VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

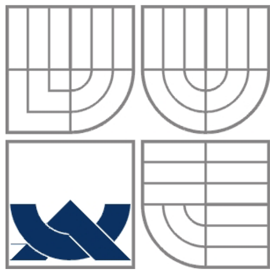
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

SIMULACE ŠÍŘENÍ TEPLA S ČASOVĚ PROMĚNNÝM ZDROJEM S VYUŽITÍM GPU

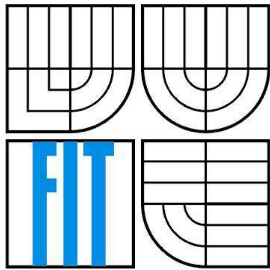
BAKALÁŘSKÁ PRÁCE
BACHELOR‘S THESIS

AUTOR PRÁCE PAVEL HÁLA
AUTHOR

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

SIMULACE ŠÍŘENÍ TEPLA S ČASOVĚ PROMĚNNÝM ZDROJEM S VYUŽITÍM GPU

SIMULATION OF THE HEAT DIFFUSION WITH A TIME-VARYING SOURCE ON GPUS

BAKALÁŘSKÁ PRÁCE
BACHELOR‘S THESIS

AUTOR PRÁCE PAVEL HÁLA
AUTHOR

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

Zadání

1. Seznamte se s metodami simulace šíření tepla v heterogenním prostředí, kde je intenzita, pozice a velikost zdroje časově proměnlivá. Zaměřte se především na metody konečných prvků a konečných diferencí.
2. Prostudujte techniky efektivní implementace algoritmů na grafické kartě (GPU). Zaměřte se na parametry architektury ovlivňující výkonnost. Prostudujte rovněž techniky měření výkonnosti a efektivity.
3. Navrhněte efektivní implementaci simulace šíření tepla s časově proměnným zdrojem s ohledem na architekturu vícejádrových procesorů a grafických karet.
4. Navrženou koncepci implementujte s využitím C/C++ pro CPU a CUDA/OpenCL pro GPU.
5. Rozšiřte navrženou koncepci o možnost využití více GPU v rámci jednoho systému.
6. Porovnejte výkonnost implementací běžících na CPU a GPU pomocí běžných metrik a zhodnoťte výhody a nevýhody jednotlivých architektur.

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Jiřího Jaroše, Ph.D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Pavel Hála
21.5.2014

Poděkování

Chtěl bych poděkovat svému vedoucímu práce Jiřímu Jarošovi za jeho cenné rady, které mi pomohly při tvorbě této práce.

Abstrakt

Tato bakalářská práce se zabývá simulací šíření tepla v živých tkáních, které je dodáváno časově proměnným externím tepelným zdrojem. Simulace byla implementována pomocí metody konečných diferencí čtvrtého řádu v prostoru a prvního v čase. V rámci práce byla nejprve implementována vícevláknová verze využívající procesoru CPU. Následně bylo implementováno několik verzí pro grafickou kartu GPU s důrazem na maximální adaptaci algoritmu na danou architekturu a co nejlepší využít výpočetního potenciálu grafické karty. Experimentálním měřením se ukázalo, že nejrychlejší je naivní algoritmus využívající pouze globální paměť grafické karty. Dále byla zkoumána efektivita Gauss-Seidelovy obarvovací metody, jejíž cílem je redukce paměťové náročnosti. Na CPU se tato metoda ukázala použitelná, neboť její nejrychlejší verze byla pouze o 13% pomalejší, ale při použití této metody je možné snížit paměťovou náročnost až na polovinu. Implementace této metody na GPU byla 2x pomalejší a její přínos proto není tak velký. Na CPU bylo dosaženo maximálního výkonu 32GFLOPS zatímco na GPU 135GFLOPS. To odpovídá 10% (CPU) a 9% (GPU) maximálního teoretického výkonu obou architektur.

Abstract

This bachelor's thesis deals with the simulation of the heat transfer inside human tissue injected by an external time varying heat source. The proposed implemented simulation is based on a 4th order in space and 1st order in time finite-difference time domain method. First, a multithreaded CPU version was implemented. Subsequently, several GPU accelerated versions were implemented taking into account architecture aspect of the GPU. The experimental results showed that the fastest GPU kernel was the naive one using only the GPU global memory. Next, the usefulness of the Gauss-Seidel‘s method was investigated. The CPU implementation of the method was evaluated as usable because of being only 13% slower while saving up to 50% of memory resources. However, the GPU implementation was twice as slow as the naïve version mainly due to shared memory size limits. The peak performance in terms of GFLOPS reached 32 and 135 on CPU and GPU, respectively. This corresponds to 10% and 9% of the theoretical potential of given architectures.

Klíčová slova

šíření tepla, metoda konečných diferencí, Gauss-Seidelova metoda, CUDA, OpenMP, paralelizace

Keywords

heat transfer, finite-difference time domain method, Gauss-Seidel method, CUDA, OpenMP, parallelization

Obsah

Obsah 1
1 Úvod 2
2 Metody šíření tepla 3
2.1 Šíření tepla v lidském těle 3
2.2 Numerické metody 4
2.2.1 Metoda konečných prvků 4
2.2.2 Metoda konečných diferencí v časové doméně 4
2.2.3 Přesnost a řád metody 4
3 Knihovny pro práci s GPU 6
3.1 Rozdíl mezi CPU a GPU 6
3.2 Knihovny pro paralelizaci 6
3.2.1 OpenCL 7
3.2.2 OpenMP 7
3.2.3 OpenACC 7
3.2.4 CUDA 7
3.3 Architektura CUDA 8
3.3.1 Hardware karet NVIDIA 8
3.3.2 Kernely a vlákna 9
3.3.3 Typy pamětí 10
3.3.4 Knihovna Thrust 11
4 Návrh implementace 12
4.1 Výpočet nové teploty bodu podle okolí 12
4.2 Datový model 12
4.3 Iterační smyčka programu 14
5 Implementace 16
5.1 Implementace šíření tepla na CPU 16
5.1.1 První verze (referenční) 16
5.1.2 Cache blocking 16
5.1.3 Obarvovací metoda (Gauss-Seidel) 17
5.1.4 Zdroj tepla 18
5.2 Implementace šíření tepla na GPU 18
5.2.1 Naivní kernel 18
5.2.2 Naivní kernel – obarvovací metoda 18
5.2.3 Kernel využívající sdílenou paměť 19
5.2.4 Kernel využívající sdílenou paměť s alternativním načítáním okrajů 19
5.2.5 Kernel využívající sdílenou paměť počítající 8 bodů na vlákno 19
5.2.6 Obarvovací metoda se sdílenou pamětí 20
5.2.7 Kernel využívající texturní paměť 20
5.2.8 Kernel využívající 3D texturu 21
5.2.9 Kernel využívající surface 21
5.2.10 Kernely počítající 3 iterace 21
5.2.11 Kernely zdroje tepla 22
5.3 Implementace na více GPU 22
6 Výkonové porovnání variant 24
6.1 CPU verze 24
6.2 GPU kernely 27
7 Závěr 31

1 Úvod

Využití masivního paralelizmu pro urychlení výpočtů je horké téma poslední dekády, kdy cena hardwaru umožňující běh paralelních programů klesla na takovou úroveň, že si jej může dovolit prakticky každý.

U procesorů nešlo donekonečna zvyšovat frekvenci, neboť s frekvencí rostla kvadraticky spotřeba. Výrobci procesorů se proto rozhodli opustit koncept vysoce taktovaného procesoru s jedním jádrem a vydali se směrem vícejádrových procesorů [20].

Grafické karty byly od počátku paralelní zařízení, ale protože se jednalo o specializované zařízení, nešly využít k obecným výpočtům. Výrobci grafických karet si uvědomili potenciál svých výrobků a postupně měnili architekturu a software svých čipů tak, aby je bylo možné využít pro obecné výpočty. Dnes je toto možné a s rostoucí popularitou knihoven jako OpenCL a CUDA jednoduché[1]. Akcelerace výpočtů na GPU je atraktivní, protože nabízí maximální teoretický výkon v řádech jednotek TFLOPS[1], zatímco CPU se pohybují řádově ve stovkách GFLOPS[2]. Protože GPU nabízejí řádově vyšší výkon než CPU, jsou často lepší pro výpočetně náročné aplikace než CPU.

Tato práce se zabývá využitím GPU pro simulaci šíření tepla v živých tkáních. Šířením tepla je zde myšlen přenos teplené energie látkou. Šíření tepelné energie je způsobeno neustálým pohybem částic hmoty, kdy dochází ke srážkám částic a tím pádem dochází k předávání kinetické energie. Takováto simulace najde uplatnění v různých odvětvích průmyslu jako například zateplení domů, návrh chladičů nebo výzkum léčby rakoviny pomocí ultrazvuku, kdy jsou ultrazvukové vlny vysílány do nádoru, dokud rakovinné buňky teplem neodemřou.

[1] http://nvidianews.nvidia.com/News/NVIDIA-Introduces-GeForce-GTX-TITAN-DNA-of-the-World-s-Fastest-Supercomputer-Powered-by-World-s-Fa-925.aspx

[2] http://download.intel.com/support/processors/xeon/sb/xeon_E5-2600.pdf

2 Metody šíření tepla

Tato kapitola pojednává o šíření tepla v lidském těle a o numerických metodách, které je možné použít pro simulaci tohoto problému.

2.1 Šíření tepla v lidském těle

Tepelná energie se může šířit vedením, prouděním a zářením[21]. Tato práce se zabývá vedením (kondukcí) tepla v tělesech.

Lidské tělo je složitá struktura, kterou nelze aproximovat homogenním prostředím. Nejen že se tělo skládá z různých materiálů, musí se brát v potaz různé další okolnosti jako například žíly, které vedou teplo skrze lidskou tkáň a do systému (výřez těla) mohou vnášet nebo odebírat energii. Pro výpočet takto složitého problému použijeme Pennovu rovnici (Pennes bioheat transfer equation)[3]:

$$\rho_{ti} C_{ti} \frac{\partial T}{\partial t} = k_{ti} \nabla^2 T_{ti} - \rho_{bl} C_{bl} v_h . \nabla T_{ti} + q_m \quad (1)$$

- ρ_{ti} – hustota tkáně
- C_{ti} – měrná tepelná kapacita tkáně
- T – teplota bodu
- t – čas
- k_{ti} – součinitel tepelné vodivosti tkáně
- T_{ti} – teplota bodu
- ρ_{bl} – hustota krve
- C_{bl} – měrná tepelná kapacita krve
- v_h – rychlost průtoku krve
- q_m – externí zdroj tepla

Rovnice (1) je diferenciální rovnice. První část rovnice říká, že změna teploty v každém bodě zkoumané domény je dána druhým gradientem teploty (rychlost změny teploty) a součinitelem tepelné vodivosti (jak dobře materiál vede teplo) v daném bodě. Tato část rovnice je tzv. difúzí tepla [16] – částice materiálu do sebe naráží a tím si předávají kinetickou energii.

Druhá část rovnice říká, jaká část tepla je z domény odvedena perfúzí (prokrvením). Krev má vlastní tepelnou kapacitu na jednotku objemu a protéká určitou rychlostí. Čím větší je měrná tepelná kapacita krve, rychlost proudění krve a gradient teploty (rozdíl mezi teplotou tkáně a krve), tím více se odvede tepla.

Třetí část rovnice je tepelný zdroj.

Pro zjednodušení si tuto rovnici můžeme představit následovně: nová teplota v každém bodě zkoumané domény je rovna součtu aproximace teploty podle okolních bodů (toto je samozřejmě pravda až při aproximaci pomocí metody konečných diferencí nad doposud nedefinovaným diskrétním prostorem), tepla přivedeného/odvedeného žílami a vlivu externího zdroje, který do systému přidává energii.

V loňské práci [4], na kterou tato práce navazuje, šlo pouze o výpočet nové teploty na základě okolí (konduktivity) a statického zdroje s konstantním tepelným výkonem. Cílem této práce je vypočítat novou teplotu bodu na základě okolí a navíc i externího zdroje, jehož poloha a výkon se

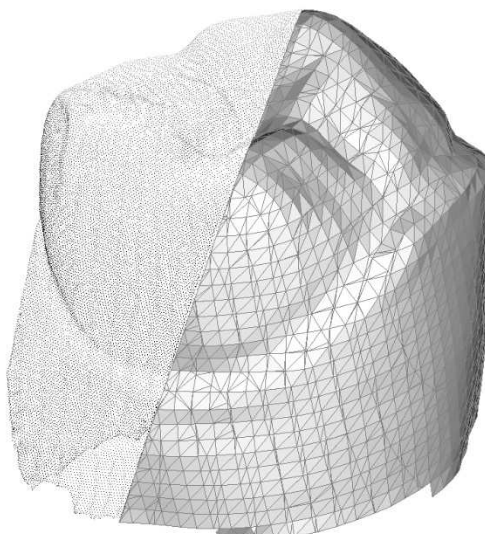
v čase mění. V loňské práci je také zmíněno, že odvod tepla žilami nemá velký vliv. Toto tvrzení však bylo vyvráceno použitím přesnějšího simulačního modelu.

2.2 Numerické metody

Tato podkapitola popisuje dvě numerické metody, které je možné použít pro simulaci šíření tepla.

2.2.1 Metoda konečných prvků

Metoda konečných prvků je numerická aproximační metoda sloužící k simulaci fyzikálních modelů včetně šíření tepla. Metoda je založená na rozdělení spojitého prostoru na konečný počet menších části. Prostor je diskretizován. Jednotlivé části mohou mít různé vlastnosti a to včetně velikosti (znázorněno na Obrázek 1), ale jedna část má v celém svém objemu stejné vlastnosti. Zjišťované parametry jsou určovány v jednotlivých uzlových bodech, ve kterých jsou jednotlivé části spojeny. Hlavní myšlenkou tedy je, že prvky nejsou rozloženy v prostoru pravidelně a v každém prvku (uzlovém bodě) se počítají dané rovnice [15].



Obrázek 1: Vizualizace rozdělení objektu na konečný počet menších částí[3]

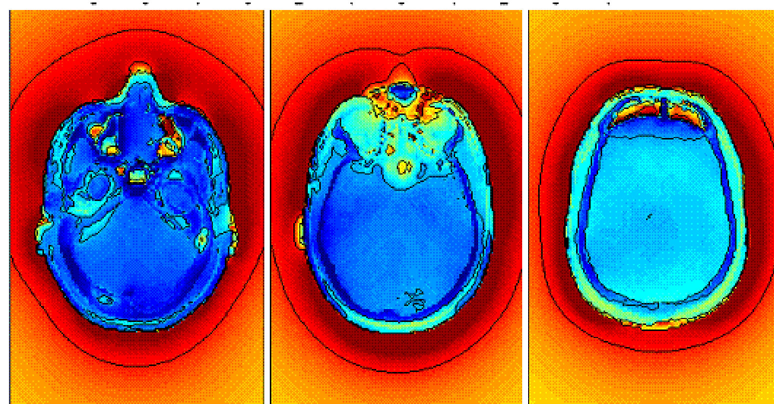
2.2.2 Metoda konečných diferencí v časové doméně

Metoda konečných diferencí v časové doméně je numerická metoda, která se používá při aproximaci řešení soustavy diferenciálních rovnic. V rámci této metody je (na rozdíl od metody konečných prvků) systém rovnoměrně rozdělen na stejně velké a pravidelné části (mřížku) viz Obrázek 2. Díky tomuto faktu je velmi jednoduché metodu paralelizovat. Při zpracování dat na GPU je vhodné, když jsou přístupy do paměti pravidelné.

