Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Paralelizace výpočtu šíření koherentního světla

Plzeň 2014 Petr Podávka

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 5. května 2014

Petr Podávka

Poděkování

Rád bych poděkoval Ing. Petru Lobazovi za cenné rady, věcné připomínky a vstřícnost při konzultacích a vypracování bakalářské práce.

Abstract

The *rayleigh* library, that is being developed at the Department of Computer Science and Engineering, University of West Bohemia, provides a way of coherent light propagation calculating in a free space between two parallel planes. The calculation of the light propagation is done using convolution form of Rayleigh-Sommerfeld diffraction integral, where the computation of filtered convolution kernels is needed. Calculation of such kernels is time-consuming matter and this bachelor thesis presents a way of decreasing the time needed for the computatiton by parallelization of the algorithm used by the *rayleigh* library. This thesis describes methods of paralellization available for CPU and GPU and presents two different implementations using Pthread library and OpenCL standard.

Obsah

1 Úvod

Pro mnoho vědních oborů, jako třeba počítačem generovaná holografie nebo fourierovská optika, je velmi důležitá schopnost modelování propagace koherentního světla volným prostorem, speciálně mezi dvěma rovnoběžnými rovinami.

Pro tuto úlohu se počítá se dvěma plochami, *source* v rovině $z = 0$, a *target* v rovině $z = z_0$, kde *source* je osvětlena koherentním světlem, nejlépe rovinnou vlnou, a úkolem je spočítat rozložení světla v rovině *target*.

Tuto úlohu řeší knihovna *rayleigh* vyvíjená na Katedře informatiky a výpočetní techniky na Západočeské univerzitě v Plzni. Využívá k tomu aproximaci Rayleigh-Sommerfeldovým integrálem 1. druhu, jenž lze díky své podobě konvoluce převést na numerický výpočet za použití tří rychlých Fourierových transformací (z angl. Fast Fourier Transform – FFT). Při diskretizaci úlohy je však nutné uvažovat korektní vzorkování jak *source*, tak vzorkování vlastního osvětlení a samotného Rayleigh-Sommerfeldova konvolučního jádra. To vede k velmi vysokému vzorkování, většinou kratšímu než je vlnová délka světla, a tedy k velkým paměťovým a časovým nárokům.

Tato práce má za úkol snížit časovou náročnost těchto výpočtů paralelizací algoritmu počítajícího jádro R-S integrálu. Práce je rozdělena na dvě části. V teoretické části bude vysvětlen princip propagace světla R-S integrálem, bude zde popsán způsob propagace světla metodou filtrované konvoluce a také rozebrány algoritmy, které tuto propagaci počítají, respektive algoritmy počítající část této propagace, konkrétně jádro konvoluce h. Zároveň zde budou diskutovány možnosti paralelizace uvedených algoritmů a dostupné prostředky pro implementaci takto paralelizovaných algoritmů. V praktické části pak budou popsány jednotlivé implementace s ukázkami kódu specifickými pro vybraný způsob paralelizace. V závěru praktické části budou srovnány časy běhu představených implementací s jejich sekvenční verzí při výpočtu jader různých velikostí. Výsledné implementace budou schopné běhu na CPU a grafické kartě, tedy GPU.

2 Teoretická část

2.1 Rayleigh-Sommerfeldův integrál

V celé práci se počítá s použitím pouze monochromatického světla. Dokonale monochromatické světlo je zároveň koherentní. V reálném světě neexistuje zdroj, který by byl schopný emitovat dokonale monochromatické světlo. Naštěstí ale není potřeba úplně dokonalého zdroje a téměř vždy stačí jenom téměř dokonalé, jakým je například *laser*, jehož světlo je dostatečně koherentní k provádění optických experimentů. Koherentní světlo pak lze definovat v libovolném bodě v prostoru jako

$$u(P,t) = A(P)\cos[\omega t - \phi(P)], \tag{2.1}$$

kde $A(P)$ je amplituda, ω je úhlová frekvence a $\phi(P)$ je fáze v bodě P v prostoru (x, y, z). Veličina $u(P,t)$ pak představuje idealizovanou skalární veličinu popisující elektromagnetické pole v bodu P a času t. Rovnici (2.1) můžeme přepsat na ekvivalentní tvar

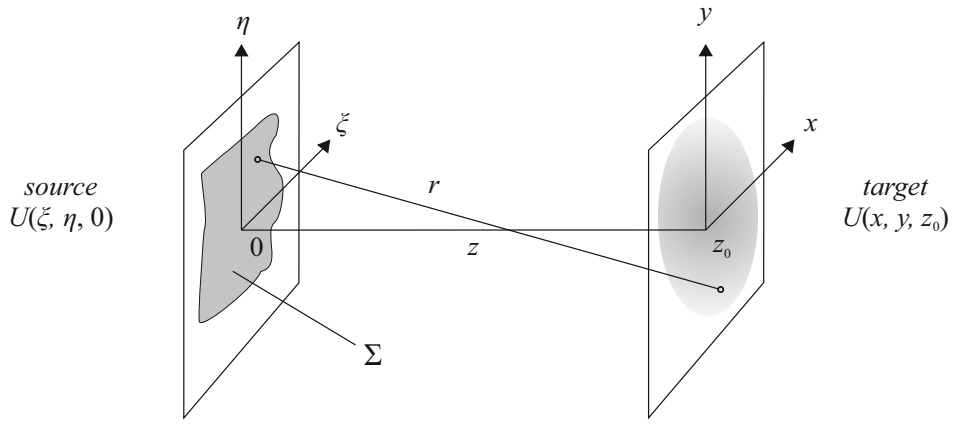
$$u(P,t) = \mathrm{Re}\{A(P)\exp[j\phi(P)]\exp[-j\omega t]\}.$$

Pro lepší manipulaci lze tato rovnice zapsat v plné komplexní podobě [1]. Díky tomu se může v zápisu opomenout časově proměnný člen, bude totiž v rámci experimentování s monochromatickým světlem vždy implicitně předpokládán. Z veličiny $u(P,t)$ tak vzniká tzv. fázor $U(P)$, který jediným komplexním číslem (komplexní amplitudou) definuje elektromagnetické pole v bodu (x, y, z)

$$U(x,y,z) = A(x,y,z)\exp[j\phi(x,y,z)].$$

Ze 2D plochy *source* se šíří koherentní světlo na plochu *target*. Na ploše *source* o rozsahu Σ je pole popsané jako $U(\xi,\eta,0)$. Plocha *source* emituje světlo na plochu *target*. Ta je rovnoběžná s plochou *target* a je popsána jako $U(x,y,z_0)$. Plochu *target* pak lze vyjádřit *Rayleigh-Sommerfeldovým integrálem* [6][5]

$$U(x,y,z_0) = \frac{-1}{2\pi}\iint_{\Sigma} U(\xi,\eta,0)\frac{\partial}{\partial z}\frac{\exp(jkr)}{r}d\xi d\eta. \tag{2.2}$$



Obrázek 2.1: Znázornění propagace ze *source* na *target*.

V této rovnici je $U(x, y, z_0)$ výsledná komplexní amplituda v libovolném bodu *target*, $U(\xi, \eta, 0)$ je komplexní transmitance *source*, Σ je rozsah *source*, $k = \frac{2\pi}{\lambda}$ je vlnové číslo. Proměnná r je vzdálenost mezi body $[x, y, z_0]$ a $[\xi, \eta, 0]$ a protože jsou roviny *source* a *target* rovnoběžné, je r dáno jako

$$r = \sqrt{(x-\xi)^2 + (y-\eta)^2 + z_0^2}. \tag{2.3}$$

Protože jsou plochy *source* a *target* na rovnoběžných rovinách, může se integrál z rovnice (2.2) zapsat jako konvoluce [6]

$$U(x, y, z_0) = \iint U(\xi, \eta, 0) h(x-\xi, y-\eta, z_0) d\xi d\eta. \tag{2.4}$$

Konvoluční jádro (*kernel*) je rovno

$$h(x, y, z) = \frac{-z}{2\pi}\left(jk - \frac{1}{r}\right)\frac{\exp(jkr)}{r^2},$$

kde

$$r = \sqrt{z^2 + x^2 + y^2}.$$

Konvolučním teorémem se pak formuluje rovnice (2.4) jako

$$U(x, y, z_0) = \mathcal{F}^{-1}\{\mathcal{F}[U(\xi, \eta, 0)]\mathcal{F}[h(x-\xi, y-\eta, z_0)]\}, \tag{2.5}$$

kde $\mathcal{F}$ a $\mathcal{F}^{-1}$ značí Fourierovu a inverzní Fourierovu transformaci.

Tato práce předpokládá řešení Rayleigh-Sommerfeldovým integrálem ze referenční. Je však vhodné uvést, že to není jediné možné řešení daného problému. Pole na ploše *target* lze spočítat také pomocí Fresnelovy aproximace nebo Fraunhoferovy aproximace [6]. Tyto aproximace zakládají na skutečnosti, že vzdálenost mezi plochami *source* a *target* je mnohonásobně větší než velikost polí na těchto plochách počítaných. Pak lze vzdálenosti jednotlivých bodů (rovnice (2.3)) různě aproximovat a zjednodušit tím výpočet. Pro tyto aproximace pak existují jistá omezení, kdy se například počítá jenom s kulovou vlnoplochou, vzdálenost z musí být v určitém poměru vůči velikosti polí a vlnové délce světla λ, atd. Lze pak použít různá odvozená kritéria pro odhadnutí, zda bude aproximace vhodná či nevhodná.

2.2 Diskretizace

Analytický výpočet Rayleigh-Sommerfeldova integrálu je pro obecnou podobu $U(source)$ nemožný. S nástupem dostatečně výkonné výpočetní techniky však lze vztah (2.2) diskretizovat a k jeho řešení použít rychlé Fourierovy transformace (Fast Fourier Transform, zkratkou FFT), viz rovnice (2.5). Postup při diskretizaci je následující. Nejprve se nahradí dvojný integrál dvojnou sumou. Dále se diskretizuje pole *source* vzorkováním funkce $U(source)$ při splnění Nyquistova limitu, který požaduje vzorkovací frekvenci ostře větší než je dvojnásobek maximální frekvence obsažené v obraze. Dále se můžou počítat komplexní amplitudy propagovaného světla ve vzdálenosti z v oblasti *target*. Je-li vzdálenost mezi vzorky v *source* i *target* stejná, a to Δ, platí [5]

$$target[x, y] = U(x\Delta, y\Delta, z_0) = (source[,] \otimes h_1[,])[x, y, z],$$

kde $\otimes$ představuje diskrétní konvoluci a pole $h_1[,]$ je Rayleigh-Sommerfeldovo jádro konvoluce definované obecně parametrem *ups* jako

$$h_{ups}[x, y] = h(x\Delta/ups, y\Delta/ups). \tag{2.6}$$

V mnoha situacích (typicky pro malé z) se ukazuje, že vzorkování pole $h[,]$ je příliš hrubé. Pro korektní výpočet je tedy nutné navzorkovat funkci $h()$ *ups*-krát jemněji. Pro aplikaci konvoluce je však potřeba, aby pole $source[,]$ bylo rovněž *ups*-krát jemněji vzorkované. Nové vzorky pole $source[,]$ se tedy musí odhadnout interpolací. To je uděláno následujícím způsobem: mezi každé dva prvky pole *source* se vloží $ups - 1$ nulových vzorků, čímž se získá $source_{ups}$.

