VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2018 Bc. Filip Sedláček

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY
DEPARTMENT OF CONTROL AND INSTRUMENTATION

ZPRACOVÁNÍ OBRAZU S VELKÝMI DATOVÝMI TOKY - VYUŽITÍ CUDA/OPENCL
HIGH DATA RATE IMAGE PROCESSING USING CUDA/OPENCL

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR
Bc. Filip Sedláček

VEDOUCÍ PRÁCE
SUPERVISOR
Ing. Peter Honec, Ph.D.

BRNO 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Kybernetika, automatizace a měření**
Ústav automatizace a měřicí techniky

Student: Bc. Filip Sedláček ***ID:*** 161633
Ročník: 2 ***Akademický rok:*** 2017/18

NÁZEV TÉMATU:

Zpracování obrazu s velkými datovými toky - využití CUDA/OpenCL

POKYNY PRO VYPRACOVÁNÍ:

Cílem zadání je optimalizovat algoritmus systému UniscanDETECTOR pro detekci vad v materiálu pomocí CUDA/OpenCL knihoven pro zajištění vyšší propustnosti.

1. Seznamte se s HW a SW architekturou a nástroji pro akceleraci výpočtu pomocí CUDA aOpenCL.
2. Vytvořte interface a GUI pro práci s CUDA a OpenCL.
3. Navrhněte vhodné metody pro jednoprůchodový algoritmus detekce vad v materiálu s ohledem na plánované vysoké datové toky (2.2 GB/s na kameru).
4. Vytipujte vhodný HW architekturu, otestujte a optimalizujte jednotlivé přístupy.
5. Vyhodnoťte a porovnejte se stávajícím jednoprůchodovým algoritmem.

DOPORUČENÁ LITERATURA:

HLAVAC V., SONKA M., BOYLE R.: Image Processing, Analysis, and Machine Vision, ISBN 978-0495082521

Termín zadání: 5.2.2018 ***Termín odevzdání:*** 14.5.2018

Vedoucí práce: Ing. Peter Honec, Ph.D.
Konzultant:

doc. Ing. Václav Jirsík, CSc.
předseda oborové rady

ABSTRAKT

Hlavným cieľom tejto práce je návrh optimalizácie algoritmu detekujúceho vady v produkovanom netkanom textile. Algoritmus vyvinula spoločnosť CAMEA spol. s.r.o. Dôsledkom zmeny aktuálneho kamerového systému za výkonnejší, bude potreba aktuálny algoritmus optimalizovať a vybrať hardvér s vhodnou architektúrou, na ktorom budú výpočty vykonávané. V práci budu detailnejšie popísané programovacie techniky softvérovej architektúri CUDA a frameworku OpenCL. Pomocou týchto nástrojov navrhneme implementáciu paralelného ekvivalentu aktuálneho algoritmu, popíšeme rôzne optimalizačné metódy a navrhneme GUI k testovaniu týchto metód.

KĽÚČOVÉ SLOVÁ

GPGPU, GPU, CPU, CUDA, OpenCL, optimalizácia, paralelizácia, detekcia defektov

ABSTRACT

The main objective of this research is to propose optimization of the defect detection algorithm in the production of nonwoven textile. The algorithm was developed by CAMEA spol. s.r.o. As a consequence of upgrading the current camera system to a more powerful one, it will be necessary to optimize the current algorithm and choose the hardware with the appropriate architecture on which the calculations will be performed. This work will describe a usefull programming techniques of CUDA software architecture and OpenCL framework in details. Using these tools, we proposed to implement a parallel equivalent of the current algorithm, describe various optimization methods, and we designed a GUI to test these methods.

KEYWORDS

GPGPU, GPU, CPU, CUDA, OpenCL, optimization, parallelization, defect detection

SEDLÁČEK, Filip. *Zpracování obrazu s velkými datovými toky - využití CUDA/OpenCL*. Brno, 2018, 108 s. Diplomová práca. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky. Vedúci práce: Ing. Petr Honec, Ph.D

VYHLÁSENIE

Vyhlasujem, že som svoju diplomovoú prácu na tému „Zpracování obrazu s velkými datovými toky - využití CUDA/OpenCL“ vypracoval(a) samostatne pod vedením vedúceho diplomovej práce, využitím odbornej literatúry a ďalších informačných zdrojov, ktoré sú všetky citované v práci a uvedené v zozname literatúry na konci práce.

Ako autor(ka) uvedenej diplomovej práce ďalej vyhlasujem, že v súvislosti s vytvorením tejto diplomovej práce som neporušil(a) autorské práva tretích osôb, najmä som nezasiahol(-la) nedovoleným spôsobom do cudzích autorských práv osobnostných a/alebo majetkových a som si plne vedomý(-á) následkov porušenia ustanovenia § 11 a nasledujúcich autorského zákona Českej republiky č. 121/2000 Sb., o práve autorskom, o právach súvisiacich s právom autorským a o zmene niektorých zákonov (autorský zákon), v znení neskorších predpisov, vrátane možných trestnoprávnych dôsledkov vyplývajúcich z ustanovenia časti druhej, hlavy VI. diel 4 Trestného zákoníka Českej republiky č. 40/2009 Sb.

Brno ....

podpis autora(-ky)

POĎAKOVANIE

Rád by som poďakoval vedúcemu diplomovej práce pánovi Ing. Petr Honec, Ph.D. za odborné vedenie, konzultácie, trpezlivosť a podnetné návrhy k práci.

Brno ....

podpis autora(-ky)

OBSAH

ZOZNAM OBRÁZKOV

ZOZNAM TABULIEK

ZOZNAM VÝPISOV

ÚVOD

Prioritnou úlohou tejto práce je optimalizácia algoritmov detekujúcich vady v netkanom textile počas procesu jeho výroby. Algoritmy sú vyvíjané spoločnosťou CAMEA, spol. s.r.o., s ktorou bude činnosť na tejto práci priebežne konzultovaná. Celý systém detekcie vád, vrátane potrebného hardvéru, bol implementovaný do výroby v spoločnosti PEGAS NONWOVENS a.s., ktorá vo veľkom produkuje netkané textílie.

Systém pozostáva zo štyroch 70kHz riadkových kamier s rozlíšením 4096pix snímajúcich povrch čerstvo vyprodukovaného netkaného textilu, ktorý sa namotáva na valce s rýchlosťou až 2000 m/min. Pri spomínanej snímkovacej frekvencii kamier sa jedná o pomerne veľké toky dát, ktoré sú algoritmami spracovávané v reálnom čase. Cieľom spoločnosti CAMEA je v blízkej budúcnosti nahradiť aktuálny kamerový systém štyrmi rýchlejšími kamerami *ELIIXA+ 16k monochrome* s frekvenciou snímkovania až 200kHz a rozlíšením 11kpix. Kamery budú pravdepodobne pracovať v režime s rozlíšením 8192pix, čo znamená dátový tok 1,6 GB/sec z jedinej kamery. K realizácii tohto vylepšenia je nutné optimalizovať najfrekventovanejšiu časť algoritmu, k čomu použijeme hardvérovú/ softvérovú architektúru od spoločnosti NVIDIA Corp. - **CUDA** (Compute Unified Device Architecture) a framework pre programovanie heterogénnych zariadení **OpenCL** (Open Computing Language).

V tejto diplomovej práci sa budeme najskôr venovať detailnejšiemu teoretickému rozboru týchto dvoch nástrojov, spolu s názornou ukážkou ich praktického využitia. Budeme sa snažiť zhrnúť a vysvetliť tie najužitočnejšie programovacie techniky, ktoré bude možné využiť aj pri nadväzujúcom vývoji a optimalizácii paralelného algoritmu. Je to nevyhnutný krok pokiaľ chceme neskôr s istotou stanoviť dostatočne výkonný hardvér (v podobe GPU), ktorý nám umožní vykonať paralelné výpočty na dátach prichádzajúcich v reálnom čase z procesu výroby. Rovnako treba určiť vhodné optimalizačné metódy z hľadiska softvéru, ktoré dokážu využiť možnosti GPU na maximum.

V hlavnej časti tejto práce sa pokúsime navrhnúť paralelný ekvivalent aktuálneho algoritmu jak pomocou **CUDA** tak pomocou **OpenCL** frameworku. Tieto dva nástroje navzájom porovnáme a zhodnotíme ich využiteľnosť vo vyššie popísanom procese. Celkovým riešením problému tejto práce je všeobecne GPGPU (General-purpose computing on graphic processing units), čo je rozsiahla problematika pojednávajúce o paralelnom svete počítačov.

Na záver vytvoríme grafické užívateľské prostredie pre prácu s oboma nástrojmi (CUDA, OpenCL), kde bude možné porovnať dopady rôznych druhov optimalizácii na navrhnutý paralelný algoritmus.

1 PARALELNÉ VÝPOČTY

1.1 Vzostup paralelizmu

V posledných rokoch sa značná časť počítačového priemyslu sústreďuje na paralelné výpočty. V priebehu 30tich rokov bola jedna z hlavných metód zvyšovania výpočtového výkonu založená na zvyšovaní taktu hodín procesoru. Kvôli fundamentálnym limitáciám pri výrobe integrovaných obvodov (výkonové, tepelné obmedzenia, rapídne sa približujúci limit fyzickej veľkosti tranzistorov) boli výrobcovia v posledných rokoch nútení nájsť k tejto metóde alternatívu. Hľadanie dodatočnej výkonnosti pre osobné počítače vyvoláva veľmi dobrú otázku: *Nebolo by lepšie, miesto zlepšovania výkonu jednotlivých procesorových jednotiek, ich vložiť do počítača niekoľko?* Toto riešenie zaručilo zvyšovanie výpočtového výkonu bez potreby neustáleho zvyšovania frekvencie procesorových hodín. [3]. V roku 2005 začali vedúci výrobcovia ponúkať procesory s dvoma počítačovými jadrami. V priebehu nasledujúcich rokov tento vývoj nasledovalo vydanie troj-, štvor-, šesť- až osem-jadrových centrálnych procesorových jednotiek.

1.2 Základný princíp paralelných výpočtov

Paralelné výpočty *(Parallel Computing)* sú typom výpočtovej architektúry, v rámci ktorej niekoľko procesorov vykonáva alebo spracováva výpočet súbežne. Paralelizovaním sme schopní rozložiť komplikovaný výpočet na jednoduchšie parciálne výpočty, ktoré sú spracovávané viacerými procesorovými jednotkami v rovnakom čase. [4]. Existuje niekoľko typov paralelných výpočtov:

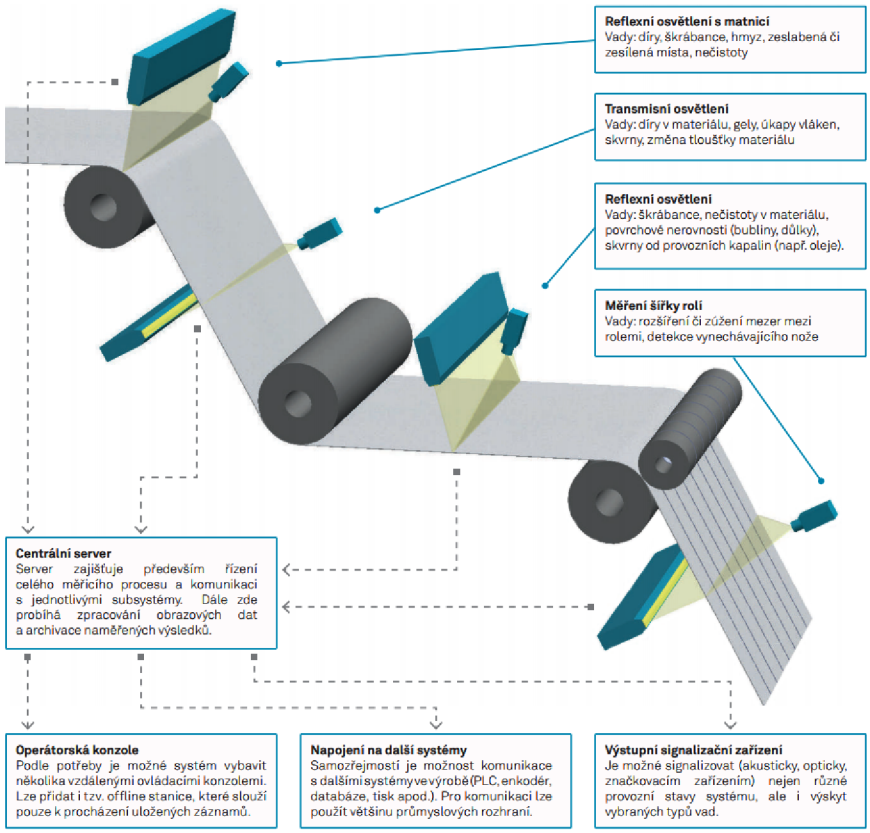
- **na úrovni bitov** - založené na zväčšovaní dĺžky slova procesoru
- **na úrovni inštrukcií** - prekrývanie strojových inštrukcií *(pipelining)*; rozdelenie spracovania jednej inštrukcie medzi rôzne časti procesoru a tým dosiahnutie spracovania viacerých inštrukcií naraz
- **na úrovni dát** - rovnaké výpočty sú prevádzané na rovnakých alebo rozdielnych skupinách dát
- **na úrovni úloh** - rozloženie jednej komplexnej úlohy na parciálne úlohy, ktoré sú vykonávané viacerými procesormi naraz

Paralelné výpočty *(Parallel Computing)* úzko súvisia so **súbežnými výpočtami** *(Concurrent Computing)*. Tieto dva pojmy sú často používané súčasne, no majú odlišný význam. Výpočet sa totiž môže diať paralelne bez použitia súbežného spracovania (paralelizmus na úrovni bitov) a tak isto súbežné spracovanie nemusí

využívať paralelizmus (multitasking na single-core CPU). Pri súbežnom spracovaní, na rozdiel od paralelného, úlohy väčšinou nie sú rozdelené medzi procesory, a ak áno, tieto úlohy môžu byť úplne rozdielneho charakteru a vyžadujú medziprocesovú komunikáciu. [5]

1.3 Rozdiel medzi GPU a CPU

Hlavným rozdielom medzi výpočtami na GPU a CPU je, že GPU je vhodné najmä pre riešenie problémov, ktoré je možné vyriešiť pomocou mnohých paralelných výpočtov. Rovnaký program je uplatňovaný na viacero dátových elementov súčasne s vysokou aritmetickou intenzitou - pomerom medzi počtom aritmetických operácii a operácií s pamäťou. Tento spôsob zaručuje nižšie nároky na riadenie výpočtového procesu resp. *(flow control)*, a pretože sa všetky dáta spracovávajú rovnakým programom s vysokou aritmetickou intenzitou, latencia prístupu do pamäte môže byť prekrytá výpočtami.



Obr. 1.1: Prehľad a popis systému.[1]

1.4 Uplatnenie paralelných výpočtov v tejto práci

Hlavným cieľom tejto práce je spraviť výskum o aktuálnych možnostiach optimalizácie algoritmov na spracovávanie veľkého objemu dát. Jedná sa predovšetkým o optimalizáciu pomocou SW/HW architektúry **Compute Unified Device Architecture - CUDA** od spoločnosti nVIDIA Corp. alebo frameworku **Open Computing Language - OpenCL**. Nadobudnuté poznatky následne implementovať do algoritmu vyvinutého spoločnosťou **©2017 CAMEA, spol. s.r.o.**, ktorý riadi systém (viď obr. 1.1) pre vizuálnu detekciu chýb vznikajúcich pri výrobe nekonečných pásov z rôznych materiálov (napr. netkané textílie, fólie, papier, valcové plechy atď.). Systém je schopný detegovať rôzne typy chýb, akými sú diery, zalisované cudzie predmety, zmeny štruktúry produkovaného materiálu atď. Scéna je osvetlená výkonným riaditeľným LED osvetlením. Povrch materiálu je snímaný vysokorýchlostnými riadkovými kamerami s pokročilým riadením objektívu, ktoré disponujú **rozlíšením 4096pix a frekvenciou snímania 70kHz**. To odpovedá dátovému toku cca 280MB/s z jednej kamery. Tieto kamery však majú byť nahradené kamerami s frekvenciou snímania 200kHz @ 16k, z čoho pri použití v režime 8192pix plynie obrovský dátový tok približne rovný 1,6GB/s. Dôsledkom navrhnutého vylepšenia je potreba optimalizovať frekventovanú časť algoritmu, ktorá detekuje miesta porušené počas výrobného procesu, čo bude predmetom tejto práce. K testovacím účelom budeme v práci používať 2 PC zostavy.

1. PC zostava (ďalej len **HWConfig GT755M**)
   - CPU: **Intel(R) Core(TM) i7-4700MQ CPU @ 2.40GHz**
   - GPU: **NVIDIA GeForce GT 755M (Kepler arch.)**
   - RAM: **8GB**

2. PC zostava (ďalej len **HWConfig GTX1050Ti**) - PC zostava poskytnutá spoločnosťou CAMEA, spol. s.r.o.
   - CPU: **Intel(R) Core(TM) i5-650 CPU @ 3.20GHz**
   - GPU: **NVIDIA GeForce GTX 1050Ti (Pascal arch.)**
   - RAM: **4GB**

2 CUDA - COMPUTE UNIFIED DEVICE ARCHITECTURE

V Novembri 2006, spoločnosť NVIDIA predstavila CUDA®, **General-purpose computing on graphic processing unit - GPGPU** programovací model, ktorý využíva paralelný výpočtový engine v grafických procesoroch s cieľom riešiť komplexné úlohy oveľa efektívnejšie ako prostredníctvom CPU. CUDA ponúka programovacie prostredie, ktoré nám v tejto práci umožní optimalizovať algoritmy na spracovanie veľkých tokov dát v jazyku C. Tento programovací jazyk bude použitý vo všetkých nasledujúcich príkladoch a riešeniach obsiahnutých v tejto práci.

2.1 Vývojové prostredie

Všetky príklady CUDA multivláknových programov v teoretickej časti tejto práce budú písané v programovacom prostredí Microsoft Visual Studio Community 2015 na platforme Windows 10 Home, ©2017 Microsoft Corporation.

NVIDIA GPU ovládače

Každému používateľovi poskytuje spoločnosť NVIDIA ovládače potrebné pre komunikáciu programu s CUDA-kompatibilnou grafickou kartou.[1] Ak bola vaša CUDA-kompatibilná grafická karta nainštalovaná správne, s veľkou pravdepodobnosťou máte potrebný software na komunikáciu už nainštalovaný v systéme. Každopádne, určite nebude na škodu uistiť sa, či tomu tak naozaj je. Na stránkach NVIDIA Drivers je možné tento nástroj stiahnuť a následne nainštalovať.

CUDA vyvojový toolkit

V tomto štádiu sme schopní úspešne spustiť skompilovanú CUDA aplikáciu na našej CUDA-kompatibilnej grafickej karte. K tomu, aby sme mohli vyvíjať vlastné aplikácie je potrebný nástroj voľne stiahnuteľný na stránkach NVIDIA CUDA Toolkit. Tento nástroj obsahuje kompilátor *nvcc* potrebný pre kompiláciu multivláknových programov na grafickej karte.

2.2 Abstrakcie modelu

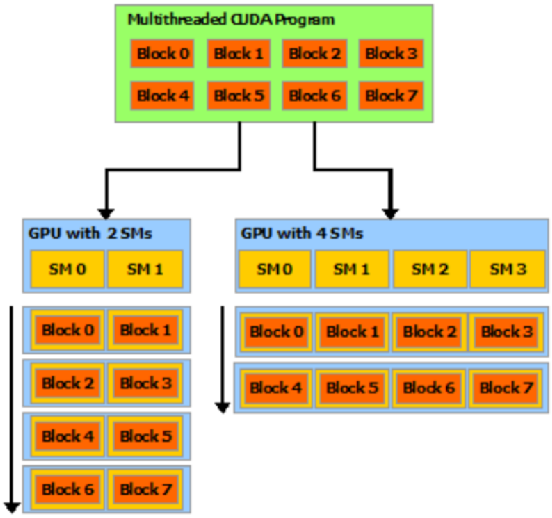
Model CUDA paralelného programovania je dizajnovaný tak, aby predišiel zdĺhavému procesu učenia sa jeho užívateľov. Preto ponúka 3 hlavné úrovne abstrakcie,

[1] Zoznam CUDA kompatibilných kariet: `https://developer.nvidia.com/cuda-gpus`

ktoré tvoria minimálnu sadu rozšírení jazyka C:

- **skupiny vlákien**
- **zdieľanú pamäť**
- **bariérová synchronizácia**

Tieto abstrakcie nútia programátora rozdeliť komplexnú úlohu na menšie, čiastkové úlohy, ktoré môžu byť riešené nezávisle a paralelne pomocou tzv. **vláknových blokov** (ďalej len **blokov**). Na každej čiastkovej úlohe potom kooperatívne pracujú všetky vlákna z tohto bloku. Každý jeden **blok** je možné priradiť voľnému mul-



Obr. 2.1: Automatické rozdelenie úloh medzi multiprocesory (SM).[2]

tiprocesoru na GPU, v akomkoľvek poradí, súbežne alebo sekvenčne, čo umožňuje vykonávanie CUDA programu na viacerých multiprocesoroch. (2.1). Súčasťou GPU sú takzvané **(Streaming Multiprocessors - SM)**. CUDA program je rozdelený do vláknových blokov s užívateľom definovaným počtom vlákien a tieto bloky sa vykonávajú nezávisle na sebe. To znamená že GPU s väčším počtom SM vykoná program rýchlejšie ako GPU s menším počtom SM.

2.3 Volanie kernelov

V paralelnom programovaní, prostredníctvom rozšírenej verzie jazyka C, sú funkcie vykonávané na GPU nazývané ako **kernely**.Tento typ funkcií sa vykonávajú paralelne N krát pomocou N CUDA vlákien. Na rozdiel od klasickej funkcie adresovanej CPU, sú kernely odlíšené deklaračným identifikátorom `__global__`. Tento identifikátor varuje kompilátor, že funkcia má byť skompilovaná pre *zariadenie* a nie pre *hostiteľa*. Za *zariadenie* je v CUDA terminológií považovaná grafická karta (GPU),

a za *hostiteľa* je považovaný procesor (CPU). Syntax volania kernelov v programe je takmer rovnaký až na časť `<<<...>>>`, ktorá obsahuje parametre ohľadom počtu blokov a počtu vlákien v jednom bloku. Každé vlákno, ktoré spracováva kernel, má priradené špecifické ID. Užívateľ k tomuto ID pristupuje cez premennú `threadIdx`, ktorá je súčasťou rozšírenia jazyka C.

Výpis 2.1: Príklad definície a volania kernelu.

```
1  __global__ void add(int n, float *x, float *y)
2  {
3    int idx = threadIdx.x;     //index vlákna
4    y[idx] = x[idx] + y[idx]; //každé vlákno vykoná jeden súčet
5  }
6
7  int main()
8  {
9    ...
10   add <<<1, N>>> (n, x, y); //volanie kernelu s N vláknami
11   ...
12 }
```

2.4 Hierarchia vlákien

Premenná `threadIdx` je 3-dimenzionálny vektor. Každé vlákno môže byť teda identifikované 1-rozmerným, 2-rozmerným alebo 3-rozmerným indexom. Tak isto aj bloky (obsahujúce N vlákien) môžu mať 1-rozmerný, 2-rozmerný alebo 3-rozmerný index - `blockIdx`. Tento spôsob indexovania bol zavedený za účelom jednoduchšieho a prehľadnejšieho zaobchádzania s vektormi, maticami alebo 3-rozmernými objektami. CUDA ponúka viac vstavaných premenných určených k indexácii, k zisteniu veľkosti blokov, indexu blokov alebo veľkosti mriežky:

- `threadIdx` - indexovanie vlákien
- `blockIdx` - indexovanie blokov v mriežke
- `blockDim` - počet vlákien v bloku
- `gridDim` - počet blokov v mriežke

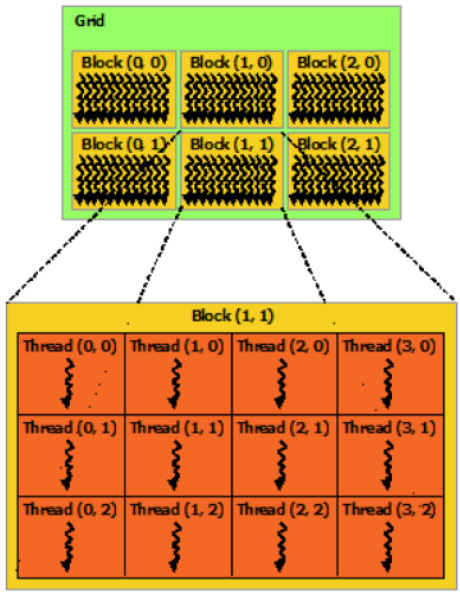
Index vlákna a ID vlákna sú dva rozdielne pojmy pokiaľ pracujeme s 2-rozmerným alebo 3-rozmerným blokom. V prípade 1-rozmerného bloku sa ID vlákna zhoduje s jeho indexom. Naopak pri 2-rozmernom bloku, ktorý obsahuje napr. NxN vlákien má vlákno na pozícii (x, y) index $(y.N+x)$. Analogicky to platí pre 3-rozmerný blok, kde má vlákno na pozícii (x, y, z) index $(z.N.N + y.N + x)$.

Každé zariadenie má limitovaný počet vlákien na jeden blok. Je to z toho dôvodu, že vlákna príslušné rovnakému bloku zdieľajú pamäť jedného jadra procesoru.

V dnešnej dobe je limit vo väčšine grafických procesorov 1024 vlákien na jeden blok. Našťastie, každý kernel môže byť spracovávaný viacerými blokmi s rovnakým počtom vlákien, takže výsledný počet vlákien podieľajúcich sa na spracovávaní jedného kernelu je rovný súčinu:

$$\text{počet všetkých vlákien} = \text{počet vlákien v bloku} \times \text{počet blokov}$$

Rovnako ako vlákna, tak aj bloky sú usporiadané do 1-rozmernej, 2-rozmernej alebo 3-rozmernej **mriežky**. K indexu bloku v mriežke pristupujeme prostredníctvom vstavanej premennej `blockIdx`, k zisteniu veľkosti bloku slúži `blockDim` a k zisteniu počtu blokov v mriežke `gridDim`. Do špeciálnych zátvoriek používaných pri



Obr. 2.2: Organizácia blokov v mriežke.[2]

volaní kernelu `<<<...>>>` píšeme ako prvý parameter **počet blokov** a druhý parameter **počet vlákien**. Tieto parametre môžu mať dátový typ `int` alebo `dim3` čo je vstavaný dátový typ v CUDA Toolkit.

V príklade 2.2 je definovaný kernel **cube**, ktorý počíta 3. mocninu každého elementu v poli. Na 2. riadku kódu sa vypočíta index - **idx** aktuálneho vlákna a použije sa v hranatých zátvorkách v 3. a 4. riadku. Vlákno s indexom **idx** uloží hodnotu pola **d_in** do premennej **f**, vypočíta 3. mocninu hodnoty uloženej v premennej **f** a uloží výsledok na miesto s indexom **idx** v poli **d_out**. Ďalej si všimnime volanie kernelu na riadku 35. Parametre v špeciálnych zátvorkách hovoria kompileru, že kernel bude spracovávaný **ARRAY_SIZE/1024** blokmi, kde každý z nich obsahuje 1024 vlákien. K ostatným častiam kódu sa vrátime neskôr.

Výpis 2.2: Kernel počítajúci 3. mocninu každého elementu vo vektore

```
__global__ void cube(float * d_out, float * d_in){
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  float f = d_in[idx];
  d_out[idx] = f * f * f;
}

int main(int argc, char ** argv){
  const int ARRAY_SIZE = 2048;
  const int ARRAY_BYTES = ARRAY_SIZE * sizeof(float);

  // genrovanie vstupného pola na CPU (hostitelovi)
  float h_in[ARRAY_SIZE];
  for (int i = 0; i < ARRAY_SIZE; i++){
    h_in[i] = float(5);
  }
  float h_out[ARRAY_SIZE];

  // deklarovanie pointerov na GPU (zariadeni)
  float * d_in;
  float * d_out;

  // alokovanie pamati na GPU
  cudaMalloc((void**)&d_in, ARRAY_BYTES);
  cudaMalloc((void**)&d_out, ARRAY_BYTES);

  // transfer vstupného pola do globálnej pamate GPU
  cudaMemcpy(d_in,h_in,ARRAY_BYTES,cudaMemcpyHostToDevice);

  //definovanie počtu blokov a vlákien, ktorými bude kernel
  //spracovaný
  dim3 numBlocks(1024);
  dim3 numThreads(ARRAY_SIZE / 1024);

  // volanie kernelu
  cube << <numBlocks, numThreads >> >(d_out, d_in);

  // kopírovanie výsledku z GPU späť do CPU
  cudaMemcpy(h_out,d_out,ARRAY_BYTES,cudaMemcpyDeviceToHost);

  // uvolnenie alokovanej pamati na GPU
```

```
41    cudaFree(d_in);
42    cudaFree(d_out);
43
44    return 0;
45  }
```

Premenná `Threads` typu dim3 je uvedená na riadku 27. Definuje počet vlákien v jednom bloku - v našom prípade 16x16=256 vlákien/blok. Rovnako premenná `Blocks` určuje vo volaní kernelu (riadok 30) počet blokov, s ktorými má byť kernel spracovávaný. Kód kernelu zároveň musí byť napísaný tak, aby nezáviselo na poradí v akom budú bloky pridelené k spracovaniu určitých častí kernelu. Inými slovami, jednotlivé časti kernelu nemusia byť vykonávané sekvenčne, ale môžu byť spracovávané akýmkoľvek blokom v hocijakom čase. Splnenie tejto podmienky umožňuje nadriadenému plánovaciemu systému nezávisle rozdeliť bloky medzi viaceré jadrá. Úlohou programátora je teda napísať program tak, aby bolo vyťaženie jadier čo najefektívnejšie.

2.5 Synchronizačné bariéry

Vlákna v jednom bloku môžu medzi sebou zdielať (čítať alebo zapisovať) dáta prostredníctvom **zdielanej pamäte**. Prístup do tejto pamäti musí byť synchronizovaný. Aj napriek tomu, že vlákna pracujú na elementárnych úlohách paralelne, nie všetky vlákna budú schopné vykonať danú úlohu v rovnakom čase. Ako príklad uvedieme situáciu, kedy vlákna A a B načítajú dátový element z globálnej pamäti a uložia ho do zdielanej pamäti. Povedzme, že vlákno A chce čítať dátový element vlákna B. Tu môže nastať problém, pretože vlákno B totiž ešte nemuselo dokončiť zápis do zdieľanej pamäte a vlákno A chce už čítať tento element. To môže viesť k neočakávanému správaniu programu a vzniká tzv. - **súbeh** (angl. *race condition*).

K zaisteniu korektnej kooperácie vlákien ich musíme synchronizovať. CUDA Toolkit poskytuje užívateľom synchronizačnú bariéru `__syncthreads()`. Takže vlákna v rámci jedného bloku môžu pokračovať v akcii iba v prípade, že sa všetky dopracovali k tejto bariére. Súbehu sa vyhneme bariérou `__syncthreads()` hneď po zápise vlákien do zdielanej pamäti a tesne pred čítaním vlákien zo zdieľanej pamäti.

Je dôležité uvedomiť si, že v programe, ktorý obsahuje divergentné časti (napr. podmienky if, switch) môže volanie `__syncthreads()` spôsobiť deadlock - 'nekonečné' čakanie na synchronizáciu vlákien. Preto sa odporúča volanie bariéry v miestach kódu kde nedochádza k divergencii.

2.6 Hierarchia pamätí

CUDA vlákna môžu počas behu pristupovať k dátam v rôznych pamäťových oblastiach ako je znázornené na obrázku (2.5). Pri dizajnovaní algoritmu v kerneli je treba dbať na to, že každá pamäťová oblasť prináša svoje výhody, ale aj nevýhody. Každá z nich má napr. inú latenciu prístupu a veľkosť, takže pri optimalizácii algoritmu musíme väčšinou pristúpiť na kompromis. Jednotlivé oblasti si v tejto kapitole podrobnejšie popíšeme.

- **registrová pamäť**
- **lokálna pamäť**
- **zdielaná pamäť**
- **globálna pamäť**
- **konštantná pamäť**

2.6.1 Registrová pamäť

Skalárne premenné deklarované v tele kernelu, ktoré neobsahujú žiaden atribút v ich deklarácii sú štandardne uložené v registrovej pamäti. Latencia prístupu je síce veľmi malá, no počet registrov dostupných pre jeden blok je limitovaný. Polia, ktoré sú deklarované v tele kernelu sú uložené v registrovej pamäti iba v prípade, že sú indexované konštantami (index musí byť determinovaný už počas kompilácie). Každé vlákno má pridelenú vlastnú registrovú pamäť, ktorá je dostupná iba počas existencie daného vlákna. Čítanie a zapisovanie do tejto pamäti nemusí byť synchronizované.

