VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

# DIPLOMOVÁ PRÁCE

Brno, 2016 Bc. Marek Bielczyk


# VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

## FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

## ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS


# ALGORITMY ČÍSLICOVÉHO ZPRACOVÁNÍ OBRAZU NA GRAFICKÝCH KARTÁCH
THE ALGORITHMS OF DIGITAL IMAGE PROCESSING ON GRAPHICS CARDS

**DIPLOMOVÁ PRÁCE**
MASTER'S THESIS


**AUTOR PRÁCE** — **Bc. Marek Bielczyk**
AUTHOR

**VEDOUCÍ PRÁCE** — **Ing. Jiří Přinosil, Ph.D.**
SUPERVISOR



**BRNO 2016**

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

# Diplomová práce

magisterský navazující studijní obor **Telekomunikační a informační technika**
Ústav telekomunikací

***Student:*** Bc. Marek Bielczyk ***ID:*** 145971
***Ročník:*** 2 ***Akademický rok:*** 2015/16

**NÁZEV TÉMATU:**

## Algoritmy číslicového zpracování obrazu na grafických kartách

**POKYNY PRO VYPRACOVÁNÍ:**

Prostudujte možnosti využití grafických karet při zpracování obrazového signálu, přičemž se zaměřte zejména na technologie CUDA a OpenCL. Na základě získaných poznatků zvolte algoritmy zpracování a analýzy obrazu vhodné pro implementaci pomocí těchto technologií. Vybrané algoritmy následně implementujte a proveďte srovnání výkonu a uživatelského komfortu za použití jednotlivých technologií i se standardní CPU realizací.

**DOPORUČENÁ LITERATURA:**

[1] J. SANDERS; E. KANDROT: CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley, 2011.

[2] A. MUNSHI: The OpenCL specifications, Khronos Group, 2013.

***Termín zadání:*** 1.2.2016 ***Termín odevzdání:*** 25.5.2016

***Vedoucí práce:*** Ing. Jiří Přinosil, Ph.D.
***Konzultant diplomové práce:***

**doc. Ing. Jiří Mišurec, CSc.**, *předseda oborové rady*

## ABSTRAKT

Cílem práce je ukázat možnosti využití grafických karet při zobrazování obrazového signálu. Práce je zaměřena obzvlášť na technologie CUDA a OpenCL. V řešení se nejdříve zaměříme na samostatnou grafickou kartu a ukážeme si postupný vývoj jejich komponentů a následný projevený efekt ve výkonu grafické karty. Poté si ukážeme samotné technologie CUDA a OpenCL, a také ukázky z kódů s vysvětlením , co který kód způsobí. Výstupem práce je několik programů, definovaných pro obě technologie a pro oba vykonavatele (CPU vs GPU). Přínosem této práce je vidět rozdíly mezi vykonavateli a tím i poukázaní na správnou volbu při návrhu vlastních algoritmů.

## KLÍČOVÁ SLOVA

Grafická karta, GPU, GPGPU, CUDA, AMD APP, OpenCL, Grafický procesor, Paralelismus, Paralelní výpočty, Programování

## ABSTRACT

Purpose of this work is show possibility of using grapichs cart for imaging a video signal. This work is particularly focused on technology CUDA and OpenCL. The solution is first focused on graphics cart and show how has been changed components and how has been changed performaces of graphics cart. Then show CUDA and OpenCL technology itself, and show samples of codes with explain, what which code do. Output of this work is some programs, witch defined for both technology and for both procesors unit. Contribution of this work is show differents between procesors unit, witch can be used to right choose for design your own algorithm.

## KEYWORDS

Graphics cart, GPU, GPGPU, CUDA, AMD APP, OpenCL, Graphics processor unit, Parallelism, Parallel Computing, Programming

BIELCZYK, Marek *Algoritmy číslicového zpracování obrazu na grafických kartách*: diplomová práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2016. 64 s. Vedoucí práce byl Ing. Jiří Přinosil, Ph.D.

# PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Algoritmy číslicového zpracování obrazu na grafických kartách“ jsem vypracoval(a) samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor(ka) uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil(a) autorská práva třetích osob, zejména jsem nezasáhl(a) nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom(a) následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ..... .....

podpis autora(-ky)

# PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu Ing. Jiří Přinosil, Ph.D. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.


Brno ..... .....

podpis autora(-ky)


Faculty of Electrical Engineering
and Communication
Brno University of Technology
Purkynova 118, CZ-61200 Brno
Czech Republic

http://www.six.feec.vutbr.cz

## PODĚKOVÁNÍ

Výzkum popsaný v této diplomové práci byl realizován v laboratořích podpořených z projektu SIX; registrační číslo CZ.1.05/2.1.00/03.0072, operační program Výzkum a vývoj pro inovace.

Brno . . . . . .

. . . . . .

podpis autora(-ky)


EVROPSKÁ UNIE
EVROPSKÝ FOND PRO REGIONÁLNÍ ROZVOJ
INVESTICE DO VAŠÍ BUDOUCNOSTI

# OBSAH

# SEZNAM OBRÁZKŮ

# SEZNAM TABULEK

# ÚVOD

Dnešní grafické karty jsou velmi výkonné a tento dnešní trend umožnil využít grafické karty, nejen k výpočtu zobrazení ať už ve 2D či 3D grafice, ale i k výpočtům obecných algoritmů. K tomu dnes využíváme již vytvořené knihovny. První se objevily grafické knihovny OpenGL a od společnosti Microsoft knihovna DirectX, která však zpočátku kulhala za knihovnou OpenGL. Později přišli se svými architekturami sami dva největší výrobci grafických procesorů. První byla společnost ATI s architekturou ATI Stream. Ta však byla zpočátku založena na uzavřeném programovacím jazyku. Pak přišla společnost Nvidia se svou architekturou CUDA. S příchodem této architektury došlo k přechodu u architektury ATI Stream na otevřený standard OpenCL.

K tomu všemu však předcházel dlouhý vývoj grafické karty a to nejen samotného grafického procesoru ale i ostatních komponentů grafické karty jako typ použitých pamětí, kvalita digitál-analogového převodníku a typ výstupního konektoru.

V této práci se v první polovině budeme zabývat samotnou grafickou kartou. Vývojem grafického procesoru, pamětí určených do grafických karet a ostatních komponentů grafické karty. Rozvedeme pojem GPGPU a co všechno se pod tímto API modelem skrývá, hlavně se zaměříme na architekturu CUDA a standard OpenCL. Po-té si na vybereme tři algoritmy s různou náročností a ty ve druhé části vytvoříme. Algoritmy budeme spouštět jak na centrálním procesoru tak i na grafickém procesoru s využitím paralelních technologií. Vznikne nám tím devět programů. Jejichž řádky jsou v práci popsané. Celý kód bude napsán v programovacím jazyku „c++“ s využitím grafické knihovny OpenCV pro načítaní obrázků a měření výsledků.

V závěru zapíšeme výsledky doby zpracování algoritmů a shrneme výsledky a na jejich základě bude odvoditelné, který procesor a kterou architekturu lze efektivně využít na daný výpočetní problém.

# 1 STAVBA GRAFICKÝ KARET

Grafická karta je nezbytná komponenta počítače, která se stará o grafické výpočty a výstup na monitor, nebo jinou zobrazovací jednotku (projektor, televize aj.). Dnes lze nalézt dva typy grafických karet buď integrované (většina kancelářských počítačů), kdy je grafický procesor integrován do severního můstku na základní desce, nebo externí, kdy komunikace probíhá přes slot PCI-Express na základní desce, dříve se externí grafické karty připojovaly přes slot AGP, ten se nyní již nepoužívá. Mezi důležité parametry grafické karty patří velikost a typ paměti, typ jádra, šířka sběrnice, frekvence grafického čipu, pamětí a počtu stream procesorů. Dalším důležitým kriteriem při výběru grafické karty jsou i požadované výstupy (HDMI, VGA, DVI, DisplayPort). [2]

## 1.1 Grafický procesor

Jinak také GPU je v informačních technologiích specializovaný procesor, který se nachází na grafické kartě. Tento specializovaný procesor má za úkol zajišťovat grafické výpočty nutné pro vykreslování dat v operační paměti na grafický výstup. U dnešních grafických karet se tento specializovaný procesor používá i pro jiné výpočty (např. kryptografii). GPU procesory jsou velmi rychlé až tak, že paměti nestačí poskytovat data. Tento problém způsobuje, že vzniká tzv. doba latence, řešením bývá využití rychlých cache paměti, které mají malou velikost. GPU procesory tak poskytují alternativu pro algoritmy, jenž mají podstatnou část totožných operaci a umožňují tak tyto operace provádět paralelně, tento typ operací se nazývá „stream processing.“ Toto umožňuje GPU pomocí mnoha hardwarových jader zpracovávat mnoho softwarových vláken, které provádějí synchronně totožné operace s různými daty tzv. „SIMT neboli single-instruction multiple-threads.“ Navíc může několik vláken sdruženě přistupovat do paměti současně tzv. „memory coalescing“. Dnešní GPU procesory běžně obsahují i stovky jader. [5] [9]

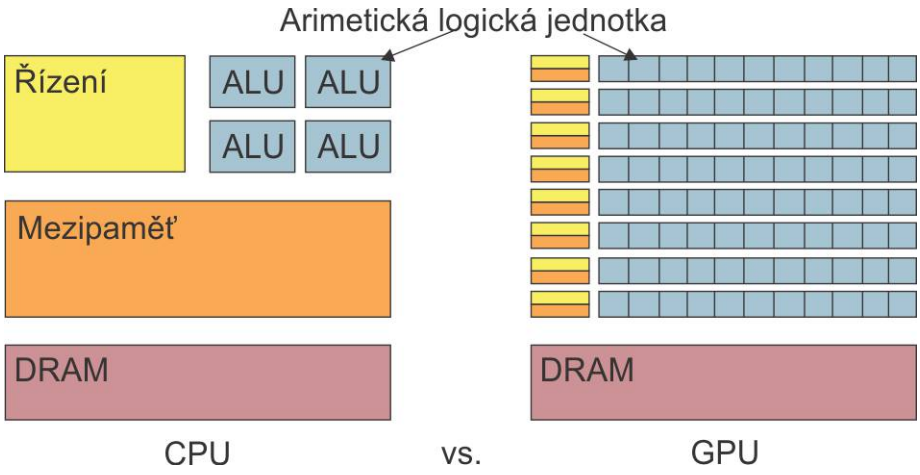
### 1.1.1 Stavba grafického procesoru

Ve stavbě GPU procesoru lze nalézt tyto komponenty: TMU, ROP, řadič pamětí aj. Grafický procesor je přímo určený na výpočty operací s pohyblivou desetinnou čárkou a při výpočtech grafických scén se stává specializovaným procesorem. Pro urychlení výpočtů používá několik obvodů:

- **Unifikované shadery** - používají se na upravené části čipu vyhrazeného pro práci shaderů. Tato oblast umí zpracovávat pixel, vertex, i geometry shader

úlohy, což nám umožňuje využít shadery optimálně (Dříve byla každá část čipu specializována pro daný shader.) [13]

- **řadič paměti** - zajišťuje komunikaci mezi grafickou pamětí a GPU.
- **jednotka TMU** (Texture mapping unit) - stará se o nanášení textur na objekty.
- **jednotka ROP** (Render Output unit) - zařizuje konečný výstup z grafické karty. [10]




Obr. 1.1: Rozdíl mezi CPU a GPU. [5]

### 1.1.2 Grafické akcelerátory a jejich vývoj

Výkonově je gpu procesor závislý na schopnosti výpočtu matematických operací s plovoucí čárkou a proto je pro tento záměr specializován. Navíc muže být do grafického procesoru implementováno operace, jenž umožňují pracovat se základními grafickými prvky a tak způsobit, že mohou být grafické objekty vykresleny rychleji, než by bylo možné je vykreslit pomocí centrálního procesoru počítače (jednotky CPU). Hlavní výhodou GPU je možnost využití některé z technik urychlení neboli akcelerace manipulace s grafickými daty, které by bez využití akcelerace byly výrazně pomalejší. V počátcích byli mezi nejběžnějšími operacemi 2D počítačové grafiky operace nazývané BitBLT, která byla obvykle realizována pomocí speciálního zařízení nazývané „blitter“ a operace, jenž sloužily pro kreslení geometrických tvarů (obdélníků, trojúhelníků, kruhů a oblouků). [10]

#### Rok 1970

Uvedený rok se vyznačoval využitím hardwarového mixování grafických a textových režimů, pozicování a zobrazení spritů a pro další efekty. Toho bylo dosaženo použitím

čipů ANTIC a CTIA, které byly použity v 8bitových počítačích Atari. ANTIC byl čip, který byl speciálně určen pro mapování textu a grafických dat do výstupu. Jeho konstruktér byl Jay Miner, který navrhl i čip pro počítač Commodore Amiga.

### Rok 1980

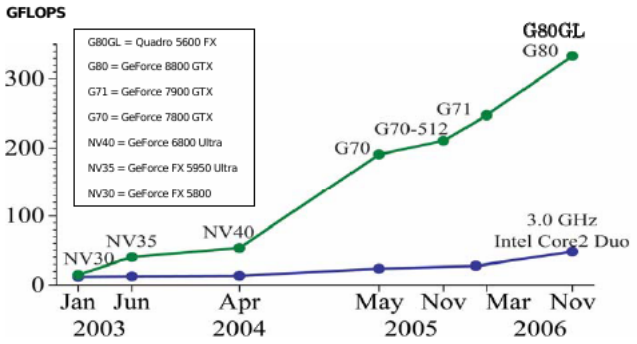
V tomto roce přišel první masově vyráběný počítač, jenž již obsahoval „blitter“ ve svém video hardwaru. Šlo o počítač Commodore Amiga. A taktéž přišel grafický systém 8514 od IBM a ten se stal jedním z prvních systémů PC videokaret provádějící jednoduchou 2D grafiku. Výjimečnost počítače Amiga spočívalo v tom, že obsahoval již plnohodnotný grafický akcelerátor , ten už obsahoval funkce pro generování grafiky a grafický procesor s svou primitivní sadou strojových instrukcí.

