Hands-on Session 1
MU5IN160

Mengqian XU, Edoardo CARRÀ

September 22, 2024

1 Introduction

This report outlines the process of optimizing the image blur algorithm through multiple versions. Each optimization targets specific computational redundancy, improving algorithm performance by reducing computation and memory access. Below are a short description of the two target architectures: *Denver 2* and *Cortex-A57*, and each version of the code, a bried analysis.

2 Target Architectures

Jetson TX2 is a NVidia-designed and-produced board for embedded AI computing. It is built around a 256-core NVIDIA Pascal GPU architecture with 256 NVIDIA CUDA cores, a Dual-Core NVIDIA Denver 2 64-Bit CPU and Quad-Core ARM Cortex-A57 MPCore.

2.1 Denver 2 architecture

Denver is a CPU microarchitecture from Nvidia [Den], capable of executing ARMv8 instruction thanks to an hardware translator. It also implements dynamic code optimization directly on the hardware. Its main features are:

1. It is 7-wide in-order superscalar.
2. It can fetch up to 2 ARMv8 instructions per cycle.
3. It has dynamic branch prediction with Branch Target Buffer and Global History Buffer.
4. The pipeline of Denver 1 has 15 stages, the mispredict penalty is 13 cycles. (We did not find how many stages there are for Denver 2, but we expect something very similar.)

Notice that the hardware translator performs very aggressive optimizations of the code. It performs loop unrolling, register renaming, memory access reordering and many other optimization. So, even if no compiler optimizations are set, we have to take this feature into account in our analysis.

Given the latter feature, we expect Denver 2 it will outperform the Cortex-A57, when no compiler optimization will be applied. Moreover, we expect some unexpected behavior due to the hidden optimization performed by the hardware.

2.2 Cortex-A57

The ARM Cortex-A57 is a CPU designed by ARM Holdings [Cor]. Implements the ARMv8-A 64-bit instruction set. The main features are:

1. Out of order, speculative issue 3-way superscalar execution pipeline.
2. 2-level dynamic predictor with Branch Target Buffer (BTB) for fast target generation
3. Branch misprediction penalty = 16-19 cycles

3 Optimizations

We follow an *iterative* optimization process starting with avoiding the management of the borders. So, we found two possible ways to achieve that: the first which is cache friendly and one which is not, due to non-coalesced memory accesses.

3.1 Blur Kernel WITHOUT Border (default_nb)

When handling image boundaries, it is impossible to compute the average of 9 neighboring pixels as some are missing. Therefore, in the *'default_nb'* version, we start by processing the middle section of the image from *x+1* to *y+width-1* and handle the borders separately. Instead of performing boundary checks and complex calculations for edge pixels, we simply copy the values of neighboring pixels, which simplifies the process and boosts performance.

This optimization removes the need for 'if' conditions that check if a pixel is on the edge of the image, reducing the number of branches in the code. For processors like Cortex A57, which have limited branch prediction depth only two levels), avoiding these branches improves efficiency. Branch mispredictions can cost 16-19 cycles, which adds significant overhead when processing large images. By simplifying border handling, we reduce the likelihood of such costly mispredictions, leading to a performance gain. Conversely, we observed a significant performance drop when running \textit{default_nb} on Denver 2, likely due to the hardware accelerator unit, which seemed to perform more efficiently with the previous version of the code.[1]

3.2 Loop Unrolling on X-axis (optim1)

In the 'optim1 version, loop unrolling was applied on the X-axis. This optimization increases the number of instructions, but reduces the number of control flow jumps and condition checks within the loop, simplifying the overall control overhead. By fully unrolling the loop, there is no need for an epilogue, making the code more efficient in theory.

However, despite reducing control overhead, 'optim1' actually performed slower than *'default_nb'* on Denver 2 in the non-coalesced version . This is because while loop unrolling reduces jumps, it increases code size, which can negatively affect performance depending on the processor's architecture and cache behavior. In this case, the larger code size may have led to cache inefficiencies or instruction cache overflows, negating the expected benefits of loop unrolling. On the other hand, we can observe a slight increase in the performance when running the application on Cortex-A57 and in the coalesced version on the Denver 2 .

3.3 Loop Unrolling on Y-axis (optim2)

Building on 'optim1', the next step in 'optim2' is to manually unroll the loop along the Y-axis, further reducing control overhead. By directly writing the operations for 'i-1', 'i', and 'i+1', we eliminate condition checks and jumps for each iteration. This effectively removes the Y-axis loop, reducing the number of branches and simplifying the code.

