# Arhitectura GPU NVIDIA CUDA

Arhitectura NVIDIA FERMI aici [https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf], Tesla 2050, fep queue ibm-dp.q

Arhitectura NVIDIA KEPLER aici [https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf], Tesla K40M, fep queue hp-sl.q

Prima arhitectura NVIDIA complet programabila a fost G80 (ex. Geforce 8800 [http://www.nvidia.com/page/8800_tech_briefs.html], lansat in anul 2006). Cu aceasta arhitectura s-a trecut de la unitati hardware fixe vertex/pixel la cele de unified shader care puteau procesa atat vertex/pixel cat si geometry. Evolutia arhitecturilor GPU de la NVIDIA este detaliata aici [http://s08.idav.ucdavis.edu/luebke-nvidia-gpu-architecture.pdf].

Implementarea NVIDIA pentru GPGPU se numeste CUDA (Compute Unified Device Architecture) si permite utilizarea limbajului C pentru programarea pe GPU-urile proprii. Lista de GPU-uri ce suporta API-ul CUDA sau OpenCL se regaseste pe site-ul oficial aici [https://www.geforce.com/hardware/technology/cuda/supported-gpus] sau pe wiki aici [https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units]. Fiecare noua arhitectura are un codename (ex Fermi, Pascal) si este reprezentata de un “compute capability” (list aici [https://developer.nvidia.com/cuda-gpus]). Cu cat arhitectura este mai noua, cu atat sunt suportate mai multe facilitati din API-urile CUDA si OpenCL.

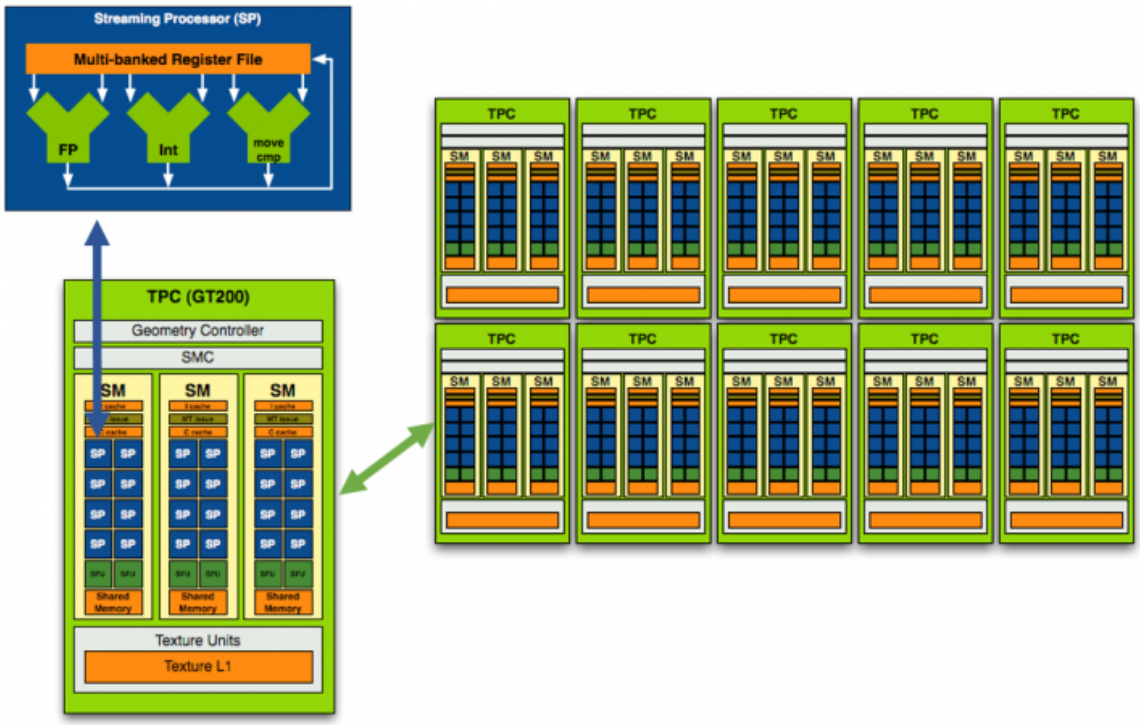
Unitatea GPU este potrivita pentru paralelismul de date SIMD (Single Instruction Multiple Data), astfel aceleasi instructiuni sunt executate in paralel pe mai multe unitati de procesare. Datorita faptului ca acelasi program este executat pentru fiecare element de date, sunt necesare mai putine elemente pentru controlul fluxului. Si deoarece calculele sunt intensive computational, latenta accesului la memorie poate fi ascunsa prin calcule in locul unor cache-uri mari pentru date.

