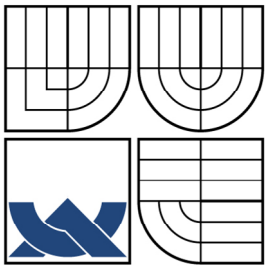
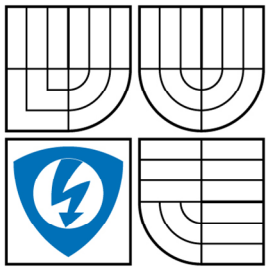


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A
KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV AUTOMATIZACE A MĚŘÍCÍ TECHNIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF CONTROL AND INSTRUMENTATION

DISTRIBUCE MATEMATICKÝCH VÝPOČTŮ V PC

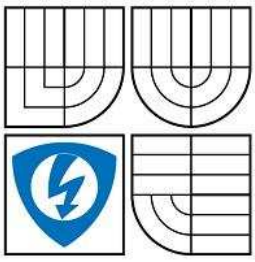
DISTRIBUTED MATHEMATICAL CALCULATION ON THE PC

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE — TOMÁŠ FUKSA
AUTHOR

VEDOUCÍ PRÁCE — Ing. PETYOVSKÝ PETR
SUPERVISOR

BRNO 2009



VYSOKÉ UČENÍ
TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

Ústav automatizace a měřicí techniky

Bakalářská práce

bakalářský studijní obor
Automatizační a měřicí technika

Student:	Tomáš Fuksa	***ID:***	73121
Ročník:	3	***Akademický rok:***	2008/2009

NÁZEV TÉMATU:

Distribuce matematických výpočtů v PC

POKYNY PRO VYPRACOVÁNÍ:

Úkolem studenta je nastudovat a demonstrovat možnosti distribuce výpočtu v PC pomocí grafických procesorů. Zadání předpokládá znalosti programovacího jazyka C/C++.

1. Seznamte se s výpočetními prostředky současných grafických karet (GPGPU).
2. Nastudujte možnosti distribuce obecných matematických výpočtů prováděných v grafických procesorech.
3. Definujte výhody a nevýhody použití GPGPU pro řešení úloh.
4. Zvolte vhodný vývojový nástroj pro implementaci algoritmů výpočtu v grafickém procesoru.
5. Navrhněte a realizujte několik příkladů demonstrující výpočetní možnosti těchto grafických procesorů.
6. Určete rychlost výpočtu stejné úlohy při použití GPGPU a běžného CPU. Dále srovnejte náročnost obou implementací.
7. Zhodnoťte dosažené výsledky.

DOPORUČENÁ LITERATURA:

[1] Harris, M. : Mapping computational concepts to GPUs; SIGGRAPH 2005
[2] Prata, S. : Mistrovství v C++, Computer press 2004, ISBN 80-251-0098-7
[3] Virius, M. : Jazyky C a C++, Grada 2006, ISBN 80-247-1494-9

Termín zadání:	9.2.2009	***Termín odevzdání:***	1.6.2009

Vedoucí práce: Ing. Petr Petyovský

prof. Ing. Pavel Jura, CSc.
Předseda oborové rady

ABSTRACT

Cílem této bakalářské práce je zhodnocení a možností programů vykonávaných na grafických procesorech. Implementace několika příkladů a jejich srovnání s obdobnými programy vykonávanými na běžném procesoru. Díky masivnímu paralelismu grafických karet je možné rychle zpracovávat velkými objemy dat. Součástí práce je výběr a popis vhodného vývojového nástroje k demonstrování těchto vlastností.

Klíčová slova: Grafický procesor, GPGPU, Nvidia, CUDA, Paralelismus, OpenCL.

ABSTRACT

The goal of this bachelor's thesis is to evaluate the possibility of the programs for the graphics processors. Next goal of this thesis is to implement several examples, comparisons with similar programs for common processors. Thanks to the massive parallelism of graphics cards it is possible to quickly process with a large volume of data. Part of this work is about choosing and describing of a suitable SDK to demonstrate these characteristics.

Keywords: Graphic processor, GPGPU, Nvidia, CUDA, Paralelism, OpenCL.

Bibliografická citace

FUKSA, Tomáš. *Distribuce matematických výpočtů v PC*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2009. 58 s., 3 přílohy. Vedoucí práce. Ing. Petyovský Petr

P r o h l á š e n í

„Prohlašuji, že svou diplomovou práci na téma " Distribuce matematických výpočtů v PC" jsem vypracoval 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 uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.“

V Brně dne : Podpis:

P o d ě k o v á n í

Děkuji tímto svému vedoucímu ing. Petru Petyovskému za cenné připomínky a rady při vypracování diplomové práce a za zapůjčení studijního materiálu.

V Brně dne : Podpis:

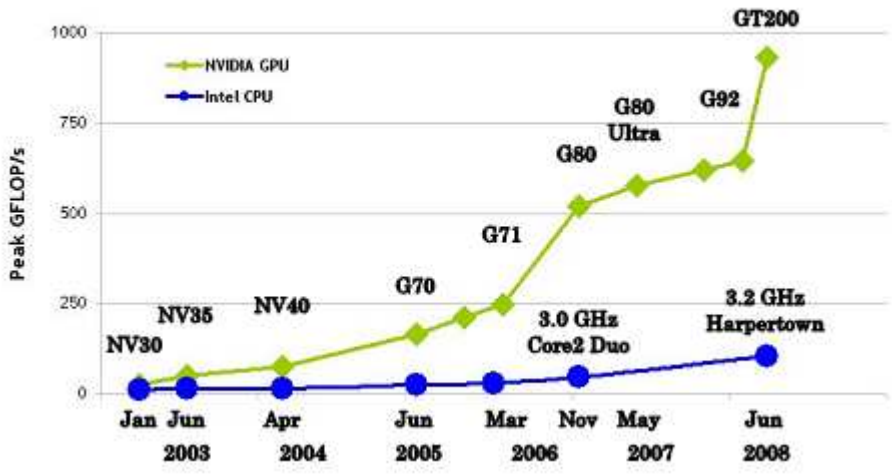
Obsah

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

1. ÚVOD

V posledních několika letech lze sledovat trend prudkého zrychlování běžně dostupných výpočetních prostředků. K nejrychleji se vyvíjejícím se komponentům patří bezesporu grafické karty viz Obrázek 1, jejichž vývoj je hnán dopředu potřebami trhu s počítačovými hrami. Tyto aplikace jsou velice náročné na výpočetní čas a jejich potřebám je přizpůsobena architektura GPU (Graphic Processing Unit) tak, že umožňují provádět mnoho paralelních výpočtů současně, což dohromady s velkou propustností paměti poskytuje obrovský výpočetní výkon, který se postupně začal využívat i pro urychlení výpočetně náročných negrafických aplikací. Tento trend vedl zpětně k vývoji karet specializovaných pro masivní paralelní úlohy - Nvidia Tesla (viz 2.2).



Obrázek 1 - Zobrazení rozdílů výkonů CPU a GPU Převzato z [1]

2. VÝHODY A PRINCIPY PROGRAMOVÁNÍ NA GPU

Hlavní výhodou grafických karet je paralelismus prováděných operací. V jednu chvíli běží mnoho vláken provádějících vždy stejné instrukce, které jsou však aplikovány na odlišná data - což umožňuje rychlejší zpracování daného objemu dat, než kdyby byla tato data zpracovávána lineárně na klasickém procesoru.

Tento princip je ale zároveň hlavní nevýhodou programování na GPU, protože všechna vlákna musí provádět stejné instrukce. Proto je třeba vykonávaný program optimalizovat tak, aby každá jeho část zbytečně nezdržovala ostatní vlákna. Je potřeba si dávat pozor na větvení, cykly a přístup do globální paměti. Při větvení musí program projít obě větve, přičemž použije výsledky pouze jedné. V případě cyklů se ve všech vláknech cyklus opakuje tolikrát, dokud neskončí nejdelší cyklus.

2.1 SROVNÁNÍ VÝKONŮ GPU A CPU

V současných, nedávno vydaných grafických kartách jsou patrné velké rozdíly ve výkonu (Tabulka 1a Tabulka 2) - ten je závislý hlavně na ceně a datu vydání komponentu. Zároveň je třeba si uvědomit, že se jedná o výrobcem udávané hodnoty, které jsou dosahovány za ideálních podmínek a paměťová propustnost či výpočetní výkon nebude v praxi těchto hodnot dosahovat z důvodů např. neoptimálně napsaného kódu.

Jméno	Propustnost paměti [GB/sec]	GFLOPS	Rychlost vytváření pixelů [MPixels/sec]	Rychlost vytváření textur [Texels/sec]
ATi Radeon HD 4830	57.6	736	9200	18400
ATi Radeon HD 4550	12.8	96	2400	4800
nVidia GeForce GTX 260 (216 Shaders)	111.888	803.52	16128	41472
nVidia GeForce 9400 GT	12.8	44.8	4400	4400
nVidia GeForce 8600 GTS	32	92.8	5400	10800
nVidia GeForce 8800GT	57.6	336	9600	33600
nVidia GeForce 9600GT	57.6	208	10400	20800

Tabulka 1 Parametry několika grafických karet.

Převzato z [10]

CPU	Výkon [GFLOPS]
AMD Athlon (600 mhz)	2.4(SP), 1(DP)
Pentium 4 (2 ghz)	8 (SP)
Pentium 4 (3 ghz)	12
Athlon 64 X2 4600 (2.4GHz)	14.7
G5 Dual (2.3GHz)	30

Tabulka 2 Výkon vybraných procesorů.

2.2 NVIDIA TESLA

Rozvoj matematických a vědeckých výpočtů na grafických kartách vedl zpětně k vývoji specializovaného hardwaru, který už není určený přímo pro grafické aplikace a zábavní průmysl, ale pro masivní zrychlení vědeckých výpočtů.

Firma Nvidia přišla s řadou karet Tesla určených pro řešení specifických paralelních úloh. Samotná karta Tesla C1060 má výkon téměř jeden TFLOPS a 4GB společné paměti [12].

Tyto karty lze skládat do výpočetních clusterů což navyšuje výkon. Jako příklad může být uveden osobní superpočítač AMAX ServMax PSC-2 obsahující 4 karty Nvidia Tesla s výpočetním výkonem 4 TFLOPS a 16GB společné paměti [13].

2.3 VÝBĚR PLATFORMY

Pro implementaci a praktickou realizaci příkladů jsem se rozhodl pracovat s vývojovým nástrojem Nvidia CUDA a to z několika důvodů – jazykem pro psaní funkcí na grafické kartě je C++ obohacené o další prvky a s tím související snadná integrace do vývojového prostředí MS Visual C++ (viz 5.6). Nvidia CUDA je dostupná zdarma a obsahuje dobře srozumitelnou nápovědu, průvodce a také velké množství příkladů aplikací s přiloženým zdrojovým kódem.

Konkurenční vývojový nástroj od firmy ATI/AMD byl v době výběru prostředí placený a dostupný jen pro některé typy karet (ATI FireStream) a až v průběhu práce došlo k jeho otevření široké komunitě vývojářů[16].

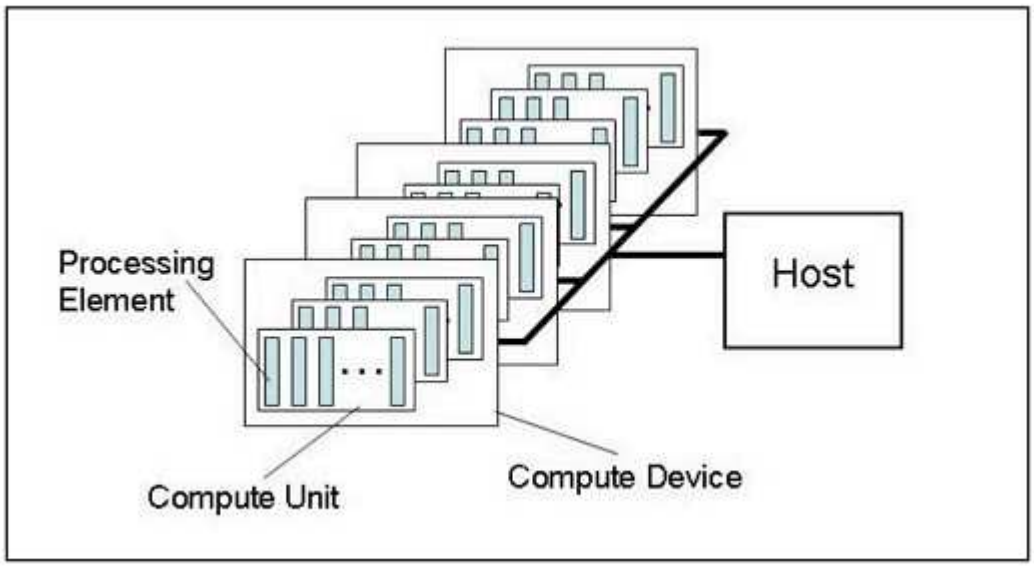
Další výhodou Nvidia CUDA je snadná implementace do C++ aplikací v podobě předkompilovaných knihoven (viz 5.5.13).

Programování v Nvidia CUDA se mnoho vývojářů a tak lze na oficiálním fóru nalézt velké množství vyřešených problémů, což velmi usnadňuje vývoj aplikace.

