VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Podpora paralelizace pro makrojazyk ImageJ

Parallelization Support for ImageJ Macro Language

2020

Vojtěch Ondřej

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Vojtěch Ondřej**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Podpora paralelizace pro makro jazyk ImageJ
Parallelization Support for ImageJ Macro Language

Jazyk vypracování: čeština

Zásady pro vypracování:

ImageJ je systém pro analýzu a zpracování obrazu vyvíjený v jazyce Java. Tato aplikace je velmi populární v komunitě biologů z důvodů snadné rozšiřitelnosti pomocí různých pluginů a maker. Cílem bakalářské práce je přidat podporu paralelního provádění maker na více výpočetních uzlech.

1. Popište systém maker, jejich implementaci a návaznosti v prostředí ImageJ.
2. Navrhněte rozšíření makro jazyka tak, aby nově umožňoval paralelní vykonávání operací.
3. Implementujte navržené řešení v prostředí ImageJ.
4. Pro navržené řešení vytvořte sadu demonstračních maker.
5. Proveďte testy a vyhodnoťte dosažené výsledky.

Seznam doporučené odborné literatury:

[1] Javier Fernandez Gonzalez: Mastering Concurrency Programming with Java 9, Packt Publishing, 2017, ISBN 978-1785887949
[2] Jurjen Broeke et al.: Image Processing with ImageJ - Second Edition, Packt Publishing, 2015, ISBN 978-1785889837
[3] Jérôme Mutterer et al.: ImageJ Macro Language Programmer's Reference Guide v1.46d, dostupné z https://imagej.nih.gov/ij/docs/macro_reference_guide.pdf

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Mgr. Ing. Michal Krumnikl, Ph.D.**

Datum zadání: 01.09.2019
Datum odevzdání: 30.04.2020

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 15.5.2020

Vojtěch Ondrieš

....

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.

V Ostravě 15.5.2020

Vojtěch Ondřej
....

Abstrakt

Fiji je open-source platforma založená na ImageJ, která se zabývá zpracováním obrazu pro vědecké disciplíny. Ve Fiji chybí podpora paralelizace, a proto je cílem práce návrh a implementace pluginu, který paralelně zpracovává konvoluční funkci na superpočítači s využitím OpenMPI. Konvoluční funkce se skládá z aplikace filtru na vstupní obrázek a vrácení výstupního obrázku. Proces implementace je v práci popsán a demonstrován. Navržený plugin lze využít i v rámci makrojazyka ImageJ. Na závěr je implementace testována na superpočítači s různými vstupními parametry.

Klíčová slova: Fiji; ImageJ; OpenMPI; Konvoluce; Paralelizace

Abstract

Fiji is an open-source platfrom based on ImageJ software that focuses on scientific image processing. Fiji lacks paralelization support and therefore the goal of this thesis is to develop and implement plugin that uses parallelization to apply convolution algorithm on image by using OpenMPI. Convolution consists of applying convolution filter on image and returns processed image. Detailed process of implementation is described and showcased. The plugin can be also used in ImageJ macro language. The implementation is then tested on supercomputer with variable input parameters.

Keywords: Fiji; ImageJ; OpenMPI; Convolution; Parallel

Obsah

Seznam použitých zkratek a symbolů **8**

Seznam obrázků **9**

Seznam tabulek **10**

Seznam výpisů zdrojového kódu **11**

1 Úvod **12**

2 Fiji **14**
- 2.1 Paralelizace ve Fiji . . . 15

3 Paralelní architektury pro masivně paralelní stroje **17**
- 3.1 OpenMPI . . . 17
- 3.2 OpenCL . . . 17
- 3.3 CUDA . . . 18
- 3.4 Apache Spark . . . 18
- 3.5 Apache Hadoop . . . 19
- 3.6 MapReduce . . . 19
- 3.7 Amdahlův zákon . . . 20
- 3.8 Gustafsonův zákon . . . 20

4 Konvoluce **21**
- 4.1 Definice . . . 21
- 4.2 Kernely . . . 21
- 4.3 Separovatelná a neseparovatelná konvoluce. . . . 24
- 4.4 Separovatelné kernely . . . 26
- 4.5 FFT konvoluce . . . 27
- 4.6 Popis paralelizace . . . 28
- 4.7 Problém distribuce dat mezi uzly . . . 28
- 4.8 Paralelní konvoluce . . . 29

5 Implementace konvoluce **31**
- 5.1 Tvorba pluginu . . . 31
- 5.2 Ošetření vstupů konvoluční funkce . . . 33
- 5.3 Paralelizace skrze OpenMPI . . . 33
- 5.4 Instalace pluginu . . . 43

6 Testování **44**
6.1 Popis vývojového prostředí 44
6.2 Proces testování 44
6.3 Naivní implementace 45
6.4 FFT implementace 46
6.5 Optimalizovaná naivní implementace 47
6.6 Separovatelná implementace 47
6.7 Porovnání implementací 48
6.8 Porovnáni paralelní a sériové implementace 50
6.9 Paralelní separovatelná implementace a FFT 51

7 Závěr **55**

Literatura **56**

Seznam použitých zkratek a symbolů

MPI	–	Message Passing Interface
CUDA	–	Compute Unified Device Architecture
NUMA	–	Non-Uniform Memory Access
RMI	–	Remote Method Invocation
SPIM	–	Selective Plane Illumination Microscopy
RMI	–	Message Passing Interface
FFT	–	Fast Fourier Transform
DFT	–	Discrete Fourier Transform
OpenCL	–	Open Computer Language
GPU	–	Graphical Processing Unit
CPU	–	Central Processing Unit
RDDs	–	Resilient Distributed Datasets
I/O	–	Input/Output
SSH	–	Secure SHell

Seznam obrázků

1 Konvoluce obrazu s kernelem 3×3. 22
2 Přetečení kernelu 25
3 Rozdělení pixelů pro konvoluci 25
4 Rozdělení pixelů pro separovatelnou konvoluci 26
5 Rozdělení dat mezi uzly 29
6 Naivní konvoluce, rozdělení pixelů. 38
7 Separovatelná konvoluce rozdělení pixelů 41
8 Graf výkonnosti naivní implementace. 46
9 Graf výkonnosti FFT implementace. 47
10 Graf výkonnosti vlastní naivní implementace. 48
11 Graf výkonnosti separovatelné implementace. 49
12 Porovnání výkonnosti implementací na datech velikosti 1000×1000 50
13 Porovnání výkonnosti implementací na datech velikosti 2000×2000 51
14 Porovnání výkonnosti implementací na datech velikosti 5000×5000 52
15 Porovnání výkonnosti implementací na datech velikosti 10000×10000 52
16 Porovnání výkonnosti paralelní implementace na datech velikosti 10000×10000 . 53
17 Porovnání výkonnosti paralelní implementace na datech velikosti 5000×5000 . . . 53
18 Porovnání paralelní a FFT implementace na datech velikosti 10000×10000 54

Seznam tabulek

1 Výkonnost naivní implementace na jednom uzlu 45
2 Výkonnost FFT implementace na jednom uzlu 46
3 Výkonnost vlastní naivní implementace na jednom uzlu 48
4 Výkonnost separovatelné implementace na jednom uzlu 49

Seznam výpisů zdrojového kódu

1 Vnořené cykly konvoluce. . . . 24
2 Anotace pluginu. . . . 31
3 Hlavní funkce pluginu. . . . 31
4 Run funkce pluginu. . . . 32
5 Inicializace OpenMPI. . . . 33
6 Omezení počtu volání příkazů. . . . 33
7 Přetečení kernelu a rozšíření obrázku. . . . 34
8 Extrakce obrázku ze souboru. . . . 34
9 Funkce rozšiřující obrázek. . . . 35
10 Rozdělení obrázku mezi uzly. . . . 35
11 Komunikace kořenového uzlu. . . . 36
12 Komunikace posledního uzlu. . . . 37
13 Komunikace prostředních uzlů. . . . 37
14 Volání konvolucí. . . . 38
15 Výpočet konvoluce nad středem obrazu. . . . 38
16 Výpočet konvoluce horní hrany obrazu. . . . 39
17 Výpočet konvoluce levé hrany obrazu. . . . 40
18 Střed separovatelné konvoluce v horizontálním směru. . . . 41
19 Levá hrana separovatelné konvoluce v horizontálním směru. . . . 41
20 Horní hrana separovatelné konvoluce ve vertikálním směru. . . . 42
21 Složení obrázku do jednoho celku. . . . 43
22 Ukázka volání pluginu v ImageJ makrojazyku. . . . 43

1 Úvod

S příchodem moderních technologii v oblasti biologie a mikroskopie se již příliš často nesetkáme se situací, kdy by vědecký pracovník nahlížel do mikroskopu a sledoval pohyb buněk nebo jejich vývoj. V dnešní době jsou k mikroskopům připojeny moderní kamery, které snímají snímky daleko rychleji a detailněji, než je lidské oko schopno postřehnout. To ovšem přináší nové výzvy v oblasti zpracování dat. V historii byly kamery pomalejší a negenerovaly obrovská kvanta dat, a tudíž se data dala poměrně efektivně zpracovat na stolním počítači. Moderní kamery dokážou snímat desítky až stovky vysoce kvalitních snímků za vteřinu, a to znamená stovky megabytů dat za vteřinu. Tyto data se musí uložit a zpracovat. Klasické stolní počítače již bohužel nezvládají zpracování takového množství dat, a proto je tendence přecházet na superpočítače, superpočítačová centra nebo cloudy. Superpočítačová centra jsou ideálním řešením zpracování velkého počtu dat, jelikož nabízejí výkon několika set násobně převyšující výkony standardních stolních počítačů. Zpracování dat na superpočítačích je ale bohužel logisticky mnohem náročnější než u standardních počítačů, a proto je nutné nalézt vhodné řešení k implementaci programů pro zpracování dat. K implementaci těchto programu je tedy potřeba zvážit výběr vhodné platformy.

V současné době existuje řada platforem pro zpracování biologických dat, například Fiji[1], ImageJ[2] nebo KNIME[3]. Tato práce se soustředí na platformu Fiji, která je oblíbená v komunitě biologů z důvodů snadné rozšířitelnosti funkčnosti o různé pluginy a makra. Fiji je platforma na zpracování biologických dat odvozená od platformy ImageJ. Oproti ImageJ, které by se dalo považovat jako univerzální nástroj ke zpracování obrazu, se Fiji soustředí právě na pluginy navržené pro vědeckou komunitu. Fiji bylo původně navrženo pro neurovědu, ale postupem času nakumulovalo spoustu dalších pluginů pro jiné vědecké obory jako biologie, parazitologie, genetika a jiné. Fiji poskytuje širokou škálu pluginů, přehledný panel nástrojů a srozumitelnou dokumentaci díky čemuž se stalo oblíbeným nástrojem ve vědecké komunitě. Jelikož je Fiji open-source, nabízí i možnosti vlastního a poměrně jednoduchého vývoje pluginů, právě díky srozumitelné dokumentaci. Ve Fiji bohužel chybí podpora paralelizace, která se jeví jako vhodný nástroj pro zpracování velkého množství biologických dat. Cílem této práce je rozšířit Fiji o konvoluční filtry podporující paralelizaci. Původní záměr byl využití maker nicméně v této době jsou již makra zastaralé řešení, které mají omezenou funkcionalitu a nepodporují využití OpenMPI[4], které v této práci budeme využívat. Navržené řešení se tak zaměřilo na návrh a implementaci paralelního ImageJ2 pluginu využívajícího rozhraní OpenMPI. Plugin ale zůstává zpětně kompatibilní s makrojazykem ImageJ a může v něm být využit.

OpenMPI je jedna z nejpopulárnějších implementací MPI (Message Passing Interface), což je dominantní programovací vzor pro paralelní aplikace na počítačích s rozdělenou pamětí. OpenMPI je primární paralelní platforma využívaná k programování na superpočítačích, která nabízí bohatou sadu funkcí jako například kolektivní komunikaci procesů, která chybí v obecných návrzích typu soketů nebo RMI (Remote Method Invocation), z nichž oba jsou spíše orientovány na klient-server aplikace. OpenMPI nám umožňuje efektivní rozdělení dat mezi výpočetními uzly

a jejich následnou komunikaci, což se skvěle hodí právě pro rozdělení vstupních dat a výpočtu nad nimi. Dalším problémem, který musíme řešit při implementaci programů pro zpracování dat na superpočítačích je výběr vhodné platformy. Vhodných platforem je několik. Patří mezi ně například OpenMPI, kterým se dají zpracovat data na jednotlivých uzlech superpočítačů nebo CUDA, kde se data zpracovávají na grafických kartách. Tyto a další platformy si níže podrobněji popíšeme a porovnáme jejich výhody a nevýhody.

2 Fiji

Fiji[1] je open-source platforma pro zpracování obrazu, která vychází z platformy ImageJ[2]. ImageJ spolu s jinými podobnými platformami jako například KNIME[3] jsou softwarové platformy pro analýzu biologických snímků. Komerční platformy jako Imaris[1], Volocity[2] and Amira[3] se specializují na jednoduchost použití a dostupnost velkého množství různých funkcí pro zpracování obrazu zaměřených spíše na nezkušené uživatele. Ve většině případů jsou pak důležité detaily algoritmů pro zpracování obrazu uživateli nedostupné, což není vhodné pro výzkum. Naopak u ImageJ, který je open-source, jsou detaily algoritmů pro zpracování obrazu transparentní. Další velkou výhodou ImageJ je také jeho dlouhodobá existence, rozsáhlá osvojitelnost a jednoduchá rozšiřitelnost pluginů. Všechny tyto výhody z něj dělají vhodný a jeden z nejčastěji používaných softwarů ve vědecké komunitě napříč disciplínami.

Nicméně ImageJ byl vyvinut biology pro biology a jeho architektura neodpovídá moderním softwarovým principům. Díky tomu je méně přívětivý pro vývojáře, kteří pracují na nových algoritmech. Aby se předešlo úpadku vývojářů, byl vyvinut nový open-source projekt Fiji, který vylepšuje vnitřní architekturu ImageJ a umožňuje vývojářům zaměřit se na vývoj inovativních algoritmů pro analýzu biologických dat. Fiji přináší výkonné nástroje pro rychlý přesun od vývoje nových algoritmů k užitečným nástrojům pro zpracování obrazu. Vývojář může také využít algoritmy z jádra Fiji skrze různé skriptovací jazyky k vývoji nových algoritmů. Fiji také poskytuje stabilní distribuční systém, který zajišťuje, že se nově vyvinuté algoritmy dostanou k uživatelům co nejrychleji. Fiji bylo navrhnuto jako ekosystém, který spojí biologii s informatikou, kde vývojáři a vědci můžou spolupracovat na tvorbě nových výkonných algoritmech použitých pro vědecký výzkum. Fiji udržuje kompatibilitu s ImageJ a rozšiřuje jeho jádro o dodatečnou funkcionalitu. Fiji je tedy ve zkratce open-source distribuce ImageJ, která obsahuje širokou škálu knihoven, pluginů pro výzkum biologie, skriptovací jazyky, rozsáhlé tutoriály a kvalitní dokumentaci.

Analýza obrázků biologických vzorků je velmi zdlouhavý proces, který si pro každý snímek z kolekce obrázků obvykle vyžaduje spuštění několika algoritmů za sebou. ImageJ naštěstí podporuje skriptování, které využívá sérii jednoduchým programovacích příkazů, jimiž definuje sekvenci algoritmů, které se aplikují na sekvenci snímků. V ImageJ je také dostupný jednoduchý makrojazyk, který umožňuje nahrávání příkazů a sestavování jednoduchých programů, a proto je velmi oblíbený mezi uživateli. Fiji rozšiřuje funkcionalitu maker a skriptů ImageJ o řadu dalších skriptovacích jazyků jako Jython, JavaScript a podobně. Tyto skriptovací jazyky mohou být použity bez nutnosti znalosti Javy, ve které je ImageJ napsaný a které využívá makrojazyk ImageJ. V porovnání s makrojazykem ImageJ poskytují skriptovací jazyky pokročilejší syntaxi, zatímco si zachovávají relativní jednoduchost. Díky tomu jsou skriptovací jazyky přívětivější pro

[1] Imaris: Software for 3D and 4D Imaging, www.bitplane.com
[2] Volocity: A universal solution for 3D image analysis, quorumtechnologies.com
[3] Amira: 2D–5D visualization and analysis software, www.thermofisher.com

běžného programátora i pro profesionálního vývojáře. Makrojazyk ImageJ je již bohužel zastaralý a nepodporuje OpenMPI (viz kapitola 3), které budeme hojně využívat pro implementaci paralelizace. Z tohoto důvodu se v této práci nebude využívat a namísto něj se použije plugin, který je pro úkol paralelizace mnohem přívětivější.

2.1 Paralelizace ve Fiji

Fiji nemá velkou podporu paralelizmu, nicméně pár projektů se již pokusilo o paralelizaci ve Fiji. Jedním z těchto příkladů je projekt: „*An automated workflow for parallel processing of large multiview SPIM recordings*“[5].

