5.CUDA-programare masiv paralelă folosind procesoare grafice

CUDA este o platformă de calcul paralel și o interfață de programare a aplicațiilor dezvoltată cu scopul de a accelera operațiile de calcul masiv pe date multiple prin folosirea puterii de calcul disponibilă în procesoarele grafice GPU. Această platformă permite dezvoltatorilor și inginerilor de software să utilizeze o unitate de procesare grafică cu procesor CUDA pentru procesarea generală, abordare numită GPGPU (engl.General-Purpose computing on Graphics Processing Units). Termenul GPGPU denotă un procesor grafic cu o flexibilitate ridicată de programare, capabil de a rezolva și probleme generale de procesare. In executie, o arhitectură de tip GPU foloseste paradigma SIMD (single instruction multiple data) din taxonomia Flynn, ceea ce presupune schimb rapid de context între thread-uri, planificarea în grupuri de thread-uri și orientare către prelucrări masive de date.

5.1.Obiective

- Prezentarea conceptelor de programare paralelă CUDA
- Prezentarea unor exemple de algoritmi accelerați cu această bibliotecă
- Utilizarea unor instrumente pentru analiza de performanță

5.2. Concepte

Plăcile grafice au devenit atât de puternice încât sunt folosite pentru calcul matematic divers, cum ar fi operații complexe cu matrici, vectori operații necesare pentru diverse simulări vizuale și fizice complexe în diverse domenii cum ar fi :industria auto, prelucrări video și de imagini,criptografie,design electronic,simulări fizice diverse, prelucrări multimedia. Unitatile tip GPU sunt potrivite pentru paralelismul de date, intensiv computationale. Datorită faptului că aceleași instrucțiuni sunt executate pentru fiecare element, nu sunt necesare mecanisme complexe pentru controlul fluxului. Ierarhia de memorie este simplificată comparativ cu cea a unui procesor x86/ARM. Deoarece calculele sunt intensive computațional, latența accesului la memorie poate fi ascunsă prin paralelism (massive multithreading, SIMT sau Single Instruction Multiple Threads) în locul folosirii extensive a memoriei cache.

NVIDIA a susținut această tendință prin lansarea bibliotecii CUDA (Compute Unified Device Architecture), pentru a permite dezvoltatorilor de aplicații să scrie cod care poate fi încărcat pe o placă de tip NVIDIA pentru a fi executat de GPU-urile acesteia .Multe aplicații care procesează seturi mari de date, pot utiliza un model de programare bazat pe paralelismul de date pentru a accelera calculele, de exemplu în randarea 3D procesarea de imagini diverse seturi mari de pixeli și arce sunt mapate către threaduri paralele.

Spre deosebire de CPU , GPU-urile sunt optimizate pentru calcul paralel pe multiple seturi de date, diferenţa majoră între arhitectura CPU şi cea a GPU constă în modalitatea de acces la memorie: CPU-ul folosește între 1 și 3 canale de memorie de câte 64 biţi lăţime fiecare, în timp ce GPU-urile actuale pot folosi până la 8 canale de memorie paralele, de 64 biţi lăţime fiecare. nVidia a identificat acest potenţial al GPU-urilor şi a proiectat o interfaţă de programare prin care se permite anumitor programe accesul la această putere de calcul. Acest model a fost implementat în toate GPU-urile începand cu seria 8800, fiind suportat în principal pe GPU-urile NVIDIA construite pe arhitectura Tesla, iar plăcile grafice care susțin execuția aplicațiilor ce folosesc biblioteca CUDA sunt seriile GeForce 8, Quadro, și Tesla.

Platforma CUDA reprezintă un nivel software care oferă acces direct la setul de instrucțiuni virtuale din GPU și la componentele software specifice pentru executarea așa numitelor nuclee de calcul, fiind concepută pentru a lucra cu limbaje de programare precum C, C ++ și Fortran.Spre deosebire de API-urile pentru procesare grafică cum ar fi Direct3D și OpenGL, care necesită abilități avansate în programarea grafică, această nouă abordare facilitează specialiștilor în programarea

paralelă să utilizeze resursele GPU relativ simplu. Complementar, CUDA susține cadre de programare de nivel mai înalt precum OpenACC și OpenCL. OpenCL, ca standard alternativ suportat de Khronos și implementat de majoritatea producătorilor de GPU (inclusiv NVIDIA ca o extensie la CUDA). Problema majoră însă constă în faptul că suportul oferit este incomplet și standardul este mult mai restrictiv decât CUDA și cu complexitate mai mare în programare.

5.2.1. Modelul de programare

Modelul de programare CUDA este un model heterogen în care este utilizat atât CPU pentru controlul programului, cât și GPU pentru prelucrările masive de date. Placa grafică GPU ,numită în terminologia CUDA dispozitiv (engl.device), reprezintă un co-procesor "multithread" al CPU numit gazdă (engl. host) care poate executa un număr foarte mare de thread-uri identice în paralel. Thread-urile CUDA sunt threaduri "ușoare" (engl. light-threads),ceea ce înseamnă că mecanismele suplimentare necesare pentru crearea acestora, comutarea contextului și planificarea lor sunt foarte rapide. Scopul final al unei procesări în modelul CUDA este să se creeze cât mai multe astfel de threaduri în așa fel încât să se utilizeze cât mai complet și optim resursele hardware disponibile oferite de procesorul grafic. Intrucât procesoarele grafice pot rula mii de thread-uri concomitent, acestea devin eligibile pentru aplicații ce permit un grad mare de paralelizare, astfel principalul avantaj al arhitecturii CUDA este capacitatea de procesare masivă de date într-un interval de timp rezonabil și limitat.

Stiva software CUDA este ilustrată în Figura 5.1 ,iar în modelul de programare se folosesc următoarele concepte :

- *Host/Gazdă* = CPU (unitatea centrală de procesare) + memoria sa
- *Device/Dispozitiv* = set de multiprocesoare GPU (unitatea de procesare grafică) + memoria sa
- *Multiprocesor* = set de procesoare + memorie partajată
- *Bloc* (bloc de thread-uri) = grup de threaduri care execută un cod unic (kernel) pe un set de date identificate de threadID și blockID. Thread-urile pot comunica prin memoria partajată
- *Kernel* = programul (codul) asociat fiecărui thread executat în GPU
- *Grid* = grup (arie) de blocuri de thread-uri care execută kernelul
- *Warp* = grup de thread-uri care pot fi planificate simultan pentru execuție (de ex. dimensiunea unui warp = 32)

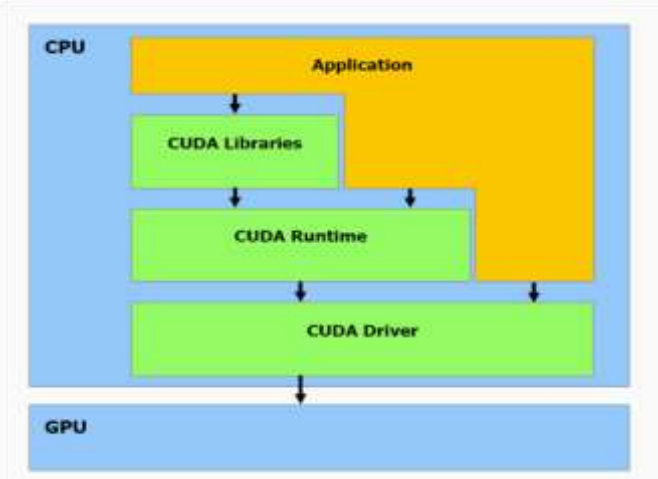

Figura 5.1. Stiva software CUDA [2]

Platforma CUDA utilizează un limbaj similar limbajului C ++ , care posedă extensii pentru a folosi caracteristicile specifice GPU-urilor și anume : apeluri specifice modelului de memorie partajată și pentru sincronizare între thread-uri, respectiv calificatori care se aplică funcțiilor și

variabilelor pentru diferențierea lor. Funcțiile specifice CUDA sunt numite kernel-uri, un astfel de kernel poate fi o funcție sau un program complet invocat de către CPU, ce este executat de N ori în paralel pe un GPU folosind un număr N de thread-uri.

