


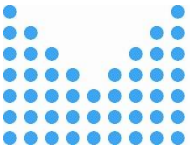

Příjemci podpory: ***Poskytovatel:***

Typ výsledku dle UV č. 837/2017	Evidenční číslo (příjemce)	Rok vzniku
R- software	CZ.01.1.02/0.0/0.0/21_374/0026902	2022
ISBN-ISSN	**Webový odkaz na výsledek**	**Kde a kdy publikováno**
N/A	https://www.fit.vut.cz/research/product/754/	Online

Anotace k výsledku:

Výstupem projektu je software umožňující HW akcelerované zpracování obrazových dat z dopravních kamer prostřednictvím GPU na vestavěných zařízeních od společnosti Intel. V rámci tohoto softwaru byla naimplementována abstraktní knihovna pro práci s GPU zdroji, knihovna implementující funkce této abstraktní knihovny nad knihovnou OpenCL a knihovna využívající abstraktní knihovnu k akcelerovaným konverzím mezi datovými typy obrazů, ke změnám velikosti obrazu s použitím interpolace, k převodům Bayerovy mřížky na plný RGB obraz a k převodům barevných prostorů na odstíny šedi. Knihovna zároveň umožňuje různé nastavení datových typů, paměťových prostorů, počtů elementů, interpolací a mnoha dalších nastavení, offsety v obrazech a mnoho dalších parametrů. Pro validitu implementace byly kombinace těchto parametrů otestovány oproti referenční procesorové implementaci.
V této zprávě se vyskytuje popis jednotlivých částí systému, příklad využití softwaru v praxi a vyhodnocení výkonnosti.

Příjemci podpory: ***Poskytovatel:***

1 Úvod

Zpracování obrazových dat z dopravních kamer patří vzhledem k jejich objemu k výpočetně náročným operacím. Algoritmy využívané pro detekce, klasifikace a další aplikace vyžadují vstupní obraz o určitých parametrech a jejich výstupem jsou obrazy a metadata pro další zpracování. Za účelem zpracování vstupních dat ze senzorů těmito algoritmy a pro možnost řetězení těchto algoritmů je pro jejich napojení nutné zajistit předzpracování obrazu ve formě převodů mezi vstupními a výstupními formáty obrazu těchto algoritmů, ořezávání těchto obrazů a další operace s nimi. Pro integraci těchto algoritmů do kamer je navíc vzhledem k omezeným možnostem chlazení a velikosti těchto kamer nutné použít vestavěné zařízení s nízkým příkonem, které má omezenou výpočetní kapacitu. Díky tomu je nutné zpracovávat celou výpočetní pipelinu, a to včetně předzpracování obrazu, s co největším důrazem na efektivitu využití zdrojů. Z těchto důvodů vznikl software umožňující HW akceleraci metod předzpracování obrazu prostřednictvím GPU, který je popsaný v tomto výstupu projektu.

2 Technická specifikace software

Popisovaný software je napsaný prostřednictvím jazyků C++ a OpenCL C a implementuje algoritmy pro předzpracování obrazu s využitím GPU platformy od společnosti Intel prostřednictvím knihovny OpenCL. Software je rozdělen na 3 části:

- knihovna GpuFramework implementuje rozhraní pro abstrakci objektů využívajících GPU (fronty, buffery, kernely, programy atd.)
- knihovna OpenCLGpuFramework implementuje metody knihovny GpuFramework prostřednictvím nástavby nad knihovnou OpenCL
- knihovna PPIFramework implementuje metody předzpracování obrazu (Bayerův filtr, rescale atd.) s využitím knihovny GpuFramework

2.1 Knihovna PPIFramework

Knihovna PPIFramework je reprezentována třídou `PpiFramework`. Tato třída poskytuje 4 metody pro předzpracování obrazu. Všechny níže popsané metody je možné omezit jen na část obrazu prostřednictvím kombinace offsetů ve vstupním a výstupním obrazu a velikosti zpracovávané části. Díky těmto parametrům je možné kromě kompletního zpracovávání obrazu pracovat i s výřezem vstupního obrazu a výsledek umístit na jakékoliv místo ve výstupním obraze. Obrazy zároveň můžou být různě velké. Vstupní a výstupní obrazy je možno umístit jak v texturovací, tak v lineární paměti.

