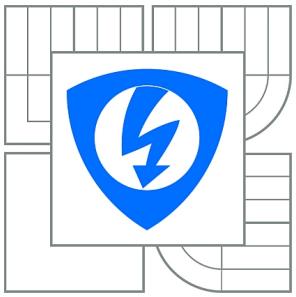
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

DEPARTMENT OF CONTROL AND INSTRUMENTATION

VYUŽITELNOST KNIHOVNY CUDA V PRAKTICKÉM ZPRACOVÁNÍ OBRAZŮ

APPLICABILITY OF THE CUDA LIBRARY IN PRACTICAL IMAGE PROCESSING

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

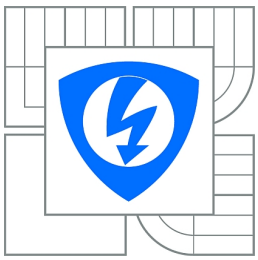
AUTOR PRÁCE — ROBERT KORČUŠKA

AUTHOR

VEDOUCÍ PRÁCE — Ing. STANISLAV KLUSÁČEK, Ph.D.

SUPERVISOR

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav automatizace a měřicí techniky

Bakalářská práce

bakalářský studijní obor
Automatizační a měřicí technika

Student:	Robert Korčuška	***ID:***	136543
Ročník:	3	***Akademický rok:***	2012/2013

NÁZEV TÉMATU:

Využitelnost knihovny CUDA v praktickém zpracování obrazů

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s možnostmi paralelního zpracování obrazových dat. Navrhněte způsob paralelizace alespoň tří jednoduchých obrazových filtrů. Implementujte vybrané algoritmy v jazyce C/C++ pro CPU a GPU. Proveďte srovnání rychlosti výpočtů vzhledem ke složitosti filtru, velikosti masky a velikosti zpracovávaného obrazu. Proveďte diskuzi získaných výsledků. Bakalářská práce bude realizována ve spolupráci s UTEE VUT v Brně.

DOPORUČENÁ LITERATURA:

[1] FARBER, Rob. CUDA application design and development. Morgan Kaufmann, 2011, 315 s. ISBN 978-012-3884-268.
[2] SANDERS, Jason. CUDA by example: an introduction to general-purpose GPU programming. 1st print. Upper Saddle River: Addison-Wesley, 2010, 290 s. ISBN 978-013-1387-683.
[3] Dokumentace platformy CUDA, http://developer.nvidia.com/nvidia-gpu-programming-guide
[4] JAN, J. Medical Image Processing, Reconstruction and Restoration. CRC Press, 2008.

Termín zadání:	11.2.2013	***Termín odevzdání:***	27.5.2013

Vedoucí práce: Ing. Stanislav Klusáček, Ph.D.
Konzultanti bakalářské práce: Ing. Jan Mikulka, Ph.D.

doc. Ing. Václav Jirsík, CSc.
Předseda oborové rady

UPOZORNĚNÍ:

Abstrakt

Táto práca obsahuje základný teoretický rozbor spracovávania obrazu filtrovaním, teóriu paralelného spracovania dát a podrobnejší popis štandardu CUDA. Práca popisuje využitie CUDA v oblasti paralelného spracovania obrazu. V demonštračnej aplikácii porovnáva rýchlosť spracovania obrazu vo funkci bežiacej na CPU oproti funkci na GPU a podrobnejšie popisuje základné metódy pre paralelné programovanie na platforme CUDA.

Klúčové slová

CUDA, filtrovanie, GPU, Paralelné spracovanie dát, spracovanie obrazu.

Abstract

This thesis contains basic theoretical information about image processing, parallel data processing and information about CUDA standard in detail, also desribes aplicability of CUDA in parallel data processing. Testing application compares speed of the image processing in serial CPU application and GPU parallel application, and describes basic methods of parallel programming in the CUDA platform.

Keywords

CUDA, image filtering, GPU, parallel data processing, image processing.

KORČUŠKA, R. *Využitelnost knihovny CUDA v praktickém zpracování obrazů.* Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2013. 55 s. Vedoucí bakalářské práce Ing. Stanislav Klusáček, Ph.D.

Prehlásenie

Prehlasujem, že som svoju bakalársku prácu na tému „Využitelnost knihovny CUDA v praktickém zpracování obrazů“ vypracoval samostatne pod vedením vedúceho semestrálneho projektu, využitím odbornej literatúry a ďalších informačných zdrojov, ktoré sú všetky citované v práci a uvedené v zozname literatúry na konci práce. Ako autor uvedeného semestrálneho projektu ďalej prehlasujem, že v súvislosti s vytvorením tohto semestrálneho projektu som neporušil autorské práva tretích osôb, najmä som nezasiahol nedovoleným spôsobom do cudzích autorských práv osobnostných alebo majetkových a som si plne vedomý následkov porušenia ustanovenia §11 a nasledujúcich autorského zákona č. 121/2000 Sb., o práve autorskom, o právach súvisiacich s právom autorským a o zmene niektorých zákonov (autorský zákon), v znení neskorších predpisov, vrátane možných trestnoprávnych dôsledkov vyplývajúcich z ustanovenia časti druhej, hlavy VI. diel 4 Trestného zákonníka č. 40/2009 Sb.

V Brne dňa: 25. mája 2013 ……

podpis autora

Poďakovanie

Ďakujem vedúcemu bakalárskej práce, pánu Ing. Stanislavovi Klusáčkovi, Ph.D. za možnosť spracovávať túto tému bakalárskej práce a cenné pripomienky v záverečnom spracovaní práce.
Osobitné poďakovanie patrí konzultantovi bakalárskej práce, pánu Ing. Janu Mikulkovi, Ph.D. za ochotu a dobrý prístup ku konzultáciám, účinnú metodickú, pedagogickú a odbornú pomoc a ďalšie užitočné rady pri spracovaní mojej bakalárskej práce.

V Brne dňa: 25. mája 2013 ……

podpis autora

OBSAH

ÚVOD

V dnešnej dobe rapídne narastá množstvo údajov, ktoré je potrebné spracovať. Sériové spracovanie vyžaduje vysoký výpočtový výkon, a preto nastáva potreba tieto množstvá spracovávať efektívnejšie. Paralelné spracovanie nám túto možnosť priamo ponúka.

Táto práca slúži ako úvod k rýchlemu spracovaniu dát a zameriava sa na efektívne využívanie výkonu v spracovaní obrazu. Práca s obrazom je jedna z mnohých úloh, ktoré paralelizmom dokážeme podstatne urýchliť. Práca sa zameriava na spracovanie obrazu pomocou GPU technológie CUDA, ktorú vyvinula spoločnosť Nvidia. Dôležitosť zrýchľenia výpočtov je viditeľná najmä v automatizovaných prostriedkoch, analýze a vyhodnocovaní snímaného obrazu. Úprava obrazu hrá dôležitú úlohu tam, kde z neupravených dát nie je možné odhadnúť výsledok. Napríklad v lekárskom vybavení. Tam kde je nutné z obrazu určiť diagnózu a obraz nie je dokonale kontrastný, nám pomáhajú rôzne filtrácie a detekcie hrán. Rýchlosť je dôležitým faktorom, pretože v medicíne je dôležitá každá sekunda. Paralelné spracovanie na GPU nám umožňuje podobné úpravy s minimálnym oneskorením.

V prvej kapitole sa popisuje obecne spracovanie obrazu. V druhej je vysvetlené paralelné spracovanie dát na GPU a tretia detailnejšie popisuje štandard CUDA. V ďalších kapitolách je štandard CUDA využitý v aplikácii na spracovanie obrazu, ktorá porovnáva spracovanie obrazu na CPU a GPU. Spracovanie na grafickej karte je ďalej optimalizované tak, aby sa dosiahol čo najvyšší výkon. Celá aplikácia je podrobne popísaná spolu s metódami na spracovanie obrazu. V poslednej kapitole sú analyzované výsledné rýchlosti spracovania a porovnaná ich efektivita.

1 SPRACOVANIE DIGITÁLNEHO OBRAZU

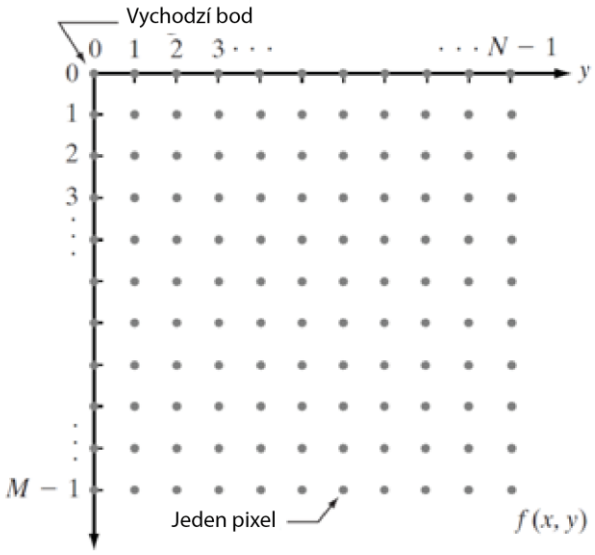
Predmetom nášho záujmu sú metódy predspracovania obrazu, využívajúce pre výpočet jasu bodu vo výstupnom obraze len lokálne okolie zodpovedajúceho bodu vo vstupnom obraze. Časť operácií lokálneho predspracovania sa tiež nazýva aj filtrácia, podľa zavedenej terminológie teórie signálov. Pod časť filtrácie obrazu patrí mnoho metód. Obecne ich vieme rozdeliť na dve kategórie, potlačenie šumu – dolná priepusť a zvýraznenie hrán – horná priepusť. Budeme sa zaoberať vyhladzovaním obrazu ktoré patrí do skupiny ktorá šum potláča, a gradientnými operáciami ktoré zvýrazňujú hrany.

Pod vyhladzovaním obrazu rozumieme potlačenie vyšších frekvencií v obraze, čo je potlačenie náhodného šumu. Zároveň však môže dochádzať k degradácii ostrosti obrazu – náhlych zmien jasovej obrazovej funkcie, ako napríklad ostré hrany, body, čiary, ktoré majú dôležitý význam.

Gradientné operácie majú opačný efekt, zdôrazňujú vyššie frekvencie. Sú zvýraznené tie obrazové elementy, v ktorých sa obrazová funkcia náhle mení. Výsledkom je zvýraznenie hrán v obraze. Nevýhodou týchto operácií je, že sú zvýraznené aj šumové body. Kombináciou týchto gradientových a vyhladzovacích operácií dokážeme obraz vyhladiť a ostriť súčasne. Takýto postup predstavuje napríklad algoritmus rotujúcej masky.

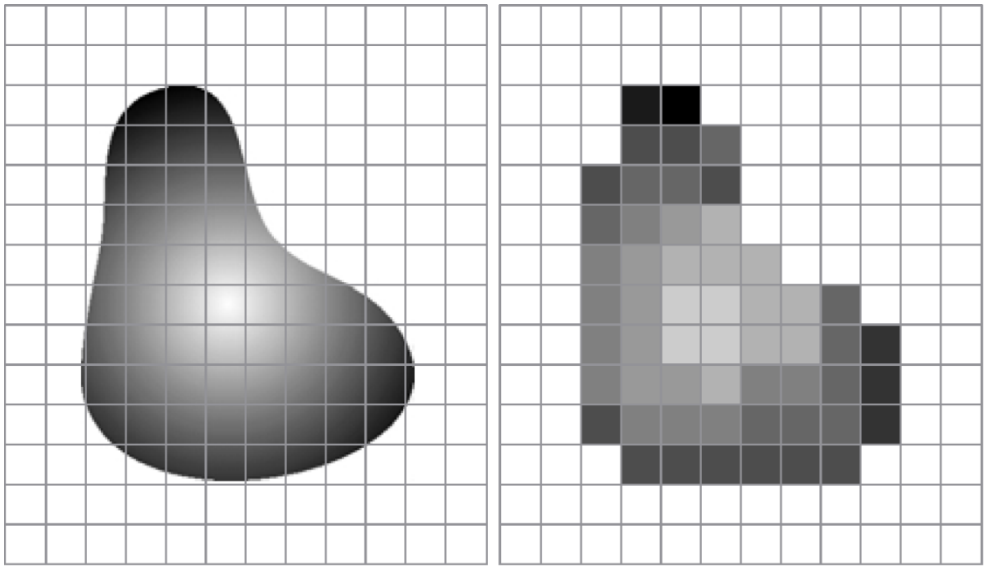
1.1 DIGITALIZÁCIA OBRAZU

Pri snímaní obrazu, napríklad fotoaparátom, dochádza k potrebe diskretizovať snímaný obraz tak, aby ho bolo možné uložiť do pamäte. Snímaný obraz je kontinuálny $f(x, y)$ a dochádza k prevedeniu na diskrétny (digitálny) tvar podľa možností obrazového snímača. Obraz sa prevádza na tzv. pixmapu o rozmeroch M, N, kde každý bod predstavuje jeden pixel.



Obrázok 1: Pixmapa snímanej scény. [1]

Samotné vzorkovanie prebieha výberom hodnoty (pixelu) v určitom okamihu *f*(*x*, *y*) tak, aby korešpondovala s mapou snímača, viď obrázok č. 2.



Obrázok 2: Digitalizácia snímaného obrazu. [1]

Daným vzorkovaním dochádza k mnohým nepresnostiam, podľa obrázku vidíme, že hrany sú nerovnomerné, hranaté a je možné týmto procesom stratiť dôležité informácie. Napríklad v medicíne, pri rôznych snímaniach ľudského tela je dôležité zachytiť každý detail, čo nám pri nedostatočnej kvalite snímača spôsobuje nemalé problémy. Obzvlášť ak na snímač pôsobia rôzne nepriaznivé vplyvy, ktoré znemožňujú nasnímať dokonalý obraz. Jedná sa o rôzne typy šumu, chyby senzorov, radiácie. Na to aby sme znížili tieto

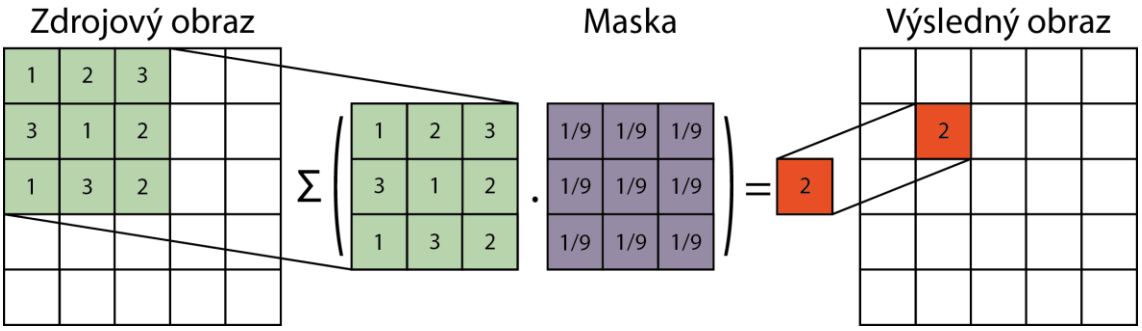
vplyvy pred samotným finálnym výstupom, používame rôzne predspracovania a filtrácie podľa našich požiadaviek. [1]

1.2 DISKRÉTNA KONVOLÚCIA

Pri úpravách obrazu pracujeme s bodmi a preto hovoríme o diskrétnom spracovaní. Pod konvolučným filtrom rozumieme takú metódu spracovania obrazu, ktorá systematicky prechádza celý obraz a na výpočet novej hodnoty bodu využíva malé okolie O reprezentatívneho bodu. Táto hodnota je zapísaná do nového obrazu. Diskrétna konvolúcia má tvar:

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x+s, y+t), \quad (1)$$

kde f je obrazová funkcia pôvodného obrazu, g je obrazová funkcia nového obrazu, w sa nazýva konvolučná maska alebo konvolučné jadro a udáva koeficienty jednotlivých bodov v okolí O. Najčastejšie sa používajú obdĺžnikové masky s nepárnym počtom riadkov a stĺpcov, pretože v tom prípade môže reprezentatívny bod ležať v strede masky.



Obrázok 3: Princíp diskrétnej konvolúcie.

Matica, ktorá obsahuje hodnoty jednotlivých pixelov, je spracovávaná konvolúciou s maskou ktorá pokrýva okrem počítaného pixelu aj okolie. Konvolučná maska sa aplikuje na okolí každého pixelu samostatne. Pre riešenie krajných hodnôt kde nie je možné získať pre výpočet všetky body okolia sa používajú rôzne metódy:

- Obraz je zväčšený o polovicu veľkosti masky a okraje sa doplnia nulami.
- Maska sa pohybuje v rozsahu obrazu, kde výsledkom je zmenšený obraz.
- Zriedkavo sa využíva zmenšená maska.
- Využitie zrkadlenia obrazu.

Transformácie v lokálnom okolí bodu sa delia na dve skupiny:

Vyhladzovanie – tieto metódy sa snažia potlačiť šum v obraze, ale rozostrujú hrany.

Ostrenie – detekcia hrán a čiar, ktorá zosilňuje šum.

Podľa matematických vlastností môžeme metódy predspracovania rozdeliť na:

Lineárne metódy – novú jasovú hodnotu bodu počítajú ako lineárnu kombináciu vstupných bodov, napr. priemerovací filter.

Nelineárne metódy – berú do úvahy len body s určitými vlastnosťami, napr. mediánový filter.

