TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: B2646 – Informační technologie
Studijní obor: Informační technologie

Využití technologie CUDA pro realizaci výpočtů

Using CUDA technology for the realization of calculations

Bakalářská práce

Autor: **Martin Peš**
Vedoucí práce: Ing. Přemysl Svoboda
Konzultant: Ing. Tomáš Martinec, Ph.D.

V Liberci 19. 5. 2011

Prohlášení

Byl(a) jsem seznámen(a) s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu TUL.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Bakalářskou práci jsem vypracoval(a) samostatně s použitím uvedené literatury a na základě konzultací s vedoucím bakalářské práce a konzultantem.

Datum

Podpis

Poděkování

Tímto bych chtěl poděkovat vedoucímu své bakalářské práce, Ing. Přemyslu Svobodovi za vedení, pomoc s prací a věnovaný čas.

Abstrakt

Tato práce se zabývá technologií CUDA, vytvořenou společností NVIDIA, za účelem poskytnout jednoduchý přístup k programování paralelních aplikací na současných grafických kartách ze svého portfolia. Cílem bylo porovnat její rychlost s dalšími programovacími jazyky Java, C++, C#, se zaměřením na výpočty. Nejprve tedy seznamuje se samotnou technologií a jejím využití v současnosti. Následně rozebírá práci v jazyce CUDA C, popisuje vývojové nástroje a testuje některé podstatné prvky, ovlivňující porovnávání. Také nastiňuje problémy přechodu ze sekvenčního na paralelní zpracování a omezení z toho plynoucí. Zdůvodňuje výběr konkrétních algoritmů, vysvětluje nutné úpravy pro jejich implementaci a porovnává naměřené časy. Celkovým výsledkem je porovnání rychlostí CUDA C s ostatními jazyky a sada programů pro testování rychlosti různých operací i algoritmů.

Klíčová slova

CUDA, NVIDIA, GPGPU, Java, C#, C++

Abstract

This work presents CUDA technology, created by NVIDIA in order to provide easy access to the programming of parallel applications on today's graphics cards from its portfolio. The main goal was to compare its speed in computations with other programming languages Java, C++ and C#. Firstly introduces the technology and its use in present time. Then analyzes CUDA C language, describes the development tools and tests some of the essential elements affecting the comparison. Also outlines the problems in transition from sequential to parallel processing and the resulting limitations. Justifies the choice of specific algorithms, explains the adjustments necessary for their implementation, and compares the measured times. Substantiates the specific algorithms selection, explains the adjustments necessary for their implementation, and compares the measured times. The overall result is the comparison of CUDA C language with other languages and a set of programs for testing the speed of various operations and algorithms.

Keywords

CUDA, GPGPU, NVIDIA, Java, C#, C++

Obsah

Seznam použitých pojmů a zkratek

ALU – Arithmetic Logic Unit

API – Application Programming Interface

CC – Compute Capability

CPU – Central Processing Unit

CUDA – Compute Unified Device Architecture

DRAM – Dynamic Random-Access Memory

ECC – Error Correcting Code

GK – Grafická karta

GPGPU – General-purpose computing on graphics processing units

GPU – Graphic Processing Unit

RAM – Random Access Memory

SDK – Software Development Kit

SIGGRAPH – Special Interest Group on GRAPHics and Interactive Techniques

SM – Streaming Multiprocessor

SSE – Streaming SIMD Extensions

VS – Microsoft Visual Studio

1. Úvod

CUDA je paralelně výpočetní architektura vyvinutá společností NVIDIA. Poskytuje přístup ke zpracování kódu grafickými jádry, kterých je několikanásobně více než jader u klasických procesorů. Díky jejich úzkému zaměření lze využít jejich výpočetní výkon pro zvýšení rychlosti některých algoritmů, nebo celých programů. Vzhledem k nízké ceně grafických karet a vysokému, stále stoupajícímu výkonu jsou čím dál tím více používány v početně náročných operacích, jako je zpracování obrazu, videa a různých signálů.

Pro programování se nejčastěji využívá CUDA C, což je původní jazyk C a části C++ s několika rozšířeními, existují ale i další nadstavby pro jiné platformy. Výrobce poskytuje velké množství nástrojů, ukázkových zdrojových kódů a rozsáhlou elektronickou dokumentaci. Elektronická příručka programování od výrobce se také ukázala jako nejspolehlivější zdroj informací, pro tvorbu této práce. Během vývoje programů bylo třeba řešit velké množství problémů, od specifikace toho, jak správně porovnávat algoritmy a co do srovnání zahrnout až po konkrétní převod ze sekvenčního na paralelní zpracování úloh. Porovnání byla zaměřena spíše na jednodušší algoritmy, které měly nalézt silné a slabé stránky technologie CUDA.

Přes zásadní rozdíly programování na CPU a GPU se tato práce snaží poskytnout náhled na rychlostní srovnání mezi jazyky C++, C#, Java pro klasický procesor a CUDA C pro grafickou kartu.

2. Technologie CUDA

2.1. Historie

Hlavním účelem grafických karet bylo a stále zůstává zpracování 3D prostoru a zobrazování obrazu. Díky jejich stoupajícímu výkonu se ale postupně prosazovaly i jinde – v obecných výpočtech. Pro tyto účely se původně musely algoritmy uzpůsobit tak, aby využívaly grafické operace, což bylo značně omezující. V tuto dobu se užívalo OpenGl nebo Direct3D, data se ukládala jako textury a grafická aplikace samotná byl chtěný algoritmus. V praxi toto řešení bylo složité a nepřehledné, výsledky však přinášelo a pozitivně se podepisovalo na rychlosti.

S tím jak postupně grafická karta přejímala funkce, které původně vykonával procesor, získávala na flexibilitě. Obzvláště programovatelné pixel shadery znamenaly velký skok kupředu, bylo možné zrychlit maticové počty. V roce 2003 byla na konferenci SIGGRAPH jedna celá sekce věnována výpočtům na GPU, pojmenována jako GPGPU (General-purpose computing on graphics processing units), což je zkratka označující toto odvětví dodnes [1].

Na Stanfordské univerzitě v té době vyvíjeli kompilátor jazyka Brook, určeného na zpracování dat paralelně, pro možnost spuštění na GPU, výsledkem byl BrookGPU. Přestože nedokončený do finální verze, prvky pozdější CUDA jsou patrné už zde. Hlavní vývojář Ian Buck poté práci na nové technologii pro GPGPU také započal a od roku 2005 vyvíjel CUDA (Compute Unified Device Architecture), za výrazné pomoci od Nvidie, která ho následně zaměstnala.

Uvedena byla s novou architekturou grafických karet, označovanou jako G80. Ta přinesla velké změny, z původně specificky určených pixel a vertex jednotek se staly jednotky unifikované, tedy schopné vykonávat různé operace dle potřeby aplikace, nazvané CUDA procesory. Hlavním záměrem bylo poskytnout programátorům paralelní architekturu tak, aby ji mohli jednoduše využít a zároveň propojení klasické aplikace s GPU působilo celistvě.

V roce 2007 bylo vypuštěno CUDA SDK a další nástroje verze 1.0, umožňující programovat na nových GPU. Od této chvíle vychází nové verze téměř pravidelně každý rok a další generace grafických karet přináší další možnosti a vylepšení. Vývoj v této oblasti jde velmi rychle kupředu. V současné době se již testuje čtvrtá verze SDK,

od které se očekává další zjednodušení přístupu do paměti, a automatické rozdělení práce mezi několika grafickými kartami, které doposud musel řešit uživatel.

2.2. Architektura a vývoj

Mezi architekturou CPU a GPU je mnoho rozdílů, klasický procesor byl od počátku navržen pro sekvenční zpracování, na rozdíl od paralelního u GPU, které mělo za úkol zpracovávat velké množství pixelů. Toto je promítnuto v samotném hardwarovém návrhu. U procesoru jsou dominantní obvody pro řízení procesoru, predikce větvení, paměť cache apod., ALU je zde v menšině. Grafický procesor naopak má velké množství samostatných jednotek, sdružujících skupiny ALU, vlastní cache a řídící obvody [2].

Obr. 1: rozdíly v architektuře CPU a GPU

Verzi CUDA architektury označuje termín *compute capability* (dále CC), volně přeloženo jako schopnost výpočtů, nebo výpočetní způsobilost. Toto označení nemá nic společného s verzí vývojových nástrojů, jedná se o rozdíl v hardwarovém provedení. Všechny nové grafické karty Nvidie obsahují určité množství jednotek označených jako *Streaming Multiprocessor* (SM), jejich počet závisí na konkrétním modelu karty a architektuře.

Tab. 1: compute capability v závislosi na architektuře GPU

Compute capability	GPU Architektura
1.0	G80
1.1	G86, G84, G98, G96, G96b, G94, G94b, G92, G92b
1.2	GT218, GT216, GT215
1.3	GT200, GT200b
2.0	GF100, GF110
2.1	GF108, GF106, GF104, GF114, GF116

První CC 1.0 na kartách řady GeForce 8800 a Tesla 870 bylo obrovským posunem v GPGPU, přesto zde byla řada omezení, které řešily až další generace. Jedná se především o dvojnásobnou přesnost operací s plovoucí řádovou, podporu atomických operací a další.

Následující CC 1.1, 1.2 a 1.3, které s dalšími řadami GK přicházely, rozšiřovaly možnosti využití. Byla zajištěna zpětná kompatibilita, tedy aplikace fungující na původní verzi budou fungovat na všech následujících. Opačně tomu tak není, například atomické operace nelze provádět na verzi menší než 1.1.

Architektura označovaná jako Fermi s CC 2.0 výrazněji změnila hardware. Oproti předchozí verzi přibyla L2 cache a ECC kontrola paměti, zvýšil se počet registrů a velikosti pamětí uvnitř jednotlivých jednotek.

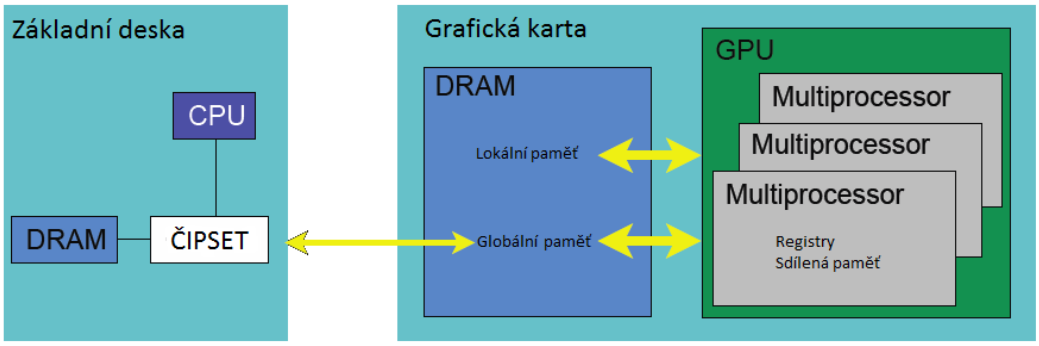
Následující verze CC 2.1 je evolucí Fermi, původní počet 32 CUDA jader příslušících každému SM se zvýšil na 48. Ty mají k dispozici 8 speciálně funkčních jednotek, provádějících některé matematické operace jako sinus a odmocninu, ty lze provést v jednom instrukčním cyklu.

