

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta biomedicínského inženýrství
Katedra přírodovědných oborů

**Paralelní zpracování signálů pomocí
grafických adaptérů**

Diplomová práce

Vedoucí práce: Ing. Jan Mužík
Student: Bc. Michal Gavalec

červen 2009

Paralelní zpracování signálů pomocí grafických adaptérů

Tato práce se zabývá možnostmi využití moderních grafických adaptérů pro obecné výpočty, zejména pro zpracování signálu. Nejdříve je stručně vysvětlena problematika obecných výpočtů na grafickém procesoru. Pak jsou uvedeny příklady již fungujících aplikací a některých vědeckých prací. Poté je stručně popsán proces zpracování dat na grafické kartě, se zaměřením na programovatelné části. Dále je uveden přehled použitelných programovacích prostředků. Nakonec jsou popsány implementace některých algoritmů pro zpracování na grafickém procesoru, jejich výhody a nevýhody a dosažené výsledky.

Graphics adapters in parallel signal processing

This work deals with possibilities of using modern graphics adapters for general-purpose computing, especially for signal processing. At first, the topic of general-purpose computation on graphics processing units is shortly described. Then some already working applications and scientific works are listed. Then the data processing on graphics card is shortly described, with aim on programmable parts. Then the review of usable programming resources is listed. At last, the implementations of some algorithms to work on graphics processing units are described. Their advantages, disadvantages and acquired results are also listed.

Poděkování

Rád bych poděkoval vedoucímu mé diplomové práce Ing. Janu Mužíkovi za vedení, podporu a připomínky při zpracování této práce.

Prohlášení

Prohlašuji, že jsem bakalářskou/diplomovou/disertační* práci s názvem:

Paralelní zpracování signálů pomocí grafických adaptérů
vypracoval(a) samostatně. Všechny zdroje, z nichž jsem při zpracování čerpal(a), v práci řádně cituji a jsou uvedeny v seznamu použité literatury.

V dne

.....
podpis

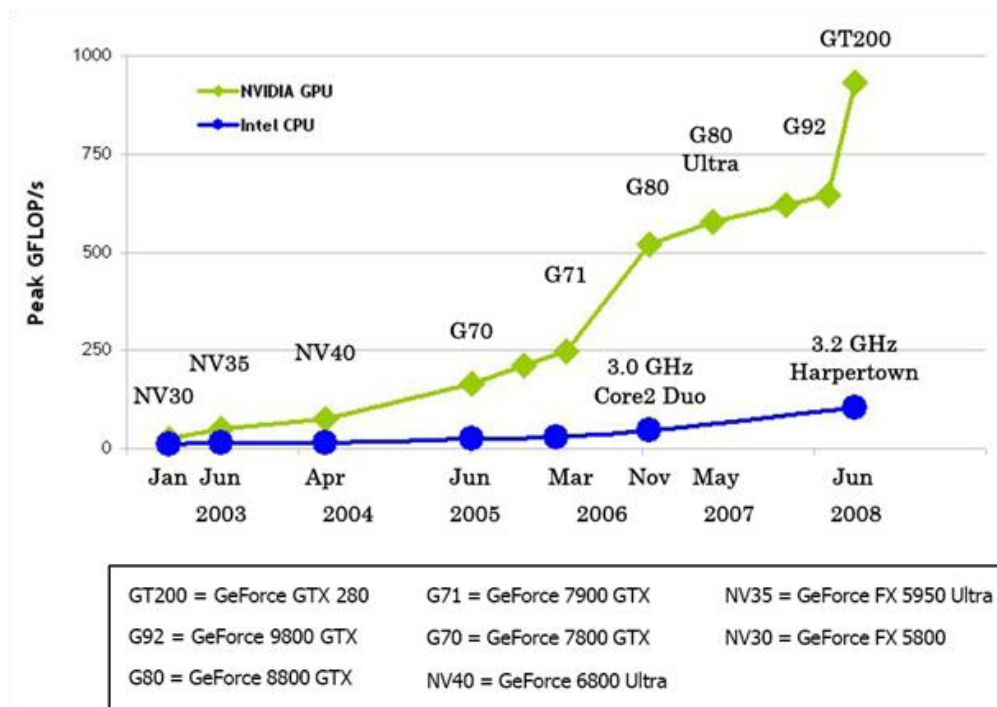
* hodící se zaškrtněte

Obsah

Úvod	1
1. Obecné výpočty na grafických procesorech	3
2. Zpracování dat na grafické kartě	6
3. Prostředky k programování GPGPU	8
3.1 Používající grafické rozhraní a s nutností znát programování tohoto grafického rozhraní.....	8
3.2 Používající grafické rozhraní, ale bez nutnosti znát programování tohoto grafického rozhraní...	9
3.3 Architektury od výrobců grafických karet	9
4. Rychlá Fourierova transformace (FFT)	11
4.1 Implementace v HLSL	12
4.2 Implementace v CUDA.....	14
4.3 Testování implementace FFT.....	16
5. Spektrogram	19
6. Wavelet transformace.....	21
6.1 Implementace DWT.....	21
6.2 Testování DWT	23
7. Konečná impulzní odezva (FIR).....	26
7.1 Implementace v HLSL	26
7.2 Implementace v MS Accelerator	27
7.3 Implementace v CUDA.....	28
7.4 Testování implementace FIR	29
8. Nekonečná impulzní odezva (IIR)	34
8.1 Implementace v MS Accelerator	34
8.2 Implementace v CUDA.....	35
8.3 Testování implementace IIR	35
9. Dvojměrná Sheppardova interpolace	40
9.1 Implementace v HLSL	40
9.1.1 Vypočtení váhovacích matic	40
9.1.2 Vypočtení výsledné mapy.....	41
9.2 Implementace v CUDA.....	42
9.2.1 Vypočtení váhovacích matic	42
9.2.2 Vypočtení výsledné mapy.....	43
9.3 Testování implementace 2D Sheppardovy interpolace	44
10. Shrnutí všech výsledků	46
Závěr	47
Seznam použité literatury	48

Úvod

Grafické adaptéry jsou v současnosti nejrychleji vyvíjenou oblastí výpočetní techniky. Tento vývoj je hnán kupředu požadavky na realistickou trojrozměrnou grafiku. Grafické procesory (GPU – Graphics Processing Unit) se díky tomu vyvinuly ve vysoce paralelizované, vícejádrové procesory s ohromným výpočetním výkonem. Tímto výkonem v paralelních výpočtech mohou grafické procesory vysoce překonávat normální procesory (CPU).



Obr. 1: Porovnání výkonu GPU a CPU, [1].

Mnoho složitých výpočetních algoritmů, jejichž počítání na klasických procesorech trvá velmi dlouho, by tak předěláním pro paralelní zpracování na grafickém procesoru mohlo získat znatelné zvýšení výkonu a zkrácení výpočetního času. Grafická karta tak může za zlomek pořizovací ceny zastat práci výkonného superpočítače.

Cílem této diplomové práce je prozkoumat možnosti využití vysokého výkonu grafických procesorů pro algoritmy digitálního zpracování signálu. Potom některé vybrané algoritmy implementovat pro výpočet na GPU a zjistit, zdali a jak velkého výkonového zisku tím bylo dosaženo.

Vybranými algoritmy zpracování signálu jsou:

- FFT – rychlá Fourierova transformace
- Spektrogram
- Wavelet transformace
- FIR – konečná impulzní odezva
- IIR – nekonečná impulzní odezva
- 2D Sheppardova interpolace

Implementované algoritmy dále upravit tak, aby byly v budoucnu použitelné v připravovaném programu LiveMap, který má sloužit ke zpracování a vizualizaci biologických signálů v reálném čase.

V následujícím textu se budu nejdříve věnovat bližšímu uvedení do problematiky obecných výpočtů na grafických procesorech (GPGPU – General-Purpose computation on Graphics Processing Units), tomu jak probíhá zpracování na grafické kartě a co umožňuje zpracovávat jiné než grafické výpočty. Dále uvedu přehled dostupných programovacích prostředků použitelných pro vývoj algoritmů počítaných na GPU. Nakonec popíšu možné implementace zadaných algoritmů, jejich výhody a nevýhody, použité postupy a dosažené výsledky.

1. Obecné výpočty na grafických procesorech

Grafické adaptéry zpracovávají data vysoce paralelním způsobem. Děje se tak při každém počítání pixelů a promítání obrazových dat na monitor. Grafické procesory jsou k tomu svojí stavbou uzpůsobeny. Oproti CPU obsahují mnohem více výpočetních jednotek (ALU) určených ke zpracování dat, než jednotek pro kontrolu a ukládání, viz obr. 1.1.



Obr. 1.1: Porovnání stavby CPU a GPU, [1].

To je základem jejich vysokého výpočetního výkonu. Díky vývoji v poslední době lze využít tohoto výkonu v paralelních operacích i pro negrafické výpočty. Algoritmus, který je jinak na CPU velmi náročný a zabere mnoho výpočetního času, lze paralelizováním a zpracováním na GPU i několikanásobně zrychlit. Mnohdy lze dosáhnout takového zvýšení výkonu, které by jinak odpovídalo zpracování algoritmu na superpočítači. Grafická karta však stojí jen zlomek ceny superpočítače, proto je možnost provádění obecných výpočtů na GPU velmi zajímavá nejen pro vědce, ale i pro komerční sféru. Oblasti využití jsou například:

- operace s velkými maticemi/vektory (BLAS)
- skládání proteinů (molekulární dynamika)
- finanční modelování
- FFT (SETI, zpracování signálu)
- raytracing
- fyzikální simulace (textilu, tekutin, kolizí, ...)
- porovnávání sekvencí (skryté markovovy modely)
- rozpoznávání řeči/obrazu (skryté markovovy modely, neuronové sítě)
- databáze
- třídění/vyhledávání
- medicínské zobrazování (segmentace, zpracování obrazu)
- a mnoho dalších

Nejznámější již úspěšně provozovanou aplikací je zřejmě projekt Folding@home [2], který se zabývá skládáním molekul prostřednictvím distribuovaných výpočtů. Umožňuje každému stáhnout si klientskou aplikaci a poskytovat pro tyto výpočty výkon svého počítače v době, kdy není aktivně používán. Klienti provádějící výpočty na GPU vykazují až 100násobné zrychlení oproti běžnému CPU.

Hráči počítačových her mohou také znát rozhraní PhysX [3] od firmy nVidia, které umožňuje využít část výkonu grafické karty, nebo k tomu vyhrazenou druhou grafickou kartu, pro počítání realistické fyziky ve hrách.

Z provedené rešerše starších prací souvisejících s tématem GPGPU nebo digitálního zpracování signálů jsem vybral následující nejzajímavější:

Práce *Considerations on the FFT variants for an efficient stream implementation on GPU* [4] se zabývá různými algoritmy rychlé Fourierovy transformace (FFT) vhodnými k implementaci na GPU. Autoři hledají nejvýhodnější variantu nabízející nejvyšší výkon. Výsledným řešením je Pease algoritmus implementovaný v jazyce Brook, který nabízí až 3,4 násobný výkon oproti CPU variantě.

Práce *FFT and Convolution Performance in Image Filtering on GPU* [5] řeší použití algoritmů FFT a konvoluce při filtrování obrazů. Autoři porovnávají oba algoritmy jak při zpracování na GPU, tak na CPU. Závěrem jsou uvedeny výhody a nevýhody jednotlivých přístupů.

Článek *Discrete Wavelet Transform on Consumer-Level Graphics Hardware* [6] ukazuje implementaci diskrétní vlnkové transformace jako SIMD algoritmu kompletně zpracovávaném na GPU. Tato implementace je založena na programovacím jazyce Cg a grafickém rozhraní OpenGL a nabízí několikanásobné zrychlení výpočtu především pro obrazy velkých rozměrů.

Scientific Computing on Commodity Graphics Hardware [7] je článek, který se zabývá celkově vědeckými výpočty na grafickém procesoru, popisuje architekturu GPU, aplikace, programovací jazyky a nástroje vhodné pro GPGPU.

Práce *A CUDA-Supported Approach to Remote Rendering* [8] pojednává o možnostech využití grafických procesorů při vzdáleném renderování, tj. zobrazování a zpracování dat mezi dvěma vzdálenými počítači. Pro implementaci na GPU využívá rozhraní CUDA od společnosti nVidia.

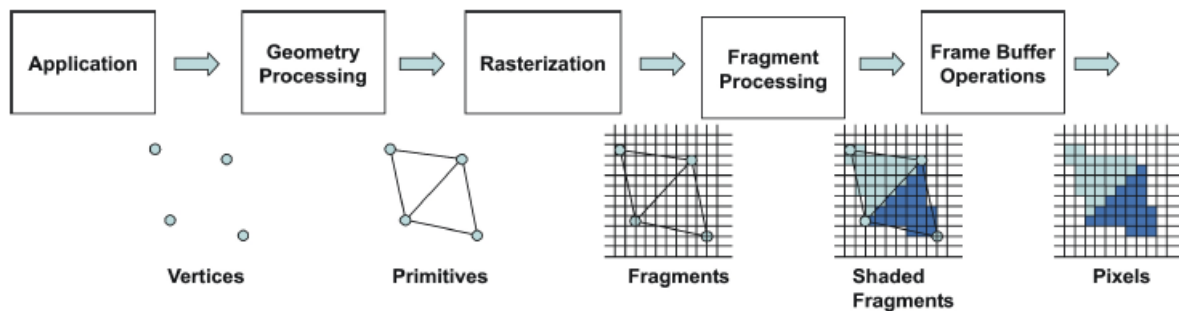
CPU-GPU Multithreaded Programming Model: Application to the Path Tracing with Next Event Estimation Algorithm [9] je prací o využití vícevláknového zpracování při simulaci globálního osvětlení virtuální scény za použití tzv. path tracing algoritmu. Tato vícevláknová implementace dokáže využít několika GPU a CPU zároveň s vysokým výkonovým ziskem.

Práce *Fast genetic programming on GPUs* [10] ukazuje použití GPU pro urychlení genetických algoritmů v několika různých problémech. Pro implementaci genetických algoritmů na GPU používá nástroje a knihovny rozhraní MS Accelerator.

Celá oblast GPGPU je v poslední době velmi atraktivní a je zpracováváno mnoho projektů a výzkumů. Hodně těchto prací lze nalézt na portálu CUDA Zone [11] firmy nVidia. Za poslední 2 roky se zde nashromáždilo téměř 300 projektů zabývajících se využitím GPGPU v mnoha různých oblastech.

2. Zpracování dat na grafické kartě

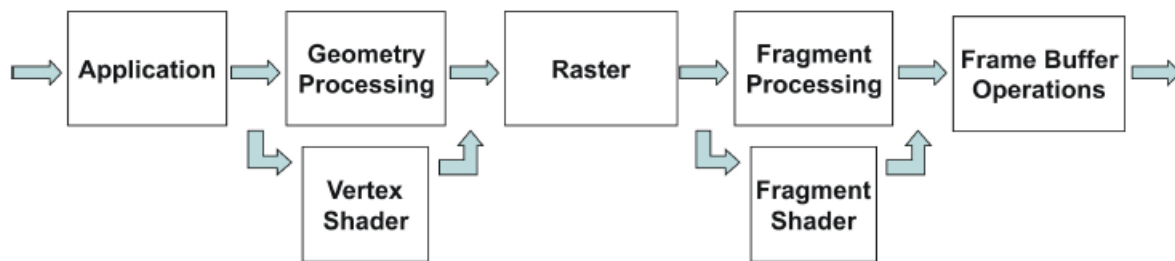
Data se na grafické kartě zpracovávají v tzv. renderovacím řetězci (rendering pipeline), který se skládá z několika na sebe navazujících operací a jehož výstupem jsou pixely zobrazitelné na monitoru. Dříve býval renderovací řetězec implementován jako pevný (Fixed-function rendering pipeline), bez možnosti modifikace. Zjednodušené znázornění tohoto řetězce je na obr. 2.1.



Obr. 2.1: Pevný renderovací řetězec, [12].

Aplikace (Application) je zodpovědná za vytváření virtuální 3D scény. Tato scéna je popsána souborem jednoduchých geometrických primitiv (body, čáry, trojúhelníky). Každé primitivum je definováno sadou vertexů a popis těchto vertexů (poloha, barva, normála) vstupuje do renderovacího řetězce. Ve fázi **geometrického zpracování (Geometry Processing)** se z těchto vertexů poskládají jednotlivá primitiva a scéna se převede ze 3D do 2D projekce. **Rasterizace (Rasterization)** navzorkuje primitiva do fragmentů výsledné scény a při **zpracování fragmentů (Fragment Processing)** je simulována interakce těchto fragmentů se světlem a je jim přiřazena konečná barva a průhlednost. Při tom se s výhodou využívá náhledů do 1D, 2D i 3D datových polí nazývaných textury. V poslední fázi řetězce (**Frame Buffer Operations**) se různými testy vyhodnotí příspěvek každého fragmentu do výsledné scény. Například fragmenty, které jsou zakryty jinými, se vyřadí a z ostatních se stanou pixely konečné scény zobrazené na monitoru.

Dnes se používá programovatelný renderovací řetězec (Programmable rendering pipeline), který má dvě programovatelné části, nazývané shadery. Vertex shader nahrazuje část geometrického zpracování a fragment shader (častěji zvaný pixel shader) nahrazuje fázi zpracování fragmentů, viz obr. 2.2.



Obr. 2.2: Programovatelný renderovací řetězec, [12].

Programování těchto shaderů umožňuje nejen lépe ovládat zpracování na grafické kartě, a dosáhnout tak mnohem lepších grafických efektů, ale také implementovat negrafické výpočetní algoritmy. Je to první způsob využití GPU pro obecné výpočty, při kterém se používá grafické rozhraní a speciální programovací jazyk k programování shaderů.

Grafické rozhraní DirectX 10, používané v operačním systému Microsoft Windows Vista, představilo třetí programovatelný shader, tzv. geometry shader. Ten přímo navazuje na vertex shader a umožňuje vytvářet i odstraňovat jednotlivé vertexy, což dříve nebylo možné. Nabízí tím mnoho nových možností při tvorbě grafických efektů, ale i při programování výpočetních algoritmů.