2.1.1 BayersFilter

Metoda `BayersFilter` provádí převod obrazu z Bayerovy masky, který je RAW výstupem většiny barevných obrazových senzorů, do RGB nebo RGBA kanálového obrazu s plným rozlišením. Metoda podporuje pouze datový typ `unsigned char` s jakýmkoliv posunutím Bayerovy masky (`BGGR`, `RGBG`, `GRBG`, `RGGB`). Výpočet jednotlivých kanálů pixelů se provádí prostřednictvím interpolace okolních

pixelů daného kanálu. Tuto interpolaci je možné nastavit na nejbližšího souseda, nebo na lineární interpolaci sousedních pixelů.

Parametry metody `PpiFramework::BayersFilter`:

- `src` – 1 kanálový vstupní obraz vygenerovaný třídou `GpuFramework`
- `dst` – 3 nebo 4 kanálový výstupní obraz vygenerovaný třídou `GpuFramework`
- `src_start` – 2D pozice začátku čteného regionu vstupního obrazu
- `dst_start` – 2D pozice začátku zapisovaného regionu výstupního obrazu
- `roi_size` – 2D velikost zpracovávaného regionu
- `mask_type` – zarovnání Bayerovy masky (`BGGR`, `RGBG`, `GRBG`, `RGGB`)
- `interpolation_type` – typ interpolace pixelů (`NEAREST` – nejbližší soused, `LINEAR` – bilineární interpolace)
- `queue` – GPU fronta pro asynchronní zpracování
- `event` – výstupní event pro profilovací účely nebo synchronizaci GPU front

2.1.2 ConvertRGBGray

Metoda `ConvertRGBGray` slouží k převodu obrazu s 3 nebo 4 kanály do odstínů šedi. Převádět je možné pouze obrazy v rámci stejného datového typu (`unsigned char`, `unsigned short`, `unsigned int`, `float`). Vzhledem k tomu, že je možné nastavit různé váhy vstupních kanálů, lze metodu využít jak pro převod RGB, tak i jiných 3 nebo 4 kanálových obrazů na stupně šedi.

Parametry metody `PpiFramework::ConvertRGBGray`:

- `src` – 3 nebo 4 kanálový vstupní obraz vygenerovaný třídou `GpuFramework`
- `dst` – 1 kanálový výstupní obraz vygenerovaný třídou `GpuFramework`
- `src_start` – 2D pozice začátku čteného regionu vstupního obrazu
- `dst_start` – 2D pozice začátku zapisovaného regionu výstupního obrazu
- `roi` – 2D velikost zpracovávaného regionu
- `coeffs` – koeficienty vah vstupních kanálů
- `queue` – GPU fronta pro asynchronní zpracování
- `event` – výstupní event pro profilovací účely nebo synchronizaci GPU front

2.1.3 ImageResize

Metoda `ImageResize` umožňuje provést změnu velikosti obrazu nebo jeho částí. Metodu je možné nastavit na lineární interpolaci okolních pixelů, nebo na interpolace prostřednictvím nejbližšího souseda. Pro úsporu výpočetního výkonu GPU jednotek lze volitelně zapnout HW akceleraci interpolace pomocí texturovacích jednotek. Tato HW akcelerace je podporovaná pouze u vstupních bufferů umístěných v texturovací paměti pro nalezení nejbližšího souseda u libovolného datového typu, nebo pro lineární interpolaci u datového typu `float` (s výjimkou 3 kanálových obrazů s vypnutou emulací 3 kanálů prostřednictvím 4 kanálů).

Parametry metody `PpiFramework::ImageResize`:

- `src` – vstupní obraz vygenerovaný třídou `GpuFramework`
- `dst` – výstupní obraz vygenerovaný třídou `GpuFramework`
- `src_start` – 2D pozice začátku čteného regionu vstupního obrazu
- `dst_start` – 2D pozice začátku zapisovaného regionu výstupního obrazu
- `src_roi` – 2D velikost zpracovávaného regionu ve vstupním obraze
- `dst_roi` – 2D velikost přepisovaného regionu ve výstupním obraze
- `interpolation_type` – typ interpolace pixelů (`NEAREST` – nejbližší soused, `LINEAR` – bilineární interpolace)
- `queue` – GPU fronta pro asynchronní zpracování
- `event` – výstupní event pro profilovací účely nebo synchronizaci GPU front
- `interpolation_in_hw` – zapnutí akcelerace interpolace prostřednictvím texturovacích jednotek (aktivuje se v případě, že je pro vstupní obraz dostupná)
- `resize_in_local_mem` – zapnutí cachování zpracovávaného regionu skupiny vláken prostřednictvím lokální paměti (snižuje nároky na paměťovou propustnost u neakcelerované lineární interpolace)

2.1.4 ConvertFormat

Metoda `ConvertFormat` převádí obrazy se stejným počtem kanálů mezi různými datovými typy (`float`, `unsigned char`, `unsigned short`, `unsigned int`).

