De La Salle University - Manila

Term 3, A.Y. **2022-2023**

In partial fulfillment

of the course

In **CEPARCO (S11)**

Execution Time Comparison: GPU-CPU Memory Transfer

Submitted by:

ALONZO, Jose Anton S.

AVELINO, Joris Gabriel L.

CRUZ, Airon John R.

HERNANDEZ, Pierre Vincent C.

June 5, 2023

I. Results

Table 1. Kernel Execution and Data Transfer Time for **Old CUDA Method**

Thread Count	256			512			1024		
Vector Size	**Host to Device**	**Device to Host**	**Kernel**	**Host to Device**	**Device to Host**	**Kernel**	**Host to Device**	**Device to Host**	**Kernel**
2^{20}	1.259ms	2.166ms	60.958us	919.42us	2.642ms	66.946us	939.13us	2.137ms	77.220us
2^{22}	4.206ms	13.49ms	229.49us	4.259ms	13.22ms	253.17us	4.313ms	13.74ms	294.00us
2^{24}	17.19ms	61.50ms	900.30us	17.15ms	61.76ms	997.24us	17.17ms	57.98ms	1.150ms

Table 2. Kernel Execution and Data Transfer Time for **Unified Memory Method**

Thread Count	256			512			1024		
Vector Size	**Host to Device**	**Device to Host**	**Kernel**	**Host to Device**	**Device to Host**	**Kernel**	**Host to Device**	**Device to Host**	**Kernel**
2^{20}	443.07us	724.25us	150.58us	506.01us	734.17us	152.84us	510.17us	725.82us	156.76us
2^{22}	1.85ms	2.91ms	629.49us	1.95ms	2.911ms	692.36us	2.09ms	2.90ms	619.93us
2^{24}	7.31ms	11.65ms	2.62ms	7.97ms	11.63ms	2.52ms	8.29ms	11.65ms	2.50ms

Table 3. Page Fault Information for **Unified Memory Method**

Thread Count	256			512			1024		
Vector Size	**CPU Page Fault**	**GPU Page Fault**	**GPU Page Fault Time**	**CPU Page Fault**	**GPU Page Fault**	**GPU Page Fault Time**	**CPU Page Fault**	**GPU Page Fault**	**GPU Page Fault Time**
2^{20}	36	19	2.75ms	36	17	2.53ms	36	16	2.30ms
2^{22}	144	81	11.34ms	144	72	16.17ms	144	64	12.84ms
2^{24}	576	333	48.62ms	576	299	54.08ms	576	258	41.56ms

Table 4. Kernel Execution and Data Transfer Time for **Data Prefetching with Memory Advising**

Thread Count	256			512			1024		
Vector Size	**Host to Device**	**Device to Host**	**Kernel**	**Host to Device**	**Device to Host**	**Kernel**	**Host to Device**	**Device to Host**	**Kernel**
2^{20}	349.1180us	322.203 us	61.159 us	348.6710us	322.218 us	67.834 us	349.4710us	322.186 us	77.791 us
2^{22}	1.399353ms	1.28898 ms	229.92 us	1.40523 ms	1.28815 ms	254.15 us	1.40322 ms	1.28853 ms	295.30 us
2^{24}	5.61819 ms	5.15311 ms	903.24us	5.61659 ms	5.1541 ms	1.0013 ms	5.6023 ms	5.15365 ms	1.1541 ms

Table 5. Kernel Execution and Data Transfer Time for **Data Transfer and Initialization as a CUDA Kernel Method**

Thread Count	256			512			1024		
Vector Size	**Host to Device**	**Device to Host**	**Kernel**	**Host to Device**	**Device to Host**	**Kernel**	**Host to Device**	**Device to Host**	**Kernel**
2^{20}	N/A	729.9490us	61.148us	N/A	723.1370us	67.226us	N/A	727.2310us	78.572us
2^{22}	N/A	2.917099ms	228.68us	N/A	2.902514ms	252.97us	N/A	2.897641ms	296.47us
2^{24}	N/A	11.60060ms	897.18us	N/A	11.70451ms	997.06us	N/A	11.67123ms	1.1662 ms

Table 6. Page Fault Information for **Data Transfer and Initialization as a CUDA Kernel Method**

Thread Count	256			512			1024		
Vector Size	**CPU Page Fault**	**GPU Page Fault**	**GPU Page Fault Time**	**CPU Page Fault**	**GPU Page Fault**	**GPU Page Fault Time**	**CPU Page Fault**	**GPU Page Fault**	**GPU Page Fault Time**

