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 POMOCÍ KNIHOVNY OPENACC

SIMULATION OF HEAT DIFFUSION IN THE BRAIN USING THE OPENACC LIBRARY

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **JOSEF OŠKERA**
AUTHOR

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

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Oškera Josef**
Program: Informační technologie
Název: **Simulace šíření tepla v mozku pomocí knihovny OpenACC**
Simulation of Heat Diffusion in the Brain Using the OpenACC Library
Kategorie: Paralelní a distribuované výpočty

Zadání:

1. Seznamte se s knihovnou OpenACC pro paralelní výpočty na 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 OpenACC.
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 OpenACC pro využití v dalších modulech balíku k-Wave.
9. Zhodnoťte dosažené výsledky

Literatura:

- Dle pokynů vedoucího.

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

Vedoucí práce: **Jaroš Jiří, doc. Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 30. října 2020

Abstrakt

Cílem této práce je přepsat implementaci šíření tepla v mozkové tkáni naspanou v jazyce Matlab (dostupnou v balíku k-Wave) do jazyka C/C++, akcelerovat ji na GPU za pomoci knihovny OpenACC a CUDA, a následně tyto knihovny porovnat ve výkonnosti a náročnosti implementace. V řešení je popsáno jak programovat grafickou kartu, a jak tyto znalosti aplikovat. Vytvořený program je schopen simulovat šíření tepla na CPU i GPU.

Abstract

The aim of this work is to rewrite the implementation of heat transfer in brain written in programming language Matlab (available in the k-Wave package) into C / C ++, accelerate it on GPU using library OpenACC and CUDA, and then compare these libraries in performance and complexity of implementation. The solution describes how to program a graphics card and how to apply this knowledge. The created program is able to simulate heat dissipation on CPU and GPU.

Klíčová slova

C/C++, k-Wave, k-Wave-Diffusion, OpenACC, CUDA

Keywords

C/C++, k-Wave, k-Wave-Diffusion, OpenACC, CUDA

Citace

OŠKERA, Josef. *Simulace šíření tepla v mozku pomocí knihovny OpenACC*. Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Doc. Ing. Jiří Jaroš, Ph.D.

Simulace šíření tepla v mozku pomocí knihovny OpenACC

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.

.....
Josef Oškera
19. května 2021

Poděkování

Tímto bych chtěl poděkovat panu Doc. Ing. Jiřímu Jarošovi Ph.D. za jeho velkou trpělivost o ochotu vždy pomoci. Tato práce byla podpořena Ministerstvem školství, mládeže a tělovýchovy České republiky prostřednictvím e-INFRA CZ (ID:90140).

Obsah

Kapitola 1

Úvod

Cílem této bakalářské práce je přepsat již existující aplikaci pro simulaci šíření tepla v mozkové tkáni. Aplikace je napsána v jazyce Matlab, je součástí balíku k-Wave a slouží pro medicínské účely, kdy se pomocí ultrazvuku vypalují tkáně v mozku například pro zatavení porušené cévy, nebo vypálení nádoru a tato aplikace simuluje šíření tepla a ničení tkání. Důvod proč vůbec přepisovat existující program je ten že Matlab je velmi náročný na výpočetní prostředky, jako paměť a procesorový čas. Přepsáním do jazyka C++ získáme nejen paměťovou úsporu, ale i rychlejší výpočet. Jazyk C++ je na rozdíl od Matlabu kompilovaný, díky čemuž můžeme využít pokročilé optimalizace pro konkrétní procesor, jako například použití SIMD instrukcí. Nejdůležitější vlastností jazyka C++ je ovšem možnost použití knihoven pro programování na GPU, jako například OpenACC, CUDA, nebo OpenCL.

Proč tedy je dobré použít OpenACC, když tu máme léty prověřenou knihovnu CUDA nebo OpenCL? Se stálým zvyšováním výkonu počítačů se snižuje cena procesorového času a naopak se zvyšuje cena času programátora, proto vznikají technologie jako OpenACC, které mají za cíl urychlit vývoj programů i za cenu ztráty nějakého procenta výkonu. Konkrétně OpenACC má být náhradou za OpenCL nebo CUDA a slouží k paralelnímu programování, kdy programátor označí kód který má být paralelní a o zbytek se postará kompilátor - na GPU (ale i na CPU) podobně jako OpenMP.

V následují části kapitoly se budu věnovat obecným rozdílům mezi programováním pro CPU a GPU.

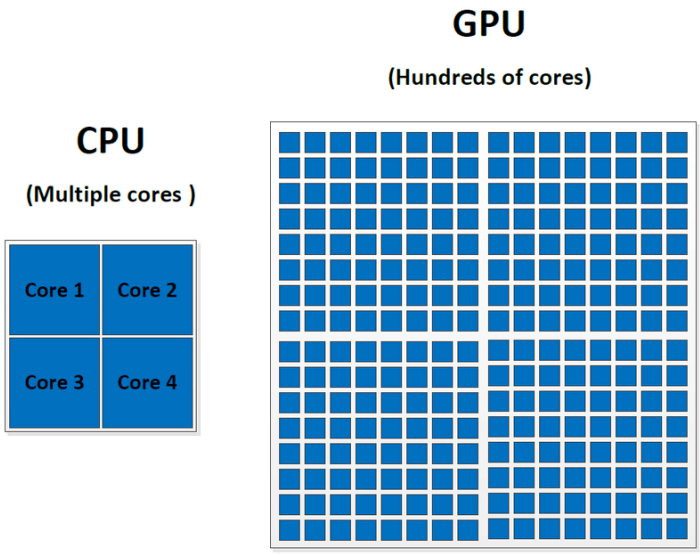
1.1 Paralelizace

Paralelizace je velmi dobrý způsob jak zrychlit běh programu, protože většina dnešních procesorů je vícejádrová. Ovšem ne všechen kód je paralelizovatelný, nelze moc dobře paralelizovat kód který pracuje nad stejnými daty a který je jakkoliv synchronizovaný (platí když se chceme vyhnout synchronizaci pomocí mutexů, nebo semaforů, které většinou v programování na GPU ani nelze použít[1]). Ideální úlohou pro paralelizaci je nějaký maticový nebo vektorový výpočet (matice je několik vektorů v paměti za sebou), tvořený cyklem, který postupně nad prvky vektoru provádí nějaké operace. Čím vícekrát se tento cyklus opakuje a jeho sekvenční část je kratší, tím je výhodnější použít pro výpočet GPU. V nejlepším případě se čas výpočtu na jednom jádře dělí počtem použitých jader, ale většinou to bude horší, protože čistý výkon jádra se sice dá škálovat, ale například propustnost pamětí již tak dobře ne.

[1] CUDA sice nějaké možnosti synchronizace má, ale OpenACC již ne

Například jeden ze strojů, na kterém bude probíhat testování obsahuje 2x Intel(R) Xeon(R) E5-2620 v3, kde každý má 6 jader a 1 vlákno na jádro[2], takže teoretické urychlení výpočtu je až 12x. Tento stroj (dále náš stroj) obsahuje také NVIDIA GTX 1080 8GB, tato karta má 2560 CUDA jader, hrubé porovnání je zobrazeno na obrázku 1.1, která jsou sice mnohem primitivnější a pracují na nižší frekvenci než CPU jádra, ale je jich 213-krát více než v našem procesoru a navíc jeden stroj může obsahovat několik těchto karet (náš má konkrétně 4). Intel(R) Xeon(R) E5-2620 v3 má teoretický výkon FP32 230 GFLOPs, NVIDIA GTX 1080 má teoretický výkon FP32 8.2 TFLOPs, takže při správné úloze grafická karta jasně vítězí.

Další nevýhodou pro CPU je výrazně pomalejší paměťový systém než u GPU. E5-2620 v3 má teoretickou paměťovou propustnost při použití 2 CPU (a všech 2x4 kanálů s 2133MHz DDR3 [3]) 136,5 GB/s. NVIDIA GTX 1080 (8GB GDDR5) má paměťovou propustnost 320.26 GB/s a například AMD Radeon VII (16GB HBM2) má paměťovou propustnost 1024 GB/s, takže rozdíl může být vysoký.[4] U CPU kvůli tomuto omezení může dojít k rapidnímu úpadku výkonu. CPU má ale možnost disponovat vetším množstvím paměti než grafická karta. Pokud se ovšem data nevejdou na kartu, a musely by se zpracovávat na více částí, tak dojde ještě k většímu zpomalení, protože teoretická rychlost PCI-E 3.0 x16 činí 16GB/s a u PCI-E 4.0 x16 32GB/s. [5]



Obrázek 1.1: Názorné porovnání CPU a GPU[12]

1.2 Programování GPU

Grafický procesor na rozdíl od klasického CPU, které běžně má jednotky až desítky jader, jich obsahuje stovky až tisíce. Aby byly zachovány rozměry (tím pádem i vyrobitelnost) grafického čipu, tak jsou tyto jádra (značí se CUDA jádra v případě karet NVIDIA, nebo Stream Processor v případě AMD) velmi jednoduchá (obsahují hlavně moduly pro práci a

[2] nejspíš vypnutý HyperThreading
[3] http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20E5-2620%20v3.html
[4] https://videocardz.com
[5] https://cs.wikipedia.org/wiki/PCI-Express#PCI-Express_3.0

celými čísly a čísly s plovoucí čárkou) a jsou spřažená do skupin (nazývaných Warp) a tyto spřažená jádra mají společný Program Counter a L1 instrukční cache a nachází se v SM procesorech (Streaming Multiprocesor) (jak můžete vidět na obrázku 1.2). Tudíž mohou vykonávat jen jednu stejnou instrukci. CUDA jádra jsou pouze jednovláknová. Jednodušší přirovnání tedy je že jádro klasického procesoru dokáže provádět jednu instrukci nad jedním číslem a SM procesor dokáže provádět jednu instrukci nad polem 32 čísel (SM procesory mohou obsahovat i více než 32 jader). Datová pole by tedy měla mít počet prvků dělitelný 32.

Další důležitý rozdíl v programování CPU a GPU je ten, že GPU má vlastní paměť a je nutné data překopírovávat mezi pamětí GPU a pamětí hostujícího počítače podle toho kde se aktuálně pracuje s daty. V ideálním případě celý výpočet probíhá na GPU a data se kopírují jen na začátku výpočtu a pak na jeho konci.

Důležitou součástí programování na GPU je určení si datového typu ve kterém budou probíhat výpočty. Pokud zásadně nepotřebujeme přesnost tak vždy volíme datový typ float32 nebo int32, protože pokud použijeme float64 (double), tak buď bude výpočet probíhat ve více cyklech nebo ve speciálních modulech, kterých bude pravděpodobně mnohem méně než modulů pro 32-bit čísla a výsledkem by byl i mnohonásobně pomalejší výpočet.

Také důležitou informací je, že je důležité se co nejvíce vyhýbat dělení, protože na rozdíl od sčítání a násobení, stojí dělení stovky cyklů.

Důležité je také vědět že CUDA jádra mezi sebou mohou komunikovat, ale jen v rámci SM procesoru. Každý SM procesor obsahuje sdílenou paměť, do které mají přístup všechny CUDA jádra co patří do příslušného SM procesoru. Synchronizace CUDA jader napříč SM procesory je nemožná, protože SM procesory mají sice společný přístup do globální paměti, ale není zajištěna koherence vyrovnávacích pamětí.

1.3 Historie Grafických karet

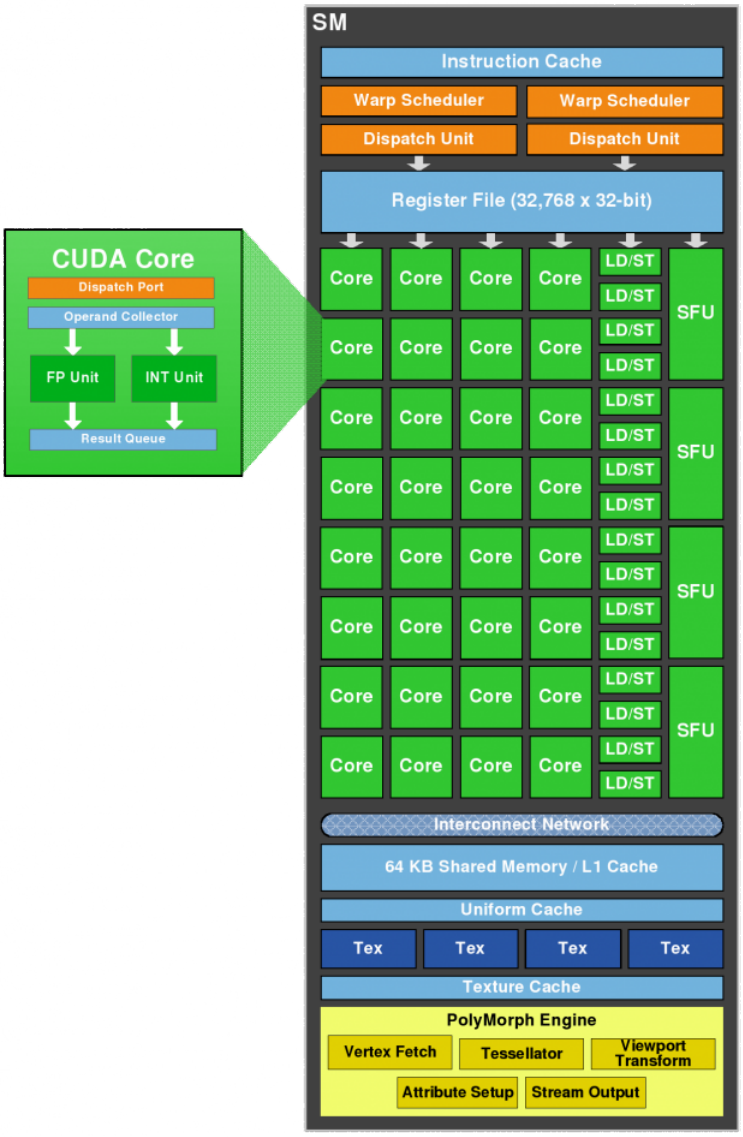
1.4 Historie výpočtů na grafických kartách

1.5 SIMD vs. SIMT

SIMD (Single Instruction Multiple Data) je počítačová architektura, kde se pomocí jedné instrukce zpracuje více dat najednou. Jsou to tedy vektorové operace pracující s celými poli dat.

