CS205 · C/C++ Program Design

Matrix Multiplication

Monad

November 28, 2022

目录

1 分析与优化

1.1 开始之前

在开始之前，我们先说明一些性能测试的约定。

仅使用 $O(n^3)$ 算法

虽然目前有像 Strassen 等算法，可以把复杂度降低至 $O(n^3)$ 以下。但是我认为，这个 Project 是想让我们即使在算法不变的情况下，如何通过合理的优化，优化矩阵乘法的速度。

所以本报告就专注于最朴素的 $O(n^3)$ 算法，并对其进行优化。

使用 `-O3` 编译

首先，本报告中的所有结果均使用 `-O3` 编译。

因为所有不开 `-O3` 的性能测试都是耍流氓，无论是用 `register`，还是把 `i++` 改成 `++i` 的优化，都只是关公面前耍大刀而已。这些细枝末节的优化，只要加上 `-O3`，编译器都会帮我们做好，甚至能比我们做得更好。俗话说得好，「人生三大错觉：我比编译器聪明，我超越了标准库，我能管理好内存」。目前编译器的优化十分强大，它甚至可以把一些运算直接向量化（下面会提到）。

所以我们现在的优化目标，基本上都是如何让编译器生成出更优秀的指令，而不仅仅是写出更优秀的代码（除非手写汇编）。

以“计算时间”与“GFLOPS”衡量

然后，本报告中的结果都以“计算时间”和“GFLOPS”衡量。

其中 GFLOPS，即 Giga Floating-point Operations Per Second（每秒浮点运算次数，十亿），可以衡量 CPU 的 throughput。具体于矩阵乘法而言，对于一个大小为 n, m, k 的矩阵乘法而言，即 $\mathbb{R}^{n\times m} \leftarrow \mathbb{R}^{n\times k} \times \mathbb{R}^{k\times m}$，它的总浮点运算为 $2nmk$。如果计算时间为 s，那么 GFLOPS 就是

$$\text{GFLOPS} = \frac{2nmk}{10^9 \cdot s}$$

其中计算时间用 Google Benchmark 执行，对于每个测试点均运行至少 20 秒，以总时间除以迭代次数作为单次矩阵乘法的耗时。然后对 16 到 1024 的区间上，以 16 为步长，将其中的每一个值作为矩阵的长宽，测量乘法的耗时，绘制成图。

当然这与题目要求的 16、128、1k、8k、64k 有所区别，主要的考虑是这样以 16 为步长，并且以 GFLOPS 为衡量标准，可以看到 CPU 的性能发挥得如何，也直观地看到随着矩阵大小的增长，GFLOPS 的变化趋势，以此来分析缓存的影响。并且 GFLOPS 无论是在 n 较小还是较大的时候，都具有较好的辨识度，不像时间那样，线与线之间会过宽或者过窄。

1.2 暴力

1.2.1 纯暴力

首先按照惯例我们还是从暴力开始，即

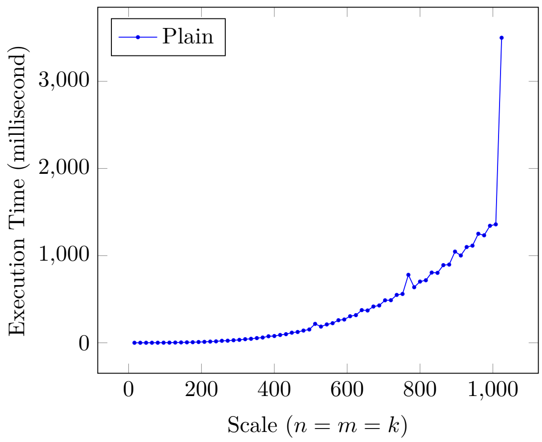
```
void plain_gemm(size_t N, size_t M, size_t K, const float *lhs, const float *rhs, float *dst) {
    for (size_t r = 0; r < N; r++)
        for (size_t c = 0; c < M; c++)
            for (size_t k = 0; k < K; k++)
                C(r, c) += A(r, k) * B(k, c);
}
```

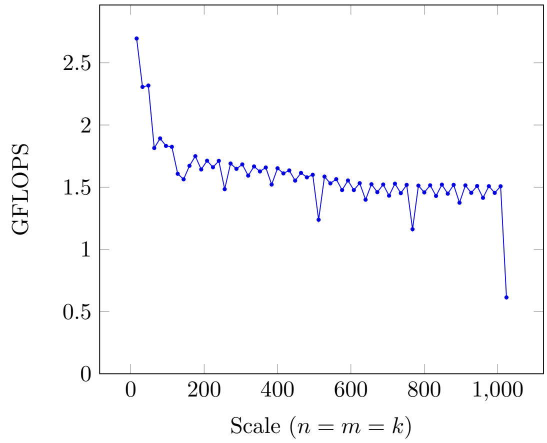
这里用了 `A`，`B`，`C` 三个宏来帮助我们访问三个数组，它们的定义分别是

```
#define A(i, j) lhs[ i * M + j ]
#define B(i, j) rhs[ i * K + j ]
#define C(i, j) dst[ i * M + j ]
```