2^{20}	24	12	1.388216 ms	24	10	1.252215 ms	24	11	1.638263 ms
2^{22}	96	48	7.487443 ms	96	47	4.406403 ms	96	45	5.157349 ms
2^{24}	384	204	21.21692 ms	384	182	18.66946 ms	384	171	27.40591 ms

II. Analysis

It can be seen that the old CUDA (Table 1), data prefetching with memory advice (Table 4), and data transfer and initialization as a CUDA kernel methods' average execution time shows that it has a positive correlation with the number of threads and vector size. As for the Unified Memory method in Table 2, at 2^{20} vector size, its average execution time is positively correlated with the number of threads. While at 2^{24} vector size, its average execution time is negatively correlated with the number of threads. Only the 2^{22} vector size of Table 2 lacked a significant trend or correlation with the average execution time and number of threads. In general, however, it can be deduced that the CUDA kernel for performing 1D convolution, especially through the grid-stride loop implementation, is much faster and more efficient in execution in contrast to the C/C++ implementation of the function where execution times have reached thousands of microseconds as vector size increased.

A notable observation with regard to the data transfer time of the old CUDA memory transfer method as seen in Table 1, as well as that of the Unified Memory method in Table 2, is that it takes much longer to transfer data from device back to host in comparison to transferring data from the host to the device. This can be mainly attributed to the size of the data being transferred. This is more evident in the case of the Unified Memory transfer method. Upon inspecting the Unified Memory profiling result printed after execution, it can be seen that the total size being transferred from Device to Host is double the data size transferred from Host to Device. A concrete example of this can be displayed in Figure 1, wherein the Total Size of data transferred from Host to Device is 4 MB while the data transferred from Device to Host is 8 MB,

given the convolution kernel was executed on 2^20 elements in the vector using 256 threads. A similar pattern can be observed as the vector size and thread count varies.

```
==8580== Unified Memory profiling result:
Device "Tesla T4 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
      37  110.70KB  4.0000KB  984.00KB  4.000000MB  443.9950us  Host To Device
      48  170.67KB  4.0000KB  0.9961MB  8.000000MB  720.4710us  Device To Host
      19         -         -         -           -  2.725606ms  Gpu page fault groups
Total CPU Page faults: 36
```

Figure 1. Unified Memory profiling result for 1-D Convolution on 2^20 elements with 256 threads

As for the data transfer time of the data transfer and initialization as a CUDA kernel method. It didn't have a host to device transfer since the input and output vectors were initially placed in the GPU's physical memory due to it being initialized first in the kernel before the convolution kernel is executed. The transfer from device to host was caused by the error checking routine that accesses the input and output vector, which causes the pages of both vectors to be moved or transferred into the CPU's physical memory. In comparison to the other three approaches, the CUDA program with prefetching and memory advising demonstrated the fastest data transfer time in both directions: from host to device and vice versa.

In the case of the old CUDA memory transfer method, the simulation results show that neither CPU nor GPU page faults occurred during the execution of the program. With the programmer explicitly calling the cudaMemcpy() functions to transfer data between the GPU and the CPU, this ensures that both memory spaces of the two processors essentially possess their own copy of the data. Considering that this method entails non-unified memory, there is a prevention of any instance where either processor prematurely accesses a portion of the other's memory and subsequently requests data that is not initially present.

In the program with data prefetching and memory advising, `cudaMemPrefetchAsync` is used twice. First, it prefetches the 'in' array from GPU memory to CPU memory. Second, it prefetches the 'out' array from CPU memory to GPU memory. By performing these prefetch operations, the program ensures that the necessary data is already available in the desired memory location when it is accessed, minimizing page faults. Meanwhile, two memory advising calls are also made for the 'in' array. First, it advises that the 'in' array stays in the host memory using `cudaMemAdviseSetPreferredLocation`. Second, it advises that the `in` array is read-only using `cudaMemAdviseSetReadMostly`. These memory advising hints allow the GPU memory manager to optimize the memory allocation and access patterns, reducing the appearance of CPU and GPU page faults.

In the unified memory program without data prefetching and memory advising, it allows the CPU and GPU to share a single virtual memory address space, providing a unified view of memory. When unified memory is used without prefetching or memory advising, the memory manager employs a demand paging strategy. Specifically, demand paging means that the memory pages are transferred between the CPU and GPU only when necessary. If a page is accessed and is not currently present in the GPU memory, a page fault occurs. Therefore, since the program did not employ any explicit prefetching or memory advising hints, the memory manager relies on demand paging to transfer the required pages, which leads to occurrence of page faults— as reflected in **Table 3**. Moreover, without the implementation of prefetching or memory advising techniques within the program, the memory manager may need to perform frequent data migrations, which can introduce page faults as data is moved between CPU and GPU memory and vice versa.