Cutlass SGEMM Analysis

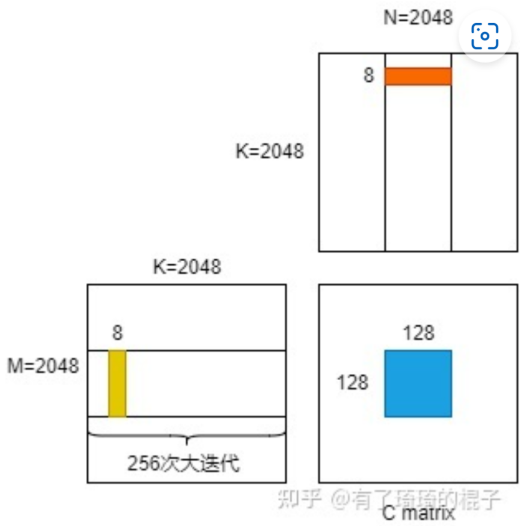
Review of GEMM Implementation

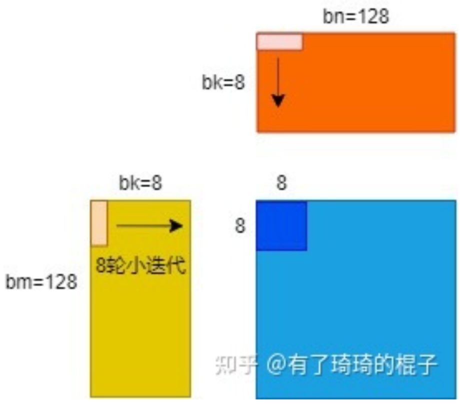
GEMM general structure

- infrastructure
  1. A -> mxk & B -> kxn & C -> mxn
  2. $C = \alpha * A \times B^T + \beta * C$
- solution
  1. m = M * bm & n = N * bn & k = K * bk
  2. a grid = M * N blocks , each block calculate a bm * bn area
  3. for a block , each time load bm * bk data from A and bk * bn data from B to shared memory
  4. there are K rounds for a block to calculate the target block
  5. bm = X * rm & bn = Y * rn
  6. there are X * Y threads, and each thread calculate a block of rm * rn
- implement

```
// Device function to compute a thread block's accumulated matrix product
__device__ void block_matrix_product(int K_dim) {

    // Fragments used to store data fetched from SMEM
    value_t frag_a[ThreadItemsY];
    value_t frag_b[ThreadItemsX];

    // Accumulator storage
    accum_t accumulator[ThreadItemsX][ThreadItemsY];

    // GEMM Mainloop - iterates over the entire K dimension - not unrolled
    for (int kblock = 0; kblock < K_dim; kblock += BlockItemsK) {

        // Load A and B tiles from global memory and store to SMEM
        //
        // (not shown for brevity - see the CUTLASS source for more detail)
        ...

        __syncthreads();

        // Warp tile structure - iterates over the Thread Block tile
        #pragma unroll
        for (int warp_k = 0; warp_k < BlockItemsK; warp_k += WarpItemsK) {

            // Fetch frag_a and frag_b from SMEM corresponding to k-index
            //
            // (not shown for brevity - see CUTLASS source for more detail)
            ...
```

```
            // Thread tile structure - accumulate an outer product
            #pragma unroll
            for (int thread_x = 0; thread_x < ThreadItemsX; ++thread_x) {
                #pragma unroll
                for (int thread_y=0; thread_y < ThreadItemsY; ++thread_y) {
                    accumulator[thread_x][thread_y] += frag_a[y]*frag_b[x];
                }
            }
        }

        __syncthreads();
    }
}
```





