Assignment 4: Experimenting with OpenMP and the Halide compiler

Roel Jordans

2020

Organization

In Assignment 3 we tried to optimize the application by hand and later by using the compiler. These optimizations are however limited to a single core. The current trend, now that we seem to have hit a frequency limit, is to add more and more cores/threads to processors. Even mobile phones today come with quad or octa core processor. One obvious way to speedup modern applications is to parallelize over the available cores using threading. There are several ways of doing this. In the first part of this last assignment we are going to use OpenMP[1] for parallelizing the code over multiple cores. In the second (and last) part we will use Halide for optimizing this same algorithm again.

1 Compiling the program with OpenMP support

We start assignment 4 with the version you got after doing assignment 3. If you do not have this, or it is not working, start with a clean checkout[2]. To compile the application with OpenMP support there are several steps we need to take. First we need to include the `OpenMP` header file:

```
#include <omp.h>
```

Next we are going to modify the `Makefile` to link against and use `OpenMP`.

```
CFLAGS:=-I. -O3 -g3 -Wall -Wextra -std=c99 -D_POSIX_C_SOURCE=200000L -fopenmp
LIBS:=-lpng -lm -lrt -lomp
```

Here `-lomp` tells the linker to link against `libomp.so` and `-fopenmp` tells clang to use OpenMP. Now we can compile and run the application[3].

```
make && ./filter in.png out.png
```

This however results in the following error:

```
./filter: error while loading shared libraries: libomp.so: cannot open shared object
    file: No such file or directory
```

[1] `http://openmp.org/wp/`
[2] See assignment 3 how to obtain a clean checkout
[3] && tells the shell to execute filter if make was successful

It seems that the `OpenMP` library cannot be found at run-time. This library is part of LLVM; Because we ship the latest LLVM in */home/pcp20/opt/* which is not in the default library paths, so to make it known we need to set the following environment variable.

```
export LD_LIBRARY_PATH=/home/pcp20/opt/lib/
```

This tells the runtime linker to look in the right directory. Now we should be able to run the program.

Exercise 1

Annotate your program with the `#pragma omp parallel for` before each for loop to use OpenMP.[a]

[a] Take special care of for loops that update a single variable. See the OpenMP website for hints on how to solve race-conditions.

You will notice this won't give a big speed improvement. One possible reason is that a lot of variables are considered shared, and writing to these variables is protected. To improve this we need to make sure there are not to many unneeded shared variables. For example; variables like loop iterators should be private in the loop.

A good way to fix this is to manually set the shared variables.

Exercise 2

Lets see if we can improve performance by reducing the amount of shared variables. Start by setting the default to not allow any shared variable:

```
#pragma omp parallel for default(none)
```

If you compile the code, the compiler will warn you about the variables used inside the for loop. Then add all the variables you think need to be shared

```
#pragma omp parallel for default(none) shared(Kernel, sum)
```

Variables like loop iterators or hold intermediate values should be private to the loop. Constants that are only read, should be marked as constants. When the number of shared variables is minimized it should greatly improve the obtained speedup.

We can improve the performance by doing manual tiling. With tiling we devide the input data up into large blocks an have each thread process one block.

Exercise 3

By doing a manual tiling we can improve performance further, below is the start of tiling a loop:

```
#pragma omp parallel default(none) shared(output, input, ysize,hsize)
{
    int step = ( ysize ) / omp_get_num_threads ();
    int cpu  = omp_get_thread_num ();
    int stop = min ( ( cpu + 1 ) * step, wh );
    int start = cpu * step;
```

You can then use the calculated start/stop as the loop bounds. Why does this help?

Note

To check that multiple threads are created, you can 'trace' the program. The following command will list the amount of threads created:

```
strace -f -c ./filter in.png out.png 2>&1 | grep -c "Process .* attached"
```

The `strace` tool traces system calls and signals. Strace will print to stderr a *Process pid attached* for each thread or fork. The `grep` command counts these.

When done right the `CONTRAST` and `END FILTER` total execution time can be reduced to 1 second.

In total we should have obtained a total reduction of execution time of a factor 8 in assignment 3 and 4.

