Laborator 3

Sîrbu Matei-Dan

29 octombrie 2020

1 Texturi și imagini color

Domeniul „nativ" al aplicațiilor CUDA este procesarea imaginilor.[1] Acest domeniu depășește cadrul laboratorului; ne vom rezuma doar la câteva example cu rol introductiv. Scopul acestei lucrări este de a prezenta câteva exemple mai complexe de programe CUDA, exemplificând în mod special procesarea cu texturi și imagini.

1.1 Procesarea texturilor

Un termen împrumutat din grafica 3D – textura – poate fi considerată o matrice uni-, bi- sau tri-dimensională de *texeli* (texture-elements). Texelii pot fi reprezentați prin scalari (byte, float), sau 4-tuple (byte4, float4).

În CUDA, texturile se disting ca o zonă de memorie specială, care poate fi citită cu ajutorul unor funcții de acces speciale `tex1D(x)`, `tex2D(x, y)`, respectiv `tex3D(x, y, z)`. Texturile oferă următoarele facilități:

- pentru dispozitivele mai vechi, citirea din memoria de textură este mai rapidă decât accesul din memoria globală dispozitiv[1],
- se pot citi și elemente de la coordonate ne-întregi, interpolarea (liniară) a valorilor efectuându-se de către hardware (de ex: `float a = tex2D(1.5, 3.25)`),
- coordonatele care depășesc domeniul texturii $[0 \dots N-1]$ se ajustează automat, fie forțându-se la marginile domeniilor $0, N-1$ fie calculând modulo N, configurabil,

[1] Arhitectura CUDA 2.x (Fermi) oferă memorie cache și memoriei globale

Primul program CUDA

Scopul acestui exemplu este de a ne familiariza cu mediul CUDA, compilarea unui program, lucrul cu memoria dispozitiv, modelul de execuție multi-threaded divizat în blocuri.

În loc de tradiționalul *"Hello World"*, primul exemplu va consta din stocarea în memorie a șirului de întregi 0, 1, 2, ... N-1, echivalentul paralel al următorului cod C:

```
for(i=0; i<N; i++)
        a[i]=i;
```

Chiar și pentru acest banal exemplu, este nevoie de următorii pași:

1. alocarea memoriei pe dispozitiv și gazdă (pointerii a_cpu, respectiv a_gpu)
2. apelul CUDA: Lansarea a N threaduri. Fiecare thread are asociat un indice unic, intrinsec, de la 0 la $N-1$. Pseudo-codul este:

   ```
   Thread(i) executa:
       a[i]=i;
   terminare
   ```

3. așteptarea terminării tuturor threadurilor
4. copierea rezultatului din memoria dispizitiv în memoria gazdă[2]
5. afișarea șirului rezultat.

În pașii 2-3, execuția se desfășoară pe procesorul grafic, iar procesorul gazdă așteaptă rezultatele în procedura `cudaThreadSynchronize()`. În continuare vom prezenta codul unei posibile implementări CUDA, folosind un vector de threaduri.

```
// Codul care va rula pe GPU
__global__ void threadScriere(int * a, int n) {
        // indicele threadului
        int i = blockDim.x * blockIdx.x + threadIdx.x;
        if (i < n)
                a[i] = i;
}
// functia main() din C standard, se executa pe CPU
int main(int argc, char ** argv) {
    int n = 1024;  // dimensiune
    int *a_cpu;    // pointer la memoria gazda
```

Etapele unui program care folosește CUDA sunt, de obicei:

1. alocarea memoriei pe dispozitiv,
2. copierea datelor de intrare din memoria gazdă în memoria dispozitiv,
3. apel nucleu CUDA: lansarea unei grile de threaduri,

[2]Copierea are loc, de obicei pe magistrala pe care se află placa video, iar viteza acestuia limitând performanța sistemului, se caută minimizarea numărului de transferuri.

4. calcul pe dispozitivul GPU

5. sincronizarea gazdă-dispozitiv prin așteptarea terminării execuției tuturor threadurilor,

6. preluarea rezultatului din memoria dispozitiv.

Pașii 1-3,5-6 sunt executate de către calculatorul gazdă, iar 4 este executat de procesorul grafic. În acest timp, procesorul gazdă (CPU) este inactiv, sau poate fi programat pentru alte activități. Astfel, putem considera GPU ca un *co-procesor* care poate prelua secțiunile de calcul intensiv din program.

Bibliografie

[1] Răzvan Andonie, Angel Cațaron, Honorius Gâlmeanu, Zoltan Gaspar, Lorentz Istvan, Mihai Ivanovici, Lucian Sasu, *Algoritmi și structuri de date, Îndrumar de laborator*, Universitatea „Transilvania" din Brașov, 2012