PPaDS MMXXI

Matúš Jókay, Château de la Mûre
matus.jokay@stuba.sk

uim.fei.stuba.sk/predmet/i-ppds
konzultácie elektronicky
mail, discord

Zdroj prednášky

https://docs.nvidia.com

https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf

Obsah prednášky

- Vývoj GPU
- Optimalizácia zariadenia
- Prístupy do pamäti zariadenia podľa CUDA verzie

Vývoj GPU

Prvá časť

Prehľad hw generacií NVidia

Generácia HW	Výpočtové možnosti (CC)		SP / SM je SPperSM	threads/blok threads/SM	warps/SM
2008 Tesla		GT200	240 / 30 = 8		
2010 Fermi	2.x	GF104	512 / 16 = 32	1024, 1536	48
2012 Kepler	3.x	3.0 GK104	1536 / 8 = 192	1024, 2048	64
		3.5 GK110	2880 / 15 = 192	1024, 2048	64
2014 Maxwell	5.x	5.2 GM200	3072 / 24 = 128	1024, 2048	64
2016 Pascal	6.x	6.0 GP100	3584 / 56 = 64	1024, 2048	64
2017 Volta	7.x	7.0 GV100	5120 / 80 = 64	1024, 2048	64
2018 Turing	7.5	7.5 TU102	4608 / 72 = 64	1024, 1024	32
2020 Ampere	8.x	8.6 GA102	10496/82 = 128	1024, 2048	64

Vývoj GPU

- SP, viaceré SP tvoria SM
- Viaceré SM tvoria GPC (GPU Processing Cluster)
- SM obsahuje jednotky nielen pre spracovanie textúr a INT32, ale aj FP32a FP64
- SM obsahuje TC (Tensor Core) (operácia A*B + C na maticiach 4x4)
- Zväčšuje sa L1, L2 a priepustnosť zberníc

Vývoj GPU

- Plánovač vlákien!
  - Volta zavádza PC a Stack pre každé vlákno warpu! (využitie: divergencia vykonávania kódu na úrovni warpu)
  - Predtým PC a Stack iba pre celý warp spolu
- Zvyšuje sa konkurentnosť / paralelizmus
- Vylepšovanie algoritmov plánovania vlákien (problém vyhladovenia, divergencie)

Vývoj GPU

- Na počiatku bolo možné vykonávať na GPU jediný kernel
- Postupne viacero, toho istého procesu
- Potom aj viacero, rôznych procesov
  - A to je bezpečnostné riziko!
  - Zavádza sa oddelenie adresného priestoru GPU pre jednotlivé procesy
  - Ampere umožňuje rozdeliť hw zdroje GPU na 7 plne nezávislych častí

Vývoj GPU

- Z pohľadu aplikačného programátora sa rozhranie neustále zjednodušuje

- Príklad: alokácia/uvoľňovanie pamäte pomocou “klasických” funkcií OS: malloc() a free()
- Zavedenie jednotnej pamäťovej technológie (Unified Memory Technology)

Vývoj GPU

- Pre paralelné výpočty je často dôležitá spolupráca vlákien pri dosahovaní výsledku
- Zavádza sa koncept Kooperatívnych skupín vlákien (Cooperative Groups v CUDA 9.x)
- Kým neprišiel tento koncept, jestvovala jediná možnosť synchronizácie vlákien bloku: __syncthreads()
- Niekedy je potrebné synchronizovať menej vlákien než je celý blok

```
__global__ void cooperative_kernel(...)
{

    // obtain default "current thread block" group
    thread_group my_block = this_thread_block();

    // subdivide into 32-thread, tiled subgroups
    // Tiled subgroups evenly partition a parent group into
    // adjacent sets of threads - in this case each one warp in size
    thread_group my_tile = tiled_partition(my_block, 32);

    // This operation will be performed by only the
    // first 32-thread tile of each block
    if (my_block.thread_rank() < 32) {
        …
        my_tile.sync();
    }
}
```

Vývoj GPU

- Spájanie viacerých GPU
  - V rámci jedného “zariadenia” (grafická karta)
  - V rámci počítača (SLI)

- IO (RTX IO)
  - Prenos údajov medzi NVMe SSD a GPU

NVIDIA DGX-1 System Specifications

Specification	DGX-1 (Tesla P100)	DGX-1 (Tesla V100)
GPUs	8x Tesla P100 GPUs	8x Tesla V100 GPUs
TFLOPS	170 (GPU FP16) + 3 (CPU FP32)	1 (GPU Tensor PFLOP) + 3 (CPU FP32)
GPU Memory	16 GB per GPU / 128 GB per DGX-1 Node	16GB per GPU/128 GB per DGX-1 Node
CPU	Dual 20-core Intel® Xeon® E5-2698 v4 2.2 GHz	Dual 20-core Intel® Xeon® E5-2698 v4 2.2 GHz
FP32 CUDA Cores	28,672	40,960
Tensor Cores	--	5120
System Memory	Up to 512MB 2133 MHz DDR4 LRDIMM	Up to 512MB 2133 MHz DDR4 LRDIMM
Storage	4x 1.92TB SSD RAID 0	4x 1.92TB SSD RAID 0
Network	Dual 10 GbE, 4 IB EDR	Dual 10 GbE, 4 IB EDR
System Weight	134 lbs	134 lbs
System Dimensions	866 D x 444 W x 131 H (mm)	866 D x 444 W x 131 H (mm)
Packing Dimensions	1180 D x 730 W x 284 H (mm)	1180 D x 730 W x 284 H (mm)
Power	3200 W (Max). Four 1600 W load-balancing power supplies (3+1 redundant), 200-240 V(ac), 10 A	3200 W (Max). Four 1600 W load-balancing power supplies (3+1 redundant), 200-240 V(ac), 10 A
Operating Temperature Range	10 - 35°C	10 -35°C

Unified Memory

Unified Memory

- Súčasť programovacieho modelu CUDA
- Od verzie CC 6.0

- Definuje menežovaný (managed) pamäťový priestor, v rámci ktorého všetky procesory (CPU) “vidia” jednotnú súvislú pamäťovú oblasť bez rozlišovania typu zariadenia pri prístupe do pamäte

