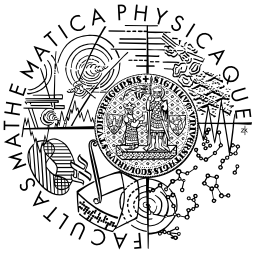


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Jan Kubový

Prostředí pro lifting

Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. Josef Pelikán
Studijní program: Informatika
Studijní obor: IPSS

Praha 2018

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Název práce: Prostředí pro lifting

Autor: Jan Kubový

Katedra: Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. Josef Pelikán, Katedra softwaru a výuky informatiky

Abstrakt: Cílem práce je vytvořit knihovnu, pomocí které bude možno snadno vytvářet výpočetní sítě a dále s nimi experimentovat. Pojmem výpočetní sítě jsou myšleny algoritmy, které je možné rozdělit na jednoduché části (uzly), ze kterých se následně vytvoří větší výpočetní celek. Hlavní zaměření této knihovny je snadné experimentování s transformacemi založenými na liftingu. Pro tato zapojení existují inverzní operace, čehož se využívá při bezeztrátové komprimaci dat nebo signálu. Důraz u této práce byl kladen na jednoduchost tvorby nových uzlů a následných zapojení. Nedílnou součástí práce je i ukázka několika transformací založených na liftingu.

Klíčová slova: wavelety lifting šifrování steganografie

Title: Environment for Lifting

Author: Jan Kubový

Department: Department of Software and Computer Science Education

Supervisor: RNDr. Josef Pelikán, Department of Software and Computer Science Education

Abstract: The aim of the thesis is to create a library that will provide ease way to creating and experimenting with computing networks. The concpet of computing netowork can be explained as algorithms whitch can be devided into small simple parts (nodes). The main focus of this library is to easily experiment with transformations based on lifting. There are inverse operations for these connections, which are used for lossless compression of data or signal. Emphasis was put on the simplicity of creating new nodes and subsequent connections. An integral part of the work is also an example of several transformations based on lifting.

Keywords: wavelets lifting encryption steganography

Rád bych na tomto místě poděkoval RNDr. Josefovi Pelikánovi za odborné vedení, rady a veškerý čas, který mi věnoval při zpracování této práce. Dále děkuji všem, kteří mě jakkoli podpořili při studiu.

Obsah

1. Úvod

1.1 Zaměření práce

Cílem této práce je vytvořit knihovnu, pomocí které lze snadno implementovat různé výpočetní sítě transformující vstupní signál. Tyto sítě se skládají z výpočetních uzlů či segmentů a definic jejich vzájemného propojení. Samotný výpočet pak probíhá přeposíláním mezivýsledků mezi jednotlivými částmi výpočetní sítě. Důraz byl kladen na jednoduchou tvorbu nových uzlů a snadné změny v již existujícím zapojení.

Hlavní podmínkou pro tuto knihovnu byla možnost implementovat výpočetní sítě, umožňující transformace založené na liftingu, u kterých jsou známé inverzní operace.

Dále bylo v plánu vytvořit různé sítě pro demonstraci funkčnosti a všestrannosti knihovny. Jednou z podmínek práce byla univerzalita knihovny, aby byla schopná pracovat s různými typy signálu. Na vstupu tedy může být jak zvuk, tak i barvy pixelů obrázku.

1.2 Výpočetní síť

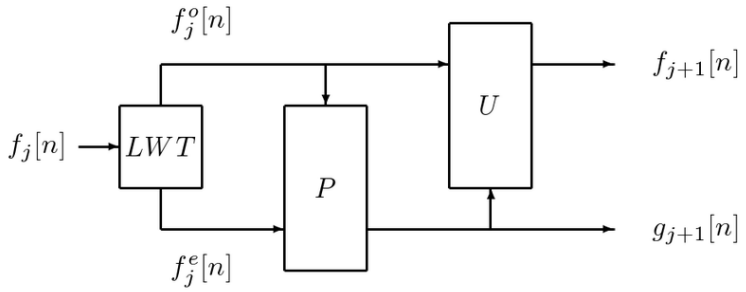
Pro správné porozumění textu je třeba definovat pojem výpočetní síť. Jedná se o algoritmy nebo transformace vstupních dat, které je možné rozdělit na malé části (uzly), ze kterých se následně skládá výsledný výpočetní celek. Typickým příkladem takového zapojení jsou šifrovací algoritmy. Ty jsou často vytvářeny pomocí malých částí (například jedna část může počítat XOR mezi dvěma signály), které se dále propojují. Podobným způsobem fungují i transformace založené na liftingu, kterými se budeme zabývat v této práci.

1.3 Lifting

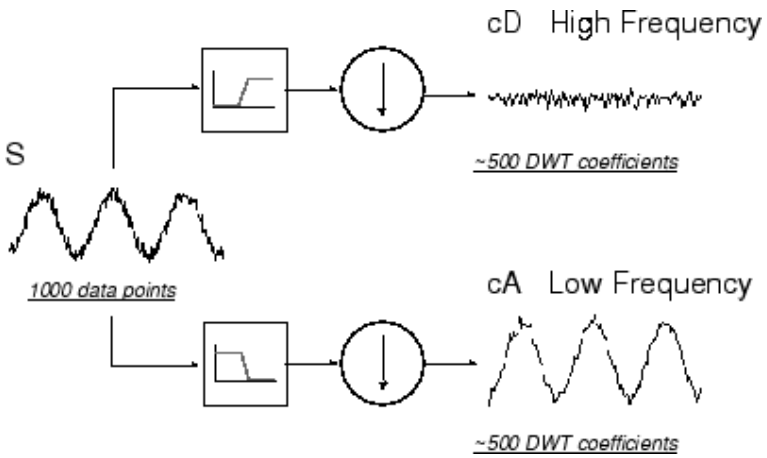
Lifting je výpočetní schéma, které představil Sweldens a kol. (1995), pro práci s diskrétními vlnkovými transformacemi (DWT). Toto schéma se dělí na tři části: rozklad signálu, predikce a aktualizace. V první části se rozděluje signál na dvě části (u diskrétních systémů se často dělí na dvě skupiny se sudým a lichým indexem). Další fází je predikce, kde dochází k první transformaci signálu. Poslední fází je aktualizace, kde dochází k další změně signálu, což je třeba pro zachování určitých vlastností výstupního signálu po predikci. Toto schéma je znázorněné na obrázku 1.1. Podrobný matematický rozbor výpočetního schématu je dostupný v článku Sweldens (1997). Část predikce a aktualizace se může opakovat a často se tak i děje u složitějších výpočetních schémat. Pro naše účely stačí výše zmíněný obecný popis schématu a splnění podmínky, aby operace (kulaté uzly ve schématu 1.1) byly invertibilní.

Samotná invertibilita funkce je velice důležitá pro její využití, protože se často používá pro bezeztrátovou transformaci signálu. Příkladem takové transformace je komprimace obrázku standardem JPEG 2000. Jak již bylo zmíněno v úvodu, lifting je schéma zapojení diskrétní vlnkové transformace, které poprvé představil

Wim Sweldens. Z této myšlenky vychází obecný lifting autorů Rolón a Salembier (2007). Příklad rozkladu zvukového signálu pomocí DWT je vidět na obrázku 1.2.



Obrázek 1.1: Obecný lifting - schéma zapojení.



Obrázek 1.2: Příklad rozložení zvukového signálu pomocí DWT.

1.4 Celočíselný lifting

U liftingových transformací se často počítá s reálnými hodnotami, což může být problém při ukládání výsledku do existujících formátů podporující pouze celočíselné hodnoty. Typickým příkladem je transformace obrázků, kde se výsledek ukládá do formátu podporující pouze celočíselné hodnoty jednotlivých barevných kanálů. Proto se přišlo s úpravou liftingových transformací, díky které mohou být mezivýsledky stále reálné, ale výstupní hodnoty celé sítě jsou celočíselné. Matematický rozbor a příklady různých výpočetních sítí jsou dostupné prostřednictvím práce autora Calderbank a kol. (1998).

1.5 Komprese dat

Komprese je proces, při kterém se zmenší objem dat potřebný pro uložení dané informace. Rozlišujeme komprese podle možnosti rekonstrukce původních dat na ztrátové a bezeztrátové. Pojem kompresní poměr značí kolikrát se zmenšila

velikost dat. Jak vyplývá z názvu, u ztrátové komprese se vytrácí část dat a komprimovaná data jsou pouze přibližná originálu. Tento druh komprese se nejčastěji využívá u multimediálních dat jako je obraz (video) nebo audio. V případě bezeztrátových kompresí je kompresní poměr horší než u ztrátových kompresí, ale umožňují zrekonstruovat původní data.

1.6 Komprimace obrázků

Komprimace obrázku je proces, při kterém se zmenší objem dat reprezentující daný obrázek. Stejně jako u komprese dat, existují dva druhy komprimace – ztrátová a bezeztrátová. Při komprimaci obrázků se nejčastěji používá ztrátová komprese pro větší kompresní poměr. Jedna z nejjednodušších metod komprimace obrázku je redukce barevného prostoru. Tím se získá obrázek menší velikosti, ale s lehce odlišnými barvami. Příkladem takovéto komprimace je obrázek 1.3. Nevýhodou je znatelný rozdíl mezi originálním a komprimovaným obrázkem, proto se spíše používají složitější metody, které se snaží změnit vzhled co nejméně.

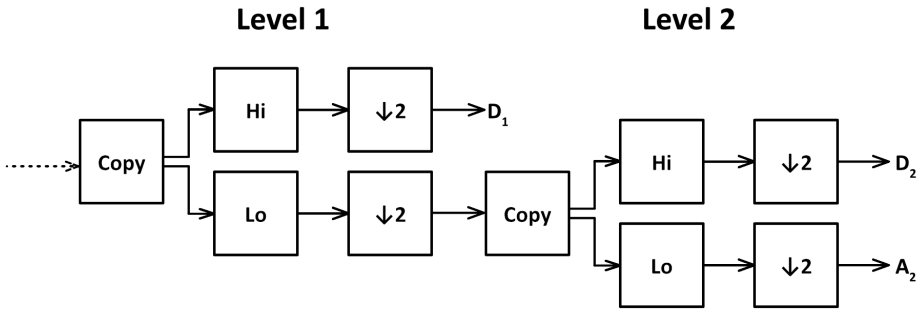
Obrázek 1.3: Převedení barev obrázku ze tří barevných kanálů na jeden. Tím lze obrázek teoreticky zmenšit na třetinu.

My se zaměříme na bezeztrátovou kompresi založenou na diskrétní vlnkové

transformaci (DWT), popsané v kapitole Lifting. Konkrétně jde o 2D DWT, protože se transformace použije jednou na řádky a podruhé na sloupce pixelů obrázku. Tímto postupem se zmenší komprimovaný obrázek na čtvrtinu. Při této transformaci se oddělí nízké frekvence (komprimovaný obraz) a vysoké frekvence (detaily, o které komprimovaný obraz přišel).

1.7 Mallatův rozklad

Jedná se o transformaci signálu, při které se rozdělí na čtyři stejně velké části. Jedna z částí je aproximace původního signálu se čtvrtinovým rozlišením. Ostatní části v sobě zachovávají informace, o které signál s aproximací přišel. Celková velikost signálu se nezmění, pouze se rozdělí na čtyři části (viz obrázek 1.4). Původně se při tomto rozkladu použila vlnková transformace, ovšem princip rozkladu se dá použít na jakékoli transformace, ke kterým existuje inverzní operace, a která rozdělují vstupní signál na dva výstupní. Počet úrovní rozkladu je určen podle počtu opakování rozkladu na již aproximovaném (zmenšeném) signálu. Matematický rozbor tohoto rozložení je uveden v článku Mallat (1989).



Obrázek 1.4: Dvě úrovně Mallatova rozkladu obrázku. Zkratky Hi (Lo) reprezentuje filtr vysokofrekvenční (nízkofrekvenční), D detaily a A aproximace.

