Programming and Understanding a Real Processing-in-Memory Architecture

Instructors: Geraldo F. Oliveira, Prof. Onur Mutlu

1. Introduction

In this lab, you will work hands-on with a real processing-in-memory (PIM) architecture. You will program the UPMEM PIM architecture [1, 2, 3, 4] for several workloads and will experiment with them. Your main goals are (1) to become familiar with the UPMEM PIM system organization (as an example of real-world memory-centric computing system), (2) to understand the UPMEM programming model and write your own code, and (3) to understand the microarchitecture and instruction set architecture (ISA) of UPMEM's PIM core (called *DRAM Processing Unit, DPU*).

As we introduced in this tutorial, the UPMEM PIM architecture is composed of multiple DPUs (up to 2,560), each of which has access to its own DRAM bank (called *Main RAM, MRAM*) and its own scratchpad memory (called *Working RAM, WRAM*). You can find a full description of the UPMEM PIM system in [3, 4].

2. Lab Resources

You can download the necessary materials for this lab from here: `https://events.safari.ethz.ch/heart24-memorycentric-tutorial/lib/exe/fetch.php?media=template.zip`

2.1. Source Code

The source code that we provide contains templates for tasks 1 (Section 4) and 2 (Section 5). For the rest of tasks, you can use the same template as for task 2. You can find the templates in the folder `template`. Look for `//@@` to find the places where you need to insert code. Do **NOT** modify any files or folders unless explicitly specified in the list below.

- `task1`
  - `Makefile`
  - `host`
    - `app.c`: Host CPU code (**modifiable**).
  - `dpu`
    - `task.c`: DPU kernel code. It is empty in this template because it is not needed for task 1.
  - `support`
    - `common.h`: Common definitions. Note that `T` is `int64_t` for this task.
    - `params.h`: Functions to read input parameters from command line.
    - `timer.h`: Timing functions.
- `task2`
  - `Makefile`
  - `host`
    - `app.c`: Host CPU code (**modifiable**).
  - `dpu`
    - `task.c`: DPU kernel code (**modifiable**).
  - `support`
    - `common.h`: Common definitions. Note that there are definitions for different data types and size of transfers between MRAM and WRAM.

  * `params.h`: Functions to read input parameters from command line.
  * `timer.h`: Timing functions.
  * `cyclecount.h`: Functions for performance counters (cycles and instructions). Check SDK documentation for more details about the performance counters (Section "Measuring performances" [2]).

3. Your Task 0/4: Installing the UPMEM SDK

In this tutorial, we will use the UPMEM functional simulator, available in the UPMEM SDK, to compile and execute code. Note that when using the functional simulator, you can only measure reliably a number of instructions. For this tutorial, we will measure the number of DPU cycles for a given task to have an indication of UPMEM's performance. However, in real scenarios, you should not rely on the measured number of cycles.

If you have access to a system with a supported Linux version, you can install the UPMEM SDK natively from the UPMEM website [1, 2]. If you encounter issues with the installation or do not have access to a system with a supported Linux version, you can use the Dockerfile we provide, along with the associated shell scripts for either Windows or Unix-based host systems.

3.1. Using the Dockerfile

Using the Dockerfile requires Docker [5] to be installed on your system.

☞ From a shell terminal, you can execute the `docker/start_docker.sh` shell script (`docker\start_docker.bat` on Windows):

```
$ wget -c 'https://events.safari.ethz.ch/heart24-memorycentric-tutorial/lib/exe/fetch.php?media=
    template.zip' -O template.zip
$ unzip template.zip
$ cd template/docker/
$ sudo su
# ls
Dockerfile start_docker.bat start_docker.sh
```

The script will automatically build the Docker image (which will take a few minutes the first time) and then start an interactive shell within it. The working directory of the host machine where the docker was started will be mounted to the Docker (try running `ls` inside the docker). The code for this lab can then be compiled and executed using this interactive shell.

☞ Open a second terminal, and copy the template files for task 1 and task 2 into your Docker image:

```
$ pwd
$ cp -r ../task1/ ../task2/ .
```

4. Your Task 1/4: Transferring Data between Main Memory and PIM-enabled Memory

Your goal is to get familiar with different types of data transfers between the host main memory and the PIM-enabled memory. You are provided with a template for this task. Find more details in Section 2.

Your tasks are as follows:

1. Write a host program that exercises all types of data transfers between the host main memory and one or multiple MRAM banks. Concretely, there are three types of data transfers [2]: (1) serial, (2) parallel, and (3) broadcast. Serial and parallel transfers move data from main memory to the MRAM banks or vice versa. Broadcast transfers can only happen from the main memory to the MRAM banks.
2. Evaluate all different types of data transfers for data transfers of size (1) 1MB, (2) 24MB, (3) 48MB per DPU. Use different numbers of DPUs between 1 and 64.