- now the above optimization can reach 80% cublas

Advance Optimization

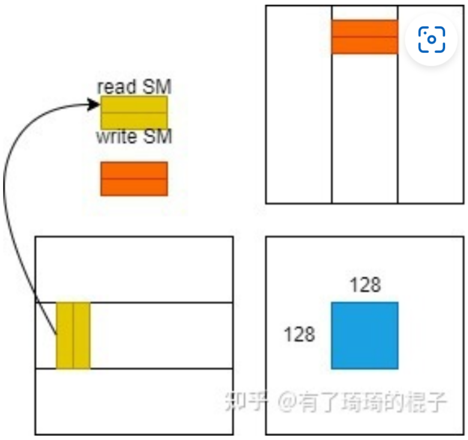
- transpose matrix A (smemA) and use LDS.128 (vector read instruction)
- try LDGSTS ?
- prefetch & read/write into different buffer (double buffer)

  we use read SM & write SM to represent the two shared memory

  1. load the data to write SM and write reg

2. for each iteration, switch read SM & write SM to avoid the latency of accessing mem

```
for k in 256 big_loop:
    prefetch next loop data to write_SM
    // compute in read_SM
    for iter in 8 small_loop:
        prefetch next loop data to write_REG
        compute in read_REG
```



- implement

```
#define TILE_K 16
    __shared__ float4 smemA[2][TILE_K * 128 / 4];
    __shared__ float4 smemB[2][TILE_K * 128 / 4];
    float4 c[8][2] = {{make_float4(0.f, 0.f, 0.f, 0.f)}};
    float4 ldg_a_reg[2];
    float4 ldg_b_reg[2];
    float4 a_reg[2][2];
    float4 b_reg[2][2];

    // transfer first tile from global mem to shared mem
    load_gmem_tile_to_reg(A, 0, ldg_a_reg);
    load_gmem_tile_to_reg(B, 0, ldg_b_reg);

    store_reg_to_smem_tile_transpose(ldg_a_reg, 0, smemA[0]);
    store_reg_to_smem_tile(ldg_b_reg, 0, smemB[0]);
    __syncthreads();

    // load first tile from shared mem to register
    load_smem_tile_to_reg(smemA[0], 0, a_reg[0]);
    load_smem_tile_to_reg(smemB[0], 0, b_reg[0]);

    int write_stage_idx = 1; //ping pong switch
    do {
        i += TILE_K;
        // load next tile from global mem
        load_gmem_tile_to_reg(A, i, ldg_a_reg);
        load_gmem_tile_to_reg(B, i, ldg_b_reg);

        int load_stage_idx = write_stage_idx ^ 1;

    #pragma unroll
```

```
        for(int j = 0; j < TILE_K - 1; ++j) {
            // load next tile from shared mem to register
            load_smem_tile_to_reg(smemA[load_stage_idx], j + 1, a_reg[(j +
1) % 2]);
            load_smem_tile_to_reg(smemB[load_stage_idx], j + 1, b_reg[(j +
1) % 2]);
            // compute matrix multiply accumulate 8x8
            mma8x8(a_reg[j % 2], b_reg[j % 2], c);
        }

        if(i < K) {
            // store next tile to shared mem
            store_reg_to_smem_tile_transpose(ldg_a_reg, 0,
smemA[write_stage_idx]);
            store_reg_to_smem_tile(ldg_b_reg, 0, smemB[write_stage_idx]);
            // use double buffer, only need one sync
            __syncthreads();
            // switch
            write_stage_idx ^= 1;
        }

        // load first tile from shared mem to register of next iter
        load_smem_tile_to_reg(smemA[load_stage_idx ^ 1], 0, a_reg[0]);
        load_smem_tile_to_reg(smemB[load_stage_idx ^ 1], 0, b_reg[0]);
        // compute last tile mma 8x8
        mma8x8(a_reg[1], b_reg[1], c);
    } while (i < K);

    store_c(c, C);
```