1.8 Entropie dat

Informační entropie je vlastnost množiny dat určující střední hodnotu informace jedné instance dat. Z matematického hlediska u 2D pole hodnot o velikost $N * M$ se výsledek počítá následovně.

$$H(X) = -\sum P(s_i) log_2 P(s_i)$$

$$P(s_i = \frac{\mid \{(x,y) : X(x,y) = s_i\} \mid}{N * M}$$

V této práci se entropií dat budeme zabývat u porovnávání různých liftingových transformací, kde podle této hodnoty budeme určovat, která transformace nám lépe zkomprimovala obrázek. Výše zmíněný výpočet budeme používat u Mallatova rozkladu, kde X bude reprezentovat jednu část rozkladu a $|X|$ počet pixelů

v X. Celkovou entropii S budeme počítat takto:

$$S = \sum_{X} \frac{|X|}{N * M} * H(X)$$

1.9 Použití knihovny

Práce s knihovnou je rozdělena na dvě části. První částí je definování jednotlivých uzlů a výpočetních sítí, které budeme chtít používat. Je možné použít nějaké již existující uzly a sítě, které jsou součástí knihovny. V základu jsou dostupné uzly pro práci s obrázky, výpočetní uzly a další, o kterých si řekneme dále v textu. Druhou částí je samotné spouštění výpočtu sítě na zadaných vstupech. V této části je možné použít připravené `Testery`, které se starají o opakované spouštění výpočtu s různými parametry. Tento způsob je vhodný při testování nové výpočetní sítě nebo hledání vhodných parametrů sítě.

Knihovna je implementována v jazyce C# pro jeho jednoduchost a rozšířenost. Záměrem bylo co nejvíce usnadnit budoucím uživatelům práci s knihovnou a vytváření vlastních výpočetních sítí. Proto jsme vybrali výše zmíněný programovací jazyk, který splňuje stanovené požadavky. Jazyk C# navíc umožňuje snadnou práci s grafickým rozhraním, čehož by se dalo využít při případném rozšiřování knihovny.

Příklad definice zapojení počítající Mallatův rozklad dvou úrovní obrázku podle liftingu DaubechiesWaveletD4:

```
int level = 2;
string inputName = "../lenna.png";
string outputName = "../lenna_mallat.png";

Bitmap input = new Bitmap(inputName);
Bitmap output = new Bitmap(input.Width, input.Height, input.PixelFormat);
var imageReader = new BitmapReader<int>(input);
var mallat = new MallatDecomposition<int, DaubechiesWaveletD4<int>>
        (imageReader).Build(input.Size, level);
var mallatMerger = new MallatMerger<int>
        (mallat.GetRecalculatedSize(), level, mallat);
var imageWriter = new BitmapWriter<int>(output, outputName, mallatMerger);
imageWriter.Process();
```

2. Existující knihovny

2.1 Obecná koncepce knihoven

Knihoven pro tvorbu výpočetních sítí je celá řada. Nabízí se tedy otázka, k čemu by bylo dobré vytvářet další knihovnu, která pracuje na podobném principu. V dnešní době se vyskytují hlavně knihovny specializované na umělou inteligenci, jako je například TensorFlow od společnosti Google. Dalším typem zaměření je vícevláknový výpočet s důrazem na vysoký výkon. Příkladem takového přístupu je knihovna Bobox psaná v jazyce C++.

2.2 Bobox

Projekt Bobox vznikl za účelem zjednodušení psaní paralelních programů pro zpracování velkého množství dat. Značná výhoda této knihovny je možnost psaní sériového kódu, který se následně zpracovává paralelně podle zapojení jednotlivých boxů. Veškeré problémy se psaním paralelních algoritmů jako je třeba synchronizace dat při čtení z více vláken je ponechána na samotné knihovně. To usnadňuje použití, protože se uživatel může soustředit na psaní výkonného kódu a nemusí se starat o zmíněné problémy při paralelním zpracování dat. Vývojáři této knihovny si napsali vlastní zpracování požadavků (Tasků). Výpočet jednoho boxu odpovídá jednomu Tasku, který se zařadí do připraveného Threadpoolu.

Princip výpočtu probíhá od zdroje dat do dalších boxů. Zjednodušeně řečeno se výpočet jednoho boxu spustí, až když jsou dostupná veškerá data na vstupu. Problém nastane, když se při výpočtu jednoho boxu potřebují data, která na vstupu nebyla. Příkladem takové situace jsou právě výpočty u nějakých liftingových zapojeních. U výpočetní sítě `(4,2) interpolating transformation` (Calderbank a kol., 1998), která pracuje podle níže napsaných rovnic, si můžeme všimnout dvou problémů:

$$d_{1,l} = s_{0,2l+1} - \lfloor \frac{9}{16}(s_{0,2l} + s_{0,2l+2}) - \frac{1}{16}(s_{0,2l-2} + s_{0,2l+4}) + \frac{1}{2} \rfloor$$
$$s_{1,l} = s_{0,2l} + \lfloor \frac{1}{4}(d_{1,l-1} + d_{1,l}) + \frac{1}{2} \rfloor$$

Po dosazení $l = 0$ při počítání prvního výstupu d_1 a s_1 dostaneme následující rovnice.

$$d_{1,0} = s_{0,1} - \lfloor \frac{9}{16}(s_{0,0} + s_{0,2}) - \frac{1}{16}(s_{0,-2} + s_{0,4}) + \frac{1}{2} \rfloor$$
$$s_{1,0} = s_{0,0} + \lfloor \frac{1}{4}(d_{1,-1} + d_{1,0}) + \frac{1}{2} \rfloor$$

Hned vidíme, že u výpočtu $d_{1,0}$(resp. $s_{1,0}$) potřebujeme vstup $s_{0,-2}$(resp. výsledek výpočtu $d_{1,-1}$), k čemuž jsou potřeba data mimo rozsah vstupu. Na první pohled by se mohlo zdát, že problém vyřešíme tím, že bychom na vstup poslali nějaká nulová data na začátku, ale to by nám rozhodilo výpočet. Řekněme, že bychom měli správně spočítaná data pro $d_{1,-1}$ a $d_{1,0}$, ale poté u výpočtu $s_{1,0}$

bychom jako první na řadě měli data $s_{0,-4}$, které jsem museli přidat na začátek a tudíž by výsledek nebyl správný.

Je možné tento problém vyřešit nějakými podmínkami ve výpočtu, kde by se testovalo, kolik již přišlo dat a případně je zahazovat, aby nedocházelo k popisovanému problému, ale to zásadním způsobem ztěžuje přípravu takovéto sítě. Případná lehká změna by vyústila ke komplexní změně všech boxů, kde by mohlo k těmto problémům docházet. Předpokládáme-li, že uživatel knihovny chce co nejjednodušeji experimentovat s různými sítěmi a jejími parametry, je vhodné aby takovéto změny byly co nejsnadnější. Druhým zdánlivě možným řešením problému by mohlo být generování nulových dat za běhu v uzlu, ve kterém by byla potřeba data mimo rozsah vstupu. To ale není možné, jelikož průchodem sítí by se data mohla změnit a výpočet by probíhal se špatnými daty.

2.3 TensorFlow

Druhou zmiňovanou knihovnou pro výpočty pomocí výpočetních sítí je TensorFlow. Projekt vznikl pro trénování neuronových sítí a lze říci, že pomocí této knihovny je možné vytvářet i liftingová zapojení. Rozsáhlost tohoto projektu je však v našem případě spíše zápor, protože hodně částí je specializováno na činnosti, které v našem případě nejsou potřeba. Zároveň ale bohužel neposkytuje potřebné nástroje pro snadnou implementaci typických výpočetních sítí, jako je třeba Mallatův rozklad `n` úrovní.

3. Popis knihovny

3.1 Popis návrhu

Jak již bylo řečeno v sekci 1.1, cílem bylo vytvořit prostředí, ve kterém bude možné vytvářet výpočetní celky, primárně pak liftingové sítě. Pro tento účel vznikla následující práce, umožňující tvorbu výpočetních uzlů, které se pak skládají do sítí a je možné s nimi následně experimentovat.

Dodávaná knihovna `EnvironmentForLifting.dll` je psaná v jazyce C# a nabízí se používat ji ze stejného jazyka. Není to však podmínkou, pro snadnější zápis je možné využít například jazyk IronPython. Výpočetní síť je složena z jednotlivých uzlů a výpočetních segmentů. Každý uzel může mít libovolný počet vstupů a právě jeden nebo žádný výstup. Uzly s nulovým počtem výstupů odpovídají posledním uzlům v síti, které ukládají nebo nějak jinak zpracovávají výsledky výpočtu sítě. Z pohledu uživatele je možné na jeden výstup připojit více uzlů, což je interně zajištěno logickým prvkem zvaný `DataStream`, který se nachází mezi veškerými uzly. Jeho úkolem je ukládat mezivýsledky výpočtů a v případě potřeby poskytovat data připojeným uzlům. `DataStream` není viditelný pro uživatele a proto se jím nemusí nijak zabývat. Rozhraní uzlu a segmentu si popíšeme v následujících podkapitolách.

Výpočet je zahájen od posledního uzlu, který se dotazuje na výsledky předchozích uzlů. Ty se v případě potřeby ptají stejným způsobem na data na svých vstupech. Výhodou tohoto přístupu je možnost omezit tok výpočtu v případě, že některé větve nejsou třeba. Je třeba myslet na fakt, že u liftingových sítí se často pracuje s daty, které nejdou bezprostředně po sobě. U návrhu knihovny se kladl důraz na jednoduchý zápis matematických rovnic. V těch se často vyskytují indexované mezivýsledky. Z toho důvodu bylo přistoupeno k indexaci mezivýsledků jednotlivých uzlů, kde se následující uzel dotazuje předchůdce na data podle indexu nikoli na data následující. Ukázku si rozebereme v podkapitole věnované uzlům.

Instance třídy `Data` představuje jednu část signálu, který se posílají mezi jednotlivými uzly. Samotná data jsou neměnná, a proto při jakékoli operaci vznikají data nová. Tím je možné sdílet data mezi uzly i vlákny bez obavy ztráty konzistence v průběhu výpočtu. Data jsou generická a interně se jedná o pole hodnot `T`, na kterých existují určité matematické operace. V zásadě se jedná o numerické typy od `byte` po `decimal`.

3.2 Data

Veškerá komunikace mezi jednotlivými uzly probíhá přes instance třídy `Data`. Vnitřek třídy obsahuje následující:

- `T[] _signal`: Pole hodnot typu `T`. Jde o přeposílaný signál, se kterým se pracuje.
- `bool isEmpty`: Pravdivostní hodnota určující, jestli jde o platná data nebo o zarážku.

- `Data<T> operator ...`: Takové generické operátory, u kterých je jednoznačné jak mají pracovat. Například zde není násobení `Data<T>` a `Data<T>`, protože není jisté, jestli by uživatel chtěl násobení po složkách nebo třeba každou položku s každou. Operátory jsou použity pro jednodušší zápis výpočtů.

- `ToX(Data<T>)` a `XToT(Data<X>)`: Funkce pro konverzi dat z a do typu `T`.

- `Data<float> FloorWithHalfAdded(Data<double> data)`: Funkce pro zaokrouhlení signálu. Hlavně se používá v celočíselných liftingových sítích.

Ke generičnosti dat jsme přistoupili ve snaze vytvořit knihovnu co nejvíce univerzální a dát uživateli možnost výběru podle jeho nároků. Této vlastnosti se využilo u celočíselných liftingových zapojení, kde se používá neceločíselný typ pouze u mezivýsledků výpočtu. Při programování bylo třeba vyřešit problém, jak zajistit možnost psát operátory u generických dat. V jazyku C# není možné vynutit statické operátory rozhraním a proto není možné takto omezit generický typ. Jednou z možných řešení je použití klíčové slovo `dynamic`, který způsobí překlad dané operace až za běhu podle daného typu. Tím by se přišlo nejen o typovou kontrolu za překladu, což je klad, o který nechceme přijít, ale je to podstatně pomalejší, než počítání pomocí negenerických operací. My jsme přistoupili k jinému řešení, a to k použití třídy `Expression`, kde lze definovat jednotlivé operace a přeložit je při překladu celého projektu. Je třeba ručně napsat možné operace a nechat je ručně přeložit, ale odměnou nám je typová kontrola a podstatně rychlejší zpracování operací. Jelikož je knihovna psaná pro tvorbu výpočetních sítí, přistoupili jsme k druhé variantě.