然后因为我们开了 `-O3`，所以我们也不用担心 `register` 或者 `i++` 与 `++i` 的区别影响性能，编译器能比我们做得更好。

然后暴力跑出来的效率是：

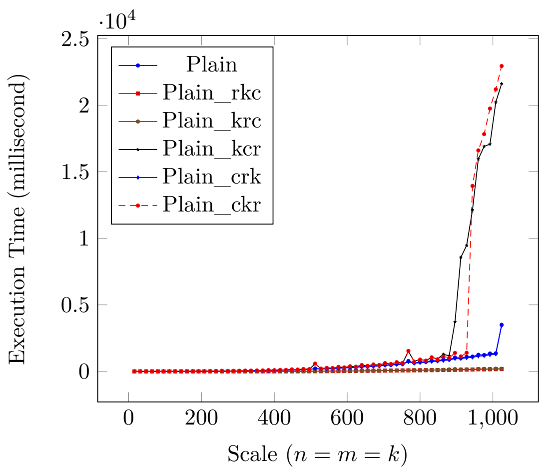


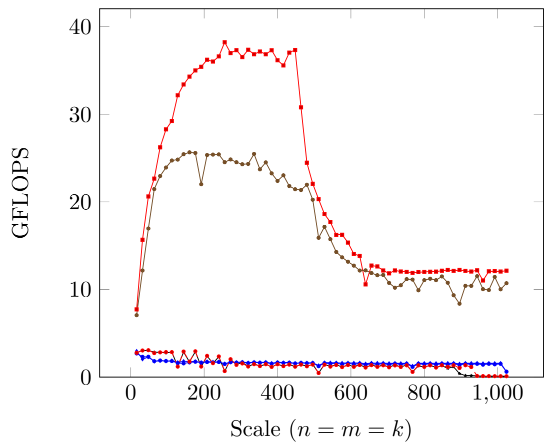


看起来，也就那样……？我们试试能不能做得更好。

1.2.2 更换循环顺序

以上面的暴力为基础，仅调换 `r`，`c`，`k` 的相对位置，然后测算不同排列对时间的影响。它们的耗时如下图所示：



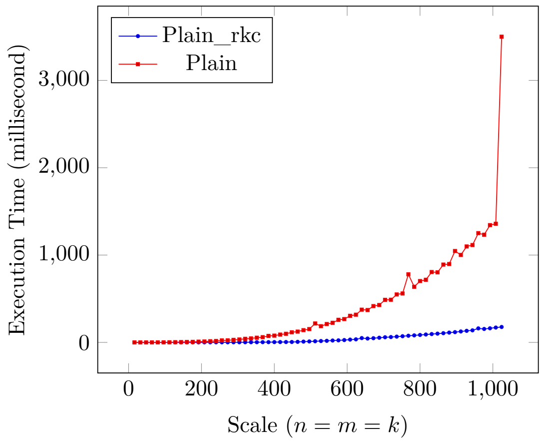


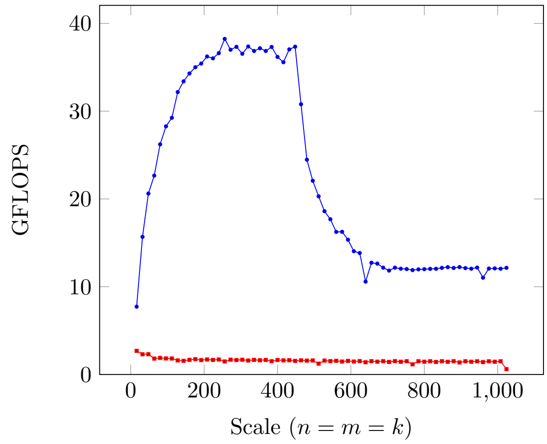
可以看到，rkc 的效率十分优秀，然后 krc 紧随在它后面。对于 rck 和 crk 中规中矩。但是像 ckr 和 kcr 就比较惨烈，在 n 超过 900 左右的时候，GFLOPS 就开始大幅下降（耗时也把其它顺序"甩"在后面）。

我们这里重点分析一下 rkc。对于 rkc 而言，它的代码是

```
void plain_gemm(size_t N, size_t M, size_t K, const float *lhs, const float *rhs, float *dst) {
    for (size_t r = 0; r < N; r++)
        for (size_t k = 0; k < K; k++)
            for (size_t c = 0; c < M; c++)
                C(r, c) += A(r, k) * B(k, c);
}
```

然后我们单独把它的性能拿出来与纯暴力对比一下：





顺序访问的效率

可以看到，GFLOPS 从 1.5 上升到了 10（对于 n 较大的时候）至 40（n 较小的时候）。这个效率提升足足有 10 倍，确实有点唬人。

并且后面我手动循环展开，或者手动用 AVX512 进行优化的时候，它们的效果甚至都没有它好。我当时都有点头疼，为什么只是一个调换循环顺序，就可以提升近 10 倍，而且还能吊打我随随便便的优化？

要回答这个问题，首先看最内层循环，只有 `c` 在迭代，这时候对于 B 和 C 而言都是顺序访问，不像其它循环顺序那样跳着访问。

如果是其它循环跳着访问的话，那么几乎每次操作都是 cache miss，那么 CPU 每次都要从更低级的储存（内存）中调取数据，而这个过程是非常慢的，可以消耗几个时钟周期。那么这么一叠加起来，CPU 真正做计算的时间就很少，真正的时间都花在数据读写上了，就十分地不值。

如果想 rkc 这样子连续，CPU 的缓存相对来说就不会那么容易失效，从而提高运算效率。

然后出于好奇“为什么手动循环展开”反而会降低效率，我反编译了一下：

```
  76b50:       48 83 7c 24 20 06       cmpq   $0x6,0x20(%rsp)
  76b56:       0f 86 94 01 00 00       jbe    76cf0 <plain_rkc_gemm+0x350>
  76b5c:       c4 e2 7d 18 0a          vbroadcastss (%rdx),%ymm1
  76b61:       31 ff                   xor    %edi,%edi
  76b63:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
  76b68:       c5 fc 10 04 38          vmovups (%rax,%rdi,1),%ymm0
  76b6d:       c4 c2 75 a8 04 3c       vfmadd213ps (%r12,%rdi,1),%ymm1,%ymm0
  76b73:       c4 c1 7c 11 04 3c       vmovups %ymm0,(%r12,%rdi,1)
  76b79:       48 83 c7 20             add    $0x20,%rdi
  76b7d:       49 39 fd                cmp    %rdi,%r13
  76b80:       75 e6                   jne    76b68 <plain_rkc_gemm+0x1c8>
  76b82:       4d 39 f7                cmp    %r14,%r15
  76b85:       0f 84 07 01 00 00       je     76c92 <plain_rkc_gemm+0x2f2>
  76b8b:       c5 fa 10 02             vmovss (%rdx),%xmm0
  76b8f:       c4 c1 7a 10 1a          vmovss (%r10),%xmm3
  76b94:       49 8d 3c 0e             lea    (%r14,%rcx,1),%rdi
  76b98:       c4 c2 61 99 04 bb       vfmadd132ss (%r11,%rdi,4),%xmm3,%xmm0
  76b9e:       48 8b 7c 24 18          mov    0x18(%rsp),%rdi
  76ba3:       c4 c1 7a 11 02          vmovss %xmm0,(%r10)
  76ba8:       49 39 ff                cmp    %rdi,%r15
  76bab:       0f 86 e1 00 00 00       jbe    76c92 <plain_rkc_gemm+0x2f2>
  76bb1:       4c 8b 4c 24 08          mov    0x8(%rsp),%r9
  76bb6:       c5 fa 10 02             vmovss (%rdx),%xmm0
```

可以看到，编译器对于这样连续而简单的计算，可以直接给我们生成 AVX512 指令，从而利用 CPU 的 SIMD 功能进行加速。而且还能看到这指令还自带循环展开，十分的牛。

而如果我们手动进行循环展开，反而会让编译器感到迷惑，从而不给我们生成高效的指令。所以现在写代码，除了能直接写出高性能的代码之外，另一个途径也可以是写出能让编译器看得懂的代码，从而给我们生成高效的指令。

L2 缓存

另外一个十分有意思的现象就是，在 n 比较小的时候，GFLOPS 可以达到将近 40，但是当 n 超过 400 之后，GFLOPS 就开始大幅下降，只剩下了 10 左右。这十分地有意思。

首先要说明的是，运行 benchmark 的设备，L2 缓存有 1024 KiB。然后在 $n = 512$ 左右的时候，单个矩阵的大小就是 $4 \times n^2 = 1024$ KiB，非常巧合，它跟 L2 大小十分相近。

那么一个合理的分析就是，当矩阵大小较小的时候，B 和 C 两个矩阵就可以整个塞进 CPU 的 L2 缓存中，在整个矩阵乘法的过程中不会被换入和换出，从而不会消耗很多的数据传输时间。当矩阵大小太大的时候，矩阵就无法塞入 L2 中，在数据换入换出上就会消耗一定的时间，从而降低效率。

1.2.3 分块

既然编译器已经为我们自动地进行了循环展开、用 AVX512 指令等优化方法，而且上面也说道，手动加优化也只能使性能劣化。当时我确实一度非常忧郁，本来还想对比这些优化方法所带来的性能提升，但是编译器直接给我们全做了，这简直就是「走别人的路，让别人无路可走」，确实有点麻。虽然这也可以从一个侧面证明循环展开与 AVX512 的威力。