2.2.3 Přesnost a řád metody

Simulace může provést velké množství iteraci (stovky až tisíce), proto je nutné, aby byl algoritmus co nejpřesnější. Zároveň je důležité, aby byl algoritmus rychlý. Proto je nutné zvolit kompromis mezi přesností a rychlostí. Proto byla vybrána metoda konečných diferencí 4. řádu, která je dostatečně přesná a která byla použita v loňské práci.

[3] http://www.final-surface.com/triangulation.php



Obrázek 2: Ukázka vzorkování do pravidelné mřížky

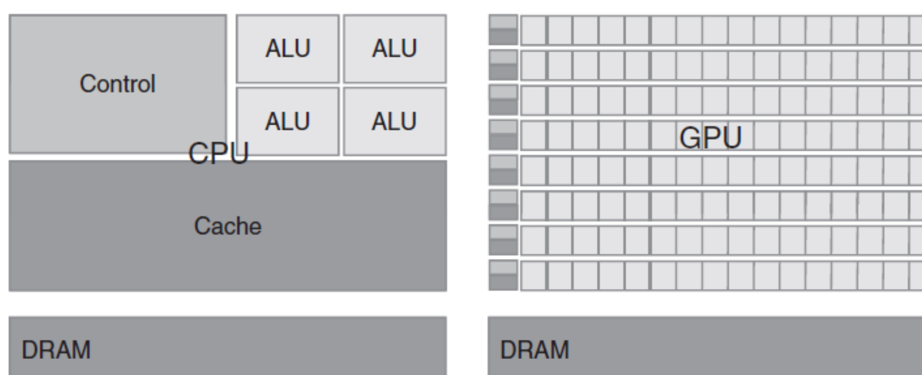
3 Knihovny pro práci s GPU

V této kapitole jsou popsány hlavní rozdíly mezi x86 architekturou dnešních CPU a architekturou grafických karet, především karty společnosti Nvidia. Dále jsou zde uvedeny knihovny sloužící k paralelizaci algoritmů a podrobně je popsána jedna z těchto knihoven – CUDA.

3.1 Rozdíl mezi CPU a GPU

Klasické CPU architektury x86 je komplexní čip pro širokou sadu úloh [7]. Pro dosažení rozumného výkonu na široké škále aplikací musí řešit složité problémy typu predikce skoků, vykonávání instrukcí mimo pořadí, správa pamětí cache v několika úrovních, několik způsobů přístupu do paměti a spoustu jiných. CPU je zaměřeno na výkon jednoho vlákna, které většinou pracuje nad jednou sadou dat. Hovoříme o architektuře Single Instruction, Single Data (SISD). Moderní procesory mohou v dnešní době mít až 16 jader [5], takže paralelizace na CPU je v dnešní době možná. Dnešní procesory podporují i instrukce typu Single Instruction Multiple Data (SIMD), kdy je několik hodnot uloženo do speciálního registru a nad všemi hodnotami je provedena stejná operace. Abychom využili maximální hrubý výkon dnešních procesorů, je třeba tyto instrukce využívat.

Dnešní GPU se skládají ze stovek až tisíců jednoduchých jader (Obrázek 3), které jsou určeny pro řešení datově paralelních úloh s minimem komunikace a větvení. GPU například nemají ochranu paměti sloužící k detekci přístupu mimo pole. Implementace této funkce je totiž poměrně složitá a má výrazný dopad na výkon. Do budoucna se počítá se zavedením (unifikované virtuální adresování [6]). Jednoduchost jednotlivých jader (jsou skalární, in-order, žádná predikce skoku, minimum paměti) umožňuje jejich masivní replikaci na čipu. GPU jsou stavěná s úmyslem využití masivní paralelizace, zejména pro počítání s čísly s plovoucí desetinnou čárkou.



Obrázek 3:Ilustrace rozdílné architektury CPU a GPU [1]

Nejedná se zcela o architekturu SIMD, ve které několik jednotek provádí stejnou instrukci nad množinou dat. Tento označení se spíše užívá u vektorových procesorů a instrukcí. Firma Nvidia označuje svoji GPU architekturu pojmem Single Instruction, Multiple Threads (SIMT). Spočívá v tom, že skupina vláken vykovává v jednu chvíli stejnou instrukci.

3.2 Knihovny pro paralelizaci

V této podkapitole jsou stručně popsány čtyři knihovny, pomocí kterých je možné paralelizovat algoritmus na vícejádrových CPU a na GPU.

3.2.1 OpenCL

Open Computing Language (OpenCL) [8] je otevřený a standardizovaný framework pro vývoj programů na velikou škálu platforem. Původně vyvíjen firmou Apple, dnes je OpenCL spravováno konsorciem Kronos, do kterého patří významné hardwarové firmy jako Nvidia, AMD, Intel, Qualcomm, IBM aj. Velikou výhodou OpenCL je jeho nezávislost na platformě a to jak z pohledu operačního systému (Windows, Linux, MacOS), tak z pohledu hardwaru – může běžet na CPU, GPU a jiných čipech. První verze jazyka vychází z C (C99), druhá verze má plnou podporu C++11.

3.2.2 OpenMP

Open Multi-Processing je soustava knihoven a direktiv překladače umožňující velmi pohodlné programování vícevláknových aplikací [10]. Podporovanými jazyky jsou C, C++ a Fortran. Pokud máme blok kódu a chceme ho provést paralelně, označením bloku kódu příslušnou direktivou překladače dosáhneme toho, že jednotlivé úlohy v rámci bloku budou provedeny paralelně (Obrázek 4).

```
1  #pragma omp parallel for num_threads(3)
2  for (int i=0; i<9; i++)
3  {
4      //9 iteraci cyklu bude
5      //rovnomerne rozdeleno mezi
6      //3 vlakna - tj. kazde vlakno
7      //provede 3 iterace cyklu
8  }
```

Obrázek 4: Příklad jednoduchosti použití OpenMP

OpenMP je multiplatformní, mezi podporované operační systémy patří Windows, Linux, MacOS, Solaris aj. Spravuje jej neziskové konsorcium OpenMP Architecture Review Board, mezi jejichž členy jsou společnosti AMD, Intel, Nvidia, IBM, TI, Fujitsu, Microsoft, Oracle aj.

3.2.3 OpenACC

Podobně jako OpenMP, OpenACC je kolekce direktiv překladači pro programování vícevláknových aplikací [9]. Hlavním cílem je snížení zátěže CPU a přesunutí výpočetně náročných bloků kódu na (grafický) akcelerátor. Podporuje stejné jazyky co OpenMP, celkově jsou tyto dvě knihovny velmi podobné.

3.2.4 CUDA

Compute Unified Device Architecture (CUDA) je proprietární technologie společnosti Nvidia. CUDA je softwarová i hardwarová architektura. Protože jde o proprietární technologii, CUDA programy lze spouštět jen na grafických kartách Nvidia. Architektura umožnuje psát programy v nadstavbách jazyků C, C++ a Fortran. Podporované operační systémy jsou Windows, Linux MacOS a Android (s Nvidia chipsetem Tegra).

CUDA a OpenCL jsou si velmi podobné a přepsat kód z jednoho do druhého není náročné. Existují i pomocné nástroje, které dokáží konvertovat již napsané kódy. Například pro převod CUDA

kódu do OpenCL existuje nástroj Swan[4]. Firma AMD má na svých stránkách podrobný návod, jak převést CUDA kód do OpenCL[5].

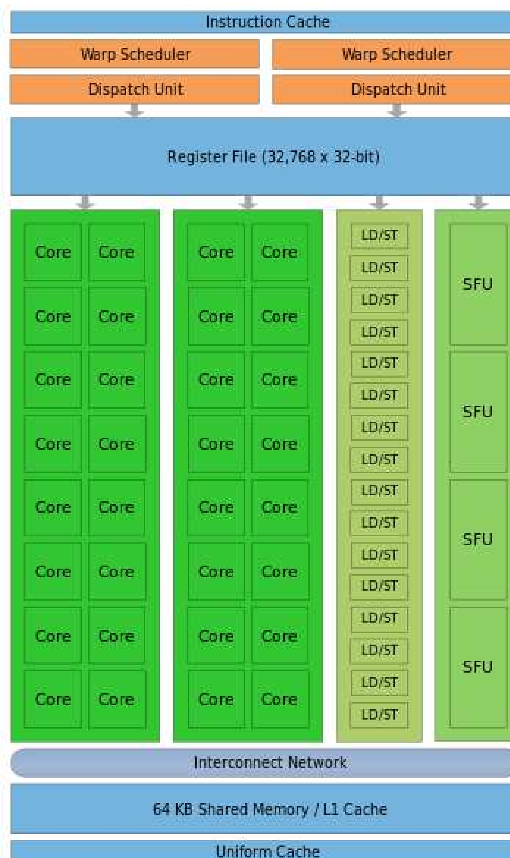
Protože je můj projekt nástavbou loňského projektu, budu používat platformu CUDA stejně jako bylo použito v loňském projektu.

3.3 Architektura CUDA

Tato podkapitola podrobněji pojednává o softwarové i hardwarové stránce knihovny CUDA.

3.3.1 Hardware karet NVIDIA

Základním stavebním prvkem grafických karet Nvidia je tzv. streaming multiprocesor (SM). Schéma SM je možné vidět na Obrázek 5. SM obsahuje několik skalárních procesorů (SP), v novějších architekturách označované jako cuda core. V rámci SM běží několik vláken (každé na jednom SP), které provádějí stejnou instrukci nad různými daty.



Obrázek 5: Streaming multiprocesor architektury Fermi [2]

Každý SP obsahuje aritmeticko-logickou jednotku pro výpočty s celými čísly (32 nebo 64 bitový integer) a matematický koprocesor pro práci s čísly s proměnlivou desetinnou čárkou (každý SP má koprocesor pro práci s 32b floaty, o 64b jednotku se jich dělí několik). Dále SM obsahuje několik special function unit (SFU), které se starají o výpočty typu dělení, sinus, cosinus, odmocnina aj. Počet SFU jednotek je menší než počet SP. Například u architektury Fermi [2] jeden SM obsahuje 32 SP a

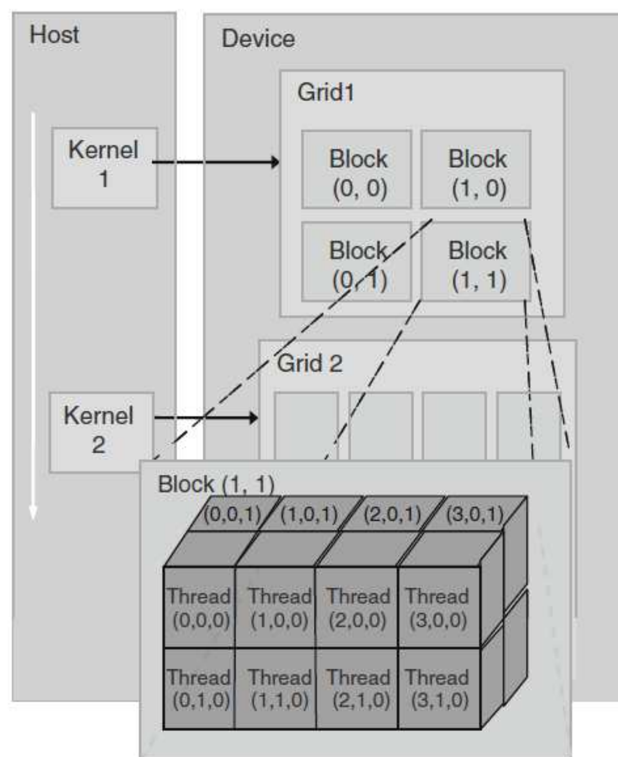
[4]http://www.multiscalelab.org/swan
[5]http://developer.amd.com/resources/heterogeneous-computing/opencl-zone/programming-in-opencl/porting-cuda-applications-to-opencl/

4 SFU. Aby mohl SM pracovat s globální pamětí grafické karty, obsahuje load/store jednotky. Dále lze v SM najít instrukční cache, registry, sdílenou paměť, L1 cache globální paměti a plánovač warpů. Jejich význam bude vysvětlen dále.

3.3.2 Kernely a vlákna

Pokud chceme spustit na procesoru velké množství vláken, využívá se speciální funkce zvaná kernel. Kernel specifikuje kód, který se bude v rámci všech vláken vykonávat. Kernelů může na jednom GPU běžet více, každý se svojí sadou instrukcí (současně pouze u architektury Kepler, u starších architektur je možné spustit zároveň pouze jeden kernel). Při volání kernelu je možné předat parametry vláknům stejně jako u volání běžné funkce v C/C++. Před voláním kernelu je nutné specifikovat, kolik vláken se má kernel vytvořit.

Vlákna v rámci jednoho kernelu vykonávají stejný kód, ale každý nad jinými daty. Proto jsou vlákna organizována do struktur v několika úrovních a podle své pozice jednoznačně identifikovatelná (Obrázek 6). Skupina vláken se sdružuje do bloku. V rámci bloku mohou být uspořádaná v 1D, 2D nebo 3D mřížce. V rámci vlákna existuje proměnná threadIdx, ve které je uložen index vlákna v rámci bloku. Tuto hodnotu vlákna použijí k identifikaci dat, nad nimiž provádí výpočet například pomocí mapovací funkce pro přístup do pole. Skupina bloků je sdružena do tzv. gridu. Bloky v rámci gridu jsou (stejně jako vlákna v bloku) uspořádány do 1D, 2D anebo v novějších verzích CUDA i ve 3D mřížkách. K identifikaci bloku v gridu slouží proměnná blockIdx [1].



Obrázek 6: Organizace vláken do bloků a gridů [1]

Vlákna jsou seskupována i na hardwarové vrstvě. Když je blok přidělen SM, je rozdělen po 32 vláknech do tzv. warpů. Pokud je warp aktivní, běží všechna jeho vlákna, každé vlákno na jednom SP. Maximální počet warpů na SM se liší u různých architektur (48 warpů u architektury Fermi, 64 u Kepler [11]).

3.3.3 Typy pamětí

Při psaní kernelů je programátorovi k dispozici několik druhů pamětí lišící se velikostí a rychlostí. Paměťová propustnost je asi nejužším hrdlem při psaní CUDA kernelů, proto je třeba volit vhodné typy pamětí pro ukládání potřebných dat.

3.3.3.1 Globální paměť

Globální paměť grafické karty je společná všem běžícím kernelům. Je založena na technologii dynamic random access memory (DRAM), proto má vysoké přístupové latence, řádově stovky cyklů. Tato paměť je největší. Pokud programátor spoléhá jen na tuto paměť, může zjistit, že využívá jen zlomek hrubého výkonu karty.