V některých případech je třeba označit data, která již nejsou platná. K tomu slouží výše popsaná proměnná `isEmpty`. Pro testování je předem připravena vlastnost třídy `IsEmpty`, která nám vrátí požadovanou hodnotu. Uživatel se ve výpočetních uzlech nemusí starat o to, jestli jde o platná nebo neplatná data, protože jakékoli počítání s neplatnými daty vede opět k neplatným datům.

3.3 Synchronní a Asynchronní výpočet

Knihovna je vytvořena tak, aby bylo možné program spouštět sériově i paralelně. První zmiňovaný způsob funguje následovně. Když nějaký výpočetní uzel potřebuje data od předchozích uzlů, tak se jich synchronně zeptá. To znamená, že se volání zanoří a ve výpočtu se pokračuje, až když jsou data dostupná. To by teoreticky mohlo vést k chybě přetečení zásobníku při volání funkcí, ale reálně k tomuto nedochází, protože výpočetní sítě nejsou tak rozsáhlé.

Původní návrh vícevláknového výpočtu využíval již připravené `Tasky` v jazyce C#. Myšlenka byla, že instance třídy `Data<T>` se obalí `Taskem` a veškeré výpočty se budou počítat mimo hlavní vlákno. Jestli-že nějaký uzel potřebuje data z předchozího uzlu, zeptá se ho asynchronně a ten mu vrátí `Task`. To způsobí, že se mohou počítat všechna data potřebná pro výpočet uzlu zároveň. Výpočet se ale takto může hodně větvit a může vznikat velké množství instancí `Tasku`. V rámci experimentů se ukázalo, že tento způsob nepřináší žádné výhody, ba naopak výpočet zpomaluje.

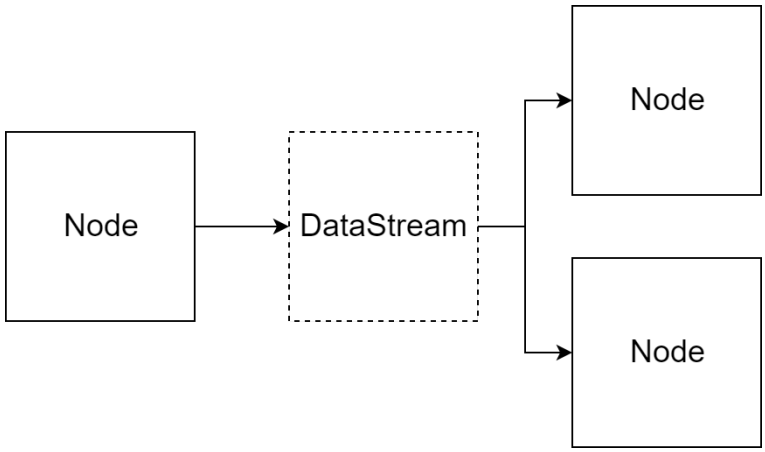
Nakonec jsme přišli s jiným návrhem, kde vícevláknový výpočet používá synchronní metody výpočtu v celé síti, které jsou volány z výstupního uzlu paralelně. Využili jsme vlastnosti knihovny, že jsou data indexována a ve výstupním uzlu vytváříme `Tasky`, ve kterých probíhá synchronní výpočet pro jedna výstupní data. Každý `Task` odpovídá dotazu na jedna výstupní data. Tímto způsobem se výpočet razantně zrychlí a z pohledu uživatele zjednoduší návrh sítě, protože již nebude muset psát asynchronní metody výpočtu v každém uzlu a segmentu.

3.4 DataStream

Při vytváření sítí je třeba nějakým způsobem zajistit předávání dat mezi jednotlivými uzly. K tomuto účelu vznikla třídy `DataStream`, která kombinuje dvě funkce. Nejprve je třeba napsat, že každý uzel má právě jeden `DataStream`, který ho může žádat o nějaká data. Pouze na tuto konkrétní instanci třídy `DataStream` jsou ve skutečnosti napojeny ostatní uzly, které se jejím prostřednictvím ptají na data uzlu.

První funkcí tohoto prvku v síti je uchovávání mezivýsledků, aby nebylo třeba při opakovanému přístupu ke stejným datům pokaždé data počítat znovu. Interně si ke každému uzlu, který se může ptát na data, pamatuje index posledních vyžádaných dat. Z těchto údajů můžeme za určitých okolností vyvodit, že nějaká data již nejsou potřeba a je možné je bezpečně zahodit. V některých situacích není vlastnost ukládání mezivýsledků žádána a je možné ji vypnout. To sice vede k delšímu běhu výpočtu, protože se některé části výpočtu upakují, ale na druhou stranu je snížena paměťová náročnost programu.

Druhou funkcí je umožnění napojit více uzlů na výstup jednoho uzlu. K tomuto větvení dochází u výpočetních sítí zcela běžně. Jak již bylo zmíněno dříve, hlavním cílem bylo zajistit co nejpohodlnější způsob tvorby zapojení. Proto je třída `DataStream` pro uživatele skryta. Více o fungování tohoto systému bude popsáno v sekci Propojení uzlů.

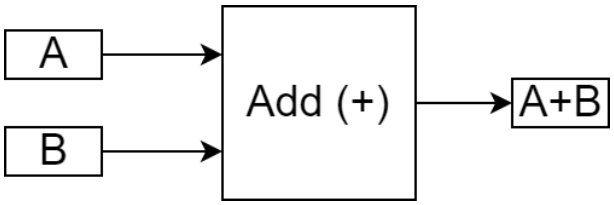


Obrázek 3.1: Ilustrace zapojení uzlů přes DataStream.

3.5 Uzel

Uzel je hlavní a zároveň nejmenší částí jakékoli výpočetní sítě. Lze je dělit do dvou kategorií, podle jejich účelu:

- Výpočetní uzel: Jak vyplývá z názvu, tyto uzly počítají nové hodnoty na základě přijatých dat. Příkladem tohoto typu může být sčítací uzel 3.2. Přijme data ze dvou vstupů, signál sečte a data předá do výstupního bufferu.
- Vstupní/výstupní uzel: S tímto typem se setkáme v každé vytvořené síti. Vždy je třeba dodat data do sítě a následně uložit nebo jinak zpracovat výsledek. Vstupní uzly se vyznačují tím, že nemají žádného předchůdce. Výstupní uzly sice mají předchůdce, ale na ně již žádný uzel nenavazuje. Typicky se starají o nashromáždění výsledků, které následně uloží nebo vypíší na konzoli.



Obrázek 3.2: Příklad výpočetního uzlu, který sčítá data.

Jelikož se propojování uzlů provádí už při inicializaci uzlu, je třeba správně napsat konstruktor uzlu. V něm je možné nastavit, kolik vstupů bude daný uzel vyžadovat. Příklady propojování uzlů si popíšeme v následující sekci. Při inicializaci uzlu se automaticky inicializuje i `DataStream` napojený na výstup tohoto uzlu. Z pohledu uživatele žádá o data následující uzel, ale ve skutečnosti se tak děje prostřednictvím zmíněného `DataStream`u.

Každý uzel je potomkem abstraktní třídy `Node`, která obsahuje následující metody a vlastnosti.

- `Node<T> this[int]`: Indexer, který vrací výstup uzlu. U jednoduchého uzlu je dostupný pouze jeden výstup, ale segment může obsahovat výstupů více.
- `InputStreamsEnvelope<T>[] Input`: Vlastnost, přes kterou se přistupuje k jednotlivým vstupům uzlu u výpočtu.
- `Data<T> GetData(int)`: Synchronní verze funkce, která počítá výsledek výpočtu uzlu podle zadaného indexu dat.
- `Task<Data<T>> GetDataAsync(int)`: Asynchronní verze předešlé funkce.
- `void Process()`: Metoda určující běh uzlu. Používá se u výstupních uzlů, u kterých chceme zahájit synchronní běh výpočtu sítě.
- `void ProcessAsync()`: Stejná metod jako `void Process()` s tím rozdílem, že výpočet bude asynchronní.

- `void Reset()`: Metoda, kterou se vyvolá resetování všech uzlů napojený na uzel, na kterém byla tato metoda zavolána. V praxi se využívá k resetování a zahození mezivýsledků celé sítě při spuštění na výstupním uzlu.
- `ParameterStorage`: Vlastnost uzlu přes kterou se nastavují parametry celé sítě. Při změně na nějakém uzlu se instance parametrů propaguje přes všechny předchozí uzly až na začátek sítě.
- `void SetParametersForThisNode(ParameterStorage)`: Nastavením procedury určíme, jak se má uzel zachovat po přijetí parametrů.
- `int DataLength`: Vlastnost, pomocí které se nastavuje délka dat. Jinými slovy se nastavuje, kolik bude mít prvků pole v instanci třídy `Data`.
- `void SetDataLengthForThisNode(int)`: Přepsáním této metody je možné určit, jak se má uzel zachovat při změně délky dat.
- `int OldDataKeepCount`: Stejně jako v předešlém případu se i zde jedná o vlastnost. Nastavuje všem `DataStreamům` po cestě výpočtu, kolik starých dat si mají uchovávat, respektive jak stará data již mají považovat za zbytečná a zahazovat je.
- `void SetOldDataKeepCountForThisNode(int)`: K předešlé vlastnosti se váže metoda určující, jak se má uzel zachovat při změně zmiňovaného počtu ukládaných dat.
- `bool DropOldData`: Nastavení, zda se mají mezivýsledky uchovávat nebo zahazovat.
- `void SetDropOldDataForThisNode(bool)`: Metoda definující postup, jak se má uzel zachovat při změně předešlé vlastnosti.

Každý uzel je potomkem abstraktní třídy `Node`, která obsahuje výše zmíněné metody, funkce a vlastnosti. Žádná metoda není abstraktní, takže při psaní potomků této třídy se můžou implementovat pouze ty části, které budou potřeba. Například uzel, který ukládá získaná data na disk nebude obsahovat metody `GetData(int)` a `GetDataAsync(int)`, protože se nikdo tohoto uzlu na mezivýsledky ptát nemá.

3.6 Propojení uzlů

Při návrhu knihovny jsme se snažili přijít s takovým propojováním uzlů, aby z pohledu uživatele bylo třeba napsat minimum kódu a zároveň to bylo co nejvíce odolné na případné chyby. Proto se uzly propojují již v konstruktoru, čímž je poskytnuta kontrola za překladu, jestli dané uzly mají potřebné vstupy pro správné fungování. Při vytváření uzlu se zadá, jací budou jeho předchůdci, kterých se při výpočtu bude ptát na data. Interně se uzel napojí na výstupní `DataStream`, který přísluší danému uzlu. Tím je zajištěno, že každý uzel má maximálně jeden výstup, ale data z něj může čerpat libovolný počet uzlů. Toto oddělení je opět kvůli zjednodušení práce uživatele knihovny, kde z jeho pohledu není důležité, jak se uzly dotazují na data předchůdců, ale konkrétně jakých uzlů se mají dotazovat.

```
// Vytvoření vstupních uzlů
var inputNode1 = new Constant<int>().SetConstant(new[] { 42 });
var inputNode2 = new Constant<int>().SetConstant(new[] { 6 });

// Vytvoření uzlu pro sčítání, který má 2 vstupy, a to inputNode1 a
// inputNode2
var add = new Add<int>(inputNode1, inputNode2);

// Vytvoření výstupního uzlu, který je napojený na uzel add.
var printer = new Print<int>(add);
```

Na příkladu jsme definovali jednoduchou výpočetní síť, který má dva vstupní uzly vracející konstantní data. Dále obsahuje jeden sčítací uzel a jeden výstupní uzel, který vypisuje výsledky na konzoli.

3.7 Výpočetní segment

V této části práce si popíšeme, co je myšleno pojmem výpočetní segment. Jedná se o část výpočetní sítě, která má stejné rozhraní jako uzel (je taktéž potomkem třídy `Node`). Rozdíl je v tom, že se pro definování výpočtu použijí již existující uzly. Vytvoří se část výpočetní sítě, se kterou se pracuje úplně stejně jako s uzlem. Rozdílem oproti uzlu je neomezený počet výstupů.