Unified Memory

- Nie je potrebné používať cudaMemcpy*() funkcie

- Čas vykonávania kódu sa ovšem nezníži, pretože stále sa musia údaje presúvať medzi hlavnou pamäťou a pamäťou grafickej karty
- Zlepšuje sa čitateľnosť kódu, znižuje sa jeho zložitosť

Unified Memory

- V programe je možné využívať ukazateľ bez ohľadu na to, kde sa údaje nachádzajú

- Nie je potrebné volať funkcie na explicitnú migráciu údajov medzi rôznymi pamäťami
- Akákoľvek alokácia vytvorená pomocou menežovanej pamäte automaticky migruje údaje tam, kam patria

Unified Memory

- Alokácia menežovanej pamäte sa robí dvoma spôsobmi
  - Volaním cudaMallocManaged(), ktorá má rovnaké rozhranie ako cudaMalloc()
  - Definovaním globálnej __managed__ premennej, ktorá má podobný význam ako __device__ premenná

Unified Memory

- Model jednotnej pamäte je v CUDA prítomný od verzie CC 3.0

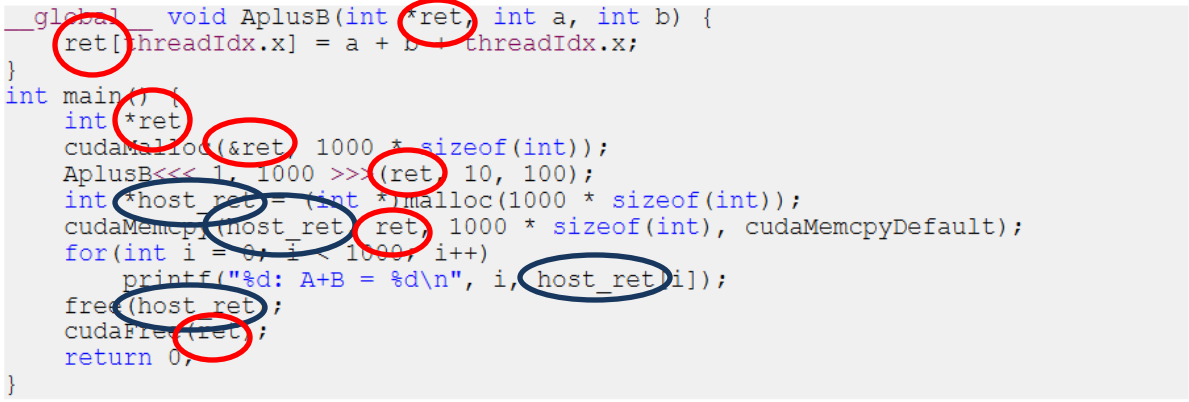
- Avšak až od verzie CC 6.0 je možné použiť na (de)alokáciu pamäte funkcie OS (malloc()/free())

```cpp
__global__ void AplusB(int *ret, int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
    int *ret;
    cudaMalloc(&ret, 1000 * sizeof(int));
    AplusB<<< 1, 1000 >>>(ret, 10, 100);
    int *host_ret = (int *)malloc(1000 * sizeof(int));
    cudaMemcpy(host_ret, ret, 1000 * sizeof(int), cudaMemcpyDefault);
    for(int i = 0; i < 1000; i++)
        printf("%d: A+B = %d\n", i, host_ret[i]);
    free(host_ret);
    cudaFree(ret);
    return 0;
}
```

```cpp
__global__ void AplusB(int *ret, int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
    int *ret;
    cudaMalloc(&ret, 1000 * sizeof(int));
    AplusB<<< 1, 1000 >>>(ret, 10, 100);
    int *host_ret = (int *)malloc(1000 * sizeof(int));
    cudaMemcpy(host_ret, ret, 1000 * sizeof(int), cudaMemcpyDefault);
    for(int i = 0; i < 1000; i++)
        printf("%d: A+B = %d\n", i, host_ret[i]);
    free(host_ret);
    cudaFree(ret);
    return 0;
}
```

```cpp
__global__ void AplusB(int *ret, int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
    int *ret;
    cudaMallocManaged(&ret, 1000 * sizeof(int));
    AplusB<<< 1, 1000 >>>(ret, 10, 100);
    cudaDeviceSynchronize();
    for(int i = 0; i < 1000; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
    cudaFree(ret);
    return 0;
}
```

```cpp
__device__ __managed__ int ret[1000];
__global__ void AplusB(int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
    AplusB<<< 1, 1000 >>>(10, 100);
    cudaDeviceSynchronize();
    for(int i = 0; i < 1000; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
    return 0;
}
```



```cpp
__global__ void AplusB(int *ret, int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
    int *ret;
    cudaMallocManaged(&ret, 1000 * sizeof(int));
    AplusB<<< 1, 1000 >>>(ret, 10, 100);
    cudaDeviceSynchronize();
    for(int i = 0; i < 1000; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
    cudaFree(ret);
    return 0;
}
```

```cpp
__device__ __managed__ int ret[1000];
__global__ void AplusB(int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
    AplusB<<< 1, 1000 >>>(10, 100);
    cudaDeviceSynchronize();
    for(int i = 0; i < 1000; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
    return 0;
}
```

```
__global__ void AplusB(int *ret, int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
    int *ret;
    cudaMallocManaged(&ret, 1000 * sizeof(int));
    AplusB<<< 1, 1000 >>>(ret, 10, 100);
    cudaDeviceSynchronize();
    for(int i = 0; i < 1000; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
    cudaFree(ret);
    return 0;
}
```

```
__device__ __managed__ int ret[1000];
__global__ void AplusB(int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
    AplusB<<< 1, 1000 >>>(10, 100);
    cudaDeviceSynchronize();
    for(int i = 0; i < 1000; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
    return 0;
}
```

```cpp
__global__ void AplusB(int *ret, int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
    int *ret;
    cudaMallocManaged(&ret, 1000 * sizeof(int));
    AplusB<<< 1, 1000 >>>(ret, 10, 100);
    cudaDeviceSynchronize();
    for(int i = 0; i < 1000; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
    cudaFree(ret);
    return 0;
}
```

```cpp
__device__ __managed__ int ret[1000];
__global__ void AplusB(int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
    AplusB<<< 1, 1000 >>>(10, 100);
    cudaDeviceSynchronize();
    for(int i = 0; i < 1000; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
    return 0;
}
```