Z hardwarového hlediska jsou vertex a pixel shadery zpracovávány v pevně daném počtu vertex a pixel jednotek grafického procesoru. To však někdy může způsobit pokles výpočetního výkonu. Pokud se například vertex shader zahltí daty, tak pixel shader musí čekat a zůstává nevyužit. Poslední dvě generace grafických karet proto obsahují tzv. unifikované shadery, výpočetní jednotky schopné zastat funkci kteréhokoliv shaderu podle toho jak je zrovna potřeba. Unifikované shadery ale také umožňují mnohem lepší přístup k potenciálu grafického procesoru. Oba hlavní výrobci grafických karet vytvořili programovací architektury, které mohou přímo využívat výpočetní jednotky bez nutnosti grafického rozhraní. Je to druhý způsob využití GPU pro obecné výpočty, při kterém nejsou nutné znalosti grafického zpracování.

3. Prostředky k programování GPGPU

Jak již bylo naznačeno v minulé kapitole, je možné rozdělit programování GPGPU na dva způsoby. První využívá grafické rozhraní, a jsou tedy při jeho použití nutné znalosti zpracování dat na grafické kartě. Druhý způsob díky použití speciální architektury grafické rozhraní nepotřebuje a jeho programování se podobá programování běžných aplikací. Existuje ještě skupina zařaditelná mezi tyto dvě, která sice využívá grafické rozhraní, ale díky různým knihovnám její programování nevyžaduje přímo zacházet s prvky grafického zpracování.

3.1 Používající grafické rozhraní a s nutností znát programování tohoto grafického rozhraní

Sem patří programovací jazyky určené k programování shaderů, které se běžně používají pro vytváření speciálních grafických efektů. Při použití pro GPGPU je nutné dodržet postupy zpracování dat na grafické kartě. Vstupní hodnoty se proto musí vložit do textury, která se přiřadí k výpočtovému shaderu. Ten se spouští stejným způsobem jako při počítání grafických dat. Poté co provede potřebné výpočty, může výsledek zobrazit, nebo ho opět uloží do textury.

Tyto programovací jazyky jsou:

- **Cg** (C for graphics) od firmy nVidia, lze ho použít pro obě grafická rozhraní DirectX i OpenGL
- **HLSL** (High-Level Shading Language) od firmy Microsoft, použitelný pouze pro rozhraní DirectX
- **GLSL** (OpenGL Shading Language) pro rozhraní OpenGL

Jazyky Cg a HLSL byly původně vyvíjeny společně firmami nVidia a Microsoft jako jeden programovací jazyk, později se z komerčních důvodů rozdělili. Nicméně jsou syntakticky téměř totožné, což umožňuje snadný přechod mezi nimi.

3.2 Používající grafické rozhraní, ale bez nutnosti znát programování tohoto grafického rozhraní

Zde jsou programovací prostředky, které byly vytvořeny k usnadnění programování shaderů pro GPGPU. Poskytují knihovny, které obsahují příkazy umožňující programovat běžným způsobem. Tyto příkazy se pak vnitřně převedou na příkazy pro shadery a proces zpracování na grafické kartě je poté zcela stejný jako v předcházejícím případě. Výhodou tedy je, že programátor prakticky nepřijde do styku s grafickým rozhraním a nemusí tudíž mít žádné znalosti jeho programování.

Patří sem například:

- **Brook** (Stanford University) – je jazyku C podobný programovací jazyk, ale navržen tak aby využíval možnosti paralelního zpracování dat. Lze ho použít pro všechny možné platformy, jak grafických karet (ATI, nVidia), rozhraní (OpenGL, DirectX) i operačních systémů (Windows, Linux). Je ho možné stáhnout z internetu, poslední verze v05-beta1 je z listopadu 2007. [13]
- **RapidMind** – je komerční platforma pro vývoj software pro vícejádrové procesory, tedy nejen GPU, ale i CPU a lze jej použít i pro procesory Cell (např. osmijádrový procesor z herní konzole Playstation3). Je nabízen jako kompletní standardní knihovna integrovatelná do jazyka C++. [14]
- **MS Accelerator** (Microsoft Research) – je vyvíjen jako knihovna do rozhraní .NET, která za chodu kompiluje kód pro zpracování na GPU. Je k dispozici ke stažení z internetu, poslední verze 1.1.1 je z července 2007. [15]
- **Brahma** – je open-source knihovna pro rozhraní .NET 3.5, která používá novou LINQ syntaxi ke snadnějšímu programování GPU algoritmů. V současné době je zatím pouze ve verzi 0.3. [16]

3.3 Architektury od výrobců grafických karet

Sem patří architektury a rozhraní vytvořené přímo výrobcí grafických karet. Jelikož mají přímý přístup ke kapacitám grafických karet, nepotřebují žádné pomocné grafické rozhraní, a mohou tak lépe zacházet s výpočetními prostředky.

Patří sem:

- **CUDA** (Compute Unified Driver Architecture) – architektura od firmy nVidia navržená pro grafické karty nVidia od řady 8 a vyšší. Používá vlastní programovací jazyk C for CUDA. Obsahuje již také některé základní matematické knihovny. Má velkou výhodu v tom, že dokáže využívat speciální rychlou paměť (tzv. shared memory), která je umístěna přímo v grafickém procesoru. V současné době je to asi nejlépe vyvinutý prostředek pro programování GPGPU. Jsou k dispozici knihovny, umožňující programovat i v jiných jazycích, např. C# (knihovna CUDA.NET), Java, Fortran, Python, Matlab. [11]
- **CAL** (Compute Abstraction Layer) – architektura od firmy AMD/ATI určená pro grafické karty řady R600 a vyšší. Používá k programování vylepšený jazyk Brook+. Bohužel je stále ve formě beta, a poslední verze již dokonce nejsou volně dostupné (pouze po registraci). [17]
- **OpenCL** (Open Computing Language) – zcela nová architektura vyvíjená konsorciem KHRONOS Group, sdružujícím desítky firem z oboru výpočetní techniky. Není sice přímo od výrobců grafických karet, ale tito výrobci jsou ve sdružení KHRONOS také. OpenCL by měl být nový standard pro paralelní programování nejen grafických, ale i jiných vícejádrových procesorů. Zatím byla vydána pouze specifikace, první implementace se očekávají v druhé polovině roku 2009. [18]

4. Rychlá Fourierova transformace (FFT)

Fourierova transformace signálu nám poskytne informaci o spektru tohoto signálu. FFT je nejrozšířenější algoritmus pro výpočet Fourierovy transformace, používaný v mnoha oblastech, např. digitální zpracování signálu, řešení parciálních diferenciálních rovnic, rychlé násobení čísel vysokých řádů, apod. FFT je založena na diskretní Fourierově transformaci (DFT), jejíž základní vzorec je:

$$F(k) = \sum_{n=0}^{N-1} f(n) \cdot W_N^{nk}, \quad 0 \leq k < N$$

$$W_N^{nk} = e^{-i2\pi \frac{nk}{N}}$$

Tato transformace se dá přepsat jako součet dvou DFT o poloviční délce, kdy je jedna složena ze sudých prvků a druhá z lichých prvků původní posloupnosti:

$$\begin{aligned} F_k &= \sum_{j=0}^{N-1} e^{2\pi ijk/N} f_j \\ &= \sum_{j=0}^{N/2-1} e^{2\pi ik(2j)/N} f_{2j} + \sum_{j=0}^{N/2-1} e^{2\pi ik(2j+1)/N} f_{2j+1} \\ &= \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j} + W^k \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j+1} \\ &= F_k^e + W^k F_k^o \end{aligned}$$

Toto rozložení se dá rekurzivně opakovat až do momentu, kdy vznikne jednoprvková transformace, která se již velmi snadno vypočítá, protože se přímo $F_k = f_n$. Dalším trikem je zjistit, který prvek odpovídá jaké kombinaci sudo-lichých rozložení. Je dokázáno, že je to převrácená binární hodnota (bit reversal). Základní dva přístupy k řešení FFT jsou poté DIT (decimation in time) a DIF (decimation in frequency), podle toho zdali se nejdříve provede bit reversal a poté se hodnoty skládají a násobí váhovým koeficientem W , nebo naopak. Protože se využívá rozkladu posloupnosti vždy na dvě poloviční, tak signál, který chceme zpracovat těmito metodami, musí mít délku mocniny dvou, nebo je nutné použít okno o délce mocniny dvou.

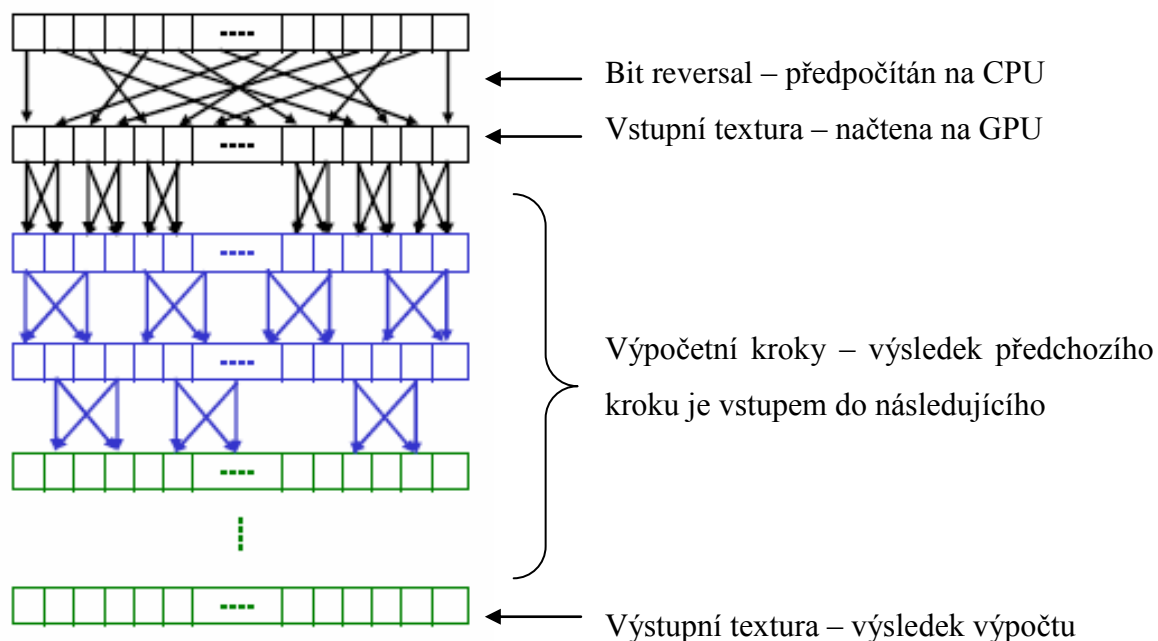
Jelikož je algoritmus FFT důležitý pro mnoho oblastí, prací které se zabývají implementací FFT na GPU je poměrně hodně. Pravděpodobně první byla *The FFT on a GPU* [19]. Autoři použili DIT algoritmus, avšak dosažený výkon byl horší, než výkon pro CPU vysoce optimalizované FFTW knihovny.

Další zajímavou implementací je *Fourier Volume Rendering on the GPU Using a Split-Stream-FFT* [20]. Autoři použili DIF a pro optimalizaci algoritmu na GPU navrhli metodu rozdělených proudů (split-stream). Výsledný výkon je lepší než předchozí implementace i než výkon algoritmu FFTW.

Zatím pravděpodobně nejrychlejší implementací je *GPUFFT* [21], která používá Stockham autosort algoritmus a má být až třikrát rychlejší než ostatní implementace. Je však navržena pouze pro grafické karty nVidia.

4.1 Implementace v HLSL

První implementace FFT na GPU je realizována v shader-programovacím jazyku HLSL s použitím grafického rozhraní XNA, které využívá programovací jazyk C#. Jelikož tato implementace používá DIT algoritmus, vstupní signál musí mít délku mocniny dvou (2^n). Signál je nejdříve načten ze souboru do pole hodnot `inSignal`. Zároveň je zjištěna velikost signálu (počet hodnot). Poté se s prvky pole `inSignal` provede převrácení binární hodnoty (bit reversal) a výsledkem se přepíše původní pole. Bit reversal je nutné takto předpočítat na CPU, jelikož při výpočtu na GPU pracujeme se všemi hodnotami najednou (jedna operace se provádí se všemi prvky pole), a není tedy možný přístup k jednotlivým prvkům pole a jejich přehazování. V dalším kroku je na základě velikosti načteného signálu vytvořena textura, do níž jsou načteny hodnoty z pole `inSignal`. Každý pixel textury odpovídá jedné hodnotě z pole. Potom je vytvořena náhledová (`Lookup`) textura, která obsahuje předpočítané váhové koeficienty W a hodnoty 0 nebo 1 pro rozlišení „sudých“ a „lichých“ pixelů v každém výpočetním kroku. Počet výpočetních kroků je odvozen od velikosti signálu – pokud má signál 2^n hodnot, je počet výpočetních kroků roven n . Systém výpočetních kroků je naznačen na obr. 4.1.



Obr. 4.1: Schéma výpočetních kroků

Vstupní textura a náhledová textura jsou načteny do grafické paměti a je spuštěn výpočet na GPU. Pro výpočet je použit pouze pixel shader s tím, že je v cyklu spuštěn tolikrát, kolik je výpočetních kroků. Kód pixel shaderu se provádí pro každý pixel a vypadá takto:

```
float4 value;
float vzd = pow(2, p);

float4 b = tex2D(LUT, float2(texCoord.x, 1/(2*np) + p/np));

if (b.x == 0) value = tex2D(ScreenS, texCoord) + tex2D(ScreenS, texCoord +
float2(vzd/n, 0)) * b.y;

if (b.x == 1) value = tex2D(ScreenS, texCoord - float2(vzd/n, 0)) -
tex2D(ScreenS, texCoord) * b.y;

return value;
```

V každém výpočetním kroku se nejdříve z náhledové textury zjistí, který pixel je v aktuálním kroku „sudý“ a který je „lichý“. Podle toho se určí, které hodnoty ze vstupní textury se spolu buď sečtou, nebo odečtou a vynásobí váhovým koeficientem. Výsledek každého kroku se uloží do pracovní textury, která se použije jako vstupní textura následujícího kroku. Na konci výpočtu je poslední (výstupní) textura s výsledky uložena do normální paměti. Potom jsou hodnoty z této textury uloženy do pole `outSignal` a z tohoto pole do výstupního souboru.

Jak jsem zjistil při podrobnějším testování, implementace v grafickém rozhraní XNA má některé nevýhody, zvláště je to:

- omezení maximální velikostí textury (4096 x 4096, u novějších karet 8192 x 8192)
- nutnost vykreslovat zároveň s výpočtem grafické okno
- někdy i omezení velikostí tohoto okna
- celkově není možnost vytvořit pouze nevizuální třídu s výpočtním algoritmem

Kvůli požadavku na vytvoření nevizuální třídy jsem se proto rozhodl najít lepší řešení implementace, a to v architektuře CUDA.

4.2 Implementace v CUDA

Pro využití CUDA v jazyce C# je k dispozici knihovna CUDA.NET. Jelikož FFT je jeden ze základních algoritmů, je již v CUDA implementován, a to v knihovně CUFFT. Mohu ho tedy využít a pouze upravit tak, aby fungoval v mém programu. Mým úkolem bylo vytvořit nevizuální třídu obsahující funkci pro výpočet FFT jednoho signálu a funkci pro FFT bloku více signálů.

Do funkce `ComputeFFT` vstupuje pole hodnot signálu `iseries`, které je typu `double`. Nejprve jej proto převedeme do pole vektorového typu `Float2`, kde prvky z pole `iseries` budou tvořit reálnou část a imaginární část vyplníme nulami.

```
// Allocate host memory for signal
Float2[] h_signal = new Float2[size];

// Initialize the memory for signal
for (int i = 0; i < size; i++)
{
    h_signal[i].x = (float)iseries[i];
    h_signal[i].y = 0;
}
```

Poté toto pole překopírujeme do paměti grafické karty a připravíme plán FFT, který určuje velikost a typ transformace.

```
// Allocate device memory for signal
// Copy host memory to device
CUdeviceptr d_signal = cuda.CopyHostToDevice<Float2>(h_signal);

// CUFFT plan
CUFFT fft = new CUFFT(cuda);
fft.Plan1D(size, CUFFTType.ComplexToComplex, 1);
```

Spustíme transformaci a výsledek nakopírujeme zpět z grafické paměti do systémové paměti.

```
// Transform signal and kernel
fft.ExecuteComplexToComplex(d_signal, d_signal, CUFFTDirection.Forward);

// Copy device memory to host
cuda.CopyDeviceToHost<Float2>(d_signal, h_signal);
```

Abychom dostali správný výstup z funkce, musíme ještě převést výsledek do typu double a zároveň ho upravit tak, aby ukazoval správně amplitudu.

```
// Transform signal to output
for (int i = 0; i < size; i++)
{
    // Transforming signal to properly show amplitudes
    oseries[i] = Math.Sqrt(Math.Pow((double)h_signal[i].x, 2) +
                           Math.Pow((double)h_signal[i].y, 2)) / size;
}
```

Nakonec vyčistíme paměť grafické karty.

```
// Destroy CUFFT context
fft.Destroy();

// Cleanup memory
cuda.Free(d_signal);
```

Při počítání bloku FFT funkcí `ComputeBlockFFT` je postup obdobný, jen na vstupu a výstupu jsou dvourozměrná pole se signály jakoby seřazenými pod sebou. Na začátku vstupní pole převedeme na jednorozměrné, ve kterém jsou signály seřazené za sebe.

```
// Initialize the memory for signal
for (int y = 0; y < sizeY; y++)
{
    for (int x = 0; x < sizeX; x++)
    {
        h_signal[sizeX * y + x].x = (float)iseries[y, x];
        h_signal[sizeX * y + x].y = 0;
    }
}
```

