HPC Homework1:Report

2023316026 张凯

Tsinghua University

zhang-k23@mails.tsinghua.edu.cn

1. 基础评测

首先，评测了作业中给出的初步代码文件，包括 naive 实现、block 实现和 blas 实现的结果，如图 1.1-1.3 所示：

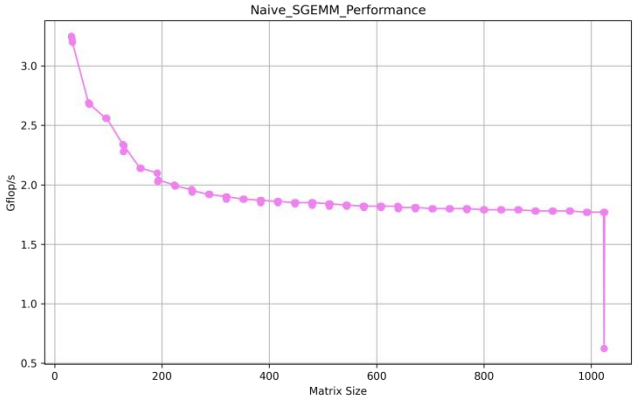

图 1.1 Naive 实现 SGEMM 实验结果

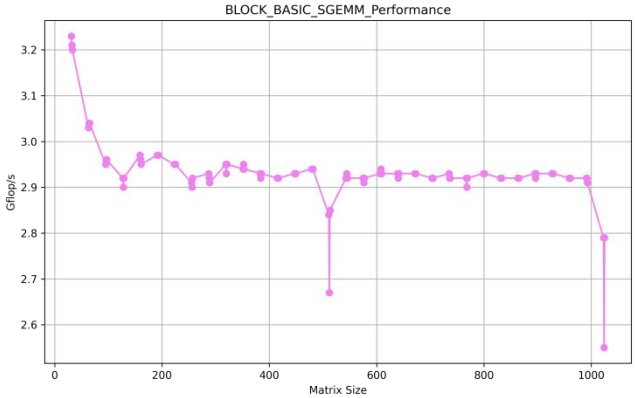

图 1.2 Block 基础实现 SGEMM 实验结果

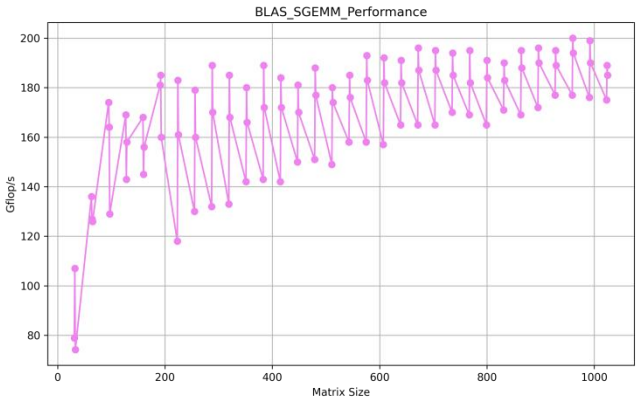

图 1.3 BLAS 实现 SGEMM 实验结果

由实验结果可知，BLAS 的矩阵乘法性能几乎是 Naive 版本的 200 倍，是巨大的性能提升。同时，作业中给出的基础 Block 版本性能相对 Naive 版本只有 1～2 倍的提升，相对 BLAS 性能还有巨大的优化空间，下面我们逐步进行优化。

2. 逐级优化策略及实验结果

作业中基础的 Block 版本代码保存为 sgemm-blocked-v0.c，下面进行逐级的优化，并比较和上一级的优化结果。

2.1 编译选项优化策略

首先进行编译选项 OPT 的加入，包括 O3，硬件架构适配、fast 等，如下：

```
OPT = -march=native -O3 -fomit-frame-pointer -mtune=native -fast
```

