VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

INTERPOLACE OBRAZOVÝCH BODŮ

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. TOMÁŠ MINTĚL
AUTHOR

BRNO 2009

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

INTERPOLACE OBRAZOVÝCH BODŮ

PIXEL INTERPOLATION METHODS

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE Bc. TOMÁŠ MINTĚL
AUTHOR

VEDOUCÍ PRÁCE Ing. MICHAL ŠPANĚL
SUPERVISOR

BRNO 2009

Abstrakt

Tato diplomová práce se zabývá akcelerací interpolačních metod s využitím GPU a architektury NVIDIA® CUDA™. Grafický výstup je reprezentován demonstrační aplikací pro transformaci obrazu nebo videa s použitím vybrané interpolace. Časově kritické části kódu jsou přesunuty na GPU a vykonány paralelně. Pro práci s obrazem a videem jsou použity vysoce optimalizované algoritmy z knihovny OpenCV, od firmy Intel.

Abstract

This master's thesis deals with acceleration of pixel interpolation methods using the GPU and NVIDIA® CUDA™ architecture. Graphic output is represented by a demonstrational application for geometrical image transforms using chosen interpolation method. Time critical parts of the code are moved on the GPU and executed in parallel. There are used highly optimized routines from the OpenCV library, made by the Intel company for an image and video processing.

Klíčová slova

Interpolace obrazových bodů, transformace obrazu, grafická karta, GPU, NVIDIA® CUDA™

Keywords

Pixel interpolation methods, image transformation, graphics card, GPU, NVIDIA® CUDA™

Citace

Mintěl Tomáš: Interpolace obrazových bodů. Brno, 2009, diplomová práce, FIT VUT v Brně.

Interpolace obrazových bodů

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Michala Španěla. Další informace mi poskytli Ing. Radovan Josth a Bc. Petr Němeček.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Tomáš Mintěl
24. 5. 2009

Poděkování

Rád bych poděkoval vedoucímu této diplomové práce, Michalu Španělovi za ochotu a rady, které mi podával během vypracovávání této práce.

© Tomáš Mintěl, 2009
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah 1
1 Úvod 2
2 Využití GPU pro akceleraci výpočtů 3
2.1 Grafická karta 3
2.2 GPU 3
2.3 Porovnání současných CPU a GPU 5
2.4 Co je CUDA? 6
2.5 Jádra 6
2.6 Hierarchie vláken 7
2.7 Hierarchie pamětí 9
2.8 Rozdíl mezi programem na CPU a GPU 12
3 Geometrické transformace obrazu 15
4 Interpolace 18
4.1 Interpolace obrazových bodů 18
4.2 Nejbližší soused 18
4.3 Bilineární interpolace 19
4.4 Bikubická interpolace 21
4.5 Sinc interpolace 24
5 Návrh řešení 26
5.1 Přehrávání videa 30
6 Implementace 31
6.1 OpenCV 31
6.2 Problém optimalizace 31
6.3 Paměťová náročnost 33
6.4 Přístup mimo rozsah textury 34
6.5 Mapování pamětí 35
7 Testování a porovnání interpolací 36
7.1 Obrazové výsledky interpolací 37
7.2 Časové výsledky interpolací 40
7.3 Výsledky testování videa 42
8 Závěr 43
Seznam příloh 45
Příloha 1 46

1 Úvod

Diplomová práce se zabývá geometrickými transformacemi obrazu se zaměřením na interpolaci obrazových bodů. Jejím cílem je nastudování interpolačních metod a jejich implementace pro architekturu NVIDIA® CUDA™, což je paralelní výpočetní architektura grafických karet. Myšlenkou je přenesení výpočetních operací z CPU do GPU pomocí právě CUDA, tak aby bylo možné výpočet co možná nejvíce paralelizovat a tím jej urychlit. Grafický výstup bude reprezentován aplikací pro transformaci obrazu a videa.

Tato diplomová práce vzniká součastně s diplomovou prací kolegy Bc. Petra Němečka, jehož práce se zabývá geometrickými transformacemi obrazu.

První kapitola se jmenuje úvod a slouží k rychlému přiblížení tématu a k popisu jednotlivých kapitol.

V druhé kapitole si popíšeme Hardware, který je pro tuto práci podstatný. Jedná se o grafickou kartu a grafickou procesorovou jednotku. Je zde uvedeno i srovnání výpočetních výkonů nejnovějších grafických procesorových jednotek s výpočetním výkonem procesoru od firmy Intel. V další časti je podrobněji přiblížen paralelní systém CUDA a jsou zde vysvětleny základní programovací principy tohoto systému.

Ve třetí kapitole si pouze přiblížíme geometrické transformace obrazu. Vysvětlíme si základní pojmy a ukážeme si princip základních transformací.

Čtvrtá kapitola pojednává o samotných interpolacích. Ukážeme si jednotlivé interpolační metody, jejich klady i zápory a na transformaci si ukážeme, jaké jsou vizuální výsledky jednotlivých metod.

V páté kapitole je návrh řešení, který detailně popisuje vytvořenou aplikaci. Postupně jsou zde popsány všechny bloky demonstrační aplikace.

Šestá kapitola popisuje Implementaci. Je zde popsána knihovna OpenCV, paměťová náročnost programu a blíže je popsán problém optimalizace. Je zde také uveden způsob mapování pamětí a problém indexace mimo texturu.

V sedmé kapitole je souhrn testování. Nejprve jsou interpolace představeny pomocí vizuálních výsledků. Následně jsou prezentovány časové výsledky společně s porovnáním výkonu jednotlivých grafických karet. V této kapitole se také setkáme s testováním transformace videa.

Poslední kapitolou je závěr, kde je stručně shrnuta tato diplomová práce.

2 Využití GPU pro akceleraci výpočtů

Tato kapitola pojednává o nejdůležitějším hardwarovém vybavení počítače z pohledu této diplomové práce. Jedná se o grafickou kartu, přesněji pak grafický čip neboli GPU. Je zde také uvedeno porovnání výpočetních výkonů moderních procesorů a grafických karet. Ve druhé části této kapitoly je přiblížen systém NVIDIA® CUDA™ a jsou zde přiblíženy základní prvky programování v tomto systému.

Při psaní této kapitoly jsem čerpal z [8], [9], [10], [11], [13], [14] a [15].

2.1 Grafická karta

V současné době se grafické karty skládají většinou z pěti komponent:

- Systémové rozhraní
- Grafická paměť
- Grafický procesor – GPU
- Frame buffer
- RAMDAC

Systémové rozhraní je komponenta sloužící ke komunikaci s procesorem. Dříve se jednalo většinou o AGP, dnes je to ve většině případů sběrnice PCI Express. Paměť se pohybuje mezi 128MB až 2 048 MB. Slouží k ukládání objektů, textur a dat přicházejících z počítače. V paměti uložená data následně putují do grafického procesoru, který vypočítá všechny pozice, pohyby a rozhraní objektů 3D scény a vytvoří z nich obraz. Protože je grafický čip pro nás nejpodstatnější části, tak si jej přiblížíme v další podkapitole.

Obraz vzniklý v grafickém čipu je dále předán do frame bufferu, odkud putuje do RAMDAC (Random Access Memory Digital/Analog Converter). Ten zajišťuje výstup na monitor. A to jak pro VGA, tak i pro DVI, HDMI či DisplayPort.

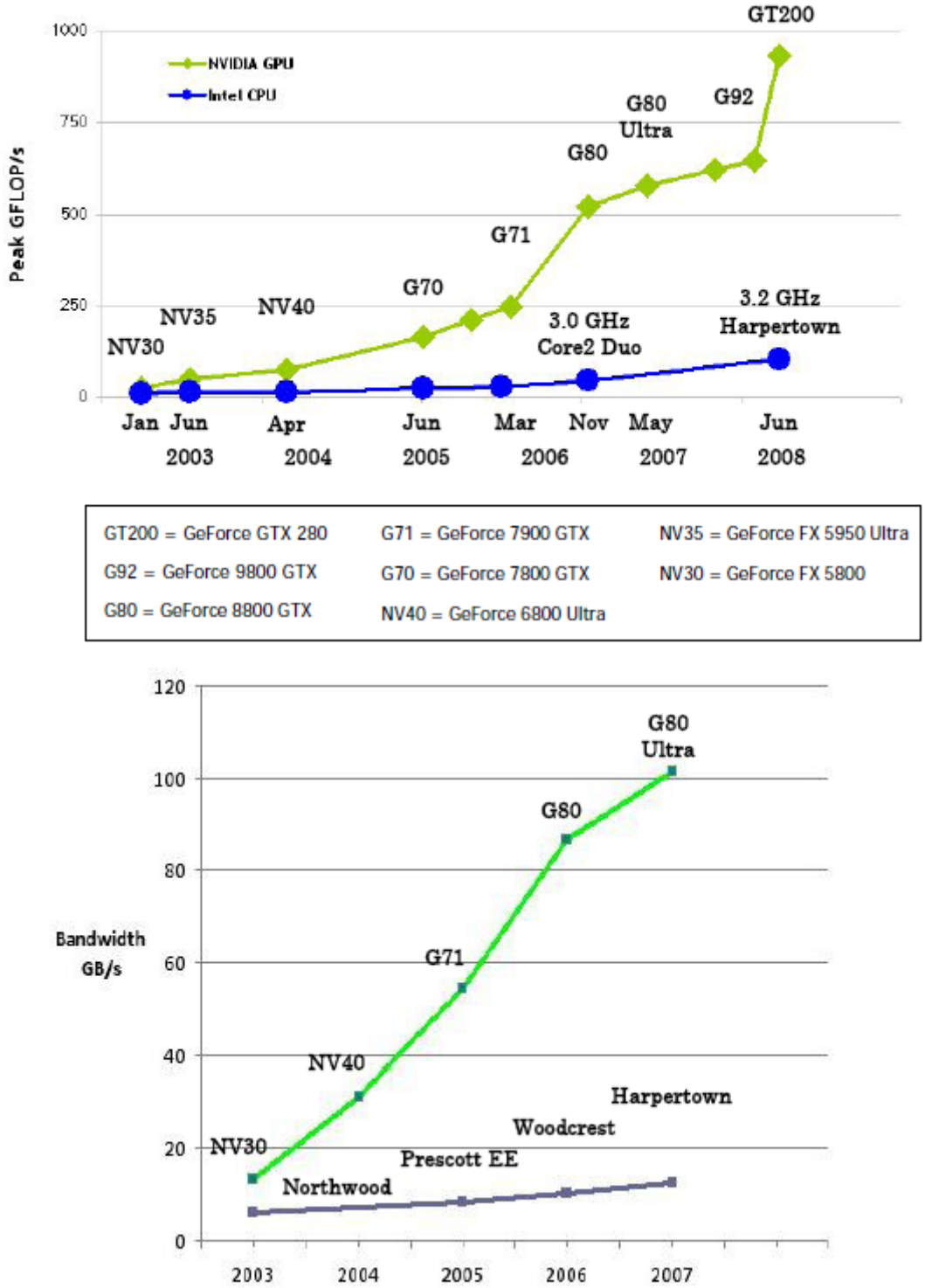
2.2 GPU

Graphic Processing Unit - Grafická procesorová jednotka

Jedná se o čip na grafické kartě počítače obsahující stovky milionů tranzistorů, mozek grafické karty, vypočítává data, jež dostává od hlavního procesoru. Grafický čip se stará o vykreslování grafiky a data z výstupu jsou potom posílána na zobrazovací zařízení (monitor, projektor a další). Princip ALU jednotek spolupracujících v rámci GPU vede k silnému paralelnímu výpočetnímu výkonu. Začíná se stále častěji využívat i při vědeckotechnických výpočtech, které jsou specializovaný na jejich výpočetní jednotky, které pracují paralelně a tak dosahují mnohdy 10krát a více ve výkonu oproti standardním výpočtům na CPU.

Dnešní GPU se díky obrovské tržní poptávce po složitých 3D grafických operacích prováděných v reálném čase vyvinuly ve vysoce paralelní, mnohovláknové a vícejadrové procesory s obrovským výpočetním výkonem a velmi velkou šířkou pásma paměti. Vývoj je zobrazen na Obrázek 2.1. Na obrázku je jasně vidět trend posledních let, kde GPU zvyšuje svůj výkon téměř exponenciálně, kdežto výkon CPU se zvyšuje mnohonásobně pomaleji.

Tuto vlastnost využijeme pro zrychlení zpracovaní geometrických transformací a interpolaci jednotlivých bodů. Požadavkem je dosažení takové rychlosti, abychom byli schopni geometricky transformovat i obraz videa. To znamená zvládnout alespoň 24 snímku za jednu vteřinu.



Obrázek 2.1: Operace v plovoucí čárce za vteřinu a šířka pásma pamětí CPU a GPU, převzato z [13]

2.3 Porovnání součastných CPU a GPU

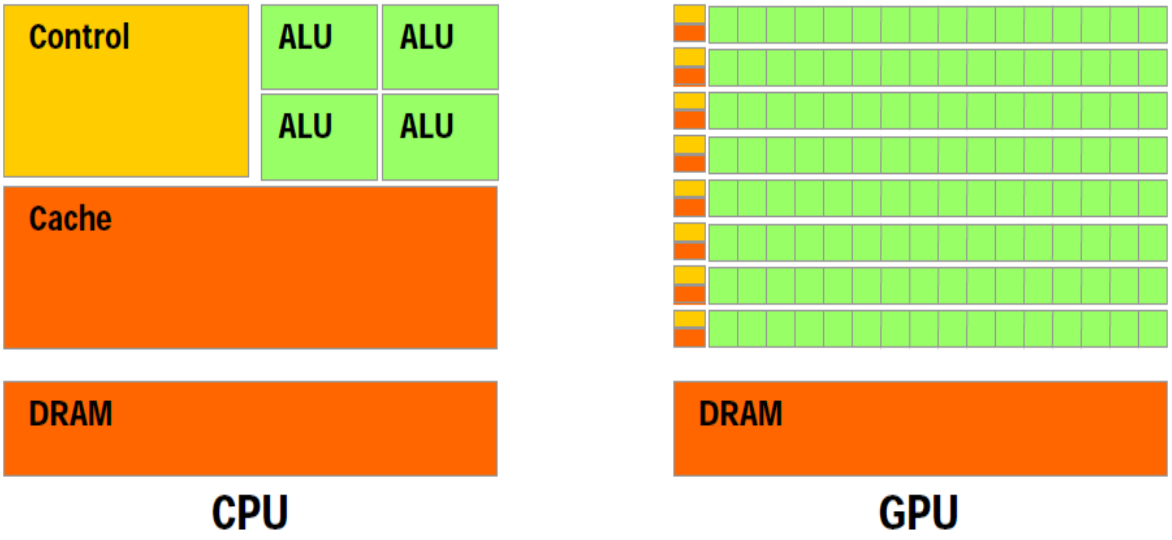
Grafické karty dokáží v určitých případech (například v případech simulací či finančních analýz) počítat až 150x rychleji než CPU. Klíčem k úspěchu jsou zde flexibilní, nezávisle programovatelné shadery moderních GPU, které z nich činí „General Purpose GPU" (GP GPU, víceúčelové GPU).

