Efektivita na GPU

Petr Holub
hopet@ics.muni.cz

PV197
2012–11–19

Přehled přednášky

Literatura

- Park I. K., Singhal N., Lee M. H., Cho S., Kim C. W., "Design and Performance of Evaluation of Image Processing Algorithms on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, 2010 (zatím pouze v elektronické verzi)
- Cope B., Cheung P. Y. K., Luk W., Howes L., "Performance Comparison of Graphics Processors to Reconfigurable Logic: A Case Study", *IEEE Transactions on Copmuters*, vol. 59, no. 4, April 2010
- Best Practices Guide – CUDA 2.2, 2009 **`http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf`**
- Wil Braithwaite, "The CUDA architecture: The Art of performance optimization", Siggraph 2009, **`http://developer.download.nvidia.com/presentations/2009/SIGGRAPH/asia/6_cuda_optimization.pdf`**

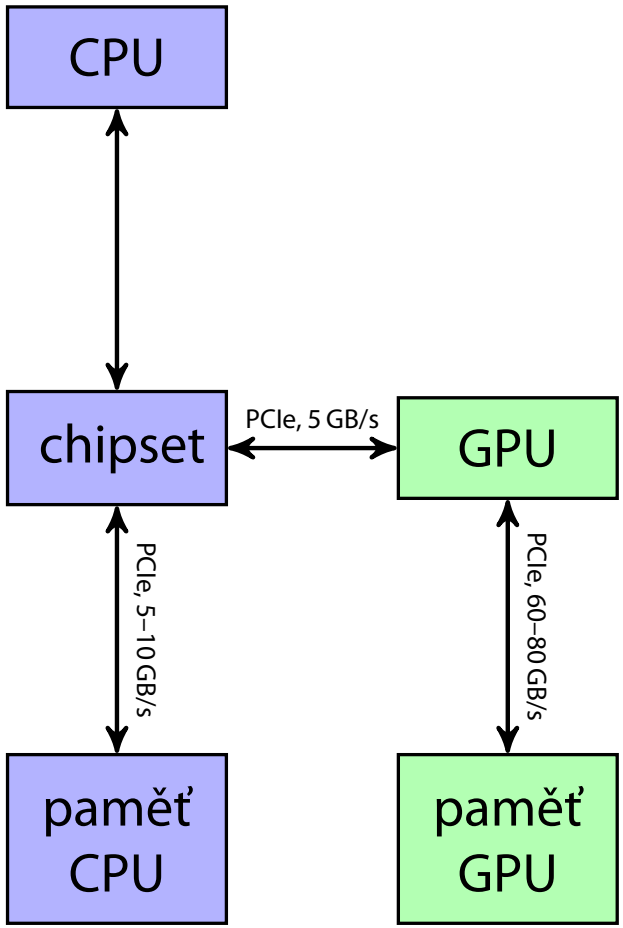
Shrnutí práce s pamětí

- Práce s pamětí
  - omezení pamětí
    - omezená šířka pásma mezi host a device (cca 6 GB/s pro PCI-e x16 Gen2)
    - latence globální paměti (teoreticky 141 GB/s, cca 400–600 cyklů latence)

$$\underbrace{1107 \times 10^6}_{\text{takt paměti [Hz]}} \times \underbrace{(512/8)}_{\text{Inteface paměti}} \times \underbrace{2}_{\text{DDR}} / 10^9 = 141{,}6\,\text{GB/s}$$

(ev. 132 GB/s při dělení 1024^3)

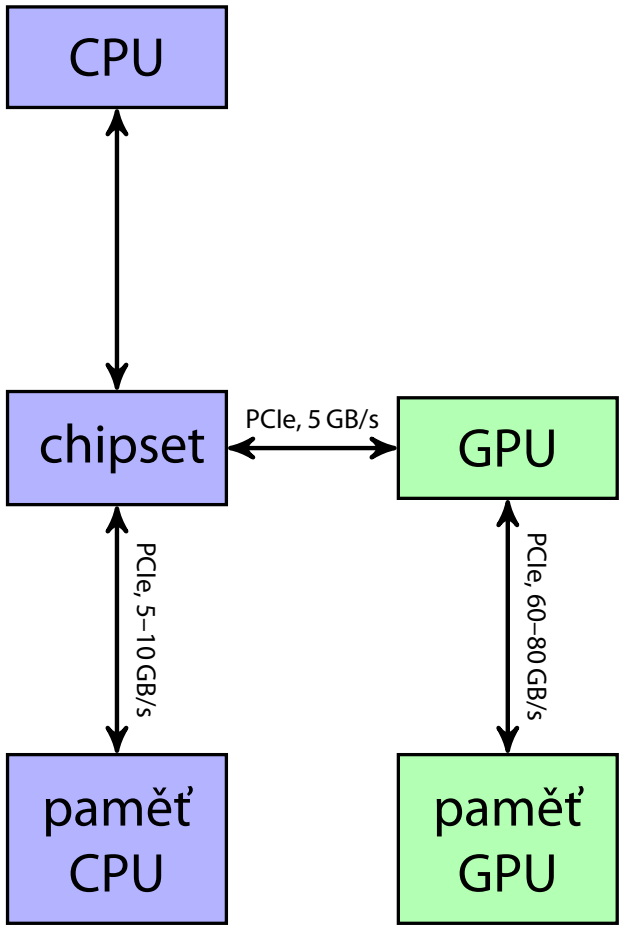
    - omezená velikost sdílené paměti
    - omezený počet registrů



Zdroj: Wil Braithwaite, "The CUDA architecture: The Art of performance optimization"

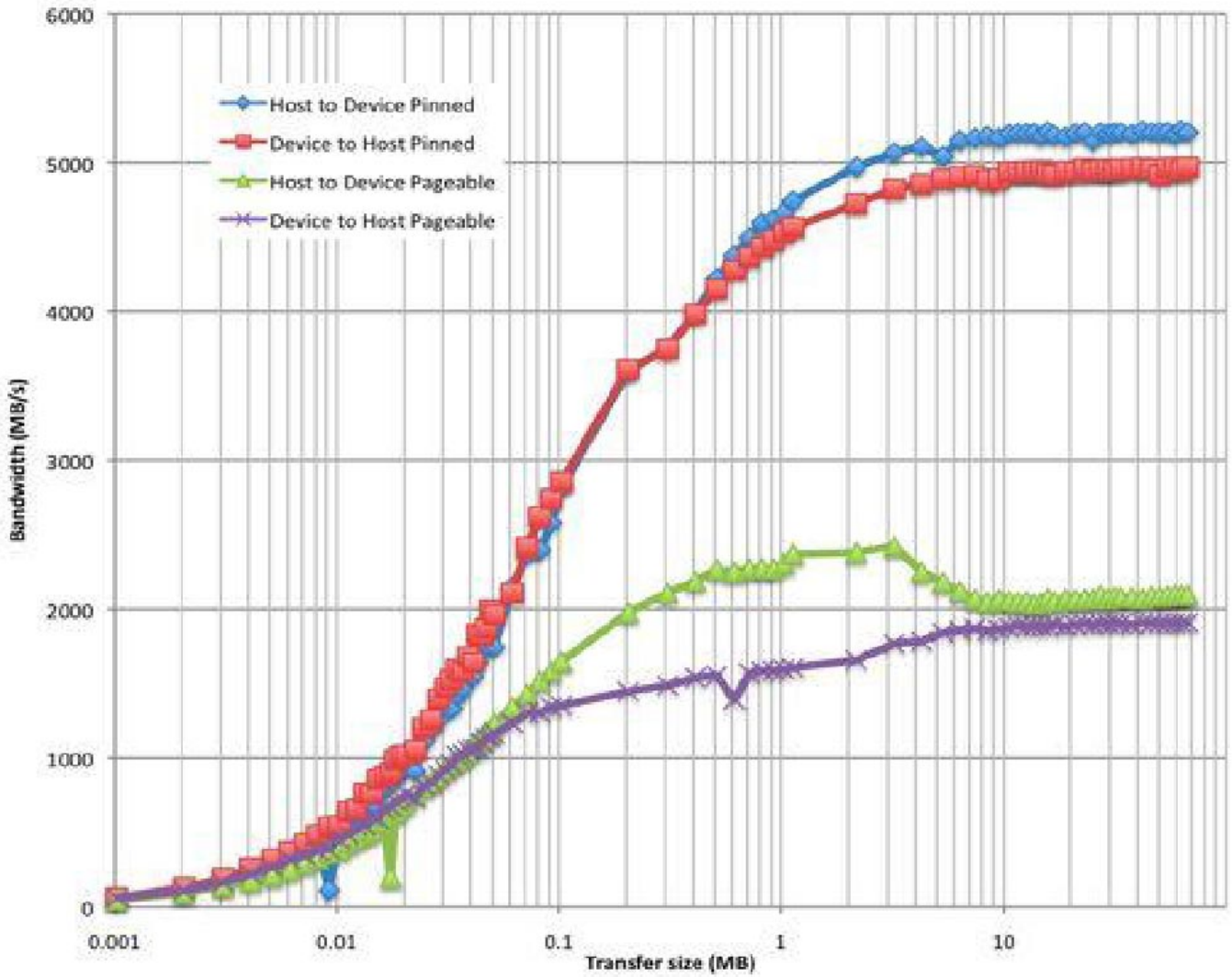
Shrnutí práce s pamětí

- Práce s pamětí
  - optimalizace
    - maximální využití sdílené paměti a registrů
    - koalescentní přístup do globální paměti a vyhnutí se partition campingu
    - překrývání výpočtů a přístupu k datům
    - využití asynchronních přenosů: `cudaMemcpyAsync(...);`
    - přiměřené používání page-locked paměti: `cudaHostAlloc(...);`
    - co největší přenosy host ↔ device najednou



Zdroj: Wil Braithwaite, "The CUDA architecture: The Art of performance optimization"

Shrnutí práce s pamětí



Zdroj: Wil Braithwaite, "The CUDA architecture: The Art of performance optimization"

Pokročilá práce s pamětí

- Další triky s pamětí
  - rozvrhnout, co cacheovat a co opakovaně počítat
  - mapování OpenGL bufferu do adresního prostoru zařízení (device)
    1. zaregistrujte si buffer pomocí CUDA-C
       `cudaGLRegisterBufferObject(GLuint buffObj);`
    2. namapujte zaregistrovaný buffer do globální paměti zařízení (vrátí adresu)
       `cudaGLMapBufferObject(void** devPtr,GLuint buffObj);`
    3. použijte adresu v kernelu
    4. odmapujte buffer
       `cudaGLUnmapBufferObject(GLuint buffObj);`
    5. odregistrujte buffer
       `cudaGLUnregisterBufferObject(GLuint buffObj);` (potřeba pouze pokud je buffer cíl rendrování)
    6. použijte buffer v OpenGL

    - může pomoci odstranit přenosy host ↔ device
    - automatické DMA mezi kartami Tesla a Quadro (momentálně přes host)
    - vykreslování z pixel buffer object pomocí `glDrawPixels` nebo `glTexImage2D`

Zdroj: Wil Braithwaite, "The CUDA architecture: The Art of performance optimization"

Pokročilá práce s pamětí

Textura generovaná pomocí CUDA:

```
// setup code:
cudaGLRegisterBufferObject(pbo);
// CUDA texture generation code:
unsigned char *d_buffer;
cudaGLMapBufferObject((void**)&d_buffer, pbo);
prep_texture_kernel<<<...>>>(d_buffer);
cudaGLUnmapBufferObject(pbo);
// OpenGL rendering code:
glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, pbo);
glBindTexture(GL_TEXTURE_2D, tex);
glTexSubImage2D(GL_TEXTURE_2D,0,0,0,256,256,GL_BGRA,GL_UNSIGNED_BYTE,0);
```

Zpracování snímku pomocí CUDA:

```
// OpenGL rendering code:
// ...
// CUDA post-processing code:
unsigned char *d_buffer;
cudaGLRegisterBufferObject(pbo);
cudaGLMapBufferObject((void**)&d_buffer, pbo);
post_process_kernel<<<...>>>(d_buffer);
cudaGLUnmapBufferObject(pbo);
cudaGLUnRegisterBufferObject(pbo);
```

Zdroj: Wil Braithwaite, "The CUDA architecture: The Art of performance optimization"

Pokročilá práce s pamětí

- Další triky s pamětí
  - Write-Combining

```
cudaHostAlloc((void**)&h_data, num_bytes,
              cudaHostAllocWriteCombined);
```

