南開大學
Nankai University

计算机学院

并行程序设计报告

SIMD 编程

姓名：章壹程

学号：2313469

专业：计算机科学与技术

2025 年 4 月 25 日

目录

1 前言

1.1 实验环境

- **CPU：** Kunpeng-920 aarch64 64-bit
- **操作系统：**Linux master_ubss1 5.10.0-235.0.0.134.oe2203sp4.aarch64 #1 SMP Wed Nov 6 13:38:26 CST 2024 aarch64 aarch64 aarch64 GNU/Linux
- **编译器：** 10.3.1 (GCC) aarch64-linux-gnu
- **perf 版本：** 5.10.0-254.0.0.158.oe2203sp4.aarch64

1.2 开源代码

所有代码均可在https://github.com/u2003yuge/parallel-programming上下载，包括但不限于实验所需代码、可视化代码、实验图像。

2 RoI Pooling 算法

2.1 算法简介

ROI pooling 是对（Region of Interest）进行 Pooling 操作，广泛应用在物体检测的研究领域。其目的是对输入 feature map 中的不同大小的 ROI 利用最大池化方法获得固定大小的输出 feature map。

2.2 算法设计

2.2.1 朴素算法

假设 input 的 shape 为 $N \times H \times W \times C$，roi 的 shape 为 $NUM\_ROIS \times 5$，其第二个维度五个值分别该感兴趣区域所在的图片 N_i，以及该区域的左上角和右下角的坐标 x_1, y_1, x_2, y_2，output 的 shape 为 $NUM\_ROIS \times pooled\_height \times pooled\_width \times C$，其中 $pooled\_height$ 和 $pooled\_width$ 表示经过最大池池化后获得的固定大小的 feature map 的长和宽，scale 为缩放因子。

朴素算法为依次枚举每个 NOI，再枚举 $pooler\_height$ 和 $pooler\_width$，计算得到其对应的 input 块（该块各通道分别进行最大池化后恰分别为输出中的一个值），再枚举该块的每一个构成像素向量，对其通道分别求 max 存到 output 中。

时间复杂度为 $O(\sum_{i=1}^{NUM\_ROIS} scale^2 * (roi_{i4} - roi_{i1}) * (roi_{i3} - roi_{i2}) * C)$，额外占用的空间复杂度为 $O(C)$。时间复杂度是仅与 ROI、scale 和 C 相关的，保证了运算次数分摊到参与池化的每个值为常数级别，因此不具备从算法上优化的可能，只能考虑从体系架构与并行角度进行优化。

2.2.2 体系结构优化

循环展开

循环展开和 neon 本质相同，做了简单的尝试与 neon 比较效率。

cache 优化

我们可以从以下角度解释为什么不适合对 RoIPooling 进行 cache 优化：

1. **ROI 本身不连续**。ROI 是在原图中框选出来一个矩形感兴趣区域，该区域对应的 input 在原图中本身就可能不连续，因此做 cache 优化意义不大。

2. **为实现 cache 优化会增加大量额外计算与读写开销**。cache 优化的可行方案为优先枚举 ROI 框选区域的行与列，而 ROI 是按块计算的，每一行也会被分成若干块。因此我们需要频繁的比较是否切换至下一块，而比较操作会严重影响流水线导致效率降低；与此同时每一行切的每一块的临时结果我们都需要存储下来，为了不增大额外占用的空间我们需要直接访问 output 进行读写操作，而 output 可能很庞大无法全部放入缓存，还需要系统自动调整，也是不如 $O(C)$ 更可能直接放入缓存中。

3. **考虑到 ROIPooling 后续的使用场景，使用 cache 优化无意义**。考虑到算子最终用途，*input* 和 *output* 得要我们手动从内存提取至 L2 缓存中，那么利用系统本身的 cache 预测无意义。

2.2.3 neon 优化

neon 优化非常简单且显然，我们利用长度为 C 的 MAXC 向量与所选块内的所有长度为 C 的向量对位比较求最大值。这里的对位比较就可以使用 neon 进行 SIMD 优化。

