Introduction to CUDA and OpenCL
Lab 5 report

02.04.2020

Piotr Litwin
Paweł Skalny

Introduction

During fifth lab we focused on the memory. We took a look at page faults and the data prefetching technique. As it is all about performance we have taken a lot of measurements. Starting by analyzing samples from our common area, we observed the impact of data migration betwewn CPU and GPU on time of execution. Then we tested what would happen if we smartly mark the data which will be used before accessing it.

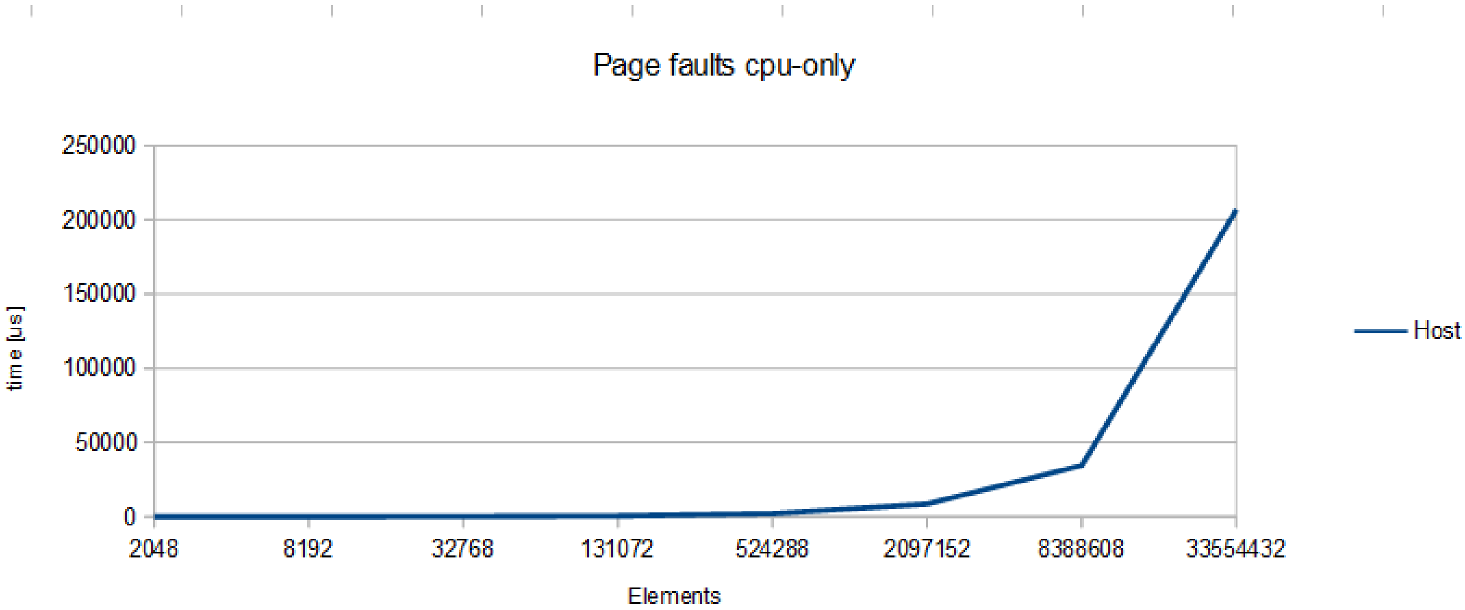
Results

We analyzed the time of execution for different data migrations and depending on the size of the input data. All measurements were made ten times. Below is the set of results.

CPU only

Elements	Host [us]
2048	51
8192	81
32768	163
131072	571
524288	2133
2097152	8692
8388608	34546
33554432	206843