编译选项优化后的版本代码不变，保存为 sgemm-blocked-v1.c，运行对比实验结果如图 2.1 所示：

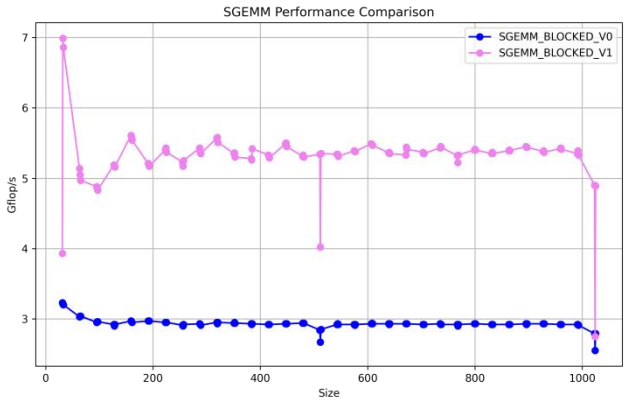

图 2.1 BLOCK V1 SGEMM 实验结果

仅仅加入编译选项优化后，程序性能即能够提升 2～3 倍。

2.2 分块大小更改+循环顺序调整

通过 lscpu 指令打印 CPU 信息：

虚拟化：	VT-x
L1d 缓存：	48K
L1i 缓存：	32K
L2 缓存：	1280K
L3 缓存：	18432K

考虑到 L1 Data Cache 大小为 48KB，64*64 的 float 类型矩阵大小为 16KB，正好可以容纳下 3 个矩阵 A，B，C，因此，更改分块大小为 64。

此外，为了更好地访存，更改原来外部以及分块内部 for 循环中 i，j，k 的循环顺序，代码保存为 sgemm-blocked-v2.c，运行对比实验结果如图 2.2 所示：

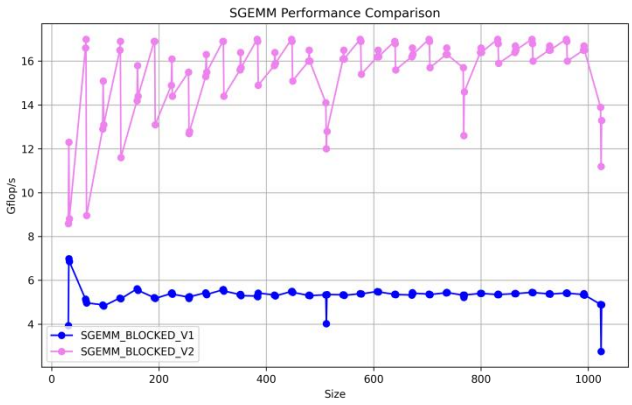

图 2.2 BLOCK V2 SGEMM 实验结果

性能得到了较高的提升，优化了 3 倍左右，继续进行优化。

2.3 二级分块策略+循环展开

对 64*64 分块内部进一步做了 16*16 的分块，所有的矩阵乘法都分块为 16*16 的矩阵乘法，即 Ci = Ai * Bi + Ci 中，每个矩阵都是 16*16 的大小，这是为了后续 AVX512 向量化指令优化，将 do_block()修改，做了循环展开等细微的优化。代码保存为 sgemm-blocked-v3.c，运行对比实验结果如图 2.3 所示：

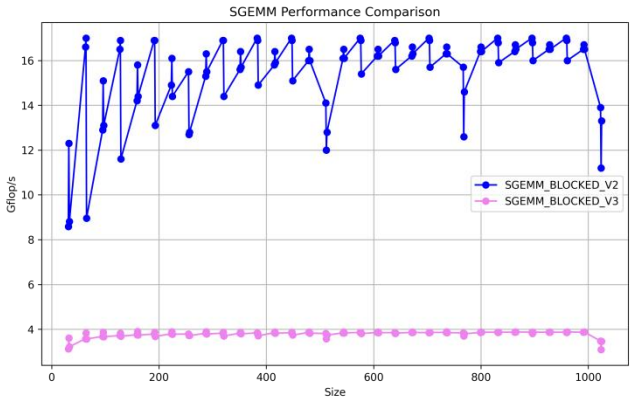

图 2.3 BLOCK V3 SGEMM 实验结果

如果只是简单的分块之后，性能下降了很多，下面继续优化。

2.4 AVX512 指令向量化

在 2.3 中，二级分块后，性能变差了，现在，我们通过将 16*16 的分块全部做向量化指令实现，提升系统的性能。

对于所有的16*16分块, C分块的每一列都是16维向量, 可以正好用AVX512的一个寄存器存储, C的每一列向量 Ci 相当于 Bi 向量的每个元素*向量 Ai 求和, 示意图如下：

