VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

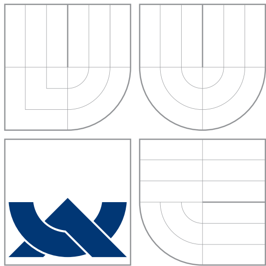
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

NUMERICKÁ SIMULACE ŠÍŘENÍ TEPLA S VYUŽITÍM GPU

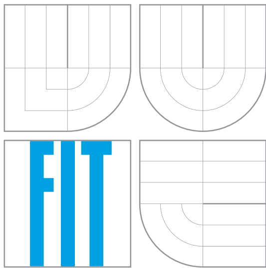
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE MICHAL HRADECKÝ
AUTHOR

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

NUMERICKÁ SIMULACE ŠÍŘENÍ TEPLA S VYUŽITÍM GPU

HEAT DIFFUSION SIMULATION ON GPU

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MICHAL HRADECKÝ
AUTHOR

VEDOUCÍ PRÁCE — Ing. JIŘÍ JAROŠ, Ph.D.
SUPERVISOR

BRNO 2013

Abstrakt

Tato práce se zabývá numerickou simulací šíření tepla v lidských tkáních. Navržený algoritmus pracuje s metodou konečných diferencí v časové doméně (FDTD), kterou aplikuje na řídící rovnici popisující tento systém. Pro implementaci je využito moderní grafické karty, jejíž výkon je porovnán s efektivní implementací na vícejádrovém procesoru. Výstupem práce je sada několika rozdílně optimalizovaných algoritmů pro grafické karty firmy NVIDIA. Experimentální výsledky ukazují, že využití sdílené paměti je v tomto případě kontraproduktivní a nejlepšího výsledku dosáhl algoritmus založený na registrech. Celkové zrychlení dosažené pomocí karty NVIDIA GeForce GTX 580 vzhledem k 4 jádrovému procesoru Intel Core i7 920 činí 18,5, což koresponduje s teoretickými možnostmi obou architektur.

Abstract

This thesis deals with the simulation of heat diffusion in human tissues. The proposed algorithm uses a finite-difference time-domain method, which is applied on the governing equation describing the system. A modern graphics card is used to accelerate the simulation. The performance achieved on the GPU card is compared with the implementation exploiting a modern multicore CPU. The output of this thesis is a set of differently optimized algorithms targeted on NVIDIA graphics cards. The experimental results reveal that the use of shared memory is contraproductive and the best performance is achieved by a register based implementation. The overall speedup of 18.5 was reached when comparing a NVIDIA GeForce GTX 580 with a quad-core Intel Core i7 920 CPU. This nicely corresponds with the theoretical capabilities of both architectures.

Klíčová slova

simulace šíření tepla, metoda konečných diferencí, CUDA, GPGPU, akcelerace na GPU, vícejádrové procesory

Keywords

heat distribution simulation, FDTD method, CUDA, GPGPU, GPU acceleration, Multicore CPU

Citace

Michal Hradecký: Numerická simulace šíření tepla
s využitím GPU, bakalářská práce, Brno, FIT VUT v Brně, 2013

Numerická simulace šíření tepla s využitím GPU

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Jiřího Jaroše, Ph.D.

....
Michal Hradecký
15. května 2013

Poděkování

Chtěl bych poděkovat vedoucímu své práce Ing. Jiřímu Jarošovi, Ph.D. za cenné rady a čas, který mi během tvorby práce věnoval. Dále bych chtěl poděkovat své přítelkyni za podporu a trpělivost, kterou se mnou při tvorbě práce měla.

© Michal Hradecký, 2013.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Simulace šíření tepla **4**
2.1 Metody konečných prvků a konečných diferencí 5
2.1.1 Historie 5
2.1.2 Metoda konečných prvků 5
2.1.3 Metoda konečných diferencí v časové doméně 5
2.2 Přesnost a výběr řádu 6
2.3 Diskretizace 6
2.3.1 Okrajové podmínky 6
2.4 Odvození výsledného vztahu 6

3 Použití grafických karet pro obecné výpočty **8**
3.1 Rozdíly mezi GPU a CPU 8
3.2 Knihovny pro GPGPU 9
3.3 CUDA 10
3.3.1 Výpočetní model 10
3.3.2 Mapování výpočetního modelu na hardware 12
3.3.3 Paměťový model 12

4 Implementace **15**
4.1 Popis algoritmu 15
4.2 Vstupní data 17
4.3 Složitost algoritmu 17
4.4 Validita algoritmu 17
4.5 Měření výkonnosti algoritmu 17
4.6 Implementace na procesoru 18
4.6.1 Specifikace procesoru 18
4.6.2 Měření výkonnosti algoritmu na procesoru 18
4.6.3 Naivní algoritmus pro šíření tepla 19
4.6.4 Paměťová lokalita 19
4.6.5 Verze s efektivním využitím cache pamětí 19
4.6.6 SSE verze algoritmu 21
4.6.7 Paralelní verze s OpenMP 22
4.7 Implementace na grafické kartě 23
4.7.1 Parametry karty 23
4.7.2 Single / double precision 24
4.7.3 Rozměry matic se vstupními daty 24

4.7.4 Implementace algoritmu 24
4.7.5 Naivní kernel 25
4.7.6 Kernel pro 4 krychle 8x8x8 26
4.7.7 Kernel s dlaždicemi 16x16 28
4.7.8 Kernel s obdélníkovým blokem 28
4.7.9 Kernel bez sdílené paměti 30
4.7.10 Kernel s generovanými okraji 31
4.8 Výsledky implementací 33

5 Závěr 34

A Návod na zprovoznění aplikace 37
A.1 Překlad aplikace 37
A.2 Spuštění aplikace 37

Kapitola 1

Úvod

Od počátku rozvoje počítačů pomáhají numerické simulace v řešení rozličných problémů, u kterých nelze spočítat exaktní řešení. Používají se ve vědeckých i průmyslových projektech. Příkladem mohou být simulace atmosférických jevů k předpovědi počasí nebo modelování deformace automobilu.

Simulace šíření tepla nachází upotřebení v mnoha odvětvích lidské činnosti. V technických oborech můžeme například simulacemi efektivně navrhovat ideální tvary chladičů nebo modelovat zateplení domů různými materiály, aniž bychom je museli reálně postavit. Jednou z oblastí využití je i plánování léčby rakovinných nádorů pomocí zacíleného ultrazvuku o vysoké intenzitě neboli HIFU, prováděného na Australské národní univerzitě[1].

Tento typ léčby využívá vlastností mechanických ultrazvukových vln, které při průchodu tkání zanechávají teplo. Ultrazvukový paprsek se zacílí do místa nádoru, kam je vysílán do té doby, než je tkáň v místě nádoru vystavena takové intenzitě tepla, že rakovinné buňky odumřou, a nastává nekróza [17].

Čas působení ultrazvuku a jeho intenzita musí být takové, aby byly dostatečné ke zničení nádoru a zároveň zbytečně nepoškozovaly tkáň v jeho okolí. Tyto hodnoty jsou typické pro každého pacienta a typ tkáně.

Simulace takového systému není triviální. Výpočet s využitím běžného procesoru by zabral příliš velké množství času a výsledky by již nemusely být relevantní (stav v těle či poloha pacienta se změní). Proto se nabízí využití moderních grafických karet, s jejichž pomocí je možné několikanásobně urychlit celý proces simulace. Velká část práce je tedy věnována technologii CUDA a implementaci algoritmů na ní.

Cílem této práce je akcelerace jedné části ultrazvukové simulace, která má na starosti simulovat šíření tepla v dané doméně pomocí metody konečných diferencí. Kód, který v této práci vznikl, je lehce aplikovatelný na spoustu podobných problémů. Toto téma jsem si vybral, protože mě zajímá potenciál, který grafické karty nabízejí, a chtěl jsem se s ním naučit pracovat.

[1] http://www.anu.edu.au/

Kapitola 2

Simulace šíření tepla

Tato kapitola popisuje a vymezuje fyzikální podstatu řešeného problému. Lidské tělo nelze považovat za homogenní prostředí, různé části těla mají různé fyzikální vlastnosti, které se mění s rostoucí či klesající teplotou. Jedná se tedy o simulaci ve 3D heterogenním prostředí. Při výpočtech vycházím z diskrétní verze rovnice *Pennes bioheat transfer equation*[20], která definuje výpočet změny teploty v lidské tkáni za určitý čas a vypadá takto:

$$\rho_t C_t \frac{\delta T(x,t)}{\delta t} = k_t \nabla^2 T(x,t) + V \rho_b C_b (T_b - T(x,t)) + Q(x,t) \tag{2.1}$$

- $T = T(x,t)$ - teplota v bodě x v čase t
- k_t neboli λ - součinitel tepelné vodivosti tkáně (anglicky conductivity)
- C_t - měrná tepelná kapacita tkáně (anglicky specific heat capacity)
- C_b - měrná tepelná kapacita krve
- ρ_t - hustota tkáně
- ρ_b - hustota krve
- V - rychlost průtoku krve (anglicky perfusion rate)
- $Q(x,t)$ - teplo dodané ultrazvukem bodu x

Tuto rovnici můžeme též schematicky zapsat takto:

$$T_{nová} = T_{aproximace-okolí} + T_{tok-krve} + T_{přidané-teplo} \tag{2.2}$$

- $T_{aproximace-okolí}$ - šíření tepla mezi buňkami z teplejších oblastí do chladnějších
- $T_{tok-krve}$ - odvod tepla krví proudící v tepnách
- $T_{přidané-teplo}$ - teplo dodané do těla pomocí ultrazvukové vlny

Mým úkolem v této práci je akcelerovat výpočet první části této schématické rovnice $T_{aproximace-okolí}$.

Proces šíření tepla tepnami není tak důležitý a není v simulaci obsažen, protože bylo dokázáno, že proudění krve nemá velký vliv na odvod tepla ze systému[17]. Část $T_{přidané-teplo}$,

tedy teplo dodávané pomocí ultrazvuku, $Q(x,t)$, může být dodáno zvlášť po každé iteraci algoritmu, proto s ním v algoritmu nepočítám.

Zbývá tedy tato rovnice

$$\rho_t C_t \frac{\delta T(x,t)}{\delta t} = k_t \nabla^2 T(x,t) \tag{2.3}$$

Z ní si vyjádříme Laplaceův operátor ∇^2, který vyjadřuje sumu druhých derivací teploty podle jednotlivých os.

$$\nabla^2 T(x,t) = \frac{\delta^2 T(x,t)}{\delta x^2} + \frac{\delta^2 T(x,t)}{\delta y^2} + \frac{\delta^2 T(x,t)}{\delta z^2} \tag{2.4}$$

Derivace 2. řádu aproximujeme některou z metod, popsaných v následují sekci.

2.1 Metody konečných prvků a konečných diferencí

Metody konečných prvků a konečných diferencí se používají pro řešení systému, který vede na parciální diferenciální rovnice, což je právě v tomto případě. Jsou to numerické metody, tedy neposkytují přesné řešení problému, ale pouze řešení přibližné. Jak přesné toto řešení bude, závisí na volbě metody a jejích parametrů.

2.1.1 Historie

Teoretické základy těchto metod byly položeny již za 2. světové války. Čistě o metodě konečných prvků mluvíme od roku 1960, kdy Clough publikoval práci, ve které toto spojení použil. Od 60. letech 20. století se začaly metoda rozšiřovat, protože začaly být dostupné výpočetní prostředky pro její použití [3].

2.1.2 Metoda konečných prvků

Metoda v angličtině nazývaná Finite elements method, zkráceně FEM. Funguje tak, že si nejprve rozdělíme systém (1D,2D,3D) na konečný počet prvků (podoblastí, např. trojúhelníky ve 2D), které mohou mít navzájem jiné vlastnosti a mohou být jinak velké (dobře se tímto simulují heterogenní materiály), síť prvků nemusí být rovnoměrná. Mezi prvky nejsou žádné mezery, prvky (podoblasti) jsou spojeny v uzlových bodech a dají dohromady zpět celou oblast. Neznámou veličinu aproximujeme pomocí funkcí v uzlových bodech. Poté dosadíme do rovnic a soustavy vyřešíme [12].

2.1.3 Metoda konečných diferencí v časové doméně

Metoda v angličtině nazývaná Finite-difference time-domain method, zkráceně FDTD. Tato metoda funguje tak, že systém rozdělíme sítí s rovnoměrným (v každé dimenzi může být různým) krokem. Následně parciální derivace nahradíme ve všech bodech rozdíly (diferencemi) neznámých hodnot [1]. Metoda je jednodušší na implementaci a také se díky pravidelné síti dobře paralelizuje .

V tomto případě je lepší využít metodu konečných diferencí, protože pro efektivní zpracování dat v GPU je dobré mít data uložená v rovnoměrně rozložené matici. Přístup do paměti je potom pravidelný, což je při práci s maticemi v GPU velmi žádané.

