Praktikum im WS 2022
Quantitative Analyse von Hochleistungssystemen (LMU)
Evaluation of Modern Architectures and Accelerators (TUM)

Sergej Breiter, MSc., Minh Thanh Chung, MSc.,Amir Raoofy, MSc., Bengisu Elis,MSc.,
Dr. Karl Fürlinger, Dr. Josef Weidendorfer

Assignment 02 – Due: 10.11.2022

In the first assignment, we focused on experimenting with the vector triad microbenchmark on CPU-based systems. For many reasons, including performance, cost- and power-efficiency, accelerators, e.g., GPUs are becoming more widespread in HPC environments. In this assignment, we investigate the performance of the same microbenchmark to study basic hardware characteristics (peak performance, memory bandwidth, and latency). For this, we use the same benchmarks and adapt them to use OpenMP offloading directives to use GPU resources in BEAST, i.e., AMD MI50 and Nvidia V100 GPUs.

Compiler Usage for Offloading :

While learning low-level programming interfaces such as OpenCL, CUDA, and HIP is essential, in this assignment, we focus on a directive-based programming interface, i.e., OpenMP. This simplifies the development process and portability of your code. For that, you need to annotate your vector triad microbenchmark code with appropriate OpenMP offloading directives and rely on the compilers for generating GPU codes. On BEAST systems, we use various compiler toolchains on different systems:

(a) **Toolchain on ThunderX systems**

On ThunderX systems LLVM compiler with offloading support to Volta devices through Nvidia Parallel Thread Execution (NVPTX) is available. In the following, we provide instructions for using this toolchain for compilation:

- setup environment:

```
$ module load cuda/11.1.1
$ module load llvm/11.0.0_nvptx_offloading
```

- minimal flags to enable offloading at compile time:

```
$ clang -fopenmp -fopenmp-targets=nvptx64 \
  -Xopenmp-target=nvptx64 -march=sm_70 <your code file>
```

(b) **Toolchain on Rome Systems**

On Rome systems Clang compiler, which is based on LLVM and uses AMD Graphics Core Next (GCN) for code generation, is provided. In the following, we provide instructions for using this toolchain for compilation:

- the default environment on Rome machines is already setup login.
- minimal flags to enable offloading at compile time:

```
$ clang -fopenmp -target x86_64-pc-linux-gnu \
  -fopenmp-targets=amdgcn-amd-amdhsa \
   -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx906 <your code file>
```

- Caveat: the "clang" binary used should be found from path /opt/rocm/llvm/bin/

For this set of tasks, please use the provided vector triad implementation similar to assignment #1 called `assignment2.cpp`. This implementation already offloads the vector triad to GPU by OpenMP offloading. When experimenting with this code (although there is a `Makefile`), make sure to use the compilers available on BEAST systems according to the above instructions.

1. **Offloading :** Make sure that the triad computation is offloaded to GPU by inspecting the corresponding code lines in the source file (on lines 31-37) as well as by checking GPU status while running the code (see Assignment #0 for hints).

2. **Workload Distribution:** Make sure full utilization of all GPU threads is achieved by adding necessary clauses to pragma directives.

3. **Evaluating Data Transfer:** For your microbenchmark run following experiments.

   - **Variant 1:** initialize data on the host CPU before the main loop of the vector triad and then map the data to the GPU domain.
   - **Variant 2:** initialize data on GPU. For this variant, make sure your data is resides on GPU memory after initialization and not freed before going on to computation. Using `#pragma omp target enter data` clause may help as well as other approaches.

   Compare the performance differences for these two experiments and explain the differences. Hint: in the second variant, you only need to use `map(alloc)` clause, while in the first variant, you need to use `map(to)` clause.

4. **Loop Scheduling Policy:** Use the necessary scheduling scheme to enable memory coalescing (see the lecture slides) and compare the performance with and without specifying any scheduling scheme. Explain your results.

5. **Execution Configuration on Target Device:** In the lecture, you learned OpenMP clauses that configure the configuration of teams and threads for kernel execution on target devices. Use these clauses for the following subtasks:

   (a) Use proper configurations for scheduling loop iterations with proper workload distribution to relevant numbers for teams/threads to see the effects. For the following tasks, use the most optimal solutions from this task and all previous tasks.

   (b) Use the combinations of league and team size configurations from the following set: {(t, T)} = { (32,1), (64,1), (128,1), (256,1), (512,1), (1024,1), (2048,1), (4096,1)}. Here, "t" denotes the number threads per team, "T" denotes the number of teams. Find the optimum number of threads per Team at which you get the max performance on each platform of BEAST with GPUs. Explain why you have this certain optimum number for threads per Team.

   (c) Using t* (your optimum number of threads per team from the previous part), use the combinations of league and team size configurations from the following set: {(t, T)} = { (t*,20), (t*,40), (t*,60), (t*,80), (t*,100), (t*,120), (t*,140), (t*,160), (t*,180), (t*,200), (

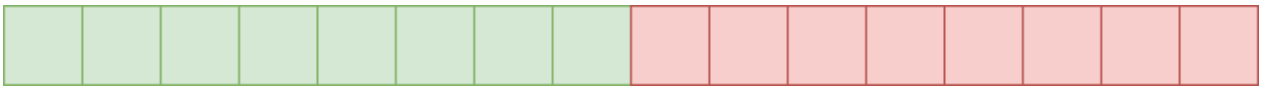

Figure 1: Illustration of distribution and assignment of loop iterations (e.g., accessing elements of an array of 16 elements) to different devices; here, we are using an illustration of 16 loop iterations that are divided between two devices. The different colors show the assignment of loop iterations to different devices.

t*,220)}, and find the optimum number of teams for your optimum number of threads per team on each platform of BEAST with GPUs. Explain why you have this certain optimum number of teams at which you get the max performance.

(d) You can use this optimal number of teams and threads from this point on for the following tasks.

6. **Flop Rate and Memory Bandwidth:** Conduct similar experiments as in assignment #1 and produce *Performance (Flop rates)* vs. *Vector Length* plots. Repeat these experiments on different systems in BEAST with Nvidia and AMD GPUs. Compare the performance results from the CPU-only experiments you have from assignment #1. To compare with CPU performance, use the configuration with the optimal configurations (i.e., number of threads and affinity).

7. **Multi-GPU Experiments:** There are multiple GPUs both on Rome and ThunderX systems. In this task, we use all of the available GPUs on a system (e.g., Rome or ThunderX) for the vector triad microbenchmark. For this, we split the total workload, e.g., the iterations of the vector triad loop, into smaller chunks and assign them to different devices for execution. Fig. 1 illustrates an example of such splitting of workload, a.k.a., partitioning and decomposition, for splitting the workload between two devices.

   One possible scheme to perform this partitioning for the vector triad microbenchmark is to divide the arrays into as many equal chunks as the number of GPUs in the system (e.g., two chunks for the ThunderX). There are multiple possible schemes to invoke execution on multiple devices, e.g., using OpenMP tasking or nested parallel regions—You can opt to use any correct scheme here. However, here we rely on the execution of separate `target` regions: You may use as many separate `target` regions as the number of devices, assign one chunk to each device, and map the chunks separately for each target region, and offload the computation of that specific chunk. You need to find the right clauses to ensure the execution is not blocked in one of the target regions, and the CPU can invoke execution on multiple devices simultaneously. Measure the overall Flop rates and aggregated memory bandwidth achieved when utilizing all the GPUs on different systems of BEAST.