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

Vincent Bode, MSc., Minh Chung, MSc., Dennis-Florian Herr, MSc., Bengisu Elis, MSc., Dr. Karl Fürlinger, Amir Raoofy, MSc., Dr. Josef Weidendorfer

Assignment 04 – Due: 02.06.2022

In the first three assignments, we focused on experimenting with various microbenchmarks on CPU-based systems. For many reasons, including performance, cost- and power-efficiency, accelerators, e.g., GPUs are becoming more wide-spread in HPC environments. In this assignment, we investigate the performance characteristics (peak performance, memory bandwidth, and latency) of the Vector Triad (Part I) and Matrix Multiplication (Part II) microbenchmarks on GPUs. For this purpose, you will adapt these benchmarks to use OpenMP offloading directives to utilize GPU resources in BEAST, i.e., AMD MI100 and Nvidia V100 GPUs.

Before going ahead with tests and analysis on the microbenchmarks, it is important to keep in mind that we investigate the Vector Triad as a memory-bound and Matrix Multiplication as a compute-bound benchmark. This information may be helpful when explaining the observed results from the microbenchmarks.

While learning lower-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:

1 Compiler usage for Offloading:

1. **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>
```

2. **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 after 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/

2 Part I : Vector Triad

For this set of tasks please use the provided vector triad implementation similar to assignment #1 called `assignment4_part_i.cpp`. This implementation already offloads the vector triad to GPU by OpenMP offloading. When experimenting with this code (although there is a `Makefile`) please 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).

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 try following experiments.

   - **Variant 1:** initialize data on CPU before the main loop of the vector triad and then map the data to GPU domain.
   - **Variant 2:** initialize data on GPU. For this variant make sure your data is persistent 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 specification of any scheduling scheme. Explain your results.

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

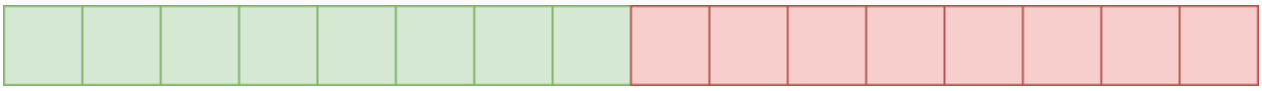

Figure 1: Illustration of distribution and assignment of loop iterations to different devices; here, we are using an illustration of 16 loop iterations that are divided between two devices. The different colors show the assignment of loop iterations to different devices.

(a) Use proper configurations for scheduling loop iterations with proper workload distribution to relevant number of teams/threads to see the effects. For 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)}. 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 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), (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 optimal number of threads and pining on CPU.

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.

8. **Power Measurements:** For the only CPU, single GPU and multi GPU versions of your code check the power consumption and present the results in a table comparing speed up and power consumption. For speed up calculations take the only CPU version of your code as baseline (with speed up 1).

   To take power measurements use The Data Center Data Base (DCDB) system monitoring framework installed on the BEAST gateway node.

   - To use DCDB you should:
     `source /opt/dcdb/dcdb.bash`.
   - You can list available ac power sensors by:
     `dcdbconfig sensor list|grep power_ac`.
   - Sensor values and timestamps can be read by :
     `dcdbquery /beast/<node name>/<sensor name> now-5m now`
     This last command outputs the average power measurements from 5 minutes before until now, reported by DCDB every 30sec.

   Power Distribution Unit(PDU) is a device with multiple power outlets, designed to distribute electric power, especially to racks of computers and networking equipment located within a data centre. Power consumption from each outlet can be measured, for example for sensor "*(/beast/pdu3/power5)*", which is power provided by outlet 5 of PDU 3. Note that each BEAST node has 2 PDUs for redundancy and the total power consumption of a node is the sum of measurements taken from both PDUs. The *power_ac* virtual sensor output performs this summation of two PDUs. In addition, the power measurements reported by DCDB are obtained by averaging PDU-internal power measurements, that are taken every 30msecs. For correct and interference free measurements, determine how much sleep time is necessary before and after the benchmark runs.

3 Part II : Matrix Multiplication

For this set of tasks please use the provided matrix-matrix multiplication (MM) implementation similar to assignment #2 called `assignment4_part_ii.cpp`. This implementation already offloads the MM to GPU by OpenMP offloading.

1. **Offloading with Optimal Data Transfer Policy:** Make sure that the MM computation is offloaded to GPU. From your results of Part I task 3, choose the best data initialization policy and make sure this policy is used to initialize and migrate your matrices to GPU.

2. **Optimizations:** Unlike Assignment #2, where we used loop interchange (reordering the loops) as the first step, we use another technique for optimization. For this you need to introduce two variants of matrix multiplication benchmark, both with the same "ijk" loop ordering.

   Variant 1 : Store the elements of both multiplicand $\mathbf{B}$ and multiplier $\mathbf{C}$ in row-major layout.
   Variant 2 : Use row-major layout for multiplicand $\mathbf{B}$ and column-major layout for multiplier $\mathbf{C}$.

   For both variants complete the following tasks:

   (a) Apply cache blocking (similar to Assignment #2)

(b) Use appropriate loop scheduling policies and vectorization directives, based on your experience in Part I of this Assignment.

(c) Run similar experiments to those in Part I task 6 to measure FLOP rates and memory bandwidth utilization of your code on GPU.

(d) Explain your results and compare it to the results you achieved in Assignment #2, for matrix multiplication benchmark on CPUs.

3. **Execution Configuration on Target Device:** Iin Part I you learned how to use OpenMP clauses that configure execution of teams and threads on target devices in practice. Use these clauses for the following subtasks:

   (a) Make sure as much parallelism as possible is achieved by using OpenMP clauses by adjusting loop iterations, and number of teams/threads.

   (b) Use all possible combinations of league and team size combinations from the following set: $\{(t, T)\} = \{ 8, 16, 32, 40, 48, 64, 72, 80, 88, 96, 104, 112, 120, 128, 256, 512,1024\} \otimes \{ 32, 64, 80, 128, 256, 512, 1024\}$ and plot the flop rate vs. league/team size combination plot in 3D. Here, "t" denotes the number threads per team, "T" denotes the number of teams. Perform this task only for a single matrix size for each thread and team size combination but make sure the matrix size is large enough for the number of threads and teams you have for each combination. Explain your observations for each system on BEAST with GPUs.

   (c) Find the optimum team number vs thread number configuration at which you get the maximum performance and explain the reason for it. You can use this optimal combination for the following tasks.

4. **Power Measurements:** Repeat the power measurement experiments from Part I task 8 for the matrix matrix multiplication. Use the guidelines and commands in Part I. For CPU only and single GPU versions of your code:

   (a) Present your results for speed up and power consumption, in a table.

   (b) Plot Mflops per Watt (Mflops/Watt) vs data set size results, similar to previous assignments' Mflops/sec vs data set size plots.