Table 1. Time of execution using only CPU

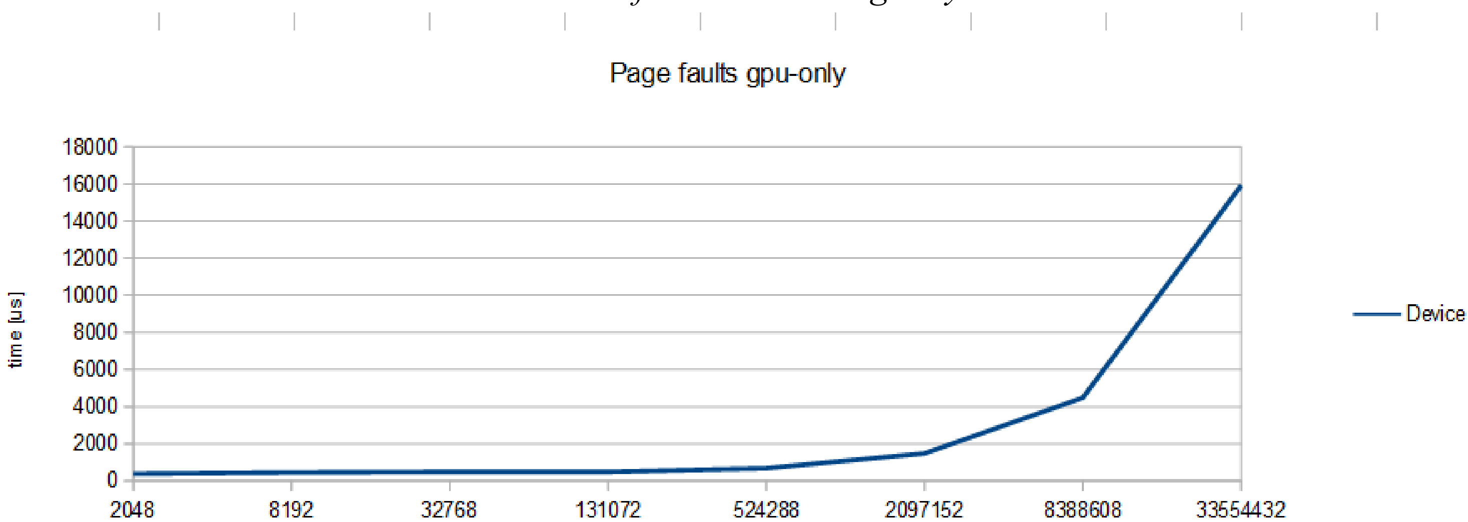


Pic. 1. Plot from Table 1.

GPU only

Elements	Device [us]
2048	378
8192	448
32768	478
131072	482
524288	679
2097152	1478
8388608	4482
33554432	15928