Ten se pak konvoluje jádrem $\mathit{filter}[,]$, čímž se získá výsledná podoba pole *source*, tedy $\mathit{source}^{\mathit{fin}}_{\mathit{ups}}[,] = \mathit{source}_{\mathit{ups}}[,] \otimes \mathit{filter}[,]$. Konvoluční jádro $\mathit{filter}[,]$ se pak volí s ohledem na funkci $U(\mathit{source})$. Implementace a výběr vhodného jádra však není předmětem této práce a v dalších úvahách bude považováno za předem dané a ideální. Jeho délka bude lichá a dána jako $2\mathit{fwh}+1$. Může se pak využít asociativnosti konvoluce a řekne se, že [5]

$$\begin{aligned}\mathit{target}_{\mathit{ups}}[,] &= (\mathit{source}_{\mathit{ups}}[,] \otimes \mathit{filter}[,]) \otimes h_{\mathit{ups}}[,] = \\ &= \mathit{source}_{\mathit{ups}}[,] \otimes (\mathit{filter}[,]) \otimes h_{\mathit{ups}}[,]) = \\ &= \mathit{source}_{\mathit{ups}}[,] \otimes h^{\mathit{fin}}_{\mathit{ups}}[,].\end{aligned}$$

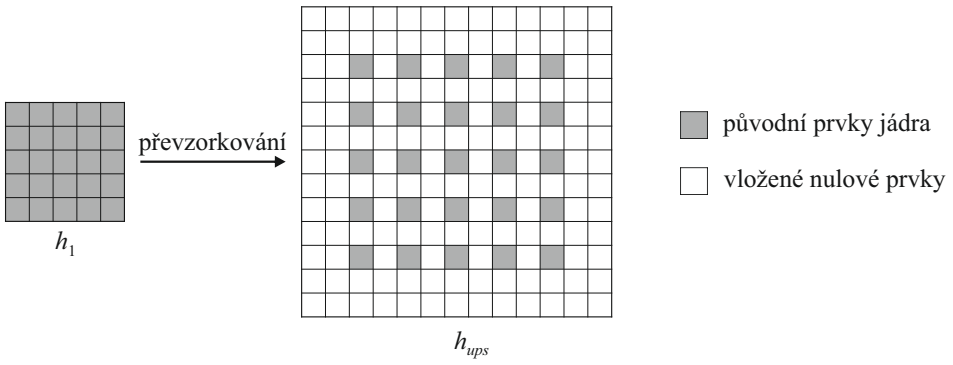
Výsledek je však nutné *ups*-krát podvzorkovat. Může se tedy psát [5]

$$\mathit{target}[,] = \mathit{source}[,] \otimes h_1^{\mathit{fin}}[,], \tag{2.7}$$

kde

$$h_1^{\mathit{fin}}[x,y] = h^{\mathit{fin}}_{\mathit{ups}}[\mathit{ups}\cdot x, \mathit{ups}\cdot y]. \tag{2.8}$$

Pro představu je na obrázku 2.2 jádro $h_1[,]$ o rozměrech 5×5 rozšířeno na $h_{\mathit{ups}}[,]$ s $\mathit{ups}=2$. Šedé buňky představují původní prvky jádra $h_1[,]$, bílé buňky jsou pak vložené prvky s nulovou hodnotou. Okraje o šířce dva jsou nutné z důvodu použití filtru $\mathit{fwh} = \mathit{ups} = 2$.



Obrázek 2.2: Převzorkování jádra h_1 o rozměrech 5×5 na h_{ups} při $\mathit{ups}=2$.

Protože právě paralelizaci výpočtu (2.8) řeší tato práce, bude v následujících odstavcích popsán základní algoritmus výpočtu $h_1^{\mathit{fin}}[,]$ a budou diskutovány možné optimalizace, které pak do daného algoritmu budou zakomponovány. Protože je tento algoritmus poměrně složitý a v této práci není možné podat zcela detailní vysvětlení, může se nezasvěcený čtenář nejprve seznámit s technickou zprávou [5].

Za základní lze považovat algoritmus 2.1. Tento algoritmus vypočítá převzorkované jádro $h_{ups}[,]$, které pak konvoluje s polem $filter[,]$. Princip algoritmu je přímočarý. Konstanty M, N jsou rozměry kernelu $h_1^{fin}[,]$, *filter* je filtr, který je aplikován na jádro $h_{ups}[,]$ po jeho výpočtu a fwh je polovina šířky filtru. Pole h_{ups} má rozměry $(ups(M-1)+2fwh+1)\times(ups(N-1)+2fwh+1)$ a představuje převzorkované jádro $h_{ups}[,]$. Pole h_{ups}^{fin} má stejné rozměry a představuje převzorkované filtrované jádro $h_{ups}^{fin}[,]$. Pole h_1^{fin} má pak rozměry $M \times N$ a představuje finální jádro $h_1^{fin}[,]$. Algoritmus vypočítá prvky pole h_{ups}, které následně konvoluje s polem *filter* a podvzorkuje. Protože v rovnici (2.8) nebyla uvažována konečnost všech polí, musí být rovnice ještě upravena, aby odpovídala způsobu rozšíření znázorněném na obrázku 2.2, tedy aby brala v úvahu kraje o šířce fwh. Upravená rovnice

$$h_1^{fin}[x,y] = h_{ups}^{fin}[ups \cdot x + fwh, ups \cdot y + fwh] \tag{2.9}$$

se pak použije pro podvzorkování kernelu $h_{ups}^{fin}[,]$.

Vstup: M, N, fwh, $filter[,]$
Výstup: h_1^{fin}
1 **for** *všechny prvky z* h_{ups} **do**
2 | vypočítej hodnotu prvku podle rovnice (2.6);
3 **end**
4 $h_{ups}^{fin} = h_{ups} \otimes filter$;
5 podvzorkování h_{ups}^{fin} na h_1^{fin} podle rovnice (2.9);

Algoritmus 2.1: Neoptimalizovaný výpočet propagačního jádra.

Přestože pro konvoluci podle rovnice (2.7) je potřeba pouze podvzorkované jádro $h_1^{fin}[,]$, v algoritmu 2.1 je v paměti drženo po celou dobu výpočtu celé pole h_{ups}. Algoritmus 2.1 totiž počítá cyklickou konvoluci polí h_{ups} a *filter* pomocí FFT a nebo aplikací definičního vztahu

$$\begin{aligned} h_{ups}^{fin}[x,y] &= buffer[x,y] \otimes filter[x,y] \\ &= \sum_{i=-fwh}^{fwh} \sum_{j=-fwh}^{fwh} buffer[x-i, y-j] \cdot filter[i,j]. \end{aligned}$$

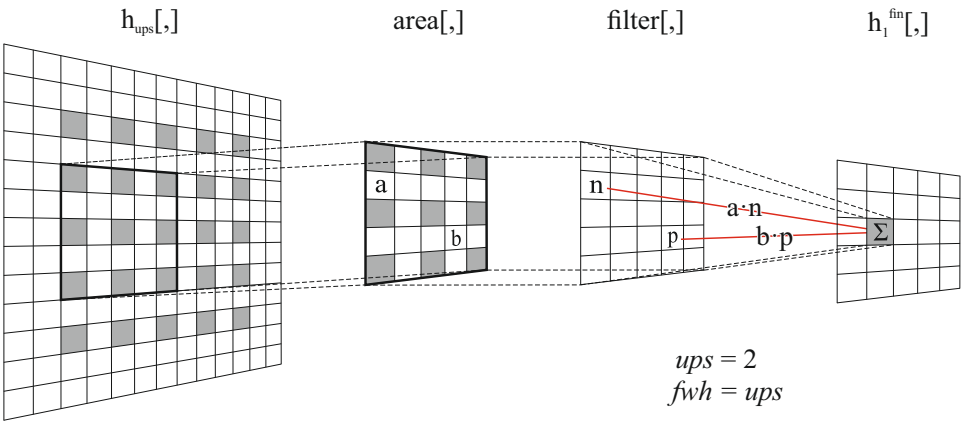
Při následném podvzorkování h_{ups}^{fin} na h_1^{fin} jsou zahazovány všechny prvky, které byly do jádra přidány jeho rozšířením. Není tedy potřeba je v konvoluci počítat. Tak lze ušetřit výpočetní čas a zároveň snížit paměťovou náročnost

algoritmu. Jádro h_{ups} se bude počítat po menších oblastech, které odpovídají prvkům jádra h_1^{fin}. Tyto oblasti se pak po prvcích vynásobí s polem $filter[,]$, provede se suma těchto násobků a výsledná čísla budou odpovídat prvkům z h_1^{fin}.

Každému prvku $h_1^{fin}[x, y]$ odpovídá oblast $area_{x,y}[,]$. Ta je shodná s částí kernelu $h_{ups}[\langle ups \cdot x; ups \cdot x + 2fwh\rangle, \langle ups \cdot y; ups \cdot y + 2fwh\rangle]$. Rozměry této oblasti jsou $(2fwh + 1) \times (2fwh + 1)$ a číselně lze vyjádřit jako

$$area_{x,y}[i, j] = h_{ups}[ups \cdot x + i, ups \cdot y + j], \quad i, j \in [0, 2fwh]. \tag{2.10}$$

Toto pole se pak po prvcích vynásobí s polem $filter[,]$, provede se suma všech výsledných prvků a tím se získá prvek $h_1^{fin}[x, y]$, viz obrázek 2.3.



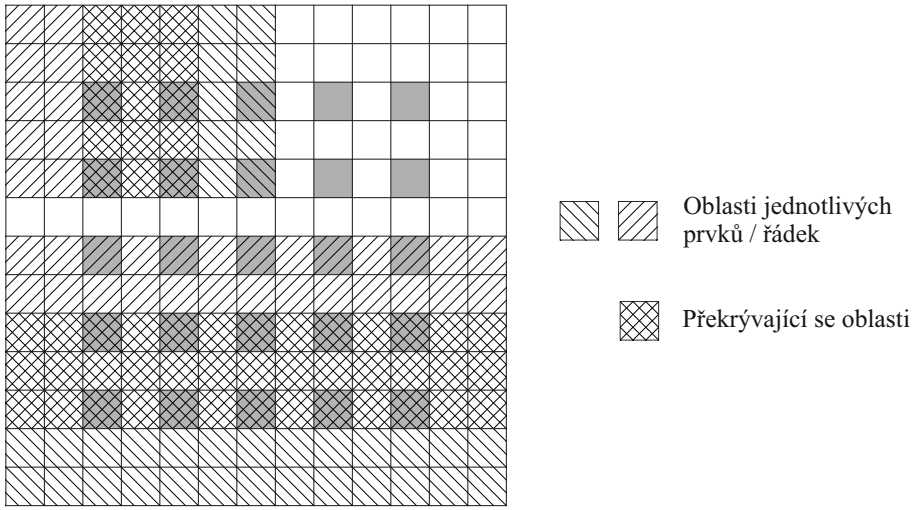
Obrázek 2.3: Aplikace *filtru* na část převzorkovaného jádra.