Programarea CUDA implică rularea codului pe două platforme diferite, concomitent: un sistem gazdă cu unul sau mai multe procesoare și unul sau mai multe dispozitive CUDA NVIDIA GPU activate. De cele mai multe ori GPU-urile NVIDIA sunt frecvent asociate cu procesarea grafică, însă acestea sunt de asemenea motoare aritmetice puternice, capabile sa ruleze mii de thread-uri în paralel.Compilatorul NVIDIA (nvcc) este capabil să separe codul sursă în: funcții dispozitiv – procesate de compilatorul NVIDIA, respectiv funcții gazdă – procesate de compilator standard (de exemplu, gcc).

Codul care rulează pe host poate gestiona memoria atât pe aceasta cât și pe dispozitiv. Tot din host se lansează și kernel-urile pentru a fi executate de dispozitiv (placa grafică cu procesoarele sale). Având în vedere caracterul heterogen al modelului de programare CUDA, o secvență tipică de operații pentru un program CUDA C/C++ conține 5 pași ,iar fluxul de procesare este ilustrat în Figura 5.2.:

1. declararea și alocarea memoriei gazdă
2. inițializarea datelor gazdă
3. transferare date de la gazdă la dispozitiv
4. executarea unuia sau mai multor kernel-uri pe dispozitiv în mod paralel
5. transferul rezultatelor de la dispozitiv la gazdă pentru afișare și eliberarea memoriei

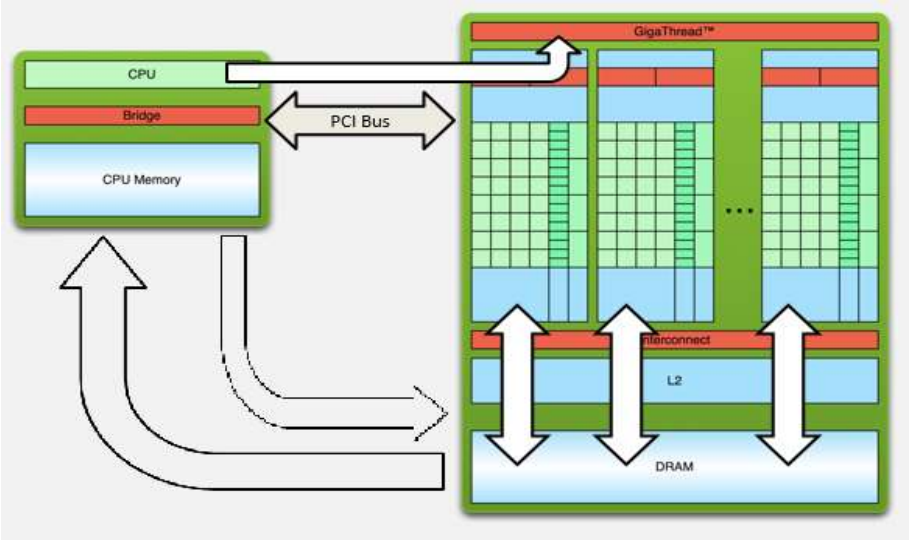

Figura 5.2. Flux de procesare CUDA [2]

Un exemplu de cod scris în CUDA pentru *adunarea a doi întregi*:

__global indică o funcție care: rulează pe dispozitiv și este apelată din gazdă

NUME_Kernel <<< Numar_blocuri_grila, Numar_thread-uri_bloc >>>(parametrii); marchează un apel din codul gazdă la codul dispozitiv (”Kernel launch”)

Exemplu: Adunarea a 2 numere întregi

```
//un kernel simpul pentru adunarea a 2 întregi
__global__ void add(int *a, int *b, int *c)
 { *c = *a + *b; }

int main(void) {
    int a, b, c;   // Copiile variabilelor a, b, c  de pe gazda
```

```
int *d_a, *d_b, *d_c;  // Copiile variabilelor a, b, c  de pe dispozitiv
int size = sizeof(int);
// Alocare spatiului pentru copiile variabilelor a, b, c de pe dispozitiv
cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);
// Setarea valorilor de intrare
a = 2;
b = 7;
// Copierea valorilor de intrare în dispozitiv
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);
// Lansarea kernelului add() pe GPU
add<<<1,1>>>(d_a, d_b, d_c);
// Copierea rezultatelor inapoi pe gazda
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
// Eliberarea memoriei
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
return 0; }
```

Execuția paralelă este descrisă de funcția kernel care este executată pe un set de thread-uri în paralel pe GPU. Acest cod kernel este un cod C ++ pentru doar un singur thread. Numărul de blocuri thread și numărul de thread-uri din acele blocuri care execută acest kernel în paralel sunt date explicit la apelul funcției. O funcție kernel poate fi invocată în cadrul codului serial, iar pentru a apela funcția kernel *configurația specifică execuției* trebuie specificată și anume numărul de thread-uri într-un bloc thread și numărul de thread-uri dintr-un grid.

Biblioteca CUDA conține următoarele componente : o componentă care rulează pe host și oferă funcții de control și accesare a unuia sau mai multor dispozitive de calcul host, o componentă dispozitiv, care rulează pe dispozitiv și oferă funcții specifice acestuia, o componentă comună, care oferă tipuri încorporate de vectori și un subset de funcții al bibliotecii standard C++, ce sunt acceptate atât în codul gazdă, cât și în cel al dispozitivului. Aceste componente și utilitatea lor sunt prezentate în Tabelul 6.1.

Componenta host	Suportă dispositive multiple
	Memoria: liniară sau tablouri CUDA
	Interoperabilitate OpenGL și DirectX
	Asincronicitate: funcția __global__ și cele mai multe funcții revin în aplicație anterior momentului în care dispozitivul încheie execuția taskului solicitat
Componenta dispozitiv	Funcții de sincronizare
	Conversie de tip
	Casting de tip
	Funcții atocmice (operații de tip read-modify-write pe cuvinte de 32 biti din memoria globală)
Componenta comună	Tipuri Built-in de vectori (float1, float2, int3, ushort4, etc)
	Creare constructor de tip : int2 i = make int2(i, j)
	Funcții matematice (standard math.h pe CPU)
	Funcții de timp pentru benchmarking
	Referințe la structuri de threaduri

Tabel 5.1. Componente CUDA și semnificația lor

Pentru a declara un grid și un bloc thread, CUDA oferă un tip de date predefinit, un vector de tip integer care specifică dimensiunile grid-ului și a blocurilor thread. În apelul funcției kernel variabilele grid și bloc sunt scrise între trei paranteze unghiulare <<< grid, block >>>, în acest mod în apel , grid-ul și blocurile thread sunt create *dinamic.*

Valorile acestor variabile grid și bloc trebuie să fie mai mici decât dimensiunile permise maximale. Funcția kernel are întotdeauna un tip de returnare void și un calificator __global__ care înseamnă că aceasta este o funcție kernel care urmează a fi executată pe GPU.

5.2.2. Managementul threadurilor

Arhitectura CUDA este construită în jurul unei *matrice scalabile de multiprocesoare* cu mai multe fire de execuție (numite și Multithreaded Streaming Multiprocessors - SMS). Când un program CUDA de pe host invocă o grilă kernel, blocurile rețelei sunt enumerate și distribuite multiprocesoarelor ce au capacitate de execuție disponibilă. Thread-urile unui bloc de thread-uri se execută simultan pe un singur multiprocesor, deasemenea mai multe blocuri de thread-uri se pot executa concurent pe un singur multiprocesor.

