GPU-CPU Memory Transfer Analysis - Group 2

- Steven John A. Pascaran
- Charlyne Arajoy Carabeo

Introduction

For our analysis to measure the performance of the different data transfer methods, we have used the percentage increase formula and also by getting the ratio using Unified Memory as the base below:

Percentage Increase Formula

$$\text{Percentage Increase} = \frac{\text{Final Value} - \text{Starting Value}}{|\text{Starting Value}|} \times 100$$

A. Unified memory introduced in CUDA 6

Analysis:

For the implementation of Unified memory we have seen how the larger input size results in longer execution time. However as we increase the number of threads we can see that the execution time decreases.

We can see the table below that shows the percentage of increase in runtime on the compute method as the number of threads increases using unified memory. For the input size 2^{20} and 2^{24} we can see that there has been a decrease in the execution time for them when we increased the number of threads from 256 to 512. We saw a decrease of 15.6526% for the input size 2^{20} and a decrease of 31.0614% for the input size 2^{24}. However, there was an increase in the execution time for both 2^{20} and 2^{24} when we increase the thread count further, from 512 to 1024. We can see that they have increased execution time by 0.00667289% and 16.1136% respectively. However for the input size 2^{22} we can see that as the number of threads increases the execution decreases. For the said input size we have seen a 7.40576% decrease in execution time when the thread count was raised to 512 and it was further decreased by another 3.59631% when we raised the thread count to 1024.

Percentage of Increase in runtime on compute method	2^{20}	2^{22}	2^{24}
256 to 512	-15.6526%	-7.40576%	-31.0614%
512 to 1024	0.00667289%	-3.59631%	16.1136%

It is also good to note that using unified memory alone results in 3 CPU Page faults.

B. **Prefetching of data with memory advice**

Analysis:
For our implementation of Prefetching of data with memory advice, we also used Unified Memory (cudaMallocManaged), Grid-Stride Loop, and prefetching, which all contributed to its better performance than with the other data transfer methods.

As we have observed in the Data table we acquired from nvprof, something that is notably different from Prefetching of data with memory advice from other data transfer methods, is that its execution time is considerably faster than the other 3 data transfer methods. We also observed that it has eliminated the page faults that are one of contributing factors of the execution time from the other data transfer methods such as Unified Memory and Data Transfer as a CUDA kernel. And that the Total Size for it is considerably larger than Unified Memory where the former has a total size of 4MB, 16MB, and 64MB depending on the number of inputs, and the latter having 64,000KB for Host to Device and 128,000KB for Device to Host.

We also noticed that as the number of threads increases, the execution time increases from around 9% up to 16% as seen in the table below. And that the execution time for data transfers between Host to Device and Device to Host remains consistent and close to the running time regardless of the number of threads and the size of the input.

Percentage of Increase in runtime on compute method	2^{20}	2^{22}	2^{24}
256 to 512	9.90911%	9.99779%	10.6712%
512 to 1024	14.2774%	16.1726%	10.6712%

As for the 2nd table below, we have compared the execution time of the compute method of the Unified Memory vs Prefetching with Memory Advise, and can see that the execution time improved by a lot from 1997.254% up to 3799.642%.

Percentage of Increase in runtime on compute method in contrast with Unified Memory using the ratio	2^{20}	2^{22}	2^{24}
256	2973.904%	3372.027%	3799.642%
512	2282.259%	2838.514%	2366.848%

1024	1997.254%	2355.488%	2381.152%

C. Data transfer or initialization as a CUDA kernel

Analysis:
In this method, we can see that using Data transfer/initialization as a CUDA kernel results in having 2 entries in the GPU activities 1 for the compute kernel and 1 for the transfer kernel. It can also be noted that increasing the thread count does not guarantee that it will run faster.

On the table below which shows the percentage of increase in runtime on compute method for data transfer or initialization as a CUDA kernel we can see that increasing threads from 256 to 512 has affected the execution significantly depending on the input size. For 2^{20} we saw a 8.2695% increase in the execution time meaning it ran slower compared to using 256 threads only. However, for the input size 2^{22} and 2^{24} we can see a significant decrease in the execution time meaning the method ran faster using 512 threads. The decrease in execution time is 10.2474% for the input size 2^{22}, and a 28.1466% decrease for 2^{24}. When we further increase the thread count from 512 to 1024 we can see a decrease in the execution time for input size 2^{20} and 2^{24} which is a decrease of 7.25586% and 12.0589% respectively. This translates that increasing the thread count from 512 to 1024 worked well for those input sizes as the execution time went down. However, for input size 2^{22} we can see that there has been a 27.4638% increase in the execution time, meaning it ran slower with more threads in this input size.

Percentage of Increase in runtime on compute method	2^{20}	2^{22}	2^{24}
256 to 512	8.2695%	-10.2474%	-38.1466%
512 to 1024	-7.24486%	27.4638%	-12.0589%

It is also good to note that using Data transfer of initialization as a CUDA kernel results in 2 page faults.

Old method of transferring data between CPU and memory (memCUDA copy)

Analysis:
In this data transfer method, we used Grid-Stride Loop and the old method of transferring which is the memCUDACopy. We also noticed that compared to Unified Memory, the old method doesn't have any page faults but it has an additional GPU activity which is the cudaMemCpy. And cudaMemcpy takes more time to execute compared to Unified Memory. And as seen in the

1st table, the execution time improves as the input becomes larger. But as the input size grows, the improvement when the threads increase is much lower.

Percentage of Increase in runtime on compute method	2^{20}	2^{22}	2^{24}
256 to 512	14.2063%	15.7166%	16.6121%
512 to 1024	14.0848%	13.9571%	12.9676%

It’s also good to note that even if the compute method is faster than compared to Unified Memory, the memCudaCpy takes a lot more time making it slower than with Unified Memory. And in our case we only did Device to Host and didn’t do any memCudaCpy for Host to Device.