VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

BAKALÁŘSKÁ PRÁCE

Brno, 2016 Jaromír Polášek

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

VYUŽITÍ PARALELIZOVANÝCH MATEMATICKÝCH OPERACÍ V OBLASTI ZPRACOVÁNÍ DAT
SIGNAL PROCESSING USING PARALLEL MATHEMATICAL OPERATIONS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **Jaromír Polášek**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Zdeněk Mžourek**
SUPERVISOR

BRNO 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní obor **Teleinformatika**
Ústav telekomunikací

Student: Jaromír Polášek ***ID:*** 164770
Ročník: 3 ***Akademický rok:*** 2015/16

NÁZEV TÉMATU:

Využití paralelizovaných matematických operací v oblasti zpracování dat

POKYNY PRO VYPRACOVÁNÍ:

Moderní měřící přístroje a různé senzorické sítě umožnují zaznamenávat neustále zvětšující se množství dat. I přes narůstající výpočetní výkon klasických procesorů přestávají být běžně používané algoritmy zpracování signálů dostatečně rychlé pro analýzu takového množství dat. Jako jedno z účinných řešení tohoto problému se ukazuje paralelní implementace těchto algoritmů.
Cílem této práce je porovnání rychlosti klasických a paralelizovaných implementací různých algoritmů pro zpracování dat jako je například konvoluce nebo Fourierova transformace na datových souborech různých velikostí.

DOPORUČENÁ LITERATURA:

[1] HAGER, Georg a WELLEIN, Gerhard. Introduction to High Performance Computing for Scientists and Engineers. Boca Raton, FL, USA: CRC Press, Inc., 1st edition, 2010, ISBN 9781439811924.

[2] EIJKHOUT, Viktor. Introduction to High Performance Scientific Computing. Second edition, 2014, ISBN 978--257-99254-6

[3] BARLAS, Gerassimos. Multicore and gpu programming: an integrated approach. 1st edition. Waltham, MA: Elsevier, 2014, ISBN 978-012-4171-374.

Termín zadání: 1.2.2016 ***Termín odevzdání:*** 1.6.2016

Vedoucí práce: Ing. Zdeněk Mžourek
Konzultant bakalářské práce:

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

ABSTRAKT

Tato Bakalářská práce se zabývá zrychlení výpočtů funkcí paralelními výpočty, zprostředkovaných grafickými kartami NVDIA, prostřednictvím technologie CUDA. Teoretická část popisuje obecné principy paralelních výpočtů a základní vlastnosti a parametry grafických karet NVDIA. Teoretické část se také věnuje základním principům technologie CUDA. Konec teoretické části se věnuje knihovnám FFTW a CuFFT. Praktická část se zabývá srovnání výkonu GPU a CPU na funkcích `filter2D` a `Canny` a možnostem praktického zrychlení výpočtu rychlé konvoluce. V praktické části jsou také popsány ukázky kódu, který byly použity pro srovnání výkonu GPU a CPU. Výsledky těchto programů jsou následně zaneseny do grafů a zhodnoceny.

KLÍČOVÁ SLOVA

`Canny`, CPU, CUDA, cuFFT, GPU, GPGPU, grafy, FFT, FFTW, `filter2D`, funkce, OpenCV, paralelní výpočty, paralelní zpracování, rychlá konvoluce, výpočty

ABSTRACT

This Bachelor thesis deals with the acceleration of function calculations, using parallel computing mediated by NVDIA graphics cards via CUDA technology. The theoretical part describes the general principles of parallel computing and the basic characteristics and parameters of graphics cards NVDIA. The theoretical part also deals with basic principles of CUDA technology. End of the theoretical part focuses on FFTW and cuFFT libraries. The practical part deals with the comparison of the performance between GPU and CPU functions `filter2D` and `Canny` and practical possibilities of accelerating fast convolution calculation. The practical part also describes sample code that was used to compare the performance between GPU and CPU. The results of this program are then plotted and evaluated.

KEYWORDS

`Canny`, CPU, CUDA, cuFFT, GPU, GPGPU, graphs, fast convolution, FFT, FFTW, `filter2D`, functions, OpenCV, parallel computing, parallel processing, calculations

POLÁŠEK, Jaromír *Využití paralelizovaných matematických operací v oblasti zpracování dat*: bakalářská práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2015. 71 s. Vedoucí práce byl Ing. Zdeněk Mžourek

PROHLÁŠENÍ

Prohlašuji, že svou bakalářskou práci na téma „Využití paralelizovaných matematických operací v oblasti zpracování dat“ jsem vypracoval samostatně pod vedením vedoucího bakalářské 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 bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu Ing. Zdeněkovi Mžourkovi za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

Brno ....

podpis autora

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

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

PODĚKOVÁNÍ

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

Brno . . . . . .

. . . . . .

podpis autora

EVROPSKÁ UNIE
EVROPSKÝ FOND PRO REGIONÁLNÍ ROZVOJ
INVESTICE DO VAŠÍ BUDOUCNOSTI

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

ÚVOD

Informační technologie v posledních letech zaznamenali neuvěřitelný posun. Stále větší nároky na výpočetní výkon donutily programátory hledat nové řešení pro urychlení výpočetních operací. Mezi jedno z těchto řešení patří paralelní zpracování. Paralelní zpracování využívá toho, že ne všechny operace zpracovávané výpočetní jednotkou jsou na sebe závislé a nemusejí být prováděni sériově za sebou. Velké množství operací probíhajících paralelně přestávají CPU stíhat, proto se čím dál víc do výpočetních úkonů zapojují GPU. GPU obsahují velké množství jader, které mohou pracovat na sebe nezávisle a proto jsou ideální pro paralelní výpočty. Dále jejich druh pamětí je mnohem rychlejší než klasické paměti RAM. Cílem této práce je porovnat výkon GPU a CPU pomocí výpočtu různých operací ze signály.

V první kapitole jsou popsány základy paralelního programování. Kapitola se také věnuje dvěma základním zákonům, které popisují zrychlení jaké lze dosáhnout paralelním zpracováním. Tyto zákony jsou matematicky odvozeny a popsány. Poslední část kapitoly popisuje dělení granularity paralelismu. V této části jsou popsány tři základní typy a u každého typu je uvedena část kódu programu, který slouží jako příklad.

Druhá kapitola se věnuje vývoji grafických karet. Následně jsou popsány rozdíly mezi GPU a CPU. Druhá část kapitoly taky popisuje Flynnovu taxonomii paralelních architektur, která rozděluje počítače podle toho kolik datových a instrukčních toků mohou zpracovávat. Poslední část kapitoly se věnuje typům pamětí v počítačích využívajících paralelní programování.

Třetí kapitola je zaměřena na popis grafických karet NVDIA a to konkrétně čipu Kepler GK110. U tohoto čipu jsou popsány základní výhody oproti staršímu typu Fermi. Druhá část této kapitoly popisuje technologii CUDA. Jsou zde uvedeny základní principy technologie CUDA. Podrobněji je popsána knihovna CUFFT z kterou se nejvíc pracuje.

Čtvrtá kapitola se věnuje porovnání rychlosti aplikace signálových operací na CPU a GPU. V první části jsou popsány základní použité operace se signály. Následuje popis a vysvětlení použitých funkcí z knihovny OpenCV. Poslední část kapitoly tvoří stručný popis použitého programu a zhodnocení výsledků, které jsou posléze zaneseny do grafů a tabulek.

Pátá kapitola je zaměřená na zrychlení výpočtu rychlé konvoluce. V první části kapitoly je matematicky nastíněna rychlá konvoluce a následně jsou popsány použité knihovny FFTW a cuFFT. Ve druhé části je popsán použitý kód pro uskutečnění rychlé konvoluce. V poslední části jsou zhodnoceny výsledky dvou možností zrychlení výpočtu rychlé konvoluce, tyto výsledky jsou následně zaneseny do tabulek a grafů.

1 ZÁKLADY PARALELNÍHO PROGRAMOVÁNÍ

V této kapitole jsou popsány základy paralelního programování, dva základní zákony jako je Amdahlův zákon a Gustafsonův zákon. V této kapitole je taktéž popsána granularita paralelního procesu.

1.1 Paralelní výpočty obecně

Paralelní programováni je styl programování kde se snaží programátor o to, aby se všechny výpočty prováděli současně a to z důvodu urychlení práce. Paralelní programování bylo používáno i v dřívější době, ale jeho hlavní rozkvět nastal v posledních pěti letech. Tento rozvoj nastal hlavně kvůli tomu, že další zvyšování frekvencí na počítačích přestalo být efektivní a to z důvodu neúnosné energetické spotřebě a tepelnému vyzařování. Kvůli těmto problémům a rozvoji vícejádrových procesorů se stalo paralelní programování hlavním způsobem zvýšení výkonu PC [11].

Dříve byly všechny programy řešeny jako sériové algoritmy. Algoritmus byl napsán jako sériový sled instrukcí. Tyto instrukce byly vykonávány v centrální výpočetní jednotce počítače. V jeden okamžik nemohlo být prováděno více instrukcí. To bylo dáno architekturou těchto čipů. Paralelní výpočty byly pouze „předstírány“ rychlým přepínáním mezi probíhajícími procesy. Paralelní programování naopak využívá několik výpočetních jednotek ve stejný čas. Sled instrukcí je rozdělen na několik částí a tím je umožněno, že více výpočetních jednotek může pracovat na jednom problému. Výpočetní jednotky nemusí být umístěny pouze na jeden počítač, pro vyřešení problému může být využita technologie založena na internetovém přenosu jako je například *Cluster*.

Zvyšování frekvence CPU bylo hlavním důvodem zvyšování výpočetního výkonu počítačů zhruba do roku 2004. Doba zpracování programu je rovna počtu instrukcí vynásobená průměrnou dobou instrukcí. Tím pádem zvyšování frekvence mikroprocesoru snižuje průměrnou dobu vykonání instrukce. Z toho plyne, že zvyšování frekvence nám zvedá výpočetní výkon mikroprocesoru. Bohužel spotřeba energie mikroprocesoru P je dána vzorcem:

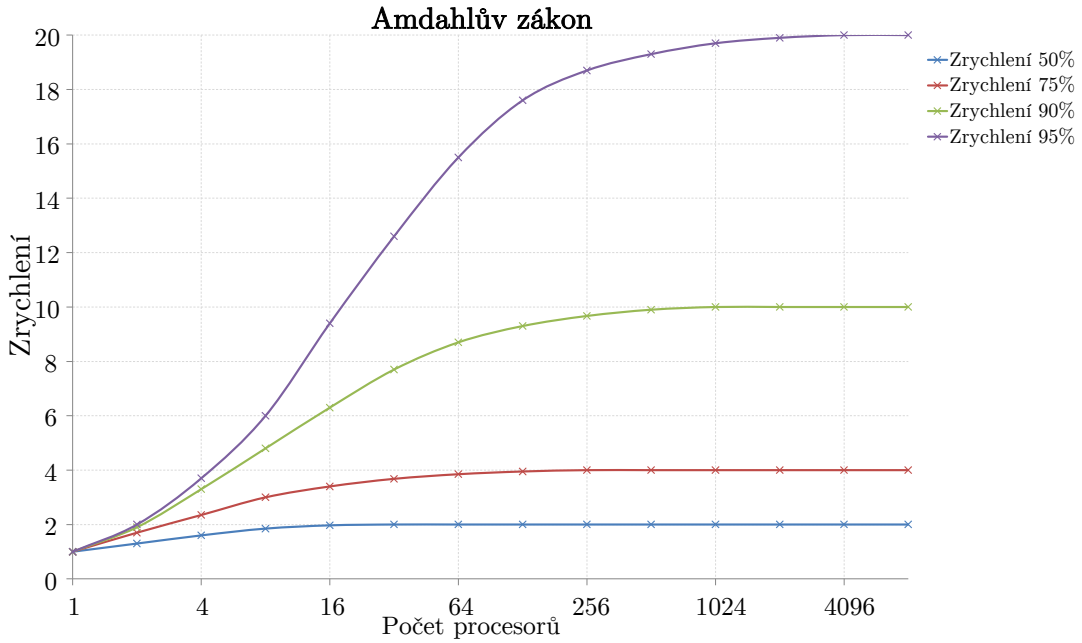
$$P = C \cdot U^2 \cdot F, \tag{1.1}$$

kde C je kapacita při přepnutí hodinového taktu, U je napětí na mikroprocesoru a F je frekvence mikroprocesoru. Z tohoto vzorce vyplývá, že zvyšování frekvence zvyšuje energetickou spotřebu čipu [11].

1.2 Amdahlův zákon

Amdahlův zákon taky známý jako Amdahlův argument je používán pro nalezení celkového maximálního zlepšení systému, kde je zlepšena pouze část programu. Nejčastěji je použit v paralelním programování pro zjištění urychlení systému při zapojení více procesorů.

Zrychlení programu v paralelním programování při použití více mikroprocesorů je limitované časem potřebným pro splnění sériové části programu. Například pokud výpočet programu zabere čtyřicet hodin při použití jednoho mikroprocesoru a z toho dvě hodiny nelze paralelně zpracovávat a zbývající hodiny mohou být paralelně zpracovány. Tak navzdory tomu kolik mikroprocesorů se bude věnovat paralelnímu zpracování programu, nejmenší potřebný čas pro zpracování programu nikdy nebude nižší než kritické dvě hodiny. Maximální teoretické zrychlení bude vždy limitované. Na grafu 1.1 můžeme tuto skutečnost sledovat. Grafem je vyjádřena závislost zrychlení počítače na počtu mikroprocesorů provádějících výpočetní výkon. Čtyři křivky nám tuto skutečnost graficky znázorňují, každá křivka odpovídá jinému procentuálnímu stupni paralelního zpracovávání programu [6, 7].



Obr. 1.1: Amdahlův zákon.

1.2.1 Matematické odvození Amdahlůva zákona

Program zpracováván systémem schopným paralelního zpracování může být roz-

dělen na dvě části:

- Část na kterou nelze využít paralelního zpracování.
- Část na kterou lze využít paralelního zpracování.

Doba provedení celého úkolu před zapojení paralelního zpracování je označena jako T. Zahrnuje část, která nedokáže využít paralelního zpracování i druhou část využívající paralelního zpracování. Procento doby zpracování celého úkolu použité na část využívající paralelní zpracování před zapojením upraveného algoritmu se nazývá p. Takže část, která nevyužívá paralelního zpracování je označena $1 - p$. Z toho vyplývá že [7]:

$$T = (1 - p)T + pT. \tag{1.2}$$

Tento vzorec charakterizuje část, která dokáže využít paralelního zpracování a je akcelerována faktorem s po nasazení paralelního zpracování. V důsledku toho, čas zpracování části nevyužívající paralelního zpracování zůstává stejný, zatímco část která i dokáže využít je:

$$\frac{p}{s}T. \tag{1.3}$$

Teoretická doba zpracování $T(s)$ celého úkolu po vylepšení je tedy:

$$T(s) = (1 - p)T + \frac{p}{s}T. \tag{1.4}$$

Amdahlův zákon pak vyjadřuje teoretické zrychlení z latencí celé úlohy:

$$S_{\text{latency}}(s) = \frac{T}{T(s)} = \frac{1}{1 - p + \frac{p}{s}}. \tag{1.5}$$

Amdahlův zákon se uplatňuje pouze v případech kdy je velikost problému stejná. V praxi, čím více výkonnějších výpočetních zdrojů bývá k dispozici, tak tím více zdrojů má tendenci být používáno pro stejný problém. Tudíž čas strávený v paralelní části programu často roste mnohem rychleji něž sériová část programu. V tomto případě nám mnohem lépe vystihuje paralelní výkon Gustafsonův zákon.

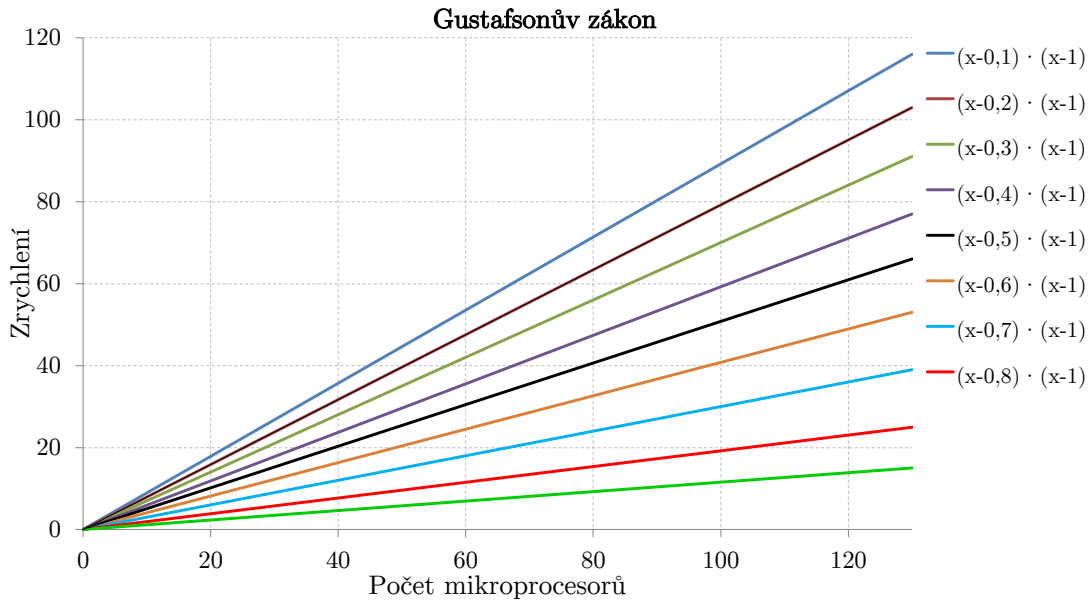
1.3 Gustafsonův zákon

Gustafsonův zákon(taky známý jako Gustaf-Barrisův zákon) je zákon počítačové techniky, který říká že výpočty obsahující velké datové bloky mohou být efektivně paralelně zpracovány. Gustafsonův zákon slouží jako protiklad k Amdahlův zákonu. Gustafsonův zákon byl poprvé přednesen Johnem L. Gustafsonem a Edwinem H. Barsisem.
Gustafsonův zákon je definován jako:

$$S(P) = P - \alpha(P - 1),$$

kde P je počet mikroprocesorů, S je zrychlení a α je největší část, kterou nelze paralizovat.

Na obrázku 1.2 můžeme vidět závislost zrychlení úlohy na počtu mikroprocesorů pro různé velikosti α. Z grafu vyplývá že čím dál tím větší velikost α tím je maximální teoretické zrychlení S menší.



Obr. 1.2: Gustafsonův zákon.

Gustafsonův zákon byl navržen tak aby odpověděl na nedostatky Amdahlůva zákona, který nedokáže plně popsat situaci při zapojení více strojů. Gustafsonův zákon místo toho navrhuje, že programátoři mají tendenci nastavovat velikost zpracovávaného programu tak aby využil veškerý dostupný výpočetní výkon. Takže čím rychlejší paralelní vybavení je k dispozici, tím rychleji můžeme vyřešit větší problémy za stejný čas [6].

1.3.1 Matematické odvození Gustafsonova zákona

Doba zpracování programu na paralelním stroji je složena z

$$(a + b), \tag{1.6}$$

kde a je sekvenční čas a b je paralelní čas, na jakémkoliv procesoru P.

