Programovanie akcelerátorov GPGPU – CUDA

Vznik GPGPU bol v počiatkoch motivovaný vývojom grafických kariet. V 80-tych a začiatkom 90-tych rokov minulého storočia postačoval na hranie jednoduchých 2D hier výkon CPU a úlohou grafickej karty bolo zabezpečiť zobrazenie pixelov na monitore. S príchodom čoraz náročnejších hier bola požiadavka na čoraz väčší výkon procesorov CPU. V dôsledku toho boli procesory doplnené o ďalšie vektorové výpočtové jednotky s podporou špeciálnych inštrukcií, ako napríklad MMX, SSE, AVX a ďalších. Avšak v tomto období nastal prielom aj v oblasti grafických kariet a objavili sa prvé GPU (Graphics Processing Unit). Ich úlohou bolo odľahčiť prácu CPU tým, že zložité matematické výpočty s pohyblivou desatinnou čiarkou prevzali na seba GPU. To umožnilo podstatné zrýchlenie mnohých aplikácií vrátane hier a rozvoj 3D počítačovej grafiky [1–3].

Koncom minulého storočia boli GPU schopné pracovať s 3D obrazom reprezentovaným vo forme trojuholníkov. Tieto trojuholníky vedeli GPU transformovať v aritmetike s pohyblivou desatinnou čiarkou a následne na ne mapovať rôzne textúry. Takto vytvorený obraz bol následne prevedený na 2D reprezentáciu pixelov zobrazených na monitore. Potenciál týchto zariadení neostal nepovšimnutý vedeckou komunitou. Pomocou využitia rozhrania OpenGL sa pokúsili pretransformovať vedecký problém na problém počítačovej grafiky, ktorý by bola schopná GPU riešiť. Toto riešenie nebolo príliš efektívne, avšak vďaka nemu si výrobcovia GPU uvedomili, že cieľová skupina záujemcov o GPU môže byť podstatne širšia ako len hráči hier.

Spoločnosť Nvidia, ako jedna z najväčších producentov GPU, prišla s myšlienkou prechodu na GPGPU (General Purpose GPU), čím umožnila programátorom ľahší prístup k používaniu tohto hardvéru na rôzne typy výpočtov. V roku 2007 spoločnosť Nvidia predstavila platformu a rozhranie **CUDA** (Comupte Unified Device Architecture) [4]. Vďaka nemu bolo možné vytvoriť program, ktorý používa GPGPU bez nutnosti, aby programátor musel mať špeciálne poznatky z oblasti počítačovej grafiky. Následne o dva roky neskôr sa objavila alternatíva vo forme **OpenCL** rozhrania [5], ktoré umožňuje vytvárať porovnateľný program aj pre GPGPU iných výrobcov, ako sú Intel, AMD a ďalší. Okrem toho vznikli aj rôzne ďalšie jazyky a rozhrania pre programovanie na GPGPU. Uvedieme aspoň niektoré z nich: OpenACC [6, 7], DirectCompute [8].

Filozofia používania GPU pre výpočty zostala od počiatku rovnaká a GPU nebola nikdy vnímaná ako plnohodnotný procesor, ale ako pomocný koprocesor, ktorý vykonáva svoju prácu v spolupráci s hostiteľským CPU. Preto aj paralelné programy využívajúce výpočtový výkon GPGPU pozostávajú z dvoch častí. Hostiteľskej časti, ktorá je vykonávaná na CPU a časti zariadenia, ktorá je vykonávaná na GPGPU.

V nasledujúcej časti sa pokúsime zhrnúť niekoľko základných poznatkov potrebných pre začínajúcich programátorov CUDA paralelných programov. Táto problematika je však oveľa komplexnejšia, ako z pohľadu vytvárania CUDA programov, tak aj z pohľadu ich ladenia a efektivity. Preto odporúčame čitateľovi štúdium ďalších dokumentov a manuálov zo sekcie https://developer.nvidia.com/CUDA-toolkit alebo iných zdrojov.

1 Jednoduchý program pre GPGPU

Pri písaní programu v prostredí CUDA máme na výber z dvoch možnosti API:

- CUDA runtime API,
- CUDA driver API.

Prvá možnosť je rozhranie na vyššej úrovni, čo znamená, že vytváranie programov s použitím CUDA runtime API je zväčša jednoduchšie, pretože poskytuje implicitnú inicializáciu, manažovanie kontextu a modulov. Pri preklade zdrojového kódu v jazyku C prekladačom `nvcc` je kernel súčasťou jedného spustiteľného súboru. Naopak pri použití CUDA driver API je potrebné písať viac a zložitejšieho zdrojového kódu, avšak máme možnosť lepšej kontroly nad vykonávaním CUDA programu. CUDA driver API nie je závislé na určitom programovacom jazyku, pretože narábame iba s cubin objektami. Vo všeobecnosti je vhodné začať najprv s používaním CUDA runtime API a až po nadobudnutí určitých skúseností a pri potrebe lepšej kontroly nad používaním GPGPU je vhodné použiť rozhranie CUDA driver API.

Skôr, ako začneme vytvárať svoj vlastný program v prostredí CUDA, je vhodné získať niekoľko informácií o zariadení alebo zariadeniach GPGPU, ktoré budeme používať. Súčasťou vývojového prostredia CUDA SDK (Software Development Kit) je aj sada vzorových príkladov. K tomuto účelu nám poslúži hotový príklad `deviceQuery` alebo `deviceQueryDrv`. Na základe týchto informácií môžme lepšie prispôsobiť návrh paralelného programu a zvýšiť jeho kompatibilitu s rôznymi architektúrami GPGPU. Výstup programu obsahuje rôzne užitočné informácie a môže vyzerať nasledovne:

```
./devicequery starting...
CUDA Device Query (Runtime API) version (CUDART static linking)
Detected 1 CUDA Capable device(s)
Device 0: "Tesla K20m"
Conc 0: 1
Async 0: 2
  CUDA Driver Version / Runtime Version          8.0 / 8.0
  CUDA Capability Major/Minor version number:    3.5
  Total amount of global memory:                 4742 MBytes (4972412928 Bytes)
  (13) Multiprocessors x (192) CUDA Cores/MP:    2496 CUDA Cores
  GPU Clock rate:                                706 MHz (0.71 GHz)
  Memory Clock rate:                             2600 MHz
  Memory Bus width:                              320-bit
```

```
  L2 Cache Size:                                 1310720 bytes
  Max Texture Dimension Size (x,y,z)             1D=(65536), 2D=(65536,65536),
                                                 3D=(4096,4096,4096)
  Max Layered Texture Size (dim) x layers        1D=(16384) x 2048,
                                                 2D=(16384,16384) x 2048
  Total amount of constant memory:               65536 Bytes
  Total amount of shared memory per block:       49152 Bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     2147483647 x 65535 x 65535
  Maximum memory pitch:                          2147483647 Bytes
  Texture alignment:                             512 Bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     no
  Integrated GPU sharing host memory:            no
  Support host page-locked memory mapping:       Yes
  Alignment requirement for surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports unified addressing (uva):      Yes
  Device PCI bus ID / PCI location ID:           32 / 0
  Compute Mode:
Default (multiple host threads can use ::cudaSetDevice() with device simultaneously)
```

Ako sme už uviedli, CUDA program pozostáva z dvoch častí. Hostiteľská časť programu môže byť v podstate obyčajný program napísaný v jazyku C, prípadne v inom jazyku. Druhú časť programu predstavujú funkcie, ktoré budú vykonávané na zariadení GPGPU. Funkciu zariadenia, ktorá má byť spustiteľná z hostiteľskej časti programu, označujeme ako **kernel** a definujeme ju pomocou špecifikátora `__global__` použitého pri definícii funkcie kernelu. Okrem kernelu je možné definovať aj tzv. funkcie zariadenia pomocou špecifikátora `__device__`, ktoré je možné volať a vykonávať len v rámci časti programu vykonávanej na zariadení GPGPU. Posledným typom funkcií sú funkcie definované pomocou špecifikátora `__host__`. Tieto je možné volať a vykonávať iba v rámci hostiteľskej časti programu. Vo funkciách pre zariadenie GPGPU je možné použiť:

- všetky matematické operátory,
- riadiace štruktúry (if, for, while, case, goto),
- transcendentálne matematické funkcie s jednoduchou presnosťou,
- volanie `__device__` funkcií
- ukazovatele,
- štruktúry a statické polia,
- vstavané funkcie CUDA,
- šablóny v C++,
- preťažovanie funkcií v C++,
- funkcie z knižníc.

