REPORT 2

Introduction to CUDA and OpenCL

AGH-Faculty of Physics and Applied Computer Science

authors:
Kinga Pyrek
Aleksandra Rolka

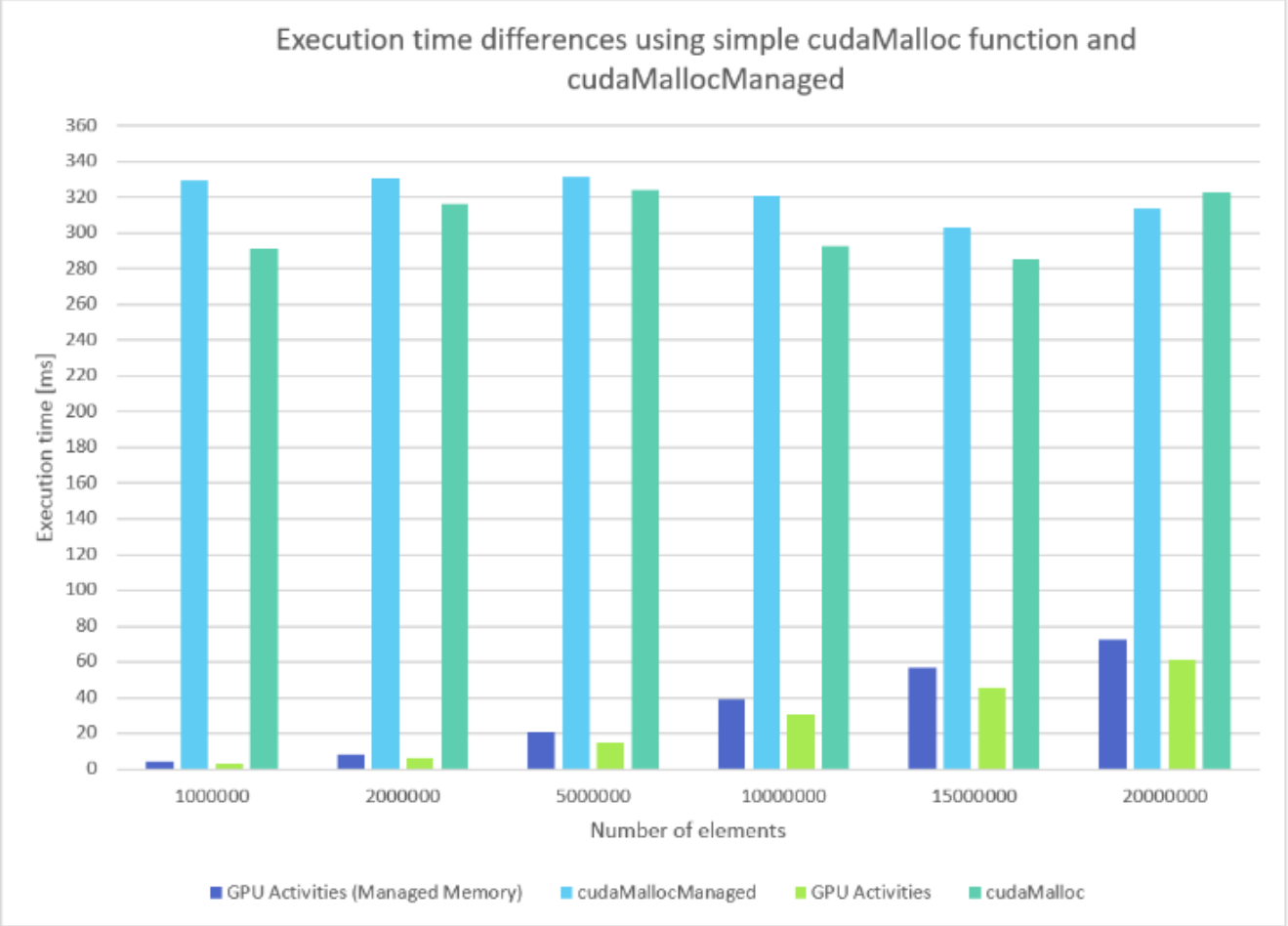
On the second laboratory classes we were talking about how to handle large data. First of all we learnt that CUDA employs a SIMT (Single Instruction Multiple Thread) architecture. It allows to group threads in warps and each warp consists of 32 threads. They all execute the same instruction at the same time. Being aware of the warp structure of our GPU we can manage the available resources in more efficient way.

We have been introduced to **cudaMallocManaged** function, which allocate memory on the device. It's a part of the Unified Memory system. We use it to simplify access to data on the CPU and GPU.

It's also easy to use cudaMallocManaged, because we replace : *malloc, cudaMalloc, cudaMemcpy, CUDA memcpy DtoH and CUDA memcpy HtoD* with this one function. But we need to remember about two things when we're modyfing our code:

- CudaMallocManaged doesn't free allocated memory, so we need to take care of it by using **cudaFree** function
- Normally kernel execution is asynchronous, so while the GPU device is executing our kernel, the CPU can continue to work on some other commands, instructions, etc., instead of waiting. However when we use this synchronization command: **cudaDeviceSynchronize** the CPU is instead forced to idle until all the GPU work has completed before doing anything else.

We changed our code from first lab classes (“vectorAdd”) to apply Managed Memory Utility. We played with the code by changing number of elements in the vector in orginal and our new version of code (with cudaMallocManaged and cudaFree). On this basis, we have analyzed execution time of each situation:



As we can see there is no big difference in execution time after we used function: cudaMallocManaged.

Results show slight diffrences in the time needed to perform our program. But this is not the reason we use cudaMallocManaged. Fast prototyping refers to the time it takes you to write the code not the speed of the code. So it’s very useful in cuda programming.

It’s better to have a simplier code than to bother about the memory while execution time is pretty the same.

Our last task was to make sure that we wouldn’t copy too much data to our GPU.
In the first place we have to check its size and how much memory we can actually put on GPU, because our GPU will always fail computing too large data.

To check it we used functions included in deviceQuery sample:

```
int dev, driverVersion = 0, runtimeVersion = 0;

  for (dev = 0; dev < deviceCount; ++dev) {
    cudaSetDevice(dev);
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, dev);

    printf("\nDevice %d: \"%s\"\n", dev, deviceProp.name);

    char msg[256];
#if defined(WIN32) || defined(_WIN32) || defined(WIN64) || defined(_WIN64)
    sprintf_s(msg, sizeof(msg),
              "  Total amount of global memory:                 %.0f MBytes "
              "(%llu bytes)\n",
              static_cast<float>(deviceProp.totalGlobalMem / 1048576.0f),
              (unsigned long long)deviceProp.totalGlobalMem);
#else
    snprintf(msg, sizeof(msg),
              "  Total amount of global memory:                 %.0f MBytes "
              "(%llu bytes)\n",
              static_cast<float>(deviceProp.totalGlobalMem / 1048576.0f),
              (unsigned long long)deviceProp.totalGlobalMem);
#endif
    printf("%s", msg);

    printf(" Maximum memory pitch that you can send to GPU:  %lu bytes. Please,
    do reckon with it.\n", deviceProp.memPitch);

    if(size==0 || size > deviceProp.memPitch)
    {
        fprintf(stderr, "Too much data\n");
        return 0;
    }
  }
```

This part of program inform us about capacity of GPU and stops the program, when we’re trying to use too large data. It prevents us from crushing the program.