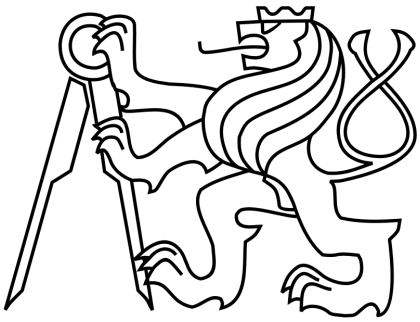
České vysoké učení technické v Praze
Fakulta stavební
Katedra mechaniky



BAKALÁŘSKÁ PRÁCE

Efektivní algoritmy pro vyhodnocení statistických deskriptorů

Vedoucí práce: Ing. Jan Sýkora, Ph.D.

Praha 2012

Jan Havelka

Název práce:	Efektivní algoritmy pro vyhodnocení statistických deskriptorů
Autor:	Jan Havelka
Katedra:	Katedra mechaniky
Vedoucí práce:	Ing. Jan Sýkora, Ph.D.

Abstrakt: Homogenizační metody jsou stále více využívanější přístupy pro modelování chování heterogenních materiálů. Základním principem homogenizace je nahrazení heterogenní mikro-/mezostruktury ekvivalentním homogenním materiálem. Jestliže se navíc zabýváme náhodnou strukturou je vhodné jednotkovou buňku (PUC) představující periodickou část mikro-/mezostrukturu nahradit statisticky ekvivalentní periodickou jednotkovou buňkou (SEPUC), která uchovává základní materiálové vlastnosti ve statistickém smyslu. Jedním z vhodných statistických deskriptorů pro definování SEPUC je funkce lineal path. Jedná se o deskriptor nižšího řádu, jehož hlavní výhodou je schopnost popsat určité informace o fázové spojitosti zkoumaného média. Nevýhodou je naopak jeho vysoká výpočetní náročnost. V práci je ukázáno přeformulování sekvenčního programu, který analyzuje digitální obraz pomocí funkce lineal path, napsaném v jazyku C do paralelního schématu napsaném v jazyku CUDA C - využívající výpočetní kapacitu NVIDIA grafických procesorů.

Klíčová slova: Analýza digitálního obrazu, lineal path function, homogenizace, statisticky ekvivalentní periodická jednotková buňka, grafický procesor, CUDA

Title:	Efficient algorithms for evaluation of statistical descriptors
Author:	Jan Havelka
Department:	Department of Mechanics
Supervisor:	Ing. Jan Sýkora, Ph.D.

Abstract: Homogenization methods are still the most approach to modeling of heterogeneous materials. The main principle is to represent the heterogeneous microstructure with an equivalent homogeneous material. When dealing complex random microstructures, the unit cell representing exactly periodic morphology needs to be replaced by a statistically equivalent periodic unit cell (SEPUC) preserving the important material properties in the statistical manner. One of the statistical descriptors suitable for SEPUC definition is the lineal path function. It is a low-order descriptor based on a more complex fundamental function able to capture certain information about the phase connectedness. Its main disadvantage is the computational cost. In this work is presented the reformulation of the sequential C code for evaluation of the lineal path function into the parallel C code with CUDA extensions enabling the usage of computational potential of the NVIDIA graphics processing unit.

Keywords: Lineal path function, homogenization, statistically equivalent periodic unit cell, graphics processing unit, CUDA

Prohlašuji, že jsem tuto bakalářskou práci napsal samostatně, pouze za odborného vedení vedoucího práce Ing. Jana Sýkory, Ph.D. Dále prohlašuji, že veškeré podklady, ze kterých jsem čerpal, jsou uvedeny v seznamu literatury. Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 9. 5. 2012 Jan Havelka

Rád bych poděkoval vedoucímu bakalářské práce Ing. Janu Sýkorovi, Ph.D. a Ing. Anně Kučerové, Ph.D., za rady a čas, který mi věnoval při přípravě nejen této bakalářské práce. Dále bych chtěl poděkovat své rodině za podporu.

Tato práce byla podpořena projektem číslo P105/12/1146, uděleným Grantovou agenturou České republiky.

Obsah

Slovník pojmů

API *"Application Programming Interface"* označuje v informatice rozhraní pro programování aplikací. Tento termín používá softwarové inženýrství. Jde o sbírku procedur, funkcí či tříd nějaké knihovny (ale třeba i jiného programu nebo jádra operačního systému), které může programátor využívat. API určuje, jakým způsobem jsou funkce knihovny volány ze zdrojového kódu programu. [1]

Broadcast je v informatice zpráva, kterou v počítačové síti přijmou všechna připojená síťová rozhraní. Pomocí broadcastů se v současných sítích šíří například informace o sdílení v sítích Microsoft (pomocí protokolu SMB) nebo identifikace zařízení (CDP protokol). [2]

CPU *"Central Processing Unit"* je základní výpočetní jednotka každého počítače, známá spíše jako "procesor".

CUDA *"Compute Unified Device Architecture"* je hardwarová a softwarová architektura, která umožňuje na GPU spouštět programy napsané v jazycích C/C++, FORTRAN nebo programy postavené na technologiích OpenCL, DirectCompute a jiných.[3]

FLOPS *"FLoating-point OPerations per Second"* je zkratka pro počet operací v plovoucí řádové čárce za sekundu, což je obvyklé měřítko výkonnosti počítačů. Výkon dnešních špičkových superpočítačů se pohybuje v řádu stovek miliard FLOPS. [4]

GPU "Graphics Processing Unit"je grafická výpočetní jednotka. Jedná se o druh počítačového čipu umístěného na grafické kartě.

Latence V počítačové technice je latence zpoždění mezi vydáním povelu paměťovému zařízení a výstupem prvních dat. U grafických karet se využívá tohoto termínu obdobně, s tím rozdílem, že se se latence neměří přímo v čase, ale v počtu cyklů, což je přímo závislé na použitém hardwaru.[5]

Počítačový klastr je seskupení volně vázaných počítačů, které spolu úzce spolupracují, takže navenek mohou pracovat jako jeden počítač. Obvykle jsou propojeny počítačovou sítí. Klastry jsou obvykle nasazovány pro zvýšení výpočetní rychlosti nebo spolehlivosti s větší efektivitou než by mohl poskytnout jediný počítač, přičemž jsou levnější než jediný počítač o srovnatelné rychlosti nebo spolehlivosti.[6]

RAM *"Random-Access Memory"* je v informatice typ elektronické paměti, která umožňuje přístup k libovolné části v konstantním čase bez ohledu na její fyzické umístění. [7]

SIMD *"Single Instruction, Multiple Data"* což v překladu znamená - s jedním tokem instrukcí, s vícenásobným tokem dat. Počítač používající více stejných procesorů, které jsou řízeny společným programem. Zpracovávaná data jsou různá, takže každý procesor pracuje s jinou hodnotu, ale všechny procesory současně provádějí stejnou instrukci. [8]

SPMD *"Single Program, Multiple Data"* což v překladu znamená - stejný program, s vícenásobným tokem dat. Všechny procesory vykonávají stejný program, ale jsou na sobě nezávislé (nejsou synchronizovány). Jednotlivé procesory musí mít svůj řadič, který řídí rychlost přísunu instrukcí a jejich provádění. [8]

Vlákno/Thread označuje v informatice odlehčený proces, pomocí něhož se snižuje režie operačního systému při změně kontextu, které je nutné pro zajištění multitaskingu (zdánlivého běhu více úloh zároveň, který je zajištěn jejich rychlým střídáním na procesoru) nebo při masivních paralelních výpočtech. Zatímco běžné procesy jsou navzájem striktně odděleny, sdílí vlákna nejen společný paměťový prostor, ale i další struktury. [9]

Warp tento termín pochází původně ze slova "weaving", první paralelní technologie. Jedná se bez ohledu na výpočetní možnosti zařízení o skupinu 32 vláken, které se spouští zároveň.

Úvod

Tato práce je zaměřena na modelování náhodných dvoufázových heterogenních materiálů, což je více oborový vědní problém s celou řadou relevantních inženýrských aplikací. Heterogenní materiál se v nejobecnějším smyslu skládá z oblastí různých materiálů (fází) nebo stejného materiálu v odlišných stavech. Příkladem takového média jsou například kompozitní materiály, suspenze, porézní nebo biologická média.

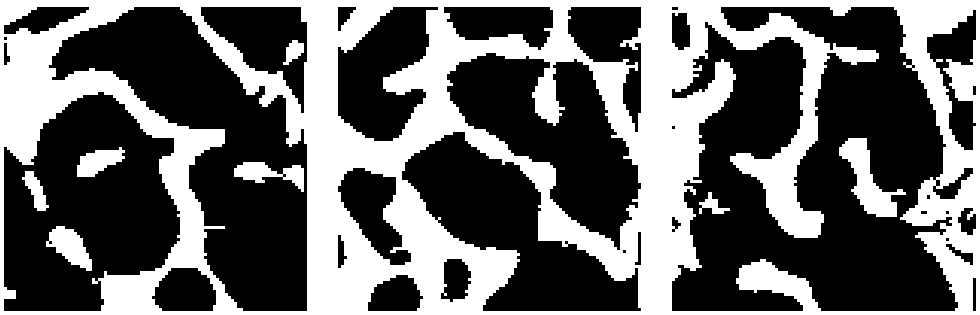
Modelování náhodných heterogenních materiálů se používá k predikci makroskopických nebo efektivních fyzikálních vlastností. Jedním z nejvíce používaných principů je homogenizační metoda, která je založena na předpokladu nahrazení heterogenní mikro-/mezostruktury ekvivalentním homogenním materiálem [10]. V současné době jsou používány nejvíce tyto dva hlavní homogenizační přístupy:

I Výpočetní homogenizace, viz [11, 12]

II Teorie efektivního média, viz [13]

Zatímco první soubor homogenizačních technik studuje rozložení lokálních polí v typickém heterogenním vzorku materiálu za pomocí numerických metod, druhá skupina odhaduje odezvu vzorku analyticky na základě geometrických informací studovaného média, jako je například křivka zrnitosti atd.

V bakalářské práci se zaměříme na první skupinu homogenizačních technik, založených na konceptu statisticky ekvivalentní periodické jednotkové buňky, dále jen SEPUC. Pro vygenerování SEPUC je nejdříve nutné referenční médium určitým způsobem popsat. K tomuto účelu se využívá statistických deskriptorů.



Obrázek 1: Mikrostruktury spongiózní kosti získané počítačovou tomografií [14]

Každý z těchto deskriptorů má různé vlastnosti, použití a výpočetní náročnost. Cílem této práce bylo vytvořit statisticky ekvivalentní buňku na základě zadané mikrostruktury, kterou v našem případě představovala spongiózní kost (obr. 1). Jednou z významných vlastností této mikrostruktury je spojitost jednotlivých fází.

Pro popis spongiózní kosti je nejvíce vhodný deskriptor lineal path, právě kvůli schopnosti zachovat určité informace o fázové spojitosti. Z hlediska výpočetní náročnosti, zejména při optimalizačním procesu SEPUC, je její použití časově velice náročné a značná část této práce je zároveň věnována technologii výpočtu na grafických procesorech, kterou byla funkce lineal path optimalizována.

1 Lineal path funkce

Funkce lineal path, viz [15], patří do skupiny funkcí nižšího řádu zkoumající mikro-/mezostrukturu materiálu. Do této skupiny rovněž patří např. fázové objemové zastoupení, specifický povrch, distribuce velikosti pórů, dvoubodová pravděpodobnost a další, které jsou speciálními případy obecné n-bodové distribuční funkce. Tyto funkce vykazují podstatný nedostatek, kterým je nepřesná interpretace velkých fázových shluků v popisu struktury. Tato vlastnost může hrubě ovlivnit počítané makroskopické vlastnosti zkoumaného média. Lineal path oproti ostatním funkcím zachovává více detailních informací o fázové spojitosti, její matematický popis vychází z definice elementární funkce λ_r [15], pro kterou platí následující vztah

$$\lambda_r(x_1, x_2, \alpha) = \begin{cases} 1, & \text{pro } x_1 x_2 \subset D_r(\alpha) \\ 0, & \text{pro ostatní případy} \end{cases}, \tag{1}$$

Tato funkce nabývá hodnoty 1 pro úsek $\mathbf{x_1 x_2}$ který je celý obsažen ve fázi r vzorku α a hodnoty nula v ostatních případech. Lineal path funkce potom představuje pravděpodobnost, že úsek $\mathbf{x_1 x_2}$ leží ve fázi $\mathbf{r}$. Funkce je definována následujícím vztahem

$$L_r(x_1, x_2) = \overline{\lambda_r(x_1, x_2, \alpha)}, \tag{2}$$

kde práva strana představuje střední hodnotu elementární funkce. Podmínkou statistické homogenity a isotropie se předpis funkce zjednoduší na následující tvar

$$L_r(x_1, x_2) = L_r(x_1 - x_2) = L_r(\|x_1 - x_2\|). \tag{3}$$

Jestliže body $\mathbf{x_1}$ a $\mathbf{x_2}$ mají stejné prostorové souřadnice, vypočtená hodnota lineal path funkce se rovná objemovému zastoupení fáze $\mathbf{r}$. Na druhou stranu, pro body $\mathbf{x_1}$ a $\mathbf{x_2}$ dostatečně vzdálené od sebe je vypočtená hodnota lineal path funkce rovna nule.

