# **“探究层内不同overlap方法的最佳使用范式”**

**分析报告**

1. **测试环境**

由于实际GPU数量和性能的限制，整体实验选择在AutoDL云服务器上进行，选用的GPU为2卡A100 PCle(40GB)，能够支持GPU间P2P通信，其驱动为570.124.04，相应的架构为sm80，CPU为Intel Xeon Processor (Skylake, IBRS)，操作系统为Ubuntu 22.04。

1. **基于融合的方式：Flux**
2. **原理简介**

Flux使用CUTLASS生成高质量的GEMM，并通过NVSHMEM实现了底层计算通信算子的重写，相较以往方法具有更高的计算通信加速，可以适用于各类大模型的训练与推理中。具体来说，Flux对ReduceScatter和AllGather通信原语都进行了重写，使其能和GEMM计算相融合，实现最高96%的重叠效果。

对于ReduceScatter，Flux将其融合到了GEMM的尾部来实现重叠。由于ReduceScatter可以进一步解耦为AlltoAll+Local Reduce，且只有AlltoAll需要设备间通信，因此更确切地说，Flux是将AlltoAll的写分支融合到了GEMM尾部。此算法需要GPU间支持P2P通信才可正常运行。

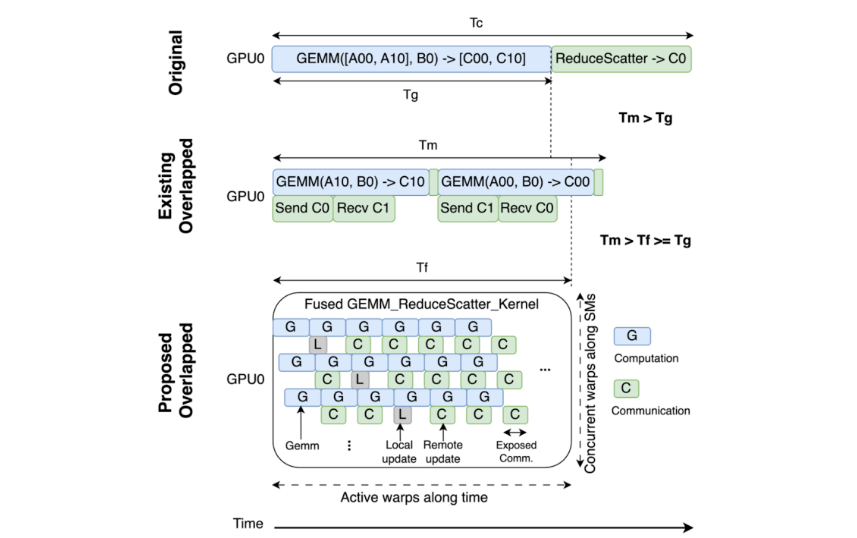


图1 ReduceScatter重叠技术对比

现有的重叠策略能够通过切分GEMM内核为一系列更小的内核来减少计算通信总时间Tm，但受限于GPU利用率不足，仍逊色于原始GEMM计算时间Tg。而Flux通过上下文切换和数百个在SM间并发活跃的Warp来隐藏通信延迟，能在不影响GEMM计算效率的前提下，实现Tf与原始GEMM计算Tg相当的性能，仅在末尾引入少量通信开销。

对于AllGather，Flux将其融合到了GEMM的头部来实现重叠，更具体地说是将AllGather的信号检查功能融合到了GEMM的前序阶段。由于只有通信的等待逻辑被融合到了GEMM内核中，而非整个通信操作，因此此算法并不依赖P2P通信。在内核端，GEMM计算被WaitSignal函数阻塞，直至主机端该通信分块准备就绪时其信号值才设置为True。主机端则通过pull和push两种方式执行相应的通信过程。

