Implementace algoritmu IDW a jeho vizualizace na GPU

Vladislav Trnka, 2021

Abstrakt

Algoritmus IDW – Inverse distance weighting, česky vážení inverzní vzdáleností – je metoda na aproximaci nějaké hodnoty v závislosti na uzlových bodech. Každý uzlový bod přispívá do celkového výsledku na základě inverzní vzdálenosti k hledanému bodu. Metoda vytváří hladké přechody mezi na uzlovými body, a proto je často využívána v rámci geografie. V rámci práce byl vytvořen framework pro kompletní vizualizaci algoritmu a jeho fungování. Obsahuje implementaci na různých výpočetních platformách včetně CPU a GPU, pokročilou správu uzlových bodů a různé možnosti grafické vizualizace.

Algoritmus IDW

Samotná rovnice (1, 2), která definuje IDW je velmi jednoduchá. Jedná se o typický příklad zlomku se dvěma sumami, kde suma v čitateli funguje jako akumulátor výsledku a jmenovatel poté jako normalizátor do zadaného rozsahu.

Na vstupu algoritmus očekává sadu uzlových bodů společně s funkční hodnotou a dále jeden bod, pro který je třeba aproximovat hodnotu. Prvky nejsou nazývány body náhodou, neboť pro správnou funkčnost IDW je třeba, aby mezi body byla definovaná metrika vzdálenosti. Dokonce se dle toho i algoritmus jmenuje.

$$u(\boldsymbol{x}) = \frac{\sum w_i(\boldsymbol{x}) u_i}{\sum w_i(\boldsymbol{x})} \tag{1}$$

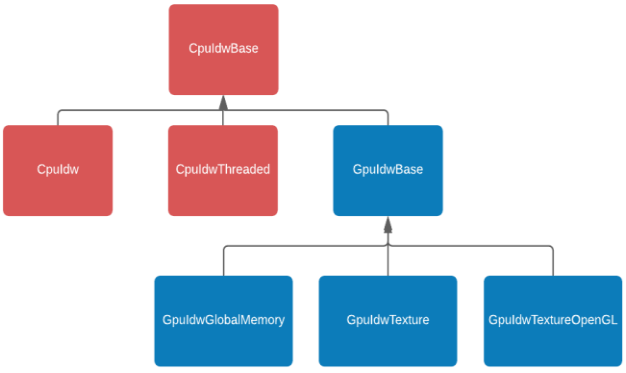
$$w_i(\boldsymbol{x}) = \frac{1}{dist(\boldsymbol{x}, x_i)^{\boldsymbol{p}}} \tag{2}$$

Parametr p zde funguje jako další způsob váhování. Čím vyšší je parametr p, tím větší se přikládá váha nejbližšímu okolím. Naopak s menší hodnotou p je výsledná hodnota hledaného bodu ovlivněná i uzlovými body, které jsou dál a obraz je v daném případě více pozvolný.

Obecný popis implementace

Aplikace je napsaná v jazyce C++ ve verzi 11 za použití knihoven CUDA, OpenGL a fmt. CUDA slouží pro výpočty na grafických kartách, OpenGL pro zobrazení obrazu na obrazovce a fmt je knihovna pro formátování řetězců a jejich vypsání na obrazovku.

Aplikace obsahuje dvě důležité třídy – DataManager a CpuIdwBase. První slouží jako centrální úložiště všech dat, které aplikace potřebuje ke svému běhu. Jedná se např. o různé nastavení nebo polohu uzlových bodů. Druhá třída s názvem CpuIdwBasem je rodičovská třída, pro všechny různé implementace algoritmu IDW. Mezi její hlavní funkce patří refresh, která způsobuje nové generování obrazu a dále drawOpengl, která zajišťuje vykreslení obrazu na obrazovku počítače.



V rámci celé aplikace je použit následující vzor, kdy data jsou nejprve generována jako výšková mapa do matice typu uint8_t. Ta zároveň může sloužit jako černobílá (případně jinak jednobarevná) varianta.

Pro tvorbu barevného obrazu je pak použito mapování již vypočtené výškové mapy do různých barevných prostorů. Každý prostor je pouhý slovník, který na základě hodnoty vrátí výslednou barvu.