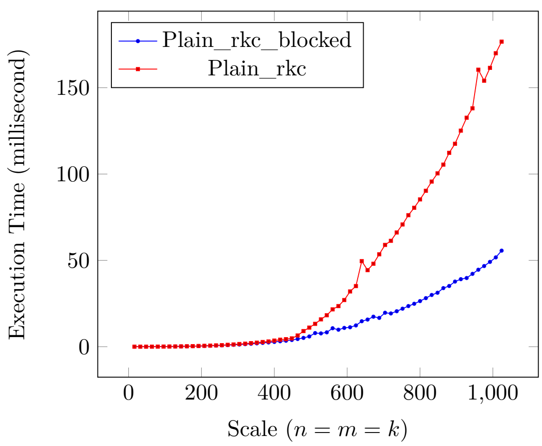
不过话说回来，这些方法编译器都帮我们卷了，我们真的只能止步于此了吗？不是。

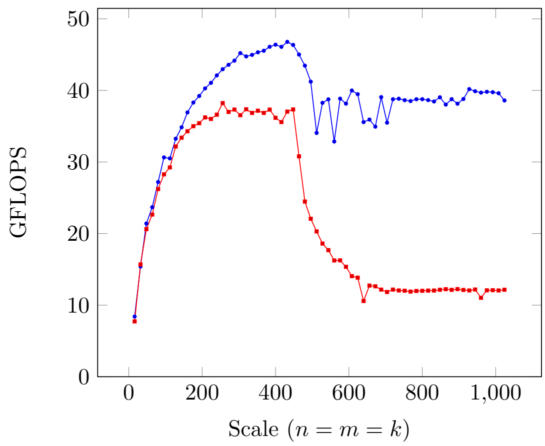
虽然编译器可以对一些简单的运算做优化，但是对于计算顺序的优化，至少目前来说，还是比较少涉及的。而计算顺序，往往与矩阵的访问和局部性非常相关，从而会影响效率。所以我们可以对计算的顺序进行调整，增加访问的局部性，从而加快效率。

一个十分显而易见的途径就是分块。

```
void plain_rkc_blocked_gemm(size_t N, size_t M, size_t K, const float *lhs, const float *rhs, float *dst) {
    // 分块
    for (size_t r0 = 0; r0 < N; r0 += 16)
        for (size_t k0 = 0; k0 < K; k0 += 16)
            // 块内计算
            for (size_t r = 0; r < 16; r++)
                for (size_t k = 0; k < 16; k++)
                    for (size_t c = 0; c < M; c++)
                        C(r0 + r, c) += A(r0 + r, k0 + k) * B(k0 + k, c);
}
```

然后分块之后的效率，与分块之前的对比如下，



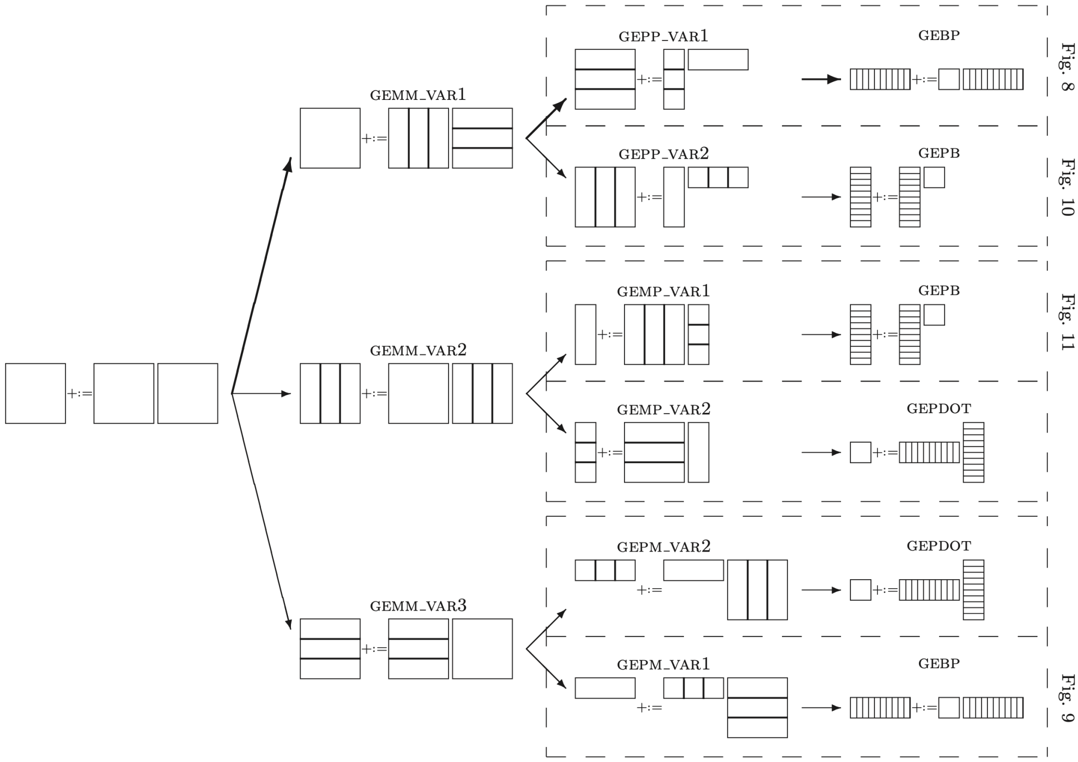


可以看到，在 $n = 400$ 之前，GFLOPS 比分块之前的还要高一点。然后在 $n = 400$ 之后，GFLOPS 虽然也下降了一点点，但是也没有大幅下降，最终也可以挺在 40 左右，非常优秀。

1.3 GEPB

1.3.1 基础 GEPB

上面也提到，要进行进一步优化，最主要的一点就是要调整执行顺序，提高访问局部性。然后经过翻阅论文，发现 GOTO 的 Anatomy of High-Performance Matrix Multiplication 非常有意思，它里面提出了一种分块方式：

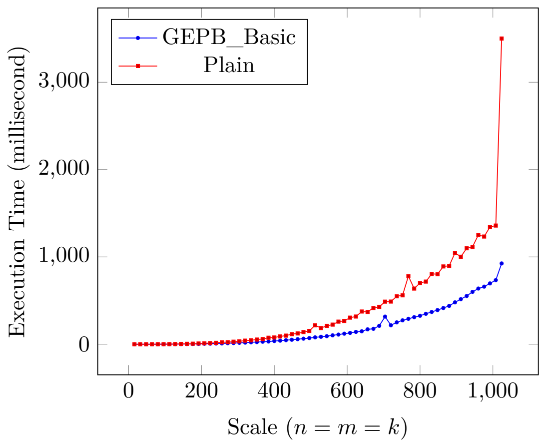


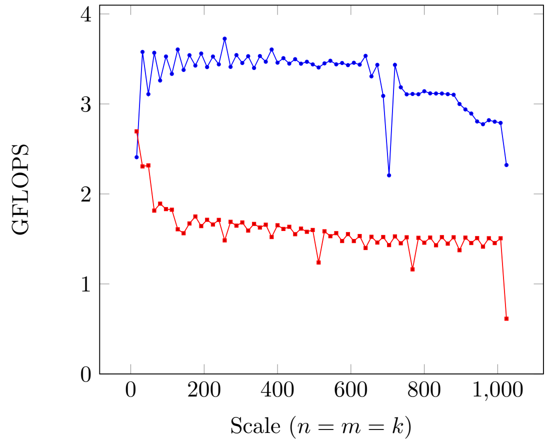
然后论文分析说明，对于 row major 矩阵，采用 GEPB（即 Fig. 10）方式最为高效。

