TDDD56 Multicore and GPU computing
Lab 3: High-level parallelism with skeleton programming

August Ernstsson
august.ernstsson@liu.se

November 18, 2020

Contents

1 Introduction and motivation

In this lab, you will work with a *high-level parallel programming* framework. High-level parallel programming aims to abstract and simplify architecture-specific properties of parallel computer systems, and often targets multiple types of parallelism such as either multi-core CPU or GPU-based systems, or sometimes hybrid execution on both at the same time. High-level interfaces are also crucial for programming cloud computer systems, for example.

In earlier labs you explicitly described in your programs how multiple processor cores cooperate by using constructs such as threads and locks. The similarities to a sequential program performing the same task are relatively small. With high-level parallel programming, the goal is to write programs with sequential semantics, that is, there is no explicit parallelism control in the program code (such as locks). The framework, language, or compiler that provides high-level parallelism is instead responsible to extract parallelism from the sequential semantics in the program. To do this reliably, the functionality offered in the interface is typically reduced to a selection of common patterns.

1.1 Skeleton programming

Skeleton programming is an approach to high-level parallel programming inspired by functional programming, specifically higher-order functions. In functional programming, operations on data is typically expressed by functions like *map* and *reduce* (or fold), where the arguments are not only the data to operate on but also the operation itself, as a function object. Map and reduce are examples of *data parallel* skeletons; there also exist task-parallel skeletons, but those are outside the scope of this lab. In skeleton programming, map and reduce are examples of *skeletons*. The function objects are called *user functions*.

For example, the map operation can accept the data set $[1, 2, 3, 4]$ and the operation $x \rightarrow x^2$. The output will be the data set $[1, 4, 9, 16]$. As each individual element can be computed independently of the others, this is a prime target for parallelization; in fact, the map operation is "embarrassingly" parallel. Reduce is not as trivial, but parallelization can still be done. Given the data set $[1, 2, 3, 4]$ and the operation $x, y \rightarrow x + y$, the output will be 10. Naively summing the data set from left to right will cause sequentialisation due to the dependencies; however, note that we can compute the partial sums $1 + 2$ and $3 + 4$ in parallel, even with the final addition improving from three to two time steps in total. Applying a tree structure like this will make the time complexity logarithmic, assuming that enough parallelism is available in the hardware.

By performing tree-structured reduction, we assume that the operation is *associative*, which is true for addition. This is another example of how high-level parallel frameworks restrict the interface: non-associative reductions cannot be parallelized in general, so most frameworks will not accept those. Even with map we made several assumptions, such as that the operator does not have any observable side effects.

1.2 SkePU

SkePU is a C++ framework for high-level parallel programming implementing the skeleton programming approach. It is a research tool developed at LiU with support for multi-core CPU and GPU backends, meaning that the programs can run either on the CPU or a GPU. SkePU can be made to automatically select which backend to use at runtime, depending on the operation and data size. To facilitate switching between CPU and GPU, SkePU implements smart containers which will automatically manage data across the CPU/GPU boundary in an optimized manner. SkePU supports the following skeletons:

- Map
- Reduce

- MapReduce
- Scan
- MapOverlap

As mentioned earlier, skeletons need a user function (i.e., an operator) to be used. User functions in SkePU are normal C++ functions at first sight, but care has to be taken when writing these functions. User functions should be self-contained, which means free functions without external dependencies or side effects[1]. Additionally, the code inside a user function must not use any C++-specific features (syntax or standard library), the reason being that SkePU transforms this code to the GPU languages OpenCL and CUDA, which has C as a least common denominator.

Below is an example of a user function in SkePU.

Listing 1: User function in SkePU

```
float addOneFunc(float a)
{
        return a+1;
}
```

It is used to instantiate a skeleton like below. The number `<1>` indicates a unary Map, that is, elements will be picked from only one container argument later. The use of the `auto` type specifier is mandatory, as the actual type of a SkePU skeleton is implementation-defined.

