Project : Memory Design and Kernel Implementation

Group member:

Zijie Huang: zijieh@andrew.cmu.edu (Andrew ID: zijieh)

Ningyan Zhang: ningyanz@andrew.cmu.edu (Andrew ID: ningyanz)

Yueze Cao: yuezec@andrew.cmu.edu (Andrew ID: yuezec)

1. **Updated Kernel Design and Implementation:**
    a. **Motion Blur (Zijie Huang & Ningyan Zhang):**

In our previous kernel design, we primarily relied on scalar operations and did not fully utilize the available registers. Additionally, we did not adequately address the dependencies between instructions, which could potentially impact performance. In our latest motion kernel design, we utilized **2** SIMD registers for constants, **4** SIMD registers for inputs with some degree of reuse, and **8** SIMD registers for outputs, leaving two registers unused. Notably, the number of outputs in the kernel has been expanded to **40**. More importantly, we have transitioned from using scalar operations to employing SIMD Fused Multiply-Add (**FMA**) instructions.

```
172     for (int j = 0; j < numPixels; j += 128) {
173
174         ymm0 = _mm256_set_pd(1/2, 1/2, 1/2, 1/2);
175         ymm1 = _mm256_set_pd(1/6, 1/6, 1/6, 1/6);
176
177         ymm2 = _mm256_loadu_pd(&input[j]);
178         ymm3 = _mm256_loadu_pd(&input[j+4]);
179         ymm4 = _mm256_loadu_pd(&input[j+8]);
180         ymm5 = _mm256_loadu_pd(&input[j+12]);
181
182         ymm6 = _mm256_fmadd_pd(ymm2, ymm0, ymm6);
183         ymm7 = _mm256_fmadd_pd(ymm3, ymm0, ymm7);
184         ymm8 = _mm256_fmadd_pd(ymm4, ymm0, ymm8);
185         ymm9 = _mm256_fmadd_pd(ymm5, ymm0, ymm9);
186
187         ymm2 = _mm256_loadu_pd(&input[j+16]);
188
189         ymm6 = _mm256_fmadd_pd(ymm3, ymm1, ymm6);
190         ymm7 = _mm256_fmadd_pd(ymm4, ymm1, ymm7);
191         ymm8 = _mm256_fmadd_pd(ymm5, ymm1, ymm8);
192         ymm9 = _mm256_fmadd_pd(ymm2, ymm1, ymm9);
193
194         ymm3 = _mm256_loadu_pd(&input[j+20]);
195
196         ymm6 = _mm256_fmadd_pd(ymm4, ymm1, ymm6);
197         ymm7 = _mm256_fmadd_pd(ymm5, ymm1, ymm7);
198         ymm8 = _mm256_fmadd_pd(ymm2, ymm1, ymm8);
199         ymm9 = _mm256_fmadd_pd(ymm3, ymm1, ymm9);
200
201         ymm4 = _mm256_loadu_pd(&input[j+24]);
202
203         ymm6 = _mm256_fmadd_pd(ymm5, ymm1, ymm6);
204         ymm7 = _mm256_fmadd_pd(ymm2, ymm1, ymm7);
205         ymm8 = _mm256_fmadd_pd(ymm3, ymm1, ymm8);
206         ymm9 = _mm256_fmadd_pd(ymm4, ymm1, ymm9);
207
```