    - paměť není cacheovaná ani cache koherentní
    - PCI nedělá snooping
    - podle CUDA 2.2 Pinned Memory APIs (**http://www.fcsc.es/download/Archivo%20Cursos/CUDA_Unileon_2009/CUDA2.2PinnedMemoryAPIs.pdf**) může poskytnout až o 40 % větší výkon
    - může zvýšit i výkon pro zápis procesorem (Write Combining Memory Implementation Guidelines, **http://download.intel.com/design/PentiumII/applnots/24442201.pdf**): agregací zápisu, obcházením L1/L2 cache
    - problematické čtení – potřeba používat paměťové bariéry před čtením hodnot (**_mm_sfence** na Linuxu, **_WriteBarrier** na Windows, u SSE4 lze pro čtení použít instrukci **MOVNTDQA**); bariéry provádí CUDA driver a jsou relativně pomalé

Zdroj: Wil Braithwaite, "The CUDA architecture: The Art of performance optimization"

Pokročilá práce s pamětí

- Zero-copy mapování paměti
  - přímý přístup k datům v paměti CPU
  - schopnost tohoto se testuje pomocí pole `canMapHostMemory` dotazu `cudaDeviceProp`
  - automatický přenos dat po PCIe dle potřeby
  - relativně pomalé
    - pro jednorázově použitá malá data
    - pokud dokáže vysoký podíl výpočtu maskovat latenci
  - zajímavé zejména v kombinaci s integrovanými kartami – `integrated` pole `cudaDeviceProp`
    - na UMA architektuře odpadá přenos úplně
    - Nvidia ION

Zdroj: Wil Braithwaite, "The CUDA architecture: The Art of performance optimization"

Shrnutí práce s procesory

- Vlákna a multiprocesory
  - potřebujeme využít výpočetní výkon karty
  - kolik warpů potřebujeme k maskování latence globální paměti?
    - řekněme, že potřebujeme 100 aritmetických instrukcí k maskování latence (400 taktů latence / 4 takty na instrukci)
    - řekněme, že máme 8 aritmetických instrukcí (8×4 takty) na 1 přístup do globální paměti (400 taktů latence)
    - $^{100}/_{8} \approx 13$ warpů
  - kolik warpů potřebujeme k maskování read-after-write latence registrů?
    - latence je cca 24 taktů
    - $^{24}/_{4} \approx 6$ warpů

Zdroj: Wil Braithwaite, "The CUDA architecture: The Art of performance optimization"

Shrnutí práce s procesory

- Vlákna a multiprocesory
  - obsazení (occupancy)

$$O = \frac{\text{\# warpů běžících na MP v daný okamžik}}{\text{maximální \# souběžných warpů}}$$

    - c. c. 1.2: maximálně 32 warpů
      c. c. 1.1: maximálně 24 warpů
    - $^{13}/_{32} = 40\,\%$ obsazení pro maskování latence globální paměti (c. c. 1.2)
    - $^{6}/_{32} = 18{,}75\,\%$ obsazení pro maskování latence registrů (c. c. 1.2)
  - více vláken $\stackrel{?}{=}$ větší výkon
    - záleží na zdrojích požadovaných vláknem
    - záleží na uspořádání vláken

Zdroj: Wil Braithwaite, "The CUDA architecture: The Art of performance optimization"

Shrnutí práce s procesory

- Limity multiprocesoru:

```
kepler$ ./deviceQuery

 CUDA Device Query (Runtime API) version (CUDART static linking)

Found 1 CUDA Capable device(s)

Device 0: "GeForce GTX 580"
  CUDA Driver Version / Runtime Version          4.10 / 4.10
  CUDA Capability Major/Minor version number:    2.0
  Total amount of global memory:                 1535 MBytes (1609760768 bytes)
  (16) Multiprocessors x (32) CUDA Cores/MP:     512 CUDA Cores
  GPU Clock Speed:                               1.57 GHz
  Memory Clock rate:                             2010.00 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 786432 bytes
  Max Texture Dimension Size (x,y,z)             1D=(65536), 2D=(65536,65535), 3D=(2048,204
  Max Layered Texture Size (dim) x layers        1D=(16384) x 2048, 2D=(16384,16384) x 2048
```

Shrnutí práce s procesory

- Limity multiprocesoru:

```
  Total amount of constant memory:                65536 bytes
  Total amount of shared memory per block:        49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                      32
  Maximum number of threads per block:            1024
  Maximum sizes of each dimension of a block:     1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:      65535 x 65535 x 65535
  Maximum memory pitch:                           2147483647 bytes
  Texture alignment:                              512 bytes
  Concurrent copy and execution:                  Yes with 1 copy engine(s)
  Run time limit on kernels:                      No
  Integrated GPU sharing Host Memory:             No
  Support host page-locked memory mapping:        Yes
  Concurrent kernel execution:                    Yes
  Alignment requirement for Surfaces:             Yes
  Device has ECC support enabled:                 No
  Device is using TCC driver mode:                No
  Device supports Unified Addressing (UVA):       Yes
  Device PCI Bus ID / PCI location ID:            1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously)
```

- Pozor na dostatečný počet vláken/warpů v bloku (≥ 96)
- Pozor na limity v počtu potřebných registrů

Shrnutí práce s procesory

- Příklad limitů pro 8132 registrů a 24 warpů:
  - 10 registrů na vlákno, 256 vláken na blok
    - každý blok použije 2650 registrů $\Longrightarrow$ mohou běžet 3 bloky (7680 registrů)
    - $256 \times 3/32 = 24$ warpů může běžet současně $\Longrightarrow$ může dosáhnout 100 % využití
  - 17 registrů na vlákno, 256 vláken na blok
    - každý blok použije 4352 registrů $\Longrightarrow$ může běžet 1 blok (4352 registrů)
    - $256 \times 1/32 = 8$ warpů může běžet současně $\Longrightarrow$ může dosáhnout jen 33 % využití
  - 17 registrů na vlákno, 128 vláken na blok
    - každý blok použije 2176 registrů $\Longrightarrow$ mohou běžet 3 bloky (6528 registrů)
    - $128 \times 3/32 = 12$ warpů může běžet současně $\Longrightarrow$ může dosáhnout 50 % využití

Zdroj: Wil Braithwaite, "The CUDA architecture: The Art of performance optimization"

Shrnutí práce s procesory

- Určení používaných zdrojů
  - přeložíme s **–cubin**
  - výsledný .cubin soubor obsahuje

```
architecture {sm_10}
abiversion {0}
modname {cubin}
code {
        name = MyKernel
        lmem = 0    // lokalni pamet per blok
        smem = 68   // sdilena pamet per blok
        reg = 20    // pocet registru per vlakno
        bar = 0
        bincode {
                0xa0004205 0x04200780 0x40024c09 0x00200780
```

  - nebo použijeme –ptxas-options=-v

```
ptxas info : Used 4 registers, 60+56 bytes lmem, 44+40 bytes
smem, 20 bytes cmem[1], 12 bytes cmem[14]
```

Zdroj: Wil Braithwaite, "The CUDA architecture: The Art of performance optimization"

Shrnutí práce s procesory

- Omezení tlaku na registry
  - kompilátor se snaží počet registrů minimalizovat
  - `-maxrregcount=<N>` umožňuje nastavit požadovaný maximální počet registrů na kernel
  - přetečení do lokální paměti způsobí zpomalení
- Heuristiky pro velikost mřížky
  - # bloků > # multiprocesorů
    - aby všechny multiprocesory měly alespoň jeden blok k vykonávání
  - # bloků / # multiprocesorů > 2
    - na jednom procesoru může běžet více bloků
    - bloky nečekající v `__syncthreads();` udržují zátěž hardware
  - # bloků > 1000
    - rezerva do budoucnosti přes několik generací karet

Zdroj: Wil Braithwaite, "The CUDA architecture: The Art of performance optimization"

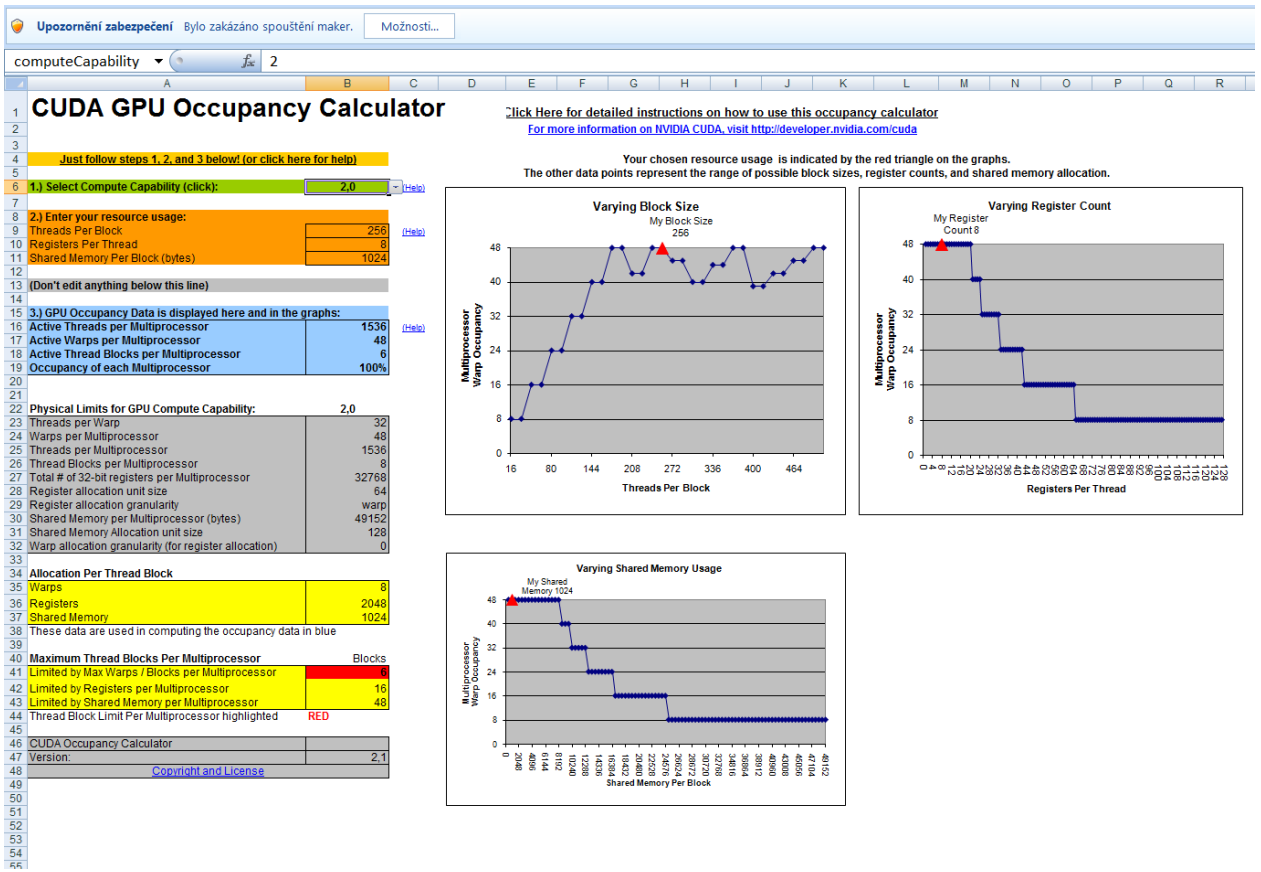
Shrnutí práce s procesory

- Heuristiky na velikost bloku
  - čím více vláken v bloku, tím méně může jedno vlákno používat registrů
  - čím více vláken v bloku, tím hlubší pipeline a lepší maskování latence
  - počty vláken v bloku by měly být násobky 64
    - minimalizace bank konfliktů na registrech
    - rozumné: 192 nebo 256

Zdroj: Wil Braithwaite, "The CUDA architecture: The Art of performance optimization"

Shrnutí práce s procesory

- CUDA Occupancy Calculator
  `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`



`Zdroj: Wil Braithwaite, "The CUDA architecture: The Art of performance optimization"`

Metriky z pohledu obrazových algoritmů

1. Paralelní podíl
2. Poměr mezi operacemi v plovoucí čárce a přístupů do paměti
3. Počet operací v plovoucí čárce na pixel
4. Počet přístupů do paměti na pixel
5. Míra větvení
6. Závislost úloh

- Srovnávané algoritmy implementovány na CPU
  - výpočet metrik před implementací na CUDA

Zdroj: Park *et al.*, "Design and Performance Evaluation of Image Processing Algorithms on GPUs"

Metriky z pohledu obrazových algoritmů

- Paralelní podíl
  - odpovídá Amdahlovu zákonu se všemi důsledky
  - při složeném algoritmu odpovídá poměrům částí
- Poměr mezi operacemi v plovoucí čárce a přístupů do paměti
  - jakékoli přístupy (sdílená, lokální, globální paměť)
  - skrývání latence překrýváním výpočtem
- Počet operací v plovoucí čárce (FP) na pixel
  - při výpočtech v plovoucí čárce překonávají GPU cca 20× CPU
  - obrazové zpracování zahrnuje typicky hodně operací v plovoucí čárce
  - nepřímo koreluje i s přístupy do paměti
- Počet přístupů do paměti na pixel
  - GPU mají přibližně 10× větší kapacitu přístupu do paměti než CPU
  - per pixel charakteristika umožňuje často využití sdílené paměti
  - problémy se sekvenčními přístupy do paměti u některých algoritmů – omezují paralelismus