SIMT (Single Instruction Multiple Threads) má stejný cíl jako SIMD (tzn. zpracovat co nejvíce dat během jedné instrukce) ale jde na to jinou metodou. Místo ho aby jedna instrukce zpracovala pole dat, máme pole jednoduchých jader zpracovávají jednu stejnou instrukci nad daty, ale s rozdílným offsetem.

Rozdíl mezi těmito dvěma metodami se pokusím znázornit na příkladu následující kapitoly patřící k obrázku 2.1.



Obrázek 1.2: SM procesor [13]

Kapitola 2

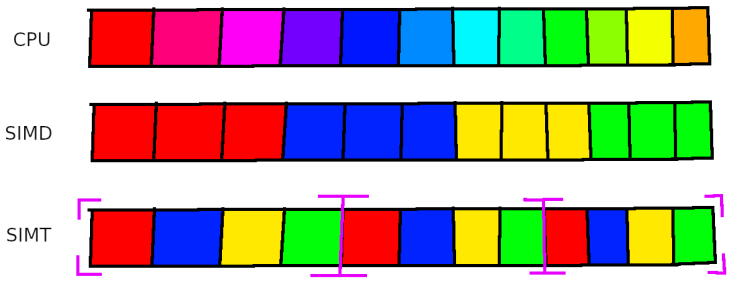
CUDA

CUDA je nejstarší zveřejněnou technologií pro paralelní výpočty na GPU. Využívá vlastní upravené syntaxe jazyka C/C++ a má definované vlastní datové typy, klíčová slova, struktury, přidává nové definice a knihovní funkce.

Použití CUDA je proto odlišné od OpenACC. Programování v CUDA je mnohem blíže hardwarovému složení procesoru grafické karty, a proto styl programování je velmi odlišný.

Budeme uvažovat příklad, kdy máme projít pole a přičíst ke každému prvku konstantu. Na procesoru bychom udělali cyklus, který by prvek po prvku proházel pole a přičítal k němu konstantu. Pokud máme 4 jádra, tak pole můžeme rozdělit na 4 části a každé jádro projde pole ve svém vlastním cyklu nebo v lepším případě postupně zpracovávat části pole pomocí SIMD instrukcí. Na GPU ale máme takových jader třeba 2560. Pokusím se to přiblížit na následujícím modelovém příkladu. Uvažujme pole o 12-ti prvcích, 1-jádrový procesor, 1-jádrový procesor podporující tří-prvkové instrukce SIMD a Procesor architektury SIMT obsahující jeden SM procesor s 4 jádry. Na obrázku2.1 lze vidět v prvním případě 1 jádro jak prochází pole. Každá barva značí jinou iteraci. V případě druhém vidíme jak prochází pole SIMD jádro. Každá barva také značí iteraci cyklu. V případě posledním vidíme SIMT procesor. Každá barva značí jedno jádro. Fialové čáry oddělují iterace cyklu, nebo jednotlivé SM procesory (threadbloky).

Procházení pole funguje tedy tak, že se pole rozdělí na bloky a každý blok se spustí právě na jednom SM procesoru. Blok se dělí na vlákna. Maximální počet vláken s bloku se může lišit u každé grafické karty, ale běžně dosahuje velikosti 1024*1024*64 vláken. Každé vlákno vykonává kernel a má k dispozici strukturu s vlastní pozicí v rámci bloku, index bloku a velikost bloku, z těchto 3 informací si každé vlákno vypočítá index do zpracovávaného



Obrázek 2.1: SIMD vs SIMT

pole. Každých 32 vláken (warp) synchronizovaně vykonávají instrukce kernelu. Názorná implementace jednoduchého kernelu ve k vidění v kapitole 7.

Nejdůležitější klíčové slovo co přidává CUDA je ___global___ kterým se označuje funkce, která bude vykonávána na GPU.

Kapitola 3

OpenACC

Tato kapitola se věnuje tomu co to je OpenACC, jak se používá, jaké jsou výhody a na závěr jakých lze dosáhnout výkonů ve velmi jednoduchém programu.

„OpenACC je uživatelsky řízený paralelní programovací model založený na direktivách. Je určen pro vědce a inženýry, kteří mají zájem přenést své kódy na širokou škálu heterogenních hardwarových platforem a architektur HPC s výrazně menším úsilím, než je vyžadováno u modelu na nízké úrovni. OpenACC podporuje programovací jazyky C, C++ a Fortran.“[2] Jinými slovy OpenACC je knihovna a soubor direktiv pomocí kterých programátor převede CPU kód na kód pro GPU jen s minimálním zásahem do původního procesorového kódu.

OpenACC je jednoduší na použití oproti OpenCL nebo CUDA a díky němu je možno mít jeden kód pro použití na CPU, jako jednovláknovou nebo vícevláknovou[1] aplikaci, nebo GPU aplikaci. OpenACC umí využít více GPU zároveň a navíc lze kombinovat s OpenMP a díky tomu jednoduše spravovat více GPU. Dále OpenACC obsahuje knihovní funkce pro práci s GPU.

Direktiva je řetězec, nebo řetězce oddělené mezerou, řádek začínající znakem „#“ a končící koncem řádku (lze sloučit řádky pomocí „\“ na konci řádku), které slouží jako řídící konstrukce pro preprocesor. Příklady direktiv: #define, #elif, #else, #endif, #error, #ident, #if, #ifdef, #ifndef, #include, #line, #pragma, #undef, #warning.

OpenACC využívá direktivu „#pragma acc“ kterou následuje libovolný počet acc klauzulí oddělených mezerou. Takováto direktiva platí pouze pro následující řádek, nebo blok kódu. Direktivy pro Fortran začínají „!$acc“ a jsou většinou stejné jako v C/C++.

3.1 Programování s OpenACC

Programátor pouze označí direktivou části kódu, které mají být paralelizovány a kompilátor vše zařídí. Ve složitějších případech může programátor více specifikovat, jak chce kód paralelizovat (můžeme tomu říkat optimalizace), protože se může stát že kompilátor může kód paralelizovat špatně, nebo vůbec, proto je dobré mít vždy zapnuté výpisy kompilátoru o provedené akceleraci.

[1] OpenACC umí target „multicore“ při kterém paralelizuje na CPU podobně jako OpenMP

3.2 Optimalizace

Jsou tři úrovně optimalizace a to gang, worker a vector, které pokud nejsou vynucené programátorem, tak jsou automaticky nastaveny kompilátorem.

První úrovní optimalizace je gang. Ten způsobí to iterace cyklu jsou rozděleny mezi gangy (práce je rozdělena mezi SM procesory) (ekvivalentem jsou threadbloky v CUDA) a zpracovány paralelně. Počet gangů může programátor specifikovat.

Dalšími úrovněmi jsou worker a vector. Vector znamená kolik se použije vláken v jednom warpu. Worker značí, kolik se těchto warpů použije v rámci gangu. Při porovnání s CUDA zjistíme že CUDA nemá ekvivalent pro nastavení počtu workerů.

Vynecháním klauzule worker získáme s použitím gang a vector přímý ekvivalent CUDA a to bloky a vlákna v bloku.

V CUDA lze nastavit množství vláken a bloků ve 3 dimenzích. V OpenACC stejného účinku dosáhneme, pokud pro každý vnořený cyklus nastavíme počet gangů a vectorů. Máme například 3-dimenzionální matici a pro každou dimenzi do sebe vnořené 3 cykly. Pokud v nejvnořenějším cyklu nastavíme gangy a vectory, tak ekvivalentem v CUDA je nastavení pro dimenzi X. Nadřazený cyklus - dimenze Y. A zbývajícím cyklem dimenzi Z.

Další užitečné možnosti optimalizace jsou klauzule seq, reduction a collapse. Seq vynutí sekvenční zpracování smyčky. Collapse způsobí, že kompilátor se pokusí sloučit následujících N vnořených cyklů, a reduction zase že, proměnná bude existovat v každém vlákně samostatně a na konci se nadevšemi instancemi provede určitá operace, jako například suma, maximum atd. například skalární součin vektorů je ideální příklad, kdy je potřeba na konci výpočtu sečíst všechny součiny prvků vektorů.

3.3 správa paměti

OpenACC z pohledu programátora nerozlišuje mezi ukazatelem do paměti PC a mezi ukazatelem do paměti GPU a vždy si kompilátor ukazatele vnitřně převádí. Díky tomu není potřeba udržovat 2 ukazatele na stejná data nacházející se na různých místech.

OpenACC nabízí 3 různé možnosti jak spravovat paměť

1. Automaticky - kompilátor sám zařídí kdy, na jak dlouho, kolik a jaká data se zkopírují na GPU. Pro tuto možnost je třeba nastavit managed mód při kompilaci např „-ta=nvidia:managed“. Toto je velice pohodlné, ale nemusí to být nejefektivnější řešení, navíc managed mód funguje pouze u grafických karet od nvidie.

2. Poloautomaticky pomocí klauzulí, pro blok kódu, které jako parametr obsahují seznam ukazatelů, nebo identifikátorů oddělených čárkou, dále jen data. Pokud cílem je ukazatel na pole prvků lze za ukazatel napsat [startovní index : počet prvků]. Např. #pragma acc copy(ukazatel[i:n], ...) - tím se od indexu „i“ zkopíruje na GPU počet prvků „n“ daného typu.

3. Ručně pomoci klauzulí musí programátor alokovat a uvolňovat paměť na GPU a navíc ji kopírovat z/na GPU. Takto lze dosáhnou nejlepší optimalizace, protože data mohou zůstat na kartě i když je zrovna nepotřebujeme. Navíc je možné mapovat pomocí knihovní funkce paměť PC do paměťového prostoru GPU, např. při nedostatku paměti karty.

Příklady direktiv naleznete v Příloze A.

3.4 Testování

Testování probíhalo za účelem ověření základní funkčnosti a jaký má význam použít OpenACC místo obyčejného programování na CPU. Jako testy byly zvoleny kopírování paměti, součet 2 matic, součin 2 matic, jacobiho iterační metoda, transpozice matice, a signálování mezi kernely. Výsledky jsou v mocninách 10 (pro účely této kapitoly kilo = 1000). Výsledky jsou znázorněné dvěma lineárními grafy, přičemž v pravém grafu je přiblížená zajímavá část levého.

„FLOPS je zkratka pro počet operací v pohyblivé řádové čárce za sekundu (FLoating-point Operations Per Second), což je obvyklé měřítko výpočetní výkonnosti počítačů.“ [2]

Testování probíhalo na stroji:

- CPU: 2x Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz, 6 jader, HT zřejmě vypnutý, 16 MB L3, 6x 256K L2, 6x 32K+32K L1
- RAM: 2x 32GB DDR3 2133MHz 4 kanály
- GPU: 4x GeForce GTX 1080 8GB

Pro tento účel jsem naprogramoval malou testovací aplikaci microbenchmark (tester).

3.4.1 Microbenchmark

Tato aplikace slouží ke spouštění testů, které testují výpočetní výkon při používání určitých operací, nebo výpočtů nad 2D maticemi, které slouží pro přibližnou představu výkonu. Jednotlivé testy mohou obsahovat více implementací dané operace, pro porovnání jednotlivých implementací mezi sebou. Podle vstupních parametrů nastaví cílovou velikost matic a počet iterací, opakovaně spouští jednotlivé implementace testů pro do maxima se zvyšující velikostí matic a na závěr spustí testy a automatickou volbou vybere nejlepší metody pro každou velikost matice.

Tester při měření výkonu ignoruje čas potřebný pro zkopírování dat na GPU, protože cílem při výpočtech na GPU je co nejmenší přenos dat mezi GPU a CPU. Iterace slouží ke zprůměrování výsledků měření a také pro zpřesnění měření velmi rychlých operaci. Nejprve se spustí měření času a teprve poté se vykoná x iterací stejného výpočtu, přičemž po skončení se měření zastaví a čas se vydělí počtem iterací.