V závislosti od verzie zariadenia a jeho výpočtových možností (compute capability – CC) a verzie CUDA je možné použiť:

- reálne čísla s dvojnásobnou presnosťou a matematické funkcie (CC $\geq$ 1.3), avšak aritmetika s dvojnásobnou presnosťou je 2 až 24-krát pomalšia v závislosti od architektúry GPGPU, než aritmetika s jednoduchou presnosťou,
- rekurzívne volanie funkcií (CC $\geq$ 2.0),
- ukazovatele na funkcie (CC $\geq$ 2.0, CUDA $\geq$ 3.1),
- nepolymorfické triedy v C++ (CC $\geq$ 2.0),
- funkciu `printf` (CC $\geq$ 2.0, CUDA $\geq$ 3.1),
- funkcie `malloc` a `new` (CC $\geq$ 2.0, CUDA $\geq$ 3.2),
- polymorfické triedy v C++ (CC $\geq$ 2.0, CUDA $\geq$ 4.0),

V rámci časti programu vykonávanej na zariadení GPGPU nie je možné použiť:

- polia s dynamicky určenou veľkosťou
- systémové volania, I/O, manažment pamäte,
- reálne čísla typu `long double`,
- statické premenné deklarované vo funkciách zariadenia.

Takto vytvorený kernel je potom možné zavolať v hostiteľskej časti programu pomocou špeciálnej syntaxe:

```
<<< Dg, Db, Ns, S >>>
```

pričom medzi trojicou zátvoriek uvádzame informácie o počte a organizácii vlákien v blokoch, o počte a organizácii blokov v gride a okrem toho môžme doplniť aj nepovinné informácie. Rozmer a geometriu je možné definovať pomocou špeciálneho typu `dim3`, ktorý je odvodený od typu `uint3` a obsahuje tri komponenty `x`, `y` a `z`. V prípade, že niektorý z komponentov nebude špecifikovaný, použije sa prednastavená hodnota rovná 1. Parameter `Dg` je typu `dim3` a udáva dimenziu gridu (počet blokov) v smeroch `Dg.x`, `Dg.y` a `Dg.z`. Parameter `Db` je typu `dim3` a udáva dimenziu bloku (počet vlákien) v smeroch `Db.x`, `Db.y` a `Db.z`. Premenné `Dg` a `Db` je možné definovať a inicializovať nasledovne:

```
dim3 Dg(16, 8, 4);
dim3 Dg(64, 32);
```

V druhom prípade bude rozmer `Dg.z` rovný 1. Ďalšie dva parametre sú nepovinné. Parameter `Ns` je typu `size_t` a určuje počet bajtov zdieľanej pamäte, ktorá má byť pre blok dynamicky alokovaná popri staticky alokovanej pamäti. Túto pamäť je možné využiť pri definovaní premenných typu `__shared__`. Prednastavená hodnota je rovná 0. Parameter `S` je typu `cudaStream_t` a určuje prúd, v rámci ktorého má byť kernel vykonávaný. Prednastavená hodnota je rovná 0, čo znamená, že všetky kernely sú vykonávané v tom istom prúde za sebou. V prípade, že niektorá u premenných prekročí dovolený rozmer alebo množstvo dostupnej zdieľanej pamäte, program skončí s chybou.

Pripomeňme si, že procesor GPGPU pozostáva z veľkého množstva viacvláknových prúdových multiprocesorov (SM). Pri zavolaní ker-

nelu z hostiteľskej časti programu sa vytvorí požadovaný grid blokov a tieto bloky sa priradia jednotlivým multiprocesorom. Vlákna v rámci jedného bloku sú vykonávané paralelne jedným multiprocesorom. Jednotlivé bloky vlákien môžu byť vykonávané paralelne rôznymi multiprocesormi. Po dokončení všetkých vlákien z bloku začne multiprocesor vykonávať ďalší blok [4].

Tieto multiprocesory sú navrhnuté tak, aby boli schopné vykonávať stovky vlákien súbežne, pričom sú manažované v duchu SIMT (Single Instruction, Multiple Thread) architektúry. Vykonávané inštrukcie sú prúdovo spracované, čo umožňuje paralelizmus v rámci vlákna. Paralelizmus na úrovni vlákien vykonávaných priamo hardvérom je samozrejmý. Na rozdiel od CPU, procesor GPGPU nie je schopný predpovedania vetvenia a špekulácie.

Model vykonávania programu CUDA spočíva v tom, že multiprocesor vytvára, manažuje, plánuje a vykonáva vlákna v zväzkoch po 32 paralelných vlákien označovaných ako **warp**. Vlákna v rámci jedného zväzku sú spustené súčasne na rovnakej adrese programu, pričom majú vlastné počítadlo inštrukčných adries a stav registrov, čo im umožňuje nezávislé vetvenie. Vlákna sú do zväzkov rozdeľované vždy v poradí s rastúcim identifikátorom vlákna, počnúc vláknom s ID rovným 0.

Všetky vlákna v zväzku musia vykonávať tú istú inštrukciu. To znamená, že najväčšiu efektívnosť dosiahneme, ak všetky vlákna idú rovnakou cestou v programe. Ak by nastala situácia, že sa vlákna na základe nejakej hodnoty začnú vetviť, najprv bude všetkými vláknami vykonaná prvá vetva a následne opäť všetkými vláknami druhá vetva.

Je potrebné si uvedomiť, že ideálna veľkosť bloku vlákien by nemala byť menšia než 32, pretože vykonávanie celého zväzku trvá rovnako dlho bez ohľadu na to, či je v ňom 32 alebo menej vlákien. Preto by ideálna veľkosť bloku mala byť násobkom 32, maximálne do limitu 1024 vlákien na blok. Taktiež je potrebné uvedomiť si, že moderné GPGPU disponujú viacerými multiprocesormi, takže môžu vykonávať niekoľko blokov vlákien paralelne. To znamená, že rozdelenie potrebného počtu vlákien do blokov a gridu by malo zabezpečiť plné využitie všetkých dostupných multiprocesorov. Okrem toho novšie GPGPU umožňujú vykonávať viacero nezávislých programov súčasne.

CUDA program je možné napísať pomocou ľubovoľného textového editora alebo v integrovanom vývojovom prostredí (IDE). Zdrojový súbor programu CUDA má `.cu` príponu.

Úloha: Vyskúšajte program skompilovať a spustiť pre rôzne hodnoty premenných gs a bs.

Na nasledujúcom programe 1 si ukážeme, akým spôsobom je možné vytvoriť kernel vykonávaný na zariadení GPGPU. V CUDA programoch od verzie výpočtových možností 2.0 (Compute Capability – CC) a verzie CUDA 3.1 je možné v časti programu vykonávaním na GPGPU používať výpis na štandardný výstup pomocou funkcie `printf`, čo značne uľahčuje prácu pri ladení a testovaní CUDA programu.

Pri návrhu CUDA paralelných programov väčšinou vychádzame z údajovej dekompozície. To znamená, že každé vlákno vykonáva tie

isté operácie, ale nad inými údajmi. Na rozlíšenie toho, nad ktorými údajmi má vlákno operovať je možné použiť vstavanú premennú – identifikátor vlákna `threadIdx` v rámci bloku vlákien. Keďže vlákna môžu byť organizované v jednom, dvoch alebo troch rozmeroch, táto premenná má v sebe tri komponenty (`threadIdx.x`, `threadIdx.y` a `threadIdx.z`). Analogicky vieme identifikovať aj blok vlákien v rámci gridu pomocou premennej `blockIdx`. Vlákna a bloky sú číslované zaradom vo všetkých rozmeroch od čísla 0.