2.6.2 Lokálna pamäť

Do tejto pamäti sa ukladajú:

- Akékoľvek premenné, ktoré sa už nezmestia do registrovej pamäti vyčlenenej pre celý kernel.
- Polia, do ktorých sa pristupuje premennými (nie konštantami).
- Veľké štruktúry, ktoré by zabrali v registrovej pamäti veľa miesta.

Lokálna pamäť má originálne rovnako pomalý prístup ako globálna pamäť (100x pomalšia ako registrová). V grafických kartách s výpočtovým výkonom vyšším ako 2.0, je vlastne lokálna pamäť medzipamäťou *cached*, čo znižuje jej latenciu. Rovnako, ako pri registrovej pamäti, tak aj lokálna pamäť je pridelená každému vláknu zvlášť a má rovnakú životnosť ako vlákno. Čítanie a zapisovanie do tejto pamäti nemusí byť synchronizované.

2.6.3 Globálna pamäť

Deklarácie premenných alebo polí, ktoré chceme aby boli uložené v zdieľanej pamäti musí obsahovať atribút `__device__`. Prístup do tejto pamäti je veľmi pomalý (100x pomalší ako do registrovej). Výhodou je, že táto pamäť môže mať kapacitu až 12GB (závisí to na type GPU). Na rozdiel od registrovej, lokálnej a zdieľanej pamäti sa na manipuláciu používajú funkcie:

- `cudaMalloc()` - Alokácia pamäti na GPU. (viď. 23., 24. riadok 2.2)
- `cudaMemcpy()` - Čítanie/ zápis do globálnej pamäti. (viď. 27., 38. riadok 2.2)
- `cudaFree()` - Uvoľnenie alokovanej pamäti na GPU. (viď. 41., 42. riadok 2.2)

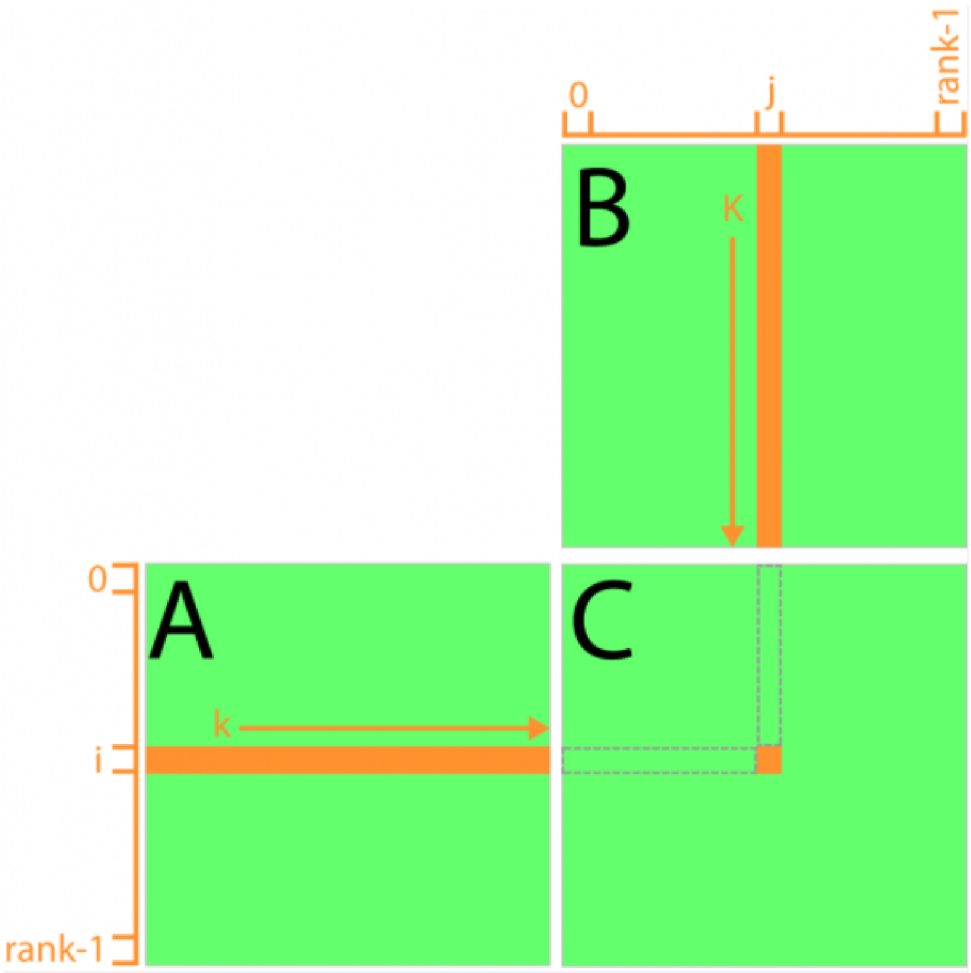
Globálna pamäť je dostupná v prebehu celej aplikácie všetkými vláknami vo všetkých kerneloch. Je treba zobrať na vedomie, že vlákna z rôznych blokov nie je možné synchronizovať pomocou `__syncthreads()` a teda môže dôjsť k situácii, že viac vlákien chce pristúpiť k rovnakej adrese v pamäti. Možným riešením je rozdelenie úlohy do viacerých kernelov a tieto kernely synchronizovať na úrovni CPU. Pointer do globálnej pamäte sa často predáva ako parameter kernelu (viď. 1. riadok 2.3). Nasleduje príklad kernelu, ktorého funkciou je násobenie matíc s využitím globálnej pamäti.

Výpis 2.3: Násobenie matíc s využitím globálnej pamäti.

```
1  __global__
2  void MatrixMult_Global( float* C, const float* A,
3                          const float* B, unsigned int rank )
4  {
5      // Výpočet indexu riadku
6      unsigned int i = ( blockDim.y * blockIdx.y ) + threadIdx.y;
7      // Výpočet indexu stlpcu
8      unsigned int j = ( blockDim.x * blockIdx.x ) + threadIdx.x;
9
10     unsigned int index = ( i * rank ) + j;
11     float sum = 0.0f;
12     for ( unsigned int k = 0; k < rank; ++k )
13     {
14         sum += A[i * rank + k] * B[k * rank + j];
15     }
16     C[index] = sum;
17 }
```

Parametre kernelu **A**, **B** a **C** sú pointre do globálnej pamäti. Na 6. a 8. riadku sa zisťujú súradnice vlákna spracovávajúceho kernel. Do premennej index na 10. riadku sa zapíše poradie prvku v matici (pri riadkovo-dominantnom indexovaní). Na

12. riadku prechádzame cez všetky elementy riadku **i** matice **A** a všetky elementy stĺpcu **j** matice **B**. Po ukončení cyklu **for** získame skalárny súčin riadku **i** a **j**, a túto hodnotu uložíme na príslušné miesto vo výstupnej matici **C** (obr. 2.3).



Obr. 2.3: Násobenie matíc s využitím globálnej pamäti. [6]

Po analýze algoritmu 2.3 zistíme, že k výpočtu **i-teho riadku** výslednej matice **C** musíme k i-temu riadku matice A pristúpiť práve **rank-krát**. Analogicky platí, že k výpočtu **j-teho stĺpcu** výslednej matice **C** musíme k j-tému stĺpcu matice **B** pristúpiť práve **rank-krát**. Povedzme, že matica **C** má rozmery **N x M**. V tom prípade je počas výpočtu každý element matice **A** využitý **M-krát** a každý element matice **B** je využitý **N-krát**. Každé použitie jedného elementu matice **A** alebo **B** je sprevádzané jedným čítaním z globálnej pamäti, čo vedie k nízkej efektivite a rýchlosti programu.

2.6.4 Zdielaná pamäť

Deklarácie premenných alebo polí, ktoré chceme, aby boli uložené v zdieľanej pamäti, musia obsahovať atribút `__shared__`. Prístup do tejto pamäti je veľmi rýchly (100x rýchlejší ako do globálnej). Nevýhodou je, že každý SM má limitovaný adresový priestor tejto pamäti. Zdieľaná pamäť musí byť deklarovaná v rámci kernelu, no jej životnosť sa vzťahuje k životnosti bloku vlákien. Každé vlákno z rovnakého bloku má prístup do tejto pamäti, takže tieto prístupy musia byť synchronizované príkazom `__syncthreads()`, aby sa predišlo stavu, kedy viac vlákien bude zapisovať/čítať z jednej adresy. K vôli jej vysokej rýchlosti je veľmi efektívne kopírovať dáta z globálnej do zdielanej pamäti a tým pádom zredukovať počet prístupov kernelu do globálnej pamäti.

Ako príklad uvediem kernel násobiaci matice **A** a **B** s využitím zdieľanej pamäti 2.4. Algoritmus spočíva v rozdelení matíc **A** a **B** na submatice (dlaždice) o veľkosti **BLOCK_SIZE**, ktoré sú uložené v zdieľanej pamäti (obr. 2.4). Každá submatica má veľkosť odpovedajúcu veľkosti jedného bloku. Práve preto môžeme každé jedno vlákno z bloku použiť na kopírovanie jedného elementu z matice **A** do submatice **sA** resp. z matice **B** do submatice **sB**. Touto technikou sme schopní zredukovať počet prístupov do globálnej pamäti na **rank/BLOCK_SIZE**, kde **BLOCK_SIZE** je veľkosť bloku aj submatice a **rank** je hodnosť matice.

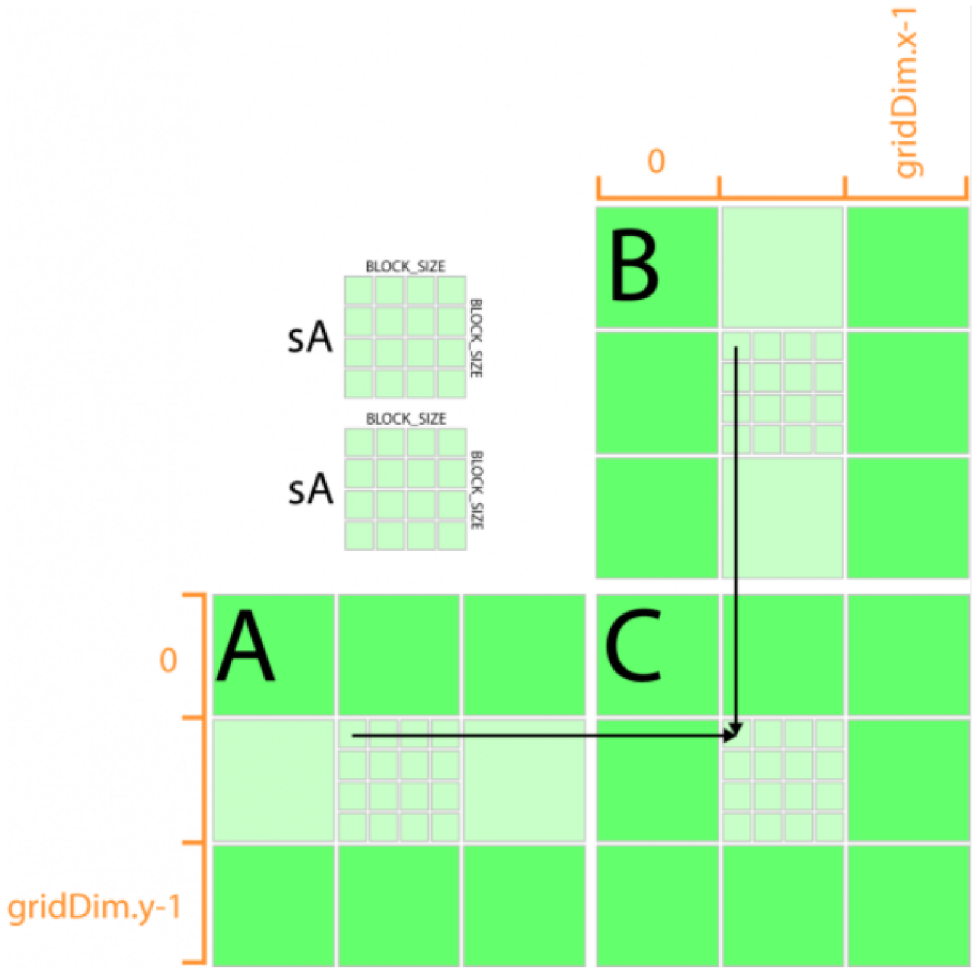
Výpis 2.4: Násobenie matíc s využitím zdielanej pamäti.

```
1  #define BLOCK_SIZE 32 //velkosť bloku
2
3  __global__ void MatrixMultKernel_Shared( float* C, float* A,
4                              float* B, int rank )
5  {
6      int tx = threadIdx.x; int ty = threadIdx.y;
7      int bx = blockIdx.x;  int by = blockIdx.y;
8
9      // Alokovanie zdielanej pamati pre submatice sA a sB
10     __shared__ float sA[BLOCK_SIZE][BLOCK_SIZE];
11     __shared__ float sB[BLOCK_SIZE][BLOCK_SIZE];
12
13     int j = ( blockDim.x * bx ) + tx; //index stlpcu
14     int i = ( blockDim.y * by ) + ty; //index riadku
15
16     int index = ( i * rank ) + j;
17     float sum = 0.0f;
18
19   // Prechádzanie jednotlivých submatíc a počítanie prvkov
```

```
20    // matice C
21      for(int phase=0; phase<rank/BLOCK_SIZE; ++phase)
22      {
23          // Kopírovanie dát z matíc A a B (glob. pamať)
24          // do submatic sA a sB (zdielaná pamať)
25          sA[ty][tx] = A[i * rank + (phase * BLOCK_SIZE + tx)];
26          sB[ty][tx] = B[(phase * BLOCK_SIZE + ty) * rank + j];
27          __syncthreads();
28
29          for( unsigned int k = 0; k < BLOCK_SIZE; ++k )
30          {
31              sum += sA[ty][k] * sB[k][tx];
32          }
33          __syncthreads();
34      }
35      C[index] = sum;
36  }
```

V nasledujúcom texte si kernel popíšeme:

- **(riadok 6. - 7.)** –> prebieha ukladanie ID vlákien a blokov v oboch rozmeroch x a y.
- **(riadok 10. - 11.)** –> prebieha alokácia zdieľanej pamäti pre submatice o veľkosti **BLOCK_SIZE** (veľkosť jedného bloku vlákien).
- **(riadok 13. - 14.)** –> sú načítané aktuálne indexy stĺpcu **j** a riadku **i**.
- **(riadok 16.)** –> sa počíta aktuálny index elementu výstupnej matice **C**, do ktorého sa má uložiť výsledok skalárneho súčinu **i-teho** riadka matice **A** a **j-teho** stĺpcu matice **B**.
- **(riadok 21. - 34.)** –> prechádzame cez všetky submatice matíc **A** a **B**. V každej iterácii naplníme submatice **sA** a **sB** dátami z matíc **A** a **B** uložených v globálnej pamäti. Všetky vlákna (v ramci jedného bloku), ktoré kopírovali dáta z globálnej pamäti do submatíc, synchronizujeme funkciou **___syncthreads()**.
- **(riadok 29.- 32.)** –> prechádzame všetky elementy submatíc a počítame skalárny súčin každého riadku submatice **sA** s každým stĺpcom submatice **sB**. Po ukončení tohoto cyklu všetky vlákna (v rámci jedného bloku) opäť synchronizujeme.
- **(riadok 35.)** –> uložíme vypočítané skalárne súčiny uložené v premennej **sum** na správny index matice **C**.



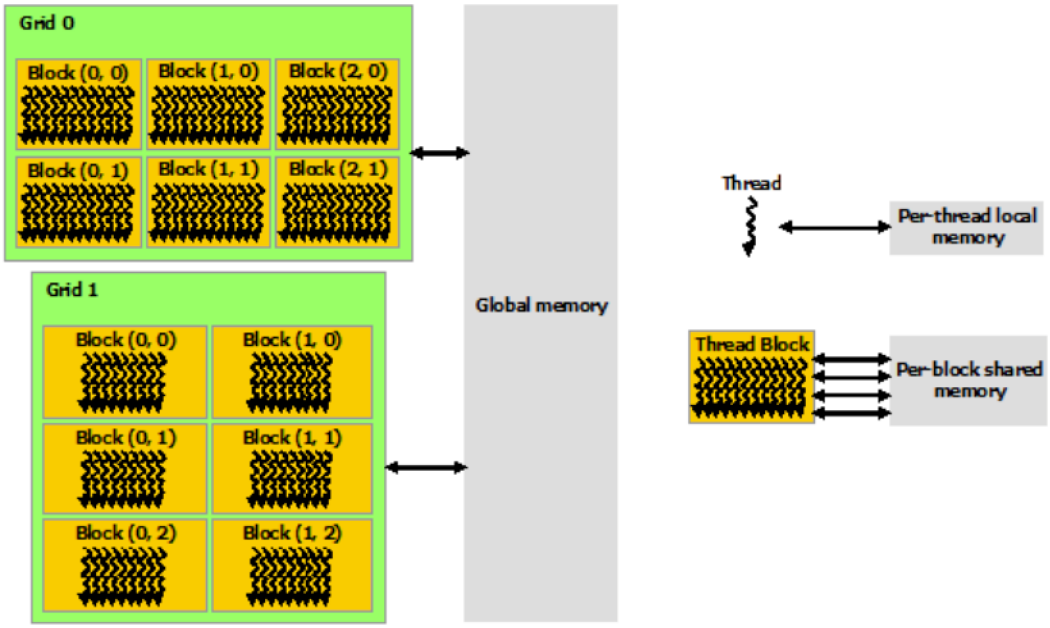
Obr. 2.4: Násobenie matíc s využitím zdielanej pamäti. [6]

2.6.5 Konštantná pamäť

Deklarácia premenných alebo polí musí obsahovať atribút `__constant__`. Rovnako ako globálna pamäť je sprístupnená všetkým vláknam zo všetkých kernelov (ak nie je explicitne nastavené ináč) a zdieľa rovnaký pamäťový priestor. Je rýchlejšia, lebo slúži ako medzipamäť (cache) a je určená pre dáta, ktoré nechceme, aby boli počas behu programu menené. To znamená, že vlákna z nej môžu iba čítať.

2.7 Heterogénne programovanie

Heterogénny program využíva výpočtový výkon GPU aj CPU. V prípade, že programátor napíše program v čistom jazyku C, tento kód môže spustiť len na CPU. Takže vzniká otázka, ako napíšeme kód, ktorý je spustiteľný aj na GPU? Na riešenie tohto problému slúži model CUDA. Umožňuje programovať oba procesory (GPU



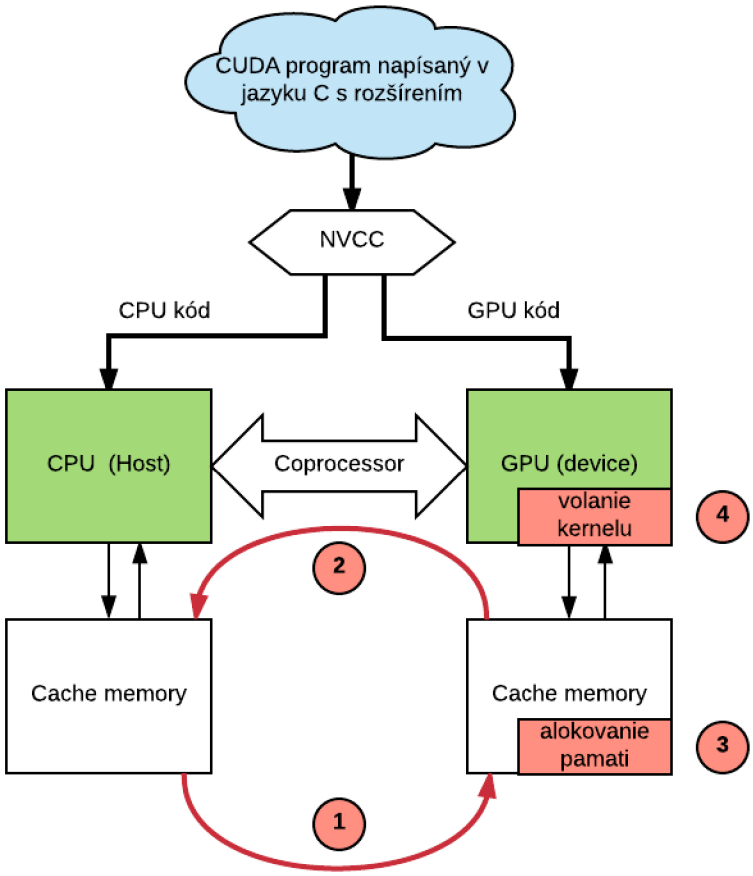
Obr. 2.5: Hierarchia pamäťových oblastí. [2]

a CPU) iba s jedným programom. CUDA program obsahuje časti písané čistým jazykom C určené pre spracovanie v CPU (hostiteľovi) a časti kódu (využívajúce CUDA rozšírenú verziu jazyka C) určené pre spracovávanie na GPU (zariadení). CUDA kompilátor `nvcc` jednoducho rozdelí kód medzi GPU a CPU a vygeneruje kód pre každý procesor zvlášť. CUDA zároveň predpokladá, že:

1. *zariadenie* je koprocesorom *hostiteľa*.
2. *hostiteľ* aj *zariadenie* majú vlastné, oddelené pamäte, kde ukladajú dáta (napr. DRAM)

CPU je nadradené nad GPU, vykonáva hlavný program a posiela riadiace inštrukcie do GPU. Je teda zodpovedné za (2.6):

1. transfer dát z pamäti CPU do pamäti GPU.
2. transfer dát z pamäti GPU späť do pamäti CPU.
   (a) CUDA Toolkit poskytuje funkciu `cudaMemcpy()`.
3. alokovanie pamäti na GPU.
   (a) CUDA Toolkit poskytuje funkciu `cudaMalloc()`.
4. volanie kernelov na GPU, ktoré prevádzajú paralelné výpočty.



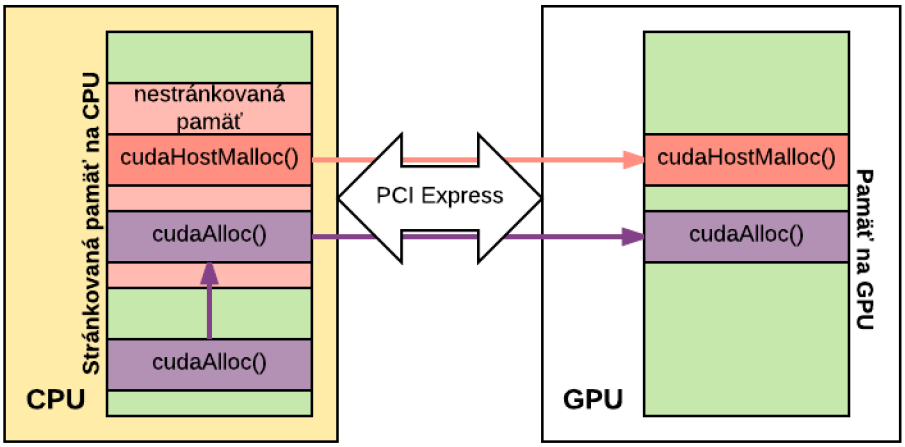
Obr. 2.6: Princíp heterogénneho programovania

2.8 Pamäť s blokovaným stránkovaním (page-locked memory)

V predchádzajúcich príkladoch sme počítali s alokáciou pamäti na GPU pomocou funkcie `cudaMalloc()` a jej uvoľňovaním s `cudaFree()`. Pamäť na CPU bola alokovaná známou funkciou `malloc()` a uvoľnená pomocou `free()`. CUDA Runtime však ponúka vlastný mechanizmus alokácie pamäti na CPU: `cudaHostAlloc()`. Rozdiel medzi `malloc()` a `cudaHostAlloc()` je v tom, že `cudaHostAlloc()` alokuje **pamäť s blokovaným stránkovaním** (angl. *page-locked* alebo *pinned* memory). Ak poznáme fyzickú adresu nami alokovaného zásobníka, tak GPU môže využiť DMA ku kopírovaniu dát z pamäti hostiteľa alebo do pamäti hostiteľa bez zásahu CPU. Inými slovami, pamäť s blokovaným stránkovaním (ďalej len *nestránkovaná pamäť*) je umiestnená v RAM, takže zariadenie (GPU) s ňou môže manipulovať bez intervencie CPU tzn. asynchrónne. Keby sme chceli kopírovať stránkovanú pamäť, CUDA driver aj v tomto prípade použije DMA, no ku kopírovaniu dát dôjde v podstate až

2-krát (viď obr. 2.7):

1. Zo **stránkovanej pamäti** na CPU do **nestránkovanej pamäti**
2. Z nestránkovanej pamäti prostredníctvom DMA do pamäti GPU.



Obr. 2.7: Rozdiel medzi kopírovaním stránkovanej (fialová šípka) a nestránkovanej pamäti (červená šípka)

Tento prístup zapríčiní spomalenie kopírovania. Rýchlosť bude závislá od toho, ktorá zo zberníc FSB (Front Side Bus) alebo PCIe (PCI Express) je pomalšia. V systémoch s veľkú disparitu rýchlostí medzi zbernicami môže používanie nestránkovanej pamäti zvýšiť výkon až dvojnásobne. V dnešnej dobe sú rýchlosti zberníc PCI Express a FSB skoro identické, no aj napriek tomu môžeme týmto spôsobom docieliť patrné zvýšenie výkonu. Ako príklad uvedieme alokáciu pamäti na GPU použitím oboch funkcii:

Výpis 2.5: Alokácia nestránkovej pamäti na CPU využitím `cudaHostAlloc()`.

```
1  int main(){
2    int size = 128;
3    int full_size = size*10;
4    int *host_a, *dev_a;
5
6    //alokácia nestránkovanej pamati (page-locked)
7    cudaHostAlloc((void**)&host_a, full_size * size(*host_a),
8          cudaHostAllocDefault);
9    //alokácia stránkovanej pamati (pagable)
10   cudaMalloc((void**)&dev_a, full_size * sizeof(int));
```

```
11
12    //kopírovanie pamati z CPU (host_a) do GPU (dev_a)
13    cudaMemcpy( dev_a, host_a, full_size * sizeof(int),
14        cudaMemcpyHostToDevice);
15
16    //volanie nejakého kernelu
17    kernel<<<full_size/16,16,0>>>( dev_a );
18
19    //kopírovanie pamati z GPU (dev_a) do CPU (host_a)
20    cudaMemcpy( host_a, dev_a, full_size * sizeof(int),
21        cudaMemcpyDeviceToHost);
22
23    //uvolnenie nestránkovanej pamati
24    cudaFreeHost(host_a);
25    //uvolnenie stránkovanej pamati
26    cudaFree(dev_a);
27 }
```

Používanie funkcie `malloc()` je skoro totožné s funkciou `cudaHostAlloc()` až na jednu výnimku. Posledný argument obsahuje **flagy**, ktorými je možné nastaviť alokáciu rôznych druhov nestránkovanej pamäti. K nastaveniu štandardnej nestránkovanej pamäti sa používa flag `cudaHostAllocDefault`. Ďalšie možné hodnoty flagov budú použité a vysvetlené v nasledujúcich kapitolách tejto práce [2]. Alokovanú pamäť je potrebné vždy na konci programu uvolniť. Na to slúži obdoba funkcie `cudaFree()` teda - `cudaFreeHost()`. Na záver tejto podkapitoly je nutné poznamenať, že využívanie pamätí bez stránkovania sa nepoužíva výhradne na zlepšenie výkonu. Slúži skôr ako podmienka pre uskutočnenie **streamov**, ktorým sa budem venovať v nasledujúcej podkapitole.

2.9 Streams

Používaním **CUDA sreamov** môžeme citeľne zrýchliť našu aplikáciu. CUDA streamy reprezentujú radu GPU operácií, ktoré sa vykonajú v špecifickom poradí. Jedná sa najmä o operácie typu:

- volanie kernelov
- operácie s pamäťou
- manipulácia s udalosťami (eventami)

[2] Referenčný manuál je dostupný na stránke `http://docs.nvidia.com/cuda/pdf/CUDA_Toolkit_Reference_Manual.pdf`

Poradie v akom sa operácie pridajú do streamov určuje, v akom poradí budú tieto operácie vykonané. Streamy by sme si teda mohli predstaviť ako *tasky* na GPU, ktoré sa môžu (aj nemusia) vykonávať súčasne. Je to závislé od implementácie. Ako príklad si uvedieme funkciu **main** z predchádzajúcej kapitoly (2.5) s tým, že riadky 12 - 21 nahradíme s nasledujúcou časťou kódu:

Výpis 2.6: Asynchrónne, obojsmerné kopírovanie pamäti medzi GPU a CPU s využitím `cudaMemcpyAsync()`.

```
1  //asynchronné kopírovanie pamati z CPU (host_a) do GPU (dev_a)
2  cudaMemcpyAsync( dev_a, host_a, full_size * sizeof(int),
3          cudaMemcpyHostToDevice, stream1);
4
5  //......Volanie nejakého kernelu......
6  kernel<<<full_size/16,16,0,stream1>>>( dev_a );
7
8  //asynchronné kopírovanie pamati z GPU (dev_a) do CPU (host_a)
9  cudaMemcpyAsync( dev_a, host_a, full_size * sizeof(int),
10         cudaMemcpyDeviceToHost, stream1);
```

Na rozdiel od prípadu (2.5), využijeme na kopírovanie pamäti funkciu `cudaMemcpyAsync()`. Rozdiel medzi `cudaMemcpyAsync()` a `cudaMemcpy()` je v tom, že `cudaMemcpy()` sa vykoná rovnako ako štandardná C funkcia `memcpy()` - teda synchrónne. Naopak, `cudaMemcpyAsync()` je vykonávaná asynchrónne, čo znamená, že funkcia po vrátení hodnoty ešte nemusí byť dokončená, či vôbec začatá. Volaním `cudaMemcpyAsync()` uložíme požiadavku na kopírovanie dát do tzv. **streamu**, čo je posledný argument vo funkcii `cudaMemcpyAsync()`. Programátor má istotu, že kopírovanie prebehne skôr, ako sa spustí ďalšia úloha vložená do toho istého streamu, napr. volanie kernelu na riadku 6. Ako si môžeme všimnúť, aj volanie kernelu obsahuje v špeciálnych zátvorkách ďalší argument. Je to názov streamu, do ktorého je vložená požiadavka na volanie tohto kernelu. `Plánovať asynchrónne kopírovanie pamäti, môžeme len v prípade, že na CPU bola alokovaná nestránkovaná pamäť (page-locked memory).`

V nasledujúcej časti kódu si ukážeme ako deklarovať streamy, tak aby sme ich mohli používať vo funkcii `main()` z príkladu (2.5).

Výpis 2.7: Inicializácia streamov.

```
1  \\inicializácia stremu.
2  cudaStream_t stream1;
3  cudaStreamCreate( &stream1 );
```

Aby sme sa na konci funkcie `main()` ubezpečili, že všetky výpočty na kerneloch a kopírovania pamäti medzi GPU a CPU sú ukončené, musíme stream synchronizovať a na záver tento stream odstrániť:

Výpis 2.8: Synchronizácia a odstránenie streamu.

```
1  //synchronizácia streamu
2  cudaStreamSynchronize( stream1 );
3  //odstránenie streamu
4  cudaStreamDestroy( stream1 );
```

2.10 Používane viacerých CUDA streamov

Tak ako sme už spomenuli vyššie, efektívne používanie viacerých CUDA streamov môže viesť k výraznému zlepšeniu výkonu aplikácie. Ale ako dosiahneme **efektívne** používanie streamov? Na úvod je potrebné si uvedomiť dôležitý fakt. **Poradie v akom zaradíme operácie do streamov (kopírovanie pamäti, časovanie, volanie kernelov...atď), ovplyvní spôsob akým CUDA driver naplánuje ich vykonávanie.** Pozrime sa na nasledujúcu časť kódu.