1.3 VYHLADZOVACIE FILTRE

Pri filtrácií šumu predpokladáme, že susedné body v nezašumenom obraze majú rovnakú, alebo pomerne blízku hodnotu jasu. Zašumené obrazové prvky potom môžeme opraviť na základe okolitých bodov. Metóda sa uplatní tam, kde potrebujeme potlačiť šum a odstrániť rôzne osamotené chyby bodov v obraze.

Priemerovací filter

Najjednoduchšou metódou vyhladzovania obrazu je priemerovanie. Novú jasovú hodnotu získame ako aritmetický priemer jasových hodnôt z okolia O. Najčastejšie používané masky veľkosti 3x3:

$\frac{1}{9} \times$

1	1	1
1	1	1
1	1	1

$\frac{1}{16} \times$

1	2	1
2	4	2
1	2	1

Obrázok 4: Masky primerovacích filtrov. [6]

Obrázok č. 4 zobrazuje dva 3x3 vyhladzovacie filtre. Maska naľavo robí štandardný priemer pixlov pod maskou. Jednoducho môžeme koeficienty z masky prepísať do rovnice:

$$R = \frac{1}{9}\sum_{i=1}^{9} z_i. \qquad (2)$$

Z hľadiska výpočtových schopností hardwaru a náročnosti operátora delenia, je výhodné prvky najprv sčítať a potom ich vydeliť ako celok. Pri opačnom postupe sa výpočtový čas predlžuje.

Druhá maska zobrazuje vážený priemer. Týmto spôsobom sa pixelom priradí odlišný koeficient podľa toho, ktorý pixel má vyššiu dôležitosť pred ostatnými. V tejto maske stredný pixel má najvyššiu váhu, rohové pixely najnižšiu a po výpočte majú menší vplyv na výsledok.

Konvolučné masky veľkosti 5x5, 7x7 sa vyrábajú analogicky.

Nevýhodou tejto metódy je, že v značnej miere rozostruje hrany. Tento problém sa rieši použitím rotujúcej masky. Okolo reprezentatívneho bodu rotuje malá maska. Pre každú masku sa vypočíta jasový rozptyl. Nová hodnota sa vyráta podľa masky s najmenším rozptylom. Takto sa nepoškodia hrany a metóda má dokonca mierne ostriace účinky.

Ďalšou metódou na riešenie problému rozostrenia hrán je priemerovanie s prahovaním. Myšlienkou tejto metódy je, že rozdiel medzi novou a starou hodnotou jasu nesmie presahovať prahovú hodnotu T.

Mediánový filter

Je to nelineárna metóda vyhladzovania, ktorej cieľom je eliminovať veľké jasové rozdiely v okolí bodu. Jasové hodnoty bodov vpadajúce do filtračnej masky sa usporiadajú podľa veľkosti. Nová jasová hodnota bude medián tejto postupnosti. Mediánový filter je vhodný na potláčanie impulzného šumu. Nevýhodou metódy je, že sa môžu poškodiť tenké čiary a ostré rohy. Najčastejšie sa používa maska 3x3. [2]

1.4 OSTRIACE FILTRE

Prudké zmeny prechodov jasu zodpovedajú hranám v obraze. Úlohou ostriacich filtrov je nájsť lokalitu, v ktorej dochádza k veľkej zmene jasu. Hrana je vektorová veličina a je určená veľkosťou a smerom. Tieto veličiny vychádzajú z gradientu obrazovej funkcie v obrazovom elemente. Je nutné podotknúť že ostrenie hrán obrazu je nezávislé na vlastnostiach objektu, ktorého hranica má byť určená. Určenie samotných hraníc objektu patrí už do problematiky segmentácie obrazu.

Operátory pre detekciu hrán v digitálnom obraze vychádzajú z parciálneho diferenciálneho operátoru. Prevedením parciálnej derivácie v smeroch x a y získame vektor, ktorý udáva veľkosť Δg (3) a smer gradientu ϕ (4):

$$|\Delta g(x,y)| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}, \tag{3}$$

$$\varphi = \arctan \frac{\left(\frac{\partial f}{\partial y}\right)^2}{\left(\frac{\partial f}{\partial x}\right)^2}. \quad (4)$$

Vykonávať pre každý bod obrazu zložité derivácie je z hľadiska výpočtovej techniky nepraktické a veľmi zdĺhavé. Preto sa využíva hranový operátor.

Sobelov operátor

Táto metóda je príkladom aproximácie digitálneho gradientu, aproximuje prvú deriváciu a preto je smerovo závislý. Odhaduje prvú deriváciu, na to využíva niekoľko masiek. Smer gradientu sa odhaduje hľadaním tej masky, ktorá zodpovedá najväčšej veľkosti gradientu. Operátor je možné vytvoriť pre rôzne veľkosti. Pre kompletné spracovanie jedného bodu je nutné aplikovať masky zo všetkých ôsmich smerov. Pri aplikácií špecifických smerových masiek môžeme detekovať konkrétne smery hrán.

Základné dve masky Sobelovho operátora rozmeru 3x3:

−1	−2	−1	−1	0	1
0	0	0	−2	0	2
1	2	1	−1	0	1

Obrázok 5: Masky Sobelovho operátora. [1]

Masky pre ďalšie smery vzniknú tým že pootočíme masku. [2]

2 PARALELNÉ SPRACOVANIE DÁT

Boli časy keď sa na paralelné výpočty pozeralo ako na exotickú záležitosť, pretože sa veľmi zriedkavo používali v počítačovej vede. V posledných rokoch sa pohľad na paralelné výpočty zmenil. Počítačový svet sa uberá smerom kde takmer každý začínajúci programátor musí byť zbehlý v paralelnom programovaní, aby jeho práca bola efektívna.

Paralelizmom sa začalo zaoberať hlavne po dosiahnutí hranice technických možností zvyšovania frekvencie výpočtovej jednotky. Jedna z príčin je zvyšovanie produkcie odpadového tepla so stúpajúcou frekvenciou a rastúca energetická náročnosť. Táto skutočnosť viedla k zvyšovaniu počtu výpočtových jadier, čo má za následok nutnosť použitia paralelných výpočtov. [3]

2.1 VYUŽITIE PARALELNÉHO SPRACOVANIA DÁT V SPRACOVANÍ OBRAZU

Paralelné spracovanie dát sa dlhú dobu využívalo len v špecializovaných aplikáciách. V súčasnej dobe nachádzame uplatnenie vo všetkých smeroch výpočtových aktivít. Najväčšie využitie má paralelne spracovanie pri potrebe výpočtu v reálnom čase, napríklad dekompresia videa, filtrovanie signálov, šifrovanie. Významne sa urýchľuje aj spracovanie obrazu, kde je nutné pri úpravách používať komplexné algoritmy na každý bod obrazu. Rýchlosť spracovania tu zohráva veľkú rolu, najmä v situáciách keď je každá ušetrená sekunda dobrá, napríklad v medicíne.

2.2 PARALELNÉ SPRACOVANIE OBRAZU A NÁROKY NA HARDWARE

Na zrýchlenie paralelných výpočtov potrebujeme čo najviac exekučných jadier, aby sa úloha mohla vykonať čo najrýchlejšie. Najdôležitejšou časťou výpočtového hardwaru pre spracovanie obrazu je CPU (Central Processing Unit), pre ich univerzálnosť a rýchlosť. Podľa Moorovho zákona sa CPU výkon zdvojnásobí každé dva roky, avšak tento trend dosiahne fyzikálne hranice výroby, a nebude možné zvyšovať frekvenciu a preto sa využívajú iné spôsoby ako zvýšiť výkon procesora. Trendom je rast možnosti spracovávať viac úloh súčasne, výpočtové jednotky pre vektorové spracovanie, zvýšenie počtu jadier CPU. V dnešnej dobe sú grafické procesory (GPU) považované za veľmi efektívne v paralelných výpočtoch. Ich pôvodným zámerom bolo poskytnúť grafický výstup pre užívateľov a ich výpočtové schopnosti sa stali komplexnými a prekonali štandardné CPU vo výpočtovej rýchlosti aj pamäťovej priepustnosti. GPU výrobná technológia nie je obmedzená technickými možnosťami a tak sa výkonový rozdiel medzi CPU stále viac rozširuje. GPU architektúra sa zakladá na masívnej

paralelizácii jednej úlohy a je schopná vykonávať tisíce vláken v rovnakom čase. Z toho dôvodu vznikla požiadavka na využitie grafického procesora pre všestranné použitie – GPGPU.

Spracovanie obrazu je ideálne pre implementáciu na GPU procesory, pretože paralelizácia je poskytovaná na per-pixel operácie. Týmto sa výkon procesora využije naplno. Nevýhoda GPU je jeho nízky výkon v double-precission operáciách, čo pri spracovaní obrazu prehliadame. Najväčšie obmedzenie je vo veľkosti globálnej pamäti, ktorej má menej ako CPU (približne 4 – 8 krát menej). Pri množstve spracovávaných dát je možné ľahko prekročiť rozsahy GPU pamäti najmä pri kontinuálnych realtime úpravách, alebo 3D obrazoch. Tento problém ale nepredstavuje prekážku v implementácii obrazových algoritmov na GPU. Veľká časť algoritmov na spracovanie obrazu potrebuje k vykonaniu operácie len lokálne informácie, napríklad malú časť z obrazu najbližších susedov konkrétneho pixelu. Je teda možné spracovať celý obraz po segmentoch a tie pospájať do výsledného obrazu. Príklady algoritmov sú lineárne a nelineárne filtre s relatívne malými jadrami, detekcia hrán, kombinácie obrazov a ďalšie. V tejto práci sa budem venovať využitiu paralelizácie na GPU v aplikácii filtračných algoritmov. [3]

3 CUDA

Spolu s využitím grafického procesora pre všeobecné využitie bolo nutné vytvoriť aj programovací jazyk schopný využiť schopnosti grafického procesoru. Vzniklo ich viacero (OpenGL, OpenCL), najviac optimalizovaným na rýchlosť spracovania je CUDA od výrobcu grafických čipov Nvidia. Nevýhodou tohto programovacieho jazyka je viazanosť na konkrétny hardware od rovnakého výrobcu. V tejto kapitole sú popísane základné údaje a hlbší pohľad na možnú aplikáciu a využitie pre spracovanie obrazu.

3.1 ŠTANDARD CUDA

CUDA (Compute Unified Device Architecture) je paralelná výpočtová architektúra vyvinutá spoločnosťou Nvidia pre univerzálne použitie. Samotný štandard vznikol v roku 2007. Použitie tohto štandardu je viazané na GPU od spoločnosti Nvidia, čo nám zaručuje využitie výpočtových schopností s veľkou účinnosťou. Rozdiel oproti univerzálnym štandardom závisí od vhodnej optimalizácie a ďalších parametrov, pohybuje sa v rozmedzí jednotiek až desiatok percent. Programovací jazyk pre CUDA vychádza zo štandardu jazyka C, pričom jeho štruktúra je podobná C alebo C++. Výhodou pre programátorov je aj široké spektrum nástrojov pre vývoj softwaru pre CUDA štandard, napríklad CUDA SDK Development Kit, debugger, rôzne vedecké knihovne napríklad CUFFT a CUBLAS.

Komunikácia medzi štandardným systémom a GPU je zabezpečená pomocou CUDA aplikácií. Vďaka multitaskingu sa operačný systém stará o prístup ku GPU pomocou konkrétnych appletov. Sú to hardware driver (CUDA driver), Application Programming Interface (API) a jeho runtime (CUDA Runtime). Vykonávacia aplikácia podľa potreby využíva CUDA aplikácie aby efektívne využívala prostriedky GPU. [4]

Program pre CPU ma nasledujúcu podobu:

```
void increment_cpu (float *a, float b, int N)
{
     for (int idx = 0; idx < N; idx++ )
          a[idx] = a[idx] + b;
}
void main()
{
…
increment_cpu(a,b,N);
}
```

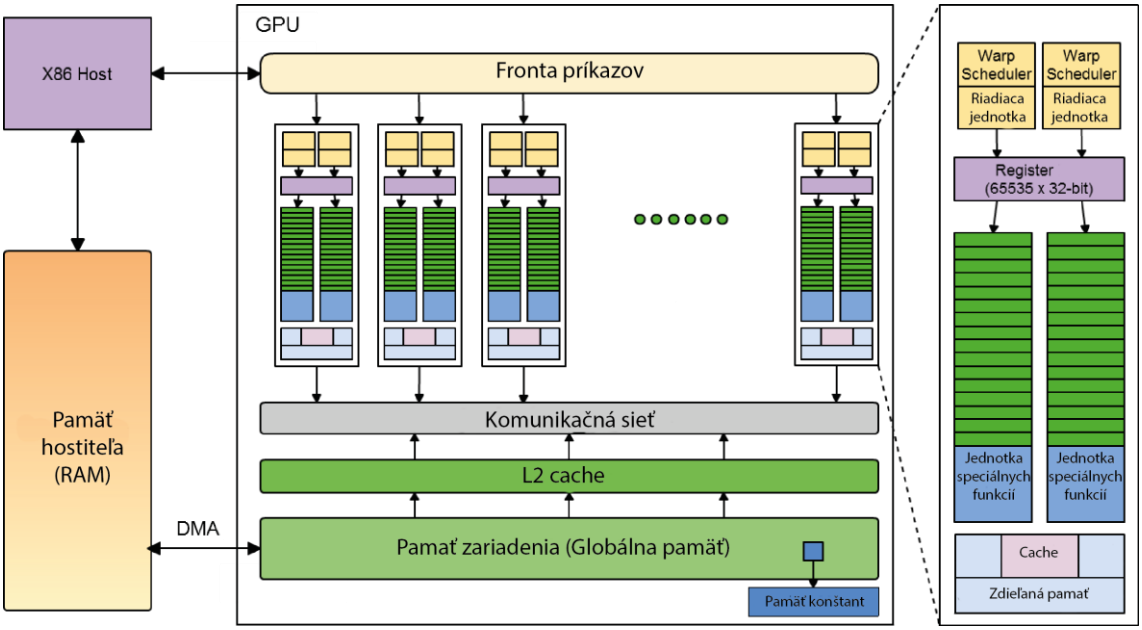
Program pre GPU má nasledujúcu podobu:

```
__global__void increment_gpu(float *a, float b, int N)
{
     int idx = blockIdx.x*blockDim.x + threadIdx.x;
     if (idx < N)
          a[idx] = a [idx] + b;
}
void main()
{
…
dim3 dimBlock (blocksize);
dim3 dimGrid (ceil(N/(float)blocksize) );
increment_gpu<<<dimGrid,dimBlock>>>(a,b,N);
}
```

Rozdiely v programovacom jazyku sú nutné z dôvodu implementácie kódu na viacvláknové procesory. CUDA podporuje kompatibilitu s rôznymi univerzálnymi programovacími jazykmi ako Fortran, OpenCL, C alebo DirectX compute. Na to, aby sme hlbšie pochopili výhody CUDA platformy si priblížime fungovanie GPU. [5]

3.2 ARCHITEKTÚRA GPU

Štandardné CPU má niekoľko výpočtových jadier a rozmanité programovacie možnosti. GPU ma tieto možnosti obmedzené, ale vďaka veľkému počtu aritmeticko-logických jednotiek sa používa k paralelnému spracovaniu dát. Tu dosahuje vyšších výkonov než CPU. GPU je rozdelené na časti – multiprocesory. Každý multiprocesor má 8 procesorov. Procesory sú 32 bitové architektúry SIMT (Single Instruction Multi-Thread), ktorá bola založená na SIMD architektúre. Pri každom hodinovom cykle vykonáva multiprocesor rovnakú operáciu len s rôznymi dátami. Pamäť GPU zahŕňa registre, zdieľanú pamäť, pamäť pre konštanty, textúry a globálnu pamäť dostupnú pre celé GPU a slúžiacu na prenos dát na RAM a naopak.



Obrázok 6: CUDA z pohľadu GPU. [7]

3.3 EXEKUČNÝ MODEL

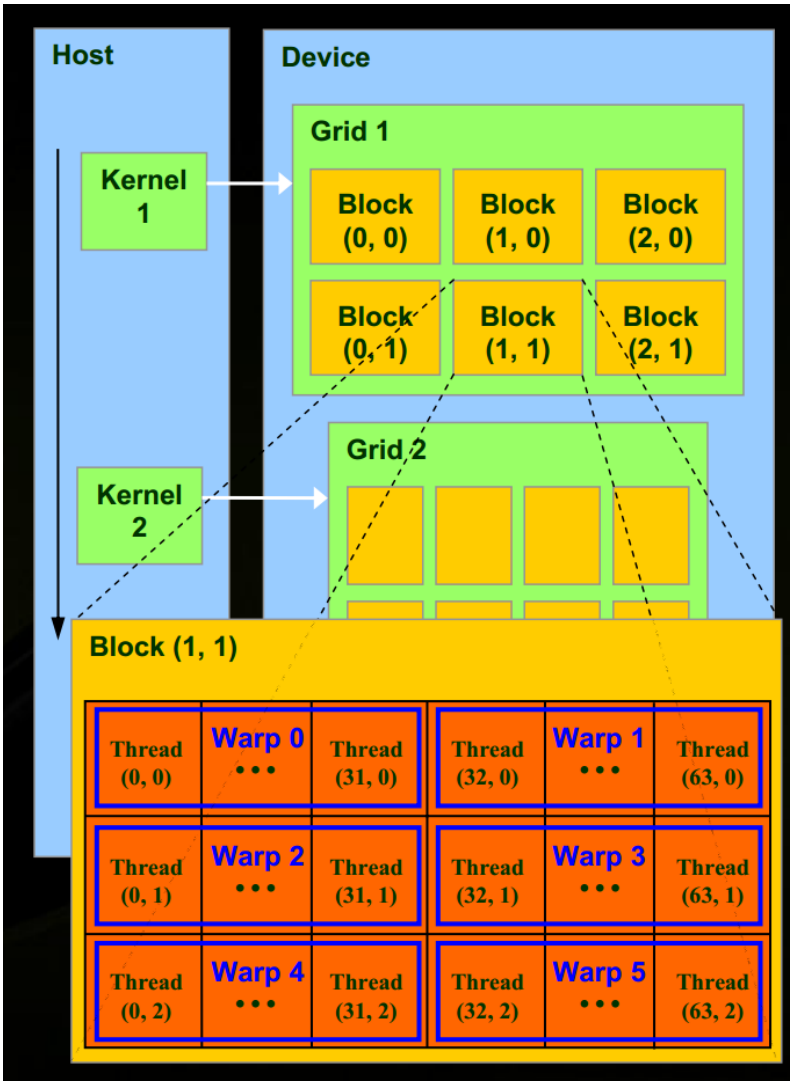
Vlákna v GPU sú organizované do blokov. Samotné bloky sú umiestnené do mriežky. Blok je skupina vlákien, ktoré sú vykonávané v jednej vlne výpočtov.

