters	
Timesteps	300
Delta	0.5 [s]
Domain size	128, 128
Grid spacing	0.001, 0.001 [m]
Boundary	periodic
Heterogeneous	False
Perfusion	False
kSpace correction	True
Sources	0
Sensors	0
Store final	True
Store final max	False
Hardware info	
Node	B450
CPU	x86_64
CPU memory	31.37 [GiB]
GPU	NVIDIA GeForce RTX 3060 Ti
GPU memory	7.79 [GiB]
Software info	
Script name	/home/mk/Dokumenty/kwave_diffusion/kwave_diffusion.py
Python	CPython
Python version	3.8.10
CuPy version	9.5.0
Operating system	Linux-5.4.0-109-generic-x86_64-with-glibc2.14
Running the simulation	
Compare results	
Max absolute error	3.0517578e-05
Max relative error	8.211642e-07
Simulation finished successfully	
Total duration	1.838 [s]
Single step	6.125 [ms]

Obrázek 4.5: Ukázka spuštění programu z příkazové řádky.

Druhým způsobem testování bylo využití vytvořených metod v *Matlabu* pro spuštění existujících unit testů s mým řešením. Tyto unit testy přímo zobrazí chybu oproti referenčnímu řešení jak lze vidět na obrázku 4.6.

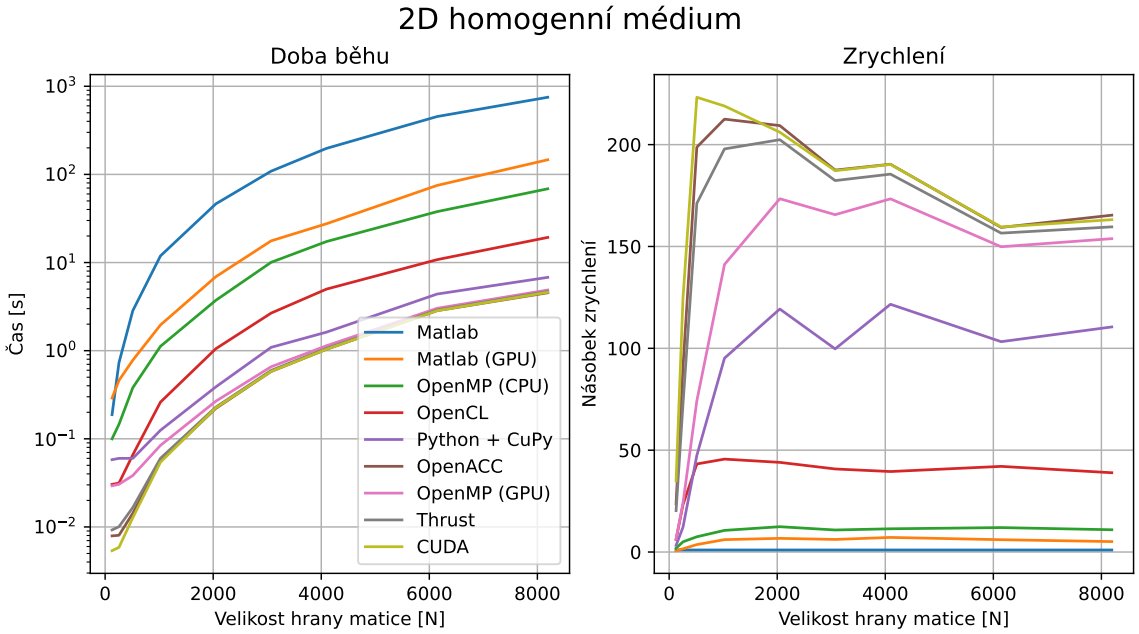


Obrázek 4.6: Ukázka využití unit testů v *Matlabu* pro testování mého řešení. Absolutní chyba (*Error*) se pohybuje v řádu 10^{-5}. V porovnání je vidět moje implementace (*k-Wave*) oproti analytickému řešení (*Exact solution*).

Kapitola 5

Zhodnocení dosažených výsledků

Tato kapitola se bude věnovat vyhodnocení dosažených výsledků. Budou zde porovnány jednotlivé implementace z hlediska časové a paměťové náročnosti a také jejich náročnost na vytvoření. Výsledky byly měřeny na různých variantách simulace, přičemž měření *C++* implementace je omezeno pouze na základní variantu simulace s homogenním médiem a *k-Space* korekcí bez zdroje tepla. Všechny implementace dosahovaly maximální relativní chyby pod 10^{-5} (největší chyby dosahovaly simulace vodivým okrajem domény). Ve všech měřeních je zobrazen čas běhu simulace o 300 časových krocích.

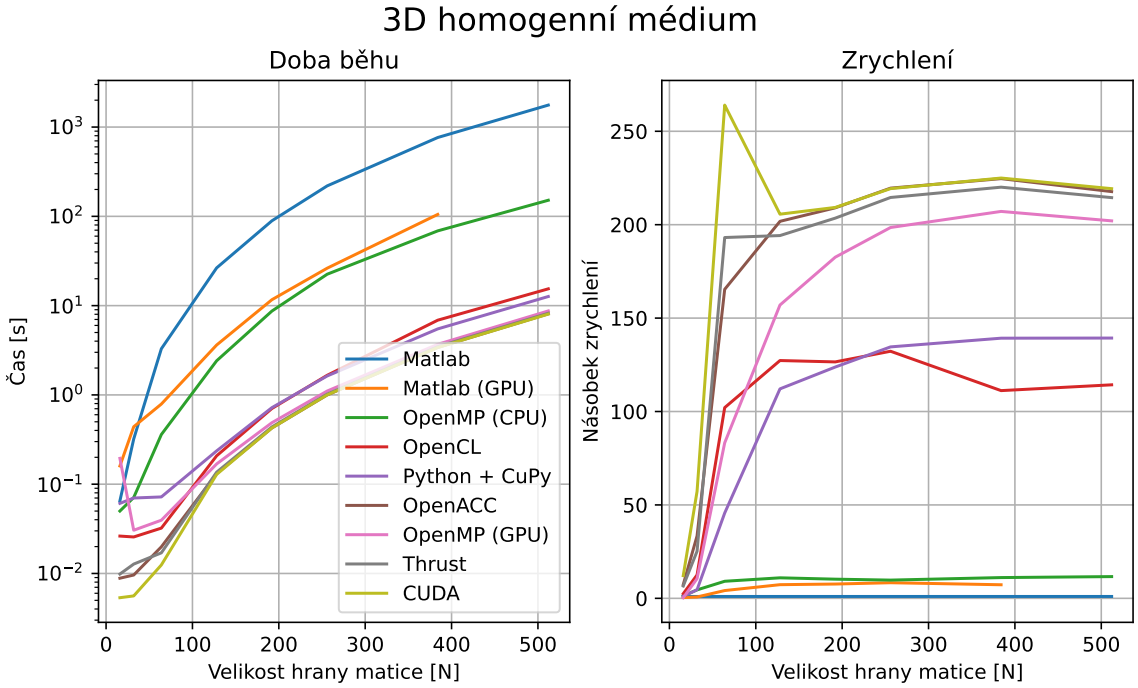


Obrázek 5.1: Srovnání různých verzí 2D homogenní simulace na procesoru *AMD 3700X* a grafické kartě *nvidia RTX 3060ti*.

5.1 Časová náročnost

První srovnání na grafu 5.1 ukazuje běh všech implementací na 2D doméně. Pro měření byl využit procesor *AMD 3700X* a grafická karta *nvidia RTX 3060ti*. *Matlab* verze na procesoru

slouží jako reference pro porovnání a dle očekávání je nejpomalejší. Implementace *OpenMP* varianty na procesoru představuje přibližně desetinásobné zrychlení, což ukazuje, že původní implementace neběží na procesoru optimálně. I při měření bylo patrné, že *Matlab* nedokáže vytížit všechny vlákna procesoru na 100 % ale přibližně jen na 50 %. V porovnání s nejlepší verzí na procesoru je nejlepší verze na grafické kartě zhruba 18× rychlejší.