Zdroj: Park *et al.*, "Design and Performance Evaluation of Image Processing Algorithms on GPUs"

Metriky z pohledu obrazových algoritmů

- Míra větvení
  - větvení přes `if`, `switch`, `do`, `for`, `while`
  - algoritmy pro zpracování obrázků často používají větvení na základě výsledku bitových operací
  - počítá se na základě rozptylu délky běhu jednotlivých vláken
  - berou se vlákna po blocích velikosti 32
- Závislost úloh
  - při zpracování obrázků se závislosti většinou řeší sekvenčním spouštěním CUDA kernelů
  - sleduje se počet bariér

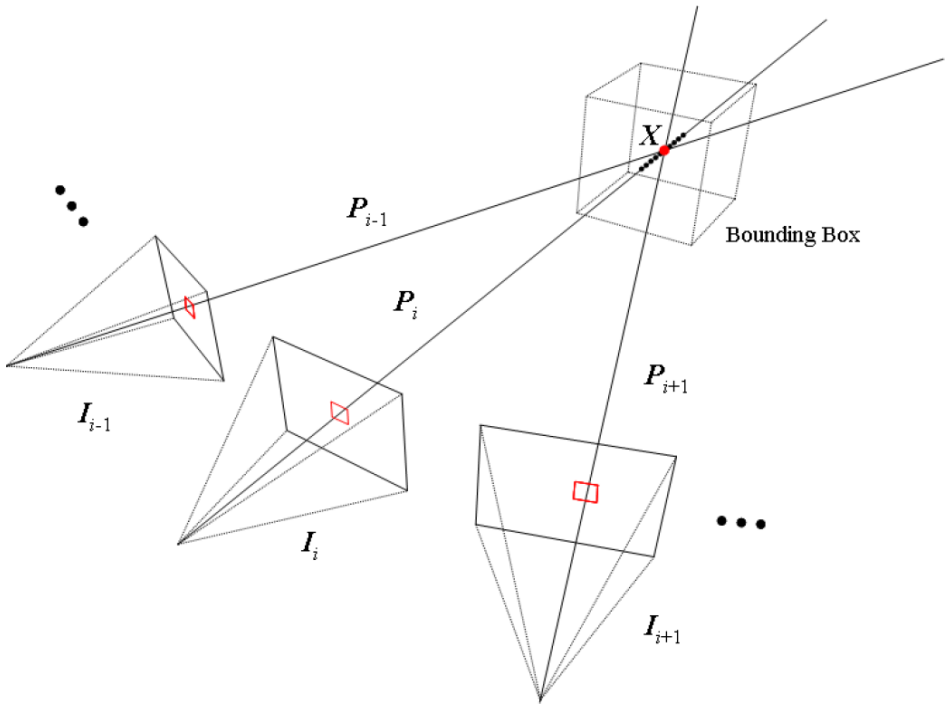
Zdroj: Park *et al.*, "Design and Performance Evaluation of Image Processing Algorithms on GPUs"

Relativní důležitost metrik

- Paralelní podíl > Větvení > FP operace per pixel >
  přístupy do paměti per pixel >
  poměr mezi FP operacemi a přístupy do paměti > závislost úloh
  1. Amdahl limituje vše
  2. omezení SIMT modelu
  3. algoritmy na zpracování obrazu mají obecně hodně FP operací
  4. poměr je závislý na předchozích dvou hodnotách (proč ho zavádět?)
  5. závislost úloh určuje obtížnost implementace (opravdu?)

Studované pilotní algoritmy

- Stereo srovnávání obrázků (Multiview Stereo Matching, MVS)



Studované pilotní algoritmy

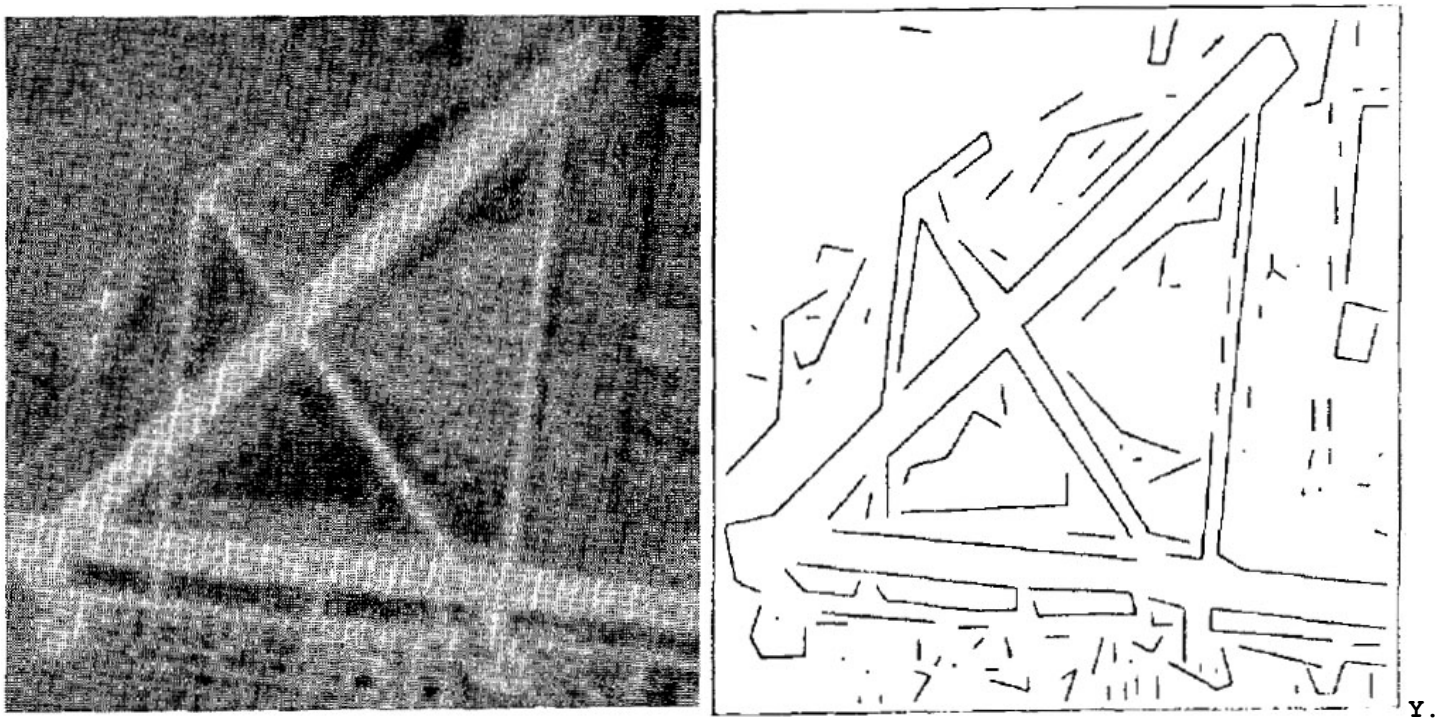
- Stereo srovnávání obrázků (Multiview Stereo Matching, MVS)
  - popis algoritmu
    - vstup:
      zkalibrované obrazy $I = I_0, \ldots, I_{N-1}$
      projekční matice $P = P_0, \ldots, P_{N-1}$
    - výstup:
      3D body $X = X_0, \ldots, X_{M-1}$
    - porovnávání lokálních oken mezi jednotlivými obrazy
    - hloubka se pro bod (x,y) v referenčním obrazu určí přeložením příslušné oblasti z I_i na oblasti v obraz I_{i-1}
    - bod s minimálním součtem absolutních odchylek (SAD) a normalizovanou cross-korelací (NCC) je uložen jako best-match
    - opakujeme pro ostatní sousedící obrazy (např. I_{i+1})
    - bod je korektně určen, pokud počet best-match je nad hranici $\mathrm{MIN\_COUNT}$
    - opakujeme pro všechny body a všechny referenční obrazy
  - složitost: $\mathcal{O}(N^2WHL)$
    N …počet vstupních obrazů, W resp. H …vodorovné resp. svislé rozlišení, L …velikost ohraničujícího boxu

Studované pilotní algoritmy

- Stereo srovnávání obrázků (Multiview Stereo Matching, MVS)
  - mapování na GPU
    - porovnávání lokálních oken je nezávislé – dobře mapuje na GPU
    - vlákno ~ pixel, tj. $W \times H$ vláken
    - $\mathcal{O}(N)$ volání kernelu pro výpočet hloubky pro jeden referenční obraz
    - $\mathcal{O}(L)$ volání uvnitř kernelu
    - obrázky se nakopírují do globální paměti
    - koeficienty lokálního okna jdou do sdílené paměti – častý přístup
    - složitost: $\mathcal{O}(\frac{N^2 WHL}{T_{max}})$
      T_{max}…maximální počet vláken na GPU (např. 12288 na G80)

Studované pilotní algoritmy

- Získávání lineárních charakteristik (Linear Feature Extraction)
  - rozpoznávací aplikace: budovy, silniční pruhy, …



T. Zhou, "Linear Feature Extraction Based on an AR Model Edge Detector"
http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=01169687

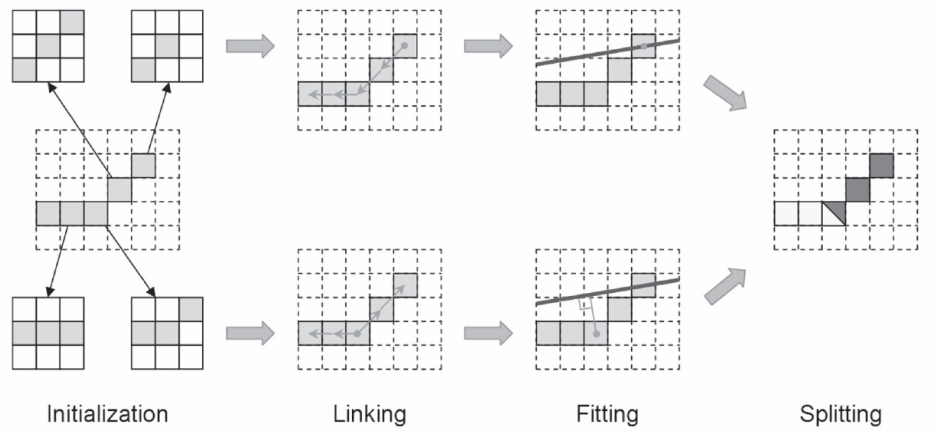
Studované pilotní algoritmy

- Získávání lineárních charakteristik (Linear Feature Extraction)
  - popis algoritmu
    - vstup: obrázek
    - výstup: obrázek s detekovanými hranami
    - použijeme algoritmus Nevatia-Babu s Cannyho detektorem hran

    1. detekce hran
    2. zúžení hran
    3. vytvoření řetízků hran na základě spojitosti v 8 směrech
    4. fitting čar na řetízky
    5. pokud segment čáry dává větší chybu než hraniční hodnota, je čára rozdělena na 2
    6. poslední krok iterativně opakujeme

Studované pilotní algoritmy

- Získávání lineárních charakteristik (Linear Feature Extraction)
  - mapování na GPU



    - 6 per-pixel kernelů

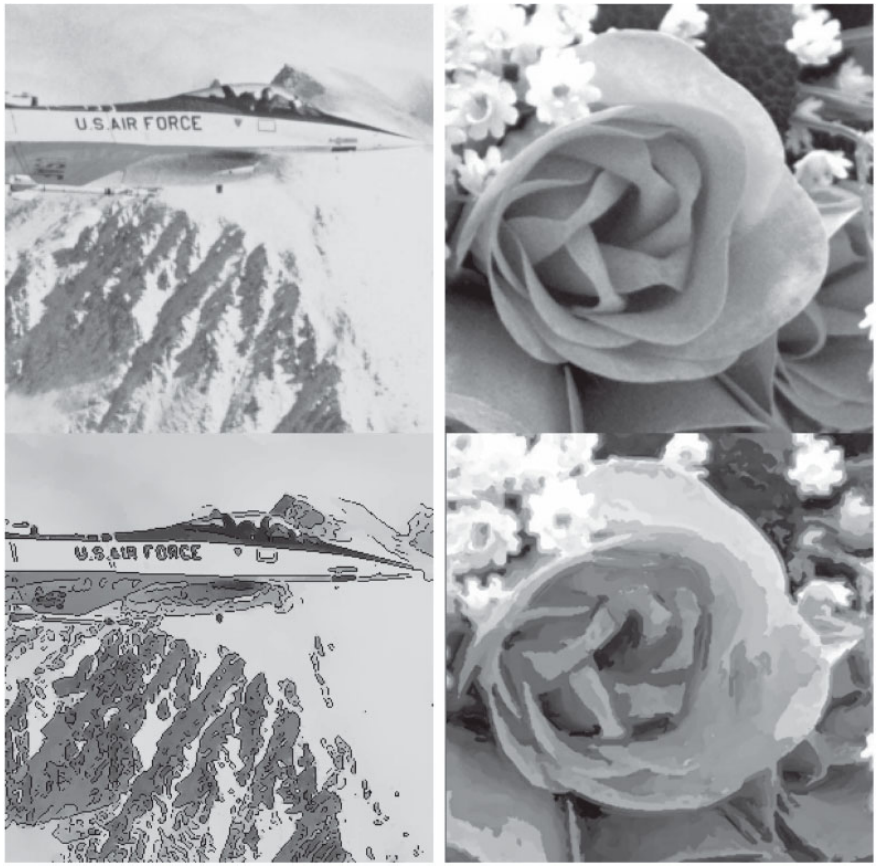
1. Cannyho detektor hran jako per-pixel filtr
2. klasifikace pixelů podle okolí 3×3 – Initialization
3. nalezení co nejdelších souvislých částí, určení počátečního a koncového bodu – Linking
4. výpočet odchylky od úsečky proložené mezi počáteční a koncový bod – Fitting
5. pokud odchylka přesahuje D_{max}, rozdělí se souvislý řetízek na dva
6. iterujeme poslední 3 kroky, dokud vznikají nové segmenty