Každé vlákno v bloku, ktoré spracováva inštrukcie jadrá (kernelu) má svoje thread ID. Programátor dokáže pracovať aj s konkrétnymi vláknami. Na identifikáciu mu slúži preddefinovaná premenná threadIdx. ThreadIdx je typu dim3, čo je trojzložkový vektor. V praxi môžeme využiť počet dimenzií ktorý nám vyhovuje, v spracovaní obrazu stačí dvojdimenzionálne pole. ID vlákna (x, y) bude teda pre 2D $y.Dx+x.$, pre trojdimenzionálne pole bude ID vlákna (x, y, z) $x+yDx + zDxDy$. Dx, Dy a Dz je veľkosť dimenzie. Pre pole rozmeru 10x20 je $Dx = 10$, $Dy = 20$ a $Dz = 1$.

Mriežka blokov vláken (grid) je spracovávaná na multiprocesore. Podľa schopností hardwaru a programovania aplikácie sa bloky vykonávajú po jednom alebo viacero naraz. Každý blok sa rozdelí na tzv. warpy, podbloky optimalizované pre hardware. Rozdelenie je vždy rovnaké, každý warp obsahuje po sebe idúce ID vláken. Spracovanie warpu prebieha za pomoci scheduleru, ktorý prepína medzi warpami v bloku tak, aby čo najlepšie využil schopnosti multiprocesoru. V programe je možné doplniť synchronizačný blok, príkaz `syncThreads()` na ktorom sú vlákna zastavené dovtedy, kým sa do tohto bodu nedostanú všetky vlákna v bloku. Je to výhodné najmä vtedy, ak sa po synchronizácií vlákna načítavajú dáta z pamäti. Priebeh prístupu do pamäti je vhodné optimalizovať pre všetky vlákna súčasne, vytvoriť združený prístup

do pamäti. Pri nezosynchronizovanom bloku vláken potom dochádza k časovému oneskoreniu výpočtov.

Jeden blok vláken je spracovávaný vždy jedným multiprocesorom, nie je možný presun na druhý. Je to z dôvodu načítavania pamäti na čip multiprocesoru, ktorý umožňuje rýchly prístup k dátam medzi vláknami. Táto pamäť je zdieľaná pre práve bežiaci blok vláken. Pri prekročení rozsahu pamäti registrov na multiprocesore blok nemôže byť vykonávaný a aplikácia sa nespustí. [8]



Obrázok 7: Model štandardu CUDA [8].

3.4 PAMÄŤOVÝ MODEL

CUDA programovací model predpokladá, že všetky vlákna sa vykonávajú na fyzicky oddelenom zariadení od hostiteľa na ktorom beží aplikácia. Z toho vyplýva, že zariadenie musí udržiavať vlastný pamäťový priestor. CPU má host memory, GPU má

device memory. Medzi oddelenými pamäťovými priestormi je nutné zabezpečiť prenos dát. Problém nastáva keď každé vlákno vyžaduje vlastné dáta. Pri úzkej zbernici sa tvoria rady a program čaká na doručenie dát. Preto je prioritou programátora zabezpečiť dostatočne rýchly prínos dát na spracovanie pre kernel. Na to slúži niekoľko druhov pamätí:

Registre (registers): sú implementované priamo na čipe, ku konkrétnemu registru môže pristupovať len jedno vlákno. Je to najrýchlejšia pamäť v GPU, priepustnosť je približne 8000 GB/s.

Lokálna pamäť (local memory): vyhradená pre konkrétne vlákno. Nie je priamo na čipe pri výpočtovom jadre. Rýchlosť prístupu je porovnateľná s globálnou pamäťou pretože sa nachádza v DRAM GPU.

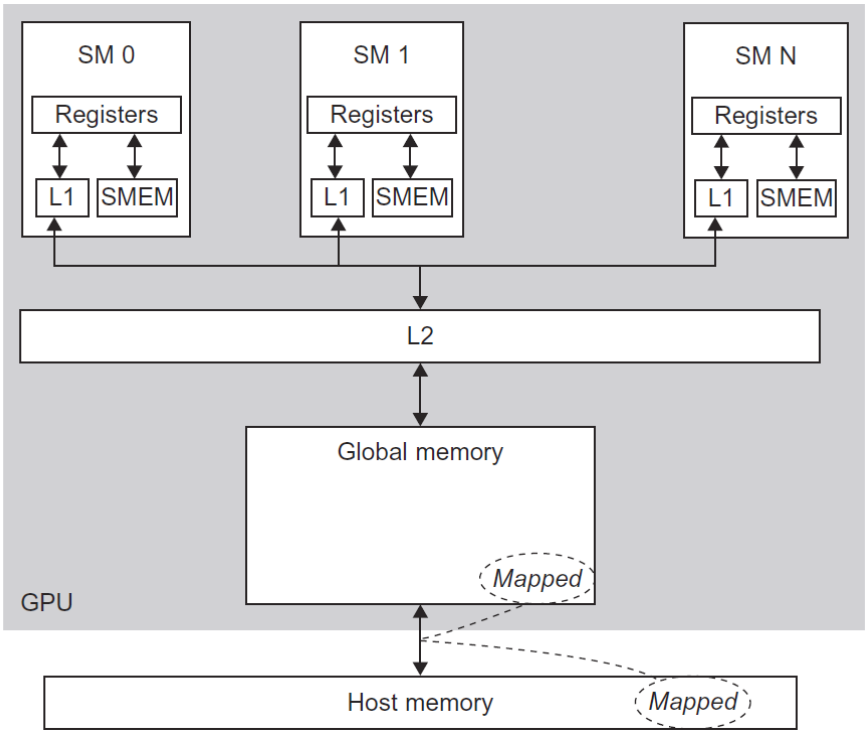
Zdieľaná pamäť (shared memory - SMEM): je priamo na čipe, najčastejšie využívaná pre svoju rýchlosť a poskytuje priestor pre prístup všetkých vlákien v jednom bloku. Nevýhodou je že jej kapacita je obmedzená na 16 – 48 KB, ktoré sú organizované do 32 bitových bankov. Priepustnosť je približne 1600 GB/s.

Globálna pamäť (global memory): nachádza sa v DRAM GPU, dosahuje veľkú kapacity. Slúži ako úložisko dát pre beh programu v GPU. Dáta sú prístupne pre všetky vlákna spustené programom. Priepustnosť je oveľa nižšia, približne 177 GB/s.

Mapovaná pamäť (mapped memory): je to pamäť, ktorá sa nachádza mimo GPU, v RAM počítača. Funkciou je podobná ako globálna pamäť, je však najpomalšia a preto aj málo využívaná. Výhodou je, že nemusíme využívať pamäťový priestor GPU. Rýchlosť je však obmedzená priepustnosťou zbernice.

Pamäť konštánt (constant memory): je to špeciálny typ pamäte, ktorý je prístupný len na čítanie pre vlákna GPU, dáta do tejto pamäte zadáva funkcia v CPU ešte pred spustením CUDA funkcie. Výhodou je spôsob čítania pamäti. V prípade že rovnaké dáta číta viacero vláken program naraz, jediný prístup a čítanie pamäte sa rozošle pre všetky vlákna naraz, čo nám šetrí celkový prenosový čas.

Pamäťový model a prepojenia sú zobrazené na obrázku č. 8. V obrázku sú zakomponované aj pamäti L1 a L2. Tie slúžia ako všeobecná cache. Využívať naplno ich môžu až GPU, ktoré majú výpočtové schopnosti vyššej verzie 2.0. Z tohto dôvodu na starších zariadeniach podporujúcich verziu 1.0 alebo 1.1 bežia aplikácie pomalšie. L1 a L2 cache bez zásahu programátora udržiavajú najčastejšie využívané dáta v rýchlo dostupnom pamäťovom priestore. Významne urýchľujú nepravidelné prístupy do pamäte, čo je pre veľa algoritmov rozhodujúci faktor vo výpočtovom výkone.

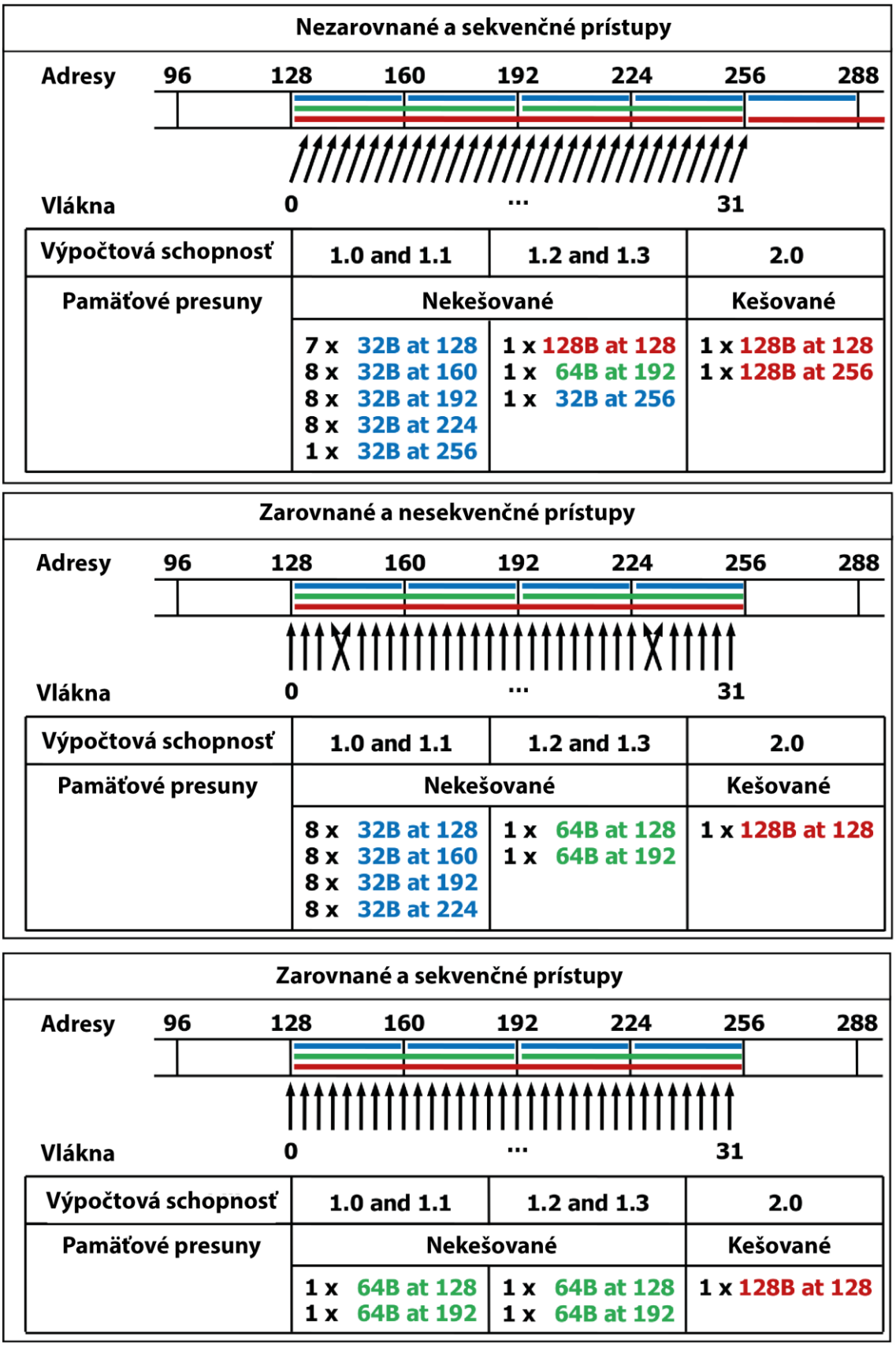


Obrázok 8: Pamäťový model GPU [3]

GPU je silne optimalizované pre sekvenčný prístup do globálnej pamäte. Programátor by sa mal vyhnúť náhodným prístupom do nej. Ideálny postup je načítať dáta do zdieľanej pamäte multiprocesora, potom previesť výpočty a nakoniec zapísať výsledok do globálnej pamäte. Na urýchlenie čítacích operácií nám môže poslúžiť zlúčený prístup do pamäte. Urýchlenie je založené na znížení nutných prístupov. Zarovnaný na rozsah dát a zároveň sekvenčný prístup do pamäte je najvýhodnejší. V jednom kroku GPU dokáže z globálnej pamäte kopírovať až 128 B sekvenčne naraz pri výpočtovej schopnosti 2.0. Podľa veľkosti používaného dátového typu dochádza k úspore. Napríklad dátový typ char má 8 B, na jeden prístup do pamäte nakopírujeme 16 premenných typu char pre 16 vláken. Podmienky pre využitie zlúčeného prístupu do pamäte sú:

1. Štandardizovaná veľkosť pamäťového elementu ktoré prenášame, 4, 8 alebo 16 Bytov.
2. Vlákna musia pristupovať k pamäťovým elementom sekvenčne. Vlákno idx pristupuje k bloku idx, vlákno idx + 1 pristupuje k bloku pamäte idx + 1.
3. Všetky elementy ku ktorým pristupujeme, musia byť v rovnakom bloku pamäte a adresa prvého prvku musí byť zarovnaná násobku veľkosti elementu ktoré sme schopný naraz kopírovať.

Nižšie sú zobrazené ukážky prístupov do pamäte. Pri nedodržaní podmienok dochádza k nárastu počtov prístupov do pamäte a spomaleniu rýchlosti výpočtov. [4]



Obrázok 9: Rôzne prístupy do pamäte. [4]

4 APLIKÁCIA NA SPRACOVANIE OBRAZU

V tejto časti je popísaná aplikácia na ktorej demonštrujeme výpočtové schopnosti grafického hardwaru a porovnávame výkon so spracovaním obrazu na CPU. Program je napísaný vo v programovacom jazyku C/C++ s frameworkom Microsoft .NET verzia 4 a pre výpočty na GPU využíva nadstavbu Nvidia CUDA verzia 4.1.

Na spustenie programu je potrebné mať k dispozícii počítač s grafickou kartou od výrobcu Nvidia. Grafická karta musí podporovať výpočty CUDA. Minimálne požiadavky na GPU je jadro verzie G80 a vyššie (verzie Nvidia GeForce 8000). Tiež sú potrebné softwarové predpoklady ako operačný systém Windows XP a vyšší, Microsoft .NET framework 4, Microsoft Visual C++ 2010 Redistributable Package 32 bit, najnovší driver pre grafickú kartu Nvidia.

Microsoft .NET framework 4 je knihovňa, ktorá zabezpečuje grafickú nadstavbu programu. Microsoft Visual C++ 2010 Redistributable Package 32 bit, obsahuje komponenty z Microsoft Visual C++ knihovní potrebné na chod aplikácie vyvinutej v rozhraní Microsoft Visual Studio 2010 na počítači, ktorý nemá Microsoft Visual C++ 2010 nainštalovaný. 32 bitovú verziu používame kvôli kompatibilite s 32 bitovými verziami operačných systémov a z dôvodu, že Microsoft Visual Studio 2010 nemá vlastnú 64 bitovú verziu a kompilácia programu s rozšírením CUDA 64 bit zvyšuje náročnosť. Rovnako možnosti 64 bit adresácie pamäťového priestoru program nevyužije. Najnovší driver pre grafickú kartu Nvidia je potrebný pre bezproblémový chod CUDA kódu programu, nakoľko staršie verzie driverov pre prvé podporované grafické karty ešte neobsahujú podporu pre CUDA výpočty.

Na editáciu zdrojového kódu programu je potrebné mať k dispozícii rovnaké prostiedky ako na spustenie aplikácie naviac je potrebné mať nainštalované Microsoft Visual Studio 2010, najnovší driver pre grafickú kartu Nvidia, CUDA toolkit 4.1 32 bit. Doporučené je doinštalovať rozšírenie Nvidia Nsight monitor 2.2 32 bit z dôvodu rozšírenia aplikácie Visual Studio 2010 o pokročilé debugovanie, monitorovanie aplikácie a pamäťového priestoru na grafickej karte. Keďže Visual Studio 2010 neobsahuje natívnu podporu pre CUDA, nedokáže graficky rozlíšiť kód programu tak, aby bol užívateľom prehľadne čitateľný.