Je to takto rozděleno z důvodu optimalizace. Zápis pouze do výškové mapy by měl být až 4x rychlejší, neboť je to pouze čtvrtina objemu dat. Pro praktickou práci se však černobílý model moc nehodí, neboť lidské oči mají problém dobře rozeznat jednotlivé drobné změny. Na to je právě dobré barevné mapování, kde jsou všechny změny mnohem výraznější a lépe lidským okem pozorovatelné. Pokud se ukáže, že barevné generování je dostatečně rychlé, je možné tento mezikrok s výpočtem do výškové mapy úplně odstranit.

Popis jednotlivých implementací IDW

CPU Implementace

Implementace na procesoru je velmi jednoduchá a nachází se ve třídě CpuIdw. Pro každý bod ze vstupní matice se vypočte jeho hodnota algoritmem IDW. Vypočtení hodnoty je pouze zápis IDW rovnice do kódu, jedná se tedy o for-cyklus, který projde všechny uzlové body a spočte potřebné mezivýsledky. Ty jsou na konci sečteny a uloženy jako hodnota daného pixelu. Tento proces se musí opakovat pro každý pixel, je tedy třeba přidat další dva for-cykly, kde jeden iteruje přes výšku obrazu a druhý přes šířku.

Pro případné vytvoření barevného obrazu jsou třeba zase dva for-cykly iterující přes stejné proměnné. Ty pouze přečtou výškovou hodnotu na dané pozici a dle zvolené barevné palety uloží barvu do barevného obrazu.

```
for (int h = 0; h < height; ++h) {
   for (int w = 0; w < width; ++w) {

      wiSum = 0;
      outputSum = 0;

      for anchorPoint in anchorPoints {
          dist = computeDistance(currentPoint, anchorPoint);

         wiSum += wi =  1 / pow(dist, p);
         outputSum += wi * point.value;
      }

      output[h,w]=outputSum / wiSum;
   }
}
```

Ukázka kódu 1 - Výpočet hodnoty IDW na CPU

CPU implementace ve vláknech

Jelikož vypočtení hodnoty jakéhokoliv bodu v obraze je nezávislé na jakémkoliv jiném bodě, lze řešení velmi dobře paralelizovat. Implementace se nachází ve třídě CpuIdwThreaded. Ta rozdělí obraz na několik menších částí, které poté zadá jako samostatný úkol do vlákna. Vlákno vypočte všechny pixely ve své části a poté se ukončí. Hlavní vlákno vždy spustí předem definovaný počet dílčích vláken, a když nějaké doběhne, tak spustí novou úlohu. Takto se po částech vytvoří výsledný obraz. Samotný výpočet hodnoty jednoho pixelu je úplně stejný, jako v případě sériové implementace.

Výpočet barevného obrazu funguje na stejném principu. Obraz je rozdělen do několika menších částí, které se rozdělí mezi vlákna. Ty, dle již vypočtené výškové hodnoty a barevné mapy, uloží barevnou hodnotu.

GPU implementace pomocí globální paměti

Prvotním nápadem, pro paralelizaci IDW na grafické kartě, bylo použít paralelní redukci. Hlavním důvodem k tomu bylo, že ve vzorečku IDW se nachází dvě sumy. Tato myšlenka však byla brzy nahrazena jinou, a to použitím stejného výpočetního modelu jako v případě vláken. Tedy rozdělení obrazu do částí a spočtení celé hodnoty jedním vláknem.

Kdyby však celá úloha byla postavena trochu jinak, tak by použití redukce dávalo smysl. Úloha by v tom případě musela být následující. Muselo by se počítat pouze pár zajímavých bodů, které by měly stovky uzlových bodů. Tato práce se však zaměřuje na jiný postup, kde je třeba znát hodnotu pro všechny body a uzlových bodů je spíše méně.

Implementace na GPU je tedy téměř totožná jako implementace pomocí vláken. Jediný rozdíl je v tom, že CPU vlákna vždy vypočtou více pixelů, zatímco GPU vlákno vždy vypočte pouze jeden pixel. Velikost bloku vláken byla stanovena na hodnotu 256 v klasickém uspořádání do čtverce, tedy 16x16.

V případě tvorby barevného obrazu je postup analogický jako výše.

