BinaryBackwardKernel 分析与优化

顾文豪

September 27, 2025

1 代码分析

本代码要实现的功能是根据上层传来的 A 的偏导数，B 的偏导数，以及 A 的值，B 的值，得出要反向传播的值, 粗略的分析，函数的瓶颈在于 B 累加时可能会产生大量的原子操作，并且 offset 计算过于零散，导致访存不合并。

1.1 Nsight systems 分析

由于 BinaryBackward 核函数多用在 AI 的训练中，所以我是直接在 H800 上用训练了三次 llama3 得到的结果来分析，未优化之前 BinaryBackward 核函数占用总时间大概为 5%, 如下图: 接下来，我们着手分析一下 BinaryBackward 核函数在 llama3 训练中出现的具体问题。

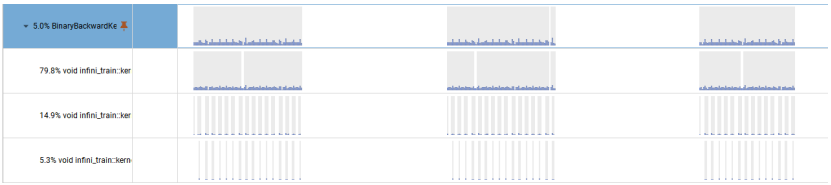

Figure 1: Nsight Systems 分析结果

1.2 Nsight compute 分析

接下来我们查看了 nsight compute 的分析报告，尝试用这个报告为指导去优化函数。在 summary 中，ncu 指出:

1. 尾效应: 在训练 llama3 中，会出现当前内核的启动结果是 2 个完整波和 1 个包含 465 个线程块的部分波。假设所有线程块执行时间相同，那么这个部分波可能占据内核总运行时间的 33.3%(当然很多情况下可能做不到这种优化): 可以看到在这个核函数中网格数为 2048，每个网

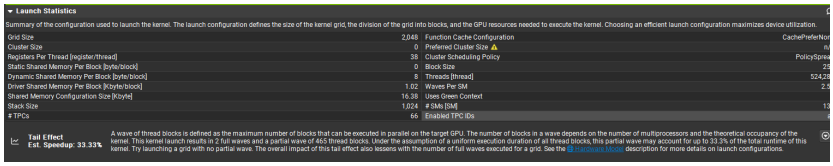

Figure 2: launch statistics 分析结果

格的线程为 256，而在单个 H800 中，SM 的个数为 132，每个 SM 的最大线程数为 2048，而当前的 kernal 的理论占用率为 75%, 如下图：