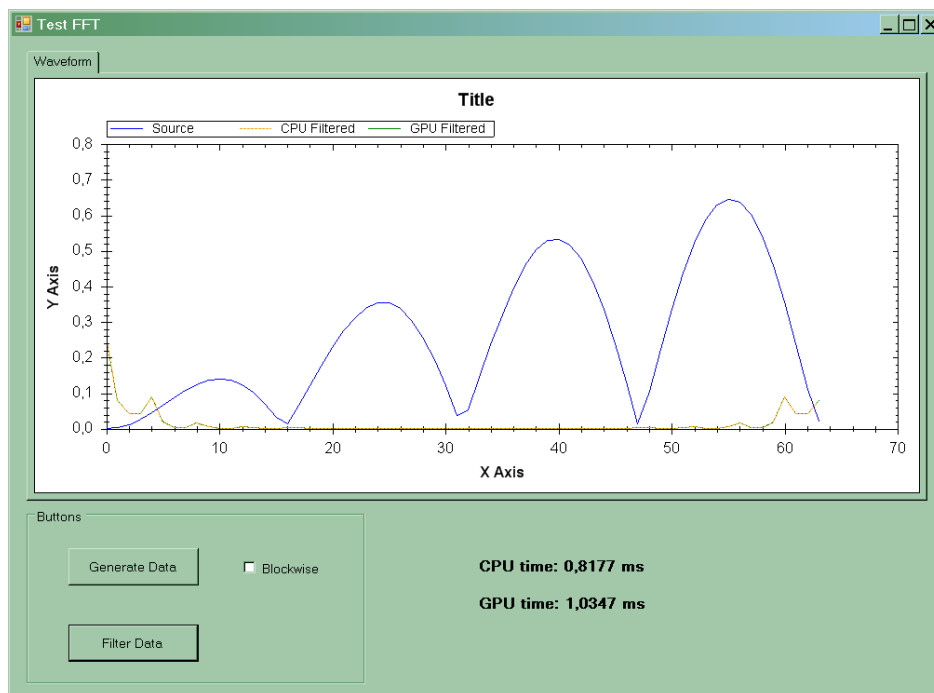
Toto musíme také zohlednit při vytváření plánu pro FFT, kdy je nutné definovat velikost jednoho signálu `sizeX` a celkový počet signálů `sizeY`.

```
// CUFFT plan
CUFFT fft = new CUFFT(cuda);
fft.Plan1D(sizeX, CUFFTType.ComplexToComplex, sizeY);
```

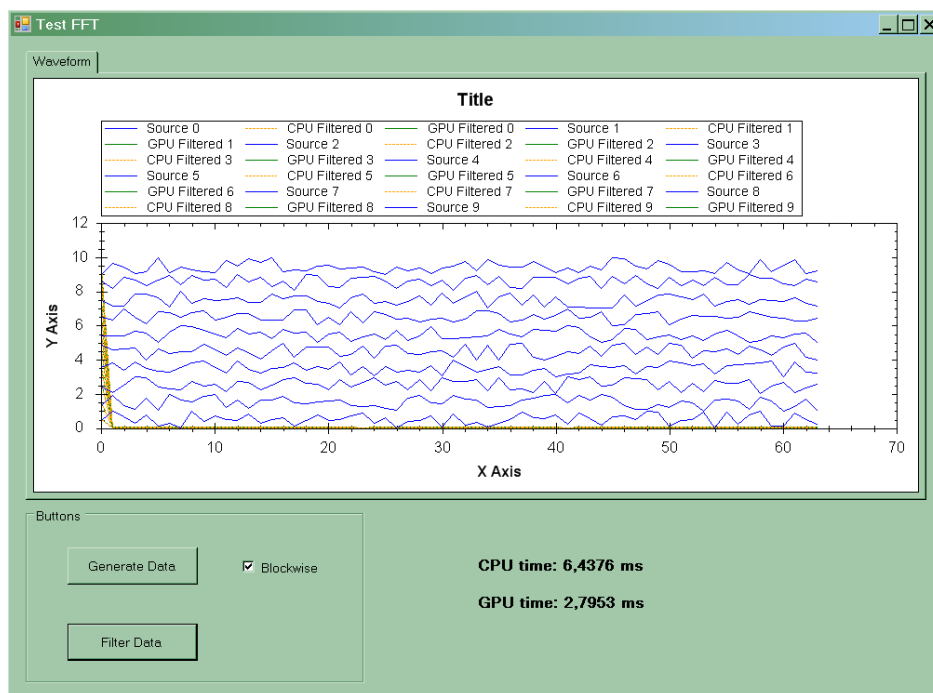
Další postup je stejný jako u 1D FFT, jen výsledek transformace musíme převést na dvourozměrné pole.

4.3 Testování implementace FFT

Pro otestování správnosti a výpočetního výkonu byla vytvořena jednoduchá aplikace TestFFT. Tato aplikace umožňuje generovat a transformovat jeden signál o libovolné délce i blok libovolného počtu více signálů. Program zobrazuje v okně graf původního signálu i graf výsledku transformace (spektra). Dále umožňuje měřit výpočetní časy jak GPU tak CPU verze FFT. Změřené časy se zobrazují přímo v okně programu a zároveň jsou ukládány do souboru pro pozdější vyhodnocení. Pro srovnání výkonu jsou zde připraveny dva algoritmy FFT pro CPU. Prvním je obyčejný FFT algoritmus napsaný přímo v C#, druhým je knihovna FFTW implementovaná pro C#. FFTW je považována za nejrychlejší algoritmus výpočtu FFT, bohužel je vytvořena pro jazyk C a její použití v C# přes implementační knihovnu její činí v některých případech pomalejší.

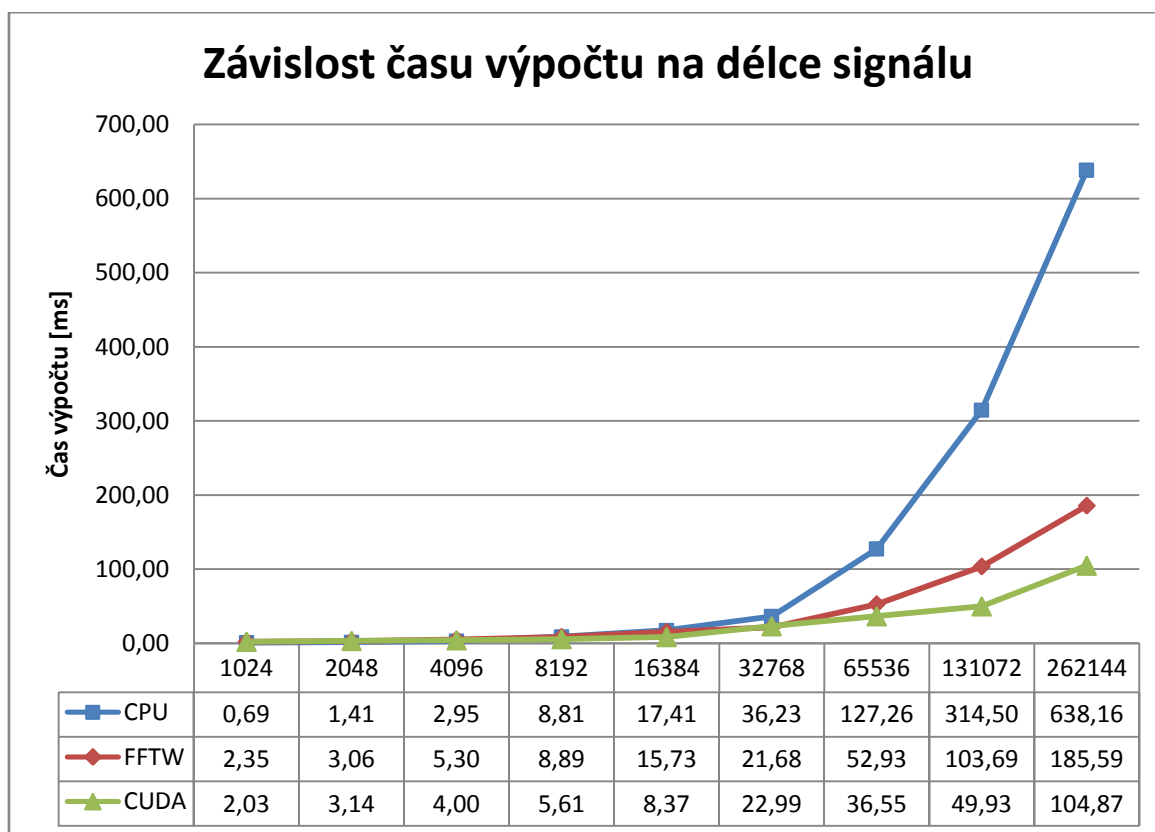


Obr. 4.2: Okno aplikace TestFFT pro 1 signál



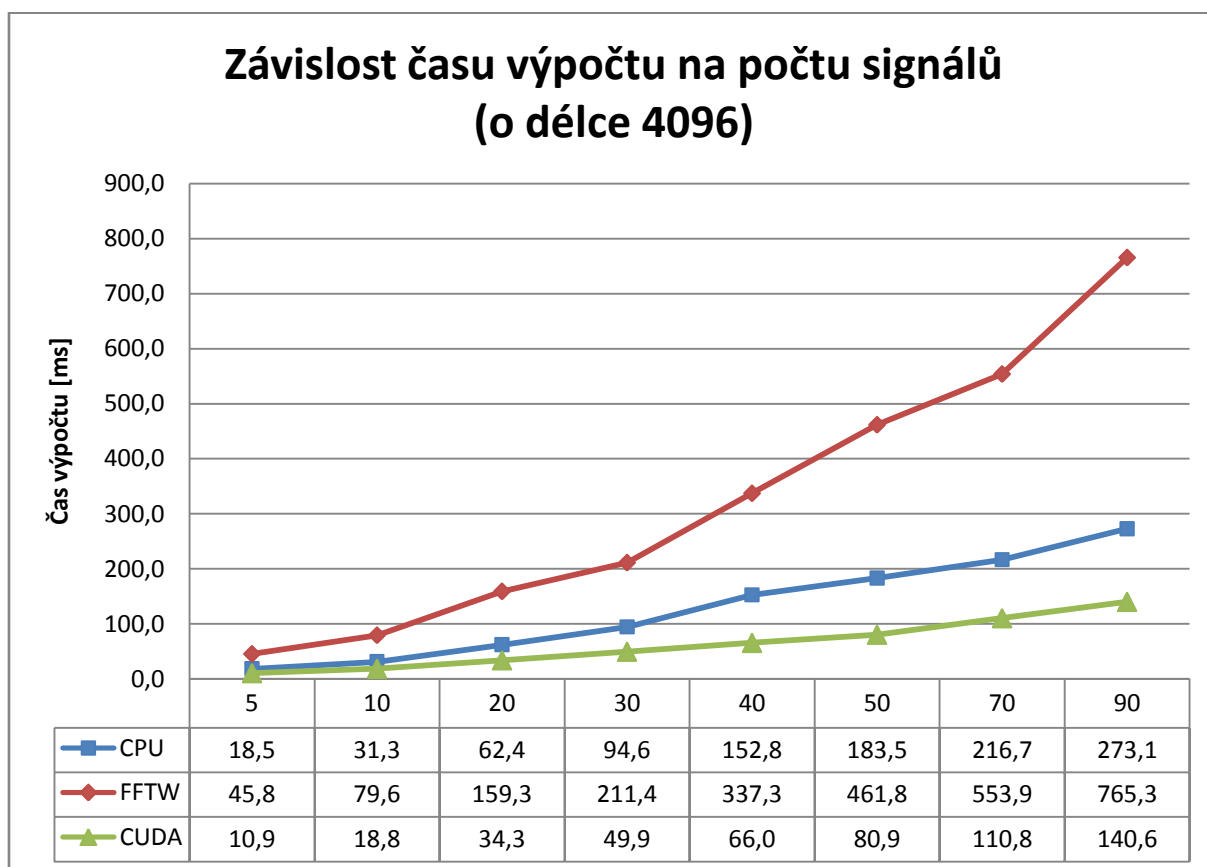
Obr. 4.3: Okno aplikace TestFFT pro blok 10 signálů

Z naměřených hodnot pro různá nastavení jsem vytvořil grafy, aby byl lépe vidět rozdíl mezi jednotlivými typy algoritmu. První graf zobrazuje závislost času výpočtu na délce 1 signálu.



Graf 4.1: Závislost času výpočtu jednotlivých algoritmů FFT na délce signálu

Jak je patrné z grafu 4.1, pro krátké signály je nejrychlejší normální CPU verze. Od délky 8192 hodnot však začíná být rychlejší zpracování na GPU prostřednictvím CUDA.



Graf 4.2: Závislost času výpočtu jednotlivých algoritmů FFT na počtu signálů

Druhý graf 4.2 zobrazuje závislost času výpočtu na počtu najednou zpracovávaných signálů o délce 4096 hodnot. S podivem je zde algoritmus FFTW vždy nejpomalejší. Domnívám se, že to může být způsobeno nutností implementace FFTW z C do C#. Jinak je vždy nejrychlejší algoritmus v CUDA počítaný na GPU.

Testováním v aplikaci TestFFT jsem dále přišel na to, že FFTW zvládne spočítat signály s jakoukoliv délkou, zatímco CUDA to umí pouze do délky 512 hodnot. Nad tuto délku je nutné používat pouze signály s délkou rovnou mocnině dvou. CPU verze umí počítat pouze s délkami signálů o mocnině dvou. Proto jsem vytvořil i funkce, které umí doplnit signál nulami na délku mocniny dvou, pokud je to potřeba. Avšak v takovém případě je potom výsledné spektrum posunuté, proto se to nehodí pro praktické použití, ale jen pro testování.

5. Spektrogram

Spektrogram zobrazuje spektrum signálu v čase. Docílí se toho tím, že se původní signál rozdělí na segmenty, které by se měly překrývat, a z těchto segmentů se pomocí FFT vypočítají spektra. Tato spektra se pak vedle sebe naskládají do matice, která se zobrazí jako obrázek. Na tomto výsledném spektrogramu pak jedna osa jakoby odpovídá času a druhá obsahuje informace o spektrálních složkách. Intenzita nebo barva jednotlivých bodů navíc vyjadřuje energii (amplitudu) té které spektrální složky.

Při implementování GPU verze spektrogramu jsem pro výpočet FFT použil třídu CudaFFT zhotovenou při implementaci FFT. K zobrazení v okně aplikace je použito grafické rozhraní XNA, takže výsledek výpočtu, pole hodnot, se před zobrazením ukládá do textury. Při zobrazení je pak použita náhledová textura k získání barevné hodnoty z velikosti amplitudy. Pixel shader který to provede, vypadá takto:

```
float4 DrawColoredMap(float2 texCoord: TEXCOORD0) : COLOR0
{
    float4 output;
    float temp;

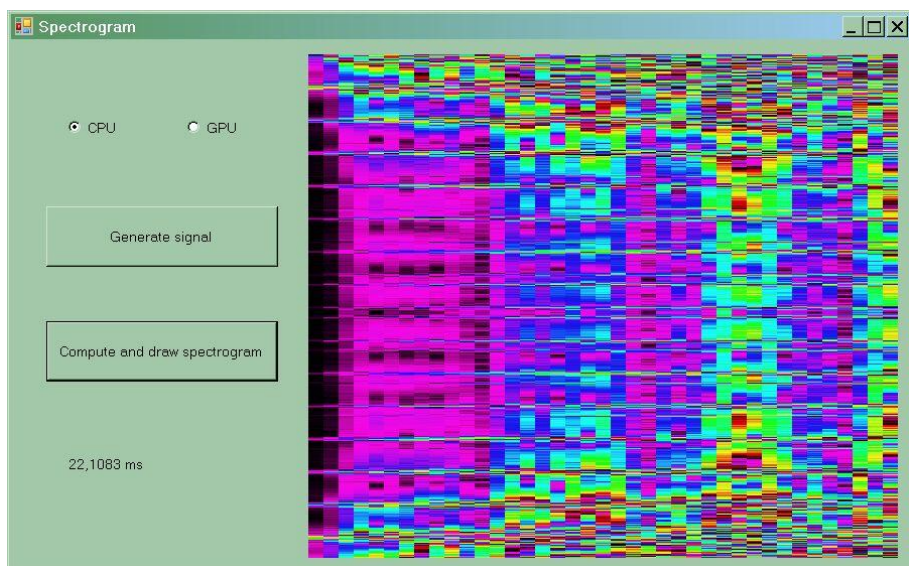
    float4 c = tex2D(ScreenS, texCoord);
    temp = c.r;
    temp = temp * 666;
    output = tex1D(ColorMap, temp);

    return output;
}
```

Výsledný spektrogram v okně testovacího programu je vidět na obr. 5.1. Tento program generuje signál o délce 10000 hodnot a rozdělí jej na segmenty dlouhé 500 hodnot. Tyto segmenty se překrývají z 50%, takže jejich výsledný počet je 39. Z nich se vypočítají jednotlivá spektra, naskládají se do matice a zobrazí. Jelikož se provádí výpočet pro větší množství segmentů, porovnání výkonu bude shodné s výpočtem FFT více signálů najednou. Ovšem jak je patrné z tabulky 5.1, zvýšení výkonu není tak vysoké. Je to kvůli tomu, že je signál rozdělen na příliš malé segmenty.