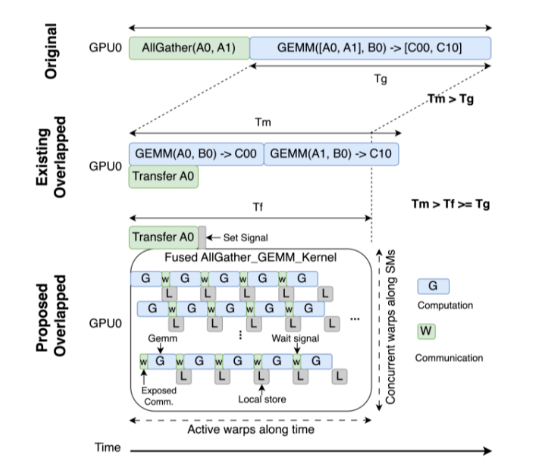


图2 AllGather重叠技术对比

AllGather中的长延时来自于等待指令，此现象出现于每个Warp的开端。由于Flux把信号检查功能融合到了Kernel中，其延时就取决于相应数据传输的到达时间。对于数据已抵达的分块，其延时几乎为0，对于数据未就绪的分块，Warp的上下文切换能掩盖其等待延时。最终，Flux在不影响GEMM计算效率的前提下，仅在开头引入少量通信开销。

1. **测试结果**

为确保一致性，我们选用GEMM+ReduceScatter作为测试场景，输入矩阵的M、N、K分别为4096、12288和49152，选用的数据精度为FP16。为了减少误差，确保训练的稳定性，我们进行10轮迭代，并取其平均值作为测试结果，相应的测试指令为：./launch.sh test/python/gemm\_rs/test\_gemm\_rs.py 4096 12288 49152 --dtype=float16 --iters=10。

测试结果如下：

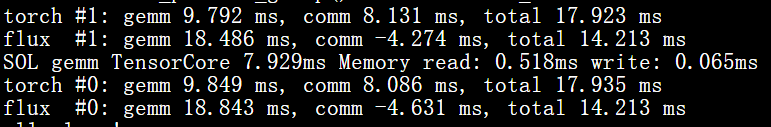


图3 Flux测试结果

由于此处输出的gemm和comm并不单纯表示计算时间和通信时间，因此我们只关注总时间total。可以看到，相比Pytorch + NCCL的非重叠计算通信，Flux在两块GPU上都实现了加速，加速比为1.2615。

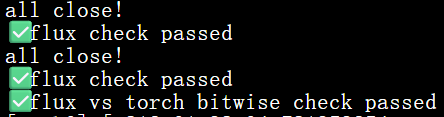


图4 Flux正确性评估

在正确性评估中，Flux在两块GPU上都通过了位运算检查，表明其结果和非重叠计算通信完全一致，进一步证明了其有效性。

1. **优劣势及使用场景分析**

* 以Flux为代表的基于融合的计算通信重叠方法优势为：

1. 在内核中可以实现以分块为粒度的细粒度重叠，并减少不必要的

GEMM内核开销；

1. 可以适配sm80、sm89和sm90架构的GPU，且在PCle设备上的加速

更优，能有效减少全局内存的写回和读取；

1. 支持dense和MoE等多种模型，可以根据不同任务选用不同策略；

* 以Flux为代表的基于融合的计算通信重叠方法劣势为：

1. 该类方法大多需要结合NVSHMEM来对计算和通信算子进行重写，

对现有工作的复用率较低，开发难度大；

1. 针对不同的通信原语需要专门进行开发和调优；

基于融合的计算通信重叠方法适用于计算和通信耦合度高、延迟敏感的场景，通常在大规模和高性能的集群中表现更优异，例如大规模的Transformer推理、高性能计算中的算子通信等。

1. **基于分解的方式：TransformerEngine**
2. **原理简介**

基于分解的计算通信重叠方法会把运算的张量分为多个部分，对应的计算和通信操作也可以相应进行分解，通过重排这些分解后的操作，可以有效解决数据依赖问题。为了避免通信过程中直接占用或修改计算线程正在使用的主数据而导致的数据竞争，该类方法通常会把发送和接收的分块数据拷贝到一个单独的用户缓冲区中，来确保通信操作访问的数据在整个传输过程中保持稳定。

