Barcelona Supercomputing Center
Centro Nacional de Supercomputación

HEAT EQUATION

Okke van Eck
okke.vaneck@bsc.es
June 19, 2024

1 Introduction

Welcome to the BSC Heat Equation assignment! The goal of this assignment is to briefly give you hands-on experience with the basic tools required to asses the performance of your GPU porting. In the examples, we will look at the effects of poor memory management, which is the most common bottleneck for GPU performance. Can you figure out what limits performance and when to use which profiling tool?

We will use the heat equation as our toy problem. It is a partial differential equation used to model the transfer and diffusion of heat. No need to know the specifics, we provide you with a full model that works straight out of the box.

2 Code repository

You can find the code on Lumi at the following path:

`/scratch/project_465000527/ovaneck/heat_equation_assignment`

When copying paths, you might loose the underscores and need to add them yourself! Please copy the code to your personal environment before proceeding, like:

```
cp -r /scratch/project_465000527/ovaneck/heat_equation_assignment \
    $HOME/heat_equation_assignment
```

Before you can work with the model, it is good to understand the structure of the repository. We can see the following structure:

```
heat_equation_assignment/
├── bin/ ..... Contains binaries.
├── debug/ ..... Contains runtime debug output.
├── info/ ..... Contains compilation output.
├── profiling/ ..... Contains profiling tools.
│   ├── res-<platform>-<binary>/ ..... Profiling results directory.
│   ├── gpu_modelfactors.py ..... Visualizer for hardware counters.
│   ├── profile.sh ..... Script for profiling.
│   ├── rocprof_counters.txt ..... Contains the hardware counters to collect.
│   ├── requirements.txt ..... Python requirements for gpu_modelfactors.py.
│   ├── select_gpu ..... Required script for using GPUs on Lumi.
│   └── workloads.txt ..... Defines the experiments to run.
├── src/ ..... Contains the source code.
│   ├── core/ ..... Contains files with computations.
│   ├── helpers/ ..... Contains model helpers.
│   ├── main/ ..... Contains main functions.
│   └── Makefile ..... Compiles the source code into binaries.
├── allocate_gpu_node.sh ..... Script for acquiring a GPU node.
├── helpers.sh ..... Script only used internally.
├── load_modules.sh ..... Script for loading the modules.
└── README.md ..... Instruction for the repository.
```

The red directories are ones that will be created in the process of compiling the application and collecting performance related data. The other directories and fields will be discussed along the way during this assignment. There are 3 versions of the heat equation model:

1. `heat_sequential`: non-paralellized version
2. `heat_gpu_good`: a proper GPU enabled version
3. `heat_gpu_sloppy`: a sloppy GPU enabled version

You will profile the two GPU versions and compare the output to see what are exactly the differences. The `src/` directory contains all the source files of all versions, separated in 3 sub-folders: `core/`, `helpers/`, and `main/`:

- `core/`: contains the time step functions for updating the model
- `helpers/`: contains some boilerplate code for the model
- `main/`: the program definitions with main functions

Take a look at the main and core source files of the `heat_sequential` version. Can you see how the program works, where the model is updated, and how the algorithm for updating works?

3 Building and executing the model

Next, we will take a look at how to compile the model and execute it. First and foremost, we have to specify the platform we are operating on. Since we are working on Lumi, we can simply set `PLATFORM` via:

```
export PLATFORM=lumi
```

Once you have set the platform, it is time to acquire a GPU node to perform the assignment on. This can easily be done by executing the `allocate_gpu_node.sh` script:

```
./allocate_gpu_node.sh
```

This might take some time depending on how busy the supercomputer is. After the script is done, you will be automatically be SSH-ed into the allocated GPU node. So from now onwards, everything executed happens inside the GPU node.

Before executing the model, we have to compile it. The compilation procedure depends on some modules that need to be loaded. This can be done by **sourcing** the `load_modules.sh` script. Next, we can compile the source. Go into the `src/` directory and compile the source code using the `Makefile` by running `make`. These three steps should look like:

```
source load_modules.sh
cd src
make
```

If everything went successfully, you should now have the `bin/` folder in the root of the repository. This folder contains all 3 binaries of the previously discussed versions. You can try them out by simply executing them. Each binary takes 3 optional arguments: `nx` and `ny` specifying the size of the grid, and `nsteps` indicating the number of time steps to take. Their defaults are 2000, 2000, and 500 respectively. You can execute for example the `heat_sequential` version, providing (roughly) the following output:

```
cd ../bin
./heat_sequential 2000 2000 500
> Average temperature at start: 59.763305
>              Iterations took:  1.092    seconds.
>          Average temperature: 59.281239
>  Reference value for default: 59.281239
```

Try the GPU versions as well and see if you notice any differences between them!

