18-645 Final Design Report

Xiaoying Li
Department of ECE
Carnegie Mellon University
Mountain View, United States
xiaoyin3@andrew.cmu.edu

Guanyang He
Department of ECE
Carnegie Mellon University
San Jose, United States
guanyanh@andrew.cmu.edu

Sabina Chang
Department of ECE
Carnegie Mellon University
Mountain View, United States
enhanc@andrew.cmu.edu

***Abstract*—This document summarizes the updated design for three fast kernels to achieve the CNN algorithm in a single channel image processing task. The three kernels are for the convolution, the max pooling, and the relu layer respectively. SIMD instructions, packing, data padding, and parallelization with OpenMP are the main techniques used to speed up the implementation.**

I. Updated Design

A. Convolution Layer

For the conv layer, we have changed and optimized the kernel design significantly since the midterm report. We have additionally updated the kernel size and improved the data layout by requiring packing and unpacking before and after calling the kernel. To report some important parameters, the input dimension to this layer will be 28×28. The output of this layer will be 27×27. The stride size is 1. The filter dimension is 2×2. And lastly the kernel size will be 7×5. Detailed summaries are provided in the below.

1) Kernel Design: Firstly, during the midterm, we were unable to issue enough FMA instructions in our kernel design, so that the instruction pipelines for each of the two available FMA units on department machine are filled. For our problem scope, we were focusing on single channel image input only. Despite that we had a large enough kernel size so we could identify enough independent operations, there were not enough registers for holding the FMA instruction outputs. This imposes constraints on how many parallel FMA instructions we could issue. Thanks to TA's suggestion, we noticed we were only reusing registers for filters but rarely for inputs. Therefore, we modified to assign only two registers for inputs and reuse them. As such, we have much more available registers for storing FMA outputs. The Fig. 1 shows the traditional way of doing convolution of 2×2 filter with stride 1 and our idea of using SIMD FMA.

As can be seen we could parallelize the independent operations for the multiplication of input elements and filter elements using SIMD FMA. Now in Fig. 2 assume on the left of step 1 and step 2 represent a single channel input matrix containing pixel data with matrix dimension 7×5. The 4 element array on the right of the input matrix in Fig. 2 is a filter of size 2×2 that is stored in row major order. Our kernel in step 1 looks at the first column of vector of length 4 and the first column of filter matrix first ($A1 - J1$) and in step 2 it processes the second column of vector of length 4 ($A2 - J2$),

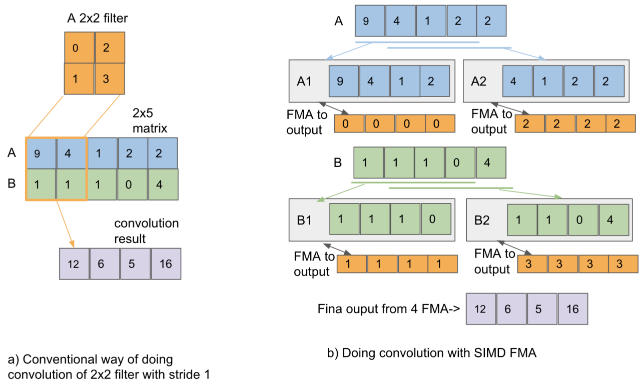


Fig. 1. Perform Convolution using SIMD

with the middle three values overlapping with $A1 - J1$, and the second column of filter matrix.

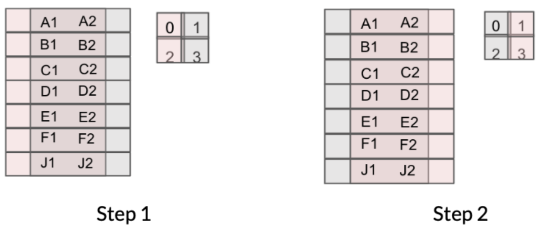


Fig. 2. Conv Layer Inputs

Below are the steps that contain our pseudocode for the kernel design. As can be seen, the step 1 and 2 are able to be filled with 12 FMA instructions each to prevent stall in pipelines and achieve high performance. R1 and R2 are reused filter registers; R5 and R11 are reused input registers; the rest are all dedicated to output registers. Each output register stores the multiplication between the corresponding input elements and a filter element per step; after step 2, each output register hold the results for operations on two of such filter elements.

