Arhitecturi de tip GPGPU

Intro

Procesorul grafic (GPU - graphics processing unit) reprezinta un circuit electronic specializat in crearea si manipularea imaginilor trimise catre o unitate de display (e.g. monitor). Termenul GPGPU (general purpose graphics processing unit) denota un procesor grafic cu o flexibilitate ridicata de programare, capabil de a rezolva si probleme generale. In executie, o arhitectura de tip GPU foloseste paradigma SIMD (single instruction multiple data, taxonomia Flynn), ceea ce presupune, schimb rapid de context intre thread-uri, planificarea in grupuri de thread-uri si orientare catre prelucrari masive de date. Procesorul grafic dispune si de un spatiu propriu de memorie (GPU dedicat → VRAM, GPU integrat → RAM).

Unitatile tip GPU sunt potrivite pentru paralelismul de date, intensiv computationale. Datorita faptului ca aceleasi instructiuni sunt executate pentru fiecare element, nu sunt necesare mecanisme complexe pentru controlul fluxului. Ierarhia de memorie este simplificata comparativ cu cea a unui procesor x86/ARM. Deoarece calculele sunt intensive computational, latenta accesului la memorie poate fi ascunsa prin paralelism (massive multithreading, SIMT sau Single Instruction Multiple Threads) in locul folosirii extensive a memoriei cache.

Nu orice algoritm paralel ruleaza optim pe o arhitectura GPGPU. De principiu probleme de tip SIMD sau MIMD se preteaza rularii pe GPU-uri.

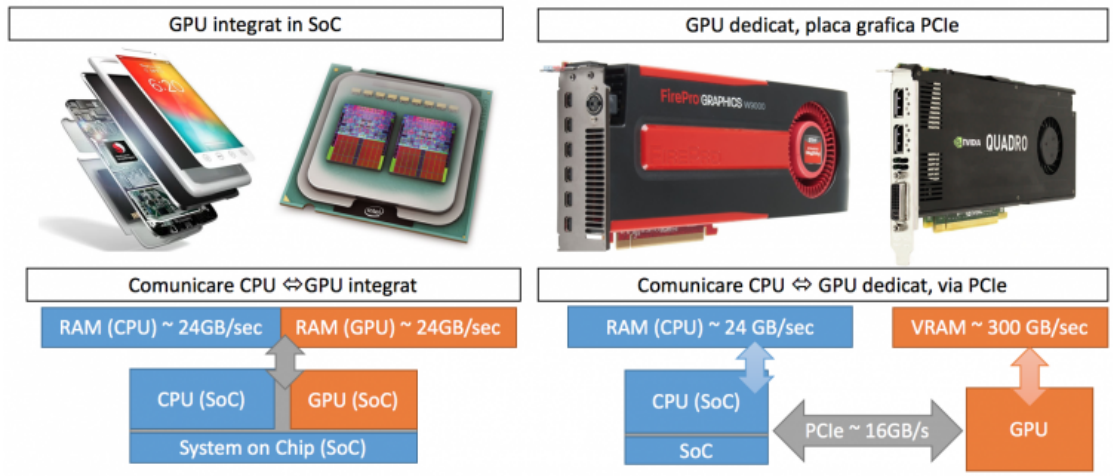
In multe cazuri, termenul de GPGPU apare atunci cand unitatea GPU este folosita ca si coprocesor matematic. In ziua de azi, majoritatea unitatilor de tip GPU sunt si GPGPU. In ultimii ani folosirea unitatilor GPGPU a luat amploare. Acest lucru se datoreaza:

- diferentelor de putere de procesare bruta dintre CPU si GPU in favoarea acestora din urma
- standardizarea de API-uri care usureaza munca programatorilor pentru a folosi GPU-ul
- raspandirea aplicatiilor ce pot beneficia de pe urma paralelismului tip SIMD
- regasirea unitatilor GPU atat in unitatile computationale consumer (PC, Smartphone, TV etc) cat si cele industriale (Automotive, HPC etc).

Principalii producatori de core-uri IP (intellectual property) tip GPU sunt:

- Intel http://en.wikipedia.org/wiki/List_of_Intel_graphics_processing_units [http://en.wikipedia.org/wiki/List_of_Intel_graphics_processing_units]
- Nvidia http://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units [http://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units]
- Amd http://en.wikipedia.org/wiki/List_of_AMD_graphics_processing_units [http://en.wikipedia.org/wiki/List_of_AMD_graphics_processing_units]
- Imagination http://en.wikipedia.org/wiki/List_of_PowerVR_products [http://en.wikipedia.org/wiki/List_of_PowerVR_products]
- Qualcomm http://en.wikipedia.org/wiki/Adreno [http://en.wikipedia.org/wiki/Adreno]
- Vivante http://en.wikipedia.org/wiki/Vivante_Corporation [http://en.wikipedia.org/wiki/Vivante_Corporation]

Daca un IP de GPU este integrat pe aceeasi pastila de siliciu a unui SoC (system on chip), acesta se numeste GPU integrat (integrated GPU). Exemple de SoC-uri cu IP de GPU integrat includ procesoarele x86 Intel si Amd cat si majoritatea SoC-urilor pentru dispozitive mobile bazate pe arhitectura ARM (ex. Qualcomm Snapdragon). Un GPU integrat imparte mare parte din ierarhia de memorie cu alte IP-uri (ex core-uri ARM/x86, controller PCIe/USB/SATA/ETH). Pe de alta parte un GPU dedicat (discrete GPU) presupunea integrarea IP-ului de GPU pe o placa cu memorie dedicata (VRAM) cat si o magistrala PCIe/AGP8x/USB pentru comunicare cu sistemul. Exemple de GPU-uri dedicate sunt seriile de placi grafice Geforce (Nvidia) si Radeon (Amd).



