DEEC
DEPARTAMENTO DE ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES
TÉCNICO LISBOA

HIGH-PERFORMANCE COMPUTING ARCHITECTURES

Project 3

Abstract: The third lab consists in acceleration of a simple program, by relying on a heterogeneous system composed of a CPU and a GPU. For this purpose, students will exploit the CUDA application programming interface and data-level parallelism to offload computationally intensive (regular) tasks to a GPU.

1 INTRODUCTION

For the development of this work, students should use CUDA, a parallel computing platform and application programming interface model created by Nvidia to enable general-purpose computing on their own GPUs. For the purpose of this work, students will use a heterogeneous machine composed of:

- CPU: Intel Core i7-7700K CPU @ 4.20GHz (4 cores, supporting up to 8 concurrent threads)
- GPU: NVIDIA GeForce GTX 1070 (15 SMs, each with 128 CUDA cores – a total of 1920 CUDA cores)
- Memory: Host 16 GB / device 8GB.

To connect to the machine, students should use cuda1 or cuda2 machines, as in the previous lab.

Machine names: `cuda1.scdeec.tecnico.ulisboa.pt`
`cuda2.scdeec.tecnico.ulisboa.pt`
kmeans source folder: `/extra/acedes/kmeans_source`
kmeans data folder: `/extra/acedes/kmeans_data`

To help the development of the work, students are advised to use the information on the lecture slides, but can also use other information on the web, e.g.,

- https://www.tutorialspoint.com/cuda/cuda_introduction.htm
- David B. Kirk, Wen-mei W. Hwu. "Programming Massively Parallel Processors: A Hands-on Approach", Morgan Kauffman.

NVIDIA also provides multiple examples of CUDA applications, which are normally distributed with the CUDA drivers and API. On the target machine, the CUDA examples are available at:

```
/extra/NVIDIA_CUDA-11.2_Samples/
```

All applications have been previously compiled and are available at:

```
/extra/NVIDIA_CUDA-11.2_Samples/bin/x86_64/linux/release/
```

One particularly useful application is `deviceQuery`, which allows to query the number of available NVIDIA GPUs, as well as its properties. In this case, by running the command:

```
/extra/NVIDIA_CUDA-11.2_Samples/bin/x86_64/linux/release/deviceQuery
```

We get the following output shown in Figure 1.

To develop test programs, students should rely on NVIDIA compiler. For this, students can either: (1) re-write the code using one single file that includes both host and device code; or (2) use multiple files, one for host code, others for device code. In either case, always remember that device code must be placed inside a file with extension .cu. If using a single file (option 1) you can compile the code by using the following line:

```
nvcc -O3 -lm <source code.cu> -o <output binary file>
```

```
/extra/NVIDIA_CUDA-10.2_Samples/bin/x86_64/linux/release/deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 1070"
  CUDA Driver Version / Runtime Version          10.2 / 10.2
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 8120 MBytes (8513978368 bytes)
  (15) Multiprocessors, (128) CUDA Cores/MP:     1920 CUDA Cores
  GPU Max Clock rate:                            1721 MHz (1.72 GHz)
  Memory Clock rate:                             4004 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 2097152 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536),
                                                 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device
simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 10.2, CUDA Runtime Version =
10.2, NumDevs = 1
Result = PASS
```

Figure 1 - Output of the example

`/extra/NVIDIA_CUDA-10.2_Samples/bin/x86_64/linux/release/deviceQuery`

2 WORK PLAN

The objective of this work is to accelerate an application, available at the host folder `/extra/acedes/kmeans_source`. The folder contains the following files:

- `README`: instructions on how to run kmeans
- `kmeans.c, kmeans.h`: main file and header
- `cluster.c, getopt.c, getopt.h, unistd.h`: helper functions and header files.
- `Kmeans_clustering.c`: main computational file **←this is the file you need to accelerate**
- `Makefile`: script for compilation and cleaning. Running "`make compile_all`" will generate three binary files, which will compile your program to run on host with flags -O2 and -O3 (i.e., with and without autovectorization) and to run on the GPU (notice that the current source files do not include any GPU kernel, hence nothing will

be run on the GPU at the moment – this is just a placeholder). Running "`make run_all`" will run all generated binary files, indicating the kernel execution time.

- Folder `~/labs3-data/`: set of pre-generated inputs, with matrices of size 256x256, 512x512, 1024x1024, 2048x2048, and 4096x4096.
- Folders `obj/` and `bin/`: default location of the compiled object files and linked binaries, respectively.

Start by copying the kmeans source files to your own folder location, e.g.:

```
cp -r /extra/acedes/kmeans_source kmeans
```

Move to the created folder and compile the binaries:

```
cd kmeans
make compile_all
```

Run the application with the default input:

```
make run_all
```

Application acceleration:

Students can apply any transformation to the code that may help minimizing the overall execution time. The results (speed-up) should always consider as reference the time of the original source code, when compiled with flag -O3.

To measure the execution time, the `clock_gettime()` function can be used (part of the `<time.h>` library). This allows accurate time measurements with nanosecond precision. The main file already uses this function to measure time.

Application acceleration tips:

1. Start by implementing a basic kernel on the GPU. A simple use case you can try is the addition of two matrices. Make sure you can run a kernel on the GPU and that the result is correct. You can also check the CUDA examples available at the host folder on `/extra/NVIDIA_CUDA-11.2_Samples/`
2. Start by merging the inline function `euclid_dist_2` into the `find_nearest_point` function (file `kmeans_clustering.c`). This will provide you with more opportunities for parallelization (and thus acceleration). You can also try to merge everything onto the main `do..while` loop in function `kmeans_clustering`. Validate that the output is correct.
3. Start by parallelizing the inner-most loop, which corresponds to the original `euclid_dist_2` function. This will not provide you with speed-ups, but ensures you dominate CUDA programming. Validate the output is correct.
4. Identify which loops can be parallelized and identify the parallelism level (total number of iterations in each parallel loop).
5. Try to ensure the loops you accelerate are the ones with the highest parallelism level and avoid multiple data transfers (if you can transfer data to the GPU only once, better).
6. Measure your speed-ups with different datasets.

3 REPORT FORMAT

The project should be completed until April 10, with the up to 6-page report being submitted online (via Fenix), including the corresponding code. Do not forget to clean the directory before submitting your work. The report should follow the template for the IEEE Computer society journals as in the previous assignment:

https://www.ieee.org/publications_standards/publications/authors/author_templates.html

The report should be structured as follows:

- **Abstract**: 100-150 words summarizing the problem, solution, and results.
- **1. Introduction**: Explain the difference (in architecture) between CPUs and GPUs and identify how (and in which cases) GPUs can be used to accelerate the execution of an application.
- **2. Methodology:** Explain how (and why) the original source code was modified to improve the performance.
- **4. Results:** Describe the attained results and draw your conclusions regarding what are the used solutions.
- **5 Conclusions:** Summarize the report, including approach and results.

Please notice that while the report can be shorter than 6 pages, no extra page will be considered, i.e., pages 7 and beyond will be ignored for evaluation.

4 DEADLINES

April 16 (23:59): Submission of the report PDF file and code.

April 20 (23:59): Delayed submissions (with penalty)