CUTLASS源码阅读笔记

0.预备知识

0.1 LD

- leading dimension：如果是列主序，就是同行两个元素间的内存距离。如果目前计算的矩阵是某个内存矩阵的子矩阵，那么内存位置的偏移量应该由ld来指明。
  - 行优先存储的ld是现实世界矩阵的列数
  - 列优先存储的ld是现实世界矩阵的行数

```
// C语言里，数组是行存的，我们仍旧用行存表示数组
double A[M][K], B[K][N];
double C[M][N];
// cublasSgemm理解方式：leading dimension是 独立于计算、存储、框架的 矩阵属性，用来指明二维数组相邻周期同位置元素间的内存距离。那么以行主序存的A数组，这个属性就应该是现实世界矩阵的列数K。在cublas的参数注释里写到
// 1. `cublasSgemm参数lda`是矩阵A存储方式的二维leading dimension，即K。
// 这个参数与`cublasSgemm参数transa`造成的影响没有关系。
// 明确这一点后，因为cublas是按列主序的方式识别`cublasSgemm参数A`传入的矩阵的。因此，直接将行主序存储的A矩阵进行运算，会导致错误的计算结果。所以，
// 2. `cublasSgemm参数transa`提供了一个预处理方法 CUBLAS_OP_T。
// 我们通过将行主序存储的A矩阵进行转置，得到OP(A)，它的内存表达形式，就是A矩阵的列存形式。cublas就能正确的识别A矩阵该有的样子，并进行计算。这里需要清楚的是：OP(A)是现实世界A矩阵的列存形式，它仍旧是M*K的，只是存储形式与原A矩阵不同而已，不要把它理解成一个新的矩阵；
// 同样的，函数输出为争取结果C矩阵的列存，
// 3. `cublasSgemm参数ldc`为矩阵C存储方式的leading dimension，即现实世界矩阵的行数M。
// 4. `cublasSgemm参数m`：OP(A)和C的行数都是M；
cublasStatus_t cublasSgemm(cublasHandle_t handle,
                           cublasOperation_t transa, cublasOperation_t transb,
                           int m, int n, int k,
                           const float           *alpha,
                           const float           *A, int lda,
                           const float           *B, int ldb,
                           const float           *beta,
                           float           *C, int ldc)


// cublas要求参与计算的矩阵是列存的，因此，我们选择CUBLAS_OP_T，将原矩阵进行转置，那么转置后的计算矩阵OP(A)/OP(B)就是列存的原矩阵
// 计算结果为列存的C
cublasStatus_t stat = cublasDgemm(handle, CUBLAS_OP_T, CUBLAS_OP_T, M, N, K,
&alpha, d_A, K, d_B, N, &beta, d_C, M);
```

0.2 术语和知识

- 带default的类是专门负责模板推导的代码。
- Tile size。含义是每个CTA负责结果矩阵中M * N（128 * 128）的部分，每次把A矩阵的M * K（128 * 32）和B矩阵的 K * M放入Shared Memory中。

```
// This code section describes the tile size a thread block will compute
using ShapeMMAThreadBlock = cutlass::gemm::GemmShape<128, 128, 32>;  // <-
threadblock tile M = 128, N = 128, K = 32
```

- Warp size。用tile size来除以warp size就是每个CTA的warp数（每一维前者都必须是后者的整数倍）。M与N的含义是该warp负责的大小。warp size.k如果小于tile_size.k,那么这里就是使用了SlicedK的方法，反之则为单纯的IterationK。

```
// This code section describes tile size a warp will compute
using ShapeMMAWarp = cutlass::gemm::GemmShape<64, 64, 32>;  // <- warp tile M =
64, N = 64, K = 32
```

- 单条Tensor Core指令负责的部分的大小

```
// This code section describes the size of MMA op
using ShapeMMAOp = cutlass::gemm::GemmShape<8, 8, 4>;  // <- MMA Op tile M = 8, N
= 8, K = 4
```

0.100 模板元编程

- 初见，编译期展开

```
template <size_t N>
struct SumUp
{
    enum { value = SumUp<N-1>::value + N };
};
// 模板特化
template <>
struct SumUp<0>
{
    enum { value = 0 };
};
// 编译期展开为5 + SumUp<4> + SumUp<3> + SumUp<2> + SumUp<1> + SumUp<0>。因为我们使用模板特化，特化了当模板参数为0时value的值，模板展开便会以此为递归基停止递归的展开。最后便为5 + 4 + 3 + 2 + 1 + 0 = 15。这都是在编译期就提前算好的哦~
```

