Report Lab3: Programming a Real Processing-in-Memory Architecture

Introduction

This lab report is kept to a minimum due to tight time constraints.

Task 1

From the plots we see that DPU->CPU transfer has a much wider bandwidth then the CPU->DPU transfer. Further there are clear differences for the memory load. For a bigger input array we get a bigger bandwidth this makes sense as more data needs to be transmitted. From the plots no clear differenc between parallel, broadcast and serial transmission can be established. It cannot be concluded wether this is due to little performance difference between the three or due to an implementation error. The biggest eyecatcher in the plots is the jum at ~48 DPUs it my assumption is that this is due to a threshold in memory grouping being hit but agian it cannot be distinguished between an actual performance effect and an implementation error. Assuming that there is no implementation error we see that for 1 DPU the maximum CPU->DPU bandwidth is at ~13MB/s and the maximum DPU->CPU bandwidth at ~520 MB/s. For 64 DPUs the maximum bandwidth is at ~29 MB/s for CPU->DPU and ~80 MB/s for DPU->CPU.

Task 2

From the plot we can see that for increasing tasklets the number of instruction per tasklet decreases exponentially which seems to incentivise the usage of as many tasklets as possible as they allow for a higher degree of parallelisation. This is as expected as a similar effect can be seen when parallelising code on multiple cores.

Task 3

For this task we evaluate the operations addition(AXPY), subtraction(AXMINY), multiplication(AXMULY), and division(AXDIVY). From the plot we can see that for all non-floating point operations the addition and subtraction have a very similar performance and the multiplication and division have a very similar performance. This can be explained by the similarity of their implemntation in UPMEM logic. We can also see that the operations are optimised for 4 byte values such as INT32, as both SHORT and CHAR suffer from a decrease in performance due to being 2 byte and 1 byte instructions which are treated the same as 4 byte instructions. Therfore the cost for each individual entry is the same but for the smaller types we need to push more entries to achieve the same memory usage. On the other hand INT64 takes longer due to it being an 8 byte instruction which agian creates additional effort to handle. For FLOAT and DOUBLE we clearly see the impact of floating point operations which take longer as they require more involved logic. Further we can also see a clear spike in the floating point divisions. Which is natural as it is the most complicated floating point operation.

Task 4

For this task we compare the effect of different synchronization methods. The first observation we can make is that their relative performance degrades for increasing vector size, while their absolute performance differenc seems to stay at a very similar level. This makes sense as they only handle the last step of the vector reduction which is very similar for all vector sizes. We can see that in general buth MUTEX and ZERO_TASKLET_COLLECTION outperform their peers. This is due to them implementing simpler exectuion schemes which need a lower amount of instructions.

