高效、易用、可拓展我全都要：OneFlow CUDA Elementwise 模板库的设计优化思路

撰文｜郑泽康、姚迟、郭冉、柳俊丞

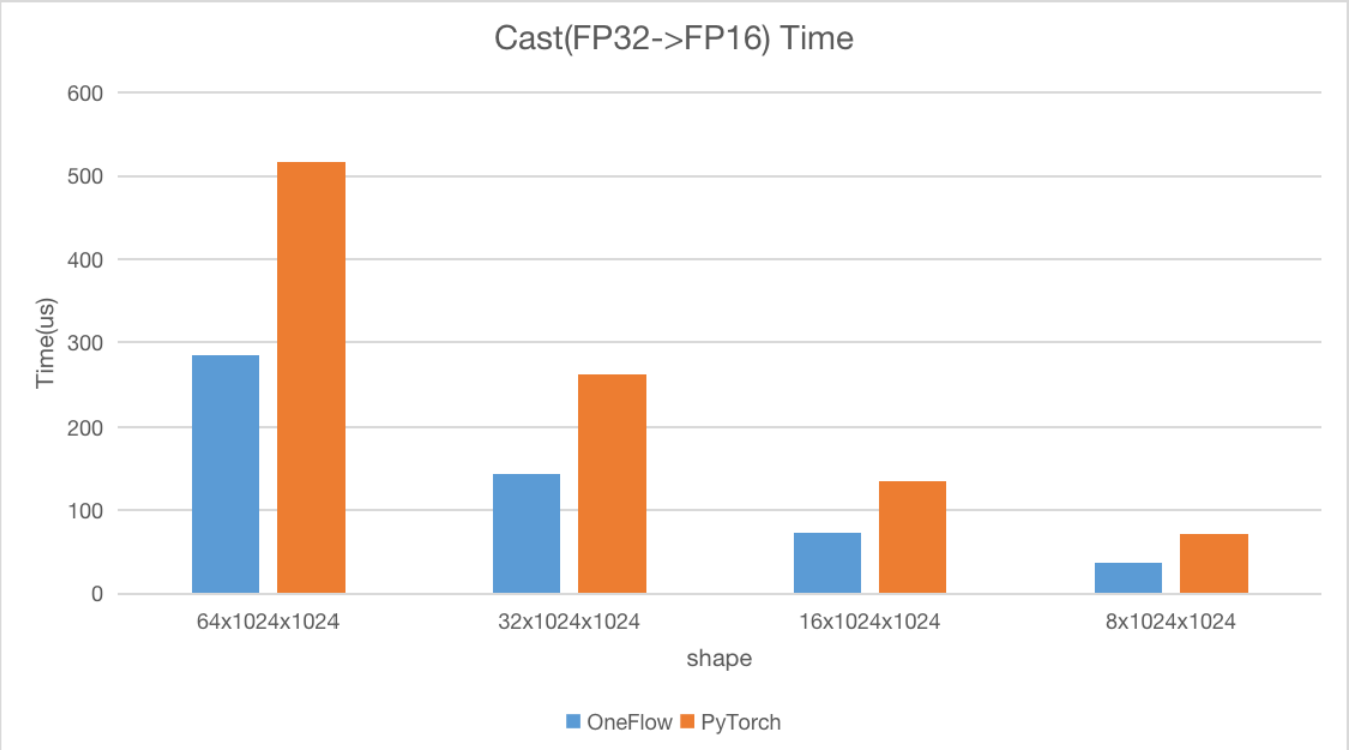
逐元素操作（也叫 Elementwise 操作）是指对 Tensor 中的每个元素应用一个函数变换，得到最终输出结果。在深度学习里，有很多算子属于 Elementwise 算子范畴，比如常用的激活函数（如ReLU、GELU ），ScalarMultiply（对 Tensor 每个元素都乘上一个标量）等操作。