Listing 2: Skeleton instantiation in SkePU

```
auto addOneMap = skepu2::Map<1>(addOneFunc);
```

The skeleton is called/applied on SkePU containers like below.

Listing 3: Calling a skeleton instance in SkePU

```
addOneMap(res, input);
```

More information about SkePU can be found in the user guide, see the course lab webpage.

1.3 Backend specification

You can control which backend is used by SkePU by providing a backend specification (`BackendSpec`). This is set up in the lab skeleton files by reading a string argument from the command line:

- `CPU` for a sequential backend,
- `OpenMP` for a multi-threaded backend,
- `OpenCL` for an OpenCL-based GPU backend,
- `CUDA` for a CUDA-based GPU backend (disabled by default in the labs).

The respective backends needs to be enabled when building the program to work at run-time. (See `Makefile.include`).

It is also possible to set parameters for each backend. In the lab, you will only need to use `spec.setCPUThreads(<integer value>)` which controls the number of threads used with the OpenMP backend. If this is not specified, it will use the default from OpenMP, which tends to be equal to the number of available cores.

[1] Text logging from a user function can be useful for debugging and will work in certain cases.

2 Exercises

The lab consists of three parts:

1. Dot product implementation (warm-up).
2. Averaging blur filter in three variants.
3. Median filtering.

2.1 Dot product

Implement a dot product computation using the MapReduce skeleton in SkePU. Use the `dotproduct.cpp` file as a starting point, and look at `addone.cpp` for inspiration. Remember that the AddOne program uses a unary Map (one input vector), while dot product requires a binary Map (two input vectors). (Where do you specify this?)

Then make another implementation, this time using separate Map and Reduce skeletons. You will need to add another Vector container to the program for this. Why?

Compare the performance of your MapReduce and Map + Reduce implementations. Also compare performance of the MapReduce implementation with different backends.

Choose a suitable vector size for your measurements, so the computation at least takes on the order of milliseconds to execute. You can also test multiple input sizes.

Question 1.1: Why does SkePU have a "fused" MapReduce when there already are separate Map and Reduce skeletons?
Hint: Think about memory access patterns.

Question 1.2: Is there any practical reason to ever use separate Map and Reduce in sequence?

Question 1.3: Is there a SkePU backend which is always more efficient to use, or does this depend on the problem size? Why? Either show with measurements or provide a valid reasoning.

Question 1.4: Try measuring the parallel backends with `measureExecTime` exchanged for `measureExecTimeIdempotent`. This measurement does a "cold run" of the lambda expression before running the proper measurement. Do you see a difference for some backends, and if so, why?

2.2 Averaging filters

In this exercise (and the next) you will implement image filters. The filters operate on Matrix containers with the data-parallel SkePU skeleton MapOverlap, a variant of Map but where a region of the container is accessible instead of just one element.

Study and test out the averaging filter in `average.cpp`. For each pixel in the output image, it computes the average pixel color of the pixels in an surrounding region in the input image. This produces a rough blurring effect.

Make sure you understand how the MapOverlap skeleton is used to accomplish this task. The region radius is set inside `main` and appears in the user function as the parameters `oj` (x-radius) and `oi` (y-radius). *Note: Use only radius values in the range from 2 to 50.* Note that pixels are represented as three bytes in the matrices, so the overlap will be larger in the x-dimension. See figure 1 for an illustration of the image as seen by the user function.