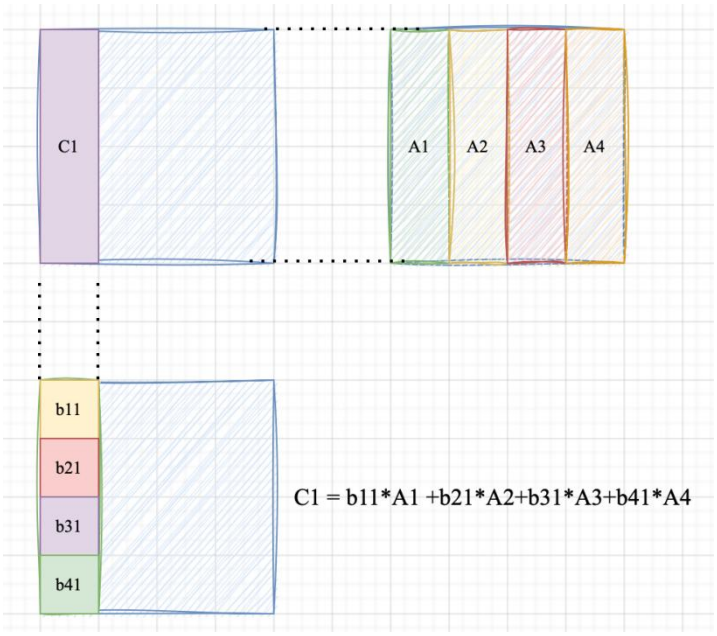

示意图 1 分块向量化示意图

向量化的代码是非常简洁易懂的，对应上图：

```
static void do_block_avx(int lda, float *A, float *B, float *C) {
    for(int j = 0; j < 16; ++j){
```

```
        __m512 c = _mm512_loadu_ps(&C[lda * j]); // C = (c0,c1,...,c15)
        for(int k = 0; k < 16; k++){
          __m512 a = _mm512_loadu_ps(&A[lda * k]);
          __m512 b = _mm512_set1_ps(B[k + lda * j]);
          c = _mm512_fmadd_ps(a, b, c);
        }
        _mm512_storeu_ps(&C[lda * j], c);
}
}
```

对于其他不规则的形状，依然用旧的 do_block 函数处理。代码保存为 sgemm-blocked-v4.c，运行对比实验结果如图 2.4 所示：

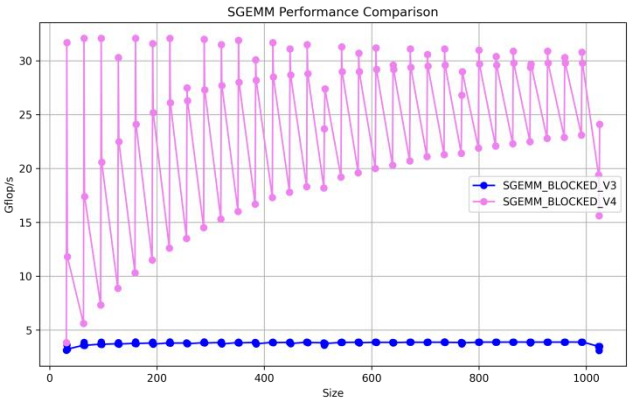

图 2.4 BLOCK V4 SGEMM 实验结果

性能有了明显的提升，但是可以看到，对于不是 16 倍数的矩阵情形，性能明显不够好，性能波动幅度很大。

2.5 AVX512 矩阵核函数循环展开

按步长为 4，循环展开 AVX 向量化指令的矩阵核函数，由于我采用了手动的展开，代码长度很长，这里不再展示。对于其他不规则的形状，依然用旧的 do_block 函数处理。代码保存为 sgemm-blocked-v5.c，运行对比实验结果如图 2.5 所示：

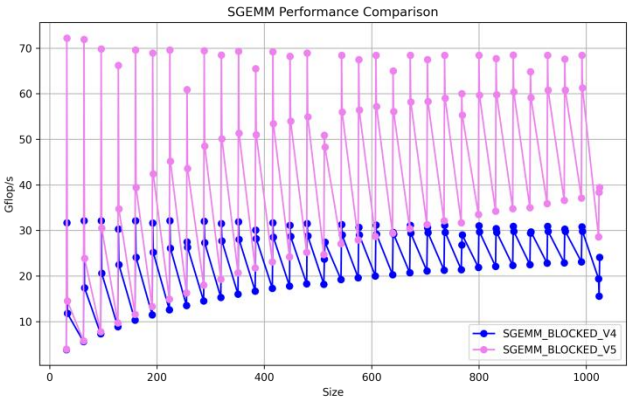

图 2.5 BLOCK V5 SGEMM 实验结果

性能有了进一步的明显提升，峰值性能已经达到 70Gflop/s 左右，但是对于非 16 倍数的情况，性能仍然较差，这主要是由于边界情况需要额外处理导致的，但是，我们先忽略非峰值情况，进一步提升峰值性能。