Motivul discrepantei intre performanta paralela dintre CPU si GPU este faptul ca GPU sunt specializate pentru procesare masiv paralela si intensiva computational (descrierea perfecta a taskurilor de randare grafica) si construite in asa fel incat majoritatea tranzistorilor de pe chip se ocupa de procesarea datelor in loc de cachingul datelor si controlul fluxului executiei.

La GPU-urile NVIDIA, un Streaming Processor (SP) este un microprocesor cu executie secventiala, ce contine un pipeline, unitati aritmetico-logice (ALU) si de calcul in virgula mobila (FPU). Nu are un cache, fiind bun doar la executia multor operatii matematice. Un singur SP nu are performante remarcabile, insa prin cresterea numarului de unitati, se pot rula algoritmi ce se preteaza paralelizarii masive.

SP impreuna cu Special Function Units (SFU) sunt incapsulate intr-un Streaming Multiprocessor (SM/SMX). Fiecare SFU contine unitati pentru inmultire in virgula mobila, utilizate pentru operatii transcendente (sin, cos) si interpolare. MT se ocupa cu trimiterea instructiunilor pentru executie la SP si SFU.

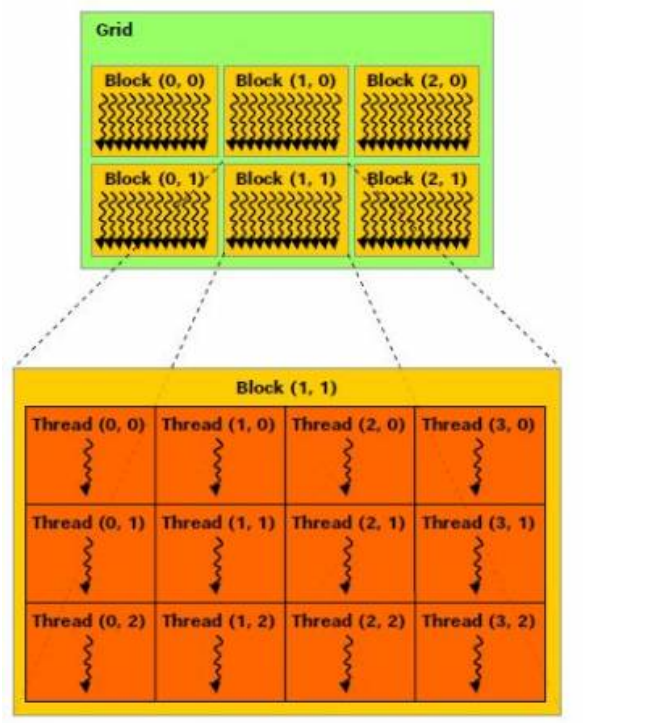
Pe langa acestea, exista si un cache (de dimensiuni reduse) pentru instructiuni, unul pentru date precum si memorie shared, partajata de SP-uri. Urmatorul nivel de incapsulare este Texture / Processor Cluster (TPC). Acesta contine SM-uri, logica de control si un bloc de handling pentru texturi. Acest bloc se ocupa de modul de adresare al texturilor, logica de filtrare a acestora precum si un cache pentru texturi.