GPU implementace za využití sdílené paměti

Sdílená paměť je rychlá část paměti, ke které mají přístup vždy vlákna z jednoho warpu. Její využití je výhodné, pokud se nějaká data mají číst více než jednou. V daném případě je lepší data načíst prvně do sdílené paměti a poté s nimi pracovat zde. Na konci výpočtu je třeba vypočtenou hodnotu ze sdílené paměti zpátky překopírovat do paměti globální.

Po zanalyzování řešeného problému vyšlo, že by využití sdílené paměti nemělo mít žádný význam na rychlosti řešení. Předpokladem k tomuto závěru bylo, že kernel ani jednou nečte data a pouze na konci uloží hodnotu.

GPU implementace pomocí texturovací paměti

Od CUDA verze CC2, tedy cca od roku 2010, existuje možnost zapisovat do textur v rámci kernelu. Takové textury se jmenují surfaces. Výhodou tohoto přístupu by měla být lepší paměťová propustnost. Textury jsou totiž určeny pro přístup do prvků, které jsou prostorově blízko sebe, tedy v rámci obrazu jsou to vždy okolní pixely. Jelikož jeden blok vláken vždy zpracovává pixely blízko sebe, je šance, že použitím textur se zvýší výsledná rychlost programu.

Samotná implementace černobílého kernelu se poté nijak výrazně neliší od kernelu fungujícím pomocí globální paměti. Jediný rozdíl je v použitém způsobu uložení vypočtené hodnoty. V případě globální paměti stačí použít operátor přiřazení, zatímco zde je potřeba použít speciální funkci.

Implementace pomocí textur jsou vlastně dvě implementace, které se liší v jednom podstatném detailu. Ta první, základní, používá stejný mechanismus přenosu obrazu na obrazovku, jako všechny minulé metody. Tedy vytvoření obrazu, jeho nakopírování do paměti RAM, a poté zpětné nakopírování do paměti VRAM a jeho zobrazení na monitoru.

Tento krok je však víc než zbytečný, a proto existuje možnost, jak data v paměti VRAM rovnou zobrazit na monitoru. Jmenuje se OpenGL interop. Tento způsob je trochu složitější na naprogramování a má jistou režii, neboť je třeba vždy texturu zamknout a poté odemknout, ale v celkovém výsledku se to vyplatí. Tohoto přístupu bylo využito pouze u barevné varianty, černobílá funguje bez interopu.

Ze stejného důvodu, pak vychází i jedna nevýhoda. Jelikož je celý textový stav do obrazu vkládán na straně CPU, tak tato metoda má výpis stavu pouze do konzole.

```
void mapColorInteropTexture() {

    //texture has to be mapped every time it's used by cuda
    cudaGraphicsMapResources(1, &colorGraphicsResource);
    cudaArray_t viewCudaArray;
    cudaGraphicsSubResourceGetMappedArray(&viewCudaArray,
colorGraphicsResource, 0, 0);
    cudaResourceDesc viewCudaArrayResourceDesc;
    viewCudaArrayResourceDesc.resType = cudaResourceTypeArray;
    viewCudaArrayResourceDesc.res.array.array = viewCudaArray;

    cudaCreateSurfaceObject(&colorSurfObject, &viewCudaArrayResourceDesc);
}

void unmapColorInteropTexture() {
    cudaDestroySurfaceObject(colorSurfObject);
    cudaGraphicsUnmapResources(1, &colorGraphicsResource);
}
```