- reach 97.5% cublas

surpass cutlass

- SASS (Shader Assembly)
  - register bank conflict
  - register reuse

Cutlass GEMM Implementation

code structure

- interface

  `cutlass::gemm::device::Gemm<A_type,A_save,B_type,B_save,C_type,C_save>`

- select gemm shape

  > cutlass/include/cutlass/gemm/device/gemm.h

```
template <
```

```
    /// Element type for A matrix operand
    typename ElementA_,
    /// Layout type for A matrix operand
    typename LayoutA_,
    /// Element type for B matrix operand
    typename ElementB_,
    /// Layout type for B matrix operand
    typename LayoutB_,
    /// Element type for C and D matrix operands
    typename ElementC_,
    /// Layout type for C and D matrix operands
    typename LayoutC_,
    /// Element type for internal accumulation
    typename ElementAccumulator_ = ElementC_,
    /// Operator class tag
    typename OperatorClass_ = arch::OpClassSimt,
    /// Tag indicating architecture to tune for
    typename ArchTag_ = arch::Sm70,
    /// Threadblock-level tile size (concept: GemmShape)
    typename ThreadblockShape_ = typename DefaultGemmConfiguration<
        OperatorClass_, ArchTag_, ElementA_, ElementB_, ElementC_,
        ElementAccumulator_>::ThreadblockShape,
    /// Warp-level tile size (concept: GemmShape)
    typename WarpShape_ = typename DefaultGemmConfiguration<
        OperatorClass_, ArchTag_, ElementA_, ElementB_, ElementC_,
        ElementAccumulator_>::WarpShape,
    /// Instruction-level tile size (concept: GemmShape)
    typename InstructionShape_ = typename DefaultGemmConfiguration<
        OperatorClass_, ArchTag_, ElementA_, ElementB_, ElementC_,
        ElementAccumulator_>::InstructionShape,
    /// Epilogue output operator
    typename EpilogueOutputOp_ = typename DefaultGemmConfiguration<
        OperatorClass_, ArchTag_, ElementA_, ElementB_, ElementC_,
        ElementAccumulator_>::EpilogueOutputOp,
    /// Threadblock-level swizzling operator
    typename ThreadblockSwizzle_ =
        typename threadblock::GemmIdentityThreadblockSwizzle<>,
    /// Number of stages used in the pipelined mainloop
    int Stages =
        DefaultGemmConfiguration<OperatorClass_, ArchTag_, ElementA_,
ElementB_,
                                 ElementC_, ElementAccumulator_>::kStages,
    /// Access granularity of A matrix in units of elements
    int AlignmentA =
        DefaultGemmConfiguration<OperatorClass_, ArchTag_, ElementA_,
ElementB_,
                                 ElementC_,
ElementAccumulator_>::kAlignmentA,
    /// Access granularity of B matrix in units of elements
    int AlignmentB =
        DefaultGemmConfiguration<OperatorClass_, ArchTag_, ElementA_,
ElementB_,
                                 ElementC_,
ElementAccumulator_>::kAlignmentB,
    /// If true, kernel supports split-K with serial reduction
```

```
    bool SplitKSerial = false,
    /// Operation performed by GEMM
    typename Operator_ = typename DefaultGemmConfiguration<
        OperatorClass_, ArchTag_, ElementA_, ElementB_, ElementC_,
        ElementAccumulator_>::Operator,
    /// Gather operand A by using an index array
    bool GatherA = false,
    /// Gather operand B by using an index array
    bool GatherB = false,
    /// Scatter result D by using an index array
    bool ScatterD = false>
class Gemm ;
```

> cutlass/include/cutlass/gemm/device/default_gemm_configuration.h

```
template <
  typename OperatorClass,
  typename ArchTag,
  typename ElementA,
  typename ElementB,
  typename ElementC,
  typename ElementAccumulator
>
struct DefaultGemmConfiguration;
```

