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 mult simplificata comparativ cu cea a unui core de procesor x86/ARM. Deoarece calculele sunt intensive computational, latenta accesului la memorie poate fi ascunsa prin calcule in locul unor cache-uri mari pentru date.

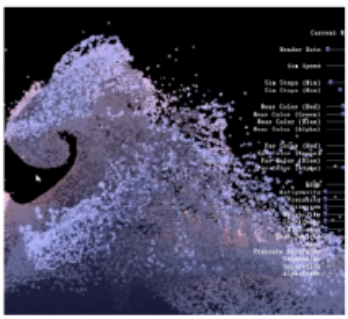
Nu orice algoritm paralel ruleaza optim pe o arhitectura GPGPU.

In cele mai multe din 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).

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)
SmartTV, Smartphone - accelerare video, recunoastere faciala/audio
Simulari fizice - NVIDIA Physx, Folding@Homel
Prelucrari multimedia - filtre imagini GIMP/Photoshop
Alte domenii - arhivare (WinZip), encriptare

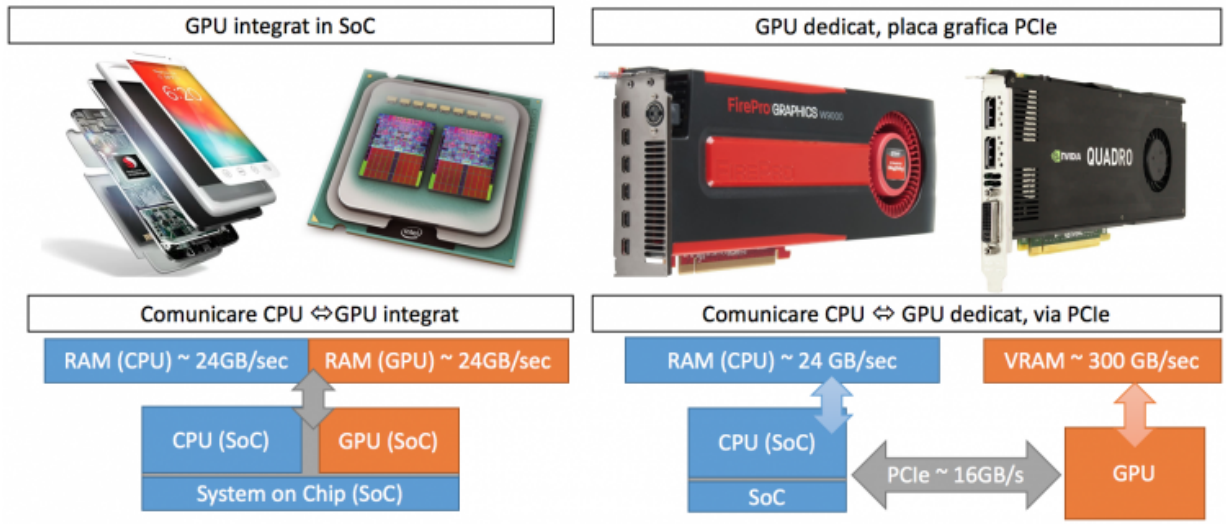




Principalii producatori de core-uri IP (intellectual property) tip GPU sunt:

- 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).



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 scrie 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 GPU se va face folosind OpenCL, mai exact versiunea 1.2.

De ce OpenCL ?

OpenCL este un API introdus in 2008, dezvoltat si mentinut de catre grupul Khronos (Apple, Intel, Amd, Nvidia, etc). Majoritatea companiilor ce dezvolta IP-uri de tip GPU, implementeaza OpenCL ca API in stack-ul lor. Pe baza evolutiei arhitecturilor si a cererilor business, exista numeroase versiuni de OpenCL. O noua versiune de OpenCL introduce noi functiontionalitati dar necesita de cele mai multe ori arhitecturi noi (sau modificari insemnate in stack). O versiune noua de OpenCL extinde versiunea mai veche - de exemplu versiunea OpenCL 2.0 reprezinta in mare o extensie asupra versiunii OpenCL 1.2.

