量化计算怎么写?

5.1.1 整数运算

Integral Computing

```
float c = a * b;

int    c = a * b;

char  c = a * b;
```

5.1.1 整数运算

Integral Computing

```
float c = a * b;

int   c = a * b;

char  c = a * b;
```

指令	微指令个数	指令延迟	指令吞吐量
ADD	1	1	4
ADDSS	1	3	2
PADDB	1	1	4
MUL(r8)	1	3	1
MUL(r32)	2	3	1
MULPS	1	3	2
DIVSS	1	10.5	0.3

5.1.2 运算与访存

Computing & Memory Access

float c = a * b;　　读a, b 写c, 总计 96 bit

int　c = a * b;　　读a, b 写c, 总计 96 bit

char　c = a * b;　　读a, b 写c, 总计 24 bit

5.1.2 运算与访存

Computing & Memory Access

float c = a * b;　　读a, b 写c, 总计 96 bit

int　c = a * b;　　读a, b 写c, /

char　c = a * b;　　读a, b 写c, /

操作	时间代价
取指	100（访主存） 1（访缓存）
译码	>1
访存	100（访主存） 1（访缓存）
执行	1~5
写回	100（访主存） 1（访缓存）

5.1.3 向量化运算

SIMD Computing

在处理器中，为了加速大规模运算，通常会设计专用的向量化运算指令或向量化处理器。

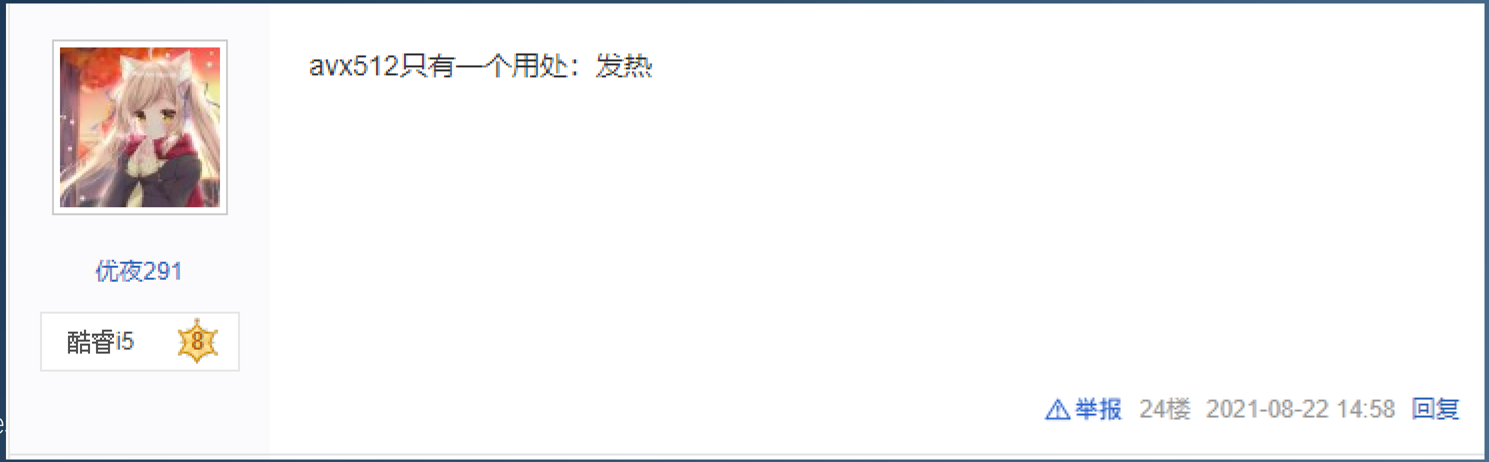
典型地，在CPU中提供AVX512指令集，可以一次性处理512bit的数据。GPU中提供TensorCore，也可以一次性处理大量数据。

5.1.3 向量化运算

SIMD Computing

在处理器中，为了加速大规模运算，通常会设计专用的向量化运算指令或向量化处理器。

典型地，在CPU中提供AVX512指令集，可以一次性处理512bit的数据。GPU中提供TensorCore，也可以一次性处理大量数据。



5.1.3 向量化运算

SIMD Computing

```
char function(char* array) {
    return (
        array[0] + array[1] + array[2] + array[3] +
        array[4] + array[5] + array[6] + array[7] +
        array[8] + array[9] + array[10] + array[11] +
        array[12] + array[13] + array[14] + array[15]
    );
}
```

