CS 250B: Modern Computer Systems

Lab 1: Stencil codes for heat dissipation simulation

Due: 2022-05-27

Overview

In this lab, you will take a naïve software implementation of a scientific simulation algorithm, and achieve higher performance on a given, off-the-shelf computer system. The target application is stencil codes for two-dimensional heat dissipation simulation.

The software will be given two 2-dimensional matrices, each storing the temperature and the thermal conductivity of a location on a two-dimensional plane. The purpose of the software is to simulate the process of heat dissipation over N time-steps, by repeatedly executing the 5-point stencil kernel for heat dissipation on all points in the matrix.

The provided codebase includes a naïve implementation of the simulator, which has very bad memory and computation utilization. Your goal is to improve it. The baseline code should be descriptive enough to provide you with all the information about the algorithm.

Provided Material

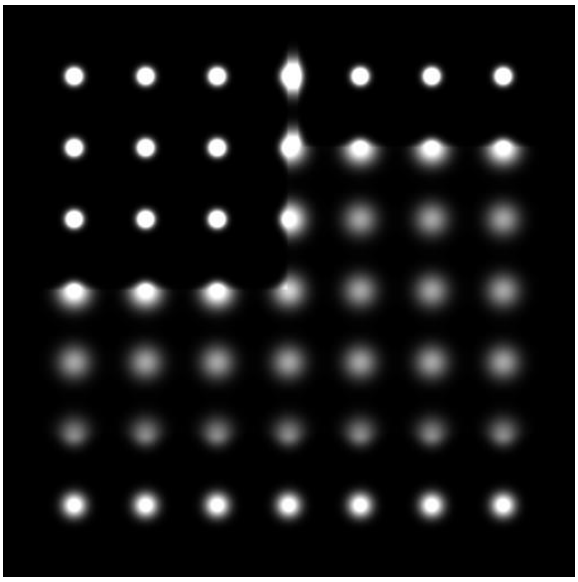
There are six files in the provided source directory, which perform various functions including initial data generation (datagen.cpp), the actual stencil computation (main.cpp, stencil.cpp), and data visualization (visualize.py). The file of main interest is ***stencil.cpp, which is the only file you should modify***.

An example data file is also provided (init.dat), which is what datagen.cpp will generate without modification.

Compilation and Execution

Simply execute “make” to generate two executables, obj/datagen and obj/stencil. datagen is not important right now unless you want to experiment with different initial settings. Since an “init.dat” is already provided, simply execute “./obj/stencil 4096”, to execute the stencil for 1024 time-steps. It will take a few minutes to finish.

Once execution is done, it will have created an output file, “output.dat”. This file can be viewed visually be running the visualizer script, “visualize.py” using python: “python3 visualize.py” without arguments. The output will be something like this:



In this visualization, the darker pixels correspond to lower temperature, and the brighter pixels correspond to higher temperature.

In the provided settings, there are regions of low thermal conductivity in the upper left corner, the upper right corner, and the bottom eighth of the simulation area, and this can be seen from the rendering output. If you give a longer time step, you will be able to see what happens further in the future. This image will also be automatically saved to “render.png”.

Suggested Approaches

You will implement the function, “step_optimized” in stencil.cpp. This function takes 7 arguments. The first 5 of the 7 arguments are the same as the provided “step_naive” in main.cpp and should be self-explanatory. Of the other two arguments, “threads” is the maximum number of threads this function is allowed to spawn (this is given as a command-line argument), and “substeps” is the number of simulation steps this function should execute. Having sub-steps allows for *cache optimizations such as temporal blocking*.

There are three major points of improvement in this application: Cache optimization, multithreading, and SIMD instructions. The suggested process is to start with cache optimization, then move on to multithreading, and then SIMD. Furthermore, it is suggested to start with fixed-size blocks for cache optimization, instead of trying to do cache-oblivious algorithms first, as it may be tricky to get right the first time.

First, I suggest you try rectangular, temporal blocking, instead of the trapezoidal blocking introduced in the cache-oblivious algorithms lecture. Once that works, you can try the more complicated trapezoidal one.

Code Description

The variables temp and temp2 are scratchpad regions. The input to the function is in temp in a dense, row-major array format, and the state of the grid after “substeps” number of stencil iterations should also be stored in “temp”. The size of each allocated memory region is “`sizeof(float) * width *`

`height * SUBSTEP`". The input and output data are assumed to be in the lower addresses of the allocated memory region.

You may or may not need to use the additional scratchpad region "temp2", depending on your approach. Just make sure the output is located in the lower "`sizeof(float) * width * height`" bytes of "temp".

Some Assurances

- "substeps" will always be an even number.
- Width and height will always be a power of two, larger than 128.
- Thread count *may not be a power of two.*

Execution

You can execute your custom implementation using the following command:

./obj/stencil 1024 4 init.dat n

The last argument "n" specifies you want to execute your new implementation instead of the naïve one. The other arguments are: number of steps, threads, and input data, respectively.

If your system has "perf" installed, you can check for cache access characteristics by executing something like "sudo perf stat -e cache-references,cache-misses,L1-dcache-loads,L1-dcache-load-misses ./obj/stencil 1024 4 init.dat n"

More information about perf can be found here: https://perf.wiki.kernel.org/index.php/Tutorial

Submission and Grading

Please submit only "***stencil.cpp***".

Full marks will be given if it can outperform a multithreaded, rectangularly blocked implementation with statically-sized blocks.

We will have a derby at the end with anonymized names, so please try your best!