Pro srovnání použijeme jedny z nejnovějších čipů od dvou největších výrobců grafických čipů. Firmu nVidia bude zastupovat jádro GT200, které můžeme najít v kartách s označením GeForce GTX260 a GeForce GTX280. Druhou firmou je firma ATI s chipem RV770, který je obsažen v kartách Radeon HD4850 a Radeon HD4870.

GT200 dosahuje výkonu až 933 GFLOPS (FLOPS je zkratka z anglického Floating Point Operation Per Second, tedy výkon s operacemi v plovoucí čárce), RV770 však dosahuje výkonu téměř 1 200 GFLOPS. Pro srovnání - čtyřjádrový procesor Intel Core 2 Quad Q6600 ve stejném testu vykazuje 21,4 GFLOPS, takže zdaleka nedosahuje výkonu GPU.

Důvodem tak markantního rozdílu výkonu výpočtu v plovoucí čárce mezi CPU a GPU je v tom, že GPU je specializovaná pro tyto výpočty a je vysoce paralelní, přesně to co je potřeba pro vytváření grafiky a tedy navržen tak, že většina tranzistorů je věnována na zpracování dat, nikoliv řízení, či ukládaní dat jak schematicky ilustruje Obrázek 2.2.

Největší nevýhodou GPU v porovnání s CPU je to, že hodnoty s plovoucí čárkou lze počítat pouze 32bitově. Složité kalkulace velkého množství dat vyžadují někdy vyšší přesnost a 64bitové zpracování. ATI i nVidia reagují na tyto požadavky úpravou současných GPU, která jim zajistí dvojnásobnou přesnost při výpočtech s plovoucí čárkou. V současné době tato vlastnost ovšem stále není dostupná.



Obrázek 2.2: Tranzistory GPU jsou více zaměřeny na zpracování dat, převzato z [13]

2.4 Co je CUDA?

CUDA - Compute Unified Device Architecture

Nástup vícejádrových procesorů a mnohojádrových grafických čipů znamená, že i běžné počítače se dnes stávají paralelními systémy. Navíc jejich paralelismus jde stále dál i s Moorovým zákonem, který říká: „složitost součástek se každý rok zdvojnásobí při zachování stejné ceny“.

CUDA je obdoba programovacího jazyku C pro GPU na grafických kartách nVIdia, přesněji pak paralelní programovací model a softwarové prostředí navržené k jednoduchému a rychlému překonání problému paralelismu výpočtu mezi jádry grafické karty při zachování nízkého nároku na učení pro programátory jazyka C. K základnímu programování v prostředí CUDA musí programátor pochopit, jak pracují tři základní prvky. Jedná se o hierarchii skupin vláken, sdílenou paměť a synchronizace. CUDA je navržena tak, že uživatel vůbec nepotřebuje znát počet využitelných GPU. Program lze psát jednoduše tak, že napíšeme jednotlivé jádra a ty pak spouštíme paralelně.

CUDA je volně přístupná a obsahuje rozsáhlou dokumentaci a podporu ze strany vývojářů. Do budoucna by kromě jazyku C měla přibýt také podpora jazyku C++ a Forton. CUDA je podporována grafickými kartami s čipem označeným minimálně 8xxx.

Podrobněji si části CUDA a programování v tomto modelu popíšeme v následujících podkapitolách.

2.5 Jádra

CUDA verze pro jazyk C jej rozšiřuje zavedením speciálních funkcí zvaných „kernels“ neboli jádra. Takové jádro se liší od standardní funkce tím, že není spuštěno jednou, ale hned N-krát. A to paralelně pomocí N-různých CUDA vláken.

Vytvoření jádra doprovází speciální syntaxe. Funkce je označena klíčovým slovem **__global__** a každé spuštění takového jádra musí být určeno počtem CUDA vláken. Tento počet je zadán mezi dalšími speciálními značkami: „<<<“ a „>>>“. Pro ukázku si vytvoříme prázdné jádro, které použijeme i v dalších příkladech.

```
// definice jádra
__global__ void sample_kernel(float* A, float* B, float* C)
{
      // Kód jádra
}
int main()
{
      // spuštění jádra
      sample_kernel <<<1, N>>>(A, B, C);
}
```

Každé vlákno, které provádí jádro, dostane jedinečné ID, které je dostupné v jádru pro programátora skrz proměnnou **threadIdx.** Pro ilustraci použijeme již dříve vytvořené jádro. Ukázkové jádro sečte pole A a pole B. Výsledek uloží do pole C. Všechna pole mají stejnou velikost. Velikost je N a N-krát také spustíme jádro, jak můžeme vidět v hlavní funkci. Každé vlákno tak provede jeden součet.

```
// definice jádra
__global__ void sample_kernel (float* A, float* B, float* C)
{
      int i = threadIdx.x;
      C[i] = A[i] + B[i];
}

int main()
{
      // spuštění jádra
      sample_kernel <<<1, N>>>(A, B, C);
}
```

2.6 Hierarchie vláken

Již zmíněná proměnná **threadIdx** je trojrozměrný vektor, takže může být adresován jako jedno, dvou nebo tří-dimenzionální a tím vytvářet jedno, dvou nebo tří-dimenzionální blok vláken. Toto rozdělení přirozeně vytváří jednoduchou práci s elementy jako je vektor, pole, matice apod. Dvou-rozměrné použití si ukážeme na příkladu, ve kterém se sčítají dvě stejně velké matice o rozměrech NxN a výsledek je uložen do matice C:

```
// definice jádra
__global__ void MatAdd(float A[N][N], float B[N][N],float C[N][N])
{
      int i = threadIdx.x;
      int j = threadIdx.y;
      C[i][j] = A[i][j] + B[i][j];
}

int main()
{
     // spuštění jádra
     dim3 dimBlock(N, N);
     MatAdd<<<1, dimBlock>>>(A, B, C);
}
```

Index vlákna a jeho ID má k sobě jednoduchou vazbu. Pokud se jedná o jedno-dimenzionální blok, je index i ID totožné. Při použití dvourozměrného bloku o velikosti (Dx,Dy) se ID vypočte pro vlákno s indexem (x,y), rovnicí ID = (x + y * Dx). Při použití tři-dimenzionálního bloku o velikosti (Dx, Dy, Dz) se ID vypočte pro vlákno s indexem (x, y, z) rovnicí (x + y*Dx + z*Dx*Dy).

Vlákna v rámci jednoho bloku mohou mezi sebou spolupracovat sdílením dat skrz jednu ze sdílených pamětí a synchronizovat své výpočty ke koordinaci přístupu k paměti. Přesněji řečeno, je možné určit body synchronizace v jádře, voláním vnitřní funkce **__syncthreads** ().Tato funkce funguje, jako určitá bariéra před kterou se všechny jádra zastaví a musí počkat než na stejné místo dorazí všechny ostatní vlákna, až poté mohou pokračovat všechny vlákna nezávisle dál.

Pro účinnou spolupráci se sdílenou pamětí se očekává paměť s velmi rychlou odezvou u každého jádra procesoru, dále to, že funkce **__syncthreads()** zabere zanedbatelně málo času a předpokládá se to, že všechny vlákna jednoho bloku přísluší ke stejnému jádru procesoru. Počet

vláken na blok je tedy omezen limitem paměťových prostředků jednoho procesorového jádra. Na dnes používaných GPU může blok vláken obsahovat až 512 vláken.

Jádro může být spuštěno pomocí několika bloků vláken, které jsou pravidelné, a každý blok obsahuje stejný počet vláken. Celkový počet použitých vláken pak dostaneme vynásobením počtu použitých bloků a počtu vláken na jeden blok. Bloky vláken jsou uspořádány v jednorozměrné nebo dvourozměrné mřížce jak můžeme vidět na Obrázek 2.3. Rozložení mřížky se určuje prvním parametrem syntaxe „<<<….>>>". Každý blok uvnitř mřížky je jednoznačně identifikovatelný jednodimenzionálním nebo dvojdimenzionálním indexem uvnitř jádra pomocí vnitřní proměnné blockIdx. Rozměry bloku jsou pak přístupné uvnitř jádra pomocí vnitřní proměnné blockDim. Na dalším příkladu pak ukážeme použití těchto proměnných pro výpočet aktuálně počítaného prvku dvourozměrného pole. Indexy souřadnic pole se vypočítají vynásobením šířky bloku a jeho indexu. Tím dostaneme index, který reprezentuje první vlákno daného bloku. Proto k němu musíme přičíst ještě aktuální index vlákna. Upravený původní příklad:

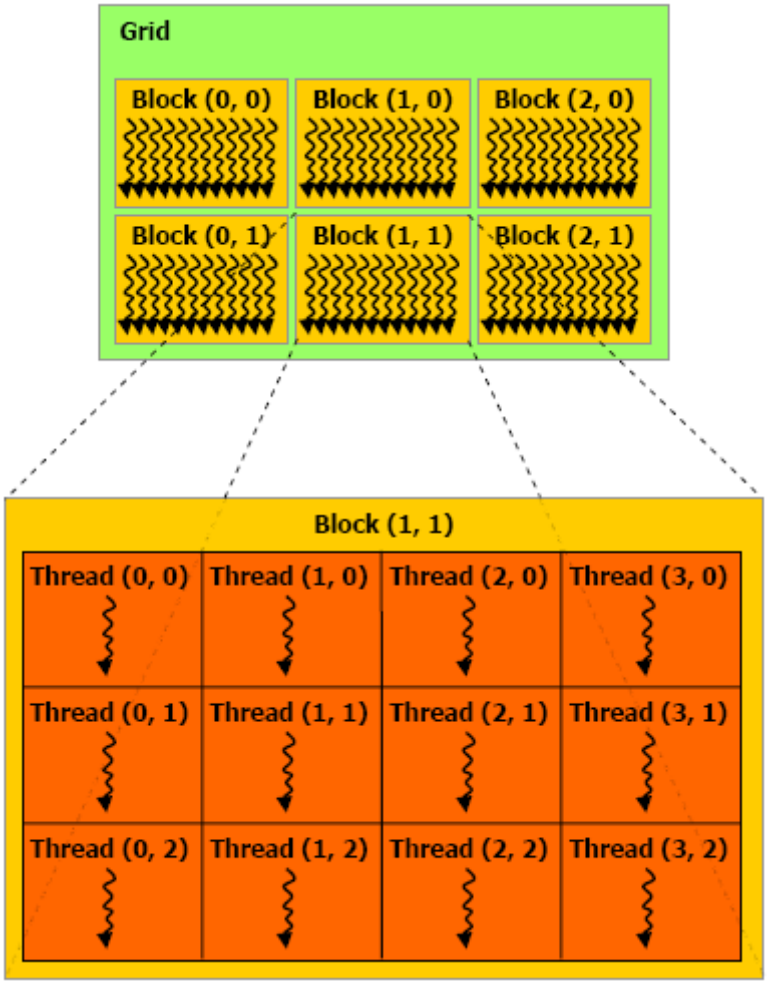
```
// definice jádra
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
      // výpočet indexu vlákna probíhá přesně podle vzorce popsaného
      // v minulém odstavci
      int i = blockIdx.x * blockDim.x + threadIdx.x;
      int j = blockIdx.y * blockDim.y + threadIdx.y;
      // kontrola zda nejsme za hranicemi pole
      if (i < N && j < N)
      C[i][j] = A[i][j] + B[i][j];
}

int main()
{
      // spuštění jádra
      // jádro bude mít bloky velikosti 16x16
      dim3 dimBlock(16, 16);
      // počet bloků je ošetřen tak, aby se provedl potřebný počet
vláken
      dim3 dimGrid((N + dimBlock.x - 1) / dimBlock.x,
      (N + dimBlock.y - 1) / dimBlock.y);
      MatAdd<<<dimGrid, dimBlock>>>(A, B, C);
}
```

Velikost bloku 16x16 je nejběžněji používaná velikost bloku, ovšem tato velikost může být nastavena jakkoliv jinak. Při nastavování počtu použitých bloků je důležité toto číslo volit tak aby byl proveden dostatečný počet vláken. Vzorec `(N + dimBlock.x - 1) / dimBlock.x` nám zajistí při velikosti pole N to, že počet bloků bude pro jednodimenzionální pole dostatečný. Obdobně pak provedeme výpočet ve druhé dimenzi pro dvoudimenzionální pole.

Bloky vláken musí být prováděny nezávisle na sobě, musí být možné je provádět v libovolném pořadí, paralelně nebo sériově. Tento požadavek nezávislosti umožňuje blokům vláken být spouštěna v jakémkoliv pořadí na jakémkoliv počtu procesorových jader, což umožní programátorům psát kód bez nutnosti znát počet procesorových jader.

Vlákna se neprovádí úplně samostatně a nezávisle na ostatních. Jsou i v rámci bloků rozdělena do takzvaných **warpů.** Tyto svazky vláken fungují tak, že všechny vlákna v rámci jednoho warpu provádějí ve stejnou chvíli stejnou instrukci. A to jak operace výpočetní, tak operace přistupující do paměti. Pokud je v těle jádra někde podmínka `if` a jedna část vláken uvnitř warpu splní podmínku, druhá nesplní a bude pokračovat větví `else`, bude se nejprve provádět jedna z těchto větví a všechna vlákna, která budou chtít provádět druhou větev, budou v nečinnosti čekat a až po dokončení první větve se začnou provádět. Během provádění druhé větve pak budou čekat vlákna, která už provedla svou část v minulém kroku. To znamená, že nemá smysl vkládat do jádra podmínky na zkrácení provádění, pokud je pravděpodobné, že tuto podmínku nesplní všechna vlákna.