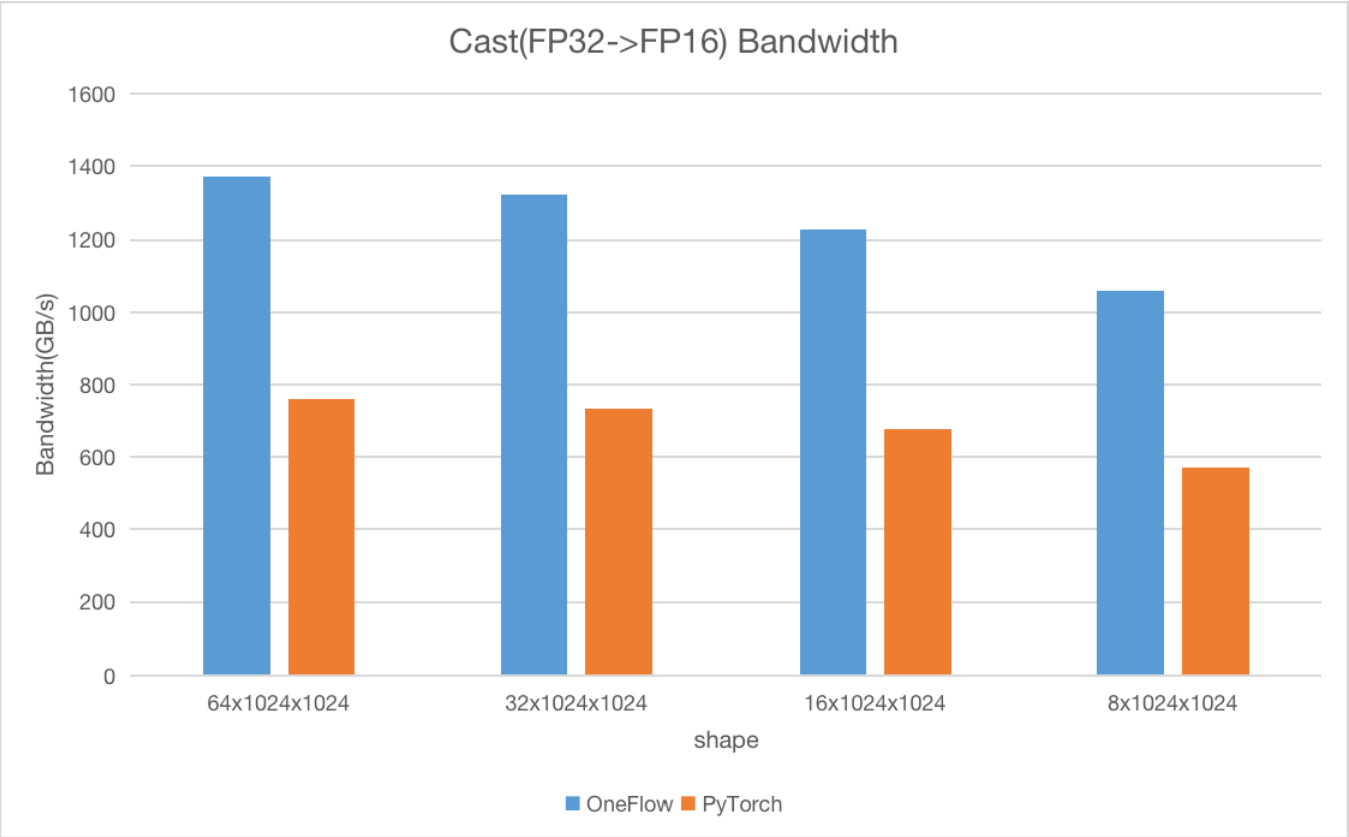
为此 OneFlow 针对这种 Elementwise 操作抽象出一套 CUDA 模板，**开发者只需把计算逻辑封装到一个结构体内，即可获得一个 CUDA Elementwise 算子**，以 ReLU 为例：

```
// Write ReLU Functor.
template<typename T>
struct ReluFunctor {
  OF_DEVICE_FUNC T operator()(T x) const {
    const T zero_val = static_cast<T>(0);
    return (x > zero_val) ? x : zero_val;
  }
};

// Use CUDA Elementwise Template.
OF_CUDA_CHECK((cuda::elementwise::Unary(ReluFunctor<T>(), elem_cnt, dx->mut_dptr<T>(),
                                        x->dptr<T>(), ctx->stream()->As<ep::CudaStream>()->cuda_stream()
```

这样一套简单易用的 Elementwise 模板**不仅提高了开发效率，也能保证计算性能。**我们在 NVIDIA A100 40GB 环境下使用 Nsight Compute，和 PyTorch 的 Cast 算子进行测试，测试用例是将 float32 类型的 Tensor 转换为 half 类型，比较两者的运行时间和带宽，在各个数据形状情况下，**OneFlow 均能比 PyTorch 快 80-90%，并接近机器理论带宽。**