CUDA toolkit 4.1 32 bit obsahuje kompilátory a funkcie potrebné pre použitie CUDA v kombinácii Microsoft Visual Studio 2010. Definuje aj vlastné prípony pre zdrojové kódy CUDA .cu pre vlastné funkcie a .cuh pre hlavičky vlastných funkcií. Nvidia Nsight monitor 2.2 32 bit dopĺňa do Microsoft Visual Studio 2010 možnosti pre analýzu kódu CUDA a sprehľadňuje kód CUDA grafickou nápoveďou. Analýza kódu zahŕňa zber počítadiel, štatistík a zber hodnôt použitých v CUDA kernel volaniach. Výhodou je

prehľadná časová os v ktorej je znázornená doba trvania určitej procedúry alebo funkcie spracovávanej grafickou kartou. Pri použití CUDA kernelov môže nastať situácia keď program presiahne časový limit, obvykle 2 sekundy, po ktorom sa výpočet na grafickom zariadení zastaví a systém považuje túto udalosť za chybu. Túto vlastnosť spôsobuje watchdog v systéme Windows ktorý sleduje programy využívajúce grafické drivery na určité výpočty a v prípade že program využíva prostriedky zobrazovacieho zariadenia dlhšie ako je povolená doba, watchdog funkciu zastaví a resetuje grafický driver. Funkciu watchdog je možné vypnúť v registroch systému Windows s nastavením

Cesta:
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\GraphicsDrivers
Názov: TdrLevel
Typ hodnoty: REG_DWORD
Hodnota: TdrLevelOff (0) - vypnutie funkcie

4.1 POPIS ROZHRANIA GUI

Grafické užívateľské rozhranie aplikácie pre spracovanie obrazu, je napísané s rozšírením cez framework Microcoft .NET 4 pre zjednodušený návrh. Program pomocou konvolučných masiek aplikuje na vstupný obraz rôzne filtre, ktoré môžu byť preddefinované, alebo zadávané užívateľom. Najskôr je potrebné načítať vstupný obraz. Program podporuje len formát súboru .bmp bitmapu, nezáleží na farebnej hĺbke, program obraz konvertuje na šedotónový. Obraz načítame kliknutím na tlačidlo Load, po ktorom sa zobrazí štandardné dialógové okno pre výber súboru s obrazom. Výber potvrdíme kliknutím na tlačidlo OK. Potom nastavíme parametre programu. Na výber máme spustenie kódu cez funkciu CPU alebo GPU. Po označení je potrebné zvoliť variant obrazového filtra. Štandardne je predvolený užívateľsky zadávaný obrazový filter.

Užívateľ má na výber veľkosti konvolučných masiek: 3x3, 5x5, 7x7 a 9x9. Užívateľsky zadávané filtre pri výbere typu spracovania cez grafickú kartu majú k dispozícii tri typy využitia možnosti GPU. Možnosti 3x3, 5x5, 7x7 a 9x9 využívajú pre spracovanie len globálnu pamäť GPU na všetky operácie. Možnosti 3x3 const mem, 5x5 const mem, 7x7 const mem a 9x9 const mem využívajú pre spracovanie globálnu pamäť GPU pre vstupný obraz, a pamäť konštánt pre konvolučné jadro. Možnosti 3x3 optimized, 5x5 optimized, 7x7 optimized a 9x9 optimized využívajú okrem globálnej pamäte pre načítanie vstupu a pamäte konštánt pre konvolučné jadro aj pamäť zdieľanú, ktorá výrazne urýchľuje načítavanie a spracovanie obrazových dát v definovaných blokoch kernelov. Pre rôznorodosť možných užívateľsky definovaných filtrov je pridaná možnosť „Divide by kernel size“ na vydelenie nesymetrických konvolučných masiek. Napríklad pre použitie obyčajného priemerovacieho filtra zadáme do vstupnej tabuľky

pre masku samé jednotky a označíme voľbu „Divide by kernel size“, aby sa spracovaný bod konvolučnou maskou vydelil veľkosťou masky. Naopak pri použití Laplacovho detektora hrán, ktorý je symetrický, nie je nutné voľbu zatrhávať.

K dispozícii sú aj niektoré preddefinované obrazové filtre, najmä tie ktoré nie je možné zadávať ako jednu konvolučnú masku alebo majú iný spôsob spracovania vstupných dát. Dostupný je mediánový filter vo všetkých veľkostiach, Sobelov operátor veľkosti 3x3, a Laplacov hranový detektor veľkosti 3x3 a 5x5. Pre použitie týchto preddefinovaných filtrov je nutné označiť možnosť „Use common filter“.

Po nastavení parametrov je možné spustiť spracovanie kliknutím na tlačidlo „Run“. Počas chodu spracovania program nereaguje na vstupné požiadavky od užívateľa. Po spracovaní za na pravej strane programového okna zobrazí ukážka výstupného obrazu, ktorý si môžeme porovnať s obrazom vstupným, ktorý je v ľavej časti programového okna, pre lepšie zistenie rozdielov. Po spracovaní je možnosť uložiť výstupný obraz na ľubovoľné miesto v počítači kliknutím na tlačidlo „Save“.

Ako doplnkové možnosti sú k dispozícii nástroje na sledovanie spracovania a detekciu chýb. Označením možnosti „Debug mode“ pred spustením spracovania obrazu sa zobrazí konzolové okno, do ktorého program vypisuje doby trvania určitých procedúr, prípadne pri vzniku chyby, najmä pri spracovaní cez GPU. K chybám dochádza hlavne pri nedostatku globálnej pamäte GPU, ktorú obvykle využívajú programy podporujúce urýchľovanie zobrazovania cez GPU.

Pre zistenie parametrov grafickej karty podporujúcej CUDA môžeme použiť funkciu „GPU info“, ktorá zavolá externý program na výpis potrebných informácií pre zistenie výpočtových možností CUDA na danej grafickej karte. Pri nenájdení grafickej karty, ktorá podporuje možnosti CUDA program vypíše, že nenašiel podporované zariadenie. Rovnako nebude možné ani spustiť proces spracovania obrazu cez prostriedky GPU.

Obrázok 10: Grafické rozhranie programu

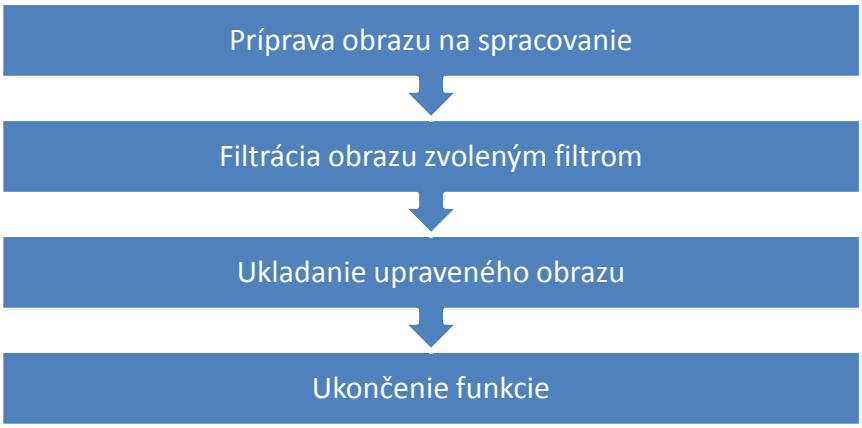
5 IMPLEMENTÁCIA VYBRANÝCH METÓD

Obsahom tejto kapitoly je popis kódu programu a rozbor použitých metód. Ďalej implementácia metód na spracovanie obrazu cez CPU a spracovanie obrazu cez GPU a jeho optimalizácia na využitie dostupných prostriedkov. Tiež porovnanie efektivity spracovania obrazu pre rôzne masky a veľkosti vstupného obrazu a výber optimálnej metódy. Ako aj porovnanie efektivity programu pri rozdielnych typoch grafického hardwaru.

5.1 IMPLEMENTÁCIA METÓD PRE CPU

Metódy pre CPU sú vytvorené tak, aby vykonávali spracovanie obrazu sériovo, tak aby bol zvýraznený rozdiel medzi paralelným spracovaním na GPU. Využitie viacerých vláken CPU by viedlo k určitému paralelizmu, ktorý nechceme porovnávať s GPU.

Priebeh programu v CPU:



Dostupné filtre sú členené do štyroch kategórií:

- Užívateľsky explicitne zadávané hodnoty filtra pre spracovanie – veľkosti 3x3, 5x5, 7x7 a 9x9
- Mediánový filter – veľkosti 3x3, 5x5, 7x7 a 9x9
- Sobelov operátor
- Laplaceov hranový detektor – veľkosti 3x3 a 5x5

Užívateľsky explicitne zadávané hodnoty filtra pre spracovanie

Príprava obrazu na spracovanie:

Po načítaní vstupného obrazu cez formulár, vyplnení tabuľky pre konvolučnú masku a zavolaní spracovania sa odošle príkaz pre vykonanie funkcie. Funkcia obsahuje vlastné parametre ako cesta k vstupnému súboru, konvolučné jadro v poli prvkov a deliteľ pre konvolučné jadro. Vo funkcii sa načíta obraz do pamäte cez knihovňu EasyBMP ktorá definuje dátový typ BMP a umožňuje jednoduchú prácu s dátami v obraze.

```
BMP Input;
Input.ReadFromFile(cesta_k_súboru_cez_parameter_funkcie);
```

Pretože program pracuje so šedotónovymi obrazmi, na ošetrenie chýb pri načítaní farebnej bitmapy je zavolaná konverzia na šedotónový obraz v pomere 0,299 pre červený kanál 0,587 pre zelený kanál a 0,114 pre modrý kanál obrazu.

```
// convert each pixel to grayscale using RGB->YUV
for( int j=0 ; j < Input.TellHeight() ; j++)
{
     for( int i=0 ; i < Input.TellWidth() ; i++)

          {
               int Temp = (int) floor(0.299*Input(i,j)->Red
+ 0.587*Input(i,j)->Green + 0.114*Input(i,j)->Blue );
               BYTE TempBYTE = (BYTE) Temp;
               Input(i,j)->Red = TempBYTE;
               Input(i,j)->Green = TempBYTE;
               Input(i,j)->Blue = TempBYTE;
          }
     }
```

Nakoniec sa zadefinuje globálne farebná hĺbka obrazu na 8 bitov. Tým je príprava obrazu dokončená.

```
// Create a grayscale color table if necessary
if( Input.TellBitDepth() < 16 )
     { CreateGrayscaleColorTable( Input );}
```

Filtrácia obrazu zvoleným filtrom:

Metóda je navrhnutá ako vnorený cyklus, ktorý pre každý bod obrazu postupne vypočíta novú filtrovanú hodnotu. Pre každý pixel v závislosti na veľkosti masky je nutné poskytnúť pre výpočet okolie bodu. Čím väčšia je veľkosť masky, tým časovo náročnejšie je spracovanie jediného obrazového bodu. Cyklus je ošetrený o okrajové body, ktoré nemôže spracovať, pretože by dochádzalo k čítaniu pamäte mimo povolený rozsah.

Ukážka spracovania veľkosťou masky 3x3:

```
int sum=0;
  for( int j=1 ; j < Input.TellHeight() ; j++)
  {
       for( int i=1 ; i < Input.TellWidth() ; i++)
       {
          if( (j > 1) && (j < (Input.TellHeight()-1)) &&
          (i > 1) && (i < (Input.TellWidth()-1)) )
            {
                 sum=0;
                 for(int l= -1; l < 2; l++)
                 {
                      for(int k= -1; k < 2; k++)
                      {
     sum += konvol_jadro[k+1][l+1] * input(i+l,j+k)->Red;
                      }
                 }
            }
          sum=sum/divid; //delenie veľkosťou masky
          if (sum > 255) { sum=255;} //kontrola presahu
          Input(i,j)->Green =abs(sum);
       }
  }
```

Veľkosti 5x5, 7x7 a 9x9 sú vytvárane identickým postupom s rozdielnymi veľkosťami masky a dĺžkou cyklov.

Ukladanie upraveného obrazu:

Keďže spracovanie prebieha len pre jeden farebný kanál, je nutné skopírovať výstupne dáta na všetky kanály. Následne sa obraz uloží do dočasného súboru out.bmp v adresári, kde je uložený program so všetkými potrebnými náležitosťami pre formát súboru bitmapy. Uložený obraz sa po ukončení funkcie načíta do grafického rozhrania a zobrazí sa ukážka výstupného obrazu.

Ukončenie funkcie:

Na záver spracovania sa uvoľnia použité pamäťové prostriedky a program čaká na ďalší vstup od užívateľa.

Mediánový filter

Príprava obrazu na spracovanie:

Po načítaní vstupného obrazu cez formulár a zavolaní spracovania sa odošle príkaz pre vykonanie funkcie. Funkcia obsahuje vlastný parameter cestu k vstupnému súboru. Vo funkcii sa načíta obraz do pamäte cez knihovňu EasyBMP ktorá definuje dátový typ BMP a umožňuje jednoduchú prácu s dátami v obraze.

```
BMP Input;
Input.ReadFromFile(cesta_k_súboru_cez_parameter_funkcie);
```

Pretože program pracuje so šedotónovymi obrazmi, na ošetrenie chýb pri načítaní farebnej bitmapy je zavolaná konverzia na šedotónový obraz v pomere 0,299 pre červený kanál 0,587 pre zelený kanál a 0,114 pre modrý kanál obrazu. Konverzia obrazu je identická so spracovaním pre explicitne zadávané masky.

Filtrácia obrazu zvoleným filtrom:

Metóda je navrhnutá ako vnorený cyklus, ktorý pre každý bod obrazu postupne vypočíta novú filtrovanú hodnotu. Pre každý pixel v závislosti na veľkosti masky je nutné poskytnúť pre výpočet okolie bodu. Čím väčšia je veľkosť masky, tým časovo náročnejšie je spracovanie jediného obrazového bodu. Cyklus je ošetrený o okrajové body, ktoré nemôže spracovať, pretože by dochádzalo k čítaniu pamäte mimo povolený rozsah. Princíp mediánového filtra je založený na čiastkovom utriedení vstupných prvkov a výbere strednej hodnoty.

Ukážka spracovania mediánovým filtrom veľkosťou masky 3x3:

```
for (int mm = 1; mm<Input.TellHeight()-1; ++mm)
{
   for (int nn = 1; nn < Input.TellWidth() - 1; ++nn)
   {
int k = 0;
      int window[9];
      for (int j = mm - 1; j < mm + 2; ++j)
         for (int i = nn - 1; i < nn + 2; ++i)
            window[k++] = Input(i,j)->Red;
       //kopírovanie dat do pomocnej premeenej
       int n=9;
```

```
 int L = 0;
 int Q = n-1;
 k = n / 2;
 int o;int p;
 while (L < Q)
 {
  float x = window[k];
  o = L; p = Q;
do
 {
  while (window[o] < x)o++;
  while (x < window[p])p--;
  if (o <= p)
  {
    float t = window[o];
    window[o] = window[p];
    window[p] = t;
    o++;p--;
  }
 } while (o <= p);
  if (p < k)  L = o;
  if (k < o)  Q = p;
 }//triedenie poľa a hľadanie medianu
         //   Výsledok – prostredný element
         Input(nn,mm)->Green = window[k];
      }   }
```

Veľkosti 5x5, 7x7 a 9x9 sú vytvárane identickým postupom s rozdielnymi veľkosťami masky a dĺžkou cyklov.

Ukladanie upraveného obrazu:

Keďže spracovanie prebieha len pre jeden farebný kanál, je nutné skopírovať výstupne dáta na všetky kanály. Následne sa obraz uloží do dočasného súboru out.bmp v adresári, kde je uložený program so všetkými potrebnými náležitosťami pre formát súboru bitmapy. Uložený obraz sa po ukončení funkcie načíta do grafického rozhrania a zobrazí sa ukážka výstupného obrazu.

Ukončenie funkcie:

Na záver spracovania sa uvoľnia použité pamäťové prostriedky a program čaká na ďalší vstup od užívateľa.

Sobelov operátor

Príprava obrazu na spracovanie:

Príprava obrazu pre spracovanie Sobelovým operátorom je identická s mediánovým filtrovaním. Po načítaní vstupného obrazu cez formulár, vyplnení tabuľky pre konvolučnú masku a zavolaní spracovania sa odošle príkaz pre vykonanie funkcie. Funkcia obsahuje vlastný parameter, cestu k vstupnému súboru. Vo funkcii sa načíta obraz do pamäte cez knihovňu EasyBMP. Rovnako je zavolaná konverzia na šedotónový obraz v pomere 0,299 pre červený kanál 0,587 pre zelený kanál a 0,114 pre modrý kanál obrazu. Nakoniec sa zadefinuje farebná hĺbka obrazu na 8 bitov. Tým je príprava obrazu dokončená.

Filtrácia obrazu zvoleným filtrom:

Metóda je navrhnutá ako vnorený cyklus, ktorý pre každý bod obrazu postupne vypočíta novú filtrovanú hodnotu. Pre každý pixel v závislosti na veľkosti masky je nutné poskytnúť pre výpočet okolie bodu. Cyklus je ošetrený o okrajové body, ktoré nemôže spracovať, pretože by dochádzalo k čítaniu pamäte mimo povolený rozsah. Rozdiel oproti užívateľom definovanej masky je v použití dvojitej masky. Cyklus vypočítava dve hodnoty pre každú masku zvlášť, následne ich absolútne hodnoty sčíta a uloží do výsledku. Veľkosť masky je len 3x3 pretože táto veľkosť je dostačujúca na zvýraznenie hrán pomocou tohto operátora.

Ukladanie upraveného obrazu:

Ukladanie upraveného obrazu prebieha rovnako ako u vyššie spomenutých obrazových filtrov.

Ukončenie funkcie:

Na záver spracovania sa uvoľnia použité pamäťové prostriedky a program čaká na ďalší vstup od užívateľa.