Nově je také možné rozdělit lokální paměť o velikosti 64 kB u každého SM. Možnosti jsou ale pouze dvě, 16/48 kB nebo 48/16 kB mezi L1 cache nebo sdílenou paměť. Cache slouží pro uchovávání referencí z globální paměti, to je vhodné v případě, že se očekává častý nebo nepředpovídaný přístup do větších částí paměti. Oproti tomu sdílená paměť slouží k uchovávání dat mezi vlákny v bloku, čímž se dá zamezit příliš častému a pomalému přístupu jednotlivých vláken do globální paměti. Rozdíly v přístupu do sdílené a globální paměti společně s výkonem jsou řešeny v další části této práce.

Paměťový model architektury definuje několik pojmů:

- *Registry* – paměť přímo v každém SM, přístupná konkrétnímu vláknu po celou dobu jeho běhu.
- *Lokální paměť* – pokud není dostatek registrů, data se ukládají do globální paměti, ale stále pouze pro a po dobu běhu konkrétního vlákna.
- *Sdílená paměť* – paměť, kterou sdílí všechny vlákna ve stejném bloku, po ukončení práce celého bloku zaniká. Umístěna v jednotlivých SM.

- *Globální paměť* – paměť DRAM grafické karty, přístupná všem vláknům a procesoru. Uchovává data od alokace až po dealokaci.
- *Paměť textur a konstantní paměť* – paměť přístupná všem vláknům pouze pro čtení, je uložena v paměti DRAM.
- *Paměť DRAM CPU* – jedná se o označení paměti počítače, GPU do této paměti nemá přímý přístup.



Obr. 2: schéma rozložení paměti

Pokud využíváme grafickou kartu, na které chceme spouštět CUDA aplikace i pro zobrazování obrazu, musíme počítat s tím, že část grafické paměti si karta rezervuje. Pro rozlišení 1920x1080 se velikost použité paměti pohybuje kolem 150MB. Zobrazení má přednost vždy, grafický ovladač může uvolnit paměť, kterou aktuálně využíváme, pokud ji nemá dostatek. To pak vede obvykle k pádu aplikace nebo neočekávanému chování, problémy bývají také při změně rozlišení.

Možným faktorem omezující použitelnost CUDA může být obvykle menší velikost paměti na grafické kartě a její nerozšiřitelnost. Lépe jsou na tom specializované karty Tesla, které mají kolem 4 GB paměti [3], jsou však mnohem dražší, než klasické GK. Protože GPU nemůže do RAM počítače přímo přistupovat, je třeba data rozdělit a zpracovávat paměťově náročné úlohy po částech s kopírováním mezivýsledků do RAM. Tím je ale negativně ovlivněna rychlost. Data musí projít přes PCI-Express sběrnici, jejíž udávaná propustnost je 4 GB/s (8 GB/s obousměrně). Z toho plyne, že úzkým hrdlem výkonu aplikací na GPU bude právě tato sběrnice a nutnost kopírování dat pro další práci, konkrétní výsledky budou také řešeny v průběhu práce.

2.3. Využití

Vzhledem ke svému paralelnímu pojetí lze pomocí CUDA dosáhnout zrychlení určitých aplikací, především těch, které s velkým množstvím dat provádí značné množství výpočtů. Zatímco udávaný hrubý výkon nejsilnějších CPU je kolem

100 GFLOPS, některé GPU dosahují dnes jednotky TFLOPS, hlavní testovaná karta GeForce 460 GTX 768 MB v této práci má teoretický výkon 907.2 GFLOPS [4].

Možná reálná využití [5]:

- *Kódování a převod videa* – komplexní a výpočetně náročné operace s plovoucí desetinnou čárkou. Některé již vyvinuté aplikace umí využít CUDA a zrychlily několikanásobně převod videa.
- *Lékařský výzkum* – využití pro výpočty a zobrazování výsledků magnetické rezonance. Sledování, počítání rozložení prvků v krvi, vizualizace, animace a analýza organických molekul.
- *Zapojení do distribuovaných výpočtů* – GPU je možno zapojit do některého z projektů založených na distribuovaných výpočtech, zabývajících se mnoha odvětvími, jako je astrologie, kryptografie nebo biologie.
- *Sledování paprsků (raytracing)* – zrychlení metody vykreslování počítačové grafiky sledováním paprsků vycházejících z kamery, oproti klasickým přístupům.
- *Fyzikální výpočty* – možnost simulace téměř reálného fyzikálního chování objektů a kapalin.
- *Analýza obchodních trhů* – pro CUDA již existuje několik aplikací zabývajících se obchodními trhy, jejich vyhodnocováním a simulacemi chování.

Další možná použití jsou ve zpracování obrazu, signálu, zvuku, lineární algebře, ale také v mnoha menších programech, na jednotlivých částech algoritmů, které se dají paralelizovat. Na webových stránkách společnosti NVIDIA je nyní možno nalézt přes tisíc různých možných aplikací. Taktéž program MATLAB dokáže využívat CUDA [5].

Z dostupných informací lze usoudit, že se obvykle neporovnává výkon CUDA proti programovacím jazykům, ale rozdíl mezi programem určeným pro CPU a GPU. Srovnání rychlostí navíc s dalšími jazyky může poté působit zavádějícím dojmem, protože výkony jednotlivých kombinací grafické karty a procesoru se mohou velmi lišit.

3. Programování CUDA

3.1. API a definice

Existují dvě hlavní API pro programování CUDA aplikací. Obě založené na rozšíření jazyka C, často označovaného jako CUDA C.

- *CUDA Driver API* – nízkoúrovňová API, složitější na programování, ale poskytuje větší kontrolu nad celkovým kódem, je potřeba více řídící práce pro spuštění samotných výpočtů.
- *CUDA Runtime API* – postavená nad Driver API, zaobaluje a zjednodušuje některé funkce, umožňuje soustředit se více na samotné algoritmy a řešení úloh.

CUDA je možno programovat i v jiných jazycích například Python nebo Perl, ale za pomocí knihoven nebo různých vazeb na jazyk C, tyto vazby navíc obvykle obstarává třetí strana. Také je možno využít obecnějšího OpenCL které je nadstavbou nad CUDA a konkurenčním ATi Stream, nebo DirectCompute z balíku DirectX společnosti Microsoft.

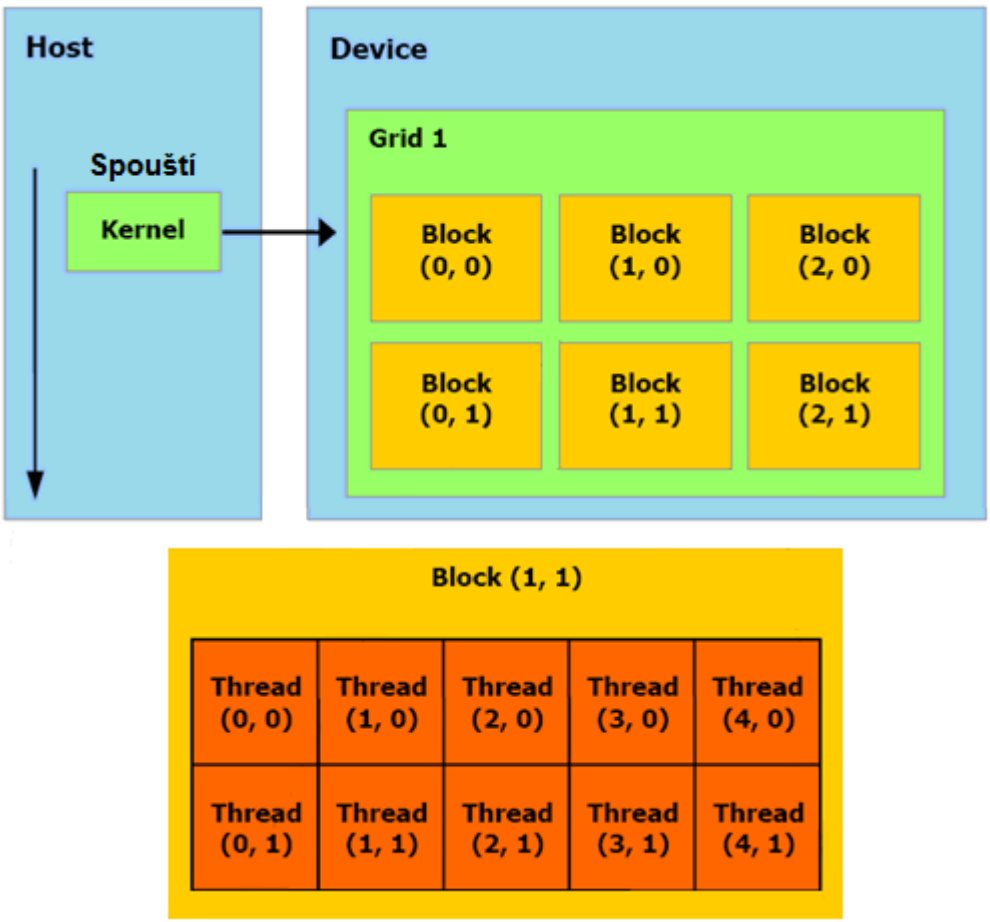
V této práci bylo užito CUDA Runtime API, protože se více blíží klasickému programování v C a jedná se zde primárně o porovnání rychlosti samotných algoritmů, než o nízkoúrovňovou funkcionalitu grafické karty. Konkrétně využitým jazykem je CUDA C.

Programovací model definuje některé základní pojmy, důležité pro pochopení problematiky:

- *Host* – označení pro CPU, nebo část, která se vykonává běžně bez využití grafické karty.
- *Device* – grafická karta s pamětí, někdy jen GPU.
- *Kernel* – část kódu, která je určena pro vykonávání na GPU, definuje se jako klasická funkce v jazyce C s různým prefixem.
- *Thread* – vlákno programu na GPU, jedná se v podstatě o instanci kernelu.
- *Block* – sdružení několika vláken, vykonávajících stejný kód.
- *Grid* – rozdělení bloků do trojrozměrné mřížky.
- *Warp* - skupina vláken, spouštěných naráz grafickým procesorem

Exekuční model vypadá následovně:

- Jeden blok vláken je zpracováván na jednom SM.
- Na jednom SM lze zpracovávat více bloků.
- Bloky jsou rozděleny do několika warpů.
- Vlákna ve stejném bloku mohou být synchronizována.



