Bakalářská práce

Tomáš Kříž

23. května 2014

Čestné prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne …….. ……..

Podpis autora práce

Anotace

Práce obsahuje problematiku a dokumentaci implementace algoritmu FFT pro grafické karty. V práci je popsán momentální stav různých implementací algoritmu FFT, popisuje řešení problému a ukazuje poznatky, které pomohou implementovat a zrychlit FFT pro grafické karty. Na konci práce je dokumentace a porovnání vybraných implementací s implementací popisované v této práci.

Anotation

This work contains problematics and documentation of implementation of FFT algorithm for graphic cards. There is described current situation of different implementations of FFT algorithm. Work contains resolution of problematics and shows knowledge about implementation and improvement of FFT algorithm for graphic cards. There is documentation and comparison of the implementation described at the end of this document.

Obsah

1 Úvod

Diskrétní foutrierova transformace je známá už mnoho let a v dnešní době je, ve své zrychlené formě známé jako rychlá fourierova transformace (FFT), nejpoužívanější technikou při zpracování různých dat. Není tedy příliš udivující, že na vypočítání této transformace vzniklo mnoho postupů, algoritmů, které v době, kdy se počítače rozšiřovaly, použila nejedna knihovna pro výpočet FFT. Dnes existuje plno variant a knihoven, které počítají FFT. Ať už jde o odlehčenou variantu KissFFT, léty prověřené a velmi používané FFTW, nebo vysoce optimalizované, ale placené MKL od firmy Intel.

V dnešní době se všechny výpočty soustředí na rychlost, protože přesnost už je většinou dostačující. Tento trend pomáhají udržovat nové technologie mezi které patří i výpočty na grafické kartě. Přestože tato technologie je stará více než pět let, pořád není příliš rozšířená. Knihoven na výpočet FFT je pomálu a je třeba přemýšlet, která knihovna nabízí co nejméně omezení. Cílem mé práce je použití takových prostředků, abych tyto rozdíly smazal a bylo možné tuto knihovnu použít bez ohledu na výrobce hardware nebo operačního systému.

Nakonec práce porovnám svoji implementaci s jinými implementcemi, abych zjistil zda je moje implementace vůbec použitelná, něčím výhodná a v kterém směru by se měly ubírat následné optimalizace a vylepšení. Porovnám její nejdůležitější parametry, kterými je přesnost a rychlost.

2 Existující implementace

Nyní představím implementace, které již existují a zároveň napíšu i jejich chyby. Na začátku této kapitoly zmíním implementace, které budu používat pro porovnávání a poté zmíním ještě několik dalších implementací, které jsou velmi známé.

2.1 FFTW

Toto je snad nejznámější implementace FFT na světě. Je rychlá a její přesnost je vysoká. V dnešní době už podporuje i SSE/SSE2 instrukce a více vláken. Od tohoto stavu už je jen krůček k masivní paralelizaci, kterou nabízí grafické karty a díky které jsou následující dvě implementace rychlejší. Tím jsem již také naznačil jediný problém této implementace. Tato implementace je pouze pro procesory a z toho plyne omezení rychlosti. V dnešní době už se totiž nezvyšuje frekvence procesoru, ale počet jeho jader. Přesto však počet jader procesoru je v řádu desítek, většinou jednotek, což je pouze zlomek oproti stovkám jader, které můžou nabídnout dnešní grafické karty.

2.2 CuFFT

Firma NVidia vydává knihovnu CuFFT [7] spolu s jejím frameworkem CUDA 3.3. Tato implementace je funkcemi velmi podobná již zmíněnému FFTW, ale narozdíl od něj je určená pro grafické karty. Její nevýhoda je dána jejím frameworkem, který je určený pouze pro grafické karty od firmy NVidia.

2.3 clAmdFft

Tato implementace pochází od firmy AMD, která používá a podporuje framework OpenCL 3.2. Díky tomu není tato implementace omezena na jednoho výrobce grafických karet a je možné ji spustit i na jiných zařízeních, které OpenCL podporují. Její hlavní nevýhodou je podpora pouze některých délek transformací. V technické dokumentaci [5] jsou uvedeny délky do 2^{24}, které lze rozložit na $2^x \times 3^y \times 5^z$, což je pouze velmi malé množství délek. Navíc některé dnešní grafické karty dokáží zpracovat i větší délky, pokud k tomu mají dostatek paměti.

2.4 Další zástupci

Nyní uvedu další zástupce, které však ve své práci nebudu porovnávat. Jako první uvedu Intel Mathematic Kernel Library (Intel MKL) [8]. Tato knihovna pochází od firmy Intel, která se zabývá výrobou procesorů a proto není překvapení, že MKL je knihovna určená pro procesory. Kromě FFT tato knihovna implementuje vektorové funkce, matematickou statistiku a lineární algebru. Jedná se však o profesionální knihovnu, a proto není zdarma. Její cena začíná na 499$.

Velmi často se též používá knihovna KissFFT [9]. Tato knihovna je velmi jednoduchá a často používaná pro prototypy programů. Její heslo je [9]"Keep it simple, stupid.". Je šířená pod BSD licencí a to znamená i přístupnost zdrojového kódu. Tato knihovna je však často nahrazována FFTW ve chvíli, kdy programátor vytváří oficiální a konečnou verzi programu.

Další implementace již uvádět nebudu, jelikož pro procesor jich existuje velké množství a pro grafické karty jsem v době vytváření této práce další nenašel.

3 Vývoj výpočtů na grafických kartách

Výpočty na grafických kartách umožnilo přidání unifikovaných shaderů do jader grafických karet, není proto divu, že právě v OpenGL vznikly první programy, které k negrafickým výpočtům používaly grafické procesory. Grafické karty se vyznačují výrazně vyšším výpočetním výkonem za který může masivní paralelizace.

Nevýhoda spočívá v jejím připojení na sběrnici PCI Express, jejíž přenosová rychlost je maximálně 15.75 GB/s (x16, verze 3.0). Navíc PCI Express je synchronní sběrnice a proto přenos nemusí začít okamžitě.

3.1 OpenGL

V OpenGL se od verze 2.0 objevila GLSL. V ní jsou definované vstupní a výstupní parametry na začátku souboru direktivami in, uniform a out. In značí vstup, out výstup a uniform označuje konstanty. Původně nebylo navrženo, že by direktiva out dovolovala vracet výstupní data zpět do procesoru, ale nebylo to ani zakázané. Díky tomu bylo možné chytrým navržením data vypočítat ve vertex shaderu a vrátit je jako texturu. Toto však bylo velmi náročné na znalost OpenGL a zkušenosti programátora.

3.2 OpenCL

Za vznikem OpenCL stojí skupina Khronos Group, která stojí i za standartem OpenGL. Není proto náhodou, že oba standarty mají definované sdílení paměti a umí spolupracovat. Standart OpenGL vznikl 18. listopadu 2008. Už od začátku byl navržený tak, aby funkce, které spouští na zařízeních, zvané kernel, byly napsány v jazyce C99 s rozšířením o vektorové formáty a synchonizační příkazy. Nyní už standart podporují nejen grafické karty, ale i velká část dnes prodávaných procesorů a OpenCL se tak stal standartem pro heterogenní systémy. Ve své práci jsem se rozhodl použít právě tento standart, protože je nezávislý na platformě a operačním systému. Nyní standart OpenCL podporují dokonce i některá mobilní zařízení, například tablety a chytré mobilní telefony. Nyní ještě uvedu několik dalších možností a důvody, proč jsem se rozhodl je nepoužít.

3.3 CUDA

Tento framework je soustředěný okolo firmy NVidia a těší se popularitě, která je podporována bohatou podporou ze strany firmy. Pod tímto frameworkem je napsáno plno knihoven, které NVidia zdarma dodává, včetně knihovny CuFFT, kterou budeme později porovnávat. Jeho hlavní nevýhoda je v jeho omezenosti. Přesto, že na tomto frameworku staví OpenCL, není možné použít pro výpočty jiná zařízení, než grafické čipy od Nvidie. Mezi další nevýhody patří třeba nutnost předkompilovaných kernelů.

