Introduction to CUDA and OpenCL

Ilona Tomkowicz, Zofia Pieńkowska

December 21, 2019

Contents

Contents **1**

1 Managed memory - various use cases. **2**

1.1 What happens when unified memory is accessed only by GPU? 2

1.2 What happens when unified memory is accessed only by CPU? 3

1.3 What happens when unified memory is accessed by GPU, then by CPU? 3

1.4 What happens when unified memory is accessed by CPU, then by GPU? 3

1.5 Conclusions 4

2 Memory management analysis with prefetching **4**

2.1 Managed memory with asynchronical prefetching. 4

2.2 Conclusions 5

1 Managed memory - various use cases.

To check how the managed memory work we used the code *vector_add_standard*. In the first step we extract the information about the number of streaming multiprocessors present on the device. The GPU we use has 10 SM which perform the actual computatuon, each of which schedules threads in groups of 32 threads (wraps - lab3). The grid has a layout of 256 threads per block and 320 blocks per grid. The kernel code makes the operation of adding two vectors of size 2^{24}. In this approach we use a unified memory, which creates pool of managed memory, that is shared between CPU and GPU. There is no need to copy vectors, as both device and host have access to them using the same memory address (one pointer pointing to the same adress value). Data copying is done automatically and does not require the programmer to allocate separate variables for host and device. This process is not separated on "host to device" and "device to host", but is listed as "cuda device synchronise".

The largest vector possible to allocate and process in this approach cannot be checked, because it is limited by the size of long int (2^{31} still gets processed in a relatively short time - 3us). What is interesting, runnign kernel in a loop for different values does not increase the summed time ideally linearly. E.g. for 5 repetitons it is 11us (should be 15), for 10 - 22us (not 30 as expected) and for 1000 - 2000 us. In comparison to the same version of code without use of smart pointers the performance is nearly the same. It is an expected result, the main difference is in the fact, that the code is neatly written and easier to read when we use managed memory.

1.1 What happens when unified memory is accessed only by GPU?

To test this case simple kernel and host functions, filling a vector with values was written. The kernel layout has 32 blocks and 1024 threads in each block. GPU computations took 28ms, moreover memory management (allocation and freeing) took respectively 289 ms and 33 ms. There are 415 page fault groups in total time of 31ms.

1.2 What happens when unified memory is accessed only by CPU?

In this situation the line measuring the kernel time performance disappeared, as expected. Memory management timing does not differ much from the version only on GPU as far as allocation is concerned, but freeing has decreased drastically by 80%. The number of page fault groups is not displayed, instead there is an information about CPU page faults (stating: 384) with no information about time.

1.3 What happens when unified memory is accessed by GPU, then by CPU?

Allocation and freeing of memory takes similiar amount of time as in the previous case. There are 778 page fault groups (from GPU) and 391 CPU page faults. Beyond it the information about CPU thrashes (16), the time of copying data from host to device (36us) and from device to host (10ms). Thrashing occurs when a computer's virtual memory resources are overused, leading to a constant state of paging and page faults. It means that page faults are the reason for thrashes. It is related to use of CPU, but it is also listed in the GPU activities. It means that page faults in both: device and host occure collaterally.

1.4 What happens when unified memory is accessed by CPU, then by GPU?

Allocation time is the same as in previous case, but freeing time is 58ms, which is the highest number in comparision to previous experiments. However, it is nearly twice as much as in the case of "only GPU" and over 8 times as much as in the remaining cases. It might be caused by the fact, that the last operation conducted on the data was an operation on the device side. It explains why times in 1st and 4th experimetn are similiar. Page faults also occure in this case, but they are not bound by thrashes value. It means, that faults happen independently, first on CPU, then on GPU. There is no information about data copying from device to host, so it does not take place. Data copying from host to device takes around 3 times longer than such a transfer in the previous experiment.

1.5 Conclusions

The only huge difference in above approaches is the page faults issue. Page faults for GPU and CPU are listed separetely. Generally page faults is an exception thrown by hardware in a stuation when a program wants to get access to a page that is not mapped. To handle the exception mapping should be provided, so that the page would become accessible at the location in physical memory on a condition, that the access was not illegal. To understand why in some cases page faults happen more frequently than in other we need to check how exactly does the access by the managed memory happen.

In a case, when the GPU accesses unified memory, first new pages of memory are allocated on GPU. Then pages on CPU are unmapped, because this connection won't be needed any longer. Data copy from CPU to GPU is done, so that new pages on GPU need to be mapped. After these operations pages on CPU can be freed. The same procedure, but in the opposite direction happens when the data allocated by managed memory mechanism are accessed by CPU.

Management process does not skipp data copying, but it hides it from the programmer. When we use both, host and device the data is still copied - pasted between their memories.

2 Memory management analysis with prefetching

2.1 Managed memory with asynchronical prefetching.

On-demand data transfer (tested in the previous paragraph) can slow down the overall execution of the program. Especially when the program deals with big data structures, because it needs to wait for that data to be sent. It creates a queue: wait for transfer - compute - wait for transfer - copute, so on. The stage: "wait for transfer" might be shortened even to zero in some cases if the data are sent in advance, before they are needed. Word async means that the transfer will occur in the most suitable time, when the device is not overloaded with work so the copying shouldn't block other operations.

To do so we use the line on a cudaMallcManaged ponter:

```
cudaMemPrefetchAsync(variable_name, size, deviceId);
```

after allocationg and initialising data, but before launching the kernel. The most essential operations listed in the profiler report are presented below. Performance with and without profiling (on-demand) is compared.

operation	**compu.**	**init.**	**p. faults**	**CPU p. faults**	**H2D**	**D2H**	**sync**
prefetching	2ms	-	1536	-	34ms	10ms	42ms
no prefetching	151ms	-	1536	1120	43ms	10ms	151ms

Table 1: Performance of program running on GPU with variables initialised on CPU

operation	**compu.**	**init.**	**p. faults**	**CPU p. faults**	**H2D**	**D2H**	**sync**
prefetching	2ms	2	384	-	-	10ms	5ms
no prefetching	2ms	80	384	1808	-	10ms	82ms

Table 2: Performance of program running on GPU with variables initialised on GPU

operation	**compu.**	**init.**	**p. faults**	**CPU p. faults**	**H2D**	**D2H**	**sync**
prefetching	2ms	2	-	-	-	-	5ms
no prefetching	2ms	82	1811	-	-	10ms	85ms

Table 3: Performance of program running on GPU and CPU with variables initialised on GPU

2.2 Conclusions

Prefetching results in decreasing page faults - no page faults on the CPU side occure when the number of page faults on GPU remains unchanged. Each page fault increases the driver's processing time, so the less of them occur, the better. Operations of initialisation variables on GPU take less time with prefetching. Also synchronisation of between device and host takes longer.

Unified memory is not the only approach in which prefetching can be used. There are also function equivalents for explicit data copying between host and device. Using *cudaMemcpyAsync* will do the same job in such a situation.