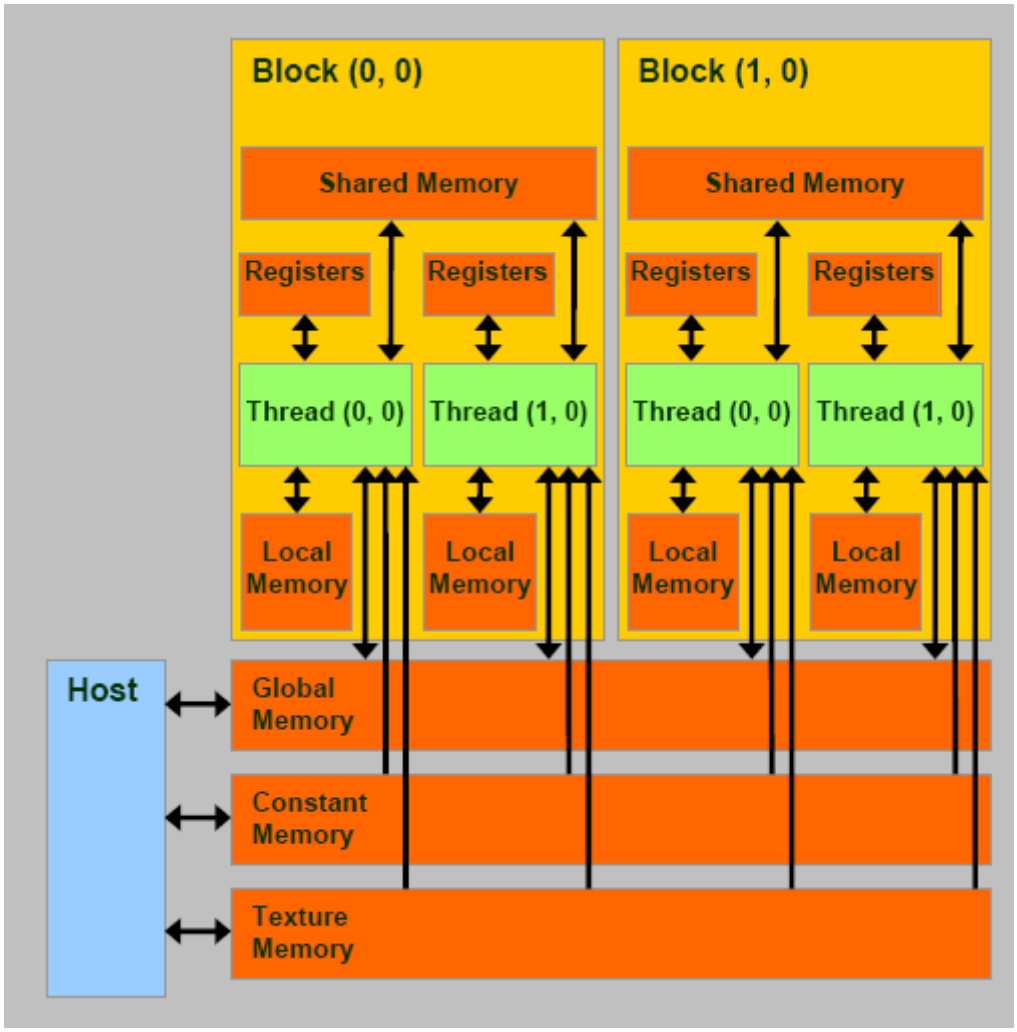
Obrázek 2.3: Hierarchie vláken, převzato z [13]

2.7 Hierarchie pamětí

Vlákna mohou během svého provádění přistupovat k datům z různých pamětí, jak je patrné na Obrázek 2.4. Každé vlákno má svou vlastní lokální paměť a své registry. Každý blok vláken má svojí sdílenou paměť, která je přístupná pro všechny vlákna daného bloku. Sdílená paměť vzniká a zaniká společně se vznikem a zánikem bloku vláken. Všechny vlákna mohou navíc přistupovat do globální paměti. Vyjmenované paměti slouží jak pro čtení, tak pro zápis dat.

Dále mohou všechna vlákna přistupovat do dvou dalších pamětí, které jsou společné pro všechny, ale jsou pouze pro čtení dat. Jedná se o konstantní a texturovací paměť.

Každá paměť je navržena a optimalizovaná pro jiné použití a v závislosti na volbě paměti dochází k naprosto odlišným časovým náročnostem programu. Jednotlivé paměti si vysvětlíme dále. V Tabulka 2-1 jsou shrnuty nejdůležitější údaje všech dostupných pamětí.



Obrázek 2.4: Paměti na grafické kartě, převzato z [15]

Globální paměť

Jedná se o největší paměť a tvoří téměř celou kapacitu paměti grafické karty. Jelikož je tato paměť jedinou, ze které může číst CPU, používá se pro výstup dat z grafické karty zpět do hlavního programu. Na druhou stranu není příliš vhodná k běžnému používání pro práci s daty ve funkcích jádra. Globální paměť nemá vyrovnávací paměť, proto je velmi důležité ji správně používat, aby nedocházelo při čtení a zápisu dat z této paměti k příliš velkému nárůstu časové náročnosti programu. Nejefektivnějšímu přístup k této paměti dochází při čtení nebo zápisu všech vláken najednou. Pokud tato situace nastane, je tato instrukce optimalizována tak, že se čtou větší bloky dat a tím se sníží počet přístupu k paměti. Tato operace je prováděna automaticky. CUDA ovšem nabízí i několik funkcí, které umožňují programátorovi čtení většího počtu dat v jedné instrukci a tím také zmenšení počtu přístupu k paměti. Tyto funkce jsou ale dost specifické, proto je zde nebudeme dále popisovat.

K práci s globální paměti je třeba ji nejdříve alokovat v hlavním programu. To je možné udělat několika způsoby. Nejjednodušším je použít funkci **cudaMalloc()**, která vrací ukazatel na pole alokované na device. Tento parametr je pak třeba předat jádru při jeho volání.

Lokální paměť

Stejně jako globální paměť, ani lokální nemá vyrovnávací paměť, takže přístup k datům je časově stejné náročný jako u globální paměti. Čtení z lokální paměti zabere stejně jako čtení z paměti globální o 400 až 600 hodinových cyklů více, než například z konstantní nebo sdílené paměti. Výhodou lokální paměti je, že data jsou viditelná pouze v rámci jediného vlákna.

Do lokální paměti jsou automaticky ukládaný všechny struktury a pole, vytvořená uvnitř jádra, která by zabrala příliš mnoho místa v registrech a také pole, u kterých kompilátor nemůže zjistit, zda budou mít konstantní velikost.

Konstantní paměť

Konstantní paměť je první s vyrovnávací pamětí. Jak již bylo uvedeno dříve rozdíl mezi paměti s vyrovnávací paměti a bez ní je značný. Pokud jsou data, která jsou jádrem požadována uložena ve vyrovnávací paměti, je přístup k nim kolem osmi hodinových cyklů. Pokud ne, je čtení stejně pomalé jako u paměti globální.

Pro všechna vlákna v rámci jednoho bloku je čtení z konstantní paměti stejně rychlé jako z registru, dokud čtou data ze stejné adresy. Časová náročnost lineárně roste podle počtu přístupu k různým adresám jednotlivými vlákny. Proto je doporučeno, k dosažení maximální rychlosti, aby všechna vlákna ve stejnou chvíli využívala data z jediné adresy.

Velikost konstantní paměti je většinou 64kB.

Texturovací paměť

Pro tuto paměť platí, co se týče rychlosti přístupu k datům to stejné jako pro konstantní paměť. Stejně jako konstantní, také obsahuje vyrovnávací paměť. Vyrovnávací paměť je optimalizována pro 2D prostor, proto vlákna, která čtou data, uložená v texturovací paměti v blízkosti kolem sebe, dosáhnou nejlepších výsledků.

Čtení z grafické paměti pomocí speciálních funkcí pro textury může přinést výhody, oproti alternativám čtení grafické paměti z globální nebo konstantní paměti:

- Odezva adresování paměti je mnohem lépe optimalizována, proto oproti jiným pamětem přináší lepší výsledky při náhodném přístupu k datům.
- Data načítaná po větších blocích mohou být rozdělena do více proměnných během jediné operace.
- Adresace může být prováděna pomocí 16 nebo 32 bitových integerů, nebo může být převedena na 32bitové adresování v plovoucí čárce v rozsahu [0,0 až 1,0] nebo [-1,0 až 1.0]

Nicméně, ani během provádění jediného jádra není texturovací vyrovnávací paměť chráněna proti zápisu skrz globální paměť. Pokud tedy dojde pomocí globální paměti k zápisu dat, která jsou následně čtena z textury, jsou načtena data nedefinovatelná. Toto jde brát v úvahu pouze tehdy, pokud je textura přiřazena nad lineární paměti. Pokud je přiřazena nad CUDA polem, tento problém odpadá, protože jádro nemůže do tohoto pole zapisovat.

Sdílená paměť

Vzhledem k tomu, že se tato paměť nachází přímo na grafickém čipu, přístup k datům je mnohem rychlejší než k lokální nebo globální paměti. Ve skutečnosti, pro všechna vlákna v rámci jednoho warpu, je přístup ke sdílené paměti stejně rychlý jako k registru do té doby, než dojde mezi vlákny ke konfliktu paměťových bank.

Pro dosažení vysoké šířky pásma je sdílená paměť rozdělena na stejně velké paměťové moduly, zvané paměťové banky, ke kterým se může přistupovat současně. Takže jakékoliv čtení nebo zápis, který požaduje čtení nebo zápis N různých adres, které spadají do N různých paměťových bank, může být prováděn současně. Tímto je reálná šířka pásma N-krát větší než šířka pásma jediného modulu.

Největší nevýhodou sdílené paměti je její malá velikost. V současné době je velikost sdílené paměti u všech grafických typů, podporujících CUDA, velký 16KB.

Registry

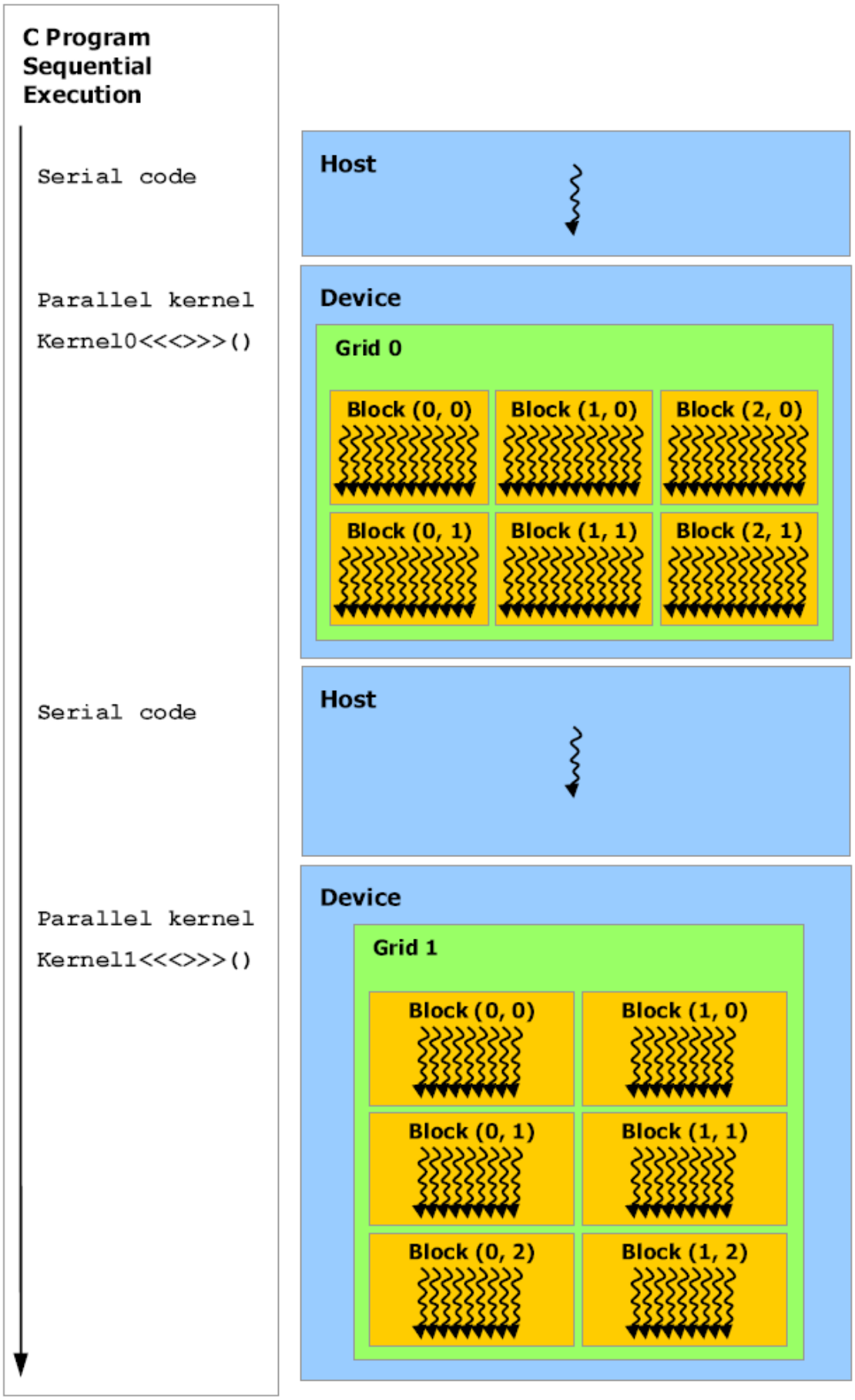
Obecně platí, že přístup k registrům nestojí žádný hodinový cyklus navíc, než kolik stojí instrukce, ale zpoždění může nastat díky závislostem čtení ihned po zápisu, nebo díky konfliktům registrových paměťových bankách. Kompilátor a plánovač vláken se snaží instrukce optimalizovat co nejvíce tak, aby nedocházelo k právě zmíněným konfliktům. K nejlepším výsledkům dochází, pokud je počet vláken na jeden blok vláken, násobkem 64. Počet registrů je omezen a je podstatným prvkem, který určuje možnost spuštěných bloků na jednom multiprocesoru.

Paměť	Umístění	Vyrovnávací paměť	Práva	Přístup
Registry	Na čipu	Ano	Čtení / zápis	Jediné vlákno
Lokální paměť	Mimo čip	Ne	Čtení / zápis	Jediné vlákno
Sdílená paměť	Na čipu	Ano	Čtení / zápis	Všechna vlákna v rámci bloku
Globální paměť	Mimo čip	Ne	Čtení / zápis	Všechna vlákna + CPU
Konstantní paměť	Mimo čip	Ano	Čtení	Všechna vlákna + CPU
Texturovací paměť	Mimo čip	Ano	Čtení	Všechna vlákna + CPU

Tabulka 2-1: Souhrn pamětí grafické karty

2.8 Rozdíl mezi programem na CPU a GPU

Běh programu je postaven tak, že je spuštěn hlavní program. Ten má na starost připravení veškerých potřebných dat jak pro sebe, tak pro jádro. To znamená, že alokuje paměť pro svoje potřeby na operační paměti počítače, alokuje paměť a plní ji pro potřeby jádra. Stará se o paměť globální, texturovací nebo konstantní. Ostatní paměti jsou řízeny přímo z jádra. Po alokaci paměti naplní tyto prostory, daty podle potřeby, to znamená například to, že nakopíruje data z operační paměti do globální paměti na grafické kartě. Po provedení všech potřebných příprav, spustí jádro. To je prováděno na grafickém čipu. Po provedení všech vláken je třeba vypočtená data opět převést z grafické karty do operační paměti, tak aby s nimi mohl program dále pracovat. Zjednodušený princip práce hlavního programu a jádra je přiblížen na obrázku 2.5.



Obrázek 2.5: Princip vykonávání programu rozděleného na CPU a GPU, převzato z [15]

Označení CUDA funkcí

CUDA zavádí tři speciální kvalifikátory funkcí. Ty určují, zda se funkce bude provádět na CPU nebo na GPU a zda je možno danou funkci volat z CPU či GPU. Stručně jsou následující údaje shrnuty v Tabulka 2-2.

Kvalifikátor **__device__** označuje funkci, která je prováděna na GPU a je jí možno volat také pouze z funkce prováděné na GPU.

Dalším kvalifikátorem je **__global__**, který označuje jádro. Jak již bylo řečeno dříve, jádro je funkce spouštěna z CPU a je prováděná na grafické kartě. Takto označena funkce musí vracet `void`.