2.2 Přesnost a výběr řádu

Jak již ze zaměření práce na akceleraci vyplývá, chceme mít výsledek vypočítaný co nejrychleji a s co nejmenšími nároky na paměť a výkon. Ovšem nepřesný či špatný výsledek, který poskytne algoritmus, který bude po několika stovkách iterací nepřesný, může mít zvláště v oblasti medicíny nedozírné následky. Proto je nutné určit optimální poměr mezi přesností a náročností, kterou volba algoritmu a jeho přesnosti přinese.

Potřebujeme tedy zvolit řád použité metody. Pokud již dopředu budeme uvažovat podmínky výpočtu na grafických kartách, tak například pro výpočet jednoho bloku nových hodnot metodou konečných diferencí 8. řádu bychom potřebovali navíc okolí 4 bodů do všech dimenzí. To při rozměrech bloku 16x16x16=4096 bodů dělá celkem 16x16x24=6144 bodů navíc. Byla tedy zvolena metoda konečných diferencí 4. řádu, kde je toto množství poloviční a z hlediska přesnosti je stále dostatečná. Tento řád byl také vybrát z důvodu předchozího použití v práci [17], která je součástí celého výzkumu.

2.3 Diskretizace

Na doporučení vedoucího byly pro diskretizaci úlohy zvoleny následující hodnoty:

- $\Delta z = \Delta y = \Delta z = 1mm$
- $\Delta t = 100\mu s$

Tyto hodnoty specifikují, jak je spojitá doména teploty ve tkáni rozdělena do diskretizované mřížky po krocích Δz, Δy a Δz v patřičných dimenzích. Časový krok Δt určuje, po jak dlouhém časovém intervalu se bude určovat nová hodnota teploty.

2.3.1 Okrajové podmínky

Jako okrajová podmínka je využita Dirichletova okrajová podmínka [16], která předepisuje hodnoty funkce v okrajových bodech. V tomto modelu je počítáno s tím, že teplota tkáně na okraji modelu zůstává neměnná, 37°C, jako je teplota lidského těla.

2.4 Odvození výsledného vztahu

Podle zvoleného 4. řádu metody konečných prvků můžeme zapsat 2. derivaci funkce f s krokem h v bodě x pomocí centrálních diferencí takto:

$$f'' \approx \frac{f\left(x-2h\right) - 16f\left(x-h\right) + 30f\left(x\right) - 16f\left(x+h\right) + f\left(x+2h\right)}{12h} \tag{2.5}$$

Takto převedeme diferenciální rovnice pro všechny dimenze x, y a z. Je potřeba od sebe odlišovat bod $T\left(x,t\right)$ a krok v ose x, Δx, každé x znamená něco jiného. Převedená rovnice pro osu x vypadá následovně:

$$\frac{\delta^2 T\left(x,t\right)}{\delta x^2} \approx \frac{\left(T^n_{i-2,j,k} - 16T^n_{i-1,j,k} + 30T^n_{i,j,k} - 16T^n_{i+1,j,k} + T^n_{i+2,j,k}\right)}{12(\Delta x)^2} \tag{2.6}$$

Pokud rovnice pro všechny osy sečteme a z každé vytkneme $\frac{1}{12}$, tak dostaneme toto vyjádření Laplaceova operátoru.

$$\nabla^2 T\left(x,t\right) \approx \frac{1}{12}\left(\frac{\left(T^n_{i-2,j,k} - 16T^n_{i-1,j,k} + 30T^n_{i,j,k} - 16T^n_{i+1,j,k} + T^n_{i+2,j,k}\right)}{(\Delta x)^2} + \frac{\left(T^n_{i,j-2,k} - 16T^n_{i,j-1,k} + 30T^n_{i,j,k} - 16T^n_{i,j+1,k} + T^n_{i,j+2,k}\right)}{(\Delta y)^2} + \frac{\left(T^n_{i,j,k-2} - 16T^n_{i,j,k-1} + 30T^n_{i,j,k} - 16T^n_{i,j,k+1} + T^n_{i,j,k+2}\right)}{(\Delta z)^2}\right) \tag{2.7}$$

Dále můžeme v rovnici 2.3 nahradit derivaci teploty podle času za:

$$\frac{\delta T\left(x,t\right)}{\delta t} \approx \frac{T^{n+1}_{i,j,k} - T^n_{i,j,k}}{\Delta t} \tag{2.8}$$

Potom ještě sjednotíme fyzikální vlastnosti bodu do jedné proměnné β.

$$\beta = \frac{\lambda}{\rho C_t} \tag{2.9}$$

A nakonec z upravené rovnice 2.3 vytkneme $T^{n+1}_{i,j,k}$, čímž dostaneme výsledný vztah.

$$T^{n+1}_{i,j,k} = T^n_{i,j,k} + \beta \Delta t \frac{1}{12}\left(\frac{\left(T^n_{i-2,j,k} - 16T^n_{i-1,j,k} + 30T^n_{i,j,k} - 16T^n_{i+1,j,k} + T^n_{i+2,j,k}\right)}{(\Delta x)^2} + \frac{\left(T^n_{i,j-2,k} - 16T^n_{i,j-1,k} + 30T^n_{i,j,k} - 16T^n_{i,j+1,k} + T^n_{i,j+2,k}\right)}{(\Delta y)^2} + \frac{\left(T^n_{i,j,k-2} - 16T^n_{i,j,k-1} + 30T^n_{i,j,k} - 16T^n_{i,j,k+1} + T^n_{i,j,k+2}\right)}{(\Delta z)^2}\right) \tag{2.10}$$

Kapitola 3

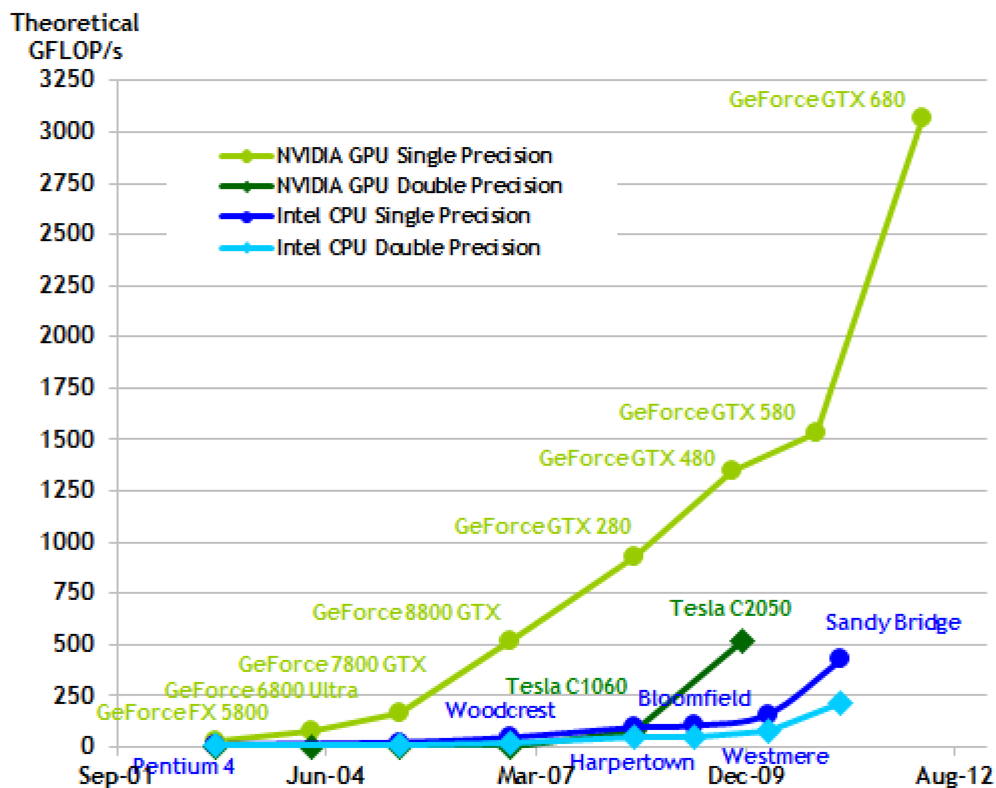
Použití grafických karet pro obecné výpočty

Původně měly grafické karty sloužit pouze ke zpracování a posílání obrazu do zobrazovacího zařízení. V posledních letech se ale ukazuje, že grafické karty mají velký potenciál při paralelním zpracování velkého množství stejných dat a jsou využívány k akceleraci různorodých algoritmů a simulací z oblasti fyziky, bioinformatiky a genetických algoritmů. Tomuto použití grafických karet pro obecné výpočty se říká GPGPU.

3.1 Rozdíly mezi GPU a CPU

Přestože moderní procesory mají dnes až 16 jader [2], paralelní zpracování dat na nich není tak efektivní jako na GPU. Je to proto, že jádra na CPU a GPU jsou orientována na zpracování jiného typu dat. Jádro v CPU je zaměřeno na efektivní sekvenční zpracování jednoho vlákna a jedna instrukce zpracovává většinou jen jedna data (výjimkou jsou vektorová rozšíření SSE nebo AVX). Tomuto principu se říká SISD - Single instruction - single data. CPU jádro je velmi složité, implementuje složité algoritmy predikce skoků, samo si řídí efektivní využívání cache pamětí, dokáže si měnit pořadí zpracovávaných instrukcí. V jednom procesoru je takovýchto jader velmi omezené množství. V současné době je to nejvíce 16 jader u AMD Opteron procesorů [2]. Velký rozdíl je také ve velikosti cache pamětí, které jsou v CPU zastoupeny v řádu několika MB a rozděleny do několika úrovní, přičemž ty nejvyšší jsou většinou společné pro celé CPU.

Naproti tomu jedno jádro v GPU je velmi jednoduché a neumí tyto složité algoritmy pro predikci skoků a nemá zabudovanou tak silnou ochranu paměti (proti přístupu mimo hranice pole). Díky absenci této režie může plocha čipu obsahovat více výpočetních jednotek, z čehož GPU profitují. V moderních GPU je takovýchto jader až několik stovek, což dává ve výsledku obrovský výkon jak je vidět v Obrázku 3.1. GPU se zaměřuje se na efektivní paralelní zpracování velkého množství dat, a provádění operací nad nimi - zejména těch v plovoucí řádové čárce. Využívá principu SIMT - Single instruction - multiple threads [13], kdy několik vláken vykonává v jeden okamžik tu samou instrukci. Vytvoření jednoho GPU vlákna a přepínání kontextu je téměř okamžité [11].



Obrázek 3.1: Srovnání teoretického výkonu CPU a GPU v GFLOP/s [13]

3.2 Knihovny pro GPGPU

CUDA je API pro psaní aplikací na GPU, vyvíjené společností NVIDIA od roku 2006. Je určeno k využití GPU pro obecné výpočty, ovšem pouze pro grafické karty NVIDIA, na kartách s čipy od ostatních výrobců neběží, k otestování funkčnosti vytvářené aplikace je tedy nutnost mít přístup ke kartě NVIDIA. Protože NVIDIA byla první, kdo s tímto řešením přišel, existuje pro CUDA větší množství knihoven jako jsou cuBLAS či Thrust, které poskytují vývojářům již optimalizované funkce pro práci s vektory a maticemi. Tyto knihovny má OpenCL také, ale není jich takové množství, což se může do budoucnosti změnit. NVIDIA poskytuje velmi kvalitní nástroje pro ladění a optimalizaci algoritmů psaných právě pro CUDA.

OpenCL je otevřené standardizované API založené na jazyce C, které se zaměřuje obecně na programování paralelních systémů. Je od roku 2008 spravováno konsorciem Khronos[1], ve kterém je zastoupena většina výrobců moderních procesorů a grafických karet jako jsou AMD, Intel, NVIDIA, Apple, ARM, Xilinx. Je multiplatformní a může být implementováno nejen v grafických kartách, ale i v procesorech a dalších zařízeních. Vydání specializovaných nástrojů pro ladění a vývoj aplikací u OpenCL ovšem záleží na výrobci konkrétního zařízení[11].

DirectCompute je proprietární technologií společnosti Microsoft. Hardwarově není nijak vázaná, ale ke svému běhu potřebuje prostředí systému Windows, přesněji rozhraní DirectX,

[1] https://www.khronos.org/opencl/

kterého je součástí [15]. Proto byla tato technologie zavrhnuta hned ze začátku a dále se o ní nebudu zmiňovat.

zhodnocení Co se týká stylu psaní algoritmu tak jsou na tom CUDA i OpenCL podobně. Protože v případě této práce se jedná o vyladění pro jednu konkrétní grafickou kartu, tak není třeba brát v úvahu výhodu OpenCL - přenositelnost. Navíc i ostatní části celého projektu jsou psány pro CUDA, tak je využito CUDA.