Uvedený postup lze pak vidět zapsaný v podobě pseudokódu jako algoritmus 2.2. Tento algoritmus je paměťově optimalizovaný, protože v paměti po dobu jeho běhu drží pouze malou pomocnou paměť o rozměrech $(2fwh + 1)^2$ a výsledný kernel h_1^{fin} o rozměrech $M \cdot N$. Naproti tomu algoritmus 2.1 v paměti držel celé převzorkované jádro $h_{ups}[,]$ o rozměrech $(ups(M-1)+2fwh+1) \times (ups(N-1)+2fwh+1)$. To je asi ups^2-krát méně, protože typicky je $fwh = a \cdot ups$, kde a je velmi malé (obvykle max. 3) a ups mnohem menší než jsou rozměry jádra h_1^{fin} (obvykle 1–10) [5].

Vstup: M, N, *fwh*, *filter*
Výstup: h_1^{fin}
for *všechny dvojice* (x, y) **do**
    vypočítej hodnotu prvků $area_{x,y}$ podle rovnice (2.10);
    vynásob pole $area_{x,y}$ po prvcích s *filter*;
    $h_1^{fin}[x, y] = \sum area_{x,y}$
end

Algoritmus 2.2: Paměťově optimalizovaný výpočet propagačního jádra.

Další optimalizací bude recyklace výpočtů z pole $area[,]$. Prvky rozšířeného jádra h_{ups} jsou od sebe vzdálené *ups*. Pole $area[,]$ má rozměry $(2fwh + 1) \times (2fwh + 1)$. Je tedy zřejmé, že části polí $area[,]$ vypočítané pro sousedící prvky se budou částečně překrývat a je nasnadě použít výsledky z pole $area_{x,y}[,]$ při výpočtu pole sousedícího prvku, $area_{x+1,y}[,]$. Stejný princip platí jak pro jednotlivá pole, tak i pro celé řádky nebo sloupce v pomyslném $h_{ups}[,]$, viz obrázek 2.4.



Obrázek 2.4: Překrývání oblastí jádra h_{ups}, $ups = 2$, s použitím filtru šířky $fwh = ups$.

Je tedy možné při výpočtu jádra $h_1^{fin}[,]$ recyklovat výpočty mezi sousedícími prvky v jednom řádku a také mezi řádky samotnými. Tato myšlenka v sobě skrývá ještě jednu optimalizaci a tou je filtrování jednotlivých částí $area[,]$ po řádcích a sloupcích zvlášť. Mějme dva vektory $filter_x[]$ a $filter_y[]$

o délce *fw* tak, že $\mathit{filter}[i,j] = \mathit{filter}_y[i] \cdot \mathit{filter}_x[j]$. Pak platí

$$\mathit{column}[j] = \sum_{i}^{fw} \mathit{area}_{x,y}[i,j] \cdot \mathit{filter}_x[i] \tag{2.11}$$

$$h_1^{\mathit{fin}}[x,y] = \sum_{i}^{fw} \mathit{column}[i] \cdot \mathit{filter}_y[i]. \tag{2.12}$$

Sloupečky *column*[ ] lze pak ukládat a místo recyklace jednotlivých prvků mezi řádky v h_{ups} se budou recyklovat části těchto vektorů.

Uvedení těchto principů do praxe znázorňuje algoritmus 2.3. Vstupem jsou konstanty M, N, rozměry kernelu $h_1[,]$. Dále konstanta *fwh*, polovina šířky filtru, a také data filtrů pro konvoluci po řádcích, filter_x, a sloupcích, filter_y. Proměnná *fw* značí šířku filtru, *upsampledLinePiece* je jeden řádek oblasti *area*, jedná se tedy o vektor s délkou *fw*. V proměnné *downsampledLine* jsou pak ukládány sloupečky, které vzniknou konvolucí vektorů *upsampledLinePiece* s filter_x. Pole *downsampledLine* má rozměry $M \times fw$.

Tento algoritmus je implementován v knihovně *rayleigh*, která je vyvíjena na Katedře informatiky a výpočetní techniky na Západočeské univerzitě v Plzni.

Vstup: M, N, fwh, $filter_y$, $filter_x$
Výstup: h_1^{fin}

```
1  fw = 2fwh + 1;
2  alokuj upsampledLinePiece[fw];
3  alokuj downsampledLine[M, fw];
4  for j ← 0 to N do
5  |  if j == 0 then
6  |  |  k = 0;
7  |  else
8  |  |  posuň řádky downsampledLine;
9  |  |  k = fwh;
10 |  end
11 |  for k to fw do
12 |  |  for i ← 0 to M do
13 |  |  |  if i == 0 then
14 |  |  |  |  l = 0;
15 |  |  |  else
16 |  |  |  |  posuň prvky v upsampledLinePiece;
17 |  |  |  |  l = fwh;
18 |  |  |  end
19 |  |  |  for l to fw do
20 |  |  |  |  vypočítej upsampledLinePiece[l];
21 |  |  |  end
22 |  |  |  downsampledLine[i, k] = upsampledLinePiece · filter_x;
23 |  |  end
24 |  end
25 |  for i ← 0 to M do
26 |  |  h_1^fin[i, j] = downsampledLine[i, ] · filter_y
27 |  end
28 end
```

Algoritmus 2.3: Paměťově a výpočetně optimalizovaný výpočet propagačního jádra.

2.3 Paralelizace na CPU

V této části bude diskutován úvod do problematiky paralelismu na CPU, vhodnost využití paralelismu pro uvedený problém, algoritmus 2.3 bude převeden do paralelní podoby a budou zde popsány rozdíly mezi jednotlivými implementacemi, jaké jsou jejich pro a proti.

2.3.1 Procesy a vlákna

Mluví-li se o paralelním běhu programu na CPU, je na výběr mezi implementací pomocí procesů nebo vláken. Použití procesů je výhodné ve chvíli, kdy záleží na robustnosti programu. Selhání jednoho procesu neovlivní běh ostatních. Nevýhodou je vyšší počet neúspěšných pokusů o zápis do mezipaměti a také, že sdílení dat mezi procesy vyžaduje explicitní řízení, které může být nákladné. Výhoda vláken je v nižší ceně sdílení dat mezi vlákny. Jedno vlákno může uložit data do paměti a ostatní vlákna pak mohou s těmito daty hned pracovat. Aplikace využívající vlákna také mohou mít nižší počet neúspěšných přístupů do mezipaměti. Nevýhodou je, že chyba jednoho vlákna nejspíš zapříčiní pád celého procesu [4].

V této práci jsou z uvedených důvodů použita vlákna. Protože bude každé vlákno počítat část kernelu h^{fin}, lze využít jednoduššího sdílení paměti. Pokud by měla nastat chyba v jednom vlákně, znamenalo by to, že implementace je chybná, a proto ani robustnost není v této situaci problém.

Při naivní představě o paralelním výpočtu si lze myslet, že takový výpočet musí být vždy rychlejší než výpočet sekvenční. Bohužel, není to úplně pravda. Paralelní výpočet vyžaduje přípravu prostředí, zavedení vláken do paměti a také jejich synchronizaci. To vše jsou jevy, které mohou v případě výpočtu jen malého množství dat způsobit, že paralelní verze algoritmu běží déle než jeho sekvenční protějšek. Zároveň to také znamená, že algoritmus běžící paralelně na N procesorech nebude nikdy N-krát rychlejší. Řešený problém však bude pracovat s maticemi s miliony prvků a většími. Navíc pro každý prvek této matice se provádí velké množství výpočtů. Lze tedy předpokládat, že v běžné praxi bude paralelní implementace vhodnější než implementace sekvenční, protože se bude pracovat s dostatečně velkými množinami dat. Je pak otázkou testování určit, pro které kombinace rozměrů matic a délky filtrů bude vhodné použít paralelní implementaci a pro které naopak implementaci sekvenční.

V této chvíli je vhodné uvést, že naše implementace bude součástí knihovny *rayleigh* vyvíjené panem Ing. Petrem Lobazem. Knihovna je implementována v jazyce C a odpovídá standardu C99.

Existují knihovny pro vysokoúrovňové paralelní programování, jako je třeba OpenMP. Ty pak umožňují vývojáři přenechat mechanismus řízení vláken a sdílení dat běhovým knihovnám kompilátoru. Z důvodu optimálního běhu je však žádoucí mít plnou kontrolu nad všemi mechanismy souvisejícími s paralelním během aplikace. Jazyk C poskytuje standardní implementaci vláken, avšak až od standardu C11. Pro zachování kompatibility s knihovnou *rayleigh* je však nutné použít standard C99 a musí být tedy použita jedna z externích knihoven, případně knihovna Win32 systému Windows. Druhá jmenovaná volba je zcela akceptovatelná, protože právě teď není zapotřebí multiplatformní implementace. Není ale vyloučeno, že v budoucnu bude toto potřeba. Stroje založené na platformě UNIX využívají standardu POSIX, jenž určuje standardy programování pro přenositelné unixové aplikace. Většina unixových systémů a systémů Unixu podobných se drží klíčových rysů těchto standardů. Aplikace naprogramovaná s ohledem na tyto standardy tak bude přenositelná mezi implementacemi Unixu a Unixu podobnými operačními systémy, jako je Linux, FreeBSD nebo Mac OS X, který je rovněž založený na jádru podobnému Unixu. Operační systém Microsoft Windows neimplementuje standardy POSIX přímo, ačkoliv existuje několik řešení, která umožňují programům napsaným pomocí rozhraní POSIX běh na platformách Windows. Ze standardu POSIX je zajímavá především knihovna Pthreads, využívaná pro paralelní programování. Jeden z nejúplnějších portů této knihovny na platformu Windows poskytuje knihovna Pthreads-w32, která implementuje většinu funkcí standardu Pthreads a umožňuje tyto funkce využívat v systémech Windows [4].

2.3.2 Paralelizace algoritmu

Algoritmus 2.3 je vhodný pro datovou paralelizaci. V datově paralelní aplikaci provádí více vláken stejné operace na samostatných prvcích dat. Všechna tedy provádějí stejnou úlohu, ale nad jinými indexy pole [4]. Otázkou tedy je, s jakými indexy bude jaké vlákno pracovat. Jedno vlákno by mohlo počítat jeden index, skupinu indexů, každý N-tý index z pole h^{fin}. Stejně tak by mohl být index nahrazen řádkem nebo sloupcem.

Z důvodu paměťových a výpočetních optimalizací uvedených v algoritmu

2.3 jsou výpočty jednotlivých indexů pole h^{fin} mezi sebou závislé.

