On-the-Fly Lowering Engine: Offloading Data Layout Conversion for Convolutional Neural Networks

概括：和硬件相关，将基于GEMM的卷积中的Lowering过程卸载到硬件执行，以提升效率。使用OLE，Lowering的矩阵既不预先计算，也不存储在主存中。相反，硬件引擎从原始输入矩阵中动态生成Lowering之后的矩阵，以减少内存占用和带宽需求。此外，硬件卸载消除了Lowering操作的CPU周期，并与Lowering计算重叠，以隐藏性能开销。

问题：许多深度学习框架利用基于基本矩阵乘法（GEMM）的卷积来加速CNN的执行。基于GEMM的卷积提供了更快的卷积，但需要一个称为Lowering（即im2col）的数据转换过程，这将会导致显著的内存开销并降低性能。

如果将输入矩阵（输入特征映射）的大小增加（在下边的图2中有解释），从而将矩阵乘法的点积应用于特定的卷积窗口，则可以加速与GEMM的卷积操作。此数据布局转换过程称为Lowering（即im2col)。

虽然Lowering过程可以显著提高GEMM计算中的并行性，但它产生了性能和内存开销，减少了通过基于GEMM的卷积可以实现的潜在收益。这是因为Lowering过程涉及大量的内存流量，需要对所有卷积操作执行。此外，Lowering过程重复了相同的输入数据，显著增加了卷积操作的内存占用。基于软件的Lowering过程不必要地消耗CPU周期，浪费内存存储和带宽。

由于Lowering过程只是用最小的计算量重新定位和复制数据，因此为此任务使用处理器周期和分离存储空间可能是效率低下的。因此，我们在内存层次结构中添加了一个称为Lowering引擎（OLE）的简单硬件逻辑，以动态地生成Lowering的矩阵。

1、输入矩阵存储在内存中而不需要复制，因此不需要额外的内存空间来容纳Lowering的矩阵。

2、通过将Lowering过程卸载到硬件上，OLE可以消除CPU指令开销。

3、由于进行GEMM计算时，矩阵是被一行一行读入的，OLE可以通过动态的生成Lowering后的矩阵的请求部分，将降低过程与GEMM计算重叠。

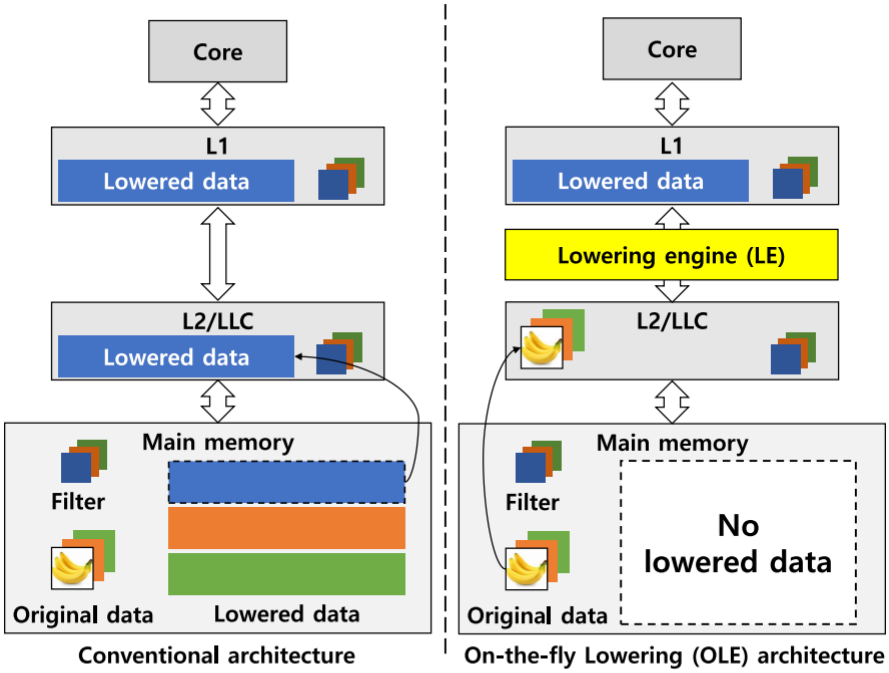

FIGURE 8. A comparison between memory hierarchies with and without OLE. OLE contains no lowered matrix in L2, LLC, and main memory.