V microbenchmarku jsou naprogramovány následující testy:

3.4.1.1 Kopírování paměti

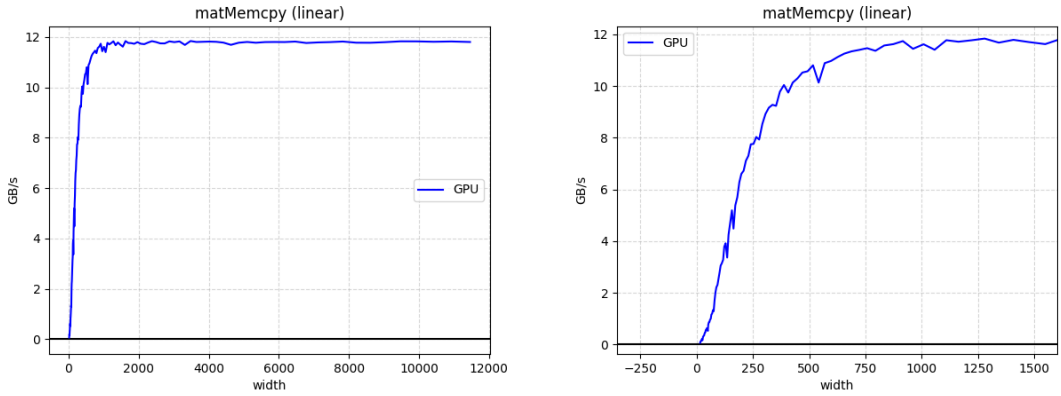
Má za úkol zjistit, zda OpenACC dokáže využít plnou propustnost při kopírování dat. Velikost kopírovaných dat byla cca 0 až 67MB a 1GB.

Kopírování 5MB až 570MB (šířka 2D matice cca 1000 až 12000) dat dosáhlo rychlosti průměrně 11,8 GB/s, pokud běžel program na druhém socketu tak pouze 4,5 GB/s

Kopírování dat o velikosti 0 až 570MB je vyjádřeno následujícím grafem 3.1

Z výsledků je patrné že se sice neblíží teoretické propustnosti PCI-E 3.0 16x, která činí 16GB/s, ale to může být dáno režií protokolu.

[2] https://cs.wikipedia.org/wiki/FLOPS

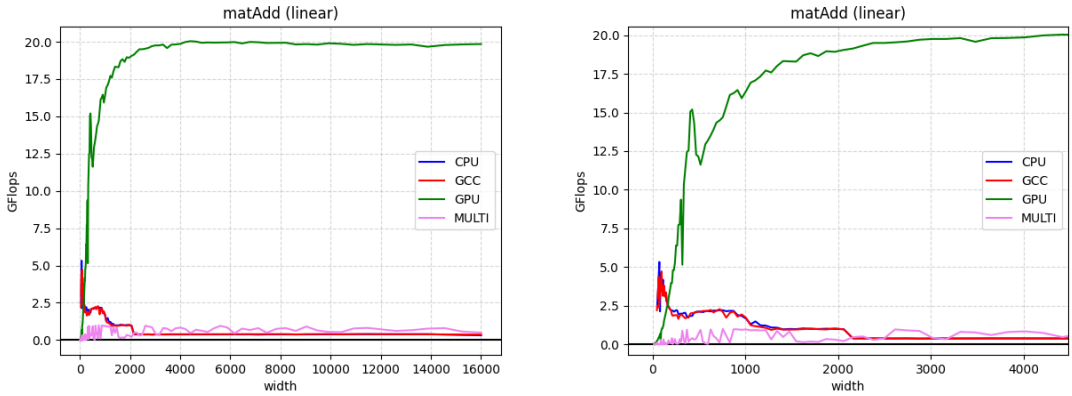


Obrázek 3.1: Kopírování dat na GPU

3.4.1.2 součet matic

Sčítání matic je ve výsledku sčítání vektorů (typu float) mat_a, mat_b a výsledek ukládá do vektoru mat_r. Každá matice má šířku strany 16 až 16000.

Na grafu 3.2 lze vidět, že maximální rychlost výpočtu na GPU je oproti jednovláknovému výpočtu na CPU až 20-krát vyšší. Při výpočtu na procesoru na 1 vlákně lze vidět snižování výkonu, jak se postupně zaplňují cache jednotlivých úrovní.

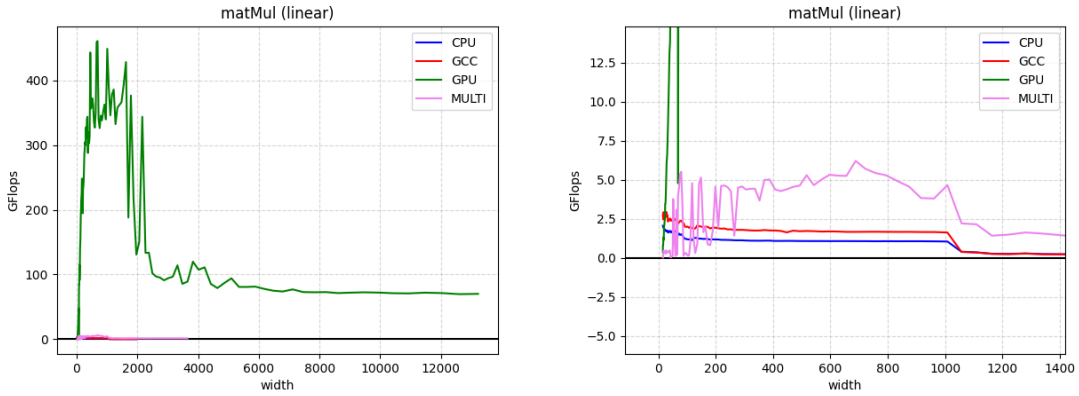


Obrázek 3.2: součet vektorů

3.4.1.3 součin matic

Součin dvou matic (typu float) mat_a, mat_b a následné uložení do matice mat_c. Každá matice má šířku strany 16 až 16000. Na grafu 3.2 lze vidět, že maximální rychlost výpočtu na GPU je oproti jednovláknovému výpočtu na CPU až 90-krát vyšší. Nejvýkonnější implementace na GPU je tehdy, pokud je nejvnořenější cyklus prováděn sekvenčně, tzn. není paralelizován. Maximum výkonu je asi 450GFlops, což je skoro polovina výkonu (1086.89 GFlop/s), kterého dosahuje program matrixMul z balíku cuda-samples [3]

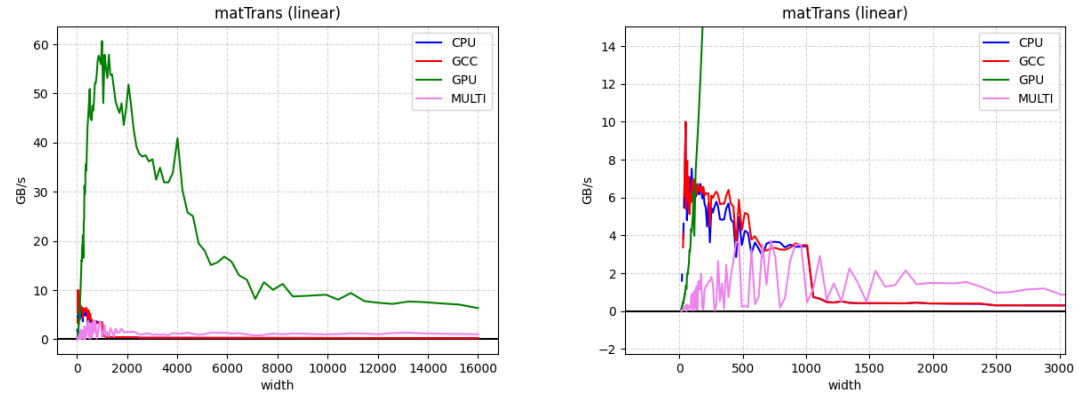
[3] https://github.com/NVIDIA/cuda-samples.git



Obrázek 3.3: součin matic

3.4.1.4 transpozice matice

Transpozice matice mat_a do matice mat_b. Tento test podobný součtu matic, ale je méně limitovaný rychlostí paměti, protože používá pouze 2 matice a díky tomu je dosaženo vyššího výkonu.

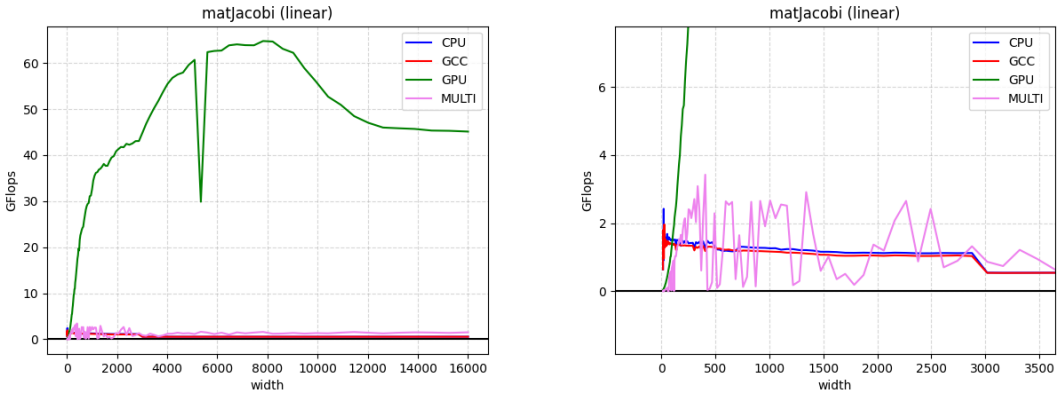


Obrázek 3.4: transpozice matice

3.4.1.5 Jacobiho iterace

Implementace je převzata z [5, 7]

Na grafu 3.5 lze vidět, že maximální rychlost výpočtu na GPU oproti jednovláknovému je až 64-krát vyšší.



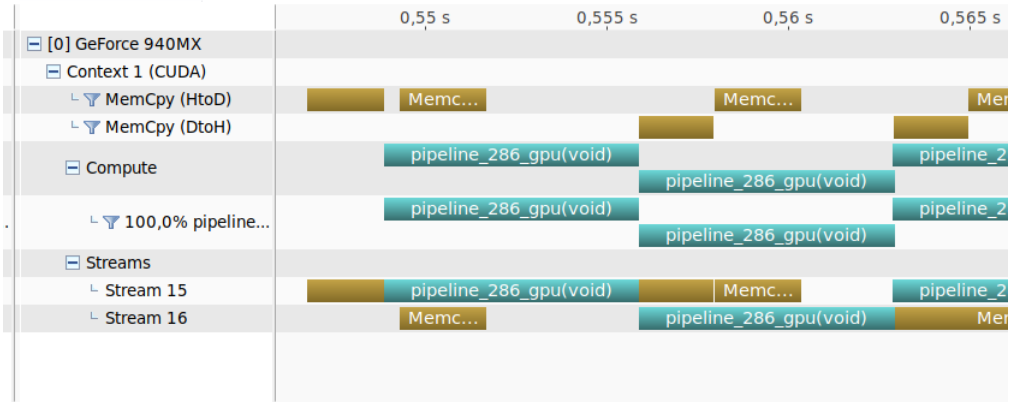
Obrázek 3.5: jacobiho iterace

Stejné testy z dalších PC jako například superpočítače Barbora lze vidět v Příloze B. V ní zjistíme že ne vždy se jednoduše vyplatí počítat na GPU.

3.4.2 signálování mezi kernely

Testování probíhalo na mém stroji s GPU Geforce 940MX. Na následujícím obrázku je zobrazena část pgi profileru.

Náplní testu bylo pouze naplnění vektoru, s tím, že úloha byla rozdělena do dvou asynchronních kernelů (streamů). Zlaté pruhy značí kopírování dat a tyrkysové pruhy značí výpočetní část. Lze zde vidět, že zatím co výpočetní část streamu 16 čeká na dokončení výpočtu streamu 15, tak data pro stream 16 se již mezitím zkopírovaly na GPU. Dále lze pozorovat, že ihned po ukončení výpočtu streamu 15, začal počítat stream 16 a zároveň se data ze streamu 15 kopírují zpět do hostitelského systému, přičemž se následně kopírují pro stream 15 nová data.



Kapitola 4

Simulace šíření tepla

V této kapitole se věnuji implementaci výpočtu simulace šíření tepla kWaveDiffusion, naprogramovaného v jazyce Matlab, a je součástí balíku k-wave. Nejprve trochu přiblížím, co kWaveDiffusion počítá. Tato práce ovšem nemá za cíl nijak upravovat rovnici nebo metody použité v kWaveDiffusion, takže si vystačíme s tím že výpočet je realizován k-space pseudospektrální metodou a šíření tepla je řešeno diferenciální rovnicí:

$$A * dT/dt = div(Kt * grad(T)) - B * (T - Ta) + Q$$

kde:

- A = hustota [kg/m^3] * měrná tepelná kapacita [J/(kg.K)] cílového tělesa
- Kt = je tepelná vodivost [W/(m.K)] tělesa
- B = hustota krve [kg/m^3] * měrná tepelná kapacita krve [J/(kg.K)] * průtok krve [1/s]
- Ta = teplota krve [°C]
- Q = dodávané teplo [W/m^3]

V homogenním tělese jsou tepelné koeficienty přepočítány na difúzní koeficienty

- diffusion coefficient [m^2/s] = Kt / A

koeficient prokrvení je dán vztahem

- perfusion coefficient [1/s] = B / A

4.1 Důležité vlastnosti Matlabu

Rád bych zmínil, že programování v Matlabu je lehce odlišné od programování v ostatních jazycích, jako například C/C++, python, java atd.

