TÉCNICO LISBOA

Assignment 6

- Tutorial -

Parallel and Heterogeneous Computing Systems

Computação em Sistemas Paralelos e Heterogéneos (CSPH)

2024/2025

Overview

In this assignment, you will learn how to work with SYCL, using it to write portable, efficient code that can target both CPUs and GPUs. You will start with a quick warmup, parallelizing matrix norm calculation on the CPU. The second assignment involves parallelizing the creation of a histogram, which involves the partitioning of data into separate bins. In the third assignment, you will parallelize the K-Means Clustering algorithm, which is commonly used in unsupervised machine learning tasks, in both CPU and GPU. Finally, the last step is to parallelize the calculation of a covariance matrix from a dataset, which is also a common task in dimensionality reduction algorithms for machine learning. Serial implementations are provided for all of these, so you only need to create the appropriate SYCL kernels, target the correct devices, and perhaps shift some things around to get things optimized a little further. Good luck!

Deadlines and Submission

NO SUBMISSIONS! This work serves as a tutorial and preparation – NO EVALUATION. It is intended for your practice and learning via examples and tutorials. This assignment provides a set of exercises for you to practice! You can resolve them at our lab session or at home. Surely, you can discuss your solutions and doubts with the teacher of your labs in our next lab session. Again, there is no submission for this part of the work and your solutions will not be graded!

You can consider these examples as a representative set of problems to get you prepped for the work to be developed in the class (which will be evaluated). Those assignments are not yet provided to you!

Environment Setup

For this assignment, you should run the final versions of your programs on the LSD2 machines. As you know, the machines are equipped with an NVIDIA RTX 4070 Ti and an Intel i7-11700. After accessing, you need to setup the Intel oneAPI environment by running the following command on the terminal

```
source /opt/intel/oneapi/setvars.sh
```

You can check afterwards with the command `sycl-ls` that the SYCL runtime can identify both the Intel CPU and the NVIDIA GPU from the lab machines.

Note: If you would like to run the programs in this assignment on your own machine, you should download the Intel oneAPI base toolkit (see here for Linux and Windows installation: https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit-download.html). After installation, you can run the previous commands as usual. If things get too confusing, just use the lab machines.

The assignment starter code is available on the course webpage (section: Labs).

Background: Learning SYCL

In this assignment, we'd like you to use SYCL for portable CPU and GPU parallelization. SYCL is an API that defines abstractions to enable heterogeneous device programming, an important capability in the modern world which has not yet been solved directly in ISO C++. You can use SYCL to execute parallel application on CPUs, GPUs, and even FPGAs (although FPGAs are not covered here). Besides our slides from theoretical classes, SYCL is well documented online and here are some useful links to get you started:

- SYCL: https://www.khronos.org/sycl/
- A SYCL cheat sheet: https://www.khronos.org/files/sycl/sycl-2020-reference-guide.pdf

Part 1: Parallelizing the Frobenius Norm

As a simple warm up exercise, your first goal is to parallelize the calculation of the Frobenius Norm of a matrix in `problem1.cpp`. The algorithm is very simple: the norm is calculated by summing the squares of all numbers in the matrix and, afterwards, taking the square root of the accumulated sum (see https://mathworld.wolfram.com/FrobeniusNorm.html). In `problem1.cpp`, you can find a serial implementation for this algorithm, as well as a SYCL implementation. This SYCL implementation calls a parallel kernel (through the `parallel_for` API) over a 2D ND-range with `NxN` (where `N` is the matrix size) work-items, where each work-group has `32x32` work-items. To accumulate the sums of all matrix cells, atomic operations are used. The execution time profiling is done after the kernel runs (to ensure that the kernel is finished, we use `event.wait()`, where this event is the kernel submission). After compiling the code with `make`, you can run it with

```
./frobenius N
```

where `N` is the matrix size. This will display the average runtime of each implementation, across 5 runs. If you omit the matrix size, a default of 1024x1024 will be used.

1. Try changing your ND-range sizes to explore the impact on the kernel's execution time.

2. As discussed in the theoretical lecture, atomics in SYCL can be replaced with reduction. Implement the reduction on the SYCL kernel and test the algorithm on various matrix sizes (note that if your implementation is incorrect, an error will be printed to the terminal).

3. Finally, modify the device selector in `defaultQueue` to be `sycl::gpu_selector_v` (to select a GPU) to run the SYCL kernel on the GPU. Do not forget to also be careful with the memory allocations and initializations (e.g., if you allocate the matrix on the device, you cannot initialize it on the host: you will have to do `memcpy`).

Part 2: Parallelizing a Histogram

In the second exercise, your goal is to parallelize the creation of a histogram in `problem2.cpp`. Given an array of `N` elements from 0.0 to 1.0 and `B` bins, the serial implementation (`serialHistogram`) calculates the number of elements that fall within each bin. After compiling the code with `make`, you can run it with

```
    ./histogram N B
```

where `N` is the number of data points and `B` is the number of bins. This will display the average runtime of each implementation, across 5 runs. If you omit the number of bins, a default of 10 will be used. If you also omit the number of data points, 1024x1024 datapoints will be used.

