Report II

Large data structures
Memory Managed utility and how to protect
program from too large data

Paweł Lipiór, Michał Żoczek

October 30, 2019

1 How large data structures are handled

During computer calculations often deal with large data structures. Computer is a great tool to do these calculations but we have to remember that it also has limited hardware resources. During our test with `vectorAdd` sample we have encountered some of these problems. Firstly, vector length has been limited by `Int` maximum size so we have been forced to use `long unsigned Int`. In some cases we could be limited by system memory capacity but we have been using simple dynamically allocated table of floats so our resources were enought.

```
1 unsigned long long numElements = 1<<30;
2 unsigned long long size = numElements * sizeof(float);
```

However the most limited resource often is device memory capacity. In case we suspect data structure is too large to handle it whole at the same time in GPU memory we can split data into smaller ones and do calculations in steps. If we have access to Pascal or later GPU we can also use management memory to automate this process.

2 Using Managed Memory

While programming in CUDA we can we see that memory host and device management is tedious and in some cases hard to do. It takes time to write a right code, test and debug it. Fortunately it can be avoided to some extent by using unified memory – a memory address space that is accessible by both host and device. We can just replace all functions responsible for allocate host and device memory and copying data between these by on simple function `cudaMallocManaged`. Below there is comparison of these two methods.

```
// Allocate the host input vector A
   float *h_A = (float *)malloc(size);

   // Allocate the host input vector B
   float *h_B = (float *)malloc(size);

   // Allocate the host output vector C
   float *h_C = (float *)malloc(size);

    // Allocate the device input vector A
   float *d_A = NULL;

   // Allocate the device input vector B
   float *d_B = NULL;

   // Allocate the device output vector C
   float *d_C = NULL;

    // Copy the host input vectors A and B in host memory to the device
        input vectors in
   // device memory
   err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
   }
```

```
// Allocate the vector A
   float *A;
   checkCudaErrors(cudaMallocManaged(&A, size));

   // Allocate the vector B
   float *B;
   checkCudaErrors(cudaMallocManaged(&B, size));

   // Allocate the vector C
   float *C;
   checkCudaErrors(cudaMallocManaged(&C, size));
```

This way it is possible to simplify the code and avoid mistakes and reduce time needed to release final and fully-featured calculation tool. On the other hand `cudaMallocManaged` can not be smarter than the programmer and we have to take care that all process are fine. E.g we can not allocate more memory than the device hardware has.

3 Protecting program from too large data

Another troublesome aspect of working with GPU is managing resources, especially memory. While using managed memory makes it easier, it does not mean that it takes care of everything. One such aspect that is left for programmer is to make sure that data used during kernel execution will not exceed available

GPU memory. In program used for this exercise it was achieved by first checking GPU global memory using `cuDeviceGetProperties` function and then comparing obtained number of bytes with size of data.

```
if(deviceProp.totalGlobalMem < size)
    {
      fprintf(stderr, "Data size too large\n");
        exit(EXIT_FAILURE);
    }
```

Owing to these simply if statement there is no possibility to exceed the size of device memory. There is also limitation of number of blocks that can be used at once, but for modern GPUs its so big $(2^{31} - 1)$ that it is rarely a problem.