Arhitectura hardware a placilor grafice GPU-CUDA

Arhitectura GPU-CUDA – **G**raphical **P**rocessing **U**nit – **C**omputer **U**nified **D**evice **A**rchitecture (prezentata in figura 1) consta in *N* multiprocesoare (***Streaming Multiprocessors - SM***), fiecare avind *M* nuclee (***cores*** sau ***streaming processors*** sau ***procesoare***). Fiecare procesor are registre dedicate (proprii). Exista o memorie partajata ***(shared***) comuna tuturor procesoarelor din multiprocesor. Fiecare multiprocesor are doua memorii cache – ***constant cache*** si ***texture cache***. Toate multiprocesoarele partajeaza o memorie globala (DRAM).

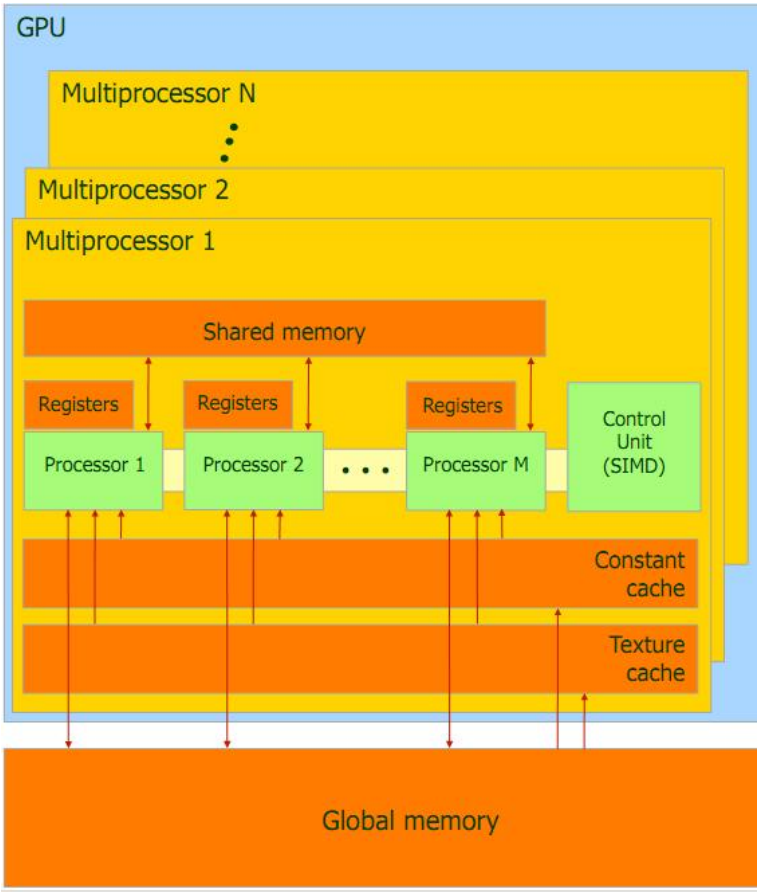

Figura 1. Arhitectura GPU-CUDA

Principalele caracteristici ale placilor GPU-CUDA sint:

- Paralelism masiv (Single Instruction Multiple Threads):
  - acelasi cod este executat in parallel de un numar mare de thread-uri.
  - thread-urile pot partaja date pe mai multe niveluri.
- Prelucrari heterogene (CPU-CPU):
  - GPU: prelucrari de date (simple) intensive.
  - CPU: controlul programului

Modelul de programare CUDA

Placa grafica (GPU sau device) reprezinta un co-procesor “multi threaded” al CPU (host) care poate executa un numar foarte mare de thread-uri identice in paralel.

Thread-urile CUDA sint simple (lightweight) ceea ce inseamna ca prelucrarile suplimentare necesare pentru crearea acestora si comutarea contextului (planificarea thread-urilor) consuma putin timp.

Scopul final al unui program CUDA este sa se declare cit mai multe astfel de thread-uri in asa fel incit sa se utilizeze cit mai complet resursele hardware disponibile.

Structura unui program CUDA

Definitii generale

Device = GPU= Set de multiprocesoare
Multiprocesor = SM = set de procesoare + memorie partajata (shared)
Bloc (bloc de thread-uri) = grup de threaduri (de tip SIMD) care executa un cod unic (kernel) pe un set de date identificate de threaded si blockID. Thread-urile pot comunica prin memoria partajata
Kernel = programul (codul) asociat fiecarui thread executat in GPU
Grid = grup (arie) de blocuri de thread-uri care executa kernelul
Warp = grup de thread-uri care pot fi planificate simultan pentru executie (dimensiunea warp = 32)

Fiecare SM proceseaza loturi de blocuri, in mod secvential.