Exercise 4

Can you think of more ways (think of the method explained during the lectures) how you can improve the speed of the parallelization using OpenMP. The most gain is obtained by removing/reducing access to shared variables.

2 The Halide DSL

In this second part of the assignment we will be using the Halide[4] language and compiler in order to optimize our filter. We will try to investigate the advantages and limitations of using a DSL over a traditional C/C++ implementation. Halide is capable of both JIT (just-time) and AOT (ahead-of-time) compilation methods. In the rest of this assignment we will be investigating both[5].

To help you get started on this assignment we have prepared a few versions of the Halide code for the filtering algorithm. You can find those in the Git repository on the servers:

```
$ git clone /home/pcp20/material/repos/assignment4b.git
```

2.1 JIT compiling our code

The file `filt_jit.cpp` contains the Halide source code that implements the filter of assignment 3. (along with a few initializations in C++). You can compile the file using a single command to the compiler driver. However, since we're using external libraries we will need to point out the locations of our new header files and external libraries to the compiler. In the end the resulting command will look as follows[6]:

```
$ clang++ -std=c++11 -g filt_jit.cpp -o filt \
   -I/home/pcp20/opt/halide/tools -I/home/pcp20/opt/halide/include \
   -L/home/pcp20/opt/halide/bin -lHalide \
   -lpthread -ldl -lpng
```

In this process we link our filter executable to the Halide library, which lives in a non-standard library location. As a result Linux won't be able to find it when starting your program unless you provide extra

[4] `http://halide-lang.org/`

[5] You can find more details for all the sections and exercises of this assignment in the tutorial section of the Halide website `http://halide-lang.org/tutorials/tutorial_introduction.html`

[6] The \ character at the end of the line allows you to split a single command into multiple lines.

search locations. These can be provided through the LD_LIBRARY_PATH environment variable. You can run the compiled code using the following command[7]:

```
$ LD_LIBRARY_PATH=/home/pcp20/opt/halide/bin ./filt
```

Exercise 5

Compile and run the Halide source code as explained above. It uses the `in.png` image as input and produces an `out.png` image as output. If everything works correctly the output image should look identical to the output of assignment 3. However, be prepared to wait a while before it finishes[a]...

[a] Or kill the program when it takes too long.

2.2 Inspecting the schedule

Something must have gone wrong running the code like this. It seems like it needs ages to complete. Let's try inspecting the loop nest that Halide generates. You can use the `.print_loop_nest()` method to help you debug the structure of your loop nest, an example of this is provided at the end of the code in the TODO section. Uncomment it and we can try again, your source code should now look like:

```
FrameBuffer.print_loop_nest();//this statement will print the loop nest of our Func
```

This loop nest does not look like our previous C implementation at all... Perhaps Halide's default schedule is causing the problems. Remember that we can set the compute granularity of each Func in our pipeline as we wish. Let's try setting it to root. This means that we want to completely evaluate every stage in our pipeline before moving to the next one.

Exercise 6

Set the compute level of each `Func` in our code to root[a] (e.g. `edgeBuffer.compute_root()`, in the schedules section of the source code) and rerun the program. This time the structure of the loop nest should be similar to the C implementation. You should also be able to see how much time was needed to realize each function/stage in our pipeline as well as the output image.

[a] You can find the important `Funcs` here by looking at the produce/consume order in the loop nest structure that you printed previously.

2.3 Scheduling update definitions

We can define a Halide `Func` in multiple passes. The first one is called the pure definition and the subsequent ones are called update or recursive definitions. We use `Func::update(int)` to get a handle to an update step of a function for the purposes of scheduling. For example, the following line vectorizes the first update step of a function `f` across `x`.

```
      f.update(0).vectorize(x, 4);
```

To get a handle of the second update step we would just change 0 to 1 as f.update(1) etc.

[7] You might want to export this library path as part of the initialization scripts of your shell to avoid having to add it each time you want to test your filter

Exercise 7

We are now ready to optimize our code. Try different optimizations (reorder, split, tile, parallel, vectorize, compute_at etc) in order to improve the performance of our implementation. Remember that the algorithm and the optimization schedules are completely separated so changing the structure of your loop nest through different optimizations cannot affect the final output of your code.

