

ŽILINSKÁ UNIVERZITA V ŽILINE

FAKULTA RIADENIA A INFORMATIKY

BAKALÁRSKA

PRÁCA

ANDREJ BELIANČÍN

**Využitie GPU paralelizovaných výpočtov na generovanie
umelých dát**

Vedúci práce: Ing. Peter Tarábek, PhD.

Registračné číslo: 213/2017

Žilina, 2018

ŽILINSKÁ UNIVERZITA V ŽILINE

FAKULTA RIADENIA A INFORMATIKY

BAKALÁRSKA

PRÁCA

ŠTUDIJNÝ ODBOR: INFORMATIKA

ANDREJ BELIANČÍN

**Využitie GPU paralelizovaných výpočtov na generovanie
umelých dát**

Žilinská univerzita v Žiline

Fakulta riadenia a informatiky

Školiace pracovisko: Katedra matematických metód a operačnej analýzy

Žilina, 2018

ZADANIE TÉMY BAKALÁRSKEJ PRÁCE.

Študijný odbor : Informatika

Meno a priezvisko

Andrej Beliančin

Osobné číslo

555769

Názov práce v slovenskom aj anglickom jazyku

Využitie GPU paralelizovaných výpočtov na generovanie umelých dát

Generating artificial data using parallel processing on GPU

Zadanie úlohy, ciele, pokyny pre vypracovanie

(Ak je málo miesta, použite opačnú stranu)

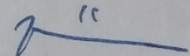
Cieľ bakalárskej práce:

Pri vývoji metód počítačového videnia ako sú detekcia, rozpoznávanie a segmentácia objektov je jedným z úzkym miest vývoja dostupnosť dostatočne reprezentatívneho datasetu. Jednou možnosťou, ako tento problém riešiť, je využitie metód predspracovania obrazu na generovanie umelých dát. Pod pojmom generovanie umelých dát je myslená buď tvorba drobných zmien v obraze alebo úprava vstupných dát do formy, ktorá je jednoduchšia na ďalšie spracovanie. Táto práca sa zameriava na implementovanie vybraných metód spracovania obrazu na GPU. Cieľom práce je nielen implementácia metód, ale aj výber samotných vhodných metód. Vhodnosť metódy sa bude posudzovať podľa možnosti jej využitia pre samotné generovanie dát, ako aj možnosti úspory času vďaka paralelnej implementácii. Z tohto dôvodu bude výber metód(y) pre implementáciu na GPU robený v súčinnosti s vedúcim práce počas jej riešenia na základe analýzy súčasného stavu.

Obsah:

1. Analýza súčasného stavu zameraná na identifikovanie vhodných metód úpravy obrazu.
2. Oboznámenie sa s programovacím modelom CUDA.
3. Výber metód(y) vhodných pre implementáciu na GPU.
4. Implementácia na CPU a GPU. CPU implementácia bude slúžiť ako referenčná implementácia pre testovanie správnosti GPU implementácie. CPU implementácia môže byť prebratá z tretej strany.
5. Experimentálne zhodnotenie a záver.

Meno a pracovisko vedúceho BP: Ing. Peter Tarábek, PhD., KMMOA, ŽU
Meno a pracovisko tutora BP:

5.2.2018 

vedúci katedry
(dátum a podpis)

Čestné vyhlásenie

Čestne prehlasujem, že som prácu vypracoval samostatne s využitím dostupnej literatúry a vlastných vedomostí. Všetky zdroje použité v bakalárskej práci som uviedol v súlade s predpismi.

Súhlasím so zverejnením práce a jej výsledkov.

V Žiline, dňa 30. 4. 2018

Andrej Beliančín

Pod'akovanie

Rád by som na tomto mieste poďakoval vedúcemu mojej bakalárskej práce Ing. Petrovi Tarábekovi, PhD. za odbornú pomoc, pripomienky, vecnú kritiku a cenné rady pri tvorbe práce.

Abstrakt

BELIANČÍN, Andrej: *Využitie GPU paralelizovaných výpočtov na generovanie umelých dát*. [Bakalárska práca]. – Žilinská univerzita. Fakulta riadenia a informatiky, Katedra matematických metód a operačnej analýzy; – Školiteľ/Vedúci: Ing. Peter Tarábek, PhD.– Stupeň odbornej kvalifikácie: Bakalár v odbore Informatika – Žilina: FRI ŽU v Žiline, 2018. – 76 s.

Cieľom tejto bakalárskej práce je využitie GPU paralelizovaných výpočtov na generovanie umelých dát. Na základe analýzy metód predspracovania obrazu bola vybraná tá metóda, ktorá je vhodná pre úpravu obrazu a ktorej operácie je možné paralelizovať na grafickej karte. Pri rozhodovaní, ktorá metóda bude implementovaná, sme brali ohľad na dostupnosť a kvalitu existujúcich CPU a GPU implementácii konkrétnej metódy. Po analýze vybraných metód predspracovania obrazu a porovnaní existujúcich implementácií sme sa rozhodli implementovať metódu diskretnej dvojrozmernej konvolúcie, ktorá je základom pre mnoho úloh predspracovania obrazu a z pohľadu použitia existuje viacero druhov konvolúcie (1:1:1, 1:N:N a N:M:M), o ktorých sa bližšie pojednáva v práci. Požiadavkou bolo implementovať túto metódu na GPU prostredníctvom programovacieho modelu CUDA. Konvolúcia je výpočtovo náročná operácia a použitím GPU paralelizovaných výpočtov sa dá očakávať zrýchlenie konvolúcie v porovnaní s CPU implementáciou. Snahou bolo vytvoriť GPU implementáciu efektívnejšiu ako tá, ktorá už existuje. GPU implementácie sme vytvárali iteráčným postupom, v každej iterácii je možné pozorovať isté zlepšenie oproti predošlej. Pri experimentálnom porovnaní sme došli k zisteniu, že naša implementácia je podstatne rýchlejšia ako CPU implementácia z OpenCV a dokonca, za určitých podmienok 2 až 3 násobne rýchlejšia ako existujúca GPU OpenCL implementácia z knižnice OpenCV, ktorá je považovaná za štandard v oblasti spracovania obrazu.

Kľúčové slová: predspracovanie obrazu, konvolúcia, programovanie na grafickej karte, CUDA

Abstract

BELIANČÍN, Andrej: Generating artificial data using parallel processing on GPU. [Bachelor thesis]. – University of Žilina. Faculty of Management Science and Informatics, Department of Mathematical Methods and Operations Research; – Supervisor: Ing. Peter Tarábek, PhD.– Degree of qualification: Bachelor of Informatics – Žilina: FRI ŽU in Žilina, 2018. – 76 p.

The aim of this bachelor thesis is generating artificial data using parallel processing on GPU. Based on analysis of methods used in image preprocessing, such method was selected, which is suitable for image processing and can be easily parallelized on GPU. When deciding, which method to implement, we took into consideration the availability and quality of existing CPU and GPU implementations of concrete method. After the analysis of selected methods of image preprocessing and the comparison of existing implementations, we decided to implement method called discrete two-dimensional convolution, which is fundamental for numerous tasks in image preprocessing and there are more kinds of convolution(1:1:1, 1:N:N, N:M:M), which will be discussed in detail in thesis. Our implementation had to satisfy the requirement to be implemented on GPU through CUDA programming model. Convolution is a computationally intensive operation and it is expected to significantly accelerate convolution by applying parallel processing on GPU compared to its CPU implementation. We made an effort to create GPU implementation more effective than existing one. GPU implementations are result of iterative process, where one can observe progress in every iteration. After experimental evaluation we found out that our GPU implementation is considerably faster, compared to CPU implementation from OpenCV and under certain circumstances 2-3 times faster than existing GPU OpenCL implementation from OpenCV, which is considered as standard in image processing.

Key words: image preprocessing, convolution, programming on graphic card, CUDA

Obsah

Zoznam obrázkov	9
Zoznam tabuliek	10
Úvod	11
1 Oboznámenie sa s programovacím modelom CUDA.....	13
1.1 Programátorský model	13
1.2 Hierarchia vlákien	15
1.3 Pamäťová hierarchia	16
1.4 Hardvérová implementácia	17
1.5 Reprezentácia čísel s pohyblivou desatinnou čiarkou na GPU.....	19
2 Vybrané metódy predspracovania obrazu.....	21
2.1 Bodové jasové transformácie	22
2.1.1 Ekvalizácia histogramu	22
2.2 Geometrické transformácie	25
2.3 Predspracovanie pomocou lokálnych operátorov	25
2.3.1 Diskrétna dvojrozmerná konvolúcia	26
2.3.2 Najdôležitejšie konvolučné jadrá	27
3 Výber metódy pre implementáciu na GPU.....	30
3.1 Testovanie existujúcich implementácií.....	31
4 Implementácia	37
4.1 Metriky používané pri hodnotení kvality GPU implementácie	38
4.2 Konvolúcia z pohľadu implementácie	39
4.3 Vstup a výstup.....	41
4.4 CPU implementácia	42
4.5 GPU implementácia	43
4.5.1 Konvolúcia 1:1:1	43
4.5.2 Konvolúcia 1:N:N	55
4.5.3 Konvolúcia N:M:M.....	62
5 Experimentálne zhodnotenie	68
Záver	73
Zoznam použitej literatúry	74
Zoznam príloh.....	76

Zoznam obrázkov

Obrázok 1: Usporiadanie vlákien	15
Obrázok 2: CUDA pamäťový model	16
Obrázok 3: Ilustrácia jednotlivých prípadov	21
Obrázok 4 : Obraz a jeho histogram pred ekvalizáciou	23
Obrázok 5: Obraz a jeho histogram po ekvalizácii	23
Obrázok 6: Geometrická transformácia súradníc v rovine	25
Obrázok 7: Príklad výpočtu konvolúcie	26
Obrázok 8: Vstupný obraz	28
Obrázok 9: Obraz filtrovaný priemerovacím filtrom	28
Obrázok 10 : Obraz filtrovaný horizontálnym Sobelovým filtrom	29
Obrázok 11: Obraz filtrovaný Laplaceovým filtrom	29
Obrázok 12 : Konvolúcia 1:N:N	40
Obrázok 13 : Konvolúcia N:M:M	40
Obrázok 14 : Prvý vzor pre načítanie do zdieľanej pamäte	47
Obrázok 15 : Druhý vzor pre načítanie do pamäte	49
Obrázok 16 : Testované rozmery subdlaždíc	52
Obrázok 17 : Operácie vykonávané synchrónne	57
Obrázok 18 : Operácie vykonávané asynchrónne	57
Obrázok 19 : Vizualizácia výpočtu	60
Obrázok 20 : Vizualizácia výpočtu	61
Obrázok 21: Vizualizácia výpočtu	69

Zoznam tabuliek

Tabuľka 1 Test CLAHE CPU	33
Tabuľka 2 : Test CLAHE CUDA.....	33
Tabuľka 3 : Test CLAHE OPENCL.....	33
Tabuľka 4: Test konvolúcia CPU	34
Tabuľka 5: Test konvolúcia CUDA	34
Tabuľka 6 : Test konvolúcia OPENCL	34
Tabuľka 7 :Implementácia Kernel naive	44
Tabuľka 8 : Implementácia Kernel naive improved.....	45
Tabuľka 9 : Vplyv šírky konvolučného jadra na počet aktívnych a pasívnych vlákien.....	48
Tabuľka 10 : Implementácia Kernel shared full block: Vzor 1	50
Tabuľka 11 : Implementácia Kernel shared incomplete block: Vzor 2	50
Tabuľka 12 : Implementácia Kernel shared full block: Vzor 1, rozmer subdlaždice 2x2 ..	53
Tabuľka 13: Implementácia Kernel shared full block: Vzor 1, rozmer subdlaždice 3x3 ...	53
Tabuľka 14: Kernel shared incomplete block: Vzor 2, rozmer subdlaždice 2x2	54
Tabuľka 15: Kernel shared incomplete block: Vzor 2, rozmer subdlaždice 3x3	54
Tabuľka 16: Finálne implementácie kernelov pre konvolúciu 1:1:1	55
Tabuľka 17: Implementácia Kernel shared threads.....	59
Tabuľka 18:Kernel shared managed 1:1:1	61
Tabuľka 19 : Kernel shared managed 1:N.N.....	61
Tabuľka 20: Kernel shared multi.....	65
Tabuľka 21: Kernel shared threads multi	65
Tabuľka 22: Prvá implementácia, Kernel shared multi.....	66
Tabuľka 23: Druhá implementácia, Kernel shared threads multi	66
Tabuľka 24: Porovnanie výsledkov CPU implementácie z OpenCV s implementáciou Kernel shared threads	68
Tabuľka 25: Porovnanie GPU OpenCL implementácie z OpenCV s implementáciou Kernel shared threads	70
Tabuľka 26: Porovnanie GPU OpenCL implementácie s Kernel shared threads pri rôznych rozmeroch obrazu	71

Úvod

V dobe, v ktorej žijeme, sme denne zaplavovaní kvantom dát. Nie sme iba pasívni konzumenti, ale sami generujeme nové dáta, formou dokumentov, mailov, statusov, fotografiami alebo videom. Práve dáta v obrazovej podobe sú najobjemnejšie. Disciplína, venujúca sa získavaniu informácii z obrazových dát sa nazýva počítačové videnie. Počítačové videnie sa snaží napodobniť ľudské videnie pomocou výpočtovej techniky. Základom počítačového videnia je spracovanie dvojrozmerného obrazu, na ktoré používa algoritmy rôznej úrovne abstrakcie. Tieto algoritmy sú výpočtovo náročné a obrovský dátový tok neumožňuje ich spracovanie tradičnými prostriedkami. Tu nám prichádzajú na pomoc grafické procesory.

Grafické procesory (GPU) sa vplyvom enormného dopytu trhu vyvinuli na vysoko paralelný, viacvláknový a mnohojadrový procesor s ohromnou výpočtovou silou a vysokou pamäťovou priepustnosťou. V súčasnosti neslúžia iba účelu, na ktorý boli pôvodne určené – rendering v počítačovej grafike, ale aj na spracovanie obrazu, zvuku, kódovanie a dekódovanie videa, spracovanie signálov, fyzikálne simulácie či v kryptografii. Vo všeobecnosti môžeme povedať, že GPU sú vhodné na dátovo paralelné výpočty, to jest na rôznych dátových prvkoch je vykonávaný rovnaký algoritmus, nezávisle od ostatných prvkov s vysokým počtom operácií vykonaných nad prvkom.

Využitie grafickej karty na akceleráciu algoritmov je hlavnou motiváciou pre výber témy bakalárskej práce. Naskytuje sa nám príležitosť oboznámiť sa s prostredím a programovaním na grafickej karte, osvojiť si základné koncepty a algoritmy spracovania obrazu a vytvoriť vlastné, unikátne riešenie zadaného problému, ktoré by mohlo byť dokonca kvalitnejšie ako existujúce.

Práca je organizovaná nasledovne. V prvej kapitole sa oboznámime s programovacím modelom CUDA v rozsahu potrebnom pre pochopenie implementácie. Zadefinuje základné pojmy používané pri programovaní na GPU, popíšeme pamäťovú hierarchiu a reprezentáciu čísel s pohyblivou desatinnou čiarkou na GPU.

Pokračujeme druhou kapitolou, čitateľovi v nej predstavíme vybrané metódy predspracovania obrazu, menovite ekvalizáciu histogramu adaptívnou metódou a diskrétnu dvojrozmernú konvolúciu, o ktorých predpokladáme, že sú vhodní kandidáti na implementáciu na GPU.

V tretej kapitole analyzujeme vybrané metódy z pohľadu zložitosti, vhodnosti generovania umelých dát a možnosti paralelizácie na GPU. Na základe analýzy a porovnania existujúcich implementácií vybraných metód sa rozhodneme implementovať vyššie zmienenú metódu konvolúcie.

Kapitola štyri je jadrom celej práce a obsahuje vyčerpávajúci opis procesu implementácie. Zavedieme metriky používané pri porovnávaní jednotlivých GPU implementácií a popíšeme jednotlivé druhy konvolúcie z implementačného hľadiska. Pre validáciu výsledkov sme vytvorili CPU implementácie a následne, iteračným postupom vytvárame GPU implementácie pre každý druh konvolúcie. Popri implementácii optimalizujeme aj prenos dát medzi CPU a GPU, ktorý je často jediným úzkym hrdlom GPU výpočtu.

Experimentálne zhodnotenie našej a existujúcej implementácie sa nachádza v poslednej kapitole.

1 Oboznámenie sa s programovacím modelom CUDA

Možnosť použitia grafickej karty ako akcelerátora výpočtov nám umožňujú rôzne frameworky. Trh ponúka viacero frameworkov umožňujúcich programovanie na grafickej karte. Medzi najúspešnejšie z nich patria OpenCL a CUDA (Compute Unified Device Architecture). OpenCL, na rozdiel od CUDA, je open-sorce framework. Otvorenosť tohto frameworku je pre neho výhodou aj nevýhodou. Faktom je, že OpenCL je použiteľná na grafickej karte ľubovoľného výrobcu, ale lepší výkon dosahuje na kartách od spoločnosti AMD.¹ CUDA je podporovaná iba na grafických kartách od NVIDIA, ktorá ju vytvorila. Otvorenosť OpenCL mu bráni dosahovať kvalít proprietárnej CUDA, ktorá má veľkú podporu vývojárov a je staršia ako OpenCL. S frameworkom CUDA je možné dosiahnuť vyšší výkon, preto považujeme CUDA za vhodnejší framework pre riešenie stanoveného problému. Mnoho ľudí si pod pojmom CUDA predstavuje programovací jazyk alebo API. CUDA je viac ako to. CUDA je paralelný framework a programovací model, ktorý robí využitie grafických kariet NVIDIA jednoduchým a elegantným pre všeobecné problémy. Programátor stále píše programy v jazyku C, C++ alebo Fortran, ktoré sú obohatené o pár syntaktických konštrukcií a kľúčových slov. Existujú aj nástroje pre volanie CUDA funkcií z vysokoúrovňových jazykov ako Java alebo Python ale nie je s nimi možné dosiahnuť výkonu ako v čistom C/C++, jedným z dôvodov je fakt, že sa nejedná o čisto kompilované jazyky, obsahujúce automatickú správu pamäte.[1]

1.1 Programátorský model

Program, využívajúci CUDA je zložený z častí, ktoré sú vykonávané buď na CPU (v literatúre označovaný ako host) alebo na GPU (v literatúre označovaný ako device). Vo všeobecnosti vyzerá priebeh časti programu využívajúcej výpočtový výkon GPU nasledovne:

- vyhradenie pamäte na GPU,
- presun dát z pamäte RAM do pamäte GPU,
- spustenie výpočtov na GPU,
- presun výsledkov z pamäte grafickej karty naspäť do RAM.

¹ <https://create.pro/blog/opencl-vs-cuda/>

Vďaka pár kľúčovým slovám, novej syntaxe a kompilátoru je možné jednoducho integrovať CUDA do jazyka C alebo C++. Výpočty na GPU sú spúšťané hostom zavolaním tzv. kernelu. čo je špeciálna funkcia vykonávaná každým spusteným vláknom. Kernel je každá funkcia definovaná špecifikátorom `__global__`, bez návratovej hodnoty.

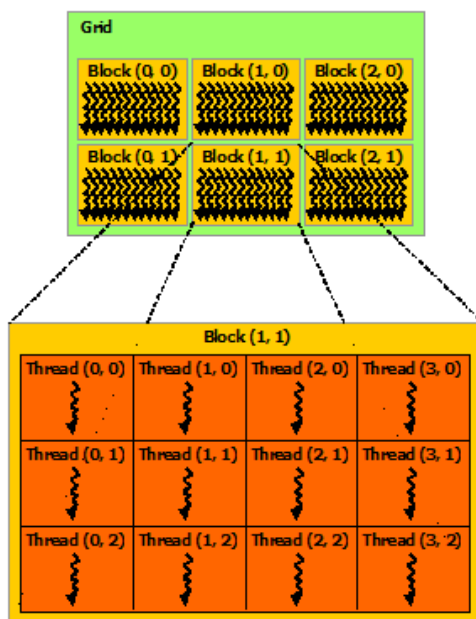
```
__global__ void kernel(parametre)
{
    telo_funkcie;
}
```

Kernel môže byť volaný z CPU aj z GPU ale je vykonávaný len na GPU. Ak chceme kernel zavolať, spustíme ho týmto spôsobom:

```
kernel <<<GS, BS, SM, ST >>>(parametre);
```

Okrem parametrov samotnej funkcie sa nachádzajú ďalšie v špicatých zátvorkách. Parameter GS predstavuje rozmer mriežky, teda počet spúšťaných blokov, BS určuje počet vlákien v bloku. SM značí počet bajtov dynamicky alokovanej zdieľanej pamäte pre blok a ST predstavuje prúd pripojený k danému volaniu. Prúdy umožňujú za určitých, pevne stanovených podmienok paralelizovať operácie s GPU. Pre ilustráciu, v 2 rôznych prúdoch prebieha výpočet a v treťom môžeme kopírovať z hosta na device alebo naopak. Parametre GS a BS sú parametre povinné, ostatné 2 sú nepovinné. Ako sme mohli postrehnúť, vlákna sú združené v hierarchii blokov a bloky do mriežok. V nasledujúcej podkapitole si priblížime túto problematiku a zmysel tohto kategorizovania.[3]

1.2 Hierarchia vlákien



Obrázok 1: Usporiadanie vlákien

Blok

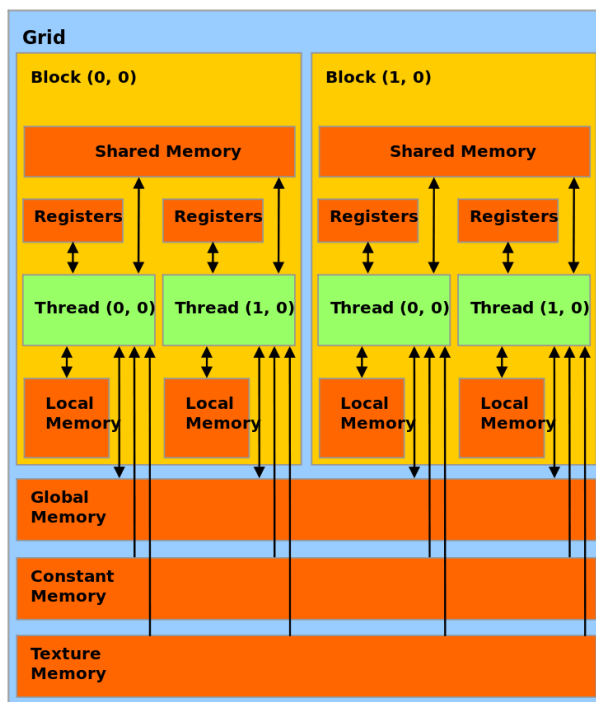
Na Obrázok 1 si môžeme všimnúť, že vlákna sú usporiadané do 1D, 2D alebo 3D blokov, kde vlákna v rámci rovnakého bloku môžu zdieľať dáta a môžu sa synchronizovať. Takýto spôsob usporiadania poskytuje prirodzený spôsob pre reprezentáciu vektorov alebo matic. Počet vlákien v jednom bloku je obmedzený výpočtovými schopnosťami zariadenia. Na súčasných GPU je počet vlákien v jednom bloku 1024. Kernel ale môže byť spustený s množstvom rovnako veľkých blokov a potom je celkový počet vlákien daný ako súčin počtu vlákien v bloku a počet blokov. Každý blok vlákien musí byť schopný pracovať nezávisle na ostatných, paralelne alebo sériovo, aby bola zabezpečená škálovateľnosť systému. Každé vlákno je v rámci bloku unikátne identifikované pomocou vstavanej štruktúry.