Step 1	***Step 2***
R1=bc[0], R2=bc[2]	R1=bc[1], R5=ld(B2)
R5=ld(B1), R11=ld(E1)	R2=bc[3]
R16=fma(R5,R2)	R16=fma(R5,R2), R11=ld(E2)
R15=fma(R5,R1)	R15=fma(R5,R1)
R12=fma(R11,R1), R5=ld(C1)	R12=fma(R11,R1), R5=ld(C2)
R13=fma(R11,R2)	R13=fma(R11,R2)
R14=fma(R5,R1), R11=ld(D1)	R14=fma(R5,R1), R11=ld(D2)
R10=fma(R5,R2)	R10=fma(R5,R2)
R3=fma(R11,R1), R5=ld(F1)	R3=fma(R11,R1), R5=ld(F2)
R4=fma(R11,R2)	R4=fma(R11,R2)
R8=fma(R5,R1), R11=ld(J1)	R8=fma(R5,R1), R11=ld(J2)
R6=fma(R5,R2)	R6=fma(R5,R2)
R5=ld(A1)	R5=ld(A2)
R7=fma(R11,R2)	R7=fma(R11,R2)
R9=fma(R5,R1)	R9=fma(R5,R1)

For the step 3 and step 4, we need to sum up the pair of registers, each of which hold the multiplication results between input elements and two of the filter elements. By summing up register values like below, we have eventually R16-R12, plus R8 holding the output of conv layer from the first row to the six row. That is, the output is a 6×4 matrix stored in step 4.

Step 3	***Step 4***
R16 = add(R16,R9)	store(R16, 0)
R15 = add(R15,R10)	store(R15, 4)
R14 = add(R14,R4)	store(R14, 8)
R13 = add(R13,R3)	store(R13, 12)
R12 = add(R12,R6)	store(R12, 16)
R8 = add(R8,R7)	store(R8, 20)

There are some caveats to notice. In the grid shown above, if the drawn kernel is placed in the middle as one of the many such kernels which together make up the overall real input image matrix, then the first row, last row, first column, and last column may be needed for the computation of other kernels on the top, bottom, left, and right of this kernel, in order to have the outputs to be correct. We present more details in the data packing section.

2) Kernel Size: Second, we updated our kernel size to make it 7×5 instead of 6×5 now. With a change in kernel design, we are now able to compute more independent operations with only 16 registers. We decided the kernel size based on that we allocate 2 registers to hold two filter elements and 2 registers to hold two input elements, and assign all the rest to outputs. The reason is the cause of bottleneck during our previous design was that the same output register for storing FMA results are used too frequently, not after 10 cycles (5 latency $\times$ 2 IPC throughput), so that it causes dependency on output registers and stalls in pipelines. Our reasoning this time is to give as many as possible output registers to FMA instructions so that there's no dependency anymore. With this design in mind, we also needed at least two registers for holding filter elements so that enough FMA instructions with the input can be performed to reach 10 independent FMA instructions. Therefore, the register number, latency and throughput of FMA collectively helped us determine the kernel size.

3) Data Packing: The Fig. 3 below helps us better illustrate how we design the packing procedure for the conv layer and the motive behind it. The grid is a 28×28 input matrix that is of the same dimension of the data being used for our baseline [1]. The grey overlapping rectangles placed on the grid are each represented as a kernel. The pink squares are each viewed as a filter of 2×2, which matches the size of the filter in our defined problem. The filter matrix itself will be stored in a row major order, so is each kernel input matrix shown in the grey rectangular below.

Our packing procedure looks at the first column of the grey rectangles and stores each kernel input matrix in row major order, and then it looks at the second column. Starting from the second column and up to all future columns, the packing stores one duplicate column of input elements shown in the overlap at point B in the figure. The rest follows the same, the packing still goes through the second column of kernel input matrix row by row and moves to the next column.