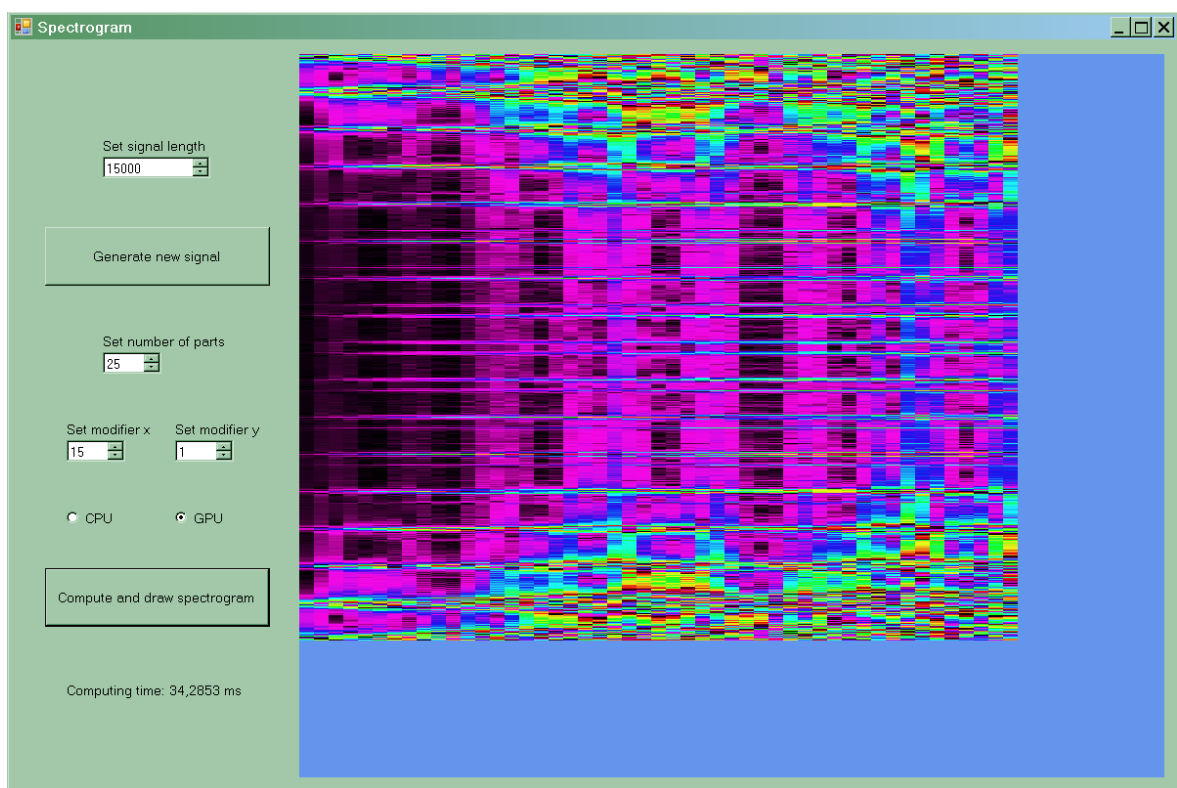
	CPU	GPU - CUDA
Čas výpočtu	22,26 ms	18,67 ms

Tab. 5.1: Časy výpočtu testovacího spektrogramu



Obr. 5.1: Spektrogram

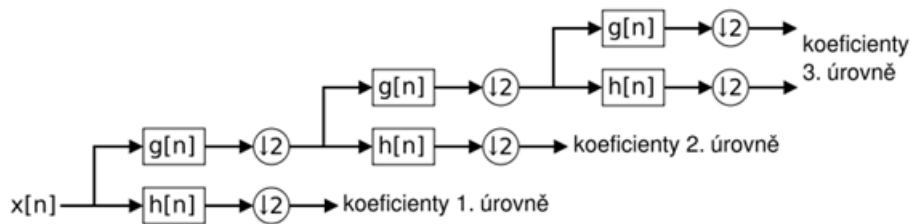
Druhá verze testovacího programu (viz obr. 5.2) umožňuje přímo v okně programu nastavovat parametry spektrogramu, jako je délka signálu, počet základních segmentů, modifikátory v osách x a y, takže lze upravovat zobrazení spektrogramu tak, aby bylo co nejlepší.



Obr. 5.2: Nastavitelný spektrogram

6. Wavelet transformace

Wavelet (česky též vlnková) transformace je integrální transformace umožňující získat časově-frekvenční popis signálu. Uplatňuje se například při odstranění šumu, detekci příznaků nebo kompresi signálů. Při digitálním zpracování se používá tzv. diskrétní wavelet transformace (DWT). Ta se vypočítá opakovanou dekompozicí původního signálu na koeficienty různých úrovní, viz obr. 6.1.



Obr. 6.1: Postupná dekompozice při wavelet transformaci, [22].

Koeficienty vyšších úrovní nám poskytují informaci o celkovém trendu signálu, zatímco koeficienty nižších úrovní zachycují doplňkové informace a jemnosti. Toho lze využít při kompresi, kdy se nejmenší koeficienty do určité úrovně vynulují. Dojde pak sice ke ztrátě určité informace, ale například při kompresi obrázků jde o úrovně, které by lidské oko stejně nebylo schopno rozeznat.

6.1 Implementace DWT

Pro implementaci wavelet transformace jsem zvolil základní Haarův wavelet pro signály o velikosti 2^n hodnot. Je použita architektura CUDA a knihovna CUDA.NET pro snadnější programování v jazyce C#. Implementace obsahuje funkce pro výpočet DWT jednoho signálu, bloku signálů po řádcích (DWT více signálů najednou), bloku signálů po sloupcích a plnou 2D wavelet transformaci (blok signálů či obrázek se transformuje nejdříve po řádcích a poté po sloupcích). Jelikož zde hrozí, že se jednotlivé thready, a při dlouhém signálu i celé bloky, desynchronizují, je výpočtový kernel opakovaně spouštěn pro každý dekompoziční krok. Tím je zaručeno, že každý thread a každý blok dokončí výpočet ve stejném okamžiku a nedojde k pomíchání hodnot.

```

for (int width = length >> 1; ; width >= 1)
{
    // Setup execution parameters
    cuda.SetFunctionBlockShape(cudaWT, 512, 1, 1);
    cuda.SetParameter(cudaWT, 0, (uint)d_osignal.Pointer);
    cuda.SetParameter(cudaWT, IntPtr.Size, (uint)d_isignal.Pointer);
    cuda.SetParameter(cudaWT, IntPtr.Size * 2, (uint)width);
    cuda.SetParameterSize(cudaWT, (uint)(IntPtr.Size * 2 + 4));

    int gridWidth = (length > 512) ? length / 512 : 1;
    // Launch kernel
    cuda.Launch(cudaWT, gridWidth, 1);

    if (width == 1)
    {
        // Copy device memory to host
        cuda.CopyDeviceToHost<float>(d_osignal, h_signal);
        break;
    }
    else
        cuda.CopyDeviceToDevice(d_osignal, d_isignal, sizeof(float) *
            (uint)width);
}

```

Pro každý krok se nejdříve upraví parametry spouštěného kernelu, takže každý kernel zpracovává vždy poloviční signál než v předchozím kroku. Po dokončení výpočtu kernelem se zkontroluje, zdali už nejsme na konci dekompozice (to pokud je délka zpracovávaného signálu rovna 1). Jestliže ano, zkopíruje se výsledek z grafické paměti do systémové paměti a dekompoziční cyklus se ukončí příkazem `break`. Pokud ne, zkopíruje se výsledek tohoto kroku zpět na vstup výpočtu, a ten pokračuje dalším krokem dekompozice. Výpočtový kernel, který se provádí pro každý dekompoziční krok, vypadá takto:

```

extern "C" __global__ void CudaWaveletTransform(float* osignal, float*
isignal, int width)
{
    int globalID = blockIdx.x * blockDim.x + threadIdx.x;
    if (globalID >= width) return;

    float data0 = isignal[globalID * 2];
    float data1 = isignal[globalID * 2 + 1];

    osignal[globalID] = (data0 + data1) / 2;
    osignal[globalID + width] = (data0 - data1) / 2;
}

```

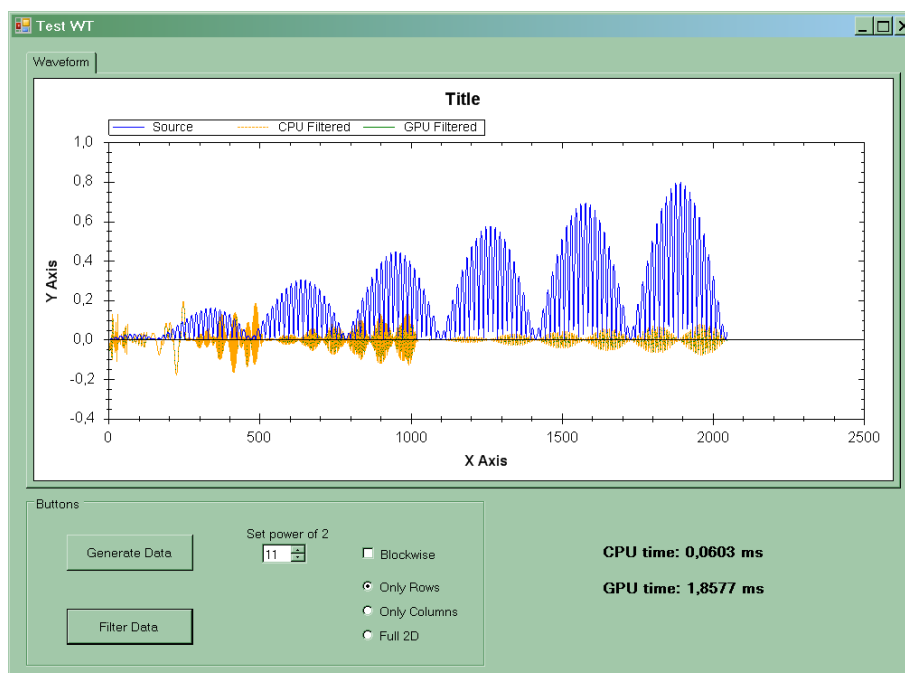
Každý thread si nejdříve zjistí svoji globální souřadnici `globalID` ze souřadnice bloku `blockIdx` ve kterém se nachází, rozměru tohoto bloku `blockDim` a svojí souřadnice `threadIdx` uvnitř tohoto bloku. Použity jsou zde pouze `x` složky souřadnic, jelikož zpracování probíhá jen v jedné dimenzi. Následně se při kontrole vyřadí nepotřebné thready.

Pak se zjistí data ze vstupního pole `isignal`, provede se rozklad, a výsledky se uloží do výstupního pole `osignal`.

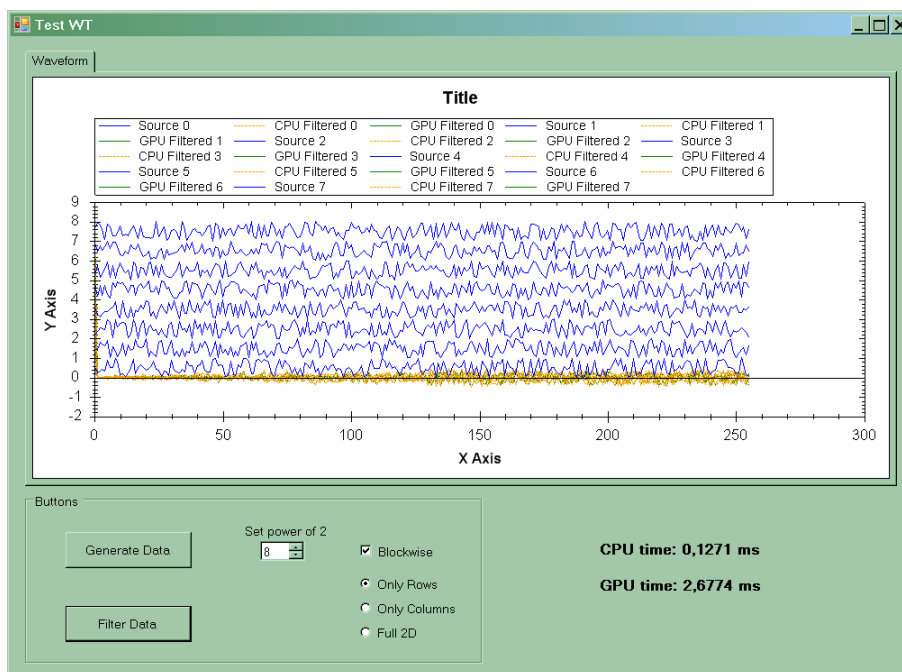
Transformace po řádcích či po sloupcích vypadá téměř stejně, jen se provádí jiná indexace zdrojových a cílových polí. Musí se vždy zohlednit, že pro zpracování v kernelu je blok signálů uložen do 1D pole, kde jsou jednotlivé signály seřazeny za sebou. Při plně 2D wavelet transformaci je pak stejným způsobem provedena nejdříve transformace po řádcích a poté transformace po sloupcích.

6.2 Testování DWT

K testování byla vytvořena jednoduchá aplikace TestWT, která zobrazuje zdrojový i oba výsledné signály. Opět je zde pro srovnání vytvořen také algoritmus DWT počítaný na CPU. Oba algoritmy, pro CPU i GPU, jsou implementovány jako nevizuální třídy se stejnými funkcemi a vstupními i výstupními parametry. Program umí měřit čas potřebný k výpočtu obou verzí algoritmu, zobrazuje ho v okně aplikace a ukládá do souboru. Přímou v okně programu lze zadávat délku zpracovávaného signálu jako mocninu 2 a zvolit zpracování jednoho signálu nebo bloku více signálů. Při zpracování více signálů lze poté přepínat mezi transformací pouze po řádcích, po sloupcích, nebo plně 2D zpracování.

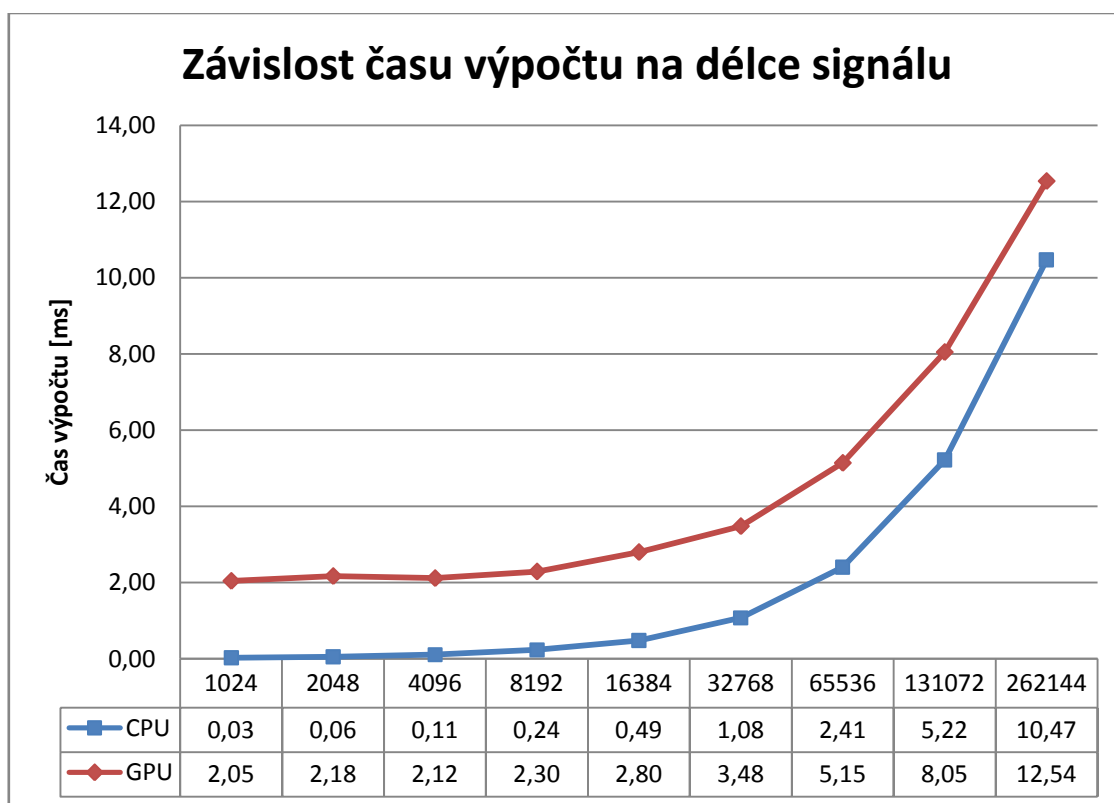


Obr. 6.2: Zpracování jednoho signálu v programu TestWT

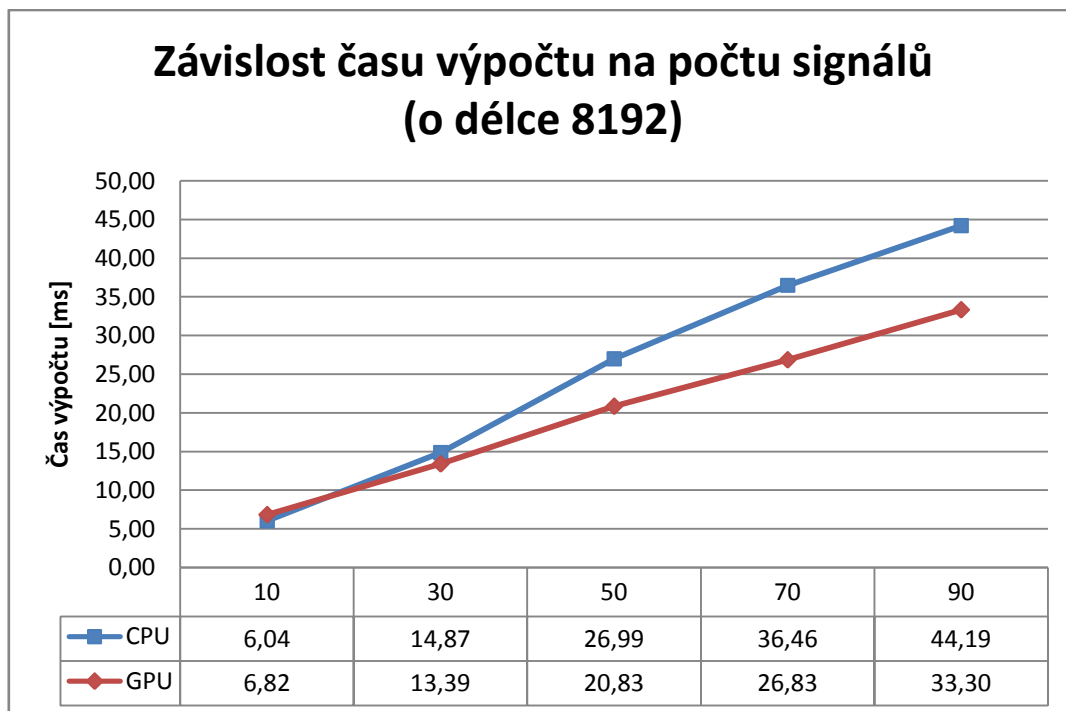


Obr. 6.3: Zpracování 8 signálů v programu TestWT

Rychlost algoritmů jsem porovnával ve třech různých nastaveních. Jako první je závislost na délce signálu, viz graf 6.1. Z tohoto grafu je bohužel patrné, že zpracování na GPU je pro jeden signál vždy pomalejší než na CPU.