Posledním kritériem výběru byla dostupnost hardware pro testování demonstračních programů – grafická karta s čipem Nvidia je součástí mého počítače a také mi byl umožněn přístup k několika dalším zdrojům na fakultě elektrotechniky VUT.

Nvidia CUDA je dále vyvíjena a současný vývoj směřuje k implementaci OpenCL do této platformy.

OpenCL je otevřený standart pro všeobecné paralelní programování na různorodých systémech. Poskytuje jednotné programové prostředí pro softwarové vývojáře a umožňuje psát efektivní a přenosný kód pro velké množství zařízení [15].



Obrázek 2 – Model platformy OpenCL. Převzato z [15]

3. ARCHITEKTURA GEFORCE 8800

Uvnitř karty GeForce 8800, kterou jsem si vybral pro popis architektury GPU, se nachází osm multiprocesorů. Multiprocesor je jednotka obsahující dvakrát osm skalárních procesorů věnujících se převážně operacím sčítání a násobení (jsou taktovány na dvojnásobné frekvenci) a dva páry Super Function Unit umožňující goniometrické operace, logaritmy a podobně.

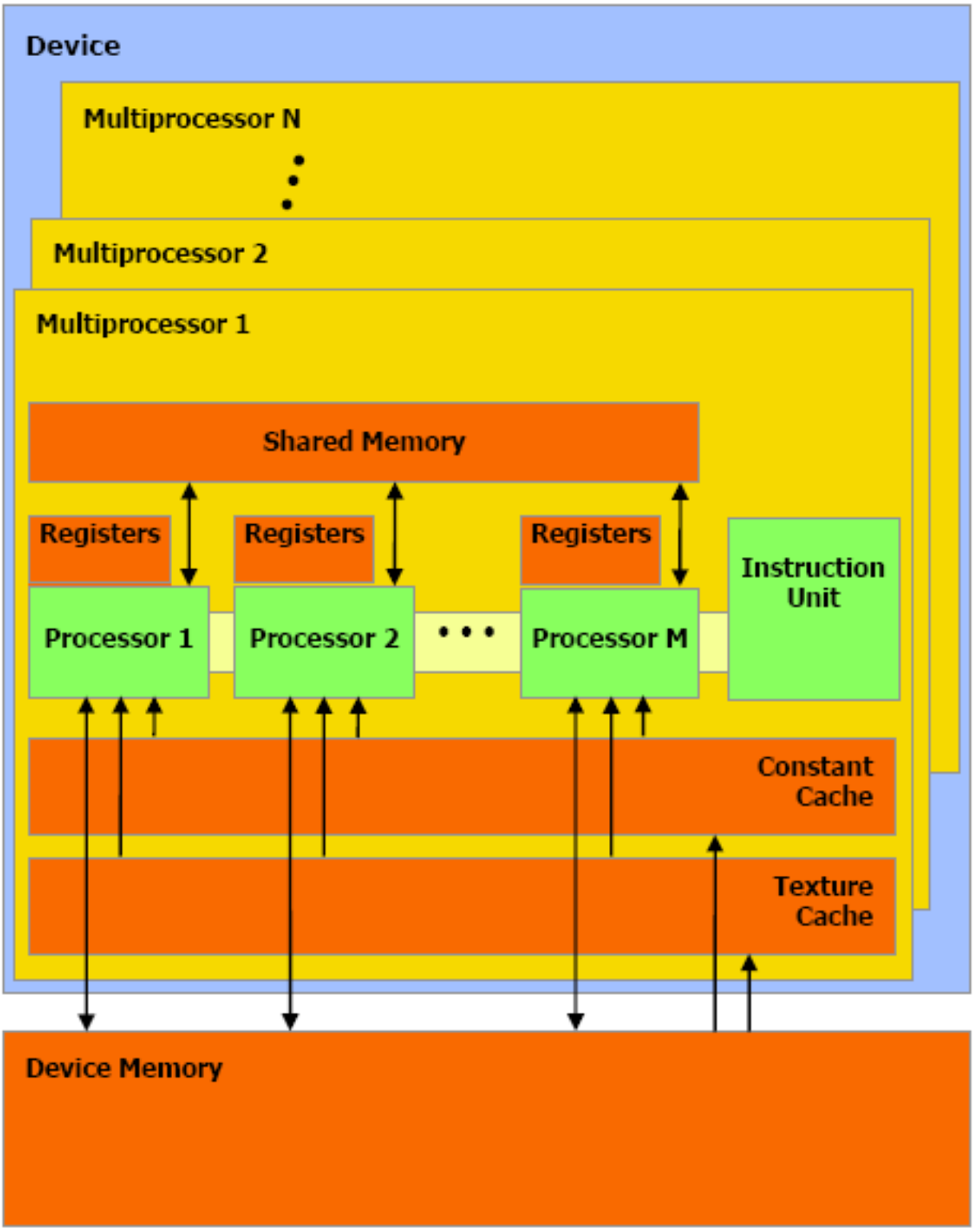
Tím, že každý multiprocesor obsahuje všechny prvky 2x máme k dispozici 16 multiprocesorů (viz Obrázek 3).

Multiprocesor vždy zpracovává skupinu 32 vláken (thread) souhrnně nazývanou warp. Dvojnásobná frekvence tedy umožňuje provést instrukci na 16 skalárních procesorech 2x během jednoho hlavního cyklu. Super Function Unit je čtyřikrát pomalejší, avšak výpočet není prováděn s vysokou přesností, což způsobuje, že výpočet není tak časově náročný. Za jeden cyklus je možné učinit 256 operací.

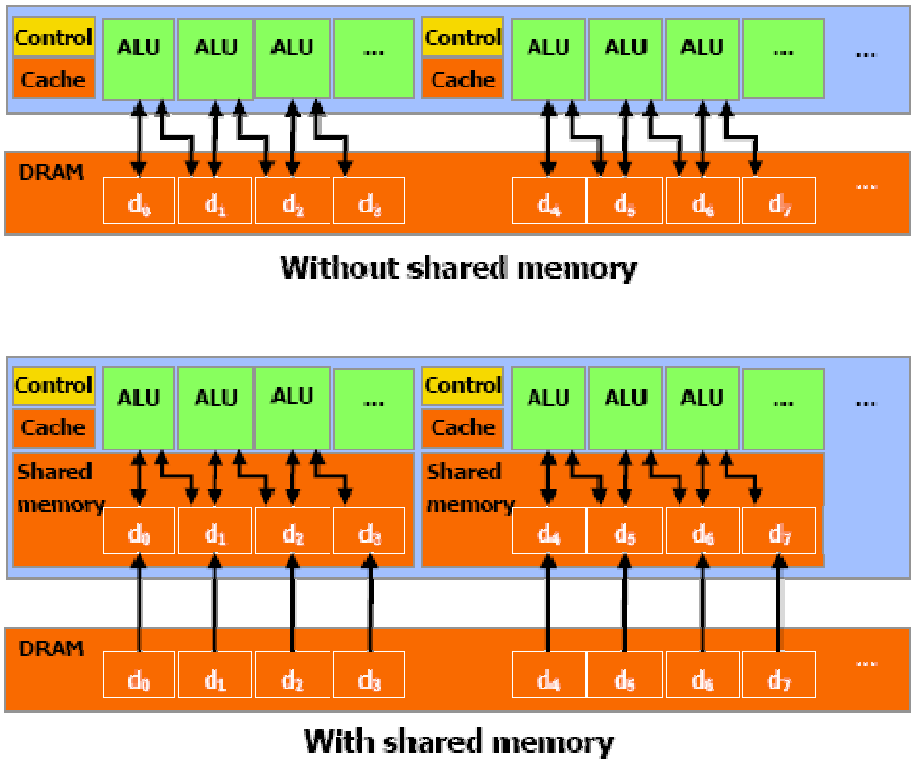
Na GeForce 8800 je možné zapisovat kamkoliv do pamětí a také odkudkoliv číst. Pokud se však přistupuje ke sdílené paměti, je zde vzhledem k neexistenci vyrovnávací paměti vysoká latence (200-300 cyklů). Aby se nemuselo neustále přistupovat k hlavní sdílené paměti, jsou multiprocesory vybaveny vlastní sdílenou pamětí o velikosti 16KB. Čím více vláken je jedním multiprocesorem zpracováváno, tím méně paměti má jedno vlákno k dispozici. Tato paměť se dělí na šestnáct banků a přistupuje se k ní pomocí šestnácti 32bitových sběrnic. (Schéma viz Obrázek 4)

Existuje zde vyrovnávací paměť pro texturovací jednotky o velikosti 8 KB na jeden multiprocesor. Nelze do ní zapisovat, ale v případě dobře seřazených požadavků umožňuje efektivní čtení. Dále jsou zde registry, které snižují latenci při velkém množství k nim přistupujících vláken (snižuje se ale množství paměti vyhrazené pro jedno vlákno).

Poslední pamětí GeForce 8800 je 64KB pro konstanty - tato paměť má cache a je rozdělena rovnoměrně mezi všechny multiprocesory.



Obrázek 3 - Schéma architektury Nvidia GeForce 8800. Převzato z [1]



Obrázek 4 - Schéma paměťového prostoru GeForce 8800. Převzato z [1]

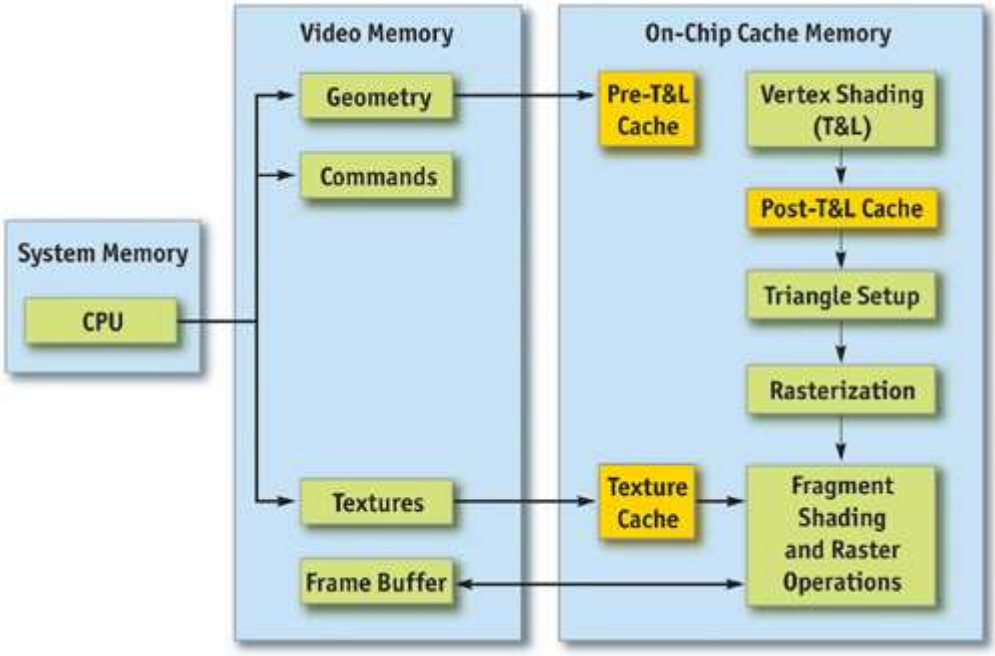
3.1 ROZDÍLY MEZI CPU A GPU

Hlavní rozdíl mezi GPU a CPU je v masivním paralelismu GPU. Grafická karta je určená na zpracování velkého množství dat v co nejkratším čase. U CPU jde o co největší rychlost prováděných obecných instrukcí. Pokud by byl spuštěn běžný program na grafické kartě, nedokázal by využít jejího paralelismu a ve výsledku by běžel mnohem pomaleji.

Obrázek 5 - Schéma rozdílného uspořádání CPU a GPU. Převzato z [1]

4. PRŮBĚH ZPRACOVÁNÍ DAT NA GPU (GRAFICKÁ PIPELINE)

Grafická pipeline je posloupností instrukcí prováděných na různých částech GPU v průběhu zpracování dat. Výpočet je rozdělen do několika celků, které jsou schopny pracovat paralelně. Základní sadou dat jsou vrcholy a textury. Tato data reprezentují většinou souřadnice vrcholů polygonů a informaci o jejich barvě. Nicméně nezáleží, jestli tato data jsou opravdu informace o obrazu, nebo data pro jiný matematický výpočet – s daty se bude pracovat v každém případě stejně.



Obrázek 6 - Schéma grafické pipeline. Převzato z [1]

4.1 VERTEX SHADING

Do této části vstupuje sada vrcholů (vertex – pozice modelu, normály vrcholů, koordináty textur) a ta je následně zpracována na sadu atributů vhodných pro oříznutí a rasterizaci.

4.2 TRIANGLE SETUP A RASTERIZACE

V této fázi se z vrcholů vytvoří jednoduché geometrické obrazce - trojúhelníky. Z těch se následně vytvoří rastrová informace tak, že se vektorové údaje uvnitř oblasti trojúhelníků převedou na fragmenty (jednotky obsahující informaci o souřadnicích, barvě, hloubce a průhlednosti). Fragment je tedy více než jen obrazový bod, který nese pouze informaci pozice a barvy.