Vytváření propojení s použitím segmentů je téměř totožné jako s uzly. Jediný rozdíl nastane v případě, že má segment více výstupů. V těchto situací se určí výstup pomocí indexace. Jestliže žádný index není uveden, bere se první výstup segmentu.

```
// Vytvoření vstupního uzlu
var inputNode1 = new Constant<int>().SetConstant(new[] { 42 });

// Vytvoření segmentu, který rozděluje data na dva výstupy
var splitter = new Split<int>(2, inputNode1);

// vytvoření a napojení dvou výstupních uzlů na dva různé výstupy
// segmentu
var printer1 = new Print<int>(splitter[0]);
var printer2 = new Print<int>(splitter[1]);
```

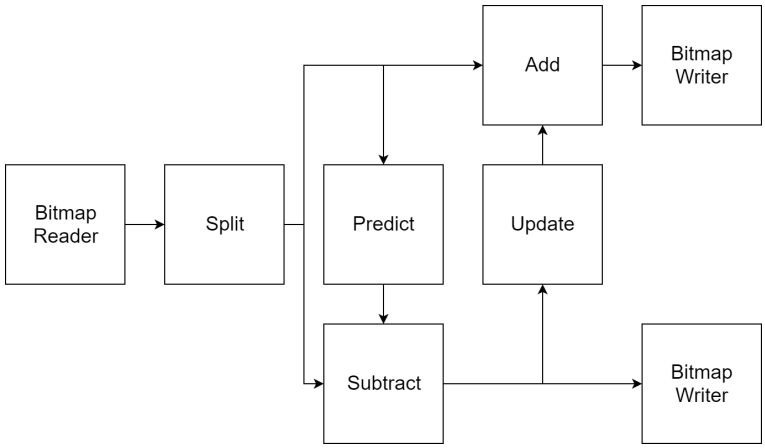
Oproti uzlu výpočetní segment obsahuje navíc následující části.

- `Node<T>[] inputNodes`: Pole vstupních uzlů, které se používají při vytváření segmentu místo vlastnosti `Input`.
- `Node<T>[] outputNodes`: Pole výstupních uzlů, které je třeba při konstrukci segmentu naplnit výstupními uzly segmentu.
- `void BuildSegment()`: Abstraktní metoda, ve které je třeba definovat vytvoření vnitřního zapojení segmentu. V některých složitějších segmentech, kde jsou potřeba dodatečné parametry se nechá tato metoda prázdná a vytvoří se jiná s potřebnými parametry.

Pomocí výpočetních segmentů je možné zjednodušit stavbu výpočetních sítí a odstranit z nich duplicitní části. Příkladem může být Mallatův rozklad, kde se segment reprezentující konkrétní liftingové zapojení vyskytuje několikrát v závislosti na počtu úrovní rozkladu.

3.8 Výpočetní síť

Finální zapojení jednotlivých uzlů a výpočetních segmentů do jednoho celku se nazývá výpočetní síť. Jedná se o zapojení, které provádí všechny kroky výpočtu. Jelikož je výpočet řízen takzvaně od konce, síť se ovládá přes poslední uzel. Pro ukázku je na obrázku 3.3 ilustrování zapojení základní liftingové sítě, kde první uzel čte obrázek z disku a poslední dva výstupní uzly ukládají výsledek na disk.



Obrázek 3.3: Příklad výpočetního uzlu, který sčítá data.

3.9 Tester

V této sekci si popíšeme, jak je možné automaticky spouštět více výpočtů stejné sítě, což se hodí zejména pak u testování a experimentování s novými zapojeními. K tomuto účelu vnikla třída `Tester`, pomocí které je možné automatického spouštění docílit. Jedná se o abstraktní třídy definující metody `void RunSync()` a `void RunAsync()`, které představují spuštění testování. Již v knihovně existují tři konkrétní testovací třídy.

- `ParameterTester<T>`: Třída, pomocí které je možné spouštět danou síť s různými parametry, o kterých byla zmínka v sekci Uzel.
- `RepeateTester<T>`: Tento tester spouští síť opakovaně. Hodí se při randomizovaných algoritmech.
- `MallatTester<T>`: Specifický tester připravený na spouštění komprimace obrázku Mallatovým rozkladem pomocí různých liftingových segmentů.

Při vytváření nové instance testeru je třeba v konstruktoru předat seznam výstupních uzlů, přes které se bude výpočet spouštět. V případě potřeby si uživatel může dopsat vlastní tester, který bude splňovat jeho specifické požadavky. Samotná práce s ním již je velice snadná.

```
// Inicializace potřebných proměnných
string inputName = "../lenna.png";
Bitmap input = new Bitmap(inputName);

// Vytvoření výpočetní sítě
var imageReader = new BitmapReader<float>(input);
var mallat =  new MallatDecomposition
        <float, Haar<float>>(imageReader).Build(input.Size, 1);
var imageWriter = new BitmapWriter<float>(
        new Bitmap(input.Size.Width, input.Size.Height), "", mallat);

// Inicializace testera s˜potřebnými parametry
var tester = new ParameterTester<float>(imageWriter)
        .SetNextParameters(new ParameterStorage()
                .SetParameter("outputName", "../1.png"))
        .SetNextParameters(new ParameterStorage()
                .SetParameter("outputName", "../2.png"))
        .SetNextParameters(new ParameterStorage()
                .SetParameter("outputName", "../3.png"));

// Spuštění výpočtu
tester.RunSync();
```

Toto je příklad použití parametrického testeru. Výpočetní síť počítá Mallatův rozklad druhého stupně pomocí liftingového zapojení Haar, kde pomocí parametrů nastavujeme název výstupního obrázku.

4. Práce s knihovnou

V této kapitole si ukážeme, jak se pracuje s knihovnou. Popíšeme si postup od vytvoření nového uzlu až po vytvoření celé výpočetní sítě. Ukážeme si, co všechno knihovna poskytuje pro uživatele, na co je dobré myslet a dát si pozor. Veškeré části výpočetní sítě (myšleno uzly nebo segmenty) jsou potomkem abstraktní třídy `Node`, aby je bylo možné propojit. V případě segmentu se vytváří potomek třídy `Segment`, který je taktéž potomkem třídy `Node`.

4.1 Uzel

V této části si popíšeme konkrétní příklad uzlu, který je již obsažen v knihovně. Ukážeme si, jak vypadá tvorba jednoduchého výpočetního uzlu a jaké jsou jeho důležité části.

```
public class Subtract<T> : Node<T>
{
        public Subtract(Node<T> node1, Node<T> node2) :
                base(node1, node2) { }
        public override Data<T> GetData(int n)
        {
                var data1 = Input[0].GetData(n);
                var data2 = Input[1].GetData(n);
                return data1 - data2;
        }
        public override Task<Data<T>> GetDataAsync(int n)
        {
                var data1 = Input[0].GetDataAsync(n);
                var data2 = Input[1].GetDataAsync(n);
                return Task.Run(() => {
                        var result = data1.Result - data2.Result;
                        data1 = null;
                        data2 = null;
                        return result;
                });
        }
}
```

Jedná se o uzel `Subtract<T>`, který odečítá od dat z prvního vstupu data z druhého vstupu. Je třeba zdůraznit některé důležité části implementace pro správné porozumění, jak se vytváří nový uzel.

- `: Node<T>`: Každý uzel musí být potomkem abstraktní třídy `Node<T>`, kde parametr `T` značí, jaký typ dat protéká uzlem.
- `: base(node1, node2)`: Je třeba volat konstruktor předka, který se postará o napojení vstupů uzlu.
- `override Data<T> GetData(int n)`: Synchronní verze zpracování dat indexu `n`.

- `override Data<T> GetDataAsync(int n)`: Asynchronní verze zpracování dat index `n`. Aby výpočet probíhal opravdu asynchronně, je třeba k položce `.Result` přistupovat až ve vytvořeném `Tasku`, v opačném případě se sice bude vytvářet nový `Task`, ale bude se zpracovávat synchronně, což je nežádoucí.
- `data1 - data2`: V definování výpočtu je možné používat matematické operátory.
- Jelikož při dotazování na vstupní data používáme index `n`, lze pohodlně zapsat výpočet matematických rovnic.

Jak bylo psáno v předešlé kapitole, není třeba přepisovat všechny virtuální metody pro správnou funkčnost uzlu. Pokud uživatel ví, že nebude chtít využívat asynchronní verzi, nemusí se s její implementací vůbec zabývat. To zkrátí definici uzlu na méně než polovinu.

Je možné definovat stejný uzel pomocí existujícího uzlu `UniversalNode<T>`, kterému se parametry konstruktoru zadají delegáti, určující chování zpracování dat a vstupní uzly. Inicializace stejného uzlu pomocí této metody bude vypadat následovně a používá se zejména u vytváření nové výpočetní sítě nebo segmentu, kde je výpočet specifický a s největší pravděpodobností se nebude využívat nikde jinde. Proto není třeba definovat novou třídu, která by zastávala stejnou funkci. Opět je možné definovat jenom jednu nebo obě verze zpracování dat, takže definice by se razantně zkrátila při používání pouze synchronní verze.

```
var subtruct = new UniversalNode<T>(
        (input, n) => {
                var data1 = input[0].GetData(n);
                var data2 = input[1].GetData(n);
                return data1 - data2;
        }, (input, n) => {
                var data1 = input[0].GetDataAsync(n);
                var data2 = input[1].GetDataAsync(n);
                return Task.Run(() => {
                        var result = data1.Result - data2.Result;
                        data1 = null;
                        data2 = null;
                        return result;
                });
        }, inputNode1, inputNode2);
```

4.2 Výpočetní segment

Zde si popíšeme, jak vytvořit jednoduchý výpočetní segment a jedno liftingové zapojení z již existujících uzlů. Postup je podobný jako u nového uzlu, ale jsou zde jisté rozdíly. Jeden z nich je udávaný počet výstupů, kterých může být i více než jeden.

První z popisovaných segmentů je `Split<T>`, který rozděluje proud dat na `n` výstupů. Nejčastější použití je rozdělení na dva. Ve skutečnosti nejde o opravdové

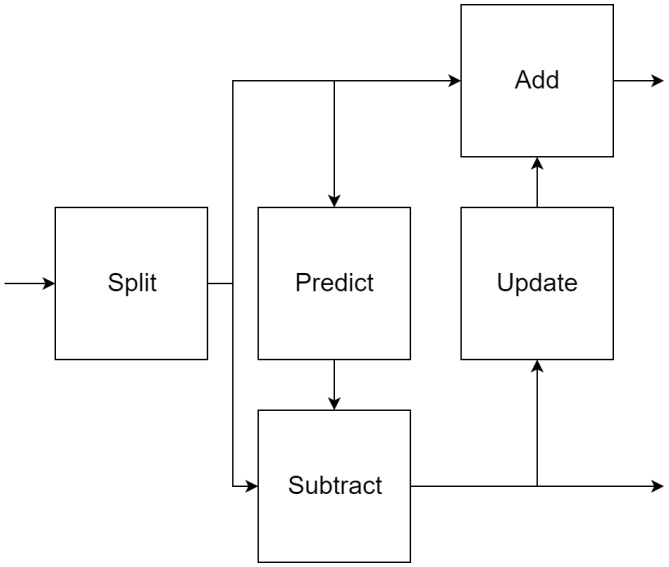
rozdělení toku dat, ale pouze o přepočet indexu dat. Jelikož se v celé knihovně nepracuje s proudem dat v pravém slova smyslu, ale o indexovaná data, tak rozdělení dat na lichá a sudá je ve skutečnosti pouze přepočet indexu podle čísla vstupu a indexu požadovaných dat.

```
public class Split<T> : Segment<T>
{
  public Split(int outputCount, Node<T> predecessor)
    : base(outputCount, predecessor) { }
  public override void BuildSegment()
  {
    var outputCount = outputNodes.Length;
    for (int i = 0; i < outputCount; i++)
    {
      var offset = i;
      outputNodes[i] = new UniversalNode<T>(
        (InputStreamsEnvelope<T>[] input, int dataNumber) =>
        {
          int newDataNumber = dataNumber * outputCount + offset;
          return input[0].GetData(newDataNumber);
        },
        (InputStreamsEnvelope<T>[] input, int dataNumber) =>
        {
          int newDataNumber = dataNumber * outputCount + offset;
          return input[0].GetDataAsync(newDataNumber);
        },
        inputNodes[0]);
    }
  }
}
```

- `: Segment<T>`: Nový segment je potomkem abstraktní třídy `Segment<T>`, která je potomkem `Node<T>`.
- `: base(outputCount, predecessor)`: Volání konstruktoru předka s prvním parametrem určujícím, kolik výstupů bude mít daný segment a s druhým parametrem definujícím jediného předka tohoto segmentu.
- `override void BuildSegment()`: Přepsání jediné abstraktní metody třídy `Segment<T>`, která určuje, jak se sestaví zapojení uvnitř segmentu. Tato metoda se volá na konci konstruktoru předka.
- `outputNodes[i] =`: Přiřazení výstupního uzlu do pole `outputNodes`. Tímto způsobem se musí přiřadit každý výstupní uzel v segmentu.
- `new UniversalNode<T>`: Inicializace univerzálního uzlu, o kterém byla řeč v předešlé sekci. Jeden uzel odpovídá jednomu výstupu segmentu a po přijetí požadavku na nějaká data přepočítá index podle počtu výstupů a indexu dat a pošle požadavek dál.