Unified Memory

- Zaujímavosťou je podpora virtuálnej pamäte na grafických kartách
- Od CC 6.0 je zavedená podpora 49-bitovej adresy na zariadení… 48 pre virtuálnu adresu OS a bit navyše, aby bolo možné adresovať aj pamať zariadenia
- Táto funkcionalita je zavedená kvôli tomu, aby mohli výpočty využívať viac pamäte, než je fyzická veľkosť pamäte

CUDA streams

CUDA streams

- Prúdy (streams) sú nástrojom, pomocou ktorého sa dá ľahko definovať, ktoré kernely (hlavné funkcie spúšťané na GPU) sú závislé a ktoré nie

CUDA streams

- Kernely vložené do spoločného prúdu sa vykonávajú SEKVENČNE jeden za druhým
- Kernely z viacerých prúdov sa vykonávajú KONKURENTNE

Stratégie optimalizácie výkonu

Druhá časť

Stratégie optimalizácie výkonu

Stratégie optimalizácie výkonu

1. Maximalizácia paralelného vykonávania za účelom maximálneho vyťaženia zariadenia

Stratégie optimalizácie výkonu

1. Maximalizácia paralelného vykonávania za účelom maximálneho vyťaženia zariadenia

2. Optimalizácia využívania pamäte za účelom maximalizácie pamäťovej priepustnosti

Stratégie optimalizácie výkonu

1. Maximalizácia paralelného vykonávania za účelom maximálneho vyťaženia zariadenia

2. Optimalizácia využívania pamäte za účelom maximalizácie pamäťovej priepustnosti

3. Optimalizácia využitia inštrukcií za účelom maximalizácie priepustnosti inštrukcií

Stratégie optimalizácie výkonu

- Ktorú stratégiu použiť na ktorú časť programu sa vo všeobecnosti nedá povedať

- Dá sa povedať, čo kedy nepoužiť
- Napríklad je zbytočné optimalizovať kód, ak je úzkym hrdlom aplikácie prístup k údajom v pamäti

Stratégie optimalizácie výkonu

- Optimalizačné úsilie treba zamerať tým smerom, kde je úzke hrdlo systému
- Ako?

Stratégie optimalizácie výkonu

- Optimalizačné úsilie treba zamerať tým smerom, kde je úzke hrdlo systému
- Ako?

- Meraním a monitorovaním – napríklad použitím profilerov
- Porovnaním teoretických možností zariadenia s výsledkami merania profilovacích nástrojov

Optimalizácia 1 – využitie zariadenia

Optimalizácia 1 – využitie zariadenia

- Aplikácia by mala byť tak štruktúrovaná, aby v čo najväčšej miere využívala paralelizmus a ten vhodne mapovala na patričné komponenty systému s cieľom v čo najväčšej miere využiť jeho kapacitu výpočtu

Optimalizácia 1 – využitie zariadenia

- Aplikácia by mala byť tak štruktúrovaná, aby v čo najväčšej miere využívala paralelizmus a ten vhodne mapovala na patričné komponenty systému s cieľom v čo najväčšej miere využiť jeho kapacitu výpočtu
- Tri pohľady na využitie zariadenia
  1. Aplikačná úroveň
  2. Úroveň zariadenia
  3. Úroveň multiprocesora SM

Optimalizácia 1 – využitie zariadenia

- Úroveň aplikácie
  - Súčasné využitie nezávislých častí systému (CPU, GPU, zbernice) pomocou asynchrónneho vykonávania kódu
  - Každý typ procesora by mal robiť tú činnosť, pri ktorej má najvyšší výkon (CPU sériové časti programu, GPU paralelné časti výpočtu)

Optimalizácia 1 – využitie zariadenia

- Úroveň aplikácie
  - Čo ak pri paralelnom behu na GPU je nutné synchronizovať výpočtové vlákna (napr. kvôli výmene údajov)?

Optimalizácia 1 – využitie zariadenia

- Úroveň aplikácie
  - Čo ak pri paralelnom behu na GPU je nutné synchronizovať výpočtové vlákna (napr. kvôli výmene údajov)?
  - Dve situácie môžu nastať: buď vlákna patria do toho istého bloku alebo nie

Optimalizácia 1 – využitie zariadenia

- Úroveň aplikácie
  1. Ak vlákna patria do toho istého bloku, môžu využiť funkciu __syncthreads()
  2. Vlákna nepatriace do toho istého bloku musia zdieľať údaje pomocou globálnej pamäte zariadenia minimálne v dvoch (sekvenčných) volaniach kernel funkcií (jedna údaje zapíše do globálnej pamäte, druhý ich prečíta)
     - Tento druhý prípad je veľmi neoptimálny a mal by sa minimalizovať

Optimalizácia 1 – využitie zariadenia

- Úroveň zariadenia
  - Aplikácia by mala zapojiť do výpočtu čo najviac SM
  - Maximalizácia paralelného behu kernelov sa dá dosiahnuť pomocou ich konkurentného vykonávania pomocou prúdov (streams)

	Compute Capability												
Technical Specifications	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**	**7.0**	**7.2**	**7.5**	**8.0**	**8.6**
Maximum number of resident grids per device (Concurrent Kernel Execution)	32				16	128	32	16	128	16	128		

Optimalizácia 1 – využitie zariadenia

- Úroveň GPU multiprocesora
  - MP využíva paralelný beh vlákien vo warpoch
  - Teda využitie zariadenia je priamo dané počtom warpov zapojených do výpočtu
  - Počet hodinových cyklov nutných na pripravenie vykonania inštrukcie pre warp sa nazýva *latencia*
  - Plné využitie warpu sa dosiahne, ak všetky warp plánovače majú v každom hodinovom cykle vždy k dispozícii nejakú inštrukciu na vykonanie

Optimalizácia 1 – využitie zariadenia

- Úroveň GPU multiprocesora
  - Najčastejší prípad, kedy warp nie je pripravený vykonať inštrukciu, nastáva, keď ešte nie je pripravený operand inštrukcie

Optimalizácia 1 – využitie zariadenia

- Úroveň GPU multiprocesora
  - Najčastejší prípad, kedy warp nie je pripravený vykonať inštrukciu, nastáva, keď ešte nie je pripravený operand inštrukcie
    - Ak ide o operand v registri, latencia je v jednotkách cyklov
    - Ak ide o operand v pamäti, môže ísť o **stovky** cyklov!