Mriežka

Bloky sú organizované do 1D, 2D alebo 3D mriežky. Ako vlákno, tak aj každý blok je možné unikátne identifikovať pomocou vstavanej štruktúry. Počet blokov v mriežke je zvyčajne určený rozmerom spracovávaných dát alebo počtom procesorov v GPU. [1][2]

1.3 Pamäťová hierarchia

Existuje niekoľko druhov pamätí, ku ktorým majú vlákna prístup. Každá pamäť prináša kompromisy, s ktorými treba rátať pri návrhu algoritmov. Pamäte sa líšia rôznou latenciou prístupu, veľkosťou adresného priestoru, životnosťou a rozsahom prístupu. Ku každému druhu pamäte uvedieme vybrané parametre, ak to bude možné.



Obrázok 2: CUDA pamäťový model

Registre

Prístup k registrom je veľmi rýchly, ale počet registrov je vysoko obmedzený. Zvyčajne sa do nich ukladajú premenné skalárneho typu. O rozhodnutí, či premenná bude uložená v registri rozhoduje prekladač. Ako možno vidieť na Obrázok 2, registre sú prístupné iba jednému vláknu a ich životnosť je obmedzená na životnosť vlákna.

Lokálna pamäť

Všetky premenné, ktoré sa nedostanú do registrov sú umiestnené do lokálnej pamäte. Má rovnakú latenciu ako globálna pamäť, ale môže byť uložená do cache pamäte. Z hľadiska životnosti a rozsahu prístupu zdieľa charakteristiky s registrami.

Zdieľaná pamäť

Pamäť zdieľaná blokom vlákien, má rovnakú životnosť, ako blok. Prístup k zdieľanej pamäti je veľmi rýchly, približne 10-krát rýchlejší ako prístup ku globálnej pamäti. Modifikácia zdieľanej pamäte musí byť synchronizovaná, pokiaľ nie je garantované, že každé vlákno pristupuje k pamäti, z ktorej sa nebude čítať ani zapisovať iným vláknom v bloku. S ohľadom na to, že zdieľaná pamäť je rýchlejšia ako globálna, je efektívne kopírovať globálnu pamäť do zdieľanej pri viacnásobnom prístupe ku globálnej pamäti a ušetriť značne časové prostriedky.

Globálna pamäť

Jedná sa o pamäť, ktorá poskytuje veľký adresný priestor, je zdieľaná medzi všetkými vláknami, má rovnakú životnosť ako program (pokiaľ nie je uvoľnená), ale prináša nevýhody: jej latencia je vysoká. Preto je veľká snaha zredukovať počet prístupov ku globálnej pamäti. Pamäť je viditeľná pre všetky vlákna a aj pre hosta, ktorý ju alokuje a dealokuje. Pri globálnej pamäti si musí programátor dávať pozor na prístup k nej, pretože vykonávanie vlákien je nemožné synchronizovať medzi viacerými blokmi. Jedinou cestou, ako zabezpečiť synchronizovaný prístup k tejto pamäti je rozdeliť problém do viacerých, menších kernelov a synchronizovať kernely na hostovi.

Pamäť konštánt a pamäť textúr

Určené iba na čítanie, môžu byť v cache pamäti, sú deklarované na hostovi, majú rovnakú životnosť ako program a sú viditeľné všetkým vláknam. Latencia prístupu je menšia ako ku globálnej pamäti ale väčšia ako k zdieľanej.[4][5]

1.4 Hardvérová implementácia

Architektúra NVIDIA grafických kariet je postavená na škálovateľnom poli mnohojadrových prúdových multiprocessorov. Keď CUDA program na hostovi spustí kernel, bloky mriežky sú očíslované a distribuované na multiprocessory s voľnou výpočtovou kapacitou. Vlákná bloku sú spúšťané na jednom multiprocessore súbežne, pričom na jednom multiprocessore možno spustiť viacero blokov. Multiprocessor je navrhnutý tak, aby súčasne vykonával stovky vlákien. Na spravovanie takého veľkého množstva vlákien upotrebuje unikátnu architektúru zvanú SIMT (Single-Instruction, Multiple-Thread).

Architektúra stavia multiprocessor do úlohy hlavnej vykonávacej jednotky, ktorá vytvára, spravuje, plánuje a vykonáva vlákna v skupine 32 paralelných vlákien nazývanej warp. Ak je multiprocessoru daných viac blokov na vykonávanie, rozdelí ich do warpov a plánuje sa až ich vykonávanie. Spôsob, akým je blok rozdelený do warpov je vždy rovnaký, vlákna sú zaradované do warpov vzostupne podľa ich ID. Vlákna vo warpe začínajú vykonávanie na rovnakej adrese, ale každé vlákno má vlastný počítač inštrukcii a stavové registre, čím sa vykoná nezávisle na ostatných. Najrozumnejšie je dosiahnuť stav, v ktorom vlákna vo warpe vykonávajú rovnaké inštrukcie. V závislosti od dát dochádza k podmienenému vetveniu a vlákna divergujú. V tom prípade warp vykonáva každú vzniknutú vetvu samostatne, pričom vlákna z inej vetvy musia čakať. Kontext vykonávania (registre, počítače inštrukcii) je pre každý warp uchovávaný „na čipe“ počas celej existencie warpu, tým prepínanie medzi kontextami nestojí nič a plánovač vyberá taký warp, ktorého všetky vlákna sú pripravené vykonať nasledujúcu inštrukciu.

Počet blokov a warpov sídliačich na multiprocessore pre daný kernel závisí na počte registrov a zdieľanej pamäte použitej kernelom a zdrojoch multiprocessora. Samozrejme existuje konečný, maximálny počet blokov a warpov sídliačich na multiprocessore. Tieto limity, ako aj počet registrov, veľkosť zdieľanej pamäte dostupnej pre blok, maximálny počet vlákien spustených v bloku, počet vlákien vo warpe a ďalšie technické informácie sú dané výpočtovou kategóriou zariadenia. Všetky výpočtové schopnosti zariadenia, pre vybranú výpočtovú kategóriu, sú uvedené tu², kde možno nájsť pre komparáciu detailný prehľad nielen hardvérových, ale aj softwarových schopností grafických kariet rôznych generácií.[3]

Po zavedení pojmov ako warp a globálna pamäť môžeme zaviesť pojem zhlučovanie pamäte. Globálna pamäť pre GPU je analógiou pamäte RAM pre CPU. Pristupuje sa k nej cez 32-, 64- alebo 128-bajtové pamäťové transakcie. Tieto pamäťové transakcie musia byť prirodzene zarovnané: iba 32-, 64- alebo 128-bajtové segmenty, ktorých prvá adresa je násobkom ich veľkosti, môžu byť čítané alebo modifikované v jednej transakcii. Keď warp vykonáva inštrukcie pristupujúce ku globálnej pamäti, zhlučuje pamäťové prístupy vlákien do jednej alebo viacerých transakcií, v závislosti od distribúcie pamäťových adries.

² <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities>

Ak pamäť nie je zarovnaná, počet prístupov ku globálnej pamäti sa zväčší a plytvá sa tak prenosovou šírkou zbernice pamäťovej karty ale aj časom vykonávania warpu. Ak chceme zvýšiť priepustnosť globálnej pamäte, musíme maximalizovať zhľukovanie a to nasledovne:

- dodržiavaním optimálnych prístupových vzorov k pamäti pre danú výpočtovú kategóriu,
- používať dátové typy, ktoré spĺňajú zarovnanie,
- zarovnávať dáta, napríklad pri prístupe k viacrozmerným maticiam.

Architektúra používa little-endian reprezentáciu ukladania dát. Endianita definuje poradie ukladania bajtov príslušného údajového typu. V prípade little-endian sa na pamäťové miesto s najnižšou adresou uloží najmenej významný bajt.

1.5 Reprezentácia čísel s pohyblivou desatinnou čiarkou na GPU

Je všeobecne známe, že čísla s pohyblivou desatinnou čiarkou ponúkajú reprezentáciu, ktorá dovoľuje aproximovať reálne čísla na počítači s kompromisom medzi rozsahom a presnosťou. Kódovanie a funkcionálnosť čísel s desatinnou čiarkou sú definované v IEEE 754 štandarde³, ktorý je masovo implementovaný v počítačových systémoch už od roku 1985 a riadi sa ním aj CUDA. Je dôležité zvážiť mnohé aspekty správania sa čísel s pohyblivou desatinnou čiarkou, najmä v prostredí heterogénneho prostredia (CPU + GPU), kde sú operácie prevádzané na rozličných typoch hardvéru, aby bol dosiahnutý čo najvyšší výkon s požadovanou presnosťou.

Štandard spomenutý vyššie kóduje binárne dáta s pohyblivou desatinnou čiarkou do troch polí: jednobitové pole pre znamienko, nasledované bitmi exponentu a bity pre mantisu s fixnou bázou. Bežným formátom je 32 bitový float. Ako štandard tvrdí, float zahŕňa 1 bit pre znamienko, 8 bitov pre exponent a 23 bitovú mantisu. Ďalším formátom je 64 bitový double, zahŕňa 1 bit pre znamienko, 11 bitov pre exponent a 52 pre mantisu. Aj pri float a double je báza pre exponent rovná dvom.

³ <http://ieeexplore.ieee.org/document/4610935/?reload=true>

Hlavný dôvodom, prečo je pre nás podstatné poznať reprezentáciu čísel s pohyblivou desatinnou čiarkou, je nutnosť verifikácie výsledkov medzi CPU a GPU. Je zřejmé, že výsledky nebudú rovnaké a to z nasledujúcich príčin:

- operácie s pohyblivou desatinnou čiarkou nie sú asociatívne, preusporiadanie operácií v dôsledku paralelizácie môžu zmeniť výsledok
- rôzne architektúry podporujú rôzne levely presnosti a zaokrúhľovania pri rôznych podmienkach (príznaky kompilátora ako level optimalizácie)
- každý kompilátor interpretuje štandard rôzne - nešpecifikuje, ako majú byť implementované funkcie ako sínus, kosínus a podobne, prevod čísel s pohyblivou desatinnou čiarkou do binárnej reprezentácie⁴

V praktickej časti bude poukázané na použitú metódu pre porovnávanie výsledkov medzi CPU a GPU.[12]

⁴ <https://www.appinf.com/download/FPIssues.pdf>

2 Vybrané metódy predspracovania obrazu

Spracovanie obrazu môžeme vo všeobecnosti chápať ako prevod analógového signálu na digitálny a jeho následné spracovanie. Tento proces je možné rozdeliť do viacerých etáp. Je zrejmé, že uvedené rozdelenie nie je jediné možné. Môžeme ich vyjadriť nasledujúcou schémou:

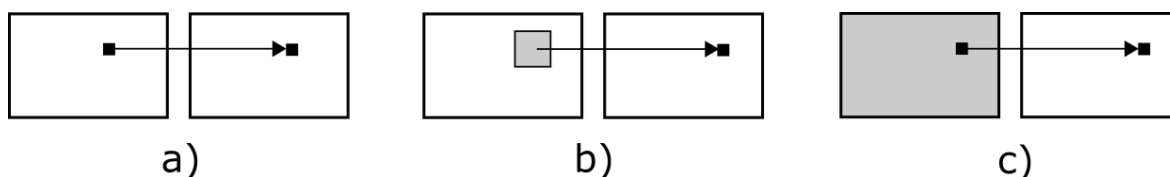
- snímanie a digitalizácia,
- predspracovanie,
- segmentácia,
- popis objektov,
- klasifikácia.

Z pohľadu tejto práce nás zaujímajú hlavne metódy predspracovania obrazu, ktorým sa budeme v nasledujúcej kapitole vo veľkej miere venovať. Pod predspracovaním obrazu môžeme rozumieť operácie s obrazom na najnižšej úrovni abstrakcie. Vstupom aj výstupom sú dáta, ktoré obsahujú veľkosť jas v každom obrazovom pixely. Cieľom predspracovania je zlepšenie obrazu z hľadiska ďalšieho spracovania. Napríklad môžeme opraviť pixel skreslený šumom na základe hodnôt jeho susedov alebo potlačiť neželané deformácie a skreslenia. Je dôležité pripomenúť, že v priebehu predspracovania nezískame žiadnu novú informáciu. Môžeme len niektorú informáciu potlačiť alebo zvýrazniť. [6][7] Metódy predspracovania je možné rozdeliť na 3 základné typy: [11]

- jasové transformácie
- geometrické transformácie
- lokálne predspracovanie (predspracovanie pomocou lokálnych operátorov).

Z hľadiska veľkosti okolia prispievajúceho k transformácii vstupného bodu na výstupný poznáme nasledovné typy operácií:

- a) bodové (výstupný bod je transformáciu hodnoty jediného vstupného bodu) ,
- b) lokálne (výstupný bod je tvorený hodnotou vstupného bodu a jeho lokálneho okolia),
- c) globálne (na hodnote výstupného bodu sa podieľajú všetky body vstupného obrazu).[9]



Obrázok 3: Ilustrácia jednotlivých prípadov

2.1 Bodové jasové transformácie

Medzi bodové transformácie patria:

- jasové korekcie,
- zmena jasovej stupnice,
- vyrovnanie histogramu.

Jasové korekcie modifikujú jas bodu, berúc do úvahy jeho hodnotu a jeho polohu v obraze. Pri zmene jasovej stupnice je určitá hodnota jasu vo vstupnom obraze transformovaná na inú výstupnú hodnotu, a to bez ohľadu na pozíciu v obraze.[7]

2.1.1 Ekvalizácia histogramu

Pred popisom ekvalizácie histogramu musíme definovať pojem histogram. Histogram jasu daného obrazu $f(x,y)$ je funkcia $H(p)$, ktorá pre každú úroveň jasu udáva počet pixlov v obraze, ktoré majú túto úroveň. Ekvalizácia histogramu sa veľmi často používa na zvýšenie kontrastu v obraze. Ekvalizáciou sa snažíme dosiahnuť ideálny histogram, ktorý obsahuje rovnaký počet z každej zastúpenej jasovej hodnoty. Matematická definícia histogramu bola prevzatá zo skrípt Milana Ftáčnika.[7]

Nech $\langle p_0, p_k \rangle$ je interval jasov vo vstupnom obraze a $H(p)$ je histogram vstupného obrazu. Cieľom je nájsť monotónnu jasovú transformáciu $q = T(p)$, aby výsledný histogram $G(q)$ bol rovnomerný pre celý výstupný interval jasov $\langle q_0, q_k \rangle$. Z požiadavky na monotónnosť zobrazenia plynie

$$\sum_{i=0}^k G(q_i) = \sum_{i=0}^k H(p_i) \quad (1)$$

Súčty v rovnici (1) môžeme chápať ako distribučnú funkciu diskrétného rozdelenia. Pre obraz s N riadkami a stĺpcami a žiadanému histogramu $G(q)$, odpovedá konštantná hustota pravdepodobnosti, ktorej hodnota g_c je

$$g_c = \frac{N^2}{q_k - q_0} \quad (2)$$

Výsledok rovnice (2) nahradí ľavú stranu rovnice (1). Rovnomerný výstupný histogram by bolo možné získať pre spojité rozdelenie, pre ktoré by rovnica (1) nadobudla tvar

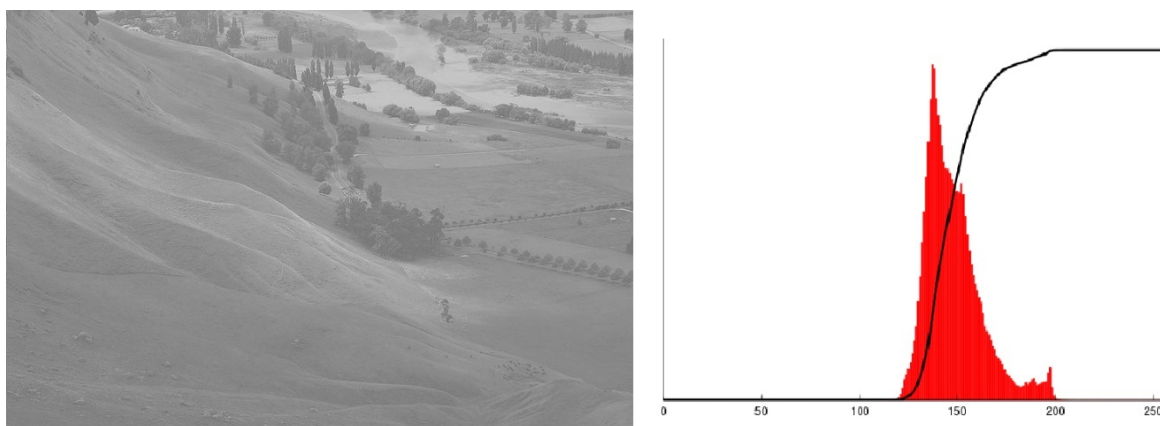
$$N^2 \int_{q_0}^q \frac{1}{q_k - q_0} ds = \frac{N^2 (q - q_0)}{q_k - q_0} = \int_{p_0}^p H(s) ds \quad (3)$$

Teraz môžeme zapísať výslednú jasovú transformáciu T

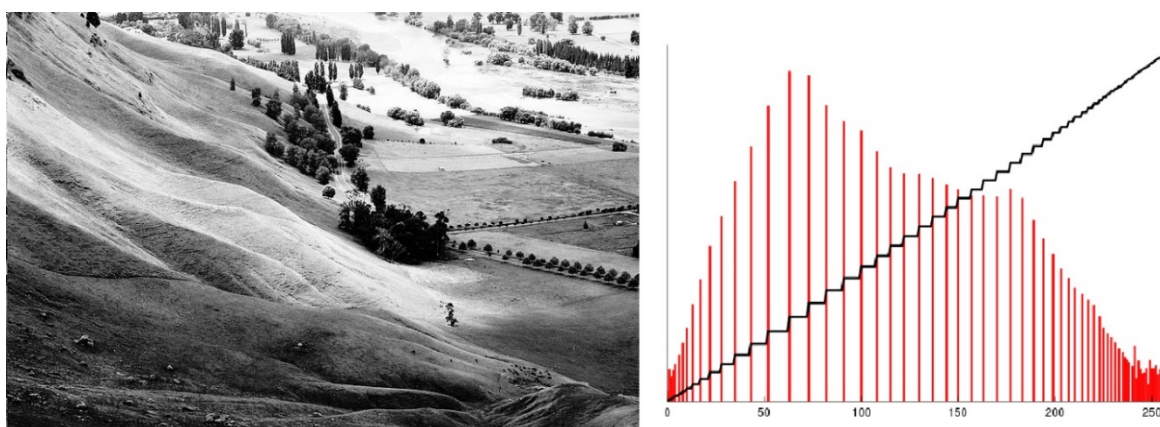
$$q = T(p) = \frac{q_k - q_0}{N^2} \int_{p_0}^p H(s) ds + q_0 \quad (4)$$

Integrál v rovnici (4) sa nazýva kumulatívny histogram. V digitálnych obrazoch sa aproximuje súčtom, a preto výsledný histogram nie je vyrovnaný ideálne. Pre digitálny obraz rovnicu (4) môžeme prepísať

$$q = T(p) = \frac{q_k - q_0}{N^2} \sum_{i=p_0}^p H(i) + q_0 \quad (5)$$



Obrázok 4 : Obraz a jeho histogram pred ekvalizáciou



Obrázok 5: Obraz a jeho histogram po ekvalizácii

Ekvalizácia histogramu adaptívnou metódou AHE(Adaptive histogram equalization)

Jedná sa o modifikáciu metódy ekvalizácie histogramu. Bežná ekvalizácia histogramu používa rovnakú transformáciu získanú z histogramu obrazu na transformovanie všetkých pixelov. Na obraz sa díva ako celok. Táto metóda funguje spoľahlivo, pokiaľ je hodnota všetkých bodov obrazu podobná v celom obraze. Avšak, ak obraz obsahuje oblasti svetlejšie alebo tmavšie ako väčšina obrazu, kontrast v týchto oblastiach nebude dostatočne zvýšený.

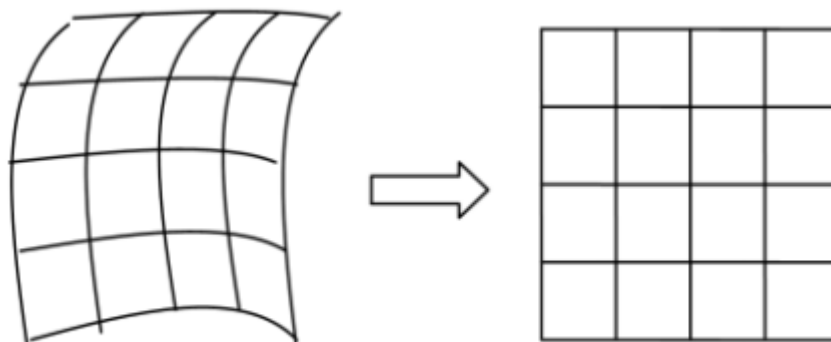
Ekvalizácia histogramu adaptívnou metódou zlepšuje túto situáciu transformáciou každého bodu s transformačnou funkciou odvodenou z blízkeho okolia bodu. V svojej najjednoduchšej forme, každý bod je transformovaný na základe histogramu okolitých bodov. Získanie transformačnej funkcie je rovnaké ako pri bežnej ekvalizácii histogramu. Body, nachádzajúce sa na okrajoch obrázka, vyžadujú zvýšenú pozornosť, pretože ich okolie neleží celé v obraze. Parametrom metódy je veľkosť blízkeho okolia bodu.