Okrem identifikačného čísla vlákna a bloku vieme pomocou vstavaných premenných zistiť dimenziu bloku a gridu. Premenné opäť obsahujú tri komponenty odpovedajúce trom rozmerom. Pre dimenziu bloku vlákien je to premenná `blockDim` a pre dimenziu gridu je to premenná `gridDim`.

Po zavolaní kernelu bude hostiteľský program pokračovať vo vykonávaní ďalších príkazov. Toto môžme využiť na súčasné vykonávanie úloh na CPU a GPGPU. V prípade, že potrebujeme zabezpečiť, aby hostiteľský program na určitom mieste počkal na dokončenie časti programu vykonávanej na GPGPU, je možné zavolať funkciu `cudaDeviceSynchronize`. Pomocou tejto funkcie je možné obe časti programu synchronizovať.

Zdrojový kód 1: CUDA program identifikujúci vlákna

```
#include <stdio.h>
#include <cuda.h>

__global__ void identify()
{
  printf("blok: x=%d y=%d z=%d\tvlakno: x=%d y=%d z=%d\n",
    blockIdx.x, blockIdx.y, blockIdx.z, threadIdx.x,
    threadIdx.y, threadIdx.z);
  printf("griddim: x=%d y=%d z=%d\tblokdim: x=%d y=%d z=%d\
    n", gridDim.x, gridDim.y, gridDim.z, blockDim.x,
    blockDim.y, blockDim.z);
}

int main(int argc, char* argv[])
{
  //definícia dimenzie gridu
  dim3 gs(2,2);
  //definícia dimenzie bloku
  dim3 bs(32);
  //volanie kernelu
  identify<<<gs, bs>>>();
  //synchronizácia - dokončenie kernelu
  cudaDeviceSynchronize();

  return(0);
}
```

Kompilovanie a spúšťanie programov pre GPGPU

Na kompilovanie zdrojového kódu, ktorý obsahuje zmes zdrojového kódu pre CPU a GPGPU je možné kompilovať naraz pomocou prekladača `nvcc`. Keďže používané GPGPU nemusia byť rovnakej výpočtovej schopnosti (niektoré funkcionality a inštrukcie podporujú iba GPGPU s vyššou výpočtovou schopnosťou), pri preklade je potrebné určiť, pre ktorú verziu sa má program skompilovať.

Uvedené parametre prekladu je možné nastaviť pomocou prepínačov prekladača `nvcc`. Pomocou prepínača `-code` je možné určiť, pre ktorú architektúru GPGPU má byť vygenerovaný binárny kód (objekt cubin) kompatibilný. Pomocou prepínača `-arch` je možné určiť výpočtovú schopnosť cieľového zariadenia GPGPU. Oba parametre nastavené predchádzajúcimi prepínačmi je možné skombinovať pomocou prepínača `-gencode`. Takže výsledný príkaz, pomocou ktorého môžme program skompilovať, môže byť jeden z nasledujúcich riadkov:

```
nvcc soft.cu -gencode arch=compute_35,code=sm_35 -o soft.x
nvcc soft.cu -arch=compute_35 -code=sm_35 -o soft.x
```

alebo skrátene:

```
nvcc soft.cu -arch=sm_35 -o soft.x
```

Takto skompilovaný program bude možné spustiť na GPGPU s výpočtovou schopnosťou 3.5 a vyššou. Programy skompilované pre nižšie hodnoty výpočtovej schopnosti je možné spustiť na GPGPU s vyššou hodnotou výpočtovej schopnosti. Zoznam podporovaných hodnôt je možné získať z manuálu prekladača.

Skompilovaný program je možné spustiť bežným spôsobom, napríklad použitím nasledujúceho príkazu:

```
./soft.x
```

2 Pamäť zariadenia GPGPU

Ukázali sme si, ako vieme navzájom rozlíšiť jednotlivé vlákna a ako vieme určiť, ktorú časť údajov má spracovať ktoré vlákno. Avšak nevyhnutným predpokladom k tomu, aby údaje mohli byť na GPGPU spracované je to, aby sa nachádzali v pamäti zariadenia GPGPU. V nasledujúcej časti sa budeme zaoberať spôsobom prenosu údajov medzi hostiteľskou pamäťou RAM a pamäťou zariadenia GPGPU.

Alokácia pamäte

Prv než môžeme začať prenášať údaje medzi pamäťami, je nevyhnutné v pamäti zariadenia GPGPU alokovať potrebnú pamäť, do ktorej majú byť údaje uložené.

Funkcia cudaMalloc

```
cudaError_t cudaMalloc(void **devPtr, size_t size)
```

Poznámka: Na alokáciu pamäte je možné použiť aj funkcie cudaMallocArray, cudaMalloc3D, cudaMalloc3DArray, cudaMallocPitch.

Funkcia slúži na dynamické alokovanie pamäte v globálnej pamäti zariadenia. Množstvo alokovanej pamäte je možné zadať v bajtoch pomocou argumentu `size`. Funkcia vráti pomocou argumentu `devPtr` ukazovateľ na adresu alokovaného miesta v pamäti zariadenia GPGPU. Funkciu je možné zavolať z hostiteľskej časti programu aj z časti programu zariadenia GPGPU.

Funkcia cudaFree

```
cudaError_t cudaFree(void *devPtr)
```

Poznámka: Na uvoľnenie pamäte je možné použiť aj funkciu cudaFreeArray.

Funkcia slúži na uvoľnenie dynamicky alokovanej pamäte na adrese určenej ukazovateľom v argumente `devPtr`. Funkciu je možné zavolať z hostiteľskej časti programu aj z časti programu zariadenia GPGPU.

Funkcia cudaMallocHost

```
cudaError_t cudaMallocHost(void **devPtr, size_t size)
```

Funkcia slúži na dynamické alokovanie nestránkovateľnej pamäte v hostiteľskej pamäti. Takto alokovanú hostiteľskú pamäť je možné adresovať priamo zo zariadenia. Okrem toho prenos údajov do zariadenia z takejto pamäte je rýchlejší v porovnaní s pamäťou alokovanou funkciou `malloc`. Množstvo alokovanej pamäte je možné zadať v bajtoch pomocou argumentu `size`. Funkcia vráti pomocou argumentu `devPtr` ukazovateľ na adresu alokovaného miesta v pamäti zariadenia GPGPU.

Funkcia cudaFreeHost

```
cudaError_t cudaFreeHost(void *devPtr)
```

Funkcia slúži na uvoľnenie dynamicky alokovanej hostiteľskej pamäte na adrese určenej ukazovateľom v argumente `devPtr`.

Prenos údajov

Na zariadení GPGPU nie je možné vykonávať I/O operácie, a preto je potrebné, aby bolo možné údaje, ktoré sa majú spracovať a výsledky presúvať medzi hostiteľskou pamäťou a pamäťou zariadenia GPGPU.

Prenos údajov medzi pamäťami hostiteľskou a zariadenia sa uskutočňuje pomocou zbernice PCIe. To znamená, že tento prenos je podstatne pomalší než priamy prístup do pamäte. V prípade, že výpočtová náročnosť časti programu, ktorá má byť vykonaná na GPGPU nie je dostatočne veľká, môže sa stať, že čas, ktorý ušetríme paralelným spracovaním údajov na GPGPU v porovnaní so spracovaním na CPU

bude kratší, ako čas potrebný na prenos údajov medzi pamäťami hostiteľskou a zariadenia GPGPU. Toto riziko je potrebné vždy vopred dobre zvážiť. Z uvedeného dôvodu je potrebné čo najviac obmedziť množstvo prenášaných údajov, prípadne čo najviac prenosu údajov vykonávať súbežne s vykonávaním výpočtov na zariadení GPGPU.