3.3 CUDA

Veškeré informace uvedené v této práci platí pro CUDA Toolkit 5.0[2], který byl v době psaní aktuální, v následujících a předchozích verzích se některé informace mohou mírně lišit.

Většina informací pro tuto sekci jsem čerpal z knih CUDA application design and development [6], Programming massively parallel processors[11] a z příručky C CUDA Programming Guide [13].

Psát programy pro CUDA lze v několika programovacích jazycích, z nichž budu jmenovat ty nejvýznamnější: CUDA C/C++, FORTRAN, Java nebo Python. CUDA C/C++ využívá ke svému zápisu rozšíření jazyka C/C++. Překlad probíhá pomocí proprietárního překladače `nvcc` od firmy NVIDIA, jehož zkompilovaný kód lze následně připojit k programům psaným v C/C++ [13]. CUDA for FORTRAN je používána pro vědecké výpočty a pyCUDA obsahuje API pro práci s CUDA v Pythonu, čímž může výrazně urychlit a ulehčit práci.

3.3.1 Výpočetní model

kernely Kernel je speciální funkce, která může běžet pouze na GPU. Při volání se mu předává, jaká má být jeho struktura v GPU, kolik má obsahovat bloků a vláken a jak mají být široké v jednotlivých dimenzích (ukázka v Algoritmu 1), popř. další parametry stejně jako u funkcí jako v C/C++. Zároveň může na jednom GPU běžet i více kernelů, což je označováno jako *kernel concurency*.

bloky a vlákna Všechna vlákna jednoho kernelu vykonávají stejný kód. Aby mohla vlákna vykonávat stejný kód, ale každé nad rozdílnými daty, je potřeba získat pro každé vlákno unikátní označení - index vlákna. A protože jsou zpracovávaná data nejčastěji uložena v maticích, je tomu přizpůsobeno i označení vláken (1D, 2D i 3D matice). V CUDA jsou 2 úrovně označení. Prvním z nich je dělení na bloky a druhé je dělení na vlákna.

Každý kernel si můžeme rozdělit podle potřeby do bloků, a to v 1D, 2D nebo 3D. Typicky se bloky používají k rozdělení dat do menších kusů, která se poté dobře vlezou do pomocných pamětí, které jsou velmi rychlé, ale jejich velikost je značně omezená. V každém vláknu potom můžeme zjistit, v jakém bloku se vlákno nachází pomocí struktury `blockIdx.x` (popř. v 2D i `blockIdx.y` a v 3D `blockIdx.z`), dostupné z každého vlákna. Rozměry jednotlivých dimenzí bloku jsou jedním ze vstupních parametrů při spouštění kernelu. Mřížce, kterou všechny bloky z jednoho kernelu dohromady tvoří, se říká grid, jak je znázorněno na Obrázku 3.2.

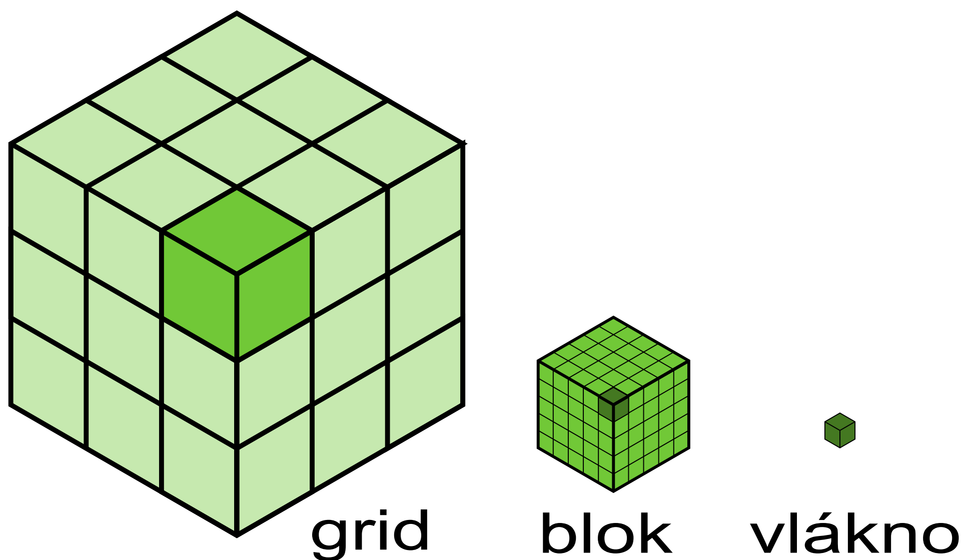
Druhou, nižší, úrovní je dělení do vláken, která mohou mít stejnou strukturu jako bloky (1D, 2D nebo 3D). K označení vlákna se přistupuje podobně jako u bloků, hodnoty jsou

[2]https://developer.nvidia.com/cuda-toolkit

Algoritmus 1: Ukázka definice a volání kernelu

```
1  __global__ void Kernel(float *A, float *B)
2  {
3      ...
4  }
5  int main()
6  {
7      ...
8      // počet bloků v gridu v osách (x,y,z)
9      dim3 dimGrid(8,32,32);
10     // počet vláken v bloku v osách (x,y,z)
11     dim3 dimBlock(32,8,1);
12     // spuštění kernelu
13     Kernel<<<dimGrid, dimBlock>>>(A, B);
14     ...
15 }
```

uloženy ve struktuře `threadIdx.x` (popř. `threadIdx.y`, `threadIdx.z`). Toto nám umožňuje bez velkého úsilí přiřadit vláknu unikátní data v matici, což ukázáno v Algoritmu 2. Rozměry jednotlivých dimenzí bloku jsou opět jedním ze vstupních parametrů při spouštění kernelu, jak je vidět v Algoritmu 1.



Obrázek 3.2: Rozdělení gridu na bloky a vlákna

Algoritmus 2: Ukázka získání jednoho bodu z 1D matice, každé vlákno dostane jiný

```
1 float bod = matice[blockIdx.x*blockDimx.x + threadIdx.x];
```

3.3.2 Mapování výpočetního modelu na hardware

Architektura grafické karty je rozdělena do několika částí nazývaných *stream multiprocessor*, nebo jen zkráceně SM. Těchto jednotek je v moderních kartách architektury Fermi nebo Kerpler až několik desítek. Při běhu programu jsou na tyto stream multiprocesory přidělovány jednotlivé bloky a to tak, že 1 blok je vždy celý přidělen na určitý multiprocessor. Na jednom multiprocessoru může být zároveň přidělen i větší počet bloků, záleží na počtu hlavně na počtu vláken a velikosti sdílené paměti (viz dále), kterou daný blok potřebuje k běhu.

Každý stream multiprocessor se skládá z několika dalších částí[19]:

1. výpočetních CUDA jader
2. sdílené paměti
3. registrů, které se rovnoměrně rozdělí vláknům
4. L1 cache globální paměti
5. plánovače warpů

Na výpočetních jádrech je vždy vykonávána v jednom čase stejná instrukce. Pokud je blok přidělen na stream multiprocessor, tak je rozdělen do skupin po 32 vláknech, který se říká *warp*. Všechna vlákna v jednom warpu jsou vykonávána zároveň, každé z vláken warpu běží na jednom výpočetním jádře. To může znamenat velkou výhodu, ovšem je třeba mít na paměti, že pokud kód obsahuje větvení, tak je napřed vykonávána ta část vláken warpu, pro kterou platí podmínka a ostatní vlákna z warpu stojí a nejsou využita. Teprve poté se provede druhá část vláken a stojí ta první. Podobný problém je u rozvětvených příkazů `case`. Takové situaci se říká *warp divergency*. Je tedy výhodné, aby pro všechna vlákna v daném warpu byla podmínka vyhodnocena stejně.

3.3.3 Paměťový model

Paměť v GPU je hierarchicky rozdělena od pomalejších pamětí, které mají velkou kapacitu po rychlé paměti, které mají malou kapacitu.

Globální paměť Anglicky global memory, často označována jako *device memory*. Globální paměť je společná paměť pro všechny kernely na GPU, často je umístěna přímo na grafické kartě. Před spuštěním kernelu je potřeba nakopírovat vstupní data z operační paměti systému (nazývané také *host memory*) do globální paměti a po operacích je případně zpět. Výjimkou je tzv. *ZeroCopy* přístup do paměti, kdy přistupujeme do operační paměti systému přímo z GPU, ovšem za cenu zvýšené latence. Přístup do globální paměti má velkou latenci (přibližně 400 taktů) [13], proto je při optimalizaci snaha o snížení přístupů do globální paměti na minimum.

Globální paměť u řady Fermi má 2 úrovně cache pamětí. L2 cache je společná pro celou kartu. Každý multiprocessor má svoji L1 cache, která má konfigurovatelnou velikost se sdílenou pamětí.

Constant memory Paměť konstant je malá paměť o velikosti 64 KB, která slouží pouze ke čtení. Data je do ní potřeba nahrát před začátkem provádění kernelu pomocí funkce `cudaMemcpyToSymbol()`. Je podobně pomalá jako globální paměť, ale má svoji paměť cache, takže při vícenásobném použití se její použití vyplatí. Její použití je výhodné, když všechny vlákna ve warpu přistupují k buňce na stejné adrese (tzv. broadcasting), tedy například různé fyzikální koeficienty, které se při běhu programu nemění.

Shared memory Sdílená paměť je paměť společná pro 1 blok. Všechny vlákna z daného bloku z ní můžou číst i do ní zapisovat. Protože je rychlá (viz Tabulka 3.1), slouží často k uchování hodnot z globální paměti, které jsou využity vícekrát (i jiným vláknem než je načetlo). Tím lze jednoduše zredukovat počet přístupů do globální paměti. Je též prostředkem mezivláknové komunikace, tehdy je ovšem dobré použít atomických variant příkazů, které zajistí výlučný přístup nebo bariér, které synchronizují chování.

Local memory je pamět dostupná pouze z jednoho vlákna, ale je velmi pomalá. Ukládají se v ní příliš velké struktury nebo proměnné typu pole, které jsou definované přímo ve vlákně a nevešly by se do registrů. Její použití není příliš vhodné.

Registry jsou nejrychlejší pamětí, kterou může programátor využít. Je dostupná pouze v rámci 1 vlákna a jejich množství je omezeno počtem na jeden stream multiprocessor. Do registrů jsou ukládány všechny proměnné jednoduchých typů a pomocné proměnné vlákna. Dříve, před zařízeními architektury Fermi, byly registry, které přesáhly počet registrů dostupných pro jeden stream multiprocessor, ukládány do globální paměti. Od Fermi jsou tyto tzv. *spilled registry* ukládány do L1 cache, takže tolik nezpomalují běh, ale můžou způsobit nárůst výpadků v cache [6].

Texturní paměť je speciální paměť, která je svázána s nějakým úsekem globální paměti. Její výhoda oproti použití přímo globální paměti spočívá ve využití vlastních cache pamětí, které mohou být využity prostorově. To znamená, že data nejsou do cache ukládána v cache lines po 128 B, jako je tomu u globální paměti, ale spolu s jedním bodem se načte i jeho prostorové okolí, a to jak 1D, 2D nebo 3D. Textury však umožňují pouze čtení, protože zápis do nich by vyvolal potřebu zneplatnění dat uložených do cache pamětí, což by bylo náročné a ztratili bychom výkon. Texturní paměť umožňuje hardwarovou lineární/bilineární interpolaci, kdy při požadavku na určitý bod nevrátí pouze jeho přesnou hodnotu, ale průměrovanou hodnotu z jeho okolí. Interpolace probíhá automaticky v hardware, čímž snižuje náročnost. To je dobré zejména při zpracování a vykreslování obrazu. Textura také zaobaluje přístupy k paměti, které jsou mimo její hranice, a říká, co se v takovém případ bude dělat. V kódu na procesoru je svázána textura s konkrétním úsekem globální paměti a v kódu kernelu proběhne tzv. *fetch*, který hodnotu načte [6].