Blocurile procesate la momentul current se numesc blocuri active. Thread-urile din blocurile active vor fi thread-uri active.

Registrele si memoria “shared” sint impartite de thread-urile active.

Modelul de programare hibrid CPU+GPU CUDA presupune definirea unui (sau unor) nuclee (kernels). Programul are 2 sectiuni – una seriala, care se executa in CPU si una paralela care se executa in fiecare dim procesoarele din GPU, in paralel. In codul sursa va exista o sectiune seriala si unul sau mai multe apeluri ale kernel-ului(elor).

Organizarea aplicatiei se va face ca in figura 2 in grila (grid), blocuri si thread-uri. O grila contine mai multe blocuri si un bloc are mai multe threaduri.

O procesare hibrida CPU – GPU are urmatoarele faze, conform figurii 3.

- aloca memorie in memoria CPU
- aloca memorie in memoria GPU
- copiaza datele de prelucrat din memoria CPU in memoria GPU
- lanseaza kernel-ul (in paralel in toate nucleele GPU)
- copiaza rezultatele din memoria GPU in memoria CPU
- elibereaza memoria CPU si GPU

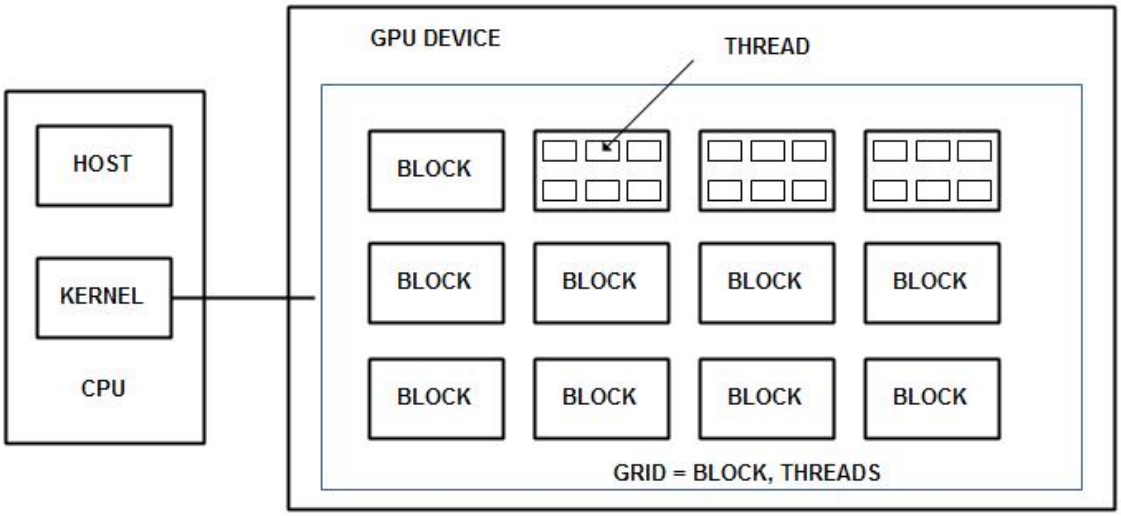

Figura 2. Organizarea thread-urilor in modelul de programare CUDA

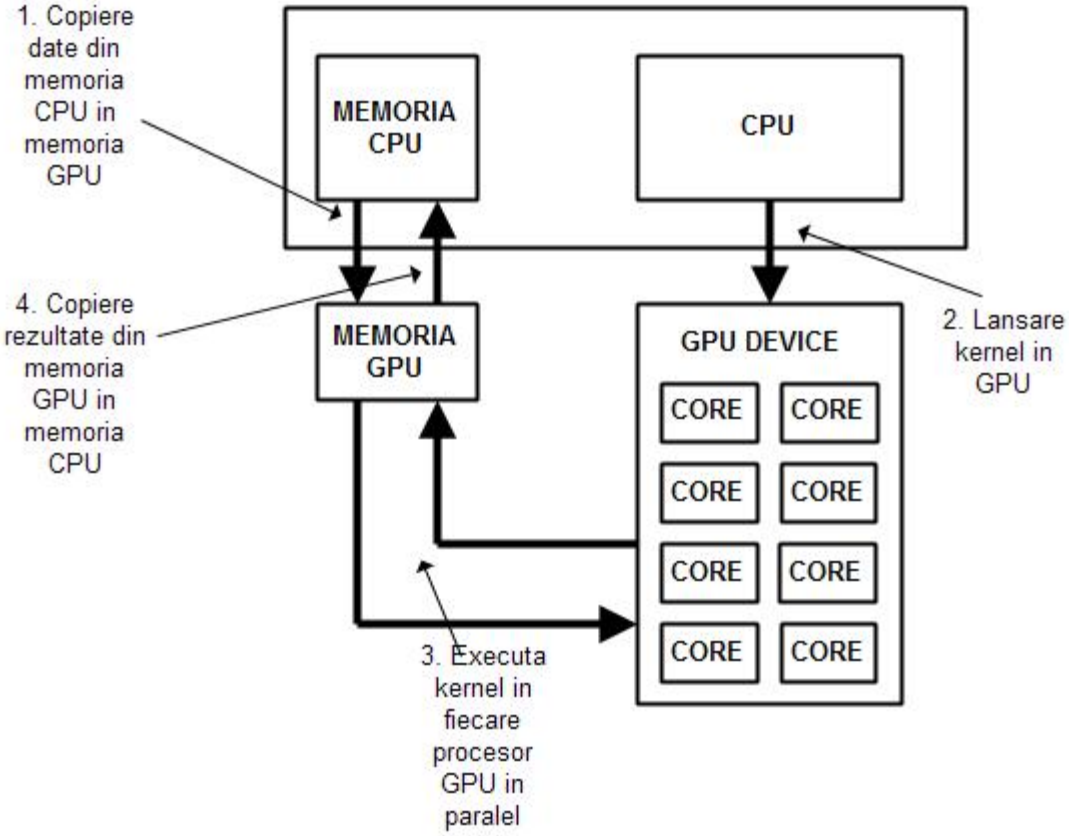

Figura 3. Fazele unei procesari hibride CPU-GPU

Exista functii dedicate pentru copierea din si in memoria GPU, dupa cum urmeaza:

`cudaMalloc` – aloca memorie in memoria GPU

cudaMemcpy – copiaza blocuri de memorie din memoria CPU in memoria GPU sau din memoria GPU in memoria CPU.

Un kernel se defineste prin:

```
__global__ void  NUME_Kernel(parametrii);
```

si se lanseaza (din codul rulat pe CPU) astfel:

```
NUME_Kernel <<< Numar_blocuri_grila, Numar_thread-uri_bloc >>>(
parametrii);
```

Se poate masura timpul de executie prin definirea unui timer:

```
StopWatchInterface *timer = NULL;

sdkCreateTimer(&timer);
sdkStartTimer(&timer);
```

si

```
sdkDeleteTimer(&timer);
```

Un exemplu de cod, care ilustreaza etapele de executie ale unui cod CUDA, este prezentat in anexa.