Table 2. Time of execution using only GPU

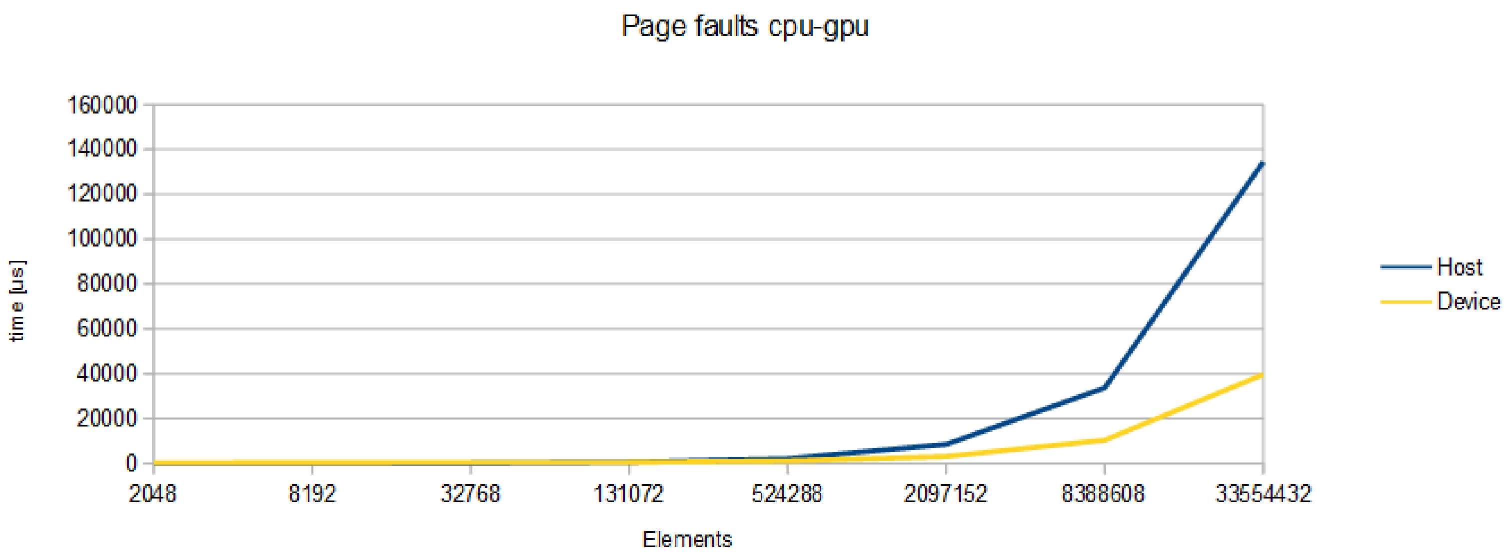


Pic. 2. Plot from Table 2.

CPU to GPU

Elements	Host [us]	Device [us]
2048	56	325
8192	88	401
32768	157	492
131072	563	594
524288	2147	1096
2097152	8536	3145
8388608	33732	10391
33554432	134468	39663

Table 3. Time of execution using CPU then GPU

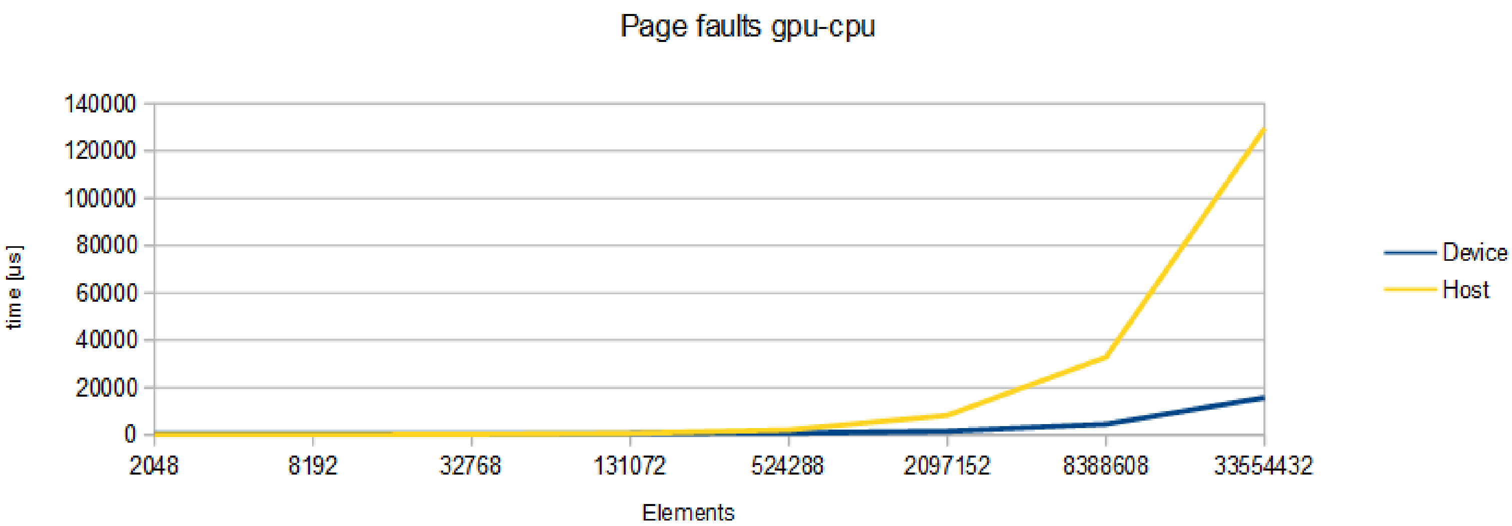


Pic. 3. Plot from Table 3.

GPU to CPU

Elements	Device [us]	Host [us]
2048	429	64
8192	446	69
32768	441	163
131072	532	594
524288	710	2081
2097152	1516	8244
8388608	4508	32850
33554432	15711	129634

Table 4. Time of execution using GPU then CPU



Pic. 4. Plot from Table 4.

