18-645 Project: Canny Edge Detection

Zhen Guang Low, Balasubramanian Srinivasa Vaithiyanathan, Balasubramanyam Evani
{jzlow, balasubs, bevani}@andrew.cmu.edu

I. Introduction

Image Processing techniques are frequently utilized to enhance the quality and utility of images by isolating and emphasizing essential features or minimizing unwanted noise. These techniques play a pivotal role in a wide range of applications across various fields, including computer vision, medical imaging, remote sensing, and more. The current project implements Canny edge detector, an image processing method that detects a wide range of edges in an image. Edge detection plays a crucial role in computer vision, image processing, and pattern recognition. Canny edge detection can be broken into four crucial stages -

- Smoothing: The smoothing filter applies Gaussian filter which can be thought of as a 2-D convolution operator, which serves to soften images by eliminating fine details and noise
- Sobel Filtering: The Sobel filter calculates first order image gradients for both vertical and horizontal edges
- Non-max Suppression: Non-max suppression compares a pixel of interest with its surrounding pixels then set the central pixel value if it has a value larger values in the gradient direction
- Hysteresis Thresholding: The Hysteresis thresholding approach is very beneficial in differentiating the real edges from the one's which aren't by examining and comparing each pixel of an image with a defined High and Low threshold

Nonetheless, its computational demands can pose significant challenges in real-time scenarios, especially when dealing with extensive images or a vast image database with multiple entries. In the current project we have made use manual vectorization using SIMD-kernel implementation using Intel® x86 intrinsic functions that provide access to assembly instructions without writing assembly code on ece006.ece.local.cmu.edu having Intel ®Xeon ®CPU E5-2640 v4 @ 2.40GHz (Broadwell x86_64 architecture). Manual vectorization usually outperforms the compiler based automatic vectorization process as it provides finer-grained control over the generated instructions. We leverage the machine's AVX (Advanced Vector Extensions) capabilities, by applying integer AVX integer intrinsics to process 16-bit signed integers. We therefore achieve the ability to process sixteen 16-bit signed integer values simultaneously which is double the single precision floating-point operations.

To enhance the performance of our implementations, we first identify independent operations essential to our code and optimize their execution. By efficiently interleaving data loading and computation, we maximize the utilization of system resources, resulting in improved execution speed and overall computational efficiency. We also conduct a performance evaluation by comparing our OpenMP-based parallel kernel to a SIMD-based kernel across various image sizes. Additionally, we perform benchmarking against OpenCV counterparts, demonstrating the substantial performance improvements achieved by our custom implementations. Finally, we provide a concise summary of our kernel implementations, by presenting a graphical plot illustrating the peak throughput attained and future improvements that could be made.

The video for the project is uploaded here for grading purposes: [Video Link] . The following sections provides a detailed representation of kernels developed for the four stages of the Canny edge detection.

II. Kernel Designs

A. Sobel Filtering

Sobel edge detection is a technique used to identify edges in an image through a first order gradient-based approach. In the horizontal direction, a gradient (Gx) is computed by convolving the horizontal mask shown in Fig. 1 with the image. Similarly, a vertical gradient (Gy) is calculated by applying the vertical mask depicted in Fig. 1 in a similar convolution process. The gradient magnitude, is then determined using the following formula -

$$\nabla G = \sqrt{G_x^2 + G_y^2}$$

To mitigate the computational expense associated with the square root operation, we opted for a less computationally intensive approximation, which is commonly used in many implementations -

$$\nabla G = |G_x| + |G_y|$$

1) Sobel X Kernel: Sobel X filtering involves three steps - (1) Sliding the respective filter over the input image, (2) performing the weighted multiplication of the overlapping 3 × 3 region, and (3) Summing-up the calculated products. These operations are performed iteratively over the entire image. The kernel identified is -

$$\nabla G_x = |P_3 + 2 \times P_6 + P_9 - (P_1 + 2 \times P_4 + P_7)|$$

where, the P represents the pixel values overlapping the 3x3 Sobel X filter. The independent operations that make up the kernel are the individual weighted products performed between

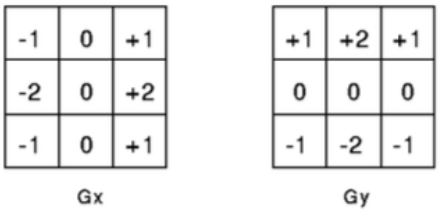

Fig. 1: (a) Sobel X Filter, (b) Sobel Y filter

the image and the filter at different positions. The dependent instructions that make up the independent operations are the add instructions that sum up the products of the overlapped region. In the current implementation *addition* and *subtraction* instructions are used to implement the weighted sums and products. The latency and throughput of instructions used in this kernel implementation are shown in below table. The

Instruction	Latency	Throughput	AVX intrinsic	Port
vlddqu	5	2	_mm256_lddqu_si256	1* p23
vpaddw	1	2	_mm256_add_epi16	1* p15
vpsubw	1	2	_mm256_sub_epi16	1* p15
vpabsw	1	2	_mm256_abs_epi16	1* p15

TABLE I: Sobel kernel instructions Latency and Throughput

bottleneck here is the subtract/add instructions and absolute value calculations which use the functional unit p15. The limited number of registers (assuming 16) and the loads-store operations would also pose to be a bottleneck in achieving the peak performance.

In this project, we have used SIMD instructions along with other optimization techniques which is described as follows to implement the kernel. First, the original image is padded with a value of 2, adding one pixel of padding to both the left and right sides (as shown in 2). This is done to ensure that all input image pixels are processed by the kernel and to reduce complexity. Secondly, the input data type is

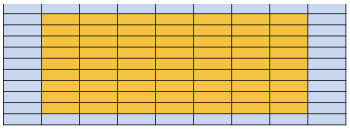

Fig. 2: Padded image, blue cells represent the padding, orange cells represent the original image

converted from unsigned 8-bit to 16-bit signed bit to ensure accuracy in calculations involving addition, subtraction, and ultimately, the absolute value. Thirdly, to maximize register usage, a kernel of size **6 x 16** is utilized, yielding a total of **96 outputs**. In total, **8 registers are reserved for loading inputs**, **6 registers for storing the output, and 1 register for storing the zero** value, which is useful in computing negative weights as illustrated in the Sobel X filter in Figure. 1. The described method, shown in Figure. 3, involves output loading O_{1-6} and input loading as follows: (1) Initially, the leftmost I_{1-8} is loaded to compute the left column Sobel X weights, which are then added to the outputs. (2) Subsequently, the 8 input registers are reused to load the right side pixels of the outputs, used to compute the right column Sobel X weights, and added to the output. Finally, once the 96 outputs are calculated, their absolute values are taken for storage. To further optimize performance, several considerations have