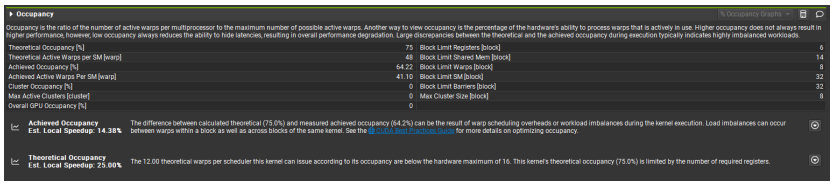

Figure 3: occupancy

于是经过两轮 wave，仍然会有一些线程未启动，这样就产生了本 kernal 中的尾效应 (tail effect) 的问题。

2. 访存模式不佳：根据 ncu 的分析，平均每个 sector 中仅有 17.2B 被使用掉，造成了一定的浪费, 具体的数据如下图：

	Instructions	Requests	Wavefronts	% Peak	Sectors	Sectors/Req	Hit Rate	Bytes	Sector Misses to L2	% Peak to L2	Returns to SM	% Peak to SM
Local Load	0	0	0	0	0	0	0	0			667,456	13.37
Global Load	360,448	360,448			507,904	1.41	73.70	16,252,928	133,582	0.67	-	-
Global Load To Shared Store (access)	0	0	360,448	7.22	0	0		0			-	-
Global Load To Shared Store (bypass)	0	0			0	0	-	0	0	0	-	-
TMA Load	0	-	-	-	-	-	-	-	0	0	-	-
DSMEM Load	0	0	0	0	0	-	-	0	0	0	-	-
Surface Load	0	0	0	0	0	0	0	0	0	0	-	-
Texture Load	0	0	0	0	0	0	0	0	0	0	0	0
Global Store	16,384	16,384	16,384	0.33	65,536	4	0	2,097,152			-	-
Local Store	0	0	0	0	0	0	0	0	65,536	1.31	-	-
TMA Store	0	-	-	-	-	-	-	-	0	0	-	-
DSMEM Store	0	0	0	0	0	-	-	0	0	0	-	-
Surface Store	0	0	0	0	0	0	0	0	0	0	-	-
TMA DSMEM Copy	0	-	-	-	-	-	-	-	0	0	-	-
Global Reduction	16,384	16,384	16,384	0.33	16,384	1	0	524,288	16,384	0.02	-	-
DSMEM Reduction	0	0	0	0	0	-	-	0	0	0	-	-
TMA Global Reduction	0	-	-	-	-	-	-	-	0	0	-	-
TMA DSMEM Reduction	0	-	-	-	-	-	-	-	0	0	-	-
Surface Reduction	0	0	0	0	0	0	0	0	0	0	-	-
Global Atomic ALU	0										see above	see above
Global Atomic CAS	0	0	0	0	0	0	0	0	0	0		
Surface Atomic ALU	0										see above	see above
Surface Atomic CAS	0	0	0	0	0	0	0	0	0	0		
Loads	360,448	360,448	360,448	7.22	507,904	1.41	73.70	16,252,928	133,582	0.67	667,456	13.37
Stores	16,384	16,384	16,384	0.33	65,536	4	0	2,097,152	65,536	1.31	-	-
Atomics & Reductions	16,384	16,384	16,384	0.33	16,384	1	0	524,288	16,384	0.02	-	-
Total	393,216	393,216	393,216	7.67	589,824	1.50	63.46	18,874,368	215,502	2.00	667,456	13.37

Figure 4: occupancy

猜测主要原因是因为在计算 offset 中一直使用乱序的访存。

3. 指令融合：根据分析得到的计算指令来看，多数运算均为整数型运算（这也反映了计算 offset 占据的大部分的时间）, 其中的浮点型运算根据 nuc 的指导可以使用指令融合。

2 代码优化

在 nuc 的分析中发现，在计算 offset 的时候产生了大量的整数型运算，这占据了大多数时间，但是实际上两次的运算所用的元素都基本相同。一定程度上减少了取模的开销，修改代码如下：

```cpp
// ---------------- 优化版（opt）：一次拆 idx，同时得到 A/B offset ----------------
__device__ inline void CalcOffsetsContigOut_opt(
    int64_t idx, int ndim,
    const int64_t* __restrict__ a_strides, const int64_t* __restrict__ a_shape,
    const int64_t* __restrict__ b_strides, const int64_t* __restrict__ b_shape,
    const int64_t* __restrict__ out_shape,
    int64_t& __restrict__ a_off, int64_t& __restrict__ b_off)
{
    a_off = 0; b_off = 0;
    #pragma unroll
    for (int i = ndim - 1; i >= 0; --i) {
        const int64_t dim = out_shape[i];
        int64_t oi = 0;
        if (dim > 1) { oi = idx % dim; idx /= dim; }
        const int64_t ai = (a_shape[i] == 1) ? 0 : oi;
        const int64_t bi = (b_shape[i] == 1) ? 0 : oi;
        a_off += ai * a_strides[i];
        b_off += bi * b_strides[i];
    }
}
```