3.4 Stream

Tento framework vznikl v dílně AMD v prosinci 2007. Osobně ho považuji za předchůdce OpenCL, jelikož byl navržen na základě ANSI C. Nicméně nedlouho poté, co Khronos Group vydal standart OpenCL, AMD prohlásil (9. prosince 2008), že vývoj Stream bude zastaven a místo něj bude podporován standart OpenCL.

3.5 Direct Compute

Tento framework je protějškem OpenCL, který vytvořil Microsoft podobně jako je Direct3D k OpenGL. Tento framework je téměř neznámý a podporovaný výhradně zařízeními na kterých běží operační systém od firmy Microsoft. Jedním z mých cílů byla nezávislost na operačním systému. Použil jsem proto ve zdrojovém kódu pouze standartní knihovny a OpenCL.

4 Představení frameworku OpenCL

Nyní blíže popíši standart OpenCL. Standart OpenCL vznikl v roce 2008. V roce 2010 se dočkal verze 1.1 a rok poté i verze 1.2, a 18. listopadu 2013 byla vydána verze 2.0. Ve verzi 1.1 přibyly například vektory velikosti 3 nebo podpora zadávání příkazů pro grafickou kartu z více vláken a pro více zařízení. Ve verzi 1.2 je možné zařízení rozdělit a vyčlenit na výpočty pouze část zařízení. To znamená, že pokud je výpočet časově náročný, můžeme grafickou kartu rozdělit a část nechat na zobrazení obrazu na monitoru. Ve verzi 2.0 přibyla podpora pro mobilní zařízení (mobilní telefony nebo tablety). Dále přibyly například atomické typy ze standartu C11, nebo sdílená virtuální paměť.

Verze pro kterou je tato knihovna programována je 1.1, protože grafické karty od firmy NVidia nepodporují vyšší verze. OpenCL se skládá z hlavičkových souborů a dynamické knihovny. Jelikož OpenCL je standart, jeho implementace se liší s různými výrobci. Například jsem pracoval s implementací od firmy NVidia, kde všechny obyčejně neblokující příkazy zablokovaly vlákno na velkou část času po který se příkaz vykonával. Následné blokující příkazy pak byly téměř zbytečné.
V OpenCL se rozlišuje zařízení na kterém chceme vykonávat výpočty dvěmi čísly. Jedno určuje platformu, neboli výrobce, zařízení a druhé označuje samotné zařízení.

OpenCL je definováno kontextem (Context). Je to vlastně inicializace knihovny. Kontext se vytváří nad zařízením, nebo skupinou zařízení pod stejnou platformou. Z toho plyne, že pokud chce uživatel použít více než jedno zařízení, musí tato zařízení být od stejného výrobce. Kontext spravuje příkazové fronty, programy

a paměť. Funkce, které patří ke kontextu jsou vykonávané téměř výhradně na procesoru a z velké části slouží pouze k získání informací o daném zařízení.

Ke každému zařízení je třeba vytvořit frontu příkazů. Velká část zařízení, které podporují OpenCL se nacházejí za sériovými sběrnicemi nebo jsou sdílené (například grafické karty jsou na PCIe sběrnici a používají se zároveň i ke kreslení grafiky), a proto je třeba vytvořit tuto frontu. Příkazy, které jsou přidávány do fronty, se provádějí postupně kdykoliv to zařízení dovolí. Do front lze přidávat příkazy typu pošli data, stáhni data, spusť program a blokovací příkazy vyčištění fronty a čekání na dokončení fronty příkazů.

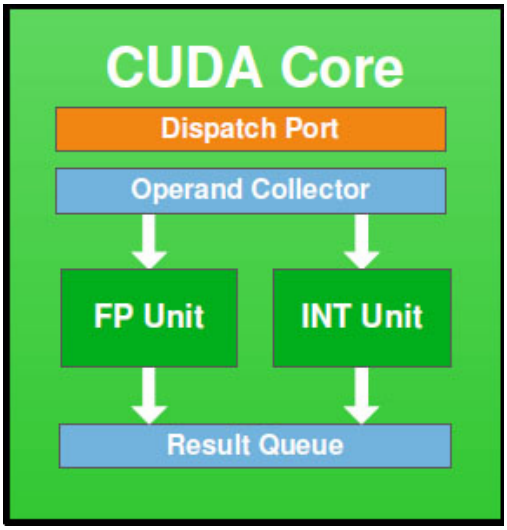
4.1 Abstrakce

Standart OpenCL poskytuje jistou úroveň abstrakce. Programátor nemusí znát úplné podrobnosti o zařízení pro které programuje a zároveň je tento systém možné použít i na jiné systémy, než jsou grafické karty. Tato úroveň je částečně i nebezpečná pro nováčky, protože jim skrývá mnoho věcí o tom, jak dané zařízení funguje. Tato skutečnost je nevíce poznat ve chvíli, kdy spustíme program pro grafickou kartu na procesoru, který podporuje OpenCL. Procesor, který má několik úrovní vyrovnávacích pamětí a jen malou paralelizaci je mnohdy několikanásobně pomalejší a méně využitý, než když by s použil jiný framework (například OpenMP), nebo optimalizace pro procesory. Naopak program optimalizovaný pro procesory bude na grafické kartě pomalejší, protože grafická karta má specifický způsob čtení z paměti. Další rozdíl je například v tzv. multiprocesoru (Obr. 1). Multiprocesor je jednotka uvnitř GPU. Tato jednotka obsahuje několik procesorů, které mají umí jednoduché aritmetické operace, dále multiprocesor obsahuje společnou FPU a řídící jednotku. V následujícím odstavci se pokusím přiblížit abstrakci, kterou představuje OpenCL a zároveň i přidat několik poznatků a faktů o grafických kartách.

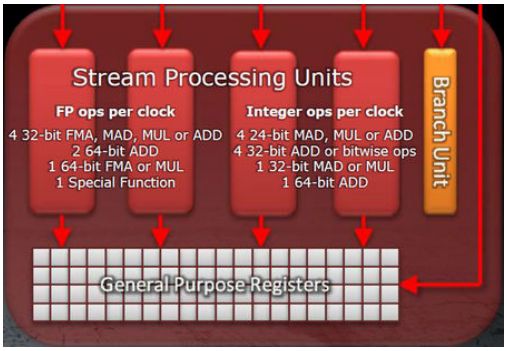
4.1.1 Vykonávací model

OpenCL rozeznává dva druhy programů. Takzvaný kernel, který je vykonávaný na zařízení a hostovací program, který je vykonáván na procesoru hostujícího počítače.

Kernel by se dal představit jako třída. Jeho instance se nazývá work-item a je definovaný indexem, to je celočíselný vektor jehož každá složka jde standartně od nuly do N. To znamená, že každý kernel vykonává stejný kód jenom s jinými parametry. Tyto instance mají přístup k registrům, které jsou v podstatě privátním prostorem. Tyto registry jsou jednotky paměti, do které je rychlý přístup, ale její velikost je značně omezená. Navíc není možné do ní nakopírovat data před



(a) CUDA Core architektura (NVidia) [11]

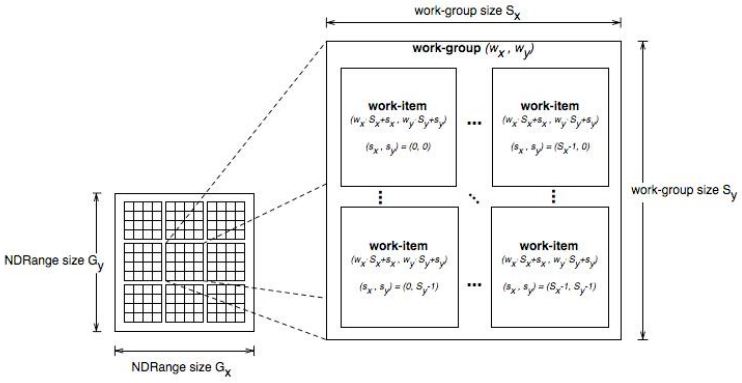


(b) VLIW4 architektura (AMD) [12]

Obrázek 1: Porovnání architektur multiprocesorů AMD a NVidia

spuštěním kernelu. Tato paměť se používá pro ukládání proměnných uvnitř work-itemu. Přístup k ní typicky trvá jeden takt.