Laplaceov hranový operátor

Príprava obrazu na spracovanie:

Príprava obrazu pre spracovanie Laplaceovým hranovým operátorom je identická s mediánovým aj sobelovým filtrovaním. Po načítaní vstupného obrazu cez formulár, vyplnení tabuľky pre konvolučnú masku a zavolaní spracovania sa odošle príkaz pre vykonanie funkcie. Funkcia obsahuje vlastný parameter, cestu k vstupnému súboru. Vo

funkcii sa načíta obraz do pamäte cez knihovňu EasyBMP. Rovnako je zavolaná konverzia na šedotónový obraz a definícia farebnej hĺbky 8 bitov.

Filtrácia obrazu zvoleným filtrom:

Taktiež ako u predchádzajúcich typoch je metóda navrhnutá ako vnorený cyklus, ktorý pre každý bod obrazu postupne vypočíta novú filtrovanú hodnotu. Pre každý pixel v závislosti na veľkosti masky je nutné poskytnúť pre výpočet okolie bodu. Cyklus je ošetrený o okrajové body, ktoré nemôže spracovať, pretože by dochádzalo k čítaniu pamäte mimo povolený rozsah. Užívateľ nezadáva masku, tá je už preddefinovaná. Výpočet prebieha rovnako ako u užívateľmi zadávanej masky. Veľkosť masky je dostupná vo veľkostiach masky 3x3 a 5x5.

Ukladanie upraveného obrazu:

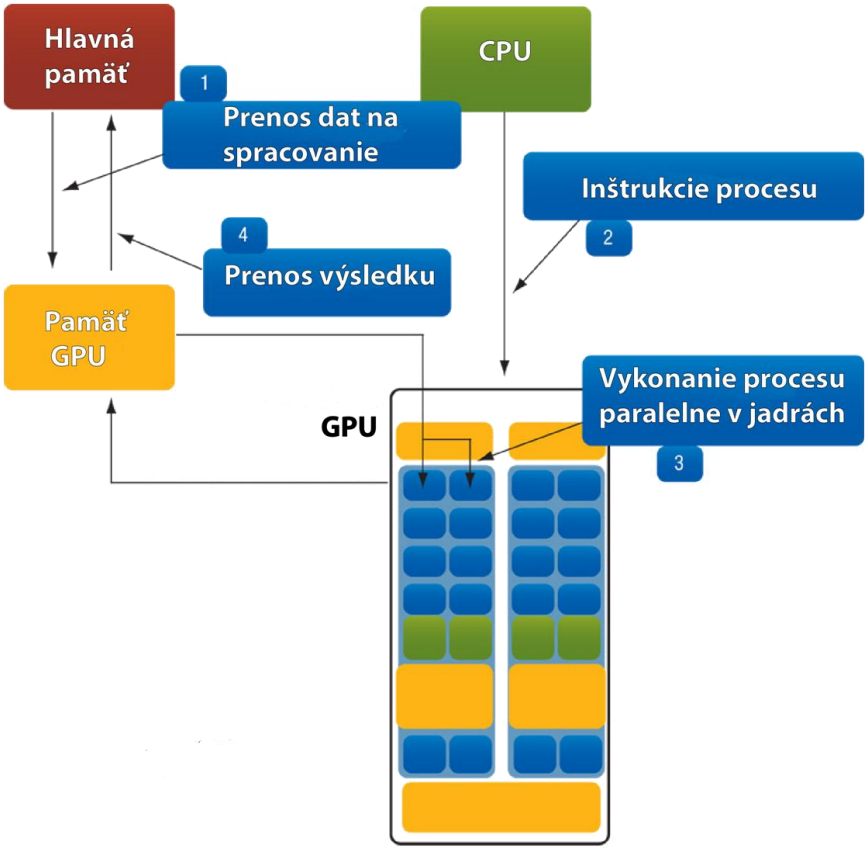
Ukladanie upraveného obrazu prebieha rovnako ako u vyššie spomenutých obrazových filtrov.

Ukončenie funkcie:

Na záver spracovania sa uvoľnia použité pamäťové prostriedky a program čaká na ďalší vstup od užívateľa.

5.2 IMPLEMENTÁCIA METÓD PRE GPU

Metódy pre GPU sú vytvorené tak, aby vykonávali spracovanie obrazu paralelným spôsobom, tak aby využitie dostupných prostriedkov grafickej karty bolo čo najefektívnejšie.
Základná časť ako príprava obrazu a následné dodatočné spracovanie po aplikácii filtračnej funkcie prebieha na CPU a samotná filtračná funkcia prebieha na GPU.



Obrázok 11: Priebeh spracovania dát v GPU

Na obrázku č. 11 je schematicky znázornený priebeh procesu spracovania obrazu na GPU. Pred samotným začatím procesu je nutné vykonať prípravu obrazu na spracovanie.

Príprava obrazu na spracovanie rovnaká pre všetky metódy spracovania na GPU:

Po načítaní vstupného obrazu cez formulár, vyplnení tabuľky pre konvolučnú masku a zavolaní spracovania sa odošle príkaz pre vykonanie funkcie. Funkcia obsahuje vlastné parametre ako cesta k vstupnému súboru, konvolučné jadro v poli prvkov a deliteľ pre konvolučné jadro. Vo funkcii sa načíta obraz do pamäte cez knihovňu EasyBMP ktorá definuje dátový typ BMP a umožňuje jednoduchú prácu s dátami v obraze.

```
BMP Input;
Input.ReadFromFile(cesta_k_suboru_cez_parameter_funkcie);
```

Pretože program pracuje so šedotónovymi obrazmi, na ošetrenie chýb pri načítaní farebnej bitmapy je zavolaná konverzia na šedotónový obraz v pomere 0,299 pre červený kanál 0,587 pre zelený kanál a 0,114 pre modrý kanál obrazu.

```
// convert each pixel to grayscale using RGB->YUV
for( int j=0 ; j < Input.TellHeight() ; j++)
{
     for( int i=0 ; i < Input.TellWidth() ; i++)

          {
               int Temp = (int) floor(0.299*Input(i,j)->Red
+ 0.587*Input(i,j)->Green + 0.114*Input(i,j)->Blue );
               BYTE TempBYTE = (BYTE) Temp;
               Input(i,j)->Red = TempBYTE;
               Input(i,j)->Green = TempBYTE;
               Input(i,j)->Blue = TempBYTE;
          }
     }
```

Potom sa zadefinuje globálne farebná hĺbka obrazu na 8 bitov.

```
// Create a grayscale color table if necessary
if( Input.TellBitDepth() < 16 )
     { CreateGrayscaleColorTable( Input );}
```

Keďže funkcia CUDA spracováva obraz ako jednorozmerné pole, je potrebné vstupný obraz dátového typu bitmapa previesť na jednoduché jednorozmerné pole prvkov. Funkcia zadeklaruje jednorozmerné pole obraz a alokuje priestor o veľkosti rovnakej ako je veľkosť vstupného obrazu. Do poľa sa potom jednoduchým cyklom prekopírujú dáta z vstupného obrazu typu bitmapa. Na spracovanie obrazu nám stačí jeden farebný kanál pretože pri šedotónovom obraze sú všetky tri farebné kanály identické.

```
Unsigned char *obraz;
obraz = (unsigned char
*)malloc(Input.TellHeight()*Input.TellWidth()*
sizeof(unsigned char));
for (int x = 0; x < Input.TellWidth(); x++)
        {
     for (int y = 0; y < Input.TellHeight(); y++)
          {
          obraz[y*Input.TellWidth()+x]=Input(x,y)->Blue;
           }
        }
```

Dostupné filtre pre spracovanie obrazu na GPU sú členené nasledovne:

- Užívateľsky explicitne zadávané hodnoty filtra pre spracovanie – veľkosti 3x3, 5x5, 7x7 a 9x9
    a) bez optimalizácie rýchlosti spracovania,
    b) s optimalizáciou využitia pamäte pre konštanty,
    c) s optimalizáciou využitia pamäte pre konštanty a zdieľanej pamäte.
- Mediánový filter - veľkosti 3x3, 5x5, 7x7 a 9x9 plne optimalizované s využitím zdieľanej pamäte.
- Sobelov operátor – je optimalizovaný pomocou využitia pamäte pre konštanty a zdieľanej pamäte.
- Laplaceov hranový detektor – veľkosti 3x3 a 5x5 – je optimalizovaný pomocou využitia pamäte pre konštanty a zdieľanej pamäte.

Na základe schematického znázornenia postupu spracovania obrazu v jednotlivých krokoch postupujeme nasledovne.

Užívateľsky explicitne zadávané hodnoty filtra pre spracovanie

Prenos dát na spracovanie

Najskôr je potrebná alokácia jednorozmerného poľa pre GPU v rozmeroch rovnakých ako je veľkosť dodaného poľa v ktorom je pôvodný obraz. Na alokáciu a odovzdanie ukazateľa sa používa ekvivalent kódu C – malloc, na CUDA je to cudaMalloc. Alokácia prebieha v globálnej pamäti GPU. CudaMalloc má dva vstupné parametre: ukazateľ na alokovanú pamäť a veľkosť pre alokáciu. Výstupný parameter je enumerátor cudaError_T.

```
unsigned char *dev_a,*dev_c;//deklarace
cudaMalloc((void**) &dev_a,Height*Width*sizeof(unsigned
char)); //vstup do GPU, Height a Width – rozmery obrazu
cudaMalloc((void**) &dev_c,Height*Width*sizeof(unsigned
char)); //výstup z GPU
```

Pokračujeme kopírovaním dát zo vstupného poľa do pamäti na GPU. Využíva sa príkaz cudaMemcpy. Dáta kopíruje do globálnej pamäti. Parametre cudaMemcpy:

1. Parameter: ukazateľ cieľovej pamäti,
2. Parameter: ukazateľ zdrojovej pamäti,
3. Parameter: veľkosť kopírovanej pamäti v bitoch,
4. Parameter: enumerátor definujúci z akého druhu pamäti do akého bude kopírované.

```
cudaMemcpy(dev_a,input,Height*Width*sizeof(char),cudaMemcpy
HostToDevice)//kopírovanie z input obrazu do pamäti GPU
```

Rovnakým spôsobom sa do pamäti GPU nakopíruje aj konvolučná maska ak je potrebná. Napr. Mediánový filter ju nevyužije, a naopak je potrebná pre užívateľom zadávané masky.

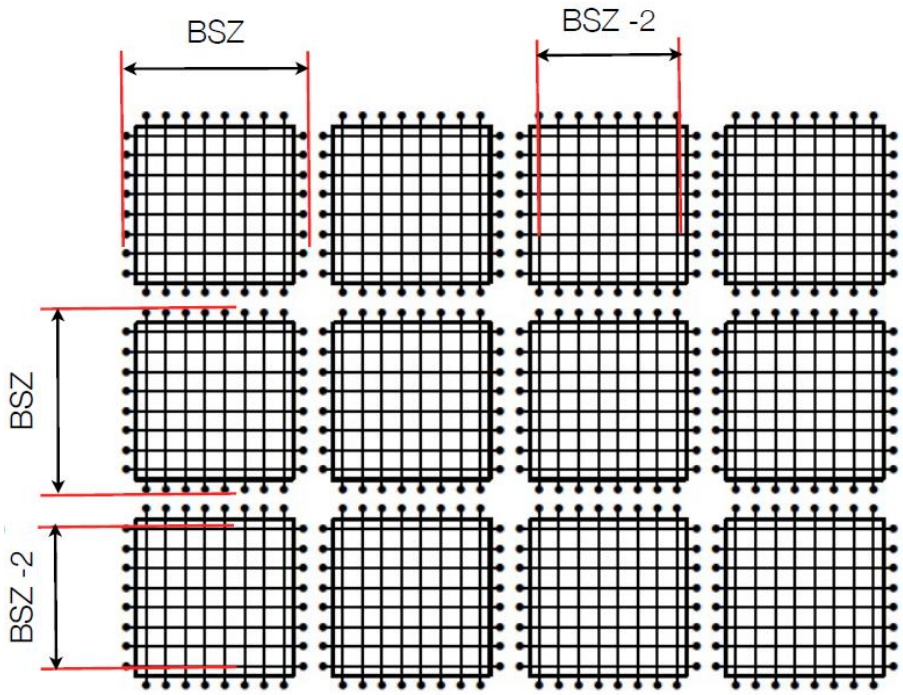
Inštrukcie procesu GPU

V druhom kroku definujeme rozmer poľa jadier vykonávaných v jednom bloku a rozmer poľa samotných blokov. Tým zabezpečíme vykonanie paralelnej operácie po celej ploche obrazu. Je nutné obraz rozdeliť na menšie bloky, aby sa optimálne využili možnosti hardwaru podľa exekučného modelu GPU. Polia sú trojdimenzionálne, ale my využívame len x a y dimenzie.

```
Dim3 threadsPerBlock(BLOCK_W,BLOCK_H);//počet vláken
v bloku
dim3 grid(Width/TILE_W,Height/TILE_H);//mriežka blokov
```

Rozmery musia byť definované ako konštanty, nie je možné ich definovať ako premenné hodnoty. Ide o konštanty typu veľkosti konvolučnej masky, veľkosti mriežky blokov, mriežky vláken.

```
#define TILE_W 16 //šírka mriežky blokov
#define TILE_H 16 // výška mriežky blokov
#define R 1 // polomer konvolučného filtra
#define D (R*2+1) // priemer konvolučného filtra
#define S (D*D)  // veľkosť konvolučného filtra
#define BLOCK_W (TILE_W+(2*R)) //šírka mriežky vláken
#define BLOCK_H (TILE_H+(2*R)) //výška mriežky vláken
```



Obrázok 12: Delenie obrazu na menšie bloky s prekrývaním

Pri použití konvolučného filtra s maskou musíme počítať s nutnosťou poskytnúť na spracovanie všetky body potrebné pod konvolučnú masku. Pri rozdelení obrazu na bloky ktoré sa spracúvajú postupne musíme o tieto body rozšíriť mriežku pre vlákna v nastavení parametrov CUDA funkcie. Napríklad pri použití konvolučného filtra ktorý má masku o veľkosti 3x3 definujeme veľkosť mriežky blokov o veľkosti 16x16, a veľkosť mriežky vláken o veľkosti 18x18 viď obrázok 12. Týmto nastavením dosiahneme bezproblémový prístup k okrajovým bodom zasahujúcim mimo počítanú časť obrazu. Pri tejto metóde nie je využitých 100 % dostupných prostriedkov GPU, pretože časť vláken sa pri spracovaní obrazu použije len na načítanie potrebných dát, inak ostávajú nevyužité. Nevýhoda tejto metódy sa prejavuje až pri použití rozmerných konvolučných masiek pri ktorých by percento nevyužitých vláken pre výpočet prevyšovalo počet vláken ktoré počítajú kovolučný filter. Pre veľkosť každého filtra sú tieto nastavenia rozdielne

Vykonanie procesu paralelne v jadrách

V treťom kroku zadávame volanie vykonávajúcej funkcie na GPU. Volanie je podobné ako štandardný zápis aj s parametrami, do volania sa dopĺňajú medzi <<< a >>> parametre pre vykonávanie kernelov, prvý parameter je pole blokov, druhý je pole kernelov v jednom bloku, tretí je voliteľný pri ktorom sa udáva veľkosť zdieľanej pamäte pre blok kernelov.

```
cuda_funkce<<<grid, threadsPerBlock>>>( dev_a, dev_c,
Input.TellWidth(),Input.TellHeight(),divid, kerneldata);
//parametre: (vstupný obraz, výstupný obraz, šírka obrazu,
výška obrazu, deliteľ pre konvolučnú masku, konvolučná
maska)
```

Podľa výberu užívateľa sa zavolá daný spôsob spracovania obrazu. Užívateľsky explicitne zadávané hodnoty filtra pre spracovanie majú preddefinované viaceré varianty v závislosti na úrovni optimalizácie. Táto kapitola je zameraná implementáciu metód ktoré nemajú žiadnu špecifickú optimalizáciu.

Hlavička funkcie CUDA je doplnená o atribút __global__, pretože funkcia bude aplikovaná na všetky volané vlákna programu.

Ukážka neoptimalizovanej funkcie pre spracovanie užívateľom zadávanú hodnotu masky o veľkosti 3x3. Funkcie pre väčšie veľkosti masiek sú vytvárané analogicky.

```
__global__ void cuda33userdefined( unsigned char
*input,unsigned char *output, unsigned int width, unsigned
int height, int divid, int *kerneldata)
{
     unsigned int x=blockIdx.x*TILE_W+threadIdx.x - R;
     unsigned int y=blockIdx.y*TILE_H+threadIdx.y - R;
     x = max (0,x);
     x = min (x,width -1);
     y = max (y,0);
     y = min (y, height -1);
     unsigned int index = y*width +x;
     syncthreads();
     // len vlákna ktoré nezasahuju mimo obraz pokračujú
     if (   (threadIdx.x >= R) && (threadIdx.x < (BLOCK_W -
R))  && (threadIdx.y >= R) && (threadIdx.y < (BLOCK_H -R)))
     {
     float sum = 0;
     for (int dy=-R; dy<=R;dy++)
     {
          for (int dx=-R; dx<=R;dx++)
          {
               float i = input[index + (dy*width) + dx];
               sum += i*kerneldata[(dx+R)*D+(dy+R)];
          }
     }
```

```
      sum=sum/divid;
      if (sum > 255) { sum=255; }
      output[index] = round(sum);
      }
}
```

Prenos výsledku

Po ukončení funkcie prebiehajúcej na GPU pokračuje program na CPU kopírovaním upravených dát naspať do pamäte RAM. Následne dáta uloží do nového výstupného obrazu. Kopírovanie z GPU do RAM je znovu za pomoci cudaMemcpy, tento krát s hodnotou 4. parametra cudaMemcpyDeviceToHost.