Další popisovaný segment slouží k diskrétní vlnkové transformaci, která se používá u komprimace obrázků. Schéma zapojení je vidět na obrázku 4.1. V našem

případě je používán nepřímo prostřednictvím segmentu `MallatDecomposition`. Pokud bychom chtěli vytvořit výpočetní segment podle obrázku, kde dva uzly `Predict` a `Update` jsou konkrétní výpočty, popíšeme segment následovně.



Obrázek 4.1: Schéma zapojení výpočetního segmentu reprezentující základní diskrétní vlnkovou transformaci.

```
public class BaseWavelet<T> : Segment<T>
{
  public BaseWavelet(Node<T> predecessor) : base(2, predecessor) { }
  public override void BuildSegment()
  {
    var Split = new Split<T>(2, inputNodes[0]);
    var Predict = new UniversalNode<T>(
      (input, dataNumber) =>
      {
        // načtení potřebných dat
        var data1 = input[0].GetData(dataNumber - 1);
        var data2 = input[0].GetData(dataNumber);
        // provedení výpočtu
        return (data1 + data2) / 2;
      }, (input, dataNumber) =>
      {
        // načtení potřebných Tasků
        var data1 = input[0].GetDataAsync(dataNumber - 1);
        var data2 = input[0].GetDataAsync(dataNumber);
        return Task.Run(() =>
        {
          // přístup k datům a provedení výpočtu
          return (data1.Result + data2.Result) / 2;
        });
         // definování vstupů uzlu
      }, Split[0]);
    var Subtract = new Subtract<T>(Split[1], Predict);
```

```
    var Update = new UniversalNode<T>(
      (input, dataNumber) =>
      {
        // načtení potřebných dat
        var data1 = input[0].GetData(dataNumber - 1);
        var data2 = input[0].GetData(dataNumber);
        // provedení výpočtu
        return (data1 + data2) / 4;
      }, (input, dataNumber) =>
      {
        // načtení potřebných Tasků
        var data1 = input[0].GetDataAsync(dataNumber - 1);
        var data2 = input[0].GetDataAsync(dataNumber);
        return Task.Run(() =>
        {
          // přístup k datům a provedení výpočtu
          return (data1.Result + data2.Result) / 4;
        });
         // definování vstupů uzlu
      }, Subtract);
    var Add = new Add<T>(Split[0], Update);
    outputNodes[0] = Add;
    outputNodes[1] = Subtract;
  }
}
```

Častěji se však dostaneme do situace, kde budeme mít liftingové zapojení popsané pomocí rovnic. Stejný liftingový segment popsaný pomocí rovnic bude vypadat takto:

$$d_{1,l} = s_{0,2l+1} - \frac{s_{0,2l-2} + s_{0,2l}}{2}$$
$$s_{1,l} = s_{0,2l} + \frac{d_{1,l-1} + d_{1,l}}{4}$$

Z těchto rovnic sice je možné zkonstruovat stejný popis segmentu, ale je to podstatně složitější než přímo přepsat tyto rovnice. Stejný segment vytvořený výše popisovaný postupem bude vypadat následovně.

```
public class BaseWavelet<T> : Segment<T>
{
  public BaseWavelet(Node<T> predecessor) : base(2, predecessor) { }
  public override void BuildSegment()
  {
    var d = new UniversalNode<T>(
      (input, dataNumber) =>{
        // načtení potřebných dat
        var data1 = input[0].GetData(2 * dataNumber + 1);
        var data2 = input[0].GetData(2 * dataNumber - 2);
        var data3 = input[0].GetData(2 * dataNumber);
        // provedení výpočtu
        return data1 - ((data2 + data3) / 2);
```

```
      }, (input, dataNumber) => {
        // načtení potřebných Tasků
        var data1 = input[0].GetDataAsync(2 * dataNumber + 1);
        var data2 = input[0].GetDataAsync(2 * dataNumber - 2);
        var data3 = input[0].GetDataAsync(2 * dataNumber);
        return Task.Run(() =>
        {
          // přístup k datům a provedení výpočtu
          return data1.Result - ((data2.Result + data3.Result) / 2);
        });
         // definování vstupů uzlu
      }, inputNodes[0]);
    var s = new UniversalNode<T>(
      (input, dataNumber) => {
        // načtení potřebných dat
        var data1 = input[0].GetData(2 * dataNumber);
        var data2 = input[1].GetData(dataNumber - 1);
        var data3 = input[1].GetData(dataNumber);
        // provedení výpočtu
        return data1 + ((data2 + data3) / 4);
      }, (input, dataNumber) => {
        // načtení potřebných Tasků
        var data1 = input[0].GetDataAsync(2 * dataNumber);
        var data2 = input[1].GetDataAsync(dataNumber - 1);
        var data3 = input[1].GetDataAsync(dataNumber);
        return Task.Run(() =>
        {
          // přístup k datům a provedení výpočtu
          return data1.Result + ((data2.Result + data3.Result) / 4);
        });
         // definování vstupů uzlu
      }, inputNodes[0], d);
    outputNodes[0] = s;
    outputNodes[1] = d;
  }
}
```

4.3 Výpočetní síť

V této části práce se zaměříme na vytváření finální výpočetní sítě. Ukážeme si, jak se pracuje s knihovnou v jazyku C# a dále, jak lze to samé vytvořit pomocí skriptovacího jazyka IronPython. Začneme s prvním ze zmiňovaných jazyků, protože už samotná knihovna je v něm psaná a veškeré doplňování nových uzlů se taktéž píše v jazyce C#. Jelikož se tato knihovna specializuje na liftingové transformace, použijeme základní liftingové zapojení.

Vstupní uzel bude číst obrázek z disku, rozdělí ho na jednotlivé pixely a ty bude posílat do sítě. Jestliže se někdo bude dotazovat na pixel, který se nenachází na obrázku (záporný index nebo index větší než je počet pixelů), vrátí data, která mají stejnou délku jako ostatní a ve všech položkách bude 0. Dru-

hou částí bude již vytvořený segment `BaseWavelet<T>` představující liftingové schéma, které má dva výstupy. Výstupní uzly budou dva a oba budu ukládat obrázky, které budou mít poloviční šířku než obrázek vstupní. I když je liftingové schéma z principu zpracování invertovatelné, námi ukládané obrázky budou pouze náhledy (nebude možné z nich zrekonstruovat původní obrázek), protože výstup výpočtu je racionální číslo, které se nedá uložit do klasického formátu obrázku. Výsledná výpočetní síť již byla vyobrazena na ilustraci 3.3.

```
// inicializace potřebných proměnných
string inputName = "lenna";
Bitmap input = new Bitmap(inputName + ".png");
Bitmap outputAproximation = new Bitmap(input.Width/2,
                                       input.Height,
                                       input.PixelFormat);
Bitmap outputDetail = new Bitmap(input.Width/2,
                                 input.Height,
                                 input.PixelFormat);

// vytvoření jednotlivých uzlů
var imageReader = new BitmapReader<float>(input);
var baseWavelet = new BaseWavelet<float>(imageReader);
var imageWriterAproximation =
    new BitmapWriter<float>(outputAproximation,
                            inputName + "_aproximation.png",
                            baseWavelet[0]);
var imageWriterDetail =
    new BitmapWriter<float>(outputDetail,
                            inputName + "_detail.png",
                            baseWavelet[1]);

// spuštění výpočtu na posledních uzlech výpočetní sítě
imageWriterAproximation.Process();
imageWriterDetail.Process();
```

Je třeba popsat nějaké části kódu pro lepší porozumění, jak se vytváří výpočetní síť.

- `new BitmapReader<float>(input)`: Ve špičatých závorkách se uvádí typ dat, které se budou posílat mezi uzly. U vstupního uzlu to také znamená, jaký typ dat má vytvářet. `input` je u tohoto konstruktoru instance třídy `Bitmap`, ze kterého se čtou jednotlivé pixely.
- `baseWavelet[0]`: Pomocí hranatých závorek se určí, který výstup segmentu se má použít. Pokud se index nepoužije, automaticky se pracuje s prvním výstupem. Indexy výstupů jdou použít i u uzlů, ale tam použití postrádá význam, protože uzly mohou mít pouze jeden výstup a při snaze přistoupit k jinému dojde k chybě.
- `.Process()`: Spuštění synchronního výpočtu na výstupním uzlu. V tomto konkrétním případě jsou výstupy odděleny, takže je možné případně počítat pouze jeden z výstupů.

Jak bylo zmíněno na začátku této sekce, definování a spouštění výpočetních sítí lze i pomocí jazyka IronPython. Kód vykonávající stejný výpočet by se zapsal v tomto jazyce následovně.

```
// tato část se stará o naimportování potřebných knihoven
// krom té naší i třídy, které budeme používat
import clr
clr.AddReferenceToFileAndPath('EnvironmentForLifting.dll')
from EnvironmentForLifting import *
clr.AddReference('System.Drawing')
from System.Drawing import Bitmap

// inicializace potřebných proměnných
level = 2
inputName = "../lenna"
input = Bitmap(inputName + ".png")
outputAproximation = Bitmap(input.Width/2, input.Height,
    input.PixelFormat)
outputDetail = Bitmap(input.Width/2, input.Height, input.PixelFormat)

// vytvoření jednotlivých uzlů
imageReader = BitmapReader[float](input)
baseWavelet = BaseWavelet[float](imageReader)
imageWriterAproximation = BitmapWriter[float](outputAproximation,
    inputName + "_aproximation.png", baseWavelet[0])
imageWriterDetail = BitmapWriter[float](outputDetail,
    inputName + "_detail.png", baseWavelet[1])

// spuštění výpočtu
imageWriterAproximation.Process()
imageWriterDetail.Process()
```

Můžeme si všimnout lehkých rozdílů v syntaxi, jako je třeba použití hranatých závorek místo špičatých pro definování typu uzlu nebo absenci definování typu proměnných. Ovšem z celkového hlediska je kód téměř totožný, takže není pro uživatele problém se naučit používat oba jazyky a vybrat si na daný problém ten vhodnější.

5. Implementované části

V následující kapitole si popíšeme konkrétní příklady již implementovaných částí (uzlů a segmentů) v knihovně. Ukážeme si uzly pro práci s obrázky, speciálním formátem PFM pro ukládání obrázku s reálnými hodnotami barev a v neposlední řadě již vytvořené liftingové zapojení.

5.1 Základní části

Začneme se základními části, které se využívají téměř v každé výpočetní síti, a proto je třeba znát jak pracují. Nebudeme si zde rozebírat celý kód implementace, ale popíšeme si, jak daný uzel či segment pracuje a případně upozorníme na nějaké podstatné části.

5.1.1 UniversalNode

Prvním z představených uzlů bude univerzální uzel. To je takový uzel, kterému je třeba v konstruktoru předat delegáty, které teprve definují, jak bude probíhat výpočet uzlu. Nejčastější využití tohoto uzlu je při definování nového segmentu nebo celé výpočetní sítě. Konkrétně u definování takových výpočetních uzlů, kde není moc pravděpodobné, že by se podobný uzel mohl využít i někde jinde.