Aplicatii arhitecturi GPGPU

Exemple de domenii ce folosesc procesare GPGPU: prelucrari video si de imagini, simulari de fizica, finante, dinamica fluidelor, criptografie, design electronic (VLSI). Exemple de aplicatii pentru GPGPU:
Automotive - self driving cars (BMW, Continental etc)

- https://www.nvidia.com/en-us/self-driving-cars/partners/bmw/ [https://www.nvidia.com/en-us/self-driving-cars/partners/bmw/]
- https://blogs.nvidia.com/blog/2018/09/18/audi-unveils-e-tron-electric-suv/ [https://blogs.nvidia.com/blog/2018/09/18/audi-unveils-e-tron-electric-suv/]

Inteligenta artificala - antrenare retele neurale, inferenta

- https://www.forbes.com/sites/forbestechcouncil/2017/12/01/for-machine-learning-its-all-about-gpus/ [https://www.forbes.com/sites/forbestechcouncil/2017/12/01/for-machine-learning-its-all-about-gpus/]
- https://www.quora.com/Why-are-GPUs-well-suited-to-deep-learning [https://www.quora.com/Why-are-GPUs-well-suited-to-deep-learning]

Criptomonede - mining via hashing

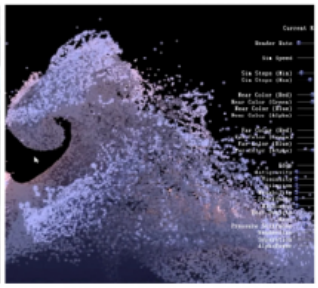
- https://coincentral.com/best-gpu-for-mining-2018-edition/ [https://coincentral.com/best-gpu-for-mining-2018-edition/]

SmartTV, Smartphone - accelerare video, recunoastere faciala/audio
Simulari fizice - NVIDIA Physx, Folding@Homel

- https://blogs.nvidia.com/blog/2018/11/13/weather-predicted-sc18-gpu-hpc-jensen-huang/ [https://blogs.nvidia.com/blog/2018/11/13/weather-predicted-sc18-gpu-hpc-jensen-huang/]

Prelucrari multimedia - filtre imagini GIMP/Photoshop
Alte domenii - arhivare (WinZip), encriptare





Programarea GPGPU

In cadrul unui sistem ce contine o unitate IP de tip GPU, procesorul general care coordoneaza executia este numit “HOST” (CPU) pe cand unitatea care efectueaza calculele este numita “DEVICE” (GPU). O unitate GPU contine un procesor de comanda (“command processor”) care citeste comenzile scrise de catre HOST (CPU) in anumite zone din RAM mapate spre access atat catre unitatea GPU cat si catre unitatea CPU. Toate schimbarile de stare in cadrul unui GPU, alocarile/transferurile de memorie si envenimentele ce tin de sistemul de operare sunt controlate de catre CPU (HOST).

In general, o prelucrare de date folosind unitatea GPU, necesita in prealabil un transfer din spatiul de memorie de la CPU catre spatiul de memorie de la GPU. In cazul unui procesor grafic dedicat acest transfer se face printr-o magistrala (PCIe, AGP, USB...). Viteza de transfer RAM-VRAM via magistrala este inferioara vitezei RAM sau VRAM. O potentiala optimizare in transferul RAM↔VRAM ar fi intercalarea cu procesarea. In cazul unui procesor integrate transferul RAM↔VRAM presupune o mapare de memorie, de multe ori translatata printr-o operatie de tip zero copy.

Programarea unui GPU se face printr-un API (Application Programming Interface). Cele mai cunoscute API-uri orientate catre folosirea unui GPU ca coprocesor matematic sunt: Cuda, OpenCL, DirectCompute, OpenACC, Vulkan. Dezvoltarea de cod pentru laboratoarele de GPU se va face folosind Cuda.

