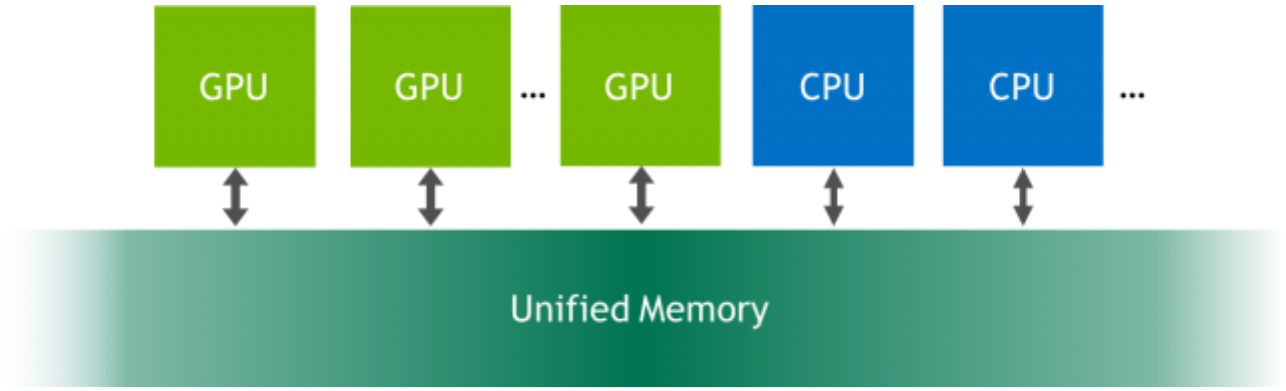
Advanced CUDA

Spatiu unificat memorie

De la CUDA 6.0 [http://developer.download.nvidia.com/compute/cuda/6_0/rel/docs/CUDA_Toolkit_Release_Notes.pdf], NVIDIA a schimbat semnificativ modelul de programare prin facilitarea comunicarii unitatii CPU (host) cu unitatea GPU (device), in mod transparent prin acelasi set de adrese de memorie virtuale. Astfel exista posibilitatea ca prin acelasi pointer de memorie sa se scrie date atat de catre CPU cat si de catre GPU. Evident transferurile de memorie au loc intre spatii diferite de adresare (ex RAM vs VRAM), dar acest lucru se intampla transparent la nivel de aplicatie CUDA / pentru programator.



Mai jos avem un exemplu de folosire a memoriei unificate. Singura diferenta fata de alocarea pe CPU/HOST este ca memoria trebuie alocata cu cudaMallocManaged si dealocata cu cudaFree.

```
#include <iostream>
#include <math.h>

// CUDA kernel to add elements of two arrays
__global__
void add(int n, float *x, float *y)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20;
  float *x, *y;

  // Allocate Unified Memory -- accessible from CPU or GPU
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Launch kernel on 1M elements on the GPU
  int blockSize = 256;
  int numBlocks = (N + blockSize - 1) / blockSize;
  add<<<numBlocks, blockSize>>>(N, x, y);

  // Wait for GPU to finish before accessing on host
  cudaDeviceSynchronize();

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
```

```
  cudaFree(x);
  cudaFree(y);

  return 0;
}
```

Operatii atomice CUDA

CUDA ofera acces la multiple operatii atomice tip citire-modificare-scriere. Acestea presupun serializarea accesului in contextul mai multor thread-uri. Functiile sunt limitate la anumite tipuri de date:
- int
- unsigned int
- unsigned long long int
- float
- double

Exemple de functii atomice:
- atomicAdd [https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomicadd] - atomicSub [https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomicsub] - atomicExch [https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomicexch] - atomicMin [https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomicmin] - atomicMax [https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomicmax] - atomicInc [https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomicinc] - atomicDec [https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomicdec] - atomicAnd [https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomicand] - atomicOr [https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomicor]

A se consulta cu atentie documentatia CUDA inainte de folosirea unei operatii atomice (legat de contextul in care se aplica, cum opereaza, limitari etc).

