ECE 415| HPC, HW1

Spyridon Stamoulis 03775, Vasileios Grapsopoulos 03791, Apostolos Nikolaidis 03790

University of Thessaly
`spstamoulis@uth.gr, vgrapsopoulos@uth.gr, apnikolaidis@uth.gr`

System Architecture and Environment Specifications

– **Virtualization Platform:** VMware (Full Virtualization)
– **Operating System:** Ubuntu Server with XFCE Desktop Environment
– **System Purpose:** The system was configured as a server-based environment to increase performance with a minimal desktop interface for access and monitoring.

Virtualized CPU, Memory, and Cache Topology

Table 1: Virtualized CPU and Memory Topology

Component	**Metric**	**Value**
Architecture	Instruction Set	x86_64
Virtual Address Size	Maximum Virtual Address Space	48 bits
Logical CPUs	Total Online CPUs	2
NUMA Topology	Non-Uniform Memory Access Nodes	1

Table 2: Cache Hierarchy Summary

Cache Level	**Access Type**	**Size (Total)**
L1d	Data Cache	96 KiB
L2	Secondary Cache	2.5 MiB
L3	Last Level Cache	24 MiB

Note: The cache line size for all levels is 64 bytes.

System and Software Specifications

Table 3: System and Software Specifications

Category	**Component**	**Value**
Hardware	Processor Model	12th Gen Intel(R) Core(TM) i5-12450H
Hardware	Memory (RAM)	3.8 GiB (approx. 4 GB)
Software	Operating System	Ubuntu 24.04.3 LTS
Software	Kernel Version	6.8.0-85-generic
Software	Compiler Version	Intel(R) oneAPI DPC++/C++ Compiler 2025.2.1

Optimizations

0.1 NOTES

1. We run all tests 21 times with the script test.c and we removed the fastest and slowest times. With the 19 times left we calculated the average value and the standard deviation.
2. We run all tests the same day in the same virtual machine to get the numbers shown in the report. We did not suspend it or shut it down the process.

Manual Optimizations without Compiler Flags (-O0)

00_orig: Our original time was: 2.598832 seconds
Standard Deviation: +/- 0.095252 seconds

01_loop_interchange: In this step we changed the i,j loops putting the j loops inside. This helps with cache locality and significantly improves the times.

Our time was: 1.227064 seconds
Standard Deviation: +/- 0.003735 seconds

02_loop_unroll: Here we fully unrolled the loops in the convolution2D function as we observed that there were only 9 iterations.

Our time was: 0.946562 seconds
Standard Deviation: +/- 0.027696 seconds

03_loop_fusion: In this step we fused the main loop with the loop that calculates PSNR. Here we saw improvement in the time, but in the next steps after 06_pow_inlining we saw decrease in time using the loop fusion.
(The reduced performance is possibly due to data dependencies of the $PSNR+=t*t$ instruction, (which is simplified and you can find it after step 6), in order for the auto vectorization of the compiler to work) For this reason we left it here to be visible as a useful change in the first stages, but we left it out of all the rest.

NOTE: These times are for the machine with the intel processor. In the other machines loop fusion was always slower in every step.

Our time was: 0.824754 seconds
Standard Deviation: +/- 0.003737 seconds

04_func_inlining: Here we removed the loop fusion as we mentioned before and we performed function inlining of the convolution function.

Our time was: 0.913228 seconds
Standard Deviation: +/- 0.004742 seconds

05_operation_optimization: This optimization involves three smaller steps.

1. **Inlining Operator Matrices**
   First, instead of reading the horizontal and vertical Sobel operator values from memory on each iteration, we hardcoded these constant values directly into the convolution logic. This eliminates repeated memory lookups, ensuring the operator values are available in registers or as immediate values in the instruction stream, which is significantly faster.

   Our time: 0.890265 seconds
   Standard Deviation: +/- 0.001712 seconds

2. **Strength Reduction**
   Next, we replaced computationally expensive arithmetic operations with cheaper equivalents. This included:
   - Eliminating multiplications by 0 and 1, as they are redundant.
   - Replacing multiplications by 2 with a faster bitwise left shift (`« 1`).
   - Replacing expressions like x + (-1 * y) with x - y.