- At point A and point B, the overlaps show that the adjacent kernels require that same overlapped row or column already looked at by another kernel to compute a result. The outputs computed this way will be correct since as shown by the pink filter matrix at row $a1$, $a2$ and column $b1$, $b2$ the outputs concatenated together from results of these adjacent kernels will seamlessly connect and follow the right order. As illustrated above we needed to store overlapping columns at point B since kernel input matrices are went over column by column, but at point A we did not need to store extra one overlapping row of input elements. We could pass that row of input elements when invoking the kernel and don't spend extra memory on that.
- It can be seen at point C, the last grey rectangle goes over the bottom of the input grid, since the kernel dimension could not be divided perfectly. In our packing we added padding to the missing 3×5 number of input elements to complete this kernel by defaulting the element values to be 0 so it doesn't affect the conv layer accuracy. After the kernels are ran, we need to unpack the output matrix too and can ignore the last four rows of outputs for these point C type of kernels since they were used to pad the kernel input. Similarly is the case for the kernel at point D, where we pad one additional column $d2$ of input elements and during unpacking we ignore the last column of the 6×4 kernel output matrix.

Therefore, when invoking the kernels, we write an outer for loop that goes over the seven columns of kernel input matrix and an inner for loop that passes each kernel input matrix in that column row by row to invoke the kernel. After we are done, each kernel will produce a output of 6×4 matrix. The

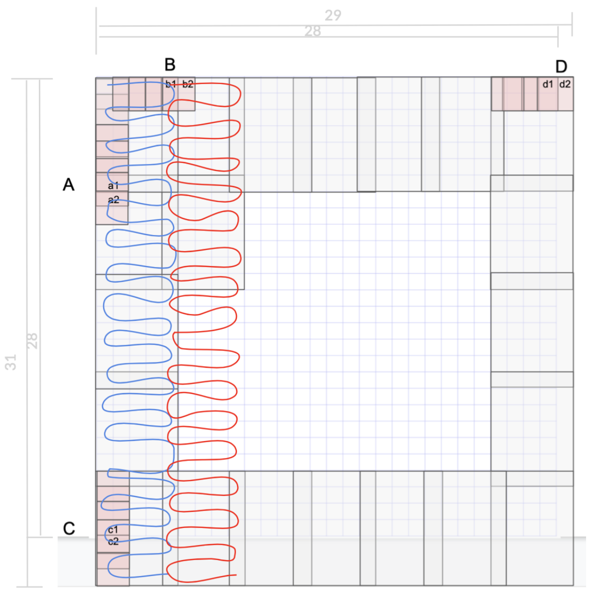


Fig. 3. Conv Layer Packing

total output dimension will be 27×27. We need to perform unpacking on the output matrix. It does the following things:

- Ignore the output elements that came from the padded inputs as explained above.
- Restore the output matrix to row major order, the same data layout as the input matrix. There are seven columns of 27×4 sized column in the output matrix, each of which stored in row major order but overall such columns are stored column by column. We rearrange the layout back to the plain row major order to complete this layer's computation.

B. Max Pooling Layer

The input into this layer is a 27×27 matrix from the convolution layer. The max pooling window is 2×2 with a stride of 2. So in order for the window to fit the input matrix, we drop the last column and the last row and use a 26×26 input matrix.

1) Kernel Design: The max pooling kernel design remains unchanged from our initial design. The basic idea of this kernel is summarized in Fig. 4 by using an example of a 4×4 input matrix. By using all available 16 SIMD registers and filling the pipelines of the chosen SIMD instructions, we can expand to a 12×4 input matrix. The following steps give the pseudo code for the max pooling kernel algorithm.