Parametry metody `PpiFramework::ConvertFormat`:

- `src` – vstupní obraz vygenerovaný třídou `GpuFramework`
- `dst` – výstupní obraz vygenerovaný třídou `GpuFramework`
- `src_start` – 2D pozice začátku čteného regionu vstupního obrazu
- `dst_start` – 2D pozice začátku zapisovaného regionu výstupního obrazu
- `roi` – 2D velikost zpracovávaného regionu
- `queue` – GPU fronta pro asynchronní zpracování
- `event` – výstupní event pro profilovací účely nebo synchronizaci GPU front

2.2 Knihovna GpuFramework

Knihovna GpuFramework reprezentovaná třídou `GpuFramework` slouží k tvorbě abstraktních tříd pro práci s GPU zdroji. Tyto zdroje jsou popsány v následujících podkapitolách. Knihovna byla vytvořena s důrazem na použití v platformách se sdílenou CPU/GPU operační pamětí (možnost mapování GPU Bufferů a plnění ze strany CPU, využití zero-copy memory atd.).

2.2.1 Buffery

Metoda `createBuffer` instance třídy `GpuFramework` vytváří 1D/2D GPU buffery (třídy `GpuBuffer`) daného datového typu a počtu komponentů na element. U Bufferu je možné nastavit, jestli bude buffer uložen v texturovací, nebo v lineární paměti. Buffer je možné kopírovat prostřednictvím jeho metod z jiného GPU bufferu, z CPU bufferu, plnit ho definovaným elementem, měnit jeho velikost

nebo ho mapovat pro zápis/čtení z hosta. U 3 komponentových bufferů lze volitelně nastavit emulaci prostřednictvím 4 komponentů.

Parametry metody `GpuFramework::createBuffer`:

- `data_type` – datový typ bufferu reprezentovaný výčtovým typem
- `data_type_size` – velikost datového typu
- `size` – 2D velikost bufferu
- `line_size` – skutečná šířka řádku v bajtech (používaná pro zarovnání)
- `components` – počet komponentů v pixelech obrazu
- `buffer_type` – umístění GPU bufferu (`LINEAR` – lineární paměť, `TEXTURE_2D` – texturovací paměť)

2.2.2 Fronty

Metoda `createQueue` instance třídy `GpuFramework` vytváří fronty (třídy `GpuQueue`) pro asynchronní běh GPU operací (běh kernelů, kopírování dat, mapování atd.).

Parametry metody `GpuFramework::createQueue`:

- `profiling_enabled` – zapnutí profilování GPU operací (běh kernelů, kopírování dat atd.)

2.2.3 Události

Metoda `createEvent` instance třídy `GpuFramework` vytváří události (třídy `GpuEvent`), které je možné použít pro měření doby běhu GPU operace (běh kernelů, kopírování dat atd.), synchronizace mezi frontami, nebo čekání na dokončení operací z fronty před událostí při zachování běhu dalších operací.

2.2.4 Programy

Metoda `createProgram` instance třídy `GpuFramework` vytváří a kompiluje GPU programy (třídy `GpuProgram`) z předem daného souboru. GPU programy obsahují sadu kernelů (GPU funkcí), které jsou volatelné z hosta a dále mohou zahrnovat sadu ostatních GPU funkcí, které jsou volány z těchto kernelů.

Parametry metody `GpuFramework::createProgram`:

- `filename` – cesta k souboru se zdrojovým kódem GPU programu
- `defines` – definice, se kterými se budou kernely kompilovat

2.2.5 Kernely

Metoda `getKernel` instance třídy `GpuProgram` vytváří kernely (třídy `GpuKernel`) ze zkompilovaného GPU programu. U kernelu je možné nastavit jeho argumenty, velikost dynamické lokální paměti a spustit kernel o definovaném počtu vláken.

