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 08 – Due: 12.01.2023

The main goal of this assignment is to implement and measure performance metrics of a Matrix-Matrix Multiplication benchmark with CUDA/HIP programming platforms and observe the effects of the usage of different memory types on GPUs. For this assignment, you are expected to use NVIDIA V100 (Ice2 node) and AMD MI100 (Rome2 node) on the BEAST system. When compiling your GPU code on Icelake systems please load the cuda module. On Rome systems, the default environment already provides necessary modules.

CUDA and HIP are parallel computing platforms and programming models for GPUs, created by NVIDIA and AMD respectively. They provide their respective profiling/optimization tools, libraries, SDKs, and programming APIs with a wide range of capabilities. Although provided by different vendors, CUDA and HIP embrace the same programming model and provide similar components. Due to their similar nature, the following introductory information is provided only for CUDA. Please keep in mind that although the naming convention may vary, the concepts explained below are valid for HIP as well.

Throughout this assignment, there are three webpages that might be helpful:

- Quick Start: `https://developer.nvidia.com/blog/easy-introduction-cuda-c-and-c/`
- CUDA API documentation : `https://docs.nvidia.com/cuda/cuda-runtime-api/index.html`
- GPU Memory Architecture: `https://cvw.cac.cornell.edu/gpu/memory_arch` and `https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/`

1. **Basic CUDA CPU code Structure:** In CUDA programming model the application code consists of 2 parts - CPU code and GPU kernel. The GPU kernel is designated by the `__global__` attribute and includes the computation, that is to be running on GPU. The minimum CPU code consists of the following stages:

    1. Declare and allocate host and device memory.
    2. Initialize host data.
    3. Transfer data from the host to the device.
    4. Execute one or more kernels (kernel launch and synchronization).

5. Transfer results from the device to the host.

Similar to the OpenMP offloading, in CUDA model the GPU kernel is executed by Teams and Threads. In CUDA naming convention Teams are called Blocks and the set of all teams is called a Grid. Also different than OpenMP model, Grid and Blocks are 3D data structures (specified by datatype `dim3`) with x, y and z dimensions. The 3D nature of Grid and Blocks is helpful when mapping data points to hardware threads, especially in scientific computation applications.

2. **GPU memory types:** On GPUs there are several different types of memories with different access latency. The types of the memories available together with their scopes, lifetimes and declerations are given in the table 1. The global memory has the lowest bandwidth but typically has the largest size. The constant memory allows read-only memory access by the device and provides faster data access in comparison to global memory. The shared memory is per block and is faster than global memory and constant memory, but is slower than the registers. Although the registers are the fastest among other memory types, they can only hold a small amount of data. The local memory is slower than the registers and is used to hold the data that cannot fit into the register.

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
__device__ __shared__ int sharedVar	Shared	Block	Kernel
__device__ int globalVar	Global	Grid	Application
__device__ __constant__ int constVar	Constant	Grid	Application

Table 1: CUDA variable decleration type qualifiers and properties of each type [1]

3. **Assignment Tasks :** First perform all the tasks until part **??** on Ice node with CUDA version of the code. After translating your code to HIP by task **??** you can perform experiments on AMD platforms as well.

(a) **Query device properties :** Use the CUDA code provided to query device properties. Inspect the API documentation to find out the most relevant data from structure cudaDeviceProp. If needed, adjust the number of blocks and threads according to the information found from device properties. Explain the reasoning. *Hint:* `https://docs.nvidia.com/cuda/cuda-runtime-api/structcudaDeviceProp.html#structcudaDeviceProp`

(b) **Global Memory Matrix Multiplication:** Check and understand the implementation of matrix multiplication by using the allocated global memories. Make necessary adjustments to measure the FLOP rates and memory bandwidth utilization of your code on GPU, similar to assignment #5 and assignment #2 Part I task #6.

(c) **Tiled implementation:** Check and understand the implementation of matrix multiplication with tiled implementation. Find the best tile size for best performance. Measure FLOP rates and memory bandwidth utilization.

(d) **Shared Memory :** In your tiled implementation and carry over the tile data to shared memory. Your tiles can be carried over to shared memory by using `__share__` attribute as in 1. Measure FLOP rates and memory bandwidth utilization. Report the observed improvements and explain the reason.

(e) **Host-Device Memory transfer optimisations :** To optimise memory transfers between CPU and GPU CUDA provides zero-copy memory which is a component of CUDA's Unified Virtual Adressing system. Zero copy memory allocations require data to be allocated on the pinned (unpagable) segment of the CPU and the transfer of such allocations to GPU require one copy less than the transfer of the normal device memory allocations. To optimise the memory transfer time between CPU and GPU find the necessary CUDA API call in the documentation and change the memory allocation and transfer calls accordingly. (*Hint:* `https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/`). Once the transfers are optimised compare the total execution time of the application for different dataset sizes to measure effect of zero copy memory.

(f) **Hipify:** `hipify` is a tool provided by the ROCm software stack of AMD. `hipify` is helpful in translating applications using CUDA into a HIP code. It parses the given CUDA code and switches `cuda` prefix in function names from CUDA API by `hip`. It should be noted that the hipify cannot translate things such as NVIDIA PTX code, hard-coded hardware-specific code, or code that are not in the HIP API. Therefore the output code should always be checked before compiling.

- Hipify your code (assignment8.cu and cudautil.h) on ice2 node by `hipify-perl <input-file> -o=<output-file>` . Check if your code is properly hipified and make necessary corrections. Report the parts of code that were not translated by `hipify` tool.
- Repeat the same experiments in all previous parts with your new HIP version of the code on AMD GPUs on rome systems.
- Plot the FLOP rate vs. dataset size plots to compare HIP implementation vs. CUDA implementation on AMD and NVIDIA platforms respectively and compare the two.

(g) **OpenMP Performance Comparisons:** Plot and compare the performance results of the tiled shared memory implementations in HIP and CUDA in part d with the best performing OpenMP GPU offloading implementation from assignment#5 and best performing OpenMP CPU implementation from assignment#4 on Rome and Ice systems.

References

[1] David B. Kirk and Wen-mei W. Hwu. 2010. Programming Massively Parallel Processors: A Hands-on Approach (1st. ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.