Pentru a rula cod paralel pe dispozitiv (de exemplu, funcția add() să se execute de N ori în paralel, pentru adunarea a 2 vectori), sintaxa este:

add<<<1,1>>> → add<<<N,1>>>

Fiecare invocare paralelă a lui add() se numește **bloc**.

Un set de blocuri formează un *grid.*

Fiecare invocare a funcției se poate referi la indexul blocului său utilizând *blockIdx.x*

*Exemplu***:**

```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];}
```

Kernelurile CPU sunt proiectate pentru a minimiza latența pentru unul sau mai multe thread-uri ce se execută întreșesut utilizând secvențial procesorul, în timp ce GPU-urile sunt proiectate să se ocupe de un număr foarte mare de thread-uri concurente în scopul de a maximiza randamentul în procesare prin creșterea vitezei de execuție.În CUDA, un bloc poate fi împărțit în thread-uri paralele. Referirea la indexul unui thread se face utilizând *threadIdx.x* (vezi Figura 5.3.).

Modelul CUDA structurează un calcul paralel folosind abstracțiunile threaduri, blocuri și griduri. Thread-ul este doar o execuție a unui kernel cu un anumit index. Fiecare thread folosește indexul său pentru a avea acces la elemente, astfel încât thread-urile procesează în mod cooperativ întregul set de date.

Programul are două secțiuni – una serială, care se execută în CPU și una paralelă care se execută în fiecare din procesoarele din GPU, în paralel. In codul sursă va exista o secțiune serială și unul sau mai multe apeluri ale kernel-ului .Organizarea aplicației se va realiza ca în Figura 5.3. în manieră grilă, blocuri și thread-uri, astfel o grilă conține mai multe blocuri și un bloc are mai multe threaduri.

*Exemplu***:** Adunare cu un bloc cu mai multe thread-uri:

```
// Lansarea kernel-ului add() pe GPU cu N blocuri și THREADS_PER_BLOCK thread-uri
add<<< N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);
```

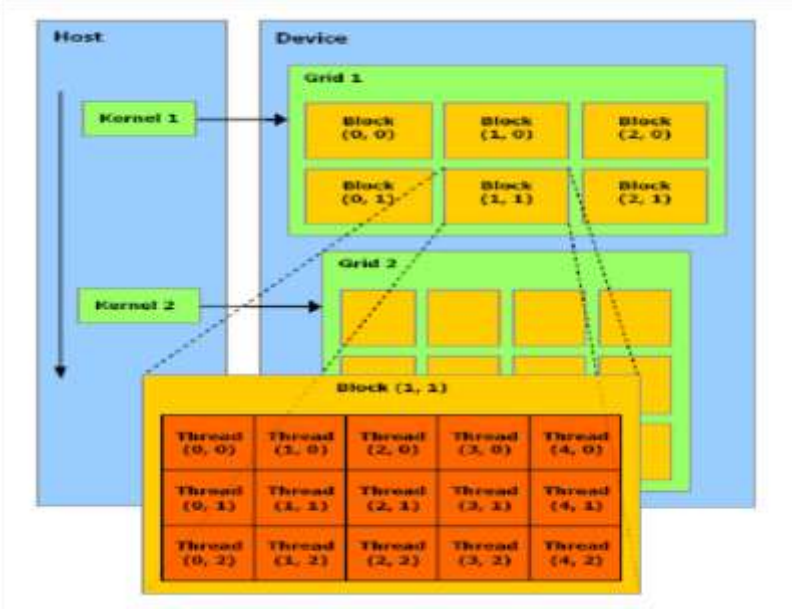

Figura 5.3. Structuri de threaduri

5.2.3. Modelul de memorie și modelul hardware

Memoria partajată este rapidă în comparație cu memoria device și în mod normal necesită aceeași durată de timp cât este necesară pentru a accesa registrele, fiind numită și Parallel Data Cache (PDC). Memoria partajată este "locală" fiecărui multiprocesor spre deosebire de memoria device și permite astfel o sincronizare locală mult mai eficientă. Aceasta este divizată în mai multe părți, astfel fiecare bloc thread dintr-un multiprocesor își accesează partea *proprie* de memorie partajată și această memorie partajată nu este accesibilă celorlalte blocuri thread ale acelui multiprocesor sau al oricărui alt multiprocesor. Toate thread-urile dintr-un bloc thread care au aceeași durată de viață ca și blocul din care fac parte utilizează această parte a memoriei atât pentru operații de citire cât și de scriere. Întrucât spațiul de memorie partajată este relativ limitat,aceasta trebuie folosită eficient.

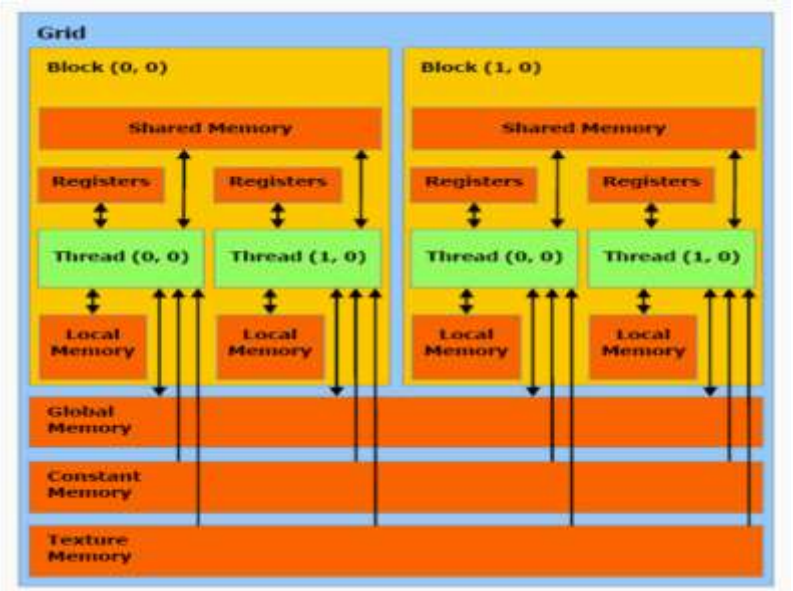

Figura 5.4. Modelul de memorie [2]

Toate multiprocesoarele accesează o memorie mare device globală atât pentru operații gather cât și pentru operații scatter, memorie care este relativ înceată deoarece nu oferă caching.

Fiecare multiprocesor are și cache-urile sale read-only pentru a crește viteza operațiilor de citire. Acestea sunt memoriile constant cache și texture cache.Fiecare thread conține de asemenea memoria lui locală. În mod normal variabilele locale ale funcțiilor kernel sunt alocate aici, însă pot fi alocate și în memoria globală. Într-un bloc, thread-urile partajează datele între ele utilizând *shared memory*.Memoria dispozitivului se numește *memorie globală*. Datele nu sunt vizibile thread-urilor din alte blocuri.

Pentru a declara variabile in memoria partajată se utilizează calificatorul **__shared__** și pentru a declara în memoria globală se folosește calificatorul **__device__.**

Dispozitivul este implementat ca un set de multiprocesoare așa cum este ilustrat în Figura 5.4. Fiecare multiprocesor are o arhitectură de tip SIMD (o singură instrucțiune pe date multiple), astfel la orice ciclu de ceas dat, fiecare procesor al multiprocesorului execută aceeași instrucțiune, dar operează pe date diferite.

Fiecare multiprocesor are o memorie on-chip de unul din următoarele patru tipuri: (i) un set de registre locale pe 32 de biți pe procesor; (ii) un cache de date paralel (memorie partajată) care este partajat de toate procesoarele și implementează un spațiu de memoria partajată (iii) o memorie cache constantă numai pentru citire care este partajată de către toate procesoarele și care accelerează citirile din spațiul de memorie constant și care este implementată ca o regiune de memorie a dispozitivului numai pentru citire; (iv) un cache doar pentru citire care este partajat de toate procesoarele și care accelerează citirile din spațiul de memorie a texturii, care este implementat ca regiune de memorie numai pentru citire din dispozitiv .