Klíčový předpoklad Gustafsonova zákona je, že celková práce udělaná pomocí paralelního procesu se mění lineárně z počtem mikroprocesorů. Jak můžeme vidět na obrázku 1.2. To znamená že b by mělo být fixní, protože počet procesorů P se může lišit. Tomuto závěru odpovídá vzorec:

$$(a + P)b. \tag{1.7}$$

Zrychlení $S(P)$ je pak:

$$S(P) = \frac{\Big((a + P)b\Big)}{(a + b)}, \tag{1.8}$$

definující α

$$\alpha = \frac{a}{(a + b)}, \tag{1.9}$$

kde α je část zpracování programu kterou nelze paralizovat.

Zrychlení pak odpovídá vzorci:

$$S(P) = \alpha + P(1 - \alpha) = P - \alpha(P - 1). \tag{1.10}$$

Pokud je α malé, zrychlení přibližně odpovídá velikosti P, jak je vyžadováno. Dokonce může nastat případ kdy se α zmenšuje a zároveň se P(společně z velikostí problému) zvyšuje. Pak nastává to že se S blíží monotónně k P z růstem P. Z toho vyplývá to, že pomocí Gustafsonova zákona můžeme překonat limity stanovené Amdahlovým zákonem.

Gustafsonův zákon se setkává z problémem toho, že ne každý program pracuje s velkými datovými bloky. Proto je dobré uvažovat Gustafsonův zákon pouze v případech, kdy dochází k zpracování velkých datových bloků. Například algoritmy s nelineární dobou běhu se jen velmi těžce popisují Gustafsonovým zákonem [6].

1.4 Rozdělení granularity paralelismu

Tato sekce se věnuje otázce „Kolik paralelních prostředků je pro program dostupných?“.

Při granularitě paralelismu nás zajímají tři věci:

- Maximální počet akcí které mohou být prováděny paralelně.
- Druh akcí které se mají provést paralelně.
- Obtížnost akci provedených paralelně.

1.4.1 Datový paralelismus

Pro program je velmi běžné mít ve svém kódu smyčky, které se provedou pro všechny elementy ve velkém datovém bloku:

```
for (i=0, i<10000; i++)
     x[i] = 2*y[i];
```

Jednotlivé výpočty tohoto kódu nezávisí na výpočtech předchozích, a proto je možné je provádět nezávisle na sobě. Díky této skutečnosti lze využít paralelního výpočtu, protože výpočet pak bude tzv. deterministický neboli výsledek bude vždy stejný. Kód tohoto stylu je příklad datového paralelismu. Pokud bychom měli tolik procesorů kolik je řadových elementů, kód by vypadal mnohem jednodušeji. Každý procesor by vykonával příkaz:

```
a = 2*b.
```

Pokud kód obsahuje dominantně smyčky, může být efektivně vykonán všemi procesory v *lockstep* módu[1]. Architektury založené na této myšlence existují ale nejsou moc používány. Procesory založené na této architektuře mohou pracovat pouze v lockstep módu. Architektury postavené na zpracovávání řadových elementů na jednou, se používají například pro zpracovávání obrazu kde každý jednotlivý pixel zpracovává jiný procesor. Kvůli tomu jsou GPU z velké části založené na datovém paralelismu. V následujícím případu si tyto příklady více rozvedeme [6, 11]:

```
for 0 ≤ i < do
```

$$
\begin{aligned}
i_{leva} &= \mathrm{mod}(i-1, max) \\
i_{prava} &= \mathrm{mod}(i+1, max) \\
a_i &= (b_{i_{leva}} + b_{i_{prava}})/2
\end{aligned}
$$

Na PC které dokáže využít datový paralelismus. Kód může být implementován jako:

```
bleft ← shiftright(b)
bright ← shiftleft(b)
a ← (bleft + bright)/2
```

[1]Lockstep mód je technika používána pro dosažení vysoké spolehlivosti mikroprocesoru. To se děje tak, že je přidán druhý identický mikroprocesor, který kontroluje a potvrzuje operace hlavního mikroprocesoru.

kde `shiftleft/right` instrukce slouží k přesunu dat mezi procesory. Proto druhý algoritmus potřebuje pro svoji efektivitu to, aby procesory byly schopny mezi sebou komunikovat.

1.4.2 Instrukční paralelismus

Instrukční paralelismus pracuje stále na úrovni jednotlivých instrukcí, ale instrukce nemusí být podobné. Na příkladu níže můžeme sledovat kód využívající instrukční paralelismus.

```
a ← b + c
d ← e * f
```

Oba příkazy jsou na sebe nezávislé a tak mohou být vypracovány paralelně. Tento druh paralelního zpracování je pro člověka příliš těžkopádný na identifikaci, ale překladače jsou velmi efektivní pří zpracovávání takového kódu. Instrukční paralelismus je nezbytný pro získání dobrého výkonu z moderního *superskalárního* PC[2]. Hlavní úkol překladače a procesoru je identifikovat a získat výhody z použití instrukčního paralelismu. Nakolik je možné využít instrukčního paralelismu v programu se liší dle druhu programu. Ve vědě nebo grafice se nachází obrovské uplatnění instrukčního paralelismu, zato v kryptografii je toto uplatnění velmi omezené [6, 11].

Mikroprocesorové techniky použité pro uskutečnění instrukčního paralelismu jsou například:

- Proudové zpracováni instrukcí(*Instruction Pipelining*) kde se zpracovávání více instrukcí může částečně překrývat.
- Superskalární zpracování (VLIW) kde více výpočetních jednotek je používáno pro zpracování mnoha instrukcí paralelně.
- Přejmenování registrů, který odkazuje na techniku sloužící k tomu aby se zabránilo zbytečné serializace operací programu, způsobené znovu užíváním registrů.
- Spekulativní zpracování instrukcí, které umožňuje vypracovat instrukci před tím než je jisté, kde by se mělo vypracování provést.

[2]Superskalární PC obsahuje více jak jednu frontu zřetězení informací.

1.4.3 Úkolový paralelismus

Úkolový paralelismus pracuje na principu identifikace celých podprogramů které mohou být zpracovány paralelně. Například hledání v datovém stromu může být implementováno jako:

```
if optimal (root) then
        exit
else
   parallel:SearchInTree(leftchild),SearchInTree(rightchild)
        Procedure SearchInTree(root)
```

Funkce pro hledáni v příkladu není synchronní a počet úloh není fixní, může libovolně růst nebo se snižovat. V praxi není moc vhodné zpracovávat příliš mnoho úloh zároveň, protože práce více procesů v čase znamená větší režii pro správce výpočetního výkonu PC, tím pádem od jistého množství procesů začne být režie natolik vysoká, že se takový počet paralelních procesů už nevyplatí. Proto procesory jsou nejvíc efektivní pokud pracují na menším počtu procesů. Proto se úlohy vkládají do front. Ukázka dvou typů kódu je například:

```
while there are task left do,
```

nebo:

```
wait until a processor becomes inactive;
    spawn a new task on it
```

Naproti příkladu z datového paralelismu, pořadí dat není přesně dané předem. Takže tento druh paralelního zpracování je nejvíce vhodný pro programování vláken. Například skrze knihovnu OpenMP nebo programovací jazyky jako je třeba C++ nebo Java [6, 7].

2 VYUŽITÍ GRAFICKÝCH KARET V MODERNÍM PROGRAMOVÁNÍ

V této kapitole je popsán vývoj grafických karet a s ním spojené změny práce s procesory, vlákny a paměťmi. Dále jsou zde uvedeny informace o rozdílech mezi GPU a CPU při práci s instrukcemi a daty.

2.1 Vývoj grafických karet

V minulosti se ke všem výpočetním operacím používal CPU a GPU byl využíván pouze jako koprocesor[1]. Kvůli náročnosti výpočetních výkonů nebylo nutné, aby aplikace byly psány jako více vláknové, proto byla většina aplikací napsána jako jedno vláknové operace (tzv. *single-thread*) a to znamená, že procesor si svoji práci vykonává postupně bez jakékoliv možnosti dělení práce. Postupem času se na trhu začaly objevovat procesory, které dokáží pracovat s několika vlákny najednou (tzv. *multi-thread*). Tato technologie umožňuje procesoru zpracovávat více vláken najednou a tím zrychlit celkový výpočetní proces. Bohužel i nejlepší současné procesory dokáží zpracovat pouze 32 vláken na jednou. Pro moderní aplikace využívající paralelní zpracování není 32 vláken dostatečných. Procesor navíc nezpracovává pouze výpočetní program, zároveň musí udržovat procesy nutné pro běh operačního systému. Tím pádem nedokáže zaměřit svou celou výpočetní kapacitu na zpracovávaný program [11, 12].

Grafická karta má odlišnou strukturu. Je složena z velkého množství *stream* procesorů (řádově stovky až tisíce), které jsou schopné samostatně a velmi rychle zpracovávat jednotlivé výpočty. Této struktury dokáží plně využít moderní technologie jako je CUDA pro karty NVDIA nebo OpenCL™ pro karty AMD i NVDIA. Tyto technologie dokáží z jednovláknového instrukce vytvořit instrukci vícevláknovou, která je rozdělena mezi stream procesory a ty i paralelně zpracují. Díky tomuto řešení dojde k velmi velkému urychlení celého vypočtu a to řádově 10–100× podle náročnosti výpočtu [12].

Na první pohled se může zdát, že při přihlédnutí ke všem skutečnostem by nám grafická karta měla teoreticky posloužit jako opravdu velice výkonný počítač. Opak je ale pravdou. Grafické karty jsou sice schopny poskytnout obrovský výpočetní výkon při zpracování dat, ale nejsou schopné provádět standardní obslužné operace[2],

[1] Koprocesorem je myšleno to, že se grafická karta nikdy samostatně nezapojovala do výpočetních výkonů a pracovala výhradě jako pomocný výpočetní blok pro CPU.

[2] Mezi standardní obslužné operace patří například obsluha přerušení od Hardwaru.

jak tomu je u CPU počítače. Proto celý systém využití grafických karet při složitých výpočtech funguje tak, že všechny obslužné činnosti kromě samotných výpočtů provádí CPU a GPU si „zavolá na pomoc“ až při výpočtech. Tento fakt ale není ničemu na škodu, protože díky tomu není zatěžován CPU a počítač se velmi zrychlí [12].

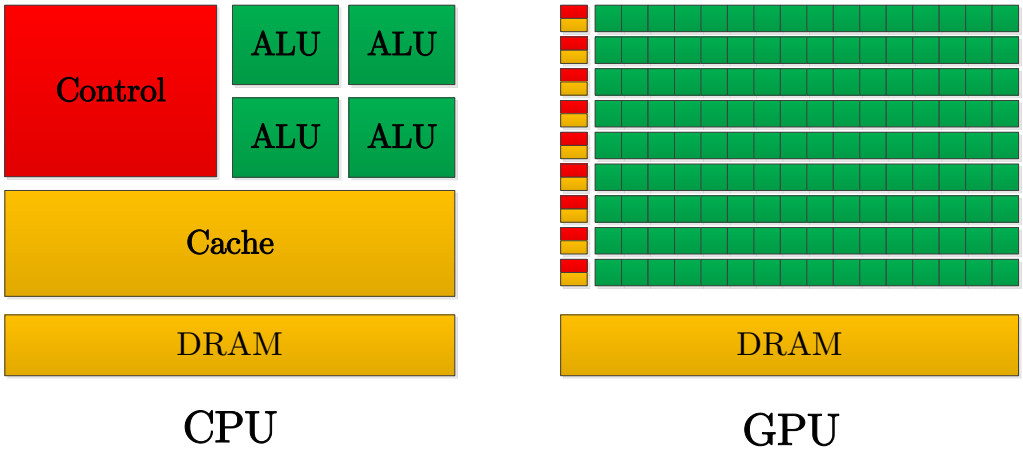
Díky stále většímu zájmu o využití grafických karet pro zrychlení aplikací, byla firmou NVDIA navržena speciální řada karet TESLA. Tyto karty obsahují obrovské množství stream procesorů. Například Tesla k40 obsahuje 2880 stream procesorů. Tato karta je založena na architektuře Tesla. Největší rozdíl mezi těmito kartami a standardními uživatelskými byla absence obrazového konektoru, ale od řady Tesla-C tyto karty obsahují 1 obrazový port [4, 1].
Grafické karty Tesla nachází nejčastější využití v těchto případech:

- V simulacích a ve výpočtech velkého rozsahu.
- Generace náročných obrazů pro použití ve vědě. Jako je například generace grafů pomocí programu Gunrock.
- Použití společně s CUDA nebo OpenCL.

2.2 Rozdíl GPU a CPU

V dnešní době existují dva směry vývoje mikroprocesorů a to vícejádrové (*multi-core*) a mnohojádrové (*many-core*). Vícejádrové mikroprocesory jsou klasické mikroprocesory ve stolním PC nebo mobilu. Tyto mikroprocesory většinou zastávají obecnou funkci ale mohou se starat jak o speciální programy puštěné v systému, tak i o obecné programy spojené z operačním systémem. Mají velmi velkou vyrovnávací paměť (*cache*). Oproti tomu vícejádrové mikroprocesory mají malou vyrovnávací paměť (cache). Nejčastěji se tyto mikroprocesory zabudovávají do grafických karet. Grafické procesory nemohou pracovat jako obecné obslužné procesory a to kvůli tomu že nemají dostatečnou paměť *cache* a nedostatečnou kontrolní část čipu. Většinou slouží pro grafické operace na počítači nebo pro speciálně zaměřené výpočetní programy [21].

Kvůli velkým odlišnostem ve své struktuře, nelze pro mikroprocesory použít stejný nízkoúrovňový programovací jazyk. Některé vysokoúrovňové jazyky tento problém už řeší (např. Haskell nebo Erlang). Struktura obou typů mikroprocesorů je graficky znázorněná na obrázku 2.1. Na obrázku jsou znázorněny struktury klasického čtyř jádrového procesoru a grafické karty. Na struktuře procesoru je vidět velký blok kontroly a paměti *cache*. Mikroprocesor obsahuje čtyři výpočetní jádra, které mohou být navíc ještě rozděleny virtuálně pomocí technologie *hyper-threading*. Struktura mikroprocesoru GPU obsahuje mnoho výpočetních jader, ale skoro žádné



Obr. 2.1: Srovnání struktury CPU a GPU.

kontrolní obvody, proto se více hodí na specializovanou práci. Oba typy mikroprocesorů obsahují na svém obvodovém čipu velkou paměť DRAM a to řádově GB.

2.3 Flynnova taxonomie paralelních architektur

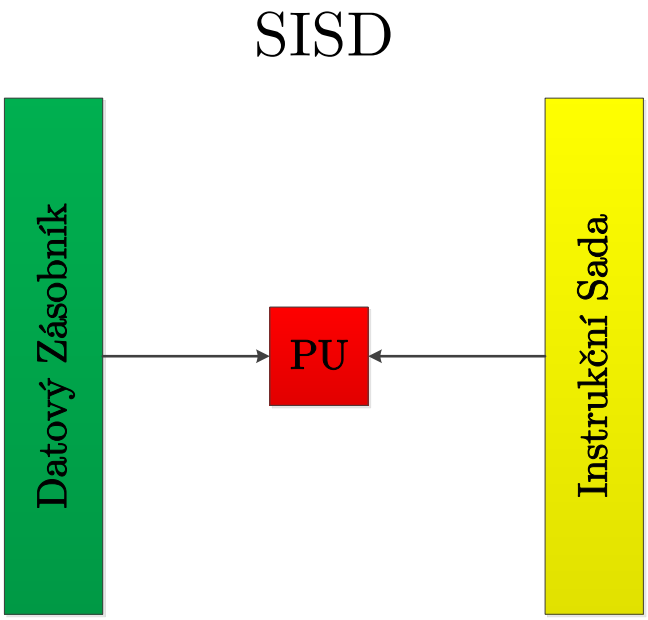
Flynnova taxonomie paralelních architektur je dělení počítačů podle toho kolik datových a instrukčních toků mohou zpracovávat. Toto rozdělení obsahuje čtyři typy.

2.3.1 SISD

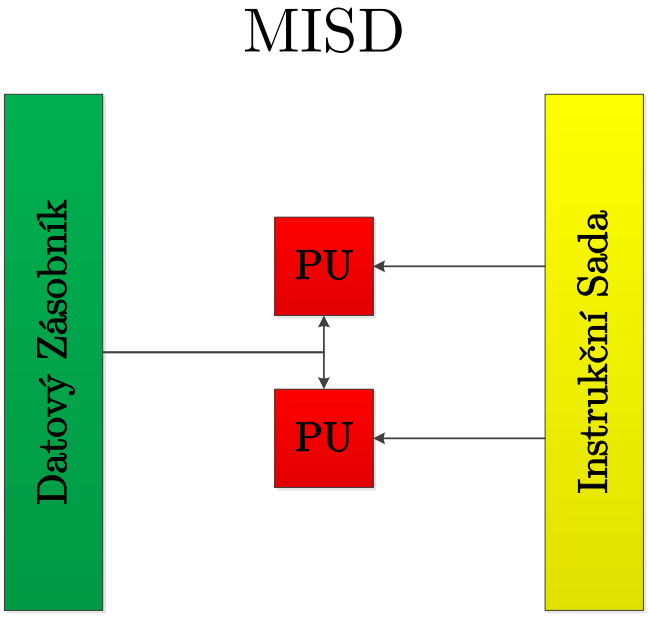
Obsahuje jeden datový a jeden instrukční tok(Single Instruction Multiple Data stream). Diagram procesu je ukázán na obrázku 2.2. Jedná se o klasickou architekturu kde jsou instrukce prováděny za sebou na jednom datovém toku. SISD například využívá procesor INTEL 8051. Tyto procesory jsou nejvíce využívány v jednoduchých přístrojích jako jsou například pračky [6].

2.3.2 MISD

Obsahuje více instrukčních proudů a jeden datový proud(Multiple Instruction Single Data stream). Několik instrukčních proudů vede instrukce z programu do několika výpočetních jednotek, kde jsou tyto instrukce paralelně prováděny a pak jsou po jednom datovém toku vedeny do datové paměti. Diagram této architektury je ukázán na obrázku 2.3. Využití MISD je velmi minimální a to z toho důvodu, že jak SIMD tak MIMD jsou výhodnější pro aplikace využívané v běžném provozu. MISD byl ovšem využíván v počítačích raketoplánů [6, 15].



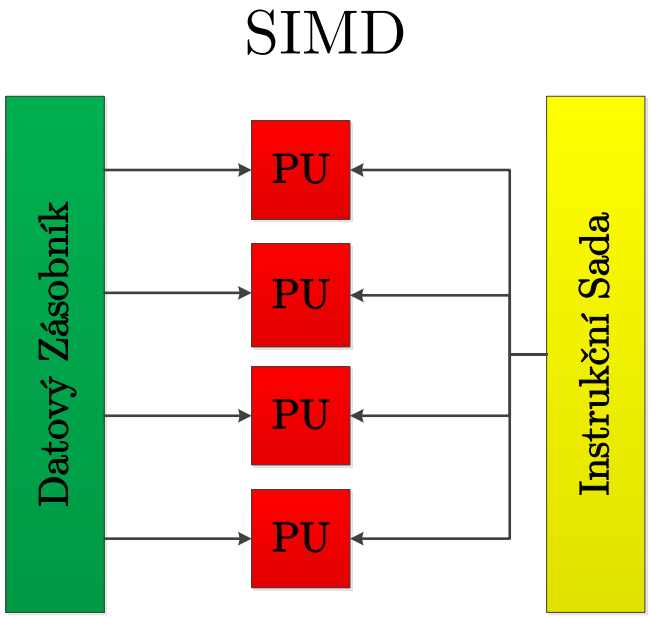
Obr. 2.2: SISD diagram.



