Report

Managed memory: page faults and preallocation

Paweł Lipiór, Michał Żoczek

November 7, 2019

1 Page faults

Page fault occures when proces is trying to access memory page that is not currenty mapped. The system then allocates neccessary memory to proces (or kills it, if it is impossible). Despite its name it is not necessarily problem - on the contrary it is common occurence, especially in modern systems utilizing virtual memory. It is good practice to minimalize number of page faults though, as waiting for space allocation slows program execution. In CUDA starting from CUDA 9 and GPUs using Pascal architecture, page faults became an important aspect of using managed memory as starting from this GPU generation CUDA program tries to access data when it needs it, instead of copying all of it to memory from the start.

2 Page faults for CPU and GPU functions

To check how page faults' effect on performance of programs using CPU and/or GPU tests were ran using `nvprof` and `nvvp`. Results of these test were collected in Tab. 1.

Used processing unit	**Number of faults on CPU**	**Number of faults on GPU**	**Comments**
Only CPU	384	-	-
Only GPU	-	5555	-
First GPU then CPU	384	5132	-
First CPU then GPU	394	5642	Handling faults on CPU and GPU is done in parallel.

Table 1: Number of page faults during program execution

It is not hard to see that number of page faults on the GPU is much higher than on the CPU (despite shorter execution time) and thus has greater impact on performance.

3 NVIDIA Visual Profiler Tool

NVIDIA Visual Profiler is very useful tool to optimize your CUDA code by menage memory resources. `Nvvp` allows us to graphically show time used on allocate memory, copy data from host to device and vice versa and time of kernel execution. Also `nvvp` can show us Page Faults if they appear.

To present how it works we have used modified `vectorAdd` sample code:

- `vectorAdd_standard`
- `vectorAdd_prefetch_GPU`
- `vectorAdd_prefetch_GPUCPU_init_GPU`
- `vectorAdd_prefetch_GPU_init_GPU`

Results of all profiling are shown below.