První závislost lze pozorovat při výpočtu jednoho indexu matice bufferu *downsampledLine*. Vlákno N počítá prvek $[x, y]$ z matice h^{fin} a vlákno $N+1$ počítá prvek $[x+1, y]$. Vlákno N tedy spočítá $upsampledLinePiece_N$ na indexech $[0]$ až $[fw-1]$, vlákno $N+1$ by počítalo stejný rozsah ve svém bufferu $upsampledLinePiece_{N+1}$. Výsledky těchto dvou bufferů se však překrývají, proto je vhodné využít větší buffer $upsampledLinePiece_t[0]$ o rozměru $fwh * N + N + fwh$, který bude představovat sdílenou paměť, do které budou ukládat obě vlákna výsledky svých výpočtů a ty pak recyklovat mezi sebou. Problém nastává ve chvíli, kdy je tento buffer naplněný a obě vlákna by chtěla provést konvoluci po řádcích, tedy řádek 22 algoritmu 2.3. Před provedením této operace je totiž nutné všechna vlákna synchronizovat, aby se zajistila konzistence dat a předešlo se souběhu vláken (anglicky race condition). Stejný problém by nastal, pokud by bylo v každém vlákně počítáno více indexů nebo pokud by výpočet probíhal po sloupcích místo po řádcích. Nejjednodušší způsob, jak se vyhnout použití synchronizačních primitiv je, aby každé vlákno počítalo jeden nebo více řádků matice h_{ups}. Minimalizovat nutnost synchronizace je v zájmu věci, protože přehnané synchronizační nároky vedou ke špatným možnostem škálování paralelní aplikace [4].

Vlákno N tedy počítá řádek y matice h^{ups}, vlákno $N+1$ pak řádek $y+1$. Aby byl zužitkován výpočetní výkon, muselo by být spuštěno Q vláken tak, aby $fw \bmod Q = 0$. Pokud by tato podmínka nebyla splněna, znamenalo by to nutnost pozdržet běh některých vláken v době výpočtu konvoluce po sloupcích. Toto tvrzení lze pozorovat v následujícím příkladu. Šířka filtru je $fw = 5$ a v jednu chvíli mohou paralelně běžet čtyři vlákna. V jedné iteraci se spočítají čtyři řádky z pěti potřebných pro konvoluci. V další iteraci první vlákno spočítá poslední chybějící řádek a zbylá tři vlákna jsou buď bez práce, a nebo ukládají své výsledky do dalšího nezávislého bufferu. Takové řešení by pak přineslo ještě větší komplexnost algoritmu kvůli nutnosti dynamických posunů v bufferu pro recyklaci výsledků. Místo toho může každé vlákno počítat fw řádků, kdy každé vlákno bude mít svůj vlastní buffer $downsampledLine_N$ a po naplnění tohoto bufferu dané vlákno provede konvoluci po sloupcích a uloží výsledky do sdílené paměti, tedy h_1^{fin}. Ve výsledku vypočítá každé vlákno jeden celý řádek matice h_1^{fin}.

Nyní vyvstává otázka, které řádky matice h_1^{fin} bude počítat které vlákno. Pro vyhnutí se nutnosti synchronizace bude každé vlákno počítat souvislou skupinu řádek. Toto rozhodnutí lze ilustrovat na následujícím příkladu.Vlákno N počítá řádek y matice h_1^{fin}, a vlákno $N+1$ počítá řádek $y+1$ téže ma-

tice. Obě tato vlákna mají svůj vlastní buffer *downsampledLine* i přes to, že část výsledků v těchto polích je společná. Pro umožnění recyklace výsledků výpočtů vlákna N ve vláknu $N+1$ je nutné zavést buffer $downsampledLine_t$ o rozměrech $[fwhN + N + fwh, X]$ a před konvolucí po sloupcích synchronizovat vlákna, aby se zabránilo jejich souběhu. Tomu se lze vyhnout tak, že každé vlákno bude počítat souvislou část pole h_1^{fin}. Pokud je M vláken t_0 až t_{M-1}, bude vlákno N počítat řádky $(Y/M)N$ až $(Y/M)(N+1)-1$. Každé z těchto vláken pak bude mít svůj vlastní buffer *downsampledLine* a synchronizace pak nebude nutná. Pro úplnost je nutné uvést, že cenou za vyhnutí se synchronizaci je jistá redundantnost výpočtů, a to v hraničních řádcích. Tedy vlákno N bude počítat prvky bufferu *downsampledLine* do řádku $(Y/M)(N+1)-1$ včetně, vlákno $N+1$ pak svůj buffer pro řádky počínaje řádkem $(Y/M)(N+1)$ z matice h_1^{fin}. Tyto dva buffery budou sdílet část prvků, ale protože každé vlákno používá pro tato pole vlastní paměť, není možné tyto výsledky sdílet.

Tento kompromis je přitěžující pro konvoluční jádra ve tvaru obdélníku, jehož šířka je větší než výška. Je pak otázkou testování zjistit, při jakém škálování a rozměrech jádra je vhodné použít tuto implementaci a kdy naopak implementaci zahrnující synchronizaci společné paměti.

2.4 Paralelizace na GPU

Paralelizace programů na CPU je dnes již celkem rozšířená. Na druhou stranu, možnosti využití GPU k akceleraci výpočtů nejsou tak běžné v obecném povědomí. Začátek této kapitoly se proto bude věnovat krátkému úvodu do této problematiky.

2.4.1 OpenCL

Nejrozšířenější grafické akcelerátory jsou v dnešní době grafické karty společností AMD a NVIDIA. Obě tyto společnosti mají své uzavřené implementace a API, díky kterým může koncový uživatel psát obecné programy pro zpracování dat na těchto zařízeních. NVIDIA nazývá svoji technologii CUDA (Compute Unified Device Architecture), AMD naproti tomu AMD FireStream. Ani jedna z těchto technologií nelze použít na kartách druhého výrobce. Kvůli maximální univerzálnosti bude tedy použit standard OpenCL

(Open Computing Language), což je průmyslový standard pro paralelní programování heterogenních systémů, který podporuje nejenom drtivá většina moderních grafických karet, ale také procesorů [3].

Specifikace OpenCL se skládá ze čtyř částí nazývaných modely. Ty lze shrnout jako:

1. Model platformy: Specifikuje, že je pouze jeden procesor, který řídí vykonávání programu (host) a jeden nebo více procesorů schopných vykonávat OpenCL kód (zařízení). Definuje abstraktní rozhraní fyzických zařízení, které používá programátor pro psaní OpenCL programu (kernel), který se provádí na zařízení(ch).

2. Exekuční model: Definuje nastavení OpenCL prostředí na hostu a jak je kernel prováděn na zařízení. To zahrnuje vytvoření OpenCL kontextu na hostu, zprostředkování mechanismu komunikace mezi hostem a zařízením a definici modelu paralelního zpracování dat na zařízení.

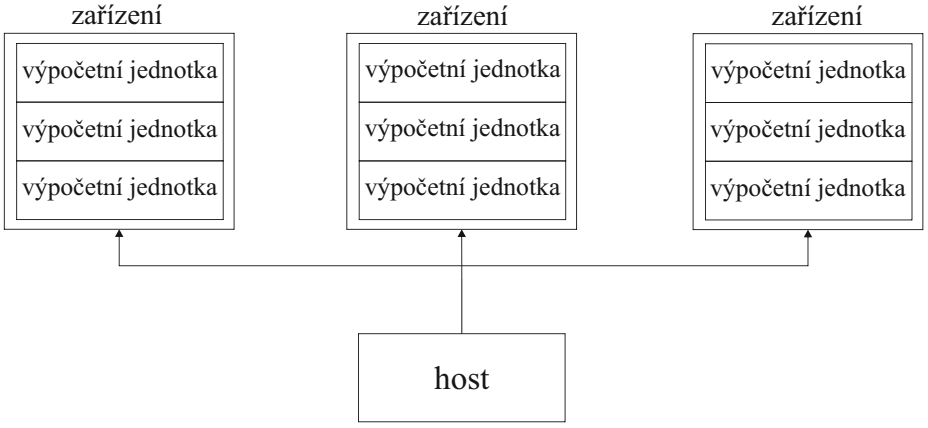
3. Paměťový model: Definuje abstraktní hierarchii paměti používanou kernelem, nezávislou na skutečné architektuře paměti konkrétních zařízení.

4. Programový model: Definuje, jak je model paralelního zpracování dat aplikován na vlastním hardwaru.

V typické situaci je OpenCL aplikace spuštěna na hostu s x86 CPU, který používá GPU zařízení jako akcelerátor. Model platformy definuje vztah mezi těmito dvěma zařízeními. Exekuční model zajistí, aby host vytvořil kernel, program, který bude provádět zařízení, a specifikoval způsob, jakým bude na zařízení paralelizován. Data v kernelu jsou programátorem alokována na specifická místa abstraktní hierarchie. Paměťový model pak tyto místa namapuje do paměti konkrétního zařízení. V závěru pak programový model vytvoří kontext, který bude provádět kernel a který se namapuje na vlastní zařízení.

V reálném světě existuje mnoho výrobců hardwaru a každý z těchto výrobců implementuje OpenCL na svých zařízeních různě. Tyto specifické implementace OpenCL API můžeme pokládat za různé platformy. Počet využitelných zařízení je pak limitován vybranou platformou. Pokud je vybrána platforma výrobce A, nemůže taková pracovat s GPU výrobce B. Model platformy také obsahuje abstraktní architekturu zařízení, kterou programátor

využívá při psaní OpenCL programu. Výrobce pak tuto abstraktní architekturu váže na konkrétní hardware. Model platformy definuje zařízení jako pole výpočetních jednotek, které na sobě fungují zcela nezávisle.



Obrázek 2.5: Relace mezi hostem a zařízeními.

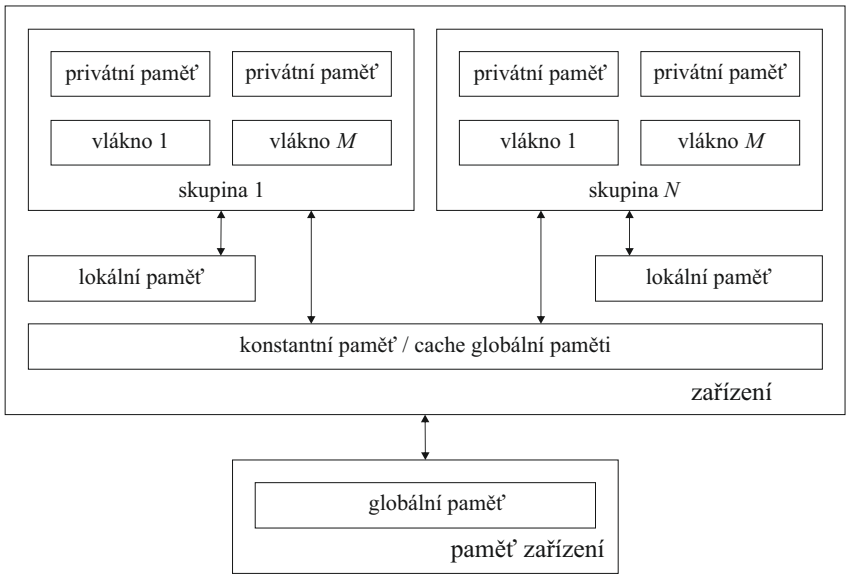
Aby mohl host spustit na zařízení kernel, musí se pro toto zařízení vytvořit kontext. Kontext je abstraktní objekt uchovávaný v paměti hosta, který představuje nástroj pro správu spojení hosta a zařízení na dané platformě. Kontextem je spravována paměť zařízení a programátor pomocí něj volá funkce OpenCL API. Veškerá komunikace mezi hostem a zařízením je uskutečněna vkládáním příkazů do příkazové fronty. Příkazová fronta je mechanismus správy požadavků hosta na zařízení. Každé zařízení, byť na stejné platformě, má vlastní příkazovou frontu.