- 编译器常量 constexpr

```
// 这个模板程序接受一种类型和该类型的值，然后我们通过constexpr来在编译期固定它的值，我们就可以使用此技巧在编译期表示一个字面值常量。又在此基础上对.()和T()函数进行了重载，以方便我们获取里面存放的值
template <typename T, T v>
struct integral_constant
{
    static constexpr T value = v;
    typedef T value_type;
    typedef integral_constant<T, v> type;

    constexpr value-type operator()() const noexcept { return value; }
    constexpr operator value_type() const noexcept { return value; }
```

```
};
// 现在，我们可以在编译期表示一个bool
// 只需要通过true_type::value，便可在编译期表示布尔类型的true。
typedef integral_constant<bool, true> true_type;
typedef integral_constant<bool, false> false_type;
typedef integral_constant<float, 2.33> float_constant;

// 类似技巧：
template <typename T> struct identity { typedef T type; };
template <typename T> using  identity_t = typename identity<T>::type;
// 我们可以用identity来在编译期表示一个类型。在用到我们自己的类型时，前面要加上typename来告诉编译期我这一个值代表一种类型，编译器才不会报错。
```

- 进阶1

```
template <bool B>
struct TmpIf { constexpr static size_t value = 1; };
// 这里使用模板特化构成的分支，来实现了一个编译期的if语句。如果模板值为false，value则会特化为0；
// 如果模板值为true，函数模板便会使用一般的版本，即value为1
template <>
struct TmpIf<false> { constexpr static size_t value = 0; };

// if constexpr
// 它会在编译期判断需要的分支，舍弃掉不需要的分支，然后相对应的代码。
template <bool b>
int TmpIf()
{
    if constexpr (b)
        return 1;
    else
        return 0;
};
```

```
struct unused {}; // unused的定义

template <bool B, typename T = void>
struct enable_if {};

template <typename T>
struct enable_if<true, T> { typedef T type; };

template <bool B, typename T = void>
using enable_if_t = typename enable_if<B, T>::type;
// 通过对第二模板参数的默认值设置为void，在模板展开时，若编译期检测到B为false时，无论你有没有给定T且无论T是什么，编译期都会选择默认分支来进行代码的生成。此时enable_if就是一个dummy struct(哑类)，也是我们模板元编程中一个特殊值，unused的定义
```

```
template <class T> struct is_const_value : false_type {};
template <class T> struct is_const_value<const T*> : true_type {};
template <class T> struct is_const_value<const volatile T*> : true_type {};

template <class T> struct is_const : is_const_value<T*> {};
template <class T> struct is_const<T&> : false_type {};
```

```
template <typename >
struct is_function : false_type {};

template <typename ReturnType, typename ...Args>
struct is_function<ReturnType(Args...)> : true_type {};

template <typename ReturnType, typename ...Args>
struct is_function<ReturnType(Args..., ...)> : true_type {};

template <class T> struct remove_const { typedef T type; };
template <class T> struct remove_const<const T> { typedef T type; };
template <class T> struct remove_const<const T[]> { typedef T type[]; };
template <class T, size_t N> struct remove_const<const T[N]> { typedef T type[N];
};
```

- 进阶2

```
// 求数组的阶数，就是维度
// 如果是rank<int> 直接返回0
template <typename T>
struct rank : public integral_constant<size_t, 0> {};
// 如果是rank<int[4]> 则展开为 rank<int>::value+1
template <typename T>
struct rank<T[]> : public integral_constant<size_t, rank<T>::value + 1> {};
// 如果是rank<int[4][5][6][7]> 模板此时会判断到第三个实现，即此函数为一个T[N]，这表示它为一个指针数组。在这我们使用多了一个模板参数size_t N来让模板自动检测到多维数组。此时的常量便会展开为rank<int[5][8][5]> + 1。函数会接着展开，直到到达rank<int>
template <typename T, size_t N>
struct rank<T[N]> : public integral_constant<size_t, rank<T>::value + 1> {};

template <typename T>
constexpr auto rank_v = rank<T>::value;
```

```
template <size_t... Intergrals>
struct static_max {};

template <size_t I0>
struct static_max<I0> { static const constexpr size_t value = I0; };

template <size_t I0, size_t I1, size_t... I>
struct static_max<I0, I1, I...>
{
    static const constexpr size_t value = (I0 <= I1) ?
        static_max<I1, I...>::value :
        static_max<I0, I...>::value;
};
```

```
// 短路and
template <typename ...>
struct conjunction : true_type {};

template <typename T>
struct conjunction<T> : T {};

template <typename T, typename... Args>
struct conjunction<T, Args...> :
conditional<bool(T::value), conjunction<Args...>, T>::type {};
```

```
template<int... N>
struct index_seq{};

template<int N, int... M>
struct make_index_seq : public make_index_seq<N-1, M...>{};

template<int... M>
struct make_index_seq<0, M...>: public index_seq<M...>{};
```