Aby bylo možné program řídit i s menší přesností, sdružují se jednotlivé work-itemy do skupin - work-group. Tyto skupiny mají výhodu exkluzivního přístupu k lokální paměti. Lokální paměť je malý blok paměti, typicky 64kB, který spolu sdílí každá skupina. Typicky se používá buď k uložení dalších informací, které se už do registrů nevejdou nebo k uložení informací, které se mezi work-itemy sdílejí.



Obrázek 2: Příklad rozdělení work-itemů do skupin v celém rozsahu G [2]. Větší obrázek je v příloze dokumentu.

4.1.2 Paměťový model

V OpenCL jsou definované čtyři typy paměti. Globální, konstantní, lokální a privátní. Liší se jejich umístěním, typem přístupu k nim, jejich velikostí a jejich rychlostí.

Globální paměť je největší a také nejpomalejší. Velikost této paměti se většinou udává jako velikost paměti grafické karty a v dnešní době je v jednotkách gigabytů. K této paměti se přistupuje synchronní sběrnicí se šířkou několik desítek až stovek bitů. Tento typ přístupu je ideální pro data, která jsou seřazená, nebo na pořadí nezáleží. Nejrychlejší program proto umí efektivně využít všechny prvky při každém čtení, protože je možné díky tomu přečíst i desítky prvků a přiřadit je jednotlivým wok-itemům.

Konstantní paměť je speciální typ globální paměti do které může zapisovat pouze host. Tato část paměti je typicky upravená, aby bylo možné z ní rychle číst. Tato paměť je omezená a počítá se do globální paměti. Často bývá tato sekce umístěná na začátku globální paměti a je k ní možné přistupovat přes zvláštní sběrnici.

Lokální paměť je malá paměť velikosti několik desítek kB a přístup k ní je pouze z work-itemu. Tato část paměti je sdílená a uzavřená pro každou skupinu. Hostující aplikace ji může pouze vytvořit. Tato paměť je rychlejší, než obě předešlé.

Privátní paměť představuje registry. Je to malý kousek paměti, ke kterému lze přistupovat pouze z work-itemu a je vyhrazená každému zvlášť. Tato paměť je nejrychlejší ze všech a také nejmenší. Vejde se do ní jen několik bytů dat.

Vlastností paměti v OpenCL je, že není chráněná. To znamená, že je na programátorovi, aby zajistil její konzistenci. Jediná možnost je, že bude víc vláken najednou číst hodnotu. Jiný přístup k jednomu prvku paměti je nepřípustný. Bude to fungovat, ale stejně jako u vícevláknových aplikací není jistý výsledek. Tento přístup je rychlejší, než přístup s ochranou, který má většina operačních systémů. U grafických aplikací to totiž není potřebné chránit paměť a proto se ochrana paměti nechává na programátorovi.

Synchronizace se v OpenCL provádí dvěmi způsoby. Z hostující aplikace je možné synchronizovat pouze začátek a konec programu nebo paměťového přenosu, ale ne jejich průběh. Uvnitř kernelů je možné synchronizovat jednotlivé work-itemy na úrovni přístupu do paměti. Synchronizují se buď úplně všechny work-itemy skrz globální paměť, nebo se synchronizují pouze skrz lokální paměť. Když se synchronizují skrz lokální paměť, synchronizuje se pouze skupina itemů, tudíž zbytek není od výpočtů zdržován. Tato synchronizace je realizována pozastavením zasažených vláken pomocí speciální instrukce. Až všechna vlákna narazí na tuto instrukci, výpočet pokračuje. Blíže však tyto synchronizace popsané nejsou. Je tedy na samotné implementaci knihovny OpenCL, jak se s tímto problémem vypořádá.

5 Představení FFT

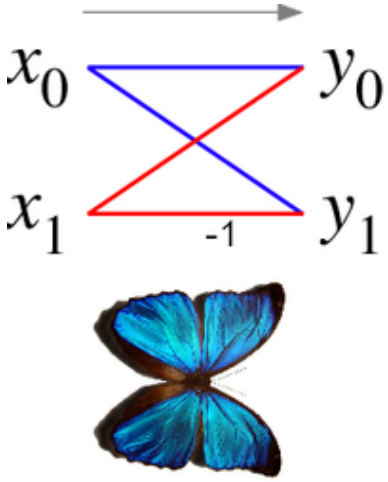
Algoritmus FFT je vylepšením původního algoritmu DFT (Diskrétní fouriérova transformace). Oba algoritmy převádí řadu vzorků libovolné funkce na seřazenou sadu amplitud komplexních funkcí sinus o různých frekvencích. Algoritmus DFT je zadán velmi jednoduchým vzorcem (1). Tento algoritmus je možné bez problémů paralelizovat. Nejrychlejší variantou je však paralelizování algoritmu FFT.

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-i2\pi kn/N}; k \in Z \tag{1}$$

$$X[N_2 k_1 + k_2] = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x[N_1 n_2 + n_1] e^{-\frac{2\pi i}{N_1 N_2}(N_1 n_2 + n_1)(N_2 k_1 + k_2)} \tag{2}$$

Je několik různých algoritmů FFT. Nejznámější je Cooley–Tukey algoritmus, který používá strategii rozděl a panuj. Jedna fáze Cooley–Tukey algoritmu rozdělí data délky $N = N_1 N_2$. V první fázi na N_1 menších FFT o délce N_2, po spočítání těchto menších FFT se každý prvek vynásobí proměnnou, které se říká Twiddle factor, zkráceně jen Twiddle. V druhé fázi se data přerozdělí na N_2 menších FFT o délce N_1. Nejznámější verze je tzv. Radix-2, kde N_1 je velikosti 2. Tato verze je známá, protože je jednoduše rozdělitelná. Data se dělí na sudé a liché indexy a pro každé zvlášť se počítá DFT (3). Tento algoritmus lze velmi jednoduše použít rekurzivně a tím se dostat k tomu, že se počítají FFT délky 2 (nebo jiného prvočísla: Radix-3, Radix-5, atp.), které jsou jednoduché. Takto malým FFT se kvůli jejich tvaru říká motýlci (Obr. 3).

$$X[k] = \sum_{m=0}^{N/2-1} x[2m] e^{-i2\pi k/N(2m)} + \sum_{m=0}^{N/2-1} x[2m+1] e^{-i2\pi k/N(2m+1)} \tag{3}$$



Obrázek 3: Diagram FFT velikosti 2 a jeho podoba s motýlem

Další algoritmy jsou například Bluesteinův a Raderův. Oba algoritmy se používají na výpočet FFT o délkách, které se špatně dělí. Algoritmy mají jeden společný znak, kterého se běžně využívá. Tyto algoritmy jde přepsat jako konvoluci.Výše zmíněné FFTW implementuje Raderův algoritmus. Já si vybral variantu Bluesteinova algoritmu, které se říká Chirp-Z transformace.

