# CUTLASS: Fast Linear Algebra in CUDA C++

## 写在最前面

Scott Grey提供了一套成熟有效且里程碑式的CUDA 汇编工具：MaxAs。这个工具提供了一个矩阵乘法的汇编demo，除了介绍如何使用MaxAs以外也介绍了矩阵乘法的GPU优化思路，XiaoyuWang将其译为中文，而一年多以前我在这篇文章的指导下也尝试来理解汇编层面的GPU SGEMM优化，后将其汇编代码进行注释（GitHub仓库）以便于对照Scott Grey提供的伪代码加深自己的理解。

本文以GEMM为切入点介绍了若干优化思路，但如果单纯的希望理解GEMM的GPU优化思路我认为以上提及的资料已经足够学习了。近期因为工作的需要，需要了解和使用CUTLASS，在网上一通搜索发现可用的中文资料实属有限。本着奉献大家巩固知识的目的，我将NVIDIA blog上的一篇自认为质量不错的CUTLASS introduction进行了一些翻译，希望能为热爱CUDA的中文母语读者提供一些帮助。本人能力有限，如有翻译错误欢迎指正。

## CUTLASS: Fast Linear Algebra in CUDA C++

矩阵乘法是若干科学应用—尤其是深度学习—中的重要部分。很多现代神经网络中的操作是用或是可以被表示成矩阵运算操作。

以NVIDIA cuDNN库为例，其中的卷积神经网络使用了多种多样的矩阵乘法，这些矩阵乘法有比如传统的直接卷积，如image-to-column处理的图像和卷积核的矩阵乘法操作[1]；又比如基于快速傅立叶变换（Fast Fourier Transforms, FFT[2]）卷积操作中的core routine操作抑或是Winograd方法[3]。

在开发cuDNN的过程中，我们首先从cuBLAS中的高性能通用矩阵乘法（general matrix multiplication, GEMM）的实现开始出发，补充和定制它们来实现高效卷积运算。今天，我们对于这些 GEMM 策略和算法的理解对于为深度学习中的许多不同问题和应用实现最佳的性能至关重要。

有了CUTLASS，我们可以给所有人一种像搭积木一样的技术和结构，通过用CUDA C++编写含有高性能GEMM的新算法。稠密线性代数的灵活高效运用对于深度学习乃至更广阔GPU计算生态系统起至关重要的作用。

# Introducing CUTLASS

今天，我们向大家介绍CUTLASS（CUDA Templates for Linear Algebra Subroutines），CUTLASS是一个基于CUDA C++模板和抽象的为了使用CUDA kernel实现各个层级和尺度的高性能GEMM计算代码包。与其他的一些稠密线性代数GPU模板库（如MAGMA[4]）不同的是，CUTLASS的设计初衷是将GEMM中一些“可变的部分”分解成若干C++抽象模板实现的基础组建，这种设计可以使开发者轻松的定制到他们自己的CUDA kernel中。我们同时将我们的CUTLASS开源到GitHub，希望能让大家使用模板库实现CUDA GEMM技术。

我们的CUTLASS源码包含了对混合精度计算操作的广泛支持，提供了专门用于处理8bit整型、半精度浮点（FP16）、单精度浮点（FP32）以及双精度浮点（FP64）的数据移动和乘-累加的代码抽象。CUTLASS最令人兴奋的功能莫属能利用图灵架构Tensor Core加速的WMMA API来实现矩阵乘法运算。Tesla V100的这种可编程矩阵乘-累加单元—Tensor Core—能取得125 Tensor TFLOP/s的超高性能。

# Efficient Matrix Multiplication on GPUs

GEMM就是指计算 $C=\alpha A*B+\beta C$ $\textbf{C}=\alpha \textbf{A} * \textbf{B} + \beta \textbf{C}$ ，其中 $A$ $\textbf{A}$ 、 $B$ $\textbf{B}$ 和 $C$ $\textbf{C}$ 都是矩阵， $A$ $\textbf{A}$ 是 $M\times K$ $M \times K$ 的矩阵， $B$ $\textbf{B}$ 是 $K\times N$ $K \times N$ 的矩阵， $C$ $\textbf{C}$ 是 $M\times N$ $M \times N$ 的矩阵。为了简化，我们在后续的介绍中都假定 $\alpha=\beta=1$ $\alpha = \beta = 1$ 。接下来我们将展示如何利用CUTLASS支持的任意尺度函数实现定制的元素操作。

最简单的实现包含了三层循环嵌套：

```
for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j)
        for (int k = 0; k < K; ++k)
            C[i][j] += A[i][k] * B[k][j];
```

矩阵 $C$ $\textbf{C}$ 中位于 $(i,j)$ $(i, j)$ 的元素就是矩阵 $A$ $\textbf{A}$ 的第 $i$ $i$ 行和矩阵 $B$ $\textbf{B}$ 的第 $j$ $j$ 列两 $K$ $K$ 维向量的内积结果。理想情况下性能应当受限于算术吞吐量，确切来说对于一个大型方阵，即 $M=N=K$ $M = N = K$ ，矩阵乘法的数学操作复杂度为 $O(N3)$ $O(N^3)$ ，而数据量为 $O(N2)$ $O(N^2)$ ，同时计算强度为 $N$ $N$ （Jie: 这里的计算强度可以被理解为同一个数据被重复利用的次数）。然而能够利用好理论计算强度要求重复使用每个元素 $O(N)$ $O(N)$ 次。不幸的是，为了重复利用元素，上述“内积”算法需要我们在片上高速缓存（Jie: 如NVIDIA设备中的

Shared Memory）占据一大块空间，因此在 $M$ 、 $N$ 和 $K$ 增大的时候无法实现。

好在我们可以颠倒循环的嵌套顺序，即通过将最内层 $K$ 次的循环提到最外层，取得好一些的性能。这种实现在最外层的一次循环中只需要加载一次矩阵 $\mathbf{A}$ 的列和矩阵 $\mathbf{B}$的行，*然后进行外层乘法*计算并把结果累加到矩阵 $\mathbf{C}$中。在此之后矩阵 $\mathbf{A}$的这一列和矩阵 $\mathbf{B}$的这一行就再也不会用到了。

```
for (int k = 0; k < K; ++k)     // 现在K次循环位于最外层了
    for (int i = 0; i < M; ++i)
        for (int j = 0; j < N; ++j)
            C[i][j] += A[i][K] * B[k][j];
```

这种实现的一个问题是它要求所有矩阵 $\mathbf{C}$中的 $M\times N$ 个元素都处于活动状态，最好以储存每条乘-累加指令的结果，理想情况是内存写入速度和乘-累加指令执行速度一样快，但我们可以通过将矩阵 $\mathbf{C}$分成若干 $Mtile\times Ntile$ 矩阵片（tile）来减小工作集的大小（Jie: 这里我理解是因为之前说同一次最外层循环只加载一次矩阵 $\mathbf{A}$的列和矩阵 $\mathbf{B}$的行，假设加载的是 $\mathbf{A}$的第 $k$ 整列及 $\mathbf{B}$的第 $k$ 整行，这二者的计算实际上会在矩阵 $\mathbf{C}$中 $M*N$ *个元素中的每一个进行自加操作，因此这里提到“要求所有矩阵* $\mathbf{C}$*中的* $M\times N$ *个元素都处于活动状态”，如果进行拆分成tile的话就不要一次读入整行整列）。我们对每个tile都执行*外层乘法，因此有以下代码：

```
for (int m = 0; m < M; m += Mtile)               // 沿着M维迭代
    for (int n = 0; n < N; n += Ntile)           // 沿着N维迭代
        for (int k = 0; k < K; ++k)
            for (int i = 0; i < Mtile; ++i)      // 计算一个tile
                for (int j = 0; j < Ntile; ++j) {
                    int row = m + i;
                    int col = n + j;
                    C[row][col] += A[row][k] * B[k][col];
                }
```