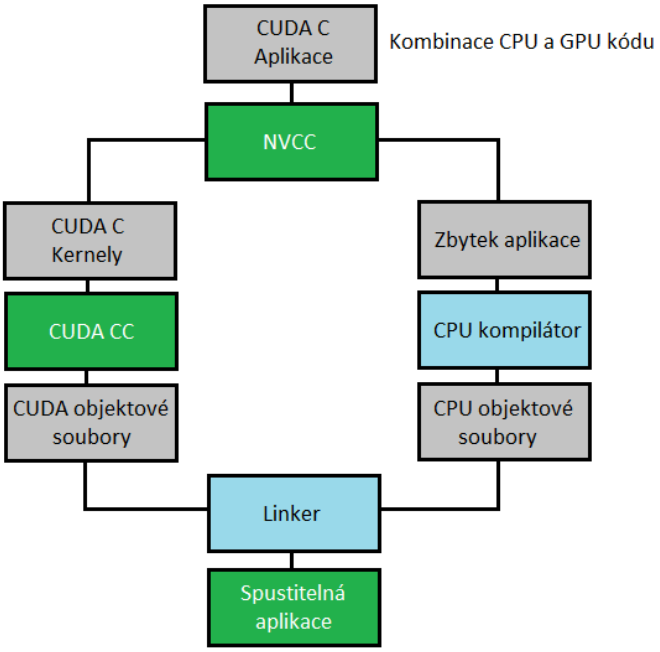
Obr. 3: schéma rozdělení vláken mřížky

3.2. Vývojové prostředí a nástroje

Pro správnou funkci programů je třeba mít nainstalovány některé z ovladačů, které obsahují potřebné funkce. K dispozici jsou i ovladače určené přímo pro vývoj, které podporují CUDA na všech grafických kartách, nejsou však nezbytné, protože novější klasické ovladače již mají vše potřebné. Ke spuštění aplikací na počítači bez vývojových nástrojů je také potřeba přiložit patřičné knihovny.

Potřebným prvkem pro tvorbu je *CUDA Toolkit*, obsahující základní knihovny a překladač kódu. Jeho součástí je také *Compute Visual Profiler*, nástroj pro profilaci kódu a zjištění vytíženosti grafické karty při běhu aplikace. Společně se instalují i některé příručky [2] [6] a dokumentace. Pro první seznámení je vhodné nainstalovat také *GPU Computing SDK code samples*, což je balík spustitelných příkladů a zdrojové kódy, ze které se dají rozšiřovat nebo využít pro vyzkoušení funkčnosti CUDA.

Nejpodstatnější složkou je *nvcc* překladač v balíku *CUDA Toolkit*, ten se stará o správné rozdělení a kompilaci kódu. Jeho funkce je blíže popsána na obrázku.



Obr. 4: postup kompilace programu

Samotný kód je možno psát v jakémkoliv vývojovém prostředí pro c nebo c++, pokud použijeme správný překladač.

Dalším z nástrojů je *NVIDIA Parallel Nsight*, určený pro integraci s Visual Studiem, kterému poskytuje přímou možnost ladění, profilace a analýzy jednotlivých vláken za běhu. Bohužel většina funkcí je omezena tím, že potřebují pro svou funkci druhou grafickou kartu, tedy na kartě co zároveň zobrazuje, nesmí být spuštěna naše aplikace. Je možné připojit Nsight k jinému PC a pracovat s kartou jeho. Také tento nástroj není dostupný volně, ale je nutné se registrovat jako vývojář CUDA aplikací a vlastnit Visual Studio 2010 Professional nebo vyšší [7]. Pro zmíněné obtíže jsem tento nástroj nepoužíval.

Naopak použitelným nástrojem se ukázal *Compute Visual Profiler*. Po výběru aplikace, se kterou chceme pracovat, proběhne několikrát její spuštění, při tom se zaznamenávají veškeré informace o vytížení, přenosech a přístupech v paměti. Výsledky lze tabulkově nebo v grafech zobrazit a dají nám důležité informace o tom, kde jsou slabá místa, případně porovnání jak dlouho trvají různé akce. Tyto data můžeme porovnávat mezi několika verzemi aplikace a díky tomu optimalizovat její výkon.

Psaní kódu je výhodné například v již zmíněném Visual Studiu společnosti Microsoft, překládat se dá konkrétně ve verzi 2008, ale lze použít i 2010 pokud předchozí verze bude nainstalována také. VS 2010 bylo použito i pro tyto testy, protože se předpokládá, že nové SDK budou spolupracovat spíše s touto novější verzí.

Problémy nastaly hned v úvodu, bylo nutné dodržet přesný postup založení projektu a jeho nastavení v daném pořadí, jinak prostředí kód pro CUDA nerozeznalo, nebylo možné využít našeptávače a program se odmítal správně přeložit. Po krocích byl tedy postup takovýto:

- Založit nový konzolový projekt (Win32 Console Application)
- V nabídce Build Customizations vybrat položku CUDA 3.2 (Runtime API)
- Vytvořit ve zdrojových kódech soubor s příponou *.cu*
- V nastavení projektu, záložce general nastavit Platform toolset na v90
- U nastavení linkeru přidat *cudart.lib* a *cuda.lib* do Additional Dependencies

V tuto chvíli VS rozeznalo projekt jako CUDA C/C++ a po přidání hlavičkových souborů do zdrojového kódu bylo možné programovat se zapnutým našeptávačem a výsledný kód překládat. Ve vlastnostech projektu je mnoho dalších nastavení, které

mají vliv jak na výkon, tak na běh programu. Lze jej zkompilovat pro několik verzí CC naráz s tím, že se použije nejvyšší z těch, kterou grafická karta použije. Společně s direktivami preprocesoru jde napsat kód, který půjde přeložit pro všechny architektury, ale zároveň pokročilé funkce využije jen, pokud jsou na té konkrétní podporovány.

3.3. Struktura programu a programování v CUDA C

Definice kernelu, spustitelného na grafické kartě se uvozuje klíčovým slovem, *__global__*, nebo *__device__*, pokud tuto funkci chceme volat z již spuštěného kernelu na GPU. Následuje běžný zápis funkce jako v jazyce C, tedy návratová hodnota, název funkce a vstupní parametry. Návratová hodnota u globálního kernelu musí být vždy typu void.

```
__global__ void prvniKernel(int a, int b)
{
     //kód funkce
}
```

Obr. 5: příklad kernelu __global__

Volání kernelu se provádí za pomocí speciálního operátoru, tří ostrých závorek, kde uvnitř jsou rozměr mřížky a bloku. Dohromady určují, kolik vláken bude funkci provádět. Definice rozměrů se provádí pomocí proměnné typu dim3, která vychází z vektoru int3 a poskytuje trojrozměrnou strukturu, která se využije pro indexování jednotlivých vláken. Maximální rozměr konkrétních dimenzí a jejich součet určuje verze CC. V případě že používáme jen jeden rozměr, je možné psát jej přímo do volání kernelu.

```
int a = 1; int b = 2;
dim3 dimMrizka(16,16,1); //16x16 bloků
dim3 dimBlok(128,1,1); //128 vláken v bloku
//celkem 32768x proběhne prvniKernel
prvniKernel<<<dimMrizka,dimBlok>>>(a,b);
```

Obr. 6: volání kernelu

Při běhu každé kernel funkce jsou k dispozici přednastavené proměnné *gridDim*, *blockDim*, *blockIdx* a *threadIdx*, také typu dim3. Pomocí nich lze ve funkci určit, o které vlákno se jedná a pokud chceme například přistupovat do pole, lze toto číslo použít jako unikátní index.

```
dim3 dimMrizka(65535,1,1);
dim3 dimBlok(512,1,1);

zapisIndexu<<<dimMrizka,dimBlok>>>(d_a);

__global__ void zapisIndexu(int * pole)
{
     int index = threadIdx.x + blockIdx.x * blockDim.x;
     pole[index] = index;
}
//data zůstávají v paměti grafické karty
```

Obr. 7: využití dimenzí pro indexování pole

Na obr. 6 lze vidět možné využití, pokud bychom chtěli naplnit pole hodnotou indexu, tedy čísly od nuly až po rozměr pole - 1. Sekvenčně na CPU tuto úlohu řešíme cyklem a hodnoty zapisujeme postupně, po jedné. V paralelním přístupu využíváme toho, že hodnotu zapisuje každé vlákno a úloha by měla být hotova rychleji. Nevýhodou plynoucí z principu práce na GPU je to, že ačkoliv algoritmus již svou práci vykonal, data jsou uložena v paměti grafické karty. Pokud je chceme využívat dále, například zapsat do souboru, musíme je nejprve zkopírovat zpět do paměti RAM počítače. Tento přenos ale v mnoha případech zabere mnohem více času, než samotný užitečný algoritmus. Proto je důležité vykonávat s daty na GPU co nejvíce operací naráz a až když máme finální výsledek, tak jej zkopírovat zpět. Doba přenosu a výhodnost použití CUDA pro různý počet operací je diskutována v dalších částech práce.

```
float * h_a = (float *)malloc(8192*sizeof(float));
float * d_a;
cudaMalloc((void**)&d_a, 8192*sizeof(float));

//pole není nutno kopírovat na GPU, protože hodnoty
//přepisujeme vlastní konkrétní hodnotou
zapisIndexu<<<16,512>>>(d_a); //16 bloků s 512 vlákny

cudaMemcpy(h_a, d_a, 8192*sizeof(float),
cudaMemcpyDeviceToHost); //zkopírování dat zpět z GK
```

Obr. 8: alokace paměti a kopírování hodnot

Samotná alokace paměti se provádí podobně jako v jazyce C, nejprve alokujeme ukazatel, následně použitím *cudaMalloc()* alokujeme potřebný počet bajtů v paměti grafické karty. Pomocí další funkce *cudaMemcpy()* zajišťujeme zkopírování dat. Je nutné kopírovat vždy do alokované paměti o stejné velikosti. Čtvrtý parametr nám

určuje, zda se jedná o přenos z RAM počítače na grafickou kartu nebo naopak. Tato funkce také na rozdíl od kernelu, který po spuštění probíhá asynchronně, způsobuje synchronizaci GPU a CPU, tedy data nebudou zkopírována, než kernel ukončí veškerou svou práci a program se na této funkci zastaví do doby, než se všechna data přenesou. Stejného efektu lze dosáhnout i pomocí zavolání *cudaThreadSynchronize()*, což pouze čeká na ukončení vláken kernelu.

Použití sdílené paměti dosáhneme vytvořením proměnné, uvozené klíčovým slovem *__shared__*, tato rychlá pamět je však velikostně velmi omezená a nelze do ní ukládat příliš rozsáhlá pole dat. Využijeme jí spíše pro uchovávání jednotlivých hodnot mezivýsledků apod.

Pokud potřebujeme vlákna v rámci bloku synchronizovat, je možné toho docílit zapsáním *__syncthreads()*, v tu chvíli se čeká až se na toto místo dostanou všechna vlákna, poté se pokračuje v provádění instrukcí dále. Pokud bychom použili synchronizaci na nesprávném místě, například uvnitř podmínky, kam se nedostanou všechna vlákna, kód by nikdy nepokročil dál a pravděpodobně by nastal pád, nebo by aplikace přestala odpovídat.

V kódu kernelů musíme proti původnímu C nebo C++ dodržet některá pravidla. Nemožnost použít rekurzi (do CC verze 2.0) a deklarovat statické proměnné uvnitř funkce patří mezi ty hlavní. Dále nelze využít pointerů na funkce a v C++ nesmíme použít virtuální funkce. Musíme také dávat pozor na čtení a zápis jednotlivých vláken do stejných míst v paměti. Největší obtíží přesto zůstává přizpůsobování algoritmu pro paralelní běh, kdy se kód od původní verze pro CPU, může velmi lišit a zvolení optimálního rozložení počtu vláken a bloků pro konkrétní využití.

