如何优化CPU GEMM?

在 《玩转SIMD指令编程》 一文中，介绍了SIMD的概念和基础用法，也通过 **Gelu**、**Softmax** 和 **Matrix Transpose** 等比较有代表性的编程示例来演示 SIMD 该怎么玩，不过作为一篇入门级的教程，并没有进一步深入讨论一些性能优化Q技巧。

从本文开始，我准备写一系列文章来讨论计算密集型**GEMM**、**Conv**(winograd 和im2col) 算法的底层实现，同大家一起探讨更多性能优化的知识。

在深度学习推理框架或者训练框架中，GEMM 和 Conv 是典型的**计算密集型算子**，例如在 Bert 和 Conformer 模型的 self-attention 模块中存在大量矩阵运算，因此深度学习框架中 GEMM 算子的底层实现好坏将会直接影响模型的推理或训练延时。

图1 conformer 模型中的矩阵运算

介绍如何进行 GEMM 优化的文章很多，即使在知乎上随手搜索 **GEMM优化** 词条也会有几十个条目，其中也不乏一些内容翔实、条理清楚的好文章。不过，从我个人比较主观的分析来看，大部分文章停留在方法论层面的介绍，没有落实到具体的代码实现上，**理论和实践之间还是有不可跨越的鸿沟**，作为一个愣头青程序员，没能看到代码总是感觉少了点意思。

另一方面，在 GitHub: How To Optimize GEMM 项目中，作者通过清晰明了的代码和文档向读者介绍内存对齐、向量化、矩阵分块和数据打包等关键技术，此外，作者还给出了每一个步骤的优化点、优化效果对比和分析，实属不可多得的GEMM优化入门读物，强烈推荐！但 GitHub: How To Optimize GEMM 作为一个入门级的项目，旨在粗粒度介绍矩阵乘算法的优化思路，并没有针对某个硬件进行针对性优化，也没有深入优化 micro kernel 的代码实现，因此该项目中的矩阵乘实现仍然存在较大的优化空间。

那么，能不能在介绍矩阵乘优化原理Q的基础时搭配相应的代码实现，并且最终取得可观的性能表现呢?

> ***Talk is cheap. Show me the code.*** — *Linus Torvalds*

当然，这篇文章就是想做这个事情，**本文目标**有三点

1. 介绍如何在x64 CPU 上优化矩阵乘算法的思路;
2. 实现一份可运行的高性能矩阵乘算法;
3. 性能数据可复现;

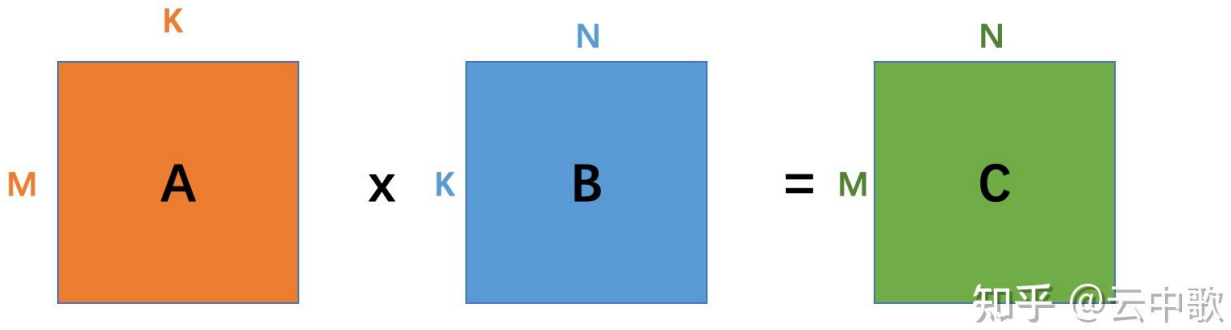

图2 矩阵乘运算

矩阵乘运算是大学本科的基础知识，原理十分简单，此处不在赘述其数学公式和讲解。

基础知识

选取一个合适的度量指标是性能优化工作的基础，通常我们使用 GFLOPS 来衡量一个算子的性能。

区分 FLOPS 和 FLOPs

每秒浮点运算次数(floating point operations per second, **FLOPS**)，即每秒所执行的浮点运算次数，是一个衡量硬件性能的指标。下表列举了常见的 FLOPS 换算指标。

缩写	解释
MFLOPS	每秒进行百万次 （10^6） 次浮点运算的次数
GFLOPS	每秒进行十亿次 （10^9） 次浮点运算的次数
TFLOPS	每秒进行万亿次 （10^12）次浮点运算的次数
PFLOPS	每秒进行千万亿次（10^15）次浮点运算的次数
EFLOPS	每秒进行百亿亿次（10^18）次浮点运算的次数

浮点运算量(floating point operations, **FLOPs**)是指浮点运算的次数，是一个衡量深度学习模型计算量的指标。

此外，从**FLOPs**延伸出另外一个指标是乘加运算量MACs。

乘加运算量(multiplication and accumulation operations, **MACs**)是指乘加运算的次数，也是衡量深度模型计算量的指标。在Intel AVX指令中，扩展了对于**乘加计算**(fused multiply-add, **FMA**)指令的支持，

即在支持AVX指令的CPU上，可以通过FMA计算单元使用一条指令来执行类似 A \times B + C 的操作，参考 Intel® C++ Compiler Classic Developer Guide and Reference 中对于 _mm256_fmadd_ps 指令的介绍。一次乘加运算包含了两次浮点运算，一般地可以认为 MACs = 2FLOPs。

计算 CPU 的 FLOPS

从上一小节中得知，FLOPS 是一个衡量硬件性能的指标，那么我们该如何计算 CPU 的FLOPS 呢?

