PPaDS MMXX

Matus Jokay, C-503, matus.jokay@stuba.sk
Roderik Ploszek, C-512, roderik.ploszek@stuba.sk

uim.fei.stuba.sk/predmet/i-ppds
konzultacie dohodou

Zdroj prednasky

https://docs.nvidia.com

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

Obsah prednasky

- Vyvoj GPU
- Optimalizacia zariadenia
- Pristupy do pamati zariadenia podla CUDA verzie

Vyvoj GPU

Prva cast

Prehlad hw generacii NVidia

Generacia HW	Vypoctove moznosti (CC)		SP / SM = 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

Vyvoj GPU

- SP, viacere SP tvoria SM
- Viacere SM tvoria GPC (GPU Processing Cluster)
- SM obsahuje jednotky nielen pre spracovanie textur a INT32, ale aj FP32a FP64
- SM obsahuje TC (Tensor Core) (operacia A*B + C na maticiach 4x4)
- Zvacsuje sa L1, L2 a priepustnost zbernic

Vyvoj GPU

- Planovac vlakien!
  - Volta zavadza zavadza PC a Stack pre kazde vlakno warpu! (vyuzitie: divergencia vykonavania kodu na urovni warpu)
  - Predtym PC a Stack iba pre cely warp spolu
- Zvysuje sa konkurentnost / paralelizmus
- Vylepsovanie algoritmov planovania vlakien (problem vyhladovenia, divergencie)

Vyvoj GPU

- Na pociatku bolo mozne vykonavat na GPU jediny kernel
- Postupne viacero, toho isteho procesu
- Potom aj viacero, roznych procesov
  - A to je bezpecnostne riziko!
  - Zavadza sa oddelenie adresneho priestoru GPU pre jednotlive procesy

Vyvoj GPU

- Z pohladu aplikacneho programatora sa rozhranie neustale zjednodusuje

- Priklad: alokacia/uvolnovanie pamate pomocou “klasickych” funkcii OS: malloc() a free()
- Zavedenie jednotnej pamatovej technologie (Unified Memory Technology)

Vyvoj GPU

- Pre paralelne vypocty je casto dolezita spolupraca vlakien pri dosahovani vysledku
- Zavadza sa koncept Kooperativnych skupin vlakien (Cooperative Groups v CUDA 9.x)
- Kym neprisiel tento koncept, jestvovala jedina moznost synchronizacie vlakien: __syncthreads()
- Niekedy je potrebne synchronizovat menej vlakien nez je cely 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();
    }
}
```

Vyvoj GPU

- Spajanie viacerych GPU

- V ramci jedneho “zariadenia” (graficka karta)
- V ramci pocitaca (SLI)

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

- Sucast programovacieho modelu CUDA
- Od verzie CC 6.0

- Definuje menezovany (managed) pamatovy priestor, v ramci ktoreho vsetky procesory “vidia” jednotnu suvislu pamatovu oblast bez rozlisovania typu zariadenia pri pristupe do pamate

Unified Memory

- Nie je potrebne pouzivat cudaMemcpy*() funkcie

- Cas vykonavania kodu sa ovsem neznizi, pretoze stale sa musia udaje presuvat medzi hlavnou pamatou a pamatou grafickej karty
- Zlepsuje sa citatelnost kodu, znizuje sa jeho zlozitost

Unified Memory

- V programe je mozne vyuzivat ukazatel bez ohladu na to, kde sa udaje nachadzaju

- Nie je potrebne volat funkcie na explicitnu migraciu udajov medzi roznymi pamatami
- Akakolvek alokacia vytvorena pomocou menezovanej pamate automaticky migruje udaje tam, kam patria

Unified Memory

- Alokacia menezovanej pamate sa robi dvoma sposobmi
  - Volanim cudaMallocManaged(), ktora ma rovnake rozhranie ako cudaMalloc()
  - Definovanim globalnej __managed__ premennej, ktora ma podobny vyznam ako __device__ premenna

Unified Memory

- Model jednotnej pamate je v CUDA pritomny od verzie CC 3.0

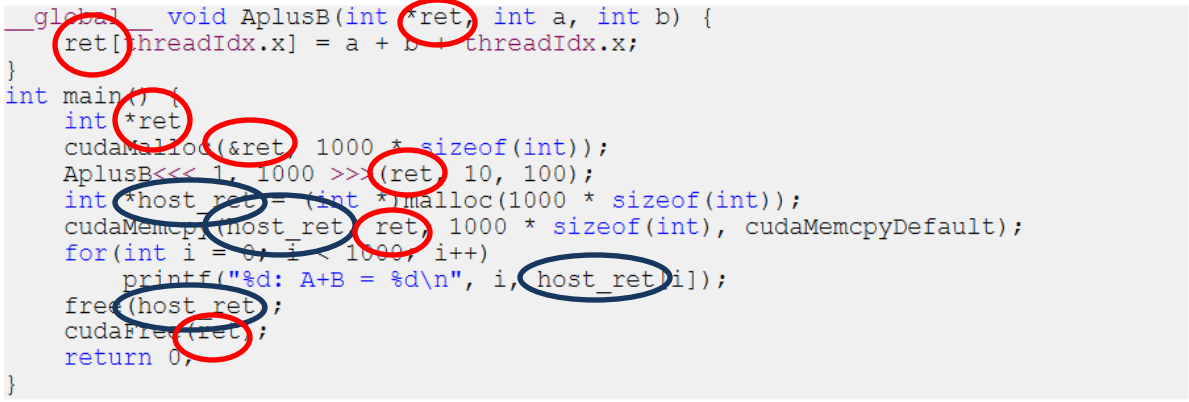
- Avsak az od verzie CC 6.0 je mozne pouzit na (de)alokaciu pamate funkcie OS (malloc()/free())