Problém s vysokou přístupovou dobou je možné částečně vyřešit tak, že úlohu rozdělíme na veliké množství vláken. Pokud pak skupina vláken čeká na data z globální paměti, plánovač warpů v SM nenechá tato vlákna plýtvat výpočetním výkonem jednotky a na jejich místě nechá běžet vlákna, která na paměť nečekají. Toto technika většinou nestačí k plnému vytížení GPU, a proto je vhodné práci s globální pamětí co nejvíce omezit.

Globální paměť má naštěstí vyrovnávací cache paměti. Starší karty měli jen L1 cache v každém SM. L1 a sdílená paměť se nachází fyzicky ve stejné paměti a poměrově se o toto místo dělí (například u architektury Kepler je možné dělení 16/48 a 48/16 KB [11]). Vyrovnávací paměť slouží pouze k přeuspořádání přístupů do paměti tak, aby se četly vždy co největší souvislé bloky (např. 128B).

Před spuštěním kernelu je vhodné si do globální paměti přesunout data z operační paměti systému, protože přístup do ní je ještě pomalejší.

3.3.3.2 Konstantní paměť

Jedná se o relativně malou paměť na zařízení (64KB) [13]. Nahrát data do této paměti lze pouze před spuštěním kernelu a při běhu kernelu z ní lze pouze číst. Rychlostně je zhruba stejně rychlá jako globální paměť, ale je ukládána v L1 cache paměti. Nejefektivněji ji lze využít, pokud všechna vlákna ve warpu přistupují k jedné hodnotě v konstantní paměti. Konstantní hodnota je pak rozhlášena všem vláknům ve warpu.

3.3.3.3 Texturní paměť

Podobně jako konstantní paměť slouží texturní paměť pouze ke čtení. Texturní paměť má vlastní cache se zajímavými vlastnostmi. Při ukládání do cache neukládá sekvenci po sobě jdoucích bajtů, ale po přístupu k bodu se ukládá do cache i jeho prostorové okolí. Tato paměť je proto vhodná, pokud skupina vláken náhodně přistupuje k blízkým datům [14].

3.3.3.4 Sdílená paměť

Tuto paměť sdílí všechna vlákna v rámci jednoho bloku a mohou z číst i do ní zapisovat. Sdílená paměť je několikanásobně rychlejší než paměť globální, proto je vhodné ji využívat jako programátorem řízenou vyrovnávací paměť pro hodnoty z globální paměti, ke kterým budeme přistupovat více než jednou nebo když hodnotu potřebuje více vláken.

3.3.3.5 Lokální paměť

Paměť se fyzicky nachází v globální paměti a používá se pro proměnné typu pole nebo když SM dojdou zdroje, kde by mohl data ukládat (př. plné registry). Každé vlákno má přístup jen ke své lokální paměti.

3.3.3.6 Registry

Registry jsou alokovány jednotlivým vláknům, každé vlákno má přístup pouze ke svým registrům. Registry jsou nejrychlejší typ paměti. Počet registrů v SM je omezen danou architekturou. Do registrů se implicitně ukládají lokální proměnné vlákna. Pokud nestačily registry v SM, u starších architektur byly přebytečné proměnné ukládány do globální paměti, u novějších se přebytečné ukládají do L1 cache.

3.3.4 Knihovna Thrust

Knihovna Thrust je součást CUDA Toolkitu. Skládá se z C++ šablon založených na Standard Template Library (STL) [12]. Tato knihovna šablon velmi zjednodušuje programátorovu práci při psaní komplexních úloh.

Thrust obsahuje kontejner pro práci s vektory. S vektory se pracuje stejně jako s `std::vector`, takže tyto generické kontejnery mohou obsahovat libovolné datové typy a také je možné je dynamicky zvětšovat nebo zmenšovat.

Dále knihovna obsahuje soubor mnoha běžných algoritmů napsaných v CUDA a jejichž ekvivalent většinou existuje v STL. Nabízí funkce transformace, redukce, řazení aj.

4 Návrh implementace

Šíření tepla budeme simulovat tak, že novou teplotu bodu v matici vypočítáme podle stávající teploty bodu, teploty okolních bodů a fyzikální vlastnosti všech těchto bodů. Dále v simulaci musíme počítat se zdrojem tepla, který zvyšuje celkovou energii systému.

4.1 Výpočet nové teploty bodu podle okolí

Pro výpočet teploty bodu byla zvolena metoda konečných diferencí. V prostoru budeme využívat metodu 4. řádu (4 okolí), zatímco dopřednou integraci v čase budeme provádět pouze na základě předchozího času. Druhá derivace obecné funkce s krokem *s* vypadá následovně:

$$f'' \approx \frac{f(x-2s) - 16f(x-1s) + 30f(x) - 16f(x+s) + f(x+2s)}{12s} \tag{2}$$

Novou teplotu bodu v jedné ose tedy vypočítáme:

$$\frac{\delta^2 T(x,t)}{\delta x^2} \approx \frac{1}{12} * \frac{(T_{i-2}^n - 16T_{i-1}^n + 30T_i^n - 16T_{i+1}^n + T_{i+2}^n)}{(\Delta x)^2} \tag{3}$$

Výsledný vztah, který bude počítat teplotu v 3D prostoru bude vypadat takto:

$$\begin{aligned}\nabla^2 T(x,t) \approx \frac{1}{12} * \Bigg(& \frac{T_{i-2,j,k}^n - 16T_{i-1,j,k}^n + 30T_{i,j,k}^n - 16T_{i+1,j,k}^n + T_{i+2,j,k}^n}{(\Delta x)^2} \\ & + \frac{T_{i,j-2,k}^n - 16T_{i,j-1,k}^n + 30T_{i,j,k}^n - 16T_{i,j+1,k}^n + T_{i,j+2,k}^n}{(\Delta y)^2} \\ & + \frac{T_{i,j,k-2}^n - 16T_{i,j,k-1}^n + 30T_{i,j,k}^n - 16T_{i,j,k+1}^n + T_{i,j,k+2}^n}{(\Delta z)^2} \Bigg)\end{aligned} \tag{4}$$

4.2 Datový model

Protože starší verze CUDA knihoven nepodporovaly přesun složitějších struktur do paměti zařízení, všechny 2D a 3D matice jsou v programu reprezentovány 1D polem. Přístup do různých dimenzí se řeší vlastní mapovací funkcí.

Vstupní data jsou uložena v souboru ve formátu HDF5, ta jsou načtena do programu pomocí existujících knihoven. HDF5 formát byl zvolen proto, že data je pak dále možné zpracovat například pomocí MATLABu nebo jiného programu, který tento formát podporuje. Mezi vstupní data patří počáteční teplota systému ve všech bodech, fyzikální vlastnosti všech bodů v systému a adresy bodů, které body budou v průběhu simulace zahřívány externím zdrojem a velikost energie kterou tento zdroj do systému v jednotlivých bodech v čase vloží.

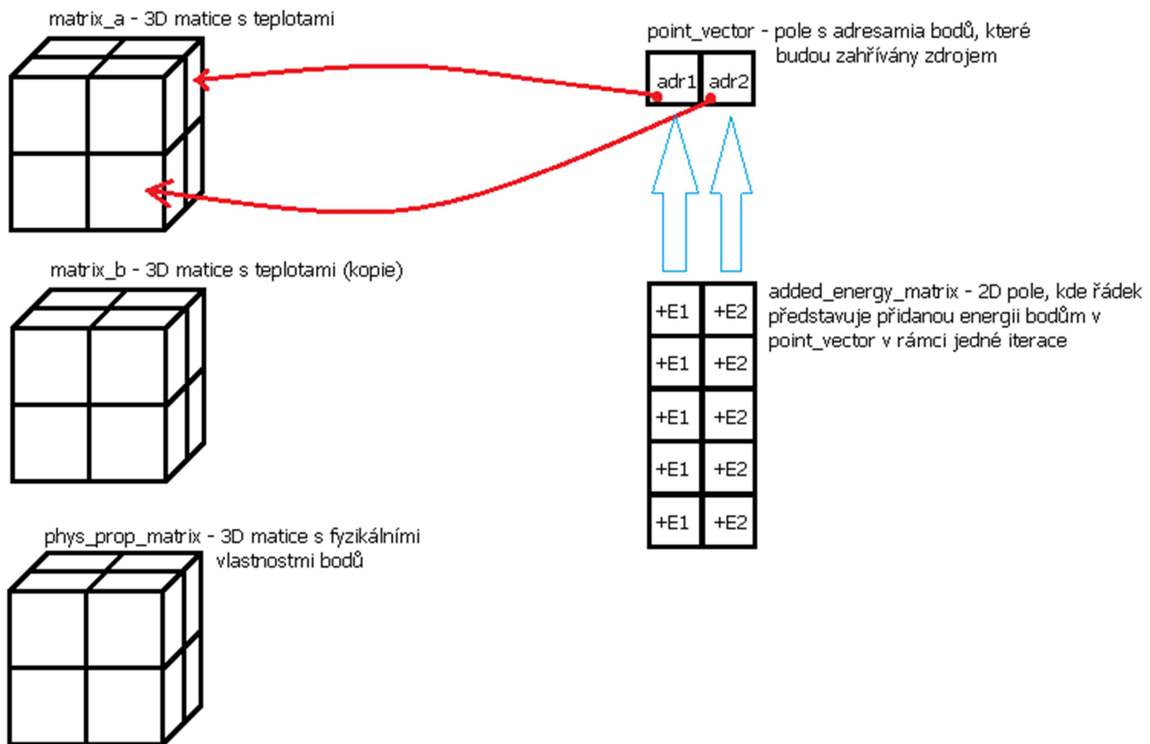
Obrázek 7 popisuje datový model implementace. Pro uchování teplot a výpočet nových teplot jsou použity dvě stejně velké 3D matice (zdrojová a cílová), které jsou v programu reprezentovány 1D polem.

V prvních jednoduchých verzích simulace, kdy se počítá s homogenním prostředím, není potřeba fyzikálních vlastností jednotlivých bodů, protože všechny body mají vlastnosti stejné. V pokročilejších verzích ale budou nezbytné. Fyzikální vlastnosti jednoho bodu je možné sloučit do

jedné hodnoty (součin tepelné vodivosti tkáně vydělený součinem měrné tepelné kapacity a hustoty). Proto matice s fyzikálními vlastnostmi bude stejně jako matice s teplotami 1D pole reprezentující 3D matici.

Tepelný zdroj je reprezentován dvěma poli. První je seznam všech bodů (respektive jejich adres), které budou v průběhu simulace zahřívány. Druhé pole je matice přírůstku energie v bodech v různých časech. Velikost matice je počet zahřívaných bodů krát počet iterací, kdy je zdroj aktivní.

Pokud se ale bude zahřívat větší množství bodů, například kdyby se zahřívala celá matice o velikosti 512^3 bodů, matice s přidanou energií by byla velká 1GB krát počet iterací, kdy je zdroj aktivní. Pro načtení celé matice by nemusela stačit paměť počítače. Tento problém je možné řešit postupným načítáním dat zdroje při běhu programu.

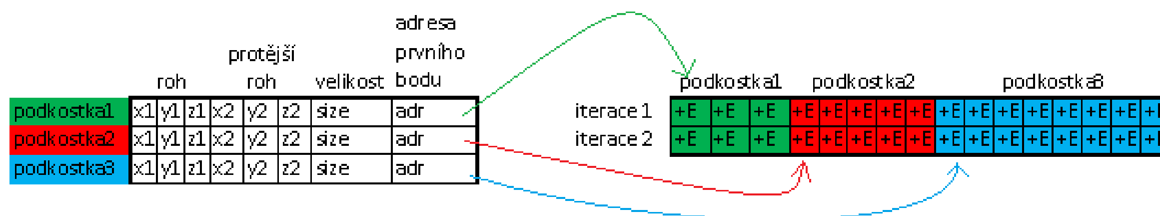


Obrázek 7: Datový model

Při implementaci pokročilejších funkcí šíření tepla na GPU se ukázal model tepelného zdroje popsaného výše jako nevhodný a pro použití v těchto funkcích velmi pomalý. Každé vlákno, které má za úkol vypočítat teplotu (většinou jen) pro jeden bod, by muselo projet celý seznam zahřívaných bodů a zjistit jestli je bod zahříván. Takovýto průchod by do kernelu vložil velké množství podmínek, které by výrazně zpomalily algoritmus. Proto byl navrhnut alternativní datový model (Obrázek 8) pro proměnlivý zdroj tepla. Místo adres jednotlivých zahřívaných bodů je v alternativním modelu seznam zahřívaných podkrychlí a pole přidané energie v jednotlivých bodech, které se nachází v těchto podkrychlích. Vlákno tedy pouze kontroluje, jestli se nenachází v jedné z podkrychlí, kterých je většinou méně. Datový model alternativního zdroje tepla je znázorněn na Obrázek 8. Oba dva modely jsou ekvivalentní a lze jimi popsat stejný zdroj.

Jedna podkrychle je definovaná následující strukturou: tři integery pro souřadnice jednoho rohu (*x*, *y*, *z*), další tři pro souřadnice protějšího rohu, počet bodů v podkrychli a adresa prvního bodu podkrychle v matici přidané energie. Seznam všech podkrychlí je ve výsledku pole struktur nahraných do konstantní paměti pro rychlý přístup (GPU verze). Matice přidané energie v jednotlivých bodech zůstala bez změny.

Odvod či přívod tepla žílami je možné simulovat přidáním druhého zdroje se stejnou datovou strukturou jednoho z modelů. Ochlazení bude vykonáno zápornou hodnotou v matici přidané energie.



Obrázek 8: Datový model alternativního tepelného zdroje

4.3 Iterační smyčka programu

Iterační smyčku programu (Obrázek 9) je možné rozdělit na 3 hlavní části: funkci pro výměnu tepla mezi body, prohození ukazatelů vstupní a výstupní matice a poslední část – zahřátí bodů zdrojem. V návrhu není ukončující podmínka pro cyklus, cyklus proběhne pevným počtem iterací.

V první části smyčky je spuštěna funkce pro výpočet šíření tepla v 3D matici. Funkce bude později nahrazena GPU kernelem plnící stejnou funkci. Funkce prochází bod po bodu zdrojovou maticí (`matrix_a`) a na základě 12ti okolí (sousední dva body v obou směrech na každé ze tří os) vypočítá novou hodnotu a tu uloží do cílové matice (`matrix_b`). Nová teplota bodu se vypočítá pomocí vzorce popsaného výše.

```
1   for (int i=0; i<ITERATIONS; i++)
2   {   //1. sireni tepla
3       heat_exchange(matrix_a, matrix_b, phys_prop_matrix);
4
5       //2. prohozeni matic
6       exchange=matrix_a;
7       matrix_a=matrix_b;
8       matrix_b=exchange;
9
10      //3. pridani energie ze zdroje do systemu
11      heat_generator(matrix_a, i, point_vector, added_energy_matrix);
12  }
```

Obrázek 9: Struktura iterační smyčky programu

Pokud by se matice sekvenčně procházela a byla by jen 2D, u každého bodu by se muselo provést 8 kontrol, jak daleko je bod od kraje, aby při přístupu k okolním bodům náhodou nečetlo mimo pole. Toto je velké zpomalení na CPU, ale na GPU by takové velké množství podmínek mělo drtivý dopad. Proto jsou v každé ose na kraji matice přidány statické okraje - dvě vrstvy bodů, u kterých se nová teplota nepočítá. Tím odpadnou podmínky a urychlí se celý algoritmus. Nevýhodou je, že nad těmito body není volána funkce pro simulaci šíření tepla, pouze jsou načítány pro výpočet nové teploty blízkých bodů a body se proto chovají jako chladič systému. Ukázalo se, že při čtení se kvůli statickým okrajům nečte zarovnaně, takže způsobí jisté zpomalení.