Chirp-Z transformace je ve své podstatě konvoluce. Konvolují se spolu dva signály, vstupní signál, ze kterého chceme vypočítat Fourierovu transformaci a takzvaný z-chirp, sinusoida s lineárně vzrůstající frekvencí. Tento signál je popsaný funkcí (4). Tato samotná modifikace ale neřeší problém u velikostí se špatnou nebo neexistující faktorizací. Narozdíl od normálního FFT však pomocí této úpravy mohu rozšířit délku polí a doplnit vstupní signál na libovolnou délku, která je alespoň dvojnásobně dlouhá. Díky tomu je možné s jednoduchým FFT, které má pouze faktorizaci velikosti 2, vypočítat fourierovu transformaci signálu libovolné délky. Mimo tohoto použití se chirp-z transformace používá pro přiblížení, zoom, transformace, kde poté mohou vystoupit frekvence, které na původním výsledku transformace nebyly vidět kvůli nespojitosti transformce, která je ze své podstaty dikrétní.

$$y[x] = e^{-i\pi x^2/N} = \cos\left(\frac{\pi x^2}{N}\right) - i\sin\left(\frac{\pi x^2}{N}\right) \tag{4}$$

Nyní vysvětlím, proč je možné pomocí konvoluce provést fourierovu transformaci a jakou roli v tom hraje výše zmíněný chirp-z signál. Celé je to velmi jednoduché, pokud se podíváme na funkci, která spočítá jeden člen diskrétní fourierovy transformace (1) a porovnáme ji se vzorcem, který spočítá jeden člen konvoluce (5). Je zřejmé, že pokud substitujeme $f[n]$ za x_n, potom $g[n-m]$ je právě chirp-z signál a celá konvoluce se počítá pouze na intervalu $\langle 0; N-1\rangle$ (6).

$$(f * g)[n] = \sum_{n=-M}^{M} f[m]g[n-m] \tag{5}$$

$$(f * g)[n] = \sum_{n=0}^{N} f[m]g[n-m] \tag{6}$$

Při použití výše zmíněného systému pro počítání FFT libovolné délky se vstupní data doplní neutrálním prvkem (většinou se doplňuje nulami, ale je možné i doplnit střední hodnotou signálu) do požadované velikosti a chirp-z signál se upraví. Velikost musí být alespoň dvojnásobná, čímž odstraníme chyby, které by mohly být vneseny samotnou konvolucí. Chirp-z signál se musí také rozšířit, aby byly velikosti stejné. Doplní se tak, že při velikosti $n > N - 1$ a zároveň $n < M - N$ se doplní nulami. V intervalu $\langle M - N; M \rangle$ bude doplněn o obrácený chirp-z signál, neboli $y((M-1)-n)$. Provede se konvoluce obou doplněných signálů a výsledkem je FFT o stejné velikosti, jako jsou původní vstupní data, na začátku výstupních dat konvoluce. Provést konvoluci je nejrychlejší za pomoci algoritmu FFT. Obě rozšířená pole se transformují, vynásobí a poté se provede zpětná transformace FFT. Toto je možné, protože násobení ve frekvenční oblasti se rovná konvoluci v časové oblasti [3].

5.1 Existující implementace

Nyní uvedu existující implementace, které budu používat na srovnávání s mou implementací. Jakožto zástupce knihoven pro procesory jsem vybral FFTW. V dnešní době jde o nejlepší a zároveň nejpoužívanější implementaci. Její nevýhoda spočívá v omezení na procesory (CPU).

Nemohu nevybrat implementaci CuFFT, kterou jsem již zmínil výše. Je to oficiální implementace FFT algoritmu pro CUDA framework, která pochází přímo od NVidie. Je rozšířená (používá ji například Matlab), ale je omezná pouze na procesory od NVidie.

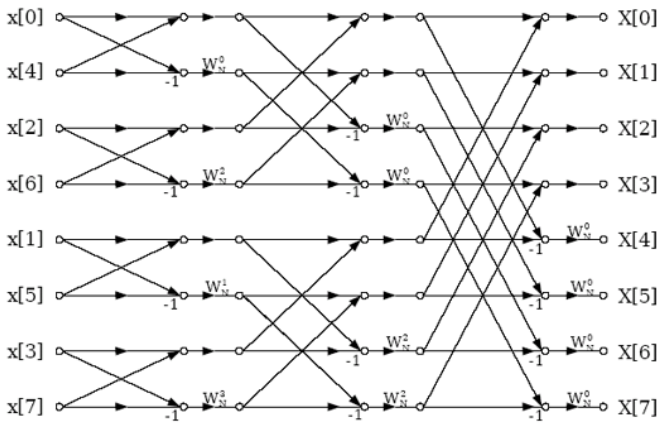
Jako poslední jsem vybral implementaci od konkurenta NVidie, AMD, která se nazývá clAmdFFT. Tato implementace je sice napsána pro OpenCL, ale má hned několik omezení. Například omezení velikostí vstupních dat pouze na násobky 2, 3 a 5. Dále má maximální velikost vstupních dat 2^{22}.

6 Realizace

Knihovna je realizována v jazyce C. Použity jsou pouze standartní knihovny, čímž zajistím nezávislost na operačním systému. Pro výpočty na grafických kartách

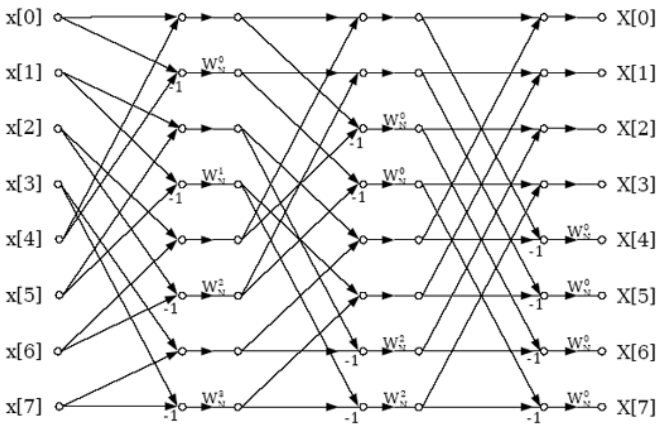
použiji framework OpenCL, který jsem již představil výše. Budu používat Cooley–Tukey algoritmus s faktorizací 2, 4, 8 a 16. Pro zbytek velikostí použiju Bluesteinův algoritmus. Oproti několikanásobnému používání clAmdFFT budu mít výhodu toho, že data zůstanou v paměti grafické karty. Cooley–Tukey algoritmus FFT je složený ze tří vnořených cyklů. Jeden iteruje přes vrstvy, druhý přes skupiny a třetí přes samotné "motýlky". Synchronizaci po vrstvách bude provádět část kódu v hostujícím zařízení.

Chytrým využitím symetrie skupin "motýlků" je možné vypočítat koeficienty z pořadí ve skupině. Potom každé vlákno, které GPU vytvoří bude počítat právě jednoho "motýlka". Zároveň je zajištěna paměťová bezkonfliktnost mezi vrstvami synchronizací.



Obrázek 4: Datový diagram radix-2 Cooley–Tukey (DIT) FFT algoritmu [1]

Na obrázku (Obr. 4) je znázorněn datový tok FFT. Je zde vidět nutnost seřadit vstupní indexy. GPU není určená k počítání FFT tudíž nemá hardware podporu automatického řazení. Tato možnost je ponechána uživateli a je možné ji naprogramovat. Druhou možností je seřadit samotný strom. Když seřadíme vstupní indexy, dostaneme následující graf (Obr. 5).



Obrázek 5: Datový diagram po seřazení vstupních indexů [1]

Toto řazení zkomplikuje paměťové uspořádání FFT, i přesto tato úprava neohrozí výkon implementace. Zato nebudeme muset na konci, nebo na začátku FFT řadit. V konečném důsledku tato úprava ušetří čas.

Inverzní transformace se počítá pomocí dopředné. Před samotnou transformací se vymění imaginární a reálné členy a po transformaci se zase vymění zpět. Poté se výsledek normalizuje.

6.1 Rozdělení programu mezi GPU a CPU

Program je samozřejmě potřeba rozdělit na část, která poběží na grafické kartě a bude počítat samotné FFT a program, který bude tyto výpočty řídit a poběží na procesoru počítače. Kernely, které běží na grafické kartě jsou malé a jednoduché. Postrádají totiž jakýkoliv zaváděcí kód. Proto je potřeba mít program, který zařídí všechno kromě samotného počítání. Důležitá je například i příprava dat a jejich nahrání do paměti grafické karty.

6.2 Programová část - GPU

Na grafickém čipu se počítá samotné jádro FFT, takzvaní "motýlci". Jednotliví "motýlci" jsou jednoznačně identifikováni pomocí indexu, který udává jejich pořadí od začátku. Všechny proměnné, které dávají dohromady proměnnou twiddle se vypočítají pomocí jejich indexu a konstanty, která udává vzdálenost skupin v právě počítané vrstvě. Z indexu též plynou i indexy vstupních a výstupních dat. Díky těmto krokům je zaručena vysoká paralelizace FFT.