OpenCL aplikace pracují především s velkými vícerozměrnými maticemi. Tato data musí být na zařízení fyzicky přítomna před zahájením výpočtu. Pro přenos dat do paměti zařízení je nutné je nejdříve zapouzdřit jako abstraktní paměťový objekt. OpenCL definuje dva typy paměťových objektů, *buffer* a *image*. Buffer je ekvivalent pole jazyka C. Image je abstraktní objekt, umožňující zarovnání dat v paměti a jiné optimalizace, které mohou zvýšit výkon.

Paralelní běh kernelu je v OpenCL členěn do skupin. V každé skupině (*work group*) je spuštěn určitý počet vláken (*work item*). Každé vlákno je identifikováno v rámci skupiny pomocí lokálního ID a v rámci celého výpočtu pomocí globálního ID. Způsob uspořádání OpenCL vláken umožňuje seskupit podmnožinu vláken z pracovní skupiny do jednoho kontextu vůči jednomu hardwarovému vláknu. Ve výsledku lze na to pohlížet jako na zobecnění *sin-*

gle instruction multiple data (SIMD), kde je jedna instrukce prováděna nad vektorem dat. V případě OpenCL si jednotlivé části vektoru udržují vlastní čítač až do bodu synchronizace. Na GPU může být vykonáváno až 64 vláken v jednom kroku jako jedno hardwarové vlákno na SIMD jednotce [3].

Paměť v rámci aplikace může být mapována na různé úrovně přístupu. Nejobecnější úroveň je globální paměť. V rámci zařízení se jedná o největší, ale zároveň také nejpomalejší paměť. Je tedy vhodné ji používat jako streamovací paměť. Používá se především pro uchování vstupních a výstupních dat, případně pro uchování mezivýsledků mezi jednotlivými kernely. Další úroveň je paměť lokální. Ta je sdílená mezi vlákny jedné skupiny, ostatní skupiny nemohou číst její obsah. Bývá výrazně menší a také rychlejší, obvykle se jedná o cache paměť čipu. Následuje privátní paměť, tedy paměť přístupná konkrétním vláknům. Poslední typ je konstantní paměť. Tato paměť je přístupná všem vláknům po celou dobu běhu programu, avšak pouze pro čtení. Je součástí globální paměti, ale díky omezení zápisu je možné ji umístit do části paměti, která je optimalizována nebo přímo určena pro časté čtení. Popsaná hierarchie úrovní paměti odpovídá vizualizaci na obrázku 2.6 [3].



Obrázek 2.6: Hierarchie paměťového modelu OpenCL.

Velikost globální paměti je u většiny zařízení počítána ve stovkách MB až jednotkách GB. Maximální velikost souvisle alokované paměti může být

menší než je kapacita zařízení. OpenCL programy je nutné optimalizovat, co se přístupů ke globální paměti týče. Pravidla pro tyto optimalizace bývají složitá a u různých zařízení se liší. Novější zařízení mohou mít speciální hardware pro zvýšení výkonu při přístupu více vláken k jedné oblasti globální paměti. Stejný účinek má umístění takových dat do konstantní paměti, která je pro tento způsob přístupu hardwarově optimalizována. Minimální velikost konstantní paměti je 64 KB [2][3].

Velikost privátní paměti není ve standardu OpenCL specifikovaná. Její konkrétní velikost nelze zjistit, ale při překročení kapacity je tato paměť přesunuta do cache paměti nebo rovnou do globální paměti, což způsobí významný pokles výkonu aplikace. Nelze tedy říct, kolik privátní paměti může program použít, jen že by mělo jít o objem co možná nejmenší. Lokální paměť je rychlá (obvykle SRAM) paměť, určená ke sdílení dat mezi vlákny v jedné pracovní skupině a k umožnění jednotného přístupu. OpenCL specifikace stanoví minimální velikost této paměti na 32 KB. Host může lokální paměť alokovat, ale nemůže do ní zapisovat ani z ní číst data. Pokud má program zpracovávat vstupní data poskytnutá hostem, může být lokální paměť použita pro snížení počtu přístupů ke globální paměti tak, že před zahájením výpočtu každé vlákno zkopíruje zpracovávaná data z globální paměti do lokální paměti, provede výpočet a po skončení výsledek zapíše zpět do globální paměti [2][3].

2.4.2 Paralelizace pro použití s OpenCL

Z použití GPU k akceleraci plynou jistá omezení, která lze přiblížit uvedením konkrétních čísel. V další kapitole se tedy budou všechny úvahy vztahovat k jednomu hardwaru, a to grafické kartě AMD HD5850.

Před zahájením výpočtu na GPU je nutné alokovat na zařízení paměť a také poslat na zařízení vstupní data. Pro optimální vytížení zařízení je nutné poskytnout dostatečné množství dat ke zpracování [3]. AMD HD5850 může alokovat až 536 MB souvislé paměti, což odpovídá čtvercovému poli komplexních čísel dvojité přesnosti o rozměrech 5790×5790. Velikost lokální skupiny bude 256, tedy v jednu chvíli může na zařízení běžet až 256 paralelních výpočtů. Algoritmus 2.3 představil několik optimalizací, které byly výhodné pro sekvenční výpočet. Tyto optimalizace bylo do jisté míry nutné omezit při paralelizaci na CPU a následující odstavce ukáží, že pro paralelizaci na GPU je nutné tyto optimalizace odstranit úplně.

Pokud by každé vlákno počítalo jeden prvek matice h^{fin}, musela by vlákna mezi sebou sdílet výsledky svých výpočtů. Při převzorkování $ups = fwh = 40$ by to znamenalo pomocnou paměť o rozměru $81 \cdot 40 \cdot 256 + 41$, tedy asi 13 MB. To vylučuje využití lokální paměti pro sdílení výpočtů, která má na AMD HD5850 kapacitu 32 kB. Je tedy nutné použít globální paměť, která je výrazně pomalejší. Dále je nutné použít synchronizaci této paměti před zahájením konvoluce po řádcích. Výsledky této konvoluce se pak musí uložit do další pomocné paměti o velikosti $81 \cdot 5790$, která musí být nutně uložena v globální paměti kvůli možnosti sdílení mezi různými lokálními skupinami. Nad touto pamětí musí opět proběhnout synchronizace před zahájením konvoluce po sloupcích. Všechny uvedené skutečnosti znamenají zpomalení výpočtu.

Těmto problémům se nelze vyhnout ani snížením granularity paralelismu. Pokud bude každé vlákno v lokální skupině počítat například jeden celý řádek matice h^{fin}, bude muset použít buffer o velikosti alespoň 1×81 pro recyklaci výpočtů v řádcích. Synchronizace mezi vlákny před konvolucí po řádcích je tedy odstraněna, avšak využití lokální paměti je opět vyloučeno, protože každé vlákno potřebuje tuto paměť o velikosti 81×256, tedy asi 332 kB. Požadavky na globální buffer pro sdílení výsledků před konvolucí po sloupcích zůstávají stejné.

Paměťová kapacita a počet paralelně běžících vláken se mohou na různých grafických kartách lišit, a proto není možné algoritmus optimalizovat na jeden specifický hardware. Řešením tohoto problému je odstranit veškeré optimalizace, které přinesl algoritmus 2.3, a implementovat základní algoritmus 2.2. Každé vlákno bude počítat jeden prvek matice h^{fin} bez sdílení společných výsledků z oblastí h_{ups}. K výpočtu tedy vlákno nepotřebuje žádné pole a všechny mezivýsledky bude ukládat do rychlé privátní paměti.

Algoritmus 2.3 představil způsob výpočtu prvků jádra h^{fin} tak, že pro výpočet krajních prvků a prvků, které odpovídají krajům pole *source*, je počítáno $(2fwh+1)^2$ prvků z jádra h_{ups}. Díky recyklování výpočtů však pro výpočet všech ostatních prvků jádra h^{fin} stačí počítat jenom fwh^2 prvků z h_{ups}, protože zbylé prvky byly spočítány v předchozích iteracích. Prvků, pro které se počítá jenom fwh^2 prvků z jádra h_{ups} je pak nepoměrně víc než těch, kde je nutné počítat přes celou šířku filtru. Naproti tomu v předchozích odstavcích uvedený algoritmus počítá pro získání jednoho prvku jádra h^{fin} vždy $(2fwh+1)^2$ prvků jádra h_{ups}. Tento algoritmus tedy musí spočítat asi 4-krát víc prvků matice h_{ups} než algoritmus 2.3. Diskuze o paměťové (ne)úspornosti je z důvodu použití odlišné platformy (jiná paměťová omezení, odlišný způsob práce s pamětí) bezpředmětná.

3 Realizační část

Úkolem této práce je paralelizace algoritmu popsaného v předešlých kapitolách, respektive paralelizace jeho sekvenční implementace. Ta je součástí knihovny vyvíjené panem Ing. Petrem Lobazem a byla dodána jako materiál pro zpracování. Výstupem práce jsou dvě paralelní implementace, jedna využívající knihovnu pthreads, druhá standard OpenCL. Obě tyto verze zakládají na implementaci sekvenční.

Zmíněná knihovna má schopnost výpočtu propagace velkých polí rozdělením propagace na menší části. V dalším textu se o této metodě vyskytnou zmínky. Protože detailní vysvětlení principu této techniky není předmětem této práce, budou zde uvedeny pouze základní skutečnosti důležité k pochopení kontextu v dalším textu. Pro propagaci velkých polí R-S integrálem nemusí stačit kapacita dostupného hardwaru. Proto lze rozdělit plochu *source* a *target* na několik menších ploch a každou plochu z roviny *source* propagovat zvlášť na každou plochu v rovině *target*. Rozměr jádra R-S integrálu bude vždy alespoň $(M + N - 1)$, kde M a N jsou rozměry ploch *source* a *target*.

3.1 Pthreads implementace

Implementace využívá port knihovny Pthreads, Pthreads-win32[1]. Předkompilované `.dll` a `.lib` knihovny včetně hlavičkových souborů je možné stáhnout z webových stránek projektu Pthreads-win32. Verze použitá v této implementaci je 2.9.1. Pro využití funkcí knihovny Pthreads je nutné do zdrojového kódu vložit hlavičkový soubor `pthread.h`. Při překladu se musí program linkovat s knihovnou `pthreadVC2.lib` a pro běh vlastního programu využívajícího knihovnu Pthreads musí být přístupná dynamická knihovna `pthreadVC2.dll`.

Aby byl výsledný program co možná nejvíce dynamický, zjišťuje se počet dostupných procesorových jader automaticky pomocí Windows API, které je přístupné po vložení hlavičkového souboru `windows.h`. Právě počet použitých vláken je jeden z parametrů funkce `kernelCalculateRSFT_threaded2()`, která zajišťuje paralelní výpočet jádra. Tato funkce je řídící funkce zajišťující alokaci prostředků, spuštění vláken a jejich úspěšné dokončení. Kód,

[1] `https://www.sourceware.org/pthreads-win32/`

který je spuštěn v samotných vláknech, je umístěn ve funkci `kernelCalculateRSFT_thread2()`.