Filosofia din spatele arhitecturii este permiterea rularii unui numar foarte mare de threaduri. Acest lucru este facut posibil prin paralelismul existent la nivel hardware.

Documentatia NVIDIA recomanda rularea unui numar cat mai mare threaduri pentru a executa un task. Arhitectura CUDA de exemplu suporta zeci de mii de threaduri, numarul acestora fiind mult mai mare decat unitatile fizice existente pe chip. Acest lucru se datoreaza faptului ca un numar mare de threaduri poate masca latenta accesului la memorie.

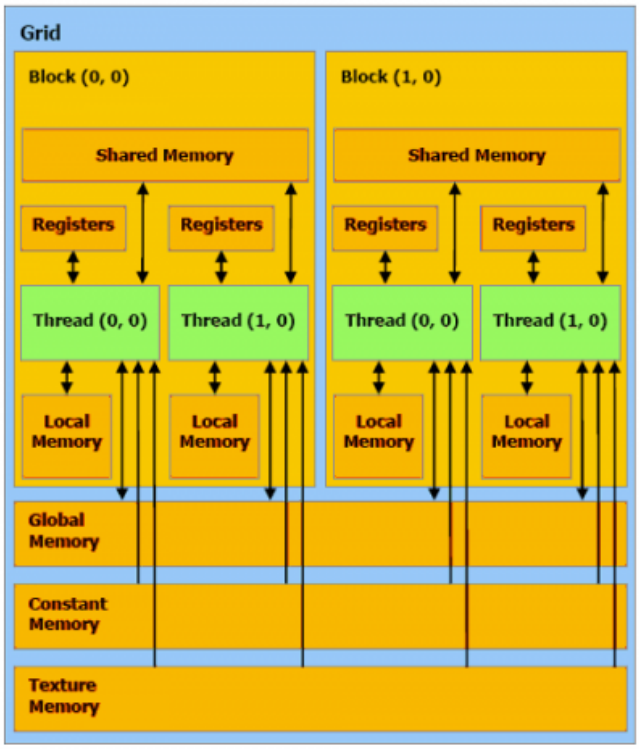
Urmarind acelasi model modular ca si arhitectura, threadurile sunt incapsulate in blocuri (thread blocks / warps), iar blocurile in grile (thread grid). Fiecare thread este identificat prin indexul threadului in bloc, indexul blocului in grila si indexul grilei. Indexurile threadurilor si ale blocurilor pot fi uni/bi/tri-dimensionale, iar indexul grilei poate fi uni sau bi-dimensional. Acest tip de impartire are rolul de a usura programare pentru probleme ce utilizeaza structuri de date cu mai multe dimensiuni. Se poate observa ca thread-urile dintr-un thread block trebuie sa execute cat mai multe instructiuni identice spre a nu irosi resurse.




Threadurile dintr-un bloc pot coopera prin partajarea de date prin intermediul memoriei shared si prin sincronizarea executiei. Functia de bariera functioneaza doar pentru threadurile dintr-un bloc. Sincronizarea nu este posibila la alt nivel (intre blocuri/grila etc.). Mai multe explicatii se regasesc in Laboratorul 9 [http://cs.curs.pub.ro/wiki/asc/asc:lab9:index].

## Ierarhia de memorie

Intelegerea ierharhiei de memorie este esentiala in programarea eficienta a unitatii GPU. Capacitatea mare de executie in paralel a unui GPU necesita ascunderea latentei de acces catre memoria principala (fie VRAM pentru dGPU sau RAM pentru iGPU).




