1.Nvpp output analysis GPU/CPU/GPU+CPU/CPU+GPU page faults

Implementing Unified Memory in CUDA programs brings visible benefits in both simplicity and performance. One of the tools that make this possible is memory page faulting, which from the Pascal architecture is supported by hardware.

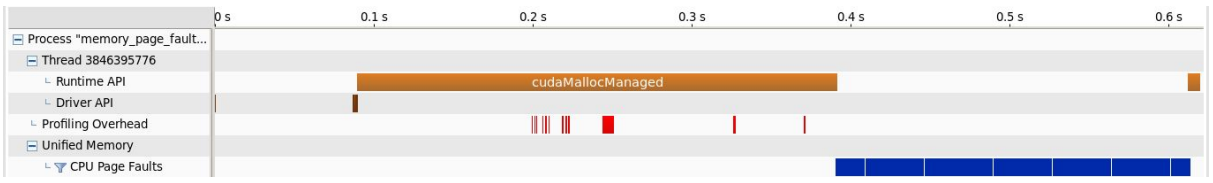
Page Fault happen when a process attempts to access a memory page, and that section is not currently available in the program's physical memory. Hardware then raises an exception which is handled by the Memory management unit. Handling usually relies on OS that makes that page accessible form virtual memory into the physical memory. Page faults can be divided into two categories: minor and major. The first one can occur when memory is shared between processes. If the page is loaded into memory but not marked as loaded, the MMU only needs to make it actually visible to the program, without any need to fetch data. Major PF happens when at the time of making the fault the page is not loaded and requested memory page needs to be fetched from secondary memory. It's a mechanism that allows programs to use more memory than is physically possible.

Page faults itself are not a dangerous phenomena, and they happen in most cases without any problems. However when the program tries to access a page which does not exist or memory which belongs to other process, the page fault handler will return an a segmentation fault, resulting in terminating the program.

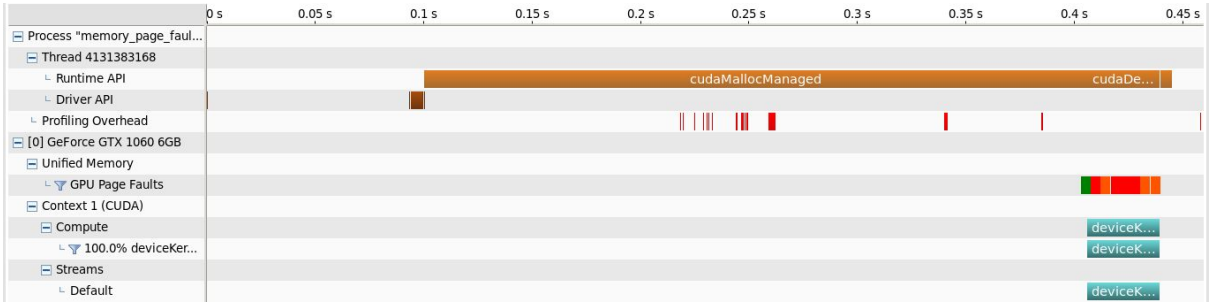
To understand the page faults we have used a simple code, consisting of device Kernel and host function, both of which initialize a given array. Our task was to analyze the output of the Nvidia Visual Profiler with different configurations of those two functions - executing with the hostFunction or Kernel alone, and when both of them are used in a different order.

With Visual Profiler we can clearly see the differences in each version. Let's discuss the case where only the one initialization function is used . The first thing that stands out in the CPU-only program is the duration of CPU page fault, taking approximately over 200ms. When we compare it to the version which utilizes kernel, we see huge improvement - here GPU page fault take roughly 50ms, giving almost 4x time decrease. The rest of elements remains roughly equal in terms of duration. Only difference is that with the GPU we need to make sure that memory is synchronised by using the device synchronise function. Output of the visual profiler is presented below:

CPU only

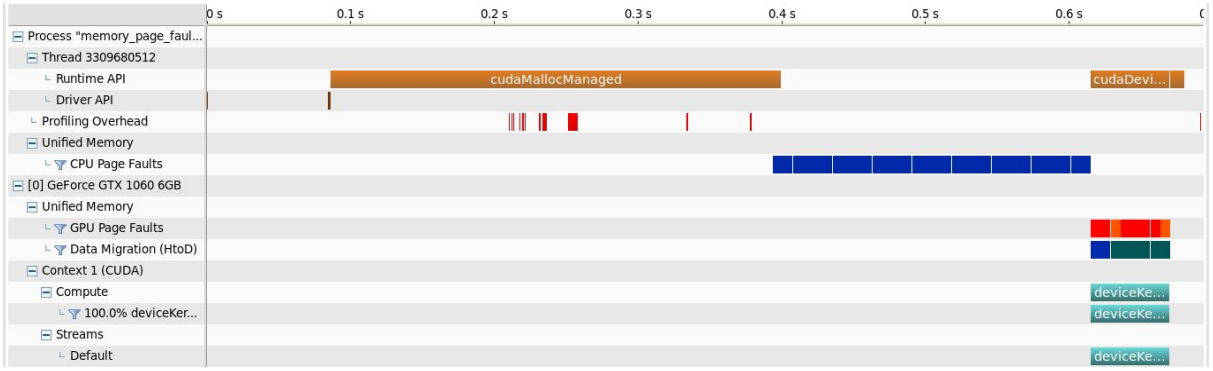


GPU only

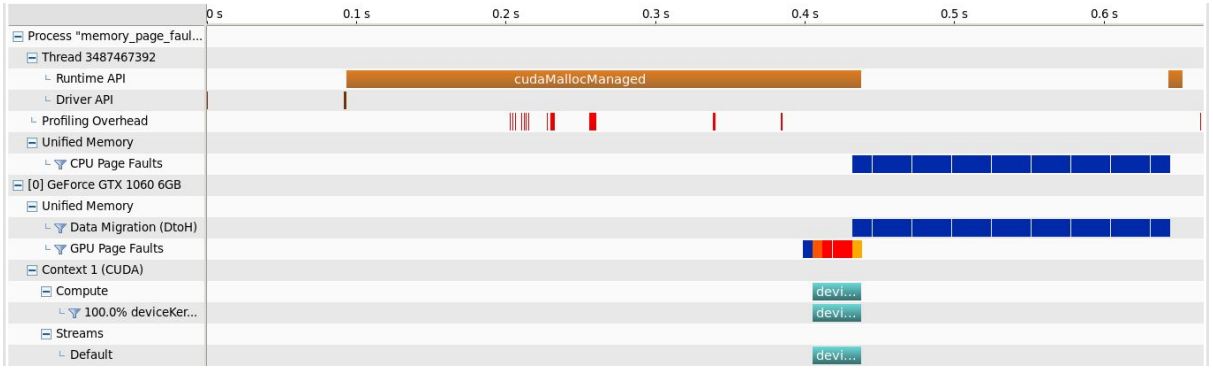


Differences start to appear when both functions are launched in program. All of the previous parts are still present, however because we use GPU and then CPU or CPU and then GPU the data needs to be migrated from host to device and vice versa. We see that page faults of the second process appear parallelly with data migration from hardware it was executed on. Output of the visual profiler is presented below:

CPU first



GPU first



There is noticeable difference it total elapsed time between those two versions. In case when kernel is the first process, the computation takes about 20 ms less time. Also, even when GPU is still processing array, the data migration from device begins. This might be due to the fact that we are not limited by the host memory throughput.