5.1.3 向量化运算

SIMD Computing

```
char function(char* array) {
    return (
        array[0] + array[1] + array[2] + array[3] +
        array[4] + array[5] + array[6] + array[7] +
        array[8] + array[9] + array[10] + array[11] +
        array[12] + array[13] + array[14] + array[15]
    );
}
```

6.1.1 量化乘法

Quantized Mul

```
void Mul(
float** input_a, float** input_b,
float** output, const unsigned int num_of_elements) {
        for (unsigned int i = 0; i < num_of_elements; i++)
                for (unsigned int j = 0; j < num_of_elements; j++)
                        output[i][j] = input_a[i][j] * input_b[i][j];
}
```

6.1.1 量化乘法

Quantized Mul

```
void Mul(
char** input_a, char** input_b,
char** output, const unsigned int num_of_elements,
const float scale_a, const float scale_b, const float scale_c) {
        for (unsigned int i = 0; i < num_of_elements; i++)
                for (unsigned int j = 0; j < num_of_elements; j++)
                        output[i][j] = input_a[i][j] * input_b[i][j];
}
```

6.1.1 量化乘法

Quantized Mul

4.1 量化算子

Quantize Function

```
float value = 1.0; float scale = 0.1;

int qt32  = round_fn(value / scale);
char qt8 = clip(qt32, Q_MIN, Q_MAX)
```

4.1.3 反量化算子

Dequantize Function

```
char value = 1; float scale = 0.1;
float deq = (value * scale);
```

```
_ele
cons
_ele
for (unsigned int j = 0; j < num_of_elements; j++)
        output[i][j] = input_a[i][j] * input_b[i][j];
}
```

6.1.1 量化乘法

Quantized Mul

```
void Mul(
char** input_a, char** input_b,
char** output, const unsigned int num_of_elements,
const float scale_a, const float scale_b, const float scale_c) {
        for (unsigned int i = 0; i < num_of_elements; i++)
                for (unsigned int j = 0; j < num_of_elements; j++)
    output[i][j]  = input_a[i][j] * scale_a * input_b[i][j] * scale_b / scale_c;
}
```

6.1.1 量化乘法

Quantized Mul

```
void Mul(
char** input_a, char** input_b,
char** output, const unsigned int num_of_elements,
const float scale_a, const float scale_b, const float scale_c) {
        for (unsigned int i = 0; i < num_of_elements; i++)
                for (unsigned int j = 0; j < num_of_elements; j++)
                    output[i][j] = clip(round_fn(input_a[i][j] * input_b[i][j] * (s_a*s_b/s_c)));
}
```

$$output[i][j] = clip(round\_fn(input\_a[i][j] * input\_b[i][j] * \frac{s_a s_b}{s_c}));$$

6.1.1 量化乘法

Quantized Mul

```
void Mul(
char** input_a, char** input_b,
char** output, const unsigned int num_of_elements,
const float scale_a, const float scale_b, const float scale_c) {
        for (unsigned int i = 0; i < num_of_elements; i++)
                for (unsigned int j = 0; j < num_of_elements; j++)
   output[i][j] = clip(round_fn((input_a[i][j] * input_b[i][j]) << round(log2(sa*sb/sc))));
}
```

$$output[i][j] = clip(round\_fn((input\_a[i][j] * input\_b[i][j]) \ll round(\log_2 \frac{s_a s_b}{s_c})));$$

6.1.1 量化乘法

Quantized Mul

```
output[i][j] =
    (input_a[i][j] - offset_a) * scale_a *
    (input_b[i][j] - offset_b) * scale_b / scale_c + offset_c;
```

在对称量化基础上，再引入 offset_a，offset_b，offset_c

6.1.1 量化乘法

Quantized Mul

```
output[i][j] =
    (input_a[i][j] - offset_a) * scale_a *
    (input_b[i][j] - offset_b) * scale_b / scale_c + offset_c;
```

记作:

$$c = (((b - o_b)(a - o_a)\frac{s_a s_b}{s_c})) + o_c$$

6.1.1 量化乘法

Quantized Mul

$$c = (((b - o_b)(a - o_a)\frac{s_a s_b}{s_c})) + o_c$$

展开得：

$$c = ab\frac{s_a s_b}{s_c} - ao_b\frac{s_a s_b}{s_c} - bo_a\frac{s_a s_b}{s_c} + o_a o_b\frac{s_a s_b}{s_c} + o_c$$