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. Deoarece una din zonele tinta pentru CUDA este 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.

Arhitectura AMD GCN

In ultimii ani AMD s-a concentrat in partea de GPGPU pe framework-ul OpenCL cat si sa initieze modele noi de programare hibrida CPU-GPU. Procesoarele lor ce contin GPU sunt numite APU (Application Processing Unit) si in prezent ofera o integrare stransa intre CPU si GPU. Deoarece insa la nivel de industrie nu exista modele business care sa justifice integrarea stransa, suportul este mai mult experimental.

In prezent arhitectura AMD pentru GPGPU este GCN (Graphics Core Next). Aceasta presupune o impartire in clustere SIMD denumite CU (Compute Units). Fiecare CU are 4 unitati SIMD Vectoriale (fiecare SIMD poate procesa 16 operatii numere 32bit), 4 seturi VGPR 64KiB (Vector General Purpose Registers), 1 unitate scalara (ex branching), 4KiB GPR (General Purpose Registers). Deoarece un CU poate procesa simultan pana la 64 operatii, unitatea de baza in scheduling este de 64 thread-uri si se numeste “wavefront”.

Ultima versiunde de AMD APP SDK 3.0, suporta OpenCL 2.0. Sunt suportate in aceasta versiune atat unitatile GPU cat si unitatile CPU (prin multithread+SSE/AVX).

Arhitectura INTEL GEN

Incepand cu generatia de procesoare INTEL Ivy Bridge, majoritatea procesoarelor consumer (desktop, mobile) contin o unitate tip GPU. Acestea se disting in functie de generatie (GEN4, GEN5) cat si in functie de segment performanta (GT1, GT2, GT3). Unitatea de baza este denumite "subslice" si contine un numar de 8 unitati executie sau EU (Execution Units). Un slice contine 2 sau mai multe subslice-uri in functie de generatie. Nivelul tinta de performanta unde este incadrat GPU-ul va determina numarul de slice-uri. Astfel GT1 are 1 slice, GT2 are 2 slice-uri in timp ce GT3 are 4 slice-uri.

Ultima versiunde de INTEL OpenCL SDK (Beignet ptr Linux), suporta OpenCL 2.0. Versiunea OpenCL pentru CPU este diferita de versiunea de GPU (SDK-uri separate).

Introducere in OpenCL

Deoarece OpenCL este un API general ce permite mixarea de arhitecturi de la diferiti vendori acesta impune un model de programare mai complex. In OpenCL sunt definite atat functii de HOST (ex CPU) cat si un limbaj bazat pe C99 cu functii si tipuri specializate pentru DEVICE (ex GPU). Codul OpenCL (derivat C99) pentru DEVICE (GPU), este unul bazat pe o paradigma functionala. Functiile executabile de catre GPU sunt marcate corespunzator prin "kernel". In alte cuvinte un kernel in OpenCL este o functie ce urmeaza a fi executata pe DEVICE. In continuare vom analiza cum sa dezvoltam aplicatii in API-ul OpenCL, fara insa a intra in vreo arhitectura anume.

In OpenCL, codul pentru DEVICE este diferit de cel pentru HOST. Functiile pentru DEVICE au prefixul "kernel" si sunt compilate separat fata de codul pentru HOST.