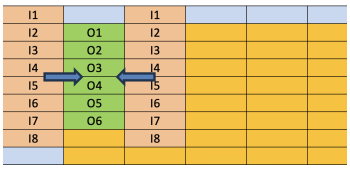

Fig. 3: Loading of inputs and outputs

been implemented. Firstly, as demonstrated in Table I, the function $\_mm256\_lddqu\_si256$ is utilized for loading purposes instead of $\_mm256\_load\_si256$. This choice is based on the potential performance benefits when data crosses a cache line boundary. Secondly, as observed in the same table, the primary computational bottleneck is identified in the operations of addition, subtraction, and absolute value calculations, since they all utilize the same functional unit. To mitigate this issue, the approach adopted involves calculating the output as soon as the input and output data become available to keep the identified functional unit busy. This strategy leads to a process of pairwise loading and computation. An example of this method, taken directly from the source code, is illustrated in Figure 4.

```
i1 = _mm256_lddqu_si256((__m256i *)(inputPointer + (i - 1) * N + j - 1));
i2 = _mm256_lddqu_si256((__m256i *)(inputPointer + i * N + j - 1));
i1 = _mm256_sub_epi16(zeros, i1);
i2 = _mm256_sub_epi16(zeros, i2);


o1 = _mm256_lddqu_si256((__m256i *)(outputPointer + i * N + j));
o2 = _mm256_lddqu_si256((__m256i *)(outputPointer + (i + 1) * N + j));
o1 = _mm256_add_epi16(i1, o1);
o2 = _mm256_add_epi16(i2, o2);
```

Fig. 4: Pair wise loading and compute

2) Sobel Y kernel: The Sobel Y filtering process encompasses the same three steps as identified in the Sobel X filtering, with the primary difference being the alteration in weights. The kernel identified for this is -

$$\nabla G_y = |P_1 + 2 \times P_2 + P_3 - (P_7 + 2 \times P_8 + P_9)|$$

where, the P represents the pixel values overlapping the 3 $\times$ 3 Sobel Y filter. The bottlenecks and optimizations mentioned in the previous section are applicable here as well, with the kernel size maintained at 6 $\times$ 16, producing 96 outputs. Similar to the Sobel X filtering, 8 input registers and 6 output registers are used, with the 8 input registers being reused. The key difference lies in the method of loading the inputs. This process is illustrated in Figure 5. The 6 input registers are utilized three times, with the numbers indicating the sequence of loading and I_{1-8} representing the starting positions for each load.

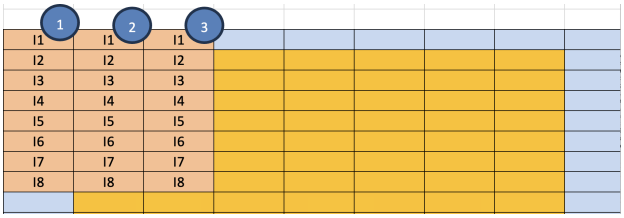

Fig. 5: Sobel Y kernel input loading

3) Final Addition Kernel: Finally, to accumulate the gradients calculated previously using the Sobel X and Sobel Y kernels, a third and final addition kernel is introduced. This kernel is distinctively defined with a larger **kernel size of 192**. The key independent operations in this phase are the individual pixel sums, which can be efficiently executed by utilizing the full capacity of **12 output registers** and **2 input registers**, with the latter being reused. A similar optimization strategy is applied here, where the add computations are carried out as soon as the input becomes available. This approach is designed to keep the add functional unit bottleneck engaged as much as possible. A pictorial representation is given in Figure. 6.

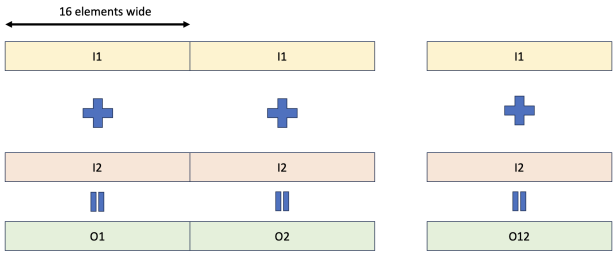

Fig. 6: Addition Kernel to calculate total gradient

4) Results and Discussion: **Throughput:** After adding instructions, we must take the absolute value before calculating the final output for both the Sobel X and Sobel Y kernels. The peak throughput is calculated based on the throughput of $\_mm256\_abs\_epi16$, which is 2, meaning that it can process 32 pixels in parallel. However, because we need to wait for weighted sums (which take 6 cycles), the peak throughput can be calculated as 32 / 6 = 5.3 pixels per cycle. The plot showcasing the achieved throughput for Sobel X and Sobel Y is shown in Figure. 7. The throughput decreases

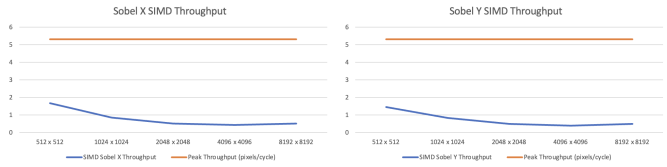

Fig. 7: Sobel X, Y Throughput Plot

as the image resolution increases, indicating the need for better memory alignment. This alignment improvement was attempted in the current project and is discussed in the next subsection. In addition to the throughput plot, experiments on the number of cycles taken for both the OpenCV SIMD Kernel implementation and the parallel SIMD Kernel (using OpenMP) were carried out. In general, the SIMD Kernel implementation outperforms OpenCV with an 11.21x speedup. However, for the parallel SIMD implementation (utilizing OpenMP's parallel for with a fixed number of threads set to 8), itś important to note that, although the results in Table 8 show a minor speedup compared to the base SIMD kernel, multiple runs reveal that this difference is not always consistent. In some cases, we observed performance degradation for lower-resolution images, but there is some noticeable improvement for resolutions of 8192 $\times$ 8192 and above.

Sobel Filtering					
Image Size	Runs	CPU cycles SIMD	CPU cycles OpenCV	xSpeedup	Parallel SIMD
512 x 512	100	539009	1.07E+07	19.85x	503253
1024 x 1024	100	3.73E+06	4.74E+07	12.7x	2.73E+06
2048 x 2048	100	1.92E+07	1.77E+08	9.21x	1.79E+07
4096 x 4096	100	9.74E+07	8.12E+08	8.34x	9.44E+07
8192 x 8192	10	3.92E+08	3.45E+09	8.8x	3.60E+08
16384 x 16384	10	1.61E+09	1.35E+10	8.39x	1.55E+09

Fig. 8: SIMD Sobel vs OpenCV vs Parallel SIMD Sobel

The qualitative comparison of output results of the SIMD Sobel and OpenCV is show in Figure. 9 which shows the correctness of the implementation. For code please check **sobelX_SIMD**, **sobelY_SIMD**, functions in **sobellib.cpp**.