Ekvalizácia histogramu adaptívnou metódou s obmedzeným kontrastom CLAHE(Contrast limited adaptive histogram equalization)

Zvyčajne má AHE tendenciu neprimerane zosilniť kontrast v oblastiach s takmer konštantným kontrastom, lebo histogram v takýchto oblastiach je vysoko koncentrovaný. Ako výsledok môže AHE v týchto oblastiach zvýšiť šum. Ekvalizácia histogramu adaptívnou metódou s obmedzeným kontrastom je variant AHE, v ktorom je zosilnenie kontrastu obmedzené. V CLAHE, zosilnenie kontrastu v blízkosti daného bodu je dané sklonom transformačnej funkcie. Toto je úmerné sklonu kumulatívnej distribučnej funkcie a tým pádom aj hodnote histogramu v danom bode. CLAHE obmedzuje zosilnenie kontrastu orezaním histogramu vopred zadanou hodnotou pred počítaním kumulatívnej distribučnej funkcie. Dôsledkom je obmedzenie sklonu kumulatívnej distribučnej funkcie a tým aj transformačnej funkcie. Parametrom metódy je veľkosť blízkeho okolia bodu a prahová hodnota pre orezanie histogramu.[10]

2.2 Geometrické transformácie

Geometrická transformácia 2D obrazu je vektorová funkcia T , ktorá zobrazí bod (x,y) do bodu (x_0, y_0) . Môžeme si to predstaviť ako vyrovňovanie deformovanej mriežky nakreslenej na elastickej podložke. Hľadáme teda takú kombináciu pôsobiacich síl (t. j. parametrov transformačnej funkcie), aby sa kresba na elastickej podložke čo najviac podobala pravidelnej mriežke. Geometrické transformácie vypočítajú na základe súradníc bodov vo vstupnom obraze súradnice bodov vo výstupnom obraze. Nejedná sa teda o transformáciu hodnoty funkcie, ale jej súradníc. Vďaka tomu môžeme odstrániť geometrické skreslenie vzniknuté pri snímaní obrazu (napr. korekcie geometrických porúch objektívu kamery, oprava skreslenia družicového snímku spôsobená zakrivením zemegule). [6]



Obrázok 6: Geometrická transformácia súradníc v rovine

2.3 Predspracovanie pomocou lokálnych operátorov

Podstatou činnosti lokálnych operátorov je systematické prechádzanie (zvyčajne po riadkoch) celého obrazu po jednotlivých pixeloch a úprava každého pixelu na základe nejakého vzťahu závislého na okolí pixelu v jeho bezprostrednom okolí. Aplikáciou lokálnych operátorov je vyhladzovanie a detekcia hrán. Úlohou vyhladzovania je odstránenie šumu, čoho sprievodným javom je neželané rozmazanie hrán, ktoré boli na originálnom obraze ostré. Detekcia hrán je prvou fázou pri detekcii objektov v obraze. Detekcia hrán, alebo inak nazývané ostrenie obrazu, na rozdiel od vyhladzovania, zvyrazňuje vysokofrekvenčné časti spektra (hrany). Nežiadúcim sprievodným javom je aj zvýraznenie šumu. Oba navonok protichodné postupy majú rovnakú matematickú podstatu založenú na konvolúcii.

2.3.1 Diskrétna dvojrozmerná konvolúcia

Diskrétna dvojrozmerná konvolúcia je základom mnohých operácií používaných pri predspracovaní obrazu. Operácia diskkrétnej dvojrozmernej má tvar:

$$g(x, y) = \sum_{n, m \in O} h(x - m, y - n) f(m, n). \quad (6)$$

Pre skrátenejší zápis používame znak „hviezdička“

$$f(x, y) * h(x, y). \quad (7)$$

F je obrazová funkcia pôvodného obrazu, g je obrazová funkcia nového obrazu, h sa nazýva konvolučná maska, konvolučné jadro alebo filter a udáva koeficienty jednotlivých bodov v okolí O . Najčastejšie sa používajú obdĺžnikové masky s nepárnym počtom riadkov a stĺpcov (3 x 3, 5 x 5...15 x 15), pretože v tom prípade môže reprezentatívny bod ležať v strede masky a pozícia, na ktorú zapíšeme výslednú hodnotu je daná jednoznačne.

Vlastnosti konvolúcie

Komutatívnosť. Operácia konvolúcie je komutatívna, čo môžeme vyjadriť vzťahom

$$f(x) * h(x) = h(x) * f(x). \quad (8)$$

Asociatívnosť. Konvolúcia je asociatívna, čomu zodpovedá vzťah

$$(f(x) * h(x))_1 * h(x)_2 = f(x) * (h(x)_1 * h(x)_2). \quad (9)$$

Výpočet konvolúcie

Pri výpočte konvolúcie používame postup daný definíciou. Na Obrázok 7 je znázornený výpočet jedného výstupného prvku.

Vstup				Konvolučné jadro				Výstup			
	0	1	2		0	1	2		0	1	2
0	1	2	3	0	-1	-2	-3	0			
1	4	5	6	1	2	5	3	1		96	
2	7	8	9	2	1	2	4	2			

Obrázok 7: Príklad výpočtu konvolúcie

$F(1,1)$ vypočítame ako $1 * -1 + 2 * -2 + 3 * -3 + 4 * -2 + 5 * 5 + 6 * 3 + 7 * 1 + 8 * 2 + 9 * 4 = 96$. Pre jednoduchosť, každú výstupnú hodnotu počítame ako sumu súčinov hodnoty konvolučného jadra a príslušného pixelu obrazu.

Je zrejmé, že výpočet hodnôt na okraji obrazu bude vyžadovať špeciálny postup. Sú uplatňované nasledujúce techniky.

- Doplnenie chýbajúcich hodnôt nulovými hodnotami alebo konštantou.
- Zrkadlenie. Chýbajúce hodnoty sa doplnia zrkadlovo rovnaké ku hodnotám pixelov na kraji obrazu.
- Rozšírenie. Chýbajúce hodnoty sa doplnia hodnotami na okraji obrazu.
- Zmenšenie výstupného obrazu. Z programátorského hľadiska je postup najmenej vhodný, hlavne pri viacnásobnom použití konvolúcie.[13]

Zisk filtra

Pokiaľ nie je naším cieľom obraz pri konvolúcii zosvetliť alebo stmaviť, tak zisk filtra musí byť rovný jednej. Filter preto navrhujeme tak, aby súčet všetkých hodnôt konvolučného jadra bol rovný jednej.[13]

2.3.2 Najdôležitejšie konvolučné jadrá

Ako už vieme, konvolúciu môžeme použiť hlavne v prípade, ak chceme obraz vyhladiť, rozmazať hrany, odstrániť šum alebo ak chceme získať hranový obraz. Podstatný je samotný vhodný návrh konvolučného jadra. Nič nám nebráni odvodiť či navrhnúť konvolučné jadro sami, ale v tejto časti ďalej uvedieme známe, frekventovane používané filtre.

Vyhladzovacie filtre

Medzi najznámejšie vyhladzovacie filtre patria priemerovací a Gaussov filter. Priemerovací filter, ako názov napovedá, má hodnoty konvolučného jadra definované tak, aby výsledkom konvolúcie bol aritmetický priemer okolia bodu daného veľkosťou použitého konvolučného jadra. Predpokladáme, že susedné body majú ten istý, alebo veľmi podobný jas. Príklad konvolučného jadra s rozmerom 3 x 3 má nasledovný tvar:

$$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}. \quad (10)$$

Gaussov filter získame, ak v konvolučnom jadre zvýšime význam bodu uprostred, môžeme zvýšiť aj význam jeho susedov. Je odvodený od Gaussovej dvojrozmernej funkcie, ktorú nebudeme uvádzať. Gaussove filtre môžu vyzeráť nasledovne:

$$\frac{1}{10} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}. \quad (11)$$



Obrázok 8: Vstupný obraz



Obrázok 9: Obráz filterovaný priemerovacím filtrom

Hranové filtre

Medzi populárne hranové filtre patria Prewittov filter, Sobelov filter alebo Laplacian. Prewittov a Sobelov filter sa skladajú z vertikálneho a horizontálneho konvolučného jadra. Konvolučné jadrá vertikálneho a horizontálneho Prewittovho sú dané v tvare:

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}. \quad (12)$$

Hodnoty vertikálneho a horizontálneho Sobelovho filtra sú nasledujúce:

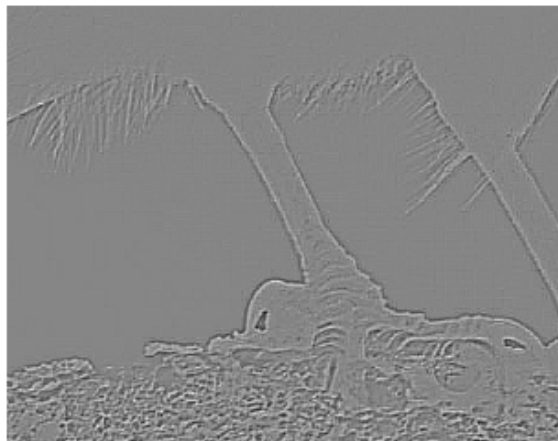
$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}. \quad (13)$$

Použitím konvolučných jadier získame príspevok zvlášť pre x -ovú a y -ovú hodnotu v danom bode. Ak chceme získať kompletný hranový obraz, použijeme nasledujúci vzťah:

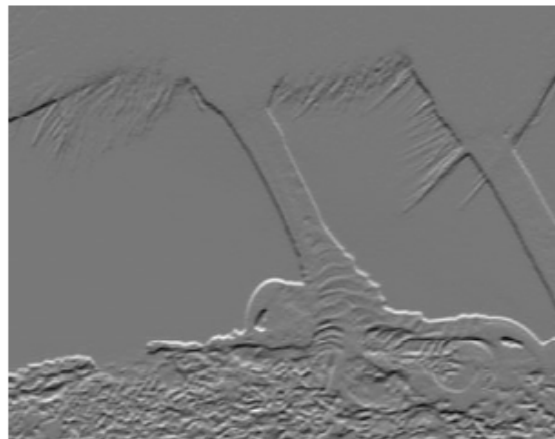
$$G = \sqrt{G_x^2 + G_y^2}. \quad (14)$$

Laplacian je často používaný pri detekcii pohybu. Laplacian je definovaný ako suma druhej diferencie v danom pixely. Definícia Laplacianu pre rozmer 3 x 3 môže nadobúdať tieto hodnoty:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}. \quad (15)$$



Obrázok 10 : Obraz filtrovaný horizontálnym Sobelovým filtrom



Obrázok 11: Obraz filtrovaný Laplaceovým filtrom

3 Výber metódy pre implementáciu na GPU

Po zoznámení sa s programátorským modelom CUDA a popise základných metód predspracovania obrazu nasleduje kapitola, v ktorej vyberieme jednu konkrétnu metódu, ktorá bude prínosná pre generovanie umelých dát z už existujúcich.

Pri vypracovaní teoretickej časti sme si vedome vybrali niektoré metódy predspracovania, ktoré sme popísali podrobnejšie ako iné, čím sme ich predbežne pokladali za kandidátov na implementáciu na GPU. Kandidáti sú nasledujúci: ekvalizácia histogramu adaptívnou metódou a diskretná dvojrozmerná konvolúcia, ktorá tvorí základ pre mnohé filtre a postupy vhodné na riešenie tejto úlohy.

Našou ďalšou úlohou je zistiť, či sú metódy vhodné pre implementáciu na GPU a ak sú, nájsť existujúce GPU implementácie a porovnať ich kvalitu s CPU implementáciami a v závere uskutočniť rozhodnutie v prospech metódy, ktorú budeme implementovať na GPU.

Analýza zložitosti

Ekvalizácia histogramu adaptívnou metódou

Vychádzajúc z matematickej definície ekvalizácie histogramu a definície CLAHE, horná asymptotická časová zložitosť CLAHE je

$$O(w * h * f^2), \quad (16)$$

kde w je šírka obrazu, h je výška obrazu a f je veľkosť blízkeho okolia bodu.

Diskretná dvojrozmerná konvolúcia

Hornú asymptotickú zložitosť konvolúcie sme určili z jej definície a je nasledovná

$$O(w * h * f^2), \quad (17)$$

kde w je šírka obrazu, h je výška obrazu a f je šírka konvolučného jadra.

Trieda zložitosti metód je pri konštantnej výške a šírke obrazu kvadratická, je dostatočne veľká na to, aby sme mohli uvažovať o implementácii na GPU. Samotná trieda zložitosti ale nie je dostatočnou informáciou a dôvodom na to, aby sme metódu implementovali na GPU. Existujú algoritmy s kvadratickou zložitou, ktoré neboli a nebudú implementované na GPU (napr. Bubble sort, Insertion sort). Ide práve o povahu

operácií vykonávaných metódami nad každým pixelom obrazu, čo ich robí vhodnými metódami pre implementáciu na GPU. Jednoducho povedané, CLAHE počíta pre každý pixel obrazu histogram blízkeho okolia pixelu, konvolúcia taktiež počíta hodnotu výstupného pixelu na základe hodnoty susedných pixelov. Z toho vyplýva, že tieto operácie možno vykonávať paralelne, nezávisle pre každý pixel obrazu. Až vlastnosť paralelizácie vykonávaných operácií hovorí o tom, že metóda je vhodná pre implementáciu na GPU, ktorá je primárne určená na riešenie problémov takéhoto druhu.

3.1 Testovanie existujúcich implementácií

Predtým, ako preskúmame dostupné, existujúce implementácie, predstavíme systém, na ktorom budú metódy spúšťané a testované.

Základné informácie o systéme:

- Procesor
 - Intel i5-7400
 - Počet fyzických jadier: 4
- RAM
 - 16GB DDR4 2400 MHz
- Disk
 - Samsung 850 Pro
- Grafická karta
 - MSI GTX 1050 GAMING X 2G
- Operačný systém
 1. Windows 10
 - Verzia CUDA
 - 8.0
 - Verzia OpenCV
 - 3.3.1

Grafická karta, NVIDIA GeForce GTX 1050, na ktorej budeme implementovať vybrané metódy a testovať existujúce, má výpočtovú kategóriu 6.1. Vymenujeme niektoré kľúčové vlastnosti, ktoré môžu ovplyvniť implementáciu:

- maximálny počet vlákien v bloku je 1024,
- veľkosť warpu je 32, maximálny počet blokov na multiprocesore je 32,
- maximálny počet registrov na blok je 65 KB
- maximálna veľkosť zdieľanej pamäte na blok je 48 KB,
- 5 multiprocesorov.

Implementácie CLAHE

Najskôr sa budeme venovať existujúcim implementáciám CLAHE. Menovite sme našli a vyskúšali implementácie CLAHE z knižnice OpenCV(Open Source Computer Vision).⁵ OpenCV je open-source knižnica zameraná na počítačové videnie v reálnom čase. Obsahuje dátové štruktúry, funkcie a algoritmy pre prácu s obrazom. Knižnica je multiplatformová a voľne dostupná pre akademické a komerčné použitie. Niektoré jej služby budeme používať aj my, ale o tom viac v kapitole 4.

Výhodou tejto knižnice je skutočnosť, že ponúka implementáciu na CUDA, OpenCL a CPU, čo nám výrazne uľahčí porovnanie a kvalitu jednotlivých implementácií. Jedinú metriku, ktorú budeme používať na porovnanie implementácií bude čas strávený výpočtom a v prípade GPU započítame aj čas nutný pre prenos dát z CPU do GPU a späť. Vybranú metriku považujeme za dostatočnú, lebo sme tieto algoritmy neimplementovali, nepoznáme implementačné detaily a zaujíma nás až ich výsledná rýchlosť z pohľadu koncového používateľa. Čas budeme merať v μ s.

V nasledujúcich riadkoch popíšeme spôsob testovania. Ako vieme z podkapitoly 2.1.1., parametrom metódy CLAHE je veľkosť blízkeho okolia bodu a prahová hodnota pre orezanie histogramu. Prahovej hodnote sme priradili konštantnú hodnotu, lebo pri experimentovaní nemala vplyv na čas trvania výpočtu. Rozmery blízkeho okolia bodu nadobúdajú variabilné hodnoty a stanovili sme ich na 7x7, 15x15 a 30x30. Počet opakovaní pre každý rozmer je 100. Pri CPU a OpenCL implementácii sa v každom opakovaní aplikuje operácia CLAHE. Pri CUDA implementácii sa okrem samotnej aplikácie CLAHE musí vstupný obraz manuálne preniesť z CPU do GPU pamäte a výstupný obraz z GPU do CPU. Zdôrazňujeme, že pri OpenCL implementácii nie sú tieto kroky nutné a prenos pamäte medzi CPU a GPU prebieha automaticky na pozadí. Budeme merať čas jedného opakovania. Vstupný obraz je čiernobiely a jeho rozmer je 1920x1200 pixelov.

Pred začiatkom merania vykonáme jeden zahrievací cyklus, aby nedošlo ku skresleniu nameraných výsledkov vplyvom inicializácie GPU a ďalších rutín spojených s knižnicou OpenCV.

⁵ <https://opencv.org/>

Výsledky testov sú znázornené v nasledujúcich tabuľkách.

Tabuľka 1 Test CLAHE CPU

CLAHE CPU				
Veľkosť blízkeho okolia bodu	Minimum	Maximum	Priemer	Smerodajná odchýlka
7x7	2830,33	5456,55	3168,5564	306,1789
15x15	2726,91	3899,04	2965,8441	183,2774
30x30	2807,12	4607,99	3058,5432	245,0161

Tabuľka 2 : Test CLAHE CUDA

CLAHE CUDA				
Veľkosť blízkeho okolia bodu	Minimum	Maximum	Priemer	Smerodajná odchýlka
7x7	2106,71	3048,44	2416,0275	196,2606
15x15	1590,27	2598,91	1911,245	190,6510
30x30	1573,2	2502,31	1853,7814	204,4632

Tabuľka 3 : Test CLAHE OPENCL

CLAHE OPENCL				
Veľkosť blízkeho okolia bodu	Minimum	Maximum	Priemer	Smerodajná odchýlka
7x7	1972,91	3501,74	2460,961	175,5413
15x15	1881,77	3516,41	2163,035	249,2525
30x30	1875,63	2879,83	2300,554	123,6628

Implementácie konvolúcie

Podobne ako pri CLAHE, tak aj pri konvolúcii vyskúšame jej implementácie z knižnice OpenCV. OpenCV ponúka implementáciu na CPU, CUDA a OpenCL. Budeme používať rovnakú metriku ako v prípade CLAHE. V ďalších riadkoch popíšeme spôsob testovania. V najjednoduchšom prípade je parametrom konvolúcie rozmer konvolučného jadra a hodnoty konvolučného jadra. Hodnoty konvolučného jadra nemajú vplyv na výpočtový čas, operácie násobenia a sčítania musia byť vykonané nezávisle od hodnôt. Experimentovali sme s konvolučnými jadrami rôznych rozmerov: 3x3, 5x5, 7x7, 9x9, 11x11, 13x13 a 15x15, aj keď najpoužívanejšie rozmery pri predspracovaní obrazu sú do rozmeru 9x9. Počet opakovaní pre každý rozmer je 100. Pri CPU a OpenCL implementácii sa v každom opakovaní aplikuje operácia konvolúcie. Pri CUDA implementácii sa okrem výpočtu musí uskutočniť prenos medzi CPU a GPU pamäťou. Budeme merať čas jedného opakovania. Vstupný obraz je čiernobiely a jeho rozmer je 1920x1200 pixelov. Vykonáme zahrievanie podobne ako pri CLAHE.

Výsledky testov sú znázornené v nasledujúcich tabuľkách.

Tabuľka 4: Test konvolúcia CPU

Šírka konvolučného jadra	Minimum	Maximum	Priemer	Smerodajná odchýlka
3	2518,01	3577,17	2661,9321	160,1891
5	4718,58	5443,58	4911,2948	159,1725
7	8699,89	10584,7	9014,2955	254,7737
9	28964,8	31269,5	29509,782	412,4454
11	42813,7	45673,4	43440,021	501,0113
13	19222,2	21076,6	19707,687	375,5194
15	19275,1	22225,9	19791,397	446,4738

Tabuľka 5: Test konvolúcia CUDA

Šírka konvolučného jadra	Minimum	Maximum	Priemer	Smerodajná odchýlka
3	13162,8	16205,8	14292,959	558,1127
5	13110,9	18077	14415,887	622,0710
7	13118,1	17580,3	14425,279	749,2622
9	13052,9	24664,7	14734,26	1560,5142
11	12928,7	18215,2	14702,488	917,4485
13	12445,3	20966,4	14692,084	1533,8977
15	12719,1	26624	15616,549	2418,5751

Tabuľka 6 : Test konvolúcia OPENCL

Šírka konvolučného jadra	Minimum	Maximum	Priemer	Smerodajná odchýlka
3	2093,4	2486,95	2254,502	85,12939
5	2222,42	5327,19	2509,429	318,898
7	2468,18	5520,38	2835,949	467,2745
9	2629,63	5690,71	3089,065	308,1
11	3068,59	6466,22	3447,724	350,6992
13	3336,53	6643,03	3829,827	318,4742
15	4025	7629,48	4340,307	360,3323

Zhodnotenie testov

CLAHE

Testovali sme 3 rôzne implementácie CLAHE: CPU, CUDA a OpenCL. Ukázalo sa, že veľkosť blízkeho okolia bodu nemá významný vplyv na výpočtový čas. Preto prejdeme k porovnaniu medzi samotnými implementáciami. Rozdiel medzi CPU a GPU implementáciou nie je až taký výrazný, ako by sme mohli očakávať. Dávame tomu za príčinu to, že aj keď je výpočet na grafickej karte rýchlejší ako na CPU, musí dochádzať k prenosu pamäte medzi CPU a GPU pamäťou, čo výrazne ovplyvní výpočet. V prípade CUDA implementácie sme odmerali čas potrebný pre prenos dát a priemerne činí 700 μ s, 30-40%

času sa minie iba na prenos dát. Veľkosť prenášaných dát je 2,19 MB, čo znamená, že každý pixel zaberá 1 B. Ďalším dôvodom, prečo je CPU implementácia taká efektívna, je aj použitie viacerých vlákien pri výpočte. Implementácia CUDA a OpenCL sú si podobné, sú približne 3-násobne rýchlejšie ako implementácia na CPU.