Exemplificarea utilizarii CUDA in efectuarea calculelor paralele

1. Se deschide in Visual Studio proiectul **vectorAdd_vs2008 care** contine ca exemple doua prelucrari de vectori (adunare si SAXPY).
2. Se studiaza codul **vectorAdd.cu** si se identifica fazele prelucrarii hibride (din figura 3).
3. Se compileaza si se executa codul, pentru cele doua operatii (modificindu-se numarul de elemente ale vectorilor si configuratia kernelului).
4. Se compara timpii de executie CPU si GPU pentru diferite configurari.

```
/*
Adunarea vectorilor: C = A + B
Operatia Vector Sum Alfa multiply X Plus Y (SAXPY) : Z = alfa*X+Y
*/

#include <stdio.h>

// pentru CUDA runtime routines (cu prefixul "cuda_")
#include <cuda_runtime.h>

// Utilitati si timere
#include <helper_functions.h>    // include cuda.h si cuda_runtime_api.h

// testarea erorilor CUDA
#include <helper_cuda.h>

// Functia de adunare a vectorilor: C = A + B
void vectorAdd_CPU(const float *A, const float *B, float *C, int numElements);
// Functia SAXPY : C = alfa*A + B
void saxpy_CPU(const float *A, const float *B, float *C, int numElements, float alfa);

// defineste o variabila timer
StopWatchInterface *timer = NULL;

// Codul din CUDA Kernel-uri

__global__ void  vectorAdd(const float *A, const float *B, float *C, int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < numElements)
    {
        C[i] = A[i] + B[i];
    }
}

__global__ void saxpy(const float *A, const float *B, float *C, int numElements, float alfa)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < numElements)
      {
          C[i] = alfa*A[i] + B[i];
      }
}

//Programul principal in host (CPU)

int main(void)
{
    // Initilaizare cod de eroare err (pentru verificarea valorilor intoarse de functiile CUDA)
    cudaError_t err = cudaSuccess;

    // Afiseaza kungimea vectorilor si calculeaza dimensiunea necesara a memoriei
    int numElements = 60000000;
    size_t size = numElements * sizeof(float);

    //printf("[Vector addition of %d elements]\n", numElements);
    printf("[Vector SAXPY of %d elements]\n", numElements);

    // Alocare memorie (CPU) pentru  vectorul  A
    float *h_A = (float *)malloc(size);

    // Alocare memorie (CPU) pentru  vectorul B
    float *h_B = (float *)malloc(size);

    // Alocare memorie (CPU) pentru  vectorul C
    float *h_C = (float *)malloc(size);

    // Verifica daca alocarea s-a facut
    if (h_A == NULL || h_B == NULL || h_C == NULL)
```

```
    {
        fprintf(stderr, "Failed to allocate host vectors!\n");
        exit(EXIT_FAILURE);
    }

    // Initializare vectori A, B si C in memoria CPU
    for (int i = 0; i < numElements; ++i)
    {
        h_A[i] = rand()/(float)RAND_MAX;
        h_B[i] = rand()/(float)RAND_MAX;
    }

    // Alocare memorie (GPU) pentru  vectorul A
    float *d_A = NULL;
    err = cudaMalloc((void **)&d_A, size);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to allocate device vector A (error code %s)!\n", cudaGetErrorString
    (err));
        exit(EXIT_FAILURE);
    }

    // Alocare memorie (GPU) pentru  vectorul B
    float *d_B = NULL;
    err = cudaMalloc((void **)&d_B, size);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to allocate device vector B (error code %s)!\n", cudaGetErrorString
    (err));
        exit(EXIT_FAILURE);
    }

    // Alocare memorie (GPU) pentru  vectorul C
    float *d_C = NULL;
    err = cudaMalloc((void **)&d_C, size);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to allocate device vector C (error code %s)!\n", cudaGetErrorString
    (err));
        exit(EXIT_FAILURE);
    }