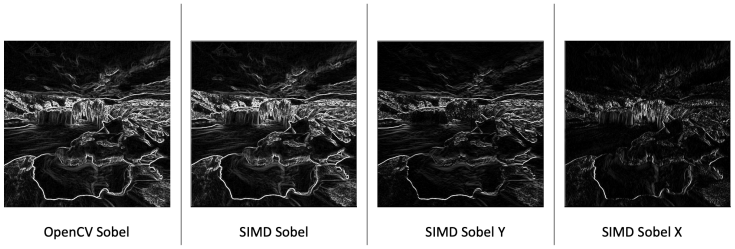

Fig. 9: OpenCV Sobel vs SIMD Sobel Qualitative results

5) Future Improvements - Memory Design: The current implementation uses consecutive rows. If the row width is large enough, these kinds of loads might lead to performance degradation due to cache misses. Hence, one potential solution we considered to mitigate this issue was to arrange these row elements in consecutive order. We attempted this approach in the current project, however, it did not improve performance. Instead, we observed degradation in both the number of cycles and throughput. The reasons for this are not known, and if the semester were longer, we might have had time to understand why this happened. For now, it remains outside the scope of our investigation.

B. Gaussian Smoothing

In this project we have implemented a distinct Gaussian-shaped filter, specifically the 3 $\times$ 3 version. shown in Figure. 10. The independent operations involve the convolutions performed between the image and the filter at different pixel

1/16	1/8	1/16
1/8	1/4	1/8
1/16	1/8	1/16

Fig. 10: Gaussian Filter

positions. The dependent instructions that constitute these independent operations are the 'add' instructions, which sum up the products of the overlapped image and filter regions. The kernel for Gaussian smoothing was carried out similarly to Sobel filtering, as discussed in the previous section.

The final kernel for Gaussian smoothing, similar to Sobel X and Sobel Y, uses a 6x16 kernel, producing 96 outputs by utilizing 8 input registers and 6 output registers. We follow a similar instruction pattern as shown in II, and we use $\_mm256\_srli\_epi16$ on **functional port 1*p0**, which performs a logical right shift with a latency of 1 and throughput of 1. This shift operation is used for division, as all the weights in the overlapping region involve division by multiples of 2. The inputs are loaded in a manner similar to what is shown in Figure 5. Additionally, similar to the optimization approach applied to Sobel, instruction interleaving is performed to initiate the main computation, which is carried out using add instructions, as soon as possible.

1) Results and Discussion: The peak throughput is calculated based on the throughput of $\_mm256\_add\_epi16$, which is 2, meaning that it can process 32 pixels in parallel. However, because we need to wait for weighted sums (which take 9 cycles), the peak throughput can be calculated as 32 / 9 = 3.5 pixels per cycle. The plot showcasing the achieved throughput is shown in Figure. 11. The throughput decreases

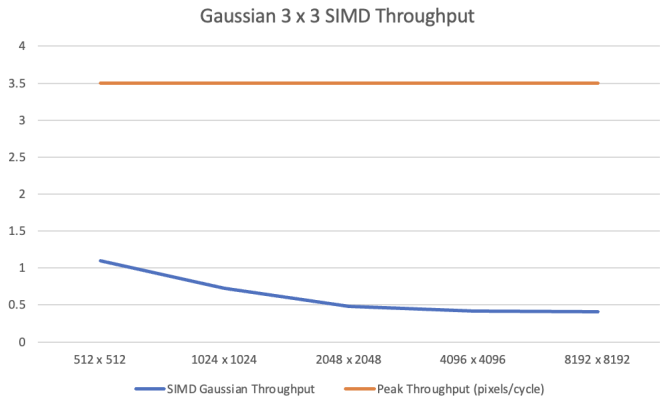

Fig. 11: Gaussian Throughput plot

as anticipated when the image dimensions increase, a trend we believe is due to the same reasons discussed in the Sobel section. Additionally, a comparative analysis of the number of CPU cycles required for SIMD Gaussian and OpenCV Gaussian is conducted, as illustrated in Figure 12. It is evident that the SIMD Gaussian achieves an average speedup of 2.2x. Furthermore, while the parallel Gaussian outperforms the base SIMD Gaussian slightly, but mutliple runs show that the benefits of parallelization do not appear to be consistently guaranteed.

3 x 3 Gaussian Smoothing					
Image Size	Runs	CPU cycles SIMD	CPU cycles OpenCV	xSpeedup	Parallel SIMD
512 x 512	100	310699	1.21E+06	3.9x	209320
1024 x 1024	100	1.61E+06	3.98E+06	2.47x	1.15E+06
2048 x 2048	100	6.57E+06	1.28E+07	1.94x	6.40E+06
4096 x 4096	100	4.12E+07	5.22E+07	1.26x	3.94E+07
8192 x 8192	10	1.80E+08	3.41E+08	1.9x	1.52E+08
16384 x 16384	10	6.90E+08	1.37E+09	1.9x	6.84E+08

Fig. 12: SIMD Gaussian vs OpenCV vs Parallel SIMD Gaussian

Finally, the qualitative results comparing SIMD gaussian and OpenCV gaussian is show in Figure. 13 which shows the correctness of the implementation. For source code please check **gaussianBlur3x3_SIMD** function in **sobellib.cpp**

Fig. 13: OpenCV vs SIMD Gaussian Qualitative results

C. Non-max Suppression

Non-max suppression compares a pixel of interest with its surrounding pixels then set the central pixel value if it has a lower value in the gradient direction. NMS is implemented by first finding the gradient direction by comparing the magnitude of each of 8 quadrants (denoted by 8 surrounding pixels). Next, the central pixel value will be compared with the values of the surrounding pixels of the gradient direction. Lastly, the central pixel value will be set to 1 if it is smaller than the each opposing value; otherwise, it will be set to 0. This indicates the edge present in the current pixel and hence will be set to 1.

In order to achieve better performance, independent operations must be identified then parallelize them as much as possible. The independent operations in this algorithm are the loading of pixel values. Due to the sequential nature of NMS, the dependent operations are the comparison of the gradient magnitudes, comparison of the pixel of interest against the central pixel and setting the central pixel value. In order to achieve good throughput, we have to ensure that the comparison done are as little sequential as possible.

In this algorithm, a naive implementation requires nested if conditions to compare between the different directions. A

sample code can be found in Listing 1 in Appendix A to illustrate this. In this example, there lies multiple considerations for speedup. This implementation consist of nested-if condtions that can lead to a massive slowdown due to branch mispredictions. In addition, the need to check the values of each pixel and its surrounding values can lead to further slowdown as fetching (assuming row major order) has to be done for each row of 3 pixels.

It is important to control the comparisons done with the neighbouring pixels to ensure that there is little branching conditions that may happen in the design. This is important as good SIMD performance can be impacted by conditional branching when these branching are almost entirely random and dependent on the image vectors and gradients. By assuming that the matrix values would be kept in row-major order, 16 3x3 kernels will be processed at the same time by loading aligned bits of these matrix into a SIMD register. To reduce branching in this algorithm, OR logic will be used instead to propagate edges that are found so that pixels can be set accordingly. Below details the steps to the proposed kernel.

