Technologies of Computing Systems

LAB Demo – UPMEM PIM programming

Authors: Daniel Pacheco, Leonel Sousa

1. Introduction to programming UPMEM PIM

UPMEM PIM is the most used hardware employing processing-near-memory technology, which brings computation closer to memory by setting processing cores near the memory banks. This device equips each rank of DDR4 memory with 64 processing units, called DPUs. Each DPU has access to a DRAM bank, the MRAM, and a scratchpad memory, the WRAM. They can also run up to 24 threads, which in the context of UPMEM PIM are usually named tasklets. The execution of these tasklets can be managed with multiple tools, namely barriers.UPMEM PIM is always run together with a host CPU, which is responsible for sending data to the DPUs, receiving it, and launching the kernels.

This laboratory will introduce the use and programming of the UPMEM PIM device. To do so, the functional simulator provided by UPMEM will be used to perform a matrix-matrix multiplication, following 3 stages. In the first, you will perform the referred operation using a single DPU. In the second, you will divide the computation throughout multiple DPUs. In the last, you will expand upon the second by using multiple tasklets per DPU.

In this guide, Section 2 will provide the required information on how to install all the required software tools for performing this lab. Section 3 will guide along the execution of the work that is expected you perform in this laboratory.

2. Installing the required software tools

The instructions on how to install the required software to compile and run UPMEM PIM code using the simulator are provided in https://sdk.upmem.com/2024.2.0/01_Install.html.

1. Dependecies

- Operating systems must be Debian 10, Ubuntu 20.04 LTS, Ubuntu 22.04 LTS, or Rocky 8
- Python3 must be installed.

2. Installation (for Ubuntu)

- Download the ".tar.gz file" that corresponds to your operating system.
- Extract the file to a directory of your choice: "tar -xvf file.tar.gz -C /path/to/directory".
- Open your .bachrc file: "nano ~/.bashrc".
- Source the script in the file by adding the following line to the end of your ".bashrc" file: "source ~/path/to/directory/extracted_file/upmem_env.sh".
- Save and exit (to exit press Ctrl+Shift+X, then press Y to save).

3. Matrix-Matrix multiplication on UPMEM PIM

To start the work, download the corresponding files on the course page which contain the code from which you will be starting your implementation. These files are "host.c", which is the code that will be running on the host CPU, "kernel.c", which is the code that will be running on the UPMEM PIM device, and a "makefile".

To compile the program use the command make, and to run it use "./host.o". This program performs a multiplication between two 16x16 integer matrices, and outputs the matrix multiplication result on the CPU and on UPMEM PIM along with the number of errors found (values different on both implementations), so the objective is to get this number to 0 on all stages.

There are comments on the code that will help guide you, do not ignore them!

1 DPU

The first part of this work consists of performing the matrix multiplication using a single DPU. This means that this DPU will receive both matrices and will output a full matrix.

In this stage, communication with the host CPU is done using the "dpu_copy_to()" and the "dpu_copy_from()" methods. Within the DPU, the data has to be sent from the MRAM, where it was received from, to the WRAM, where it will be computed, by using the "mram_read()" method. The output will be sent from the WRAM to the MRAM using the mram_write() method.

16 DPUs

The second part of this work will involve using multiple DPUs. In particular, each DPU will receive a row of the first matrix (matrix1), the complete second matrix (matrix2), and output a row of the final result.

The first change that has to be done is regarding the “dpu_alloc()”, where now 16 DPUs must be allocated. Then, the communication between the host CPU and the DPUs will also be different, as you apply the “dpu_broadcast_to()” and “dpu_push_xfer()” methods to perform these transactions. In the kernel code, the size of the buffers needs to be adjusted to the data each DPU will be receiving, and the arguments of the matrix_matrix_multiply() function change also to compute only one row.

16 DPUs and 16 tasklets

The third and final stage of this work will exercise the usage of tasklets, as you use 16 tasklets per DPU, each responsible for computing a different element of the row assigned to that DPU.

To use 16 tasklets, you must add the -DNR_TASKLETS=16 flag to the makefile command that compiles the kernel code, so you will be replacing “dpu-upmem-dpurte-clang -o kernel kernel.c” with “dpu-upmem-dpurte-clang -DNR_TASKLETS=16 -o kernel kernel.c”

The buffers are shared by all tasklets, so their size will be the same as in the second step. However, the “mram_read()” and “mram_write()” methods should only be performed by one tasklet, which is tasklet 0. You can access the index of a thread using the method “me()”, which outputs an integer that, since you are using 16 tasklets, will be between 0 and 15. You can use an if statement to reserve a section of the code for a specific tasklet.

Note that all tasklets need to wait for the MRAM buffers to be ready before starting the multiplication, and all results must be computed before sending the results back to the MRAM, so you will need to use barriers. On the “matrix_multiply()” arguments, you need to use the tasklet index to specify which sections of the input and output you will be computing.