2 CUDA technologie

2.1 Historie

Počátkem osmdesátých a devadesátých let minulého století enormně vzrostl zájem o paralelní výpočty, přesněji řečeno datově paralelní výpočty, kdy jsou stejné operace vyhodnocovány paralelně pro různé datové elementy. V tomto období bylo prozkoumáváno a postaveno mnoho druhů architektur a technologií, včetně Cray 1, Cray 2, XMP, YMP, CM1, CM2 a CM5 atd.

Výsledkem tohoto úsilí byly superpočítače v pravém slova smyslu. Byly výkonné, exotické a velmi drahé. Mělo k nim přístup jen velmi omezené množství privilegovaných lidí většinou z národních laboratoří nebo předních univerzit. Navzdory limitované dostupnosti těchto zařízení byl stále velký zájem o paralelní výpočty a v té době vzniklo velké množství článků a prací na toto téma. Programovací jazyky, modely a datově paralelní algoritmy byly vyvíjeny a řešila se celá řada problémů, včetně designu architektury nových počítačů.

Veškerý tento vývoj byl v určitém smyslu limitován. Počet prodaných zařízení byl velmi malý, stejně jako počet lidí, kteří s nimi uměli pracovat. Důsledkem bylo nahrazení těchto velkých a drahých zařízení skupinami menších počítačů, jako byl například Beowulf nebo Legion klastr. Přešlo se tedy z používání počítačů s masivně paralelními procesory k velkým skupinám počítačů s mikroprocesory, kde jsou data rozdělena na velké části, které jsou převzaty jednotlivými počítači ve skupině. Této technologii se říká SPMD a nahradila SIMD technologii masivně paralelních procesorů.

Počátkem roku 2000 snaha o zrychlování mikroprocesorů narazila na technologické možnosti a tento progres se dramaticky snížil. Důsledkem bylo, že postavení výkonnějšího klastru si žádalo stavění fyzicky větších zařízení, což negativně ovlivnilo celkovou rozšiřitelnost, prostorovou a energetickou náročnost a grafické karty se začaly jevit jako vhodná alternativa.

GPU jsou počítačové čipy umístěné na grafických kartách a v herních konzolách. Jsou to masivně paralelní zařízení, navržené operovat s bilióny pixelů

a milióny polygonů za sekundu, a to vše díky vysoké úrovni paralelizace. GPU jsou mnohajádrové počítačové čipy, které obsahují stovky skalárních procesorů, na rozdíl od standardních CPU, které mají v dnešní době 2, 4, 6 nebo 8 jader. Jsou schopné vyhodnotit desítky tisíc vláken současně a mají výkon až 1 TFLOPS. Svojí stavbou jsou podobné dřívějším superpočítačům a v mnoha případech je překonávají.

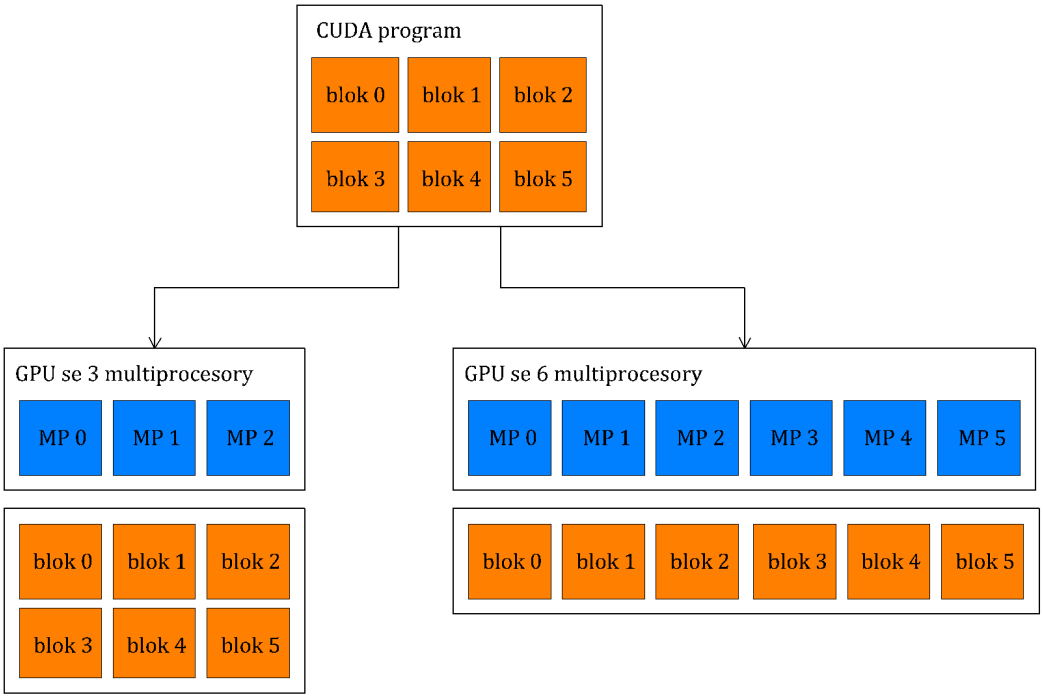
Grafické karty tedy mají velmi silný výpočetní potenciál. Pokud je ale chceme využít pro negrafické účely, musíme nejdříve vytvořit program přes grafické API, což je velmi obtížné a ne vždy proveditelné. Určitou změnu v tomto procesu přináší technologie CUDA. Je to paralelní výpočetní model a softwarové prostředí, které se skládá z malých rozšíření standardního C++ kódu.

CUDA je heterogenní výpočetní model, využívající CPU a GPU v částech programu, kde jsou nejsilnější. Sériové části programu jsou vyhodnoceny na CPU a paralelní části jsou pak převzaty GPU, což umožňuje postupnou změnu kódu z CPU k využití GPU.

Závěrem se tedy dá říci, že grafické karty jsou svým výkonem nástupcem nynějších superpočítačů a CUDA je jakýmsi nástrojem, který dává uživateli možnost využít tento výkon mimo grafické prostředí pro obecné účely. Současně jsou cenově velmi dostupné a najdeme je ve většině dnešních počítačů. Výpočty na masivně paralelních zařízeních se tedy přenesly z hrstky privilegovaných pracovišť a pracovníků do domácích počítačů a programování na nich se stává tak samozřejmým jako na standardních CPU.

2.2 Programovací model

Příchod vícejádrových CPU a mnohajádrových GPU znamená, že současné procesorové čipy jsou nyní paralelní systémy, jejichž paralelismus pokračuje v měřítku Mooreova zákona, který říká, že počet tranzistorů, které mohou být umístěny na integrovaný okruh se při zachování stejné ceny zhruba každých 18 měsíců zdvojnásobí [16]. Určitou výzvou je tedy vývoj softwaru, který bude schopný využít vzrůstající počet výpočetních jader, stejně jako grafické



Obrázek 2: Rozdělení bloků mezi multiprocesory

aplikace. Programovací model CUDA je navržen pro překonání tohoto problému tím, že automaticky rozděluje bloky vláken, viz kapitoly 2.2.2, 2.2.3 a (obr. 2), rovnoměrně mezi všechny multiprocesory. Na programátorovi tedy zůstává volba velikosti a počtu bloků, což přímo ovlivňuje rozšiřitelnost na různá zařízení a využití výpočetního výkonu.

Ve své podstatě se programovací model skládá ze tří klíčových abstrakcí - kernelů, hierarchie vláken a paměťového modelu. Tyto abstrakce navádějí programátora rozdělit problém na hrubé části, které se dají vyřešit zvlášť v paralelních blocích vláken. Každá z těchto částí může být řešena paralelně ve spolupráci se všemi vlákny uvnitř bloku.

Tento rozklad zachovává expresivitu jazyku umožněním vzájemné spolupráce jednotlivých vláken v bloku při řešení každé části problému a zároveň umožňuje automatickou škálovatelnost. Bloky vláken musí být mezi sebou datově nezávislé a každý může být naplánován na jakýkoliv dostupný multiprocesor, v libovolném pořadí, současně nebo sekvenčně. To znamená, že každý CUDA program lze spustit na libovolném počtu multiprocesorů viz (obr. 2). Programátor je současně zbaven povinnosti vytvářet aplikaci pro

určitý počet multiprocesorů a nemusí ani znát jejich počet.

Jak již bylo zmíněno v úvodu, programování na grafických kartách s sebou nese určité změny a rozšíření oproti standardnímu C++ kódu. V následujících kapitolách si osvětlíme základní ovládací prvky a pojmy, s jakými se programátor setkává.

2.2.1 Kernel

CUDA umožňuje programátorům definovat funkce, nazývané *kernely*, které jsou obdobou standardních C++ funkcí. Zavoláním této funkce se spustí daný kód $N-$krát paralelně, N odlišnými *CUDA vlákny*. Oproti CUDA funkci, je obyčejná C++ funkce spuštěná na procesoru vyhodnocena v čase jen jednou na jednom datovém elementu.

Kernel je definován **__global__** deklarací a počet CUDA vláken, kterými je spuštěn, se definuje ve volání funkce použitím nové <<<...>>> syntaxe. Každé vlákno, které spouští kernel, má vlastní index přístupný z kernelu vestavěnou proměnnou **threadIdx**.

Pro názornou ukázku použití předchozích výrazů slouží následující příklad sčítání dvou vektorů A a B velikosti N a do vektoru C.

Algoritmus 1: Příklad paralelního součtu dvou vektorů

```
1  // Definice kernelu
2  __global__ void Soucet(int* A, int* B, int* C) {
3      int idx = threadIdx.x;
4      C[idx] = A[idx] + B[idx];
5  }
6  int main() {
7      // Alokace a kopírování paměti na grafickou kartu
8      // Volání kernelu s N vlákny
9      Soucet<<<1, N>>>(A, B, C);
10 }
```

Každé z N vláken tedy sčítá jediný prvek z vektorů A a B a výsledek ukládá na stejnou pozici vektoru C.

2.2.2 Hierarchie vláken a bloků

Každé vlákno může představovat maximálně tří-složkový vektor, což znamená, že vlákna jsou identifikována jedním, dvěma, nebo třemi indexy, které jsou přístupné z kernelu vestavěnou proměnnou **threadIdx**. Toto uspořádání poskytuje přirozený způsob, jak přistupovat k prvkům v doméně jako jsou vektor, matice nebo objem.

Index vlákna v bloku a adresa, na kterou vlákno přistupuje, jsou spolu úzce svázány a vypočítají se podle následujících vztahů. Pro jednorozměrný blok vláken je index vlákna totožný s adresou

$$idx = x \tag{4}$$

Pro dvourozměrný blok vláken velikosti (D_x, D_y) je adresa vlákna s indexem (x, y) rovna

$$idx = x + yD_x \tag{5}$$

Pro třírozměrný blok vláken velikosti (D_x, D_y, D_z) je adresa vlákna s indexem (x, y, z) rovna

$$idx = x + yD_x + zD_xD_y \tag{6}$$

Bloky vláken jsou abstrakcí vytvořenou pro programátora. Na grafické kartě se ve skutečnosti fyzicky nikde nevyskytují, jejich význam je v rovnoměrném rozdělení skupin vláken mezi jednotlivé multiprocesory pro jakékoliv zařízení viz (obr. 2).

Bloky vláken tvoří opět jedno-, dvou- nebo tří-dimenzionální mřížku (obr. 3). Počet bloků vláken je obvykle určen objemem zpracovávaných dat, počtem multiprocesorů na grafické kartě a využitím sdílené paměti a registrů jednotlivými bloky.

Každý blok v mřížce může představovat maximálně tří-složkový vektor a může být identifikován jedním, dvěma nebo třemi indexy, které jsou přístupné z kernelu vestavěnou proměnnou **blockIdx**. Dimenze bloku vláken je

přístupná proměnnou **blockDim**. Všechny proměnné jsou názorně ukázány na (obr. 3).

Počet vláken v bloku a počet bloků v mřížce je specifikován při spouštění kernelu a může být typu **int** nebo **dim3**.

2.2.3 Spouštění vláken

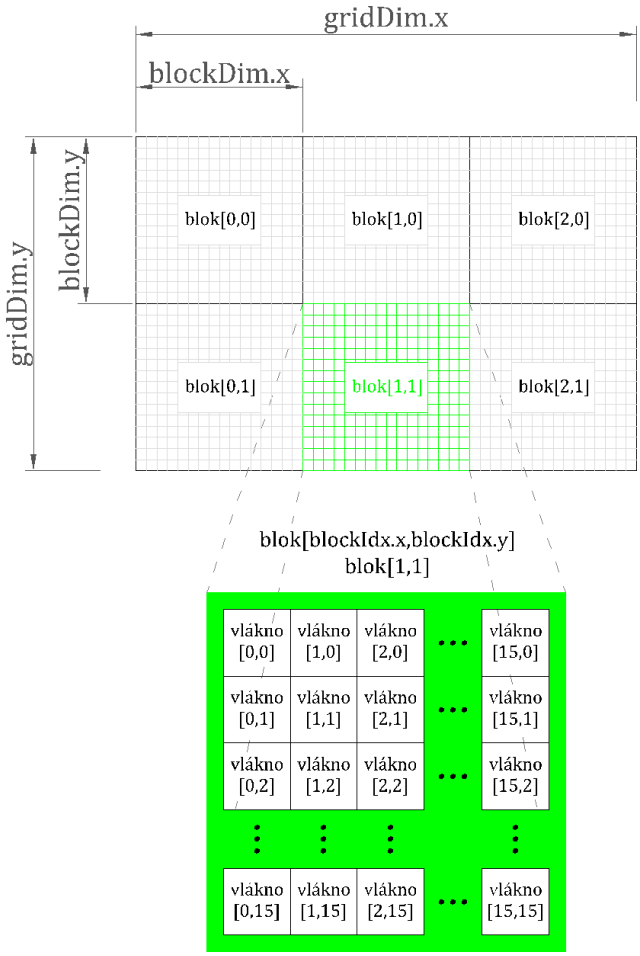
Nejdříve se bloky vláken při spuštění kernelu rovnoměrně rozdělí mezi jednotlivé multiprocesory. Každý multiprocesor pak vytváří, řídí, plánuje a spouští zároveň 32 paralelních vláken, tzv. *warpů*.