We also measured number of page faults for both CPU and GPU:

Elements	CPU page faults	GPU page faults
2048	1	2
8192	1	1
32768	2	2
131072	4	5
524288	6	12
2097152	24	42
8388608	96	161
33554432	384	586

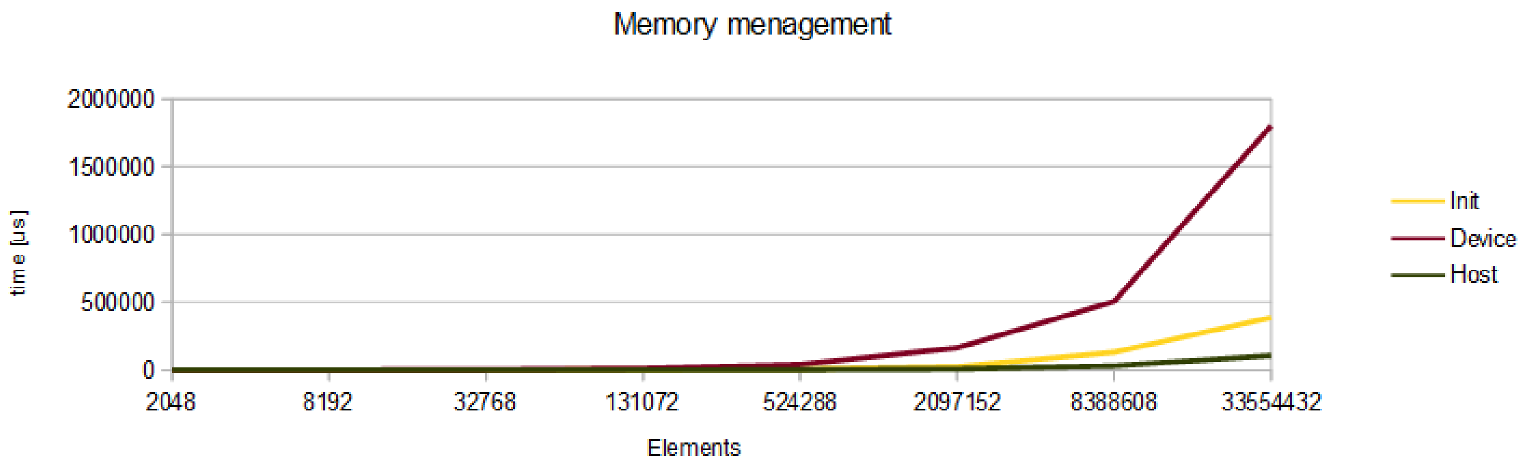
Table 5. Number of page fouls

Then we compared memory management to extension with prefetching:

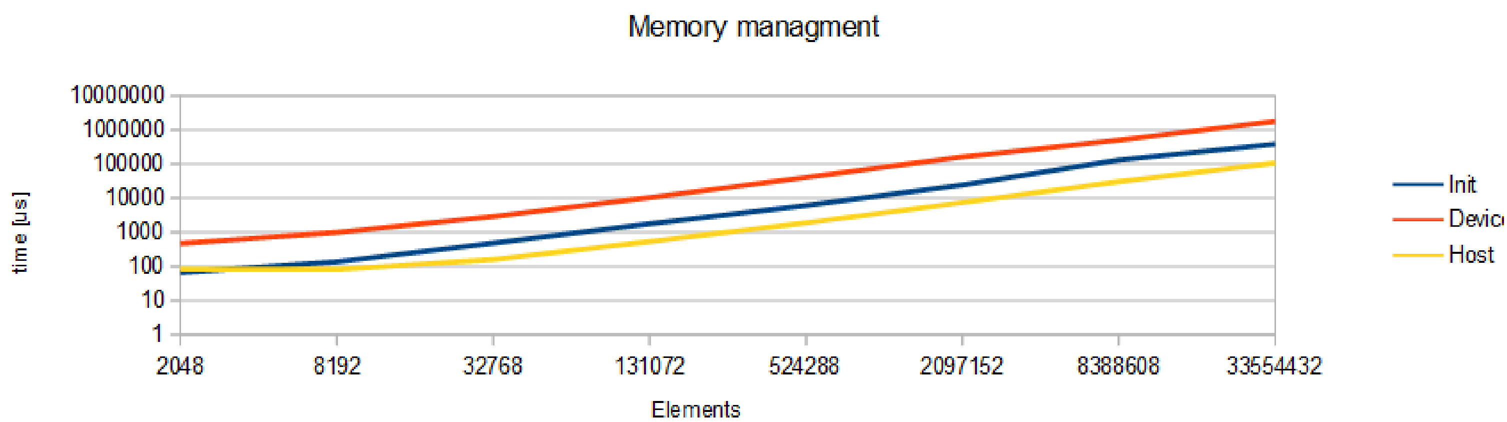
memory management

Elements	Init [us]	Device [us]	Host [us]
2048	67	472	83
8192	137	993	82
32768	483	2911	160
131072	1782	10468	538
524288	6127	40952	1900
2097152	24513	163044	7500
8388608	131429	505348	30709
33554432	387042	1801173	106841

Table 6. Time of execution using memory managment



Pic. 5. Plot from Table 6.

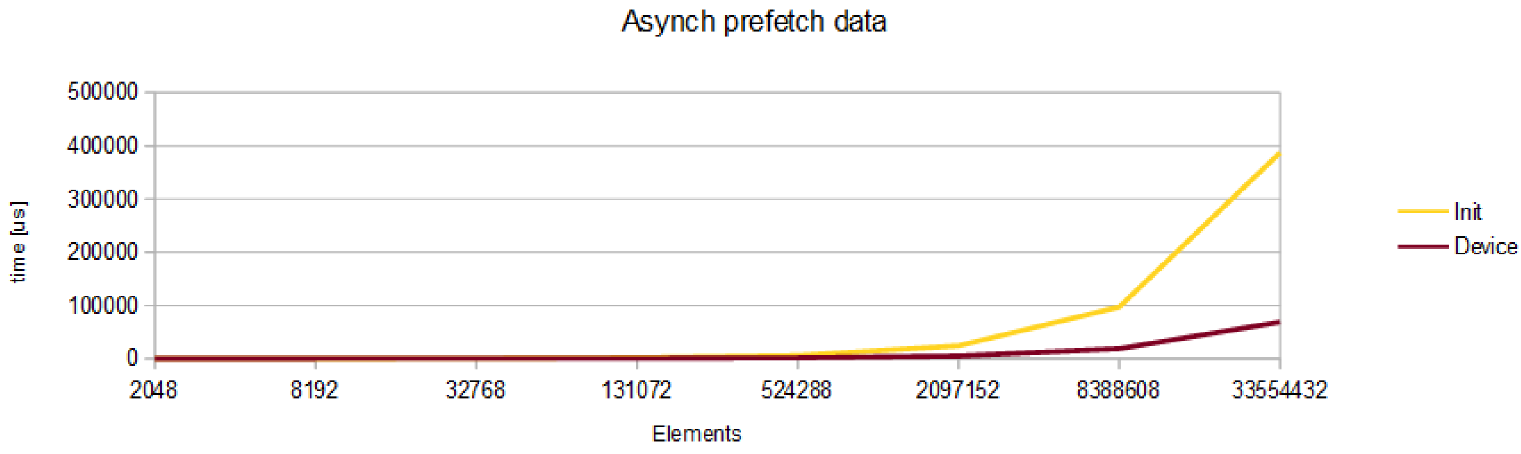


Pic. 6. Plot from Table 6. in logarithmic scale.