### Rok 1990

V roce 1990 přišel velký vzestup operačního systému Microsoft Windows, který způsobil zájem o vysoký výkon a vysoké rozlišení 2D bitmapové grafiky (což byla předtím jen doména unixových systému a OS Macintosh). Rozvíjející se trh umožnil výrobcům PC grafik se soustředit na vývoj hardwaru, který bude podporovat programové rozhraní Windows Graphics Device Interface (GDI). V roce 1991 vytvořila a představila společnost S3 Graphics první jednočip s 2D akcelerací pojmenovaný 86C911. Na tento čip následně vzniklo několik napodobenin. Později v roce 1995 mají všichni výrobci grafických čipu implementovanou podporu 2D akcelerace na svých čipech. Časem byly drahé víceúčelové grafické koprocesory nahrazeny specializovanými akcelerátory určenými pro prostředí Windows, které dosahovaly vyšších výkonů. V průběhu 90tých let se 2D akcelerace GUI dále rozvíjela a díky zlepšování výrobních podmínek došlo ke zvýšení úrovně integrace grafických čipu. Došlo k vývoji různých API pro různé úkoly pro příklad grafická knihovna Microsoft WinG pro Windows 3.X a její pozdější rozhraní DirectDraw pro hardwarovou akceleraci ve 2D her ve Windows 95 a novějších.

Také renderování 3D grafiky se začalo stávat čím dál běžnější v počítačových hrách a herních konzolách, což začalo zvyšovat zájem o hardwarovou 3D akceleraci. K masovému rozšíření došlo v páté generaci herních konzoly Nitendo 64 a PlayStation. První čipy s 3D akcelerací byly S3 ViRGE, ATI Rage a Matrox Mystyque. Jednalo se o předchozí generace 2D akcelerátorů s přidanými 3D vlastnostmi. Tyto čipy byly levné avšak ne příliš podařené. Mnohé z nich byly však díky své jednoduchosti implementace byly pinově kompatibilní s předchozími generacemi. Výkonové 3D grafické čipy se z počátku nacházeli pouze na samostatných kartách, kde měly za úkol pouze 3D akcelerační funkce (tzn. že neuměly 2D akceleraci). Mezi takové karty patří například 3dfx Vodoo. Další vývoj však umožnil integrovat 2D a 3D

akceleraci do jednoho čipu. Jedny z prvních takto vyrobených čipů lze nalézt na čipsetech Verite od Revditionu. Další technologie, která se objevila na počátcích 90tých




Obr. 1.2: Vývoj výkonu nVidia GPU vs. Intel CPU. [1]

let objevilo profesionální grafické API známe pod názvem OpenGL, které se stalo dominantním na PC. Největší předností OpenGL bylo rozšíření hardwarové podpory, přes to se koncem 90tých let mezi vývojáři her stalo velmi populární DirectX. Toto rozhraní, na rozdíl od OpenGL, bylo založené na striktně jednotné podpoře hardwaru (požadavek Microsoftu). Toto omezení učinilo, že byl DirectX méně populární, protože mnoho výrobců ve svých GPU poskytovalo svoje specifické vlastnosti, které rozhraní OpenGL dokázalo využít oproti DirectuX, který tak byl často o krok pozadu.

Později se však přístup Microsoftu změnil a začal více spolupracovat z vývojáři grafických karet, čímž se DirectX zaměřil na podporu konkretního hardwaru. První rozhraním byl Direct3D 5.0, který již schopen konkurovat hardwarově specifickým a proprietárním knihovnám. Avšak mnohem zajímavější se stal Direct 3D 7.0, který představil podporu hardwarového T&L (Transformace a osvětlení), což představilo významný krok ve 3D renderování. Tato technologie je předchůdce pozdějších jednotek „vertex“ a „pixel shader.“ První karta, která obsahovala tuto technologii, byla NVIDIA GeForce 256 (známá také jako NV10).

### 2000 až do současnosti

Na počátku tohoto období se k možnostem GPU přidaly i programovatelné jednotky „shader.“ Což umožnilo zpracovávat pixely nebo vertexy programem předtím, než dojde k jejich zobrazení. První takto pracující čip přišel od společnosti NVIDIA a jednalo se o GeForce 3. Avšak ani konkurenční společnost neusnula na vavřínech a tak v říjnu 2002 představila společnost ATI svůj Radeon 9700 (známý také pod názvem R300). Jednalo se o první čip, který podporoval Direct3D 9.0. Toto rozhraní

sebou přinesl pixelové a vertexové shadery, které mělo implementované smyčky a matematické operace s plovoucí řádovou čárkou což umožnilo využít model GPGPU.

## 1.2 Paměť grafické karty

Základním parametrem grafické karty je kapacita, která je udávaná dnes nejčastěji v GB. Dalšími základními parametry je Frekvence (udávaná v [Hz]), která udává počet cyklu za sekundu, CAS latency (udávaná v [CAS] neboli délce čekání), tento parametr udává jak dlouho trvá než se data objeví na pinech paměti po zadání adresy požadovaného sloupce, u tohoto parametru platí čím nižší je tato hodnota tím je paměť grafické karty rychlejší. Tento parametr se podílí na celkové odezvě každé paměti známou spíše pod názvem „memory timings“ (udávaná v [ns]). Posledním základním parametrem grafické karty je velikost sběrnice (udává se v [bit]), který značí kolik dat je možné dostat z paměti do grafické karty za jeden takt.

Jedním z parametrů, který nás na grafické kartě může zajímat, je celková propustnost (známe pod názvem „bandwidth“). Tento údaj lze ze základních parametrů spočítat a to když vynásobíme frekvenci a šířku sběrnice a nakonec toto číslo podělíme osmi. [4]

### 1.2.1 Typy pamětí

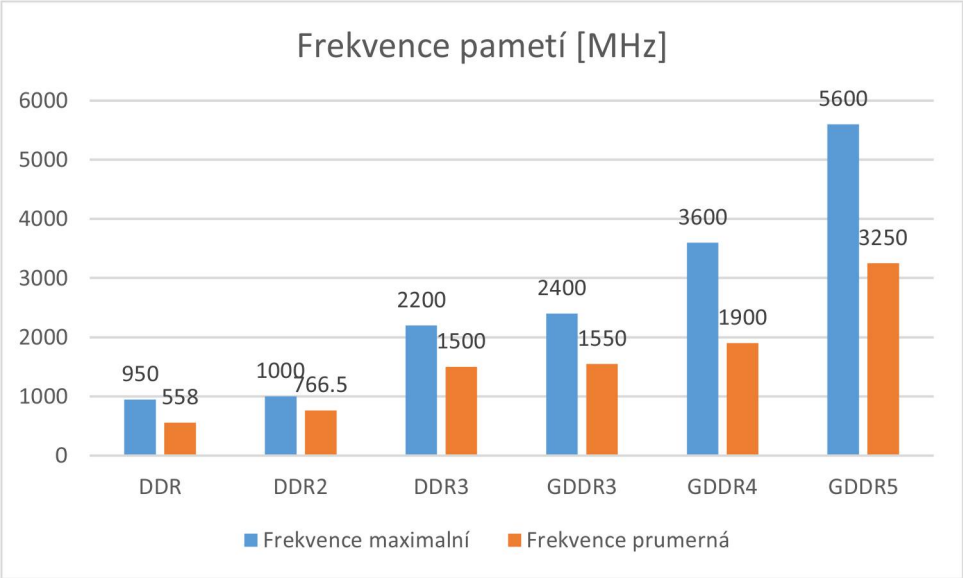
Dle použitých technologii lze paměti rozdělit takto:

#### DDR SDRAM

Tato technologie tvoří základ všech pozdějších pamětí. Tato technologie přenáší dva bity v jednom taktovacím impulzu, což značí že máme dvojnásobnou efektivní frekvenci. Bývaly napájeny napětím o hodnotě 2,4V až 2,6V (výjimkou mohou být modely určeny pro taktování nebo úsporu energie). Hodnoty parametru CAS latency u těchto pamětí byly CL2, CL2.5, CL3. Běžně používané frekvence u této technologie dosahují 166 až 950 MHz, což vytváří teoretickou propustnost 1,2 až 30,4 GB/s. Jedním ze zástupců této technologie lze jmenovat grafickou kartu Nvidia GeForce 5200 (ta používala 64bitovu sběrnici). [4]

#### DDR2

Jedná se o nástupce předchozího typu. Hlavním rozdílem oproti předchozí technologie je zvýšení na dvojnásobek taktovací rychlosti, což způsobilo snížení na čtvrtinu potřebné doby pro přenesení dat. Hodnoty napětí se u těchto karet také snížily a to od 1,2V až po 2,4V. Vzniklou nevýhodou této technologie bylo navýšení parametru




Obr. 1.3: Frekvence pamětí v různých typech pamětí. [4]

CAS latency a to na CL4 až CL7, kterou tato technologie kompenzuje kratší dobou cyklu. U těchto karet se setkáváme s efektivní frekvenci od 533 až 1000 MHz a propustností od 8,5 až do 16 GB/s. Zástupcem této technologie je například Nvidia GeForce GT 240 (sběrnice je opět 64bitová). [4]

### DDR3

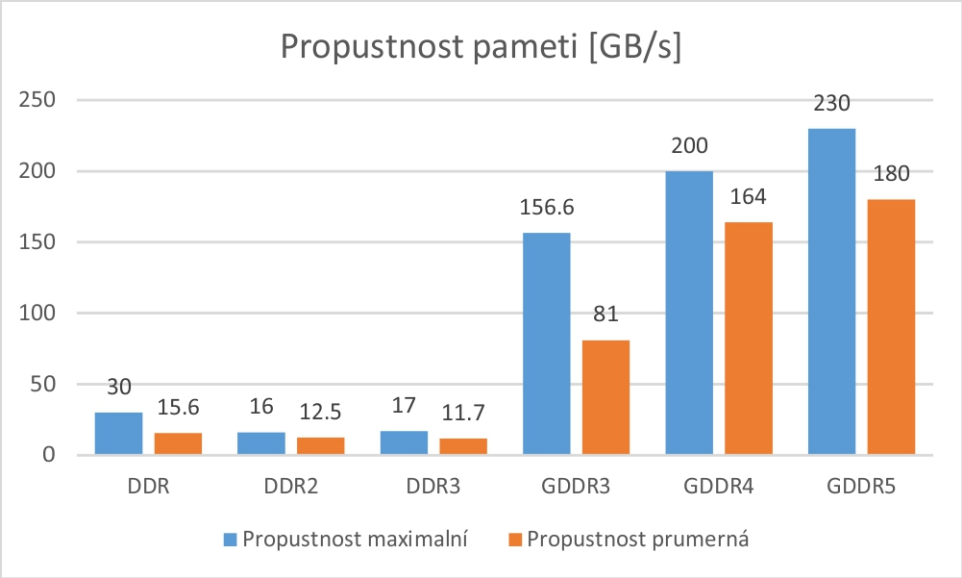
Opět došlo ke zdvojnásobení taktovací rychlosti. K provozu využívají napětí velikosti od 1.35V až do 1,5V. Tento typ pamětí je dnes stále hojně využíván jako operační paměť počítače. Opět došlo i ke zvýšení parametru CAS latency a to na hodnoty CL10 až CL15. V této technologii se využívá efektní frekvence 800 až 2200 MHz a hodnota propustnosti se nachází v intervalu od 6,4 až 17 GB/s. Zástupcem této technologie je například velmi známá grafická karta Nvidia GeForce 9800GT (používá 256bitovou sběrnici). [4]

### GDDR

Jedná se pouze o přejmenování již zmíněné technologie. Tento název patří pamětem DDR SDRAM. [4]

### GDDR2

Paměti založené na této technologii velmi trpěly na masivní vyzařování tepla a velkou spotřebu a tak se příliš v grafických kartách neobjevovaly.




Obr. 1.4: Propustnost pamětí v různých typech pamětí.[4]

Tyto karty využívaly efektivní frekvenci od 200 až po 533 MHz. Tato řada má kořeny v řadě DDR1, i když tuto řádu co se týče efektivní frekvence předběhla. K funkci tato grafická karta potřebovala napětí 2,5V. Zástupcem této řady je Nvidia FX 5700 ultra (velikost sběrnice je 128 bitů). [4]

### GDDR3

V této řadě došlo k tomu, že původní DDR2 čipy byly optimalizovány a to vedlo ke snížení napěťových nároků a také ke snížení vyzařování tepla i při zvýšení taktů. Parametr CAS latency zůstal stejný jako u pamětí DDR2. V této řadě se efektivní frekvence pohybuje od 700 až do 2400 MHz a disponuje hodnotou propustnosti od 5,6 do 156,6 GB/s. Hlavní výhodou od předchozích řad je možnost hardwarového resetu pamětí, což znamená že paměť je schopna okamžitě uvolnit svou kapacitu. Zástupcem této řady je například Nvidia 6800 Ultra Extreme (Velikost sběrnice je 256 bitů). [4]

### GDDR4

Rozdílem této řady oproti předchozí je zvýšení efektivní frekvence, která se pohybuje od 2000 až do 3600 MHz a zvýšila se propustnost od 128 až do 200 GB/s. Tato změna sebou přináší i negativní odezvu a to zvýšení hodnoty CAS latency a také zvýšení tepelných výdajů. Z těchto důvodu se nakonec tato technologie příliš nepoužívala.

Zástupce můžeme najít pouze u firmy ATI, která s touto řadu lehce experimentovala. [4]

#### GDDR5

Hlavní změnou oproti předchůdcům je, že dokáže přenášet 4 bity za jeden takt. Efektivní frekvence se pohybuje od 900 až do 5600 MHz s propustností 130 až 230 GB/s. Došlo i ke snížení napětí na 1,35V. Další změnou je i sníženi CAS latency. Tato technologie byla zpočátku doménou společnosti ATI, avšak nyní je standardem u vyšších modelů Nvidie. Zástupcem může být například Nvidia GTX 460 (je taktovaná na 1800 MHz a sběrnicí 256bitů). [4]

## 1.3 Čip obsahující základní informace

Stejně jako na základní desce i grafická karta obsahuje „BIOS“. Bios je program, který má za úkol řídit nastavení a funkce grafické karty. Obsahuje informace jakou hodnotu má efektivní frekvence jádra, na jaké frekvenci pracují paměťové moduly nebo výpočetní jednotky (shadery) a podporuje snadnou instalaci softwaru od výrobce (tzv. ovladače). Různé grafické karty mohou být odlišné pouze v nastavení biosu, což umožňuje bios změnit (updatovat) a zvýšit tak výkon grafiky. Tato změna biosu, však není podporována výrobci a může při ní dojit k poškození grafické karty. [3]

## 1.4 Konvertor signálu

Konvertor digitálního signálu (RAMDAC, jinak známy i pod zkratkou DAC). Tento prvek má za úkol převádět digitální obraz vytvářený počítačem na analogový signál, který lze zobrazit klasickými CRT monitory a některými LCD monitory. Rychlost konvertoru se udává v MHz. U tohoto prvku záleží čím vyšší rychlost konvertoru, tím vyšší je obnovovací frekvence. V dnešní době je v trendu využívat přímo digitální výstup (DVI, HDMI). [16]

## 1.5 Výstupy grafické karty

- VGA - je realizován konektorem D-SUB 15, tento konektor má 3 řady po 5ti pinech. Signál na tomto konektoru probíhá spojitě (analogově), to znamená, že úroveň signálu je reprezentována hodnotou napětí. Tento výstup se objevil koncem 80tých let (Dříve se používal konektor D-SUB 9). Na monitoru se výsledné barvy skládají ze 3 barev a to červené, modré a zelené (režim RGB).