图1 使用 lscpu 命令查看系统信息

上图中，红框中几条关键信息

1. CPU(s), 逻辑核数量;
2. CPU family, CPU系列标识，用以确定CPU属于哪一代产品。更多关于 Intel CPU Family 信息，可以参考 Intel CPUID;
3. Model, 型号标识可用来确定处理器的制作技术以及属于该系列的第几代设计（或核心），型号与系列通常是相互配合使用的，用于确定计算机所安装的处理器是属于某系列处理器的哪种特定类型。
4. Model name, CPU型号名称
5. CPU MHZ: 主频

下面以 **"Xeon Platinum 8260Y"** 细致地解释下 CPU 型号名称中隐藏的信息。

图2 Xeon Platinum 8260Y CPU

- **Xeon** Platinum 8260Y**:** Intel 公司推出的**至强处理器**系列，具备丰富的指令集支持和出色的性能表现，主要针对服务器市场。除至强处理器之外，Intel 公司推出的**酷睿处理器**在桌面市场具备更高的知名度;
- Xeon **Platinum 8**260Y**:** Intel 至强系列处理器分为四个级别，性能由高到低依次是铂金级Platinum（8，9）、黄金级Gold（6，7）、白银级Silver（4）和青铜级Bronze（3）;
- Xeon Platinum 8**2**60Y**:** 处理器架构代号，1 代表Skylake ，2 代表 Cascade Lake
- Xeon Platinum 82**60Y:** SKU和Extra Options信息可以参考 Cascade Lake 架构介绍

计算CPU FLOPS时需要两点关键信息，下面分别计算下 AVX2 和 AVX512 指令集的GFLOPS。

1. CPU 主频
2. FMA 单元数

AVX2

```
单周期双精度浮点计算能力 = 2(FMA数量)* 2(乘加) * 256 (YMM寄存器宽度) / 64(双精度浮点数位数) = 16
单周期双精度浮点计算能力= 2(FMA数量)* 2(乘加) * 256 (YMM寄存器宽度) / 32(双精度浮点数位数) = 32
双精度FLOAPS = 2.5(CPU主频) * 16(单周期双精度浮点计算能力) = 40GFLOPS
单精度FLOAPS = 2.5(CPU主频) * 32(单周期单精度浮点计算能力) = 80GFLOPS
```

AVX512

```
单周期双精度浮点计算能力 = 2(FMA数量)* 2(乘加) * 512 (YMM寄存器宽度) / 64(双精度浮点数位数) = 32
单周期双精度浮点计算能力= 2(FMA数量)* 2(乘加) * 512 (YMM寄存器宽度) / 32(双精度浮点数位数) = 64
```

```
双精度FLOAPS = 2.5(CPU主频) * 16(单周期双精度浮点计算能力) = 80 GFLOPS
单精度FLOAPS = 2.5(CPU主频) * 32(单周期单精度浮点计算能力) = 160 GFLOPS
```

指令集	精度	理论峰值算力
AVX2	double	40 GFLOPS
AVX2	float	80 GFLOPS
AVX512	double	80 GFLOPS
AVX512	float	160 GFLOPS

至此，我们已经明白了单核心CPU的理论峰值算力，下面开始进入实战环节！

基础矩阵乘实现和优化

本节内容作为正式优化的序章，会介绍两点内容

1. 如何实现基础的 GEMM 算法并测量其性能数据；
2. 如何通过一行代码达到十倍的性能提升；

此处约定本文中A，B分别为左、右输入矩阵，C为输出矩阵，并且三者的形状信息如下

A: M \times K 的输入矩阵

B: K \times N 的输入矩阵

C: M \times N 的输出矩阵

基础 GEMM 实现和度量

下面的代码应该都不陌生，矩阵乘算法是编程初学者经典的练习题之一。

```
void naive_row_major_sgemm(const float* A, const float* B, float* C, const int M,
                const int N, const int K) {
  for (int m = 0; m < M; ++m) {
    for (int n = 0; n < N; ++n) {
      for (int k = 0; k < K; ++k) {
        C[m * N + n] += A[m * K + k] * B[k * N + n];
      }
    }
  }
}
```

从矩阵乘的原理可知，矩阵乘算法的浮点运算量为 2 \times M \times N \times K，所以

GEMM : GFLOPs = \frac{2 \times M \times N \times K}{latency} \times 10^{-9}

下面实现一个朴素的GFLOPs 计算函数，相应的代码均会在 GitHub 仓库中提供。

```
void Benchmark(const std::vector<int64_t>& dims,
               std::function<void(void)> func) {
  const int warmup_times = 10;
  const int infer_times = 20;

  // warmup
  for (int i = 0; i < warmup_times; ++i) func();

  // run
  auto dtime = dclock();
  for (int i = 0; i < infer_times; ++i) func();

  // latency
  dtime = dclock() - dtime;

  // compute GLOPs
  auto flops = 2.0f * product(dims) * 1.0e-09;
  flops = flops * infer_times / dtime;

  // print
  std::cout << std::setw(20) << " GFLOPs: " << flops << std::endl;
}
```

实测，naive_row_major_sgemm 的性能数据如下

Shape(M, N, K)	**GFLOPs**
(64, 64, 64)	1.97
(128, 128, 128)	1.65
(256, 256, 256)	1.44
(512, 512, 512)	0.95
(1024, 1024, 1024)	0.62

从测试数据来看，随着矩阵尺寸的增大，GFLOPs 在不断下降。从上文的分析中可知，单核CPU的理论峰值算力是**80 GFLOPS**，naive_row_major_sgemm 和理论峰值算力之间的差距非常大，完全没有发挥出CPU的算力。

naive_row_major_sgemm 性能极差的核心原因是**在计算时发生了大量的cache miss。**

图3 基础 GEMM 实现示例

在分析清楚 naive_row_major_sgemm 性能极差的主要原因后，我们通过**循环重排**来优化访存。注意，naive_row_major_sgemm 和 optimize_row_major_sgemm 虽然只有一行代码的差距，但是性能却相差近十倍!

```
void optimize_row_major_sgemm(const float* A, const float* B, float* C, const int M,
                const int N, const int K) {
  for (int m = 0; m < M; ++m) {
    for (int k = 0; k < K; ++k) {
      for (int n = 0; n < N; ++n) {
        C[m * N + n] += A[m * K + k] * B[k * N + n];
      }
    }
  }
}
```

Shape(M, N, K)	naive GFLOPs	optimize GFLOPs
(64, 64, 64)	1.97	11.20
(128, 128, 128)	1.65	11.84
(256, 256, 256)	1.44	12.04
(512, 512, 512)	0.95	11.43
(1024, 1024, 1024)	0.62	10.79