Posledním kvalifikátorem je **__host__**. Takto označená funkce je prováděna na CPU a je jí možno volat také pouze z CPU. Funkce, která je označená kvalifikátorem **__host__** nebo neobsahuje žádný kvalifikátor, bude zkompilována úplně stejně.

Pokud je funkce označena jak kvalifikátorem **__host__**, tak kvalifikátorem **__device__**, bude zkompilována, jak pro CPU, tak pro GPU. Naopak nelze použít společně **__global__** a **__host__**.

K různým kvalifikátorům se vážou různá omezení funkcí. Pro **__device__** a **__global__** platí, že nepodporují rekurzi, uvnitř svého těla nemohou deklarovat statické proměnné a nemohou mít různý počet argumentů.

Kvalifikátor	Prováděn na:	Volán pouze z:
__device__ float DeviceFunc()	GPU	GPU
__global__ void KernelFunc()	GPU	CPU
__host__ float HostFunc()	CPU	CPU

Tabulka 2-2: Stručný souhrn kvalifikátorů funkcí v CUDA

3 Geometrické transformace obrazu

Tato kapitola pouze lehce přiblíží, co jsou geometrické transformace obrazu. Bude zde vysvětleno několik základních pojmů.

Tato kapitola je převzata z [1].

Geometrické transformace jsou jedním z nejdůležitějších a nejpoužívanějších nástrojů moderní počítačové grafiky. Prvním krokem pro akceleraci počítačové grafiky proto většinou bývá hardwarová implementace právě geometrických transformací.

Geometrické transformace můžeme chápat jako změnu pozice vrcholu v aktuálním souřadnicovém systému nebo jako změnu souřadnicového systému.
Mezi základní geometrické transformace obrazu patří:

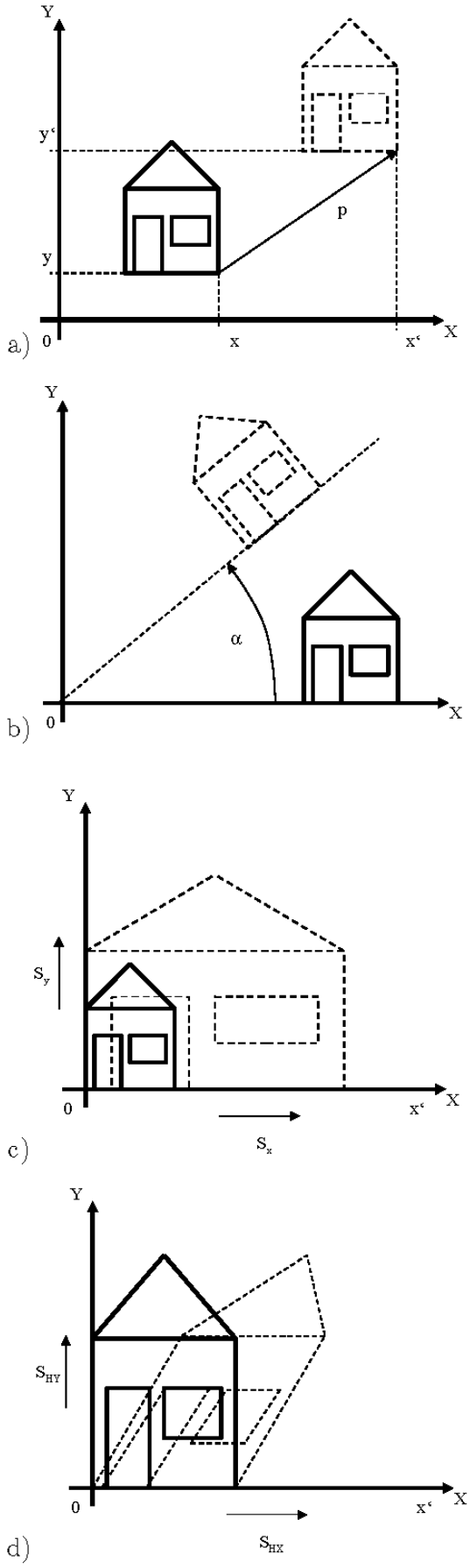
- **Měřítko**
- **Posun**
- **Rotace**
- **Zkosení**

Pomocí výše uvedených základních transformací lze provádět i transformace složitější a to jejich skládáním. Graficky jsou tyto transformace zobrazeny na Obrázek 3.1.

Geometrické transformace vypočtou na základě souřadnic bodů ve vstupním obraze, souřadnice bodů ve výstupním obraze. V digitálním zpracování obrazu geometrické transformace dovolují odstranit geometrické zkreslení vzniklé při pořízení obrazu (např. korekce geometrických vad objektivu kamery, oprava zkreslení družicového snímku způsobená zakřivením zeměkoule). Mezi korekcí geometrického zkreslení a geometrickými transformacemi, jako je posunutí nebo rotace, není principiální rozdíl. Použijí se stejné algoritmy.

Geometrická transformace plošného obrazu je vektorová funkce, která zobrazí bod do nového bodu. Transformace je definována dvěma složkovými vztahy:

$$x' = T_x(x, y), \;\; y' = T_y(x, y) \tag{3.1}$$



Obrázek 3.1: Geometrické transformace: a) posun, b) rotace, c) měřítko, d) zkosení, převzato z [1]

Transformační rovnice T_x a T_y mohou být buď známy předem, jako je tomu např. v případě rotace, posunu nebo zvětšení obrazu, nebo je možné hledat transformační vztah na základě znalosti původního i transformovaného obrazu. Při hledání transformace se obvykle využívá několika známých (tzv. lícovacích) bodů, které v obou obrazech odpovídají identickému objektu a lze je snadno najít.

Geometrická transformace se skládá ze dvou kroků:

- **Transformace souřadnic bodů** – najde k bodu ve vstupním obraze s diskrétními souřadnicemi odpovídající bod ve výstupním obrazu. U výstupního bodu musíme počítat se spojitými souřadnicemi (reálná čísla), protože poloha výsledného bodu nemusí souhlasit s celočíselnou mřížkou. Tato plošná transformace má tedy bodový charakter.
- **Interpolace obrazových bodů** – té bude věnována celá další kapitola.

Lineární transformace je zobrazení f z jednoho vektorového prostoru do druhého $V \rightarrow W$, které zachovává lineární kombinace. Pro libovolný vektor $x(x_1, x_2, \dots x_n)$ a skalár $\alpha_1, \alpha_2, \dots \alpha_n$ platí $f(\alpha_1 x_1,\ \alpha_2 x_2, \dots \alpha_n x_n) = \alpha_1 f(x_1),\ \alpha_2 f(x_2), \dots \alpha_n f(x_n)$. Všechny základní geometrické transformace jsou lineární.

4 Interpolace

Interpolace je proces stanovení hodnoty funkce v bodu, ležícím mezi dvěma vzorky. Tímto procesem vzniká kontinuální funkce z původních diskrétních vstupních vzorků. Zatímco vzorkování generuje nekonečnou šířku pásma signálu z jednoho, který je kmitočtově omezen, interpolace je v opačné roli: snižuje šířku pásma signálu aplikováním dolní propusti na diskrétní signál. To znamená, že interpolace rekonstruuje signál ztracený při vzorkování, vyrovnáváním získaných dat pomocí interpolační funkce.

Zdroje této kapitoly jsou [2], [3], [4], [5], [6], [7].

4.1 Interpolace obrazových bodů

Proces interpolace je jedním ze základních operací při zpracování obrazu. Manipulace s digitálními obrazy často vyžaduje převzorkováni obrazových bodů. Nejčastěji se jedná o použití interpolační metody po aplikování geometrické transformace. Obvykle pozice transformovaného bodu nezapadá přímo do souřadnicového systému. Proto musí být použit, k určení bodu výsledného obrazu, interpolační algoritmus. Časová náročnost algoritmu, která je ve většině případů úměrná jeho kvalitě, je rozhodujícím faktorem pro výslednou kvalitu transformovaného obrazu.

Interpolace, ve spojení s geometrickými transformacemi, se dělá tak, že se vezme bod transformovaného obrazu, určí se jeho umístění v původním obraze a vypočítá se podle vybrané metody barva nového bodu. Hodnota barvy se vypočítá, jako součet násobení okolích hodnot barev s jejich váhami.

V následujících kapitolách si popíšeme nejpoužívanější metody interpolací obrazových bodů. Pro přiblížení si všechny typy předvedeme na příkladech. Jako zdroj byl použit Obrázek 4.1 větvičky, o rozměrech 200x124 bodů, který jsme pomocí každé metody zvětšili na trojnásobek, to znamená na rozměr 600x372 bodů. Test byl proveden v programu Adobe Photoshop CS3 za pomoci integrovaných funkcí.

Obrázek 4.1: Původní obrázek

4.2 Nejbližší soused

Tato metoda interpolace je nejjednodušší a byla pravděpodobně první metodou používanou pro zvětšování rastrových obrazů. Barvu bodu v transformovaném obraze tvoří barva nejbližšího bodu původního obrazu. V této metodě se nepočítají žádné váhy, jak bylo uvedeno dříve. Nejbližší bod má automatický váhu rovno jedné. Pokud dojde ke zvětšení obrazu například na trojnásobek velikosti, bude jeden bod původního obrazu představovat 3x3 bodu výsledného obrazu, samozřejmě se stejnou

barvou. Pokud by došlo ke zmenšení, řekněme dvojnásobnému, došlo by k redukci bodů a to tak, že by se do výsledného obrazu promítl pouze každý druhý bod.

Mohlo by se zdát, že vzhledem k nevýhodám, které s sebou metoda „nejbližšího souseda“ přináší, se už tento postup při zvětšení fotografií nepoužívá. Není tomu tak – setkáte se s ním například v některých programech při zvětšení na obrazovce nástrojem „lupa“ nebo je využita rychlost této metody při zobrazování rychlých náhledů před dopočítáním pomocí bilineární, bikubické či jiné náročnější interpolace, kterou se tento snímek překryje. Pro trvalé zvětšování fotografií je však postup založený na kopírování okolních bodů nevhodný.

Na Obrázek 4.2 můžeme vidět, jak se každý bod změnil ve čtverec. Tyto čtverce pak tvoří na šikmých hranách schodovité přechody. Ovšem obraz není vůči původnímu nijak rozostřen. Při zpětné transformaci – transformaci zmenšení na původní velikost, bychom dostali opět původní obraz bez jakékoliv ztráty kvality.

Obrázek 4.2: Zvětšený obrázek pomocí metody nejbližší soused

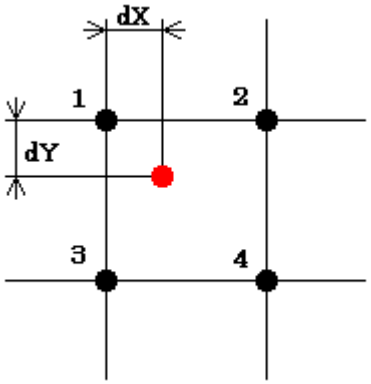
4.3 Bilineární interpolace

Bilineární interpolace se skládá ze dvou lineárních interpolací. Metoda určuje novou hodnotu podle váženého průměru čtyř bodů v nejbližším 2x2 okolí původního obrazu. Lineární metoda znamená, že se váhy určí jako pouhá vzdálenost bodu v původním obrazu od transformovaného bodu.

Na Obrázek 4.3 je názorně ukázáno, jak vypadá okolí transformovaného bodu. Nejbližší okolní čtyři body se použijí pro výpočet.

Nejprve se vypočte hodnota funkce prvního řádku. V případě bilineární interpolace se jedná o použití dvou bodů. U každého z těchto bodu vypočteme váhu, podle rovnice (4.1), kde za x dosadíme X-ovou vzdálenost příslušného a transformovaného bodu. Váhu poté vynásobíme hodnotou barevné složky daného bodu. Dostaneme dvě čísla, která sečteme. Tím dostaneme hodnotu prvního řádku. Totéž provedeme pro druhý řádek. Jako poslední krok provedeme obdobný výpočet jako pro první a

druhý řádek, ovšem váhy vypočteme pomocí Y-ové vzdálenosti a jako barvy bodů použijeme vypočtené hodnoty prvního a druhého řádku.



Obrázek 4.3: Okolí bodu při bilineární interpolaci

Tato metoda odstraňuje neduhy minulé metody. Průměrování většího počtu okolních bodů má za následek vytvoření anti-aliasing efektu, takže nedochází k vytváření ostrých hran. Proto je výsledný obraz z pohledu pozorovatelé reálnější. Jak můžeme vidět na Obrázek 4.4, hrany nejsou tak schodové jako u minulé metody, ovšem pořád jsou vidět i při tak malém zvětšení pouhým okem.

Vypočet váhy bodu použitého při výpočtu:

$$w = 1 - x \tag{4.1}$$

Proměnná w je váha používaného bodu, proměnná x pak udává jeho vzdálenost od transformovaného bodu.

Obrázek 4.4: Zvětšený obrázek pomocí Bilineární interpolace

4.4 Bikubická interpolace

Bikubická interpolace je opět složitější, než předchozí uvedená metoda. Jak již bylo uvedeno, čím složitější metoda, tím lepší výsledný obraz. Místo linií využívá křivek, kterými spojuje šestnáct okolních obrazových bodů a pomocí nich pak vypočítá potřebnou barevnou hodnotu pro výsledný bod. Bikubické interpolace lze dosáhnout buď použitím Lagrangeových polynomů, kubické spline křivky, nebo kubického konvolučního algoritmu. Tyto matematické metody jsou vysvětleny dále.

Z pohledu použitelnosti je tato metoda nejlepší volbou. Má dostatečně kvalitní vizuální výsledky za přijatelný nárůst času oproti metodám nejbližší soused či bilineární interpolace.

Bikubická interpolace se používá ve více variantách. Například Adobe Photoshop nabízí tři varianty této metody:

- Klasickou Bikubickou
- Hladší Bikubickou – doporučena pro zvětšování obrazu
- Ostřejší Bikubickou – doporučena pro zmenšování obrazu

Pro zvýšení přesnosti při zvětšení či jiné transformaci se postupuje takzvaným pravidlem „110 procent“. Toto pravidlo znamená, že se obraz nezvětší najednou, ale postupuje se po 10ti procentech. Tím se sníží vliv velkých zvětšení. Samozřejmě je pro tuto variantu potřeba mnohem více času.

Bikubická interpolace je nejčastěji používanou metodou grafickými editačními softwary, ovladačem tiskáren, digitálními fotoaparáty apod.

Na Obrázek 4.5 a Obrázek 4.6 jsou vidět dvě metody bikubické interpolace. Vlevo je hladší varianta, hrany jsou zahlazeny a obraz vypadá přirozeně. Na pravé straně máme ostřejší variantu, která je spíše vhodná pro zmenšování. Hrany jsou ostré a přechody jsou zvýrazněny. Obě metody jsou ovšem viditelně přirozenější než metody v minulých kapitolách.

Bikubická interpolace se může provádět i na větším okolí bodu, například na 6x6 okolí.