Výpis 2.9: Využívanie viacerých CUDA streamov súčasne

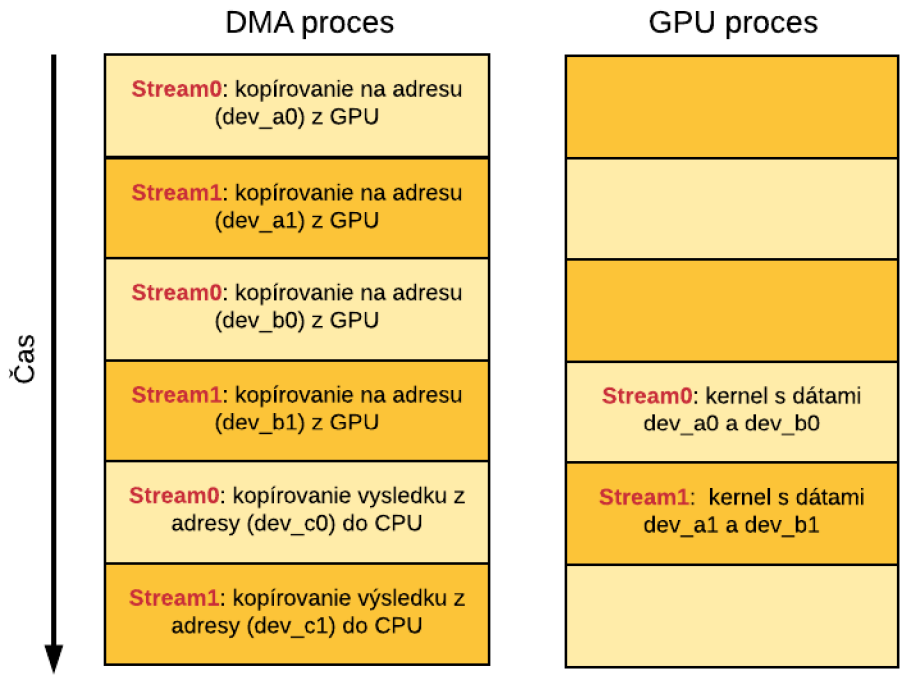
```
1  int main(){
2    int N = 128;
3    int full_size = N*10;
4    int *host_a, *host_b, *host_c;
5    //buffer pre stream0 na GPU
6    int *dev_a0, *dev_b0, *dev_c0;
7    //buffer pre stream1 na GPU
8    int *dev_a1, *dev_b1, *dev_c1;
9
10   //inicializácia stream0 a stream1
11   //...(vid predchádzajúce kapitoly)
12   //alokácia pamati pre hostitela - cudaMalloc()
13   //...(vid predchádzajúce kapitoly)
14   //alokácia pamati pre zariadenie - cudaHostAlloc()
15   //...(vid predchádzajúce kapitoly)
16
17   for(int i=0; i<full_size; i+=N*2){
18     //kopírovanie časti vstupného bufferu cez stream0
19     cudaMemcpyAsync( dev_a0, host_a+i, N * sizeof(int),
```

```
20              cudaMemcpyHostToDevice, stream0);
21      //kopírovanie časti vstupného bufferu cez stream1
22      cudaMemcpyAsync( dev_a1, host_a+i+N, N * sizeof(int),
23              cudaMemcpyHostToDevice, stream1);
24      //kopírovanie časti vstupného bufferu cez stream0
25      cudaMemcpyAsync( dev_b0, host_b+i, N * sizeof(int),
26              cudaMemcpyHostToDevice, stream0);
27      //kopírovanie časti vstupného bufferu cez stream1
28      cudaMemcpyAsync( dev_b1, host_b+i+N, N * sizeof(int),
29              cudaMemcpyHostToDevice, stream1);
30
31      //operácie s buffermi dev_a a dev_b
32      kernel<<<N/16,16,0,stream0>>>(dev_a0, dev_b0, dev_c0);
33      kernel<<<N/16,16,0,stream1>>>(dev_a1, dev_b1, dev_c1);
34
35      //kopírovanie časti výsledného bufferu cez stream0
36      cudaMemcpyAsync( host_c+i, dev_c0, N * sizeof(int),
37              cudaMemcpyDeviceToHost, stream0);
38      //kopírovanie časti výsledného bufferu cez stream1
39      cudaMemcpyAsync( host_c+i+N, dev_c1, N * sizeof(int),
40              cudaMemcpyDeviceToHost, stream1);
41  }
42  //synchronizácia streamov, odstránenie streamov a
43  //uvolnenie pamati.
44 }
```

Inštrukcie sme rozdelili medzi **stream0** a **stream1**, tak aby napr. začiatok vykonania kernelu cez stream1 nebol závislý na ukončení kopírovania `*dev_a0, *dev_b0, *dev_c0` do pamäti GPU. Tak isto sme týmto riešením eliminovali závislosť vykonania kernelu cez stream0 na ukončení kopírovania `*dev_a1, *dev_b1, *dev_c1` do pamäti GPU. Počas kopírovania dát potrebných pre kernel cez stream0, môže prebiehať spracovávanie kernelu s už pripravenými dátami cez stream1. Poradie streamov v čase je znázornené na obrázku 2.8.

2.10.1 Nekopírovaná pamäť s blokovaným stránkovaním (zero-copy memory)

V predchádzajúcich kapitolách sme porovnávali pamäť so stránkovaním (angl. *pagable memory*) a s blokovaným stránkovaním. Buffer alokovaný v nestránkovanej hostiteľskej pamäti bude vždy pripravený na DMA prenos do GPU. Funkcia `cudaHostAlloc()`



Obr. 2.8: Vykonávanie streamov v plynúcom čase z príkladu 2.9.

vlastne eliminuje spoluprácu CPU počas kopírovania pamäti do GPU. V príklade (2.5) sme v tejto funkcii použili posledný parameter flag `cudaHostAllocDefault`, ktorý hovorí o štandardnom nastavení pamäti s blokovaným stránkovaním. Avšak tejto funkcii môžme predávať aj iné flagy, ktoré ponúkajú akúsi nadstavbu:

- `cudaHostAllocMapped` - Zaistí priamy prístup CUDA kernelu do pamäti na CPU alokovanej pomocou `cudaHostAlloc()`. Táto pamäť nevyžaduje kopírovanie medzi CPU a GPU. Hovoríme teda o **nekopírovanej hostiteľskej pamäti** (angl. *zero-copy host memory*).
- `cudaHostAllocWriteCombined` - Nezmení funkcionalitu, ale v niektorých prípadoch môže značne zvýšiť výpočtový výkon pre buffre čítané iba GPU.
- `cudaHostAllocPortable` - Alokovaná nestránková pamäť bude zdieľaná medzi vláknami CPU. Budeme sa jej venovať v ďalších kapitolách.

Alokácia nekopírovanej pamäti na CPU bude teda vyzerať takto:

Výpis 2.10: Alokácia nekopírovanej pamäti na CPU

```
...
float *a, *dev_a;
int size = 258;
```

```
//alokácia nekopírovanej pamati.
cudaHostAlloc((void**)&a, size * sizeof(float),
        cudaHostAllocWriteCombined |
        cudaHostAllocMapped);
...
```

Týmto sme zaistili, že buffer `a` bude dostupný priamo z CUDA kernelu, bez nutnosti jeho kopírovania pomocou `cudaMemcpy()` resp. `cudaMemcpyAsync()`. Tu vzniká ďalší problém. GPU má totiž iný virtuálny pamäťový priestor ako CPU. Takže buffer `a` má inú prístupovú adresu z CPU ako z GPU. Ak teda chceme pristupovať k bufferu `a` priamo z GPU, musíme získať validný pointer na jeho adresu. Ten získame volaním:

```
...
cudaHostGetDevicePointer( &dev_a, a, 0 );
//...volanie nejakého kernelu...
cudaThreadSynchronize();
...
```

Pointer `dev_a` v skutočnosti ukazuje na nekopírovanú pamäť v CPU, no vďaka `cudaHostGetDevicePointer()` sme dosiahli, že kernel tento pointer vníma akoby ukazoval do pamäťového priestoru GPU. Volaním `cudaThreadSynchronize()` synchronizujeme CPU a GPU. V dobe vykonávania kernelu je obsah bufferu v nekopírovanej pamäti nedefinovaný, takže použitím `cudaThreadSynchronize()` zaistíme, že kernel je ukončený a nekopírovaná pamäť, na ktorú ukazuje `dev_a`, má správny obsah.

2.11 Používane viacerých GPU

NVIDIA ponúka užívateľom možnosť pridať viac grafických kariet do rozdielnych PCI Express slotov, ktoré sú spojené **SLI (Scalable Link Interface) mostíkmi**. V tejto kapitole si ukážeme, ako paralelizovateľnú úlohu rozložiť medzi viac GPU prepojených s SLI technológiou tak, aby sme dosiahli maximálny výpočtový výkon.

Úlohou nasledujúceho programu bude vypočítať skalárny súčin dvoch vektorov na 2 grafických kartách. Každá grafická karta vypočíta skalárny súčin z polovice vektoru `a` a z polovice vektoru `b`. Na začiatku programu musíme najskôr zistiť koľko CUDA-kompatibilných GPU sa nachádza v našom systéme. Takže hlavný program aplikácie bude začínať nasledujúcou časťou kódu.

Výpis 2.11: Používanie viacerých GPU v aplikácii.

```
#define N = 1024 * 1024;

//zariadeniu-špecifická dátová štruktúra
struct DataStruct {
  int deviceID; //id GPU
  int size;   //velkosť vektoru
  float *a;   //vektor a
  float *b;   //vektor b
  float returnValue; //výsledný skalár
};

int main( void ) {
  DataStruct data[2];
  int GPUcount;
  //zistenie počtu CUDA kompatibilných GPU
  cudaGetDeviceCount( &GPUcount );
  if (GPUcount < 2) {
    printf("Potrebujeme␣aspoň␣dve␣CUDA␣kompatibilné"
    "␣GPU.␣Bolo␣nájdených:␣%d␣zariadení\n", GPUcount);
  }
  //pokračovanie ďalej
```

Ďalej alokujeme štandardnú pamäť na CPU klasickou funkciou `malloc()`.

```
  //pokračovanie main()

  float *a = (float*)malloc(sizeof(float) * N);
  float *b = (float*)malloc(sizeof(float) * N);

  //vyplnenie pamati dátami
  fill_buffers(a, b);

  //inicializia dát pre obe GPU
  data[0].deviceID = 0; data[1].deviceID = 1;
  data[0].size = N/2;   data[1].size = N/2;
  data[0].a = a;        data[1].a = a + N/2;
  data[0].b = b;        data[1].b = b + N/2;

  //pokračovanie ďalej
```

K vôli prehľadnosti funkciu `fill_buffers()` nikde definovať nebudeme. Jednalo by

sa však o jednoduchý **for cyklus**, ktorý plní oba zásobníky naraz náhodnými hodnotami. V tomto bode programu prichádza dôležitá vec. **Zásadnou podmienkou pri používaní viacerých grafických kariet je, že každá z nich musí byť ovládaná iným CPU vláknom.** K manažmentu CPU vlákien budú v nasledujúcom kóde použité 2 funkcie:

- `start_thread()`
- `end_thread()`

Prvý parameter funkcie `start_thread()` bude pointer na funkciu `routine()`, ktorú chceme volať novo vytvoreným vláknom. Druhým parametrom bude štruktúra typu `DataStruct` obsahujúca dáta pripravené na spracovanie pre 1. GPU. Funkcia `start_thread()` teda vytvorí nové vlákno, ktoré zavolá funkciu `routine()`, s argumentom typu `DataStruct`. Druhým volaním funkcie `routine()` s argumentom `DataStruct` sa z hlavného programu automaticky vytvorí druhé vlákno.

```
  //pokračovanie main()

  //vytvorenie prvého vlákna, ktoré volá funkciu routine()
  CUTThread thread = start_thread( routine, &(data[0]) );
  //ďalšie volanie funkcie routine() prebieha z vlákna
  //hlavného programu
  routine( &(data[1]) );
  //vzájomné čakanie vlákien
  end_thread( thread );

  free( a );
  free( b );
  printf("Skalarny sucet vektorov je: %d",
      data[0].returnValue + data[1].returnValue);
  return 0;
} //ukoncenie main()
```

`end_thread()` zaistí čakanie vlákna hlavného programu na novo vytvorené vlákno. Teraz sa pozrieme na obsah funkcie `routine()`. Jediný vstupný argument tejto funkcie je pointer na typ `void` a taktiež vracia pointer na tento typ. Typ `void` sa v tomto prípade využíva z dôvodu jednoduchého použitia v rôznych ďalších implementáciách funkcie `start_thread()`.

```
void *routine( void *pData ){
  DataStruct *data = (DataStruct*)pData;
```

```
  cudaSetDevice(data -> deviceID);
  //pokračovanie ďalej
```

Obe vlákna vytvorené na CPU volajú funkciu `cudaSetDevice()`. Každé z nich tejto funkcii predá zariadeniu-špecifické ID, čím zaistíme, že každé vlákno bude manipulovať s inou grafickou kartou. Nasleduje deklarácia a alokácia pamäti na CPU a GPU, čo vyžaduje volanie základných funkcii, ktoré sme rozoberali vyššie v tejto práci. Preto uvedieme len stručný postup v komentároch.

```
  //pokračovanie routine()
  int size = data->size; //dlzka vektorov
  float *a, *b, *partial_c, c; //pamať na CPU
  float *dev_a, *dev_b, *dev_partial_c; //pamať na GPU

  //naplnenie pamati alokovanej na CPU datami
  a = data->a;
  b = data->b;
  //alokovanie pamati pre čiastočný výsledok na CPU
  partial_c = (float*)malloc( blocksPerGrid*sizeof(float) );

  //alokovanie pamati na GPU pomocou cudaMalloc()...
  //kopírovanie pamati do GPU pomocou cudaMemcpy()...
  //volanie kernelu počítajúceho vektorový súčin...
  //kopírovanie pamati spať do CPU...
  //spracovanie výsledku...
  //uvolnenie pamati na GPU - cudaFree()...
  //uvolnenie pamati na CPU - free()...
} //ukončenie routine()
```

2.11.1 Prenosná pamäť s blokovaným stránkovaním (portable memory)

V kapitole 2.8 sme sa oboznámili s pamäťou s blokovaným stránkovaním. Zjednodušene, je to vlastne pamäť alokovaná na CPU, ktorá má stále miesto vo fyzickej pamäti a nemôže dôjsť k jej realokácií. Je podstatné uvedomiť si poslednú dôležitú vec. **Nestránkovaná pamäť sa javí ako nestránkovaná iba pre vlákno, ktoré túto pamäť alokovalo.** Všetky ostatné vlákna považujú túto pamäť za stránkovanú, čo v prípade nesprávneho použitia môže v lepšom prípade až dvojnásobne predĺžiť dobu kopírovania medzi CPU a GPU. V tom horšom prípade, môže dôjsť k

snahe kopírovať túto pamäť pomocou `cudaMemcpyAsync()`, čo by viedlo k drastickej smrti programu, pretože `cudaMemcpyAsync()` kopíruje iba nestránkovanú pamäť.

Existuje však riešenie, ktorým je **prenosná nestránkovaná pamäť** (angl. *portable pinned memory*). Túto pamäť je možné prenášať medzi CPU vláknami tak, že každé z vlákien ju bude považovať za nestránkovanú. Toho dosiahneme použitím vyššie spomínaného flagu *cudaHostAllocPortable* vo funkcii `cudaHostAlloc()`. Tento flag môžeme skombinovať s ďalšími, vyššie použitými `cudaHostAllocWriteCombined` a `cudaHostAllocMapped`.

Keby sme tento druh pamäti chceli využiť v príklade 2.11, musel by tento program prejsť tromi zásadnými zmenami. Ešte pred samotnou alokáciou prenosnej nestránkovanej pamäti pre vektory `a` a `b` je potrebné nastaviť zariadenie, na ktorom bude program bežať - `cudaSetDevice()`. Ďalej musíme oznámiť vybranému GPU, že budeme manipulovať s nekopírovanou pamäťou - *cudaSetDeviceFlags()*. Až teraz použijeme funkciu `cudaHostAlloc()` namiesto obyčajnej `malloc()` funkcie k alokácii vektorov `a` a `b`. Nasledujúci kód je náhradou za časť kódu alokujúceho pamäť na CPU v príklade 2.11.

```
int main( void ){
  ...
  cudaSetDevice( 0 ); //volba GPU
  //povolenie GPU manipulovať s nekopírovanou pamaťou
  cudaSetDeviceFlags( cudaDeviceMapHost );
  //alokácia nekopírovanej, nestránkovanej pamati na CPU
  cudaHostAlloc( (void**)&a, N*sizeof(float),
          cudaHostAllocWriteCombined |
          cudaHostAllocPortable |
          cudaHostAllocMapped );
  cudaHostAlloc( (void**)&b, N*sizeof(float),
          cudaHostAllocWriteCombined |
          cudaHostAllocPortable |
          cudaHostAllocMapped );
  ...

  //uvolnenie pamati na GPU - cudaFreeHost()..
  ...
} //ukončenie main()
```

Nesmieme zabudnúť, že pamäť na CPU bola v tomto prípade alokovaná pomocou `cudaHostAlloc()`, takže ju musíme na konci hlavnej funkcie `main()` uvoľniť volaním `cudaFreeHost()`.

Ďalšia zmena by sa prejavila hneď na začiatku funkcie `routine()`. Našou snahou je, aby tento program využíval obe grafické karty. V príklade 2.11 sme hneď na začiatku funkcie `routine()` volali `cudaSetDevice()`, aby sme zaistili, že každé vlákno ovláda iné GPU. V tomto modifikovanom príklade sme už raz `cudaSetDevice()` museli volať z hlavného vlákna (pretože sme alokovali nestránkovanú, prenosnú pamäť). Takže teraz musíme vo funkcii `routine()` nastaviť aj 2. zariadenie:

```
void* routine( void *pData ) {
  DataStruct *data = (DataStruct*)pData;
  if (data->deviceID != 0){
    cudaSetDevice( data->deviceID );
    cudaSetDeviceFlags( cudaDeviceMapHost );
  }
  //pokračovanie ďalej
```

Vektory `a` a `b` sme v hlavnom programe umiestnili do **nestránkovanej, nekopírovanej, prenosnej pamäti na CPU**. To že je táto pamäť nekopírovaná, nám umožní do nej pristupovať priamo z GPU z čoho vyplýva 3. zmena. Namiesto `cudaMemcpy()` budeme volať `cudaHostGetDevicePointer()`, čím dostaneme validný pointer do pamäti CPU.

```
  //pokračovanie routine()
  ...
  cudaHostGetDevicePointer( &dev_a, a, 0 );
  cudaHostGetDevicePointer( &dev_b, b, 0 );
  cudaMalloc( (void**)&dev_partial_c,
          blocksPerGrid*sizeof(float) );
  //volanie kernelu počítajúceho vektorový súčin...
  //kopírovanie  čiastočného výsledku spať do CPU
  cudaMemcpy( partial_c, dev_partial_c,
              blocksPerGrid*sizeof(float),
              cudaMemcpyDeviceToHost );
  //dokončenie výpočtu na CPU...
  //uvolnenie pamati na GPU - cudaFree()
  //uvolnenie pamati na CPU - free()
  //zápis výsledku do štruktúry data
} //ukončenie routine()
```

2.12 Časovanie

K časovaniu volaní CUDA funkcii je prakticky možné použiť rovnako buď CPU alebo GPU časovače. Je tu však jedna vec na ktorú treba dávať pozor, pri používaní CPU časovačov. Je kriticky dôležité zapamätať si, že CUDA API funkcie sú asynchrónne, čo znamená, že vracajú kontrolu späť do CPU bez ohľadu na to či svoju úlohu dokončili alebo nie. Na to isté je treba dávať pozor aj v prípade volania kernelov, a asynchrónnych prenosov dát medzi CPU a GPU.

CPU časovače. Aby sme teda zabránili možným komplikácia a nepresnostiam pri použití CPU časovačov, je dôležité volať funkciu *cudaDeviceSynchronize* okamžite po spustení a ukončení CPU časovania. Táto funkcia zablokuje vlákna na CPU, až kým všetky CUDA volania nebudú dokončené.

GPU časovače. CUDA API sprostredkováva špeciálne funkcie na vytvorenie, odstránenie a zaznamenávanie udalostí pomocou časových známok.

```
cudaEvent_t start, stop;
float time;

cudaEventCreate(&start); //vytvorenie start udalosti
cudaEventCreate(&stop);  //vytvorenie stop udalosti

cudaEventRecord(start, 0);
kernel<<<grid, threads>>>(out_data, in_data, length);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

cudaEventElapsedTime(&time, start, stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Funkcia *cudaEventRecord* vloží udalosti *start* a *stop* do predvoleného streamu. Keď zariadenie narazí v streame na tieto udalosti uloží ich časovú známku. *cudaEventElapsedTime* po skončení časovanie vráti čas, ktorý ubehol medzi udalosťami *start* a *stop*. Vrátená hodnota reprezentuje čas v milisekundách s rozlíšením na polovicu mikrosekundy.

3 OPENCL - OPEN COMPUTING LANGUAGE

OpenCL štandard bol predstavený spoločnosťou Apple a neskôr sa stal súčasťou organizácie Khronos Group. Cieľom OpenCL je zjednodušenie prístupu k paralelnému programovaniu a poskytnutie zrýchlenia paralelného hardvéru pri vykonávaní kódu. Je považovaný za prvý voľne dostupný štandard pre GPGPU. Poskytuje uniformné programovacie prostredie k vytvoreniu efektívneho a univerzálneho kódu pre viacjadrové procesory ako GPU, CPU alebo DSP. Portabilita OpenCL kódu medzi platformami od rôznych výrobcov (napr. AMD, Intel, NVIDIA, TI,...atď), môže byť obrovským prínosom oproti CUDA SW architektúre, ktorá je určená len pre HW od spoločnosti NVIDIA. Portabilita je zaručená prostredníctvom kompilovania kernelov na hostiteľskom procesore počas behu programu. [8].

3.1 Tok programu v OpenCL

Každá OpenCL aplikácia pozostáva z **hostiteľského kódu** a **kódu zariadenia**. **Hostiteľ** (CPU) koordinuje a zoraďuje povely na vykonanie kernelov alebo dátových transferov. **Zariadenie** vykonáva kernel pomocou skupiny jadier často nazývanej ako **NDRange**. Hostiteľský OpenCL C kód na hostiteľovi prevádza nasledujúce úkony [8]:

1. Alokuje a inicializuje hostiteľskú vyrovnávaciu pamäť.
2. Vyhľadá všetky dostupné **platformy** a **zariadenia**
3. Nastaví platformu
4. Na základe nastavenej platformy vyhľadá všetky dostupné zariadenia a vyberie jedno, na ktorom budu spracovávané kernely.
5. Vytvorí **kontext** pre vybrané zariadenie.
6. Vytvorí **príkazovú radu**.
7. Vytvorí vyrovnávacie pamäte na zariadení.
8. Nakopíruje obsah vyrovnávacích pamätí z hostiteľa do vyrovnávacích pamätí na zariadení.
9. Nastaví argumenty pre kernel.
10. Spustí kernel na zariadení.
11. Nakopíruje obsah vyrovnávacích pamätí zo zariadenia späť do vyrovnávacích pamätí na hostiteľovi (tento krok nie je potrebný, ak potrebujeme ponechať obsah pamäti na zariadení za účelom jej ďalšieho spracovávania).
12. Počká kým sa dokončia všetky príkazy v príkazovej rade.
13. Na záver z pamäti uvoľní všetky alokované vyrovnávacie pamäte a objekty.

Ďalej v tomto texte sa bude detailnejšie zaoberať jednotlivými pojmami týkajúcich sa toku programu v OpenCL

3.2 Architektúra OpenCL

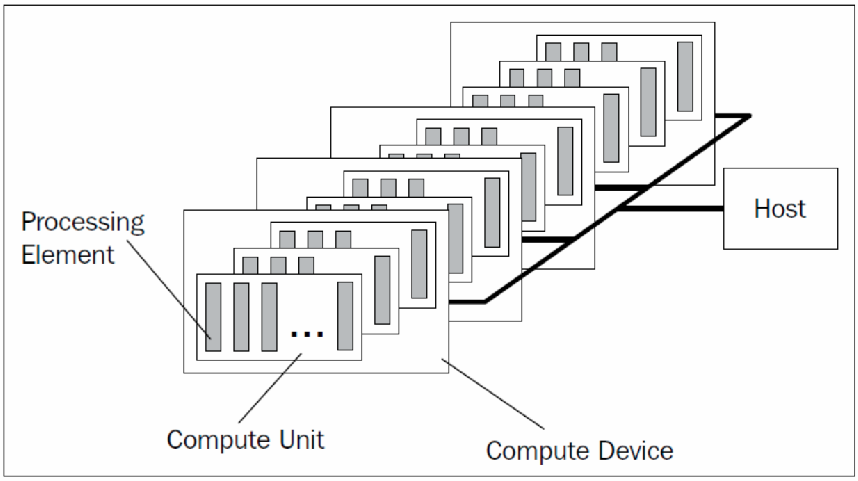
Na rozdiel od NVIDIA®CUDA frameworku, ktorý je kompatibilný iba so zariadeniami od NVIDIe, sú knižnice vytvorené v OpenCL multiplatformné a tým pádom ponúkajú akceleráciu naprieč mnohými zariadeniami s paralelnou architektúrou. OpenCL preto disponuje nízkoúrovňovou hardvérovou abstrakciou a programovacím modelom, ktorý zaručuje podporu pre rôzne hardvérové architektúry.

OpenCL framework je popísaný hierarchiou nasledujúcich modelov [8]:

- Model platformy
- Model pamäti
- Výkonný model
- Programovací model

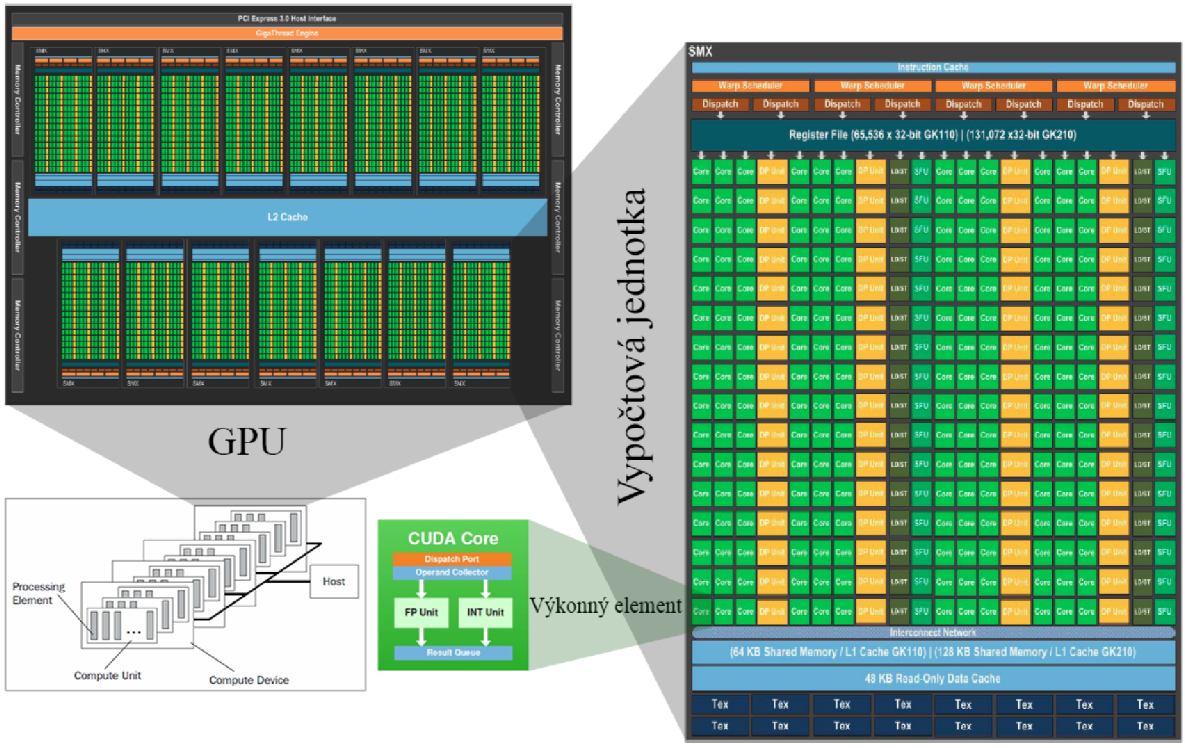
3.2.1 Model Platformy

Model platformy pozostáva z hostiteľskej jednotky (najčastejšie CPU) a viacerých zariadení, ktoré sú k nej pripojené (CPU, GPU, DSP...atď.) obvykle zbernicami s vysokou priepustnosťou dát. Každé OpenCL zariadenie obsahuje niekoľko **výpočtových jednotiek** (angl. compute unit), ktoré je možné ďalej rozložiť na **výkonné elementy** (angl. processing elements) (viď obr. 3.1). Každý **výkonný element** vykonáva instanciu kernelu konkrétneho **pracovného elementu**.



Obr. 3.1: OpenCL model platformy. [8]

Ako príklad uvedieme OpenCL architektúru GK110 grafickej karty od NVIDIA®(viď obr. 3.2). GPU je spojené s hostiteľom a disponuje 15 **výpočtovými jednotkami**, kde každá z nich obsahuje 192 **výkonných elementov**.



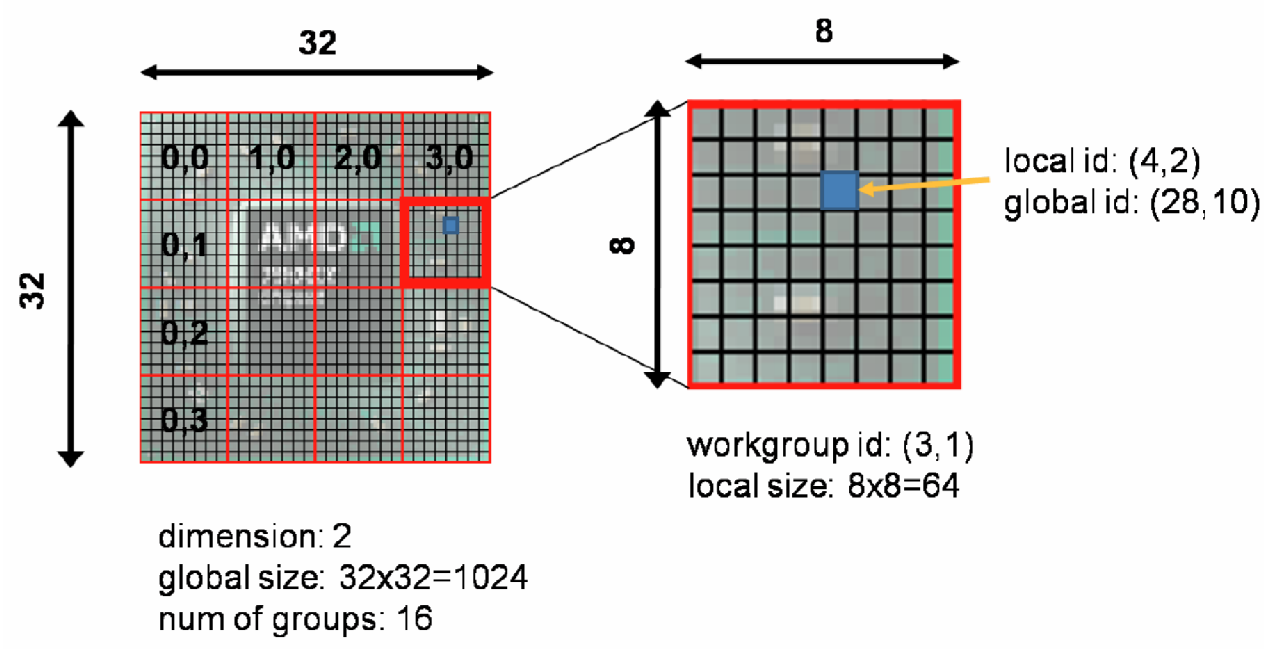
Obr. 3.2: Architektúra čipu GK110 grafickej karty od NVIDIA®

3.2.2 Výkonný model

Dve hlavne jednotky v tomto modeli sú **hostiteľský program** a **kernel**. Kernel je vždy vykonávaný na OpenCL zariadení, zatiaľ čo hosťovský program beží na hostiteľovi. Po príkaze hostiteľa sa kernel odošle do zariadenia, na ktorom sa vytvorí N-dimenzionálny indexovaný priestor - **NDRange**. Na spracovávanie lineárneho pola by sme teda zvážili použitie 1-dimenzionálneho priestoru, na spracovanie obrazu 2-dimenzionálneho a na spracovanie objemu 3-dimenzionálneho priestoru. Každému prvku v tomto priestore je priradená jedna instancia kernelu - **pracovný element**. **Kernely** sú podobné klasickým C funkciám, ktoré však môžu byť dátovo-paralelné alebo úlohovo-paralelné. Do **príkazovej rady** sú zadávané v určitom poradí a ich vykonávanie môže byť závislé alebo nezávislé na tomto poradí.

Abstrakcia OpenCL umožňuje zoskupovať **pracovné elementy** do **pracovných skupín**. Rovnako ako každý **pracovný element**, tak aj **pracovná skupina** má

svoj špecifický index. Jednotlivé **pracovné elementy** z rovnakej **pracovnej skupiny** sa naraz vykonávajú na zariadení. Tento systém umožňuje všetkým pracovným elementom z rovnakej pracovnej skupiny pristupovať do zdielanej pamäti (viď. 3.2.3) a ich vzájomnú synchronizáciu [8].



Obr. 3.3: Príklad NDRange s 32x32 **pracovnými elementami** a 4x4 **pracovnými skupinami**. [11]

Na obrázku 3.3 je uvedený príklad 2-dimenzionálneho NDRange priestoru s počtom 32x32 pracovných elementov a 4x4 pracovných skupín. Každý jeden z pracovných elementov má priradený špecifický **lokálny** a **globálny** index. Modrou farbou zvýraznený pracovný element má **lokálny** index (4,2) a **globálny** index (28,10)

Obrázok 3.4 porovnáva skalárnu a paralelnú implementáciu výpočtu druhej mocniny prvkov v poli. V prvom prípade je potrebné vo *for* cykle iterovať cez všetky prvky pola a vypočítať ich druhú mocninu. V paralelnej implementácii stačí zistiť index aktuálneho pracovného elementu, ktorý vykoná svoju instanciu kernelu.

V tejto sekcii sme diskutovali o **pracovnom elemente, pracovnej skupine, lokálnom indexe** a **globálnom indexe**. Každá z týchto hodnôt môže byť determinovaná počas behu kernelu použitím OpenCL SDK vstavaných funkcii:

- *get_global_id(int dim)* - globálny index pracovného elementu
- *get_local_id(int dim)* - lokálny index pracovného elementu
- *get_num_groups(int dim)* - počet pracovných skupín v NDRange priestore
- *get_group_size(int dim)* - veľkosť pracovnej skupiny

- *get_group_id(int dim)* - lokálny index pracovnej skupiny