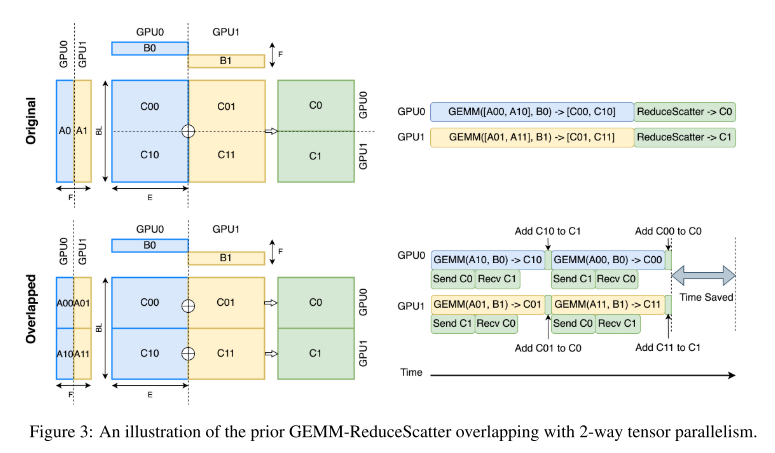


图5 基于分解的计算通信重叠方案

图5将A矩阵进行了进一步的划分，由此GEMM就会分为两部分，GPU0需要GPU0上计算的C00和GPU1上计算的C01，这样GPU0在计算C10时就可以同步得到GPU1上的C1来结合C10，GPU1同理。第一步完成后，GPU0继续计算C00，并将带有C10的C1发送给GPU1，同时接收带有C01计算结果的C0，和当前计算的C00相加得到最终的C0，GPU1同理。

TransformerEngine为NVIDIA提供的专门使用FP8精度的库，通过使用TransformerEngine模块搭建模型，并用fp8\_autocast装饰前向计算过程，可以在任何框架中使用FP8。相比当前大模型训练广泛采用FP16+BF16混合精度（AMP），FP8能在控制精度误差的情况下，实现更快的计算速度和更少的资源占用，从而降低训练通信量和提高吞吐量。

在FP8计算中，输入的两个矩阵可以是FP8两种精度的任意组合，两个FP8矩阵在完成一次Tensor Core运算后会输出FP16/FP32的高精度结果，因此矩阵计算过程中存在高低精度的转换。

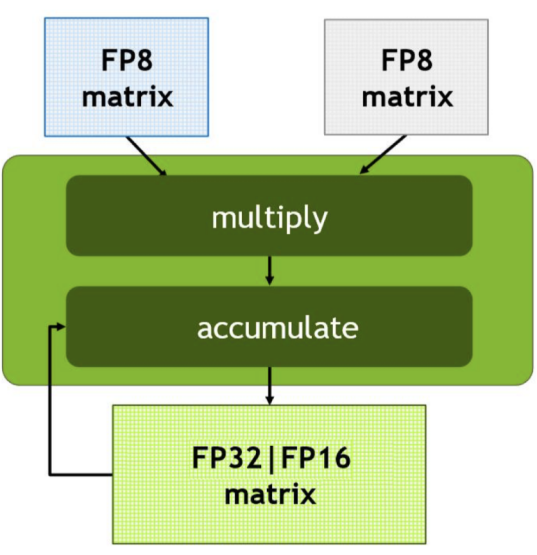


图6 FP8计算过程

1. **测试结果**

由于TransformerEngine没有提供直接的GEMM+ReduceScatter测试脚本，为了确保一致性，我们将example中te\_layer\_with\_overlap.py里的所有layer都指定为线性层Linear，来模拟分布式训练中通过用户缓冲区实现的GEMM+ReduceScatter。我们使用FP32+BF16混合数据精度，并设置输入的批次大小b为2，序列长度s为2048，注意力头数量n为96，注意力头维度d为128，来模拟相同的输入矩阵维度M和N，即4096和12288。此外，我们还在原先的te\_layer\_with\_overlap.py中添加了计时逻辑，同样进行10轮迭代取其平均执行时间作为测试结果，相应的测试指令为：torchrun --nproc\_per\_node=2 te\_layer\_with\_overlap.py和torchrun --nproc\_per\_node=2 te\_layer\_with\_overlap.py --no-comm-overlap。