Parametry metody `GpuProgram::getKernel`:

- `kernel_name` – jméno kernelu

2.2.6 Pomocná třída GpuProgramCompiler

Pomocná třída `GpuProgramCompiler` slouží k automatizaci správy, ke kompilaci programů a k získávání kernelů dle požadavků uživatele. Díky této třídě lze zpracovávat stejné kernely a programy více objekty bez nutnosti si tyto kernely předávat, nebo je znovu vytvářet. Třída používá veřejnou metodu `getKernel`, která dle cesty k programu a jeho definic nejprve zjistí, zda již byl program načten a zkompilován. Pokud program s danými definicemi nenajde v databázi, vytvoří ho. Následně kontroluje, jestli byl vytvořen požadovaný kernel, v opačném případě ho vytvoří a na závěr ho předá uživateli formou ukazatele.

Parametry konstruktoru `GpuProgramCompiler::GpuProgramCompiler`:

- `framework` – instance třídy GpuFramework

Parametry metody `getKernel`:

- `program_filename` – cesta k souboru se zdrojovým kódem GPU programu
- `defines` – definice, se kterými se budou kernely kompilovat
- `kernel_name` – jméno kernelu

2.3 OpenCLGpuFramework

Knihovna OpenCLGpuFramework je v současnosti jedinou vytvořenou implementací abstraktní třídy `GpuFramework`. Knihovna je kombinací wrapperu nad objekty knihovny OpenCL a nově vytvořenými objekty pro usnadnění práce s buffery a programy dle požadavků GpuFrameworku. Implementace frameworku OpenCLGpuFramework je od uživatele využívající GpuFramework kompletně odstíněna s výjimkou metody `getGpuClFramework` pro jeho tvorbu. Knihovna umožňuje využít akceleraci na GPU/CPU platformách, kde je OpenCL C podporováno.

Parametry metody `getGpuClFramework`:

- `device_from_stdin` – volba zařízení pro výpočet prostřednictvím standardního vstupu

3 Použití softwaru v praxi

Software je možné využít pro předzpracování obrazu pro detektory objektů, pro skládání detekovaných objektů do nového obrazu, pro převody mezi rozměry a datovými typy mezi různými algoritmy dle požadavku, pro přípravu obrazu pro uložení do videa/obrazového souboru a v mnoha dalších aplikacích.

Jako reálný příklad je možné uvést převod videa ze senzoru, který dodává snímky ve formátu Bayerovy masky v datovém typu `unsigned char` pro algoritmus detektoru objektů, který vyžaduje datový typ `float` s požadovanou šířkou obrazu při zachování poměru stran. Zjednodušený zdrojový kód použití knihovny pro tento případ je znázorněn ve zdrojovém kódu 1.

```
unsigned int camera_w, camera_h, camera_pitch, algo_w, algo_h, algo_pitch;
getCameraSizes(&camera_w, &camera_h, &camera_pitch, ...); // zjištění vstupní velikosti obrazu
getAlgoSizes(&algo_w, &algo_h, &algo_pitch, ...); // zjištění požadavků na velikost obrazu
std::shared_ptr<GpuFramework> framework = getGpuClFramework(); // konstruktor OpenCL frameworku
std::shared_ptr<GpuQueue> queue = framework->createQueue(true); // vytvoření fronty
GpuVec2<unsigned int> zero_vec(0, 0); // startovací pozice zpracování bufferů
GpuVec2<unsigned int> camera_vec(camera_w, camera_h); // rozměry vstupu z kamery
GpuVec2<unsigned int> resized_vec(algo_w, camera_h * algo_h/ camera_w); // požadovaný výstup
GpuType char_t = templateToGpuType<unsigned char>();
GpuType float_t = templateToGpuType<float>();

// buffer kamery
std::shared_ptr<GpuImage> camera_frame = framework->createBuffer(char_t, sizeof(unsigned char),
        camera_vec, camera_pitch, 1, GpuBufferType::LINEAR);
// buffer z převodu na rgba
std::shared_ptr<GpuImage> rgba_frame = framework->createBuffer(char_t, sizeof(unsigned char),
        camera_vec, camera_vec.x * 4, 4, GpuBufferType::LINEAR);
// buffer z převodu na grayscale (texturovací typ, aby bylo možné akcelerovat
std::shared_ptr<GpuImage> gray_frame = framework->createBuffer(char_t, sizeof(unsigned char),
        camera_vec, camera_vec.x, 1, GpuBufferType::TEXTURE_2D);
// buffer z převodu do float typu
std::shared_ptr<GpuImage> float_frame = framework->createBuffer(float_t, sizeof(float),
        camera_vec, camera_vec.x * sizeof(float), 1, GpuBufferType::LINEAR);
// výstupní buffer
std::shared_ptr<GpuImage> output_frame = framework->createBuffer(float_t, sizeof(float),
        algo_vec, algo_pitch, 1, GpuBufferType::LINEAR);

// create framework
PpiFramework ppi_framework(*framework);
// mapování pro zápis prvního snímku
camera_vec->map(*queue, false, true);
// cyklus zpracování obrazu ze senzoru
while(getFrame(camera_vec->ptr, ...))
{
        // odmapování bufferu pro použití na GPU
        camera_vec->unmap(*queue);
        // převod Bayerovy masky na RGBA
        ppi_framework.BayersFilter(*camera_frame, *rgba_frame, zero_vec, zero_vec,
                camera_vec, BayerMaskTypes::BGGR, InterpolationTypes::LINEAR, *queue, nullptr);
        // převod RGBA na odstíny šedi
        ppi_framework.ConvertRGBGray(*rgba_frame, *gray_frame, zero_vec, zero_vec,
                camera_vec, GpuVec4<float>(0.2125, 0.7154, 0.0721, 0.0), *queue, nullptr);
        // převod unsigned char na float
        ppi_framework.ConvertFormat(*gray_frame, *float_frame, zero_vec, zero_vec, camera_vec,
                *queue, nullptr);
        // změna velikosti obrazu
        ppi_framework.ImageResize(*float_frame, *output_frame, zero_vec, zero_vec, camera_vec,
                resized_vec, InterpolationTypes::LINEAR, *queue, nullptr);
        // mapování obrazu pro zpracování algoritmem
        output_frame->map(*queue, true, false);
        // synchronizace mapování pro externí použití
        queue->sync();
        float *output_ptr = (float *)(output_frame->ptr);
        // zpracování snímku algoritmem
        processFrame(output_ptr, ...);output_frame->unmap(*queue);
        // mapování pro zápis dalšího framu
        camera_vec->map(*queue, false, true);
        // synchronizace
        queue->sync();
}
camera_vec->unmap(*queue);
```