2.6 AVX512 矩阵核函数修改为非方阵核函数

可以发现，二级分块后，仍然进行了三次 for 循环，但是对于 16*16 的 Ci 分块矩阵来说，其本质是由形状为 16*K 的 Ai 矩阵与形状为 K*16 的 Bi 矩阵相乘得到的，而之前全部都进一步拆分成 16*16 的方阵处理，这样增加了循环数量，且引入了更多的非整数倍边界情况，损失了性能。

因此，我去除了外层的 for 循环中的 k 循环部分，将 do_block_avx 函数进一步优化为 do_block_avx_16k16 的函数，代码保存为 sgemm-blocked-v6.c，运行对比实验结果如图 2.6 所示：

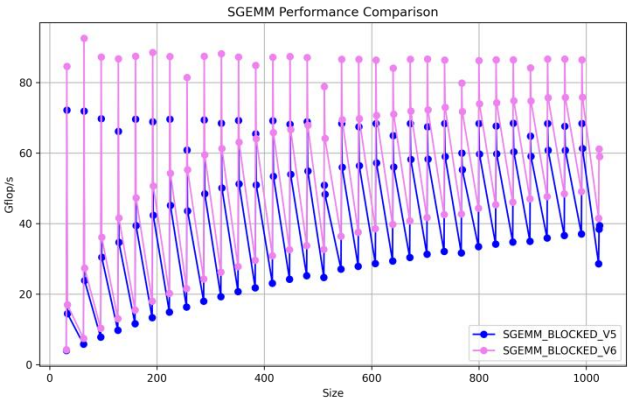

图 2.6 BLOCK V6 SGEMM 实验结果

程序性能进一步提升，峰值性能达到了 90Gflop/s 左右，下面，在进一步提升峰值性能之前，我们将着重解决程序性能不稳定的问题。

2.7 边界 Packing 处理

对于程序性能不稳定的情况，分析主要原因是非整数情况边界，采用了平凡的 do_block 函数处理，这样会大大降低程序的性能。因此，希望将边界进行补齐处理，这样可以全部由 AVX512 向量化指令处理，会加快程序运行的速度。

补齐的方式有很多种，我尝试了下面几种：

(1) 一开始就对整个矩阵全部补齐为 16*16 的整数倍情形，这样效率很低，且矩阵较大，需要提前处理的数据越多，会增加很多不必要的损失。

(2) 循环时遇到小分块不是 16*16 情形，都补齐为 16*16 情形，这样效率也较低，会做多次重复的补齐。

(3) 对 16*K 的 A 矩阵和 K*16 的 B 矩阵做补齐，K 的长度和 16 倍数无关，只需要检测 A 分块的行及 B 分块的列是否满足 16，若小于 16，进行补齐处理，同时对于所有的数据，仅做一次补齐，避免重复操作。

最终，我参考：https://github.com/jiegec/sgemm-optimize/tree/master/sgemm 实现了补齐的策略，这个策略是非常巧妙的，对 A 分块的行做了 lda =16 的列主序蛇形存储，保证了访存的连续性，示意图如下：