测试结果如下：

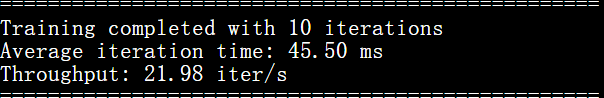


图7 TransformerEngine非重叠测试结果

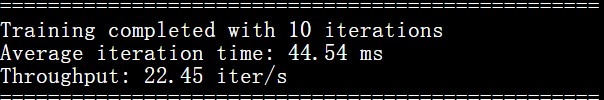


图8 TransformerEngine重叠测试结果

图7为不使用用户缓冲区时Pytorch + NCCL的计算通信时间，图8为使用用户缓冲区后Pytorch + NCCL的计算通信时间。可以看到，使用用户缓冲区后每轮迭代的时间都有缩短，加速比为1.0216。

1. **优劣势及使用场景分析**

* 以TransformerEngine为代表的基于分解的计算通信重叠方法优势为：

1. 支持多种数据精度，并提供了高度优化的一体化算子，可以提升计算

效率；

1. 提供了丰富且易用的多框架API，可轻松集成到现有LLM框架中；
2. 可以解除计算与通信间的数据依赖，实现较好的重叠效果；

* 以TransformerEngine为代表的基于分解的计算通信重叠方法劣势为：

1. 该类方法性能严重依赖于分块的执行顺序和执行时间，在涉及许多流

和事件的实际生产环境中会变得不可预测；

1. 将较大的内核拆分为一系列较小的内核并行执行会导致GPU流式多

处理器利用率不足，增加计算耗时；

1. 为保证切分出的矩阵乘法的连续性，其切分维度往往受限，无法支持

以分块为粒度的细粒度重叠；

基于分解的计算通信重叠方法适用于数据天然可分块、用户缓冲区可直接访问的场景，由于该方法对底层硬件和框架没有强依赖，因此常见于分布式训练和多节点的数值模拟中。

1. **基于信号的方式：FlashOverlap**
2. **原理简介**

FlashOverlap同样使用CUTLASS生成高质量的GEMM，并使用NCCL进行相关通信。相较于传统基于分解或融合的计算通信重叠方案，FlashOverlap提出了基于控制信号的计算通信重叠方案，能够实现低侵入矩阵乘法和无侵入通信，对于通信瓶颈显著的GPU来说最高可以达到1.65倍的提升。



图9 不同计算通信重叠方法对比

具体来说，FlashOverlap使用两个Stream，一个Stream用于计算，另一个Stream用于通信。在第一个Stream的计算内核中，FlashOverlap添加了Flag信号，当一部分计算结果就绪后，对应的Flag会被设置为True。同时，另一个Stream里被插入了一个用于等待阻塞的内核，当该内核读到Flag为True时会退出，并调用随后的通信算子。

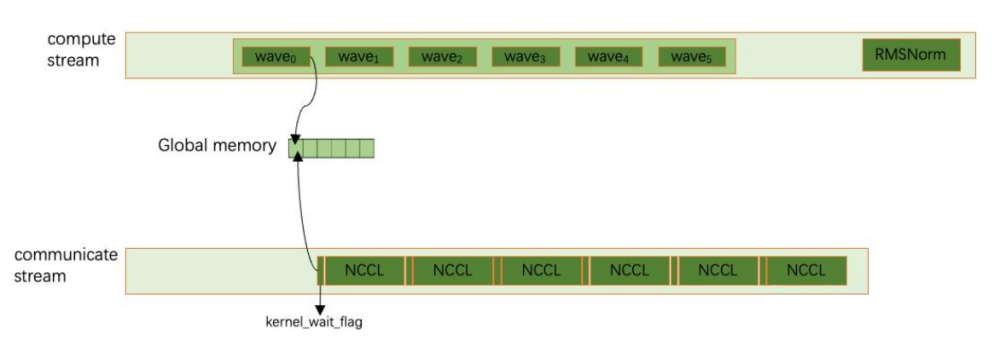


图10 FlashOverlap实现机制

FlashOverlap还引入了Map Table来记录每个分块应该存储位置和由于数据连续性要求而实际存储位置的映射关系，来实现同组内分块的连续存储。Map Table的逻辑被融合到了计算和通信操作后的Element-Wise算子中，来实现对存储偏移影响的纠正。