**Register File**

```
/* marcam pentru compilator regValPi in register file */
__private float regValPi = 3.14f;
```

```
/* compilatorul cel mai probabil oricum incadreaza regVal2Pi ca registru */
float regVal2Pi = 2 * 3.14f;
```

- Cea mai rapida forma de memorie de pe GPU
- Accesibila doar de catre thread, durata de viata este aceeasi ca si a threadului
- Un kernel complex poate determina folosirea unui numar mare de registrii si astfel:
  - limitarea executiei multor thread-uri simultan
  - register spill, atunci cand valorile registrilor sunt salvate in memoria globala

#### Local Memory

```
/* fiecare work item salveaza un element */
__local float lArray[lid] = data[gid];
```

- in functie de implementarea hardware, 100GB/sec → 2TB/sec
- pentru GPU o memorie rapida, actioneaza ca un cache L1/alt register file, la CPU de regula este doar o portiune din RAM
- accesibila tuturor threadurilor dintr-un bloc (warp/wavefront), durata de viata este aceeasi ca si a blocului
- trebuie evitate conflictele de access (bank conflicts)

#### Constant Memory

```
__const float pi = 3.14f
```

- in functie de implementarea hardware, 100GB/sec → 1TB/sec
- in general performanta foarte buna, (cache L1/L2, zona dedicata),
- are durata de viata a aplicatiei kernel

#### Global Memory

```
__kernel void process(__global float* data){ ... }
```

- in functie de implementarea hardware, 30GB/sec → 500GB/sec
- Video RAM (VRAM), de regula cu o capacitate intre 1GB si 12GB in functie de placa video
- memorie dedicata specializata doar pentru placile grafice discrete (GPU-urile integrate in CPU folosesc RAM)
- in general latime mare de banda (256-512 biti) si chipuri de memorii de mare viteza (GDDR5)

#### Host Memory (RAM)

- in general, 4GB/sec → 30GB/sec
- pentru acces din kernel trebuie transfer/mapare explicita RAM→VRAM pe partea de host/CPU
- memoria RAM accesibila direct de CPU si indirect de GPU via DMA si magistrala PCIe
- viteza de transfer (throughput/latenta) este limitata de magistrala PCIe cat si de memoria RAM

Caracteristici GPU K40m (coada hp-sl.q), via query device properties CUDA

```
Device 0: "Tesla K40m"
  CUDA Driver Version / Runtime Version          9.1 / 9.1
  CUDA Capability Major/Minor version number:    3.5
  Total amount of global memory:                 11441 MBytes (11996954624 bytes)
  (15) Multiprocessors, (192) CUDA Cores/MP:     2880 CUDA Cores
  GPU Max Clock rate:                            745 MHz (0.75 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  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
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 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 Domain ID / Bus ID / location ID:   0 / 8 / 0
```

Caracteristici GPU M2070 (coada ibm-dp.q), via query device properties CUDA

```
Device 0: "Tesla M2070"
  CUDA Driver Version / Runtime Version          9.1 / 9.1
  CUDA Capability Major/Minor version number:    2.0
  Total amount of global memory:                 5302 MBytes (5559156736 bytes)
  (14) Multiprocessors, ( 32) CUDA Cores/MP:     448 CUDA Cores
```

```
  GPU Max Clock rate:                            1147 MHz (1.15 GHz)
  Memory Clock rate:                             1566 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 786432 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65535), 3D=(2048, 2048, 2048)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (65535, 65535, 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 Domain ID / Bus ID / location ID:   0 / 20 / 0
```

## Optimizare accesului la memorie

Modul cum accesam memoria impacteaza foarte mult performanta sistemului. Cum putem avea arhitecturi foarte diferite din pct de vedere al ierarhiei de memorie este important de inteles ca nu putem dezvolta un program care sa ruleze optim in toate cazurile. Un program CUDA este portabil caci poate fi usor rulat pe diferite arhitecturi NVIDIA CUDA, insa de cele mai multe ori trebuie ajustat in functie de arhitectura pentru o performanta optima.