Sekvenční implementace využívá několik pomocných pamětí. Paralelní verze používá buffery stejné velikosti a konstrukce. Mimo pole `data`, které odpovídá poli h^{fin} a do kterého jsou ukládány konečné výsledky, není žádný z těchto bufferů sdílený mezi vlákny. Pro zjednodušení alokace paměti a řešení případných chyb je paměť pro tyto buffery alokována v řídící funkci a vláknům jsou předány ukazatele na tuto paměť.

```
upsampledLinePiece = (DOUBLE2**) malloc(thdc * sizeof(DOUBLE2*));
if (upsampledLinePiece == NULL)
        GOEXCEPTION;

for(i=0; i<thdc; i++) { /* thdc = number of threads */
        upsampledLinePiece[i] = (DOUBLE2*) malloc(filterWidth_x *
            sizeof(DOUBLE2));
        if (upsampledLinePiece[i] == NULL)
                GOEXCEPTION;
} /* upsampledLinePiece[thread_id][filter_x]; */
```

Díky tomuto přístupu může být veškerá paměť alokovaná pro běh funkce dealokována na jednom místě v řídící funkci. Pro uchování informací o spuštěných vláknech je dále vytvořeno pole datového typu `pthread_t`. Každému indexu odpovídá jedno vlákno.

```
threads = (pthread_t*) malloc(thdc * sizeof(pthread_t));
if (threads == NULL)
        GOEXCEPTION;
```

Každému vláknu je potřeba předat mnoho různých parametrů. Při vytváření vlákna funkcí `pthread_create()` je však dovoleno předat vláknu pouze jeden argument. Řešením je vytvořit strukturu, která bude obsahovat všechny potřebné parametry, a vytvářenému vláknu předat ukazatel na danou strukturu. Vlákna sdílí některé parametry, v některých se liší. Je potřeba alokovat tolik struktur, kolik je vláken, a každou naplnit daty zvlášť.

```
thread_args = (struct thread_args*) malloc(thdc * sizeof(struct
    thread_args));
if (thread_args == NULL)
        GOEXCEPTION;
```

Mnohé z parametrů předávaných vláknům jsou společné pro všechna vlákna. Jedná se například o šířku filtru, rozměry ploch *source* a *target*, rozměry počítaného jádra, atp. Mimo to jsou vláknům předávány parametry unikátní pro každé vlákno. Mezi tyto parametry patří index vlákna, ukazatel na paměť bufferů, počáteční řádek výpočtu a také konečný řádek výpočtu. Počáteční řádek je výsledkem jednoduchého vzorce $j = kernelSamples\_y/thdc * i$, kde $kernelSamples\_y$ označuje počet řádek výsledného jádra h^{fin}, $thdc$ je celkový počet vláken a i je ID konkrétního vlákna. Protože není jisté, že bude počet řádků jádra h^{fin} dělitelný beze zbytku počtem vláken, je získání indexu posledního řádku obtížnější.

```
if (i == thdc-1)
        j_end = kernelSamples_y;
else
        j_end = kernelSamples_y/thdc*(i+1);
```

Touto podmínkou je zajištěna spravedlivá distribuce výpočtu mezi všechna vlákna s tím, že poslední vlákno vypočítá řádky, které nelze rozdělit mezi ostatní, a kterých může být maximálně $thdc - 1$.

Pak nezbývá než všechna vlákna spustit funkcí `pthread_create()` a počkat na jejich skončení funkcí `pthread_join()`.

3.1.1 Běh vlákna

Každé vlákno počítá dané řádky stejným způsobem, jakým by počítala sekvenční implementace všechny řádky jádra h^{fin}. Vlákna nesdílí buffery mezi sebou, nepoužívají žádná synchronizační primitiva. Jediná sdílená paměť je pole `data`, do kterého jsou ukládány finální výsledky. Protože se do této paměti pouze zapisuje a protože každé vlákno zapisuje do jiné její části, není potřeba synchronizace ani této paměti.

Pseudokód implementace funkce `kernelCalculateRSFT_thread2()` se příliš neliší od algoritmu 2.3. V následujících odstavcích budou rozebrány a vysvětleny jednotlivé úseky tohoto algoritmu spolu s ukázkami konkrétní implementace.

V průběhu výpočtu vlákno ví z předaných argumentů, kolikátý řádek jádra h^{fin} je právě počítán. Ve dvou situacích může nastat, že musí vlákno počítat rozšířený řádek v celé výšce, tedy `filterWidth_y` řádků z h_{ups}. První

situace je, když probíhá výpočet první řádky v rámci vlákna. Druhá situace nastává ve chvíli, kdy má jádro h^{fin} více řádek než plocha *target*. Když je pak počítán řádek `j == targetSamples_y`, nemůžou být recyklovány výsledky z předchozích výpočtů a musí být proveden kompletní výpočet. Je tedy zavedena nová proměnná `row` značící, kolik řádek vlákno spočítalo.

Posun řádek je v bufferu `downsampledLine` proveden kopírováním ukazatelů na celé řádky mezi tímto bufferem a bufferem `downsampledLineBuffer`.

```
for (k=0; k<filterWidth_y; k++)
        downsampledLineBuffer[idt][k] = downsampledLine[idt][k];

for (k=0; k<filterWidth_y-filterShift_y; k++)
        downsampledLine[idt][k] =
            downsampledLineBuffer[idt][k+filterShift_y];

for (k=filterWidth_y-filterShift_y; k<filterWidth_y; k++)
        downsampledLine[idt][k] =
            downsampledLineBuffer[idt][k-filterWidth_y+filterShift_y];
```

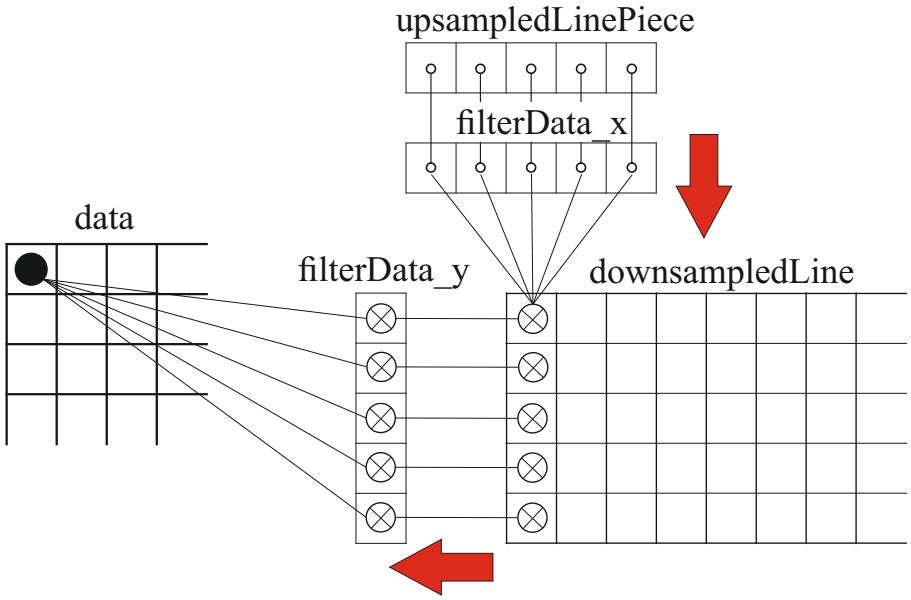
Každý řádek bufferu `downsampledLine` představuje jeden řádek rozšířeného jádra h_{ups} již konvoluvaného po řádcích. Pokud proběhl posun, znamená to, že byly recyklovány výsledky z výpočtu předchozího řádku. Stačí tedy počítat řádky `fwh` až `filterWidth_y`-1. V opačném případě musí být počítány všechny řádky tohoto bufferu.

Výpočet bufferu `downsampledLine` probíhá po řádcích, kde pro každý prvek v daném řádku je vypočten buffer `upsampledLinePiece`. Dva v řádku sousedící prvky bufferu `downsampledLine` budou sdílet část bufferu `upsampledLinePiece`. Proto je před samotným výpočtem proveden posun prvků v tomto bufferu a jsou počítány jenom chybějící prvky. Posun prvků má stejný princip jako posun řádek v bufferu `downsampledLine`.

```
for (k=filterShift_x; k<filterWidth_x; k++) {
        upsampledLinePiece[idt][k-filterShift_x][RE] =
            upsampledLinePiece[idt][k][RE];
        upsampledLinePiece[idt][k-filterShift_x][IM] =
            upsampledLinePiece[idt][k][IM];
}
```

Po získání kompletního bufferu `upsampledLinePiece` je toto pole po prvcích vynásobeno s polem `filterData_x` a součet těchto násobků je uložen

do `downsampledLine`. Následuje zmíněný posun prvků `upsampledLinePiece` a výpočet sousedního prvku. Tímto způsobem je naplněn celý `downsampledLine`, jehož sloupce jsou pak vynásobeny po prvcích s `filterData_y` (konvoluce po sloupcích) a suma těchto násobků je prvek pole h^{fin}. Vizualizace postupu při výpočtu a znázornění vztahů mezi buffery je na obrázku 3.1.



Obrázek 3.1: Postup výpočtu jednoho prvku jádra h^{fin}.

Pro uložení výsledného prvku jádra h^{fin} je potřeba znát odpovídající index v poli `data`. Ten je počítán při konvoluci po sloupcích. Před zahájením konvoluce se z právě počítané řádky `j` zjistí index prvního prvku odpovídající řádky v poli `data`, který se pak v průběhu cyklu inkrementuje.

```
index = j * kernelSamples_x;
/* go through columns in downsampledLine */
for (i=0; i<kernelSamples_x; i++, index++) {

        /* convolution by column */
        for (filter_y = 0; filter_y < filterWidth_y; filter_y++) {
                filteredKernelValue[RE] += filterData_y[filter_y] *
                    downsampledLine[filter_y][i][RE];
        }
        data[index][RE] = filteredKernelValue[RE];
}
```

3.2 OpenCL implementace

V případě implementace knihovnou Pthread byla pro výpočet jádra volána funkce `kernelCalculateRSFT_threaded2()`. Podobně se pro výpočet jádra pomocí OpenCL volá funkce `kernelCalculateRSFT_cl()`. Kvůli přehlednosti je však kód související s OpenCL umístěn ve zdrojovém souboru `r_cl.c`, konkrétně ve funkci `init_cl()`.

Úkolem funkce `kernelCalculateRSFT_cl()` je zpracování vstupních parametrů. Jedná se tedy například o zjištění velikosti počítaného jádra, vynulování paměti určené pro uchování výsledků, výpočet dodatečných parametrů, atp. Součástí této funkce je také provedení Fourierovy transformace nad vypočtenými daty.

Pro zahájení výpočtu vlastních dat se volá funkce `init_cl()`. Této funkci je předáváno několik parametrů, všechny související s konkrétním jádrem. Jedná se například o rozměry jádra, rozměry polí *source* a *target*, ukazatel na paměť pro uložení výsledků, data filtrů, atp.