6.1.1 量化乘法

Quantized Mul

$$c = (((b - o_b)(a - o_a)\frac{s_a s_b}{s_c})) + o_c$$

展开得：

$$c = ab\frac{s_a s_b}{s_c} - ao_b\frac{s_a s_b}{s_c} - bo_a\frac{s_a s_b}{s_c} + o_a o_b\frac{s_a s_b}{s_c} + o_c$$

记作：

$$c = \mathrm{rescale}((b - o_b)(a - o_a), s_a, s_b, s_c, o_c)$$

6.1.2 量化加法

Quantized Add

```
void Add(
char** input_a, char** input_b,
char** output, const unsigned int num_of_elements,
const float scale_a, const float scale_b, const float scale_c) {
        for (unsigned int i = 0; i < num_of_elements; i++)
                for (unsigned int j = 0; j < num_of_elements; j++)
    output[i][j] = (input_a[i][j] * scale_a + input_b[i][j] * scale_b) / scale_c;
}
```

6.1.2 量化加法

Quantized Add

output[i][j] = (input_a[i][j] * scale_a + input_b[i][j] * scale_b) / scale_c;

output[i][j] = (input_a[i][j] * $\frac{s_a}{s_c}$ + input_b[i][j] * $\frac{s_b}{s_c}$);

6.1.2 量化加法

Quantized Add

```
output[i][j] = (input_a[i][j] * scale_a + input_b[i][j] * scale_b) / scale_c;
```

要求 scale_a，scale_b 一致！

output[i][j] = (input_a[i][j] + input_b[i][j] * $\frac{s_i}{s_c}$);

$$c = \text{rescale}(a + b + o_b + o_a, s_i, 1, s_c, o_c)$$

6.1.3 量化激活函数

Quantized Activation

```
void Clip(
        float** input, float** output, float min, float max,
        const unsigned int num_of_elements) {
        for (unsigned int i = 0; i < num_of_elements; i++)
                for (unsigned int j = 0; j < num_of_elements; j++)
                {
                        output[i][j] = MAX(input[i][j], min);
                        output[i][j] = MIN(input[i][j], max);
                }
}
```

6.1.3 量化激活函数

Quantized Activation

```
void Clip(
        char** input, char** output, float min, float max
        const float in_scale, const float out_scale, const unsigned int num_of_elements) {
        for (unsigned int i = 0; i < num_of_elements; i++)
                for (unsigned int j = 0; j < num_of_elements; j++)
                {
                        output[i][j] = MAX(input[i][j] * in_scale, min) / out_scale;
                        output[i][j] = MIN(input[i][j]  * in_scale, max) / out_scale;
                }
}
```

6.1.3 量化激活函数

Quantized Activation

output[i][j] = MAX(input[i][j] * in_scale, min) / out_scale;

要求 in_scale, out_scale 一致!

output[i][j] = MAX(input[i][j] * scale, min) / scale;

6.1.3 量化激活函数

Quantized Activation

```
output[i][j] = MAX(input[i][j] * in_scale, min) / out_scale;
```

要求 in_scale, out_scale 一致！

```
output[i][j] = MAX(input[i][j] * scale, min) / scale;

output[i][j] = MAX(input[i][j], min / scale);
```

6.1.3 被动量化算子

Passive Quantized Operator

output[i][j] = MAX(input[i][j], min / scale);

式中 scale 与 min 参数无关，此时称 min 参数被动量化

常见的被动量化参数包括：

Bias(Gemm, Conv, Lstm),

min(Clip), max(Clip),

padding value(Pad),

6.1.3 被动量化算子

Passive Quantized Operator

output[i][j] = MAX(input[i][j], min / scale);