> Kstages: 2 KAlignmentA: 1 KAlignmentB: 1 KAlignmentC: 1

- kernel call

> cutlass/include/cutlass/gemm/device/gemm.h

```

Status operator()(cudaStream_t stream = nullptr) {
    return run(stream);
}

Status run(cudaStream_t stream = nullptr) {

    ThreadblockSwizzle threadblock_swizzle;

    dim3 grid =
threadblock_swizzle.get_grid_shape(params_.grid_tiled_shape);
    dim3 block(GemmKernel::kThreadCount, 1, 1);

    cudaError_t result;

    int smem_size = int(sizeof(typename GemmKernel::SharedStorage));

    if (smem_size >= (48 << 10)) {
      result = cudaFuncSetAttribute(Kernel<GemmKernel>,

cudaFuncAttributeMaxDynamicSharedMemorySize,
                                    smem_size);
```

```
      if (result != cudaSuccess) {
        return Status::kErrorInternal;
      }
    }

    // kernel call
    cutlass::Kernel<GemmKernel><<<grid, block, smem_size, stream>>>
(params_);
    // kernel call

    result = cudaGetLastError();

    return result == cudaSuccess ? Status::kSuccess :
Status::kErrorInternal;
}
```

> cutlass/include/cutlass/device_kernel.h

```
template <typename Operator>
__global__
void Kernel(typename Operator::Params params) {
  // Dynamic shared memory base pointer
  extern __shared__ int SharedStorageBase[];

  // Declare pointer to dynamic shared memory.
  typename Operator::SharedStorage *shared_storage =
      reinterpret_cast<typename Operator::SharedStorage *>
(SharedStorageBase);

  Operator op;

  op(params, *shared_storage);
}
```

- DefaultGemm structure

> cutlass/include/cutlass/gemm/kernel/default_gemm.h

```
template <
  /// Element type for A matrix operand
  typename ElementA,
  /// Layout type for A matrix operand
  typename LayoutA,
  /// Access granularity of A matrix in units of elements
  int kAlignmentA,
  /// Element type for B matrix operand
  typename ElementB,
  /// Layout type for B matrix operand
  typename LayoutB,
  /// Access granularity of B matrix in units of elements
  int kAlignmentB,
  /// Element type for C and D matrix operands
  typename ElementC,
  /// Element type for internal accumulation
```

```
  typename ElementAccumulator,
  /// Threadblock-level tile size (concept: GemmShape)
  typename ThreadblockShape,
  /// Warp-level tile size (concept: GemmShape)
  typename WarpShape,
  /// Warp-level tile size (concept: GemmShape)
  typename InstructionShape,
  /// Epilogue output operator
  typename EpilogueOutputOp,
  /// Threadblock-level swizzling operator
  typename ThreadblockSwizzle,
  /// If true, kernel is configured to support serial reduction in the
epilogue
  bool SplitKSerial,
  /// Operation performed by GEMM
  typename Operator,
  /// Use zfill or predicate for out-of-bound cp.async
  SharedMemoryClearOption SharedMemoryClear,
  /// Gather operand A by using an index array
  bool GatherA,
  /// Gather operand B by using an index array
  bool GatherB,
  /// Scatter result D by using an index array
  bool ScatterD
>
struct DefaultGemm<
  ElementA, LayoutA, kAlignmentA,
  ElementB, LayoutB, kAlignmentB,
  ElementC, layout::RowMajor,
  ElementAccumulator,
  arch::OpClassTensorOp,
  arch::Sm75,
  ThreadblockShape,
  WarpShape,
  InstructionShape,
  EpilogueOutputOp,
  ThreadblockSwizzle,
  2,
  SplitKSerial,
  Operator,
  SharedMemoryClear,
  GatherA,
  GatherB,
  ScatterD
> {

  /// Define the threadblock-scoped matrix multiply-accumulate
  using Mma = typename cutlass::gemm::threadblock::DefaultMma<
    ElementA,
    LayoutA,
    kAlignmentA,
    ElementB,
    LayoutB,
    kAlignmentB,
    ElementAccumulator,
```

```
    layout::RowMajor,
    arch::OpClassTensorOp,
    arch::Sm75,
    ThreadblockShape,
    WarpShape,
    InstructionShape,
    2,
    Operator,
    false,
    SharedMemoryClear,
    GatherA,
    GatherB
  >::ThreadblockMma;

  static const int kPartitionsK = ThreadblockShape::kK / WarpShape::kK;

  /// Define the epilogue
  using Epilogue = typename
cutlass::epilogue::threadblock::DefaultEpilogueTensorOp<
    ThreadblockShape,
    typename Mma::Operator,
    kPartitionsK,
    EpilogueOutputOp,
    EpilogueOutputOp::kCount,
    ScatterD
  >::Epilogue;

  /// Define the kernel-level GEMM operator.
  using GemmKernel = kernel::Gemm<Mma, Epilogue, ThreadblockSwizzle,
SplitKSerial>;
};
```