In general pentru arhitecturile de tip GPU, memoria locala este impartita in module de SRAM identice, denumite bancuri de memorie (memory banks). Fiecare banc contine o valoare succesiva de 32 biti (de exemplu, un int sau un float), astfel incat accesele consecutive intr-un array provenite de la threaduri consecutive sa fie foarte rapid. Bank conflicts au loc atunci cand se fac cereri multiple asupra datelor aflate in acelasi banc de memorie.

Conflictele de access la bancuri de memorie (cache) pot reduce semnificativ performanta.

Cand are loc un bank conflict, hardware-ul serializeaza operatiile cu memoria (warp/wavefront serialization), si face astfel toate threadurile sa astepte pana cand operatiile de memorie sunt efectuate. In unele cazuri, daca toate threadurile citesc aceeasi adresa de memorie shared, este invocat automat un mecanism de broadcast iar serializarea este evitata. Mecanismul de broadcast este foarte eficient si se recomanda folosirea sa de oricate ori este posibil.
Spre exemplu daca linia de cache este alcatuita din 16 bancuri de memorie. Avem urmatoarele situatii care impacteaza performanta accesului la cache.

```
__kernel func(...) {
...
   __local int *array;
   x = array[get_local_id(0)];    // performanta 100%, 0 bank conflicts
   x = array[get_local_id(0)+1];  // performanta 100%, 0 bank conflicts
   x = array[get_local_id(0)*4];  // performanta 25%, 4 bank conflicts
   x = array[get_local_id(0)*16]; // performanta 6%,  16 bank conflicts
...
}
```

In cazul arhitecturilor de tip CPU, memoria locala este doar o regiune din RAM. Optimizarile pentru a tine datele critice in memoria locala pentru GPU nu ar prezenta deci aceleasi imbunatatiri de performanta.

## Debug aplicatii CUDA

Cele mai des intalnite probleme sunt cele de acces invalid la memorie. Nu de putine ori vom observa ca aceste accese invalide pot crea efecte secundare sau erori ce apar/sunt semnalate abia ulterior.
Sa luam de exemplu cazul in care 1 thread acceseaza 1 element de date si sa dam in executie mai multe thread-uri decat elemente de memorie alocate (8×16 ~ 128 thread-uri vs 100 elemente), dat fiind MAGNITUDE=1.

```
...
#define MAGNITUDE       (1)
#define NUM_BLOCKS      8 * MAGNITUDE
#define NUM_THREADS     16
#define NUM_ELEM        100 * MAGNITUDE

__global__ void kernel_compute(int* data) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    // invalid access
    data[idx] = 1111 * idx;
}

int main(int argc, char *argv[]) {
    int* data = NULL;

    HANDLE_ERROR( cudaMalloc(&data, 1 * sizeof(int)) );

    // launch kernel
    kernel_compute<<<NUM_BLOCKS, NUM_THREADS>>>(data);
    HANDLE_ERROR( cudaDeviceSynchronize() );
```

```
    return 0;
}
```

Daca rulam programul vom observa ca nu intoarce nici o eroare. Deoarece sunt putine accese invalide HW-ul nu semnaleaza vreo problema.

CUDA insa ofera aplicatii care sa analizeze si sa detecteze accese invalide cu o precizie ridicata. Daca rulam de exemplu cuda-memcheck [https://docs.nvidia.com/cuda/cuda-memcheck/index.html] vom vedea instant ca avem accese invalide la memorie.

```
$ cuda-memcheck ./example_debug
...
========= Invalid __global__ write of size 4
=========     at 0x00000050 in kernel_compute(int*)
=========     by thread (0,0,0) in block (7,0,0)
=========     Address 0x13059c01c0 is out of bounds
=========     Saved host backtrace up to driver entry point at kernel launch time
=========     Host Frame:/usr/lib64/nvidia/libcuda.so.1 (cuLaunchKernel + 0x2cd) [0x22b12d]
=========     Host Frame:./example_debug [0x1a2ab]
=========     Host Frame:./example_debug [0x374fe]
=========     Host Frame:./example_debug [0x3650]
=========     Host Frame:./example_debug [0x354b]
=========     Host Frame:./example_debug [0x3565]
=========     Host Frame:./example_debug [0x349e]
=========     Host Frame:/lib64/libc.so.6 (__libc_start_main + 0xf5) [0x223d5]
=========     Host Frame:./example_debug [0x3289]
=========
...
=========
========= ERROR SUMMARY: 128 errors
```