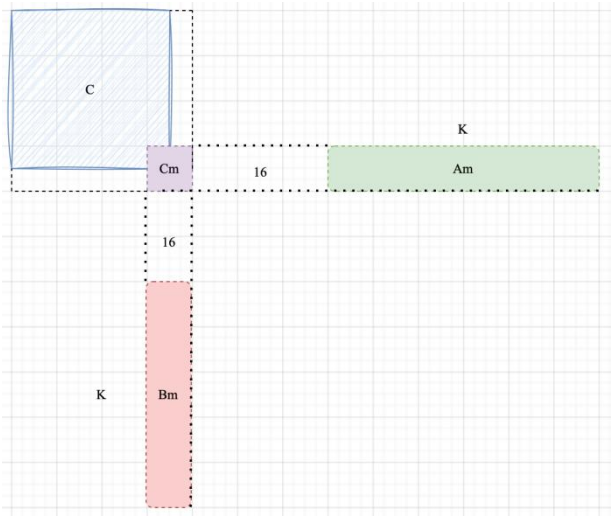

示意图 2 Packing 示意图

即对于矩阵不是 SMALL_BLOCK_SIZE=16 倍数的情形，比如 63<64，会将 A，B，C 的“缺失”部分都做对应的补齐，同时确保每次只补齐一次。代码保存为 sgemm-blocked-v7.c，运行对比实验结果如图 2.7 所示：

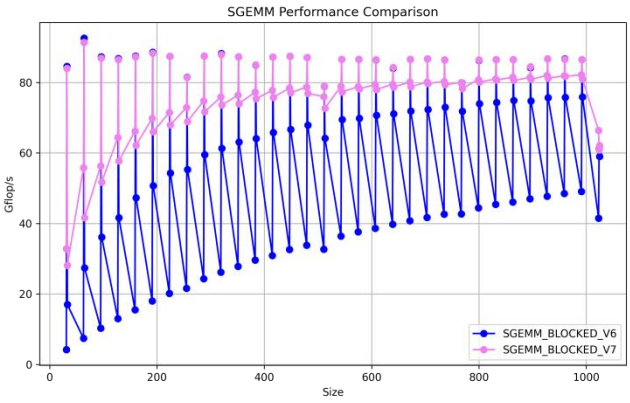

图 2.7 BLOCK V7 SGEMM 实验结果

可以看到，峰值性能基本没有提升，但是非幂次的矩阵乘法提升了很多，除了少数数据点仍然较差以外，基本解决了程序性能不稳定的问题。

2.8 矩阵 Packing 优化+一级分块循环去掉 k

上述 2.7 中的 Packing 依然存在比较多的冗余，我是判断了如果不是 16 的倍数，那么将整个矩阵元素都需要 Packing 到整数倍的大小，这显然会增加一些时间开销。

因此，修改 Pack 代码，仅仅对边界情况进行 Packing，即 Packing 的 AA 矩阵为 SMALL_BLOCK_SIZE * K，Pack 的 BB 矩阵为 K*SMALL_BLOCK_SIZE，仅 Packing 一次 AA 和 BB，也就是示意图 2 中的绿色矩阵 AA 和红色矩阵 BB。

同时，由于现在可以处理任意 K 的 A，B 矩阵，我们将一级分块直接去掉 k 的循环，这样会提升部分峰值性能。代码保存为 sgemm-blocked-v8.c，运行对比实验结果如图 2.8 所示：

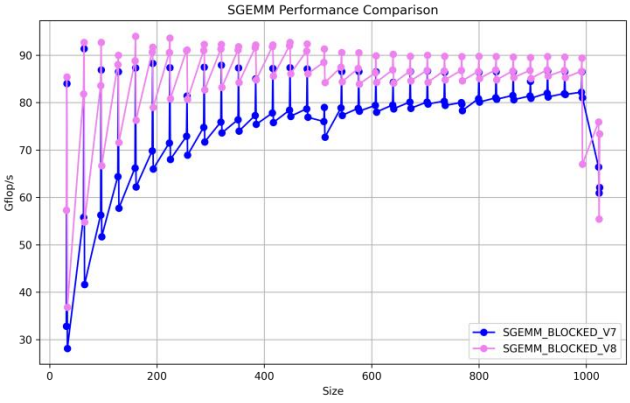

图 2.8 BLOCK V8 SGEMM 实验结果

可以看到，无论是峰值性能还是其他非整数倍的情形，都有了一定的提升，

程序性能更加稳健了。但是，现在最重要的问题就是峰值性能相比 mkl 还有很大的提升空间。已经基本搞定了 Packing 的优化，现在想要提升峰值性能，我们需要从整数倍的简单情形入手，进一步大幅度提升程序性能。

2.9 Kernel 乘法优化

非常明确的是，如果仅仅考虑峰值情形，那就是二级分块 Kernel 维度整数倍的最简单情形，那么决定程序运行时间的关键就在于 Kernel 矩阵乘法的速度。之前已经采用了向量化指令的方法，看上去已经比较完美了，但是之前的实验结果告诉我们，这里一定还有巨大的优化空间，必须要从这里入手进一步优化。