Obrázek 5.2: Srovnání různých verzí 3D homogenní simulace na procesoru *AMD 3700X* a grafické kartě *nvidia RTX 3060ti*.

Využití automatické paralelizace na grafické kartě v *Matlabu* vede přibližně k pětinásobnému zrychlení. Což dosahuje zhruba poloviny výkonnosti *OpenMP* varianty na procesoru. Tento výsledek odpovídá výkonnosti *CuPy* varianty bez využití jakýchkoli optimalizací (není uvedena v grafu). Důvody tak nízkého zrychlení jsou stejné jako v případě *CuPy*. Je vytvořeno příliš mnoho jednotlivých kernelů pro každou operaci.

Čtvrtá nejpomalejší varianta je *OpenCL* implementace. Zde je problémem nízký výkon knihovny *clFFT*, která brzdí celou simulaci, jak se dalo očekávat z experimentů v sekci 3.4. Zpomalení oproti *CUDA* verzi je však pouze přibližně trojnásobné, což značí, že kromě knihovny *clFFT* (která vykazovala až 5× horší výkon než ekvivalentní volání *cuFFT*), jsou ostatní výpočetní kernely podobně efektivní jako v *CUDA* verzi. I tak je *OpenCL* verze druhá nejpomalejší *GPU* varianta simulace.

Implementace v *Pythonu* přináší v testovaných příkladech alespoň stonásobné zrychlení. Problémem je zde automatická správa paměti *CuPy*, která vytváří zbytečné kernely pro kopie dat v rámci hlavní simulační smyčky. Jednoho takového konkrétního kernelu se mi při optimalizaci kódu nepodařilo i přes veškeré experimentování zbavit a tento kernel je tedy zodpovědný za pozorovaný nárůst času simulace. Dalším zdrojem zpomalení je také *just in time* kompilace kernelů *CuPy* po spuštění hlavní smyčky.

Dle očekávání ale nejvyššího zrychlení dosahují *C++* implementace v *OpenMP*, *OpenACC*, *Thrust* a *CUDA*, které se od sebe časově liší jen minimálně. I zde je však *CUDA* verze

stabilně nejrychlejší, což je způsobeno některými drobnými problémy *OpenMP* a *OpenACC* jako například synchronizace při výpočtu *DFFT* pomocí *cuFFT*.

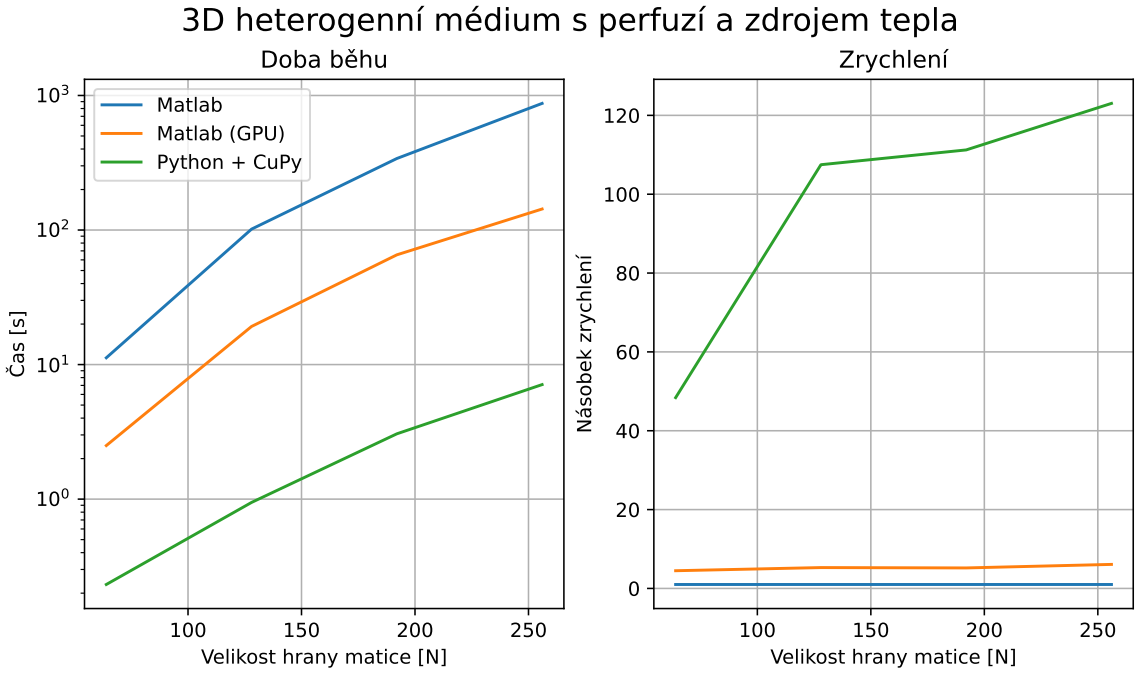
Srovnání na grafu 5.2 ukazuje totéž jako u 5.1, pouze na 3D doméně. Dosažené zrychlení je v tomto případě ještě vyšší. Relativní pořadí jednotlivých implementací však zůstává víceméně nezměněno až na obrácené pořadí mezi *OpenCL* a *Python* verzí v menších doménách. Při spuštění *Matlab* verze na *GPU* došla paměť u domény o velikosti 512^3.

5.1.1 Složitější verze simulace

Kromě porovnání základní verze simulace s homogenním prostředím jsem provedl srovnání i na složitějších variantách simulace. Zde je však k dispozici pro srovnání jen verze v *Pythonu* a *Matlabu*, které jsou z pohledu funkčnosti kompletní. Celkově na všech měřeních lze sledovat stejný trend jak tomu bylo u homogenní varianty. Srovnání opět ukazuje vícevláknovou *Matlab* implementaci na *CPU AMD 3700X* a *Python* implementaci na grafické kartě *RTX 3060ti*.

Graf 5.3 zobrazuje měření simulace na heterogenní doméně s perfuzí a zdrojem tepla. U větších velikostí domény opět pozorujeme alespoň stonásobné zrychlení oproti původní implementaci.

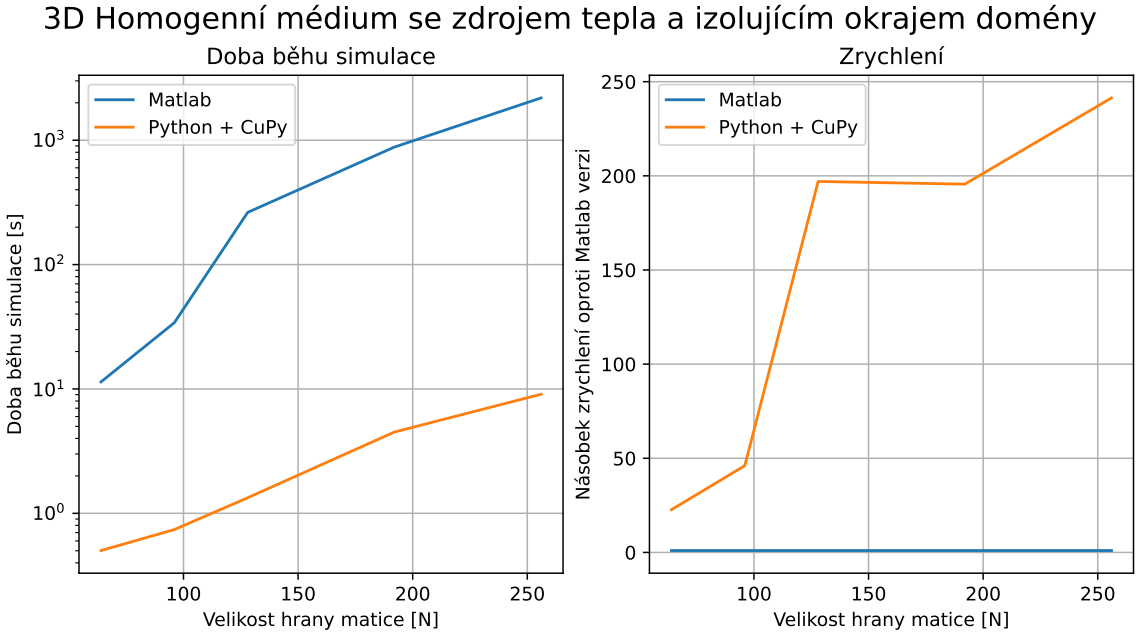
Na Obrázku 5.6 lze vidět zpomalení simulace při použití náročnějších variant jako simulace s heterogenním médiem, perfuzí a izolujícími okrajovými podmínkami. Zpomalení se pohybuje přibližně v rozmezí 4 až 6 krát, což lze očekávat vzhledem k vyššímu počtu provedených Fourierových transformací v jedné iteraci.