Výpočet váhy zdrojového bod při výpočtu *sline 16* interpolace:

$$w = \left(\left(x - \frac{9.0}{5.0}\right) * x - \frac{1.0}{5.0}\right) * x + 1.0, \qquad 0 < x < 1$$

$$w = \left(\left(-\frac{1.0}{3.0} * (x-1) + \frac{4.0}{5.0}\right) * (x-1) - \frac{7.0}{15.0}\right) * (x-1), \qquad 1 < x < 2$$

(4.2)

Výpočet váhy zdrojového bod při výpočtu *sline 36* interpolace:

$$w = \left(\left(\frac{13.0}{11.0} * x - \frac{453.0}{209.0}\right) * x - \frac{3.0}{209.0}\right) * x + 1.0, \qquad 0 < x < 1$$

$$w = \left(\left(-\frac{6.0}{11.0} * (x-1) + \frac{270.0}{209.0}\right) * (x-1) - \frac{156.0}{209.0}\right) * (x-1), \qquad 1 < x < 2$$

$$w = \left(\left(\frac{1.0}{11.0} * (x-2) - \frac{45.0}{209.0}\right) * (x-2) + \frac{26.0}{209.0}\right) * (x-2), \qquad 2 < x < 3$$

(4.3)

Obrázek 4.5: Hladší bikubická interpolace

Obrázek 4.6: Ostřejší bikubická interpolace

Lagrangeova interpolace

Interpolace pomocí Lagrangeova polynomu patří k jednomu z nejstarších pokusu vést křivku libovolným počtem bodů. Aby bylo možno polynom vypočítat, je nutno znát souřadnice n bodů, jimiž má křivka procházet a dále musí být splněna podmínka, že posloupnost interpolovaných bodů musí být uspořádaná dle souřadnice x:

$$x_i < x_{i+1} \; pro \; i = 0..n$$

Interpolační polynom stupně n je potom definován vztahem:

$$L_n(x) = \sum_{i=0}^{n} y_i p_i(x)$$

kde polynom p_i(x) se vypočítá podle vzorce:

$$p_i(x) = \frac{(x - x_0) \dots (x - x_{i-1})(x - x_{i+1}) \dots (x - x_n)}{(x_i - x_0) \dots (x_i - x_{i-1})(x_i - x_{i+1}) \dots (x_i - x_n)} \; pro \; i = 0..n$$

$$p_i(x_j) = \delta_{ij} = \begin{cases} 1 \; pro \; i = j \\ 0 \; pro \; i \neq j \end{cases}$$

Kubická spline křivka

Spline křivka je křivka složená z polynomu stupně k, přičemž v opěrných bodech je zajištěna spojitost až do (k – 1) derivací. Mějme dáno (n + 1) opěrných bodů P_0,..., P_n a (n + 1) reálných parametrů u_0 <...< u_n. Interpolační kubická spline křivka P(u) je složena z kubických polynomů P_i (u), které musí splňovat tyto podmínky:

$P_i\,(u_i) = P_{i-1}\,(u_i)$
$P'_i\,(u_i) = P'_{i-1}\,(u_i)$
$P''_i\,(u_i) = P''_{i-1}\,(u_i)$, i = 1, …, n-1

Volbou čísel u_0,..., u_n. se určujeme parametrizaci spline křivky. Pro parametrizaci obvykle volíme u_i = i (Uniformní parametrizace).

Pro výpočet kubické spline křivky lze použít kubické polynomy ve tvaru:

P_i (u) = a_i + b_i (u - u_i) + c_i(u $-$ $u_i)^2$ + d_i(u $-$ $u_i)^3$, i = 0,...,n - 1

Tyto polynomy dosadíme do podmínek pro spline křivku a vyřešíme soustavu 4 × n rovnic.

Spline křivka je určena 4 × n koeficienty a_i , b_i , c_i a d_i. Ze zadání (n + 1) opěrných bodů a z podmínek pro kubický spline však dostáváme pouze (4×n - 2) podmínek. Aby byla soustava jednoznačně řešitelná, musíme ještě určit další dvě podmínky. Nejčastěji se volí jedna z následujících možností:

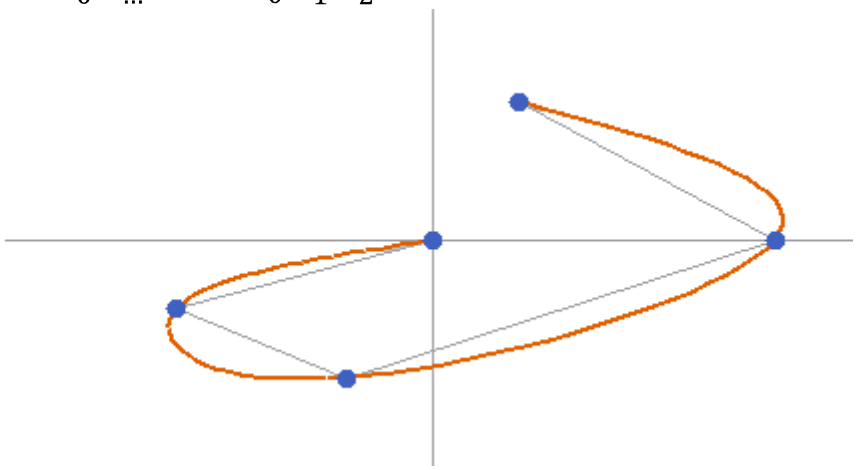
Vektory prvních derivací v krajních bodech P'_0 a P'_n.

Vektory druhých derivací v krajních bodech P''_0 a P''_n. Pokud použijete P''_0 = 0 a P''_n = 0 hovoříme o přirozené spline křivce. Křivka je uzavřená.

Při výpočtu kubické spline křivky se obvykle postupuje tak, že nejdříve vypočteme tečné vektory P'_0 ,…, P'_n. Jednotlivé kubiky P_i (u) pak můžeme vyjádřit pomocí Hermitovy interpolace.

Jako ukázku implementace jsme zvolili druhou možnost (přirozený kubický splajn). Z podmínek $P''(u_0)$ = 0 a $P''(u_n)$ = 0 dostaneme soustavu (n+1) rovnic pro výpočet tečných vektorů P'_0 ,…, P'_n. Zvolíme-li uniformní parametrizaci, pak platí:

$$\begin{bmatrix} 2 & 1 & 0 & & & \dots & 0 \\ 1 & 4 & 1 & 0 & & \dots & 0 \\ 0 & 1 & 4 & 1 & 0 & \dots & 0 \\ \vdots & & & & & & \vdots \\ 0 & \dots & & 0 & 1 & 4 & 1 \\ 0 & \dots & & & 0 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} P'_0 \\ P'_1 \\ \vdots \\ P'_n \end{bmatrix} = \begin{bmatrix} 3 \cdot (P_1 - P_0) \\ 3 \cdot (P_2 - P_0) \\ \vdots \\ 3 \cdot (P_n - P_{n-2}) \\ 3 \cdot (P_n - P_{n-1}) \end{bmatrix}$$



Obr. 2.1.3.2 Ukázka Kubické spline křivky

4.5 Sinc interpolace

Interpolace pomocí této metody je nejlepší s nejvěrohodnějšími výsledky ze všech dosud používaných metod. Ovšem pokud chceme dosáhnout tak dobrých výsledků, musíme ji provádět alespoň na bloku 16x16 okolí bodu, což je 256 bodů obrazu. Je proto jasné, že je tato metoda jednoznačně ze všech nejpomalejší.

Metoda se dá použít i na menších oblastech, jako je 6x6, 8x8, 10x10 a podobně, okolí bodů, ovšem sinc interpolace se používá především při snaze co možná nejvěrněji transformovat a následně interpolovat původní obraz a v takovém případě už se počítá s tím, že taková operace zabere poměrně hodně času. Při použití této interpolace například při jádru 8x8 dostaneme obdobné výsledky jako při použití metody bicubické a to při podstatně delším čase.

Sinc interpolace vychází z Lanczosovy funkce. Ta je pojmenována po maďarském matematikovi a fyzikovi Corneliu Lanczosovi. Svého rozšíření dosáhla zejména díky poměrně dobré kvalitě při převzorkováni a malému vzrůstu výpočetní složitosti ve srovnání s dříve používanými, obdobně složitými metodami.

Funkce je definovaná takto:

$$L(x) = sinc(x)sinc\left(\frac{x}{a}\right), \qquad -a < x < a, x \neq 0$$
$$L(x) = 1, \qquad x = 0$$
$$L(x) = 0, \qquad jinde$$

(4.4)

Proměnná „a“ udává velikost použitého jádra. U interpolace obrazových bodů se tedy obvykle jedná o hodnotu 8.

Funkce sinc(x) má pak tento tvar:

$$sinc(x) = \frac{\sin(\pi x)}{\pi x} \tag{4.5}$$

Váhy jednotlivých zdrojových bodů se vypočtou takto:

$$w = \left[\frac{\sin(\pi x)}{(\pi x)}\right] * \left[\frac{sin\left(\frac{\pi x}{a}\right)}{\left(\frac{\pi x}{a}\right)}\right] \tag{4.6}$$

5 Návrh řešení

Tato diplomová práce je vypracovávána společně s diplomovou prací kolegy Bc. Petra Němečka. Kolegova diplomová práce je zaměřena na geometrické transformace obrazu. Tato diplomová práce je pak zaměřena na interpolace obrazových bodů. Při běžném programování na procesoru, by se implementovaly funkce zvlášť s tím, že by se vytvořilo rozhraní pro předávání potřebných dat. Ovšem obě diplomové práce jsou zaměřeny na programování v paralelním programovacím modelu NVIDIA® CUDA™. Pro vytvoření maximální možné optimalizace a vytvoření aplikace, která je zaměřena především na rychlost výpočtu bylo potřeba funkce propojovat.

Hlavní části této diplomové práce je vytvoření sady interpolačních funkcí a jejich porovnání s metodami použitými v knihovně OpenCV. Proto bude vytvořena konzolová aplikace, která bude provádět jednotlivé geometrické transformace s různými interpolacemi. Tytéž transformace bude provádět i pomocí OpenCV. Aplikace bude schopna transformovat také video.

Aplikace bude mít několik bloků
- Zadání vstupních parametrů
- Inicializace CUDA
- Načtení obrázku nebo prvního snímku videa
- Podle zvolené transformace se vyžádají další parametry
- Alokace a naplnění potřebných polí
- Spuštění jádra
- Provádění jádra
- Kopírování dat zpět z paměti grafické karty
- Provedení transformace pomocí OpenCV
- Výstup buď pomocí vizualizace, nebo uložení do souboru
- Uvolnění veškerých použitých polí.

Zadání vstupních parametrů

Jelikož se bude jednat o konzolovou aplikaci, bude ovládána a spouštěna pomocí několika parametrů. Prvním parametrem bude soubor, který chceme transformovat. Nebude třeba nijak rozlišovat dalším parametrem, zda se jedná o obrázek nebo o video. O co se jedná, si už aplikace zjistí sama. Druhým parametrem bude typ zvolené transformace. Při špatně zadaném parametru pro transformace, se neprovede žádná transformace a obraz zůstane nezměněn. Třetí parametr pak bude udávat zvolenou interpolaci. Pokud je špatně zadán parametr interpolace, zvolí se automaticky bilineární interpolace. Čtvrtý, popřípadě pátý parametr bude sloužit k ovládání ostatních funkcí aplikace, to znamená, zda se má výsledek vypisovat do souboru, zda se má zobrazit po dokončení výpočtu a podobně. Přesněji je nastavení všech parametrů popsáno v dalších částech této práce.

Inicializace CUDA

Ihned se spustí inicializace CUDA. Jedná se o funkci, která pomocí funkcí CUDA nejprve ověří, zda systém obsahuje nějakou grafickou kartu. Pokud ano, vrátí tato funkce počet těchto grafických karet v systému. Poté se prochází všechny grafické karty a testuje se, zda podporují CUDA systém. První grafická karta, která splní tuto podmínku, bude vybrána a bude se používat při výpočtech. Pokud systém neobsahuje žádnou grafickou kartu, je aplikace ukončena, stejně tak pokud žádná grafická karta nepodporuje systém CUDA.

Tato funkce nijak neporovnává výkonnost grafických karet, pokud by bylo více nalezených podporujících CUDA systém. Jednoduše se vybere první dostupná. Na většině běžných počítačů se setkáme pouze s jedinou grafickou kartou, popřípadě se dvěma, které jsou ovšem většinou stejně výpočetně silné.

Při emulaci je tato funkce upravena tak, že vždy vrací kladný výsledek. Emulační mód bude vysvětlen dále.

Načtení obrázku nebo prvního snímku videa

Po dokončení inicializace CUDA dojde k pokusu načíst první snímek videa. Pokud tato funkce nevrátí počet snímku, znamená to, že soubor buďto není video souborem, nebo je tento soubor poškozen. V takovém případě se aplikace pokusí načíst obrázek. Tato funkce je ošetřena tak, že při nezdaru, ukončí celou aplikaci s chybou. Veškeré tyto funkce pro obrázek či video budou použity z knihovny OpenCV.

Další parametry vstupu

Bude se jednat o doplňující parametry, které se budou lišit podle zadané transformace. Například pro rotaci nám bude stačit vyžádat si od uživatele úhel, o který se má obraz pootočit. U transformace zvětšení nebo zmenšení, potřebujeme dva parametry. A to parametry pro zvětšení/zmenšení v ose X a totéž v ose Y.

Aplikace bude také umožňovat skládání jednotlivých transformací. Při spuštění v tomto režimu bude od uživatele vyžadovat zadání počtu transformací, ze kterých se bude výsledná transformace skládat. Jelikož je každá transformace tvořena pomocí transformační matice, i tato složená transformace je tvořena jedinou maticí. Ta vznikne násobením matic jednotlivých transformací. Každá přidaná matice bude násobit tu původní zprava. Při zadávání jednotlivých transformací budou opět postupně vyžadovány potřebné parametry, jako je například úhel otočení pro rotaci.

Po zpracování všech parametrů ještě dojde k vypočtení velikosti výsledného obrazu, který se zvětšuje u transformace zvětšení/ zmenšení.

Alokace potřebných polí

První pole, které potřebujeme vytvořit je pole, pro výstupní data z GPU v globální paměti grafické karty. Toto pole předáváme grafické kartě prázdné. Pole má velikost podle výšky a šířky výsledného obrazu vynásobeno velikostí tří proměnných typu `unsigned char`, a to z důvodu tří barevných složek každého bodu. To znamená, že velikost tohoto pole bude, pro obrázek o velikostech 1280x1024, 4MB. Toto pole je předáváno jádru při spuštění. Při výpočtu je každý prvek pole použit právě jednou a to při zápisu hodnoty jednotlivé složky barvy výsledného bodu. Při tak malém počtu přístupu k poli nevadí, že je pole v globální paměti. Samotná alokace tohoto pole je provedena funkcí `cudaMalloc()`, která má dva parametry. Prvním je ukazatel na data, který bude používán dále v programu a druhým je velikost alokované lineární paměti.