```
__kernel void square(__global const float *a,
                     __global float *result)
{
    int id = get_global_id(0);
    result[id] = a[id] * a[id];
}
```

```
void square(const float *a, int n,
            float *result)
{
    for (int i = 0;i < n;i++)
        result[i] = a[i] * a[i];
}
```

Obr. 3.4: Porovnanie paralelnej (vľavo) a skalárnej (vpravo) implementácie.

3.2.3 Model pamäti

V pamäťovo koherentných systémoch je zaručené, že dáta uložené v lokálnej vyrovnávacej pamäti procesoru sú konzistentné naprieč všetkými procesormi. Programátor sa teda v prípade pamäťovo koherentného systému nemusí zaoberať rozdeľovaním pamäti. OpenCL pamäťový model je rovnako dobre prispôsobený na prácu s pamäťovo koherentnými systémami. Programátor však musí poznať spôsob rozdeľovania dát do jednotlivých pamäťových regiónov na to, aby dosiahol maximálny výkon na paralelných, heterogénnych systémoch. OpenCL štandard teda definuje 4 pamäťové regióny (viď. obr. 3.5), kde každý z nich je prístupný všetkým pracovným elementom, ktoré vykonávajú kernel [8]:

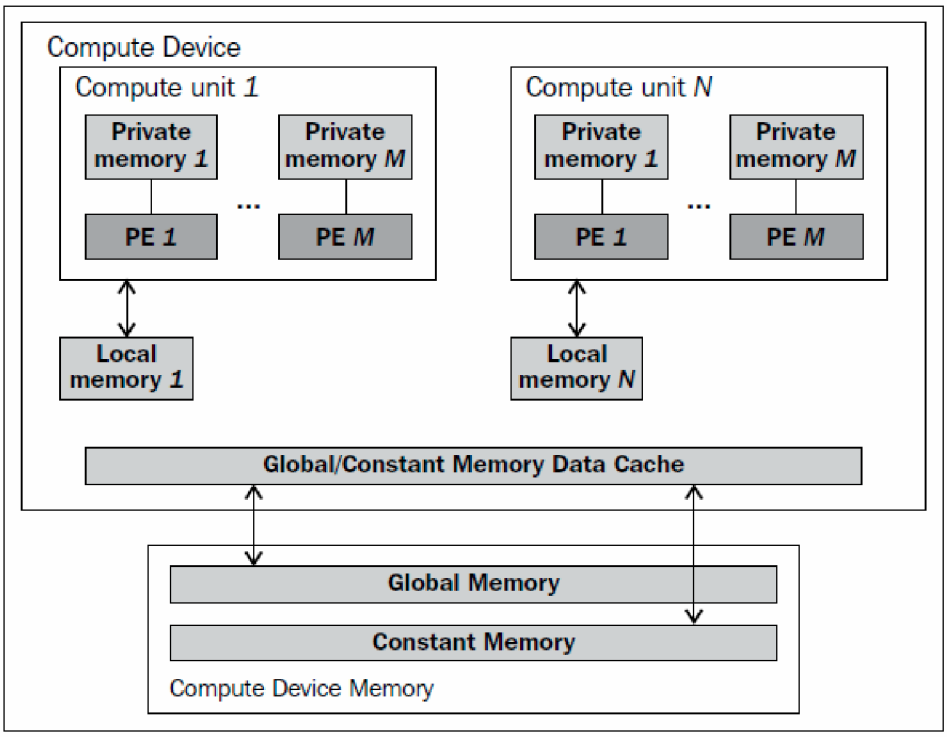
Globálna pamäť Je to najväčšia pamäť v zariadení. Prístup do tejto pamäti má veľkú latenciu, no môže byť zarovnaný, čo zaručí splývajúce čítanie a teda zvýšenie celkového výkonu. Všetky pracovné prvky zo všetkých pracovných skupín v priestore NDRange majú právo čítať resp. zapisovať do tejto pamäti. Globálna pamäť môže byť zachytávaná rýchlymi vyrovnávacími pamäťami. Globálna pamäť sa identifikuje kľúčovým slovom **___global**.

Konštantná pamäť Je inicializovaná hostiteľom, pracovné elementy z nej môžu iba čítať. Je to región globálnej pamäti, ktorá zostáva konštantná počas behu programu. V prípade niektorých zariadení je táto pamäť načítaná do rýchlej vyrovnávacej pamäte, čo výrazne zrýchli jej čítanie. Identifikuje sa kľúčovým slovom **___constant**.

Lokálna pamäť Z pohľadu HW je lokálna pamäť najbližšie k výkonovej jednotke (on-chip), čo zaručuje až 100-násobné zrýchlenie prístupu oproti globálnej pamäti. Z pohľadu abstrakcie OpenCL má každý pracovný element z rovnakej pracovnej skupiny prístup do tejto pamäti. To znamená, že pracovné elementy z rovnakej

pracovnej skupiny môžu zdielať dáta uložené v lokálnej pamäti. Lokálna pamäť sa identifikuje kľúčovým slovom **___local**.

Privátna pamäť Každý pracovný element má k dispozícii vlastnú privátnu pamäť. Táto pamäť je v HW mapovaná na výkonné elementy, ktoré vykonávajú kernel. Obvykle ju používa OpenCL kompilátor k alokácii všetkých lokálnych premenných v kernely. Prípadné zmeny v tomto regióne sú viditeľné iba pre daný pracovný element. Privátna pamäť sa identifikuje kľúčovým slovom **___private**.



Obr. 3.5: OpenCL pamäťový model. [8]

Vzhľadom k pamäťovej architektúre na obrázku 3.5) je jasné, že pracovné elementy z rovnakej pracovnej skupiny sú v vykonávané výkonovými elementami v rovnakej výkonovej jednotke. To znamená, že pracovná skupina je vždy asociovaná s jednou výkonnou hardvérovou jednotkou.

3.3 Príklad OpenCL hostiteľského programu

V tejto sekcii ukážeme základnú kostru implementácie OpenCL. Krok po kroku prejdeme vytvorenie OpenCL objektov, ktoré sú nevyhnutné pre kooperáciu hostiteľa (CPU) so zariadením (GPU). Popíšeme vzťah medzi **zariadením, kontextom,**

programom, kernelom, pamäťovými objektami a príkazovými radami. Úlohou funkcie *getErrorString* je správa chybových hlásení, ktoré vracajú funkcie z OpenCL API pri ich nesprávnom použití.

Hneď po zahrnutí OpenCL knižníc (#include CL/cl.h) do aplikácie definujeme názvy premenných, ktoré budeme neskôr používať. V prvom kroku získame informáciu o dostupných **platformách** a **zariadeniach** v systéme.

```
11 /*START MAIN FUNCTION*/
12 int main(int argc, char **argv)
13 {
14     cl_uint num_platforms = 0, num_devices = 0, num_kernels = 1;
15     cl_uint chosen_device, chosen_platform;
16
17     //Loading platform IDs.
18     cl_char platform_info[1024] = { 0 };
19     getErrorString(clGetPlatformIDs(5, &platform, &num_platforms));
20     if (!num_platforms)
21         printf("ERROR: No platforms has been found for chosen platform\n"); exit(1);
22     chosen_platform = wait_for_user_choice(num_platforms);
23     getErrorString(clGetPlatformInfo(platform[chosen_platform], CL_PLATFORM_NAME,
24                                      sizeof(platform_info), platform_info, NULL));
```

Funkcia *clGetPlatformIDs* na 19. riadku nám vráti na adrese *platform[]* zoznam všetkých **dostupných platforiem** v systéme a na adresu *num_platforms* zapíše ich počet. Funkcia *clGetPlatformInfo* na riadku 23 nám vráti informácie o konkrétnej platforme zo zoznamu *platform[]*. Podľa toho akú informáciu chceme obdržať, musíme nastaviť konkrétny flag v prvom argumente funkcie. V tomto prípade chceme získať názov platformy čomu odpovedá flag CL_PLATFORM_NAME v prvom argumente funkcie.

```
26  //Loading device IDs
27  cl_char device_info[1024] = { 0 };
28  getErrorString(clGetDeviceIDs(platform[chosen_platform], CL_DEVICE_TYPE_GPU,
29                                 1, &device, &num_devices));
30  if (!num_devices)
31      printf("ERROR: No devices has been found for chosen platform\n"); exit(1);
32  chosen_device = wait_for_user_choice(num_devices);
33  getErrorString(clGetDeviceInfo(device[chosen_device], CL_DEVICE_NAME,
34                                 sizeof(device_info), device_info, NULL));
```

Keď už máme k dispozícii list všetkých dostupných platforiem, môžeme pomocou funkcie *clGetDeviceIDs* na riadku 28 zistiť **dostupné zariadenia** pre konkrétnu platformu. Všetky nájdené zariadenia sa zapíšu na adresu zoznamu *device* a ich počet na adresu *num_devices*. Prvým argumentom nastavujeme typ zariadení ktoré chceme hľadať (CL_DEVICE_TYPE_GPU vs CL_DEVICE_TYPE_CPU). Druhým argumentom môžeme nastaviť koľko zariadení daného typu chceme nájsť. Funkciou *clGetDeviceInfo* získame požadované informácie o vybranom zariadení.

```
36 //Creating command queue and context for chosen device.
37 cl_context_properties props[3] = { CL_CONTEXT_PLATFORM, 0, 0 };
38 props[1] = (cl_context_properties)platform[chosen_platform];
39 ctx = clCreateContext(props, num_devices, &device[chosen_device],
40                       NULL, NULL, &err);
41 getErrorString(err);
42 queue = clCreateCommandQueue(ctx, device[chosen_device],
43                              CL_QUEUE_PROFILING_ENABLE, &err);
44 getErrorString(err);
```

Už sme našli ID zariadenia, s ktorým chceme ďalej pracovať. Toto ID môže byť asociované s nejakým **kontextom**. OpenCL API používa kontext na správu **príkazovej rady, programových objektov, kernelových objektov** a rovnako sa používa aj pri **transfere pamäti medzi hostiteľom a vybraným zariadením**. Kontext môže byť asociovaný s viacerými zariadeniami. Na jeho vytvorenie slúži funkcia - *clCreateContext* - volaná na riadku 39.

Jej nultým argumentom je zoznam vlastností *props*, ktorý je definovaný na riadkoch 37, 38. Nultý element pola *props* je CL_CONTEXT_PLATFORM a značí, že chceme vytvoriť kontext pre platformu s ID zadaným na prvom mieste v tomto poli.

Prvý argument *clCreateContext* označuje počet zariadení v zozname *device* a druhým argumentom je samotný zoznam zariadení - *device*. Ak všetko prebehne úspešne, funkcia vráti hodnotu *cl_context*.

Na riadku 42 je volaná funkcia *clCreateCommandQueue*, ktorá na základe špecifického kontextu vytvorí **príkazovú radu**. Príkazová rada bude ďalej slúžiť k posielaniu príkazov z hostiteľa do zariadenia, ktoré je asociované s kontextom. Druhým argumentom je v tomto prípade flag *CL_QUEUE_PROFILING_ENABLED*, ktorý povolí profilovanie, resp. časovanie transferu dát a vykonávania kernelu pomocou udalostí (viď. Udalosti a ich monitorovanie). Ak kontext obsahuje zariadenia typu GPU aj CPU, je nutné vytvoriť pre oba typy individuálnu príkazovú radu. Funkcia *clCreateCommandQueue* vráti hodnotu *cl_command_queue*.

```
46 //Loading source code from e file.
47 long opencl_file_size;
48 const char *filename = "device.cl";
49 const char *kernel_code = load_program_source(filename, &opencl_file_size);
50 program = clCreateProgramWithSource(ctx, num_kernels,
51                                     (const char **)&kernel_code, NULL, &err);
52 getErrorString(err);
```

Súčasťou OpenCL programu je kolekcia kernelov. Kernely sú písané v OpenCL C jazyku a sú kompilované pre konkrétne zariadenia až za behu programu. Ako sme spomenuli vyššie, kernel je označený identifikátorom **___kernel** a je definovaný dvoma rôznymi spôsobmi:

- ako pole konštantných reťazcov v hostiteľskom programe
- ako klasická funkcia v separátnom súbore.

Programový objekt je vytvorený funkciou *clCreateProgramWithSource*. Tento objekt enkapsuluje zdrojový program alebo binárny súbor, poslednú úspešne skompilovanú verziu programu, zoznam zariadení, pre ktoré je program určený a nastavenia pre zostavenie programu (angl. building options). V tejto ilustrácii počítame s tým, že samotný kernel (viď obr. 3.4 vľavo) je uložený v separátnom súbore, takže ho potrebujem načítať do konštantného reťazca, čo sa deje na riadku 49 pomocou funkcie *load_program_source*. Keby sme do programového objektu chceli zahrnúť viac zdrojových súborov s kernelmi, musíme vytvoriť pole reťazcov, kde každý z nich bude uchovávať obsah jednotlivých zdrojových súborov. Ich počet definujeme v 1. argumente funkcie *clCreateProgramWithSource*, ktorá je volaná na riadku 50.

```
54 //Building program.
55 err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
56 if (err < 0) {
57     getErrorString(err);
58     size_t log_size = 0;
59     ///Find size of log and print to std output
60     clGetProgramBuildInfo(program, device[chosen_device],
61                           CL_PROGRAM_BUILD_LOG, 0, NULL, &log_size);
62     char *program_log = (char*)malloc(log_size + 1);
63     program_log[log_size] = '\0';
64     clGetProgramBuildInfo(program, device[chosen_device],
65                           CL_PROGRAM_BUILD_LOG, log_size + 1, program_log, NULL);
66     printf("Build info:\n%s\n", program_log);
67     free(program_log);
68     exit(1);
69 }
```

Keď už máme vytvorený programový objekt, je na čase prejsť k ďalšiemu kroku, čím je kompilácia a linkovanie programového objektu. **Spustiteľný program** (angl. executable) je zostavený prostredníctvom funkcie *clBuildProgram* na riadku 55. Či je tento program zostavený pre jedno alebo viac zariadení, je definované už v samotnom programovom objekte *cl_program*, ktorý je do funkcie *clBuildProgram* predaný pomocou nultého argumentu. Tretím argumentom je možné predať funkcii *clBuildProgram* špeciálne nastavenia pre kompilátor. Jedná sa napríklad o nastavenia preprocesoru, optimalizácie a vstavaných matematických operácii.

Počas kompilácie programového objektu môže dôjsť k rôznym chybám. Tieto chyby sú ukladané do logu, ktorý je súčasťou programového objektu. Prístup do logu je možný prostredníctvom funkcie *clGetProgramBuildInfo* volanej na riadku 64.

Akonáhle máme k dispozícií spustiteľný program, môžeme vytvoriť **objekt kernelu** - *cl_kernel*. Každý objekt kernelu enkapsuluje samostatný kernel spolu s argumentami, ktoré sú s nim asociované. Funkcia *clCreateKernel* volaná na riadku

```
71 //Creating kernel.
72 char kernelname[100];
73 sprintf(kernelname, "kernel%d", 0);
74 kernel = clCreateKernel(program, kernelname, &err);
75 getErrorString(err);
```

74, nám teda vráti objekt *cl_kernel*, ktorý bude neskôr priradený do už vytvorenej, príkazovej rady. Ešte pred tým je však nevyhnutné nastaviť argumenty pre objekt kernelu. Ak kernel na vstupe požaduje nejaké **pamäťové objekty** (buffer, obraz), je nutné tieto objekty najskôr definovať.

```
77 //Creating buffers with data.
78 cl_int N = 1024;
79 size_t d_input_size = N * sizeof(cl_int);
80 size_t d_output_size = N * sizeof(cl_int);
81 int *h_input = (int*)malloc(d_input_size);
82 for (int i = 0; i < N; i++)
83     h_input[i] = i;
84 int *h_output = (int*)malloc(d_output_size);
85 memset(h_output, 0, d_output_size);
86
87 cl_mem d_input = clCreateBuffer(ctx, CL_MEM_READ_ONLY, d_input_size, NULL, &err);
88 getErrorString(err);
89 cl_mem d_output = clCreateBuffer(ctx, CL_MEM_READ_WRITE, d_output_size, NULL, &err);
90 getErrorString(err);
```

Na riadkoch 81-85 sme alokovali a inicializovali pamäťový zásobník umiestnený v hostiteľskej RAM. Kernel však k tejto pamäti dosah nemá (za podmienok v ilustrovanom príklade), a preto je nutné alokovať rovnako veľký región pamäti aj v zariadení, kam sa pamäť z RAM hostiteľa následne prekopíruje. Pamäť na zariadení je alokovaná funkciami *clCreateBuffer* na riadkoch 87 a 89. Je explicitne definovaná kontextom (*cl_context*), typom pamäti (*CL_MEM_READ_WRITE* | *CL_MEM_READ_ONLY* | *CL_MEM_WRITE_ONLY*) a jej veľkosťou.

```
92 //Setting kernel arguments.
93 getErrorString(clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&d_input));
94 getErrorString(clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*)&d_output));
95 getErrorString(clSetKernelArg(kernel, 2, sizeof(int), (void*)&N));
```

Teraz môžeme pristúpiť k **nastaveniu argumentov kernelu**, čo vykonáme funkciou *clSetKernelArg* na riadkoch 93-95. Každý jeden argument sa nastavuje jedným volaním tejto funkcie. Parametrami funkcie *clSetKernelArg* sú objekt kernelu, poradie nastavovaného argumentu, veľkosť pamäti, ktorá je cez argument predávaná a ukazovateľ na túto pamäť.

V tomto bode všetky argumenty kernelu referujú ku konkrétnej pamäťovej oblasti. Posledná vec pred samotným vykonaním kernelu je **prekopírovať dáta** inicializované na hostiteľovi (CPU) do pamäti zariadenia (GPU), ku ktorej má prístup

aj samotný kernel. Rovnako ako CUDA, aj OpenCL ponúka ekvivalentné spôsoby transferu pamäti medzi hostiteľom a zariadeniami, ktoré v princípe fungujú rovnako ako v CUDA (viď. Pamäť s blokovaným stránkovaním (page-locked memory), Nekopírovaná pamäť s blokovaným stránkovaním (zero-copy memory)):

- **kopírovanie stránkovanej pamäti** hostiteľa do pamäti zariadenia
- **kopírovanie nestránkovanej pamäti** hostiteľa do pamäti zariadenia
- **mapovanie stránkovanej pamäti** hostiteľa do pamäti zariadenia
- **mapovanie nestránkovanej pamäti** hostiteľa do pamäti zariadenia

```
97  const size_t local[3] = { 32, 0, 0 };
98  const size_t global[3] = { 1024, 0, 0 };
99  //Writting data from host to device.
100 getErrorString(clEnqueueWriteBuffer(queue, d_input, CL_TRUE, 0, d_input_size,
101                                      h_input, 0, NULL, NULL));
102 getErrorString(clEnqueueWriteBuffer(queue, d_output, CL_TRUE, 0, d_output_size,
103                                      h_output, 0, NULL, NULL));
104 clFinish(queue);
105 //Executing kernel.
106 err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global, local, 0, NULL, NULL);
107 clFinish(queue);
108 //Writting data from device to host.
109 getErrorString(clEnqueueReadBuffer(queue, d_output, CL_TRUE, 0, d_output_size,
110                                    h_output, 0, NULL, NULL));
111 clFinish(queue);
```

Pre ilustráciu použijeme najjednoduchší spôsob, čím je kopírovanie stránkovanej pamäti hostiteľa do zariadenia. O to sa postará funkcia *clEngueueWriteBuffer* (riadky 100-103), ktorá zaradí príkaz na zápis do príkazovej rady. Z toho vyplýva, že vstupné parametre tejto funkcie musia zahŕňať objekt príkazovej rady - *cl_command_queue*. Ďalšími dôležitými argumentami funkcie *clEnqueueWriteBuffer* je ukazovateľ na pamäťovú oblasť (buffer) v zariadení, do ktorého sa bude zapisovať, veľkosť tejto pamäti a ukazovateľ na pamäť v hostiteľovi, z ktorej sa dáta budú kopírovať.

Na riadku 106 sa volá funkcia *clEnqueueNDRangeKernel*, ktorá zapíše príkaz na vykonanie kernelu do príkazovej rady. Nultým argumentom je teda objekt príkazovej rady *cl_command_queue* a prvým argumentom je objekt kernelu, ktorý sa má vykonať *cl_kernel*.

Z kapitoly Výkonný model vyplýva, že každý pracovný element spracováva svoju instanciu kernelu. OpenCL využíva zariadenia na paralelné výpočty tak, že jednotlivé instancie kernelu sú vykonávané na rôznych častiach N-dimenzionálneho priestoru NDRange. Druhý argument funkcie *clEnqueueNDRangeKernel* udáva počet dimenzii priestoru. Štvrtý argument udáva globálny počet pracovných elementov vo

všetkých dimenziách a piaty argument udáva počet pracovných elementov v jednej pracovnej skupine.

Funkcia *clEngueueReadBuffer* na riadku 109 má rovnaké parametre ako *clEngueueWriteBuffer*. Jediným rozdielom je, že v tomto prípade transfer dát prebieha opačne - z pamäti zariadenia (GPU) do pamäti hostiteľa (CPU).

```
113     for (int i = 0; i < N; i++)
114         printf("%d\n", h_output[i]);
115 
116     //Release CPU memory objects.
117     free(h_output);
118     free(h_input);
119     //Release OpenCL memory objects.
120     clReleaseMemObject(d_output);
121     clReleaseMemObject(d_input);
122     clReleaseProgram(program);
123     clReleaseKernel(kernel);
124     clReleaseCommandQueue(queue);
125     clReleaseContext(ctx);
126     return 0;
127 }
128 /*END MAIN FUNCTION*/
```

Na záver programu prebehne vypísanie elementov výsledného vektoru do konzole a uvoľnenie všetkých alokovaných objektov ako v pamäti hostiteľa tak aj v pamäti zariadenia [10],[8].

3.4 Udalosti a ich monitorovanie

Udalosti (angl. events) sú objekty dátového typu *cl_event*. Tvoria interface medzi hostiteľskou aplikáciou a OpenCL implementáciou. Môžeme ich použiť troma rôznymi spôsobmi [8], [8],[9]:

1. Monitorovanie OpenCL operácii a príkazov. Operáciami sa rozumie predovšetkým transfer dát medzi zariadením a hostiteľom, a vykonanie kernelu v NDRange priestore. V OpenCL udalosti špecifikujú **aktuálny stav príkazov zaradených do príkazovej rady**. Môžu oznamovať hostiteľovi, že daný príkaz v príkazovej rade bol na zariadení úspešne dokončený.

2. Synchronizácia príkazov. Ďalšou užitočnou vlastnosťou udalostí je možnosť **synchronizácie príkazov**. Napríklad príkaz na vykonanie kernelu, ktorý bol zaradený do príkazovej rady, bude čakať, kým sa vykonajú všetky potrebné transfery

dát z hostiteľa do zariadenia a až po ich dokončení sa sám vykoná. V predchádzajúcej kapitole sme si mohli všimnúť príkazy *clEnqueue{Read/Write}Buffer* alebo *clEnqueueNDRangeKernel*.

```
clEnqueueReadBuffer(***, cl_uint num_events_in_wait_list,
          const cl_event *event_wait_list,
          cl_event *event);
```

Každý z týchto príkazov pred samotným vykonaním, musí čakať na dokončenie *num_events_in_wait_list* udalostí v zozname *event_wait_list*. Po dokončení príkazu *clEnqueueReadBuffer* by mal posledný z parametrov *event* obsahovať pointer k udalosti *cl_event*. Udalosť môže byť následne použitá k sledovaniu príkazu *clEnqueueReadBuffer*, ktorý je zaradený do príkazovej rady.

3. Profilovanie udalostí. Profilovanie je dôležitý nástroj na optimalizovanie výkonu aplikácii. V OpenCL k tomu slúžia *cl_event* udalosti, ktoré obsahujú informáciu o časovaní. Túto informáciu je možné získať funkciou *clGetEventProfilingInfo*. Na to, aby sme obdržať informáciu o časovaní určitej operácie, musí byť príkazová rada, do ktorej bola táto operácia začlenená, vytvorená s flagom CL_QUEUE_PROFILING_ENABLE.

*clCreateCommandQueue(***, CL_QUEUE_PROFILING_ENABLE, ***)*

Ak už máme vytvorenú príkazovú radu s povoleným profilovaním, tak môžeme použiť funckiu