对于矩阵 $\mathbf{C}$中的每一个tile，矩阵 $\mathbf{A}$与矩阵 $\mathbf{B}$中的tile只需要读取一次，这样就可以达到 $O(N)$ 的计算强度。矩阵 $\mathbf{C}$ tile的大小的是依据L1缓存或是寄存器数量的上限决定的。这样外层的循环就能被简单的并行化了，这是个巨大的提升！

进一步的重构为利用局部性和并行性提供了更多的基础。我们可以通过在块中沿着维度 $K$ 累加矩阵的积而不必只累加向量外层的积（Jie: 这里我理解还是在说拆分成tile执行累加免于一下子加载整个矩阵 $\mathbf{C}$）。我们常称此概念为**矩阵乘法累加 (accumulating matrix products, AMP)** 。

## Hierarchical GEMM Structure

CUTLASS通过将计算过程分解成**线程块片（thread block tile）**、**线程束片（warp tile）**和**线程片（thread tile）**的层次结构并将AMP的策略应用于此层次结构来高效率的完成基于GPU的拆分成tile的GEMM。这个层次结构紧密地反映了NVIDIA CUDA编程模型，如图1所示。你可以看到从global memory到shared memory的数据移动（矩阵到thread block tile）；从shared memory到寄存器的数据移动（thread block tile到warp tile）；从寄存器到CUDA core的计算（warp tile到thread tile）。

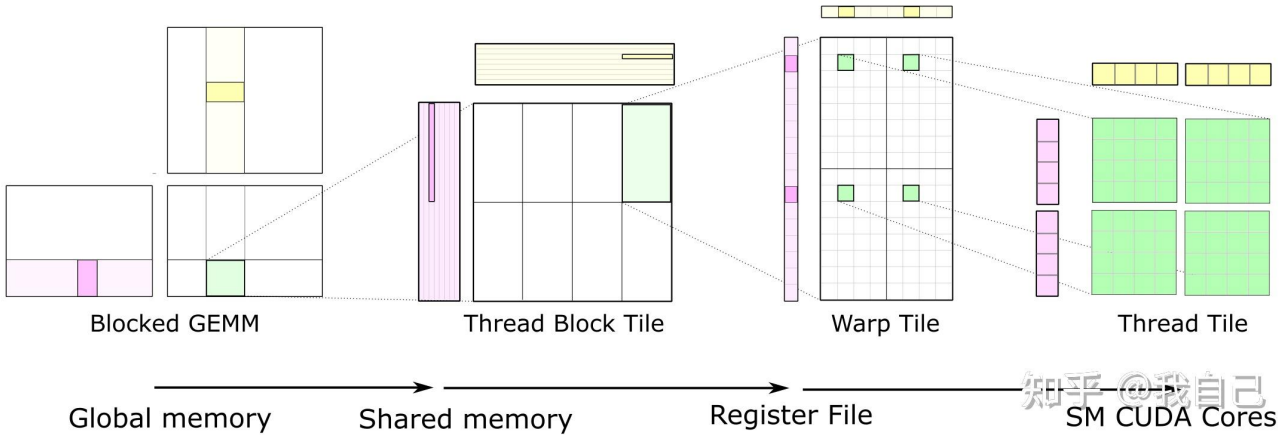


*图1，完整的GEMM层次将数据从低速存储器转移到高速存储器，高速存储器会在许多数学运算中大量复用。*

## Thread Block Tile

每个thread block通过不停从输入矩阵中读取一块矩阵数据并计算一次累计矩阵乘法（$\mathbf{C} \mathrel{+}= \mathbf{A} * \mathbf{B}$）。图2展示了一个thread block的计算同时高亮出一次主循环用到的数据块。

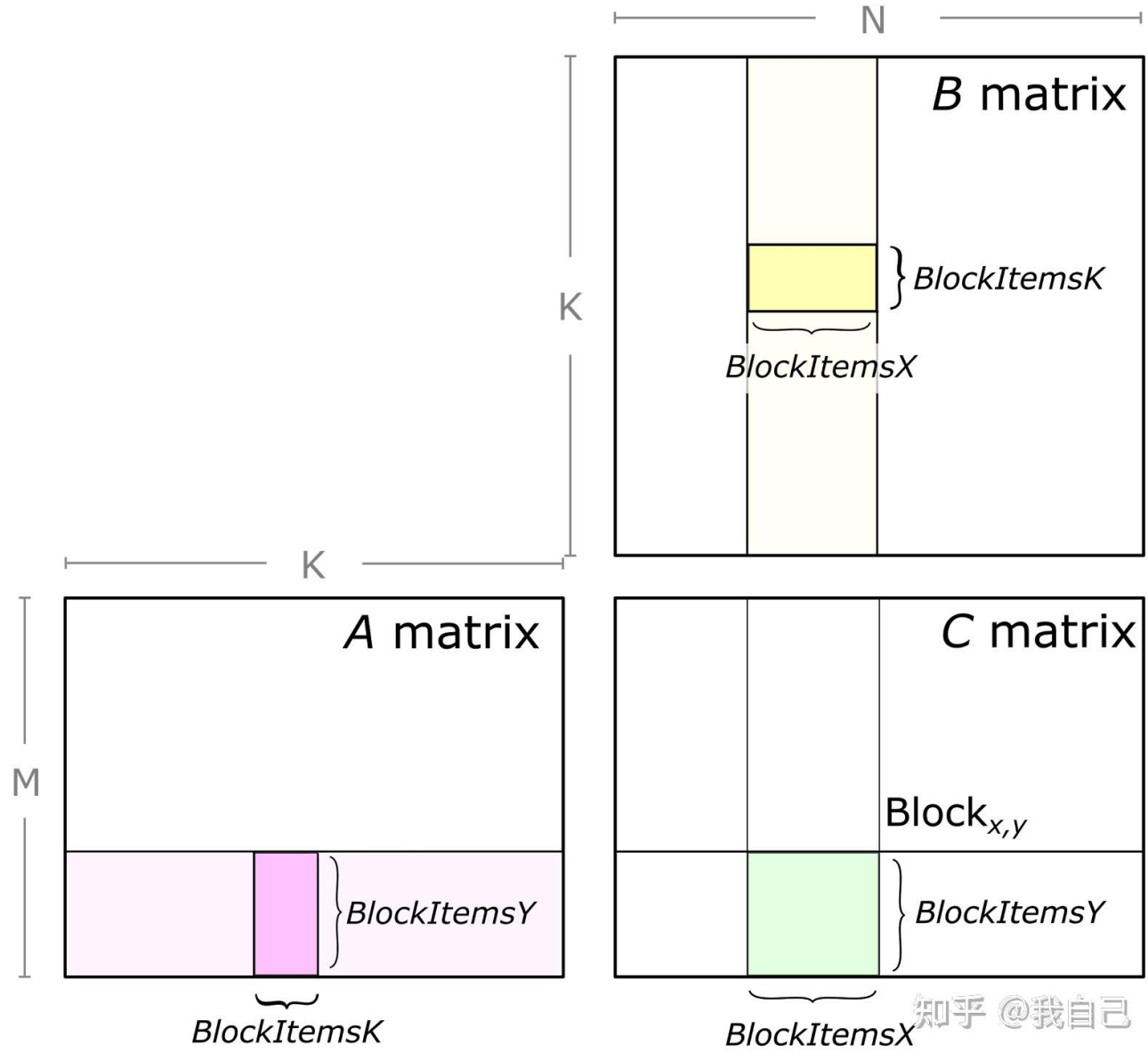


*图2，GEMM问题可以被拆解到一个thread block进行的计算。矩阵* C\mathbf{C} *的子矩阵用绿色画出。此子矩阵是由矩阵* A\mathbf{A}*的tile和矩阵* B\mathbf{B} *的子矩阵做矩阵乘法得到的。具体来说是沿着维度* KK *将矩阵划成若干tile，对每个tile都进行矩阵乘法并累加结果得到。*