Figure 5: 显然的优化方案

通过测试程序对比的话，大多数情况下是可以达到加速的。

```
(base) artoriagu@WIN-JVDVMNLRFPR:/mnt/c/Users/Administrator/InfiniTrain/infini_train/src/kernels/cuda$ ./compare
Case #0  A=2,3,4,5,  B=2,3,4,5,  O=2,3,4,5,
  time_ref = 0.0509728 ms   time_opt = 0.0185024 ms   speedup = 2.75493x
[output_a] PASS  max_abs=0 @ 0  max_rel=0
[output_b] PASS  max_abs=0 @ 0  max_rel=0
==> PASS
Case #1  A=2,3,4,5,  B=1,1,4,5,  O=2,3,4,5,
  time_ref = 0.009488 ms   time_opt = 0.0069248 ms   speedup = 1.37015x
[output_a] PASS  max_abs=0 @ 0  max_rel=0
[output_b] PASS  max_abs=3.8147e-06 @ 10  max_rel=2.28129e-07
==> PASS
Case #2  A=2,3,4,5,  B=2,3,4,1,  O=2,3,4,5,
  time_ref = 0.0052864 ms   time_opt = 0.0045888 ms   speedup = 1.15202x
[output_a] PASS  max_abs=0 @ 0  max_rel=0
[output_b] PASS  max_abs=0 @ 0  max_rel=0
==> PASS
Case #3  A=3,5,4,  B=1,1,1,  O=3,5,4,
  time_ref = 0.0062272 ms   time_opt = 0.005664 ms   speedup = 1.09944x
[output_a] PASS  max_abs=0 @ 0  max_rel=0
[output_b] PASS  max_abs=0 @ 0  max_rel=0
==> PASS
Case #4  A=2,3,4,5,  B=2,3,4,1,  O=2,3,4,5,
  time_ref = 0.005776 ms   time_opt = 0.0042464 ms   speedup = 1.36021x
[output_a] PASS  max_abs=0 @ 0  max_rel=0
[output_b] PASS  max_abs=0 @ 0  max_rel=0
==> PASS
Case #5  A=2,3,4,5,  B=1,3,1,5,  O=2,3,4,5,
  time_ref = 0.0054272 ms   time_opt = 0.0040608 ms   speedup = 1.33649x
[output_a] PASS  max_abs=0 @ 0  max_rel=0
[output_b] PASS  max_abs=7.62939e-06 @ 1  max_rel=2.80027e-07
==> PASS

Summary 6/6 passed.
```

Figure 6: test1

更改 offset 的计算方法，应用到 llama3 上，捕捉这个核函数的时间，可以看到他在总体的时间占比降低了一些，并且也可以在事件窗口看到最长执行时间明显小于优化前的最长执行时间.

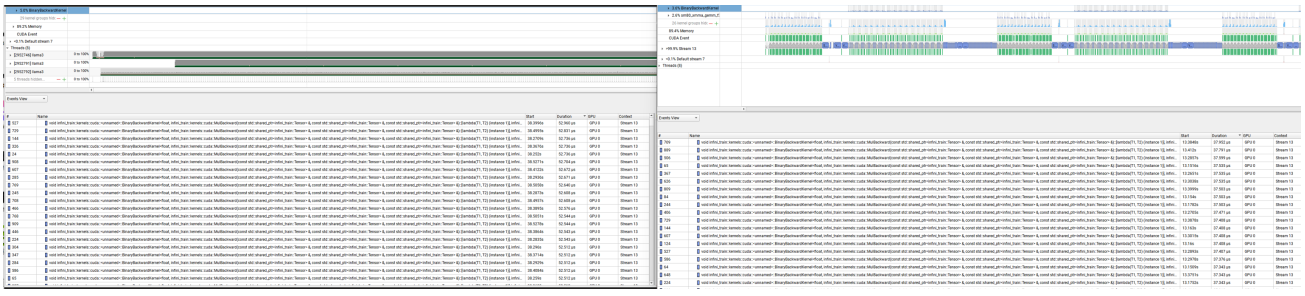

Figure 7: llama3 优化前后对比

当然，上面的优化方案只是在计算 offset 中降低了一些复杂的运算（集中体现在将 A 和 B

合并，降低了近乎一倍的 dim 和 mod 操作），接下来我将尝试更改一下访存方式，以达到访存连续。

近期我又阅读了一下题目需求，以及让我优化的原始代码。发现在题目要求的情况下，a 的 offset 就和 idx 相等，因为无论是题目或者代码都没有说 A 是广播的情况 (事实上，当 A 被广播后, 原来写的方法显然是错误的)。基于这个观点，A 的访存天然是连续的, 并且计算是平凡的，我们只考虑 B 就好。