Tabulka 3.1: Porovnání pamětí v CUDA zařízeních. [6]

typ paměti	dosažitelnost	životnost	propustnost	mají cache
globální paměť	grid	aplikace	177 GB/s	ano
paměť konstant	grid	aplikace		ano
sdílená paměť	blok	kernel	1600 GB/s	ne
lokální paměť	vlákno	kernel		ano
registry	vlákno	kernel	8000 GB/s	ne
texturní paměť	grid	aplikace		ano

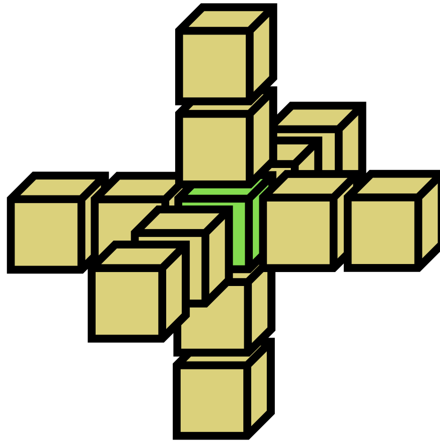
Kapitola 4

Implementace

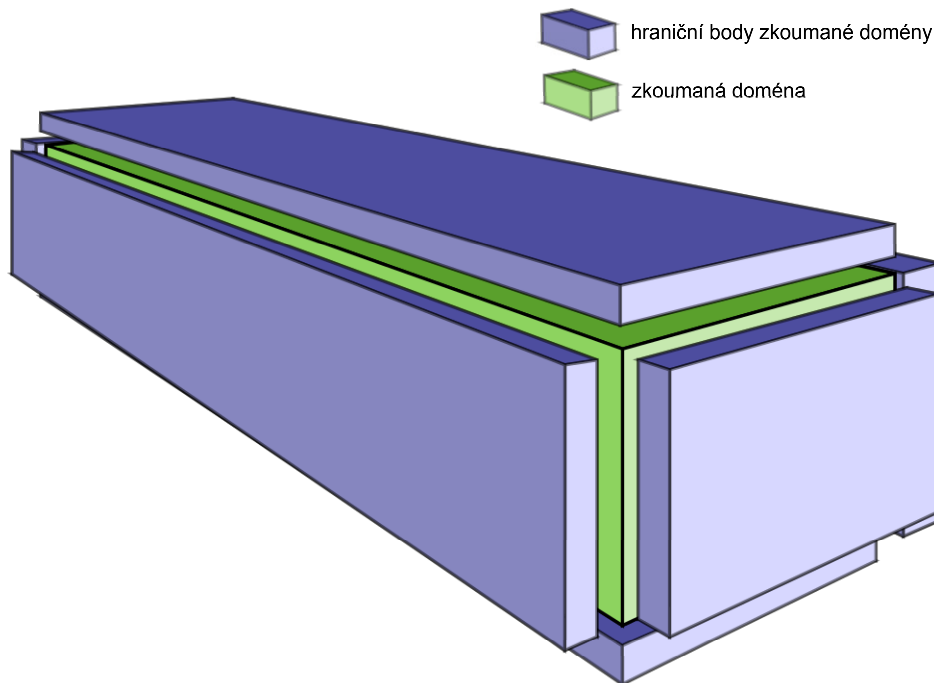
Celá aplikace je psána v jazycích C/C++, které umožnují psát dostatečně nízkoúrovňově a zároveň umožňují pohodlné psaní algoritmů. Problém, který tato aplikace řeší, tedy šíření tepla, je založen na principu zjišťování teploty v bodě a jeho blízkém okolí a následném výpočtu nové teploty z těchto hodnot a z koeficientů prostředí. Největším problémem je jak efektivně načítat matice se vstupními daty, aby byla co nejlépe využita hierarchie pamětí a mezipamětí. Proto byla napřed vyvíjena verze programu, která počítala průměr z 6ti okolí bodu, která řešila prakticky stejný problém jako finální aplikace a dobře se na ní prováděly úpravy a optimalizace. V této kapitole bych chtěl ukázat, jakými úvahami se dospělo až k finální nejrychlejší verzi aplikace, a které optimalizace se ukázaly mít největší vliv na urychlení.

4.1 Popis algoritmu

Z výsledného vztahu 2.10 vyplývá, že algoritmus potřebuje jako vstupy 2 matice. Matici definujeme jako pole se všemi body mřížky z 3D domény o velikosti N_x, N_y, N_z, které obsahuje hodnoty v diskretizovaných bodech domény. První matice obsahuje koeficienty prostředí $\beta = \frac{\lambda}{\rho C_t}$, které zůstávají po dobu simulace konstantní. Každý bod, pro který budeme počítat novou hodnotu teploty, má svůj koeficient prostředí. Druhá matice obsahuje vstupní teplotu tkáně v čase T_0 (dále jen matice T_0). Jelikož v každém bodě počítáme s jeho 12ti okolím, tj. do vzdálenosti 2 v každé dimenzi (viz Obrázek 4.1), tak i pro body na okraji domény musí být známé hraniční body. Body na okraji domény jsou všechny body, které mají libovolnou z dimenzí X, Y nebo Z rovnu 0, 1, N nebo $N-1$, kde N je velikost domény v dané dimenzi. Těmito hraničními body je konstantní hodnota teploty těla $37,0°$C, jak ukazuje Obrázek 4.2.



Obrázek 4.1: Bod(zelený) a jeho 12ti okolí



Obrázek 4.2: Zkoumaná doména a její hraniční body

Hraniční body můžeme implementovat dvěma způsoby:

1. Můžeme nechat původní velikost matice a soustavou podmínek ověřovat, zda načítaný bod neleží na okraji domény v některé ze tří dimenzí (nebo 1 hodnotu vedle okraje). Podmínek je opravdu velké množství. Paměťová náročnost tohoto algoritmu je menší, protože nemusí držet v paměti několikrát tu samou hodnotu. To se ale zmenšuje se zvětšující se velikostí domény. Pro doménu 256^3 je velikost okolí přibližně $4,6\%$ původní velikosti. Pro doménu 1024^3 je to ale již pouze zanedbatelných $1,1\%$.

2. Druhá možnost je zvětšit matici T_0 ve všech dimenzích o 2 body a naplnit je hodnotou $37,0$. Toto řešení je sice paměťově náročnější, ale odpadá u něj potřeba jakýchkoli

podmínek, což je u optimalizace velmi důležité, jelikož je nemusíme vyhodnocovat. Pro každý bod si tím ušetříme vyhodnocení podmínky, zda neleží na okraji některé dimenze.

Výstupní matice bude mít stejné rozměry jako matice T_0, protože záměrem je po každé iteraci algoritmu tyto 2 matice prohodit a z výstupů udělat vstupy pro další iteraci.

4.2 Vstupní data

Vstupní data s hodnotami teploty v čase T_0 a parametry prostředí β jsou uložena ve formátu HDF5 [1], který vyvíjí skupina HDF Group. Slouží k ukládání a efektivní práci s velkým objemem dat rozličných datových typů. Jeho výhodou je, že balík funkcí pracující s tímto formátem je již předinstalovaný v Matlabu, takže je velmi jednoduché vytvářet testovací data. Poskytuje také rozhraní pro C a C++, takže se s ním pracuje velmi dobře.

Úprava výsledného vztahu pro výpočet Výsledný vztah 2.10 obsahuje množství dopředu známých konstant, které můžeme předpočítat a ušetřit práci při výpočtu. Konkrétně takto předpočteme výraz $\Delta t \frac{1}{12}$. Abychom se zbavily operace dělení náročné na výpočet, nebudeme v rovnici dělit číslem $(\Delta x)^2$, ale násobit předpočtenou konstantou $\frac{1}{(\Delta x)^2}$. Totéž provedeme pro $(\Delta y)^2$ a $(\Delta z)^2$. Tyto předpočtené výrazy jsou u CPU i GPU verze algoritmů definovány v souboru `defines.h`.

4.3 Složitost algoritmu

Algoritmus má lineární složitost $O(N)$, protože nezávisle na počtu bodů se každý bod v rámci jedné iterace počítá pouze jednou. S každým dalším přidaným bodem se lineárně zvýší náročnost výpočtu.

4.4 Validita algoritmu

Validování správnosti algoritmu probíhalo porovnáním s výsledky jiných implementací. Pro jejich vytvoření jsem použil skripty v Matlabu. Ty jsou sice pomalé při výpočtech, ale jejich implementace je jednoduchá. Další výhodou Matlabu je i snadná vizualizace získaných dat. Skripty jsou dostupné v adresáři se zdroji.

Při výpočtech nastává problém s aritmetikou plovoucí řádové čárky, protože v ní operace nejsou asociativní [18]. Například $(a + b) + c \neq a + (b + c)$. Je tedy nutné počítat s chybou, která vznikne při výpočtech a zaokrouhlování. K porovnání dvou matic bylo nutné implementovat vlastní porovnávací algoritmus.

4.5 Měření výkonnosti algoritmu

Všechny výpočty simulace probíhají v plovoucí řádové čárce. Pro zjištění výkonnosti algoritmu je tedy výhodné změřit, kolik takových operací zvládne algoritmus vykonat za jednotku času. Efektivitu algoritmu tedy měříme v jednotkách FLOP/s, tedy kolik operací v plovoucí řádové čárce je program provede za sekundu, více je lépe [7].

[1] http://www.hdfgroup.org/HDF5/

4.6 Implementace na procesoru

4.6.1 Specifikace procesoru

Akcelerace verze na procesoru probíhala na školním serveru `pcjaros-gpu.fit.vutbr.cz`, jehož parametry jsou popsány v tabulce 4.1. Implementace algoritmů jsou uloženy v souboru `heat_cpu/mereniCPU.cpp`.

Tabulka 4.1: Specifikace procesoru, na kterém byla prováděna akcelerace[10].

Procesor	Intel Core i7 920
takt	2,66 GHz
počet jader	4
L3 cache	8192 KB
L2 cache	256 KB
L1 cache - data	32 KB
L1 cache - instrukce	32 KB
verze SSE	4.2

4.6.2 Měření výkonnosti algoritmu na procesoru

PAPI (Performance Application Programming Interface) Knihovnu PAPI jsem využil k zjištění výkonu jednotlivých algoritmů. Umožňuje nízkoúrovňové měření výkonnosti u moderních mikroprocesorů. Dokáže využít čítače poskytované procesorem a počítat tak nízkoúrovňové metriky popisující běh vybrané části aplikace, jako jsou počty vykonaných instrukcí, počet cyklů procesoru, úspěšné a neúspěšné zásahy do cache (anglicky cache hit a cache miss)). Neměří v celé aplikaci, ale pouze určité úseky kódu, které jsou vyznačeny programátorem.

PAPI poskytuje dvě aplikační rozhraní, která jsem obě využil. Vysokoúrovňové rozhraní poskytuje předdefinované funkce pro testy, což dělá jeho zařazení do programu velmi jednoduché. Například funkce `PAPI_flops()`, která dává základní přehled o tom, jak dlouho trvalo vykonávání programu a kolik se vykonalo operací v plovoucí řádové čárce, z čehož následně vypočítá počet FLOP/s. Tuto funkci jsem využil pro měření výkonnosti v testech.

Druhé, nízkoúrovňové rozhraní poskytuje přístup přímo k samotným čítačům procesoru, přičemž uživatel si může navolit kombinaci čítačů, které chce měřit. K zjištění všech dostupných čítačů lze použít knihovnou poskytovanou utilitu `papi_avail`. Zároveň lze mít spuštěné pouze omezené množství čítačů a nejsou přípustné všechny kombinace [8]. Toto rozhraní jsem využíval při vývoji k měření výpadků v cache pamětech.

OMP_get_wtime Funkci `OMP_get_wtime()` knihovny OpenMP jsem použil k měření času při běhu nízkoúrovňového rozhraní PAPI. Funkce vrací, kolik sekund ve skutečnosti uplynulo od „nějakého bodu v minulosti“. Tento bod nelze specifikovat, ale je garantováno, že se po dobu běhu aplikace nezmění. Používá se tak, že si označíme počáteční a koncový bod v kódu a pak mezi nimi uděláme rozdíl[2].

[2] http://gcc.gnu.org/onlinedocs/libgomp/omp_005fget_005fwtime.html

4.6.3 Naivní algoritmus pro šíření tepla

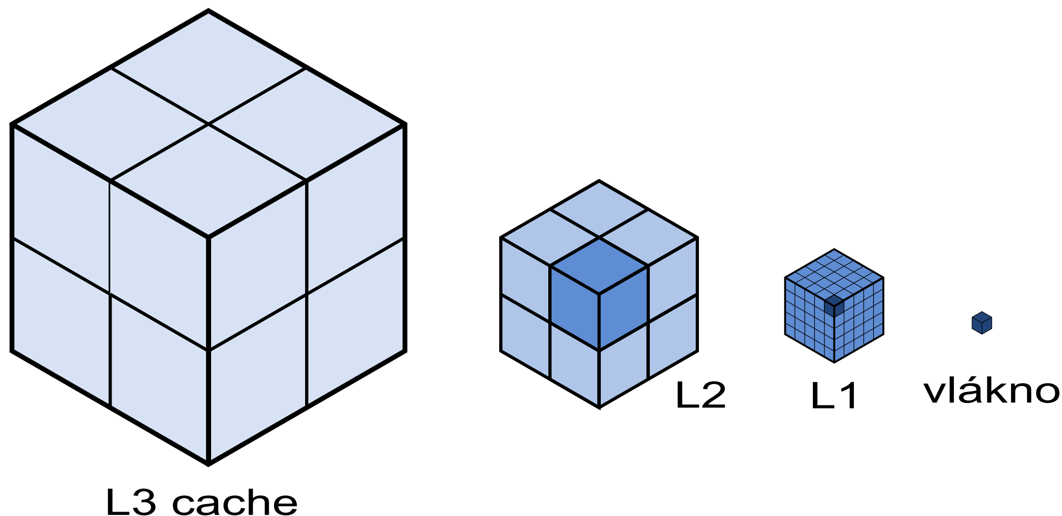
Naivní algoritmus je takový algoritmus, který se nesnaží provádět žádné optimalizace. Implementujeme jej proto, abychom měli s čím porovnávat výkon akcelerované verze, a abychom měli fungující algoritmus, ze kterého můžeme vycházet. Implementovaná verze naivního algoritmu se skládá pouze ze tří for cyklů, které každý iterují přes jednu dimenzi matice T_0. Uvnitř nejvnitřnějšího cyklu je proveden samotný výpočet. Implementace algoritmu se nachází ve funkci `fdtdHeatNaive::RunAlgorithm`.