1. indexuje pole od 1
2. operátory začínající tečkou jako například .+ nebo .* nebo .^ jsou důležité při práci s maticemi, protože na rozdíl od obyčejných operátorů, které pracují s maticemi jako celky (např součin matic), tyto operátory pracují pouze s prvky matic, takže výsledkem A .* B není součin matic, ale matice jejíž prvky jsou výsledkem součinu prvků matic A a B o stejném indexu. Příklad můžete vidět na obrázku 4.1

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} .* \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 9 & 12 \end{bmatrix}$$

Obrázek 4.1

3. Fourierova transformace vrací komplexní pole o neredukované velikosti.
4. při použití prvkového operátoru .+, .* atd. se nerozlišuje, jestli je operand konstanta nebo pole čísel.

4.2 Implementace v Matlabu

- obj.FT(x) je metoda vracející výstup z dopředné Fourierovy transformace nad polem x
- obj.IT(x) je metoda vracející výstup ze zpětné Fourierovy transformace nad polem x

Hlavní cyklus výpočtu vypadá takto

```
for t_index = 1:Nt
    if use_perfusion
        p_term = - obj.perfusion_coeff .* obj.IT( kappa .* obj.FT(obj.T - obj.
    blood_ambient_temperature) );
    end
    if obj.flag_homogeneous
        d_term = obj.diffusion_p1 .* obj.diffusion_p2 .* obj.IT( -obj.k.^2 .* kappa .*
    obj.FT(obj.T) );
    else
        T_FT = obj.FT(obj.T);
        switch obj.dim
            case 2
                d_term = obj.diffusion_p1 .* ( obj.IT( deriv_x .* obj.FT( obj.
    diffusion_p2 .* obj.IT( deriv_x .* T_FT ) ) ) + obj.IT( deriv_y .* obj.FT( obj.
    diffusion_p2 .* obj.IT( deriv_y .* T_FT ) ) ) );
            case 3
                d_term = obj.diffusion_p1 .* ( obj.IT( deriv_x .* obj.FT( obj.
    diffusion_p2 .* obj.IT( deriv_x .* T_FT ) ) ) + obj.IT( deriv_y .* obj.FT( obj.
    diffusion_p2 .* obj.IT( deriv_y .* T_FT ) ) ) + obj.IT( deriv_z .* obj.FT( obj.
    diffusion_p2 .* obj.IT( deriv_z .* T_FT ) ) ) );
        end
    end
    obj.T = obj.T + dt .* ( d_term + p_term + q_term );
    obj.cem43 = obj.cem43 + dt ./ 60 .* ( 0.5 .* (obj.T >= 43) + 0.25 .* (obj.T >= 37 &
     obj.T < 43 ) ).^(43 - obj.T);
end
```

Výpis 4.1: výpočet šíření tepla

V celém výpočtu jsou použity pouze operace: dopředná a zpětná rychlá Fourierova transformace, sčítání matic a násobení prvků matic o stejných indexech. Hlavní cyklus výpočtu se dá rozdělit do 4 částí:

1. výpočet p_term = teplo odvedené proudící krví
2. výpočet d_term = změna tepla v důsledku šíření tepla tělesem
   (a) pro homogenní těleso
   (b) pro heterogenní těleso
3. výpočet T = nová teplota tělesa
4. výpočet cem43 = informace o poškození tkáně v důsledku působení tepla

Dále výpočet z pohledu programátora zesložitují tyto problémy, které jinak za něho řeší jazyk Matlab:

- Do výpočtu vždy vstupují obj.T, obj.diffusion_p2, obj.diffusion_p1 a dt. Podle kombinace flagů „use_perfusion“ a „obj.flag_homogeneous“ vstupují do výpočtu k, kappa, deriv_x, deriv_y, deriv_z, obj.blood_ambient_temperature a obj.perfusion_coeff.
- Výstupem je vždy nová teplota obj.T a poškození tkání obj.cem43.
- Podle vstupních dat programu se tyto proměnné mohou objevit jako konstanta nebo pole dat: obj.diffusion_p1, q_term, obj.blood_ambient_temperature a obj.perfusion_coeff.

Výpočetní cyklus tedy není moc složitý, ale obsahuje poměrně mnoho větvení v závislosti na vstupních datech, které Matlab řeší automaticky díky svému návrhu.

Kapitola 5

Implementace v C++

V této kapitole vysvětlím aktuální návrh a později podrobněji implementaci programu heatsim, ale nejdříve proberu původní návrh.

V původním návrhu bylo, že využiji již existující aplikaci k-Wave-Fluid-OMP, která simuluje proudění kapalin a je napsána v C++ s využitím knihovny OpenMP. Využít se měla velká část kódu pro správu matic, načítání souborů, zpracování parametrů a vzorkování výpočtu. Po částečné implementaci simulace úpravou fluidního kódu se ukázalo, že je to velice pracné a nestabilní řešení, protože například smazání neaktivního kódu způsobilo zprovoznění a správnou funkci FFT[1] ale mé výpočetní kernely[2] stále nefungovaly jak měly a proto jsem s využitím kódu pro správu paměti z microbenchmarku paralelně programoval vlastní verzi simulace. Nakonec jsem zavrhl implementaci s použitím fluidního kódu a pokračoval ve své vlastní, přičemž jsem použil některé nápady z fluidního kódu, jako například kontejner pro sdružení matic či použití HDF5 souborů pro komunikaci s programem kWaveDiffusion.

5.1 Struktura heatsim

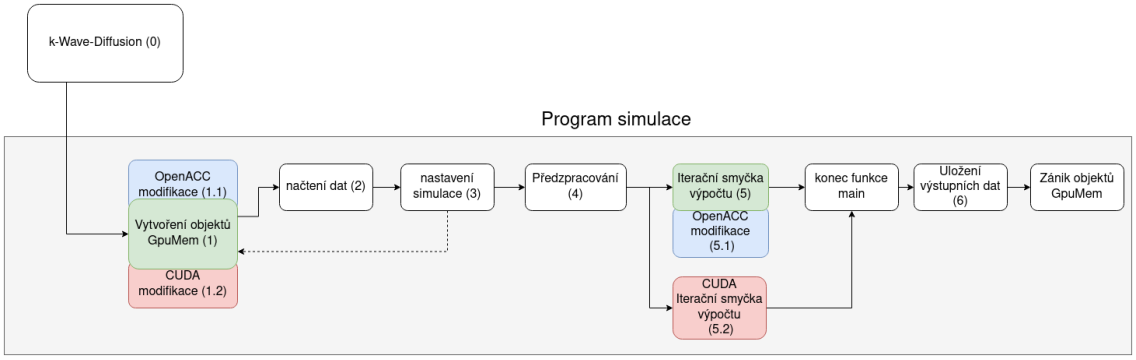
V rámci řešení jsem navrhl následující strukturu programu heatsim (dále jen program), kterou můžete názorně vidět na přiložených diagramech 5.1 a 5.2. Tyto diagramy jsou pro lepší představu trochu více abstraktní než je reálná struktura programu. Diagram 5.1 znázorňuje fáze, v jakých se může program nacházet. Plné šipky znamenají přímé „volání“ a čárkované šipky znamenají znovu spuštění nad jinými daty. Diagram 5.2 znázorňuje práci a pohyb s velkými a malými daty. Velká data, neboli datové matice, jsou znázorněny tlustými šipkami a malá data (řídící proměnné atd.) tenkými šipkami. Tlusté jednosměrné šipky znamenají přesun dat a obousměrné reprezentují práci s daty.

Program obsahuje jedinou třídu GpuMem (1) a hlavními částmi jsou výpočetní smyčka (5), výpočetní kernely, kontejner matic (10), FFT funkce a pomocné funkce.

Součástí návrhu je upravený kWaveDiffusion (0) a program heatsim nahrazuje pouze jeho výpočetní část a to z důvodu že příprava dat v kWaveDiffusion trvá minimální dobu oproti samotnému výpočtu. kWaveDiffusion vygeneruje vstupní soubory (9 a 9.1) pro program heatsim a spustí ho. Program vytvoří objekt třídy GpuMem, který v rámci inicializace načte hdf5 soubor s řídícím polem dat (2), následně se nastaví některé parametry simulace a

[1] knihovní funkce pro rychlou Fourierovu transformaci

[2] kernelem se označuje funkce spouštějící se na GPU, ale pro jasné označení funkcí určených pro výpočet, budu tento název používat i když funkce poběží na CPU



Obrázek 5.1: fáze programu

vytvoří se objekty třídy GpuMem pro všechny matice (2.1), co se mohou ve výpočtu objevit, ale alokují se jen podle nastavení simulace, díky čemuž se sníží paměťová náročnost.

Následuje předzpracování (4), kde se matice kappa, k , deriv_xyz (které mají dimenze o plné velikosti a figurují ve výpočtu společně maticemi o redukované velikosti) zmenšují z plné na redukovanou velikost. Díky tomu se sníží paměťová náročnost, a hlavně lze takto jednoduše provádět výpočet v CUDA verzi programu. Také se zde některé vstupy, které se objevují v homogenní i heterogenní formě, rozbalí ve formě konstanty do pole těchto konstant, a tím se sníží počet složitost některých částí výpočtu a nemusí se vytvářet další přetížení výpočetních kernelů. Nevýhodou je v určitých situacích zvýšená náročnost na paměť.

Jako poslední, pokud probíhá výpočet na GPU se nakopírují data na GPU, a spustí se iterační smyčka (5). V této smyčce se spouští kernely a FFT funkce. Výsledkem v každém cyklu je nová teplota T a poškození tkání cem43.

Po ukončení výpočtu se případně zkopírují data zpět do hostujícího počítače a ukončí se program. Objekty třídy GpuMem nastavené jako výstupní v rámci destruktoru uloží datovou matici (6) na disk ve formátu hdf5.

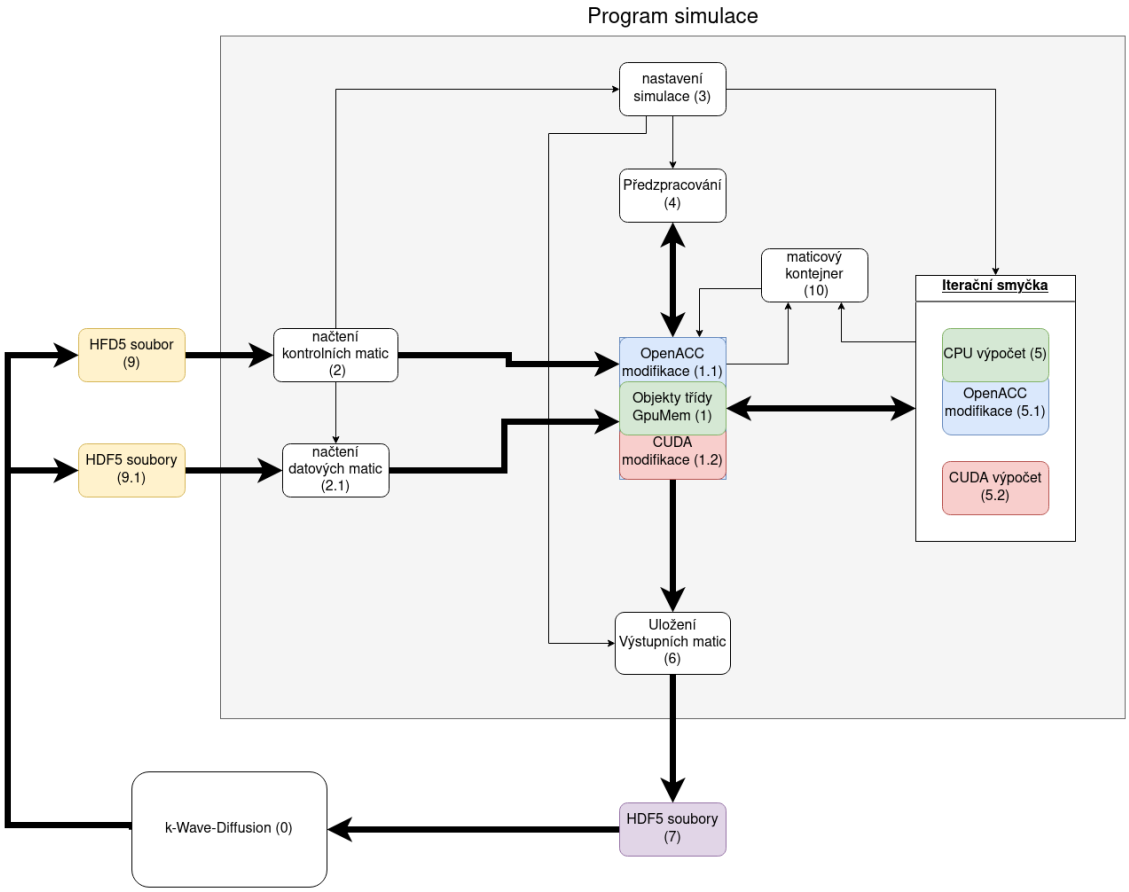
KWaveDiffusion načte výstupní soubory z heatsim (7) a pokračuje ve vlastním postprocesingu a případném zobrazení grafů.