Protože kernel počítající jádro neimplementuje žádné optimalizace, není potřeba uchovávat filtry zvlášť jako dva vektory. Dokonce je výhodné vytvořit jeden obdélníkový filtr. K tomuto je alokováno pole požadovaného rozměru a naplněno podle rovnice $filter[i,j] = filter_y[i] \cdot filter_x[j]$.

```
filterDataMatrix = (float*) malloc(sizeof(float) * filterWidth_y *
    filterWidth_x);

for(j = 0; j < filterWidth_y; j++) {
       for(i = 0; i < filterWidth_x; i++) {
              filterDataMatrix[j * filterWidth_x + i] = (float)
                  (filterData_y[j] * filterData_x[i]);
       }
}
```

Na úspěšném volání všech OpenCL API funkcí závisí správné vykonání výpočtu. Součástí některých ukázek kódu bude tedy i kontrola úspěšnosti volání API funkcí. Obecně jsou výjimky ošetřeny uvnitř podmínky, která kontroluje úspěšnost předchozího volání. Ošetření výjimky spočívá ve vytisknutí chybového výpisu a ve skoku na návěští, kde se dealokuje paměť, uvolní OpenCL objekty a volající funkci se vrátí chybová hodnota.

Vlastní inicializace spočívá v připojení k platformě, nalezení zařízení odpovídajícího typu, vytvoření kontextu a příkazové fronty. Funkce pro připojení k platformě a funkce pro vytvoření příkazové fronty jsou proti výjimkám ošetřeny odlišně. První jmenovaná vrací návratovou hodnotu signalizující úspěch nebo neúspěch volání. Druhá jmenovaná vrací ukazatel na strukturu, který v případě chyby bude `NULL`.

```
err = clGetPlatformIDs(1, &cpPlatform, NULL);
if (err != CL_SUCCESS) {
        ELOG0("Error: Failed to find a platform!");
        GOEXCEPTION;
}

commands = clCreateCommandQueue(context, device_id, 0, &err);
if (!commands) {
        ELOG0("Error: Failed to create a command commands!");
        GOEXCEPTION;
}
```

Zdrojový kód OpenCL C je součástí zdrojového kódu programu v podobě konstanty `const char *KernelSource`.

```
const char *KernelSource =                    "\n" \
         "__kernel void double_precision(      \n" \
         "    __global  float2* output,        \n" \
         "    __constant float* filterData,    \n" \
         "    int targetSamples_y,             \n" \
         ...
         "\n";
```

Součástí `r_cl.c` je funkce `load_kernel()`, která načítá kernel z textového souboru z dané plné cesty. Tato funkce je určena pro budoucí rozšíření a byla použita k účelům testování různých kernelů. Ve stávající implementaci není volána.

Stejně jako obyčejný zdrojový kód v jazyce C je nutné přeložit do spustitelného programu, musí být ze zdrojového kódu OpenCL C vytvořen *program* a ten zkompilován. OpenCL program je překládán za běhu skrze sérii API volání. To umožňuje systému zahrnout optimalizace pro konkrétní zařízení [3]. *Program* je ze zdrojového kódu vytvořen voláním funkce `clCreateProgramWithSource()`. Následuje kompilace voláním funkce `clBuildProgram()`. Ošetření výjimky při volání této funkce je složitější než ošetření uváděná

dříve. Mimo výpisu chyby totiž zahrnuje také vypsání logu kompilátoru.

Zjištění velikosti lokální skupiny je provedeno voláním funkce `clGetKernelWorkGroupInfo()`. Každé vlákno počítá jeden prvek jádra h^{fin}. Celkový počet počítaných prvků `global = kernelSamples_x * kernelSamples_y` pak musí být beze zbytku dělitelný velikostí lokální skupiny. Není však zaručeno, že počítané jádro bude mít odpovídající počet prvků. Případné chybě se předchází zvětšením proměnné `global` na nejbližší vyšší násobek velikosti lokální skupiny. Výkon aplikace tím není nijak ovlivněn a výpočet probíhá s maximální možnou velikostí lokální skupiny.

```
clGetKernelWorkGroupInfo(kernel, device_id,
    CL_KERNEL_WORK_GROUP_SIZE, sizeof(local), &local, NULL);

if ((kernelSamples_x * kernelSamples_y) % local == 0)
        global = kernelSamples_x * kernelSamples_y;
else
        global = ((kernelSamples_x * kernelSamples_y) / local + 1)
            * local;
```

Funkce `clCreateBuffer()` vytváří objekty bufferů. Ty představují alokované místo v paměti zařízení, na které lze poslat data příkazem `clEnqueueWriteBuffer()` a číst z něj příkazem `clEnqueueReadBuffer()`. První jmenovaný je použit pro zápis dat filtru, druhý zase pro čtení výsledných dat. Funkce `clEnqueueWriteBuffer()` je spjatá s kontextem, ne s konkrétním zařízením. Transfer dat tedy nemusí proběhnou okamžitě při volání funkce [3].

Parametry kernelu musí být nastaveny každý zvlášť funkcí `clSetKernelArg()`. Požadavek na vykonání kernelu je zařazen do příkazové fronty voláním funkce `clEnqueueNDRangeKernel()`. Volání funkce `clFinish()` je pak blokující, dokud nejsou ukončeny všechny příkazy dříve vložené do příkazové fronty. Skončení blokace signalizuje ukončení kernelu na zařízení. Lze tedy načíst výsledek výpočtu funkcí `clEnqueueReadBuffer()`.

Z důvodů paměťové úspory jsou finální hodnoty prvků jádra h^{fin} na zařízení ukládána jako `float`. Host však používá pro uložení těchto hodnot datový typ `double`. Nelze použít běžné přetypování, protože se jedná o komplexní čísla, tedy vektory s dvěma prvky. Přesun výsledků ze zařízení na hosta musí být tedy řešen přes pomocnou paměť `results`. Výsledky jsou přesunuty ze zařízení do této pomocné paměti funkcí `clEnqueueReadBuffer()` a poté ručně zkopírovány na správné místo.

```
for(i = 0; i < kernelSamples_y * kernelSamples_x; i++) {
        data[i][RE] = results[2 * i];
        data[i][IM] = results[2 * i + 1];
}
```

Mimo funkce `cl_init()` obsahuje soubor `r_cl.c` ještě dvě další funkce. Tyto funkce slouží k testovacím účelům a při normálním použití knihovny nejsou nikdy volány. Mohou však být využity v budoucím rozšíření nebo optimalizaci.

- `cl_info()` – Funkce zjistí všechna dostupná OpenCL zařízení na všech platformách a do `stdout` vypíše vybrané vlastnosti jednotlivých zařízení. Funkce nemá žádné vstupní parametry. V případě chyby je vráceno `EXIT_FAILURE`.
- `cl_test()` – Funkce vybere jedno GPU zařízení, které inicializuje a nechá jím spočítat jednoduchý kernel. Pole `data` o rozměrech 1024×1024 je na hostu naplněno náhodnými čísly a na zařízení je každý prvek umocněn na druhou. Výsledky jsou pak porovnány s mocninami vypočtenými na hostu a je vypsán podíl shodných prvků. V případě chyby je vráceno `EXIT_FAILURE`.

3.2.1 OpenCL kernel

Kernel je programová funkce, která je spouštěna na zařízení. Zdrojový kód kernelu psaný v jazyce OpenCL C je na první pohled podobný jazyku C. Kernel má vstupní parametry, lokální proměnné a standardní řídící strukturu. Jedna instance kernelu se v OpenCL terminologii označuje jako *work item*, v této kapitole dále označována jako *vlákno*. OpenCL vlákno (kernel) je snaha o vyjádření paralelismu stejným způsobem, jakým by mohl být vyjádřen Win32 nebo POSIX knihovnami [3].

Kernel musí začínat klíčovým slovem `__kernel` a jeho návratová hodnota musí být `void`. Dále, stejně jako v jazyce C, následuje název funkce a výčet vstupních parametrů.

```
__kernel void double_precision() {}
```

Vstupní parametry kernelu částečně odpovídají vstupním parametrům funkce `cl_init()`. Jedná se o rozměry jádra, ploch *source* a *target*, hodnotu převzor-

kování, vzorkovací vzdálenost, atp. Tyto parametry mají datový typ `double`, při výpočtu na grafice je však rozumné používat jednoduchou přesnost, protože je rychlejší a hardware GPU je pro počítání s jednoduchou přesností lépe přizpůsoben. Zavádí se tedy lokální proměnné typu `float` a probíhá přetypování. Součástí vstupních parametrů jsou také ukazatele `output` a `filterData`.

Do paměti `output` budou ukládány výsledky výpočtu. Protože musí být tato paměť přístupná všem vláknům, musí být typu `__global`. Datový typ `float2` označuje vektorový datový typ, kdy má každý prvek dvě složky. Tyto složky zde odpovídají reálné a imaginární složce komplexního čísla. Ukládání na index pak probíhá přetypováním vektoru.

```
output[gid] = (float2) (value_RE, value_IM);
```

V paměti `filterData` je uložen filtr, jenž se aplikuje na vypočtenou oblast jádra h_{ups}. Protože musí být filtr do této paměti zapsán hostem, nelze použít rychlou lokální paměť. Filtr je určen pouze pro čtení, bude tedy namísto toho použit typ `__constant`. Data pak budou uložena v části globální paměti, která je více optimalizovaná pro rychlý přístup. Velikost konstantní paměti se pohybuje v průměru kolem 64 KB. To je při použití datového typu `float` dostatečná paměť pro filtr o rozměru 128×128.

Protože každé vlákno počítá právě jeden prvek matice h^{fin}, stačí pro určení čísla řádku a sloupce počítaného prvku jenom globální ID vlákna zjištěné voláním funkce `get_global_id()`. Výpočet řádku a sloupce je pak otázkou jednoduché aritmetiky. Proměnná `j` značí řádek, `i` zase sloupec v matici h^{fin}.

```
j = gid / kernelSamples_x;
i = gid % kernelSamples_x;
```

Tyto údaje jsou nutné pro určení pozice počítaného prvku jádra vůči ploše *source* a výpočtu x-ové a y-ové složky vektoru r. Výpočet oblasti pak probíhá ve dvou vnořených cyklech `for`. V těch se vypočítá dodatečná úprava složek vektoru r, vypočítá se hodnota prvku rozšířeného pole a vynásobí se odpovídajícím prvkem filtru. Součet těchto filtrovaných hodnot je po skončení cyklů uložen jako výsledný prvek jádra do paměti `output`.

Je nutné uvést, že v celém kernelu je použita jednoduchá přesnost. Výjimku tvoří výpočet filtrovaného prvku oblasti jádra h_{ups}.

```
value_RE += temp * sin(temp_exponent) * filterValue;
```

Tento výpočet totiž zahrnuje násobení velkého čísla `temp` s funkcí sin() nebo cos() proměnné `temp_exponent`, kde je pro získání správného výsledku nezbytná co možná nejvyšší přesnost. Proměnné `temp` a `temp_exponent` jsou tedy jediné proměnné datového typu `double`.