Zdrojový kód 1: Ukázka zjednodušeného použití softwaru pro předzpracování obrazu ze senzoru.

4 Vyhodnocení výkonnosti

Výkonnost jednotlivých částí softwaru byla otestována na následující vestavěné platformě:

Vestavěné zařízení	Intel E3950 (4 jádra/4 vlákna)
GPU	HD Graphics 505 (18 multiprocesorů 650MHz)
Paměť RAM	8GB DDR4 (64bit 38.4GB/s)
Typická spotřeba	12W
Operační systém	openSUSE Leap 15.2
Kernel	5.3.18
GCC	8.2.1

V následujících podsekcích je vyhodnocena rychlost implementovaných metod předzpracování obrazu. Všechny implementace byly testovány na 4K obraze (8MPix). Výstupem jsou tabulky ukazující rychlosti kombinace datových typů, interpolací a umístění bufferů. Z výstupů testů lze vyvodit vhodnou strategii posloupnosti operací, umístění paměťových bufferů mezi operacemi a nastavení emulace 3 kanálového obrazu za pomocí 4 kanálového. Notace datových typů ve výsledcích odpovídá jazyku OpenCL (např. `uchar4` -> 4 komponentový `unsigned char`).

4.1 Převod Bayerovy mřížky na multikanálový obraz

Propustnost Bayerovy konverze do RGB byla sice měřena u všech možných zarovnání filtru (GRBG, GBRG, BGGE, RGGB), nicméně propustnost byla vzhledem ke stejnému množství operací u všech typů stejná. Z tohoto důvodu není zarovnání filtrů u výsledků přítomno. Propustnost jednotlivých variant je ukázána v Tabulce 1.

Debayerova filtrace (8MPix)	Buffer->Buffer	Buffer->Textura	Textura->Buffer	Textura->Texura
Lineární filtrace uchar3	**1598 Mpix/s**	384 Mpix/s	1596 Mpix/s	593 Mpix/s
Lineární filtrace uchar4	1351 Mpix/s	864 Mpix/s	1467 Mpix/s	942 Mpix/s
Nejbližší soused uchar3	**1887 Mpix/s**	389 Mpix/s	1709 Mpix/s	646 Mpix/s
Nejbližší soused uchar4	1579 Mpix/s	859 Mpix/s	1581 Mpix/s	1073 Mpix/s

Tabulka 1: Rychlost převodu z Bayerovy masky do 3 nebo 4 kanálového obrazu pro 8MPix obraz.