2.1.1 Selective Plane Illumination Microscopy (SPIM)

Selective Plane Illumination Microscopy (SPIM) umožňuje dlouhodobé tří dimenzionální snímaní vyvíjejících se organizmů ve vysokém rozlišení. Výsledkem jsou obrovská kvanta dat, která musí být interaktivně zpracována skrze dedikované aplikace. Postupné kroky zpracování dat mohou být jednoduše automatizovány a každý snímek může být zpracován samostatně. Díky tomu lze tento proces poměrně jednoduše paralelizovat na superpočítačích. V tomto projektu autor vytváří automatizovaný tok procesů ke zpracování velkého množství SPIM dat z různých úhlů, kanálů a úrovní osvětlení. Tento tok je tvořen jak sekvenčně na jednotlivých počítačích, tak i paralelně na superpočítačích[5].

2.1.2 Fiji Archipelago

Další projekt, který implementuje paralelizace ve Fiji je FijiArchipelago[4]. Fiji Archipelago je nástroj ulehčující programátorům přenos jiných pluginů nebo funkcionalit psaných ve Fiji/Imagej po síti na jiné počítače. Tento plugin funguje na principu klient-server, kdy kořenový uzel (root) spouští a udržuje cluster, a klienti jsou další počítače připojeny ke clusteru. Klienti pak zpracovávají výpočetní úkony rozesílané kořenovým uzlem. Fiji Archipelago tak využívá celý cluster k výpočtu složitějších programů. Jelikož se program obvykle skládá z více dělitelných částí, můžeme tyto části rozdělit mezi klientské uzly v clusteru a urychlit tak vykonání programu. Cluster pak může fungovat dvěma způsoby. První způsob je, že kořenový uzel spouští klientské uzly a rozděluje jim data. Druhý způsob je založen na principu, že se prvně manuálně spustí klientský uzel, který potřebuje spustit nějakou úlohu a ten zašle pokyn ke spuštění kořenového uzlu, a tedy inicializaci clusteru. Komunikace mezi uzly je implementována buďto skrze standardní I/O protokol přes SSH anebo případně skrze posílání nezabezpečených soketů.

Posledním projektem, který si popíšeme a který docílil určité paralelizace Fiji je KNIME.

[4]FijiArchipelago: plugin that brings Cluster functionality to Fiji, https://imagej.net/FijiArchipelago

2.1.3 KNIME a KNIME Cluster Execution

KNIME[3], The Konstanz Information Miner je modulární vývojové prostředí, které umožňuje jednoduché složení a spuštění kódu. Uživatel skládá kód využitím jednotlivých modulů čímž tvoří výsledný program. KNIME je kompatibilní s ImageJ a umožňuje kombinovat prvky KNIME s prvky ImageJ. Je navrhnut jako studijní a výzkumná platforma pro spolupráci uživatelů, která umožňuje jednoduchý návrh nových algoritmů a nástrojů.

Jedním z těchto nástrojů je KNIME Cluster Execution[6], který poskytuje tenkou spojovací vrstvu mezi KNIME a vysoce výkonným výpočetním clusterem. Díky této vrstvě může být každá aplikace fungující v KNIME distribuovaná v clusteru mezi dostupné uzly. Clustery jsou často nečinné, jelikož si jejich údržba a provoz vyžaduje velkou režii. Přenos dat do clusteru a uvedení samotného clusteru do chodu jsou pro uživatele zbytečně náročné operace, a proto byl vyvinut KNIME Cluster Executor, který zjednodušuje interakci s clusterem a snižuje nároky na provoz a údržbu clusteru tím, že se sám stará o přesun výpočetně náročných operací z vědeckých počítačů přímo do clusteru.

3 Paralelní architektury pro masivně paralelní stroje

Masivně paralelními stroji chápeme stroje, které mají velké množství výpočetních jednotek pracujících na jedné úloze[7]. Takové stroje pak můžeme rozdělit do třech rodin. První rodina obsahuje stroje složené z velkého množství počítačů, kterým říkáme superpočítače. Pro tuto rodinu existují různé platformy pro paralelizaci jako například OpenMPI nebo OpenCL. Obě tyto platformy jsou popsány níže. Druhá rodina obsahuje výpočetní jednotky, které jsou samy schopné masivní paralelizace ale které jsou částí většího stroje. Příkladem můžou být grafické karty v počítačích. Paralelní platforma pro tuto rodinu je například CUDA, které je taktéž popsána níže. Třetí rodinou, kterou si v této práci popíšeme je tzv. cloud. Cloud využívá internetu na ukládání dat a aplikací namísto pevného disku. Dalo by se říct, že cloud je internet a proto všechny výpočty a přenosy dat probíhají na síti[8]. Platformy pro paralelizaci strojů v této rodině jsou například Apache Spark nebo Apache Hadoop.

3.1 OpenMPI

MPI (Message Passing Interface) je jazykově nezávislá komunikační knihovna používaná k vytváření kódu pro paralelní počítače. Podporuje komunikaci mezi procesy, které představují paralelní program běžící na systémech s rozdělitelnou pamětí. Komunikace může být jak kolektivní, tak čistě mezi uzly. Cílem MPI je vysoká výkonnost, škálovatelnost a přenositelnost. Implementace skrze MPI je natolik chytrá, že dokáže rozpoznat, jestli je program využíván na systémech se sdílenou pamětí a optimalizovat tak její chování. Vyvíjeni programů za pomocí MPI může mít své výhody oproti například NUMA (Non-Uniform Memory Access) architekturám protože MPI klade důraz na lokalizaci paměti[9].

3.2 OpenCL

OpenCL[10] neboli Open Computer Language je framework, který je považován za standard při programování počítačů složených z více CPU, GPU a jiných procesorů. Tyto heterogenní systémy se staly osamocenou třídou platforem na jejichž požadavky se OpenCL zaměřuje. S OpenCL můžeme napsat jednoduchý program, který poběží na různorodých platformách od mobilních telefonů nebo laptopů až po jednotlivé uzly v superpočítačích. OpenCL disponuje velmi dobrou přenositelností, protože odhaluje hardwarové prvky a neskrývá je za elegantní abstrakcí jako to dělají jiné platformy. V OpenCL musí vývojář specificky definovat platformu, její kontext, a jak bude práce vykonána skrze dostupná zařízení. Vysoko úrovňové programovací modely často potřebují nějakou pevnou základnu pro tvorbu aplikací, kterou může být právě OpenCL. Zajímavým projektem, který implementuje jak ImageJ tak OpenCL je CLIJ[11].

3.2.1 CLIJ: GPU-accelerated image processing

CLIJ[11] se dá považovat za most mezi OpenCL a ImageJ. Je to zároveň i Fiji plugin, který umožňuje uživateli s omezenými programovacími dovednostmi vytvořit GPU-accelerated program k urychlení zpracování obrazu. Volně programovatelné aplikace v CLIJ umožňují docílit urychlení zpracování vstupního obrazu i více než desetkrát oproti neparalelním aplikacím. Tohoto zrychlení je možné docílit jak na systémech se špičkovými GPU, tak na dostupných přenositelných laptopech s integrovaným GPU. CLIJ klade důraz na dobrou dokumentaci, přehledné ukázkové kódy, interoperabilitu (schopnost různých systému spolupracovat) a rozšiřitelnost. CLIJ je založeno na ClearCL[5], JOCL[6], Imglib2[12], ImageJ a SciJava[7]. CLIJ doplňuje jádro ImageJ přeprogramovatelnými prvky, které využívají OpenCL9 framework pro spuštění kódu na GPU. V CLIJ je dlouhá řada základních funkcí pro zpracování obrazu jako například hledání minima a maxima, 3D a 2D projekce, warping a morfing obrazu a podobně.

3.3 CUDA

The Compute Unified Device Architecture (CUDA) je paralelní výpočetní platforma a programovací model vytvořený společností nVidia. CUDA poskytuje vývojářům přístup ke strojovým instrukcím a paměti paralelních grafických jednotek (GPU) nVidia. CUDA je vývojářům přístupná přes CUDA-accelerated knihovny nebo adresáře překladačů. CUDA pak dále poskytuje rozšíření pro různé programovací jazyky jako C/C++ a Fortran a spoustu dalších rozhraní jako OpenCL, Microsoft Direct Compute a C++ AMP. CUDA pracuje na kódu nižší úrovně než výše zmíněná rozhraní a proto je pro práci s ní potřeba vyšších programovacích dovedností[9]. Jelikož ale CUDA poskytuje poměrně jednoduchou a minimalistickou abstrakci paralelismu a obsahuje známou sémantiku jazyka C umožňuje zkušenějším programátorům psát paralelní aplikace s relativní lehkostí[13].

Platformy, které jsme si zatím popsali využívají CPU a GPU na počítačích. Nyní se zkusíme podívat na cloudy. Popíšeme si dva příklady paralelních platforem na cloudu a jsou to Apache Spark a Hadoop.

3.4 Apache Spark

Apache Spark je open-source programovací framework pro zpracování velkých dat. Vznikl jako výpočetní nástroj nové generace pro zpracování velkých dat a překonal výkonnost Hadoop MapReduce (viz kapitola 3.6) algoritmu což vedlo k revoluci ve zpracování velkých dat. Spark udržuje lineární škálovatelnost MapReduce algoritmu a jeho toleranci k chybám nicméně jej rozšiřuje o pár podstatných věcí. Spark je výrazně rychlejší, a to až sto násobně při určitých aplikacích. Spark je také výrazně jednoduší k programování díky bohatému API v Pythonu, Javě

[5]ClearCL: Multi-backend Java Object Oriented Facade API for OpenCL, https://biii.eu/clearcl
[6]JOCL: Java bindings for OpenCL, http://www.jocl.org/
[7]SciJava: collaboration of projects providing software for scientific computing, https://scijava.org/

a dalších[14]. Programový model Sparku je sice podobný s MapReduce algoritmem ale rozšiřuje jej o abstraktní sdílení dat nazývané RDDs (Resilient Distributed Datasets). Užitím tohoto rozšíření umí Spark zpracovat široké množství procesů, které si původně vyžadovaly samostatné specializované enginy. Takové procesy zahrnují například SQL, streamování, strojové učení a zpracovávaní grafů. Tato implementace využívá stejné optimalizace jako specializované enginy a má podobnou výkonnost, nicméně běží jako knihovna nad společným enginem což znamená, že se mnohem snadněji implementuje[15].

3.5 Apache Hadoop

Apache Hadoop je škálovatelný, open-source programovací framework založený na Javě, který se využívá při zpracování a ukládání velkých dat. Hadoop se využívá na více uzlových paralelních systémech, které zpracovávají velká množství dat v řádech terabytů. Hadoop využívá vlastní souborový systém HDFS, jenž vyniká rychlými přenosy dat, které dokáží ustát i selhání uzlu. HDFS využívá MapReduce (viz kapitola 3.6) algoritmu pro rozdělení velkých dat na menší data, která se pak zpracovávají. HDFS je navržen tak aby spolehlivě uložil velká data mezi velké množství dostupných uzlů[16]. Technologie Apache Hadoop se využívá pro zpracování řady funkcí jako například optimalizace vyhledávání dotazů, vytěžování dat z informaci (data mining) nebo ke strojovému učení[17].

3.6 MapReduce

MapReduce je programový model a implementace pro zpracování a generování velkých dat, které jsou pak přístupné velkému množství dostupných funkcí. Uživatel specifikuje výpočet pomocí mapy dat a redukční funkce, díky které pak runtime systém automaticky paralelizuje výpočet rozložením dat mezi dostupné uzly. Runtime systém také řeší případně chyby uzlů a zpracovává komunikaci mezi uzly tak aby byla efektivně využita síť a volné místo na disku. Využití MapReduce je poměrně přímočaré, a proto také velmi oblíbené a hojně využívané programátory, kteří již implementovali více než desítky tisíc programů využívajících MapReduce[18].

Jak můžeme vidět výše, platforem pro vývoj paralelních aplikací je velká spousta a nám nezbývá než si vybrat pouze jednu. Platformy využívající cloud jsou velmi zajímavá a lákavá možnost. Nicméně pracují s opravdu velkými kvanty dat, které se u zpracování obrázku konvoluční funkcí zatím neobjevují. Jelikož je tato práce řešena ve spolupráci s centrem IT4Innovation a jejich superpočítači, na kterých je využití CUDA spíše minoritní, můžeme se rovnou podívat na OpenCL versus OpenMPI. OpenCL je lákavá platforma hlavně z důvodu, že umí pracovat jak s GPU, tak s CPU, je snadno přenositelná a rozšiřitelná. Nicméně pro tento projekt byla zvolena platforma OpenMPI, která funguje už desetiletí, je dostupná snad na všech superpočítačích, je spolehlivá a léty ověřená.

Všechny výše zmíněné platformy ovšem čelí jistým omezením paralelizace. Z důvodu těchto omezení bohužel nemůžeme paralelizovat do nekonečna a dosáhnout tak nekonečného zrychlení aplikací. Tato omezení pak popisují Amdahlův a Gustavsonův zákon níže:

3.7 Amdahlův zákon

Amdahlův zákon popisuje, že paralelizací programu můžeme dosáhnout jen poměrně omezeného zrychlení. Zákon je postaven na rozdělení programu do dvou částí. První z těchto částí je sekvenční část programu, které nelze paralelizovat a druhá část je paralelizovatelná část programu. Amdahlův zákon říká, že výsledné zrychlení programu je závislé na době trvání sekvenční části programu. Program tedy nikdy nepoběží rychleji, než je doba trvání sekvenční části programu, nehledě na počet využitých uzlů v paralelizovatelné části programu[19].

3.8 Gustafsonův zákon

Gustavsonův zákon využívá nedostatku Amdahlova zákonu, kdy Amdahl s rostoucím počtem uzlů neměnil velikost řešeného problému. Gustafsonův zákon nepopisuje dodatečného zrychlení výpočtu k Amdahlovu zákonu, nicméně říká, že za stejnou dobu výpočtu můžeme při vyšším počtu uzlů dosáhnout mnohem přesnějších výpočtů. Gustafsonův zákon tedy dokazuje, že pokud měníme náročnost nebo velikost problému, můžeme dosáhnout libovolného zrychlení[20].

4 Konvoluce

Cílem této práce je přizpůsobení zvoleného filtru tak aby jej bylo možné paralelizovat. Pro tuto práci byl zvolen konvoluční filtr, neboť je to dostatečně náročná operace, dá se jednoduše škálovat do velkých rozměrů a slouží jako ideální benchmark k porovnání výkonnosti různých implementací. V této kapitole si nejdříve popíšeme, co to vlastně konvoluce je, uvedeme si příklady konvolučních kernelů, rozdělíme si konvoluci na separovatelnou a neseparovatelnou a popíšeme si jejich výhody a nevýhody.

4.1 Definice

Konvoluce[21] je matematická operace dvou signálu f a g definována jako:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(x)\, g(t-x)\, dx \tag{1}$$

(f*g)(t) je často považováno za odfiltrovanou verzi vstupního signálu f(t), kde g(t) je jádro využívané k filtraci. Jedna ze základních vlastností této operace je definována konvolučním teorémem[21], který tvrdí že:

$$F\{f * g\} = kF\{f\}F\{g\} \tag{2}$$

Kde F je Fourierova transformace signálu. Pak tedy konvoluce v čase je rovna roznásobení hodnot ve frekvenční doméně. To znamená, že při správně zvoleném kernelu můžeme buďto násobit nebo odebrat dané frekvence signálu. Při digitálním zpracování obrazu můžeme tuto techniku použít například pro zaostření nebo naopak rozmazání obrazu. Existují i další aplikace jako například algoritmy hledání hran, nicméně na ty se podíváme později.

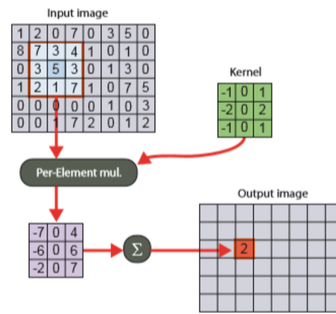
Pokud je obraz znázorněn jako dvou dimenzionální pole diskrétního signálu, pak můžeme použít diskrétní dvou dimenzionální konvoluci za použití diskrétního kernelu[21].

$$(y \cdot k)[i,j] = \sum_n \sum_m y[i-n, j-m,] \cdot k[n,m] \tag{3}$$

Jednoduše řečeno, konvoluce je produkt roznásobení hodnot kernelu a hodnot vstupního obrazu k získání výsledné hodnoty zpracovávaného pixelu. Obrázek 1 znázorňuje aplikaci kernelu na vstupní obraz. Můžeme vidět, že hodnoty kernelu se násobí s hodnotami pixelu na stejné pozici. Hodnoty se sečtou a vydělí celkovým počtem hodnot. Tím získáme průměr hodnot, a tedy i hodnotu výsledného pixelu. Kernel se v dalším kroku posune na vedlejší pixel a proces se opakuje[21].