Obr. 2.3: MISD diagram.

2.3.3 SIMD

Obsahuje jeden instrukční proud a více datových proudů (Single Instruction Multiple Data stream). Popisuje počítač z více výpočetními jednotkami, které provádějí stejnou operaci na více datových tocích. Znamená to že na výpočetních jednotkách je zpracováváno paralelně několik výpočtů, ale pouze jeden proces(instrukce) v daný moment. Diagram této architektury je ukázán na obrázku 2.4. Architektura nachází svoje největší využití v jednoduchých aplikacích jako je upravování kontrastu obrazu nebo nastavování hlasitosti audia. Většina moderních CPU je navržena tak aby využila maximálního zlepšení výkonu při použití SIMD instrukcí v multimediálních aplikacích [15, 17].



Obr. 2.4: SIMD diagram.

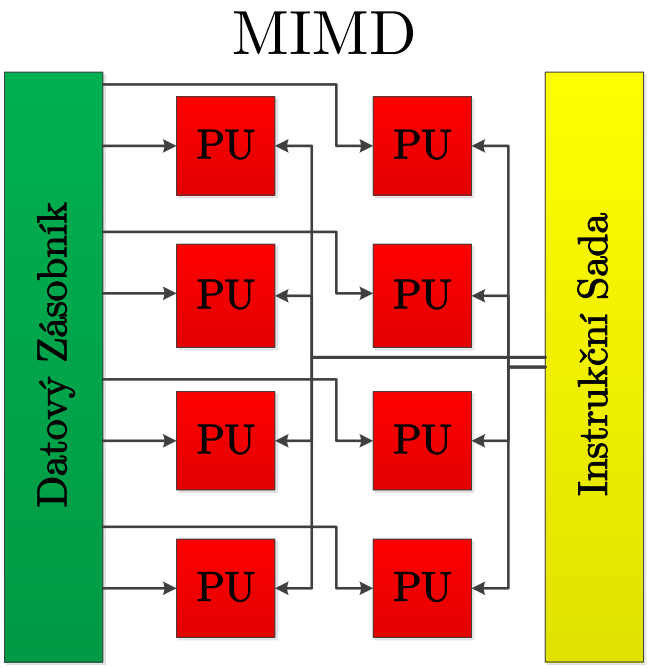
2.3.4 MIMD

Obsahuje více instrukčních proudů a více datových proudů (Multiple Instruction Multiple Data stream). Počítače používající architekturu MIMD mají několik výpočetních jednotek, které pracují asynchronně a nezávisle na sobě. V jeden okamžik různé výpočetní jednotky mohou pracovat na různých instrukcích s různými daty. Diagram této architektury je ukázán na obrázku 2.5. MIMD architektura může být použita ve velmi velkém množství aplikací např.(počítačem asistovaná výroba, simulace, modelování součástek nebo komunikační přepínače) [17]. V moderní době je

10 nejvýkonějších superpočítačů na světě je založeno na MIMD architektuře. Díky velkému rozsahu této architektury se ještě dělí do dvou podskupin [12].

SPMD (Single program, multiple data streams)
Několik nezávislých výpočetních jednotek souběžně zpracovává stejný program. Na rozdíl od SIMD, které pracuje na bodech synchronně tzv. *Lockstep*, pracují výpočetní jednotky v bodech nezávislých na sobě. SMPD je nejběžnější model paralelního programování [15].

MPMD (Multiple programs, multiple data streams)
Několik nezávislých výpočetních jednotek souběžně zpracovává více programů. Typická struktura tohoto systému je, že jeden mikroprocesor je zvolen jako host. Tento mikroprocesor zpracovává jeden program, který rozesílá data z ostatním mikroprocesorům pro speciální program a tyto ostatní mikroprocesory odesílají výsledky programu zpátky hostovi. Mezi zástupce této architektury patří například PS3.



Obr. 2.5: MIMD diagram.

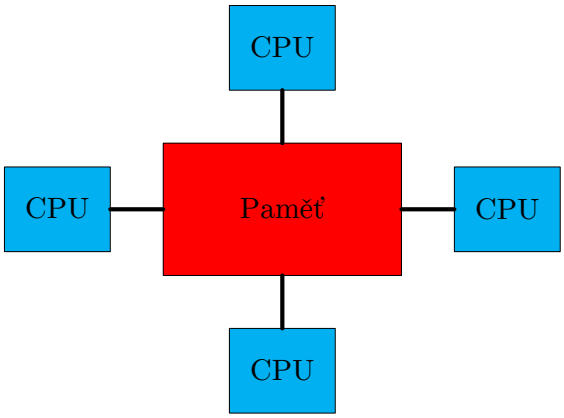
2.4 Typy pamětí

U počítačů využívající některý druh paralelního programování rozlišujeme tři obecné typy pamětí.

2.4.1 Sdílená paměť

První typ paměti se nazývá paměť sdílená. Systémy využívající sdílenou paměť tvoří velmi velkou část paralelního programování. Všechny procesory v PC sdílejí globální paměť. Komunikace a synchronizace mezi jednotlivými úkoly běžícími na různých mikroprocesorech je prováděna zápisem a čtením z globální sdílené paměti. Struktura sdílené paměti je graficky znázorněna na obrázku 2.6.

Při použití sdílené paměti se musíme potýkat s dvěma zásadními problémy. První problém je snižování výkonu při přístupu několika mikroprocesorů do sdílené paměti ve stejný čas. Tento problém se většinou řeší pomocí vyrovnávací paměti. Takové řešení bohužel, ale vede ke druhému problému a to je problém z koherencí paměti. Pokud jsou data ve vyrovnávací paměti uložena několikrát, vzniká problém s koherencí. Kopie dat ve vyrovnávací paměti jsou koherentní, pokud mají stejnou hodnotu. Problém nastane, když některý mikroprocesor přepíše hodnotu jedné kopie dat. Data se tak stanou nekonzistentní [17].



Obr. 2.6: Struktura sdílené paměti.

U sdílené paměti rozlišujeme tři typy přístupu mikroprocesorů k paměti a to je UMA, NUMA a COMA.

UMA (Uniform Memory Access)

V UMA má každý mikroprocesor přístup ke sdílené paměti přes propojovací síť. Toto spojení pracuje na stejném principu jako propojeni samostatného mikroprocesoru z pamětí. Mikroprocesory mají stejný přístupový čas do jakékoliv části sdílené paměti. Pro propojení mikroprocesorů ze sdílenou pamětí se může využít sběrnice, několik sběrnic nebo křížový přepínač. Tyto systémy často bývají nazývány symetrické multiprocesory. Mezi zástupce toho to systému bývají například multiprocesorové servery od firmy Oracle [6, 11].

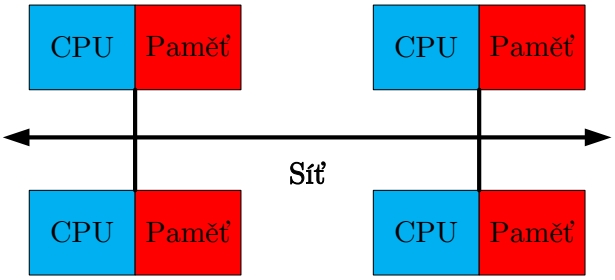
NUMA (Nonuniform Memory Access)
V systému NUMA má každý mikroprocesor přidělenou část sdílené paměti. Tato paměť má přiřazený jedinečný paměťový prostor. Takže jakýkoliv mikroprocesor může přistupovat k jakémkoliv pamětním prostoru pomocí její reálné adresy. Přístupová doba k této paměti závisí na vzdálenosti od mikroprocesoru. V NUMA systémech je používáno několik druhů spojení, jako je například stromová hierarchie [6, 11].

COMA (Cache-Only Memory Access)
Pracuje na stejném principu jako NUMA. Na rozdíl od NUMA je, ale paměťový prostor výhradně tvořen vyrovnávací pamětí(*cache*). COMA systém vyžaduje, aby data byly přeposlány mikroprocesoru, který je vyžaduje. Neexistuje žádná paměťová hierarchie, protože paměťový prostor je tvořen výhradně pamětí cache. Příklad tohoto systému je například přístroj The Kendall Square Research's KSR-1.

2.4.2 Distribuovaná paměť

Druhý typ paměti se nazývá paměť distribuovaná. Na rozdíl od sdílené paměti kde všechny mikroprocesory pracují ze sdíleným paměťovým prostorem, u paměti distribuované má každý mikroprocesor svůj osobní paměťový prostor. Výpočetní úlohy mohou pracovat pouze z lokální pamětí. Pokud jsou data potřebné k provedení úkolu uložena mimo lokální paměťový prostor, musí mikroprocesor komunikovat se vzdáleným mikroprocesorem, který má potřebná data uloženy ve svém lokálním paměťovém prostoru. Mikroprocesor nemůže sám přistoupit k datům uloženým lokální paměti jiného mikroprocesoru. Struktura distribuované sítě je graficky znázorněna na obrázku 2.7 [4, 21].

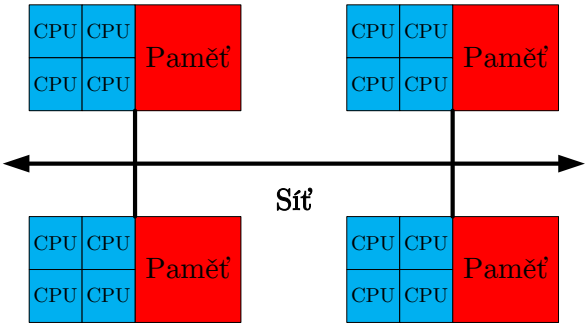
Architektura distribuované sítě je většinou složena z mikroprocesoru, paměťového bloku a libovolného spojovacího členu, který umožňuje komunikaci mikroprocesorů. Spojení je většinou realizováno z bodu do bodu (point to point) nebo nezávislá technologie zajistí přepínací síť. Pro spojení může být využitá například technologie *Ethernet*.



Obr. 2.7: Struktura distribuované paměti.

2.4.3 Distribuovaná sdílená paměť

Třetí typ pamětí se nazývá paměť distribuovaná sdílená. Jedná se o kombinaci dvou předcházejících typů. Tato architektura umožňuje fyzicky odděleným pamětím spolu komunikovat na jednom logicky adresovaném prostoru. Na rozdíl od sdílené paměti kde „sdílená“ znamená, že struktura obsahuje jednu fyzickou centrální paměť, u distribuované sdílené nám „sdílená“ vyjadřuje jeden sdílený adresový prostor. Stejná fyzická adresa na dvou odlišných mikroprocesorech vyjadřuje stejné místo v paměti. Struktura distribuované sdílené sítě je graficky znázorněna na obrázku 2.8 [21].



Obr. 2.8: Struktura distribuované sdílené paměti.

Distribuovaná sdílená paměť se skládá z mnoha na sebe nezávislých výpočetních bloků z vlastními lokálními paměťovými moduly, které jsou spojeny přes síť

z ostatními výpočetními bloky. Někdy se tato struktura nazývá *multicomputer*. Distribuovaná sdílená paměť se dá uskutečnit i softwarovou cestou a to většinou pomocí speciálního operačního systému nebo programovacích knihoven zaměřených na tento problém.
Mezi výhody této sítě například patří:

- Velmi dobré škálování systému.
- Je levnější než stejně vyřešený systém využívající několik mikroprocesorů.
- Průchodnost není problém, protože systém není omezován jedinou sběrnicí.
- Může zvýšit výkonost systému, zrychlením přístupové doby.

Mezi nevýhody této sítě například patří:

- Data jsou duplikována.
- Lokalizace vzdálených dat.
- Kolik dat je přeneseno v jediné operaci tzv.*Granularita*.

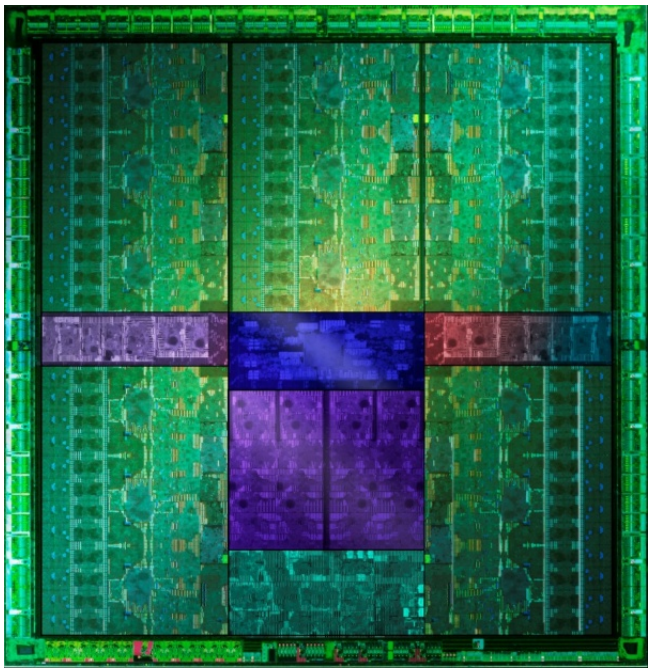
3 GRAFICKÉ KARTY NVDIA

V této kapitole se budeme věnovat grafickým kartám Nvdia. Popíšeme si čip Kepler GK110. Hlavně s zaměříme na strukturu a nové vlastnosti tohoto čipu. V této kapitole bude taky obecně popsáno vývojářské prostředí CUDA a její pod knihovnu CUFFT.

3.1 Kepler GK110

Složen z 7,1 miliardy tranzistorů, čip Kepler GK110 byl v době svého uvedení na trh největší a nejrychlejší mikroprocesor který byl kdy vyroben. Čip GK110 byl navržen z úmyslem toho aby se stal nejrychlejší mikroprocesor pro paralelní výpočty. Díky tomu našel velké uplatnění ve speciálních grafických kartách Tesla. Tento čip byl použit například u grafické karty GTX 680 [14].

Kepler GK110 zprostředkovává výpočetní výkon přesahující 1 TFLOP v aplikacích využívajících aritmetiky z dvojitou přesností. Dosahuje taky mnohem vyšší výpočetní efektivity vůči staršímu čipu Fermi. Další výhodou čipu GK110 je to že, mnohem lépe spotřebovává elektrickou energii. Výkon na Watt spotřeby je asi 3× lepší než u architektury Fermi. Na obrázku 3.1 můžeme vidět vnitřní uspořádání mikroprocesoru Kepler GK110 [1].



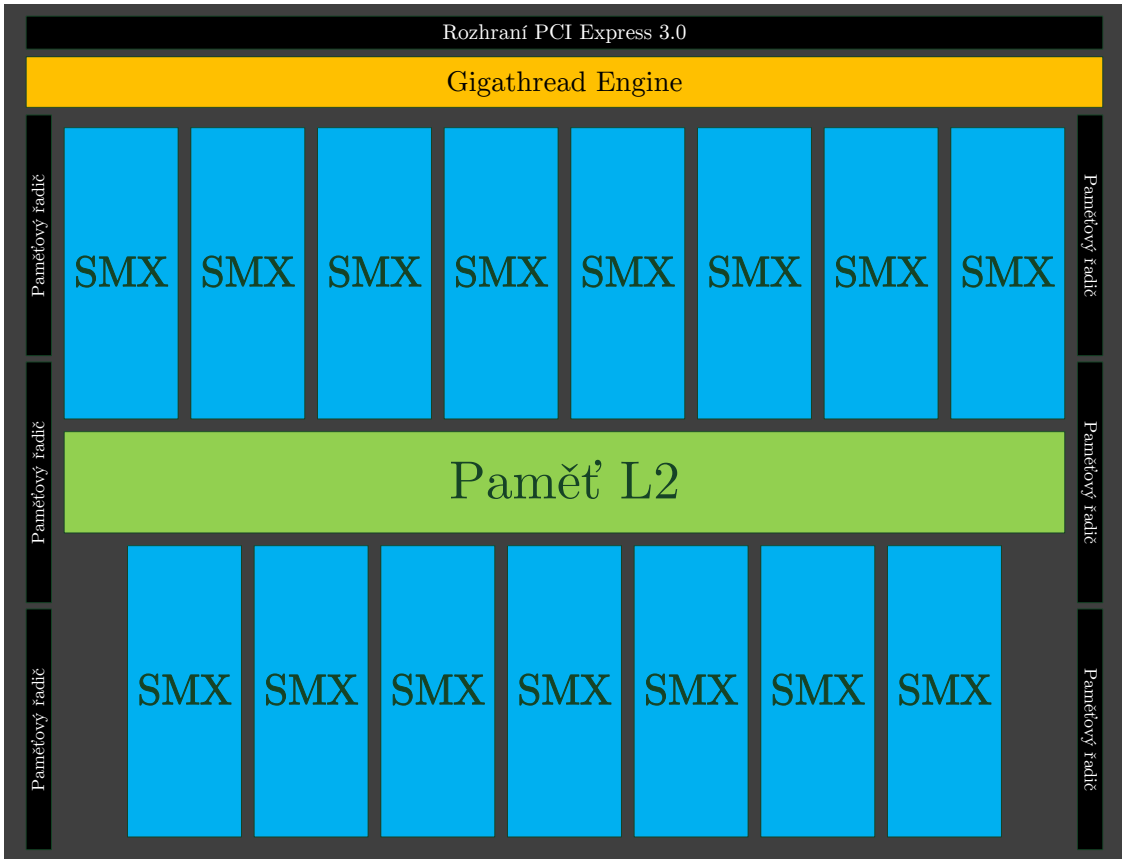
Obr. 3.1: Vnitřní uspořádání integrovaného čipu Kepler GK110 (převzato z [14]).

3.1.1 Architektura čipu Kepler GK110

Plný čip Kepler GK110 obsahuje 15 SMX jednotek a šest 64bitových paměťových řadičů. Blokové schéma celého čipu můžeme vidět na obrázku 3.2 [14].

Mezi hlavní body této architektury patří:

1. Nová architektura SMX mikroprocesoru.
2. Vylepšený paměťový subsystém, obsahující nové možnosti pro použití vyrovnávací paměti, zvyšující propustnosti paměti a rychlejší implementace DRAM I/O paměti.
3. Hardwarová podpora designu pro zpřístupnění nových programovacích možností.



Obr. 3.2: Celkové blokové schéma integrovaného čipu Kepler GK110.

V tabulce 3.1 je porovnání různých generací grafických mikroprocesorů Nvdia. Největší rozdíl architektury Kepler GK110 od starších čipů je dán počtem registrů na vlákno. Čip Kepler GK110 může použít až 255 registrů na vlákno oproti starším čipům, které mohou nasadit pouze 6 4 registrů na vlákno.

Tab. 3.1: Tabulka porovnání různých typů grafických čipů Nvdia.