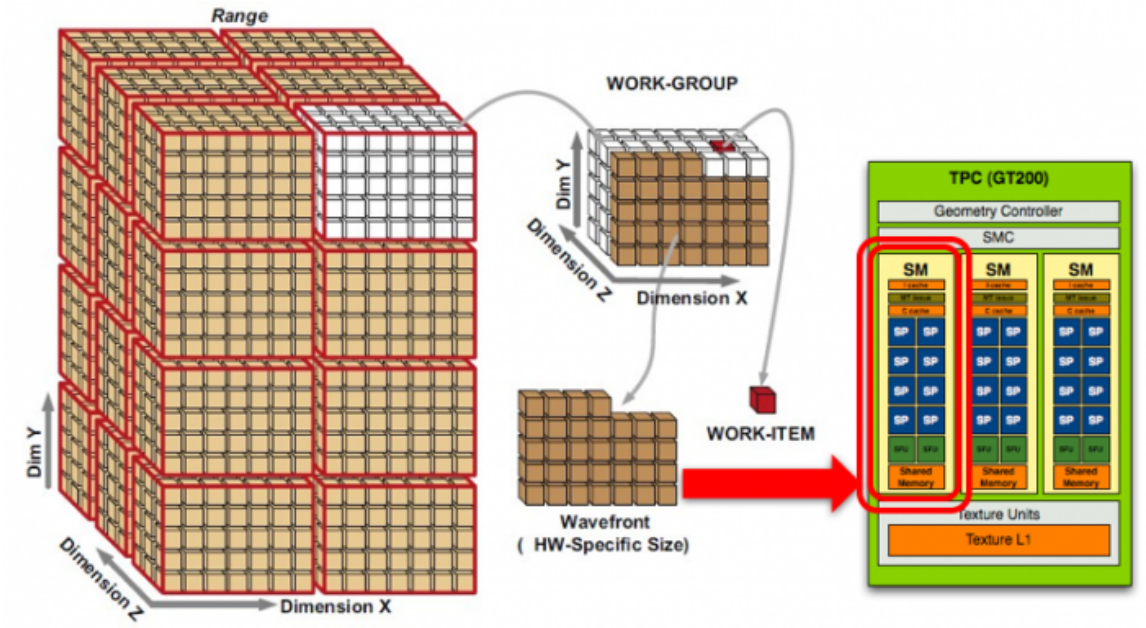
Mai jos avem definit un kernel, gpuProcess, care are ca argument un pointer de tip float marcat cu "global" (address space qualifier). Acesta denota faptul ca memoria este cea de GPU. Alti marcatori denota zone speciale din ierarhia de memorie precum "local" care denota memoria cache. Functia get_global_id este folosita pentru a returna un identificator unic in executia de tip NDRANGE folosita de OpenCL si discutata mai jos.

```
__kernel void gpuProcess(__global float* data)
{
        uint gid = get_global_id(0);
        data[gid] = 1.1f * data[gid];
}
```

Modelul de executie NDRANGE denota maparea intre date si instructiuni. In functia de kernel, se defineste setul de instructiuni ce se va executa repetat pe date. Functia responsabila cu maparea executiei intre instructiuni si date este clEnqueueNDRangeKernel.

```
size_t globalSize = 128;
clEnqueueNDRangeKernel(cmdQueue, kernel, 1, NULL, &globalSize, NULL, 0, NULL, NULL);
```

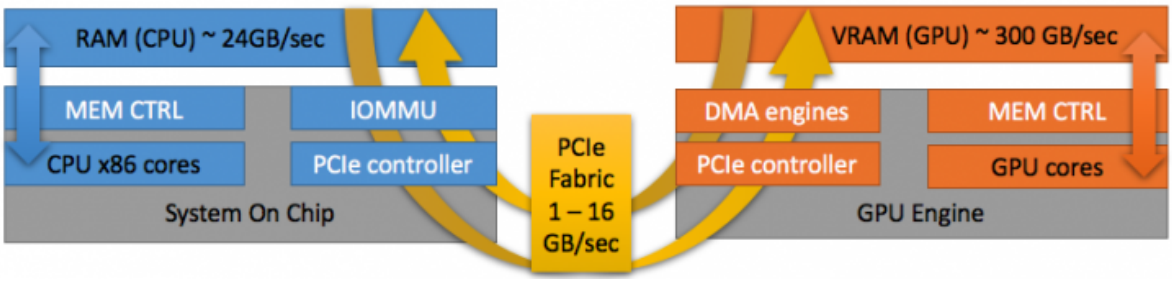
Dimensiunea globala a problemei denota numarul de instantieri ale unui kernel, identificate unic printr-un id intors de catre functia get_global_id. Fiecare instantiere unica, identificata prin get_global_id, reprezinta un work-item. Setul global de work-items reprezinta maparea problemei pe arhitectura (maparea SIMD). Un thread poate executa multiple instantieri ale unui kernel (work-items). Imaginea de mai jos prezinta configuratii NDRANGE de dimensiuni 1D, 2D respectiv 3D.