Warp vykonává jednu společnou instrukci v čase a je tedy důležité, aby všechna vlákna ve warpu procházela stejným algoritmem. Pokud se vlákna ve warpu rozcházejí, instrukce se vyhodnocují sekvenčně, resp. jsou vynechána odkloněná vlákna, která jsou vyhodnocena až po ukončení činnosti všech ostatních vláken. Každý warp se vyhodnocuje samostatně a tak tento jev může nastat pouze v podmínkách, kde jsou řešeny celky menší, než je velikost warpu.

Po rozdělení bloků vláken mezi multiprocesory sdílí všechny bloky jednu paměť daného multiprocesoru, která je velmi omezená a počet vláken na blok je tím limitován. Na současných grafických kartách může blok vláken obsahovat maximálně 1024 vláken.

Jak již bylo naznačeno v předchozích odstavcích, kernel může být spuštěn mnoha stejnými bloky a celkový počet vláken, kterými je spuštěn, je roven počtu vláken na blok vynásobený počtem bloků. Na následujícím jednoduchém příkladu je ukázána popsaná hierarchie vláken.

Mějme například 2D obrázek o rozměrech 48x32 pixelů, pro který vytvoříme 2D mřížku, kde každé vlákno bude operovat s jedním pixelem v obrázku. Mřížka má tedy celkem 48x32 vláken, která jsou uspořádána v blocích. Volba velikosti bloku je závislá pouze na hardwarových limitech, ale s ohledem na rychlost výpočtu a velikost obrázku vytvoříme čtvercové bloky s počtem vláken v násobcích velikosti warpu (32), tedy například 16x16. Výsledný počet bloků, který budeme potřebovat, je 3 a 2 bloky (obr. 3). Pokud ovšem naše



Obrázek 3: Rozdělení domény na bloky vláken

data nedovolují takovéto mapování, je vždy možné použít 1D mřížku. Volání kernelu pro dvourozměrnou mřížku pak bude vypadat následovně

Algoritmus 2: Příklad alokace dvourozměrné mřížky

```
1 int main() {
2     // Alokace a kopírování paměti na grafickou kartu
3     // Vytvoření bloku o 16x16 vláknech
4     dim3 dimBlock( 16,16 );
5     // Vytvoření mřížky o 3x2 blocích
6     dim3 dimGrid( 3,2 );
7     // Volání kernelu
8     kernel<<< dimGrid, dimBlock >>>( image );
9 }
```

Aby bylo možné rozdělit jednotlivé bloky na multiprocesory pro jakýkoliv hardware, mezi jednotlivými bloky nesmí být žádná datová závislost a musí být umožněno jejich paralelní nebo sériové spouštění.

2.3 Paměťový model

Programovací model CUDA zároveň předpokládá, že hostitelské zařízení a grafická karta mají své vlastní, fyzicky oddělené, paměťové prostory v DRAM. Jednotlivá vlákna mohou přistupovat k datům celkem ze šesti různých míst. Každé vlákno má svojí vlastní lokální paměť a zároveň má přístup k registrům. Každý blok vláken má přístup do sdílené paměti viditelné jen pro vlákna z daného bloku. Všechna vlákna, nehledě na rozdělení na bloky, mají přístup do globální paměti. Je zde navíc paměť pro textury a konstanty, do které mají přístup všechna vlákna. Aby byla umožněna vzájemná kooperace hostitelského zařízení a grafické karty jsou globální paměť, paměť pro textury a konstanty přístupné i z hostitelského zařízení voláním speciálních funkcí umožňující alokaci, mazání a přesun z jedné paměti na druhou a obráceně. Všechna tato paměťová místa mají rozdílné využití, velikost, latenci a umístění na grafické kartě.

2.3.1 Globální paměť (global memory)

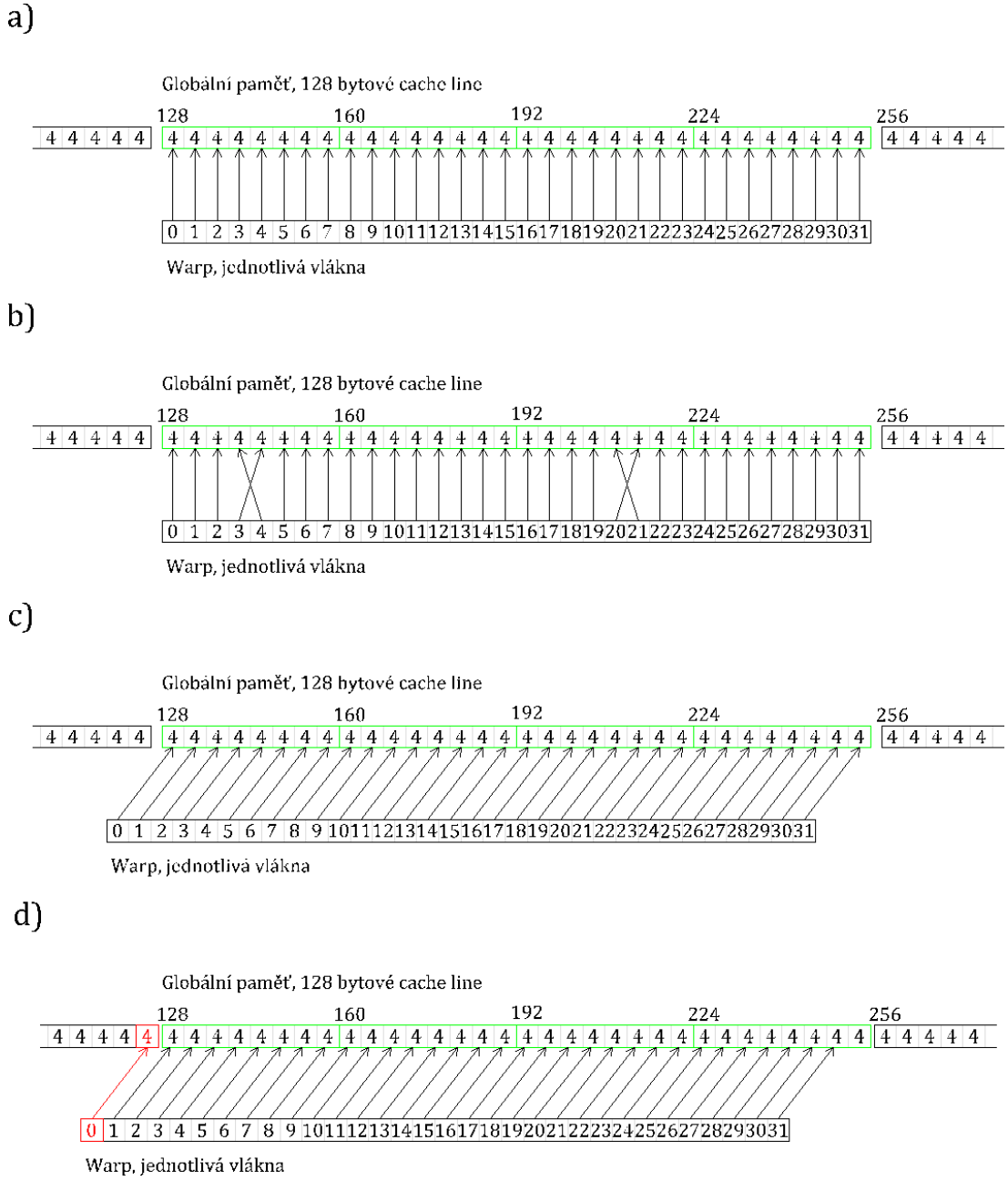
Jedná se většinou o GDDR5 paměť, která je umístěna mimo výpočetní čip, což má největší vliv na latenci, která se pohybuje v řádu stovek cyklů. V porovnání s CPU jde o analogii k RAM pamětem (v našem případě DDR3). Její velikost je v řádu gigabytů a mohou k ní neomezeně přistupovat všechna vlákna z mřížky. Alokovaný blok v globální paměti má životnost od alokace do skončení programu.

Globální paměť je nejcitlivější na manipulaci s daty. Je třeba si uvědomit, že pro umožnění paralelního výpočtu využívající data uložená v globální paměti, čtení i zápis musí být rovněž paralelizovány. Na grafických kartách se tak děje podle jasně daných pravidel, která jsou závislá především na výpočetních schopnostech, jejichž nerespektování může vyústit v sekvenční průběh a následně výrazné zpomalení celého výpočtu.

Přístup do paměti je zprostředkován 32-, 64- nebo 128-bytovými transakcemi, případně jejich kombinacemi. U novějších grafických karet postavených na architektuře Fermi s výpočetními možnostmi 2.0 a vyššími, je přístup do

globální paměti zprostředkován přes *vyrovnávací paměť*, jejíž velikost je 128 bytů. Pro dosažení maximální propustnosti globální paměti je třeba, aby byly všechny požadavky na čtení nebo zápis uspořádány v násobcích velikosti vyrovnávací paměti a první adresa v každém warpu ležela právě na začátku těchto segmentů (obr. 4). Pro upřesnění je důležité podotknout, že každé vlákno z warpu čte nebo zapisuje hodnotu o velikosti 4 byty. Pokud jsou tyto podmínky splněny, je propustnost globální paměti maximální a mluvíme o úplné koalescenci.

Koalescence tedy znamená, že určité části v paměti jsou čteny postupně jednotlivými vlákny ve warpu. Je řešena pro celé warpy, nikoliv pro samotná vlákna a je tedy důležité, aby všechna vlákna v každém warpu spadala právě do jednoho bloku v paměti.



Obrázek 4: Schéma koalescentního přístupu do globální paměti [17]:
(a) přímý sekvenční přístup - plná koalescence,
(b) přímý vystřídaný přístup - plná koalescence (stejný blok mezipaměti),
(c) posunutý přístup - plná koalescence,
(d) posunutý přístup - 2 nezávislé transakce (2 rozdílné bloky mezipaměti).

2.3.2 Sdílená paměť (shared memory)

Je umístěna na každém z multiprocesorů, její latence je tedy velmi nízká. Pokud nedochází ke konfliktům paměťových banků, pohybuje se okolo jednoho až dvou cyklů. Aby se dosáhlo takto vysoké propustnosti, je sdílená paměť rozdělena na stejně velké paměťové moduly, nazývané *banky*, které jsou v čase přístupné současně. Vzhledem k tomu, že na architektuře Fermi se vyhodnocují celé warpy najednou, má sdílená paměť na každém multiprocesoru 32 banků, každý o velikostí 32 bitů.

Každý požadavek na zápis nebo čtení z n adres této paměti, který spadá do n odlišných paměťových banků, může být vyhodnocen současně. Celková propustnost paměti je pak n krát vyšší než propustnost jediného modulu.

Avšak když požadavek na dvě nebo více adres spadá do jediného banku, nastává paměťový konflikt a přístup k paměti probíhá sekvenčně. Pokud nastane tato situace, požadavky s paměťovými konflikty se rozdělí na samostatné bezkonfliktní požadavky, snižující propustnost faktorem rovným počtu samostatných požadavků. Pro čtení ze sdílené paměti existuje výjimka, a to pokud všechna vlákna přistupují k jediné adrese. Hodnota je přečtena pouze jednou a přenesena všem vláknům pomocí tzv. broadcastu.

Celková velikost na každém multiprocesoru je omezena v řádu desítek kilobytů a je rozdělena mezi aktivní bloky každého multiprocesoru. Je přístupná pouze z GPU všem vláknům v daném bloku. Má stejnou funkci jako $L1$ vyrovnávací paměť CPU. Po ukončení funkce bloku vláken, je tato paměť uvolněna pro další bloky.

2.3.3 Paměť pro konstanty a textury (constant, texture memory)

Tento druh paměti využívá místa RAM na grafické kartě a data jsou dále zpracovávána přes vyrovnávací paměť. Jsou jednotné pro všechny multiprocesory, a tudíž k nim mají přístup všechna vlákna. Jsou specifické tím, že přístup pro zápis je povolen pouze z CPU a funkce spuštěná na grafické kartě má přístup jen pro čtení. Jejich latence je srovnatelná s globální pamětí a velikost je v řádu desítek kilobytů.