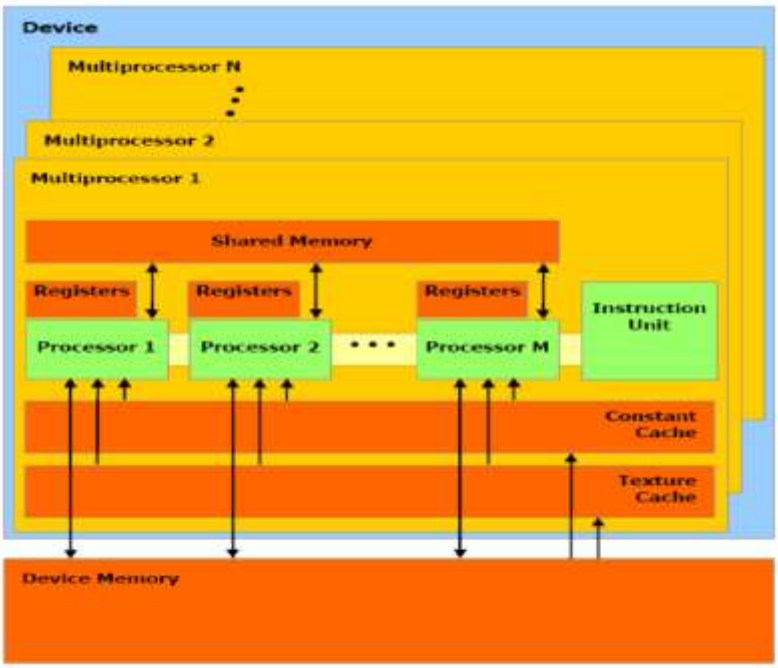

Figura 5.5. Modelul hardware [2]

Spațiile de memorie locale și globale sunt implementate ca regiuni de citire și scriere ale memoriei dispozitivului și nu sunt stocate în cache. Fiecare multiprocesor accesează cache-ul texturii printr-o unitate de textură care implementează diversele moduri de adresare și filtrarea datelor.

O grilă de blocuri de threaduri este executată pe dispozitiv prin executarea unuia sau mai multor blocuri pe fiecare multiprocesor , astfel fiecare bloc este împărțit în grupuri SIMD ale threadurilor numite warpuri; fiecare din aceste warpuri conține același număr de threaduri, numit mărimea warp, și este executat de multiprocesor într-un mod SIMD; un planificator de threaduri trece periodic de la un warp la altul pentru a maximiza utilizarea resurselor computaționale ale multiprocesorului.Un bloc este procesat de un singur multiprocesor, astfel încât spațiul de memorie partajat se află în memoria partajată pe-chip, ceea ce duce la accesări foarte rapide . Regiștrii multiprocesorului sunt alocați între threadurile blocului. Dacă numărul de registre utilizate pe thread

înmulțit cu numărul de threaduri din bloc este mai mare decât numărul total de registre per multiprocesor, blocul nu poate fi executat și nucleul corespunzător nu se va lansa. Mai multe blocuri pot fi procesate simultan de același multiprocesor prin alocarea registrelor multiprocesorului și a memoriei partajate între blocuri

Modul cum accesam memoria impacteaza foarte mult performanța sistemului. Cum putem avea arhitecturi foarte diferite din punct de vedere al ierarhiei de memorie este important de înteles ca nu putem dezvolta un program care să ruleze optim în toate cazurile. Un program CUDA este portabil și astfel poate fi usor rulat pe diferite arhitecturi NVIDIA CUDA, insa de cele mai multe ori trebuie ajustat în funcție de arhitectură pentru o performanță optimă.In general pentru arhitecturile de tip GPU, memoria locală este impărțită în module de SRAM identice, denumite bancuri de memorie (memory banks). Fiecare banc conține o valoare succesiva de 32 biți (de exemplu, un int sau un float), astfel încât accesele consecutive intr‑un array provenite de la threaduri consecutive să fie foarte rapid. Conflicte au loc atunci cand se fac cereri multiple asupra datelor aflate în acelasi banc de memorie.

Conflictele de access la bancuri de memorie (cache) pot reduce semnificativ performanța.Cand are loc un astfel de conflict, hardware‑ul serializează operațiile cu memoria (warp/wavefront serialization), și determină astfel toate threadurile să aștepte până când operațiile de memorie sunt efectuate. In unele cazuri, dacă toate threadurile citesc aceeați adresa de memorie shared, este invocat automat un mecanism de broadcast iar serializarea este evitată. Mecanismul de broadcast este foarte eficient și se recomandă folosirea sa de oricate ori este posibil.

5.2.4. Sincronizarea threadurilor

În scopul de sincronizare a thread-urilor API-ul CUDA oferă o funcție hardware de tip barieră de threaduri (thread-barrier numită *syncthreads()* care se comportă ca un punct de sincronizare. Întrucât thread-urile sunt programate în hardware, această funcție este implementată în hardware. Thread-urile vor aștepta în punctul de sincronizare până când toate thread-urile au ajuns în acest punct. Comunicarea între thread-uri dacă este necesară este posibilă prin memoria partajată a fiecărui bloc. Așadar, sincronizarea thread-urilor este posibilă numai la nivel de bloc thread. Deoarece s-ar putea ca thread-urile unui bloc să comunice între ele, aceste thread-uri trebuie executate pe același processor, de aceea, este garantată execuția unui bloc thread pe un singur processor.

Pentru a maximiza utilizarea resurselor disponibile, asignarea numărului de thread-uri pe bloc și numărului de blocuri thread pe grid ar trebui alese cât mai adecvat. Un număr mai mic de thread-uri pe bloc provoacă latență la încărcare la citirile memoriei device și de asemenea un bloc pe multiprocesor îl poate face pe acesta să fie inactiv în timpul sincronizării thread-urilor. Așadar, ar trebui să fie cel puțin două blocuri pentru fiecare multiprocesor în dispozitiv (numărul de blocuri pe grid ar trebui să fie cel puțin 100). De asemenea, ar trebui să se asigneze numărul de thread-uri pe bloc în multiplii ai dimensiunii warp, deoarece astfel se diminuează numărul warp-urilor subpopulate.

5.2.5. Controlul execuției

Întrucât funcția kernel rulează pe dispozitiv, memoria trebuie alocată în dispozitiv din timp înainte de invocarea funcției kernel și dacă funcția kernel trebuie să execute operații pe anumite date, atunci datele trebuie copiate din memoria host în memoria device anterior procesării.

Memoria device poate fi alocată fie ca memorie liniară, fie ca vectori CUDA. Calificatorul __device__ la începutul unei variabile specifică faptul că spațiul pentru această variabilă este alocat în memoria device. API-ul CUDA are și funcții de alocare și dealocare a memoriei device în timpul execuției, cum ar fi cudaMalloc(), cudaFree() etc. În mod similar, după execuția funcției kernel, datele din memoria device trebuie copiate înapoi în memoria host pentru a putea afișa rezultatele. Pentru a copia date în și din device spre host , API-ul CUDA oferă câteva funcții: *cudaMemCpyToSymbol(), cudaMemCpyFromSymbol(), cudaMemCpy(),* etc.