> cutlass/include/cutlass/gemm/kernel/gemm.h

```
CUTLASS_DEVICE
void operator()(Params const &params, SharedStorage &shared_storage) {

    // Compute threadblock location
    ThreadblockSwizzle threadblock_swizzle;

    cutlass::gemm::GemmCoord threadblock_tile_offset =
        threadblock_swizzle.get_tile_offset(params.swizzle_log_tile);

    // Early exit if CTA is out of range
    if (params.grid_tiled_shape.m() <= threadblock_tile_offset.m() ||
      params.grid_tiled_shape.n() <= threadblock_tile_offset.n()) {

      return;
    }

    // Compute initial location in logical coordinates
    cutlass::MatrixCoord tb_offset_A{
      threadblock_tile_offset.m() * Mma::Shape::kM,
      threadblock_tile_offset.k() * params.gemm_k_size,
    };
```

```
    cutlass::MatrixCoord tb_offset_B{
      threadblock_tile_offset.k() * params.gemm_k_size,
      threadblock_tile_offset.n() * Mma::Shape::kN
    };

    // Problem size is a function of threadblock index in the K dimension
    int problem_size_k = min(
      params.problem_size.k(),
      (threadblock_tile_offset.k() + 1) * params.gemm_k_size);

    // Compute threadblock-scoped matrix multiply-add
    int gemm_k_iterations = (problem_size_k - tb_offset_A.column() +
Mma::Shape::kK - 1) / Mma::Shape::kK;

    // Compute position within threadblock
    int thread_idx = threadIdx.x;

    // Construct iterators to A and B operands
    typename Mma::IteratorA iterator_A(
      params.params_A,
      params.ref_A.data(),
      {params.problem_size.m(), problem_size_k},
      thread_idx,
      tb_offset_A,
      params.gather_A_indices);

    typename Mma::IteratorB iterator_B(
      params.params_B,
      params.ref_B.data(),
      {problem_size_k, params.problem_size.n()},
      thread_idx,
      tb_offset_B,
      params.gather_B_indices);

    // Broadcast the warp_id computed by lane 0 to ensure dependent code
    // is compiled as warp-uniform.
    int warp_idx = __shfl_sync(0xffffffff, threadIdx.x / 32, 0);
    int lane_idx = threadIdx.x % 32;

    //
    // Main loop
    //

    // Construct thread-scoped matrix multiply
    Mma mma(shared_storage.main_loop, thread_idx, warp_idx, lane_idx);

    typename Mma::FragmentC accumulators;

    accumulators.clear();

    if (!kSplitKSerial || gemm_k_iterations > 0) {
      // Compute threadblock-scoped matrix multiply-add
      mma(gemm_k_iterations, accumulators, iterator_A, iterator_B,
accumulators);
```

```
    }

    //
    // Epilogue
    //

    OutputOp output_op(params.output_op);

    //
    // Masked tile iterators constructed from members
    //

    threadblock_tile_offset =
        threadblock_swizzle.get_tile_offset(params.swizzle_log_tile);

    //assume identity swizzle
    MatrixCoord threadblock_offset(
      threadblock_tile_offset.m() * Mma::Shape::kM,
      threadblock_tile_offset.n() * Mma::Shape::kN
    );

    int block_idx = threadblock_tile_offset.m() +
threadblock_tile_offset.n() * params.grid_tiled_shape.m();

    // Construct the semaphore.
    Semaphore semaphore(params.semaphore + block_idx, thread_idx);

    // If performing a reduction via split-K, fetch the initial
synchronization
    if (kSplitKSerial && params.grid_tiled_shape.k() > 1) {

      // Fetch the synchronization lock initially but do not block.
      semaphore.fetch();

      // Indicate which position in a serial reduction the output operator
is currently updating
      output_op.set_k_partition(threadblock_tile_offset.k(),
params.grid_tiled_shape.k());
    }

    // Tile iterator loading from source tensor.
    typename Epilogue::OutputTileIterator iterator_C(
      params.params_C,
      params.ref_C.data(),
      params.problem_size.mn(),
      thread_idx,
      threadblock_offset,
      params.scatter_D_indices
    );

    // Tile iterator writing to destination tensor.
    typename Epilogue::OutputTileIterator iterator_D(
      params.params_D,
      params.ref_D.data(),
      params.problem_size.mn(),
```

```
      thread_idx,
      threadblock_offset,
      params.scatter_D_indices
    );

    Epilogue epilogue(
      shared_storage.epilogue,
      thread_idx,
      warp_idx,
      lane_idx);

    // Wait on the semaphore - this latency may have been covered by
iterator construction
    if (kSplitKSerial && params.grid_tiled_shape.k() > 1) {

      // For subsequent threadblocks, the source matrix is held in the 'D'
tensor.
      if (threadblock_tile_offset.k()) {
        iterator_C = iterator_D;
      }

      semaphore.wait(threadblock_tile_offset.k());

    }

    // Execute the epilogue operator to update the destination tensor.
    epilogue(output_op, iterator_D, accumulators, iterator_C);

    //
    // Release the semaphore
    //

    if (kSplitKSerial && params.grid_tiled_shape.k() > 1) {

      int lock = 0;
      if (params.grid_tiled_shape.k() == threadblock_tile_offset.k() + 1) {

        // The final threadblock resets the semaphore for subsequent grids.
        lock = 0;
      }
      else {
        // Otherwise, the semaphore is incremented
        lock = threadblock_tile_offset.k() + 1;
      }

      semaphore.release(lock);
    }
}
```