4 Profiling the model

Now that we have working models, it is time to profile their performance. This is done from the `profile/` folder. In this folder, you can see 4 files: `profile.sh`, `rocprof_counters.txt`, `select_gpu`, and `workloads.txt`. Both `rocprof_counters.txt` and `select_gpu` can be ignored for this assignment. The `profile.sh` script handles all the complexities of profiling with tools, and thus provides an easy interface for you to use. It uses **rocprof**[1] under the hood. For more detailed information on how to use them, we redirect you to its documentation pages as linked in the footnote. Lastly, there is the `workloads.txt` file in which you can specify the experiments you want to run for your analysis.

There are 3 ways in which we can collect information of an application's performance:

1. Collect a trace and hardware counters during execution
2. Investigate the compiler runtime environment information
3. Investigate the compilation information

Each of the subsections below will investigate one of the ways using our two binaries `heat_gpu_good` and `heat_gpu_sloppy`. It will give you insight in what each method has to offer and what could be useful for you in the future. Note that all steps assume you are still inside your allocated GPU node!

4.1 Collecting a trace and hardware counters

First and foremost, you should know that collecting a trace and hardware counters are two separate steps. However, often both are collected with the same tool

[1] `https://github.com/ROCm/rocprofiler`

(like rocprof) and thus we discuss them together. The `profile.sh` script is essentially an easy interface for rocprof and collects the trace and hardware counters for you. This can simply be done by running the script with the binary name as a first argument. So to profile the `heat_gpu_good` version, we would run:

```
cd profiling
./profile.sh heat_gpu_good
```

This will run the profiling tools on the provided binary and store the trace and hardware counters in a results folder. The folder will be named `res-<platform>-<binary>/`, where the platform and binary name are replaced.

To fully understand the performance of a GPU application, we need to profile the application under multiple workloads. This can be specified in the `workloads.txt` file, where each line is a configuration that will be profiled by the `profile.sh` script. It is difficult to say in the beginning what workloads are sufficient to fully load the GPU, so you just have to try some and see! An extended `workloads.txt` file could look something like:

```
nx ny nsteps
------------
1000 1000 500
2000 2000 500
...
```

You will see that adding more lines adds sub-folders in the results directory with the pattern `<nx>_<ny>_<nsteps>/` after you run the `profile.sh` script. In this folder you will find the output and error messages in separate log files, as well as a sub-folder with the profile results. Feel free to explore the created files, however, in Section 5 we will provide you with automated tools for a detailed analysis.

4.2 Compiler runtime environment information

The compiler runtime environment is also capable of providing us with information on the application. It can output information on every offloading operation, where the amount of detail can be controlled. This is regulated through an environment variable which is different for each platform. Please read the subsection below that is for the platform you are using.

4.2.1 Lumi - Cray CCE

On Lumi, you can set the `CRAY_ACC_DEBUG` environment variable to either 1, 2, or 3. With `CRAY_ACC_DEBUG=1`, you can see the data transfers with source lines, as well as kernel executions. An example of one iteration is:

```
CRAY_ACC_DEBUG=1 ./heat_gpu_good 2000 2000 3
> ACC: Transfer 2 items (to acc 64128320 bytes, to host 0 bytes)
      from main/main_gpu_good.F90:34
> ACC: Transfer 2 items (to acc 0 bytes, to host 0 bytes)
      from main/main_gpu_good.F90:37
> ACC: Execute kernel heat_solve_gpu_good_$ck_L37_1_cce$noloop$form
      async(auto) from main/main_gpu_good.F90:37
> ACC: Wait async(auto) from main/main_gpu_good.F90:37
> ...
```