当一个CPU核心在GEMM计算期间访问Lowering后的矩阵时，如果请求的数据不在缓存中，它将略过L1缓存。然后发送一个填充请求转到Lowering引擎，将丢失的数据提取到L1缓存中。然后，Lowering引擎生成读取请求，以从下一个较低级别的缓存（即L2缓存）中获取所请求的输入数据。Lowering引擎用获取的原始数据构造Lowering矩阵的**请求部分**，并将其安装在L1缓存中。

处理延迟不会降低性能。

GEMM计算的性能对L1缓存的延迟和L2缓存的容量很敏感。通过将LE放在L1和L2缓存之间，输入矩阵（即输入特征映射）以Lowering之后的格式存储在L1缓存中，同时以原始格式存储在L2缓存中。这样在L2中原始矩阵所占的内存容量不会太大，在L1中也不会占用太大的容量，Lowering过程可以和GEMM计算重叠，不会产生较大的性能损失。

Lowering过程发生在虚拟地址中，而LE与物理地址中的L1和L2缓存通信，需要Lowering到原始地址转换和虚拟物理地址转换。

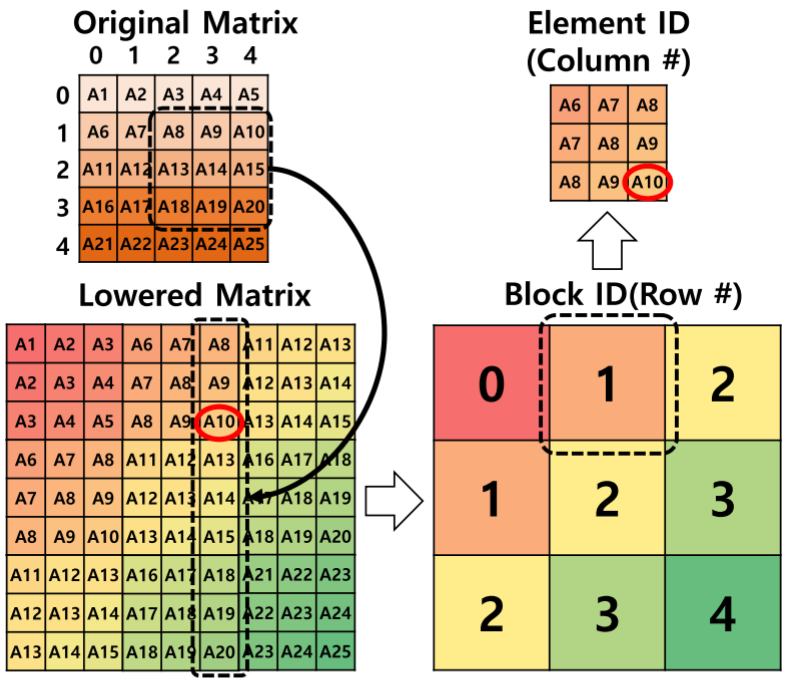

FIGURE 9. In address conversion, the proposed method uses Block ID and Element ID that indicates the row and column number of original matrix. It is used to calculate address of original data with equation 5.

首先，可以根据块内的数据复制模式将降低的矩阵划分为块，并且所有块都具有相同的复制模式。其次，在降低的矩阵中有许多相同的块。例如，在图9中，一个降低的矩阵被划分为9个块，它们具有相同的复制模式，其中反对角线上的元素来自相同的原始地址。降低矩阵的对角线上的块是相同的。为降低矩阵中的每个元素计算一个块ID和一个元素ID，然后使用给出的公式来计算原始矩阵中元素的地址。

在传统的高速缓存层次结构中，数据传输单元是一个高速缓存线（即，64字节的数据）。因此，当Lowering矩阵的L1缓存丢失发生时，LE构造一个包含所请求的输入数据的缓存行（我们将其称为Lowering行），然后将其加载到L1缓存中。构造一个Lowering行需要获取原始矩阵的多个高速缓存行（我们将其称为原始行）。这是因为L1和L2缓存在不同的数据布局中保存了输入矩阵。