Z výsledku je zřejmé, že je vhodné se vyhnout zápisu do texturovací paměti, a to speciálně v případě 3 komponentové varianty. Důvodem je nepodpora 24bitových textur na straně HW, a tím nutnost emulace této podpory prostřednictvím 1 komponentové textury.

4.2 Převod multikanálového obrazu do odstínů šedi

Výsledky rychlosti konverze z RGB formátu do odstínů šedi jsou znázorněny v tabulce 2.

RGB->Gray (8MPix)	Buffer->Buffer	Buffer->Textura	Textura->Buffer	Textura->Texura
uchar3	1264 Mpix/s	1080 Mpix/s	1542 Mpix/s	1466 Mpix/s
uchar4	**1961 Mpix/s**	1610 Mpix/s	1932 Mpix/s	1456 Mpix/s
ushort3	848 Mpix/s	805 Mpix/s	1235 Mpix/s	1226 Mpix/s
ushort4	1026 Mpix/s	1166 Mpix/s	**1329 Mpix/s**	1240 Mpix/s
uint3	694 Mpix/s	541 Mpix/s	**1076 Mpix/s**	808 Mpix/s
uint4	630 Mpix/s	608 Mpix/s	847 Mpix/s	698 Mpix/s
float3	694 Mpix/s	541 Mpix/s	**1079 Mpix/s**	792 Mpix/s
float4	628 Mpix/s	606 Mpix/s	857 Mpix/s	700 Mpix/s

Tabulka 2: Rychlost převodu z 3 a 4 kanálového obrazu do odstínu šedi pro 8MPix obraz.

Z výsledků je patrná vhodnost využití texturovací paměti u vstupního bufferu, a to speciálně u větších datových typů, a schopnost rychle načítat nezarovnané datové typy (`uchar3`, `ushort3`, `uint3`, `float3`) prostřednictvím texturovací paměti.

4.3 Převody datových typů

Výsledky rychlosti převodů datových typů jsou popsány v tabulce 3.

	Buffer->Buffer	Buffer->Textura	Textura->Buffer	Textura->Texura
uchar1->ushort1	219 MPix/s	1971 MPix/s	**2247 MPix/s**	1611 MPix/s
uchar1->uint1	2482 MPix/s	1308 MPix/s	**2512 MPix/s**	1422 MPix/s
uchar1->float1	2512 MPix/s	1388 MPix/s	**2541 MPix/s**	1360 MPix/s
float1->uchar1	1831 MPix/s	1645 MPix/s	**1953 MPix/s**	1532 MPix/s
uint1->ushort1	1635 MPix/s	1686 MPix/s	**1829 MPix/s**	1560 MPix/s
uint1->uchar1	1853 MPix/s	1641 MPix/s	**1961 MPix/s**	1545 MPix/s
uchar2->ushort2	**2257 MPix/s**	1306 MPix/s	2252 MPix/s	1387 MPix/s
uchar2->uint2	**1280 MPix/s**	812 MPix/s	1257 MPix/s	909 MPix/s
uchar2->float2	**1286 MPix/s**	814 MPix/s	1258 MPix/s	898 MPix/s
float2->uchar2	1040 MPix/s	1206 MPix/s	**1331 MPix/s**	1246 MPix/s
uint2->ushort2	1175 MPix/s	989 MPix/s	**1441 MPix/s**	1112 MPix/s
uint2->uchar2	1037 MPix/s	1216 MPix/s	**1305 MPix/s**	1237 MPix/s
uchar3->ushort3	624 MPix/s	592 MPix/s	**774 MPix/s**	671 MPix/s
uchar3->uint3	454 MPix/s	349 MPix/s	**531 MPix/s**	460 MPix/s
uchar3->float3	455 MPix/s	328 MPix/s	**530 MPix/s**	483 MPix/s
float3->uchar3	514 MPix/s	482 MPix/s	**710 MPix/s**	697 MPix/s
uint3->ushort3	425 MPix/s	395 MPix/s	**557 MPix/s**	541 MPix/s
uint3->uchar3	517 MPix/s	461 MPix/s	**708 MPix/s**	706 MPix/s
uchar4->ushort4	**1217 MPix/s**	753 MPix/s	1211 MPix/s	827 MPix/s
uchar4->uint4	**643 MPix/s**	458 MPix/s	**643 MPix/s**	344 MPix/s
uchar4->float4	639 MPix/s	439 MPix/s	**645 MPix/s**	202 MPix/s
float4->uchar4	634 MPix/s	628 MPix/s	**846 MPix/s**	750 MPix/s
uint4->ushort4	554 MPix/s	505 MPix/s	**709 MPix/s**	526 MPix/s
uint4->uchar4	634 MPix/s	632 MPix/s	**845 MPix/s**	738 MPix/s
uchar4->uchar3	**1170 MPix/s**	755 MPix/s	1005 MPix/s	858 MPix/s
uchar4->ushort3	**826 MPix/s**	609 MPix/s	797 MPix/s	629 MPix/s
uchar4->uint3	**544 MPix/s**	366 MPix/s	525 MPix/s	503 MPix/s
uchar4->float3	**543 MPix/s**	362 MPix/s	518 MPix/s	508 MPix/s
float4->uchar3	**855 MPix/s**	690 MPix/s	588 MPix/s	640 MPix/s
uint4->ushort3	**663 MPix/s**	548 MPix/s	462 MPix/s	528 MPix/s
uint4->uint3	**460 MPix/s**	353 MPix/s	377 MPix/s	399 MPix/s
uint4->uchar3	**865 MPix/s**	662 MPix/s	594 MPix/s	644 MPix/s
uchar3->uchar4	1335 MPix/s	909 MPix/s	**1553 MPix/s**	1292 MPix/s
uchar3->ushort4	1269 MPix/s	678 MPix/s	**1441 MPix/s**	849 MPix/s
uchar3->uint4	985 MPix/s	418 MPix/s	**1051 MPix/s**	346 MPix/s
uchar3->float4	983 MPix/s	423 MPix/s	**1040 MPix/s**	345 MPix/s
float3->uchar4	719 MPix/s	534 MPix/s	**1137 MPix/s**	894 MPix/s
uint3->ushort4	656 MPix/s	437 MPix/s	**975 MPix/s**	582 MPix/s
uint3->uint4	617 MPix/s	320 MPix/s	**838 MPix/s**	297 MPix/s
uint3->uchar4	723 MPix/s	538 MPix/s	**1146 MPix/s**	881 MPix/s