Optimalizácia 1 – využitie zariadenia

- Úroveň GPU multiprocesora
  - Ďalším dôvodom latencie môže byť synchronizačné obmedzenie (__syncthreads())
  - Keďže čakať na seba môžu iba vlákna toho istého bloku, tak v tomto prípade je vhodné zvýšiť počet blokov programu

	Compute Capability												
Technical Specifications	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**	**7.0**	**7.2**	**7.5**	**8.0**	**8.6**
Maximum number of resident blocks per SM	16		32								16	32	16
Maximum number of resident warps per SM	64										32	64	48
Maximum number of resident threads per SM	2048										1024	2048	1536

Optimalizácia 1 – využitie zariadenia

- Úroveň GPU multiprocesora
  - Na využitie MP má obrovský vplyv aj počet použitých registrov!
  - Príklad: nech kernel (funkcia) využíva 64 registrov; nech je kernel spustený v blokoch, z ktorých každý má 512 vlákien (a takmer nevyužívajú zdieľanú pamäť)
  - 64*512 = 64 * 0.5k = 64/2k = 32k registrov 1 blok

	Compute Capability											
Technical Specifications	**3.0**	**3.2**	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**	**7.0**	**7.5**
Number of 32-bit registers per multiprocessor	64 K			128 K	64 K							
Maximum number of 32-bit registers per thread block	64 K	32 K	64 K				32 K	64 K		32 K	64 K	
Maximum number of 32-bit registers per thread	63	255										

- Priklad: nech kernel (funkcia) vyuziva 64 registrov; nech je kernel spusteny v blokoch, z ktorych kazdy ma 512 vlakien (a takmer nevyuzivaju zdielanu pamat)
- 64*512 = 64 * 0.5k = 64/2k = 32k registrov 1 blok

	Compute Capability											
Technical Specifications	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.5
Number of 32-bit registers per multiprocessor	64 K			128 K	64 K							
Maximum number of 32-bit registers per thread block	64 K	32 K	64 K				32 K	64 K		32 K	64 K	
Maximum number of 32-bit registers per thread	63	255										

- Priklad: nech kernel (funkcia) vyuziva 64 registrov; nech je kernel spusteny v blokoch, z ktorych kazdy ma 512 vlakien (a takmer nevyuzivaju zdielanu pamat)
- 64*512 = 64 * 0.5k = 64/2k = 32k registrov 1 blok
- Majme napr. zariadenie s podporou CC 6.0

	Compute Capability											
Technical Specifications	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.5
Number of 32-bit registers per multiprocessor	64 K			128 K	64 K							
Maximum number of 32-bit registers per thread block	64 K	32 K	64 K				32 K	64 K		32 K	64 K	
Maximum number of 32-bit registers per thread	63	255										

- Priklad: nech kernel (funkcia) vyuziva 64 registrov; nech je kernel spusteny v blokoch, z ktorych kazdy ma 512 vlakien (a takmer nevyuzivaju zdielanu pamat)
- 64*512 = 64 * 0.5k = 64/2k = 32k registrov 1 blok
- Majme napr. zariadenie s podporou CC 6.0
- !!! 64k / 32k = 2 bloky môžu byť na 1 MP !!!

	Compute Capability											
Technical Specifications	**3.0**	**3.2**	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**	**7.0**	**7.5**
Number of 32-bit registers per multiprocessor	64 K			128 K	64 K							
Maximum number of 32-bit registers per thread block	64 K	32 K	64 K				32 K	64 K		32 K	64 K	
Maximum number of 32-bit registers per thread	63	255										

– Keď by sme pridali čo i len 1 register navyše, na MP bude môcť bežať už len jeden blok!

– Takže sa môže stať, že takmer polovica warpov bude na MP nevyužitá!

Optimalizácia 1 – využitie zariadenia

- Úroveň GPU multiprocesora
  - Pri využití registrov si treba dať pozor na to, že VŽDY sa “ukrajuje” z miesta vyhradeného pre registre MP násobok 32 bitov
  - 8-bitová premenná zaberie 32 bitov
  - 48-bitová premenná zaberie 64 bitov
  - atď

Optimalizácia 1 – využitie zariadenia

- Úroveň GPU multiprocesora
  - Experimenty, experimenty a ešte raz experimenty!
  - Zároveň s tým štúdium parametrov zariadenia
  - Počet vlákien v bloku VŽDY volíme ako násobok veľkosti warpu, aby sme nemrhali výpočtovými zdrojmi (našťastie, veľkosť warpu je jedna z mála hodnôt, ktoré sa zatiaľ nikdy nemenili…)

Optimalizácia 1 – využitie zariadenia

- Úroveň GPU multiprocesora

	Compute Capability												
Technical Specifications	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**	**7.0**	**7.2**	**7.5**	**8.0**	**8.6**
Warp size	32												

  - Počet vlákien v bloku VŽDY volíme ako násobok veľkosti warpu, aby sme nemrhali výpočtovými zdrojmi (našťastie, veľkosť warpu je jedna z mála hodnôt, ktoré sa zatiaľ nikdy nemenili…)

Optimalizácia 1 – využitie zariadenia

- Situácia nie je úplne zúfalá… jestvuje pár API funkcií, ktoré pomáhajú vypočítať optimálny počet blokov na základe využitia registrov a zdieľanej pamäte

Optimalizácia 1 – využitie zariadenia

- Situácia nie je úplne zúfalá… jestvuje pár API funkcií, ktoré pomáhajú vypočítať optimálny počet blokov na základe využitia registrov a zdieľanej pamäte

- cudaOccupancyMaxActiveBlocksPerMultiprocessor vracia počet konkurentne vykonávateľných blokov na 1 MP

Optimalizácia 1 – využitie zariadenia

- Vrátena hodnota tejto funkcie (#blocks/MP) sa dá prepočítať aj na iné metriky
  - Vynásobením WarpsPerBlock dostaneme počet konkurentne vykonávateľných warpov na 1 MP
  - Vydelením takto získanej hodnoty maximálnym počtom warpov na 1 MP dostaneme využitie