2.3 编程实现

neon 优化算法

```
#define MSMIN(x, y) ((x) < (y) ? (x) : (y))
#define MSMAX(x, y) ((x) > (y) ? (x) : (y))
int ROIPooling(int input_n, int input_h, int input_w, int input_c, int num_rois,
    float scale, int pooled_height, int pooled_width, const float *in_ptr, float
    *out_ptr, const float *roi, float *max_c, int tid, int thread_num) {
    if (thread_num == 0) {
        return -1;
    }

    int in_strides_2 = input_c;
    int in_strides_1 = in_strides_2 * input_w;
    int in_strides_0 = in_strides_1 * input_h;

    int out_strides_2 = input_c;
    int out_strides_1 = pooled_height * out_strides_2;
    int out_strides_0 = pooled_width * out_strides_1;

    int units = (num_rois + thread_num - 1) / thread_num;
    int roi_st = tid * units;
    int roi_end = MSMIN(num_rois, roi_st + units);
    if (roi_st >= num_rois) {
        return 0;
    }
    int batch_size = input_n;
    int height_ = input_h;
    int width_ = input_w;
```

```
25      int channels_ = input_c;
26
27      const int roi_stride = 5;
28      int roi_ind_st = roi_st * roi_stride;
29      for (int i = roi_st; i < roi_end; ++i) {
30          int roi_batch_ind = (int)roi[roi_ind_st];  // batch_index
31          if (roi_batch_ind >= batch_size) {
32              return -1;
33          }
34          int roi_start_h = (int)round(roi[roi_ind_st + 1] * scale);  // top-left x1
35          int roi_start_w = (int)round(roi[roi_ind_st + 2] * scale);  // top-left y1
36          int roi_end_h = (int)round(roi[roi_ind_st + 3] * scale);    // bottom-right x2
37          int roi_end_w = (int)round(roi[roi_ind_st + 4] * scale);    // bottom-fight y2
38
39          int roi_height = MSMAX(roi_end_h - roi_start_h + 1, 1);
40          int roi_width = MSMAX(roi_end_w - roi_start_w + 1, 1);
41
42          float bin_size_h = (float)roi_height / (float)pooled_height;
43          float bin_size_w = (float)roi_width / (float)pooled_width;
44          const float *batch_data = in_ptr + in_strides_0 * roi_batch_ind;
45
46          for (int ph = 0; ph < pooled_height; ++ph) {
47              for (int pw = 0; pw < pooled_width; ++pw) {
48                  int hstart = (int)floor(ph * bin_size_h);      // block xi_1
49                  int wstart = (int)floor(pw * bin_size_w);      // block yi_1
50                  int hend = (int)ceil((ph + 1) * bin_size_h);   // block xi_2
51                  int wend = (int)ceil((pw + 1) * bin_size_w);   // block yi_2
52
53                  hstart = MSMIN(MSMAX(hstart + roi_start_h, 0), height_);
54                  hend = MSMIN(MSMAX(hend + roi_start_h, 0), height_);
55                  wstart = MSMIN(MSMAX(wstart + roi_start_w, 0), width_);
56                  wend = MSMIN(MSMAX(wend + roi_start_w, 0), width_);
57
58                  bool is_empty = (hend <= hstart) || (wend <= wstart);
59                  for (int j = 0; j < channels_; ++j) {
60                      max_c[j] = is_empty ? 0 : -FLT_MAX;
61                  }
62                  int pooled_index = i * out_strides_0 + ph * out_strides_1 + pw *
                        out_strides_2;
63                  int bd_index = hstart * in_strides_1;
64                  for (int h = hstart; h < hend; ++h) {
65                      int wi = bd_index + wstart * in_strides_2;
66                      for (int w = wstart; w < wend; ++w) {
67                          int channels_end = channels_&(~3);
68                          for (int c = 0; c < channels_end; c+=4) {
69                              float32x4_t max_c_v = vld1q_f32(&max_c[c]);
70                              float32x4_t in_v = vld1q_f32(&batch_data[wi + c]);
71                              max_c_v = vmaxq_f32(max_c_v, in_v);
72                              vst1q_f32(&max_c[c], max_c_v);
```

```
                        // max_c[c] = MSMAX(batch_data[wi + c], max_c[c]);
                    }
                    for(int c=channels_end;c<channels_;c++){
                        max_c[c] = MSMAX(batch_data[wi + c], max_c[c]);
                    }
                    wi += in_strides_2;
                } // in_w end;
                bd_index += in_strides_1;
            } // in_h end
            for (int j = 0; j < channels_; ++j) {
                out_ptr[pooled_index + j] = max_c[j];
            }
        }
    }
    roi_ind_st += roi_stride;
}
return 0;
}
```