2.3.4 Registry

Pole registrů je umístěno na každém z multiprocesorů a jejich rozdělení mezi jednotlivá vlákna plánuje nepřímo překladač. Každé vlákno přistupuje pouze ke svým registrům. Tato paměť je velmi rychlá, avšak není přímo přístupná programátorovi, lze pouze omezovat možnosti překladače.

2.3.5 Lokální paměť

Jedná se o obdobu registrů s tím rozdílem, že je tato paměť fyzicky umístěna v globální paměti a je využívána pouze tehdy, když dojde k vyčerpání registrů.

2.3.6 Shrnutí vlastností pamětí

Následující tabulka dává stručný přehled vlastností jednotlivých druhů pamětí

Typ paměti	Umístění	Cache	Přístup	Viditelnost	Životnost
Registry	Na čipu	-	Čtení/Zápis	1 vlákno	Vlákno
Lokální	Mimo čip	Ano	Čtení/Zápis	1 vlákno	Vlákno
Sdílená	Na čipu	-	Čtení/Zápis	Vlákna v bloku	Blok
Globální	Mimo čip	Ano	Čtení/Zápis	Všechna vlákna, host	Do uvolnění
Paměť konstant	Mimo čip	Ano	Čtení	Všechna vlákna, host	Do uvolnění
Paměť textur	Mimo čip	Ano	Čtení	Všechna vlákna, host	Do uvolnění

Tabulka 1: Shrnutí vlastností jednotlivých druhů pamětí [18]

2.4 Hardwarová omezení

Programovací i paměťový model je svázán určitými omezními, konkrétně počet bloků a vláken na multiprocesoru je limitován jak využitím prostředků jednotlivými vlákny nebo bloky, tak výpočetními možnostmi dané grafické karty, viz kapitola 2.5. Následující tabulka shrnuje důležitá omezení

Maximální rozměr mřížky bloků v jedné dimenzi	65535
Maximální x-, y- rozměr bloku vláken	1024
Maximální z-rozměr bloku vláken	64
Maximální počet vláken v bloku	1024
Velikost warpu	32
Počet 32-bitových registrů na multiprocesor	32000
Maximální množství sdílené paměti na multiprocesor	48KB
Počet sdílených paměťových banků	32
Množství lokální paměti na vlákno	512KB
Velikost konstantní paměti	64KB

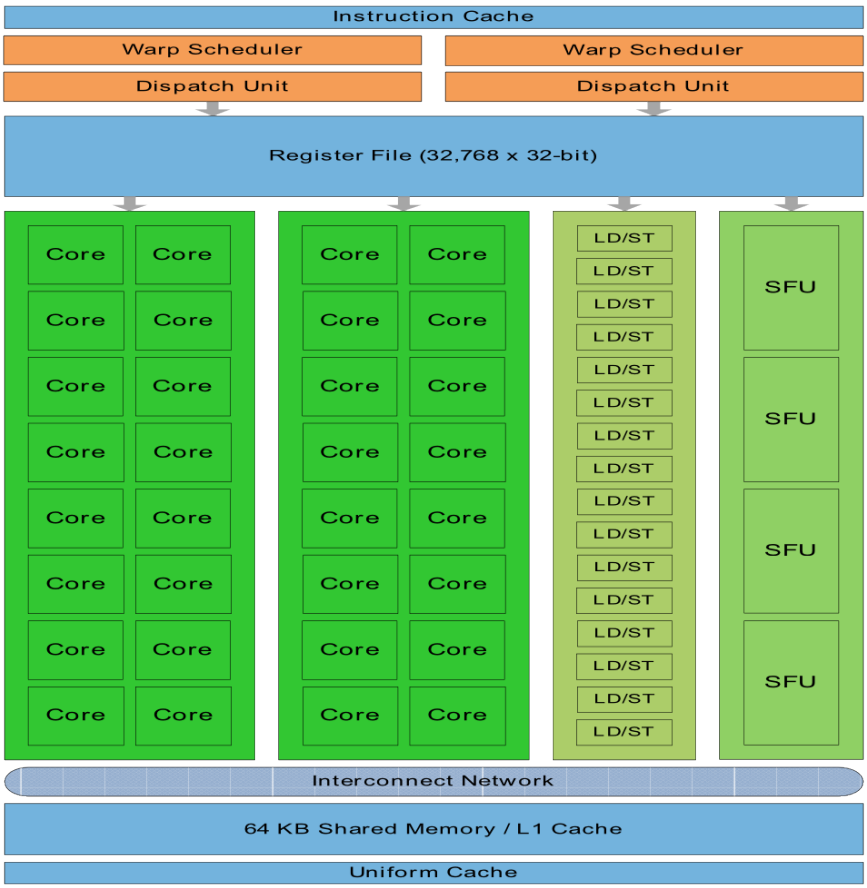
Tabulka 2: Shrnutí hardwarových omezení (Fermi 2.0)

2.5 Výpočetní možnosti

Každá grafická karta má podle architektury, na které je postavena, různé výpočetní možnosti. K rozlišení se používají dvě revizní čísla, kde první značí použitou architekturu a druhé vylepšení jádra dané architektury.

Dále se budeme věnovat pouze architektuře Fermi s výpočetními možnostmi 2.0, která je v současnosti nejpokročilejší počítačovou architekturou GPU. Rozdílnost architektur spočívá zejména ve stavbě multiprocesorů, které se v případě Fermi 2.0 skládají z

- 32 CUDA jader pro aritmetické operace,
- 4 speciálních výpočetních jednotek (SFU - "Special Function Unit"),
- 2 warp plánovačů,
- 16 load/store jednotek a několika pamětí.

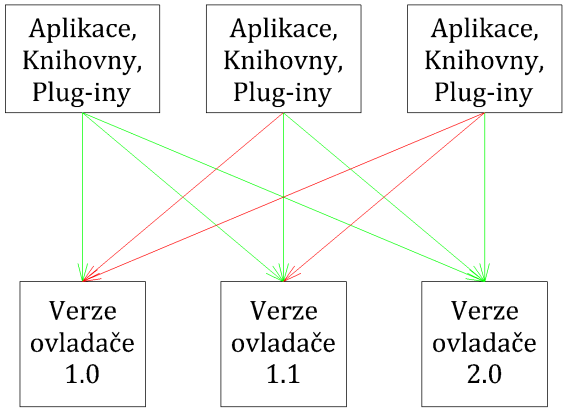


Obrázek 5: Schéma mikroprocesoru architektury Fermi [3]

2.6 Verze a kompatibilita

Existují dvě verze čísel, které určují s čím nebo jestli je daný program kompatibilní se systémem, na kterém je spuštěn. První jsou výpočetní možnosti zařízení, popisující základní hardwarové specifikace a funkce zařízení, zmíněné v předchozí kapitole. Druhé číslo je pak verze CUDA ovladače, který popisuje funkce a možnosti daného ovladače.

Verze ovladače je uvedena v hlavičkovém souboru. To umožňuje programátorovi zkontrolovat, jestli jejich aplikace vyžaduje novější, než aktuálně nainstalovaný ovladač. To je důležité, protože ovladač je zpětně kompatibilní, což znamená, že aplikace, plug-iny a knihovny sestaveny na určité verzi ovladače budou stále funkční s následujícím vydáním ovladače, viz (obr. 6). Zároveň nejsou ovladače kompatibilní směrem vpřed, což znamená, že aplikace, plug-iny a knihovny sestaveny na určité verzi ovladače nebudou funkční s předchozím vydáním ovladače.



Obrázek 6: Kompatibilita zařízení s programem [18]

Stejně jako ovladače se chová i verze výpočetních možností, tj. je zpětně kompatibilní, ale není možné využívat některých nových funkcí na starých zařízeních.

2.7 Optimalizace výkonu

V předchozích odstavcích jsme se dozvěděli, jak rámcově fungují grafické karty a jak je lze využívat pro obecné výpočty. Seznámili jsme se také s mnoha omezeními, které snižují výpočetní výkon. Smysl této kapitoly tedy spočívá v zobecnění těchto omezení a zavedení jednoduchých pravidel, jimiž by se měl programátor řídit.
Optimalizaci výkonu je v zásadě možno dosáhnout třemi základními postupy:

- Maximalizace paralelizace k dosažení maximálního využití.
- Optimalizace využití paměti k dosažení maximální propustnosti paměti.
- Optimalizace instrukcí k dosažení maximální propustnosti instrukcí.

Který ze směrů povede k nejvyššímu nárůstu výkonu pro danou část kódu, vždy záleží na tom, v jaké oblasti dochází právě k největším ztrátám na výkonu. Například optimalizací propustnosti instrukcí u kernelu, který je největší měrou limitován operacemi s pamětí, se nemůže dosáhnout významného zrychlení. Optimalizace by tedy vždy měla být v režii měření a monitorování výkonu, naříklad pomocí CUDA profileru, známého jako *Parallel Nsight*. K optimalizaci propustnosti paměti například slouží poměr teoretické maximální a dosažené propustnosti, který říká, jak velký je prostor pro zlepšení daného kernelu. Je tedy důležité si uvědomit, že se nelze soustředit vždy na jediný směr a ten aplikovat, ale analyzovat vždy celou funkci spuštěnou na grafické kartě.

2.7.1 Maximalizace paralelizace

Pro dosažení maximálního výkonu by funkce spuštěná na grafické kartě měla být postavena tak, aby co nejvíce vnitřních částí této funkce bylo paralelizováno a současně, aby objem této paralelizace byl co největší. Pokud se systém neudržuje zaneprázdněný po většinu času nebo nejsou multiprocesory využity rovnoměrně, dochází ke značným ztrátám výkonu. Hlavním smyslem tohoto

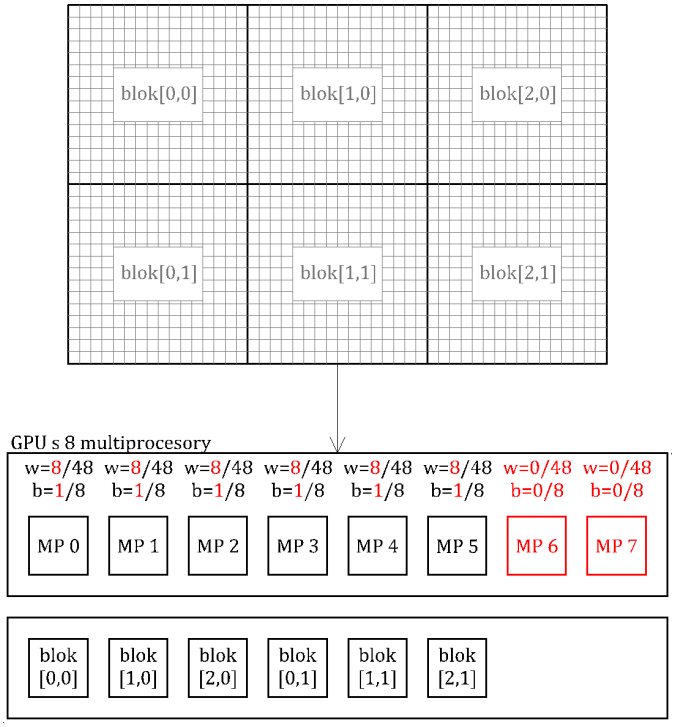
směru optimalizace je tedy vhodně zvolit velikost bloku vláken jak na základě řešeného problému, tak k jedinému multiprocesoru i jejich celkovému počtu. Pro úplnou transparentnost si tento problém rozvedeme. Protože se nejedná jen o softwarovou implementaci, nejdříve si uvedeme hardwarové limity, ze kterých budeme následně čerpat.

warp limit/multiprocesor	48
blok limit/multiprocesor	8
thread limit/blok	1024

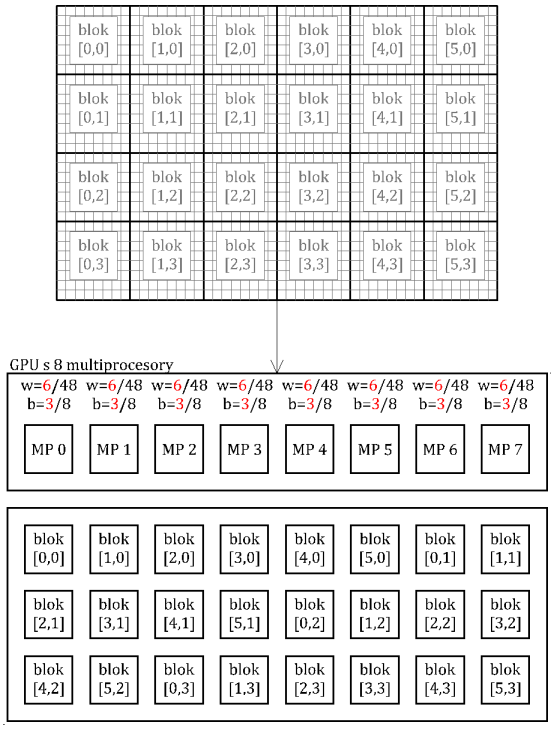
Tabulka 3: Hardwarové omezení (architektura Fermi 2.0)

Plánování spouštění vláken se odehrává na úrovni warpů. Obsazení multiprocesorů je záležitost počtu bloků a to vše je řízeno velikostí problému a počtem vláken na blok. Zkusme tedy optimalizovat příklad z kapitoly 2.2.3, kde jsme uvažovali 3x2=6 bloků, každý po 16x16=256 vláknech. Dnešní grafické karty mají většinou okolo osmi a více multiprocesoru, což by v tomto případě znamenalo, že dva z nich nebudou vůbec využity (6 bloků, 8 multiprocesorů). Je tedy důležité navrhnout takový počet vláken na blok, aby počet bloků v rámci možností přesáhl násobně počet multiprocesorů a zároveň, aby velikost každého bloku byla dělitelná velikostí warpu. V tomto případě bude tedy vhodnější volba 8x8=64 vláken na blok, velikost mřížky pak bude 6x4=24 bloků, viz (obr. 7b).