因此，我进行了反汇编，阅读了程序的汇编代码。之前的程序中，采用了 16*16 的 C 分块，这样 B 中的每个分量只用了一次，而 A 的访存不够连续，因此，如果希望能够进一步降低访存时间，同时充分利用向量化指令，应当增大二级分块的列长度，所以我尝试了对 C 的新分块，包括 32*8 分块、64*4 分块等，发现 32*8 分块后性能明显提升，代码保存为 sgemm-blocked-v9.c，运行对比实验结果如图 2.9 所示：

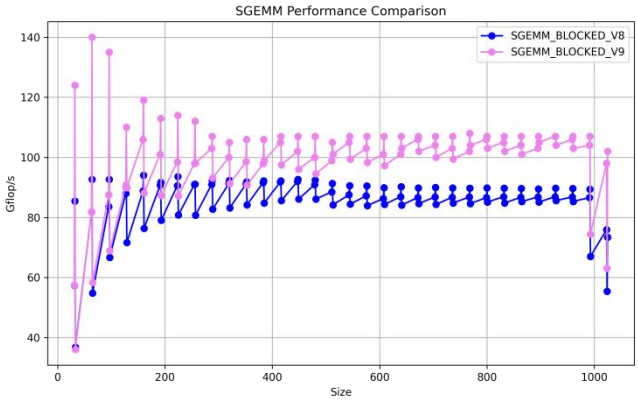

图 2.9 BLOCK V9 SGEMM 实验结果

无论是峰值性能还是其他非整数倍的情形，都有了非常大的提升，已经快要接近 mkl 性能的 80%了，目标近在眼前，继续进行优化和尝试。

2.10 局部性优化

针对 M，N，K 三个维度，增加了不同的分块长度变量，进行参数的微调。发现重新在一级分块中加入 k 维度，可以提升程序的性能，这是由于如果 K 过大，程序内存局部性会变差，而加入 k 维度后，会提升内存的局部性。代码保存为 sgemm-blocked-v10.c，运行对比实验结果如图 2.10 所示：

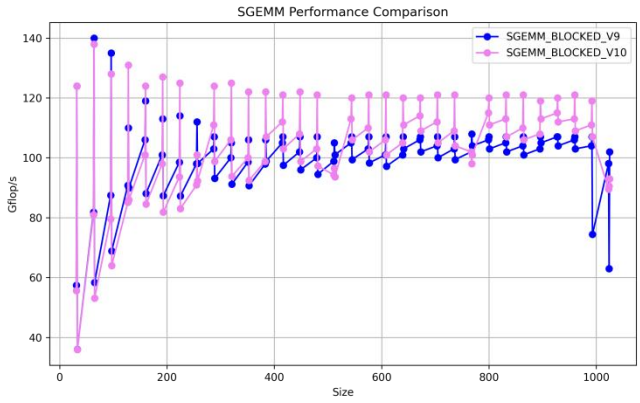

图 2.10 BLOCK V10 SGEMM 实验结果

可以看到，重新调整一级分块参数后，对于矩阵 Size 较大的情形，程序性能有了明显的提升，但是仍然不够稳定，继续进行优化和尝试。

2.11 边界 Packing 进一步优化

整数倍的峰值性能依然很难优化，因此，为了不浪费时间，先进一步优化非整数的 Packing 策略，提升普通数据点情形，提高程序性能稳定性。针对 32*8 的分块做了进一步的 Packing 优化调整，代码保存为 sgemm-blocked-v11.c，运行对比实验结果如图 2.11 所示：