Příkladem takového použití je převedení následující rovnice do výpočetního uzlu, který bude umět pouze synchronní verzi výpočtu.

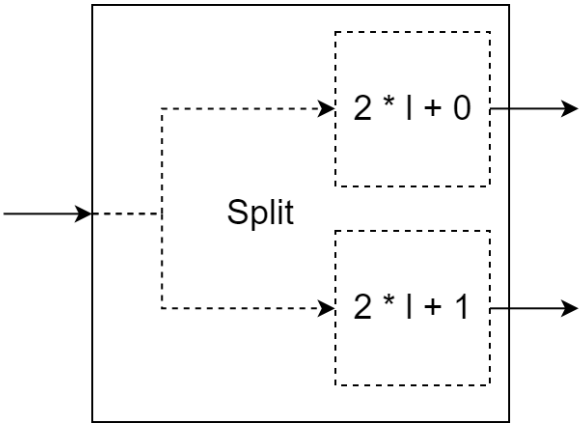
$$out_l = \frac{in_{l-1} + in_{l+1}}{2}$$

```
var out = new UniversalNode<T>(
  (in, l) =>
  {
    var data1 = in[0].GetData(l - 1);
    var data2 = in[0].GetData(l + 1);
    return (data1 + data2) / 2;
  }, inputNode);
```

5.1.2 Split

Častým případem ve výpočetních sítích je situace, kde je třeba rozdělit data na sudá a lichá. Obecně na každá `n`-tá data. K tomu slouží uzel `Split`, kterému se v konstruktoru krom předcházejícího uzlu zadá i počet výstupů. Interně přepočítává index dat a posílá požadavek o uzel blíže k vstupu.

Opačným uzlem ke `Split` je `Merge`, který se naopak stará seřazení dat z více vstupů do jednoho. Taktéž funguje pouze pomocí indexů, kde se podle požadavku spočítá, ze kterého vstupu na jaký index se má dotazovat.



Obrázek 5.1: Vnitřek uzlu Split pro 2 výstupy.

5.2 Bitmapové uzly

V další části si popíšeme uzly, které ukládají nebo jinak pracují s obrázky. Jelikož se liftingové zapojení nejčastěji používají ke zpracování obrázků, je dostupnost předpřipravených uzlů pro jejich zpracování nutná.

5.2.1 BitmapReader

Začneme s uzlem, který čte obrázek z disku a rozdělují ho na jednotlivé pixely. Podle formátu se určí, kolik barevných kanálů se bude posílat. Již při inicializaci uzlu se načtou data obrázku do paměti, protože to je rychlejší způsob než se pokaždé dotazovat na data konkrétního pixelu.

5.2.2 BitmapWriter

Uzel, který se naopak stará o ukládání pixelů do obrázku na disk se jmenuje `BitmapWriter`. Je třeba upozornit, že všechny dostupné formáty neumožňují ukládání reálných hodnot. Z toho důvodu bereme obrázky uložené tímto uzlem pouze jako náhledy, jelikož by se z nich nedala původní data zrekonstruovat. Bylo třeba vyřešit situaci, jak se zachovat v případě, že hodnota, kterou se snažíme uložit je mimo rozsah hodnot. Výpočtem se velice snadno může stát, že výsledek bude záporný nebo větší než `byte`. My jsme se rozhodli k oříznutí hodnot, jelikož nějakou informaci ztrácíme už tím, že můžeme uložit pouze celočíselné hodnoty. Záporné hodnoty namapujeme na 0 a větší hodnoty než je limit na 255. Další z možností bylo použití operace modulo, ale to by způsobilo šum a náhled by moc neodpovídal realitě.

5.2.3 ImageDiagonalFlipper

Dva předcházející uzly berou pořadí pixelů po řádcích od shora dolů. V některých případech, jako třeba u Mallatova rozkladu, je třeba číst pixely nikoli po řádcích, ale po sloupcích zleva doprava. Abychom tohoto chování docílili, vytvořili jsme uzel `ImageDiagonalFlipper`, kterému je třeba zadat rozlišení obrázku, aby mohl správně přepočítávat index dat.

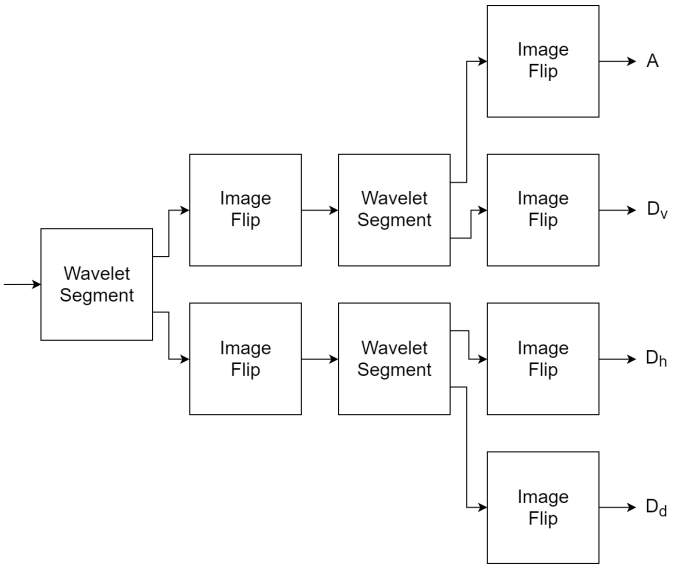
5.2.4 MallatDecomposition

Hlavním použitím liftingových zapojení je Mallatův rozklad o kterém byla řeč v kapitole 1.7. Bylo třeba připravit segment, kterému by bylo možné nějak předat, jaký liftingový segment se má použít a kolik úrovní rozkladu chceme vytvořit. Problém spočíval v tom, jak správně zadat typ segmentu a jak ho zkonstruovat. Tuto nesnázi jsme obešli použitím třídy `Activator` dostupný v jazyce C#, který využívá `Reflections` k zavolání konstruktoru. To má sice dopad na rychlost, ale s ohledem na to, že zkonstruování výpočetní sítě je zanedbatelné oproti samotnému výpočtu, nechali jsme toto řešení.

Předpokládejme, že již máme napsaný liftingový segment `Haar<T>` a chceme pomocí něj vytvořit Mallatův rozklad 2. úrovně. Kód zkonstruování bude vypadat následovně.

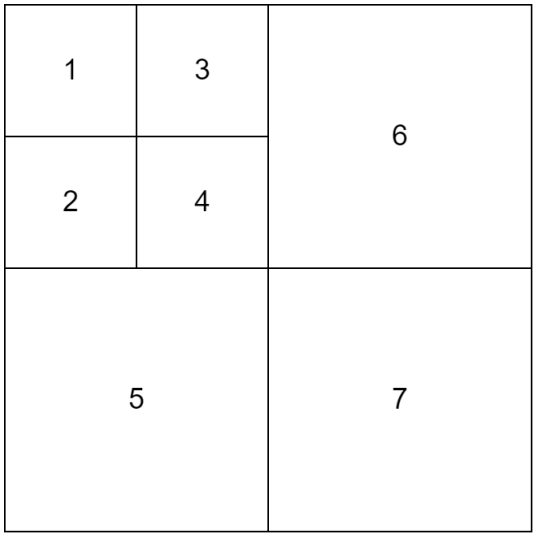
```
Bitmap image;
int levels = 2;
Node<float> predecesorNode;
var mallat = new MallatDecomposition<float, Haar<float>>
    (predecesorNode).Build(image.Size, levels);
```

Segment `mallat` bude mít `x` výstupů, kde $x = (levels - 1) * 3 + 4$. Pro dvě úrovně je pořadí výstupů vyobrazeno na ilustraci 5.3 a konkrétní příklad na obrázku 5.4. Je to z toho důvodu, že každý oddíl rozkladu bude mít vlastní výstup. Jestliže bychom chtěli mít výstup jeden a to seřazené pixely zleva doprava, napojíme na `mallat MallatMerger`. Jedna úroveň Mallatova rozkladu je na obrázku 5.2 a v případě více úrovní se vždy na výstup `A` napojí stejné zapojení. Rozlišení výstupního obrázku musí být dělitelné 2^{levels}, takže se musí případně rozšířit.

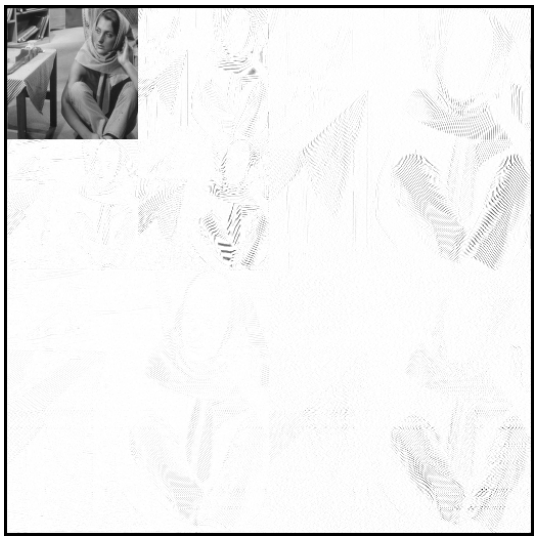


Obrázek 5.2: Výpočetní segment MallatDecomposition.

Inverzní zapojení Mallatova rozkladu je `MallatDecompositionInverse` a konstruuje se stejně jako dopředný rozklad. Vstup má pouze jeden, takže kdyby někdo chtěl zapojit nejprve rozklad a poté inverzní operaci, je třeba mezi tyto dva segmenty napojit `MallatMerger`.



Obrázek 5.3: Pořadí výstupů Mallatova rozkladu 2. úrovně.



Obrázek 5.4: Ukázka Mallatova rozkladu 2. úrovně pomocí liftingového zapojení 9-7 s invertovanými detaily a rámečkem pro lepší viditelnost.

5.2.5 MallatEntropyEvaluator

Jeden z možných testů na kvalitu rozkladu je výpočet entropie výsledných dat podle vzorce z kapitoly 1.8. K tomuto účelu slouží uzel `MallatEntropyEvaluator`, který entropii dat vypisuje na konzoli.

5.3 PFM uzly

Jak jsme si již psali u uzlů zpracovávající obrázky, pomocí normálních formátů není možné uložit výsledek Mallatova rozkladu tak, aby z něho bylo možné opět spočítat originál. Z toho důvodu jsme vytvořili vstupní a výstupní uzel, pracující s formátem PFM. To je formát pro uložení obrázku pomocí typu `float` a díky tomu jsme schopni uložit neceločíselné i záporné hodnoty, které mohou výpočtem vzniknout. Pokud si chce uživatel prohlédnout takto vytvořený soubor, nestačí

k tomu obyčejný prohlížeč obrázků, ale musí použít nějaký specializovaný.

5.4 Liftingové segmenty

Liftingová zapojení můžeme rozdělit na dva druhy. Normální, kde se na výsledek neklade žádný speciální nárok a na celočíselné, kde vyžadujeme, aby ze vstupu celých čísel opět vznikla celá čísla. Tento druh je nejčastěji používaný u zpracovávání obrázků, protože chceme využívat již existujících formátů pro uložení. Je také méně náročný na velikost. V této sekci budeme využívat popsané liftingové zapojení ze dvou zdrojů a to ze článku Daubechies a Sweldens (1998) a ze článku Calderbank a kol. (1998). `BaseWavelet` jsem si již představili a popsali v kapitole 4.2, proto postoupíme ke složitějšímu zapojení a to k `DaubechiesWaveletD4`. Ten je definován následujícími rovnicemi.

$$
\begin{aligned}
d_l^{(1)} &= x_{2l+1} - \sqrt{3}x_{2l} \\
s_l^{(1)} &= x_{2l} + \frac{\sqrt{3}}{4}d_l^{(1)} + \frac{\sqrt{3}-2}{4}d_{l+1}^{(1)} \\
d_l^{(2)} &= d_l^{(1)} + s_{l-1}^{(1)} \\
s_l &= \frac{\sqrt{3}+1}{\sqrt{2}}s_l^{(1)} \\
d_l &= \frac{\sqrt{3}-1}{\sqrt{2}}d_l^{(2)}
\end{aligned}
$$

Pomocí těchto rovnic je vytvořen liftingový segment stejným způsobem jako je nastíněno v kapitole 4.2.