```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

- Zaujimavostou je podpora virtualnej pamate na grafickych kartach
- Od CC 6.0 je zavedena podpora 49-bitovej adresy na zariadeni… 48 pre virtualnu adresu OS a bit navyse, aby bolo mozne adresovat aj pamat zariadenia (bit navyse zdvojnasobuje…)
- Tato funkcionalita je zavedena kvoli tomu, aby mohli vypocty vyuzivat viac pamate, nez je fyzicka velkost pamate

CUDA streams

CUDA streams

- Prudy (streams) su nastrojom, pomocou ktoreho sa da lahko definovat, ktore kernely (hlavne funkcie spustane na GPU) su zavisle a ktore nie

CUDA streams

- Kernely vlozene do spolocneho prudu sa vykonavaju SEKVENCNE jeden za druhym
- Kernely z viacerych prudov sa vykonavaju KONKURENTNE

Strategie optimalizacie vykonu

Druha cast

Strategie optimalizacie vykonu

Strategie optimalizacie vykonu

1. Maximalizacia paralelneho vykonavania za ucelom maximalneho vytazenia zariadenia

Strategie optimalizacie vykonu

1. Maximalizacia paralelneho vykonavania za ucelom maximalneho vytazenia zariadenia

2. Optimalizacia vyuzivania pamate za ucelom maximalizacie pamatovej priepustnosti

Strategie optimalizacie vykonu

1. Maximalizacia paralelneho vykonavania za ucelom maximalneho vytazenia zariadenia

2. Optimalizacia vyuzivania pamate za ucelom maximalizacie pamatovej priepustnosti

3. Optimalizacia vyuzitia instrukcii za ucelom maximalizacie priepustnosti instrukcii

Strategie optimalizacie vykonu

- Ktoru strategiu pouzit na ktoru cast programu sa vo vseobecnosti neda povedat

- Da sa povedat, co kedy nepouzit
- Napriklad je zbytocne optimalizovat kod, ak je uzkym hrdlom aplikacie pristup k udajom v pamati

Strategie optimalizacie vykonu

- Optimalizacne usilie treba zamerat tym smerom, kde je uzke hrdlo systemu
- Ako?

Strategie optimalizacie vykonu

- Optimalizacne usilie treba zamerat tym smerom, kde je uzke hrdlo systemu
- Ako?

- Meranim a monitorovanim – napriklad pouzitim profilerov
- Porovnanim teoretickych moznosti zariadenia s vysledkami merania profilovacich nastrojov

Optimalizacia 1 – vyuzitie zariadenia

Optimalizacia 1 – vyuzitie zariadenia

- Aplikacia by mala byt tak strukturovana, aby v co najvacsej miere vyuzivala paralelizmus a ten vhodne mapovala na patricne komponenty systemu s cielom v co najvacsej miere vyuzit jeho kapacitu vypoctu

Optimalizacia 1 – vyuzitie zariadenia

- Aplikacia by mala byt tak strukturovana, aby v co najvacsej miere vyuzivala paralelizmus a ten vhodne mapovala na patricne komponenty systemu s cielom v co najvacsej miere vyuzit jeho kapacitu vypoctu
- Tri pohlady na vyuzitie zariadenia
  1. Aplikacna uroven
  2. Uroven zariadenia
  3. Uroven multiprocesora SM

Optimalizacia 1 – vyuzitie zariadenia

- Uroven aplikacie
  - Sucasne vyuzitie nezavislych casti systemu (CPU, GPU, zbernice) pomocou asynchronneho vykonavania kodu
  - Kazdy typ procesora by mal robit tu cinnost, pri ktorej ma najvyssi vykon (CPU seriove casti programu, GPU paralelene casti vypoctu)

Optimalizacia 1 – vyuzitie zariadenia

- Uroven aplikacie
  - Co ak pri paralelnom behu na GPU je nutne synchronizovat vypoctove vlakna (napr. kvoli vymene udajov)?