Figure 1: The kernel code of Motion Blur

```
199         ymm9 = _mm256_fmadd_pd(ymm3, ymm1, ymm9);
200
201         ymm4 = _mm256_loadu_pd(&input[j+24]);
202
203         ymm6 = _mm256_fmadd_pd(ymm5, ymm1, ymm6);
204         ymm7 = _mm256_fmadd_pd(ymm2, ymm1, ymm7);
205         ymm8 = _mm256_fmadd_pd(ymm3, ymm1, ymm8);
206         ymm9 = _mm256_fmadd_pd(ymm4, ymm1, ymm9);
207
208         ymm2 = _mm256_loadu_pd(&input[j+64]);
209         ymm3 = _mm256_loadu_pd(&input[j+68]);
210         ymm4 = _mm256_loadu_pd(&input[j+72]);
211         ymm5 = _mm256_loadu_pd(&input[j+76]);
212
213         ymm10 = _mm256_fmadd_pd(ymm2, ymm0, ymm10);
214         ymm11 = _mm256_fmadd_pd(ymm3, ymm0, ymm11);
215         ymm12 = _mm256_fmadd_pd(ymm4, ymm0, ymm12);
216         ymm13 = _mm256_fmadd_pd(ymm5, ymm0, ymm13);
217
218         ymm2 = _mm256_loadu_pd(&input[j+80]);
219
220         ymm10 = _mm256_fmadd_pd(ymm3, ymm1, ymm10);
221         ymm11 = _mm256_fmadd_pd(ymm4, ymm1, ymm11);
222         ymm12 = _mm256_fmadd_pd(ymm5, ymm1, ymm12);
223         ymm13 = _mm256_fmadd_pd(ymm2, ymm1, ymm13);
224
225         ymm3 = _mm256_loadu_pd(&input[j+84]);
226
227         ymm10 = _mm256_fmadd_pd(ymm4, ymm1, ymm10);
228         ymm11 = _mm256_fmadd_pd(ymm5, ymm1, ymm11);
229         ymm12 = _mm256_fmadd_pd(ymm2, ymm1, ymm12);
230         ymm13 = _mm256_fmadd_pd(ymm3, ymm1, ymm13);
231
232         ymm4 = _mm256_loadu_pd(&input[j+88]);
233
234         ymm10 = _mm256_fmadd_pd(ymm5, ymm1, ymm10);
235         ymm11 = _mm256_fmadd_pd(ymm2, ymm1, ymm11);
236         ymm12 = _mm256_fmadd_pd(ymm3, ymm1, ymm12);
237         ymm13 = _mm256_fmadd_pd(ymm4, ymm1, ymm13);
```

Figure 2: The kernel code of Motion Blur cont.

b. Edge (Zijie Huang & Yueze Cao):

In our last report, we had not yet finalized the design of the kernel of Edge. Similar to Motion Blur we are incorporating Fused Multiply-Add (FMA) operations in this kernel. In terms of register allocation, We are utilizing **2** SIMD registers for constants, **4** SIMD registers for inputs, and **8** SIMD registers for output. Finally, the size of the kernel is also **40**.

```
15  // Pseudo(kernel)
16  // ymm0, ymm1 for constants
17  // ymm2 ~ ymm5 for inputs
18  // ymm7 ~ ymm14 for outputs
19  ymm0 = broadcast - 1;
20  ymm1 = broadcast 8;
21
22  ymm2 = load_pd input0;
23  ymm3 = load_pd input1;
24  ymm4 = load_pd input2;
25  ymm5 = load_pd input3;
26
27  ymm7 = FMA(ymm0, ymm2, ymm7);       // ADD upper left (first output row)
28  ymm8 = FMA(ymm0, ymm3, ymm8);
29  ymm9 = FMA(ymm0, ymm4, ymm9);
30  ymm10 = FMA(ymm0, ymm5, ymm10);
31
32  ymm2 = load_pd input4;
33
34  ymm7 = FMA(ymm0, ymm3, ymm7);       // ADD directly above (first output row)
35  ymm8 = FMA(ymm0, ymm4, ymm8);
36  ymm9 = FMA(ymm0, ymm5, ymm9);
37  ymm10 = FMA(ymm0, ymm2, ymm10);
38
39  ymm3 = load_pd input5;
40
41  ymm7 = FMA(ymm0, ymm4, ymm7);       // ADD upper right (first output row)
42  ymm8 = FMA(ymm0, ymm5, ymm8);
43  ymm9 = FMA(ymm0, ymm2, ymm9);
44  ymm10 = FMA(ymm0, ymm3, ymm10);
45
46  ymm2 = load_pd input6;
47  ymm3 = load_pd input7;
48  ymm4 = load_pd input8;
49  ymm5 = load_pd input9;
```