   These changes reduce the number of cycles required for the arithmetic computations within the main loop.

   Our time: 0.884379 seconds
   Standard Deviation: +/- 0.000851 seconds
3. **Common Subexpression Elimination and Address Calculation Optimization**
   - We reduced the number of additions and subtractions by rearranging operations.
   - We removed multiplications like i*SIZE by using helper variables (help_4_mult, output_idx, input_idx).

Our final time for this version: 0.876508 seconds
Standard Deviation: +/- 0.015177 seconds

06_pow_inline: We removed the pow function as we observed that the pow uses only power 2.

Our time was: 0.160689 seconds
Standard Deviation: +/- 0.002835 seconds

NOTE: Also at this point we tried to loop unroll the main loop but it was slower.

07_LUT (Look-Up Table): A major performance bottleneck was the repeated calls to the `sqrt` function inside the main loop. Our optimization in this step was to replace this computationally expensive operation with a memory-bound Look-Up Table (LUT). The development was a three-stage process.

1. **Initial Implementation and Worst-Case Sizing**
   First, we performed a worst-case analysis to determine the required size of the LUT. The variable `p` is the sum of the squared horizontal (G_x) and vertical (G_y) gradients. To create a robust LUT, it must be large enough to handle the maximum possible value of `p`.
   Given that input pixels are `unsigned char` (0-255), the maximum gradient magnitude occurs when pixel values of 255 align with the positive Sobel operator coefficients and 0 with the negative ones. The sum of positive coefficients is 4.
   The maximum possible value for G_x is therefore: $max(G_x) = (255 \times 4) - (0 \times 4) = 1020$.
   The maximum value for `p` occurs when both gradients are at their peak: $max(p) = max(G_x)^2 + max(G_y)^2 = 1020^2 + 1020^2 = 2,080,800$.
   To handle every possible input, our initial LUT was sized with 2,080,801 entries. This approach successfully replaced approximately 16 million sqrt calls with simple array lookups.

   Our time: 0.159822 seconds
   Standard Deviation: +/- 0.001121 seconds
2. **Optimizing the LUT Population**
   Next, we optimized the one-time setup cost of the LUT itself. Instead of populating it with `sqrt` calls, we developed a more efficient integer-based algorithm that calculates squares. This method relies on a conditional branch that is only taken when the loop index is a perfect square. A modern branch predictor learns this highly predictable pattern, mispredicting only on the rare occasions when the branch is taken. For the full-sized LUT, this results in only approximately 1442 mispredictions ($\sqrt{2,080,800} \approx 1442$) out of over 2 million iterations, making the setup loop extremely efficient.

   Our time: 0.159034 seconds
   Standard Deviation: +/- 0.000398 seconds

3. **Domain-Specific LUT Reduction**
   Our final and most crucial refinement came from analyzing the problem's domain constraints. The final output pixel values are `unsigned char` and are therefore clamped to 255. This implies that any result from `sqrt(p)` that is greater than 255 is irrelevant. Consequently, we only need to store pre-calculated values for `p` up to $255^2 = 65,025$. By reducing the LUT size from over 2 million entries to just 65,026, we drastically reduced its memory footprint and setup time, yielding a significant performance gain.

Our final time for this version: 0.155526 seconds
Standard Deviation: +/- 0.001545 seconds

08_unroll_LUT_PSNR (Loop Unrolling):

1. **Unrolling the LUT Population Loop**
   First, we unrolled the loop responsible for populating the Look-Up Table. This way we reduce the number of iterations and thus the overhead associated with loop control (counter increments, conditional checks, and branches). This creates a larger, straight-line block of code for the CPU to process. We experimented with unroll factors of 2, 4, 8, 16, 32, and 64, finding that a factor of 16 provided the optimal balance between reduced overhead and code size.

   Our time: 0.136190 seconds
   Standard Deviation: +/- 0.006241 seconds
2. **Unrolling the PSNR Calculation Loop**
   Next, we applied the same technique to the loop that calculates the Peak Signal-to-Noise Ratio (PSNR). We again tested a wide range of unroll factors (2, 4, 8, 16, 32, 64, 128, and 256) and determined that, as with the LUT loop, a factor of 16 yielded the best performance. Beyond this point, we observed diminishing returns, likely due to increased register pressure or instruction cache limitations.