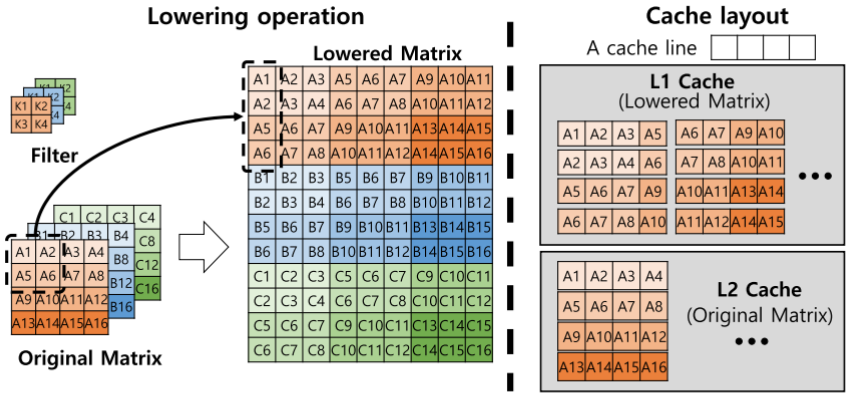

FIGURE 10. An example of lowering operation and its L1/L2 cache layouts with OLE. For simplicity, it is assumed that a cache line holds four matrix elements.

图10展示了使用OLE的L1和L2缓存的数据布局。我们假设一个缓存行包含四个数据元素。由于矩阵在存储器中按行主顺序存储，因此Lowering矩阵的四个连续元素（例如，A1、A2、A3和A5）存储在L1缓存上的同一缓存行中。但是，在L2缓存上，这四个元素分散在两条缓存线中，因为L2缓存使用了原始数据布局保存输入矩阵。

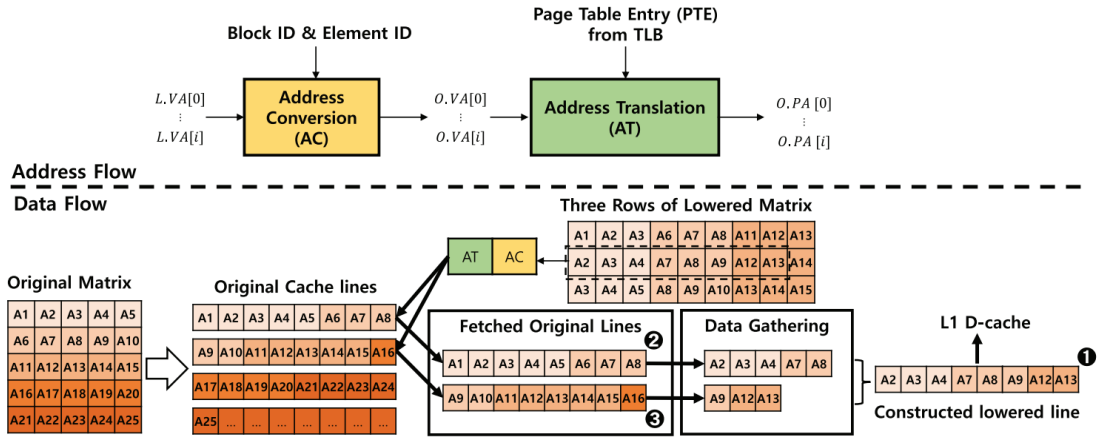

FIGURE 11. **Entire flow of on-the-fly lowering process.**

图11显示了Lowering过程的整个流程。为了构造Lowering行，地址转换单元首先对Lowering行的所有数据元素执行地址转换。它将Lowering行上的每个元素的虚拟地址（L. VA[i] 1）转换为分散在多个原始行上的相应元素的虚拟地址（O. VA[i] 1）。

在传统的缓存层次结构中，L2缓存通过物理地址访问。因此，在使用上述的地址转换方法，将L. VAs为请求的Lowering行的每个元素转换为O. VAs之后，LE将O. VAs转换为原始物理地址（O. PAs）。对于这个地址转换，LE利用了存储在TLB中的页面表条目（PTE）。使用翻译后的物理地址，LE向L2缓存发送读取请求，以获取包含所需数据元素的缓存行。

对于这个地址转换，LE利用了存储在TLB中的页面表条目（PTE）。使用翻译后的物理地址，LE向L2缓存发送读取请求，以获取包含所需数据元素的缓存行。

图11显示了构造一个Lowering行的示例。在本例中，从L2缓存中获取两个原始行（❷、❸），以构造一个包含Lowering矩阵的第二行中的前八行元素的Lowering行（❶）。然后，从这两条原始行中将所需的元素收集到一条Lowering行中。然后，构造的Lowering行被发送到L1缓存。