Figure 3: The kernel pseudo code of Motion Blur

```
49    ymm5 = load_pd input9;
50
51    ymm7 = FMA(ymm0, ymm2, ymm7);       // ADD left (first output row)
52    ymm8 = FMA(ymm0, ymm3, ymm8);
53    ymm9 = FMA(ymm0, ymm4, ymm9);
54    ymm10 = FMA(ymm0, ymm5, ymm10);
55
56    ymm11 = FMA(ymm0, ymm2, ymm11);     // ADD upper left (second output row)
57    ymm12 = FMA(ymm0, ymm3, ymm12);
58    ymm13 = FMA(ymm0, ymm4, ymm13);
59    ymm14 = FMA(ymm0, ymm5, ymm14);
60
61    ymm2 = load_pd input10;
62
63    ymm7 = FMA(ymm1, ymm3, ymm7);       // MUL 8 (first output row)
64    ymm8 = FMA(ymm1, ymm4, ymm8);
65    ymm9 = FMA(ymm1, ymm5, ymm9);
66    ymm10 = FMA(ymm1, ymm2, ymm10);
67
68    ymm11 = FMA(ymm0, ymm3, ymm11);     // ADD directly above (second output row)
69    ymm12 = FMA(ymm0, ymm4, ymm12);
70    ymm13 = FMA(ymm0, ymm5, ymm13);
71    ymm14 = FMA(ymm0, ymm2, ymm14);
72
73    ymm3 = load_pd input11;
74
75    ymm7 = FMA(ymm1, ymm4, ymm7);       // ADD right (first output row)
76    ymm8 = FMA(ymm1, ymm5, ymm8);
77    ymm9 = FMA(ymm1, ymm2, ymm9);
78    ymm10 = FMA(ymm1, ymm3, ymm10);
79
80    ymm11 = FMA(ymm0, ymm4, ymm11);     // ADD upper right (second output row)
81    ymm12 = FMA(ymm0, ymm5, ymm12);
82    ymm13 = FMA(ymm0, ymm2, ymm13);
83    ymm14 = FMA(ymm0, ymm3, ymm14);
84
85    ymm2 = load_pd input12;
86    ymm3 = load_pd input13;
87    ymm4 = load_pd input14;
88    ymm5 = load_pd input15;
89
```

Figure 4: The kernel pseudo code of Motion Blur cont.

```
89
90    ymm7 = FMA(ymm0, ymm2, ymm7);        // ADD lower left (first output row)
91    ymm8 = FMA(ymm0, ymm3, ymm8);
92    ymm9 = FMA(ymm0, ymm4, ymm9);
93    ymm10 = FMA(ymm0, ymm5, ymm10);
94
95    ymm11 = FMA(ymm0, ymm2, ymm11);      // ADD left (second output row)
96    ymm12 = FMA(ymm0, ymm3, ymm12);
97    ymm13 = FMA(ymm0, ymm4, ymm13);
98    ymm14 = FMA(ymm0, ymm5, ymm14);
99
100   ymm2 = load_pd input16;
101
102   ymm7 = FMA(ymm0, ymm3, ymm7);        // ADD directly below (first output row)
103   ymm8 = FMA(ymm0, ymm4, ymm8);
104   ymm9 = FMA(ymm0, ymm5, ymm9);
105   ymm10 = FMA(ymm0, ymm2, ymm10);
106
107   ymm11 = FMA(ymm1, ymm3, ymm11);      // MUL 8 (second output row)
108   ymm12 = FMA(ymm1, ymm4, ymm12);
109   ymm13 = FMA(ymm1, ymm5, ymm13);
110   ymm14 = FMA(ymm1, ymm2, ymm14);
111
112   ymm3 = load_pd input17;
113
114   ymm7 = FMA(ymm0, ymm4, ymm7);        // ADD lower right (first output row)
115   ymm8 = FMA(ymm0, ymm5, ymm8);
116   ymm9 = FMA(ymm0, ymm2, ymm9);
117   ymm10 = FMA(ymm0, ymm3, ymm10);
118
119   ymm11 = FMA(ymm0, ymm4, ymm11);      // ADD right (second output row)
120   ymm12 = FMA(ymm0, ymm5, ymm12);
121   ymm13 = FMA(ymm0, ymm2, ymm13);
122   ymm14 = FMA(ymm0, ymm3, ymm14);
123
124   ymm2 = load_pd input18;
125   ymm3 = load_pd input19;
126   ymm4 = load_pd input20;
127   ymm5 = load_pd input21;
128
129   ymm11 = FMA(ymm0, ymm2, ymm11);      // ADD lower left (second output row)
130   ymm12 = FMA(ymm0, ymm3, ymm12);
131   ymm13 = FMA(ymm0, ymm4, ymm13);
132   ymm14 = FMA(ymm0, ymm5, ymm14);
```