Optimalizacia 1 – vyuzitie zariadenia

- Uroven aplikacie
  - Co ak pri paralelnom behu na GPU je nutne synchronizovat vypoctove vlakna (napr. kvoli vymene udajov)?
  - Dve situacie mozu nastat: bud vlakna patria do toho isteho bloku alebo nie

Optimalizacia 1 – vyuzitie zariadenia

- Uroven aplikacie
  - Vlakna patria do toho isteho bloku, mozu vyuzit funkciu __syncthreads()
  - Vlakna nepatria do toho isteho bloku, udaje musia zdielat pomocou globalnej pamate zariadenia minimalne v dvoch (sekvencnych) volaniach kernel funkcii (jedna zapise do globalnej pamate udaje, druha ich cita)
  - Tento druhy pripad je velmi neoptimalny a mal by sa minimalizovat

Optimalizacia 1 – vyuzitie zariadenia

- Uroven zariadenia
  - Aplikacia by mala zapojit do vypoctu co najviac SM
  - Maximalizacia paralelneho behu kernelov sa da dosiahnut pomocou ich konkurentneho vykonavania pomocou prudov

	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 grids per device (Concurrent Kernel Execution)	16	4	32				16	128	32	16	128	

Optimalizacia 1 – vyuzitie zariadenia

- Uroven GPU multiprocesora
  - MP vyuziva paralelny beh vlakien vo warpoch
  - Teda vyuzitie zariadenia je priamo dane poctom warpov zapojenych do vypoctu
  - Pocet hodinovych cyklov nutnych na pripravenie vykonania instrukcie pre warp sa nazyva *latencia*
  - Plne vyuzitie warpu sa dosiahne, ak vsetky warp planovace maju v kazdom hodinovom cykle vzdy k dispozicii nejaku instrukciu na vykonanie

Optimalizacia 1 – vyuzitie zariadenia

- Uroven GPU multiprocesora
  - Najcastejsi pripad, kedy warp nie je pripraveny vykonat instrukciu, nastava, ked este nie je pripraveny operand instrukcie

Optimalizacia 1 – vyuzitie zariadenia

- Uroven GPU multiprocesora
  - Najcastejsi pripad, kedy warp nie je pripraveny vykonat instrukciu, nastava, ked este nie je pripraveny operand instrukcie
    - Ak ide o operand v registri, latencia je v jednotkach cyklov
    - Ak ide o operand v pamati, moze ist o stovky cyklov!

Optimalizacia 1 – vyuzitie zariadenia

- Uroven GPU multiprocesora
  - Dalsim dovodom latencie moze byt synchronizacne obmedzenie (__syncthreads())
  - Kedze cakat na seba mozu iba vlakna toho isteho bloku, tak v tomto pripade je vhodne zvysit pocet blokov programu

	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

Optimalizacia 1 – vyuzitie zariadenia

- Uroven GPU multiprocesora
  - Na vyuzitie MP ma obrovsky vplyv aj pocet pouzitych registrov!
  - 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

	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 mozu byt 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										

– Ked by sme pridali co i len 1 register navyse, na MP bude moct bezat uz len jeden blok!

– Takze sa moze stat, ze takmer polovica warpov bude na MP nevyuzita!

Optimalizacia 1 – vyuzitie zariadenia

- Uroven GPU multiprocesora
  - Pri vyuziti registrov si treba dat pozor na to, ze VZDY sa “ukrajuje” z miesta vyhradeneho pre registre MP nasobok 32
  - 8-bitova premenna zaberie 32 bitov
  - 48-bitova premenna zaberie 64 bitov
  - atd

Optimalizacia 1 – vyuzitie zariadenia

- Uroven GPU multiprocesora
  - Experimenty, experimenty a este raz experimenty!
  - Zaroven s tym studium parametrov zariadenia
  - Pocet vlakien v bloku VZDY volime ako nasobok velkosti warpu, aby sme nemrhali vypoctovymi zdrojmi (nastastie, velkost warpu je jedna z mala hodnot, ktore sa zatial nikdy nemenili…)

Optimalizacia 1 – vyuzitie zariadenia

- Uroven GPU multiprocesora
  - Experimenty, experimenty a este raz experimenty!

	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**
Warp size	32											

  - Pocet vlakien v bloku VZDY volime ako nasobok velkosti warpu, aby sme nemrhali vypoctovymi zdrojmi (nastastie, velkost warpu je jedna z mala hodnot, ktore sa zatial nikdy nemenili...)