V druhé fázi se prohodí ukazatele na cílovou (`matrix_b`) a zdrojovou (`matrix_a`) matici pomocí pomocného ukazatele.

V závěrečné třetí fázi je spuštěna funkce pro simulaci zdroje tepla. Stejně jako u první fáze bude třetí fáze později funkce nahrazena kernelem se stejnou funkčností. Funkce postupně prochází

všechny body, které jsou uloženy v poli s adresami zahřívaných bodů (point_vector). Do všech bodů přidá energii, jejíž množství je uloženo v 2D matici added_energy_matrix.

5 Implementace

V této kapitole jsou popsány varianty implementace šíření tepla a tepelného zdroje na CPU a GPU. Nejprve byl implementován referenční algoritmus šíření tepla na CPU, podle kterého byly implementovány všechny ostatní verze. Některé se snažily urychlit algoritmus, jiné šetřit pamětí na úkor zpomalení.

5.1 Implementace šíření tepla na CPU

Tato kapitola pojednává o třech funkcích pro výpočet šíření tepla a o dvou funkcích simulující zdroj tepla.

5.1.1 První verze (referenční)

Nejdříve byla implementována jednoduchá funkce šíření tepla ve 2D homogenním prostředí, která počítala novou teplotu pomocí 4-okolí a bodu samotného. Tato funkce byla dále vylepšena změnou výpočtu nové teploty. Místo 4-okolí se počítalo s dvěma body nad, pod nalevo a napravo. Problém u této funkce byl ten, že se počítala nová teplota pro všechny body včetně krajních, takže se v rámci algoritmu muselo kontrolovat, jestli při prozkoumávání okolí nepřistupujeme mimo pole. V tomto případě to bylo 8 podmínek. Tento problém a jeho řešení je popsáno výše. Z každé strany matice se přidaly dvě vrstvy hodnot, nad kterými se výpočet nové teploty neprovádí, tudíž odpadnou všechny kontroly, ale tyto body jsou zároveň chladičem systému. Nakonec byl implementován výpočet nové teploty v 3D prostředí.

V rámci tří zanořených cyklu (každý pro jednu dimenzi) se prochází celá matice. Uvnitř nejhlouběji zanořeného cyklu se pomocí mapovací funkce vypočítá adresa bodu do jednorozměrného pole, pro který se bude počítat nová teplota. Poté se vypočítá nová teplota pomocí 12-okolí ze zdrojové matice a tato hodnota se zapíše do cílové matice. Velikost matice, nad kterou se počítá, může být u této metody libovolně veliká, pokud se vejde do paměti.

Tato funkce slouží jako referenční při implementaci všech dalších verzí i jako kontrola, že ostatní funkce a kernely správně fungují.

5.1.2 Cache blocking

Cache blocking metoda (známá také jako „loop tiling“) je optimalizační metoda průchodu velikým polem, kde je snaha co nejlépe využít všechny vyrovnávací paměti mezi operační pamětí a procesorem a minimalizovat cache miss[19].

Při implementaci této funkce šíření tepla byla brána v potaz L3, L2 a L1 vyrovnávací paměť. Původní velká matice je virtuálně rozsekána na menší submatice, které se vejdou do L3. Tato menší submatice je opět rozsekána na ještě menší submatice, které se vejdou do L2, a ty se dále rozsekají tak, aby se vešly so L1. V rámci submatice v L1 paměti se teprve počítá šíření tepla stejným způsobem jako v referenční verzi.

„Virtuálním rozsekáním“ se zde myslí pouze pořadí v jakém jsou body v matici procházeny tak, aby se zachovala co největší prostorová lokalita. Velikost těchto submatic byla vypočítaná na základě velikosti příslušných vyrovnávacích pamětí na testovaném procesoru, proto má tato optimalizace smysl jen když víme, na jakém hardware bude program běžet (což je možné zjistit za běhu).

Funkce je implementována pomocí dvanácti zanořených cyklů (3 dimenze krát 3 úrovně, 3 dimenze pro normální průchod nejmenší submaticí). Uvnitř nejhlubšího cyklu se musí komplikovaně

sestavit adresa. Adresa se v každé dimenzi vypočítá podle toho, která submatice byla načtena do které vyrovnávací paměti, což by mohlo celkem vážně zpomalit implementaci této metody.

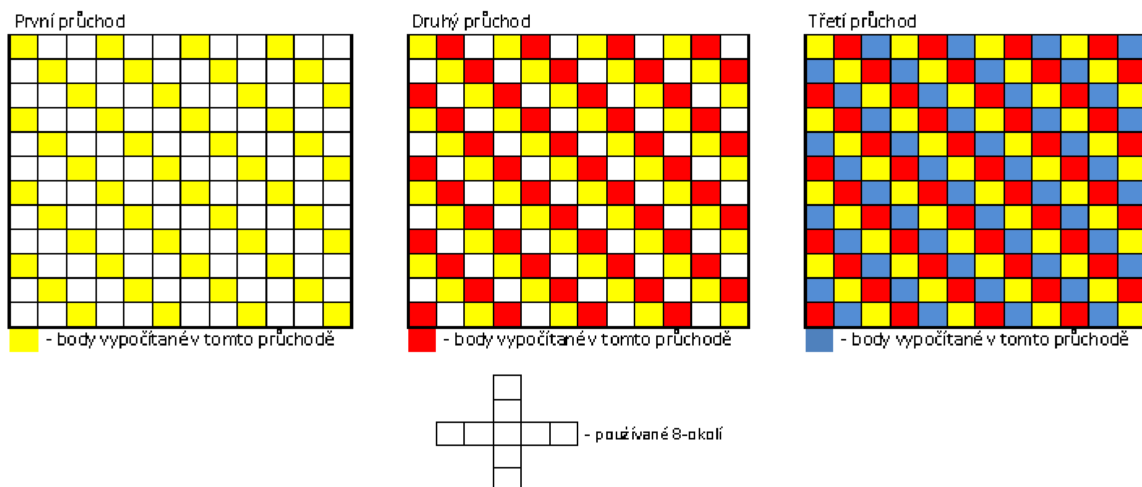
Metoda tolik neurychlí matice, které jsou menší než několikanásobek největší vyrovnávací paměti, což v našem případě není problém, protože algoritmus je testován na maticích až o velikosti 512^3 (2x4x512^3 = 1GB vs 8MB L3 v Intel Core i7 920 na kterém byl program testován). U menších matic dojde k menšímu urychlení, neboť se bude dělat cache blocking pouze na L1 a L2.

Aby tato funkce počítala novou teplotu všech bodů v matici, musí být velikost matice ve všech osách násobkem velikosti submatice v L3 (pro Intel Core i7 920 byla zvolena velikost 128x64x64 (X*Y*Z)).

5.1.3 Obarvovací metoda (Gauss-Seidel)

Při implementaci této funkce byla použita obarvovací metoda vycházející z Gauss-Seidelovy metody pro výpočet lineárních rovnic [22]. Obrovskou výhodou této metody je, že není potřeba dvou matic (zdrojové a cílové), ale stačí pouze jedna, což výrazně sníží paměťovou náročnost programu. Protože máme jenom jednu matici, nová teplota bodu se částečně počítá z již vypočítaných hodnot v rámci té samé iterace a částečně z bodů vypočítavých v iteraci předchozí. I přes tuto vlastnost metoda konverguje.

V rámci jedné iterace je nová teplota bodu vypočítaná právě jednou pro každý bod v matici, ale výpočet je rozdělen na průchody. V rámci jednoho průchodu se vypočítá nová teplota pro maximální možný počet bodů. Aby mohla být vypočítaná nová teplota bodu v průchodu, nesmí se v okolí, ze kterého se nová teplota počítá nacházet bod, pro který již byla nová teplota v tom samém průchodu vypočítána (Obrázek 10).



Obrázek 10: Demonstrace jedné iterace obarvovací metody na 2D matici

Vzhledem k tomu jaké je použito okolí pro výpočet nové teploty jednoho bodu, stačí tři průchody/barvy. Referenční funkci stačilo lehce upravit. Tři zanořené cykly se vnoří do jednoho dalšího cyklu, který určí o, jaký průchod se jedná. Cyklus pro dimenzi x se upraví tak, že v rámci průchodu se bude počítat každý třetí bod. U mapovací funkce nastává malý problém. Je třeba zajistit správné počáteční posunutí v ose x (0 – 2 bodů) na základě toho kde se nacházíme v ose y a z a o jaký se jedná průchod, aby bylo splněno výše zmíněné pravidlo. Posunutí lze vypočítat pomocí operace modulo, což by mohlo zpomalit tuto funkci.

V prvních verzích algoritmu bylo nutné, aby byla velikost matice v ose x dělitelná třemi, aby tato funkce počítala novou teplotu všech bodů v matici. Tento problém byl naštěstí vyřešen optimalizací

výpočtu počátečního posunu v ose *x* a průchodu osou *x*. Ve finální verzi této funkce může být velikost matice libovolná.

5.1.4 Zdroj tepla

Implementovat původní zdroj tepla nebylo složité. V rámci jednoho cyklu se projde seznam všech zahřívaných bodů a podle toho o kolikátou iteraci se jedná, se přičte na danou adresu energie z matice přírůstku energie.

Alternativní zdroj tepla se skládá ze čtyř zanořených cyklů. Nejvyšší určuje, kterou podkostku bude algoritmus zahřívat, další tři cykly se pohybují v zahřívané matici od jednoho rohu podkostky k protějšímu. Adresa zahřívaného bodu se vypočítá pomocí mapovací funkce do matice. Podle toho ve které iteraci jsme, o kterou podkostku se jedná a o který bod v podkostce je zahříván, se vypočítá adresa do seznamu přidané energie. Výpočet velice usnadní adresa prvního bodu ve struktuře definující podkostku.

5.2 Implementace šíření tepla na GPU

V této podkapitole je popsáno 11 kernelů, které simulují šíření tepla s různým stupněm optimalizace. Nakonec jsou popsány kernely simulující zdroj tepla.

5.2.1 Naivní kernel

Tento kernel je nazván naivní, protože nevyužívá žádných optimalizačních technik, které programování v CUDA nabízí. Oproti referenční funkci jsou odstraněny tři zanořené cykly a kernel je zavolán tak, že každé vlákno počítá novou teplotu právě jednoho bodu. Adresa bodu, pro který se má vypočítat nová teplota, se vypočítá pomocí blockIdx a threadIdx v každé dimenzi. Velikost bloku tohoto kernelu může být libovolná, ale je vhodné, aby velikost bloku v každé dimenzi byla dělitelem velikosti matice v dané dimenzi (bez statických okrajů). Velikost bloku samozřejmě nesmí překročit mez danou architekturou GPU (Fermi může mít maximálně 1024 vláken na blok).

Před spuštěním simulace na GPU je potřeba zkopírovat zdrojovou a cílovou matici do globální paměti GPU pro rychlejší přístup. To platí skoro pro všechny ostatní kernely. Tento kernel využívá pouze globální paměti, což by mohlo kernel velmi zpomalit.

5.2.2 Naivní kernel – obarvovací metoda

Tento kernel používá obarvovací metodu, takže nepotřebuje zdrojovou a cílovou matici, stačí mu jedna. Úspora paměti na GPU, které mají řádově jednotky GB paměti, je ještě výraznější. Ani tento kernel nepoužívá žádných optimalizačních metod a pracuje pouze s globální pamětí.

Nevýhodou tohoto kernelu je to, že se musí v rámci jedné iterace spustit třikrát – jednou pro každý průchod. V rámci jednoho kernelu (=průchodu) se vypočítá nová teplota třetiny bodů v matici. Toto je bohužel nutné, neboť v rámci kernelu nelze synchronizovat všechna vlákna. Taková synchronizace je možná pouze na úrovni bloků. Spuštění kernelu sebou nese jistou režii, což by mohlo lehce zpomalit výpočet.

Další problém je ten, že sousední vlákna přistupují do matice s rozestupem tři. Globální paměť je nejrychlejší, když všechna vlákna ve warpu (32 vláken) přistupují do sousedních 32 hodnot. Takovéto čtení z paměti je vyřízeno jako jeden požadavek[1]. Kvůli nevhodnému čtení by mohlo u tohoto kernelu dojít k dalšímu zpomalení. Tento problém by bylo možné vyřešit transformací matice tak, aby byly body zpracovávané ve stejném průchodu u sebe.

Velké zpomalení tohoto kernelu by mohl způsobit výpočet adresy do matice, kdy je potřeba operace modulo. Operaci modulo může vykonat pouze SFU jednotka, která je pouze jedna na každých osm cuda core. Zde by mohlo dojít k dalšímu zpomalení kernelu.

Pokud by byl kernel příliš pomalý, problémy spojené s rychlostí by mohly zcela zastínit výhodu úspory místa.

Pro velikost bloku platí stejné pravidlo jako u předchozího kernelu (blok může být jakkoli velký), pouze musí splnit podmínku, že trojnásobek velikosti bloku v ose x musí být dělitelem velikosti matice v ose x (bez statických okrajů).

5.2.3 Kernel využívající sdílenou paměť

Po implementaci naivních kernelů bylo třeba optimalizovat, proto tento kernel využívá sdílenou paměť. Místo toho aby každé vlákno vykonalo 13 přístupů do globální paměti, každé vlákno v rámci bloku zkopíruje hodnotu bodu ze zdrojové matice v globální paměti do submatice uložené ve sdílené paměti. Vlákna, která načítají body na kraji submatice, musí zkopírovat okraje (další čtyři body). Po načtení všech hodnot do sdílené paměti dojde k synchronizaci všech vláken v bloku (všechna vlákna musí načíst potřebné hodnoty do sdílené paměti, než se může s touto pamětí počítat dále). Nová teplota bodu se počítá podle okolí uloženého ve sdílené paměti, která je mnohem rychlejší.

Co by mohlo tento kernel zpomalit je načítání okrajů, které vnáší do kernelu tři podmínky, které způsobí „warp divergency“ (vlákno nebo skupina vláken ve warpu vykonává jinou větev programu než zbytek). Na druhou stranu jedno vlákno načítá maximálně pět hodnot z globální paměti, přičemž většina z nich načítá pouze jednu hodnotu, což by mohlo kernel výrazně zrychlit oprati naivnímu kernelu.

I tento kernel (stejně jako první) může mít libovolně velký blok, pokud je jeho velikost dělitelem velikosti matice v dané ose.

5.2.4 Kernel využívající sdílenou paměť s alternativním načítáním okrajů

Předchozí kernel nenačítal vhodně statické okraje do sdílené paměti. Vlákna s nulovým indexem v dimenzi načítají okraje v dané dimenzi. V tomto kernelu se načítají okraje podle toho, jaký *z* index vlákno má. Názorně je tento přístup ukázán na Obrázek 11. Toto alternativní načítání okrajů by mělo způsobit menší počet divergencí kvůli podmínkám. Co by ale tuto metodu mohlo zpomalit ve výpočet adresy okrajů podle z indexu. U předchozího kernelu stačilo zmenšit/zvětšit adresu o daný offset pro získání adresy okraje.