Obrázek 5.3: Srovnání *Matlab* implementace na *CPU* a *GPU* a *Python* implementace na *GPU* pro heterogenní médium s perfuzí a zdrojem tepla. Použitý hardware: *AMD 3700X* a *RTX 3060ti*.

Grafy 5.4 a 5.5 zobrazují zrychlení dosažené při simulaci využívající diskrétní trigonometrické simulace. Ke spuštění této varianty simulace v *Matlabu* je nutné nainstalovat

speciální *DTT* modul [1]. Uvedený modul obsahuje předkompilované soubory pro *Windows* a *MacOS*. Tato knihovna nepodporuje transformace na více vláknech a proto je míra zrychlení implementace na *GPU* ještě mnohem vyšší než u ostatních srovnání (zbytek simulace, kromě transformace běží stále paralelně). Moje implementace zde dosahuje zrychlení přibližně dvěstěkrát oproti původnímu řešení.



Obrázek 5.4: Srovnání *Matlab* implementace na *CPU AMD 3700X* a *Python* implementace na *RTX 3060ti* pro homogenní médium s izolujícím okrajem domény a zdrojem tepla.

5.1.2 Karty *AMD*

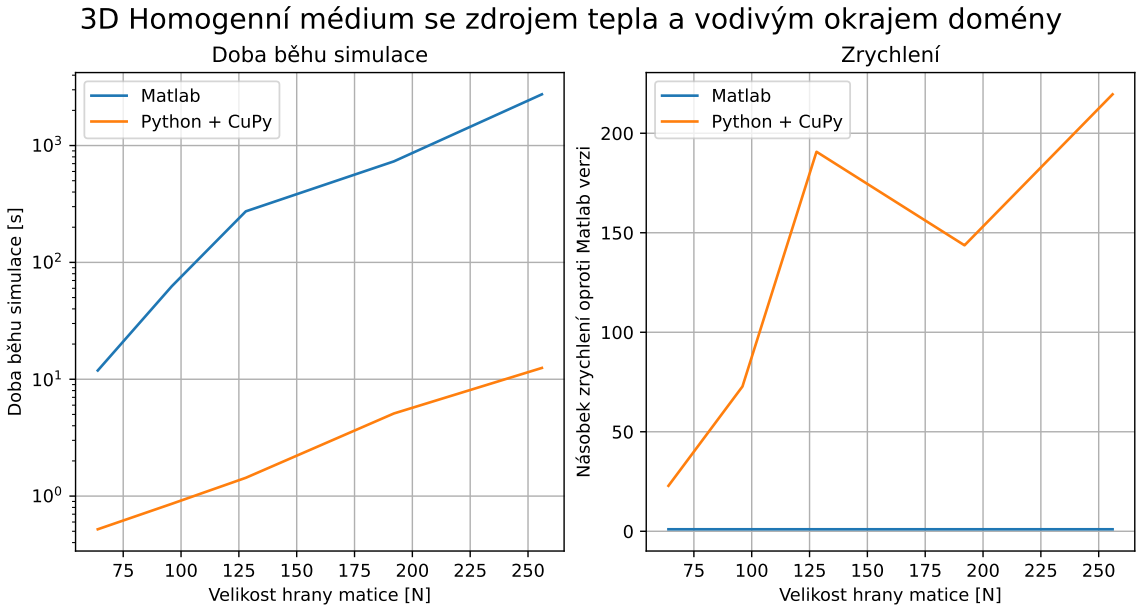
Pro srovnání s kartou *nvidia RTX 3060ti* jsem zvolil kartu *AMD RX 6700XT*. Tyto karty se pohybují ve stejné cenové kategorii a jejich výkon je podobný. *Nvidia* karta nabízí teoretický výkon 16.2 Tflop/s oproti 13.2 Tflop/s v případě *AMD*. Řešená úloha je kvůli nízké aritmetické intenzitě však vázaná především na propustnost paměti. V případě karty *AMD* je to 384 GB/s, v případě *nvidia* karty 448 GB/s.

Srovnání doby běhu lze vidět na grafu 5.7. Na $2D$ doméně je výkon karty *AMD* výrazně lepší, než čeho dosahuje *OpenCL* varianta na kartě *nvidia* a oproti *CUDA* verzi se liší přibližně o 60 %. Po překročení hranice 4096 prvků hrany matice však dojde k obratu a karta *nvidia* má opět vyšší výkon. Na druhou stranu v případě $3D$ domény je karta *AMD* konzistentně horší než obě implementace na kartě *nvidia*. Tyto výsledky odpovídají zjištěním z experimentů s *clFFT* v sekci 3.4.

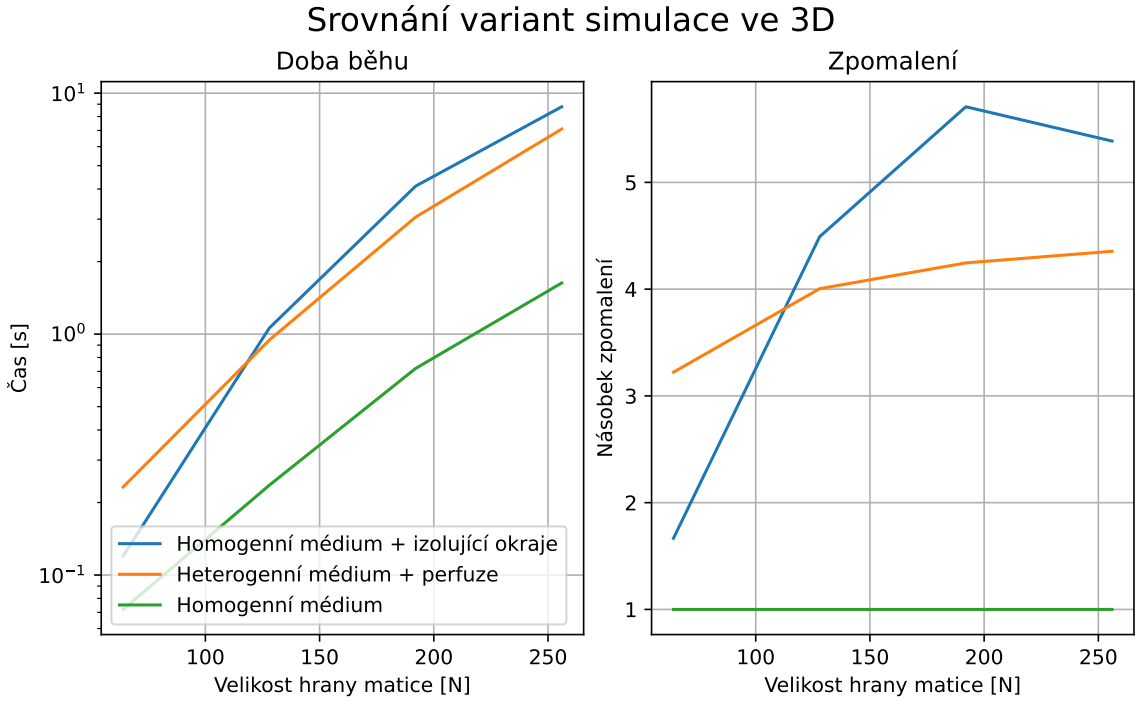
5.1.3 Karty *nvidia*

Python verze simulace byla otestována na několika kartách *nvidia*. Výsledky tohoto porovnání lze vidět v grafu 5.8. Výsledky měření odpovídají očekávanému výkonu. Řešená