Figure 14 shows an overview of the operations that have to be executed to complete each output.

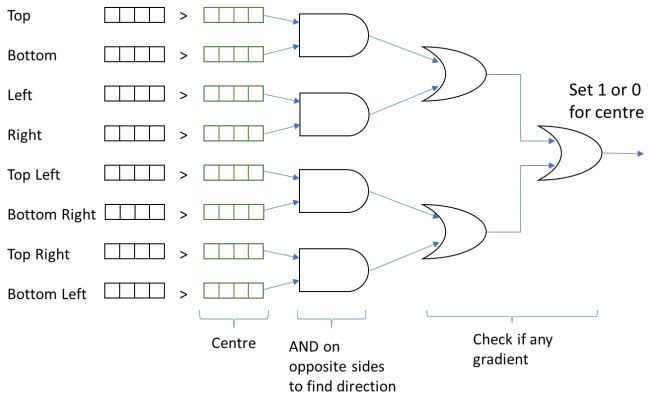

Fig. 14: Overview of proposed vectorisation of non-max suppression

In a given 3x3 pixel:

Top Left	Top	Top Right
Left	Centre	Right
Bottom Left	Bottom	Bottom Right

There is a minimum of 15 operations (8 Compare, 4 AND, 3 OR) and this requires a minimum of 9 output registers for each output computed. To reduce the number of output registers, the registers will be reused as there are independent compare, AND and OR operations. By maximizing the number of output registers used here, the latency of each kernel can be reduced by ensuring that each functional units are at max throughput via instruction level parallelisation (ILP) by pipelining instructions as much as possible. This is further explained in the implementation.
Loading sequence in each SIMD register:

Register 1	Top Left 1	...	Top Left 15	Top Left 16
Register 2	Top 1	...	Top 15	Top 16
...	...	...	...	...
Register 8	Bottom 1	...	Bottom 15	Bottom 16
Register 9	Bottom Right 1	...	Bottom Right 15	Bottom Right 16

1) Load 16 pixels of Top Left, Top, Top Right, Left, Centre, Right, Bottom Left, Bottom and Bottom Right in each SIMD register.
2) Compare each of the directional pixel with centre to find where anything more than the centre values will be set to 1 otherwise it will return a mask of 0.
3) Use a logical AND operation on the returned values of Top and Bottom, Left and Right, Top Left and Bottom Right, and Top Right and Bottom Left to find the gradient direction.
4) A series of 2 OR operations can be used on the results of Top and Bottom, Left and Right and Top Left and Bottom Right, and Top Right and Bottom Left to get the final direction of the matrix

Pixel reuse is also implemented by discarding pixel values in the left column after compare is executed and loading an addition column to the right of the right column. This is illustrated below:

Top Left(discard)	Top	Top Right	Load New
Left(discard)	Centre	Right	Load New
Bottom Left(discard)	Bottom	Bottom Right	Load New

In addition to branching conditions, the way images are loaded have to be considered carefully. This implementation has been benchmarked against the Intel Xeon E5-2640 v4 which has a cacheline of 64 bytes. This means that in pixels that are 16bits wide, each cacheline would be able to hold 32 pixels each. This means that of each 16 pixels loaded in each register, loading consecutive two registers would yield a greater utilization of each cacheline in the L1 cache. However, the loading sequence requires discarding the left column after processing in order to load the next set to right pixels. In ideal conditions, this would mean that the values would still be available in the cache before these values are replaced under LRU.

When designing this kernel, operations are scheduled and designed by optimizing two metrics: functional unit utilization and independent operations. From the above explanation, the instructions are executed with the following parameters:

- `_mm256_loadu_si256` has a latency of 3 and throughput of 2 on functional unit p23
- `_mm256_cmpgt_epi16` has a latency of 1 and throughput of 2 on functional units p15 and p23
- `_mm256_and_si256` has a latency of 1 and throughput of 3 on functional units p15
- `_mm256_or_si256` has a latency of 1 and throughput of 3 on functional units p15

In this implementation, the identified bottleneck would be the use of functional unit p15. On each CPU, it is only possible to execute 3 instructions on the p15 unit at any time. This resource conflict is also shared with the `_mm256_cmpgt_epi16` which can also run on the p15 unit.

As the _mm256_or_si256 function is at the end of the processing chain, this instruction throughput will be the the bottleneck. Therefore, the theoretical peak achievable in this kernel would be 1*16=16 operations per cycle, or a throughput of 16. Given that the implementation is running on the Intel Xeon e5-2640 v4 CPU running at 2.4MHz - 3.4 MHz, this means that the theoretical peak throughput of this kernel would be

$$Throughput(GOPS) = 16*(2.4e9+3.4e9)/2 = 46.4GOPS \quad (1)$$

The theoretical peak of the kernel is obtained by ensuring no resource conflicts and maximizing the functional units used. This kernel is requires a series of operations that lead to the final_mm256_or_si256 to compute the output. As a result, the theoretical peak can only be the fastest at which this function can produce outputs.

Listing 2 shows the kernel of the implemented design. The kernel can also be separated into 3 sections: the prologue, steady state and the epilogue. The prologue runs a 3x3 kernel on the very first pixel of each row while preparing registers with pixels that can be reused. The steady state phase takes these pixels and reuses the middle and right column pixels for the following pixel in row-major order. This allows a recursive kernel that repeats itself till the end of the row. The kernel will end one step before the end for the epilogue where no more pixels will be loaded and computation will finish before moving to the next row. The same process of prologue, steady state and epilogue will begin again for every row until the second last row.

With the kernel above, performance is measured against a opencv implementation of non-max suppression. The graphs below compares the performance of our implementation against the opencv implementation as well as show the kernel's performance against increasing problem size.

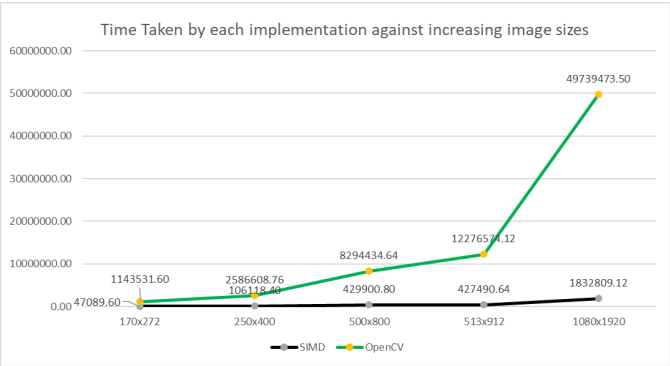

Fig. 15: Time taken by SIMD and OpenCV implementation against varying image sizes