Our final time for this version: 0.101316 seconds
Standard Deviation: +/- 0.000899 seconds

08.5_loop_tiling:

We considered implementing loop tiling, hypothesizing that it would improve cache performance. The Sobel filter's working set requires holding three rows of the 4096-pixel image in memory. Assuming 8-byte double pixels, this working set is 96 KiB (3 * 4096 * 8 bytes), which fits exactly within the 96 KiB L1 data cache. This suggests that while L1 cache capacity is not exceeded, performance might still be sensitive to access patterns.

Contrary to expectations, the tiled version was slower. The most likely explanation is the effectiveness of the CPU's hardware prefetcher. The linear, predictable memory access of the original row-by-row processing allows the prefetcher to hide memory latency by fetching data before it is needed. The more complex access pattern of loop tiling likely disrupted this mechanism. Consequently, the computational overhead of the tiling logic outweighed any potential cache benefits, resulting in a net performance loss. For this reason, the optimization was not included in the final version.

09_add_pointers: We added a pointer to the input matrix to use them instead of the indexes we used to help with extra additions of +j and +1, -1, +2. We also initialized the pointer in the outer loop (i loop) to save many millions of initializations. Finally, we did all operations involving the variables (help_p_vect, help_p_horiz) in one line for each.

Our time: 0.077481 seconds
Standard Deviation: +/- 0.001342 seconds

010_unroll_main_loop: At this point unrolling the main loop was faster so we tried by factors of 2, 4, 8, 16, 32 and the best was by 4.

Our time: 0.069670 seconds

Standard Deviation: +/- 0.001136 seconds

NOTE: We also added restrict in the output and golden pointers (we did not see any important difference in time, but we kept it just in case).

Analysis with Compiler Optimizations (-fast)

After finishing -O0, we run the 00_orig with the ffast-math flag to see the starting time.
Our time was: 0.585806 seconds
Standard Deviation: +/- 0.0004295 seconds
Then we ran the final version (v10) with the ffast-math flag, and it was also improved.
Our time was: 0.030982 seconds
Standard Deviation: +/- 0.000366 seconds
Finally, we adjusted the v10, to match the compiler's optimizations

Remove_LUT_unroll_loops Although we previously used a Look Up Table (LUT) in -O0 and it increased performance, with the -fast flag, we reverted to using the sqrt() function. As the -Qopt-report=max -qopt-report-file=stdout, report pointed out, the LUT forced the CPU into slow, scattered memory lookups, while sqrt() could be converted into a single, fast SIMD instruction.

Float_precision Then we used single precision sqrtf() instead of double precision, to decrease the computation. The double to float precision change, lets a single SIMD instruction, process twice as many pixels, effectively doubling the potential performance. Although PSNR dropped from inf to 57.3897, it is an excellent tradeoff. (PSNR > 40)
Also we used fminf which is faster, because it is a branchless operation that replaces a standard if-else structure. This avoids costly branch misprediction penalties, allowing the CPU's instruction pipeline to execute without stalling.

Anti_loop_unrolling Manual loop unrolling was removed because it interferes with the compiler's far more powerful automatic optimizations. The complex, hand-written code prevents the auto-vectorizer from recognizing the simple, repetitive pattern,it needs to apply highly efficient SIMD instructions.

Add_const_and_restrict Adding const promises the compiler the input data is read-only, while restrict guarantees the pointer is the only one pointing to that memory location. This eliminates pointer aliasing concerns, giving the compiler important information to perform aggressive optimizations like auto-vectorization (SIMD). This improved time, unlike -O0 which this change did not have a significant impact in v10.
We also declared the loop counters i and j with the register keyword, as well as other variables, but runtime did not decrease, and found out that the compiler optimizes it itself. Our time was: 0.026902 seconds
Standard Deviation: +/- 0.000888 seconds