下面我们会逐个介绍这套模板的设计思路以及优化技巧。

设置合理的 BlockSize 和 GridSize

关于设置线程块个数和线程数量的问题，我们在《如何设置CUDA Kernel中的grid_size和block_size？》一文中有讨论过，这里我们的设置规则还稍微有点区别。 在CUDA 官方文档 Compute Capabilities 中提到了：

- 主流架构里，每个 Block 最大寄存器数量是 64 K
- 每个线程所能使用的最大寄存器数量是 255 个

在使用最大寄存器数量的前提下，那每个 Block 最多能启动 `64 * 1024 / 255 = 256` 个线程（往2的倍数取整），因此这里我们设定了一个常量 `constexpr int kBlockSize = 256;`。

而 Grid Size 大小的设置规则在 `GetNumBlocks` 这个函数中：

```
constexpr int kBlockSize = 256
constexpr int kNumWaves = 32;

inline cudaError_t GetNumBlocks(int64_t n, int* num_blocks) {
  ...
  /*
  n: The number of the elements.
  sm_count: The number of the SM.
  tpm: The maximum resident threads in per multiprocessor.
  */
  *num_blocks = std::max<int>(1, std::min<int64_t>((n + kBlockSize - 1) / kBlockSize,
                                                   sm_count * tpm / kBlockSize * kNumWaves));
```

```
  return cudaSuccess;
}
```

- 线程块最小个数为1
- 线程块最大个数是从 处理所有元素所需最小的线程总数 和 wave 数目*GPU 一次可以调度 SM 数量 * 每个 SM 最大 block 数 中取最小值，这里我们的 wave 数目设置为固定32大小

在数据量较小的情况下，不会启动过多的线程块。在数据量较大的情况下，尽可能将线程块数目设置为数量足够多的整数个 wave，以保证 GPU 实际利用率够高。

使用向量化操作

大部分 Elementwise 算子的计算逻辑较为简单，瓶颈主要是在带宽利用上。在英伟达的博客中 CUDA Pro Tip: Increase Performance with Vectorized Memory Access 提到使用向量化操作能够提升读写的带宽，而 CUDA 里也提供了一系列数据类型来支持向量化操作，如float2, float4，就是将2个或4个 float 数据作为一个整体。在一些高性能训练推理库如 LightSeq 就使用了大量的 float4 类型：

```
template <typename T>
__global__ void ker_layer_norm(T *ln_res, T *vars, T *means, const T *inp,
                               const T *scale, const T *bias, int hidden_size) {
  // step 0. compute local sum
  float l_sum = 0;
  float l_square_sum = 0;
  const float4 *inp_f4 = (const float4 *)inp + blockIdx.x * hidden_size; // use float4
  for (uint idx = threadIdx.x; idx < hidden_size; idx += blockDim.x) {
    float4 val = inp_f4[idx];
    ...
  }
}
```

在实际中，我们的算子需要支持不同数据类型（如 int, half ），如果采用 CUDA 内置的向量化数据类型操作，显然要给每个算子写多个版本，增加了开发负担。为此我们实现了一个 Pack 数据结构，用于灵活支持不同数据类型的向量化。

我们先定义了一个 PackType 类型类型来代表向量化的数据，它代表的（向量化后的）数据大小为 sizeof(T) * pack_size。

```
template<typename T, int pack_size>
struct GetPackType {
  using type = typename std::aligned_storage<pack_size * sizeof(T), pack_size * sizeof(T)>::type;
};

template<typename T, int pack_size>
using PackType = typename GetPackType<T, pack_size>::type;
```

然后实现了一个 union 类型 Pack，它内部定义了 PackType<T, pack_size> storage; 来占用空间：

```
template<typename T, int pack_size>
union Pack {
  static_assert(sizeof(PackType<T, pack_size>) == sizeof(T) * pack_size, "");
  __device__ Pack() {
    // do nothing
  }
  PackType<T, pack_size> storage;
  T elem[pack_size];
};
```

与 storage 共享内存的，还有 T elem[pack_size]; 。**这样方便后续的 Elementwise 操作：在后续计算里，我们对 elem 数组中的每个元素都应用 functor，得到输出结果。**