Based on your analysis, answer the following questions.

1. Create plots for the measured data transfer bandwidth for all combinations of data transfer size, type, direction, and number of DPUs.
2. What is the maximum bandwidth for one DPU? What is the maximum bandwidth for 64 DPUs? Do these numbers fulfill your expectations? Explain.

4.1. Compilation

The `Makefile` for task 1 contains the following input parameters and default values:

- `NR_DPUS ?= 1`: Number of DPUs that the program will use.
- `NR_TASKLETS ?= 16`: Number of tasklets per DPU that the program will use.
- `TRANSFER ?= PARALLEL`: Type of data transfer (`SERIAL`, `PARALLEL`, `BROADCAST`).
- `PRINT ?= 0`: Print log from the DPU kernel.

For task 1, you will only have to use `TRANSFER` and `NR_DPUS`. To compile with the default parameters:

```
$ make
```

To compile, for example, with serial transfers and 64 DPUs:

```
$ NR_DPUS=64 TRANSFER=SERIAL make
```

The compiled binaries will be in the `bin` folder. You can run the program with the default input:

```
$ ./bin/host_code
```

You can check the possible input arguments with `-h`:

```
$ ./bin/host_code -h
```

For example, you can run the program to transfer 2MB of data (i.e., 262144 64-bit elements) between the host main memory and the MRAM banks, and vice versa, and repeat the experiment 10 times after 2 times of warm-up:

```
$ ./bin/host_code -w 2 -e 10 -i 262144
```

5. Your Task 2/4: AXPY

Your goal is to get familiar with the DPU kernel launch and execution, and the performance scaling for different numbers of PIM threads (called *tasklets* in the UPMEM architecture). You are provided with a template for this task. Find more details in Section 2.

Your tasks are as follows:

1. Write a DPU kernel that executes the AXPY operation ($y = y + alpha \times x$) [6] on every element of a vector. You have to (1) transfer two input vectors, `Y` and `X`, to the MRAM bank/s, (2) perform the AXPY operation with a variable number of tasklets, (3) write the results to the output vector, `Y`, and (4) transfer the output vector back to the host main memory.
2. Allocate as much WRAM as needed, and use `mram_read` and `mram_write` to move data between MRAM and WRAM.
3. Your code should produce correct results for any data type (e.g., `char`, `short`, `int`, `long long int`, `float`, `double`) and vector size. Hint: Make sure all data transfers are 8-byte aligned [2].
4. Evaluate the performance of your kernel for all numbers of tasklets between 1 and 24. Use input vectors of, at least, 8 MB (per DPU).

5. Run your kernel for all numbers of tasklets between 1 and 24, and make sure that the kernel produces correct results in all cases (i.e., for any number of tasklets and DPUs). Use input vectors of, at least, 8 MB (per DPU).

Based on your analysis, answer the following questions.

1. Compare the performance of your kernel for different MRAM-WRAM and WRAM-MRAM transfer sizes (e.g., 8, 32, 128, 512, 1024, 2048 bytes). Do it for numbers of tasklets equal to 1, 4, 8, and 16. Report your observations.
2. Create a plot for the execution time with different numbers of tasklets. Use vectors of 32-bit integers (`int`).
3. Report your observations about the previous plot. For what number of tasklets does the performance saturate?
4. Use the performance counters to count the number of execution cycles and the number of executed instructions (see Section 2). Compare these numbers and enumerate your observations.

5.1. Compilation

The `Makefile` for task 2 contains the following input parameters and default values:

- `NR_DPUS ?= 1`: Number of DPUs that the program will use.
- `NR_TASKLETS ?= 16`: Number of tasklets per DPU that the program will use.
- `BLOCK ?= 10`: Size of the blocks (chunks) of data moved between MRAM and WRAM. The actual size in bytes is 2^{BLOCK}.
- `TYPE ?= INT32`: Datatype of the input arrays (`CHAR`, `SHORT`, `INT32`, `INT64`, `FLOAT`, `DOUBLE`).
- `TRANSFER ?= PARALLEL`: Type of data transfer (`SERIAL`, `PARALLEL`).
- `PRINT ?= 0`: Print log from the DPU kernel.
- `PERF ?= NO`: Use of performance counters for cycle or instruction count (`CYCLES`, `INSTRUCTIONS`)

To compile with the default parameters:

```
$ make
```

To compile, for example, with parallel transfers, 64 DPUs, 12 tasklets per DPU, 32-bit floating point datatype, 512-byte MRAM-WRAM and WRAM-MRAM data transfers, and counting instructions executed by the tasklets:

```
$ NR_DPUS=64 NR_TASKLETS=12 BLOCK=9 TYPE=FLOAT TRANSFER=PARALLEL PERF=INSTRUCTIONS make
```

As in task 1, the compiled binaries will be in the `bin` folder. For example, you can run the AXPY program to operate on 4MB arrays (e.g., 1048576 32-bit floating-point elements), with a value of `alpha` equal to 20, and repeat the experiment 10 times after 2 times of warm-up:

```
$ ./bin/host_code -w 2 -e 10 -i 1048576 -a 20
```

6. Your Task 3/4: Operations and Data Types

Your goal is to analyze how the DPU performs for different types of data and operations.