Figure 5: The kernel pseudo code of Motion Blur cont.

```
ymm14 = FMA(ymm0, ymm5, ymm14);

ymm2 = load_pd input22;

ymm11 = FMA(ymm0, ymm3, ymm11);     // ADD directly below (second output row)
ymm12 = FMA(ymm0, ymm4, ymm12);
ymm13 = FMA(ymm0, ymm5, ymm13);
ymm14 = FMA(ymm0, ymm2, ymm14);

ymm3 = load_pd input23;

ymm11 = FMA(ymm0, ymm4, ymm11);     // ADD lower right (second output row)
ymm12 = FMA(ymm0, ymm5, ymm12);
ymm13 = FMA(ymm0, ymm2, ymm13);
ymm14 = FMA(ymm0, ymm3, ymm14);
```

Figure 6: The kernel pseudo code of Motion Blur cont.

c. Rotate and Zoom (Yueze Cao & Ningyan Zhang):

In our previous design we used too many registers for storing constants when computing. However, considering the rotate and zoom algorithm is not dealing with the content of the matrix, we only have to manipulate the index so we store the content of the new index into the old index's content.

```
for(y=0; y<HEIGHT; y++)
{
    for(x=0; x<WIDTH; x+=24)
    {
        ymm0 = _mm256_setr_pd(M[0][0],M[0][0],M[1][0],M[1][0]);
        ymm1 = _mm256_setr_pd(-CenterX*M[0][0] + (y-CenterY)*M[0][1] + CenterX, -CenterX*M[0][0] + (y-CenterY)*M[0][1] + CenterX,
                              -CenterX*M[1][0] + (y-CenterY)*M[1][1] + CenterX, -CenterX*M[1][0] + (y-CenterY)*M[1][1] + CenterX);

        ymm2 = _mm256_setr_pd(x+0,x+1,y+0,y+1);
        ymm3 = _mm256_setr_pd(x+2,x+3,y+2,y+3);

        ymm4 = FMA(ymm3,ymm0,ymm1);
        ymm5 = FMA(ymm4,ymm0,ymm1);

        ymm2 = _mm256_setr_pd(x+4,x+5,y+4,y+5);
        ymm3 = _mm256_setr_pd(x+6,x+7,y+6,y+7);

        ymm6 = FMA(ymm3,ymm0,ymm1);
        ymm7 = FMA(ymm4,ymm0,ymm1);

        ymm2 = _mm256_setr_pd(x+8,x+9,y+8,y+9);
        ymm3 = _mm256_setr_pd(x+10,x+11,y+10,y+11);

        ymm8 = FMA(ymm3,ymm0,ymm1);
        ymm9 = FMA(ymm4,ymm0,ymm1);I

        ymm2 = _mm256_setr_pd(x+12,x+13,y+12,y+13);
        ymm3 = _mm256_setr_pd(x+14,x+15,y+14,y+15);

        ymm10 = FMA(ymm3,ymm0,ymm1);
        ymm11 = FMA(ymm4,ymm0,ymm1);

        ymm2 = _mm256_setr_pd(x+16,x+17,y+16,y+17);
        ymm3 = _mm256_setr_pd(x+18,x+19,y+18,y+19);

        ymm12 = FMA(ymm3,ymm0,ymm1);
        ymm13 = FMA(ymm4,ymm0,ymm1);

        ymm2 = _mm256_setr_pd(x+20,x+21,y+20,y+21);
        ymm3 = _mm256_setr_pd(x+22,x+23,y+22,y+23);

        ymm14 = FMA(ymm3,ymm0,ymm1);
        ymm15 = FMA(ymm4,ymm0,ymm1);
```