V první verzi jsem se dostal ke kódu, který byl velmi podobný kódu [10] až na rozdělení výpočtu do funkcí a ukazatelovou aritmetiku. V kódu, který se vyskytuje

na těchto stránkách mě zaujalo především použití ukazatelové aritmetiky, protože tím se v mém kódu uvolnilo místo pro dvě proměnné, a proto jsem se rozhodl upravit svůj kód. Skončil jsem prakticky s totožnou verzí, jako uvedl pan Eric Bainville. Jediný rozdíl je v konstantě pro zpětný převod, pro který jsem se rozhodl použít metodu, která používá dopředné FFT.

Zbytek programového vybavení pro GPU obsahuje rozšíření původní metody pro počítání "motýlků". Tím jsem zajistil, že se najednou počítají dvě FFT místo jednoho. Další metody už jsou spíše podpůrného typu. Jsou zde metody pro normalizaci FFT, násobení dvou FFT uprostřed konvoluce a manipulace s daty, jako například rozšíření původního pole na nejbližší velikost 2^n a vygenerování stejně dlouhého z-chirp signálu, zabalení a rozbalení oddělených složek dat do vektorového formátu.

Nyní blíže rozeberu části kódů, které nejsou primitivní, jako například rozšíření pole, nebo generování z-chirp signálu. Nejprve rozeberu kód samotného kernelu Radix-2. Vstupní indexy jsou vždy stejné. Pouze se k ukazateli přičte, které vlákno se počítá a vstupní indexy do DFT o velikosti 2 jsou 0 a t, kde t je rovno počtu vláken. Výstupní indexy jsou trošku složitější. Je třeba přeskakovat indexy po skupinách motýlků. První výstupní index se počítá pomocí vzorce (7), kde & znamená bitovou operci AND. Proměnná k značí posun výstupních indexů, id je identifikátor jednotlivého vlákna na grafické kartě a p je proměnná, která značí velikost skupiny motýlků. Ve zdrojovém kódu je také pro jednoduchost nahrazeno násobení dvěmi bitovým posunem vlevo pro zvýšení rychlosti.

$$k = (id \times 2) - (id \ \& \ \ (p-1)) \tag{7}$$

Poté následuje funkce, která spočítá jednoduché DFT délky 2 a uloží výsledky. DFT délky dvě je jednoduché a lze ho spočítat podle vzorce (8), který je zadán ve vektorovém tvaru. V tomto vzorci X a Y představuje komplexní čísla. Ve zdrojovém kódu jsou tyto čísla uložena jako vektory. Tyto vektory se poté samozřejmě násobí stejným způsobem jako se násobí komplexní čísla.

$$[Y_1; Y_2] = \left[X_1 + W^{\frac{p}{N}} \times X_2; X_1 - W^{\frac{p}{N}} \times X_2\right] \tag{8}$$

6.3 Programová část - CPU

Tato část výpočtů se stará o logicky správné řazení vrstev, nahrávání dat a nastavování parametrů jako je vzdálenost skupin motýlků ve vrstvě. Dále se stará o spouštění podpůrných kernelů, o jejich správné dokončení a o přesuny dat mezi knihovnou a uživatelem knihovny. Pomocí této části si uživatel vybere zařízení, které bude výpočty provádět.

Inicializační funkce založí kontext, frontu příkazů pro zařízení a zkompiluje zdrojové kódy kernelů pro dané zařízení. Tím je hotová inicializace. Před samotným spuštěním výpočtů je potřeba naalokovat pole vstupních a výstupních dat a místo pro proměnné konstanty potřebné během výpočtu.

Další funkce, které se vykonávají na procesoru, zajistí přesun dat do paměti vybraného zařízení, což je nejčastěji realizováno nastavením periferie DMA. Stejným způsobem je také realizován přesun dat zpátky do paměti RAM.

Jak jsem již zmínil je třeba kernely zavést a spustit, což se děje sérií příkazů, při které se nastaví vstupní, výstupní a konstantní místa v paměti jako vstupní parametry kernelu a do fronty příkazů se přidá příkaz k provedení kernelu.

Neméně důležité jsou příkazy umožňující synchronizaci. Synchronizace se provádí zablokováním vlákna dokud se fronta příkazů nevyprázdní. Je možné také provést příkaz, který pouze vyprázdní frontu. Pokročilejší programy pro grafické karty mohou využívat systému událostí (tzv. Event), avšak v mé knihovně toto nebylo potřeba.

6.4 Interface knihovny

Knihovna má několik výčtových typů, které určují nastavení některých funkcí, struktury které funkce přijímají, nebo vracejí a několik funkcí, kterými se knihovna ovládá. Knihovnu je možné použít jako staticky, nebo dynamicky linkovanou v závislosti na potřebách uživatele.

V hlavičkovém souboru knihovny jsou také definovány návratové hodnoty funkcí.

6.4.1 Funkce

*void ** **_oclCreateInstance** (void)

Funkce vytvoří prázdnou instanci knihovny a alokuje proměnné potřebné pro identifikaci instance OpenCL. Do paměti se vytvoří proměnné pro ukazatele na kontext, frontu příkazů a velikost lokální skupiny.

Další funkce používají instanci knihovny vrácenou pomocí předcházející funkce. Tato instance je pouze ukazatel typu void na strukturu definovanou uvnitř knihovny. Tuto strukturu jsem uzavřel v knihovně proto, aby nebylo potřeba mít na počítači nainstalovaný SDK (Software development kit) pro OpenCL, pokud chce uživatel knihovnu pouze použít.

int **_oclInit** (void * instance, int platform, int device, bool mobile)

Funkce se pokusí inicializovat OpenCL pro zařízení zadané číslem platformy a zařízení. Parametr mobile je zde proto, že na zařízeních s menší pamětí (typicky grafické karty mobilních zařízení např. notebooků) je pro rychlejší výpočet nutné zakázat Radix-16, který má velikou náročnost na velikost registrů daného zařízení.

int **_oclJoinInstance** (void * instance, exportHandle, bool mobile)

Pokud už je OpenCL inicializované, je možné připojit tuto knihovnu a tím sdílet i paměťový prostor. Je potřebná instance knihovny, správně naplněná struktura exportHandle a zda se jedná o mobilní zařízení (viz. výše).

OCLArray **_oclCreateBuffer** (void * instance, int size, clFftType, writeType, int * error)

Pokud je již OpenCL nainicializované, tahle funkce naalokuje prostor na grafické kartě. Pro naalokování je potřeba velikost (počet prvků), typ pole z výčtového typu clFftType, typ přístupu k poli z výčtového typu writeType a ukazatel na celočíselnou proměnnou do které bude vrácena chyba. Pokud je vrácená chyba po zavolání rovna nule, funkce vrátí označení naalokovaného úseku paměti.

int **_oclWriteToBuffer** (void * instance, OCLArray buffer, int size, clFftType, const void * data)

Pomocí této funkce se nastaví nahrávání dat do paměti vybraného zařízení. K úspěšnému zavolání je potřeba mít správně naalokovaný prostor v paměti a nepřekročit velikosti naalokovaných pamětí. Tato funkce pouze nastaví kopírování, které může stále probíhat i poté, co funkce vrátí hodnotu nula a proto je potřeba zavolat před použitím blokovací funkci (viz. níže). Parametry této funkce jsou instance knihovny, označení naalokovaného pole, velikost, typ a ukazatel na data v RAM

int **_oclPackToBuffer** (void * instance, OCLArray buffer, int size, float * real, float * imag)

Pomocí této funkce se nastaví nahrávání dat do paměti vybraného zařízení. K úspěšnému zavolání je potřeba mít správně naalokovaný prostor v paměti a nepřekročit velikosti naalokovaných pamětí. Tato funkce pouze nastaví kopírování, které může stále probíhat i poté, co funkce vrátí hodnotu nula a proto je potřeba zavolat před použitím blokovací funkci (viz. níže). Narozdíl od předchozí funkce, tato funkce použije oddělené pole pro reálnou a imaginární složku, nahraje je na grafickou kartu, kde je sloučí do vektorového typu, který výpočty na grafické kartě používají. Použité parametry jsou instance knihovny, označení naalokovaného prostoru, délka pole a dvě pole s reálnou a imaginární částí dat.