V tomto případě na úrovni aplikace již nemůžeme zavádět další změny, které by vedly k výraznému urychlení, protože zde narážíme na problém nedostatečného objemu paralelizace a nikdy nebudeme moci dosáhnout plného využití a výkonu dané grafické karty. Dostáváme se tedy na úroveň hardwaru, resp. úroveň multiprocesoru.



(a)



(b)

Obrázek 7: Optimální rozložení bloků vláken pro daný příklad: (a) Původní rozložení; (b) Nové rozložení

Nabízí se tedy otázka: Pokud počítáme zvlášť každý prvek v doméně, jak velká musí být, aby byla jakákoliv grafická karta plně využita? Odpověď je relativně prostá a vychází z hardwarových limitů grafické karty. Velikost domény se vypočítá podle následujícího vztahu

$$D = C_{mp} \cdot W_{mp} \cdot W_{size} \tag{7}$$

C_{mp} je počet multiprocesorů grafické karty,

W_{mp} je maximální počet současně spuštěných warpů na každém multiprocesoru (pro Fermi 2.0 je rovna 48),

W_{size} je velikost warpu (pro všechny architektury rovna 32).

Pro náš konkrétní příklad by pak doména měla obsahovat D prvků

$$D = 8 \cdot 48 \cdot 32 = 12288 \tag{8}$$

Minimální velikost domény zároveň udává maximální počet současně aktivních vláken pro danou grafickou kartu.
Minimální velikost bloku pro danou doménu za současného maximálního využití multiprocesorů je pak vyjádřeno následující rovnicí

$$B_{min} = \frac{W_{mp}}{B_{mp}} \cdot W_{size} \tag{9}$$

B_{mp} je maximální počet současně spuštěných bloků na každém multiprocesoru (pro Fermi 2.0 je rovna 8).

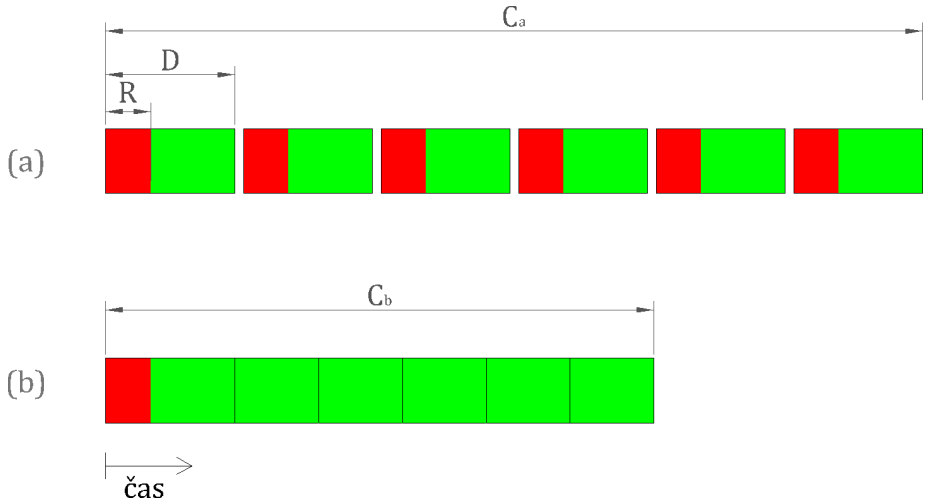
2.7.2 Optimalizace využití paměti

Operace s pamětí jsou jedním z největších inhibitorů grafických karet. Nesprávným nakládáním s pamětí lze degradovat celý výpočet a výrazně ho zpomalit. I přes stálé vylepšování architektury je dopad těchto operací na výpočetní výkon významný a je třeba mu věnovat pozornost. Problém lze hrubě rozdělit do dvou základních skupin, které si dále rozvedeme.

Přesun dat mezi hostitelským zařízením a grafickou kartou

Tento přesun je zprostředkována pomocí PCI Express rozhraní, které má vzhledem k operačním pamětem velmi nízkou propustnost. Nabízí se několik způsobů jak zamezit zbytečným ztrátám na výkonu právě prostřednictvím komunikace přes PCI Expres rozhraní.

Nejdříve je tedy vhodné minimalizovat jakékoliv přesuny paměti pouze na ty nejnutnější a uvědomit si, že režie spojená s každým převodem je pro systém velice náročná. Pokud se tedy nemůžeme vyhnout několika převodům, měli bychom spojit několik malých převodů do jednoho velkého a to i za předpokladu, že daná data v určitou chvíli nepotřebujeme. Je to vždy výhodnější než provádět každý převod samostatně, viz (obr. 8).



Obrázek 8: Časová náročnost spojená s přesunem paměti:
(a) Kopírování samostatných celků; (b) Kopírování jediného celku (R je režie pro zpracování požadavku na přesun paměti, D je čas potřebný ke zpracování jednoho převodu, C_a je celkový čas potřebný k přenesení všech dat, C_b) je celkový čas při přesunu všech dat najednou

Samotné omezení přesunů paměti je možné také přesunutím části kódu z CPU na GPU i za předpokladu, že se sníží paralelizace vnitřních částí této funkce. Tento postup se užívá jen zřídka a pokud máme například kernel, který je volán v cyklech s použitím většiny stejných dat, je výhodné obětovat čas jednotnému přesunu dat před zahájením cyklu, které díky životnosti globální paměti skrze celý běh programu nahráváme na grafickou kartu pouze jednou.

Operace s pamětí během výpočtu

Jak již bylo zmíněno v kapitole 2.3.1 pro dosažení maximální propustnosti paměti je důležité zachovat několik základních pravidel, jako je například koalescence nebo konflikty paměťových banků. Současně by se měl klást důraz na využití paměti umístěné na čipu každého multiprocesoru, jako je například sdílená paměť, viz kapitola 2.3.2.

Poměru instrukcí využívajících paměť na čipu k instrukcím využívajícím paměťový prostor mimo čip se říká aritmetická intenzita. Čím vyšší je tento poměr, tím méně musí být spuštěno warpů na multiprocesoru, aby došlo k eliminaci dopadu vysoké latence globální paměti na výpočetní čas a naopak. Například pro poměr 15, je pro úplné "schování" latence okolo 600 cyklů, což je standardní latence globální paměti, zapotřebí spuštění přibližně 40 warpů na každém multiprocesoru.

2.7.3 Optimalizace instrukcí

Všechny instrukce jsou různě náročné pro systém a to jak z pohledu jednotlivých operací, tak celých kernelů.

Z hlediska jednotlivých operací je vhodné omezit užívání goniometrických a složitějších matematických funkcí, případně je nahradit méně náročnými vnitřními, méně přesnými, ale rychlejšími funkcemi uvnitř kernelu.

Z hlediska celých kernelů je třeba se zaměřit na usměrňování toku instrukcí, jako jsou například *if, do, switch, for* nebo *while* podmínky a cykly, které mohou výrazně ovlivnit propustnost aplikace způsobením odklonu vláken uvnitř warpů, viz kapitola 2.2.3.

3 Vlastní implementace

Základní funkce lineal path bez algoritmu pro rekonstrukci mikrostruktury byla nejdříve vytvořena v jazyce C++ a využívala pouze jednoho jádra procesoru. Autor si na dané sekvenční verzi vyzkoušel některé aspekty programování a výsledná implementace byla využita také pro kontrolu a optimalizaci paralelní verze. Jak již bylo naznačeno, dále byla tato funkce implementována a optimalizována pro využití grafickými procesory za účelem zrychlení výpočtu. Při dosažení dostatečného výkonu, byla funkce implementována do algoritmu pro rekonstrukci mikrostruktury pomocí simulovaného žíhání. Jedná se o pravděpodobnostní optimalizační metoda prohledávání stavového prostoru založená na simulaci žíhání oceli.

Celý problém je velmi komplexní a jeho řešení se skládá z mnoha menších částí, které si zde představíme a některé z nich si rozebereme podrobněji, aby bylo zřejmé, jak algoritmus funguje.

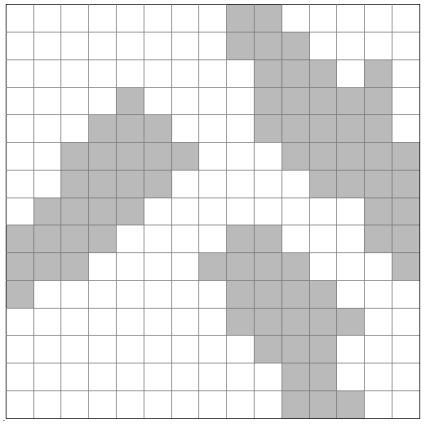
3.1 Lineal path funkce

Základním principem funkce lineal path je prokládání obrázku různě dlouhými a orientovanými vektory a rozhodování, jestli leží v celé své délce ve zkoumané fázi. Přechod z analogového prostředí naráží na určité komplikace, pokud chceme celý problém digitalizovat.

3.1.1 Digitalizace obrazu

Jak již bylo zmíněno v úvodu, práce se zabývá zejména rekonstrukcí mikrostruktury, která je charakteristická fázovou spojitostí. Aby bylo možné takovéto médium nějakým způsobem analyzovat, je zapotřebí ho nejprve přenést do počítače jako digitální obraz skládající se z jednotlivých pixelů, v našem případě pouze černých a bílých. Protože je takovýto obrázek charakterizován barevným RGB modelem s 8 bitovou hloubkou, je každý pixel definován tří-složkovým vektorem nesoucím informaci o zastoupení každé z barev s rozsahem 0 až $(2^8-1) = 255$ možností. Takovéto značení je v prostředí

pouze jednoho odstínu černé a bílé zcela zbytečné a proto se digitalizovaný obraz dále převede na matici nul a jedniček (obr. 9).



0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
0	0	0	0	0	0	0	0	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0	1	1	1	0	1	0
0	0	0	0	1	0	0	0	0	1	1	1	1	1	0
0	0	0	1	1	1	0	0	0	1	1	1	1	1	0
0	0	1	1	1	1	1	0	0	0	1	1	1	1	1
0	0	1	1	1	1	0	0	0	0	0	1	1	1	1
0	1	1	1	1	0	0	0	0	0	0	0	0	1	1
1	1	1	1	0	0	0	0	1	1	0	0	0	1	1
1	1	1	0	0	0	0	1	1	1	1	0	0	0	1
1	0	0	0	0	0	0	0	1	1	1	1	0	0	0
0	0	0	0	0	0	0	0	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0	0	1	1	1	0	0	0
0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
0	0	0	0	0	0	0	0	0	0	1	1	1	0	0

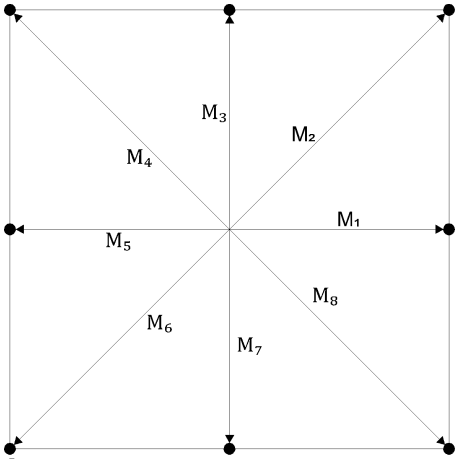
Obrázek 9: Digitalizace média

3.1.2 Generování vektorů

Pod vektory si můžeme představit jednoduše cesty mezi dvěma pevně zvolenými body spojené nejkratší vzdáleností, což bude základ následné aplikace. V analogovém prostředí se čáry jeví jako spojité, to se však mění přechodem do digitálního prostředí, kde se pohybujeme na úrovni pixelů. Body a čáry jsou definovány v tomto prostředí pouze v jednotlivých pixelech. Pro jednoznačné určení jakými pixely bude cesta vedena takovým způsobem, aby byla co nejkratší, byl využit Bresenhamův algoritmus, viz [19]. Pro vyhodnocení celé funkce lineal path je zapotřebí, aby byly do výpočtu zahrnuty cesty všech orientací a všech délek. Počet cest tedy bude záviset na dimenzích vstupní matice a bude roven $2N^2$.