4.3 FRAGMENT SHADING

Také se nazývá Pixel shading. V této části jsou zpracovány jednotlivé fragmenty - informace o barvě, hloubce, světlosti a podobně se sloučí v jednu informaci.

Také se zde rozhoduje, jestli se fragment vůbec použije nebo se zahodí.

5. PROGRAMOVÁNÍ NA NVIDIA CUDA

5.1 KERNEL

Funkce zvané kernel se provádějí paralelně na zařízení. Každý jednotlivý běh kernelu má své unikátní identifikační číslo, které zaručuje jednotlivým vláknům přistupovat k jiným datům. Tato data jsou většinou reprezentována jako pole či matice a identifikační číslo vlákna tak umožní přístup k datům buňky pole.

Vlákna spolu mohou spolupracovat za použití přístupu do sdílené paměti. Je však potřeba zajistit, aby všechna vlákna tento přístup provedla. Pro koordinaci vláken se používá funkce *__syncthreads()*, jež se umístí do místa programu, ve kterém chceme, aby se všechna vlákna synchronizovala.

Protože jsou vlákna spouštěna po blocích s pevným množstvím vláken na blok (určuje se při volání kernelu) je možné, že při zpracování dat se vykoná větší množství vláken, než je počet datových struktur. Proto je potřeba uvnitř kernelu testovat podmínku, která povolí přístup do struktury pole jen těm vláknům, jejichž identifikační číslo bude uvnitř intervalu počtu prvků pole.

Například pokud budeme mít pole 10 prvků a kernel, který bude vykonán ve 3 blocích po 4 vláknech, tak se kernel vykoná souběžně 12x. Uvnitř kernelu se tedy musí zjistit identifikační číslo vlákna a u vláken s číslem větším než 10 se všechny instrukce přeskočí, případně se provede výpočet nad existujícími, ale prázdnými daty.

5.2 URČENÍ IDENTIFIKAČNÍHO ČÍSLA VLÁKNA

Identifikační číslo vlákna se zjistí za pomoci zabudovaných proměnných `blockIdx.x, blockDim.x, threadIdx.x`. Kde

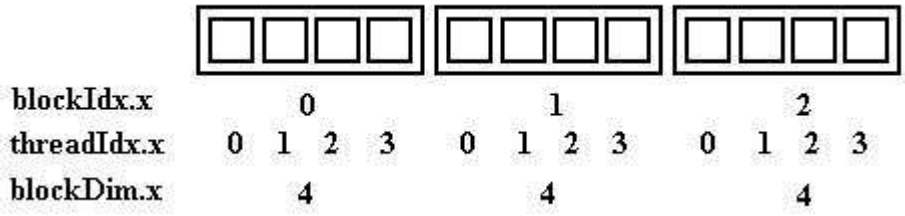
`blockIdx.x` obsahuje číslo bloku vláken

`blockDim.x` reprezentuje počet vláken na blok

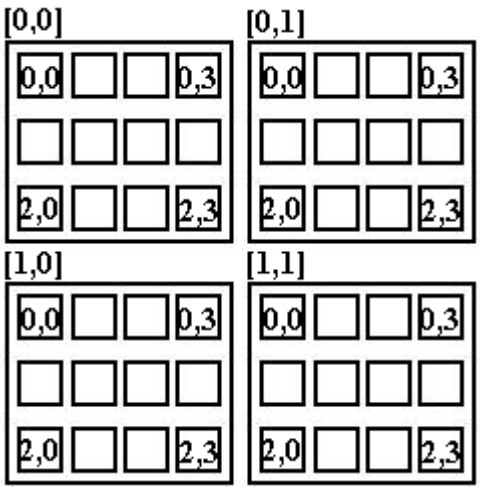
`threadIdx.x` obsahuje identifikační číslo vlákna v rámci bloku

a to vztahem

```
blockIdx.x*blockDim.x + threadIdx.x
```



Obrázek 7 - Schéma 3 bloků po 4 vláknech



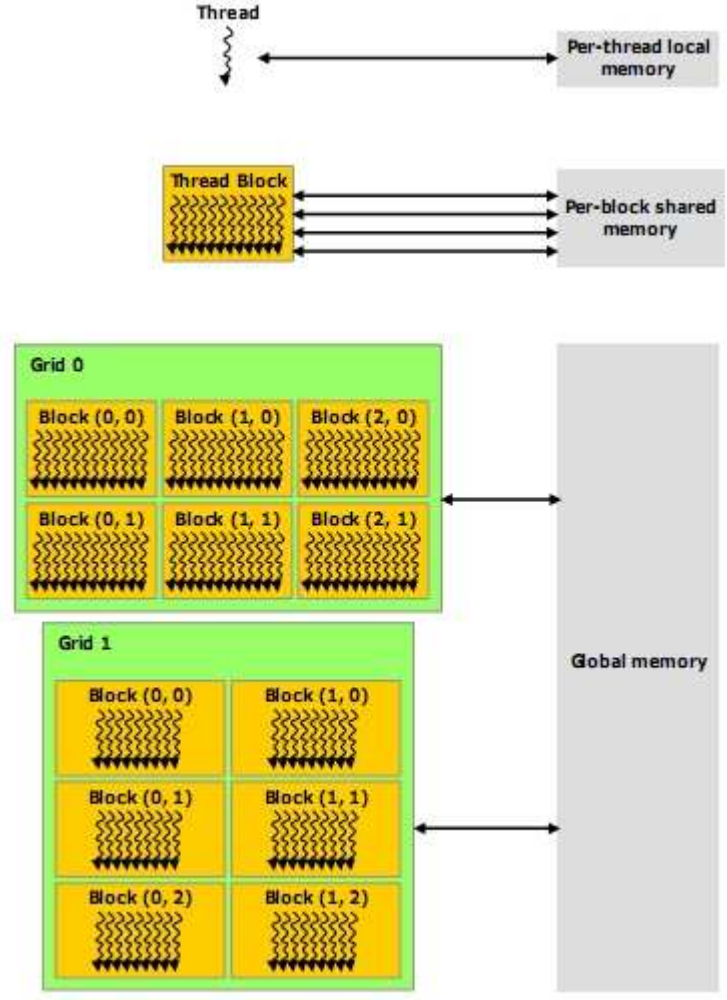
Obrázek 8 - Schéma dvojrozměrného řazení bloků a vláken

Bloky i vlákna mohou být řazeny i dvoj- či trojrozměrně. Přístup k nim je potom realizován přes `blockIdx.x`, `blockIdx.y`, `blockIdx.z`. (Stejně tak i pro jednotlivá vlákna - přes `threadIdx.x`, `threadIdx.y`, `threadIdx.z`) viz Obrázek 8.

5.3 HIERARCHIE PAMĚTI

Vlákna mohou přistupovat do různé úrovně paměti. Každé vlákno má svoji vlastní paměť na proměnné (per-thread memory). Dále může přistupovat do paměti sdílené vlákny v jednom bloku (per-block memory), nebo do globální paměti zařízení (global memory). Viz Obrázek 9.

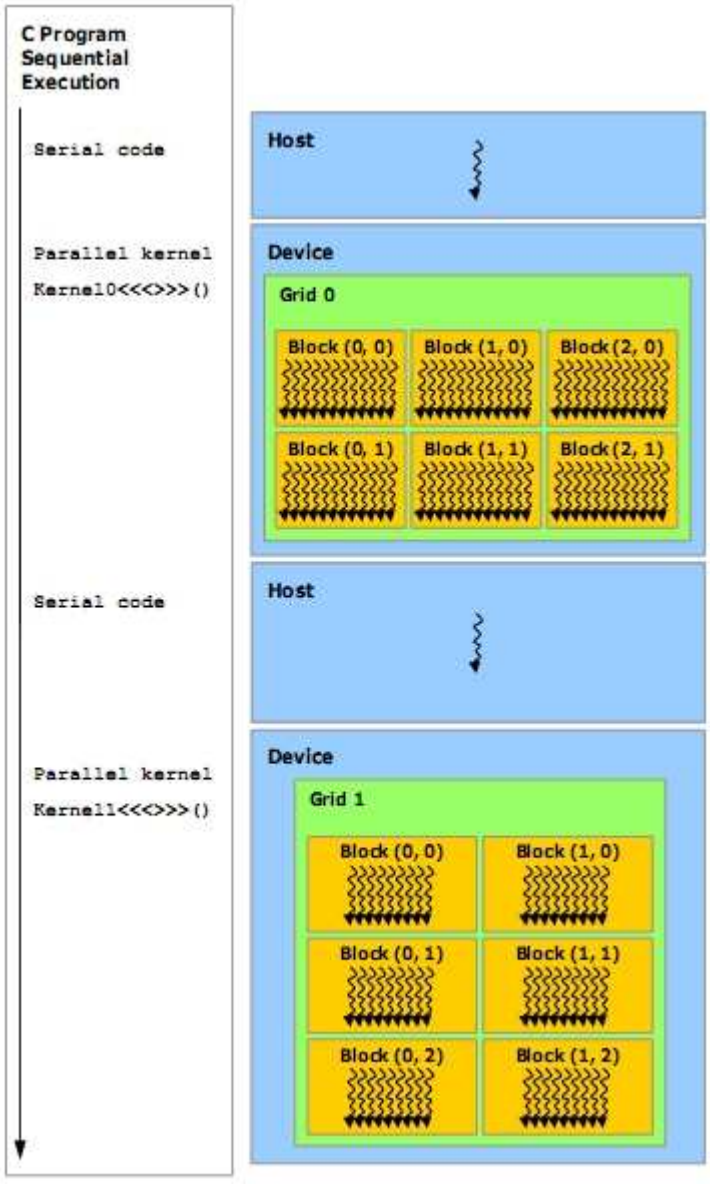
Existují také dva druhy speciální paměti pouze pro čtení – globální paměť pro konstanty a pro textury, které jsou optimalizovány k jiným paměťovým úkonům (viz Obrázek 3).



Obrázek 9 - Schéma hierarchie paměti. Převzato z [1]

5.4 HOST A ZAŘÍZENÍ

Vlákna spouštěná programem jsou vykonávána na odděleném zařízení, které se tak chová jako koprocesor. Zařízením (device) je zde myšlena grafická karta, podporující rozhraní CUDA. Program běžící na CPU (host) se stará o správu paměti a spouštění jednotlivých kernelů viz Obrázek 10.



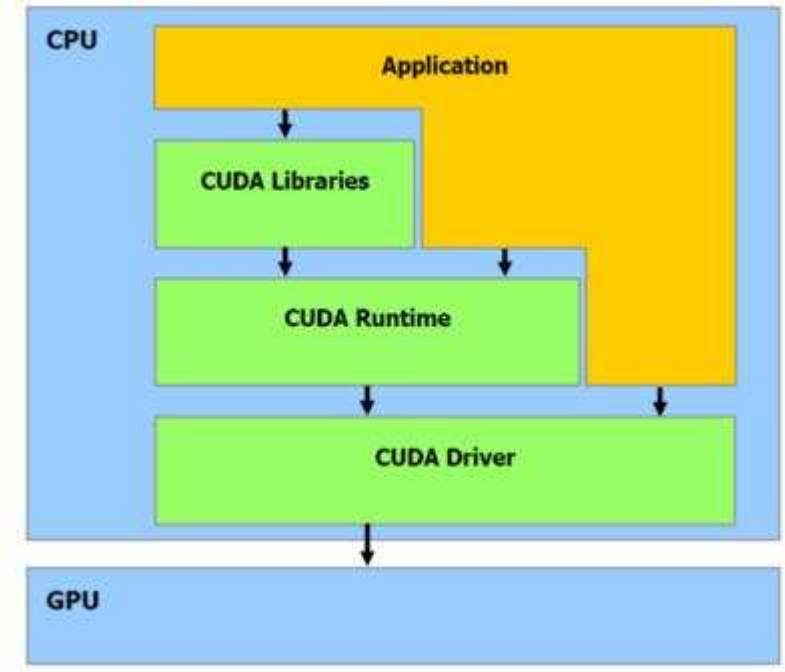
Obrázek 10 - Schéma možného běhu programu a spouštění kernelů na zařízení. Převzato z [1]

5.5 CUDA API

Cuda API je rozšířením jazyka C [14]. Tato rozšíření v podobě nových klíčových slov se dají rozdělit do dvou skupin.

Při psaní aplikace je možné využívat dvě úrovně API:

- rozhraní nižší úrovně - Driver API
- rozhraní vyšší úrovně - Runtime API



Obrázek 11 - Znázornění přístupu aplikace k GPU přes různé úrovně API Převzato z [1]

Driver API umožňuje mít nad programem větší kontrolu, ale je obtížnější na programování, protože programátor se přímo zabývá inicializací, zatížením modulů a podobně. Driver API také umožňuje získat podrobnější informace ze zařízení, než je možné získat z runtime API. Například dostupná volná paměť zařízení lze zjistit jen pouze za pomocí funkcí driver API.

Cuda runtime API je jednodušší na správu kódu, neboť obsahuje např. implicitní inicializace či správu modulů.

Mezi driver a runtime API není žádný markantní rozdíl v rychlosti vykonávání kódu.

Všechny rozdíly mezi driver a runtime API jsou pouze v hostitelském kódu a nemají na kernel žádný vliv (respektive je lze zavolat pouze z hosta).