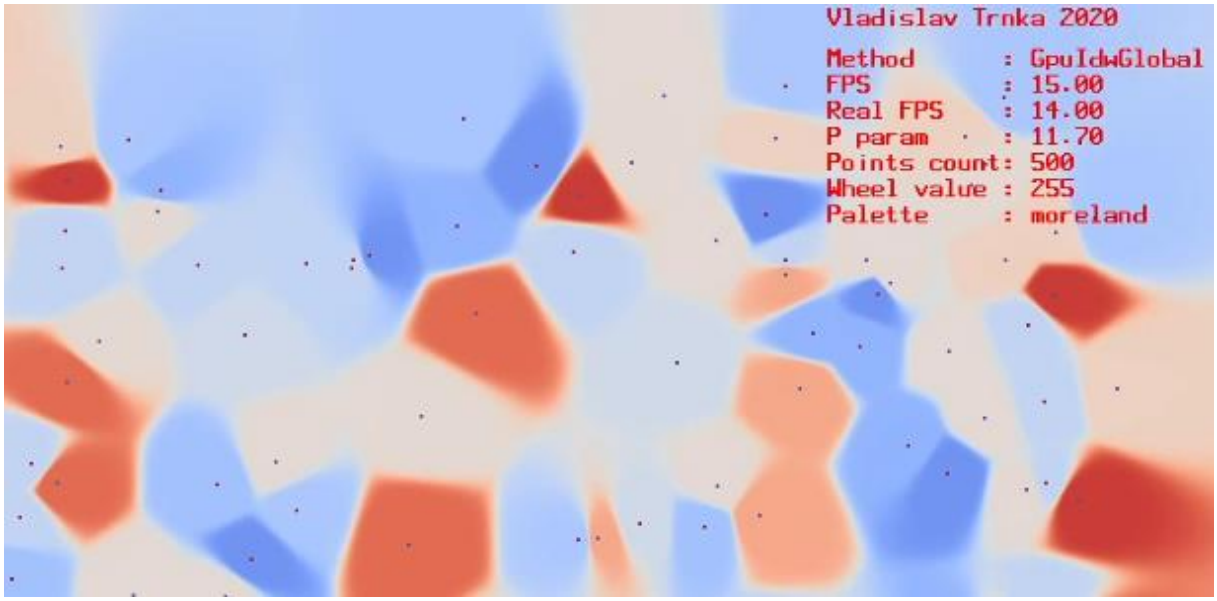
Ukázka kódu 2 - Způsob odemykání a zamykání textury pro použití v CUDA

Uživatelské rozhraní

Aplikace se skládá pouze z jedné obrazovky, která ukazuje vygenerovaný obraz a přes něj nakreslené jednoduché textové statistiky, jako např. právě použitou metodu výpočtu nebo počet FPS.

Aplikace je plně ovladatelná pomocí klávesnice a myši, kde myš slouží k přidávání a odebírání uzlových bodů na obrazovce. Klávesnice ovládá zbylé volby. Všechny možnosti jsou uživateli zobrazeny do příkazové řádky hned po spuštění programu. Zároveň je tuto nápovědu možné kdykoliv vyvolat stisknutím tlačítka h.

Důležité klávesy jsou tabulátor, šipky a numerický blok. Tabulátor postupně přepíná mezi jednotlivými metodami, šipky ovládají zvolenou barevnou paletu a hodnotu parametru p a numerický blok slouží k práci s uloženými datasety. Celkem je připraveno 9 datasetů, rozdělených po trojicích dle očekávané velikosti obrazu. Je možné i uložit a poté zpětně načíst uživatelské pozice.



Obrázek 1 - Ukázka uživatelského rozhraní

Testování

Hlavní částí testování je ověření, že jsou vypočtené výsledky správné a měření rychlosti renderování obrazu na různě velkých datových sadách.

Rychlost renderování byla měřena ve více scénářích vždy s jiným variabilním parametrem. Nejdůležitější parametr je počet uzlových bodů společně s velikostí generovaného obrazu. Mezi další zajímavé scénáře poté patří např. změna parametru p nebo otestování na jiném GPU. Spíše ze zvědavosti bylo změřeno maximální možné spočitatelné množství počtu uzlových bodů.

Primární počítač na testování obsahoval grafickou kartu NVIDIA 1080 Ti a procesor Intel i7-7700K. Sekundární počítač měl GPU o generaci novější, přesněji NVIDIA 2080 Ti a procesor o dvě generace novější Intel i7-9700K. Oba dva počítače měly 32GB RAM a operační systém Windows 10.

Celkem bylo otestováno devět datových sad, které jsou zároveň přístupné uživateli. Každá datová sada byla vždy testována na rozlišení, pro které byla vytvořena. Některé testy proběhly pouze v rámci GPU implementací, neboť datová sada již byla pro CPU moc náročná.

Ověření správnosti výsledků bylo provedeno pouze vizuálně, avšak v tomto případě je to dostačující. Při přepínání mezi jednotlivými IDW metodami by šlo jednoduše odhalit, kdyby nějaké metoda generovala jiný výsledek.