Lansările de Kernel-uri sunt asincrone, comanda revine imediat la CPU, astfel CPU trebuie să se sincronizeze înainte de a consuma rezultatele.

cudaMemcpy()	Blochează procesorul până când copia este finalizată Copierea începe când au fost terminate toate apelurile CUDA Anterioare
cudaMemcpyAsync()	Asincron, nu blochează CPU-ul
cudaDeviceSynchronize()	Blochează procesorul până când toate apelurile CUDA anterioare au terminat

Apelurile de kernel sunt asincrone din punctul de vedere al procesorului, deci dacă se apeleaza două kerneluri în succesiune, al doilea va fi lansat fără sa mai aștepte terminarea primului.În GPU, în cazul în care nu sunt specificate fluxuri diferite pentru a executa kernel-ul, acestea vor fi executate în ordinea în care au fost apelate ,astfel dacă este specificat un flux, ambele sunt executate în serie și numai după ce primul kernel este terminat se va executa cel de al doilea .

5.2.6. Tratarea erorilor

Toate funcțiile CUDA au o valoare returnată, care poate fi utilizată pentru a verifica erorile care apar în timpul executării lor (*cudaError).*

- Erori de apeluri API Cuda. De exemplu, un apel la cudaMalloc () ar putea eșua.
- Erori de la apelurile de kernel Cuda. De exemplu, ar putea exista acces la memorie invalidă în interiorul unui nucleu.

Exemplu:

```
if ( cudaSuccess != cudaMalloc( &fooPtr, fooSize ) )
       printf( "Error!\n" );
```

Invocările kernelului CUDA nu returnează nici o valoare. Eroarea de la un apel de kernel CUDA poate fi verificată dupa executarea acestuia prin apelarea metodei cudaGetLastError ():

Exemplu:

```
fooKernel<<< x, y >>>(); // Kernel call
if ( cudaSuccess != cudaGetLastError() )
       printf( "Error!\n" );
```

5.2.7. Gestionarea dispozitivelor

Pentru mașini multi-GPU, utilizatorii pot să selecteze care GPU este ales pentru utilizare. În mod implicit driverul CUDA selecteaza cel mai rapid GPU ca dispozitiv 0. NVIDIA furnizează câteva mijloace prin care pot fi interogate și gestionate dispozitivele GPU fiind importantă interogarea informațiilor deoarece, aceasta poate fi folosită în setarea configurării execuției kernel-ului la runtime.

- Aplicația poate interoga și selecta GPU-uri:
  cudaGetDeviceCount(int *count)
  cudaSetDevice(int device)
  cudaGetDevice(int *device)
  cudaGetDeviceProperties(cudaDeviceProp *prop, int device)

- Thread-uri multiple pot partaja un dispozitiv
- Un singur thread poate gestiona mai multe dispozitive
  cudaSetDevice(i) – pentru a selecta dispozitivul curent
  cudaMemcpy(…)- pentru copii peer-to-peer

5.2.8. Transferul de date între Host și Device

Întrucât lățimea de bandă între memoria device și memoria host este mult mai mică comparativ cu lățimea de bandă între dispozitiv și memoria device care este foarte mare, este de dorit minimizarea transferului de date între host și device. În scop de optimizare este utilă crearea și distrugerea structurilor de date direct în memoria device (în loc să le copiem) și efectuarea de transferuri mari prin combinarea și prelucrarea a mai multor transferuri mai mici pentru a diminua costurile transferului.

5.2.9. Timpul de execuție

Măsurarea timpului de execuție al unui program CUDA se realizează prin definirea unui timer și utilizarea funcțiilor aferente:

```
StopWatchInterface *timer = NULL;
sdkCreateTimer(&timer);
sdkStartTimer(&timer);
sdkDeleteTimer(&timer);
```

5.2.10. Calificatori pentru funcții și variabile

Calificatori de Tip de Funcție

Cele trei tipuri principale de calificatori de funcție în CUDA sunt *device, global și host.*

1. *__device__* :Funcțiile cu calificatorul device sunt executate pe dispozitiv. Aceste funcții sunt apelabile numai din device.
2. *__global__ :*Funcțiile cu calificatorul global sunt executate pe dispozitiv dar sunt apelabile numai din host.
3. *__host__* :Funcțiile cu calificatorul host sunt executate pe host și sunt apelabile numai din host.

Când nu se folosește niciun calificator, înseamnă că funcția va rula pe host; este echivalent cu funcțiile declarate cu calificatorul __host__.

Calificatori de Tip de Variabilă

Cele trei tipuri principale de calificatori de variabile în CUDA sunt device, constant, și shared.

1. *__device__* :Variabilele declarate cu __device__ se află pe dispozitiv. Alți calificatori de tip sunt folosiți opționali împreună cu __device__. Dacă o variabilă este declarată numai cu calificatorul __device__ atunci această variabliă se află în memoria globală și are aceeași durata de viață ca aplicația. Întrucât aceasta se află în memoria globală, este accesibilă din toate thread-urile (din grid) și din host prin librăria runtime.

2. *__constant__* :Acest calificator este folosit pentru a aloca constante pe dispozitiv. Este folosit opțional împreună cu calificatorul __device_. Această constantă se află în memoria constant, și are același durata de viață ca aplicația. Este accesibilă din toate thread-urile (din grid) și din host prin biblioteca runtime.

3. *__shared__* :Acest calificator este folosit pentru a aloca variabile shared. Este folosit opțional împreună cu calificatorul __device__. Variabilele shared se află în memoria share a unui bloc thread, și are durata de viață a blocului respectiv. Este accesibilă numai din thread-urile din blocul respectiv.

Variabile Incorporate. Aceasta este o listă cu câteva dintre variabilele încorporate în CUDA:

1. gridDim: este de tip dim3 și conține dimensiunile grid-ului.
2. blockIdx: este de tip uint3 și conține indexul blocului din grid.
3. blockDim: este de tip dim3 și conține dimensiunile blocului.
4. threadIdx: este de tip uint3 și conține indexul thread-ului din bloc.
5. warpSize: este de tip int și conține dimensiunea warp-ului în thread-uri.

Componenta Device Runtime. Componentele device runtime sunt folosite numai în funcțiile device și sunt prefixate o liniuță de subliniere __. Cele mai importante funcții sunt :funcții Matematice, de Sincronizare, Atomice, de Texturare.

Modul Device Emulation. Un mod device emulation este oferit în scopuri de debugging. Opțiunea –device nu este folosită împreună cu comanda nvcc compile. Aceasta emulează doar dispozitivul. Thread-urile și blocurile thread sunt create în host.

Debugging-ul nativ al host-ului (ca de exemplu Visual Studio de la Microsoft) poate fi folosit pentru a seta breakpoints și pentru inspectarea datelor, fiind în special folositor în operațiile de input sau output spre fișiere sau spre ecran, cum ar fi utilizarea funcției printf(), care nu se poate rula pe dispozitiv.

5.2.12. Reguli și elemente de optimizare în scrierea codului

Un set minim de extensii la limbajul C, care permit programatorului să vizeze porțiuni ale codului sursă pentru a fi executate pe dispozitiv Cele mai importante aspecte ce vizează regulile necesare în implementarea unor aplicații care să beneficieze optim de modelul CUDA sunt :