CUDA C definuje mnoho dalších funkcí, které mají zrychlit specifické úlohy, mezi nejpoužitelnější se řadí rychlé matematické funkce, které bývají podobně jako kernel uvozeny podtržítky. Patří mezi ně například *__sinf()* nebo *__sqrtf()*, vykonávající sinus, respektive odmocninu s určitou, předem danou přesností. Celý seznam funkcí lze nalézt v příručce NVIDIA CUDA C Programming Guide [2] volně dostupné na webových stránkách Nvidie. Také se v ní nachází všechny potřebné informace o poskytovaných datových typech a přesnosti jednotlivých funkcí.

4. Další porovnávané programovací jazyky

Z vybraných porovnávaných jazyků se jako nejrychlejší předpokládá C a C++, protože jsou tzv. hardwaru nejblíže. Přesto ale tomu tak nemusí být vždy, zvláště v jednoduchých numerických operacích a při využití rozšířených instrukčních sad. Velmi záleží na nastavení jednotlivých kompilátorů i na kompilátorech samotných, několik kompilátoru stejného jazyka může produkovat program s velkými rozdíly ve výkonu. V rozsáhlejších aplikacích by se pravděpodobně rozdíly mezi jazyky projevily víc. Ty však nebylo možné rozumným způsobem vyzkoušet, kvůli úzkému zaměření CUDA na paralelní výpočty.

4.1. C/C++

Protože CUDA je založena na rozšíření jazyka C, bylo s tímto jazykem možno provést nejpřesnější srovnání z hlediska využitých funkcí. Algoritmy byly psány ve Visual Studiu 2010 Professional a používal se jeho vlastní překladač. Vzhledem k tomu, k různým odlišnostem a mnoha druhům kompilátorů [8], nelze pokládat výsledky jako absolutní, ale vždy jen v odkazu na konkrétní překladač, nastavení a určitý hardware. Oním nastavením jsou myšleny hlavně různé optimalizace, které kód také dokáží zrychlit. Další věcí je použití rozšířených instrukčních sad jako SSE, x86-64 a další. Z jejich původně vypnutého využívání, byla ručně zapnuta podpora instrukcí SSE2, protože bez nich podávalo C velmi špatné výsledky. Pro potřeby této práce zůstala zbylá nastavení ponechána standardně a testovala se výsledná aplikace z režimu Release, který je určen pro finální kompilaci. Pro nástin možného zrychlení však byly vyzkoušeny i přínosy některých optimalizací a nastavení jako přesnost operací s plovoucí řádovou čárkou.

Doba běhu algoritmu se stopovala původně za pomocí funkce *ftime()* a ukládala se do struktury *timeb*. V pozdější fázi byla speciálně pro tato měření vytvořena třída *Casovac*, která dokáže měřit s vyšší přesností. Výsledný čas se vypočítává z časové známky v *QueryPerformanceCounter()* a frekvence z *QueryPerformanceFrequency()*. Díky podpoře C++ se tato třída využila i u CUDA C.

4.2. Java

Dalším porovnávaným jazykem je Java, interpretovaný, objektově orientovaný jazyk, fungující na tzv. platformě Java. Zdrojové kódy jsou volně k dispozici a je dále vyvíjena jako open source a je dostupná zdarma. Umožňuje využití aplikací, na různých

platformách a systémech, díky běhu programu ve virtuálním stroji Javy - Java Virtual Machine (JVM). Nevýhodou je pomalejší start programu, kvůli překladu původního mezikódu před spuštěním.

Algoritmy byly psané ve vývojovém prostředí Eclipse Helios [9] a za použití Java JDK 6 update 25. Hotový kód byl vyexportován jako JAR spustitelný soubor. Některé programy se neobešly bez parametru -Xmx1024m pro zvýšení maximálního možného využití paměti pro JVM na 1024 MB.

Měření času probíhalo pomocí funkce *System.nanoTime(),* jejíž rozdíl před a po provedení algoritmu dal výsledný čas běhu. Všechna měření se prováděla několikrát pro minimalizaci chyby.

4.3. C#

Posledním porovnávaným je C#, jednoduchý, moderní a mnohoúčelový programovací jazyk [10] vyvinutý společností Microsoft. Společně s platformou .NET Framework poskytuje vhodné podmínky pro vývoj desktopových i webových a mobilních aplikací. Přestože je navržen jako nenáročný na paměť a prostředky, nemá za cíl přímo konkurovat jazyku C a dalším, hardware bližším jazykům, které na tom s rychlostí budou vždy o něco lépe. Práce s prostředím i jazykem je velmi příjemná, ze všech testovaných bych jí hodnotil jako nejvíce uživatelsky přívětivou.

Algoritmy byly psané a překládané taktéž v prostředí Microsoft Visual Studia 2010. Využilo se platformy .NET Framework verze 4.0 s klientským profilem. Zapnutá zůstala optimalizace kódu a cílem byla architektura x86. Výsledkem je vždy konzolová aplikace provádějící daný algoritmus. Časy jsou postupně zapisovány do textového souboru.

Stopování doby běhu poskytuje objekt *System.Diagnostics.Stopwatch*, přesnost v milisekundách je pro porovnání dostačující. Veškeré testy mají možnost provést se několikrát během jednoho spuštění programu, pomocí prvního vstupního argumentu, který udává počet opakování.

5. Práce a měření na GPU

Než se přistoupilo k samotným algoritmům pro porovnávání rychlosti všech vybraných jazyků, bylo potřeba zjistit základní charakteristiky běhu programu na grafické kartě. Za pomocí CUDA bylo tedy provedeno několik praktických testů, pro zjištění toho, co může algoritmy a celkový běh ovlivňovat.

5.1. Propustnost paměti, kopírování dat a nastavení hodnot

Důležitým faktorem k posouzení, zda bude možné CUDA využít pro plánovanou aplikaci je propustnost paměti a její velikost. Ačkoliv použitá grafická karta dokáže využít rozhraní PCI-Express x16 2.0, na testované desce je pouze starší verze 1.0. Dle specifikací tedy bude mít teoretickou propustnost 4 GB/s (8 GB obousměrně) oproti až 8 GB/s (16 GB/s) možných. Pokud bychom chtěli například zpracovávat obraz o rozlišení 1920x1080 (tzv. Full HD), formátu RGB, bylo by potřeba přenášet zhruba 6,2 MB. Dle předpokladů by měl přenos trvat zhruba 1,4 ms, měření však ukázalo velký rozdíl.

Tab. 2: doba přenosu dat mezi pamětí počítače a grafickou kartou v ms

	32 MB	**64 MB**	**128 MB**	**256 MB**	**512 MB**	**1920x1080 RGB**
přenos na GK	30	61	120	246	486	6
přenos z GK	29	60	118	239	480	5
celkem	59	121	238	485	966	11

Z naměřených výsledků vyplývá, že reálně přenos trvá 5-6 ms, tedy asi 4x déle. Při zpracovávání Full HD videa o frekvenci 25 snímků za sekundu v reálném čase, by algoritmus, na zpracování jednotlivých obrazů, měl 35 ms. Rozmezí hodnot se u přenosu 512 MB dat pohybovalo mezi 480-492 ms, tedy je nutné vždy počítat s menší odchylkou, propustnost v tomto případě vychází na 1140 MB/s. Přenášení dat je velmi časově náročná operace a minimalizování přenosů by často mělo velký vliv na rychlost celé aplikace. Tyto nároky jsou velmi specifické pro každou hardwarovou konfiguraci a nelze je brát obecně.

Pro případy, kdy chceme zapisovat do vynulovaného pole, je možné pomocí funkce *cudaMemset* nastavit části paměti zvolenou, vždy stejnou hodnotu. Díky tomu není potřeba pole vytvářet na CPU a ušetříme čas, který by zabrala jeho inicializace

a následný přenos. Z testování vyplynulo, že na 512 MB datech celá operace zabere 8 ms, to je 60x méně času než pokud bychom pole kopírovali.

5.2. Sekvenční zpracování

Zajímavým testem je pokus se sekvenčním zpracováním algoritmu, jediným vláknem GPU. V tomto případě se dá použít stejná algoritmická funkce jako v klasickém C nebo C++, ačkoliv stále s omezeními. Pro tento test byl vybrán algoritmus na výpočet Fibonacciho posloupnosti. Tak je označována posloupnost přirozených čísel, začínající 0, 1 a kde každé další číslo je součtem předchozích dvou [11]. Pokud chceme spočítat hodnotu určitého čísla v posloupnosti, sčítáme dvě čísla, než se dostaneme na index čísla, které bylo zadáno. Vzhledem k tomu, že je potřeba znát výsledek předchozí operace (tedy předchozí dvě čísla po součtu), není možné tento způsob výpočtu rozumně paralelizovat. Výsledek se celý nevejde do žádného datového typu, ale pro toto porovnání stačí, pokud výstup funkce na CUDA bude roven výstupu z jazyka C.

Naměřené časy byly 31 ms pro C a 558 ms CUDA pro výpočet hodnoty 10 miliontého čísla, rozdíl je tedy markantní, u tohoto algoritmu osmnáctinásobný. Použití jednoho vlákna a sekvenčního algoritmu se ukázalo jako nevýhodné, jednotlivé CUDA procesory nejsou rychlejší než klasický procesor. Nicméně je dobré vědět, že jako krajní možnost, je i takovéto řešení možné.

5.3. Varianty paralelního řešení algoritmů

Několik variant převodu sekvenčního algoritmu na paralelní bylo vyzkoušeno na histogramu. Vstupem byly hodnoty 0 až 255, tedy hodnoty, které reálně mohou reprezentovat například snímek v odstínech šedi. V tomto případě lze použít datový typ *unsigned int8*, čímž zmenšíme počet přenášených bajtů. Sekvenční řešení je poměrně jednoduché, projít všechna data a inkrementovat pole histogramu na indexu, reprezentující načtenou hodnotu.

```
for(int i = 0; i < pocethodnot; i++)
{
     histogram[data[i]]++;
}
```

Obr. 9: sekvenční tvorba histogramu

Paralelních variant na CUDA se nabízí hned několik:

1) Spustit tolik vláken, kolik je dat. Každé vlákno řeší jednu hodnotu, zapisuje do globální paměti. Vzhledem k možné kolizi vláken při přístupu k inkriminované hodnotě, je nutno použít atomické funkce, které se starají o to, aby vlákna přistupovaly k hodnotám postupně. Algoritmus ale zpomalují.

```
__global__
void histogram1(unsigned __int8 * data, int * histogram)
{
     int index = threadIdx.x +  blockIdx.x * blockDim.x;
     atomicAdd(&histogram[data[index]], 1);
}
```

Obr. 10: paralelní tvorba histogramu - varianta 1

2) Spustit 256 vláken, každé vlákno zapisuje svou hodnotu do globální paměti, pokud na ní narazí. Tato varianta bude nejpomalejší, protože všechna vlákna prochází celé pole dat a často zapisují do globální paměti.

```
__global__
void histogram2(unsigned __int8 * data, int * histogram,
int pocethodnot)
{
     for (int i = 0; i < pocethodnot; i++)
     if (threadIdx.x == data[i]) histogram[threadIdx.x]++;
}
```