Estimated operations per seconds is derived by dividing the calculated operations per second (OPS) with the time taken. Calculated operations per seconds is calculated using:

$$OPS = (rows-1)*(cols-16)*(8CMP+4AND+3OR) \quad (2)$$

$$OPS = (rows-1)*(cols-16)*(15) \quad (3)$$

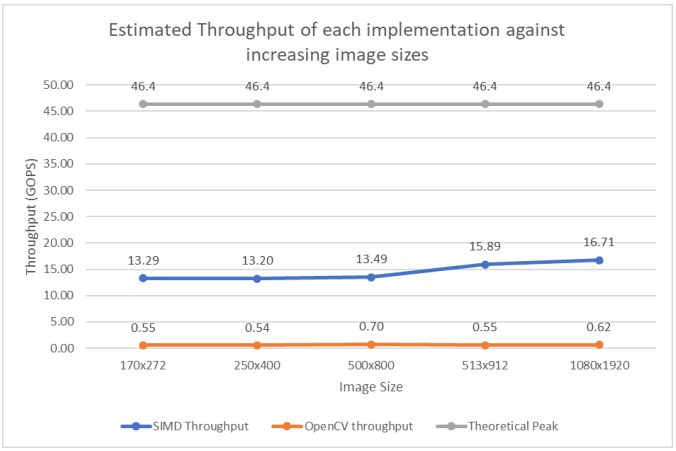

Fig. 16: Estimated throughput of SIMD and OpenCV implementation against varying image sizes

Below shows an example of the comparison of our SIMD implementation compared against a OpenCV implementation of NMS.

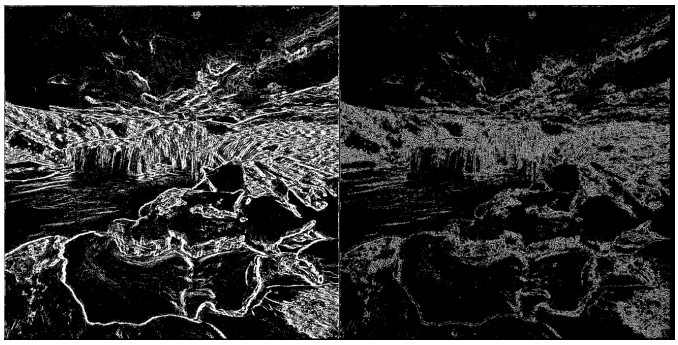

Fig. 17: Comparison of image output of nms implementations in SIMD(left) and OpenCV(right)

According to results measured above, there is an estimated $\approx$ 20 to $\approx$ 30 speedup in the SIMD against the OpenCV implementation.

Alternatively, the instructions can be scheduled differently by optimizing the number of independent instructions. This can be done by pairing each load instruction (except the center pixel) with a corresponding _mm256_cmpgt_epi16.

The pseudo algorithm can be found in Appendix A. In this alternative implementation, a total of 6 registers is used with an estimated throughput of $\approx$10 cycles required for every output. Pipelining is possible starting with the 7th cycle as the _mm256_loadu_si256 would be occupied until then. This makes pipelining very inefficient with the bubbles/gaps happening during pipelining. By attempting to reduce the register usage to increase the number of independent operations would not be very successful as the functional units would be at high utilization and would not produce much improved throughput over an implementation that focuses on maximising utilization of functional units.

Moving forward, more can be studied on how to calculate the local maxima and minima to produce a more accurate non-max suppression. This portion has been excluded in this

implementation as it was deemed too exhaustive in this section. However, gradient direction can be included via a separate masked vector which can be used with a logical AND to mask directions that will not be relevant for the algorithm.

D. Hysteresis thresholding

The **Hysteresis thresholding** approach is very beneficial in differentiating the real edges from the one's which aren't by examining and comparing each pixel of an image with a defined **'High'** and **'Low'** threshold. Any edges with intensity greater than 'High' are the sure edges and those with intensity less than 'Low' are considered to be non-edges. The edges with intensity between 'High' and 'Low' thresholds are classified as edges only if they are connected to a sure edge otherwise discarded.

In order to achieve maximum performance and throughput, the **independent operations** for the kernel must be identified. The independent instructions are derived from the chain of dependent instructions (independent operations), which are further interleaved and parallelized as much as possible to avoid bubble in the pipeline. The independent operations in this algorithm involved are the loading of pixel values and the comparisons that are performed between each pixel intensity with the set threshold .

When designing this kernel, operations are scheduled and designed by optimizing two metrics: functional unit utilization and independent operations.

AVX intrinsic	Latency	Throughput	Port
_mm256_loadu_si256	3	2	p23
mm256_cmpgt_epi16	1	2	p15
_mm256_or_epi16	1	3	p15

TABLE II: Hysteresis kernel instructions Latency and Throughput

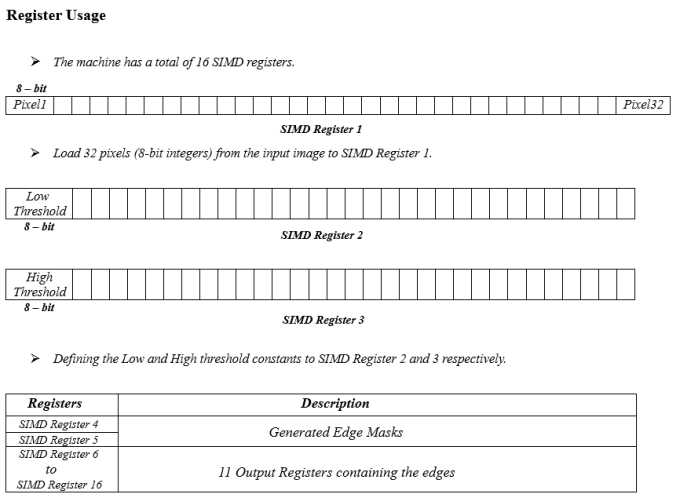

Fig. 18: Register Usage

For a given non-max suppressed image :

1) Load Image Data: Load the 16 (16-bit) pixels into the SIMD register. Note - need to ensure that the data is aligned for SIMD operations. The input register is further re-used in subsequent computations.
2) Set Threshold Values: Define your high threshold (T_high) and low threshold (T_low) values.
3) SIMD Operations: Use SIMD instructions to compare each pixel's intensity value with T_high and T_low simultaneously.
4) SIMD instructions allow you to perform these comparisons in parallel for multiple pixels.Create two masks, one for pixels above T_high (strong edges) and one for pixels with T_low (no edges).
5) Bitwise operations are performed between the strong edge and low edge mask to generate the final edge map.

Loading	High Th	Low Th	Bitwise-OR
vmovdqa	vpcmpgtb	vpcmpgtb	vpor

Fig. 19: Dependent Instruction chain (independent operation)

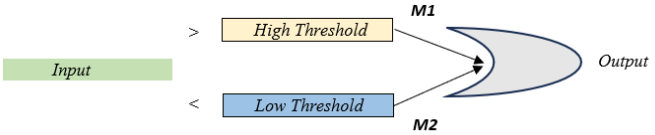

Fig. 20: Overview of proposed vectorization of Hysteresis Thresholding

Loading images into a cache can significantly improve the performance and efficiency of kernel, especially in scenarios where image pixels are repeatedly accessed or displayed. Caching images offers several advantages. This implementation has been benchmarked against the `Intel Xeon E5-2640 v4` which has a cacheline of 64 bytes. This means that in pixels that are 16 bits wide, each cacheline would be able to hold 32 pixels each. This means that of each 16 pixels loaded in each register, loading consecutive two registers would yield a greater utilization of each cacheline in the L1 cache.