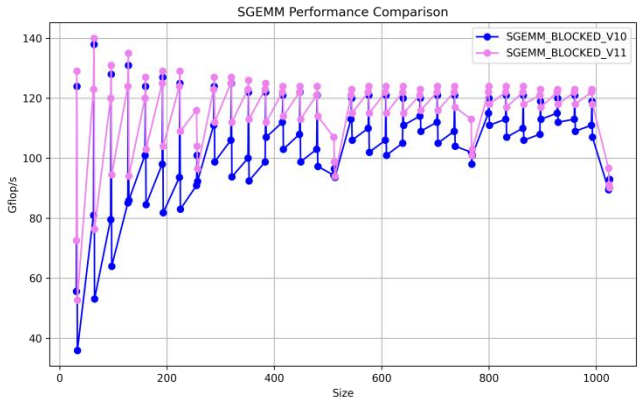

图 2.11 BLOCK V11 SGEMM 实验结果

程序性能稳定性得到了进一步提升，下面将着重于优化峰值性能。

2.12 一级分块 Packing A

发现之前一直缺失的部分是一级分块中没有考虑对矩阵做 Packing 处理，当时主要顾虑是觉得提前 Packing 一级分块大矩阵会牺牲一定的性能。但是，为了进一步提高数据的局部性，进行 Packing 处理，首先对分块后的 A 进行 Packing，参考了如下的部分实现：

https://github.com/yzhaiustc/Optimizing-DGEMM-on-Intel-CPUs-with-AVX512F/tree/master

代码保存为 sgemm-blocked-v12.c，运行对比实验结果如图 2.12 所示：

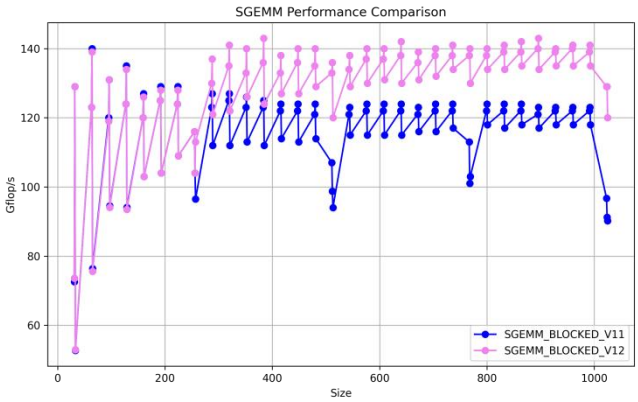

图 2.12 BLOCK V12 SGEMM 实验结果

程序性能得到了不错的提升，可见 Packing 策略是非常有效的，尤其是对于矩阵 Size 较大的情形，下面将继续对 B 矩阵分块进行 Packing 处理。

2.13 一级分块 Packing B + 64 * 4 Kernel

对 B 矩阵分块进行 Packing 处理，并增加了 64 * 4 的 Kernel，代码保存为 sgemm-blocked-v13.c，运行对比实验结果如图 2.13 所示：

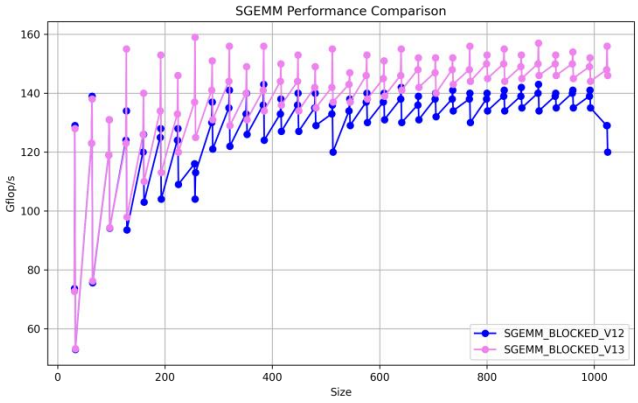

图 2.13 BLOCK V13 SGEMM 实验结果

程序性能进一步提升了，胜利近在眼前，我们继续提升，微调分块参数。

2.14 分块参数微调

微调增大一级分块大小 K=256，代码保存为 sgemm-blocked-v14.c，运行对比实验结果如图 2.14 所示：

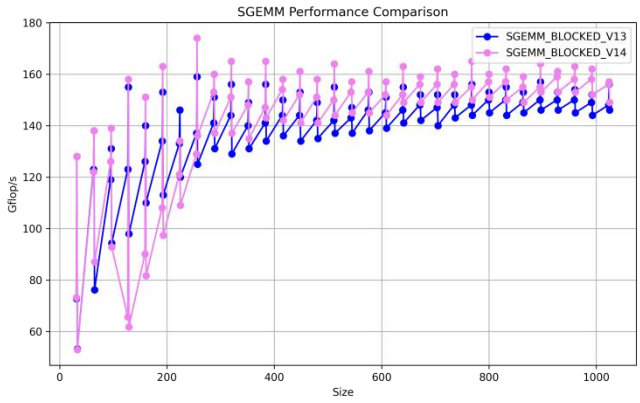

图 2.14 BLOCK V14 SGEMM 实验结果

程序性能进一步提升了，但是稳定性又变差了，需要进一步处理。

2.15 分块微调+参数微调+边界优化

进一步细化分块，在一级分块后将分块同时按 64*4 及 32*8 进行二级分块，即允许分块中存在两种不同的 Kernel 计算。同时微调参数，针对不同矩阵大小区间做了处理，并适当增大 N 的值。针对分块做了边界的补齐优化，代码保存为 sgemm-blocked-v15.c，运行对比实验结果如图 2.15 所示：