Obr. 11: paralelní tvorba histogramu - varianta 2

3) Spustit 256 vláken, každé vlákno zapisuje svou hodnotu do registru, pokud na ní narazí. Teprve po skončení cyklu zapíší vlákna výsledné číslo do globální paměti. Zrychlení oproti předchozí variantě je téměř dvojnásobné.

```
__global__
void histogram3(unsigned __int8 * data, int * histogram,
int pocethodnot)
{
     int pocet = 0;
     for (int i = 0; i < pocethodnot; i++)
          if (threadIdx.x == data[i]) pocet++;

     histogram[threadIdx.x] = pocet;
}
```

Obr. 12: paralelní tvorba histogramu - varianta 3

4) Spustit tolik vláken, kolik je dat. Každé vlákno obsluhuje jednu hodnotu a zapisuje do sdílené paměti pro blok. Vytvořeny jsou tedy menší histogramy, řešící úsek 256 hodnot. Teprve po skončení cyklu kopírují jednotlivá vlákna hodnoty do globální paměti. Je nutné zde využít synchronizace, aby výsledek nebyl zapsán dříve, než dokončí všechna vlákna zápis. Tato varianta je velmi rychlá. Nutnost použít atomických instrukcí je i zde.

```
__global__
void histogram4(unsigned __int8 * data, int * histogram)
{
     __shared__ int shistogram[256];
     int index = threadIdx.x +  blockIdx.x * blockDim.x;
     shistogram[threadIdx.x] = 0;
     __syncthreads();

     atomicAdd(&shistogram[data[index]], 1);
     __syncthreads();

     atomicAdd(&histogram[threadIdx.x],
               shistogram[threadIdx.x]);
}
```

Obr. 13: paralelní tvorba histogramu - varianta 4

5) Nejlepší varianta je úpravou předchozí, využití sdílené paměti se ukázalo jako správné. Jednotlivá vlákna mohou zpracovávat více hodnot než jednu, optimální se ukázalo až 64 hodnot řešených v jednom vláknu.

```
__global__
void histogram5(unsigned __int8 * data, int * histogram,
int pocethodnot)
{
     __shared__ int sharedhistogram[256];
     int index = threadIdx.x +  blockIdx.x * blockDim.x;
     sharedhistogram[threadIdx.x] = 0;
     __syncthreads();
     for(int i = 0; i < 64; i++)
     {
     atomicAdd(&sharedhistogram[data[index+(pocethodnot/64)
*i]], 1);
     }
     __syncthreads();
     atomicAdd(&histogram[threadIdx.x],
               sharedhistogram[threadIdx.x]);
}
```

Obr. 14: paralelní tvorba histogramu - varianta 5

Výsledná rychlost zpracování grafickou kartou závisí také na rozložení hodnot, kvůli častějšímu přístupu do stejné části paměti v případě nepoměrného rozložení. Doba zpracování stoupá lineárně s počtem dat.

Tab. 3: rychlost zpracování histogramu v závislosti na počtu dat – poměrné zastoupení hodnot

	16 mil.	64 mil.	256 mil.
CPU (ms)	23,49	93,40	373,91
Varianta 1 (ms)	14,67	58,43	233,39
Varianta 2 (ms)	2640,95	10597,40	42426,82
Varianta 3 (ms)	1425,58	5701,99	22807,92
Varianta 4 (ms)	3,15	11,80	46,97
Varianta 5 (ms)	1,76	6,74	26,50
Přenos dat (ms)	11,31	40,10	160,01

Tab. 4: rychlost zpracování histogramu v závislosti na počtu dat – nepoměrné zastoupení hodnot

	16 mil.	64 mil.	256 mil.
CPU (ms)	23,57	94,14	378,93
Varianta 1 (ms)	22,63	94,36	377,28
Varianta 2 (ms)	6161,81	24648,62	98592,89
Varianta 3 (ms)	1425,60	5702,08	22807,93
Varianta 4 (ms)	7,97	31,70	126,56
Varianta 5 (ms)	5,82	23,92	94,74
Přenos dat (ms)	11,31	40,10	160,01

Ke každé variantě je třeba připočítat přenos dat, pokud není histogram součástí nějakého rozsáhlejšího projektu a data nebyla načtena již dříve. Dle srovnání vychází varianty 2 a 3 jako nevhodné. Varianta 1 díky své jednoduchosti a obstojné rychlosti působí jako nejuniverzálnější řešení, lze také použít, pokud nebude dostatek sdílené paměti pro další varianty. Z hlediska algoritmu je nejrychlejší varianta 5, která správným poměrem počtu operací a zápisů do paměti dosahuje výsledků až 14x rychleji, respektive 4x v případě velmi nepoměrného zastoupení hodnot. Procesor se s velkými rozdíly v zastoupení vypořádával téměř stejně jako s podobným zastoupením, přesto ale byl pomalejší než GPU. Pokud bychom algoritmus optimalizovali na konkrétní případ, očekávali převahu některých hodnot apod., určitě by bylo možné algoritmus ještě více zrychlit.

5.4. Nastavení kompilátoru

Kompilátor umožňuje vstup mnoha různých parametrů, určujících chování překladače při zpracování. Ty zásadní, které by mohly ovlivnit běh algoritmu, byly také otestovány.

Optimalizace mají čtyři možná nastavení

- Disabled (/Od) – vypnuté optimalizace, používá se při ladění
- Minimize Size (/O1) – cílem je minimalizovat velikost programu
- Maximize Speed (/O2) – cílem je maximalizovat rychlost programu
- Full Optimization (/Ox) – kombinace předchozích dvou

Na rychlost zpracování histogramu (z předchozí podkapitoly) na GPU nemělo žádné z těchto nastavení vliv. CPU původně testované s parametrem */Ox*, bylo v rámci chyby měření stejně rychlé při /O1, kdy se velikost programu snížila přibližně o jeden kilobajt. /O2 nepřineslo změnu žádnou. Vypnutí optimalizací se projevilo negativně na rychlosti zpracování procesorem, doba se prodloužila dvojnásobně.

Parametr *use_fast_math*, zajišťuje použití méně přesných matematických operací, respektive převedení všech funkcí které to umožňují na jejich varianty s menší přesností, ale vyšší rychlostí. Program byl po kompilaci s tímto parametrem schopen vypočítat odmocninu a sinus zhruba 4x rychleji, celkový vliv závisí na konkrétních funkcích a jejich počtu.

Velmi důležité je určení cílové architektury a compute capability pomocí atributu *gencode*. Například *-gencode=arch=compute_10,code=\"sm_10,compute_10\"* označuje, že program je kompilován pro CC verze 1.0. Pokud se v kódu vyskytují funkce, které tato verze nepodporuje, nebude překlad úspěšný a zobrazí chybovou hlášku. Lze kompilovat naráz pro více architektur a po spuštění se použije ta nejvyšší, kterou grafická karta podporuje.

6. Realizované algoritmy a porovnání

Původní předpoklad o tom, že pro porovnání algoritmy postačí pouze přepsat v určitém jazyce, se ukázal jako nesprávný. Mezi jazyky pracující na CPU, výrazný rozdíl není, po provedení specifických úprav šlo použít velmi podobný kód. Důvodem k zamyšlení bylo použití objektových typů oproti primitivním, v Javě je časté použití objektových, protože se dají uložit do seznamů a množin, jsou však pomalejší. Pro objektivnější srovnání byly vybrány primitivní, protože velké množství algoritmů si vystačí s poli, kam jdou ukládat tyto typy. Srovnání je navíc objektivnější, protože ostatní jazyky používají spíše neobjektové varianty.

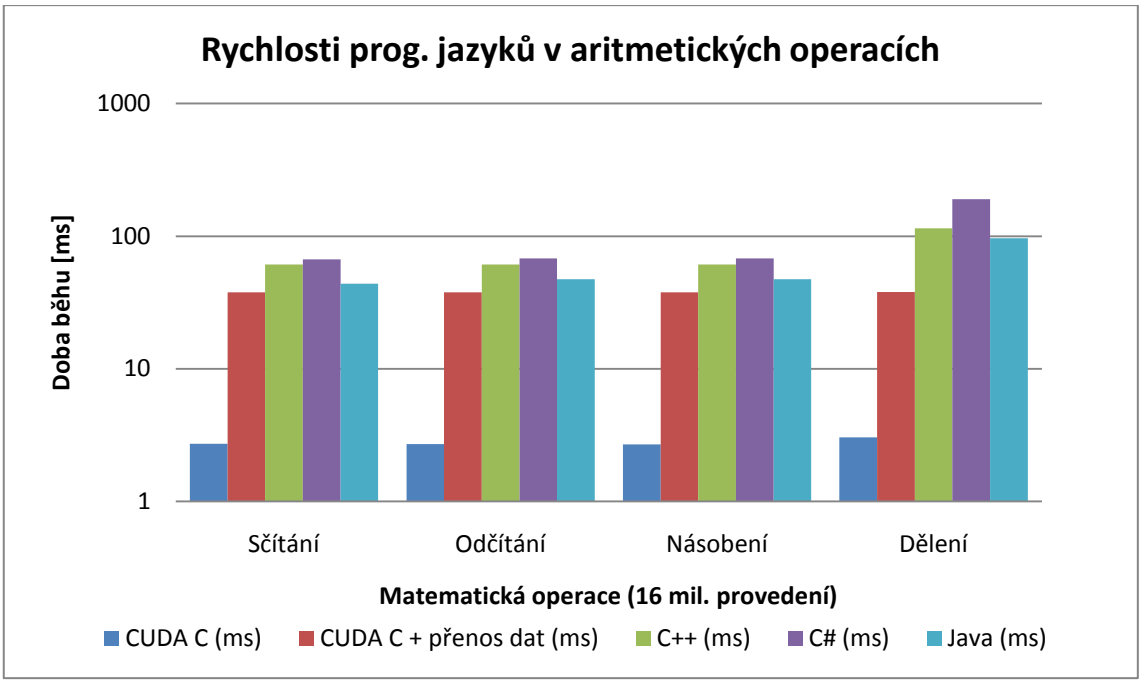
Pro paralelní zpracování na CUDA je negativním jevem jakékoliv větvení, protože všechna vlákna na sebe čekají, aby mohla pokračovat ve vykonávání společné části kódu. Nelze ale spoléhat na to, že ji provedou v setříděném pořadí, tedy vlákno s indexem 1 může dokončit operaci později než vlákno s indexem vyšším. To je zcela zásadní problém, pokud potřebujeme některé části realizovat postupně, jako například u mnoha druhů třídění. Vhodné algoritmy jsou tedy ty, kde nezávisí na předchozím stavu prvku, který zpracovává jiné vlákno a nejlépe pokud jednotlivá vlákna zapisují každé do jiné proměnné.

Problémem v samotném porovnání bylo hlavně rozdílné chování různých implementací algoritmu, kdy na určité sadě dat mohou být algoritmy rychlejší a na další pomalejší, jak bylo ukázáno již na příkladu s histogramem.