式中 scale 被输入和输出同时共享，此时算子的运算不改变量化参数，我们称这类算子为被动量化算子。常见的被动量化算子包括：

Pad, Clip, Relu, MaxPooling, Reshape, Concat, Split, Transpose, Slice, Permute

6.1.4 量化矩阵乘

Quantized Gemm

```
void MatMul(
        ELEMENT_TYPE** input, ELEMENT_TYPE** weight, ELEMENT_TYPE* bias,
        ELEMENT_TYPE** output, const unsigned int num_of_elements) {

        ACCUMULATOR_TYPE Accumulator[16];
        for (unsigned int i = 0; i < num_of_elements; i += 4) {
                // Pack A[i: i + 4][.], Send Packed A to L2
                ELEMENT_TYPE* packedA = LhsPackElement(input, num_of_elements, i);
                for (unsigned int j = 0; j < num_of_elements; j += 4) {
                        // Pack B[.][j: j + 4], Send Packed B to L2
                        ELEMENT_TYPE* packedB = RhsPackElement(weight, num_of_elements, j);

                        for (unsigned int k = 0; k < num_of_elements; k += 1) {
                                // Accumulator = A[i: i + 4][.] * B[.][j: j + 4]
                                MatMul4x4(packedA, packedB, Accumulator, k);
                        }

                        for (unsigned int k = 0; k < 4; k += 1) {
                                output[i + k][j + 0] = Accumulator[0] + bias[i + k];
                                output[i + k][j + 1] = Accumulator[0] + bias[i + k];
                                output[i + k][j + 2] = Accumulator[0] + bias[i + k];
                                output[i + k][j + 3] = Accumulator[0] + bias[i + k];
                        }
                }
        }
}
```

数据送上L2
改变数据排布
8bit向量化加速
8bit可以容纳更多数据

分块矩阵乘
8bit 向量化加速
结果为16 - 32位

Bias Add
32bit 向量化加速
结果为32 - 64位
Rescale 到 int8

6.1.4 量化矩阵乘

Quantized Gemm

```
// Pack A[i: i + 4][.], Send Packed A to L2

ELEMENT_TYPE* packedA = LhsPackElement(input, num_of_elements, i);

__declspec(noinline) ELEMENT_TYPE* LhsPackElement(
        ELEMENT_TYPE** input, unsigned int num_of_element, unsigned int row) {
        ELEMENT_TYPE* packed = new ELEMENT_TYPE[num_of_element * 4];
        unsigned int k = 0;
        for (unsigned int i = 0; i < num_of_element; i++) {
                for (unsigned int j = 0; j < 4; j++) {
                        packed[k++] = input[i][row + j];
                }
        }
        return packed;
}
```

6.1.4 量化矩阵乘

Quantized Gemm

```
// Accumulator = A[i: i + 4][.] * B[.][j: j + 4]

MatMul4x4(packedA, packedB, Accumulator, k);

__declspec(noinline) void MatMul4x4(
        ELEMENT_TYPE* packedA, ELEMENT_TYPE* packedB,
        ACCUMULATOR_TYPE* accumulator, unsigned int offset) {
        // 所有计算仅操作下列元素
        // PackedA[offset: offset + 16]
        // PackedB[offset: offset + 16]
        // C[16] , 共计48个，可以全部送入寄存器与L1 Cache
        accumulator[0] = packedA[0] * packedB[0] + accumulator[0];
        accumulator[1] = packedA[0] * packedB[1] + accumulator[1];
        accumulator[2] = packedA[0] * packedB[2] + accumulator[2];
        accumulator[3] = packedA[0] * packedB[3] + accumulator[3];
        // ...
}
```

6.1.4 量化矩阵乘

Quantized Gemm

```
output[i + k][j + 0] = Accumulator[0] + bias[i + k];

accumulator[0] = packedA[0] * packedB[0] + accumulator[0];

output[i + k][j + 0] = packedA[0] * scale_A * packedB[0] * scale_B + bias[i + k];
```

要求 bias_scale 与 scale_A * scale_B 一致！

猜猜看 bias_offset 是谁？

6.1.4 量化矩阵乘

Quantized Gemm

$$output[i + k][j + 0] = Clip(round(packedA[0] * packedB[0] + bias[i + k]) * \frac{s_a s_b}{s_c}));$$

1. int8 矩阵乘在累加器中运行，结果为int32
2. 处理bias_add，结果为int32
3. 执行rescale，结果为fp32
4. 取整，截断，结果为int8