Nyní se přesuneme k druhému jmenovanému druhu liftingových zapojení a to k celočíselným. Začneme ukázkou rovnic pro celočíselný `DaubechiesWaveletD4`, který se v této práci označuje zkráceně `D4`.

$$
\begin{aligned}
d_l^{(1)} &= x_{2l+1} - \lfloor \sqrt{3}x_{2l} + \frac{1}{2} \rfloor \\
s_l^{(1)} &= x_{2l} + \lfloor \frac{\sqrt{3}}{4}d_l^{(1)} + \frac{\sqrt{3}-2}{4}d_{l-1}^{(1)} + \frac{1}{2} \rfloor \\
d_l^{(2)} &= d_l^{(1)} + s_{l+1}^{(1)} \\
s_l &= (\sqrt{3}+1)s_l^{(1)} \\
d_l &= \frac{1}{\sqrt{3}+1}d_l^{(2)}
\end{aligned}
$$

Konstruování takovýchto zapojení je téměř totožné jako u minulých příkladů s jedinou výjimkou. Bylo potřeba nějak zajistit, aby se správně počítaly hodnoty zaokrouhlení. Musíme mít na paměti, že při použití `Data<int>` bude mezivýsledek zaokrouhlován po každé operaci. K tomu dochází z důvodu, že obecně jakékoli operace s instancemi `Data<T>` vždy vede opět k `Data<T>`. Proto je potřeba v místech, kde by mohlo dojít k tomuto nechtěnému chování, přetypovat instance dat na `double` pomocí funkce `Data<T>.ToDouble(data)` a poté použít funkci `Data<double>.FloorWithHalfAdded(data)` k zaokrouhlení nakonec použít funkci `Data<T>.DoubleToT(data)` k opětovnému přetypování na `T`.

Příklad použití si ukážeme na dalším zapojení a to interpolační transformaci `(2,2)` definované následovně.

$$d_l = x_{2l+1} - \lfloor \frac{1}{2}(x_{2l} + x_{2l+2}) + \frac{1}{2} \rfloor$$
$$s_l = x_{2l} + \lfloor \frac{1}{4}(d_{l-1} + d_l) + \frac{1}{2} \rfloor$$

Pro jednoduchost si ukážeme pouze synchronní verze výpočtu při popisování segmentu v metodě `BuildSegment()`.

```
var d = new UniversalNode<T>(
(input, dataNumber) =>
{
    // získání dat pro výpočet a jejich přetypování na double
    var data1 = Data<T>.ToDouble(input[0].GetData(2 * dataNumber + 1));
    var data2 = Data<T>.ToDouble(input[0].GetData(2 * dataNumber));
    var data3 = Data<T>.ToDouble(input[0].GetData(2 * dataNumber + 2));

    // výpočet s˜použitím zaokrouhlení
    var result = data1 -
        Data<double>.FloorWithHalfAdded((1/2d) * (data2 + data3));

    // přetypování na T a vrácení výsledku
    return Data<T>.DoubleToT(result);
, inputNodes[0]);
var s˜= new UniversalNode<T>(
(input, dataNumber) =>
{
    // získání dat pro výpočet a jejich přetypování na double
    var data1 = Data<T>.ToDouble(input[0].GetData(2 * dataNumber));
    var data2 = Data<T>.ToDouble(input[1].GetData(dataNumber - 1));
    var data3 = Data<T>.ToDouble(input[1].GetData(dataNumber));

    // výpočet s˜použitím zaokrouhlení
    var result = data1 +
        Data<double>.FloorWithHalfAdded((1/4d) * (data2 + data3));

    // přetypování na T a vrácení výsledku
    return Data<T>.DoubleToT(result);
, inputNodes[0], d);
outputNodes[0] = s;
outputNodes[1] = d;
```

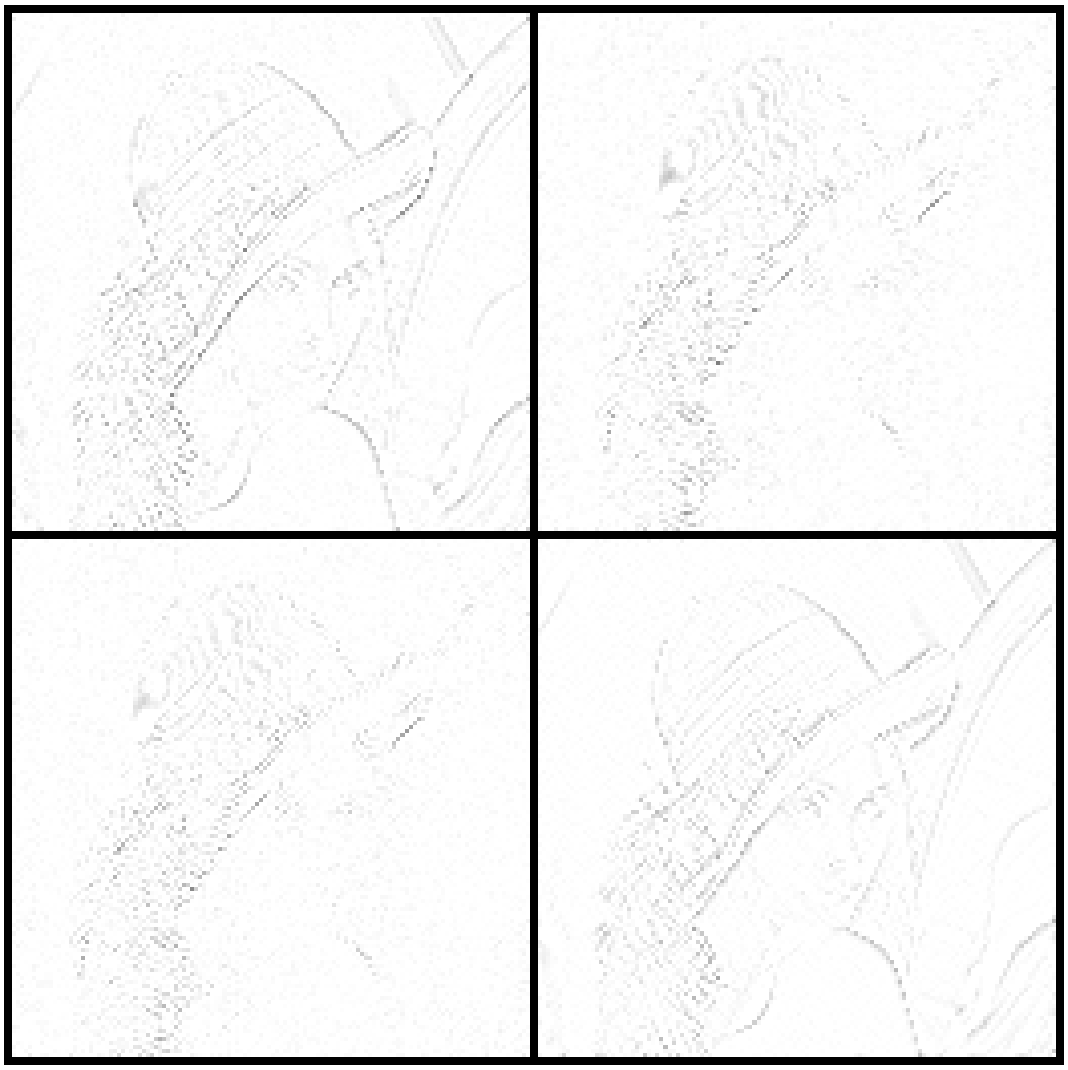
Po přidání konstruktoru se jedná o celou definici liftingového segmentu, který již lze používat.

Stejným způsobem jsme implementovali ještě celočíselné liftingové zapojení `(4,2)`, `(2,4)`, `(4,4)`, `(2+2,2)` a `(9-7)`, které jsou popsané v článku Calderbank a kol. (1998). V poslední kapitole 6.4 budeme tyto implementované segmenty využívat a budeme porovnávat jejich kvality.

5.5 Ukázka výstupu

Pro představu čtenáře si v této části práce ukážeme zpracované obrázky pomocí různých liftingových zapojení představených v Calderbank a kol. (1998). Aby byla nějaká změna patrná, invertovali jsme barvy a vyřízli oblast číslo 4 (5.3) z výsledku Mallatova rozkladu.

Obrázek 5.5: Originál obrázku "lenna".



Obrázek 5.6: Oblast 4 Mallatova rozkladu dvou úrovní čtyřech různých liftingových zapojení. Zleva jde o D4, následuje interpolace (4,2), poté v dalším řádku pokračuje (9-7) a nakonec (2+2,2). U všech jsou invertovány barvy.

6. Experimenty

V této poslední kapitole si ukážeme výsledky různých experimentů související s implementací knihovny a s výpočetními sítěmi vytvořené pomocí této knihovny. Díky výsledkům těchto experimentů můžeme dojít k závěru, které z navhovaných řešení je nejvhodnější.

6.1 Expressions

První z experimentů bude zaměřen na rychlost matematických operací podle způsobu implementace třídy `Data`. Přišli jsme se třemi návrhy implementace. První z možností bylo pevně nastavit typ dat, které se budou mezi uzly posílat. Jedná se o nejtriviálnější implementaci a teoreticky bychom měli dojít k nejvyšší rychlosti operací. Na druhou stranu by to omezovalo uživatele ve volbě konkrétních dat, která lze používat. Druhou možností je použití přetypování na `dynamic`, což by značně ulehčilo práci programátora této knihovny při implementaci generických operací. Daň za toto použití by byla pravděpodobně znatelné snížení rychlosti a ztráty typové kontroly za překladu. Poslední z možností bylo využít `Expressions`, který je dostupný v jazyce C# a ručně připravit veškeré možné matematické operace. Předpokládali jsme, že toto řešení by mělo být podstatně rychlejší než varianta druhá, ale za cenu podstatně většího množství kódu, který jsme museli napsat.

Pro otestování vlastností jsme napsali metodu, která počítá dvě různé matematické operace na velkém množství dat pomocí všech třech vyjmenovaných řešení. Konkrétně jsme použili 10^9 výpočtů pro každý z řešení. Počítali jsme sčítání dvou instancí dat a dělení dat hodnotou typu `int`. Pro snadnější čtení výsledků jsme použili jednotku `MOPS`, což značí počet milionů operací za sekundu.

10^9 operací	data + data	data / int	koeficient
Native	73 MOPS	73 MOPS	1.00x
Expressions	51 MOPS	56 MOPS	0.77x-0.70x
Dynamic	15 MOPS	23 MOPS	0.32x-0.21x

Tabulka 6.1: Rychlost operací podle způsobu implementace.

Z tabulky 6.1 lze vyčíst, že náš předpoklad byl správný a použití `Expressions` se nám vyplatilo podstatně vyšší rychlostí než pomocí `dynamic`. Oproti předem stanovenému typu dat nám poskytuje generičnost za cenu zpomalením o $0.23-0.3$. Usoudili jsme, že možnost volby typu je pro uživatele přínosnější, jelikož se jedná o testovací knihovnu, nikoli o knihovnu cílenou k co nejvyššímu výkonu.

6.2 Rychlost výpočtu podle použitého typu

Další z experimentů, které jsme provedli bylo měření závislosti rychlosti matematických operací na zvoleném typu generických dat. Jelikož jsme se rozhodli k použití generických dat místo pevně zvoleného typu, je třeba ukázat, jaký bude mít vliv na výkon zvolení daného typu. Stejně jako v minulém experimentu jsme

zvolili dvě matematické operace, a to sčítání dvou instancí dat a dělení dat typem `int`. Opět používáme jednotku `MOPS`, což značí milión operací za sekundu.

10^9 operací	data + data	data / int
byte	60 MOPS	61 MOPS
sbyte	61 MOPS	61 MOPS
short	59 MOPS	62 MOPS
ushort	60 MOPS	62 MOPS
int	61 MOPS	63 MOPS
uint	61 MOPS	61 MOPS
long	50 MOPS	44 MOPS
ulong	50 MOPS	46 MOPS
float	56 MOPS	61 MOPS
double	49 MOPS	56 MOPS
decimal	23 MOPS	18 MOPS

Tabulka 6.2: Rychlost operací podle použitém typu.