> cutlass/include/cutlass/gemm/threadblock/mma_pipelined.h

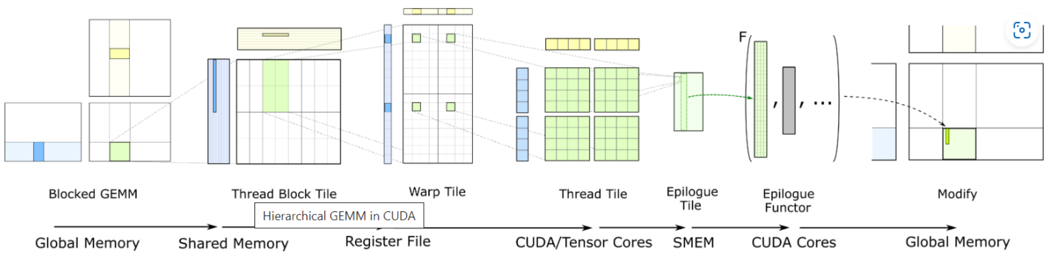
```
CUTLASS_DEVICE
  void operator()(
    int gemm_k_iterations,                          ///< number of
iterations of the mainloop
```

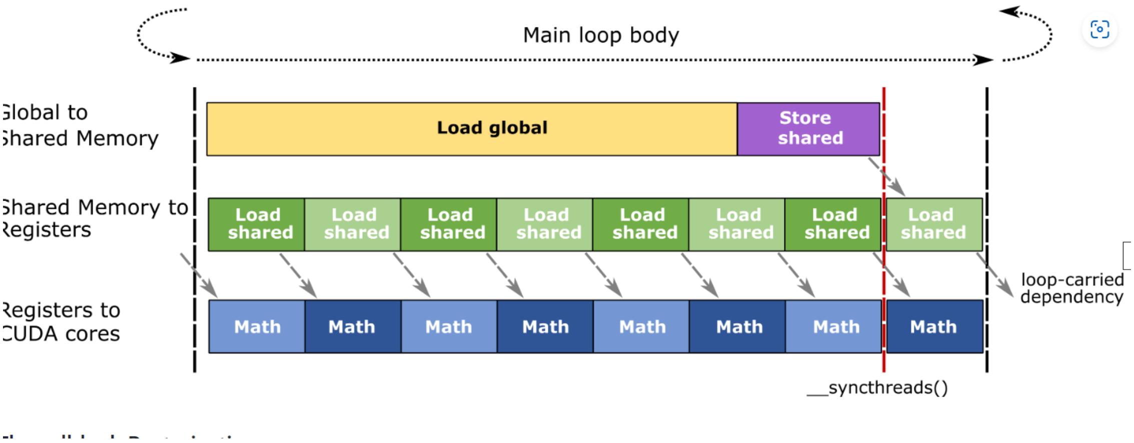
```
    FragmentC &accum,                                    ///< destination
accumulator tile
    IteratorA iterator_A,                                ///< iterator over A
operand in global memory
    IteratorB iterator_B,                                ///< iterator over B
operand in global memory
    FragmentC const &src_accum,                          ///< source
accumulator tile
    TransformA transform_A = TransformA(),               ///< transformation
applied to A fragment
    TransformB transform_B = TransformB()) {             ///< transformation
applied to B fragment

    //
    // Prologue
    //

    // Perform accumulation in the 'd' output operand
    accum = src_accum;

    FragmentA tb_frag_A;
    FragmentB tb_frag_B;

    tb_frag_A.clear();
    tb_frag_B.clear();

    // The last kblock is loaded in the prolog
    iterator_A.load(tb_frag_A);
    iterator_B.load(tb_frag_B);

    ++iterator_A;
    ++iterator_B;

    this->smem_iterator_A_.store(transform_A(tb_frag_A));
    this->smem_iterator_B_.store(transform_B(tb_frag_B));

    ++this->smem_iterator_A_;
    ++this->smem_iterator_B_;

    __syncthreads();

    // Pair of fragments used to overlap shared memory loads and math
instructions
    WarpFragmentA warp_frag_A[2];
    WarpFragmentB warp_frag_B[2];

    this->warp_tile_iterator_A_.set_kgroup_index(0);
    this->warp_tile_iterator_B_.set_kgroup_index(0);

    this->warp_tile_iterator_A_.load(warp_frag_A[0]);
    this->warp_tile_iterator_B_.load(warp_frag_B[0]);

    ++this->warp_tile_iterator_A_;
    ++this->warp_tile_iterator_B_;
```

```
    Operator warp_mma;

    int smem_write_stage_idx = 1;

    // Avoid reading out of bounds
    iterator_A.clear_mask(gemm_k_iterations <= 1);
    iterator_B.clear_mask(gemm_k_iterations <= 1);

    // Issue loads during the first warp-level matrix multiply-add *AFTER*
issuing
    // shared memory loads (which have the tighest latency requirement).

    //
    // Mainloop
    //

    // Note: The main loop does not support Base::kWarpGemmIterations == 2.