CUDA thread block tile接下来又会被拆分到线程束中（warp，一组执行相同SIMT指令的线程）。线程束能帮助组织GEMM的计算过程同时还是WMMA API的显示部分，稍后将详细介绍。

图3展示了thread block层次的矩阵乘法细节。矩阵 A\textbf{A}和矩阵B\textbf{B}的tile从global memory中被加载到shared memory中，这样就能被同一个thread block中的warp互相访问。如图3所示，这个thread block输出的tile按照空间又被划分给若干warp。我们将输出tile的这块存储器称为*累加器（accumulators）*，原因是它存储了矩阵乘法累加结果。由于每个accumulator每次数学运算都会被更新一次，因此他需要驻留在SM中最快的内存：寄存器。

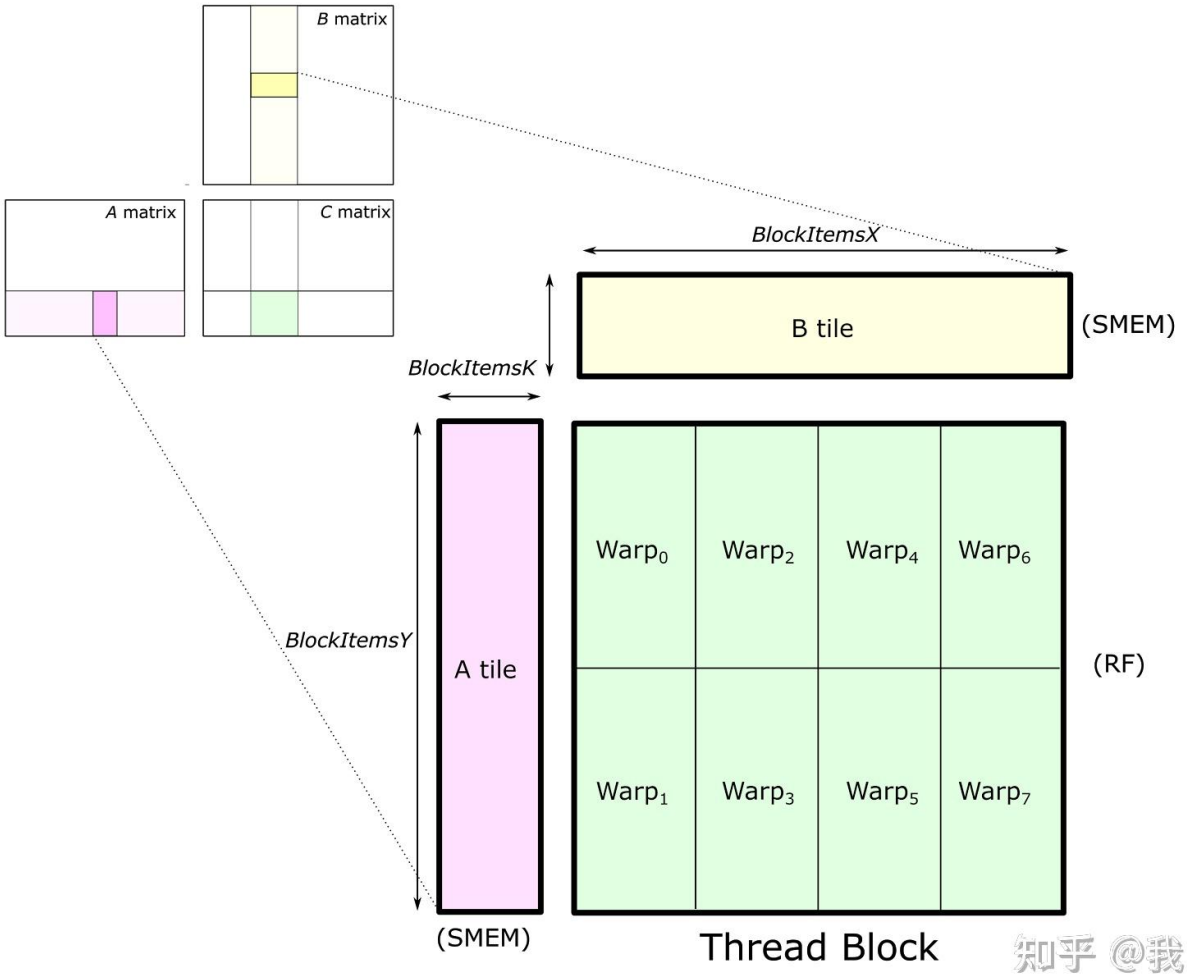


*图3，thread block结构将矩阵 C\mathbf{C}的tile拆分到若干warp中进行后续计算，每个warp存储一块彼此互不重叠的2D tile。每个warp用寄存器存储它自己的对应的accumulator元素。矩阵 A\textbf{A}和矩阵 B\textbf{B}中的tile都是被存储到shared memory，当然此shared memory可以被同一个thread block的warp访问。*

参数 BlockItems{X,Y,Z}BlockItems\{X, Y, Z\} 是开发者指定的编译时（compile-time）常量，它被用来调整目标处理器的GEMM计算和特定GEMM的配置（如 M,N,KM, N, K 数据类型等）。图中我们用了一个8 warp（256 thread）的thread block来介绍实现，通常这是CUTLASS大型SGEMM（FP32 GEMM）的tile尺寸的典型配置。

## warp Tile

一旦数据储存到了shared memory，每个warp都会计算通过沿着维度 KK 迭代计算一系列矩阵乘法累加，从shared memory加载子矩阵（或称片段，fragment），

然后计算外层乘法的累加。图4给了一个详细的图示。fragment的尺寸通常比整个维度 KK 小很多，这么设置的目的是最大化从shared memory加载的数据的计算强度从而避免shared memory的带宽成为瓶颈。