In codul de mai jos se lanseaza un kernel concurrentRW in configuratie numBlocks=8 fiecare cu cate 10 thread-uri.

```
#include <iostream>

#define NUM_ELEM        8
#define NUM_THREADS     10

using namespace std;

__global__ void concurrentRW(int *data) {
...
}

int main(int argc, char *argv[]) {
    int* data = NULL;
    bool errorsDetected = false;

    cudaMallocManaged(&data, NUM_ELEM * sizeof(unsigned long long int));
    if (data == 0) {
        cout << "[HOST] Couldn't allocate memory\n";
        return 1;
    }

    // init all elements to 0
    cudaMemset(data, 0, NUM_ELEM);

    // launch kernel writes
    concurrentRW<<<NUM_ELEM, NUM_THREADS>>>(data);
    cudaDeviceSynchronize();
    if (cudaSuccess != cudaGetLastError()) {
        return 1;
    }

    for(int i = 0; i < NUM_ELEM; i++) {
        cout << i << ". " << data[i] << endl;
        if(data[i] != (NUM_THREADS * (NUM_THREADS - 1) / 2)) {
            errorsDetected = true;
        }
    }

    if(errorsDetected) {
        cout << "Errors detected" << endl;
    } else {
        cout << "OK" << endl;
    }
```

```
    return 0;
}
```

Functie concurrentRW citeste valoarea de la adresa data[blockIdx.x], o incrementeaza cu threadId si apoi o scrie.
In acest caz avem 10 thread-uri care fac operatii citire/scriere la aceeasi adresa, deci un comportament nedefinit.

```
__global__ void concurrentRW(int *data) {
    // NUM_THREADS try to read and write at same location
    data[blockIdx.x] = data[blockIdx.x] + threadIdx.x;
}
```

Exemplu rezultat:

```
0. 9
1. 9
2. 9
3. 9
4. 9
5. 9
6. 9
7. 9
Errors detected
```

Corect ar fi folosirea functie atomicAdd pentru a serializa accesul.

```
__global__ void concurrentRW(int *data) {
    // NUM_THREADS try to read and write at same location
    atomicAdd(&data[blockIdx.x], threadIdx.x);
}
```

Rezultatul rularii este:

```
0. 45
1. 45
2. 45
3. 45
4. 45
5. 45
6. 45
7. 45
OK
```

Operatii atomice system wide

Unitatile GPU ce au Compute capability 6.x permit largirea scopului operatiilor atomice. De exemplu atomicAdd_system garanteaza ca operatia este atomica cand atat thread-urile de pe unitatea GPU cat si cele de pe unitatea CPU incearca sa acceseze datele. Mai jos avem un exemplu de folosire al functiei atomicAdd_system.

```
__global__ void mykernel(int *addr) {
  atomicAdd_system(addr, 10);       // only available on devices with compute capability 6.x
}

void foo() {
  int *addr;
  cudaMallocManaged(&addr, 4);
  *addr = 0;

   mykernel<<<...>>>(addr);
   __sync_fetch_and_add(addr, 10);  // CPU atomic operation
}
```

Operatii asincrone CUDA

In CUDA, urmatoarele operatii sunt definite ca fiind independente si pot fi executate concurent:
- Calcule pe unitatea host
- Calcule pe unitatea device
- Transfer memorie host → device
- Transfer memorie device → host

- Transfer memorie device → device

Nivelul de concurenta o sa depinda si de capabilitatea unitatilor GPU (compute capability).
In continuare vom explora mai multe scenarii de executie concurenta a operatiilor descrise.

Executie asincrona Host si Device

Folosind apeluri asincrone, operatiile de executie catre device sunt puse in coada avand controlul intors catre host instant. Astfel unitatea host poate continua executia fara sa fie blocata in asteptarea executiei.
Urmatoarele operatii sunt asincrone relativ la host:

- Lansari de kernel
- Copieri in cadrul spatiului de memorie a unui device
- Copiere memorie host → device, avand < 64 KB
- Copiere memorie host → device, avand functii cu sufix Async
- Functii memorie set

Pentru a face debug unor scenarii de executie asincrona se poate dezactiva complet executia asincrona setand variabila de mediu CUDA_LAUNCH_BLOCKING la 1. Executia de kernels este sincrona cand se ruleaza cu un profiler (Nsight, Visual Profiler).

```
result = cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice, stream1)
```