Daca insa avem multe accese invalide (de exe '#define MAGNITUDE (1024 * 1024)') o sa vedem ca API-ul arunca erori la executia de kernel.

```
$ ./example_debug
an illegal memory access was encountered in example_debug.cu at line 33
```

In acest caz eroare semnalata apare la cudaDeviceSynchronize() desi problema este la kernel.

Folosind insa cuda-gdb [http://developer.download.nvidia.com/GTC/PDF/1062_Satoor.pdf] putem gasi rapid ca problema este la executia de kernel, atunci cand se acceseaza zone de memorie nealocate.

```
$ cuda-gdb example_debug
NVIDIA (R) CUDA Debugger
9.1 release
...
(cuda-gdb) run
Starting program: lab8_sol/example_debug
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
[New Thread 0x2aaaacecf700 (LWP 20762)]
[New Thread 0x2aaaad0d0700 (LWP 20763)]

CUDA Exception: Device Illegal Address
The exception was triggered in device 0.

Thread 1 "example_debug" received signal CUDA_EXCEPTION_10, Device Illegal Address.
[Switching focus to CUDA kernel 0, grid 1, block (16731,0,0), thread (0,0,0), device 0, sm 8, warp 13, lane 0]
0x0000000000ad4380 in kernel_compute(int*)<<<(8388608,1,1),(16,1,1)>>> ()
```

## Analiza de performanta in aplicatiile CUDA

In aceast sectie vom explora cateva metode pentru a evalua performantele programelor CUDA.

### Timing via executie kernel (host/CPU)

Putem masura timpul de executie al diverselor operatii (executie kernel, transfer date etc), cand acestea sunt blocante . Astfel obtinem timpi de executie al operatiilor, asa cum sunt percepute din perspectiva host/CPU. Aceasta metoda nu este foarte precisa deoarece in timpul de executie sunt incluse si toate operatiile de control CPU↔GPU.

Mai jos avem un exemplu de folosire a functiei cudaDeviceSynchronize pentru a forta o blocare pe partea de host/CPU pana cand toate operatiile pe partea de GPU au fost executate.

```
cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

t1 = myCPUTimer();
saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);
cudaDeviceSynchronize();
t2 = myCPUTimer();

cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
```