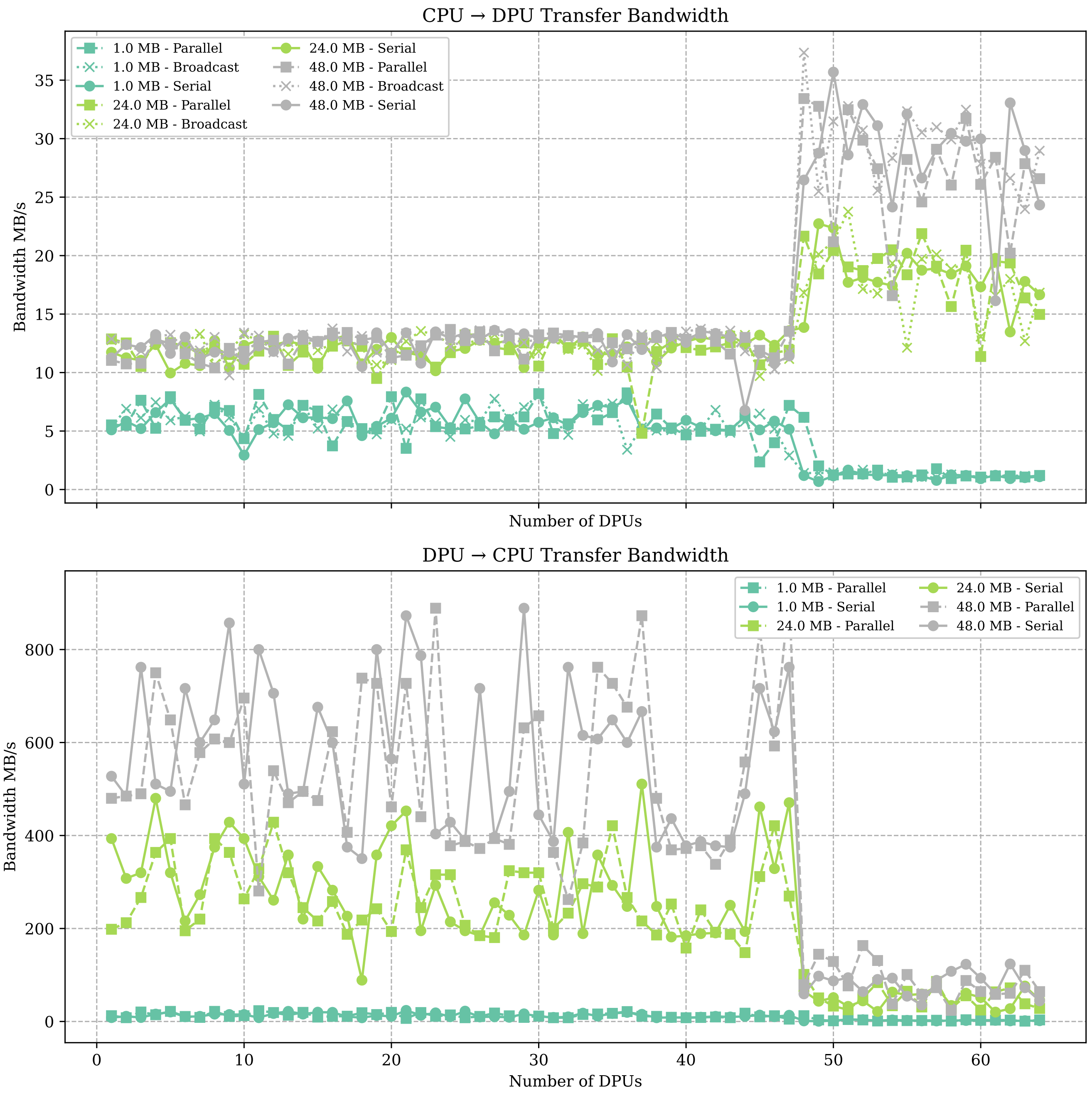

Figure 1: Task1_plot

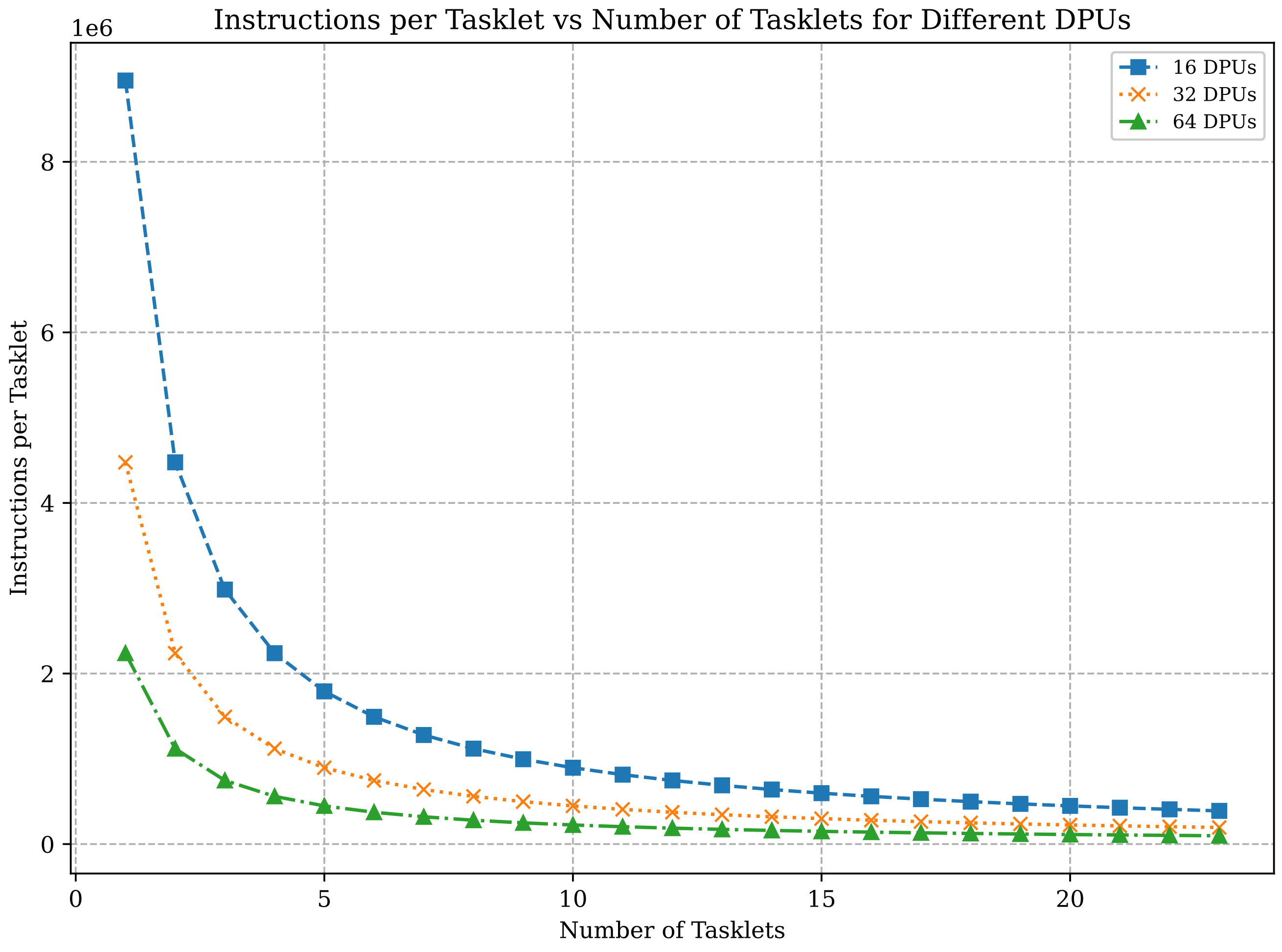