Dalším možným přístupem je využití CUDA knihoven, které obsahují předem připravené funkce pro práci s poli dat, maticemi a podobně a není potřeba se vůbec zabývat psaním kernelu. Jako příklad lze uvést funkci

```
cublasAlloc(int n, int elemSize, void **devicePtr)
```

pro alokování paměti na zařízení či

```
cublasSgemm (char transa, char transb, int m, int n, int k, float alpha, const float *A, int lda, const float *B, int ldb, float beta, float *C, int ldc)
```

pro vypočítání součinu matic A a B, který vynásobí skalárem alpha a přičte součet k součinu matice C a skaláru beta čili

```
C = alpha * A * B + beta * C
```

5.5.1 Funkční kvalifikátory

Mezi funkční kvalifikátory jsou zařazeny deklarace funkcí, které lze spouštět na hostu či zařízení.

`__device__`	funkce se vykoná na zařízení, volání lze provést pouze ze zařízení
`__global__`	funkce se vykoná na zařízení, funkci spouští pouze host
`__host__`	funkci volá host a také se na něm vykoná

Tabulka 3 – Funkční kvalifikátory typů proměnných

5.5.2 Specifikace typů proměnných

`__device__`	Deklaruje proměnné na zařízení, přístup k těmto proměnným mají jak jednotlivá vlákna, tak i hostující program.
`__constant__`	Toto klíčové slovo umožní vznik konstanty v části paměti vyhrazeném pro konstanty. Většinou se používá zároveň s `__device__`
`__shared__`	Vytvoří proměnnou na zařízení. Tato proměnná je sdílená a je dostupná pouze v rámci bloku vláken, ve kterém byla vytvořena.

Tabulka 4 – Specifikace typů proměnných

5.5.3 Specifikace volání

Do této části jsou zahrnuty definice způsobu zavolání funkcí deklarovaných za pomoci `__global__`. Do klasického volání funkce je vložen prvek

`<<< Dg, Db, Ns, S >>>`,

který určuje, jakým způsobem bude kernel vykonán.

`Dg`	Číslo typu dim3 (třídimenzionální CUDA vektor typu `integer`), která určuje velikost matice bloků. Vytváří se pouze dvourozměrné matice.
`Db`	Specifikuje rozměr a velikost každého bloku (až 3 rozměry).
`Ns`	Určuje velikost sdílené paměti, která je přiřazena každému bloku.
`S`	Určí asociovaný stream (viz. 5.5.10).

Tabulka 5 – Specifikace volání

Parametry `Ns` a `S` nejsou povinné – pokud jsou vynechány je jejich hodnota nula.

Například funkce deklarovaná

```
__global__ void SampleKernel (float * vstup);
```

Bude následně zavolána

```
SampleKernel <<< Dg, Db, Ns >>>(vstup);
```

5.5.4 Zabudované proměnné

CUDA obsahuje několik základních proměnných, ze kterých lze za běhu programu vyčítat informace potřebné pro řízení toku programu nebo přístupu k paměti. Mezi tyto zabudované proměnné patří například již výše zmíněné (viz 5.2) `blockIdx`, `blockDim`, `threadIdx`.

Další je např. `gridDim`, která v sobě nese hodnotu rozměru mřížky.

5.5.5 Typy proměnných

Další součástí CUDA API jsou například nové vektorové typy. Jedná se o jedno, až čtyřrozměrné struktury vycházející ze základních typů jazyka C. Za všechny uvedu pouze `int1`, `int2`, `int3` a `int4` což jsou struktury obsahující jeden až čtyři rozměry typu `integer`. Přístup k hodnotám je realizován přes prvky `x`, `y`, `z` a `w` struktury.

5.5.6 Matematické funkce

Odlišná architektura zařízení způsobuje, že bylo potřeba vytvořit vlastní způsoby provádění matematických operací zohledňující jiné aritmetické principy, než při vykonávání na obecném procesoru. Aritmetické funkce zde mají dva ekvivalenty – v základní přesnosti (single precision) a v dvojité přesnosti (double precision).

Jsou zde realizovány všechny potřebné operace od prostého sčítání až po logaritmy, goniometrické funkce či zaokrouhlování.

Dále existují speciálně upravené funkce (například `__sinf(x)`) které jsou optimalizované pro rychlost na úkor přesnosti. Také lze při kompilaci použít parametr `use_fast_math`, který převede všechny aritmetické operace na jejich rychlejší ekvivalenty - pokud existují.

5.5.7 Správa paměti

Dynamicky alokovaná paměť je na zařízení vytvářena obdobou funkce `malloc()` a to funkcí `cudaMalloc()`, která vytvoří lineární blok paměti. Pro vytvoření pole se používá `cudaMallocAray()`. Uvolnění této paměti probíhá pomocí `cudaFree()` respektive `cudaFreeArray()`. Pro každý druh alokované paměti (lineární, pole, 2D struktura,…) jsou definovány kopírovací funkce. Jeden ze vstupů do funkce určuje, kterým směrem (zařízení, host) se bude kopírovat.

Příklad:

```
const int size=100;
float*h; //ukazatel na pamet hosta
float*d; //ukazatel na pamet zarizeni
h = (float *)malloc(size); //alokace pameti na hostu
cudaMalloc((void **) &d, size); //alokace pameti na
zarizeni

cudaMemcpy(h,d,sizeof(float)*N,cudaMemcpyDeviceToHost);
//kopirovani z pameti zarizeni do pameti hosta //smer
urcuje klicove slovo cudaMemcpyDeviceToHost
```

5.5.8 Asynchronost

Některé procedury vrací řízení zpět do hlavního programu dřív, než se dokončí. Mezi tyto procedury patří všechny funkce `__global__`, dále funkce provádějící kopírování paměti pouze v rámci zařízení a funkce nastavující paměť na zařízení na určitou hodnotu.

5.5.9 Stream

Streamy umožňují organizovat vícenásobné spuštění kernelu, čímž se způsobí jeho paralelní překrytí a vykonání, což umožní lepší využití hardwaru a přispívá ke

zvýšení výkonu. Také umožňují asynchronní kopírování paměti mezi hostem a zařízením.

5.5.10 Události

Další možností řízení toku programu při asynchronním běhu jsou události (events). Je možné označit začátek výpočtu na zařízení a také konec výpočtu na zařízení. V některých případech se nehodí, aby program pokračoval dál dřív, než se výpočet na zařízení dokončí - v tom případě lze použít zarážku `cudaEventSynchronize(udalost)`, která pozdrží vykonávání programu do doby, než nastane předem definovaná `udalost`.

5.5.11 Moduly

Moduly jsou dynamicky nahrávatelné balíčky kódu pro zařízení, podobně jako jsou DLL soubory ve Windows. Umožňují tak například ve vlastním programu využívat funkce a data třetí strany.

5.5.12 Typy souborů

Kompilátor nvcc (viz 5.7) rozpoznává několik typů souborů jako soubory zdrojového kódu viz Tabulka 6.

.cu	zdrojový kód CUDA obsahující kód hosta a funkce zařízení
.cup	předzpracovaný zdrojový kód (.cu)
.gpu	vyextrahovaný kód pro zařízení (meziprodukt při kompilaci)
.cubin	soubor modulu

Tabulka 6 – Typy souborů CUDA

Dále rozpoznává všechny klasické typy souborů známé z C a C++ a několik typů souborů vzniklých jako meziprodukt kompilace, dále pak knihovny či objekty.

5.5.13 Integrace do C++ aplikace

V případě, že je potřeba zavolat některou z funkcí (nikoliv kernel) obsaženou ve zdrojovém kódu CUDA z aplikace psané v C nebo C++ je třeba ve zdrojovém kódu CUDA tuto funkci vyčlenit pro volání z jiné části programu za pomoci klíčových slov

```
extern „C“
```

V C++ části aplikace je třeba deklarovat volané funkce ve stejné podobě, jako se nacházejí ve zdrojovém kódu CUDA.

Příklad:

Sample.cu

```
__global__ void SampleKernel(int * vstup_vystup)
{ … }

extern „C“ void cuda_funkce(int * vstup, int * vystup)
{ …
// alokace paměti na GPU, kopírování vstup do GPU a další
// přípravné operace
SampleKernel<<<blocks,threads>>>(d_vstupvystup)
// zkopírování výsledku z GPU, uvolnění paměti
…
}
```

Main.cpp

```
#include <stdio.h>
…
extern „C“ void cuda_funkce(int * vstup, int * vystup);
int main()
{
…
cuda_funkce(vstup,vystup);
…
}
```

5.6 VÝVOJ V PROSTŘEDÍ MS VISUAL C++ 2005

CUDA program je možné napsat v libovolném textovém editoru a následně zkompilovat za pomoci kompilátoru nvcc (viz 5.7) obsaženého v instalaci CUDA SDK. Takto napsaný program je ovšem obtížné ladit a je potřeba znát a správně zapsat parametry kompilace.

V prostředí Windows je nejlepší volbou MS Visual C++ 2005 či MS Visual C++ 2005 Express (od verze CUDA 2.1 je podporováno i MS Visual C++ 2008) do kterého je doinstalován modul pro tvorbu CUDA projektů. Existuje také modul zvýrazňování klíčových CUDA slov a funkcí pro MSVC++).

Visual studio se postará o sestavení programu či knihovny funkcí.

5.6.1 Ladění

Vývojový nástroj neumožňuje přímo přistupovat ladícím nástrojem do těla kernelu, pokud se vykonává na GPU. Z tohoto důvodu je možné zkompilovat program v takzvaném emulačním režimu, při kterém se všechny operace jinak prováděné paralelně na GPU provedou lineárně na procesoru. V tomto režimu je již možné vstupovat dovnitř kernelů a provádět ladění - kernel se chová, jako by nad ním byl proveden vnořený cyklus počtu bloků a vláken.

V případě že by byl volán kernel

```
dim3 Gird(1000), Threads(255);
SampleKernel<<<Gird,Threads>>>(void);
```

vykonal by se cyklus

```
for (blockIdx.x=0;block.x<1000, blockIdx.x++)
     for(threadIdx.x=0; threadIdx.x<255; threadIdx.x++)
          //kód funkce SampleKernel
```

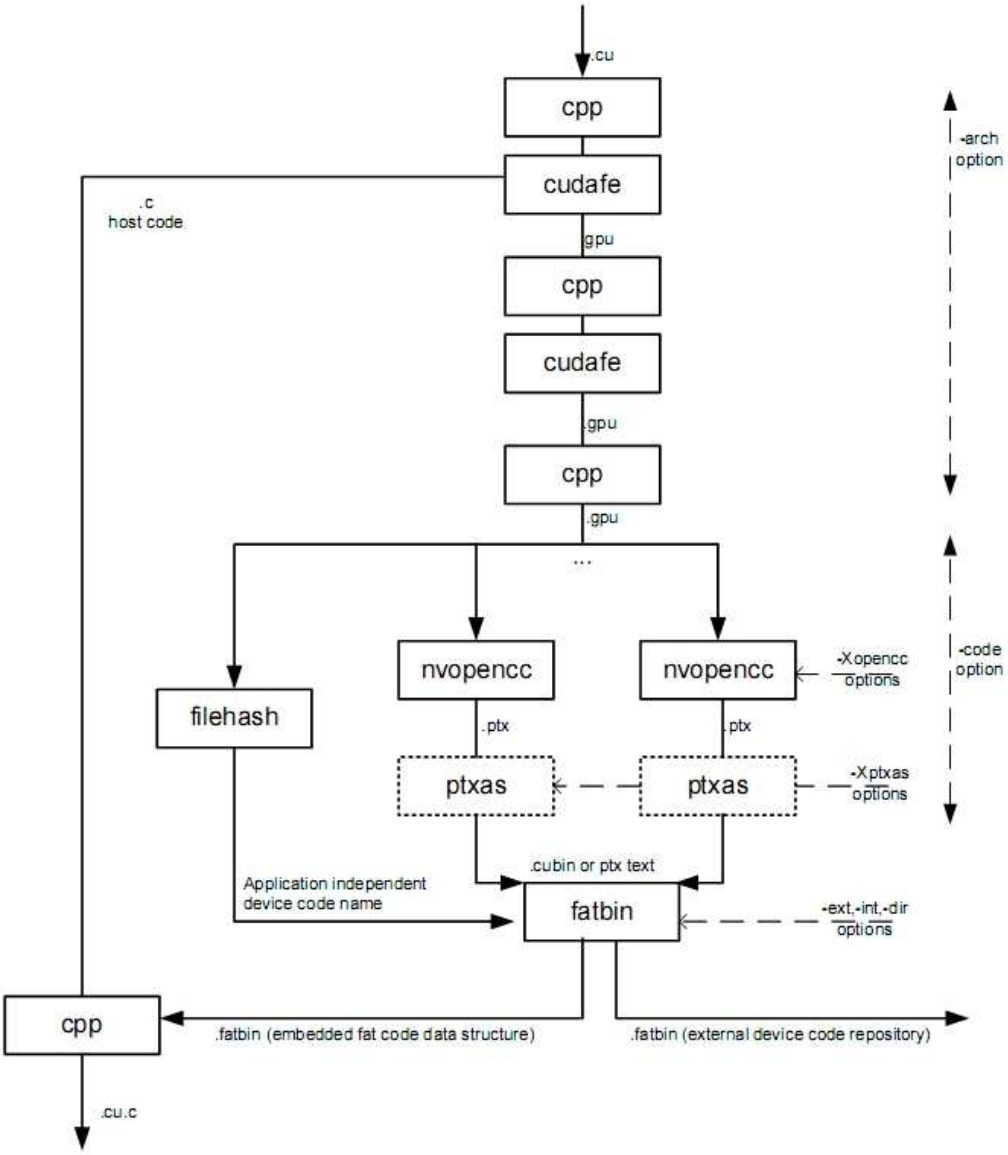
Kompilace v emulačním režimu se provádí parametrem `deviceemu` pro nvcc. V MS Visual C++ je pro CUDA projekty přímo volba `EmuDebug`, která kompilaci a ladění kódu usnadňuje. MS Visual C++ však nativně nerozpoznává řídící CUDA znaky kernelů „<<< >>>“ a nedokáže při krokování vstoupit do těla kernelu a proto je třeba do něj vložit breakpoint.