[1] *Matlab* DTT: `https://www.mathworks.com/matlabcentral/fileexchange/75071-matlab-discrete-trigonometric-transform-library`

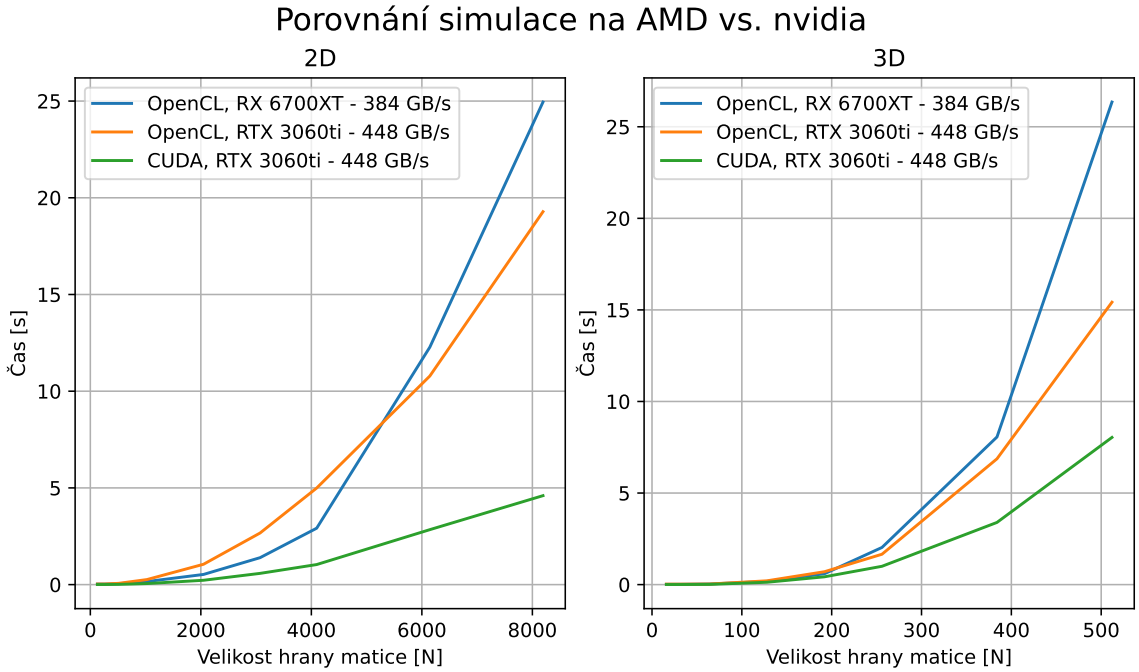


Obrázek 5.5: Srovnání *Matlab* implementace na *CPU AMD 3700X* a *Python* implementace na *RTX 3060ti* pro homogenní médium s vodivým okrajem domény a zdrojem tepla.



Obrázek 5.6: Srovnání různých variant simulace v *Pythonu* a jejich zpomalení oproti základní variantě.

úloha je primárně brzděná propustností paměti kvůli nízké aritmetické intenzitě, a proto je naměřený výkon přímo úměrný propustnosti paměti. To je dobře vidět třeba na srovnání karet *RTX 3060ti* a *RTX 3070*. Karta *RTX 3070* dosahuje téměř identického výkonu na této úloze, přestože má k dispozici více výpočetních jader. Jediná anomálie vzniká při po-



Obrázek 5.7: Doby běhu simulace na $2D$ a $3D$ doméně s využitím *OpenCL* na kartách *AMD* a *nvidia*. Pro referenci je zobrazena i doba běhu *CUDA* verze na kartě *nvidia*

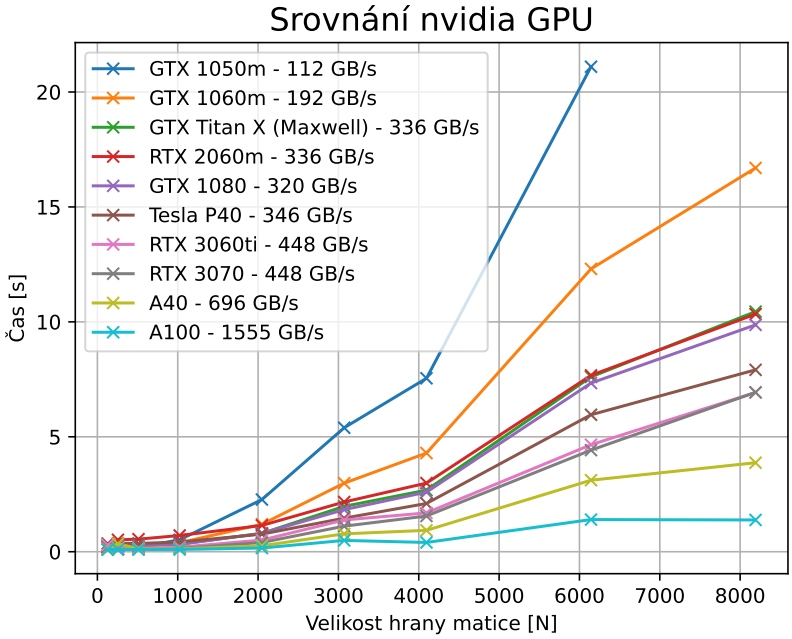
rovnání karet *GTX 1080* a *RTX 2060m*, kde *RTX 2060m* má nepatrně vyšší propustnost paměti, přesto je pomalejší. Rozdíl mezi těmito dvěma kartami se ale zdá na všech velikostech matice konstantní a neroste s velikostí matice, jako je tomu u ostatních srovnání. Je možné, že jde o chybu měření, např. vlivem operačního systému, *CPU* nebo verze *Pythonu*, které nejsou u všech měření v tomto grafu stejné. Karta *GTX 1050m* kvůli omezené paměti ($2GB$) nedokáže spustit simulaci na mřížce 8192^2. Nejvyššího výkonu dosahují dle očekávání karty *A40* a *A100*.

5.2 Paměťová náročnost

Tabulka 5.1 zobrazuje paměťovou náročnost jednotlivých implementací. Mezi knihovnami *CUDA*, *OpenMP* a *OpenCL* nejsou významné rozdíly. Je to způsobeno tím, že jde o knihovny kde je paměť spravována explicitně uživatelem. Implementace v jazyce *Python* je však výrazně horší. V této základní variantě simulace pozorujeme paměťové nároky vyšší přibližně o 60 %. Vzhledem k tomu, že grafická karta použitá k měření má pouze $8GB$ paměti, je vidět, že matice vetší než 512^3 představuje problém, který se bude jen prohlubovat při použití jiných modelů simulace (např. heterogenní médium), které jsou ve své podstatě ještě náročnější na paměť.

Matlab v tomto porovnání vykazuje nejvyšší paměťovou náročnost. Při výpočtu je použit datový typ *double*. Pokud započteme tento faktor, po použití typu *float* by byla paměťová náročnost *Matlabu* stále asi o 50 % vyšší než verze v *Pythonu* a o 150 % vyšší než nejlepší varianta (*OpenCL*).

Přestože tedy implementace v *Pythonu* není zcela optimální z hlediska potřebné paměti, stále přináší výrazné zlepšení oproti původnímu stavu.