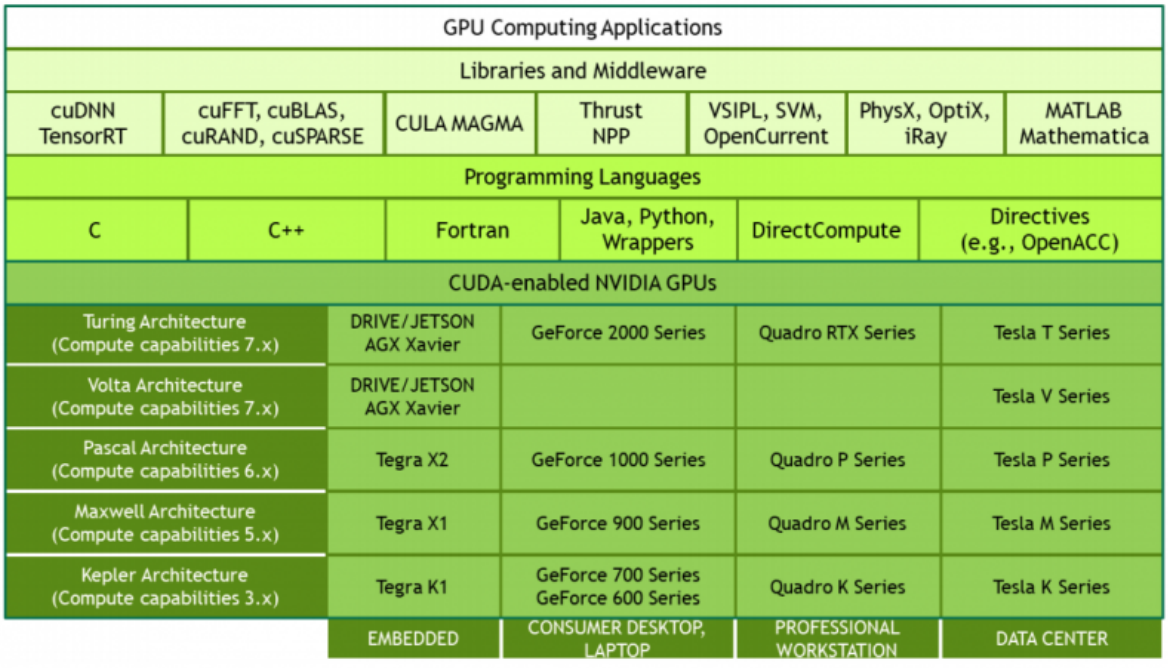
De ce CUDA ?

CUDA este un API introdus in 2006 de catre NVIDIA pentru GPU-urile sale. In prezent CUDA este standardul de facto pentru folosirea unitatilor GPU in industrie si cercetare. Aceasta se datoreaza faptului ca este o platforma stabila cu multe facilitati. O noua versiune de CUDA introduce noi functiontionalitati dar acestea uneori necesita versiuni recente ale arhitecturilor fiind dezactivate daca nu exista suport hardware. O versiune noua de CUDA extinde versiunea mai veche - de exemplu versiunea CUDA 9.0 reprezinta in mare o extensie/update asupra versiunii CUDA 8.0. In mare toate GPU-urile oferite de NVIDIA sunt suportate, diferenta fiind la facilitatile suportate. Singura limitare majora a platformei CUDA este ca suporta numai unitati de procesare de tip GPU de la NVIDIA.

Un standard alternativ la CUDA este OpenCL, suportata de Khronos ca standard si implementata de majoritatea producatorilor de GPU (inclusiv NVIDIA ca o extensie la CUDA). Problema majora la OpenCL este ca suportul este fragmentat si standardul este mult mai restrictiv decat CUDA si mai complicat de scris programe.

Arhitectura NVIDIA CUDA

Implementarea NVIDIA pentru GPGPU se numeste CUDA (Compute Unified Device Architecture) si permite utilizarea limbajului C pentru programarea pe GPU-urile proprii cat si extensii pentru alte limbaje (ex Python). Deoarece una din zonele tinta pentru CUDA este HPC (High Performance Computing), in care limbajul Fortran este foarte popular, PGI ofera un compilator de Fortran ce permite generarea de cod si pentru GPU-urile Nvidia. Exista binding-uri pana si pentru Java (jCuda), Python (PyCUDA) sau .NET (CUDA.NET). Framework-ul/arhitectura CUDA expune si API-ul de OpenCL prin intermediul caruia vom interactiona cu GPGPU-ul Nvidia Tesla disponibil pe ibm-dp.q.



Arhitectura CUDA (toate GPU-urile, seriile Geforce (consumer), Tesla (HPC), Jetson (automotive)).
Driver cu suport Windows, Linux, ce suporta atat CUDA API cat si OpenCL API.
Framework/toolkit compilator cu suport CUDA/OpenCL API (nvcc), debugger/profiler (CUDA API only)
Numeroase biblioteci si exemple CUDA/OpenCL API

Unitatea de baza in cadrul arhitecturii CUDA este numita SM (Streaming Multiprocessor). Ea contine in functie de generatie un numar variabil de Cuda Cores sau SP (Stream Processors) - de regula intre 8SP si 128SP. Unitatea de baza in scheduling este denumita “warp” si alcatuita din 32 de thread-uri. Vom aborda mai amanuntit arhitectura CUDA in laboratorul urmator. Ultima versiune de CUDA 8.0 suport OpenCL 1.2.