2.4 Using the Autoscheduler

Halide also contains a builtin scheduler that attempts to automatically schedule a pipeline given an estimation of the problem size. The Autoscheduler groups together stages in order to maximize producer-consumer locality and parallelism. It can also output the schedule in a text format such that developers can use it as a starting point and further optimize their implementation when necessary.

Exercise 8

In this exercise we will use the Autoscheduler to generate a schedule for our implementation. Compile and run the `filt_jit_auto.cpp` file using the method described above. You should now see the schedule that that was automatically generated for each stage along with each stage's execution time. You can use that schedule to further optimize your code.

2.5 AOT compilation

Normally, we would write our whole program as a Halide implementation and use the JIT compilation method. However, in large scale projects it might be possible that parts of our pipeline cannot be described in Halide at all[8]. AOT compilation allows us to port only parts of our project to Halide (usually the ones that we want to optimize), while the rest are left unchanged.

The first step in our process after writing the Halide description of the code we want to optimize (`filt_ao.cpp`), is to compile it using the following command:

```
$ clang++ -std=c++11 -g filt_ao.cpp -o filt_hal \
   -I /home/pcp20/opt/halide/include \
   -L /home/pcp20/opt/halide/bin \
   -lHalide -lpthread -ldl
```

We can then run the Halide program in order to generate the header and library file that we want to include in our C++ project [9]. To run it again include the Halide runtime library location:

```
$ LD_LIBRARY_PATH=/home/pcp20/opt/halide/bin ./filt_hal
```

We are now ready to integrate this Halide implementation of our filter whenever we want by including the header file in our C++ code (`filt_c_ao.cpp`) and linking with our static library with the following command:

```
$ clang++ -std=c++11 -g filt_c_ao.cpp filt_ao.a -o filt \
    -I/home/pcp20/opt/halide/tools -I/home/pcp20/opt/halide/include \
    -lpthread -ldl -lpng
```

[8] Remember from our Halide lecture that the language is not Turing complete.

[9] A `filt_ao.h` header and a `filt_ao.a` static library file should be created after running `filt_hal`

You can notice that during this step we did not need to link with the Halide library (-lHalide) when compiling the final executable like we previously did, nor do we need to include the Halide Runtime to run it. We just need to link with the static library we previously generated (`filt_ao.a`).

Exercise 9

Follow the above procedure to compile and run your code using the AOT compilation method[a].

[a] You should copy the optimization schedule you used during exercise 7 or 8 and paste it in the source code of `filt_ao.cpp`

2.6 Using the GPU

Halide can also generate code for GPUs using either OpenCL or CUDA. We make the decision about whether to use the GPU for each `Func` independently. If you have one `Func` computed on the CPU, and the next computed on the GPU, Halide will do the copy-to-gpu under the hood.

For example, to compute a function `f` using the GPU in 16-wide one-dimensional thread blocks, first we split the index into blocks of size 16 and then we tell CUDA that our `Vars` 'block' and 'thread' correspond to CUDA's notions of blocks and threads, or OpenCL's notions of thread groups and threads.

```
Var block, thread;
f.split(i, block, thread, 16);
f.gpu_blocks(block).gpu_threads(thread);
```

This is a very common scheduling pattern on the GPU, so there's a shorthand for it:
`f.gpu_tile(i, 16);`
Similarly, to compute a function in 2D 8x8 tiles using the GPU: `f.gpu_tile(x, y, 8, 8);`

Note

To run the code for this exercise on a machine with a GPU you can use `co14.ics.ele.tue.nl` or `ti.ics.ele.tue.nl`.

Exercise 10

In this exercise we will try to schedule the `Funcs` that correspond to the local color correction stages of the pipeline (`I`, `Mask`, `Mask2`, `Maskf`, `output`) to run on the GPU[a]. The rest can either be optimized for the CPU or set to `compute_root`.

[a] You should use the `filt_jit_GPU.cpp` file which has CUDA enabled as an extra feature of the target.