When setting `CRAY_ACC_DEBUG=2`, you also see the transferred variables, memory operations, and the GPU setup for a kernel execution:

```
CRAY_ACC_DEBUG=2 ./heat_gpu_good 2000 2000 3
> ...
> ACC: Start transfer 2 items from main/main_gpu_good.F90:34
> ACC:       allocate, copy to acc 'current' (128 bytes)
> ACC:       allocate, copy to acc 'current%data' (32064032 bytes)
> ACC:       attach pointer 'current%data' (96 bytes)
> ACC:       allocate, copy to acc 'previous' (128 bytes)
> ACC:       allocate, copy to acc 'previous%data' (32064032 bytes)
> ACC:       attach pointer 'previous%data' (96 bytes)
> ACC: End transfer (to acc 64128320 bytes, to host 0 bytes)
> ACC: Start transfer 2 items from main/main_gpu_good.F90:37
> ACC:       present 'currdata(:,:)' (32064032 bytes)
> ACC:       present 'prevdata(:,:)' (32064032 bytes)
> ACC: End transfer (to acc 0 bytes, to host 0 bytes)
> ACC: Execute kernel heat_solve_gpu_good_$ck_L37_1_cce$noloop$form
      blocks:2000 threads:256 async(auto) from
      main/main_gpu_good.F90:37
> ACC: Wait async(auto) from main/main_gpu_good.F90:37
> ...
```

Lastly, setting `CRAY_ACC_DEBUG=3` gives us the most detailed information possible on offloading operations. For transfer operations, we now also see the pointers, flags, detailed memory operations, detailed data structure information, and meta data transfers. For kernel executions we also have more information on the resource usage, as well as flags, cache pointers, and caching behavior. All this information is too big to show in an assignment, so we encourage you to play around and see for yourselves!

4.3 Compilation information

It is also possible to acquire information on the GPU porting during compile time. The compiler performs all kinds of optimizations, which it can report on

by specifying some flags. These flags are different for each compiler. The output is a list of optimizations, which also contain information on GPU offloading. The subsections below will show you how to filter this information per platform.

4.3.1 Lumi - Cray ftn

Within the Cray environment, we use the `ftn` compiler. The `ftn` compiler outputs optimization information through the `-hmsgs` flag, like so:

```
ftn -hmsgs -c gpu_heat_good.F90 -o gpu_heat_good
```

Our heat equation assignment is a bit more complex, and thus we added an option to the Makefile which produces and stores this output for you. Simply go back into the source directory, clean the current installation, and then re-make with `info` specified:

```
cd src
make clean
make info
```

This created an `info/` folder in the top level `heat_equation_assignment` folder with the compilation steps of all three binaries in separate files. Please open the file and see for your self what it provides! You could for example take a look at the different information types that are reported through:

```
cat gpu_good_compiler_info.txt | grep ftn-
```

As you can see, the compiler does *many* optimizations for us. The heat wave equation code is fairly small, but for real life applications specialized tools are required for filtering out useful information.

5 Analyzing the performance data

In the previous steps we have collected information on the performance of our GPU porting using a variety of methods. Now it is time to analyze those findings and make conclusions on the performance of our applications. The subsections below give an overview of how to analyze the data, but it will be up to you to come to a conclusion!

For the trace and hardware counters, it is required to pull the results from the supercomputer to your local device. This can be done with a couple simple steps. First, you need to get the path to your `heat_equation_assignment/` folder. The easiest way is to go to the folder and echo your working directory, for example:

```
echo $PWD
> /users/vaneckok/heat_equation_assignment/
```

We will refer to this path as `<heat_equation_path>` below. Next step is to pull the results from collecting the traces and hardware counters. This can simply be done with the following `scp` command in your **local terminal**:

```
scp -r <platform>:<heat_equation_path>/profiling/res-\* .
scp -r <platform>:<heat_equation_path>/profiling/gpu_modelfactors.py .
scp -r <platform>:<heat_equation_path>/profiling/requirements.txt .
```

Note that you need to replace `<platform>` with the hostname you use to connect with the supercomputer!

5.1 Analyze a trace

The `profile.sh` script collected a trace of the execution and stored it in the results folder. To view traces, you need to use a separate program like Perfetto[2]. Each GPU platform has its own tool for viewing their traces, and thus we will cover each platform separately.

5.1.1 AMD - Lumi

To analyze an AMD trace, we use an external tool called Perfetto[2]. Simply go to their website as put in the footnotes, and click on "Open trace file" in the menu. Then select the `rocprof.json` file in the `rocprof` folder of the results that you previously copied over using `scp`. This should open the trace, which roughly looks like the one below in Figure 1.