根据上表中的数据，可以直接体会到性能优化的魔力。一行代码，十倍加速。

图4 优化访存后的 GEMM 实现示例

BLAS 接口简介

截止到目前为止，已经具有 **naive_row_major_sgemm** 和 **optimize_row_major_sgemm** 两份实现，虽然**optimize_row_major_sgemm** 在性能上有一定的优化，但距离真正的高性能计算Q库的要求还相差甚远。

即使抛开性能问题不谈，目前 optimize_row_major_sgemm 也很难视为一个合格的库函数，因为该函数在接口定义上太过随意，别人很难直接复用。众所周知，矩阵乘优化已经是非常成熟的课题了，其中自然衍生了许多标准，以方便不同开发者或者研究人员之间工作的交流和复用，其中最基础的便是 BLAS接口规范。

BLAS（basic linear algebra subroutine）是一系列基本**线性代数运算**函数1的**接口（interface）**标准。 这里的线性代数运算是指例如矢量的线性组合，矩阵乘以矢量，矩阵乘以矩阵等。接口在这里指的是诸如哪个函数名实现什么功能，有几个输入和输出变量，分别是什么。

注意 BLAS 是一个接口的标准而不是某种具体**实现（implementation）**。简单来说，就是不同的作者可以各自写出不同版本的 BLAS 库，实现同样的接口和功能，但每个函数内部的算法可以不同。这些不同导致了不同版本的 BLAS 在不同机器上运行的速度也不同。

C := alpha \times A \times B + beta \times C

- **A,** 形状为(M, K)的列主序矩阵
- **B,** 形状为(M, K)的列主序矩阵
- **C,** 形状为(M, K)的列主序矩阵

```
void sgemm(char transa, char transb, int M, int N, int K, float alpha,
            const float* A, int lda, const float* B, int ldb, float beta,  float* C, int ldc);
```

- **transa,** 设置矩阵A是否转置的标识位，'N' 表示不转置, 'T' 表示转置；
- **transb,** 设置矩阵A是否转置的标识位，'N' 表示不转置, 'T' 表示转置；
- **M,** M 维度的值；
- **N,** N 维度的值；
- **K,** K 维度的值；
- **alpha,** 系数；
- **A,** A 矩阵指针；
- **lda,** A矩阵 leading dimension的值；
- **B,** B 矩阵指针；
- **ldb,** B矩阵 leading dimension的值；
- **beta,** 系数；
- **C,** 结果矩阵C矩阵指针；
- **ldc,** C矩阵 leading dimension的值；

注： **leading dimension**，对于一个 MxN 的行优先矩阵，leading dimension 为 N；对于一个 MxN 的列优先矩阵，leading dimension 为 M。

介绍完 BLAS 接口之后，我们以 BLAS 接口的格式编写一份 **列优先的矩阵乘实现** 作为后续优化工作的比较基准。

```
void naive_col_major_sgemm(
        char transa,
        char transb,
        int M, int N, int K,
        const float alpha,
        const float * src_a, int lda,
        const float * src_b, int ldb,
        const float beta,
        float * dst, int ldc)
{
    int a_stride_m = transa == 'n' ? 1 : lda;
    int a_stride_k = transa == 'n' ? lda : 1;
    int b_stride_k = transb == 'n' ? 1 : ldb;
    int b_stride_n = transb == 'n' ? ldb : 1;

    for(int m=0;m<M;m++) {
        for(int n=0;n<N;n++) {
            float acc = 0.f;
            const float * a_ptr = src_a + m * a_stride_m;
            const float * b_ptr = src_b + n * b_stride_n;

            for(int k=0;k<K;k++) {
                acc += a_ptr[0] * b_ptr[0];
```

```
                a_ptr += a_stride_k;
                b_ptr += b_stride_k;
            }

            dst[m + n * ldc ] = alpha * acc + beta * dst[m + n * ldc];
        }
    }
}
```

深度优化矩阵乘实现

从本节起，开始演示如何优化矩阵乘算法，以达到 80% 以上的硬件性能利用率。

一般而言，矩阵乘优化有以下技巧，在GEMM、GEMV的实现中都可以去套用。

1. 循环重排；
2. 数据分块；
3. 数组打包；
4. 向量指令集；
5. 寄存器优化；
6. 多线程；

在**基础函数乘实现和优化**一节中得知，矩阵乘实现性能差的原因在与数据 cache miss 率很高，因此我们进行的一个优化就是数据打包。

```
void avx2_col_major_sgemm(char transa, char transb,int M, int N, int K, float alpha, float* A, int lda,
                          float* B, int ldb, float beta, float* C, int ldc) {
  if (alpha == 0) return;

  float beta_div_alpha = beta / alpha;

  constexpr int Mr = 64;
  constexpr int Kr = 256;

  constexpr int mr = 16;
  constexpr int nr = 6;

  // Cache a is 64 x 256
  float* pack_a = (float*)_mm_malloc(Mr * Kr * sizeof(float), 32);
  // Cache b is 256 x N
  float* pack_b = (float*)_mm_malloc(Kr * DivUp(N, nr) * sizeof(float), 32);

  float* tmp_pack_a = pack_a;
  float* tmp_pack_b = pack_b;

  for (int k = 0; k < K; k += Kr) {
    float cur_beta = 1.0 / alpha;
    if (k == 0) cur_beta = beta_div_alpha;

    int cur_k = std::min(K - k, Kr);

    // jump to k-th row of matrix B
    pack_no_trans(B + k, ldb, tmp_pack_b, Kr, cur_k, N);

    for (int i = 0; i < M; i += Mr) {
      int cur_m = std::min(M - i, Mr);

      pack_trans(A + i + k * lda, lda, tmp_pack_a, Kr, cur_k, cur_m);

      for (int j = 0; j < N;) {
        int cur_n = std::min(int(N - j), nr);
```

```
        float* cur_c = C + i + j * ldc;

        float* packed_cur_b = tmp_pack_b + DivDown(j, nr) * Kr + j % nr;

        sgemm_block_n(cur_m, cur_n, cur_k, alpha, tmp_pack_a, lda, packed_cur_b,
                      ldb, cur_beta, cur_c, ldc);
        j += cur_n;
      }
    }
  }

  _mm_free(pack_a);
  _mm_free(pack_b);
}
```

