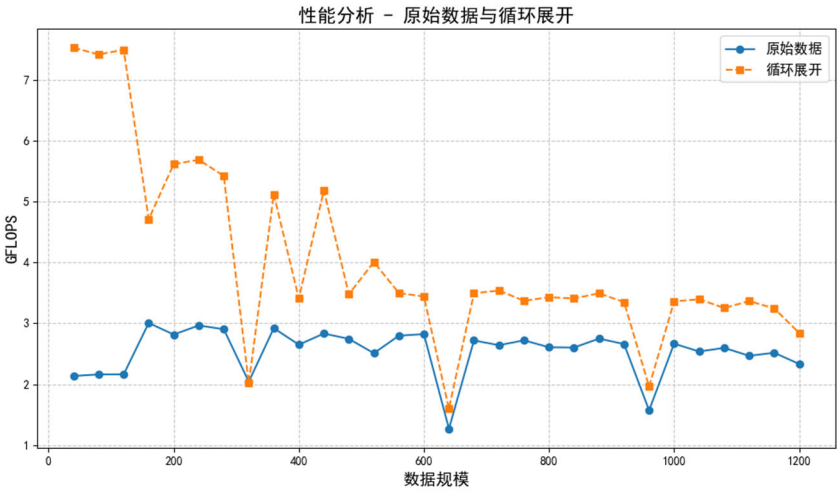
针对矩阵乘法的优化方案

首先，矩阵乘法代码来源：https://github.com/flame/how-to-optimize-gemm/blob/master/src/HowToOptimizeGemm.tar.gz，优化基于解压后的 MMult1.c

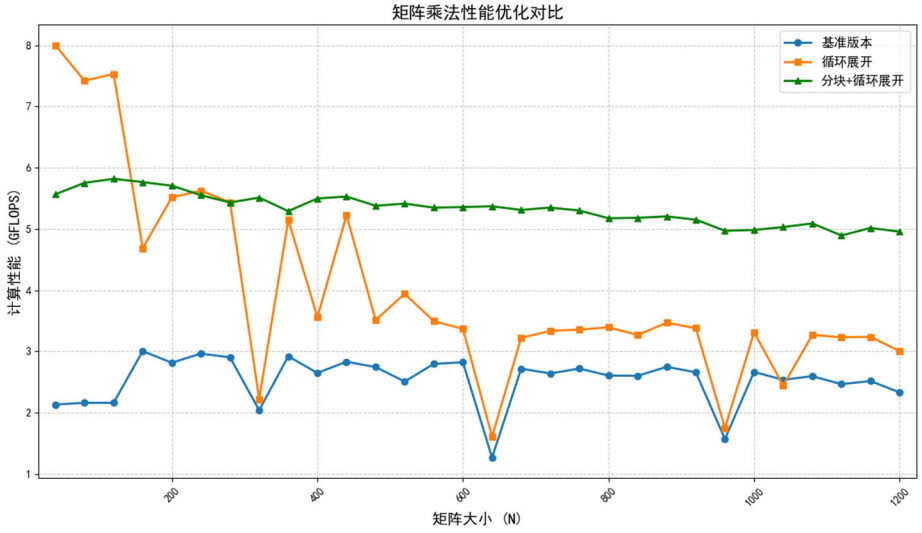
通过 vtune 工具查看程序分析后发现，AddDot 函数花费了 62.4%的 CPU 时间，所以着手优化该函数。

首先想到了循环展开的优化方法（MMult2），通过将主循环由一次处理一组数据提升到一次处理四组数据，来提升性能，同时添加了局部变量 temp，降低了内存访问。下面附上性能对比图：