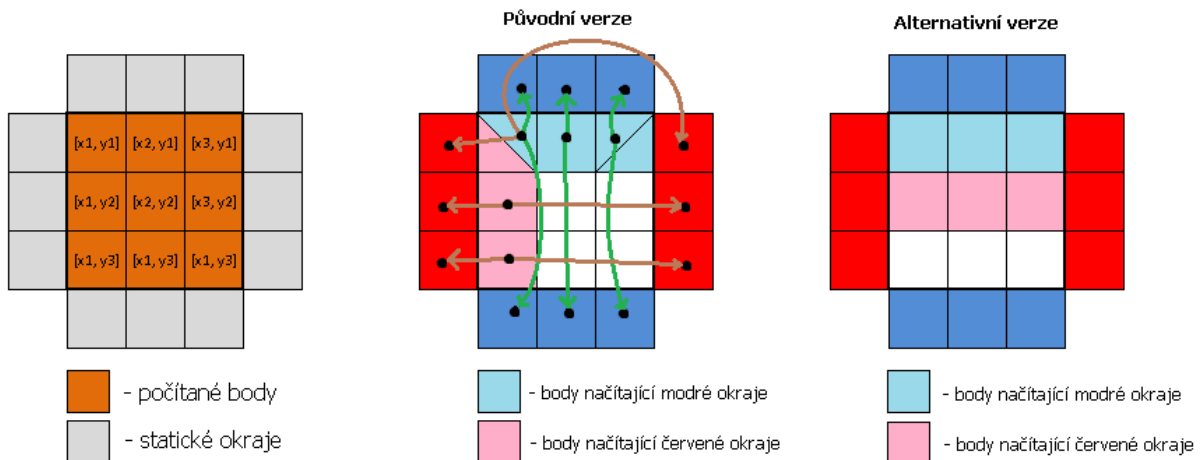
Velikost bloku musí být v každé ose stejná, aby se správně načetly všechny okraje. Jinak na velikosti bloku nezáleží. Kernel byl spouštěn s velikostí bloku 8x8x8 a velikost submatice ve sdílené paměti byla 12x12x12 (6912 B), což by mohlo vést k nezarovnaným přístupům do paměti (nečte se 32 hodnot vedle sebe v paměti).

5.2.5 Kernel využívající sdílenou paměť počítající 8 bodů na vlákno

Sdílená paměť je poměrně malá (Fermi 48KB), ale v předchozím kernelu se její velikost moc nevyužila. Submatice ve sdílené paměti měla velikost 5 – 10 KB v závislosti na velikosti bloku. V rámci tohoto kernelu je submatice ve sdílené paměti zvětšena dvakrát v každém rozměru, čímž se dostaneme na 32KB (pevná velikost bloku pro tento kernel). Jedno vlákno tedy načte 8 hodnot do sdílené paměti (plus okraje) a vypočítá novou teplotu pro 8 bodů (malou subkrychli $2 \times 2 \times 2$). Díky

tomu že se zvětší poměr hodnot v submatici vůči statickým okrajům ($12^3/8^3 = 3,375$ vs. $20^3/16^3 = 1,953$), mohlo by dojít k menšímu zrychlení kernelu.

Pro tento kernel byla zvolena statická velikost bloku 8^3 vláken, aby bylo zajištěno optimální zaplnění sdílené paměti (submatice ve sdílené paměti bude veliká 32KB).



Obrázek 11: názorná ukázka alternativního načítání okrajů na zjednodušeném 2D modelu

5.2.6 Obarvovací metoda se sdílenou pamětí

Protože naivní kernel využívající obarvovací metodu je pomalý, byl napsán tento kernel jako snaha metodu urychlit s využitím sdílené paměti. Jak bylo napsáno výše, všechny vlákna kernelu nelze synchronizovat, toto je možné pouze na úrovni bloku. Tento kernel ale není nutné pouštět jednou pro každý průchod iterace, stačí ho pustit jednou za iteraci. Tohoto je dosaženo tím, že průchody jsou synchronizovány na úrovni bloků, ale kvůli tomu přibyla nutnost mít dvě matice (zdrojovou a cílovou).

Všechna vlákna v rámci bloku načtou submatici do sdílené paměti. Poté dojde k synchronizaci. Provede se první průchod, synchronizace, druhý průchod, synchronizace, třetí průchod, synchronizace a nakonec se nové hodnoty uloží do cílové matice.

Tento kernel je vyloženě experimentální. Nemá výhodu nižší paměťové náročnosti a nikdy nečte souvislý blok paměti z globální paměti v rámci warpu. Očekávaná rychlost je mezi naivním odbarvovacím kernelem a kernely využívající sdílenou paměť co nepoužívají obarvovací metodu.

Kernel počítá s velikostí bloku 4x8x8 (X*Y*Z) vláken, velikost submatice ve sdílené paměti je 12x8x8.

5.2.7 Kernel využívající texturní paměť

Další pokus o optimalizaci využívá texturní paměti pro zdrojovou matici. To by mělo pomoci při načítání okolí, neboť texturní paměť ukládá do cache prostorové okolí. Zdrojová matice v globální paměti byla namapována na 1D texturu pomocí příkazu `bind`. Kernel vypadá téměř identicky jako naivní kromě toho, že hodnoty nečte z globální, ale z texturní paměti. Mezi iteracemi je třeba odmapovat texturu ze zdrojové matice, prohodit pointery mezi cílovou a zdrojovou a zase namapovat texturu na zdrojovou.

Namapovat 1D texturu na 1D pole je možné jen pro pole o maximální velikosti 31768 hodnot[17], což nám dá zhruba krychli o hraně 31 hodnot. Navíc 1D textura nebude do cache paměti ukládat prostorové okolí, ale pouze okolí v ose *x*.

Velikost bloku kernelu může být libovolná jako u naivního kernelu.

5.2.8 Kernel využívající 3D texturu

Předchozí kernel byl nepoužitelný pro větší matice, což řeší tento kernel. Abychom mohli použít 3D texturu, nemůžeme ji namapovat na pole v globální paměti. K tomu slouží speciální druh pole v CUDA zvaný `cudaArray`[18]. `CudaArray` je datové pole optimalizované pro namapování na texturní paměť a pro rychlé čtení. Do `cudaArray` nelze přímo z GPU zapisovat, ale umožňuje velikost matice až 4096^3.

Kernel vypadá téměř identicky jako předchozí, pouze pro čtení z texturní paměti se použije příkaz pro čtení ze 3D texturní paměti místo čtení z 1D texturní paměti.

Použití `cudaArray` sice umožňuje použití velkých matic, ale pro nemožnost zápisu do `cudaArray` je také nevhodný. To proto, že mezi iteracemi se musí zkopírovat hodnoty z cílové matice do zdrojového `cudaArray`, což je obzvláště u velkých matic časově náročné.

I tento kernel může mít libovolně velký blok stejně jako naivní kernel.

5.2.9 Kernel využívající surface

CUDA přímo neumožňuje zapisovat do `cudaArray`. Když se ale `cudaArray` namapuje pomocí na CUDA objekt `surface`, přes `surface` je možné do `cudaArray` zapisovat. Kernel tedy nepracuje se zdrojovou a cílovou maticí v globální paměti, ale se zdrojovým a cílovým `surface`. Mezi iteracemi je třeba prohodit ukazatele na `surface`. Tento kernel má předpoklady být nejrychlejší ze tří kernelů využívající texturní paměť, protože umožňuje použití velkých matic a nemusí mezi iteracemi kopírovat hodnoty.

Stejně jako ostatní kernely pracující s texturní pamětí tento kernel může být spouštěn s libovolně velkým blokem.

5.2.10 Kernely počítající 3 iterace

Tento kernel představuje finální pokus co nejvíce využít sdílenou paměť a omezit načítání hodnot z pomalé globální paměti. Toho bylo dosaženo tím, že kernel počítá tři iterace, ale hodnoty k tomu potřebné si načte do sdílené paměti před první iterací a pak už čte pouze ze sdílené paměti. Aby blok vláken mohl počítat tři iterace, musí si do sdílené paměti načíst větší okolí než normálně. Pro každou iteraci je třeba zvětšit submatici v každé dimenzi o 4 hodnoty (ve výsledku v každé dimenzi o 12 více hodnot). Zároveň je nutné mít ve sdílené paměti zdrojovou a cílovou submatici, což se zvýší paměťové nároky na sdílenou paměť. Ostatní kernely nepočítají novou teplotu v krajních dvou vrstvách matice, tento kernel nepočítá krajních vrstev šest, proto kernel nemá stejnou výstupní matici jako ostatní kernely.

Pro maximální využití sdílené paměti byla zvolena velikost bloku 8x4x4 vláken, což je zároveň velikost výstupní submatice bloku. Takto velký blok ale musí načíst submatici o velikosti 20x16x16 kvůli okolí pro každou iteraci. Cílová a zdrojová matice tak dohromady zabere 40,96KB sdílené paměti (ze 48KB).

Každé vlákno v bloku musí načíst 18 hodnot z globální paměti do sdílené, každé okrajové vlákno bloku musí načíst ještě dalších 36 hodnot (poslední dvě vrstvy statických okrajů). Po načtení všech hodnot dojde k synchronizaci vláken v bloku. Poté se v rámci první iterace vypočítá nová hodnota všech bodů v submatici (jedno vlákno počítá 18 bodů) ze zdrojové submatice do cílové submatice ve sdílené paměti. Opět dojde k synchronizaci, prohodí se pointery na submatice a proces se opakuje pro druhou a třetí iteraci.

I když se tento kernel snaží co nejvíce využit sdílenou paměť, obrovská režie by mohla tento kernel velkým způsobem zpomalit. Blok musí načíst matici o velikosti 5120 hodnot, ale výstupní

matice je veliká pouze 128 hodnot (poměr 40:1). Pro výpočet jednoho výstupního bodu musí vlákno vypočítat 18 bodů (jedna iterace).

Při snaze zmenšit tuto obrovskou režii byla implementována alternativní verze tohoto kernelu využívající obarvovací metodu. Tento kernel nemá ve sdílené paměti zdrojovou a cílovou submatici, ale jen jednu submatici. Díky uvolněnému místu bylo možné zvětšit výstupní matici a tím zmenšit poměr mezi vstupní (20×20×24) a výstupní submaticí (8×8×12). Jedno vlákno musí načíst 3×8 hodnot z globální paměti a krajní vlákna bloku v ose *y* a *z* musí načíst další 3x16 hodnot z globální paměti do sdílené. Byť se zdá že jedno vlákno obarvovací metody načítá více hodnot než původní varianta kernelu, a tudíž bude pomalejší, jedno vlákno vypočítá tři výstupní body (kvůli třem průchodům obarvovací metody). I této varianty je poměr mezi vstupní a výstupní submaticí příliž velký (9600:768 = 12,5)

Pro použití této metody (normální i obarvovací) je velikost sdílené paměti příliš malá a měly by smysl, kdyby velikost sdílené paměti byla řádově jednotky MB, kdyby byl poměr mezi vstupní a výstupní maticí menší než 2.

Kvůli těmto kernelům byl navrhnut alternativní zdroj tepla, který by se staral o dodávky energie mezi iteracemi v rámci kernelu. Původní datový model by do již tak pomalého kernelu přidal velké množství podmínek, což by extrémně zpomalilo kernely, ale protože jsou kernely už tak dost pomalé, zdroj tepla není v tomto kernelu implementován.

První varianta kernelu počítá s velikostí bloku 8x4x4, obarvovací varianta počítá s velikostí 4×8×8 (X×Y×Z).

5.2.11 Kernely zdroje tepla

Původní zdroj tepla je implementován pomocí krátkého kernelu. Počet vláken kernelu je stejný jako počet zahřívaných bodů. Každé vlákno se stará právě o jeden zahřívaný bod. Podle threadIdx a blockIdx si vlákno sestaví adresu do pole zahřívaných bodů. Podle této adresy a podle toho o jakou iteraci se jedná, přidá adekvátní energii z matice přidané energie do bodu v zahřívané matici. Byla implementována i verze tohoto kernelu, která umí číst a zapisovat do surface.

U alternativního zdroje tepla je počet vláken kernelu stejný jako počet bodů v zahřívané matici. V rámci vlákna se vypočítá adresa bodu ohřívané matice podle threadIdx a blockIdx. Projede se seznam všech zahřívaných subkostek a kontroluje se, zdali je bod danou subkostkou zahříván. Pokud ano, vypočítá se adresa do seznamu přidané energie a energie je přidána do bodu v zahřívané matici. Tento kernel funguje jak pro matice v globální paměti, tak pro matice v surface.

Tělo tohoto kernelu je napsáno tak, že je možné jej vložit nakonec většiny kernelů počítajících šíření tepla s velmi malými úpravami. To by mohlo urychlit celý algoritmus, protože vlákna s body které jsou zahřívány, by zredukovala zápis do cílové matice ze dvou na jeden zápis.

5.3 Implementace na více GPU

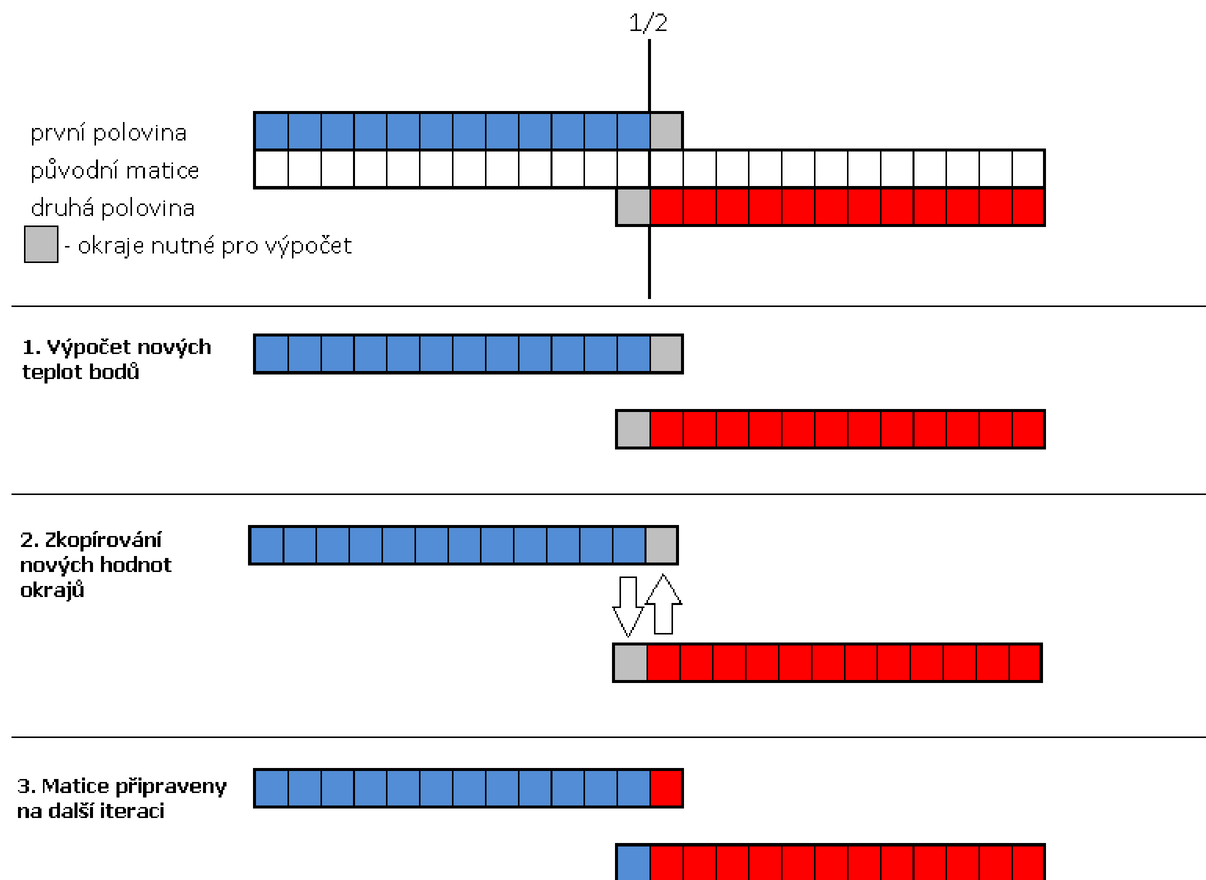
Při psaní bakalářské práce nebyl k dispozici hardware s více GPU, proto tato část práce není tak obsáhlá jako ostatní. Byla navržena a implementována verze algoritmu využívající 2 GPU, ale bohužel nebyla otestována její funkčnost.