The size of the kernel is **176** . Since we have a total of **11** simd output registers each containing **16** pixels within it, a total of **176** (**11** simd registers * **16** pixels each = **176**) output pixels containing information about the edges are computed and stored. This is achieved by re-using some of the input registers and derived threshold mask registers.

The theoretical peak of the kernel is obtained by ensuring no resource conflicts and maximizing the functional units and available registers used. Hysteresis Theresholding requires two `_mm256_cmpgt_epi16` operation (throughput of 2, latency 1, P15) and 1 `_mm256_or_si256` (throughput of 3, latency 1, P15) operation to generate a single output. Therefore, the theoretical peak achievable in this kernel would be 3 instructions in 2 cycles = 1.5 instructions/cycle or (24 operations per cycle).

In this implementation, the identified bottleneck would be the usage of functional unit p15. There is a resource

conflict caused by the instructions `_mm256_cmpgt_epi16` and `_mm256_or_si256` , both of which end up being executed on the same functional unit. Moreover, the instruction `_mm256_or_si256` is dependent on the mask generated by `_mm256_cmpgt_epi16` and therefore ends up being the the bottleneck of my kernel

$$Throughput(IOPS) = 3 * 16/2 = 24 \tag{4}$$

Based on the implemented kernel design, an intermediate data layout is not required. Intermediate data layout allows blocks of data to be stored within L1 or L2 cache for faster retrieval if they are not replaced in cache. In this implementation, pixel data used are relative to its position and the same data will be accessed when going down the number of rows. Should intermediate data layout be used, it will further complicate the logic for data retrieval as blocks of data will consist of repeated information from the row before and after.

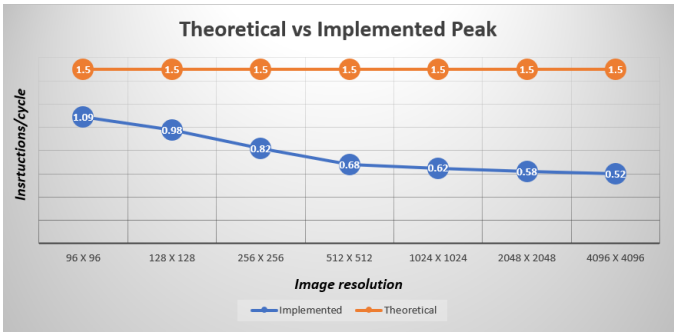

Fig. 21: A comparison of implemented results against theoretical peak performance.

The performance of the designed kernel is measured and benchmarked against a opencv implementation of canny-hysteresis thresholding. The graphs below compares the performance of our implementation against the opencv implementation as well as show the kernel's performance against increasing image resolution.

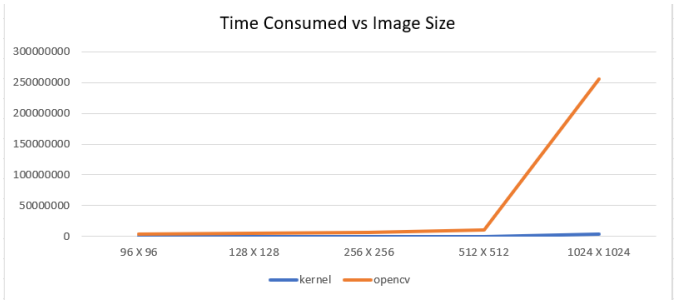

Fig. 22: Hysteresis Benchmarking

Though the performance of the kernel was around 70 times better than the opencv function there seems to be a compromise in the output at certain regions of the image . Future work, involves handling the corner cases when the neighbouring pixels are distributed over multiple registers which at times becomes to very difficult to handle connected edges.

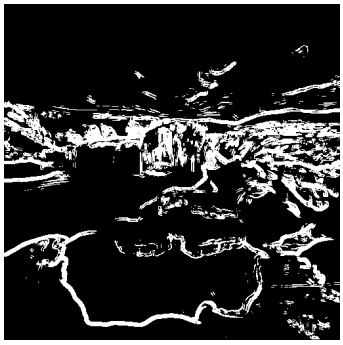

Fig. 23: Kernel Output

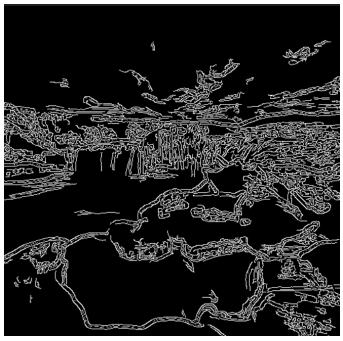

Fig. 24: OpenCv Output

In conclusion, leveraging SIMD instructions can significantly enhance the performance of hysteresis thresholding algorithms by exploiting parallel processing capabilities inherent in modern CPUs, thereby accelerating image processing tasks and improving computational efficiency.

III. Appendix

A. *Non-max Suppression*

Listing 1: Naive implementation of non-max suppression

```
for (int i=1;i<m-2;i++){
  for (int j=1; j<n-2; j = j++){
    int M = i*n + j;
    if (a[M-1-m] > a[M]){   //top left
        if(a[M+1+m] > a[M]))  { //bottom right
            b[M] = 1;
        }
        b[M] = 0;
    }
    else if (a[M+1-m] > a[M]){ //top right
        if (a[M-1+m] > a[M]) { //bottom left
            b[M] = 1;
        }
        b[M] = 0;
    }
    else if ((a[M-m] > a[M]) { //top
        if (a[M+m] > a[M]) { //bottom
            b[M] = 1;
        }
        b[M] = 0;
    }
    else if (a[M-1] > a[M]){ //left
        if (a[M+1] > a[M])) { //right
            b[M] = 1;
        }
        b[M] = 0;
    }
```

```
        else {
          b[M] = 0;
        }
      }
    }
```

Algorithm 1 Alternative fast code

function NON_MAX_SUPPRESSION
Define $input[m*n]$, $output[n*n]$
for $i \in [1, m-2)$ **do**
for $j \in [1, n-16)$ **do**
$ymm0 = load\ C$
$ymm1 = load\ TL$

$ymm2 = load\ BR$
$ymm3 = load\ T$
$ymm1 = COMPARE - TL$

$ymm4 = load\ B$
$ymm5 = load\ TR$
$ymm2 = COMPARE - BR$
$ymm3 = COMPARE - T$

$ymm2 = AND\ TL\ BR$
$ymm4 = COMPARE - B$
$ymm5 = COMPARE - TR$
$ymm1 = load\ BL$