Compute capability

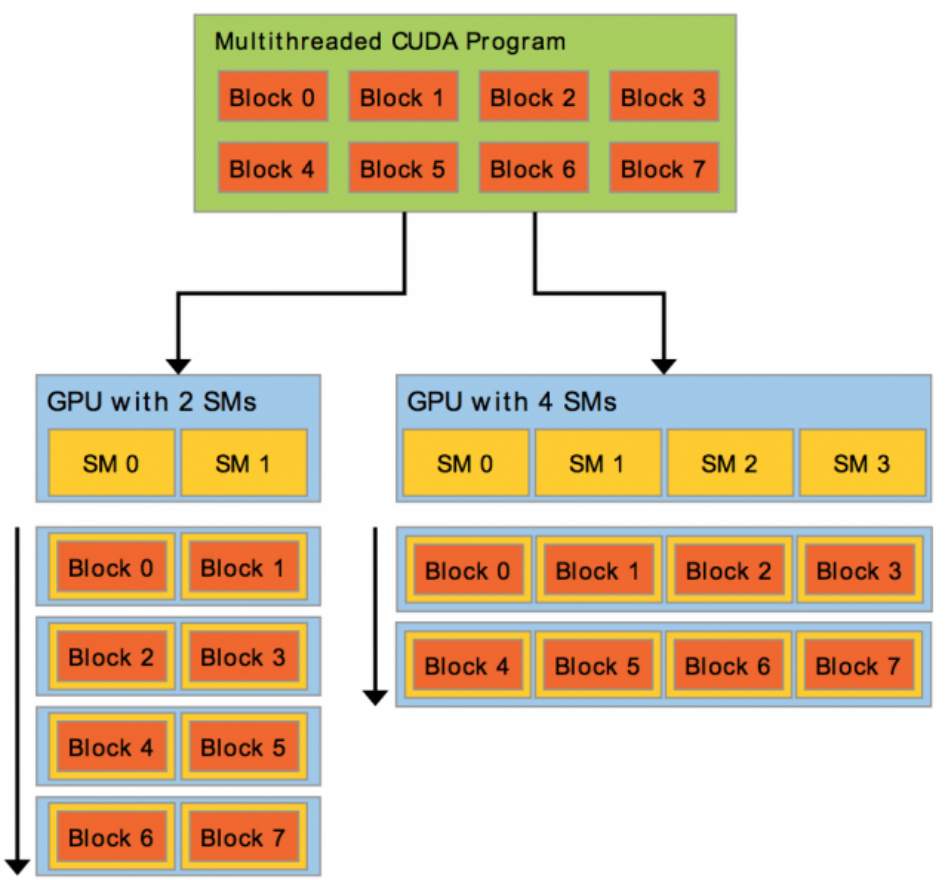
Versiunea de “compute capability” a unui SM (Streaming Multiprocessor), in cadrul arhitecturii CUDA, este reprezentat de un format X.Y, unde X este versiunea majora pe cand Y este versiunea minora. Partea majora identifica generatia din care face parte arhitectura. Astfel revizia 7 denota arhitectura Volta, 6 este pentru arhitectura Pascal, 5 pentru arhitectura Maxwell, 3 pentru arhitectura Kepler, 2 pentru Fermi iar 1 pentru Tesla. Partea minora identifica diferente incrementale in arhitectura si posibile noi functionalitati. Stiind versiunea majora si cea minora cunoastem facilitatile hardware oferite de catre arhitectura. GPU-urile care au aceasi versiune suporta aceleasi capabilitati.

O lista a GPU-urile NVIDIA si versiunile lor majore/minore se regaseste aici [https://developer.nvidia.com/cuda-gpus].
In cadrul cozii hp-sl.q se regasesc GPU-uri Tesla K40M [https://www.nvidia.com/content/pdf/kepler/tesla-k40-active-board-spec-bd-06949-001_v03.pdf], iar in cadrul cozii ibm-dp.q GPU-uri Tesla 2070 [https://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf].

Programarea in CUDA

CUDA extinde limbajul C prin faptul ca permite unui programator sa defineasca functii C, denumite kernels, care urmeaza a fi executе de N ori in paralel de N thread-uri CUDA. Scopul este de a abstractiza arhitectura GPU astfel incat partea de scheduling cat si gestiunea resurselor se face de catre stack-ul software CUDA impreuna cu suportul hardware. Figura de mai jos denota distribuirea thread-urilor catre 2 arhitecturi partitionate diferit.

Un kernel se defineste folosind specificatorul global iar executia sa se face printr-o configuratia de executie folosind <<<...>>> . Configuratia de executie denota numarul de blocks si numarul de thread-uri dintr-un block. Fiecare thread astfel poate fi identificat unic prin blockIdx si threadIdx.