	Compute Capability											
Technical Specifications	**3.0**	**3.2**	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**	**7.0**	**7.5**
Maximum number of resident blocks per multiprocessor	16				32							16
Maximum number of resident warps per multiprocessor	64											32
Maximum number of resident threads per multiprocessor	2048											1024

```
// Device code
__global__ void MyKernel(int *d, int *a, int *b)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    d[idx] = a[idx] * b[idx];
}

// Host code
int main()
{
    int numBlocks;        // Occupancy in terms of active blocks
    int blockSize = 32;

    // These variables are used to convert occupancy to warps
    int device;
    cudaDeviceProp prop;
    int activeWarps;
    int maxWarps;

    cudaGetDevice(&device);
    cudaGetDeviceProperties(&prop, device);

    cudaOccupancyMaxActiveBlocksPerMultiprocessor(
        &numBlocks,
        MyKernel,
        blockSize,
        0);

    activeWarps = numBlocks * blockSize / prop.warpSize;
    maxWarps = prop.maxThreadsPerMultiProcessor / prop.warpSize;

    std::cout << "Occupancy: " << (double)activeWarps / maxWarps * 100 << "%" <<
 std::endl;

    return 0;
}
```

Optimalizácia 1 – využitie zariadenia

- Jestvujú ďalšie API funkcie na heuristickú analýzu
- Viď dokumentácia…

Optimalizácia 2 – priepustnosť pamäte

Optimalizácia 2 – priepustnosť pamäte

- Zásadným krokom je minimalizácia prenosu údajov

- Ak to nie je možné, nerobiť prenosy s malou šírkou pásma (t.j. CPU vs GPU)

Optimalizácia 2 – priepustnosť pamäte

- Najhoršie sú na tom prenosy medzi CPU a GPU
- Potom prenosy medzi globálnou pamäťou GPU a zdieľanou pamäťou
- Nasleduje prenos medzi zdieľanou pamäťou a súborom registrov GPU

Optimalizácia 2 – priepustnosť pamäte

- Aké kroky musí urobiť vlákno, keď chce preniesť údaje z/do globálnej pamäte GPU?

Optimalizácia 2 – priepustnosť pamäte

- Aké kroky musí urobiť vlákno, keď chce preniesť údaje z/do globálnej pamäte GPU
  - Údaje z pamäte GPU do zdieľanej pamäte
  - Synchronizovať sa s ostatnými vláknami bloku (aby sa zaistila validita zdieľaných údajov pre všetky vlákna bloku)
  - Spracovať údaje
  - Znovu sa synchronizovať (ak je to potrebné)
  - Zapísať údaje do pamäte GPU

Optimalizácia 2 – priepustnosť pamäte

- Zvlášť si treba dávať pozor na prístupy vlákien do globálnej pamäte zariadenia (o tom ešte bude reč)

- Treba sa im vyhýbať

Optimalizácia 2 – priepustnosť pamäte

- Typy prístupov, ktoré teda treba riešiť

Optimalizácia 2 – priepustnosť pamäte

- Typy prístupov, ktoré teda treba riešiť

- Prenos údajov medzi hostom a zariadením
- Prenos údajov medzi (globálnou) pamäťou zariadenia a zdieľanou/lokálnou pamäťou
  - Pamäť GPU sa delí na 5 typov
  - globálna pamäť, pamäť konštánt, pamäť textúr, zdieľaná pamäť a lokálna pamäť

Optimalizácia 2 – priepustnosť pamäte

- Prenos host – zariadenie

Optimalizácia 2 – priepustnosť pamäte

- Prenos host – zariadenie
  - Vždy treba minimalizovať tento typ prenosu
  - Napríklad tým, že sa väčšia časť kódu prenesie z hosta na zariadenie (aj za cenu toho, že sa naplno nevyužije paralelizmus zariadenia – keď budú vlákna medzi sebou blokované)

Optimalizácia 2 – priepustnosť pamäte

- Prenos host – zariadenie
  - Vždy treba minimalizovať tento typ prenosu
  - Napríklad tým, že sa väčšia časť kódu prenesie z hosta na zariadenie (aj za cenu toho, že sa naplno nevyužije paralelizmus zariadenia – keď budú vlákna medzi sebou blokované)
  - Ak už treba prenos robiť, tak je vhodné agregovať viac menších prenosov do jedného väčšieho

Optimalizácia 2 – priepustnosť pamäte

- Pamäťové prístupy v rámci zariadenia

Optimalizácia 2 – priepustnosť pamäte

- Pamäťové prístupy v rámci zariadenia

- Globálna pamäť
- Lokálna pamäť
- Zdieľaná pamäť
- Pamäť konštánt
- Pamäť textúr

Optimalizácia 2 – priepustnosť pamäte

- **Globálna pamäť**

- Prístup do tejto pamäte sa robí v tzv. transakciách o veľkosti 32, 64 alebo 128 bajtov
- Prvá adresa, na ktorú sa v rámci transakcie pristupuje, musí byť zarovnaná na násobok veľkosti transakcie!!!

Optimalizácia 2 – priepustnosť pamäte

- **Globálna pamäť**

- Keď warp vykonáva inštrukciu, ktorá pristupuje ku globálnej pamäti, pamäťové prístupy vlákien v rámci warpu spojí do jednej alebo viacerých transakcií v závislosti od
  1. Veľkosti údajov, ku ktorým vlákno pristupuje
  2. Samotných adries, ku ktorým sa pristupuje

Optimalizácia 2 – priepustnosť pamäte

- **Globálna pamäť**

- Čím viac transakcií pre vlákna warpu treba urobiť, tým viac sa zmenšuje priepustnosť
  - Majme napríklad 32-bajtové transakcie, ale tak nešťastne, že každé vlákno z takejto transakcie využije iba 4 bajty
  - 32/4 = 8, takže 8x sa nám zmenší priepustnosť pamäte

Optimalizácia 2 – priepustnosť pamäte

- **Globálna pamäť**

- Koľko transakcié na vykonanie inštrukcie warpu je potrebných a ako to ovplyvní priepustnosť, závisí od CUDA verzie, ktorú zariadenie podporuje

Optimalizácia 2 – priepustnosť pamäte

- **Globálna pamäť**

- Vo všeobecnosti platí, že je potrebné čo najviac zlučovať pamäťové prístupy do čo najmenšieho počtu transakcií
- Ako?