	Fermi GF100	**Fermi GF104**	**Kepler GK104**	**Kepler GK110**
Maximální verze CUDA	2.0	2.1	3.0	3.5
Vlákna/Warp	32	32	32	32
Maximální Warp/Multiprocesor	48	48	64	64
Maximální počet vláken/Multiprocesor	1536	1536	2048	2048
Maximální počet bloků/Multiprocesor	8	8	16	16
32bitové registry/Multiprocesor	32 768	32 768	65 536	65 536
Maximální počet registrů/Vlákno	63	63	63	255
Maximální počet vláken/Blok	1024	1024	1024	1024
Rozložení sdílené paměti	16 kB 48 kB	16 kB 48 kB	16 kB 32 kB 48 kB	16 kB 32 kB 48 kB
Maximální rozměr X dimenze	2^{16-1}	2^{16-1}	2^{32-1}	2^{32-1}

3.1.2 Architektura průtokového multiprocesoru(SMX)

Čip Kepler GK110 přináší novou architekturu průtokového multiprocesoru zvanou „SMX“. Každá jednotka SMX obsahuje 192 CUDA jader s jednoduchou přesností (*single–precision*). Každé jádro má plně zřetězené aritmetické jednotky a jednotky pro zpracování plovoucí čárky. SMX obsahuje IEEE 754–2008 vyhovující operacím jak s jednoduchou tak dvojitou přesností a zahrnující slučovací a násobící operace zvané „FMA“. SMX čipu Kepler GK110 si taky zachovává jednotky pro speciální funkce(SFU) zavedené architekturou Fermi. Tyto jednotky se používají pro přibližný výpočet transcendentálních operací. SMX čipu Kepler GK110 obsahuje 8× víc jednotek něž čip Fermi GF110. Architekturu SMX je nakreslena na obrázku 3.3.

Podobně jako u čipu Fermi GF110, jádra využívají primární GPU clock spíše než 2× shader clock(Bavíme se o kmitočtu na kterém pracuje jádro GPU.). Zpracovávání výpočetních jednotek při vyšší frekvenci hodin umožňuje čipu dosáhnou požadovanou propustnost s menším počtem výpočetních jednotek. To způsobuje problém se spotřebou energie, tato problematika je vysvětlena v kapitole 1.1. Proto je architektura SMX čipu Kepler GK110 zaměřena spíše na více výpočetních jader běžících na nižší frekvencích [14].

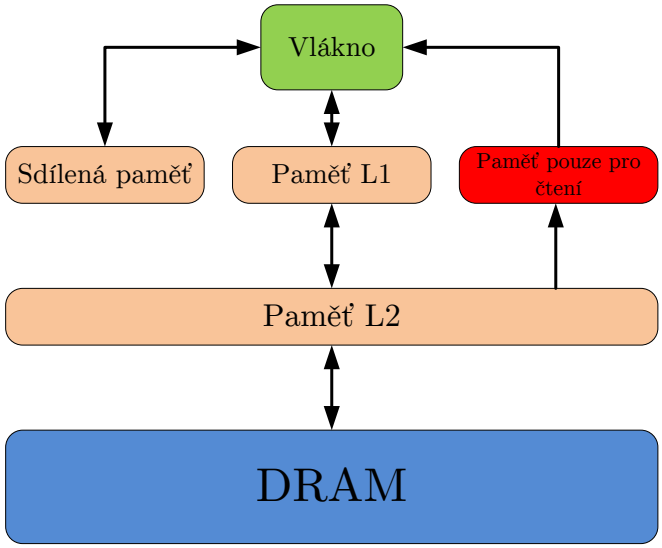
Obr. 3.3: Schéma SMX jednotky čipu Kepler GK110.

3.1.3 Paměťové subsystémy L1, L2

Hierarchie pamětí čipu Kepler GK110 je uspořádána podobně jako u čipu Fermi. Kepler architektura využívá pro zápis a čtení z dat jednotou paměť L1. Každá jednotka SMX obsahuje jeden blok paměti L1 a jeden blok sdílené paměti. U čipu Kepler GK110 může navíc překladač využít nový blok paměti, který je určen pouze pro čtení po dobu běhu funkce. Celou hierarchii paměti můžeme sledovat na obrázku 3.4.

Sdílená paměť a vyrovnávací paměť L1 64 kB

V Kepler architektuře má každá jednotka SMX dostupných 64 kB paměti. Tato paměť může být rozdělena jako 48 kB pro sdílenou paměť a 16 kB pro paměť L1



Obr. 3.4: Blokové schéma uspořádání jednotlivých částí paměti čipu Kepler GK110.

nebo 16 kB pro sdílenou paměť a 48 kB pro paměť L1. Rozdělit paměť v tomto poměru bylo možné už i architektury Fermi. Kepler dokáže rozdělit tyto paměti ještě na 32 kB pro sdílenou paměť a 32 kB pro paměť L1. Použitelná šířka pásma sdílené paměti je 256 B za jeden takt jádra [14].

Paměť určená pouze na čtení 48kB
Kepler architektura zavádí nový druh paměti, která je určená pouze na čtení za dobu trvání funkce. Velikost této paměti je 48kB. V předešlé architektuře Fermi byla tato paměť dostupná pouze texturovací jednotce. Kepler architektura zpřístupnila tuto paměť taky jednotce SMX. Paměť může být použita automaticky překladačem nebo zadána manuálně programátorem a to příkazem `const__restrict`.

Paměť L2
Čip Kepler GK110 obsahuje 1536kB dedikované vyrovnávací paměti L2 a to je dvojnásobek od starší architektury Fermi. Paměť L2 slouží jako hlavni sjednocovací bod pro všechny jednotky SMX. Stará se všechny nahrávací,ukládací a texturové požadavky a zároveň poskytuje efektivně rychlý přenos dat pro GPU [14].

3.1.4 Nové technologie čipu Kepler

Architektura Kepler obsahuje nové technologie, které mají za úkol zvýšit využití, rychlost a flexibilitu GPU. Mezi hlavní technologie uvedené čipem Kepler patří:

- **Dynamický Paralelismus** – Přidává GPU možnost generovat si pro sebe vlastní práci, tuto práci synchronizovat a kontrolovat plánování práce pomocí svého vlastního hardwaru bez využití CPU. Díky dynamickému paralelismu dokáže GPU flexibilně reagovat na různé situace které mohou nastat při zpracování programu [14].

- **Hyper–Q** – Hyper–Q umožňuje CPU s více jádry zahájit práci na jediné GPU současně, tím dramaticky zvyšuje využití GPU a snižuje dobu nečinnosti CPU. Hyper–Q zvyšuje celkový počet pracovních front(připojení) mezi hostem a GPU a to až na 32 souběžných hardwarových připojení.

- **Řídící jednotka souřadnicové sítě** – Dynamický paralelismus potřebuje pro svoji funkci rychlou a pokročilou správu souřadnicové sítě. Nová jednotka pro správu souřadnicové sítě(*GMU*) spravuje a upřednostňuje zpracování mřížek přímo na GPU. GMU jednotka může pozastavit odesílání nových mřížek, taky může mřížky suspendovat dokud není GPU schopné je znovu zpracovávat.

3.2 CUDA

CUDA je platforma určená pro paralelní programování. CUDA zastává i funkci API a byla vytvořená firmou Nvdia. CUDA umožňuje vývojáři využít grafickou kartu pro obecné výpočty, které by se jinak nedaly na grafické kartě zpracovat. Tento postup se nazývá *General-purpose computing on graphics processing units* (GPGPU). Platforma CUDA je software, který dává přímý přístup programátorovi k instrukční sadě grafické karty. Platforma CUDA je navržená tak aby mohla pracovat s několika programovacími jazyky jako je například C, C++, Python nebo Fortran. Díky této vlastnosti je velmi univerzální. CUDA taky podporuje programovací rámce jako je například OpenACC a OpenCL [1, 4, 8].

3.2.1 Základní principy CUDA

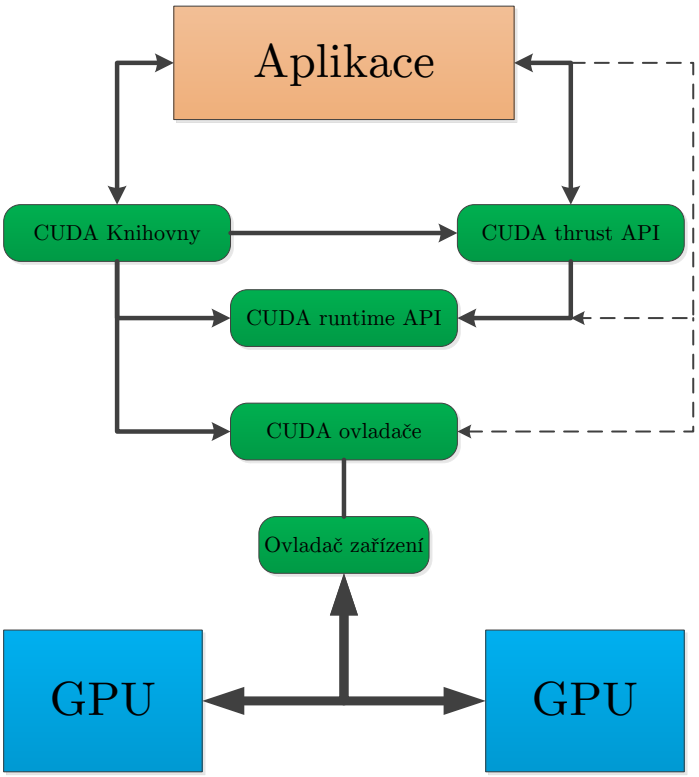
Při práci z platformou CUDA musíme zvažovat šest základních principů. Tyto principy jsou:

1. **GPU podporující platformu CUDA jsou oddělené prvky, které jsou nainstalovány v PC**.
   Ve většině případů jsou GPU spojeny ze zbytkem PC pomocí vysokorychlostní sběrnice jako je například PCI-E. Většina PC může obsahovat dvě až čtyři GPGPU. Toto číslo, ale závisí na schopnostech PC a to například kolik má PCI-E slotů, napájení nebo chlazení komponentů. Každé GPU je oddělené zařízení, které pracuje asynchronně na procesoru. Komunikaci mezi zařízeními

zprostředkovává například vysokorychlostní sběrnice PCI-E. CUDA k přenosu dat používá dva různé druhy:

(a) Explicitní datový přenosu pomocí příkazu `cudaMemcpy()`. Nejčastěji používán pro přenos dat mezi vektory.

(b) Implicitní datový přenos ve známé paměti. Tento přenos synchronizuje paměťový prostor uživatelské zařízení s paměťovým prostorem GPU. Nepotřebuje pro svoji funkci zásah programátora. Například aplikace může nahrát datový set na uživatelském zařízení, zde ho namapovat pro GPU, která ho pak sama zpracuje. Některé méně výkonné GPU využívají paměti uživatelského zařízení pro zvýšení vlastního výkonu. Strukturu softwaru CUDA můžeme sledovat na obrázku 3.5.



Obr. 3.5: Struktura softwaru CUDA platformy.

2. **GPGPU pracuje v odděleném paměťovém prostoru od zbytku PC.** Až na několik velmi slabých GPU má každá GPU svoji vlastní paměť RAM, která byla navržena tak aby poskytovala GPU mnohem větší paměťovou propustnost než klasická paměť PC. Moderní paměťové systémy GPGPU dosahuji propustnost až 160–200 GB/s narozdíl od klasických pamětí, které dosahují

propustnosti jen okolo 8–20 GB/s [8].

3. **CUDA programy používají kernely**.
   Kernely jsou pod-rutiny vytvořené uživatelským zařízením aby byly uskutečněny na GPU. Je důležité poznamenat že kernely nejsou funkce, takže nemohou vrátit hodnotu. Většina aplikací pracujících na GPGPU tráví většinu času v jedné nebo maximálně několika výpočetních rutin. Transformace těchto rutin na kernely může výrazným způsobem akcelerovat aplikaci. Kernel je definován pomocí deklarace `_global_`.
4. **Volání kernelů je asynchronní**.
   Díky své asynchronní povaze nemohou kernely vracet hodnotu. Pro lepší efektivitu může být vytvořena fronta kernelů, která udržuje GPGPU v provozu po co nejdelší dobu. GPGPU potřebuje vědět kdy je kernel nebo fronta kernelů dokončena proto potřebuje nějakou formu synchronizace. Dvě nejběžnější formy synchronizace jsou:
   (a) Explicitní voláni `cudaThreadSynchronize()`, které funguje jako bariéra. Přikazuje uživatelskému zařízení aby počkalo až se všechny kernely dokončí.
   (b) Uskutečnění datového přenosu pomocí `cudaMemcpy()` jako `cudaThreadSynchronize()`.
5. **Základní jednotka práce GPU je vlákno**.
   Každé vlákno je ze softwarového hlediska samostatné a nezávislé na ostatních vláknech. Každé vlákno pracuje jako kdyby mělo svůj vlastní mikroprocesor z registry a jedinečnou identitou. Kolik vláken může zároveň existovat definuje hardware. Práci vláken kontroluje zabudovaný ovladač, který se nachází na každé grafické kartě. přepínání mezi jednotlivými vlákny probíhá tak rychle, že ze softwarového hlediska je prakticky bez výkonového snížení. Kernel by měl pracovat co z největším množstvím vláken [8, 18].
6. **Největší oblast sdílené paměti na GPU je nazýván globální paměť**.
   Obvykle dosahuje velikosti GB RAM, většina dat pro aplikace je umístěna v této paměti. Globální paměť využívá techniky slučováni (*coalescing*). Tato technika kombinuje několik datových přenosů do jedné velké přenosové operace aby tato operace mohla dosáhnou nejvyšší přenosové rychlosti. Obecně nejvyššího výkonu nastane pokud jsou paměťové přístupy sloučeny do bloků o velikosti 128 kB. Přístupová doba globální paměti je až 600× pomalejší než přístup k registru. CUDA programátoři by si měli uvědomit ,že i když paměťová propustnost GPGPU může připadat jako velmi vysoká, zdaleka nedosahuje výpočetních možností GPGPU, proto je velmi důležité znovu využívání dat v programu [8].

4 APLIKACE PARALELNÍCH VÝPOČTŮ

V této kapitole se nejprve seznámíme se základními signálovými operacemi. Následně si probereme použitý typ algoritmů. Poté provedeme srovnání výpočetního výkonu CPU a GPU na Lineárním 2D konvolučním filtru a Cannyho detektoru hran, co se týče rychlosti zpracování různých typů výpočetních mikroprocesorů a v závislosti na velikosti vstupních dat.

4.1 Základní operace se signály

4.1.1 Konvoluce

Konvoluce dvou spojitých signálů $x(t)$ a $h(t)$ je operace, která je vyjádřena vztahem:

$$\begin{aligned} y(t) &= h(t) * x(t) = \int_{-\infty}^{\infty} h(\tau)x(t-\tau)\mathrm{d}\tau \\ &= x(t) * h(t) = \int_{-\infty}^{\infty} x(\tau)h(t-\tau)\mathrm{d}\tau. \end{aligned} \tag{4.1}$$

Na obrázku 4.1 je graficky ukázán výpočet konvoluce. Konvoluce je jedna ze základních operací se dvěma signály. Jestli $h(t)$ vyjadřuje impulzní charakteristiku kmitočtového filtru, tak konvoluce koná operaci kmitočtové filtrace vstupního spojitého signálu $x(t)$ kmitočtovým filtrem s impulzní charakteristikou $h(t)$ [16].

4.1.2 Korelace

Korelace je velmi podobná konvoluci. Korelace vyjadřuje míru podobnosti průběhu dvou signálů. Korelace $\gamma_{x,h}(\tau)$ dvou spojitých signalů $x(t)$ a $h(t)$(pokud integrál konverguje) se rovná:

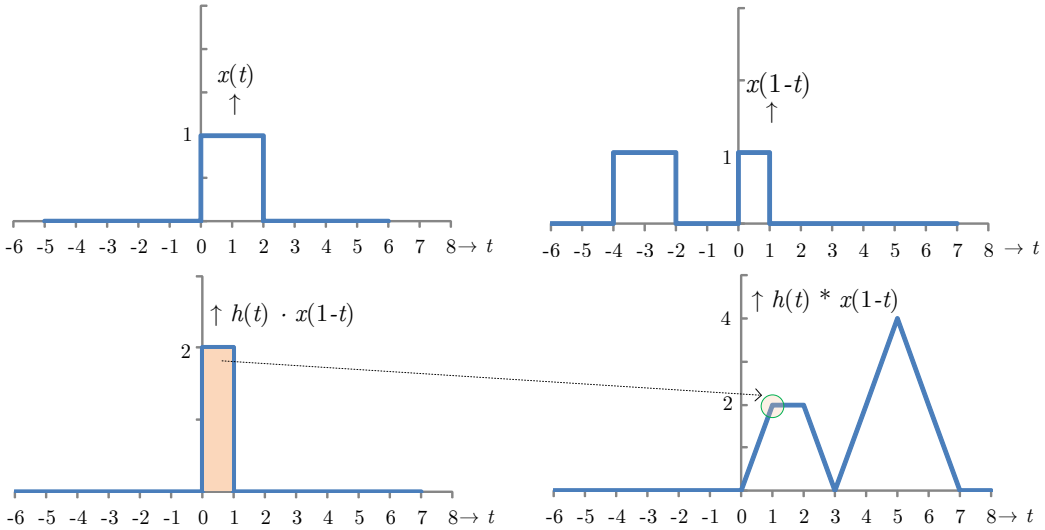
$$\gamma_{x,h}(\tau) = \int_{-\infty}^{\infty} x(t)h(t+\tau)\mathrm{d}\tau, \;\; \tau \in \mathbb{R}. \tag{4.2}$$

Korelace je graficky znázorněna na obrázku 4.2

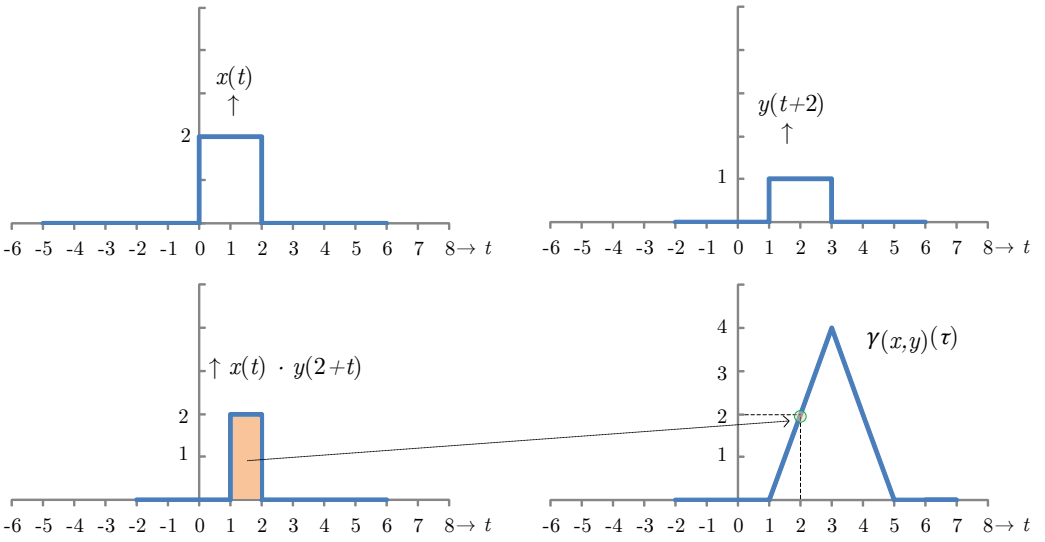
4.1.3 Lineární diskrétní konvoluce

Lineární diskrétní konvoluci posloupností $x_1[n]$ a $x_2[n]$ myslíme posloupnost $y[n]$ danou vztahem:

$$y[n] = x_1[n] * x_2[n] = x_2[n] * x_1[n] = \sum_{m=0}^{N-1} x_1[m]x_2[n-m]. \tag{4.3}$$

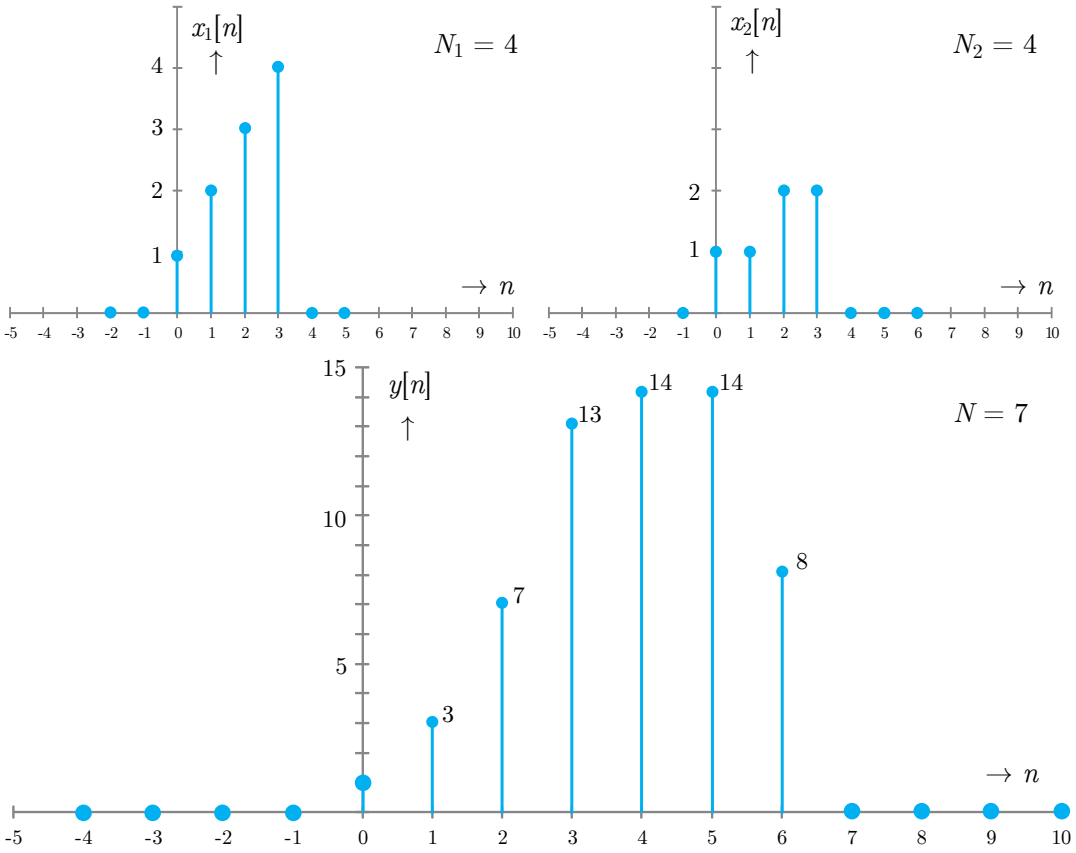


Obr. 4.1: Příklad konvoluce dvou signálů.