int **_oclReadFromBuffer** (void * instance, OCLArray buffer, int size, clFftType, void * data)

Pomocí této funkce se nastaví nahrávání dat z paměti vybraného zařízení do RAM. K úspěšnému zavolání je potřeba mít správně naalokovaný prostor v paměti a nepřekročit velikosti naalokovaných pamětí. Tato funkce pouze nastaví kopírování, které může stále probíhat i poté, co funkce vrátí hodnotu nula a proto je potřeba zavolat před použitím blokovací funkci (viz. níže). Parametry této funkce jsou stejné jako u funkce _oclWriteToBuffer.

int **_oclUnpackFromBuffer** (void * instance, OCLArray buffer, int size, float * real, float * imag)

Pomocí této funkce se nastaví nahrávání dat z paměti vybraného zařízení do RAM. K úspěšnému zavolání je potřeba mít správně naalokovaný prostor v paměti a nepřekročit velikosti naalokovaných pamětí. Tato funkce pouze nastaví kopírování, které může stále probíhat i poté, co funkce vrátí hodnotu nula a proto je potřeba zavolat před použitím blokovací funkci (viz. níže). Narozdíl od předchozí funkce, tato funkce použije vektorový typ, na grafické kartě je rozdělí a zkopíruje do oddělených polí pro imaginární a reálnou složku. Parametry jsou stejná jako u_oclPackToBuffer.

int **_oclTransform** (void * instance, OCLArray input, OCLArray output, int size, bool fwd)

Touto funkcí se spustí transformace. Proměnná fwd je pro určení směru transformace. True značí dopřednou a false zpětnou transformaci. Další parametry jsou instance knihovny, vstupní a výstupní prostor alokovaný na grafické kartě. Transformovat lze pouze pole ve vektrovém formátu.

int **_oclBlock** (void * instance)

Při zavolání této funkce se volající vlákno zablokuje a čeká na vyprázdnění fronty příkazů. Většina příkazů pro OpenCL, které přímo upravují paměť je neblokující a pouze se do fronty příkazů přidá příkaz k provedení, který je proveden ve chvíli, kdy je zařízení volné.

int **_oclSetLocalItemSize** (void * instance, int)

Funkce nastaví vnitřní proměnnou, která ovlivňuje velikost skupiny. Tato velikost určuje počet vláken, které se najednou provádí. Ideální je, když je tato hodnota mocninou dvojky. Zároveň není vhodné, aby byla maximální, protože registry jsou omezené a čím je program náročnější, tím víc je potřeba registrů, ale protože počet registrů je omezený, klesá potom počet vláken, který lze spustit najednou.

int **_oclGetMaxLocalItemSize** (void * instance)

Funkce vrátí maximální počet vláken, který lze spustit na zařízení.

int **_oclGetPlatforms** (ocl_platforms **)

Funkce vrátí strukturu ve které jsou uložené všechny dostupné zařízení, které podporují OpenCL. Tato funkce sama alokuje strukturu, a proto není třeba alokovat strukturu předem.

int **_oclDestroyBuffer** (void * instance, OCLArray *)

Funkce vrátí paměť alokovanou funkcí _oclCreateBuffer.

int **_oclDeinitializeInstance** (void * instance)

Funkce odpojí OpenCL a vrátí alokované prostředky jako například paměť konstant a zkompilované kernely.

int **_oclDeinitialize** (void * instance)

Funkce vrátí alokovanou paměť knihovny samotné. Zavoláním této funkce se vrátí i poslední zbytky paměti, které knihovna zabírá.

6.5 Ukázka použití knihovny

Používání knihovny jsem se snažil zjednodušit, přesto je třeba znát některé základní znalosti o OpenCL, jako například posloupnost inicializací. Jednoduchý příklad je uvedený v příloze (Listing: 2). V příkladu je ukázáno, jak na grafickou kartu nakopírovat data rozdělené do dvou polí, následné spuštění transformace a nakopírování polí zpět. Dále je zde ukázána inicializace na implicitní zařízení a poté i vrácení veškeré alokované paměti.

7 Výsledky

Výsledky budu porovnávat dvěmi způsoby. Nejprve porovnám přesnost. Pro jednoduchost budu porovnávat přesnost bez ohledu na rychlost. U implementací FFT je známé, že si vždy autor vybere jednu nebo více forem výsledků, které většinou vybírá tak, aby co nejvíce odpovídaly požadavkům. Já vybral normalizované FFT a výslednou přesnost budu porovnávat s velmi známým programem Matlab, který používá knihovnu FFTW.

Rychlost budu následně porovnávat na poli náhodných prvků bez kontroly přesnosti. Připomínám, že pro porovnání rychlosti použiji knihovny FFTW, CuFFT a AmdClFft. Přesnost výše uvedených implementací uvádí sami autoři na svých internetových stránkách, a není cílem této práce kontrola autorů.

Obě části jsem měřil od velikosti 2^6 do velikosti 2^{24} pro délky rovné 2^x. Dále jsem zvolil 18 prvočísel, která jsou blízko velikostem 2^x.

7.1 Přesnost

Přesnost je počítána podle vzorce, který použili tvůrci FFTW [4]. Vzorec (9) počítá relativní chybu vůči předem vypočítanému a přesnému DFT. Jeho druhá část je vlastně p-norma, kde pro ověření použiji $n = \infty$. Tento speciální případ se nazývá maximová norma. Maximová se nazývá proto, že $|x|_\infty = \max(x_i)$. Znamená to, že budu udávat přesnost podle nejhoršího výsledku. Data pro jejich velikost nemohu přiložit, protože jeden datový soubor s daty délky 2^{24} je veliký 256 MB. Dohromady jedna verze velikostí 2^x a prvočíselných délek má 1,5 GB. Místo toho přikládám zdrojový kód, který jsem použil pro generování dat (Listing: 1).

$$\frac{\sum |x(i) - ref(i)|}{\sum |ref(i)|} \tag{9}$$

V tabulce 1 jsou zapsány odchylky od výstupu programu Matlab na stejných datech. Je zřejmé, že v programu bude chyba, protože přesnost pro prvočísla prudce klesá s velikostí. Může se také jednat o zaokrouhlovací chybu, která se několikanásobně sečetla a vynásobila. Přesnost pro velikosti 2^x ale odpovídá zaokrouhlovací chybě. Přesnost formátu float ve kterém je program napsán je dle normy IEEE 754 je 24 bitů mantisy a 8 bitů exponent. Přesnost je tedy $\log_{10}(2^{24}) = 7,225$ desetiných míst. Pro formát double je na většině grafických karet prodávaných dnes rychlost pouze poloviční. Starší grafické karty nemusí formát double podporovat vůbec, nebo jsou ještě víc pomalejší. Toto omezení pochází z nepotřebnosti formátu double při grafických výpočtech a proto je počet FPU (floating point unit) na grafických kartách malý. Nejčastěji jeden multiprocesor má pouze jednu FPU podporující formát double.

x	2^x	Prvočíslo
6	1,2874e-007	2,1330e-005
7	2,2058e-007	4,0088e-005
8	2,4579e-007	1,1402e-004
9	3,4087e-007	3,0382e-004
10	4,7064e-007	8,7210e-004
11	5,2492e-007	2,0099e-003
12	6,6258e-007	5,6410e-003
13	8,9960e-007	1,3859e-002
14	1,0657e-006	4,1791e-002
15	1,3571e-006	9,8840e-002
16	1,6630e-006	5,7355e-001
17	1,8882e-006	1,5937e+000
18	2,2652e-006	1,8255e+000
19	2,6342e-006	4,6980e+000
20	3,0899e-006	1,3117e+001
21	3,4738e-006	3,6309e+001
22	3,9754e-006	9,7194e+001
23	4,5408e-006	2,5902e+002
24	7,1841e-006	5,7174e+002

Tabulka 1: Přesnost oproti výstupu z funkce $fft()$ programu Matlab

Další vývoj této knihovny by se tedy měl ubírat především tímto směrem, protože odchylky během počítání s prvočísly jsou příliš veliké. Přesnost je velmi

často nejdůležitější parametr podle kterého si podobné kniovny vybíráme.