3.3 Měření rychlosti

Měření a porovnání rychlosti jednotlivých implementací bylo provedeno odečítáním procesorových cyklů potřebných pro výpočet jádra. Procesorový cyklus je čas potřebný k provedení strojové instrukce. Tato metoda vyžaduje přítomnost hardwarové součásti, čítače cyklů, na testovacím zařízení. Tento čítač je však již běžnou součástí procesorů architektury x86. Změřené hodnoty jsou pouze orientační, protože odečtení cyklů nebere v potaz pouze cykly spotřebované programem pro výpočet jádra, nýbrž také cykly potřebné pro běh operačního systému a dalších spuštěných aplikací spotřebované v průběhu výpočtu. V rámci jednoho systému však lze naměřené hodnoty porovnat a určit relativní rychlosti měřených implementací.

K odečtení počtu cyklů byly použity dva nástroje. Prvním je knihovna `FFTW`[2] a její modul `cycle.h`. Ten implementuje pohodlný způsob odečtení procesorových cyklů na různých platformách, neumožňuje však převod do jednotek času. To umí druhý použitý nástroj, API `QueryPerformanceCounter` z knihovny `Windows.h`. Tento nástroj má schopnost zjistit při odečtení cyklů frekvenci procesoru a naměřené hodnoty lze tedy převést například do milisekund. Právě z tohoto důvodu budou zde uvedené grafy prezentovat hodnoty naměřené nástrojem `QueryPerformanceCounter`.

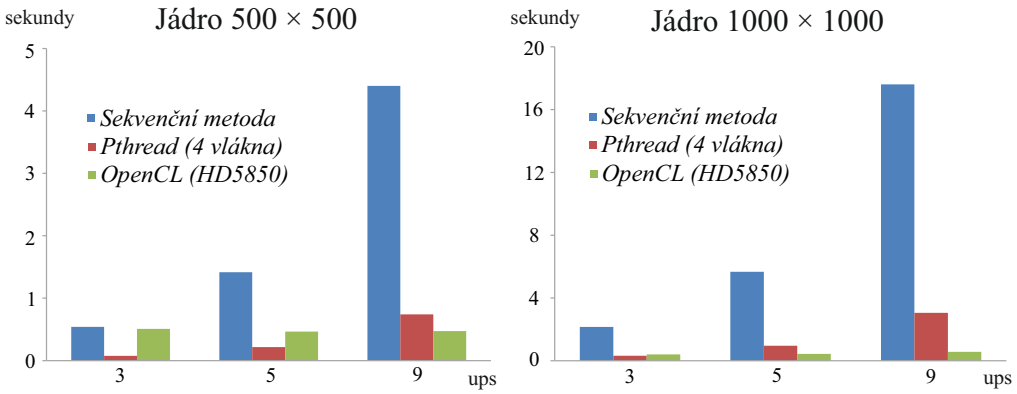
U sekvenční a vícevláknové implementace byl měřen čas výpočtu jádra h^{fin}. Do měření nebyl zahrnut čas potřebný pro alokaci paměti a čas výpočtu Fourierovy transformace jádra. Naproti tomu čas OpenCL verze zahrnuje alokaci paměti `results` určené pro vyzvednutí výsledku v jednoduché přesnosti ze zařízení a zkopírování těchto dat do paměti poskytnuté knihovnou *rayleigh*, která používá dvojitou přesnost.

[2] `http://www.fftw.org/`

3.3.1 Výsledky měření

Byl měřen čas výpočtu čtvercových jader různých velikostí v kombinaci s různými *ups*. Aby byly naměřené hodnoty reprezentativní, byl výpočet každé kombinace velikosti jádra a *ups* proveden 30-krát. Nejnižší naměřená hodnota je pak označena za výsledný čas.

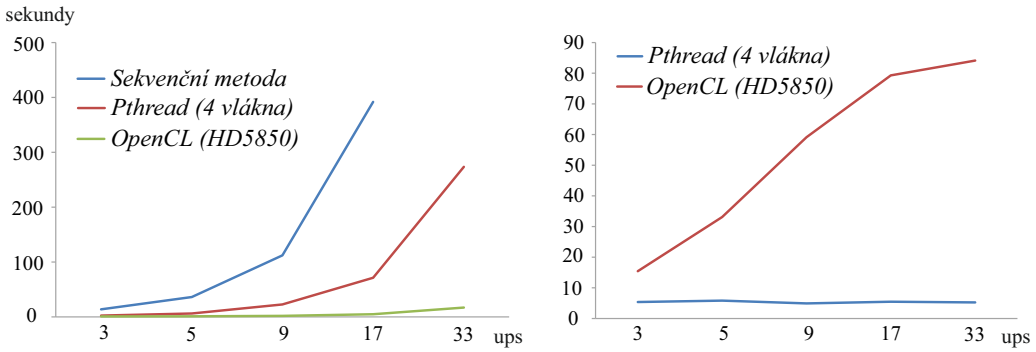
Pro využití plného potenciálu GPU je nutné poskytnout grafické kartě dostatečné množství dat ke zpracování. To je způsobeno nutností inicializace zařízení, transferu dat do a ze zařízení, atp. Obrázek 3.2 představuje grafy doby výpočtu jader o rozměrech 500×500 a 1000×1000. Na levém grafu lze pozorovat jistý konstantní minimální čas výpočtu OpenCL implementace. Tento čas odpovídá inicializaci zařízení a jiným rutinám s tím spojeným. Tyto úkony jsou oproti inicializaci vláken v CPU implementaci časově mnohem náročnější a způsobují, že pro malá pole a malé hodnoty *ups* poskytuje rychlejší výpočet vícevláknová CPU implementace.



Obrázek 3.2: Srovnání rychlosti výpočtu jader malých rozměrů v závislosti na *ups*.

Levý graf na obrázku 3.3 ukazuje trend růstu času výpočtu jádra o rozměrech 2500×2500 v závislosti na zvětšujícím se *ups*. Pravý graf ukazuje relativní urychlení popsaných implementací oproti sekvenční verzi výpočtu jádra velikosti 2500×2500. Pthread verze vykazuje konstantní zrychlení. Zrychlení OpenCL implementace má v grafu trend spíše lineárně rostoucí.

Přestože Pthread implementace byla testována na čtyřjádrovém procesoru se čtyřmi vlákny, je změřené relativní urychlení vyšší než pět. Prakticky by však nemělo být větší než čtyři. Tuto anomálii nelze v rámci této práce ni-



Obrázek 3.3: Trend růstu času výpočtu jádra o rozměru 2500×2500 při vzrůstajícím *ups* a relativní urychlení Pthread a OpenCL implementace oproti sekvenční verzi. V pravém grafu ukazuje svislá osa násobek rychlosti sekvenční verze při stejném výpočtu. Hodnota 40 tedy znamená 40-krát rychlejší výpočet než ten, provedený sekvenčně.

jak vysvětlit. Měření pomocí modulu `cycle.h` a `QueryPerformanceCounter` poskytlo téměř shodné výsledky. Může se jednat o rozdílné vytížení procesoru ostatními aplikacemi při měření jednotlivých implementací, případně o chybné použití nástrojů použitých k měření. Tato anomálie by neměla mít vliv na trendy pozorovatelné v jednotlivých grafech.

V průběhu testování OpenCL implementace bylo zjištěno, že na testovacím zařízení není program při výpočtu velkých jader v kombinaci s velkým převzorkováním stabilní. Původ této nestability nebyl určen, částečně z důvodu obtížného debugování kernelu OpenCL programu. Příčinou může být například zarovnávání alokované paměti na zařízení, optimalizace kernelu při kompilaci, atp.

4 Závěr

Modelování šíření koherentního světla mezi dvěma rovnoběžnými rovinami *source* a *target* může být řešeno Rayleigh-Sommerfeldovým integrálem. Ten je kvůli obtížnému (nemožnému) analytickému výpočtu diskretizován a jeho konvoluční podoba je řešena numericky za použití tří rychlých Fourierových transformací. Tato práce představila dvě různé paralelní implementace algoritmu, který vypočítá filtrované konvoluční jádro h pro zadané plochy *source* a *target*. Sekvenční implementace tohoto algoritmu je součástí knihovny *rayleigh* vyvíjené na Katedře informatiky a výpočetní techniky na Západočeské Univerzitě v Plzni. Tato práce necílí na začlenění těchto implementací do této knihovny. Pojednává pouze o vlastní paralelizaci a samotné implementaci.

První představená verze je založena na klasickém přístupu k paralelizaci. Je určena k běhu na CPU, využívá knihovnu Pthreads-win32 a vlákna. Tato implementace je určena pro běh na běžně dostupných vícejádrových procesorech. Testování rychlosti výpočtu a stability implementace probíhalo na čtyřjádrovém procesoru při spuštění čtyř vláken. Tato implementace používá v téměř nezměněné podobě stejný algoritmus jako implementace sekvenční. Výsledné zrychlení výpočtu tedy souvisí především s počtem dostupných jader (procesorů).

Druhá verze používá standard OpenCL, který umožňuje běh této implementace na mnoha různých zařízeních (x86, ARM, GPU). Určena je však především pro GPU kvůli schopnosti OpenCL optimalizovat a provádět výpočet na SIMD čipech. To přináší vysoký nárůst výkonu oproti běhu na CPU. Pro umožnění vysoké škálovatelnosti postrádá algoritmus implementovaný v této verzi výpočetní a paměťové optimalizace algoritmu použitého v sekvenční a vícejádrové CPU verzi. I přes tuto nevýhodu bylo naměřeno výrazné zrychlení oproti sekvenční verzi algoritmu, a to především u polí velkých rozměrů. OpenCL implementace není schopná distribuovat výpočet na více než jedno zařízení.

Z provedených měření lze soudit, že OpenCL verze je vhodná pro výpočet velkých jader h. Naproti tomu pro výpočet jader malých rozměrů je výhodné použít Pthread implementaci z důvodu časové náročnosti přípravy OpenCL zařízení.

Literatura

[1] ERSOY, Okan K. *Diffraction, fourier optics and imaging.* Wiley-Interscience, Hoboken, 2007. ISBN 978-0-471-23816-4.

[2] FARBER, Rob. Part 2: Opencl^{TM} – memory spaces. *CodeProject.com* [online], 2010 [cit. 2014-05-01]. Dostupné z: `http://www.codeproject.com/Articles/122405/Part-2-OpenCL-Memory-Spaces`.

[3] GASTER, Benedict. *Heterogeneous computing with OpenCL.* Morgan Kaufmann/Elsevier, Waltham, 2013. ISBN 978-0-12-405520-9.

[4] GOVE, Darryl. *Programování aplikací pro vícejádrové procesory.* CPRESS, Brno, 2011. ISBN: 978-80-251-3487-0.

[5] LOBAZ, Petr. Výpočet propagace světla volným prostorem metodou filtrované konvoluce. Technical report, University of West Bohemia, 2012. DCSE/TR-2012-07.

[6] VOELZ, David George. *Computational Fourier Optics: A MATLAB Tutorial.* SPIE Press, Bellingham, 2011. ISBN: 978-0-8194-8204-4.

Seznam obrázků