Optimization & Tuning of Harris corner detection algorithm for SIMD and Multicore Architecture

Adarsh Patil

May 4, 2015

Abstract

The objective of the assignment is to optimize the Harris corner detection algorithm for performance using locality, SIMD and multicore parallelism transformations.
Using suitable compiler flags, transforms and optimizations we obtain a speed up of **11.5X** over unparallelized reference implementation and **13.5X** over OpenCV using **GCC 4.9 compiler** and **11.3X** over unparallelized reference implementation and **14.6X** over OpenCV using **ICC 15.0 compiler**. All experiments were performaned on Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz [Haswell µarch, 4 core, 64 KB L1 private / 256 KB L2 private / 8 MB L3 shared cache].

1 Optimizations

1.1 Compiler flags

OpenMP - Autoparallelization construct
Profile Guided Optimizations - Rather than programmer-supplied frequency information, profile-guided optimisation use the results of profiling test runs of the instrumented program to optimize the final generated code. Since each run of the program has the same sequence of execution irrespective of the output size this optimization performs well.
Less precise floating point Math - allows reordering of instructions to something which is mathematically the same but not exactly the same in floating point, assumes all math is finite and enables reciprocal approximations for division.
Strict Aliasing - Allow the compiler to assume the strictest aliasing rules

GCC

- -O3
- -fprefetch-loop-arrays
- -fopenmp

- -ffast-math
- -fprofile-generate, -fprofile-use
- -fstrict-aliasing

ICC

- -O3
- -no-prec-div
- -ansi-alias
- -xhost
- -openmp
- -prof-gen, -prof-use

1.2 Code Optimization

- Tiling / Blocking - To improve locality of computation and increase reuse.
- Tile sized private arrays to store intermediate results - Intermediate results computed are only needed for the next stage of the pipeline and need not be persisted till the end of the computation. Each core / thread can compute these intermediate results in cache as private data.
- Unroll Jam i loop for Sxx, Syy and Sxy computation - Improves Locality and reuse for elements of array Ixx, Iyy and Ixy respectively used in the stencil computatation.

1.3 Pragma

- pragma omp parallel for
- pragma {GCC} ivdep

1.4 Others

A pointer that is marked *__restrict__* guarantees that the region of memory referenced by that pointer is the only way to access that region over the scope of that pointer.

2 Evaluation

2.1 Input Image

- Image size: 21600px X 10800px
- Size on disk: 29 MB

2.2 Processor

- Intel Core i7-4770 CPU @ 3.40GHz
- 32KB L1 iCache, 32KB L2 dCache private
- 256KB L2 private
- 8MB L3 shared

2.3 Speedup

The performance numbers were taken with 5 repeated runs. The first run was discarded as warm up, and the average of the next four is used to compute speedup.
The performance was tuned for 5 tile size from 16, 32, 64, 128 and 256. The input image has 2 tileable dimensions. We explore uniform tile sizes along both dimensions, i.e. 16x16, 32x32, 64x64 etc. for simplicity of comparison. Based on this tuning, the tile size of 128x128 was found to be ideal for the execution.

Table 1: **Single Core Execution Time (in ms) & Speedup -Vectorization**

	OpenCV	Reference	Optimized	Speedup by locality transforms
No Vectorize	3515.29	3767.32	2442.4	1.54
Vectorize	3566.35	3035.41	930.90	3.26
Vectorization Speedup	-	1.24x	2.62x	

Table 2: **Execution Time (in ms) & Speedup using GCC 4.9**

	OpenCV	Reference	Optimized	Speedup w.r.t reference
1 core	3566.35	3035.41	930.90	3.26x
2 core	-	1990.6	422.53	4.71x
4 core	-	1940.92	264.73	7.34x
Speedup by parallelism	-	1.56x	3.52x	

Table 3: **Execution Time (in ms) & Speedup using ICC 15.0**

	OpenCV	Reference	Optimized	Speedup w.r.t reference
1 core	3567.95	2755.83	904.61	3.04x
2 core	-	1617.88	355.724	4.54x
4 core	-	1444.89	243.19	5.94x
Speedup by parallelism	-	1.90x	3.72x	

From Table 1 we observe that the locality optimization for vectorization i.e. using private scratch pads for intermediate computation and aligning vector additions help to improve performance by about 4.05x (=3767/930) times on a single core. This is due to the SSE2 (256bit instruction) vectorization.

From Table 2 and 3 we can see than ICC performs about 8.6% better than GCC on optimized code 10.2% better than GCC on reference (non-optimized) code.

3 Video Performance

Optimizations for harris corner detection was implemented for frame processing in video files by creating a shared object (.so) with the same transformations applied to the algorithm. GCC 4.9.2 was used for the compilation.
Python OpenCV program was used as a driver to switch between the various implementations.

3.1 Evaluation

A Full HD (1920x1080) resolution video file at 24fps (Big Buck Bunny) was used for evaluation on the same hardware as above.
The first 150 frames were considered for performance evaluation and speedup. First 50 run with vanilla OpenCV, frames 50-100 run reference algorithm and frames 100-150 run optimized algorithm in Harris Corner detection mode. The average execution time over 50 frames is reported below.

Table 4: Average Execution Time (in ms) for 50 frames (GCC)

	OpenCV	Reference	Optimized	Speedup w.r.t reference
4 core	65	55.18	19.84	2.78x

4 References

1. GCC Optimization Options - https://gcc.gnu.org/onlinedocs/gcc-4.9.2/gcc/Optimize-Options.html
2. Ulrich Drepper, What every programmer should know about memory http://lwn.net/Articles/250967/http://lwn.net/Articles/250967/
3. ICC Optimization Options - https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler
4. Big Buck Bunny - http://bbb3d.renderfarming.net/download.html