Dříve byly barvy vedeny digitálně, avšak z příchodem barevné palety o 16ti milionech barev počet pinu v konektoru nebylo dostatek pinu a tak se i barvy přesunuly na analogový signál (na tož stačily tři vodiče) a pro CRT obrazovky byl tento způsob nejsnažší na zobrazení, což však již neplatí pro LCD monitory, u kterých dochází opět k přepočtu na digitální signál a tím i ke ztrátám a především ke snížení kvality obrazu. [17] Velkou výhodou tedy je jeho jednoduchost. Nevýhodou je jednoznačná náchylnost na různé okolní vlivy, které ovlivňují průběh signálů a v obrazu se objevují defekty (duchy a neostrost).

- CVBS - neboli kompozitní signál - jde o klasicky TV výstup, který reprezentuje cinch konektor. Jde o jeden z nejstarších způsobu přenosu obrazu, kdy obraz je přenášen po jednom signálovém vodiči (dříve pouze jasová složka->černobílý obraz, později byly namodulovány i barvy). Tento výstup má pevně dané výstupní rozlišení a jakákoliv změna se tvoří až v zobrazovacím zařízením, kde muže docházet ke zmenšování nebo k roztahovaní, či oříznuti. K opravám při přeškálování obrazu se využívají různé filtry. Maximální rozlišení tv výstupu bylo 800x600 nebo 1024x768. Pro tento výstup existují dvě normy a to NTFS a PAL, kdy PAL se používal v Evropě. [17]
- S-Video - jedna se o velmi podobný konektor jako zastarale konektory PS/2 (pro klávesnici, myš). Tento konektor nahradil konektor CVBS a stejně jako on se hodí spíše pro připojení televize. Na rozdíl od předchozího konektoru jsou barvy vedeny zvlášť a výsledkem je jednoznačně vetší ostrost obrazu ale především v lepším podání barev. Základní verze konektoru je normována se 4 piny a je dodržována všemi výrobci. Avšak se na některých kartách objevuje ten samy konektor se sedmi piny ten již není normován ale 4 funkce původních 4 pinů zůstává stejná a pouze se mění funkce přidaných 3 pinů. Tyto tři piny mohou sloužit buď pro TV vstup, nebo muže byt na těchto pinech jiný signál např. CVBS, který pak lze získat redukcí dodávanou výrobcem, a odpadá tam nutnost další redukce, která by jen mohla obraz více degradovat. Využití tohoto konektoru spočívá v sledování videa na televizi. [18]
- Y-Pb-Pr - neboli komponentní výstup - tento výstup je datován s příchodem HD televizí. Je realizován pomocí tří cinch kabelu. Signál v těchto kabelech je přenášen rozdílově a barevné složky se z něj vypočítávají. To zaručuje vyšší kvalitu než například u VGA. Tento výstup se vetšinou u grafických karet integruje do konektoru S-Video a vyvádí se pomocí příslušné redukce. Maximální výstupní rozlišení je 720p nebo 1080i.[18]

- DVI - jedná se o jednoznačně nejpokročilejší výstup na grafické kartě. Digitální přenos je bez ztrát kvality a tím pádem je výsledný obraz čistý a ostrý i při velmi vysokých rozlišeních. Výhodou tohoto portu je, že umožňuje na výstupu kromě digitálního i analogový signál (DVI-D pouze digitální, DVI-I je analogový i digitální). U portu DVI existují ještě dvě varianty a to single-link a dual-link, jejich rozdíl spočívá ve schopnosti zobrazení maximálního rozlišení. Pro single-link je maximální rozlišení 1920x1200 a pro dual-link je to 2560x1600. Další takovou výhodou je i při absenci portu HDM lze jednoduše kompenzovat redukci DVI->HDMI. DVI také jako HDMI umí i implementovanou ochranu HDCP, která je nutná pro přehrávání HD filmu z nástupců klasických DVD.[18]
- HDMI - je trendem nyní pro připojení počítače k televizi. HDMI se přenáší nekomprimovaný obrazový a zvukový signál v digitálním formátu. Existují 4 typy konektorů, kdy nejpoužívanější je Typ A s 19 piny a je fyzicky kompatibilní s single link DVI-D. Typ B má 29 pinů a umožňuje rozlišení WQUXGA (3840x2400) a je kompatibilní s dual-link DVI-D, avšak tento konektor se zatím moc nepoužívá. Typ C-mini a Typ D jsou konektory určené pro přenosná zařízení a jsou menši než typ A. U technologie HDMI není definována maximální délka kabelu ale s rostoucí délkou vzniká na vodiči útlum a proto běžně muže kabel dosáhnout délky 12 až 15m. Lze mít ale HDMI na delší délku pomocí převodníku a nástavců se můžeme dostat až na 50m s nástavci na optické kabely dokonce i přes 100m. Nejčastěji se používá rozlišení 1080p ale poslední verze HDMI umožňují i rozlišení 4k. [18]
- DisplayPort - je digitální konektor využívaný k přenosu nekomprimovaného obrazového signálu s podporou až 8mi kanálového zvuku a ochranu DPCP (DisplayPort Content Protection), u kterého se využívá 128bitové šifrování založeno na AES. Maximální rozlišení je WQXGA (2560x1600) na 3m, pro delší vzdálenost do 15m zvládá rozlišení 1080p (1920x1080). DisplayPort dokáže emitovat HDMI nebo DVI signál ale nefunguje obraceně. Tato technologie spoléhá na paketový přenos dat, jeho kořeny jsou založeny na malých datových paketech známých jako mikro pakety, které mohou vložit hodinový signál do datového toku. [19]

# 2 VÝPOČTY NA GRAFICKÉM PROCESORU

Neboli General-Purpose computing on Graphics Processing Units (GPGPU) je metoda využití grafického procesoru, nejen ke zobrazovacím výpočtům ale i k obecným algoritmům. Prostředkem této možností jsou programovatelné shadery. Využíti výpočtů na grafické kartě je vhodné u algoritmu, které mezi sebou mají minimální nebo žádné vazby. [14]

## 2.1 Technologie CUDA

CUDA- vytváří platformu umožňující paralelní výpočty a upravuje API model, jenž byl navrhnut společností NVIDIA. Tato architektura byla představena v roce 2006 a poprvé vydána 23. června 2007 a umožňuje softwarovým programátorům využívat na grafických kartách, na kterých již CUDA je implementována, paralelizaci k výpočtů obecných algoritmů - známe pod zkratkou GPGPU. Tato platforma je softwarovou vrstvou, která poskytuje přímí přístup k prostředkům virtuální instrukční sadě GPU a paralelních výpočetních prvků.

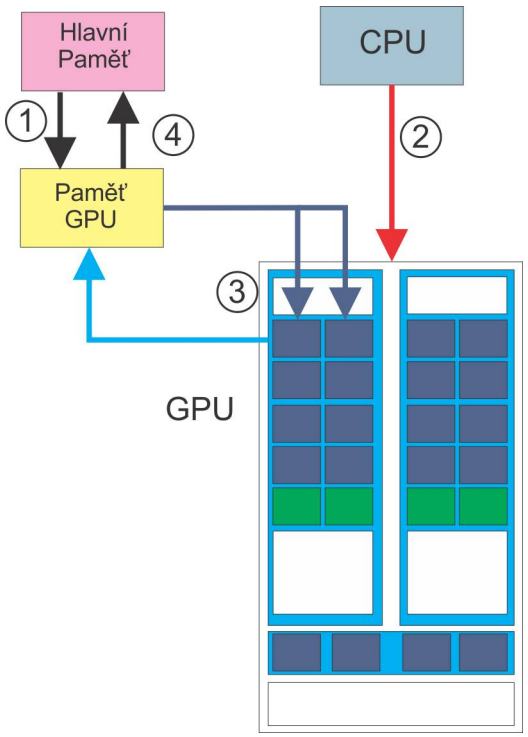
Platforma CUDA je navrhnuta na programování v programovacích jazycích C, C++ a FORTRAN. To umožňuje snadno programátoru využívat paralelní zpracování požadavků na zdrojích grafické karty. Rovněž CUDA podporuje knihovny jako OpenACC a OpenCL. Za pomocí třetích stran CUDA podporuje další programovací jazyky jako Pythlon, Perl, Fortran, Java, Ruby, Lua, Haskell, R, Matlab, IDL.

Této funkce grafické karty se dnes hojně využívat například v herním průmyslu, kdy GPU neslouží jen pro grafické renderování ale i pro fyzikální výpočty herních efektů ( fyzické efekty jako kouř, oheň) příklady najdeme v technologii PhysX a Bullet. Platforma CUDA může být využita i pro zrychlení negrafických aplikací například ve výpočetní biologii, kryptografii a v dalších oblastech.

Dále CUDA poskytuje jak nízkou úroveň tak i vysokou úroveň API. V počátcích bylo řešení CUDA SDK vytvořeno pro veřejnost na operační systémy Microsoft Windows a Linux. Operační systém MacOS byl přidán až ve verzi 2.0. (14. února 2008).

Platforma CUDA byla implementovaná do všech grafických procesorů (GPU) od verze grafické karty G8x, včetně verzi GeForce, Quadro a Tesla. Poslední verze CUDA je 7.5 vydaná 8. září 2015

Na obrázku je možné vidět příklad zpracování dat pomoci CUDA na grafické kartě GeForce 8800. V prvním kroku se zkopírují data z hlavní paměti do paměti grafické karty. Potom jakmile dojde k instrukci od procesoru CPU dojde ke zpracovaní dat, které vykoná GPU paralelně na každém jádru. Po dokončení zpracování jsou výsledky opět zkopírovány z paměti GPU zpět na hlavní paměť. [1] [7]




Obr. 2.1: Průběh zpracovaní pomocí CUDA.

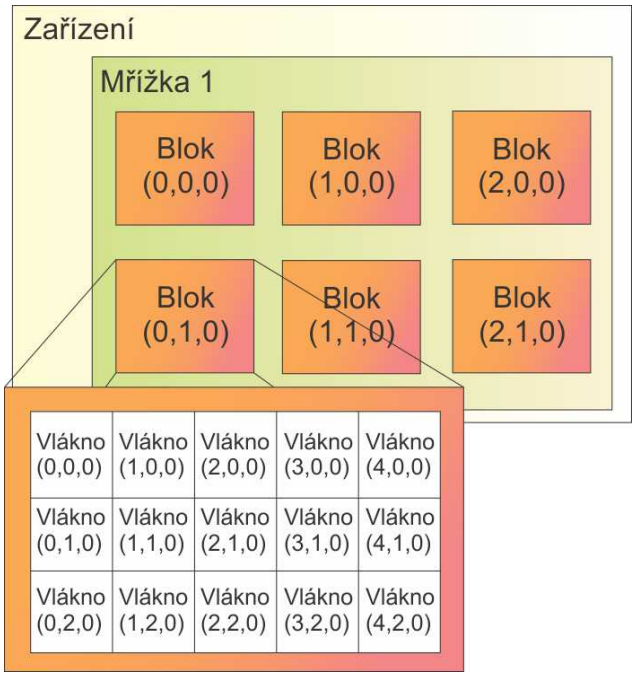
### 2.1.1 Architektura

Architektura CUDA je definována na modelu kernel-host. V Kernerové části se nacházejí uživatelsky definované funkce, které se na zařízení pomocí vláken paralelně zpracují. GPU na zpracovaní využívá několika „Streaming multiprocessors,“ které umožňují zpracovávat velké množství vláken dohromady. Řízení zpracování vláken obstarává tzv Hardware thread monitor. Další třídění vláken probíhá do bloků a to až do 3 dimenzí, ty ještě mohou být rozdělené do mřížek po 1 až 2 dimenzí. [7]

### 2.1.2 Model paměti

Hardwarové řešení grafických karet NVIDIA vyčleňuje 6 typů pamětí.

- Registry - slouží jen pro jedno vlákno, je umístěna na čipu a je velmi rychlá




Obr. 2.2: Architektura CUDA.[7]

- Globální - je přístupná pro všechny bloky i pro hostitele. Je umístěna mimo čip a nemá cache a je k ní pomalý přístup. Pro maximální využíti přenosové rychlosti je nutný sdružený přístup. Avšak pro implementaci není nutný žádný koherenční protokol.
- Sdílená - je přístupný pro všechny vlákna ale pouze uvnitř jednoho bloku. Nachází se na čipu, je velmi rychlá a neobsahuje cache. Je dále rozdělená na 16 bank a v rámci jedné banky může dojít k současnému přístupu k různým adresám, což vede k serializaci.
- Lokální - je přístupná pouze pro jedno vlákno. Nachází se mimo čip, je pomalá a neobsahuje cache. Jedná se o část globální paměti.
- Konstantní - je přístupná pro všechna vlákna. Nachází se mimo čip, je pomalá ale obsahuje cache. Jedná se o část globální paměti avšak určené pouze pro čtení.
- Texturová - je přístupná pro všechny vlákna. Nachází se mimo čip, ale obsahuje cache a funguje ve dvou režimech pomalý přístup (miss) nebo rychlý přístup (hit). Jde o část paměti určené pouze pro čteni (speciální 2D kešování).

K urychlení přístupu k pamětem je možné využít Broadcast, který slouží k urychlení přístupu na stejné adresy ve sdílené pamětí, je však pouze pro čtení a tak nedochází ke konfliktu bank. [7]

### 2.1.3 Výhody platformy CUDA

- Rozdělené čtení - kód lze číst z libovolných adres v paměti
- Sjednocená virtuální paměť (Od vezre 4.0)
- Sjednocená paměť (Od verze 4.0)
- Sdílená paměť - CUDA umožňuje rychlé sdílené oblasti paměti, která mohou být sdílená mezi vlákny. To muže být použito jako uživatelský řízená vyrovnávací paměť, jenž umožní větší šířku pásma, než je možné pomocí textury na vyhledávání.
- Velká podpora ze strany Nvidia, možno stáhnout spoustu knihoven a příkladů.

### 2.1.4 Nevýhody platformy CUDA

- Program v CUDA je spustitelný pouze na grafických kartách Nvidia s označením CUDA-enabled
- Omezená podpora v systému Linux a přenositelnost mezi verzemi.