2.4 性能测试

2.4.1 neon、循环展开与平凡算法比较

设置 $N = 4, H = W = 64, NUM\_ROIS = 256, scale = 1.0, pooled\_height = pooled\_width = 16$，取 $C = 4, 8, 16, 32, 64, 128$，比较 neon 优化、循环展开与平凡算法之间的效率。

	naive(ms)	unroll(ms)	neon(ms)	neon/naive	unroll/naive
4	4.76	4.25	3.10	0.65	0.89
8	7.76	5.85	4.06	0.52	0.75
16	15.19	10.36	4.91	0.32	0.68
32	29.09	19.56	7.65	0.26	0.67
64	64.57	42.02	18.00	0.28	0.65
128	137.68	106.89	40.97	0.30	0.78

表 1: 不同算法在不同通道数下的用时以及优化算法与朴素算法用时之比。

当 C 充分大时，neon 能优化 70% 的时间开销，而相比之下 unroll 仅能优化不到 25%。

2.4.2 neon 多路并行比较

设定 $N = 4, H = W = 64, C = 128, NUM\_ROIS = 256, scale = 1.0, pooled\_height = pooled\_width = 16$，比较四路、八路与十六路优化的效果。

	4 路 (ms)	8 路 (ms)	16 路 (ms)
16	5.09	6.37	4.54
32	8.22	7.49	12.78
64	19.16	19.06	18.14
128	42.64	43.71	41.14

表 2: 不同多路优化在不同通道下的用时比较。

多路优化的结果和 4 路一致，没有明显的优化效果，几乎可以认为使用多路优化对于 neon 优化算法无效。

2.4.3 O0 和 O2

各参数与表1 使用的参数一致。

	naive			neon			neno/naive	
	O0(ms)	O2(ms)	O2/O0	O0(ms)	O2(ms)	O2/O0	O0	O2
4	24.07	4.76	0.20	14.40	3.10	0.22	0.60	0.65
8	47.83	7.76	0.16	23.85	4.06	0.17	0.50	0.52
16	87.69	15.19	0.17	41.47	4.91	0.12	0.47	0.32
32	170.65	29.09	0.17	80.60	7.65	0.09	0.47	0.26
64	345.83	64.57	0.19	171.67	18.00	0.10	0.50	0.28
128	687.91	137.68	0.20	347.64	40.97	0.12	0.51	0.30

表 3: 使用 O0 与 O2 优化时朴素算法和 neon 优化算法的用时以及 O0 与 O2 优化用时比值

O2 优化为朴素算法节省了 90% 的时间，对于 neon 算法甚至能节省 90% 的时间，这使得开启 O2 优化拉大了使用 neon 算法和朴素算法之间的距离。

2.5 profiling

使用 perf 对各算法的 L-dcache-load-misses、L1-dcache-loads、branch-misses、cache-misses、cycle、instruction、IPC 进行分析，参数使用和表 1 一致。

	L1-dcache-load-misses	L1-dcache-loads	branch-misses	cache-misses
naive(O2)	66.54%	13.85%	7.26%	78.39%
unroll(O2)	**35.82%**	13.66%	**5.10%**	**35.83%**
neon(O2)	46.73%	**5.05%**	5.33%	57.73%
neon(O0)	63.93%	32.70%	7.71%	64.10%

表 4: perf 分析结果 1

	cycle	instruction	IPC
naive(O2)	406764561	632201809	1.554220475
unroll(O2)	251271003	462269906	**1.839726431**
neon(O2)	**106511976**	**187852180**	1.763671907
neon(O0)	977385990	1167347710	1.194356909

表 5: perf 分析结果 2

在使用 O2 优化的情况下，unroll 整体的 cache-misses、brach-misses、IPC 均优于 neon，但是 cycle 和 instruction 的数目却劣与 neon。而未使用优化的 neon 算法在众多方面甚至比不过开启了 O2 优化的朴素算法，这体现了 O2 优化的重要性。

2.6 汇编代码比较

我们修改的部分只有比较大小的最内层函数，利用 godbolt 可以查看相应的汇编代码。

```
.L18:
        ldr     s0, [x2, x0, lsl 2]
        ldr     s1, [x1, x0, lsl 2]
        fcmpe   s0, s1
        fcsel   s0, s0, s1, gt
        str     s0, [x1, x0, lsl 2]
        add     x0, x0, 1
        cmp     w3, w0
        bgt     .L18
```