Your tasks are as follows:

1. Modify your AXPY DPU kernel to make it a vector addition ($y = y + x$) and to support other operations besides addition (i.e., subtraction, multiplication, division).
2. Evaluate the performance of your new kernel for different operations (addition, subtraction, multiplication, division) and data types (`char`, `short`, `int`, `long long int`, `float`, `double`).

Based on your analysis, answer the following questions.

1. Using the performance counters (see Section 2), measure the number of instructions and execution cycles for each data type and operation. For each operation and data type, you can calculate the average of both number of instructions and number of execution cycles.
2. Enumerate your observations about your evaluation.
3. We recommend you use the LLVM object file dumper (see Section 2) to read the assembly code of your program, and use this information to explain your observations.

7. Your Task 4/4: Vector Reduction

Your goal is to get familiar with the synchronization primitives for intra-DPU communication (i.e., across tasklets). To do so, you will write a DPU kernel that performs the parallel reduction of an input vector.

Your tasks are as follows:

1. Your vector reduction DPU kernel should have four different versions: (1) final reduction with a single tasklet, (2) final tree-based reduction with barriers, (3) final tree-based reduction with handshakes, (4) final reduction with mutexes.
2. Evaluate the performance of the different versions of your kernel for different numbers of tasklets. Use vector sizes of 16 KB, 1 MB, 16 MB.
3. Run the different versions of your kernel for different numbers of tasklets. Make sure all versions of your code produce correct results for any number of tasklets and DPUs. Count the number of executed instructions. Use vector sizes of 16 KB, 1 MB, 16 MB.

Based on your analysis, answer the following questions.

1. Enumerate your observations about your evaluation. What is the best performing version for different vector sizes?

8. Bonus Task: Implement RGB to Grayscale Conversion

In this task, your goal is to implement the RGB to grayscale conversion of an image [7] on a variable number of DPUs. The image should have 8-bit red, green, and blue channels. For each image pixel, you perform a conversion to obtain a grayscale value from the R, G, and B values. Compare the following three methods:

1. Lightness method: $grayscale = \frac{min(R,G,B)+max(R,G,B)}{2}$.
2. Average method: $grayscale = \frac{R+G+B}{3}$.
3. Luminosity method: $grayscale = 0.3 * R + 0.59 * G + 0.11 * B$.

Based on your analysis, answer to the following questions.

1. How do the different methods scale for different numbers of tasklets?
2. What is the fastest of the methods? Explain.
3. What is the slowest of the methods? Explain.

Think about potential optimizations of your code, implement them, and reevaluate. A general recommendation is to try to reduce the number of executed instructions.

9. Tips

- **LLVM Object File Dumper** You can use this tool to print the contents of object files [8]. The `-S` option displays the source code interleaved with the disassemble code: `$ llvm-objdump -S ./bin/dpu_code`

- Another useful tool with a similar purpose as the LLVM Object File Dumper is Compiler Explorer [9].
- **Please do not distribute the provided program files. These are for exclusive individual use of each participant of this tutorial. Distribution and sharing violates the copyright of the software provided to you.**
- **Read this handout in detail.**
- **Read the UPMEM programming guide [2].**
- **If needed, please ask questions to the instructor/s.**
- When you encounter a technical problem (e.g., a compilation error), please first read the error messages.

10. Acknowledgement

This tutorial was first developed by Dr. Juan Gómez Luna.

References

[1] UPMEM. UPMEM Software Development Kit (SDK). `https://sdk.upmem.com`, 2023.

[2] UPMEM. UPMEM User Manual. `https://sdk.upmem.com/2023.1.0/`, 2023.

[3] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernández, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture. arXiv:2105.03814 [cs.AR], 2021.

[4] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System. *IEEE Access*, 2022.

[5] Docker Inc. Docker. `https://www.docker.com`, 2023.

[6] Wikipedia. Basic Linear Algebra Subprograms. Level 1. `https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms#Level_1`, 2023.

[7] Wikipedia. Grayscale. `https://en.wikipedia.org/wiki/Grayscale`, 2023.

[8] LLVM. llvm-objdump - LLVM's Object File Dumper. `https://llvm.org/docs/CommandGuide/llvm-objdump.html`, 2023.

[9] Compiler Explorer. Compiler Explorer for DPU. `https://dpu.dev`, 2023.