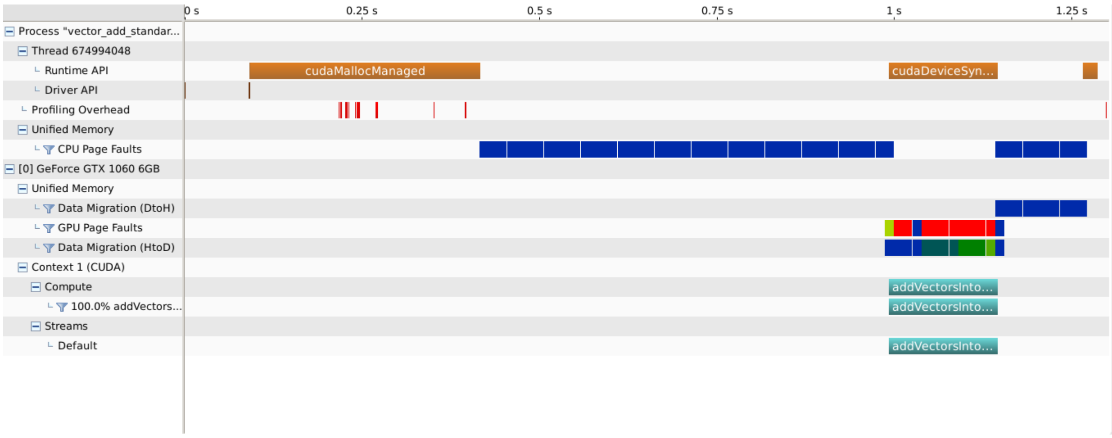

Figure 1: Profiling for vectorAdd_standard

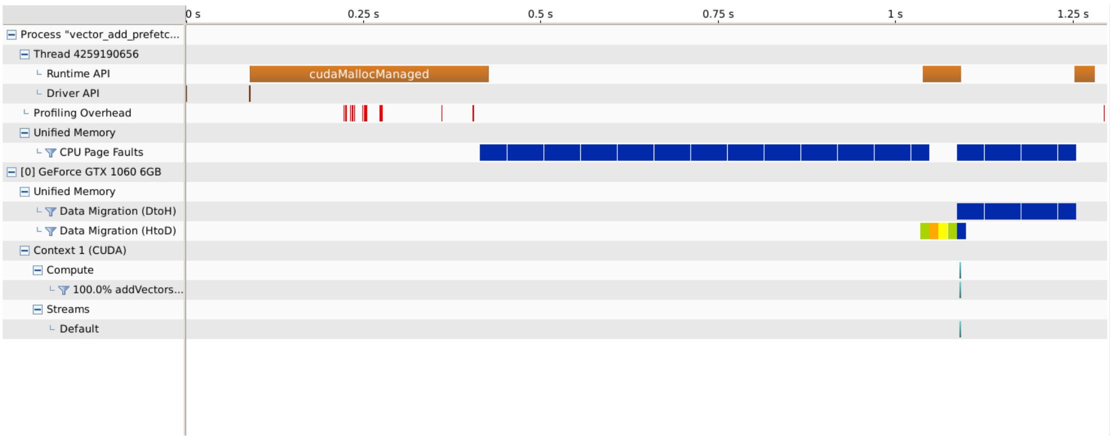

Figure 2: Profiling for vectorAdd_prefetch_GPU

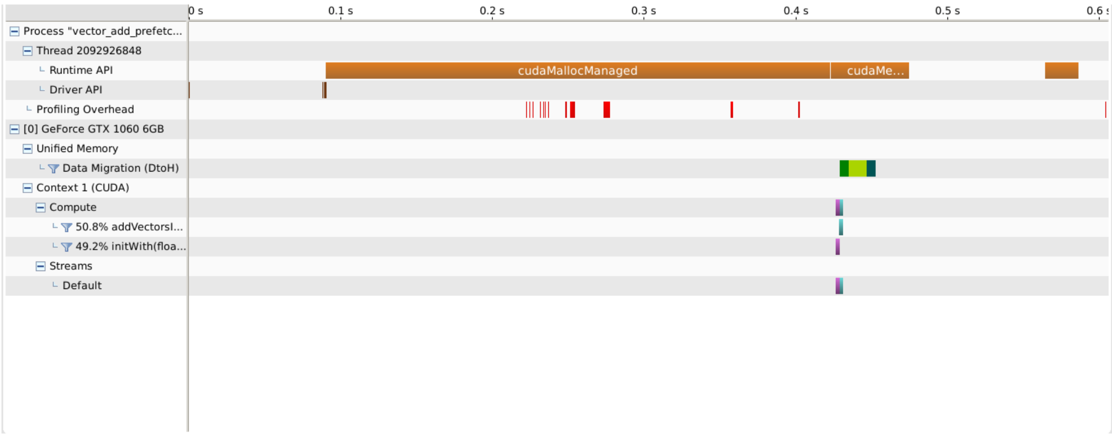

Figure 3: Profiling for vectorAdd_pretetch_GPUCPU_init_GPU

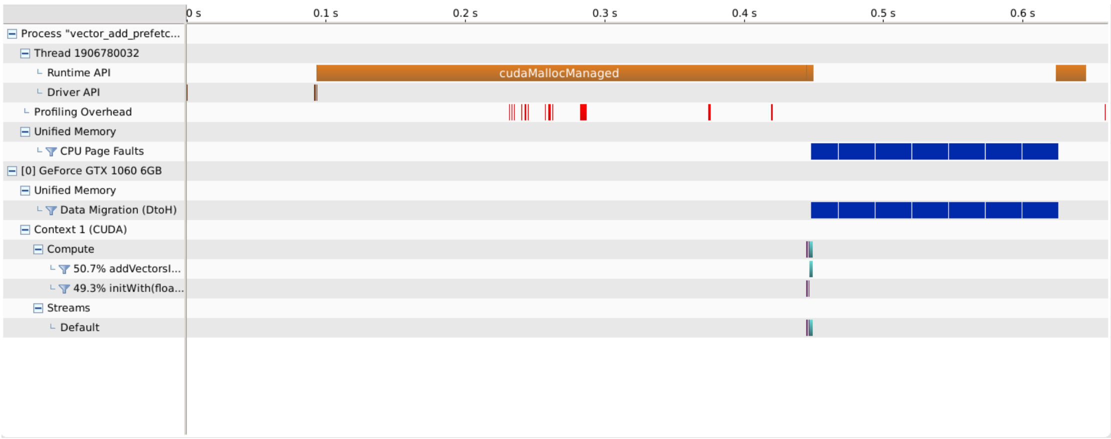

Figure 4: Profiling for vectorAdd_prefetch_GPU_init_GPU

Comparing those four results we can tell that there are some differences in time of execution each program. E.g. comparing 1 and 2 we can see that thanks to prefetching GPU we have avoided GPU page faults so time of copying data from device to host was a little bit less.

Next we can compare 1 and 4. We can see that thanks to initialize data on the GPU we can save some time on copying data from host to device but at a cost of time spend to initialize data on GPU.

Afterward we can compare 3 and 4. Both of them has data initialized on GPU but only one of them has CPU prefetching. This causes that in `vectorAdd_prefetch_GPUCPU_init_GPU` we can not see CPU page faults and that causes that we can save some time on copying data from device to host.

Summarizing using `nvvp` are relatively easy and it is useful tool if we try to improve performance of our CUDA calculation project. Thanks `nvvp` we can show how prefetching mechanism works and how it affects performance of our program. Moreover we was able to check if initializing data on GPU makes any sense.