As a result, the code in 'optim2' runs faster than 'optim1', since we completely removed the loop in the Y direction. This reduces the number of control flow instructions, leaving only two loops in the final version. In theory, this should improve performance on processors like the Cortex-A57, which has a *limited branch prediction depth* (only 2 levels). However, the expected performance boost is not fully realized due to the presence of function calls inside the loop, which still introduce overhead. This shows that while unrolling loops reduces some control overhead, the presence of other factors, such as function calls, can still limit performance gains, as seen in the Cortex-A57. Finally, we observed a significant gain in the performance when running the application on Denver 2.

3.4 Inline Function Calls (optim3)

In 'optim3', the main optimization involves inlining the functions inside the loop. Normally, each function call incurs overhead from saving the current state, jumping to the function, and returning afterward. Inlining eliminates this overhead as the function code is directly inserted at the call point.

[1] Please note that the hardware accelerator in Denver 2 is optimized for AI workloads, such as convolution operations. Since our approach deviates from a standard convolution product, this may substantially diminish the performance benefits provided by the accelerator.

This optimization resulted in a noticeable reduction in execution time on the Cortex-A57, with a decrease of nearly 5 seconds. This suggests that the architecture is particularly sensitive to the overhead of function calls within loops, while the Denver 2 seems to be not affected by this optimization.

3.5 Variables Rotation (optim4)

In 'optim4', variable rotation along the X-axis is used to reduce redundant calculations and memory accesses. When processing pixels, the first and second columns of the 3x3 neighborhood overlap with the second and third columns of the previous neighborhood. By rotating variables, we avoid recalculating values for these overlapping columns. As the kernel slides to the right, the values from the middle column shift to the left, and the values from the right column shift to the middle, with only the rightmost column being recalculated.

Although memory accesses are non-coalesced in the Y-direction during each iteration, the horizontal sliding along the X-axis benefits from improved cache usage since the data for neighboring pixels is likely to remain in the cache due to the previous accesses on the same row. This optimization reduces redundant computations and improves overall performance, especially when processing images in a column-by-column manner.

With this optimization, we achieved the highest increase in performances, reaching a speedup of **2.35x** on Cortex-A57 and **2.15x** on Denver 2 with respect to *optim3*.

3.6 Manage Borders (optim5)

In this version, for pixels at the image edges, we apply the logic from 'blur_do_tile_default' to ensure accurate processing. For non-border pixels, we maintain the optimization method from *optim4*, using variable rotation and loop unrolling to reduce redundant calculations.

By restoring border handling, we ensure that edge pixels are processed correctly, even though this introduces some computational overhead. However, the overall performance did not degrade significantly in case of Cortex-A57, as the optimized variable rotation is still applied to the non-border regions and the number of the new pixel is of the order of n against n^2 pixel in total .

It is interesting to note that, in the case of Denver 2, there is a significant performance gain, with execution time being **cut in half** compared to *optim4*. This improvement is likely due to the hardware accelerator's optimizations when handling border operations and when handling a complete convolution.

3.7 [Bonus] Reduction (optim6)

We didn't get completely the question, but we think we may already implemented the column reduction in optim4.

3.8 [Bonus] Compare with Different Levels of Compiler Optimization

When running the code with different levels of optimization, we observed a substantial performance gain. Comparing the default version to *optim5*, compiled using the *-O3* flag, we achieved a speed-up of **25.96x** on the Cortex-A57 and **12.92x** on Denver 2.

Firstly, we expect to see a performance gain as the optimization level and kernel version are increased. This trend is evident in the case of Cortex-A57, as shown in both Figure 1 and Figure 2. However, in the case of Denver 2, we observe that increasing the optimization level can sometimes lead to a slight decrease in performance. This could be attributed to Denver 2's hardware accelerator, which may not fully benefit from certain optimizations applied at higher levels.

For each optimization of the kernel, we run the program **multiple times** to get the best (lowest) execution time. In this way, we can obtain a more accurate result, since the execution time can vary depending on the **system load.** Then we run the program using different compilation optimization flags to see how compiler optimization affects execution time.

Finally, every kernel version was **tested** by calculating the image hash (using easypap *-sh* **SHA256**). As a base result, we used the output of *default_nb*, after having verified visually the correctness of the result. We did it, especially when running the code with \ textit-O3 to ensure correctness. The compiler's optimizations are highly aggressive, and in some cases, they can produce incorrect results.

