Homework Submission

Your writeup should follow the writeup guidelines. Your writeup should include your answers to the following questions:

1. **Baseline**

   Get the source code and run the `baseline` project as shown in {doc}`walk_through`.

   a. Determine the throughput of `baseline` in pictures per second. This is your baseline. We use `-O2` for the baseline, so you should keep using -O2 for the rest of the homework. Ignore overhead such as loading and storing pictures for this and the following questions. (1 line)

   ```
   Total latency of Scale is: 1.66015e+07 ns.
   Total latency of Filter is: 2.75428e+08 ns.
   Total latency of Differentiate is: 3.29935e+07 ns.
   Total latency of Compress is: 1.50079e+08 ns.
   Total time taken by the loop is: 4.75124e+08 ns.
   ------------------------------------------------------------
   Average latency of Scale per loop iteration is: 166015 ns.
   Average latency of Filter per loop iteration is: 2.75428e+06 ns.
   Average latency of Differentiate per loop iteration is: 329935 ns.
   Average latency of Compress per loop iteration is: 1.50079e+06 ns.
   Average latency of each loop iteration is: 4.75124e+06 ns.
   Application completed successfully.
   ```

   Throughput of baseline in pictures per second = 1 / Max average latency per loop iteration.

   `Filter` takes the largest time.

   1 / 2.75428e+06 pictures per second.

2. **Coarse-grain parallelism**

   We will parallelize the application by processing half of each picture on core 0 and the other half on core 1, a form of coarse-grain, data-level parallelism. The initial implementation can be found in `hw3/assignment/coarse_grain`. We have parallelized `Scale` already for you.

   a. Can we parallelize all streaming functions in our application, i.e. `Filter_horizontal`, `Filter_vertical`, `Differentiate`, and `Compress` in the same way as `Scale`? Motivate your answer. Assume that we synchronize our cores between each producer-consumer pair. (3 lines)
      - `Filter_horizontal` can be parallelized using loop limits, the same as `Scale`, since each loop computation is independent of previously computed values.
      - `Filter_vertical` can also be parallelized for the same reasons as above.
      - `Differentiate` can also be parallelized for the same reasons as above.
      - `Compress` can also be parallelized for the same reasons as above.

   b. What speedup do you expect from parallelizing the functions that you considered parallelizable in the previous question? [Include an equation for the expected parallel runtime and show the

equation you use for computing the speedup as well as your final, numeric result. Report both per function speedup and overall application speedup.] (5--7 lines)

- `Filter_horizontal` and `Filter_vertical` can be sped-up by 2x since `Scale_coarse` has 2 threads.
- `Differentiate` can be sped-up by 2x since `Scale_coarse` has 2 threads.
- `Compress` can be sped-up by 2x since `Scale_coarse` has 2 threads.

TODO: Use Ahmdal's law to compute speedup

c. Complete the implementation by parallelizing the functions that you considered parallelizable in the previous question. Provide the relevant sections of code in your report.

- `Filter_horizontal`

```
void Filter_horizontal_coarse(const unsigned char *Input, unsigned
char *Output, int Y_Start_Idx, int Y_End_Idx)
{
for (int Y = Y_Start_Idx; Y < Y_End_Idx; Y += 2)
    for (int X = 0; X < OUTPUT_WIDTH; X++)
    {
    unsigned int Sum = 0;
    for (int i = 0; i < FILTER_LENGTH; i++)
        Sum += Coefficients[i] * Input[Y * INPUT_WIDTH + X + i];
    Output[Y * OUTPUT_WIDTH + X] = Sum >> 8;
    }
}
```