Výsledky testování

Testování FPS barevné varianty

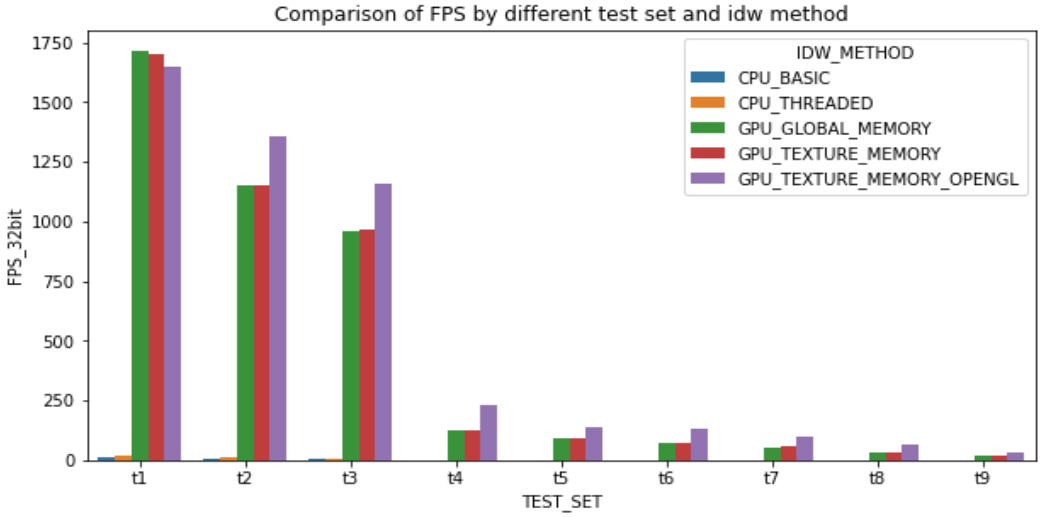
Nejdůležitější jsou výsledky rychlosti u barevné varianty. Z grafu č.1 lze vidět, že CPU metody jsou použitelné pouze pro rozlišení 256x256. Jednovláknový test dosahuje kolem 10 FPS v nejlehčím testu a 3 FPS v nejtěžším testu daného rozlišení. Vícevláknový výkon je poté zhruba dvakrát vyšší. Pro další testovací sady byl výkon rychle ověřen a výkon již nebyl dostatečný.

Výsledky na grafické kartě jsou výrazně lepší. V případě nejmenšího testovacího obrazu jsou hodnoty FPS v řádu stovek. V případě dalších testovacích sad je třeba použít přiblížený graf č.2, který lépe ukazuje rozdíly mezi vybranými metodami. Lze vidět, že interop varianta renderování přináší zhruba dvojnásobný výkon. Avšak všechny metody nabízí dostatečné FPS ve všech testech.

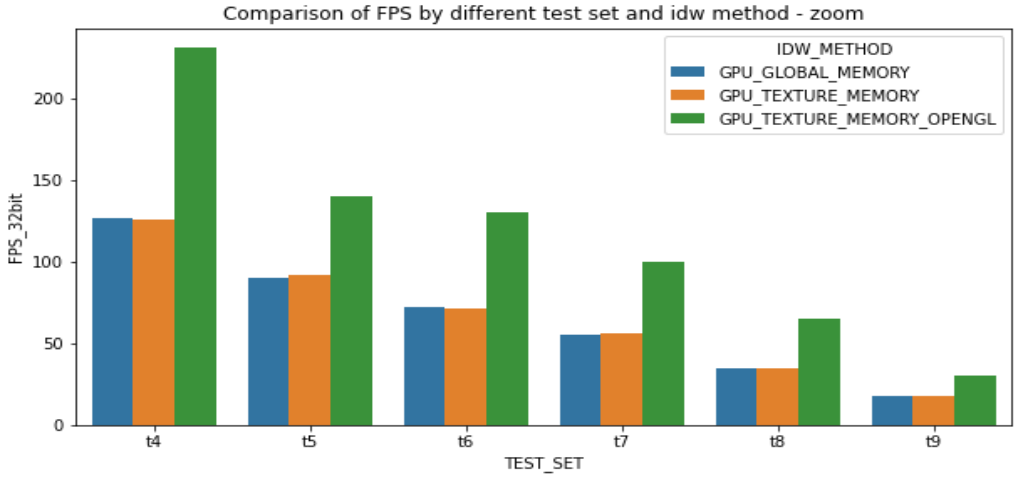
Z grafu lze také vyčíst, že metody pomocí globální a texturovací paměti jsou stejně rychlé. Z jednoho pohledu to dává smysl, neboť obě dvě dělají přesně to samé. Zároveň jsem však očekával, že texturovací paměť bude trochu rychlejší z důvodu již výše popsaného lepší přístupu do paměti.

