1.Handling large data structures

During the classes we were trying to find the limit of the length of the vector, that we could have added.

At the beginning we were slowly increasing the length from 50 up to 1 000 000 000. The length of 10^9 was too large for our GPU. During the test we noticed, that the memory of our GPU we were using when we were adding vectors of length of 10^5, was around 1GB. Knowing that our GPU has 6GB of memory, we were able to add vectors of 500 millions elements.

In the next phase of the classes, basing on our device properties (size of the memory - 6 371 475 456 bytes), and the size of the float (4 bytes), and the knowledge, that we have to find memory, for three vectors, we calculated that 530 956 288 is the maximum length of our vector.

2.Memory management utility

Writing programs in CUDA requires thinking about memory management - allocating it in Device, Host and copying data between them due to the fact that they are physically different elements separated via PCIe.

However with usage of CUDA Unified Memory, instead of having two different pools of memory - accessed separately by System and Device, we create single memory address space which is shared between both CPU and GPU. Migration of data is done automatically when it is needed. What is worth noting, that the problem with allocating and copying is not gone entirely, it is just hidden and performed by the utility. With Pascal architecture Unified Memory has hardware support for virtual memory page faulting and migration with Page Migration engine.

The main reason why do we use Unified Memory is simplicity. As almost the whole work regarding allocating data is done by the system, we do not have to worry that something might have not been done in this case properly. Also the code becomes easier to read as there is less code which allocates and moves data.

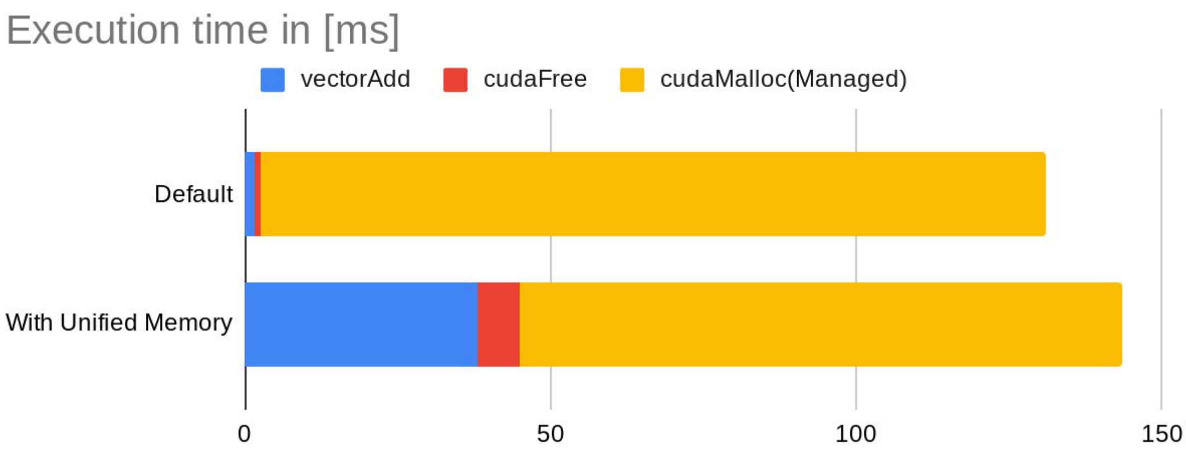
Another benefit of using Memory Management is performance. Thanks to migrating data on demand we can achieve similar or sometimes even better levels of speed compared to using separate memory pools. This is all thanks to complex operations of CUDA driver and runtime.

We have conducted a simple test by comparing vectorAdd sample program execution time with and without using Unified Memory. Both programs were run with same size of array and same grid configuration. We conducted three tests on each version to decrease the probability of random events. Results are presented in table below:

Table 1: Execution time per function

	vectorAdd	cudaFree	cudaMalloc(Managed)	total
Default	1.617ms	1.093ms	128.36ms	175.27ms
With Unified Memory	38.198ms	6.7055ms	98.506ms	178.07ms

On the base of the tables above we created charts to visualize the data.



If we want to look at the comparison quality-wise, we can spot clear differences, between using Unified Memory, and the default functions.