并且根据论文的建议，这里选取了 16 作为分块大小。把这种分块方式写成代码，就是

```
void gepb_gemm(size_t N, size_t M, size_t K, const float *lhs, const float *rhs, float *dst) {
    for (size_t k0 = 0; k0 < K; k0 += 16)            // Layer 1
        for (size_t c0 = 0; c0 < M; c0 += 16)        // Layer 2
            for (size_t r = 0; r < N; r++)           // Layer 3
                for (size_t c = 0; c < 16; c++)
                    for (size_t k = 0; k < 16; k++)
                        C(r, c0 + c) += A(r, k0 + k) * B(k0 + k, c0 + c);
}
```

然后跑一下性能测试，大概是这样的：





但是好像⋯⋯就这样看的话，就比暴力快一丢丢，貌似也好不到哪里去?

不过别急，后面还有分析和优化。

1.3.2 重排 B 矩阵

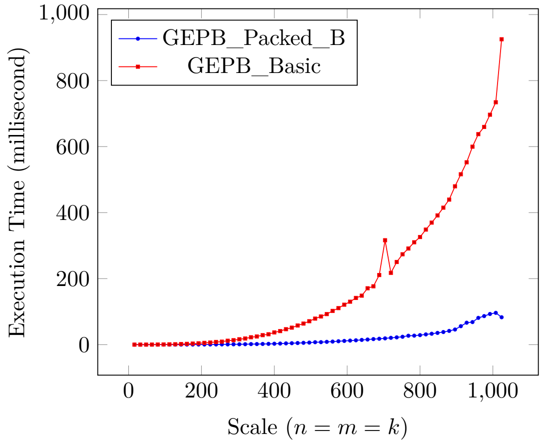
经过简单分析一下上面的代码，可以发现 B 矩阵在 Layer 3 里面，只会访问到其中 16×16 的区域。但是像上面那样访问的话，因为这 16×16 的区域是分散在不连续的内存中的，访问的话会产生非常大的 cache miss。所以我们可以在 Layer 3 外面，提前把 B 的这一区域重排为一段连续的内存，这样就能提高缓存命中率。

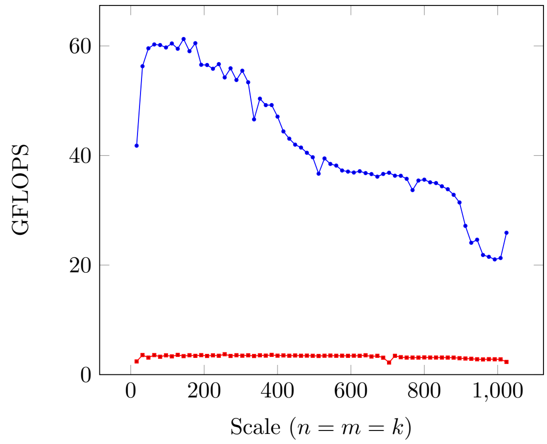
要做到这点，我们定义一个局部变量 temp（因为足够小，所以可以放在栈上），然后把代码更新成下面这样：

```
void gepb_packed_b_gemm(size_t N, size_t M, size_t K, const float *lhs, const float *rhs, float *dst) {
    float temp[16 * 16];

    for (size_t k0 = 0; k0 < K; k0 += 16) {            // Layer 1
        for (size_t c0 = 0; c0 < M; c0 += 16) {        // Layer 2
            for (size_t c = 0; c < 16; c++)
                for (size_t k = 0; k < 16; k++)
                    temp[c * 16 + k] = B(k0 + k, c0 + c);

            for (size_t r = 0; r < N; r++)             // Layer 3
                for (size_t c = 0; c < 16; c++)
                    for (size_t k = 0; k < 16; k++)
                        C(r, c0 + c) += A(r, k0 + k) * temp[c * 16 + k];
        }
    }
}
```

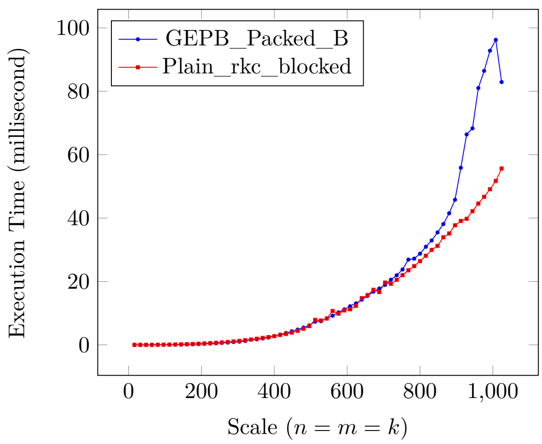
这样，原本在 Layer 3 中进行的 r 组不连续的内存访问，就被扔到了循环外面，以一组不连续的内存访问的代价，把 B 中 16×16 的区域重排为一段连续的内存，就可以加快 Layer 3 的速度。

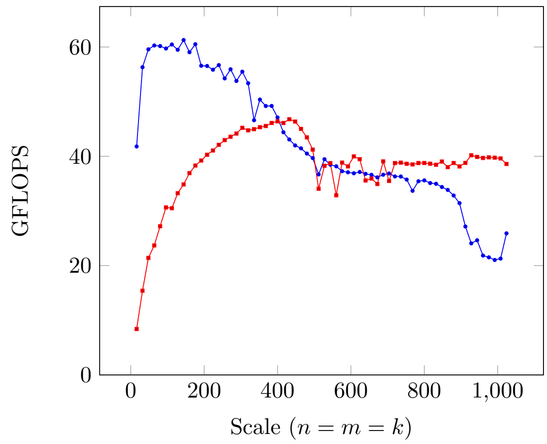
然后经过这么一番优化之后，它的速度来到了：





确实非同反向。然后我们也可以用它来跟分块之后的 rkc 对比一下：





虽然后面稍微有一点拉胯，但是不管怎么说，现在这个方法已经与分块的 rkc 相差不大了，只要再优化一点点，我们就可以超越它。

1.3.3 重排 A 矩阵

然后按照类似的思路，我们也可以发现，在 Layer 2 里面，我们也只会访问到 A 中 $16 \times N$ 的部分，然后这个部分在内存中也是不连续的。

我们可以如法炮制，将 A 矩阵在 Layer 2 外面重排，重排完之后的 A' 就只剩下 $16 \times N$ 那么大，并且除了内存连续了之外，它的大小也可以塞入 L2 缓存中，对性能的提升也有很大的帮助（或者说 GFLOPS 受矩阵大小的影响较小）。