Výhodou tohoto řešení, kdy KWaveDiffusion akceleruje výpočet pomocí heatsim, je, že pro uživatele KWaveDiffusion se téměř nic nezmění. Nevýhodou je zvýšená paměťová náročnost, protože běží heatsim i KWaveDiffusion zároveň.

5.1.1 Vstupy Heatsim (9)

Jediné vstupy co program přijímá jsou 1D/2D/3D pole ve formátu HDF5 a jsou to:

- controll_mat.h5 - řídící matice o velikosti 3 pole, obsahující info zda se jedná o homogenní nebo heterogenní simulaci, počet dimenzí a info zda se bude počítat chlazení krví (use_perfusion).
- export_dim.h5 - řídící matice o velikosti počet dimenzí + 1, obsahuje velikosti dimenzí X, Y, Z a počet kroků simulace
- T.h5 - datová matice se vstupní teplotou
- k.h5, kappa.h5, deriv_xyz.h, diffusion_p1.h5, diffusion_p2.h5, perfusion_coeff.h5, blood_ambient_temperature.h5 - datové matice obsahující jednu nebo pole konstant



Obrázek 5.2: pohyb dat

- dt.h5 - konstanta s velikostí kroku simulace
- q_term.h5 - datová matice obsahující zdroj tepla

5.1.2 Výstupy Heatsim (7)

výstupy jsou 2D/3D pole stejně jako vstupy ve formátu HDF5

- T.h5 - finální teplota tkáně
- cem43.h5 - poškození tkáně v důsledku působení tepla

5.1.3 typy datových matic

Všechny datové matice jsou ve formátu

- podle velikosti
  - 1-prvkového pole (konstanta)
  - 2D/3D pole o plné velikosti X, Y, Z
  - 2D/3D pole o redukované velikosti X/2 + 1, Y, Z
- podle datového typu
  - float
  - FloatComplex = std::complex<float>

5.1.4 Třída GpuMem (1)

Třída template<class TYPE> class GpuMem je templatovaná třída pro správu matic v paměti. Slouží pro správu paměti na GPU, kde se stará o alokování paměti na GPU, kopírování dat na GPU a z GPU zpět do paměti počítače a dealokaci. Dále slouží pro načítání a ukládání svého obsahu na disk do formátu hdf5. Objekt třídy může existovat bez alokované paměti pro data, díky čemuž může existovat po celou dobu běhu programu, i když není potřeba a není nutné je nějak složitě dynamicky vytvářet a vyhledávat.

- Důležité metody
  - GpuMem(uint64_t size, std::string name, bool use_gpu, bool load, bool store, bool alloc);
    - konstruktor, který podle svých argumentů alokuje paměť pro data a případně zařídí nahrání dat z hdf5 souboru do paměti.
    - uloží záznam o sobě do maticového kontejneru
    - argumenty:
      - size - požadovaný počet prvků k alokaci
      - name - jméno matice, slouží ke specifikování jména vstupního souboru a také pro ladění programu
      - use_gpu - určuje zda se při volání metod pro práci s GPU opravdu dané metody provedou.
      - load - konstruktor načte data ze souboru hdf5 a ignoruje argument size.

        - store - destruktor před uvolněním paměti uloží data do souboru.
        - alloc - zapíná/vypíná alokaci paměti pro data, při vypnutém stavu zároveň ignoruje příznak load.
  - virtual ~GpuMem()
    * destruktor, který před uvolněním paměti, pokud je nastavená hodnota ___store, uloží data do hdf5 souboru.
    * vymaže záznam o sobě z kontejneru
  - TYPE * data()
    * vrátí ukazatel na data
  - TYPE * g_data();
    * vrátí ukazatel na data v paměti GPU
  - void createOnDev()
    * alokuje data na GPU
  - void deleteOnDev()
    * uvolní data na GPU
  - void cpyTD()
    * zkopíruje data na GPU a případně i alokuje data na GPU
  - void cpyTH()
    * zkopíruje data z GPU zpět do paměti PC

5.1.5 FFT

Funkce fftR2C() a fftC2R() slouží k výpočtu dopředné a zpětné Fourierovy transformace pomocí knihovní funkce *fftwf_execute_dft_r2c()* a *fftwf_execute_dft_c2r()* z knihovny *fftw3.h*, nebo *cufftExecR2C()* a *cufftExecC2R()* z knihovny *cufft.h* v závislosti na tom jestli výpočet probíhá na CPU nebo GPU. To která dvojice se použije je určeno při kompilaci pomocí direktivy GPU_ACC nebo GPU_CUDA.

Vstupní pole dopředného FFT je plné velikosti a výstupní pole je velikosti redukované. Kvůli tomuto faktu se musí některé datové matice redukovat z plné velikosti, aby bylo efektivnější s nimi provádět operace.

Výstup z fftC2R() je potřeba normalizovat, v našem případě vynásobit konstantou *idivider* = 1 / (součin velikostí dimenzí)

5.1.6 maticový kontejner (10)

Kdyby to byla třída, tak by se dala označit jako jedináček. Kontejner matic (objektů třídy GpuMem) slouží primárně k provádění hromadných operací nad objekty GpuMem jako je alokace a kopírování dat na GPU a zpět. Dále slouží pro vypočítání použité paměti. Sekundární účel je pro ladění programu, kdy provádí výpisy jaké objekty jsou alokované, jak jsou velké, jejich ukazatele na data a další.

5.1.7 výpočetní smyčka (5)

Výpočetní smyčka provádí předem stanovený počet kroků simulační metody a pracuje v homogenním nebo heterogenním módu podle nastavení simulace. Vnitřek smyčky je vložen do souborů homo.hpp pro homogenní mód a nohomo.hpp pro heterogenní mód a načten pomocí #include do příslušného místa ve smyčce pro zvýšení přehlednosti a zachování možnosti používat lokální proměnné z funkce *main*. V obou módech se může počítat p_term rozdíl je pouze ve vstupních parametrech kdy v homogenním módu jsou *perfusion_coeff* a *blood_ambient_temperature* konstanty. V každém kroku je vypočtena nová hodnota d_term, T a cem43.

V následující části představím implementaci jednotlivých částí simulace do kroků. Jednotlivé kroky pro přehlednost buď obsahují volání FFT funkce, nebo vnitřek cyklu příslušného kernelu. Pokud nějaká proměnná obsahuje index *r*, tak to znamená že v kernelu je cyklus který prochází plné dimenze. Pokud se v kroku vyskytuje index *c*, tak kernel obsahuje cyklus procházející redukované dimenze. Díky tomu že se v kernelech vyskytují pouze operace .+ a .* je možné procházet 2D i 3D matice jako 1D vektor.

Výpočet potřebuje větší množství datových objektů (matic) a dělí se na vstupně/výstupní datové objekty *T, cem43, dt, q_term, k ,kappa, perfusion_coeff, blood_ambient_temperature, deriv_x, deriv_y, deriv_z, diffusion_p1, diffusion_p2* a pomocné datové objekty *d_term, p_term, tmpFft, tmpTff, fft_1, fft_2, fft_3 a tff_1, tff_2, tff_3.*

5.1.7.1 Perfuze

Neboli „průtok krve určitou tkání, nebo orgánem“[8] je zde použit pro výpočet *p_term,* což je teplo odebrané proudící krví. V k-Wave-Diffusion je implementovaný jako 5.1

```
1 p_term = - obj.perfusion_coeff .* obj.IT( kappa .* obj.FT(obj.T - obj.
      blood_ambient_temperature) );
```

Výpis 5.1: odvod tepla

Tento výpočet jsem rozložil do 5 kroků a to

1. p_term[r] = T[r] - temp[r];[3]
2. fftR2C(p_term, tmpFft);
3. tmpFft[c] *= kappa[c];
4. fftC2R(tmpFft, p_term);
5. p_term[r] *= -coef[r] * idivider; [4]

5.1.7.2 homogenní d_term

Výpočet změny tepla v homogenním tělese vypadá v matlabu takto 5.2

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

Výpis 5.2: homogenní d_term

[3] temp = obj.blood_ambient_temperature
[4] coef = obj.perfusion_coeff

a je rozložen do 4 kroků:

1. fftR2C(T, tmpFft);
2. tmpFft[c] = -k[c] * kappa[c] * tmpFft[c];
3. fftC2R(tmpFft, tmpTff);
4. d_term[r] = diffusion_p1 * diffusion_p2 * tmpTff[i] * idivider;

5.1.7.3 heterogenní d_term

Výpočet změny tepla v heterogenním tělese

```
1 T_FT = obj.FT(obj.T);
2     switch obj.dim
3         case 2
4             d_term = obj.diffusion_p1 .* ( obj.IT( deriv_x .* obj.FT( obj.diffusion_p2
      .* obj.IT( deriv_x .* T_FT ) ) ) + obj.IT( deriv_y .* obj.FT( obj.diffusion_p2 .*
      obj.IT( deriv_y .* T_FT ) ) ) );
5         case 3
6             d_term = obj.diffusion_p1 .* ( obj.IT( deriv_x .* obj.FT( obj.diffusion_p2
      .* obj.IT( deriv_x .* T_FT ) ) ) + obj.IT( deriv_y .* obj.FT( obj.diffusion_p2 .*
      obj.IT( deriv_y .* T_FT ) ) ) + obj.IT( deriv_z .* obj.FT( obj.diffusion_p2 .* obj.
      IT( deriv_z .* T_FT ) ) ) );
```

Výpis 5.3: heterogenní d_term

pro zpřehlednění zde nebudu vypisovat stejné kroky pro všechny dimenze, ale pomyslně sloučím proměnné *deriv_x, deriv_y, deriv_z* do *deriv* a obdobně pro *fft_1, fft_2, fft_3 a tff_1, tff_2, tff_3* budu zde označovat jako *fft* a *tff.*

1. fftR2C(T, tmpFft);
2. fft[c] = tmpFft[c] * deriv[c];
3. fftC2R(fft, tff);
4. tff[r] *= idivider * diffusion_p2;
5. fftR2C(tff, fft);
6. fft[c] *= deriv[c];
7. fftC2R(fft_1, tff_1);
8. tff[r] *= idivider;
9. d_term[r] = diffusion_p1[r] * ((tff_1[r]) + (tff_2[r]) + (tff_3[r]));

5.1.7.4 T

```
1 obj.T = obj.T + dt .* ( d_term + p_term + q_term );
```

Výpis 5.4: výsledná teplota

Kernel pro výpočet teploty je velmi primitivní: T[r] += dt * (d_term[r] + p_term[r] + q_term[r]); Na rozdíl od matlabu se musí použít více verzi tohoto kernelu, v závislosti na kombinaci existence q_term a p_term.

5.1.7.5 cem43

```
1 obj.cem43 = obj.cem43 + dt ./ 60 .* ( 0.5 .* (obj.T >= 43) + 0.25 .* (obj.T >= 37 & obj
      .T < 43 ) ).^(43 - obj.T);
```

Výpis 5.5: cem43

Výpočet cem43 jsem pro přehlednost rozdělil do 3 kroků

1. float tmp = 0.5 * (T[i] >= 43) + 0.25 * ((T[i] >= 37) & (T[i] < 43));
2. float power = pow(tmp, 43.0 - T[i]);
3. cem43[r] = cem43[r] + dt * (1.0/60) * power;

Kapitola 6

Akcelerace pomocí OpenACC

Tato kapitola se věnuje převodu procesorového kódu do kódu pro grafické karty za použití technologie OpenACC a jejími problémy.

Nejprve je potřeba si uvědomit že GPU má vlastní paměťový prostor, takže vhodné si kopírování na GPU nějak zjednodušit nebo zautomatizovat, aby programátor nemusel ke každému poli psát #pragma acc copyin(tmp[0:1]) atd. V našem případě se o to postarají metody createOnDev(), deleteOnDev(), cpyTD() a cpyTH() (1.1), které se pomocí direktivy *GPU_ACC* kompilují jen v případě kompilace na GPU, jinak jsou prázdné. Takto zapouzdřené pragmy občas kompilátor nevidí, v tom případě je potřeba mu trochu pomoct užitím #pragma acc data present(), která za běhu programu zjistí jestli data jsou opravdu na GPU a případně násilně ukončí program s chybou. Než se pustíme do přeměny samotné výpočetní části (5.1) je potřeba vyřešit problém s FFT. Funkci která počítá FFT na CPU nemůžeme jednoduše přesunout na GPU pomocí pragmy, zde se nabízí možnost nechat FFT na CPU, ale je to velice neefektivní řešení, protože před každým voláním a po jejím dokončení by se musela kopírovat paměť a tomu chceme zamezit. Proto použijeme FFT používající GPU z knihovny *cufft.h*, která je naprogramovaná v CUDA a tudíž vyžaduje ukazatele do paměti GPU. Ty zajistí kompilátor užitím #pragma acc host_data use_device().

Nyní je možný převod výpočetních kernelů. Ten je v našem případě velmi jednoduchý, protože všechny kernely prochází 1D vektory a všechny operace nad prvky těchto vektorů jsou na sobě nezávislé. Proto si vystačíme pouze s #pragma acc parallel loop.