Funkcie typu `__global__` nemôžu vracať výsledok ako návratovú hodnotu, a teda návratový typ funkcie musí byť `void`. Naopak kernelu je možné predať určité informácie s použitím argumentov funkcie. Takýmto spôsobom je možné predať ukazovatele na polia za predpokladu, že tieto sa nachádzajú v rámci jednotného adresného priestoru (pamäť zariadenia GPGPU alebo nestránkovateľná hostiteľská pamäť). Ďalej argumentami funkcie kernelu môžu byť aj skalárne hodnoty, napríklad typu `int`, `float`, ..., `struct`, ktoré sú predávané hodnotou. Pre ukladanie týchto argumentov nie je potrebné špeciálne alokovať miesto v pamäti zariadenia. Predané hodnoty argumentov sú uložené v zdieľanej pamäti.

Keďže cez argumenty funkcie kernel nie je možné predať všetky hodnoty uložené v zložitejších údajových štruktúrach ako je pole alebo matica, tieto je potrebné explicitne prekopírovať medzi pamäťami hostiteľskou a zariadenia GPGPU.

Funkcia cudaMemcpy

```
cudaError_t cudaMemcpy(void *dst, const void *src,
            size_t count, enum cudaMemcpyKind kind)
```

Funkcia slúži na prekopírovanie údajov z jedného umiestnenia v pamäti určeného ukazovateľom `src` na druhé miesto v pamäti určené ukazovateľom `dst`. Funkcia prekopíruje určitý počet bajtov zadaný pomocou argumentu `count`. Smer kopírovania určuje posledný argument `kind`, ktorý môže nadobúdať jednu z hodnôt:

Poznámka: Na prekopírovanie údajov je možné použiť aj funkcie cudaMemcpy2D, cudaMemcpy2DToArray, cudaMemcpy2DFromArray, cudaMemcpy2DArrayToArray.

- `cudaMemcpyHostToHost`,
- `cudaMemcpyHostToDevice`,
- `cudaMemcpyDeviceToHost`,
- `cudaMemcpyDeviceToDevice`.

Okrem toho je možné použiť aj hodnotu `cudaMemcpyDefault`, pri ktorej je smer kopírovania určený na základe adries v ukazovateľoch.

Táto funkcia v sebe obsahuje implicitnú bariéru, ktorá zabezpečí, že vykonávanie programu bude pokračovať až po dokončení kopírovania údajov.

Funkcia cudaMemcpyAsync

```
cudaError_t cudaMemcpyAsync(void *dst, const void *src,
  size_t count, cudaMemcpyKind kind, cudaStream_t stream)
```

Funkcia má podobný význam ako funkcia `cudaMemcpy` s tým rozdielom, že kopírovanie údajov sa vykonáva asynchrónne. To znamená, že po zavolaní funkcie sa pokračuje vykonávaním ďalších príkazov.

Význam argumentov je rovnaký ako pri funkcii `cudaMemcpy`, pričom pribudol posledný argument `stream`, ktorý určuje prúd, v rámci ktorého sa má prenos údajov vykonať.

Asynchrónny prenos údajov je možné využiť zvlášť pri tých GPGPU, ktoré umožňujú súčasné vykonávanie výpočtov a kopírovanie údajov. Takto je možné významne skrátiť čas potrebný na vykonanie časti programu na zariadení GPGPU.

Riešená úloha

V nasledujúcej úlohe sa budeme zaoberať problémom sčítania dvoch vektorov s použitím zariadenia GPGPU. Prvý a najjednoduchší prípad spočíva v tom, že počet použitých vlákien organizovaných v jednorozmernom bloku vlákien bude odpovedať počtu prvkov vektorov. V takomto prípade bude hodnota `threadIdx.x` priamo reprezentovať indexy prvkov vektora. Riešenie je uvedené ako program 3.

Takéto riešenie je však postačujúce len v prípade, že počet prvkov vektora nie je väčší ako maximálny počet vlákien v rámci bloku. Toto riešenie okrem toho nie je efektívne, pretože sčítanie všetkých prvkov vektora sa vykonávalo iba v jednom bloku, a teda s použitím iba jedného multiprocesora.

Na riadkoch 5-8 vidíme definíciu funkcie kernelu. V programe budeme potrebovať dve sady ukazovateľov na polia (riadok 12), v ktorých budú uložené vektory. Prvá trojica `hA`, `hB` a `hC` bude použitá ako ukazovatele na alokovanú hostiteľskú pamäť (riadky 14-16), zatiaľ čo druhá trojica `dA`, `dB` a `dC` bude slúžiť ako ukazovatele na alokovanú pamäť zariadenia GPGPU (riadky 23-25). Môžme si všimnúť, že vektor `C` nie je potrebné kopírovať na GPGPU a naopak vektory `A` a `B` nie je potrebné kopírovať späť z GPGPU.

Oba problémy je možné vyriešiť vytvorením viacerých blokov vlákien, pričom každému vláknu môžme zároveň prideliť niekoľko prvkov vektora na základe round-robin algoritmu. Funkcia kernelu bude potom vyzerať tak, ako je uvedené v programe 2.

Zdrojový kód 2: CUDA funkcia na sčítanie dvoch vektorov

```
1 __global__ void vecAdd(float* A, float* B, float* C)
2 {
3   for(int i = blockIdx.x * blockDim.x + threadIdx.x; i <
      size; i += blockDim.x * gridDim.x ) {
4     C[i] = A[i] + B[i];
5   }
6 }
```

Zdrojový kód 3: CUDA program na sčítanie dvoch vektorov

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#define MAX 1024
__global__ void vecAdd(float* A, float* B, float* C)
{
  C[threadIdx.x] = A[threadIdx.x] + B[threadIdx.x];
}
int main(int argc, char* argv[])
{
  //ukazovatele na vektory
  float *hA, *hB, *hC, *dA, *dB, *dC;
  //alokácia hostiteľskej pamäte
  hA = (float*) malloc(sizeof(float) * MAX);
  hB = (float*) malloc(sizeof(float) * MAX);
  hC = (float*) malloc(sizeof(float) * MAX);
  //inicializácia vstupných a výstupného vektoru
  for(int i = 0; i < MAX; i++) {
    hA[i] = hB[i] = i;
    hC[i] = 0;
  }
  //alokácia pamäte zariadenia GPGPU
  cudaMalloc(&dA, sizeof(float) * MAX);
  cudaMalloc(&dB, sizeof(float) * MAX);
  cudaMalloc(&dC, sizeof(float) * MAX);
  //kopírovanie údajov do GPGPU pamäte
  cudaMemcpy(dA, hA, sizeof(float) * MAX,
    cudaMemcpyHostToDevice);
  cudaMemcpy(dB, hB, sizeof(float) * MAX,
    cudaMemcpyHostToDevice);
  //definícia dimenzie gridu a bloku
  dim3 gs(1), bs(MAX);
  //volanie kernelu
  vecAdd<<<gs, bs>>>(dA, dB, dC);
  //synchronizácia - dokončenie kernelu
  cudaDeviceSynchronize();
  //kopírovanie údajov z GPGPU pamäte
  cudaMemcpy(hC, dC, sizeof(float) * MAX,
    cudaMemcpyDeviceToHost);
  //uvoľnenie pamäte zariadenia GPGPU
  cudaFree(dA);
  cudaFree(dB);
  cudaFree(dC);
  //výpis výsledkov
  for(int i = 0; i < MAX; i++) {
    printf("%f ", hC[i]);
  }
  //uvoľnenie hostiteľskej pamäte
  free(hA);
  free(hB);
  free(hC);