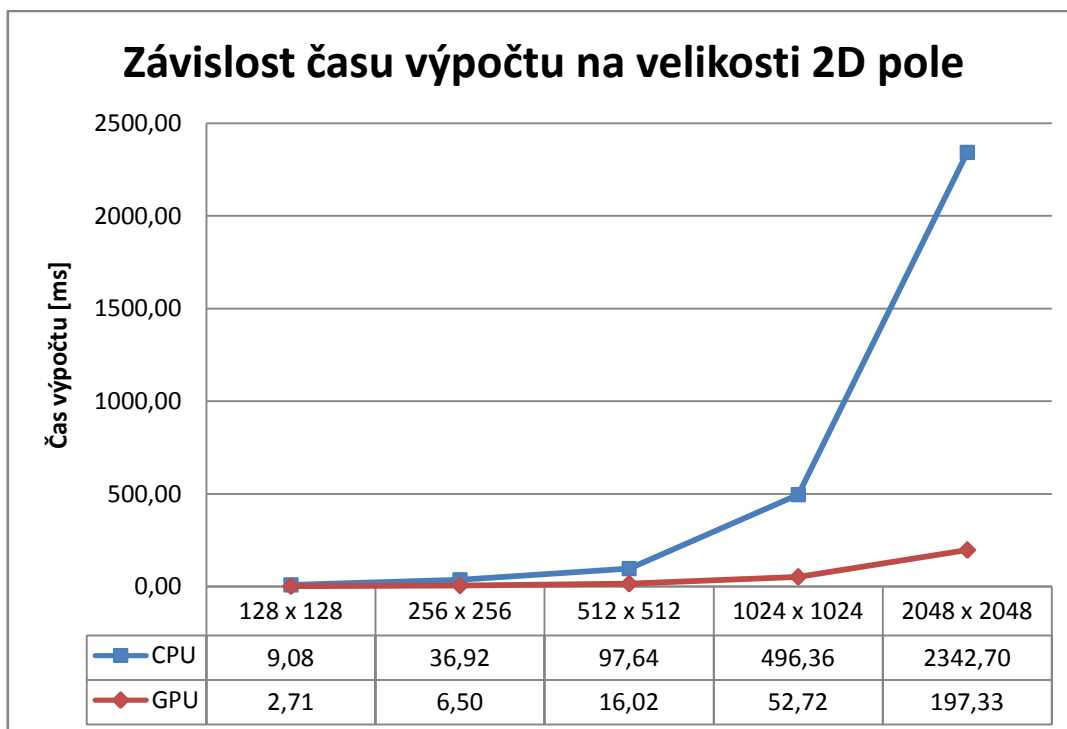


Graf 6.1: Závislost času výpočtu DWT na délce jednoho signálu



Graf 6.2: Závislost času výpočtu DWT na počtu signálů

O něco lepší výsledky jsou vidět při počítání více signálů najednou (graf 6.2). Každý signál měl 8192 hodnot a konečně se zde projevila výhoda paralelního zpracování. Nejlepší výsledky však dostaneme pro plnou 2D wavelet transformaci, jak je zřejmé z grafu 6.3, kde je měřen čas potřebný k transformaci 2D pole o rozměrech mocniny dvou.



Graf 6.3: Závislost času výpočtu na velikosti 2D pole při plné 2D wavelet transformaci

7. Konečná impulzní odezva (FIR)

FIR filtr je jedním z typů digitálního filtru. Nazývá se konečná impulzní odezva (finite impulse response), protože jeho odezva na jednotkový impulz v konečném počtu vzorkovacích intervalů spadne k nule. Odezva FIR filtru se vypočítá dle následujícího vzorce:

$$y[n] = \sum_{i=0}^N b_i x[n-i],$$

kde $x[n]$ jsou prvky vstupního signálu, $y[n]$ jsou prvky výstupního signálu, b_i jsou koeficienty filtru a N je tzv. řád filtru.

7.1 Implementace v HLSL

Implementace FIR filtru na GPU je stejně jako FFT realizována v shader-programovacím jazyku HLSL s použitím grafického rozhraní XNA, které využívá programovací jazyk C#. Nejdříve jsou načteny hodnoty signálu a koeficienty filtru z příslušných souborů do polí hodnot `inSignal` a `filter`. Zároveň je zjištěna velikost signálu a řád filtru. Poté jsou vytvořeny dvě textury, jedna pro signál a druhá pro filtr, a jsou do nich načteny hodnoty z příslušných polí. Obě textury jsou pak načteny do grafické paměti a je spuštěn výpočet na GPU. Pro výpočet je použit pouze pixel shader a jeho hlavní kód vypadá takto:

```
for (int i = 0; i < f; i++)
{
    float4 ai = tex2D(ScreenS, texCoord - float2(i*a, 0));
    if (texCoord.x - i*a < 0.0) ai = (0,0,0,0);

    float4 bi = tex2D(Filter, float2(b1 + i*b, 0.5));

    sum += ai * bi;
}

return sum;
```

Nejdříve je zjištěna hodnota prvku vstupního signálu a_i z textury signálu. Poté je provedena kontrola, zdali jsme nešáhli vedle textury, a pokud ano, tak je za a_i dosazena nula. Následně je z textury filtru zjištěna hodnota koeficientu filtru b_i . Hodnoty a_i a b_i jsou poté vynásobeny. Vše se opakuje tolikrát, kolik je řád filtru f a výsledek se načítá do hodnoty `sum`, jež je nakonec jako prvek výstupního signálu uložena do výstupní textury. Ta je potom

uložena do normální paměti. Z výstupní textury jsou hodnoty uloženy do pole `outSignal` a z tohoto pole nakonec do výstupního souboru.

Podobně jako u implementace FFT v HLSL se i zde vyskytují stejné problémy pramenící z nutnosti použití grafického rozhraní. Nejpálčivější je nemožnost vytvořit nevizuální třídu, zde jsem však navíc narazil na jakési omezení velikosti řádu filtru. Při testování na mé starší grafické kartě (nVidia GeForce 7600 GT) nebylo možné použití vyššího řádu filtru než cca 100. Při vyšších hodnotách se grafická karta zahltla a program spadl. Domnívám se, že to bylo způsobené použitím for cyklu uvnitř pixel shaderu, který se při vysokém řádu filtru stal příliš náročný. Jelikož byla jedním z požadavků na FIR filtr možnost použití filtrů o vysokém počtu koeficientů (až tisíce), bylo nutno hledat jinou cestu implementace. Protože jsem v té době ještě neměl k dispozici grafickou kartu kompatibilní s CUDA, rozhodl jsem se pro MS Accelerator.

7.2 Implementace v MS Accelerator

Rozhraní MS Accelerator je vytvořeno jako knihovna pro .NET prostředí. Programovacím jazykem je zde proto C#. MS Accelerator umožňuje programovat GPU bez nutnosti přímého používání grafického rozhraní DirectX. Nepracuje tedy s pixely a texturami, ale používá tzv. paralelní pole hodnot (parallel array) k reprezentaci hodnot určených ke zpracování na GPU. Obsahuje také speciální příkazy pro práci s těmito poli. Výpočetní část kódu implementovaného algoritmu vypadá takto:

```
DFPA pa = new DFPA(workSig);

FPA sum = new FPA(0, pa.Shape);
for (int i = 0; i < filterCoef.Length; i++)
{
    sum += PA.ShiftDefault(pa, 0f, -i) * filterCoef[i];
}

DFPA result = PA.Evaluate(sum);

float[] outSig;
PA.ToArray(result, out outSig);
```

Nejdříve jsou hodnoty signálu převedeny z normálního datového pole `workSig` do paralelního pole `pa`. Poté je vytvořeno paralelní pole `sum` pro ukládání výsledků součtů, které má stejný tvar jako pole `pa` (což je zajištěno parametrem `pa.Shape`). Následně se provede for cyklus

procházející hodnoty filtru. V tomto cyklu se do pole `sum` postupně nasčítávají hodnoty signálu vynásobené odpovídající hodnotou filtru. Je zde použit příkaz `PA.ShiftDefault`, který při každém průchodu posouvá pole hodnot o jedno políčko, čímž zajišťuje použití správné hodnoty signálu $x[n - i]$. Na rozdíl od implementace v HLSL, kde se pro každý pixel provádí vynásobení hodnoty signálu s hodnotou filtru nezávisle na ostatních, v tomto případě se díky uložení hodnot v paralelním poli toto násobení provádí pro všechny hodnoty signálu v celém poli najednou. To umožňuje, aby se for cyklus použil jen jednou pro celý výpočet. Nejenže tím dojde k úspoře výpočetního času, ale zároveň je pak možné použití filtru o vysokém počtu koeficientů (max. řád filtru asi 600). V poslední části výpočetního kódu se příkazem `PA.Evaluate` odešle výpočet na grafický procesor a výsledek se uloží do paralelního pole `result`, které se následně převede na výstupní datové pole `outSig`.

7.3 Implementace v CUDA

I když implementace FIR v MS Accelerator funguje dobře, jakmile byla k dispozici architektura CUDA, zkusil jsem implementovat FIR i tímto způsobem. Jako v ostatních případech je použita knihovna CUDA.NET pro programování v C#. Výpočetní kernel pro zpracování jednoho signálu je takovýto:

```
extern "C" __global__ void cudaFir(float* osignal, float* isignal, float*
filter, int filterSize, int width)
{
    int globalID = blockIdx.x * blockDim.x + threadIdx.x;
    if (globalID >= width) return;

    float yi = 0.0f;

    for (int i = 0; i < filterSize; i++)
    {
        float xi;

        if (globalID - i < 0)
            xi = 0.0f;
        else
            xi = isignal[globalID - i];

        yi += xi * filter[i];
    }

    osignal[globalID] = yi;
}
```

Každý thread si nejdříve zjistí svoji globální souřadnici `globalID` a nepotřebné thready jsou hned poté vyřazeny z dalšího zpracování. Poté se spustí for cyklus procházející filtr.

V každém kroku cyklu se zjistí hodnota x_i ze vstupního pole `isignal`. Pokud se indexem dostaneme mimo toto pole, tak se za x_i dosadí nula. Hodnota x_i se vynásobí s hodnotou z filtru a přičte se do výsledné hodnoty y_i . Ta se po projití celého cyklu zapíše do výstupního pole `osignal`.

Výpočetní kernel pro zpracování bloku více signálů vypadá takto:

```
extern "C" __global__ void cudaFir2D(float* osignal, float* isignal, float*
filter, int filterSize, int width, int height)
{
    int globalIdx = blockIdx.x * blockDim.x + threadIdx.x;
    int globalIDy = blockIdx.y * blockDim.y + threadIdx.y;
    if (globalIdx >= width || globalIDy >= height) return;

    float yi = 0.0f;

    for (int i = 0; i < filterSize; i++)
    {
        float xi;

        if (globalIDy * width + globalIdx - i < 0)
            xi = 0.0f;
        else
            xi = isignal[globalIDy * width + globalIdx - i];

        yi += xi * filter[i];
    }

    osignal[globalIDy * width + globalIdx] = yi;
}
```

Je vcelku stejný jako pro jeden signál, jen zjištění globální souřadnice probíhá ve dvou dimenzích, stejně tak i vyřazení nepotřebných threadů. Dále je pak změněno indexování do polí tak, aby odpovídalo tomu, že signály jsou v poli seřazeny postupně za sebou.

7.4 Testování implementace FIR

Podobně jako u předchozích algoritmů byla pro otestování správnosti a výpočetního výkonu implementace FIR vytvořena jednoduchá aplikace `TestFir`. Algoritmy FIR v MS Accelerator a v CUDA jsou zde implementovány jako nevizuální třídy a upraveny tak, aby umožňovaly zpracování jednoho signálu i bloku více signálů. Pro srovnání `TestFir` také obsahuje třídu s algoritmem FIR počítaným na CPU. Aplikace vytváří hodnoty testovacího signálu po blocích, čímž umožňuje simulovat zpracování v reálném čase. Pro správný výsledek FIR

algoritmu při takovémto postupném zpracování je nutné zajistit pamatování hodnot předchozího bloku. To se děje pomocí pole `prevSig`:

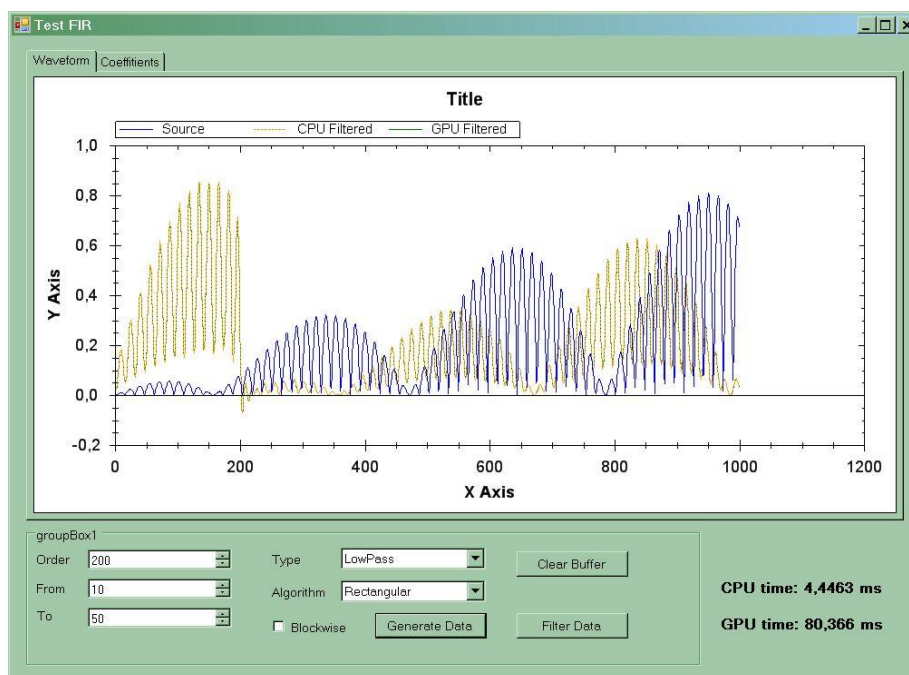
```
float[] workSig = new float[prevSig.Length + inSig.Length];
Array.Copy(prevSig, workSig, prevSig.Length);
Array.Copy(inSig, 0, workSig, prevSig.Length, inSig.Length);
```

Pracovní pole `workSig` se vždy před odesláním na GPU složí z hodnot předchozího bloku `prevSig` a z nových hodnot `inSig`. Po skončení výpočtu se pak pole `prevSig` přepíše posledními hodnotami z pole `workSig`:

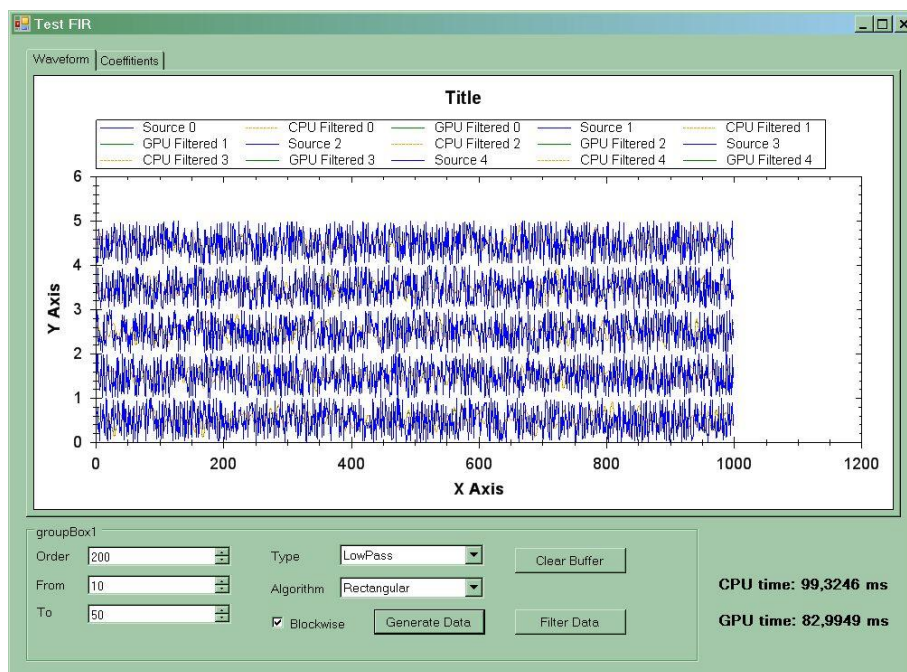
```
for (int i = 0; i < prevSig.Length; i++)
{
    prevSig[prevSig.Length - 1 - i] = workSig[workSig.Length - 1 - i];
}
```

Tímto je zajištěno, že se při výpočtu použije vždy správná hodnota $x[n - i]$. Obdobně, ale ve dvou rozměrech je to uděláno i pro blok více signálů.

Aplikace TestFir dále zobrazuje grafy zdrojového i obou výsledných signálů, jak GPU tak CPU verze algoritmu. Umožňuje také měření času potřebného pro výpočet a jeho zobrazení přímo v aplikaci, nebo zápis do souboru. Příklady, jak vypadá okno aplikace TestFir, jsou na obr. 7.1 a obr. 7.2.