Práce mezi dvěma GPU je rozdělena tak, že každé GPU počítá nové teploty v jedné polovině matice. Před prvním spuštěním kernelu se matice (zdrojová i cílová) rozdělí v polovině osy z na dvě matice, každé matici so přidá okolí o velikosti dvou desek v ose *z* (2 krát velikost matice o ose *x* krát velikost matice o ose *y* bodů). Každá polovina je uložena v globální paměti GPU, který s danou polovinou pracuje.

Až je paměť připravena, pustí se hlavní smyčka programu. Na začátku smyčky se spustí dva kernely, každý na jedné GPU nad svojí polovinou dat. Počká se na dokončení obou kernelů. Ještě než se vymění pointery mezi zdrojovou a cílovou maticí, je třeba zkopírovat nové hodnoty okrajů v místech, kde se matice překrývají. Krajní hodnoty matic, které se nachází v polovině původní matice, slouží každé polovině jako statické okraje, a proto je třeba přepsat je novými teplotami vypočítanými druhým GPU (Obrázek 12). Kopírování se provádí přes pomocný buffer v operační paměti hostujícího zařízení. Poté se vymění pointery mezi zdrojovou a cílovou polovinou matice v paměti obou GPU. Tato varianta nemá implementovaný zdroj tepla. Kopírování by mohlo být urychleno přímým kopírováním z paměti jedné karty do druhé.

Protože na každém GPU je spouštěn naivní kernel, velikost bloku může být libovolná, splňuje-li pravidlo, že velikost bloku je v každé ose dělitelem (půlky) matice.

Tato varianta implementace by mohla být až dvakrát rychlejší než naivní kernel, ale kopírování okrajů mezi iteracemi by ji mohlo lehce zpomalit. Varianta pracující se dvěma GPU by mohla být urychlena tím, že by se v rámci jednoho kernelu počítalo více iterací. Poměr mezi vstupní a výstupní maticí (zde se tím myslí poloviny původní matice, ne submatice ve sdílené paměti) by byl v tomto případě blízký jedné. Otázka zní, jak rychlejší by taková varianta byla, neboť když bude kernel počítat n iterací anebo se kernel počítající jednu iteraci pustí n-krát, velikost kopírované paměti mezi maticemi bude stejný.



Obrázek 12: Výměna okrajů mezi dvěma GPU

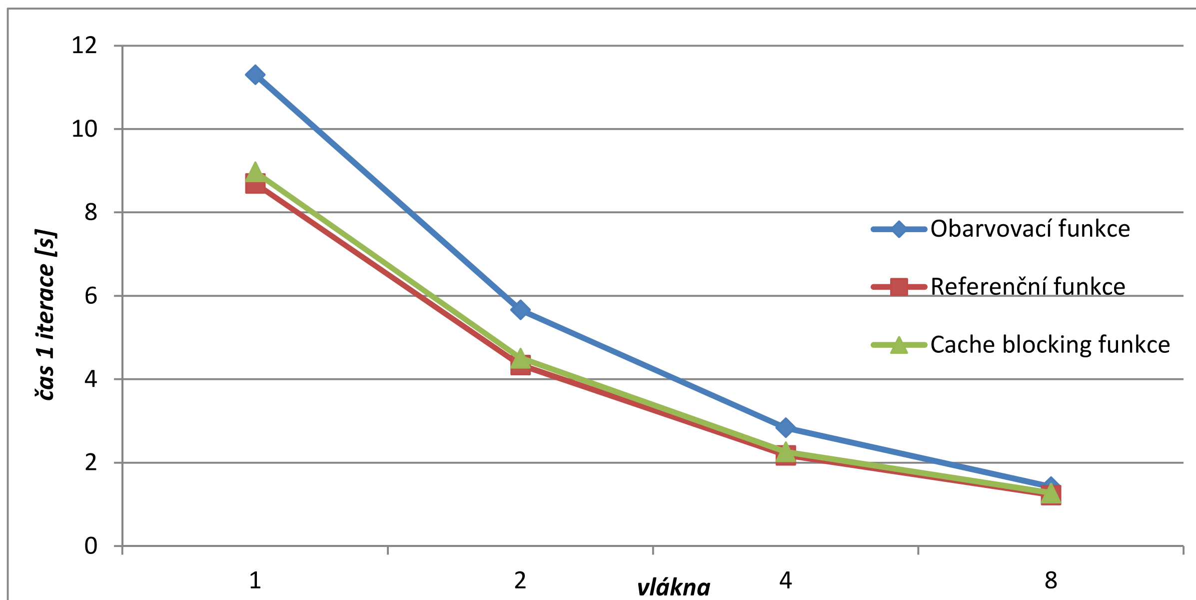
6 Výkonové porovnání variant

Čas běhu všech variant implementace byl měřen pomocí funkce `omp_get_wtime`, která je součástí knihovny OpenMP. Všechny varianty byly také testovány na těchto čtyřech velikostech matice (X×Y×Z): 260^3, 516^3, 244×260^2, 484×516^3. Kvůli metodám využívající obarvovací metodu jsou některé matice v ose x dělitelné třemi (po odečtení 4 reprezentující statické okraje), aby mohla být rychlost těchto variant správně porovnána.

6.1 CPU verze

Všechny tři funkce (referenční, obarvovací a cache blocking) byly testovány na stroji pcjaros-gpu, který obsahuje čtyřjádrový procesor Intel Core i7 920 (takt 2,66GHz; podpora hyperthreadingu; 8192KB L3 cache; 256 KB L2 cache; 2×32KB L1 cache pro instrukce a data). Na pcjaros-gpu byl program přeložen s defaultní optimalizací g++ překladače. Paralelizace byla zajištěna pomocí OpenMP direktiv. Funkce byly otestovány jako jednovláknové a s využitím 2, 4 a 8 vláken. Měřil se běh 10 iterací s vypnutým zdrojem. Zdroj byl vypnutý, protože jeho velikost (počet ohřívaných bodů) může být libovolná a tím by do měření vnášel další proměnnou.

Z časů v Graf 1 je vidět, že nejrychlejší je referenční funkce, funkce využívající cache blocking je pomalejší o 2-5%. Obarvovací metoda je pomalejší o zhruba 15%. Všechny 3 funkce téměř ideálně škálují s rostoucím počtem vláken u matice velikosti 516^3, cache blocking funkce je stejně rychlá u malé matice (260^3) při užití 4 a 8 vláken což je dané tím, že nejvyšší cyklus, který rozseká matici do L3 a který je urychlován direktivou OpenMP, neprovede tolik iterací, aby mohl být paralelizován na více jak 4 vlákna. Tuto vlastnost předvedla cache blocking funkce u všech následujících měření.

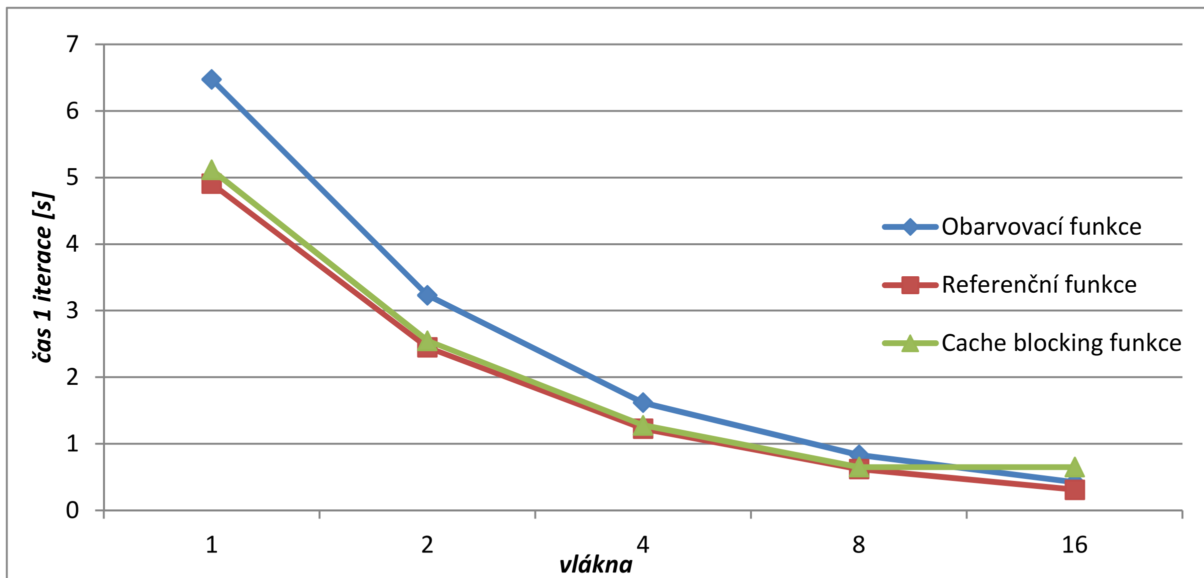


Graf 1: Čas 1 iterace na pcjaros-gpu, matice 516^3

Další měření bylo provedeno na superpočítači Anselm[6] (Graf 2). Program byl spuštěn se stejnými parametry jako na pcjaros-gpu, pouze byla upravena velikost submatic pro cache blocking funkci a program je navíc testován i jako 16 vláknový. Program běžel na dvojici CPU Intel Sandy Bridge E5-2665 (frekvence 2,4 GHz; 8 jader, 20MB L3 cache). Na Anselmu program běžel v průměru o 70-80%

[6] https://docs.it4i.cz/anselm-cluster-documentation/hardware-overview

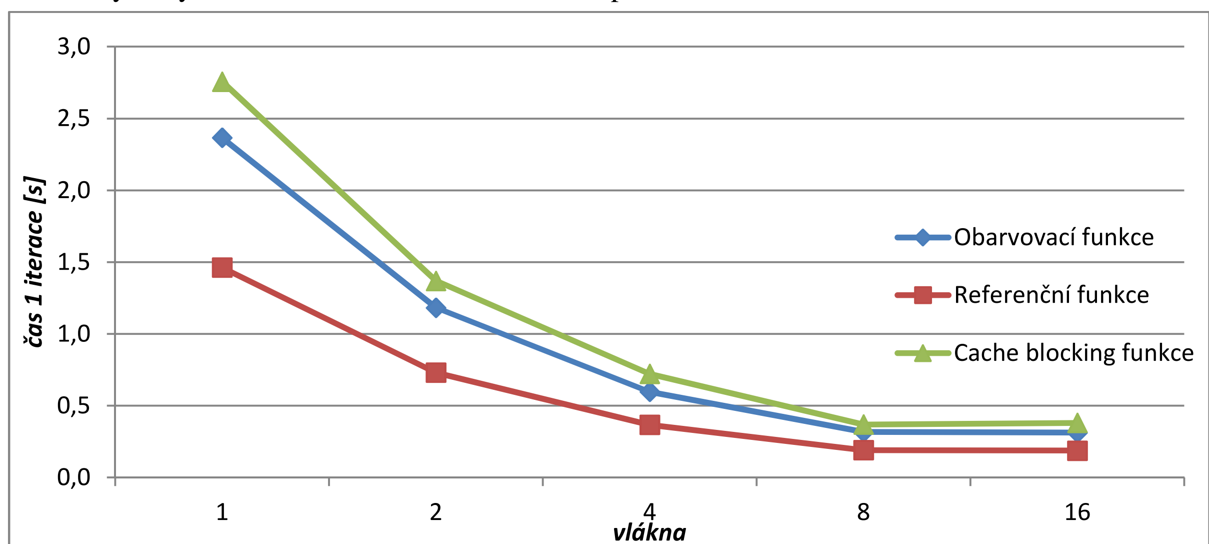
rychleji oproti pcjaros-gpu. U velké matice (516^3) byla referenční a obarvovací funkce skoro dvojnásobně rychlejší při užití 16 vláken oproti 8. Cache blocking funkce byla stejně rychlá při užití 8 a 16 vláken. Cache blocking metoda ukazuje u velké matice (516^3) při užití 16 vláken stejný problém jako u malé matice (260^3) a 8 vláken. Příčina nulového zrychlení je stejná: málo iterací cyklů pro paralelizaci pomocí OpenMP direktiv. Tato vlastnost u velké matice se projevila u všech následujících měření.



Graf 2: Čas 1 iterace na Anselmu, matice 516^3, defaultní optimalizace `g++` překladače

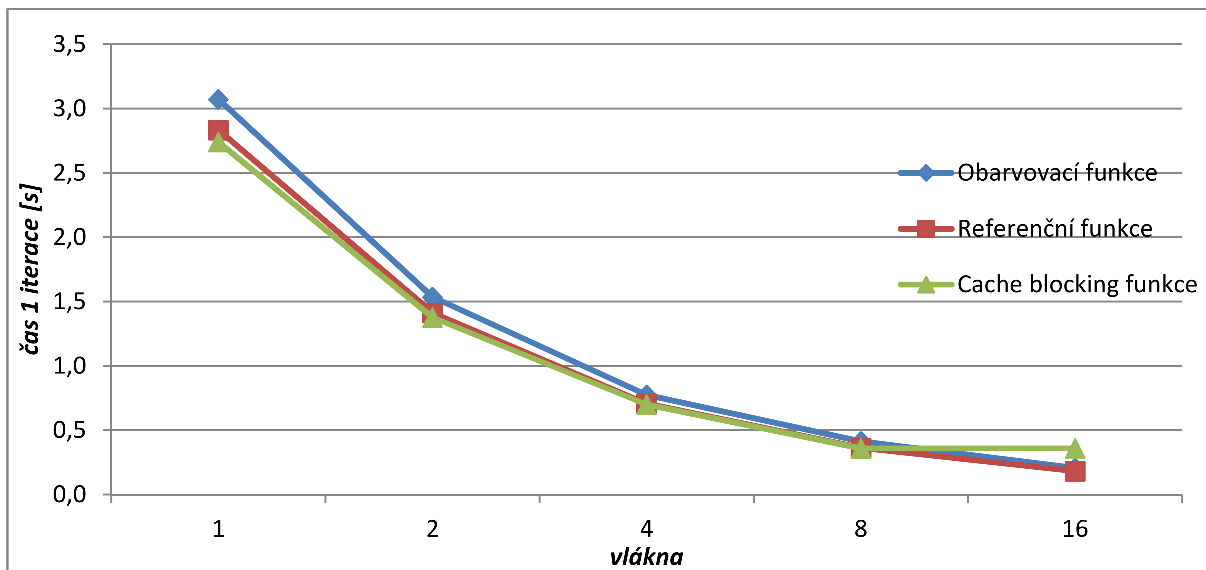
Dalšího urychlení bylo dosaženo překladem na Anselmu s maximální optimalizací: Intel překladač s parametrem `-fast`, překladač `g++` s parametrem `-O3` a nakonec `g++` s `-O3` kombinovaný s rozbalením cyklů.