```
cl_int clGetEventProfilingInfo(cl_event event,
               cl_profiling_info param_name,
               size_t param_value_size,
               void *param_value,
               size_t *param_value_size_ret);
```

Voľbou parametru *param_name* určíme, akú časovú značku chceme získať. Máme na výber z nasledujúcich parametrov:

- **CL_PROFILING_COMMAND_START** - čas spustenie príkazu
- **CL_PROFILING_COMMAND_END** - čas ukončenia príkazu
- **CL_PROFILING_COMMAND_QUEUED** - čas zaradenia príkazu do príkazovej rady
- **CL_PROFILING_COMMAND_SUBMIT** - čas pridelenia zariadeniu, ktorému odpovedá aktuálna príkazová rada

Na adrese *param_value* dostaneme 64 bitovú *cl_ulong* hodnotu, čo odpovedá jednej z vybraných časových známok (v nanosekundách).

3.5 Synchronizačné bariéry

OpenCL C špecifikuje tzv. **synchronizačné bariéry** zaisťujúce synchronizáciu pracovných elementov z rovnakej pracovnej skupiny. Bariéra zastaví vykonávanie pracovného elementu v určitom bode kernelu a počká, kým všetky ostatné pracovné elementy z rovnakej pracovnej skupiny dosiahnu rovnaký bod v kerneli. Tieto bariéry však nedokážu synchronizovať pracovné elementy z rôznych pracovných skupín. Ak by sme chceli synchronizovať pracovné elementy z rôznych pracovných skupín, musíme čakať, kým sa vykonávanie kernelu úplne dokončí. Poznáme 2 druhy bariér:

- **barrier(CLK_LOCAL_MEM_FENCE);** - zaručuje správne poradie operácii v lokálnej pamäti. Používa sa najmä pri čítaní za zápise do *lokálnej pamäti* označenej identifikátorom *__local*.
- **barrier(CLK_GLOBAL_MEM_FENCE);** - Používa sa najmä pri čítaní za zápise do *globálnej pamäti* označenej identifikátorom *__global*.

V zriedkavých prípadoch, keď chceme použiť obe bariéry naraz, použijeme *barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE)*[9]

4 CUDA VS OPENCL FRAMEWORK

V tejto kapitole uvedieme pozitíva a negatíva oboch softvérových architektúr.

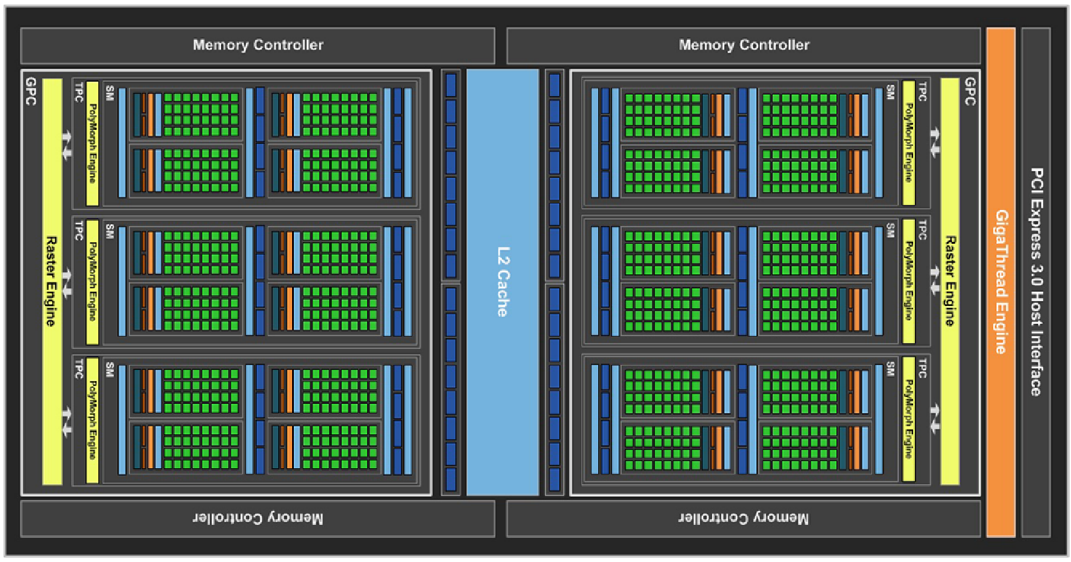
Kód OpenCL je prenosný medzi HW od rôznych výrobcov. Keďže každý výrobca do svojich zariadení vkladá svoju jedinečnú hardvérovú architektúru, je dôležité, aby programátor pochopil komplikovaný koncept zariadenia, pre ktoré aplikáciu vyvíja. To vedie k označeniu OpenCL za náročnejší nástroj pri GPGPU programovaní v porovnaní s CUDA. Ďalšou daňou za vysokú portabilitu OpenCL kódu, je strata optimalizácie pri migrácii kódu medzi platformami. Použitie OpenCL frameworku pre GPGPU sa javí ako dobrá voľba v prípade, že programátor pracuje s aplikáciami, ktoré nepodporujú framework CUDA. Napríklad na prácu s multimédiami, kde je potrebný masívny výpočtový výkon pre rendering, je určite lepšou voľbou OpenCL s grafickou kartou od AMD. Karty od NVIDIA tiež podporujú OpenCL framework, no nie sú tak efektívne ako v prípade AMD kariet.

Proprietárne právo na CUDA softvérovú architektúru má spoločnosť NVIDIA, čo z nej robí vývojársky nástroj aplikovateľný iba na hardvér od NVIDIe. Zjednodušuje niektoré programovacie konštrukcie a mechanizmy, čo ju v porovnaní s OpenCL robí jednoduchším nástrojom pre začiatočníkov v oblasti GPGPU. Pretože CUDA je kompatibilná iba s HW od jediného výrobcu, optimalizácia kódu pre rôzne platformy by mala byť viac deterministická. NVIDIA vedie spoľahlivú vývojársku komunitu špecializovanú práve na CUDA, s ktorou bol aj vývoj tejto práce čiastočne konzultovaný.

V tejto práci sa snažíme pomocou GPGPU optimalizovať časť programu detekujúceho defekty na netkanom textile v reálnom priemysle 24/7. Zo zadania teda plynie, že platforma, na ktorej sa bude optimalizovaný program vykonávať, je nemenná a tým pádom nemusí byť zabezpečená prenositeľnosť kódu medzi platformami od rôznych výrobcov. Je teda jasné, že voľba CUDA SW architektúry je lepšia voľba.

5 VÝBER EXPERIMENTÁLNEHO HW

Tento krok bol konzultovaný vo vývojárskej komunite spoločnosti NVIDIA. Požiadavkou bolo, nájsť vhodný hardvér s paralelnou architektúrou (GPU), ktorý by bol schopný cez PCIe gen3x16 prijať, spracovať a odoslať výsledok späť do réžie CPU s rýchlosťou toku 1,6GB/s. Riadková kamera bude mať frekvenciu snímania 200kHz (viď. 9.1), čo znamená, že do frame grabberu posiela cez sériový interface CoaXPress 200 000 snímkov s rozlíšením 8192pix. Frame grabber (viď. 9.3) slúži ako zásobník dát medzi výpočtovou jednotkou (CPU) a kamerou. Keď sú dáta z kamery pripravené v zásobníku, CPU požiada o doručenie jedného obrazu s rozlíšením 8192x1024 pixelov. Úlohou samotného GPU teda bude predspracovaný obraz prijať z CPU, vykonať optimalizovaný paralelný výpočet na detekciu defektov a následne poslať späť do CPU.



Obr. 5.1: Mikroarchitektúra Pascal zvolenej grafickej karty NVIDIA GeForce GTX 1050Ti.[17]

Na základe týchto požiadavok však nie je možné určiť konkrétnu grafickú kartu. Určite sa zatiaľ neoplatí investovať do high-end Tesla grafických kariet pre dátové centra v priemysle 24/7, ktoré sú predávané za viac ako 50000Kč. V počiatočnom štádiu vývoja ešte nie sme schopní na 100% tvrdiť, že optimalizácia pomocou GPU bude racionálny krok alebo práve naopak. Preto sme sa rozhodli kúpiť lacnejšiu, experimentálnu grafickú kartu **NVIDIA GeForce GTX 1050Ti**. V pomere cena/kvalita táto karta disponuje nadštandardnými výkonom vďaka 4GB globálnej pamäti a modernej mikroarchitektúre **Pascal**, ktorú si popíšeme nižšie. To bol jeden z hlavných dôvodov prečo sme sa rozhodli použiť ju na experimentálne účely.

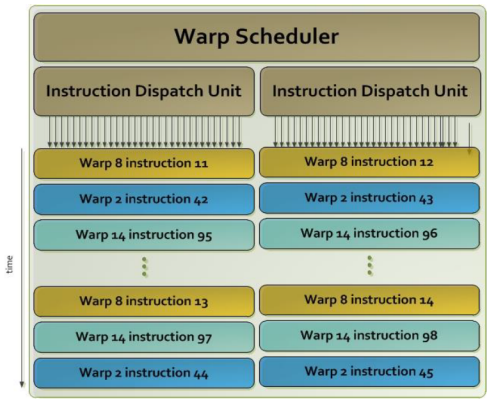
Architektúra moderných grafických kariet je veľmi komplexná, no napriek tomu sa pokúsime popísať zloženie jej častí, ktoré majú určitý súvis s optimalizáciou aktuálneho sekvenčného algoritmu.

Hardvér tejto grafickej karty je zložený z dvoch GPC (angl. Graphics Processing Clusters), šiestich SM a štyroch 32-bitových pamäťových ovládačov. Grafický čip je 128-bitovou pamäťovou zbernicou (4 x 32-bit) spojený s grafickou pamäťou GDDR5 s kapacitou 4GB.

Obr. 5.2: Dispozícia streaming multiprocesoru (SM) grafickej karty NVIDIA GeForce GTX 1050Ti.[12]

Každý SM obsahuje PolyMorph Engine, ktorý umožňuje vertexovým shaderom čítať dáta z textúr, transformovať ich a upravovať perspektívu. Okrem PolyMorph Enginu je súčasťou SM aj 128 CUDA jadier (dohromady 1152 CUDA jadier v GPU), 256kB registrov, 96kB zdielanej pamäti, 48kB L1 medzipamäti a 8 textúrových jednotiek.

Jedna z najdôležitejších technológii, ktorá sa objavuje naprieč všetkými architektúrami od Fermi až po Pascal, je GigaThread ™plánovač vlákien. Tento plánovač, rozdeľuje vláknové bloky medzi jednotlivé SM, čo sa deje na úrovni grafického čipu.



Obr. 5.3: Princíp warpového plánovača v SM.[14]

SM je vysoko paralelný multiprocesor, ktorý plánuje distribuovanie warpov (skupín 32 vlákien) medzi svoje CUDA jadrá. Každý multiprocesor (SM) obsahuje 4 **plánovače warpov** a 8 IDU (angl. Instruction Dispatch Units), čo umožňuje multiprocesoru vykonávať 4 warpy súčasne. Štyri plánovače warpov vyberú 4 warpy a na každom z nich prevedú 2 nezávislé inštrukcie v každom hodinovom cykle (viď obr.5.3)[14],[12].

6 VYBRANÉ OPTIMALIZAČNÉ POSTUPY

Optimalizačných stratégii je vo svete paralelného programovania heterogénnych zariadení neúrekom. V tejto diplomovej práci určite nestihne implementovať a popísať väčšinu z nich. Niektoré z optimalizačných stratégii však majú vyššiu prioritu pred ostatnými. Počas paralelizácie sekvenčného algoritmu sa teda budeme snažiť využiť metódy s najvyšším vplyvom na výkon programu. Tieto metódy budú vysvetlené v nasledujúcich sekciách. Ešte pred tým však uvedieme zoznam najbežnejších optimalizačných metód [9].

1. - **Minimalizácia prenosu pamäti medzi GPU a CPU + preferovanie prístupov do globálnej pamäti oproti prístupom do pamäti hostiteľa**.
2. - Jeden objemný transfer dát je priaznivejší ako mnohé parciálne transfery.
3. - **Splývajúci prístup do globálnej pamäti GPU.**
4. - Použitie zdieľanej pamäti zaručuje až 100x rýchlejší prístup k dátam. Treba sa vyvarovať jej preťaženiu, čo by viedlo k uloženiu časti dát do globálnej pamäti. Slúži na komunikáciu vlákien z jedného vláknového bloku.
5. - **Vyhýbanie sa bankovým konfliktom, ktoré zvyšujú počet sekvenčných inštrukcii vykonaných nad warpom.**
6. - Počet paralelne bežiacich blokov by mal byť minimálne rovný počtu SM.
7. - Počet vlákien by mal byť násobkom veľkosti warpu.
8. - **Zabránenie divergencie vlákien v rámci jedného warpu.**
9. - Pokiaľ je to možné, vyhýbať sa synchronizácii vlákien.
10. - Low-level optimalizácia na úrovni inštrukcii

6.1 Rozvetvovanie a divergencia

Každá programová inštrukcia deliaca vykonávanie kódu do viacerých vetiev môže výrazne ovplyvniť výkonnosť aplikácie. Takýmito inštrukciami sú napríklad *(switch, if, do, for, while)*. V kapitole 5 sme spomínali, že multiprocesory (SM) v modernejších grafických kartách sú schopné vykonávať inštrukcie až na 4 warpoch súčasne. Divergencia a rozvetvovanie bohužiaľ prináša zvyšovanie počtu inštrukcii na jednom warpe.

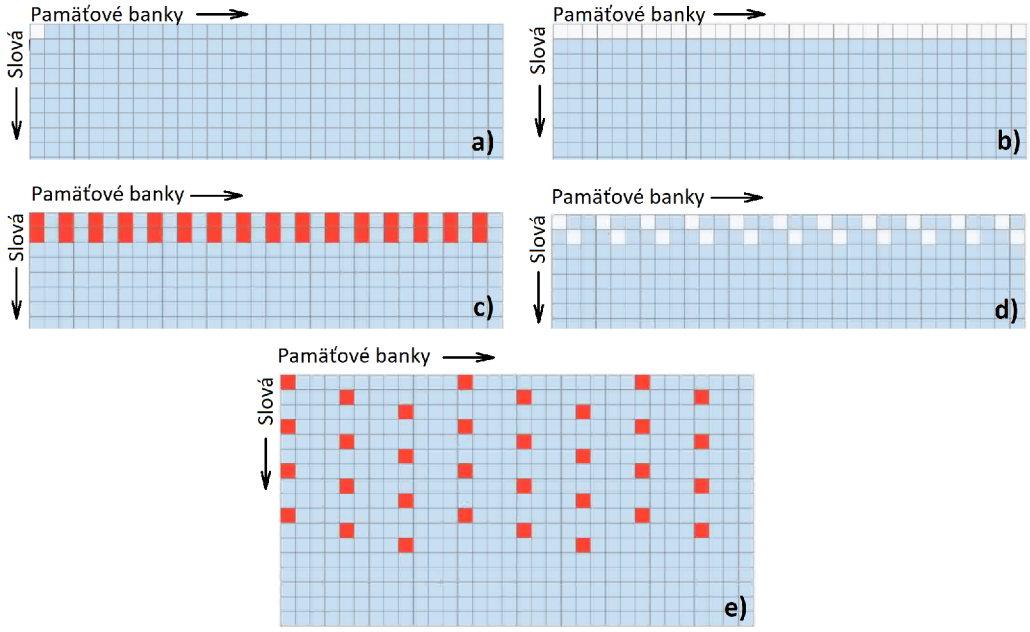
Nižšie navrhnutá implementácia kernelu (viď. 7), ktorú sme navrhli v tejto práci sa však bez rozvetvenia toku programu nezaobíde. Pretože spôsob distribuovania warpov medzi multiprocesory (SM) je deterministický, je možne každú inštrukciu riadiacu tok programu napísať tak, aby sa minimalizoval počet divergentných warpov. Znamenalo by to, že všetkých 32 vlákien z jedného warpu, ktoré sú aktuálne priradené multiprocesoru, budú pokračovať v rovnakej vetve programu [7], [9].

6.2 Bankové konflikty v zdieľanej pamäti

Zdieľaná pamäť je rozdelená na rovnako veľké pamäťové moduly - **banky**. Jednotlivé vlákna môžu za určitých podmienok naraz pristupovať k N adresám, ktoré odpovedajú N rôznym pamäťovým bankám. To znamená, že rýchlosť prístupu bude *N-krát* rýchlejšia ako sekvenčný prístup do jednotlivých bánk.

Ak chce niekoľko vlákien pristupovať na adresu v rovnakej pamäťovej banke, dochádza k **bankovému konfliktu** a tento prístup bude serializovaný. Hardware totiž rozdelí konfliktné prístupy na niekoľko bezkonfliktových, ktoré sa vykonajú sekvenčne za sebou. To vedie k zníženiu výkonu.

Sú tu však dve výnimky - **broadcast a multicast**. Prvá z nich nastáva v prípade, že **všetky** vlákna z warpu pristupujú na adresu v rovnakej banke. Hodnota na tejto adrese bude prečítaná iba raz a následne **broadcastovaná** medzi všetky vlákna z warpu. **Multicast** je zredukovaná verzia broadcastu a nastáva práve vtedy, ak iba niekoľko vlákien z jedného warpu pristupuje do rovnakej banky.



Obr. 6.1: Bankové konflikty.

Zdieľaná pamäť obsahuje 32 bánk a adresový priestor v každej z nich je zarovnaný na 32-bitové slová. Priepustnosť zdieľanej pamäti je 32-bitov na jednu pamäťovú banku za jeden hodinový cyklus. Jeden z najprirodzenejších spôsobov prístupu vlákien do zdieľanej pamäti je pomocou indexu **threadIdx**. Predpokladajme, že pracujeme s polom slov (integers) uloženým v zdieľanej pamäti. K jednotlivým prvkom pola budeme pristupovať prostredníctvom násobkov indexu threadIdx.

Na obrázku 6.1 sú zobrazené rôzne spôsoby prístupov do zdieľanej pamäti:

(a) Všetky vlákna z jedného warpu pristupujú do jednej banky, takže sa jedná o **broadcast**. K tejto situácii dôjde napríklad po vykonaní príkazov *arr[threadIdx.x*0], arr[12], arr[blockIdx.x*3].*
(b) V tomto prípade, každé vlákno z jedného warpu pristupuje k slovu na pozícii *threadIdx.x*. Napríklad *arr[threadIdx.x]*. Zabezpečený rovnako rýchly prístup ako v prípade (a).
(c) Vlákna z warpu pristupujú k adresám, ktoré sú 2-násobkom ich threadIdx.x. *arr[threadIdx.x*2]*. Tento prístup však bude **pomalší** ako predchádzajúce, pretože vlákna pristupujú k 2 hodnotám z každej druhej banky.
(d) Vlákna z warpu pristupujú k adresám, ktoré sú 3-násobkom ich threadIdx.x. *arr[threadIdx.x*3]*. Tento prístup bude rovnako rýchly ako v prípade (a) a (b). Nedochádza k bankovým konfliktom.
(e) Vlákna z warpu pristupujú k adresám, ktoré sú 12-násobkom ich threadIdx.x. *arr[threadIdx.x*12]*. Tento prístup je **veľmi pomalý**, pretože sa jedná, pretože vlákna pristupujú k 4 hodnotám každej 4. banky.

.

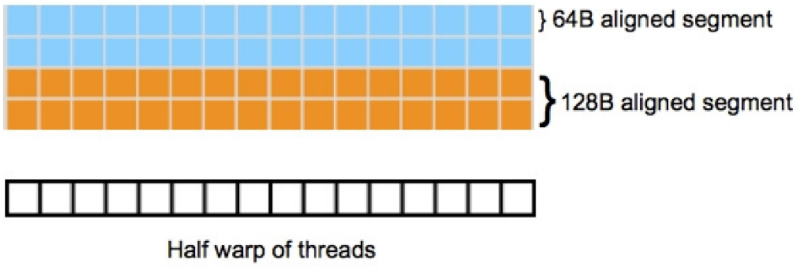
Našťastie, v modernejších grafických kartách nám HW pomáha skryť latenciu spôsobenú bankovými konfliktmi. Ak na SM beží veľa vlákien naraz, plánovač prítomný v SM je schopný v prípade bankového konfliktu prepnúť na iný warp.

6.3 Splývajúci prístup do globálnej pamäti GPU

Jedna z najdôležitejších vecí, na ktoré treba dbať pri optimalizácii kódu určeného pre CUDA GPU architektúru je zaručenie **splývania prístupov do globálnej pamäti DRAM**. GPU to umožňuje prostredníctvom vlákien z jedného **polovičného warpu** - skupiny 16 sekvenčných vlákien. Týmto spôsobom je možné zredukovať počet pamäťových transakcií na minimum. Globálna pamäť je usporiadaná do segmentov s veľkosťou zarovnanou na 16 a 32 slov. Na obrázku 6.2 je uvedený príklad, kde sú znázornené dva 128 bajtové segmenty (odlíšené farebne). Dole na obrázku je tzv. **polovičný warp**, ktorého vlákna pristupujú k 16 32-bitovým slovám spodnej časti segmentu.

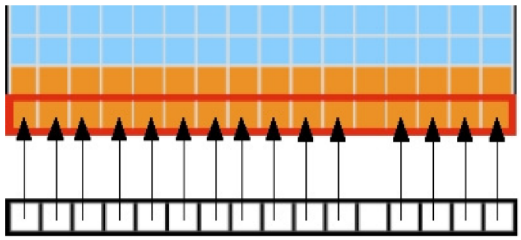
Aby sme prístup do globálnej pamäti mohli považovať za splývajúci, musia vlákna z polovičného warpu pristupovať k:

- 8-bitovým slovám 32 bajtového segmentu alebo
- 16-bitovým slovám 64 bajtového segmentu, alebo
- 32-bitovým/64-bitovým slovám 128 bajtového segmentu.



Obr. 6.2: Dva 128-bajtové segmenty v globálnej pamäti.[9]

Prvý koncept splývajúceho prístupu je ilustrovaný na obrázku 6.3. K-té vlákno pristupuje ku k-tému slovu v segmente. Ide o **prístup do sekvenčnej pamäti zarovnanej so segmentom**. Všetky vlákna sa však nemusia podieľať na prístupe. Ak vlákna pristupujú k 16 32-bitovým slovám, vykoná sa jediná 64 bajtová SIMD transakcia, a to aj v prípade, že by vlákna pristupovali zmiešane k rôznym slovám tohto segmentu.

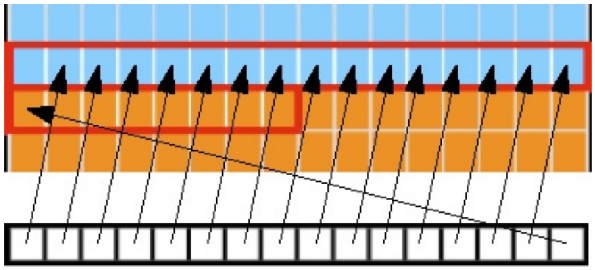


Obr. 6.3: Prístup do sekvenčnej pamäti **zarovnanej** so segmentom.[9]

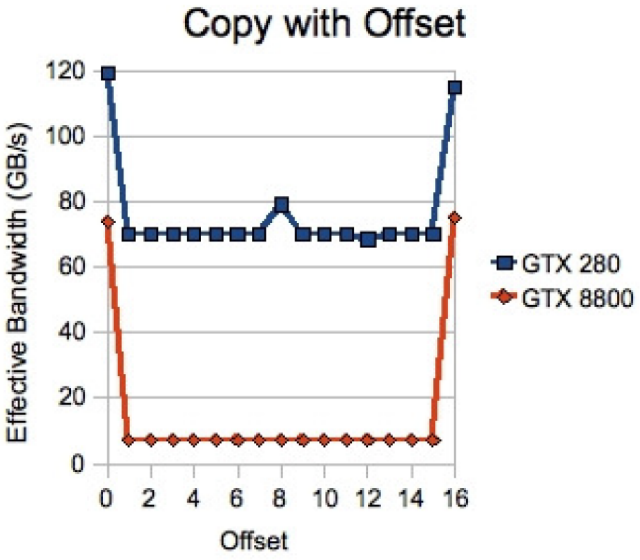
Druhý koncept ilustruje situáciu, v ktorej vlákna polovičného warpu síce pristupujú do sekvenčnej pamäti, no táto pamäť nie je zarovnaná so segmentom. Hovoríme o **prístupe s offsetom**. Môžu nastať dva prípady. Ak adresový priestor, do ktorého jednotlivé vlákna pristupujú, odpovedá 128 bajtovému segmentu, tak sa vykoná jediná 128-bajtová SIMD inštrukcia (viď obr. 6.3). Ak by vlákna pristupovali k adresám dvoch rôznych 128-bajtových segmentov, ako je uvedené na obr. 6.4, tak by sa musela vykonať jedna 32-bajtová a jedna 64-bajtová inštrukcia.

Ako dôkaz drastického zníženia výkonu v prípade **prístupu do pamäti s offsetom** bol prevedený test na grafických kartách GTX280 a GTX8800. Výsledky sú znázornené na obrázku 6.5. Z grafu vyplýva, že prístup vlákien do globálnej pamäti s offsetom 0 a 16 vyžaduje iba jednu inštrukciu. Naopak, offset väčší ako 0 a menší ako 16 spôsobí v prípade GPU GTX 8800 až 8-násobnú degradáciu výkonu.

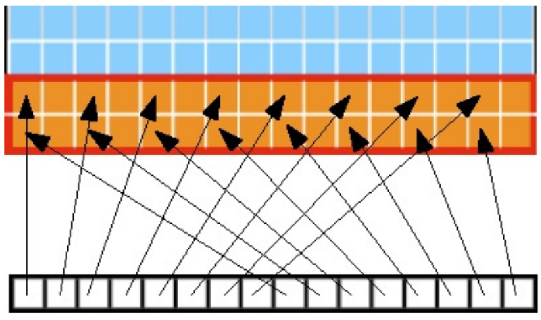
Ďalším konceptom je **prístup do pamäti s rôznym krokom**. Na obrázku 6.6 vlákna polovičného warpu pristupujú do pamäťového segmentu s krokom 2.



Obr. 6.4: Prístup do sekvenčnej pamäti **nezarovnanej** so segmentom.[9]



Obr. 6.5: Závislosť priepustnosti na zvolenom offsete.[9]



Obr. 6.6: Prístup do sekvenčnej pamäti s rôznym krokom.[9]

Vlákna budú pristupovať na každú druhú pozíciu v pamäťovom segmente. V tomto prípade sa síce vykoná iba jedna 128-bajtová inštrukcia, no na obrázku je

vidieť, že veľa adries v segmente ostane nevyužitých. Naviac, s rastúcou hodnotou kroku, by sa exponenciálne znižoval výkon, pretože vlákna by z jedného polovičného warpu pristupovali do rôznych segmentov.[9], [7]

6.4 Optimalizácia dátového transferu

Pri optimalizácii sériového kódu prevodom na paralelný kód, je potrebné dbať na fakt, že paralelný kód nebude vykonávaný na CPU, ale na určitom zariadení s paralelnou architektúrou. V našom prípade to bude experimentálna grafická karta **NVIDIA GeForce GTX 1050Ti**. Aby sme na obraze s rozlíšením 8192x1024 pixelov, mohli pomocou paralelného algoritmu detekovať defekty, musíme najskôr všetky obrazové dáta skopírovať z pamäti CPU do globálnej pamäti GPU. Našim cieľom bude poskytnúť tieto dáta grafickej karte tak, aby sme zachovali maximálnu možnú priepustnosť dátových transferov. To môžeme dosiahnuť využívaním pamätí s rýchlym prístupom. V tejto kapitole budeme teda diskutovať, rôzne spôsoby optimalizácie spomínaných dátových transferov.[7],[9]

Priepustnosť medzi globálnou pamäťou GTX 1050Ti a jej on-chip pamäťou (**112 GB/sec**) je omnoho vyššia ako priepustnosť medzi hostiteľskou pamäťou RAM a globálnou pamäťou GTX 1050Ti (ak počítame so zbernicou *PCI Express x16 Gen2* tak **8GB/sec**). Z toho vyplýva, že potrebujeme čo najviac obmedziť transfery dát medzi CPU a GPU. Všetky pomocné premenné by teda mali byť vytvorené/ odstránené už v pamäti zariadenia bez toho, aby sme ich museli kopírovať medzi pamäťami GPU a CPU.

V našom prípade však k takémuto kopírovaniu musí dôjsť. Každý jeden obrázok textilu je potrebné nakopírovať do pamäti zariadenia na to, aby mohol byť podrobený paralelnému algoritmu. Priaznivé by určite bolo kopírovať všetky tieto dáta naraz ako celok, než vykonávať individuálne prístupy vlákien do pamäti s nízkou priepustnosťou.

Ako bolo spomenuté v podkapitole 2.8, spôsob ako zrýchliť prenos medzi CPU a GPU je používaním *pamäti s blokovaným stránkovaním*. Pretože zabezpečenie čo najrýchlejšieho transferu dát je v tejto aplikácii kritický, otestujeme a porovnáme rôzne druhy, a z výsledkov určíme, ktorý z nich by bol najpriaznivejší.

V navrhnutej optimalizovanej aplikácii budeme okrem iného pracovať aj s pamäťovými zásobníkmi alokovanými funkciou *malloc* na CPU. Jednotlivé zásobníky si ďalej zadefinujeme.

- **image_data** - je to ukazovateľ na zásobník s 8192x1024 8-bitovými hodnotami dátového typu *unsigned char*, ktoré reprezentujú jednu snímku textilu.

- **defect_limits** - je to ukazovateľ na zásobník o veľkosti 40x1024 hodnôt dátového typu (integer). Počet riadkov tohto zásobníku odpovedá riadkom snímku (výške snímku). Napríklad, v prípade detekovanej chyby na súradnici y=123 v zásobníku *image_data*, sa na riadok 123 v tomto zásobníku uloží hodnota súradnice x1 začiatku chyby a hodnota súradnice x2 konca chyby.
- **counter_array** - ukazovateľ na pole s 1024 hodnotami dátového typu *integer*. Dĺžka tohto pola tiež odpovedá výške snímku. Jednotlivé hodnoty v tomto poli slúžia ako počítadlo chýb detekovaných na určitých riadkoch v snímku. Napríklad, ak na súradnici y=123 bola detekovaná chyba, tak sa hodnota na 123. mieste v tomto poli inkrementuje o hodnotu 2 (hodnota x1 a x2). Vďaka tomuto polu budú vlákná vykonávajúce kernel vedieť, na akú pozíciu v zásobníku *d_defect_limits* majú zapisovať súradnice x1 a x2.

Obsah vyššie popísaných zásobníkov sa pri každom analyzovanom snímku skopíruje do rovnako veľkých pamäťových regiónov v globálnej pamäti GPU. K týmto zaśobníkom budú mať prístup všetky vlákna vykonávajúce kernel. Aby sme odlíšili zásobníky na GPU od zásobníkov na CPU, pomenovali sme ich rovnako s tým, že sme k ich menám pridali predponu **d_** ako *device*.

- **d_image_data** - pamäť pre obrazové dáta (ekvivalent **image_data**)
- **d_defect_limits** - pamäť pre ukladanie súradníc začiatkov a koncov chýb na jednotlivých riadkoch (ekvivalent **defect_limits**)
- **d_counter_array** - počítadlo zapísaných súradníc do pamäti *d_defect_limits*. (ekvivalent **counter_array**).

Pretože obe grafické karty z dostupných systémov *HWConfig GT755M* a *HWConfig GTX1050Ti* sú od spoločnosti NVIDIA, obe prioritne podporujú softvérovú architektúru CUDA. To by teoreticky malo znamenať, že optimalizáciou pomocou CUDA by sme mohli dosiahnuť lepšieho výkonu. Práve preto jednotlivé metódy transferu popíšeme na testovacom CUDA programe. Výkonnostný test však prevedieme s oboma nástrojmi (CUDA, OpenCL) na oboch systémoch HWConfig GT755M a HWConfig GTX1050Ti. V priloženom CD budú dostupné zdrojové kódy pre testovanie oboch frameworkov.

6.4.1 Kopírovaná stránkovaná pamäť

V tejto sekcii popíšeme implementáciu jedného z najprimitívnejších spôsobov kopírovania pamäti medzi hostiteľom a zariadením. V prvom kroku alokujeme na GPU 3 zásobníky **d_image_data**, **d_defect_limits**, **d_counter_array**. Rovnako musíme na CPU alokovať miesto pre **defect_limits** a **counter_array**. Zásobník

image_data alokovať nemusíme, pretože obrazové dáta sú už zapísané v RAM na CPU a ukazovateľ na tieto dáta nazveme *input*.

```
defect_limits = (int *)malloc(defect_limits_size);
counter_array = (int *)malloc(counter_array_size);

HANDLE_ERROR(cudaMalloc(&d_defect_limits, defect_limits_size));
HANDLE_ERROR(cudaMalloc(&d_counter_array, counter_array_size));
HANDLE_ERROR(cudaMalloc(&d_image_data, size_in));
```

Táto časť kódu bude statická a volaná iba raz pri inicializácii programu. Tieto zásobníky nie je potrebné alokovať vždy, keď CPU dostane k spracovaniu novú snímku. Teraz pristúpime k časti programu, ktorá sa vykoná vždy, keď CPU dostane k spracovaniu ďalšiu snímku. Pomocou funkcie *memset* vynulujeme všetky pomocné zásobníky **defect_limits** a **counter_array** a funkciou *cudaMemcpy* nakopírujeme ich obsah do alokovaných regiónov v pamäti GPU.

```
HANDLE_ERROR(cudaMemcpy(d_image_data, input,
                        size_in, cudaMemcpyHostToDevice));
HANDLE_ERROR(cudaMemcpy(d_counter_array, counter_array,
                        counter_array_size, cudaMemcpyHostToDevice));
HANDLE_ERROR(cudaMemcpy(d_defect_limits, defect_limits,
                        defect_limits_size, cudaMemcpyHostToDevice));
```

Teraz zavoláme kernel s potrebnými GPU zásobníkmi medzi jeho parametrami.

```
cuFindDefects_kernel << < numBlocks, numThreads, localMemSize >> >
                    (d_image_data, width, height, d_defect_limits,
                    d_counter_array, limitDown, limitUp);
```

A na záver skopírujeme dáta späť do pamäti CPU.

```
HANDLE_ERROR(cudaMemcpy(defect_limits, d_defect_limits,
                        defect_limits_size, cudaMemcpyDeviceToHost));
```

Tento spôsob transferu dát by mal byť teoreticky najpomalší. Jeho rýchlosť však nezávisí len na hardvérovom vybavení GPU. Nezanedbateľný vplyv na transfer dát má aj CPU, pevný disk, systémové zbernice a rovnako aj operačný systém. Volaním funkcie *cudaMemcpy*, CUDA ovládač musí najskôr prekopírovať dáta alokované funkciou *malloc* zo stránkovanej pamäti (pevný disk) do oblasti pamäti s blokovaným stránkovaním (fyzická pamäť RAM). Až potom sa môže zahájiť DMA prenos dát do globálnej pamäti GPU.

Tab. 6.1: Meranie rýchlosti prenosu pre *kopírovanú stránkovanú pamäť*. (milisekundy)

CPU -> GPU	CUDA	OpenCL
HWConfig GT755M	2,7597	2,9189
HWConfig GTX1050Ti	2,7062	3,1355

6.4.2 Kopírovaná pamäť s blokovaným stránkovaním

Hneď po spustení aplikácie alokujeme funkciou *cudaHostMalloc* 2 pamäťové zásobníky **defect_limits**, **counter_array**. Na rozdiel od predchádzajúcej implementácie 6.4.1, sme v tomto prípade alokovali pamäť na CPU s blokovaným stránkovaním. Ďalším krokom je alokácia 3 pamäťových zásobníkov **d_image_data**, **d_defect_limits**, **d_counter_array** v pamäti GPU funkciou *cudaMalloc*. Táto alokácia pamäťového priestoru sa vykoná iba raz v priebehu programu.

```
HANDLE_ERROR(cudaHostAlloc((void **)&defect_limits, defect_limits_size, 0));
HANDLE_ERROR(cudaHostAlloc((void **)&counter_array, counter_array_size, 0));

HANDLE_ERROR(cudaMalloc(&d_defect_limits, defect_limits_size));
HANDLE_ERROR(cudaMalloc(&d_counter_array, counter_array_size));
HANDLE_ERROR(cudaMalloc(&d_image_data, size_in));
```

Ďalej môžeme pristúpiť k časti kódu, ktorá sa vykoná pre každú snímku zvlášť. Rovnako ako v sekcii 6.4.1, aj tu musíme v prvom rade funkciou *memset* vynulovať zásobníky **defect_limits** a **counter_array**. *Input* je ukazovateľ na obrazové dáta uložené niekde v stránkovanej pamäti CPU. Funkcia *cudaHostRegister* zablokuje stránkovanie pre oblasť pamäti, na ktorú ukazuje *input*. Až teraz môžeme volať funkcie *cudaMemcpyAsync*, ktoré prekopírujú zásobníky z CPU do GPU.

```
HANDLE_ERROR(cudaHostRegister(input, size_in, 0));
HANDLE_ERROR(cudaMemcpyAsync(d_image_data, input, size_in,
                             cudaMemcpyHostToDevice, 0));
HANDLE_ERROR(cudaMemcpyAsync(d_counter_array, counter_array,
                             counter_array_size, cudaMemcpyHostToDevice, 0));
HANDLE_ERROR(cudaMemcpyAsync(d_defect_limits, defect_limits,
                             defect_limits_size, cudaMemcpyHostToDevice, 0));
```

Akonáhle sú zásobníky skopírované do pamäti GPU, môžeme zavolať zavolať kernel.

```
cuFindDefects_kernel << < numBlocks, numThreads, localMemSize >> >
                    (d_image_data, width, height, d_defect_limits,
                    d_counter_array, limitDown, limitUp);
```

Po spracovaní obrazu skopírujeme obsah zásobníku **d_defect_limits** späť do pamäti CPU - **defect_limits**. Nesmieme zabudnúť na to, že zásobník **defect_limits** má stále zablokované stránkovanie, ktoré znovu povolíme funkciou *cudaHostUnregister*.

```
HANDLE_ERROR(cudaHostUnregister(input));
HANDLE_ERROR(cudaMemcpyAsync(defect_limits, d_defect_limits,
                             defect_limits_size, cudaMemcpyDeviceToHost, 0));
```

Implementácia zaručí okamžitý DMA prenos hneď po volaní funkcie *cudaMemcpyAsync*, bez potreby kopírovať zásobníky do pamäti s blokovaným stránkovaním. **Tento spôsob transferu dát, by mal byť pre optimalizačné účeli najvhodnejší.**

Tab. 6.2: Meranie rýchlosti prenosu pre *kopírovanej pamäti s blokovaným stránkovaním.* (milisekundy)

CPU -> GPU	CUDA	OpenCL
HWConfig GT755M	1,7729	2,7675
HWConfig GTX1050Ti	2,0122	2,7590

6.4.3 Nekopírovaná pamäť s blokovaným stránkovaním

Rovnako ako v predchádzajúcej implementácii 6.4.2 alokujeme funkciou *cudaHostMalloc* pamäť s blokovaným stránkovaním pre zásobníky **defect_limits** a **counter_array**. Je tu však jeden rozdiel, a to vo flagu *cudaHostAllocMapped*. Týmto flagom oznamuje funkcii *cudaHostMalloc*, že chceme alokované zásobníky v pamäti CPU namapovať do adresového priestoru pamäti GPU. Funkcia *cudaHostGetDevicePointer* spraví z premenných **d_defect_limits** a **d_counter_array** ukazovatele na túto namapovanú pamäť v GPU. Táto alokácia a mapovanie pamäťového priestoru sa vykoná iba raz v priebehu programu.