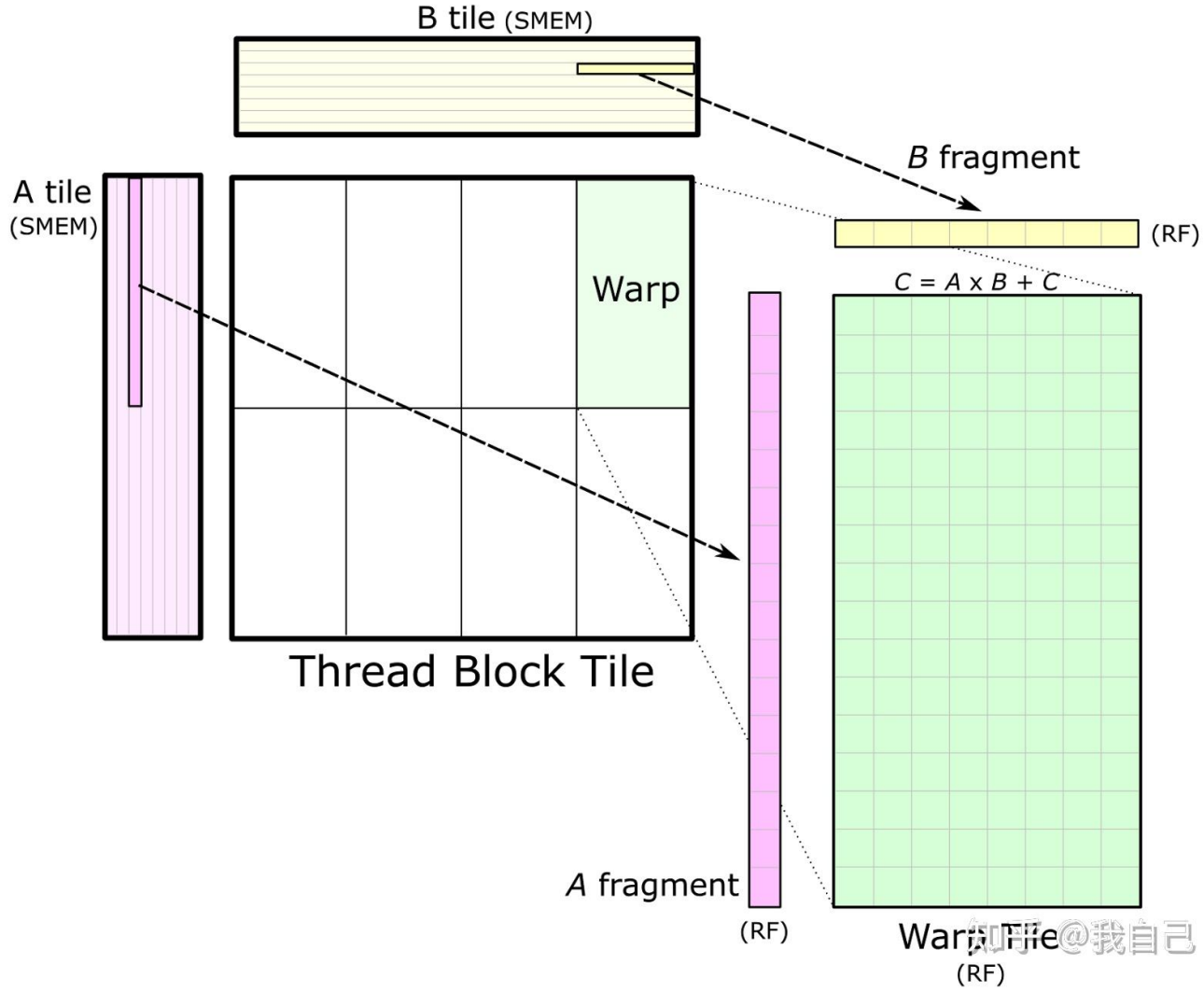


*图4，单个warp通过不停从相应的shared memory（SMEM）将数据（矩阵 A\textbf{A}和矩阵 B\textbf{B}的fragment）加载到寄存器（RF）来计算矩阵乘法累加，最后计算外层乘法。*

图4还描述了几个warp之间shared memory的共享。处于同一thread block中的warp，有相同行号的warp读取矩阵 A\textbf{A}中的相同fragment，有相同列号的warp读取矩阵 B\textbf{B}中的相同fragment。

我们注意到，以warp为中心的GEMM结构能够高效的实现高性能GEMM kernel而**不依赖**隐式的warp同步。CUTLASS GEMM kernel的实现在恰当的时候调用__syncthreads()而实现了良好的同步。

## Thread Tile

CUDA编程模型是定义在thread block和单个thread上的。因此，warp结构实际是被映射到单个线程的操作上。线程无法访问彼此的寄存器，因此我们必须选择一种组织方式，这种组织方式需要能够让寄存器保存的数值被同一线程的多条数学指令复用。这就要求单个thread之内进一步存在2D tile结构，如图5所示。每个thread向CUDA core发射（issue）一串独立的数学指令并计算外层乘积的累加。

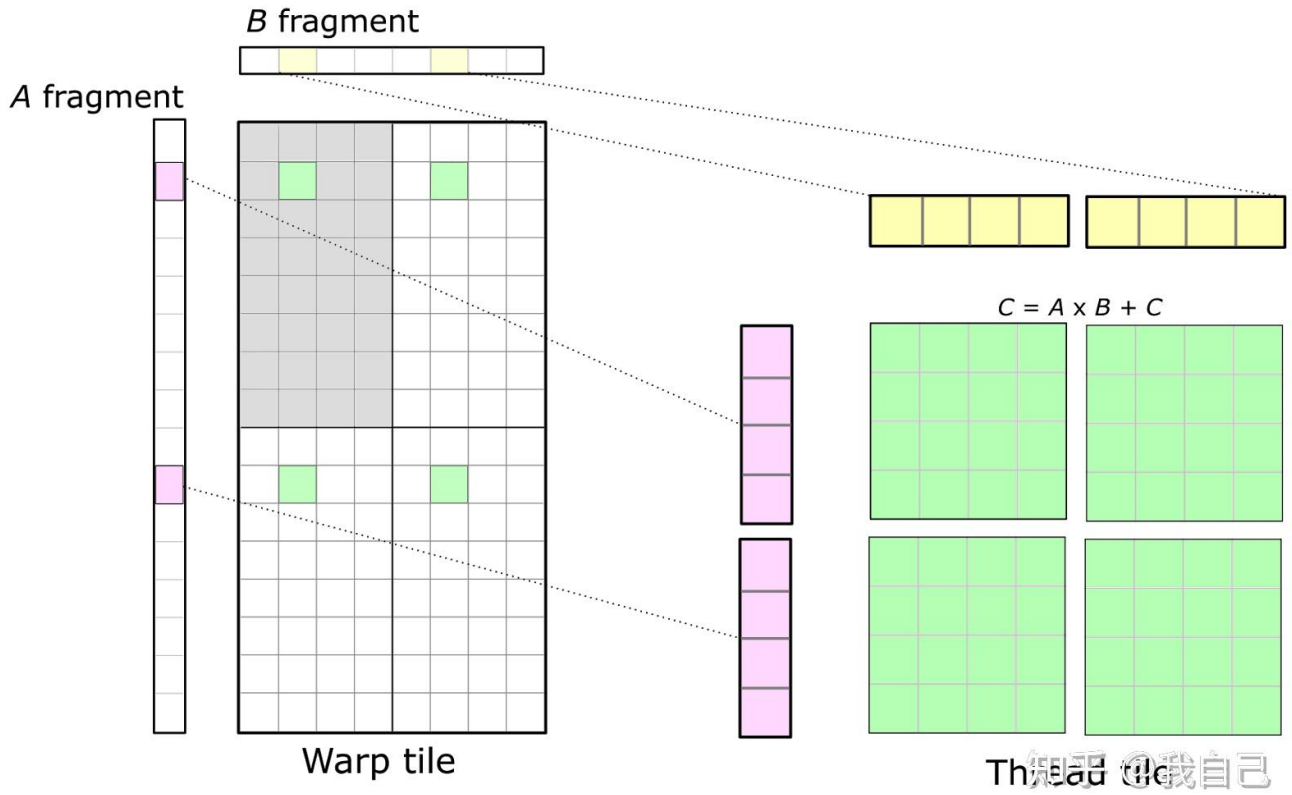


*图5，一个独立的线程（右侧）通过计算矩阵 $\textbf{A}$和矩阵 $\textbf{B}$保存在其寄存器的fragment来参与一次warp级的矩阵乘法（左侧）。warp的accumulator的一个拆分用绿色表示，它是warp中的一些thread，通常被用一组2D tile的形式排布。*

在图5中，warp tile左上角四分之一的部分被涂成灰色。灰色区域的32个块代表一个warp之内的32个线程。这种排列将会导致多个线程在同一行或在同一列，也就是说会有若干线程去取矩阵 $\textbf{A}$或是矩阵 $\textbf{B}$中fragment里的同一个元素。为了最大化计算强度，这种简单的结构可以被复制构成完整的warp级的accumulator tile，即整个 $8\times8$ 的thread tile由一个 $8\times1$ 和 $1\times8$ 的外层乘法得到。可以用绿色的四个accumulator tile说明此算法。