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   51.33%  8.6400us         2  4.3200us  4.3200us  4.3200us  [CUDA memcpy HtoD]
                   26.24%  4.4160us         1  4.4160us  4.4160us  4.4160us  vectorAdd(float const *, float const *, float*, int)
                   22.43%  3.7760us         1  3.7760us  3.7760us  3.7760us  [CUDA memcpy DtoH]
      API calls:   99.56%  265.78ms         3  88.593ms  6.9140us  265.76ms  cudaMalloc
                    0.18%  487.56us         1  487.56us  487.56us  487.56us  cuDeviceTotalMem
                    0.11%  297.59us        96  3.0990us     838ns  97.429us  cuDeviceGetAttribute
                    0.05%  132.14us         3  44.046us  6.9840us  107.63us  cudaFree
                    0.04%  119.22us         3  39.739us  21.092us  52.800us  cudaMemcpy
                    0.02%  65.931us         1  65.931us  65.931us  65.931us  cudaLaunchKernel
                    0.02%  42.743us         1  42.743us  42.743us  42.743us  cuDeviceGetName
                    0.00%  10.336us         1  10.336us  10.336us  10.336us  cuDeviceGetPCIBusId
                    0.00%  9.4290us         1  9.4290us  9.4290us  9.4290us  cudaDeviceSynchronize
                    0.00%  4.1900us         3  1.3960us  1.0470us  2.0950us  cuDeviceGetCount
                    0.00%  2.6540us         1  2.6540us  2.6540us  2.6540us  cudaGetLastError
                    0.00%  2.5840us         2  1.2920us  1.1170us  1.4670us  cuDeviceGet
                    0.00%  1.2570us         1  1.2570us  1.2570us  1.2570us  cuDeviceGetUuid
```

Profiling results - Default (not connected with the chart above)

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  955.66us         1  955.66us  955.66us  955.66us  vectorAdd(float const *, float const *, float*, int)
      API calls:   99.27%  310.01ms         3  103.34ms  9.1490us  309.95ms  cudaMallocManaged
                    0.31%  956.13us         1  956.13us  956.13us  956.13us  cudaDeviceSynchronize
                    0.15%  468.99us         1  468.99us  468.99us  468.99us  cuDeviceTotalMem
                    0.09%  290.47us        96  3.0250us     838ns  94.146us  cuDeviceGetAttribute
                    0.07%  203.80us         1  203.80us  203.80us  203.80us  cudaGetDeviceProperties
                    0.05%  153.02us         3  51.007us  17.810us  88.070us  cudaFree
                    0.04%  115.94us         1  115.94us  115.94us  115.94us  cudaLaunchKernel
                    0.01%  39.181us         1  39.181us  39.181us  39.181us  cuDeviceGetName
                    0.01%  28.845us         1  28.845us  28.845us  28.845us  cudaSetDevice
                    0.00%  10.266us         1  10.266us  10.266us  10.266us  cuDeviceGetPCIBusId
                    0.00%  4.4700us         1  4.4700us  4.4700us  4.4700us  cudaGetDeviceCount
                    0.00%  3.8410us         3  1.2800us     978ns  1.8850us  cuDeviceGetCount
                    0.00%  3.4220us         1  3.4220us  3.4220us  3.4220us  cudaGetLastError
                    0.00%  2.5140us         2  1.2570us     978ns  1.5360us  cuDeviceGet
                    0.00%  1.1180us         1  1.1180us  1.1180us  1.1180us  cuDeviceGetUuid

==40695== Unified Memory profiling result:
Device "GeForce GTX 1060 6GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
       4  32.000KB  12.000KB  52.000KB  128.0000KB  13.92000us  Host To Device
       5  25.600KB  4.0000KB  60.000KB  128.0000KB  12.80000us  Device To Host
```

Profiling results - Unified Memory (not connected with the chart above)

Processes distinctive for each way:

Unified Memory:	**Default:**
cudaMallocManaged	cudaMemcpy cudaMalloc CUDA memcpy DtoH CUDA memcpy HtoD

The difference is obvious after the explanation in the beginning of the second section.

Implementation of Unified Memory is fairly simple. Instead of using `malloc()` to allocate data, We use `cudaMallocManaged(void** ptr, size_t size)`, a function which returns pointer `ptr` which can be accessed by device or host. Since the memory is shared, we do not need to create different vectors for device and host. Furthermore we do not need to copy data, it means that any `cudaMemcpy` functions are no longer required. Using this method sparks another problem - synchronization. Accessing unprocessed data by host might be a major source of problems. Function `cudaDeviceSynchronize()` allows compute device to finish requested tasks by blocking host thread. After all the work is done we still need to free memory by using `cudaFree()`.

In the table below we have shown differences between both approaches of managing memory:

Table 2: Differences between code without and with Unified Memory.

Default	**With Unified Memory**
`float *h_A = (float *)malloc(size);`	`float *A; cudaMallocManaged(&A,numElements*sizeof(float));`
`cudaMalloc((void **)&d_A, size);`	--
--	`cudaDeviceSynchronize();`
`cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);`	--
`cudaMemcpy(h_A, d_A, size, cudaMemcpyDeviceToHost);`	--
`cudaFree(d_A);`	`cudaFree(A);`

Profiling program which uses Memory Management provides us with slightly more information. In the table there is a new section, where we can see the result of Unified Memory profiling.