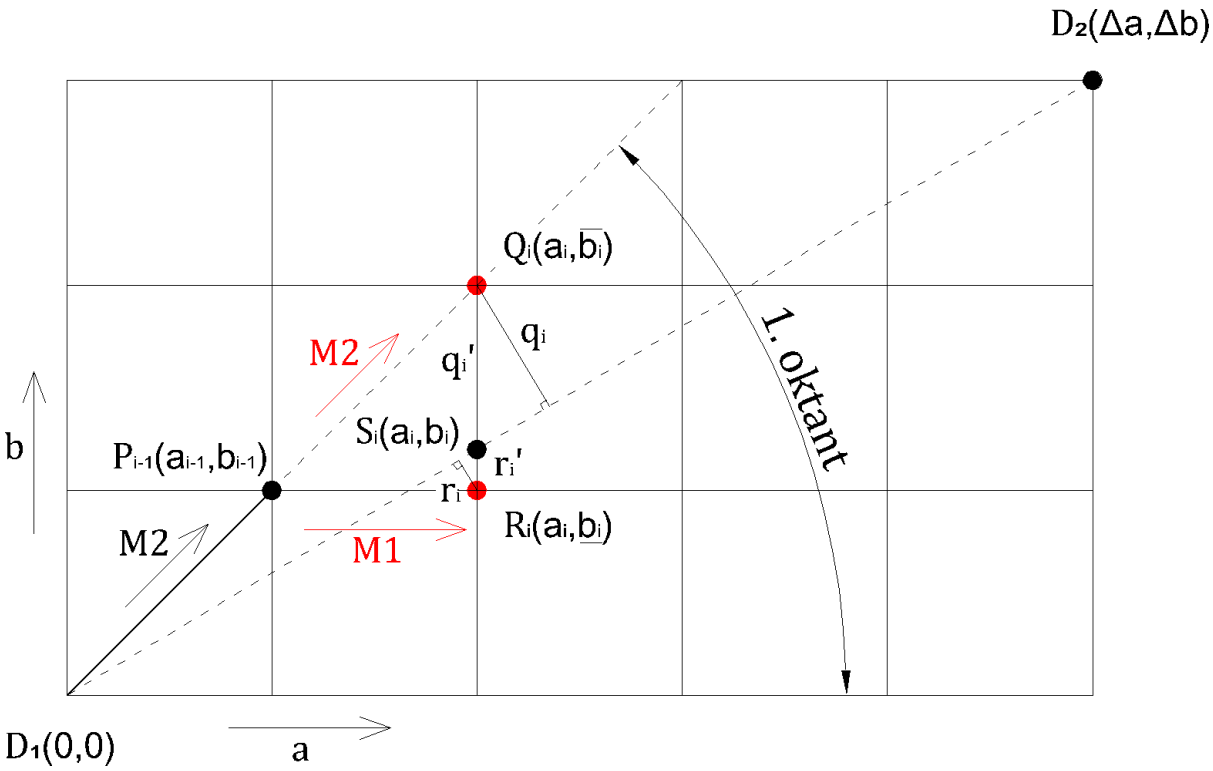
3.1.3 Bresenhamův algoritmus

Celý algoritmus funguje tak, že rozdělí obraz na 8 dílů (obr. 10), kterými je zároveň definováno 8 možných pohybů. Pro naše účely budeme pracovat pouze s prvním oktantem a pohyby $M1$ a $M2$. Zbytek cest se dopočítá zrcadlením.



Obrázek 10: Rozdělení obrazu [19]

Principem algoritmu je porovnání vzdáleností jednotlivých pixelů od referenční čáry mezi body $D1$ a $D2$ (obr. 11).



Obrázek 11: Porovnání vzdáleností pro pohyb plotru [19]

Když algoritmus dospěje do bodu $P_{(i-1)}$, rozhoduje se porovnáním vzdáleností r_i a q_i o pohybu $M1$ do bodu R_i pro $q_i > r_i$, nebo $M2$ do bodu Q_i, pro $q_i < r_i$. Z podobnosti trojúhelníků platí vztah (10) a můžeme tedy zavést

novou proměnnou ∇ s rovnicí (11), která bude rozhodovat podmínkou (12) o následujícím pohybu, viz [19],

$$sign(r_i - q_i) = sign(r'_i - q'_i), \tag{10}$$

$$\nabla_i = (r'_i - q'_i)\Delta a, \tag{11}$$

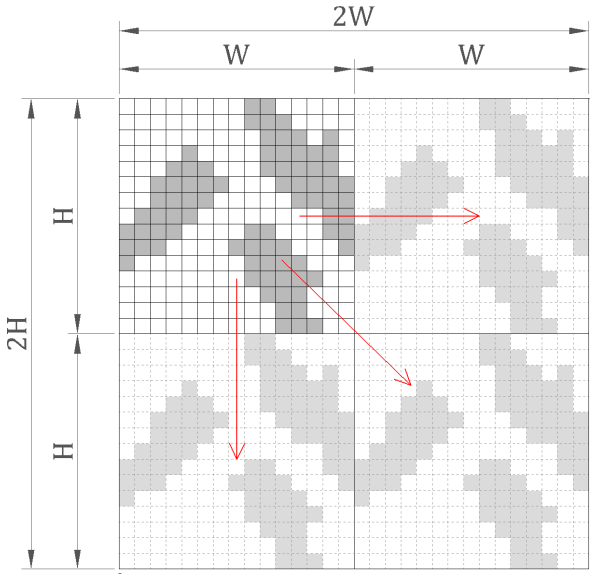
$$pokud \begin{cases} \nabla_i < 0, & \text{pohyb } m_1 \\ \nabla_i \geq 0, & \text{pohyb } m_2 \end{cases}, i = 1, ..., \Delta a \tag{12}$$

Úpravou (11) pro známé souřadnicové body v síti (obr. 11) získáme následující vztah [19]

$$\nabla_i = 2a_{(i-1)}\Delta b - 2b_{(i-1)}\Delta a + 2\Delta b - \Delta a, \tag{13}$$

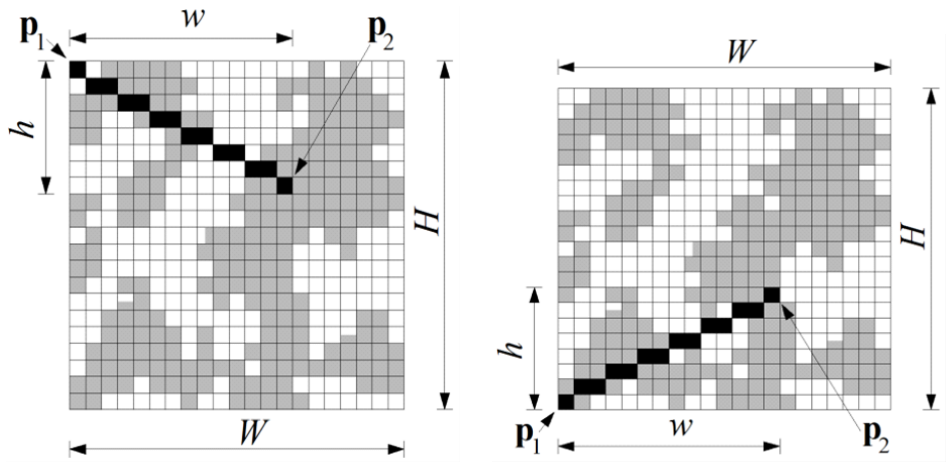
3.1.4 Vyhodnocení funkce lineal path

Popisované médium uvažujeme jako periodické a z toho důvodu budeme počítat se čtyřnásobně velkou maticí podle obrázku (obr. 12).



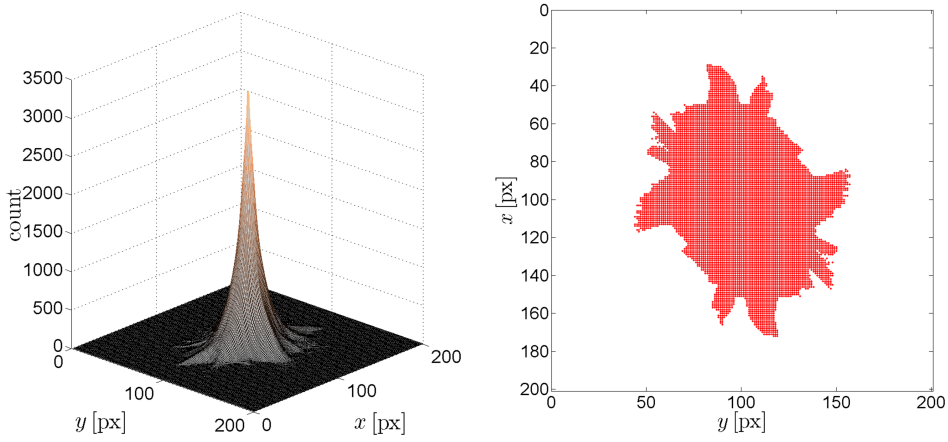
Obrázek 12: Periodicita

Vyhodnocení jako takové spočívá v kontrole jednotlivých cest, tedy souboru pixelů spojující počáteční pixel $\mathbf{p_1}$ s koncovým pixelem p_2 (obr. 13), a v počítání četnosti právě těch, které jsou celé obsaženy ve zkoumané fázi. Současně se každá z cest postupně posouvá přes celý obrázek, čímž se zjistí četnost přes celé médium.



Obrázek 13: Schéma lineal path funkce

Výsledkem je pak matice četností, kde na pozici $\mathbf{p_1} = (0, 0)$ je objemové zastoupení dané fáze a na pozici $\mathbf{p_2} = (w, h)$, je vždy zastoupení vektoru s počátkem $\mathbf{p_1}$ a koncem $\mathbf{p_2}$ v celém médiu.



(a) (b)

Obrázek 14: (a) Výsledná funkce lineal path, (b) Řez v rovině z=0

3.2 Rekonstrukce mikrostruktury

Algoritmus simulovaného žíhání nebyl předmětem této práce a byl kompletně převzat od Ing. Anny Kučerové, Ph.D. Pro naše účely byly měněny pouze jeho základní parametry. Schéma implementace celého algoritmu je znázorněno v následujícím kódu

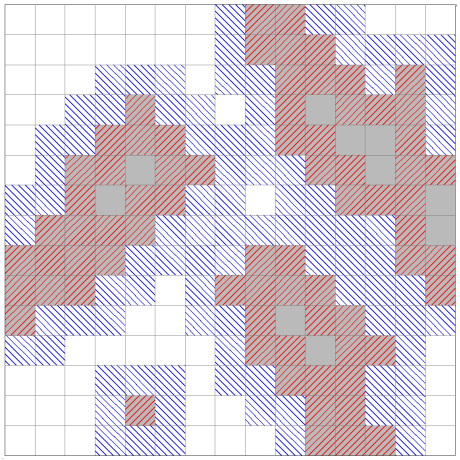
Algoritmus 3: Schéma implementace funkce simulovaného žíhání

```
int main() {
    // Načtení referenčního obrázku
    load(original_image);
    // Generováni náhodné matice
    new_image=NewMatrix(phase1_count);
    // Výpočet funkce lineal path a uložení výsledků
    value=lineal_path(original_image, new_image);
    // Funkce simulovaného žíhání
    SRA(original_image, new_image, value)
}
void SRA(int *original_image, int *new_image) {
    // Cyklus přes předem definovaný počet kroků
    while(n<steps) {
        // Výměna pixelů
        Pixel_flip(new_image);
        // Výpočet a uložení chyby mezi dvěma obrázky
        error=lineal_path(original_image,new_image);
        // Vyhodnocení výsledku
        if( probability || value>error)
            best=error;
        Else
            Flip_back
    }
}
```

3.2.1 Výběr pixelů

Původní algoritmus fungoval pouze způsobem výběru dvou náhodných pixelů odlišných barev. Při bližším pohledu na vývoj rekonstrukce po krocích (obr. 22) zjistíme, že se v původní náhodně generované matici vytvářejí nejdříve shluky, které se následně formují a rozdělují nebo spojují v závislosti na referenční mikrostruktuře. Za předpokladu, že námi rekonstruované médium je tvořeno právě celistvými shluky bez samostatných pixelů, můžeme přistoupit k následujícím úpravám.

Z pozorování vývoje rekonstrukce i ze samotné podstaty funkce lineal path má smysl určitým způsobem usměrnit náhodnost výběru, a to tak, že první vybraný pixel je bílé barvy a nachází se uprostřed černé nebo na hranici bílé fáze. Druhý vybraný pixel pak musí být opačné barvy taktéž na hranici s bílou fází. Logiku výběru pixelů názorně ukazuje obrázek (obr. 15), kdy jsou červenou barvou vyšrafovány všechny první a modrou všechny druhé výběry připadající v úvahu.



Obrázek 15: Výběr pixelů

Výsledkem této úpravy je, že se shluky vytvářejí rychleji a pouze mění svůj tvar. Zároveň je umožněno rozdělování celků a nemůže tak dojít k vytvoření jediného celku, což by pro mikrostrukturu s několika shluky vedlo k zastavení vývoje chyby nezávisle na možnostech optimalizačního algoritmu.

4 Optimalizace

Vzhledem k náročnosti výpočtu bylo nutné přistoupit k určitým optimalizačním krokům. Nejdříve byl optimalizován sekvenční kód, což je shrnuto v následující kapitole. Dále bylo věnováno značné úsilí hardwarové optimalizaci a paralelizaci problému viz kapitola 4.2.

4.1 Softwarová optimalizace

4.1.1 Dimenze

Nejprve byl celý problém převeden ze dvou dimenzí na jednu, kvůli přístupům a hledání jednotlivých segmentů v paměti. Změna byla provedena po řádcích.