## WMMA GEMM

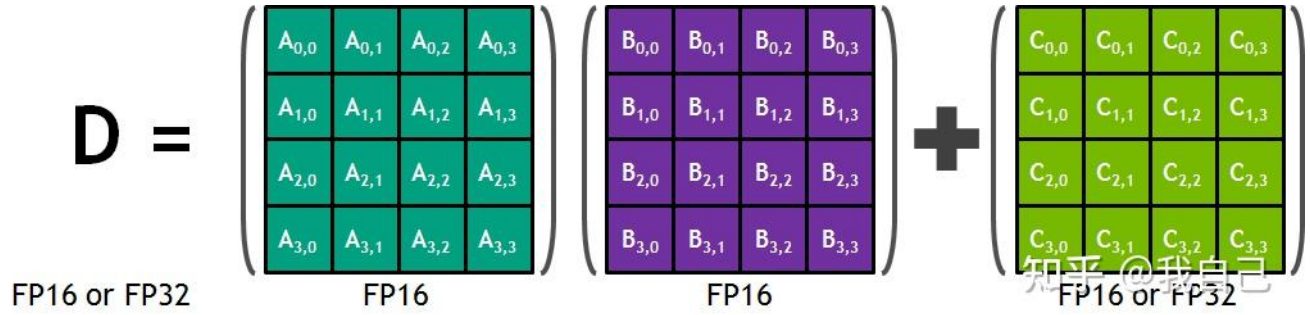


*图6，一次WMMA计算* $D=A*B+C$ $\textbf{D} = \textbf{A} \ast \textbf{B} + \textbf{C}$ *，其中* $A$ $\textbf{A}$、 $B$ $\textbf{B}$、 $C$ $\textbf{C}$、 $D$ $\textbf{D}$*都是矩阵。*

实际上，WMMA API是一个thread tile结构的替代品，这里的thread tile就是上文介绍的用来执行warp范围矩阵乘-累加操作的结构。WMMA API为开发者提供了warp内合作（warp-cooperative）的矩阵fragment读写和数学乘-累加操作的代码抽象，而不必将warp tile拆解成只被单个thread占据的纯量（scalar）和向量元素（Jie: 这里应该说WMMA编程层次是在wrap级）。

图7展示了针对CUDA WMMA API的warp tile结构。调用wmma::load_matrix_sync将会将矩阵 $A$ $\textbf{A}$和矩阵 $B$ $\textbf{B}$中的fragment加载到模板类nvcuda::wmma::fragment<>的实例中，而wrap tile的若干accumulator元素则是以nvcuda::wmma::fragment<accumulator>矩阵的形式组织的。这些fragment存储一个分布在warp的thread中的一个2D矩阵。最终，对每个accumulator frament（及相对应的矩阵 $A$ $\textbf{A}$和矩阵 $B$ $\textbf{B}$的fragment）调用nvcuda::wmma::mma_sync()来使用Tensor Core计算warp范围的矩阵乘-累加操作。

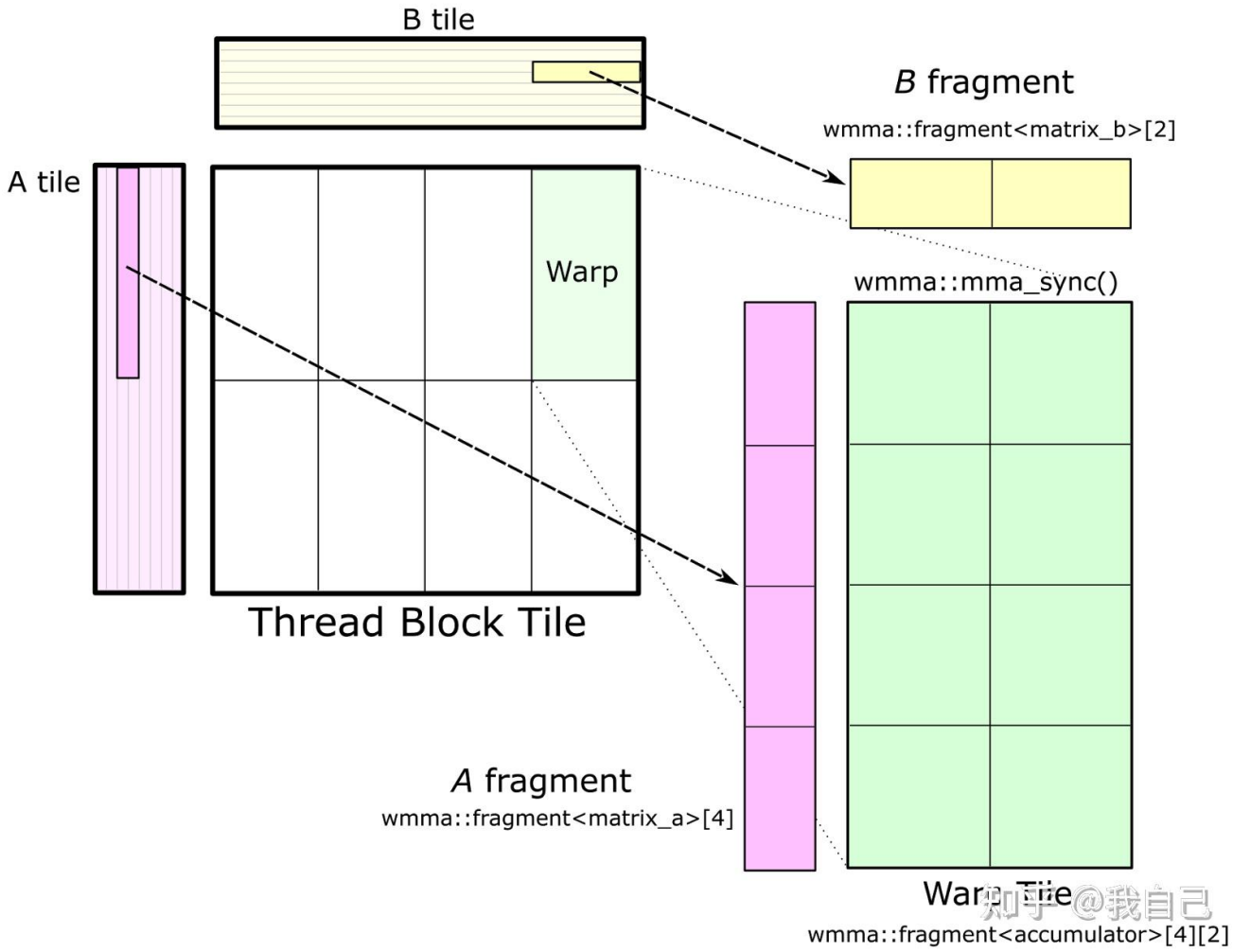


*图7，warp tile结构可以使用CUDA WMMA API应用在图灵V100的Tensor Core上。该API提供加载矩阵fragment和执行矩阵乘-累加的代码抽象。*

CUTLASS在block_task_wmma.h文件中实现了基于WMMA API的GEMM。warp tile的维度必须为由nvcuda::wmma模板定义的矩阵乘-累加的尺寸（依据CUDA Compute Capability决定）整数倍。在CUDA 9.0中，基本的WMMA 尺寸为 $16 \times 16 \times 16$ 。