对于GEMM而言，其分块的计算存在Wave现象，且每个Wave中包含的分块与Swizzling算法有关。通过对实际的GEMM进行测试，可以发现同一个Wave里的分块计算完成时间基本一致，可以被选为同一个分块分组。给定GEMM配置和GPU之后，我们能够提前得到Wave信息，进一步帮助进行分块分组选取。

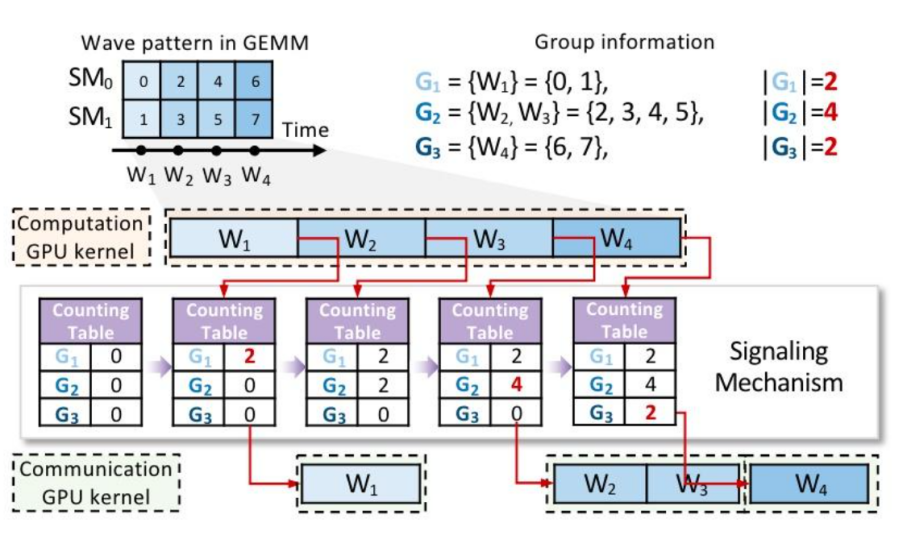


图11 FlashOverlap分块分组示意

每个分块分组包含更多的分块可以提高每次通信的数据量和通信效率，同样地，每个分块分组包含更少的分块则可以缩短通信启动的间隔，减少等待。因此，合适的分块分组需要通过调优来决定，选择一或多个Wave作为一个分块分组。

1. **测试结果**

为确保一致性，我们选用GEMM+ReduceScatter作为测试场景，输入矩阵的M、N、K分别为4096、12288和49152，选用的数据精度为FP16。为获得稳定的性能数据，我们进行了200次迭代，并使用其平均执行时间作为测试结果。和Flux不同的是，FlashOverlap会预先生成不同配置的GEMM内核，并根据需要测试的操作原语生成对应的带宽曲线。