4.2 Kernely

Existuje velké množství předdefinovaných kernelů pro různé aplikace jako například rozmazání obrazu, hledání hran, zaostření obrazu a podobně. Jelikož je konvoluce definována jako skalární



(a) Kernel 3×3

Obrázek 1: Konvoluce obrazu s kernelem 3×3. Převzato z[21]

součin dvou matic, můžeme vstupní obrázek použít libovolný symetrický kernel. Výsledek operace pak bude vždy záviset na vstupním kernelu. Matice níže znázorňují jedny z nejzákladnějších kernelů. Jsou to zleva: Vyhlazování průměrováním (Box blur), Gaussovo vyhlazování a Zaostření. Vyhlazování průměrováním a Gaussovo vyhlazování využívají denominátor a tedy jsou děleny součtem všech hodnot kernelu, naopak Zaostřování denominátor vůbec nepoužívá[21].

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}, \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}.$$

Navržení kernelu si vyžaduje opatrný výběr jeho hodnot abychom si byli jisti, že kernel dosáhne žádaných výsledků. Níže si tyto základní kernely používané ke zpracování obrazu představíme podrobněji.

4.2.1 Vyhlazování průměrováním

Průměrování neboli Box blur[21] je nejjednodušší separovatelný filtr. Tento filtr najde hodnoty sousedních pixelů každého pixelu a vypočítá z nich jejich průměr. Rozmazávání průměrování není hladký filtr, a tedy rozmazává úplně vše včetně šumu a bohužel i jemných detailů v obrazu. Výsledkem je sice rozmazaný obraz, nicméně je výrazně ovlivněna naše schopnost rozpoznat detaily v obrazu, které byly předtím viditelné. Box filtr může být znázorněn buďto jako matice anebo jako skalární součin dvou vektorů u a v.

$$Box_3 = 1/9 \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}; u = v = 1/3 \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \tag{4}$$

4.2.2 Gaussův filtr

Rozmazávání průměrováním je poměrně jednoduchá operace, nicméně jejím výsledkem není hladce rozmazaný obraz. K hladkému vyhlazování obrazu se používá Gaussův filtr[21]. Gaussův filtr je použit ve velké spoustě grafických aplikací ke snížení šumu nebo k odstranění nechtěných detailů před rozpoznáváním relevantních hran. Gaussův filtr je dolnopropustní filtr, zmírňující vysokofrekvenční komponenty obrazu. Dvou dimenzionální Gaussova funkce je produktem dvou jedno dimenzionálních Gaussových funkcí:

$$G\left(x\right) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}; G\left(x, y\right) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{5}$$

Níže je příklad Gaussova filtru 5×5 znázorněného buď maticí anebo výsledkem skalárního součinu dvou vektorů u a v:

$$G_5 = 1/256 \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}; u = v = 1/16 \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \end{bmatrix} \tag{6}$$

Aplikací Gaussova filtru dosáhneme efektivního odstranění šumu při zachování důležitých detailů v obraze.

4.2.3 Zaostřovací (Sharpness) filtr

Cílem zaostřovacího filtru[21] je zvýraznění detailů vstupního obrazu. Nejjednodušší filtr je definován maticí 3×3 a může být napsán jako kterákoliv z následujících matic:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}, \begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix} \text{ nebo } \begin{bmatrix} -k & -k & -k \\ -k & 8k+1 & -k \\ -k & -k & -k \end{bmatrix},$$

kde k je reálné číslo. Z matic můžeme vidět, že zaostřovací filtr, vypočítá pro každý pixel rozdíly jeho sousedních pixelů k původní hodnotě. Hodnota původního pixelu je vždy větší než absolutní hodnota součtu sousedních pixelů. To znamená, že původní pixel si drží původní hodnotu, která se následně navýší o rozdíl sousedních pixelů[21].

4.2.4 Filtr hledání hran

Abychom byli schopni rozpoznat hrany v obraze, musíme vypočítat gradient vstupního obrazu v určitém směru. Použitím jednoho z následujících kernelů při konvoluci, dosáhneme žádaného výsledku rozpoznání hran. Místa v obraze, kde se zásadně mění intenzita pixelu, budou mít

výrazně vyšší hodnoty než místa bez zásadní změny intenzity. Tato technika není bohužel úplně praktická, neboť může navyšovat šum v obraze a detekuje pouze hrany v určitém směru[21].

$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \text{nebo} \begin{bmatrix} -1/8 & -1/8 & -1/8 \\ -1/8 & 1 & -1/8 \\ -1/8 & -1/8 & -1/8 \end{bmatrix} \tag{7}$$

4.3 Separovatelná a neseparovatelná konvoluce.

Díky separovatelnosti kernelů máme dva způsoby, jak psát algoritmus konvoluce. Algoritmus se na základě vstupního kernelu může rozhodnout, jaký druh konvoluce použít. Definice separovatelnosti bude následovat níže.

4.3.1 Neseparovatelná konvoluce

V případě neseparovatelného kernelu se musíme spokojit s takzvanou naivní konvolucí[21]. Je to základní druh konvoluce, kdy pro každý pixel vstupního obrázku musíme udělat k^2 operací, kde k šířka vstupního kernelu. V čistě naivní konvoluci se oříznou kraje obrazu o $x(k/2)$, kde x je počet oříznutých řádků nebo sloupců a k je šířka kernelu[21]. Při dělení celých čísel se vždy zaokrouhluje dolů a proto k/2 je jedna pro k=3, dva pro k=5, tři pro k=7 a podobně. Díky tomuto oříznutí nemusíme řešit jinak problematické hrany a rohy obrazu a konvoluce se počítá jen pro střed obrazu. Kód je pak poměrně přímočarý, neboť se jedná pouze o čtyři vnořené cykly. Viz Výpis 1.

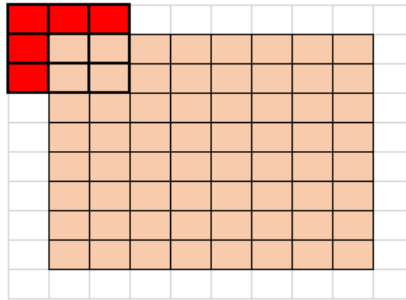
```
1  for(int row = 0; row < rows; row++) {
2    for(int col = 0; col < columns; col++){
3       ...
4      for(int y = 0; y < kernelDimension; y++) {
5        for(int x = 0; x < kernelDimension; x++) {
6          ...
7         }}
8       ...
9   }}
```

Výpis 1: Vnořené cykly konvoluce.

S čistě naivní konvolucí se dá pracovat ale bohužel jen do doby, než začneme používat větší kernely. U kernelů šířky 21, 51, 101 a více pixelů si začneme všímat nezpracovaných hran, což není ideální. Z tohoto důvodu chceme začít problematické hrany a rohy řešit. Hlavním problémem u hran a rohů obrazu je, že nemusí být úplně jasné, s jakými hodnotami násobit hodnoty kernelu, když kernel částečně zasahuje mimo vstupní obraz. Viz Obrázek 2.

Možností, jak tento problém řešit, je několik[22]:

- Použít hodnotu středového pixelu pro všechny přesahující pixely.



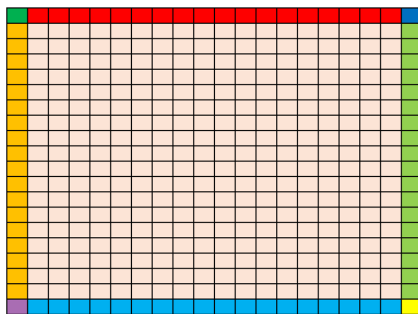
(a) Kernel 3×3

Obrázek 2: Ukázka přetečení kernelu přes hranu obrazu.

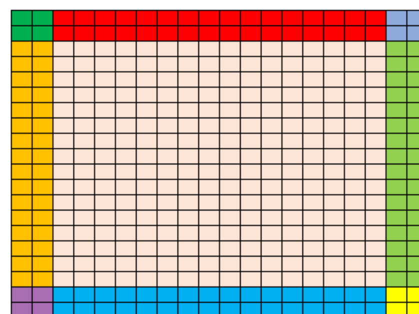
- Použít hodnotu nejbližšího validního pixelu hrany pro každý přesahující pixel
- Zabalit obraz. Tento způsob vezme hodnotu pixelu z protější hrany nebo rohu žádaného pixelu.
- Zrcadlit pixely přes hranu. Například použít namísto pixelu, který je dvě místa za hranou, hodnotu pixelu, která je dvě místa před hranou.

Bohužel, žádná z těchto možností není ideální pro všechny případy, neboť všechny mají nevýhody v určitých situacích. Například zabalení obrazu funguje rozumně do doby, dokud nemáme vstupní obraz s velkým kontrastem mezi levou a pravou stranou. U obrazu s tmavou levou stranou a světlou pravou stranou může nastat viditelný rozdíl v intenzitě na hranách a v rozích výsledného obrazu[22].

Dalším problémem, který bychom měli řešit, je výkonnost navrženého algoritmu. Mohli bychom napsat konvoluci tak, že budeme mít čtyři vnořené cykly pro celý vstupní obraz a uvnitř cyklů budeme mít spoustu podmínek, které budou hlídat a ošetřovat krajní pixely. Tento postup by byl ale značně pomalý, neboť by kód musel pro každý pixel vstupního obrazu zkontrolovat desítky podmínek, které by výrazně ovlivnily výkonnost. Vhodným řešením problému se zdá být rozdělení vstupního obrazu na několik částí. Konkrétně tedy na devět částí, viz Obrázek 3.



(a) Rozdělení pixelů pro kernel 3x3



(b) Rozdělení pixelů pro kernel 5x5

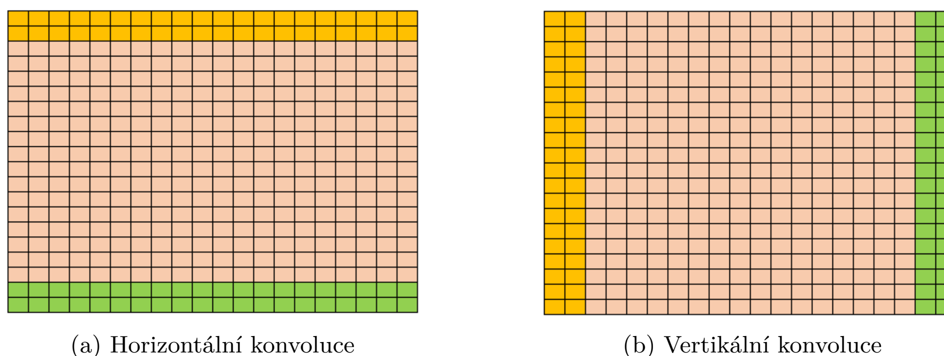
Obrázek 3: Rozdělení pixelů pro konvoluci s různými kernely.

Může se zdát, že toto řešení je poněkud krkolomné, neboť se nám délka původního kódu znásobila až devětkrát. Ve většině mnou testovaných případů ale toto řešení výkonnostně výrazně převyšuje původní řešení.

4.3.2 Separovatelná konvoluce

Základním rozdílem mezi separovatelnou a neseparovatelnou konvolucí je ten, že u separovatelné konvoluce procházíme obraz v podstatě dvakrát. V prvním kroku procházíme obraz horizontálně za pomocí horizontálního vektoru a dostaneme mezi výsledný obraz. Na tento mezi výsledný obraz pak aplikujeme vertikální konvoluci pomocí vertikálního vektoru a dostaneme výsledný obraz. Tento výsledný obraz je stejný jako výsledný obraz neseparovatelné konvoluce a navzdory tomu, že obraz procházíme dvakrát, je tento způsob výrazně rychlejší i pro větší kernely. Je to z toho důvodu, že pro každý pixel musíme udělat pouze $2k$ operací násobení a sčítání namísto k^2, jak je tomu u neseparovatelné konvoluce[21].

Dalším důvodem, proč je separovatelná konvoluce rychlejší je ten, že u separovatelné konvoluce nemusíme v algoritmu řešit rohy obrazu. Pro horizontální část konvoluce nám stačí rozdělit vstupní obraz na středovou část, levou hranu a pravou hranu. Pro vertikální část konvoluce nám stačí rozdělit vstupní obraz pouze na středovou část, horní hranu a dolní hranu. Viz Obrázek 4.



(a) Horizontální konvoluce

(b) Vertikální konvoluce

Obrázek 4: Horizontální/vertikální rozdělení pixelů pro separovatelnou konvoluci

Teď když máme jasno v rozdělení druhů konvoluce, můžeme se podívat na to, co to vlastně jsou ty separovatelné a neseparovatelné kernely.

4.4 Separovatelné kernely

Konvoluce je výpočetně velmi náročná operace, kde pro danou matici o šířce k potřebujeme k^2xy operací násobení a sčítání pro konvoluci obrazu o rozměrech xy. Právě z důvodu této náročnosti se hledala různá vylepšení, která by náročnost snížila. Z matematiky víme, že některá dvou dimenzionální jádra mohou být rozložena do dvou jedno dimenzionálních vektorů. Jeden vektor znázorňuje vertikální část a druhý horizontální část. Aplikací těchto dvou vektorů po sobě dosáhneme stejných výsledků jako aplikací celého kernelu ale s mnohem menší výpočetní

náročností. S využitím dvou jedno dimenzionální vektorů musíme použít pouze *2kxy* operací násobení a sčítání. Pro malé kernely velikosti 3×3 potřebujeme s celým kernelem udělat devět násobení a sčítání, kdežto se dvěma jedno dimenzionálními vektory nám stačí pouze šest. Tento rozdíl se nezdá moc velký pro kernely 3×3 nicméně pro větší kernely je již zvýšení výkonu velmi znatelné. Kernely, které můžeme rozdělit na dva jedno dimenzionální vektory nazýváme separovatelné. V praxi se snažíme určit, zdali je kernel separovatelný abychom mohli použít separovatelnou konvoluci, a tedy značně urychlit výpočet[21].

Konvoluční kernel je separovatelný, pokud konvoluční matice K má tu vlastnost, že může být vyjádřená jako výsledný produkt dvou vektorů u a v[21]. Pro matici 3×3:

$$K = v \otimes u = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \begin{bmatrix} u_1 & u_2 & u_3 \end{bmatrix} = \begin{bmatrix} v_1u_1 & v_1u_2 & v_1u_3 \\ v_2u_1 & v_2u_2 & v_2u_3 \\ v_3u_1 & v_3u_2 & v_3u_3 \end{bmatrix} \quad (8)$$

Vektor u je horizontální část kernelu a vektor v je vertikální část kernelu. Bohužel, jen malý zlomek kernelů je separovatelný. Z tohoto důvodu je třeba ověřit, zdali je vstupní kernel separovatelný nebo nikoliv. Zjevným ověřením separovatelnosti matice se zdá být vypočtení hodnosti matice vstupního kernelu. Má-li kernel (matice) hodnost 1, pak je kernel separovatelný[21].

4.5 FFT konvoluce

FFT konvoluce využívá skutečnosti, že násobení hodnot ve frekvenční doméně přímo odpovídá násobení hodnot v časové doméně[23]. Vstupní signál je přesunut do frekvenční domény za pomocí DFT (Diskrétní Fourierova Transformace), následně je násoben hodnotou kernelu, převedeného do frekvenční domény a na závěr je přesunut zpět do časové domény za použití inverzní DFT. Tato technika byla dostupná po dlouhou dobu nicméně kdysi nebyla pro konvoluci moc dobře použitelná. Je to z toho důvodu, že potřebný čas pro výpočet DFT byl větší než čas pro zpracování samotné konvoluce. To se změnilo v roce 1965 s příchodem Fast Fourier Transform (FFT) algoritmu[23]. Využitím FFT algoritmu k výpočtu DFT, docílíme urychlení výpočtu konvoluce ve frekvenční doméně, která dokonce předčí výpočet konvoluce v časové doméně. Výstupní data jsou pak stejná u obou algoritmů nicméně je výrazně omezen počet operací což ústí v efektivnější algoritmus. Z tohoto důvodu se FFT konvoluci také přezdívá vysoko rychlostní konvoluce. FFT konvoluce má ovšem i svá omezení. Výkonnost FFT konvoluce je téměř stejná nehledě na velikost použitého kernelu. Díky tomu FFT konvoluce vyniká při použité velkých kernelů nad ostatními implementacemi. Nicméně, při použití separovatelné konvoluce jsme schopni dosáhnout mnohem rychlejšího zpracování obrázku za použití menších kernelů, než je tomu u FFT konvoluce. Jelikož je pro konvoluci definována velká spousta malých kernelů, není pro nás FFT implementace vhodná. Z tohoto důvodu tato práce implementuje a testuje separovatelnou konvoluci. [23]

4.6 Popis paralelizace

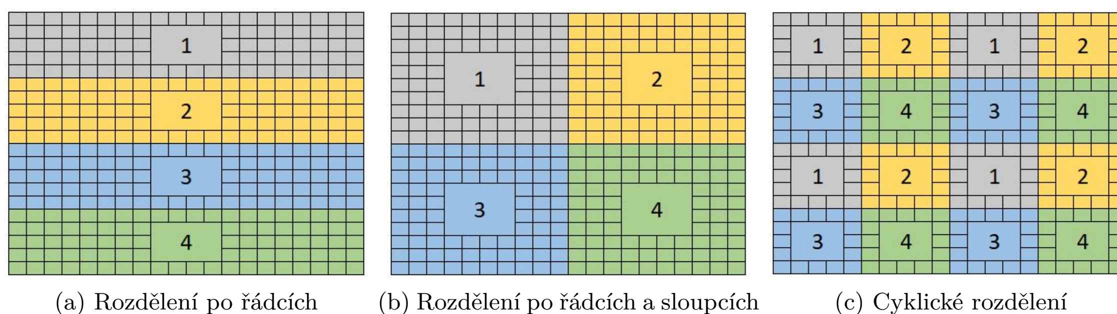
Můžeme vidět, že konvoluce není výpočetně nenáročná operace a že při velkých vstupních datech a při použití velkých kernelů může zpracování jednoho obrazu zabrat desítky vteřin i minut. Jelikož je konvoluce ve své podstatě jen opakované násobení a sčítání hodnot, dá se jí poměrně jednoduše paralelizovat. Musíme ovšem vyřešit pár problému jako je například distribuce dat mezi uzly a jejich komunikace. Principem paralelizace v této práci je rozdělení vstupního obrazu na n částí mezi n uzlů. Každý uzel pak zpracuje svou část obrazu a na závěr se tyto části spojí do jednoho výsledného obrazu. Postupným navyšováním výpočetních uzlů dosahujeme vyššího výkonu aplikace. Existuje ovšem hranice, kdy navýšení uzlů přestává mít vliv na výkony aplikace, a naopak může její výkon i snížit. Tuto problematiku popisuje Amdahlův a Gustavsonův zákon. Viz kapitola 3.