Un thread de GPU ruleaza in general 1 instantiere de kernel → 1 work-item.
Un set de work-items (AMD→wavefront, NVIDIA→warp) este in general rulat de 1 cluster de unitati (ex NVIDIA→TPC, AMD→CU).

Pentru a putea executa functia gpuProcess pe DEVICE (GPU), pe partea de HOST (CPU) efectuam urmatorii pasi:

- selectare unitatea de executie, PLATFORM/DEVICE, interogare mediu OpenCL
- initializare context si coada executie comenzi GPU, pregatire mediu OpenCL / GPU
- compilare program kernel pentru GPU, selectare kernel - pregatire mediu OpenCL / GPU
- (1) alocare memorie pentru GPU, efectuare transferuri memorie DEVICE(GPU) ⇔ HOST(CPU)
- (2) executia de kernel prin modelul NDRANGE pe DEVICE (GPU)
- (3) efectuare transferuri memorie DEVICE(GPU) ⇔ HOST(CPU)
- eliberare resurse / deinitializare mediu OpenCL



Aplicatie simpla OpenCL API

0. Kernel de OpenCL (C99 derivat), salvat in src_kernel, pentru a fi ulterior compilat si executat.

```
/* OpenCL kernel DEVICE (GPU) */
const char *src_kernel =  "" \
"__kernel void gpu_kernel(__global float *bufDevice){\n" \
"  uint gid = get_global_id(0);"          \
"  bufDevice[gid] = 1.1f * (float)gid;" \
"} \n";
```

1. Selectare unitatea de executie, DEVICE, printr-o interogare initiala in sistem.
Detalii API OpenCL clGetDeviceIDs [https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clGetDeviceIDs.html]

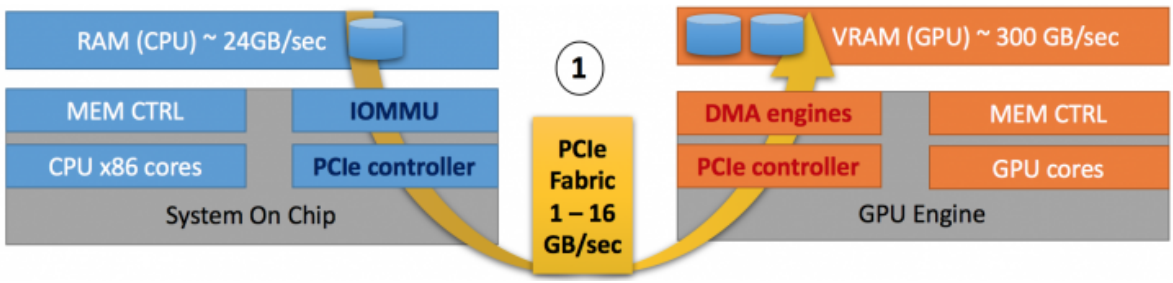
```
#include <stdio.h>
#include <stdlib.h>
#include <CL/cl.h>

/**
 * OpenCL HOST (CPU)
 */
int main(int argc, char** argv)
{
  cl_int ret;
  cl_platform_id platform;
  cl_device_id device;
  cl_context context;
  cl_command_queue cmdQueue;
  cl_program program;
  cl_kernel kernel;

  /* selecteaza prima platforma si primul device de tip GPU */
  clGetPlatformIDs(1, &platform, NULL);
  clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
```

2. Initializare context si coada de executie, unde vor fi trimise comenzile catre DEVICE (GPU).
Detalii API OpenCL clCreateCommandQueue [https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clCreateCommandQueue.html]

```
  context = clCreateContext(0, 1, &device, NULL, NULL, &ret);
  cmdQueue = clCreateCommandQueue(context, device, 0, &ret);
```