Zajímavé je také srovnání vytížení GPU na grafu č.3. Zde lze vidět, že interop metoda má výrazně menší vytížení GPU než zbylé dvě.

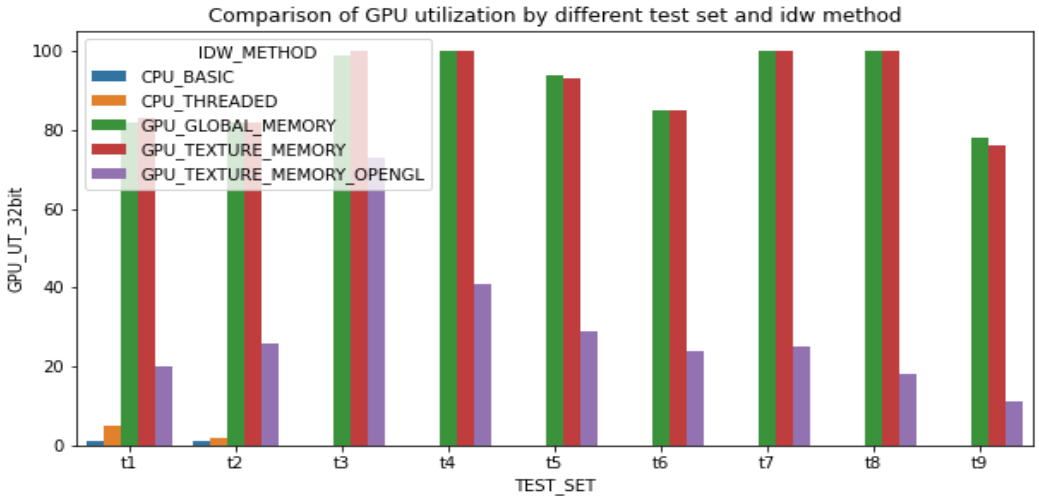
Dále bylo změřeno, že maximální počet bodů je více než dostatečný pro praktické použití. V případě 7500 bodů byla již celá obrazovka plně obsazená body a hodnota FPS byla stále asi 3.