4.7 Problém distribuce dat mezi uzly

Tento problém klade otázku, jak nejlépe rozdělit data mezi výpočetní uzly, aby každý uzel dostal svou poměrnou část dat. V našem případě nám stačí brát v potaz jen náročnost komunikace mezi uzly. V jiných případech paralelních aplikací jako například zobrazení fraktálu musíme brát v potaz i náročnost zobrazení dat. V případě, že například píšeme program k zobrazení Mandelbrotovy množiny, je vhodné, aby každý uzel dostal poměrnou část Mandelbrotovy množiny k zobrazení. V tomto případě je stejně velká plocha výpočetně různě náročná v závislosti na tom, kterou část Mandelbrotovy množiny zobrazuje. Nechceme, aby dva uzly zobrazovaly 90% Mandelbrotovy množiny, zatímco ostatní uzly zobrazovaly jen zbylých 10%. Potřebujeme tedy docílit toho, aby každý uzel stejně náročnou část ke zpracování. U konvoluce se tento problém řešit nemusí, jelikož se jedná pouze o násobení a sčítání hodnot dvou matic, ať už jsou hodnoty jakékoliv. Pokud tedy dodržíme pravidlo, že každý uzel obdrží poměrnou část ze vstupního obrazu pak každý uzel bude mít srovnatelně náročnou úlohu ke zpracování. V Obrázku 5 můžeme vidět rozdělení dat mezi uzly následujícím způsobem: Rozdělení po řádcích, rozdělení po řádcích a sloupcích a cyklické rozdělení po řádcích a sloupcích. Z důvodu pravidelnosti výpočtů konvoluce je vidět, že z pohledu celkové výkonnosti je nejlepší rozdělení dat po řádcích. Oproti ostatním možnostem, rozdělení po řádcích přináší nejmenší komunikaci mezi uzly potřebou k paralelizaci a také nejefektivnější přístup do paměti[9].

4.7.1 Komunikace mezi uzly

Rozdělíme-li vstupní obrázek po řádcích mezi N uzlů, budeme mít ve výstupním obraze N-1 rušivých přechodů, které musíme řešit. Rušivými přechody rozumíme například nerovnoměrné vyhlazení konvolučním filtrem na sousedních hranách obrázků dvou uzlů. Uživatel si pak může všimnout rozdílů intenzit mezi částmi celkového obrázku. Přechody se ve výstupním obraze objevují z toho důvodu, že každý uzel počítá konvoluci pro svou část obrazu a hrany řeší pomocí jedné z technik uvedené výše. Sám uzel neví o jeho sousedech a nemá přístup k jejich datům,



(a) Rozdělení po řádcích (b) Rozdělení po řádcích a sloupcích (c) Cyklické rozdělení

Obrázek 5: Možnosti rozdělení dat mezi uzly.

aby mohl správně vypočíst konvoluci i na hranách se sousedními uzly. Je tedy nutné zařídit, aby každý uzel, který má nějakého souseda, obdržel část dat svého souseda potřebných ke konvoluci sousedících hran. Dodáme-li uzlům část dat od jejich sousedů, dosáhneme hladkých přechodů ve výstupním obraze. V této práci se tento problém řeší přeposíláním bufferů mezi uzly. Každý uzel očekává a rozesílá své hraniční buffery o určité velikosti svým sousedům. Výpočet velikosti bufferů se odvíjí od velikosti vstupního kernelu. Pro velikost kernelu 3×3 každý uzel odesílá buffer o velikosti jednoho řádku, pro kernel 5×5 to jsou dva řádky a podobně.

Pro příklad se vrátíme k Obrázku 5a. Zde je vidět, že uzel číslo jedna bude odesílat svůj spodní buffer uzlu číslo dva a zároveň bude očekávat horní buffer od uzlu číslo jedna. Obdobně je to i pro uzel číslo čtyři. Poněkud složitější situace nastane pro prostřední uzly číslo dva a tři. Tyto uzly pracují se čtyřmi buffery namísto dvou jako to bylo u uzlů číslo jedna a čtyři. Je to z toho důvodu, že uzel číslo dva očekává spodní buffer od uzlu číslo jedna a zároveň horní buffer od uzlu číslo tři. Následně musí odeslat svůj horní buffer uzlu číslo jedna a svůj spodní buffer uzlu číslo tři. Pro uzel číslo tři je toto chování identické.

Jelikož každý uzel musí odesílat i přijímat data, je zřejmé, že taková komunikace přinese nové problémy. Jedním z těchto problémů je uváznutí komunikace. Jedná se o situaci, kdy uzel A čeká na data uzlu B, než bude sám schopen odeslat data uzlu B, který na ně čeká. V této situaci oba uzly čekají na data toho druhého a nikdo data neposílá. V této práci je tento problém řešen tak, že všechny liché uzly data prvně odesílají svým sudým sousedům a následně čekají na data od svých sudých sousedních uzlů. Všechny sudé uzly pak prvně čekají na data od svých sousedních lichých uzlů a až je obdrží, odešlou data svým lichým sousedním uzlům.

4.8 Paralelní konvoluce

Základní principy konvoluce zůstávají při paralelní aplikaci zachovány. Každý uzel počítá konvoluci na své části vstupního obrázku. Nicméně jisté rozdíly nastávají při výpočtech konvoluce u sousedících hran. Musíme například zajistit, že u uzlu, který sousedí s jiným uzlem jen spodní hranou, bude uzel využívat data z příchozího bufferu sousedního uzlu namísto klasického zrcadlení. U neseparovatelné konvoluce se jedná pouze o čtení správného pixelu z bufferu. U sepa-

rovatelné konvoluce musíme ale při horizontální konvoluci projít a zpracovat i příchozí buffery. Pokud bychom tyto buffery nezpracovali, přišli bychom o důležitou informaci při zpracování hraničních pixelů u následné vertikální konvoluce.

Při paralelní konvoluci je také důležité dodržet pravidlo dělitelnosti počtem uzlů. Při rozdělování obrázku mezi uzly musíme dbát na to, aby každý uzel obdržel určitý počet řádků. Z důvodu praktičnosti není možné, aby nějaký uzel obdržel například dvacet a půl řádku. Z tohoto důvodu je vhodné rozšířit vstupní obrázek o dodatečná data, aby se následně dal rozumně rozdělit mezi dostupné uzly. Obdržíme-li například vstupní obrázek velikosti 17×13 a máme-li dva dostupné uzly pro výpočet, je jasné, že 221 pixelů nelze nijak rozumně rozdělit na dvě části. Rozšíříme-li obrázek o jeden sloupec, dostaneme obrázek o rozměrech 18×13, což je 234 pixelů, které už umíme rozdělit mezi dva uzly. Toto řešení nám bude stačit pro dva uzly, nicméně při více uzlech zjistíme, že musíme zachovat pravidlo dělitelnosti jak pro řádky, tak i pro sloupce. Vždy je tedy důležité rozšířit obrázek tak aby sloupce i řádky byly dělitelné celkovým počtem uzlů.

Hodnoty pixelů obrázku, o které budeme obrázek rozšiřovat, by ovšem neměly být generovány náhodně. To by mohlo mít za následek ovlivnění výsledného obrazu. Vhodným řešení se nabízí rozkopírováni posledních hodnot pixelů pro každý řádek. Při rozšíření řádků jen zkopírujeme poslední řádek vstupního obrázku. Rozšířený obrázek pak standardně zpracujeme konvolucí a na závěr jej vrátíme do původních rozměrů.

5 Implementace konvoluce

V praktické části implementace se blíže podíváme na praktiky použité při programování konvoluce pro Fiji.

5.1 Tvorba pluginu

ImageJ nabízí velkou spoustu tutoriálů[8], které výrazně pomáhají s tvorbou pluginů. ImageJ také nabízí kostry pluginů, které je vhodné použit při tvorbě vlastního pluginu. Při tvorbě pluginu je vhodné dodržet určitou anotaci, aby Fiji rozpoznal, že se skutečně jedná o plugin. Anotace pluginu je vidět ve Výpise 2. Můžeme si všimnout, že náš plugin má čtyři vstupní parametry. První dva parametry jsou *LogService*, ktery se stará o výpisy uživateli, a *UIService*, ktery se stará o uživatelské prostředí. Následuje vstupní parametr *Dataset*, který obsahuje vstupní obrázek zvolený uživatelem. Na závěr máme k dispozici parametr *kernel*, který uživateli nabídne výběr z osmnácti různých kernelů pro konvoluci a pamatuje si uživatelem zvolený kernel.

```
1  @Plugin(type = Command.class, menuPath = "Plugins>Kernel")
2  public class ImageConvolution<T extends RealType<T>> implements Command, Previewable {
3     @Parameter
4     private LogService ls;
5     @Parameter
6     private UIService uiService;
7     @Parameter(type = ItemIO.INPUT)
8     private Dataset inputImg;
9     @Parameter(label = "Kernel", persist = false, style = NumberWidget.SLIDER_STYLE, min
          = "1", max = "18", stepSize = "1.0")
10    private double kernel;
```

Výpis 2: Anotace pluginu.

Abychom mohli plugin spustit potřebujeme ještě dopsat hlavní (main) funkci pluginu. Hlavní funkce se stará o spuštění Fiji, otevření vstupního obrázku a následně spuštění *run()* funkce pluginu. Ve Výpisu 3 je vidět ukázka hlavní funkce našeho pluginu.

```
1  public static void main(final String... args) throws Exception {
2     final ImageJ ij = new ImageJ();
3     ij.ui().showUI();
4     final File file = ij.ui().chooseFile(null, "open");
5     if (file != null) {
6        final Dataset dataset = ij.scifio().datasetIO().open(file.getPath());
7        ij.command().run(ImageConvolution.class, true);
```

Výpis 3: Hlavní funkce pluginu.

[8] ImageJ tutorial: A simple Maven project implementing an ImageJ command, https://github.com/imagej/example-imagej-command

Hlavní funkce je poměrně jednoduchá, nicméně *run()* už je zajímavější. V *run()* funkci si vyčteme dimenze vstupního obrázku (obvykle tiff) z hlavičky obrázku, které budeme potřebovat ke spuštění konvoluční funkce a převedení obrázku do formátu RAW. Obrázek převádíme do formátu RAW, protože naše implementace konvoluční funkce pracuje pouze s RAW obrázkem. Výpis 4 znázorňuje *run()* funkci pluginu. Na řádku 4 se pak řeší kontrola bitové hloubky obrázku, která musí být rovna osmi. Na řádku 8 kontrolujeme, jestli je vstupní obrázek kolekce obrázků nebo pouze jeden obrázek. Tato informace je důležitá při procházení vstupního obrázku a převádění do formátu RAW. U kolekce obrázků se navíc prochází i řezy, které u jednoho obrázku řešit nemusíme. Na řádku 29 se pak volá konvoluční funkce v *try/catch* bloku neboť vyhazuje různé výjimky. Detaily výjimek a chybějící části kódu v ukázkách je možné dohledat v přiloženém zdrojovém souboru.

```
1  public void run() {
2     final Img<T> image = (Img<T>)inputImg.getImgPlus();
3      System.out.println("numDimensions = " + image.numDimensions());
4      ...
5      RandomAccess<T> ra = image.randomAccess();
6      long dimensions[] = new long[image.numDimensions()];
7      image.dimensions(dimensions);
8      ...
9      int[] out = new int[(int)(dimensions[0] * dimensions[1] * zStack)];
10     int inc = 0;
11     if(isSingle){
12       for (int x = 0; x < dimensions[1]; x++) {
13          for (int y = 0; y < dimensions[0]; y++) {
14             ra.setPosition(x, 1);
15             ra.setPosition(y, 0);
16             T val = ra.get();
17             out[inc] = (int)val.getRealDouble();
18             inc++;
19     }}} else {
20       for (int c = 0; c < zStack; c++) {
21          for (int x = 0; x < dimensions[0]; x++) {
22             for (int y = 0; y < dimensions[1]; y++) {
23                ...
24                ra.setPosition(c, 2);
25                ...
26     }}}}
27     saveImgStack(out, "tmp.raw");
28    try {
29       convolve("tmp.raw", (int)dimensions[0], (int)dimensions[1], zStack, (int)kernel);
30    } catch ...
```

Výpis 4: Run funkce pluginu.

5.2 Ošetření vstupů konvoluční funkce

Abychom mohli program spustit, potřebujeme mu předložit vstupní parametry. Parametrů je celkem pět a jsou to: cesta k obrázku, šířka, výška, počet řezů a vybraný kernel. Pro definování rozměrů pro vstupní obrázek jsou nastaveny tři vstupní parametry. Je to z toho důvodu, že program umí zpracovat i vstupní kolekci obrázků (3D stack). Pokud tedy program na vstupu obdrží kolekci obrázků, potřebujeme znát šířku a výšku každého obrázku v kolekci a zároveň i počet obrázků v kolekci. Program se spustí jen s validní cestou k obrázku, validními rozměry a správně zvoleným kernelem.

5.3 Paralelizace skrze OpenMPI

Tato část se věnuje přímo paralelizaci skrze OpenMPI[4]. Abychom mohli OpenMPI použít, musíme jej prvně inicializovat. Výpis 5 popisuje volání funkce *MPI.Init()*.

```
1  MPI.Init(args);
2  int world_rank = MPI.COMM_WORLD.getRank();
3  int world_size = MPI.COMM_WORLD.getSize();
```

Výpis 5: Inicializace OpenMPI.

Mimo inicializaci OpenMPI je vhodné zavolat i funkce *getRank()* a *getSize()*. Funkce *getRank()* vrací identifikátor konkrétního uzlu. Každý dostupný uzel dostane svůj identifikátor začínajíc od nuly. Funkce *getSize()* vrací celkový počet dostupných uzlů v systému. Díky těmto dvěma funkcím máme jasný přehled o počtu dostupných uzlů a také je umíme rozlišit. Od této chvíle, jakýkoliv kód, který napíšeme, se provede všemi dostupnými uzly. V určitých situacích toto chování není žádané. Například budeme-li chtít vrátit uživateli nějakou zprávu, pak se tato zpráva vypíše přesně tolikrát, kolik je dostupných uzlů. Obdobně ne vždy chceme provést stejnou operaci všemi uzly. Například není nutné, aby každý uzel rozšířil obrázek tak aby byl rozdělitelný mezi ostatní uzly. Tomuto chování můžeme jednoduše zabránit obalením příkazů do podmínky *if*. Viz Výpis 6.

```
1  if(world_rank == 0)
2    System.out.println("Hello world!");
```

Výpis 6: Omezení počtu volání příkazů.

Tímto jednoduchým obalením zajistíme, že se zpráva *Hello World!* vypíše pouze uzlem číslo nula.