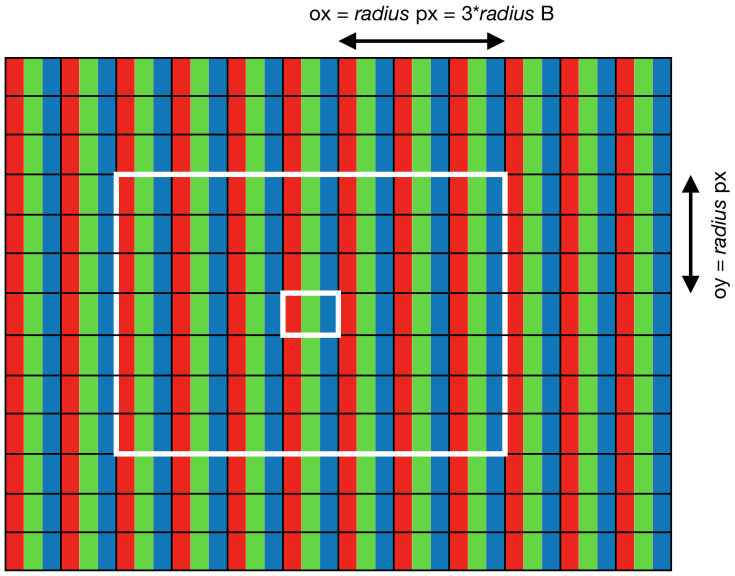

Figure 1: Memory layout of the image for the 2D filters. The inner white region represents a pixel in the output image, with the outer white region is the surrounding region in the input image (the "overlap") Each colored rectangle (subpixel) is one byte of data.

2.2.1 Separable averaging filter

Averaging filters have a property known as *separability*. This means that the two-dimensional filtering operation can be separated into two one-dimensional filters instead [2] SkePU also supports one-dimensional MapOverlaps over matrices, so your task is to reimplement the averaging filter in this way. The source file already has a few starting points to help you out.

Note: The edge pixels of the output will not be exactly the same because of differences in the edge handling. The "safe" region (inset by the filter radius from the edge) should be identical, however.

2.2.2 Gaussian filter

The averaging filter results in very rough-looking output images. To make a nicer-looking blur effect, the Gaussian blur filter has non-uniform pixel weights, so that nearby pixels have a larger contribution to the result. Implement a Gaussian filter with your separable filter as a starting point (but keep them separate in the program). The function `sampleGaussian` generates a SkePU Vector of weights (of size $2 \times radius + 1$, symmetric across the middle).

Question 2.1: Which version of the averaging filter (unified, separable) is the most efficient? Why?

2.3 Median filter

For the last assignment, your task is to implement a median filter with the MapOverlap skeleton. The median filter works just like an averaging filter, but instead of using the average pixel value of a region, the output is the median value. The challenge therefore is to sort the pixels in the region in such a way that the median value can be identified.

[2] The average pixel value in the region is also the average value of all the average values for each row. $avg_{2D}(region) = avg_{rows}(avg_{columns}(region))$. See appendix A for some more motivation.

(a) Original image.

(b) Averaging filter of radius 16.

(c) Gaussian filter of radius 16.

(d) Median filter of radius 16.

Figure 2: Example output of the image filters.

Some things to take into consideration:

- SkePU user functions cannot have side effects, so it is not possible to dynamically allocate memory in one.
- It is possible to call other functions from SkePU userfunctions, but you are recommended to avoid that for this assignment.
- Consider the problem domain (pixels) and framework restrictions when implementing the sorting. What sorting algorithms are suitable to use?
- Consider trying different approaches to show and contrast in the demo.

Question 3.1: In data-parallel skeletons like MapOverlap, all elements are processed independently of each other. Is this a good fit for the median filter? Why/why not?
Question 3.2: Describe the sequence of instructions executed in your user-function. Is it data dependent? What does this mean for e.g., automatic vectorization, or the GPU backend?

3 Practicalities

3.1 Lab skeleton structure

This section describes how to use and compile the skeleton source files provided and introduce a few key points you need to know.

data/ This directory contains PNG images of various sizes to be used for the filtering exercises. Note that the image "gray" is grayscale and only contains one color channel per pixel; the others are RGB.

reference/ Contains reference output of the test image using the average, gaussian, and median filters with a radius of 16. See figure 2.

include/ SkePU run-time source files. `make` will handle everything to do with this directory for you.

bin/ Location of the binary files generated by SkePU, but also the generated source files from source-to-source compilation.