Optimalizacia 1 – vyuzitie zariadenia

- Situacia nie je uplne zufala… jestvuje par API funkcii, ktore pomahaju vypocitat optimalny pocet blokov na zaklade vyuzitia registrov a zdielanej pamate

Optimalizacia 1 – vyuzitie zariadenia

- Situacia nie je uplne zufala… jestvuje par API funkcii, ktore pomahaju vypocitat optimalny pocet blokov na zaklade vyuzitia registrov a zdielanej pamate

- cudaOccupancyMaxActiveBlocksPerMultiprocessor vracia pocet konkurentne vykonavatelnych blokov na 1 MP

Optimalizacia 1 – vyuzitie zariadenia

- Vratena hodnota tejto funkcie (#blocks/MP) sa da prepocitat aj na ine metriky
  - Vynasobenim WarpsPerBlock dostaneme pocet konkurentne vykonavatelnych warpov na 1 MP
  - Vydelenim takto ziskanej hodnoty maximalnym poctom warpov na 1 MP dostaneme vyuzitie

	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;
}
```

Optimalizacia 1 – vyuzitie zariadenia

- Jestvuju dalsie API funkcie na heuristicku analyzu
- Vid dokumentacia…

Optimalizacia 2 – priepustnost pamate

Optimalizacia 2 – priepustnost pamate

- Zasadnym krokom je minimalizacia prenosu udajov

- Ak to nie je mozne, nerobit prenosy s malou sirkou pasma (t.j. CPU vs GPU)

Optimalizacia 2 – priepustnost pamate

- Najhorsie su na tom prenosy medzi CPU a GPU
- Potom prenosy medzi globalnou pamatou GPU a zdielanou pamatou
- Nasleduje prenos medzi zdielanou pamatou a suborom registrov

Optimalizacia 2 – priepustnost pamate

- Ake kroky musi urobit vlakno, ked chce preniest udaje z/do globalnej pamate GPU

Optimalizacia 2 – priepustnost pamate

- Ake kroky musi urobit vlakno, ked chce preniest udaje z/do globalnej pamate GPU
  - Udaje z pamate GPU do zdielanej pamate
  - Synchronizovat sa s ostatnymi vlaknami bloku (aby sa zaistila validita zdielanych udajov pre vsetky vlakna bloku)
  - Spracovat udaje
  - Znovu sa synchronizovat (ak je to potrebne)
  - Zapisat udaje do pamate GPU

Optimalizacia 2 – priepustnost pamate

- Zvlast si treba davat pozor na pristupy vlakien do globalnej pamate zariadenia (o tom este bude rec)

Optimalizacia 2 – priepustnost pamate

- Typy pristupov, ktore teda treba riesit

Optimalizacia 2 – priepustnost pamate

- Typy pristupov, ktore teda treba riesit

- Prenos udajov medzi hostom a zariadenim
- Prenos udajov medzi (globalnou) pamatou zariadenia a zdielanou/lokalnou pamatou
  - Pamat GPU sa deli na 5 typov
  - globalna pamat, pamat konstant, pamat textur, zdielana pamat a lokalna pamat

Optimalizacia 2 – priepustnost pamate

- Prenos host - zariadenie

Optimalizacia 2 – priepustnost pamate

- Prenos host – zariadenie
  - Vzdy treba minimalizovat tento typ prenosu
  - Napriklad tym, ze sa vacsia cast kodu prenesie z hosta na zariadenie (aj za cenu toho, ze sa naplno nevyuzije paralelizmus zariadenia – ked budu vlakna medzi sebou blokovane)