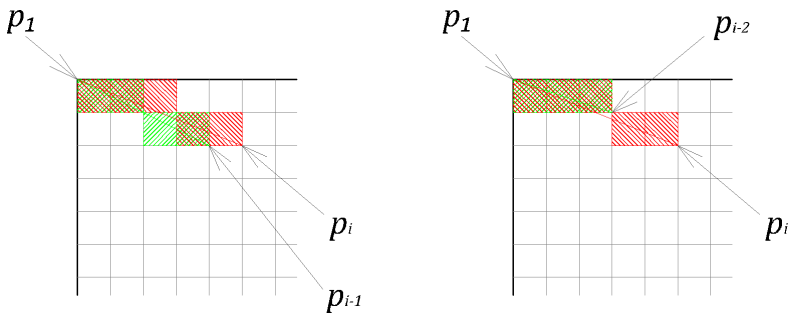
4.1.2 Omezení cest

Vzhledem k charakteru samotné funkce bylo druhým optimalizačním prvkem omezení hranice matice, pro kterou budou počítány cesty. Aby nedošlo ke ztrátám dat ve výpočtu, musí být omezení maximálně na hranici půdorysného obrazu funkce lineal path obr. 14.

4.1.3 Nulové prvky

Posledním optimalizačním krokem bylo vynechání výpočtu cest, které ve výsledné matici narazily na nulový prvek. Všechny další cesty procházející tímto bodem pak vždy musí mít nulovou četnost a nemá smysl je zahrnovat do výpočtu.

Algoritmus funguje tak, že sestaví vektor, přiřazující každému bodu v doméně první možný pixel, jehož cesta zpět do pozice $\mathbf{p_1}$ se úplně shoduje s cestou z výchozího bodu, pro který byl tento pixel určen, viz (obr. 16). Červeně je vyznačen soubor pixelů, pro který se hledá první zpětně shodný pixel. Protože se obě cesty v prvním kroku neshodují, kontroluje se v pořadí další pixel $\mathbf{p_{i-2}}$, jehož cesta je již zcela shodná se zbývající částí referenční



Obrázek 16: Schéma logiky nulových prvků

cesty a výsledek, který se ukládá na místo $\mathbf{p_i}$ ve vektoru nulových prvků, je pozice pixelu $\mathbf{p_{i-2}}$.

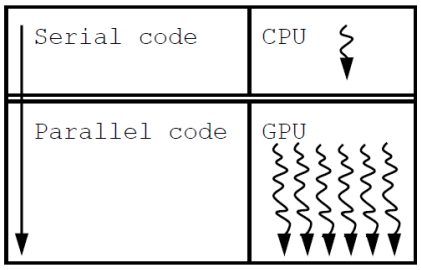
4.2 Hardwarová optimalizace

Paralelizace algoritmu se dělí v zásadě na dvě části. První z nich je samotný paralelní výpočet funkce lineal path a druhá část je hardwarová akcelerace jednotlivých výpočetních prvků.

4.2.1 Paralelizace kódu

Zjednodušené schéma výpočtu funkce lineal path ukazuje obrázek (obr. 17). Je tedy zachována sekvenční část kódu, která generuje jednotlivé cesty, nulové prvky a načítání matice. Nyní si rozebereme paralelní část kódu.

```
1  Generování_cest ();
2  for( i=0; i < počet_cest; i++ ){
3
4      // Paralelní část
5      lineal_path<<< >>>(cesta[i]);
6   }
```



Obrázek 17: Schéma implementace paralelního algoritmu

V kapitole 3.1.4 jsme se seznámili s principem výpočtu funkce lineal path. Nabízejí se v zásadě dvě varianty paralelizace. Asynchronní, kdy je četnost počítána paralelně pro všechny cesty, sekvenčně přes celou doménu a syn-

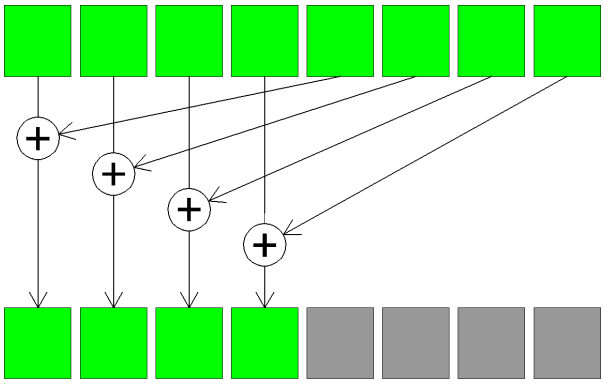
chronní, kdy je četnost počítána paralelně pro jednu cestu přes celou doménu, sekvenčně přes všechny cesty.

Vzhledem k chování grafických karet byla vybrána synchronní varianta, jejíž schéma je vidět na obrázku (obr. 19). Každé vlákno tedy zprostředkovává posun cesty a každý spuštěný kernel má na starosti jedinou cestu.

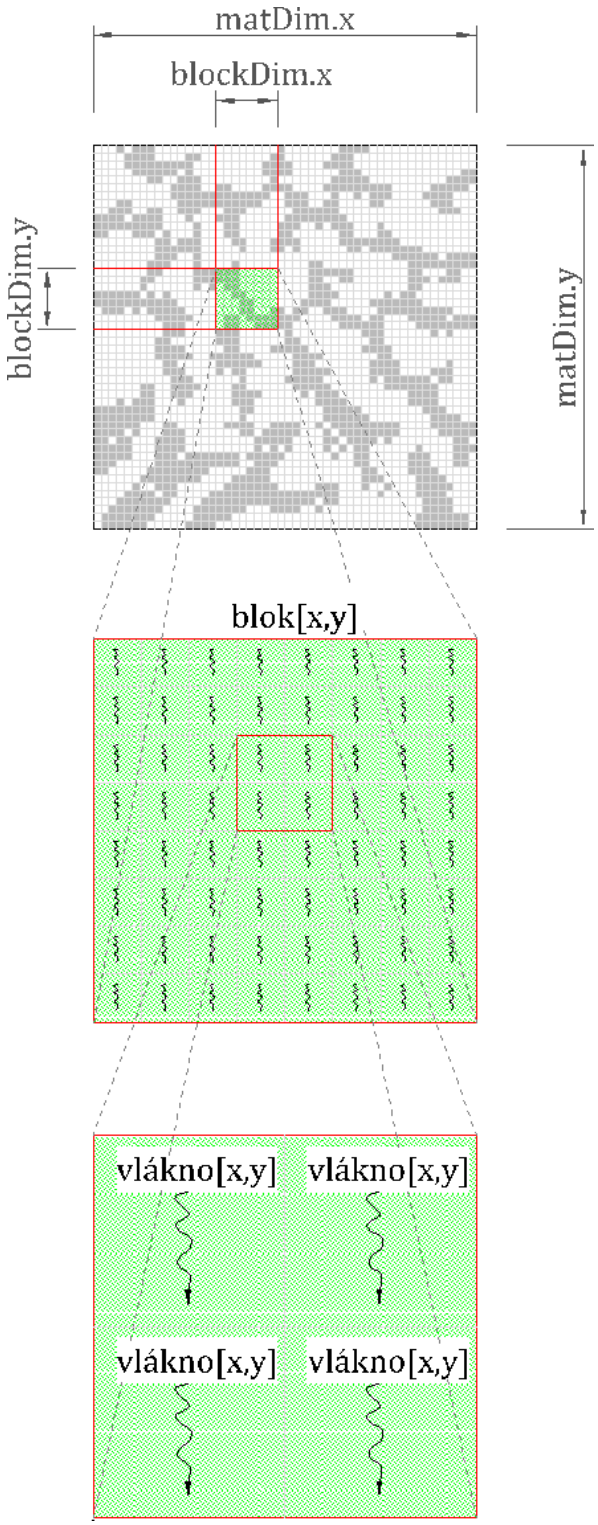
Výsledek každého vlákna je uložen do sdílené paměti podle obecného schématu (obr. 3), kde v každém poli, pro každé vlákno je uložena dílčí četnost jednotlivých posunů. Pro získání celkové četnosti jedné cesty je nutné sečíst všechny hodnoty. Tento součet je zprostředkován paralelně vždy na úrovni jednotlivých bloků, protože stejně jako jednotlivá vlákna, i sdílená paměť náleží různým blokům vláken, které se spouštějí individuálně a nejsou vždy vyhodnoceny najednou v jednom časovém okamžiku. Po součtu jednotlivých řádků bloků je součet sloupců všech bloků proveden sekvenčně pomocí *atomic* funkcí. Schéma paralelizace součtu je vidět na obrázku (obr. 18), počet cyklů potřebných k součetu N prvků se řídí rovnicí (14).

$$C = \log_2(N), \tag{14}$$

kde N je počet prvků, které se sčítají a C je počet cyklů.



Obrázek 18: Schéma jednoho cyklu paralelního součtu [20]



Obrázek 19: Schéma paralelizace

Zjednodušený algoritmus výpočtu lineal path funkce s vysvětlením jednotlivých prvků je uveden níže

Algoritmus 4: Schéma lineal path funkce a paralelního součtu

```
tidx=threadIdx.x, tidy=threadIdx.y;
 __shared__ unsigned int cache[sizex][sizey];
// Pokud je celá cesta ve fázi
if(in phase) {
    count++;
}
// Uložení do sdílené paměti
cache[tidx][tidy] = count;
// Počká na všechna vlákna v bloku
__syncthreads();
int i = blockDim.x/2;
// Paralelní součet
while (i != 0) {
    if (tidx < i)
        cache[tidx][tidy] += cache[tidx + i][tidy];
    __syncthreads();
    i /= 2;
}
//Sekvenční součet prvního sloupce cache a zápis do
   výsledné matice
if ( threadIdx.x == 0 ) {
    atomicAdd(&(mat_vysl[position]),cache[0][tidy]);
}
```

Byl vyzkoušen i algoritmus paralelního výpočtu přes všechny prvky v bloku, ale jak bylo referováno v [17, 20, 18], je náročné uspořádat všechny aritmetické jednotky pro problém s takto nízkou paralelizací a výpočet se tím celkově zpomalí.

4.2.2 Hardwarová akcelerace

Pracovní sestava:
Procesor: Intel Core i7 950 @ 3,07GHz
GPU pro zobrazování: NVIDIA GeForce 210
GPU pro výpočty: NVIDIA Quadro 4000
Operační paměť: 12GB DDR3 1600MHz CL9
Disk: OCZ RevoDrive 80GB
Základní deska: Asus Sabertooth x58
Operační systém: Microsoft Windows 7, 64bit
Verze CUDA: 4.0
Verze ParalelNsight: 1.2

Hardwarová akcelerace spočívala v přetaktování klíčových prvků jako jsou procesor, operační paměti a grafická karta.

Profesionální grafické karty řady *Quadro* vynikají vysokou stabilitou a výkonem. Jde často o podtaktované herní čipy s odblokovanými určitými funkcemi. I za předpokladu přetížení systému je každý hardware vybaven ochranou, která jej ochrání před možným fyzickým poškozením.
Původní takt jádra použité grafické karty NVIDIA Quadro 4000 je 475MHz, takt paměťových čipů 1404MHz a napětí jádra je 900mV.

Jako dlouhodobě udržitelná hodnota taktu jádra pro déle trvající výpočty se stala 730MHz, takt paměťových čipů pak 1628MHz. Tyto hodnoty byly ověřeny v rámci výpočtů trvajících několik dní, kdy byly všechny hardwarové prvky využívány plnou měrou a systém zůstal stabilní.

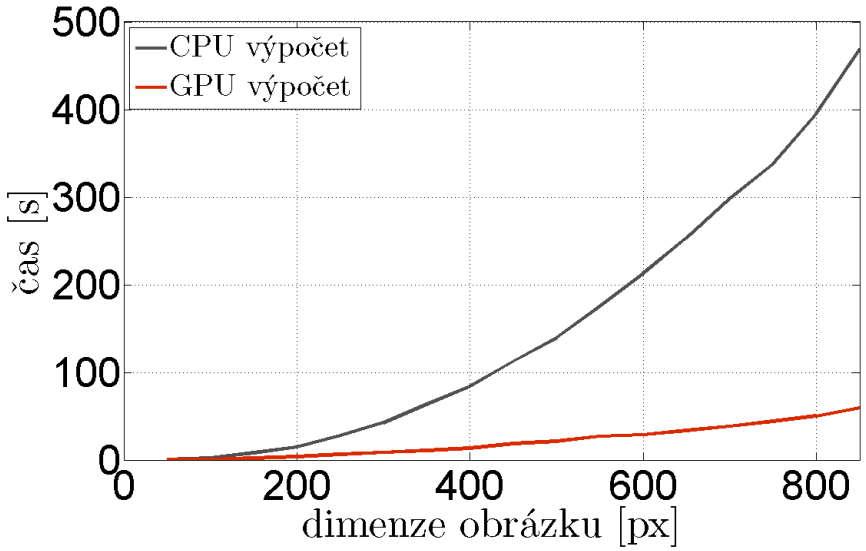
Výsledek zrychlení není závislý na velikosti domény a byl vždy přímo úměrný zvýšení taktu jednotlivých prvků. Celkem se dosáhlo přibližně 54% zrychlení všech výpočtů, což podle očekávání přibližně odpovídá i poměru přetaktování jádra, které má na výpočet oproti taktu pamětí a ostatním prvkům v systému logicky dominantní vliv.