Nutno také podotknout, že emulační režim je mnohem více výpočetně náročný, než kdyby se obdobná funkce na CPU prováděla pomocí cyklů - oproti ekvivalentní funkci na CPU trvá výpočet mnohonásobně déle - nejvíc času zabírají samotné přípravné operace nad emulovanými vlákny - prázdný kernel s 1000 bloky o jednom vláknu se na CPU prováděl 178ms přičemž ekvivalentní prázdný cyklus proběhl za 0.000195ms.

Ladění kernelu umožní vhled do vnitřní logiky funkce, avšak není schopné ošetřit chybové stavy, které mohou vzniknout jen na grafické kartě (např. nedostatečná velikost grafické paměti) a nalezení chyby vznikající např. v jediném konkrétním vláknu z mnoha (např. přístup do neplatného místa v paměti) je nesnadné.

5.7 KOMPILACE V PROSTŘEDÍ CUDA

Všechny programy, které chtějí pracovat se zařízením, musí být zkompilovány pomocí CUDA kompilátoru *nvcc*. Tento kompilátor oddělí kód určený pro zařízení od kódu určeného pro procesor. Poté zkompiluje kód určený pro zařízení. Kód hostitelského programu se přenechá na zkompilování klasickému kompilátoru jazyka C. Pro správný běh *nvcc* musí být v systému tento kompilátor přítomen. V systému MS Windows je vyžadován kompilátor obsažený v MS Visual Studio. Operace v průběhu kompilace jsou znázorněny na Obrázek 12.



Obrázek 12 - Schéma průběhu kompilace programu za pomoci NVCC.

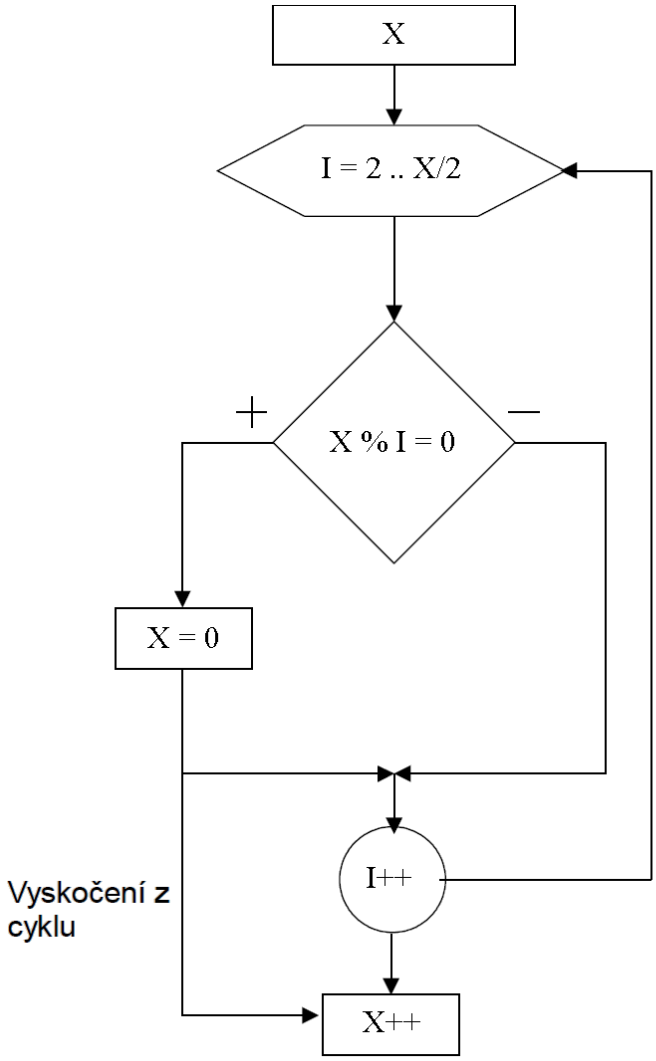
Převzato z [4]

6. PRAKTICKÁ REALIZACE

6.1 PARALELNÍ VÝPOČET PRVOČÍSEL

Hledání prvočísel v intervalu je úloha náročná na výpočetní čas. Samotné vyhledávání je řešeno hrubou silou. Zpracovávané číslo je postupně děleno všemi čísly až do své poloviny - pokud je vyděleno bezezbytku, nejedná se o prvočíslo. Ta, která prvočíslem nejsou, jsou nahrazena nulou.

Algoritmus řešení úlohy je vidět na vývojovém diagramu Obrázek 13.



Obrázek 13 – Vývojový diagram hledání prvočísel

Při tomto řešení úlohy je mnoho operací vykonáváno zbytečně (dělení sudými čísly většími než 2) a pokud by bylo cílem spočítat největší možný počet prvočísel v co nejkratším čase, byl by lepší jiný způsob (například dělení pouze předchozími nalezenými prvočísly). Pro porovnání rychlostí mezi výpočtem na GPU a CPU je tento způsob vyhovující.

V případě lineárního zpracovávání na procesoru je možné úlohu optimalizovat zarážkou - vyskočením z vyhledávacího cyklu, pokud je číslo označeno jako složené. Optimalizace je na vývojovém diagramu Obrázek 13 znázorněno větví „Vyskočení z cyklu“.

Takovéto optimalizace však není možné dosáhnout na grafickém procesoru, neboť zde je vždy vykonávána skupina 32 vláken – warp a ten je ukončen až ve chvíli, kdy výpočet dokončí všechna jeho vlákna. Protože na intervalu <2;190000> je střední hodnota vzdálenosti mezi prvočísly 8, tak téměř každý warp bude nucen čekat, až příslušné vlákno dokončí celý cyklus nad prvočíslem.

6.1.1 Implementace

Vyhledávání prvočísel je realizováno v kernelu `Prvocislo_GPU` a `Prvocislo_GPUopt`. Ekvivalentní funkce realizované na procesoru zastupují `Prvocislo_CPU` a `Prvocislo_CPUopt`. Funkce vykonávané na procesoru jsou algoritmicky podobné kernelům vykonávaným na grafické kartě. Funkce `Prvocislo_CPUopt` a `Prvocislo_GPUopt` jsou optimalizovány zarážkou. Když je číslo klasifikováno jako číslo složené, dojde k vyskočení z vyhledávacího cyklu (viz Obrázek 13 větev „Vyskočení z cyklu“). Složená čísla jsou nahrazena nulou.

Do funkce vstupuje jednorozměrné pole obsahující všechna čísla zadaného intervalu. Ve funkci jsou postupně všechny buňky otestovány na přítomnost prvočísla. Pole obsahuje 1000 prvků a pro hledání v delších intervalech je potřeba funkci zavolat vícekrát.

Program postupně vyhledává prvočísla ve zvětšujících se intervalech od 10000 do 190000 po kroku 10000 a měří čas běhu každé funkce. Měří také čas kopírování paměti, který je však zanedbatelný.

Vypočítaná prvočísla nejsou dále nijak zpracovávána, neboť by jejich kopírování a zpracování mohlo zkreslovat výsledek. Přesto byly výsledky v průběhu ladění ověřeny.

Časy jsou měřeny za pomoci funkce `clock()` knihovny `time.h` která vrací počet milisekund od spuštění programu.

Vstupní parametry všech funkcí a kernelů jsou stejné

`int * a` – ukazatel na pole nad kterým se funkce provádí

`int N` - délka pole `a`

`int max` - polovina nejvyššího čísla v poli `a`

Zdrojový kód kernelu Prvocislo_GPU:

```
__global__ void Prvocislo_GPU(int *a, int N, int max)
{

  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  if (idx<N)
  {
  int i;
  int cislo = a[idx];
    for (i=2;i<(cislo<max?cislo:max);i++)
    {
      if (cislo%i == 0)
      {
      a[idx]=0;
      // break; //v případě Prvocislo_GPUopt
      }
    }
  }
}
```

Zdrojový kód funkce Prvocislo_CPU:

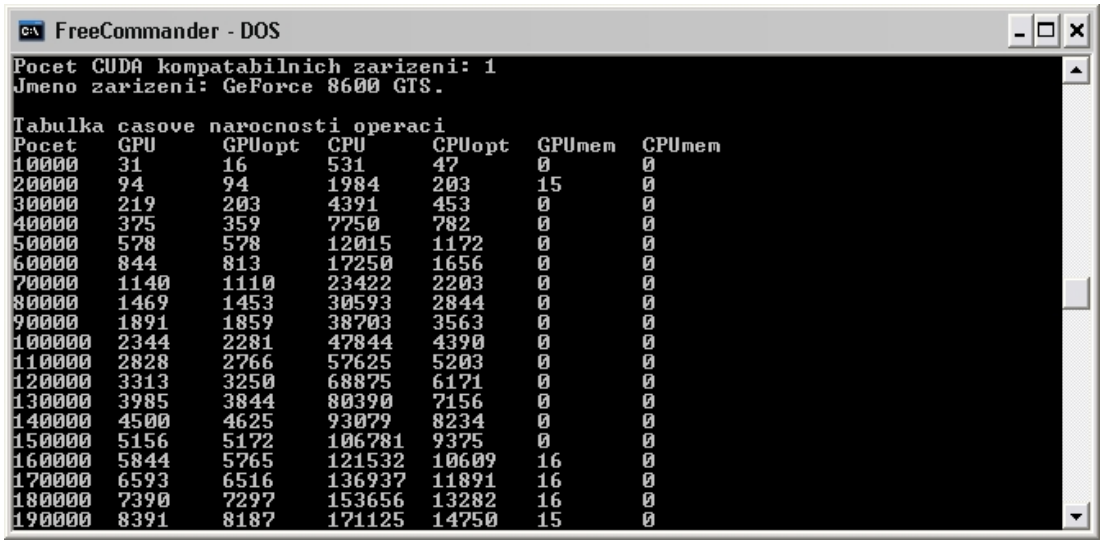
```
void Prvocislo_CPU(int * a_h, int N, int max)
{
int i;
int y;

    for (i=0;i<N;i++)
    {
    int cislo = a_h[i];
     for (y=2;y<max;y++)
      if (cislo%y == 0)
      {
        a_h[i]=0;
        // break; //v případě Prvocislo_CPUopt
      }
    }
}
```

6.1.2 Zhodnocení výsledku

Program byl testován na Nvidia GeForce 8600 GTS a GeForce 8800GT. Rychlost výpočtu (viz Tabulka 7) na GPU byla oproti CPU zhruba desetinásobná. Rozdíl mezi optimalizovanou a neoptimalizovanou verzí na GPU byl podle předpokladu zanedbatelný (přibližně 1%) zatímco optimalizovaná verze na procesoru proběhla přibližně 10x rychleji než funkce na CPU neoptimalizovaná. Rozdíl mezi rychlostí na GeForce 8600GTS a GeForce 8800GT je poměrně malý (zhruba 20%) ačkoliv 8800GT má 3,5x větší počet jader (112 oproti 32) – je to dáno tím, že kernel obsahuje dlouhý cyklus, jehož délka výpočtu závisí hlavně na frekvenci grafického čipu. Tato frekvence se u 8800GT liší právě o 20% oproti 8600GTS (viz Příloha 2).

Jako referenční CPU byl použit AMD Athlon 64X2 DualCore 2.61GHz.



Obrázek 14 – Běh programu Prvočísla na GeForce 8600GTS

	8600GTS		8800GT		CPU	
Interval	GPU[ms]	GPUopt[ms]	GPU[ms]	GPUopt[ms]	CPU[ms]	CPUopt[ms]
10000	31	16	31	31	531	47
20000	94	94	78	94	1984	203
50000	578	578	499	515	12015	1172
100000	2344	2281	1997	1950	47844	4390
150000	5156	5172	4415	4352	106781	9375
190000	8391	8187	7035	6942	171125	14750

Tabulka 7 - Výpočetní čas programu Prvočísla na GeForce 8600GTS a 8800GT ve srovnání s časem na CPU

6.2 REDUKCE ŠUMU V OBRAZE

Redukce či vyhlazování šumu vychází z předpokladu, že se hodnoty okolních pixelů liší především kvůli šumu.

Vyhlazování je výpočetně náročná úloha, při které se pro každý obrazový bod prochází jeho okolí a podle zadané konvoluční masky (Obrázek 15) přiděluje různé hodnoty důležitosti pro následný vážený průměr.