## Complete GEMM

完整的GEMM结构可以用thread block中的thread执行嵌套循环来表示，具体如下。除了最外层的main循环之外，所有的循环都有恒定的循环次数且可以被编译器执行循环展开。为了简介期间，这里省略了地址和索引的计算，详细内容可以参阅CUTLASS的源码。

```
// 用于计算thread block的矩阵乘法累加的Device函数
__device__ void block_matrix_product(int K_dim) {

    // Fragments被用来存储来自SMEM的数据
    value_t frag_a[ThreadItemsY];
    value_t frag_b[ThreadItemsX];
```

```
    // Accumulator阵列
    accum_t accumulator[ThreadItemsX][ThreadItemsY];

    // GEMM主函数 - 沿着整个维度K迭代 - 不能循环展开
    for (int kblock = 0; kblock < K_dim; kblock += BlockItemsK) {

        // 将A和B的tiles从global memory加载到SMEM
        //
        // (为了简洁没有过多展示 - 具体见CUTLASS源码)
        ...

        __syncthreads();

        // Warp tile结构 - 对Thread Block tile进行循环迭代
        #pragma unroll
        for (int warp_k = 0; warp_k < BlockItemsK; warp_k += WarpItemsK) {

            // 将SMEM中对应第k个的数据frag_a和frag_b取出
            //
            // (为了简洁没有过多展示 - 具体见CUTLASS源码)
            ...

            // Thread tile结构 - 对外层乘法累加
            #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();
    }
}
```

`WarpItemsK`代表目标数学运算的点乘的尺寸。对于SGEMM（FP32）、DGEMM（FP64）和HGEMM（FP16）来说，点乘的长度为1的纯量（scalar）乘-累加指令。对于IGEMM（8-bit 整型GEMM），CUTLASS则使用`WarpItemsK`为4的4元素整型点乘指令（four-element integer dot product instruction, IDP4A）。对于基于WMMA实现的GEMM，我们选择`wmma::fragment<>`模板类的 KK 维度，目前`WarpItemsK`被定义为16。

## Software Pipelining

tile化的矩阵乘法大量使用寄存器来保存fragment和accmulator的tile，同时还需要分配大块的shared memory。相对于片上的存储空间，如此高的需求将会限制占

有率（occupancy，即在一个SM上最大的thread block同时执行的数量）。因此，与GPU典型的计算负载相比，GEMM的实现可以用更少的warp和thread block在SM上执行。如图8所示，我们使用程序流水线来隐藏数据传输延迟，具体来说是通过在一次循环并发地执行GEMM层次结构的所有阶段与在下一次迭代期间将每个阶段的输出结果喂给相关的阶段。

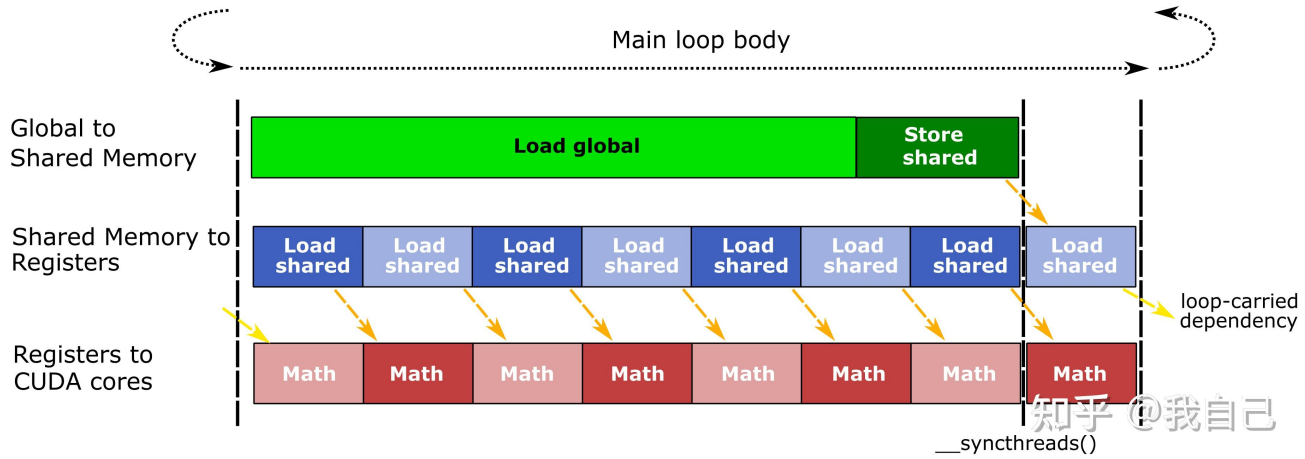


*图8，三条并发的指令流在CUTLASS的GEMM主循环中交织，橙色箭头表示数据依赖。当内存系统从global memory加载数据，SM为下一个thread tile加载fragment的时候，thread通过为当前tile执行数学指令来维持SM忙碌的状态（Jie：就是数据读取和数学指令执行彼此重叠）。*

GEMM CUDA kernel在流水线中发射三个并发的指令流，这三个指令流对应着GEMM层次结构（图1）的数据流的若干阶段。图8中每个阶段的相对大小代表指令延迟的长短，橙色箭头高亮出了每个流的数据依赖。在数据存储到shared memory后对__syncthreads()的调用将会同步所有的warp，这样就会避免了对shared memory的数据竞争。最后流水线的数学阶段与从shared memory的读取相重叠，读取的数据将会被喂到下一次主循环的第一个数学阶段。

实际上，CUDA开发者通过在程序代码中将CUDA各个阶段的CUDA语句彼此交错并依赖CUDA编译器在编译的代码中恰当地调度发射指令来实现指令级的指令并发。大量的使用#pragma unroll和编译时常数能够使CUDA编译器展开循环并将阵列元素与寄存器绑定，这二者对于性能调优都至关重要。具体例子见block_task::consume_tile()。

我们在GEMM分层结构的每一层中使用双缓冲使上游流水线能够将数据写到shared memory或寄存器而相以来的流水线从自己的元素读取。值得注意的是，这样就消除了第二个__syncthreads()因为对shared memory的读和写不是同一个位置。双缓冲的代价是shared memory和用于临时储存shared memory的寄存器的使用都需要加倍。

对延迟的隐藏实际与thread block、warp和thread tile的尺寸以及SM上活跃的数学计算单元的吞吐量都相关。虽然较大的tile能通通过增加数据复用来隐藏更多的

延迟，但SM上寄存器的数量和shared memory的容量限制了tile的最大值。幸运的是NVIDIA GPU有足够的储存资源来对足够大的GEMM tile操作。

# CUTLASS

CUTLASS是一个层次化GEMM结构的CUDA C++模板类的实现。我们打算将这些模板类包含在现有的设备端CUDA kernel和函数中，但为了方便上手和运行我们也提供一个简单的kernel和执行结构。类似于CUB，大量的模板参数和编译时常数的使用让CUTLASS具有可调性和灵活性。

CUTLASS实现了高性能GEMM的实现需要的操作的抽象。具体化的*tile loader*将数据高效地从global memory加载到shared memory，在适应原始数据的布局的同时坚固效率，同时还能无bank conlifct地加载到集村其中。对于某些布局，IGEMM需要将数据为CUDA的4元素整型点乘指令进行一些重构，重构操作在数据存到SMEM的时候就会完成。