在后文的讲解中，为方便起见，统一设置 **M = N = K = 512** 为例，来演示矩阵乘优化。

数据打包

从系统信息上看，L1 数据缓存和指令缓存均为 32 K，32K 的 L1d cache 可以容纳 32 * 1024 / 4 = 8192 个单精度浮点数。因此，当 M, N, K 足够大的时候，L1d cache 无法持有三个矩阵所有的数据，便会发生cache miss，这也解释了上文中为什么矩阵越大、性能越差。

```
L1d cache:           32K
L1i cache:           32K
L2 cache:            4096K
L3 cache:            36608K
```

在 **avx2_col_major_sgemm** 的实现代码中，为矩阵A 开辟了 64 x 256 x 4 bytes / 1024 = 64 K 的存储区域，为矩阵B 开辟了 256 x *Divp(N=512,6) = 256 x 516 x 4 bytes / 1024 = 516 K* 的存储区域，目的是防止矩阵A和矩阵B过大，以至于在L2 cache 中发生cache miss 的情况，所以一次只在L2中加载矩阵A和矩阵B的子矩阵，保证不会发生cache miss。

```
constexpr int Mr = 64;
constexpr int Kr = 256;

...

// Cache a is 64 x 256
float* pack_a = (float*)_mm_malloc(Mr * Kr * sizeof(float), 32);
// Cache b is 256 x N
float* pack_b = (float*)_mm_malloc(Kr * DivUp(N, nr) * sizeof(float), 32);
```

矩阵乘实现从计算方法上来区分，可以分为 Inner Product 和 Outer Product 两种计算方法，解释如下

1. Inner Product: 按行切分A矩阵，按列切分B矩阵，使用A矩阵的一个按行切分的子块同B矩阵按列切分的子块做矩阵乘法，即求得结果矩阵C矩阵的一个子矩阵。依次循环，求得最终结果。

图5 矩阵分块运算(inner product)

2. Outer Product: 按列切分A矩阵，按行切分B矩阵，使用A矩阵的一个按列切分的子块同B矩阵按行切分的子块做矩阵乘法，求得一个形状同C矩阵相同的中间结果矩阵。依次循环，对所有的中间结果矩阵求和，可得最终结果。下图中，将A、B矩阵切分为4个子矩阵，然后进行4次矩阵乘，再对C1、C2、C3 和 C4 进行求和，可以算出最终结果。

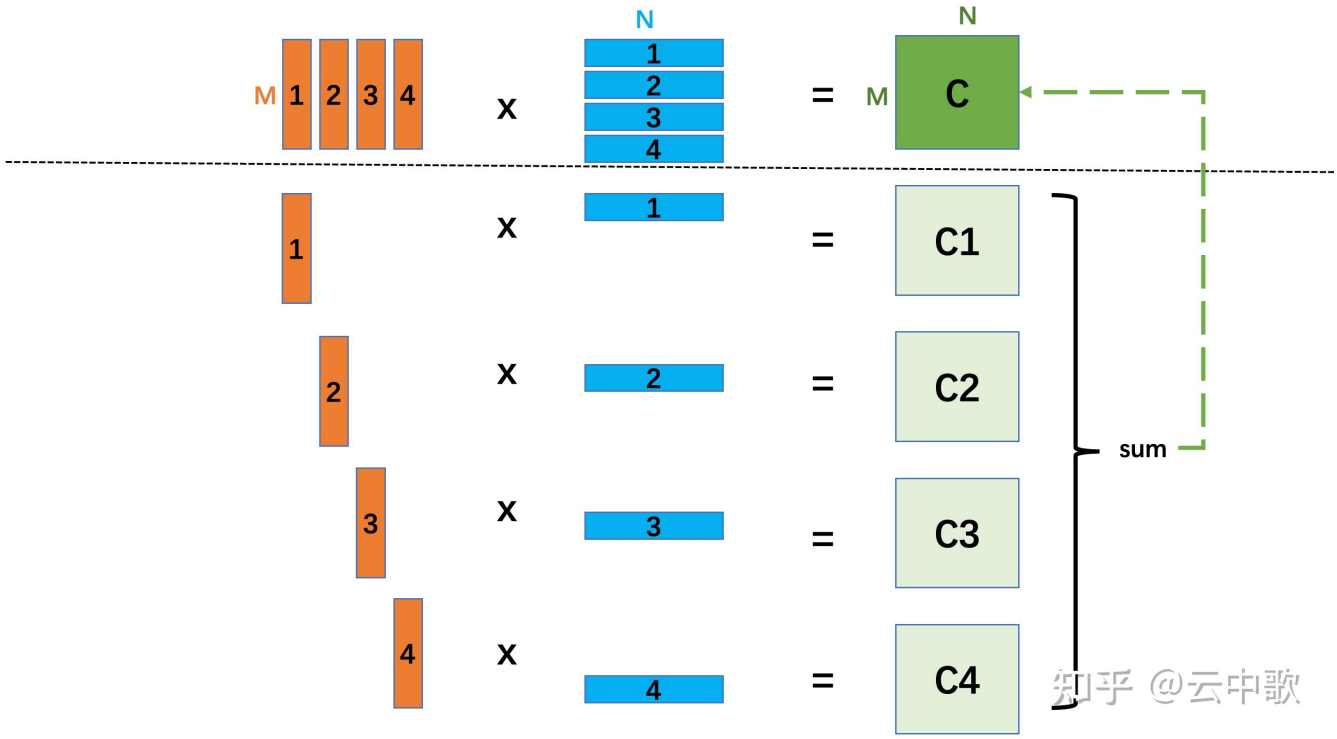

图6 矩阵分块运算示例(outer product)

在 **avx2_col_major_sgemm** 的实现代码中，按照如下方式对矩阵A（512 x 512）、B（512 x 512）进行切分计算，具体步骤如下：