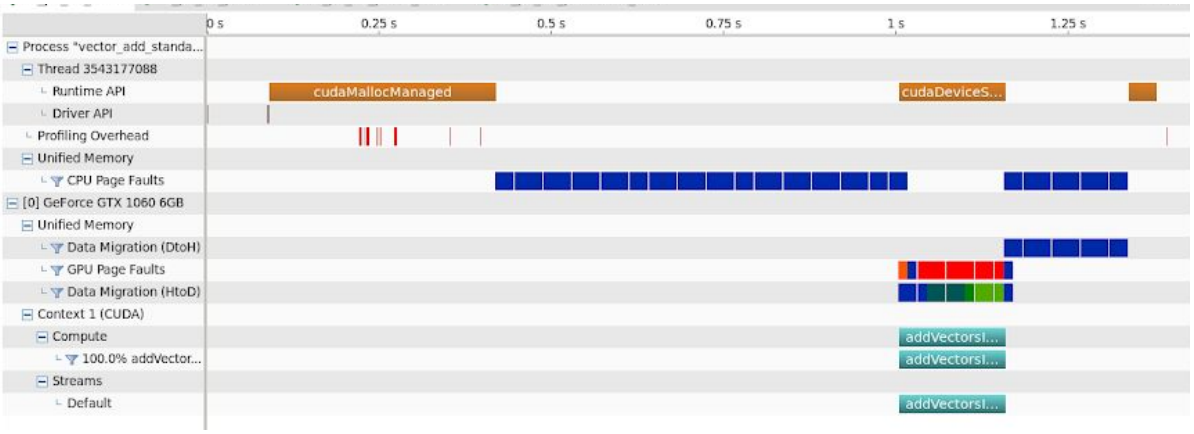
2. Memory management analysis with nvvp

In the second part of the classes we tried to further optimize our code.
We used a function *cudaMemPrefetchAsync* to prefetch data to speed up memory management, and we tried initializing data (in our case a vector) on the CPU and on the GPU as well.

cudaMemPrefetchAsync prefetches memory to the specified destination device from the base device memory.

First comparison:

1. Initializing with the **CPU**, **lack** of data prefetch
2. Initializing with the **CPU**, prefetching data to **GPU**



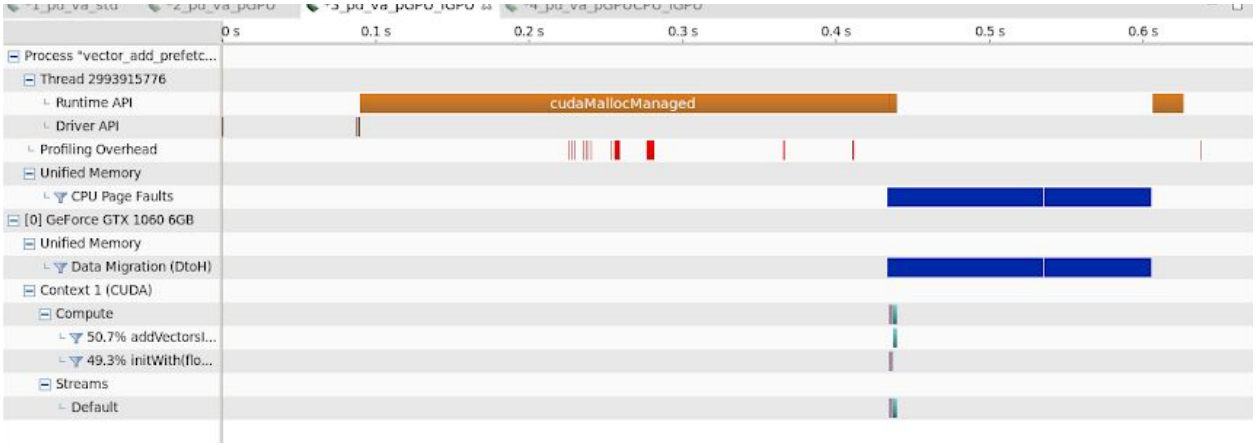
1 - Initializing with the **CPU**, **lack** of data prefetch

2 - Initializing with the **CPU**, prefetching data to **GPU**

The first point worth notifying, is that due to the usage of **cudaMemPrefetchAsync,** there is no GPU Page Faults registered in the second program. Moreover, the data migration from Host to Device takes significantly less time. We can also see a massive reduction of time consumed on *addVectorsInto* function (over 60 times less), and a small decrease of time consumed on the whole program.