Z výsledků lze vyčíst, že na použitém typu zas tak moc nezáleží. Určitě je vhodné vyvarovat se typu `decimal`, ale ten se využívá pouze v naprosto speciálních případech. Dále je vhodné nepoužívat `long` a `ulong`, jelikož je o trochu pomalejší než `int` nebo `float`.

6.3 Synchronního a asynchronního výpočet

Knihovnu jsme připravili na možnost spuštění sériového i paralelního výpočtu. Veškeré předpřipravené části v knihovně, ať už jde o uzle nebo segment, mají implementované obě verze výpočtu, takže je možné stejnou výpočetní síť pustit oběma způsoby a porovnat dosažených časů. Princip použití vícevláknového výpočtu jsme si popsali v kapitole 3.3. Naše obava, že asynchronní výpočet nebude rychlejší než synchronní, pramení z myšlenky, že čas potřebný pro vytvoření jednoho `Task`u bude nakonec větší, než jeho přínos. Pro porovnání jsme se rozhodli pro reálné příklady zpracování obrázků. Konkrétně o spuštění Mallatova rozkladu pomocí 4 různých liftingových zapojení na obrázku 5.5.

	synchronní výpočet	asynchronní výpočet	koeficient
D4	2.417s	4.970s	0.486x
(4,2)	2.522s	4.631s	0.545x
(9-7)	3.672s	7.806s	0.470x
(2+2,2)	3.290s	6.395s	0.514x

Tabulka 6.3: Rychlost výpočtu podle způsobu počítání.

Z experimentu vyplývá, že použití asynchronního výpočtu se nevyplatí, protože je dvakrát pomalejší než synchronní. Pro potvrzení názoru jsme udělali další experiment a to změřit dobu inicializace 10^9 dat různými způsoby. První způsob byl přímá inicializace typu `Data<double>`, druhá bylo vytvoření instance `Task` pomocí metody `FromResult` a poslední bylo vytvoření celého `Task`u.

10^9 dat	potřebný čas	koeficient
new Data	10.761s	1.000x
Task.FromResult()	16.372s	0.657x
new Task	57.386s	0.188x

Tabulka 6.4: Rychlost inicializace dat.

Z výsledků můžeme vyčíst, že použití `Tasků` není úplně zadarmo a je třeba přijít se způsobem, jak co nejvíce omezit počet vytvářených `Tasků`, aby cena za vytvoření nebyla vyšší než přínos. Proto jsme přišli s ještě jedním experimentem, kde budeme vytvářet `Task` až ve výstupním uzlu. V těchto `Task`ach výpočet probíhat synchronně. To by mělo značně omezit počet vytvářených `Tasků` a teoreticky by to mělo zajistit chtěné zrychlení výpočtu. Pro tento test jsme museli přidat zamykání kolekce ve třídě `DataStorageList`, protože původně třída nebyla připravena pro asynchronní zápis. V tabulce budeme tento způsob značit `Task v2`.

	synchronní výpočet	Task v2	koeficient
D4	2.437s	0.753s	3.236x
(4,2)	2.480s	0.505s	4.911x
(9-7)	3.750s	1.174s	3.194x
(2+2,2)	3.322s	0.542s	6.130x

Tabulka 6.5: Rychlost výpočtu jiným způsobem používání Tasků.

Experimentem se nám potvrdil předpoklad, že je rozumnější použití `Tasků` až ve výstupním uzlu, kde zbytek sítě je psán pro jednovláknové zpracování. Z pohledu uživatele jde o veliký klad, protože v případě psaní nových částí, může se omezit pouze na implementování synchronních verzí a o vícevláknové zpracování je třeba se starat až ve výstupních uzlech. Proto jsme všechny výstupní uzly připravené v knihovně přepsali tak, aby fungovaly podle nového způsobu asynchronního zpracování.

6.4 Entropie liftingových zapojení

V posledním experimentu již budeme používat vytvořenou knihovnu a pomocí ní otestujeme kvalitu různých liftingových zapojení představených v kapitole 5.4 podle dosažené entropie 1.8. Pomocí těchto zapojení jsme vytvořili Mallatův rozklad různých úrovní a vypočítali entropii dat. Jako testovací data nám poslouží standardní obrázky často používané u testování komprimačních algoritmů.

Z tabulek 6.6 lze vyčíst, že liftingové zapojení `(9-7)` je s přibývajícím počtem úrovní vzhledem k dosažené entropii dat nejlepší. Je ale třeba vzít v potaz i rychlost výpočtu. U úrovní `1` a `2` jsou rozdíly v rychlosti malé, u `3` úrovní již začíná být výpočet `(9-7)` zoufale pomalý (6.7) a dosažené výsledky nejsou o tolik lepší. Z toho důvodu bych doporučil použít u malého počtu úrovní právě zmiňovaný `(9-7)`, ale u vyššího počtu úrovní bych se přikláněl k použití `(4,2)`, který nemá o tolik horší entropii dat, ale je podstatně rychlejší.

level = 1	barbara	boat	finger.	flin.	house	lenna	pepper.
(2,2)	4.208	4.014	4.090	4.539	3.383	3.582	3.838
(4,2)	4.130	3.986	3.925	4.498	3.404	3.568	**3.780**
(2,4)	4.198	4.010	4.101	4.547	3.392	3.588	3.844
(4,4)	4.220	4.137	**3.872**	4.498	3.427	3.637	3.934
(2+2,2)	4.815	4.414	5.457	5.055	3.932	4.143	4.451
(D4)	4.820	4.514	5.435	5.183	3.928	4.166	4.493
(9-7)	**4.082**	**3.914**	3.934	**4.423**	**3.383**	**3.510**	3.814
(Base)	5.016	4.744	5.749	5.473	4.156	4.404	4.735
level = 2	barbara	boat	finger.	flin.	house	lenna	pepper.
(2,2)	4.910	4.672	4.945	5.354	3.955	4.138	4.471
(4,2)	4.816	4.634	4.742	5.305	3.974	4.108	4.497
(2,4)	4.890	4.679	4.971	5.377	3.979	4.153	4.489
(4,4)	4.988	4.818	4.729	5.375	4.023	4.222	4.592
(2+2,2)	**4.600**	5.156	6.502	6.022	4.606	4.812	5.222
(D4)	5.576	5.230	6.416	6.081	4.578	4.814	5.260
(9-7)	4.683	**4.503**	**4.697**	**5.168**	**3.908**	**3.990**	**4.382**
(Base)	5.919	5.609	6.933	6.534	4.932	5.193	5.633
level = 3	barbara	boat	finger.	flin.	house	lenna	pepper.
(2,2)	5,091	4,864	5,213	5,598	4,123	4,305	4,667
(4,2)	4,989	4,820	5,005	5,545	4,140	4,268	4,689
(2,4)	5,069	4,874	5,244	5,625	4,150	4,323	4,689
(4,4)	5,193	5,018	5,002	5,635	4,203	4,397	4,793
(2+2,2)	5,816	5,359	6,765	6,284	4,794	5,003	5,442
(D4)	5,775	5,420	6,645	6,318	4,754	4,995	5,472
(9-7)	**4,831**	**4,666**	**4,939**	**5,384**	**4,060**	**4,131**	**4,556**
(Base)	6,168	5,855	7,235	6,829	5,152	5,425	5,888

Tabulka 6.6: Entropie výsledných dat vzhledem k použitému obrázku a liftingovém zapojení.

level = 1	barbara	boat	finger.	flin.	house	lenna	pepper.
(4,2)	4.50s	4.03s	4.02s	4.04s	1.08s	3.94s	0.97s
(9-7)	3.53s	3.42s	3.86s	3.18s	0.90s	3.34s	0.95s
level = 2							
(4,2)	4.95s	4.95s	4.83s	4.74s	1.35s	4.80s	1.25s
(9-7)	4.92s	5.00s	4.99s	4.66s	1.38s	4.66s	1.52s
level = 3							
(4,2)	5.50s	5.60s	5.50s	5.37s	1.78s	5.44s	1.84s
(9-7)	675s	683s	725s	605s	743s	708s	751s

Tabulka 6.7: Entropie výsledných dat vzhledem k použitému obrázku a liftingovém zapojení.

Závěr

V rámci této práce byla vytvořena knihovna EnvironmentForLifting, která umožňuje jednoduchou tvorbu výpočetních sítí. Pod tímto pojmem si lze představit algoritmy, které lze rozčlenit na jednoduché transformace vstupního signálu (uzly) a cesty po kterých se přesouvají mezivýsledky. Příkladem takovéto výpočetní sítě jsou šifrovací algoritmy, u kterých jsou přesně definované jednotlivé elementární kroky. Knihovna je připravena na transformaci více-dimenzionálních signálů, příkladem může být zpracování více barevných kanálů při komprimaci obrázku. Hlavní inspirací pro knihovnu byla snadná tvorba liftingových transformací a následné experimentování s těmito zapojeními.

Dále bylo vytvořeno několik výpočetních sítí pro potvrzení funkčnosti vypracovaného systému. Výše zmiňovaná knihovna EnvironmentForLifting poskytuje kostru těchto sítí spolu s implementovanými základními uzly a výpočetními segmenty. V oblasti komprimace obrázků bylo vytvořeno celkem deset různých výpočetních segmentů založených na liftingu.

V neposlední řadě byly uskutečněny experimenty, které ukazují funkčnost některých konceptů zapracovaných do knihovny, jako je třeba zajištění matematický operátorů pro generická data. Dalším experimentem byla ukázka rozdílu výkonu počítání na základě použitého typu dat, což se může hodit budoucím uživatelům, kteří se budou rozhodovat, jestli se jim vyplatí použití nějakého typu nebo ne. Změření rozdílu výkonu mezi synchronním a asynchronním výpočtem nám ukázal, že bylo vhodné pozměnit použití `Task`ů pro dosažení vyššího výkonu. V posledním experimentu jsme již použili funkční knihovnu a změřili jsme entropii dat získaných výpočtem pomocí různých liftingových zapojení.

Vypracované téma dává příležitost k jistým rozšířením této knihovny. Jedním z nich by mohla být grafická nástavba, která by ukazovala aktuální zapojení. Dalším možným vylepšením by mohlo být přepracování ukládání výsledků do obrázku, aby ukládání do různých formátů mělo stejné rozhraní pro uživatele.

Seznam použité literatury

Calderbank, A., Daubechies, I., Sweldens, W. a Yeo, B.-L. (1998). Wavelet transforms that map integers to integers. *Applied and Computational Harmonic Analysis*, **5**(3), 332 – 369. ISSN 1063-5203. doi: https://doi.org/10.1006/acha.1997.0238. URL `http://www.sciencedirect.com/science/article/pii/S1063520397902384`.

Daubechies, I. a Sweldens, W. (1998). Factoring wavelet transforms into lifting steps. *J. Fourier Anal. Appl*, **4**, 247–269.

Mallat, G. (1989). A theory for multiresolution signal decomposition : the wavelet representation. *IEEE Transaction on Pattern Analysis and Machine Intelligence*.

Rolón, J. C. a Salembier, P. (2007). Generalized lifting for sparse image representation and coding. In *Picture Coding Symposium*, page 4. Lisbon.

Sweldens, W. (1997). The lifting scheme: A construction of second generation wavelets.

Sweldens, W. a kol. (1995). The lifting scheme: A new philosophy in biorthogonal wavelet constructions. *Wavelet Applications in Signal and Image Processing*, **3**, 68–79.

Seznam zdrojů obrázků

Odkazy na zdroje použitých obrázků v této práci platné k 18.07.2017.

Obrázek 1.1
https://commons.wikimedia.org/wiki/File:GLScheme.png

Obrázek 1.2
https://www.mathworks.com/help/wavelet/gs/discrete-wavelet-transform.html

Obrázek 1.3
https://www.pexels.com/photo/nature-summer-purple-yellow-36753

Přílohy

Přílohou této práce je CD obsahující:

- Zdrojové kódy knihovny, testů a příklady použití
- Dokumentace ke zdrojovým kódům vygenerovaná nástrojem `Doxygen`.
- Adresář s použitými obrázky a výsledky.
- Text práce ve formátu `PDF`.