Máme-li inicializované MPI, můžeme pokračovat dále. Nyní je vhodné si vypočíst overlay neboli přesah kernelu. Overlay počítáme, protože potřebujeme vědět, jak moc bude kernel přesahovat obrázek vystředíme-li kernel na nějaký hraniční pixel. Overlay budeme obecně v kódu využívat skoro všude neboť je potřebný pro inicializaci bufferů, pro výpočet dat, která budou v bufferech, v samotné konvoluci a podobně. Vypočíst overlay je triviální operace, jelikož kernel

je vždy lichý a symetrický. Stačí nám vzít velikost řádku kernelu a vydělit dvěma. Dále je pak vhodné vypočíst, jak moc musíme rozšířit vstupní obrázek, aby byl rozumně dělitelný počtem dostupných uzlů. Výpočet probíhá na bázi zbytku po dělení počtu řádku/sloupců a počtu dostupných uzlů. Tento zbytek pak odečteme od počtu dostupných uzlů a připočítáme k celkovému počtu řádků/sloupců. Viz Výpis 7.

```
1  int overlay = kernel.length / 2;
2  if(rows % world_size != 0)
3     padRow = rows + (world_size - (rows % world_size));
4  if(columns % world_size != 0)
5     padCol = columns + (world_size - (columns % world_size));
```

Výpis 7: Přetečení kernelu a rozšíření obrázku.

Obdržíme-li kolekci obrázků, je nutné zaručit, aby byl zpracován každý snímek v kolekci. Kód níže bude stejný pro každý obrázek v kolekci. Díky tomu můžeme projít celou vstupní kolekci obrázků *for* cyklem. Jediným problémem, co zbývá vyřešit, je pozice začátku každého obrázku v kolekci. Samotná implementace konvoluční funkce počítá pouze s obrázkem ve formátu RAW, a tedy nepracuje s hlavičkou obrázku. Jelikož jsou vstupní RAW data jen proud 8bitových čísel, není jasně dané, kde různé obrázky v kolekci začínají. Funkce *getArrFromRaw* ve Výpise 8 má za úkol vyextrahovat správný snímek z kolekce obrázků a přetypovat jej na IntBuffer, který pak vrátí. Funkce *padImage* je popsána níže a ukázána ve Výpise 9. Řádky 1–4 znázorňují volání funkce pro extrakci obrázku z kolekce obrázků. Řádky 5–14 pak znázorňují funkci pro extrakci obrázku.

```
1  for(int frameNum = 0; frameNum < numOfFrames; frameNum++){
2     if (world_rank == 0)
3        in = padImage(getArrFromRaw(inputFile,columns, rows,frameNum),columns,rows,
              world_size);
4     ...
5  public static IntBuffer getArrFromRaw(File f, int col, int row, int slice){
6     ...
7     IntBuffer out = MPI.newIntBuffer(col*row);
8     int slicePos = slice*col*row;
9    for(int i = 0; i < col*row; i++){
10      int i2 = (fileContent[slicePos + i] & 0xFF);
11      out.put(i,i2);
12   }
13   return out;
14 }
```

Výpis 8: Extrakce obrázku ze souboru.

Abychom mohli přejít k rozdělení vstupního obrázku mezi dostupné uzly, potřebujeme si ještě vstupní obrázek rozšířit. Funkce ve Výpise 9 tento proces popisuje. Nejdříve je nutné vytvořit

si buffer pro již rozšířený obrázek. Následně projdeme všechny řádky a doplníme hodnoty pro všechny přesahující řádky a sloupce posledními dostupnými hodnotami původního obrázku.

```
1  IntBuffer out = MPI.newIntBuffer(padRows * padCols);
2  for(int row = 0; row < padRows; row++){
3    for(int col = 0; col < padCols; col++){
4     if(row >= rows) {
5       if(col >= columns)
6         out.put((row * padCols) + col, inputArray.get(((rows-1) * columns) + columns -
              1));
7       else
8         out.put((row * padCols) + col,inputArray.get(((rows-1) * columns) + col));
9     } else {
10      if(col >= columns)
11        out.put((row * padCols) + col, inputArray.get((row * (columns)) + columns - 1)
            );
12      else
13        out.put((row * padCols) + col,inputArray.get((row * columns) + col));
14    }}}
```

Výpis 9: Funkce rozšiřující obrázek.

Máme-li rozšířený obrázek můžeme přejít k funkci *MPI.scatter()*. Tato funkce rozdělí vstupní obrázek rovnoměrně mezi dostupné uzly. Funkce má poměrně dost argumentů, které se dělí na pár částí. První část obsahuje argumenty pro vstupní buffer, druhá část pro výstupní buffer a poslední část specifikuje hlavní uzel. Prvním argumentem je vstupní buffer *in*, tento buffer obsahuje vstupní, již rozšířený obrázek. Druhý argument specifikuje počet elementů pro každý uzel. Tento argument říká funkci *MPI.scatter()*, kolik elementů (v našem případě pixelů) dostane každý uzel. Třetím argumentem je *MPI.INT*, který v podstatě jen říká, že každý element ve vstupním bufferu bude typu integer. Čtvrtým argumentem je výstupní buffer *out*. Tento buffer je specifický pro každý uzel a po skončení volání funkce *MPI.scatter()* bude obsahovat daný počet pixelů specifikovaný pátým argumentem. Pátý argument specifikuje, kolik pixelů bude ve výstupním bufferu. Šestým argumentem opět definujeme typ elementů ve výstupním bufferu. Poslední argument specifikuje, který uzel bude takzvaný root. Root obdrží první část obrázku. Viz Výpis 10, kde na řádku 1 vidíme výpočet počtu pixelů, které zpracuje každý uzel a řádky 2–3 znázorňují samotné volání funkce *MPI.scatter()*.

```
1  int elem = (padCol * padRow)/world_size;
2  MPI.COMM_WORLD.scatter(in, elem, MPI.INT, out, elem, MPI.INT, 0);
```

Výpis 10: Rozdělení obrázku mezi uzly.

Po zavolání funkce *MPI.scatter()* bude mít každý dostupný uzel svou rovnoměrnou část obrazu. V předchozích kapitolách jsme ale zjistili, že pro hladké provedení konvoluce musí mít

každý uzel ještě část obrazu svého předchůdce a následníka. Podíváme se tedy na řešení komunikace mezi uzly, která bude sloužit k přeposílání bufferů. K tomuto účelu budeme využívat MPI funkce *MPI.send()* a *MPI.recv()*. Tyto funkce jsou párové a tedy pokud pomocí *MPI.send()* pošleme zprávu jinému uzlu, musíme na cílovém uzlu zavolat *MPI.recv()*. Obě funkce mají řadu argumentů, které si teď popíšeme začínajíc *MPI.send()*. Argumenty této funkce jsou v následujícím pořadí: buffer k odeslání, počet elementů k odeslání, typ elementů k odeslání, ID cílového uzlu a ID zprávy. *MPI.recv()* má svým způsobem identické argumenty, jen si musíme pohlídat abychom data, která chceme udržet, uložili do bufferu pro přijatá data. Dále musíme zařídit, aby ID uzlu odpovídalo uzlu, který data odesílá a aby ID zprávy odpovídalo odeslané zprávě.

Samostatný proces komunikace je rozdělen do tří částí z důvodu rozdělení uzlů do třech kategorii. V první části řešíme problematiku prvního uzlu. Tento uzel dostane vrchní část obrazu a díky tomu musí rozeslat jen svou spodní hranu uzlu pod ním. Obdobně musí obdržet pouze vrchní hranu uzlu přímo pod ním. Horní buffery v tomto případě tedy vůbec neřešíme. Viz Výpis 11, kde první vidíme výpočet pozice prvního řádku bufferu v obrázku a následně inicializaci bufferu. Řádky 5–6 pak znázorňují naplnění bufferu příslušnými řádky. Řádky 9–17 pak řeší samotnou komunikaci. Můžeme si všimnout, že komunikace je řešena pouze v případě, že je v systému dostupných více uzlů.

```
1  if(world_rank == 0){
2     int lastRowIndex = ((elements_per_proc/padCol) - overlay) * padCol;
3     IntBuffer sendBuf = MPI.newIntBuffer(padCol*overlay);
4     int x = 0;
5     for(int i = lastRowIndex; i < lastRowIndex+(padCol*overlay); i++){
6        sendBuf.put(x,out.get(i));
7        x++;
8     }
9     if(world_size > 1){
10       MPI.COMM_WORLD.recv(recvBufDown, padCol*overlay, MPI.INT, world_rank+1, MSG_ID);
11       MPI.COMM_WORLD.send(sendBuf, padCol*overlay, MPI.INT, world_rank+1, MSG_ID);
12       recvBufUp = null;
13    }
14    else{
15       recvBufUp = null;
16       recvBufDown = null;
17    }
```

Výpis 11: Komunikace kořenového uzlu.

Ve druhé kategorii řešíme poslední uzel. Obdobně jako u prvního uzlu, nemusíme u posledního uzlu řešit spodní hranu. Potřebujeme tedy pouze řešit komunikaci s předchozím uzlem. Viz Výpis 12, kde první řešíme inicializace bufferů a jejich naplnění. Jelikož se jedná o vrchní buffer, nemusíme počítat jeho pozici v obrázku. Řádky 6–15 pak řeší komunikaci mezi uzly. Abychom předešli uváznutí musíme řešit i poslední uzel, neboť nevíme, jestli bude lichý nebo sudý.

```
if(world_rank == world_size - 1){
   IntBuffer sendBuf = MPI.newIntBuffer(padCol*overlay);
   for(int i = 0; i < padCol*overlay; i++){
      sendBuf.put(out.get(i));
   }
   if(world_size > 1) {
      if(world_rank % 2 == 0){
         MPI.COMM_WORLD.recv(recvBufUp, padCol*overlay, MPI.INT, world_rank-1, MSG_ID);
         MPI.COMM_WORLD.send(sendBuf, padCol*overlay, MPI.INT, world_rank-1, MSG_ID);

      }else {
         MPI.COMM_WORLD.send(sendBuf, padCol*overlay, MPI.INT, world_rank-1, MSG_ID);
         MPI.COMM_WORLD.recv(recvBufUp, padCol*overlay, MPI.INT, world_rank-1, MSG_ID);
      }
      recvBufDown = null;
   }}
```

Výpis 12: Komunikace posledního uzlu.

Ve třetí kategorii je řešena komunikace všech prostředních uzlů. V této kategorii musí každý uzel řešit jak rozesílání své horní a spodní hrany, tak obdržení hran přilehlých uzlů. Dále zde musíme řešit i uváznutí komunikace. Toto je zde řešeno tak, že liché uzly prvně odesílají data svým sudým sousedům a následně od nich data přijímají. Viz Výpis 13, kde opět první řešíme inicializaci a pozici prvního řádku spodního bufferu. Na řádku 4 se pak řeší naplnění spodního a vrchního bufferu příslušnými řádky viz Výpis 8 a 9. Řádky 5–15 pak řeší samotnou komunikaci. Opět si musíme hlídat uváznutí, a proto liché uzly prvně data odesílají a až pak data přijímají. Sudé uzly se chovají obdobně jen obráceně.

```
int lastRowIndex = ((elements_per_proc/padCol) - overlay) * Col;
IntBuffer sendBufUp = MPI.newIntBuffer(padCol*overlay);
IntBuffer sendBufDown = MPI.newIntBuffer(padCol*overlay);
...
if(world_rank % 2 == 0) {
   MPI.COMM_WORLD.recv(recvBufUp, padCol*overlay, MPI.INT, world_rank-1, MSG_ID);
   MPI.COMM_WORLD.recv(recvBufDown, padCol*overlay, MPI.INT, world_rank+1, MSG_ID);
   MPI.COMM_WORLD.send(sendBufUp, padCol*overlay, MPI.INT, world_rank-1, MSG_ID);
   MPI.COMM_WORLD.send(sendBufDown, padCol*overlay, MPI.INT, world_rank+1, MSG_ID);
} else {
   MPI.COMM_WORLD.send(sendBufUp, padCol*overlay, MPI.INT, world_rank-1, MSG_ID);
   MPI.COMM_WORLD.send(sendBufDown, padCol*overlay, MPI.INT, world_rank+1, MSG_ID);
   MPI.COMM_WORLD.recv(recvBufUp, padCol*overlay, MPI.INT, world_rank-1, MSG_ID);
   MPI.COMM_WORLD.recv(recvBufDown, padCol*overlay, MPI.INT, world_rank+1, MSG_ID);
```

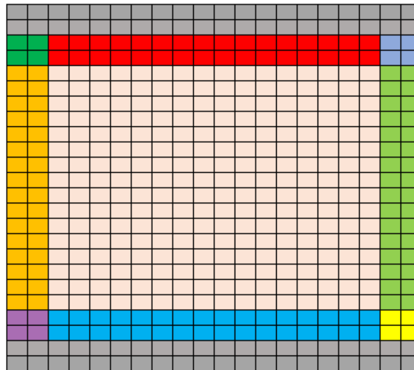
Výpis 13: Komunikace prostředních uzlů.

Nyní by už každý uzel měl mít všechny potřebné informace ke zpracování konvoluce nad svou částí dat. V závislosti na tom, jestli je jádro separovatelné nebo nikoliv, máme na výběr ze dvou volání. Při separovatelném jádru je lepší volat separovatelnou konvoluci, jelikož její výpočet je rychlejší než u naivní konvoluce. Argumenty pro volání obou konvolucí jsou téměř stejné. Separovatelná konvoluce se liší pouze argumentem *denom*, který je odmocninou celkového denominátoru kernelu. Výpis 14 ukazuje volání obou možností, kde *twoDconvolutionFast()* je separovatelná konvoluce a *twoDconvolution()* je optimalizovaná naivní konvoluce.

```
1 if(isSeparable)
2    out = twoDconvolutionFast(out, padCol, padRow/world_size, kernel, Math.sqrt(denom),
         recvBufUp, recvBufDown);
3 else
4    out = twoDconvolution(out, padCol, padRow/world_size, kernel, denom, recvBufUp,
        recvBufDown);
```

Výpis 14: Volání konvolucí.

Při standardní konvoluci se obrázek dělí na devět částí, kde se každá část řeší zvlášť. Viz Obrázek 6. Šedé části znázorňují přiložené buffery, které se nijak nezpracovávají pouze z nich čerpáme data při výpočtu konvoluce pro spodní/horní hrany.



(a) Rozdělení pixelů

Obrázek 6: Ukázka rozdělení pixelů pro standardní konvoluci.

Ve Výpise 15 je vidět kód výpočtu konvoluce nad střední části obrázku. Nejdříve vidíme již známé čtyři *for* cykly potřebné k procházení obrázku a kernelu. Všimněme si, že *for* cykly jsou upravené tak aby procházely jen střed obrázku. Následně si musíme inicializovat akumulátor, který bude akumulovat mezivýsledky násobení pixelů a hodnot kernelu po čemž pak bude obsahovat výslednou hodnotu pixelu. Na řádku 6 pak počítáme pozici pixelu, na který zrovna ukazuje pozice hodnoty v kernelu. Na závěr nesmíme zapomenout na použití denominátoru a normalizaci výsledné hodnoty pixelu.

```
1 for(int row = kenDim/2; row < rows-(kenDim/2); row++) {
2   for(int col = kenDim/2; col < columns-(kenDim/2); col++){
```

```
3        acc = 0;
4        for(int y = 0; y < kenDim; y++) {
5           for(int x = 0; x < kenDim; x++) {
6              positionIndex = ((row + y - (kenDim / 2)) * columns) + (col + x - (kenDim /
                    2));
7              acc += kernel[x][y] * inputArray.get(positionIndex);
8           }}
9        if(denom != 0)
10          acc /= denom;
11       if(acc > 255)
12          acc = 255;
13       if(acc < 0)
14          acc = 0;
15       ret.put((row * columns) + col,acc);
16    }}
```

Výpis 15: Výpočet konvoluce nad středem obrazu.

Podobný postup se ve své podstatě opakuje pro každou ze zbylých částí. Pro každou část jsou nutné čtyři *for* cykly, které danou část projdou a vynásobí hodnoty pixelů s hodnotami kernelu. Liší se ovšem v tom, že musíme řešit přesahy kernelu mimo hrany obrazu. V této práci se tento problém řeší při hranách na stranách obrazu zrcadlením a při vrchních a spodních hranách hledáním hodnot v bufferu. Ve Výpise 16 je vidět řešení horní hrany obrazu, kde máme opět čtyři *for* cykly, které tentokrát procházejí jen pixely horní hrany. Opět je nutné inicializovat akumulátor a vypočítat pozici pixelu, kam ukazuje kernel. Po vypočítání pozice pixelu následuje řešení hodnot hraničních pixelů. Jak je vidět na řádcích 7–8, pokud pozice pixelů ukazuje dovnitř obrázku (není záporná) použijeme tuto hodnotu. Pokud je pozice pixelů záporná, a tedy ukazuje mimo obraz, musíme jí buďto zrcadlit anebo hodnotu vytáhneme přímo z bufferu. Na závěr nesmíme zapomenout na použití denominátoru a normalizaci výsledné hodnoty pixelu.

```
1  for(int row = 0; row < kenDim/2; row++){
2     for(int col = kenDim/2; col < columns-kenDim/2; col++) {
3        acc = 0;
4        for(int y = 0; y < kenDim; y++) {
5           for(int x = 0; x < kenDim; x++) {
6              positionIndex = ((row + y - (kenDim / 2)) * columns) + (col + x - (kenDim /
                    2));
7              if(positionIndex >= 0)
8                 acc += kernel[x][y] * inputArray.get(positionIndex);
9              else {
10                if(up == null){
11                   mirrorX = (kenDim -1) - x;
12                   mirrorY = (kenDim -1) - y;
```

```
13                  mirrorPositionIndex = ((row + mirrorY - (kenDim / 2)) * columns) + (
                        col + mirrorX - (kenDim / 2));
14                  acc += kernel[x][y] * inputArray.get(mirrorPositionIndex);
15              }
16              else
17                  acc += kernel[x][y] * up.get(positionIndex + (columns * (kenDim/2)));
18          }}}
19      ...
20  }}
```

Výpis 16: Výpočet konvoluce horní hrany obrazu.