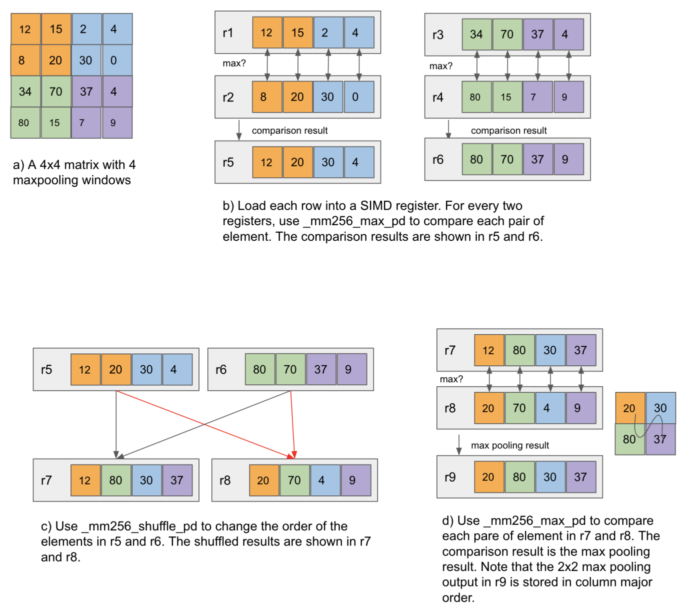
```
Step 1
R1 = load(input)
R2 = load(input+4)
R3 = load(input+8)
R4 = load(input+12)
R5 = load(input+16)
...
R12 = load(input+44)

Step 2
R13 = max(R1,R2)
R14 = max(R3,R4)
R15 = max(R5,R6)
R16 = max(R7,R8)
R1 = max(R9,R10)
R2 = max(R11,R12)

Step 3
R3 = shuffle(R13,R14)
R4 = shuffle(R13,R14)
R5 = shuffle(R15,R16)
R6 = shuffle(R15,R15)
R7 = shuffle(R1,R2)
R8 = shuffle(R1,R2)

Step 4
R9 = max(R3,R4)
R10 = max(R5,R6)
R11 = max(R7,R8)

Step 5
store(R9,output)
store(R10,output+4)
store(R11,output+8)
```



a) A 4x4 matrix with 4 maxpooling windows

b) Load each row into a SIMD register. For every two registers, use _mm256_max_pd to compare each pair of element. The comparison results are shown in r5 and r6.

c) Use _mm256_shuffle_pd to change the order of the elements in r5 and r6. The shuffled results are shown in r7 and r8.

d) Use _mm256_max_pd to compare each pare of element in r7 and r8. The comparison result is the max pooling result. Note that the 2x2 max pooling output in r9 is stored in column major order.

Fig. 4. Perform Max Pooling using SIMD

2) Data Packing: To ensure that the kernel can efficiently read in chunks of 12×4 input data, we change the memory layout of our input data. Fig.5 provides a visualization of the packing algorithm. Following this packing algorithm, the input matrix is converted into 14 12×4 panels, with one 2×2 square left from the bottom right corner of the input. We add the 2×2 square into the pack matrix manually. Lastly, the kernel can process the pack matrix in 14 iterations, where 48 consecutive elements are being read in each iteration.

C. Relu Layer

The input into this layer is a 13×13 matrix from the max pooling layer.

1) Kernel Design: The kernel design for Relu layer remains unchanged from our initial design. Fig.6 visualizes the main idea in the kernel using an example of 4 elements. By using all available 16 SIMD registers and filling the pipelines of chosen SIMD instructions, the Relu kernel can process 32 elements each time. The following steps give the pseudo code for the algorithm.

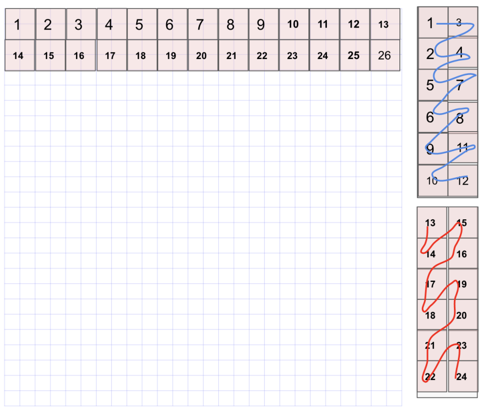


Fig. 5. Max Pooling data packing. (Left) It shows a 26×26 input. Each pink blocks denotes 4 elements covered by a 2×2 max pooling window. (Right) It shows 2 12×4 panels that will be fed into the kernel. In each panel, the pink blocks are stored in row major order (shown in blue line), with their positions differ from that in the input matrix.This panel layout ensures that when the kernel produces column major output (shown in red line), as mentioned in Fig.4 d), the output can be interpreted in fact as row major order.