    // Copiere din memoria CPU in memoria GPU (vectorii A si B)

    printf("Copy input data from the host memory to the CUDA device\n");
    err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to copy vector A from host to device (error code %s)!\n",
    cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    err = cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to copy vector B from host to device (error code %s)!\n",
    cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }
    //
    //executa codul in CPU si masoara timpul de executie
    sdkCreateTimer(&timer);
    sdkStartTimer(&timer);

    //vectorAdd_CPU(h_A, h_B, h_C, numElements);
```

```
    saxpy_CPU(h_A, h_B, h_C, numElements,2.0);

    sdkStopTimer(&timer);
    printf("CPU processing time: %f (ms)\n", sdkGetTimerValue(&timer));
    sdkDeleteTimer(&timer);
    //
    // Calculeaza numarul de blocuri
    int threadsPerBlock = 1024; // 256, 512, 1024
    int blocksPerGrid =(numElements + threadsPerBlock - 1) / threadsPerBlock;
    printf("CUDA kernel launch with %d blocks of %d threads\n", blocksPerGrid, threadsPerBlock);

    sdkCreateTimer(&timer);
    sdkStartTimer(&timer);

    // Lanseaza CUDA Kernel

    //vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);
    saxpy<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements,2.0);

    sdkStopTimer(&timer);
    printf("GPU processing time: %f (ms)\n", sdkGetTimerValue(&timer));
    sdkDeleteTimer(&timer);

    // Verifica daca au aparut erori la executia kernel-ului
    err = cudaGetLastError();

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to launch vectorAdd kernel (error code %s)!\n", cudaGetErrorString(err))↙
    ;
        exit(EXIT_FAILURE);
    }

    // Copiaza vectorul rezultat din memoria GPU in memori CPU
    printf("Copy output data from the CUDA device to the host memory\n");
    err = cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to copy vector C from device to host (error code %s)!\n",                ↙
    cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    // Verifica daca rezultatul e corect
    for (int i = 0; i < numElements; ++i)
    {
        //if (fabs(h_A[i] + h_B[i] - h_C[i]) > 1e-5)
        if (fabs(2.0*h_A[i] + h_B[i] - h_C[i]) > 1e-5)

        {
            fprintf(stderr, "Result verification failed at element %d!\n", i);
            exit(EXIT_FAILURE);
        }
    }

    // Elibereaza memori GPU
    err = cudaFree(d_A);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to free device vector A (error code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }
    err = cudaFree(d_B);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to free device vector B (error code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }
```

```
    err = cudaFree(d_C);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to free device vector C (error code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    // Elibereaza memoria CPU
    free(h_A);
    free(h_B);
    free(h_C);

    // Reset GPU
    err = cudaDeviceReset();

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to deinitialize the device! error=%s\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    printf("Done\n");
    return 0;
}

// Functiile CPU (host)

void vectorAdd_CPU(const float *A, const float *B, float *C, int numElements)
{
    int i;
    for (i=0; i<numElements; i++)
    {
        C[i] = A[i] + B[i];
    }
}

void saxpy_CPU(const float *A, const float *B, float *C, int numElements, float alfa)
{
  int i;
  for (i=0; i<numElements; i++)
      {
          C[i] = alfa*A[i] + B[i];
      }
}
```