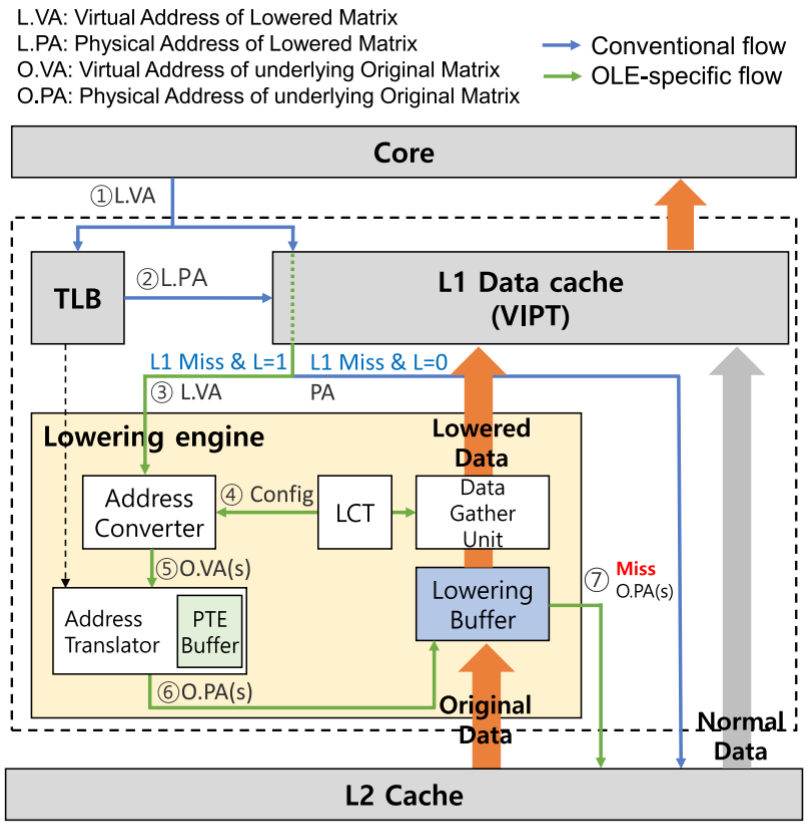

FIGURE 12. Microarchitecture of OLE.

图12展示了OLE的微观架构。LE位于L1和L2缓存之间。它由六个组件组成：Address converter、Lowering配置表（LCT）、Address translator、PTE缓冲区、Lowering缓冲区和数据收集单元。

当执行GEMM计算时，CPU核心发送一个具有虚拟地址的读取请求，以从内存中获取降低的矩阵（①）。

通过TLB将虚拟地址转换为物理地址，然后使用转换后的物理地址（②）访问L1缓存。在VIPT（Virtually Indexed, Physically Tagged）L1缓存中，TLB和L1缓存都被并行访问，以减少访问延迟。

如果请求的数据不在L1缓存中，并且它位于Lowering矩阵区域（即L=1），则读请求与Lowering数据的虚拟地址（③）一起发送到LE。如果发生L1错误，并且请求的数据不位于Lowering矩阵区域（即L=0），则读取请求将直接发送到L2缓存。

在此过程中，Address converter是指对存储在LCT（④）中的原始数据的起始地址、输入矩阵和核的维数、步幅等参数进行地址转换。LCT是通过内存映射的寄存器实现的，这样LCT的内容就可以通过一个常规的内存访问接口进行更新。转换后的虚拟地址然后由address translator转换为物理地址，然后用于访问L2高速缓存（⑤，⑥）。

为了评估性能效益，我们首先测量单个CONV层的执行时间。结果表明，使用OLE显著减少了执行时间。然后从消除降低指令和加速GEMM操作两个角度分析其改进。最后，我们对CNN模型的端到端执行情况进行了比较，结果表明，CONV层的节省可以将端到端执行时间减少33.1–47.1%。