1. 将矩阵A（512 x 512）切分为 2 x 8 = 16 个形状为 64 x 256 的子矩阵；
2. 将矩阵B（512 x 512）切分为 2 个形状为 256 x 512 的子矩阵；
3. 对矩阵B的1号子矩阵进行数据打包， 然后对矩阵A的1号子矩阵进行数据打包，对TMPA1（64x256）和 TMPB1（256 x 512）进行一次矩阵乘运算，求得图中的 c11 (64 x 512) ; 在对矩阵A的2号子矩阵进行数据打包，求得c12;依次循环，直到求得 c18;
4. 对矩阵B的2号子矩阵进行数据打包， 然后对矩阵A的9号子矩阵进行数据打包，对TMPA1（64x256）和 TMPB1（256 x 512）进行一次矩阵乘运算，求得图中的 c21 (64 x 512) ; 在对矩阵A的10号子矩阵进行数据打包，求得c22;依次循环，直到求得 c28;
5. 对中间结果矩阵进行求和，可得最终的结果矩阵 C。

图7 M=N=K=512 矩阵的切分计算示例

通过上面的介绍，相信读者已经对如何进行矩阵分块有了清晰的认识，其实矩阵分块的思想很简单，就是将原始输入矩阵切分为小矩阵，使得L2 cache可以容纳计算所需的小矩阵。

现在已经粗粒度的讲解了如何对矩阵A和矩阵B进行分块计算，那么矩阵A的子矩阵（64 x 256）和矩阵B的子矩阵（256 x 512）是如何计算的呢？

1. 在数据打包时，将子矩阵 a（64 x 256）按行进行切分，分为 4 个形状为 16 x 256 的小矩阵；
2. 在数据打包时，将子矩阵 b（256 x 512）按列进行切分，分为 86 个形状为 256 x 6 的小矩阵；当子矩阵 b 的列数不是 6 的整数倍时，需在数据打包时，进行 padding。
3. 使用子矩阵 a 的1号子矩阵（16 x 256）依次和子矩阵 b 的86个子矩阵进行矩阵乘计算，计算结果为 （16 x 256） X （256 x 6） = （16 x 6）；最终可得（16 x 6） x 86 个子矩阵；
4. 依此遍历子矩阵 a 的1、2、3、4号子矩阵进行步骤3中的运算；

图8 左矩阵(64x256)和右矩阵(256x512)的计算

上文的描述中，详细介绍如何对矩阵A的子矩阵（64 x 256）和矩阵B的子矩阵（256 x 512）进行计算，后面会结合代码对如何使用SIMD指令进行数据打包的细节演示。