1. Programarea simplă C este suportată de către compilatorul CUDA. Lipsesc folosirea funcțiilor de programare orientată pe obiect sau a funcțiilor C++ în codul dispozitivului.
2. Arhitectura eterogenă este folosită pentru a realiza interacțiunea între modelele de programare CPU și GPU. Datele ar putea să fie copiate din memoria host în memoria device, iar rezultatele sunt copiate înapoi în host din memoria device.
3. Invocarea funcției kernel: Grid-ul, blocurile thread, și thread-urile sunt create de către invocarea funcției kernel din host. Acesta este singurul mod de a le crea. Acestea nu pot fi create în funcția kernel. Mai mult, numărul de grid-uri și de blocuri thread nu trebuie să depășească valorile lor maxime permise.
4. Funcțiile kernel nu returnează niciun rezultat, adică tipul lor de returnare este mereu void. Mai mult, apelul funcției kernel este asincron. Așadar, controlul revine înapoi înainte de completarea funcției kernel pe dispozitiv. Mai multe informații pot fi găsite în ghidul de programare CUDA. Toate funcțiile cu calificativul __device__ sunt implicit inline.
5. Recursivitatea este pur și simplu interzisă în funcțiile kernel din cauza cerințelor mari de memorie pentru mii de thread-uri.
6. Alocarea și dealocarea memoriei device în timpul rulării este posibilă numai în timpul utilizării codului host și înaintea apelării codului device. Asta înseamnă că în codul device, memoria device nu poate fi nici alocată nici dealocată folosind funcții precum cudaMalloc(), cudaFree() etc. Toate alocările necesare pentru o funcție kernel specifică sunt realizate înaintea apelului acelei funcției kernel din codul host și, similar, toată memoria device alocată este dealocată după finalizarea acelei funcții kernel din codul host.
7. Memoria partajată este împărțită numai între thread-urile din același bloc thread. Thread-urile dintr-un bloc thread diferit nu o pot împărți.
8. Variabilelor incorporate cum ar fi blockIdx, threadIdx, etc, nu le pot fi asignate nicio valoare. Mai mult, nu este posibilă preluarea adreselor acestora.

9. Variabilele declarate cu calificatorii __device__, __shared__, sau __constant__ au și câteva restricții. Adresa unei variabile folosind oricare dintre acești calificatori poate fi folosită numai în codul device.

Comunicarea și sincronizarea între thread-uri sunt posibile numai la nivelul de bloc thread., iar comunicarea între blocuri thread nu este permisă. O posibilă problemă ar fi concurența și consistența datelor. Atunci când mai multe thread-uri accesează aceeași zonă de memorie, trebuie asigurat faptul că datele sunt consistente. Se pot folosi urmatoarele soluții:

- operații atomice (in care un singur thread are acces la o anumită resursă la un anumit moment de timp)
- sincronizarea threadurilor
- proiectarea sau modificarea algoritmului ăn asa maniera încât să nu apară conflicte

O altă problemă o constituie accesul la memorie. Intrucât accesele la memorie sunt mult mai lente decât timpul de realizare a diverselor instrucțiuni, apare situația în care procesorul grafic trebuie să aștepte (stă neutilizat-idle) până o anumită operație cu memoria este îndeplinită. Pentru acest lucru se pot folosi optimizări precum:

- utilizarea memoriei mai rapide: partajată și locală
- accesul uniform la memorie (coalesced access): evitarea salturilor în memorie pentru date conectate logic și gruparea lor pentru eficientizarea acceselor
- evitarea ramificării codului: evitarea instrucțiunilor condiționale în care ramurile să aibă timpi de prelucrare disproportionați, deoarece thread-urile se lansează în grupuri (warp-uri) de câte 32, iar fiecare thread așteaptă după terminarea execuției celorlalte thread-uri
- paralelizarea apelurilor: apelurile pentru GPU de manipulare a memoriei și execuție de funcții pot fi făcute pe canale (stream-uri) distincte, astfel încât execuția uneia să fie condiționată de terminarea execuției tuturor celorlalte

NVIDIA Visual Profiler este o aplicatie, parte a NVIDIA CUDA Toolkit, care poate să ofere multe informatii cu privire la problemele de performanță dintr‑o aplicație CUDA. Se crează o sesiune si se indică binarul împreună cu argumente la care se dorește profilare.Sunt oferite informații detaliate asupra timpilor de execuție, ocuparea resurselor GPU cât și indicații despre cum se poate îmbunătăți performanța (informații suplimentare : https://docs.nvidia.com/cuda/profiler‑usersguide/index.html)

5.3. Exemple

5.3.1.Exemplu de structură de cod CUDA (sursa [3])

Program	Observatii
const int N = 1024; const int blocksize = 16;	Set grid size

__global__
void add_matrix(float* a, float *b, float *c, int N)
{
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y +
threadIdx.y; int index = i + j*N; if (i < N
&& j < N)
c[index] = a[index] +
b[index]; }
	Compute kernel

int main() {	

float *a = new float[N*N];
float *b = new float[N*N];
float *c = new float[N*N];
for (int i = 0; i < N*N; ++i) a[i] = 1.0f; b[i] = 3.5f;
	CPU memory allocation

float *ad, *bd, *cd;
const int size = N*N*sizeof(float);
cudaMalloc((void**)&ad, size);
cudaMalloc((void**)&bd, size);
cudaMalloc((void**)&cd, size);
	GPU memory allocation

cudaMemcpy(ad, a, size, cudaMemcpyHostToDevice); cudaMemcpy(bd, b, size, cudaMemcpyHostToDevice);	Copy data to GPU

dim3 dimBlock(blocksize, blocksize);
dim3 dimGrid(N/dimBlock.x, N/dimBlock.y);
add_matrix<<<dimGrid, dimBlock>>>(d, bd, cd, N);
	Execute kernel

cudaMemcpy(c, cd, size, cudaMemcpyDeviceToHost);	Copy result back to CPU
cudaFree(ad); cudaFree(bd); cudaFree(cd); delete[] a; delete[] b; delete[] c; return EXIT_SUCCESS;	Clean up and return
}	

Instrucțiuni de execuție a exemplului de aplicație. Pentru a folosi modelul de programare masiv paralelă CUDA în implementarea de aplicații este nevoie de:

- Un GPU care suportă CUDA
- O versiune compatibilă de Windows
- O versiune compatibilă de Microsoft Visual Studio
- NVIDIA CUDA Toolkit (disponibilă la http://developer.nvidia.com/cuda-downloads) –versiunea curentă este 9.1

Pentru crearea unui nou proiect se vor urma etapele :New Project -> NVIDIA -> CUDA 9.1 -> CUDA 9.1 Runtime.
Pentru configurare se vor alege proprietățile corespinzătoare , astfel : -> Properties -> Configuration Properties -> VC++ Directories , unde se vor realiza următoarele configurări:
Include Directories: C:\ProgramData\NVIDIA Corporation\CUDA Samples\v9.1\common\inc\
Library Directories: C:\ProgramData\NVIDIA Corporation\CUDA Samples\v9.1\common\lib\

Pentru implementarea paralelă utilizând frameworkul CUDA se poate utiliza librăria Cuda C++ Thrust (https://developer.nvidia.com/thrust) care permite implementarea aplicațiilor paralele de înaltă performanță cu un efort mai mic de programare. Performanța execuției poate fi măsurată utilizând deasemenea extensia Concurency Visualizer din Visual Studio 201x.

5.3.2.Exemplu de paralelizare a unui program secvențial care calculează distanța între un punct specific și toate celelalte puncte dintr-o matrice 2D de dimensiune NxN (sursa [4]).