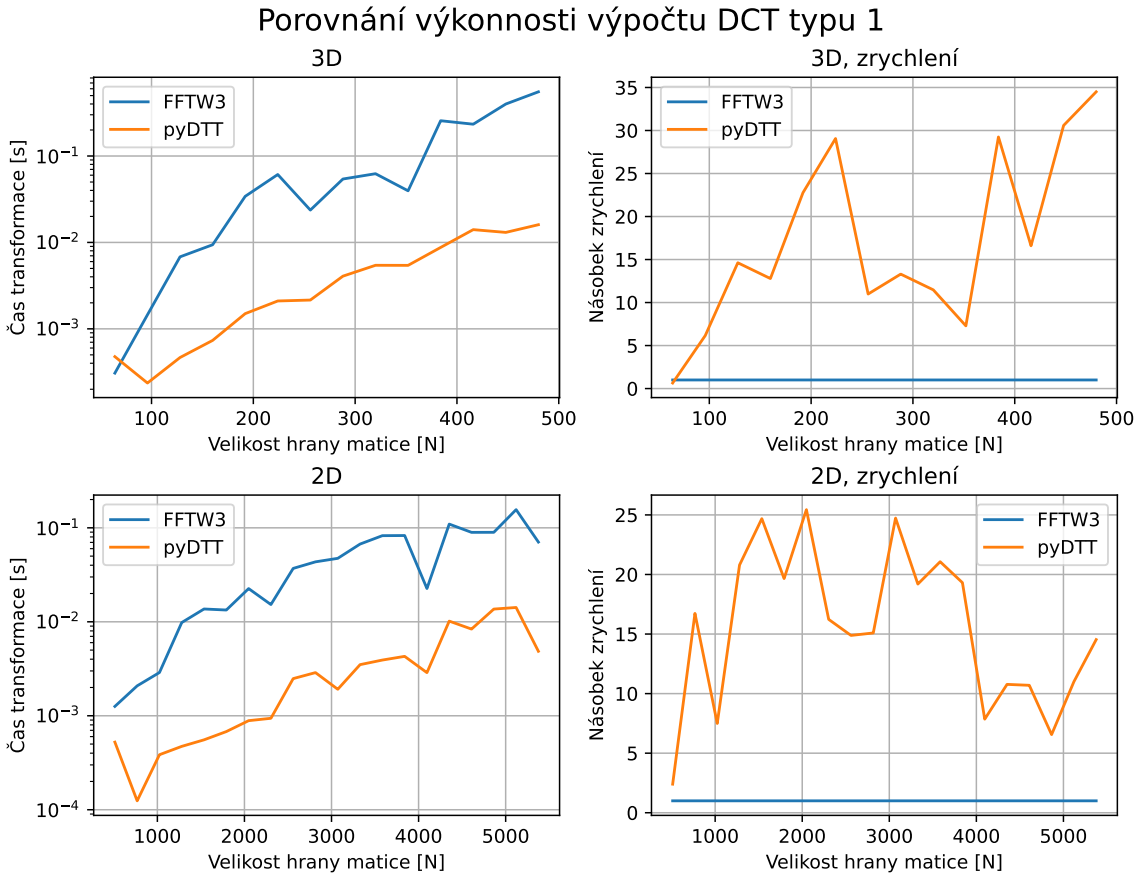
Obrázek 5.8: Doby běhu simulace na $2D$ homogenní doméně s využitím *Python* verze na několika kartách *nvidia*. U jednotlivých karet je také uvedena jejich stanovená propustnost paměti.

Tabulka 5.1: Paměť potřebná pro běh jednotlivých implementací pro simulaci s homogenním médiem, *k-Space* korekcí, bez perfuze a s velikostí mřížky $512 \times 512 \times 512$. *Matlab* využívá oproti ostatním datový typ *double* místo *float*. Tabulka zobrazuje paměť využitou na *GPU*, kromě hodnoty pro *Matlab*, kde je zobrazena paměť na *CPU*.

CUDA [MiB]	OpenMP [MiB]	OpenCL [MiB]	Python [MiB]	Matlab [MiB]
3735	3737	3727	6115	18636

5.2.1 Výkon diskrétních trigonometrických transformací

Na grafech 5.9 a 5.10 je zobrazen výkon diskrétních trigonometrických transformací v procesorové knihovně *FFTW 3* oproti mému řešení na grafické kartě. Na grafech lze vidět průměrně desetinásobné zrychlení ve všech variantách transformací, což je očekávatelné zrychlení oproti optimální implementaci na *CPU* a přibližně odpovídá i poměru propustnosti paměti mezi *CPU* a *GPU* ($47GiB/s$ vs. $448GiB/s$). Knihovna *FFTW 3* byla nastavena na využití všech 16-ti vláken a při vytváření plánu transformace byl použit mód *FFTW_MEASURE*, který provádí měření výkonnosti při tvorbě plánu a má značný pozitivní vliv na výkon oproti módu *FFTW_ESTIMATE*, za cenu delší inicializace plánu. Srovnání však ukazuje pouze dobu transformace samotné. Dosažené zrychlení u diskrétní sinové transformace je nižší než u cosinové transformace, protože oproti cosinové transformaci navíc probíhá doplnění vstupu o nuly, což není implementováno v tomtéž kernelu jako kopie vstupu (*DTT* modul nevyužívá *raw* kernely v *CuPy*).

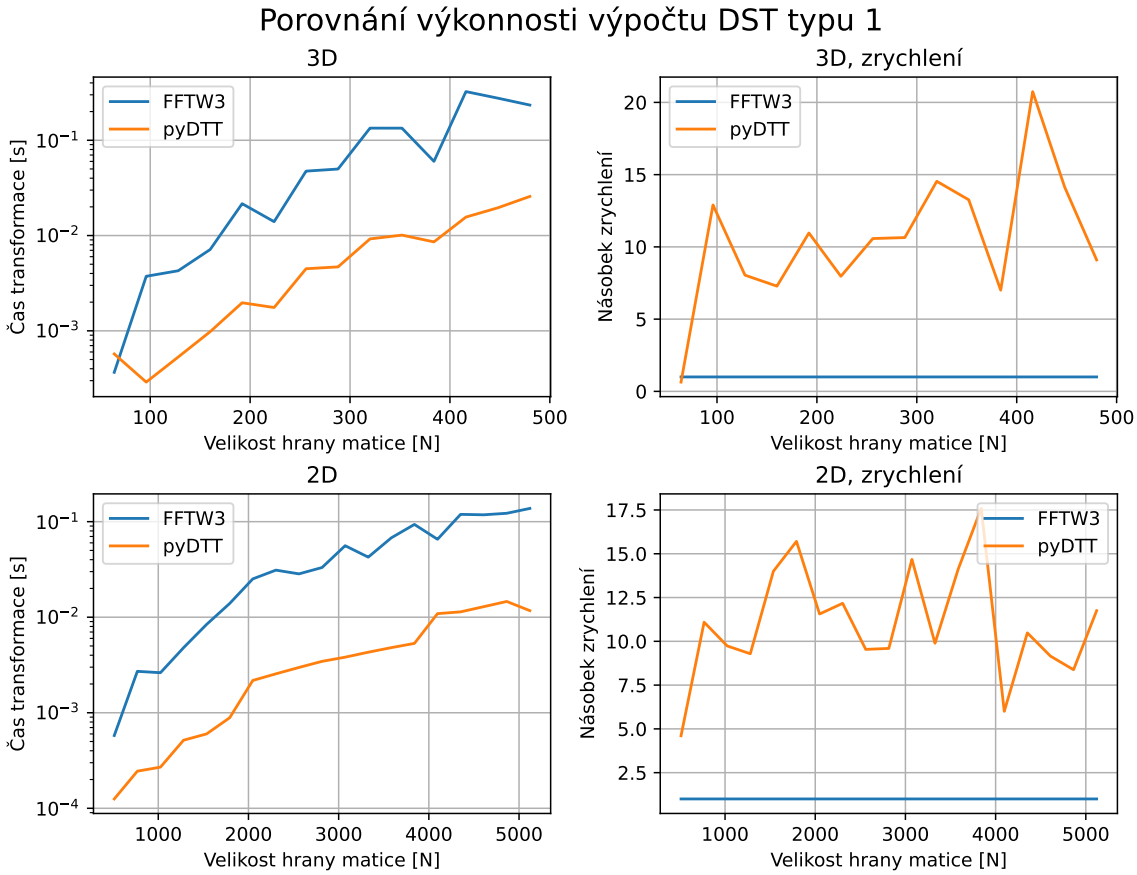


Obrázek 5.9: Doby běhu diskrétní cosinové transformace na $2D$ i $3D$ matici. Grafy porovnávají implementaci v knihovně *FFTW 3* na procesoru s mojí implementací na *GPU* (*pyDTT*). Na pravé straně je vykresleno zrychlení oproti *FFTW 3*. Procesorová transformace běžela na 16-ti vláknech procesoru *AMD 3700X* a *GPU* verze na kartě *RTX 3060ti*. Plán transformace *FFTW 3* byl vytvořen v módu *FFTW_MEASURE*.