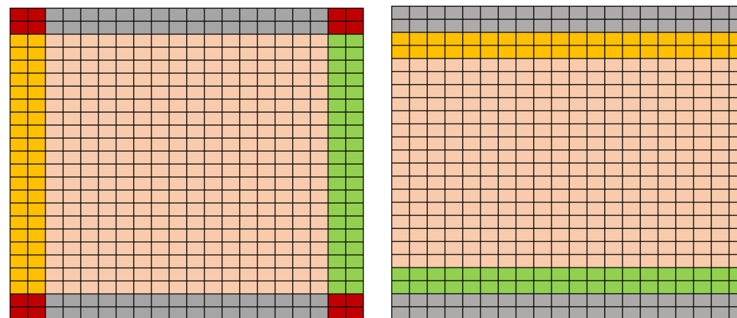
Ve Výpise 17 je vidět řešení levé hrany, kde prvně procházíme *for* cykly pouze pixely levé hrany, inicializujeme akumulátor a počítáme pozici pixelů. Následně pak řešíme hodnoty přesahující hranice obrázku. Na rozdíl od toho, jak to bylo u horní hrany, zde nemáme žádný buffer, ze kterého bychom mohli čerpat data. Z tohoto důvodu musíme všechny hodnoty, které jsou mimo obrázek zrcadlit.

```
1   for(int row = kenDim/2; row < rows-kenDim/2; row++){
2       for(int col = 0; col < kenDim/2; col++) {
3           acc = 0;
4           for(int y = 0; y < kenDim; y++) {
5               for(int x = 0; x < kenDim; x++) {
6                   positionIndex = ((row + y - (kenDim / 2)) * columns) + (col + x - (kenDim /
                        2));
7                   if(positionIndex >= 0){
8                       if(positionIndex >= (row + (y - kenDim/2)) * columns)
9                           acc += kernel[x][y] * inputArray.get(positionIndex);
10                       else{
11                          mirrorX = (kenDim -1) - x;
12                          mirrorY = (kenDim -1) - y;
13                          mirrorPositionIndex = ((row + mirrorY - (kenDim / 2)) * columns) + (
                                col + mirrorX - (kenDim / 2));
14                          acc += kernel[x][y] * inputArray.get(mirrorPositionIndex);
15                      }}
16                  else{
17                      mirrorX = (kenDim -1) - x;
18                      mirrorY = (kenDim -1) - y;
19                      mirrorPositionIndex = ((row + mirrorY - (kenDim / 2)) * columns) + (col +
                            mirrorX - (kenDim / 2));
20                      acc += kernel[x][y] * inputArray.get(mirrorPositionIndex);
21                  }}}
22          ...
23  }}
```

Výpis 17: Výpočet konvoluce levé hrany obrazu.

Pravá a spodní hrana se řeší podobně jako levá a vrchní jen obráceně. Rohy se řeší kombinací výše zmíněných řešení.

Nyní přecházíme k separovatelné konvoluci. Implementace tohoto druhu konvoluce využívá lehce pozměněné rozdělení pixelů. Další změna v této konvoluci oproti naivní konvoluci je ta, že separovatelná konvoluce je složená ze dvou konvolucí. Z tohoto důvodu je vhodné mít dvě funkce a volat je po sobě. Poslední důležitou změnou je fakt, že při separovatelné konvoluci v horizontálním směru musíme zpracovat i buffery abychom neztratili důležité informace potřebné při vertikální konvoluci. Obrázek 7a a 7b znázorňuje rozdělení pixelů při separovatelné konvoluci v horizontálním a pak ve vertikálním směru.



(a) Rozdělení pro horizontální část (b) Rozdělení pro vertikální část

Obrázek 7: Ukázka rozdělení pixelů pro separovatelnou konvoluci.

Při horizontální konvoluci se pak v kódu namísto celého kernelu prochází jen horizontální vektor daného kernelu viz Výpis 18. Na řádku 8 se pak aplikuje denominátor a normalizujeme hodnotu výsledného pixelu. Obdobný kód se pak spouští pro středy horního a dolního bufferu.

```
1 for(int row = 0; row < rows; row++) {
2   for(int col = kenDim/2; col < columns-(kenDim/2); col++){
3     acc = 0;
4     for(int x = 0; x < kenDim; x++) {
5       positionIndex = (row * columns) + (col + x - (kenDim / 2));
6       acc += vectorX[x] * inputArray.get(positionIndex);
7     }
8       ...
```

Výpis 18: Střed separovatelné konvoluce v horizontálním směru.

V horizontální konvoluci se pak ještě ošetřují hrany. Kód ve Výpise 19 ukazuje řešení levé hrany zrcadlením. Obdobný kód je pak použit pro levé hrany horního a dolního bufferu. Převrácený kód se pak obdobně použije na pravé hrany obrázku a pravé hrany horního a dolního bufferu.

```
1 for(int row = 0; row < rows; row++){
2   for(int col = 0; col < kenDim/2; col++) {
```

```
3        acc = 0;
4        for(int x = 0; x < kenDim; x++) {
5          positionIndex = (row * columns) + (col + x - (kenDim / 2));
6          if(positionIndex >= 0){
7            if(positionIndex >= (row * columns))
8              acc += vectorX[x] * inputArray.get(positionIndex);
9            else{
10             mirrorX = (kenDim -1) - x;
11             mirrorPositionIndex = (row * columns) + (col + mirrorX - (kenDim / 2));
12             acc += vectorX[x] * inputArray.get(mirrorPositionIndex);
13           }}
14         else{
15           mirrorX = (kenDim -1) - x;
16           mirrorPositionIndex = (row * columns) + (col + mirrorX - (kenDim / 2));
17           acc += vectorX[x] * inputArray.get(mirrorPositionIndex);
18         ...
```

Výpis 19: Levá hrana separovatelné konvoluce v horizontálním směru.

Po zpracování celého obrazu a bufferů horizontální konvolucí je automaticky volána vertikální separovatelná konvoluce. Tato funkce obdrží již data zpracovaná horizontální konvolucí a dále je zpracovává ve vertikálním směru. V této funkci již buffery zpracovávat nemusíme, a tak nám stačí zpracovat pouze střed obrázku, horní hranu a dolní hranu. Výpočet středu vertikální konvoluce je téměř totožný s horizontálním. Jediným rozdílem je, že se aplikuje vertikální vektor kernelu ve vertikálním směru. Z tohoto důvodu se rovnou přesuneme na řešení horní hrany. Viz Výpis 20.

```
1  for(int row = 0; row < kenDim/2; row++){
2    for(int col = 0; col < columns; col++) {
3      acc = 0;
4      for(int y = 0; y < kenDim; y++) {
5        positionIndex = ((row + y - (kenDim / 2)) * columns) + col;
6        if(positionIndex >= 0)
7          acc += vectorY[y] * inputArray.get(positionIndex);
8        else {
9          if(up == null){
10           mirrorY = (kenDim -1) - y;
11           mirrorPositionIndex = ((row + mirrorY - (kenDim / 2)) * columns) + col;
12           acc += vectorY[y] * inputArray.get(mirrorPositionIndex);
13         } else
14           acc += vectorY[y] * up.get(positionIndex + (columns * (kenDim/2)));
15         ...
```

Výpis 20: Horní hrana separovatelné konvoluce ve vertikálním směru.

Po skončení separovatelné nebo naivní konvoluce již má každý uzel zpracován svou část obrázku. Dalším krokem tedy je složení těchto částí do jednoho celku. K tomu nám slouží

funkce *MPI.gather()*, která je volána na každém uzlu sjednocuje jednotlivé části každého uzlu do jednoho celku. *MPI.gather()* má šest argumentů v následujícím pořadí: Odesílaný buffer, počet elementů v odesílaném bufferu, typ každého elementu v bufferu, přijímající buffer, počet elementů zaslaných do přijímajícího bufferu, typ každého elementu a kořenový uzel nula. Volání funkce je ukázáno ve Výpise 21.

```
1 MPI.COMM_WORLD.gather(out, elem, MPI.INT, in, elem, MPI.INT, 0);
```

Výpis 21: Složení obrázku do jednoho celku.

Nyní bychom měli mít složený obrázek v jednom jediném bufferu. Nesmíme ještě zapomenout na oříznutí obrázku zpět do jeho původních rozměrů a následně jej uložit. Pokud byla na vstupu kolekce obrázků, uložíme mezivýsledek a celý proces opakujeme s následujícím obrázkem v kolekci.

5.4 Instalace pluginu

Instalace a spuštění pluginu jsou poměrně přímočaré. V příloze této práce jsou dva soubory, které budeme potřebovat. Jedná se o *ImageConvolution.jar* a *mpi.jar*. Je nutné si dát pozor na použitou verzi *mpi.jar*, neboť je platformě závislé a je dané pro konkrétní instalaci konkrétní verze OpenMPI. Dále budeme potřebovat aplikaci Fiji a mít nainstalované knihovny OpenMPI. Samotná instalace pak znamená pouhé vložení souborů *ImageConvolution.jar* a *mpi.jar* do složky plugins, která se nachází v instalační složce Fiji. Po spuštění aplikace by si pak Fiji mělo samo dohledat a nainstalovat nové pluginy. Jedním z těchto pluginů bude náš ImageConvolution plugin, který by se nachází v menu „Plugins“ v hlavní liště Fiji. Pro spuštění pluginu je nutné mít předem otevřený obrázek, na který chceme aplikovat konvoluční filtr. Dále je vhodné mít otevřenou konzoli ImageJ, kde se vypisuje status konvoluce a popisky filtrů zvolené uživatelem. Máme-li otevřený obrázek a konzoli, můžeme v menu Plugins zvolit ImageConvolution, který nám nabídne možnost vybrání konvolučního filtru. Po potvrzení výběru se konvoluce s daným filtrem spustí a výsledný obrázek se uloží pod názvem *result.raw* do instalační složky Fiji.

5.4.1 Využití pluginu v ImageJ makrech

Plugin je navržen v souladu s doporučenými postupy pro vývoj ImageJ2 pluginů a je tak i zpětně kompatibilní s ImageJ a jeho makrojazykem. Ve výpise 22 je možno vidět ukázku makro skriptu, který otevře demonstrační obrázek Cell Colony a na něj aplikuje námi vyvinutý plugin pro konvoluci s velikosti kernelu 5×5.

```
1 run("Cell Colony (31K)");
2 run("ImageConvolution", "inputimg=Cell_Colony.jpg kernel=5.0");
```

Výpis 22: Ukázka volání pluginu v ImageJ makrojazyku.

6 Testování

Tato kapitola popisuje testovací prostředí, jak testování probíhalo a jakých výsledků bylo dosaženo. Výsledky jsou pak znázorněny tabulkami a grafy porovnávající výkonnost implementace separovatelné konvoluce s naivní implementací. Pro zajímavost je porovnána i FFT (Fast Fourier Transform) implementace.

6.1 Popis vývojového prostředí

Práce je vyvíjena na superpočítači z IT4Innovation centra, konkrétně pak na stroji Salamon. Salamon je složen z 1008 výpočetních uzlů ze kterých je 576 klasických výpočetních uzlů a 432 akcelerovaných výpočetních uzlů. Každý z těchto uzlů představuje výkonný počítač s 24 jádry (dva dvanácti jádrové Intel Xeon procesory) a 128 GB RAM. Uzly jsou spojeny vysokorychlostní InfinityBand a Ethernetovou sítí. Uzly sdílejí 500 TB diskové paměti pro uložení dat a programů uživatelů. Program je psán v jazyce Java 8 a pro spuštění je použit virtuální stroj Java 1.9.0+181. V programu jsou využity funkce z knihovny OpenMPI verze 4.0.0-GCC-6.3.0-2.27.

6.2 Proces testování

Testy byly prováděny jak na testovacích obrázcích stažených z Fiji, tak na náhodně generovaných obrázcích. Testovací obrázky stažené z Fiji byly použity hlavně k zaručení správnosti konvoluční funkce. Na náhodně generovaných vstupních obrázcích by nemuselo být dobře vidět drobné chyby konvolučního algoritmu jako například nerovné přechody na hranách sousedních uzlů. Ve finální fázi testování se spustil stejný obrázek jak ve vlastní implementaci, tak v implementaci dostupné ve Fiji a porovnávalo se, jestli obě implementace vrací stejný výsledek při použití stejných parametrů.

Testování proběhlo na postupně se zvětšujících obrázcích. Využité obrázky byly postupně velikosti 1000×1000, 2000×2000, 3000×3000, 4200×4200, 5000×5000, 10000×10000. Dále se pro testování využívaly kernely různých velikostí. Jelikož konvoluce spoléhá jen na násobení a sčítání matic, nemá pro účely testování výkonnosti moc velký smysl vymýšlet kernely pro různé aplikace jako například zaostřování, rozpoznávání hran a podobně. Z tohoto důvodu byly jako kernely výhradně použity pouze generované Gaussovské filtry různých velikostí. Testovány byly následující velikosti kernelů: 3×3, 5×5, 7×7, 9×9, 11×11, 13×13, 15×15, 17×17, 19×19, 21×21, 51×51 a 101×101. Poslední proměnnou byl počet uzlů využitých na superpočítači. Testovaly se následující počty uzlů: 1, 2, 4, 8, 16, 32, 64. Testování více než 64 uzlů nemělo smysl, neboť už při 32 uzlech bylo ve většině případu vidět zpomalení aplikace z důvodu náročnosti komunikace mezi uzly.

Testování na superpočítači bylo řešeno pomocí skriptů, pro každou hodnotu počtu uzlů se spouštěly dva cykly, které prošly všemi kombinacemi velikosti kernelu a obrázků. Výsledné doby běhu jednotlivých kombinací se vypisovaly do výsledného textového souboru.

Při implementaci se v první řadě testovaly velmi malé obrázky s malými kernely, u kterých byly snadno rozpoznatelné chyby v implementaci. Jakmile byl kód odladěn na malých obrázcích, začaly se testovat větší, reálné obrázky s většími kernely, kde se dále dolaďovaly implementační chyby. Jakmile obrázky překročily hranici 1000×1000 pixelů, začaly se řešit velké kernely, kde se dolaďovaly problémy s přeposíláním bufferů, jemné detaily při přechodech mezi částmi počítanými jednotlivými uzly a také hrany obrázků. Po vyladění výše zmíněných chyb se přešlo k optimalizaci výkonnosti programu jako například implementace separovatelné konvoluce, odstranění redundantního kódu, ladění výpisů a podobně.

Abychom mohli přejít k porovnání výkonnosti této aplikace, je dobré si zvolit referenční bod. Jako první referenční bod byla zvolena naivní implementace konvoluce přímo z definovaných funkcí v ImageJ. Jelikož naivní implementace patří mezi nejpomalejší, zvolíme si jí jako nejpomalejší referenční bod. Obdobným postupem zvolíme nejrychlejší referenční bod, který je v našem případě implementace konvoluce přes FFT (Fast Fourier Transform). FFT implementace je mnohem rychlejší než standardní implementace konvoluce, jelikož vůbec neaplikuje násobení a sčítání hodnot kernelu s hodnotami obrázku. Konvoluce se řeší přímo ve frekvenčním spektru obrazu. Tato funkce se dá nalézt mezi funkcemi definovanými v ImageJ.

Jelikož FFT i naivní implementace nejsou paralelní a nedá se je v rozumné době paralelizovat, byly testovány pouze pro jeden uzel. Pro získání referenční hodnot to ale bude dostačující.