## 2.2 Technologie AMD APP SDK

Dříve známé jako ATI FireStream ale je možné setkat se i s názvem AMD Stream. Jde o programovací prostředí pro grafickou kartu pro využití jejich výkonu k výpočtům na základě programovatelných shaderů. Karty AMD (tehdy známe jako ATI) byli prvními, které umožňovali i využit grafickou kartu pro jiné účely než grafické. Ze začátku vycházely na uzavřeném rozhraní „Close to Metal“ a byli nedostupné pro veřejnost. To se změnilo s příchodem CUDA od Nvidie a přechodem na otevřený programovací rozhraní OpenCL. Poslední verze AMD APP SDK je 3.0 vydaná 25.srpna 2015 a tato verze podporuje OpenCL 2.0. Jelikož se nyní jedna spíše o rozšíření jazyka OpenCL je popis architektury dále rozveden v další kapitole. [6]

### 2.2.1 **Výhody platformy** AMD APP

- Možné využívat s menšími úpravami i na kartách Nvidia (Jedná se o OpenCL)
- Optimalizace kódu OpenCL
- Pro vývojáře nabízí ke stažení příklady a knihovny.

### 2.2.2 **Nevýhody platformy** AMD APP

- Nic navíc oproti standardu OpenCL

## 2.3 Standard OpenCL

Jedná se stejně jako u CUDA o knihovnu umožňující zpracování operací (funkcí) na jádru grafické karty. Hlavní zbraní OpenCL, že je multiplatformní. To znamená, že lze jej programovat a spouštět na jakémkoliv operačním systému (Windows, Apple, Linux) a stejně tak na jakémkoliv hardwaru (GPU od Nvidie, AMD, Intel nebo IBM). Dokonce jej lze provozovat i na některých virtuálních strojích (WMware).

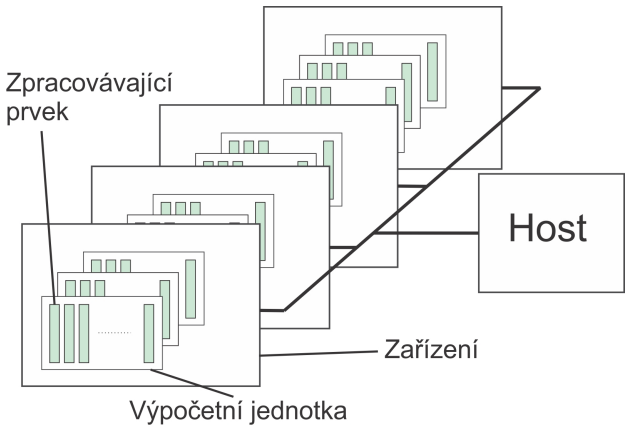
### 2.3.1 Historie standardu OpenCL

Prvotní návrh standardu OpenCL přišel ze společnosti APPLE. Ta však v polovině roku 2008 nechala vývoj tohoto projektu konsorciu Khronos_group. Zde se sešli zástupci všech významných společností vyrábějící grafické čipy. (AMD, Nvidia a další) a následně na to byl vydán po 5 měsících standard OpenCL ve verzi 1.0. k uveřejnění tohoto standardu došlo 8. prosince 2008.

### 2.3.2 Architektura standardu OpenCL

#### Model platformy

Tento model spočívá v tom, že host je připojen k jednomu nebo více zařízení OpenCL. Jakékoliv OpenCL zařízení je rozdělené na jedno nebo více výpočetních jednotek, která jsou dále dělena na jeden nebo více procesních elementů. Výpočty na zařízení nastávají uvnitř procesních jednotek. Aplikace OpenCL je definována jak na hostovy tak na zařízení . Kdy kód probíhá na hostovi dokud host neodešle kernel kód jako příkaz z hosta na zařízení OpenCL. Zařízení OpenCL pak zpracuje výpočet na svých precesních elementech. [20]

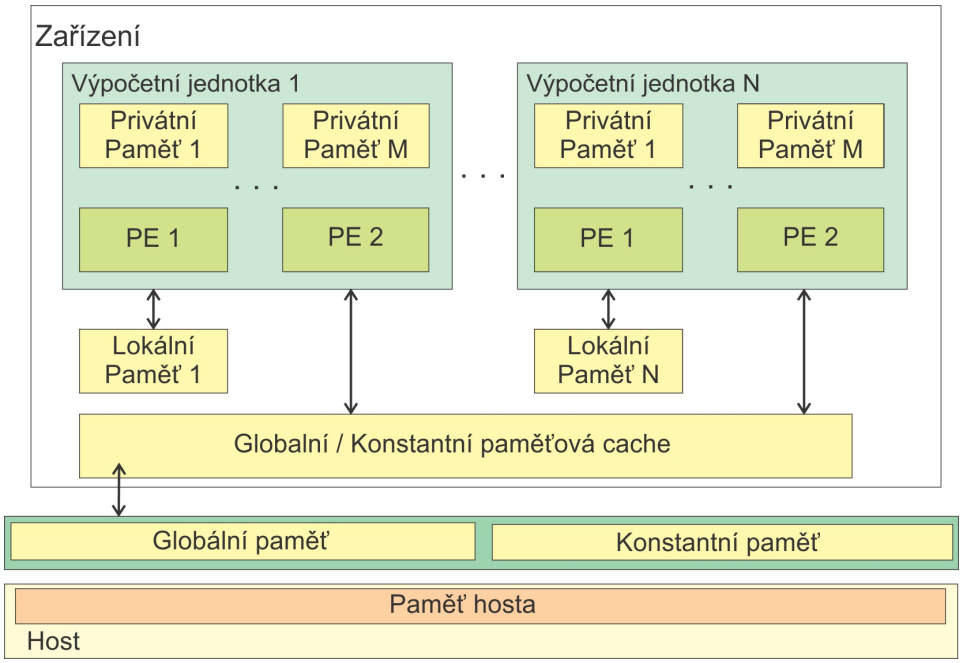



Obr. 2.3: Model platformy OpenCL. [20]

### Paměťový model

OpenCL kernel nám dává přístup ke 4 různým typům pamětí.

- Globální paměť - mohou jí číst všechny instance kernelu. Jedná se o fyzickou paměť zařízení
- Konstantní paměť - mohou jí číst všechny instance kernelu avšak oproti globální do ní nemůžou zapisovat. Jedná se také o fyzickou paměť zařízení, kterou však lze využít efektivněji než globální. Musí zde byt podpora ze strany zařízení. Tato část pamětí muže byt nastavena a data do ní zapsaná pouze hostem.
- Lokální paměť - mohou jí číst všechny instance kernelu v rámci skupiny. Fyzicky se jedná o sdílenou paměť pro každou výpočetní jednotku.
- Privátní paměť - paměť, kterou lze použít v rámci instance kernelu. Fyzicky se jedná o registry, které muže použít každý procesní element. [20]

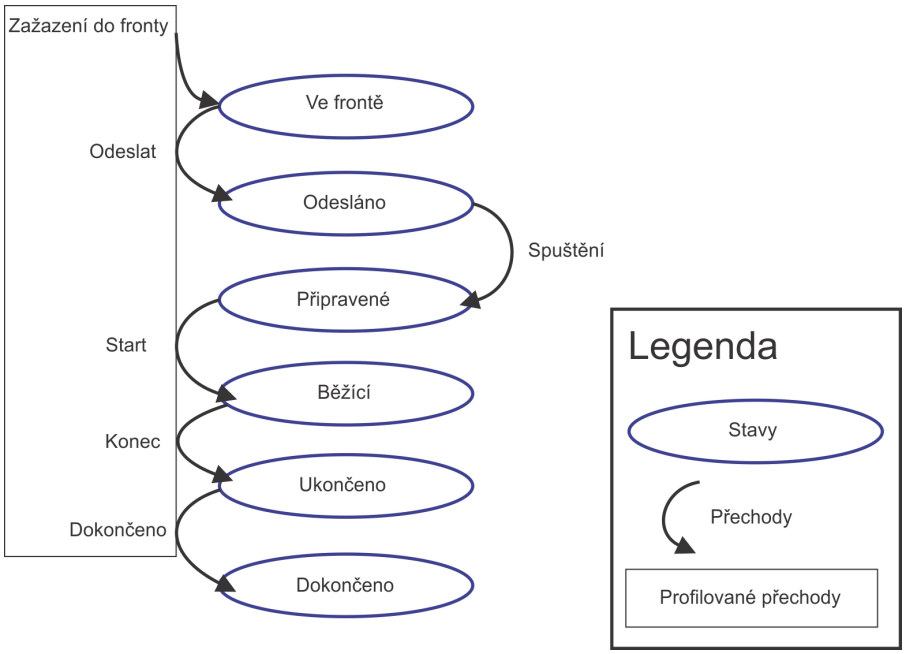



Obr. 2.4: Paměťový model OpenCL. [20]

### Exekuční model (provádějící)

Běh tohoto modelu probíhá na dvou odlišných úrovních heterogenního paralelního stroje. Základní program (Aplikace) je vykonávaná na hostovi. Tento program má za úkol komunikovat mezi hostem za zařízením, spouštět a kontrolovat výpočty na těchto zařízeních. Na samotném zařízení běží tzv. kernel (jádro). Který je napsány v jazyce OpenCL C. K výpočtu používá jedno či více vláken, které jsou zpracované pomocí procesních elementu daného zařízení.

Dále se na základě kernelu zvolí počet instanci kernelu a specifikaci indexového prostoru (jeden až tří rozměrný). Pomocí těchto hodnot se určí počet skupin a přiřadí se indexy (Globální, Lokální, Skupinový). [20]




Obr. 2.5: Stavy a přechody definované v exekučním modelu OpenCL. [20]

**Kontexty a příkazové fronty** Host program pomocí OpenCL API vytvoří a řídí kontext. V kontextu jsou uloženy informace o zařízení, kernelu, paměťových objektech využitých při výpočtech aj. Mezi další funkce, které OpenCL API umožňuje, hlavnímu programu patří komunikovat se zařízením pomocí příkazových front. Kdy ke každému zařízení patří jiná skupina příkazových front. Příkazové fronty je možno zpracovávat tak jak jdou za sebou „in order“ nebo nezávisle na pořadí „out of order“. Každý příkaz generuje událost, čímž lze hledat chyby v kódu. Typy příkazových front lze rozdělit do 3 skupin:

- Kernel-zařazovací příkazy - Zařadí kernel pro spuštění na zařízení
- Paměťové příkazy - Přesun dat mezi hostovou pamětí a pamětí na zařízení
- Synchronizační příkazy - Explicitní synchronizační body, které definují přísné vazby mezi příkazy [20]

### Programovací model

Standard OpenCL se obzvlášť soustřeďuje na datově paralelní programovací model. Tento model definuje výpočet jako souběh stejných instancí kernelu zpracovávající datové složky vstupní datové struktury. Kdy jejich jednoznačnou identifikaci zajišťuje exekuční model pomocí indexování a je umožněna úzká komunikace mezi instancemi. Druhým programovým modelem je Úlohově paralelní programovací model (multitasking). Ten umožňuje souběh několika různých instancí kernelu ne však stejných. [20]

### OpenCL framework

OpenCL framework umožňuje aplikacím využívat hosta a jeho nebo více OpenCL zařízení jako jeden heterogenní paralelní počítačový systém. Framework obsahuje následující komponenty [20]

- OpenCL Platform layer - platformní vrstva jenž umožňuje host programu najít OpenCL zařízení, jejich funkce a vytvářet kontexty.
- OpenCL Runtime - umožňuje host programu měnit kontexty jakmile byly vytvořeny.
- Synchronizační příkazy - vytváří spustitelný program obsahující OpenCL. Podporuje normu ISO C99 s rozšířením pro paralelní aplikace.

### Jazyk OpenCL C

Tento programovací jazyk vychází z normy ISO C99 a přináší určitá rozšíření i omezení jazyka „c“: [20]

Rozšíření jazyka:

- Vektorové datové typy
- Datové typy a funkce podporující práci s obrázky a jejich filtrováním
- Kvalifikátory adresního prostoru a přístupových práv.
- Kernelové funkce

Omezení jazyka:

- Ukazatele na funkce, pole proměnné délky a bitová pole jsou zakázaná.
- Velká většina hlavičkových souborů standardní knihovny jazyka C je nedostupná.
- Rekurzivní funkce nejsou povolené.
- Kernelové funkce nesměji deklarovat argumenty typu ukazatel a nic nevracet.

### 2.3.3 OpenCL profil embedded

Jde o odlehčenou verzi standardu OpenCL pro mobilní zařízení, či vestavěná zařízení nedisponující dostatečným výkonem pro běh celé verze. V tomto odlehčeném profilu jsou některé částí standardu buď nepovinné (podpora 3D obrazu) nebo zcela odstraněné (64bitové číselné typy). [20]

### 2.3.4 Standard OpenCL 2.0

Standard OpenCL 2.0 přinesl nové možnosti rozšíření. [20]

- přináší sdílení pamětí CPU a GPU (snižuje režii systému)
- přináší možnost větší autonomie GPU(možnost efektivního využití nových algoritmů).

### 2.3.5 Výhody OpenCL

- Možnost spustit na jakémkoliv Hardwaru.
- Možnost programovat na jakémkoliv Softwaru.
- Optimalizace
- Rozšíření pro paralelní aplikace

### 2.3.6 Nevýhody OpenCL

- Složitost jazyka OpenCL C.
- K přístupu většině knihoven jazyka „c“ nutné oddělit jádro od zbytku programu.
- Omezení některých funkcí jazyka „c“
- Složitější konfigurace programovacího prostředí (Nvidia).

## 2.4 Knihovna OpenCV

Jedná se o knihovnu obsahující programovací funkce hlavně zaměřený na projekt počítačového vidění v reálném čase. Tato knihovna obsahuje více než 500 optimalizovaných algoritmů. A je licencován pod BSD licencí, což umožňuje její využití zdarma i pro komerční sféru a lze ji provozovat multiplatformně (Windows, Linux, Mac). OpenCV je napsáno v jazyku „C++“ a je pro něj primárně určeno. Lze jej též provozovat na „C“ a existuje i přiřazení na jazyky jako Pythlon, Java, Matlab/Octave. Pomocí třetích stran je možné využít i na další programovací jazyky. Poslední vydaná verze je 3.1 ze dne 21. prosince 2015. [11]

### 2.4.1 Historie knihovny OpenCV

Vývoj tohoto projektu se datuje do roku 1999, ve vývojovém centru společnosti Intel v Ruském Nižním Novogorodu. V počátcích tohoto projektu byly stanoveny tyto cíle:

- Význam rozšířeného počítačového vidění a tím, že neposkytnou pouze otevřený kód ale také optimalizovaný kód i pro základní vizi struktury
- Šíření znalostí počítačového vidění tím, že poskytuje klasickou strukturu, na které mohou vývojáři stavět, čímž se kód stane snadněji čitelným a přenositelným.
- Rozšířené počítačové vidění založené na bázi komerčních aplikaci vytvořený jako přenosné výkonově-optimalizovaný kód dostupný zdarma - s licencí, která nemusí být otevřena nebo osvobozena

První alfa verze OpenCV se dostala k veřejnost na konferenci IEEE o počítačovém vidění a rozpoznávání vzorů v roce 2000. Avšak první verze vyšla až v roce 2006. [12]

# 3 IMPLEMENTOVANÉ ALGORITMY

V této kapitole rozvedeme tři vybrané algoritmy, které následně implementujeme na obě technologie i na centrální jednotku počítače. Jedná se o algoritmus Barevné konverze, který byl vybrán na základě jednoduché operace se vstupními daty. Dalším algoritmem je Filtrace, kde se již jedná o výpočetně náročnější operaci se vstupními daty. A třetím algoritmem je hledání minimální a maximální hodnoty ve vstupních datech, kdy tento algoritmus byl zvolen na základě složitější struktury, kterou není možné zcela paralelizovat, protože je nutno pracovat s mezivýpočty. Jejich výsledky jsou shrnuty v tabulce.