3. Alocare memorie pentru DEVICE (GPU), efectuare transferuri memorie HOST(CPU) ⇒ DEVICE(GPU). In cazul de fata nu se face transfer, doar se aloca memorie pentru CPU respectiv pentru GPU. Pentru transfer se poate folosi clEnqueueWriteBuffer.
Detalii API OpenCL clCreateBuffer [https://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/clCreateBuffer.html] clEnqueueWriteBuffer [https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clEnqueueWriteBuffer.html]

```
  /* aloca 128 elemente de tip float in memoria HOST (CPU/RAM) si apoi in memoria DEVICE (GPU/VRAM) */
  float *bufHost = (float*) malloc(sizeof(float) * 128);
  cl_mem bufDevice = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float) * 128, NULL, &ret);
```

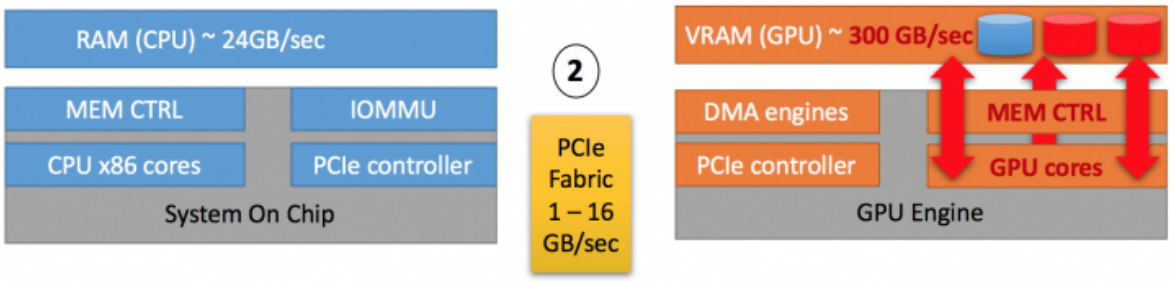


4. Compilare cod de program pentru DEVICE (GPU), selectare kernel.
Detalii API OpenCL clBuildProgram [https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clBuildProgram.html]

```
  /* creaza si compileaza programul OpenCL, selecteaza codul kernel */
  program = clCreateProgramWithSource(context, 1, (const char **)&src_kernel, NULL, &ret);
  clBuildProgram(program, 1, &device, NULL, NULL, NULL);
  kernel = clCreateKernel(program, "gpu_kernel", &ret);
```

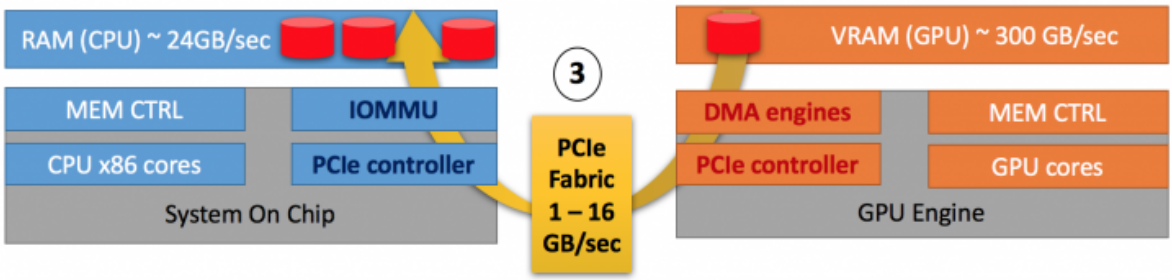
5. Setare argumente kernel si executia prin modelul NDRANGE. Modelul de executie NDRANGE denota o grupare a problemei pe arhitectura.
Detalii API OpenCL clEnqueueNDRangeKernel [https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clEnqueueNDRangeKernel.html]

```
  /* seteaza argumentele ce urmeaza a fi pasate la executia codului de kernel */
  clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&bufDevice);

  /* executa codul de kernel prin modelul NDRANGE */
  size_t globalSize = 128;
  clEnqueueNDRangeKernel(cmdQueue, kernel, 1, NULL, &globalSize, NULL, 0, NULL, NULL);
```