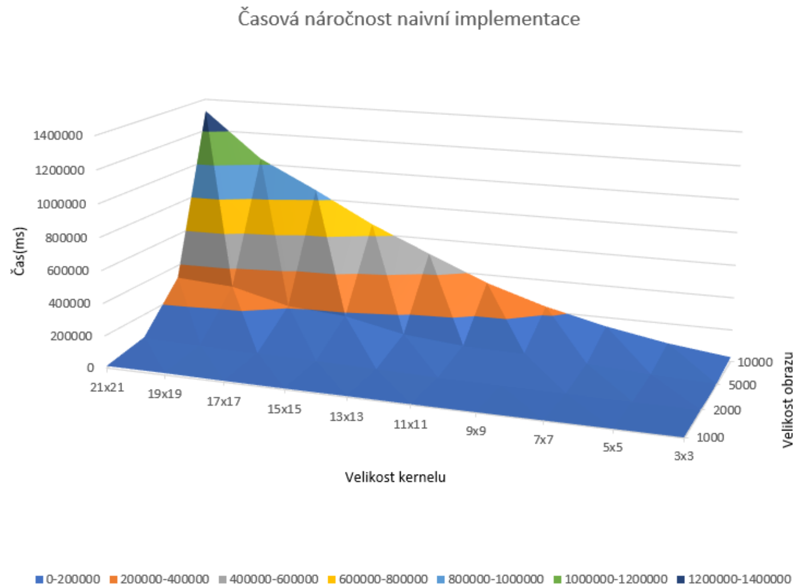
6.3 Naivní implementace

Jak již zbylo zmíněno výše v textu, základní naivní implementace je nejpomalejší. Tabulka 1 popisuje její výkonnost při měření s námi zvolenými parametry.

Tabulka 1: Výkonnost naivní implementace na jednom uzlu. Čas je udávaný v ms.

Kernel	1000 × 1000	2000 × 2000	5000 × 5000	10000 × 10000
3×3	379	1447	7002	29553
5×5	731	3091	21198	75895
7×7	1374	6691	43246	143345
9×9	2168	10733	66101	231447
11×11	4201	13677	72554	347840
13×13	5710	22457	105213	503377
15×15	5991	29647	182325	662769
17×17	7855	37922	209037	855574
19×19	11559	47168	298397	1035312
21×21	12576	57401	322030	1324387

Jak je vidět, doba výpočtu u naivní implementace rapidně roste při navýšení velikosti kernelu i vstupního obrázku. Graf v Obrázku 8 tyto hodnoty znázorňuje.



(a) Naivní implementace

Obrázek 8: Graf výkonnosti naivní implementace.

Z grafu lze vidět, že i při malých kernelech se můžeme u větších obrázků pohybovat v řádu několika minut a při použití velkých kernelů i v řádu desítek minut. Z tohoto důvodu se naivní implementace nevyužívá a využívají se rychlejší druhy implementací.

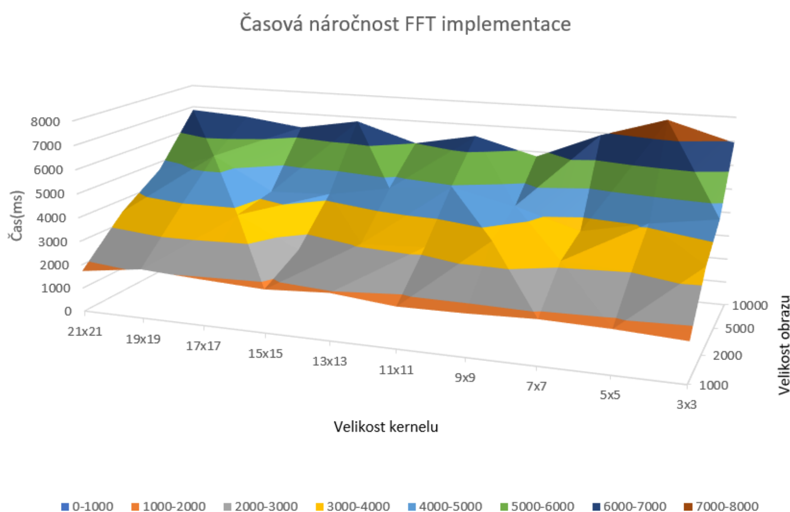
6.4 FFT implementace

FFT implementace je jedna z nejvýkonnějších implementací, které jsou zatím dostupné. Její výkonnost je obzvláště vidět u použití velkých kernelů. Při použití malých kernelů může ovšem zaostávat za jinými implementacemi. Naměřené hodnoty při použití FFT implementace jsou vypsány v Tabulce 2.

Tabulka 2: Výkonnost FFT implementace na jednom uzlu. Čas je udávaný v ms.

Kernel	1000 × 1000	2000 × 2000	5000 × 5000	10000 × 10000
3×3	1635	3492	4505	6945
5×5	1769	3030	4095	7730
7×7	1830	3156	3533	6907
9×9	1742	3713	4354	5796
11×11	1707	3210	4968	6510
13×13	1959	3074	4677	6003
15×15	1809	2557	4200	6817
17×17	1931	3810	4092	6370
19×19	2079	3830	5494	6695
21×21	1729	3475	4669	6853

Můžeme vidět, že FFT je výrazně rychlejší než naivní implementace, neboť i pro velké kernely a velké obrázky se pohybujeme v řádech vteřin namísto v řádech minut, jak bylo vidět u naivní implementace. V Obrázku 9 je graf znázorňující výkonnost FFT implementace. Zajímavostí této implementace je, že na velikosti kernelu příliš nezáleží, jelikož doba výpočtu se odvíjí od velikosti vstupního obrázku.



(a) FFT implementace

Obrázek 9: Graf výkonnosti FFT implementace.

6.5 Optimalizovaná naivní implementace

Máme-li stanové maximální a minimální referenční hranice výkonnosti, můžeme začít porovnávat vlastní implementaci. První se podíváme na naivní, optimalizovanou implementaci. Tato implementace se liší od typické naivní implementace tím, že rozděluje zpracovávaný obrázek do několika kategorii, které řeší samostatně. Toto relativně snadné řešení výrazně zrychluje dobu výpočtu. Viz Tabulka 3.

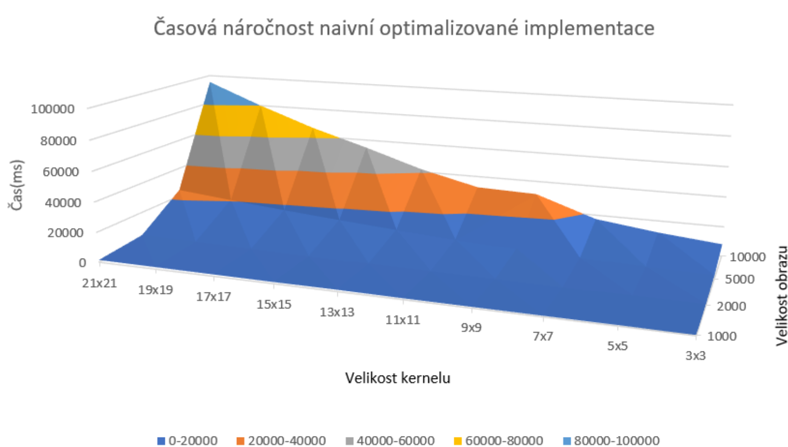
Obrázek 10 znázorňuje graf výkonnosti optimalizované, naivní implementace. Stále vidíme rapidní nárůst časové náročnosti při použití velký kernelů na velkých obrázcích, nicméně porovnáme-li tyto hodnoty s hodnotami typické naivní implementace, uvidíme masivní zrychlení. Můžeme si také všimnout, že tato implementace je rychlejší než FFT implementace při použití menších kernelů. Je to z toho důvodu, že FFT implementace nabírá své výkonnosti až při větších obrázcích a při použití velkých kernelů. Tento rozdíl je ještě výraznější při použití separovatelné konvoluce.

6.6 Separovatelná implementace

Při optimalizaci naivní implementace jsme mohli vidět výrazné snížení časové náročnosti. Použijeme-li stejnou optimalizaci na separovatelnou implementaci uvidíme dodateční snížení časové náročnosti

Tabulka 3: Výkonnost vlastní naivní implementace na jednom uzlu. Čas je udávaný v ms.

Kernel	1000 × 1000	2000 × 2000	5000 × 5000	10000 × 10000
3×3	311	741	2391	8143
5×5	557	914	3343	12029
7×7	628	1138	4690	17312
9×9	700	1433	6389	31076
11×11	790	1749	8443	32348
13×13	1075	2152	10992	41949
15×15	1016	2625	13849	54495
17×17	1149	3155	17023	66109
19×19	1306	3737	20517	79934
21×21	1477	4356	24354	95213



(a) Vlastní naivní implementace

Obrázek 10: Graf výkonnosti vlastní naivní implementace.

nosti. Tabulka 4 zobrazuje data naměřená při využití optimalizované separovatelné implementace. Obrázek 11 pak znázorňuje graf této implementace, kde můžeme vidět, že separovatelnou implementací jsme dosáhli výrazného zrychlení obzvláště u menších obrázků s menšími kernely.

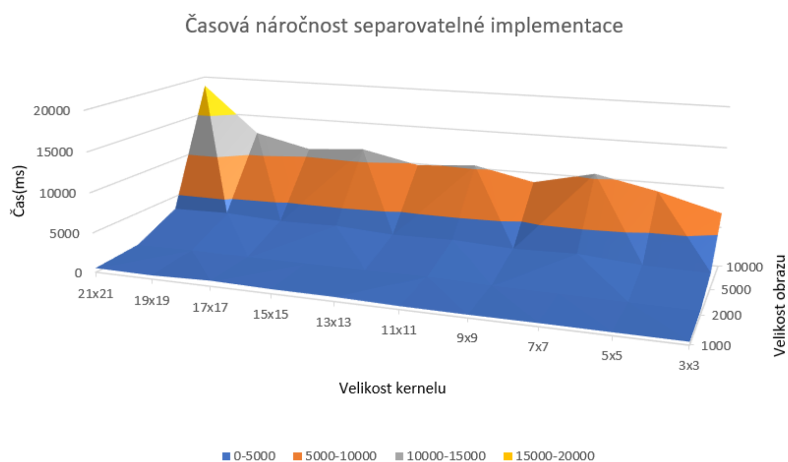
6.7 Porovnání implementací

Obrázek 12 porovnává výkonnost implementací pro vstupní obrázek velikosti 1000×1000. Můžeme si všimnout, že zatímco typická naivní implementace stoupá velmi rychle, ostatní implementace si drží víceméně konstantní růst.

Obrázek 13 porovnává výkonnost implementací pro vstupní obrázek velikosti 2000×2000. Zde můžeme vidět, že časová náročnost typické naivní implementace roste výrazně rychleji než časová náročnost ostatních implementací. Můžeme si také všimnout, že optimalizovaná naivní implementace začíná pomalu růst. Zatímco časová náročnost FFT implementace zůstává nadále

Tabulka 4: Výkonnost separovatelné implementace na jednom uzlu. Čas je udávaný v ms.

Kernel	1000 × 1000	2000 × 2000	5000 × 5000	10000 × 10000
3×3	367	560	2019	6802
5×5	374	593	2131	8905
7×7	396	683	2711	10521
9×9	369	733	2525	8785
11×11	394	961	2886	10390
13×13	546	843	2772	9824
15×15	556	831	3191	11418
17×17	691	827	3084	10873
19×19	463	950	3454	12576
21×21	523	867	3324	18786



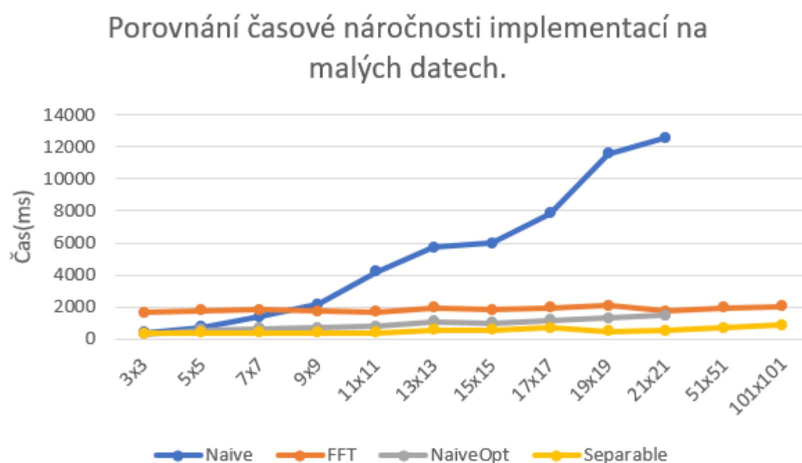
(a) Separovatelná implementace

Obrázek 11: Graf výkonnosti separovatelné implementace.

víceméně konstantní, časová náročnost separovatelné implementace začíná při velkých kernelech stoupat.

Obrázek 14 porovnává výkonnost implementací pro vstupní obrázek velikosti 5000×5000. Zde jsme již úplně vynechali typickou naivní implementaci, neboť je úplně mimo rozsah náročnosti ostatních implementací. Náročnost optimalizované naivní implementace nám nadále stabilně roste, nicméně je stále v rozsahu náročnosti FFT a separovatelné implementace. Za zmínku určitě stojí, že separovatelná implementace, která doposud byla méně náročnější než FFT, již překonala náročnost FFT při využití velkých kernelů.

Obrázek 15 porovnává výkonnost implementací pro vstupní obrázek velikosti 10000×10000. Zde již ani nemá význam mluvit o typické naivní implementaci, neboť její náročnost je o řády výš než u ostatních implementací. Optimalizovaná implementace zde již také zcela utekla mimo rozsah náročnosti FFT a separovatelné implementace. Můžeme zde také krásně vidět, proč je FFT považována za jednu z nejrychlejších implementací. Zatímco náročnost FFT implementace



Obrázek 12: Porovnání výkonnosti implementací na datech velikosti 1000×1000

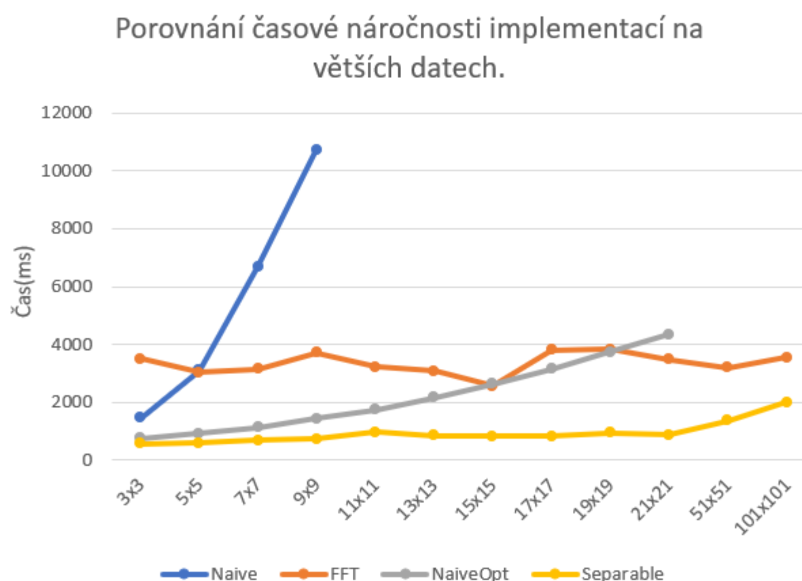
je stále víceméně konstantní, náročnost separovatelné implementace nám velmi rychle stoupá.

Z výše uvedených poznatků se dá usoudit, že FFT implementace je obecně lepší pro velké kernely, kdežto u menších kernelů vyniká separovatelná implementace. To je také hlavní důvod, proč byla implementována varianta separovatelné konvoluce namísto FFT. Velká spousta kernelů používaných při konvoluci je předem definována a má rozměry 3×3 nebo 5×5. Příkladem jsou například zaostření obrazu nebo hledání hran.

6.8 Porovnáni paralelní a sériové implementace

Sériovou implementaci separovatelné a optimalizované naivní konvoluce jsme si již popsali výše a zjistili jsme jejich výhody, nevýhody a jak se srovnávají s FFT konvolucí. U separovatelné a optimalizované, naivní implementace máme ještě jeden prostředek, jak proces výpočtu konvoluce urychlit a tím je paralelizmus. Paralelizmus nám dodatečně pomůže snížit časovou náročnost aplikace a to až několikanásobně. Bohužel, jak popisuje Amdahlův zákon, nemůžeme paralelizovat do nekonečna. Obrázek 16 ukazuje graf, který porovnává výkonnost aplikace s různým počtem uzlů pro separovatelnou implementaci. V grafu je krásně vidět omezení definované Amdahlovým zákonem. Všimněme si, že náročnost výpočtu se s každým navýšením výpočetních uzlů snižuje až do doby, kdy dosáhneme kritické hranice a s přibývajícím počtem dalších uzlů výpočetní náročnost opět roste. Zpomalení je způsobeno nutnou komunikací mezi uzly. V tomto případě je bod zlomu mezi osmi a šestnácti výpočetními uzly.

Při snížení velikosti vstupního obrázku na 5000×5000 je nárůst náročnosti aplikace mnohem znatelnější. Obrázek 17 obsahuje graf výpočetní náročnosti této implementace na vstupním obrázku velikosti 5000×5000. Zde si již můžeme všimnout, že bod zlomu je mezi čtyřmi a osmi uzly namísto osmi a šestnácti uzly, jak tomu bylo u většího obrázku. Tento trend se opakuje pro každý menší obrázek.