5 Výsledky

5.1 Zrychlení funkce

Veškerá zrychlení výpočtu a uvedené výpočetní časy jsou závislé v zásadě na dvou faktorech. Prvním z nich je použitý hardware, jehož dopad na výpočetní čas lze určitým způsobem odvodit. Druhý faktor je ovšem závislý na volbě mikrostruktury, konkrétněji na zastoupení a rozložení jednotlivých fází. Nelze se tedy na výsledky uvedené níže koukat jako na definitivní. Obecně lze říci, že tvar křivek (obr. 20) zůstává pro různé mikrostruktury stejný a časová závislost na dimenzi vstupního obrázku je vždy exponenciální, stejně jako podíl obou křivek.

Samostatné optimalizační kroky lze od sebe oddělit a zkoumat tím současně jejich vliv na sebe. Pro korektnost ještě zmíníme, že (obr. 20b) a jeho numerické výsledky v tab. 4, jsou počítány se stejnými softwarovými optimalizacemi, na stejném hardwaru a byla vyhodnocována pouze jedna fáze zkoumaného média. Zároveň se jedná o výsledky bez vlivu přetaktování hardwaru.



Obrázek 20: Porovnání výpočtů na CPU a GPU

Standardní				*Optimalizovaná*				
D/ML [px]/[px]	GPU [s]	CPU [s]	**poměr**	D/ML [px]/[px]	GPU [s]	CPU [s]	**poměr**	**celkové zrychlení**
50/50	0.179	0.328	**1.83**x	50/50	0.156	0.265	**1.69**x	**2.10**x
100/100	1.075	4.617	**4.29**x	100/100	0.722	2.371	**3.28**x	**6.39**x
150/150	3.957	21.653	**5.47**x	150/150	1.954	8.018	**4.10**x	**11.08**x
200/200	10.209	66.425	**6.51**x	200/200	3.621	14.742	**4.07**x	**18.34**x
250/250	22.276	169.245	**7.59**x	250/250	6.427	27.612	**4.29**x	**26.33**x
300/300	43.429	357.022	**8.22**x	300/250	8.413	42.338	**5.03**x	**42.44**x
350/350	76.644	649.195	**8.47**x	350/250	10.909	63.304	**5.80**x	**59.51**x
400/400	127.841	1127.897	**8.82**x	400/250	13.481	84.287	**6.25**x	**83.67**x
450/450	209.693	1821.911	**8.69**x	450/250	18.608	122.569	**6.59**x	**97.91**x
500/500	315.951	2846.712	**9.01**x	500/250	21.145	139.698	**6.61**x	**134.64**x

Tabulka 4: Porovnání výpočtů na CPU a GPU (D=dimenze zkoumaného obrázku, ML=maximální délka segmentu)

Dalším zajímavým zjištěním byl fakt, že vlivem rozložení a zastoupení jednotlivých fází se mění poměr softwarového a hardwarového zrychlení a to tím způsobem, že čím je zastoupení zkoumané fáze vyšší a její rozložení spojitější, nabývá hardwarové zrychlení mnohem větší váhy.

5.2 Rekonstrukce

Nejdříve bylo přistoupeno k rekonstrukcím jednoduchých mikrostruktur (obr. 21), aby se ověřila celková funkčnost algoritmu. Dalšími důvody byly pozorování vývoje chyby, přeskupování pixelů a optimální nastavení funkce simulovaného žíhání.

(a) (b) (c) (d)

Obrázek 21: Testovací mikrostruktury

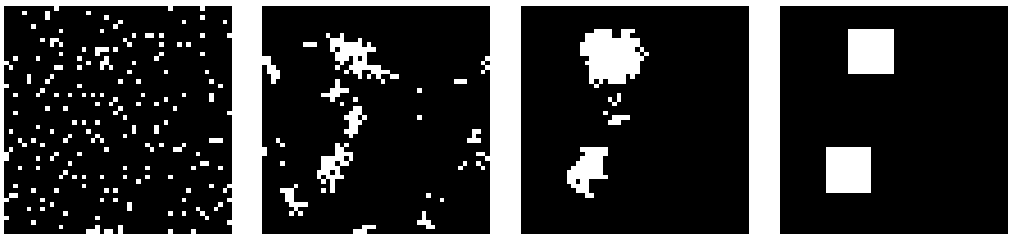
Za účelem ověření funkčnosti algoritmu byla rekonstruována pouze bílá fáze daných mikrostruktur a výsledek by tedy měl obsahovat stejné množství, velikost i tvar bílých objektů. Na ukázku je zde předveden graficky vývoj rekonstrukce (obr. 22) v několika krocích od náhodně vygenerované mikrostruktury až po výslednou mikrostrukturu s nulovou chybou. Pro úplnost si uvedeme nastavení funkce simulovaného žíhání

SRAP.Tmin = 0.000001;

SRAP.Tmax = 0.001;

SRAP.Nstep = 2000000;

SRAP.Nre = 5;



(a) (b) (c) (d)

Obrázek 22: Vývoj rekonstrukce mikrostruktury

Kompletní rekonstrukce bez chyby pro mikrostrukturu (obr. 21b) byla sestavena v 29200 krocích a trvala 17 minut. Druhá mikrostruktura (obr. 21c) byla sestavena v 28000 krocích a trvala 1 hodinu a 8 minut. Z těchto příkladů lze mimo jiné vidět vliv rozložení a zastoupení dané fáze.

Vzhledem k faktu, že jsme byli prakticky bez problémů schopni rekonstruovat mikrostrukturu na základě bílé fáze, lze tím dokázat funkčnost algoritmu. Pro rekonstrukci jako takovou, je ale důležité zahrnout do výpočtu obě fáze, aby bylo vystiženo i vzájemné rozložení obou fází, což je výpočetně mnohem náročnější a chyba má tendenci se ustálit na relativně vysokých hodnotách. Výpočet byl vyzkoušen opět na jednoduchém příkladu

(obr. 21d) a byla rekonstruována nejdříve bílá fáze, která byla posléze použita jako referenční matice pro rekonstrukci černé fáze. Pro rekonstrukci na mikrostruktury na základě obou fází bylo zároveň nutno upravit nastavení funkce simulovaného žíhání následovně

SRAP.Tmin = 7.5;

SRAP.Tmax = 750.0;

SRAP.Nstep = 200000;

SRAP.Nre = 5;

Výsledná mikrostruktura je ukázána na (obr. 23c). Porovnáním všech výsledných obrázků je přímo vidět vliv rekonstruované fáze, kdy pro bílou fázi není respektována tloušťka černé uprostřed bílých čar (obr. 23a), pro černou fázi není naopak respektována tloušťka bílých čar (obr. 23b) a konečně při zohlednění obou fází se díky periodicitě obě čáry jen posunou v jednom směru, ale tloušťky se zachovávají.

(a) (b) (c)

Obrázek 23: Výsledná rekonstrukce testovací mikrostruktury: (a) Na základě bílé fáze, (b) Na základě černé fáze, (c) Na základě obou fází

Tehdy, kdy byla ověřena funkčnost algoritmu a získány určité zkušenosti s nastavením funkce simulovaného žíhání, bylo přistoupeno k rekonstrukci mikrostruktury, pro kterou tento algoritmus vznikal (obr. 24a). Vzhledem ke složitosti dané mikrostruktury byla funkce simulovaného žíhání nastavena následovně

SRAP.Tmin = 15.0;

SRAP.Tmax = 1500.0;

SRAP.Nstep = 1300000;

SRAP.Nre = 5;

Nejmenší chyba, respektive odchylka funkcí lineal path referenční a rekonstruované mikrostruktury, byla pro daný počet kroků 0,62%. Výsledná mikrostruktura je pak na (obr. 24b).

(a) (b)

Obrázek 24: Rekonstrukce spongiózní kosti: (a) Referenční médium, (b) Výsledek rekonstrukce

Rekonstrukce mikrostruktury (obr. 24 b) byla sestavena v 1 032 720 krocích a trvala 3 dny a 54 minut.

Závěr

Tato bakalářská práce byla orientována dvěma směry. Jednak bylo cílem se naučit pracovat s paralelně programovacím nástrojem CUDA a dále pak využít tyto znalosti při rekonstrukci mikrostruktury. Celý problém byl z počátku řešen na úrovni popisu mikrostruktury a byla vypracována jeho sekvenční varianta v jazyce C++, která byla následně optimalizována, viz kapitola 4. Z celé práce bylo věnováno nejvíce úsilí paralelizaci a celkovému urychlení výpočtu za účelem vytvoření výkonného nástroje pro popis mikrostruktury, o kterém pojednává kapitola 4.2.1.

Po dosažení přijatelných výsledků urychlení, viz (tab. 4) a (obr. 20 b), bylo možné opustit úroveň samotného popisu mikrostruktury a přejít tak k rekonstrukci implementací algoritmu do funkce simulovaného žíhání, viz kapitola 3.2.

Nejdříve byly testovány jednoduché příklady, viz (obr. 21 a, b, c, d), aby se ověřila jak funkčnost algoritmu, tak obecně chování funkce simulovaného žíhání. Z těchto příkladů bylo pak přikročeno k několika optimalizačním krokům v logice přeskupování pixelů a vyhodnocení funkce lineal path. Na těchto příkladech se rovněž určilo přibližně optimální nastavení funkce simulovaného žíhání. Výsledné rekonstruované mikrostruktury a postupy jsou popsány v kapitole 5.2.

Posledním krokem bylo tedy přistoupení k rekonstrukci spongiózní kosti (obr. 24 a) na základě obou fází. Výsledné mikrostruktury s nejmenšími chybami oproti referenčnímu médiu jsou uvedeny v kapitole 5.2. V žádném případě nebylo dosaženo úplně nulové chyby, jako tomu bylo u testovacích obrázků, ale dosáhlo se minimální odchylky 0.7% oproti referenčnímu médiu.

Vzhledem k faktu, že se cesty generují jen jednou a zůstávají v paměti po celou dobu výpočtu, je tento výpočet efektivnější a rychlejší, ale dimenze zkoumaného média je tím velmi omezená a to přibližně na obrázky do velikosti 1MPx. Tento problém je možné řešit například algoritmem, který by pokaždé generoval jen chtěnou cestu, nebo složitějším asynchronním algoritmem. Zároveň jsou takto velké obrázky výpočetně velmi náročné (tab. 4).

Dalším pokračováním této práce by tedy mohlo být optimalizování využití pamětí a zrychlení výpočtu cílové funkce, aby mohlo být přistoupeno k rekonstrukci obsáhlejšího média. Vzhledem k výpočetní sestavě, viz kapitola 4.2.2 je spíše prostor pro vylepšení na straně hardwaru, případně jeho rozšiřování.

Literatura

[1] API (04/2012).
URL http://cs.wikipedia.org/wiki/API

[2] Broadcast (03/2012).
URL http://cs.wikipedia.org/wiki/Broadcast

[3] CUDA (03/2012).
URL http://cs.wikipedia.org/wiki/CUDA

[4] FLOPS (02/2012).
URL http://cs.wikipedia.org/wiki/FLOPS

[5] Latence (04/2012).
URL http://cs.wikipedia.org/wiki/Latence

[6] Počítačový klastr (06/2012).
URL http://cs.wikipedia.org/wiki/Počítačový_cluster

[7] Operační paměť (06/2012).
URL http://cs.wikipedia.org/wiki/RAM

[8] Paralelní výpočty (06/2012).
URL http://cs.wikipedia.org/wiki/Paralelní_výpočty

[9] Vlákna (04/2012).
URL http://cs.wikipedia.org/wiki/Thread

[10] S. Torquato, Random heterogenous materials: miscrostructure and macroscopic properties, Springer-Verlag, New York, 2002.

[11] J. Zeman, M. Šejnoha, From random microstructures to representative volume elements, Modelling and Simulation in Materials Science and Engineering 15 (4) (2007) 325–335.

[12] J. Vorel, M. Šejnoha, J. Zeman, Homogenization of plain weave composites with imperfect microstructure: Part II-analysis of real-world materials, arxiv: 10014063v4.

[13] J. Vorel, M. Šejnoha, Evaluation of homogenized thermal conductivities of imperfect carbon-carbon textile composites using the mori-tanaka method, Structural Engineering and Mechanics 33 (4) (2009) 429–446.

[14] O. Jiroušek, D. Vavřík, J. Jakůbek, J. Dammer, Correlation of trabecular bone mechanical properties to its microstructure, using mCT-based FE modeling, Institute of Thermomechanics AS CR, Prague, Svratka, 2008, proc. Engineering Mechanics.

[15] B. Lu, S. Torquato, Lineal-path function for random heterogeneous materials, Physical Review E 45 (2) (1992) 922–929.

[16] Mooreův zákon (04/2012).
URL `http://cs.wikipedia.org/wiki/Mooreův_zákon`

[17] NVIDIA CUDA C Programming Guide, NVIDIA, 2010.

[18] NVIDIA CUDA C Programming Best Practices Guide, NVIDIA, 2009.

[19] J. Bresenham, Algorithm for computer control of a digital plotter, IBM System journal 4 (1) (1965) 25–30.

[20] J. Sanders, E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley Professional, 2010.

Seznam obrázků

Seznam algoritmů