6.1. Aritmetické operace

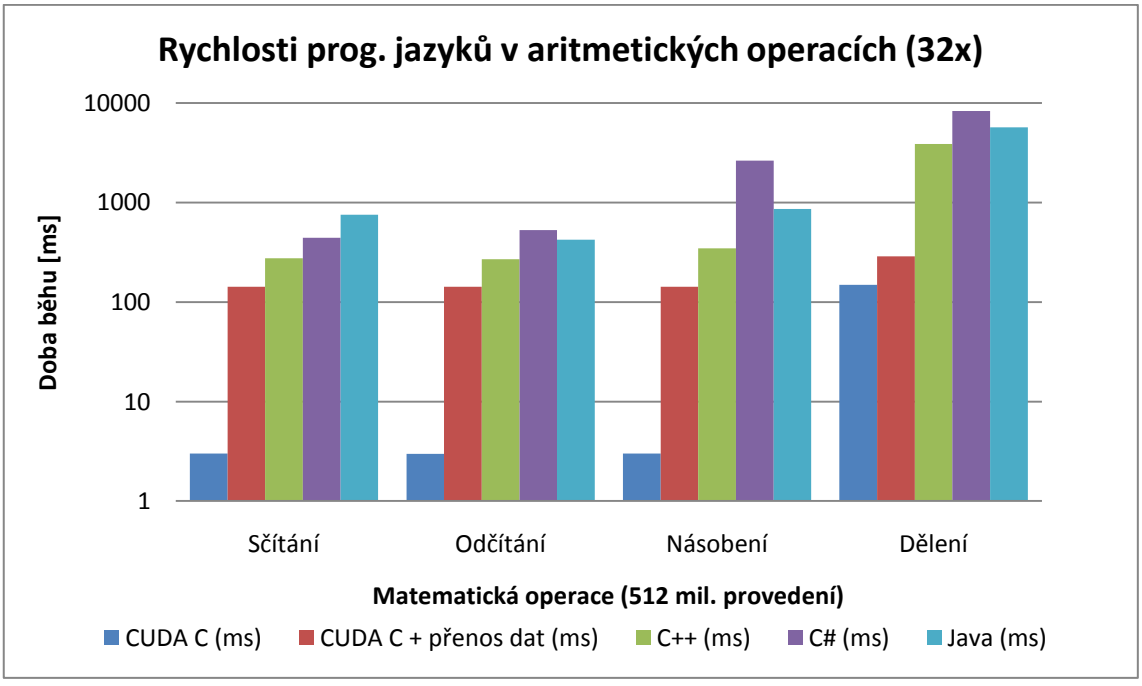
CUDA velmi rychle zvládá jednoduché matematické operace, které jsou základními prvky většiny algoritmů, ty se také staly prvním testem v porovnání všech jazyků. Testované byly základní operace: sčítaní, odčítání, násobení a dělení. Testovalo se na dvou polích o velikosti 16 mil. hodnot.

První graf zobrazuje použití jedné operace na každé hodnotě. Nejrychlejší CUDA C, ale kvůli malému počtu aritmetiky, je problém odlišit rychlost samotné funkce od režie obsluhy GPU a spouštění kernelu. Taktéž je nutno připočítat přenosy v paměti, pokud nejsou data již předem načtena. Za zmínku stojí o něco pomalejší výpočet dělení, což se projevilo u všech testovaných jazyků.



Graf 1: prog. jazyky v závislosti na aritmetické operaci provedené na každé hodnotě jednou

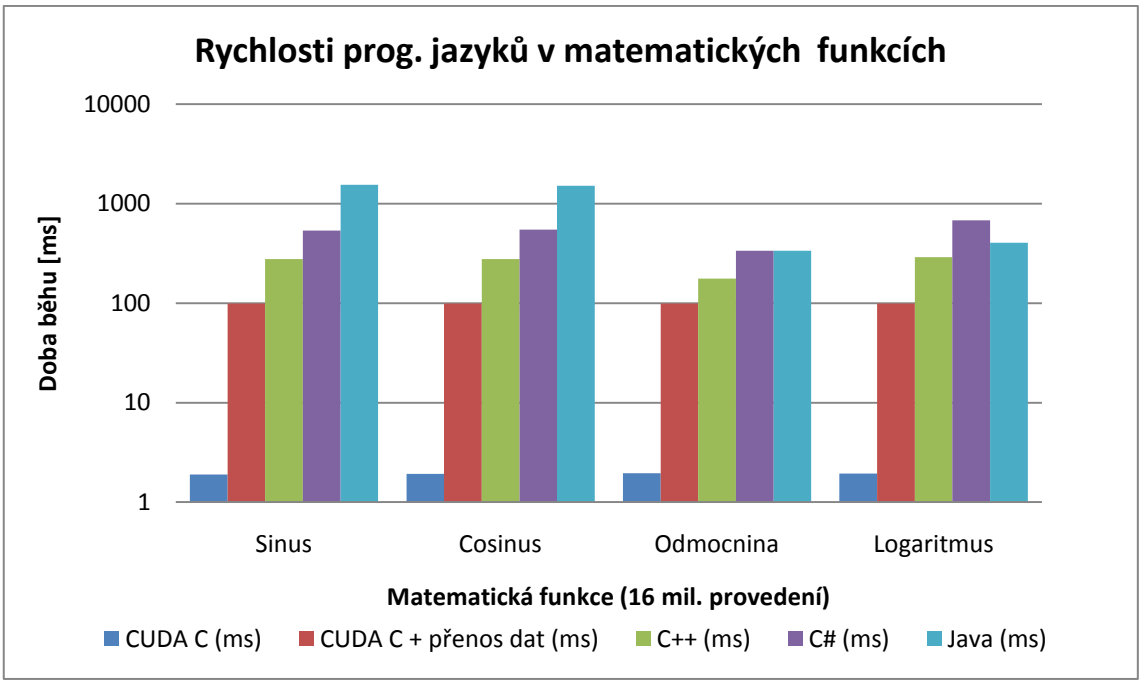
V druhém případě se každý výpočet provedl 32x, tedy dohromady jsou jednotlivé časy výsledkem 512 mil. výpočetních operací. V rychlosti už výrazně vede CUDA C, kterému stačí jednotky až desítky ms, oproti stovkám až tisícům u ostatních.



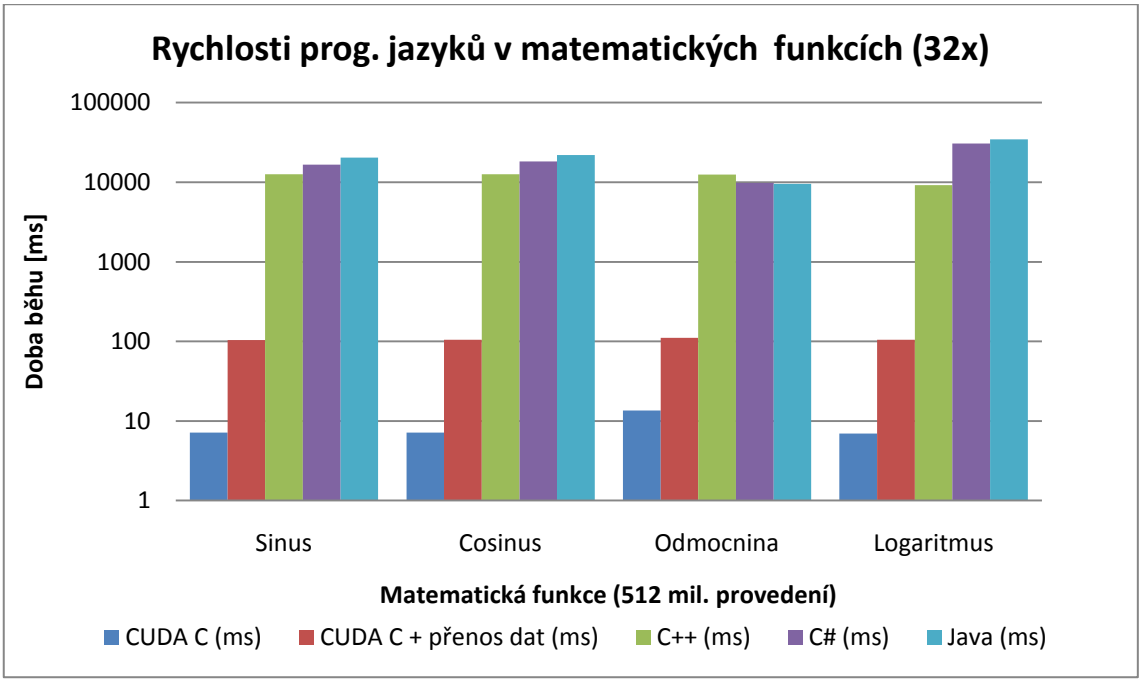
Graf 2: prog. jazyky v závislosti na aritmetické operaci provedené na každé hodnotě 32x

6.2. Goniometrické a matematické funkce

Další prvky často používané v algoritmech jsou goniometrické funkce, kvůli rozdílným implementacím se v tomto testu očekávaly větší rozdíly. Jelikož všechny jazyky řeší přesnost jinak, při ověření správnosti byla povolená odchylka ± 0,0001.



Graf 3: prog. jazyky v závislosti na matematické funkci provedené na každé hodnotě jednou

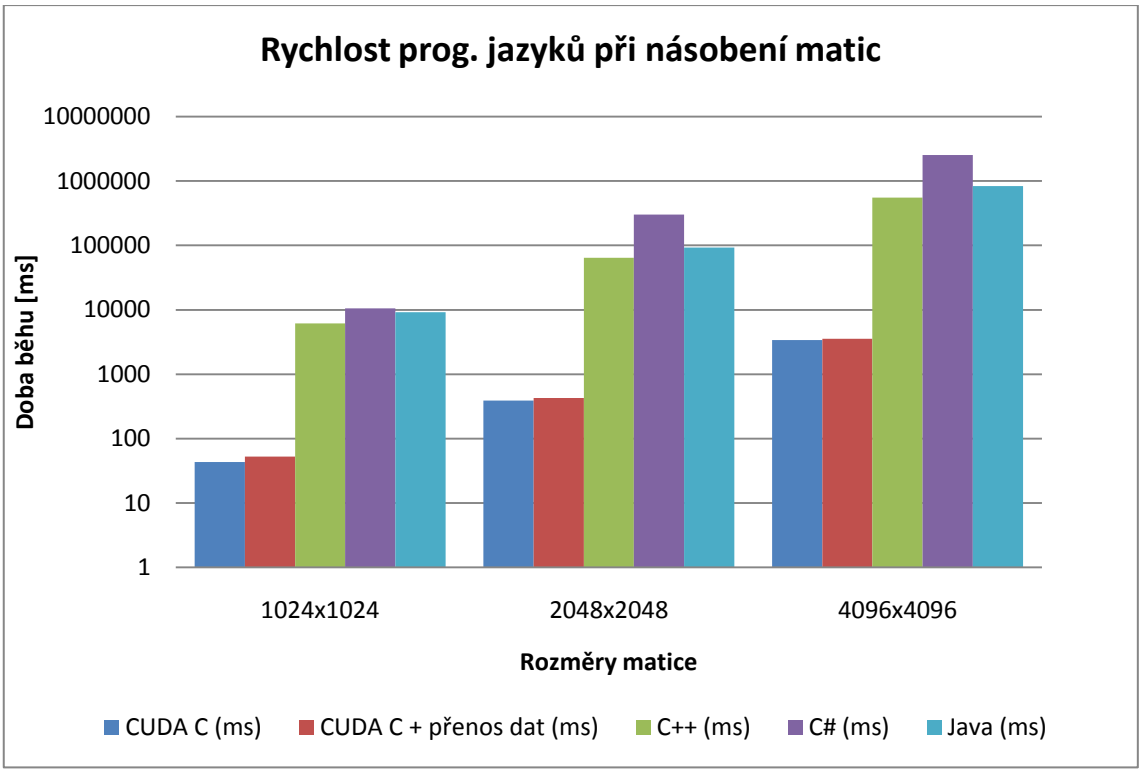


Graf 4: prog. jazyky v závislosti na matematické funkci provedené na každé hodnotě 32x

U všech otestovaných funkcí se vyplatí je provádět na GPU, i pokud provádíme jen jednu operaci na každé hodnotě.