6.1.5 量化非线性运算

Quantized Non - Linear Function

算子诸如：Exp, Tanh, Sigmoid, Softmax, Swish, Resize，内部包含非线性运算。

不可以直接量化，在不同处理器上做法不同：

在CPU, GPU上，这类算子的计算不量化，以全精度模式运行。

在FPGA, ASIC, DSP上，需要更改算子计算逻辑，以线性运算拟合或直接查表。

6.1.5 量化非线性运算

Quantized Non - Linear Function

$$exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + R_4$$

1. 以泰勒展开方式进行拟合
2. 对结果进行rescale，从而产生 int8 的结果

请注意，exp的数值范围极广，不利于量化

6.1.5 量化非线性运算

Quantized Non-Linear Function

$$exp(x, shift) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + R_4$$

4.1.4 整数量化

Power-of-2 Quantization

```
float value = 1.0; int shift = 1;
int qt32  = round_fn(value * (2 << shift)) ;
char qt8 = clip(qt32, Q_MIN, Q_MAX)
```

6.1.5 量化非线性运算

Quantized Non - Linear Function

$$exp(x, shift) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + R_4$$

x	shift	output(int32)	output(fp32)
1	0	3	2.71828
2	0	7	2.71828^2
3	0	20	2.71828^3
4	0	54	2.71828^4

对结果进行rescale，从而产生 int8 的结果

6.1.5 量化非线性运算

Quantized Non-Linear Function

```
template<typename Dtype>
void cuda_sigmoid_table_lookup(const int N,
                    const Dtype* input,
                    const Dtype* table,
                    Dtype* output,
                    int fragpos);

template<typename Dtype>
void cuda_sigmoid_simulation(const int N,
                    const Dtype* input,
                    Dtype* output);

template<typename Dtype>
void cuda_tanh_simulation(const int N,
                  const Dtype* input,
                  Dtype* output);

template<typename Dtype>
void cuda_tanh_table_lookup(const int N,
                  const Dtype* input,
                  const Dtype* table,
                  Dtype* output,
                  int fragpos);
```

```
template<typename Dtype>
__global__ static void _sigmoid_simulation(const int N,
                                    const Dtype fuzz,
                                    const Dtype* input,
                                    Dtype* output) {
  NNDCT_KERNEL_LOOP(i, N){
    if (input[i] >= 8.0)
      output[i] = 1.0 - fuzz;
    else if (input[i] < -8.0)
      output[i] = 0.0;
    else {
      int x = int(input[i] * pow(2, 9));
      output[i] = sigmoid_short_sim(x, 9, 8) / (pow(2.0, 8));
    }
  }
}
```

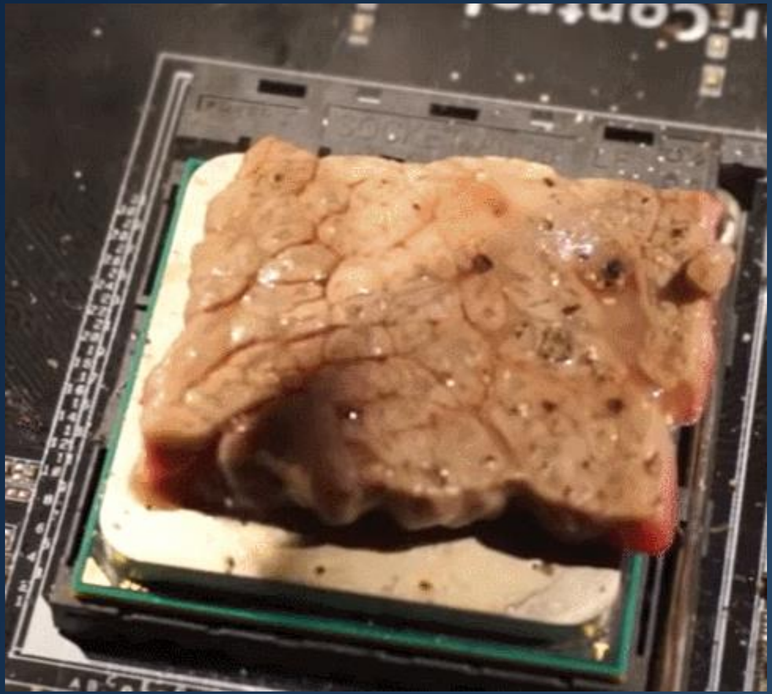
口诀

量化计算量化算，中间结果精度高
中间算完转尺度，转完尺度取整数
加法减法不能转，被动算子也一样
非线性函数查表算，不然你就等死吧

6.1.7 量化复杂算子

Quantized Complex Operation

联系我们 https://github.com/openppl-public



广告位招租

微信群

QQ群（入群密令OpenPPL）