Konvolúcia

Experimentovali sme s implementáciami CPU, CUDA a OpenCL. Zistili sme, že šírka konvolučného jadra má výrazný vplyv na výpočtový čas. Najviac to bolo zreteľné pri CPU implementácii, kde bol 16-násobný rozdiel medzi konvolučným jadrom šírky 3x3 a 11x11. Odpoveďou na to, prečo je CPU implementácia pre väčšie konvolučné jadrá (13x13 a 15x15) rýchlejšia ako pre menšie (11x11) je, že pre väčšie jadrá sa konvolúcia počíta pomocou rýchlej Fourierovej transformácie. Nečakaná je vysoká neefektívnosť CUDA implementácie. Ak by sme aj odpočítali čas potrebný pre prenos pamäte, čo je priemerne 7000 μ s, CUDA stále zaostáva nielen za OpenCL ale aj CPU implementáciou pre malé konvolučné jadrá (3x3, 5x5, 7x7), ktoré sa často používajú a vôbec sa nedokáže vyrovnáť OpenCL implementácii. Najlepšou implementáciou konvolúcie, ktorú sme testovali, je OpenCL. Pri malých konvolučných jadrách je 2-násobne rýchlejšia oproti CPU implementácii a pri väčších aj 5-násobne.

Výber metódy, ktorú budeme implementovať

Na začiatku kapitoly nebolo úplne jasné, ktorú metódu budeme implementovať. Po vykonaní testov sa ukázalo, že CPU implementácia CLAHE je vysoko optimalizovaná a nie je priepastný výkonnostný rozdiel medzi CPU a GPU implementáciami. Preto vidíme v prípade vytvorenia vlastnej GPU implementácie priestor na výkonnostné zlepšenie. Ak sa ale dívame na CLAHE ako metódu pre generovanie umelých dát a porovnáme ju s konvolúciou, konvolúcia je o niečo vhodnejší kandidát pre generovanie umelých dát, lebo existuje viacero typov použití konvolúcie a poskytuje viac možností pri generovaní umelých dát ako CLAHE, vďaka rôznym konvolučným jadrám, ktoré sú parametrom tejto metódy. Zostal nám jediný kandidát - konvolúcia.

Po tejto úvahe sme si položili otázku, či má vôbec zmysel konvolúciu implementovať na GPU, vzhľadom na kvalitnú OpenCL implementáciu. Domnievame sa ale, že konvolúcia je metóda, ktorá je frekventovane používaná pri predspracovaní obrazu, zároveň je vhodná na generovanie umelých dát a pre svoju jednoduchosť je dobrým začiatkom pre zoznámenie sa s programovaním a optimalizáciou kódu na GPU. V neposlednom rade, vidíme potenciál

pre zrýchlenie výpočtu konvolúcie v porovnaní s existujúcou CUDA implementáciou a lepšiu správu pamäte. Všetky tieto dôvody nás priviedli k rozhodnutiu, že metóda, ktorú budeme implementovať na GPU, bude konvolúcia.

4 Implementácia

V nasledujúcej kapitole sa budeme venovať podrobnému popisu implementácie a systému, ktorý vznikne ako jej výsledok. Na začiatku si zrekapitulujeme zadanie a cieľ práce a exaktne špecifikujeme požiadavky, ktoré sú na systém kladené. Neskôr popíšeme nástroje, ktoré sme pri implementácii použili, metriky, ktorými budeme hodnotiť kvalitu GPU implementácií. Pozrieme sa na konvolúciu bližšie, z pohľadu typov implementácie a popíšeme jednotlivé kroky implementácie, popri ktorých vysvetlíme návrh systému. Ako prvú vytvoríme CPU implementáciu, ktorá nám bude slúžiť ako referenčná implementácia pre porovnávanie správnosti výsledkov s GPU implementáciami. Iteračným postupom budeme vytvárať lepšie a efektívnejšie GPU implementácie, až sa dostaneme do bodu, v ktorom nebudeme vedieť vylepšiť aktuálnu implementáciu. Priebežne budeme vytvárať a dopĺňať pravidlá a údajové štruktúry pre efektívnu správu pamäte a posledným krokom implementácie bude paralelizácia prípravy dát a post-processingu. V závere kapitoly spravíme experimentálne zhodnotenie s existujúcimi implementáciami.

Cieľom práce, ako bolo veľakrát zdôraznené, je využitie GPU paralelizovaných výpočtov na generovanie umelých dát. Podmienkou, ktorá bola nastavená už pri zadávaní práce, je použitie platformy CUDA pre vytvorenie GPU implementácie, ktorá má urýchliť generovanie umelých dát. Z toho vyplývajú aj požiadavky, ktoré sú kladené na vyvíjaný systém. Systém musí vedieť načítať existujúce údaje, v našom prípade sekvenciu obrazov, nad každým obrazom vykonať vybranú metódu s požadovanými parametrami a v prípade potreby výsledok uložiť na disk. Rozhodli sme sa implementovať metódu diskkrétnej dvojrozmernej konvolúcie, ktorá bola popísaná v kapitole 2.3.1. a jej výber zdôvodnený v kapitole 3. Vstupom pre konvolúciu bude čiernobiely obraz a konvolučné jadro, ktorého rozmery môžu byť 1x1, 3x3, 5x5, 7x7, 9x9, 11x11, 13x13 a 15x15. Rozmer konvolučného jadra je obmedzený na vymenované rozmery z dôvodu, že sa jedná o najpoužívanějšíe konvolučné jadrá.

Pri implementácii sme použili jazyk C++. Dôvodov, pre výber tohto jazyka je viacero. Mohli sme použiť C, ale C++ ponúka objektovo orientované programovanie a generické programovanie a zdrojové kódy CUDA sú kompilované kompilátorom nvcc, ktorý dodržiava štandard C++. Na výber bol aj FORTRAN, ktorý je zastaraný a nepoznáme ho. Mohli sme použiť aj Java a Python, ktoré obaľujú CUDA volania do C++ kódu, ale ich problémom je automatická správa pamäte. Pre načítanie vstupných dát a ukladanie

výstupných dát používame knižnicu OpenCV. Aby sme mohli používať CUDA, museli sme si nainštalovať CUDA Toolkit⁶. Toolkit obsahuje run-time-knižnice, nástroje na debugovanie a optimalizáciu kódu, vyššie spomenutý kompilátor nvcc a ukázkové zdrojové kódy. Používame vývojové prostredie Microsoft Visual Studio 2015, ktoré vie integrovať NVIDIA Nsight⁷, pokročilý nástroj pre debugovanie a profilovanie CUDA.

4.1 Metriky používané pri hodnotení kvality GPU implementácie

Pri porovnávaní vlastných GPU implementácií nám nebude stačiť iba celkový čas výpočtu, ako tomu bolo doteraz, ale sme nútení zaviesť ďalšie metriky, ktoré budú detailnejšie. Celkový čas výpočtu začína okamihom, kedy používateľ zavolá funkciu pre výpočet konvolúcie s danými parametrami a končí, keď dostane výsledok. Do tohto času je zahrnuté aj kopírovanie vstupného obrazu z CPU do GPU, výpočet na GPU a následné kopírovanie výsledkov späť do GPU v prípade serializovanej verzie. Pri porovnávaní implementácií z pohľadu doby výpočtu nás ale nezaujíma prenos dát medzi CPU a GPU, ale iba doba výpočtu konvolúcie. Druhou metrikou preto bude čas strávený výpočtom na GPU, inak povedané, čas trvania kernelu. Ďalším údajom, ktorý môžeme získať je pamäťová priepustnosť, ktorá hovorí o rýchlosti prenosu dát. Vypočítajme teoretickú maximálnu priepustnosť, ktorú môžeme dosiahnuť na našej GPU, na základe hardvérovej špecifikácie podľa vzorca:

$$BW_T = \frac{MCR * 10^6 * \left(\frac{MIW}{8}\right) * 2}{10^9} GB/s. \quad (18)$$

MCR je frekvencia pamäte v MHz, a MIW je šírka zbernice v bitoch. Frekvencia pamäte našej grafickej karty je 1354 MHz a šírka zbernice 128 b. Po dosadení do vzorca dostávame hodnotu 43,328 GB/s, čo je horná hranica pamäťovej priepustnosti, ktorú by sme mali dosiahnuť. Skutočnú, alebo efektívnu pamäťovú priepustnosť, ktorú dosahuje kernel, vypočítame nasledovne:

$$BW_E = \frac{R_b + W_b}{t * 10^9} GB/s. \quad (19)$$

⁶ <https://developer.nvidia.com/cuda-toolkit>

⁷ <http://www.nvidia.com/object/nsight.html>

R_b je počet bajtov, ktoré kernel prečíta z globálnej pamäte a W_b je počet bajtov, ktoré kernel zapíše do globálnej pamäte a t je čas trvania kernelu v sekundách. Reálne sme dosiahli vyššiu efektívnu pamäťovú priepustnosť ako teoretickú.

Poslednou metrikou bude výpočtová priepustnosť. Bežne používanou jednotkou merania výpočtovej priepustnosti je GFLOP/s, čo znamená „Giga-Floating-point OPeration per second“, v preklade počet operácií s pohyblivou desatinnou čiarkou za sekundu. Efektívnu výpočtovú priepustnosť vypočítame vzt'ahom:

$$GFLOP/s_e = \frac{\text{Počet operácií}}{\text{trvanie kernelu v s} \cdot 10^9} GFLOP/s. \quad (20)$$

Pre meranie času trvania kernelu, efektívnej pamäťovej priepustnosti a efektívnej výpočtovej priepustnosti použijeme nástroj NVIDIA Nsight. Reporty z neho budú súčasťou prílohy. Smerodajnou metrikou bude dĺžka trvania kernelu. Bude snahou túto metriku čo najviac zlepšiť.[14]

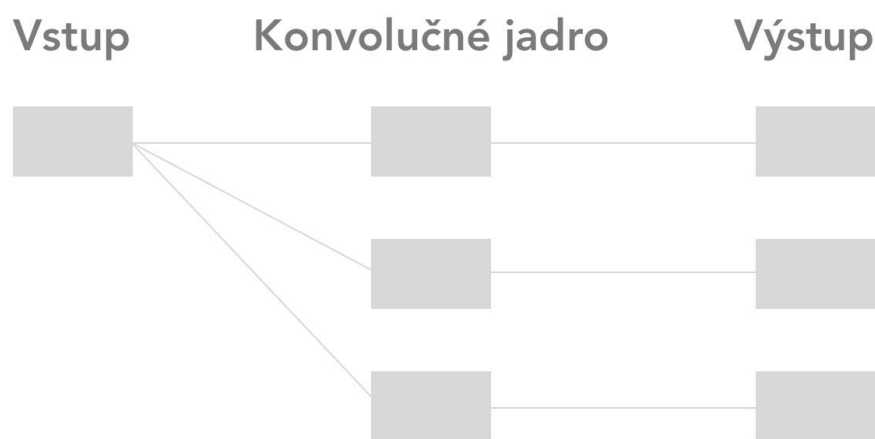
4.2 Konvolúcia z pohľadu implementácie

Doposiaľ sme sa na konvolúciu pozerali iba z teoretického hľadiska. Konvolúciu sme brali ako metódu, ktorej vstupom je jeden obraz, jedno konvolučné jadro a výstupom jeden obraz. Pri reálnych aplikáciách konvolúcie máme zriedkavo iba jeden obraz a jedno konvolučné jadro. Počet vstupných obrazov, konvolučných jadier a požadovaných výstupov býva premenlivý. Z pohľadu implementácie môžeme jednotlivé prípady od seba oddeliť a riešiť ako samostatný problém. Dôvodom pre oddelenie je získanie priestoru pre optimalizáciu.

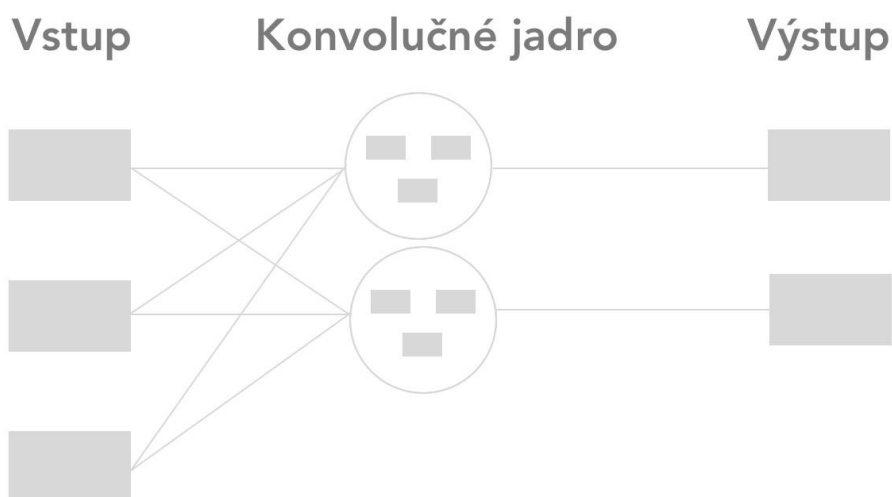
Prípad, pri ktorom máme 1 vstup, jedno konvolučné jadro a jeden výstup je príkladom konvolúcie 1:1:1 – jeden vstup, jedno konvolučné jadro, jeden výstup. Tento prípad je špeciálnym prípadom konvolúcie 1:N:N – jeden vstup, N konvolučných jadier, N výstupov, to znamená pre každé konvolučné jadro vznikne výstup z pôvodného obrazu. Tento prípad je ilustrovaný na Obrázok 11. Komplikovanejšou formou konvolúcie je prípad, v ktorom máme N vstupov, M skupín konvolučných jadier veľkosti N (jedno unikátne konvolučné jadro pre jeden vstup a pre každú skupinu jeden výstup), dokopy dostaneme M výstupov. Musí platiť, že všetky obrazy majú rovnaký rozmer a konvolučné jadrá v jednej skupine majú tiež rovnaký rozmer. Výstup pre jednu skupinu konvolučných jadier sa získa ako súčet výsledkov konvolúcie N vstupov s N konvolučnými jadrami z danej skupiny.

Uvedený prípad konvolúcie sa používa pri konvolučných výpočtoch, nazvime ho N:M:M konvolúcia. Je zobrazený na Obrázok 12.

Kroky GPU implementácie budú najskôr ukázané na konvolúcii 1:1:1. Z pohľadu používateľa sa bude jednať o konvolúciu 1:N:N, kernel avšak bude počítať pri jednom spustení iba jeden výstup. Neskôr budeme implementovať konvolúciu 1:N:N a N:M:M, pri ktorých budeme vychádzať hlavne z kernelu pre konvolúciu 1:1:1. Tieto konvolúcie budú pri jednom spustení kernelu reálne počítať viacero výstupov. Zároveň predpokladáme, že na vstupe máme k dispozícii vstupný obraz a konvolučné jadro/jadrá uložené v CPU pamäti.



Obrázok 12 : Konvolúcia 1:N:N



Obrázok 13 : Konvolúcia N:M:M

4.3 Vstup a výstup

Ako bolo zmienené, na vstup a výstup budeme používať knižnicu OpenCV. Knižnica OpenCV obsahuje funkcie pre načítanie a uloženie obrazu a údajovú štruktúru pre uloženie obrazu a informácií o ňom. My sme tieto funkcie a štruktúry zaobalili do triedy *ImageFactory*.

Vieme, že obraz je zložený z pixelov a môže byť čiernobiely alebo farebný. Pri čiernobielym obraze jeden pixel zaberá 1B pamäte- nadobúda hodnoty z intervalu $\langle 0, 255 \rangle$, čo zodpovedá údajovému typu unsigned char. Pixel čiernej farby sa uloží ako 0 a pixel bielej farby ako 255. Pixel farebného obrazu zaberie pamäťové miesto rovné veľkosti počtu farebných kanálov, kde jeden farebný kanál zaberá 1B a nadobúda hodnoty z intervalu $\langle 0, 255 \rangle$, rovnako ako pri čiernobielym obraze. Štandardom je používať 3 farebné kanály- tzv. RGB farebný model. Naša implementácia konvolúcie bude na vstupe očakávať jeden vstupný kanál. Ak by sme chceli aplikovať konvolúciu na farebný obraz, museli by sme ho najskôr rozdeliť na jednotlivé kanály, nad každým kanálom vykonať konvolúciu a výsledky z nej potom spojiť do jedného, výstupného obrazu. My ale nebudeme počítat konvolúciu s farebnými obrázkami. Ak by sme chceli počítat konvolúciu s farebnými obrázkami, môžeme výpočet riešiť ako špeciálny prípad konvolúcie $N:M:M$, kde $N = 3$ a $M = 1$. Systém bude vedieť načítať čiernobiely obraz a aj farebný, ktorý skonvertuje na čiernobiely, ktorý má jeden kanál.

Ďalšia otázkou bolo, aký údajový typ budeme používať pri výpočte konvolúcie alebo akého typu bude vstupný a výstupný obraz. Hodnoty konvolučného jadra môžu byť reálne čísla, preto sme museli pre vstup a výstup použiť údajový typ s pohyblivou desatinnou čiarkou – float. Vstupné pixely musíme previesť z unsigned char na float. Ak chceme výstupný obraz uložiť na disk, musíme spraviť opačný prevod, z float na unsigned char. Problémom je, že float nadobúda hodnoty mimo rozsahu unsigned char. Záporné hodnoty pixelov orežeme na 0 a vykonáme takzvanú normalizáciu podľa vzorca:

$$\text{výstupný pixel} = \frac{\text{vstupný pixel} - \text{minimum z rozsahu}}{\text{maximum z rozsahu} - \text{minimum z rozsahu}} * (255). \quad (21)$$

Obraz po normalizácii vieme uložiť na disk. Systém umožní ukladanie výstupov (pred normalizáciou) na disk vo forme textového súboru. Pri porovnávaní implementácií budeme používať ako vstup čiernobiely obraz o rozmeroch 1920x1200.

4.4 CPU implementácia

CPU implementácia nám bude slúžiť ako referenčná implementácia pre porovnanie správnosti výsledkov s GPU implementáciami. Mohli sme použiť implementáciu tretej strany, ale rozhodli sme sa implementovať vlastnú verziu z viacerých dôvodov. Prvým je riešenie výpočtu hraničných pixelov obrazu, sami si presne určíme, ako sa budú počítať. Druhým je dôvod, že implementácie tretích strán používajú iné algoritmy pre výpočet konvolúcie, ako ten, ktorý budeme používať my. Musíme poznamenať, že naivná implementácia konvolúcie na CPU je triviálna záležitosť. Pseudokód vyzerá nasledovne:

```

pre všetky riadky obrazu row:
  pre všetky stĺpce obrazu column:
    výsledok = 0
    pre všetky riadky konvolučného jadra y:
      pre všetky stĺpce konvolučného jadra x:
        výsledok += konvolučné_jadro[y][x] * vstup[row + y][column + x]
    výstup[row][column] = výsledok.

```

Vynechali sme kód pre obsluhu hraničných pixelov. Musíme vytvoriť dve implementácie, jednu pre konvolúciu 1:N:N a druhú pre konvolúciu N:M:M. Implementácie sa principiálne nelíšia výpočtom konvolúcie, ale majú rôzne obsluhu vstupov a výstupov.

V kapitole 3.3.1. sme si vymenovali niekoľko spôsobov, akými možno riešiť výpočet hraničných bodov. My budeme používať dva spôsoby, prvý bude rozšírenie – chýbajúce hodnoty sa doplnia hodnotami na okraji obrazu a druhý bude používať zmenšenie výstupného obrazu alebo orezanie. V druhom prípade sa obraz šírky W a výšky H , po konvolúcii s konvolučným jadrom šírky F zmenší na rozmer $W - F + 1 \times H - F + 1$. Je zrejmé, že prvý spôsob výpočtu hraničných bodov je vhodnejší, lebo obraz si zachová svoj pôvodný rozmer, ale opodstatnenie dvoch rôznych spôsobov bude vysvetlené v kapitole 4.2.3.

V teoretickej časti sme sa venovali reprezentácii čísel s pohyblivou desatinnou čiarkou na GPU. Potrebujeme overiť správnosť výsledkov medzi CPU a GPU implementáciou. Výsledok z GPU budeme považovať za správny, ak platí

$$|CPU \text{ hodnota} - GPU \text{ hodnota}| < \varepsilon, \quad (21)$$

kde ε je veľmi malé číslo, napríklad 0,001. ε bude môcť pri porovnávaní výsledkov zadávať používateľ.

4.5 GPU implementácia

4.5.1 Konvolúcia 1:1:1

Naivná implementácia

Po tom, ako sme napísali a otestovali CPU implementácie, mohli sme napísať prvú GPU implementáciu. Naším cieľom nebude preniesť CPU kód na GPU v pomere 1:1, ale zachovanie algoritmu ako takého a dosiahnutie správnych výsledkov. Pri tejto implementácii budeme vytvárať taký kernel, pri ktorom budeme programovať bez toho, aby sme mali nejaké predchádzajúce znalosti a skúsenosti s CUDA. V čase, keď sa programovala táto implementácia, tomu aj tak bolo. Preto si dovoľíme túto implementáciu nazvať naivnou. Bude slúžiť ako odrazový mostík pre zlepšovanie. Zároveň odpovie na otázku, či má zmysel optimalizovať a vylepšovať kernely. Pripomenieme, že sa jedná o implementáciu konvolúcie 1:1:1.

Prihliadajúc na to, že sme sa presunuli s prostredia CPU na GPU, nemôžeme s GPU pristupovať ku CPU adresnému priestoru priamo. Aby sme mali v GPU vstupný obraz, musíme preň alokovať priestor na GPU v globálnej pamäti a do tejto pamäte ho z CPU pamäte skopírovať. Rovnako aj s výstupom. Alokujeme ho a po výpočte ho skopírujeme z GPU pamäte do CPU. Ako sme napísali, pre vstup aj výstup budeme používať údajový typ float. Veľkosť pamäte, ktorú bude treba alokovať na GPU je $2 * \text{sizeof}(\text{float}) * \text{rozmer obrazu}$ a veľkosť pamäte, čo sa bude kopírovať smerom z CPU do GPU a z GPU do CPU je $\text{sizeof}(\text{float}) * \text{rozmer obrazu}$. Alokovanie pamäte a kopírovanie sú operácie veľmi drahé na čas, ale existujú techniky, ako tento čas zmenšiť. Týmto technikám sa budeme venovať neskôr. Zatiaľ sa snažíme optimalizovať kernel a alokácia a kopírovanie pamäte nemajú na dĺžku vykonávania kernelu žiadny vplyv. Ako so vstupom aj výstupom, aj pre konvolučné jadro musíme alokovať pamäť a nakopírovať hodnoty konvolučného jadra do GPU. V tejto verzii bude pre výpočet krajných bodov použité rozšírenie. Pre predstavu o práci kernelu uvedieme a popíšeme jeho pseudokód:

```
pozicia.x = vypocitaj absolutnu x poziciu v obraze
pozicia.y = vypocitaj absolutnu y poziciu v obraze
ak (pozicia.x >= pocet stlpcov || pozicia.y >= pocet riadkov)
    skonci
index1D = vypocitaj poziciu vo vstupnom poli
vysledok = 0
pre všetky riadky konvolučného jadra y :
    pre všetky stĺpce konvolučného jadra x :
        vstupx = vypočítaj x suradnicu pixelu a zaisti, aby patrila do rozsahu obrazu
        vstupy = vypočítaj y suradnicu pixelu a zaisti, aby patrila do rozsahu obrazu
        vysledok += konvolučné_jadro[y][x] + vstup[vstupx][vstupy]
výstup[index1D] = vysledok.
```

Teraz môžeme popísať prácu kernelu. Najskôr sa vypočíta absolútna pozícia obrazu (x-ová a y-ová súradnica v obraze). Ak táto pozícia nepatrí do obrazu, kernel končí. Ďalej sa vypočíta 1D pozícia na preindexovanie sa v jednorozmernom vstupnom poli. Potom prechádzame konvolučné jadro. Pri výpočte jednej hodnoty, ktorá sa pripočíta do celkového súčtu, vypočítame absolútnu pozíciu v obraze a zaistíme, aby táto pozícia nebola mimo rozsahu obrazu (menšia ako 0 a väčšia ako niektorý rozmer obrazu). Kernel nakoniec výsledok uloží do výstupného poľa.