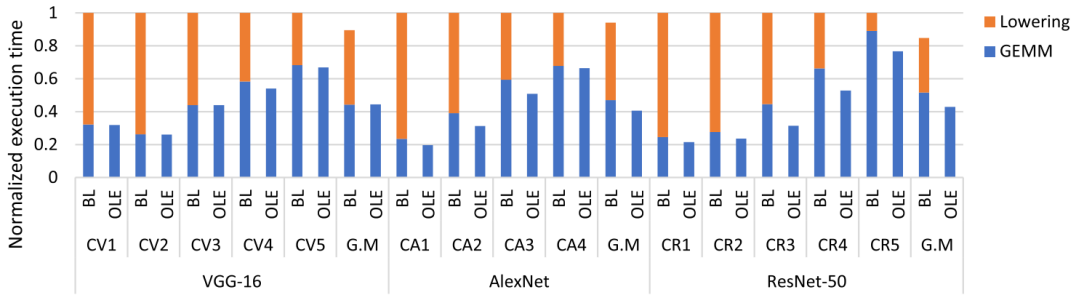

FIGURE 15. A comparison of execution times of individual convolutional layers. The execution times are normalized to the baseline (BL).

总的来说，对于VGG-16、AlexNet和ResNet-50，OLE将卷积层的执行时间平均减少了45.2%、53.1%和41.9%。

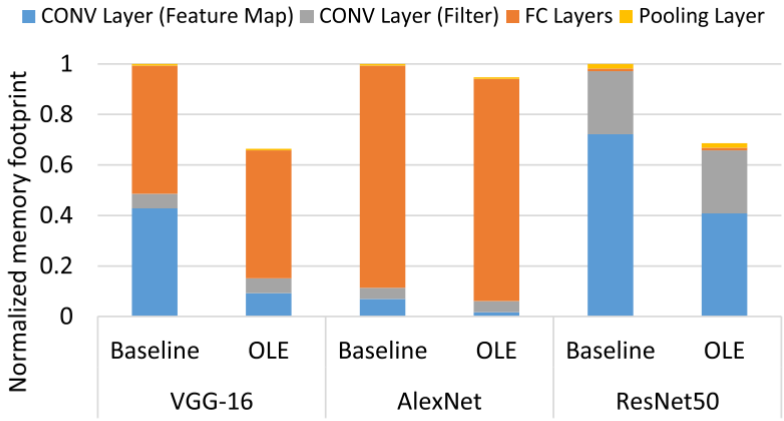

FIGURE 14. A comparison of dynamic memory allocations when executing CNN models on Darknet framework. OLE eliminates memory allocations for lowered matrix, which translates to large savings in the total memory footprints.

图14显示OLE可以通过33.5/5.3/31.3%分别减少VGG-16/AlexNet/ResNet-50的总体内存占用。

基于GEMM（General Matrix Multiply）的卷积是一种常见的卷积计算优化方法，它利用矩阵乘法的高效实现来加速卷积运算。下面是该方法的一般步骤：

1. 输入和卷积核转换为矩阵形式：将输入数据和卷积核按照一定的规则转换为矩阵形式。通常，输入被转换为一个或多个二维矩阵，其中每个矩阵表示一组输入通道的图像块。卷积核被转换为一个二维矩阵，其中每行表示一个卷积核的权重。
2. 执行矩阵乘法：使用高效的矩阵乘法算法（如BLAS库中的GEMM函数）执行输入矩阵和卷积核矩阵的乘法运算。这将生成输出矩阵，其中每个元素表示对应位置的输出特征图的像素值。
3. 输出矩阵转换为特征图：将输出矩阵转换回特征图的形式。这通常涉及到对输出矩阵进行reshape操作，以将其重新组织成输出特征图的形状。

基于GEMM的卷积方法的优势在于可以利用矩阵乘法的高度优化实现，例如使用SIMD指令集和并行计算。这种方法在一些硬件平台上（如GPU）上特别高效，因为它们在矩阵乘法方面具有强大的计算能力。

需要注意的是，基于GEMM的卷积方法可能对内存使用和数据转换要求较高，因此在实际应用中需要权衡计算效率和内存开销。此外，针对不同的卷积大小和硬件平台，可能需要采用不同的矩阵转换策略和优化技巧。