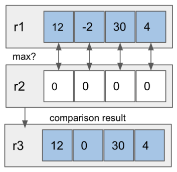


Fig. 6. Perform Relu using SIMD. The main idea is to compare a vector of 4 elements with zeros.

Step 1
R1 = setZero()
R2 = setZero()
R3 = setZero()

Step 2
R4 = load(input)
R5 = load(input+4)...
R16 = load(input+28)

Step 3
R10 = max(R1,R4)
R11 = max(R2,R5)
R12 = max(R3,R6)

Step 4
R13= max(R7,R4),store(R10,out)
R10 = max(R8,R5),store(R11,out+4
R11 = max(R9,R6),store(R12,out+8)
R12 = max(R15,R6),store(R13,out+12)
R13 = max(R16,R6),store(R10,out+16)

Step 5
store(R11,out+20)
store(R12,out+24)
store(R13,out+28)

2) Data Packing: Although no data packing is necessary for the Relu layer, the input matrix needs to be pre-processed. The 13×13 input matrix is not divisible by the kernel size 32. So we just perform Relu on the first 9 elements then pass the remaining 160 elements into the kernel. The kernel runs 5 iterations to finish the whole input.

II. Parallelization

A. Conv Layer

We parallelized the conv layer by running out several experiments using openMP. The result was that using the ”parallel task” option yield pretty good performance. The second best way of parallelization was using the ”parallel for” pragma or the ”parallel region” option. Their performance seems to be quite close. Despite what we experimented in homework 4, during which the other two ways of parallelization outperforms the ”parallel task” in the average count time of triangles, we observed in the conv layer the ”parallel task” option was consistently speeding up the computations more.

One hypothesis was that maybe as task is being pushed by a single thread and becomes ready one by one, there are available threads that pick up the task as they come, so when these threads that have a job to perform begin to compute their task, their task will be of close locality to the tasks issued recently. Whereas for the ”parallel for” pragma, as all threads are assigned a iteration there are all kinds of possible distribution of the iterations, so that it may not help with locality, especially when the input data is large.

Additionally, there are not a large number of tasks to be created so the overhead of creating the tasks by a single thread is not as large as it would have been when the task list is huge.

All in all based on empirical result, we chose to use ”parallel task” for this layer. There are only 7 threads created for the layer since the outer for loop loops across the seven columns of kernel input matrix as explained before. Hence there’s no need to create extra unused thread given they will not receive an iteration.

B. Max Pooling Layer

The max pooling kernel is run 14 times by a for loop. Each iteration is independent and can be run with parallel threads. We use the OpenMP *for* directive with 12 threads to achieve parallellization. The *for* directive is chosen because it is simple to add to existing code, and we do not need to adjust the indexing in the loop. We use the default round robin scheduling of the *for* directive since each thread does the same amount of calculation, and load balancing is not required. The thread number is chosen heuristically. We experimented with 1, 2, 4, 8, 12, 16 threads and observed the cycles taken. Using 16 threads usually gives faster result.

C. Relu Layer

The Relu kernel is run 5 times by a for loop. Each iteration is independent and can be run with parallel threads. The parallellization scheme is also the OpenMP *for* directive with 12 threads. Justification for the choice is same as the max pooling layer.

III. Performance Plots

The following plots summarize our baseline comparison [1] and our implementation performance across different input data sizes. On the horizontal axis, the plotted input data size values are corresponding to the number of independent instructions, computed from the various variable input parameter values in our problem scope.

It can be seen our implementation of a fast kernel for each layer significantly outperforms the baseline implementation. The theoretical peak (in IPC) for each of the three layers as computed during the midterm report are 16, 8, and 8 correspondingly. For the conv layer, our kernel reached $15.0205/16 = 0.9388$ of the theoretical peak; for the max pooling layer, our kernel reached $7.9129/8 = 0.9891$ of the theoretical peak; for the relu layer, our kernel reached $5.7822/8 = 0.7228$ of the theoretical peak.