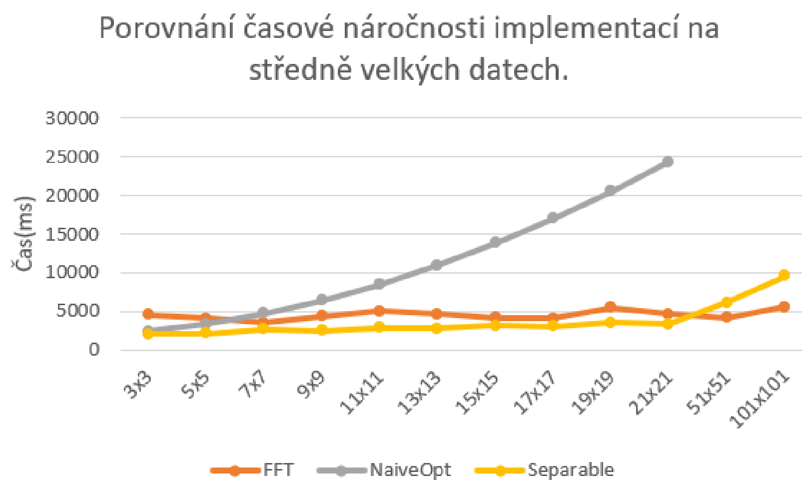


Obrázek 13: Porovnání výkonnosti implementací na datech velikosti 2000×2000

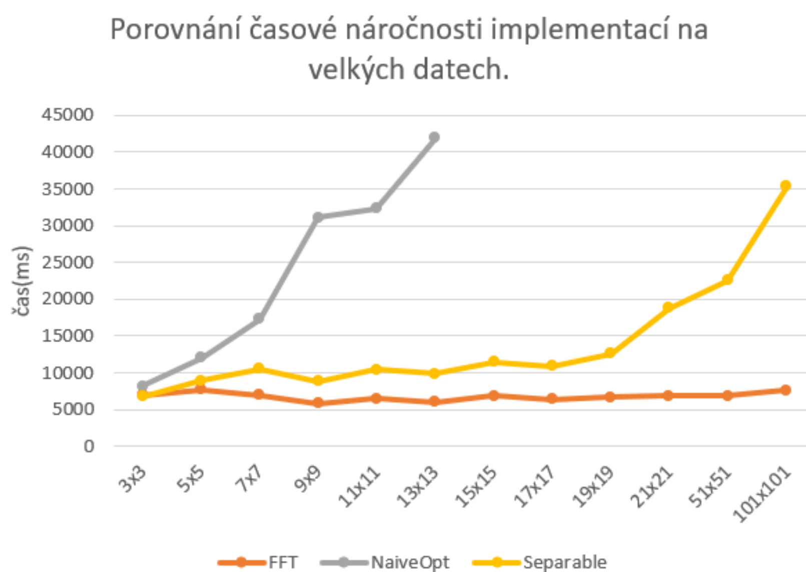
6.9 Paralelní separovatelná implementace a FFT

Jelikož jsme dosáhli dodatečného zrychlení můžeme pro zajímavost porovnat FFT implementaci s paralelní, separovatelnou implementací. Dalo by se říct, že to vůči FFT implementaci, která běží na jednom uzlu není fér. Nás ale zajímá optimalizace výkonnosti separovatelné implementace a FFT implementace je skvělým referenčním bodem. Obrázek 18 znázorňuje graf porovnání výkonnosti FFT implementace a paralelní separovatelné implementace na osmi uzlech.

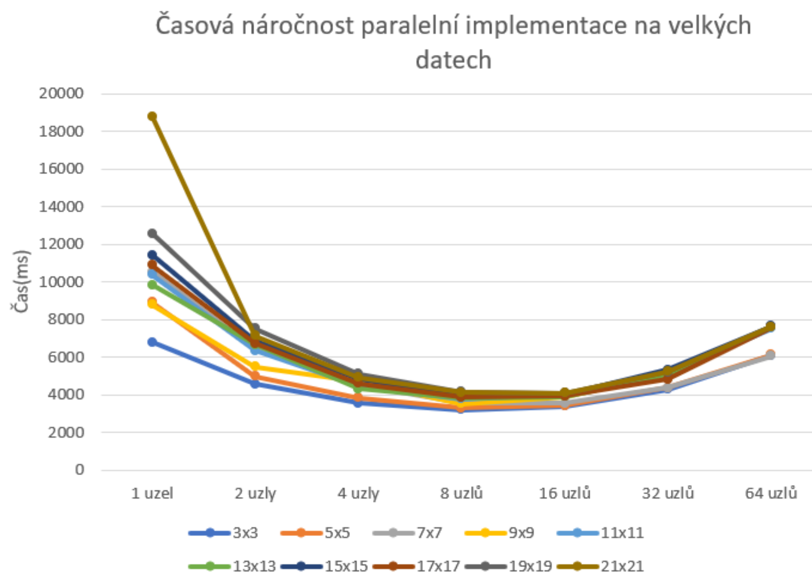
Z Obrázku 18 je vidět, že paralelizovaná implementace separovatelné konvoluce na osmi uzlech je ještě rychlejší než FFT implementace na jednom uzlu. Je možné, že bychom paralelizací FFT implementace dosáhli ještě lepších výsledků nicméně toto nebylo cílem této práce. Z výsledků testování vyplývá, že zatímco použití FFT konvoluce je obecně výhodnější pro velká data a velké kernely, pro menší data/kernely je vhodnější použití separovatelné konvoluce. Z tohoto důvodu tato práce implementuje separovatelnou konvoluci namísto FFT, neboť spousta konvolučních kernelů je předdefinovaných a jsou z pravidla malé.



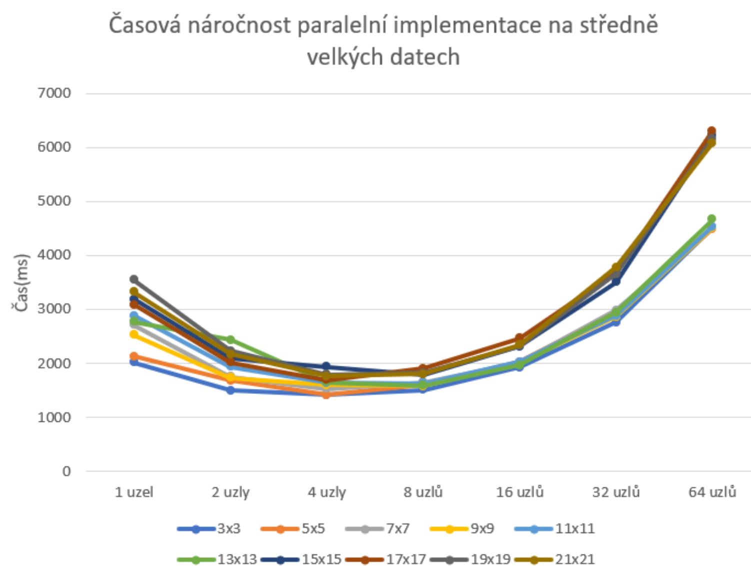
Obrázek 14: Porovnání výkonnosti implementací na datech velikosti 5000×5000



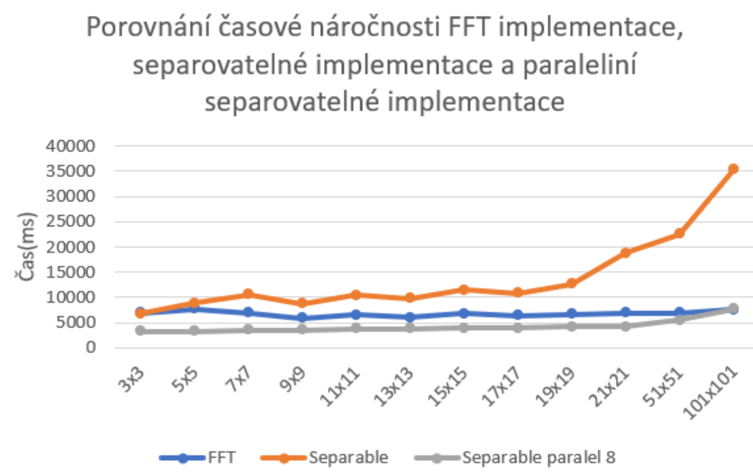
Obrázek 15: Porovnání výkonnosti implementací na datech velikosti 10000×10000



Obrázek 16: Porovnání výkonnosti paralelní implementace na datech velikosti 10000×10000



Obrázek 17: Porovnání výkonnosti paralelní implementace na datech velikosti 5000×5000



Obrázek 18: Porovnání paralelní a FFT implementace na datech velikosti 10000×10000

7 Závěr

Původním cílem práce bylo popsání systémů maker v ImageJ a jeho rozšíření o paralelizaci. V době vymýšlení zadání práce byla makra ještě použitelným řešením. Nicméně v dnešní době je již systém maker velmi zastaralý, a navíc nepodporuje využití knihoven OpenMPI, které se v této práci využívaly. Z tohoto důvodu jsme se po dohodě s vedoucím práce dohodli opustit systém maker a přejít k pluginům ImageJ, které jsou mnohem flexibilnější. Výsledný plugin je však zpětně kompatibilní s makrojazykem ImageJ.

Novým cílem práce bylo tedy vytvoření pluginu, který bude implementovat vhodnou funkci paralelním způsobem. V práci byla zvolena funkce konvoluce, neboť je to rozumně složitá operace, dá se dobře paralelizovat a dá se vhodně použít jako benchmark pro testování různých implementací. V práci se pak rozhodovalo, jakou verzi implementace konvoluce chceme použít. Po důkladném zvážení všech možností padlo rozhodnutí pro použití separovatelné implementace konvoluce. Horkým konkurentem byla FFT implementace konvoluce, která byla zamítnuta z důvodu nedostatečného výkonu při použití malých kernelů, které jsou při konvoluci často využívány. Práce pak navíc implementuje upravenou naivní konvoluci, neboť ne každý kernel je separovatelný, což je podmínkou pro separovatelnou implementaci konvoluce. Samotná implementace byla tvořena a testována na superpočítači Salamon z IT4Innovation centra.

Po implementaci pluginu byly porovnány výsledky aplikace vlastní implementace konvoluce s aplikací implementace konvoluce předinstalované v ImageJ. Následně se mohlo přejít k testování výpočetní náročnosti různých implementací. Byly testovány a porovnány následují implementace na jednom uzlu: Naivní konvoluce, FFT konvoluce, upravená naivní konvoluce a separovatelná konvoluce. Tímto bylo zjištěno, že pro jeden uzel je FFT implementace obecně rychlejší než všechny ostatní. Zajímavostí bylo, že separovatelná implementace byla u malých kernelů a vstupních obrázků srovnatelná, a dokonce i výkonnější než FFT implementace[9]. Následně byla porovnávána výkonnost separovatelné implementace při použití paralelismu.

Bylo zjištěno, že pomocí paralelismu implementace zrychluje jen do určité míry, pak nastává zlom a časová náročnost opět stoupá. Na závěr byla pro zajímavost porovnána nejefektivnější paralelní separovatelná implementace konvoluce s FFT implementací, kde bylo zjištěno, že paralelní separovatelná implementace konvoluce je rychlejší než FFT implementace až do doby, než se použijí opravdu velké obrázky s velkými kernely. Z testování vyplývá, že FFT implementace je obecně rychlejší pro velké vstupní obrázky s využitím velkých kernelů, nicméně separovatelná konvoluce je obecně rychlejší při použití malých kernelů. Výstupem práce je plnohodnotný plugin, který splňuje všechny náležitosti, aby byl instalovatelný a spustitelný ve Fiji. Tento plugin aplikuje konvoluční jádro na vstupní obrázek a vrací zpracovaný obrázek. Zdrojové kódy k projektu jsou nahrány na Githubu a jsou dostupné v příloze této práce. `https://github.com/vondrejVSB/ImageConvolution.git`

[9]FFT implementace by byla stejně náročná i pro neseparovatelné kernely, separovatelné filtry pouze zúžily množinu vstupu a proto je separovatelná konvoluce výkonnější

Literatura

1. SCHINDELIN, Johannes et al. Fiji: an open-source platform for biological-image analysis. *Nature methods*. 2012, roč. 9, č. 7, s. 676–682.
2. RUEDEN, Curtis T; SCHINDELIN, Johannes; HINER, Mark C; DEZONIA, Barry E; WALTER, Alison E; ARENA, Ellen T; ELICEIRI, Kevin W. ImageJ2: ImageJ for the next generation of scientific image data. *BMC bioinformatics*. 2017, roč. 18, č. 1, s. 529.
3. BERTHOLD, Michael R; CEBRON, Nicolas; DILL, Fabian; GABRIEL, Thomas R; KÖTTER, Tobias; MEINL, Thorsten; OHL, Peter; THIEL, Kilian; WISWEDEL, Bernd. KNIME-the Konstanz information miner: version 2.0 and beyond. *AcM SIGKDD explorations Newsletter*. 2009, roč. 11, č. 1, s. 26–31.
4. VEGA-GISBERT, Oscar; ROMAN, Jose E; SQUYRES, Jeffrey M. Design and implementation of Java bindings in Open MPI. *Parallel Computing*. 2016, roč. 59, s. 1–20.
5. SCHMIED, Christopher; STEINBACH, Peter; PIETZSCH, Tobias; PREIBISCH, Stephan; TOMANCAK, Pavel. An automated workflow for parallel processing of large multiview SPIM recordings. *Bioinformatics*. 2016, roč. 32, č. 7, s. 1112–1114.
6. KNIME. *Knime Cluster Execution*. 2016. Dostupné také z: `https://files.knime.com/sites/default/files/inline-images/KNIME_cluster-executor_productsheet_web.pdf`.
7. FAHLMAN, Scott E; HINTON, Geoffrey E; SEJNOWSKI, Terrence J. Massively parallel architectures for Al: NETL, Thistle, and Boltzmann machines. In: *National Conference on Artificial Intelligence, AAAI*. 1983.
8. GRIFFITH, Eric et al. What is cloud computing. *Retrieved from PC Mag: http://au. pcmag. com/networking-communications-software-products/29902/feature/what-is-cloud-computing*. 2016.
9. GALIZIA, Antonella; D'AGOSTINO, Daniele; CLEMATIS, Andrea. An MPI–CUDA library for image processing on HPC architectures. *Journal of Computational and Applied Mathematics*. 2015, roč. 273, s. 414–427. ISSN 0377–0427. Dostupné z DOI: `https://doi.org/10.1016/j.cam.2014.05.004`.
10. MUNSHI, Aaftab; GASTER, Benedict; MATTSON, Timothy G; GINSBURG, Dan. *OpenCL programming guide*. Pearson Education, 2011.
11. HAASE, Robert et al. CLIJ: GPU-accelerated image processing for everyone. *Nature Methods*. 2020, roč. 17, č. 1, s. 5–6.
12. PIETZSCH, Tobias; PREIBISCH, Stephan; TOMANČÁK, Pavel; SAALFELD, Stephan. ImgLib2—generic image processing in Java. *Bioinformatics*. 2012, roč. 28, č. 22, s. 3009–3011.

13. GARLAND, Michael; LE GRAND, Scott; NICKOLLS, John; ANDERSON, Joshua; HARDWICK, Jim; MORTON, Scott; PHILLIPS, Everett; ZHANG, Yao; VOLKOV, Vasily. Parallel computing experiences with CUDA. *IEEE micro*. 2008, roč. 28, č. 4, s. 13–27.
14. SHANAHAN, James G; DAI, Laing. Large scale distributed data science using apache spark. In: *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*. 2015, s. 2323–2324.
15. ZAHARIA, Matei et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*. 2016, roč. 59, č. 11, s. 56–65.
16. NANDIMATH, Jyoti; BANERJEE, Ekata; PATIL, Ankur; KAKADE, Pratima; VAIDYA, Saumitra; CHATURVEDI, Divyansh. Big data analysis using Apache Hadoop. In: *2013 IEEE 14th International Conference on Information Reuse & Integration (IRI)*. 2013, s. 700–703.
17. JOSHI, Shrinivas B. Apache hadoop performance-tuning methodologies and best practices. In: *Proceedings of the 3rd acm/spec international conference on performance engineering*. 2012, s. 241–242.
18. DEAN, Jeffrey; GHEMAWAT, Sanjay. MapReduce: simplified data processing on large clusters. *Communications of the ACM*. 2008, roč. 51, č. 1, s. 107–113.
19. HILL, Mark D; MARTY, Michael R. Amdahl's law in the multicore era. *Computer*. 2008, roč. 41, č. 7, s. 33–38.
20. GUSTAFSON, John L. Reevaluating Amdahl's law. *Communications of the ACM*. 1988, roč. 31, č. 5, s. 532–533.
21. NOVAK, Jan; KAPLANYAN, Anton; LIKTOR, Gabor; DACHSBACHER, Carsten. GPU Computing: Image Convolution. 2012.
22. LUDWIG, Jamie. Image convolution. *Portland State University*. 2013.
23. SMITH, Steven W et al. The scientist and engineer's guide to digital signal processing. 1997.