Počet prístupov jedného vlákna kernelu ku globálnej pamäti je rovný $2 * \text{šírka filtra}^2 + 1$. V Tabuľka 7 :Implementácia Kernel naive je vidieť výsledky kernelu. Názov kernelu odráža názov zodpovedajúcej triedy v aplikácii.

Tabuľka 7 :Implementácia Kernel naive

Šírka filtra	Celkový čas(μ s)	Čas trvania kernelu(μ s)	Pamäťová priepustnosť(GB/s)	Výpočtová priepustnosť(GFLOP/s)
1	7214,08	442,464	43,65	10,41
3	10682,6	2489,504	52,79	16,66
5	15403	6060,864	59,68	19,01
7	22642,1	11996,704	59,39	18,82
9	32447,6	19856,256	59,57	18,8
11	38533,8	29732,032	59,63	18,75
13	48629,9	41833,888	59,34	18,62
15	62037,3	55095,296	60,1	19,82

Už pri takejto jednoduchšej GPU implementácii sa nám podarilo, až na konvolučné jadro rozmeru 1x1, urýchliť konvolúciu pomocou GPU v porovnaní s naivnou CPU implementáciou. Je zrejmé, že táto verzia má veľa nedostatkov. Za hlavný nedostatok považujeme častý prístup ku globálnej pamäti. Tento problém sa budeme snažiť v ďalšej iterácii čiastočne odstrániť.

Vylepšenie naivnej implementácie

Problémom naivnej implementácie bol častý prístup ku globálnej pamäti. Počet prístupov vieme ale rapídne zmenšiť, ak umiestnime konvolučné jadro do inej pamäte. Dôvodom je, že každé vlákno kernelu prístupuje k rovnakým hodnotám konvolučného jadra. Vyskúšali sme konvolučné jadro umiestniť do zdieľanej pamäte a pamäte konštánt. S prvou menovanou pamäťou sme nedosiahli žiadneho výrazného zrýchlenia, odôvodňujeme to nutnosťou synchronizácie v rámci bloku a divergenciou vlákien, ktoré nenačítavali hodnoty do filtra. Umiestnenie filtra do pamäte konštánt malo výrazný dopad na výkon, hlavne pri

väčších konvolučných jadrách. Počet prístupov do globálnej pamäte sa týmto zmenšil dvojnásobne. Výhodou pamäte konštánt je, že je cachovaná a umiestnená v L2 pamäti. Konvolučné jadro musí byť nakopírované do pamäte konštánt ešte pred vykonaním kernelu, ale to isté sa muselo vykonať aj v naivnej implementácii, kde konvolučné jadro bolo nakopírované do globálnej pamäte.

Použitie pamäte konštánt pre konvolučné jadro nebolo jediným vylepšením, ktoré sme vykonali. Zmenili sme taktiež aj funkcie min a max, použili sme verzie zo štandardnej knižnice, ktoré sa líšia najviac v tom, oproti pôvodným, že sú implementované ako šablóny a sú inline (nedochádza k volaniu funkcie, ale kód funkcie je priamo vložený na miesto, odkiaľ sa funkcia volá). Poslednou úpravou bolo použitie šablóny pre parameter šírka konvolučného jadra. Takto sa z parametru šírka konvolučného jadra stane konštanta a môžeme použiť direktívu prekladača `#pragma unroll`, ktorá „rozbalí“ cyklus pri prechádzaní konvolučného jadra. Kompilátor pozná počet cyklov už v čase prekladu a ušetríme inštrukcie pre inkrementáciu a porovnanie oproti cyklom vyhodnocovaných za behu a aj pamäťové miesto pre premennú- register. Počet prístupov jedného vlákna kernelu ku globálnej pamäti je rovný šírke filtra²+ 1, čo je takmer dvojnásobne menej ako pri naivnej implementácii.

Výsledky vylepšenej naivnej implementácie je možné vidieť v Tabuľka 8.

Tabuľka 8 : Implementácia Kernel naive improved

Šírka filtra	Celkový čas(μ s)	Čas trvania kernelu(μ s)	Pamäťová priepustnosť(GB/s)	Výpočtová priepustnosť(GFLOP/s)
1	5894,76	222,144	77,27	20,74
3	6850,7	379,072	293,63	109,4
5	7435,67	826,144	372,91	139,44
7	8315,15	1366,08	444,58	165,28
9	9195,01	2195,104	459,71	170,04
11	10349,3	3234,176	467,88	172,4
13	12629,3	4565,44	464,31	170,58
15	14429,7	6137,536	460,85	168,93

Úpravy, ktoré sme vykonali oproti naivnej implementácii mali významný dopad na pozorované veličiny. Môžeme povedať, že sa jedná o najväčší skok v rýchlosti medzi dvoma iteráciami, ktorý dosiahneme. Výpočtová priepustnosť je 10-násobná oproti naivnej implementácii a dosahujeme vysokú pamäťovú priepustnosť. To, čo dokážeme na danej

implementácii vylepšiť, je zmenšiť počet prístupov ku globálnej pamäti, využitím zdieľanej pamäte a zarovnáme pamäť, aj pre vstup a výstup, aby sme maximalizovali zhlukovanie pamäti, o ktorom sa píše v kapitole 2.4.

Implementácia so zdieľanou pamäťou

V predchádzajúcich implementáciách sme nepoužívali zdieľanú pamäť. Mali sme snahu uložiť konvolučné jadro do zdieľanej pamäte, ale neúspešne. Pripomeňme si výhody zdieľanej pamäte. Sídli priamo na multiprocessore a je mnohonásobne rýchlejšia ako globálna pamäť a nevyžaduje striktné dodržiavanie prístupových vzorov ako tomu je pri globálnej pamäti. Jej nevýhodou je jej obmedzené množstvo.

Vlákno kernelu doteraz zakaždým načítavalo hodnoty všetkých svojich susedných pixelov. Môžeme využiť skutočnosť, že vlákna sú organizované v blokoch a tak môžu vlákna v rámci bloku zdieľať hodnoty svojich susedov a tým redukovať prístup ku globálnej pamäti. Teraz každé vlákno načíta jednu hodnotu vstupu z globálnej pamäte a uloží ju do zdieľanej. Pri výpočte výstupného pixelu nebudeme pristupovať ku globálnej pamäti, ale ku zdieľanej.

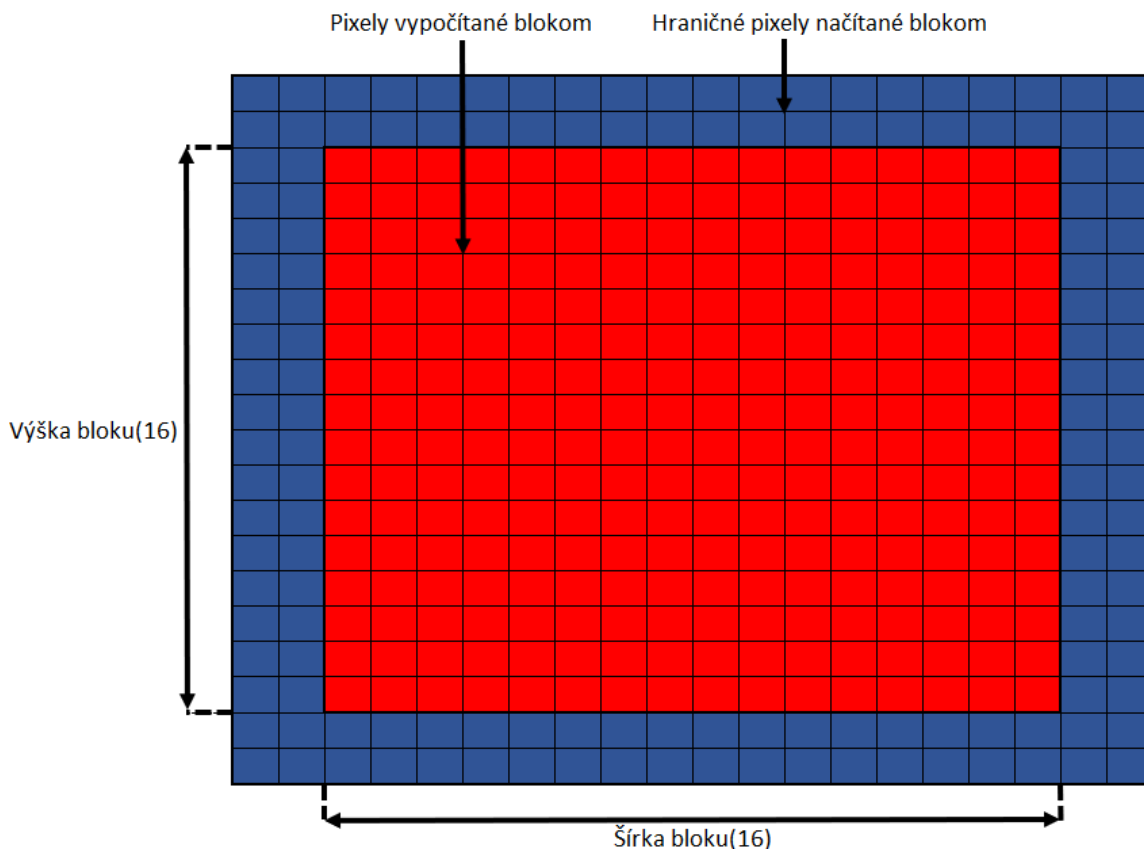
Vznikla požiadavka na efektívne načítanie globálnej pamäte do zdieľanej pamäte v rámci bloku. Pri hľadaní riešenia sme identifikovali dva rôzne vzory načítania, každý z nich má svoje výhody a nevýhody. Každý si podrobne popíšeme.

Nech blok vlákien má šírku B_w a výšku B_h . Šírka konvolučného jadra je h . V ilustračných obrázkoch bude $B_w = 16$, $B_h = 16$ a $h = 5$. Prvý vzor je znázornený na Obrázok 14. Uvedieme pseudokód pre načítanie do pamäte:

```
positionShared.x = absolutna x-ova pozicia bloku
positionShared.y = absolutna y-ova pozicia bloku
threadX = id vlakna x
threadY = id vlakna y
pre j = threadY; j < h - 1 + Bh ; j += Bh
  pre i = threadX; i < h - 1 + Bw ; i += Bw
    shared[j][i] = vstup[positionShared.y + j][positionShared.x + i]
synchronizuj vlakna v bloku
```

Vlákna bloku načítajú 2D oblasť o rozmeroch $(B_w + h - 1) \times (B_h + h - 1)$, čo je aj počet prístupov bloku ku globálnej pamäti pri načítaní. Vlákna bloku vypočítajú $B_w \times B_h$ pixelov. Vo vylepšenej naivnej implementácii je počet prístupov ku globálnej pamäti pri načítaní v rámci bloku rovný $B_w \times B_h \times h^2$. Nevýhodou tohto prístupu je, že niektoré vlákna musia načítať viac hodnôt z globálnej pamäte, zatiaľ čo iné vlákna načítali hodnoty a čakajú, čo sa prejaví dlhým časom synchronizácie vlákien v rámci bloku. Výhodou tohto prístupu

je, že nemá obmedzenie pre veľkosť konvolučného jadra a je s ním možné počítať aj konvolučné jadrá veľkých rozmerov. Počet spustených blokov zostáva rovnaký ako pri naivnej implementácii. Optimalizácii parametrov šírka a výška bloku sa v tejto implementácii zatiaľ nebudeme venovať, ale uvedomujeme si, že ich výber má dopad na čas trvania kernelu.



Obrázok 14 : Prvý vzor pre načítanie do zdieľanej pamäte

Druhý vzor je ukázaný na obrázku 14. Uvedieme pseudokód pre načítanie do pamäte:

```
threadX = id vlakna x
threadY = id vlakna y
absoluteImagePosition.x = absolutna x-ova pozicia obrazu
absoluteImagePosition.y = absolutna y-ova pozicia obrazu
shared[threadY][threadX] = vstup[absoluteImagePosition.y][absoluteImagePosition.x]
```

Potrebuje zaviesť pojem dlaždica. Jednoducho povedané, dlaždica je taká oblasť bloku, ktorá počíta výstup pre daný blok. Nech je šírka dlaždice T_w a výška dlaždice T_h . Na Obrázok 15 je $T_w = 12$ a $T_h = 12$ a dlaždica je znázornená červenou farbou. V tomto postupe každé vlákno bloku načíta jednu hodnotu z globálnej do zdieľanej pamäte. Vlákna bloku načítajú 2D oblasť o rozmeroch $B_w \times B_h$, výpočtu sa ale nezúčastňujú všetky vlákna, ale iba ich podmnožina. Pre veľkosť dlaždice musí platiť, že $T_w \leq B_w - (h - 1)$ a $T_h \leq B_h - (h - 1)$. Počet vlákien, ktoré pokračujú vo výpočte je $T_w \times T_h$ a vlákna, ktoré slúžia pre načítanie

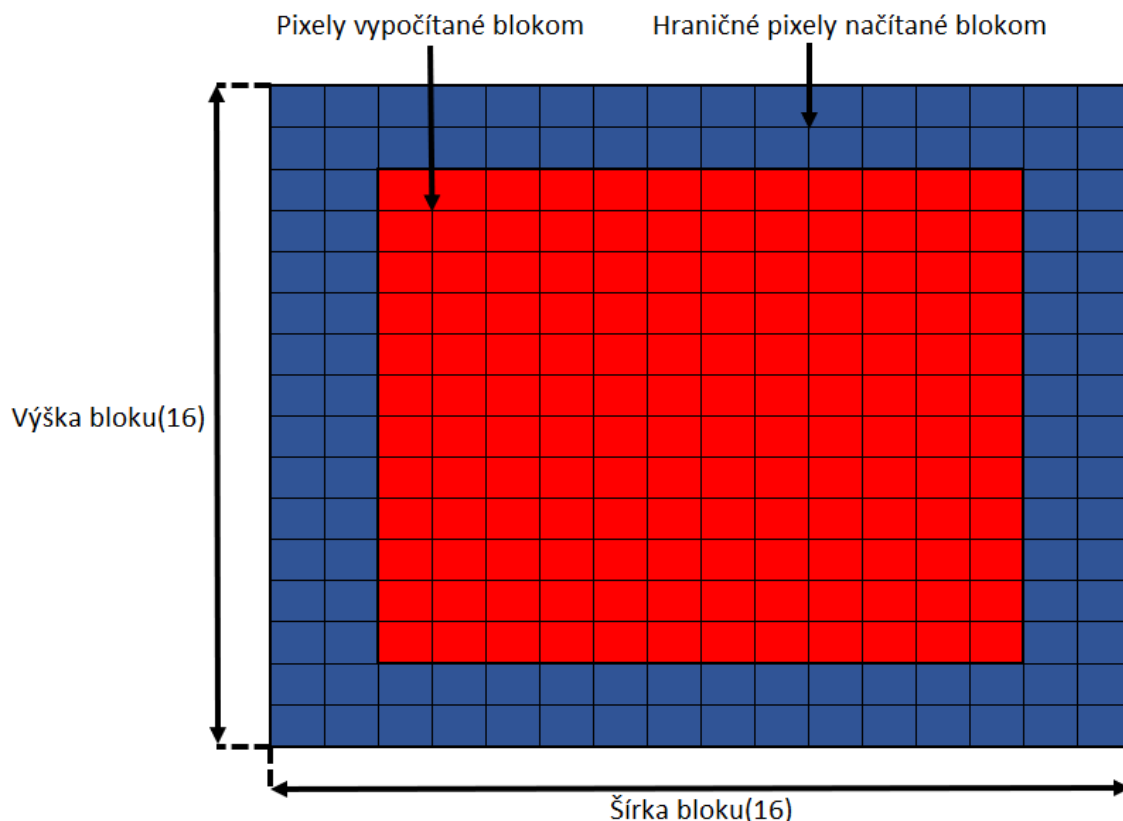
hraničných pixelov, je $B_w \times B_h - T_w \times T_h$. V Tabuľka 9 uvedieme, ako šírka konvolučného jadra vplýva na počet aktívnych a pasívnych vlákien. Počet vlákien v bloku bude 1024, čo je hardvérové obmedzenie vyplývajúce z výpočtovej schopnosti zariadenia, a dlaždica bude maximálne možného rozmeru, vyplývajúceho z obmedzenia, ktoré pre ňu platí.

Tabuľka 9 : Vplyv šírky konvolučného jadra na počet aktívnych a pasívnych vlákien

Šírka filtra	Šírka bloku	Výška bloku	Šírka dlaždice	Výška dlaždice	Počet vlákien v bloku	% aktívnych vlákien	% pasívne vlákien
1	32	32	32	32	1024	100,00	0,00
3	32	32	30	30	1024	87,89	12,11
5	32	32	28	28	1024	76,56	23,44
7	32	32	26	26	1024	66,02	33,98
9	32	32	24	24	1024	56,25	43,75
11	32	32	22	22	1024	47,27	52,73
13	32	32	20	20	1024	39,06	60,94
15	32	32	18	18	1024	31,64	68,36

Môžeme si všimnúť, že s rastúcou šírkou konvolučného jadra sa zväčšuje aj počet pasívnych vlákien, pri jadrách so šírkou 11 a vyššie je viac ako 50 % vlákien nepodieľa na výpočte. Problémom nie je len to, že sú vlákna pasívne, ale dochádza k divergencií vlákien v rámci warpu. Ak by sme chceli týmto spôsobom počítať veľké konvolučné jadrá, bolo by to nemožné, lebo by nebola dodržaná podmienka pre veľkosť dlaždice. Maximálne môžeme počítať s konvolučným jadrom o šírke 32 a aj to veľmi neefektívne.

V rámci bloku vieme vypočítať iba toľko pixelov, aký je rozmer dlaždice. Z toho vyplýva, že musíme zvýšiť aj počet spúšťaných blokov, v porovnaní s naivnou implementáciou. Ak pôvodná mriežka mala $N \times M$ blokov, teraz bude mať $N * B_w / T_w \times M * B_h / T_h$ blokov. Z toho, čo sme napísali, jediným pozitívom tohto vzoru je, že každé vlákno načíta presne 1 hodnotu z globálnej pamäte do zdieľanej a synchronizácia v bloku je rýchla. Ako pri prvom vzore, aj pri tomto zohráva výška a šírka bloku veľkú úlohu. Okrem toho máme k dispozícii parametre šírka a výška dlaždice. Spomenuté parametre zatiaľ nebudeme optimalizovať.



Obrázok 15 : Druhý vzor pre načítanie do pamäte

Ďalšou technikou, ktorá by mala ovplyvniť čas trvania kernelu je pridanie zarovnania pamäte pre vstup a výstup. Doteraz sme vstup a výstup ukladali vo veľkom jednorozmernom poli. Iba pri prvom riadku obrázku sme mali istotu, že je zarovnaný. Zarovnanie pamäte musíme zabezpečiť na hostovi. CUDA obsahuje funkciu pre alokáciu a kopírovanie pamäte pre dvojrozmerné polia. Funkcia alokuje pamäť po riadkoch obrazu s tým, že ďalší riadok sa nachádza až na adrese, ktorá je zarovnaná. Pri prístupe k pamäťovému priestoru, potrebnému pre zarovnanie pamäte, nie je definované správanie. Týmto spôsobom necháme zarovnať aj vstup a výstup. Dobrou správou je, že funkcie alokácie a kopírovania nie sú pomalšie oproti svojim jednorozmerným náprotivkom. Aby sme nemuseli v kerneli robiť kontrolu pre veľkosti obrazu, alokujeme o niečo väčší vstup a výstup, aby kernel mohol pristupovať k pixelom, ktoré sú mimo obrazu.

Pre výpočet hraničných bodov používame rozšírenie. Možno je to detail, ale v rámci kernelu sú to 4 volania funkcie min alebo max plus 1 premenná navyše pre uloženie výsledku. Tieto zdroje vieme ušetriť, ak pre výpočet hraničných bodov použijeme orezanie obrázku. Preto budeme používať pre výpočet hraničných bodov orezanie.

Poslednou úpravou, ktorá sa netýka priamo kernelu, ale prenosu dát medzi CPU a GPU bude použitie pinned memory. Pamäť alokovaná funkciou malloc na CPU je

defaultne stránkovateľná. GPU nevie pristupovať k dátam zo stránkovateľnej pamäte priamo, ale pri prenose dát musí CUDA najskôr alokovať dočasnú pinned memory a následne dáta zo stránkovateľnej pamäte preniesť do pinned memory. Môžeme ušetriť tento prenos, ak pre prenos použijeme pinned memory. Rýchlosť kopírovania sa z pôvodných 3000 MB/s zvýšila na 12000 MB/s.

Po všetkých spomenutých úpravách sa môžeme pozrieť na výsledky dvoch implementácií, ktoré sa líšia vzorom pre načítanie dát do zdieľanej pamäte. Pri prvom vzore použijeme blok o veľkosti 32x16 a pri druhom 32x32. Dlaždica bude mať maximálny rozmer vyplývajúci z obmedzenia, ktoré musí splňať.