Figure 7: Kernel design for Rotate & Zoom (FMA)

Now, our design has **12** SIMD registers holding outputs, **2** SIMD registers holding input, and **2** SIMD registers holding constants. To compute the new coordinate index, the input registers will be reused all the time and followed by FMA instructions to get the new value of x, y. However, in order to return a valid coordinate since the algorithm may compute the index out of the range **(0, WIDTH)**. We have to use the SIMD compare and add to set the content of old x, y to be **zero** if new x, y are out of range. As shown in the figure, all the outputs will be compared and reset with the WIDTH, then again with 0.

```
55          ymm0 = _mm256_set1_pd(WIDTH);
56
57          ymm1 = _mm256_cmp_pd(ymm4, ymm0, _CMP_LT_OQ);
58          ymm2 = _mm256_cmp_pd(ymm5, ymm0, _CMP_LT_OQ);
59          ymm3 = _mm256_cmp_pd(ymm6, ymm0, _CMP_LT_OQ);
60          ymm4 = _mm256_and_si256(ymm1,ymm4);
61          ymm5 = _mm256_and_si256(ymm2,ymm5);
62          ymm6 = _mm256_and_si256(ymm3,ymm6);
63
64          ymm1 = _mm256_cmp_pd(ymm7, ymm0, _CMP_LT_OQ);
65          ymm2 = _mm256_cmp_pd(ymm8, ymm0, _CMP_LT_OQ);
66          ymm3 = _mm256_cmp_pd(ymm9, ymm0, _CMP_LT_OQ);
67          ymm7 = _mm256_and_si256(ymm1,ymm7);
68          ymm8 = _mm256_and_si256(ymm2,ymm8);
69          ymm9 = _mm256_and_si256(ymm3,ymm9);
70
71          ymm1 = _mm256_cmp_pd(ymm10, ymm0, _CMP_LT_OQ);
72          ymm2 = _mm256_cmp_pd(ymm11, ymm0, _CMP_LT_OQ);
73          ymm3 = _mm256_cmp_pd(ymm12, ymm0, _CMP_LT_OQ);
74          ymm10 = _mm256_and_si256(ymm1,ymm10);
75          ymm11 = _mm256_and_si256(ymm2,ymm11);
76          ymm12 = _mm256_and_si256(ymm3,ymm12);
77
78          ymm1 = _mm256_cmp_pd(ymm13, ymm0, _CMP_LT_OQ);
79          ymm2 = _mm256_cmp_pd(ymm14, ymm0, _CMP_LT_OQ);
80          ymm3 = _mm256_cmp_pd(ymm15, ymm0, _CMP_LT_OQ);
81          ymm13 = _mm256_and_si256(ymm1,ymm13);
82          ymm14 = _mm256_and_si256(ymm2,ymm14);
83          ymm15 = _mm256_and_si256(ymm3,ymm15);
84
85          ymm0 = _mm256_set1_pd(0);
```

Figure 8: Kernel design for Rotate & Zoom (CMP & AND)

