张量核编程方法

Agenda

- WMMA API 及其用法
- CUTLASS 中的张量核编程

WMMA API 及其用法

```
template<typename Use, int m, int n, int k, typename T, typename Layout=void> class fragment;

void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned ldm);
void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned ldm, layout_t layout);
void store_matrix_sync(T* mptr, const fragment<...> &a, unsigned ldm, layout_t layout);
void fill_fragment(fragment<...> &a, const T& v);
void mma_sync(fragment<...> &d, const fragment<...> &a, const fragment<...> &b, const fragment<...> &c, bool satf=false);
```

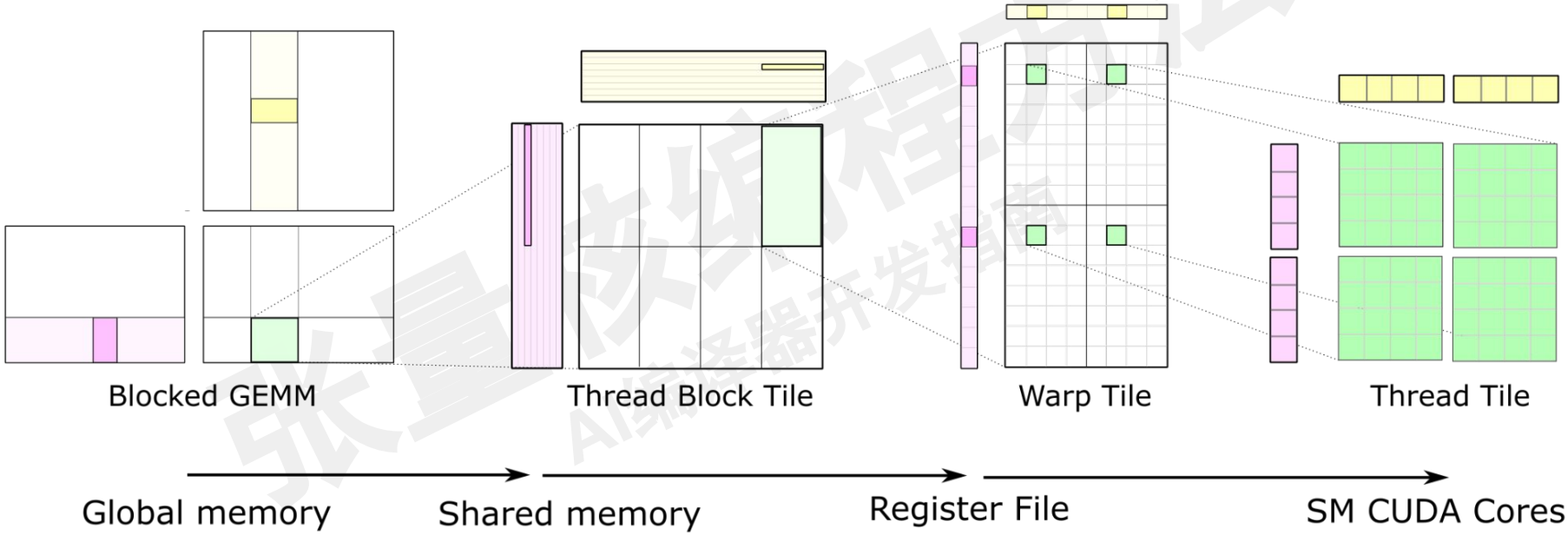
```
#include <mma.h>
using namespace nvcuda;

__global__ void wmma_ker(half *a, half *b, float *c) {
   // Declare the fragments
   wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::col_major> a_frag;
   wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::row_major> b_frag;
   wmma::fragment<wmma::accumulator, 16, 16, 16, float> c_frag;

   // Initialize the output to zero
   wmma::fill_fragment(c_frag, 0.0f);

   // Load the inputs
   wmma::load_matrix_sync(a_frag, a, 16);
   wmma::load_matrix_sync(b_frag, b, 16);

   // Perform the matrix multiplication
   wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);

   // Store the output
   wmma::store_matrix_sync(c, c_frag, 16, wmma::mem_row_major);
}
```

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#wmma

CUTLASS中的张量核编程

- CUTLASS（CUDA Templates for Linear Algebra Subroutines）CUDA C++模板化头文件库，可实现不同精度的GEMM计算。
- CUTLASS模板化库可在编译时由编译器根据输入参数实现定制化和代码优化。
- CUTLASS 中的 GEMM 实现按照设备、线程块、线程束和线程四个层级,通过将操作数矩阵划分为不同级别的数据块来组织计算。