Second comparison:

2. Initializing data with the **CPU**, prefetching data to **GPU**
3. Initializing data with the **GPU**, prefetching data to **GPU**

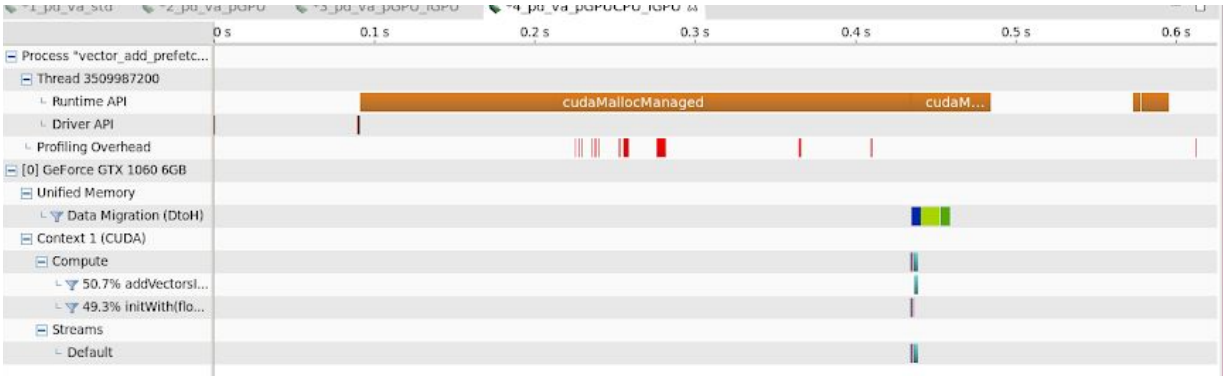


3 - Initializing data with the **GPU**, prefetching data to **GPU**

In comparison to the second program, we can see another decrease of time consumed by the program overall. Moreover due to the data initialization on the GPU there is no time consumed on HtoD data migration.

Third comparison:

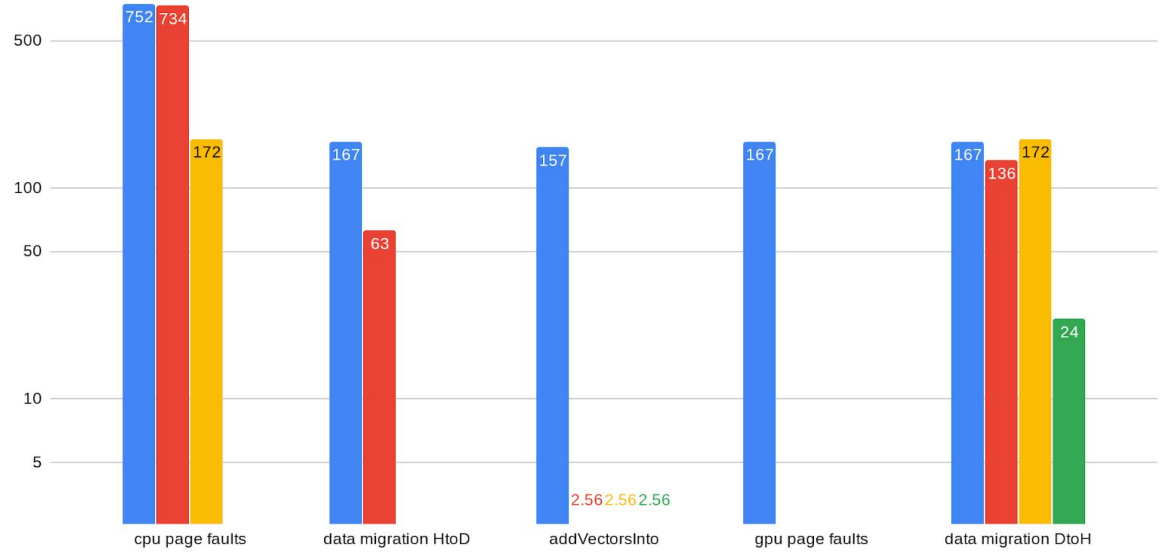
3. Initializing data with the **GPU**, prefetching data to **GPU**
4. Initializing data with the **GPU**, prefetching data to **GPU and CPU**