6.3. Násobení matic

Násobení matic je početně náročná operace, kde každý prvek výsledné matice vzniká velkým počtem násobení hodnot původních matic. Pro svou početní intenzitu a nezávislost na ostatních počítaných prvcích, je vhodná právě pro paralelní zpracování, což potvrdil i test. Obzvláště u matic s většími rozměry lze pozorovat podstatné rozdíly. CUDA C tentokrát v celkovém srovnání nebude téměř vůbec omezena přenosem dat, protože výpočtů na každé vlákno je mnoho a přenášených dat naopak menší množství.

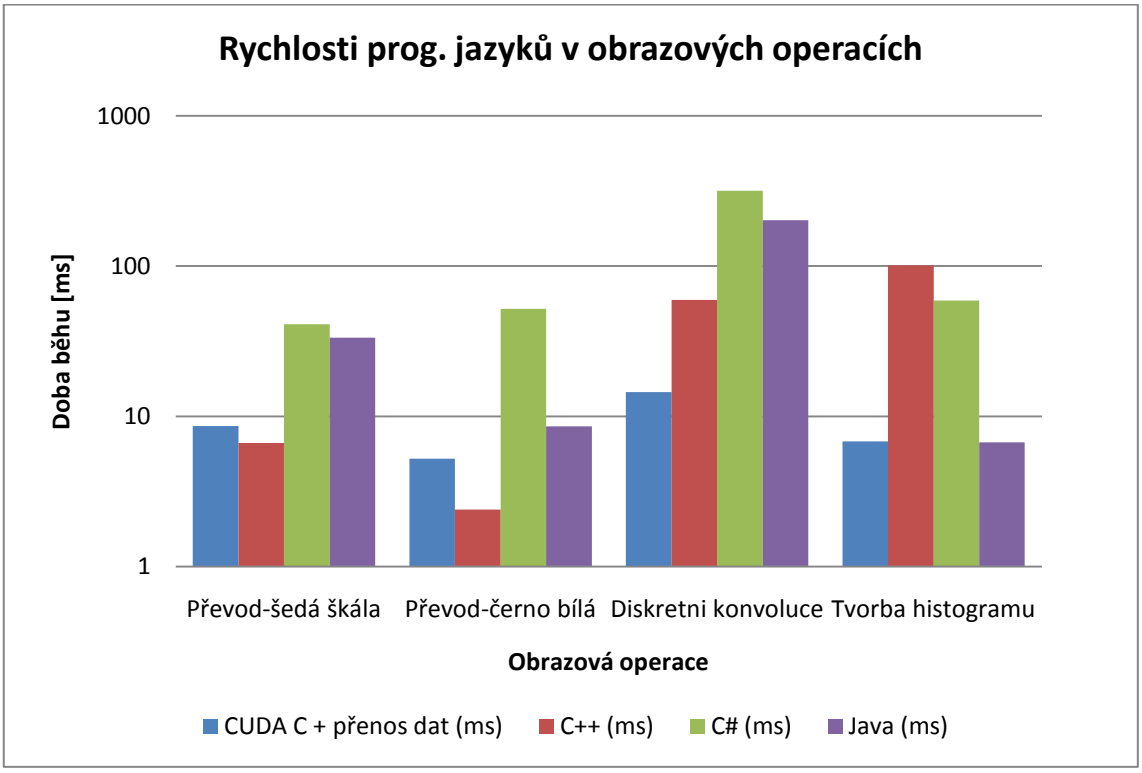


Graf 5: rychlost prog. jazyků při násobení matic

Použity byly čtvercové matice rozdílných velikostí. Zatímco většina jazyků s rostoucím počtem prvků zvyšovala svou dobu běhu rovnoměrně, u C# nastal neočekávaně velký skok. Při přechodu z matice o šířce 1024 na 2048 byl propad rychlosti třicetinásobný oproti desetinásobku u ostatních, po bližším zkoumání bylo zjištěno, že rozdíl mezi dobou výpočtu matice o šířce 2047 a 2048 je přes tři minuty. Následně již opět čas stoupal správně rovnoměrně. Příčinu tohoto jevu se nepodařilo v testovacím kódu odhalit, mohlo by se jednat o vlastnost jazyka C# při alokování paměti pro různé velikosti pole. Další možností je problém s velikostí cache procesoru.

6.4. Histogram a obrazové operace

Jako další srovnání bylo vybráno již na CUDA řešené tvoření histogramu a následně tvorba černobílého a šedotónového snímku z původně barevného. Nakonec ještě algoritmus provede rozostření pomocí konvoluce. Vstupem byl bitmapový snímek ve 24 bitové barevné hloubce a rozlišení 1920x1080.



Graf 6: rychlost prog. jazyků při provádění obrazových operací

Zde se opět výrazněji projevila nevýhoda programování na GPU, nutnost překopírovat data. Pouze pro převod obrazu do šedých odstínů nebo jiného barevného zpracování se CUDA C nevyplatí používat, pro malou náročnost na výpočetní výkon. Diskrétní konvoluce je naopak vhodná, jelikož se jedná v podstatě o násobení pixelu hodnotami matice a následné zprůměrování. Z ostatních jazyků nejvíce zajímavě působila Java, která byla nejrychlejší při tvorbě histogramu. Bohužel však práce s ní byla horši, protože neposkytuje datový typ *unsigned byte* [13], který byl použit pro reprezentaci jednotlivých barevných složek. Ačkoliv je tento problém řešitelný bitovým násobením a ukládáním do jiných datových typů, narušuje to komfort při programování, kterým Java jinak oplývá.

6.5. Celkové srovnání rychlosti

Ačkoliv již bylo řečeno, že posuzování rychlosti závisí na mnoha faktorech, následující celkové srovnání poskytuje náhled na to, co je možno očekávat, pokud uvažujeme o nasazení CUDA v praxi.

Tab. 5: poměrné zrychlení CUDA oproti nejrychlejšímu jazyku v dané operaci

	CUDA C	CUDA (vč. přenosu)
Sčítání 32 mil. hodnot	16,93x	0,32x
Sčítání 32 mil. hodnot (32x)	92,13x	1,92x
Odčítání 32 mil. hodnot	17,27x	0,32x
Odčítání 32 mil. hodnot (32x)	91,19x	1,88x
Násobení 32 mil. hodnot	17,20x	0,32x
Násobení 32 mil. hodnot (32x)	117,44x	2,42x
Dělení 32 mil. hodnot	32,60x	0,68x
Dělení 32 mil. hodnot (32x)	26,07x	13,43x
Sinus 16 mil. hodnot	145,76x	2,78x
Sinus 16 mil. hodnot (32x)	1765,48x	120,76x
Cosinus 16 mil. hodnot	143,68x	2,79x
Cosinus 16 mil. hodnot (32x)	1763,20x	120,25x
Odmocnina 16 mil. hodnot	90,54x	1,77x
Odmocnina 16 mil. hodnot (32x)	703,48x	85,61x
Logaritmus 16 mil. hodnot	150,06x	2,93x
Logaritmus 16 mil. hodnot (32x))	1323,74x	88,37x
Násobení matic 1024x1024	142,36x	116,76x
Násobení matic 2048x2048	164,61x	151,23x
Násobení matic 4092x4092	161,79x	155,55x
Převod obrazu do šedých od.	2,76x	0,77x
Převod do černo bílých od.	1,04x	0,46x
Diskrétní konvoluce (rozostření)	9,08x	4,11x
Tvorba histogramu	2,49x	0,99x

Lze prohlásit, že ke snížení rychlosti, nebo spíše k pomalejšímu zpracování dochází pouze tehdy, když je přenášeno velké množství dat a počet samotných výpočtů je zanedbatelný. Vždy se vyplatí používat CUDA pro násobení matic, jelikož zrychlení oproti klasickým jazykům je více jak stonásobné. Pokud bychom navrhovali použití v reálné aplikaci, určitě by bylo místo i pro další optimalizace, které by mohly běh dále zrychlit. Pro různé operace je možno využít i specializovaných knihoven, jež jsou součástí SDK. V testech použitá NVIDIA GeForce 460 GTX je pro výpočty vhodná, protože podporuje zatím nejnovější CC 2.1. Nebylo ale možné využít rozhraní PCI Express 2.0 pro rychlejší přenos dat, protože to neumožňovala základní deska.

Závěr

Cílem práce bylo seznámit se, naprogramovat a nakonec porovnat rychlosti programů provádějící vhodně vybrané algoritmy pomocí CUDA, konkrétně CUDA C s ostatními jazyky jako je C++, Java a C#.

Problémů, které programování na GPU přineslo je poměrně hodně a ty hlavní jsou v této práci rozebírány. Bylo nastíněno několik možností řešení převodu sekvenčního na paralelní algoritmus, výkon sekvenčního zpracování a použití specializovaných pamětí pro určité úlohy. Porovnání rychlosti je také velmi složité, nelze přímo hodnotou vyjádřit, o kolik je CUDA rychlejší nebo pomalejší. Výsledek bude vždy ovlivněn mnoha faktory. Mezi ty patří hlavně výkon a architektura samotné grafické karty a procesoru, počet numerických operací v algoritmu, nastavení kompilátorů jednotlivých jazyků, požadovaná přesnost výpočtů s plovoucí řádovou čárkou. Především je nutno počítat s přenosem dat do grafické paměti, což mnohdy zabere více času, než samotné výpočty.

Na dané počítačové sestavě provedené testy ukázaly, že v běžných aritmetických operacích jako sčítání, násobení, atp. dosahujeme pomocí CUDA C až stonásobné rychlosti výpočtů oproti jinak nejrychlejšímu C++. Goniometrické funkce lze vypočítat až 80x rychleji. Obzvláště vhodné je také použití pro násobení matic a složité obrazové operace. Ačkoliv je tato technologie vhodná jen na specifickou, paralelně řešitelnou část algoritmů, při vysokém počtu matematických operací a dat má její využití smysl.