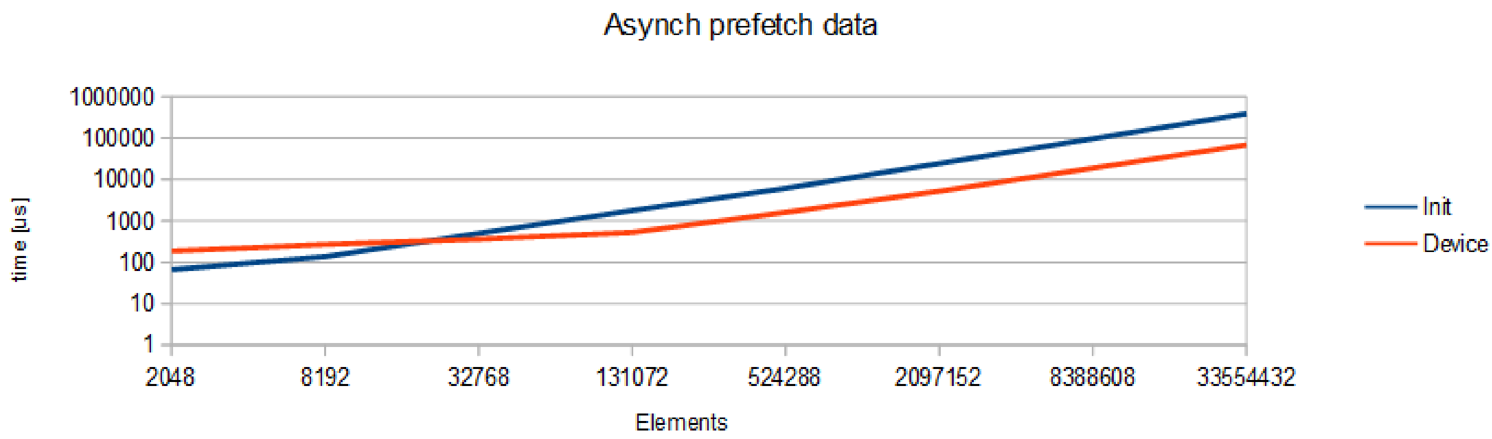
asynch prefetch data

Elements	Init [us]	Device [us]
2048	66	186
8192	136	268
32768	498	361
131072	1782	525
524288	6146	1615
2097152	24365	5213
8388608	97121	18903
33554432	387484	68800

Table 7. Time of execution using data prefetching for GPU



Pic. 7. Plot from Table 7.

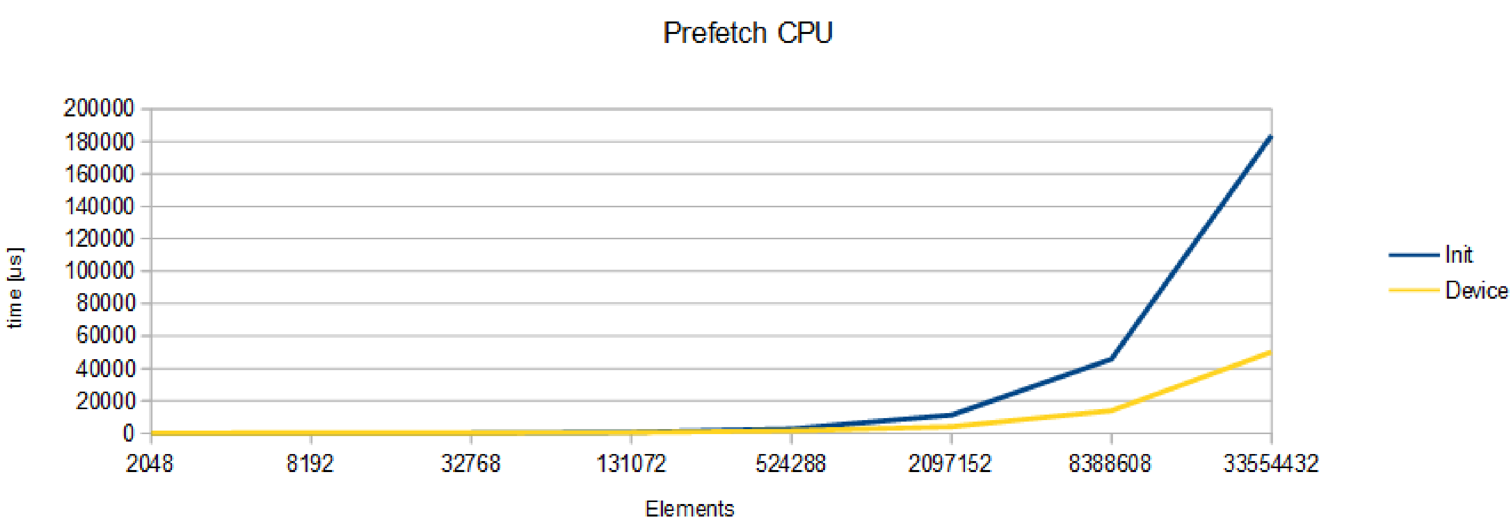


Pic. 8. Plot from Table 7. in logarithmic scale.