Mai jos avem definit un kernel, vector_add, care are ca argumente pointers de tip float respectiv size_t. Acesta calculeaza f(x) = 2x + 1/(x + 1), pentru fiecare elemente din vector. Numarul total de thread-uri este dimensiunea vectorului.

```
__global__ void vector_add(const float *a, float *b, const size_t n) {
   // Compute the global element index this thread should process
   unsigned int i = threadIdx.x + blockDim.x * blockIdx.x;

   // Avoid accessing out of bounds elements
   if (i < n) {
      b[i] = 2.0 * a[i] + 1.0 / (a[i] + 1.0);
   }
}
```

Configuratia de executie denota maparea intre date si instructiuni. In functia de kernel, se defineste setul de instructiuni ce se va executa repetat pe date. Mai jos vector_add este lansat in executie cu N thread-uri (blocks_no x block_size) organizate cate block_size thread-uri per block.

```
        // Launch the kernel
        vector_add<<<blocks_no, block_size>>>(device_array_a, device_array_b, num_elements);
```

Aplicatie HelloWorld CUDA

```
#include <stdio.h>

__global__ void kernel_example(int value) {
   /**
   * This is a kernel; a kernel is a piece of code that
   * will be executed by each thread from each block in
   * the GPU device.
   */
   printf("[GPU] Hello from the GPU!\n");
   printf("[GPU] The value is %d\n", value);
}

int main(void) {
   /**
   * Here, we declare and/or initialize different values or we
   * can call different functions (as in every C/C++ program);
   * In our case, here we also initialize the buffers, copy
   * local data to the device buffers, etc (you'll see more about
   * this in the following exercises).
   */
   int nDevices;
   printf("[HOST] Hello from the host!\n");

   /**
   * Get the number of compute-capable devices. See more info
   * about this function in the Cuda Toolkit Documentation.
   */
   cudaGetDeviceCount(&nDevices);
   printf("[HOST] You have %d CUDA-capable GPU(s)\n", nDevices);

   /**
   * Launching the above kernel with a single block, each block
   * with a single thread. The syncrhonize and the checking functions
   * assures that everything works as expected.
```

```
    */
    kernel_example<<<1,1>>>(25);
    cudaDeviceSynchronize();

    /**
    * Here we can also deallocate the allocated memory for the device
    */
    return 0;
}
```

Aplicatie compute CUDA

O aplicatie CUDA are ca scop executia de cod pe GPU-uri NVIDIA CUDA.
In cadrul laboratoarelor partea de CPU (host) va fi folosita exclusiv pentru managementul executiei partii de GPU (device).
Aplicatiilor vor viza executia folosind un singur GPU NVIDIA CUDA.

0. Definire functie kernel

In codul prezentat mai jos, functia vector_add este marcata cu "global" si va fi compilata de catre CUDA NVCC compiler [https://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf] pentru GPU-ul de pe sistem (in cazul cozii hp-sl.q va fi NVIDIA Tesla K40M).

```
/**
 * This kernel computes the function f(x) = 2x + 1/(x + 1) for each
 * element in the given array.
 */
__global__ void vector_add(const float *a, float *b, const size_t n) {
        // Compute the global element index this thread should process
        unsigned int i = threadIdx.x + blockDim.x * blockIdx.x;

        // Avoid accessing out of bounds elements
        if (i < n) {
        b[i] = 2.0 * a[i] + 1.0 / (a[i] + 1.0);
        }
}
```

1. Definire zone de memorie host si device

Din punct de vedere hardware, partea de host (CPU) are ca memorie principala RAM (chip-uri memorie instalate pe placa de baza via slot-uri memorie) iar partea de device (GPU) are VRAM (chip-uri de memorie prezente pe placa video). Cand vorbim de memoria host (CPU) ne referim la RAM, iar in cazul memoriei device (GPU) la VRAM.

La versiunile mai recente de CUDA, folosind limbajul C/C++, un pointer face referire la spatiul virtual care este unificat pentru host (CPU) si device (GPU). Adresele virtuale insa sunt translatate catre adrese fizice ce rezida ori in memoria RAM (CPU) ori in memoria VRAM (GPU). Astfel este important cum alocam memoria (fie cu malloc pentru CPU sau cudaMalloc pentru GPU) si respectiv sa facem cu atentie transferurile de memorie intre zonele virtuale definite (de la CPU la GPU si respectiv de la GPU la CPU).

```
// Declare variable to represent ~1M float values and
// computes the amount of bytes necessary to store them
const int num_elements = 1 << 16;
const int num_bytes = num_elements * sizeof(float);

// Declaring the 'host arrays': a host array is the classical
// array (static or dynamically allocated) we worked before.
float *host_array_a = 0;
float *host_array_b = 0;

// Declaring the 'device array': this array is the equivalent
// of classical array from C, but specially designed for the GPU
// devices; we declare it in the same manner, but the allocation
// process is going to be different
float *device_array_a = 0;
float *device_array_b = 0;
```

2. Alocare memorie host (CPU)

Functia malloc va intoarce o adresa virtuala ce va avea corespondent o adresa fizica din RAM.