Since each time 12 SIMD outputs will be computed, and each output contains two new coordinates **(x1, y1, x2, y2)**, we need to store the outputs into a

temporary array and using the index reference to get the pixel value from that new coordinate then store the value to the old index's content.

```
double result[48];
_mm256_storeu_pd(result, ymm4);
_mm256_storeu_pd(result + 4, ymm5);
_mm256_storeu_pd(result + 8, ymm6);
_mm256_storeu_pd(result + 12, ymm7);
_mm256_storeu_pd(result + 16, ymm8);
_mm256_storeu_pd(result + 20, ymm9);
_mm256_storeu_pd(result + 24, ymm10);
_mm256_storeu_pd(result + 28, ymm11);
_mm256_storeu_pd(result + 32, ymm12);
_mm256_storeu_pd(result + 36, ymm13);
mm256_storeu_pd(result + 40, ymm14);
mm256_storeu_pd(result + 44, ymm15);


ymm4 = _mm256_setr_pd(GetPixelR(image, result[0], result[2]),GetPixelR(image, result[1], result[3]), GetPixelR(image, result[4], result[6]),GetPixelR(image, result[5], resul
ymm5 = _mm256_setr_pd(GetPixelR(image, result[8], result[10]),GetPixelR(image, result[9], result[11]), GetPixelR(image, result[12], result[14]),GetPixelR(image, result[13],
ymm6 = _mm256_setr_pd(GetPixelR(image, result[16], result[18]),GetPixelR(image, result[17], result[19]), GetPixelR(image, result[20], result[22]),GetPixelR(image, result[21]
ymm7 = _mm256_setr_pd(GetPixelR(image, result[24], result[26]),GetPixelR(image, result[25], result[27]), GetPixelR(image, result[28], result[30]),GetPixelR(image, result[29]
ymm8 = _mm256_setr_pd(GetPixelR(image, result[32], result[34]),GetPixelR(image, result[33], result[35]), GetPixelR(image, result[36], result[38]),GetPixelR(image, result[37]
ymm9 = _mm256_setr_pd(GetPixelR(image, result[40], result[42]),GetPixelR(image, result[41], result[43]), GetPixelR(image, result[42], result[46]),GetPixelR(image, result[45]
_mm256_storeu_pd(&RotatedR[x][y],ymm4);
_mm256_storeu_pd(&RotatedR[x + 4][y],ymm5);
_mm256_storeu_pd(&RotatedR[x + 8][y],ymm6);
_mm256_storeu_pd(&RotatedR[x + 12][y],ymm7);
_mm256_storeu_pd(&RotatedR[x + 16][y],ymm8);
_mm256_storeu_pd(&RotatedR[x + 20][y],ymm9);
```

Figure 9: Kernel design for Rotate & Zoom (Store)

2. **Memory Hierarchy Design:**
    a. **Motion Blur:**

    The algorithm for Motion Blur involves using half of the RGB values of a pixel, combined with half of the average of the RGB values from the next three pixels to calculate the new RGB value for a pixel. Simply put, it can be expressed as: r0' = (r0 / 2) + ((r1 + r2 + r3) / 3) / 2. In this context, utilizing the original pixel arrangement poses a challenge for SIMD FMA operations. Therefore, it becomes necessary to preprocess and rearrange the original image pixels to create a new intermediate layout. Our approach splits 4 adjacent pixels across 4 SIMD registers, rather than placing these four consecutive pixels within a single SIMD. This preprocessing technique will result in a matrix that is more conducive to SIMD operations, facilitating an efficient computational process for the Motion Blur algorithm.