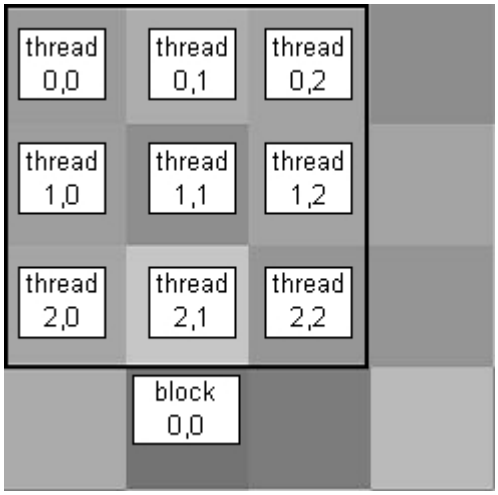
Tato úloha při lineárním zpracování znamená jeden průchod nad všemi body obrazu a pro každý bod obrazu průchod okolních bodů podle velikosti konvoluční masky - násobení hodnot pixelů s váhou masky a následné vážené průměrování.

Na grafickém procesoru jsem pro každý obrazový bod vyčlenil jeden blok vláken - počet vláken v jednom bloku je roven velikosti masky (maska je lineární pole reprezentující matici N x N viz Obrázek 16). Každé vlákno načte příslušnou hodnotu okolního pixelu a vynásobí ji váhou masky. Sečtení všech hodnot představuje na GPU složitější úlohu, neboť není možné, aby v jednom kroku do jednoho místa v paměti přičetla všechna vlákna své hodnoty. Bylo potřeba vytvořit sérii mezikroků (viz Obrázek 17), kdy poloviční počet vláken k hodnotě, kterou spravovala, přičetla hodnotu svého souseda. Potom čtvrtina vláken přičetla hodnotu výsledku předchozí operace a tento cyklus se opakoval až do doby, než v jediné buňce pole spravované prvním vláknem byl součet všech hodnot váženého průměru, ze kterého byl následně získán výsledek průměrování. Tyto operace se prováděly se sdílenou pamětí bloku, do které má grafická karta mnohonásobně rychlejší přístup než do globalní paměti (ze které byla data před začátkem operace načtena a po skončení navrácena).

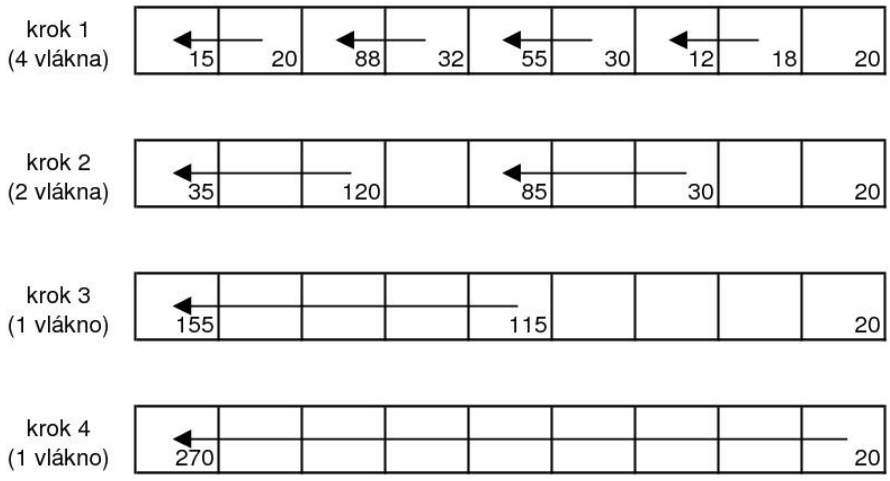
Redukce šumu způsobí rozmazání původního obrázku.

$$\begin{vmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{vmatrix} \qquad \begin{vmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{vmatrix} \qquad \begin{vmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{vmatrix}$$

Obrázek 15 – Příklad tvarů konvoluční masky



Obrázek 16 - Blok vláken pro zpracování okolí jednoho pixelu přes masku 3x3



Obrázek 17 – Příklad mezikroků při sečítání všech prvků matice 3x3

Je důležité si uvědomit, že barevný obraz je reprezentován třemi barevnými složkami RGB (červená, zelená, modrá), které jsou v poli dat, reprezentujícím obraz, řazeny za sebou. Tím jsou všechny operace s barevným, tříkanálovým obrazem třikrát výpočetně náročnější než s jednokanálovým, šedotónovým.

Podobným způsobem jako je redukce šumu je v programu realizována detekce hran v obraze - postup je velmi podobný jako při redukci šumu, jen podoba masky je odlišná (maska není váhami váženého průměru, ale představuje hodnoty sousedních pixelů, které je potřeba od sebe odečíst a podle velikosti rozdílu určit, jestli se jedná o skokovou změnu a tím pádem i o hranu). Příklad masky viz Obrázek 19.

Obrázek 18 - Rozmazání obrázku redukcí šumu maskou jedniček 9x9

6.2.1 Implementace

Program Redukce šumu pro filtrování šumu využívá pro načítání obrazu ze souboru knihovnu OpenCV. Knihovna OpenCV je sadou funkcí a tříd v jazyce C++, která implementuje velké množství algoritmů zpracování obrazu a počítačového vidění [18].

Program v sobě obsahuje deklaraci několika konvolučních masek – lineárních polí typu `integer` reprezentujících matici 3x3 až 11x11 (viz Obrázek 15). Součástí programu je funkce `cpu_filtr` a `gpu_filtr`, které mají stejné vstupně-výstupní vlastnosti. Parametry těchto funkcí jsou:

`IplImage * img` – ukazatel na strukturu obrazu knihovny OpenCV
`int filtr[]` – pole obsahující matici filtru
`int filtr_size` – šířka matice
`int num_iterations` – počet opakování aplikace filtru

přičemž funkce `cpu_filtr` se provede na CPU a `gpu_filtr` se vykoná na zařízení. Výsledkem je redukce šumu v obraze drženém ve struktuře `img`.

Podobným způsobem je vytvořena funkce `gpu_hrany`, která provede na obrázku detekci hran. Funkce se velmi podobá funkci `gpu_filtr` a má stejnou výpočetní náročnost (pro danou velikost filtru) a při srovnávacích testech nebyla použita. Pro detekci hrany se používá jiný typ konvoluční matice (Obrázek 19) jejíž součet všech prvků musí být roven nule.

Funkce se nacházejí v samostatném souboru, který je tak možné zkompilovat jako samostatnou knihovnu a přidat do jiné aplikace.

$$\begin{vmatrix} 3 & 3 & 3 \\ 3 & 0 & 3 \\ -5 & -5 & -5 \end{vmatrix}$$

Obrázek 19 – Kirschův operátor – Konvoluční matice pro detekci hran

6.2.2 Zhodnocení výsledku

Program Redukce šumu provedl redukci šumu na barevném obrázku o rozlišení 800x1143 pixelů za pomoci postupně se zvětšující masky filtru 3x3 až 11x11 (Příklad efektu rozmazání viz Obrázek 18). Program byl testován na grafických kartách GeForce 8600GTS, 8800GT a 9600GT. Výsledky jsou v Tabulka 8. Jako referenční CPU byl použit AMD Athlon 64X2 DualCore 2.61GHz.

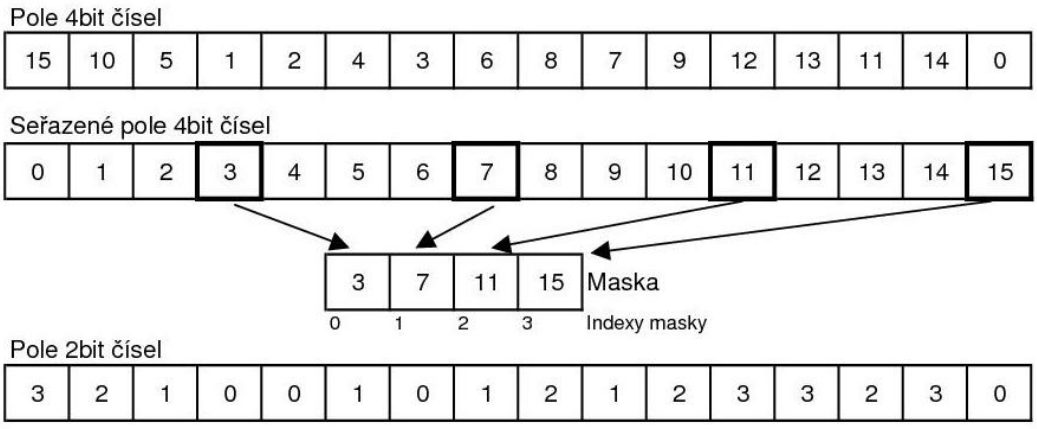
Velikost masky	8600GTS [ms]	8800GT [ms]	9600GT [ms]	CPU [ms]
3x3	757.5	201.6	357.2	939.5
5x5	932.3	262.9	428.8	1758.8
7x7	1495.2	449.9	684.2	2904.9
9x9	2270.3	652.9	1052.2	4351.3
11x11	2824.4	910.0	1312.5	6147.3

Tabulka 8 - Čas výpočtu 3 iterací redukce šumu na 3 kanálovém obrázku o velikosti 800x1143 pixelů při různých velikostech konvoluční masky

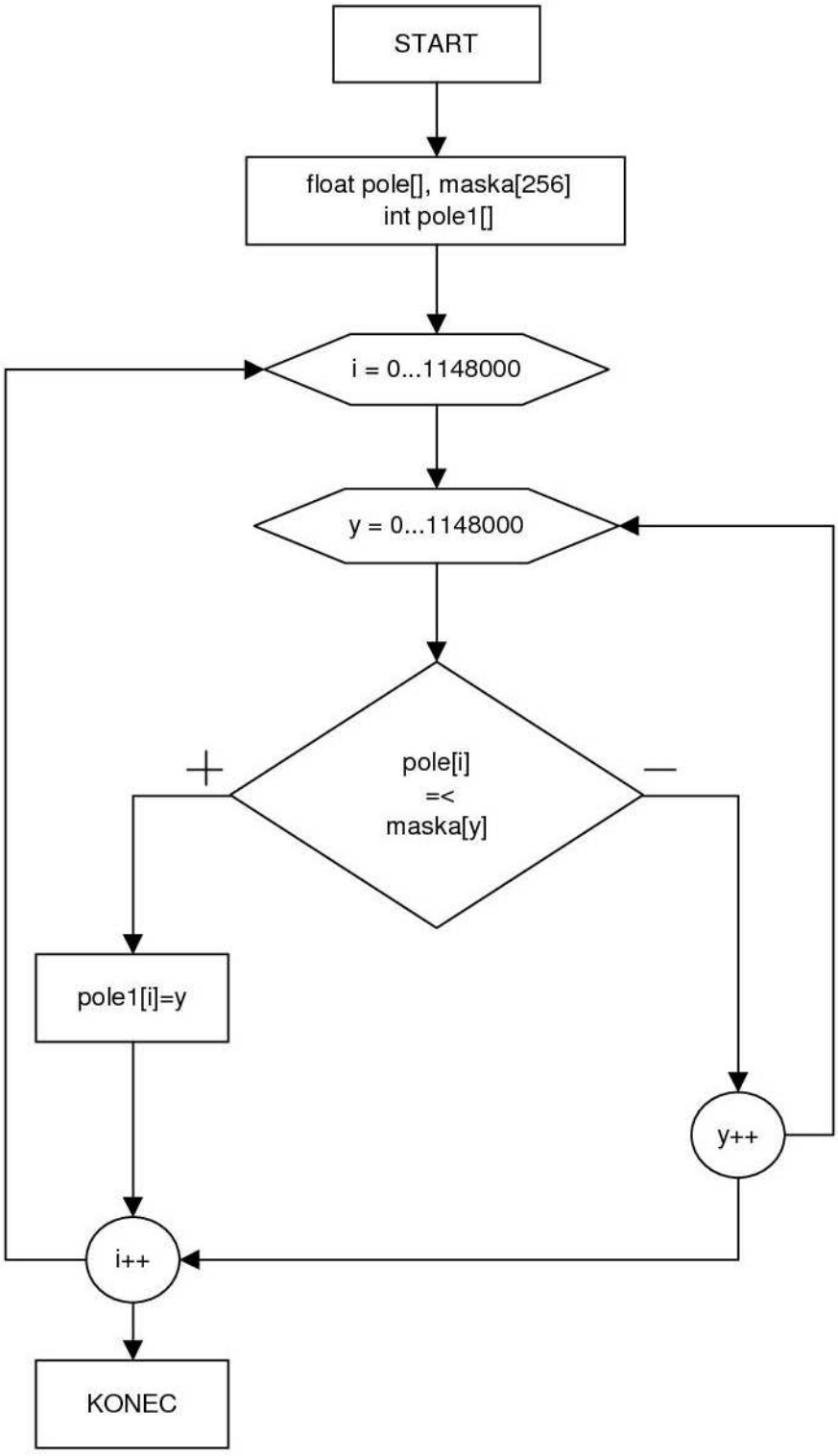
6.3 EKVALIZACE HISTOGRAMU HDR SNÍMKU

HDR snímky jsou snímky s vysokým dynamickým rozsahem – oproti klasickému obrazu, který používá 8 bitů na uložení jedné barvy, používají k uložení barev čísla s plovoucí desetinnou čárkou a je tedy prakticky neomezená.
HDR obrázky se vytvářejí složením několika různě exponovaných snímků.