Tabulka 3: Rychlost převodu mezi různými datovými typy a paměťmi pro 8MPix obraz.

Z výsledků je patrná vyšší propustnost algoritmu u vstupních obrazu umístěných v texturovací paměti s výjimkou, kde je obraz zapisován do 3 komponentového výstupu. S rostoucím počtem komponent

propustnost sice klesá, ale méně než lineárně vzhledem k počtu komponent. Z tohoto důvodu je preferované spíše používat buffery o více komponentech oproti převodům jednotlivých komponentů v samostatných bufferech.

4.4 Změna měřítka obrazu

Propustnost změny měřítka obrazu je popsána ve variantě s interpolací prostřednictvím nejbližšího souseda v tabulce 4 a ve variantě s použitím lineární interpolace v tabulce 5.

Změna měřítka - Nejbližší soused	Buffer->Buffer	Buffer->Textura	Textura->Buffer	Textura->Texura
uchar1->uchar1	2679 MPix/s	2127 MPix/s	**2943 MPix/s**	2031 MPix/s
ushort1->ushort1	2441 MPix/s	2220 MPix/s	**2745 MPix/s**	1969 MPix/s
uint1->uint1	2378 MPix/s	1388 MPix/s	**2543 MPix/s**	1590 MPix/s
float1->float1	2378 MPix/s	1388 MPix/s	**2533 MPix/s**	1502 MPix/s
uchar2->uchar2	2445 MPix/s	2176 MPix/s	**2603 MPix/s**	1960 MPix/s
ushort2->ushort2	2405 MPix/s	1472 MPix/s	**2477 MPix/s**	1621 MPix/s
uint2->uint2	**1282 MPix/s**	823 MPix/s	1174 MPix/s	967 MPix/s
float2->float2	**1282 MPix/s**	812 MPix/s	1175 MPix/s	985 MPix/s
uchar3->uchar3	1164 MPix/s	873 MPix/s	**1340 MPix/s**	924 MPix/s
ushort3->ushort3	773 MPix/s	636 MPix/s	**821 MPix/s**	601 MPix/s
uint3->uint3	524 MPix/s	340 MPix/s	**569 MPix/s**	508 MPix/s
float3->float3	523 MPix/s	355 MPix/s	**571 MPix/s**	463 MPix/s
uchar4->uchar4	**2356 MPix/s**	1476 MPix/s	2265 MPix/s	1530 MPix/s
ushort4->ushort4	**1282 MPix/s**	820 MPix/s	1166 MPix/s	997 MPix/s
uint4->uint4	**646 MPix/s**	453 MPix/s	597 MPix/s	365 MPix/s
float4->float4	**644 MPix/s**	445 MPix/s	645 MPix/s	311 MPix/s
uchar4->uchar3	1251 MPix/s	772 MPix/s	**1351 MPix/s**	963 MPix/s
ushort4->ushort3	806 MPix/s	594 MPix/s	**854 MPix/s**	750 MPix/s
uint4->uint3	509 MPix/s	350 MPix/s	**565 MPix/s**	563 MPix/s
float4->float3	507 MPix/s	318 MPix/s	**566 MPix/s**	563 MPix/s
uchar3->uchar4	**2084 MPix/s**	1419 MPix/s	1983 MPix/s	1573 MPix/s
ushort3->ushort4	1268 MPix/s	788 MPix/s	**1269 MPix/s**	595 MPix/s
uint3->uint4	**646 MPix/s**	462 MPix/s	644 MPix/s	369 MPix/s
float3->float4	**646 MPix/s**	463 MPix/s	644 MPix/s	359 MPix/s