Graf 1-Porovnání rychlosti v závislosti na velikosti datové sady



Graf 2-Porovnání rychlosti v závislosti na velikosti datové sady - přiblížení



Graf 3- GPU využití dle použité metody

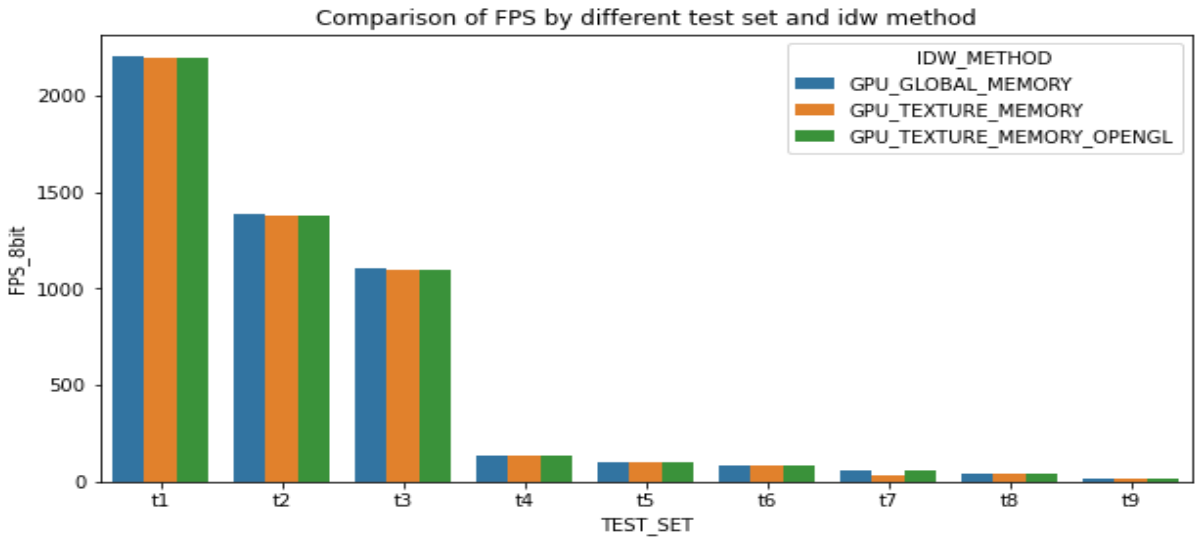
Černobílá varianta

Hlavní důvod pro rozdělení implementace na osmibitovou a barevnou variantu je předpoklad na zrychlení celého programu. Tento předpoklad se potvrdil, neboť osmibitová varianta je rychlejší. Lze to vidět po porovnání grafů 2 a 4. Hlavně u nejmenšího rozlišení jsou počty FPS až o 20% vyšší. S rostoucím rozlišením se však procentuální náskok zmenšuje až téměř zmizí.

Celkově však bylo testování černobíle varianty zvláštní, neboť byly objeveny dvě anomálie. První se týkala využití GPU. Když bylo přepnuto na texturovanou interop variantu v osmibitovém režimu, tak GPU měla nižší vytížení než v neinterop režimu při stejném počtu FPS. A to i přesto, že by měly být obě varianty v osmibitové verzi úplně stejné.

A druhá zvláštnost byla v renderování, kdy při použití osmibitové varianty jako černobílý obraz bylo dosaženo vyšších FPS než jako při použití jako jeden z barevných kanálů, tedy např. pouze zelený.

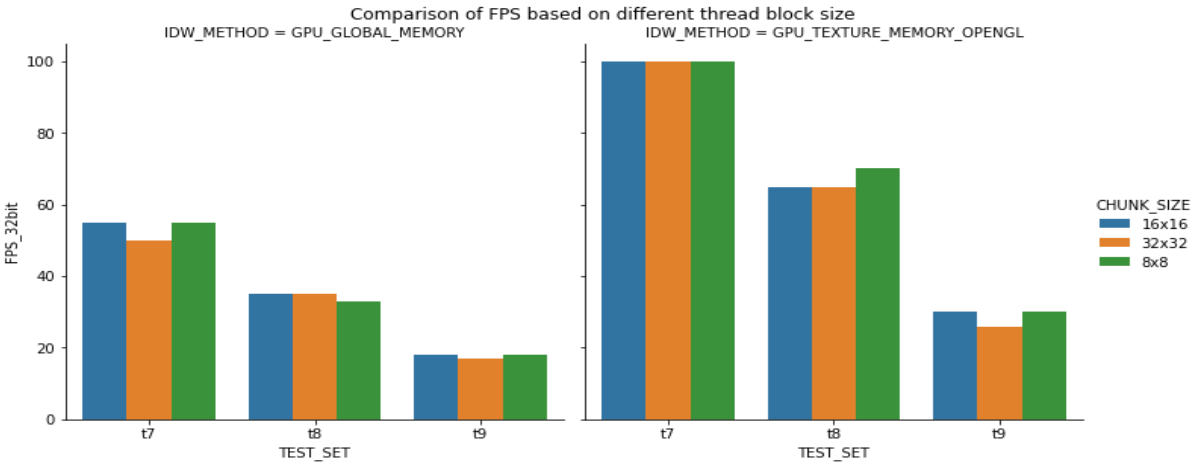
I přesto lze konstatovat, že nemá smysl používat tuto osmibitovou variantu. Barevná je ve všech případech stejně dobrá nebo výrazně lepší.



Graf 4 - Počet FPS při použití 8mibitové varianty

Testování velikosti bloku vláken

Ze srovnání velikostí bloku vláken lze usoudit, že daný parametr nemá na výkon téměř žádný vliv. Zároveň i vytížení GPU bylo podobné.