Keďže spracovanie prebieha len pre jeden farebný kanál, je nutné skopírovať výstupne dáta na všetky kanály. Následne sa obraz uloží do dočasného súboru out.bmp v adresári, kde je uložený program so všetkými potrebnými náležitosťami pre formát súboru bitmapy. Uložený obraz sa po ukončení funkcie načíta do grafického rozhrania a zobrazí sa ukážka výstupného obrazu.

Na záver funkcia uvoľní pamäte, na GPU sa uvoľnenie použitej pamäte vykonáva deštruktorom cudaFree(nazov_premennej_v_GPU).

5.3 OPTIMALIZÁCIA METÓD PRE GPU

Grafický procesor má vysoký výpočtový výkon, ale má nedostačujúce rýchlosti čítania a zápisu do globálnej pamäte. Tento nedostatok môžeme čiastočne eliminovať použitím špecifických typov pamätí. Možnosti optimalizácie sú aplikované na konkrétne metódy spracovania obrazu.

Použitie pamäti konštánt pri užívateľsky explicitne zadávanej konvolučnej maske

Pamäť konštánt používame na načítavanie konvolučnej masky pri aplikácii filtra na obraz. Táto pamäť umožňuje na jeden prístup do pamäte rozposlať dáta pre všetky vlákna funkcie CUDA, ktoré daný prvok pre výpočet vyžadujú. Toto nám umožňuje dosiahnuť zlepšenie výkonu v spracovaní obrazu a výrazne zredukovať potrebnú pamäťovú priepustnosť. Pamäť je dostupná len na čítanie pre všetky vlákna procesu počas chodu funkcie CUDA.

Inštrukcia pre definovanie pamäte konštánt je __constant__. Musí sa deklarovať mimo tela programu a mimo kernelu CUDA.

```
__constant__ float constmem[D][D];  // globálna konštantná pamäť, veľkosť poľa DxD
```

Pre naplnenie pamäte je nutné vytvoriť si rovnakú premennú o rovnakých rozmeroch ako konštantná pamäť a tú následne prekopírovať do konštantnej pamäte pomocou cudaMemcpyToSymbol.

```
cudaMemcpyToSymbol("constmem",   kernel,D*D*sizeof(float));
//konštantná pamäť, vstupné dáta, veľkosť kopírovaných dát
```

Práca s konštantnou pamäťou je vo funkcii CUDA rovnaká ako s globálnou pamäťou, s tým rozdielom, že nie je možné do nej zapisovať. Aplikácia konštantnej pamäte je demonštrovaná v metóde filtrácie obrazu kde užívateľsky explicitne zadávame hodnoty filtra pre spracovanie obrazu. Funkcia CUDA je zobrazená pre veľkosť 3x3.

```
__global__ void cuda33cmem( unsigned char *input,unsigned
char *output, unsigned int width, unsigned int height, int
divid)
{
     unsigned int x=blockIdx.x*TILE_W+threadIdx.x - R;
    unsigned int y=blockIdx.y*TILE_H+threadIdx.y - R;
     x = max (0,x);
     x = min (x,width -1);
     y = max (y,0);
     y = min (y, height -1);
     unsigned int index = y*width +x;
     syncthreads();
     // len vlákna ktoré nesiahajú mimo môžu pokračovať
     if (   (threadIdx.x >= R) && (threadIdx.x < (BLOCK_W -
R))  && (threadIdx.y >= R) && (threadIdx.y < (BLOCK_H -R)))
     {
     float sum = 0;
     for (int dy=-R; dy<=R;dy++)
     {
          for (int dx=-R; dx<=R;dx++)
          {
               float i = input[index + (dy*width) + dx];
               sum += i*constmem[dy+R][dx+R];
          }
     }
     sum=sum/divid;
     if (sum > 255) { sum=255; }
     output[index] = round(sum);
     }
}
```

Po porovnaní s neoptimalizovanou metódou vidíme minimálny rozdiel. Funkcia nemá ako vstupný parameter dáta s konvolučnou maskou, ale používa pamäť konštánt. Názov konštantnej premennej nie je v funkcii nikde definovaný, bol zadefinovaný ako globálna premenná. Rozdiel v rýchlosti spracovania je však významný.

Použitie zdieľanej pamäte pri aplikácii mediánového filtra

Zdieľaná pamäť slúži na rýchly prenos dát medzi vláknami v jednom bloku vláken. Výhodou použitia zdieľanej pamäte je rýchlosť a jej umiestnenie pri jadre na jednom čipe. Prístup k pamäti trvá len približne štyri takty a je privátna pre každý blok. Nevýhodou je jej malá veľkosť, pre štandard CUDA verzie 1.x je to 16 KB pre jeden multiprocesor.

Keďže funkcia CUDA delí obraz na menšie bloky, ktoré potom spracováva, potrebné vstupné dáta pre spracovanie daného bloku obrazu je možné nakopírovať do tejto zdieľanej pamäte a dosiahnuť tak úsporu priepustnosti pri čítaní z globálnej pamäte. Celá operácia sa definuje v kerneli CUDA. Využitie zdieľanej pamäti demonštrujeme na aplikácii mediánového filtra veľkosti 3x3.

```
__global__ void cuda33median( unsigned char *input,unsigned
char *output, unsigned int width, unsigned int height, int
divid)
{
```

Najskôr je nutné zadefinovať zdieľanú pamäť. Táto deklarácia sa robí s pripojením pred štandardné prvky pripojením `__shared__`. Tým definujeme použitie zdieľanej pamäte.

```
__shared__ unsigned char smem[BLOCK_W*BLOCK_H];
__shared__ unsigned char smem2[BLOCK_W*BLOCK_H][S];
unsigned int x=blockIdx.x*TILE_W+threadIdx.x - R;
unsigned int y=blockIdx.y*TILE_H+threadIdx.y - R;
x = max (0,x);
x = min (x,width -1);
y = max (y,0);
y = min (y, height -1);
```

Pretože spracovanie prebieha v blokoch, pre zjednodušenú prácu zadefinujeme nové indexovanie pre vlákna a pamäťové bloky

```
unsigned int index = y*width +x;
unsigned int bindex =
```

```
threadIdx.y*blockDim.y+threadIdx.x;
     unsigned int bindex2 = (threadIdx.y-
R)*TILE_W+(threadIdx.x-R);
```

Každé vlákno skopíruje z globálnej pamäte dáta jemu prislúchajúce do zdieľanej pamäte, tým dosiahneme naplnenie zdieľanej pamäte.

```
     smem[bindex] = input [index];


     syncthreads(); //synchronizacia vlaken
     // len vlákna ktoré nesiahajú mimo môžu pokračovať
     if (   (threadIdx.x >= R) && (threadIdx.x < (BLOCK_W -
R))  && (threadIdx.y >= R) && (threadIdx.y < (BLOCK_H -R)))
     {
```

Spracovanie mediánového filtra spočíva v utriedení okolia prvkov pre každý pixel obrazu. Pretože sa pri paralelnom spôsobe spracovania vyžaduje množstvo voľnej pamäte pre ukladanie a triedenie dát, využívame pre každé vlákno adresované pole o potrebných rozmeroch. Do tohto poľa sa prvky zo vstupnej zdieľanej pamäte prekopírujú a následne podľa podmienok mediánového filtra utriedia.

```
int k = 0;
for (int dy=-R; dy<=R;dy++)
{
          for (int dx=-R; dx<=R;dx++)
          {

smem2[bindex2][k++] = smem[bindex + (dy*blockDim.x) + dx];
// kopírovanie dát do pomocných premenných
          }
     }
    int n=9;//velkost pola
 int L = 0;
 int Q = n-1;
 k = n / 2;
 int o;int p;//definovanie potrebných premenných
 while (L < Q)
 {
  float x = smem2[bindex2][k];
```

```
  o = L; p = Q;
do
 {
  while (smem2[bindex2][o] < x)o++;
  while (x < smem2[bindex2][p])p--;
  if (o <= p)
  {
    float t = smem2[bindex2][o];
    smem2[bindex2][o] = smem2[bindex2][p];
    smem2[bindex2][p] = t;
    o++;p--;
  }
 } while (o <= p);
  if (p < k)  L = o;
  if (k < o)  Q = p;
 }//selektívnym algoritmom sa nájde mediánová hodnota
output[index] = smem2[bindex2][k];// zápis do výstupu
     }
}
```

Pri viacnásobných prístupoch do pamäte je výhodné použiť pamäť zdieľanú, obzvlášť pri konvolučnom spracovávaní obrazu, kedy sa jeden pixel načítava na spracovanie viacnásobne. Po jednorazovom načítaní do zdieľanej pamäte potom rýchlosť čítania a zápisu dát je výrazne rýchlejšia oproti viacnásobnému čítaniu z globálnej pamäte. Problém je pri nutnosti použiť väčšie množstvo dát. Pri aplikácii mediánového filtra väčšej veľkosti ako 9x9 nastáva problém v tom, že nedokážeme poskytnúť toľko pamäťových prostriedkov koľko ich systém vyžaduje. Napríklad v mediánovom filtrovaní veľkosťou masky 9x9 používame zdieľanú pamäť o veľkosti 12 KB pri použití veľkosti spracovávaného bloku 12x12 pixelov v jeden okamih len na triedenie a výber mediánovej hodnoty. Preto sa u väčších rozmerov masiek používa zdieľaná pamäť len čiastočne, a to na načítanie hlavných dát potrebných na spracovanie. Triedenie a zápis výsledku následne prebieha v pomalej globálnej pamäti a celý výpočet je menej efektívny.

Použitie konštantnej a zdieľanej pamäti pri filtrácii Sobelovou metódou

Kombináciou oboch pamäťových optimalizácii dosahujeme optimálne vyváženie medzi prísunom dát na spracovanie a výpočtovými schopnosťami GPU.

Možnosti konštantnej pamäte využívame pri aplikácii rôznych masiek Sobelovho filtra. Keďže konvolučných masiek môže byť podľa potreby spracovania viac, urýchlenie spracovania na GPU bude násobne efektívnejšie.

Možnosti zdieľanej pamäte využívame na prísun dát a zrýchlenie dodania potrebných podkladových dát pre spracovanie konvolučnou maskou. Pri použití konvolučnej masky väčších rozmerov stúpa efektivita využitia zdieľanej pamäte, pretože jeden pixel obrazu je nutné načítavať viacnásobne podľa rozmerov konvolučnej masky.

6 ANALÝZA DOSIAHNUTÝCH VÝSLEDKOV

Rôzne filtračné metódy použité v aplikácii nám umožňujú porovnať efektivitu riešení a optimalizácii filtračného mechanizmu. Na základe týchto porovnaní potom môžeme rozhodnúť ktoré vlastnosti grafického hardwaru je výhodné použiť pre spracovanie obrazu.

Rýchlosť spracovania v oboch aplikáciách bola zisťovaná pomocou merania rozdielu systémového času. Systémový čas zaznamenáva počet tikov od určitej počiatočnej doby. Rozlišovacia schopnosť tohto merania je štandardne 1 ms, avšak môže dochádzať k malým odchýlkam hlavne pri spracovaní času aplikáciou a zaokrúhľovaním na celé milisekundy. Na presné určenie času trvania určitých funkcií CUDA je možné použiť Nvidia Visual Profiler ktorý monitoruje priebeh programu a jeho interakcie s grafickou kartou Nvidia. Na základe monitoringu vygeneruje grafickú časovú os na ktorej je zobrazenú priebeh funkcií využívajúcich prostriedky GPU. Analýza zobrazuje aj využitie pamäťových prostriedkov a doby trvania špecifických procedúr ako napríklad kopírovanie do a z pamäti GPU alebo alokovanie priestoru na GPU cez cudaMalloc. Spracované výsledky sú zhrnuté do grafov a porovnané medzi sebou. Výsledky sú viazané na konkrétny hardware, pri použití iného hardwaru sa výsledky môžu líšiť.

Prvá sada grafov č. 1 až 4 sa vzťahuje na k stolnému počítaču s konfiguráciou

Procesor: INTEL Core 2 Duo E8400 (6M Cache, 3,00 GHz, 1333 MHz FSB)
Grafická karta: Nvidia 210 (16 CUDA jadier, 589 MHz, 512 MB GDDR3)

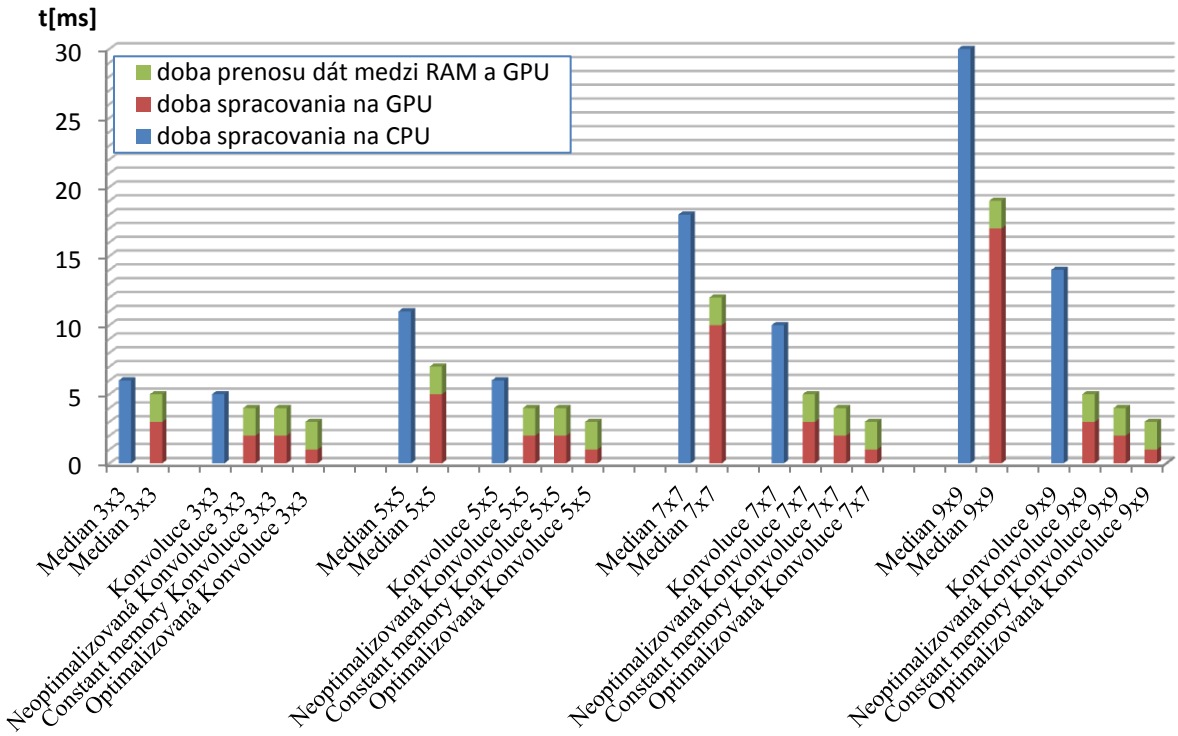
Druhá sada grafov č. 5 až 8 sa vzťahuje k notebooku s konfiguráciou prvých GPU Nvidia podporujúcich CUDA

Procesor: INTEL Core 2 Duo T9300 (6M Cache, 2,50 GHz, 800 MHz FSB)
Grafická karta: Nvidia Quadro NVS140M (16 CUDA jader, 530 MHz, 128 MB GDDR3)

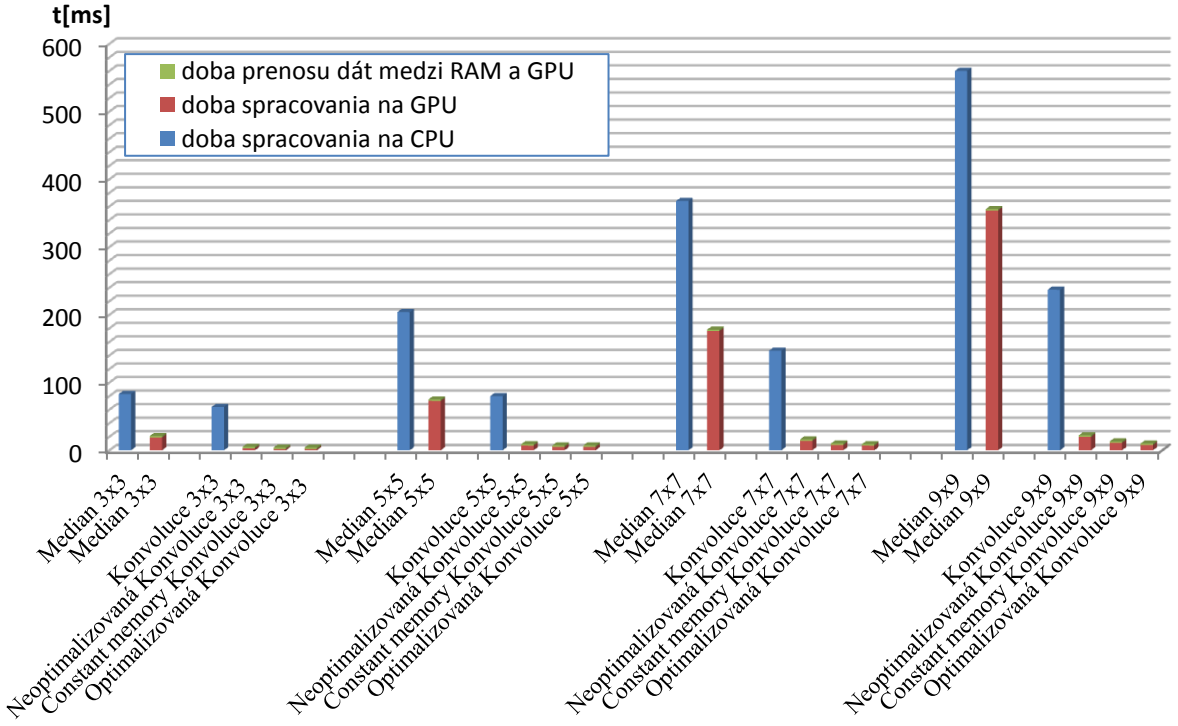
Nevýhodou grafickej karty na notebooku je jej nízky výkon a veľmi malé množstvo grafickej pamäti ktorá často vedie k rôznym problémom pri použití platformy CUDA.