当 B 是不广播的时候，显然会有

$$B\_\text{offset} = idx$$

, 当 B 进行广播的时候，可以考虑对这个数组进行压缩，从而降低 offset 的计算开销。现在 B 的 offset 计算如下：

```
__device__ __forceinline__
int64_t CalcBOffset_lvl0(uint64_t idx,
                         int eff_ndim_b,
                         const int64_t* __restrict__ eff_out_strides_b,
                         const int64_t* __restrict__ eff_b_strides_b,
                         const int64_t* __restrict__ eff_shape_b) {
    int64_t off = 0;
    #pragma unroll
    for (int j = 0; j < eff_ndim_b; ++j) {
        int64_t out_index = (idx / eff_out_strides_b[j]) % eff_shape_b[j];
        off += out_index * eff_b_strides_b[j];
    }
    return off;   // 若 eff_ndim_b==0, off 恒 0
}
```

Figure 8: 优化 2 的情况

通过 nsight systems 可以发现时间显著下降了：

#	Name	Start	Duration	GPU	Context
931	void infini_train::kernels::cuda::<unnamed>::BinaryBackwardKernel<float, infini_train::kernels::cuda::MulBackward(const std::shared_ptr<infini_train::Tensor> &, const std::shared_ptr<infini_train::Tensor> &, const std::shared_ptr<infini_train::Tensor> &)::[lambda(T1, T2) (instance 1)], infini...	38.8093s	18.624 µs	GPU 0	Stream 13
930	void infini_train::kernels::cuda::<unnamed>::BinaryBackwardKernel<float, infini_train::kernels::cuda::MulBackward(const std::shared_ptr<infini_train::Tensor> &, const std::shared_ptr<infini_train::Tensor> &, const std::shared_ptr<infini_train::Tensor> &)::[lambda(T1, T2) (instance 1)], infini...	38.8093s	11.552 µs	GPU 0	Stream 13
929	void infini_train::kernels::cuda::<unnamed>::BinaryBackwardKernel<float, infini_train::kernels::cuda::MulBackward(const std::shared_ptr<infini_train::Tensor> &, const std::shared_ptr<infini_train::Tensor> &, const std::shared_ptr<infini_train::Tensor> &)::[lambda(T1, T2) (instance 1)], infini...	38.8087s	37.568 µs	GPU 0	Stream 13
928	void infini_train::kernels::cuda::<unnamed>::BinaryBackwardKernel<float, infini_train::kernels::cuda::MulBackward(const std::shared_ptr<infini_train::Tensor> &, const std::shared_ptr<infini_train::Tensor> &, const std::shared_ptr<infini_train::Tensor> &)::[lambda(T1, T2) (instance 1)], infini...	38.808s	36.991 µs	GPU 0	Stream 13
927	void infini_train::kernels::cuda::<unnamed>::BinaryBackwardKernel<float, infini_train::kernels::cuda::AddBackward(const std::shared_ptr<infini_train::Tensor> &, const std::vector<long, std::allocator<long>> &, const std::vector<long, std::allocator<long>> &)::[lambda(T1, T2) (instance 1)], ...	38.8075s	10.464 µs	GPU 0	Stream 13
926	void infini_train::kernels::cuda::<unnamed>::BinaryBackwardKernel<float, infini_train::kernels::cuda::MulBackward(const std::shared_ptr<infini_train::Tensor> &, const std::shared_ptr<infini_train::Tensor> &, const std::shared_ptr<infini_train::Tensor> &)::[lambda(T1, T2) (instance 1)], infini...	38.8074s	18.463 µs	GPU 0	Stream 13
925	void infini_train::kernels::cuda::<unnamed>::BinaryBackwardKernel<float, infini_train::kernels::cuda::MulBackward(const std::shared_ptr<infini_train::Tensor> &, const std::shared_ptr<infini_train::Tensor> &, const std::shared_ptr<infini_train::Tensor> &)::[lambda(T1, T2) (instance 1)], infini...	38.8074s	11.680 µs	GPU 0	Stream 13