Všechny kernely vypadají zhruba takto

```
1 void kernel(float *a, float *b, float *c, int N){
2     for(int i = 0; i < N; ++i)
3         a[i] += b[i] * c[i];
4 }
```

Výpis 6.1: Jak vypadá průmerný kernel v projektu.

a po použití OpenACC takto (5.1)

```
1 void kernel(float *a, float *b, float *c, int N){
2     #pragma acc parallel loop
3     for(int i = 0; i < N; ++i)
4         a[i] += b[i] * c[i];
5 }
```

6.1 Problémy

Asi nejzákeřnějším problémem je paradoxně skutečnost, že OpenACC využívá direktivy. Pokud totiž uděláte chybu v zápisu, tak je veliká šance, že vás kompilátor nijak neupozorní. Například když napíšete #pragma host_data use_device(a,b) místo #pragma acc host_data use_device(a,b), tak rozdíl velmi snadno přehlédnete. Na první pohled se vše tváří v pořádku, protože kompilátor ignoruje pragmy které nezná a vy pak strávíte mnoho času hledáním příčiny proč daná metoda nefunguje a nakonec to poznáte až z grafického profileru, kde zjistíte, že se daná metoda spouští například již před dokončením kopírování dat na GPU, nebo se spustí na úplně jiné kartě.

Další problém na který jsem narazil byl u templatované třídy GpuMem, konkrétně u metod spravujících data na GPU (createOnDevice(), cpyTH(), atd.). U objektů s daty typu FloatComplex[1] totiž docházelo ke kopírování jen poloviny prvků z matice, což odpovídá tomu, že i když byly data objektu typu FloatComplex, tak je považoval za float, který má poloviční velikost než FloatComplex. Tento problém jsem obešel pomocí pomocného ukazatele na data vytvářeného uvnitř metod, který se použil pouze, pokud byla velikost datového typu objektu větší, než float. Velikost datového typu se ukládá při inicializaci objektu.

[1] std::complex<float>

Kapitola 7

Akcelerace pomocí CUDA

Tato kapitola pojednává o převodu implementaci pomocí CUDA.

7.1 Implementace

Práce s daty je velmi podobná OpenACC, paměť se musí alokovat na GPU a přesunout na ní data, k tomu slouží funkce cudaMalloc(), cudaFree(), cudaMemcpy(). Rozdíl je v tom že programátor musí sám udržovat 2 druhy ukazatelů a to jeden do paměti PC a druhý do paměti GPU. Pomocí direktivy *GPU_CUDA* jsou při kompilaci modifikované metody createOnDev(), deleteOnDev(), cpyTD() a cpyTH() pro použití s CUDA kódem (1.2). Pomocí direktivy *GPU_CUDA je také vyměněn obsah souborů s implementací.* Třída GpuMem obsahuje novou metodu *TYPE * g_data()*, která vrací ukazatel do paměti GPU. Pro FFT je použita také knihovna cufft.h, s tím rozdílem že se funkce volají s ukazateli do paměti GPU.

Implementace kernelů je triviální ze stejného důvodu jako u OpenACC. Kernely 6.1 jsou transformovány vždy do 2 funkcí

```
1  __global__ void __kernel(float *a, float *b, float *c, int N){
2      int i = threadIdx.x + blockDim.x * blockIdx.x;
3      if (i < N)
4          a[i] += b[i] * c[i];
5  }
6  void kernel(float *a, float *b, float *c, int N){
7      dim3 DimGrid((N-1)/CUDA_WIDTH + 1, 1, 1);
8      dim3 DimBlock(CUDA_WIDTH,1,1);
9
10     __kernel<<<DimGrid, DimBlock>>>(a, b, c, N);
11 }
```

kde kernel() je funkce která se volá z CPU a spouští funkci ___kernel() na GPU. DimBlock nastavuje kolik se bude spouštět vláken uvnitř bloku. DimGrid nastaví na kolik bloků bude rozděleno datové pole. V našem případě CUDA_WIDTH = 1024, což je počet vláken v dané dimenzi na 1 blok. Po spuštění kernelu na GPU, každé vlákno zpracovává funkci ___kernel(), přičemž dostane svoji pozici v rámci datového pole pomocí 3 proměnných threadIdx.x, blockDim.x, a blockIdx.x. „if (i < N)“ je tu zde proto, když datové pole není zarovnané na počet vláken * počet bloků, aby nedocházelo k přístupu mimo datové pole.

7.2 Problémy

Program běží bez chyby, ale dodává špatné výsledky. To se může stát pokud kompilátoru nezadáte přepínač --generate-code se správnou verzí architektury vašeho GPU, nebo ho nezadáte vůbec. Program se pak spustí bez chyby, ale kernely se nevykonají. O této skutečnosti dostanete varování od kompilátoru, které se může velmi jednoduše ztratit ve zbylém výpisu o kompilaci, a to jen pro karty nacházející se v počítači, kde probíhá kompilace. Další problém na který jsem narazil bylo že na jedné z 5 testovaných modelů grafických karet se jen na jedné z nich nespouští jeden z kernelů heterogenní simulace. O týden později se stejným kódem výpočet funguje správně.

Kapitola 8

OpenACC vs. CUDA

V této kapitole se věnuji porovnání obou technologií, jak z hlediska programování, tak z hlediska výkonového.

8.1 Programování

V obou technologiích si programátor musí zvyknout na to že se musí starat o stejná data na dvou místech. Dle mého názoru je velmi užitečné mít procesorovou verzi programu, protože ve chvíli kdy se objeví nějaké chyby v návrhu, tak je mnohem jednodušší a příjemnější ladit program na CPU. Výhodou OpenACC je skutečnost že procesorová verze programu je společná i pro akcelerovanou verzi. V případě CUDA je nutné mít procesorovou verzi zvlášť a tím spravovat dvě implementace. Při použití CUDA technologie musí programátor lépe porozumět vnitřní stavbě grafického čipu než je tomu v případě použití OpenACC. Je množství věcí, na které programátor při použití OpenACC vůbec nemusí myslet. Například kolik je ideální množství vláken na jeden blok, nebo který ukazatel má použít. Pokud by zvolil managed mód, tak se nemusí trápit ani se správou paměti za cenu výkonového propadu.

Při porovnání náročnosti obou implementací musíme vzít v potaz dva fakty:

1. Ve výpočetních kernelech byly použity pouze operace sčítání a násobení v 1D poli, nebyly tedy použity nějaké pokročilejší konstrukce kernelů, kde by se více projevily rozdíly v náročnosti.

2. Ve fázi akcelerace byl k dispozici funkční kód simulace v procesorové verzi, proto nelze přesně odhadnout náročnost implementace v CUDA bez procesorové verze.

S přihlédnutím k těmto faktům, lze říci že implementace v technologii CUDA byla náročnější pouze z hlediska času, potřebného k vytvoření kopií existujících kernelů a vytvoření spouštěcích funkcí. Tato skutečnost by se dala potlačit nějakým vhodným doplňkem editor/IDE, který by ze zvolených funkcí vytvářel klidně i prázdné kernely.

Když odečtu čas potřebný pro implementaci správy paměti, která je téměř totožná pro obě technologie a čas potřebný pro implementaci FFT, která je sdílená, tak mým hrubým odhadem je, že pro akceleraci pomocí OpenACC byl potřeba čas v řádu desítek minut a pro akceleraci pomocí CUDA byl potřeba čas v řádu hodin.

Procesor	RAM	Grafická karta
Intel Xeon Gold 6126 CPU @ 2.60GHz[1]	187G	NVIDIA Tesla V100 SXM2 16GB
Intel Xeon CPU E5-2620 v4 @ 2.10GHz	503Gi	NVIDIA Tesla P40
Intel Xeon CPU E5-2620 v3 @ 2.40GHz	62Gi	NVIDIA GeForce GTX TITAN X
Intel Xeon CPU E5-2620 v3 @ 2.40GHz	62Gi	NVIDIA GeForce GTX 1080
Intel Core i7-3770 CPU @ 3.40GHz	31Gi	NVIDIA GeForce RTX 2080 Ti
Intel Core i7-6820HQ CPU @ 2.70GHz	15Gi	NVIDIA GeForce 940MX

Tabulka 8.1: Použité stroje pro srovnání výkonu.

Grafická karta	Teoretický výkon FP32	RAM	propustnost
NVIDIA Tesla V100 SXM2 16GB	15.67 TFLOPS	16 GB	897.0 GB/s
NVIDIA Tesla P40	11.76 TFLOPS	24 GB	694.3 GB/s
NVIDIA GeForce GTX TITAN X	6.691 TFLOPS	12 GB	336.6 GB/s
NVIDIA GeForce GTX 1080	8.873 TFLOPS	8 GB	320.3 GB/s
NVIDIA GeForce RTX 2080 Ti	13.45 TFLOPS	11 GB	616.0 GB/s
NVIDIA GeForce 940MX	953.9 GFLOPS	2 GB	16.02 GB/s

Tabulka 8.2: Parametry grafických karet

8.2 Výkon

Pro testování výkonnostních rozdílů mezi implementacemi OpenACC a CUDA byly použity stroje s těmito parametry uvedené v tabulce 8.1.

Jako metodu testování jsem jsem zvolil porovnání času na dokončení jedné iterace výpočtu v heterogenní simulaci se zapnutou perfuzí a zdrojem tepla. Čas byl změřen v 50 iteraci ze 300 za pomoci funkce gettimeofday() z knihovny sys/time.h. Testování probíhalo nad daty v podobě 2D matice o šířce 128, 256 ,512 ,1024, 2048, 4096, 8000 a 8192. Výsledky testování jsou v tabulce 8.3.

Ve 2 až 9 sloupci jsou naměřené hodnoty v milisekundách. Hodnota / znamená že měření nebylo úspěšně provedeno z důvodu nedostatečné velikosti paměti na GPU.

Z tabulky je patrné, že ve většině případů je rozdíl časů potřebný pro jeden krok simulace velmi malý, velmi pravděpodobně to bude způsobeno tím že během testování jsem neměl výlučný přístup k testovaným systémům.

Při porovnání teoretických výkonů karet a propustnosti jejich pamětí z tabulky 8.2 s rychlostí výpočtu z tabulky 8.3, vyplývá zajímavé zjištění že poměr času potřebného k výpočtu vůbec odpovídá poměru teoretického výkonu grafických karet a neodpovídá ani propustnosti jejich pamětí. Poměr nesedí ani v případě kdybychom výkon karet měřili v teoretický výkon * propustnost paměti. Z toho vyplývá že program nedosahuje limitů ani procesoru grafické karty, ani propustnosti její paměti.

Zajímavé zjištění je při porovnání časů z výpočtu 8000 a 8196, kdy všechny karty dosahují kratších času v simulaci s většími maticemi.

[1] superpočítač Barbora

GPU	128	256	512	1024	2048	4096	8000	8192	Implementace
Tesla V100	0.22	0.25	0.31	0.66	2.23	8.41	47.6	33.3	OpenACC
	0.11	0.13	0.19	0.54	2.13	8.39	47.7	33.5	CUDA
Tesla P40	0.34	0.43	0.73	2.48	9.0	35.1	148	137	OpenACC
	0.34	0.45	0.72	2.45	9.06	34.7	148	137	CUDA
TITAN X	0.38	0.46	0.94	2.96	11.4	44.6	198	180	OpenACC
	0.34	0.43	0.79	2.99	11.5	44.9	198	178	CUDA
GTX 1080	0.30	0.40	0.71	2.82	10.9	42.8	173	160	OpenACC
	0.18	0.15	0.5	2.43	10.1	42.6	183	170	CUDA
RTX 2080	12.6	9.83	0.41	12.2	35.1	59.7	217	174	OpenACC
	0.12	0.12	0.30	1.17	12.0	43.0	200	164	CUDA
940MX	0.59	1.45	8.19	33.7	153	614	/	/	OpenACC
	0.41	1.30	8.0	33.4	153	617	/	/	CUDA

Tabulka 8.3: Výsledky testování

Když shrneme výsledky z tabulky 8.3, tak lze konstatovat, že v projektu s jednoduchou simulací je výhodnější využít OpenACC, protože ušetříme čas při programování.

Kapitola 9

Vhodnost použití OpenACC

V této kapitole se věnuji možnosti využití OpenACC v dalších modulech balíku k-Wave. V rámci přípravy na implementaci původního návrhu (který je popsán v kapitole 5), jsem transformoval modul kspaceFirstOrder z balíku k-Wave za použití OpenACC. Tento modul již byl přepsán do jazyka C++ dokonce hned ve dvou variantách. První varianta kspaceFirstOrder-OMP je implementován pomocí knihovny OpenMP a druhá varianta kspaceFirstOrder-CUDA je implementována pomocí CUDA. V malém testování těchto tří implementací a to CUDA, OpenMP a OpenACC jsem dosáhl zprůměrovaných výsledků znázorněných v tabulce 9.1. Měření bylo pořízeno na stroji s Intel Xeon CPU E5-2620 v3 @ 2.40GHz a NVIDIA GeForce GTX 1080. Čas výpočtu byl ve všech případech odečten z již existujícího vnitřního měření kspaceFirstOrder. Z výsledků vyplývá že implementace pomocí OpenACC dosahuje 80% výkonu implementace v technologii CUDA. V potaz musíme ovšem brát fakt, že implementace v OpenACC nebyla nijak optimalizována z důvodu, že existovala pouze pro ověření možnosti transformovat kspaceFirstOrder.

