University of Colorado - Boulder

CSPB 2400
Computer Systems | Summer 2024

Lab 4: Perf Lab

Sam Walker

Hoang Truong

Monday, July 22, 2024

College of Engineering & Applied Science
UNIVERSITY OF COLORADO **BOULDER**

Part 1

Execution Check (40 points)

GitHub Repo Link: Repo Link [https://github.com/cu-cspb-2400-summer-2024/lab4-perflab-sawa9885]

Screenshot of make test without openmp:

```
jovyan@jupyter-sawa9885:~/CSPB 2400/lab4-perflab-sawa9885$ make test
g++ -g -fopenmp -O3 -fno-omit-frame-pointer -Wall -o filter FilterMain.cpp Filter.cpp cs1300bmp.cc
./Judge -p ./filter -i boats.bmp
gauss: 210.929999..207.564738..203.161732..202.356099..198.132808..201.851632..
avg: 211.959075..239.727190..208.152873..193.906565..186.676422..193.361589..
hline: 211.071402..235.712047..207.838570..227.730749..200.257704..186.171765..
emboss: 210.081321..209.574620..205.684713..189.350345..199.083478..196.243697..
Scores are 186 186 189 193 193 196 198 199 200 201 202 203 205 207 207 208 209 210 210 211 211 227 235 239
median CPE for  is 205
Resulting score for  is 78
./Judge -p ./filter -i blocks-small.bmp
gauss: 141.340347..139.716795..139.827971..136.429491..136.720976..136.921818..
avg: 177.288389..137.540850..134.522972..139.789946..136.548496..135.252548..
hline: 150.683376..149.013914..144.857450..144.314526..146.055187..146.011631..
emboss: 147.880098..147.350704..150.576170..147.588736..148.537012..145.540230..
Scores are 134 135 136 136 136 136 137 139 139 139 141 144 144 145 146 146 147 147 147 148 149 150 150 177
median CPE for  is 144
Resulting score for  is 86
cmp filtered-avg-blocks-small.bmp tests/filtered-avg-blocks-small.bmp
cmp filtered-avg-boats.bmp tests/filtered-avg-boats.bmp
cmp filtered-emboss-blocks-small.bmp tests/filtered-emboss-blocks-small.bmp
cmp filtered-emboss-boats.bmp tests/filtered-emboss-boats.bmp
cmp filtered-gauss-blocks-small.bmp tests/filtered-gauss-blocks-small.bmp
cmp filtered-gauss-boats.bmp tests/filtered-gauss-boats.bmp
cmp filtered-hline-blocks-small.bmp tests/filtered-hline-blocks-small.bmp
All tests passed
```

Fig. 1 No OpenMP

Screenshot of make test with openmp:

```
jovyan@jupyter-sawa9885:~/CSPB 2400/lab4-perflab-sawa9885$ make test
g++ -g -fopenmp -O3 -fno-omit-frame-pointer -Wall -o filter FilterMain.cpp Filter.cpp cs1300bmp.cc
./Judge -p ./filter -i boats.bmp
gauss: 182.215908..107.157114..240.604906..197.393768..677.364655..3566.977512..
avg: 179.350496..162.793670..4544.299197..187.440865..180.773416..117.057583..
hline: 192.408761..327.164648..4176.098471..183.879496..3957.430681..277.162792..
emboss: 3701.628265..168.093208..197.033505..318.946468..369.188688..166.900772..
Scores are 107 117 162 166 168 179 180 182 183 187 192 197 197 240 277 318 327 369 677 3566 3701 3957 4176 4544
median CPE for  is 197
Resulting score for  is 79
./Judge -p ./filter -i blocks-small.bmp
gauss: 67.610821..51.301029..74.888025..55.484331..75.922619..55.665302..
avg: 53.847979..74.666080..56.329550..60.271442..53.807089..76.467255..
hline: 56.207470..52.917543..57.022261..62.456064..63.031422..59.344780..
emboss: 61.866423..61.371971..55.265730..65.369678..53.141762..57.226765..
Scores are 51 52 53 53 53 55 55 55 56 56 57 57 59 60 61 61 62 63 65 67 74 74 75 76
median CPE for  is 59
Resulting score for  is 102
cmp filtered-avg-blocks-small.bmp tests/filtered-avg-blocks-small.bmp
cmp filtered-avg-boats.bmp tests/filtered-avg-boats.bmp
cmp filtered-emboss-blocks-small.bmp tests/filtered-emboss-blocks-small.bmp
cmp filtered-emboss-boats.bmp tests/filtered-emboss-boats.bmp
cmp filtered-gauss-blocks-small.bmp tests/filtered-gauss-blocks-small.bmp
cmp filtered-gauss-boats.bmp tests/filtered-gauss-boats.bmp
cmp filtered-hline-blocks-small.bmp tests/filtered-hline-blocks-small.bmp
All tests passed
```

Fig. 2 OpenMP

Part 2-i - Loop Order

Loop Order Modification:

```
for (int plane = 0; plane < 3; plane++) {
  for (int row = 1; row < height - 1; row++) {
    for (int col = 1; col < width - 1; col++) {
      acc = 0;
      for (int j = 0; j < filterSize; j++) {
        for (int i = 0; i < filterSize; i+=2) {
```

Explanation

The modification shown above improves performance by maximizing cache locality and reducing cache misses. By iterating over the smallest dimension (color planes) in the outermost loop and the largest dimension (width) in the innermost loop, we ensure that the data accessed within the innermost loop is contiguous in memory, thus making better use of the CPU cache.