Studované pilotní algoritmy

- Získávání lineárních charakteristik (Linear Feature Extraction)
  - mapování na GPU
    - počítá se pro všechny pixely – i ty, co hrany neobsahují $\Longrightarrow$ omezení
    - redundantní hledání nejdelších souvislých čar
    - významný podíl over-computation

Studované pilotní algoritmy

- Nefotorealistický rendering



  - Cartoon-style NPR
  - Oily-style NPR

Studované pilotní algoritmy

- Nefotorealistický rendering
  - popis algoritmu: Cartoon-style NPR
    - bilaterální filtrování (plochy)
    - Cannyho detekce hran (čáry)
    - přeložení čar přes plochy
  - mapování na GPU: Cartoon-style NPR
    - per pixel pro všechny operace
    - vstupní obrázky jsou uloženy ve 2D texturové paměti

Studované pilotní algoritmy

- Nefotorealistický rendering
  - popis algoritmu: Oily-style NPR
    - rozdíl původního obrazu a gaussovsky rozostřeného obrazu per pixel (míra rozostření udává tloušťku štětce)
    - tah se generuje, pokud oblast v součtu rozdílů dosahuje nad stanovenou hranici
    - tah začíná od lokálního maxima a vede po gradientu
    - tah se uloží pouze je-li dost dlouhý
    - uložené tahy se „obtáhnou štětcem“ dané tloušťky
    - proces začíná s čistým pozadím a opakuje se od nejtlustšího štětce k nejtenčímu
  - mapování na GPU: Oily-style NPR
    - paralelizace per tah: problém s překryvem tahů
    - paralelizace per pixel: problém jak vybrat poředí tahů
    - reformuluace/heuristika: světlejší štětec se použije později než tmavší
      $\Longrightarrow$ pro daný pixel vybereme nejsvětlejší barvu překrývajících se tahů
    - musíme hledat, kterými tahy bude pixel ovlivněn (maximální vzdálenost od trajektorie tahu)
    - vstupní obrázky jsou uloženy ve 2D texturové paměti

Přehled pilotních algoritmů

Characteristics / Algorithms	Processing Domain	Degree of Concurrency	Branching Diversity of Parallel Threads
Multiview Stereo Matching	Pixel	Massive	Low
Linear Feature Extraction	Pixel / Feature	Medium	Low (Edge) High (Linking, Fitting)
JPEG2000 Encoding	Pixel / Bitplanes	Massive (DWT) Low (EBCOT)	High
Non-Photorealistic Rendering	Pixel	Massive	Low

Characteristics / Algorithms	Floating-Point Arithmetic Intensity	Other Features
Multiview Stereo Matching	Low	-
Linear Feature Extraction	High	Overcomputation occurs
JPEG2000 Encoding	High (DWT) High (EBCOT)	-
Non-Photorealistic Rendering	High	Only pixelwise convolutions

Výsledky pilotních algoritmů

- charakteristika algoritmů

Characteristics / Algorithms	Parallel Fraction (↑)	FP Computation to Memory Access Ratio (↑)	FP Computation Per Pixel (↑)
Multiview Stereo Matching	0.994	1.838	4,900
Linear Feature Extraction	0.706	5.183	1,083
JPEG2000 Encoding (DWT)	0.983	3.630	552
JPEG2000 Encoding (Tier-1)	0.834	7.630	579
Cartoon-Style NPR	0.987	8.043	45,118
Oily-Style NPR	0.992	13.363	4,972

  - MVS kompenzuje malý poměr FLOP/mem vysokou mírou paralelismu
  - DWT kompenzuje málo FLOP/pixel vysokou mírou paralelismu
  - LFE má problém s nízkou mírou paralelismu

Výsledky pilotních algoritmů

- charakteristika algoritmů

Characteristics / Algorithms	Memory Access Per Pixel (↑)	Branching diversity (↓)	Task dependency (↓)
Multiview Stereo Matching	2,665	0.117	1
Linear Feature Extraction	209	0.113	11
JPEG2000 Encoding (DWT)	152	0.138	12
JPEG2000 Encoding (Tier-1)	76	0.307	1
Cartoon-Style NPR	5,609	0.156	6
Oily-Style NPR	372	0.121	34

  - task depenednecy: implementace bude stát hodně úsilí (schoval se nám díky zvolenému přístupu EBCOT Tier-1)
    $\Longrightarrow$ Oily-Style NPR má mnoho iterací, během nichž aktualizuje buffer
  - EBCOT Tier-1 má problém s větvením, nižším paralelismem a malým počtem FLOP/pixel

Výsledky pilotních algoritmů

- benchmarky CUDA

Characteristics / Algorithms	Global memory coalescing (↑)	SM to GM access ratio (↑)	Global memory Transfer (ms) (↓)	GPU occupancy (↑)
Multiview Stereo Matching	0.0013	0.020	107.28	33%
Linear Feature Extraction	0.0256	0.090	3.382	41.65%
JPEG2000 Encoding (DWT)	0.000	5.000	3.069	33.33% (vertical) 83.33% (horizontal)
JPEG2000 Encoding (Tier-1)	0.0167	1.903	0.307	17.00%
Cartoon-Style NPR	0.1439	0.000	0.435	40.18%
Oily-Style NPR	0.1517	0.823	0.435	52.19%

  - DWT: agregace dlaždic sníží potřebný počet registrů ($\Longrightarrow$ GPU occupancy)

Výsledky pilotních algoritmů

- benchmarky CUDA

Characteristics / Algorithms	Data dependency (↓)	Source lines (Host)	Kernel lines (Device)
Multiview Stereo Matching	2	184	187
Linear Feature Extraction	36	620	318
JPEG2000 Encoding (DWT)	2	132	112 (vertical) 90 (horizontal)
JPEG2000 Encoding (Tier-1)	2	>1500	406
Cartoon-Style NPR	0	223	232
Oily-Style NPR	4	514	635

  - Data Dependency: celkový počet volání `__syncthreads();` v rámci bloku vláken

Výsledky pilotních algoritmů

- zrychlení

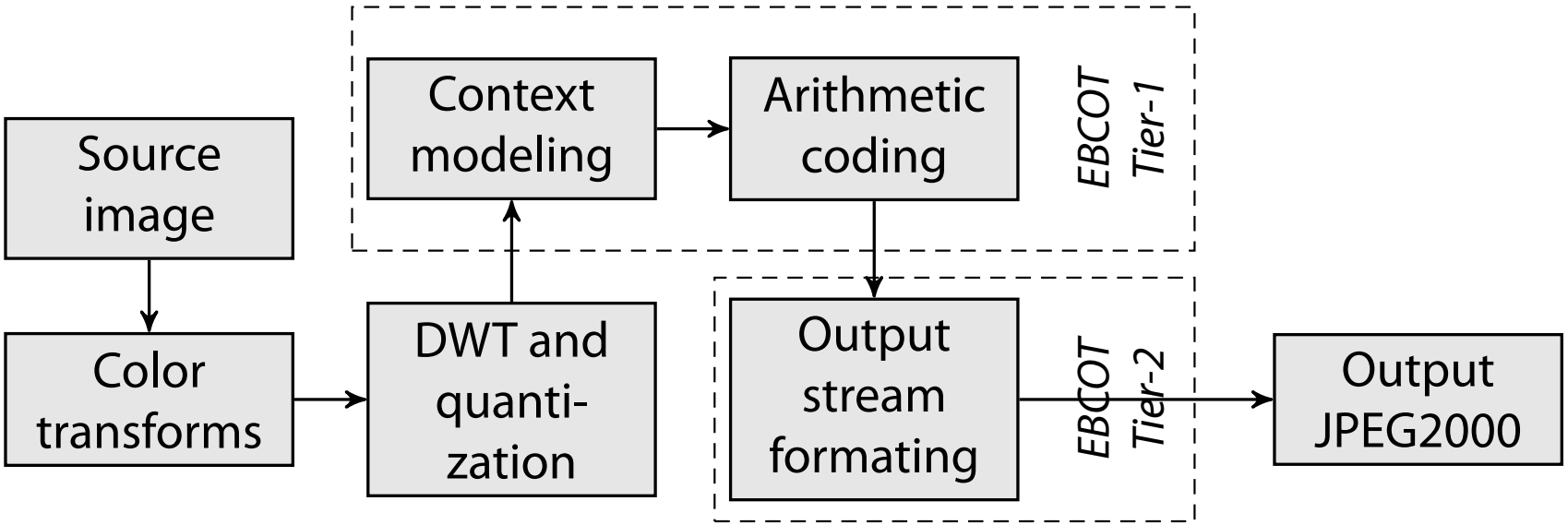
Algorithms	Data Resolution	Speedup		GPU Scalability
		G92/CPU	GX200/CPU	GX200/G92
Multiview Stereo Matching	TempleRing (47 Images)	54.18x	167.47x	3.09x
Cartoon-Style NPR	512 × 512	93.71x	149.84x	1.61x
	1024 × 768	97.51x	168.46x	1.73x
	1280 × 1024	76.79x	131.34x	1.71x
	1200 × 1800	117.30x	201.27x	1.72x
	2288 × 1712	126.64x	219.03x	1.73x
Oily-Style NPR	512 × 512	81.71x	129.72x	1.59x
	1024 × 768	69.00x	112.52x	1.63x
	1280 × 1024	105.47x	159.17x	1.51x
	1200 × 1800	83.16x	130.00x	1.56x
	2288 × 1712	85.27x	139.01x	1.63x

Výsledky pilotních algoritmů

- zrychlení

Algorithms	Data Resolution	Speedup		GPU Scalability
		G92/CPU	GX200/CPU	GX200/G92
Linear Feature Extraction	512 × 512	1.99x	2.42x	1.22x
	1024 × 768	2.53x	2.90x	1.14x
	1280 × 1024	2.43x	2.72x	1.11x
	1200 × 1800	2.65x	3.22x	1.22x
	2288 × 1712	2.33x	3.00x	1.29x
JPEG2000 Encoding (DWT)	512 × 512	4.06x	6.94x	1.71x
	1024 × 768	7.27x	12.92x	1.78x
	1280 × 1024	5.28x	8.98x	1.70x
	1200 × 1800	5.14x	9.13x	1.78x
	2288 × 1712	5.18x	9.26x	1.79x
	3024 × 2089	5.30x	9.34x	1.76x
JPEG2000 Encoding (Tier-1)	512 × 512	0.46x	0.35x	0.77x
	1024 × 768	0.60x	0.70x	1.17x
	1280 × 1024	0.68x	0.75x	1.11x
	1200 × 1800	1.21x	1.12x	0.92x
	2288 × 1712	1.12x	1.61x	1.44x
	3024 × 2089	0.73x	0.98x	1.35x