Optimalizácia 2 – priepustnosť pamäte

- **Globálna pamäť**

1. Naštudovať si optimálny vzor prístupu do globálnej pamäte pre tú-ktorú verziu CC
2. Použiť také údajové typy, veľkosti a zarovnania, ktoré vyhovujú požiadavkám (uvedené ďalej v prednáške)
3. Ak je to vhodné, urobiť explicitné doplnenie (*padding*) údajov, viď ďalej v prednáške

Optimalizácia 2 – priepustnosť pamäte

- **Globálna pamäť** – požiadavky veľkosti a zarovnania

- Inštrukcie prístupu do globálnej pamäte majú veľkosti operandov 1, 2, 4, 8 alebo 16 bajtov
- Prístup do pamäte sa realizuje JEDNOU inštrukciou iba vtedy, ak je táto veľkosť zachovaná a údaje sú na adrese, ktorá je násobkom tejto veľkosti operandu!

Optimalizácia 2 – priepustnosť pamäte

- **Globálna pamäť** – požiadavky veľkosti a zarovnania

- Ak nie je požiadavka veľkosti alebo zarovnania zachovaná, prístup sa rozdelí do viacerých inštrukcií, ktoré sa nemusia dať spojiť do jednej transakcie

Optimalizácia 2 – priepustnosť pamäte

- **Globálna pamäť** – dvojrozmerné polia

- Všeobecný vzor prístupu do pamäte v prípade dvojrozmerných polí je nasledovný

Optimalizácia 2 – priepustnosť pamäte

- **Globálna pamäť** – dvojrozmerné polia

- Všeobecný vzor prístupu do pamäte v prípade dvojrozmerných polí je nasledovný
  - Vlákno má spracovať prvok na (tx, ty) pozícii
  - Riadok dvojrozmerného poľa má šírku width
  - Pole začína na adrese BaseAddress
  - Adresa typu spĺňa požiadavky zarovnania

Optimalizácia 2 – priepustnosť pamäte

- **Globálna pamäť** – dvojrozmerné polia

- Adresa prvku je potom

  BaseAddress + width * ty + tx

- Aby takýto prístup mohol byť plne zlúčený (*coaleshed*), šírka poľa aj šírka bloku vlákien musia byť násobkom veľkosti warpu!

Optimalizácia 2 – priepustnosť pamäte

- **Globálna pamäť** – dvojrozmerné polia

- Ak nejaké pole nemá šírku riadku rovnú násobku veľkosti warpu, je potrebné takéto pole alokovať tak, že sa šírka riadku zarovná na násobok veľkosti warpu a zvyšky riadkov sa pri výpočte budú ignorovať (*padding*)

Optimalizácia 2 – priepustnosť pamäte

- **Globálna pamäť** – dvojrozmerné polia

- Takáto alokácia sa dá robiť HW nezávislo
- Viď funkcie CUDA API
  - cudaMallocPitch()
  - cuMemAllocPitch()
  - …

Optimalizácia 2 – priepustnosť pamäte

- **Lokálna pamäť**

- Využíva sa pre automatické premenné, ktoré generuje kompilátor
  - Štruktúry, ktoré zaberú príliš veľa registrov
  - Ak už nie je miesto v súbore registrov (tzv. *register spilling*)
  - Iné

Optimalizácia 2 – priepustnosť pamäte

- **Lokálna pamäť**

- Lokálna pamäť sa nachádza v pamäti zariadenia, takže pre prístupy do nej platia tie isté pravidlá a obmedzenia ako pre globálnu pamäť

Optimalizácia 2 – priepustnosť pamäte

- **Lokálna pamäť**

- Prístupy do lokálnej pamäte sa uchovávajú vo vyrovnávacej pamäti L2 podobne ako prístupy do globálnej pamäte
- To platí pre niektoré zariadenia CC 3.x a všetky zariadenia CC 5.x a 6.x

Optimalizácia 2 – priepustnosť pamäte

- **Lokálna pamäť**

- Lokálna pamäť má svoj názov nie podľa fyzického umiestnenia, ale podľa rozsahu platnosti
- Názov je veľmi zavádzajúci…

Optimalizácia 2 – priepustnosť pamäte

- **Zdieľaná pamäť**

- Nachádza sa *on-chip*, takže prenos má oveľa menšiu latenciu než prístup do globálnej či lokálnej pamäte

Optimalizácia 2 – priepustnosť pamäte

- **Zdieľaná pamäť**

- Aby sa dosiahla vysoká priepustnosť na úrovni samotného hw, zdieľaná pamäť je na úrovni hw rozdelená do rovnako veľkých pamäťových modulov nazývaných banky (*banks*)
- Ku bankám je možné realizovať SÚČASNÝ prístup

Optimalizácia 2 – priepustnosť pamäte

- **Zdieľaná pamäť**

- Ak N žiadostí o prístup k pamäti padne do N rôznych pamäťových bánk, žiadosti je možné obslúžiť súbežne
- Ak dve RôZNE adresy žiadosti o prístup padnú do tej istej banky, ide o konflikt a žiadosti sú vybavené sekvenčne

Optimalizácia 2 – priepustnosť pamäte

- **Zdieľaná pamäť**

- HW automaticky rozdeľuje konflikty do potrebného počtu nekonfliktných prístupov
- Tým sa znižuje priepustnosť úmerne počtu nezávislých prístupov

Optimalizácia 2 – priepustnosť pamäte

- **Zdieľaná pamäť**

- Ak počet nezávislých prístupov, ktoré obslúžia počiatočnú požiadavku, je N, hovoríme, že počiatočná požiadavka vyvolala N-násobný konflikt pamäťových bánk

Optimalizácia 2 – priepustnosť pamäte

- **Zdieľaná pamäť**

- Ak chceme využiť plný výkon prístupov, je potrebné nahliadnuť do dokumentácie pre túktorú CUDA verziu

- Ako hw mapuje adresy pamäťových prístupov do jednotlivých bánk

Optimalizácia 2 – priepustnosť pamäte

- **Pamäť konštánt**

- Nachádza sa v pamäti zariadenia a má zvlášť vyrovnávaciu pamäť konštánt

- Požiadavka o prístup do pamäte je rozdelená na toľko prístupov, koľko rôznych adries sa v počiatočnej požiadavke nachádza