Druhým polem, které budeme potřebovat na grafické kartě je pole v texturovací paměti. Toto pole je vytvořeno naprosto odlišným způsobem, než předchozí pole. Navíc pole bude naplněno daty původního obrazu. Jeho velikost je dána obdobně jako minulé pole. Ovšem rozměry jsou použity ze zdrojového obrazu. Pro výpočet velikosti platí to co v minulém poli, jelikož i toto pole obsahuje pro každý bod, tři proměnné typu `unsigned char`. Pole je v texturovací paměti protože se jeho data budou používat pouze pro čtení a to často tak, že požadovaný prvek pole bude v blízkosti minulého prvku. Jak již bylo uvedeno v teorii, je pro právě toto čtení texturovací paměť optimalizována. Pole je vytvořeno jako dvourozměrné pole pomocí funkce `cudaMallocArray()`, které se předávají tyto parametry: ukazatel na vytvořené pole, popis tohoto pole, výška a šířka. Popis je dán pak speciálním

datovým typem `cudaChannelFormatKind`, který určuje počet bitů, které náleží jednomu prvku a formát v jakém má vracet data přečtená z tohoto pole. Formát může být typu `float`, `unsignet` nebo `signed`. Jelikož my potřebujeme číst z tohoto pole datové typy `unsigned char`, nastavíme počet čtených bitů na 8 a formát nastaven na čtení `unsigned` hodnot. Šířku pole vypočteme vynásobením šířky vstupního obrazu třemi. Výška zůstane stejná.

Po vytvoření pole je třeba přiřadit pole textuře. To je provedeno pomocí funkce `cudaBindTextureToArray()`. Po tomto vytvoření textury ji naplníme daty a to pomocí další CUDA funkce `cudaMemcpyToArray()`, které se předává cílové pole, zdrojové pole, počet přenášených bajtů a typ přenosu. Typy jsou čtyři a určují, odkud a kam se budou data přenášet. Pro naplnění toho pole použijeme parametr `cudaMemcpyHostToDevice`.

Další pole bude sloužit jako úložiště pro data, kopírována z grafické karty. To znamená, že bude mít stejnou velikost jako pole v globální paměti grafické karty. Pole bude v operační paměti a bude přístupné pouze z programu prováděného na procesoru. K alokaci ovšem opět použijeme funkci CUDA. Jedná se o funkci `cudaHostAlloc()`, která alokuje pole v nestránkovací paměti, na rozdíl od běžné alokace. Tímto je dosaženo výrazných rozdílů při kopírování dat mezi paměti na grafické kartě a operační paměti.

Spuštění jádra

Před samotným spuštěním jádra, je třeba dvě proměnné, typu dim3. Což je datový typ, který je založen na trojrozměrném datovém typu. Je používána ke specifikování dimenzí. Každá proměnná, která není zadána, je nastavena implicitně na hodnotu 1.

První vytvoříme proměnnou, která bude specifikovat počet spouštěných vláken v jednom bloku. Zvolili jsme 16x16 vláken, čímž dostaneme 256 vláken na jeden blok vláken. Těmito bloky se pak musí pokrýt celý výsledný obraz. To uděláme pomocí další proměnné, kterou specifikujeme počet spouštěných bloků.

Výpočet počtu těchto bloků je uveden na rovnici následujících rovnicích:

```
x = ((šířka_obrazu + šířka_bloku_vláken )-1) / šířka_bloku_vláken
y = ((výška_obrazu + výška_bloku_vláken )-1) / výška_bloku_vláken
```
(5.1)

Jádru předáváme dva parametry. Odkaz na pole alokované na grafické kartě a strukturu, která obsahuje veškeré potřebné parametry transformací. Struktura uchovává informace o rozměrech vstupního obrazu, výstupního obrazu, typ a parametry transformace a zvolenou interpolační metodu.

Provádění jádra

První část jádra je napsána Bc. Petrem Němečkem a zajišťuje pro každý bod výsledného obrazu vypočet souřadnic v původním obrazu. Poté je volána funkce interpolace. Této funkci se předává ukazatel na výsledné pole, souřadnice počítaného bodu, souřadnice původního bodu a struktura s parametry.

Samotná funkce interpolace pak nejprve přičte k oběma přepočteným souřadnicím hodnotu 0.5, čímž se zarovnání bodu změní z -0,5 až 0,5 na 0 až 1. Následně musí oddělit celou a desetinou část přepočtených souřadnic. Interpolace bude vyplňovat výstupní obraz černými body všude tam, kde jsme se po transformaci dostali mimo meze původního obrazu. Pokud je bod v mezích původního obrazu, je zavolána příslušná funkce pro všechny tři barevné složky obrazu.

Interpolace nejbližší soused

Použije se pouze celá část souřadnice a jednoduše se načte hodnota z textury.

Bilineární interpolace

Tato interpolace je jednoduchá a nepotřebuje pro svoje výpočty žádnou speciální paměť. Veškeré výpočty jsou dočasně uloženy v registrech. Interpolace je počítána na 2x2okolí bodu. Nejprve je proveden výpočet prvního a druhého řádku, následně je tato interpolace provedena na výsledcích obou řádků. Výpočet váhy uveden v rovnici (4.2).

Bikubická interpolace

U této metody už nevystačíme pouze s ukládáním do registrů, respektive vystačíme, ale z důvodů optimalizace zde použijeme pole uložené ve sdílené paměti. Tím zmenšíme počet potřebných registrů. Blíže si tuto problematiku přiblížíme v implementaci.

Zvolená bikubická interpolace je počítaná na 4x4 okolí bodu. U této interpolace se nejprve vypočtou váhy bodů ve vodorovné rovině. Pomocí těchto vah následně vypočteme hodnoty všech řádku, které pak vynásobíme váhami bodů ve svislém směru. Vzorec použitý při interpolace je uveden v rovnici (4.2).

Sinc interpolace

Stejně jako u minulé metody by bylo vhodné použít k práci s daty vyrovnávací paměť. Ovšem s náročností této metody také roste paměťová náročnost, a jelikož je sdílená paměť omezena, není možné tuto optimalizaci jednoduše použít. Více bude tato problematika vysvětlena v implementaci.

Sinc interpolace bude prováděna na 16x16 okolí bodu, přičemž stejně jako v minulých metodách se nejprve vypočtou váhy vodorovných bodů, následně se vypočte hodnota daného řádku. Poté se provede výpočet vah ve svislém směru a vypočte se výsledná hodnota násobením vah s příslušnými řádky. Vzorec této interpolace je uveden v rovnici (4.6).

Kopírování dat zpět z paměti grafické karty

Paměť je kopírována z globální paměti grafické karty do pole alokovaného v nestránkované operační paměti, pomocí funkce `cudaMemcpy()`, poslední parametr je třeba nastavit na kopírování z grafické karty, to znamená, že použijeme parametr `cudaMemcpyDeviceToHost`. Rozdíl mezi kopírováním do stránkované nebo nestránkované paměti je až dvojnásobný.

Provedení transformace pomocí OpenCV

Pro porovnání jak vizuálního výstupu, tak časové náročnosti se provede stejná geometrická transformace s příslušnou interpolací, která byla vypočtena pomocí CUDA.

Výstup buď pomocí vizualizace, nebo uložení do souboru

Podle počátečních parametrů se buďto nemusí zobrazovat nic, nebo se může zobrazit vizuální výstup. Zobrazení je opět provedeno pomocí funkcí OpenCV. Aplikace zobrazí tři okna. V prvním bude zdrojový obraz, ve druhém bude zobrazen obraz transformovaný pomocí CUDA a ve třetím pak obraz transformovaný pomocí funkcí OpenCV.

Pokud je navíc zadán parametr pro uložení, výsledný soubor se uloží. Pro ukládání byl zvolen formát PNG, který podporuje bezztrátovou kompresi. Při použití formátu JPG by došlo ke zkreslení výsledku. Při použití formátu BMP, by naopak soubor zabíral příliš mnoho místa. Soubor bude uložen do stejné složky, ve které bude spuštěna aplikace. Název souboru se skládá z názvu původního obrazu, zvolené transformace a zvolené interpolace.

Uvolnění veškerých použitých polí.

Nakonec je třeba uvolnit veškerá pole, speciálně pak pole vytvořena na grafické kartě. Totéž platí o obrázcích uložených ve strukturách OpenCV. Jelikož budeme pracovat i s většími obrázky, zabereme poměrně dost paměti. Výpočet paměťové náročnosti je rozebrán v části implementace.

5.1 Přehrávání videa

Rozdíl v programu je pouze v části před zobrazením výsledků. Pokud je načteno video, program se po zpracování a zobrazení výsledku neukončí, ale pokračuje ve smyčce tak dlouho, dokud nenarazí na poslední snímek. Ještě před touto smyčkou jsou vytvořeny trackbary použité k pohybu videa po časové ose, k výběru interpolace a při rotaci k výběru úhlu rotace. V samotné smyčce je pak funkce na načítání snímků, následovaná stejnou pasáží, která je potřebná pro zpracování obrazu. To znamená nakopírování dat ze zdrojového obrazu do texturovací paměti, spuštění jádra, vyčkání dokončení provádění jádra a nakopírování dat zpět z grafické paměti. Poté je třeba každý snímek zobrazit. Tato smyčka se opakuje, dokud uživatel nestiskne klávesu Escape nebo se neprovede poslední snímek.

6 Implementace

6.1 OpenCV

OpenCV = Open Source Computer Vision
Jedná se o knihovnu pro počítačové vidění, původně vyvinutou firmou Intel. Knihovna je multiplatformí, a běží na Windows, Mac OS X, Linux, PSP, VCRT (Real-Time OS pro Smart kamery) apod. Zaměřuje se hlavně na real-time zpracování obrazu, jako takové. Dokonce je schopna využít integrované výkonové primitiva od firmy Intel, pokud jsou nainstalovány v operačním systému. OpenCV je vydáno v souladu s podmínkami BSD licencí, jedná se o software s otevřeným zdrojovým kódem.

V této práci je OpenCV používáno pro načítání obrázku a videa, pro jejich zobrazení a ukládání. Dále je použito jako referenční implementace grafických transformací a interpolací obrazových bodů. Jelikož je OpenCV velmi dobře optimalizováno je to dobrý příklad k porovnání výpočtu optimalizovaného kódu prováděného na CPU a paralelního kódu, který není plně optimalizován, ale je prováděn na GPU.

OpenCV podporuje většinu běžných obrázkových formátů. Například BMP, JPEG, JPG, JPE, PNG, PBM, PGM, PPM, TIF. Z video formátů je to pouze AVI. Video zpracovávané pomocí této knihovny je rozloženo na jednotlivé obrázky.

Zdroj tohoto textu je uveden v [12] a [16].

6.2 Problém optimalizace

Při programování v CUDA je sice spousta věcí řízena automaticky. Programátor se nemusí starat o zprávu vláken a podobně. To znamená, že může naprogramovat jednoduše alokaci polí, přesuny dat, spuštění jádra, dokonce i samotnou funkci jádra. Ovšem pokud se nepodívá na použité prostředky a dobře si nevypočítá, co nezbytně nepotřebuje a neodstraní vše nadbytečné, může výkon jednotlivých multiprocesorů nevědomky využít jen na několik procent. Záleží také hodně na použitých matematických funkcích. V dalších podkapitole si přiblížíme problém, který jsem řešil při optimalizaci.

Výpočetní schopnost

V současné době existují 4 rozdělení výpočetních schopností grafických karet podporujících CUDA systém. Je to 1.0, 1.1, 1.2 a 1.3. Co se týče fyzických omezení, jsou 1.0 a 1.1 na tom úplně stejně, podobně pak 1.2 a 1.3. Dále budeme používat proto pouze označení 1.1 a 1.3, které budou reprezentovat vždy svou skupinu. V Tabulka 6-1 jsou uvedeny hlavní parametry fyzických omezení obou skupin.

Výpočetní schopnost	1.0 a 1.1	1.2 a 1.3
Vláken na warp	32	32
warpů na multiprocesor	24	32
Vláken na multiprocesor	768	1024
Bloků vláken na multiprocesor	8	8
Sdílená paměť na multiprocesor	16384MB	16384MB
Počet registrů	8192	16384
Velikost registrační alokační jednotky	256	512

Tabulka 6-1: Porovnání výpočetních schopností grafických karet podporujících CUDA systém

Při programování ovlivňuje programátor tři zásadní věci: počet vláken jednoho bloku, počet registrů potřebných pro výpočet jediného vlákna a sdílenou paměť jednoho bloku. Problém v podstatě spočívá v tom, že na jednom multiprocesoru může běžet několik bloků vláken současně. Ovšem pouze v případě, že je dostatek volných prostředků. Pokud ne, dochází k menšímu vytížení multiprocesoru.

Například pokud používáme grafickou kartu s výpočetní schopností 1.1, spustíme jádro s 256ti vlákny, můžeme dosáhnout spuštění až tří bloků vláken na jednom multiprocesoru, ale ovšem jen v případě, že počet registrů na jedno vlákno nepřesáhne hodnotu 10 a sdílená paměť jednoho bloku nebude větší než 5120MB. Pokud splníme všechny tři podmínky, využijeme multiprocesor na sto procent. Ovšem při překročení sdílené paměti byť jen o jediný bajt, dojde k nedostatku paměti pro tři bloky a budou se moct spustit pouze bloky dva, čímž dostaneme vytížení multiprocesoru pouze 66,67 procent. Obdobně totéž platí pro registry. Při překročení poloviny použitelných prostředků buď sdílené paměti, registrů nebo počtu vláken, spustí se pouze jediný blok, čímž se vytíženost sníží na 33,33 procent.

V našem případě se jedná především o optimalizaci interpolací, proto si jednotlivé parametry všech čtyř metod vypíšeme. Všechny interpolace jsou spouštěny s bloky o velikosti 16x16, takže používáme 256 vláken.

Interpolace nejbližší soused

Tato interpolace nepotřebuje žádné výpočty přímo, ale jelikož jediné vlákno provádí i výpočet transformací, je počet použitých registrů na jedno vlákno 9. Používáme zde minimální množství sdílené paměti, přesněji pak 216 bajtů. Ani v jedné podmínce nepřekročíme limit, proto se spustí tři bloky na každém multiprocesoru při výpočetní schopnosti 1.1, při verzi 1.3 pak čtyři bloky.

Bilineární interpolace

U této interpolace již provádíme několik výpočtů, proto nám vzroste počet použitých registrů na hodnotu 16. Velikost použité sdílené paměti je shodné, to znamená 216 bajtů. Při výpočetní schopnosti 1.1 překročíme maximální počet registrů pro stoprocentní využití multiprocesoru, takže se spustí právě dva bloky. U výpočetní schopnosti 1.3 je počet registrů větší a umožní nám spustit všechny čtyři dostupné bloky.

Bikubická interpolace

Tato interpolace je opět složitější na výpočet a používá také větší sdílenou paměť. Využívá 20 registrů na jedno vlákno, a 4336 bajtů sdílené paměti. U grafických karet s výpočetní schopností 1.1 to znamená z důvodu velkého počtu použitých registrů snížení počtu použitých bloků na jeden ze tří.

To znamená pouze 33.3 procent využití multiprocesoru. Nedojde ovšem až k tak velkému nárůstu registrů, aby se to promítlo i ve verzi 1.3. u této verze budou spuštěny opět tři bloky.