4.6.4 Paměťová lokalita

Přestože se teoreticky můžeme bavit o 3D maticích, ve skutečnosti je operační paměť uspořádána jako jeden dlouhý vektor. Pokud tedy s každým bodem, pro který počítáme teplotu načteme i jeho 12ti okolí, je dobré si uvědomit, že vzdálenost bodů v 3D matici ne vždy odpovídá skutečné vzdálenosti v paměti. Nejblíže počítanému bodu je okolí v dimenzi X, které skutečně leží vedle počítaného bodu. Vzdálenější jsou okolní body v dimenzi Y a nejvzdálenější v dimenzi Z.

4.6.5 Verze s efektivním využitím cache pamětí

Klíčem k efektivnímu využití CPU je, aby program dokázal naplno využít potenciál cache pamětí, které procesor obsahuje. Účelem pamětí cache je, aby každý bod, který je v programu načítán vícekrát, byl načten z hlavní paměti ideálně jen jednou. Jsou to mezipaměti pro ukládání hodnot načítaných z paměti a v procesoru je jich několik úrovní. Čím menší číslo u úrovně cache je, tím blíže je k procesoru a tím je tato paměť rychlejší, ale má také menší kapacitu. Programátor s nimi však nemůže pracovat přímo, takže je potřeba znát jak na dané platformě fungují. U většiny moderních procesorů funguje tak, že při požadavku na čtení paměti se načte do cache zarovnaných 64 B, ve kterých požadovaný bod leží. Těmto načteným 64 B se říká *cacheline*. Požadujeme-li tento bod znovu (nebo kterýkoli z dané cacheline), pak se načte z rychlé paměti cache, namísto z řádově pomalejší hlavní paměti. Pokud v ní není, jdeme po hierarchii cache pamětí vzhůru až k hlavní paměti. Velikost cacheline jsem zjistil pomocí utility `papi_mem_info` z knihovny PAPI.



Obrázek 4.3: Schéma dělení cache pamětí na zpracovávané krychle ve kterých jsou uchovávána data pro jedno jádro.

V algoritmu jsem rozdělil zpracovávání vstupní 3D matice s hodnotami na krychle, které by se celé měly vlézt do L3 cache. Rozdělení je realizováno pomocí tří vnořených for cyklů, jeden pro každou dimenzi. Tuto krychličku jsem obdobně rozdělil na více malých krychlí nebo kvádrů, které by se měli vlézt do L2 cache a následně to samé pro L1 cache. Tato technika je známá jako *cache blocking*[5]. Velikost tohoto rozdělení můžeme teoreticky spočítat z velikostí jednotlivých cache pamětí, dávají nám ale pouze orientační hodnotu. Pokud bude procesor využívat více aplikací zároveň, bude nucen z cache pamětí naše hodnoty odstraňovat. Musíme také počítat s tím, že načítáme hodnoty z více vstupních matic a potřebujeme i matici pro zápis vypočtených hodnot. Toto rozdělelní se však ukázalo pro malou doménu 256^6 příliš zatěžující. Proto jsem z něj odstranil úroveň pro L2 cache. Implementace algoritmu se nachází ve funkci `fdtdHeatEfectiveCache::RunAlgorithm`. Výsledkem je zrychlení, které ukazuje Tabulka 4.2. Výkon obou verzí je velmi podobný protože test probíhal pouze pro malou doménu 256^6, kdy velikost paměti L3 dostačuje pro uložení několika plátů 3D matice.

Tabulka 4.2: Srovnání výkonu naivní procesorové verze s verzí, která efektivně využívá cache paměti při použití jednoho vlákna.

	výkon	čas pro doménu 256^3
Naivní algoritmus	1,49 GFLOP/s	0,31 s
Algoritmus s efektivním využitím cache	1,5 GFLOP/s	0,28 s

Konstantní proměnné

Všechny konstanty v algoritmu jsem označil jako `const`. Překladač tak může takto označené proměnné optimalizovat tím, že je uchovává v registrech a nemusí se obávat, že jejich hodnota bude změněna. Pokud je v kódu využito hodně konstant, je dobré, aby nebyly umístěny náhodně v paměti, ale byli pohromadě a jejich načtení proběhlo v rámci jedné cacheline. To jsem provedl pomocí struktury `struct konstanty`, ve které je většina konstant umístěna.

4.6.6 SSE verze algoritmu

Prvek, který může velmi zvýšit efektivitu aplikace, jsou vektorové instrukce procesoru. K urychlení výpočtu jsem použil SSE instrukce, které mohou provádět operace ne na jednom, ale na čtyřech operandech typu `float` zároveň (popř. dvou operandech typu `double`). Z paměti zároveň načtou 16 B (`4*sizeof(float)`), které musí být zarovnané na adrese, která je násobkem právě 16 B. S SSE instrukcemi lze pracovat několika způsoby:

1. přímo ve strojovém kódu, což je dost nepřehledné a složité, ovšem můžeme tak dosáhnout největšího zrychlení.
2. nechat kód vektorizovat překladačem, což ne vždy vede k ideálním výsledkům, protože překladač mnohdy kód nevektorizuje.
3. pomocí *intristics*, což jsou funkce jazyka C nebo C++, které zaobalují přístup přímo k assemblerovským instrukcím [9]. Ty jsem nakonec využil.

Výsledkem je téměř trojnásobné zrychlení oproti verzi, která pouze efektivně zacházela s cache pamětmi, jak je vidět v Tabulce 4.3. Je to dáno tím, že SSE verze při jednom vlákně není limitována propustností paměti, která je 25,6 GB/s [10]. Při velikosti domény 256^3 potřebujeme ke každému bodu načíst i jeho 12ti okolí, parametr prostředí a vypočítanou hodnotu zapsat do výsledné matice. Jde o přenesení 15 hodnot typu `float`=60 B, celkově $256^3 * 60\text{B} \approx 1$ GB. Vykonání algoritmu trvalo 0.11 s, potřebná propustnost je teoreticky $\frac{1}{0.11} = 9,09$ GB/s. Algoritmus není limitován propustností 25,6 GB/s. Implementace algoritmu se nachází ve funkci `fdtdHeatSSE::RunAlgorithm`.

Tabulka 4.3: Srovnání výkonu SSE verze s verzí která efektivně využívá cache paměti pro doménu 256^3 při použití jednoho vlákna.

	výkon	čas
Algoritmus s efektivním využitím cache	1,5 GFLOP/s	0,29 s
SSE verze algoritmu	4,4 GFLOP/s	0,11 s

V novějších procesorech Intel od řady Sandy Bridge nalezneme nástupce SSE a to je AVX, které zvládne pracovat s 8 operandy typu `float` zároveň, načítá tedy z paměti 32 B zároveň [3].

[3] http://software.intel.com/en-us/avx

4.6.7 Paralelní verze s OpenMP

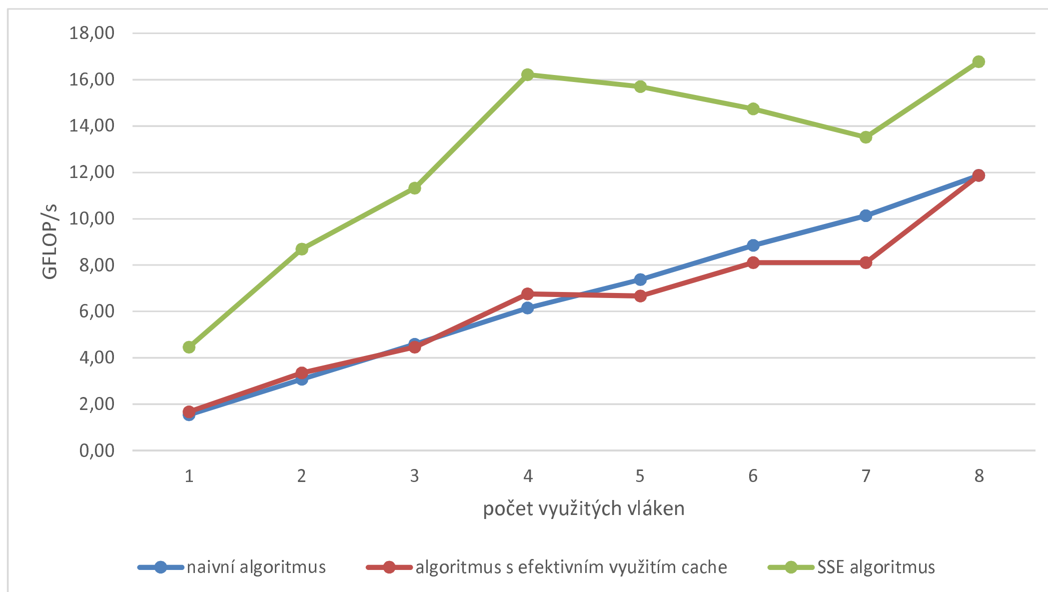
Všechny předchozí algoritmy probíhaly pouze na jednom jádru. Dalším krokem tedy bylo rozdělit práci na všechna jádra, k čemuž jsem použil OpenMP. OpenMP je rozhraní a soubor knihoven pro podporu paralelního programování systémů se sdílenou pamětí pro programovací jazyky C/C++ a FORTRAN. Je multiplatformní, podporuje jak Unix, tak Windows. OpenMP začleňujeme do programu pomocí tzv. direktiv. Ty dovolují říci překladači, které instrukce se mají vykonávat paralelně a která vlákna je budou vykonávat. Výhodou direktiv je velmi jednoduché použití, v C/C++ jsou použity přes `#pragma` instrukce pro překladač, které se ignorují, pokud jim překladač nerozumí. Často převedení programu na OpenMP znamená opravdu malé změny kódu.

Velký potenciál pro paralelní zpracování mají cykly, jejichž jednotlivé iterace se mohou vykonávat nezávisle na sobě. Využívám direktivu `#pragma omp parallel for`, která rozdělí iterace následujícího for cyklu mezi dostupná vlákna procesoru tak, že jedno vlákno pracuje na jedné iteraci daného for cyklu[4]. Spolu s ní používám direktivu `firstprivate(X)`, která zajistí, že proměnnou `X` bude mít každé vlákno ve vlastní kopii.

Použitý procesor má L3 cache společnou pro všechna jádra a L1 a L2 cache má každé jádro privátní. Paralelizoval jsem tedy až for cykly, které realizují rozdělení L3 cache na krychle, které se vejdou se do L2 cache. Všechna vlákna zaráz pracují s jednou krychlí v L3 cache a pak se společně přesunou na další. Algoritmus z toho bude benefituje v situacích, kdy se načítá krajní část nějaké L2 krychle, která zůstane v L3 cache pro použití i ostatními jádry.

Obrázek 4.4 ukazuje nárůst výkonu v závislosti na zvětšování počtu použitých vláken. Zajímavý je graf pro SSE verzi algoritmu, kde výkon lineárně stoupá do použití 4 vláken, ale klesá při použití 5, 6 a 7 vláken. Hlavní příčinou je technologie Intel HyperThreading, u které 1 fyzické jádro sdílí 2 virtuální vlákna. Pokud je aplikace neoptimalizovaná, tak HyperThreading pomáhá vykrývat volné cykly, které vznikají závislostmi v programu. U algoritmů optimalizovaných pro rychlost je to ale kontraproduktivní, protože obě vlákna sdílejí jednu SSE(floating point) jednotku a mají stejnou L1 cache. U situace se 4 vlákny využívá každé vlákno jedno fyzické jádro. Je dobře vidět, že naivní algoritmus není nijak limitován výpadky v cache pamětech ani propustností a jeho výkon roste přesně lineárně.