Tabuľka 10 : Implementácia Kernel shared full block: Vzor 1

Šírka filtra	Celkový čas(μ s)	Čas trvania kernelu(μ s)	Pamäťová priepustnosť(GB/s)	Výpočtová priepustnosť(GFLOP/s)
1	7363,44	256,256	66,99	17,98
3	6865,23	532,704	43,3	77,85
5	7502,26	705,856	34,96	163,21
7	7822,77	801,024	32,82	281,88
9	8228,55	929,152	26,56	401,71
11	8415,53	1177,344	28,02	473,58
13	9531,76	1507,936	23,12	516,44
15	9911,19	1976	18,6	524,7

Tabuľka 11 : Implementácia Kernel shared incomplete block: Vzor 2

Šírka filtra	Celkový čas(μ s)	Čas trvania kernelu(μ s)	Pamäťová priepustnosť(GB/s)	Výpočtová priepustnosť(GFLOP/s)
1	8281,9	226,752	76,71	20,59
3	7557,76	273,696	85,69	151,53
5	7698,66	412,256	59,37	182,12
7	8029,38	650,24	44,66	354,35
9	8324,23	951,648	29,55	392,21
11	9161,34	1535,232	24	369,26
13	10000,6	2284,992	16,05	340,81
15	11479	3380,64	13,72	308,18

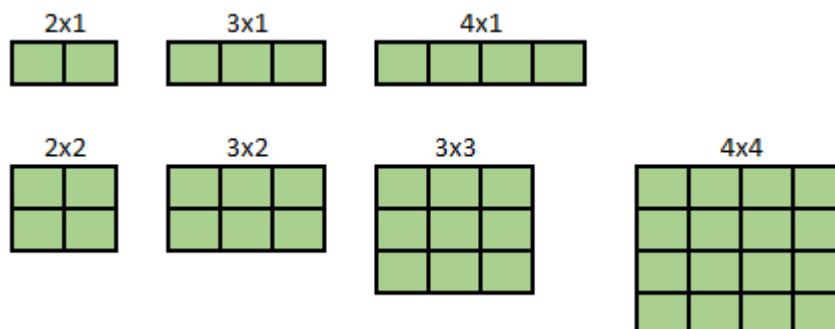
Načítanie dát z globálnej pamäte do zdieľanej pamäte, zarovnanie pamäte a použitie orezania pre výpočet hraničných pixelov prinieslo zlepšenie. Najväčší vplyv na posun vo výkone malo použitie zdieľanej pamäte. Vzor 1 dosahuje lepšie výsledky pre konvolučné

jadrá šírky 9 a viac. Pre malé jadrá (1x1, 3x3) je pomalší ako vylepšená naivná implementácia. Dávame tomu za príčinu dlhej synchronizácií vlákien v rámci bloku. Ako sme mohli očakávať, výkon vzoru 2 degraduje s rastúcou šírkou konvolučného jadra. Pre jadrá šírky 1, 3, 5 a 7 dosahuje vynikajúce výsledky z pohľadu času trvania kernelu.

Ak by sme optimalizovali parametre jednotlivých vzorov (šírka, výška bloku, dlaždice), nachádzali by sme sa v štádiu, v ktorom, by sme mohli nadobudnúť pocit, že už nie je čo zlepšiť. Pre malé konvolučné jadrá by sme použili vzor 1 a pre veľké vzor 2. Po podrobnej analýze kernelov sme zistili, že kernely nie sú dostatočne vyťažené a najväčšie percento času strávi kernel pamäťovými operáciami. Potrebujeme zvýšiť množstvo práce, ktoré kernel vykonáva a to tak, že kernel nebude počítat iba jeden výstupný pixel, ale viacero pixelov a zároveň bude načítavať viac dát z globálnej pamäte do zdieľanej. Tým sa zníži aj celkový počet spúšťaných blokov. Je potrebné určiť, aké rozmery bude mať oblasť, ktorú bude kernel počítat. Nazvime túto oblasť subdlaždica. Šírka subdlaždice bude P_w a výška P_h . Je nutné určiť vhodné rozmery subdlaždice Po prvé preto, lebo od rozmerov subdlaždice závisí počet spustených blokov. Ako príklad uvidíme obraz, ktorého rozmery sú 1920 x 1200 a veľkosť bloku je 32 x 32. Veľkosť subdlaždice bude 2x2. Pôvodne by sme museli spustiť ($1920 / 32 = 60$; $1200 / 32 = 37,5$; $60 \times 38 = 2280$) 2280 blokov. Ak použijeme subdlaždicu, počet blokov klesne na ($1920 / 2 / 32 = 30$; $1200/2/32 = 18.75$; $30 \times 19 = 570$), 570, čo je 4-krát menej blokov. Potom počet blokov pri použití subdlaždice je možné vypočítať ako podiel pôvodného počtu blokov a súčinu výšky a šírky subdlaždice. Po druhé, musíme myslieť na vzor k prístupu ku globálnej pamäti. Naše obrazy sú v pamäti ukladané po riadkoch, preto nie je vhodné používať subdlaždie, ktoré majú veľkú výšku a malú šírku, lebo by sme každý pixel načítavali z pamäťovej adresy, ktorá má inú lokalitu. Po tretie, veľkosť subdlaždice vplýva aj na použité prostriedky kernelu. Čím väčší rozmer subdlaždice, tým potrebuje kernel viac registrov pre uloženie výsledkov. Zvýši sa aj počet potrebnej zdieľanej pamäte, ktorú vyžaduje blok na svoje spustenie. Pretože množstvo zdieľanej pamäte pripadajúcej na blok je obmedzené, môže nastať situácia, pri ktorej nebude možné blok spustiť, lebo vyžaduje väčšie množstvo zdieľanej pamäte, aké sú hardvérové limity zariadenia. Po štvrté, musíme myslieť na parametre ako veľkosť bloku, v prípade druhého vzoru aj veľkosť dlaždice a aj šírku konvolučného jadra. Určiť vhodnú veľkosť subdlaždice preto nie je jednoduchá úloha.

Pri počiatočnom výbere subdlaždice sme si vybrali subdlaždice, o ktorých sme predpokladali, že pri ich použití bude dodržaný vzor prístupu ku globálnej pamäti

a nespotrebuju veľké množstvo registrov pre uloženie výsledkov. Následne výber subdlaždice prebiehal experimentálne. Na Obrázok 16 sú znázornené subdlaždice, ktoré sme testovali.



Obrázok 16 : Testované rozmery subdlaždíc

Očakávali sme, že najlepšie si budú počínať subdlaždice, ktorých výška je jedna, lebo kernel pristupuje k pamäťovým miestam, ktoré sú uložené za sebou, ale nebolo tomu tak. Najlepšie výsledky dosahovali pre malé konvolučné jadrá subdlaždice veľkosti 2x2 a pre veľké 3x3.

V tejto implementácii sme taktiež upravili spôsob, akým pristupujeme ku globálnej pamäti z pohľadu počtu prenášaných bajtov v jednej transakcii. Doteraz sme pri každom prístupe ku globálnej pamäti čítali a zapisovali údaje o veľkosti údajového typu float. Pri subdlaždici 4x1 sme museli 4-krát pristúpiť k pamäťovým miestam, ktoré sa nachádzajú v pamäti fyzicky za sebou. Túto skutočnosť môžeme využiť a v jednej transakcii preniesť údaje o veľkosti $4 * \text{sizeof(float)}$ začínajúc na adrese prvého prvku subdlaždice. Tým zväčšíme množstvo prenášaných dát a znížime počet prístupov k pamäti. Tento prístup aplikujeme aj pri ostatných subdlaždiciach.

V predchádzajúcich implementáciách so zdieľanou pamäťou sme spomenuli, že pri nich nebudeme optimalizovať parametre šírka a výška bloku, prípadne šírka a výška dlaždice pri vzore 2. V aktuálnej implementácii to zmeníme a nájdeme optimálne parametre. Je nutné podotknúť, že zvolenie správnej šírky a výšky bloku má výrazný vplyv na čas behu kernelu. Výška a šírka bloku musí byť zvolená tak, aby došlo k maximálnej využitiu zdrojov GPU. Ak hovoríme o zdrojoch, tak máme na mysli počet registrov, ktoré používa blok a veľkosť zdieľanej pamäte pre blok. Taktiež musíme myslieť na to, že počet spúšťaných vlákien v bloku musí byť násobkom počtu vlákien vo warpe, kvôli zhlukovaniu pamäte a aby boli aktívne všetky vlákna vo warpe. Dosiahneme to jednoducho, parameter šírka bloku bude vždy celočíselným násobkom počtu vlákien vo warpe (32). Z toho sme určili, že šírka bloku

bude vždy 32. Zostáva nám určiť iba správnu výšku bloku. Výška bloku bola určená tak, aby sme dosiahli maximálnu utilizáciu zdrojov GPU s ohľadom na výpočtový čas. Nastali prípady, že sme mali vysokú utilizáciu, ale celkový čas trvania kernelu bol dlhší. Pri vzore 2 bolo nutné určiť aj veľkosť dlaždice. Ako sme mohli očakávať, je najlepšie, ak má dlaždica maximálny možný rozmer vzhľadom na blok. V tabuľkách nižšie sú znázornené výsledky pre vzor 1 (ďalej ho budeme volať Kernel shared full block) a vzor 2 (nazveme ho Kernel shared incomplete block), s použitím subdlaždíc o rozmeroch 2x2 a 3x3 spolu s hodnotami parametrov, ktoré dosahovali najvyšší výkon.

Tabuľka 12 : Implementácia Kernel shared full block: Vzor 1, rozmer subdlaždice 2x2

Šírka filtra	Rozmer bloku	Celkový čas	Čas trvania kernelu(μ s)	Pamäťová priepustnosť(GB/s)	Výpočtová priepustnosť(GFLOP/s)
1	32x4	6378,15	202,048	84,96	22,81
3	32x8	6784,06	227,264	90,88	182,48
5	32x8	6767,98	255,36	86,13	451,13
7	32x8	7396	305,6	76,36	738,85
9	32x8	7594,25	380,672	60,6	980,5
11	32x8	7901,28	482,336	57,56	1155,97
13	32x10	8510,59	527,104	52,11	1477,42
15	32x10	8707,71	620,48	46,17	1670,96

Tabuľka 13: Implementácia Kernel shared full block: Vzor 1, rozmer subdlaždice 3x3

Šírka filtra	Rozmer bloku	Celkový čas	Čas trvania kernelu(μ s)	Pamäťová priepustnosť(GB/s)	Výpočtová priepustnosť(GFLOP/s)
1	32x4	7028,18	278,592	154,04	16,54
3	32x8	7243,56	353,344	144,23	117,37
5	32x8	7359,82	370,144	145,41	311,23
7	32x8	7222,67	374,656	136,38	572,12
9	32x8	7720,64	460,992	124,71	809,66
11	32x7	7997,4	507,328	135,54	1115,51
13	32x6	8086,21	496,576	143,31	1576,08
15	32x8	8258,31	589,888	113,83	1757,62

Tabuľka 14: Kernel shared incomplete block: Vzor 2, rozmer subdlaždice 2x2

Šírka filtra	Rozmer bloku	Rozmer dlaždice	Celkový čas(μ s)	Čas trvania kernelu(μ s)	Pamäťová priepustnosť(GB/s)	Výpočtová priepustnosť(GFLOP/s)
1	32x16	32x16	6528,78	201,888	86,16	23,13
3	32x16	31x15	6767,88	203,552	102	203,95
5	32x16	30x14	6849,29	205,632	103,58	562,09
7	32x32	29x29	7129,76	293,728	79,5	801,37
9	32x32	28x28	7124,54	417,792	49,58	936,31
11	32x32	27x27	7374,57	562,784	43,97	1038,22
13	32x32	26x26	7430,1	678,976	36,42	1195,31
15	32x18	25x11	7905,17	849,312	37,15	1250,16

Tabuľka 15: Kernel shared incomplete block: Vzor 2, rozmer subdlaždice 3x3

Šírka filtra	Rozmer bloku	Rozmer dlaždice	Celkový čas(μ s)	Čas trvania kernelu(μ s)	Pamäťová priepustnosť(GB/s)	Výpočtová priepustnosť(GFLOP/s)
1	32x16	32x16	7075,73	267,712	96,18	17,21
3	32x16	31x15	7327,56	310,752	92,42	137,45
5	32x16	30x14	7102,56	323,904	94,04	372,28
7	32x16	30x14	7043,2	324,512	94	728,3
9	32x16	29x13	7173,08	355,392	91,83	1102,76
11	32x20	28x16	7552,95	444,224	69,15	1263
13	32x20	28x16	7679,38	570,656	53,76	1373,19
15	32x13	27x8	7903,67	605,568	59,98	1733,51

Zvýšenie množstva práce, ktorú vykonáva vlákno má merateľný dopad na čas behu kernelu. Testy ukázali, že subdlaždica veľkosti 2x2 je vhodná pre jadrá menšej šírky (1, 3, 5, 7) a subdlaždica 3x3 zas pre väčšie (9, 11, 13, 15). V tabuľkách majú žlté pozadie tie konfigurácie, ktoré zo všetkých konfigurácií dosahujú najlepší výpočtový čas pre danú šírku konvolučného jadra. Každá konfigurácia má optimálne zvolené parametre, vzhľadom na hardvérové vlastnosti grafickej karty, ktorú používame.

Za skvelý výsledok považujeme rovnaký čas trvania kernelu pre konvolučné jadrá šírky 1, 3 a 5. Pri konvolučných jadrách šírky 13 a 15 sme dosiahli 3-násobné zrýchlenie v porovnaní s implementáciou, ktorej vlákno počítalo iba jeden výstup.

Pri pohľade na pamäťovú a výpočtovú priepustnosť kernelu pre šírky 1 a 5 vidíme, že výpočtová priepustnosť pre šírku 5 je mnohonásobne vyššia ako pri šírke 1. Pamäťová priepustnosť je ale podobná. Z toho vyplýva, že kernely šírky 1,3 a 5 spomaľuje pamäťová priepustnosť zariadenia a spôsob prístupu ku globálnej pamäti a nie aritmetické operácie,

ktoré kernely vykonávajú. Pamäťovú priepustnosť zariadenia ovplyvniť nevieme, je daná výrobcom, ale pri vzore prístupu k pamäti si myslíme, že nedokážeme nájsť lepší vzor.

Pre jadrá šírky 7 a viac je naopak počet aritmetických operácií viacnásobne väčší oproti jadrám menšej šírky, najlepšie to vidieť na výpočtovej priepustnosti kernelu. Limitujúcim faktorom je rýchlosť aritmetických operácií a nie vzor prístupu ku globálnej pamäti, ako pri jadrách menšej šírky. Rýchlosť aritmetických operácií by sme mohli vylepšiť špeciálnymi hardvérovými inštrukciami, ale tieto nespĺňajú štandard IEEE 754 a po ich nasadení nebol dopad na výpočtový čas výrazný.

Po všetkých úpravách, ktoré sme spravili od naivnej implementácie až po aktuálnu sa domnievame, že sme sa dostali do bodu, v ktorom nevieme vylepšiť prístup ku globálnej pamäti a ani iným spôsobom výrazne ovplyvniť výpočtový čas kernelu. Získané implementácie považujeme týmto za finálne a budú použité ako základ pre konvolúciu 1:N:N a N:M:M. Celkovo použijeme 3 rôzne kernely, 2 rôzne vzory prístupu ku globálnej pamäti. Finálne implementácie s výsledkami a parametrami je možné vidieť v Tabuľka 16.

Tabuľka 16: Finálne implementácie kernelov pre konvolúciu 1:1:1

Šírka filtra	Rozmer bloku	Rozmer dlaždice	Celkový čas(μ s)	Čas trvania kernelu(μ s)	Pamäťová priepustnosť(GB/s)	Výpočtová priepustnosť(GFLOP/s)
Kernel shared incomplete block : Vzor 2						
Rozmer subdlaždice: 2x2						
1	32x16	32x16	6528,78	201,888	86,16	23,13
3	32x16	31x15	6767,88	203,552	102	203,95
5	32x16	30x14	6849,29	205,632	103,58	562,09
7	32x32	29x29	7129,76	293,728	79,5	801,37
Kernel shared incomplete block: Vzor 2						
Rozmer subdlaždice: 3x3						
9	32x16	29x13	7173,08	355,392	91,83	1102,76
11	32x20	28x16	7552,95	444,224	69,15	1263
Kernel shared full block: Vzor 1						
Rozmer subdlaždice: 3x3						
13	32x6	x	8086,21	496,576	143,31	1576,08
15	32x8	x	8258,31	589,888	113,83	1757,62

4.5.2 Konvolúcia 1:N:N

Doteraz sme riešili vylepšovanie implementácie pre konvolúciu 1:1:1. Konvolúciu 1:N:N by sme mohli považovať za špeciálny prípad konvolúcie 1:1:1, kde by sme pre každé konvolučné jadro spustili kernel pre konvolúciu 1:1:1, ale stratili by sme tak priestor na zlepšovanie. Pri konvolúcii 1:1:1 máme vstupný obraz načítaný v zdieľanej pamäti

a počítame výstup iba pre jedno konvolučné jadro. Kernel pre konvolúciu 1:1:1 upravíme tak, že načítanie údajov do zdieľanej pamäte zostane nezmenené, ale budeme počítat viacero výstupov pre konvolučné jadrá rovnakej šírky. Tu nastali viaceré komplikácie. Ak máme N konvolučných jadier rovnakej šírky, potrebujeme mať na GPU naalokovaných N dvojrozmerných polí pre výstup. Bolo by veľmi nerozumné, ak by sme pred každým spustením takéhoto kernelu alokovali N dvojrozmerných polí pre výstup. Na riešenie tohto problému sme použili memory pool, alebo “bazén pamäte”. Nazveme ho memory pool pitched, pitched je časť názvu funkcie, ktorá alokuje pamäť tohto memory poolu. Pri jeho použití budeme môcť pri jednom spustení kernelu vypočítať toľko výstupov, aká je veľkosť memory poolu. Pri spustení aplikácie vyalokujeme konštantný počet dvojrozmerných polí na GPU vopred určenej veľkosti. Najlepšie by bolo, ak by bol počet dvojrozmerných polí v memory poole čo najväčší a rozmer dvojrozmerného poľa mal taký rozmer, ako je najväčší obraz, s ktorým sa bude počítat. Ak je obraz väčší ako dvojrozmerné pole, dôjde k prealokovaniu memory poolu a ak je počet požadovaných výstupov väčší ako veľkosť memory poolu, tak musíme výpočet rozdeliť do viacerých kernelov. Ak sme už použili memory pool pre výstup obraz, môžeme ho rovnako použiť aj pre vstup.

Veľkosť memory poolu pitched pre vstup sme určili na 10 a pre výstup na 20. Počiatočný rozmer dvojrozmerného poľa je 2000x2000 a ďalších 300 pre každý rozmer navyše, aby kernel mohol pristupovať aj mimo rozsah obrazu. Memory pooly by mohli byť aj väčšej veľkosti, ale naša grafická karta má iba 2GB globálnej pamäte. Pri terajšej veľkosti memory poolov potrebujeme $sizeof(float) \times 2300 \times 2300 \times (10 + 20) / 2^{20} \cong 605$ MB z globálnej pamäte. Deliteľ 2^{20} slúži na prevod z B na MB. Tieto hodnoty môže programátor podľa potreby zmeniť.

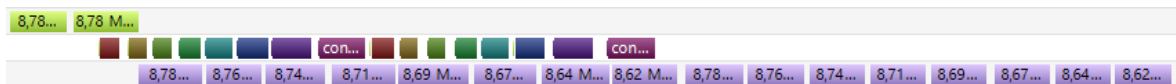
Máme memory pool pre vstup aj výstup, teraz ale potrebujeme dostať adresy výstupu do kernelu. Prenášali sme ich parametrom kernelu ako pole polí smerníkov, ale výsledok bol zarážajúci. Ako príklad uvidieme Kernel shared incomplete block so šírkou konvolučného jadra 3. Pred úpravou trval kernel približne 200 μ s a po pridaní parametra pole polí smerníkov až 2600 μ s, čo nie je prípustné. Museli sme nájsť iný spôsob, ako dostať počiatočné adresy pre výstup do kernelu. Ako pri konvolučných jadrách, aj teraz sme použili pamäť konštant. Potom sme sa opäť dostali na 200 μ s pri jednom výstupe.

Výpočtová kategória zariadenia, ktorú používame, umožňuje asynchrónny prenos dát medzi CPU a GPU a počas samotného prenosu dát dokáže zariadenie spúšťať kernely. Táto schopnosť nám príde ako veľmi vhodná a žiadúca, lebo nám umožní počas výpočtu prenášať

údaje, či už vstupný alebo výstupný obraz a tým ušetríme čas. Pre názornosť ukážeme konvolúciu 1:N:N, ktorej vstupom budú 2 obrazy a konvolučné jadrá šírky 1,3,5,7,9,11,13 a 15 a operácie počítania a kopírovania dát budú prebiehať sériovo. Tento príklad je znázornený na Obrázok 17. Na Obrázok 18 je príklad konvolúcie, kde operácie prebiehajú asynchrónne. Zeleno farbou je znázornený prenos údajov z CPU na GPU a fialovou z GPU na CPU. Ostatné farby sú kernely.



Obrázok 17 : Operácie vykonávané synchronne



Obrázok 18 : Operácie vykonávané asynchrónne

Aby sme dosiahli požadované správanie, musí byť dodržaných viacero podmienok. Operácie, od ktorých požadujeme, aby prebiehali asynchrónne, musia byť vykonávané v samostatných prúdoch. Prúd je sekvencia operácií, ktoré sa vykonávajú v poradí príchodu na GPU. CUDA operácie v rôznych prúdoch môžu byť vykonávané paralelne a operácie v rôznych prúdoch môžu byť prerušené. Všetky operácie, ktoré sme vykonávali (spúšťanie kernelov, kopírovanie pamäte), boli vykonávané v takzvanom defaultnom prúde. Aby sme dosiahli asynchrónnosť, operácie musia prebiehať mimo defaultného prúdu. Každý prúd je možné explicitne synchronizovať. Aby bolo kopírovanie pamäte asynchrónne, musí byť použitá pinned memory, o ktorej sme písali v kapitole 4.2.3 a funkcia pre asynchrónne kopírovanie. Kopírovanie pamäte musí prebiehať v rozličných smeroch a zariadenie musí mať dostatok zdrojov (registre, zdieľaná pamäť). Aby sme splnili podmienky, musíme použiť 3 rôzne prúdy. Jeden pre kopírovanie dát z CPU na GPU, druhý prúd pre spúšťanie kernelov a tretí pre kopírovanie dát z GPU na CPU. Každý prúd musí byť samostatne synchronizovaný z pohľadu CPU, preto každý prúd umiestnime do samostatného vlákna. Dostaneme tak tri vlákna a každému bude prislúchať jeden prúd. Stručne popíšeme prácu vlákien. Prvé vlákno môžeme označiť za preprocessing, bude prideľovať úlohy (pracovné dávky) druhému vláknu a načítavať vstup z CPU do GPU. Úloha je charakterizovaná počtom konvolučných jadier, šírkou konvolučného jadra (úloha má konvolučné jadrá rovnakej šírky), pamäťou inicializovanou hodnotami konvolučných jadier, adresou vstupného obrazu, začiatočným indexom do poľa výstupných adries a rozmiery spracovaného obrazu. Druhé vlákno, processing, bude preberať úlohy od prvého vlákna, spúšťať kernely a po výpočte výstupy odovzdá tretiemu vláknu. Tretie vlákno alebo postprocessing, skopíruje výstupy

z GPU do CPU pamäte. Najväčším synchronizačným a algoritmickým problémom, ktorý sme museli vyriešiť, bolo, ako použiť pamäť z memory poolu pre výstup. Budeme vždy prideliť úlohe celý adresný rozsah výstupu alebo iba toľko, koľko výstupov úloha potrebuje? Kedy budeme uvoľňovať pridelené adresy, až po tom, keď sa skopírujú všetky výstupy úlohy alebo hneď, keď je výstup spracovaný? Netreba zabúdať, že tieto adresy máme v jednorozmernom poli. Ako riešenie sme použili cyklický front. Pri vyžiadaní adresy/adries výstupu front vracia počiatočný index, kde začína prvá adresa výstupu. Tento index vieme poslať do kernelu a na základe informácie o veľkosti frontu vypočítať adresu výstupu. Táto operácia je blokujúca, ak vo fronte nie je dostatok adries. Akonáhle sa uvoľní požadovaný počet adries, index je pridelený. Do fronty možno vracat adresy po jednom, akonáhle je výstup spracovaný, môže byť opätovne použitý.