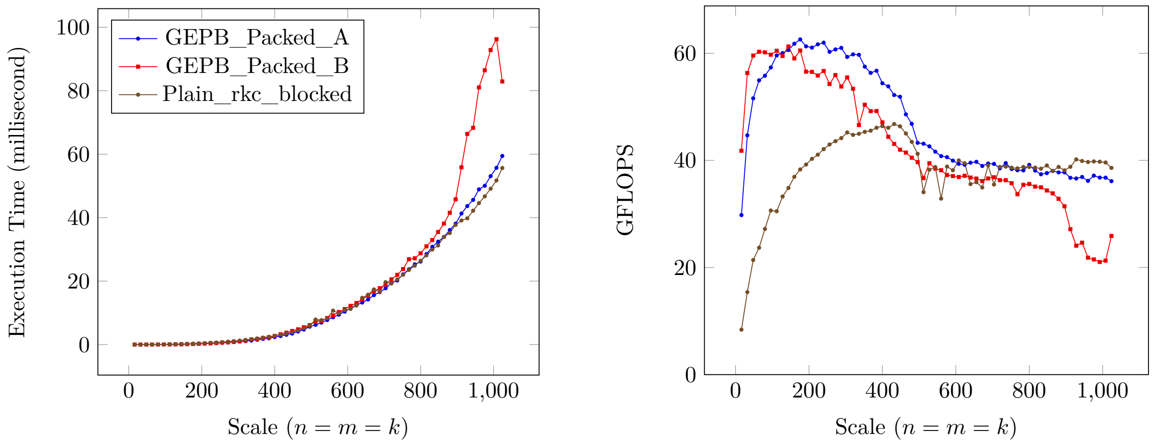
当然这里 $16 \times N$ 就不一定能塞入栈上了，所以需要先在堆上提前申请好空间，然后重排在它上面。

写成代码，就是

```
void gepb_packed_a_gemm(size_t N, size_t M, size_t K, const float *lhs, const float *rhs, float *dst) {
    float temp[16 * 16];
    float *A_packed = aligned_alloc(1024, 16 * N * sizeof(float));  // note: add error handling

    for (size_t k0 = 0; k0 < K; k0 += 16) {
        for (size_t r = 0; r < N; r++)
            for (size_t k = 0; k < 16; k++)
                A_packed[r * 16 + k] = A(r, k0 + k);

        for (size_t c0 = 0; c0 < M; c0 += 16) {
            for (size_t c = 0; c < 16; c++)
```

```
                for (size_t k = 0; k < 16; k++)
                    temp[c * 16 + k] = B(k0 + k, c0 + c);

            for (size_t r = 0; r < N; r++)
                for (size_t c = 0; c < 16; c++)
                    for (size_t k = 0; k < 16; k++)
                        C(r, c0 + c) += A_packed[r * 16 + k] * temp[c * 16 + k];
        }
    }

    free(A_packed);
}
```

然后对其进行一下性能测试，有



感觉上，虽然当 n 比较小的时候，性能比 rkc 分块优秀。但是当 n 增大的时候，GFLOPS 也开始降低，变成与 rkc 分块差不多。这也说明，当 n 增大的时候，该算法仍然受到了缓存的影响，存在访问不连续内存造成的缓存失效。

所以理论上还存在继续优化的空间。

1.3.4 重排 C 矩阵

不过我当时也确实想了很久，该重排的也重排了，分块大小也是 16 最优秀，也不知道剩下啥能优化了。然后也非常惆怅，论文里的方法，总不能和一个分块的暴力差不多吧？

不过后来确实也想到了优化的一个点，就是 C 这个矩阵，我们是一个竖条一个竖条那样访问的（就是 $N \times 16$），这样在行与行之间移动的时候，因为是不连续内存，就会产生一定的 cache miss。

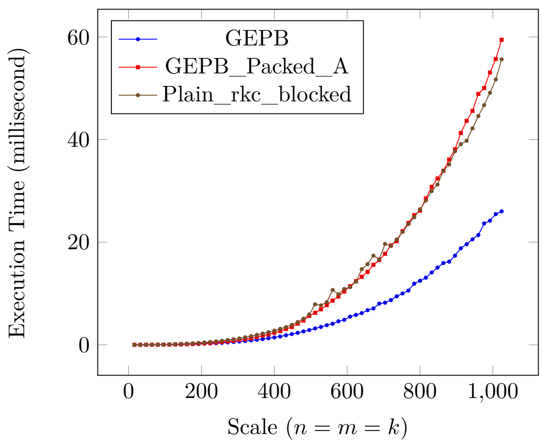
所以我们可以先把 C 给重排成 C'，让这些访问连续，然后整个矩阵乘法计算完成之后，再将其重排回 C。虽然这样子空间消耗有点大，但是性能确实是有所提升的。

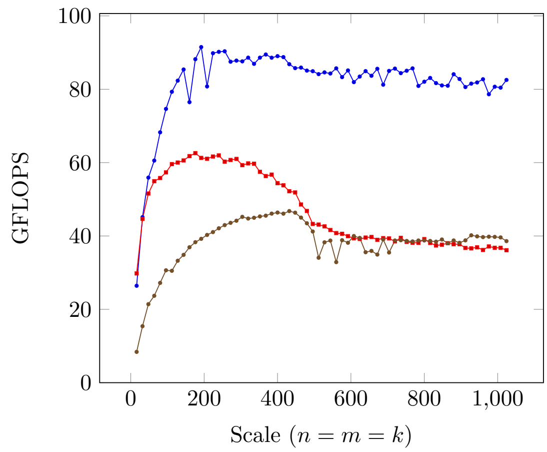
然后还是先给出代码：

```
void gepb_packet_c_gemm(size_t N, size_t M, size_t K, const float *lhs, const float *rhs, float *dst) {
    float temp[16 * 16];
    float *A_packed = aligned_alloc(1024, 16 * N * sizeof(float));
    float *C_packed = aligned_alloc(1024, N * N * sizeof(float));  // note: add error handling

    for (size_t k0 = 0; k0 < K; k0 += 16) {
        for (size_t r = 0; r < N; r++)
            for (size_t k = 0; k < 16; k++)
                A_packed[r * 16 + k] = A(r, k0 + k);
```

```
            for (size_t c0 = 0; c0 < M; c0 += 16) {
                for (size_t c = 0; c < 16; c++)
                    for (size_t k = 0; k < 16; k++)
                        temp[c * 16 + k] = B(k0 + k, c0 + c);

                float *ptr = C_packed + c0 * N;
                for (size_t r = 0; r < N; r++)
                    for (size_t c = 0; c < 16; c++) {
                        float x = 0;  // C(r, c0 + c)
                        for (size_t k = 0; k < 16; k++)
                            x += A_packed[r * 16 + k] * temp[c * 16 + k];
                        *ptr++ += x;
                    }
            }
        }

        float *ptr = C_packed;
        for (size_t c0 = 0; c0 < M; c0 += 16)
            for (size_t r = 0; r < N; r++)
                for (size_t c = 0; c < 16; c++)
                    C(r, c0 + c) = *ptr++;

        free(A_packed);
        free(C_packed);
}
```

然后性能是





可以看到性能提升十分巨大，GFLOPS 提高到了 80 到 90 左右，甩开了 rkc 分块和上一种优化一大截。更重要的是，随着 n 的增大，GFLOPS 没有明显降低，可以坚挺在 80 以上，这是非常好的一个迹象，说明我们的算法访问局部性非常优秀，cache miss 较小。

1.3.5 对齐

虽然我们申请内存的时候，使用 `aligned_alloc` 对齐了内存。但是编译器并不知道运行的时候，`gepb_gemm` 的参数以及 `A_packed` 等数组是否对齐了。