矩阵A的数据打包

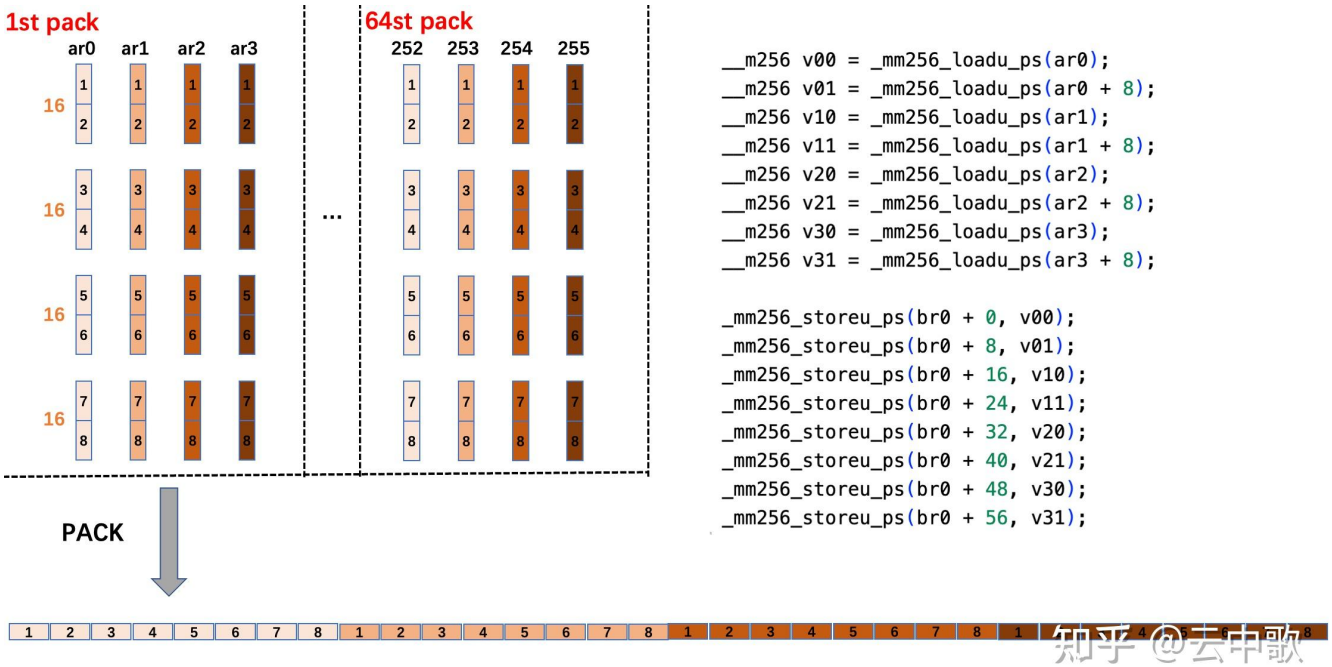

图9 矩阵A的数据打包

代码实现，暂时没进行深入讲解，比较好理解。

```
//  pack block_size on leading dimension, t denotes transpose.
//  eg. input:   A MxN matrix in row major, so the storage-format is (M, N)
//      output:  B MxN matrix in col major(N-packed), so the storage-format is
//                    (divUp(N, 16), M, 16)
void pack_trans(float* a, int lda, float* b, int ldb, int m, int n) {
  constexpr int block_size = 16;
  int i = 0;

  for (; i + 64 <= n; i += 64) {
    float* cur_a = a + i;
```

```
    float* cur_b = b + i * ldb;
    pack_trans_4x16(cur_a, lda, cur_b, ldb, m, block_size);
  }
}

void pack_trans_4x16(float* a, const int lda, float* b, const int ldb, int m, int n) {
  const int m4 = m / 4;
  const int m1 = m % 1;
  const int block_size = 64;
  const int ldbx16 = ldb * 16;  //(256 * 64)

  float* tmpa = a;
  // 保存指针 A 的4列元素
  float* ar0 = tmpa + 0 * lda;
  float* ar1 = tmpa + 1 * lda;
  float* ar2 = tmpa + 2 * lda;
  float* ar3 = tmpa + 3 * lda;

  float* tmpb = b;
  float* br0 = tmpb + 0 * ldbx16;
  float* br1 = tmpb + 1 * ldbx16;
  float* br2 = tmpb + 2 * ldbx16;
  float* br3 = tmpb + 3 * ldbx16;

  // 循环 256 / 4 = 64 次, 每次 pack 4 x 16 = 64 个数据
  for (int i = 0; i < m4; ++i) {
    {
      __m256 v00 = _mm256_loadu_ps(ar0);
      __m256 v01 = _mm256_loadu_ps(ar0 + 8);
      __m256 v10 = _mm256_loadu_ps(ar1);
      __m256 v11 = _mm256_loadu_ps(ar1 + 8);
      __m256 v20 = _mm256_loadu_ps(ar2);
      __m256 v21 = _mm256_loadu_ps(ar2 + 8);
      __m256 v30 = _mm256_loadu_ps(ar3);
      __m256 v31 = _mm256_loadu_ps(ar3 + 8);

      _mm256_storeu_ps(br0 + 0, v00);
      _mm256_storeu_ps(br0 + 8, v01);
      _mm256_storeu_ps(br0 + 16, v10);
      _mm256_storeu_ps(br0 + 24, v11);
      _mm256_storeu_ps(br0 + 32, v20);
      _mm256_storeu_ps(br0 + 40, v21);
      _mm256_storeu_ps(br0 + 48, v30);
      _mm256_storeu_ps(br0 + 56, v31);
    }
    {
      __m256 v00 = _mm256_loadu_ps(ar0 + 16);
      __m256 v01 = _mm256_loadu_ps(ar0 + 24);
      __m256 v10 = _mm256_loadu_ps(ar1 + 16);
      __m256 v11 = _mm256_loadu_ps(ar1 + 24);
      __m256 v20 = _mm256_loadu_ps(ar2 + 16);
      __m256 v21 = _mm256_loadu_ps(ar2 + 24);
      __m256 v30 = _mm256_loadu_ps(ar3 + 16);
      __m256 v31 = _mm256_loadu_ps(ar3 + 24);

      _mm256_storeu_ps(br1 + 0, v00);
      _mm256_storeu_ps(br1 + 8, v01);
      _mm256_storeu_ps(br1 + 16, v10);
      _mm256_storeu_ps(br1 + 24, v11);
      _mm256_storeu_ps(br1 + 32, v20);
      _mm256_storeu_ps(br1 + 40, v21);
      _mm256_storeu_ps(br1 + 48, v30);
      _mm256_storeu_ps(br1 + 56, v31);
    }

    {
      __m256 v00 = _mm256_loadu_ps(ar0 + 32);
      __m256 v01 = _mm256_loadu_ps(ar0 + 40);
```

```
      __m256 v10 = _mm256_loadu_ps(ar1 + 32);
      __m256 v11 = _mm256_loadu_ps(ar1 + 40);
      __m256 v20 = _mm256_loadu_ps(ar2 + 32);
      __m256 v21 = _mm256_loadu_ps(ar2 + 40);
      __m256 v30 = _mm256_loadu_ps(ar3 + 32);
      __m256 v31 = _mm256_loadu_ps(ar3 + 40);

      _mm256_storeu_ps(br2 + 0, v00);
      _mm256_storeu_ps(br2 + 8, v01);
      _mm256_storeu_ps(br2 + 16, v10);
      _mm256_storeu_ps(br2 + 24, v11);
      _mm256_storeu_ps(br2 + 32, v20);
      _mm256_storeu_ps(br2 + 40, v21);
      _mm256_storeu_ps(br2 + 48, v30);
      _mm256_storeu_ps(br2 + 56, v31);
    }

    {
      __m256 v00 = _mm256_loadu_ps(ar0 + 48);
      __m256 v01 = _mm256_loadu_ps(ar0 + 56);
      __m256 v10 = _mm256_loadu_ps(ar1 + 48);
      __m256 v11 = _mm256_loadu_ps(ar1 + 56);
      __m256 v20 = _mm256_loadu_ps(ar2 + 48);
      __m256 v21 = _mm256_loadu_ps(ar2 + 56);
      __m256 v30 = _mm256_loadu_ps(ar3 + 48);
      __m256 v31 = _mm256_loadu_ps(ar3 + 56);

      _mm256_storeu_ps(br3 + 0, v00);
      _mm256_storeu_ps(br3 + 8, v01);
      _mm256_storeu_ps(br3 + 16, v10);
      _mm256_storeu_ps(br3 + 24, v11);
      _mm256_storeu_ps(br3 + 32, v20);
      _mm256_storeu_ps(br3 + 40, v21);
      _mm256_storeu_ps(br3 + 48, v30);
      _mm256_storeu_ps(br3 + 56, v31);
    }

    ar0 += 4 * lda;
    ar1 += 4 * lda;
    ar2 += 4 * lda;
    ar3 += 4 * lda;

    br0 += block_size;
    br1 += block_size;
    br2 += block_size;
    br3 += block_size;
  }
}
```

