TÉCNICO LISBOA

Assignment 5

- 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 OpenMP, using it to parallelize code on the CPU and to offload kernels to the GPU. You will start with a quick warmup, parallelizing dense matrix-matrix multiplication on the CPU and GPU. The second assignment involves parallelizing two simple kernels with differing properties and complexities. In the third assignment, you will parallelize an iterative stencil loop[1] on the GPU. Finally, the last step is to parallelize Page Rank on the CPU. Serial implementations are provided for all of these, so you only need to decorate the loops with the appropriate OpenMP constructs, 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.

Note: If you would like to run the programs in this assignment on your own machine, you should also download graphs and reference solutions at the following link: https://drive.google.com/file/d/1--5RpTl6N2Phl569fJk9pUFYy9y-tyJG (beware: it's a 4GB file). Additionally, you will likely need to install a package that extends `gcc` to support offloading to your machine's GPU. Firstly, determine your `gcc` version with the `gcc -v` command. Then, depending on your GPU, install the following package:

```
gcc-<YOUR_GCC_VERSION>-offload-<YOUR_GPU_ISA>
```

If you have an NVIDIA GPU, the corresponding ISA is `nvptx`, while the ISA of AMD GPUs is `amdgcn`. For example, the lab machines have gcc 11 installed and have an NVIDIA GPU. As such, they require the `gcc-11-offload-nvptx` package. If you have gcc 12 and an AMD GPU, you need to install package `gcc-12-offload-amdgcn`. If things get too confusing, just use the lab machines.

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

[1] https://en.wikipedia.org/wiki/Iterative_Stencil_Loops#Example:_2D_Jacobi_iteration
[2] https://en.wikipedia.org/wiki/PageRank

Background: Learning OpenMP

In this assignment, we'd like you to use OpenMP[2] for multi-core parallelization and GPU offload. OpenMP is an API and set of C-language extensions that provides compiler support for parallelism. You can use OpenMP to tell the compiler to parallelize iterations of for loops, to manage mutual exclusion, and to send your kernels and data to the GPU. Besides on our slides from theoretical classes, OpenMP is well documented online and here are some useful links to get you started:

- The OpenMP 5.0 specification: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf
- An OpenMP cheat sheet: openmp.org/wp-content/uploads/OpenMPRef-5.0-0519-web.pdf

Part 1: Parallelizing GEMM

As a simple warm up exercise, your first goal is to parallelize general matrix-matrix multiplication using OpenMP, both on the CPU and the GPU. `gemm.cpp` has a basic serial implementation, as well as two placeholders for you to parallelize. Notice that the B matrix is already transposed, which is one of the simplest optimizations you can do for this algorithm. Your goal is to use the OpenMP pragmas you have learned so far, you should not need to change anything else about the code. After compiling your code with `make`, you should be able to run the application:

```
./gemm <matrix_size>
```

This will display the average runtime of each implementation, across 5 runs. At the start, this should look roughly equal for all three, given they all run without parallelism (see `gemm_sequential` function). If you omit the matrix size, a default of 1024x1024 will be used, which is a good size for the initial tests.

Note that here the matrices have already been expressed as simple arrays (for details, see `matrix.hpp` in the `common` folder). This structure is made to help you map the matrices to the GPU memory.

Your job: In `gemm_parallel_cpu`, simply decorate the loop(s) to parallelize it across multiple CPU threads. In `gemm_parallel_gpu`, you should do the same, now offloading the kernel to the GPU. Watch which matrices are inputs/outputs carefully, and map them accordingly, to avoid transferring redundant data to/from the device.

With the default matrix size of 1024, you should achieve a decent speedup with the CPU parallel version, and an even bigger speedup on the GPU. Try running it with other matrix sizes, such as 2048, 512, and 32. What does that tell you about the overhead of the two parallel implementations?

Part 2: Multiple Kernels

In the second exercise, in `multikernel.cpp`, you have three complete implementations of a three-kernel workload. The first kernel performs matrix multiplication, while the second kernel raises the elements of the X array to the power of the elements in Y, storing them in Z. The third kernel takes the results of the previous two, adding the elements of Z to every row of C. The aforementioned three

[2] https://www.openmp.org/specifications/

implementations are: *1)* a simple serial version, *2)* a CPU parallel version, and *3)* a GPU parallel version. You can run this application as follows:

```
./multikernel <matrix_size> <vector_size>
```

For this task, you can just run `./multikernel`, which will omit the additional arguments and use the default sizes of 1024 and 10000000. Your goal is to optimize this application further than any of the other implementations, modifying the code in `mk_parallel_opt` to achieve a superior speedup. Note the memory and computational complexity of the three kernels, and recall what was discussed in the lectures about these, and how they should help you decide whether a kernel should be offloaded. Since one of the parallel implementations uses just the CPU, and the other just the GPU, what can you do differently/better? Try sketching this application's dependency tree and think about how you can distribute the kernels to maximize your use of the hardware.

Part 3: Iterative Stencil Loop

For the third exercise, you will parallelize an iterative stencil loop on the GPU. This method is useful for a wide range of applications, such as simulating the transfer of heat across a solid, or simulating fluids.