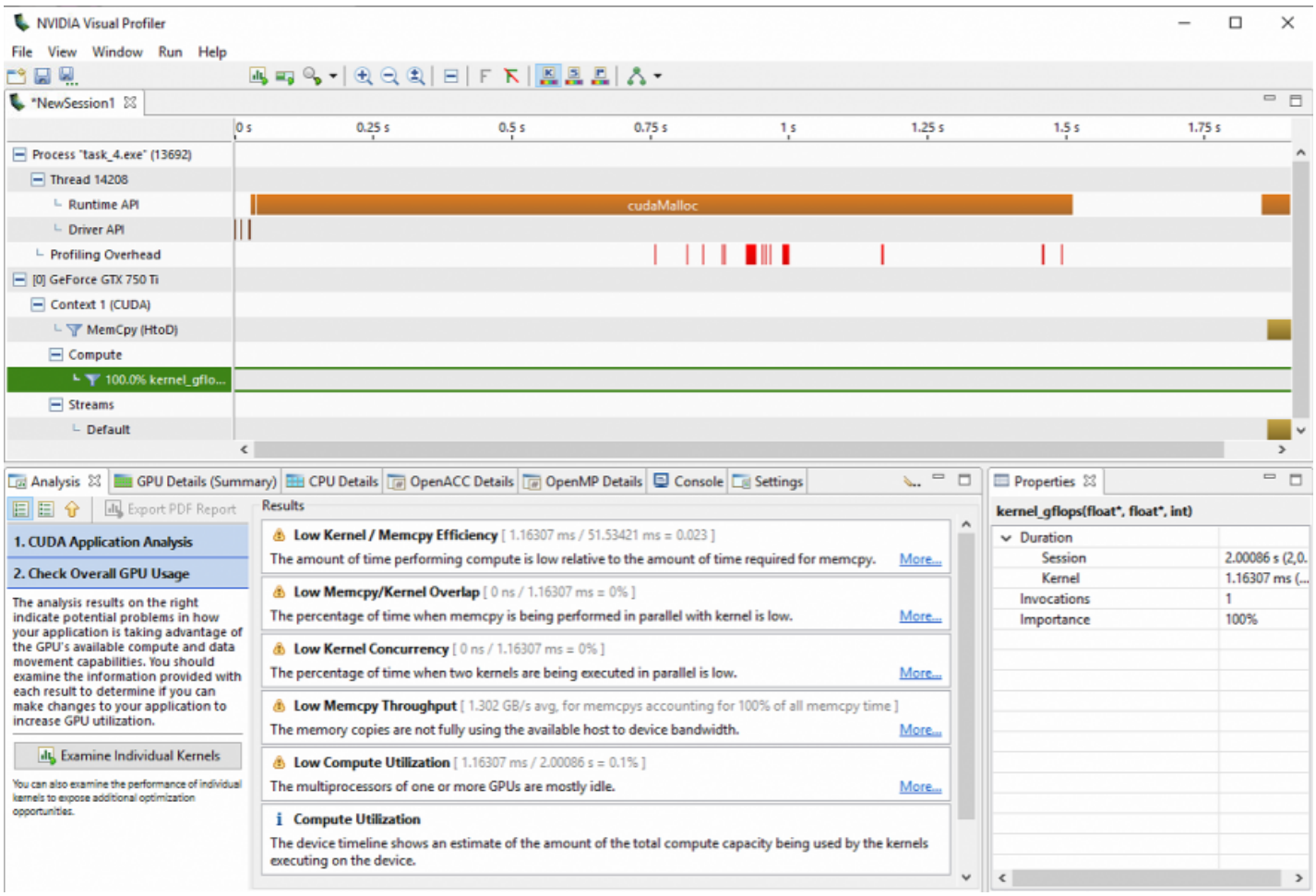
### Timing via CUDA events (device/GPU)

O varianta mai buna decat operatiile blocante sunt CUDA events. Acestea au suport hardware la GPU si ofera timpi de executie din perspectiva device/GPU. Mai jos avem un exemplu folosind CUDA events.

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

cudaEventRecord(start);
saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
cudaEventRecord(stop);

cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
```

### NVIDIA Visual Profiler

Aceasta este o aplicatie, parte a NVIDIA CUDA Toolkit, care poate sa ofera multe informatii cu privire la problemele de performanta dintr-o aplicatie CUDA. Se creaza o sesiune si se indica binarul impreuna cu argumente la care se vrea profiling. Mai jos este un sample al executiei aplicatiei task_4 din laboratorul trecut cu NVIDIA Visual Profiler.
Dupa cum se poate vedea sunt oferite informatii detaliate asupra timpilor de executiei, ocuparii resurselor GPU cat si indicatii cum se poate imbunatatii performanta. Mai multe informatii despre Visual Profiler aici [https://docs.nvidia.com/cuda/profiler-users-guide/index.html].




## Exercitii

- logati-va pe `fep.grid.pub.ro` folosind contul de pe `cs.curs.pub.ro`
- executati comanda `wget http://cs.curs.pub.ro/wiki/asc/_media/asc:lab8:lab8_skl.tar.gz [http://cs.curs.pub.ro/wiki/asc/_media/asc:lab8:lab8_skl.tar.gz] -O lab8_skl.tar.gz`
- dezarhivati folosind comanda `tar -xzvf lab8_skl.tar.gz`
- executati comanda `qlogin -q hp-sl.q` sau `qlogin -q ibm-dp48.q` pentru a intra pe o statie specializata in calcul folosind GPU-uri
- incarcati modulii de Nvidia CUDA folosind comanda `module load libraries/cuda`

Executam programele CUDA folosind cozile de executie prin submit via qsub, de pe fep.grid.pub.ro