Všetky grafy sú vytvárané pre veľkosti obrazov 128x128 pixelov, 512x512 pixelov, 1024x1024 pixelov a 4096x4096 pixelov tak aby bolo poukázané na rozdiely pri rôznych veľkostiach spracovávaných dát. Na vodorovnej osi sú zobrazené varianty obrazových filtrov pri použití rôznej optimalizácie spracovania.

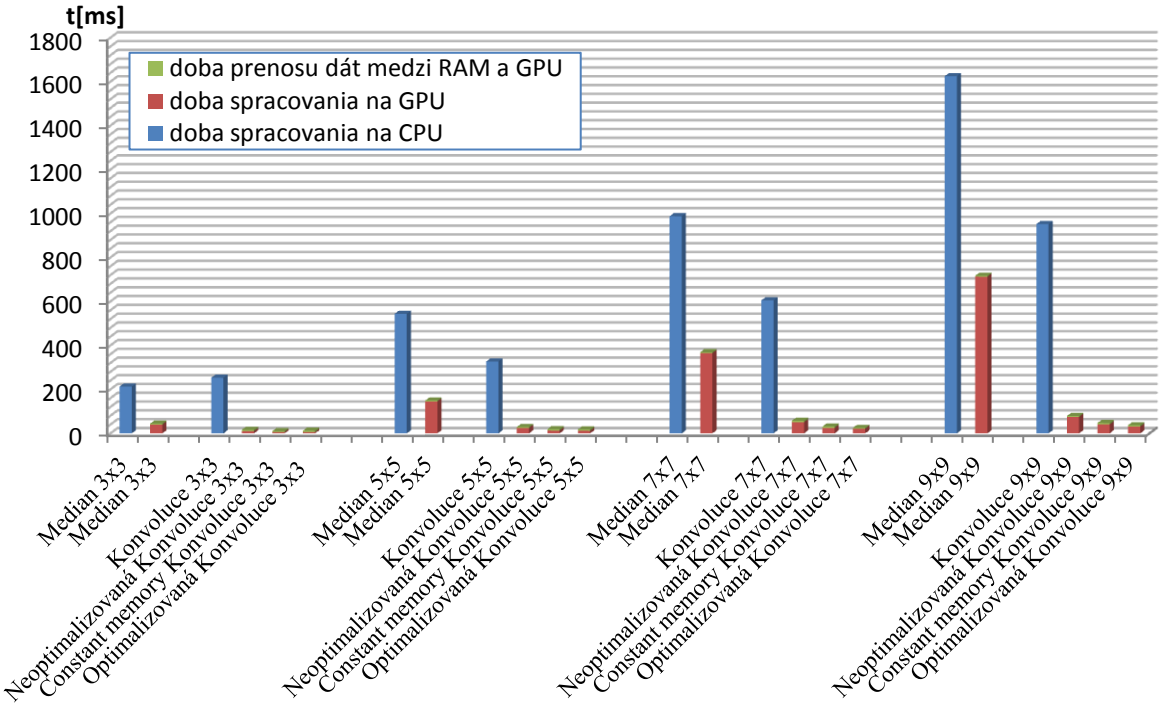
Graf č. 1: Graf rýchlostí spracovania obrazu veľkosti 128x128 pixelov na PC



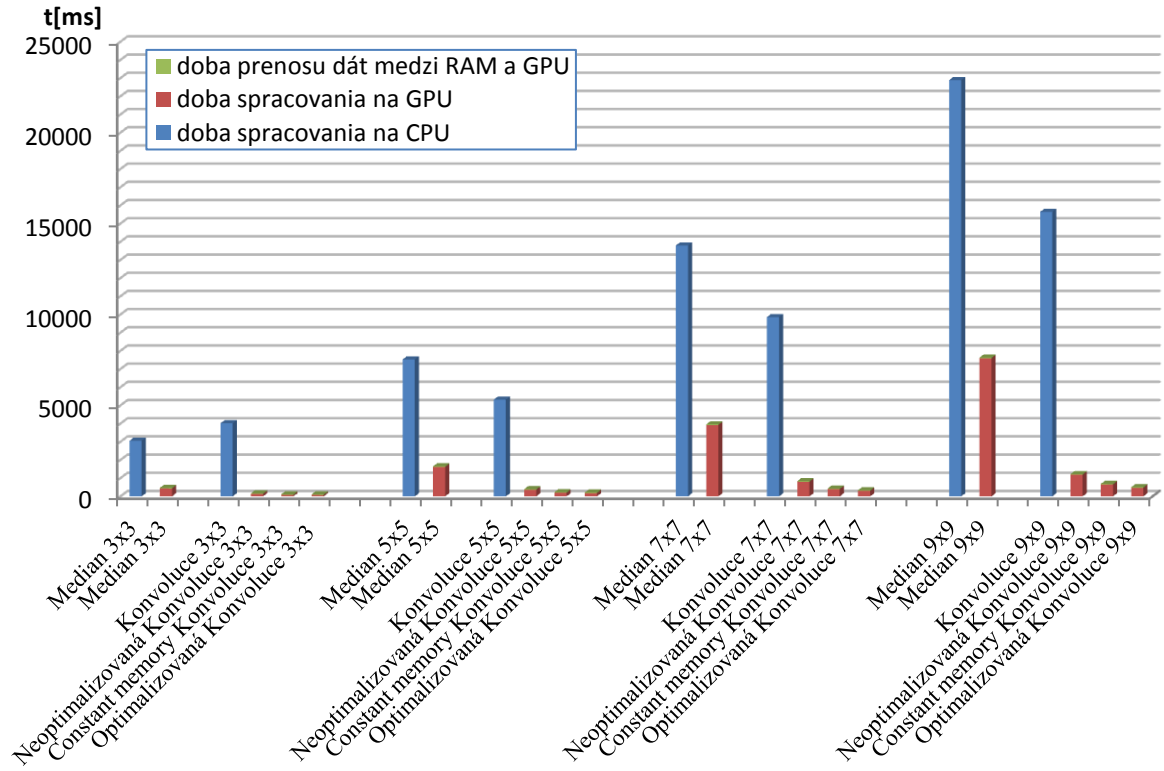
Graf č. 2: Graf rýchlostí spracovania obrazu veľkosti 512x512 pixelov na PC



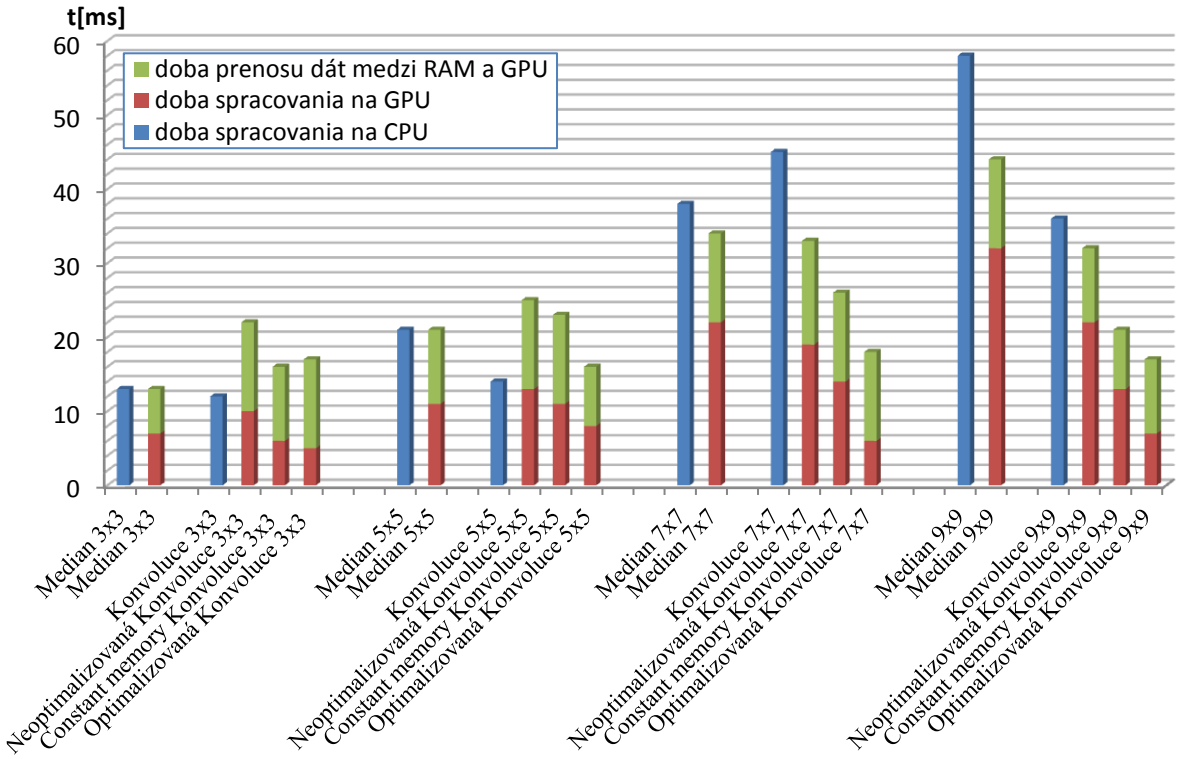
Graf č. 3: Graf rýchlostí spracovania obrazu veľkosti 1024x1024 pixelov na PC



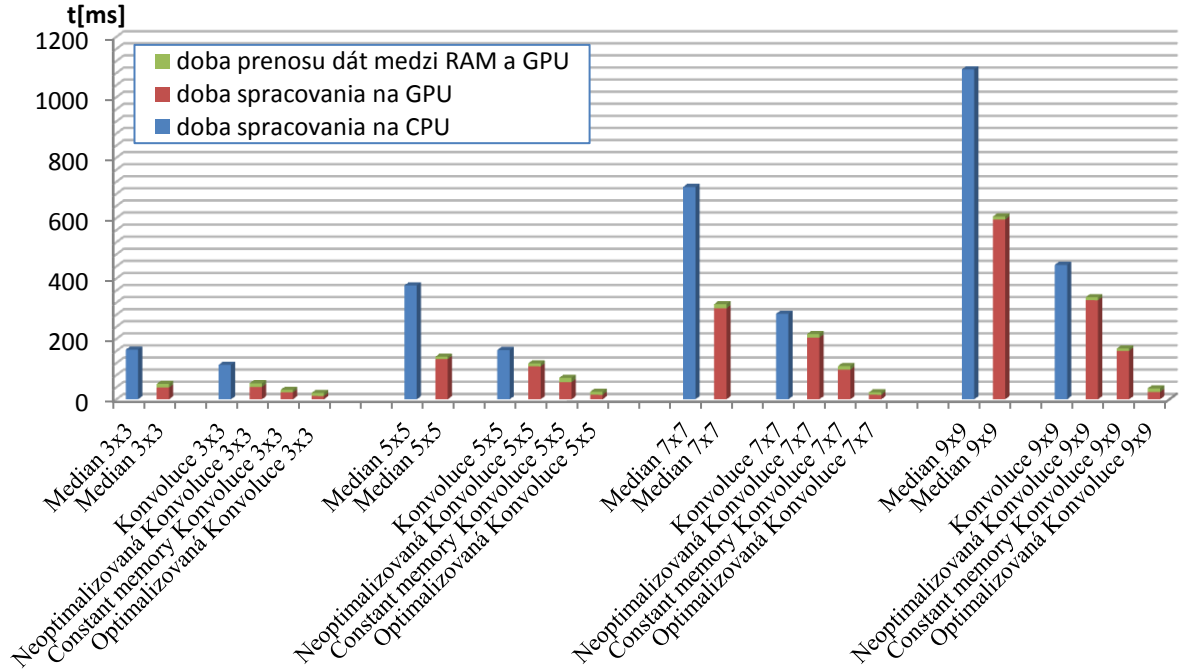
Graf č. 4: Graf rýchlostí spracovania obrazu veľkosti 4096x4096 pixelov na PC



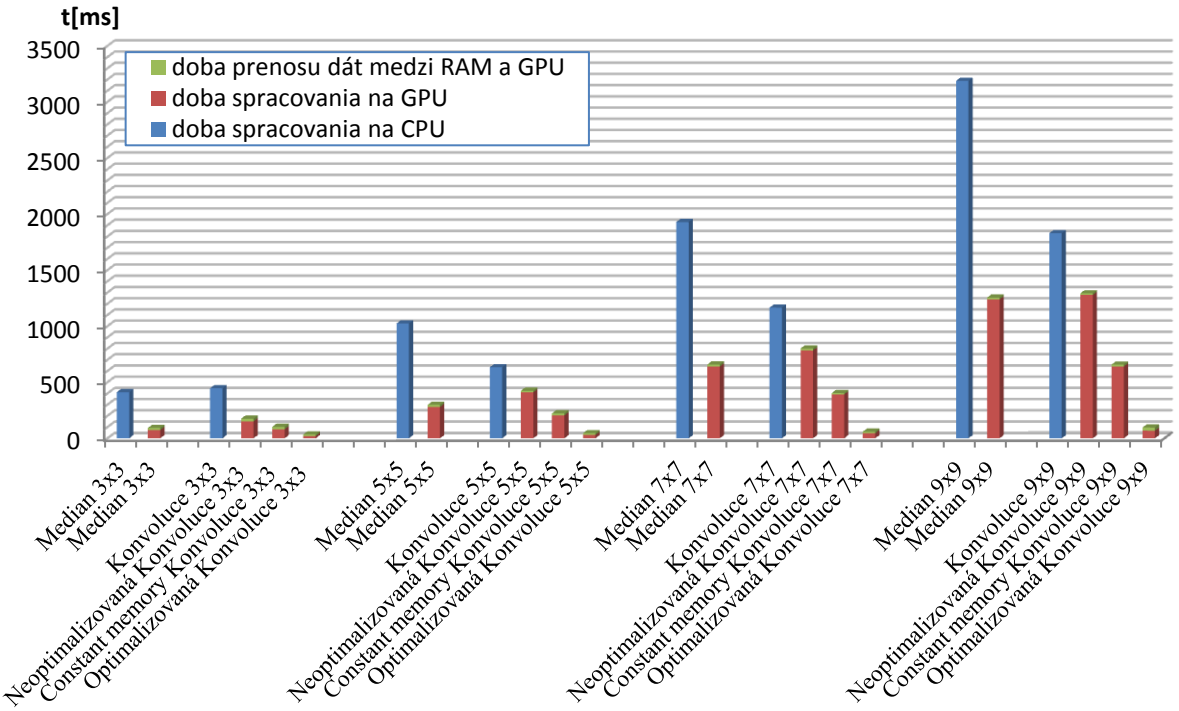
Graf č. 5: Graf rýchlostí spracovania obrazu veľkosti 128x128 pixelov na notebooku



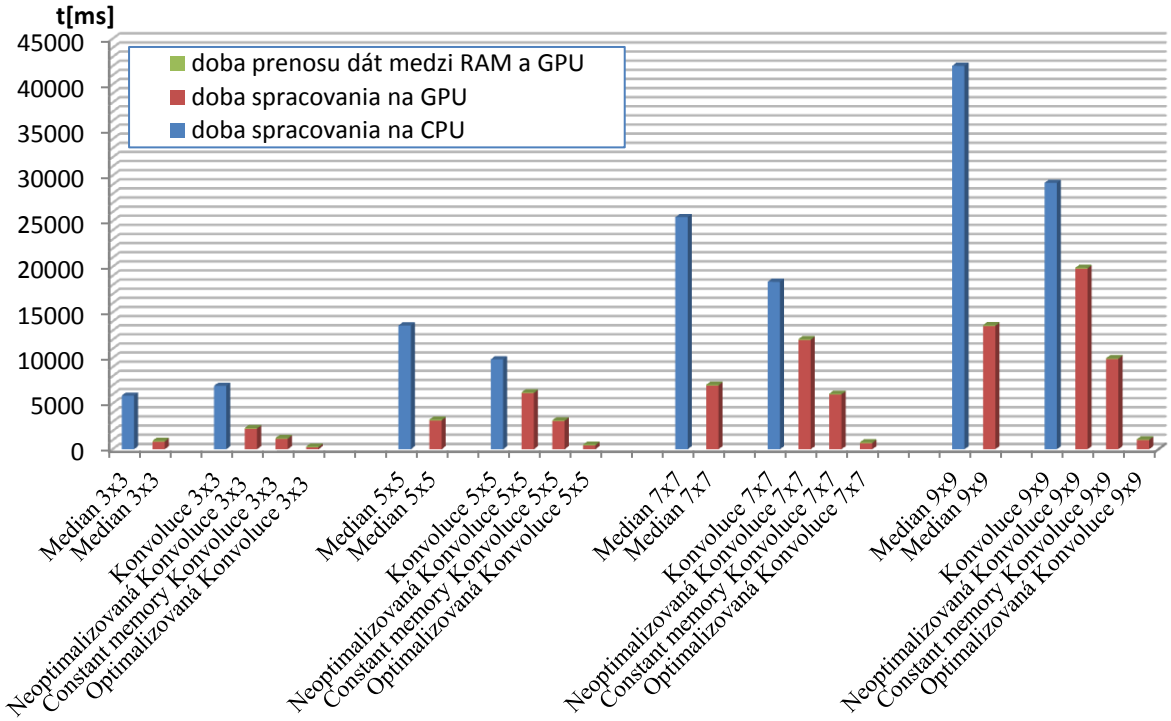
Graf č. 6: Graf rýchlostí spracovania obrazu veľkosti 512x512 pixelov na notebooku