Oblast GPGPU včetně CUDA se velmi rychle vyvíjí, za dobu trvání této práce se vystřídaly dvě hlavní verze SDK a architektury. Každá další přináší určité zjednodušení pro programátora a nové, případně rychlejší funkce. Do budoucna by bylo zajímavé zaměřit se právě na novinky, případně optimalizované knihovny dodávané s vývojovým prostředím. Taktéž se nabízí srovnání s konkurenční technologií ATi Stream a s obecnějšími OpenCL a DirectCompute.

Seznam použité literatury

[1] IXBT Labs [online]. 27.10.2008 [cit. 2011-04-20]. IXBT Labs - NVIDIA CUDA - Page 1: Introduction, CPU and GPU differences. Dostupné z WWW: <http://ixbtlabs.com/articles3/video/cuda-1-p1.html>.

[2] NVIDIA Corporation [online]. 11.9.2010 [cit. 2011-04-20]. NVIDIA CUDA C Programming Guide, Version 3.2. Dostupné z WWW: <http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf>.

[3] NVIDIA Corporation [online]. 20.8.2010 [cit. 2011-04-20]. CUDA C Best Practices Guide, Version 3.2. Dostupné z WWW: <http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf>.

[4] NVIDIA Corporation [online]. c2011 [cit. 2011-04-21]. Why choose Tesla. Dostupné z WWW: <http://www.nvidia.com/object/why-choose-tesla.html>.

[5] NVIDIA Corporation [online]. c2011 [cit. 2011-05-01]. CUDA in Action. Dostupné z WWW: <http://www.nvidia.com/object/cuda_in_action.html>.

[6] Wikipedie, otevřená encyklopedie [online]. 6.1.2011 [cit. 2011-04-22]. GeForce 400 – Wikipedie. Dostupné z WWW: <http://cs.wikipedia.org/wiki/GeForce_400>.

[7] The MathWorks, Inc. [online]. c2011 [cit. 2011-04-22]. MATLAB GPU Computing with NVIDIA CUDA-Enabled GPUs. Dostupné z WWW: <http://www.mathworks.com/discovery/matlab-gpu.html>.

[8] NVIDIA Corporation [online]. c2011 [cit. 2011-04-22]. Parallel Nsight Hardware Configurations | NVIDIA Developer Zone. Dostupné z WWW: <http://developer.nvidia.com/parallel-nsight-hardware-configurations>.

[9] AT&T Research [online] 23.10.2010 [cit. 2011-04-22] An incomplete list of C++ compilers. Dostupné z WWW: <http://www2.research.att.com/~bs/compilers.html>.

[10] The Eclipse Foundation [online] c2011 [cit 2011-05-11] Eclipse Helios. Dostupné z WWW: <http://www.eclipse.org/helios/>

[11] Ecma International [online]. 11.9.2010 [cit. 2011-04-28]. ECMA-334 C# Language Specification. Dostupné z WWW: <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf>.

[12] Algoritmy.net [online] c2011 [cit. 2011-04-28]. Fibonacciho posloupnost - Algoritmy.net. Dostupné z WWW: <http://www.algoritmy.net/article/116/Fibonacciho-posloupnost>.

[13] Oracle [online] c2011 [cit. 2011-05-10] Primitive Data Types. Dostupné z WWW: <http://download.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>

Seznam ilustrací

Seznam tabulek

Seznam grafů

Příloha A – Použitý hardware

Procesor – Intel Core 2 Duo E6600 @ 3GHz

Grafická karta – MSI N460 GTX Cyclone 768D5 (NVIDIA GeForce 460 GTX 768MB)

Paměť – RAM 4092 MB DDR2 800MHz

Základní deska – DFI Infinity P965-S

Operační systém – Microsoft Windows 7 Professional

Příloha B – Tabulky výsledných časů

1x operace sčítání				
	4 mil.	**8 mil.**	**16 mil.**	**32 mil.**
CUDA C (ms)	0,75	1,41	2,72	5,32
CUDA C + přenos dat (ms)	35,75	71,41	142,72	285,32
C++ (ms)	16,05	31,37	61,15	125,30
C# (ms)	17,00	34,00	67,00	138,00
Java (ms)	11,21	23,60	43,85	90,05

1x operace odčítání				
	4 mil.	**8 mil.**	**16 mil.**	**32 mil.**
CUDA C (ms)	0,78	1,41	2,71	5,30
CUDA C + přenos dat (ms)	35,78	71,41	142,71	285,30
C++ (ms)	15,55	31,36	61,16	128,57
C# (ms)	17,00	34,00	68,00	137,00
Java (ms)	11,23	23,90	47,45	91,53

1x operace násobení				
	4 mil.	**8 mil.**	**16 mil.**	**32 mil.**
CUDA C (ms)	0,75	1,41	2,69	5,33
CUDA C + přenos dat (ms)	35,75	71,41	142,69	285,33
C++ (ms)	15,72	31,99	61,30	126,33
C# (ms)	17,00	34,00	68,00	137,00
Java (ms)	11,63	24,56	47,42	91,67

1x operace dělení				
	4 mil.	**8 mil.**	**16 mil.**	**32 mil.**
CUDA C (ms)	0,84	1,55	3,05	5,96
CUDA C + přenos dat (ms)	35,84	71,55	143,05	285,96
C++ (ms)	28,47	57,03	114,47	229,24
C# (ms)	47,00	95,00	190,00	382,00
Java (ms)	24,17	48,37	96,76	194,31

32x operace sčítání				
	4 mil.	**8 mil.**	**16 mil.**	**32 mil.**
CUDA C (ms)	0,81	1,56	3,00	5,97
CUDA C + přenos dat (ms)	35,81	71,56	143,00	285,97
C++ (ms)	69,42	137,68	275,23	550,04
C# (ms)	111,00	221,00	442,00	894,00
Java (ms)	194,92	380,09	754,56	1511,66

32x operace odčítání				
	4 mil.	**8 mil.**	**16 mil.**	**32 mil.**
CUDA C (ms)	0,82	1,55	2,98	5,90
CUDA C + přenos dat (ms)	35,82	71,55	142,98	285,90
C++ (ms)	67,10	134,43	268,93	538,08
C# (ms)	132,00	265,00	530,00	1060,00
Java (ms)	139,67	279,68	424,01	821,84

32x operace násobení				
	4 mil.	**8 mil.**	**16 mil.**	**32 mil.**
CUDA C (ms)	0,82	1,56	3,00	5,90
CUDA C + přenos dat (ms)	35,82	71,56	143,00	285,90
C++ (ms)	86,38	173,51	346,59	692,87
C# (ms)	657,00	1318,00	2633,00	5271,00
Java (ms)	215,31	433,90	865,80	1725,85

32x operace dělení				
	4 mil.	**8 mil.**	**16 mil.**	**32 mil.**
CUDA C (ms)	37,26	74,42	148,87	297,46
CUDA C + přenos dat (ms)	72,26	144,42	288,87	577,46
C++ (ms)	969,11	1938,70	3880,25	7755,27
C# (ms)	2078,00	4158,00	8330,00	16639,00
Java (ms)	1420,73	2844,40	5697,56	11387,50

1x operace sinus				
	4 mil.	8 mil.	16 mil.	32 mil.
CUDA C (ms)	0,59	1,03	1,90	3,79
CUDA C + přenos dat (ms)	25,99	50,33	99,52	199,00
C++ (ms)	77,96	138,89	276,95	553,69
C# (ms)	135,00	269,00	539,00	1077,00
Java (ms)	558,34	1113,98	1554,07	3097,33

1x operace cosinus				
	4 mil.	8 mil.	16 mil.	32 mil.
CUDA C (ms)	0,57	1,00	1,93	3,80
CUDA C + přenos dat (ms)	25,79	49,37	99,50	200,23
C++ (ms)	70,52	169,49	277,30	680,91
C# (ms)	139,00	274,00	550,00	1100,00
Java (ms)	353,77	703,51	1515,95	2993,91

1x operace odmocnina				
	4 mil.	8 mil.	16 mil.	32 mil.
CUDA C (ms)	0,56	1,02	1,95	3,76
CUDA C + přenos dat (ms)	25,66	50,20	99,53	199,13
C++ (ms)	44,32	88,19	176,56	373,33
C# (ms)	84,00	166,00	336,00	671,00
Java (ms)	80,09	160,48	336,76	656,41

1x operace logaritmus				
	4 mil.	8 mil.	16 mil.	32 mil.
CUDA C (ms)	0,56	1,03	1,94	3,78
CUDA C + přenos dat (ms)	25,61	50,24	99,52	199,19
C++ (ms)	73,63	145,56	291,11	582,67
C# (ms)	170,00	340,00	683,00	1369,00
Java (ms)	101,08	202,43	405,52	809,18

32x operace sinus				
	4 mil.	**8 mil.**	**16 mil.**	**32 mil.**
CUDA C (ms)	1,86	3,62	7,13	14,19
CUDA C + přenos dat (ms)	26,66	52,55	104,24	213,30
C++ (ms)	3146,08	6290,13	12587,89	25177,56
C# (ms)	4148,00	8290,00	16590,00	33220,00
Java (ms)	6307,87	12568,33	20327,22	40893,00

32x operace cosinus				
	4 mil.	**8 mil.**	**16 mil.**	**32 mil.**
CUDA C (ms)	1,84	3,62	7,14	14,17
CUDA C + přenos dat (ms)	26,79	51,88	104,69	213,22
C++ (ms)	314,62	6221,24	12589,22	24881,54
C# (ms)	4542,00	9084,00	18178,00	36380,00
Java (ms)	5456,53	10948,55	22007,37	43728,59

32x operace odmocnina				
	4 mil.	**8 mil.**	**16 mil.**	**32 mil.**
CUDA C (ms)	3,46	6,81	13,52	26,94
CUDA C + přenos dat (ms)	28,29	55,01	111,34	225,87
C++ (ms)	3120,90	7538,08	12488,41	30148,57
C# (ms)	2458,00	4917,00	9840,00	19675,00
Java (ms)	2373,23	5078,33	9511,00	19081,64

32x operace logaritmus				
	4 mil.	**8 mil.**	**16 mil.**	**32 mil.**
CUDA C (ms)	1,83	3,55	6,98	13,86
CUDA C + přenos dat (ms)	26,70	51,84	104,53	212,78
C++ (ms)	2309,41	4623,60	9239,72	18497,82
C# (ms)	7652,00	15324,00	30644,00	61483,00
Java (ms)	8580,45	17248,47	34568,00	69032,28

Násobení matic			
	1024x1024	**2048x2048**	**4096x4096**
CUDA C (ms)	43,11	391,05	3407,99
CUDA C + přenos dat (ms)	52,56	425,64	3544,74
C++ (ms)	6137,10	64369,29	551390,94
C# (ms)	10509,00	300684,00	2527250,00
Java (ms)	9149,90	92562,28	838357,94

Obrazové operace				
	Převod - šedá škála	**Převod - černo-bílá**	**Diskretni konvoluce**	**Tvorba histogramu**
CUDA C + přenos dat (ms)	8,62	5,23	14,47	6,81
C++ (ms)	6,66	2,40	59,48	101,14
C# (ms)	41,00	52,00	317,00	59,00
Java (ms)	33,40	8,60	201,73	6,72