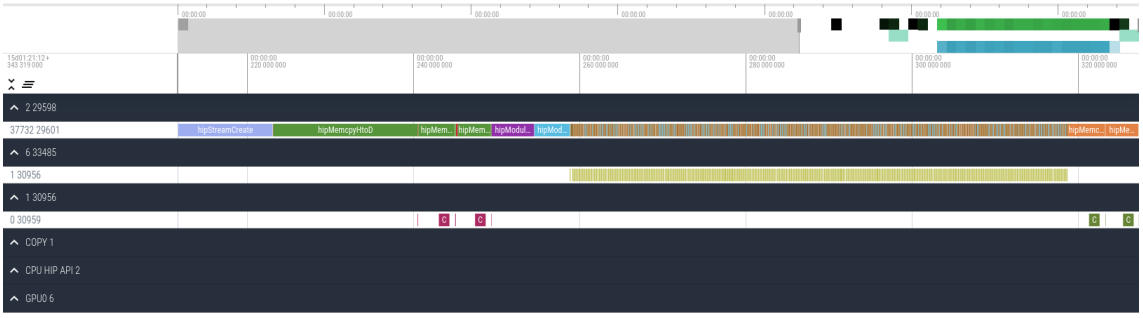

Figure 1: Example Rocprof trace visualized using Perfetto[2].

You can navigate the trace with the `a`, `w`, `s`, and `d` keys. Clicking on an item that has your interest will provide you with more details in the windows below.

5.2 Analyze hardware counters

Hardware counters are performance metrics that are collected for each kernel execution. This means that for an application with 1 kernel, like our heat equation,

[2] `https://ui.perfetto.dev/`

the number of measurements is equal to the number of time steps. You can see this is in the `rocprof.csv` file.

To make your lives easier we also offer a Python script for making easy to understand tables of the performance counters. It parses the result folders automatically and creates a column per workload. We listed all required modules in a `requirements.txt` file, which we recommend installing in a fresh Python virtual environment. Afterwards, you can simply run the script with the binary version as an argument. The total flow will look something like:

```
python -m venv venv
source venv/bin/activate
pip install -r requirements.txt
python gpu_modelfactors.py heat_gpu_good
```

This saves the hardware counter table as `scaling.png` in the results folder. Performing a good analysis of the kernel at hand requires knowing the internal structure as well. This table will provide you with a starting point of potential bottlenecks, which can be verified by looking at the source code or runtime environment. It might be that your workloads are not big enough to reach the limits of the machine and thus see differences. Don't be afraid to significantly increase them! What can you say about the kernel when looking at the table?

5.3 Analyze the compiler runtime environment

You can get runtime information from the compiler runtime environment as described in Section 4.2. To make it a bit easier for you, we automatised this through the `Makefile` as well with a `debug` target:

```
cd src/
make debug
```

This will create a `debug/` folder in the top level `heat_equation_assignment` folder with a file per binary and `CRAY_ACC_DEBUG` level. The contain the output of executing each binary for 3 time steps. Can you figure out why the sloppy implementation is so much slower than the good one? Try different levels of debug to see what works for this problem at hand. Remember that the number of time steps multiplies the length of the output!

5.4 Analyze the compiler output

If you have compiled the code with `make info`, you should have an `info/` folder in the top level `heat_equation_assignment` folder with files containing compilation output. Looking at their contents, you can see the many optimizations the compiler has performed. The ones starting with `ACCEL` are related to the GPU porting. You can filter them out through `grep`, where the `-A1` flag also shows the line after a match:

```
cat gpu_good_compiler_info.txt | grep -A1 ACCEL > good.txt
cat gpu_sloppy_compiler_info.txt | grep -A1 ACCEL > sloppy.txt
```

Can you spot the differences between the sloppy and good implementation? Don't forget to look into the source code to see the exact implementation differences!

6 Final notes to the reader

The code base of the assignment is an adapted version from the heat equation assignment of the CSC summer school[3]. Please take a look there if you want to get a broader experience with OpenMP GPU offloading.

In this assignment you have experienced how performance analysis of GPU applications roughly goes. However, do take in mind that our heat equation *only* had 1 GPU kernel. Real world climate models can have hundreds. This complicates the analysis significantly, and usually you also need to have a pre-analysis of the different kernels and their relative duration. This can be derived from the `rocprof.stats.csv` file in the profiling results.

[3] `https://github.com/csc-training/summerschool/tree/master/gpu-openmp`