5.3 Subjektivní srovnání použitých knihoven

V této sekci srovnávám získané zkušenosti z práce s jednotlivými knihovnami z několika úhlů pohledu. Shrnutí je vidět na tabulce 5.2.

Z pohledu časové náročnosti implementace přesvědčivě vede *CuPy*. Výhodou je skvělá podpora knihovních funkcí pokrývající knihovny *CUDA* ale i další funkce pro běžné operace jako transpozice, redukce apod. Kompatibilita s rozhraním *NumPy* umožňuje velmi jednoduchý převod výpočtu na *GPU* a není nutné se starat o nastavení knihoven, vytváření plánů atd. Významné zjednodušení implementace přináší i fakt, že kernely jsou kompilovány až za běhu v závislosti na typech a velikosti dodaných dat. To umožnilo bez dalšího úsilí pokrýt více variant vstupních parametrů a běhu simulace. *OpenMP* a *OpenACC* rovněž umožňuje jednoduchou tvorbu aplikací na *GPU*, zvlášť v případě, že již existuje *C++* implementace. Správa paměti a spouštění kernelů jsou u obou velmi jednoduché. Tyto knihovny nedosahují takové uživatelské přívětivosti jako *CuPy* např. z důvodů nutnosti volání některých externích knihoven, kde interoperabilita není vždy stoprocentní. Mírně negativně hodnotím uživatelskou přívětivost *CUDA*. Při implementaci hlavní simulace nebyla *CUDA*



Obrázek 5.10: Doby běhu diskrétní sinové transformace na $2D$ i $3D$ matici. Grafy porovnávají implementaci v knihovně *FFTW 3* na procesoru s mojí implementací na *GPU* (*pyDTT*). Na pravé straně je vykresleno zrychlení oproti *FFTW 3*. Procesorová transformace běžela na 16-ti vláknech procesoru *AMD 3700X* a *GPU* verze na kartě *RTX 3060ti*. Plán transformace *FFTW 3* byl vytvořen v módu *FFTW_MEASURE*.

žádný problém (pouze „Element-wise“ kernely), ale např. při tvorbě *DTT* modulu trvala implementace v *CUDA* násobně déle.

Z výkonového a paměťového hlediska byly dle očekávání *CUDA*, *OpenMP* a *OpenACC* nejlepší. *CuPy* poskytuje dobrý výkon, ale není tak paměťově efektivní. *OpenCL* dosahuje nejnižšího výkonu kvůli použití *clFFT*.

Z hlediska podpory nejlépe hodnotím *OpenCL*, protože je dostupné na *Windows* i *Linux* a funguje jak na grafických kartách *nvidia* tak *AMD*. *CuPy* má sice podporu pro *AMD* avšak jen experimentální. Podpora pro *AMD* nefunguje na *Windows* a na novějších kartách *AMD 6000*, které zatím nepodporují *ROCm*. *CUDA* a *Thrust* jsou technologie *nvidia* a jejich podpora je tak omezena pouze na tyto karty. U *OpenMP* jsem nenašel překladač schopný kompilovat *GPU* kód pro *Windows*.

Z pohledu stability nejlépe hodnotím *CUDA* a *OpenCL*. Jde o nejvíce zavedené a odladěné technologie a při jejich použití jsem nenarazil na žádné problémy. *OpenMP* má zatím problémy s podporou v překladačích a ne všechny funkce ve specifikaci jsou implementovány.

Tabulka 5.2: Subjektivní srovnání použitých knihoven. Tabulka hodnotí uživatelskou přívětivost (*UX*), Výkon, Paměťové nároky, Podporu hardware a software, stabilitu a jednoduchost nasazení na běžné platformy.

	UX	Výkon	Paměť	Podpora	Stabilita	Nasazení
CuPy	+++	+	−−	+	++	+
CUDA	−	+++	+++	+	+++	+++
Thrust	−	++	+++	+	+++	+++
OpenMP	++	+++	+++	−−	−	−
OpenACC	++	+++	+++	+	+	−
OpenCL	−−	−−	+++	+++	+++	+++

Nejjednodušší nasazení je s použitím *CUDA* a *OpenCL*. *CuPy* vyžaduje použití dalších nástrojů, ale je vcelku proveditelné. Nasazení *OpenMP* a *OpenACC* není problém na *Linuxu*, na *Windows* se mi však podařilo tyto verze simulace spustit pouze v nástroji *Docker*, což není ideální pro nasazení aplikace.

Stručně bych tyto technologie popsal takto:

- CuPy: jednoduchý vývoj, solidní výkon
- CUDA: náročný vývoj, skvělý výkon
- OpenCL: skvělá přenositelnost, náročný vývoj, slabý výkon
- OpenMP, OpenACC: jednoduchá akcelerace existujícího *C++* kódu, skvělý výkon.

5.3.1 Použití v dalších modulech *k-Wave*

Knihovna *CuPy* se ukázala jako skvělá jednoduchá knihovna pro snadné prototypování aplikace na *GPU* a s malým úsilím se lze přiblížit k výkonu na *CUDA*. Nedoporučil bych ji ale v případě úloh kritických na paměť nebo tam, kde je třeba zpracovávat data „out-of-core“, protože spotřeba paměti může být obtížně předvídatelná.

Knihovny *OpenMP* a *OpenACC* dle mého názoru v této úloze mnoho nepřinesly. Všechny kernely prakticky pracují stylem jeden prvek, jedno vlákno a nevyužívají sdílenou paměť. Rozdíl v náročnosti implementace oproti *CUDA* není významný. Pokud uvážím zatím nedokonalou podporu v překladačích, tak bych je na podobné úlohy spíše nedoporučil. *CUDA* implementace bude rychlejší, přenositelnější a stabilnější. U úloh kde již existuje *C++* varianta nebo u úloh využívajících sdílenou paměť způsobem, který není pokryt nějakou knihovnou, má *OpenMP* a *OpenACC* největší přínos.

Kapitola 6

Závěr

V rámci mé diplomové práce vzniklo několik implementací simulace šíření tepla na procesoru i na grafické kartě, které jsou řádově rychlejší než původní implementace v jazyce *Matlab*. Oproti původní implementaci na *CPU* bylo dosaženo zrychlení více než 100× při použití knihovny *CuPy* v *Pythonu* na grafické kartě, a to na všech možných variantách simulace. Dalším dobrým výsledkem je implementace v *OpenMP* na procesoru, která dosahuje okolo desetinásobného zrychlení oproti původní verzi a je rovněž rychlejší než *Matlab* s automatickou paralelizací na *GPU*.