Cod secvențial

```
const int N=16;
void main (void) {
    int i, j, x, y;
    float hgrid[N][N];
    printf( "\n\tEnter the x coordinate of node : " ); scanf_s("%d", &x);
    printf( "\n\tEnter the y coordinate of node : " ); scanf_s("%d", &y);
    // Code to find distance without using device
```

```
    for (i=0; i<N; i++){
        for (j=0; j<N; j++) {
            n = ((i-x)*(i-x))+((j-y)*(j-y));    // distance formula
            hgrid[i][j] = sqrt(n);              // distance formula
            printf("\t%.0lf", hgrid[i][j]);}
        printf("\n\n");}}
```

Cod Paralel - 1D Grid

Același program este convertit în cod paralel pentru a fi rulat pe dispozitiv.
Un grid unidimensional cu numai un singur bloc thread este folosit. Blocul thread conține 16*16 thread-uri (așadar 256 de thread-uri în total) într-o formă bidimensională.

```
const int N=16;
__device__ float dgrid[N][N]; // array on device memory

// function on device to calculate distance
__global__ void findDistance( int x, int y){
    int i = threadIdx.x;
    int j = threadIdx.y;

    float n = ((i-x)*(i-x))+((j-y)*(j-y));
    dgrid[i][j] = sqrt(n);}

void main () {
    int i, j;
    float hgrid[N][N];
    dim3 dBlock(N, N);      // thread block with total 256 threads
    printf( "\n\tEnter the x coordinate of node : " ); scanf_s("%d", &i);
    printf( "\n\tEnter the y coordinate of node : " ); scanf_s("%d", &j);
    printf( "\n\tDistance from a node!\n\n\n" );

    findDistance<<<1, dBlock>>>(i, j); // Calling kernel function
    cudaMemcpyFromSymbol( &hgrid, dgrid, sizeof(dgrid)); //copy device M to host

    printf( "\n\n\tValues in hgrid!\n\n" );
    for (i=0; i<N; i++){
        for (j=0; j<N; j++) {
            printf("\t%.0lf", hgrid[i][j]);
            printf("\n\n"); }}}
```

*Cod Paralel - 2D Grid (2 * 2)*

Același program este convertit în cod paralel pentru a fi rulat pe dispozitiv cu un grid bidimensional (2 blocuri thread în dimensiunea x și 2 în dimensiunea y). Blocul thread conține 16 * 16 thread-uri (adică 256 de thread-uri în total) într-o formă bidimensională. Așadar, în total vor rula 1024 de thread-uri în paralel pe dispozitiv.

```
const int N=16;
const int D=2;
__device__ float dgrid[N*D][N*D]; // array on device memory

// function on device to calculate distance
__global__ void findDistance( int x, int y){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    float n = ((i-x)*(i-x))+((j-y)*(j-y));
    dgrid[i][j] = sqrt(n);}

void main () {
    int i, j;
    float hgrid[N*D][N*D];
    dim3 dGrid(D,D);    // 2D grid with total 4 thread blocks
    dim3 dBlock(N, N);  // thread block with total 256 threads
```

```
    printf( "\n\tEnter the x coordinate of node : " ); scanf_s("%d", &i);
    printf( "\n\tEnter the y coordinate of node : " ); scanf_s("%d", &j);
    printf( "\n\tDistance from a node!\n\n\n" );

    findDistance<<< dGrid, dBlock>>>(i, j); // Calling kernel function
    cudaMemcpyFromSymbol( &hgrid, dgrid, sizeof(dgrid)); //copy device M to host

    printf( "\n\n\tValues in hgrid!\n\n" );
    for (i=0; i<N*D; i++){
        for (j=0; j<N*D; j++){
            printf("\t%.0lf", hgrid[i][j]);
            printf("\n\n");}}}
```

Cod Paralel - 2D Grid (4 * 4)

Același program este convertit în cod paralel pentru a fi rulat pe dispozitiv cu un grid bidimensional (4 blocuri thread în dimensiunea x și 4 în dimensiunea y). Blocul thread conține 8 * 8 thread-uri (adică 64 de thread-uri în total) într-o formă bidimensională. Așadar, în total vor rula 1024 de thread-uri în paralel pe dispozitiv.

```
const int N=8;
const int D=4;
__device__ float dgrid[N*D][N*D]; // array on device memory

// function on device to calculate distance
__global__ void findDistance( int x, int y){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    float n = ((i-x)*(i-x))+((j-y)*(j-y));
    dgrid[i][j] = sqrt(n);
}

void main () {
    int i, j;
    float hgrid[N*D][N*D];
    dim3 dGrid(D,D);    // 2D grid with total 16 thread blocks
    dim3 dBlock(N, N);  // thread block with total 64 threads
    printf( "\n\tEnter the x coordinate of node : " ); scanf_s("%d", &i);
    printf( "\n\tEnter the y coordinate of node : " ); scanf_s("%d", &j);
    printf( "\n\tDistance from a node!\n\n\n" );
    findDistance<<< dGrid, dBlock>>>(i, j); // Calling kernel function
    cudaMemcpyFromSymbol( &hgrid, dgrid, sizeof(dgrid));//copy device memory to
host

    printf( "\n\n\tValues in hgrid!\n\n" );
    for (i=0; i<N*D; i++){
        for (j=0; j<N*D; j++){
            printf("\t%.0lf", hgrid[i][j]);
            printf("\n\n"); }}}
```

5.3.3.Se vor testa exemplele ce ilustrează modul de utilizare a API-ului Cuda, exemple pe care le puteți descărca de la adresa https:/ftp.utcluj.ro /~civan/CPD/1_LABORATOR/05_CUDA

5.4. Întrebări teoretice

5.4.1.Care este avantajul major al arhitecturii CUDA?
5.4.2.Cum pot fi identificate threadurile create și executate la nivel de device
5.4.3.Prezentați ierarhia de memorie a platformei CUDA
5.4.4.Ce tipuri de calificatori pentru funcții și variabile sunt utilizați în programarea CUDA?

5.5. Probleme propuse

5.5.1.Să se implementeze algoritmii de sortare *radix,merge și bitonic* în manieră paralelă folosind mecanismele modelului de programare CUDA. Să se realizeze o comparație a timpului de execuție a acestora pentru o evaluare de performanță a lor ,raportat la complexitatea algoritmului și dimensiunea problemei, respectiv gradul de paralelism (DOP- degree of parallelism reprezintă număr de threaduri/procese ce s epot executa în paralel).

5.5.2. Implementați utilizând CUDA functia matrix_multiply_simple care va realiza inmultirea a două matrice primite ca parametru. Realizați o inmulțire optimizată a două matrice, folosind metoda Blocked Matrix Multiplication (https://en.wikipedia.org/wiki/Block_matrix). Se va folosi directiva shared pentru a aloca memorie partajata intre thread‑uri. Pentru sincronizarea thread‑urilor se folosește funcția __syncthreads. Măsurați timpul petrecut în kernel pentru fiecare din soluțiile implementate folosind evenimente CUDA. Realizați profilare pentru funcțiile implementate folosind tool‑urile nvprof si nvvp, respectiv NVIDIA Visual Profiler, la alegere.

5.6. Referințe bibliografice

1. Toolkit și tutorial http://developer.nvidia.com/cuda

2. Instrucțiuni pentru instalare și configurare
   Local :http://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/index.html
   Cloud:https://harshityadav95.medium.com/how-to-run-cuda-c-or-c-on-google-colab-or-azure-notebook-ea75a23a5962

3. Tutoriale https://www.tutorialspoint.com/cuda/cuda_tutorial.pdf
4. Exemple de aplicații :
   http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/samples.html

   https://github.com/zchee/cuda-sample

5. Librăria Cuda Thrust https://thrust.github.io/

CUDA C Programming Quick Reference[1]

Function Qualifiers

`__global__`	called from host, executed on device
`__device__`	called from device, executed on device (always inline when Compute Capability is 1.x)
`__host__`	called from host, executed on host
`__host__ __device__`	generates code for host and device
`__noinline__`	if possible, do not inline
`__forceinline__`	force compiler to inline

Variable Qualifiers (Device)

`__device__`	variable on device (Global Memory)
`__constant__`	variable in Constant Memory
`__shared__`	variable in Shared Memory
`__restrict__`	restricted pointers, assert to the compiler that pointers are not aliased (cf. aliased pointer)
– No Qualifier –	automatic variable, resides in Register or in Local Memory in some cases (local arrays, register spilling)

Built-in Variables (Device)

dim3 gridDim	dimensions of the current grid (gridDim.x, ...) (composed of independent blocks)
dim3 blockDim	dimensions of the current block (composed of threads) (total number of threads should be a multiple of warp size)
uint3 blockIdx	block location in the grid (blockIdx.x, ...)
uint3 threadIdx	thread location in the block (threadIdx.x, ...)
int warpSize	warp size in threads (instructions are issued per warp)