## CUTLASS GEMM Device Functions

下面的来自dispatch.h的例子定义了一个block_task类型并且实例化了一个单精度列优先矩阵的GEMM。block_task_policy_t定义GEMM tile尺寸并会在下一节详细说明。

```
/// CUTLASS SGEMM example
__global__ void gemm_kernel(
    float *C,
    float const *A,
    float const *B,
    int M,
    int N,
    int K) {

    // 定义GEMM tile的尺寸 - 下一节讨论
    typedef block_task_policy <
        128, // BlockItemsY: 一个tile一行的高
        32, // BlockItemsX - 一个tile一列的宽
        8, // ThreadItemsY - 一个thread-tile一行的高
        4, // ThreadItemsX - 一个thread-tile一列的宽
        8, // BlockItemsK - 一个的tile的深度
        true, // UseDoubleScratchTiles - 是否使用双缓冲SMEM
        block_raster_enum::Default // Block光栅化策略
    > block_task_policy_t;

    // 定义epilogue functor，epilogue functor的作用见下文
    typedef gemm::blas_scaled_epilogue<float, float, float> epilogue_op_t ;

    // 定义block_task
    typedef block_task <
```

```
        block_task_policy_t,
        float,
        float,
        matrix_transform_t::NonTranspose,
        4,
        matrix_transform_t::NonTranspose,
        4,
        epilogue_op_t,
        4,
        true
    > block_task_t;

    // 声明静态分配的shared storage
    __shared__ block_task_t::scratch_storage_t smem;

    // 构建与执行
    block_task_t(
        reinterpret_cast(&smem),
        &smem,
        A,
        B,
        C,
        epilogue_op_t(1, 0),
        M,
        N,
        K).run();
}
```

分配的shared memorysmem被实例化的block_task_t用来储存block级tile的矩阵运算数据。

epologue_op_t是一个在矩阵乘法运算结束后的用来更新输出矩阵的functor，这个functor会被作为（block_task_t的）模板参数。这种设计让我们能轻松的将矩阵乘法和用户定义的元素操作相结合，详细内容见后文。CUTLASS提供gemm::blas_scaled_epilogue函子来计算GEMM操作$C=\alpha*AB+\beta*C$ $\textbf{C} = \alpha * \textbf{A}\textbf{B} + \beta * \textbf{C}$ （在(epilogue_function.h被定义）。

## CUTLASS GEMM Policies

CUTLASS定义了GEMM层次结构各个层次的tile尺寸，这些编译时常数尺寸被用以下声明组织的特例化模板类gemm::block_task_policy所组织。

```
template <
    int BlockItemsY,            /// 矩阵C一个tile行的高
    int BlockItemsX,            /// 矩阵C一个tile列的宽
    int ThreadItemsY,           /// 矩阵C一个thread tile行的高
    int ThreadItemsX,           /// 矩阵C一个thread tile列的宽
```

```
    int BlockItemsK,             /// 一个tile的深度
    bool UseDoubleScratchTiles, /// shared memory是否是双缓冲的flag
    grid_raster_strategy::kind_t RasterStrategy  /// Grid rasterization strategy
> struct block_task_policy;
```

各个可用GEMM分块结构policy在dispatch_policies.h被定义，我们在下文将展示一个policy。这个policy把矩阵乘法操作分解成CUDA块，每个块负责输出矩阵的一个 $128\times32$ $128 \times 32$ tile。储存 $\textbf{A}$ A\textbf{A}和 $\textbf{B}$ B\textbf{B}的thread block tile的尺寸分别为 $128\times8$ $128 \times 8$ 和 $8\times32$ $8 \times 32$ 。该policy针对矩阵 $\textbf{C}$ C\textbf{C}在其 $N$ N维上相对较小的GEMM计算进行了优化。

```
/// 一个为“高”SGEMM实例化的policy
template <>
struct gemm_policy<float, float, problem_size_t::Tall> :
    block_task_policy<
        128,        // BlockItemsY - 一个tile行的高
        32,         // BlockItemsX - 一个tile列的宽
        8,          // ThreadItemsY - 一个thread tile行的高
        4,          // ThreadItemsX - 一个thread tile列的宽
        8,          // BlockItemsK - 一个tile的深度
        true,       // UseDoubleScratchTiles - shared memory是否是双缓冲的flag
        grid_raster_strategy::Default>   // Grid rasterization strategy
{};
```

thread tile fragment的尺寸分别是 $\text{ThreadItemsY}\times1$ \text{ThreadItemsY}\times1 $\text{ThreadItemsY} \times 1$ 和 $\text{ThreadItemsX}\times1$ \text{ThreadItemsX}\times1 $\text{ThreadItemsX} \times 1$ 。在上例的情况下，矩阵 $\textbf{A}$ A\textbf{A}和矩阵 $\textbf{B}$ B\textbf{B}的fragment分别是 $8\times1$ $8 \times 1$ 和 $4\times1$ $4 \times 1$ 。

定义了policy类型之后，我们就能定义gemm::block_task的类型。这个模板有如下的参数列表。

```
template <
    /// block_task_policy参数化
    typename block_task_policy_t,

    /// 被乘值类型（矩阵A和B）
    typename value_t,
    /// accumulator类型（矩阵C和纯量）
    typename accum_t,

    /// 矩阵A的布局
    matrix_transform_t::kind_t  TransformA,

    /// 操作数A的alignment（用byte为单位）
    int LdgAlignA,

    /// 矩阵C的布局
```

```
    matrix_transform_t::kind_t  TransformB,

    /// 操作数B的alignment（用byte为单位）
    int LdgAlignB,

    /// 矩阵乘法的epilogue functor
    typename epilogue_op_t,

    /// 操作数C的alignment（用byte为单位）
    int LdgAlignC,

    /// 是GEMM支持的矩阵成分或是BlockItems{XY}的整数倍
    bool Ragged
> struct block_task;
```

value_t和accum_t分别指定了源操作数和accumulator矩阵的类型。TransformA和TransformB分别指定了操作数 $\textbf{A}$和 $\textbf{B}$的布局。即使我们没有讨论矩阵布局的细节，CUTLASS支持行优先和列优先输入矩阵的各类组合。

LdgAlignA和LdgAlignB指定alignment，如此可以启用CUTLASS设备代码来使用向量内存操作。以一个8字节的alignment为例，允许CUTLASS从一个双元素的向量中加载float数（Jie: 应该是说一下子加载两个float数）。这样减少了代码体积同时也通过减少GPU的飞行中内存操作的数量进一步提升性能（Jie: 众所周知，GPU schuduler issue内存访问指令之后首先会设置flag然后不会等待内存正式加载到寄存器而是继续issue下一条指令）。更重要的是，ragged指示矩阵维度是否是任意的尺寸（满足algnment的要求）。如果模板参数是false，那么矩阵 $\textbf{A}$、$\textbf{B}$和 $\textbf{C}$都应当有block_task_policy的tile参数的整数倍。

## Fusing Element-wise Operations with SGEMM

深度学习计算通常要求在GEMM计算之后进行一些简单的元素操作，比如后接一个激活函数。这些不限制带宽的层能够与GEMM操作末尾相结合，这样的好处是能够消除额外的kernel启动开销同时也能避免global内存的一次额外读写。

下面的例子展示了一个GEMM模板类的用例，此用例的功能是对带放缩的矩阵乘法操作后的结果加偏置后再执行ReLU（将结果截断成非负值）操作。以传参数的方法（参数可以是额外的矩阵或是张量地址或是额外的放缩functor）能直接地将末尾阶段拆成一个functor且不会拖累GEMM的实现。

首先我们定义一个类来实现gemm::epilogue_op语义。构造函数和其他的类方法在这里不做展示，但是元素偏置和ReLU操作在这里用类的的调用函数实现。

```
template <typename accum_t, typename scalar_t, typename output_t>
struct fused_bias_relu_epilogue {
```

```
    // 传给epilogue的额外成员数据
    scalar_t const *Bias;
    accum_t threshold;

    /// 在主机端和设备端都可以调用的的构造函数用于初始化数据成员
    inline __device__ __host__
    fused_bias_relu_epilogue(
        scalar_t const *Bias,
        accum_t threshold
    ): Bias(Bias), threshold(threshold) { }

    /// 执行 + ReLu 操作
    inline __device__ __host__
    output_t operator()(
        accum_t accumulator,  /// 矩阵乘法结果的数据
        output_t c,           /// accumulator矩阵C的原数据
        size_t idx            /// c的id，可能会被用来从其他矩阵加载数据
        ) const {

        // 通过放缩矩阵乘法结果，加偏置，加放缩的accumulator元素得到结果

        accum_t result = output_t(
            alpha * scalar_t(accumulator) +
            Bias[i] +                          // 读取并加偏置
            beta * scalar_t(c)
        );

        // 调用截断函数
        return max(threshold, result);
    }
};
```