Executie asincrona programe kernel

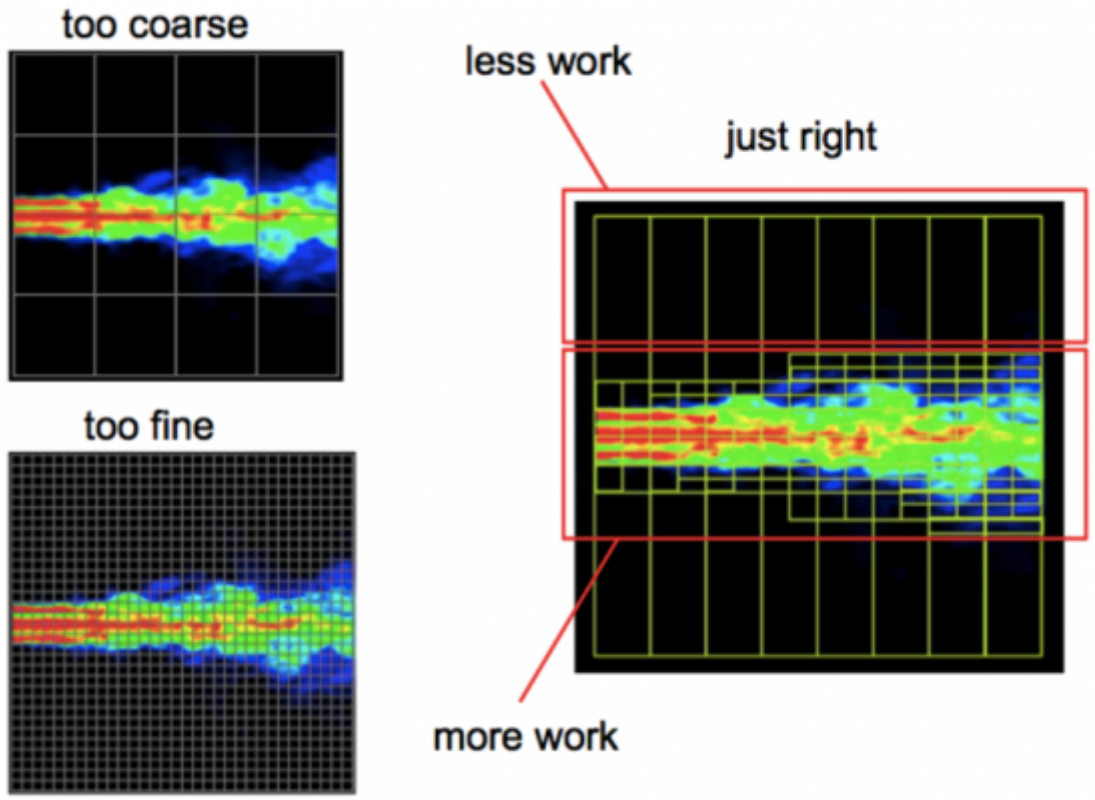
Arhitecturile cu compute capability 2.x sau mai nou, pot executa in paralel instante de kernel diferite. Aceste unitati de executie o sa aibe proprietate concurrentKernels setata la 1 (se face query la device properties inainte). Numarul maxim de lansari asincrone de kernele diferite este dependent de arhitectura (se verifica in functie de compute capability). Singura restrictie este ca programele kernel sa fie in acelasi context.

Executie si transfer date asincron

Anumite device-uri pot executa un transfer asincron memorie alaturi de o executie de kernel. Acest lucru este dependent de compute capability si se poate verifica in device property asyncEngineCount. Deasemenea se pot face transferuri de memorie intra-device simultan cu executia de kernel cand atat device property concurrentKernels si asyncEngineCount sunt 1.

Dynamic Paralellism

Paralelismul dinamic consta in posibilitatea de a lansa programe kernel din thread-urile ce ruleaza pe device/GPU. In alte cuvinte, unitatea GPU poate sa isi atribuie noi task-uri/thread-uri fara interventia unitatii host/CPU. Aceasta manifestare este utila in problemele unde maparea threaduri↔date nu este simpla/triviala. De exemplu, in situatia unde unele thread-uri ar avea prea putin de lucru, iar altele prea mult (imaginea de mai jos, simulare fluide) - o situatia debalansata computational.



Cerintele pentru paralelism dinamic sunt CUDA 5.0 ca Toolkit si respectiv Compute Capability 3.5. O lista cu GPU-uri NVIDIA si Compute Capability se regaseste aici [https://developer.nvidia.com/cuda-gpus]. Pana acum am lucrat pe coada hp-sl.q care are GPU-uri K40M (Compute Capability 3.5) si ibm-dp.q/ibm48-dp.q care are GPU-uri C2050 (Compute Capability 2.0). Astfel doar coada hp-sl.q suporta paralelism dinamic pe placile GPU K40M.

```
#include <stdio.h>
__global__ void childKernel()
{
   printf("Hello ");
}
__global__ void parentKernel()
{
   // launch child
   childKernel<<<1,1>>>();
   if (cudaSuccess != cudaGetLastError()) {
      return;
   }
   // wait for child to complete
   if (cudaSuccess != cudaDeviceSynchronize()) {
      return;
   }
   printf("World!\n");
}
int main(int argc, char *argv[])
{
   // launch parent
   parentKernel<<<1,1>>>();
   if (cudaSuccess != cudaGetLastError()) {
      return 1;
   }
   // wait for parent to complete
   if (cudaSuccess != cudaDeviceSynchronize()) {
      return 2;
   }
   return 0;
}
```