Optimalizacia 2 – priepustnost pamate

- Prenos host – zariadenie
  - Vzdy treba minimalizovat tento typ prenosu
  - Napriklad tym, ze sa vacsia cast kodu prenesie z hosta na zariadenie (aj za cenu toho, ze sa naplno nevyuzije paralelizmus zariadenia – ked budu vlakna medzi sebou blokovane)
  - Ak uz treba prenos robit, tak je vhodne agregovat viac mensich prenosov do jedneho vacsieho

Optimalizacia 2 – priepustnost pamate

- Pamatove pristupy v ramci zariadenia

Optimalizacia 2 – priepustnost pamate

- Pamatove pristupy v ramci zariadenia

- Globalna pamat
- Lokalna pamat
- Zdielana pamat
- Pamat konstant
- Pamat textur

Optimalizacia 2 – priepustnost pamate

- Globalna pamat

- Pristup do tejto pamate sa robi v transakciach o velkosti 32, 64 alebo 128 bajtov
- Prva adresa, na ktoru sa v ramci transakcie pristupuje, musi byt zarovnana na nasobok velkosti transakcie!!!

Optimalizacia 2 – priepustnost pamate

- Globalna pamat

- Ked warp vykonava instrukciu, ktora pristupuje ku globalnej pamati, pamatove pristupy vlakien v ramci warpu spoji do jednej alebo viacerych transakcii v zavislosti od
  1. Velkosti udajov, ku ktorym vlakno pristupuje
  2. Samotnych adries, ku ktorym sa pristupuje

Optimalizacia 2 – priepustnost pamate

- Globalna pamat

- Cim viac transakcii pre vlakna warpu treba urobit, tym viac sa zmensuje priepustnost
  - Majme napriklad 32-bajtove transakcie, ale tak nestastne, ze kazde vlakno z takejto transakcie vyuzije iba 4 bajty
  - 32/4 = 8, takze 8x sa nam zmensi priepustnost pamate

Optimalizacia 2 – priepustnost pamate

- Globalna pamat

- Kolko transakcii na vykonanie instrukcie warpu je potrebnych a ako to ovplyvni priepustnost, zavisi od CUDA verzie, ktoru zariadenie podporuje

Optimalizacia 2 – priepustnost pamate

- Globalna pamat

- Vo vseobecnosti plati, ze je potrebne co najviac zlucovat pamatove pristupy do co najmensieho poctu transakcii
- Ako?

Optimalizacia 2 – priepustnost pamate

- Globalna pamat

1. Nastudovat si optimalny vzor pristupu do globalnej pamate pre tu-ktoru verziu CC
2. Pouzit take udajove typy, velkosti a zarovnania, ktore vyhovuju poziadavkam uvedenymi dalej v prednaske
3. Ak je to vhodne, urobit explicitne doplnenie (*padding*) udajov, vid dalej v prednaske

Optimalizacia 2 – priepustnost pamate

- Globalna pamat – poziadavky velkosti a zarovnania

- Instrukcie pristupu do globalnej pamate maju velkosti operandov 1, 2, 4, 8 alebo 16 bajtov
- Pristup do pamate sa realizuje JEDNOU instrukciou iba vtedy, ak je tato velkost zachovana a udaje su na adrese, ktora je nasobkom tejto velkosti operandu!

Optimalizacia 2 – priepustnost pamate