- `Filter_vertical`

```
void Filter_vertical_coarse(const unsigned char *Input, unsigned
char *Output, int Y_Start_Idx, int Y_End_Idx)
{
for (int Y = Y_Start_Idx; Y < Y_End_Idx; Y += 2)
    for (int X = 0; X < OUTPUT_WIDTH; X++)
    {
    unsigned int Sum = 0;
    for (int i = 0; i < FILTER_LENGTH; i++)
        Sum += Coefficients[i] * Input[(Y + i) * OUTPUT_WIDTH +
X];
    Output[Y * OUTPUT_WIDTH + X] = Sum >> 8;
    }
}
```

- `Differentiate`

```
void Differentiate(const unsigned char *Input, unsigned char
*Output, int Y_Start_Idx, int Y_End_Idx)
{
```

```
for (int Y = Y_Start_Idx; Y < Y_End_Idx; Y += 2)
    for (int X = 0; X < WIDTH; X++)
    {
    int Average = 0;
    if (Y > 0 && X > 0)
        Average = (Input[WIDTH * (Y - 1) + X] + Input[WIDTH * Y +
X - 1]) / 2;
    else if (Y > 0)
        Average = Input[WIDTH * (Y - 1) + X];
    else if (X > 0)
        Average = Input[WIDTH * Y + X - 1];

    unsigned char Diff = Input[WIDTH * Y + X] - Average;

    Output[Y * WIDTH + X] = Diff;
    }
}
```

- Compress

```
int Compress(const unsigned char *Input, unsigned char *Output,
int Size_Start_Idx, int Size_End_Idx)
{
for (int i = Size_Start_Idx; i < Size_End_Idx; i++)
{
    unsigned long long Code = Codes[Input[i]];
    int Code_length = Code_lengths[Input[i]];

    for (int j = 0; j < Code_length; j++)
    {
    Byte = (Byte << 1) | ((Code >> (Code_length - 1 - j)) & 1);

    if (++Length % 8 == 0)
    {
        Output[Length / 8 - 1] = Byte;
        Byte = 0;
    }
    }
}

if (Length % 8 > 0)
    Output[Length / 8] = Byte;

return Length / 8 + 1;
}
```

TODO: Finish parallel implementations or modify answer above

d. Measure the throughput of your parallel implementation.

e. Validate your results. Make sure that your parallel version produces the same answers as the original serial version. Explain how you validated your results; report any discrepancies in your final implementation. (3--5 lines)
f. Compare your measurement with your ideal, expected speedup. (1 line)
g. If your speedup is different from ideal, expected, what effects are likely to be responsible for the difference? (1-3 lines)

3. **Pipelining**

As an alternative to coarse-grain, data-level parallelism, we will investigate a pipelined implementation in this question. The initial implementation can be found in `hw3/assignment/pipeline_2_cores`. The provided stream has only 100 frames, but assume in your performance computations that you are dealing with a stream of infinite length.

a. Report the throughput of the initial pipelined implementation on 2 cores in pictures per second. (1 lines)

```
Total time taken by the loop is: 3.4019e+08 ns.
Application completed successfully.
```

Core 1 works on Scale and Filter. Core 0 works on Filter, Differentiate and Compress.

To compute the throughput of the initial pipelined implementation, we need to analyze the latencies of each of the stages in the pipeline. Since the pipeline is split amongst 2 cores, either of these cores could be the bottleneck in the system. Hence, we should determine:

- Avg. Latency of Scale
- Avg. Latency of Core1 Filter
- Avg. Latency of Core0 Filter
- Avg. Latency of Core0 Differentiate
- Avg. Latency of Core0 Compress

These latencies can be measured by using the `stopwatch` class. Once determined, we know that the throughput of the pipeline is `1 / MAX Latency` of the above stages.

TODO: Implement `stopwatch`

b. What is the best performance that one could theoretically achieve with a pipelined mapping of the streaming application on 2 cores over the single ARM core solution? (1 line)

```
Where is the bottleneck? How does pipelining help in
hiding the bottleneck?
```

Given the information about average latencies of each stage from the previous part, we know that even the best possible mapping of the application over 2 cores will have a throughput of at-

most the maximum of the minimum latency on each core while still scheduling all the stages in the pipeline on either core.