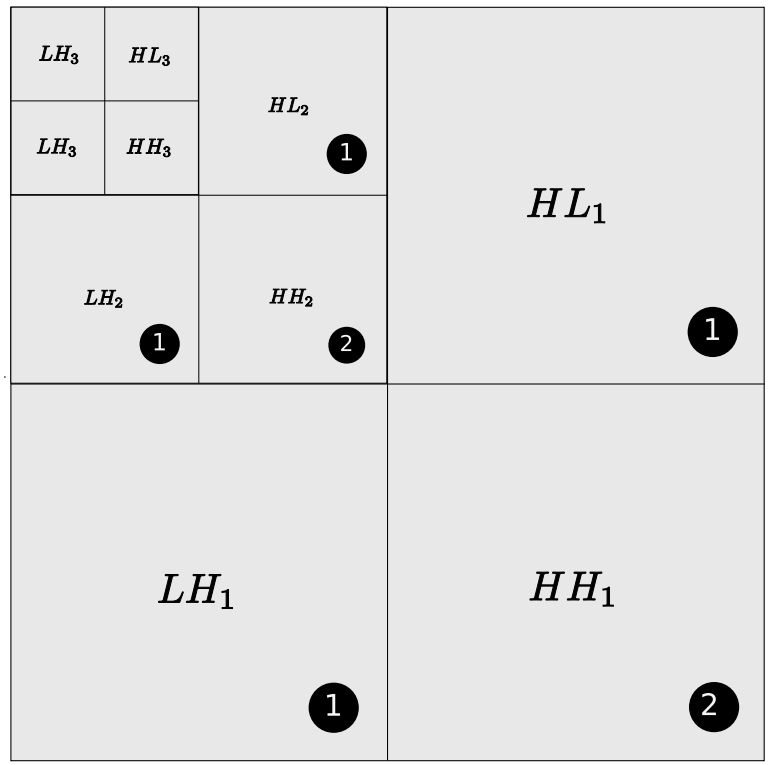
JPEG2000 – přehled procesu

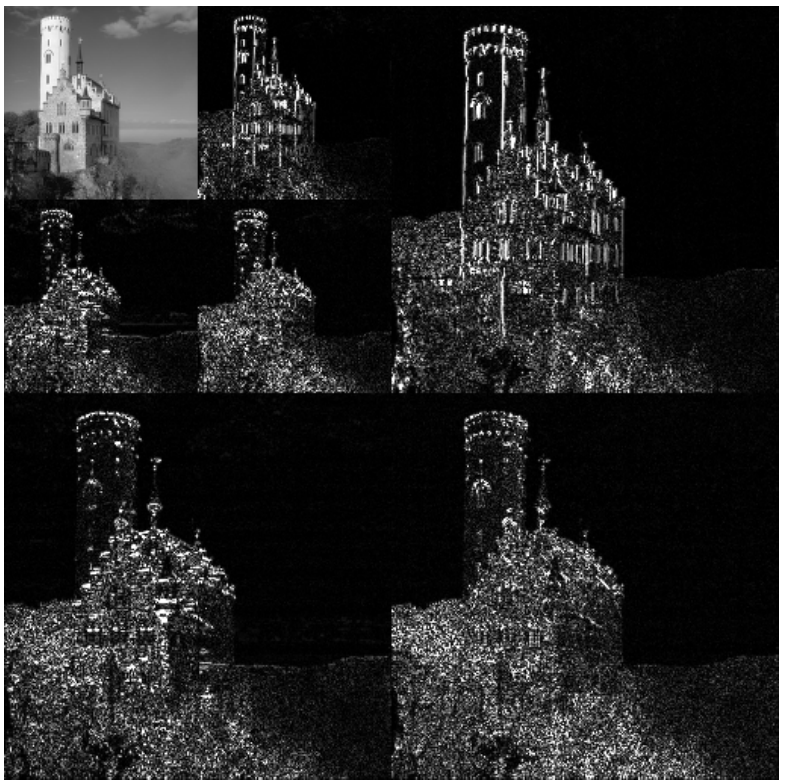


- Literatura z naší zahrádky :-))
  - MATELA, Jiří. GPU-Based DWT Acceleration for JPEG2000. In Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science. Brno : NOVPRESS s.r.o., 2009. od s. 136-143, 8 s. ISBN 978-80-87342-04-6.
  - MATELA, Jiří - RUSŇÁK, Vít - HOLUB, Petr. Efficient JPEG2000 EBCOT Context Modeling for Massively Parallel Architectures. In Storer, James A. and Marcellin, Michael W.. Data Compression Conference (DCC), 2011. Washington, DC, USA : IEEE Computer Society, 2011. od s. 423-432, 10 s. ISBN 978-0-7695-4352-9.
  - MATELA, Jiří - ŠROM, Martin - HOLUB, Petr. Low GPU Occupancy Approach to Fast Arithmetic Coding in JPEG2000. In Z. Kotásek et al.. MEMICS 2011, LNCS 7119 - to appear. Heidelberg : Springer, 2011. od s. 136-145.

Diskrétní vlnková transformace

- Rozložení obrazu do rekurzivně se opakujících pásů LL, HL, LH, HH





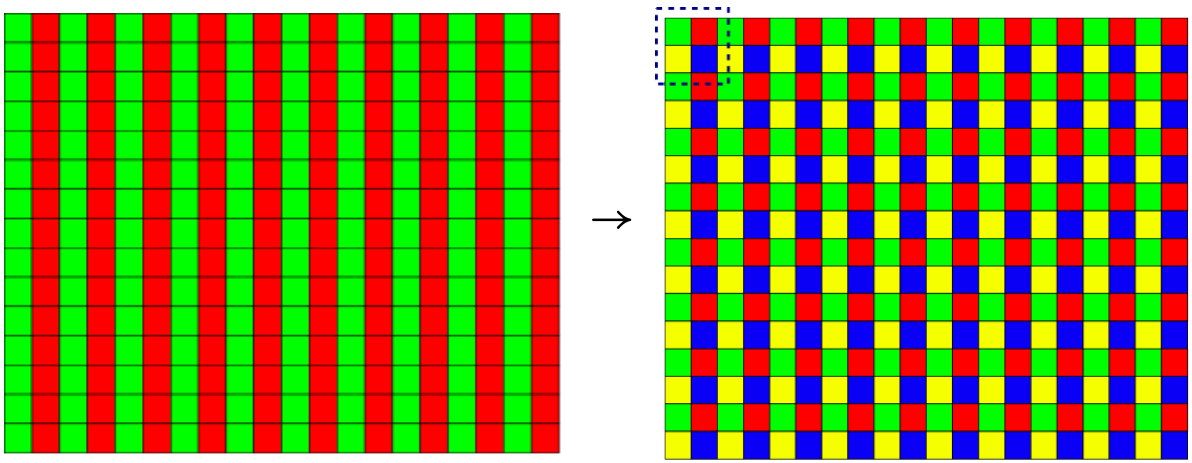
Diskrétní vlnková transformace

- Lifting schéma
  - low-pass a high-pass filtry

$$d_i^1 = d_i^0 - \frac{1}{2}(s_i^0 + s_{i+1}^0)$$

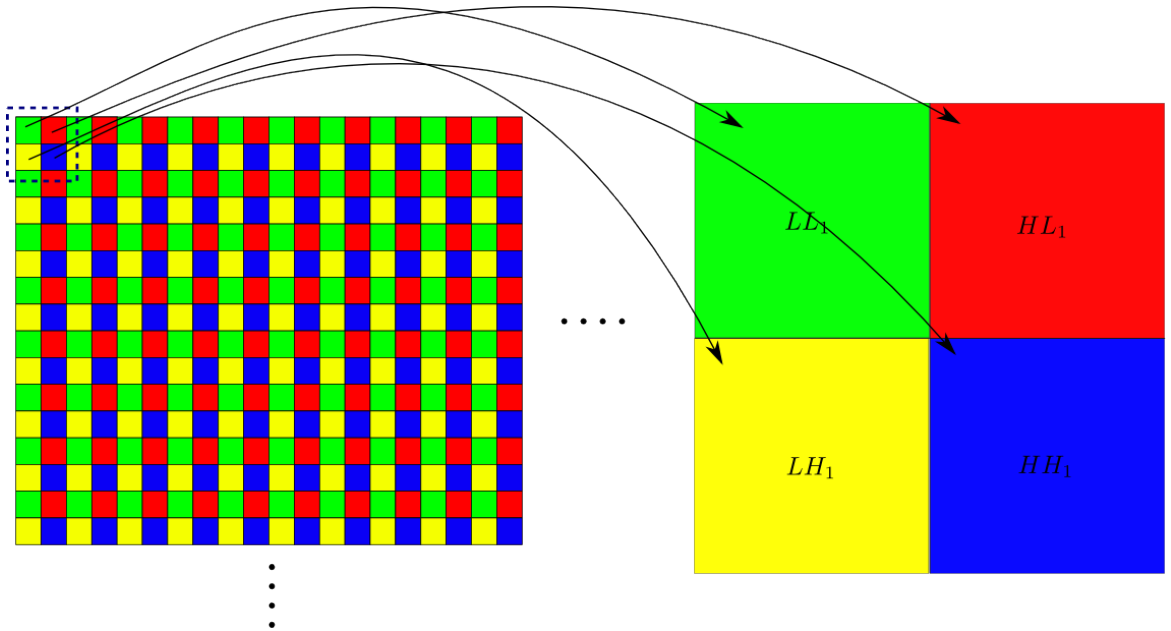
$$s_i^1 = s_i^0 + \frac{1}{4}(d_{i-1}^1 + d_i^1)$$

- Aplikace na řádky, poté na sloupce



Diskrétní vlnková transformace

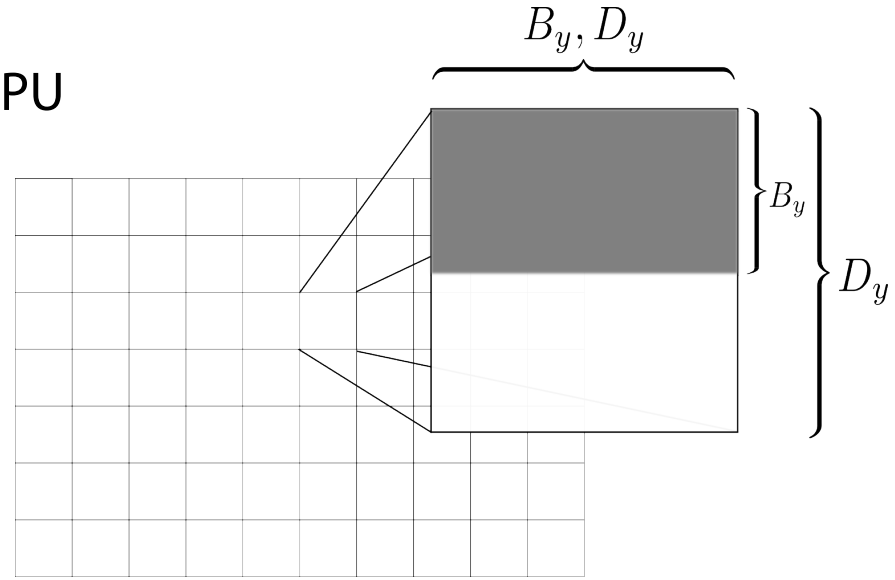
- Lifting schéma
  - přeuspořádání výsledného obrazu



- Mapování na GPU
  - 2D thread block
  - každé vlákno zpracovává jeden lichý a jeden sudý prvek

Diskrétní vlnková transformace

- Mapování na GPU



1. načtení z globální do sdílené paměti
2. aplikace lifting schéma na řádky

$$s[T_x][2T_y+1] = s[T_x][2T_y+1] + p \cdot (s[T_x][2T_y] + s[T_x][2T_y+2])$$
$$s[T_x][2T_y] = s[T_x][2T_y] + u \cdot (s[T_x][2T_y-1] + s[2T_y+1])$$

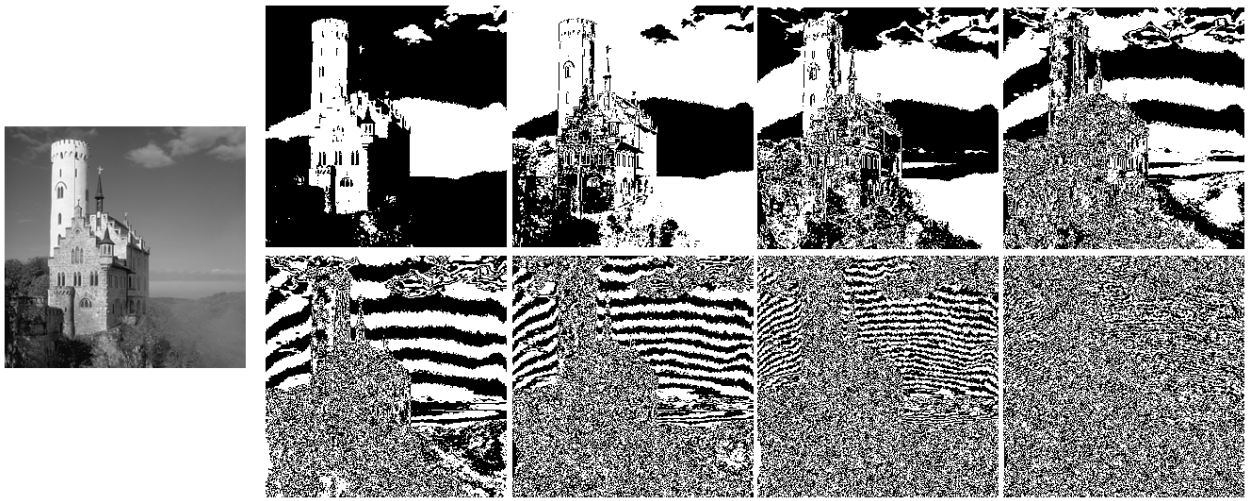
3. aplikace lifting schéma na sloupce (prohození T_x a T_y)
4. uložení výsledků do globální paměti s přeuspořádáním
   - uložení sudých pak lichých řádků
   - v rámci řádku první polovina vláken čte LL (resp. LH), druhá polovina HL (resp. HH) $\Longrightarrow$ koalescentní přístup do globální paměti