- Globalna pamat – poziadavky velkosti a zarovnania

- Ak nie je poziadavka velkosti a zarovnania zachovana, pristup sa rozdeli do viacerych instrukcii, ktore sa nemusia dat spojit do jednej transakcie

Optimalizacia 2 – priepustnost pamate

- Globalna pamat – dvojrozmerne polia

- Vseobecny vzor pristupu do pamate v pripade dvojrozmernych poli je nasledovny

Optimalizacia 2 – priepustnost pamate

- Globalna pamat – dvojrozmerne polia

- Vseobecny vzor pristupu do pamate v pripade dvojrozmernych poli je nasledovny
  - Vlakno ma spracovat prvok na (tx, ty) pozicii
  - Riadok dvojrozmerneho pola ma sirku width
  - Pole zacina na adrese BaseAddress
  - Adresa typu splna poziadavky zarovnania

Optimalizacia 2 – priepustnost pamate

- Globalna pamat – dvojrozmerne polia

- Adresa prvku je potom

$$BaseAddress + width * ty + tx$$

- Aby takyto pristup bol plne zluceny (*coaleshed*), sirka pola aj sirka bloku vlakien musia byt nasobkom velkosti warpu!

Optimalizacia 2 – priepustnost pamate

- Globalna pamat – dvojrozmerne polia

- Ak nejake pole nema sirku riadku rovnu nasobku velkosti warpu, je potrebne taketo pole alokovat tak, ze sa sirka riadku zarovna na nasobok velkosti warpu a zvysky riadkov sa pri pouzivani ignoruju (*padding*)

Optimalizacia 2 – priepustnost pamate

- Globalna pamat – dvojrozmerne polia

- Takato alokacia sa da robit HW nezavislo
- Vid funkcie CUDA API
  - cudaMallocPitch()
  - cuMemAllocPitch()
  - …

Optimalizacia 2 – priepustnost pamate

- Lokalna pamat

- Vyuziva sa pre automaticke premenne, ktore generuje kompilator
  - Struktury, ktore zaberu prilis vela registrov
  - Ak uz nie je miesto v subore registrov (tzv. *register spilling*)
  - Ine

Optimalizacia 2 – priepustnost pamate

- Lokalna pamat

- Lokalna pamat sa nachadza v pamati zariadenia, takze pre pristupy do nej platia tie iste pravidla a obmedzenia ako pre globalnu pamat

Optimalizacia 2 – priepustnost pamate

- Lokalna pamat

- Pristupy do lokalnej pamate sa uchovavaju vo vyrovnavacej pamati L2 podobne ako pristupy do globalnej pamate
- To plati pre niektore zariadenia CC 3.x a vsetky zariadenia CC 5.x a 6.x

Optimalizacia 2 – priepustnost pamate

- Lokalna pamat

- Lokalna pamat ma svoj nazov nie podla fyzickeho umiestnenia, ale podla rozsahu platnosti
- Nazov je velmi zavadzajuci…

Optimalizacia 2 – priepustnost pamate

- Zdielana pamat

- Nachadza sa *on-chip*, takze prenos ma ovela mensiu latenciu nez pristup do globalnej ci lokalnej pamate

Optimalizacia 2 – priepustnost pamate

- Zdielana pamat

- Aby sa dosiahla vysoka priepustnost na urovni samotneho hw, zdielana pamat je na urovni hw rozdelena do rovnako velkych pamatovych modulov nazyvanych banky (*banks*)
- Ku bankam je mozne realizovat SUCASNY pristup