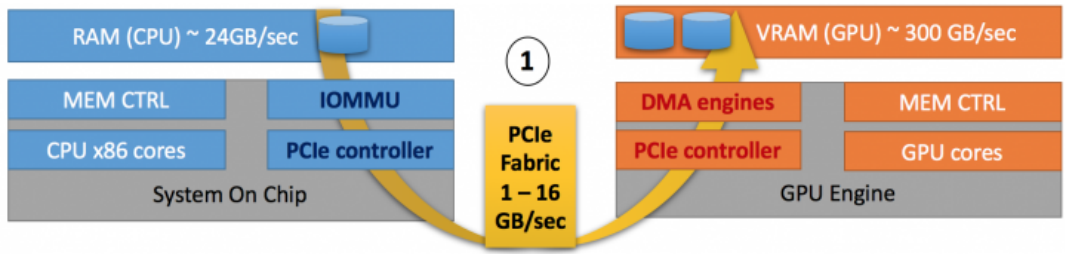
```
// Allocating the host array
host_array_a = (float *) malloc(num_bytes);
host_array_b = (float *) malloc(num_bytes);
```

3. Alocare memorie device (GPU)

Functia cudaMalloc va intoarce o adresa virtuala ce va avea corespondent o adresa fizica din VRAM.

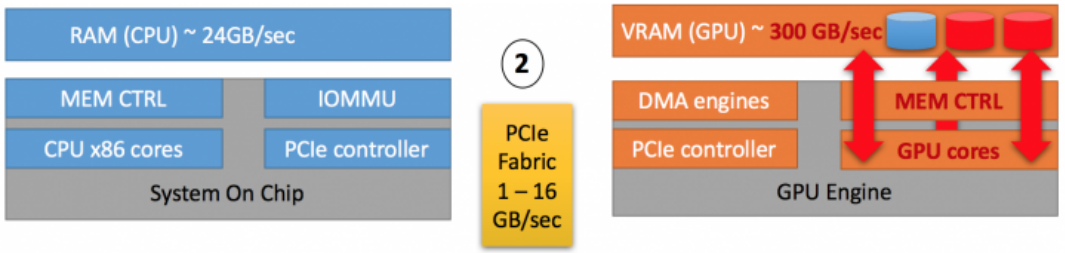
```
// Allocating the device's array; notice that we use a special
// function named cudaMalloc that takes the reference of the
// pointer declared above and the number of bytes.
cudaMalloc((void **) &device_array_a, num_bytes);
cudaMalloc((void **) &device_array_b, num_bytes);

// If any memory allocation failed, report an error message
if (host_array_a == 0 || host_array_b == 0|| device_array_a == 0 || device_array_b == 0) {
  printf("[HOST] Couldn't allocate memory\n");
  return 1;
}
```

4. Initializare memorie host (CPU) si copiere pe device (GPU)




```
// Initialize the host array by populating it with float values
for (int i = 0; i < num_elements; ++i) {
  host_array_a[i] = (float) i;
}

// Copying the host array to the device memory space; notice the
// parameters of the cudaMemcpy function; the function default
// signature is cudaMemcpy(dest, src, bytes, flag) where
// the flag specifies the transfer type.
//
// host -> device: cudaMemcpyHostToDevice
// device -> host: cudaMemcpyDeviceToHost
// device -> device: cudaMemcpyDeviceToDevice
cudaMemcpy(device_array_a, host_array_a, num_bytes, cudaMemcpyHostToDevice);
```

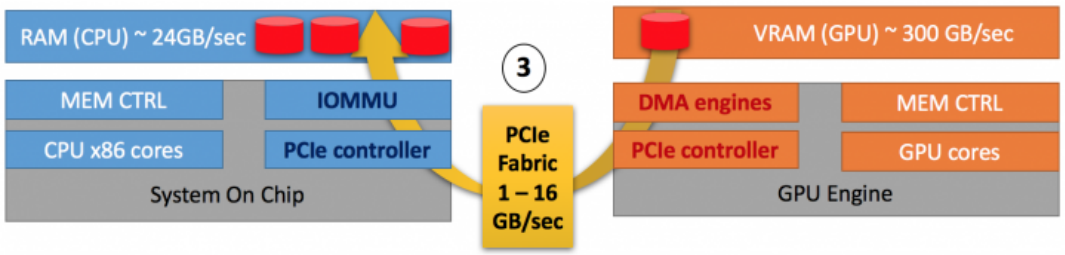
5. Executie kernel




```
// Compute the parameters necessary to run the kernel: the number
// of blocks and the number of threads per block; also, deal with
// a possible partial final block
const size_t block_size = 256;
size_t blocks_no = num_elements / block_size;

if (num_elements % block_size)
  ++blocks_no;

// Launch the kernel
vector_add<<<blocks_no, block_size>>>(device_array_a, device_array_b, num_elements);
```

6. Copiere date inapoi de la device (GPU) catre host (CPU)

```
// Copy the result back to the host memory space
cudaMemcpy(host_array_b, device_array_b, num_bytes, cudaMemcpyDeviceToHost);

// Print out the first 10 results
for (int i = 0; i < 10; ++i) {
  printf("Result %d: 2 * %1.1f + 1.0/(%1.1f + 1.0)= %1.3f\n",
  i, host_array_a[i], host_array_a[i], host_array_b[i]);
}
```