Diskrétní vlnková transformace

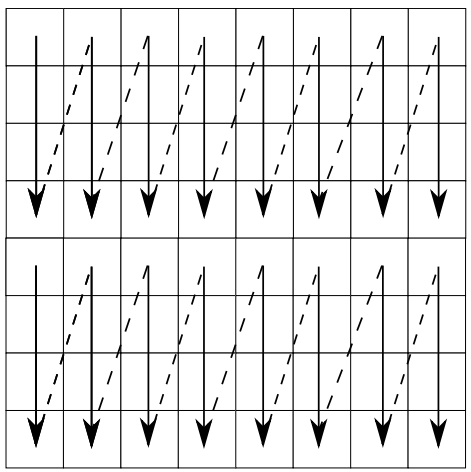
Implementation	512×512	1920×1080	Speedup
JasPer	6ms	55ms	N/A×
CUDA DWT	0.12ms	0.81ms	67.9×

Modelování kontextu

- Hledání kontextu pro kompresi aritmetickým adaptivním kodérem
- Jde se po jednotlivých bitplanech

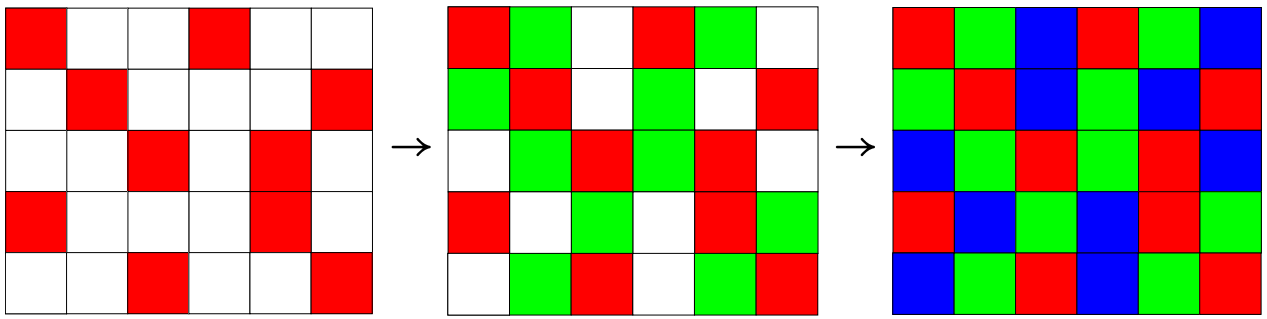


- Bitplane skenován podle vzoru (scan-pattern)



Modelování kontextu

- 3 fáze:
  1. Significance Propagation Pass (SPP)
  2. Magnitude Refinement Pass (MRP)
  3. Cleanup Pass (CUP)



Modelování kontextu

- 4 kódovací operace
  1. Zero Coding (ZC)
  2. Run-Length Coding (RLC)
  3. Magnitude Refinement Coding (MRC)
  4. Sign Coding (SC)
- každý bit je zakódován jednou nebo více operacemi právě v jedné fázi
- výstupem kódovacích operací je pár *CX,D*

Modelování kontextu

- Určení fáze pro daný bit
  - záleží na jeho stavu
  - záleží na stavu jeho sousedů
  - to vše se vyvíjí (proměnné σ, σ', η), jak bity prochází jednotlivými fázemi podle scan-pattern – :-((((((((((
- Dle definice vysoce sekvenční proces
  - umožňuje paralelizmus jedině na úrovni code block
  - … s jistými se speciálními výjimkami (causal mode)

Modelování kontextu

- Reformulace problému
  - náhrada původních stavových proměnných σ, σ', η za proměnné ρ, τ
  - nové proměnné lze předpočítat paralelně pro každý bit
  - výpočet ρ

Bitplane index $1 \ldots n$

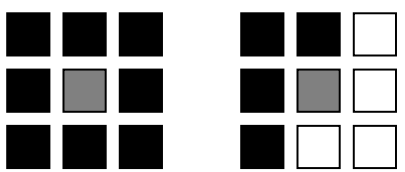
Value of a pixel at x, y

$$\rho^{p}_{x,y} = \begin{cases} 1, & \text{if } \gamma_{x,y} > 2^{p} \\ 0, & \text{otherwise} \end{cases}$$

Pixel position in the code block

Modelování kontextu

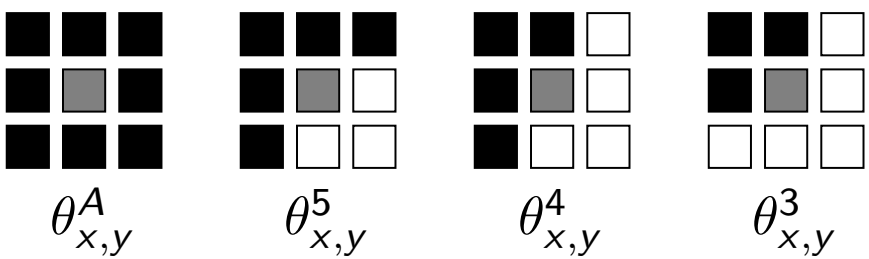
- Reformulace problému
  - výpočet τ
    1. $\tau^p_{x,y} = 1 \Leftrightarrow \rho^{p+1}_{x,y} = 0 \wedge \gamma^p_{x,y} = 1 \wedge \bigvee_{(i,j)\in\theta^A_{x,y}} (\rho^{p+1}_{i,j} = 1)$
    2. $\tau^p_{x,y} = 1 \Leftrightarrow \rho^{p+1}_{x,y} = 0 \wedge \gamma^p_{x,y} = 1 \wedge \bigvee_{(i,j)\in\theta^4_{x,y}} (\tau^{p+1}_{i,j} = 1)$
    3. Step 2 is repeated until there is no new $\tau^p_{x,y} = 1$ found



$\theta^A_{x,y}$ $\theta^4_{x,y}$

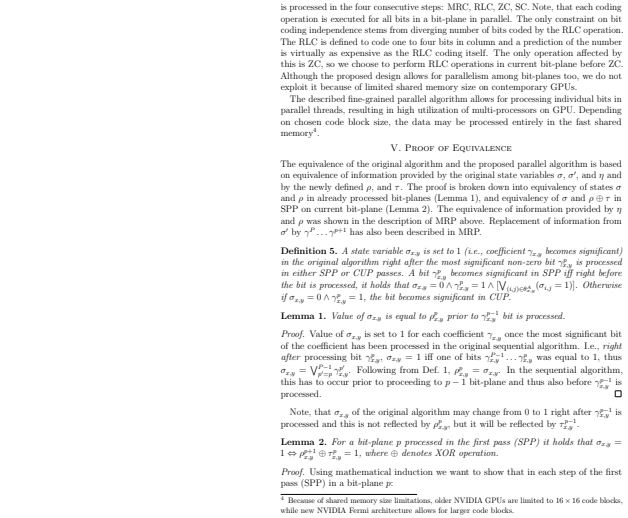
Modelování kontextu

- Reformulace problému
  - pomocná proměnná δ
    - $\delta^p_{x,y} = 1$ indicates a position $\gamma^p_{x,y}$ is in PN in SPP
    - $\delta^p_{x,y} = \bigvee_{(i,j)\in\theta^A_{x,y}} (\rho^{p+1}_{i,j} = 1) \vee \bigvee_{(i,j)\in\{\theta^5_{x,y},\theta^4_{x,y},\theta^3_{x,y}\}} (\tau^p_{i,j} = 1)$
    - Where selection of surrounding $\theta_{x,y}$ depends on position of the bit $\gamma^p_{x,y}$ on y-axis.



Modelování kontextu

- Důkazy ekvivalence
  - **Původní:**
    Bit je kódován v SPP, pokud není signifikantní ($\sigma = 0$) a je v preferovaném okolí (preferred neighborhood).
  - **Paralelní:**
    Bit je kódován v SPP, pokud není signifikantní ($\rho^{p+1} = 0$) a je v preferovaném okolí ($\delta^p = 1$).
  - …následuje důkaz ekvivalence indukcí ;-)))


is processed in the four consecutive steps: MRC, RLC, ZC, SC. Note, that each coding operation is executed for all bits in a bit-plane in parallel. The only constraint on bit coding independence stems from diverging number of bits coded by the RLC operation. The RLC is defined to code one to four bits in column and a prediction of the number is virtually as expensive as the RLC coding itself. The only operation affected by this is ZC, so we choose to perform RLC operations in current bit-plane before ZC. Although the proposed design allows for parallelism among bit-planes too, we do not exploit it because of limited shared memory size on contemporary GPUs.

The described fine-grained parallel algorithm allows for processing individual bits in parallel threads, resulting in high utilization of multi-processors on GPU. Depending on chosen code block size, the data may be processed entirely in the fast shared memory[4].

V. Proof of Equivalence

The equivalence of the original algorithm and the proposed parallel algorithm is based on equivalence of information provided by the original state variables σ, σ', and η and by the newly defined ρ, and τ. The proof is broken down into equivalency of states σ and ρ in already processed bit-planes (Lemma 1), and equivalency of σ and $\rho \oplus \tau$ in SPP on current bit-plane (Lemma 2). The equivalence of information provided by η and ρ was shown in the description of MRP above. Replacement of information from σ' by $\gamma^{p} \ldots \gamma^{p+1}$ has also been described in MRP.

Definition 5. *A state variable $\sigma_{x,y}$ is set to* 1 *(i.e., coefficient $\gamma_{x,y}$ becomes significant) in the original algorithm right after the most significant non-zero bit $\gamma^p_{x,y}$ is processed in either SPP or CUP passes. A bit $\gamma^p_{x,y}$ becomes significant in SPP iff right before the bit is processed, it holds that $\sigma_{x,y} = 0 \wedge \gamma^p_{x,y} = 1 \wedge \bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\sigma_{i,j} = 1)$]. Otherwise if $\sigma_{x,y} = 0 \wedge \gamma^p_{x,y} = 1$, the bit becomes significant in CUP.*

Lemma 1. *Value of $\sigma_{x,y}$ is equal to $\rho^p_{x,y}$ prior to $\gamma^{p-1}_{x,y}$ bit is processed.*

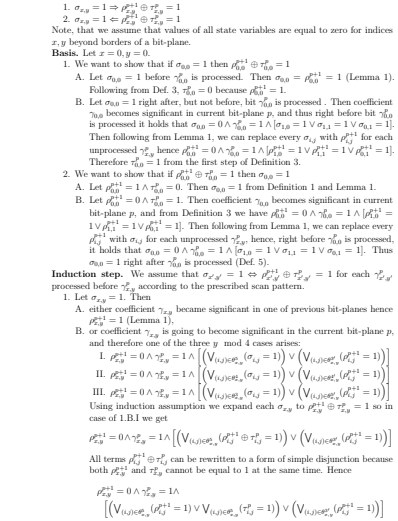
Proof. Value of $\sigma_{x,y}$ is set to 1 for each coefficient $\gamma_{x,y}$ once the most significant bit of the coefficient has been processed in the original sequential algorithm. I.e., *right after* processing bit $\gamma^p_{x,y}$, $\sigma_{x,y} = 1$ iff one of bits $\gamma^{P-1}_{x,y} \ldots \gamma^p_{x,y}$ was equal to 1, thus $\sigma_{x,y} = \bigvee_{p'=p}^{P-1} \gamma^{p'}_{x,y}$. Following from Def. 1, $\rho^p_{x,y} = \sigma_{x,y}$. In the sequential algorithm, this has to occur prior to proceeding to $p-1$ bit-plane and thus also before $\gamma^{p-1}_{x,y}$ is processed. □

Note, that $\sigma_{x,y}$ of the original algorithm may change from 0 to 1 right after $\gamma^{p-1}_{x,y}$ is processed and this is not reflected by $\rho^p_{x,y}$, but it will be reflected by $\tau^{p-1}_{x,y}$.

Lemma 2. *For a bit-plane p processed in the first pass (SPP) it holds that $\sigma_{x,y} = 1 \Leftrightarrow \rho^{p+1}_{x,y} \oplus \tau^p_{x,y} = 1$, where $\oplus$ denotes XOR operation.*

Proof. Using mathematical induction we want to show that in each step of the first pass (SPP) in a bit-plane p.