Part 2-ii - Strength Reduction

Strength Reduction Modifications:

```
// Precompute jOffsets
int jOffsets[filterSize];
for (int j = 0; j < filterSize; ++j) {
  jOffsets[j] = j * filterSize;
}
```

```
acc += input->color[plane][row + i - 1][col + j - 1] * filterValues[jOffsets[j] + i];
if(i+1<filterSize)
  acc += input->color[plane][row + i][col + j - 1] * filterValues[jOffsets[j] + i+1];
```

Explanation

Strength reduction is an optimization technique that replaces an expensive operation with a less costly one. In this case, we utilize two methods: precomputing the offsets and using addition rather than direct assignment within the innermost loop, thereby improving performance.

By precomputing the offsets in the array `jOffsets`, we eliminate the need for multiplication within the innermost loop.

The first image shows the precomputation of offsets, while the second image illustrates the use of rolling addition to further reduce computational overhead rather than directly assigning these values to the color 3d array. These optimizations collectively enhance the performance by minimizing the number of expensive operations within the innermost loops.

Part 2-iii - Code Motion

Code Motion Modification:

```
// Precompute jOffsets
int jOffsets[filterSize];
for (int j = 0; j < filterSize; ++j) {
  jOffsets[j] = j * filterSize;
}
```

Explanation

Code motion is an optimization technique that improves performance by moving invariant computations outside of loops to reduce redundant calculations. In the example above, we precompute the offsets for the filter indices and store them in the `jOffsets` array. By doing this, we eliminate the need for repetitive multiplications within the innermost loop, thereby reducing the computational overhead. Another simple example is that the original code had filter -> getsize in the for loop conditional checks and by storing those as a variable at the very beginning we don't have to call a get function every time the for loop triggers.

Part 2-iv - Loop Unrolling

Loop Unrolling Modification:

```
for (int j = 0; j < filterSize; j++) {
  for (int i = 0; i < filterSize; i+=2) {
    acc += input->color[plane][row + i - 1][col + j - 1] * filterValues[jOffsets[j] + i];
    if(i+1<filterSize)
      acc += input->color[plane][row + i][col + j - 1] * filterValues[jOffsets[j] + i+1];
  }
}
```

Explanation

Loop unrolling is an optimization technique that involves duplicating the loop body multiple times to decrease the loop overhead and increase the instruction-level parallelism. In this case, we experimented with unrolling the inner loop by different factors, including more than twice in the inner loop and multiple times in the outer `j` loop. However, we found that unrolling the inner loop once, as shown in the code snippet, provided the most effective performance improvement. By unrolling the inner loop once, we reduce the number of iterations and the overhead associated with loop control, while maintaining a balance between code size and execution efficiency. This approach effectively minimizes the computational overhead and enhances the performance of the loop without significantly increasing the code complexity or memory usage.

Part 2-v - Other Modifications

GCC Flag Modification:

```
CXXFLAGS= -g -fopenmp -O3 -fno-omit-frame-pointer -Wall
```

OpenMP Modification:

```
#pragma omp parallel for
```

Explanation

Compiler optimization levels control the extent and types of optimizations performed by the compiler. The optimization levels range from -O0 to -O3:

- -O0: No optimization; this level is the default and focuses on reducing compilation time and ensuring the most straightforward debugging.
- -O1: Enables basic optimizations that do not require significant compilation time. It improves performance without substantially increasing the compilation time.
- -O2: Enables further optimizations beyond -O1, including more aggressive code transformations and inlining. It is a good balance between compilation time and performance improvement.
- -O3: Enables all the optimizations from -O2 and includes more aggressive optimizations like loop unrolling, vectorization, and function inlining. It focuses on maximizing performance, even if it increases compilation time and code size.

We choose to use -O3 because it provides the highest level of optimization, focusing on maximizing the performance of the generated code, which is critical for computationally intensive tasks.

For parallel processing, we use OpenMP, a widely used API for multi-platform shared-memory parallel programming. The -fopenmp flag enables the OpenMP support in the compiler.

The pragma omp parallel for directive is used to parallelize the execution of loops. It distributes the iterations of the loop across multiple threads, allowing for concurrent execution and reducing overall runtime. We do not use the collapse() keyword because it is intended for nested loops where the iterations can be flattened into a single loop. However, in our case, the parallelization is applied at the outermost loop level, and the structure of the nested loops does not benefit from collapsing. This approach provides a clear and effective way to utilize multiple threads without introducing unnecessary complexity.