所以编译器生成指令的时候，例如，就有可能生成 `_mm512_loadu_ps` 而非 `_mm512_load_ps`，而前者是用于加载非对齐内存的，会有一定的性能损失。所以我就想，能不能告诉编译器，这些数组是已经对齐好了的，让它生成更高效的指令呢？

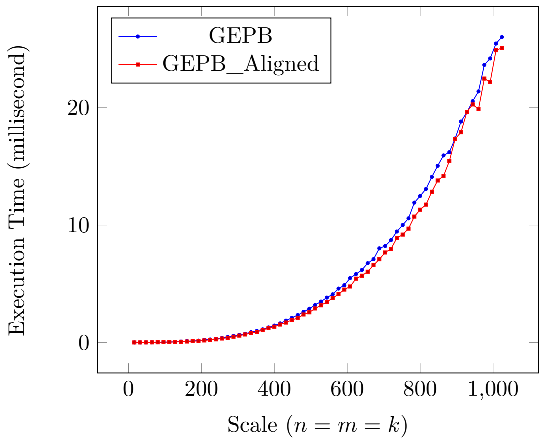
答案是可以的。比如说我们在声明一个指针的时候（即 `float *array`），我们可以在它前面加上 `__attribute__((aligned( /* alignment HERE */ )))`，即

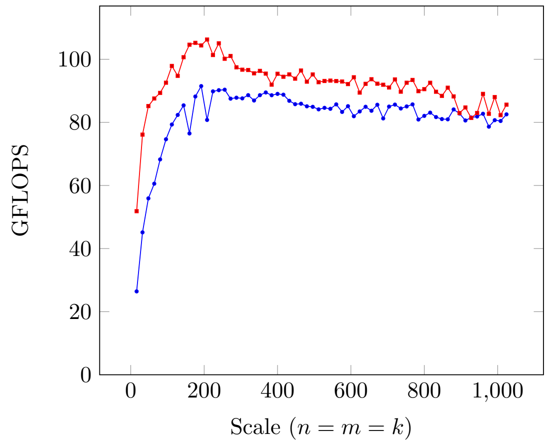
```
__attribute__((aligned(1024))) float *A_packed;
```

然后对于函数参数，我们首先需要用 `typedef` 定义一个对齐的 `float`，再把它用在参数声明里面：

```
typedef float __attribute__((aligned(1024))) float_aligned;

void example_memcpy(float_aligned * __restrict__ dst, const float_aligned * __restrict__ src);
```

这样编译器就知道这些指针是已经对齐了的。然后跑一下性能测试：





确实比之前的版本又有了进一步的提升。

1.3.6 分析

然后我们用 `objdump` 看看编译器给我们生成了什么指令：

Piece 1

```
76c03:  c5 f8 29 79 30          vmovaps %xmm7,0x30(%rcx)
76c08:  48 01 f8                add     %rdi,%rax
76c0b:  62 f1 fe 08 6f 18       vmovdqu64 (%rax),%xmm3
76c11:  c5 f8 29 1e             vmovaps %xmm3,(%rsi)
76c15:  62 f1 fe 08 6f 60 01    vmovdqu64 0x10(%rax),%xmm4
76c1c:  48 8b 8c 24 a0 00 00    mov     0xa0(%rsp),%rcx
76c23:  00
76c24:  c5 f8 29 66 10          vmovaps %xmm4,0x10(%rsi)
76c29:  62 f1 fe 08 6f 68 02    vmovdqu64 0x20(%rax),%xmm5
76c30:  c5 f8 29 6e 20          vmovaps %xmm5,0x20(%rsi)
76c35:  62 f1 fe 08 6f 70 03    vmovdqu64 0x30(%rax),%xmm6
76c3c:  48 8b 84 24 38 01 00    mov     0x138(%rsp),%rax
76c43:  00
76c44:  c5 f8 29 76 30          vmovaps %xmm6,0x30(%rsi)
76c49:  48 01 f8                add     %rdi,%rax
76c4c:  62 f1 fe 08 6f 10       vmovdqu64 (%rax),%xmm2
76c52:  48 8b b4 24 80 00 00    mov     0x80(%rsp),%rsi
76c59:  00
76c5a:  c5 f8 29 11             vmovaps %xmm2,(%rcx)
76c5e:  62 f1 fe 08 6f 58 01    vmovdqu64 0x10(%rax),%xmm3
76c65:  c5 f8 29 59 10          vmovaps %xmm3,0x10(%rcx)
76c6a:  62 f1 fe 08 6f 60 02    vmovdqu64 0x20(%rax),%xmm4
76c71:  c5 f8 29 61 20          vmovaps %xmm4,0x20(%rcx)
76c76:  62 f1 fe 08 6f 68 03    vmovdqu64 0x30(%rax),%xmm5
76c7d:  48 8b 84 24 40 01 00    mov     0x140(%rsp),%rax
```

Piece 2

```
77019:   62 42 2d 20 b8 fc       vfmadd231ps %ymm12,%ymm26,%ymm31
7701f:   c4 62 15 98 a4 24 20    vfmadd132ps 0x220(%rsp),%ymm13,%ymm12
77026:   02 00 00
77029:   c4 c1 7c 58 c6          vaddps %ymm14,%ymm0,%ymm0
7702e:   62 91 74 28 58 cf       vaddps %ymm31,%ymm1,%ymm1
77034:   62 41 44 20 59 fb       vmulps %ymm11,%ymm23,%ymm31
7703a:   c5 24 59 9c 24 80 02    vmulps 0x280(%rsp),%ymm11,%ymm11
77041:   00 00
77043:   c4 c1 7c 58 c4          vaddps %ymm12,%ymm0,%ymm0
77048:   62 42 3d 20 b8 fa       vfmadd231ps %ymm10,%ymm24,%ymm31
7704e:   c4 62 25 98 94 24 60    vfmadd132ps 0x260(%rsp),%ymm11,%ymm10
77055:   02 00 00
77058:   62 91 74 28 58 cf       vaddps %ymm31,%ymm1,%ymm1
7705e:   62 41 54 20 59 f9       vmulps %ymm9,%ymm21,%ymm31
77064:   c5 34 59 8c 24 c0 02    vmulps 0x2c0(%rsp),%ymm9,%ymm9
7706b:   00 00
7706d:   c4 c1 7c 58 c2          vaddps %ymm10,%ymm0,%ymm0
77072:   62 42 4d 20 b8 f8       vfmadd231ps %ymm8,%ymm22,%ymm31
77078:   c4 62 35 98 84 24 a0    vfmadd132ps 0x2a0(%rsp),%ymm9,%ymm8
7707f:   02 00 00
77082:   62 91 74 28 58 cf       vaddps %ymm31,%ymm1,%ymm1
77088:   62 61 64 20 59 ff       vmulps %ymm7,%ymm19,%ymm31
7708e:   c5 c4 59 bc 24 00 03    vmulps 0x300(%rsp),%ymm7,%ymm7
```

可见，编译器确实看懂了我们写的代码，它不仅把这些简单的运算向量化，而且它似乎还把 Layer 3 里面的两重循环都给展开了，然后从上到下 SIMD 指令密度非常高，并且没有条件跳转，在加上前面对缓存局部性的优化，加起来可以有效利用 CPU 的长流水线，加快运算效率。