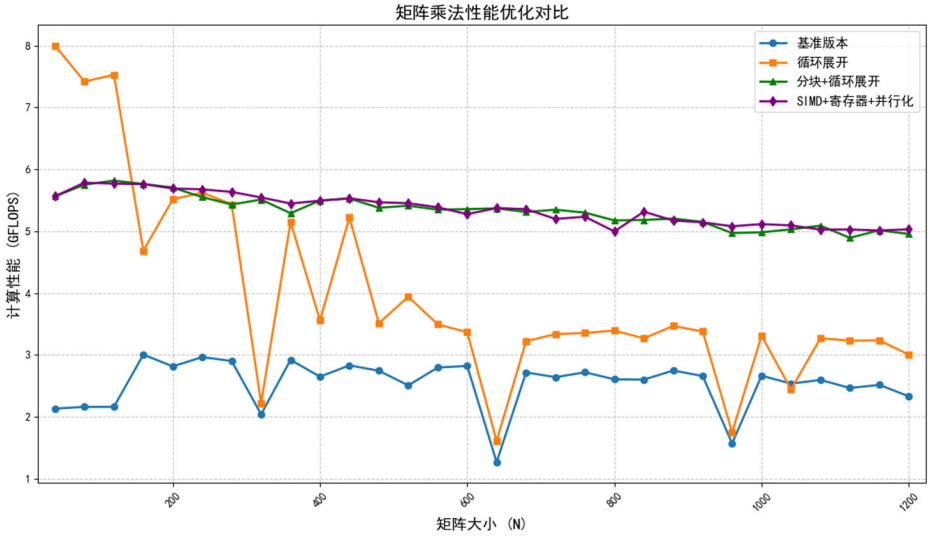
图表显示在数据规模为 320、640、960 时，性能大幅下降，不难发现这些数都是 320 的倍数，由于数据都是 double 类型的，所以字节数都是 2560 的倍数，那么我们假设 CPUL1 的缓存是 32KB，缓存行为 64 字节，缓存组相连度为 8 路，那么每个缓存组可以存储 64 个缓存行，而 2560 字节正好是缓存行大小的 40 倍，可能会导致缓存抖动和缓存行冲突，所以通过专门的测试程序 test_320 来测试 320 附近的矩阵大小和性能，结果显示在矩阵大小为 319 和 321 时性能是正常水平，而矩阵大小是 320 时性能显著下降约 38%，这证实了我们的猜测，所以有了 MMult3 优化。

MMult3 所采用的优化方案为将块的大小更改为 32x32，这样块内访问的会是连续的 32 个元素，并且块与块之间的跳转不再是 320 的倍数，以此来优化性能，同时使用了二维分块，在 32x32 的块内部再继续细分为 8x8 的小块以充分利用缓存，虽然最高性能有所下降，但总性能依旧是上升的，并且性能释放更加平稳，没有出现当数据量大时性能大幅下降的情况，下面附上性能对比图：

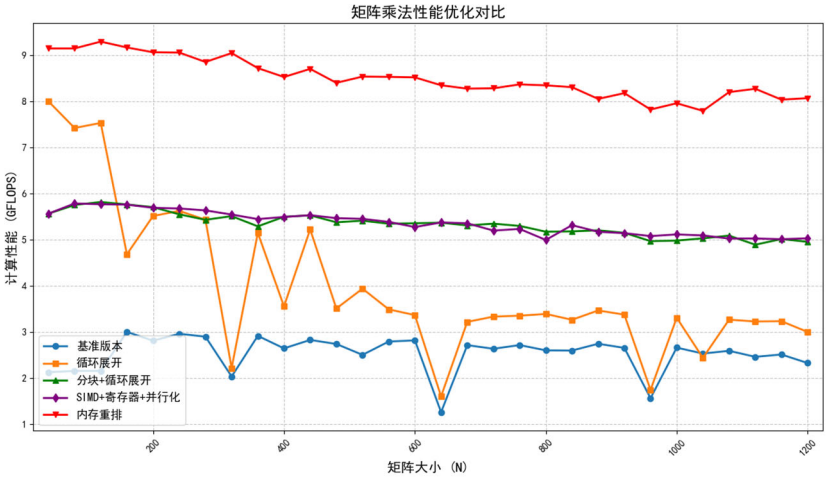


接下来对 MMult3 进行 vtune 分析，其中 do_block 函数花费了 59.4%的 CPU 时间，所以对该函数进行优化。

MMult4 使用了 SIMD+寄存器+并行化的方式进行优化，首先通过 SIMD 进行向量化，可以通过并行计算同时处理多个操作，其次，寄存器可以减少内存访问以增加性能，不过性能并未有太大的提升，下面是性能对比图：



MMult5 使用了内存重排技术进行优化，其可以提高缓存命中率，将原先不连续的存储连续起来，增加了预取元素的效率，同时减少内存带宽压力，一次加载多个元素以更好的利用内存带宽，以及内存对齐，增加了内存访问的效率，这次优化大大提升了性能，下面附上性能对比图：



至此，针对矩阵乘法的优化告一段落。