```
HANDLE_ERROR(cudaHostAlloc((void **)&defect_limits, defect_limits_size,
                           cudaHostAllocMapped));
HANDLE_ERROR(cudaHostAlloc((void **)&counter_array, counter_array_size,
                           cudaHostAllocMapped));

HANDLE_ERROR(cudaHostGetDevicePointer(&d_defect_limits, defect_limits, 0));
HANDLE_ERROR(cudaHostGetDevicePointer(&d_counter_array, counter_array, 0));
```

Funkciou *memset* vynulujeme zásobníky **defect__limits** a **counter__array**. Rovnako aj adresový priestor s obrazovými dátami, na ktorý ukazuje ukazovateľ *input* musíme namapovať do pamäti GPU a získať ukazovateľ na tieto dáta. Na to slúži už spomínaná funkcia *cudaHostRegister* s nastaveným flagom *cudaHostRegisterMapped* a k obdržaniu ukazovateľa použijeme funkciu *cudaHostGetDevicePointer*.

```
HANDLE_ERROR(cudaHostRegister(input, size_in, cudaHostRegisterMapped));
HANDLE_ERROR(cudaHostGetDevicePointer(&d_image_data, input, 0));
```

Pretože sa jedná o **nekopírovanú pamäť**, nie je potrebné kopírovať dáta z pamäti CPU do GPU a môžme priamo zavolať kernel.

```
cuFindDefects_kernel << < numBlocks, numThreads, localMemSize >> >
                        (d_image_data, width, height, d_defect_limits,
                        d_counter_array, limitDown, limitUp);
```

Po ukončení kernelu stačí pomocou *cudaHostUnregister* odblokovať stránkovanie pre *input*.

```
HANDLE_ERROR(cudaHostUnregister(input));
```

Zásobník s výsledkami, na ktorý ukazuje **defect__limits** môže CPU ďalej spracovávať bez potreby kopírovania späť do pamäti CPU. Napriek tejto výhode, je používanie **nekopírovanej pamäti s blokovaným stránkovaním** v praxi obmedzené. Pamäť s takýmito atribútmi totiž nie je zachytávaná (angl. cached) v rýchlych vyrovnávacích pamätiach GPU, preto by mali vlákna čítať a zapisovať z tejto pamäti iba raz a ich prístup do nej, by mal byť splývajúci (viď. Splývajúci prístup do globálnej pamäti GPU). Mapovanie pamäti je teda využiteľné len v prípade že:

- zariadenie je napríklad integrovaná grafická karta, ktorá zdieľa pamäť RAM s CPU.
- naraz načítame veľký objem dát (stovky MB až GB), s ktorými prevádzame množstvo výpočtov a počas toho potrebujeme skryť latenciu prístupov do pamäti.

- hostiteľ potrebuje pristupovať k dátam počas behu kernelu.
- GPU nemá dostatok globálnej pamäti na uloženie dát.

Tab. 6.3: Meranie rýchlosti prenosu pre *nekopírovanej pamäti s blokovaným stránkovaním.* (milisekundy)

CPU -> GPU	CUDA	OpenCL
`HWConfig GT755M`	0,3987	4,2559
`HWConfig GTX1050Ti`	0,7079	5,0380

6.5 Zhrnutie

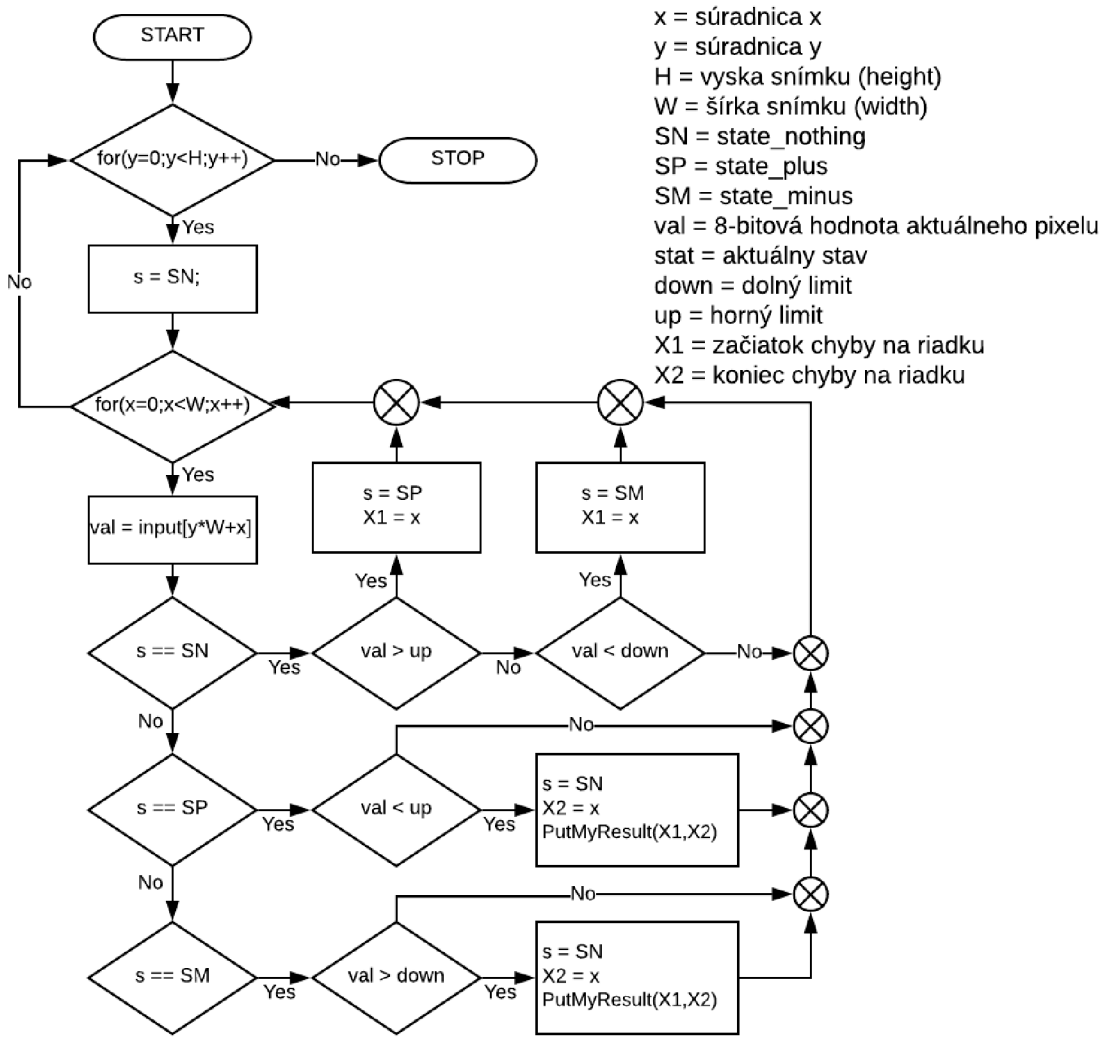
Z nameraných rýchlostí rôznych metód transferu pamäti medzi GPU a CPU, môžeme s istotou prehlásiť, že najoptimálnejší bude prenos **kopírovanej pamäti s blokovaným stránkovaním**. Relatívne dobré výsledky priniesla aj metóda mapovania pamäti do pamäťového priestoru GPU použitím CUDA frameworku. Použitie tejto metódy by však malo za následok spomalenie vykonávania kernelu, pretože jednotlivé vlákna by museli s vysokou latenciou pristupovať do pamäti CPU.

Graficky znázornené výsledky a rôzne druhy porovnaní jednotlivých pamäťových transferov sú súčasťou príloh - *PRÍLOHA A, PRÍLOHA B.*

7 NÁVRH A OPTIMALIZÁCIA KERNELU

V tejto kapitole detailnejšie rozoberieme štruktúru pôvodnej časti sekvenčného kódu, ktorý bude predmetom optimalizácie. V druhej sekcii popíšeme návrh paralelného algoritmu, resp. kernelu pre paralelnú architektúru, ktorého výstupy budú ekvivalentné sekvenčnej časti pôvodného kódu.

7.1 Rozbor a časovanie pôvodného algoritmu



Obr. 7.1: Zjednodušený diagram pôvodného algoritmu bežiaceho na CPU.

Aktuálny algoritmus detekujúci defekty na 8bpp snímkoch textilu prichádzajúcich z frame grabberu je veľmi komplexný. Jeho celková optimalizácia prevodom

na algoritmus paralelný, by viedla k dlhodobému vývoju, na ktorom by sa musel podieľať tím programátorov. Vzhľadom ku konštrukcii aktuálneho programu, nie je ani nutné robiť celkovú optimalizáciu kódu. Úlohou tejto práce je optimalizovať časť kódu, ktorá zisťuje či snímka vôbec nejaké defekty obsahuje. Spolu s predspracovaním obrazu to je najfrekventovanejšia časť kódu, ktorá sa vykoná na každej jednej snímke.

Jedna sa v podstate o stavový automat v dvoch vnorených *for* cykloch (viď obr. 7.1). Jeden cyklus pre iteráciu riadkov a druhý pre iteráciu stĺpcov obrazu. Automat má tri stavy - **state_minus**, **state_plus** a **state_nothing**. Každá snímka je spracovávaná riadok po riadku. Ak na nejakom riadku narazíme na pixel s hodnotu vyššou ako stanovený *limit_up* alebo nižšou ako limit **limit_down**, zapíše do premennej **X1** aktuálnu súradnicu x **začiatku chyby** a zmení aktuálny stav na **state_plus** resp. **state_minus**. *For* cyklus ďalej pokračuje až kým nenarazí na pixel, ktorého hodnota leží v priemere medzi spodným limitom (**limit_down**) a vrchným limitom (**limit_up**). To znamená, že bol detekovaný **koniec chyby** na tomto riadku. Do premennej **X2** sa znovu zapíše aktuálna súradnica x a stav sa zmení na **state_nothing**. Akonáhle obdržíme obe hodnoty X1 a X2, predajú sa funkcii *PutMyResult*. Táto funkcia prijíma dvojice **X1**, **X2** a prostredníctvom zhlukovej analýzy, a histogramov vytvára v pamäti pole štruktúr, ktoré zahŕňajú vlastnosti jednotlivých chýb detekovaných v obraze. Na základe týchto štruktúr je ďalší algoritmus schopný tieto chyby klasifikovať. Funkcia *PutMyResult* však nie je predmetom nášho záujmu, takže časovanie tohto algoritmu bude prevedené bez použitia tejto funkcie.

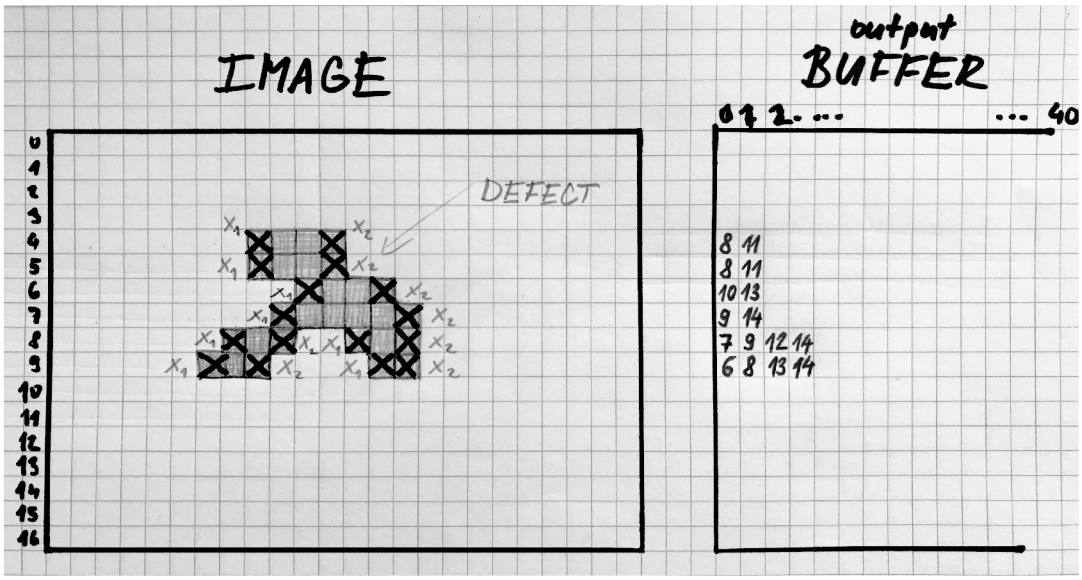
V tejto práci sa budeme snažiť paralelizovať vyššie popísaný stavový automat tak, aby po spustení na paralelnom hardvéri (GPU) vykazoval maximálne možné zrýchlenie. Na obrázku **??** sú zmerané časy spracovania 10 snímkov textilu s rozlíšením 8192x1024 pixelov 8bpp vrátane ich celkového priemeru. Na časovanie sme použili obe dostupné PC zostavy - **HWConfig GT755M** a **HWConfig GTX1050Ti**. V porovnaní vidíme, že spracovanie na CPU *Intel(R) Core(TM) i7-4700MQ CPU@2.40GHz* je až o **12ms** rýchlejšie ako v prípade *Intel(R) Core(TM) i5-450 CPU@3.20GHz*.

7.2 Návrh a optimalizácia kernelu pre GPU

.

Navrhnúť a optimalizovať aplikáciu pre GPU, ktorá by detekovala defekty na snímkoch textilu s tým, že programátor si I/O parametre môže zvoliť sám, je pomerne jednoduchá záležitosť. CUDA aj OpenCL ponúkaju veľa rôznorodých knižníc pre image processing, ktoré by sa s ľahkosťou dali uplatniť pri vývoji aplikácie tohto

druhu. V tejto práci to však také jednoduché nebolo. Ako sme už spomenuli, aktuálny algoritmus je veľmi komplexný a našou úlohou je optimalizovať časť programu popísaného diagramom 7.1. Aby nemuselo dôjsť k modifikácii celého pôvodného sériového kódu, sme sa po konzultácii s firmou dohodli, že po spracovaní obrazu na GPU, bude na výstupe dvojrozmerné pole (pamäťový zásobník) o veľkosti 40x1024 prvkov dátového typu *int*. 1024 riadkov tohto zásobníku odpovedá počtu riadkov spracovávaných snímkov s rozlíšením 8192x1024. Maximálny povolený počet chýb na riadok je aktuálne stanovený na 20, kde každá z nich je na jednom riadku reprezentovaná dvojicou x-ových súradníc **X1** a **X2**. Z toho vyplýva, že počet stĺpcov vo výstupnom zásobníku musí byť 2 x 20 = 40.



Obr. 7.2: Pravidlo zápisu súradníc X1 a X2 do výstupného zásobníku.

Napríklad, ak program detekuje chybu v snímke na súradnici y = 123 (riadok 123), tak sa na 123. riadok 0. stĺpec vo výstupnom zásobníku zapíše súradnica **X1** a na 123. riadok 1. stĺpec sa zapíše súradnica **X2**. V prípade detekovaia ďalších chýb na tom istom riadku sa dvojice x-ových súradníc zapisujú sekvenčne do nasledujúcich stĺpcov výstupného zásobníku. Celý proces zápisu výsledkov do výstupného bufferu je ilustrovaný na obrázku 7.2.

Pri písaní kernelu pre OpenCL a CUDA máme výhodu v tom, že kód OpenCL kernelu a CUDA kernelu sú takmer totožné s malými odchýlkami, na ktoré vždy upozorníme v nasledujúcom texte. Ešte pred tým, ako začneme písať kernel, musíme určiť **počet vlákien** v jednom vláknovom bloku a počet **vláknových blokov**, s ktorými sa CUDA kernel bude volať. Znamená to, určiť spôsob rozdelenia celej snímky 8192x1024 medzi vláknové bloky, ktoré sa budú paralelne spracovávať na jednot-

livých SM v GPU. Je jasné, že budeme pracovať s 2-dimenzionálnym priestorom, takže pri výbere rozmeru **vláknového bloku** musíme uviesť jeho výšku aj šírku. Z kapitoly 5 vieme, že začínajúc mikroarchitektúrou Kepler, všetky SM spracovávajú vláknové bloky vykonávaním warpov - skupín 32 vlákien naraz. To znamená, že vhodnou voľbou je násobok 32 vlákien v jednom bloku. Zároveň musíme dbať na to, aby sme neprekročili architektúrou stanovený limit vlákien na jeden blok. Grafické karty v oboch dostupných PC zostavách majú stanovený limit na 1024 vlákien na jeden blok. Veľkosť bloku určíme na *32 x 32 = 1024* vlákien. Ďalej definujeme počet vláknových blokov na šírku a výšku obrazu. Implementácia v prípade CUDA:

```
dim3 numThreads = { 32, 32 };
dim3 numBlocks = { 8192 / 32, 1024 / 32 };
```

V OpenCL tieto rozmery zapíšeme do pola s troma prvkami, ktoré musíme predať ako parameter funkcii volajúcej kernel *clEnqueueNDRangeKernel*. Do pola *local* zadávame rozmery x,y vláknového bloku (rovnako ako v CUDA *numThreads*). Pole *global* definuje rozmery celého NDRange priestoru.

```
const size_t local[3] = { 32, 32, 0 };
const size_t global[3] = { 8192, 1024, 0 };
```

Vzhľadom k návrhu riešenia, bude kernel pristupovať do 3 pamäťovými zásobníkov, s ktorými sme sa bližšie zoznámili už v podkapitole 6.4. Jedná sa o **d_image_data**, **d_defect_limits** a **d_counter_array**. Pre pripomenutie, sú to ukazovatele na pamäťové zásobníky buď v globálnej pamäti GPU, alebo v pamäti CPU, čo závisí od druhu použitého transferu dát CPU->GPU. Tieto zásobníky predáme kernelu prostredníctvom vstupných parametrov. Prototyp CUDA kernelu vyzerá nasledovne:

```
__global__ void cuFindDefects_kernel(unsigned char *input,  //d_image_data
                                     unsigned int width,
                                     unsigned int heigth,
                                     int *output,           //d_defect_limits
                                     int *counter,          //d_counter_array
                                     int limDwn,
                                     int limUp)
```

Prototyp OpenCL kernelu sa bude líšiť jedným parametrom naviac - *shInput*, ktorý predstavuje zdieľanú pamäť definovanú externe, mimo kernelu.

```
__kernel void oclFindDefects_kernel(__global unsigned char *input, //d_image_data
                                    unsigned int width,
                                    unsigned int heigth,
                                    __global int *output,              //d_defect_limits
                                    __global int *counter,             //d_counter_array
                                    int limDwn,
                                    int limUp,
                                    __local unsigned char *shInput)
```

Na začiatku tela CUDA kernelu zapíšeme do registrovej pamäti **lokálne súradnice** vlákna, ktoré kernel aktuálne vykonáva - *thread_x, thread_y*. Z toho vypočítame jeho **lokálny index** *local_idx*. Ďalej, pomocou súradníc aktuálneho bloku v mriežke a veľkosti globálnej mriežky, vypočítame globálne súradnice vlákna - *global_x, global_y*. Z globálnych súradníc vypočítame **globálny index** vlákna - *global_idx*.

```
116  unsigned int thread_x = threadIdx.x;
117  unsigned int thread_y = threadIdx.y;
118  unsigned int local_idx = thread_y * blockDim.x + thread_x;
119
120  unsigned int glob_x = (blockIdx.x * blockDim.x) + thread_x;
121  unsigned int glob_y = (blockIdx.y * blockDim.y) + thread_y;
122  unsigned int global_idx = glob_y * width + glob_x;
```

V OpenCL je výpočet globálnych súradníc vlákien totožný až na názvy vstavaných OpenCL funkcií, ktoré tieto súradnice vracajú. Tabuľka 7.1 porovnáva terminológiou pre indexovanie v CUDA a OpenCL.

Tab. 7.1: Terminológia pre indexovanie v CUDA a OpenCL

CUDA	OpenCL
gridDim	get_num_groups()
blockDim	get_local_size()
blockIdx	get_group_id()
threadIdx	get_local_id()
blockIdx * blockDim + threadIdx	get_global_id()
blockDim * gridDim	get_global_size()

V ďalšom kroku si v zdieľanej pamäti definujeme flag **defect** a zásobník **shInput**. Veľkosť zásobníka *shInput* (v bajtoch) je v prípade CUDA definovaná ako tretí parameter v «<»> zátvorkách pri volaní kernelu. V OpenCL musí byť tento zásobník predaný ako parameter do kernelu. Počet prvkov zdieľaného pamäťového zásobníka

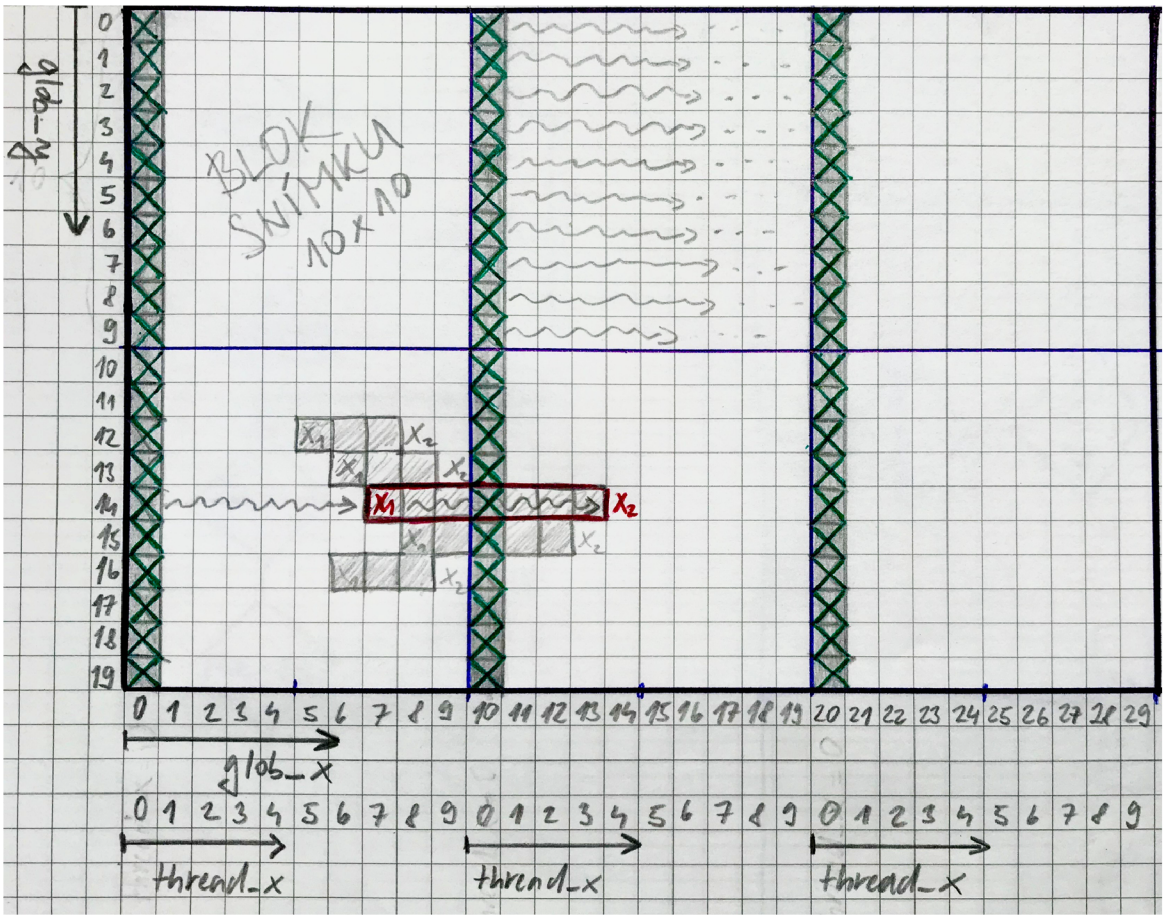
shInput korešponduje s počtom vlákien v jednom vláknovom bloku. Tým docielime, že každé vlákno z jedného bloku skopíruje, na základe svojich globálnych súradníc, jeden pixel vstupného snímku do zdieľanej pamäti. Po dokončení kopírovania je nutné vlákna zosynchronizovať funkciou *__syncthreads*.

Ak nejaké vlákno zistí, že hodnota pixelu je väčšia ako vrchný limit *limUp* alebo menšia ako spodný limit *limDwn*, tak nastaví flag *defect* uložený v zdieľanej pamäti na hodnotu 1. Opäť vlákna zosynchronizujeme.

```
124 __shared__ char defect;
125 extern __shared__ unsigned char shInput[];
126 shInput[local_idx] = input[global_idx];
127 __syncthreads();
128
129 if (shInput[local_idx] > limUp || shInput[local_idx] < limDwn)
130 {
131     defect = 1;
132 }
133 __syncthreads();
```

Teraz prichádza rad na popis hlavnej časti kernelu, ktorá vo vstupných dátach *d_image_data* vyhľadáva dvojice **X1, X2** začiatkov a koncov defektov na jednotlivých riadkoch. Následne tieto dvojice zapíše do výstupného bufferu *d_defect_limits* na pozíciu určenú polom *counter_array*.

Nasledujúcou podmienkou **if** povolíme vykonávanie hlavnej časti kernelu len vláknam z bloku, ktorý má zdieľaný flag *defect* nastavený na 1 a súradnicu *thread_x* rovnú 0. Na obrázku 7.3 vyhovujú tejto podmienke len vlákna označené zeleným krížikom zo spodných dvoch blokov zľava.



Obr. 7.3: Ilustrácia princípu kernelu na snímku 30x20 pixelov

Každé vlákno, ktoré túto podmienku splnilo dostane na starosť jeden **lokálny riadok** s 32 8-bitovými hodnotami (32 pixelov) vstupného obrazu *d_input_image* (pozn.: na obr. 7.3 má každé vlákno na starosti len 10 pixelov). Tento riadok bude začínať na indexe *global_idx* a končiť s indexom *global_idx + blockDim.x*. Nazvime index začiatku spracovávaného riadku ako *start_index* a index konca spracovávaného riadku ako *stop_index*.

```
135 if ((defect == 1) && (thread_x == 0))
136 {
137     unsigned int extend_x = 0;
138     unsigned int start_index = global_idx;
139     unsigned int stop_index = start_index + blockDim.x;
140     unsigned int x, x1, x2;
141     unsigned char value;
142     int state = state_nothing, last_state = state_nothing;
```

Hneď za podmienkou musí dané vlákno skontrolovať, či pixel snímky na indexe *(start_index - 1)* patrí do **priemeru** - hodnota väčšia ako *limDwn* a zároveň men-

šia ako *limUp*. Ak do priemeru nepatrí, znamenalo by to, že na rovnakom riadku *thread_y* z **predchádzajúceho vláknového bloku** bol detekovaný iba začiatok chyby **X1**. Koniec chyby - **X2** - bude teda pravdepodobne ležať na lokálnom riadku *thread_y* v aktuálnom vláknovom bloku (príp. až v nasledujúcich blokoch). Keby k tejto situácií došlo, vlákno spustí **while cyklus**, ktorý bude iterovať cez jednotlivé pixely od indexu *(start_index - 1)*, až kým narazí na pixel s hodnotou patriacou do priemeru. Index, na ktorom *while* cyklus skončil, sa zapíše do indexu *start_index* a až teraz sa začne vykonávať nesledujúci *for* cyklus so stavovým automatom. (pozn.: for cyklus sa ani začať nemusí, ak je *start_index* vyšší ako *stop_index*).

```
144    if (glob_x > 0)
145    {
146      unsigned int extend_x = start_index - 1;
147      if (input[extend_x] > limUp || input[extend_x] < limDwn)
148      {
149        while (input[extend_x] > limUp || input[extend_x] < limDwn)
150        {
151            extend_x++;
152        }
153        start_index = extend_x;
154      }
155    }
```

```
157    for (x = start_index; x < stop_index; x++)
158    {  //...nasleduje kód stavového automatu
159      value = input[x];
160      if (value < limDwn)
161          state = state_minus;
162      else if (value > limUp)
163          state = state_plus;
164      else
165          state = state_nothing;
```

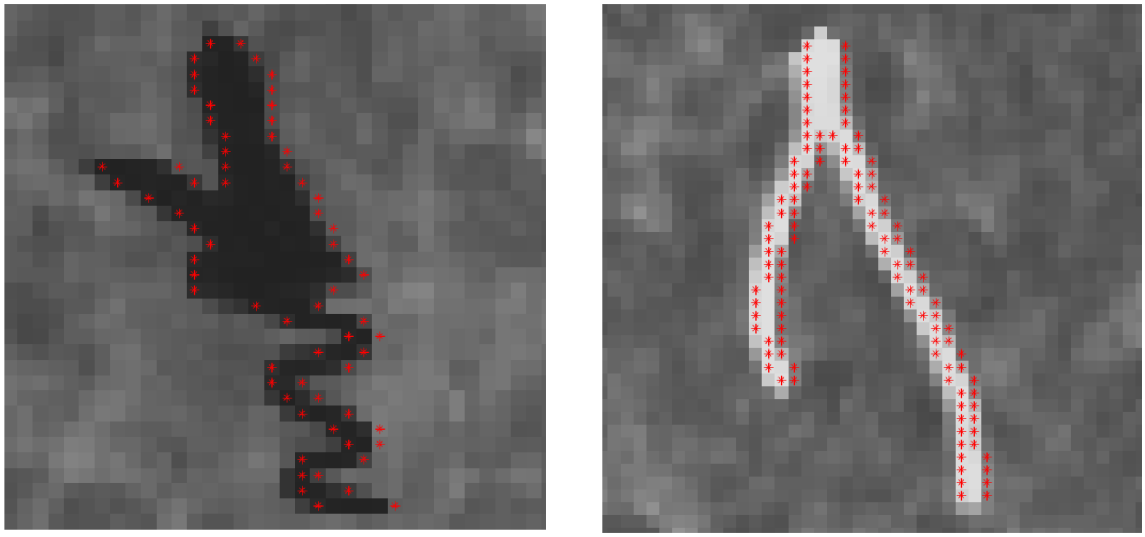
Vlákno využíva **for cyklus** k iterovaniu cez jednotlivé elementy riadku. Vo vnútri cyklu je enkapsulovaný **stavový automat s troma stavmi** - *state_minus*, *state_plus* a *state_nothing*. Ak vlákno narazí na element s hodnotou vyššou ako *limUp* alebo nižšou ako *limDwn*, zapíše do registrovej premennej **X1** súradnicu x aktuálneho elementu, ktorá sa vypočíta ako *glob_x + (x - start_index)*. Podľa toho, aký limit aktuálna hodnota prekročila, sa prejde z pôvodného stavu *state_nothing* do stavu **state_plus** resp. **state_minus**.

```
167     switch (state)
168     {
169     case state_plus:
170       if (last_state == state_nothing)
171       {
172         x1 = glob_x + (x - start_index);
173       }
174       if (x == (stop_index - 1))
175       {
176         unsigned int extend_x = x;
177         while (input[extend_x] > limUp || input[extend_x] < limDwn)
178         {
179             extend_x++;
180         }
181         x2 = glob_x + (extend_x - start_index);
182         output[glob_y * 40 + counter[glob_y]] = x1;
183         output[glob_y * 40 + counter[glob_y] + 1] = x2;
184         atomicAdd(&counter[glob_y], 2);
185       }
186       break;
```

For cyklus ďalej iteruje cez elementy riadku, až kým narazí na hodnotu, ktorá zapadá do *priemeru*. Vtedy sa stav znova nastaví späť na **state_nothing** a do registrovej premennej **X2** sa zapíše aktuálna súradnica x vypočítaná rovnako ako v predchádzajúcom prípade. Môže však dôjsť k situácii, že *for* cyklus už prešiel cez všetky pixely v lokálnom riadku a stále nenašiel súradnicu **X2** (koniec chyby). V tomto prípade sa na danom vlákne spustí *while* cyklus, ktorý bude iterovať cez nasledujúce pixely v globálnom riadku snímku, až kým nenarazí na koniec defektu **X2** alebo koniec snímku. Teraz máme k dispozícii dvojicu súradníc **X1, X2** označujúcich začiatok a koniec chyby na globálnej súradnici *glob_y*. Tieto hodnoty zapíšeme do výstupného zásobníku *d_defect_limits* na riadok *glob_y* a stĺpec daný hodnotu na pozícii *glob_y* v poli *d_counter_array*. Zjednodušene povedané, hodnoty **X1, X2** zapíšeme do zásobníka *d_defect_limits* na pozíciu *(glob_y * 40 + d_counter_array[glob_y])*. Po zápise oboch hodnôt musíme samozrejme inkrementovať counter *d_counter_array[glob_y]* o hodnotu 2.