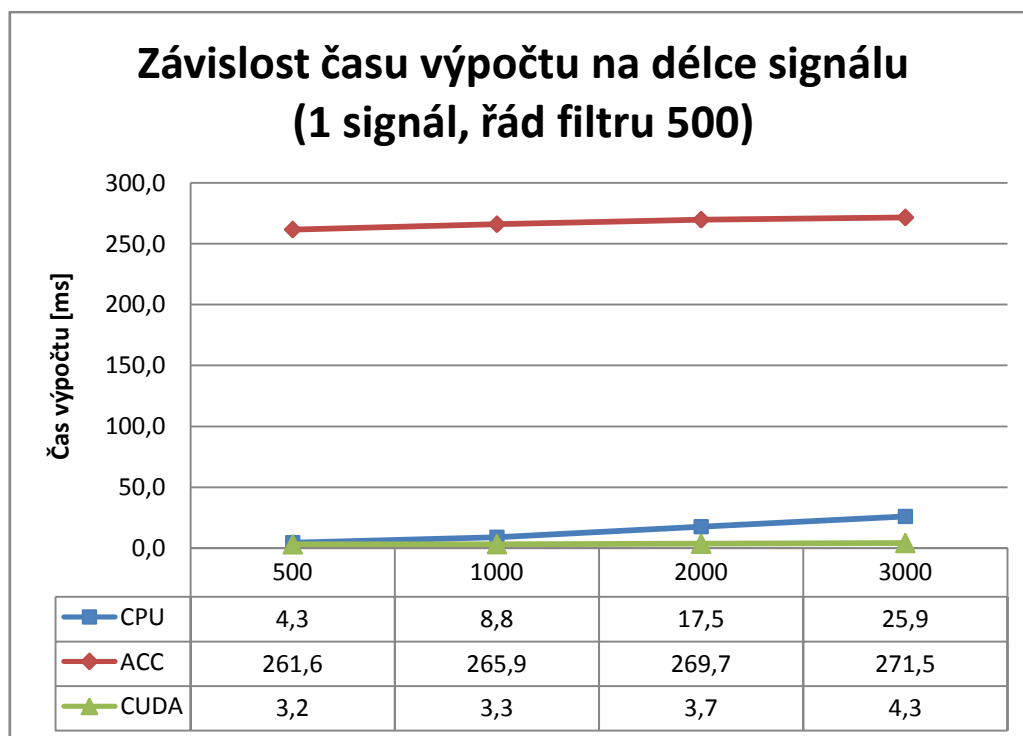


Obr. 7.1: Aplikace TestFir, zpracování jednoho signálu



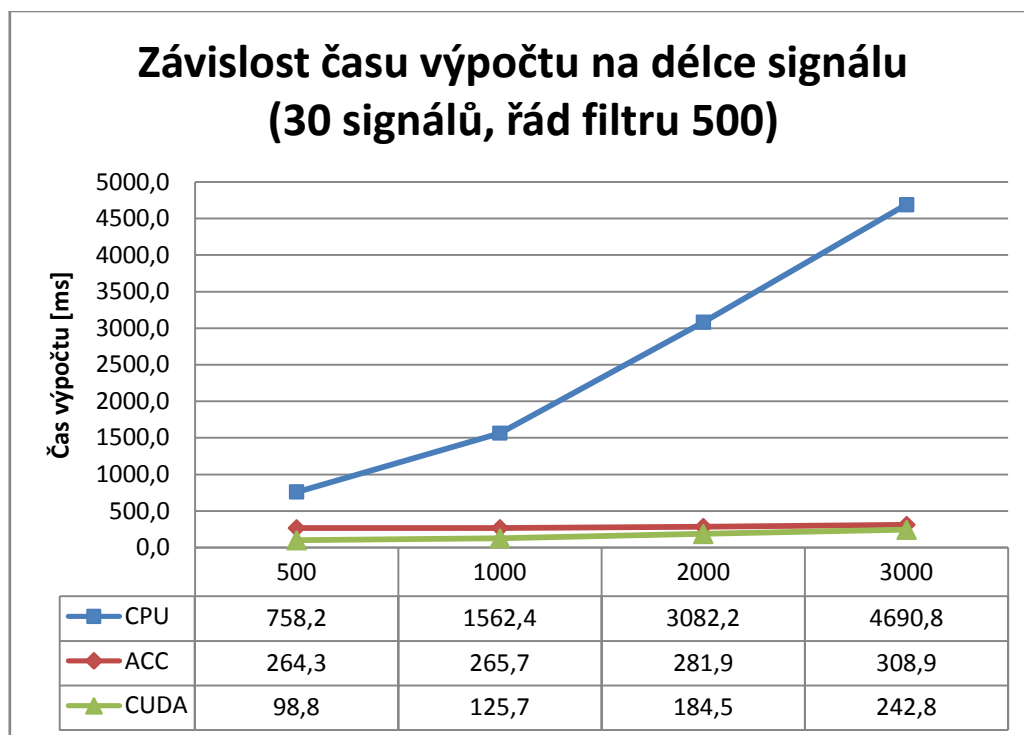
Obr. 7.2: Aplikace TestFir, zpracování pěti signálů najednou

Na obr. 7.1 je vidět zpracování jednoho signálu, čili 1D pole hodnot. Protože se grafy výsledných signálů GPU i CPU verze překrývají, je zřejmé, že implementace FIR pro zpracování na GPU dává správné výsledky. Na obr. 7.2 je vidět zpracování bloku pěti signálů najednou, kdy se použije 2D pole hodnot.



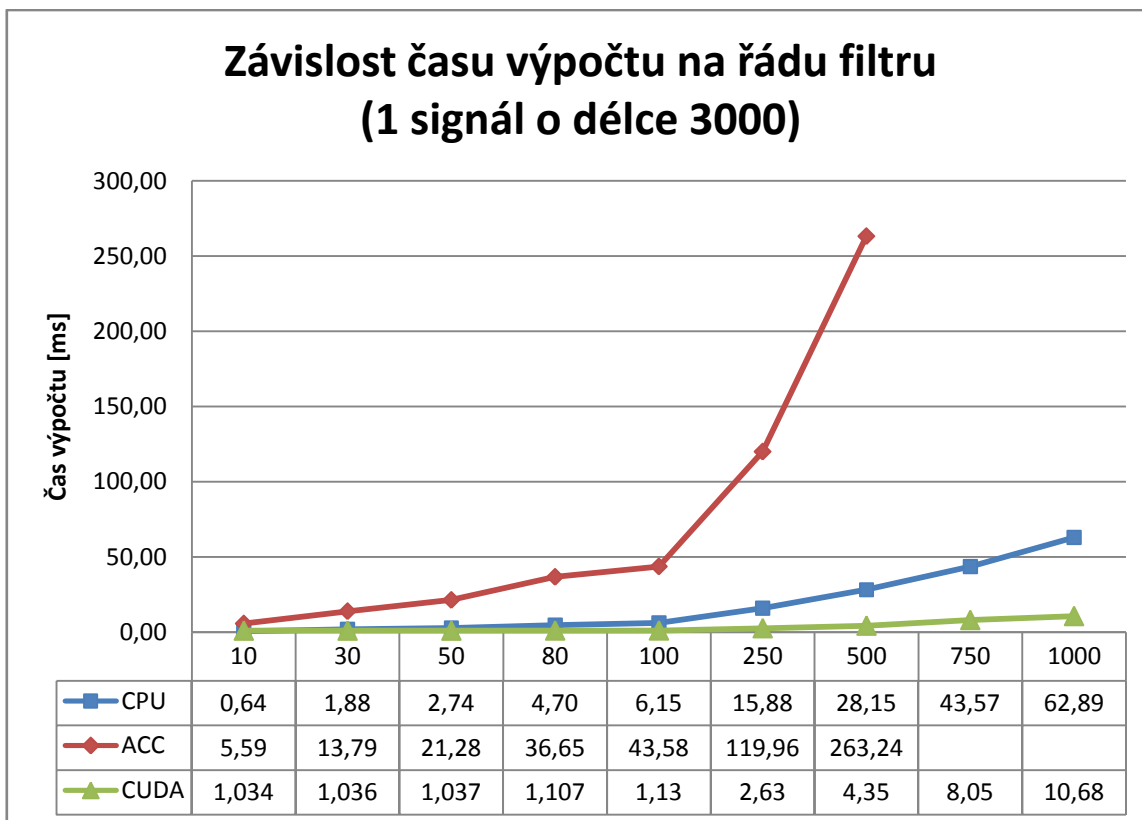
Graf 7.1: Závislost času výpočtu FIR na délce 1 signálu, při řádu filtru 500

Graf 7.1 ukazuje výhodnost použití architektury CUDA. Zatímco čas potřebný k výpočtu algoritmu v MS Accelerator je kvůli inicializaci a používání grafického rozhraní příliš vysoký, algoritmus v CUDA, který umí zacházet s grafickou kartou přímo, je dokonce rychlejší než klasický výpočet na CPU. Z grafu 7.2 je vidět, že tato inicializace a používání grafického rozhraní v MS Accelerator zabírá vždy určitý čas nezávisle na ostatních parametrech. Dále je patrné, jak pro CPU algoritmus čas výpočtu stoupá přímo úměrně s počtem najednou zpracovávaných signálů. Algoritmus v CUDA je zde výrazně rychlejší než všechny ostatní verze.

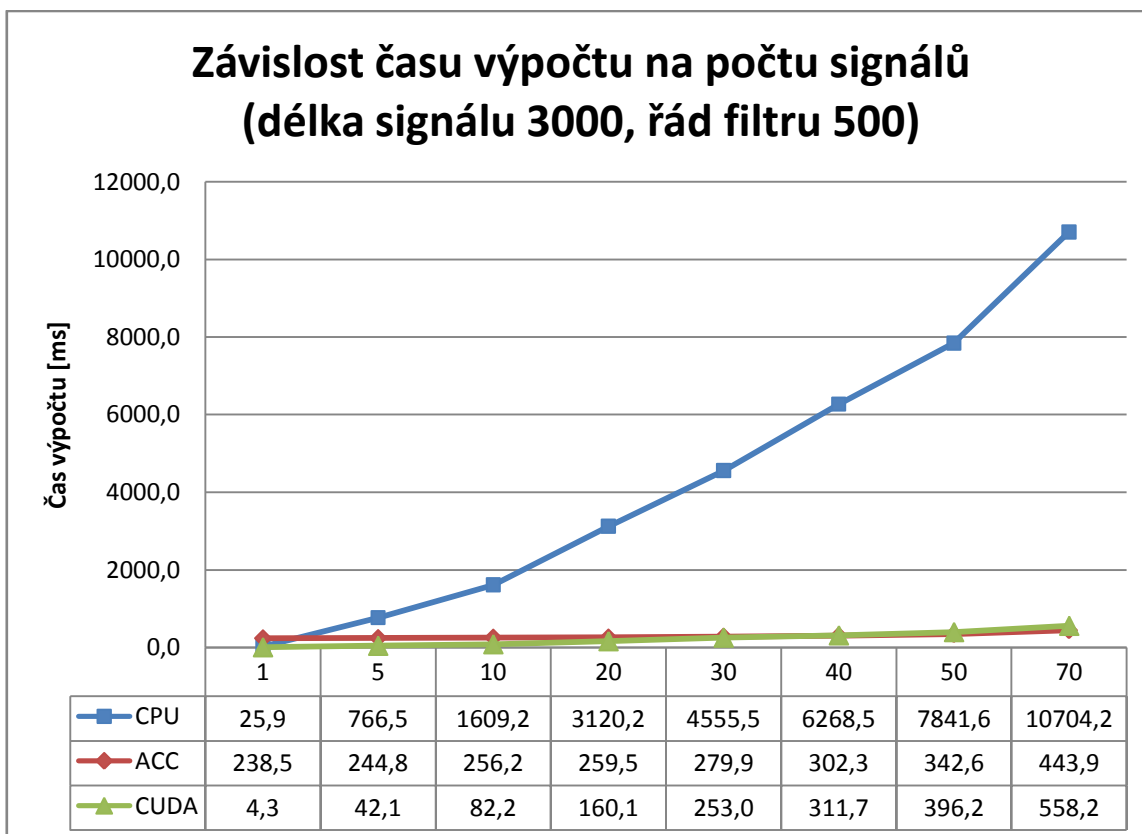


Graf 7.2: Závislost času výpočtu FIR na délce 30 signálů, při řádu filtru 500

Další graf 7.3 zachycuje závislost času výpočtu na řádu filtru měnícím se od 10 do 1000. Algoritmus v MS Accelerator zde přestává fungovat u řádů vyšších než cca 600. Zdá se, že jde o stejný problém, jaký byl u implementace v HLSL a XNA, jen se projevuje později. Zpracování v CUDA zde nabízí výrazné zvýšení výkonu zvláště pro vyšší hodnoty. Poslední graf 7.4 ukazuje časy výpočtu pro různý počet najednou zpracovávaných signálů. CUDA zde vítězí hlavně pro nízké počty signálů. Pro vyšší počty signálů se rovná, nebo je dokonce i trochu pomalejší, než algoritmus v MS Accelerator. Náročnost výpočtu na CPU v tomto případě ukazuje výhody použití paralelního zpracování na grafickém procesoru.



Graf 7.3: Závislost času výpočtu FIR na řádu filtru



Graf 7.4: Závislost času výpočtu FIR na počtu najednou zpracovávaných signálů

8. Nekonečná impulzní odezva (IIR)

IIR filtr je dalším z typů digitálního filtru, stejně jako FIR. Na rozdíl od FIR se však odezva IIR filtru na jednotkový impuls nepřiblíží k nule v konečném počtu vzorkovacích intervalů, ale je nenulová po nekonečný čas. Odtud se nazývá nekonečná impulzní odezva (infinite impulse response). Odezva IIR filtru se vypočítá dle následujícího vzorce:

$$y[n] = \sum_{i=0}^P b_i x[n-i] - \sum_{j=1}^Q a_j y[n-j],$$

kde $x[n]$ jsou prvky vstupního signálu, $y[n]$ jsou prvky výstupního signálu, b_i jsou koeficienty dopředního filtru, P je řád dopředního filtru, a_j jsou koeficienty zpětnovazebního filtru a Q je řád zpětnovazebního filtru.

Jak je vidět ze vzorce, IIR filtr obsahuje zpětnovazební člen. Tedy k vypočítání hodnoty prvku výstupního signálu $y[n]$ potřebuje nejen hodnotu prvku vstupního signálu $x[n]$, ale i výslednou hodnotu předchozího prvku z výstupního signálu $y[n-j]$. Při výpočtu na CPU se toho dosáhne snadno, jelikož se signál prochází od začátku do konce a hodnoty se vypočítávají postupně. Je tedy možné při výpočtu jedné hodnoty znát výsledek hodnoty předchozí. Při výpočtu na GPU se však všechny hodnoty počítají najednou, v jeden okamžik. Není tedy možné znát výslednou hodnotu předchozího prvku – v době, kdy je potřeba pro výpočet, ještě neexistuje. Proto je implementace celého IIR filtru pro výpočet na GPU nemožná. Ovšem při porovnání vzorců IIR a FIR si lze povšimnout, že první suma v IIR je vlastně vzorec výpočtu FIR. Takže je možné výpočet rozdělit a nejdříve vypočítat první sumu IIR (stejně jako FIR) na GPU a následně dopočítat zpětnou vazbu obvyklým způsobem na CPU.

8.1 Implementace v MS Accelerator

Výpočetní jádra části pro GPU a části pro CPU v MS Accelerator vypadají takto:

```
// 1. sum computed on GPU - in float
DFPA pa = new DFPA(workSig);

FPA sum = new FPA(0, pa.Shape);
for (int i = 0; i < filterCoefB.Length; i++)
{
    sum += PA.ShiftDefault(pa, 0f, -i) * filterCoefB[i];
}
```

```

DFPA result = PA.Evaluate(sum);

float[] outSig;
PA.ToArray(result, out outSig);

// 2. sum computed on CPU - in double
for (int i = 0; i < iseries.Length; i++)
{
    double sum1 = 0.0;
    double sum2 = 0.0;
    double yi;

    sum1 = (double)outSig[prevInSig.Length + i];

    for (int k = 1; k < CoeffitientsA.Length; k++)
    {
        if (i - k < 0)
            yi = prevOutSig[prevOutSig.Length + i - k];
        else
            yi = oseries[i - k];

        sum2 += yi * CoeffitientsA[k];
    }

    oseries[i] = sum1 - sum2;
}

```

Jak je vidět, výpočet pro GPU je totožný s výpočtem FIR. Prvky z pole `outSig` s výsledky FIR se poté stanou první sumou IIR. Výpočet na CPU pak již probíhá ve dvojnásobné přesnosti `double`. Při zpracování signálu po blocích je také nutné pamatovat si nejen hodnoty předchozího vstupního bloku, ale i hodnoty předchozích výsledků, aby se vždy správně zjistila zpětnovazební hodnota $y[n - j]$.

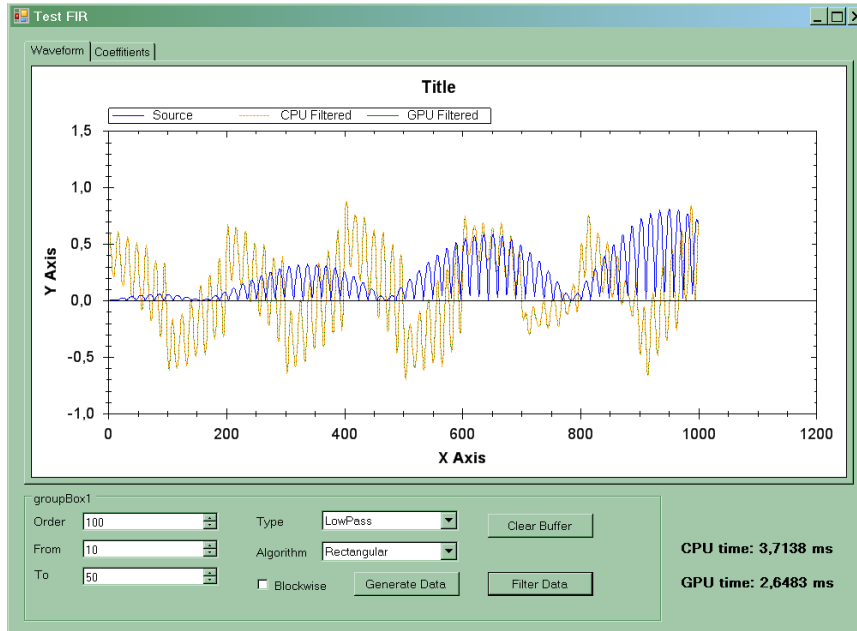
8.2 Implementace v CUDA

Implementace IIR v CUDA využívá pro výpočet první sumy (čili FIR) stejný kernel jako je pro výpočet FIR samotné. Změní se tedy pouze obslužný kód ve třídě `CudaIir.cs`, a to tak, že přibude výpočet druhé sumy na CPU. Ten je zase totožný s výpočtem použitým ve verzi v MS Accelerator.