Intel překladač výrazně urychlil běh programu (Graf 3) Běh 16 vláknové verze oproti 8 již naráží na limity propustnosti paměti. Obarvovací funkce byla urychlena o zhruba 170% kromě 16 vláknové verze, kde je urychlení „jen“ 35% oproti překladu s defaultní optimalizací. Referenční funkce byla urychlena zhruba o 220-240% kromě 16 vláknové, která byla urychlena o 50%. Cache blocking funkce byla zrychlena zhruba dvakrát nezávisle na počtu vláken.



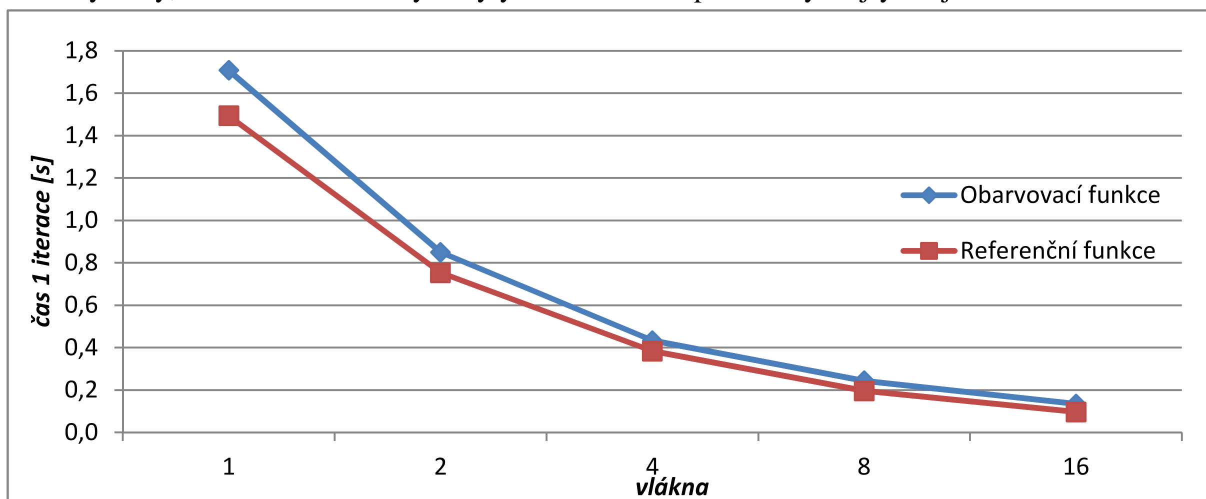
Graf 3: Čas 1 iterace na Anselmu, matice 516^3, překladač Intel (`-fast`)

Překladač `g++` byl ve většině případů pomalejší nebo stejně rychlý jako překladač od Intelu (Graf 4). Překladač `g++` ukázal lepší škálování 8 vláknové vs. 16 vláknové funkce. Obarvovací funkce byla urychlena skoro 2x oproti verzi bez optimalizace a v případě 16 vláknové verze a matice 516^3 byla dokonce rychlejší než překladač Intel. Referenční funkce ukázala zrychlení o zhruba 75%, 16 vláknová verze byla stejně rychlá jako u překladače Intel. Cache blocking funkce byla urychlena o zhruba 80%. Zajímavé je, že u tohoto překladu je rozdíl mezi referenční a obarvovací funkcí jen 13%. Velká úspora paměti na úkor takto malého zpomalení dělá z obarvovací funkce velmi praktické řešení problému.



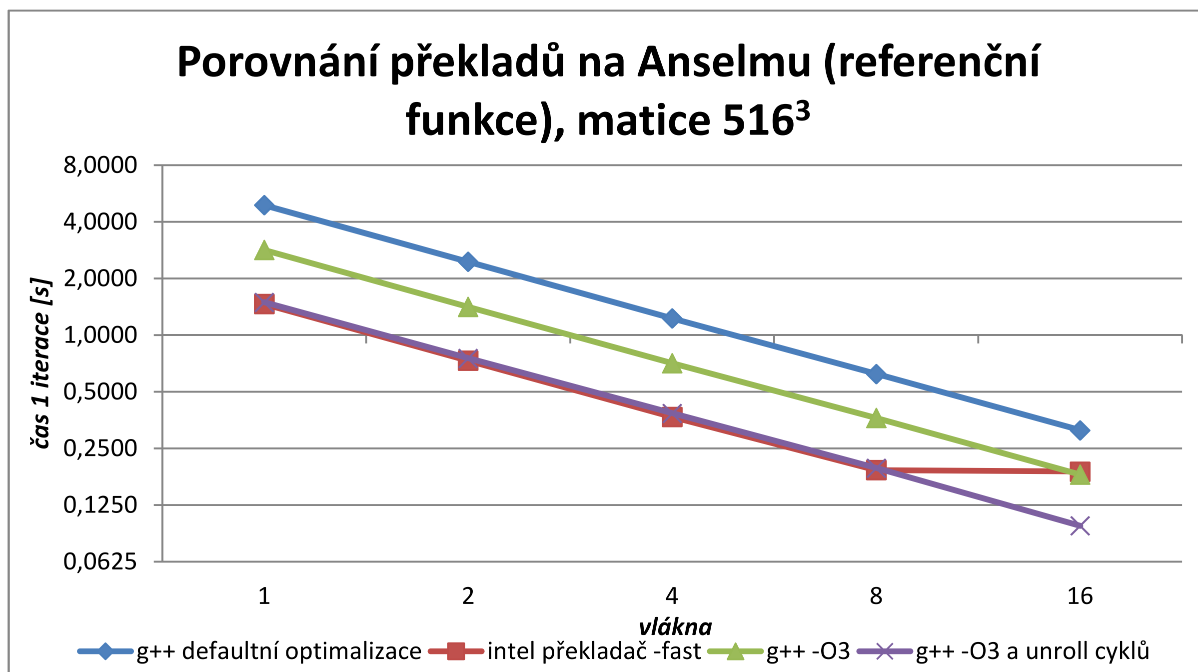
Graf 4: Čas 1 iterace na Anselmu, matice 516^3, překladač `g++ (-O3)`

Všechny funkce jsou postaveny na cyklech, tudíž vhodnou optimalizací by bylo rozbalení těchto cyklů pomocí direktivy překladače `#pragma unroll`. Verze s rozbalenými cykly (Graf 5) byla přeložena pomocí g++ překladače s parametrem `-O3`. Obarvovací funkce byla urychlena o 60-70% oproti stejnému překladu bez rozbalených cyklů a o 200-300% oproti překladu defaultní optimalizací, referenční funkce byla zrychlena o 100% oproti stejnému překladu bez rozbalení cyklů a o 200-250% oproti překladu s defaultní optimalizací. U cache blocking funkce byly bohužel špatně nastaveny velikosti submatic v jednotlivých vyrovnávacích pamětích a funkce nepočítala novou teplotu pro všechny body, takže naměřené časy nebyly validní. Tento překlad byl nejrychlejší.



Graf 5: Čas 1 iterace na Anselmu, matice 516^3, překladač `g++ (-O3)` a rozbalení cyklů

Jednovláknová referenční verze s defaultními optimalizacemi g++ překladače spuštěná na Anselmu byla 50x pomalejší než 16 vláknová verze přeložená překladačem g++ s maximální optimalizací při překladu a s rozbalenými cykly na matici o velikosti 516^3. Porovnání rychlostí všech překladů na referenční funkci můžeme vidět na Graf 6.



Graf 6: Porovnání rychlosti překladů

Protože cache blocking funkce nikdy neběžela rychleji než referenční, běh funkce byl prozkoumán pomocí rozhraní PAPI[7]. Ukázalo se, že cache blocking funkce vykoná zhruba o 30% více instrukcí oproti referenční funkci. To je dané počtem zanořených cyklů a výpočet adresy na základě všech těchto cyklů. Paradoxně při běhu cache blocking funkce dochází k řádově více výpadkům v L3 a L2 paměti (až 10x více), ale zhruba k polovičnímu počtu výpadků v L1 oproti referenční funkci. To by mohlo být způsobeno nezarovnaným přístupem do paměti kvůli statickým okrajům. Tento problém by mohl být vyřešen alokováním větší matice, posunutím platných hodnot v matici tak, aby nedocházelo k nezarovnanému čtení.

Při běhu obarvovací funkce dochází zhruba k trojnásobnému počtu výpadků v L1, L2 a L3 oproti referenční funkci, což vyplývá z toho, že matice je v rámci jedné iterace zpracovávána třemi průchody.

Nejrychlejší překlad (s rozbalenými cykly) dosahoval výkonu 32GFLOPS při běhu obarvovací funkce, 27GFLOPS při běhu referenční verze a 32GFLOPS u cache blocking funkce (i když tato funkce nepočítala novou teplotu pro všechny body v matici, pro měření hrubého výkonu to nevadí).

6.2 GPU kernely

V Tabulka 1 je shrnutí některých vlastností všech kernelů jako je počet vláken v bloku a počet bodů, pro který blok počítá novou teplotu.

Měření proběhlo na počítači pcjaros-gpu, který obsahuje kartu Nvidia GTX 580 s mikročipem architektury Fermi. Byl měřen běh 999 iterací na maticích velikosti 260^3, 516^3, 244×260^2, 484×516^3.

[7] http://icl.cs.utk.edu/papi/index.html

Kernely které mohou mít libovolnou velikost bloku byly testovány s velikostí bloku (X×Y×Z) 8×8×8, 16×8×8 a 32×8×4 vláken. Zbylé kernely byly testovány s velikostí bloku, pro kterou byly navrženy. Výkonnost kernelů v GFLOPS byla měřena pomocí konzolového nástroje nvprof.

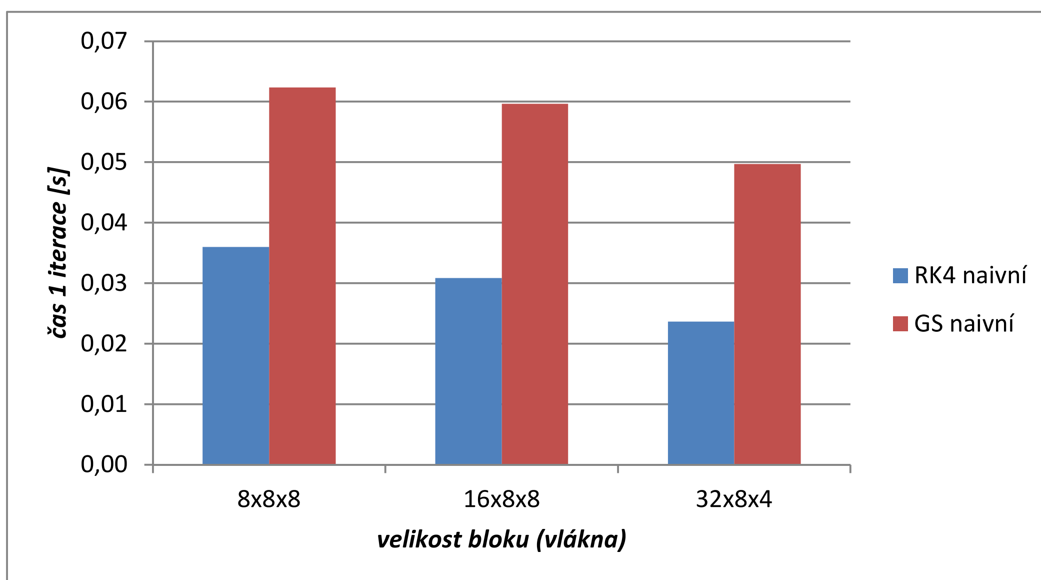
Tabulka 1: Velikost bloků jednotlivých kernelů

		blok (X×Y×Z)	
kernel	**název v grafech**	**vlákna**	**bodů**
Naivní kernel	RK4 naivní	libovolný	stejný
Naivní kernel - obarvovací metoda	GS naivní	libovolný	(X×3) ×Y×Z vláken
Kernel využívající sdílenou paměť	RK4 sdíl.p.	libovolný	Stejný
Kernel využívající sdílenou paměť s alternativním načítáním okrajů	RK4 sdíl.p.alt.	8×8×8	stejný
Kernel využívající sdílenou paměť počítající 8 bodů na vlákno	RK4 sdíl.p. 8x	8×8×8	16×16×16
Obarvovací metoda se sdílenou pamětí	GS sdíl.p.	4×8×8	12×8×8
Kernel využívající 1D texturu	RK4 1D textura	libovolný	stejný
Kernel využívající 3D texturu	RK4 3D textura	libovolný	stejný
Kernel využívající surface	RK4 surface	libovolný	stejný
Kernel počítající 3 iterace	RK4 3 iterace	8×4×4	16×12×12
Kernel počítající 3 iterace pomocí obarvovací metody	GS 3 iterace	4×8×8	20×16×16

Naivní kernel (RK4 naivní), stejně jako ostatní kernely s libovolně velkým blokem, byl zezačátku testován pouze na velikosti bloku 8x8x8. Naivní kernel vykazoval podezřele malý výkon kolem 89GFLOPS. Zvětšování bloku v ose *x* na úkor osy *y* a *z* zvýšilo výkon naivního kernelu. Tento fakt je dán tím, že když jsou vlákna bloku rozdělena po 32 do warpů a poté čtou z paměti, je vhodné, aby všechna vlákna ve warpu četla souvislý blok paměti, neboť takovýto požadavek na čtení je vyřízen rychleji, než když vlákna čtou z úplně jiných míst v paměti. Když je tedy velikost bloku v ose *x* 8 vláken, jeden warp bude číst 4 souvislé bloky paměti po 8 hodnotách. Blok velký 16 vláken v ose *x* na tom bude lépe a ideální případ nastane, když bude blok velký 32 vláken v ose *x*. Tato hypotéza se ukázala jako pravdivá a z původního výkonu 80GFPLOS (8×8×8) byl kernel urychlen na 135GFLOP (32×8×4). Bylo dosaženo 68GB/s propustnosti do globální paměti z teoretického maxima karty 193GB/s. Naivní kernel je 4× rychlejší než nejrychlejší překlad na CPU (Anselm, 16 vláken, `g++ –O3` a rozbalení cyklů) a 200× rychlejší než jednovláknová CPU verze s defaultní optimalizací při překladu.