Implementace v *CUDA*, *OpenACC*, *Thrust*, *OpenMP* (jediná implementace pro *CPU* i *GPU*) a *OpenCL* podporují základní simulaci s homogenním médiem a *k-Space* korekcí. Zmíněná implementace v *Pythonu* obsahuje veškerou funkcionalitu původní implementace ale i nějakou navíc (možnost specifikovat více zdrojů, různé způsoby zadávání zdrojů a senzorů).

Vytvořené implementace byly testovány na sadě vlastních regresních testů pokrývajících všechny kombinace funkcionality simulátoru, a také na již existující sadě testů dostupných v balíku *k-Wave* a všechny dosahují maximální relativní chyby pod 10^{-5}, což je vzhledem k přesnosti datového typu *float* zanedbatelná chyba.

Byly prozkoumány možnosti a funkce knihoven *CuPy* a *OpenMP* (ale i *CUDA*, *OpenCL*, *OpenACC*, *Thrust*, *ROCm* nebo *Julia*) pro využití v *GPGPU* a dalších modulech balíku *k-Wave*.

V rámci provedeného porovnání se ukazuje, že při základní verzi simulace šíření tepla (homogenní médium bez perfuze), má použití zvolených vysokoúrovňových knihoven (*CuPy* a *OpenMP*) negativní dopad na výkonnost zhruba o 60 % oproti nízkoúrovňové implementaci v *CUDA*. Dále při použití jazyka *Python* s *CuPy* se ukazuje neefektivní využití grafické paměti, které oproti implementaci v *CUDA* potřebuje okolo 60 % paměti navíc, což limituje použitelnost u úloh s rozsáhlými doménami ($> 512^3$). Tato neefektivita je způsobena dynamickou alokací paměti *CuPy* a je možné se jí vyhnout explicitní alokací, ovšem za cenu horší použitelnosti a větší složitosti implementace. I tak ale představuje implementace v *Pythonu* zlepšení v oblasti paměťové náročnosti oproti původnímu řešení.

Při implementaci řešené úlohy v několika jazycích bylo zjištěno, jaká je náročnost implementace v každém z nich, přičemž *Python* a *CuPy* se ukázaly jako nejjednodušší varianta. Přepis z jazyka *Matlab* do rozhraní *NumPy/CuPy* je až na několik detailů přímočarý, neli triviální. Vytvářením jednoduchých uživatelských kernelů je možné snadno dosáhnout rychlosti blížící se *CUDA* verzi. Pro dosažení maximálního výkonu je však evidentně nutné provést optimalizace, které by způsobily, že výsledný kód bude velice podobný *CUDA* verzi, což by nepřineslo takové výhody z hlediska jednoduchosti implementace.

Jako součástí řešení vznikl také modul v jazyce *Python* (opět s využitím *CuPy*) určený pro výpočty diskrétních trigonometrických transformací na *GPU*. Tento modul dosahuje oproti vícevláknové implementaci ve *FFTW 3* na procesoru zrychlení přibližně o jeden řád, což umožnilo celkové simulaci využívající tento modul dosáhnout dobrého zrychlení i v případech s jinými než periodickými okrajovými podmínkami.

Vytvořený program v *Pythonu* byl zabalen pro *Windows* ve formě instalátoru a pro *Linux* ve formě jednoduchého archivu a balíku `.deb`. V těchto balících jsou dodány všechny závislosti, a uživatel proto nemusí pro použití nijak nastavovat své prostředí nebo instalovat další závislosti. Rovněž byly vytvořeny metody, které umožňují přímé použití simulátoru z původní implementace v *Matlabu*. Použití je tedy téměř identické s *Matlab* verzí. Vytvořený program v *C++* je dodáván s konfiguračním souborem pro *Docker*, což umožňuje jednoduchou replikaci vývojového prostředí a rovněž není nutné instalovat další závislosti. I tato varianta běží jak na *Windows*, tak na *Linuxu*.

6.1 Možnosti pokračování práce

V práci by bylo možné pokračovat zejména na rozšíření funkcionality *C++* verze, především o diskrétní trigonometrické transformace v *CUDA* a přidání více možností simulace samotné do některé z verzí, které v porovnání základní simulace vychází nejlépe (*CUDA*, *OpenMP* nebo *OpenACC*). Podobné rozšíření *OpenCL* verze by mohlo být rovněž přínosné i přes nízký výkon dosažený s touto knihovnou. Důvodem k tomu je asi nejlepší přenositelnost kódu knihovny *OpenCL*, která je podporována všemi platformami i výrobci grafických karet.

Další možností pokračování práce je rozsáhlejší výzkum podpory *GPGPU* knihoven na kartách *AMD*. Primárně podpora *CuPy* na kartách *AMD* zůstává otázkou. Přínosné by ale také mohlo být zjistit možnosti překladačů z *ROCm* pro použití s *OpenMP* a *OpenACC*.

Literatura

[1] Board, O. A. R. *OpenMP Application Programming Interface, Version 5.2*. 2021. Dostupné z: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf.

[2] Group, T. K. S. W. *SYCL™ 2020 Specification (revision 4)*. 2021. Dostupné z: https://www.khronos.org/registry/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf.

[3] Jordan, H. Bio-Heat Models Revisited: Concepts, Derivations, Nondimensalization and Fractionalization approaches. *Frontiers in Physics*. 2019. DOI: 10.3389/fphy.2019.00189.

[4] Kassem, A., Hamad, M. a Haidamous, E. Image compression on FPGA using DCT. In:. Srpen 2009, s. 320 – 323. DOI: 10.1109/ACTEA.2009.5227881.

[5] Larkin, J. OpenMP on GPUs, First Experiences and Best Practices. *GTC2018 S8344*. 2018.

[6] Luo, Y. *Leveraging implicit cuda streams and asynchronous openMP offload features in LLVM*. Dostupné z: https://www.openmp.org/wp-content/uploads/2021_03_26_Leveraging_Implicit_CUDA_Streams_and_Asynchronous_OpenMP_offload_Features_in_LLV

[7] Makhoul, J. A fast cosine transform in one and two dimensions. *IEEE Transactions on Acoustics, Speech, and Signal Processing*. 1980, sv. 28, č. 1, s. 27–34. DOI: 10.1109/TASSP.1980.1163351.

[8] Mingming Ren, G. W. X. L. Discrete Sine and Cosine Transform and Helmholtz Equation Solver on GPU. *2020 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking*. 2020, s. 57–66. DOI: 10.1109/ISPA-BDCloud-SocialCom-SustainCom51426.2020.00034. Dostupné z: https://ieeexplore.ieee.org/document/9443873.

[9] OpenACC Standard.org. *The OpenACC Application Programming Interface, Version 3.2*. 2021. Dostupné z: https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.2-final.pdf.

[10] 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.

[11] Wikipedia contributors. *Discrete sine transform — Wikipedia, The Free Encyclopedia* [https://en.wikipedia.org/w/index.php?title=Discrete_sine_transform&oldid=1056063222]. 2021. [Online; accessed 11-January-2022].

[12] Wikipedia contributors. *Discrete cosine transform — Wikipedia, The Free Encyclopedia* [https://en.wikipedia.org/w/index.php?title=Discrete_cosine_transform&oldid=1064931425]. 2022. [Online; accessed 11-January-2022].

[13] Wikipedie. *CUDA — Wikipedie: Otevřená encyklopedie*. 2021. [Online; navštíveno 20. 12. 2021]. Dostupné z: https://cs.wikipedia.org/w/index.php?title=CUDA&oldid=19994145.