$ymm4 = AND\ T\ B$
$ymm3 = load\ L$
$ymm6 = COMPARE - BL$

$ymm5 = AND\ TR\ BL$
$ymm4 = OR\ TL\ BR\ T\ B$
$ymm4 = COMPARE - L$
$ymm6 = load\ R$

$ymm6 = COMPARE - R$

$ymm4 = AND\ R\ L$

$ymm4 = OR\ TR\ BL\ L\ R$

$ymm4 = OR\ lastOR$
end for
end for
end function

Listing 2: SIMD Implementation of NMS where m and n represent the height and width of input image

```
for (i=1;i<m-2;i++){
  int M = i*n + 1;
  __m256i ymm0 = _mm256_loadu_si256((__m256i*) &a[M]);

  __m256i ymm1 = _mm256_loadu_si256((__m256i*) &a[M-16-n]); //TL
  __m256i ymm2 = _mm256_loadu_si256((__m256i*) &a[M+16+n]); //BR
  __m256i ymm3 = _mm256_loadu_si256((__m256i*) &a[M-16]); //L
  __m256i ymm4 = _mm256_loadu_si256((__m256i*) &a[M+16]); //R
  __m256i ymm5 = _mm256_loadu_si256((__m256i*) &a[M-16+n]); //BL

  __m256i ymmC1 = _mm256_cmpgt_epi16 (ymm0,ymm1); //compare TL
  __m256i ymmC2 = _mm256_cmpgt_epi16 (ymm0,ymm2); //compare BR

  ymm1 = _mm256_loadu_si256((__m256i*) &a[M+16-n]); //TR
  __m256i ymm6 = _mm256_loadu_si256((__m256i*) &a[M-n]); //T

  __m256i ymmA1 = _mm256_and_si256(ymmC1,ymmC2); //and TL BR
  __m256i ymmC3 = _mm256_cmpgt_epi16 (ymm0,ymm3); //compare L
  __m256i ymmC4 = _mm256_cmpgt_epi16 (ymm0,ymm4); //compare R

  ymm3 = _mm256_loadu_si256((__m256i*) &a[M+n]); //B
  __m256i ymmA2 = _mm256_and_si256(ymmC3,ymmC4); //and L R
  ymmC1 = _mm256_cmpgt_epi16 (ymm0,ymm5); //compare BL
  ymmC2 = _mm256_cmpgt_epi16 (ymm0,ymm1); //compare TR

  ymmC3 = _mm256_cmpgt_epi16 (ymm0,ymm6); //compare T
  ymmC4 = _mm256_cmpgt_epi16 (ymm0,ymm3); //compare B
  __m256i ymmA3 = _mm256_and_si256(ymmC1,ymmC2); //and BL TR
  __m256i ymmO1 = _mm256_or_si256(ymmA1,ymmA2); //or first 2
  ymm5 = _mm256_loadu_si256((__m256i*) &a[M+32]); //R next
  ymmA2 = _mm256_loadu_si256((__m256i*) &a[M+32-n]); //TR next

  ymmA1 = _mm256_and_si256(ymmC3,ymmC4); //and BL TR
  __m256i ymmO2 = _mm256_or_si256(ymmA3,ymmA1); //or bottom 2
  ymmC1 = _mm256_loadu_si256((__m256i*) &a[M+32+n]); //BR next
  ymmC2 = _mm256_cmpgt_epi16 (ymm4,ymm1); //compare T next
  ymmC3 = _mm256_cmpgt_epi16 (ymm4,ymm2); //compare B next

  ymmA1 = _mm256_or_si256(ymmO1,ymmO2); //last or
  ymmC4 = _mm256_cmpgt_epi16 (ymm4,ymm0); //compare L next
  ymmO1 = _mm256_cmpgt_epi16 (ymm4,ymm6); //compare TL next

  _mm256_storeu_si256((__m256i*) &b[M],ymmA1);
  ymmO2 = _mm256_cmpgt_epi16 (ymm4,ymm3); //compare BL next
  ymm0 = _mm256_cmpgt_epi16 (ymm4,ymm5); //compare R next
  ymm6 = _mm256_and_si256(ymmC2,ymmC3); //and T B

  for (j=17; j<n-31; j = j +16){
    M = i*n + j;
    // ymm4 = _mm256_loadu_si256((__m256i*) &a[M]); //R
    ymmA3 = _mm256_and_si256(ymmC4,ymm0); //and L R
    ymmC2 = _mm256_cmpgt_epi16 (ymm4,ymmA2); //compare TR
    ymmC3 = _mm256_cmpgt_epi16 (ymm4,ymmC1); //compare BR

    ymmA1 = _mm256_and_si256(ymmC2, ymmO2); //and TR BL
    ymmC4 = _mm256_and_si256(ymmC3, ymmO1); //and BR TL
    ymmO1 = _mm256_or_si256(ymm6,ymmA3); //first or
    ymm6 = ymm1;
    ymm0 = ymm4;
    ymm3 = ymm2;
    ymm1 = ymmA2;
    ymm4 = ymm5;
    ymm2 = ymmC1;
    ymm5 = _mm256_loadu_si256((__m256i*) &a[M+32]); // next R
    ymmA2 = _mm256_loadu_si256((__m256i*) &a[M+32-n]); // next TR


    ymmC1 = _mm256_loadu_si256((__m256i*) &a[M+32+n]); // next BR
    ymmO2 = _mm256_or_si256(ymmA1,ymmC4); //second or
    ymmC2 = _mm256_cmpgt_epi16 (ymm4,ymm1); //compare T next
    ymmC3 = _mm256_cmpgt_epi16 (ymm4,ymm2); //compare B next

    ymmA1 = _mm256_or_si256(ymmO1,ymmO2); //last or
    ymmC4 = _mm256_cmpgt_epi16 (ymm4,ymm0); //compare L next
    ymmO1 = _mm256_cmpgt_epi16 (ymm4,ymm6); //compare TL next

    _mm256_storeu_si256((__m256i*) &b[M],ymmA1);
    ymmO2 = _mm256_cmpgt_epi16 (ymm4,ymm3); //compare BL next
    ymm0 = _mm256_cmpgt_epi16 (ymm4,ymm5); //compare R next
    ymm6 = _mm256_and_si256(ymmC2,ymmC3); //and T B
  }

  if (j >= n-31 && j+16 < n){
    M = i*n + j;
    ymmA3 = _mm256_and_si256(ymmC4,ymm0); //and L R
    ymmC2 = _mm256_cmpgt_epi16 (ymmA2,ymm4); //compare TR
    ymmC3 = _mm256_cmpgt_epi16 (ymmC1,ymm4); //compare BR

    ymmA1 = _mm256_and_si256(ymmC2, ymmO2); //and TR BL
    ymmC4 = _mm256_and_si256(ymmC3, ymmO1); //and BR TL

    ymmO1 = _mm256_or_si256(ymm6,ymmA3); //first or
    ymmO2 = _mm256_or_si256(ymmA1,ymmC4); //second or
    ymmA1 = _mm256_or_si256(ymmO1,ymmO2); //last or

    _mm256_storeu_si256((__m256i*) &b[M],ymmA1);
  }
}
```