6. Efectuare transferuri memorie DEVICE(GPU) ⇒ HOST(CPU), deinitializare
Detalii API OpenCL clEnqueueReadBuffer [https://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/clEnqueueReadBuffer.html]

```
  /* copiaza in memoria RAM (HOST, CPU) datele generate in memoria VRAM (generate de catre DEVICE, GPU) */
  clEnqueueReadBuffer(cmdQueue, bufDevice, CL_TRUE, 0, sizeof(float) * 128, bufHost, 0, NULL, NULL);

  /* afiseaza datele generate */
  for(int i = 0; i < 128; i++)
          printf("%.2f\t", bufHost[i]);

  clFinish(cmdQueue);
  clReleaseMemObject(bufDevice);
  clReleaseCommandQueue(cmdQueue);
  clReleaseContext(context);
  free(bufHost);
}
```



Compilare si executie

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 cu OpenCL (CPU, GPU) sunt ibm-dp.q, ibm-dp48.q si hp-sl.q.

Atat platforma Nvidia cat si cea Intel ofera diverse utilitare pentru procesoarele (CPU, GPU) din sistem.
Spre exemplu cu “nvidia-smi” putem interoga ce device-uri NVIDIA TESLA GPU avem disponibile. Pe ibm-dp.q se gasesc 2 platforme Platform0:NVIDIA cu suport OpenCL 1.1 si 2 DEVICE-uri GPGPU-uri Nvidia Tesla M2070 si Platform1:INTEL cu 1 DEVICE tip CPU, Intel Xeon X5650.

```
[@fep7-1 ]$ qlogin -q ibm-dp.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 |
+-------------------------------+----------------------+----------------------+
```

Pentru a folosi implementarea OpenCL de la Nvidia vom incarca modulul de CUDA. SDK-ul CUDA de la Nvidia include atat implementarea de CUDA API cat si cea de OpenCL API. Alternativ putem folosi si platforma OpenCL de la Intel.
Versiunea suportata in platforma NVIDIA este OpenCL 1.2 insa Tesla M2070 suporta doar OpenCL 1.1. Versiunea de platforma denota functionalitatea pe partea de HOST pe cand pe partea de runtime (executie efectiva kernel) fiecare DEVICE are versiunea sa.

Astfel putem compila folosind SDK-ul OPENCL INTEL, platforma ce suporta OpenCL 2.0, si da spre executie unui DEVICE NVIDIA ce suporta doar OpenCL 1.1. In acest caz ambele runtime-uri (INTEL si NVIDIA) sunt instalate si chiar daca binarul a fost compilat cu SDK-ul de la INTEL, atunci cand se executa kernelul pe NVIDIA Tesla se transfera controlul la runtime-ul de la NVIDIA.

```
[@fep7-1 ]$ qlogin -q ibm-dp.q
[@dp-wn01]$ module load libraries/cuda-8.0
[@dp-wn01]$ module load utilities/opencl
[@dp-wn01]$ cd lab10_skl/ && make
```

Prin incarcarea modulului de CUDA sunt setate caile catre fisierele header cat si biblioteci. Altfel, ar fi trebuit sa specificam manual caile, prin -I (headers), respectiv -L (biblioteci). Tot ce ramane la compilare este sa specificam ca facem link cu biblioteca OpenCL.so.

```
cl_sample: cl_sample.cpp
        g++ cl_sample.cpp -lOpenCL -o cl_sample
```

Pentru a executa un program OpenCL pe cluster se va folosi qsub, rulat de pe fep.grid.pub.ro.
Compilare ⇒ ibm-dp.q, ibm-dp48.q, hp-sl.q

Executie ⇒ fep.grid.pub.ro

```
[@fep7-1]$ cat script_cl.sh
./cl_sample -args
[@fep7-1]$ qsub -cwd -q ibm-dp.q script_cl.sh

SAU
[@fep7-1]$ qsub -q ibm-dp.q -cwd -b y ./cl_sample
```

La final verificam rezultatul executiei.

```
[@fep7-1]$ ls
lab10_bin  lab10_bin.e908569  lab10_bin.o908569  skl_device.cl
```

Exercitii

Urmariti indicatiile todo si documentatia oficiala OpenCL pentru a rezolva exercitiile.
OpenCL refcard 1.1 [https://www.khronos.org/files/opencl-1-1-quick-reference-card.pdf] prezinta o listare a tuturor functiilor OpenCL 1.1 - atat host cat si device. - urmariti indicatiile TODO din cod.