```
205       case state_nothing:
206         if (last_state == state_plus || last_state == state_minus)
207         {
208           x2 = glob_x + (x - start_index);
209           output[glob_y * 40 + counter[glob_y]] = x1;
210           output[glob_y * 40 + counter[glob_y] + 1] = x2;
211           atomicAdd(&counter[glob_y], 2);
212         }
213         break;
214       }
215       last_state = state;
216   } //koniec for cyklu
```

Zdieľanú pamäť sme v úvode kernelu použili na to, aby sa všetky vlákna z jedného vláknového bloku paralelne podieľali na zanalyzovaní blokov snímky s rozmermi 32x32 pixelov. Vlákna zisťovali, či bloky obsahujú pixel s hodnotou mimo priemer. Ak nejaký blok takýto pixel obsahuje, vlákna nastavia zdieľaný flag *defect* na 1. Podľa tohto flagu každé vlákno vie, či sa v danom bloku majú hľadať chyby alebo nie. Sme si vedomí toho že v tejto implementácii dochádza k značnej **divergencii vlákien**. Už samotnou podmienkou *if* v úvode kernelu sme mnohé vlákna z procesu spracovania vylúčili a nechali sme pracovať len vlákna z vláknového bloku, ktorý má flag *defect* nastavený na hodnotu 1. Divergencia a obmedzenie počtu vlákien sú daňou za to, že na paralelný algoritmus boli už od počiatku kladené požiadavky, vyžadujúce sekvenčný zápis dvojíc **X1, X2** do výstupného pamäťového zásobníka *d_defect_limits*. Na obrázku 7.4 môžeme vidieť príklad výsledku detekcie defektov na snímkoch s rozlíšením 8192x1024 pixelov 8bpp. Vľavo je útvar s rozmermi 35x18 pixelov a vpravo je útvar s rozmermi 30x19 pixelov.



Obr. 7.4: Výsledok detekcie defektov na snímku 8192x1024.

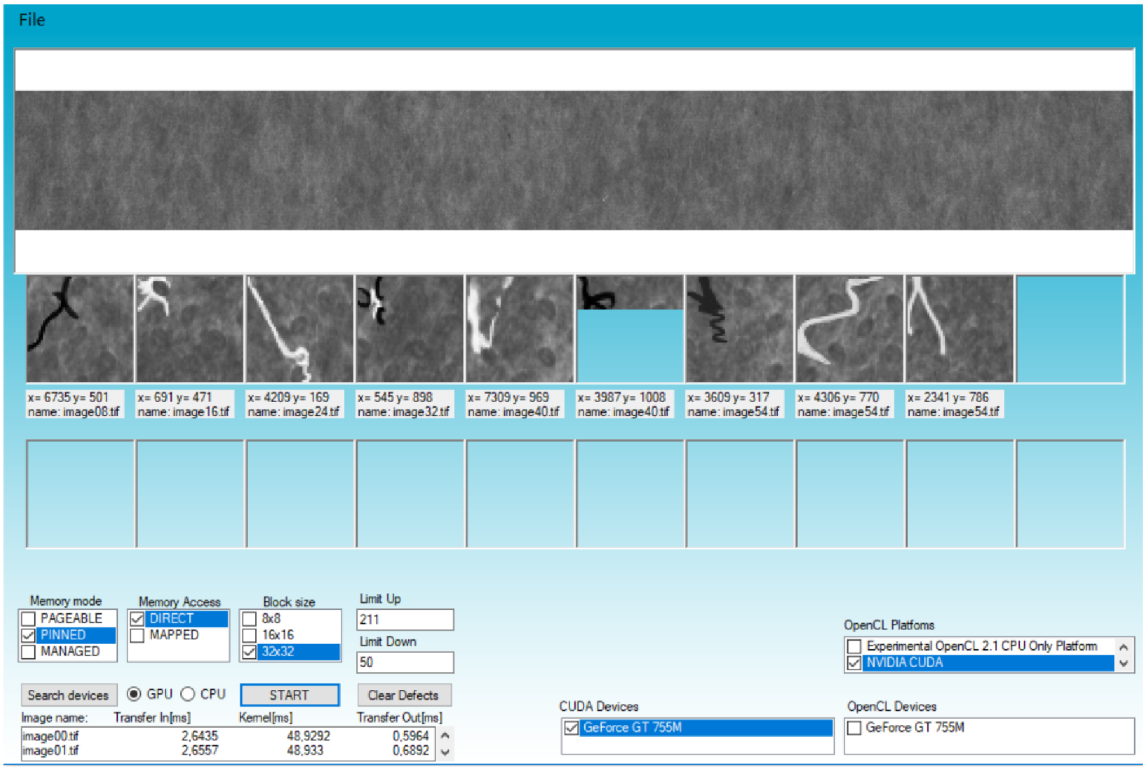
	HWConfig GT755M			HWConfig GTX1050Ti		
	CUDA	**OpenCL**	**CPU**	**CUDA**	**OpenCL**	**CPU**
Kopírovaná pamäť so stránkovaním	11,4514	12,3060	25,6672	19,9107	8,2959	38,1160
Kopírovaná pamäť s blokovaným stránkovaním	8,3942	12,2545		19,0572	7,9901	
Nekopírovaná pamäť s blokovaným stránkovaním	9,4313	14,8284		19,2339	11,5883	

Obr. 7.5: Porovnanie výkonu paralelného (na GPU) a sekvenčného (na CPU) algoritmu pre detekciu defektov s využitým rôznych pamäťových transferov

Aj napriek týmto obmedzeniam vykazuje navrhnutá implementácia veľmi dobré výsledky (viď. obr.10.1). Paralelizovaný algoritmus v podaní CUDA SW architektúri vykazuje až **3-násobné zrýchlenie** oproti spracovaniu na CPU *Intel(R) Core(TM) i7-4700MQ CPU @ 2.40GHz* (HWConfig GT755M). Testovanie na systéme *HWConfig GTX1050Ti* dopadlo v prospech OpenCL. Tu sme dosiahli až **6-násobného zrýchlenia** oproti spracovaniu na CPU *Intel(R) Core(TM) i5-650 CPU @ 3.20GHz*. Grafickú reprezentáciu tabuľky 10.1 nájdeme v prílohách - *PRÍLOHA A, PRÍLOHA B*.

8 GUI PRE CUDA A OPENCL FRAMEWORK

V nasledujúcom texte tejto kapitoly si popíšeme základné ovládanie a funkciu UI aplikácie **DefectDetection**, ktorá bola navrhnutá v prostredí Microsoft Visual Studio 2015 Community. Jedná sa o klasickú **CLR Windows Form** aplikáciu s podporou .NET Frameworku. Aplikácia je písaná v jazyku C++. K vôli problémom s linkovaním CUDA súborov (s príponou .cu) s CLR aplikáciou sme pre funkcie OpenCL a CUDA frameworku vytvorili dve separátne DLL knižnice - **useOpenCL.dll** a **useCUDA.dll**. Keďže sa jedná o dynamicky linkované knižnice, musia byť pri spúšťaní aplikácie umiestnené v rovnakom adresári, ako samotná aplikácia **DefectDetection.exe**.



Obr. 8.1: Funkcie uživateľského prostredia

Po spustení aplikácie **DefectDetection.exe** sa objaví okno so všetkými dostupnými funkciami (viď obr. 8.1). Vo vrchnom ľavom rohu je k dispozícii **File** menu. Pod nim sa nachádza priestor **PictureBox**, v ktorom sa zobrazí užívateľom vybraná snímka. Túto snímku je možné vybrať pomocou *File->Open*. V priloženom CD bude dostupný adresár s testovacími **.tif** snímkami s rozlíšením 8192x1024 pixelov, ktoré pochádzajú priamo z výrobného procesu netkaného textilu. Pod *PictureBoxom* sa nachádza 20 malých políčok zobrazujúcich detekované chyby na spracovaných sním-

koch. V spodnej tretine okna sú užívateľom voliteľné parametre. Predovšetkým ide o tri **CheckBoxListy** - **Memory mode**, **Memory Access** a **Block Size**. V každom *CheckBoxListe* môžeme zvoliť iba jednu z možnosti.

1. **Memory Mode** - ponúka možnosť vybrať typ pamäti, v ktorých budú alokované dáta transferované medzi CPU a GPU.
   - **PAGEABLE** - stránkovaná pamäť
   - **PINNED** - pamäť s blokovaným stránkovaním
   - **MANAGED** - zjednotená pamäť (angl. Unified Memory)
2. **Memory Access** - ponúka výber zo spôsobov transferu dát medzi CPU a GPU
   - **DIRECT** - kopírovanie dát medzi GPU a CPU
   - **MAPPED** - mapovanie dát do pamäti GPU
3. **Block Size** - veľkosť pracovných blokov (v OpenCL) resp. vláknových blokov (v CUDA)
   - **8x8**
   - **16x16**
   - **32x32**

Užívateľ si ďalej môže definovať vrchný a spodný limit citlivosti (Limit Up, Limit Down) pri vyhľadávaní defektov na snímku. **TextBoxy**, do ktorých sa tieto hodnoty zadávajú, prijímajú len numerické znaky. Aplikácia si sama kontroluje, či vrchný limit je vyšší ako spodný. V opačnom prípade, by aplikácia po stlačení tlačidla **ŠTART** upozornila užívateľa, že zadané hodnoty limít nie sú správne.

Prostredníctvom tzv. **RadioButtons** si užívateľ môže vybrať, či chce k detekcii použiť **CPU** alebo **GPU**. Až po zvolení jednej z možností je možné stlačiť tlačidlo **Search devices**, ktoré vyhľadá všetky dostupné zariadenia. Zariadenia kompatibilné s CUDA hardvérovou architektúrou sa zobrazia v *CheckBoxListe* **CUDA Devices**. V *CheckBoxListe* **OpenCL Platforms** sa po stlačení tlačidla **Search devices** objavia všetky vyhľadané CPU resp. GPU platformy v systéme. V prípade OpenCL si naviac musíme vybrať jednu z ponúkaných platforiem, na základe ktorej sa vyhľadajú všetky OpenCL kompatibilné zariadenia. Tieto zariadenia sa zobrazia v *CheckBoxListe* **OpenCLDevices**.

Teraz má užívateľ možnosť vybrať jedno z vyhľadaných zariadení. Vždy je možné vybrať len jedno z nich. Buď OpenCL kompatibilné alebo CUDA kompatibilné zariadenie. Akonáhle jedno zo zariadení vyberieme, sprístupní sa nám výber parametrov *Memory mode*, *Memory access* a *Block size*. Možnosti pre *Block size* sú závislé na vybranom zariadení, ktoré chceme použiť. Po vybraní zariadenia program automaticky zistí maximálny počet vlákien na jeden vláknový blok a ponúkne užívateľovi výber z tých najbežnejších rozmerov.

Tlačidlom **START** spustíme detekciu chýb na vybraných snímkoch. Aplikácia

ponúka možnosť vybrať viac snímkov k detekcii. V dvadsiatich políčkach sa zobrazia detekované chyby spolu s informáciou o názve snímky, na ktorom bola chyba detekovaná a súradnicami začiatku chyby v danom snímku. V prípade, že bol defekt detekovaný niekde na okraji snímku, zobrazí sa len časť tejto chyby tak ako je vidieť na snímke *image40.tif* na obrázku 8.1. Tlačidlo **Clear Defects** premaže obsah všetkých políčok, ktoré aktuálne zobrazujú detekované chyby.

V ľavom dolnom rohu je *TextBox*, v ktorom sa po spracovaní vybraných snímkov zobrazia výsledky meraného času. Pre každú snímku sa zobrazí čas potrebný pre jeho kopírovanie do globálnej pamäti GPU (*Transfer In*), čas vykonania kernelu (*Kernel*) a čas na skopírovanie výstupných dát späť do CPU (*Transfer Out*).

9 HARDVÉR Z PROCESU VÝROBY

Potreba optimalizácie algoritmu detekujúceho defekty v textile je dôsledkom zmeny kamerového systému v blízkej budúcnosti. V aktuálnom stave sú nad linkou nasadené 4 kamery **Basler 4096 pix @ 70 kHz**, čo dohromady produkuje tok dát 1.1GB/sec. Aktuálny algoritmus je spracovávaný na 4 jadrovom CPU Intel(R) Core(TM) i7-3820 @3.6GHz s vyťaženosťou 12%. V budúcnosti má byť aktuálne kamerové vybavenie nahradené 4 kamerami **ELIIXA+ 16k @ 200kHz**. Ktorej stručnú špecifikáciu uvedieme v nasledujúcej sekcii.

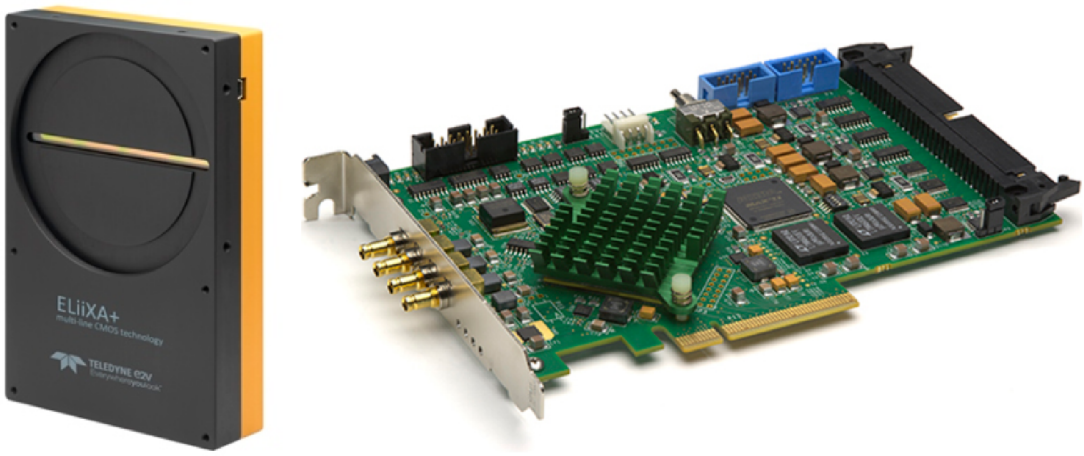
9.1 ELIIXA+ 16k monochrome

Táto kamera patrí k novej generácií priemyselných riadkových kamier od spoločnosti **Teledyne e2V**. Vďaka multi-line CMOS technológií poskytuje **riadkovú frekvenciu až 200kHz** s rozlíšením 11k pixelov a vysokou redukciou šumu. V procese bude kamera s veľkou pravdepodobnosťou pracovať v režime 8k, preto sme v tejto práci používali na experimentovanie snímky s rozlíšením 8192x1024 pixelov. Jedná sa teda o obrovský tok dát dosahujúci až 1.6GB/sec z jednej kamery. Vysoký pomer signál-šum kamery umožňuje kvalitné snímky aj pri limitovanej úrovni osvetlenia. Kamera disponuje dátovým prenosom až 1.6Gpix/sec sprostredkovaným moderným rozhraním **CoaXPress**, čo umožňuje obrovskú priepustnosť dát medzi kamerou a frame grabberom. Pixely majú rozmer 5x5 µm a sú usporiadané do 4 aktívnych radov (2 x 2 páry). Každá z dvojíc radov svetlocitlivých snímačov používa svoj vlastný ADC konvertor. Časové oneskorenie expozície medzi dvoma riadkami umožňuje kombinovať dve po sebe vykonané expozície do jednej, za účelom zdvojnásobenia svetelnej citlivosti na jednom riadku. [16]

9.2 Komunikačné rozhranie CoaXPress

Pre komunikáciu s riadkovými kamerami sa používajú tri druhy rozhraní. V systémoch, kde nie je latencia až tak kritická, by bol jasnou voľbou **gigabitový ethernet**. V priemysle sa žiada použiť kamery s vysokými frekvenciami snímania, čo vedie k obrovskému objemu dát na výstupe (v našom prípade až 1.6GB/sec na kameru). Tu je vhodné použiť buď komunikačné rozhranie **Camera Link**, alebo najmodernejšiu technológiu využívajúcu koaxiálne káble - **CoaXPress**. [15],[18]

Na rozdiel od *Camera Linku*, komunikácia CoaXPress využíva k prenosu koaxiálne káble čo umožňuje prepojenie s frame grabberom aj na viac ako 100 metrov. Aktuálna verzia podporuje jednokanálové posielanie dát z kamery rýchlosťou až



Obr. 9.1: Kamera ELIIXA+ 16k monochrome (vľavo); Cyton-CXP frame grabber (vpravo).[16],[15]

6.25Gb/sec 780MB/s. V systéme, o ktorom pojednáva táto práca, však bude požadovaný prenos cca 1.6GB/s na jednu kameru! Aj k tomuto problému má CoaXPress patričné riešenie. Je totiž možné agregovať 4 kanály a tým dosiahnuť prenos až 25Gb/sec.

9.3 Frame grabber Cyton-CXP

Frame grabber **Cyton-CXP** od spoločnosti **BitFlow** je určený pre vysokorýchlostné plošné aj riadkové kamery na zbernici CoaXPress. Obsahuje nastaviteľné časovače, čítače, PWM generátory, enkódery atď. Je schopný generovať navzájom nezávislé signály, čím môže ovládať každú kameru zvlášť. Prostredníctvom DMA je možný príjem dát bez väčšieho zaťaženia CPU. Všetky 4 kanály, ktoré karta ponúka sa dajú agregovať čo zabezpečí prenos až 25Gb/sec medzi kamerou a frame grabberom. Tieto karty sú určené pre sloty *PCIe 8x Gen 2*. PCIe zbernica je peer-to-peer čo znamená, že frame grabber nezdieľa zbernicu so žiadnym iným zariadením. To znamená, že bude komunikovať priamo s PCI chipsetom v pamäťovej zbernici a to vedie k ustálenejšej priepustnosti DMA prenosu bez ohľadu na množstvo prenášaných dát.[15], [18]

10 ZÁVER

V dnešnom priemysle sa vysokým tempom dostáva do popredia optická kontrola produkovaných výrobkov. Úspešnosť takejto kontroly síce stále nie je dokonalá, no v porovnaní s vizuálnou kontrolou alebo kontrolou pomocou klasických priemyselných snímačov a senzorov, je veľmi presná, a s doprovodom dostatočne výkonného hardvéru, vhodného osvetlenia a scény, aj rýchla.

Optická kontrola bola nasadená aj do reálneho procesu výroby v spoločnosti PEGAS NONWOVENS a.s., ktorá produkuje netkaný textil na nekonečných pásoch. Ide o komplexný systém pozostávajúci zo 4 kamier, ktorý detekuje defekty (diery, zátrhy, zalisované nečistoty...atď.) na produkovanom textile. Vlastníkom tohto systému je spoločnosť CAMEA spol. s.r.o., pre ktorú táto práca bola vyhotovená.

Riešený problém spočíval v optimalizácii vyťaženej časti detekčného algoritmu pomocou softvérovej a hardvérovej architektúry CUDA, a frameworku OpenCL určeného k programovaniu na heterogénnych zariadeniach. Potreba optimalizácie vyplýva z návrhu na výmenu aktuálneho kamerového systému za presnejší a výkonnejší. Aktuálny kamerový systém pozostáva zo 4 kamier Basler 4096 pix @ 70 kHz, ktoré dohromady produkujú tok dát 1.1GB/sec. Sústava so 4 novými kamerami Eliixa 16kpix @ 200kHz by v režime 8kpix produkovala na výstupe až 13GB/sec.

Prvým cieľom tejto práce bolo detailnejšie zoznámenie s oboma nástrojmi NVIDIA a OpenCL. Rovnako dôležité ako rozbor programovacieho modelu týchto dvoch nástrojov bol aj popis a vysvetlenie hardvérovej architektúry paralelnej výpočtovej platformy CUDA. Verím, že zhrnuté poznatky a ukážky programovacích praktík bude možné využiť aj v nadväzujúcom vývoji paralelného algoritmu. Po detailnom výskume, zvážení vstupných požiadaviek a rozbore trhu s modernými grafickými kartami, sme stanovili ako experimentálnu grafickú kartu NVIDIA GeForce GTX1050Ti, ktorá sa stala súčasťou PC zostavy s označením HWconfig GTX1050Ti.

V nasledujúcej časti tejto práce sme popísali, implementovali a otestovali rôzne optimalizačné metódy. Jednalo sa najmä o optimalizáciu transferu dát medzi pamäťou GPU a CPU, ktorý je pri paralelizácií algoritmu kritický. K testovacím účelom sme využili 2 rôzne PC zostavy - HWConfig GT755M a spomínanú zostavu HWconfig GTX1050Ti (viď detaily 1.4). Vzhľadom k dosiahnutým výsledkom (viď PRÍLOHA A, PRÍLOHA B) sme vyhlásili prenos *kopírovanej pamäti s blokovaným stránkovaním* za najpriaznivejší pre túto aplikáciu.

V hlavnej časti sme navrhli a otestovali paralelný ekvivalent aktuálneho sekvenčného algoritmu detekujúceho defekty v textile. Tento návrh je dôkladne popísaný a odôvodnený v kapitole 7.2. Do záveru som si dovolil uviesť tabuľku 10.1 s dosiahnutými výsledkami. Tabuľka obsahuje výsledky merania rýchlosti navrhnutého kernelu na oboch PC zostavách, pri použití rôznych metód transferu dát medzi GPU

a CPU. Na hardvérovej konfigurácii HWConfig GT755M sa nám detekcia chýb podarila zrýchliť **3-násobne**. Výsledky z testov na konfigurácii HWConfig GTX1050Ti dokonca poukazujú až na **6-násobné** zrýchlenie oproti rýchlosti sekvenčného algoritmu na CPU.

	HWConfig GT755M			HWConfig GTX1050Ti		
	CUDA	OpenCL	CPU	CUDA	OpenCL	CPU
Kopírovaná pamäť so stránkovaním	11,4514	12,3060	25,6672	19,9107	8,2959	38,1160
Kopírovaná pamäť s blokovaným stránkovaním	8,3942	12,2545		19,0572	7,9901	
Nekopírovaná pamäť s blokovaným stránkovaním	9,4313	14,8284		19,2339	11,5883	

Obr. 10.1: Porovnanie výkonu paralelného (na GPU) a sekvenčného (na CPU) algoritmu pre detekciu defektov s využitým rôznych pamäťových transferov

Kernel bol od počiatku vyvíjaný na PC zostave HWConfig GT755M, čo znamená, že je optimalizovaný pre mikroarchitektúru Kepler. To viedlo k nie moc uspokojivým výsledkom pri testoch na HWConfig GTX1050Ti. Rýchlosť vykonania kernelu napísaného v CUDA C bolo približne 20 ms.

Od paralelného algoritmu sa očakávalo, že po spracovaní snímky pošle späť do réžie CPU buffer obsahujúci súradnice X začiatkov a koncov chýb na jednotlivých riadkoch snímky, ktoré by sa hneď predali algoritmu pre zhľukovú analýzu. Táto kladená požiadavka na výstup značne skomplikovala implementáciu kernelu, pretože sme sa museli vysporiadať so serializovaným prístupom do globálnej pamäti GPU. Toto kritérium bolo v práci splnené.

Ďalšou podstatnou časťou práce bol vývoj GUI na testovanie nástrojov CUDA a OpenCL. Výstupom z tohto bodu je program **DefectDetection.exe**, ktorý detekuje defekty na snímkach textílu predložených užívateľom. Ide o rozsiahlejšiu *CLR Windows Form* aplikáciu s podporou .NET Frameworku.

V nadväzujúcom vývoji paralelného algoritmu by určite prinieslo pozitívne výsledky rozdelenie transferov dát a vykonávanie kernelu medzi viaceré CUDA streami resp. príkazové rady v prípade OpenCL. Týmto spôsobom by sa dalo dosiahnuť súbežného spracovania snímkov a kopírovania dát medzi pamäťou CPU a GPU. Existuje aj ďalšia možnosť, ktorou by sme mohli skoro úplne eliminovať transfery dát.

Moderné frame grabbery sú schopné posielať dáta priamo do globálnej pamäti GPU, čo by v tomto prípade viedlo k ďalšiemu zrýchleniu paralelného algoritmu.

Na záver by som spomenul ešte jednu možnosť. Podľa [3], jedinou možnosťou ako zrýchliť paralelný algoritmus na GPU je použitie viacerých GPU. Spoločnosť NVIDIA ponúka rozhranie SLI, čo umožní rozdelenie parciálnych častí algoritmu medzi viaceré grafické karty (až 4), kde každá z nich by bola riadená jedným vláknom z CPU.

LITERATÚRA

[1] CAMEA, spol. s.r.o. *Kontinuální sledování pásů: UniscanDETECTOR*[online]. [cit. 26. 11. 2017]. Brno, 2016, 4 s., Dostupné z URL: <http://www.camea.cz/underwood/download/files/detector_cat_cs_20170303_web.pdf>.

[2] *Programming Guide::CUDA Toolkit Documentation*[online]. [cit. 7. 11. 2017]. Dostupné z URL: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-general-purpose-parallel-computing-architecture>.

[3] SANDERS, J., KANDORT, E.: *CUDA by example: an Introduction to General-Purpose GPU Programming* [online]. [cit. 5. 11. 2017]. Dostupné z URL: <http://www.itp.cas.cn/kxjs/jzytz/pxyjz/201306/P020130624362235915673.pdf>.

[4] *Techopedia - The IT Education Site* [online]. [cit. 5. 11. 2017]. Dostupné z URL: <https://www.techopedia.com/definition/8777/parallel-computing>.

[5] *Wikipedia* [online]. [cit. 5. 11. 2017]. Dostupné z URL: <https://en.wikipedia.org/wiki/Parallel_computing#Bit-level_parallelism>.

[6] OOSTEN, J.: *3D Game Engine Programming* [online]. [cit. 15. 11. 2017]. Dostupné z URL: <https://www.3dgep.com/>.

[7] NVIDIA: *CUDA C Best Practices Guide.* www.nvidia.com [online]. 2009, verzia Január 2012, [cit. 4. 2018]. Dostupné z URL: <https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf>.

[8] RAVISHEKHAR, B., BHATTACHARYYA, K.: *OpenCL Programming by Example. A comprehensive guide on OpenCL programming with examples* Birmingham: Packt Publishing, 2013. ISBN 978-1-84969-234-2. [online]. [cit. 3. 2018].

[9] NVIDIA: *OpenCL Best Practices Guide.* www.nvidia.com [online]. 2009, verzia Februar 2011, [cit. 3. 2018]. Dostupné z URL: <https://hpc.oit.uci.edu/nvidia-doc/sdk-cuda-doc/OpenCL/doc/OpenCL_Best_Practices_Guide.pdf>.

[10] KHRONOS: *The OpenCL Specification.* www.khronos.org [online]. 2009, verzia 2.0, revízia 21. Júl 2015, [cit. 3. 2018]. Dostupné z URL: <https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf>.

[11] AMD: *Introduction to OpenCL™Programming. Training Guide* www.amd.com [online]. Máj 2010, revízia A, číslo publikácie: 137-41768-10. [cit. 3. 2018]. Dostupné z URL: <https://developer.amd.com/wordpress/media/2013/01/Introduction_to_OpenCL_Programming-Training_Guide-201005.pdf>.

[12] NVIDIA: *Whitepaper, NVIDIA GeForce GTX 1080* www.nvidia.com [online]. [cit. 4. 2018]. Dostupné z URL: <https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf>.

[13] NVIDIA: *Whitepaper, NVIDIA's Next Generation CUDA™Compute Architecture: Kepler TM GK110* www.nvidia.com [online]. verzia V1.1, [cit. 4. 2018]. Dostupné z URL: <https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>.

[14] NVIDIA: *Whitepaper, NVIDIA's Next Generation CUDA™Compute Architecture: Fermi™* www.nvidia.com [online]. verzia V1.1, [cit. 4. 2018]. Dostupné z URL: <https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf>.

[15] BITFLOW: *Cyton-CXP* www.bitflow.com [online]. [cit. 5. 2018]. Dostupné z URL: <http://www.bitflow.com/products/details/karbon-cxp>.

[16] TELEDYNE E2V: *ELIIXA+ 16k/8k Mono, Cmos Multi-Line Monochrome Camera* www.e2v.com [online]. [cit. 5. 2018]. Dostupné z URL: <https://www.e2v.com/content/uploads/2017/01/DSC_ELIIXAplus16K_1706.pdf>.

[17] HEXUS: *Review: EVGA GeForce GTX 1050 Ti SC Gaming* www.hexus.net [online]. [cit. 5. 2018]. Dostupné z URL: <http://hexus.net/tech/reviews/graphics/98329-evga-geforce-gtx-1050-ti-sc-gaming/>.

[18] ATEsystem: *DÍL 4: KOMUNIKACE S ŘÁDKOVOU KAMEROU* www.kamery.atesystem.cz [online]. [cit. 5. 2018]. Dostupné z URL: <http://kamery.atesystem.cz/know-how/line-scan-velky-pruvodce-radkovymi-kamerami/dil-4-komunikace-s-radkovou-kamerou/>.

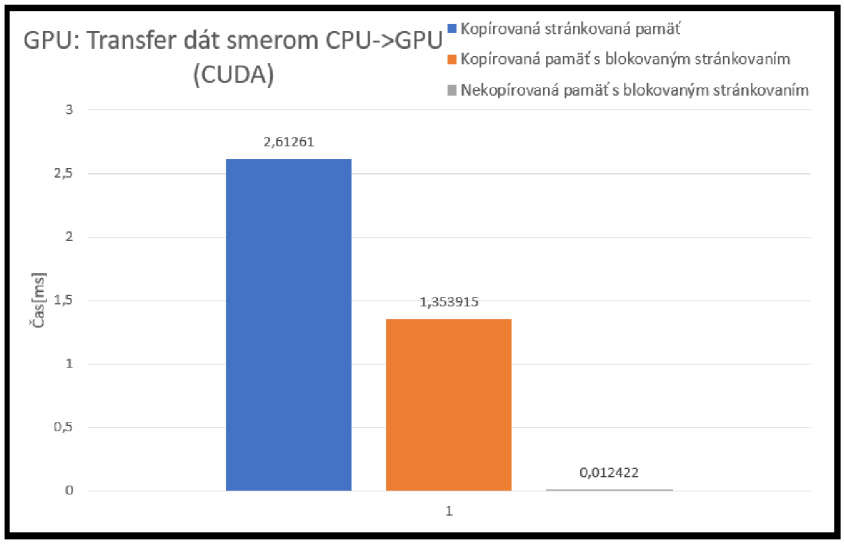
ZOZNAM SYMBOLOV, VELIČÍN A SKRATIEK

HW	Hardvér
SW	Softvér
FSB	R/W Pamäť s priamym prístupom - zbernica medzi CPU a severným mostíkom (northbridge)
GPGPU	General-purpose computing on graphics processing units
CUDA	Compute Unified Device Architecture
IPP	Integrated Performance Primitives
GPU	grafický procesor - Graphics Processing Unit
CPU	procesorova jednotka - Central Processing Unit
GPC	Graphics Processing Clusters
SM	Streaming Multiprocessor
IDU	Instruction Dispatch Unit
SLI	Scalable Link Interface - umožňuje prepojenie viac GPU na jednej základnej doske
DMA	Priamy prístup do pamati - Direct Memory Access
RAM	R/W Pamať s priamym prístupom - Random Access Memory
COI	Channel of Interest
ROI	Region of Interest
ADC	Analog to Digital Column

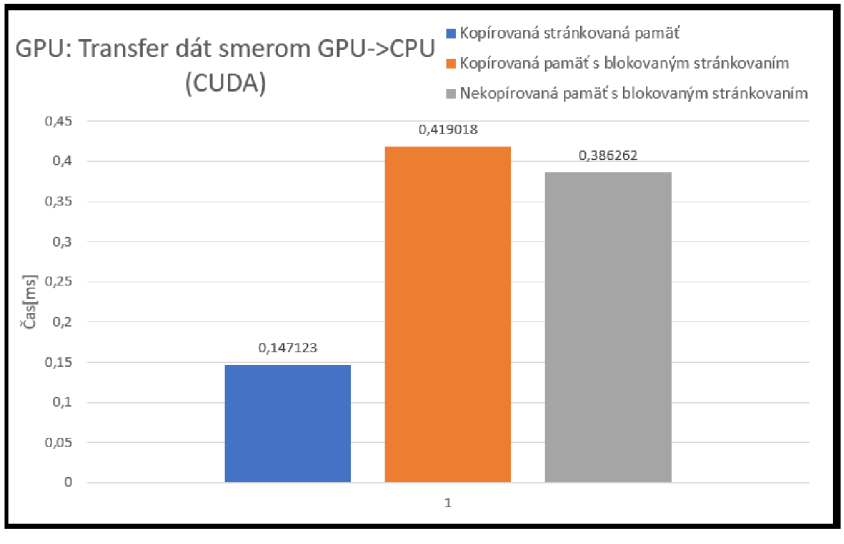
ZOZNAM PRÍLOH

A PRÍLOHA A

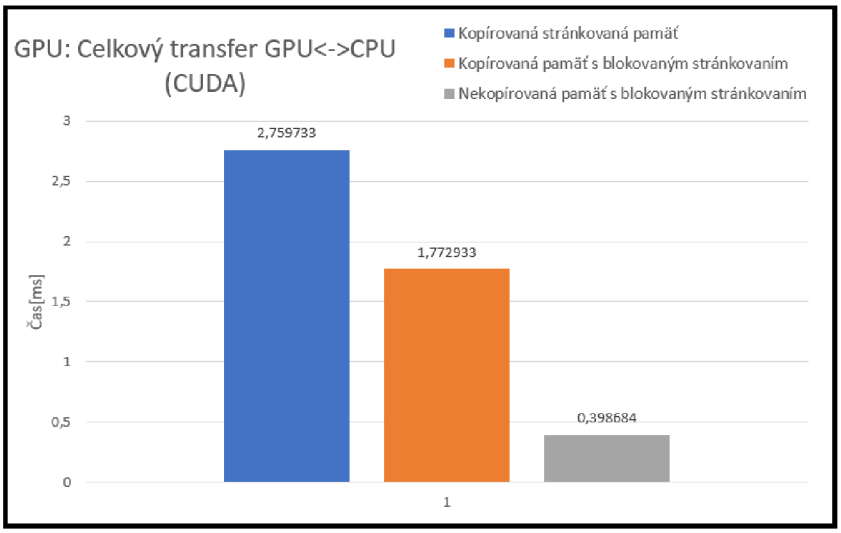
A.1 Implementácia na NVIDIA GeForce GT755M využitím CUDA softvérovej architektúry



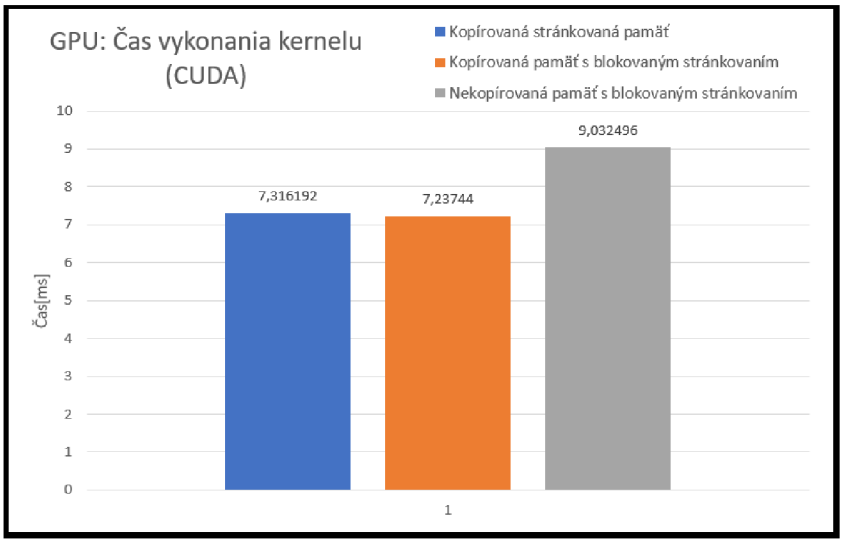
Obr. A.1: HWConfig GT755M: CUDA: Meranie rýchlosti transferu 3 druhov pamäti z CPU do GPU.



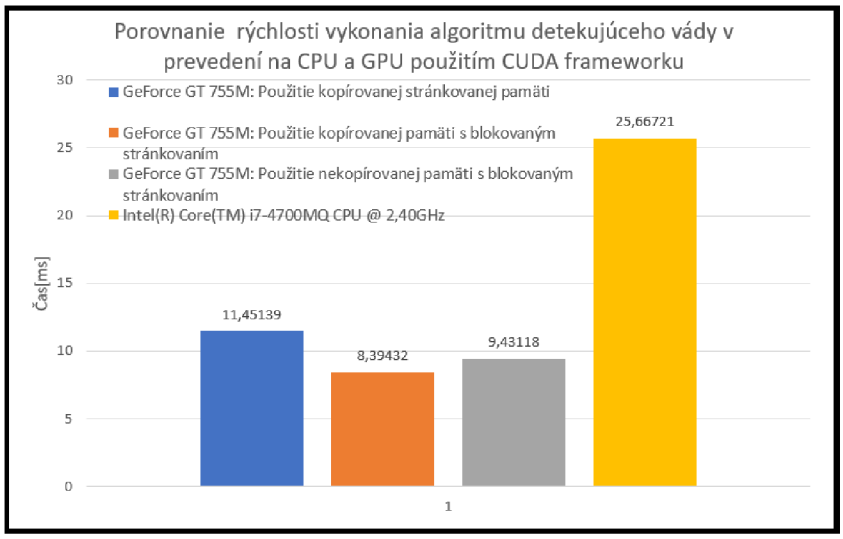
Obr. A.2: HWConfig GT755M: CUDA: Meranie rýchlosti transferu 3 druhov pamäti z GPU do CPU.



Obr. A.3: HWConfig GT755M: CUDA: Meranie rýchlosti oboch transferov pamäti medzi CPU a GPU.

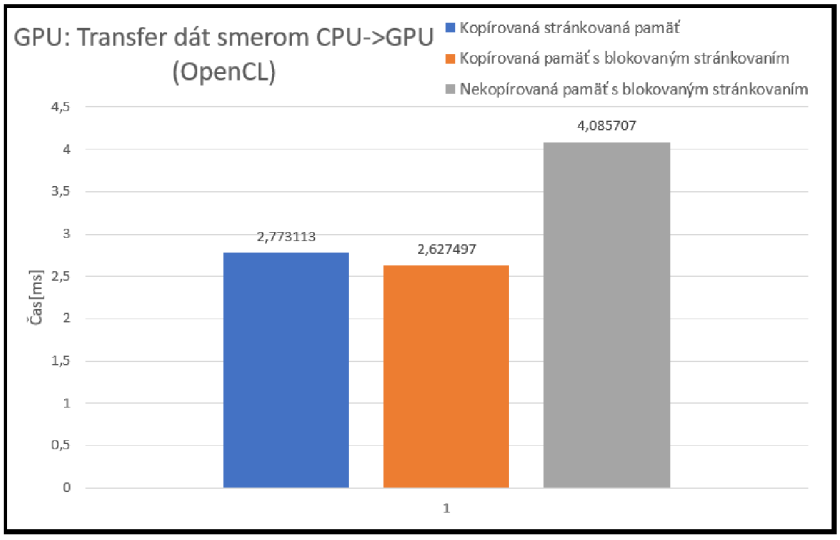


Obr. A.4: HWConfig GT755M: CUDA: Meranie rýchlosti vykonania kernelu.

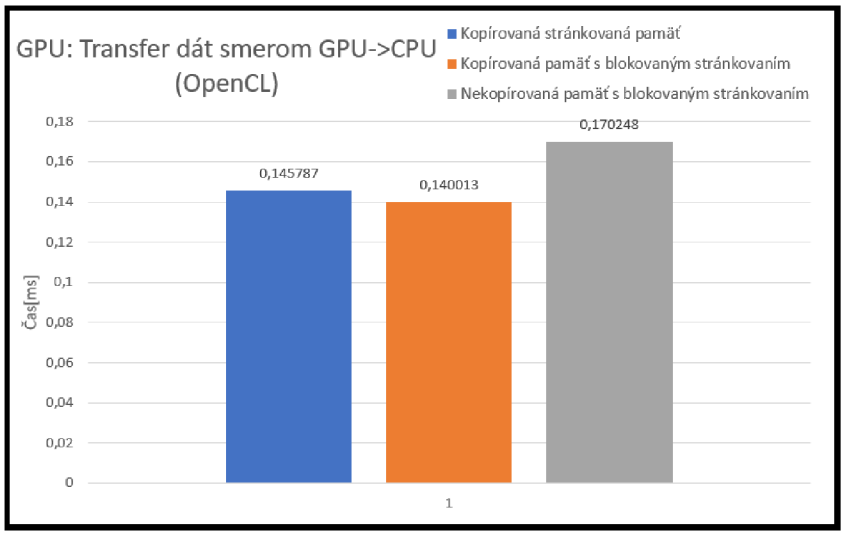


Obr. A.5: HWConfig GT755M: CUDA: Meranie rýchlosti vykonania celého paralelného algoritmu + porovnanie so sekvenčným.

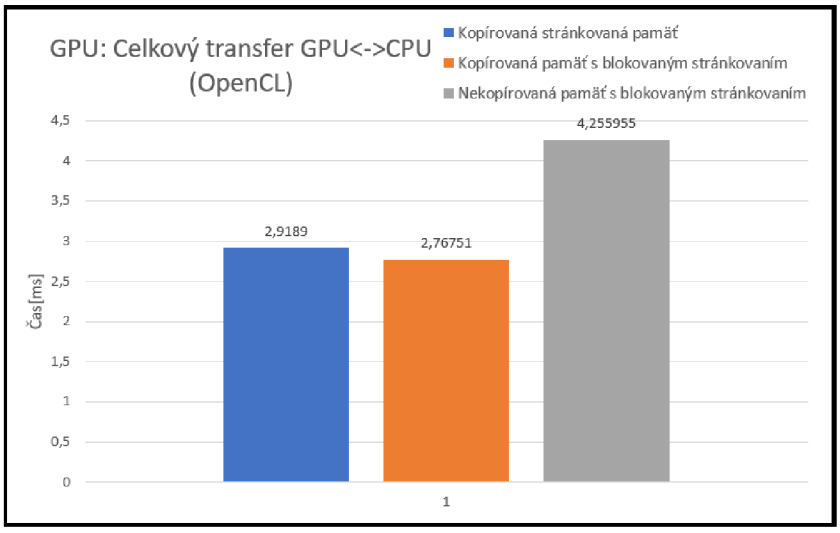
A.2 Implementácia na NVIDIA GeForce GT755M využitím OpenCL frameworku



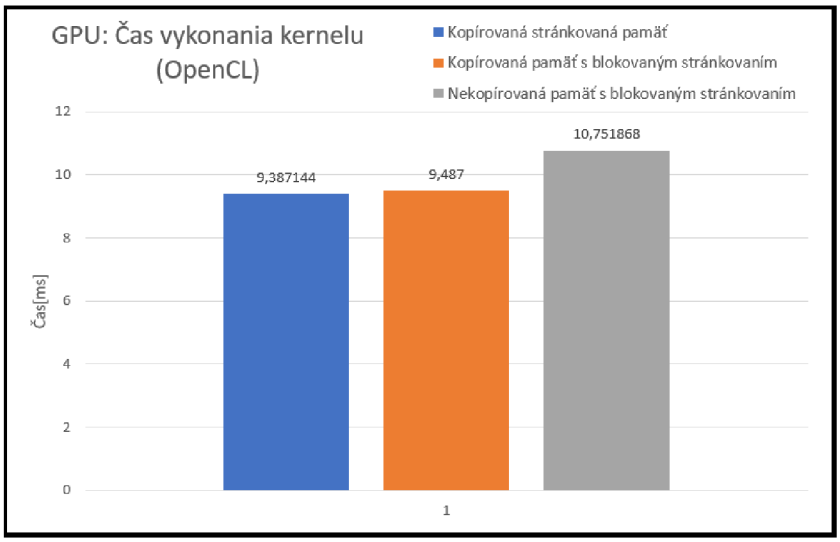
Obr. A.6: HWConfig GT755M: OpenCL: Meranie rýchlosti transferu 3 druhov pamäti z CPU do GPU.



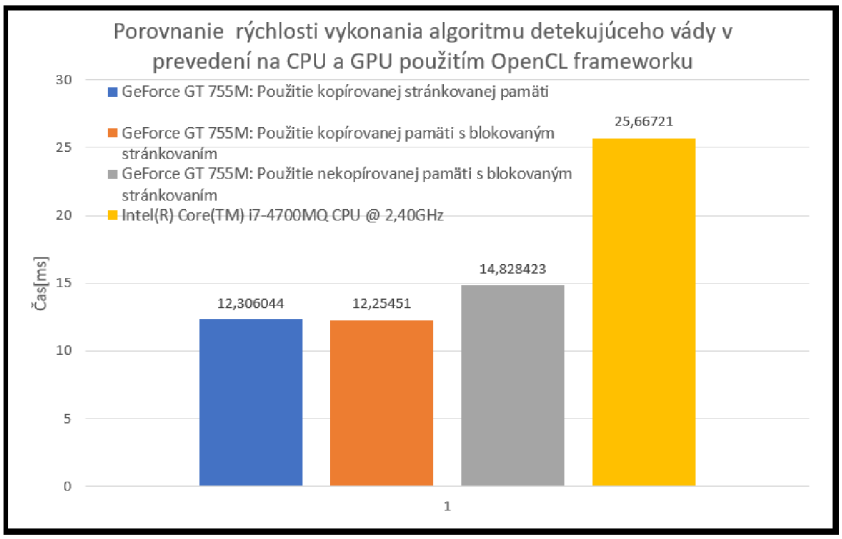
Obr. A.7: HWConfig GT755M: OpenCL: Meranie rýchlosti transferu 3 druhov pamäti z GPU do CPU.



Obr. A.8: HWConfig GT755M: OpenCL: Meranie rýchlosti oboch transferov pamäti medzi CPU a GPU.



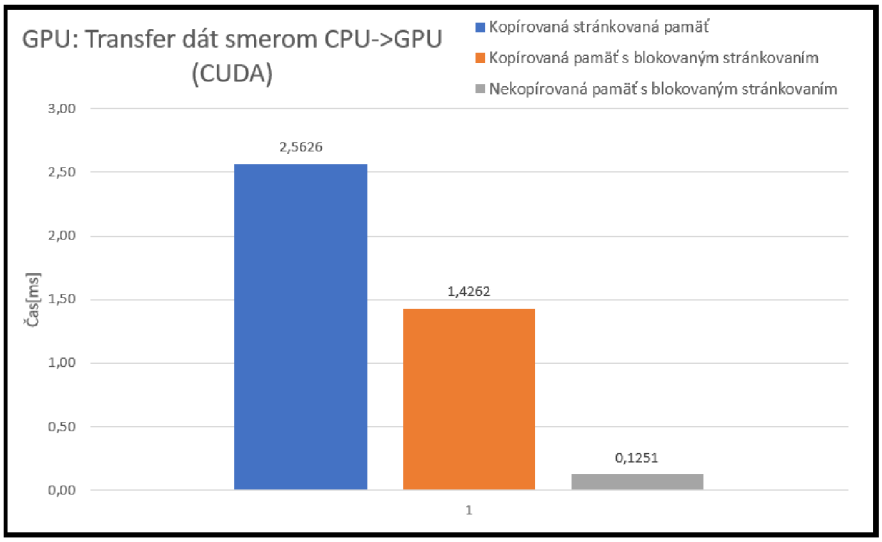
Obr. A.9: HWConfig GT755M: OpenCL: Meranie rýchlosti vykonania kernelu.



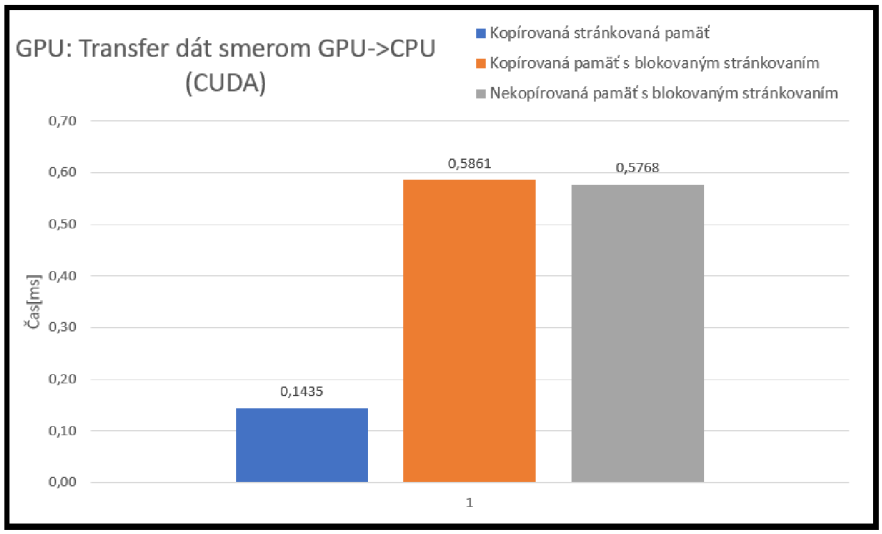
Obr. A.10: HWConfig GT755M: OpenCL: Meranie rýchlosti vykonania celého paralelného algoritmu + porovnanie so sekvenčným.

B PRÍLOHA B

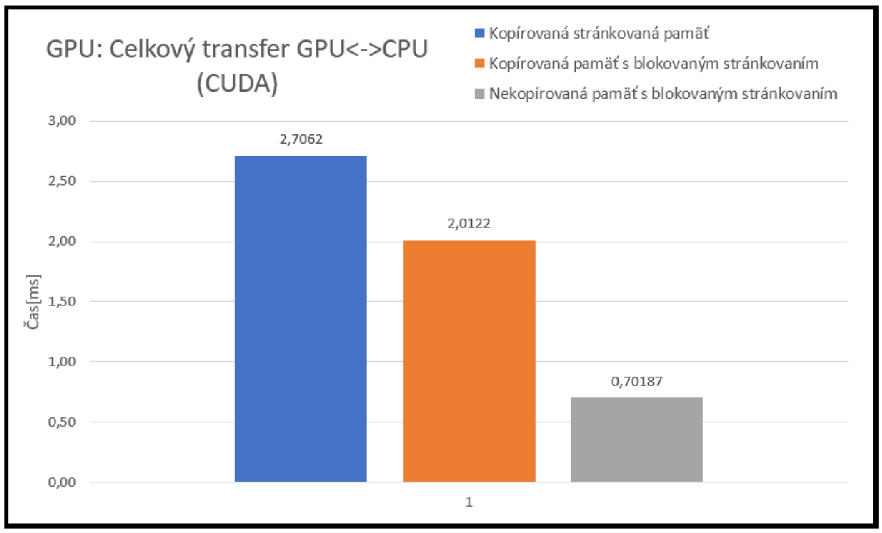
B.1 Implementácia na NVIDIA GeForce GTX1050Ti využitím CUDA softvérovej architektúry



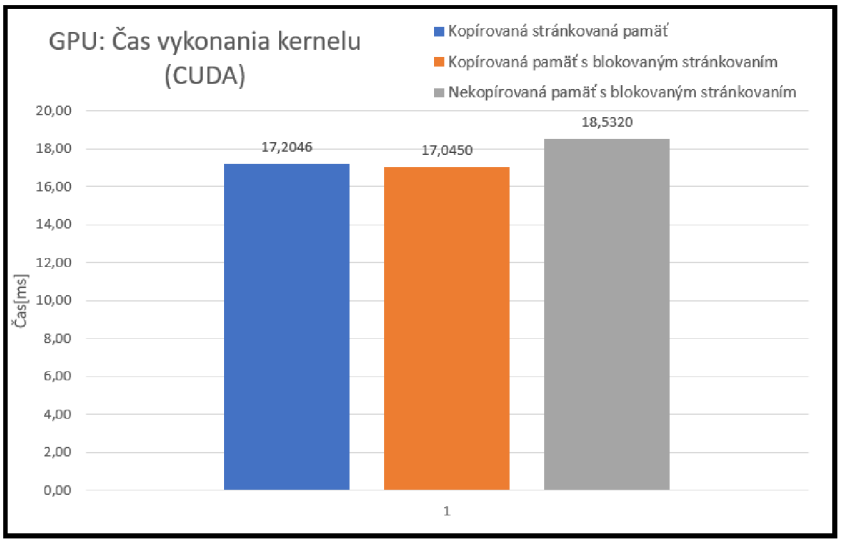
Obr. B.1: HWConfig GTX1050Ti: CUDA: Meranie rýchlosti transferu 3 druhov pamäti z CPU do GPU.



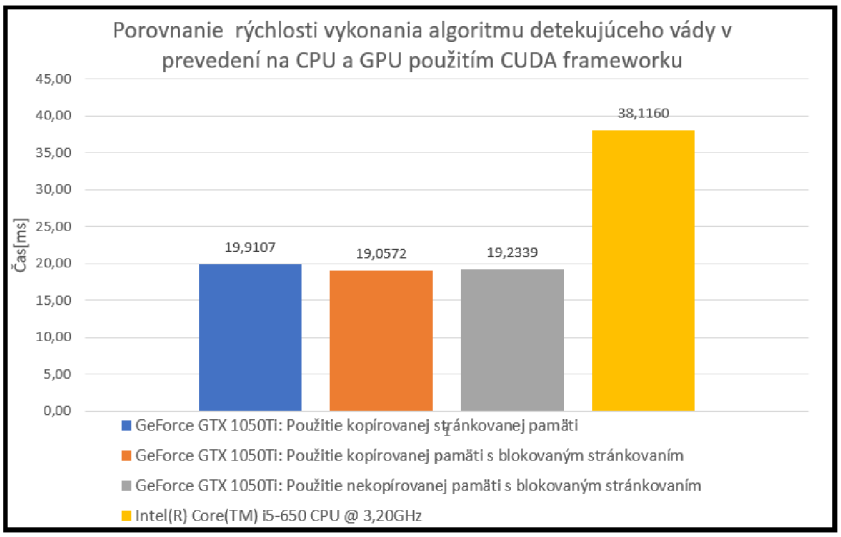
Obr. B.2: HWConfig GTX1050Ti: CUDA: Meranie rýchlosti transferu 3 druhov pamäti z GPU do CPU.



Obr. B.3: HWConfig GTX1050Ti: CUDA: Meranie rýchlosti oboch transferov pamäti medzi CPU a GPU.

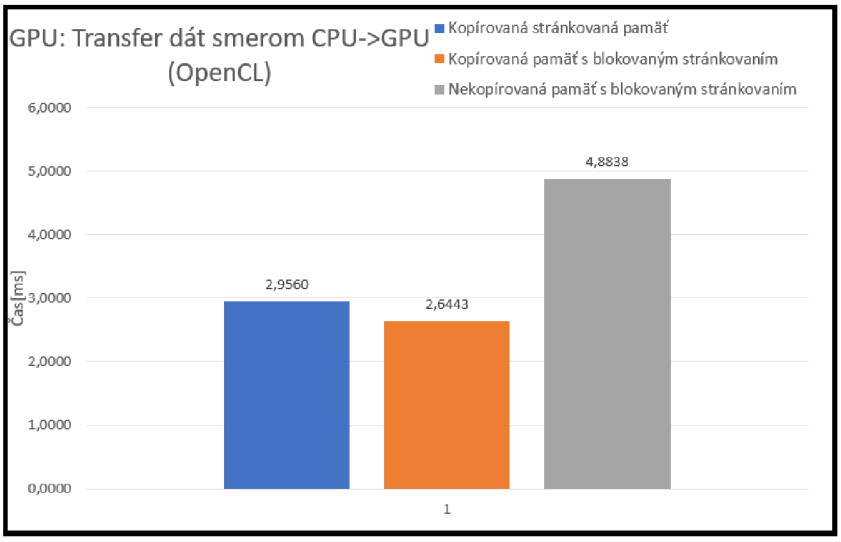


Obr. B.4: HWConfig GTX1050Ti: CUDA: Meranie rýchlosti vykonania kernelu.

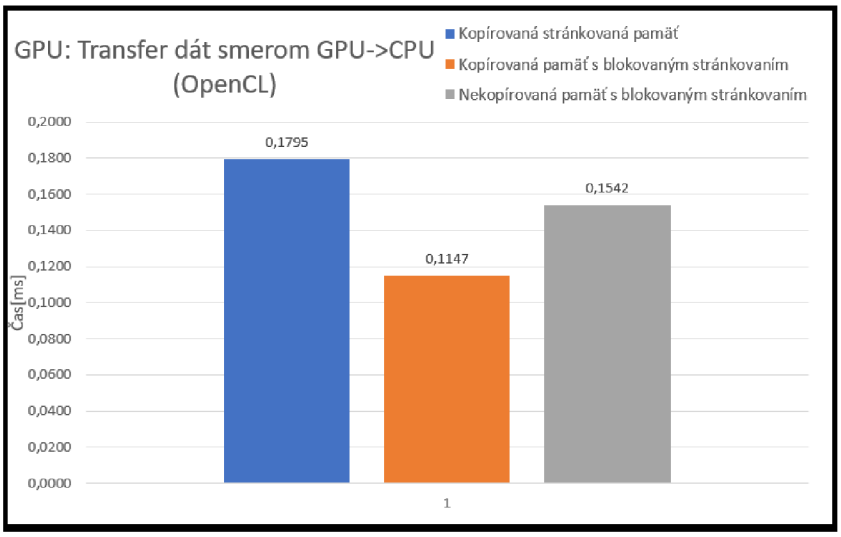


Obr. B.5: HWConfig GTX1050Ti: CUDA: Meranie rýchlosti vykonania celého paralelného algoritmu + porovnanie so sekvenčným.

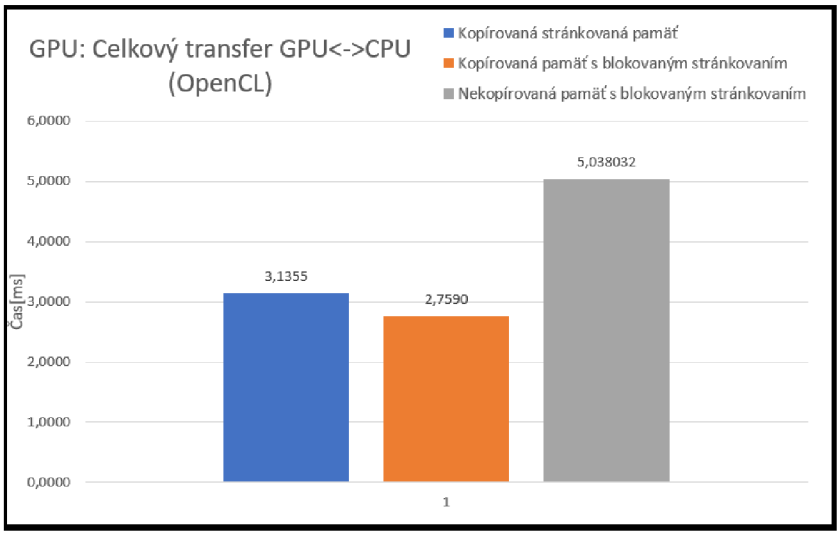
B.2 Implementácia na NVIDIA GeForce GTX1050Ti využitím OpenCL frameworku



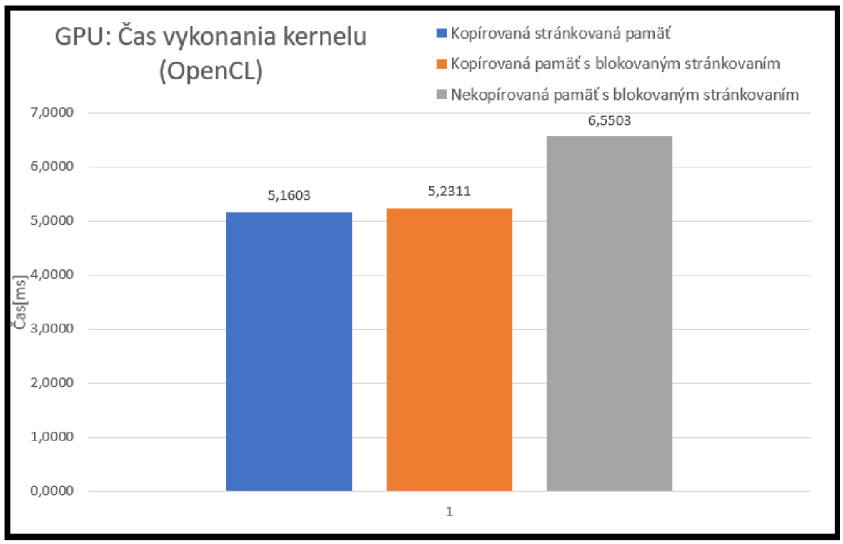
Obr. B.6: HWConfig GTX1050Ti: OpenCL: Meranie rýchlosti transferu 3 druhov pamäti z CPU do GPU.



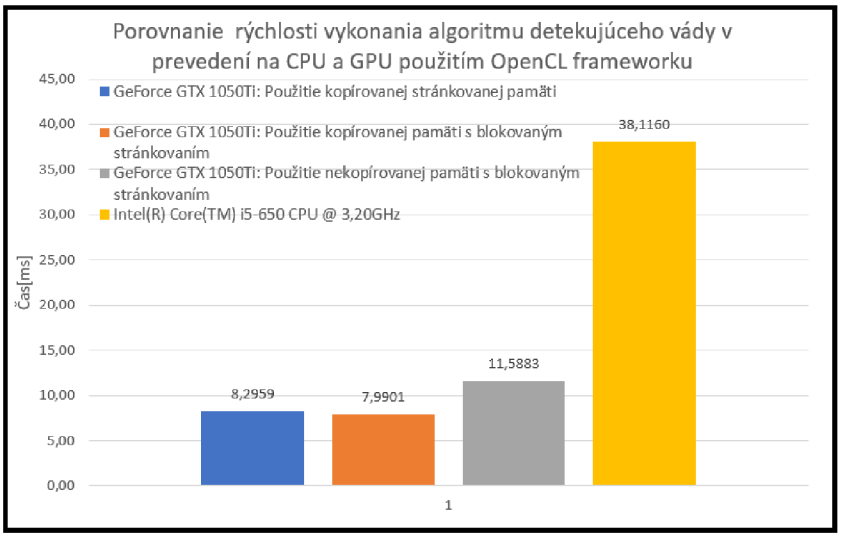
Obr. B.7: HWConfig GTX1050Ti: OpenCL: Meranie rýchlosti transferu 3 druhov pamäti z GPU do CPU.



Obr. B.8: HWConfig GTX1050Ti: OpenCL: Meranie rýchlosti oboch transferov pamäti medzi CPU a GPU.



Obr. B.9: HWConfig GTX1050Ti: OpenCL: Meranie rýchlosti vykonania kernelu.



Obr. B.10: HWConfig GTX1050Ti: OpenCL: Meranie rýchlosti vykonania celého paralelného algoritmu + porovnanie so sekvenčným.

C OBSAH PRILOŽENÉHO CD

Ako sme spomenuli vyššie, aplikácia **DefectDetection.exe** sa počas behu programu linkuje s dvoma DLL súbormi - **useCUDA.dll** a **useOpenCL.dll**. Na CD sú uložené projekty v Microsoft Visual Studio pre oba DLL súbory. Rovnako tam najdeme projekt **Startup**, ktorý slúžil na testovanie oboch DLL súborov.