图 2.1: O2 优化的朴素算法

```
.L18:
        ldp     s4, s3, [x0]
        add     x14, x14, 16
        ldr     s0, [x14, -16]
        ldp     s2, s1, [x0, 8]
        add     x0, x0, 16
        fcmpe   s0, s4
        fcsel   s0, s0, s4, gt
        str     s0, [x0, -16]
        ldr     s0, [x14, -12]
        fcmpe   s0, s3
        fcsel   s0, s0, s3, gt
        str     s0, [x0, -12]
        ldr     s0, [x14, -8]
        fcmpe   s0, s2
        fcsel   s0, s0, s2, gt
        str     s0, [x0, -8]
        ldr     s0, [x14, -4]
        fcmpe   s0, s1
        fcsel   s0, s0, s1, gt
        str     s0, [x0, -4]
        cmp     x4, x0
        bne     .L18
```

图 2.2: O2 优化的 unroll 展开算法

```
.L18:
        ldr     q1, [x14], 16
        ldr     q0, [x0]
        fmax    v0.4s, v0.4s, v1.4s
        str     q0, [x0], 16
        cmp     x4, x0
        bne     .L18
```

图 2.3: O2 优化的 neon 算法

```
.L22:
        ldr     w1, [sp, 328]
        ldr     w0, [sp, 188]
        cmp     w1, w0
        bge     .L18
        ldrsw   x0, [sp, 328]
        lsl     x0, x0, 2
        ldr     x1, [sp, 384]
        add     x0, x1, x0
        str     x0, [sp, 64]
        ldr     x0, [sp, 64]
        ldr     q0, [x0]
        nop
        str     q0, [sp, 160]
        ldr     w1, [sp, 336]
        ldr     w0, [sp, 328]
        add     w0, w1, w0
        sxtw    x0, w0
        lsl     x0, x0, 2
        ldr     x1, [sp, 216]
        add     x0, x1, x0
        str     x0, [sp, 72]
        ldr     x0, [sp, 72]
        ldr     q0, [x0]
        nop
        str     q0, [sp, 144]
        ldr     q0, [sp, 160]
        str     q0, [sp, 96]
        ldr     q0, [sp, 144]
        str     q0, [sp, 80]
        ldr     q0, [sp, 96]
        ldr     q1, [sp, 80]
        fmax    v0.4s, v0.4s, v1.4s
        nop
        str     q0, [sp, 160]
        ldrsw   x0, [sp, 328]
        lsl     x0, x0, 2
        ldr     x1, [sp, 384]
        add     x0, x1, x0
        str     x0, [sp, 136]
        ldr     q0, [sp, 160]
        str     q0, [sp, 112]
        ldr     x0, [sp, 136]
        ldr     q0, [sp, 112]
        str     q0, [x0]
        nop
        ldr     w0, [sp, 328]
        add     w0, w0, 4
        str     w0, [sp, 328]
        b       .L22
```

图 2.4: 未优化的 neon 算法

对比朴素算法与 unroll 展开优化，可以看出优化的部分是将循环中自增 i、判断跳出循环条件以及加载的次数从 4 次优化到了 1 次，均摊让辅助的指令数在总操作数的占比降低，从而起到优化作用，而随着 unroll 路数的增多，辅助的指令的占比会逐渐趋于零而优化达到瓶颈。

对比 unroll 算法与 neon 算法，neon 的汇编语句显然更简洁，指令数少得多。同时 neon 可以明显地看到用一条指令实现 load 与内存指针自增。通过优化主要指令而提高效率，同时其辅助

对比未优化与 O2 优化的 neon 算法，未优化版本有一种朴素的美，有大量的存储与读取操作，同时还有许多的气泡来避免冒险问题，整体效率肉眼可见的低。可见一个好的编译优化方式是极其重要的。

2.7 结果分析

虽然 unroll 和 neon 优化从思路上的出发点是相似的，但在具体实现与结果上大不相同。从实现上，unroll 是均摊辅助指令数来优化时间消耗，而 neon 则是通过大幅减少主要功能指令数从本质上进行优化。从结果上，unroll 优化了缓存命中问题，有更高的 IPC，而 neon 则是大幅减少了完成操作所需的指令数。

就 O2 优化与 O0 相比而言，编译器优化对运算效率的作用是巨大的，并且对追求效率的代码的优化效果更明显。

3 实验总结和思考

- 从汇编代码和流水线角度理解了 cache 优化技术和 unroll 展开技术。
- 对 SIMD 编程有了初步的认识。