Obr. 4.2: Příklad korelace dvou signálů.

Pokud má posloupnost $x_1[n]$ délku N_1 a posloupnost $x_2[n]$ délku N_1, potom výsledná konvoluce $y[n]$ má délku $N = N_1 + N_2 - 1$. Vytvoření lineární konvoluce $y[n]$ posloupností $x_1[n]$ a $x_2[n]$ je nakreslena na obrázku 4.3.



Obr. 4.3: Lineární diskrétní konvoluce $y[n] = x_1[n] * x_2[n]$ [16].

4.1.4 Diskrétní Fourierova transformace

Diskrétní Fourierova transformace je použitá pokud se při Fourierově transformaci pracuje s diskrétními hodnotami v čase tak i v hodnotách. To má za důsledek využíti pro výpočet digitálního mikroprocesoru. Mikroprocesor pracuje pouze z čísly s konečnou přesností a jeho výpočetní kapacita je omezena velikostí paměti [16, 2]. Z toho vyplývá, že standardní Fourierova transformace

$$S(f) = \int_{-\infty}^{\infty} s(t)e^{-j2\pi ft}\mathrm{d}t, \tag{4.4}$$

musí být upravena. Nahrazením signálu $s(t)$ posloupností $s(nT)$, která reprezentuje vzorkování signálu limitované na konečnou množinu čísel N kvůli konečné velikosti

paměti. Výpočty pak poskytují čísla $S(f)$ definována jako

$$S(f) = \sum_{n=0}^{N-1} s(nT)e^{-j2\pi fnT}. \tag{4.5}$$

Kvůli omezeném výpočetním výkonu, může mikroprocesor poskytovat výsledky jenom pro omezený počet frekvencí f. Proto je logické vybrat násobky určitého frekvenčního skoku Δf. Z toho vyplývá

$$S(k\Delta f) = \sum_{n=0}^{N-1} s(nT)e^{-j2\pi k\Delta fnT}. \tag{4.6}$$

4.2 OpenCV a použité funkce

4.2.1 OpenCV

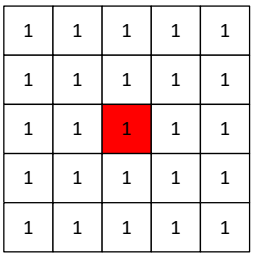
OpenCV je knihovna s volným otevřeným zdrojovým kódem. Knihovna je napsána pomocí programovacích jazyků C a C++ a může pracovat pod operačním systémem Linux, Windows nebo Mac OS X. Může ovšem pracovat i s ostatními programovacími jazyky jako je Python,Ruby nebo Matlab [3].

OpenCV byla navržena pro výpočetní efektivitu. Klade velmi velký důraz na aplikace probíhající v reálném čase. OpenCV je napsán v optimalizované verzi programovacího jazyku C a může využít více jádrové procesory. Obsahuje taky podporu knihovnu CUDA. Díky této podpoře dokáže využít výpočetního výkonu grafických karet NVDIA.

Cíl knihovny OpenCV je zpřístupnit knihovnu zaměřenou na počítačovou grafiku, která bude snadná na ovládání, ale budou se dát pomocí ní vytvořit složité programy. Knihovna OpenCV obsahuje přes 500 funkcí z mnoha oborů počítačové grafiky, jako je například inspekce továrních výrobků, kalibrace kamery, úpravy fotek, robotiky nebo uživatelské rozhraní. Protože počítačová grafika a strojové učeni je spolu většinou provázáno. OpenCV obsahuje plnou knihovnu strojového učení (MLL). Knihovna se zaměřuje na rozpoznávání statistických modelů a shromažďování informací [3].

4.2.2 Lineární 2D konvoluce `filter2D`, `gpu::filter2D`

Funkce konvoluce je definována podle toho jaké konvoluční jádro (*kernel*) je aplikováno. Konvoluční jádro je fixní řada koeficientů z přesně určeným úchytným bodem, který je většinou ve středu konvolučního jádra. Na obrázku 4.4 je ukázáno jedničkové konvoluční jádro o rozměrech 5×5. Červený čtverec ukazuje místo úchytného bodu.



Obr. 4.4: Jedničkový konvoluční filtr 5×5.

Hodnota konvoluce v určitém bodě je vypočítána tak, že se nejprve úchytný bod jádra přesune na vypočítávaný pixel a zbytek jádra překryje lokální pixely v blízkosti vypočítávaného pixelu. Pro každý korespondující bod obrazu teď máme dvě hodnoty: hodnotu jádra a hodnotu pixelu. Tyto hodnoty vynásobíme a součet výsledku všech bodů jádra pak zapíšeme do vypočítávaného pixelu. Tento proces je opakován pro každý pixel obrazu tím, že je konvoluční jádro přesouváno po obraze [3].

Proces lze samozřejmě vyjádřit matematicky formou vzorce. Pokud nadefinujeme obraz jako $R(x, y)$ a konvoluční jádro jako $F(i, j)$ (kde $0 < i < M_i - 1$ a $0 < j < M_j - 1$), a úchytný bod je umístěn na(a_i,a_j) v konvolučním jádru, pak konvoluce $H(x, y)$ je definována jako:

$$H(x, y) = \sum_{i=0}^{M_i-1} \sum_{j=0}^{M_j-1} I(x + i - a_i, y + j - a_j)G(i, j). \tag{4.7}$$

Ze vzorce vyplývá, že počet operací se rovná počtu pixelů obrazu vynásobeným počtem pixelů v konvolučním jádru. Tento proces je v knihovně OpenCV realizován funkcí `void filter2D(InputArray src, OutputArray dst, int ddepth, InputArray kernel, Point anchor, double delta, int borderType)` kde:

- **`src`** = Vstupní obraz.
- **`dst`** = Výstupní obraz.
- **`ddepth`** = Hloubka pole výstupního obrazu, pokud je -1 tak je stejná jako vstupního.
- **`kernel`** = Konvoluční jádro.
- **`anchor`** = Úchytný bod konvolučního jádra, pokud je (-1, -1) tak je úchytný bod ve středu konvolučního filtru.
- **`delta`** = Volitelná hodnota, která má být přičtena vypočteným pixelům, před tím než se uloží do `dst`.
- **`borderType`** = Metoda extrapolace pixelů.

OpenCV má ve své knihovně i speciální verzi této funkce, která pro vypočet používá grafické karty: `void gpu::filter2D(const GpuMat& src, GpuMat& dst, int`

ddepth, constMat& **kernel**, Point **anchor**, int borderType, Stream& **stream**=Stream::Null()).

Proměnné v této funkci mají stejný význam jako v předešlé funkci. Rozdíl je, že zpracovávané obrázky musí být nahrané v paměti grafické karty, chybí proměnná `delta` a funkce navíc obsahuje proměnnou `stream` určenou pro asynchronní zpracování [3].

4.2.3 Cannyho hranový detektor `Canny, gpu::Canny`

Cannyho detektor hran je založen v knihovně OpenCV na funkci Laplaceova transformace, která ke své funkci využívá Laplaceův operátor [3]:

$$\text{Laplace}(f) \equiv \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{4.8}$$

Rozdíl mezi Cannyho detektorem hran a Laplaceovou transformací je, že první derivace jsou vypočítány v x a y a potom zkombinovány do čtyř směrových derivátů. Body kde jsou tyto směrové deriváty maxima jsou kandidáti pro sestavení hran. Cannyho algoritmus se snaží tyto kandidáty sestavit do kontur. Kontury jsou vytvořeny aplikováním hysterezního prahu na pixely. To znamená že jsou zde dva prahy dolní a vrchní. Pokud má pixel gradient větší než horní hysterezní práh, tak je zvolen jako pixel hrany. Na rozdíl pokud má pixel gradient menší než dolní hysterezní práh tak je odmítnut. Jestli je gradient pixelu mezi prahy, bude pixel akceptován pouze pokud bude sousedit s pixel s gradientem větším než horní hysterezní práh [3]. Tento proces je v knihovně OpenCV realizován funkcí: `void Canny(InputArray` **`image`**`, OutputArray` **`edges`**`, double` **`threshold1`**`, double` **`threshold2`**`, int` **`apertureSize`**`=3, bool` **`L2gradient`**`=false)` kde:

- **`image`** = Vstupní obraz, musí být 8bitový a jedno kanálový.
- **`edges`** = Výstupní mapa hran obrazu.
- **`threshold1`** = Dolní hysterezní práh.
- **`threshold1`** = Horní hysterezní práh.
- **`aperatureSize`** = Velikost otvoru pro Sobelův operátor.
- **`L2gradient`** = Ukazatel jestli má být provedena přesnější forma výpočtu obrazového gradientu $L_2 = \sqrt{\frac{\mathrm{d}I}{\mathrm{d}x}^2 + \frac{\mathrm{d}I}{\mathrm{d}y}^2}$.

Tak jako v případu funkce `filter2D` obsahuje knihovna OpenCV speciální verzi funkce `Canny`, která dokáže využít výpočetního výkonu grafické karty:
`void gpu::Canny(const GpuMat&` **`image`**`, GpuMat&` **`edges`**`, double` **`thresh1`**`, double` **`thresh2`**`, int` **`apperturesize`**`=3, bool` **`L2gradient`**`=false)`

4.3 Příklad použitého kódu

V této sekci si popíšeme části kódu použitého pro testování rozdílu mezi výpočty na CPU a GPU. Program je napsaný jazykem C++. V uživatelském prostředí Microsoft Visual Studio. Při psaní kódu se využilo funkcí zahrnutých v knihovně OpenCV verze 4.9.11 . Funkce programu probíha v následujících bodech:

- Inicializuje knihovnu CUDA. Tento krok se musí provést, jinak by před prvním výpočtem pomocí knihovny CUDA probíhala inicializace, která probíhá zhruba 200 ms [10].
- Podle volby v prvním menu nahraje obrázek zvolené velikosti. Obrázek je nahrávaný jako černobílý a to kvůli požadavkům použitých funkcí. Výběr obrázku probíhá ze šesti velikostí:
  - 256×256 pixelů
  - 512×512 pixelů
  - 1024×1024 pixelů
  - 1920×1080 pixelů
  - 3840×2160 pixelů
  - 7680×4320 pixelů

```
1   system("cls"); //Refresh konzole.
2   cout << "Vyberte obrazek k zpracovani : " << endl;
3   cout << "1) Nacist obrazek 256x256 " << endl;
4   cout << "2) Nacist obrazek 512x512 " << endl;
5   cout << "3) Nacist obrazek 1024x1024 " << endl;
6   cout << "4) Nacist obrazek 1920x1024" << endl;
7   cout << "5) Nacist obrazek 3840x2160 " << endl;
8   cout << "6) Nacist obrazek 7680x4320" << endl;
9   cin >> volba;
10
11  //Menu vybrání velikosti obrázku.
12
13  switch (volba) {
14  case 1:
15      src = imread("Clovek.jpg", CV_LOAD_IMAGE_GRAYSCALE); // Příkaz pro
            nahrání obrázku
16      if (!src.data) {
17          return -1; // Pokud obrázek neexistuje, program se ukončí.
18      }
```

- Po nahrání obrázku se dostane program do druhého menu kde probíhá volba výpočetní jednotky. Volba probíhá ze tří možností:
  - Výpočet pomocí CPU. Program vypočítá funkce `filter2D` a `Canny` pomocí CPU. Tyto funkce se provedou na obrázku 5×. Po uskutečnění těchto funkcí se na obrazovku konzole vypíše čas průběhu jednotlivých funkcí. Tato procedura proběhne 29×, následně se z těchto 29 uskutečnění vypočítá aritmetický průměr průběhu obou funkcí, který se opět

vytiskne na obrazovku.

```
1  void CPU()
2  {
3
4      uchyt = Point(-1, -1); //Střed konvolučního jádra.
5      delta = 0;
6      ddepth = -1;
7      int tik = 0;
8      int ind = 0;
9      int can = 0;
10
11     matice_size = 5;
12     matice = Mat::ones(matice_size, matice_size, CV_32F) / (float)(
           matice_size * matice_size); // Vytvoření konvolučního jádra 5x5.
13     for (tik = 0; tik < 29; tik++) {
14
15         clock_t start1 = clock();
16         for (ind = 0; ind < 5; ind++) {
17             filter2D(src, dst, ddepth, matice, uchyt, BORDER_DEFAULT);
                   //Aplikování funkce filter2D
18         }
19         clock_t end1 = clock();
20
21         clock_t start2 = clock();
22         for (can = 0; can < 5; can++) {
23             Canny(src, dst, 35, 200, 3); //Aplikování funkce Canny.
24         }
25         clock_t end2 = clock();
26
27         if (volba2 == 1) {
28             cout << "Cas vypracovani ulohy 2D konvolucniho filtru:  " <<
                       casovac(start1, end1) << " ms " << endl;
29             cout << "Cas vypracovani ulohy Cannyho detektoru hran: " <<
                       casovac(start2, end2) << " ms " << endl;
30         }
31         celkcas2d = (casovac(start1, end1) + celkcas2d);
32         celkcascan = (casovac(start2, end2) + celkcascan);
33     }
34     prumcas2d = celkcas2d / 30;
35     prumcascan = celkcascan / 30;
36     if (volba2 == 1) {
37         cout << "Prumerny cas vypracovani 2D konvolucniho filtru " <<
               prumcas2d << " ms " << endl;
38         cout << "Prumerny cas vypracovani Cannyho detektoru hran " <<
               prumcascan << " ms " << endl;
39         getch();
40     }
```

– Výpočet pomocí GPU. Průběh tohoto procesu probíhá podbně jako u výpočtu pomocí CPU. V některých věcech se ovšem algoritmus liší. Obraz se musí nejprve převést do paměti GPU, kde se alokuje paměťové místo pro vstupní i výstupní obraz. To se provede pomocí funkce `gpu::GpuMat dsrc(src)` a `gpu::GpuMat ddst`. Po nahrání obrazů do paměti GPU se provedou funkce `gpu::filter2D, gpu::Canny`. Výsledný obraz je pak z paměti GPU převeden do paměti PC pomocí funkce `Mat dst(ddst)`.

– Srovnání výpočtu CPU a GPU. Nejprve se provede výpočet pomocí CPU a následně pomocí GPU. Po skončení výpočtu se zobrazí průměrná doba obou výpočtů a zrychlení GPU oproti CPU.

- Ukázka kódu funkce `main`.

```
1 int main(int argc, char** argv)
2 {
3     cv::gpu::setDevice(0); //Inicializace CUDA knihovny pro výpočet na GPU.
4     Menu1(); // Inicializace menu pro výběr obrázku.
5     Menu2(); // Inicializace menu pro výběr výpočetní jednotky.
6     getch();
7     return 0;
8 }
```

4.4 Výsledky měření

Výsledky měření jsou shrnuty v této sekci. Sekce je rozdělena do tří podsekcí. V první podsekce jsou výsledky měření pro všechny výpočetní jednotky pro různé rozlišení obrazů. Druhá podsekce se zabývá jak velké proběhlo zrychlení výpočtů po zapojení GPU u různých sestav. Poslední podsekce se zaměřuje na závislost zrychlení výpočtu funkce `filter2D` pomocí GPU na rozlišení obrazu.

4.4.1 Měření jednotlivých výpočetních jednotek

Pomocí programu, který byl popsán v minule sekci byly naměřeny hodnoty průměrného času provedení funkcí `filter2D` a `Canny` pro různé rozlišení obrazu. tyto hodnoty byly zaneseny do tabulek 4.1 a 4.2.