924	void infini_train::kernels::cuda::<unnamed>::BinaryBackwardKernel<float, infini_train::kernels::cuda::MulBackward(const std::shared_ptr<infini_train::Tensor> &, const std::shared_ptr<infini_train::Tensor> &, const std::shared_ptr<infini_train::Tensor> &)::[lambda(T1, T2) (instance 1)], infini...	38.807s	3.872 µs	GPU 0	Stream 13
923	void infini_train::kernels::cuda::<unnamed>::BinaryBackwardKernel<float, infini_train::kernels::cuda::MulBackward(const std::shared_ptr<infini_train::Tensor> &, const std::shared_ptr<infini_train::Tensor> &, const std::shared_ptr<infini_train::Tensor> &)::[lambda(T1, T2) (instance 1)], infini...	38.807s	3.904 µs	GPU 0	Stream 13
922	void infini_train::kernels::cuda::<unnamed>::BinaryBackwardKernel<float, infini_train::kernels::cuda::AddBackward(const std::shared_ptr<infini_train::Tensor> &, const std::vector<long, std::allocator<long>> &, const std::vector<long, std::allocator<long>> &)::[lambda(T1, T2) (instance 1)], ...	38.8069s	3.776 µs	GPU 0	Stream 13
921	void infini_train::kernels::cuda::<unnamed>::BinaryBackwardKernel<float, infini_train::kernels::cuda::MulBackward(const std::shared_ptr<infini_train::Tensor> &, const std::shared_ptr<infini_train::Tensor> &, const std::shared_ptr<infini_train::Tensor> &)::[lambda(T1, T2) (instance 1)], infini...	38.8069s	4.416 µs	GPU 0	Stream 13
920	void infini_train::kernels::cuda::<unnamed>::BinaryBackwardKernel<float, infini_train::kernels::cuda::MulBackward(const std::shared_ptr<infini_train::Tensor> &, const std::shared_ptr<infini_train::Tensor> &, const std::shared_ptr<infini_train::Tensor> &)::[lambda(T1, T2) (instance 1)], infini...	38.8069s	4.192 µs	GPU 0	Stream 13
919	void infini_train::kernels::cuda::<unnamed>::BinaryBackwardKernel<float, infini_train::kernels::cuda::SubBackward(const std::shared_ptr<infini_train::Tensor> &, const std::vector<long, std::allocator<long>> &, const std::vector<long, std::allocator<long>> &)::[lambda(T1, T2) (instance 1)], ...	38.8069s	3.968 µs	GPU 0	Stream 13
918	void infini_train::kernels::cuda::<unnamed>::BinaryBackwardKernel<float, infini_train::kernels::cuda::MulBackward(const std::shared_ptr<infini_train::Tensor> &, const std::shared_ptr<infini_train::Tensor> &, const std::shared_ptr<infini_train::Tensor> &)::[lambda(T1, T2) (instance 1)], infini...	38.8067s	6.912 µs	GPU 0	Stream 13
917	void infini_train::kernels::cuda::<unnamed>::BinaryBackwardKernel<float, infini_train::kernels::cuda::MulBackward(const std::shared_ptr<infini_train::Tensor> &, const std::shared_ptr<infini_train::Tensor> &, const std::shared_ptr<infini_train::Tensor> &)::[lambda(T1, T2) (instance 1)], infini...	38.8067s	7.168 µs	GPU 0	Stream 13
916	void infini_train::kernels::cuda::<unnamed>::BinaryBackwardKernel<float, infini_train::kernels::cuda::AddBackward(const std::shared_ptr<infini_train::Tensor> &, const std::vector<long, std::allocator<long>> &, const std::vector<long, std::allocator<long>> &)::[lambda(T1, T2) (instance 1)], ...	38.8067s	6.848 µs	GPU 0	Stream 13

Figure 9: 优化 2 所用的时间

但是此时根据 ncu 的分析，发现还是存在占用率过低的问题，以及尾效应。分析报告说：尾波效应会造成 25% 的浪费，以及寄存器占用过高，实际占用率仍然低于理论值。