Optimalizácia 2 – priepustnosť pamäte

- **Pamäť textúr**

- Rozdiel oproti pamäti konštánt je v tom, že pamäť textúr je optimalizovaná na 2D dimenziu

- Žiadosti o prístup vlákien toho istého warpu sa optimálne zlučujú

Optimalizácia 2 – priepustnosť pamäte

- **Pamäť textúr**

- Sídli v pamäti zariadenia a podobne ako pamäť konštánt, má vlastnú vyrovnávaciu pamäť

- Cena prístupu je podobne ako pri konštantách daná tým, či sa údaj nachádza vo vyrovnávacej pamäti alebo nie

Optimalizácia 3 – priepustnosť inštrukcií

Optimalizácia 3 – priepustnosť inštrukcií

- Maximalizáciu priepustnosti inštrukcií aplikácia dosiahne, ak
  1. Minimalizuje použitie aritmetických inštrukcií, ktoré dlho trvajú (použitie *intrinsic* funkcií miesto klasických, použitie *float* miesto *double*, …)
  2. Minimalizuje divergenciu vykonávaného kódu v rámci warpu (*if-else*)
  3. Minimalizuje počet použitých inštrukcií

Optimalizácia 3 – priepustnosť inštrukcií

- Priepustnosť inštrukcií sa meria v počte operácií za jednotku času (cyklu) na jeden multiprocesor

- Keďže vždy sa plánuje na beh aspoň 1 warp vlákien, tak pre veľkost warpu 32 jedna inštrukcia zodpovedá vykonaniu 32 operacií

Optimalizácia 3 – priepustnosť inštrukcií

- Ak máme výsledný počet operácií za jednotku času N, potom je priepustnosť daná hodnotou N/32 inštrukcií za jednotku času

- Priepustnosť je počítaná na jeden MP
- Ak ju chceme vyjadriť pre celé zariadenie, získanú hodnotu je potrebné vynásobiť počtom MP zariadenia

Optimalizácia 3 – priepustnosť inštrukcií

- Pri výpočtoch treba zohľadňovať 3 skupiny inštrukcií

1. Aritmetické inštrukcie
2. Inštrukcie ovplyvňujúce tok riadenia
3. Synchronizačné inštrukcie

Prístupy do pamäti zariadenia podľa CUDA verzie

Tretia cast

- CUDA CC 3.x (globálna a zdieľaná pamäť)

- CUDA CC 5.x (globálna a zdieľaná pamäť)
- CUDA CC 6.x (globálna a zdieľaná pamäť)
- CUDA CC 7.x (globálna a zdieľaná pamäť)
- CUDA CC 8.x (globálna a zdieľaná pamäť)

Globálna pamäť

Globálna pamäť

- V prípade podpory vyrovnávacej pamäte platí, že:

- riadok vyrovnávacej pamäte je veľký 128B
- mapuje sa na 128B pamäte zariadenia
- adresa, na ktorú sa mapuje, je zarovnaná na hodnotu 128

Globálna pamäť

- Pre žiadosť o prístup pre vlákna warpu platí

- Ak je veľkosť operandu (každého) vlákna 1, 2 alebo 4B, obslúži sa jedinou transakciou
- Ak je veľkosť operandu 8B, obslúži sa dvoma prístupmi (pre každú polovicu warpu zvlášť)
- Ak je veľkosť operandu 16B, obslúži sa štyrmi transakciami

Globálna pamäť

- Prístupy do globálnej pamäte vo verziách CUDA CC 5.x, 6.x, 7.x a 8.x idú vždy cez vyrovnávaciu pamäť

- Prístupy vo verzii CC 3.x sa líšia (viď špecifikácie jednotlivých subverzií)

Zdieľaná pamäť

Zdieľaná pamäť

- Zdieľaná pamäť má 32 bánk
- Iba CUDA CC 3.x: s možnosťou dvoch adresných režimov (tie sa dajú nastavovať a zisťovať pomocou API funkcií)

Zdieľaná pamäť

- Zdieľaná pamäť má 32 bánk
- Iba CUDA CC 3.x: s možnosťou dvoch adresných režimov (tie sa dajú nastavovať a zisťovať pomocou API funkcií)

1. 64-bitový režim (za sebou idúce 64-bitové slová sa mapujú do za sebou idúcich bánk)
2. 32-bitový režim (za sebou idúce 32-bitové slová sa mapujú do za sebou idúcich bánk)