Your job is to complete the SYCL kernel (in `problem2.cpp`, the kernel is in the function `syclHistogram`) to parallelize the histogram creation. For this particular problem, think about how you can use atomics, synchronization, and local memory to count the elements within each bin. Regarding the local memory allocation, remember the theoretical class and how the `local_accessor` is instantiated. Regarding atomics, use the `atomic_ref` class. When an instance of `atomic_ref` is created, four parameters are passed: the data type (e.g., `int`) memory order (`sycl::memory_order::relaxed`), memory scope (e.g., `sycl::memory_scope::work_group`, `sycl::memory_scope::device`) and address scope (e.g., `sycl::access::address_space::local_space`, `sycl::access::address_space::global_space`). As an example, doing atomic updates to the first integer in a local memory array (`localmemory`) would require instantiating a variable as follows:

```
sycl::atomic_ref<int,  sycl::memory_order::relaxed,  sycl::memory_scope::work_group,
sycl::access::address_space::local_space> atomic_bin(localmemory[0]);
```

If, for example, you want to add a value to `localmemory[0]`, you should write

```
atomic_bin.fetch_add(value);
```

Regarding synchronization, the `barrier` function can also be called with a parameter which defines the memory scope in which the barrier is waiting. For example, if we want to synchronize items after accesses to local memory, one would write

```
item.barrier(sycl::access::fence_space::local_space)
```

where `item` is a work-item in the kernel. After writing your implementation, test it with different values of `N` and `B` (if your code differs from the serial implementation, the terminal will print an error). After you're sure that your implementation works on CPU, feel free to also try it on GPU (be careful with the memory accesses and allocations, with the queue device selector, and with the size of your work-groups).

Part 3: Parallelizing the K-Means Clustering Algorithm

For the third exercise, you will parallelize a widely used clustering algorithm known as K-Means Clustering in `problem3.cpp`. This algorithm is simple: given a set of N data points and C centroids, the first part of the algorithm is to find, for each data point, which is the closest centroid (using Euclidean distance). The second part is to update each centroid according to the number of data points that are associated to it (if you have doubts, see https://en.wikipedia.org/wiki/K-means_clustering and also check the serial implementation - `serialKMeans` - to get a feel for the computations). This process is done iteratively until the centroids converge, but for this particular exercise, we will only consider a single iteration of this algorithm.

For this algorithm, we will consider that we are working with 2D data points (see the `Point` structure in `problem3.cpp`). After compiling the code with `make`, you can run it with

```
./kmeans N C
```

where `N` is the number of 2D data points and `C` the number of centroids. If you omit the number of centroids, a default of 10 will be used. If you also omit the number of data points, a default of 1024 is used. Your job is to complete the SYCL kernel in the `syclKMeans` function. Note that, this time, there are *two* kernels to complete, so think about how you can separate the computations of the algorithm in such a way that you are able to obtain the correct result in the end. After writing your implementation, test it with different values of `N` and `C` (if your code differs from the serial implementation, the terminal will print an error). After you're sure that your implementation works on CPU, feel free to also try it on GPU (be careful with the memory accesses and allocations, with the queue device selector, and with the size of your work-groups).

Part 4: Parallelizing a Covariance Matrix calculation

Your final task is to parallelize the calculation of a covariance matrix in `problem4.cpp` (a typical step in some dimensionality reduction algorithms, such as Principal Components Analysis). The principle is simple: given a dataset, `data`, of `N` rows and `D` columns, the first part of the algorithm is to center the data (i.e., for each column, you calculate the mean value and, afterwards, you subtract the mean from each value in that column. The second part of the algorithm is to calculate the covariance matrix. The element `ij` of this matrix is given by

$$cov_{ij} = \frac{1}{N-1}\sum_{k=1}^{N}(data_{ki} - m_i)(data_{kj} - m_j),$$

Where cov_{ij} is the position `ij` of the covariance matrix, `N` is the number of rows in the dataset, m_{i} is the mean value of the `i-th` column in the dataset and data_{ki} is the value of the dataset in row `k` and column `i` (if you have doubts about how the algorithm is structured, see https://en.wikipedia.org/wiki/Covariance_matrix and also check the serial implementation - `serialCovMat` - to get a feel for the structure of the computations). After compiling the code with `make`, you can run it with

```
./covmat N D
```

where `N` is the number of rows in the dataset and `D` is the number of columns. The `main` function creates a matrix of random numbers of size `NxD` (our dataset) and an empty covariance matrix of size `DxD`. You can also call the program without specifying `N` and `D` (the default values are 1024 and 1024).

1. Complete the SYCL kernel that implements the parallel calculation of a covariance matrix. As a first implementation, try to do it all in a *single* kernel (in `problem4.cpp`, the kernel is in the function `syclCovMatSingle`). The serial and the SYCL implementations should report the same covariance matrix, since they start from the same dataset. Therefore, if both solutions differ, it is likely that you did a mistake in your kernel implementation (the terminal will also print you an

error message). After writing your implementation, try different values of `N` and `D` and see what speedups you obtain.

2. Now, try to implement the same algorithm, but using `two` kernels (in `problem4.cpp`, the kernels are in the function `syclCovMatTwo`). After verifying the correctness of your results, try different values of `N` and `D` and see what speedups you obtain.

3. Due to how the dataset is structured, the memory accesses are not coalesced, which will hurt your performance. Think of a way of reorganizing your dataset to improve memory accesses and see if performance improves (**Hint**: think back to matrix multiplication and what was the first approach to improve the naive implementation).

4. As a last exercise, try to run your SYCL code on GPU (be careful with the memory accesses and allocations, with the queue device selector, and with the size of your work-groups).