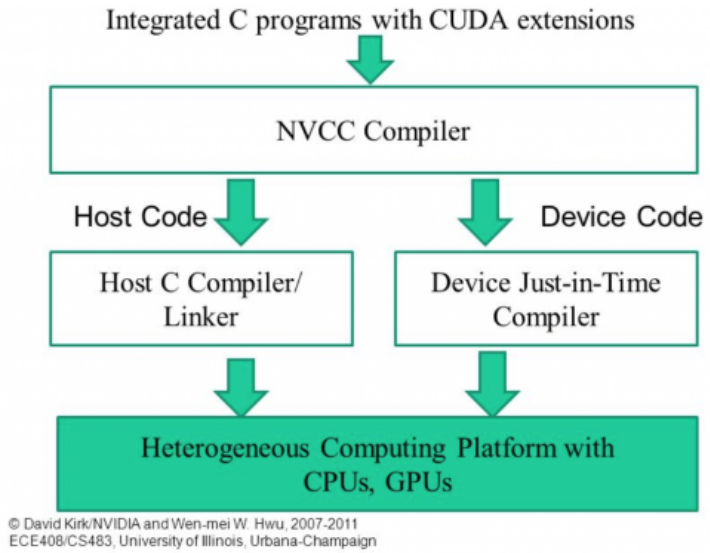


7. Cleanup

```
// Deallocate memory
free(host_array_a);
free(host_array_b);
cudaFree(device_array_a);
cudaFree(device_array_b);
```

Compilare si executie

Desi pentru un programator partile de host/CPU respectiv device/GPU pot fi in acelasi fisier *.cu, compilatorul CUDA (nvcc) le separa facand o compilare diferita pentru partea de host/CPU respectiv device/GPU. Figura de mai jos denota acest aspect.



Intrati pe frontend-ul `fep.grid.pub.ro` folosind contul de pe cs.curs.pub.ro. Executati comanda `qlogin -q ibm-dp.q` pentru a accesa una din statiile cu GPU-uri. Cozile ce au unitati GPU NVIDIA Tesla sunt ibm-dp.q, ibm-dp48.q si hp-sl.q.

```
[@fep7-1 ]$ qlogin -q hp-sl.q
[@dp-wn01]$ nvidia-smi
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 375.26                 Driver Version: 375.26                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla M2070         Off  | 0000:14:00.0     Off |                    0 |
| N/A   N/A    P0    N/A /  N/A |      0MiB /  5301MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla M2070         Off  | 0000:15:00.0     Off |                    0 |
| N/A   N/A    P0    N/A /  N/A |      0MiB /  5301MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
```

Modulele CUDA disponibile pe hpsl sunt: cuda-7.5, cuda-8.0, cuda-9.0, cuda-9.1.
Informatii despre modulul CUDA 9.1 aici [https://developer.download.nvidia.com/compute/cuda/9.1/Prod/docs/sidebar/CUDA_Toolkit_Release_Notes.pdf].

```
[@hpsl-wn03 ~]$ module avai

------------------------------------------------------------ /usr/share/Modules/modulefiles -------------------------------------------------
dot         module-git  module-info modules     null        use.own

------------------------------------------------------------ /etc/modulefiles ---------------------------------------------------------------
compilers/gnu-4.9.4            libraries/cuda         libraries/cuda-9.0                  libraries/openmpi-2.0.1-gcc-4.9.4       utilities/opencl
compilers/gnu-5.4.0            libraries/cuda-7.5     libraries/cuda-9.1                  libraries/openmpi-2.0.1-gcc-5.4.0
compilers/solarisstudio-12.5   libraries/cuda-8.0     libraries/opencv-3.1.0-gcc-4.9.4    utilities/intel_parallel_studio_xe_2016
```

Desi nu ar trebui sa existe diferente in functionalitate intre versiuni, recomandam folosirea CUDA 9.1 pentru laboratoare si teme.
Aceasta este si versiunea default atunci cand se executa “module load libraries/cuda”.

```
[@hpsl-wn03]$ module display libraries/cuda
-------------------------------------------------------------------
/etc/modulefiles/libraries/cuda:

prepend-path     PATH /usr/local/cuda-9.1/bin
prepend-path     LD_LIBRARY_PATH /usr/local/cuda-9.1/lib64
prepend-path     C_INCLUDE_PATH /usr/local/cuda-9.1/include
prepend-path     CPLUS_INCLUDE_PATH /usr/local/cuda-9.1/include
setenv           CUDA_PATH /usr/local/cuda-9.1
-------------------------------------------------------------------
```