## 3.1 Barevná konverze

Tento algoritmus bude sloužit k převodu obrázku z jednoho barevného prostoru do druhého. Protože u tohoto řešení neexistuje úplně obecné řešení bude nutné nadefinovat jednotlivé barevné prostory. Výběr barevného prostoru, ze kterého se bude převádět a na jaký se bude převádět, zadá uživatel. Funkcí tohoto programu bude vzít již definované vzorce pro převod mezi barevnými prostory dle zadání uživatele a využít je k převodu daného obrázku. Pro vyšší výkon aplikace je možné uložit tří prostorné barevné prostory (HSV, XYZ aj.) do čtyř kanálového obrazu. Jako příklad uvádím převod z RGB na YCrCb [15]

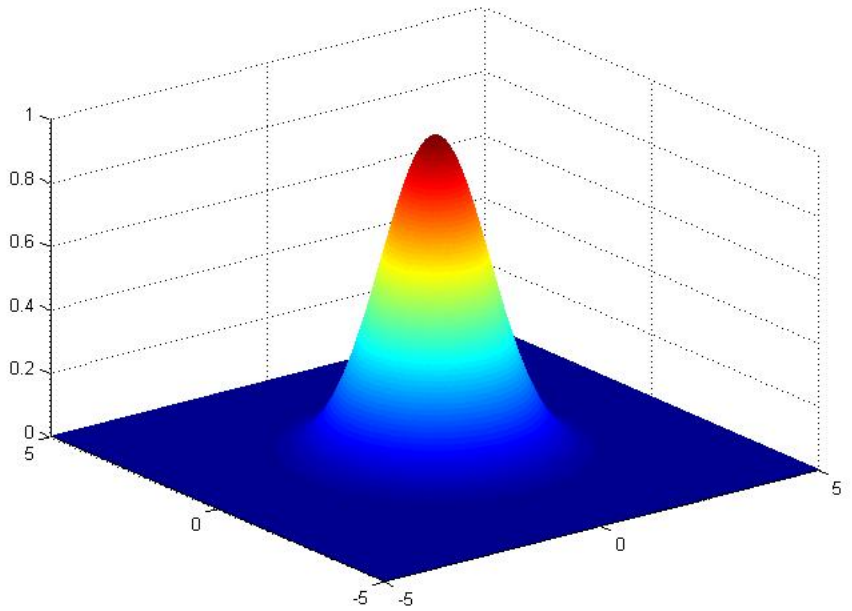
Obr. 3.1: Výstup algoritmu Barevná konverze.

```
Y = 0.299 * R + 0.587 * G + 0.114 * B
Cr = (R - Y) * 0.713 + delta
Cb = (B - Y) * 0.564 + delta
```

Kdy „Y“ je jasová složka, „R“ je červená složka, „G“ je zelená složka, „B“ je modrá složka, „Cb“ a „Cr“ jsou chrominační složky, delta je jednotka, která se mění podle počtu bitů obrázku pro 8-bitový = 128. K porovnání lze použít funkci „gpu::cvtColor.“

## 3.2 Filtrování obrazu

U těchto algoritmů máme na výběr mezi lineárním a nelineárním 2D flitrem. To znamená, že pro každý pixel (x,y) ve zdrojovém obrázku jsou sousední pixely použity k výpočtu. V případě lineárního filtru se jedná o vážený součet hodnot pixelů. V případě morfologických operací se musí jednat o minimální nebo maximální hodnoty. Vypočítaná hodnota je pak uložena na stejném místě (x, y), což zaručuje, že výstupní obraz bude mít stejnou velikost jako obraz vstupní. Za běžných okolností se funkce podporují přes multi-kanálové pole, v takových případech je každý kanál zpracován samostatně. To opět způsobí, že výstupní obraz bude mít, stejný počet kanálů jako u vstupního obrazu. [15]




Obr. 3.2: Aplikovaná Gaussova funkce.

V této práci použijeme Gaussovu 2D funkci (3.2) pro filtr. Kdy tento filtr aplikuje na černobílí obrázek tzn. pouze na jasovou složku. Když je vyhledávací okno mimo obraz, funkce použije nulové hodnoty. Ve funkci se využívá Gaussova 2D funkce, která nám vytvoří okno 5 na 5, což znamená 25 složek. Po aplikaci filtru bude obrázek lehce rozmazaný.

$$f(x,y) = A\exp(-(\frac{(x-x_0)^2}{2\sigma_x^2} + \frac{(y-y_0)^2}{2\sigma_y^2})) \quad (3.2)$$

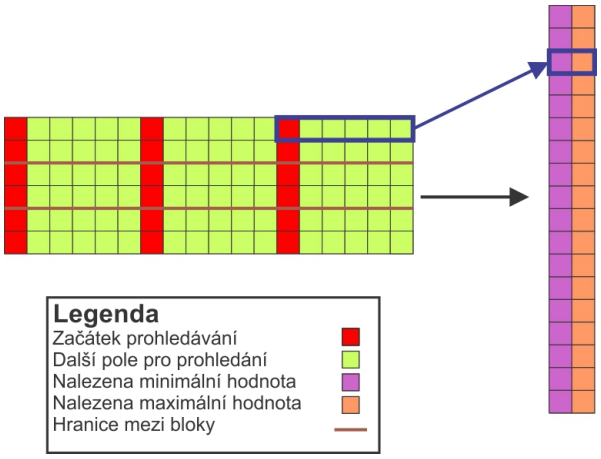
Obr. 3.3: Výstup algoritmu Filtrace.

## 3.3 Hledání min a max v matici

V tomto algoritmu prohledáme obrázkovou matici (Matrix) a jako výsledek nám vrátí nejvyšší a nejnižší číslo z celé matice. Tato funkce byla realizována rozdělení matice menších částí a každou část prohledalo jedno vlákno z grafické karty, kdy výsledkem je matice o dvou sloupcích. Počet řádků je závislý na velikosti vstupních dat. Dalším krokem je zpracovaní této dvouprvkové matice, kdy lze tuto matici vyřešit dvojím způsobem. V prvním případě můžeme zavolat ze zařízení další funkci, která nám tyto data vyhodnotí, ta je stále zpracovávaná na zařízení avšak již jen jedním vláknem. Jejichž nevýhodou je pomalejší jádro grafického procesoru ale za to nevzniká další režie. Druhou variantou je stvoření heterogenního systému, kdy konečnou práci provede již jednotka na hostovickém počítači, která disponuje vyšším výpočetním výkonem, za cenu vzniku režie při kopírování dat ze zařízení na hosta.

V rámci testování byl i proveden test, s jakou velikostí bloků, při stejném počtu vláken bude program nejrychlejší. Zde platí čím více bloku tím miň práce musí jednotlivá vlákna na zařízení vykonat ale za to také narůstá výstupní dvouprvková matice. Počet vláken byl zvolen podle procesoru grafické karty na 90 vláken.

K porovnání lze využít již definovanou funkci v knihovnách OpenCV „gpu::minMax“ nebo „gpu::minMaxLoc.“ [15]




Obr. 3.4: Ukázka vyhledávání Minimální a Maximální hodnoty (pro 3 bloky a pro 6 vláken v rámci bloku).

Tab. 3.1: Výstup algoritmu hledání min a max v matici aplikované na testované obrázky

| Malé rozlišení | | Střední rozlišení | | Velké rozlišení | |
|---|---|---|---|---|---|
| **Min:** | **Max:** | **Min:** | **Max:** | **Min:** | **Max:** |
| 0 | 250 | 4 | 195 | 0 | 227 |

# 4 INSTALACE SOFTWARU

Prvním krokem je potřeba si ověřit zdali daná grafická karta již podporuje architekturu CUDA či OpenCL. To lze ověřit na stránkách výrobce grafického čipu, nebo lze najít na internetu spoustu svobodných aplikací (např. „GPU Caps Viewer“). Po-té je vhodné zkontrolovat minimální požadavky a doporučení výrobce. Oba největší výrobci doporučují pro Operační systém Microsoft Windows Programovací prostředí Microsoft Visual studio, zde je nutno dát si pozor na správnou verzi.(Ke dní 28.11.2015 není stále podporována verze Visual studia 2015 !). Po instalaci programovacího prostředí je potřeba stáhnout rozhraní umožňující začít programovat. Zde záleží, na které architektuře budeme pracovat:

## 4.1 CUDA Toolkit

Tuto architekturu zvolíme máme-li grafickou kartu s grafickým čipem od Nvidie. Tento balík knihoven, kompilátoru, nástrojů pro ladění a dokumentace lze stáhnout přímo ze stránek Nvidie. Po kliknutí na stránku s odkazem ke stažení dostaneme na výběr, který operační systém používáme a následně musíme vybrat jeho správnou verzi. Nakonec se stránka zeptá, jestli chceme síťovou verzi (stáhne se jen instalátor a zbytek se začne stahovat při instalaci) nebo jestli chceme celý balík stáhnout hned (vhodné při instalaci na více počítačích). Po stažení si CUDA Toolkit ověří počítač na minimální požadavky, zkontroluje grafickou kartu a pohledá podporovanou verzi Visual Studia, pokud nenajde informuje o tom uživatele a zeptá se na pokračování v instalaci. Po instalaci je možné si otevřít již hotové příklady a ověřit funkčnost. Příklady lze nalézt v adresáři kam se CUDA Toolkit nainstalovala. Častý problémem se muže projevit poslední verze .NET Frameworku, která neobsahuje úplně všechno co CUDA potřebuje, a proto je nutno stáhnout a nainstalovat i starší verzi. [8]

## 4.2 AMD APP SDK

Tuto architekturu zvolíme pro případ, kdy máme grafický čip od AMD. Tento balík stejně jako CUDA Toolkit obsahuje knihovny, kompilátor, nástroje pro ladění a dokumentaci. První hlavní změnou jsou knihovny, které jsou pro OpenCL programování. Balík je možné stáhnout ze stránek AMD, kdy je v tabulce potřeba si vybrat který operační systém používáme. Zde již nemáme na výběr mezi celou a síťovou verzí. Stahuje se rovnou síťová verze, která se upravuje podle výběru při instalaci. Instalace nic neověřuje a příklady lze nalézt ve složce uživatele. Po instalaci byly příklady již plně schopné a nebylo tak nutno nic doinstalovávat. [6]

# 5 ČÁSTI ŘEŠENÍ

## 5.1 Načítání vstupních dat (obrázku)

V této části jde o načtení vstupního obrázku. Nejprve si však vytvoříme obrázkové kontejnery a to pomocí knihovny „OpenCV.“

```
cv::Mat frame, image;
```

Po-té se spustí smyčka, která očekává na zadání uživatele z klávesnice, kdy „i“ spustí zpracovávání programu a umožní načtení obrázku, a „q“, která smyčku ukončí a tím i celý program. Po spuštění zpracovaní je uživateli umožněn výběr obrázku, nejprve se však spustí filtr, který umožní využít výběr pouze obrázkových souborů, kdy lze tento filtr vypnout. K tomu slouží funkce „GetFilename()“, která vrací buď „true“ nebo „false“.

```
bool GetFilename(char* pcFileName,bool bImage=false)
...
ofn.lpstrFilter = TEXT("Image file (*.bmp)\0*.bmp;*.jpg;
*.ppm\0 All files (*.*)\0*.*\0");
...
```

V případě, že funkce vrátí „true“ je obrázek načten, pokud ne program ukončí zpracování a bude čekat na nové zpracování nebo na ukončení. Načtení obrázku je provedeno funkcí „imread()“ z knihovny „OpenCV“

```
frame=imread(cFileName,0);
```

Kdy lze druhým parametrem stanovit zda obrázek načíst se všemi barevnými složkami, či pouze jasovou složku (černobílí obrázek). Po této funkci je naposled ověřeno zdali se podařilo obrázek načíst a pokud ne program napíše chybu a opět čeká na zadání uživatele. Další funkcí, kterou využíváme z knihovny „OpenCV“, je funkce „imshow()“, která nám zobrazí obrázek uložený v obrázkovém kontejneru.

```
imshow("Image", frame);
```

Na dalším řádku je vždy spuštění již dané funkce. K měření doby provádění operace je využita funkce „getTickCount()“ a „getTickFrequency()“, které nám vrátí dobu zpracování v ms.

```
printf("Done in: %g ms\n", 1000 * ((double)getTickCount() -
        time) / getTickFrequency());
```

Posledním řádkem je funkce na uzavření všech oken, při ukončení programu.

```
destroyAllWindows();
```

## 5.2 Kód CUDA

Před samotným zpracováním výpočtu pomocí CUDA, musíme napřed alokovat proměnné a jejich velikost na grafické kartě.

```
cudaStatus = cudaMalloc((void**)&d_image, velkost *
                    sizeof(unsigned char));
```

Po-té si alokovanou paměť vyčistíme nastavením počátečních hodnot. V našem případě naplníme pole samými nulami.

```
cudaMemset(d_image, 0, cn*velikost*sizeof(unsigned char));
```

Na to si do těchto proměnných nakopírujeme data pro zpracování na gpu, a po dokončení je pomocí stejného příkazu opět zkopírujeme zpět.

```
cudaMemcpy(d_image, image, velikost*sizeof(unsigned char),
                                    cudaMemcpyHostToDevice);
cudaMemcpy(image, vysledek, velikost*sizeof(unsigned char),
                                    cudaMemcpyDeviceToHost);
```

Pak již spustíme zpracování na gpu vykonávání funkce a to pomocí volání kernelu

```
kernel <<<rows, cols >>>(vysledek, d_image, cols, cn, rows);
```

Vyčkání na dokončení všech vláken inicializujeme funkcí

```
cudaThreadSynchronize();
```