[4] Because of shared memory size limitations, older NVIDIA GPUs are limited to 16×16 code blocks, while new NVIDIA Fermi architecture allows for larger code blocks.


1. $\sigma_{x,y} = 1 \Rightarrow \rho^{p+1}_{x,y} \oplus \tau^p_{x,y} = 1$
2. $\sigma_{x,y} = 1 \Leftarrow \rho^{p+1}_{x,y} \oplus \tau^p_{x,y} = 1$

Note, that we assume that values of all state variables are equal to zero for indices x, y beyond borders of a bit-plane.

Basis. Let $x = 0, y = 0$.

1. We want to show that if $\sigma_{0,0} = 1$ then $\rho^{p+1}_{0,0} \oplus \tau^p_{0,0} = 1$
   A. Let $\sigma_{0,0} = 1$ before $\gamma^p_{0,0}$ is processed. Then $\sigma_{0,0} = \rho^{p+1}_{0,0} = 1$ (Lemma 1). Following from Def. 3, $\tau^p_{0,0} = 0$ because $\rho^{p+1}_{0,0} = 1$.
   B. Let $\sigma_{0,0} = 1$ right after, but not before, bit $\gamma^p_{0,0}$ is processed . Then coefficient $\gamma_{0,0}$ becomes significant in current bit-plane p, and thus right before bit $\gamma^p_{0,0}$ is processed it holds that $\sigma_{0,0} = 0 \wedge \gamma^p_{0,0} = 1 \wedge [\sigma_{1,0} = 1 \vee \sigma_{1,1} = 1 \vee \sigma_{0,1} = 1]$. Then following from Lemma 1, we can replace every $\sigma_{i,j}$ with $\rho^{p+1}_{i,j}$ for each unprocessed $\gamma^p_{x,y}$ hence $\rho^{p+1}_{0,0} = 0 \wedge \gamma^p_{0,0} = 1 \wedge [\rho^{p+1}_{1,0} = 1 \vee \rho^{p+1}_{1,1} = 1 \vee \rho^{p+1}_{0,1} = 1]$. Therefore $\tau^p_{0,0} = 1$ from the first step of Definition 3.
2. We want to show that if $\rho^{p+1}_{0,0} \oplus \tau^p_{0,0} = 1$ then $\sigma_{0,0} = 1$
   A. Let $\rho^{p+1}_{0,0} = 1 \wedge \tau^p_{0,0} = 0$. Then $\sigma_{0,0} = 1$ from Definition 1 and Lemma 1.
   B. Let $\rho^{p+1}_{0,0} = 0 \wedge \tau^p_{0,0} = 1$. Then coefficient $\gamma_{0,0}$ becomes significant in current bit-plane p, and from Definition 3 we have $\rho^{p+1}_{0,0} = 0 \wedge \gamma^p_{0,0} = 1 \wedge [\rho^{p+1}_{1,0} = 1 \vee \rho^{p+1}_{1,1} = 1 \vee \rho^{p+1}_{0,1} = 1]$. Then following from Lemma 1, we can replace every $\rho^{p+1}_{i,j}$ with $\sigma_{i,j}$ for each unprocessed $\gamma^p_{x,y}$, hence, right before $\gamma^p_{0,0}$ is processed, it holds that $\sigma_{0,0} = 0 \wedge \gamma^p_{0,0} = 1 \wedge [\sigma_{1,0} = 1 \vee \sigma_{1,1} = 1 \vee \sigma_{0,1} = 1]$. Thus $\sigma_{0,0} = 1$ right after $\gamma^p_{0,0}$ is processed (Def. 5).

Induction step. We assume that $\sigma_{x',y'} = 1 \Leftrightarrow \rho^{p+1}_{x',y'} \oplus \tau^p_{x',y'} = 1$ for each $\gamma^p_{x',y'}$ processed before $\gamma^p_{x,y}$ according to the prescribed scan pattern.

1. Let $\sigma_{x,y} = 1$. Then
   A. either coefficient $\gamma_{x,y}$ became significant in one of previous bit-planes hence $\rho^{p+1}_{x,y} = 1$ (Lemma 1),
   B. or coefficient $\gamma_{x,y}$ is going to become significant in the current bit-plane p, and therefore one of the three $y \mod 4$ cases arises:
      I. $\rho^{p+1}_{x,y} = 0 \wedge \gamma^p_{x,y} = 1 \wedge \left[\left(\bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\sigma_{i,j} = 1)\right) \vee \left(\bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\rho^{p+1}_{i,j} = 1)\right)\right]$
      II. $\rho^{p+1}_{x,y} = 0 \wedge \gamma^p_{x,y} = 1 \wedge \left[\left(\bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\sigma_{i,j} = 1)\right) \vee \left(\bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\rho^{p+1}_{i,j} = 1)\right)\right]$
      III. $\rho^{p+1}_{x,y} = 0 \wedge \gamma^p_{x,y} = 1 \wedge \left[\left(\bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\sigma_{i,j} = 1)\right) \vee \left(\bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\rho^{p+1}_{i,j} = 1)\right)\right]$

   Using induction assumption we expand each $\sigma_{x,y}$ to $\rho^{p+1}_{x,y} \oplus \tau^p_{x,y} = 1$ so in case of 1.B.I we get

$$\rho^{p+1}_{x,y} = 0 \wedge \gamma^p_{x,y} = 1 \wedge \left[\left(\bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\rho^{p+1}_{i,j} \oplus \tau^p_{i,j} = 1)\right) \vee \left(\bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\rho^{p+1}_{i,j} = 1)\right)\right]$$

   All terms $\rho^{p+1}_{i,j} \oplus \tau^p_{i,j}$ can be rewritten to a form of simple disjunction because both $\rho^{p+1}_{i,j}$ and $\tau^p_{i,j}$ cannot be equal to 1 at the same time. Hence

$$\rho^{p+1}_{x,y} = 0 \wedge \gamma^p_{x,y} = 1 \wedge \left[\left(\bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\rho^{p+1}_{i,j} = 1) \vee \bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\tau^p_{i,j} = 1)\right) \vee \left(\bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\rho^{p+1}_{i,j} = 1)\right)\right]$$

Both $\rho^{p+1}_{x,y}$ can be combined together with respect to surroundings θ. Hence

$$\rho^{p+1}_{x,y} = 0 \wedge \gamma^p_{x,y} = 1 \wedge \left[\left(\bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\rho^{p+1}_{i,j} = 1)\right) \vee \left(\bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\tau^p_{i,j} = 1)\right)\right]$$

Therefore $\tau^p_{x,y} = 1$ from Definition 3. The proof of the cases 1.B.II and 1.B.III is analogical to the 1.B.I.

2. Let $\rho^{p+1}_{x,y} \oplus \tau^p_{x,y} = 1$. Then
   A. either $\rho^{p+1}_{x,y} = 1$, and thus coefficient $\gamma_{x,y}$ became significant in one of previous bit-planes, hence $\sigma_{x,y} = 1$ (Lemma 1),
   B. or coefficient $\gamma_{x,y}$ is going to become significant in current bit-plane, hence one of the three $y \mod 4$ cases arises:
      I. $\rho^{p+1}_{x,y} = 0 \wedge \gamma^p_{x,y} = 1 \wedge \left[\left(\bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\rho^{p+1}_{i,j} \oplus \tau^p_{i,j} = 1)\right) \vee \left(\bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\rho^{p+1}_{i,j} = 1)\right)\right]$
      II. $\rho^{p+1}_{x,y} = 0 \wedge \gamma^p_{x,y} = 1 \wedge \left[\left(\bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\rho^{p+1}_{i,j} \oplus \tau^p_{i,j} = 1)\right) \vee \left(\bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\rho^{p+1}_{i,j} = 1)\right)\right]$
      III. $\rho^{p+1}_{x,y} = 0 \wedge \gamma^p_{x,y} = 1 \wedge \left[\left(\bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\rho^{p+1}_{i,j} \oplus \tau^p_{i,j} = 1)\right) \vee \left(\bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\rho^{p+1}_{i,j} = 1)\right)\right]$

   Using induction assumption we expand each $\rho^{p+1}_{x,y} \oplus \tau^p_{x,y} = 1$ to $\sigma_{x,y} = 1$ so in case of 2.B.I we get

$$\rho^{p+1}_{x,y} = 0 \wedge \gamma^p_{x,y} = 1 \wedge \left[\left(\bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\sigma_{i,j} = 1)\right) \vee \left(\bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\rho^{p+1}_{i,j} = 1)\right)\right]$$

   Each ρ can be rewritten to σ because those two state variables has same value before bit $\gamma^p_{x,y}$ is processed in current bit-plane p (Lemma 1). Hence

$$\sigma_{x,y} = 0 \wedge \gamma^p_{x,y} = 1 \wedge \left[\left(\bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\sigma_{i,j} = 1)\right) \vee \left(\bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\sigma_{i,j} = 1)\right)\right]$$

   Surroundings $\theta^y_{x,y}$ and $\theta^{y'}_{x,y}$ can be folded, hence

$$\sigma_{x,y} = 0 \wedge \gamma^p_{x,y} = 1 \wedge \left[\bigvee_{(i,j)\in\mathcal{H}^p_{x,y}}(\sigma_{i,j} = 1)\right]$$

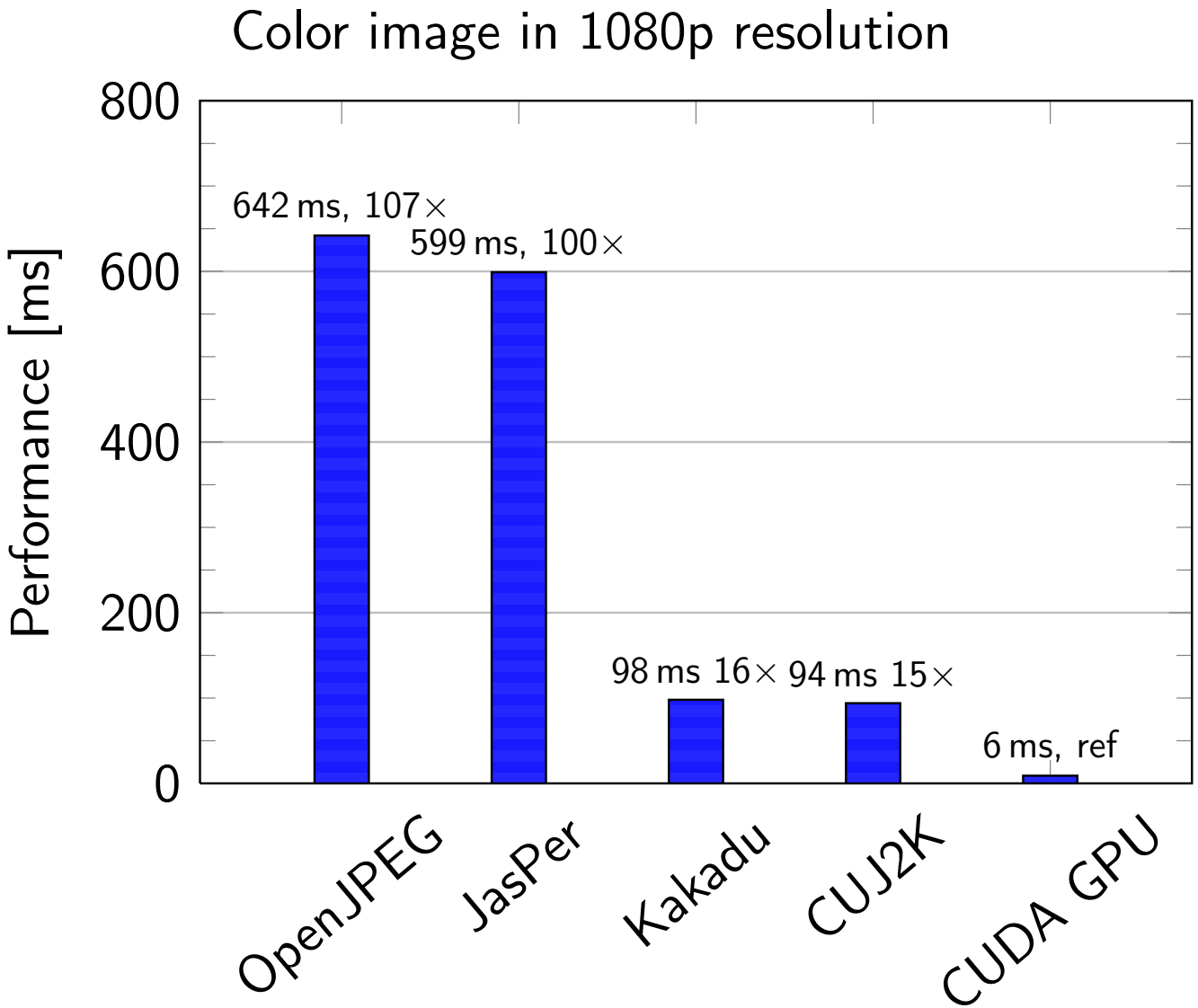
   Therefore following Def. 5 state $\sigma_{x,y} = 1$ right after bit $\gamma^p_{x,y}$ is coded. □

VI. Experimental Results

Methodology. We set up two benchmark sets focused on the EBCOT Tier-1 processing speed only. Performance of bpcuda (our GPU implementation) was compared with two open-source CPU implementations (OpenJPEG[5] 1.2, JasPer[6] 1.900.1), one commercial (Kakadu[7] 2.2.3) and one GPU implementation (CUJ2K[8] 1.1). All the CPU implementations were single-threaded.