Shared Memory

Static allocation	`__shared__ int a[128]`
Dynamic allocation (at kernel launch)	`extern __shared__ float b[]`

Host / Device Memory

Allocate pinned / page-locked Memory on host	cudaMallocHost(&dptr, size) (for higher bandwidth, may degrade system performance)
Allocate Device Memory	cudaMalloc(&devptr, size)
Free Device Memory	cudaFree(devptr)
Transfer Memory	cudaMemcpy(dst, src, size, cudaMemcpyKind kind) kind = {cudaMemcpyHostToDevice, ...}
Nonblocking Transfer	cudaMemcpyAsync(dst, src, size, kind[, stream]) (host memory must be page-locked)
Copy to constant or global memory	cudaMemcpyToSymbol(symbol, src, size[, offset[, kind]]) kind=cudaMemcpy[HostToDevice|DeviceToDevice]

Synchronizing

Synchronizing one Block	__syncthreads() (device call)
Synchronizing all Blocks	cudaDeviceSynchronize() (host call, CUDA Runtime API)

Kernel

Kernel Launch	kernel<<<dim3 blocks, dim3 threads[, ...]>>>(arguments)

CUDA Device Management

Init device (context)	cudaSetDevice(devID)
Reset current device	cudaDeviceReset()

CUDA Runtime API Error Handling

CUDA Runtime API error as String	cudaGetErrorString(cudaError_t err)
Last CUDA error produced by any of the runtime calls	cudaGetLastError()

OpenGL Interoperability

Init device (within OpenGL context)	cudaGLSetGLDevice(devID) (mutually exclusive to cudaSetDevice())
Register buffer object (must not be bound by OpenGL)	cudaGraphicsGLRegisterBuffer(&res, id, flags) `Res:` cudaGraphicsResource pointer `id:` OpenGL Buffer Id `flags:` register flags (read/write access)
Register texture or render buffer	cudaGraphicsGLRegisterImage(&res, id, target, flags)

Graphics Interoperability

Unregister graphics resource	cudaGraphicsUnregisterResource(res)
Map graphics resources for access by CUDA	cudaGraphicsMapResources(count, &res[, stream])
Get device pointer (access a mapped graphics resource) (OpenGL: buffer object)	cudaGraphicsResourceGetMappedPointer(&dptr, size, res)
Get CUDA array of a mapped graphics resource (OpenGL: texture or renderbuffer)	cudaGraphicsSubResourceGetMappedArray(&a, res, i, lvl)
Unmap graphics resource	cudaGraphicsUnmapResources(count, &res[, stream])

CUDA Texture

Textures are read-only global memory, but cached on-chip, with texture interpolation

Declare texture (at file scope)	texture<DataType,TexType,Mode> texRef
Create channel descriptor	cudaCreateChannelDesc<DataType>()
Bind memory to texture	cudaBindTexture(offset, texref, dptr, channelDesc, size)
Unbind texture	cudaUnbindTexture(texRef)
Fetch Texel (texture pixel)	tex1D(texRef, x) tex2D(texRef, x, y) tex3D(texRef, x, y, z) tex1DLayered(texRef, x, layer) tex2DLayered(texRef, x, y, layer)

CUDA Streams (Concurrency Management)

Stream = instruction sequence. Streams may execute their commands out of order.

Create CUDA Stream	cudaStreamCreate(cudaStream_t &stream)
Destroy CUDA Stream	cudaStreamDestroy(stream)
Synchronize Stream	cudaStreamSynchronize(stream)
Stream completed?	cudaStreamQuery(stream)

[1] Incomplete Reference for CUDA Runtime API. July 5, 2012. Contact: wmatthias@t-online.de. Cf. Complete Reference: "NVIDIA CUDA C Programming Guide", Version 4.0

Technical Specifications

Compute Capability	1.0	1.1	1.2	1.3	2.x	3.0
Max. dimensionality of grid	2				3	
Max. dimensionality of block	3					
Max. x-,y- or z-dimension of a grid	$2^{16}-1$					$2^{32}-1$
Max. x- or y-dimension of a block	512				1024	
Max. z-dimension of a block	64					
Max. threads per block	512				1024	
Warp Size	32					
Max. resident blocks per SM	8					16
Max. resident warps per SM	24		32		48	64
Max. resident threads per SM	768		1024		1536	2048
Number of 32-bit registers per SM	8K		16K		32K	64K
Max. registers per thread	10 (+1)		16 (+1)		63 (+1)	
Max. shared memory per SM ($\geq$2.0: configurable L1 Cache)	16 KB				48KB or 16KB	
Number of shared memory banks	16				32	
Constant memory size	64 KB					
Local memory per thread	16 KB				512 KB	
Cache working set per SM for constant	8 KB					
Cache working set per SM for texture	device dependent, 6-8 KB					
Max. instructions per kernel	2 million					512 million
Max. width for 1D texture (CUDA array)	8192				65536	
Max. width 1D texture (linear memory)	2^{27}					
Max. width×layers 1D texture	8192×512				16384×2048	
Max. width×height for 2D texture	65536×32768				65536×65535	
Max. width×height×layers for 2D texture	$8192 \times 8192 \times 512$				$16384 \times 16384 \times 2048$	
Max. width×height×depth 3D texture	2048^3					4096^3
Max. textures bound to kernel	128					256
Max. width for 1D surface	N/A				65536	
Max. width×height for 2D surface	N/A				65536×32768	
Max. surfaces bound to kernel	N/A				8	16

Architecture Specifications

Compute Capability	1.0	1.1	1.2	1.3	2.0	2.1	3.0
Number of cores (with FPU and ALU)	8				32	48	192
Number of special function units	2				4	8	32
Number of texture units	2				4	8	32
Number of warp schedulers	1				2	2	4
Number of instructions issued at once by scheduler	1				1	2	2

[1]Sources: http://en.wikipedia.org/wiki/CUDA and Nvidia

Supported GPUs

CC	GPUs	Information
1.0	G80, G92, G92b, G94, G94b	*"Unified Shader Architecture"* Supporting GPU programming with C.
1.1	G86, G84, G98, G96, G96b, G94, G94b, G92, G92b	- 32-bit Integer atomic functions for global memory, ...
1.2	GT218, GT216, GT215	*"Tesla"* - Warp vote functions, ...
1.3	GT200, GT200b	- Double-Precision, ...
2.0	GF100, GF110	*"Fermi"* - ECC, Better Caches (L1 and L2), GigaThread-Engine, better atomics, dual warp, unified address space, ...
2.1	GF104, GF114, GF116, GF108, GF106	...
3.0	GK104, GK106, GK107	*"Kepler"* - Polymorph Engine 2.0, GPU Boost, TXAA, SMX (next-generation SM)
3.5	GK110 (GPGPU)	- Dynamic Parallelism, Hyper-Q, Grid Management Unit

CUDA Memory

Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	Thread	Thread
Local	Off-chip	No*	R/W	Thread	Thread
Shared	On-chip	N/A	R/W	Block	Block
Global	Off-chip	No*	R/W	Global	Application
Constant	Off-chip	Yes	R	Global	Application
Texture	Off-chip	Yes	R	Global	Application
Surface	Off-chip	Yes	R/W	Global	Application

*) Devices with compute capability $\geq$2.0 use L1 and L2 Caches.

Occupancy

$= \frac{\#\text{active warps per SM}}{\#\text{possible warps per SM}}$ (↗ ExcelSheet "Occupancy Calculator")

Higher occupancy ≠ better performance (it's just more likely to hide latencies)

Potential occupancy limiters: Register usage, Shared Memory usage, Block size

Helpful nvcc compiler flag: `--ptxas-options=-v`(show memory usage of kernel)