Tab. 4.1: Tabulka naměřených průměrných časů výpočtu funkce `filter2D` pro různé rozlišení obrazu

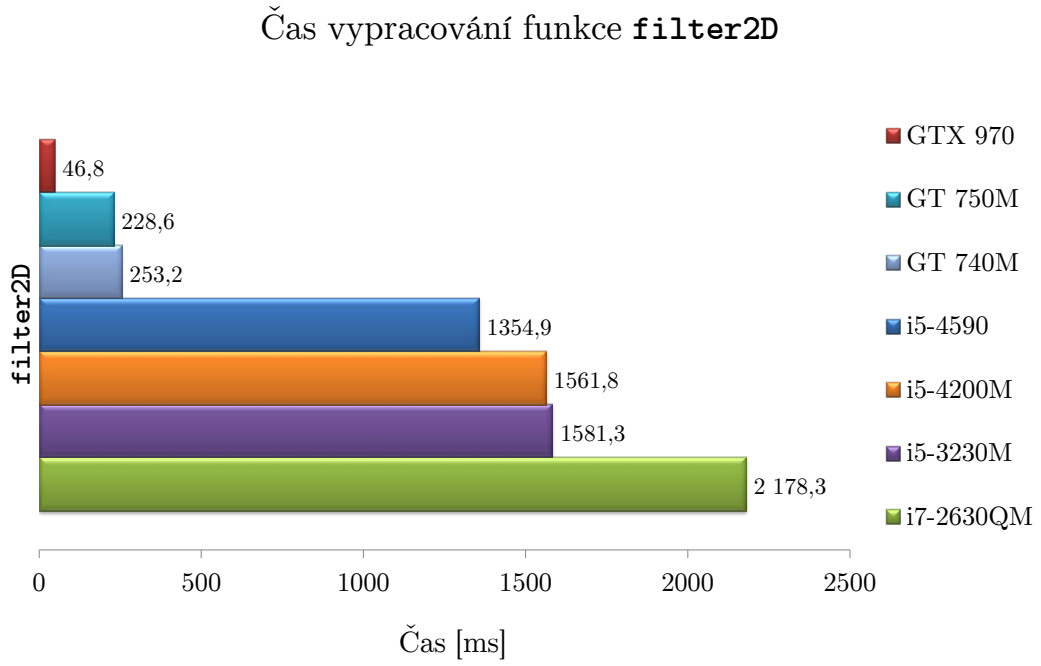
Rozlišení obrazu [px]	**Pruměrný čas výpočtu funkce `filter2D` [ms]**						
	i5-4590	**GTX 970**	**i7-2630QM**	**i5-3230M**	**GT 750M**	**i5-4200M**	**GT 740M**
256×256	13,5	2,7	17,4	15,0	8,3	14,2	5,7
512×512	46,8	3,7	63,9	48,5	12,6	54,6	13,3
1024×1024	175,9	8,2	275,0	201,1	43,7	206,9	33,8
1920×1080	342,5	14,0	487,2	397,7	74,4	403,0	62,9
3840×2160	1354,9	46,8	2178,3	1581,3	228,6	1561,8	253,2
7680×4320	5386,4	175,3	8344,7	6331,6	891,7	6462,1	1024,4

Z tabulek můžeme vyčíst, že u funkce `filter2D` je čas potřebný pro výpočet funkce pomocí GPU mnohem menší než u CPU. Funkce `filter2D` je pro zapojení paralelního výpočtu velmi vhodná a to kvůli velkému počtu nezávislých výpočtů, které probíhají současně. Z tabulky 4.1 jde vidět, že i u slabých GPU jako GT

Tab. 4.2: Tabulka naměřených průměrných časů výpočtu funkce `Canny` pro různé rozlišení obrazu

Rozlišení obrazu [px]	**Průměrný čas výpočtu funkce `Canny` [ms]**						
	i5-4590	**GTX 970**	**i7-2630QM**	**i5-3230M**	**GT 750M**	**i5-4200M**	**GT 740M**
256×256	11,0	4,5	14,8	12,5	6,3	10,9	5,8
512×512	34,1	13,0	52,7	43,7	29,6	38,4	25,7
1024×1024	109,4	25,0	168,1	134,3	107,3	127,4	103,4
1920×1080	193,7	37,0	274,5	227,0	196,7	224,1	188,7
3840×2160	907,0	111,0	1282,6	1057,4	765,7	1054,0	876,0
7680×4320	3014,4	392,3	4306,7	3521,5	3086,1	3511,5	3369,2

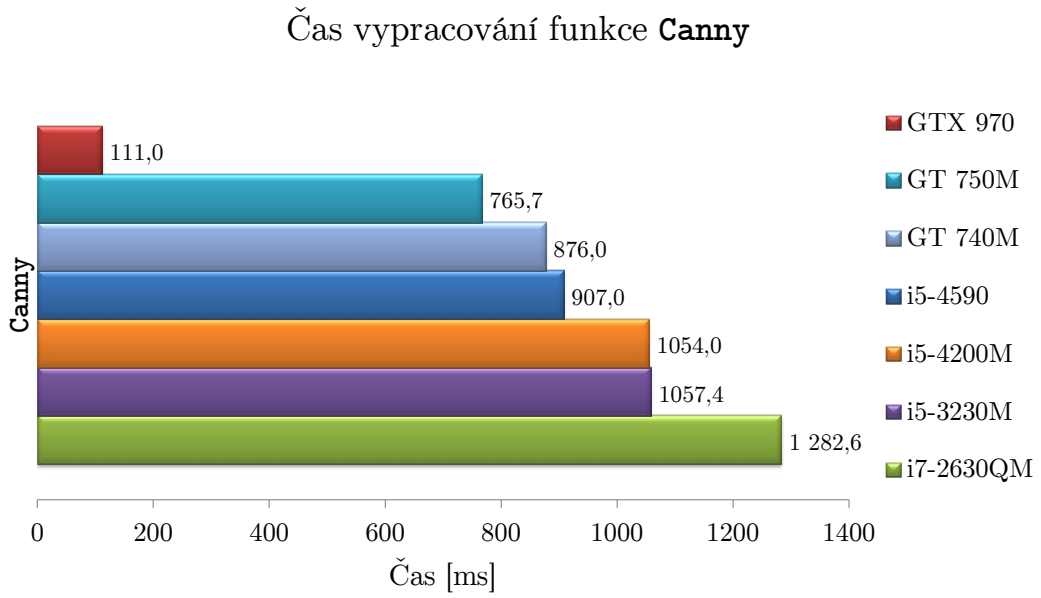
740M dojde k velkému zrychlení. Zrychlení grafické karty GTX 970 je obrovské. Z tabulky `filter2D` vyplývá, že i když procesor i7-2630QM obsahuje 8 virtuálních jader, kvůli své zastaralé architektuře není rychlejší než i5-3230M nebo i5-4200M. Čas vypracování funkce `filter2D` pro rozlišení obrazu 3840x2160 px je zobrazeno na grafu 4.5.



Obr. 4.5: Průměrný čas vypracování funkce `filter2D` u rozlišení 3840x2160 px.

Naopak pro funkci `Canny` je mnohem těžší získat zrychlení zapojením paralelních výpočtů. To je způsobeno tím, že druh výpočtu funkce `Canny` je nevhodný pro paralelní výpočty. Funkce `Canny` obsahuje velkou část sériových výpočtů, na kterou nelze nasadit paralelní zpracování. U slabších grafických karet jako je GT 740M a GT 750M je zrychlení oproti CPU minimální. U grafické karty GTX 970 je zlepšeni

větší a to díky rychlejším paměťovým přenosům a větším výpočetním výkonu GPU. Čas vypracování funkce `Canny` pro rozlišení obrazu 3840x2160 px je zobrazeno na grafu 4.6.



Obr. 4.6: Průměrný čas vypracování funkce `Canny` u rozlišení 3840x2160 px.

4.4.2 Zrychlení sestav po zapojení GPU

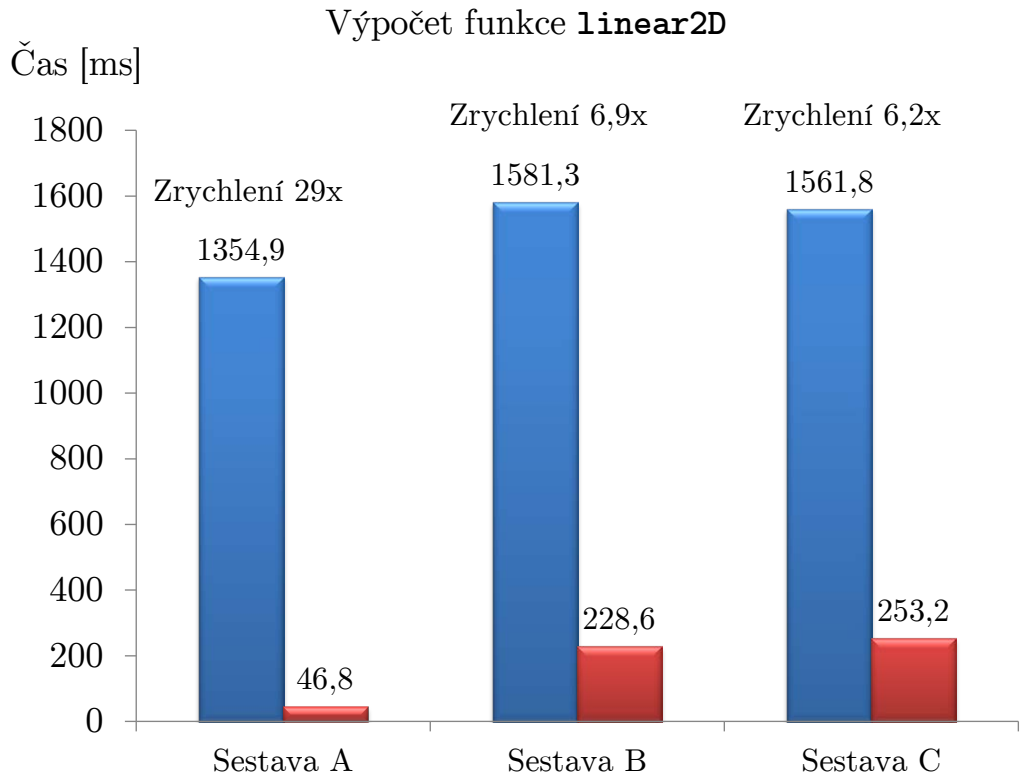
Pomocí programu uvedeném v předešlé sekci byly otestovány tři různé počítačové sestavy. Tyto sestavy jsou uvedeny v tabule 4.3. V tabulce je uveden CPU a GPU sestavy a dosažené zrychlení u funkcí `filter2D` a `Canny` po provedení výpočtu pomocí GPU. Sestava A je výkonný stolní počítač. Sestavy B a C jsou přenosné počítače značky Asus a Lenovo. Všechny počítače pracují na operačním systému Windows 10. Na grafu 4.7 je znázorněno zrychlení sestav po nasazení GPU. Funkce `filter2D`

Tab. 4.3: Tabulka sestav a jejich zrychlení po zapojení GPU.

Sestava	**Sestava A**	**Sestava B**	**Sestava C**
Procesor	i5-4590	i5-3230M	i5-4200M
Grafická karta	GTX 970	GT 750M	GT 740M
Zrychlení GPU 3840×2160 `filter2D`	29,0×	6,9×	6,2×
Zrychlení GPU 3840×2160 `Canny`	8,2×	1,4×	1,2×

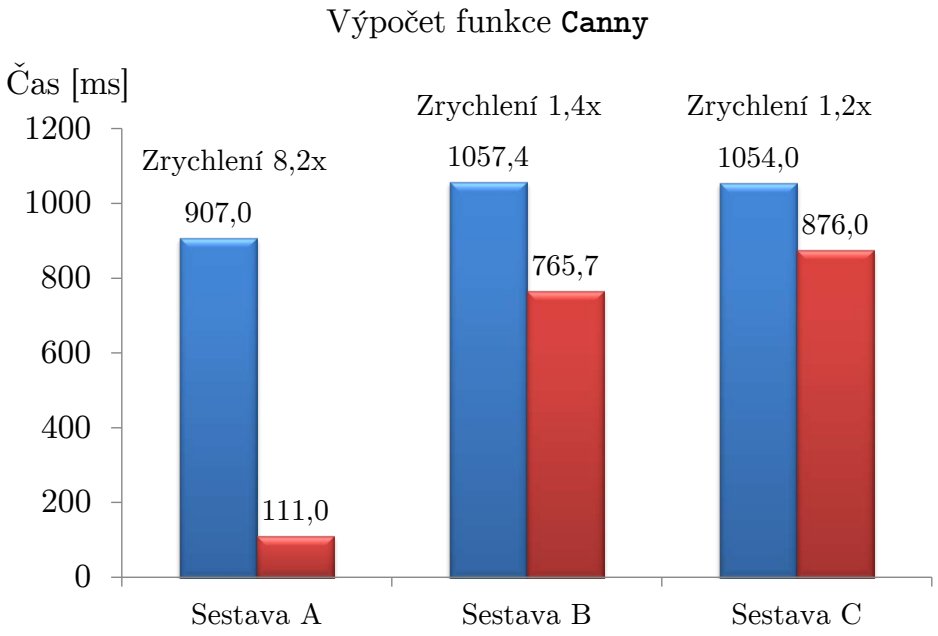
probíhala na obrázku s rozlišením 3840x2160 px. U sestavy A nastalo nejvyšší snížení doby výpočtu funkce `filter2D` a to 29×. Grafická karta GTX 970 je velmi

výkonná, obsahuje velké množství CUDA jader. Díky tomu dokáže vypočítat efektivně velké množství paralelních výpočtů. Grafická karta GTX 970 taky poskytuje rychlejší paměťové přesuny pro data. U sestav B a C byla délka výpočtu po nasazení GPU 6,9× a 6,2× kratší než u CPU. Z těchto výsledků lze usoudit, že nasazování GPU pro výpočet funkce `filter2D` je opodstatněné u každé sestavy, která obsahuje grafickou kartu NVDIA.



Obr. 4.7: Zrychlení výpočtu funkce `filter2D` po nasazení GPU.

Na grafu 4.8 je znázorněno zrychlení sestav po nasazení GPU pro výpočet funkce `Canny`. Výpočet je prováděn pro obraz s rozlišením 3840x2160 px. Výsledky zrychleni sestav se velmi liší od zrychlení funkce `filter2D`. Protože funkci `Canny` je velmi obtížné paralelizovat, protože obsahuje velkou část sériových výpočtů. Výrazné zrychlení po nasazení GPU nastává pouze u sestavy A. Důvodem tohoto zrychlení je to, že GTX 970 poskytuje větší výpočetní výkon a mnohem rychlejší paměťový přenos informací než CPU. Doba výpočtu sestavy A pomocí GPU byla 8,2× menší než u CPU. U sestav B a C nastalo zrychlení jen malé. Z těchto výsledků lze usuzovat, nasazování GPU pro výpočet funkce `Canny` je možné, ale zásadní rychlostní nárůst nabízí pouze u výkonějších GPU.



Obr. 4.8: Zrychlení výpočtu funkce `Canny` po nasazení GPU.

4.4.3 Závislost zrychlení funkce `filter2D` pomocí GPU na rozlišení obrazu.

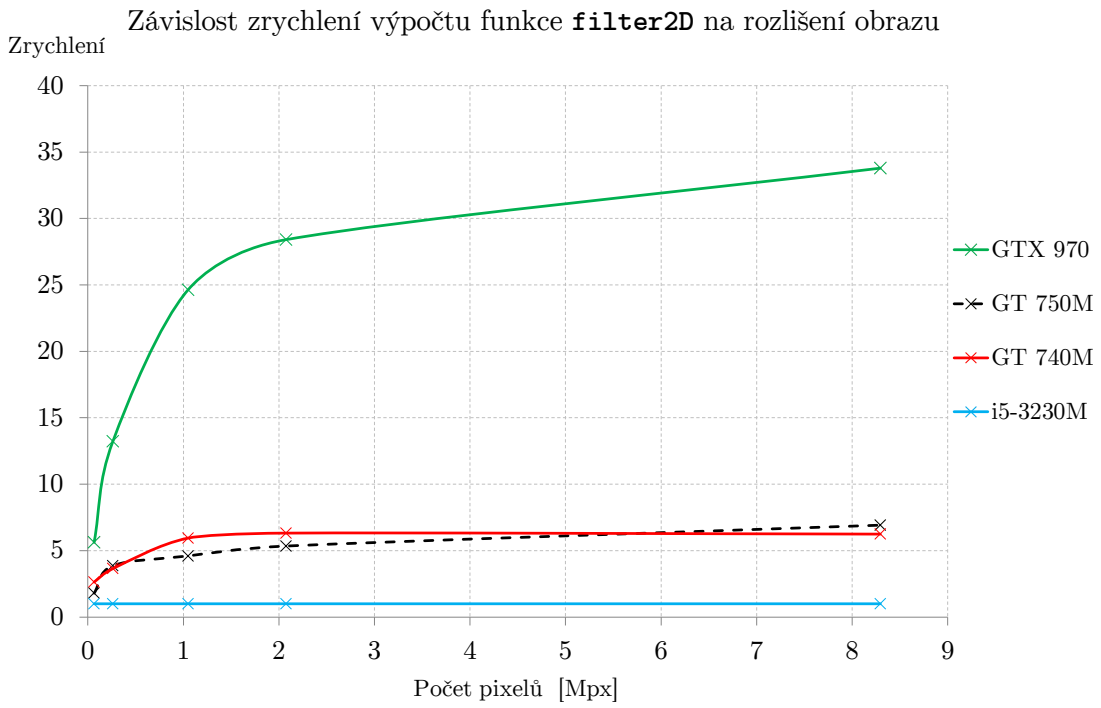
Zrychlení GPU oproti CPU není vždy stejné, mění se společně s rozlišením zpracovávaného obrazu. V tabulce 4.4 jsou uvedeny hodnoty zrychlení výpočtu pomocí GPU vůči procesoru i5-3230M pro různé rozlišení obrazu. Z tabulky vyplývá, že pro

Tab. 4.4: Zrychlení funkce `filter2D` vůči procesoru i5-3230M pro různé rozlišení obrazu.

Rozlišení obrazu [px]	**Zrychlení vůči procesoru i5-3230M**		
	GTX 970	**GT 750M**	**GT 740M**
256×256	5,6×	1,8×	2,6×
512×512	13,2×	3,9×	3,6×
1024×1024	24,6×	4,6×	5,9×
1920×1080	28,4×	5,3×	6,3×
3840×2160	33,8×	6,9×	6,2×
7680×4320	36,1×	7,1×	6,2×

malé velikosti rozlišení je zrychlení vůči CPU menší než pro obrazy velkého rozlišení. Tuto skutečnost můžeme graficky sledovat na grafu 4.9. Na grafu jsou znázorněny

násobky zrychlení GPU v závislosti na rozměru obrazu. Graf obsahuje čtyři křivky. Tři pro grafické karty a křivka pro procesor i5-3230M, která slouží jako orientační. Z grafu lze vyčíst, že pro malé rozlišení obrazu nedochází po nasazení GPU k maximálnímu zrychlení. Děje se tak z důvodu toho, že pro malé rozlišení obrazu není k dispozici dostatečný počet operací pro GPU, aby plně vytížila svoje CUDA jádra pro paralelní výpočty a dosáhla tak maximálního výkonu. Tuto skutečnost lze sledovat na křivce GTX 970, která dosahuje pro rozlišení obrazu 256x256 pouze zrychlení 5,6×. Ovšem pro rozlišení obrazu 1920x1080 je zrychlení už 28,4×. Grafická karta GT 740M dosahuje větších výkonů při malých rozlišení obrazů oproti GT 750M kvůli tomu, že testovaná GPU GT 740M měla paměť GDDR5 a GPU GT 750M pomalejší paměť DDR3. Ve vysokých rozlišeních je rychlejší GT 750M díky svým vyšším frekvencím.



Obr. 4.9: Graf závislosti zrychlení výpočtu funkce `filter2D` pomocí GPU na rozlišení obrazu.

5 ZRYCHLENÍ VÝPOČTU RYCHLÉ KONVOLUCE

Tato kapitola je zaměřena na zrychleni rychlé konvoluce pomocí nasazení paralelních výpočtů. Pro zrychlení byly využity knihovny CUDA a cuFFT. Zrychlení výpočtů je provedeno u FFT,IFFT a násobení výstupních vzorků dvou FFT transformací.

5.1 Rychlá konvoluce

Rychlá konvoluce používá princip, že násobení ve frekvenční oblasti koresponduje s konvolucí v časové oblasti. Vstupní signál je transformován do frekvenční oblasti pomocí DFT transformace, vynásoben kmitočtovou odezvou filtru a následně pomocí Inverzní DFT transformace transformován zpátky do časové oblasti [5, 2, 19].