TODO: Implement the mapping

c. Describe the mapping that achieves the best performance. (3 lines)

d. Reviewing the provided code, explain how it is able to deal with filling and draining the pipeline of operators? That is, when the application starts, there is only data for the first stage in the pipeline (`Scale`) and no data for the later stages. After the input data has been consumed by the `Scale` stage, the later stages will still have data to process. How does the code assure the program runs correctly to completion on all data? (4--6 lines)

   The program deals with filling and draining the pipeline of operators using the variable `Frame` which lets the consumer know when the producer finishes its computation.

   `Frame > 0` indicates that the previous stages in the pipeline have finished processing, and the consumer can read from the buffer.

e. Review the provided code. Explain how you can adjust the `PIPELINE_PAR` parameter (in `Filter.cpp`) to maximize throughput. (2--3 lines)

   This parameter determines how much of the image is executed in each cycle of the thread. The higher the value of the parameter, the longer it takes to execute the filter function but the more data which gets pushed into the pipeline for future processing.

   We can adjust this parameter to maximize throughput based on the average latencies of each stage in the pipeline to minimize wasted cycles and maximize throughput.

f. Adapt the implementation by changing the parameter `PIPELINE_PAR` to optimize the pipeline task or implement your own mapping to optimize the pipeline tasks. Include the sections of the code that you modified in your report.

TODO: Tune `PIPELINE_PAR` and validate results

g. Validate your results. Report on how you validated and any discrepancies. (1--3 lines)

h. Report the throughput of your new application in pictures per second. (1--2 lines)

i. Let's investigate the performance if we incorporate the optimized pipeline in a video broadcast server. The input data is read from an interface with 300 MB/s throughput. $75%$ of traffic is video traffic that is compressed using our pipeline (running on 2 processors). Assume the 2 cores can pipeline the process perfectly. The remaining $25%$ is other traffic that we protect with an error correction code (ECC) running on a dedicated hardware unit that adds $10%$ overhead in size. The hardware ECC unit processes 150 MB/s. The output of the ECC unit and compression pipeline are output to a single 2-Gigabit/s Ethernet port.

   i. Draw a streaming dataflow diagram for the network server. Indicate throughput and data transfer ratios where applicable.
   ii. What is the maximum throughput that the server can achieve? (10 lines)
   iii. Where is the bottleneck? (1 line)

iv. How much smaller do we have to make the kernel (`FILTER_LENGTH`) of `Filter` to move the bottleneck? (7 lines)

4. **More Parallelism**

   Building on techniques and observations from previous parts, create a revised implementation that uses four 64b ARM cores to achieve additional speedup. The initial implementation can be found in `hw3/assignment/pipeline_4_cores`, where we currently utilize 3 cores. The provided stream has only 100 frames, but assume in your performance computations that you are dealing with a stream of infinite length.

   a. Report the throughput of the initial pipelined implementation on 3 cores in pictures per second. (1 lines)
   b. What is the best performance that one could theoretically achieve with a pipelined mapping of the streaming application on 4 cores over the single ARM core solution? (1 line)

   ```
   Where is the bottleneck? Can you hide the bottleneck
   anymore like you did in 3b? Does the bottleneck limit the best
   performance you can achieve?
   ```

   c. Describe the mapping that achieves the best performance. Try to achieve the best speedup over the single ARM core solution.
   d. Implement your design and include your code in your report.
   e. Report speedup obtained and relate it to your solution. (3--5 lines)
   f. Validate your design and report on any discrepancies.

Deliverables

In summary, upload the following in their respective links in canvas:

- a tarball containing the 4 projects with your modified code.

  ```
  :class: dropdown, tip
  ```
  # Compress
  tar -cvzf <file_name.tgz> directory_to_compress/
  # Decompress
  tar -xvzf <file_name.tgz>
  ```

- writeup in pdf.
  ```