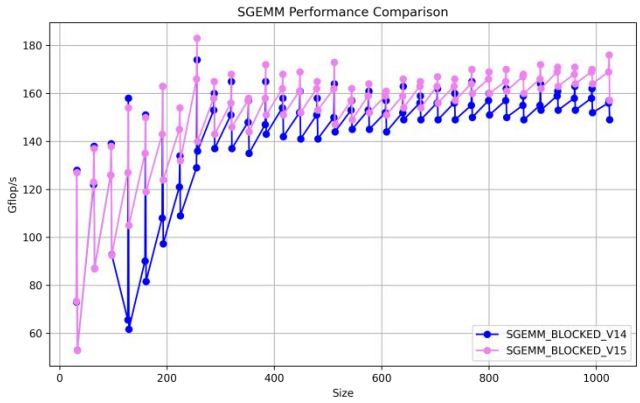

图 2.15 BLOCK V15 SGEMM 实验结果

程序性能进一步提升了，性能稳定性也得到了提升。

3. 最优性能分析

最终，程序代码经过整理和删改，保存为 sgemm-blocked.c 文件。我比较了程序和 BLAS 库的性能，运行对比实验结果如图 3.1 和 3.2 所示：

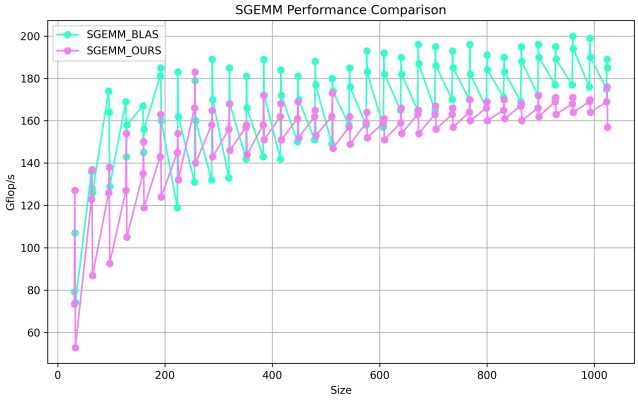

图 3.1 SGEMM 实验结果:OURS VS BLAS

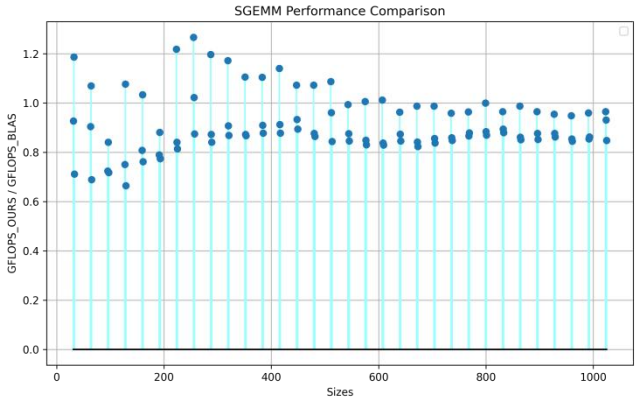

图 3.2 SGEMM 实验结果 RATIO:OURS / BLAS

从实验结果可以看到，我们的程序已经非常接近 BLAS 的性能，只有少数点性能低于 80% BLAS 基线，大部分性能都高于 80% BLAS 基线，甚至很多情况下高于 BLAS 的性能，充分证明了程序实现的稳健性。

参考文献

[1] https://github.com/jiegec/sgemm-optimize/tree/master/sgemm

[2] https://github.com/yzhaiustc/Optimizing-DGEMM-on-Intel-CPUs-with-AVX512F/tree/master

[3] https://siboehm.com/articles/22/CUDA-MMM

[4] https://siboehm.com/articles/22/Fast-MMM-on-CPU