1.4 多线程并行

对于 GEPB 而言，对最外层的循环展开任务划分有点不是很合适。因为最外层循环 Layer 1 会对整个 C 矩阵进行读写，如果进行多线程的话，就会发生竞争读写，从而会降低性能，甚至会产生错误的结果。

参考大部分 GEPB 实现，它们都是在 Layer 2 开展任务划分的，这样每个任务负责的 C 矩阵就是互相独立的，不会发生竞争读写。所以这里就在 Layer 2 对 m 进行划分，每个线程负责计算 $[m_{\text{begin}}, m_{\text{end}})$。

具体而言，代码就是

```c
struct gepb_parallel_task {
    size_t N, M, K;          // basic matrix sizes
    size_t M_start, M_end;   // range in Layer 2
    const float *lhs, *rhs;  // A, B
    float *C_packed;         // packed C
};

void* gepb_parallel_gemm_inner(void *arg) {
    struct gepb_parallel_task *task = (struct gepb_parallel_task*)arg;
    const size_t N = task->N;
    const size_t M = task->M;
    const size_t K = task->K;
    const size_t M_start = task->M_start;
    const size_t M_end = task->M_end;
    const float *lhs = task->lhs;
    const float *rhs = task->rhs;
    float *C_packed = task->C_packed;

    // each thread owns its `temp' and `A_packed'
```

```
    float temp[16 * 16];
    float *A_packed = aligned_alloc(1024, N * 16 * sizeof(float));

    for (size_t k0 = 0; k0 < K; k0 += 16) {
        for (size_t r = 0; r < N; r++)
            for (size_t k = 0; k < 16; k++)
                A_packed[r * 16 + k] = A(r, k0 + k);

        for (size_t c0 = M_start; c0 < M_end; c0 += 16) {
            for (size_t c = 0; c < 16; c++)
                for (size_t k = 0; k < 16; k++)
                    D(c, k) = B(k0 + k, c0 + c);

            float *ptr = C_packed + c0 * N;
            for (size_t r = 0; r < N; r++)
                for (size_t c = 0; c < 16; c++) {
                    float x = 0;  // C(r, c0 + c)
                    for (size_t k = 0; k < 16; k++)
                        x += A_packed[r * 16 + k] * D(c, k);
                    *ptr++ += x;
                }
        }
    }

    free(A_packed);
    return NULL;
}

void gepb_parallel_gemm(size_t N, size_t M, size_t K, const float *lhs, const float *rhs, float *dst) {
    float *C_packed = aligned_alloc(1024, N * M * sizeof(float));

    const size_t TASK = /* thread number */;
    struct gepb_parallel_task tasks[TASK];
    pthread_t threads[TASK];

    for (size_t i = 0; i < TASK; i++) {
        tasks[i] = (struct gepb_parallel_task){
            .N = N,
            .M = M,
            .K = K,
            .M_start = M * i / 16 / TASK * 16,
            .M_end = M * (i + 1) / 16 / TASK * 16,
            .lhs = lhs,
            .rhs = rhs,
            .C_packed = C_packed,
        };
    }
    tasks[TASK-1].M_end = M;

    // spawn (TASK - 1) threads
    for (size_t i = 0; i < TASK - 1; i++)
        pthread_create(&threads[i], NULL, gepb_parallel_gemm_inner, (void*)&tasks[i]);
    gepb_parallel_gemm_inner((void*)&tasks[TASK-1]);

    for (size_t i = 0; i < TASK - 1; i++)
        pthread_join(threads[i], NULL);

    float *ptr = C_packed;
```

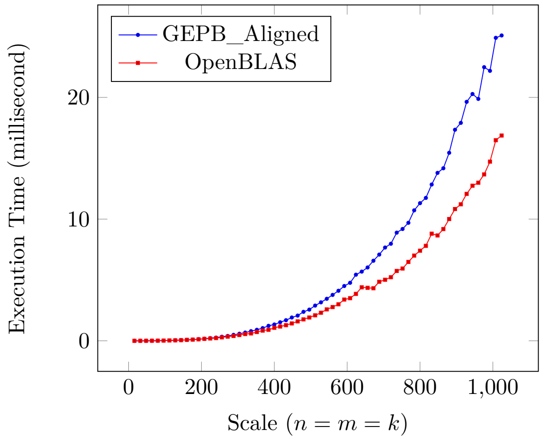
```
    for (size_t c0 = 0; c0 < M; c0 += 16)
        for (size_t r = 0; r < N; r++)
            for (size_t c = 0; c < 16; c++)
                C(r, c0 + c) = *ptr++;

    free(C_packed);
}
```

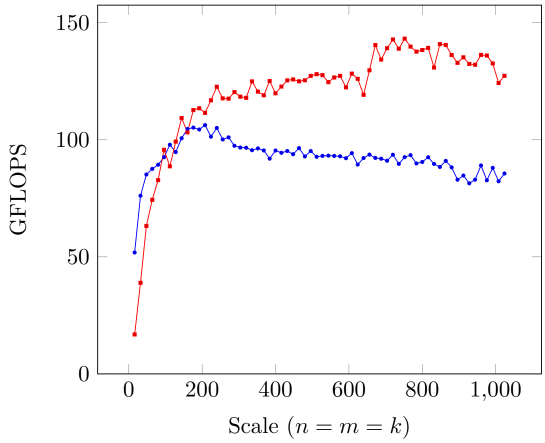
然后临时借用了一台有 16 核的设备，简单跑了一下，单线程和多线程的耗时与它们的比值如下：

Scale	1k	8k
Single Thread (μs)	48130	33411000
16 Threads (μs)	13950	3739851
Ratio	3.45	8.93

虽然这个比率谈不上十分理想，但是加速效果还是十分显著的。

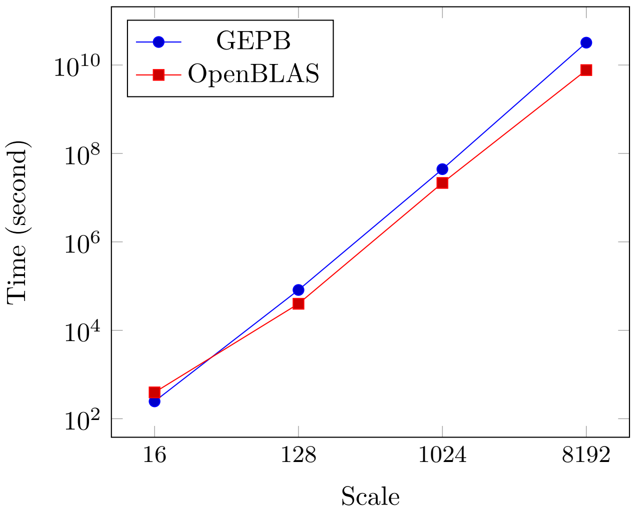
1.5 与 OpenBLAS 对比





可以看到，我们 GEPB 算法的效率能达到 OpenBLAS 的一大半。考虑到我们只写了几十行的代码，与 OpenBLAS 凭借开源的力量手写了每种 CPU 架构的汇编、还有一堆的优化相比，这个差距我觉得还是可以接受的。