7.2 Porovnání rychlosti

Rychlost jsem porovnával včetně přesunů v paměti a pouze pro přesnost single(float), neboli číslo s plovoucí desetinou čárkou o velikosti 32b. Tím jsem trošku zvýhodnil FFTW, které pracuje na procesoru a nemusí přesouvat data, ale jak bude vidět, tak ani tato výhoda nepomohla. FFTW jsem spouštěl na všech 8 jádrech. Jako procesor jsem použil Intel Core i7-3630QM. Grafická karta byla NVidia GeForce GTX 660M. Jelikož grafická karta je v mobilní verzi, výkony na stolních počítačích nebo případných serverech by tedy měly být větší. Přesto považuji toto srovnání za validní, protože všem implementacím byl dán stejný výpočetní výkon. Nakonec doplním, že jako operační systém jsem použil Windows 8 a zdrojové kódy byly zkompilovány s Microsoft Visual Studio 2010 se studentskou licencí.

Samotné měření rychlosti jsem prováděl z volajícího vlákna za pomocí time.h. Pro tento způsob jsem se rozhodl proto, abych do porovnávání zahrnul i režii, která je například na grafických kartách pro malé velikosti (například 2^9) tak velká, že většinu času není grafická karta využitá.

7.2.1 Délky 2^x

Rychlost jsem měřil pomocí stejného postupu pro všechny čtyři implementace. Inicializaci jsem do času nezapočítával. Taktéž jsem nezapočítával fázi plánování. Algoritmus změřil čas padesáti transformací, který jsem poté vydělil padesáti a dostal výsledný čas, který je vidět v tabulce 2. Z tabulky si můžeme povšimnout, že všechny implementace pro grafické karty jsou rychlejší, než výpočet na procesoru i přesto, že je nutné data přesouvat do paměti grafické karty. Výhody grafických karet jsou ale nejvíce znatelné při větších velikostech dat, kde se nároky na výpočet zvětšují a režie ze strany procesoru se stává zanedbatelnou oproti celkovému času výpočtu. Nejlépe si vedla implementace od firmy NVidia, která byla nejrychlejší.

7.2.2 Délky podle vybraných prvočísel

Jak už jsem dříve uvedl, implementace firmy AMD nemohla být porovnána, protože nepodporuje prvočíselné velikosti vstupních polí. Postup byl stejný jako u velikostí 2^x, jen velikost byla jiná. Výsledné hodnoty jsou uvedené v tabulce 3. Můžeme si povšimnout, že u FFTW je posledních pět hodnot nula. Algoritmus FFTW z neznámého důvodu nezvládl vytvořit plán pro daná prvočísla a skončil s neznámou chybou a oznámením, že se mám obrátit na autory. Nepomohlo ani úplné odstranění kódu, který dovoloval spuštění FFT na více vláknech. Tudíž konečné prvočíslo zvládly transformovat pouze dvě implementace. Mohu ještě poukázat

x	FFTW	CUFFT	clAmdFFT	CLFFT
6	1,54	0,00	0,22	0,36
7	1,84	0,00	0,22	0,40
8	2,08	0,00	0,22	0,42
9	2,16	0,00	0,24	0,42
10	2,36	0,00	0,28	0,52
11	2,56	0,00	0,36	0,54
12	2,80	0,32	0,42	0,54
13	3,24	0,00	0,78	0,60
14	3,92	0,30	0,92	0,68
15	4,64	0,30	1,06	0,80
16	6,68	0,64	1,44	1,34
17	8,22	1,56	2,92	2,06
18	11,30	1,88	6,90	2,78
19	27,42	2,82	11,32	4,34
20	59,72	5,00	22,22	7,46
21	113,58	10,50	43,68	13,26
22	216,92	20,60	87,26	27,40
23	413,22	42,54	187,88	53,66
24	799,96	85,94	379,06	104,56

Tabulka 2: Rychlost implementace algoritmu FFT na délkách 2^x. Hodnoty jsou v milisekundách.

na podobnost mezi hodnotami, které odpovídají přibližně trojnásobku oproti podobným velikostem z tabulky 2. Toto poukazuje na podobný, nebo stejný postup transformování prvočíselných velikostí u knihovny CUFFT.

Prime	FFTW	CUFFT	CLFFT
61	1,22	0,12	1,84
127	1,36	0,10	2,00
257	1,52	0,12	2,06
509	1,70	0,14	2,14
1021	1,98	0,14	2,20
2039	2,42	0,14	2,06
4093	3,22	0,18	2,50
8191	4,70	0,22	2,56
16381	7,36	0,56	2,82
32769	15,48	0,78	4,56
65537	19,32	1,70	5,52
131071	36,08	2,08	6,00
262147	76,96	3,46	13,18
524287	242,16	5,66	14,56
1048573	0,00	11,88	25,34
2097143	0,00	24,18	49,24
4194301	0,00	51,78	93,02
8388593	0,00	106,76	179,96
16777213	0,00	226,94	379,18

Tabulka 3: Rychlost implementace algoritmu FFT na polích o délkách prvočísel. Hodnoty jsou v milisekundách.

8 Závěr

Implementace ještě není plně odladěná, přesto však splňuje základní požadavky s kterými jsem ji navrhoval. Hlavičkové soubory používá pouze standartní a běžné pro každý kompilátor jazyka C/C++. Pro práci s grafickými kartami používá pouze framework OpenCL, který není omezen výrobcem a je jasně definovaný pomocí jeho dokumentace a standartu.

Velmi mě překvapilo, že moje implementace si v rychlostím testu nevedla špatně i proti tak nerovnému soupeři jako je CUFFT, které je optimalizované na grafické karty firmy NVidia a pravděpodobně používá i speciální instrukce, kvůli kterým je framework CUDA uzavřený. Přesto však další vývoj se bude soustředit na vyhledání chyb a ošetření zaokrouhlovacích chyb formátu float. Dále by se vývoj mohl ubírat vytvořením kernelů pro počítání primitivních FFT až do délky 15, případně přidat velká FFT, jelikož CUFFT používá rozklad o velikosti až 256.

9 Symboly, zkratky, pojmy

DFT Discrete Fourier Transform - Diskrétní Fouriérova transformace

FFT Fast Fourier Transform - Rychlá Fouriérova transformace

PCI Peripheral Component Interconnect

CPU Central Processing Unit

GPU Graphic Processing Unit.

SSE Streaming SIMD Extensions

SIMD Single Instruction, Multiple Data

MKL Math Kernel Library

GLSL OpenGL Shading Language

DMA Direct Memory Access

Kernel je funkce, kterou lze spustit na grafickém procesoru pomocí příkazu z CPU. Může mít vstupní parametry a konstantní parametry. Zpravidla má alespoň jeden výstupní parametr.

Host neboli hostující zařízení je jednotka, která ovládá výpočty frameworku OpenCL, většinou se jedná CPU

Reference

[1] *Alternate FFT Structures* - Douglas L. Jones
`http://cnx.org/content/m12012/latest/`, Figure 1, Figure 2

[2] *The OpenCL Specification - Version 1.1* - Khronos Group
`http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`, Figure 3.2

[3] *The Scientist and Engineer's Guide to Digital Signal Processing* - Steven W. Smith, Ph.D.
Kapitola 18. FFT Convolution
`http://www.dspguide.com/ch18/2.htm`

[4] *FFT Accuracy Benchmark Methodology* - Matteo Frigo, Steven G. Johnson
`http://www.fftw.org/accuracy/method.html`

[5] *AMD OpenCL Libraries* - Advanced Micro Devices
Internetová stránka AMD knihoven pro OpenCL. Dokumentace a knihovny ke stažení dole na stránce.
`http://developer.amd.com/tools-and-sdks/opencl-zone/opencl-libraries/amd-accelerated-parallel-processing-math-libraries/`

[6] *CUDA Parallel Computing Platform* - NVIDIA Corporation
`http://www.nvidia.com/object/cuda_home_new.html`

[7] *cuFFT* - NVIDIA Corporation
`https://developer.nvidia.com/cuFFT`

[8] *Intel Math Kernel Library* - Intel Corportion
`https://software.intel.com/en-us/intel-mkl`