Take a look at the `jacobi_sequential` function. In the while loop, you will notice the two main parts (kernels). The first part of the method iterates through all elements of the grid, updating their value based on the values of their neighbours using a stencil. The process is similar to convolution, except the grid doesn't change size at the end! The elements at the edge pose a bit of a challenge, as they don't have neighbours on all sides. As such, we present you with a slightly simpler version, in which only then central elements of the grid are updated. The cells on the side edges are always kept at 1, while the edges at the top and bottom are kept at 0. You can think of it as simulating a fluid junction, with inlets on the sides that feed it with a constant pressure, and outlets at the top and bottom.

The method's second part (kernel) copies the values back to the grid, calculating the maximum change across all cells, known as the residual. Once the residual reaches a low enough value, the simulation has converged, and the stop condition is met. In order to better visualize this process, check the diagram in the provided reference[1] and the source code, the comments should help understand it.

As before, you can compile the code with the use of the `Makefile`, and run it with the following command:

```
./jacobi <grid_size>
```

In your testing, you can omit the grid size, which will use the default of 20000. If you are just starting out and are testing for correctness, try a smaller grid size to speed up your tests. A serial implementation is provided, as well as a complete CPU parallel implementation. The former is not run due to how long it takes, but you can uncomment it in the `implementations` vector if you are curious (see the `main` function). The CPU parallel implementation should give you a few hints on how to parallelize it for the GPU. You would like to offload both kernels, so recall what you should do to keep the data on the GPU, avoiding redundant transfers. Don't forget to map the residual variable, which you need on the host to determine if the stop condition is met.

You should achieve a slight speedup compared to the reference implementation. The two implementations should report the same number of iterations, if your GPU implementation differs, you likely made a mistake! Given the fact you have two kernels, with some restructuring it should be possible to perform one on the host and one on the device, in parallel. Why would this not be a good idea for this application? Hint: it's not an issue of correctness, instead look at the number of iterations the program reports and think about what would need to happen each iteration in a heterogeneous approach.

Part 4: Parallelizing Page Rank

Background: Representing Graphs

The starter code for the page rank portion of this assignment operates on directed graphs, whose implementation you can find in the `common` folder, files `graph.h` and `graph_internal.h`. We recommend you begin by understanding the graph representation in these files. A graph is represented by an array of edges (both `outgoing_edges` and `incoming_edges`), where each edge is represented by an integer describing the id of the destination vertex. Edges are stored in the graph sorted by their source vertex, so the source vertex is implicit in the representation. This makes for a compact representation of the graph, and also allows it to be stored contiguously in memory. For example, to iterate over the outgoing edges for all nodes in the graph, you'd use the following code:

```
int numNodes = g->num_nodes;
for (int i=0; i<numNodes; i++) {
  int start = g->outgoing_starts[i];
  int end = (i == numNodes-1) ? g->num_edges : g->outgoing_starts[i+1];
  for (int v=start; v<end; v++)
    printf("Edge %u %u\n", i, g->outgoing_edges[v]);
}
```

You can also use the following code, which makes use of convenient helper functions defined in `graph.h` (and implemented in `graph_internal.h`):

```
for (int i=0; i<num_nodes(g); i++) {
  // Vertex is typedef'ed to an int. Vertex* points into g.outgoing_edges[]
  const Vertex* start = outgoing_begin(g, i);
  const Vertex* end = outgoing_end(g, i);
  for (const Vertex* v=start; v!=end; v++)
    printf("Edge %u %u\n", i, *v);
}
```

Try to convince yourself that these two codes are functionally equivalent, i.e., they do the same thing!

The Real Thing

Your final step is to optimize a basic version of the well-known page rank[3] algorithm on the CPU (no GPU in this problem!). Take a look at the serial implementation provided to you in the function `pageRank()`, in the file `pagerank.cpp`. You should optimize this function, parallelizing the code with OpenMP.

You can run your code and check correctness against the reference solution by using:

```
./pagerank <graphs_directory>/<graph_name>.graph
```

If you are working on the lab machines, we've located a copy of the graphs and solutions directory at **`/home/extra/graphs`**. Some interesting real-world graphs include: `com-orkut_117m.graph`, `soc-pokec_30m.graph`, `com-youtube_3m.graph`, `soc-livejournal1_68m.graph` and `ego-twitter_2m.graph`. Your useful synthetic, but large graphs include `random_500m.graph` and `rmat_200m.graph`. There are also some very small graphs for testing (such as `tiny`, `grid4x4` etc.)

By default, the `pagerank` program runs your page rank algorithm with an increasing number of threads (so you can assess speedup trends). However, since runtimes at low core counts can be long, you can explicitly specify the number of threads to only run your code under a single configuration.

```
./pagerank <graph_path> 16
```

Identify the loops you can parallelize and the dependencies between them. Is there some work that you can perform more efficiently? Don't be afraid to move things around! Keep in mind the fact most graphs are not uniform: the number of edges per node tends to be unpredictable, meaning a naïve approach can lead to unbalanced work sharing. Recall the various options you have for scheduling and choose one which will help balance the partitioning.

Note: The code that was given to you is very slow on purpose. In other words, if you try to run it on a single core for a very big graph, it might not finish by the end of this lecturing period ☺ However, your implementation will crunch it in a much more reasonable time!

[3] https://en.wikipedia.org/wiki/PageRank