Aby boli operácie kopírovania asynchrónne, potrebujeme alokovať pinned memory, ktorá nie je stránkovateľná. Jej alokácia je drahý proces. Aby sme sa tomu čiastočne vyhli, môžeme použiť memory pool, aj pre vstup a výstup. Nazvime ho memory pool pinned. Ak je memory pool pre vstup prázdny, pokúšame sa alokovať pinned memory. Ak zlyhá alokovanie pinned memory, alokujeme stránkovateľnú pamäť. Pri výstupe uprednostňujeme pri vyprázdnení memory poolu alokáciu stránkovateľnej pamäte, aj za cenu pomalého kopírovania. Na alokácii pinned memory sa podieľa aj GPU a táto operácia neprebíha asynchrónne, čo nám prekáža, lebo počas nej nemôžeme kopírovať pamäť ani spúšťať kernely. Ak veľkosť alokovanej pamäte v memory poole nie je dostačujúca (veľkosť požadovanej pamäte je väčšia, oproti tomu, s akou bol memory pool inicializovaný), tak používame rovnaký postup, ako keď je memory pool prázdny.

Samotný kernel si vyžadoval ešte drobné úpravy. Aby sme výkonnostne nezaostávali za kernelom pri konvolúcii 1:1:1, počet filtrov a veľkosť frontu sme museli prenášať ako parametre šablóny. Po výpočte jedného výstupu musíme synchronizovať vlákna v bloku, kvôli zhlukovaniu pamäte. Ak máme iba jeden výstup, táto synchronizácia nie je nutná. Počas testovania sme prišli na to, že pri väčších konvolučných jadrách nie je vhodné, ak kernel počíta toľko výstupov, koľko má k dispozícii pamäť z memory pool pitched pre výstup. Ak počítame pri jadrách šírky 7, 9, 11, 13 a 15 viac výstupov, pri určitom hraničnom počte výstupov vzrastie počet registrov, ktoré potrebuje vlákno a následne sa zmenší počet blokov, ktoré sa môžu spustiť a tým vzrastie aj čas výpočtu. Tieto hraničné hodnoty sme stanovili experimentovaním. Pre jadrá šírky 1, 3, a 5 sme určili počet výstupov na 10, je to polovičná hodnota z veľkosti memory pool pitched pre výstup. Pri ostatných šírkach je tento

počet menší. Teraz sa môžeme pozrieť na výsledky v Tabuľka 17. Stĺpec počet výstupov je aj hraničný počet výstupov, ktorý kernel pri jednom spustení pri danej šírke počíta.

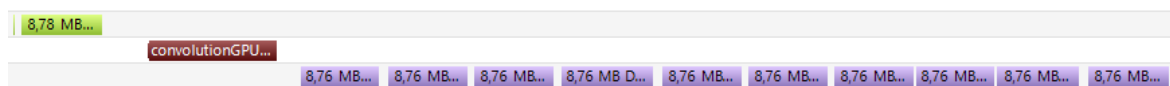
Tabuľka 17: Implementácia Kernel shared threads

Šírka filtra	Počet výstupov	Celkový čas(μ s)	Čas trvania kernelu(μ s)	Celkový priemerný čas kernelu pre jeden výstup(μ s)	Priemerný čas kernelu pre jeden výstup(μ s)	Pamäťová priepustnosť (GB/s)	Výpočtová priepustnosť (GFLOP/s)
1	10	10807,1	1092,544	1080,71	109,2544	87,57	42,74
3	10	10947,7	1112,64	1094,77	111,264	103,4	373,12
5	10	10795,3	1129,6	1079,53	112,96	100,04	1023,23
7	4	5990,67	761,856	1497,6675	190,464	75,64	1235,85
9	3	5152,63	803,52	1717,54333	267,84	89,18	1463,23
11	2	4349,98	690,272	2174,99	345,136	73,06	1658,11
13	1	3254,2	486,72	3254,2	486,72	67,93	1608
15	1	3292,8	574,496	3292,8	574,496	56,41	1804,71

Pri konvolučných jadrách šírky 1, 3 a 5 sa nám opäť podarilo posunúť hranice, ak to porovnáme s konvolúciou 1:1:1. Ak výpočet jedného výstupu trval 200 μ s, teraz trvá v priemere 111 μ s, za predpokladu, že naraz spracúvame 10 konvolučných jadier. So znižujúcim sa počtom spracovaných konvolučných jadier stúpa priemerný výpočtový čas, ale nikdy nebude v priemere horší ako 200 μ s. Pri jadrách šírky 7, 9 a 11 má stále zmysel počítať viac výstupov, ale zlepšenie nie je až také výrazné, ako pri menších jadrách. Ako bolo povedané, pri kerneloch šírky 13 a 15 rastie pri pridaní výstupov počet potrebných registrov, ktoré vyžaduje jedno vlákno a počítajú iba jeden výstup, rovnako ako pri konvolúcii 1:1:1.

Použitie memory pool pitched a pinned prinieslo z pohľadu celkového času zlepšenie. V celkovom čase sú zahrnuté všetky operácie – hlavne kopírovanie vstupu z CPU na GPU, výpočet kernelu a kopírovanie výsledkov z GPU na CPU. Celkové časy v Tabuľka 17 je možné dosiahnuť iba za ideálnych podmienok, keď máme dostatok pamäte v memory pool pinned pre výstup a nedochádza k realokácii memory pool pitched kvôli väčšiemu rozmeru obrazu. V opačnom prípade celkový čas stúpa.

Teší nás skutočnosť, že kernel pre šírku konvolučného jadra 3 dokáže v čase 1094,77 μ s vypočítať naraz 10 konvolučných jadier. Celkový čas je ale 10807,1 μ s, čo je 10-násobok času trvania kernelu. Vizualizujme tento výpočet na Obrázok 19 : Vizualizácia výpočtu.



Obrázok 19 : Vizualizácia výpočtu

Chceme poukázať, že aj keď máme veľmi rýchly kernel, tak úzkym hrdlom výpočtu je práve kopírovanie výstupov z GPU na CPU.

V CUDA 6.0 bol do programovacieho modelu zavedený komponent Unified memory⁸, ktorý definuje jednotný adresný priestor pre všetky procesory v systéme. Unified memory vytvára pool manažovanej pamäte, ktorá je zdieľaná medzi CPU a GPU a je prístupná ako CPU tak aj GPU jediným smerníkom. Unified memory funguje na základe automatického migrovania pamäte medzi CPU a GPU, čo dramaticky zjednodušuje manažment pamäte pri GPU-akcelerovaných programoch. Po alokovaní Unified memory sídli táto pamäť v GPU globálnej pamäti. Ak k nej potrebuje prístup z CPU, nakopíruje sa automaticky do RAM pamäte. Ak chceme k pamäti po prístupe z CPU pristupovať z GPU, v pozadí sa nakopíruje z CPU do GPU. Môžeme jej alokovať maximálne toľko, koľko máme k dispozícii globálnej pamäte.

Kľúčom k odstráneniu explicitného kopírovania výstupov z GPU na CPU by mohlo byť použitie zmienenej Unified memory. Skúsili sme ukladať aj vstup do Unified memory, ale neúmerne to predĺžilo spúšťanie kernelu, lebo sa kopíroval vstup z CPU na GPU (lepšie grafické karty, ako je tá, ktorú používame my, majú zabudovanú hardvérovú podporu pre migráciu Unified memory za behu kernelu⁹). Aby sme mohli použiť Unified memory pre výstup, museli sme upraviť kernel. Vychádzame z implementácie Kernel shared threads. Vieme, že výstup bol z pohľadu GPU dvojrozmerné pole a pri kopírovaní sme ho kopírovali do jednorozmernej CPU pamäte. Kernel mohol rátať aj hodnoty, ktoré sa nachádzali mimo obrazu, ale skopírovali sa iba tie potrebné. Teraz budeme mať na výstupe jednorozmerné pole a musíme ohraničiť, koľko výstupov môže vlákno kernelu uložiť. Pri subdlaždici 2x2 máme 4 možnosti, buď uložíme celú subdlaždicu 2x2, podmnožinu subdlaždice 2x1, 1x2 alebo 1x1. Pri subdlaždici 3x3 je týchto možností až 9. Vlákna preto budú pri počítaní krajných bodov obrazu divergovať. Pre Unified memory sme vytvorili memory pool,

⁸ <https://devblogs.nvidia.com/unified-memory-in-cuda-6/>

⁹ <http://www.acceleware.com/blog/Unified-Memory-on-Tesla-P100-with-CUDA-8.0>

s označením managed, čo je názov funkcie pre alokáciu Unified memory. Memory pool opäť inicializujeme pri spustení aplikácie. V Tabuľka 18 sú výsledky pre kernel spracovávajúci jeden výstup a v Tabuľka 19 výsledky pre kernel spracovávajúci viacero výstupov.

Tabuľka 18: Kernel shared managed 1:1:1

Šírka filtra	Celkový čas(μ s)	Čas trvania kernelu(μ s)	Pamäťová priepustnosť(GB/s)	Výpočtová priepustnosť(GFLOP/s)
1	1621,13	196,512	87,94	23,45
3	1696,67	198,016	104,69	208,87
5	1693,59	200,928	105,42	570,24
7	1765,17	286,784	78,78	780,94
9	1849,14	375,904	84,54	992,26
11	1909,93	375,904	84,27	1463,25
13	2102,72	485,792	67,04	1577,11
15	2301,99	614,376	52,83	1647,67

Tabuľka 19 : Kernel shared managed 1:N:N

Šírka filtra	Počet výstupov	Celkový čas(μ s)	Čas trvania kernelu(μ s)	Celkový priemerný čas kernelu pre jeden výstup(μ s)	Priemerný čas kernelu pre jeden výstup(μ s)	Pamäťová priepustnosť (GB/s)	Výpočtová priepustnosť (GFLOP/s)
1	10	2539,25	1086,56	253,925	108,656	87	42,41
3	10	2665,27	1104,064	266,527	110,4064	103,87	374,61
5	10	2611,03	1134,88	261,103	113,488	98,63	1009,59
7	4	2431,59	734,72	607,8975	183,68	75,68	1219,3
9	3	2548,39	856,864	849,463333	285,621333	79,49	1292,67
11	2	2386,4	759,168	1193,2	379,584	65,02	1449,06
13	1	2130,77	483,168	2130,77	483,168	67,56	1585,67
15	1	2220,44	617,408	2220,44	617,408	52,73	1647,59

Čas trvania kernelu sa pre konvolúciu 1:1:1 a 1:N:N zhoršil iba pre konvolučné jadro šírky 15. Ostatné časy sú podobné ako pri implementácii Kernel shared threads, z ktorej vychádzame. Na Obrázok 20 vizualizujeme výpočet pre konvolúciu 1:N:N, so šírkou konvolučného jadra 3 a 10 výstupmi, rovnako ako sme to spravili pre Kernel shared threads.



Obrázok 20 : Vizualizácia výpočtu

Môžeme si všimnúť, že použitím Unified memory sme odstránili nutnosť kopírovania výsledkov z GPU na CPU, čím sa pre konkrétny výpočet celkový čas

z pôvodných 10807,1 μ s skrátil na 2665,27 μ s. Celkový výpočet pozostáva iba z kopírovania vstupu z CPU na GPU a následného výpočtu. Vyzerá to ideálne, ale v skutočnosti tomu tak nie je. Následne sme porovnávali čas prístupu k Unified memory z CPU s časom prístupu k pamäti alokovanej operátorom new. Zistili sme, že čas potrebný pre prístup k Unified memory z CPU je 3-5 krát väčší ako ku bežnej pamäti. Odôvodňujeme to tým, že na pozadí sa musí skopírovať pamäť z GPU do CPU. Ak chceme k pamäti, ku ktorej sa raz pristupovalo z CPU opäť pristupovať z GPU, opäť sa musí kopírovať pamäť na pozadí. Toto kopírovanie prebieha tesne pred tým, ako s pamäťou pracuje GPU a je veľmi pomalé. Jediná výhoda použitia Unified memory je potom oneskorené kopírovanie výsledkov pri GPU výpočte, aj to za predpokladu že sme predtým k pamäti nepristupovali z CPU. Ďalšou nevýhodou použitia Unified memory je jej obmedzené množstvo a dlhý čas alokácie. Na základe vyššie spomenutých faktov konštatujeme, že je lepšie, ak programátor manažuje kopírovanie pamäte medzi CPU a GPU explicitne.

Po všetkých vykonaných úpravách implementáciu Kernel shared threads nevieme ďalej vylepšiť a použijeme ju ako referenčné riešenie pre výpočet konvolúcie 1:N:N (aj 1:1:1). Uprednostníme ju pred implementáciou Kernel shared managed, ktorá dosahuje lepši výpočtový čas, ale iba preto, lebo kopírovanie výsledkov z CPU na GPU posunie na neskôr. Prejdeme preto na implementáciu konvolúcie N:M:M.

4.5.3 Konvolúcia N:M:M

Pri implementácii konvolúcie M:N:N použijeme väčšinu vylepšení, ktoré sme zaviedli pri implementácii 1:N:N, menovite memory pool pitched a memory pool pinned, pridelovanie pamäte z memory pool pitched pre výstup a použitie prúdov a kernel. Kernel pre konvolúciu N:M:M sa nebude v mnohom odlišovať oproti kernelu pre konvolúciu 1:N:N, úpravy budú minimálne. Podstatný rozdiel bude v príprave dát pre kernel a ich následnom spracovaní.

Použitím neupraveného kernelu pre konvolúciu 1:N:N by sme pre každý jeden vstup museli vypočítvať konvolúciu a následne výstupy z konvolúcie sčítvať, aby sme dostali konečný výsledok pre jednu skupinu konvolučných jadier. Takýto postup používame pri CPU implementácii, ale je veľmi ďaleko od optimálneho, lebo pre každý výstup konvolúcie potrebujeme pamäť pre výstup a následne musíme tieto výstupy sčítvať, aby sme dostali celkový výsledok. Pri GPU implementácii použijeme iný postup, v ktorom potrebujeme iba pamäť pre výstup. Túto pamäť inicializujeme nulovými hodnotami a budeme v nej akumulovať výsledky z jednotlivých výpočtov konvolúcie. Takto ušetríme

pamäť a vyhneme sa sčítaniu. Aby sme nemuseli inicializovať pamäť nulami, tak pri výpočte konvolúcie pri prvom vstupe nebudeme hodnoty pripočítavať, ale rovno ich priradíme. Kernel sa zmení v tom, že namiesto priradenia hodnoty bude hodnotu pripočítavať k už existujúcej, čo znamená, že pri ukladaní výsledku sa bude pristupovať ku globálnej pamäti na dvakrát. Raz, aby sme získali hodnotu z globálnej pamäte a druhýkrát, aby sme uložili výsledok po sčítaní. Zaujímavosťou je, že lepší čas sme dostali vtedy, ak sme hodnotu načítali z globálnej pamäte do premennej, túto hodnotu sčítali s výsledkom v rámci premennej a uložili naspäť do globálnej pamäte, ako keď sme priamo nad globálnou pamäťou použili operátor +=

Vytvoríme spolu dve GPU implementácie. Prvá bude synchronná, bežiaci v jednom vlákne, bez použitia prúdov a druhá bude jej opakom. Taktiež sa budú líšiť spôsobom prípravy dát a ich spracovaním. Dôvodom pre dve implementácie je určitá dávka zvedavosti. Chceme zistiť odpoveď na otázku, o koľko dokáže zlepšiť výkon použitie prúdov. Pri konvolúcii 1:N:N použitie prúdov nepridalo príliš na výkone, lebo väčšinu času sa kopírovali dáta z GPU na CPU. Teraz budeme viac počítat' a menej kopírovať výstupy. Pre ilustráciu, nech máme 10 obrazov a jednu skupinu konvolučných jadier. V skupine je spolu 10 konvolučných jadier. Operáciu konvolúcie vykonáme 10-krát a iba raz budeme kopírovať výstup z GPU na CPU. Pri konvolúcii 1:N:N, už pri jednom vstupnom obraze a 10 konvolučným jadrám máme 10 výstupov, ktoré musíme kopírovať.

Ako sme napísali vyššie, implementácie sa budú líšiť prípravou a spracovaním dát. Použitie memory pool pitched je pre implementácie rozhodujúce, lebo budeme obmedzený ich veľkosťou. Zjednodušene popíšeme prvú implementáciu a zdôraznime rozdiely oproti druhej implementácii. V tejto implementácii budeme potrebovať pre výstup iba jednu pamäť z memory pool pitch pre výstup. Zvyšok môžeme použiť pre vstupy. Preto rozdelíme vstup(obrazy) do množín o veľkosti memory pool pitched pre vstup + memory pool pitched pre výstup - 1. Ďalej pracujeme s každou množinou samostatne. Vstupy z množiny nakopírujeme do pamäte z memory poolov. Ak máme vstupy raz v pamäti zariadenia, prejdeme všetky skupiny konvolučných jadier. Pri jednej skupine konvolučných jadier vyberieme konvolučné jadrá patriace vstupom a pre každý vstup vypočítame konvolúciu. Jedná sa potom o konvolúciu 1:1:1. Takto dostaneme medzivýsledok pre jednu skupinu konvolučných jadier. Aby sme dostali celkový výsledok, tieto medzivýsledky musíme sčítat'. Počet medzivýsledkov pre jednu skupinu konvolučných jadier je rovný celkovému počtu množín. Výhoda tohto prístupu je, že vstup sa kopíruje z CPU do GPU pamäte iba raz, stačí

nám jedna pamäť pre výstup na GPU a výkon neovplyvňuje celkový počet skupín konvolučných jadier. Nevýhodou je kopírovanie medzivýsledkov a ich nutné sčítanie po výpočte konvolúcie. Tento prístup je vhodný, ak je počet vstupov menší alebo rovný ako veľkosť memory pool pitched pre vstup $(10) + \text{memory pool pitched pre výstup } (20) - 1$. V našej implementácii je toto číslo 29.

Prejdeme k druhej implementácii. Na rozdiel od prvej implementácie bude používať prúdy a kernel bude vedieť pri jednom spustení vypočítať viac výstupov, podobne ako Kernel shared threads. O tom, ako použiť prúdy a rozdeliť prácu viacerým vláknam sme písali v kapitole 4.5.2, v tejto implementácii tomu nebude inak a nepovažujeme preto za potrebné niečo k tomu dodať. V tejto implementácii sa pozrieme na výpočet z iného pohľadu. Na prvej implementácii sa nám nepáčilo, že musíme vykonať dodatočný postprocessing dát vo forme sčítania medzivýsledkov. Použijeme taký postup, pri ktorom dostaneme po výpočte rovno výsledok. Pokiaľ sme v prvej implementácii rozdelili vstup do množín, v tejto rozdelíme skupiny konvolučných jadier do množín o polovičnej veľkosti memory pool pitched pre výstup. Nezáleží nám na tom, aké rozmery má konvolučné jadro skupiny v rámci množiny. Aby sme zhlukovali v množinách skupiny konvolučných jadier rovnakých veľkostí, z dôvodu, aby mohol kernel počítat' naraz viacero výstupov, tak pred vytvorením množín usporiadame skupiny konvolučných jadier podľa ich šírky. V rámci množiny budeme postupne prechádzať všetkými vstupmi. Vstup sa nakopíruje z CPU do GPU pamäte, vypočíta sa konvolúcia pre všetky skupiny konvolučných jadier v množine a výsledok sa ukladá do memory pool pitched pre výstup. Po prechode všetkými vstupmi máme hotový výsledok pre toľko skupín konvolučných jadier, koľko je veľkosť množiny. Problém môže spočívať v tom, že pri výpočte ďalšej množiny sa vstup musí opäť nakopírovať z CPU do GPU pamäte. Neprekáža nám to, lebo sme uložili podmienku, aby sa vstup ukladal v pinned memory, čo nám umožní rýchle asynchrónne kopírovanie, zatiaľ čo prebieha výpočet. Za nevýhodu tejto implementácie možno považovať časté kopírovanie vstupov pri veľkom počte skupín konvolučných jadier, ale tento nedostatok vyváža skutočnosť, že kopírujeme z GPU na CPU hotový výsledok a vyhneme sa postprocessingu.

Postúpime k testovaniu vytvorených implementácií. Prebrali sme kernely z implementácie konvolúcie 1:N:N a mierne sme ich upravili, ako bolo napísané vyššie. Nebude použitá metrika celkový čas, tej sa budeme venovať samostatne, keď budeme porovnávať medzi sebou výsledky prvej a druhej implementácie. V Tabuľka 20 budú kernely, ktoré spracúvajú jeden výstup, ako je to v prípade prvej implementácie (Kernel

shared multi) a v Tabuľka 21 budú kernely (Kernel shared threads multi), ktoré dokážu naraz spracovať viacero výstupov, ako pri druhej implementácii. Tento počet je maximálne 10, čo je nami určená polovičná veľkosť memory pool pitched pre výstup.