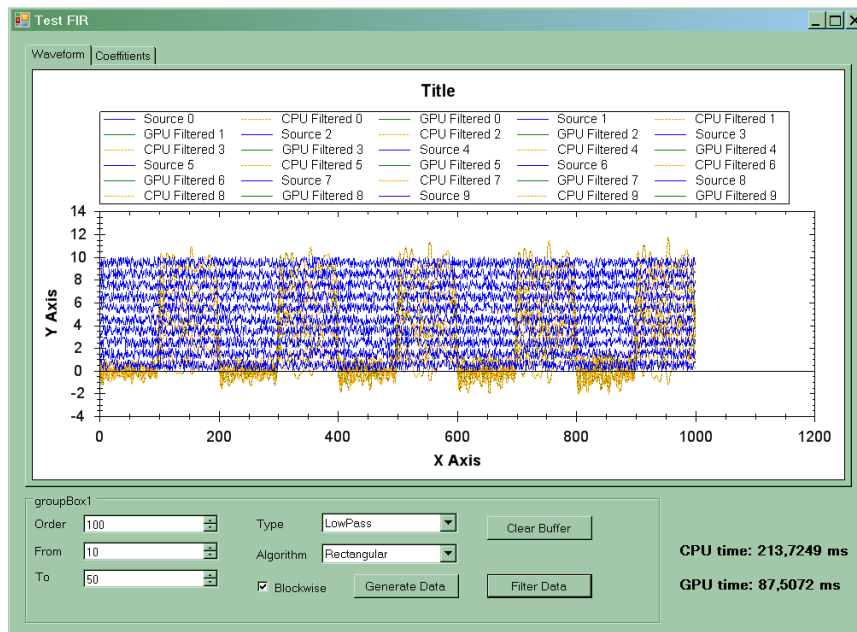
8.3 Testování implementace IIR

Pro testování je využita stejná aplikace jako pro FIR, program `TestFir`. Všechny verze algoritmu IIR (CPU, MS Accelerator a CUDA) jsou zde také implementovány jako nevizuální

třídy, které umožňují zpracování jednoho signálu i bloku více signálů. Je opět zajištěno pamatování posledních výsledků pro simulované zpracování v reálném čase a měření všech výpočetních časů. Příklady zpracování jsou na obr. 8.1 a 8.2.

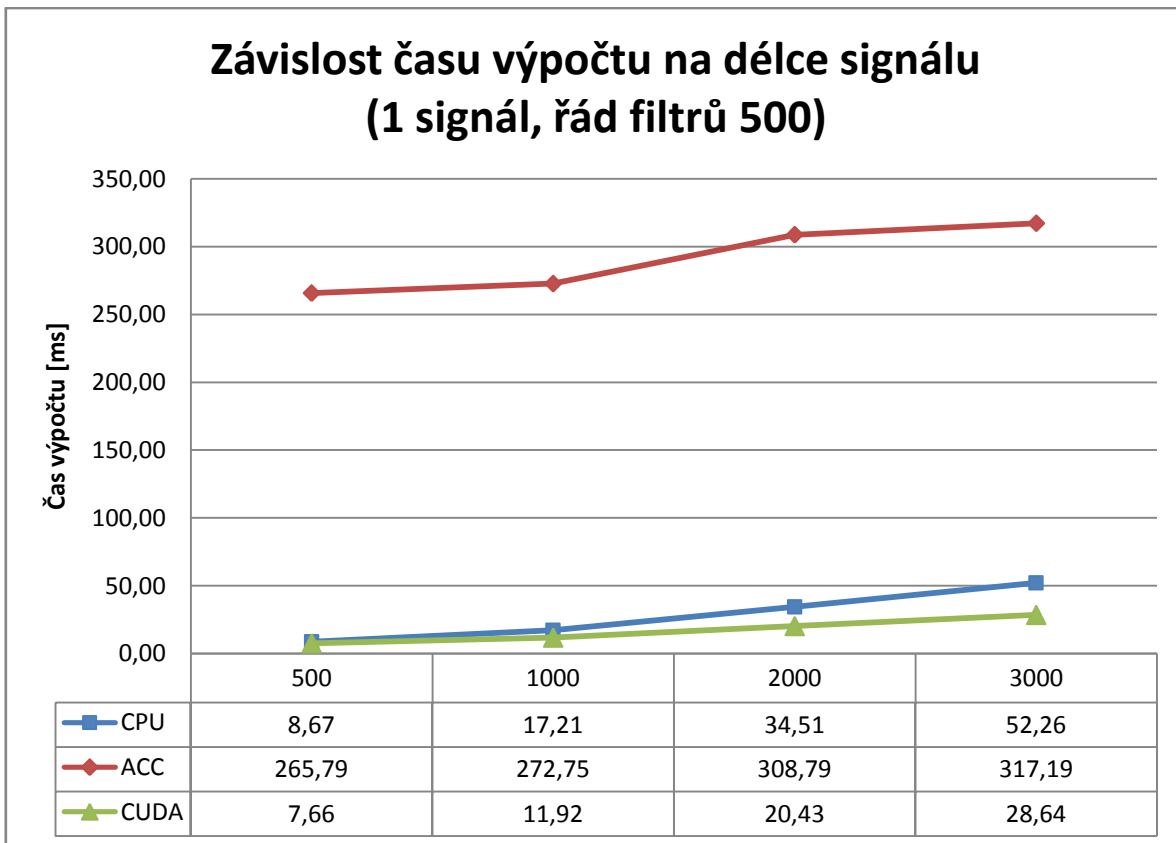


Obr. 8.1: Aplikace TestFir, IIR jednoho signálu

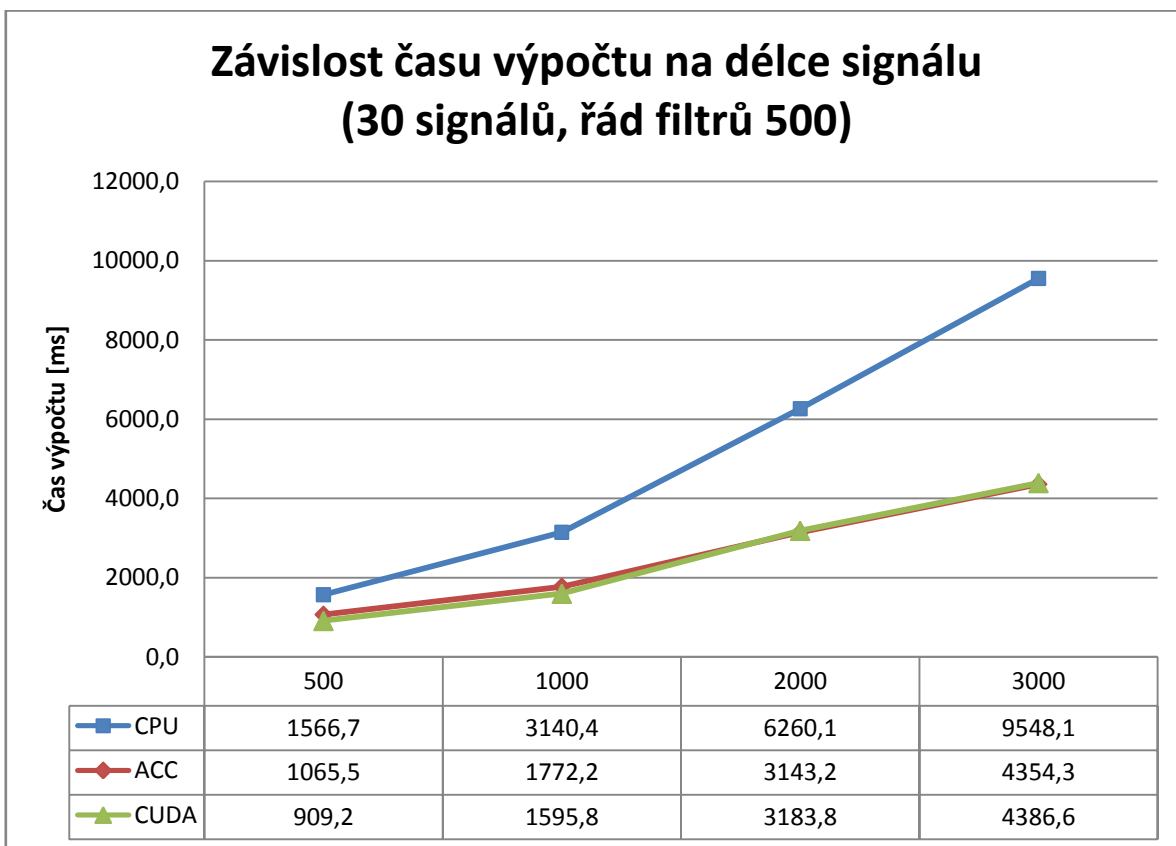


Obr. 8.2: Aplikace TestFir, IIR bloku 10 signálů

Následující grafy jsou měřeny pro stejná nastavení parametrů jako při měření FIR, navíc jsou řady obou filtrů (dopředního i zpětnovazebního) stejné a filtry obsahují stejné koeficienty.

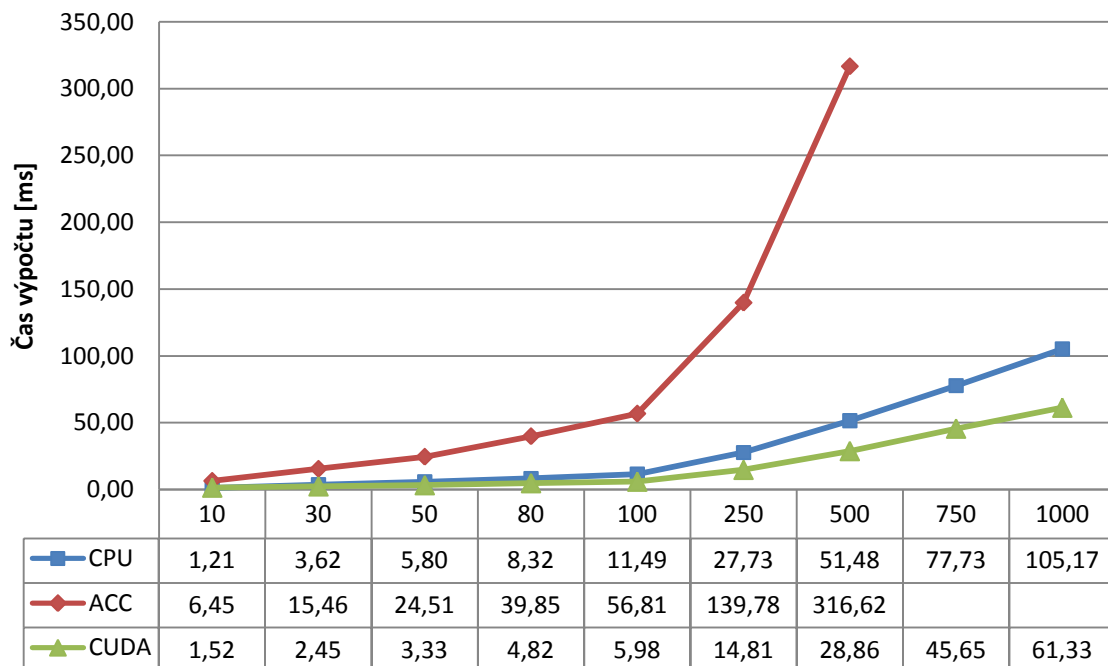


Graf 8.1: Závislost času výpočtu IIR na délce 1 signálu, při řádu filtrů 500



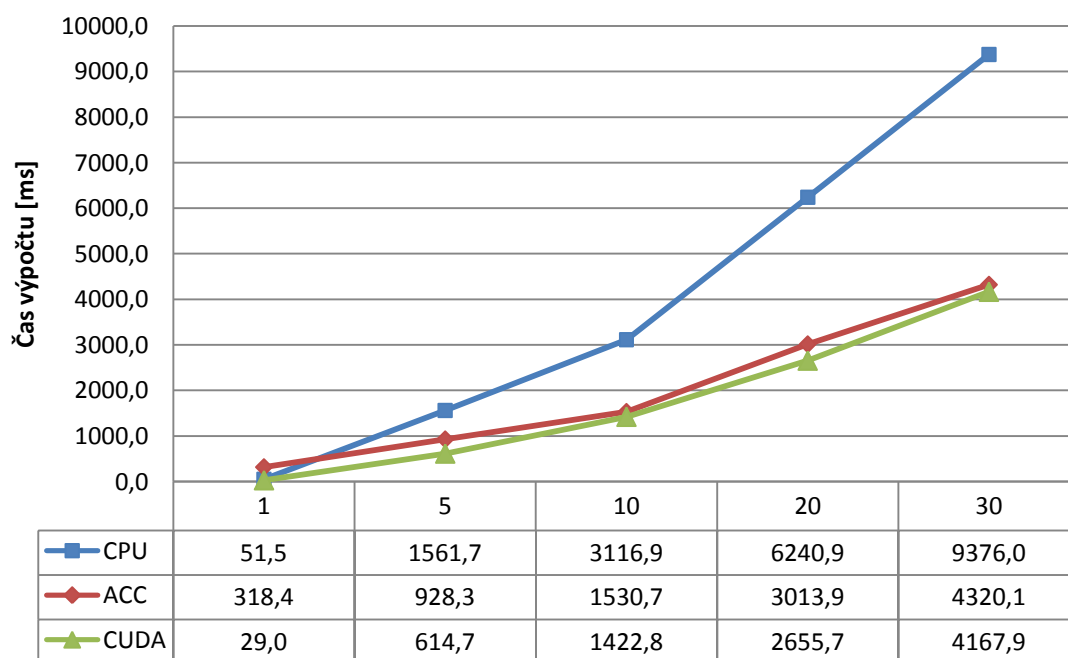
Graf 8.2: Závislost času výpočtu IIR na délce 30 signálů, při řádu filtrů 500

Závislost času výpočtu na řádu filtrů (1 signál o délce 3000)



Graf 8.3: Závislost času výpočtu IIR na řádu filtrů

Závislost času výpočtu na počtu signálů (délka signálu 3000, řád filtrů 500)



Graf 8.4: Závislost času výpočtu IIR na počtu najednou zpracovávaných signálů

Pokud budeme brát algoritmus v CUDA jako nejlepší implementaci pro GPU, lze všechny uvedené grafy výpočtu IIR shrnout do jedné věty: Nahrazením poloviny algoritmu IIR výpočtem na GPU lze ušetřit ve většině případů přibližně polovinu času výpočtu.

Z obou implementací, FIR i IIR, lze zhodnotit i použitelnost rozhraní MS Accelerator. Z grafů je patrné, že se vůbec nehodí pro výpočty s jen jedním signálem. Požadované zrychlení výpočtu u něj nastává pouze při zpracovávání více signálů najednou. Jinak je také zřejmé, že si toto rozhraní s sebou nese i neduhy způsobené závislostí na grafickém rozhraní. Jde jak o problémy při vyšších řádech filtrů, tak o přetrvávající omezení velikosti signálu maximální velikostí textury. To je důvod, proč je v testech použito max. 3000 hodnot v jednom signálu, i když by CPU a CUDA algoritmy zvládly více. Připojíme-li k tomu navíc, že rozhraní MS Accelerator již 2 roky není aktivně vyvíjené, pro seriózní programování GPGPU jej nelze doporučit.

9. Dvojměrná Sheppardova interpolace

Algoritmus 2D Sheppardovy interpolace lze použít pro určování hodnot signálu v ploše mezi několika zdroji signálu. Například pro vytváření potenciálových map mezi elektrodami EEG. Celý výpočet lze rozdělit na dvě části:

- vypočtení váhovacích matic – náročné, ale nutné počítat pouze na začátku a při změně souřadnic elektrod
- vypočtení výsledné mapy – z váhovacích matic a z aktuálních hodnot signálů na elektrodách, je to podstatně méně náročné na výpočetní výkon, ale nutné počítat pro každou novou hodnotu signálu

9.1 Implementace v HLSL

První implementace 2D Sheppardovy interpolace je realizována v shader-programovacím jazyku HLSL s použitím grafického rozhraní XNA, které využívá programovací jazyk C#.

9.1.1 Vypočtení váhovacích matic

Postup výpočtu lze rozdělit do několika kroků. Pro každý krok je pak vytvořen vlastní shader, provádějící příslušnou část výpočtu. Prvním krokem je zjištění vzdáleností všech bodů plochy od elektrody:

```
dist = float(sqrt(pow((texCoord.x - nCoord.x)/a, 2) + pow((texCoord.y - nCoord.y)/b, 2)));  
  
dist = pow(dist, distPower);  
  
if (dist < 0.1) dist = 0.0000001;  
return float4(1.0/dist,0,0,0);
```

Každý zpracovávaný pixel zná souřadnice `texCoord` své polohy ve výsledné textuře. Zadáme-li správně transformované souřadnice elektrody `nCoord`, pak je vypočítána jejich vzájemná vzdálenost `dist`. Poté výsledek umocníme koeficientem `distPower` a zkontrolujeme, zda vzdálenost není nulová nebo blízká nule. V takovém případě ji nahradíme velmi malým číslem, abychom v následujícím kroku nedělili nulou. Vzhledem k tomu, že se kód shaderu vykonává pro všechny pixely najednou, tak jsou najednou vypočítány vzdálenosti všech bodů plochy od souřadnice elektrody. Pokud máme více elektrod, pak se

výpočet provede pro každou elektrodu zvlášť. Výsledkem bude soubor textur, kde každý pixel bude mít hodnotu úměrnou vzdálenosti od příslušné elektrody. Tento soubor textur je ukládán jako pole textur. Bohužel nelze uložit texturu přímo do tohoto pole, ale je nutné ji nejdříve zkopírovat do normálního pole hodnot, a to pak uložit jako texturu do pole textur. Dochází tím ke zvyšování přenosu dat mezi grafickou a systémovou pamětí a ke snížení výpočetního výkonu.

Druhým krokem je součet všech výsledných textur z prvního kroku:

```
float4 a = tex2D(ScreenS, texCoord);
float4 b = tex2D(TexN, texCoord);
return a + b;
```

Zde se postupně textury odesílají pro výpočet na GPU tak, že se vždy sečtou a výsledek je použit jako jedna ze vstupních textur. Na tuto texturu se tedy nasčítají všechny ostatní.

Třetím krokem je konečně vytvoření vlastních matic vah (weight matrix). Ty získáme tak, že každou výslednou texturu z prvního kroku nanormujeme, tj. vydělíme součtovou texturou získanou v druhém kroku:

```
float4 a = tex2D(ScreenS, texCoord);
float4 b = tex2D(TexN, texCoord);
return b / a;
```

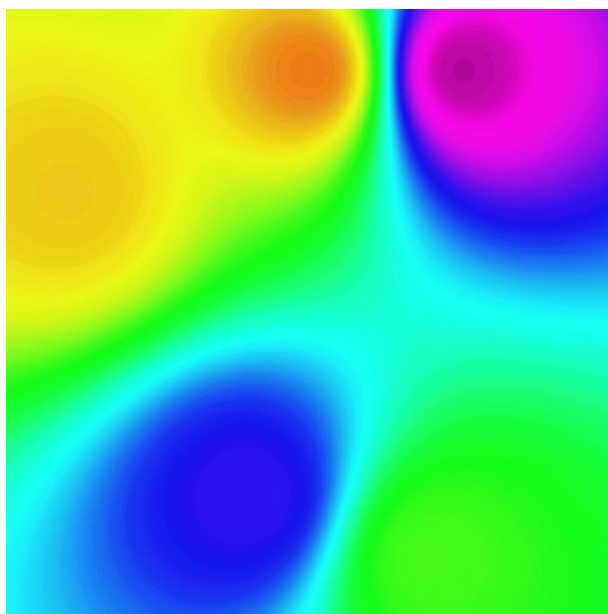
Výsledkem je soubor textur, kde každá textura tvoří matici vah pro příslušnou elektrodu. Opět je zde nutné přeukládat textury přes systémovou paměť, čímž dochází k dalšímu zpomalení.

