Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών
Τμήμα Πληροφορικής και Τηλεπικοινωνιών

Parallel Systems

Assignment 4

Name: Giorgos Nikolaou **sdi:** 1115202000154

Name: Helen Fili **sdi:** 1115202100203

---

1 Description of the problem

An **n-body** simulator predicts the individual motions of a group of objects interacting with each other gravitationally in three-dimensional space. We have a a simple, though functional, CPU-only simulator for this purpose. In its current form, this application takes about 5 seconds to run on 4096 particles, and 20 minutes to run on 65536 particles. Our task is to GPU accelerate the program, retaining the correctness of the simulation.

2 Brief description of the solution

Our initial refactoring effort targeted the `bodyForce` function, prioritizing a straightforward acceleration approach. We restructured the function as a global one, enabling it to execute on the GPU and be called from the host side.

Grid-Stride Loops: When the number of data elements exceeds the number of threads in the grid, each thread is responsible for processing a specific subset of elements, with each one advancing by a certain "stride" value to access its assigned elements.

In which loop within the `bodyForce` function can we implement this strategy?

- The iterations of the **outer** for-loop are independent. The calculation remains consistent from one iteration to the next, thus the order of computation is irrelevant. This **can** be trivially *parallelized* across CUDA threads!
- The **inner** for-loop represents work that is not trivially independent across iterations, as each iteration contributes to the result accumulated from the previous one. Hence, distributing this workload across CUDA threads would **not** follow the same approach used for the outer for-loop.

Afterwards, we *parellelized* the for-loop inside `main` responsible for integrating the interbody forces calculated by `bodyForce` into the positions of the bodies in the system. This was achieved by moving the for-loop into a global function and applying the **grid-strided loop** technique.

The next step is to explicitly define the number of blocks and the number of threads per block. For utilization and performance efficiency purposes, we set the number of threads to 64, which is a **multiple** of the warp size (32). Furthermore, to achieve scalability, we set the number of blocks to 32 times the number of Streaming Multiprocessors and is retrieved dynamically at runtime. This approach ensures that the workload is evenly distributed across the available processing units, allowing for efficient utilization of the underlying hardware resources.

The final modification involved replacing on-demand data transfers with **asynchronous memory prefetching**. Specifically, the buffer required by the `bodyForce` function for calculations is asynchronously prefetched to the active GPU device prior launching the kernel, and at the end, the data are asynchronously prefetched back to the host. This reduces page faults and data migrations to and from the device, improving overall performance.

3 Presentation of experiments and analysis

3.1 Profiler and Assessment

Using the Nsight-Systems, we can get an in-depth dive into the performance of the CUDA Accelerated application. Specifically, we execute the following cell on the given notebook:

```
!nvcc -std=c++11 -o nbody 09-nbody/01-nbody.cu
!nsys profile --stats=true --force-overwrite=true -o report ./nbody 15
```

Listing 1: Matrix-Vector Multiplication Execution

CUDA API Statistics:

Time(%)	Total Time (ns)	Num Calls	Average	Minimum	Maximum	Name
63.9	169548117	1	169548117.0	169548117	169548117	cudaMallocManaged
35.7	94697279	20	4734864.0	4510	9469451	cudaDeviceSynchronize
0.3	670127	2	335063.5	281493	388634	cudaMemPrefetchAsync
0.1	263353	1	263353.0	263353	263353	cudaFree
0.1	247582	20	12379.1	5380	42251	cudaLaunchKernel

CUDA Kernel Statistics:

Time(%)	Total Time (ns)	Instances	Average	Minimum	Maximum	Name
100.0	94567095	10	9456709.5	9449705	9467689	bodyForce(Body*, float, int)
0.0	43650	10	4365.0	4320	4576	updatePos(Body*, float, int)

CUDA Memory Operation Statistics (by time):

Time(%)	Total Time (ns)	Operations	Average	Minimum	Maximum	Operation
50.7	120191	1	120191.0	120191	120191	[CUDA Unified Memory memcpy HtoD]
49.3	117055	1	117055.0	117055	117055	[CUDA Unified Memory memcpy DtoH]

CUDA Memory Operation Statistics (by size in KiB):

Total	Operations	Average	Minimum	Maximum	Operation
1536.000	1	1536.000	1536.000	1536.000	[CUDA Unified Memory memcpy DtoH]
1536.000	1	1536.000	1536.000	1536.000	[CUDA Unified Memory memcpy HtoD]

Figure 1: Nsight-Systems Results for $n = 65536$

We make the following observations based on Figure 1:

- Even with the additional overhead introduced by `nsys`, the `bodyForce` kernel, responsible for the bulk of the work, executes for approximately half of the total time compared to the non-`nsys` run of the executable. This suggests a significant and irreducible overhead during the allocation and handling of the required memory.
- Memory transfers have been successfully minimized to the essential, involving only one transfer from the host to the device and one from the device to the host.
- A notable amount of time is spent waiting on `cudaDeviceSynchronize`. This is anticipated, given that kernel launching is non-blocking, and the main thread (CPU) is in a waiting state from the start.