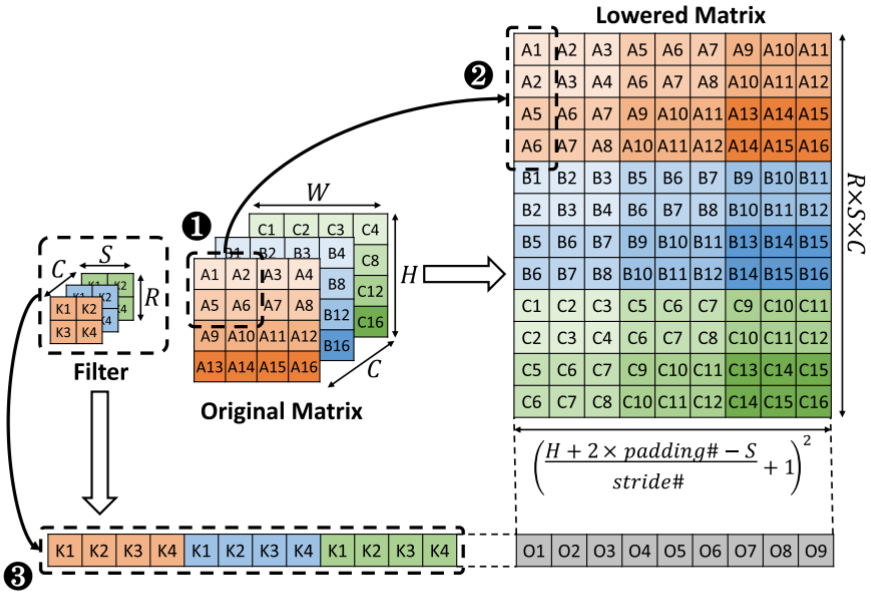

FIGURE 2. GEMM-based convolution.

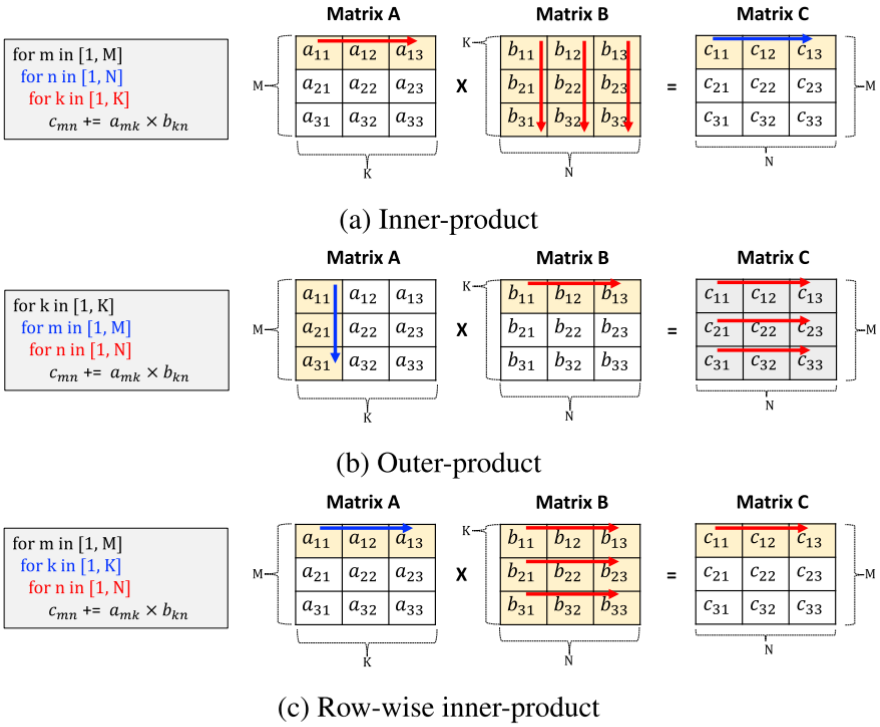

FIGURE 3. Comparision of matrix-matrix multiplication algorithms.

图3展示了3种矩阵积算法，其内存访问由较大的区别

a会造成大量的输入矩阵的数据重复读取，但是输出矩阵的每个元素只需要读一次

b会造成输出矩阵的每个元素重复读和累加（即写内存）

c是一种比较均衡的算法，行内积算法（也就是古斯塔夫森算法）避免了这种高内存流量。如图3-(c)所示，该算法通过将矩阵B行的元素乘以矩阵a一行的每个元素，每次计算一行输出矩阵C。即使该算法不能对输入和输出矩阵实现极高的重用，它也提供了相对较好的行输入和输出矩阵的重用。

在基于GEMM的卷积中，使用降低的矩阵作为矩阵轴矩阵乘法中的第二个矩阵（图3中的矩阵B）。降低矩阵中的一行连续元素在原始矩阵中也大多是连续的。因此，如果基于GEMM的卷积使用外积或行向内积算法，则GEMM检索直接来自原始矩阵的数据，而不需要降低过程，它在访问原始矩阵时也将具有很高的局部性。