Příloha A

Obsah přiloženého paměťového média

- `/`
  - `Release/` — Spustitelné binární soubory
    - `k-Wave_diffusion_solver_installer_v1_0.exe` — Instalátor pro *Windows*
    - `kwave_diffusion.deb` — Balík pro *Linux*
    - `kwave_diffusion.7z` — Balík pro *Linux*, manuální instalace
    - `kWaveDiffusion.m` — Upravený skript pro spuštění externího solveru
  - `Sources/`
    - `omp_benchmark/` — Sada testovacích příkladů *OpenMP 5.0*
    - `python/`
      - `packaging/` — Soubory potřebné k zabalení aplikace
        - `kwave_diffusion/` — Před-připravená struktura `.deb` balíku
        - `installer.iss` — Konfigurace pro *Inno setup*
      - `test_data/` — Testovací data
      - `kwave_diffusion.py` — Hlavní skript *Python* verze
      - `test.py` — Testovací skript
      - `dtt.py` — Modul pro výpočet trigonometrických transformací
      - `solverExport.patch` — Patch soubor pro export z *Matlabu*
      - `requirements.txt` — Požadavky pro instalaci *Python* modulů
      - `kwave_diffusion.spec` — Konfigurace pro *PyInstaller*
      - `...`
    - `main.cpp` — Vstupní bod *C++* verze
    - `CMakeLists.txt` — Soubor pro překlad *C++* verze
    - `...`
  - `Thesis/` — Zdrojové soubory tohoto textu
  - `Documentation/` — Dokumentace
    - `index.html`
    - `...`
  - `License.md` — Licence
  - `Dockerfile` — Soubor pro vytvoření obrazu pro *Docker*
  - `Doxyfile` — Konfigurace pro generování dokumentace

Příloha B

Manuál

B.1 Instalace na Windows

Pro instalaci na *Windows* je potřeba nainstalovat solver pomocí dodaného instalátoru: `/Release/k-Wave_diffusion_solver_installer_v1_0.exe`. Tento instalátor uloží do zadaného umístění soubor `kwave_diffusion.exe`.

Odinstalace probíhá stejně jako u každé jiné aplikace na *Windows* přes ovládací panel v sekci *Programs and Features*, nebo pomocí souboru `unins000.exe` v adresáři s instalací.

B.2 Instalace na *Linux* s *apt*

Pro instalaci na *Linux* je potřeba nainstalovat balík `kwave_diffusion.deb` pomocí příkazu:

```
sudo apt install ./kwave_diffusion.deb
```

Balík nainstaluje spustitelné soubory do umístění `/opt/kwave_diffusion/`. Program je pak možné spustit odkudkoliv voláním `kwave_diffusion` na příkazové řádce.

Balík lze odinstalovat voláním:

```
sudo apt remove kwave-diffusion
```

B.3 Instalace na ostatní distribuce *Linux*

Pro ostatní *Linuxové* distribuce je dodáván také balík `kwave_diffusion.7z` určený pro manuální instalaci. Balík je nutné ručně extrahovat do zvoleného umístění pomocí příkazu:

```
7z x kwave_diffusion.7z
```

B.4 Spuštění z *Matlabu*

Pro spuštění programu z *Matlabu* je potřeba použít upravený soubor původního solveru: `kWaveDiffusion.m`, který je dostupný ve složce `/Release/` přiloženého disku. Tento skript poskytuje navíc metody pro export dat z *Matlabu* a spuštění externího solveru:

```
    % take time steps
    kdiff.runExternalSolver("export.h5", Nt, dt, medium, "kwave_diffusion");

    % kdiff.takeTimeStep(Nt, dt);

```

Jako poslední parametr je třeba předat cestu k instalovanému solveru (v závislosti na použité metodě instalace).

B.5 Překlad *C++* verze

Preferovaný nástroj pro překlad *C++* verze je *Docker* jak na *Windows*, tak na *Linuxu*. Pro vytvoření obrazu pro *Docker* stačí spustit následující příkaz v kořenové složce projektu:

```
# Build image
nvidia-docker build --no-cache --squash -t solver:v1 .

# Run development and mount current working directory to /files inside the container
nvidia-docker run -it -v `pwd`:/files --gpus all solver:v1
```

Vytvoření obrazu může trvat přibližně půl hodiny.

B.6 Spuštění *Python* verze

Pro spuštění *Python* verze je doporučeno použít virtuální prostředí. Je nutné spustit následující příkazy ve složce `/Sources/python`:

```
# Create virtual environment
python3 -m venv .venv
source .venv/bin/activate

# Install dependencies
pip3 install -r requirements.txt

# Run solver and tests
./kwave_diffusion.py -i test_data/2D_homogeneous/input_4096.h5
./test.py test_data ./kwave_diffusion.py
```

Příloha C

Specifikace vstupního formátu

Program očekává ve vstupním `*.h5` souboru existenci těchto skupin a datasetů:

- **`/`**
  - **`simulation_flags/`**
    - `perfusion_flag` ... Povolí perfuzi $\{0, 1\}$
    - `kspace_correction_flag` ... Zapne *k-Space* korekci $\{0, 1\}$
    - `boundary_condition_flag` ... Typ okrajové podmínky $\{0, 1, 2\}$
  - **`grid_properties/`**
    - `Nx` ... rozměr domény v ose X $[N]$
    - `Ny` ... rozměr domény v ose Y $[N]$
    - `Nz` ... rozměr domény v ose Z ($= 1$ ve $2D$) $[N]$
    - `dx` ... vzdálenost bodů v ose X $[m]$
    - `dy` ... vzdálenost bodů v ose Y $[m]$
    - `dz` ... vzdálenost bodů v ose Z $[m]$
    - `Nt` ... Počet kroků simulace $[N]$
    - `dt` ... Délka jednoho kroku $[s]$
  - **`medium_properties/`**
    - `heterogeneous_flag` ... Jde o heterogenní médium $\{0, 1\}$
    - `diffusion_coeff_flag` ... Zadán difuzní koeficient $\{0, 1\}$
    - `perfusion_coeff_flag` ... Zadán perfuzní koeficient $\{0, 1\}$
    - `diffusion_coeff` ... Nepodporováno
    - `density` ... Hustota tkáně $[kg/m^3]$
    - `thermal_conductivity` ... Tepelná vodivost $[W/(m \cdot K)]$
    - `specific_heat` ... Měrná tepelná kapacita $[J/(kg \cdot K)]$
    - `perfusion_coeff` ... Perfuzní koeficient (pokud je povolen) $[s^{-1}]$
    - `blood_density` ... Hustota krve $[kg/m^3]$
    - `blood_perfusion_rate` ... Rychlost perfuze $[s^{-1}]$
    - `blood_specific_heat` ... Měrná tepelná kapacita krve $[J/(kg \cdot K)]$
    - `diffusion_coeff_ref` ... Referenční koeficient difuze $[m^2/s]$
    - `perfusion_coeff_ref` ... Referenční koeficient perfuze $[s^{-1}]$
    - `blood_ambient_temperature` ... Ambientní teplota krve $[°C]$
  - **`sources/`**
    - `number_sources` ... Počet zdrojů tepla $[N]$
    - `1/`
      - `source_mask_type` ... Typ masky zdroje $\{0, 1\}$
      - `source_mask` ... Maska zdroje (lineární indexy/rohy kvádru)

- `volume_heat_deposition` ..... Příspěvek zdroje tepla $[W/m^3]$
- `time_start` ..... Krok spuštění zdroje $[N]$
- `time_stop` ..... Krok vypnutí zdroje $[N]$

`2/`
- `...`

`.../`

`sensors/`
- `A_sensor_flag` ..... Ukládat průběžné teploty a poškození $\{0, 1\}$
- `A_max_sensor_flag` ..... Ukládat průběžné maximum $\{0, 1\}$
- `A_max_all_sensor_flag` ..... Uložit maxima teplot $\{0, 1\}$
- `A_final_all_sensor_flag` ..... Uložit finální teploty a poškození $\{0, 1\}$
- `number_sensors` ..... Počet senzorů $[N]$
- `time_downsample_factor` ..... Faktor podvzorkování $[N]$
- `record_start_index` ..... Začátek ukládání dat $[N]$

`1/`
- `sensor_mask_type` ..... Typ masky senzoru $\{0, 1\}$
- `sensor_mask` ..... Maska senzoru (lineární indexy/rohy kvádru)

`2/`
- `...`

`.../`