Tabulka 4: Rychlost změny měřítka obrazu prostřednictvím nejbližšího souseda pro 8MPix obraz.

Změna měřítka Lineární	Buffer->Buffer	Buffer->Textura	Textura->Buffer	Textura->Texura
uchar1->uchar1	1589 MPix/s	1600 MPix/s	**1668 MPix/s**	1658 MPix/s
ushort1->ushort1	1573 MPix/s	1597 MPix/s	1629 MPix/s	**1633 MPix/s**
uint1->uint1	1353 MPix/s	1207 MPix/s	**1630 MPix/s**	1390 MPix/s
float1->float1	1671 MPix/s	1206 MPix/s	**1711 MPix/s**	1450 MPix/s
uchar2->uchar2	1317 MPix/s	1313 MPix/s	**1365 MPix/s**	1200 MPix/s
ushort2->ushort2	1352 MPix/s	1201 MPix/s	**1397 MPix/s**	1105 MPix/s
uint2->uint2	1023 MPix/s	681 MPix/s	**1274 MPix/s**	936 MPix/s
float2->float2	1037 MPix/s	741 MPix/s	**1276 MPix/s**	963 MPix/s
uchar3->uchar3	**841 MPix/s**	641 MPix/s	637 MPix/s	623 MPix/s
ushort3->ushort3	592 MPix/s	511 MPix/s	**637 MPix/s**	541 MPix/s
uint3->uint3	410 MPix/s	289 MPix/s	**566 MPix/s**	418 MPix/s
float3->float3	419 MPix/s	294 MPix/s	**564 MPix/s**	416 MPix/s
uchar4->uchar4	965 MPix/s	926 MPix/s	**999 MPix/s**	844 MPix/s
ushort4->ushort4	**968 MPix/s**	655 MPix/s	967 MPix/s	849 MPix/s
uint4->uint4	548 MPix/s	344 MPix/s	**643 MPix/s**	354 MPix/s
float4->float4	526 MPix/s	361 MPix/s	**645 MPix/s**	350 MPix/s
uchar4->uchar3	1003 MPix/s	815 MPix/s	**1093 MPix/s**	763 MPix/s
ushort4->ushort3	657 MPix/s	576 MPix/s	**829 MPix/s**	654 MPix/s
uint4->uint3	409 MPix/s	287 MPix/s	**561 MPix/s**	436 MPix/s
float4->float3	384 MPix/s	271 MPix/s	**562 MPix/s**	487 MPix/s
uchar3->uchar4	**1004 MPix/s**	899 MPix/s	642 MPix/s	587 MPix/s
ushort3->ushort4	**930 MPix/s**	611 MPix/s	635 MPix/s	626 MPix/s
uint3->uint4	593 MPix/s	387 MPix/s	**627 MPix/s**	320 MPix/s
float3->float4	603 MPix/s	352 MPix/s	**639 MPix/s**	318 MPix/s

Tabulka 5: Rychlost změny měřítka obrazu prostřednictvím lineární interpolace pro 8MPix obraz.

Algoritmus změny měřítka obrazu dosahuje ve většině případů vyšších rychlostí, pokud je vstupní obraz umístěn v texturovací paměti. Podobně jako v převodu datových typů platí, že je efektivnější zpracovávat multikanálovou texturu oproti samostatným kanálům. Výjimkou je případ s načítáním z velkých 3 komponentových bufferů u varianty interpolace prostřednictvím nejbližšího souseda.