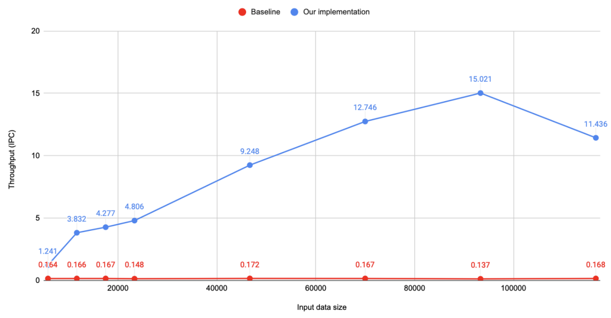


Fig. 7. Conv Layer

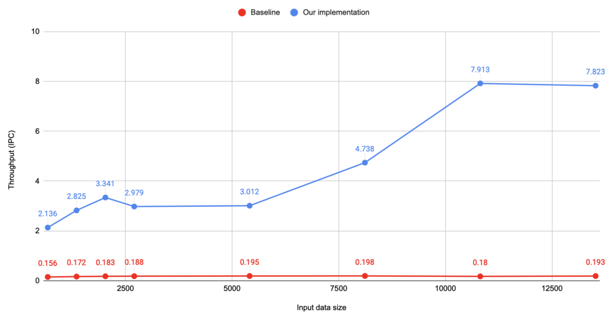


Fig. 8. Max Pooling Layer

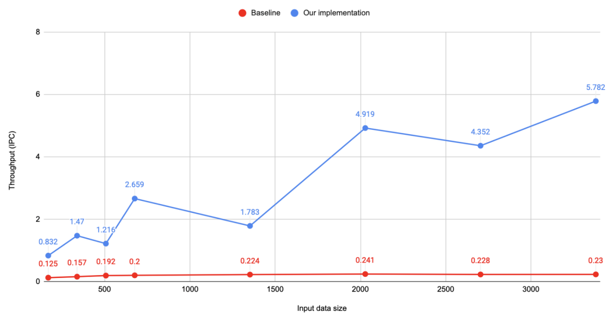


Fig. 9. Relu Layer

IV. Future Directions

- We would have tried multi-channel image input and test run our kernel design on this problem. Our current kernel design could potentially be ran multiple times to serve as a fast implementation of the multi-channel image processing problem. We did not have time to see whether our kernel will perform well in this scenario. But in reality a kernel for multi-channel images can be so much more useful for colorful images.
- For the conv layer design, when performing data packing, we did not use SIMD instructions but simple for loops. This was because the dimension of the kernel size, for instance the width being 5, was not able to support the SIMD vector which computes 4 elements together at a time. Without proper speeding up of the packing procedure, the packing could induce so much cost that the kernel speed up become meaningless.
- For the conv layer kernel, on the other hand, when loading the input data, there are a lot of input elements that are reloaded to be multiplied with different filter elements. We could explore more ways to optimize for the memory access.
- The indexing method in the max pooling packing algorithm only works with specific input dimension. It is limited to converting a 26×26 matrix into 14 12×4 panels. The packing algorithm should have more flexibility. A 2×2 pooling window of stride 2 is a common choice in CNN architecture, so we do not need to worry how to adapt the current packing algorithm to a different panel size. Instead, we need to come up with an indexing method in the packing algorithm that generalizes well to arbitrary input dimensions. The generalized packing algorithm should be able to convert arbitrary input dimension to N 12×4 panels. This way, applying our optimized max pooling kernel on arbitrary input dimension is as simply as calling the kernel N times in a loop.
- Relu layer kernel could be merged into the max pooling layer. Since Relu layer does not need any data packing, we can perform Relu operation immediately after the max pooling input is done.

References

[1] Bölük, Can, and Constantine Shablya. ”Can1357/SIMPLE_CNN: Simple Convolutional Neural Network Library. GitHub,” GitHub, 27 May 2017.
[2] CS231N Convolutional Neural Networks for Visual Recognition, Stanford University, https://cs231n.github.io/convolutional-networks/.
[3] Fog, Agner.”Introduction 4. Instruction Tables-Agner Fog.” Instruction_tables, 7 Aug, 2021, https://www.agner.org/optimize/instruction_tables.pdf.
[4] “Intel® Intrinsics Guide.” Intel, Intel, https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.