Po dokončení všech operaci na kernelu uvolníme paměť grafické karty díky příkazu

```
cudaFree(d_image);
```

Samotný kernel definujeme s klíčovým slovem „__global__“, jenž nám říká, že proces bude spuštěn na grafické kartě.

```
__global__ void kernel(unsigned char* out, unsigned char* in,
                    int cols, int cn, int rows)
```

K jednotlivým vláknům a blokům potom přistupujeme pomocí příkazů

```
int idy = threadIdx.x;
int idx = blockIdx.x
```

## 5.3 Kód OpenCL

Při zpracování na technologii OpenCL je nutné nejprve vytvořit všechny proměnné jednotky potřebné pro složení samotného jádra.

```
cl_device_id device_id = NULL;
cl_context context = NULL;
cl_command_queue command_queue = NULL;
cl_mem input;
cl_mem memobj = NULL;
cl_program program = NULL;
cl_kernel kernel = NULL;
cl_platform_id platform_id = NULL;
cl_uint ret_num_devices;
cl_uint ret_num_platforms;
cl_int ret;
```

V dalším kroku načteme cestu k souboru obsahující kernel.

```
char filename[] = "'./kernel.cl"';
```

Následně se pokusíme soubor otevřít a načíst zdrojový kód. Pokud se to nepodaří program napíše chybu a skončí.

```
fp = fopen(fileName, "r");
```

Po-té si alokujeme paměť pro kernel a načteme zdrojový kód. Po-té soubor opět zavřeme. Dále se pokusíme najít informace o zařízení a najdeme si jeho rozhraní.

```
ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT,
        1, &device_id, &ret_num_devices);
```

V dalším kroce si vytvoříme OpenCL kontext.

```
context = clCreateContext(NULL, 1, &device_id, NULL, NULL,
        &ret);
```

Dále si vytvoříme příkazovou frontu.

```
command_queue = clCreateCommandQueue(context, device_id, 0,
        &ret);
```

Pak si alokujeme paměť na zařízení. Kdy definujeme zda li je určena pouze pro čtení nebo čtení i zápis.

```
input = clCreateBuffer(context, CL_MEM_READ_ONLY, velikost *
        sizeof(unsigned char), NULL, &ret);
memobj = clCreateBuffer(context, CL_MEM_READ_WRITE, velikost
        * sizeof(unsigned char), NULL, &ret);
```

Dále vytvoříme a sestavíme spustitelný kernel, k tomu nám slouží tyto příkazy

```
program = clCreateProgramWithSource(context, 1, (const char
        **)& source_str, (const size_t *)&source_size, &ret);
ret = clBuildProgram(program, 1, &device_id, NULL, NULL,
        NULL);
```

Pak vytvoříme OpenCL kernel. To uděláme pomocí následujícího příkazu

```
kernel = clCreateKernel(program, "colorConversion", &ret);
```

Nyní si zapišeme data do proměnných alokovaných na grafické kartě

```
ret = clEnqueueWriteBuffer(command_queue, input, CL_TRUE, 0,
  sizeof(unsigned char)*velikost, image, 0, NULL, NULL);
```

Nastavíme OpenCL kernel parametry pomoci funkce „clSetKernelArg“, kdy každému musíme jeho druhý parametr zvýšit o jedničku.

```
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), &memobj);
ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), &input);
ret = clSetKernelArg(kernel, 2, sizeof(int), &rows);
ret = clSetKernelArg(kernel, 3, sizeof(int), &cols);
ret = clSetKernelArg(kernel, 4, sizeof(int), &cn);
```

Nyní jsme konečně připraveni spustit výpočet na zařízení. Nejdříve si však nastavíme si počet použitých vláken a pak již zavoláme kernel.

```
global[0] = rows;
global[1] = cols;
ret = clEnqueueNDRangeKernel(command_queue, kernel, 2, NULL,
  global, NULL, 0, NULL, NULL);
```

Příkaz „clFinish()“ nám provede synchronizaci vláken.

```
clFinish(command_queue);
```

Po dokončení výpočtu zkopírujeme data zpět na hosta

```
ret = clEnqueueReadBuffer(command_queue, memobj, CL_TRUE, 0,
    velikost * sizeof(unsigned char), image, 0, NULL, NULL);
```

A uvolníme alokovanou paměť na zařízení

```
ret = clFlush(command_queue);
ret = clFinish(command_queue);
ret = clReleaseKernel(kernel);
ret = clReleaseProgram(program);
ret = clReleaseMemObject(memobj);
ret = clReleaseCommandQueue(command_queue);
ret = clReleaseContext(context);
free(source_str);
```

Samotný kernel je definován pomoci slova „__kernel," a slova __global slouží k označení paměti.

```
__kernel void colorConversion(__global unsigned char* out,
    __global unsigned char* in, int rows, int cols, int cn)
```

A k jednotlivým vláknům lze přistupovat pomocí příkazu „get_global_id()"

```
int idx = get_global_id(0);
int idy = get_global_id(1);
```

## 5.4 Barevná konverze

Samotné zpracování pomocí algoritmu se na jednotce CPU provádí pomocí dvou cyklů, kdy pro každý pixel je tak spočítaná nová hodnota. Na zařízení je každému vláknu přiřazen jeden pixel a ten vlákno zpracuje a vrátí spočítanou novou hodnotu. Využívá se převodních vzorců pro zobrazeni z RGB na YCbCr. Aplikace algoritmu je totožná pro CUDA a OpenCL, u CPU je o něco jednoduší avšak stále na stejném principu. V první části získáme jednotlivé barvy a ty si uložíme do proměnných blue,green a red.

```
double blue = in[clr_addres]; // ziskame modrou barvu
double green = in[clr_addres + 1]; // ziskame zelenou barvu
double red = in[clr_addres + 2]; // ziskame cervenou barvu

out[clr_addres] = (unsigned char)(0.257 * red + 0.504 *
  green + 0.098 * blue + 16); //vypocet pro jasovou slozky
out[clr_addres + 1] = (unsigned char)(0.439 * red - 0.368 *
 green - 0.071 * blue + 128); //vypocet pro Cr
out[clr_addres + 2] = (unsigned char)(-0.148 * red - 0.291 *
 green + 0.439 * blue + 128);    //vypocet pro Cb
```

Pak aplikujeme vzorec, kdy spočítanou hodnotu následně převedeme na „unsigned char“ tzn. hodnota (0-255), a uložíme na pozici výstupního obrazu. První složka je výpočet jasové složky, po ní se spočítá chrominační složka Cr a pak chrominační složka Cb.

## 5.5 Filtrace

Při použití tohoto algoritmu nejdříve spočítáme složky pro Gaussovu 2D funkci, kterou pak použijeme jako filtr. Toho docílíme využitím funkce „createGaussFilter,“ kdy hodnoty filtru předáváme odkazem a nepotřebujeme tak návratovou hodnotu filtru (tudíž funkce vlastně nic nevrací (void)). Při zavolání této funkce předáme naši referenci na pole, které jsme si předtím vytvořili a do kterého budou uloženy hodnoty filtru. Dále si funkce nastaví na začátku hodnoty, pro výpočet filtru a pro jeho normalizaci. Po-té se pomocí dvou cyklů vytvoří a nakonec ještě normalizuje. Tím získáme hodnoty Gaussovy 2D funkce, které pak aplikujeme na vstupní data čili obrázek.

```
void createGaussFilter(double filter[][5])
{
        // nastavim standardni odchylku na 15.0
        double sigma = 15.0;
        double r, s = 2.0 * sigma * sigma;
        // promenna sum je pro normalizaci
        double sum = 0.0;
        // vytvorim gaussuv filtr pro 5x5
        for (int x = -2; x <= 2; x++)
        {
                for (int y = -2; y <= 2; y++)
                {
                        r = sqrt(x*x + y*y);
                        filter[x + 2][y + 2] = (exp(-(r*r) /
                                s)) / (PI * s);
                        sum += filter[x + 2][y + 2];
                }
        }

        // normalizuji gausuv filtr
        for (int i = 0; i < 5; ++i)
        for (int j = 0; j < 5; ++j)
                filter[i][j] /= sum;
}
```

Předtím než aplikujeme filtr, tak si zvětšíme vstupní obraz aby nám při prohledávání nejbližších sousedů nesahal někam mimo paměť. Víme, že budeme potřebovat dva pixely na každou stranu, takže vytvoříme nový obrázkový kontejner ale o 4 řádky a 4 sloupce větší a naplníme jej samými nulami, čímž kompenzujeme vyhledávací okno, takže když půjde mimo obrázek načte si nulovou hodnotu. Samotné nakopírování vstupního obrázku do nového kontejneru provedeme pomocí funkce „copyTo," z knihovny OpenCV. Kdy pomocí parametrů můžeme ovlivnit kam se vstupní obrázek nakopíruje, v našem případě jej posuneme o dva pixely doleva a o dva pixely níž.

```
Mat workPicture(frame.rows + 4, frame.cols + 4, CV_8UC1,
        Scalar(0));
frame.copyTo(workPicture.rowRange(2, (frame.rows + 2)).
        colRange(2,(frame.cols + 2)));
```

Aplikace filtru probíhá opět jak předešlém algoritmu, kdy pro počítaný pixel je však nutné použít dva cykly ještě pro procházení sousedních pixelů, ze kterých se také počítá výsledný pixel. Samotný výpočet probíhá tak, že se vezme pixel a vynásobí se filtrem a přičte se k už předtím spočítaným sousedům. Po dokončení sečtení všech sousedů je hodnota nakopírovaná na pozici pro výstupní obraz.

```
for (int x = 0; x < 5; x++){
        for (int y = 0; y < 5; y++){
                suma = suma + (double)in[x + (idx*cols)
                        + y + (idy)] *filter[x*5+y];
                }
        }
        out[addres] = (unsigned char)(suma);
}
```

## 5.6 Hledání minimální a maximální hodnoty

Složení tohoto programu je pro jednotku CPU velmi podobné jako v předchozích případech, opět máme dva cykly, které procházejí vstupní data a pomocí dvou jednoduchých podmínek si ukládají nejmenší a největší hodnotu nalezenou v datech. Pro GPU jsou programy odlišné od předchozích neboť nyní musíme zpracovávat více vstupních dat na jednom vláknu, abychom je mohli mezi sebou porovnat. Navíc nám pro obě technologie pro paralelní výpočty vznikly dvě cesty k dokončení vyhledávání. Avšak v první fázi převedeme obrázek na jednorozměrné pole a každému vláknu dáme zpracovávat určitý počet pixelů. Zde bohužel není jak zjistit vhodný počet pixelů pro vlákno jinak než testováním. Proto si jádro upravíme tak aby bylo možné měnit hodnoty bez složitějších zásahů do kódu. Po výběru této hodnoty si při vypracovávání uložíme první hodnoty té části, kterou vlákno zpracovává pak už procházíme zbývající pixely a porovnáváme je s nejnižší a nejvyšší naleznou hodnotou. Tyto hodnoty průběžně zapisujeme do pole „hodnoty“, kdy pro každé vlákno jsou právě dvě volná místa pro uložení vyhledávaných hodnot.

```
int idx = threadIdx.x;  //get coordinates x
int idy = blockIdx.x;   //get coordinates y

hodnoty[idx * 2 + (threads * 2 * idy)] = (int)in[idx * cols
        + (threads * cols * idy)];
hodnoty[idx * 2 + 1 + (threads * 2 * idy)] = (int)in[idx
        * cols + (threads * cols * idy)];
for (int y = 0; y < cols; y++){
        if (hodnoty[idx * 2 + (threads * 2 * idy)] > (int)in
                [(idx*cols + y) + (threads * cols * idy)]){
                hodnoty[idx * 2 + (threads * 2 * idy)] =
                 (int)in[(idx*cols + y) + (threads * cols
                        * idy)];
        }
        if (hodnoty[idx * 2 + 1 + (threads * 2 * idy)] <
                (int)in[(idx*cols + y) + (threads * cols
                        * idy)]){
                hodnoty[idx * 2 + 1 + (threads * 2 * idy)] =
                        (int)in[(idx*cols + y) + (threads *
                                cols * idy)];
        }
}
```

Po dokončení prohledávání máme jedno veliké jednorozměrné pole, kdy na lichých hodnotách máme uložené nejmenší nalezené hodnoty a na sudých nevetší nalezené hodnoty. Nyní můžeme nechat data zpracovat přes zařízení nebo data zkopírujeme a necháme zpracovat přes hosta. V obou variantách je výpočet stejný, kdy si opět uložíme první hodnoty do výsledné proměnné a ty necháme už jen jedním cyklem prohledat a nakonec vypsat.

```
vysledek[0] = hodnota[0];
vysledek[1] = hodnota[1];
for (int i = 0; i < threads * numBlocks; i++)
        {
                if (vysledek[0] > hodnota[i * 2]){
                        vysledek[0] = hodnota[i * 2];

                }
                if (vysledek[1]  < hodnota[i * 2 + 1])
                {
                        vysledek[1] = hodnota[i * 2 + 1];
                }
        }
cout << "Minimalni hodnota je: " << vysledek[0] << "\n";
cout << "Maximalni hodnota je: " << vysledek[1] << "\n";
```

# 6 TESTOVÁNÍ

Samotné testování aplikací proběhlo na počítači značky Dell typ OptiPlex 755, obsahující následující hodnoty hardwaru. Procesor Intel Core 2 Duo E6550 o výkonu 2,33 $Ghz$, použitá operační paměť měla kapacitu 6 GB(6144 MB) a jednalo se o paměti typu DDR2. Grafická karta použitá pro paralelní výpočty byla od společnosti Nvidia a jednalo se o typ GeForce GT 440, kdy kapacita paměti byla 1 GB typu DDR3 a počet jader na grafickém čipu je 96 jader o výkonu 810 $MHz$. Grafická karta má funkci „CUDA-ENABLED.“ [21]

Pro každý program a pro každý obrázek různých rozlišení byl test spuštěn několikrát a z výsledků byly odstraněny hodnoty jenž značně vybočovaly z řady. Z výsledků bylo odebráno 10 hodnot, které byly ještě zprůměrovány a pak zakresleny do přehledného grafu.