5.1.1 Rychlá kruhová konvoluce

Když,

$$\sum_{m=0}^{N-1}(s(m)(h(n-m))modN) = r(n) \equiv R(k) = S(k)H(k) \tag{5.1}$$

$r(n)$ může být vypočítáno jako $r(n) = \text{IDFT}[\text{DFT}[s(n)]\text{DFT}[h(n)]]$

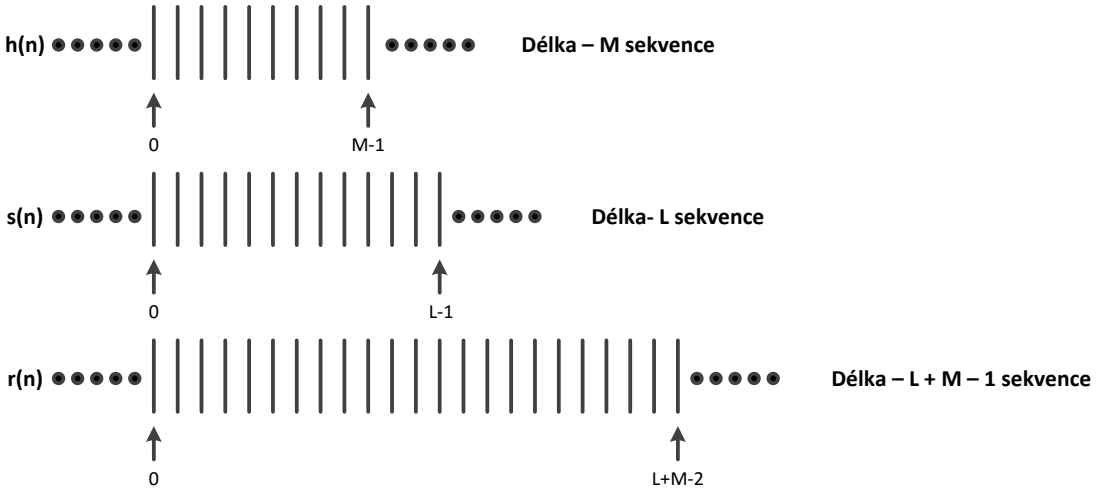
Nároky:

- Přímá metoda
  - N^2 komplexních násobků.
  - $N(N-1)$ komplexních sčítanců.
- Metoda pomocí FFT
  - 3 FFT + N násobků(Pokud je $H(k)$ předzpracovaný výpočtem pouze 2 FFT + N násobků)
  - $N + \frac{3N}{2}\log_2 N$ komplexních násobků.
  - $3(N\log_2 N)$ komplexních sčítanců.

5.1.2 Rychlá lineární konvoluce

DFT produkuje kruhovou konvoluci. Pro lineární konvoluci musíme vynulovat okraje sekvencí, tak aby se kruhový posun vždy přesouval přes nulu. K dosáhnutí lineární konvoluce použitím rychlé konvoluce, musíme vynulovat okraje DFT o délce $N \geq L + M - 1$. Tato skutečnost je vyobrazena na obrázku 5.1 [5, 19]. Zvolíme nejmenší vhodné N (většinou se jedná o nejmenší mocninu dvou větší nebo rovnou k $(L + M - 1)$)

$$r(n) = \text{IDFT}_N[\text{DFT}_N[s(n)]\text{DFT}_N[h(n)]] \tag{5.2}$$



Obr. 5.1: Uspořádání sekvencí při rychlé lineární konvoluci.

5.2 Použité knihovny

V této práci byly použity dvě hlavní knihovny pro aplikaci FFT. První knihovna byla FFTW a druhá knihovna byla knihovna cuFFT, která je součástí knihoven CUDA[9]. Dále byla použita knihovna Libsndfile. Tato knihovna je zaměřena na čtení a zápis vzorků ze zvukových souborů.

5.2.1 FFTW

FFTW je obsáhlá kolekce rychlých rutin programovacího jazyku C pro počítaní diskrétních Fourierových transformací DFT a dalších speciálních případů[9].
Mezi hlavní vlastnosti knihovny FFTW patří:

1. FFTW počítá DFT komplexních dat, reálných dat, pro reálná data se sudou nebo lichou paritou dat, (tyto symetrické transformace jsou většinou známy jako diskrétní kosinové nebo sinové transformace), nebo diskrétní Hartleyovi transformace (DHT) reálných dat.
2. Vstupní data mohou mít svévolnou délku, FFTW zapojuje $O(n \log n)$ algoritmus pro všechny délky, zahrnující i prvočísla.
3. FFTW podporuje libovolné počty dimenzí vstupních dat.
4. FFTW podporuje SSE, SSE2, AVX, Altivec a MIPS instrukční sady.
5. FFTW zahrnuje paralelní (více-vláknové) transformace.

5.2.2 cuFFT a CUFFTW

Knihovny CUFFT a CUFFTW jsou součástí balíčku CUDA. Knihovna cuFFT je navržená pro grafické karty Nvdia, tak aby bylo možné dosáhnout co nejvyššího výkonu ve výpočetních operacích typu FFT. Knihovna cuFFTW je dodávaná jako nástroj pro uživatele FFTW knihovny, tak aby mohli snadno využít výpočetního výkonu grafických karet Nvdia [13].

FFT je algoritmus typu rozděl a panuj pro efektivní výpočet diskrétních Fourierových transformací reálného i komplexního tvaru. Furierova transformace je jedna z nejdůležitějších matematických operací v počítačové fyzice a zpracování signálu. Knihovna cuFFT umožňuje uživateli jednoduše a rychle pracovat s FFT v přehledném prostředí. cuFFT dokáže plně využít výkonu grafických karet Nvdia v paralelních výpočtech funkcí FFT [13].

CUFFT podporuje různé varianty FFT algoritmů. Mezi některé tyto možnosti patří například:

1. Vysoce optimalizované algoritmy pro datové signály o různých datových dimenzích.
2. Algoritmus $O(n \log n)$ kde n vyjadřuje velikost dat použitých pro FFT.
3. Podpora Vstupních a výstupních dat v oboru reálných čísel i komplexních.
4. 1D, 2D a 3D Transformace FFT.
5. Operace s 32 i 64 bitovou plovoucí čárkou.
6. Datové schémata kompatibilní s FFTW.
7. Libovolné intra-dimenzí a inter-dimenzí přesuny prvků.
8. Transformace až 128 milionů prvků u operací s 32 bitovou plovoucí čárkou a 64 milionů prvků u operací s 64 bitovou plovoucí čárkou.
9. API knihovny FFTW.
10. Provádění příkazů ve streamech, umožnující asynchronní výpočty a přesun dat.

5.3 Ukázka kódu

Tato část je zaměřena na popis kódu programu. Program je napsaný jazykem C++ v programu Visual studio 2012. Program se skládá ze tří části. První část je přečtení a zapsání zvukového souboru. Druhá část je vypočítaní rychlé konvoluce pomocí FFTW knihovny a třetí část vypočítání rychlé konvoluce pomocí knihovny cuFFT. Pro práci ze zvukovým souborem je použita knihovna Libsndfile. Knihovna nejdříve načte soubor a informace o něm zapíše do předem přichystané struktury `sfinfo`.

Následně se alokuje paměť pro zapsání vzorků do polí. Posledním krokem je zapisováni vzorků do polí pomoci funkce `sf_readf_double`. Ukázka tohoto procesu je ukázána v kódu níže.

```
1     //Načtení struktury zvukového souboru do sfinfo.
2     SoundFile = sf_open("op.wav", SFM_READ, &sfinfo);
3     if (SoundFile == NULL) {
4         printf("Failed to open the file.\n");
5         getch();
6     }
7     //Alokování arrayí pro vzorky.
8     Samples = new double[sfinfo.channels * sfinfo.frames];
9     Samples2 = new double[sfinfo.channels * sfinfo.frames];
10
11    // Zapisování vzorků do polí.
12    sf_readf_double(SoundFile, Samples, sfinfo.frames);
13    sf_readf_double(SoundFile, Samples2, sfinfo.frames);
```

5.3.1 FFTW

V části výpočtu FFTW se nejdříve alokuje paměť pro pole, které jsou potřebné pro FFT. Po vytvoření polí se vytvoří plány, které slouží k optimalizaci a řízení FFT transformace pro zadanou konfiguraci. FFT se začne provádět pomocí příkazu `fftw_execute`. V našem případě se FFT provede 5000× a to z důvodu lepšího zpracování výsledků do grafů a tabulek hlavně pro menší délky FFT transformací.

```
1     //Alokování paměti pro pole potřebné pro FFT transformaci.
2     clock_t start1 = clock();
3     Samples = (double*)fftw_malloc(sizeof(double) * n);
4     out = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * n / 2 + 1);
5     Samples2 = (double*)fftw_malloc(sizeof(double) * n);
6     out2 = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * n / 2 + 1);
7
8     //Vytvoření plánu pro FFT transformaci.
9     plan = fftw_plan_dft_r2c_1d(n, Samples, out, FFTW_MEASURE);
10    plan2 = fftw_plan_dft_r2c_1d(n, Samples2, out2, FFTW_MEASURE);
11
12    //5000x provedení FFT transformace podle předem vytvořeného plánu.
13    clock_t start2 = clock();
14    for (int i = 0; i < 5000; ++i) {
15        fftw_execute(plan);
16        fftw_execute(plan2);
17    }
```

Násobení výsledků dvou FFT se opět provádí 5000× a to ze stejného důvodu. Násobení komplexních prvků se provádí klasicky pomocí cyklu `for`.

```
1         for (int i = 0; i < 5000; ++i) {
2             for (int i = 0; i < n / 2 + 1; i++) { // Provedení násobení FFT
                  transformací 5000x.
3                 Konec[i][0] = out[i][0] * out2[i][0] - out[i][1] * out2[i][1];
4                 Konec[i][1] = out[i][1] * out2[i][0] + out[i][0] * out2[i][1];
5             }
```

Následně se opět alokuje paměť pro výstupní data IFFT. IFFT se provádí pouze jednou a to z důvodu toho, že FFTW ničí vstupní pole při provádění IFFT z komplexních do reálných hodnot. Posledním krokem je normalizace výstupních dat. Normalizace se provede tak, že každá hodnota vzorku se podělí délkou transformace n.

```
1     vysledek = (double*)fftw_malloc(sizeof(double) * n);
2
3     plan3 = fftw_plan_dft_c2r_1d(n, Konec, vysledek, FFTW_MEASURE);
4
5     clock_t start4 = clock();
6     fftw_execute(plan3); // Provedení IFFT transformace
7     clock_t end4 = clock();
8
9     fftw_destroy_plan(plan3);
10
11    clock_t start7 = clock();
12    for (int i = 0; i < 5000; ++i) {
13        for (int i = 0; i < n; i++) { //Normalizace výsledků provedená 5000x.
14            vysledek[i] = vysledek[i] / n;
15        }
16    }
```

5.3.2 cuFFT

Výpočet rychlé konvoluce pomocí knihovny cuFFT je proveden podobně jako u knihovny FFTW. Odlišnosti nastávají v názvech funkcí při provádění FFT. U prováděni FFT pomocí GPU je důležité, aby se vzorky zapsané v polích v paměti PC, zapsaly do paměti GPU. Postup je ukázán v algoritmu níže.

```
1     // Nahrání vzorků z paměti PC do paměti grafické karty.
2     cudaMalloc((void**)&SamplesDevice, sfinfo.frames);
3     cudaMemcpy(SamplesDevice, SamplesHost, sfinfo.frames, cudaMemcpyHostToDevice
          );
4     cudaMalloc((void**)&SamplesDevice2, sfinfo.frames);
5     cudaMemcpy(SamplesDevice2, SamplesHost2, sfinfo.frames,
          cudaMemcpyHostToDevice);
```

Oproti knihovně FFTW kde je řešení násobení výstupních dat dvou FFT řešeno pomocí cyklu `for`. V aplikaci pomocí GPU můžeme plně využít paralelní sily GPU. Funkce použité pro násobení vzorků v polích jsou uvedeny v algoritmu níže.

```
1  // Komplexní škálování
2  static __device__ __host__ inline cufftDoubleComplex Scale(cufftDoubleComplex a,
       float s)
3  {
4      cufftDoubleComplex c;
5      c.x = s * a.x;
6      c.y = s * a.y;
7      return c;
8  }
9
10 // Komplexní násobení
11 static __device__ __host__ inline cufftDoubleComplex Mul(cufftDoubleComplex a,
       cufftDoubleComplex b)
```

```
12 {
13     cufftDoubleComplex c;
14     c.x = a.x * b.x - a.y * b.y;
15     c.y = a.x * b.y + a.y * b.x;
16     return c;
17 }
18
19 // Komplexní bodové násobení
20 static __global__ void ComplexPointwiseMulAndScale(cufftDoubleComplex* c,
       cufftDoubleComplex* a, cufftDoubleComplex* b, int size, float scale)
21 {
22     const int numThreads = blockDim.x * gridDim.x;
23     const int threadID = blockIdx.x * blockDim.x + threadIdx.x;
24
25     for (int i = threadID; i < size; i += numThreads) {
26         c[i] = Scale(Mul(a[i], b[i]), scale);
27     }
28 }
```

Oproti FFTW knihovně se zároveň provádí násobení i normalizace v jedné funkci. FFT a násobení vzorků se opět provádí 5000×.

5.4 Výsledky zrychlení výpočtu rychlé konvoluce

Při zrychleni výpočtu rychlé konvoluce se zaměříme na části výpočtu, které lze velmi dobře zrychlit paralelními výpočty. Prvním krokem je zrychlení výpočtu FFT a zpětné IFFT pomocí zapojení zapojení knihovny cuFFT. Druhým krokem je zrychlení výpočtu násobení výstupních komplexních prvků dvou FFT a to zapojením knihovny CUDA. Pro porovnání bude použita sestava A uvedena v tabulce4.3. Výsledky byly přečteny z programu 5× a byl na ně proveden aritmetický průměr pro omezení možnosti nepřesnosti jednoho měření.

5.4.1 Zrychlení výpočtu FFT,IFFT

Tato část je zaměřená na zrychlení výpočtu FFT a IFFT. Budou zde porovnávány 3 druhy provedení FFT a IFFT. První provedení je pomocí knihovny FFTW. Druhé provedení je opět pomocí knihovny FFTW, ale tato knihovna je zavedena pomocí knihovny cuFFTW. CuFFTW je přídavná knihovna knihovny cuFFT a umožňuje používat určité funkce knihovny FFTW. Poslední provedení je pomocí knihovny cuFFT, která výpočet FFT a IFFT uskutečňuje pomocí GPU. FFT a IFFT je prováděna pro různé velikosti n. Výsledky jednotlivých provedení jsou uvedeny v tabulkách 5.1.

Tab. 5.1: Tabulky naměřených hodnot pro různé provedení FFT a IFFT.

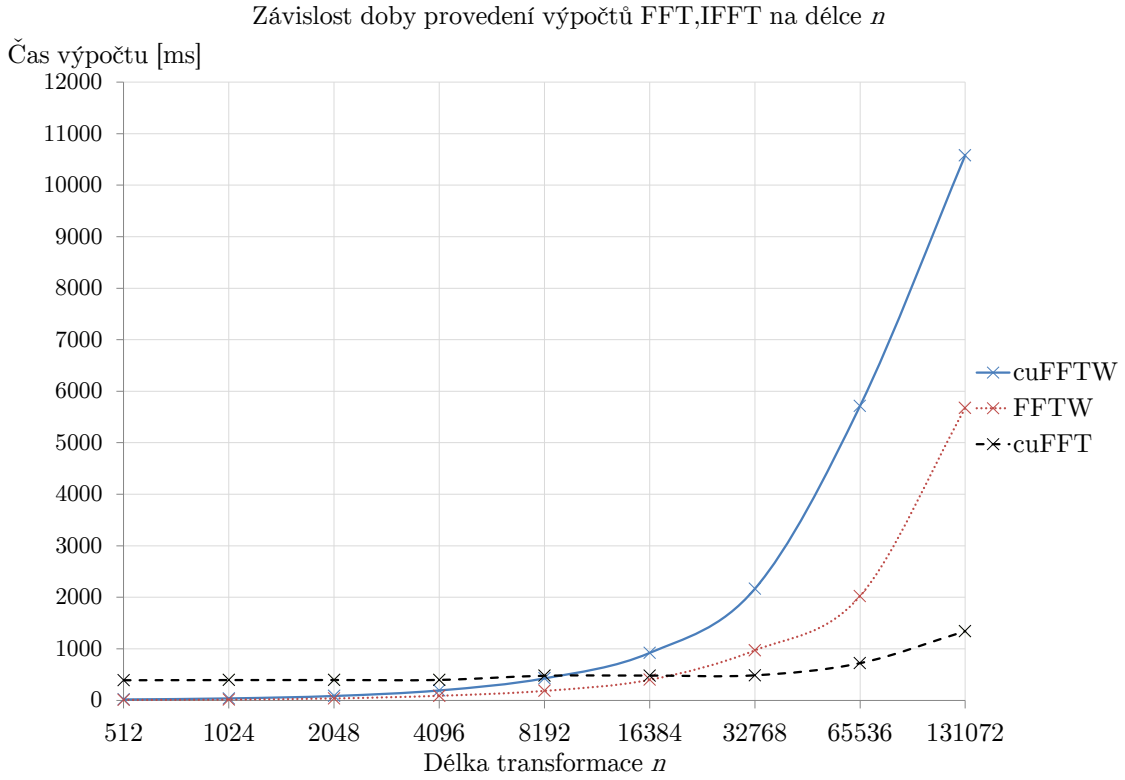
Výpočet FFT,IFFT [ms]	**Délka transformace n**							
	128	**256**	**512**	**1024**	**2048**	**4096**	**8192**	**16384**
cuFFTW	3,2	7,6	17	37,4	84,4	191,8	426,3	920,8
FFTW	2,8	4,2	7,8	15,1	37,3	88,2	183,6	398,6
cuFFT	390,4	391,8	390	393,6	394,4	396,3	477	481,2

Výpočet FFT,IFFT [ms]	**Délka transformace n**				
	32768	**65536**	**131072**	**262144**	**524288**
cuFFTW	2164,4	5715	10580,2	26821,8	63965
FFTW	970,1	2024	5674,2	16228	39920,4
cuFFT	485,6	722	1342,4	2552	4609,7

Z tabulek je patrné, že u malých délek n FFT je mnohem rychlejší FFTW a cuFFTW provedení. To je dáno tím, že knihovna cuFFT je optimalizovaná pro velké velikosti FFT a IFFT. Dalším důvodem je to, že přenos dat mezi RAM paměti PC a pamětí GPU zabere velké množství času. Čas přenosu je většinou větší než celkový čas zpracování problému pomocí CPU u malých délek n FFT. Pro malé délky n FFT kde se zpracovává jen malé množství dat není tedy práce na GPU efektivní.

GPU začíná být efektivní v tu chvíli, kdy čas přenosu dat přestane být problémem vůči délce zpracování celého výpočtu. Tuto skutečnost můžeme sledovat na transformacích délky 32768, kde je čas provedení FFT a IFFT pomocí knihovny cuFFT už asi 2× rychlejší než transformace pomocí knihovny FFTW a 4,45× rychlejší než cuFTTW. Při délce transformace 524288 je čas výpočtu FFT a IFFT pomocí knihovny cuFFT oproti knihovně FFTW dokonce 8,66× menší.