CUDA 里最大支持128 bit 的 pack 大小，而在浮点数据类型中，最小的类型（half）大小为16 bit，最多能把128 / 16=8 个 half 数据 pack 到一起，因此我们设置了这两个常量，kMaxPackBytes 表示 pack 最大字节数，kMaxPackSize 表示 pack 数据的最大个数：

```
constexpr int kMaxPackBytes = 128 / 8;
constexpr int kMaxPackSize = 8;
```

调用链

跟踪 oneflow/core/cuda/elementwise.cuh 中的实现，会发现，这套模板会分别为一元、二元、三元的 Elementwise 提供接口：Unary、Binary、Ternary，文章开始处的 ReLU 算子就使用了 Unary 的接口。进一步分析可以发现，它们经过层层调用后，其实最终都会调用到 ApplyGeneric，基本调用关系如下：

Unary/Binary/Ternary → xxxFactory → GenericLauncher<...>::Launch → ApplyGeneric(CUDA Kernel)

ApplyGeneric 这个 CUDA Kernel 中所做的主要工作是：

- 根据参数创建一个 functor
- 进入循环，针对打包（pack）后的数据，调用 ApplyPack 函数，每调用一次 ApplyPack，就处理一批 pack 后的数据
- 当最后存在元素个数不能被 pack_size 整除的情况时，需要让线程处理下尾部剩余元素

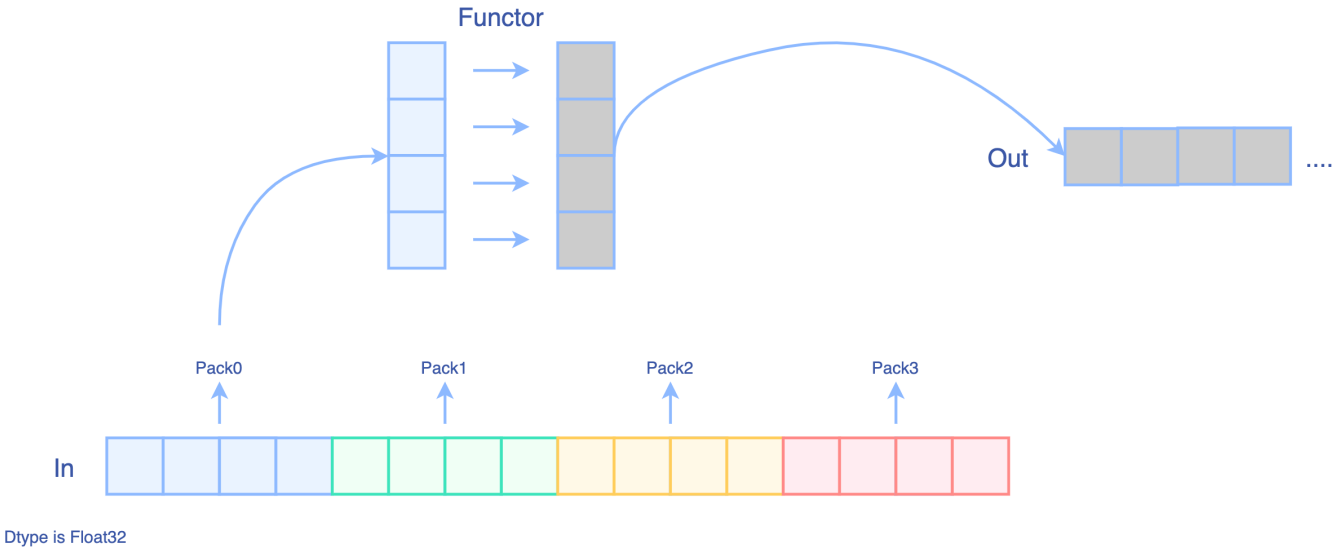
实现代码如下：

```
template<int pack_size, bool tail, typename FactoryT, typename R, typename... IN>
__global__ void __launch_bounds__(kBlockSize)
    ApplyGeneric(FactoryT factory, int64_t n_pack, PackType<R, pack_size>* pack_r,
                 const PackType<IN, pack_size>*... pack_in, int64_t n_tail, R* tail_r,
                 const IN*... tail_in) {
  auto functor = factory();
  const int global_tid = blockIdx.x * kBlockSize + threadIdx.x;
  for (int64_t i = global_tid; i < n_pack; i += blockDim.x * gridDim.x) {
    pack_r[i] = ApplyPack<pack_size, decltype(functor), R, IN...>(
        functor, (FetchPack<IN, pack_size>(pack_in + i).elem)...);
  }
  if (tail && global_tid < n_tail) { tail_r[global_tid] = functor((tail_in[global_tid])...); }
}
```