  return(0);
}
```

3 Hierarchia pamäte GPGPU

Poznámka: S použitím __managed__ špecifikátora je možné alokovať pamäť, ktorá je s určitými obmedzeniami dostupná ako z hostiteľskej časti programu, tak aj z časti programu zariadenia GPGPU.

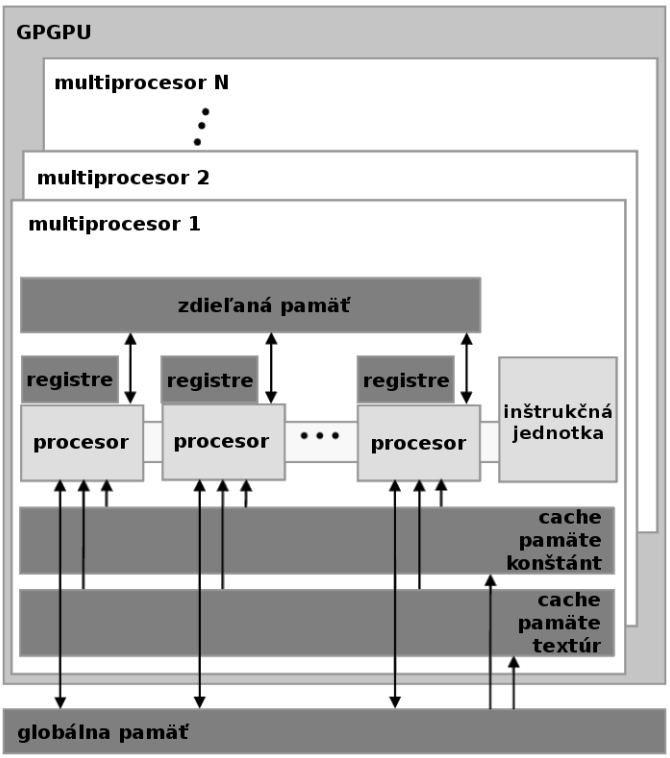
Zariadenie GPGPU disponuje viacerými druhmi pamätí, ktoré sa líšia možnosťou prístupu, rýchlosťou a veľkosťou. Schematické znázornenie môžme vidieť na obrázku 1. Hierarchiu pamäte, ku ktorej je možné zo zariadenia GPGPU pristupovať môžme rozdeliť na:

- **hostiteľská pamäť** (nestránkovateľná) – je prístupná cez zbernicu PCIe, takže prístup k údajom je pomalý,
- **globálna pamäť** zariadenia – hlavná pamäť GPGPU má kapacitu rádovo GB, údaje sú dostupné pre všetky multiprocesory, vysoká prenosová šírka (viac než 100 GB/s), vysoká latencia (niekoľko stoviek cyklov)
- **registre GPGPU** – údaje sú viditeľné len pre jedno vlákno, prístupná počas jedného cyklu, spravidla sa do nich ukladajú lokálne premenné kernelu,
- **lokálna pamäť** zariadenia – fyzicky sa nachádza v globálnej pamäti, ale údaje sú dostupné jednému vláknu,
- **zdieľaná pamäť** zariadenia – spoločná pamäť pre multiprocesor s kapacitou 16 alebo 48 kB, údaje sú viditeľné pre všetky vlákna v bloku, prístupná počas jedného cyklu, dá sa použiť ako používateľsky manažovaná vyrovnávacia pamäť pre prístup ku globálnej pamäti zariadenia a zdieľanie údajov medzi vláknami,
- **pamäť konštánt** zariadenia – špeciálna pamäť určená iba na čítanie s maximálnou veľkosťou 64 kB a 8 kB vyrovnávacou pamäťou pre multiprocesor, nie je dynamicky alokovateľná,
- **pamäť textúr** zariadenia – prístup do pamäte je pomalší ako pri zdieľanej pamäti, vhodné pre uchovanie 2D a 3D štruktúr kvôli zarovnávaniu pamäte,
- **globálna pamäť iného zariadenia GPGPU** – pomocou GPUdirect je možné pristupovať aj k údajom v pamäti iného zariadenia v rámci toho istého uzla.

Hostiteľská pamäť

Túto pamäť predstavuje hlavná pamäť počítača RAM, ktorá sa používa v rámci hostiteľskej časti programu procesorom CPU. Zariadenie GPGPU do tejto pamäte môže pristupovať priamo iba v prípade, ak je alokovaná ako nestránkovateľná, čiže fyzické umiestnenie alokovaného bloku sa nebude meniť z dôvodu výmeny stránok pri swapovaní. Prístup do takejto pamäte zo zariadenia je však výrazne pomalší v porovnaní s globálnou pamäťou GPGPU, pretože sa uskutočňuje cez zbernicu PCIe. Takúto pamäť je možné alokovať pomocou funkcie `cudaMallocHost`.

V prípade alokovania stránkovateľnej pamäte, GPGPU nemôže priamo k údajom uloženým v tejto pamäti pristupovať. V takomto prípade je potrebné údaje najprv prekopírovať do pamäte zariadenia GPGPU alebo opačne. Takúto pamäť je možné alokovať štandardne pomocou funkcie `malloc`.



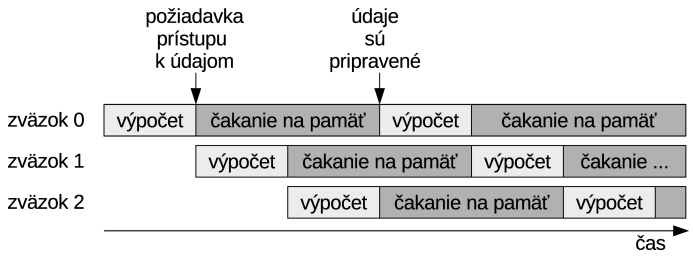
Obr. 1: Hierarchia pamäte GPGPU

Po ukončení práce je potrebné pamäť uvoľniť pomocou funkcií `cudaFreeHost` alebo `free`.

Globálna pamäť

Globálna pamäť zariadenia GPGPU predstavuje jeho hlavnú pamäť s relatívne veľkou kapacitou rádovo niekoľko GB. V prípade predávania ukazovateľov ako argumentov kernelu, tieto musia ukazovať na miesto v globálnej pamäti. Premenné, ktoré majú byť uložené v globálnej pamäti je potrebné definovať pomocou špecifikátora `__device__`. Taktiež lokálne premenné sú ukladané do globálnej pamäte v prípade nedostatočného množstva dostupných registrov.

Táto pamäť sa vyznačuje pomerne vysokou latenciou, pretože prístup k údajom, ktoré sú v nej uložené, trvá niekoľko stoviek cyklov. K tomu, aby bolo možné plne využiť celú prenosovú šírku pre prístup k údajom je potrebné vykonať niekoľko opatrení. V prvom rade je potrebné, aby počet blokov vlákien bol väčší ako počet multiprocesorov. V takomto prípade sa môže vykonávanie jednotlivých blokov prekrývať (v duchu multitaskingu). Obdobne, ak je počet vlákien v bloku väčší ako veľkosť zväzku (32 vlákien), vykonávanie zväzkov sa môže prekrývať ich striedavým vykonávaním. Vďaka tomuto je možné efektívne skrývať latenciu globálnej pamäte tak, ako je znázornené na obrázku 2. Každý multiprocesor môže striedať vykonávanie viacerých zväzkov v rámci toho istého alebo aj rôznych blokov. V prípade, že nastane požiadavka na prístup k údajom uloženým v globálnej pamäti, ktorej vybavenie môže trvať približne 500-600 cyklov, multiprocesor môže zatiaľ vykonávať vlákna z iného zväzku.



Obr. 2: Skrývanie latencie globálnej pamäte

Jeden multiprocesor dokáže takýmto spôsobom v tom istom čase manažovať 48 (architektúra Fermi) alebo 64 (architektúra Kepler) zväzkov. Podobne je limitovaný aj počet súčasne manažovaných blokov jedným multiprocesorom na 8 (architektúra Fermi) alebo 16 (architektúra Kepler) blokov. Prepínanie vykonávania medzi zväzkami je možné vykonať len v prípade, že multiprocesor disponuje dostatočným počtom registrov a zdieľanej pamäte. V opačnom prípade sa prepínanie zväzkov nemôže vykonávať.

Prístup k údajom uloženým v globálnej pamäti je možné uskutočniť s pomocou vyrovnávacej pamäte v blokoch po 128 B. Týchto 128 za sebou nasledujúcich bajtov tvorí jeden riadok pamäte cache. Je potrebné uvedomiť si, že pamäť cache nemá veľkú kapacitu a v prípade použitia 1024 vlákien v rámci multiprocesora pripadá na jedno vlákno približne iba 1/3 riadku cache pamäte. To znamená, že ak by každé vlákno chcelo pristupovať k inému riadku pamäte cache v tom istom cykle, použitie pamäte cache by bolo neefektívne.

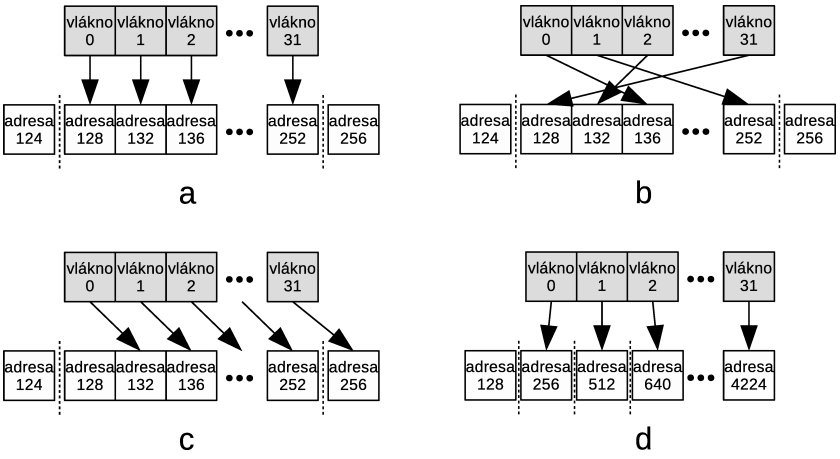
Poznámka: V architektúre Kepler je vždy prenášaný do a z pamäte cache blok 128 B bez ohľadu na to, aké množstvo z týchto údajov bude skutočne potrebných.

Naopak v prípade, ak viacero vlákien v bloku pristupuje k tomu istému riadku pamäte cache, je možné tieto prístupy združiť do jedného, čo vedie k vyššej efektívnosti programu. Počet prenesených riadkov pamäte cache v rámci jedného cyklu je obmedzený, pričom je pomalšie, ak sa vlákna vykonávané v rámci jedného zväzku snažia pristupovať k rôznym riadkom pamäte cache.

Na obrázku 3 sú znázornené 4 rôzne spôsoby prístupu vlákien v jednom zväzku k údajom v pamäti. V prípade (a) všetky vlákna pristupujú združene k tomu istému riadku pamäte cache, pričom je dodržané zarovnávanie. Takýto prístup bude vykonaný v jednej transakcii. V prípade (b) všetky vlákna pristupujú združene k tomu istému riadku pamäte cache, avšak prístup nie je zarovnaný, ale odpovedajúce prvky sú prepermutované. Takýto prístup bude vykonaný v jednej transakcii. V prípade (c) pristupujú všetky vlákna k prvkom v rade, avšak s posunutím o jeden prvok. V dôsledku toho je potrebné načítať dva riadky pamäte cache, čo bude vykonané v dvoch transakciách. V poslednom prípade (d) vlákna pristupujú k prvkom, ktoré sú v pamäti od seba vzdialené o určitý krok (napríklad 128 B). V takomto prípade bude potrebné pre každé vlákno načítať nový riadok pamäte cache, čo bude vykonané v 32 transakciách.

Registre

Každý multiprocesor disponuje určitým počtom 32-bitových registrov. Pre architektúru Fermi je to 32768 a pre architektúru Kepler je to 65536



Obr. 3: Združený a nezdružený prístup do pamäte, (a) združený prístup zarovnaný, (b) združený prístup s permutáciou, (c) nezdružený prístup nezarovnaný, (d) nezdružený prístup s krokom

registrov. Počet použitých registrov je možné určiť použitím profilera alebo s použitím prepínačov `-Xptxas -v`.

Počet použitých registrov na vlákno je možné určiť pri definícii kernelu nasledovne:

```
__global__ void __launch_bounds__(maxThreadsPerBlock,
                                  minBlocksPerMultiprocessor)
kernel(...)
{
    ...
}
```

alebo pri kompilácii programu použitím prepínača:

```
--maxrregcount počet
```

pričom minimálny počet je 16 registrov na vlákno.

Zdieľaná pamäť

Každý multiprocesor disponuje svojou vlastnou zdieľanou pamäťou veľkosti 16 alebo 48 kB v závislosti od architektúry, ktoré sú organizované v 32 rovnako veľkých moduloch, tzv. bankách, ku ktorým je možné pristupovať súčasne. Prístup do tejto pamäte majú všetky vlákna vykonávané v rámci jedného bloku na tom istom multiprocesore. Všetky požiadavky na čítanie alebo zápis údajov na n adries, pričom všetkých n adries spadá do n odlišných pamäťových bank, môže byť obslúžených súčasne, pričom prenosová šírka je rovná n-násobku prenosovej šírky jedného modulu. V prípade, že adresa dvoch požiadaviek bude prislúchať do tej istej banky, dôjde k bankovému konfliktu, ktorý zapríčiní serializáciu vykonania týchto požiadaviek. Z časového hľadiska je prístup do tejto pamäte rovnako rýchly ako prístup k údajom uloženým v registroch za podmienky, že nenastane konflikt prístupu viacerých vlákien k tej istej banke.

Zdieľaná pamäť sa zväčša používa ako používateľsky manažovaná vyrovnávacia pamäť pre prístup ku globálnej pamäti. Taktiež je možné

ju použiť na ukladanie medzivýsledkov k ďalšiemu použitiu, prípadne na zdieľanie údajov medzi vláknami v bloku.

Premenné, ktoré majú byť uložené v zdieľanej pamäti je možné definovať s použitím špecifikátora `__shared__`. V prípade definície poľa musí byť jeho veľkosť konštantná a známa už pri preklade programu.

Zdieľanú pamäť je možné alokovať aj dynamicky pomocou pamäťovej triedy `extern` a poľa s nešpecifikovanou veľkosťou nasledovne:

```
__global__ void kernel() {
extern __shared__ int dynamicSharedData[];
}
```

pričom veľkosť alokovaného miesta v bajtoch je možné určiť pri volaní kernelu nasledovne:

```
kernel<<< gs, bs, dynamicSharedMemorySize >>>();
```

Je potrebné podotknúť, že všetky dynamicky alokované polia v zdieľanej pamäti ukazujú na tú istú adresu. V prípade potreby viacerých zdieľaných polí je potrebné ich adresu upraviť ručne.

Pamäť konštánt

Poznámka: Pre zrušenie primárneho CUDA kontextu na zariadení GPGPU je možné použiť funkciu cudaDeviceReset.

Premenné, ktoré majú byť uložené v pamäti konštánt je možné definovať pomocou špecifikátora `__constant__`. Údaje uložené v tejto pamäti nie je možné meniť z časti programu vykonávanej na zariadení. Ich hodnotu je možné nastaviť pred spustením kernelu z hostiteľskej časti programu a existujú počas dĺžky existencie CUDA kontextu. Pri architektúre Fermi a Kepler sa pamäť konštánt implicitne používa na ukladanie argumentov funkcie kernelu.

Z hostiteľskej časti programu je možné manipulovať s pamäťou konštánt pomocou funkcií:
`cudaGetSymbolAddress`,
`cudaGetSymbolSize`,
`cudaMemcpyToSymbol`,
`cudaMemcpyFromSymbol`.

Pamäť textúr

Prístup k údajom uloženým v pamäti textúr je pomalší v porovnaní s prístupom k údajom v zdieľanej pamäti. Výhodou však je, že tento prístup je vždy združený a s použitím vyrovnávacej pamäte. To umožňuje efektívnu prácu so zložitejšími 2D a 3D údajovými štruktúrami. Napríklad práca s transponovanou maticou môže byť efektívnejšia, ako keby bola uložená v zdieľanej pamäti napriek tomu, že zdieľaná pamäť je rýchlejšia.

4 Synchronizácia

Už sme sa zmienili, že po zavolaní kernelu bude hostiteľský program pokračovať vo vykonávaní ďalších príkazov. Na zabezpečenie synchronizácie hostiteľskej časti programu a časti programu zariadenia GPGPU (počkať na dokončenie kernelu) je možné zavolať funkciu `cudaDeviceSynchronize`.

Synchronizáciu vlákien v bloku vlákien je možné vykonať zavolaním funkcie zariadenia `__syncthreads` v časti programu zariadenia GPGPU. Až po zavolaní tejto funkcie všetkými vláknami v bloku je možné pokračovať vo vykonávaní ďalších častí programu. V opačnom prípade hrozí riziko uviaznutia.

Okrem uvedených explicitných bariér niektoré CUDA príkazy majú v sebe implementovanú implicitnú bariéru. Pri použití zariadenia s výpočtovou schopnosťou CC < 2.0 nie je možné súčasné vykonávanie viacerých kernelov súčasne, takže druhý kernel sa začne vykonávať až po dokončení prvého. Novšie GPGPU umožňujú vykonávať až do 32 kernelov súčasne v závislosti od architektúry GPGPU. Funkcia `cudaMemcpy` neumožňuje súčasné vykonávanie kernelu a prenos údajov (okrem prenosu typu `cudaMemcpyHostToHost`). Táto funkcia blokuje aj vykonávanie hostiteľskej časti programu do momentu, kým nie je prenos údajov ukončený.

Atomické operácie

V rozhraní CUDA existuje sada špeciálnych atomických operácií. Programový model CUDA neumožňuje synchronizáciu medzi vláknami inak, ako vyššie uvedenými spôsobmi, a teda nie je možné ošetriť kritickou oblasťou prístup k zdieľanej premennej a jej modifikáciu viacerými vláknami súčasne. K tomuto účelu je možné použiť atomické operácie. Ich podstata spočíva v tom, že načítajú, modifikujú a zapíšu hodnotu jednej premennej tak, že táto činnosť nebude v kolízii s inými vláknami. Uvedieme aspoň niektoré bežne používané atomické operácie:

- `atomicAdd`,
- `atomicSub`,
- `atomicExch`,
- `atomicCAS` (Compare And Swap),
- `atomicMin` a `atomicMax`,
- `atomicInc` a `atomicDec`,
- `atomicAnd`, `atomicOr` a `atomicXor`.

Pre zabezpečenie, aby boli modifikované údaje pomocou atomických operácií viditeľné aj pre ostatné vlákna, je potrebné zavolať funkcie zariadenia `__threadfence`.

5 Efektívnosť CUDA programu

Pri návrhu efektívnych CUDA programov je potrebné sústrediť pozornosť na nasledujúce kroky [9]:

- maximalizovať paralelné vykonávanie,
- optimalizovať používanie pamäte za účelom dosiahnutia maximálnej prenosovej šírky,
- optimalizovať používanie inštrukcií za účelom spracovania maximálneho počtu inštrukcií.

V prvom rade je potrebné štruktúrovať algoritmus tak, aby ho bolo možné čo najviac paralelizovať. Následne je potrebné efektívne mapovať vykonávanie paralelných častí na hardvér dôsledným nastavením parametrov pri každom spúšťaní kernelu. K vyššej efektivite môže prispieť aj explicitné používanie viacerých prúdov pre vykonávanie inštrukcií, ako aj súčasné využívanie výkonu hostiteľského procesora a procesora zariadenia GPGPU.

Pri optimalizácii používania pamäte je v prvom rade potrebné začať minimalizovaním prenosu údajov medzi pamäťou hostiteľa a zariadenia z dôvodu jeho podstatne nižšej rýchlosti v porovnaní s prenosom údajov v rámci zariadenia GPGPU. Prístup funkcie kernelu k údajom v globálnej pamäti zariadenia je možné optimalizovať používaním zdieľanej pamäte ako vyrovnávacej pamäte. Niekedy je výhodnejšie úplne odstrániť prenos údajov aj za cenu potreby opätovného počítania potrebných údajov.

Prenosová šírka pásma sa môže dokonca rádovo líšiť v závislosti na vzore prístupu k určitému typu pamäte. Preto je potrebné organizovať pristupovanie do pamäte s ohľadom na optimálny vzor prístupu pre daný typ pamäte, ale obzvlášť pre globálnu pamäť zariadenia kvôli vysokej latencii. Neoptimálny prístup k zdieľanej pamäti sa môže prejaviť iba v prípade bankového konfliktu.

S ohľadom na optimalizáciu používania inštrukcií je potrebné vyhýbať sa používaniu aritmetických inštrukcií s nízkou priepustnosťou. Riešením je napríklad použitie nižšej presnosti všade, kde je to možné a zásadne to neovplyvní kvalitu výsledku. Druhým dôležitým aspektom je kontrola toku inštrukcií s ohľadom na SIMT povahu zariadenia.

Ďalšiu optimalizáciu CUDA programu je možné vykonať s použitím dostupných nástrojov ako sú debuger (CUDA-GDB), profiler (NVIDIA Profiler, NVIDIA Nsight Compute) alebo nástroja na kontrolu práce s pamäťou (CUDA-MEMCHECK).

Ďalšie CUDA knižnice

Pri návrhu paralelných programov je možné použiť aj existujúce CUDA knižnice. Optimalizované CUDA funkcie z týchto knižníc je možné použiť priamo prilinkovaním k existujúcemu programu alebo v kombinácii s CUDA API. K dispozícii sú nasledujúce knižnice:

- **cuBLAS** – implementácia knižnice BLAS (Basic Linear Algebra Subprograms),
- **nvJPEG** – poskytuje výkonné, GPGPU akcelerované funkcie pre dekódovanie formátu JPEG,
- **cuFFT** – implementácia knižnice FFT (Fast Fourier Transform),
- **nvGRAPH** – knižnica na prácu s grafmi umožňuje vytváranie a manipuláciu s grafmi a obsahuje sadu grafových algoritmov,
- **cuRAND** – knižnica poskytuje rozhranie pre jednoduché a efektívne generovanie kvalitných pseudonáhodných a kvázináhodných čísel,
- **cuSPARSE** – knižnica BLAS pre prácu s riedkymi maticami,
- **NPP** – knižnica obsahuje funkcie pre spracovanie 2D obrazu a spracovanie signálu,
- **Thrust** – knižnica šablón založená na STL (Standard Template Library), ktorá umožňuje jednoduchú implementáciu paralelných programov,
- **cuSOLVER** – knižnica poskytuje podobné funkcie ako knižnica LAPACK (Linear Algebra Package),
- **cuTENSOR** – knižnica pre lineárnu algebru s tenzormi, použiteľná pre počítačové videnie, hlboké učenie alebo kvantovú chémiu a výpočtovú fyziku.

Riešená úloha

V nasledujúcej úlohe sa budeme zaoberať problémom násobenia dvoch matíc A a B do výslednej matice C. Každý blok vlákien má na starosti výpočet štvorcovej podmatice C_{sub} a každé vlákno v bloku vykonáva výpočet jedného prvku matice. Pre zvýšenie efektívnosti programu sú podmatice rozdelené do malých blokov o veľkosti $block\_size$, ktoré sú následne prenášané z globálnej pamäte do zdieľanej pamäte [4].

V hostiteľskej časti programu 4 je alokované miesto pre matice v globálnej pamäti zariadenia GPGPU (riadky 7, 15, 23). Následne sú prekopírované údaje z matíc A a B z hostiteľskej pamäte do pripravenej pamäte zariadenia (riadky 9 a 17). Špecifikácia dimenzií bloku a gridu zodpovedá rozdeleniu matíc na bloky o veľkosti `BLOCK_SIZE` tak, aby bolo neskôr možné ukladať podmatice s rozmerom `BLOCK_SIZE` do zdieľanej pamäte. Po zavolaní kernelu (riadok 28) je potrebné počkať na jeho dokončenie (riadok 31). Na záver je prekopírovaná výsledná matica C späť do hostiteľskej pamäte (riadok 34) a pamäť zariadenia je uvoľnená (riadky 37-39).

V hlavnej časti programu 5, ktorá je súčasťou hostiteľskej časti programu, je dynamicky alokované miesto v hostiteľskej pamäti pre matice A, B a C (riadky 26-28). Matice A a B sú následne inicializované náhodnými číslami (riadky 34 a 35). Matica C je inicializovaná nulami (riadok 36). Následne je zavolaná hostiteľská funkcia volajúca kernel (riadok 41) a nakoniec uvoľnená pamäť (riadky 44-46).

Kľúčovou časťou kernelu (program 6) je použitie zdieľanej pamäte za účelom skrývania latencie globálnej pamäte. Na riadkoch 39 a 40 vlákna paralelne prekopírujú dané bloky matíc z globálnej pamäte do zdieľanej pamäte. Pred samotným výpočtom súčinu blokov matíc je

potrebné vlákna synchronizovať (riadok 42), aby sme mali istotu, že oba bloky sú už správne nakopírované v zdieľanej pamäti.

Zdrojový kód 4: CUDA program násobenia matíc – hostiteľská časť programu