[5] http://www.openjpeg.org [6] http://www.ece.uvic.ca/~mdadams/jasper/
[7] http://www.kakadusoftware.com [8] http://cuj2k.sourceforge.net

Modelování kontextu



Aritmetické kódování – MQ-Coder

- Binární aritmetický kodér
  - LPS do $[0,Q)$
  - MPS do $[Q,A]$
  - Q je pravděpodobnost výskytu LPS
  - výsledek C
- Adaptivní
  - mění se význam LPS/MPS
- Kontextový
  - vstup má přiřazen kontext
  - kontextu určuje stav kodéru
    - mapování symbolů na MPS/LPS
    - pravděpodobnost LPS

MPS	$A = A(1-Q)$	$C = C + AQ$
LPS	$A = AQ$	$C = C$

Aritmetické kódování – MQ-Coder

- Problémy
  - příliš hrubý paralelismus
  - podmíněné větvené závislé na vstupních datech
  - nedá se odstěhovat na CPU – na GPU máme tou dobou 2× tolik dat (*CX,D*)
- Pozitiva
  - není tak náročné jako kontextové modelování

	720p	*1080p*	*4K*
Basic	**38 ms**	**45 ms**	**93 ms**

- Nenašli jsme způsob principiální paralelizace :-(
  - analýza různých optimalizačních technik
  - někteří výzkumníci vymýšlejí jiné kodéry – problém nekompatibility se standardem

Aritmetické kódování – MQ-Coder

- Převod implementace do registrů
  - lokální datové struktury do registrů
  - průměrně: 240% zrychlení

  $\Longrightarrow$ registry jsou opravdu nejrychlejší (i vůči sdílené paměti)
- Rozvinování cyklů
  - navrženo jako optimalizační technika pro VLSI
  - zpracovává sérii MPS symbolů, pokud mají přiřazené stejné *CX*
  - použití MAD instrukce: $A = A - nQ_e$, $C = C + nQ_e$
  - paralelní prohledávání – detekce sekvencí
  - průměrně: 31% zpomalení

  $\Longrightarrow$ zrychlení zpracování ani nevyváží režii předpočítání

  $\Longrightarrow$ zastoupení sérií MPS symbolů není dostatečně časté (17–25 % sekvencí 2–32)

Aritmetické kódování – MQ-Coder

- Prefix sum
  - zpracovává sérii MPS symbolů nezávisle na *CX*
  - nemůže používat MAD (různé *CX* znamená různé Q_e)
  - Q_e jsou sčítány paralelně, součet použit pro úpravu *A* a *C*

Aritmetické kódování – MQ-Coder

- Renormalizace
  - pokud $A \approx 1$, můžeme aproximovat: $A(1-Q) \approx A - Q$ a $AQ = Q$
  - omezení počtu násobení
- Vylepšená rozšířená renormalizace
  - renormalizace se provádí, pokud A klesne pod 0x8000
  - n-krát se násobí A a C, až je $A > 0x8000$
  - existující návrh urychlení: n určit vyhledávací tabulkou a použít shift
  - publikovaný algoritmus měl chybu – neřešil korektně přetečení
  - můžeme využít CLZ instrukci místo vyhledávací tabulky
  - průměrně: 39% zrychlení

$\Longrightarrow$ shifty a CLZ jsou rychlé

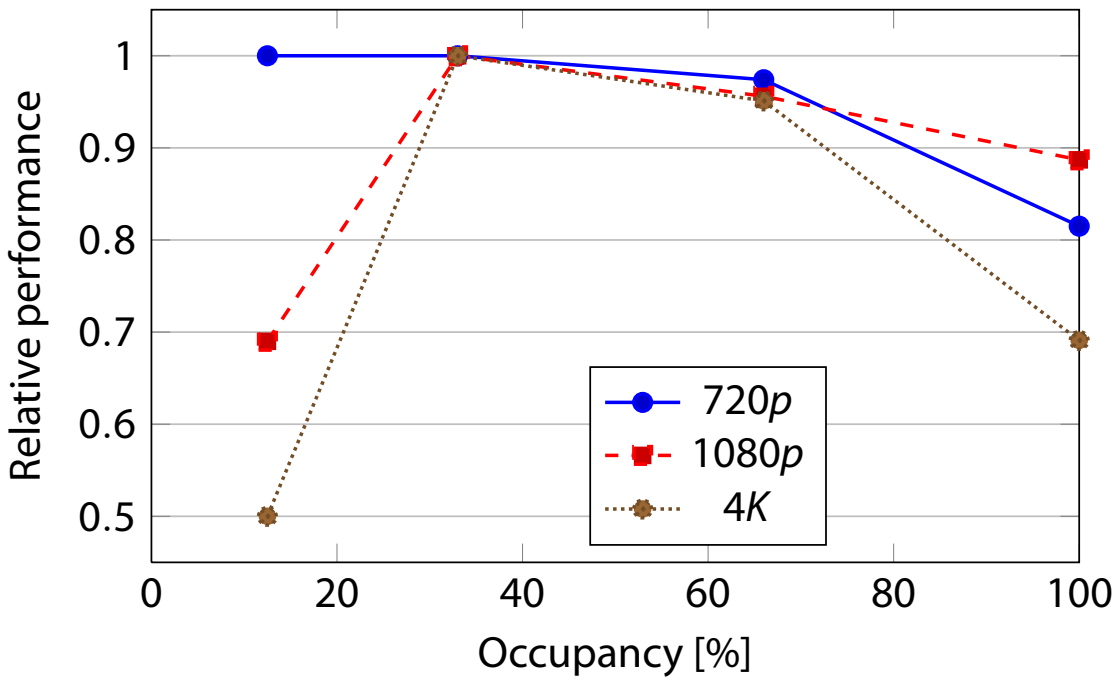
Aritmetické kódování – MQ-Coder

- Načítání dat po blocích
  - hrubé vláknění vede na nekoalescentní přístup do paměti – vlákna zpracovávají různé části bloku
  - pomůže načítat data po větších blocích[1]
  - využití **`double`** (8B) místo **`int`** (4B)
  - optimalizace počtu natahovaných **`double`** podle výkonu
  - průměrně: 33% zrychlení pro 16 natahovaných **`double`**

[1] Volkov, V.: Better Performance at Lower Occupancy. In: GPU Technology Conference 2010. (2010)

Aritmetické kódování – MQ-Coder

- Optimalizace využití karty
  - díky hrubé granularitě každé vlákno potřebuje hodně zdrojů
  - vytváří tlak na registry
  - optimalizace využití karty na výsledný výkon



Aritmetické kódování – MQ-Coder

- Shrnutí optimalizací

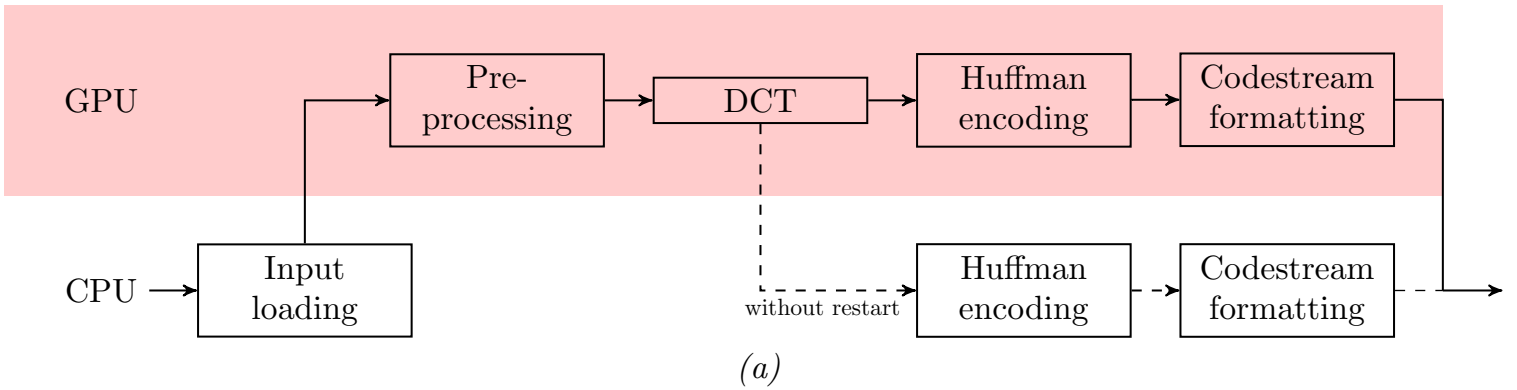
	720p		*1080p*		*4K*	
GPU Basic	38.0 ms	–	45.0 ms	–	92.9 ms	–
GPU R	16.2 ms	2.3×	18.9 ms	2.4×	48.5 ms	1.9×
GPU R+ERN	11.9 ms	3.2×	14.9 ms	3.0×	44.9 ms	2.1×
GPU R+LU	18.1 ms	2.1×	31.2 ms	1.4×	87.1 ms	1.1×
GPU R+PS	20.9 ms	1.8×	25.7 ms	1.8×	64.6 ms	1.4×
GPU R+CL	12.1 ms	3.1×	13.2 ms	3.4×	27.7 ms	3.4×
GPU R+ERN+CL	7.3 ms	5.2×	8.1 ms	5.6×	17.6 ms	5.3×

Aritmetické kódování – MQ-Coder

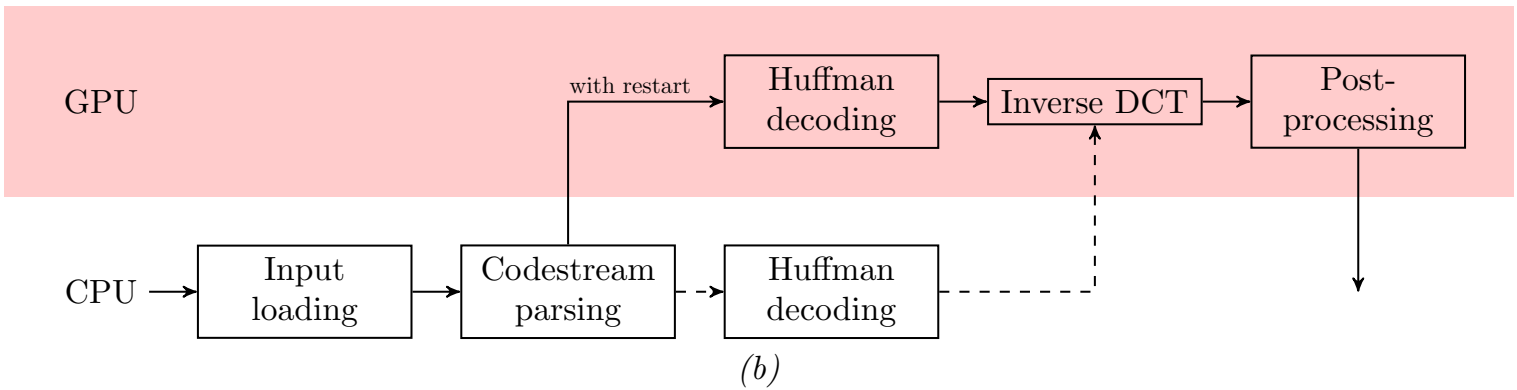
- Výsledný výkon

	720p	*1080p*	*4K*
OpenJPEG 1.4	157 ms	316 ms	1081 ms
Jasper 1.900.1	89 ms	178 ms	594 ms
Kakadu 6.4 (4 t)	41 ms	84 ms	284 ms
CUJ2K 1.1	≈25 ms	≈49 ms	≈166 ms
CUDA GPU	**7.3 ms**	**8.1 ms**	**17.6 ms**

JPEG – schéma zpracování



(a)



(b)

JPEG – Kódování entropie

- RLE + Huffman
  - 16 nul se kóduje jako speciální symbol 0xF0
  - zbytek nul se kóduje s nenulovými symboly Huffmanem
- Huffman: neadaptivní kodér
  - výběr symbolů se dá dělat plně paralelně
  - při znalosti velikosti symbolů je možné určit jejich umístění ve výsledku – prefix sum

Paralení RLE