Exercitii

- logati-va pe `fep.grid.pub.ro` folosind contul de pe `cs.curs.pub.ro`
- executati comanda `wget http://cs.curs.pub.ro/wiki/asc/_media/asc:lab9:lab9_skl.tar.gz [http://cs.curs.pub.ro/wiki/asc/_media/asc:lab9:lab9_skl.tar.gz] -O lab9_skl.tar.gz`

- dezarhivati folosind comanda `tar -xzvf lab9_skl.tar.gz`
- executati comanda `qlogin -q hp-sl.q` pentru a intra pe o statie specializata in calcul folosind GPU-uri
- incarcati modulului de Nvidia CUDA folosind comanda `module load libraries/cuda`

Debug aplicatii CUDA aici [http://cs.curs.pub.ro/wiki/asc/asc:lab8:index#debug_aplicatii_cuda]
Profiling aplicatii CUDA via nvprof aici [https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview]

1. Deschideți fișierul task1_unified.cu și completati cu alocarile/trasferurile de memorie cerute. (3p)
    I. In functia compute_NoUnifiedMem se va aloca memorie cu cudaMalloc.
    II. In functia compute_UnifiedMem se va aloca memorie cu cudaMallocManaged.
    III. Folosind nvprof analizati cum ruleaza cele 2 programe:
        a. nvprof ./task1_mem_plain
        b. nvprof ./task1_mem_unified
2. Deschideti fisierul task2_atomic.cu si urmăriți instrucțiunile TODO (3p)
    I. Completati functiile kernel_no_atomics, kernel_partial_atomics si kernel_full_atomics.
    II. Folosind nvprof analizati cum ruleaza cele 3 programe:
        a. nvprof ./task2_no_atomic
        b. nvprof ./task2_partial_atomic
        c. nvprof ./task2_full_atomic
3. Deschideti fisierul task3_atomic_exch.cu si urmăriți instrucțiunile TODO (2p)
    I. Se da un vector cu sloturi ocupate (.raw!=0) sau libere (.raw=0) si un set de elemente de inserat.
4. Deschideti fisierul task4_dynp.cu si urmăriți instrucțiunile TODO (2p)
    I. Folosind dynamic parallelism se va calcula suma primelor data[i] elemente din vectorul data[]

Resurse

Schelet Laborator 9

Solutie Laborator 9

Enunt Laborator 9

- Responsabili laborator: Grigore Lupescu, Vlad Ştefănescu, Tudor Barbu, Mihai Despotovici

Referinte

- Documentatie CUDA:
    - CUDA C Programming [https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf]
    - CUDA NVCC compiler [https://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf]
    - CUDA Visual Profiler [https://docs.nvidia.com/cuda/profiler-users-guide/index.html]
    - CUDA 9.1 Toolkit [https://developer.download.nvidia.com/compute/cuda/9.1/Prod/docs/sidebar/CUDA_Toolkit_Release_Notes.pdf]
    - CUDA GPUs [https://developer.nvidia.com/cuda-gpus]
- Acceleratoare hp-sl.q (hpsl-wn01, hpsl-wn02, hpsl-wn03)
    - NVIDIA Tesla K40M [http://international.download.nvidia.com/tesla/pdf/tesla-k40-passive-board-spec.pdf]
    - NVIDIA Tesla [https://en.wikipedia.org/wiki/Nvidia_Tesla]
- Acceleratoare ibm-dp.q (dp-wn01, dp-wn02, dp-wn03)
    - NVIDIA Tesla C2070 [https://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf]
    - NVIDIA Tesla 2050/2070 [http://www.nvidia.com/docs/io/43395/nv_ds_tesla_c2050_c2070_apr10_final_lores.pdf]
    - NVIDIA CUDA Fermi/Tesla [https://cseweb.ucsd.edu/classes/fa12/cse141/pdf/09/GPU_Gahagan_FA12.pdf]
- Advanced CUDA
    - CUDA Streams [https://devblogs.nvidia.com/gpu-pro-tip-cuda-7-streams-simplify-concurrency/]
    - CUDA Dynamic Parallelism [https://devblogs.nvidia.com/introduction-cuda-dynamic-parallelism/]