1.软件分层

> 以basic_gemm.cu为例

device

- basic_gemm.cu && device\gemm.h

```
// 1.提供对外接口
using CutlassGemm = cutlass::gemm::device::Gemm<~>;
CutlassGemm gemm_operator;
// 2. 启动内核
// cutlass::Status status = gemm_operator(args);
class Gemm<~, Operator_, ~>{
    Status run(cudaStream_t stream = nullptr){
        cutlass::Kernel<GemmKernel><<<grid, block, smem_size, stream>>>(params_);
    }
}
// 其中
typename Operator_ = typename DefaultGemmConfiguration<OperatorClass_,
~>::Operator
```

- default_gemm_configuration.h

- 这个Operator模板参数会沿着调用和模板特化一直传递下去，也就是说，算子在这里由传入的模板确参数定。

```
using Operator = arch::OpMultiplyAdd;
using Operator = arch::OpMultiplyAddSaturate;
using Operator = arch::OpMultiplyAddComplex;
```

kernel

- kernel\default_gemm.h

```
// Gemm的种类由辅助类DefaultGemm确定
using GemmKernel = typename kernel::DefaultGemm<~>::GemmKernel;
// 定义kernel-level GEMM operator
using GemmKernel = kernel::Gemm<Mma, Epilogue, ThreadblockSwizzle, SplitKSerial>;
// 定义threadblock范围的matrix multiply-accumulate
using Mma = typename cutlass::gemm::threadblock::DefaultMma<~>::ThreadblockMma;
```

- kernel\gemm.h

```
// 主循环里构建thread-scoped matrix multiply
Mma mma(shared_storage.main_loop, thread_idx, warp_idx, lane_idx);
```

threadblock

- threadblock\default_mma.h
  - 不同的MmaPolicy对应不同的Operator

```
// TensorOp 和 Simt
// Define the threadblock-scoped pipelined matrix multiply
using ThreadblockMma = cutlass::gemm::threadblock::MmaPipelined<~, typename
MmaCore::MmaPolicy>;

using MmaCore = typename cutlass::gemm::threadblock::DefaultMmaCore<~, Operator>;
```

- threadblock\default_mma_core_sm80.h

```
using MmaPolicy = MmaPolicy<MmaTensorOp, ~>;

using MmaTensorOp = typename cutlass::gemm::warp::DefaultMmaTensorOp<~, Operator,
~>::Type;
```

- threadblock\default_mma_core_simt.h

```
//  MatrixShape<0, kPaddingN>: skew for B matrix to avoid SMEM bank conflicts
using MmaPolicy = MmaPolicy<MmaWarpSimt, ~>;

using MmaWarpSimt = cutlass::gemm::warp::MmaSimt<~, Policy>;
/// Policy describing warp-level MmaSimtOp (concept: MmaSimtOp policy)
using Policy = cutlass::gemm::warp::MmaSimtPolicy<~>;
```

- threadblock\mma_pipelined.h

```
// gemm_iters() main loop
for(){ for(){ warp_mma(); } }
// MmaCore::MmaPolicy
// 所以mma_pipelined 要么执行MmaTensorOp算子，要么执行MmaWarpSimt算子
Policy::Operator warp_mma;
```

warp/thread

- warp\default_mma_tensor_op.h

```
template<~, Operator_, ~>
struct DefaultMmaTensorOp {
  using Policy = cutlass::gemm::warp::MmaTensorOpPolicy< cutlass::arch::Mma<~,
Operator_>,  ~ >;
  // Define the warp-level tensor op
  using Type = cutlass::gemm::warp::MmaTensorOp< ~, Policy, ~ >;
};
```

- warp\mma_tensor_op.h

```
// main
for(){ for(){ mma(); } }
// arch::Mma
Policy::Operator mma;
```

- warp\mma_simt.h

```
// main
mma();
//
thread::Mma<~, arch::OpMultiplyAdd, dp4a_type> mma;
```