## 6.1 Barevná konverze

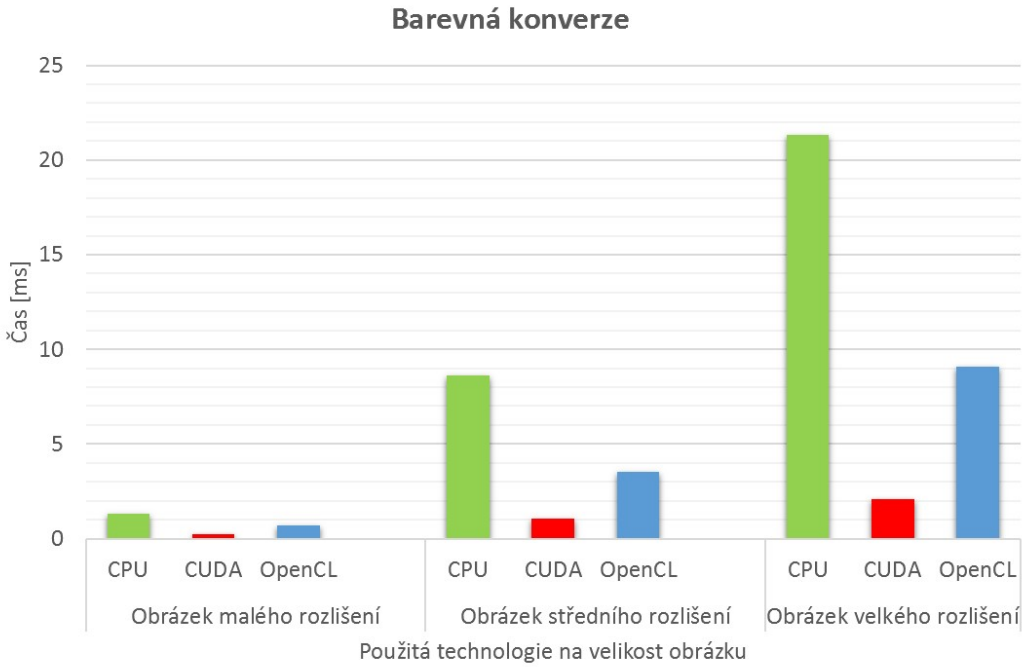
Tab. 6.1: Barevná konverze pro malé a střední rozlišení obrázku

| Obrázek malého rozlišení | | | | Obrázek středního rozlišení | | |
|---|---|---|---|---|---|---|
| CPU | CUDA | OpenCL | | CPU | CUDA | OpenCL |
| 1.3212 | 0.225259 | 0.700855 | | 8.35395 | 1.03522 | 3.62306 |
| 1.34012 | 0.241978 | 0.695136 | | 8.26859 | 1.04446 | 3.6257 |
| 1.36211 | 0.232739 | 0.705695 | | 8.63068 | 1.04622 | 3.63362 |
| 1.32692 | 0.249017 | 0.705695 | | 8.72571 | 1.05326 | 3.4858 |
| 1.32164 | 0.2169 | 0.704815 | | 8.67556 | 1.03654 | 3.4396 |
| 1.32868 | 0.223059 | 0.700855 | | 9.01476 | 1.05502 | 3.44356 |
| 1.30404 | 0.235818 | 0.694696 | | 8.65488 | 1.05414 | 3.42332 |
| 1.341 | 0.21822 | 0.705255 | | 8.54797 | 1.04757 | 3.4858 |
| 1.308 | 0.22042 | 0.703495 | | 9.08692 | 1.05414 | 3.5368 |
| 1.32472 | 0.228779 | 0.703495 | | 8.32403 | 1.0493 | 3.41716 |
| Zprůměrovaná hodnota: | | | | | | |
| 1.327843 | 0.2292189 | 0.7019992 | | 8.628305 | 1.047587 | 3.511442 |

Tab. 6.2: Barevná konverze pro velké rozlišení obrázku

| Obrázek velkého rozlišení | | |
|---|---|---|
| CPU | CUDA | OpenCL |
| 21.2949 | 2.06165 | 9.00465 |
| 21.243 | 2.04537 | 9.10936 |
| 21.488 | 2.08893 | 9.04776 |
| 21.3173 | 2.08893 | 9.06844 |
| 21.2144 | 2.09069 | 9.27698 |
| 21.0683 | 2.08981 | 9.0416 |
| 21.3002 | 2.07177 | 9.0372 |
| 21.3424 | 2.05989 | 9.09704 |
| 21.7582 | 2.09377 | 9.09968 |
| 21.3213 | 2.06913 | 9.0504 |
| Zprůměrovaná hodnota: | | |
| 21.3348 | 2.075994 | 9.083311 |

Jak je z výsledků patrné technologie pro paralelní využití grafického procesoru pro obecné výpočty nám může ušetřit na algoritmu pro barevnou konverzi spoustu času, kdy na základě velikosti obrázku vidíme trend větší efektivnosti kódu pro větší obrázek. Velmi nečekaným a zajímavým faktem je rozdíl technologií CUDA a OpenCL. Kdy technologie CUDA je více efektivní oproti technologii OpenCL. Za tímto rozdílem shledávám lepší optimalizaci technologie CUDA na grafické kartě s grafickým procesorem Nvidia i samotné zefektivnění mého jednoduchého kódu překladačem „nvcc.“ Jinak myšleno technologie OpenCL zde platí za svou univerzálnost. Což nás dostává k myšlence využití těchto technologii. Sice je technologie CUDA rychlejší ale kvůli jejímu omezení je vhodná spíše pro operace specifikované a tvořené pro určitý typ zařízení. Zato technologii OpenCL díky univerzálnosti lze využívat na různých strojích a tak se vyplatí pro širší spektrum uživatelů a zařízení. Využití centrálního procesoru není pro tento algoritmus tak zajímavé a hodí se pouze pro stroje, které nejsou schopny zvládnout ani jednu z testovaných paralelních technologií. Výjimku může tvořit opravdu malý zdroj dat, kdy vzniklá režie použití paralelních technologií s výpočtem na gpu překoná dobu trvání zpracování výpočtu na centrálním procesoru.




Obr. 6.1: Graf výsledného zpracování algoritmu Barevná konverze.

## 6.2 Filtrace

Tab. 6.3: Filtrace pro malé a střední rozlišení obrázku

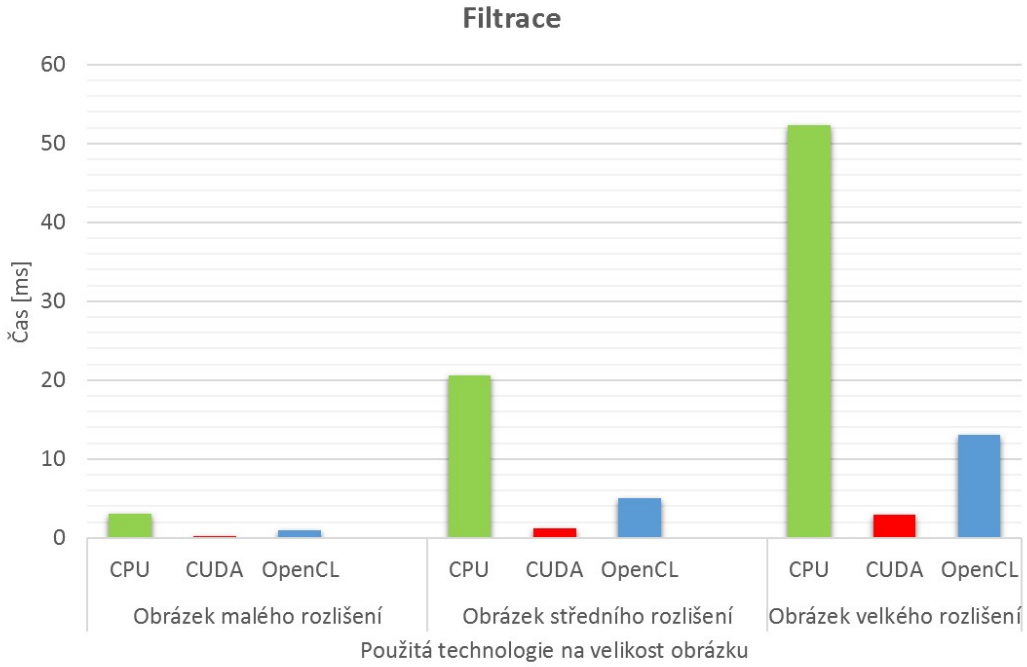
| Obrázek malého rozlišení | | | | Obrázek středního rozlišení | | |
|---|---|---|---|---|---|---|
| **CPU** | **CUDA** | **OpenCL** | | **CPU** | **CUDA** | **OpenCL** |
| 3.16155 | 0.223939 | 0.914235 | | 20.5663 | 1.19273 | 5.17788 |
| 3.1853 | 0.247697 | 0.928754 | | 20.7986 | 1.22265 | 5.11981 |
| 3.0696 | 0.249897 | 0.919075 | | 20.5272 | 1.21341 | 5.08945 |
| 3.16947 | 0.233618 | 0.925674 | | 20.6878 | 1.21561 | 4.9711 |
| 3.0588 | 0.222619 | 0.915995 | | 20.7005 | 1.19229 | 5.11409 |
| 3.07443 | 0.2446917 | 0.927434 | | 20.4436 | 1.19933 | 5.13608 |
| 3.07268 | 0.21558 | 0.926994 | | 20.437 | 1.17909 | 5.12552 |
| 3.05112 | 0.246377 | 0.927434 | | 20.3208 | 1.18437 | 5.12728 |
| 3.06388 | 0.244177 | 0.923474 | | 20.7203 | 1.20813 | 5.11409 |
| 3.09335 | 0.245937 | 0.902356 | | 20.9984 | 1.19097 | 4.91434 |
| Zprůměrovaná hodnota: | | | | | | |
| 3.100018 | 0.23745327 | 0.9211425 | | 20.62005 | 1.199858 | 5.088964 |

Tab. 6.4: Filtrace pro velké rozlišení obrázku

| Obrázek velkého rozlišení | | |
|---|---|---|
| **CPU** | **CUDA** | **OpenCL** |
| 52.2135 | 2.92968 | 13.1433 |
| 52.3842 | 2.92749 | 13.0853 |
| 52.8523 | 2.93937 | 13.036 |
| 52.1704 | 2.95829 | 13.1561 |
| 51.9513 | 2.96489 | 13.1715 |
| 52.7617 | 2.91825 | 13.1266 |
| 52.4115 | 2.91649 | 13.0919 |
| 51.9147 | 2.94157 | 13.1328 |
| 52.1096 | 2.96577 | 13.1169 |
| 52.2174 | 2.92969 | 13.1631 |
| Zprůměrovaná hodnota | | |
| 52.29866 | 2.939149 | 13.12235 |

U algoritmu filtrace, vidíme velmi obdobný průběh jako na algoritmu barevná konverze. Opět nám paralelní technologie mohou ušetřit velmi mnoho času. Oproti algoritmu Barevná konverze vidíme však několikanásobně vyšší efektivitu zpracování technologiemi pro paralelismus oproti centrální jednotce CPU. Obdobně jako na algoritmu Barevná konverze dále vidíme na výsledcích rozdíl mezi technologiemi CUDA a OpenCL, kdy CUDA je až čtyřnásobně rychlejší než OpenCL. Za tímto rozdílem opět shledáváme optimalizaci technologie CUDA na grafické kartě s grafickým procesorem Nvidia a optimalizaci mého kódu překladačem „nvcc.“ Co se algoritmu aplikace filtru týče je technologie CUDA jednoznačně nejvhodnějším řešením. Ale i technologie OpenCL může být výhodná pro používání v domácích či kancelářských prostředí bez přístupu ke grafické kartě s grafickým čipem Nvidia.

Řešení přes jednotku centrálního procesoru si bere velmi hodně strojového času a pro algoritmus aplikace filtrace bych jej nedoporučoval.




Obr. 6.2: Graf výsledného zpracování algoritmu filtrace.

## 6.3 Hledání minimální a maximální hodnoty

Tab. 6.5: Hledání minimální a maximální hodnoty pro obrázek malého rozlišení

| Obrázek malého rozlišení | | | | |
|---|---|---|---|---|
| **CPU** | **CUDA** | | **OpenCL** | |
| | **Homogenní** | **Heterogenní** | **Homogenní** | **Heterogenní** |
| 0.065939 | 0.42896 | 0.139027 | 1.58385 | 0.483515 |
| 0.0655539 | 0.413122 | 0.139027 | 1.61421 | 0.495394 |
| 0.0655539 | 0.395523 | 0.124508 | 1.57065 | 0.597025 |
| 0.0659939 | 0.405202 | 0.135507 | 1.5623 | 0.485715 |
| 0.0655539 | 0.392884 | 0.138147 | 1.5579 | 0.586466 |
| 0.0655539 | 0.4272 | 0.142107 | 1.54646 | 0.595705 |
| 0.0659939 | 0.391124 | 0.140347 | 1.58957 | 0.580746 |
| 0.0655539 | 0.406082 | 0.139907 | 1.57197 | 0.596145 |
| 0.0659939 | 0.415321 | 0.146066 | 1.59485 | 0.521792 |
| 0.0659939 | 0.405202 | 0.132868 | 1.54998 | 0.485275 |
| Zprůměrovaná hodnota: | | | | |
| 0.06576841 | 0.408062 | 0.1377511 | 1.574174 | 0.5427778 |

Tab. 6.6: Hledání minimální a maximální hodnoty pro obrázek středního rozlišení

| Obrázek středního rozlišení | | | | |
|---|---|---|---|---|
| **CPU** | **CUDA** | | **OpenCL** | |
| | **Homogenní** | **Heterogenní** | **Homogenní** | **Heterogenní** |
| 0.416201 | 1.43339 | 0.460637 | 2.67715 | 1.22441 |
| 0.415321 | 1.4743 | 0.477796 | 2.81354 | 1.59001 |
| 0.412682 | 1.43911 | 0.470316 | 2.62128 | 1.54822 |
| 0.411802 | 1.47914 | 0.466357 | 2.65735 | 1.42195 |
| 0.416641 | 1.44439 | 0.476476 | 2.75986 | 1.45626 |
| 0.411362 | 1.47034 | 0.476036 | 2.78318 | 1.43823 |
| 0.411802 | 1.4765 | 0.465037 | 2.80606 | 1.54382 |
| 0.411802 | 1.48354 | 0.487475 | 2.78274 | 1.4831 |
| 0.414442 | 1.48838 | 0.498914 | 2.72863 | 1.21693 |
| 0.412242 | 1.44439 | 0.478236 | 2.61556 | 1.41271 |
| Zprůměrovaná hodnota: | | | | |
| 0.4134297 | 1.463348 | 0.475728 | 2.724535 | 1.433564 |