[9] *KissFFT* - Mark Borgerding
`http://kissfft.sourceforge.net/`
`http://sourceforge.net/projects/kissfft/`

[10] *A simple radix-2 kernel* - Eric Bainville
`http://www.bealto.com/gpu-fft2_opencl-1.html`

[11] *NVidia CUDA Core GTX 470* - Architektura Fermi
`http://static.trustedreviews.com/`

[12] *AMD VLIW4 core* - Architekruta Cayman
`http://media.bestofmicro.com/`

A Přílohy

Listing 1: Generovaní souborů pro porovnávání přesnosti FFT

```
for sizes = 6:26;
for versions = 1:5;
primeSizes = [61 127 257 509 1021 2039 4093 8191 16381
   32769 65537 131071 262147 524287 1048573 2097143
   4194301 8388593 16777213];
primes = 1;
if primes == 1
        SIZE = sizes(j-5);
else
        SIZE = (2^sizes);
end
input = rand(1, SIZE) + 0i;
out = fft(single(input));
output = out;
name = strcat('out_prime', num2str(mult), '_', num2str(
   version), '.txt');

fprintf('Writing to file: %s\n', name);

if exist(name, 'file') == 2
    delete(name);
end

fid = fopen(name, 'a');
fwrite(fid, SIZE, 'int', 0, 'native');
fprintf('Size written\n');
fwrite(fid, real(input), 'single', 0, 'native');
fprintf('Real Input written\n');
fwrite(fid, imag(input), 'single', 0, 'native');
fprintf('Imag Input written\n');
fwrite(fid, real(output), 'single', 0, 'native');
fprintf('Real Output written\n');
fwrite(fid, imag(output), 'single', 0, 'native');
fclose(fid);
end
end
```

Listing 2: Příklad použití knihovny.

```
#include "oclFFT.h"

#define SIZE 1024

int main(void)
{
        void *instance; // Promenna pro pointer na
           instanci
        instance = _oclCreateInstance(); // Zavolani
           funkce na vytvoreni instance
        int err = _oclInit(instance, 0, 0, true); //
           Zvolani funkce inicializace OpenCL s
           defaultnim zarizenim a oznacenim zarizeni za
           mobilni

        float *r_in = new float[SIZE]; // Alokace
           vstupniho pole realnych hodnot
        float *i_in = new float[SIZE]; // Alokace
           vstupniho pole imaginarnich hodnot
        float *r_out = new float[SIZE]; // Alokace
           vstupniho pole realnych hodnot
        float *i_out = new float[SIZE]; // Alokace
           vstupniho pole imaginarnich hodnot

        for(int i = 0; i < SIZE; i++)
        {
                // Prostor pro naplneni dat do
                   alokovanych poli
        }

        OCLArray clIn, clOut;
        // Funkce pro vytvoreni vstupniho a vystupniho
           pole na graficke karte
        clIn = _oclCreateBuffer(instance, SIZE,
           FFTL_TYPE, WRTYPE_READ_WRITE, &err);
        clOut = _oclCreateBuffer(instance, SIZE,
           FFTL_TYPE, WRTYPE_READ_WRITE, &err);

        err = _oclPackToBuffer(instance, clIn, SIZE,
```

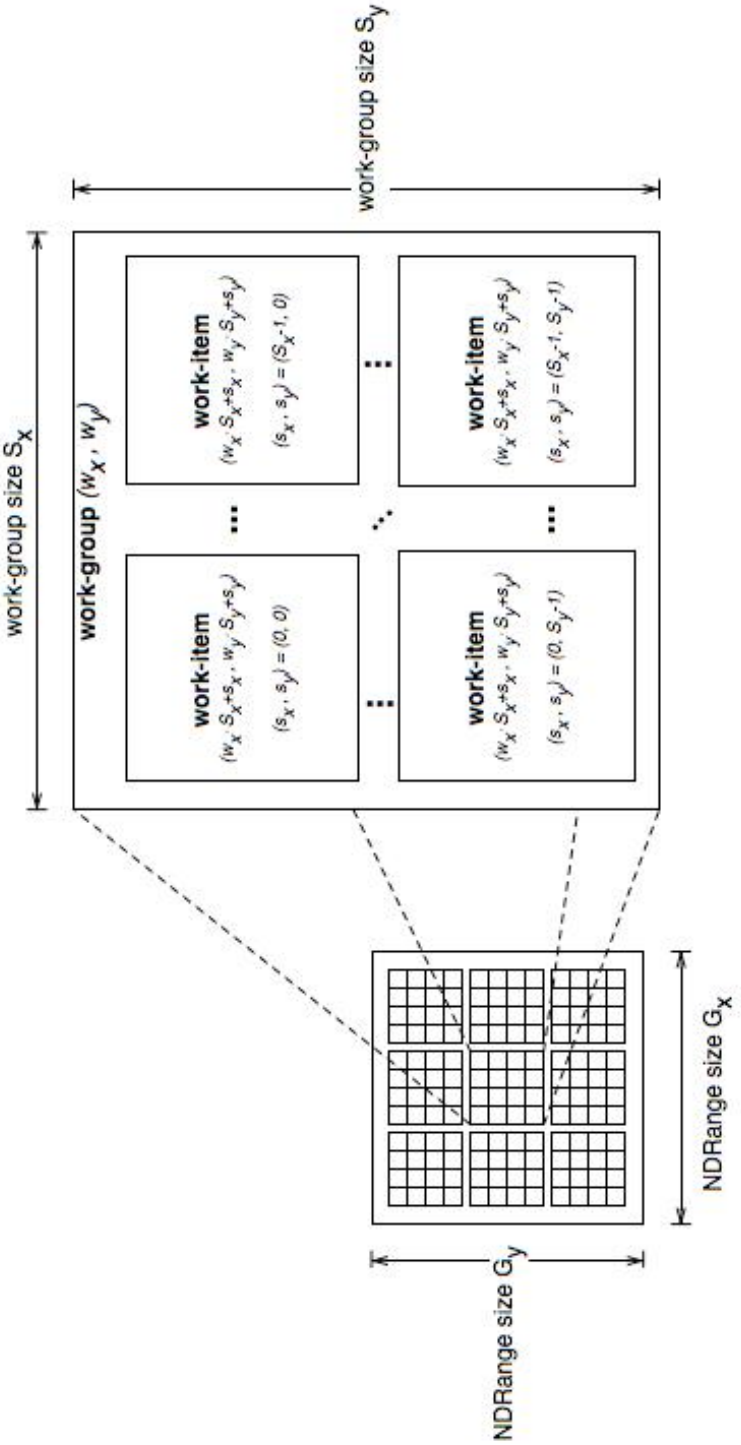
```
        r_in, i_in); // Funkce, ktera nakopiruje data
        do poli na graficke karte
    printf("Error: %d\n", err);
    err = _oclTransform(instance, clIn, clOut, SIZE,
        true); // Spusteni transformace. Blokovaci
        funkce nejsou potrebne
    printf("Error: %d\n", err);
    err = _oclUnpackFromBuffer(instance, clOut, SIZE
        , r_out, i_out); // Funkce, ktera zkopiruje
        data z poli na graficke karte do prislusnych
        poli v RAM
    printf("Error: %d\n", err);
    err = _oclBlock(instance); // Funkce, ktera
        zablokuje vykonavani na procesoru predtim, nez
        budou data pouzita. Pouziti je nutne, aby
        vysledky byly kompletni
    printf("Error: %d\n", err);


    // Vraceni alokovane pameti
    err = _oclDestroyBuffer(instance, &clIn);
    err = _oclDestroyBuffer(instance, &clOut);

    // Odpojeni knihovny od OpenCL
    err = _oclDeinitialize(instance);

    // Vraceni pameti alokovane samotnou knihovnou
    _oclDeinitializeInstance(instance);

    delete [] r_in;
    delete [] r_out;
    delete [] i_in;
    delete [] i_out;
    return 0;
}
```



Obrázek 6: Příklad rozdělení work-itemů do skupin v celém rozsahu G [2]