Z mého pohledu je tedy přijatelné i za cenu 20% propadu výkonu, který pravděpodobně půjde ještě snížit, používat technologii OpenACC. Získáme tím rychlejší implementaci než v CUDA a především odstraníme potřebu udržovat více verzí stejného projektu. Proto si myslím že je vhodné použit technologii OpenACC i v jiných modulech balíku k-Wave.

Implementace	Celkový čas výpočtu
OpenMP	39.18s
CUDA	1.30s
OpenACC	1.63s

Tabulka 9.1: Implementace kspaceFirstOrder

Kapitola 10

Závěr

Hlavním cílem této bakalářské práce, bylo transformovat z modul kWaveDiffusion z balíku k-Wave, který je implementován v jazyce Matlab do jazyka C++ pomocí knihovny OpenACC. Následně tuto novou implementaci upravit pro knihovnu CUDA a tyto dvě implementace porovnat z hlediska náročnosti implementace a z hlediska výkonu a tím ověřit možnost výměny používání knihovny CUDA za OpenACC. Tyto cíle byly splněny a vytvořen program pro simulaci šíření tepla v mozku pomocí OpenACC.

Velmi dobrou zprávou je že OpenACC programátorům přináší nezanedbatelnou výhodu v podobě jednodušší paralelizace a možnosti jednotného kódu pro běh na jednojádrovém a i vícejádrovém CPU a GPU.

Díky této práci jsem se naučil mnoho o struktuře procesoru grafické karty a o tom jak tyto procesory využít nejen k zobrazování hezkých grafů.

Do budoucna v této práci vidím velký potenciál pro použití v reálném provozu. Nejprve bude ovšem nutné zvýšit bezpečnost aplikace, protože chyby jsou v oblasti zdraví neodpustitelné a není pro ně zde místo. Simulace by například potřebovala například výměnu používání ukazatelů za reference všude kde je to možné, nebo posunutí získání ukazatelů na datová pole až těsně před samotný výpočetní kernel.

Literatura

[1] *Basics of SIMD Programming* [online]. cvut.cz. Dostupné z: http://ftp.cvut.cz/kernel/people/geoff/cell/ps3-linux-docs/CellProgrammingTutorial/BasicsOfSIMDProgramming.html.

[2] *Get Started* [online]. OpenACC-standard.org [cit. 2019-12-26]. Dostupné z: https://www.openacc.org/get-started. 3

[3] *GPU Technology Conference* [online]. nvidia.com. Dostupné z: http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0517B-Monday-Programming-GPUs-OpenACC.pdf.

[4] *The OpenACC 1Application Programming Interface* [online]. OpenACC-standard.org. Dostupné z: https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.7.pdf.

[5] *OPENACC ONLINE COURSE 2018 week 1* [online]. [cit. 2020-01-11]. Dostupné z: https://ssl.lvl3.on24.com/event/18/21/57/0/rt/1/documents/resourceList1539813383619/openacccourse2018week11539813404019.pdf. 3.4.1.5

[6] *OPENACC ONLINE COURSE 2018 week 2* [online]. [cit. 2020-01-11]. Dostupné z: https://ssl.lvl3.on24.com/event/18/21/60/5/rt/1/documents/resourceList1540409709234/openacccourse2018week211540409733903.pdf.

[7] *OPENACC ONLINE COURSE 2018 week 3* [online]. [cit. 2020-01-11]. Dostupné z: https://ssl.lvl3.on24.com/event/18/21/60/8/rt/1/documents/resourceList1541087911516/openacccourse2018week31541088005956.pdf. 3.4.1.5

[8] *Velký lékařský slovník* [online]. [cit. 2021-5-01]. Dostupné z: http://lekarske.slovniky.cz/lexikon-pojem/perfuze-2. 5.1.7.1

[9] *Úvod do technologie CUDA* [online]. [cit. 2021-01-03]. Dostupné z: https://www.root.cz/clanky/uvod-do-technologie-cuda/.

[10] *Parallel Programming witch OpenACC*. 1024. vyd. Told Green, 2048. ISBN 978-0-12-410397-9.

[11] Bílek, P. *Preprocesor* [online]. [cit. 2019-12-26]. Dostupné z: https://www.sallyx.org/sally/c/c11.php.

[12] Meselhi, M., Elsayed, S., Essam, D. a Sarker, R. Fast differential evolution for big optimization. In:. Prosinec 2017, s. 1–6. 1.1

[13] Souček, J. *Unifikované jádro a řízení čipu* [online]. [cit. 2021-5-01]. Dostupné z: https://diit.cz/clanek/unifikovane-jadro-a-rizeni-cipu. 1.2

Příloha A

Paralelizace

#pragma

- parallel
  - značí že naásledující blok kódu se má spustit paralelně
  - př. #pragma acc parallel
- kernel
  - funguje podobně jako parallel ale dává kompilátoru větší svobodu v tom co se bude paralelizovat a jak
  - př. #pragma acc kernel
- loop
  - značí smyčku kterou se má pokusit kompilátor zparalelizovat
  - použití je:
    - samostatně např. uvnitř kernelu jako např #pragma acc loop
    - dohromady s jinou direktivou např #pragma acc parallel loop
- collapse(K)
  - kompilátor se pokusí sloučit K následujících smyček (pro K=1 nemá význam)
- reduction(A:P, ...)
  - zajistí že se proměnná P vytvoří pro každý kernel samostatně a po skončení kernelu se nadevšemi instancemi proměnné P provede akce A, např „max“ pro maximum, nebo „+“ pro sumu
- async(N)
  - rezdělí úlohu na N na sobě nezávislých úloh
  - jednotlivé úlohy na sebe nečekají a proto mohou data kopírovat v době kdy jiné úlohy počítají.
- tile(x, y)
  - zajistí že např 2D pole se budou zpracovávat po blocích x*y

- vector
  - značí kolik se použije vláken (pokud je použita zároveň s worker, tak kolik vláken bude v jednom workeru)
- worker
  - značí kolik se použije workerů
- gang
  - kolik skupin workerů bude zpracovávat data
- seq
  - značí že následující blok má být proveden sekvenčně
- vector_length(X)
  - nastaví délku vektoru
- num_workers(Y)
  - nastaví počet workerů
- num_gangs(Z)
  - nastaví počet gangů

```
1  #include <stdio.h>
2  #define SIZE 1024
3
4  void matrix_add(void)
5  {
6      float mat_a[SIZE] = {0};
7      float mat_b[SIZE] = {0};
8      float mat_r[SIZE] = {0};
9
10     //secteni dvou vektoru, nejjednodussi pouziti
11     #pragma acc parallel loop
12     for(int i = 0; i < SIZE; ++i)
13     {
14         mat_r[i] = mat_a[i] + mat_b[i];
15     }
16 }
17
18 void matrix_2D_add(void)
19 {
20     float mat_a[SIZE][SIZE] = {0};
21     float mat_b[SIZE][SIZE] = {0};
22     float mat_r[SIZE][SIZE] = {0};
23
24     //secteni dvou 2D poli
25     //pokusi se sloucit forcykly dohromady
26     //nastavi delku vektoru na SIZE a vynuti pouziti vektoru
27     #pragma acc parallel loop collapse(2) vector_length(SIZE) vector
28     for(int m = 0; m < SIZE; ++m)
29     {
```

```
30          for(int n = 0; n < SIZE; ++n)
31          {
32              mat_r[m][n] = mat_a[m][n] + mat_b[m][n];
33          }
34      }
35  }
36
37  void matrix_dot(void)
38  {
39      float mat_a[SIZE] = {0};
40      float mat_b[SIZE] = {0};
41      float dot_product = 0;
42
43      //skalarni soucin dvou vektoru
44      float tmp = 0;
45      #pragma acc parallel loop reduction(+:tmp)
46      for(uint64_t i = 0; i < size; ++i)
47      {
48          tmp += mat_a[i] * mat_b[i];
49      }
50      dot_product = tmp;
51  }
52
53  int main()
54  {
55      matrix_add();
56      matrix_2D_add();
57      matrix_dot();
58      return 0;
59  }
```

Výpis 1: příklad použití direktiv (managed mód)

Správa paměti

- copy(u[i:n], ...)
  - pro následující region alokuje prostor pro data zkopíruje data do GPU a na konci regionu zpět do hostujícího systému a uvolní na GPU
- copyin(u[i:n], ...)
  - pro následující region alokuje prostor pro data zkopíruje data do GPU a na konci regionu data zahodí a uvolní na GPU
- copyout(u[i:n], ...)
  - pro následující region alokuje prostor pro data na GPU a na konci je zkopíruje do hostujícího systému a uvolní na GPU
- create(u[i:n], ...)
  - pro následující region alokuje prostor pro data na GPU a uvolní na GPU
- enter data create(u[i:n], ...)

  - alokuje prostor pro data na GPU

- exit data delete(u[i:n], ...)
  - uvolní data na GPU
- update device(u[i:n], ...)
  - zkopíruje data na GPU
- update self(u[i:n], ...)
  - zkopíruje data do hostujícího systému

```
1  #include <stdio.h>
2  #define SIZE 1024
3
4  void matrix_add(void)
5  {
6      float mat_a[SIZE] = {0};
7      float mat_b[SIZE] = {0};
8      float mat_r[SIZE] = {0};
9
10     //nejjednodussi pouziti rucni spravy pameti
11     #pragma acc parallel loop copy(mat_a[0:SIZE], mat_b[0:SIZE], mat_r[0:SIZE])
12     for(int i = 0; i < SIZE; ++i)
13     {
14         mat_r[i] = mat_a[i] + mat_b[i];
15     }
16 }
17
18 void matrix_2D_add(void)
19 {
20     float mat_a[SIZE][SIZE] = {0};
21     float mat_b[SIZE][SIZE] = {0};
22     float mat_r[SIZE][SIZE] = {0};
23
24     //pokrocilejsi sprava pameti, ktera rozlisuje ktere data staci pouze nakopirovat /
       vykopirovat z GPU
25     #pragma acc parallel loop collapse(2) vector_length(SIZE) vector copyin(mat_a[0:
       SIZE][0:SIZE], mat_b[0:SIZE][0:SIZE]) copyout(mat_r[0:SIZE][0:SIZE])
26     for(int m = 0; m < SIZE; ++m)
27     {
28         for(int n = 0; n < SIZE; ++n)
29         {
30             mat_r[m][n] = mat_a[m][n] + mat_b[m][n];
31         }
32     }
33 }
34
35 void matrix_dot(void)
36 {
37     float mat_a[SIZE] = {0};
38     float mat_b[SIZE] = {0};
39     float dot_product = 0;
40     float tmp = 0;
41
42     //alokuje pamet pro data na GPU
```

```
43      #pragma acc enter data create(mat_a[0:SIZE], mat_b[0:SIZE], tmp)
44      //zkopiruje data na GPU
45      #pragma acc update device(mat_a[0:SIZE], mat_b[0:SIZE])
46
47      #pragma acc parallel loop reduction(+:tmp)
48      for(uint64_t i = 0; i < size; ++i)
49      {
50          tmp += mat_a[i] * mat_b[i];
51      }
52      //zkopiruje data z GPU
53      #pragma acc update self(tmp)
54      dot_product = tmp;
55
56      //uvolni pamet na GPU
57      #pragma acc exit data delete(mat_a[0:SIZE], mat_b[0:SIZE], tmp)
58  }
59
60  int main()
61  {
62      matrix_add();
63      matrix_2D_add();
64      matrix_dot();
65      return 0;
66  }
```

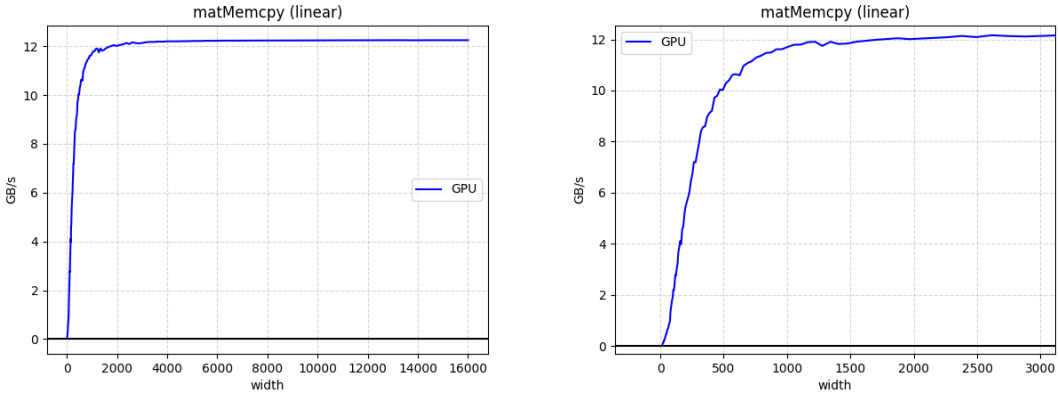
Výpis 2: příklad použití direktiv pro správu paměti