9.1.2 Vypočtení výsledné mapy

Zde se vynásobí každá matice vah aktuální hodnotou signálu na příslušné elektrodě a všechny výsledné textury se sečtou:

```
float4 a = tex2D(ScreenS, texCoord);
float4 b = tex2D(TexN, texCoord);
return a + b * actualVal;
```

Stejně jako ve druhém kroku výpočtu matic vah jsou zde textury postupně na sebe nasčítávány. Navíc je však vždy hodnota z matice vah vynásobena aktuální hodnotou signálu `actualVal`. Výsledkem je jediná textura s hodnotami rozložení signálu v ploše mezi elektrodami. Tuto texturu je možné uložit jako matici do datového pole, nebo transformovat do libovolné barevné škály a zobrazit. Příklad takového zobrazení je na obr. 9.1.



Obr. 9.1: Výsledná mapa pro 5 elektrod

9.2 Implementace v CUDA

Druhá implementace 2D Sheppardovy interpolace je realizována v architektuře CUDA, za použití knihovny CUDA.NET pro programování v jazyce C#.

9.2.1 Vypočtení váhovacích matic

```
int globalIdx = blockIdx.x * blockDim.x + threadIdx.x;
int globalIDy = blockIdx.y * blockDim.y + threadIdx.y;
if (globalIdx >= width || globalIDy >= height) return;

__shared__ float sharedSum[16][16];
sharedSum[threadIdx.y][threadIdx.x] = 0.0f;
```

Ve výpočetním kernelu si nejdříve každý thread zjistí svoje globální souřadnice `globalIdx` a `globalIDy` a nepotřebné thready se vyloučí ze zpracování. Pak se inicializuje prostor ve sdílené paměti nastavením pole `sharedSum` a jeho vynulováním. Poté se spustí následující for cyklus provádějící výpočty pro každou elektrodu:

```
for (int i = 0; i < pocet; i++)
{
    // vzdalenost od souradnice elektrody
    float dist = sqrt(pow(globalIdx - souradnice[i].x, 2) + pow(globalIDy
        - souradnice[i].y, 2));
```

```

dist = pow(dist, distPower);

if (dist == 0.0f) dist = 0.0000001f;

weightMatrix[globalIDy * width + globalIDx + i * width * height] =
    1.0f / dist;

// soucet
sharedSum[threadIdx.y][threadIdx.x] += 1.0f / dist;
}

```

Z globálních souřadnic threadu a ze souřadnic elektrody se nejprve vypočítá vzdálenost `dist`. Ta se poté umocní koeficientem `distPower` a provede se kontrola, zde není vzdálenost nulová. V takovém případě ji nahradíme velmi malým číslem, abychom v následujícím výpočtu nedělili nulou. Inverzní hodnotu vzdálenosti uložíme do pole `weightMatrix` a zároveň přičteme do pole `sharedSum` ve sdílené paměti, kde se nám po projití všech elektrod nashromáždí součet všech matic s inverzními hodnotami vzdáleností. Následně se provede druhý for cyklus, který vydělí každou matici součtem `sharedSum`, čímž získáme konečné matice vah:

```

for (int i = 0; i < pocet; i++)
{
    // normalizace
    weightMatrix[globalIDy * width + globalIDx + i * width * height] /=
        sharedSum[threadIdx.y][threadIdx.x];
}

```

Pole váhovacích matic `weightMatrix` pak necháme v paměti grafické karty, aby nedocházelo ke zbytečnému přenosu dat. Zde může zůstat po celou dobu chodu programu, a pro výpočet výsledných map se předává pouze ukazatel na toto pole `d_weightMatrix`.

9.2.2 Vypočtení výsledné mapy

Kernel pro výpočet výsledné aktuální mapy pak podle ukazatele `d_weightMatrix` najde pole váhovacích matic v paměti a použije ho pro výpočet.

```

__shared__ float sharedMap[16][16];
sharedMap[threadIdx.y][threadIdx.x] = 0.0f;

for (int i = 0; i < pocet; i++)
{
    sharedMap[threadIdx.y][threadIdx.x] += weightMatrix[globalIDy * width
        + globalIDx + i * width * height] * actualValue[i];
}

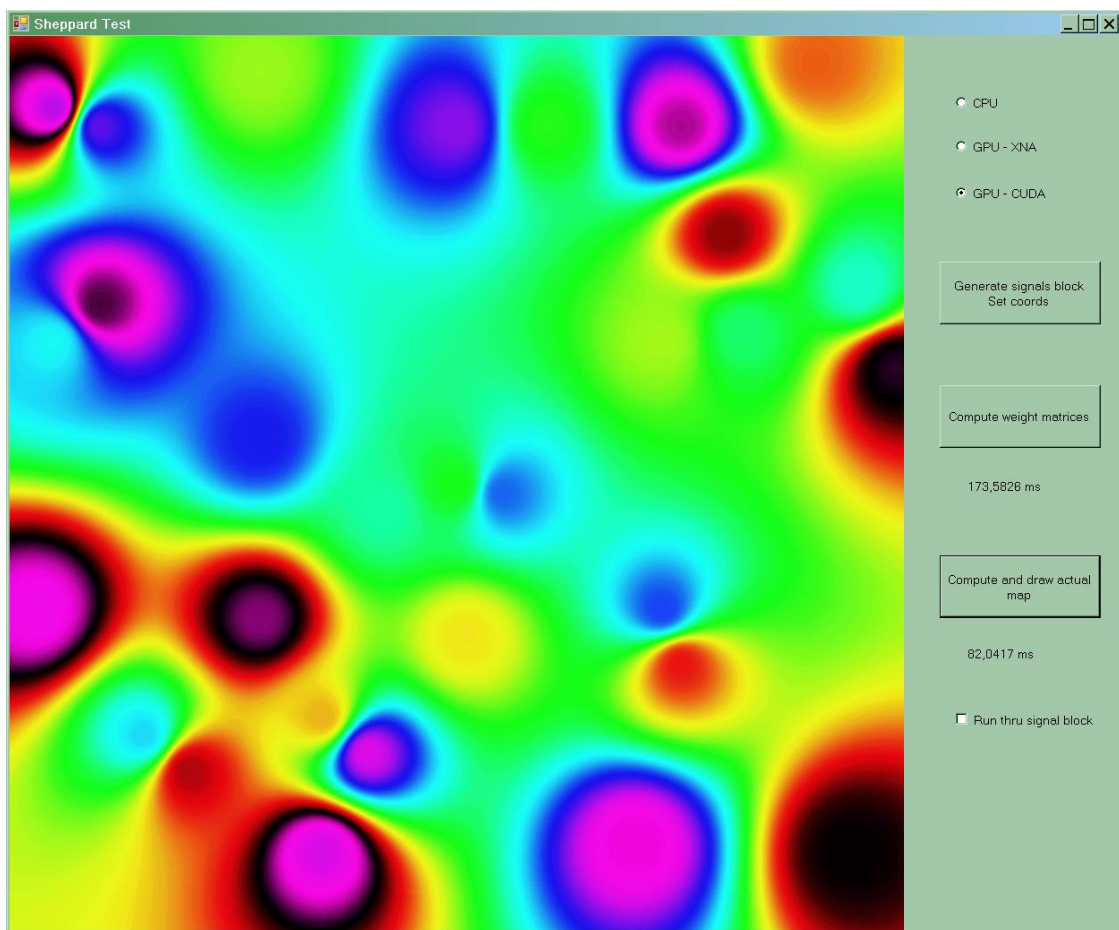
actualMap[globalIDy * width + globalIDx] =
    sharedMap[threadIdx.y][threadIdx.x];

```

Opět je při výpočtu použita rychlá sdílená paměť, tentokrát jako pole `sharedMap`. Do tohoto pole se nasčítají hodnoty z matic vah vynásobené aktuálními hodnotami signálu na elektrodách `actualValue`. Výsledky se pak uloží do výstupního pole `actualMap`, jež může být použito k vytvoření zobrazitelné textury.

9.3 Testování implementace 2D Sheppardovy interpolace

K testování výpočetního výkonu byla vytvořena aplikace Sheppard Test, ve které je možné přepínat mezi třemi verzemi algoritmu: pro CPU, pro GPU s použitím XNA (HLSL), a pro GPU s použitím CUDA. Program generuje signál na 30 elektrodách a měří časy potřebné k výpočtu váhovacích matic a výsledné mapy. Aktuální mapa se také zobrazuje v okně programu. Zaškrtnutím políčka Run lze automaticky procházet celý blok hodnot signálu a simulovat tak zpracování v reálném čase. Zobrazené potenciály se pak v mapě pohybují podle aktuálních hodnot na elektrodách.



Obr. 9.2: Program Sheppard Test se zobrazením mapy s 30 elektrodami

Výsledky měření výpočetních časů pro všechny tři verze algoritmu a obě fáze výpočtu jsou shrnuty v následující tabulce:

Část výpočtu	CPU	GPU - XNA	GPU - CUDA
Maticе Vah	16503 ms	1366 ms	174 ms
Aktuální Mapa	2444 ms	39 ms	82 ms

Tab. 9.1: Výsledné časy výpočtu 2D Sheppardovy interpolace

Jak je viditelné z tabulky, dochází použitím paralelního zpracování na GPU k výraznému zrychlení výpočtu jak maticе vah, tak aktuální mapy.

Algoritmus v XNA má výhodu v rychlejším výpočtu aktuální mapy díky přímému používání textur, které se na grafické kartě ukládají do rychlejší paměti textur. Nevýhodou má v pomalém výpočtu váhovacích matic, který je způsoben ukládáním a kopírováním mezivýsledků z grafické paměti do systémové paměti.

Algoritmus v CUDA má výhodu ve velmi rychlém výpočtu matic vah díky použití sdílené paměti umístěné přímo na grafickém procesoru a zpracování všech dat pouze v paměti grafické karty. Nevýhodou může být o trochu pomalejší výpočet aktuálních map oproti algoritmu v XNA (avšak stále mnohem rychlejší než výpočet na CPU).

10. Shrnutí všech výsledků

K přehlednému shrnutí všech hlavních výsledků pro každý zadaný algoritmus jsem vytvořil tuto tabulku:

Kolikrát je výpočet na GPU rychlejší než na CPU			
Algoritmus	1 signál	blok signálů	2D
FFT - rychlá Fourierova transformace	1,78	1,89	
Spektrogram		1,19	
Wavelet transformace	0,18	1,29	11,87
FIR - konečná impulzní odezva	6,07	20,11	
IIR - nekonečná impulzní odezva	1,82	2,25	
	matice vah	aktuální mapa	
2D Sheppardova interpolace	94,84	29,81	

Tab. 10.1: Zrychlení výpočtu algoritmu na GPU oproti CPU

Výsledné hodnoty jsou pro algoritmy naprogramované v CUDA, nastavení parametrů výpočtu (délky signálu, apod.) je vždy to nejlepší, nebo to nejpravděpodobněji využitelné.

Použitý hardware:

CPU: AMD Athlon 64 X2 3800+ (2,0 GHz)

GPU: nVidia GeForce 9600 GT, 512 MB

Z tabulky výsledků vyplývá, že implementace algoritmu na GPU nepřináší zvýšení výkonu pouze v případě wavelet transformace 1 signálu. Ve všech ostatních případech bylo dosaženo zrychlení výpočtu. Nejlepší výsledky jsou pro 2D wavelet transformaci, FIR a hlavně pro 2D Sheppardovu interpolaci, kde je u výpočtu váhovacích matic dosaženo 94násobného zrychlení v porovnání s výpočtem na CPU. Také 29násobné zrychlení výpočtu aktuální mapy je skvělý výsledek, který lze využít pro mapování signálů v reálném čase.

Všechny algoritmy jsou naprogramovány v nevizuálních třídách, připravených k použití ve vyvíjeném programu LiveMap určeném ke zpracování a vizualizaci biologických signálů v reálném čase.

Závěr

Tato práce měla za cíl uvést do problematiky paralelního zpracování signálů pomocí grafických adaptérů. Nejdříve bylo stručně vysvětleno, proč jsou obecné výpočty na grafických procesorech v současné době tak zajímavou oblastí. Pak byly uvedeny příklady již fungujících aplikací a provedena rešerše starších prací souvisejících s tématem zpracování signálů. Poté byl popsán proces zpracování dat na grafické kartě a vysvětleno, co umožňuje využití výpočetní síly grafických procesorů i pro negrafické algoritmy. Dále byl uveden přehled a stručný popis dostupných programovacích prostředků, použitelných pro vývoj algoritmů počítaných na grafickém procesoru. Poté byly popsány implementace zadaných algoritmů zpracování signálu. Byly to: rychlá Fourierova transformace, spektrogram, wavelet transformace, konečná a nekonečná impulzní odezva a dvojdimenzionální Sheppardova interpolace. U každého algoritmu byla vysvětlena jeho podstata, popsána implementace v několika možných programovacích prostředcích, byla vytvořena testovací aplikace a změřeny časy výpočtů. Naměřené časy byly vyneseny do grafů ke snadnému porovnání výpočetního výkonu GPU a CPU verzí algoritmů. Byly popsány výhody a nevýhody některých implementací a vysvětleny výsledky. Nakonec byla uvedena přehledná tabulka shrnující všechny důležité výsledky. Z této tabulky je patrné, že kromě jednoho případu bylo u všech algoritmů převedením výpočtu na grafický procesor dosaženo zvýšení výkonu. V nejlepším případě, při výpočtu 2D Sheppardovy interpolace, bylo získáno 94násobné zrychlení v porovnání s výpočtem na běžném procesoru.

Seznam použité literatury

- [1] *NVIDIA CUDA Programming Guide*. Version 2.2. URL: http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.2.pdf
- [2] Folding@home, distributed computing [online]. URL: <http://folding.stanford.edu/>
- [3] PhysX by NVIDIA [online]. URL: http://www.nvidia.com/object/physx_new.html
- [4] MARIACHAL-HERNÁNDEZ, J. G. – ROSA, F. – RODRÍGUEZ-RAMOS, J. M.: *Considerations on the FFT variants for an efficient stream implementation on GPU*. In Proceedings of the VISAPP 2006 – Image Formation and Processing. 2006.
- [5] FIALKA, Ondřej – ČADÍK, Martin: *FFT and Convolution Performance in Image Filtering on GPU*. In Proceedings of the Tenth International Conference on Information Visualisation, p. 609-614. IEEE Computer Society, 2006. URL: <http://www.cgg.cvut.cz/members/cadikm/papers/cadikm-iv06-gpu.pdf>
- [6] WONG, T.-T. – LEUNG, C.-S. – HENG, P.-A. – WANG, J.: *Discrete Wavelet Transform on Consumer-Level Graphics Hardware*. IEEE Transactions on Multimedia, Vol. 9, Nu. 3. 2007. URL: <http://www.cse.cuhk.edu.hk/~ttwong/papers/dwtgpu/dwtgpu.pdf>
- [7] YANG, Ruigang: *Scientific Computing on Commodity Graphics Hardware*. CIS 2004, LNCS 3314, p. 1100-1105. Springer-Verlag, 2004.
- [8] LEITSCH, Stefan – MARQUARDT, Oliver: *A CUDA-Supported Approach to Remote Rendering*. ISVC 2007, Part I, LNCS 4841, p. 724-733. Springer-Verlag, 2007.
- [9] CASSAGNABERE, Ch. – ROUSSELLE, F. – RENAUD, Ch.: *CPU-GPU Multithreaded Programming Model: Application to the Path Tracing with Next Event Estimation Algorithm*. ISVC 2006, LNCS 4292, p. 265-275. Springer-Verlag, 2006.
- [10] HARDING, Simon – BANZHAF, Wolfgang: *Fast genetic programming on GPUs*. EuroGP 2007, LNCS 4445, p. 90-101. Springer-Verlag, 2007.
- [11] NVIDIA CUDA Zone [online]. URL: http://www.nvidia.com/object/cuda_home.html
- [12] LÜCKE, Peter: *Volume Rendering Techniques for Medical Imaging*. 2005. URL: <http://campar.in.tum.de/twiki/pub/Students/DaLuecke/Diplomarbeit.pdf>
- [13] BrookGPU [online]. URL: <http://graphics.stanford.edu/projects/brookgpu/>
- [14] RapidMind [online]. URL: <http://www.rapidmind.com/>

- [15] Microsoft Research Accelerator Project [online]. URL: <<http://research.microsoft.com/research/downloads/Details/648909e1-cb85-46c4-9a94-3cca55971b1d/Details.aspx>>
- [16] Brahma [online]. URL: <<http://brahma.ananthonline.net>>
- [17] AMD Stream Computing [online]. URL: <<http://ati.amd.com/technology/streamcomputing/index.html>>
- [18] KHRONOS OpenCL [online]. URL: <<http://www.khronos.org/opencv/>>
- [19] MORELAND, Kenneth – ANGEL, Edward: *The FFT on a GPU*. In SIGGRAPH/Eurographics Workshop on Graphics Hardware 2003 Proceedings, p. 112-119. 2003. URL: <<http://www.cs.unm.edu/~kmorel/documents/fftgpu/fftgpu.pdf>>
- [20] JANSEN, T. – von RYMON-LIPINSKI, B. – HANSSEN, N. – KEEVE, E.: *Fourier Volume Rendering on the GPU Using a Split-Stream-FFT*. In Proceedings of the VMV'04, p. 395-403. 2004. URL: <<http://www.caesar.de/uploads/media/C04-3.pdf>>
- [21] *GPUFFTW: High Performance Power-of-Two FFT Library using Graphics Processors* [online]. URL: <<http://gamma.cs.unc.edu/GPUFFTW/index.html>>
- [22] Wikipedie, otevřená encyklopedie: Diskrétní vlnková transformace [online]. URL: <http://cs.wikipedia.org/wiki/Diskr%C3%A9tn%C3%AD_vlnkov%C3%A1_transformace>