Figure 2: Task2__plot

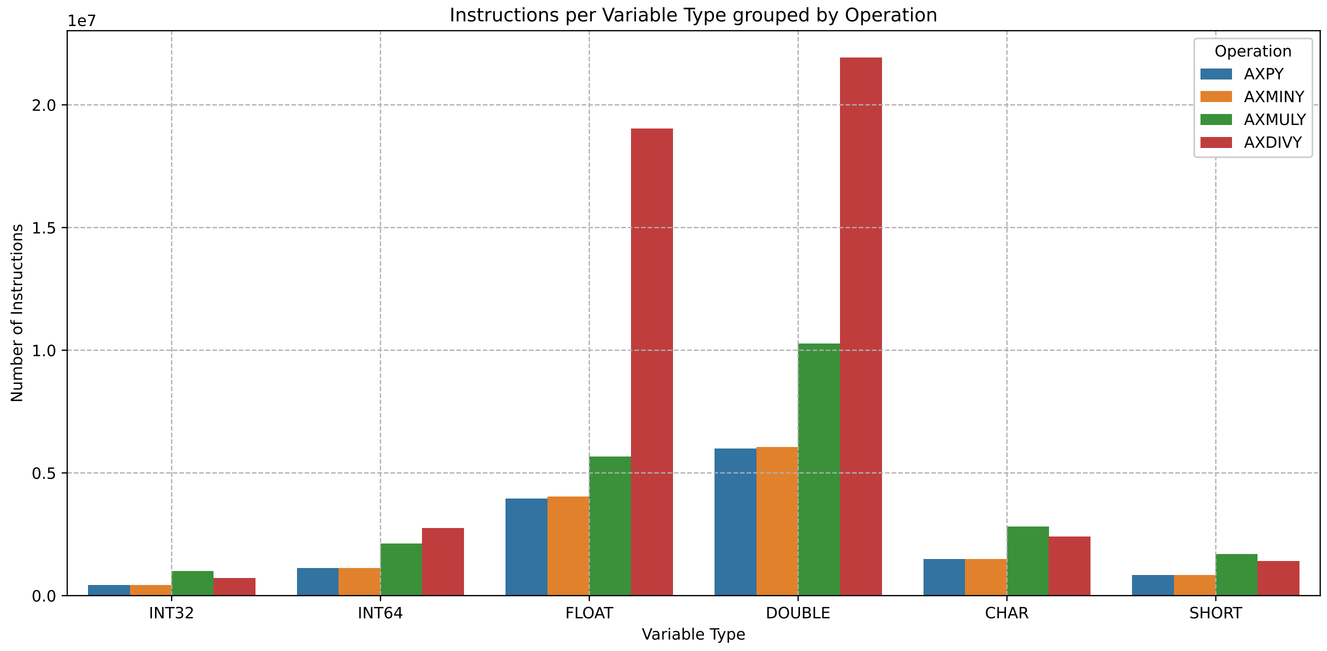