    CUTLASS_GEMM_LOOP
    for (; gemm_k_iterations > 0; --gemm_k_iterations) {
      //
      // Loop over GEMM K dimension
      //

      CUTLASS_PRAGMA_UNROLL
      for (int warp_mma_k = 0; warp_mma_k < Base::kWarpGemmIterations;
++warp_mma_k) {

        // Load warp-level tiles from shared memory, wrapping to k offset if
this is the last group
        // as the case may be.

        if (warp_mma_k == Base::kWarpGemmIterations - 1) {

          // Write fragments to shared memory
          this->smem_iterator_A_.store(transform_A(tb_frag_A));

          this->smem_iterator_B_.store(transform_B(tb_frag_B));

          __syncthreads();

          ++this->smem_iterator_A_;
          ++this->smem_iterator_B_;

          // Add negative offsets to return iterators to the 'start' of the
circular buffer in shared memory
          if (smem_write_stage_idx == 1) {
            this->smem_iterator_A_.add_tile_offset({0, -Base::kStages});
            this->smem_iterator_B_.add_tile_offset({-Base::kStages, 0});
          }
          else {
            this->warp_tile_iterator_A_.add_tile_offset(
                {0, -Base::kStages * Policy::kPartitionsK *
Base::kWarpGemmIterations});
            this->warp_tile_iterator_B_.add_tile_offset(
```

```
                {-Base::kStages * Policy::kPartitionsK *
Base::kWarpGemmIterations,
                 0});
          }

          smem_write_stage_idx ^= 1;
        }

        this->warp_tile_iterator_A_.set_kgroup_index((warp_mma_k + 1) %
Base::kWarpGemmIterations);
        this->warp_tile_iterator_B_.set_kgroup_index((warp_mma_k + 1) %
Base::kWarpGemmIterations);

        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]);

        ++this->warp_tile_iterator_A_;
        ++this->warp_tile_iterator_B_;

        if (warp_mma_k == 0) {

          iterator_A.load(tb_frag_A);
          iterator_B.load(tb_frag_B);

          ++iterator_A;
          ++iterator_B;

          // Avoid reading out of bounds if this was the last loop iteration
          iterator_A.clear_mask(gemm_k_iterations <= 2);
          iterator_B.clear_mask(gemm_k_iterations <= 2);
        }

        warp_mma(accum, warp_frag_A[warp_mma_k % 2],
                 warp_frag_B[warp_mma_k % 2], accum);
      }
    }

  }
```