Jednou funkcí při zpracování HDR snímku je převedení hodnot s velkým rozsahem držených v poli typu `float` do pole typu `char`. Pole reprezentující obraz je seřazeno, následně je z něj vybráno 255 hodnot tak, aby tyto hodnoty pole rozdělovaly na stejně velké intervaly (z pole délky 1148000 prvků se tedy vezme hodnota 4502., 9004.,... prvku) viz příklad ekvalizace pole šestnácti čtyřbitových čísel do pole šestnácti dvoubitových čísel na Obrázek 20. Funkce urychlená na GPU vezme původní pole reprezentující obraz a vytvořenou masku a cyklicky každý bod postupně porovnává s maskou, dokud hodnota v masce není větší než hodnota v poli - index masky se následně uloží do výstupního pole, které po skončení procedury obsahuje hodnoty ekvivalentní původnímu poli vhodné pro klasickou reprezentaci jako obraz – vývojový diagram lineárního průběhu funkce je na Obrázek 21. Vnější cyklus je na GPU reprezentován jednotlivými vlákny.



Obrázek 20 - Příklad ekvalizace pole čtyřbitových čísel do pole dvoubitových čísel



Obrázek 21 – Vývojový diagram funkce pro porovnání masky s polem hodnot

6.3.1 Implementace

Program Ekvalizace obsahuje funkce `Ekvalizace_CPU` a `Ekvalizace_GPU`. Mají stejné vstupní parametry a vykonávají algoritmus popsaný na Obrázek 21. Čas výpočtu měří CUDA časovač pomocí funkcí `cutStartTimer(unsigned int timer)` a `cutStopTimer(unsigned int timer)`. Po vykonání obou funkcí jsou výsledky navzájem porovnány a případné rozdíly jsou vypsány chybovým hlášením.

Podle [1] lze dosáhnout zvýšení rychlosti čtení z globální paměti grafické karty čtením po 32-bit, 64-bit nebo 128-bitových slovech. V programu je z tohoto důvodu vytvořena struktura `twinint` a `twinfloat`, kde se uvnitř kernelu `ekvalize_kernel` načtou dva prvky typu `float`, nebo uloží dva prvky typu `integer`.

Struktura twinfloat:

```
typedef struct __align__(8) {
    float a,b;
}twinfloat;
```

Protože se při výpočtu často přistupuje k hodnotám uloženým v masce a tato data se v průběhu kernelu nemění, je maska vytvořena jako sdílená konstanta.

6.3.2 Zhodnocení výsledku

Program Ekvalizace Histogramu byl testován na grafických kartách GeForce 8600GTS a 8800GT. Funkce pro ekvalizaci histogramu se provedlo nad polem o délce 10 328 064 prvků (reprezentuje šedotónový, jednokanálový obrázek 3936x2624 pixelů). Výsledky jsou v Tabulka 9.

8600GTS[ms]	8800GT[ms]	CPU[ms]
267.6	191.1	2335.8

Tabulka 9 - Porovnání doby výpočtu ekvalizace histogramu na GPU a CPU

7. ZÁVĚR

Seznámil jsem se s výpočetními prostředky současných grafických karet - jejich velký výpočetní výkon (Tabulka 1) a paralelní běh programu z nich dělají ideální hardware pro urychlení jednoduchých, avšak časově náročných algoritmů nad velkým objemem dat. Jejich velký výpočetní výkon a následný rozvoj vývojových nástrojů využívajících tento druh hardwaru vedl zpětně k vývoji specializovaných karet pro distribuované paralelní výpočty [12].

Grafický procesor (GPU) se při distribuci matematických výpočtů chová jako koprocesor, kterému hlavní program zadává data na zpracování, grafický procesor provede výpočet a hlavní program prováděný na běžném procesoru (CPU) si odebere výsledky. Program běžící na hostitelském zařízení nemusí čekat na konec výpočtu na GPU a může pokračovat dalšími instrukcemi a výsledky výpočtu převzít až ve chvíli, kdy je potřebuje.

Hlavní výhodou použití grafických procesorů je urychlení výpočtu díky velkému množství výpočetních jednotek integrovaných na grafické kartě. To umožňuje, aby jeden algoritmus zpracovával v jeden časový okamžik velké množství dat – v jednom multiprocesoru běží v jednu chvíli skupina 32 vláken tzv. warp. Ve warpu začínají všechna vlákna na stejné instrukční adrese, avšak mají volnost větvení a vykonávají se nezávisle. Zrychlení výpočtu tedy záleží převážně na tom, kolik multiprocesorů grafická karta obsahuje. Dalším důležitým parametrem, na kterém závisí rychlost výpočtu, je frekvence hodinového signálu multiprocesoru.

Hlavní nevýhodou programů vykonávaných na GPU je nutnost čekání všech vláken ve warpu dokud se nedokončí nejpomalejší z nich. Další nevýhodou je nutnost paralelního přístupu k specifikaci úloh a tím nemožnost řešení úloh, které jsou závislé na linearitě kroků programu např. sekvenční sečtení všech prvků pole, kde každý následující krok je závislý na předchozím mezivýsledku.

Jako vhodný nástroj pro implementaci algoritmů výpočtu na grafickém procesoru jsem zvolil platformu Nvidia CUDA pro její dobrou dostupnost, snadnou integraci do C++ a možnostem přístupu k CUDA kompatibilnímu hardwaru.

Za pomoci nástroje Nvidia CUDA jsem vytvořil program vyhledávající prvočísla v dlouhých intervalech.

Z výsledků (Tabulka 7 a Příloha 2) je patrno, že algoritmus běžící na GPU je přibližně 20x rychlejší (pro grafickou kartu GeForce 8600GTS) než identický algoritmus běžící na CPU. Protože však klasický program může například vyskočit z cyklu, pokud už jeho další provádění nemá smysl, lze provést optimalizaci výpočtu. Optimalizovaný program na procesoru potřeboval na výpočet pouze desetinu času neoptimalizované verze. Pokus o optimalizaci na GPU stejným způsobem však selhává, neboť je zde vždy vykonávána skupina 32 vláken a střední hodnota vzdálenosti dvou prvočísel na prohledávaném intervalu je 8, což způsobuje, že téměř vždy některé vlákno ve warpu muselo projít celý vyhledávací cyklus a tím blokovalo warp jako celek. Optimalizace výpočtu prvočísla na grafické kartě přinesla zrychlení výpočtu pouze o 1%.

Dalším krokem bylo otestování algoritmu na jiném GPU (GeForce 8800GT). Malý rozdíl mezi časy výpočtu karet GeForce 8600GTS a GeForce 8800GT je způsoben dlouhým cyklem obsaženým ve warpu, což má za následek, že hlavní roli na rychlost výpočtu měla frekvence hodinového signálu multiprocesoru.(Viz Příloha 1).

Dalším příkladem demonstrujícím výpočetní kapacitu grafických karet je filtrování šumu v obraze. Tato úloha demonstruje možnosti využití grafických karet při zpracování obrazu (tj. 2D signálu) a jejich možného využití například ve strojovém vidění.

Program prochází obraz a podle zadané konvoluční masky počítá vážený průměr okolních pixelů. Doba trvání výpočtu závisí na velikosti masky a na počtu jader grafické karty - GeForce 8800GT má 3,5x víc jader než GeForce 8600GTS a výpočet proběhl zhruba třikrát rychleji. Stejná funkce spuštěná na CPU oproti GeForce 8800GT trvala 4,6 – 6,7x pomaleji v závislosti na velikosti masky (pro větší masku byl rozdíl mezi GPU a CPU větší).

Tato úloha také ukazuje nutnost rozdělení čistě lineární úlohy (součet všech prvků pole) do několika mezikroků.

Posledním příkladem, který demonstruje další možnost využití při zpracování obrazu je ekvalizace histogramu HDR snímku. V této úloze je procházeno velké pole dat a cyklem se přiřazují hodnoty podle připravené masky.

Urychlení této úlohy je osminásobné pro kartu GeForce 8600GTS a dvanáctinásobné pro GeForce 8800GT oproti CPU.

Grafické karty tedy umožňují urychlit výpočty a zpracování velkého objemu dat, je však třeba zohlednit odlišnou filosofii i koncepci zařízení a přizpůsobit těmto faktům algoritmy. Některé způsoby optimalizace se na GPU se míjí efektem a existují zde další prvky zpomalující běh programu - např. zbytečně častý přístup do globální paměti zařízení. Pokud se podaří všechny tyto omezující podmínky vyřešit, výpočet trvá několikrát kratší dobu, než na běžném procesoru.

Dalším cílem práce by mohlo být zapracování funkcí pro zpracování obrazu na GPU do knihovny OpenCV a tím výrazné urychlení algoritmů strojového vidění implementovaných v této knihovně.

8. LITERATURA

[1] Nvidia cuda compute unified device architecture – Programing Guide version 2.1[online]. 8. 12. 2008 <http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.1.pdf>

[2] FARBER, Rob - CUDA, *Supercomputing for the Masses* [online]. 15. 4. 2008 <http://www.ddj.com/architect/207200659>

[3] Nvidia - GPU Gems [online], publikováno 2004, Datum poslední revize 15.7.2008 <http://developer.nvidia.com/object/gpu_gems_home.html>

[4] Nvidia – The CUDA Compiler Driver NVCC

[5] HUMBER, Andrew - *NVIDIA Adds OpenCL To Its Industry Leading GPU Computing Toolkit*[online]. 9. 12. 2008 <http://www.nvidia.com/object/io_1228825271885.html>

[6] BERGL, Marek – Řešení obecných úloh na grafických kartách, Praha, 2008. 69s. Diplomová práce na Českém vysokém učení technickém na fakultě elektrotechniky. Vedoucí diplomové práce Ivan Kratochvíl

[7] PECHOLT, Tomáš - Neuronová síť na grafickém procesoru, Praha, 2008. 67s. Diplomová práce na Českém vysokém učení technickém na fakultě elektrotechniky. Vedoucí diplomové práce Miroslav Skrbek

[8] KAŠPÁREK, Jaroslav - Využití grafického procesoru pro matematické výpočty, Praha, 2008. 55s. Diplomová práce na Českém vysokém učení technickém na fakultě elektrotechniky. Vedoucí diplomové práce Ivan Šimeček

[9] TRIOLET Damien - *Nvidia CUDA: preview* [online], 21.3.2007, < www.behardware.com/articles/659-1/nvidia-cuda-preview.html>

[10] Mike Thomas – Video Card Reviwes and Specifications, c2009 <http://www.gpureview.com/>

[11] DVOŘÁK, Václav – *Architektura procesorů*, 1. vydání, Brno: Vysoké učení technické v Brně nakladatelství VUTIUM, 1999. 303s ISBN: 80-213-1458-8

[12] *NVIDIA Tesla C1060 Computing Processor* [online]. c2009, [cit. 2009-05-28] <http://www.nvidia.com/object/product_tesla_c1060_us.html>

[13] *AMAX Information Technologies | NVIDIA® Tesla High Performance Computing (HPC) Solutions* [online]. c2009, [cit. 2009-05-28] <http://www.amax.com/CS_ServMaxPSC-2960-Core.asp>

[14] *C – Approved standarts* [online]. 10. 01. 2008 < http://www.open-std.org/jtc1/sc22/wg14/www/standards>

[15] OpenCL [online]. c2009 < http://www.khronos.org/opencl/>

[16] *AMD Developer Central* [online] < http://developer.amd.com/Pages/default.aspx>

[17] PETYOVSKÝ, Petr – *Reprezentace a vlastnosti obrazových dat*. Brno, Přednáška kurzu MPOV na Vysokém učení technickém fakultě elektrotechniky a komunikačních technologií.