```
root
├── Filip_Sedláček-DP.pdf
├── opencv
├── DefectDetection.exe
├── useCUDA.dll
├── useOpenCL.dll
├── Testovacie_snímky_textilu
│   └── <image01 - image54>.tif
├── DefectDetection (Projekt Microsoft Visual Studio)
│   ├── DefectDetection.sln
│   ├── DefectDetection
│   │   └── zdrojové súbory
│   └── x64
│       ├── useCUDA.dll
│       ├── useCUDA.lib
│       ├── useOpenCL.dll
│       └── useOpenCL.lib
├── Startup (Projekt Microsoft Visual Studio)
│   ├── Startup.sln .3 Startup
│   │   └── zdrojové súbory
│   └── x64
│       ├── useCUDA.dll
│       ├── useCUDA.lib
│       ├── useOpenCL.dll
│       └── useOpenCL.lib
├── useCUDA (Projekt Microsoft Visual Studio)
│   ├── useCUDA.sln
│   ├── useCUDA
│   │   └── zdrojové súbory
│   └── x64
│       ├── useCUDA.dll
│       └── useCUDA.lib
└── useOpenCL (Projekt Microsoft Visual Studio)
    ├── useOpenCL.sln
    ├── useOpenCL
    │   └── zdrojové súbory
    └── x64
        ├── useOpenCL.dll
        └── useOpenCL.lib
```