矩阵B的数据打包

图10 矩阵B的数据打包

代码实现，暂时没进行深入讲解，比较好理解。

```
id pack_no_trans_n6(float* a, const int lda, float* b, const int ldb,
                    const int m, const int n) {
  const int m8 = m / 8;
  const int m1 = m % 8;
  const int block_size = n;
```

```
  float* tmpa = a;
  float* tmpb = b;
  float* a0 = tmpa + 0 * lda;
  float* a1 = tmpa + 1 * lda;
  float* a2 = tmpa + 2 * lda;
  float* a3 = tmpa + 3 * lda;
  float* a4 = tmpa + 4 * lda;
  float* a5 = tmpa + 5 * lda;

  for (int i = 0; i < m8; i++) {
    __m256 v0 = _mm256_loadu_ps(a0);
    __m256 v1 = _mm256_loadu_ps(a1);
    __m256 v2 = _mm256_loadu_ps(a2);
    __m256 v3 = _mm256_loadu_ps(a3);
    __m256 v4 = _mm256_loadu_ps(a4);
    __m256 v5 = _mm256_loadu_ps(a5);

    __m256 unpack0 = _mm256_unpacklo_ps(v0, v1);
    __m256 unpack1 = _mm256_unpackhi_ps(v0, v1);
    __m256 unpack2 = _mm256_unpacklo_ps(v2, v3);
    __m256 unpack3 = _mm256_unpackhi_ps(v2, v3);
    __m256 unpack4 = _mm256_unpacklo_ps(v4, v5);
    __m256 unpack5 = _mm256_unpackhi_ps(v4, v5);

    __m256 shf0 = _mm256_shuffle_ps(unpack0, unpack2, 0x44);
    __m256 shf1 = _mm256_shuffle_ps(unpack4, unpack0, 0xe4);
    __m256 shf2 = _mm256_shuffle_ps(unpack2, unpack4, 0xee);
    __m256 shf3 = _mm256_shuffle_ps(unpack5, unpack1, 0xe4);
    __m256 shf4 = _mm256_shuffle_ps(unpack3, unpack5, 0xee);
    __m256 shf5 = _mm256_shuffle_ps(unpack1, unpack3, 0x44);

    __m128 low_shf1 = _mm256_castps256_ps128(shf1);
    __m256 res0 = _mm256_insertf128_ps(shf0, low_shf1, 0x1);
    __m256 res1 = _mm256_permute2f128_ps(shf0, shf1, 0x31);

    __m128 low_shf5 = _mm256_castps256_ps128(shf5);
    __m256 res2 = _mm256_insertf128_ps(shf2, low_shf5, 0x1);
    __m256 res3 = _mm256_permute2f128_ps(shf2, shf5, 0x31);

    __m128 low_shf4 = _mm256_castps256_ps128(shf4);
    __m256 res4 = _mm256_insertf128_ps(shf3, low_shf4, 0x1);
    __m256 res5 = _mm256_permute2f128_ps(shf3, shf4, 0x31);

    constexpr int vsize_in_bytes = 8;
    _mm256_storeu_ps(tmpb + 0 * vsize_in_bytes, res0);
    _mm256_storeu_ps(tmpb + 1 * vsize_in_bytes, res2);
    _mm256_storeu_ps(tmpb + 2 * vsize_in_bytes, res4);
    _mm256_storeu_ps(tmpb + 3 * vsize_in_bytes, res1);
    _mm256_storeu_ps(tmpb + 4 * vsize_in_bytes, res3);
    _mm256_storeu_ps(tmpb + 5 * vsize_in_bytes, res5);

    tmpb += 6 * vsize_in_bytes;

    // jump to another 8 float point values
    a0 += vsize_in_bytes;
    a1 += vsize_in_bytes;
    a2 += vsize_in_bytes;
    a3 += vsize_in_bytes;
    a4 += vsize_in_bytes;
    a5 += vsize_in_bytes;
  }
}
```

寄存器优化(Micro Kernel)

在数据打包的讲解中，有以下描述

> *使用子矩阵 a 的1号子矩阵（16 x 256）依次和子矩阵 b 的86个子矩阵进行矩阵乘计算，计算结果为 （16 x 256）X （256 x 6） = （16 x 6）；最终可得（16 x 6）x 86 个子矩阵；*

在 **avx2_col_major_sgemm** 的实现中，使用 A(16, 8) * B(8, 6) = C(16, 6) 的Micro Kernel，其计算思路如下，图片和下面的描述均来自一篇很好的文章 《OneDNN GEMM(AVX FP32)算法浅析》。