然后对于 16、128、1k、8k 大小的矩阵（因为对于 64k，手上确实没有超过 64G 的电脑或者服务器），结果如下：

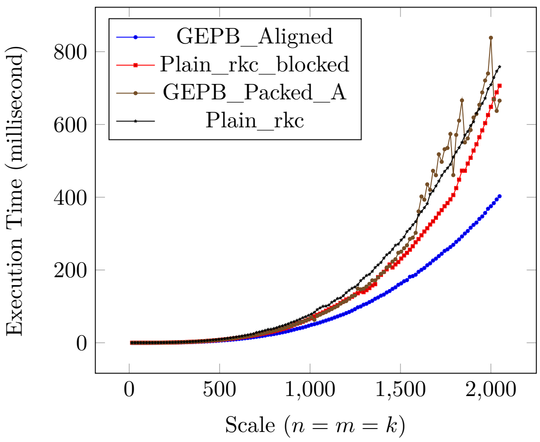


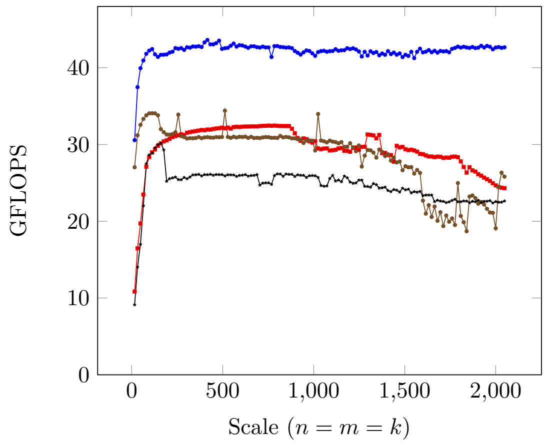
Scale	16	128	1k	8k
GEPB (μs)	247	81751	44113973	32035000000
OpenBLAS (μs)	395	39852	21509600	7673629780

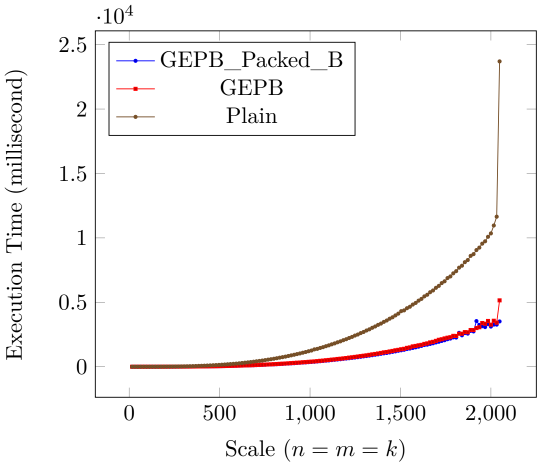
2 ARM

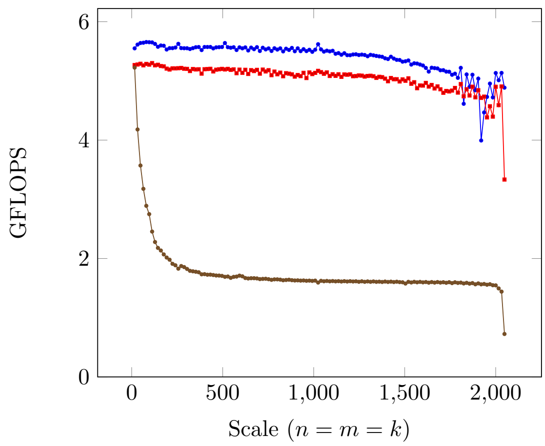
我们来看一下用 ARM 跑测试会是什么结果。

这里的测试就直接用了我手上的一台 Macbook M1，刚好就是 ARM 架构的。但是 M1 芯片是采用“统一内存”，就是内存和 CPU 都是集成在 SoC 中的，这与普通电脑中 CPU 与内存分开不同。这个区别在下面的性能测试中也会有所体现。









首先，这几种算法的相对速度和前面的差不多，这说明虽然不同架构之间虽然指令集不同，但是优化途径都是相似地——通过优化缓存局部性可以提高效率。

另外，可以看到的是，像 rkc、分块 rkc、GEPB（非最终版），它们并没有在 Intel 平台中出现的复杂的曲线。在这里它们都比较地平滑，特别是在 $n \le 1024$ 的时候，几乎看不到明显的波动。所以我把范围扩大到 [16, 2048]，才能在曲线的后面看到些许波动。

这个是因为 M1 的 L2 缓存有 4096 KiB 那么大，而且是每个核独享的。所以当矩阵大小小于 1024 时，它是可以把整个矩阵都装进去，速度就比较稳定。

到了后面 n 比较大的时候，才会出现些许波动，但是也没有像 Intel 平台那样出现大跳水。这个我认为与 SoC 的设计有关，因为 SoC 内集成了 CPU 和内存，使得 CPU 访问内存的延迟大幅缩小。而且从网上的评测来看，M1 笔记本的内存带宽非常恐怖。所以当 L2 miss 的时候，fallback 到内存也不会像 Intel 一样惨烈。

不过对于 GEPB_Packed_A 和 Plain_rkc，可以看到它们的 GFLOPS 在 $n = 128$ 的时候有所下降，估计是由于矩阵大小超过了 L1 缓存（64 KiB）所致的。

所以综上所述，在 ARM 平台，矩阵乘法的速度仍然会受到缓存的影响，提高缓存的局部性有利于提高效率。

3 总结

这个项目开始之前，我以为只要按部就班地弄一些更换循环顺序、循环展开、AVX512 什么的，就可以一步一步地优化下去。

但是实际操作下来，第一个更换循环顺序就稍微有点出戏，编译器对 rkc 的优化非常夸张，已经把我后面想用的循环展开和 AVX512 都用了，然后还跑得特别快。

虽然也可以不用 `-O3`，然后对比 `register` 还有 `i++` 和 `++i` 的区别。但是这样实际意义不大，首先 `register` 在 C++17 已经是 '*The keyword is unused and reserved*' 了，这说明 `register` 已经不推荐使用了。再者，在循环自增的 `i++` 和 `++i`，只要是个正常的编译器，都会生成出一样的指令。所以对比这两者的意义不大。更重要的是，在实际中，如果确实需要性能，没有人会傻乎乎地用 `-O0` 编译，而是会用 `-O3`。所以我就坚持在 `-O3` 的情况下进行优化。

然后在 `-O3` 之下，对于 rkc 这种逆天的顺序，除了能简单分块稍微优化一下之外，发现能做的事情特别少。因为缺少理论的支撑，不知道哪种分块、哪种顺序能提高效率，就像无头苍蝇一样。而且有由于编译器已经帮我们卷了 SIMD 和循环展开，走了我的路，我也很无奈。

不过好在研究了一下 OpenBLAS 之后，发现了 Goto 的 Anatomy of High-Performance Matrix Multiplication，就特别地妙。从论文里面学到了正经的分块方法 GEPB，并且加上矩阵重排之后，相比上面的 rkc 分块确实要好。

然后就基于这个 GEPB 做了多线程，虽然加速效果距离理论效果有点远，但是也确实有所优化。

最后与 OpenBLAS 进行对比，GEPB 能够达到 OpenBLAS 性能的 50% 以上，还是十分不错的。