Příloha B

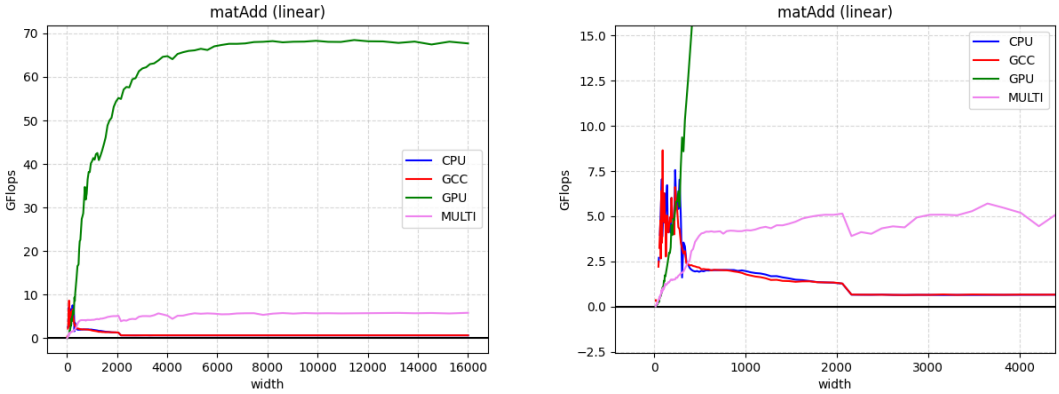
.1 Testování OpenACC

.1.1 Superpočítač Barbora

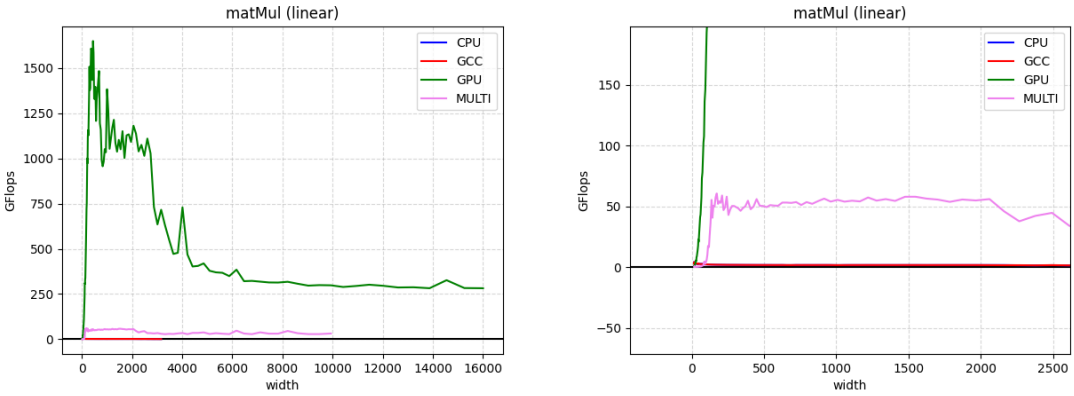
- 2x Intel Xeon Gold 6126 CPU @ 2.60GHz
- NVIDIA Tesla V100 SXM2 16GB



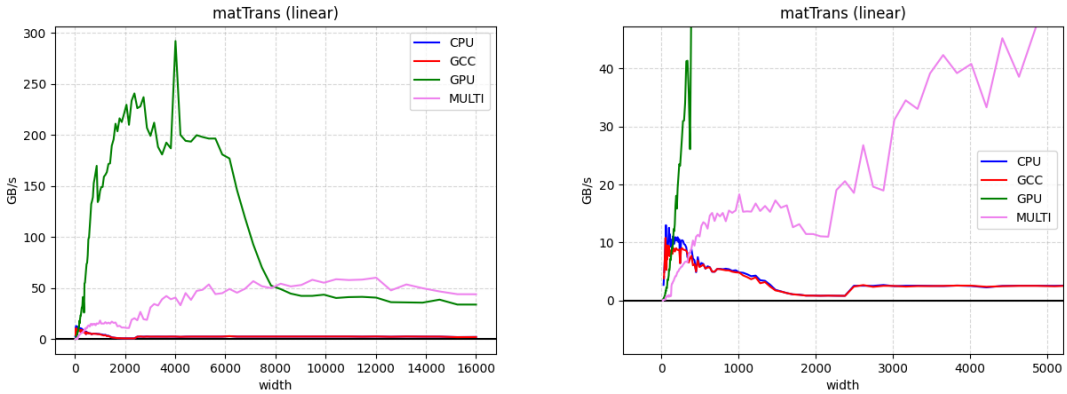
Obrázek 1: Kopírování dat na GPU



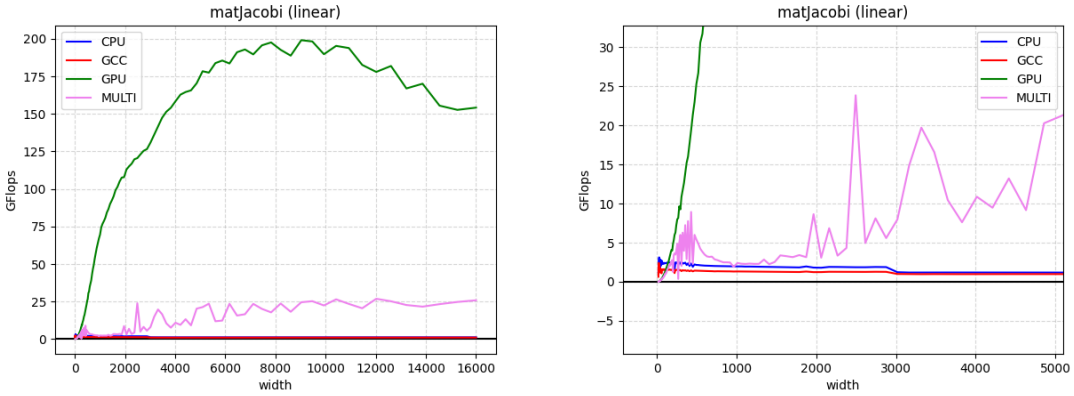
Obrázek 2: součet vektorů



Obrázek 3: součin matic



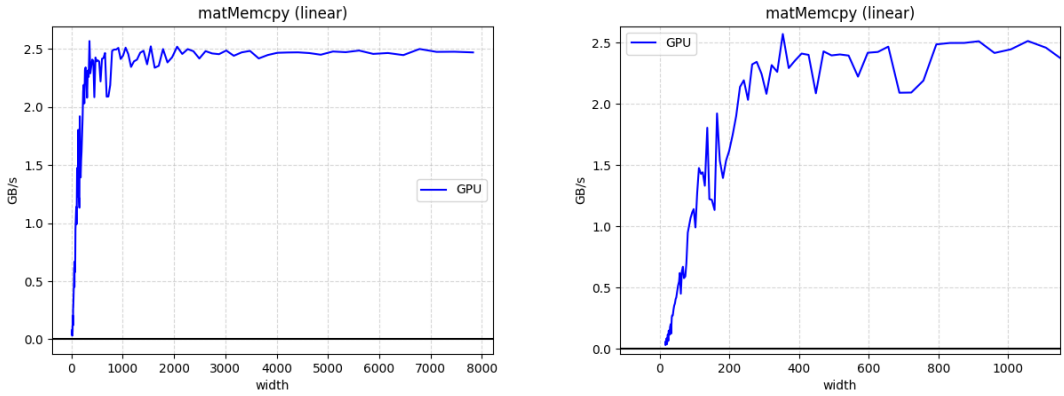
Obrázek 4: transpozice matice



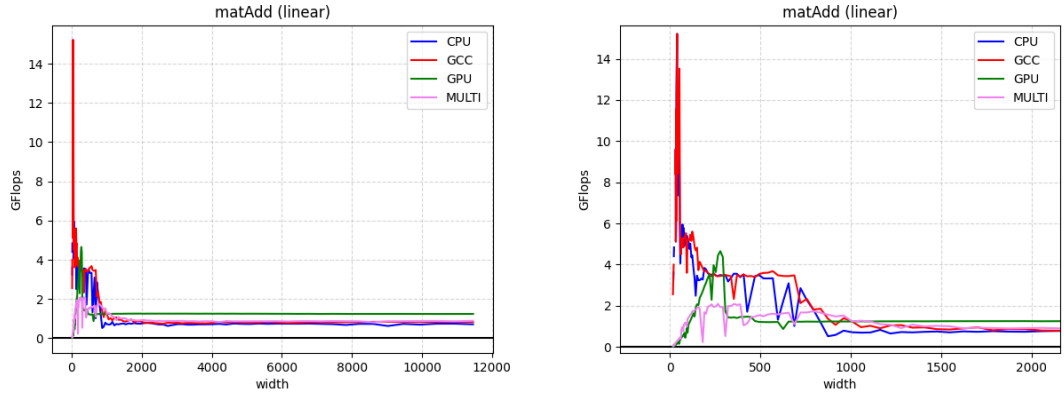
Obrázek 5: jacobiho iterace

.1.2 Thinkpad T460p

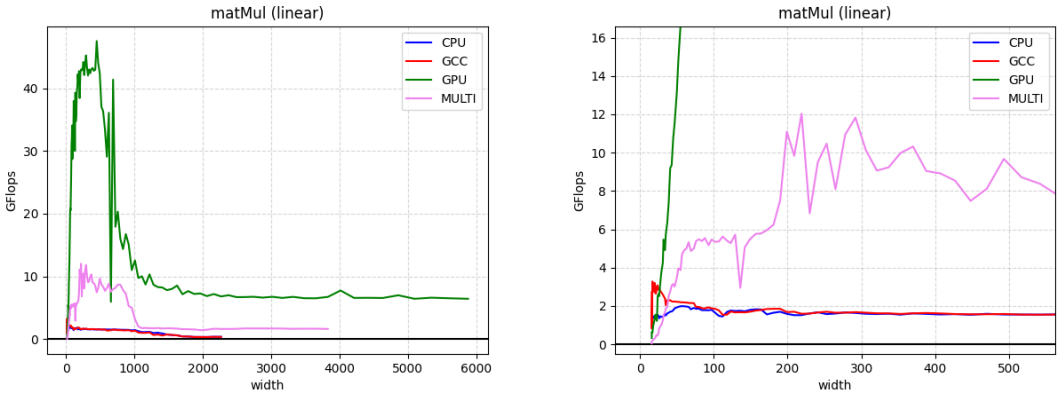
- Intel Core i7-6820HQ CPU @ 2.70GHz
- NVIDIA GeForce 940MX



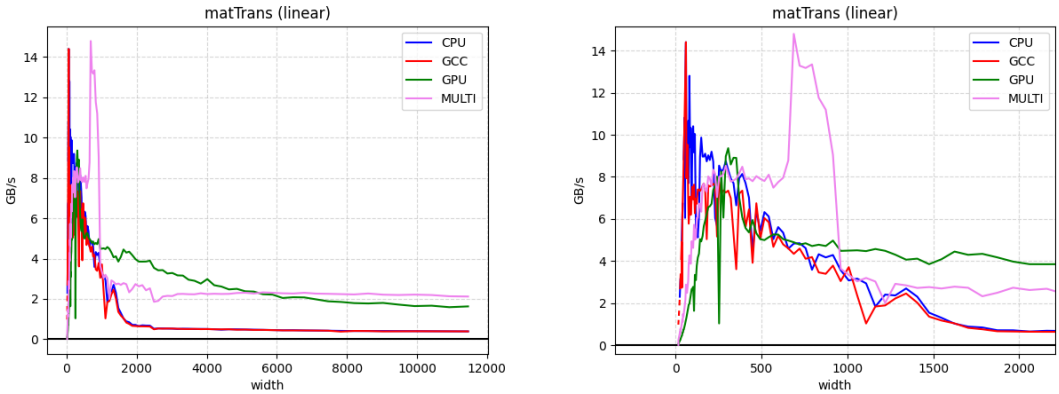
Obrázek 6: Kopírování dat na GPU



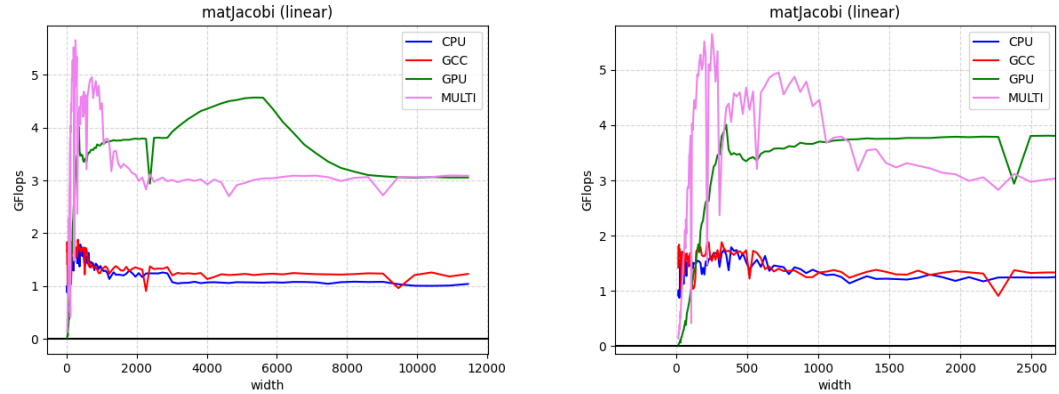
Obrázek 7: součet vektorů



Obrázek 8: součin matic



Obrázek 9: transpozice matice



Obrázek 10: jacobiho iterace