    b. **Edge:** The Edge algorithm also necessitates computations involving the RGB values of surrounding pixels. Simplified, the formula can be

articulated as: r4' = 8 * r4 - r0 - r1 - r2 - r3 - r5 - r6 - r7 - r8. Similarly, we can apply the preprocessing technique used in the Motion Blur algorithm. This involves distributing the pixels to create a new intermediate layout. By scattering the pixel points in this manner, we can leverage the pixels surrounding the red box (**Figure 10**) to ultimately compute the value of the pixel within the red box.

```
/**
 * @brief This function pre-processes the image for motion blur by shuffling and permuting the pixels.
 * @param input The image before pre-processing.
 * @param output The image after pre-processing.
 * @param numPixels The number of pixels of the input the image.
 */
void pre_process_blur(double* input, double* output, int numPixels) {
    __m256d ymm0, ymm1, ymm2, ymm3;
    __m256d ymm4, ymm5, ymm6, ymm7;
    __m256d ymm8, ymm9, ymm10, ymm11;

    for (int i = 0; i < (4 * (1 + (numPixels - 16) / 4)); i += 4) {
        ymm0 = _mm256_setzero_pd();  ymm1 = _mm256_setzero_pd();
        ymm2 = _mm256_setzero_pd();  ymm3 = _mm256_setzero_pd();
        ymm4 = _mm256_setzero_pd();  ymm5 = _mm256_setzero_pd();
        ymm6 = _mm256_setzero_pd();  ymm7 = _mm256_setzero_pd();
        ymm8 = _mm256_setzero_pd();  ymm9 = _mm256_setzero_pd();
        ymm10 = _mm256_setzero_pd();  ymm11 = _mm256_setzero_pd();

        ymm0 = _mm256_loadu_pd(&input[i]);
        ymm1 = _mm256_loadu_pd(&input[i+4]);
        ymm2 = _mm256_loadu_pd(&input[i+8]);
        ymm3 = _mm256_loadu_pd(&input[i+12]);

        ymm4 = _mm256_permute2f128_pd(ymm0, ymm2, 0x20);
        ymm5 = _mm256_permute2f128_pd(ymm1, ymm3, 0x20);
        ymm6 = _mm256_permute2f128_pd(ymm0, ymm2, 0x31);
        ymm7 = _mm256_permute2f128_pd(ymm1, ymm3, 0x31);
        ymm8 = _mm256_shuffle_pd(ymm4, ymm5, 0x0);
        ymm9 = _mm256_shuffle_pd(ymm4, ymm5, 0xf);
        ymm10 = _mm256_shuffle_pd(ymm6, ymm7, 0x0);
        ymm11 = _mm256_shuffle_pd(ymm6, ymm7, 0xf);

        _mm256_storeu_pd(&output[i*4], ymm8);
        _mm256_storeu_pd(&output[i*4+4], ymm9);
        _mm256_storeu_pd(&output[i*4+8], ymm10);
        _mm256_storeu_pd(&output[i*4+12], ymm11);
    }
}
```

Figure 10: The code of Intermediate layout

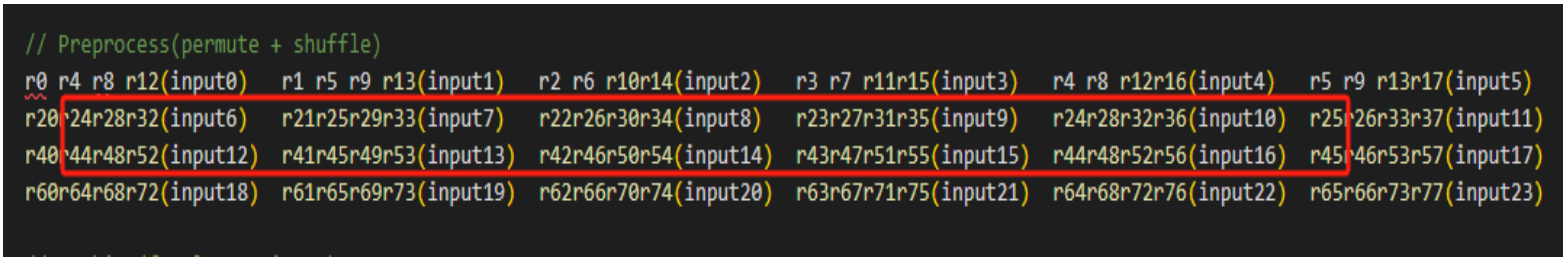

Figure 11: The intermediate layout of Edge

c. **Rotate and Zoom:**

Unlike Edge and Blur, rotate and zoom only need to deal with the index itself with some constant each time. Between each element, all the operations are independent of each other and we don't need to change the content. We simply want to speed up the process of multiplication and comparison. The input is only the index.