Tabuľka 20: Kernel shared multi

Šírka filtra	Rozmer bloku	Rozmer dlaždice	Čas trvania kernelu(μ s)	Pamäťová priepustnosť(GB/s)	Výpočtová priepustnosť(GFLOP/s)
Kernel shared incomplete block					
Rozmer subdlaždice: 2x2					
1	32x16	32x16	311,904	83,66	22,46
3	32x16	31x15	299,264	100,92	146,43
5	32x16	30x14	308,896	98,67	381,67
7	32x32	29x29	353,44	93,99	672,78
Kernel shared full block					
Rozmer subdlaždice: 3x3					
9	32x8	29x13	409,088	139,16	918,02
11	32x8	28x16	429,408	135,24	1303,82
13	32x6	x	480,416	124,61	1633,92
15	32x8	x	564,256	104,84	1841,55

Pre konvolučné jadrá šírky 9 a 11 je lepšie použiť Kernel shared full block so subdlaždicou 3x3 ako Kernel shared incomplete block so subdlaždicou 3x3, ktorý bol použitý v konvolúcii 1:N:N. Zvýšil sa počet prístupov ku globálnej pamäti, čo sa podpísalo aj na výpočtovom čase. Najviac je to viditeľné pri konvolučných jadrách šírky 1, 3 a 5, ktoré trvajú približne o 100 μ s dlhšie ako pri konvolúcii 1:1:1. Je to cena, ktorú sme ochotní podstúpiť, aby sme sa vyhli postprocessingu a kopírovaniu pamäte.

Tabuľka 21: Kernel shared threads multi

Šírka filtra	Počet výstupov	Čas trvania kernelu(μ s)	Priemerný čas kernelu pre jeden výstup(μ s)	Pamäťová priepustnosť (GB/s)	Výpočtová priepustnosť (GFLOP/s)
1	10	2065,504	206,5504	88,43	33,91
3	10	2190,144	219,0144	95,64	200,09
5	10	2236,256	223,6256	91,61	527,2
7	10	2350,592	235,0592	95,85	1011,61
9	10	4741,568	474,1568	93,43	792,02
11	10	3349,696	334,9696	134,77	1696,48
13	10	4325,888	432,5888	103,57	1814,57
15	10	5499,776	549,9776	80,97	1889,36

Ak kernel spracováva viacej výstupov, vo väčšine prípadov je rýchlejší ako kernel, ktorý počíta jeden výstup, až na konvolučné jadro šírky 9. Najvýraznejšie zlepšenie pozorujeme pri malých konvolučných jadrách (1,3,5,7). Za zmienku stojí fakt, že pri väčších konvolučných jadrách (7 a vyššie) počítame naraz maximálne stanovený počet výstupov. Pri konvolúcii 1:N:N sme museli maximálny počet výstupov pri širších konvolučných jadrách obmedziť. Namerané časy je možné dosiahnuť za podmienok, že neprebíha počas behu kernelu prenos dát. Ak sa prenášajú dáta z CPU do GPU, nie je podstatné v ktorom smere, vyžaduje si to určitú réžiu a čas trvania kernelu sa predĺži o 10-20%.

Ďalej budeme porovnávať prvú a druhú implementáciu konvolúcie N:M:M. Porovnávané budú na základe celkového času. Vytvoríme testovacie scenáre, ktoré sa líšia počtom vstupných obrázkov -parametrom N. N bude 10, 29 (počet pamäte pre vstup pri implementácii 1), 50, 96 a 128. Šírka konvolučného jadra bude v každej skupine konvolučných jadier tri. Počet skupín konvolučných jadier bude 10, 20, 30, 40 a 50. Predpokladáme, že obraz na vstupe bude uložený v pinned memory, kvôli rýchlemu kopírovaniu z CPU do GPU. Okrem porovnania implementácií budeme sledovať, ako vplýva parameter N a M na celkový výpočtový čas a hľadať závislosti medzi týmito veličinami. Čas je meraný v μ s.

Tabuľka 22: Prvá implementácia, Kernel shared multi

Počet vstupov(N)	Počet skupín konvolučných jadier(M)				
	10	20	30	40	50
10	45618,5	83923,9	121783	158475	216633
29	117785	212516	306154	401675	495766
50	222185	402751	620632	837885	1029190
96	443987	925696	1372030	1823460	2309480
128	620640	1207120	1802080	2400500	3022710

Tabuľka 23: Druhá implementácia, Kernel shared threads multi

Počet vstupov(N)	Počet skupín konvolučných jadier(M)				
	10	20	30	40	50
10	31383,9	53773,3	76895,9	99535,5	146054
29	75219,6	141790	208611	275610	364837
50	123943	237407	354439	469259	606739
96	229911	452439	672845	893805	1136250
128	304195	599219	891979	1187430	1506730

Medzi celkovým časom a počtom skupín konvolučných jadier je lineárny vzťah. Pri druhej implementácii sme sa mohli obávať, že častý prenos vstupov z CPU na GPU ovplyvní výsledok, ale nebolo tomu tak. Pozorujeme lineárny vzťah aj medzi časom a počtom vstupov

v druhej implementácii. U prvej implementácie to z vykonaných testov nevieme usúdiť. Ak porovnáme samotné implementácie, pre počet vstup 10 je druhá v priemere o 53 % rýchlejšia, pre 29 o 46 %, pre 50 o 74 %, pre 96 a 128 o 101 %. Pre nás to znamená, že už pre malý počet vstupov má zmysel použiť druhú implementáciu, pre väčší počet vstupov je až dvojnásobne rýchlejšia. Druhú implementáciu, alebo Kernel shared threads multi, budeme používať ako finálnu pre riešenie konvolúcie N:M:M.

5 Experimentálne zhodnotenie

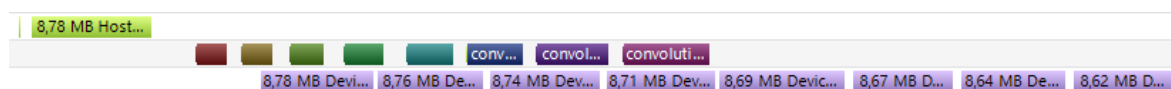
Po dokončení a otestovaní funkčnosti GPU implementácií prejdeme k ich experimentálnemu porovnaniu s existujúci implementáciami. Nebudeme ich porovnávať na základe vypočítaných výsledkov, lebo by vo väčšine prípadov boli rovnaké. Výsledky sa môžu odlišovať iba hodnotami hraničných bodov, lebo naša implementácia používa iný spôsob výpočtu krajných bodov ako existujúce implementácie. Naša implementácia používa orezanie, pričom sa zmenší výstupný obraz, v závislosti od veľkosti konvolučného jadra. Existujúce implementácie budú taktiež používať orezanie, ale výsledný obraz nebude zmenšený. Pri porovnaní implementácií sa budeme snažiť dosiahnuť rovnaké podmienky, za akých funguje naša implementácia, aby porovnanie bolo čo najviac objektívne. V testoch vykonaných v kapitole 3 bol použitý ako údajový typ prenášajúci informáciu o pixeli unsigned char. V našej implementácii ale počítame s údajovým typom float, preto musíme zmeniť aj typ, s ktorým počíta existujúca implementácia na float.

Ako existujúcu CPU implementáciu sme vybrali implementáciu z knižnice OpenCV o ktorej môžeme tvrdiť, že je špičková. Jedná sa o konvolúciu 1:1:1. Budeme ju porovnávať s implementáciou Kernel shared threads, ktorá je určená pre konvolúciu 1:N:N, ale použijeme ju ako konvolúciu 1:1:1. V našej implementácii používame rôzne memory pooly. Aby sme túto skutočnosť čiastočne vyvážili, pred výpočtom konvolúcie zväčšíme výstup na požadovanú veľkosť. Pri oboch implementáciách vykonáme 100 replikácií pre každú šírku konvolučného jadra a z nich dostaneme priemer pre celkový čas výpočtu. Na vstupe máme čiernobiely obraz o rozmeroch 1920x1200.

Tabuľka 24: Porovnanie výsledkov CPU implementácie z OpenCV s implementáciou Kernel shared threads

Šírka filtra	CPU OpenCV implementácia(μ s)	Kernel shared threads(μ s)	Čas trvania kernelu(μ s)	Pomer medzi časom CPU a GPU implementácie
1	1451,28	2751,26	201,888	0,527
3	3818,11	2754,51	203,552	1,386
5	8442,32	2780,5	205,632	3,036
7	14827,8	2846,15	293,728	5,210
9	23432,7	2841,74	355,392	8,246
11	33901,6	2929,55	444,224	11,572
13	38140,4	3081,95	496,576	12,375
15	37666,3	3156,85	589,888	11,932

Naša implementácia je, až na konvolučné jadro šírky 1, porovnateľne rýchlejšia ako existujúca CPU implementácia. Ak by sme brali do úvahy iba výpočet konvolúcie, bez kopírovania pamäte medzi CPU a GPU, už pri konvulčnom jadre šírke 3 je výpočet konvolúcie 17-násobne a pri konvulčných jadrách šírky 11 a 13 až 75-násobne rýchlejší ako pri CPU implementácii. Pre koncového používateľa nie je tento fakt podstatný a zaujíma ho iba, za koľko dostane želaný výstup. Situácia sa zlepši, ak nebudeme pre každé konvulčné jadro načítavať vstupný obraz z CPU do GPU, ale načítame ho iba raz, pri začiatku výpočtu a použijeme ho pri výpočte výstupov pre všetkých 8 konvulčných jadier. Výpočet je vizualizovaný na Obrázok 21. Potom dostaneme pre GPU implementáciu celkový čas 8166,09 μ s a pre CPU implementáciu 156961 μ s, čo je až 19-násobné zrýchlenie. V Tabuľka 17 sú výsledky pre konvolúciu 1:N:N, kde jeden kernel počíta pre jeden vstup viacero výstupov. Pri takomto prístupe je celkový čas aj pre konvulčné jadro šírky 1 lepší ako pri CPU implementácii. Mohli by sme takto pokračovať ďalej, ale je jasné, že so zvyšujúcim sa počtom vstupov a konvulčných jadier by bola naša implementácia v porovnaní s CPU implementáciou rýchlejšia a efektívnejšia. Pokračujme ďalej k porovnaniu s existujúcou GPU implementáciou.



Obrázok 21: Vizualizácia výpočtu

Ako existujúcu GPU implementáciu sme vybrali implementáciu OpenCL z knižnice OpenCV. Ide o konvolúciu 1:1:1. Údajový typ, použitý pre prenos informácie o pixely zmeníme na float. Musíme vytvoriť rovnaké podmienky, za akých pracuje implementácia Kernel shared threads pri konvolúcii 1:1:1. Pre vstupný obraz máme na GPU alokovanú pamäť, ale musí dôjsť ku kopírovaniu vstupu z CPU do GPU. Potom prebehne výpočet kernelu a na záver skopírujeme výsledok z GPU na CPU. Za týchto podmienok sme vykonali 100 replikácií a výsledky porovnania možno vidieť v tabuľke nižšie.

Tabuľka 25: Porovnanie GPU OpenCL implementácie z OpenCV s implementáciou Kernel shared threads

Šírka filtra	GPU OpenCL implementácia(μ s)	Čas trvania kernelu(μ s)	Kernel shared threads(μ s)	Čas trvania kernelu(μ s)
1	5341,74	210,581	2751,26	201,888
3	5515,83	411,889	2754,51	203,552
5	5711,48	610,099	2780,5	205,632
7	5987,71	809,555	2846,15	293,728
9	6543,02	810,613	2841,74	355,392
11	6870,59	1291,786	2929,55	444,224
13	7247,28	1557,358	3081,95	496,576
15	7791,38	1894,832	3156,85	589,888

Pre konvolučné jadrá šírky 1 a 3 je GPU implementácia pomalšia ako jej CPU náprotivok. Dôvodom je, ako môžeme tušiť, kopírovanie vstupov a výstupov. Kopírovanie vstupu z CPU na GPU trvá v priemere 2086,22 μ s a z GPU na CPU 2207,96 μ s. Rýchlosť kopírovania u OpenCL sa pohybuje v intervale od 4600 MB/s do 5000 MB/s. Naša implementácie používa pre kopírovanie pinned memory, ktorej rýchlosť kopírovania sa pohybuje okolo hodnoty 12000 MB/s. Ak porovnáme celkový čas výpočtu, potom je naša implementácia v priemere 2-násobne rýchlejšia ako OpenCL implementácia. Naša implementácia získala najväčšiu výhodu oproti OpenCL implementácii rýchlym kopírovaním pamäte medzi CPU a GPU. Ak by sme zanedbali tento fakt brali v úvahu iba čas výpočtu na GPU, tak má naša implementácia stále navrch a to pri každej šírke konvolučného jadra. Najmenší je rozdiel medzi kernelom pre konvolučné jadro šírky 1(8 μ s). Pri šírke 3 je náš kernel 2-násobne rýchlejší, pri 5 3-násobne, pri 7 2,75-násobne, pri 9 2,25-násobne a pri zvyšných približne 3-násobne. Myslíme si, že 2 až 3-násobne vyššia rýchlosť kernelov našej implementácie pre konvolúciu 1:1:1 je pre nás vynikajúcim výsledkom. Za úspech by sme osobne považovali aj zlepšenie o 50%.

Pri porovnávaní bol používaný obraz o rozmeroch 1920x1200. Teraz vykonáme experiment, v ktorom budeme porovnávať celkový čas medzi implementáciami s obrazmi rôznych rozmerov. Vstupný obraz bude z CPU na GPU prenášaný iba raz, na začiatku výpočtu. Konvolučné jadrá budú mať šírku 1, 3, 5, 7, 9, 11, 13 a 15. Vykonali sme 100 replikácií. Výsledok experimentu je možné vidieť v Tabuľka 26.

Tabuľka 26: Porovnanie GPU OpenCL implementácie s Kernel shared threads pri rôznych rozmeroch obrazu

Rozmer obrazu	GPU OpenCL implementácia(μs)	Kernel shared threads()
300x300	3923,11	1148,81
600x700	7558,82	2121,93
1920x1200	25302,7	8186,32
2000x2000	41486,3	13151,1

Naša implementácia je pri tomto experimente v priemere viac ako 3-násobne rýchlejšia. Odôvodňujeme to nielen rýchlym kopírovaním vstupov a výstupov, ale aj schopnosťou našej implementácie súčasne spúšťať kernely a kopírovať dáta z CPU do GPU a opačne.

Posledným experimentom, ktorý uskutočníme, bude mierna modifikácia predchádzajúceho experimentu. Na vstupe nebude iba jeden obraz, ale všetky 4. Po vykonaní 100 replikácii sme dostali pre implementáciu OpenCL 76837,3 μs a pre našu implementáciu 20847,9 μs. Musíme zdôrazniť, že všetky namerané výsledky bolo možné dosiahnuť iba za podmienok, že vstupný obraz je uložený v pinned memory a výstupný obraz sa kopíroval taktiež do pinned memory. Ak by sme pre posledný experiment nahradili pinned memory bežnou pamäťou dostali by sme sa na čas 65883,7 μs, čo je stále lepší výsledok ako pri implementácii od OpenCL. Na základe nameraných výsledkov hodnotíme z celkového hľadiska našu implementáciu konvolúcie 1:1:1 ako efektívnejšiu a rýchlejšiu ako implementáciu od OpenCL.

Implementácie konvolúcie 1:N:N a N:M:M nemáme ako porovnať s existujúcimi implementáciami v OpenCV. Pri hľadaní existujúcich implementácií týchto typov výpočtov konvolúcií sme našli knižnicu cuDNN¹⁰, ktorá sa venuje strojovému učeniu a obsahuje konvolúciu 1:N:N a M:N:N. Táto knižnica je ale veľmi robustná a komplexná, sama o sebe je základom pre frameworky určené na strojové učenie a aj keď sme vykonali všetky počiatočné nastavenia s rôznymi variáciami pred spustením konvolúcie, kernely počítajúce konvolúciu boli veľmi pomalé. Ako príklad uvedieme konvolúciu s konvolučným jadrom šírky 3 a vstupným obrazom o rozmeroch 1920x1200. Priemerný čas trvania kernelu bol

¹⁰ <https://developer.nvidia.com/cudnn>

3800 μ s. Náš kernel dosiahol pri rovnakom vstupe čas 200 μ s. Rozdiel je priepastný. Predpokladáme, že sme knižnicu nedostatočne naštudovali a nastavili sme nesprávne parametre pri spustení kernelu. Nakoľko detailné preštudovanie a otestovanie danej knižnice presahuje rámec tejto práce, došli sme k presvedčeniu, že nemá zmysel porovnávať našu implementáciu s implementáciou od cuDNN a preto ich ani porovnávať nebudeme.

Záver

Na záver konštatujeme, že cieľ, ktorý sme si vytýčili, sa nám podarilo dosiahnuť. Hlavným cieľom bolo využitie GPU paralelizovaných výpočtov na generovanie umelých dát. Pre jeho splnenie sme museli vybrať metódu vhodnú na generovanie umelých dát, implementovať vybranú metódu a následne ju experimentálne porovnať s existujúcou implementáciou.

Vybrali sme metódu diskkrétnej dvojrozmernej konvolúcie, ktorá poskytuje veľkú flexibilitu pri tvorbe umelých dát len malou zmenou vstupných parametrov. Počas jej implementácie na GPU sme použili rôzne prostriedky a techniky, ktorými sa nám podarilo posunúť výkon až na hardvérové obmedzenia zariadenia. Medzi najvýznamnejšie použité techniky radíme sofistikované načítanie opakovane používaných dát z globálnej do zdieľanej pamäte, použitie pamäte konštánt pre dáta konštantné počas celej doby výpočtu a redukciu počtu transakcií ku globálnej pamäti. Počas implementácie sme zistili, že nás obmedzuje rýchlosť prenosu dát medzi CPU a GPU. Zabezpečili sme preto, aby kopírovanie dát bolo čo najrýchlejšie, aby prenos dát prebiehal asynchrónne s výpočtom a vytvorili sme jednoduchý, ale účelný systém správy pamäte, aby nedochádzalo k častým alokáciám a dealokáciám pamäte.

Pri experimentálnom zhodnotení sme zistili, že naša GPU implementácia predstihne špičkovú CPU implementáciu z knižnice OpenCV, pri dávkovom spracovaní je v priemere 19-násobne rýchlejšia. Porovnaním s existujúcou GPU OpenCL implementáciou z OpenCV sme pozorovali 2-3 násobné zlepšenie na úrovni kernelu a celkové 3-násobné zlepšenie, čo považujeme za výborný výsledok v prospech našej implementácie.

Do budúcnosti vidíme možnosť vylepšenia implementácie v pridaní podpory pre konvolučné jadrá širšie ako 15 a konvolučné jadrá atypických rozmerov, kde je rozmer párne číslo (napríklad 11x6). Môžeme skúsiť namiesto jednoduchej presnosti počítať v polovičnej a zmenšiť tým objem prenášaných dát medzi CPU a GPU. Ďalším zaujímavým problémom by bol prenos implementácie do prostredia, v ktorom je viacero grafických kariet. Vzhľadom na to, že konvolúcia $N:M:M$ nemá využitie iba pri predspracovaní obrazu, ale aj v konvolučných neurónových sieťach, v budúcnosti by sme mohli vyskúšať použiť našu implementáciu ako výpočtový engine pre tieto siete.

Zoznam použitej literatúry

- [1] blogs.nvidia.com: What Is CUDA? [online] [cit 20.2.2018]. Dostupné na <<https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/>>
- [2] Wikipedia, the free encyclopedia: CUDA [online] [cit 21.2.2018]. Dostupné na <<https://en.wikipedia.org/wiki/CUDA>>
- [3] docs.nvidia.com: CUDA C Programming Guide [online] [cit 21.2.2018] Dostupné na <<http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>>
- [4] CUDA memory model [online] [cit 24.2.2018]. Dostupné na <<https://www.3dgep.com/cuda-memory-model/>>
- [5] Computer Vision and Geometry Group: CUDA Memory Architecture [online] [cit 24.2.2018].
Dostupné na <https://www.cvg.ethz.ch/teaching/2011spring/gpgpu/cuda_memory.pdf>
- [6] TOMORI Zoltán, NIKOROVIČ Matej: Počítačové videnie v praxi [online] [cit 25.2.2018].
Dostupné na <http://home.saske.sk/~tomori/Downloads/Poc_videnie/PV_2016.pdf>
- [7] Ftáčnik Milan: Pokročilé techniky predspracovania obrazu [online] [cit 25.2.2018].
Dostupné na <www.sccg.sk/~ftacnik/PV3.doc>
- [8] PETRÍK Jozef, VALKOVIČ Juraj, VEREŠPEJ Peter, Ján VALO :Predspracovanie obrazu [online] [cit 22.2.2018]. Dostupné na <neuron.tuke.sk/pluchta/Pocitacove%20Videnie/Prednasky/Predsp.doc>
- [9] matlab.fei.tuke.sk : Predspracovanie obrazu [online] [cit 25.2.2018]. Dostupné na <http://matlab.fei.tuke.sk/wiki/index.php?title=Pedspracovanie_obrazu#Jasov.C3.A9_transform.C3.A1cie>
- [10] Wikipedia, the free encyclopedia: Adaptive histogram Equalization [online] [cit 26.2.2018]. Dostupné na <https://en.wikipedia.org/wiki/Adaptive_histogram_equalization>
- [11] Predspracovanie [online] [cit 26.2.2018]. Dostupné na <<http://dip.sccg.sk/pedspra/pedspra.htm>>
- [12] WHITEHEAD Nathan, FIT-FLOREA Alex Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs [online] [cit 10.3.2018]. Dostupné na

<http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf?1g5vfjd7edlc-e4IDz4_wWl3_-3j6y_W6XJN5iQWWXJPm8EGFwjGiacervLi7TBxluCsWWVuvlRgVVvAIXA_D54YFOoZMljc9f6MUNHKAwKRQWQ1O8gbG32vMHzx1y6ACJh2utHaKyZ2yMQuvD>

3j6y_W6XJN5iQWWXJPm8EGFwjGiacervLi7TBxluCsWWVuvlRgVVvAIXA_D54YFOoZMljc9f6MUNHKAwKRQWQ1O8gbG32vMHzx1y6ACJh2utHaKyZ2yMQuvD>

[13] Šikudová E., Černeková Z., Benešová W., Haladová Z., Kučerová J.: Počítačové videnie: Detekcia a rozpoznávanie objektov, 2013, 39-47

[14] blogs.nvidia.com.: How to Implement Performance Metrics in CUDA C/C++ [online] [cit 25.3.2018]. Dostupné na < <https://devblogs.nvidia.com/how-implement-performance-metrics-cuda-cc/>>

Zoznam príloh

Príloha 1: CD obsahujúce zdrojové kódy aplikácie, používateľskú príručku, reporty z nástroja NVIDIA Nsight a prácu v elektronickej podobe (PDF)