Dalším příčinou klesání výkonu u SSE verze by teoreticky mohlo být způsobeno dosáhnutím limitu propustnosti pamětí 25,6GB/s[10]. Jak jsem spočítal výše, algoritmus při velikosti domény 256^3 teoreticky načte a zapíše do paměti 1GB dat. Pokud nejrychlejší 8 vláknová verze trvá 0.03 s, tak teoreticky algoritmus potřebuje propustnost $\frac{1}{0.03} = 33$ GB/s, což je více než propustnost pamětí. Prakticky je to velmi nepravděpodobné díky využití cache pamětí, které umožňují znovupoužitelnost jednou načtených hodnot.



Obrázek 4.4: Graf závislost výkonnosti procesorových algoritmů na počtu využitých vláken pro doménu 256^3.

4.7 Implementace na grafické kartě

4.7.1 Parametry karty

Obzvlášť u grafických karet platí, že pokud chceme urychlit algoritmus, je potřeba přesně znát parametry karty, pro kterou optimalizujeme, protože na jiné kartě se algoritmus může chovat výkonnostně úplně jinak. Grafické karty NVIDIA se dělí podle tzv. *compute capability*, kdy všechny karty se jednou compute capability mají stejné vlastnosti stream multiprocesorů. Liší se jen jejich počtem, takty a množstvím globální paměti. Od compute capability se odvíjí schopnosti využívat všechny možnosti Cuda Toolkitu.

Pro simulace byla použita karta NVIDIA GeForce 580 GTX od společnosti Asus s 1,5 GB RAM globální paměti. Karta je postavena na architektuře Fermi, která má compute capability 2.0. Parametry karty, které mě při optimalizaci nejvíce omezovaly nebo jsou pro implementaci důležité, jsem shrnul do Tabulky 4.4.

Tabulka 4.4: Parametry grafické karty, na které probíhaly simulace. [13] [14]

grafická karta	NVIDIA GeForce 580 GTX
maximum vláken na blok	1024
maximum vláken na multiprocesor	1536
stream multiprocesorů	16
jader na multiprocesoru	32
registrů na multiprocesor	32768
maximální velikost textury v 3D	2048x2048x2048
velikost warpu	32
velikost sdílené paměti/L1 cache na blok	48 kB/16 kB nebo 16kB/48kB
bank sdílené paměti	32
cacheline	128 B
propustnost paměti	192,4 GB/s

4.7.2 Single / double precision

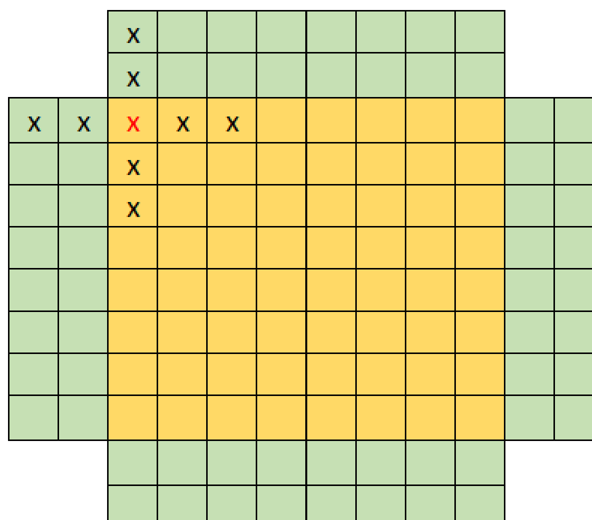
Na doporučení vedoucího byla zvolena za dostačující přesnost výpočtů v single precision (typ `float`). Je to výhodné, protože výpočet v double precision (typ `double`) zabírá víc místa v paměti (32 b vs. 64 b) a práce s ním je až několikanásobně pomalejší. V posledních generacích nastalo velké zlepšení, ale stále je zde velký rozdíl. Přesné srovnání můžeme vidět v tabulce 10 v CUDA C Programming Guide[13]. Z tabulky také vyplývá, že se nevyplatí využívat typ `int` místo typu `float` jako u CPU, protože rychlost jejich zpracování je téměř stejná, v zařízeních s compute capability 3.0 a 3.5 se dokonce vyplatí pro výpočty používat typ `float`.

4.7.3 Rozměry matic se vstupními daty

Velikost paměti v grafických kartách je oproti CPU značně omezená. Jak je zmíněno v sekci 4.1, pro výpočet potřebujeme 3 téměř stejně velké matice s datovým typem float. Velikost globální paměti karty je 1536 MB. Z toho vychází, že jedna krychlová 3D matice může mít dimenzi velkou maximálně $\sqrt[3]{\frac{1536*1024*1024}{4*3}} = 512$. To by ovšem matice nesměly mít hraniční body a v globální paměti by nesmělo být uloženo nic navíc. Takže byly zvoleny rozměry matice 256x256x256.

4.7.4 Implementace algoritmu

V následující části bych rád rozvedl všechny směry, kterými jsem se vydal při hledání nejrychlejší implementace, popsal problémy a jejich řešení. U všech kernelů je hlavně řešen problém se zarovnáním paměti, protože s každým blokem je potřeba načíst i 2 body z vedlejšího bloku, což prakticky znemožňuje ideální přístup k paměti, jak je vidět na Obrázku 4.5 (situace pouze ve 2D). Údaje o využití globální paměti pochází z programu NVIDIA Visual Profiler. Počet GFLOP/s je počítán manuálně poměrem celkového počtu provedených operací v plovoucí řádové čárce a času běhu kernelu.



Obrázek 4.5: 2D znázornění bloku; každý blok (žlutá) potřebuje načíst 2 body do každé dimenze (zelená), aby i body na hranici bloku (například bod označený červeným X) měly k dispozici 12ti okolí (8mi okolí ve 2D).

Ve všech algoritmech je matice Beta s parametry prostředí implementována pomocí 3D texturní paměti. Je to výhodné proto, že texturní paměť má svoji vlastní cache paměť a tím se vyhneme problému zamořování L2 a L1 cache paměti daty, která nejsou potřeba opakovaně, tvz. *cache polution*.

Běh algoritmů, které využívají sdílenou paměť je rozdělen na 2 části, načítání dat do sdílené paměti a samotný výpočet, které jsou odděleny synchronizací `__syncthreads()`, která zajistí, že sdílená paměť obsahuje všechny body potřebné pro výpočet.

4.7.5 Naivní kernel

Parametry	
dimGrid(x,y,z)	32,32,32
dimBlock(x,y,z)	8,8,8
Použita sdílená paměť	ne
Sdílená paměť/L1 cache	16 kB/48 kB

Napsaný algoritmus na procesoru není složité převést do CUDA. Odstraní se všechny úrovně `for` cyklů, které ve verzi pro procesor rozdělovaly paměť do kostek pro úrovně cache pamětí. V tomto kroku máme připravenou naivní verzi pro CUDA, která ovšem trpí všemi možnými neduhy. Například že jeden warp musí načíst několik cachelines, z nichž z každé je využita jen čtvrtina (8 z 32). Velikost bloku 8x8x8 je zvolna proto, že jedna "dlaždice", tedy všechny body se stejnou souřadnicí Z, je dobře rozdělitelná do 2 warpů.

Výsledky	
Využití paměti	18 GB/s
Výkon	108 GFLOP/s
Čas jedné iterace	4,48 ms

4.7.6 Kernel pro 4 krychle 8x8x8

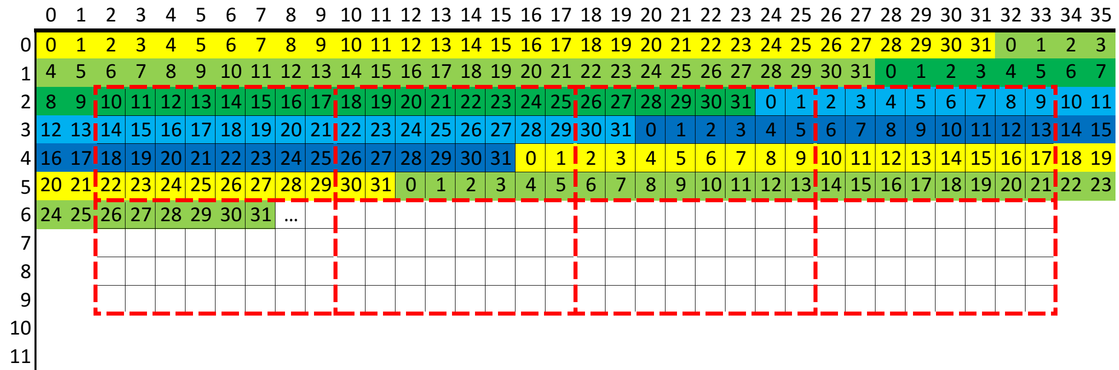
Parametry	
dimGrid(x,y,z)	32,32,8
dimBlock(x,y,z)	8,8,8
Použita sdílená paměť	ano
Sdílená paměť/L1 cache	48 kB/16 kB

Jelikož úplně zarovnaný přístup je u toho algoritmu téměř vyloučen, je zde alespoň snaha o to co nejvíce využívat načítaná data. Aby byla jednou načtená data z globální paměti využita co nejvíce, tak se zpracovává několik krychlí zároveň. Konkrétně 4 krychle 8x8x8 v ose X, protože 4 krychle*délka krychle 8*4 B na float = 128 B = délka cacheline. Když postupně načítám z globální paměti 4 kostky za sebou, dobře se mi vejdou do L1 cache, která má 16 kB. Sdílenou pamětí jsem limitován (viz. dále) na 2 bloky na jeden SM. A všechny 4 krychle zabírají právě 8*8*8*sizeof(float)=8 kB. Aby bylo data kde skladovat, byla přidána sdílená paměť. Ta by měla zabírat podle 4 krychlí po 8x8x8 bodů a 2 body okrajů každé dimenze celkově [12][12][36]. To dává dohromady 12*12*36*sizeof(float)=20,25 kB. Na jednom stream multiprocessoru tedy mohou být nejvýše 2 bloky.

Aby co nejvíce přístupů do paměti bylo zarovnaných, tak začátek každého řádku první krychle je zarovnaný na 32 B. Toho je docíleno přidáním 6 výplňových bodů do každé strany vstupní matice v ose X a tedy načítání okrajů v osách Y a Z je také velmi efektivní. Ne tak již načítání okrajů v ose X z globální paměti, ale jejich počet je snížen na 2 z celkových 8, protože zbylé okraje se načtou s načtením vedlejších krychlí.

Se sdílenou pamětí ovšem přišel problém zvaný *shared memory banks conflict*. Sdílená paměť je u Fermi rozdělena do 32 oblastí, tzv. bank. Ty jsou zde proto, aby požadavek na sdílenou paměť od každého vlákna warpu mohl být vyřízen jednou bankou a nemuselo dojít serializaci přístupů. Paměť je rozdělena do bank tak, že v bance 0 jsou buňky 0,32,64,... v bance 1 jsou buňky 1,33,65,... atd...

Pokud vlákna přistupují k nepřerušované paměti, tak k tomuto problému nemůže dojít. Ovšem zde je paměť složena více krychlí a okrajů. Na Obrázku 4.6 jsou znázorněny v červených obdélnících přístupy vláken jednoho warpu ke středovým bodům 12ti okolí. Například v prvním červeném obdélníku se přistupuje 2x do bank 14-25, což způsobuje serializovaný přístup.



Obrázek 4.6: Pohled na sdílenou paměť pro dimenzi $z = 2$. Čísla značí, ve které bance se buňka paměti nachází. Červené obdélníky označují přístupy vláken jednoho warpu ke středovým bodům 12ti okolí.

Možné řešení je rozšířit dimenzi X sdílené paměti o tolik bodů, aby vždy všechny vlákna z warpu přistupovala do jiné banky. Zde je to konkrétně 4 body, výsledná velikost sdílené paměti tedy bude [12][12][40]. Celkově tak zabere 1 blok 12*12*40*sizeof(float) Bytů = 22,5 kB, takže na jeden stream multiprocessor se stále vejdou 2 bloky. Na Obrázku 4.7 vidíme přístup již bez shared memory banks conflicts.

Obrázek 4.7: Pohled na sdílenou paměť pro dimenzi $z = 2$. Čísla značí, ve které bance se buňka paměti nachází.

Na zápise se podílejí zároveň všechna vlákna z bloku a žádné nestojí. Postupně se spočítají nové hodnoty teploty pro všechny 4 krychle a zapíšou do výstupní matice.

Aby se stále dokola nemusely počítat hodnoty pro `tx+2`, `ty+2` a `tz+2`, které jsou používány velmi frekventovaně, tak jsou nahrazeny pomocnými registry `tx_plus2`, `ty_plus2` a `tz_plus2` s jejich hodnotami.