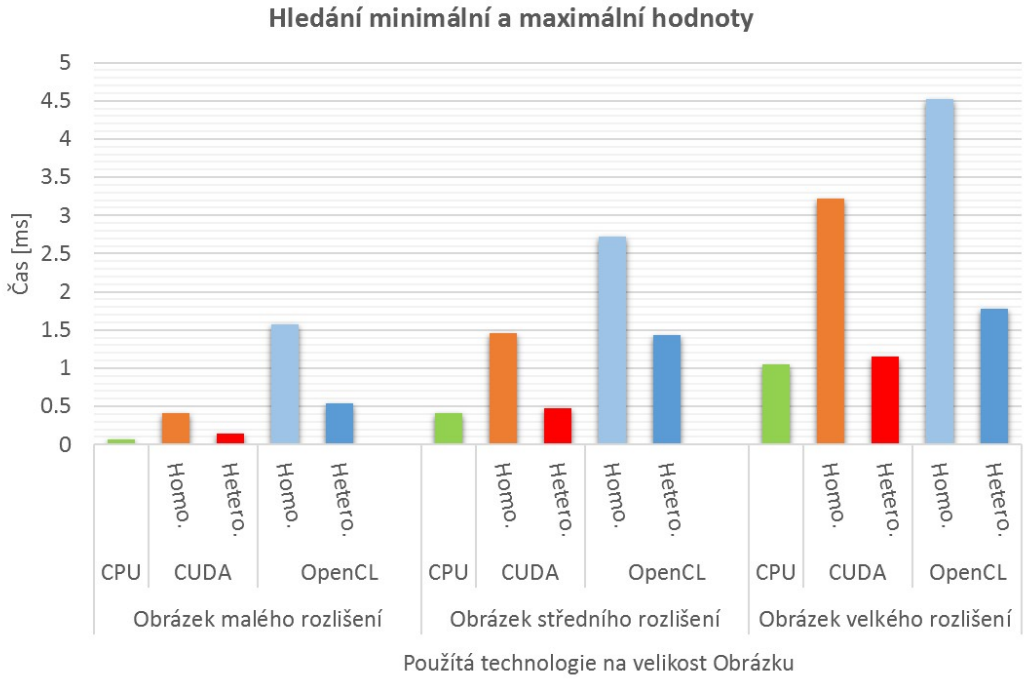
Tab. 6.7: Hledání minimální a maximální hodnoty pro obrázek velkého rozlišení

| Obrázek velkého rozlišení | | | | |
|---|---|---|---|---|
| **CPU** | **CUDA** | | **OpenCL** | |
| | **Homogenní** | **Heterogenní** | **Homogenní** | **Heterogenní** |
| 1.05106 | 3.20862 | 1.15357 | 4.49506 | 1.78579 |
| 1.05018 | 3.19322 | 1.20065 | 4.4955 | 1.79195 |
| 1.04446 | 3.26318 | 1.18129 | 4.52058 | 1.85883 |
| 1.04754 | 3.1897 | 1.12806 | 4.61165 | 1.88874 |
| 1.04622 | 3.22578 | 1.15885 | 4.56018 | 1.8993 |
| 1.04666 | 3.24734 | 1.14961 | 4.4867 | 1.66261 |
| 1.05106 | 3.20774 | 1.15665 | 4.52234 | 1.79855 |
| 1.0471 | 3.24162 | 1.11178 | 4.55006 | 1.67844 |
| 1.04886 | 3.22358 | 1.18305 | 4.50122 | 1.7484 |
| 1.04402 | 3.18706 | 1.14081 | 4.4867 | 1.63577 |
| Zprůměrovaná hodnota: | | | | |
| 1.047716 | 3.218784 | 1.156432 | 4.522999 | 1.774838 |

U tohoto algoritmu nám vyšly velmi neočekávané výsledky. Obzvlášť při zpracování na centrálním procesoru, kdy nastal děj, že režie zpracování na grafickém procesoru je vyšší než samotné zpracování tohoto algoritmu. Ve snaze zvrátit tento děj jsme vytvořili dvě řešení pro paralelní aplikace. Homogenní algoritmus, kdy celý program provádí grafický procesor, a heterogenní funkci, kdy závěrečné vyhodnocení vykonal opět centrální procesor. I přes tuto snahu a s možností měnit některé parametry a najít tak nejoptimalizovanější hodnoty, se nám podařilo pouze přiblížit se k době zpracování algoritmu jednotkou centrálního procesoru.

Na základě těchto výsledků konstatujeme, že jednoduché prohledávání, ať už většího pole dat, je výhodnější na jednotce centrálního procesoru oproti použití technologií pro paralelní zpracování.

Dále si můžeme povšimnout rozdílu mezi paralelními technologiemi CUDA a OpenCL, zde opět platí to co v předchozích algoritmech, kdy CUDA má výhodu v překladači i na běžícím hardwaru.




Obr. 6.3: Graf výsledného zpracování algoritmu hledání minimální a maximální hodnoty.

# 7 ZÁVĚR

Seznámili jsme se s problematikou grafických karet a s technologiemi pro paralelní pracování algoritmů.

Po této teoretické části jsme si vybrali možnosti jak otestovat a určit, že grafická karta může být výhodnější pro implementaci oproti klasickému centrálnímu procesoru (CPU). Takže jsme si vybrali tři rozdílné a různě náročné algoritmy, u kterých uvádíme co má být jejich výstupem a co jsme k dosažení tohoto výsledku využili. Prvním algoritmem byla barevná konverze, u které jsme pro každý pixel vypočítali novou hodnotu, podle již známého vzorce pro převod mezi barevnými prostory. Konkretně v této práci byl využit vzorec pro převod RGB na YCbCr. Druhým algoritmem byla filtrace, tento algoritmus již byl výpočetně náročnější ale stejně jako u barevné konverze jsme počítali novou hodnotu pro každý pixel. Tento výpočet spočíval ve vytvoření si převodní tabulky pomocí Gaussovy funkce a aplikací této tabulky na počítaný pixel a jeho sousedy. Třetím algoritmus slouží pro vyhledání minimální a maximální hodnoty, kdy se jedná o relativně jednoduchý algoritmus, který nelze zcela paralelizovat, kvůli nutnosti pracovat s mezivýpočty.

V navazující kapitole uvádíme z praxe samotnou instalaci programovacího prostředí a obou platforem. Ty mají své podkapitoly a to nejen proto, že jejich instalace ač obdobná není úplně stejná ale hlavně proto, že po instalaci se objevili různé problémy.

V další kapitole jsme se věnovali samotnému kódu, kdy je daná část kódu vysvětlena o to, co je jejím úkolem v programu. Tato kapitolka je rozdělena a věnuje se zvlášť oběma paralelním technologiím a zvlášť každému vybranému a implementovanému algoritmu.

Do poslední kapitoly jsme vložili samotné testování algoritmů aplikovaných, jak pro jednotku centrálního procesoru, tak pro obě paralelní technologie zpracované na grafickém procesoru.

Výsledky nám vycházely velmi dobře pro paralelní technologie. I při zvyšující se náročnosti algoritmu bylo na výsledcích pozorovatelné efektivnější zpracování kódu a zisk v úspoře času oproti vykonávání na centrálním procesoru a zdálo se, že využití paralelních technologií je jasná cesta do budoucnosti. To se však změnilo při otestování posledního algoritmu, kdy centrální procesor dokázal překonat režii nutnou k zavedení výpočtu paralelních technologii na grafický procesor. Ve všech algoritmech bylo pozorovatelné, že OpenCL je o něco pomalejší než CUDA. Tento rozdíl je s největší pravděpodobností zapříčiněn lepší optimalizací CUDA na grafické kartě s grafickým procesorem Nvidia a i samotnou optimalizaci kódu překladačem „nvcc.“

Na základě výsledků je tedy zřejmé, že využití paralelních technologií je nejvýhodnější při větším počtu stejných operací nad vstupními daty, které nejsou přímo závislé mezi sebou a kdy výpočet nám vrátí jednu hodnotu z celku, kdy celek nám vznikne po dokončení všech vláken. Sériový, nebo také sekvenční způsob zpracování je výhodný, když máme málo vstupních dat, nebo jsou vstupní data na sebe závislá a není tak možné, aby byl celý algoritmus zcela paralelizován kdy očekávaným výsledkem je jedna nebo více hodnot z celku.

# LITERATURA


[1] Zaorálek, L. *Úvod do technologie CUDA* [online]. 2009, poslední aktualizace 20. 07. 2009 [cit. 18. 11. 2015]. Dostupné z URL: <http://www.root.cz/clanky/uvod-do-technologie-cuda/>.

[2] Tišnovský, P. *Seriál Grafické karty a Grafické akcelerátory* [online]. 2005, poslední aktualizace 19. 10. 2005 [cit. 18. 11. 2015]. Dostupné z URL: <http://www.root.cz/serialy/graficke-karty-a-graficke-akceleratory/>.

[3] Hort, T. *Jak udělat z levné grafické karty dražší model? Nový BIOS!* [online]. 2008, poslední aktualizace 26. 08. 2008 [cit. 18. 11. 2015]. Dostupné z URL: <http://www.ddworld.cz/pc-a-komponenty/graficke-karty/jak-udelat-z-levne-graficke-karty-drazsi-model-novy-bios--2-2.html>.

[4] Míček, V. *Paměti v grafikách: jejich dělení, mýty a fakta* [online]. 2011, poslední aktualizace 18. 06. 2011 [cit. 18. 11. 2015]. Dostupné z URL: <http://www.cnews.cz/pameti-v-grafikach-jejich-deleni-myty-fakta>.

[5] Jun. *What is difference between GPU and CPU?* [online]. 2013, poslední aktualizace 06. 05. 2013 [cit. 23. 11. 2015]. Dostupné z URL: <http://allegroviva.com/gpu-computing/difference-between-gpu-and-cpu/#prettyPhoto>.

[6] *Developer AMD APP SDK* [online]. 2014, poslední aktualizace 2014 [cit. 23. 11. 2015]. Dostupné z URL: <http://developer.amd.com/>.

[7] *CUDA C Programming Guide* [online]. 2015, poslední aktualizace 2015 [cit. 23. 11. 2015]. Dostupné z URL: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.

[8] *CUDA Zone* [online]. 2015, poslední aktualizace 2015 [cit. 23. 11. 2015]. Dostupné z URL: <https://developer.nvidia.com/cuda-zone>.

[9] Mlčoch, T. *Výpočty pomocí grafických procesorů GPU - GPGPU* [online]. 2009, poslední aktualizace 15. 11. 2009 [cit. 23. 11. 2015]. Dostupné z URL: <http://tojaj.com/vypocty-pomoci-gpu/>.

[10] *GPU* [online]. 2015, poslední aktualizace 25. 09. 2015 [cit. 23. 11. 2015]. Dostupné z URL: <https://cs.wikipedia.org/wiki/GPU>.

[11] *OpenCV* [online]. 2016, poslední aktualizace 2016 [cit. 15. 04. 2016]. Dostupné z URL: <http://opencv.org/l>.

[12] Felsberg, M. *Robot Vision Systems* [online]. 2015, p. 12 poslední aktualizace 2015 [cit. 15. 04. 2016]. Dostupné z URL: <https://www.cvl.isy.liu.se/education/graduate/opencv/Lecture1_History.pdf>.

[13] Šulc, T. *Technologie: Unifikované shadery - co to vlastně je* [online]. 2008, poslední aktualizace 4. 4. 2008 [cit. 11. 2. 2016]. Dostupné z URL: <http://pcworld.cz/hardware/technologie-unifikovane-shadery-co-to-vlastne-je-3865>.

[14] Vykouřil, D. *Distribuované výpočty na moderních grafických kartách* [online]. 2010, poslední aktualizace 06. 10. 2010 [cit. 23. 11. 2015]. Dostupné z URL: <http://pctuning.tyden.cz/hardware/graficke-karty/18884-distribuovane-vypocty-na-modernich-grafickych-kartach>.

[15] *OpenCV doumentation* [online]. 2014, poslední aktualizace 13. 12. 2015 [cit. 13. 12. 2015]. Dostupné z URL: <http://docs.opencv.org/2.4/modules/core/doc/operations_on_arrays.html>.

[16] Dyo V,Ramirez J. R., Stevens J. *RAMDAC (random access memory digital-to-analog converter)* [online]. 2005, poslední aktualizace 01. 09. 2005 [cit. 23. 11. 2015]. Dostupné z URL: <http://whatis.techtarget.com/definition/RAMDAC-random-access-memory-digital-to-analog-converter>.

[17] Hulán R. *Výstupy grafické karty – 1.díl: VGA, CVBS* [online]. 2007, poslední aktualizace 29. 10. 2007 [cit. 23. 11. 2015]. Dostupné z URL: <http://myego.cz/item/vystupy-graficke-karty-1-dil-vga-cvbsl>.

[18] Hulán R. *Výstupy grafické karty – 2.díl: S-VHS, Ypbpr, DVI, HDMI* [online]. 2007, poslední aktualizace 04. 11. 2007 [cit. 23. 11. 2015]. Dostupné z URL: <http://myego.cz/item/vystupy-graficke-karty-2-dil-s-vhs-ypbpr-dvi-hdmi>.

[19] Skýpal J. *Neštěstí zvané DisplayPort a trochu také HDMI* [online]. 2012, poslední aktualizace 13. 03. 2012 [cit. 23. 11. 2015]. Dostupné z URL: <http://honza.info/pocitace/nestesti-zvane-displayport-a-take-hdmi/>.

[20] Lee Howes. *The OpenCL Specification* [online]. 2015, poslední aktualizace 11. 11. 2015 [cit. 13. 04. 2016]. Dostupné z URL: <https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf>.

[21] *GEFORCE® N440GT-MD1GD3/LP* [online]. 2016, poslední aktualizace 2016 [cit. 27. 04. 2016]. Dostupné z URL: <https://cz.msi.com/Graphics-card/N440GTMD1GD3LP.html#hero-overview>.

# SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

| | |
|---|---|
| CUDA | Compute Unified Device Architecture |
| GPU | Graphics processing unit |
| API | Application programming interface |
| GPGPU | General-purpose computing on graphics processing units |
| CPU | Central processing unit |
| AMD APP | AMD Accelerated Parallel Processing |
| OpenCL | Open Computing Language |
| OpenCV | Open Source Computer Vision |
| BIOS | Basic Input Output System |
| RAMDAC | Random Access Memory Digital-to-Analog Converter |
| TMU | Texture mapping unit |
| ROP | Render Output unit |

# SEZNAM PŘÍLOH

# A ZDROJOVÁ DATA (OBRÁZKY)

Pro účely měření jsme si vytvořili 3 různě veliké obrázky s různými motivy. A na ty následně aplikovali vybrané algoritmy.


Obr. A.1: Obrázek malého a středního rozlišení.

Obr. A.2: Obrázek velkého rozlišení.

# B OBSAH PŘILOŽENÉHO CD

Na přiložené medium je uloženo devět projektů z programovacího studia Visual Studio 2013 hlavní soubor je v programovacím jazyku C++ a u paralelních algoritmu jsou navíc soubory pro CUDA a OpenCL. Ve složce obrázky jsou uloženy ty obrázky, jenž sloužili jako vstupní data. Dále je zde uložen i text diplomové práce.