Sinc interpolace

Při této interpolaci bylo nutné nejvíce testovat a zkoušet různé přístupy. První přístup nechal většinu výpočtů v registrech, při použití 41 registrů se samozřejmě mohl pustit jediný blok. Dalším pokusem bylo maximální využití sdílené paměti, ovšem ani při použití téměř poloviny sdílené paměti neklesl počet registrů dostatečně. Poslední možnost, ta která vyšla časově nejlépe je využití lokální paměti. Tím dojde ke zpoždění čtení a zápisu dat používaných při výpočtech, ovšem na jednom multiprocesoru poběží více než jeden blok. Teda pouze na grafických kartách s výpočetním výkonem 1.3. V systémech s nižším výpočetním výkonem poběží pouze jeden blok na mikroprocesor.

Konkrétně je využito 18 registrů na vlákno a 216 bajtů sdílené paměti na blok vláken. Ovšem při použití 196 bajtů lokální paměti.

Programátor si může na stránkách vývojářů CUDA stáhnout excel soubor, který pomáhá vypočítávat obsazenost multiprocesorů. Soubor se jmenuje „CUDA Occupancy calculator“. V souboru programátor pouze vyplní použité prostředky svého programu a v grafech nebo pomocí vypočtených hodnot zjistí, na kolik procent využívá grafické multiprocesory. Použité prostředky může zjistit programátor při překladu přidáním parametru „--ptxas-options=-v“.

6.3 Paměťová náročnost

Program je paměťově stejně náročný jak na straně operační paměti, tak na straně paměti na grafické kartě. Jelikož je operační paměť ve většině případů větší než paměť na grafické kartě, budeme se dále zabírat především místem alokovaným na grafické kartě. Tabulka 6-2 uvádí příklady rozlišení běžných obrázků, jejich počet bodů obrazu a paměť potřebnou k alokaci na GPU.

Rozlišení obrazu	Počet bodů obrazu	Alokovaná paměť na grafické kartě
100 x 100	10,000	60,00KB
640 x 480	307,200	1,80MB
800 x 600	480,000	2,88MB
1024 x 768	786,432	4,72MB
1280 x 1024	1,310,720	7,68MB
1600 x 1200	1,920,000	11,52MB
2272 x 1704	3,871,488	23,23MB
2816 x 2112	5,947,392	35,68MB
3648 x 2736	9,980,928	59,89MB
5000 x 3000	15,000,000	90.00MB
10000 x 7500	75,000,000	450.00MB

Tabulka 6-2: Paměťová náročnost programu

Tabulka 6-3 uvádí maximální rozlišení obrázku pro jednotlivé velikosti paměti. Hodnoty jsou pouze orientační. Záleží na aktuálním využití grafické paměti

Velikost paměti grafické karty	Maximální rozlišení obrazu
128MB	5000 x 4000
256MB	7500 x 5000
384MB	10000 x 6000
512MB	11000 x 8000
896MB	14000 x 11000
1024MB	15000 x 11000
1792MB	20000 x 15000
2048MB	22000 x 15000

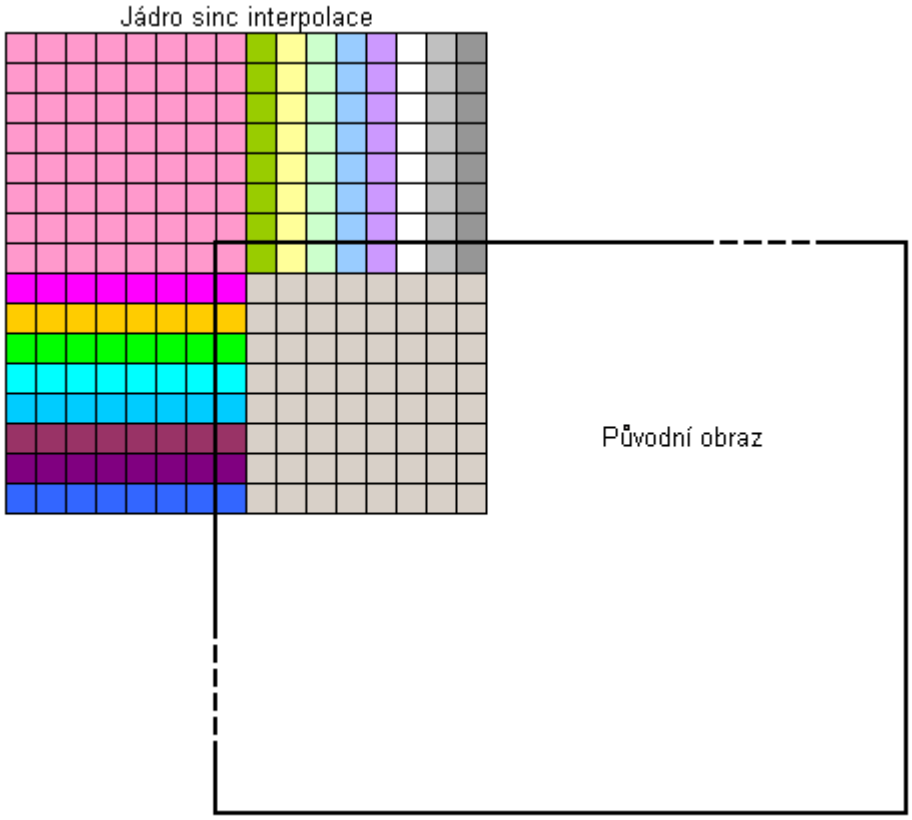
Tabulka 6-3: Maximální rozměr obrázků pro jednotlivé velikosti pamětí grafických karet

6.4 Přístup mimo rozsah textury

Jak již bylo popsáno dříve, CUDA nabízí velmi dobrou podporu textur. Proto není třeba ošetřovat přístup mimo meze textury. Při přístupu mimo meze textury, to znamená při hodnotách záporných, nebo větších než je velikost vytvořené textury, je hodnota vrácena do platných hodnot. To znamená, že pro souřadnice (-1,0) CUDA vrátí hodnotu nacházející se na souřadnicích (0,0).

V programu přistupujeme mimo meze textury u všech interpolačních metod s výjimkou metody nejbližší soused. Například při výpočtu interpolace sinc, se hned při výpočtu bodu, který měl souřadnice v původním obrazu (0,0) se většina jádra vůbec nevleze do obrazu. Všechny tyto hodnoty budou rovny hodnotě bodu na souřadnicích právě (0,0).

Obrázek 6.1: Přes jádra interpolace sinc přes okraj původního obrazunázorně tuto situaci znázorňuje, body stejné barvy budou mít stejnou hodnotu. Samozřejmě krom hodnot, které se nachází v obraze na souřadnicích minimálně bod od kraje, ty sice mají všechny stejnou barvu, nicméně při výpočtu se bude brát jejích hodnota. Například hodnota bodu na souřadnicích (0,0) se při výpočtu použije hned 64krát.



Obrázek 6.1: Přes jádra interpolace sinc přes okraj původního obrazu

6.5 Mapování pamětí

CUDA systém nabízí funkce k mapování grafické paměti na paměť operační. Vytvoření je provedeno pomocí dvou funkcí CUDA. První se vytvoří alokace pole v operační paměti pomocí funkce `cudaHostAlloc()` se speciálním parametrem `cudaHostAllocMapped`. Následně je třeba získat ukazatel na paměť alokovanou v grafické paměti. To provedeme pomocí funkce `cudaHostGetDevicePointer()`.

Takto vytvořená paměť nahradí dvě paměti – jednu pro hlavní program a druhou pro jádro. Synchronizace mezi oběma paměťovými prostory je automatická, takže se výsledky počítané na grafické kartě průběžně přenášejí do operační paměti. Není proto potřeba provádět přenosy dat před spuštěním nebo po dokončení jádra.

Tuto variantu jsme testovali, ovšem výsledky byly několika násobně pomalejší než při použití dvou paměti a kopírováním dat pomocí kopírovacích funkcí systému CUDA. Proto se ve výsledném programu tato varianta neobjevuje.

7 Testování a porovnání interpolací

V této kapitole budeme testovat rychlost a kvalitu jednotlivých interpolací na různých počítačových sestavách a hlavně na různých grafických kartách. Pro testování jsme vybrali několik dostupných počítačových konfigurací s ohledem především na výkon grafické karty, tak abychom měli zastoupení jedné z nejpomalejších, minimálně dvou středních pro porovnání závislosti na ostatních komponentech, a jednu grafickou kartu z nejvýkonnější kategorie.

Použité grafické karty

- Slabě výkonná grafická karta s čipem – GeForce 8400M GS s pamětí 128 MB
- Středně výkonná grafická karta s čipem- GeForce 9800GT s pamětí 512MB
- Referenční středně silná grafická karta s čipem – GeForce 9800GTX s pamětí 512MB
- Vysoce výkonná grafická karta s čipem - GeForce GTX 280 s pamětí 1024MB

	Nejméně výkonná GK	**Středně výkonná GK**	**Referenční středně výkonná GK**	**Nejvýkonnější GK**
CPU:	Intel Mobile Core 2 Duo T8100	AMD Athlon(tm) 64 X2 Dual Core Processor 3800+	Intel Core2 Duo E8200	Intel Core 2 Duo E6550
Takt procesoru:	2,10 GHz	2 GHz	2,66 GHz	2,30 GHz
L1 cache:	2 × 32 kB	2 × 64 kB	2 x 64 kB	2 × 32 kB
L2 cahce:	3072 kB	2 × 512 kB	6 MB	4096 kB
Paměť:	DDR2 3072 MB, 333 MHz	DDR2 2048 MB, 400 MHz	DDR2 4096 MB, 800MHz	DDR2 3072 MB, 400 MHz
GPU:	GeForce 8400M GS	GeForce 9800 GT	GeForce 9800 GTX	GeForce GTX 280
Výpočetní schopnost CUDA:	1	1,1	1,1	1,3
Velikost globální paměti v kB:	131072	523968	523968	1048320
Počet multiprocesorů:	2	16	16	30
Počet jader:	16	128	128	240
Velikost konstantní paměti v bajtech:	65536	65536	65536	65536
Velikost sdílené paměti:	16384B	16384B	16384B	16384B
Počet registrů na 1 blok:	8192	8192	8192	16384
Takt	0.80 GHz	1.62 GHz	1.62 GHz	1.30 GHz
Integrován:	Ne	Ne	Ne	Ne
Podpora mapování:	Ne	Ano	Ano	Ano

Tabulka 7-1: Parametry testovaných sestav

7.1 Obrazové výsledky interpolací

Zde si ukážeme stejné obrázky zvětšené pomocí interpolací implementovaných na systému CUDA. Použijeme stejný obraz jako v kapitole o Interpolaceích, i stejné zvětšení, to znamená trojnásobek v obou osách.

Výsledky jsou podle očekávání. Všechny metody jasně deklarují své vlastnosti. Metoda nejbližšího souseda na Obrázek 7.2 ukazuje při zvětšení vytváření shluku bodů 3x3 stejné barvy. Metoda lineární na Obrázek 7.3 zjemní aliasing hran, ovšem obraz značně rozostří. Další dvě metody mají podobné výsledky, ale na Obrázek 7.4 interpolace sinc můžeme vidět dokonalejší zachování plynulostí šikmých hran než u interpolace bikubické na Obrázek 7.5.

Obrázek 7.1: Původní obrázek

Obrázek 7.2: Obrázek zvětšený pomocí metody nejbližší soused v systému CUDA

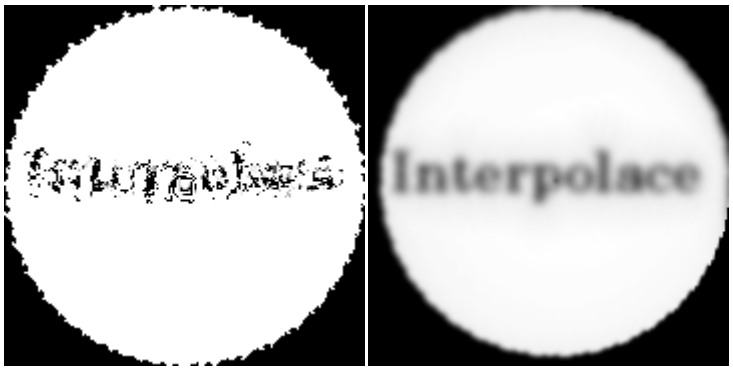
Obrázek 7.3: Obrázek zvětšený pomocí bilineární interpolace v systému CUDA

Obrázek 7.4: Obrázek zvětšený pomocí bikubické interpolace v systému CUDA

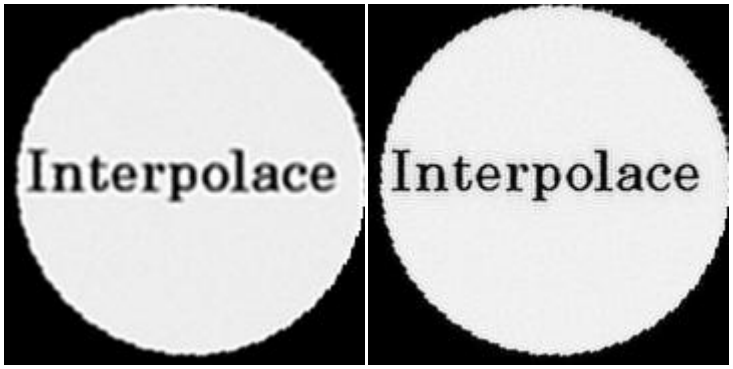
Obrázek 7.5: Obrázek zvětšený pomocí sinc interpolace v systému CUDA

Další test zobrazuje 36 rotací jednoduchého obrázku, na kterém si ukážeme zachování přesností jednotlivých interpolací. Metoda nejbližší soused je pro takovouto transformaci naprosto nepoužitelná, jak je patrné na Obrázek 7.7. Lineární interpolace na tomtéž obrázku, obraz velmi znatelně rozostří, ale dá se říct, že nápis je již alespoň čitelný. Při bikubické interpolaci je nápis pěkně čitelný, ale stále lehce rozmazaný. Interpolace sinc má podle předpokladů nejlepší výsledek, ten je ovšem kompenzován mnohem větší časovou náročností.

Obrázek 7.6: Vzor testu 2



Obrázek 7.7: Vlevo metoda nejbližší soused, vpravo lineární interpolace



Obrázek 7.8: Vlevo bikubická a vpravo sinc interpolace

7.2 Časové výsledky interpolací

Tabulka 7-2 reprezentuje naměřené časy na použitých grafických kartách. Test byl proveden pomocí geometrické transformace rotace. Tato rotace byla provedena 35x. Naměřené časy jsou v milisekundách. U nejméně výkonné grafické se posledních pět rozlišení při interpolaci sinc nepodařilo naměřit.