Graphs

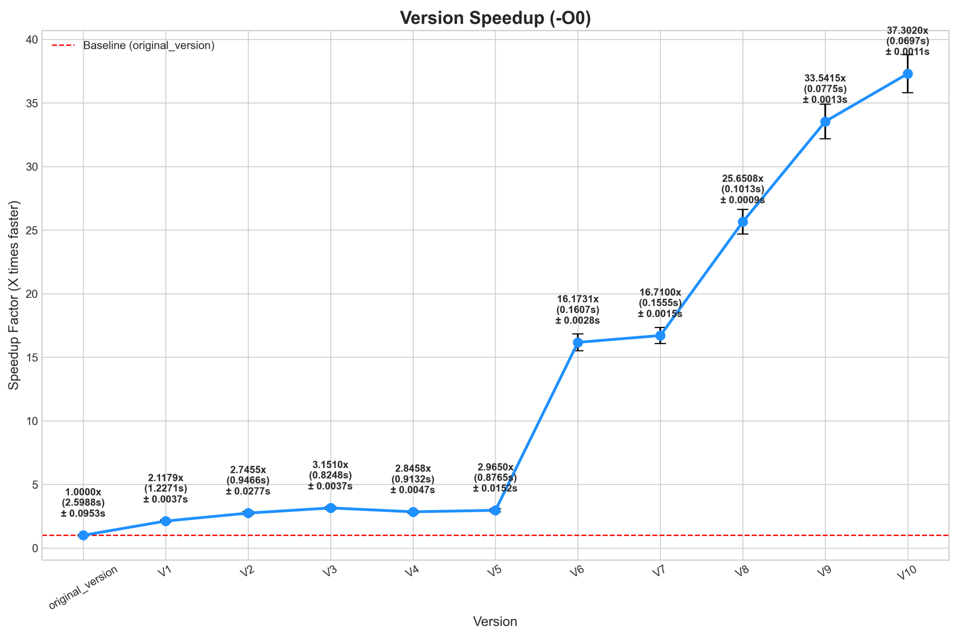

Fig. 1: Performance speedup of optimized versions (-O0) relative to the original version. The vertical axis shows the speedup factor (X times faster). The standard deviation is translated to speedup deviation for the graph

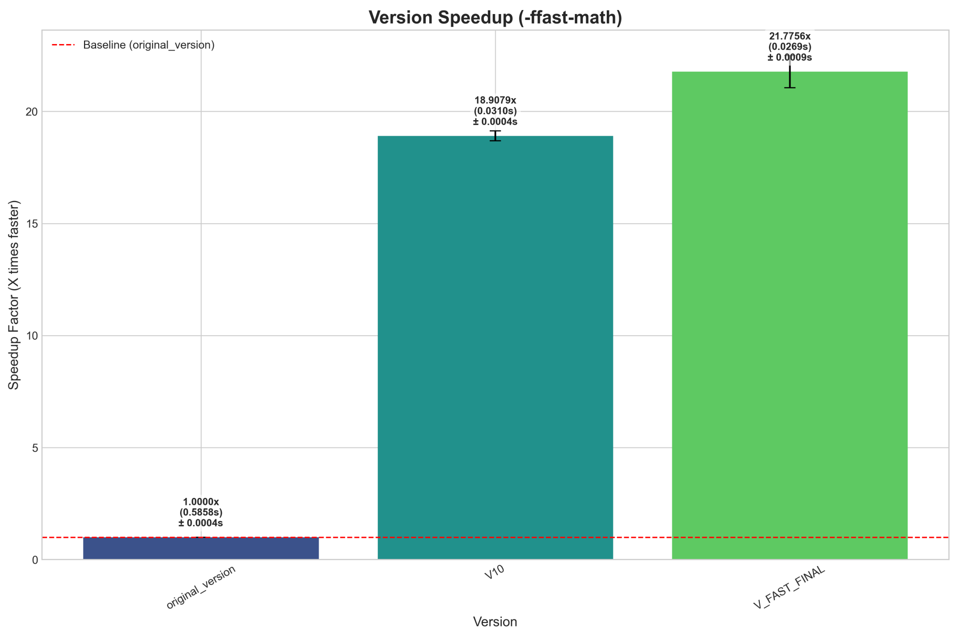

Fig. 2: Performance speedup of optimized versions (-ffast-math) relative to the original version. The vertical axis shows the speedup factor (X times faster).