图11 micro kernel 寄存器优化

Micro Kernel 的计算步骤如下描述

1. 在 micro kernel 中，首先使用12个YMM寄存器用以保存结果矩阵 C（shape 为 16x6）；
2. 通过_mm256_loadu_ps指令将A矩阵的第一列移动到两个YMM寄存器中（这里假设为YMM0以及YMM1）；
3. 对于B矩阵第一行的第一个元素，使用_mm256_broadcast_ss指令进行广播并存储到一个YMM寄存器内（这里假设为YMM2），然后使用fma指令_mm256_fmadd_ps将YMM0和YMM1内的元素与YMM2内元素对应相乘，并将结果累加到C矩阵的两个YMM寄存器内，这里假设为YMM4以及YMM5；
4. 沿着B矩阵第一行进行循环，重复步骤2，B矩阵广播当前行内其它数据时重复使用YMM2寄存器，并将计算结果依次累加到YMM6~YMM15寄存器内；
5. A矩阵前进一列，B矩阵前进一行，并重复步骤1~3，最终完成整个C(16, 6)矩阵的计算。

```
void col_major_micro_kernel_m16n6(const int K, const float alpha,
                                  const float* src_a, const int lda,
                                  const float* src_b, int ldb, const float beta,
                                  float* dst_c, int ldc) {
  constexpr int m_block_size = 16;
  constexpr int n_block_size = 6;

  // Load result matrix c (shape 16x6) into 12 x __m256 vector values
  __m256 c00 = _mm256_loadu_ps(dst_c + 0 * ldc);
  __m256 c01 = _mm256_loadu_ps(dst_c + 0 * ldc + 8);

  __m256 c10 = _mm256_loadu_ps(dst_c + 1 * ldc);
  __m256 c11 = _mm256_loadu_ps(dst_c + 1 * ldc + 8);

  __m256 c20 = _mm256_loadu_ps(dst_c + 2 * ldc);
  __m256 c21 = _mm256_loadu_ps(dst_c + 2 * ldc + 8);

  __m256 c30 = _mm256_loadu_ps(dst_c + 3 * ldc);
  __m256 c31 = _mm256_loadu_ps(dst_c + 3 * ldc + 8);

  __m256 c40 = _mm256_loadu_ps(dst_c + 4 * ldc);
  __m256 c41 = _mm256_loadu_ps(dst_c + 4 * ldc + 8);

  __m256 c50 = _mm256_loadu_ps(dst_c + 5 * ldc);
  __m256 c51 = _mm256_loadu_ps(dst_c + 5 * ldc + 8);

  // c = c * beta
  __m256 vbeta = _mm256_set1_ps(beta);
```

```
c00 = _mm256_mul_ps(c00, vbeta);
c01 = _mm256_mul_ps(c01, vbeta);

c10 = _mm256_mul_ps(c10, vbeta);
c11 = _mm256_mul_ps(c11, vbeta);

c20 = _mm256_mul_ps(c20, vbeta);
c21 = _mm256_mul_ps(c21, vbeta);

c30 = _mm256_mul_ps(c30, vbeta);
c31 = _mm256_mul_ps(c31, vbeta);

c40 = _mm256_mul_ps(c40, vbeta);
c41 = _mm256_mul_ps(c41, vbeta);

c50 = _mm256_mul_ps(c50, vbeta);
c51 = _mm256_mul_ps(c51, vbeta);


for (int k = 0; k < K; ++k) {
  __m256 a0 = _mm256_loadu_ps(src_a);
  __m256 a1 = _mm256_loadu_ps(src_a + 8);

  __m256 vb = _mm256_broadcast_ss(src_b);
  c00 = _mm256_fmadd_ps(a0, vb, c00);
  c01 = _mm256_fmadd_ps(a1, vb, c01);

  vb = _mm256_broadcast_ss(src_b + 1);
  c10 = _mm256_fmadd_ps(a0, vb, c10);
  c11 = _mm256_fmadd_ps(a1, vb, c11);

  vb = _mm256_broadcast_ss(src_b + 2);
  c20 = _mm256_fmadd_ps(a0, vb, c20);
  c21 = _mm256_fmadd_ps(a1, vb, c21);

  vb = _mm256_broadcast_ss(src_b + 3);
  c30 = _mm256_fmadd_ps(a0, vb, c30);
  c31 = _mm256_fmadd_ps(a1, vb, c31);

  vb = _mm256_broadcast_ss(src_b + 4);
  c40 = _mm256_fmadd_ps(a0, vb, c40);
  c41 = _mm256_fmadd_ps(a1, vb, c41);

  vb = _mm256_broadcast_ss(src_b + 5);
  c50 = _mm256_fmadd_ps(a0, vb, c50);
  c51 = _mm256_fmadd_ps(a1, vb, c51);

  src_a += m_block_size;
  src_b += n_block_size;
}

__m256 valpha = _mm256_set1_ps(alpha);
c00 = _mm256_mul_ps(c00, valpha);
c01 = _mm256_mul_ps(c01, valpha);

c10 = _mm256_mul_ps(c10, valpha);
c11 = _mm256_mul_ps(c11, valpha);

c20 = _mm256_mul_ps(c20, valpha);
c21 = _mm256_mul_ps(c21, valpha);

c30 = _mm256_mul_ps(c30, valpha);
c31 = _mm256_mul_ps(c31, valpha);

c40 = _mm256_mul_ps(c40, valpha);
c41 = _mm256_mul_ps(c41, valpha);

c50 = _mm256_mul_ps(c50, valpha);
```

```
  c51 = _mm256_mul_ps(c51, valpha);

  _mm256_storeu_ps(dst_c + 0 * ldc, c00);
  _mm256_storeu_ps(dst_c + 0 * ldc + 8, c01);

  _mm256_storeu_ps(dst_c + 1 * ldc, c10);
  _mm256_storeu_ps(dst_c + 1 * ldc + 8, c11);

  _mm256_storeu_ps(dst_c + 2 * ldc, c20);
  _mm256_storeu_ps(dst_c + 2 * ldc + 8, c21);

  _mm256_storeu_ps(dst_c + 3 * ldc, c30);
  _mm256_storeu_ps(dst_c + 3 * ldc + 8, c31);

  _mm256_storeu_ps(dst_c + 4 * ldc, c40);
  _mm256_storeu_ps(dst_c + 4 * ldc + 8, c41);

  _mm256_storeu_ps(dst_c + 5 * ldc, c50);
  _mm256_storeu_ps(dst_c + 5 * ldc + 8, c51);
}
```

性能数据

在经历过漫长的讲解之后，那么 avx2_col_major_sgemm 的性能究竟如何呢？且看下表中的数据，表中使用了两个比较基准作为参照，分别是

1. **Naive**, 最基础的矩阵乘算法实现，代码文中已经提供；
2. **oneDNN sgemm**，oneDNN是英特尔公司大名鼎鼎的多平台支持、高性能计算库，其前身是 mkldnn。oneDNN 在各类硬件上都进行了深度优化，特别是在Intel CPU 上，其性能数据非常具备参考价值。

Shape(M, N, K)	Naive GFLOPs	oneDNN sgemm GFLOPs	avx2_col_major_sgemm
(64, 64, 64)	1.96	32.97	35.42
(128, 128, 128)	1.65	62.69	40.36
(256, 256, 256)	1.44	73.19	65.84
(512, 512, 512)	0.95	70.06	67.65
(1024, 1024, 1024)	0.61	79.73	69.12

从数据上来看，avx2_col_major_sgemm 相较于 Naive 实现已经具备了质的飞跃，并且在多数shapes下可以取得接近 oneDNN 的性能，不过在 (128, 128, 128) 下和oneDNN 存在比较大的性能差异，这也说明 avx2_col_major_sgemm 仍然存在一定的优化空间。这很令人兴奋，不是么？

闲谈

今天是年前最后一个工作日了，抽出了一点点时间完成这篇拖了许久的文章。由于时间比较着急，文中有些地方也没有讲的很清楚，后续会慢慢补齐，也欢迎大家私信和在评论区交流讨论。

参考