Zdieľaná pamäť

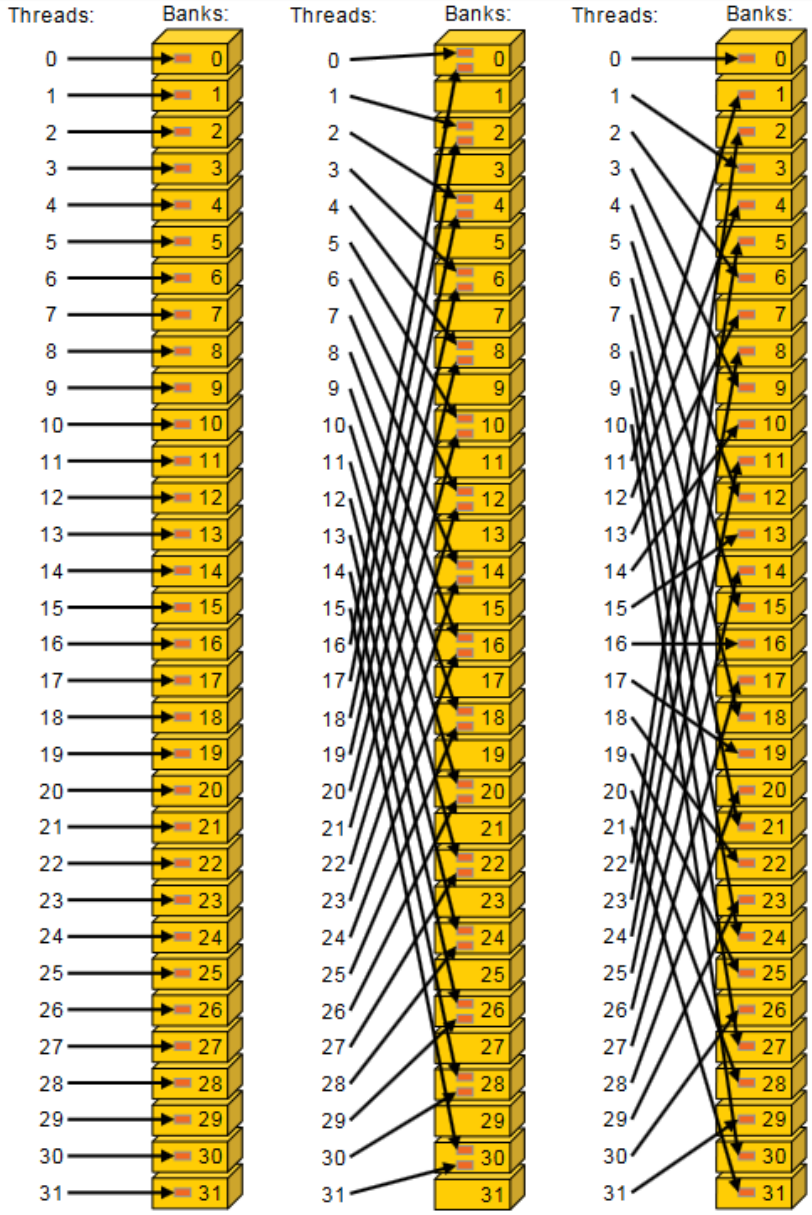
- V pripade CUDA CC 7.x a 8.x je možnosť pomocou API volaní nastaviť veľkosť zdieľanej pamäte (zvyšok sa využije pre L1)

- Konflikt sa nedeje, ak dve vlákna warpu nepristupujú k adrese (v rámci toho istého slova) v rámci jednej banky

- Čo v prípade konfliktu?

Zdieľaná pamäť

- Ak je prístup vlákien na čítanie, hodnota slova z banky sa dodá všetkým vláknam
- Ak je prístup na zápis, na každú adresu zapíše práve jedno z vlákien (ktoré súperia o zápis na adresu), pričom nie je definované, ktoré



Left

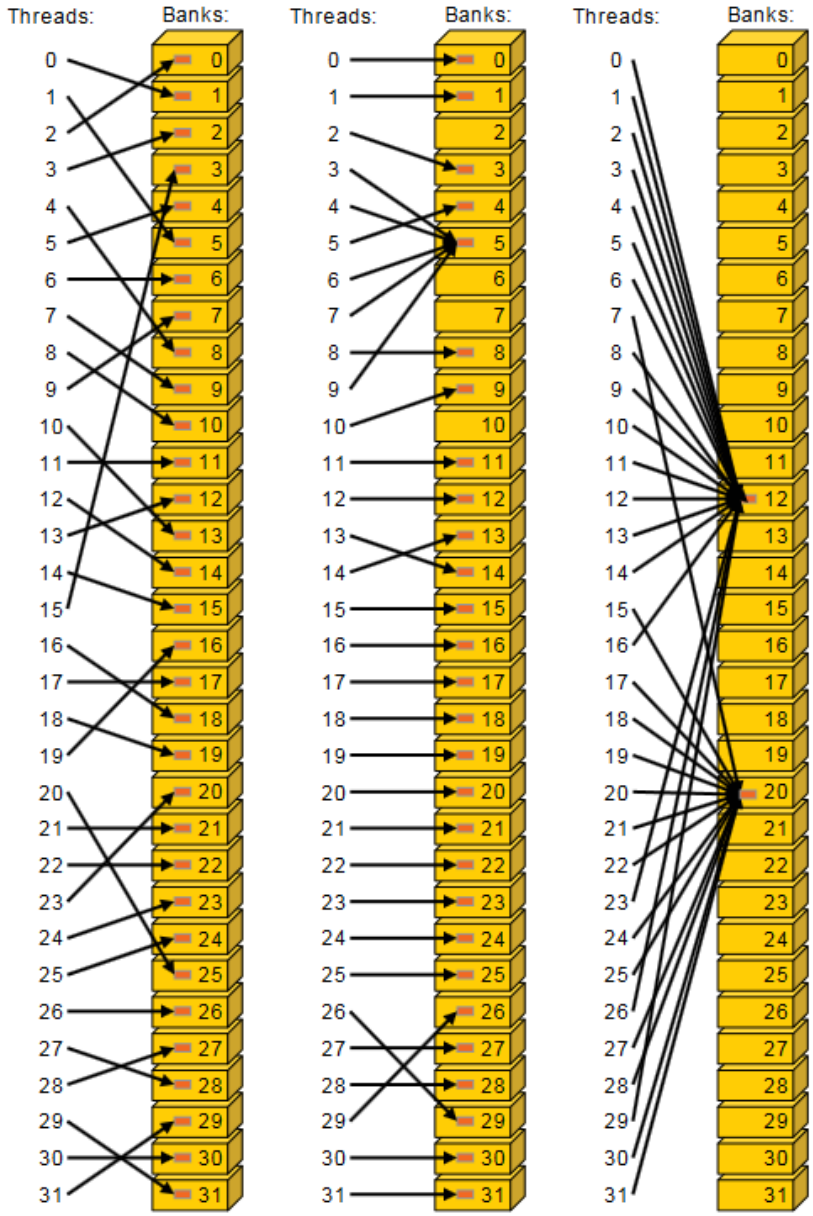
Linear addressing with a stride of one 32-bit word (no bank conflict).

Middle

Linear addressing with a stride of two 32-bit words (two-way bank conflict).

Right

Linear addressing with a stride of three 32-bit words (no bank conflict).



Left

Conflict-free access via random permutation.

Middle

Conflict-free access since threads 3, 4, 6, 7, and 9 access the same word within bank 5.

Right

Conflict-free broadcast access (threads access the same word within a bank).

Ďalšie zdroje štúdia

Ďalšie zdroje

- CUDA pomocou Numba: https://numba.pydata.org/numba-doc/latest/cuda/index.html

- CUDA Best Practices: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

- Device Memory Spaces: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#device-memory-spaces

- CUDA complet: https://docs.nvidia.com/cuda/

Ďakujem za pozornosť