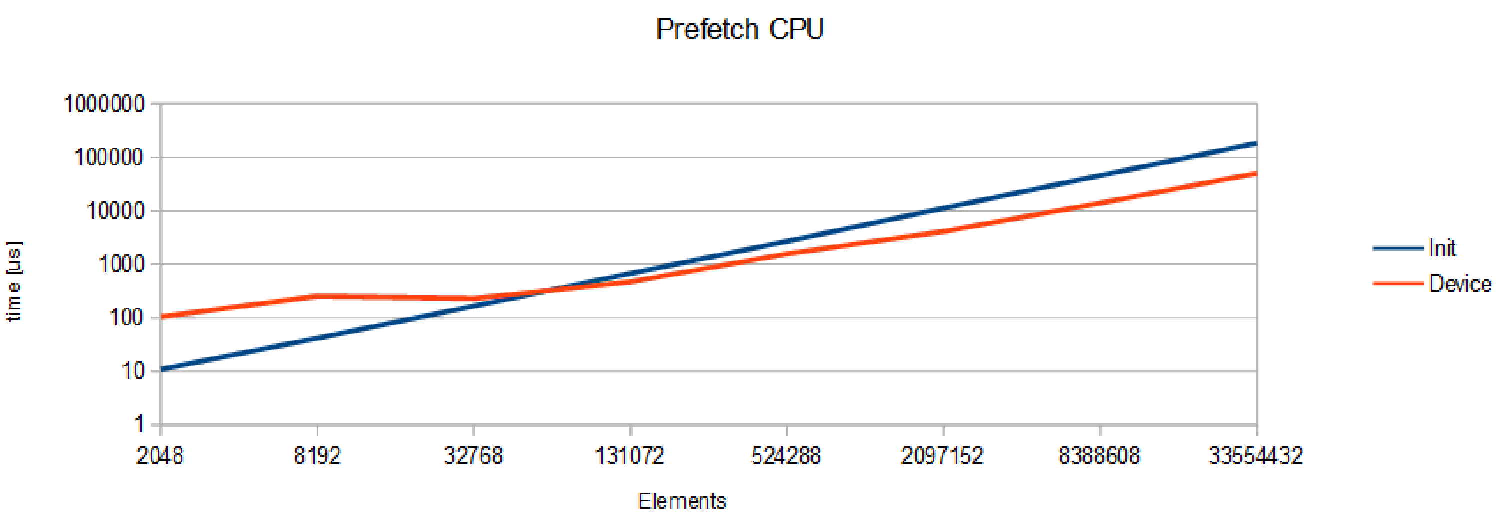
Finally, we can use prefetching for CPU as well.

Elements	Init [us]	Device [us]
2048	11	105
8192	42	254
32768	167	230
131072	675	472
524288	2704	1575
2097152	11257	4125
8388608	45827	13944
33554432	183635	50176

Table 8. Time of execution using data prefetching for GPU and CPU



Pic. 9. Plot from Table 8.



Pic. 10. Plot from Table 8. in logarithmic scale.

Conclusions

We observed, that memory managment is a good, easy to use tool. However using the memory declared in this way is generally slower *(Pic. 5. Pic. 7.)* and may cause page faults *(Table 5.)* depending on the direction of data migration. But there is a technique that combines the simplicity of managed memory and the speed of low level management. Prefetching informs that memory will be needed soon on the indicated device and starts migrating all data instead of small pieces each time kernel needs access to memory. In CUDA it works asynchronously. We can use prefetching for two ways from cpu to gpu and from gpu to cpu *(Pic. 9.)* which accelerates data reading as long as we exactly know what data will be using.