```
[@fep7-1 lab8_skl]$ qsub -cwd -q hp-sl.q -b y ./matrix_multiplication
[@fep7-1 lab8_skl]$ cat matrix_multiplication.o1047126
```

Debug aplicatii CUDA aici [http://cs.curs.pub.ro/wiki/asc/asc:lab8:index#debug_aplicatii_cuda]

Modificarile se vor face in task_gflops.cu si matrix_multiplication_skl.cu - urmariti indicatiile TODO din cod.

1. Deschideți fișierul task_gflops.cu și urmăriți instrucțiunile pentru a măsura performanța maximă a unitații GPU, înregistrând numărul de GFLOPS (2p)
2. Completati functia matrix_multiply_simple care va realiza inmultirea a 2 matrice primite ca parametru. (2p)
3. Completati functia matrix_multiply care va realiza o inmultire optimizata a 2 matrice, folosind Blocked Matrix Multiplication. Hint: Se va folosi directiva shared pentru a aloca memorie partajata intre thread-uri. Pentru sincronizarea thread-urilor se foloseste functia __syncthreads. (2p)
4. Masurati timpul petrecut in kernel pentru fiecare din solutiile implementate la ex1 si ex2. Hint: Folositi evenimente CUDA. (2p)
5. Realizati profiling pentru functiile implementate folosind tool-urile nvprof si nvvp.(2p)

## Resurse

Schelet Laborator 8

Solutie Laborator 8

Enunt Laborator 8

- Responsabili laborator: Andreea birhala, Roxana Balasoiu, Ovidiu Dancila, Mihai Volmer, Grigore Lupescu

## Referinte

- Documentatie CUDA:
  - CUDA C Programming [https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf]
  - CUDA NVCC compiler [https://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf]
  - CUDA Visual Profiler [https://docs.nvidia.com/cuda/profiler-users-guide/index.html]
  - CUDA 9.1 Toolkit [https://developer.download.nvidia.com/compute/cuda/9.1/Prod/docs/sidebar/CUDA_Toolkit_Release_Notes.pdf]
  - CUDA GPUs [https://developer.nvidia.com/cuda-gpus]
- Acceleratoare hp-sl.q (hpsl-wn01, hpsl-wn02, hpsl-wn03)
  - NVIDIA Tesla K40M [http://international.download.nvidia.com/tesla/pdf/tesla-k40-passive-board-spec.pdf]
  - NVIDIA Tesla [https://en.wikipedia.org/wiki/Nvidia_Tesla]
- Acceleratoare ibm-dp.q (dp-wn01, dp-wn02, dp-wn03)
  - NVIDIA Tesla C2070 [https://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf]
  - NVIDIA Tesla 2050/2070 [http://www.nvidia.com/docs/io/43395/nv_ds_tesla_c2050_c2070_apr10_final_lores.pdf]
  - NVIDIA CUDA Fermi/Tesla [https://cseweb.ucsd.edu/classes/fa12/cse141/pdf/09/GPU_Gahagan_FA12.pdf]
- Advanced CUDA
  - CUDA Streams [https://devblogs.nvidia.com/gpu-pro-tip-cuda-7-streams-simplify-concurrency/]
  - CUDA Dynamic Parallelism [https://devblogs.nvidia.com/introduction-cuda-dynamic-parallelism/]

asc/lab8/index.txt · Last modified: 2019/04/06 17:47 by grigore.lupescu