CUTLASS中的GEMM架构



08_turing_tensorop_gemm

主机端代码

```
Gemm gemm_op;
status = gemm_op();
```

```
typename Gemm::Arguments arguments{problem_size,
                                   tensor_a.device_ref(),  // <- reference to matrix A on device
                                   ...
```

gemm::device::Gemm

```
template <...>
class Gemm {//device level GEMM
 using GemmKernel = typename kernel::DefaultGemm<...>::GemmKernel
 Status operator()(cudaStream_t stream = nullptr) {
  cutlass::Kernel<GemmKernel><<<grid, block, smem_size, stream>>>(params_);
 }
}
```

```
typename GemmKernel::Params params_

params_ = typename
GemmKernel::Params{
 args.problem_size,
 grid_shape,
 args.ref_A.non_const_ref(),
 ...
};
```

cutlass::Kernel

```
template <typename Operator>
__global__ void Kernel(typename Operator::Params params) {
 Operator op;
 op(params, *shared_storage);
}
```

设备层

Kernel层

kernel::DefaultGemm

```
template <...>
struct DefaultGemm<...> {
 using Mma = typename cutlass::gemm::threadblock::DefaultMma<...>::ThreadblockMma;
 using GemmKernel = kernel::Gemm<Mma, Epilogue, ThreadblockSwizzle, SplitKSerial>;
};
```

kernel::Gemm

```
template <typename Mma_,...>
struct Gemm {//kernel level GEMM
 using Mma = Mma_;
 void operator()(Params const &params, SharedStorage &shared_storage) {
  Mma mma(shared_storage.main_loop, thread_idx, warp_idx, lane_idx);
  mma(gemm_k_iterations, accumulators, iterator_A, iterator_B, accumulators);
 };
```

typename Mma::IteratorA/B iterator_A/B

```
struct Params {
  cutlass::gemm::GemmCoord problem_size;
  cutlass::gemm::GemmCoord grid_tiled_shape;
  typename Mma::IteratorA::Params params_A;
  ...
```