Below, we present the results of the assessment scenarios. The experiments were conducted in the NVIDIA-provided environment, with each scenario repeated nine times.

Size	CPU Time	GPU Time	Limit	Speedup
4096	5 sec	0.1198 sec	0.900 sec	41.7
65536	20 min	0.1786 sec	1.300 sec	6719

Table 1: N-Body Performance for different sizes

Clearly, the CUDA-accelerated version of the problem demonstrates an exponential increase in execution speed compared to its serialized counterpart. This effect becomes more pronounced as the size of the problem grows. Notably, with 65536 bodies, the speedup relative to the CPU version of the algorithm is nothing short of remarkable.

3.2 CUDA Accelerated Code

As evident, converting the `bodyForce` and position integration for-loop to kernels is a straightforward process, yielding substantial performance gains.

```
__global__ void bodyForce(Body *p, float dt, int n) {

  int index = threadIdx.x + blockIdx.x * blockDim.x;
  int stride = blockDim.x * gridDim.x;

  for (int i = index; i < n; i += stride) {
    float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;

    for (int j = 0; j < n; j++) {
      float dx = p[j].x - p[i].x;
      float dy = p[j].y - p[i].y;
      float dz = p[j].z - p[i].z;
      float distSqr = dx * dx + dy * dy + dz * dz + SOFTENING;
      float invDist = rsqrtf(distSqr);
      float invDist3 = invDist * invDist * invDist;

      Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;
    }

    p[i].vx += dt * Fx; p[i].vy += dt * Fy; p[i].vz += dt * Fz;
  }
}
```

Listing 2: Kernel with Grid-Strided Loop for Calculation of Forces

```
__global__ void updatePos(Body *p, float dt, int n) {

  int index = threadIdx.x + blockIdx.x * blockDim.x;
  int stride = blockDim.x * gridDim.x;

  for (int i = index; i < n; i += stride) { // integrate position
    p[i].x += p[i].vx * dt;
    p[i].y += p[i].vy * dt;
    p[i].z += p[i].vz * dt;
  }
}
```

Listing 3: Kernel with Grid-Strided Loop for Position Integration

Below, we present the initial part of the `main` function, where all the necessary variables are initialized. Take note of the initialization of `threadsPerBlock` and `numberOfBlocks` with the values mentioned earlier.

```
int main(const int argc, const char** argv) {

    int deviceId;
    int numberOfSMs;

    cudaGetDevice(&deviceId);
    cudaDeviceGetAttribute(&numberOfSMs,
                           cudaDevAttrMultiProcessorCount,
                           deviceId);

    size_t threadsPerBlock = 64;
    size_t numberOfBlocks = 32 * numberOfSMs;

    int nBodies = 2 << 11;
    if (argc > 1) nBodies = 2 << atoi(argv[1]);

    const char * initialized_values;
    const char * solution_values;

    if (nBodies == 2 << 11) {
        initialized_values = "09-nbody/files/initialized_4096";
        solution_values = "09-nbody/files/solution_4096";
    } else { // nBodies == 2 << 15
        initialized_values = "09-nbody/files/initialized_65536";
        solution_values = "09-nbody/files/solution_65536";
    }

    if (argc > 2) initialized_values = argv[2];
    if (argc > 3) solution_values = argv[3];

    const float dt = 0.01f; // Time step
    const int nIters = 10;  // Simulation iterations

    int bytes = nBodies * sizeof(Body);
    float *buf;
    double totalTime = 0.0;
```

Listing 4: Kernel with Grid-Strided Loop for Position Integration

Here, we present the final segment of the `main` function. We allocate the necessary memory, initialize the bodies using a predefined function, asynchronously initiate the memory transfer to the GPU, and execute the simulation. Following each kernel launch, a barrier is implemented to ensure all threads complete their respective tasks before progressing. Subsequently, we initiate asynchronous memory transfer back to the host device for output verification, and finally, we deallocate the used memory.

```
    cudaMallocManaged(&buf, bytes);

    Body *p = (Body*)buf;

    read_values_from_file(initialized_values, buf, bytes);

    cudaMemPrefetchAsync(buf, bytes, deviceId);

    for (int iter = 0; iter < nIters; iter++) {
        StartTimer();

        bodyForce<<<numberOfBlocks, threadsPerBlock>>>(p, dt, nBodies);
        cudaDeviceSynchronize();

        updatePos<<<numberOfBlocks, threadsPerBlock>>>(p, dt, nBodies);
        cudaDeviceSynchronize();

        const double tElapsed = GetTimer() / 1000.0;
        totalTime += tElapsed;
    }

    double avgTime = totalTime / (double)(nIters);
    float bOpsPerSec = 1e-9 * nBodies * nBodies / avgTime;

    cudaMemPrefetchAsync(buf, bytes, cudaCpuDeviceId);

    write_values_to_file(solution_values, buf, bytes);

    printf("%0.3f Billion Interactions / second\n", bOpsPerSec);

    cudaFree(buf);
}
```

Listing 5: Kernel with Grid-Strided Loop for Position Integration