We conducted a small test to get to know it a bit better.
Using previously mentioned programs, we added vectors, which had 10, 10^3, 10^5, and 10^7 elements. The profiling results are shown below.

```
==12591== Unified Memory profiling result:
Device "GeForce GTX 1060 6GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
       2  32.000KB  4.0000KB  60.000KB  64.00000KB  6.912000us  Host To Device
       2  32.000KB  4.0000KB  60.000KB  64.00000KB  6.112000us  Device To Host
       1         -         -         -           -  528.6080us  Gpu page fault groups
Total CPU Page faults: 2
==12902== NVPROF is profiling process 12902, command: ./Lab3-01
```

10-element vector

```
==12902== Unified Memory profiling result:
Device "GeForce GTX 1060 6GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
       2  32.000KB  4.0000KB  60.000KB  64.00000KB  7.040000us  Host To Device
       2  32.000KB  4.0000KB  60.000KB  64.00000KB  6.112000us  Device To Host
       1         -         -         -           -  543.3600us  Gpu page fault groups
Total CPU Page faults: 2
==13085== NVPROF is profiling process 13085, command: ./Lab3-01
```

1 000-element vector

```
==13085== Unified Memory profiling result:
Device "GeForce GTX 1060 6GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
      24  42.666KB  4.0000KB  244.00KB  1.000000MB  101.1200us  Host To Device
      21  97.523KB  4.0000KB  0.9961MB  2.000000MB  169.7920us  Device To Host
       7         -         -         -           -  1.365952ms  Gpu page fault groups
Total CPU Page faults: 16
==13257== NVPROF is profiling process 13257, command: ./Lab3-01
```

100 000-element vector

```
==13257== Unified Memory profiling result:
Device "GeForce GTX 1060 6GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
     625  125.00KB  4.0000KB  0.9883MB  76.29688MB  7.336480ms  Host To Device
     696  168.38KB  4.0000KB  0.9961MB  114.4453MB  9.628256ms  Device To Host
     317         -         -         -           -  36.50758ms  Gpu page fault groups
Total CPU Page faults: 580
```

10 000 000-element vector

We can clearly see that for smaller amounts of data (10, 10^3 elements), Unified Memory utility uses the same amount of memory and virtually the same amount of time to transfer data.
However for larger vectors, we can see almost proportional growth of memory usage (between 10^5 and 10^7-element vectors), and significant increase in time consumption. Next to the “total size”, and “total time”, we can see the “min size”, and “max size” columns. We assume that these refer to the sizes of blocks of memory our utility uses, thus we can see not directly proportional growth between 10^5 and 10^7 vectors in memory consumption.

3.How to prevent processing too large amount of data

Running GPU programs require knowledge about the hardware you are running on: the amount of threads and memory you have available. Without it we may try to do computing which is physically impossible. This might happen f.eg. when we exceed total available memory.

In out vectorAdd code, trying to add vectors which in total have over 6GB of global memory will result in a crash, the program will simply stop responding. As programmers we know our limitations and what will happen when we fetch too much data, so naturally we avoid them. But the end user might not. In this case we have to introduce an algorithm which will prevent overloading the memory.

Firstly we gather information about hardware we possess. With DeviceQuerry we get the number of GPU's and how much memory do they have. In this example, knowledge that We have an addition of 1D arrays A+B=C, allows us to calculate how much elements each of them can consist of. Dividing total available memory by the number of vectors and size of data type (rounded down) results in the maximal size of arrays.

With that We create an if statement which checks the number of elements given by user. If it's larger than calculated limit, there are two ways of dealing with this problem. First one: we show message that memory will be overloaded and exit the program with `exit(EXIT_FAILURE)`. Another one is where we cap the vectors with maximal possible size, ignore the rest of data and print appropriate statement.

The solution We have chosen is presented in the fragment of code below:

```
       float totalmem=0;
       int deviceCount = 0;
       int dev;
       cudaGetDeviceCount(&deviceCount);
       for (dev = 0; dev < deviceCount; ++dev) {
           cudaSetDevice(dev);
           cudaDeviceProp deviceProp;
           cudaGetDeviceProperties(&deviceProp, dev);
           totalmem+=deviceProp.totalGlobalMem;
       }

       float memperarr=(int)(deviceCount*totalmem/12);
//…
     if(numElements>memperarr){
       printf("Size of array exceeds total device memory!\nMaximal number of
elements in array is %f",memperarr);
       printf("\nArray dimension will be changed to prevent crash\nWARNING! %f
elements will be lost!!\n\n",numElements-memperarr);
       numElements=memperarr;
```

```
}
```