Rozlišení obrazu	Interpolace	Nejméně výkonná GK	Středně výkonná GK	Referenční středně výkonná GK	Nejvýkonnější GK
10 x 8	Nejbližší soused	9,055	3,032	7,778	1,601
640 x 480	Nejbližší s.	204,661	49,265	53,222	24,269
1024 x 768	Nejbližší s.	514,528	124,198	136,371	59,555
1600 x 1200	Nejbližší s.	1228,691	267,233	285,301	94,840
2247 x 1704	Nejbližší s.	2534,553	504,509	534,902	285,902
2816 x 2112	Nejbližší s.	4062,875	828,269	878,291	437,659
3648 x 2048	Nejbližší s.	4749,078	982,045	1059,406	550,309
3648 x 2736	Nejbližší s.	6337,453	1313,217	1400,536	734,604
10 x 8	Bilineární	9,242	1,310	7,711	1,675
640 x 480	Bilineární	211,308	43,894	50,845	25,926
1024 x 768	Bilineární	514,171	126,219	144,077	63,902

1600 x 1200	Bilineární	1236,612	251,087	274,939	101,879
2247 x 1704	Bilineární	2486,778	505,013	554,529	306,081
2816 x 2112	Bilineární	3797,164	796,644	860,968	468,394
3648 x 2048	Bilineární	4755,846	973,377	1039,362	588,163
3648 x 2736	Bilineární	6347,937	1297,782	1395,893	785,502
10 x 8	Bikubická	11,636	2,404	10,423	2,682
640 x 480	Bikubická	1639,302	125,683	141,739	57,921
1024 x 768	Bikubická	4142,009	337,457	374,859	148,667
1600 x 1200	Bikubická	10040,750	727,060	822,847	239,413
2247 x 1704	Bikubická	20206,082	1473,210	1645,335	700,127
2816 x 2112	Bikubická	30941,561	2274,135	2536,977	1068,575
3648 x 2048	Bikubická	38416,188	2788,455	3112,214	1326,895
3648 x 2736	Bikubická	42182,825	3753,024	4173,559	1785,737
10 x 8	Sinc	42,363	18,089	20,768	25,716
640 x 480	Sinc	19822,564	1644,006	1795,234	986,650
1024 x 768	Sinc	49846,859	4188,938	4516,176	2482,910
1600 x 1200	Sinc		9892,437	10740,903	4622,285
2247 x 1704	Sinc		19898,764	21768,356	11922,091
2816 x 2112	Sinc		30475,588	34897,235	18136,922
3648 x 2048	Sinc		37563,910	42489,564	22784,428
3648 x 2736	Sinc		50890,168	56001,101	30608,355

Tabulka 7-2: Naměřené hodnoty

Tabulka 7-3 znázorňuje zrychlení na jednotlivých grafických kartách. Jako referenční je použita nejpomalejší grafická karta. Ovšem je třeba upozornit na opačný výsledek, co se týče středně výkonných grafických karet, než se očekávalo. Referenční grafická karta má podle Tabulka 7-1 stejné parametry jako středně výkonná grafická karta, ovšem s rychlejším procesorem a pamětmi. Díky tomu by měla dosahovat lepších výsledků.

Interpolace	**Nejméně výkonná GK**	**Referenční středně výkonná GK**	**Středně výkonná GK**	**Nejvýkonnější GK**
Nejbližší soused	1	3,520	4,434	8,899
Bilineární	1	2,964	5,042	8,554
Bikubická	1	7,916	12,038	25,293
Sinc	1	7,902	8,766	14,089

Tabulka 7-3: Srovnání výkonů testovaných grafických karet

Tabulka 7-4 ukazuje počet zpracovaných bodů za jednu milisekundu jednotlivými grafickými kartami při všech interpolacích. Zpracování bodu zahrnuje jak kopírování dat na GPU a zpět. Samotné provedení výpočtu interpolační funkce by bylo rychlejší, ovšem bez kopírování dat by nešlo provést, popřípadě prezentovat. Na výsledcích je vidět, že jednoduchý výpočet, který využívá bilineární interpolace je grafický čip schopen zvládnout stejně rychle jako pouhý přístup k hodnotě v textuře, používaný metodou nejbližší soused. U dalších metod už počet zpracovaných bodů za jednu milisekundu značně klesá.

Interpolace	Nejméně výkonná GK	Referenční středně výkonná GK	Středně výkonná GK	Nejvýkonnější GK
Nejbližší soused	1530	7060	6574	14010
Bilineární	1540	7289	6601	13088
Bikubická	196	2561	2289	5731
Sinc	16	193	175	332

Tabulka 7-4: Počet zpracovaných bodů za jednu milisekundu

7.3 Výsledky testování videa

Video bylo testováno pouze na středně silné grafické kartě a příslušné sestavě, jejíž parametry jsou v Tabulka 7-1. Zpracování videa probíhá obdobně jako opakované zpracování obrázku. Stejně tak zpracování jednoho snímku zabere stejný čas, co se týče operací grafické karty. Ovšem při přehrávání videa je třeba používat dvě funkce, které se při opakovaném pouštění jádra neprováděli. A to načítání jednotlivých snímků a jejich zobrazování.

Časová náročnost funkce na načtení jednoho snímku je při videu s rozlišením 1024x764 okolo 20ti milisekund. Zpracování jádra je při takovém rozlišení 5ms pro metodu nejbližší soused a lineární interpolaci, pro bikubickou interpolaci je to pak okolo 12ti milisekund a pro interpolaci sinc je to okolo 120ti milisekund. Časová náročnost vykreslení jednoho obrazu při takovém rozlišení je necelých 50ms.

Rozlišení videa	Interpolace	Počet snímků za vteřinu
720 x 576	Nejbližší soused	27
1280 x 720	Nejbližší soused	13
720 x 576	Bilineární	27
1280 x 720	Bilineární	13
720 x 576	Bikubická	21
1280 x 720	Bikubická	11
720 x 576	Sinc	10
1280 x 720	Sinc	5

Tabulka 7-5: Testování videa

Z výše uvedených hodnot plyne, že při nižším rozlišení je aplikace schopna přehrávat plynule video pouze nejbližší soused a bilineární. Při vyšším rozlišení se ani nepřiblížíme přehrávání videa v reálném čase.

Zpomalení je způsobeno z větší části nutností zobrazovat jednotlivé snímky. Aplikace by byla schopna v reálném čase zpracovávat mnohem větší rozlišení, pokud by výsledný obraz nemusela zobrazovat.

8 Závěr

Tato práce se zabývá interpolacemi obrazových bodů. Je vypracovávána společně s diplomovou prací kolegy Bc. Petra Němečka, která je zaměřena na geometrické transformace obrazu. Úkolem bylo nastudovat a následně implementovat interpolační metody v systému NVIDIA® CUDA™, což je paralelní výpočetní architektura grafických karet. Byly vytvořeny funkce pro čtyři základní interpolace. Interpolaci nejbližší soused, lineární interpolaci, bikubickou interpolaci a interpolaci sinc. Všechny tyto metody jsou podrobně popsány v kapitole 4.

Grafickým výstupem je aplikace provádějící grafické transformace obrazu s příslušnou interpolační metodou na grafickém čipu. Tato aplikace má za účel jak demonstrování vizuálních výsledků jednotlivých transformací a interpolací, tak také časové testování při několikanásobném provádění transformací. Aplikace je také schopna přehrávat video a podle zadání jej transformovat.

Porovnání vizuálních a časových vlastností interpolací OpenCV se mnou vytvořenými algoritmy nebyl možný, jelikož knihovna OpenCV má všechny interpolace upraveny, čímž ovšem tyto interpolace ztrácí jejich základní vlastnosti.

Vizuální výsledky jsou podle očekávání splněny. Tak jak byly metody představeny, tak také nakonec dopadly jejich vizuální výsledky. V oblasti testování času byla překvapivě bilineární interpolace stejně rychlá jako metoda nejbližší soused, naopak bikubická interpolace byla mnohem pomalejší než bilineární. U metody sinc se s velkou časovou náročností počítalo, takže dosáhla předpokládaných výsledků.

Budoucí vývoj interpolací vidím jako nepravděpodobný. Všechny dnes používané interpolace byly zmíněny v tomto textu a nepředpokládám objevení nové převratné interpolační metody.

Tato práce je založena na rychle se vyvíjejícím systému CUDA. Proto bych viděl velké možnosti rozšiřování této aplikace a zpracování obrazu za použití vysokého výkonu moderních grafických karet. Systém CUDA nabízí širokou škálu možností jak optimalizovat provádění jádra. Některé optimalizace se podařilo implementovat, ale ještě zbývá mnoho části, které by se daly upravit.

Literatura

[1] Němeček Petr: Geometrické transformace obrazu. Brno, 2009, diplomová práce, FIT VUT v Brně.

[2] Lagrangeova interpolační křivka. [Online] 28. 12. 2008. [Citace: 28. 12. 2008.] http://www.hyperkrychle.cz/curves/lagrange.html

[3] Bicubic interpolation,*Wikipedia* [Online] 28. 12. 2008. [Citace: 28. 12. 2008.] http://en.wikipedia.org/wiki/Bicubic_interpolation

[4] Bilineární interpolace,*Wikipedia* [Online] 26. 12. 2008. [Citace: 26. 12. 2008.] http://cs.wikipedia.org/wiki/Bilineární_interpolace

[5] Vincent Bockaert, Interpolation, [Online] 26. 12. 2008. [Citace: 26. 12. 2008.] http://www.dpreview.com/learn/?/key=interpolation

[6] Helmut Dersch, Testing Interpolator Quality, Revised June 30, 1999 http://www.all-in-one.ee/~dersch/interpolator/interpolator.html

[7] Kubická interpolace,*Wikibooks* [Online] 26. 12. 2008. [Citace: 28. 12. 2008.] http://cs.wikibooks.org/wiki/Geometrie/C2_kubická_interpolace

[8] Tomáš Slavíček ,Všudypřítomná interpolace [Online] 28. 12. 2008. [Citace: 28. 12. 2008.] http://digiarena.zive.cz/default.aspx?server=1&article=4416

[9] NVIDIA® CUDA™ [Online] 28. 12. 2008. [Citace: 28. 12. 2008.] http://www.nvidia.com/object/cuda_what_is.html

[10] Thomas Littschwager, Jak pracují špičkové grafické karty [Online] 28. 12. 2008. [Citace: 28. 12. 2008.] http://www.chip.cz/cs/clanky/jak-pracuji-spickove-graficke-karty.html

[11] GPU,*Wikipedia* [Online] 29. 12. 2008. [Citace: 29. 12. 2008.] http://cs.wikipedia.org/wiki/GPU

[12] OpenCV,*Wikipedia* [Online] 29. 12. 2008. [Citace: 29. 12. 2008.] http://cs.wikipedia.org/wiki/OpenCV

[13] Nvidia CUDA : Compute Unified Device Architecture. Nvidia CUDA Programming Guide [online]. 2007, no. 1.0 [Citace: 19. 05. 2009.].

[14] Nvidia CUDA : Compute Unified Device Architecture. Nvidia CUDA Programming Guide [online]. 2009, no. 2.2 [Citace: 20. 05. 2009.].

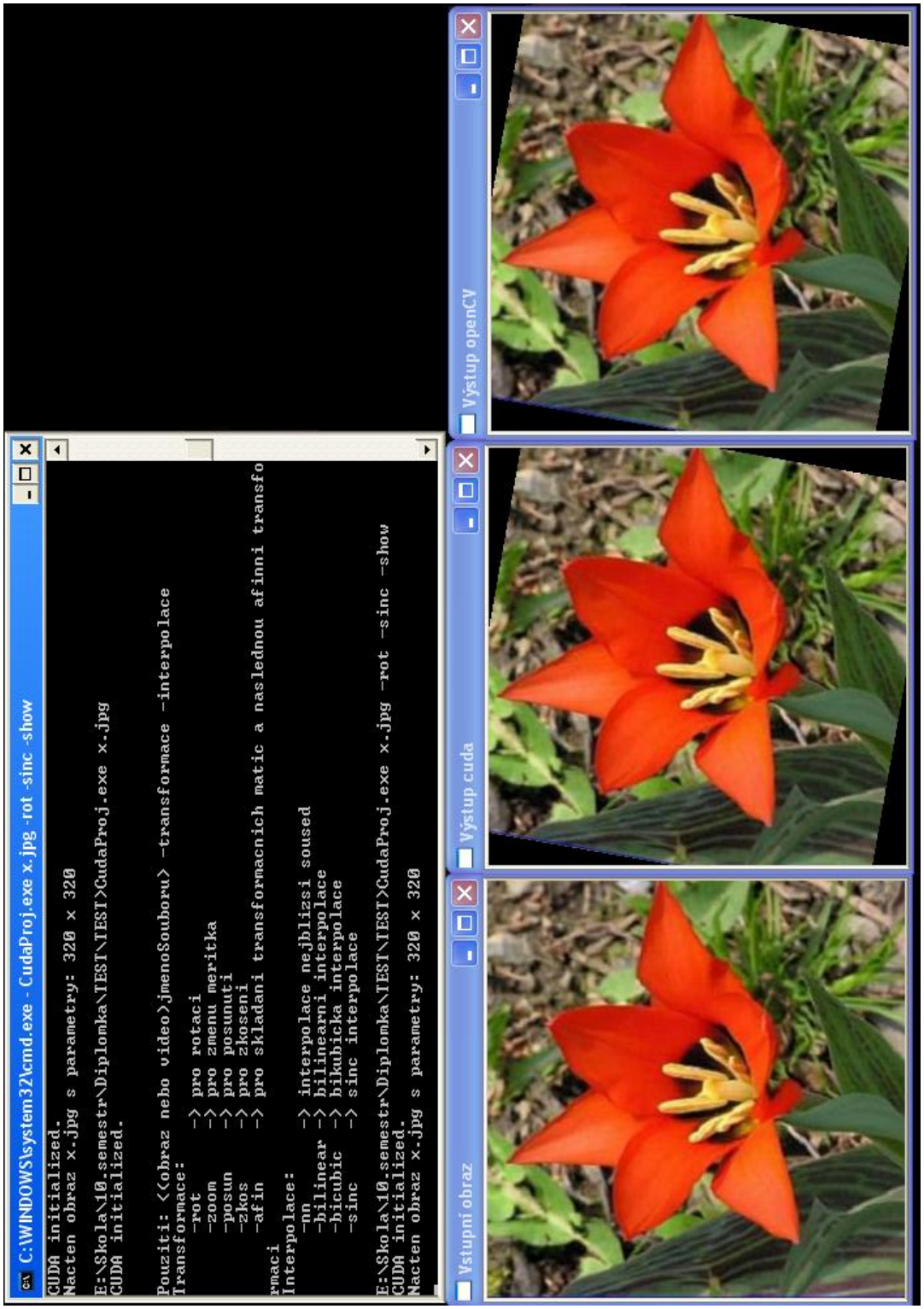
[15] Nvidia CUDA : Mike Bailey, Oregon State university [online]. 2009, [Citace: 20. 05. 2009.].

[16] BRADSKI, Gary, KAEHLER, Adrian. OpenCV : Computer Vision with the OpenCV. Mike Loukides; Robert Romano. 1005 Gravenstein Highway North, Sebastopol, CA 95472 : O'Reilly Media, Inc., 2008. 577 s. ISBN 978-0-596-51613-0.

Seznam příloh

Příloha 1. Ukázka aplikace
Příloha 2. CD

Příloha 1



Obrázek 9.1 Vzhled aplikace