Naivnímu kernelu využívající obarvovací metodu (GS naivní) překvapivě také pomohlo zvětšení bloku v ose *x*, ale ne tolik jako předchozímu kernelu. Při běhu s blokem 8x8x8 byl výkonný 51GFLOPS, s blokem 32×8×4 dosáhl výkonu 64GFLOPS. Tento kernel je zhruba 2× pomalejší než předchozí kernel při srovnání jejich nejrychlejších variant.

Kernel, který se snažil urychlit obarvovací metodu pomocí sdílené paměti (GS sdíl.p.), byl skoro stejně rychlý jako naivní obarvovací kernel. U některých matic byl o jednotky procent rychlejší a u některých pomalejší, kernel dosáhl výkonu 80GFLOPS. Tento kernel je tedy úplně k ničemu, neboť obarvovací metodu neurychluje a ještě k tomu potřebuje cílovou a zdrojovou matici, takže ztrácí výhodu úspory paměti.

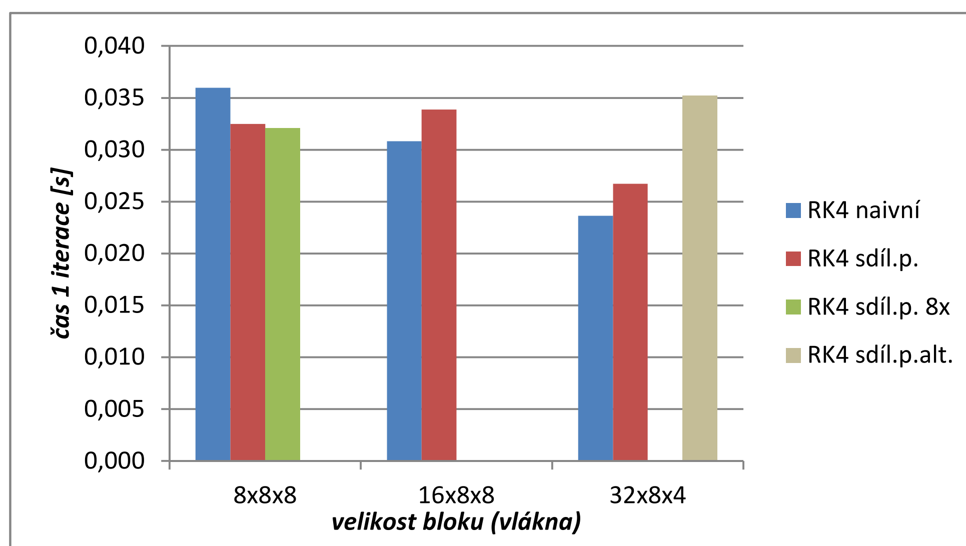


Graf 7: Srovnání naivních kernelů (matice 484×516^2)

Kernel používající sdílenou paměť (RK4 sdíl.p.) v rané fázi testování, kdy se testovalo na velikosti bloku 8x8x8, ukazoval zrychlení 20 – 30 % v závislosti na velikosti matice oproti naivnímu kernelu. Při zvětšení bloku o ose x byl tento kernel maximálně urychlen o 20% (8x8x8 vs. 32x8x4), takže byl pomalejší než nejrychlejší varianta naivního kernelu. Kernel dosáhl výkonu 109GFLOPS.

Kernel co počítá 8 bodů na vlákno s pomocí sdílené paměti (RK4 sdíl.p. 8x) byl maximálně o 1% rychlejší než předchozí kernel (RK4 sdíl.p.) s velikostí bloku 8x8x8. Výkon kernelu je 120GFLOPS.

Kernel načítající okraje do sdílené paměti podle *z* indexu (RK4 sdíl.p.alt.) byl zhruba o 12% pomalejší než první kernel se sdílenou pamětí (RK4 sdíl.p.). Toto zpomalení by mohlo být způsobené složitějším výpočtem adresy okraje podle z indexu.

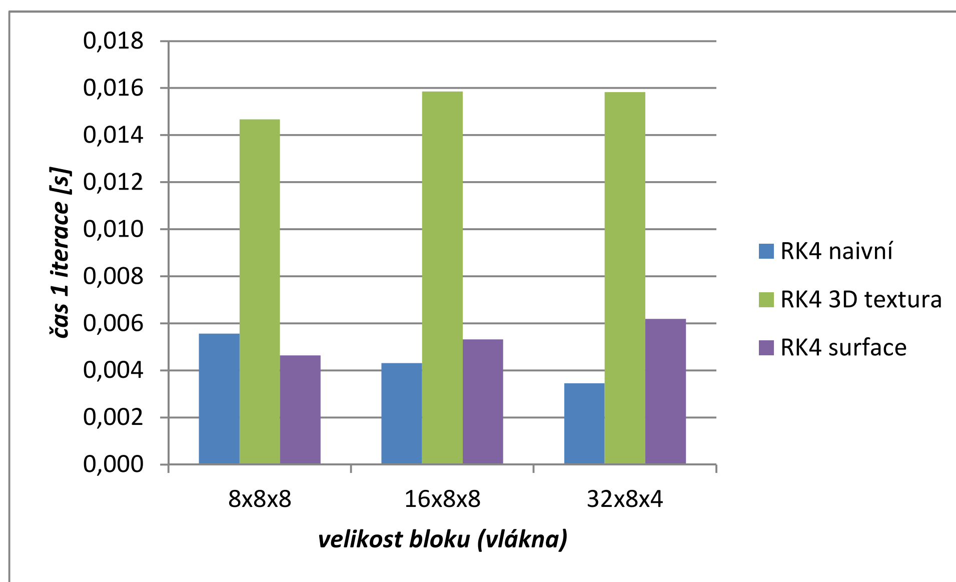


Graf 8: Srovnání kernelů využívajících sdílenou paměť (matice 484×516^2)

Všechny funkce využívající sdílenou paměť byly pomalejší než nejrychlejší varianta naivního kernelu. To může být způsobeno tím, že pro načtení okrajů submatice do sdílené paměti byly použity podmínky, které způsobují warp divergency. Kernely jsou zpomaleny tak, že následné čtení ze sdílené paměti neurychlí kernely tak, aby dohnaly naivní.

Další naměřenou skupinou kernelů jsou kernely využívající texturní paměť. První z nich využívá pouze 1D texturu (RK4 1D textura) a proto může být použit pouze na malé matice. Rychlost tohoto kernelu proto nebyla měřena, neboť počítané matice jsou pro tento kernel příliš velké. Při měření na malých maticích dosáhl výkonu 95GFLOPS.
Další kernel četl data z 3D textury (RK4 1D textura). Kernel sice správně fungoval, ale při běhu dosáhl výkonu pouze 57GFLOP. Navíc bylo nutné kopírovat mezi iteracemi hodnoty z cílové matice do zdrojového cudaArray. Tento kernel je více jak 4× pomalejší než nejrychlejší varianta naivního kernelu.

Poslední kernel využívající texturní paměť je kernel využívající surface (RK4 surface). U bloku velikosti 8×8×8 je rychlejší jak naivní a dokonce je skoro stejně rychlý jako kernel využívající sdílenou paměť (RK4 sdíl.p.). Když se ale blok v ose x zvětšil na úkor os y a z, byl tento kernel pomalejší. Nejrychlejší verze surface kernelu je o 34% pomalejší než nejrychlejší verze naivního kernelu a vykazuje výkon 90GFLOPS.



Graf 9: Porovnání kernelů využívající texturní paměť (matice 260^3)

Kernely počítající 3 iterace vykázaly pěkný výkon: normální verze (RK4 3 iterace) 120GFLOP a obarvovací (GS 3 iterace) dokonce 147GFLOP, takže dokázal lépe vytížit grafickou kartu než naivní kernel. První kernel byl více jak 20× pomalejší než naivní kernel, obarvovací byl zhruba 7× pomalejší než naivní. Režie na výpočet jednoho bodu se ukázala příliš vysoká.

Nakonec byla rychlost a výkonnost naivního kernelu porovnána se vzorovým kernelem z Nvidia knihovny příkladů. Kernel se jmenuje `FiniteDifferencesKernel` a také počítá metodu konečných diferencí. Parametry kernelu byly nastaveny tak, aby co nejvíce odpovídal naivnímu kernelu. Při nastavení stejné matice, počtu iterací, dimenzí a velikosti okolí bodu tento kernel vykonal podobné množství floating point operací (183 milionů) jako naivní (195 milionů). Kernel Nvidia dosáhl výkonu 115GFLOP (naivní 135) a byl pomalejší o zhruba 20%, i když počítal méně float operací.

7 Závěr

V rámci této bakalářské práce bylo za úkol implementovat a urychlit simulaci šíření tepla s proměnným tepelným zdrojem. Pro tento úkol byla zvolena numerická metoda konečných diferencí.

Nejdříve byla implementována jednoduchá CPU verze. Pomocí optimalizací a paralelizace byl algoritmus na CPU urychlen 50x na výkon 32 GFLOPS. To odpovídá 10% maximálního teoretického výkonu architektury.

Algoritmus byl dále implementován pomocí CUDA knihoven pro běh na GPU. Na GPU bylo implementováno celkem 11 kernelů algoritmu pro šíření tepla. Nejrychlejší z nich dosáhl výkonu 135 GFLOPS, což je 4x vyšší výkon než nejrychlejší CPU verze, která běžela na superpočítači Anselm. Tento výkon odpovídá 9% maximálního teoretického výkonu dané grafické karty. Paradoxně nejrychlejší kernel byl naivní kernel, který četl data přímo z globální paměti grafické karty. Algoritmus nebyl urychlen s použití sdílené paměti, texturní paměti nebo počítáním více iterací v rámci kernelu.

Gauss-Seidelova obarvovací metoda, která šetří pamětí, se ukázala jako dobře použitelná při implementaci na CPU. Byla jen o 13% pomalejší při úspoře až poloviny paměti. Implementace metody na GPU byla 2x pomalejší než nejrychlejší kernel, takže její přínos není tak velký.

V rámci práce byla navržena a implementována varianta pro 2 GPU, ale algoritmus nemohl otestován, neboť nebyl k dispozici hardware, na kterém by mohl být algoritmus spuštěn.

Do budoucna by mohla být vyzkoušena a odladěna implementace na dvě a více GPU. Dále by mohl být algoritmus implementován na kartě Intel Xeon Phi[8] a porovnat, jak si tato karta vede oproti procesorům stejné architektury (x86) a grafickým kartám.

[8] http://ark.intel.com/products/71992/intel-xeon-phi-coprocessor-5110p-8gb-1_053-ghz-60-core

Literatura

[1] KIRK, David a Wen-mei HWU. *Programming massively parallel processors: a hands-on approach*. Burlington: Morgan Kaufmann Publishers, 2010, xviii, 258 s. :. ISBN 978-0-12-381472-2.

[2] Nvidia Corporation: *Nvidia Next Generation CUDA Compute Architecture: Fermi.*[online]. 2009 [cit. 2014-02-18] URL<http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf>

[3] Zolfaghari, A.; Maerefat, M.: *Bioheat Transfer, Developments in Heat Transfer.* Intech, [online]. 2011 [cit. 2014-01-28], ISBN 978-953-307-569-3, str. 156. URL <http://www.intechopen.com/books/developments-in-heat-transfer/bioheat-transfer>

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

[5] Advanced Micro Devices, Inc: *AMD Opteron Series Embedded Platform.* [online]. 2012 [cit. 2014-01-27] URL <http://www.amd.com/us/products/server/processors/6000-series-platform/6200/Pages/6200-series-processors.aspx>

[6] Harris, M.: 5 Powerful New Features in CUDA 6. [online]. 2014 [cit. 2014-04-28] URL <http://devblogs.nvidia.com/parallelforall/powerful-new-features-cuda-6/>

[7] Wikipedia contributors. *X86*. Wikipedia, The Free Encyclopedia. [online] 2014 [cit. 2014-01-29] URL <https://en.wikipedia.org/wiki/X86>

[8] Wikipedia contributors. *OpenCL.* Wikipedia, The Free Encyclopedia. [online] 2014 [cit. 2014-02-05] URL <https://en.wikipedia.org/wiki/OpenCL>

[9] OpenACC: *The OpenACC Application Programming Interface.* [online]. 2011 [cit. 2014-02-15] URL <http://www.openacc.org/>

[10] OpenMP ARB: *About the OpenMP ABR and OpenMP.org.* [online] 2014 [cit. 2014-02-05] URL http://openmp.org/wp/about-openmp/

[11] Nvidia Corporation: *Nvidia's Next Generation CUDA Compute Architecture: Kepler GK110.* [online]. 2012 [cit. 2014-03-21] URL <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>

[12] Nvidia Corporation: *CUDA Toolkit Documentation*. [online]. 2011 [cit. 2014-02-09] URL <http://docs.nvidia.com/cuda/thrust/>

[13] Gupta, N.: *What is „Constant Memory" in CUDA.* [online]. 2012 [cit. 2014-04-23] URL <http://cuda-programming.blogspot.cz/2013/01/what-is-constant-memory-in-cuda.html>

[14] Gupta, N.: *Texture Memory in CUDA.* [online]. 2012 [cit. 2014-04-23] URL <http://cuda-programming.blogspot.cz/2013/02/texture-memory-in-cuda-what-is-texture.html>

[15] Wikipedia contributors. *Finite element method*. Wikipedia, The Free Encyclopedia. [online] 2014 [cit. 2014-01-29] URL <https://en.wikipedia.org/wiki/Finite_element_method>

[16] Wikipedia contributors. *Heat transfer*. Wikipedia, The Free Encyclopedia. [online] 2014 [cit. 2014-01-26]

URL <https://en.wikipedia.org/wiki/Heat_transfer>

[17] Hung, Y.: CUDA Advanced Memory Usage and Optimization. [online]. 2010 [cit. 2014-04-14] URL <http://www.math.ntu.edu.tw/~wwang/mtxcomp2010/download/cuda_04_ykhung.pdf>

[18] Gupta, N.: *CudaArray in CUDA.* [online]. 2012 [cit. 2014-04-01] URL <http://cuda-programming.blogspot.cz/2013/02/cuda-array-in-cuda-how-to-use-cuda.html>

[19] Intel Corporation: *Cache blocking Techniques.* [online]. 2013 [cit. 2014-04-20] URL <http://software.intel.com/en-us/articles/cache-blocking-techniques>

[20] Fučík, O.: *Návrh číslicových systémů (INC): Technologie.* [online]. 2013 [cit. 2014-01-25] URL <http://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/INC-IT/lectures/technologie.pdf>

[21] BARTUŠKA, Karel a Emanuel SVOBODA. *Fyzika pro gymnázia: molekulová fyzika a termika*. 4. přeprac. vyd. Praha: Prometheus, 2000, 244 s. ISBN 80-719-6200-7.

[22] Hliněná, D.: Příklady řešené na přednáškách předmětu INM: Část I.: Řešení soustavy rovnic. [online]. 2007 [cit. 2014-02-28] URL < http://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/INM-IT/lectures/prednaska1a2.pdf>