3. **Performance Plots:**

a. **Motion Blur**

Bottle neck:

The bottleneck of motion blur is FMA instruction, which should theoretically have a peak of 16. However, the performance is around **11.** We believe this issue is linked to the underutilization of registers, as there are two registers that remain unused within the kernel. This suggests an opportunity for optimization by fully leveraging all available registers, potentially enhancing the kernel's performance.

Performance Plot:

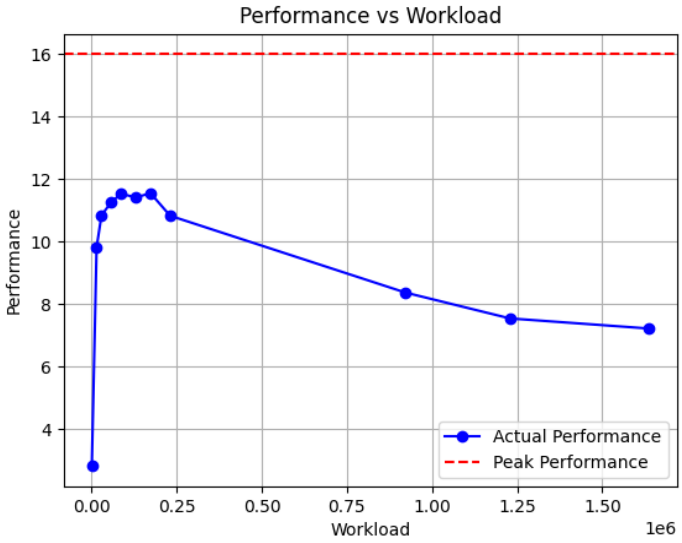

Figure 12: Performance plot for Motion Blur

b. Rotate and Zoom

Bottleneck:

We consider the SIMD FMA operation to be the bottleneck, whose theoretical peak is 16. However, as shown in the figure, the peak performance is only 10. In our kernel design, SIMD CMP, ADD, and SET all become a major issue to let the performance go down. We are still working on the kernel design for this one to try to reduce and avoid SIMD computations other than FMA.

Performance Plot:

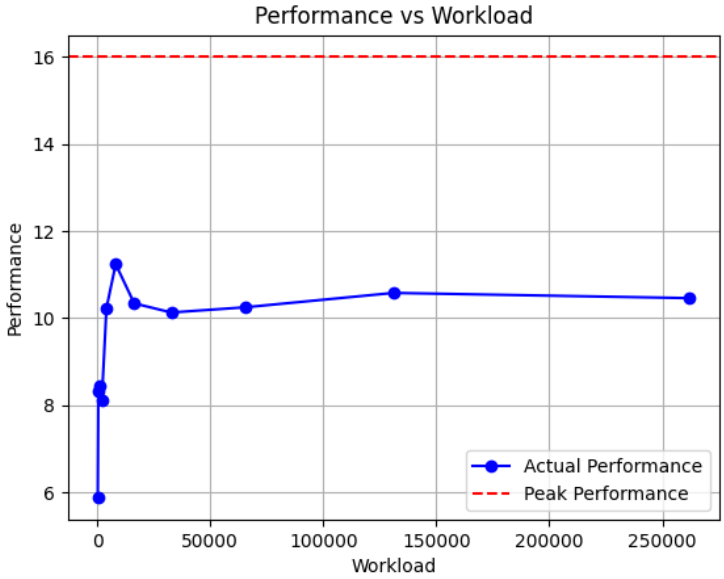

Figure 13: Performance plot for Rotate & Zoom