Výsledky	
Využití paměti	98 GB/s
Výkon	202 GFLOP/s
Čas	2,4 ms

4.7.7 Kernel s dlaždicemi 16x16

Parametry	
dimGrid(x,y,z)	22,22,22
dimBlock(x,y,z)	16,16,2
Použita sdílená paměť	ano
Sdílená paměť/L1 cache	48 kB/16 kB

Výkon algoritmu se 4 krychlemi byl srážen především neefektivním načítáním okrajů v ose X. V tomto algoritmu jsem se rozhodl vyřešit tento problém tím, že počítané body i okolí budou načítat zaráz všechna vlákna a při zápisu budou okrajová vlákna stát.

Aby byl naplno využit potenciál jednoho stream multiprocesoru, bylo zvoleno, že jeden blok se bude skládat ze dvou dlaždic 16x16, což je dohromady 512. V jednom bloku se do sdílené paměti načte 16x16x16 bodů, přičemž počítat se bude pouze pro 12x12x12. Tím se naplno využije kapacita stream multiprocessoru, protože se na něj vejdou zároveň 3 bloky (3*512 vláken zaplní všech 1536 maximálních vláken a 3*16 kB sdílené paměti dá dohromady 48 kB celkové sdílené paměti).

Slabým místem tohoto kernelu je ale výpočet a zápis zpět. Jelikož je potřeba počítat pouze pro 12x12x12=144 vláken, zbylých 256-144=112 vláken je zde zbytečných. Přebytečná vlákna v osách Y a Z jde podmínkou oddělit a předčasně ukončit. Podmínka se vykoná pro všechny warpy stejně a je vše v pořádku. Problém je s přebytečnými vlákny v ose X, kdy celá čtvrtina, tedy 4 z 16 vláken musí stát. Tomuto problému se říká *warp divergency*. Problém lze vyřešit přemapováním vláken tak, že pracovat bude prvních 144 vláken a zbylá vlákna se ukončí podmínkou podobně jako v osách Y a Z. Ovšem operace přemapování je velmi drahá, protože je nutné použít operace modulo a děleno. Navíc opět dojde k shared memory banks conflict a k serializaci přístupu, takže je výhodnější nechat problém být.

Výsledky	
Využití paměti	112 GB/s
Výkon	210 GFLOP/s
Čas jedné iterace	2,54 ms

4.7.8 Kernel s obdélníkovým blokem

Parametry	
dimGrid(x,y,z)	10,32,32
dimBlock(x,y,z)	32,8,3
Použita sdílená paměť	ano
Sdílená paměť/L1 cache	48 kB/16 kB

V přechozích kernelech bylo vždy problémem načítání okrajů v ose X, popř. zbytečné nevyužití celé načtené cacheline. Logickým krokem tedy bylo načítání tak, aby se v jednom warpu načetlo co nejvíce najednou a to z jedné části paměti. Proto má blok velikost v ose X 32=velikost warpu. Z těchto 32 je 28 počítaných bodů a 4 body okraje. Speciální vlákna pro načítání okrajů jsou pouze v ose X. Velikost dimenze Y je 8. Takto vznikne dlaždice 8x32, kterou můžeme jednoduše opakovaně využít k načtení počítaného kvádru 8x8x32, horních a dolních podstavy 2x8x32 a také okrajů v ose Y, kdy této dlaždici prohodíme osy Y a Z.

Tím, že jsou pomocná vlákna pro načítání okrajů pouze v ose X, se téměř zbavíme thread divergency. Nyní stojí pouze 4 vlákna z 32, což je pouze jedna osmina.

Využita je zde direktiva `#pragma unroll X`. Ta způsobí X krát rozbalení for cyklu, který se za ní nachází. Tím se sníží režie pro počítání koncových podmínek cyklu, protože je přesně znám počet opakování. Pokud jsou jednotlivé iterace na sobě nezávislé, můžou se také vykonávat nezávisle na sobě. Ne vždy ovšem algoritmus zrychlí, to je potřeba ověřit experimenty.

Kernel má 3 dlaždice vláken v ose Z o rozměrech 32x8, kterým rozděluje rovnoměrně práci. Naplno využívá kapacitu jednoho stream multiprocessoru, protože 32*8*3=768 vláken a potřeba sdílené paměti 32*12*12*sizeof(float)=18 kB znamená, že mohou běžet 2 bloky na stream multiprocessoru zároveň.

Výsledky	
Využití paměti	117 GB/s
Výkon	231 GFLOP/s
Čas jedné iterace	2,3 ms

4.7.9 Kernel bez sdílené paměti

Parametry	
dimGrid(x,y,z)	8,32,32
dimBlock(x,y,z)	32,8,3
Použita sdílená paměť	ne
Sdílená paměť/L1 cache	16 kB/48 kB

Jak již bylo řečeno výše, registry na architektuře Fermi, které přesáhnou počet dostupný pro jeden stream multiprocessor, jsou uloženy v L1 cache. Nově Fermi také umožnuje programátorovi nakonfigurovat si poměr sdílené paměti a L1 cache a to v poměru 16 kB/48 kB a 48 kB/16 kB. Při pohledu na Tabulku 3.1 s porovnáním pamětí se nabízí nevyužívat vůbec sdílené paměti, potřebné věci ukládat přímo do rychlých registrů. Zároveň využít konfiguraci, kdy má L1 cache velikost 48kB a použít ji místo sdílené paměti pro uchovávání hodnot z globální paměti pro další použití.

Pokud zvolíme stejně jako v předchozím případě kvádr dlouhý v ose X, tak většina hodnot, co bude načtena v ose X bude využita v současném bloku. Blok v ose Y nesmí být příliš dlouhý, protože než by se algoritmus dostal k další dlaždici, tak by již body z minulé dlaždice již v cache nebyly. Pokud ale bude velikost příliš malá, tak bude příliš velký poměr mezi počtem počítaných bodů a okraji, což znamená příliš mnoho zbytečných načítání. Jako ideální velikost bloku v ose Y se po experimentech ukázala velikost 8.

Takovýto blok má celkem 256 vláken. Protože již nejsme limitováni sdílenou pamětí, na jeden stream multiprocessor se vejde 1536/256=6 bloků, které dobře skryjí latenci při čekání na paměť. 48 kB cache paměti se teoreticky rozdělí pro 6 bloků, pro každý $\frac{48}{6} = 8$ kB. Každý blok si musí uchovávat 5 bodů na výšku * velikost dlaždice = 5*256*sizeof(float) = 5 kB. Nemělo by tedy docházet k výpadkům v L1 cache paměti.

Všechny hodnoty 12ti okolí jsou uchovávány v registrech. Jestliže jeden stream multiprocessor má k dispozici 32 768 registrů, tak ty se rozdělí na 6 bloků. Na každý blok je tedy $\frac{32768}{6} \approx 5461$ registrů. Blok má 256 vláken, tedy má k dispozici $\frac{5461}{256} \approx 21$. Podle dat které poskytuje NVIDIA Visual Profiler a i podle rozšířených informací při překladu s parametrem `--ptxas-options="-v"` používá kernel právě 21 registrů.

Uchovávání hodnot v registrech má za důsledek to, že můžeme vypustit synchronizaci nutnou u kernelů se sdílenou pamětí k zajištění toho, že všechny bloky načetly svou část globální paměti do sdílené paměti. To odstraní čekání na nejpomalejší bloky.

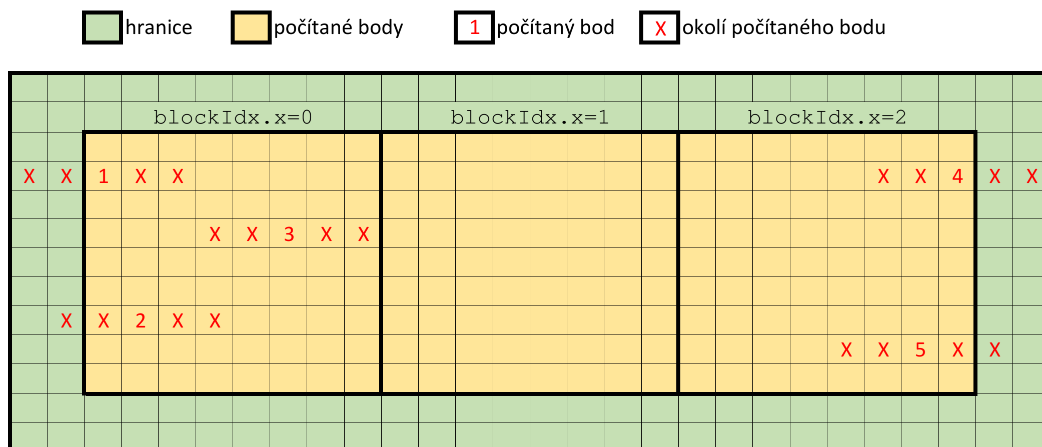
Výsledky	
Využití paměti	136GB/s
Výkon	256 GFLOP/s
Čas jedné iterace	1,9ms

4.7.10 Kernel s generovanými okraji

Parametry	
dimGrid(x,y,z)	8,32,32
dimBlock(x,y,z)	32,8,1
Použita sdílená paměť	ne
Sdílená paměť/L1 cache	16 kB/48 kB

Výše uvedené algoritmy mají všechny společnou vlastnost a to že všechny mají hraniční hodnoty uložené společně se vstupní maticí. Šetří si tím velké množství podmínek, které by musely pro každý bod vykonat, aby rozhodly, zda některý z bodů jejich 12ti okolí neleží mimo hranici. V tomto algoritmu jsem se zaměřil na minimalizaci množství těchto podmínek. Kernel je modifikovanou verzí předchozího kernelu bez sdílené paměti.

Pro každý počítaný bod s ním musíme načíst i jeho 12ti okolí. Počítaný bod bude ve vstupní matici vždy, ale některé body jeho 12ti okolí už být nemusí. V každé ose je 5 druhů bodů, lišících se tím, jak jejich 12ti okolí zasahuje přes hranice, jak můžeme vidět na 2D ilustraci problému pro osu X na Obrázku 4.8.



Obrázek 4.8: 2D ilustrace 5 druhů bodů, z nichž každý v ose X jinak přesahuje svým 12ti okolím (4 okolím ve 2D v ose X) přes hranice bloku.

V levém krajním bloku (bloku 0) narazíme pouze na 3 druhy bodů:

- Bod 1 (threadIdx.x==0), dva body z 12ti okolí přesahují hranici do záporné osy X
- Bod 2 (threadIdx.x==1), jeden bod z 12ti okolí přesahují hranici do záporné osy X
- Bod 3, žádný bod 12ti okolí nepřesahuje hranici

Obdobně je to pro pravý krajní blok (blok 2), zde jsou ale přesahy do kladné osy X. V prostředním bloku (blok 1) nikdy nebude přes hranice v ose X nic přesahovat, protože se hranice v ose X nedotýká. Stejné to bude v osách Y a Z.

Jinými slovy, hledáme vlákna, která jsou v krajním bloku na jeho okraji nebo 1 bod od okraje. Testovat každý bod 12ti okolí každého počítaného bodu je zbytečně náročné. Okrajová vlákna vybereme podmínkou a řekneme, že mají místo načítání z paměti mít pro

přetečené body konstantu. V blocích, které neleží v dané ose na hranici, jsou takové testy zbytečné, proto se je z toho budeme snažit vynechat.

Vytvoříme tedy podmínky (pro každou osu zvlášť), které určí zda se jedná o blok uprostřed a pak se ho podmínky netýkají, a nebo se jedná o blok na kraji (a jakém) a pak se přidají podmínky pro vlákna na okraji.