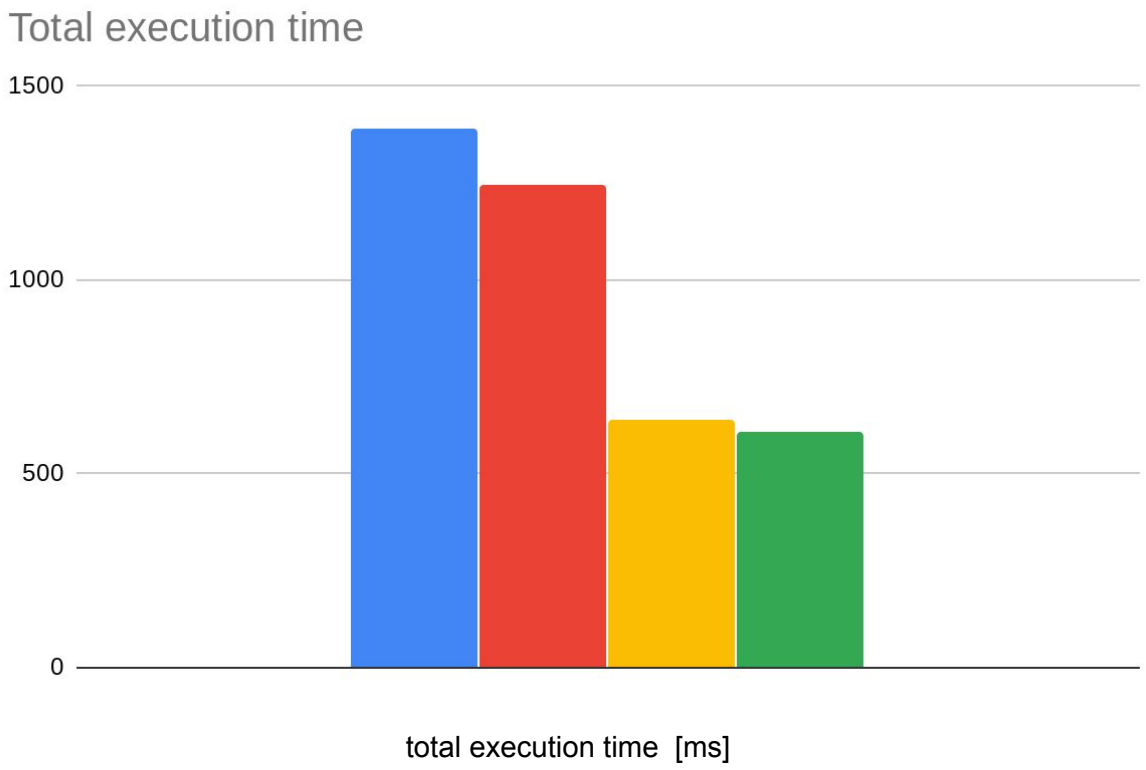
4 - Initializing data with the **GPU**, prefetching data to **GPU and CPU**

In comparison to the third iteration of our program, in fourth iteration we can see that for the first time we removed page faults on the CPU side. Moreover, we can see a huge difference in time consumed on data migration from Device to Host. There is also another improvement in overall performance.

Below, were placed charts, which depicts difference through all four iterations of our program. Iterations: 1st-blue, 2nd-red, 3rd-yellow, 4th-green.



time consumed on respective processes [ms]



total execution time [ms]

To sum up: Using *cudaMemPrefetchAsync* to prefetch data to GPU as well as CPU will increase performance of our program, and combining it with smart data initialization on GPU site, can optimize our code even more.