B. *Hysteresis Thresholding*

Listing 3: SIMD Implementation of the Hysteresis Thresholding Kernel

```
cv::Mat ht_window(cv::Mat &inputImage, uint16_t lowThreshold, uint16_t highThreshold)
{
      // convert input to CV_16S
    cv::Mat outputImage;
    outputImage.create(inputImage.size(), inputImage.depth());

    uint16_t *input = inputImage.ptr<uint16_t>();
    uint16_t *output = outputImage.ptr<uint16_t>();
      //uint16_t *output = edge_image.ptr<uint16_t>();
    const int imageSize = inputImage.rows * inputImage.cols;
    const int kernelSize = 176;  // (11 * 16pixels)

    // Calculate the number of iterations needed for the entire image
    const int numIterations = imageSize / kernelSize;

    // Load the thresholds into vectors
    __m256i lowThresholdVector = _mm256_set1_epi16(lowThreshold);
    __m256i highThresholdVector = _mm256_set1_epi16(highThreshold);
      __m256i lowMask;
      __m256i highMask;
```

```
        __m256i out1, out2, out3, out4, out5, out6, out7, out8, out9, out10, out11;
        __m256i inputVector;
    // Loop over the image in vectorized chunks
    for (int i = 0; i < numIterations; ++i)
        {
        // Load 32 pixels at a time
        //inputVector = _mm256_loadu_si256((__m256i*)&input[i * vectorSize]);
                inputVector = _mm256_loadu_si256((__m256i*)(input + (i*kernelSize)));
        // Compare against the low and high thresholds
        lowMask = _mm256_cmpgt_epi16(inputVector, lowThresholdVector);
        highMask = _mm256_cmpgt_epi16(inputVector, highThresholdVector);

        // Combine the masks using bitwise AND
        out1 =  _mm256_or_si256(lowMask, highMask);

                inputVector = _mm256_loadu_si256((__m256i*)(input + (i*kernelSize) + 16));
        lowMask = _mm256_cmpgt_epi16(inputVector, lowThresholdVector);
        highMask = _mm256_cmpgt_epi16(inputVector, highThresholdVector);
        out2 =  _mm256_or_si256(lowMask, highMask);

                inputVector = _mm256_loadu_si256((__m256i*)(input + (i*kernelSize) + 32));
        lowMask = _mm256_cmpgt_epi16(inputVector, lowThresholdVector);
        highMask = _mm256_cmpgt_epi16(inputVector, highThresholdVector);
        out3 =  _mm256_or_si256(lowMask, highMask);

                inputVector = _mm256_loadu_si256((__m256i*)(input + (i*kernelSize) + 48));
        lowMask = _mm256_cmpgt_epi16(inputVector, lowThresholdVector);
        highMask = _mm256_cmpgt_epi16(inputVector, highThresholdVector);
        out4 =  _mm256_or_si256(lowMask, highMask);

                inputVector = _mm256_loadu_si256((__m256i*)(input + (i*kernelSize) + 64));
        lowMask = _mm256_cmpgt_epi16(inputVector, lowThresholdVector);
        highMask = _mm256_cmpgt_epi16(inputVector, highThresholdVector);
        out5 =  _mm256_or_si256(lowMask, highMask);

                inputVector = _mm256_loadu_si256((__m256i*)(input + (i*kernelSize) + 80));
        lowMask = _mm256_cmpgt_epi16(inputVector, lowThresholdVector);
        highMask = _mm256_cmpgt_epi16(inputVector, highThresholdVector);
        out6 =  _mm256_or_si256(lowMask, highMask);

                inputVector = _mm256_loadu_si256((__m256i*)(input + (i*kernelSize) + 96));
        lowMask = _mm256_cmpgt_epi16(inputVector, lowThresholdVector);
        highMask = _mm256_cmpgt_epi16(inputVector, highThresholdVector);
        out7 =  _mm256_or_si256(lowMask, highMask);

                inputVector = _mm256_loadu_si256((__m256i*)(input + (i*kernelSize) + 112));
        lowMask = _mm256_cmpgt_epi16(inputVector, lowThresholdVector);
        highMask = _mm256_cmpgt_epi16(inputVector, highThresholdVector);
        out8 =  _mm256_or_si256(lowMask, highMask);

                inputVector = _mm256_loadu_si256((__m256i*)(input + (i*kernelSize) + 128));
        lowMask = _mm256_cmpgt_epi16(inputVector, lowThresholdVector);
        highMask = _mm256_cmpgt_epi16(inputVector, highThresholdVector);
        out9 =  _mm256_or_si256(lowMask, highMask);

                inputVector = _mm256_loadu_si256((__m256i*)(input + (i*kernelSize) + 144));
        lowMask = _mm256_cmpgt_epi16(inputVector, lowThresholdVector);
        highMask = _mm256_cmpgt_epi16(inputVector, highThresholdVector);
        out10 =  _mm256_or_si256(lowMask, highMask);

                inputVector = _mm256_loadu_si256((__m256i*)(input + (i*kernelSize) + 160));
        lowMask = _mm256_cmpgt_epi16(inputVector, lowThresholdVector);
        highMask = _mm256_cmpgt_epi16(inputVector, highThresholdVector);
        out11 = _mm256_or_si256(lowMask, highMask);

        // Store the result
        _mm256_storeu_si256((__m256i*)(output + (i * kernelSize)),out1);
                _mm256_storeu_si256((__m256i*)(output + (i * kernelSize) + 16), out2);
                _mm256_storeu_si256((__m256i*)(output + (i * kernelSize) + 32), out3);
                _mm256_storeu_si256((__m256i*)(output + (i * kernelSize) + 48), out4);
                _mm256_storeu_si256((__m256i*)(output + (i * kernelSize) + 64), out5);
                _mm256_storeu_si256((__m256i*)(output + (i * kernelSize) + 80), out6);
                _mm256_storeu_si256((__m256i*)(output + (i * kernelSize) + 96), out7);
                _mm256_storeu_si256((__m256i*)(output + (i * kernelSize) + 112), out8);
                _mm256_storeu_si256((__m256i*)(output + (i * kernelSize) + 128), out9);
                _mm256_storeu_si256((__m256i*)(output + (i * kernelSize) + 144), out10);
                _mm256_storeu_si256((__m256i*)(output + (i * kernelSize) + 160), out11);

    }

    // Handle the remaining pixels (if any) without SIMD
    for (int i = numIterations * kernelSize; i < imageSize; ++i)
    {
        output[i] = (input[i] > lowThreshold && input[i] > highThreshold) ? 255 : 0;
    }
    return outputImage;
}
```