Algoritmus 3: Pseudoalgoritmus výpočtu

```
1 ...
2 for {int i=0; i<počet dlaždic v ose Z; i++}
3 {
4    zjištění pozice bloku v ose X
5       začátek > načtení okolí v ose X s podmínkami pro počáteční 2 vlákna
6       uprostřed > načtení okolí v ose X bez podmínek
7       konec > načtení okolí v ose X s podmínkami pro poslední 2 vlákna
8    zjištění pozice bloku v ose Y
9       začátek > načtení okolí v ose Y s podmínkami pro počáteční 2 vlákna
10      uprostřed > načtení okolí v ose Y bez podmínek
11      konec > načtení okolí v ose Y s podmínkami pro poslední 2 vlákna
12   zjištění pozice bloku v ose Z
13      začátek > načtení okolí v ose Z s podmínkami pro počáteční 2 vlákna
14      uprostřed > načtení okolí v ose Z bez podmínek
15      konec > načtení okolí v ose Z s podmínkami pro poslední 2 vlákna
16   načtení počítaného bodu
17   výpočet a zapsání výsledku
18 }
```

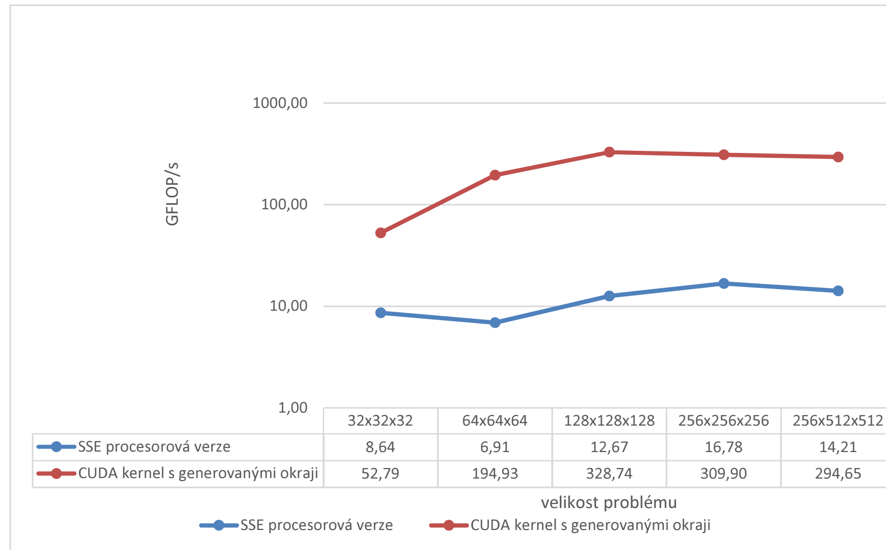
Tento přístup má ale nevýhodu, že pro každou dlaždici musíme znova počítat, v jakém bloku se nacházíme. Proto jsem vymyslel zanořenou verzi, kdy se pro všechny polohy bloku v ose Z provede podmínka na polohy bloku v ose Y a pro ni to samé s osou X. Celkem to dává $3^3 = 27$ možností. Na nejzanořenější úrovni je teprve for cyklus přes všechny dlaždice a následný výpočet. Opět je použit příkaz pro rozbalení for cyklů, `#pragma unroll`.

Podmínky v ose Z jsou potřeba v okrajových blocích pouze pro první 2 (popř. poslední 2) dlaždice vláken, takže můžeme podmínky provádět pouze pro ně a zbytek bloku provést tak, jako by se jednalo o vnitřní blok (z pohledu osy Z).

Všech 27 možností je v kódu rozepsaných, a protože kód pro samotné načítání a výpočet se často opakuje, a bylo by téměř nemožné jej udržovat, rozhodl jsem se jej generovat pomocí maker. Abych nepsal funkce, které zbytečně zdržují výpočet svým voláním, na konci každé větve je parametrické makro `VYPOCET`, kterému jsou předávány 3 parametry, každý s kódem pro načtení okolí počítaného bodu v dané ose. Takto všechnu práci zastane preprocesor.

Výsledky	
Využití paměti	143 GB/s
Výkon	310 GFLOP/s
Čas jedné iterace	1,57 ms

4.8 Výsledky implementací

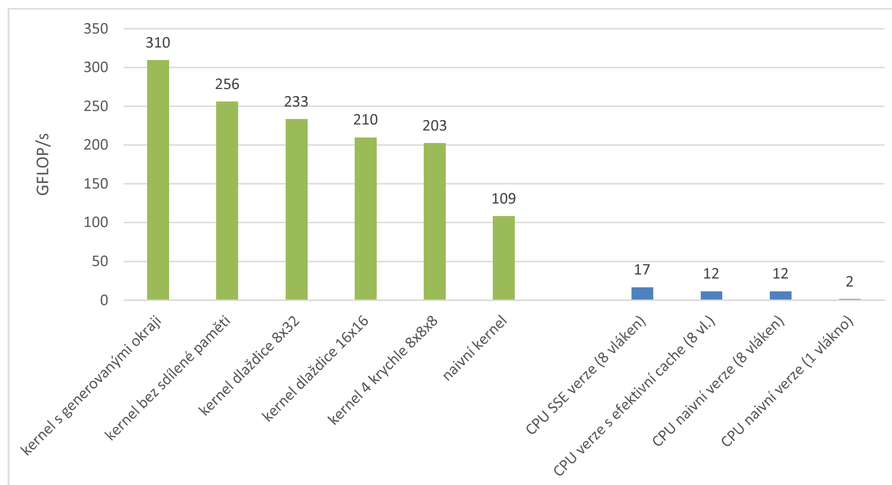


	32x32x32	64x64x64	128x128x128	256x256x256	256x512x512
SSE procesorová verze	8,64	6,91	12,67	16,78	14,21
CUDA kernel s generovanými okraji	52,79	194,93	328,74	309,90	294,65

Obrázek 4.9: Graf závislosti výkonu na velikosti problému pro nejrychlejší CPU a GPU verzi.

Obrázek 4.9 potvrzuje, že použití CUDA je vhodné pouze pro řešení velkých problémů, u kterých může být plně využit výkon zařízení. Malé problémy velikosti 32^3 nebo 64^3 nezaměstnají plně všechny výpočetní jednotky grafické karty. To samé se děje u procesorové SSE verze, která nevyužije naplno všechna vlákna, která jsou rozdělována po velkých částech do L3 cache paměti.

Na závěr práce přikládám v Obrázku 4.10 finální porovnání výkonu všech naimplementovaných algoritmů.



Obrázek 4.10: Porovnání výkonu všech implementovaných verzí na CPU a GPU pro problém 256^3. GPU implementace jsou vyznačeny zelenou barvou, CPU implementace modrou.

Kapitola 5

Závěr

Cílem této práce bylo implementovat a urychlit algoritmus šíření tepla pomocí metody konečných diferencí. Nejprve jsem vytvořil naivní procesorovou verzi, která dosáhla výkonu 1,49 GFLOP/s. K té jsem přidal efektivnější práci s cache pamětmi, převedl všechny výpočty do SSE a paralelizoval ji pomocí OpenMP, čímž jsem se dostal k výkonu 16,7 GFLOP/s. Dále jsem se zaměřil na GPU, kde jsem vyrobil několik verzí. Naivní verze pro GPU dosáhla výkonu 108 GFLOP/s. Finální verze dosáhla výkonu 310 GFLOP/s, což odpovídá 18,5 násobnému zrychlení oproti nejvýkonnější procesorové verzi. Dobrých výsledků se podařilo dosáhnout i u paměťové propustnosti. Celkem 143 GB/s z teoretické maximální propustnosti 192 GB/s grafické karty. Při problému, který nepřeje zarovnaným přístupům do paměti je to dobrý výsledek. Překvapivým závěrem experimentů je, že nejrychlejší GPU verze je založena na registrech, a nikoli na využití sdílené paměti. Velkou zásluhu na tom má L1 cache architektury Fermi.

Do budoucna by do procesorové verze mohla být implementována podpora AVX. U matice T_0 by mohlo být zajímavé odstranit hraniční hodnoty na koncích dimenzí X a Y, protože při přetečení by ukazatel ukazoval na začátek nového řádku/dlaždice, kde jsou stejné hodnoty teploty těla, čímž by se mohla ušetřit paměť. GPU verze by mohla být rozšířena o možnost zpracovávání více kroků v jednom kernelu, což by vedlo k redukci synchronizace.

Kód, který v této práci vznikl, je lehce aplikovatelný na mnoho podobných problémů, které lze řešit metodou konečných direfencí. Aplikací tohoto postupu na jiný problém lze očekávat podobná zrychlení.

Řešení práce mi umožnilo nahlédnout do světa vysoce náročných výpočtů, který mě zaujal a chtěl bych v jeho studiu dále pokračovat.

Literatura

[1] IEEE Standard for Validation of Computational Electromagnetics Computer Modeling and Simulations. *IEEE STD 1597.1-2008*, 2008: str. 27, doi:10.1109/IEESTD.2008.4957854.

[2] Advanced Micro Devices, Inc.: *AMD Opteron™6300 Series Processors.* [online]. 2013 [cit. 2013-04-28].
URL `<http://www.amd.com/us/products/server/processors/6000-series-platform/6300/Pages/6300-series-processors.aspx\#5>`

[3] Barkanov, E.: *Introdution to the finite element method.* Riga, 2001 [cit. 2013-02-11], str. 5.
URL `<http://mmc.geofisica.unam.mx/Bibliografia/Matematicas/EDP/MetodosNumericos/IntroductionToTheFiniteElementMathod.pdf>`

[4] Chapman, B.; Jost, G.; der Pas, R. V.: *Using OpenMP.* Cambridge: MIT Press, 2008, ISBN 978-0-262-53302-7, 353 s.

[5] Doerner, W.: *Cache Blocking Techniques.* [online]. 2013 [cit. 2013-02-21].
URL
`<http://software.intel.com/en-us/articles/cache-blocking-techniques>`

[6] Farber, R.: *CUDA application design and development.* Amsterdam: Elsevier, 2011, ISBN 978-0-12-388426-8, 315 s.

[7] Fosdick, L. D.: *An Introduction to High-Performance Scientific Computing*, kapitola 11. Morgan Kaufmann Publishers, 1996, ISBN 0-262-06181-3, str. 354.

[8] Innovative Computing Laboratory: *PAPI: Overview.* [online]. 2013 [cit. 2013-04-12].
URL `<http://icl.cs.utk.edu/papi/overview/index.html>`

[9] Intel Corporation: *Intel® C++ Intrinsic Reference* [online]. 2007 [cit. 2013-03-02].
URL `<http://software.intel.com/sites/default/files/m/9/4/c/8/e/18072-347603.pdf>`

[10] Intel Corporation: *Intel® Core™i7-920 Processor.* [online]. 2013 [cit. 2013-03-09].
URL `<http://ark.intel.com/products/37147>`

[11] Kirk, D.; mei Hwu, W.: *Programming massively parallel processors.* Burlington: Morgan Kaufmann Publishers, 2010, ISBN 978-0-12-381472-2, 258 s.

[12] Miroslav Španiel: *Podklady ke studiu předmětu MKP2.* 2009 [cit. 2013-01-18].
URL
`<http://mechanika2.fs.cvut.cz/old/pme/predmety/mmkp/podklady/mod.pdf>`

[13] NVIDIA Corporation: *CUDA C Programming Guide: Design Guide*. 2012 [cit. 2013-04-15], 174 s.
URL <http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf>

[14] NVIDIA Corporation: *GeForce GTX 580: Specifications*. [online]. 2013 [cit. 2013-03-14].
URL <http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-580/specifications>

[15] NVIDIA Corporation: *Direct Compute*. [online]. 2013 [cit. 2013-04-28].
URL <https://developer.nvidia.com/directcompute>

[16] Rokyta, M.: *Aplikovaná matematika IV – kap. 24: Parciální diferenciální rovnice*. 2001 [cit. 2013-02-12], str. 7.
URL <http://www.karlin.mff.cuni.cz/~rokyta/vyuka/1112/ls/F_apl_mat/ApMat_Kap_24_tisk.pdf>

[17] Sharma, S.: *Dosimetry and treatment planning for therapeutic and diagnostic ultrasound*. Bachelor of engineering, Australian National University, Department of Engineering, Canberra, 2012.

[18] Villa, O.; Chavarría-mir, D.; Gurumoorthi, V.; aj.: *Effects of Floating-Point non-Associativity on Numerical Computations on Massively Multithreaded Systems*. [online]. [cit. 2013-04-21].
URL <http://cass-mt.pnnl.gov/docs/pubs/pnnleffects_of_floating-pointpaper.pdf>

[19] Wilt, N.: *The CUDA Handbook*, kapitola Streaming Multiprocessors. 2012 [cit. 2013-04-15], str. 1.
URL <http://www.cudahandbook.com/uploads/Chapter_8._Streaming_Multiprocessors.pdf>

[20] Zolfaghari, A.; Maerefat, M.: *Bioheat Transfer, Developments in Heat Transfer*. Intech, 2011 [cit. 2013-01-23], ISBN 978-953-307-569-3, str. 156.
URL <http://www.intechopen.com/books/developments-in-heat-transfer/bioheat-transfer>

Příloha A

Návod na zprovoznění aplikace

A.1 Překlad aplikace

Pro překlad aplikací je nutno mít nainstalováno následující softwarové vybavení. V souboru Makefile je dále nutné upravit cesty k nainstalovaným knihovnám.

CPU verze

- HDF5 knihovny pro C++
- PAPI knihovny
- OpenMP
- překladač g++ verze 4.6.3

GPU verze

- HDF5 knihovny pro C++
- CUDA Toolkit verze 5.0
- překladač g++ verze 4.6.3

A.2 Spuštění aplikace

Pro spuštění aplikací je nutno mít následující hardware:

CPU verze

- Intel Core i7 procesor s podporou SSE4.2

GPU verze

- grafickou kartu nVidia podporující CUDA Compute Capability alespoň verze 2.0

Vše výše uvedené splňuje stroj `pcjaros-gpu.fit.vutbr.cz`, přístupný z vnitřní sítě VUT FIT.