Knihovna cuFFTW je pomalejší oproti knihovně FFTW z důvodu toho, že ne všechny funkce knihovny FFTW jsou podporovány knihovnou cuFFTW. Obzvláště funkce alokování paměti pro pole nejsou podporovány. Druhým důvodem je, že čistá kompilace FFTW je mnohem jednodušší a rychlejší než kompilace cuFFTW, která musí řešit problémy s kompatibilitou. Závislost doby provedení výpočtů FFT a IFFT na délce n pro různé druhy provedení můžeme sledovat na grafu 5.2. V grafu je pro přehlednost použita pouze délka transformace n od 512 do 131072.



Obr. 5.2: Závislost doby provedení výpočtů FFT,IFFT na délce n.

5.4.2 Zrychlení násobení dvou FFT

Druhá část výpočtu rychlé konvoluce, která jde velmi efektivně zrychlit je násobení výstupní vzorků FFT před provedením IFFT a normalizace vzorků. Násobení vzorků

díky velkému počtu nezávislých výpočtů, lze velmi efektivně zrychlit pomocí paralelních výpočtů. V našem případě porovnáváme dvě provedení. První provedení je klasické provedení pomocí cyklu `for`. Druhé provedení využívá knihovny CUDA pro uskutečnění paralelních výpočtů pomocí GPU. Výsledky pro různé délky n transformace FFT jsou uvedeny v tabulkách 5.2.

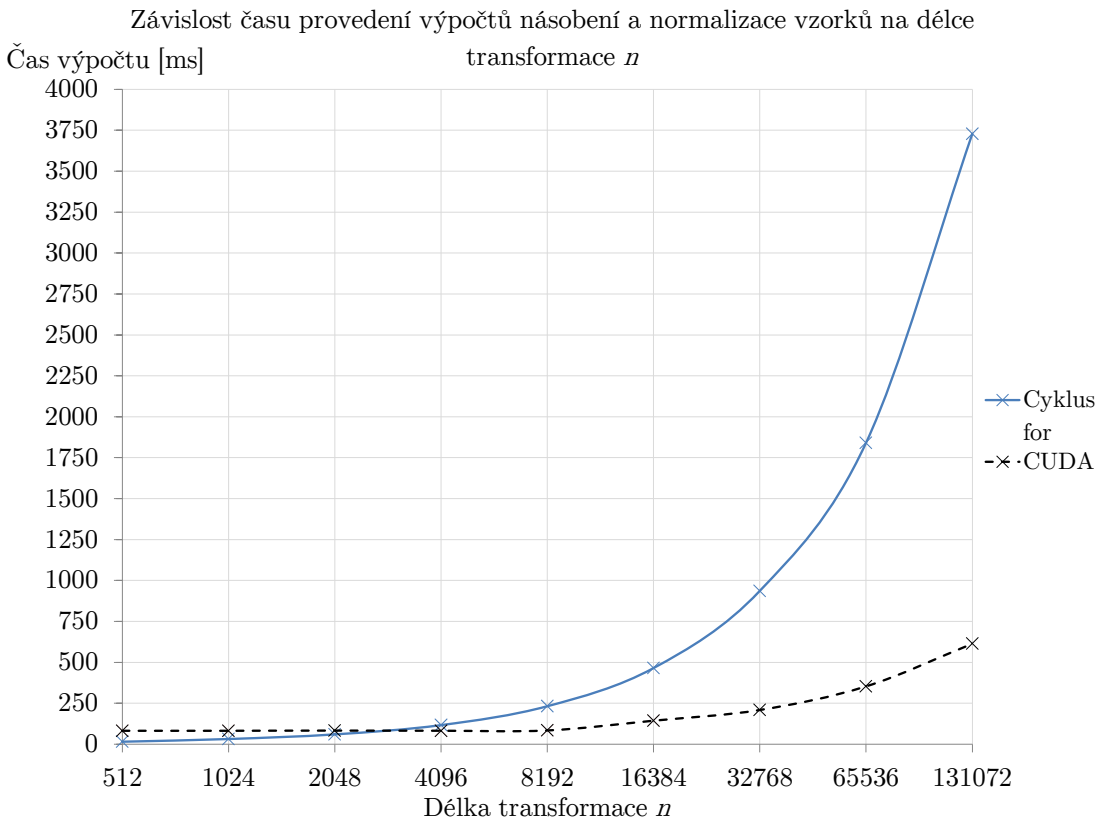
Tab. 5.2: Tabulky naměřených hodnot pro různé provedení násobení výstupních dat dvou FFT.

Čas výpočtu násobení [ms]	**Délka transformace n**							
	128	**256**	**512**	**1024**	**2048**	**4096**	**8192**	**16384**
Cyklus `for`	4	7,4	15,2	31,7	60	116,6	232	465,2
CUDA	82	81,8	82	82	83,2	82,2	84	143,6

Čas výpočtu násobení [ms]	**Délka transformace n**				
	32768	**65536**	**131072**	**262144**	**524288**
Cyklus `for`	936,8	1841,2	3728,4	7665,4	18281,1
CUDA	209	353,3	613,6	1184,8	2276,4

Z výsledků uvedených v tabulkách je patrné, že provedení CUDA není vhodné pro malé délky n FFT. Část důvodu proč je CUDA pomalejší při výpočtech aplikovaných na pole malé velikosti je to, že není optimalizovaná na tento druh práce. Druhým mnohem podstatnějším důvodem je to, že nejpomalejší část celé práce GPU je přenos dat z RAM paměti PC do paměti GPU. Proto než se data stačí přenést RAM na GPU, má CPU pro pole malé velikosti práci už hotovou. Pro malé délky n transformace FFT, tudíž pro malé pole prvků, pracuje lépe cyklus `for`. Velké délky n FFT mnohem lépe dokáže vypracovávat knihovna CUDA a to díky tomu, že GPU dokáže v jeden okamžik zpracovávat velké množství výpočtů paralelně. Při délce transformace 16384 je výpočet pomocí knihovny CUDA 3,24× rychlejší než u výpočtu pomocí cyklu `for`. U délky transformace 524288 je výpočet 8× rychlejší. Závislost času výpočtů násobení a normalizace vzorků na velikosti transformace n pro dva druhy násobení a normalizace výstupních komplexních vzorků dvou FFT je ukázána na grafu 5.3.

Násobení vzorků by se dalo zrychlit i bez pomoci GPU a to pomocí knihoven, které uskutečňují základní matematické operace skrz SIMD operace mikroprocesoru. Mezi tyto knihovny například patří knihovna yeppp! nebo libsimdpp. Pro obrovské velikosti polí vzorků bude ovšem vždy nejlepší CUDA díky obrovskému výkonu GPU pro paralelní výpočty.



Obr. 5.3: Závislost času výpočtů násobení a normalizace vzorků na velikosti transformace n.

6 ZÁVĚR

Úkolem této práce bylo porovnat rychlost různých algoritmů pro zpracování dat, jako je například konvoluce nebo FFT. Na začátku práce byl čtenář seznámen se základy paralelního zpracování. Dále zde byly popsány vlastnosti grafických karet NVDIA a jejich rozdíly vůči CPU. Poslední část teorie se věnovala problematice CUDA. Mezi stěžejní části této sekce patřil popis základních principů technologie CUDA.

První část praktické části se věnovala porovnání rychlosti výpočtů signálových operací na CPU a GPU. Nejdříve byla vysvětlena funkcí využitých pro měření. Pro měření rozdílů v rychlosti zpracování úlohy GPU a CPU byly vybrány dvě funkce `filter2D` a `Canny` a jejich upravené verze pro GPU. Uvedené funkce zahrnuje volně dostupná knihovna OpenCV. Program byl napsán pomocí programovacího jazyku C++. Algoritmus použitého programu je slovně popsán a k němu jsou uvedeny krátké části kódu.

Uvedený program byl následně odzkoušen na několika různých výpočetních jednotkách. Výsledky těchto měření, pro každou funkci, jsou zaneseny do tabulek a několika druhů grafů. První druh grafu porovnává každý komponent co se týče rychlosti provedení funkce. Druhý typ grafu se zabývá dosažením zrychlení v různých sestavách po zapojení GPU do výpočtů. Poslední graf se zabývá závislost zrychlení funkce `filter2D` pomocí GPU na rozlišení obrazu.

Druhá část praktické části se věnuje možnostem zrychlení výpočtu rychlé konvoluce. Rychlá konvoluce poskytuje dvě hlavní možnosti zrychlení výpočtu. První možnost je zrychlení výpočtu FFT a IFFT pomocí GPU s využitím knihovny cuFFT. Druhá možnost zrychlení je násobení výstupní komplexních vzorků z FFT a normalizace vzorků. Pro porovnání různých provedení těchto možností byl vytvořen program pomocí Visual Studio 2012. Algoritmus tohoto programu je vysvětlen a slovně popsán. Možnosti zrychlení výpočtu rychlé konvoluce jsou následně pomocí toho programu otestovány a zaznamenány do tabulek a grafů.

Z práce byl vyvozen závěr, že použití grafických karet pro urychlení výpočtů je opodstatněné. U funkce `filter2D` nasazení grafické karty znamenalo velké zrychlení ve vypracování celého procesu. Některé funkce se pro paralelní výpočty příliš nehodí, proto je proto nutné dbát na druh funkce. To znamená jestli funkce obsahuje dostatečný počet vhodných operací, které lze zpracovávat paralelně.

Zrychlení vypočtu rychlé konvoluce pomocí GPU je velice efektivní, ale pouze na velké velikosti transformace n. Pro malé velikosti transformace n je vhodnější využít k výpočtu CPU. Je to z důvodu toho, že GPU se musí nejdříve inicializovat. Tento proces zabere zhruba 200ms. Největším zpomalením GPU je, ale přenos dat z RAM paměti PC do paměti GPU. Tento přenos je značně pomalý a než proběhne tak CPU pro malé velikosti transformací n dokáže rychlou konvoluci vypočítat. Pro velké velikosti transformace n už přestává čas přenosu být důležitý a mnohem podstatnějším bývá čas samotného výpočtu. Zde je vhodnější GPU kvůli svému velkému výkonu při paralelních výpočtech.

LITERATURA

[1] BARLAS, Gerassimos. *Multicore and gpu programming: an integrated approach*. 1st edition. Waltham, MA: Elsevier, 2014. ISBN 978-012-4171-374.

[2] BELLANGER, Maurice. *Digital processing of signal: theory and practice*. 2nd ed. Chichester: John Wiley, 1988, 388 s. ISBN 0-471-92101-7.

[3] BRADSKI, Gary R a Adrian KAEHLER. *Learning OpenCV* [Internet]. Sebastopol: O'Reilly, 2008, [cit. 7. 12. 2015], xvii, 555 s. Dostupné z URL: `http://bit.ly/1U78ZbA`

[4] COOK, Shane. *CUDA programming: a developer's guide to parallel computing with GPUs*. Boston: Elsevier, MK, c2013, xiv, 576 p. ISBN 01-241-5933-8.

[5] DOUGLAS, J. Jones. *Fast Convolution*(1.5)[Internet]. 2004 [cit. 21. 5. 2016]. Dostupné z URL: `https://inst.eecs.berkeley.edu/~ee123/sp16/docs/FastConv.pdf`

[6] EIJKHOUT, Viktor. *Introduction to High Performance Scientific Computing* [Internet]. Second edition, 2014, [cit. 5. 11. 2015]. Dostupné z URL: `http://bit.ly/1I74ONs`

[7] El-Rewini, Hesham; Abd-El-Barr, Mostafa. *Advanced Computer Architecture and Parallel Processing* [Internet]. Wiley-Interscience. p. 2005, [cit. 1. 12. 2015]. Dostupné z URL: `http://bit.ly/1QK2QUk`

[8] FARBER, Rob. *CUDA application design and development*. Amsterdam : Boston: Elsevier ; Morgan Kaufmann, 2011, xvii, 315 s. ISBN 978-0-12-388426-8.

[9] FRIGO, Mateo a Steven G. JOHNSON. *FFTW* [Internet]. 2012, (3.3.3) [cit. 11. 5. 2016]. Dostupné z URL: `http://www.fftw.org/fftw3.pdf`

[10] Gupta, Shalini, Emami, Shervin a Brill, Frank. *OpenCV on a GPU* [Internet]. NVDIA. 2013, [cit. 10. 12. 2015]. Dostupné z URL: `http://bit.ly/2254c1r`

[11] HAGER, Georg a Gerhard WELLEIN. *Introduction to high performance computing for scientists and engineers*. Boca Raton: CRC Press, 2010, xxv, 330 s. ISBN 978-1-4398-1192-4.

[12] JASOVSKÝ, Filip. *Realizace superpočítače pomocí grafické karty*. Vysoké učení technické v Brně. Fakulta elektrotechniky a komunikačních technologií, 2014, 52 l. 1 CD-ROM.

[13] *CUFFT LIBRARY USER'S GUIDE* [Internet]. Nvdia. 2013, [cit. 4. 4. 2016]. Dostupné z URL: `http://bit.ly/1NYUavl`

[14] *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110* [Internet]. Nvdia. 2012, [cit. 21. 11. 2015]. Dostupné z URL: `http://bit.ly/1QmVmbP`

[15] PATTERSON, David A a John L HENNESSY. Computer organization and design: the hardware/software interface. 4th ed. Burlington, MA: Morgan Kaufmann Publishers, 2009, 1 sv. (různé stránkování). CD-ROM. ISBN 978-0-12-374493-7.

[16] Prof. Ing. Zdeněk Smékal, CSc. *Analýza signálu a soustav - BASS*. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2012, 252 s. ISBN 978-80-214-4453-9

[17] RODGERS, David a David RODGERS. *Improvements in multiprocessor system design* [Internet]. ACM SIGARCH Computer Architecture News. 1985, 13(3): 225-231 [cit. 28. 11. 2015]. ISSN 0163-5964. Dostupné z URL: `http://dl.acm.org/citation.cfm?id=327215`

[18] SANDERS, Jason a Edward KANDROT. *CUDA by example: an introduction to general-purpose GPU programming*. 1st print. Upper Saddle River: Addison-Wesley, 2011, xix, 290 s. ISBN 978-0-13-138768-3.

[19] SELESNICK, Ivan W. a C. Sidney BURRUS. *Fast Convolution and Filtering* [Internet]. Boca Raton: CRC Press LLC, 1999 [cit. 27. 5. 2016]. Dostupné z URL: `http://dsp-book.narod.ru/DSPMW/08.PDF`

[20] STEIN, Jonathan Y. *Digital signal processing: a computer science perspective*. New York: Wiley, 2000, 859 s. ISBN 0-471-29546-9.

[21] ŠIMEČEK, Ivan a Jaroslav SLOUP. *Programování grafických akcelerátorů*. 1. vyd. V Praze: České vysoké učení technické, 2013, 138 s. ISBN 978-80-01-05195-5.

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

API	Application Programming Interface – rozhranní pro programování aplikací
AVX	Advanced Vector Extensions – instrukční vektorová sada
COMA	Cache-Only Memory Access - paměťový přístup pouze k vyrovnávací paměti
CPU	Central Processing Unit - procesor počítače
CUDA	Compute Unified Device Architecture – architektura pro programování a implementaci na grafických kartách
cuFFT	CUDA Fastest Fourier Transform in the West – knihovna rutin pro výpočet Fourierových transformací pomocí GPU využívající funkce FFTW
cuFFTW	CUDA Fast Fourier Transform – knihovna rutin pro výpočet Fourierových transformací pomocí GPU
DFT	Discrete Fourier Transform – diskrétní Fourierova transformace
DHT	Discrete Hartley Transform – diskrétní Hartleyova transformace
FFT	Fast Fourier Transform – rychlá Fourierova transformace
FFTW	Fastest Fourier Transform in the West – knihovna rutin pro výpočet Fourierových transformací
GPGPU	General-Purpose Computing on Graphics Processing Units - výpočty prováděné pomocí grafických procesorů
GMU	Grid Managment Unit - jednotka správy souřadnicové sítě
GPU	Graphic Processing Unit – grafický procesor
IDFT	Inverse Discrete Fourier Transform – inverzní diskrétní Fourierova transformace
IFFT	Inverse Fast Fourier Transform – inverzní rychlá Fourierova transformace
MIMD	Multiple Instruction Multiple Data - více instrukcí a více datových proudů

MIPS	Microprocessor without Interlocked Pipeline Stages - instrukční sada
MISD	Multiple Instruction Single Data – více instrukcí a jeden datový proud
MPMD	Multiple Program Multiple Data - více programů a více datových proudů
NUMA	Nonuniform Memory Access - nejednotný paměťový přístup
PC	Personal Computer – osobní počítač
RAM	Random Access Memory – operační paměť počítače
SIMD	Single Instruction Multiple Data – jedna instrukce a více datových proudů
SISD	Single Instruction Single Data – jedna instrukce a jeden datový proud
SMX	Streaming Multiprocessor Architecture - průtokový mikroprocesor
SPMD	Single Program Multiple Data - jeden program a více datových proudů
SSE	Streaming SIMD Extensions - instrukční sada typu SIMD
UMA	Uniform Memory Access - jednotný paměťový přístup
VLIW	Very Long Instruction Word – velmi dlouhé instrukční slovo
a	Sekveční čas
b	Paralelní čas
$h(t)$	Spojitý signál y
$h(n)$	Vstupní posloupnost
δf	Frekveční skok
N	Délka posloupnosti
n	Délka Fourierovy transformace
p	Část úlohy využívající paralelní zpracování
P	Počet mikroprocesorů
S	Zrychlení

$S(f)$	Posloupnost čísel
$s(n)$	Vstupní posloupnost
T	Doba zpracování úlohy před zapojením paralelního zpracování
$x(t)$	Spojitý signál x
$y(t)$	Konvoluce signálů x a h
$x[n]$	Posloupnost x
$y[n]$	Konvoluce y
α	Největší část programu, kterou nelze paralizovat
$\gamma_{x,h}(\tau)$	Korelace signálů x a h
$\mathbb{R}$	Obor reálných čísel
$r(n)$	Výstupní posloupnost
GB	Jednotka digirální velikosti
kB	Jednotka digitální velikosti
px	Pixel

Obsah přiloženého CD

- \\ cuFFTW \ - Adresář z projektem využívající knihovnu cuFFTW.
- \\ FFTW \ - Adresář z projektem využívající knihovnu FFTW.
- \\ Funkce Canny a Filter2D \ - Adresář z projektem využívající funkce `Canny` a `filter2D`.
- \\ Bakalářská práce \ - Adresář z elektronickou verzí bakalářské práce.

Potřebné knihovny pro projekty

- **cuFFTW**
  - Knihovna CUDA dostupná z URL: `https://developer.nvidia.com/cuda-downloads`
  - Knihovna Libsndfile dostupná z URL: `http://bit.ly/25uFybj`
- **cuFFTW**
  - Knihovna CUDA dostupná z URL: `https://developer.nvidia.com/cuda-downloads`
  - Knihovna Libsndfile dostupná z URL: `http://bit.ly/25uFybj`
  - Knihovna FFTW verze 3.3.4 dostupná z URL: `http://www.fftw.org/install/windows.html`
- **Funkce Canny a Filter2D**
  - Knihovna CUDA dostupná z URL: `https://developer.nvidia.com/cuda-downloads`
  - Knihovna OpenCV verze 2.4 dostupná z URL: `http://opencv.org/`

Poznámky

- Všechny Projekty byly testovány debuggerem v programu Microsoft Visual Studio 2012
- Všechny knihovny byly použité ve verzi pro 64 bitové systémy.
- Program Funkce Canny a Filter2D potřebuje pro svoji funkci Grafickou kartu Nvdia s Compute Capability 3.0 a větší.