Intrati pe frontend-ul `fep.grid.pub.ro` folosind contul de pe cs.curs.pub.ro. Executati comanda `qlogin -q ibm-dp48.q` pentru a accesa una din statiile cu GPU-uri. Modificarile se vor face in host.cpp (ex 1,2,3,4) cat si in device.cl (ex 5).

1. (2p) Completati functia gpu_find - selectati prin OpenCL un device de tip GPU, apoi listati denumirea si versiunea sa de OpenCL.
2. (2p) Completati functia gpu_swap_buffers - interschimbati 2 zone memorie CPU/HOST de dimensiune BUF_32M folosind un al 3-lea buffer de dimensiune BUF_2M (prin OpenCL GPU NV Tesla).
3. (2p) Completati functia gpu_execute_kernel - executati functia kernel “kernel_id” cat sa scrieti intreaga zona de memorie de BUF_128 element.
4. (2p) Completati functia kernel_id - zonele de memorie buf_dev/buf_host vor contine urmatoarele (a) sir repetitiv elemente “0 1 0 1” (b) primele 16 elemente vor fi 1 restul 0.
5. (2p) Completati functia kernel_gflops - masurati performanta maxima a unitatii GPU (se prefera operatiile +,* sau mad). Valorile maxime teoretice pentru Tesla se gasesc aici [https://en.wikipedia.org/wiki/Nvidia_Tesla] (Tesla M2070 ~ 1030 gflops FMAD).
6. (Bonus 1p) Descarcati si compilati sursele la aplicatia clinfo (link la referinte). Listati intreg setul de proprietati la device-ul NVIDIA Tesla. Selectati cele mai importante caracteristici, argumentati.

Resurse

Schelet Laborator 10

Solutie Laborator 10

Enunt Laborator 10

- Responsabil laborator: Grigore Lupescu

Referinte

- Acceleratoare OpenCL ibm-dp.q (dp-wn01, dp-wn02, dp-wn03)

    - OpenCL ibm-dp.q, ibm-dp48.q
- Acceleratoare OpenCL hp-sl.q (hpsl-wn01, hpsl-wn02, hpsl-wn03)
    - OpenCL hp-sl.q
- Documentatie OpenCL:
    - Khronos OpenCL 1.1 [https://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/]
    - OpenCL 1.1 reference card [https://www.khronos.org/files/opencl-1-1-quick-reference-card.pdf]
    - Specificatie OpenCL 1.1 [https://www.khronos.org/registry/cl/specs/opencl-1.1.pdf]
- Arhitectura NVIDIA Fermi (fep.grid.pub.ro → Tesla 2050):
    - Nvidia Fermi Wikipedia [https://en.wikipedia.org/wiki/Fermi_(microarchitecture)]
    - Nvidia Fermi Architecture Whitepaper [http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.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]
    - Nvidia Tesla [https://en.wikipedia.org/wiki/Nvidia_Tesla]
- Arhitectura AMD GCN:
    - GCN Wikipedia [https://en.wikipedia.org/wiki/Graphics_Core_Next]
    - GCN Architecture Whitepaper [https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf]
- Arhitectura INTEL GEN:
    - Intel GEN8 Architecture [http://https://software.intel.com/sites/default/files/Compute%20Architecture%20of%20Intel%20Processor%20Graphics%20Gen8.pdf]
- Alte:
    - Khronos site [https://www.khronos.org/registry/]
    - Aplicatie clinfo [https://github.com/Oblomov/clinfo]
    - OpenCL optiuni compilare [https://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/clBuildProgram.html]

asc/lab10/index.txt · Last modified: 2017/04/22 17:51 by grigore.lupescu