Optimalizacia 2 – priepustnost pamate

- Zdielana pamat

- Ak N ziadosti o pristup k pamati padne do N roznych pamatovych bank, ziadosti je mozne obsluzit subezne
- Ak dve ROZNE adresy ziadosti o pristup padnu do tej istej banky, ide o konflikt a ziadosti su vybavene sekvencne

Optimalizacia 2 – priepustnost pamate

- Zdielana pamat

- HW automaticky rozdeluje konflikty do potrebneho poctu nekonfliktnych pristupov
- Tym sa znizuje priepustnost umerne poctu nezavislych pristupov

Optimalizacia 2 – priepustnost pamate

- Zdielana pamat

- Ak pocet nezavislych pristupov, ktore obsluzia pociatocnu poziadavku, je N, hovorime, ze pociatocna poziadavka vyvolala N-nasobny konflikt pamatovych bank

Optimalizacia 2 – priepustnost pamate

- Zdielana pamat

- Ak chceme vyuzit plny vykon pristupov, je potrebne nahliadnut do dokumentacie pre tuktoru CUDA verziu
- Ako hw mapuje adresy pamatovych pristupov do jednotlivych bank

Optimalizacia 2 – priepustnost pamate

- Pamat konstant
- Nachadza sa v pamati zariadenia a ma zvlast vyrovnavaciu pamat konstant
- Poziadavka o pristup do pamate je rozdelena na tolko pristupov, kolko roznych adries sa v pociatocnej poziadavke nachadza

Optimalizacia 2 – priepustnost pamate

- Pamat textur

- Rozdiel oproti pamati konstant je v tom, ze pamat textur je optimalizovana na 2D dimenziu
- T.j. ziadosti o pristup vlakien toho isteho warpu sa optimalne zlucuju

Optimalizacia 2 – priepustnost pamate

- Pamat textur
- Sidli v pamati zariadenia a podobne ako pamat konstant, ma vlastnu vyrovnavaciu pamat
- Cena pristupu je podobne ako pri konstantach dana tym, ci sa udaj nachadza vo vyrovnavacej pamati alebo nie

Optimalizacia 3 – priepustnost instrukcii

Optimalizacia 3 – priepustnost instrukcii

- Maximalizaciu priepustnosti instrukcii aplikacia dosiahne, ak
  1. Minimalizuje pouzitie aritmetickych instrukcii, ktore dlho trvaju (pouzitie *intrinsic* funkcii miesto klasickych, pouzitie *float* miesto *double*, …)
  2. Minimalizuje divergenciu vykonavaneho kodu v ramci warpu (*if-else*)
  3. Minimalizuje pocet pouzitych instrukcii

Optimalizacia 3 – priepustnost instrukcii

- Priepustnost instrukcii sa meria v pocte operacii za jednotku casu (cyklu) na jeden multiprocesor

- Kedze vzdy sa planuje na beh aspon 1 warp vlakien, tak pre velkost warpu 32 jedna instrukcia zodpoveda vykonaniu 32 operacii

Optimalizacia 3 – priepustnost instrukcii

- Ak mame vysledny pocet operacii za jednotku casu N, potom je priepustnost dana hodnotou N/32 instrukcii za jednotku casu

- Priepustnost je pocitana na jeden MP
- Ak ju chceme vyjadrit pre cele zariadenie, ziskanu hodnotu je potrebne vynasobit poctom MP zariadenia

Optimalizacia 3 – priepustnost instrukcii