Graf č. 7: Graf rýchlostí spracovania obrazu veľkosti 1024x1024 pixelov na notebooku



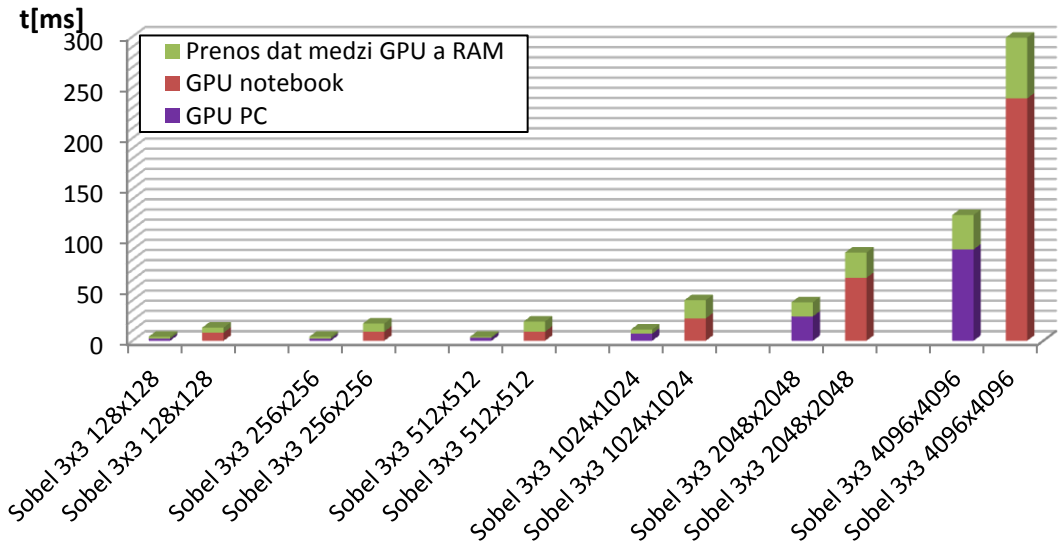
Graf č. 8: Graf rýchlostí spracovania obrazu veľkosti 4096x4096 pixelov na notebooku



Z grafov vyplýva že paralelné spracovanie obrazu na GPU je v každom prípade rýchlejšie ako na CPU. Pri menších veľkostiach obrazu a menšej veľkosti konvolučnej masky je nevýhoda že úkony potrebné pred začatím spracovania na GPU a úkony po spracovaní sú významná položka pri rozhodovaní o efektivite paralelného spracovania obrazu ako napríklad v grafe č. 5 pri použití masky veľkosti 3x3 a 5x5. Pri použití staršieho hardwaru ktorého prenosové rýchlosti zbernice a takt jadra grafického procesora nie sú dostatočne výkonné na to, aby dorovnali priame prístupy do pamäti pri spracovaní cez CPU, môžeme považovať investíciu do paralelného spracovania obrazu na grafickej karte za neefektívnu. Toto tvrdenie však vyvracia použitie väčších konvolučných masiek či iného zložitejšieho algoritmu, ktorý pri použití CPU nemáme možnosť urýchliť, napríklad použitie konvolučnej masky veľkosti 9x9 či mediánového filtra rovnakej veľkosti. Pri týchto metódach sa zvyšuje výhoda rýchlych spoločných pamäťových prístupov do pamäti GPU a rýchlosť spracovania napriek zložitejšej správe pamätí je vyššia.

Pri väčších veľkostiach obrazu je za každých okolností použitie grafickej karty efektívnejšie, násobky zrýchlenia sú rozdielne v závislosti na použití hardwaru, úrovni optimalizácie spracovania a v neposlednej rade závisia na správnej voľbe výpočtového algoritmu. Rozdiel medzi výkonom grafického hardwaru je zobrazený v grafe nižšie. Použitie výkonnejšieho hardwaru dosiahneme takmer 3 násobné zrýchlenie len pri použití základnej low-end grafickej karty z dnešnej ponuky oproti 4 roky starej grafickej karte na notebooku podporujúcej základné funkcie CUDA verzie 1.1. Rozdiely sú porovnané v grafe č. 9 na aplikácii sobelovej metódy na oboch typoch hardwaru.

Graf č. 9: Graf rýchlostí spracovania obrazu rôznych veľkosti na PC a notebooku metódou Sobelovho operátora



Pri použití nových výkonných grafických kariet architektúry Fermi podporujúcej CUDA verzie 2.0 a vyššie dosahujeme pri spracovaní obrazu vyššie rýchlosti. Je to najmä z dôvodu že nová architektúra počíta s cachovaním dát, čo pri spracovaní obrazu urýchľuje prísun dát na spracovanie do jadra GPU.

Pri spracovaní obrazu veľkosti 4096x4096 pixelov a použití konvolučnej metódy spracovania obrazu výmena grafickej karty v PC zvýšila efektivitu spracovania oproti CPU takmer 13x bez akejkoľvek optimalizácie pre spracovanie na grafickej karte.

Optimalizáciou môžeme dosiahnuť výrazné zrýchlenie doby spracovania. Základný princíp spracovania obrazu na GPU pracuje podobne ako spracovanie na CPU. Všetky dáta sú uložene v GRAM ku ktorým pristupuje grafický procesor a spracuje výsledok. Pretože je grafický procesor výkonnejší ako štandardný CPU pri paralelných operáciách potrebuje k dispozícii dostatočne rýchlu pamäť. Keďže dnes neexistuje dostatočne rýchla operačná pamäť o potrebnej veľkosti ktorá by bola komerčne dostupná pre použitie, vývojári majú možnosť použiť rozšírené registre a špecializované druhy pamätí priamo na grafickom čipe. Nevýhodou týchto pamätí je ich malá kapacita.

Optimalizácia v použití pamäti konštánt spočíva v presunutí konštantných dát ako je konvolučná maska do tejto oblasti, ku ktorej má grafický čip výrazne kratší prístup. Touto optimalizáciou pri použití rovnakých parametrov dosahujeme 23-násobné zrýchlenie doby trvania procesu oproti štandardu na CPU.

Ďalšou optimalizáciou využitia špecializovaných pamätí je použitie zdieľanej pamäti. do tejto pamäti je možné ukladať dáta s ktorými grafické jadro aktuálne pracuje a viacnásobne k nim pristupuje. Aplikáciou tejto optimalizácie spoločne s použitím pamäti konštánt dosahujeme až 31-násobné zrýchlenie spracovania obrazu veľkosti 4096x4096 pixelov konvolučnou maskou veľkosti 9x9.

Voľbou správneho algoritmu spracovania rozhodujeme o celkovej efektivite celého programu. Ideálnym spôsobom pre paralelné spracovanie obrazu na grafickej karte je znížiť nutný počet prístupov programu do pamäti. Napríklad pri vývoji mediánového filtra zmena triediaceho algoritmu ktorý pracoval na metóde buble sort na selektívny algoritmus urýchlila spracovanie obrazu štvornásobne.

Z analýzy vyplýva že paralelizácia výpočtov na grafickej karte je neefektívna pri spracovaní príliš malých obrazov najmä pri menej výkonnom hardware. V ostatných prípadoch paralelné spracovanie obrazu s využitím metód konvolúcie je omnoho efektívnejšie pred sériovým spracovaním. Prehľad maximálnej dosiahnutej efektivity výpočtov pri vybraných algoritmoch je uvedený v tabuľkách č. 1-3. Z tabuľky č. 3 ktorá popisuje mediánový filter môžeme povedať že implementácia triediacich algoritmov nieje pre GPU príliš vhodná a nedosahuje až také pozitívne zlepšenie výkonu ako

napríklad použitie sobelovho operátora ktorý používa viacnásobné statické masky, viď tabuľka č. 3. Pri použití veľkého obrazu a viacnásobných prechodov môžeme povedať že spracovanie obrazu je viac efektívne hlavne preto, že prednačítané dáta v jadre GPU ktoré využívaju menšie okolie obrazu je možné podstatne rýchlejšie spracovať s pripravenými konvolučnými maskami a dosiahnuť tak vyšší výkon ako pri volaní konvolučných metód sekvenčne za sebou.

Pomer časovej náročnosti na GPU optimalizovaných filtroch v porovnaní s CPU

Tabuľka č. 1: Spracovanie obrazu konvolučným filtrom

Veľkosť obrazu	128x128	256x256	512x512	1024x1024	2048x2048	4096x4096
Veľkosť masky 3x3	1,25	4,25	21,33	21,08	30,91	34,96
Veľkosť masky 5x5	1,50	5,50	13,33	18,17	20,38	24,85
Veľkosť masky 7x7	2,50	7,20	18,38	24,20	26,66	29,17
Veľkosť masky 9x9	3,50	8,43	26,33	25,73	29,01	31,04

Tabuľka č. 2: Spracovanie obrazu mediánovým filtrom

Veľkosť obrazu	128x128	256x256	512x512	1024x1024	2048x2048	4096x4096
Median 3x3	0,86	2,29	4,15	4,84	5,64	6,56
Median 5x5	1,57	1,82	2,76	3,65	4,10	4,56
Median 7x7	1,50	2,00	2,08	2,68	3,09	3,49
Median 9x9	1,58	1,60	1,58	2,27	2,63	3,00

Tabuľka č. 3: Spracovanie obrazu sobelovým filtrom

Veľkosť obrazu	128x128	256x256	512x512	1024x1024	2048x2048	4096x4096
Sobel 3x3	1,75	6,25	24,00	35,55	41,66	49,89

ZÁVER

Táto práca bola venovaná paralelnému spracovaniu obrazu a jeho využitiu v praxi. V prvej časti sú zhrnuté základné poznatky o spracovaní obrazu. Spracovanie obrazu je vhodné na paralelizáciu úloh, pretože obraz je možné rozdeliť na menšie časti a pracovať s nimi nezávisle, paralelne. To nám umožňuje využiť výpočtové schopnosti GPU, ktoré sú optimalizované na vykonávanie jedeného výpočtu na rôznych dátach v jeden okamih. V práci sú popísané základné vlastnosti štandardu CUDA, ktorý je optimalizovaný na rýchle výpočty. CUDA je viazaná na konkrétny hardware Nvidia. Výkon GPU bol využitý v aplikácií, ktorá demonštruje rôzne konvolučné metódy spracovania obrazu. Metódy boli vytvorené najprv pre štandardný procesor a na základe nich sa porovnávali rôzne optimalizácie pri použití grafickej karty ako urýchľovača výpočtov paralelným systémom spracovania.

Cieľom práce bolo zistiť pri akých podmienkach má zmysel použiť paralelné spracovanie obrazu. Pri použití konvolučných metód, ako napríklad rôzne priemerovacie filtre veľkosti 3x3, 5x5, 7x7, 9x9, mediánový filter, Sobelov filter, Laplaceov hranový detetektor, je veľmi výhodné uvažovať o využití paralelného spracovania obrazu na grafickej karte. Efektívnosť tejto metódy je nízka len pri špecifických situáciách, napríklad filtrovanie malých veľkostí obrazu alebo použití zastaraného hardwaru. V opačných prípadoch je násobne rýchlejšie spracovanie zreteľné. V tejto práci sa dosiahlo až 31-násobné zvýšenie rýchlosti spracovania obyčajnou konvolúciou veľkosti obrazu 4096x4096 pixelov na CPU pri použití bežne dostupného grafického hardwaru prijateľnej ceny. Dôležitým aspektom pri využití dostupného paralelného výkonu je optimalizácia algoritmov, pri ktorých musíme dbať na obmedzenia grafických kariet a ich nevýhody. Správnym určením ako budú využité dostupné pamäťové bloky grafickej pamäte môžeme dosiahnuť ďalšie značné zlepšenie výpočtového výkonu a tým skrátiť aj dobu potrebnú na spracovanie. Pri nevyužívaní dostupných optimalizácii efektivita rýchlosti výpočtov klesá. Z tohto dôvodu boli vytvorené demonštračné metódy, ktoré využívajú rôzne metódy optimalizácie. Implementácia bez akýchkoľvek optimalizácii na grafickej karte vedie k 12,7 násobnému zvýšeniu výkonu pri veľkosti obrazu 4096x4096 a konvolučnej masky 9x9. Použitie optimalizácie s využitím pamäti konštant vedie už k 22,8 násobnému zlepšeniu rýchlosti výpočtu. Použitie zdieľanej pamäti a načítavanie častí obrazu do pamäti na čipe jadra GPU vedie k zlepšeniu výkonu na úroveň 31x oproti CPU.

Ďalšou časťou práce bolo porovnať rôzne filtračné algoritmy a rozhodnúť na aký typ spracovania obrazu je grafická karta využitelnejšia. Najhorší výsledok dosiahol mediánový filter, pre ktorého použitie je nutné implementovať triediaci algoritmus.

Grafická karta je silná na výpočty, avšak zaostáva s rýchlosťou pamäti. Dosiahnutá rýchlosť spracovania pre veľkosť obrazu 4096x4096 pri použití mediánového filtra veľkosti 9x9 mala len trojnásobné zvýšenie rýchlosti spracovania. Môžeme predpokladať, že grafická karta nie je veľmi efektívna pri triedení dát a veľkej práci s dátami, ktoré produkujú množstvo komunikačných požiadavok a spomaľujú celkový výpočet.

Opačným prípadom je sobelov operátor, ktorý využíva viacero statických konvolučných masiek pre výpočet obrazového bodu. V práci bol implementovaný dvojprechodový sobelov filter, pri ktorom sme dosiahli zvýšenie výkonu takmer 50 násobné pri veľkosti obrazu 4096x4096. Z týchto výsledkov je možné usúdiť, že použitie viacerých filtrácii sekvenčne je vhodné implementovať do jedného kernelu v GPU tak, aby využíval už načítané dáta a nezvyšoval úroveň nutnej komunikácie jadra a globálnej pamäte GPU.

Urýchlenie spracovania obrazu na grafickej karte je možné len odporučiť. Jedným z dôvodov je, že grafická karta je primárne navrhnutá ako zobrazovacie zariadenie na operácie s obrazom. Rýchlosť spracovania potom závisí na použitých metódach a úrovni optimalizácie, či schopnosti programátora.

LITERATÚRA

[1] GONZALEZ, R., C., WOODS, R., E. *Digital image processing*. 2. vydanie. Upper Saddle River: Prentice Hall, 2002, 793 s. ISBN 02-011-8075-8.

[2] HLAVÁČ, V., ŠONKA, M. *Počítačové vidění*. Praha: Grada, 1992, 272 s. ISBN 80-854-2467-3.

[3] FARBER, R., *CUDA application design and development*. Amsterdam: Elsevier, 2011, č. 17, 315 s. ISBN 978-0-12-388426-8.

[4] SANDERS, J., *CUDA by example: an introduction to general-purpose GPU programming*. 1. vydanie. Upper Saddle River: Addison-Wesley, 2010, 290 s. ISBN 978-0-13-138768-3.

[5] JAN, J. *Medical image processing, reconstruction and restoration: concepts and methods*. Boca Raton: Taylor, 2006, 730 s. ISBN 08-247-5849-8.

[6] NVIDIA: *NVIDIA CUDA Compute Unified Device Architecture Programming Guide [online]*. Verzia 1.1. NVIDIA, 29.11.2007 [cit. 2013-05-10]. Dostupné z: http://www.mpe.mpg.de/~umaio/manual_NVIDIA_CUDA_Programming_Guide_1.1.pdf

[7] NVIDIA: *CUDA Image Convolution with CUDA [online]*. NVIDIA, 2007 [cit. 2013-05-10]. Dostupné z: http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_64_website/projects/convolutionSeparable/doc/convolutionSeparable.pdf

[8] NVIDIA: *CUDA Programming Model Overview [online]*. NVIDIA, 2008 [cit. 2013-05-10]. Dostupné z: http://www.scribd.com/doc/61664546/1/CUDA-Programming-Model-Overview

[9] Úvod do technologie CUDA *[online]*. 2009 [cit. 2013-05-10]. Dostupné z: http://www.root.cz/serialy/uvod-do-technologie-cuda/

ZOZNAM SYMBOLOV, VELIČÍN A SKRATIEK

ALU	Arithmetic logic unit, aritmeticko logická jednotka
CPU	Central Processing Unit, centrálna procesorová jednotka
CUDA	Compute Unified Device Architecture, výpočetne zjednotená architektúra zariadení
Device	v CUDA terminológií grafické zariadenie, výpočtová karta
GPU	Graphical Processing Unit, grafická procesorová jednotka
GPGPU	General-purpose graphics processing unit – grafická jednotka na všestranné využitie
Host	v CUDA terminológií hostiteľ grafického zariadenia
Kernel	výpočtové jadro
OpenCL	Open Computing Language, otvorený výpočtový jazyk
RAM	random access memory, operačná pamäť počítača
Thread	vlákno, primitívny exekučný blok