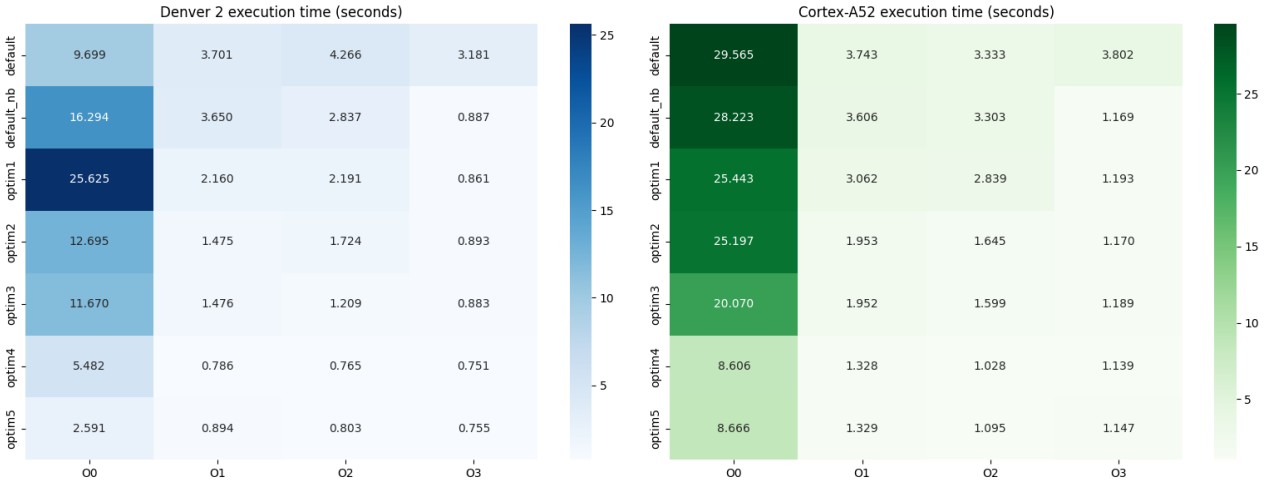

Figure 1: Heat map of the execution times of non coalesced version

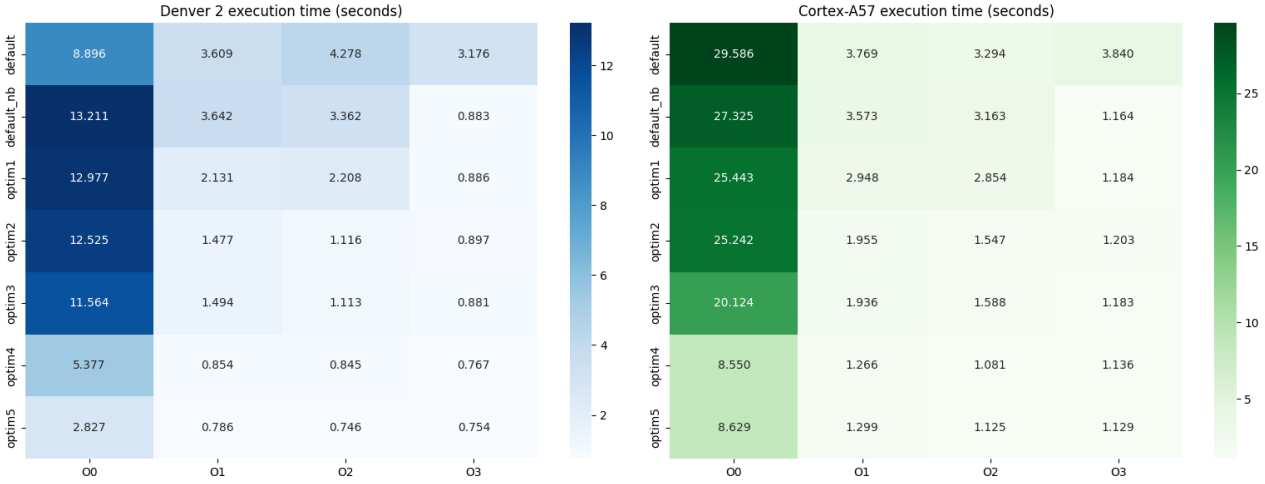

Figure 2: Heat map of the execution times of the coalesced version.

References

[Cor] Cortex-a57 arm developer. https://developer.arm.com/Processors/Cortex-A57. Accessed: 2024-09-19.

[Den] Denver 2 wikichip. https://en.wikichip.org/wiki/nvidia/microarchitectures/denver. Accessed: 2024-09-19.