Makefile The main makefile. If you want to add a program (e.g., duplicating an existing one), add the name to the `PROGS` list (excluding `.cpp` extension). This is enough for single-file programs. If there are additional files (as for the image filters), you need to also add a make recipe as per the existing ones later in the makefile.

Makefile.in Build parameters specific to the current system. This should already be set up for the lab rooms. If you want to run the labs somewhere else (**not recommended**) you will need to make changes here.

Makefile.include Build options for e.g. generated backends and optimization level. Change the `BACKENDS` list to control which backends are generated. You can also enable or disable debug logs which shows memory allocations and skeleton executions on various backends.

addone.cpp A simple demonstration of SkePU programming. This program uses a Map skeleton to increment each element of a randomized vector by 1.

dotproduct.cpp Code skeleton for exercise 1.

average.cpp Code skeleton for exercise 2.

median.cpp Code skeleton for exercise 3.

support.[h cpp], lodepng.[h cpp] Additional source code used by `average.cpp` and `median.cpp` for reading PNG images to SkePU containers.

3.2 Compiling and running

To build a program, use `make bin/<program name>`. For example, `make bin/mapreduce`.

This will first run the SkePU source-to-source compiler (see the output in the terminal) and automatically then call `g++` on the generated source files.

To run the program, just type `bin/<program name> <parameters>`.

If you get a message from the OpenCL library about version information at run time: this can be ignored.

4 Before the lab session

Before coming to the lab, we recommend you do the following preparatory work

- Familiarize yourselves with SkePU and its syntax, e.g., from the lesson slides.
- Figure out an efficient way to implement the median selection part of the median filter. *Hint: Utilize properties of the domain (we are working with image pixels!).*
- Study the questions in the lab compendium and try to come up with reasonable answers or comments.

5 During the lab session

Take profit of the exclusive access you have to the computer you use in the lab session to perform the following tasks

- Implement all the parts of the lab and test them thoroughly.
- Measure the performance of your programs when stated in the exercise description.
- Experiment with different problem sizes and backends.

6 Lab demo

Demonstrate to your lab assistant the following elements:

1. Show and explain the source code of your programs.
2. Answer the questions in the lab compendium.
3. Explain in particular your implementation of the median filter.
4. Discuss the advantages and disadvantages of high-level parallel programming. *Hint: Compare ease of use, performance, maintainability, portability...*

7 Investigate further

Feel fre to use the SkePU tools to experiment with high-level parallel programming. You can for instance:

- implement a separable edge-detection filter based on `average.cpp`;
- go back to the topic of lab 1 and generate a Mandelbrot fractal with the Map skeleton and compare the performance to your lab 1 implementation (is SkePU load-balanced?);

- investigate the Scan skeleton;
- implement low-level variants (e.g., Pthreads) of the algorithms in the lab and compare performance and source code size.

Appendix A: Separable filters

For assignment 2, you will implement a separable filter. To understand why a 2D filter can be implemented with 2 1D filters, we have to view the filtering operation as a *convolution*. A 2D convolution can be expressed as a series of matrix operations, one for each element in the data. The filter weights are contained in one matrix, W. The weight matrix is multiplied element-wise with the matrix containing the overlapping region (O) and the sum of all such products is the output value. In certain cases (separable filters), the matrix W can be decomposed into vectors v, w such that $W = vw^T$. In the assignment filters (average and gaussian), we have the additional property that $v = w$. The computation can then be split into two steps, first applying w to O, generating a vector o. The last step is then simply the dot product $o^T v$.

For averaging filter with radius 1, $W = \begin{pmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{pmatrix}$ and $w = \begin{pmatrix} 1/3 \\ 1/3 \\ 1/3 \end{pmatrix}$.

You can check that $ww^T = W$, and that the same is true for the Gaussian filter (mathematically it is, but the lab skeleton approximates the gaussian weights which introduces a small error).