[18] *OpenCV* [online]. 2009 < http://opencv.willowgarage.com/wiki/>

SEZNAM ZKRATEK A VÝRAZŮ

Zkratka/Symbol	Popis
Cache	Vyrovnávací paměť
CPU	Procesor na PC
CUDA	Compute unified device architecture
FLOPS	Počet operací s plovoucí desetinou čárkou za sekundu
GFLOPS	Miliarda operací s plovoucí desetinou čárkou za sekundu
GPU	Grafický procesor
Host	Myšlen PC nebo program vykonávaný na procesoru počítače
Kernel	Funkce prováděná na grafické kartě. Také existuje konvoluční kernel ve smyslu matice vah, avšak v tomto významu není toto slovo použito.
Konvoluční matice	Matice vah pro redukci šumu v obraze
Pipeline	Zřetězení instrukcí
Pixel	Nejmenší jednotka digitální rastrové grafiky
Stream	Součíst CUDA API umožňující vícenásobné spuštění kernelu
Texel	Nejmenší jednotka textury
TFLOPS	Bilión operací s plovoucí desetinou čárkou za sekundu
Zařízení	Myšlena grafická karta či grafický procesor

SEZNAM PŘÍLOH

Příloha 1

GeForce 8600 GTS

Major revision number:	1
Minor revision number:	1
Total amount of global memory:	268107776 bytes
Number of multiprocessors:	4
Number of cores:	32
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	16384 bytes
Total number of registers available per block:	8192
Warp size:	32
Maximum number of threads per block:	512
Maximum sizes of each dimension of a block:	512 x 512 x 64
Maximum sizes of each dimension of a grid:	65535 x 65535 x 1
Maximum memory pitch:	262144 bytes
Texture alignment:	256 bytes
Clock rate:	1.46 GHz
Concurrent copy and execution:	Yes

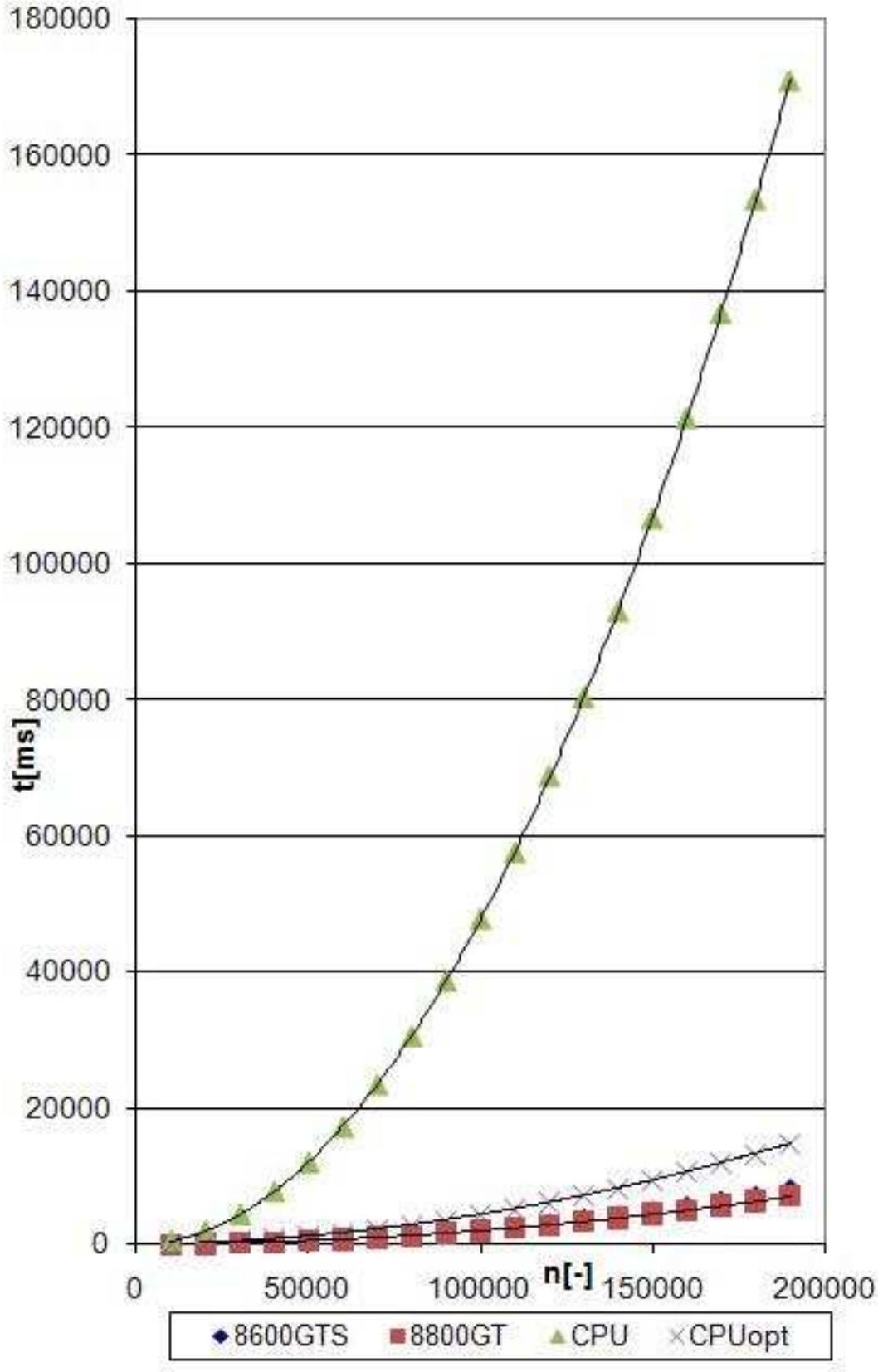
GeForce 8800 GT

Major revision number:	1
Minor revision number:	1
Total amount of global memory:	536870912 bytes
Number of multiprocessors:	14
Number of cores:	112
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	16384 bytes
Total number of registers available per block:	8192
Warp size:	32
Maximum number of threads per block:	512
Maximum sizes of each dimension of a block:	512 x 512 x 64
Maximum sizes of each dimension of a grid:	65535 x 65535 x 1
Maximum memory pitch:	262144 bytes
Texture alignment:	256 bytes
Clock rate:	1.71 GHz
Concurrent copy and execution:	No

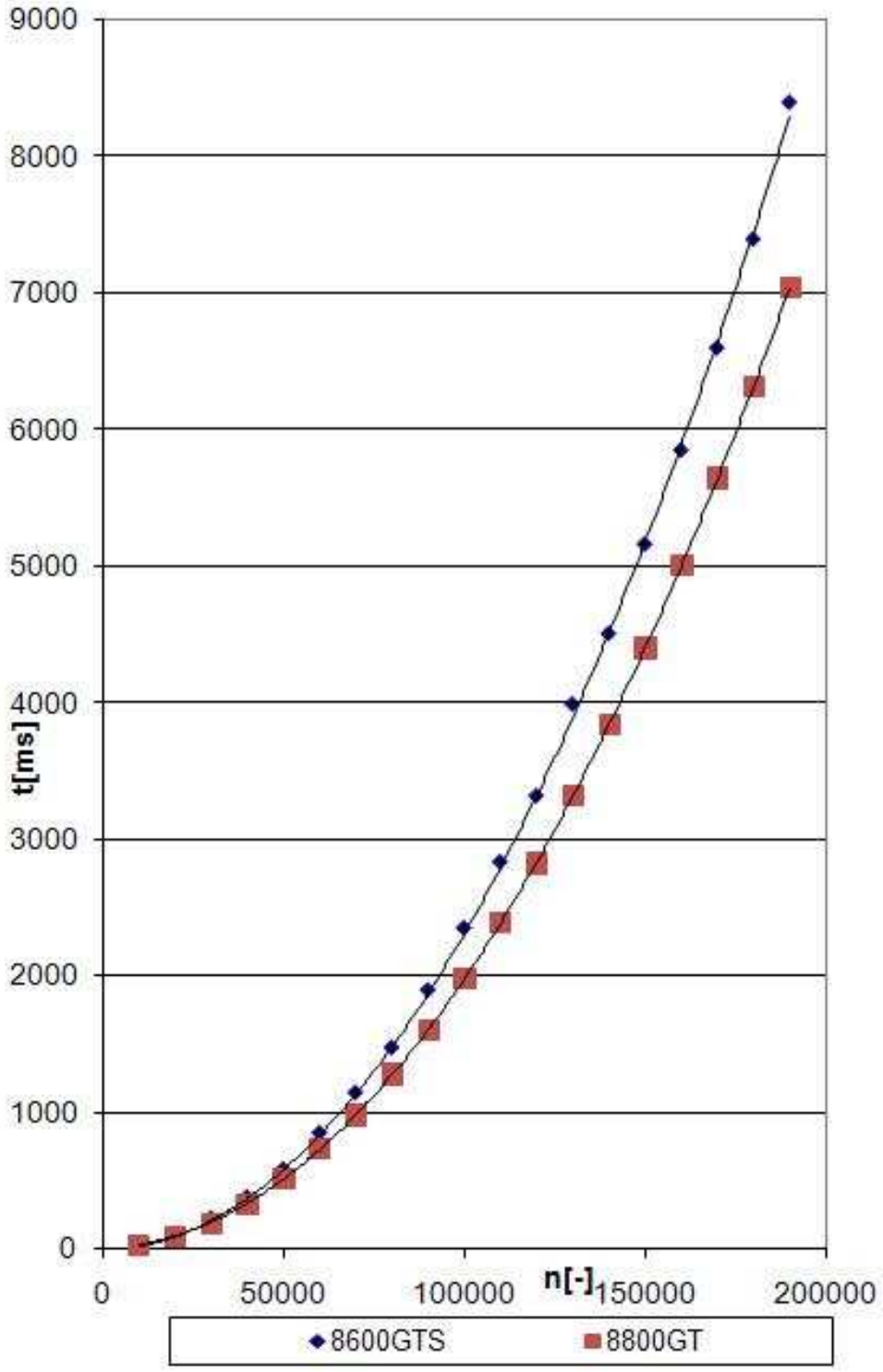
GeForce 9600 GT

Major revision number:	1
Minor revision number:	1
Total amount of global memory:	536870912 bytes
Number of multiprocessors:	8
Number of cores:	64
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	16384 bytes
Total number of registers available per block:	8192
Warp size:	32
Maximum number of threads per block:	512
Maximum sizes of each dimension of a block:	512 x 512 x 64
Maximum sizes of each dimension of a grid:	65535 x 65535 x 1
Maximum memory pitch:	262144 bytes
Texture alignment:	256 bytes
Clock rate:	1.60 GHz
Concurrent copy and execution:	Yes

Příloha 2

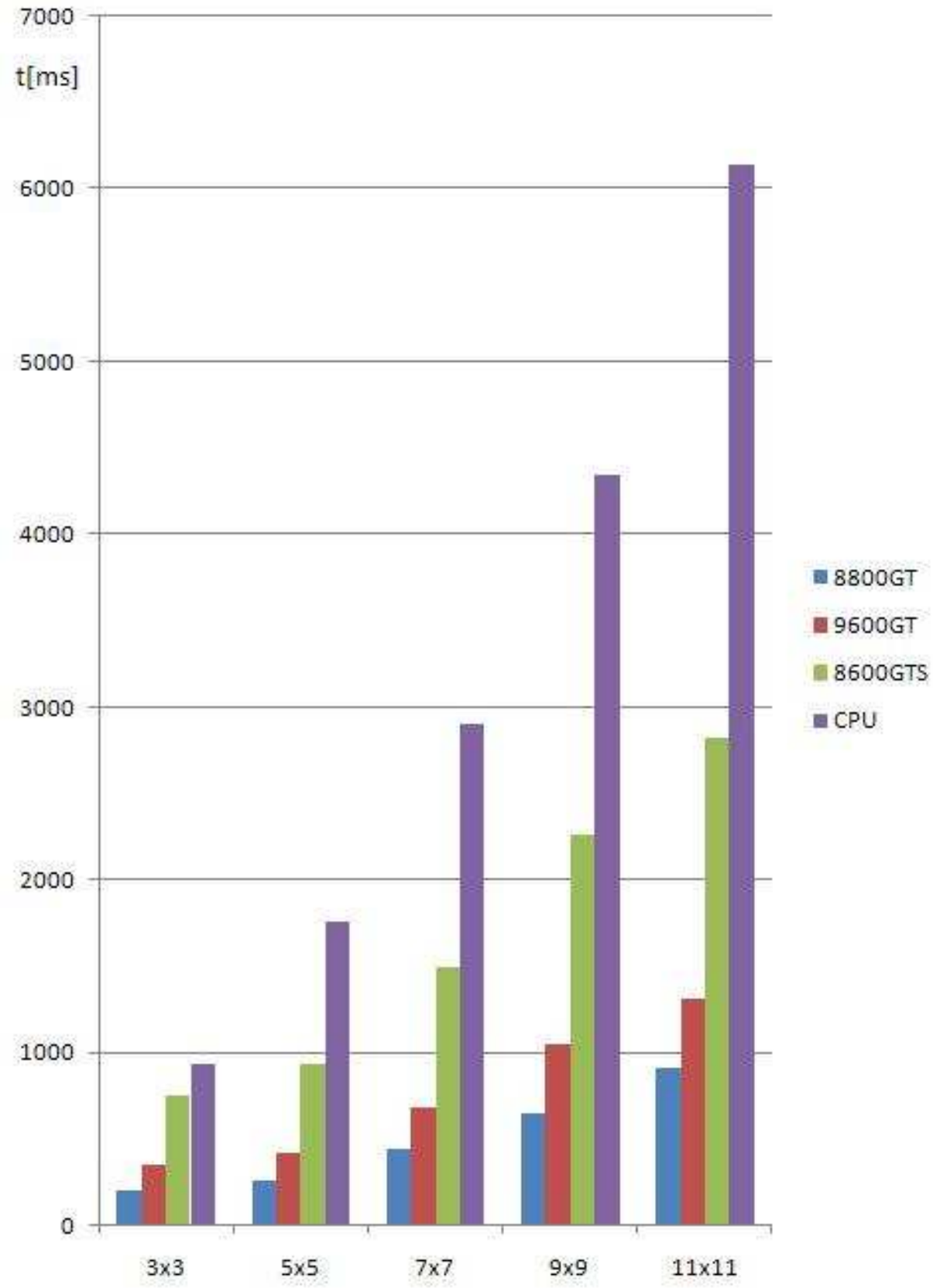


Graf časové závislosti algoritmu na počtu zpracovávaných prvků



Graf časové závislosti algoritmu na počtu zpracovávaných prvků pro GeForce 8600GTS a 8800GT

Příloha 3



Graf závislosti času výpočtu redukce šumu na velikosti konvoluční masky