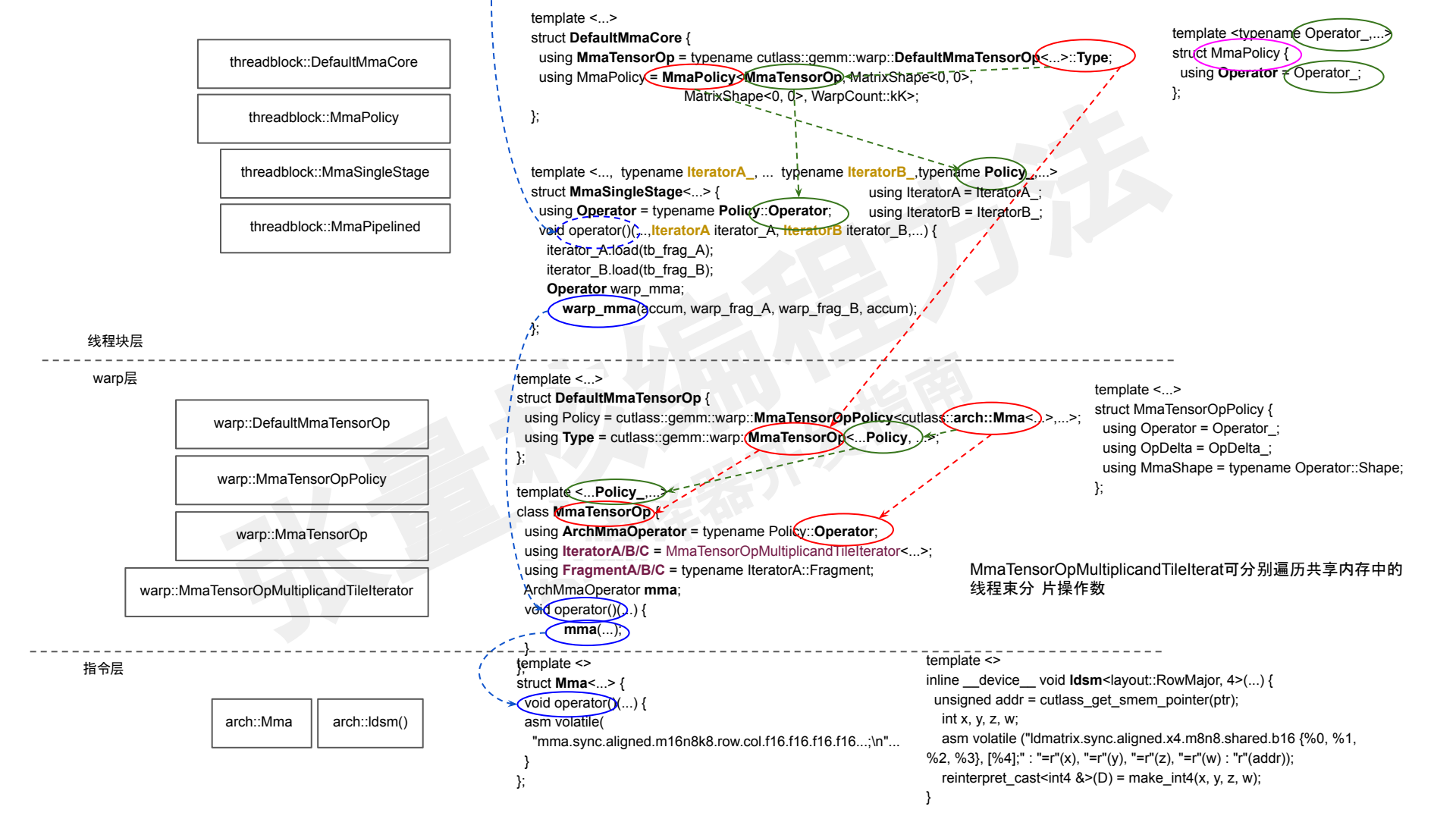
线程块层

threadblock::DefaultMma

```
template <...>
struct DefaultMma<...> {
 using MmaCore = typename cutlass::gemm::threadblock::DefaultMmaCore<...>;
 using IteratorA/B = cutlass::transform::threadblock::PredicatedTileIterator<...>;
 using ThreadblockMma = cutlass::gemm::threadblock::MmaSingleStage<..., IteratorA,..., IteratorB, ...
                                                                   typename MmaCore::MmaPolicy>;
};
```

threadblock::PredicatedTileIterator

PredicatedTileIterator遍历全局内存中的A、B 矩阵分片序列


threadblock::DefaultMmaCore
threadblock::MmaPolicy
threadblock::MmaSingleStage
threadblock::MmaPipelined
template <...>
struct DefaultMmaCore {
using MmaTensorOp = typename cutlass::gemm::warp::DefaultMmaTensorOp<...>::Type;
using MmaPolicy = MmaPolicy<MmaTensorOp, MatrixShape<0, 0>,
MatrixShape<0, 0>, WarpCount::kK>;
};
template <typename Operator_,...>
struct MmaPolicy {
using Operator = Operator_;
};
template <..., typename IteratorA_, ... typename IteratorB_,typename Policy_,...>
struct MmaSingleStage<...> {
using Operator = typename Policy::Operator;
void operator()(...,IteratorA iterator_A, IteratorB iterator_B,...) {
iterator_A.load(tb_frag_A);
iterator_B.load(tb_frag_B);
Operator warp_mma;
warp_mma(accum, warp_frag_A, warp_frag_B, accum);
};
using IteratorA = IteratorA_;
using IteratorB = IteratorB_;
线程块层
warp层
warp::DefaultMmaTensorOp
warp::MmaTensorOpPolicy
warp::MmaTensorOp
warp::MmaTensorOpMultiplicandTileIterator
template <...>
struct DefaultMmaTensorOp {
using Policy = cutlass::gemm::warp::MmaTensorOpPolicy<cutlass::arch::Mma<...>,...>;
using Type = cutlass::gemm::warp::MmaTensorOp<...Policy, ...>;
};
template <...Policy_,...>
class MmaTensorOp {
using ArchMmaOperator = typename Policy::Operator;
using IteratorA/B/C = MmaTensorOpMultiplicandTileIterator<...>;
using FragmentA/B/C = typename IteratorA::Fragment;
ArchMmaOperator mma;
void operator()(...) {
mma(...);
}
};
template <>
struct MmaTensorOpPolicy {
using Operator = Operator_;
using OpDelta = OpDelta_;
using MmaShape = typename Operator::Shape;
};
MmaTensorOpMultiplicandTileIterat可分别遍历共享内存中的
线程束分 片操作数
指令层
arch::Mma
arch::ldsm()
template <>
struct Mma<...> {
void operator()(...) {
asm volatile(
"mma.sync.aligned.m16n8k8.row.col.f16.f16.f16.f16...;\n"...
}
};
template <>
inline __device__ void ldsm<layout::RowMajor, 4>(...) {
unsigned addr = cutlass_get_smem_pointer(ptr);
int x, y, z, w;
asm volatile ("ldmatrix.sync.aligned.x4.m8n8.shared.b16 {%0, %1,
%2, %3}, [%4];" : "=r"(x), "=r"(y), "=r"(z), "=r"(w) : "r"(addr));
reinterpret_cast<int4 &>(D) = make_int4(x, y, z, w);
}