Parallel Programming & Heterogeneous Computing

Summer Term 2019

Assignment 4 (Submission deadline: 2019-07-07, 23:59 CEST)

The fourth assignment covers the development of heterogeneous applications which use FPGAs in shared memory systems. Exemplarily you will use the toolchain and technologies from Xilinx, IBM as well as MetalFS. The tasks should be solved by extending the provided skeleton applications to interact with the MetalFS operator pipeline.

General Rules

The assignment solutions have to be submitted at:

https://www.dcl.hpi.uni-potsdam.de/submit/

Our automated submission system is intended to give you feedback about the validity of your file upload. A submission is considered as accepted if the following rules are fulfilled:

- You did not miss the deadline.
- Your file upload can be decompressed with a zip / tar decompression tool.
- Your submitted solution contains the compressed *metalfs-workshop* folder

If something is wrong, you will be informed via email (console output, error code). Re-uploads of corrected solutions are possible until the deadline.

50% must be solved correctly in order to pass each assignment. Documentation should be done inside the source code.

Students can submit solutions either *alone or as team of max 2 persons*.

FPGA Development

Initialize the repository by running `make hw_project` in the root folder. The development cycle with SNAP contains three steps:

1. Test your code in the Testbench by running `make test` in the operator directory
2. Build a Simulation Model by running `make model`
3. Test your Simulation Model by running `make sim`

In the simulation window type `snap_maint` and afterwards `metal_fs /mnt`. Now you are ready to run the commands of the respective tasks.

Task 4.1: Heat Map with MetalFS

The algorithm is basically the same as in the last assignments with some minor exceptions. Initially some of the blocks are cold (value=0), some other blocks are active hot spots (value=**255**). When all block values are computed in a round, the value of the hot spot fields may be set to **255** again, depending on the live time of the hot spot during a given number of rounds. For the convenience to work with MetalFS we assume that the width of the **field is fixed at 64 elements**. We also assume that an element is an unsigned character.

You have to extend the given C++ application and offload the simulation for the rounds onto the FPGA. For that you need to implement an MetalFS operator which can compute the round of heatmap and returns the results to the host. The goal is to minimize the execution time of the complete simulation.

Extend the implementation which you can find in the following repository:

https://gitlab.hpi.de/osm-teaching/parprog2019/task4.1

The code contains `// TODO` sections to be implemented.

Build

In the `/workspace/heatmap/` directory run:

```
1. mkdir build
2. cd build
3. cmake ..
4. make
```

Input

The executable is named `heatmap` and accepts **four** parameters:

- The height of the field in number of blocks.
- The number of rounds to be simulated.
- The name of a file (in the same directory) describing the hotspots.
- *Optional parameter*: The name of a file (in the same directory) containing coordinates. If it is passed, only the values at the indicated coordinates (starting at (0, 0) in the upper left corner) are to be written to the output file.

Please remember to run the heatmap program from the simulation shell. Use an absolute path to the binary because the simulation requires you to leave the working directory unchanged.

```
Example: /workspace/heatmap/build/heatmap \

    7 17 /workspace/heatmap/random.csv
```

Task 4.2: Decrypt (MD5) with FPGAs

Extend the skeleton implementation which you can find in the following repository:

https://gitlab.hpi.de/osm-teaching/parprog2019/task4.1

The code contains `// TODO` sections to be implemented.

Perform a brute-force dictionary attack on a provided MD5-hashed password. You need to develop a MetalFS operator that performs the MD5 hashing on a given input value. You can assume that the input data is already padded and is exactly 64 byte. We provide an example dictionary (`dict.txt`) and a hash (`hash.txt`) for your validation.

Build

The operator is build using the standard workflow described above. You only need to build the `padder` and `unpadder` binaries by calling `make padder unpadder`.

Input

Your operator "md5hash" takes a file containing the binary representation of the hash to search for as a file. The operator expects a data stream of padded 64 byte fixed length strings to calculate the MD5 hash.

```
      Example:  cat  dict.txt  |  ./padder  |  /mnt/operators/md5hash  -s
./hash.txt | ./unpadder


      hd hash.txt

      5d 6c 73 d8 df ce f7 14  fe d8 8f 58 4f 04 e6 3f
```

Output

The program will print line by line the list of the successfully found password(s):

```
      TrilogyHandinessGallowsUnsignedSponsorPetticoatThinnerE
```