- thread\mma_sm61.h

```
// main
for(){ for(){ for(){ mma(); } } }
//
arch::Mma<~, arch::OpMultiplyAdd> mma;
```

arch

- arch\mma_sm80.h

```
asm volatile(
        "mma.sync.aligned.m16n8k64.row.col.s32.u4.u4.s32.satfinite {%0,%1,%2,%3},
"
        "{%4,%5,%6,%7}, {%8,%9}, {%10,%11,%12,%13};\n"
        : "=r"(D[0]), "=r"(D[1]), "=r"(D[2]), "=r"(D[3])
        : "r"(A[0]), "r"(A[1]), "r"(A[2]), "r"(A[3]), "r"(B[0]), "r"(B[1]),
          "r"(C[0]), "r"(C[1]), "r"(C[2]), "r"(C[3]));
```

- arch\mma_sm61.h

```
asm volatile("dp4a.s32.s32 %0, %1, %2, %3;"
                : "=r"(d[0])
                : "r"(A), "r"(B), "r"(c[0]));
```

2.数据流动

- mma_pipelined.h

```
void operator() (~){
    // Prologue
    prologue(iterator_A, iterator_B, gemm_k_iterations);
    // Wait until we have at least one completed global fetch stage
    gmem_wait();
    // Perform the MAC-iterations
    gemm_iters(gemm_k_iterations, accum, iterator_A, iterator_B);
}

// 启动global->shared memory pipeline 从globalMem中取第一个(kStages-1线程块主循环)迭代
所需的fragments到共享内存
void prologue(
    IteratorA &iterator_A,      ///< [in|out] iterator over A operand in global
memory
    IteratorB &iterator_B,      ///< [in|out] iterator over B operand in global
memory
    int &gemm_k_iterations)     ///< [in|out] number of threadblock mainloop
iterations remaining
{
    // Load A fragment from global A
    // Load B fragment from global B
    // Store A and B fragments to shared
    // Advance write stage
}

// Perform the specified number of threadblock mainloop iterations of matrix
multiply-accumulate.
void gemm_iters(
    int gemm_k_iterations,        ///< number of threadblock mainloop iterations
    FragmentC &accum,             ///< [in|out] accumulator tile
```

```
    IteratorA &iterator_A,        ///< [in|out] iterator over A operand in global
memory
    IteratorB &iterator_B)        ///< [in|out] iterator over B operand in global
memory
{
    // Load A fragment from shared A
    this->warp_tile_iterator_A_.load(warp_frag_A[0]);
    // Load B fragment from shared B

    for (; gemm_k_iterations > 0; --gemm_k_iterations){
        for (int warp_mma_k = 0; warp_mma_k < Base::kWarpGemmIterations;
++warp_mma_k) {
            // if is inner loop end-1
             if (warp_mma_k == Base::kWarpGemmIterations - 1) {
                  // Write fragments to shared memory
             }
            // Load A fragment from shared A && Load B fragment from shared B
            this->warp_tile_iterator_A_.load(warp_frag_A[(warp_mma_k + 1) % 2]);
            this->warp_tile_iterator_B_.load(warp_frag_B[(warp_mma_k + 1) % 2]);
            // if is inner loop begin
            if (warp_mma_k == 0) {
                // Load fragment from global A
            }
            // math instructions
            warp_mma(accum, warp_frag_A[warp_mma_k % 2],  warp_frag_B[warp_mma_k
% 2], accum);
        }
    }
}
```

3.计算

- thread\mma_sm61.h

```
/// Computes a matrix product D = A * B + C
CUTLASS_HOST_DEVICE
void operator() (~){
    /// Use 1x1x4 IDP4A sequence for bulk of computation
    ArchMmaOperator mma;
    // Compute matrix product
    for (int k = 0; k < Shape::kK / ArchMmaOperator::Shape::kK; ++k) {
        for (int n = 0; n < Shape::kN; ++n) {
            for (int m = 0; m < Shape::kM; ++m) {
                mma(tmp, ptr_A[m * Shape::kK / ArchMmaOperator::Shape::kK + k],
                        ptr_B[n * Shape::kK / ArchMmaOperator::Shape::kK + k],
tmp);
            }
        }
    }
}
```

- warp\mma_tensor_op.h

```
for (int m = 0; m < MmaIterations::kRow; ++m) {
    for (int n = 0; n < MmaIterations::kColumn; ++n) {
            mma(ptr_D[m + n_serpentine * MmaIterations::kRow],
                ptr_A[m],
                ptr_B[n_serpentine],
                ptr_D[m + n_serpentine * MmaIterations::kRow]);
        }
    }
}
```