- Pri vypoctoch treba zohladnovat 3 skupiny instrukcii

1. Aritmeticke instrukcie
2. Instrukcie ovplyvnujuce tok riadenia
3. Synchronizacne instrukcie

Pristupy do pamati zariadenia podla CUDA verzie

Tretia cast

- CUDA CC 3.x (globalna a zdielana pamat)
- CUDA CC 5.x (globalna a zdielana pamat)
- CUDA CC 6.x (globalna a zdielana pamat)
- CUDA CC 7.x (globalna a zdielana pamat)

Globalna pamat

Globalna pamat

- V pripade podpory vyrovnavacej pamate plati, ze:

- riadok vyrovnavacej pamate je velky 128B
- mapuje sa na 128B pamate zariadenia
- adresa, na ktoru sa mapuje, je zarovnana na hodnotu 128

Globalna pamat

- Pre ziadost o pristup pre vlakna warpu plati

- Ak je velkost operandu (kazdeho) vlakna 1, 2 alebo 4B, obsluzi sa jedinou transakciou
- Ak je velkost operandu 8B, obsluzi sa dvoma pristupmi (pre kazdu polovicu warpu zvlast)
- Ak je velkost operandu 16B, obsluzi sa styrmi transakciami

Globalna pamat

- Pristupy do globalnej pamate vo verziach CUDA CC 5.x, 6.x a 7.x idu vzdy cez vyrovnavaciu pamat

- Pristupy vo verzii CC 3.x sa lisia (vid specifikacie jednotlivych subverzii)

Zdielana pamat

Zdielana pamat

- Zdielana pamat ma 32 bank
- Iba CUDA CC 3.x: s moznostou dvoch adresnych rezimov (tie sa daju nastavovat a zistovat pomocou API funkcii)

Zdielana pamat

- Zdielana pamat ma 32 bank
- Iba CUDA CC 3.x: s moznostou dvoch adresnych rezimov (tie sa daju nastavovat a zistovat pomocou API funkcii)

1. 64-bitovy rezim (za sebou iduce 64-bitove slova sa mapuju do za sebou iducich bank)
2. 32-bitovy rezim (za sebou iduce 32-bitove slova sa mapuju do za sebou iducich bank)

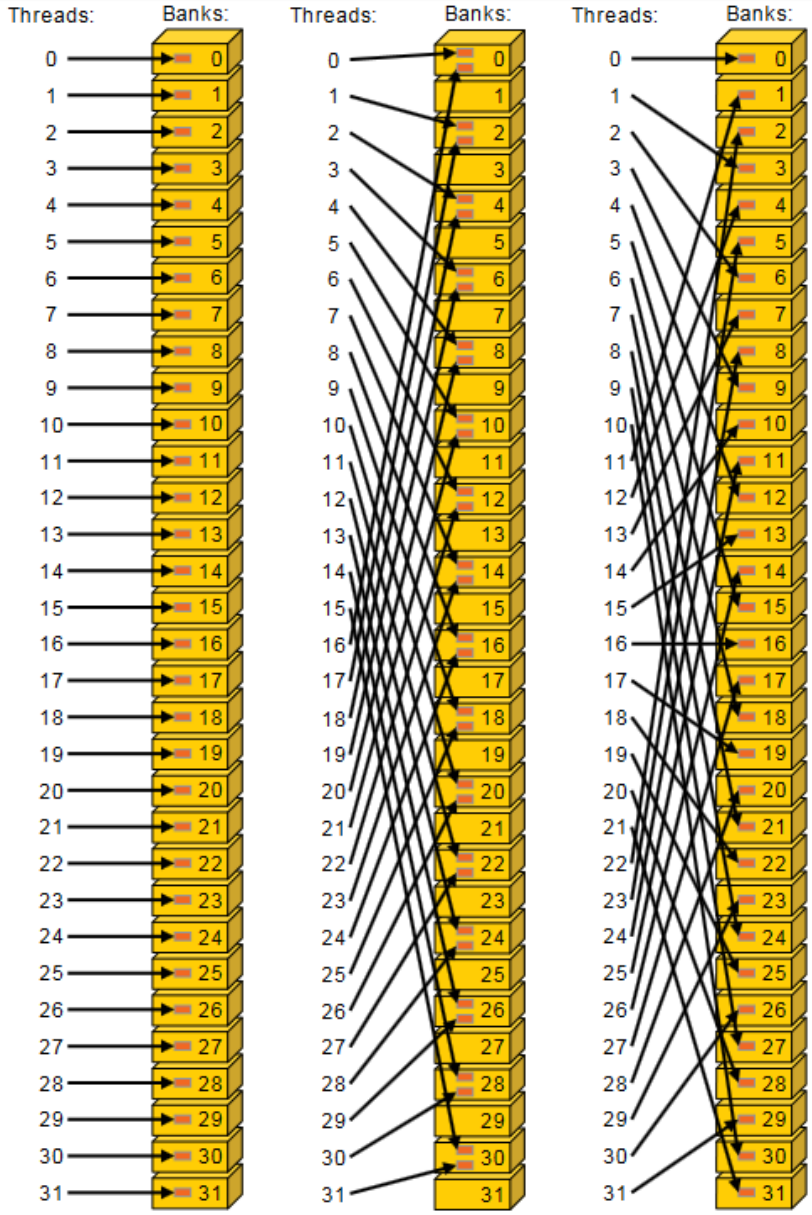
Zdielana pamat

- V pripade CUDA CC 7.x je moznost pomocou API volani nastavit velkost zdielanej pamate
- Konflikt sa nedeje, ak dve vlakna warpu nepristupuju k adrese (v ramci toho isteho slova) v ramci jednej banky
- Co v pripade konfliktu?

Zdielana pamat

- Ak je pristup vlakien na citanie, hodnota slova z banky sa doda vsetkym vlaknam

- Ak je pristup na zapis, na kazdu adresu zapise prave jedno z vlakien (ktore superia o zapis na adresu), pricom nie je definovane, ktore



Left

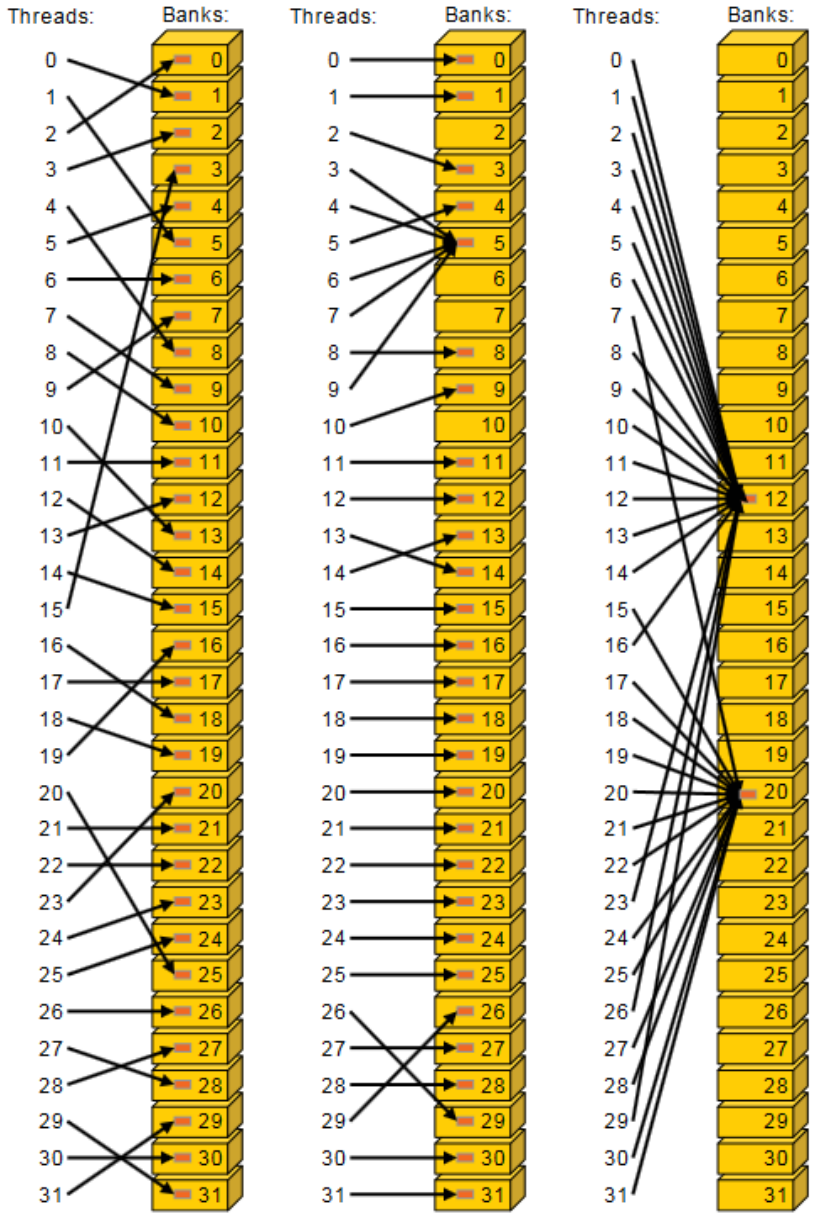
Linear addressing with a stride of one 32-bit word (no bank conflict).

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).

Dalsie zdroje studia

Dalsie zdroje

- Occupancy Calculator: https://docs.nvidia.com/cuda/cuda-occupancy-calculator/CUDA_Occupancy_Calculator.xls

- Device Memory Spaces: https://docs.nvidia.com/cuda/archive/10.1/cuda-c-best-practices-guide/index.html#device-memory-spaces

- Device Memory Spaces: https://docs.nvidia.com/cuda/archive/10.1/cuda-c-best-practices-guide/index.html#device-memory-spaces

Dakujem za pozornost