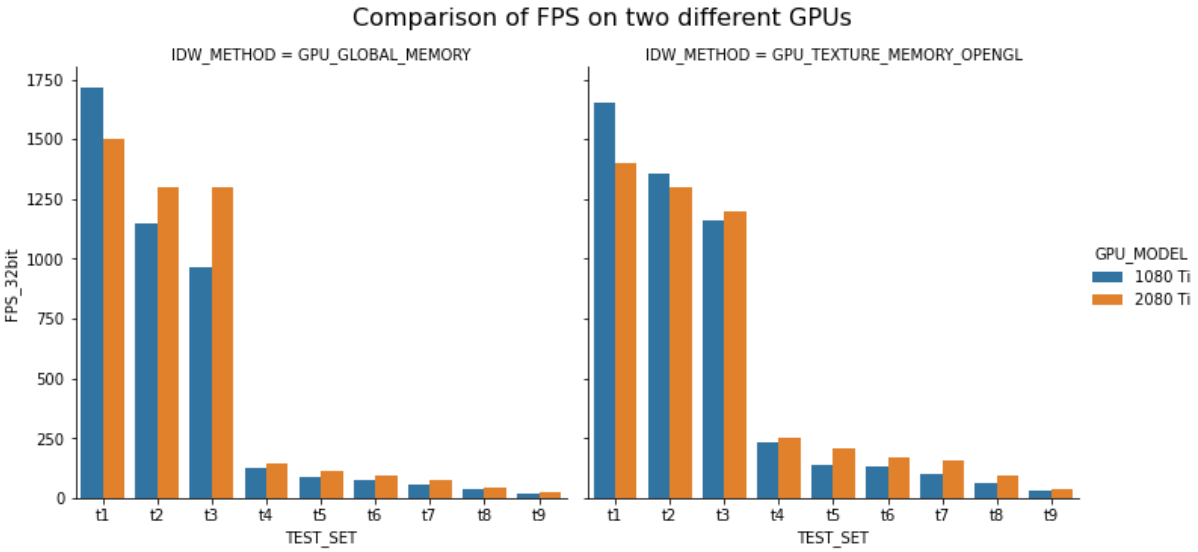


Graf 5 - Srovnání výkonu dle velikosti bloku vláken

Testování druhé grafické karty

Posledním provedeným testem bylo měření další grafické karty. Jedná se o NVIDIA 2080 Ti, která má 4352 CUDA jader na frekvenci 1350 MHz. Primární grafická karta je NVIDIA 1080 Ti s počtem 3584 CUDA jader na frekvenci 1480. Rychlým srovnáním těchto čísel lze usoudit, že by druhá karta mohla mít hodnoty FPS cca o 20% vyšší.

Předpoklad o lepším výkonu se naplnil, novější karta dosahuje vyšších FPS. Nárůst ve výkonu se pohyboval kolem již zmíněných 20%.



Graf 6 - Porovnání rychlosti na dvou různých GPU

Závěr

Práce na téma vytvoření programu pro vizualizaci IDW se zdařila. Byl naprogramován program, který je funkční, rychlý a uživatelsky příjemný na ovládání. Obsahuje několik přednastavených barevných palet, které jsou kritické pro lepší vizualizaci generovaných dat.

Nejlepší naprogramovaná varianta dosahuje dostatečných FPS pro pohodlnou práci při nízkém vytížení CPU a GPU. Bylo změřeno, že program bude stále fungovat, i když se použije veliké množství kotvících bodů.

Možné další vylepšení programu jsou různá. V rámci výkonnostní optimalizace by se vyplatilo kompletně odstranit osmibitový kanál a pracovat napřímo s barevnou maticí. Také by pro větší počet bodů stálo za zvážení vynechání vzdálených bodů, které stejně nemají žádný vliv na celkový výsledek. Omezit se např. pouze na kruh o nějakém poloměru. Další možnosti optimalizace by bylo renderování snímku v menším rozlišení a poté ho interpolovat na rozlišení větší.

Pro zlepšení použitelnosti programu by bylo vhodné implementovat nějaký vestavěný výkonnostní test, který najde nejrychlejší metodu na daném PC a tu poté použije. Stejně tak jako naimplementovat metodu ještě v OpenCl, aby aplikace nebyla závislá na GPU od NVIDIE.

Mě osobně by poté zajímala rychlost implementace ve vektorových jednotkách CPU a její srovnání s CPU vláknovou verzí. Dále pak implementace dalších metrik vzdálenosti než jen euklidovské. Ty by mohly tvořit vizuálně zajímavé obrazy.