- analysis on code





Reference

- CUTLASS: Fast Linear Algebra in CUDA C++ | NVIDIA Technical Blog
  CUTLASS: Fast Linear Algebra in CUDA C++ - 知乎 (zhihu.com) (translated version)
- 深入浅出GPU优化系列：GEMM优化（一） - 知乎 (zhihu.com)
- 深入浅出GPU优化系列：GEMM优化（二） - 知乎 (zhihu.com)
- 深入浅出GPU优化系列：GEMM优化（三） - 知乎 (zhihu.com)
- SGEMM · NervanaSystems/maxas Wiki (github.com)
- CUDA 矩阵乘法终极优化指南 - 知乎 (zhihu.com)
- cuda_sgemm/gemm.cu at master · niuhope/cuda_sgemm (github.com)
- Why use CUTLASS instead of CUBLAS for GEMM? What are the advantages of CUTLASS ? · Issue #109 · NVIDIA/cutlass (github.com)
- Liu-xiandong/How_to_optimize_in_GPU: This is a series of GPU optimization topics. Here we will introduce how to optimize the CUDA kernel in detail. I will introduce several basic kernel optimizations, including: elementwise, reduce, sgemv, sgemm, etc. The performance of these kernels is basically at or near the theoretical limit. (github.com)
- (27 封私信 / 80 条消息) 自己写的CUDA矩阵乘法能优化到多快？ - 知乎 (zhihu.com)
- CUDA矩阵乘法的优化 · wu-kan
- cutlass/efficient_gemm.md at master · NVIDIA/cutlass (github.com)