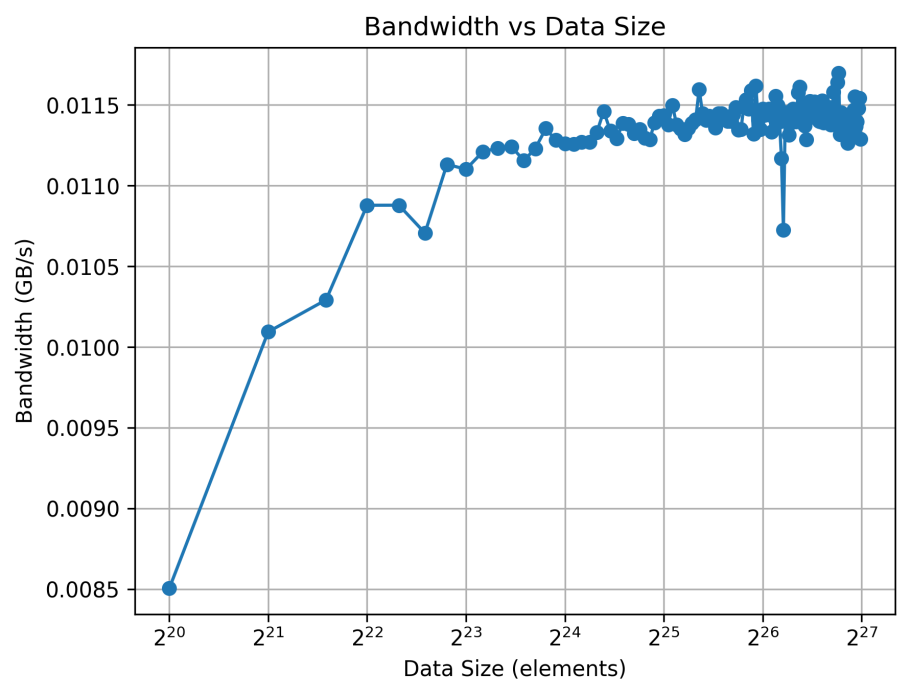


图12 ReduceScatter带宽曲线

该步骤的目的为在测试前使用预测搜索提前确定该输入矩阵对应的最优分块分组策略，避免在测试过程中进行穷举搜索而造成时间过长，生成的最优分块分组如图13所示。

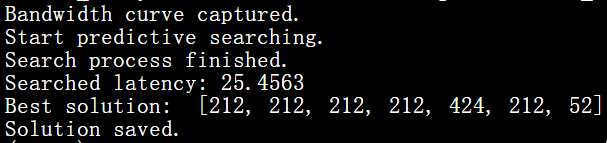


图13 最优分块分组

得到分块分组后，便可以进行相应的测试，测试指令为：CUDA\_VISIBLE\_DEVICES=0,1 python test.py --m 4096 --n 12288 --k 49152 --comm\_op reduce\_scatter。

测试结果如下：

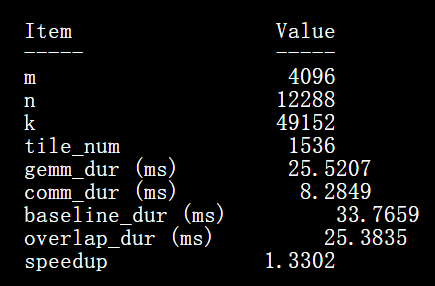


图14 FlashOverlap测试结果

可以看到，相比原先cuBLAS+NCCL的非重叠计算通信，FlashOverlap将整体计算通信时间缩短了近8ms，加速比达到了1.3302，超越了Flux和TransformerEngine在同测试场景下的表现，体现了该方法在计算与通信重叠上的优越性。

1. **优劣势及使用场景分析**

* 以FlashOverlap为代表的基于信号的计算通信重叠方法优势为：

1. 符合NVIDIA GPU内核的执行机制，对计算和通信侵入少，在PCle

设备上提升明显；

1. 通过Tile和Wave的粒度层次设计，可实现灵活可调的粒度重叠机制；
2. 通过预测搜索提前确定最优分组，能有效减少调优时间，并适配不同

GEMM输入；

1. 无需为每个通信原语实现单独开发，大大提升了开发效率；

* 以FlashOverlap为代表的基于信号的计算通信重叠方法劣势为：

1. 该类方法实现依赖Wave机制，对于部分只有一个Wave的场景而言不

可用；

1. 在高带宽互联系统中由于通信开销本身较低，该类方法提升较为有限；
2. 当前仅适配GEMM相关计算，对其他算子的支持有限；

基于信号的计算通信重叠方法适用于对启动延迟敏感或多方共同开发的场景，由于该类方法对算子的改动小，因此尤其适合流水线并行训练和梯度异步通信等实际场景。

1. **总结**

从技术发展路线来说，基于分解的方式最先出现，通过将计算和通信都分解为更小的操作来实现并行化，但会导致GPU无法被充分利用。因此，后续出现的基于融合的方式则会在分解基础上将通信操作融合到计算内核中，实现更优的重叠效果和GPU利用，但也导致计算和通信算子需要重写，开发难度大。由此，基于信号的方式应运而生，不仅对计算和通信的侵入性低，也能实现细粒度的重叠效果。然而这种方法依赖GPU上wave而实现，在部分场景下仍具有局限性，也为后续进一步的改进提出了挑战。