Figure 3: Task3_plot

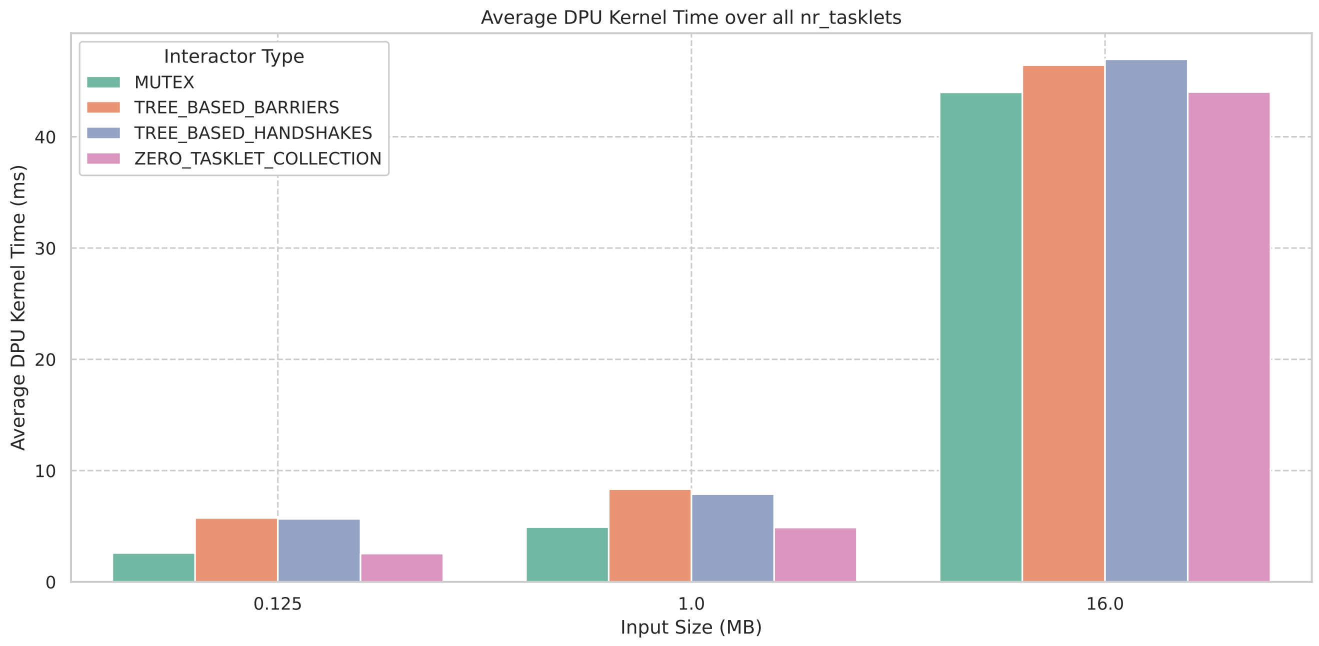

Figure 4: Task4_plot