Pentru a folosi implementarea CUDA de la Nvidia vom incarca modulul de CUDA 9.1. SDK-ul CUDA de la Nvidia include atat implementarea de CUDA API cat si cea de OpenCL API. In cadrul laboratoarelor vom programa numai folosind CUDA. Verificam mai jos ca scheletul laboratorului compileaza.

```
[@fep7-1 ~]$ qlogin -q hp-sl.q
[@hpsl-wn03 ~]$ module load libraries/cuda
[@hpsl-wn03 ~]$ wget -O lab7_skl.tar.gz http://cs.curs.pub.ro/wiki/asc/_media/asc:lab7:lab7_skl.tar.gz

--2019-01-12 16:02:13--  http://cs.curs.pub.ro/wiki/asc/_media/asc:lab7:lab7_skl.tar.gz
Resolving cs.curs.pub.ro (cs.curs.pub.ro)... 141.85.241.51, 2001:b30:800:f011:141:85:241:51
Connecting to cs.curs.pub.ro (cs.curs.pub.ro)|141.85.241.51|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 26754 (26K) [application/octet-stream]
Saving to: ‘asclab7.tar.gz’
100%[======================================================================>] 26,754      --.-K/s   in 0.003s

[@hpsl-wn03 ~]$ tar -xvzf lab7_skl.tar.gz
lab7_skl/
lab7_skl/example_1.cu
lab7_skl/example_2.cu
...
[@hpsl-wn03 ~]$ cd lab7_skl/
[@hpsl-wn03 lab7_skl]$
```

Prin incarcarea modului de CUDA sunt setate caile catre fisierele header cat si biblioteci.

```
COMPILER=nvcc
LIBS=-lm

example_1: example_1.cu
	$(COMPILER) $^ -o $@ $(LIBS)

example_2: example_2.cu
	$(COMPILER) $^ -o $@ $(LIBS)
....

clean:
	rm -rf example_1
	rm -rf example_2
...
	rm -rf *.o
```

```
[@hpsl-wn03 lab7_skl]$ make
nvcc example_1.cu -o example_1 -lm
[@hpsl-wn03 lab7_skl]$ make example_2
nvcc example_2.cu -o example_2 -lm
```

Pentru a executa un program CUDA pe cluster se va folosi qsub, rulat de pe fep.grid.pub.ro.

Modul corect de executia este folosind cozile de executie via qsub, de pe fep.grid.pub.ro

```
[@hpsl-wn03 lab7_skl]$ qsub -cwd -q hp-sl.q -b y ./example_1
Unable to run job: denied: host "hpsl-wn03.grid.pub.ro" is no submit host.
Exiting.
[@hpsl-wn03 lab7_skl]$ exit
logout
Connection to hpsl-wn03.grid.pub.ro closed.
/nfs/batch.grid.pub.ro/export/NCIT-Cluster/common/qlogin_ssh.sh exited with exit code 1
[@fep7-1 ~]$ cd lab7_skl/
[@fep7-1 lab7_skl]$ qsub -cwd -q hp-sl.q -b y ./example_1
Your job 1047126 ("example_1") has been submitted
[@fep7-1 lab7_skl]$ cat example_1.e1047126
[@fep7-1 lab7_skl]$ cat example_1.o1047126
[HOST] Hello from the host!
[HOST] You have 2 CUDA-capable GPU(s)
[GPU] Hello from the GPU!
[GPU] The value is 25
```

Nu se vor executa direct aplicatiile CUDA ci folosind qsub de pe fep catre o coada (hp-sl.q sau ibm-dp.q).

```
[@hpsl-wn03 lab7_skl]$ ./example_1
[HOST] Hello from the host!
[HOST] You have 2 CUDA-capable GPU(s)
[GPU] Hello from the GPU!
[GPU] The value is 25
```

Exercitii

Pentru inceput:
- 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:lab7:lab7_skl.tar.gz [http://cs.curs.pub.ro/wiki/asc/_media/asc:lab7:lab7_skl.tar.gz] -O lab7_skl.tar.gz`
- dezarhivati folosind comanda `tar -xzvf lab7_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 modulului 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 lab7_skl]$ qsub -cwd -q hp-sl.q -b y ./task_1
[@fep7-1 lab7_skl]$ cat task_1.o1047126
```

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

Modificarile se vor face in task_1.cu, task_2.cu si task_3.cu. Urmariti indicatiile TODO din cod. De asemenea, va recomandam sa folositi documentatia oficiala CUDA Toolkit Documentation de la adresa: https://docs.nvidia.com/cuda/ [https://docs.nvidia.com/cuda/]. Aici veti gasi informatii despre majoritatea functiilor de care aveti nevoie (folositi functia search).
Urmăriți instrucțiunile TODO din task_1.cu, task_2.cu si task_3.cu

1. Deschideți fișierul task_1.cu:
    I. Listați informații despre device-urile existente și selectați primul device (1p)
    II. Completați și rulați kernelul kernel_parity_id (1p)
    III. Completați și rulați kernelul kernel_block_id; explicați rezultatul (2p)
    IV. Completați și rulați kernelul kernel_thread_id; explicați rezultatul (2p)
2. Deschideți fișierul task_2.cu și urmăriți instrucțiunile pentru a realiza adunarea a doi vectori (2p)
3. Deschideți fișierul task_3.cu și urmăriți instrucțiunile pentru a realiza interschimbarea a doi vectori (2p)

Resurse

Schelet Laborator 7

Solutie Laborator 7

Enunt Laborator 7

- Responsabili laborator: Tudor Paraschivescu, George-Eduard Zaharia, Bianca Mihaela Cauc, Adrian Pop, 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/lab7/index.txt · Last modified: 2019/04/06 17:47 by grigore.lupescu