其后我们将这个类作为epilogue的参数传入。

```
// 使用定制epilogue functor的新代码
typedef fused_bias_relu_epilogue_t<float, float, float>
    bias_relu_epilogue_t;

/// 计算GEMM外加偏置及ReLU操作
__global__ void gemm_bias_relu(
    ...,                                       /// 省略了bias_relu_op被构造的过程
    bias_relu_epilogue_t bias_relu_op) {

    // 定义block_task.
    typedef block_task<
        block_task_policy_t,          // 与此前policy相同
        float,
        float,
        matrix_transform_t::NonTranspose,
        4,
        matrix_transform_t::NonTranspose,
        4,
```

```
        bias_relu_epilogue_t,          // 使用了新的epilogue functor类型
        4,
        true
    > block_task_t ;

    // 声明静态分配的shared memory
    __shared__ block_task_t::scratch_storage_t smem;

    // 构建和调用task
    block_task_t(
        reinterpret_cast(&smem),
        &smem,
        A,
        B,
        C,
        bias_relu_op,                  // 使用了新的epilogue算子
        M,
        N,
        K).run();
}
```

这个简单的例子展示了将同源编程技术和高性能GEMM实现相结合的价值。

# Tesla V100 (Volta) Performance

CUTLASS在性能能与cuBLAS在GEMM计算相媲美的同时兼顾高开发效率。图9展示了CUTLASS与cuBLAS的性能对比（使用CUDA 9.0编译并执行在NVIDIA Tesla V100上，计算大规模矩阵——M=1024,N=K=4096M=1024, N=K=4096 ）。图9展示各种CUTLASS支持的数据类型以及行优先列优先数据布局的性能对比。

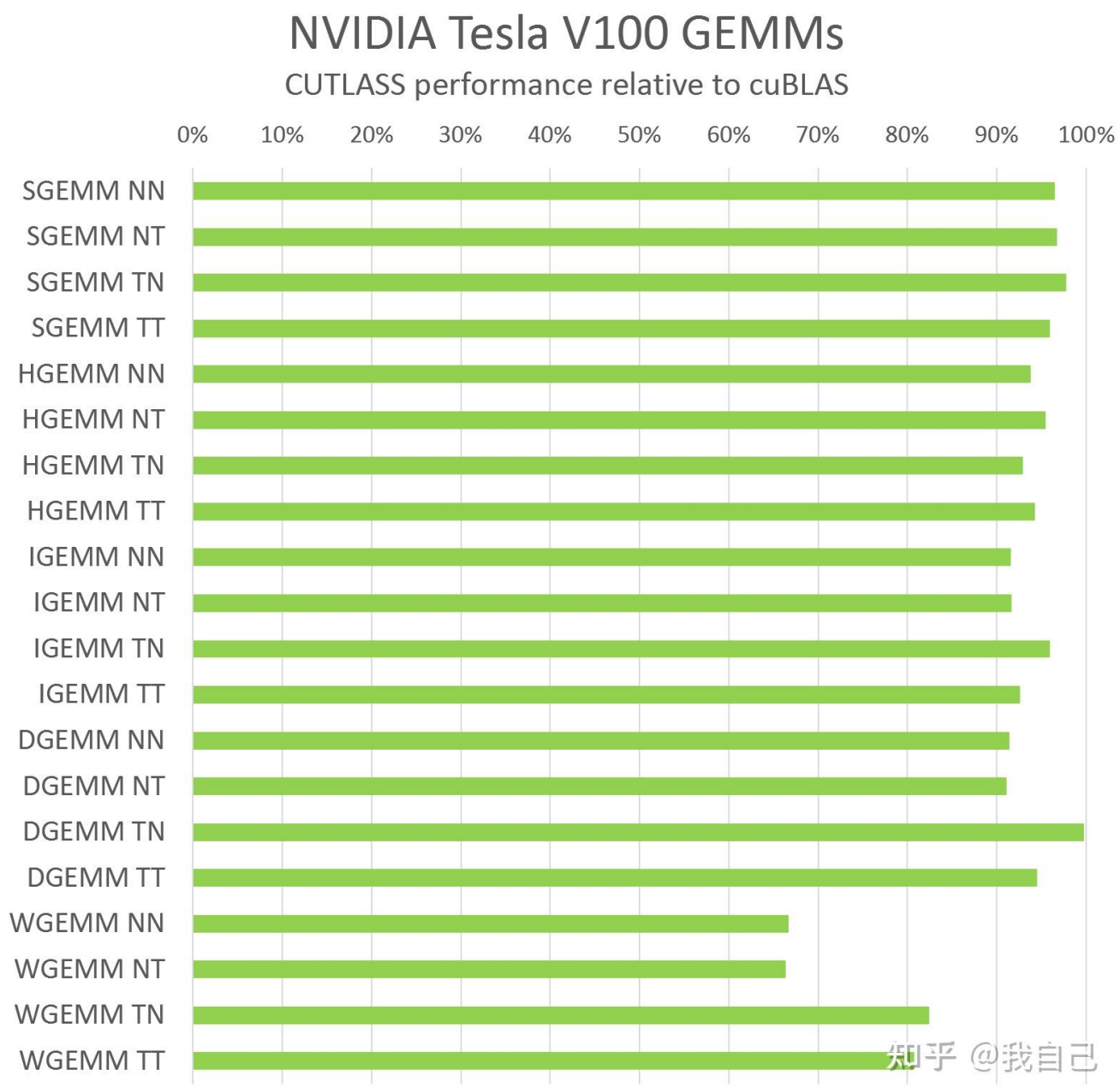


*图9，在CUDA 9编译的各种GEMM数据类型和矩阵数据布局的CUTLASS与cuBLAS对比。值得注意的是，之歌图遵循了BLAS的传统—矩阵通常都是列优先的。因此“N”代表列优先矩阵，“T”代表行优先矩阵。*

在大多数情况，CUTLASS C++能达到相比cuBLAS中手动汇编调优的kernel只有一点点性能损失。而对于WMMA GEMM（图9中的WGEMM），CUTLASS并没有取得与cuBLAS相匹敌的性能，但我们现在正与CUDA编译器和GPU架构队伍的同事密切合作，以期望能实现用CUDA就能直接写出高性能代码。

## Try CUTLASS Today!

还有很多本文没提及但又十分有趣的细节，我们推荐去CUTLASS仓库了解并亲自尝试CUTLASS。cutlass_test代码例展示了调用CUTLASS GEMM函数，确认结果以及性能测试。

## Acknowledgements

感谢Joel McCormack的技术见解和解释，特别是关于NVIDIA微架构以及cuBLAS和cuDNN采用的技术。

## References

[1] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning, arXiv:1410.0759, 2014.

[2] Michael Mathieu, Mikael Henaff, Yann LeCun. Fast training of Convolutional Networks through FFTs. arXiv:1312.5851. 2013.

[3] Andrew Lavin, Scott Gray. Fast Algorithms for Convolutional Neural Networks. arXiv:1509.09308. 2015.

[4] MAGMA. http://icl.cs.utk.edu/magma/index.html