ApplyPack函数定义如下，它对一个 pack 内的元素做了个循环，对 elem 数组中的每个元素调用 functor ，得到输出结果并返回：

```
template<int pack_size, typename FunctorT, typename R, typename... IN>
__device__
    typename std::enable_if<HasApply2<FunctorT>::value == false, PackType<R, pack_size>>::type
    ApplyPack(const FunctorT& functor, const IN... in[pack_size]) {
  Pack<R, pack_size> ret;
#pragma unroll
  for (int j = 0; j < pack_size; ++j) { ret.elem[j] = functor((in[j])...); }
  return ret.storage;
}
```

整个 Elementwise 算子调用流程如下所示：



针对 half2 数据类型优化

在 half 数据类型下，如果直接对其进行操作，其算子带宽是跟 float32 类型相当的。CUDA 官方有针对 half2 推出一系列特殊指令，如 hadd2 就可以实现两个 half2 数据的加法，进而提高吞吐量。

考虑到这种情况，OneFlow 给 ApplyPack 函数特化了一个版本，通过调用 functor 的 apply2 函数，来调用 half2 相关特殊指令，接口如下：

```
template<int pack_size, typename FunctorT, typename R, typename... IN>
__device__ typename std::enable_if<HasApply2<FunctorT>::value == true && pack_size % 2 == 0,
                                   PackType<R, pack_size>>::type
ApplyPack(const FunctorT& functor, const IN... in[pack_size]) {
  Pack<R, pack_size> ret;
#pragma unroll
  for (int j = 0; j < pack_size; j += 2) { functor.Apply2(ret.elem + j, (in + j)...); }
  return ret.storage;
}
```

以先前的 Cast 算子为例，我们在 CastFunctor 内部通过调用 __float22half2_rn 指令，将一个 float2 数据转换为一个 half2 数据。

```
template<typename From>
struct CastFunctor<half, From, typename std::enable_if<!std::is_same<From, half>::value>::type> {
  ...

  __device__ void Apply2(half* to, const From* from) const {
    float2 f2;
    f2.x = static_cast<float>(from[0]);
    f2.y = static_cast<float>(from[1]);
    *reinterpret_cast<half2*>(to) = __float22half2_rn(f2);
  }
};
```

扩展多元操作

前面已经提到，现有的 OneFlow 模板，将 Elementwise 算子进一步分为一元、二元、三元操作。并利用工厂模式，使得他们最终统一调用 ApplyGeneric。这种设计方式易于拓展：当需要支持更多输入的操作时，只需要编写对应的工厂即可。

```
template<typename FunctorT>
struct SimpleFactory {
  explicit SimpleFactory(FunctorT functor) : tpl(functor) {}
```

```
  __device__ FunctorT operator()() const { return tpl; }

 private:
  FunctorT tpl;
};

template<typename FactoryT, typename R, typename A>
inline cudaError_t UnaryWithFactory(FactoryT factory, int64_t n, R* r, const A* a,
                                    cudaStream_t stream) {
  return GenericLauncher<FactoryT, R, A>::Launch(factory, n, r, a, stream);
}

template<typename FunctorT, typename R, typename A>
inline cudaError_t Unary(FunctorT functor, int64_t n, R* r, const A* a, cudaStream_t stream) {
  return UnaryWithFactory(SimpleFactory<FunctorT>(functor), n, r, a, stream);
}

// BinaryWithFactory TernaryWithFactory ...
// Binary Ternary ...
```

至此，OneFlow 的高性能 CUDA Elementwise 模板的设计，优化手段就介绍完毕，最后再来总结下这套模板的优势：

1. 性能够高，应用这套 Elementwise 模板的算子都能打满机器的带宽，速度也够快。
2. 开发效率高，开发人员可以不用过分关注 CUDA 逻辑及相关优化手段，只需要编写计算逻辑即可。
3. 可扩展性强，目前这套模板支持了一元，二元，三元操作。若今后有需求拓展，支持更多输入时，只需要仿照编写对应的工厂即可。

其他人都在看