```
1  void MatMul(const Matrix A, const Matrix B, Matrix C)
2  {
3    //alokácia pamäte zariadenia GPGPU
4    Matrix d_A;
5    d_A.width = d_A.stride = A.width; d_A.height = A.height;
6    size_t size = A.width * A.height * sizeof(float);
7    cudaMalloc(&d_A.elements, size);
8    //kopírovanie údajov do GPGPU pamäte
9    cudaMemcpy(d_A.elements, A.elements, size,
       cudaMemcpyHostToDevice);
10
11   //alokácia pamäte zariadenia GPGPU
12   Matrix d_B;
13   d_B.width = d_B.stride = B.width; d_B.height = B.height;
14   size = B.width * B.height * sizeof(float);
15   cudaMalloc(&d_B.elements, size);
16   //kopírovanie údajov do GPGPU pamäte
17   cudaMemcpy(d_B.elements, B.elements, size,
       cudaMemcpyHostToDevice);
18
19   //alokácia pamäte zariadenia GPGPU
20   Matrix d_C;
21   d_C.width = d_C.stride = C.width; d_C.height = C.height;
22   size = C.width * C.height * sizeof(float);
23   cudaMalloc(&d_C.elements, size);
24
25   //volanie kernelu
26   dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
27   dim3 dimGrid(B.width / dimBlock.x,A.height / dimBlock.y);
28   MatMulKer<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
29
30   //synchronizácia - dokončenie kernelu
31   cudaDeviceSynchronize();
32
33   //kopírovanie údajov z GPGPU pamäte
34   cudaMemcpy(C.elements, d_C.elements, size,
       cudaMemcpyDeviceToHost);
35
36   //uvoľnenie pamäte zariadenia GPGPU
37   cudaFree(d_A.elements);
38   cudaFree(d_B.elements);
39   cudaFree(d_C.elements);
40 }
```

Zdrojový kód 5: CUDA program násobenia matíc – hlavná časť programu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <time.h>

#define MAT_SIZE 4      //rozmer matice
#define BLOCK_SIZE 2    //rozmer bloku

//matice sú uložené v row-major formáte
//M(row, col) = *(M.elements + row * M.stride + col)
typedef struct {
  int width;
  int height;
  int stride;
  float* elements;
} Matrix;

int main()
{
  srand(time(NULL));
  //alokácia pamäte pre matice A, B, C
  Matrix A, B, C;
  A.width = A.height = A.stride = MAT_SIZE;
  B.width = B.height = B.stride = MAT_SIZE;
  C.width = C.height = C.stride = MAT_SIZE;
  A.elements = (float*) malloc(A.height * A.width * sizeof(
    float));
  B.elements = (float*) malloc(B.height * B.width * sizeof(
    float));
  C.elements = (float*) malloc(C.height * C.width * sizeof(
    float));

  //inicializácia matíc A, B a nulovanie C
  int i,j;
  for(i = 0; i < MAT_SIZE; i++){
    for(j = 0; j < MAT_SIZE; j++){
      A.elements[i * A.stride + j] = rand() % 10;
      B.elements[i * B.stride + j] = rand() % 10;
      C.elements[i * C.stride + j] = 0;
    }
  }

  //volanie násobenia matíc
  MatMul(A, B, C);

  //uvoľnenie pamäte
  free((void*) A.elements);
  free((void*) B.elements);
  free((void*) C.elements);

  return(0);
}
```

Zdrojový kód 6: CUDA program násobenia matíc – časť programu pre zariadenie GPGPU

```
//prečítaj hodnotu prvku matice
__device__ float GetEl(const Matrix A, int row, int col) {
  return A.elements[row * A.stride + col];
}
//zapíš hodnotu prvku matice
__device__ void SetEl(Matrix A, int row, int col, float
    value) {
  A.elements[row * A.stride + col] = value;
}
//načítaj blok matice
__device__ Matrix GetSubMat(Matrix A, int row, int col) {
  Matrix Asub;
  Asub.width    = BLOCK_SIZE;
  Asub.height   = BLOCK_SIZE;
  Asub.stride   = A.stride;
  Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row +
      BLOCK_SIZE * col];
  return Asub;
}
//kernel pre násobenie matíc
__global__ void MatMulKer(Matrix A, Matrix B, Matrix C) {
  //identifikácia bloku
  int blockRow = blockIdx.y;
  int blockCol = blockIdx.x;
  //podmatica počítaná v bloku
  Matrix Csub = GetSubMat(C, blockRow, blockCol);
  //hodnota prvku matice C počítaná vláknom
  float Cvalue = 0;
  //identifikácia vlákna
  int row = threadIdx.y;
  int col = threadIdx.x;
  //cyklus cez všetky podmatice
  for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
    //získanie parametrov podmatíc A a B
    Matrix Asub = GetSubMat(A, blockRow, m);
    Matrix Bsub = GetSubMat(B, m, blockCol);
    //zdieľaná pamäť na uloženie blokov Asub a Bsub
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    //prenos podmatíc z globálnej do zdieľanej pamäte
    As[row][col] = GetEl(Asub, row, col);
    Bs[row][col] = GetEl(Bsub, row, col);
    //synchronizácia vlákien
    __syncthreads();
    //násobenie Asub a Bsub
    for (int e = 0; e < BLOCK_SIZE; ++e)
      Cvalue += As[row][e] * Bs[e][col];
    //synchronizácia vlákien
    __syncthreads();
  }
  //zápis výslednej hodnoty prvku
  SetEl(Csub, row, col, Cvalue);
}
```

Literatúra

[1] Tolga Soyata. *GPU Parallel Program Development Using CUDA*. Chapman and Hall/CRC, 2018 (cited on page 1).

[2] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010 (cited on page 1).

[3] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016 (cited on page 1).

[4] NVIDIA. *CUDA C Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. 2019 (cited on pages 1, 5, 19).

[5] KHRONOS group. *OpenCL*. https://www.khronos.org/opencl/. 2019 (cited on page 1).

[6] OpenACC. *What is OpenACC?* https://www.openacc.org. 2019 (cited on page 1).

[7] Alistair Hart. "The OpenACC programming model". In: *Cray Exascale Research Initiative Europe, Tech. Rep* (2012) (cited on page 1).

[8] Tianyun Ni. "Direct Compute: Bring GPU computing to the mainstream". In: *GPU Technology Conference*. 2009, p. 23 (cited on page 1).

[9] NVIDIA. *CUDA C++ Best Practices Guide*. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html. 2019 (cited on page 18).