函数库 GEMM 的单核设计与实现

小组成员：刘学建(S211000844), 张梦豪(S211000834)

随着人工智能技术往着深层神经网络的迅速发展，通常意义下一个神经网络的前向推导与预测都需要巨大的计算能力支持。在计算资源短缺的时代，充分利用好计算资源显得尤为重要。BLAS 库作为基本线性计算函数库，在充分利用计算资源进行高性能计算发挥着重要的作用。GEMM 作为 BLAS 库中使用最频繁且最重要的函数，对 GEMM 依据相应的计算平台体系结构进行底层优化设计对于充分挖掘相对应计算平台的计算性能有着重要的意义[4,18]。

在本文中，首先从四个方向(A 的行与列、B 的行与列)组合排列对 GEMM 进行了分块分析，得出了 6 种分块方案[4,18]，并从计算量、存储空间和数据搬移角度对其进行分析，淘汰了以 GEPDOT 为核心循环的 2 种分块方案。然后转入对核心循环的分析。首先对核心循环进行 Cache 和 Register 级的分块分析，找出对分块的限制之处，在不超出限制的情况下依据相应的硬件平台完成分块参数的选取配置。接下来在上述分块参数配置的情况下，从代码级优化、分块实现方案和数据封装与搬移角度三个方向对 GEMM 进行了优化实现。最后将 GEMM 从最原始的三层循环实现逐步完善优化至带有代码级优化和数据封装与搬移的基于 GEBP 的 GEPP 实现，并且对每一步完善优化进行性能测试、比较与分析，实现了一个性能超出最原始三层循环实现约 8 倍的 GEMM 单核实现方案。

1.1 GEMM 分块分析

在基于多级存储结构的计算平台中，协调各级存储之间关系并充分利用 Cache 搬移数据至处理器的速度远高于内存的优势是实现高性能计算的关键。利用矩阵分块实现矩阵乘法是提高 Cache 命中率充分利用 Cache 优势的一个体现。

假设矩阵 A 的规模为$M \times K$，矩阵 B 的规模为$K \times N$，矩阵 C 的规模为$M \times N$，基于上述矩阵规模的矩阵乘法分块实现机理如下：

$$A \rightarrow \begin{pmatrix} A_{00} & A_{01} & \cdots & A_{0,w-1} \\ A_{10} & A_{11} & \cdots & A_{1,w-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{u-1,0} & A_{u-1,1} & \cdots & A_{u-1,w-1} \end{pmatrix}, \quad B \rightarrow \begin{pmatrix} B_{00} & B_{01} & \cdots & B_{0,v-1} \\ B_{10} & B_{11} & \cdots & B_{1,v-1} \\ \vdots & \vdots & \ddots & \vdots \\ B_{w-1,0} & B_{w-1,1} & \cdots & B_{w-1,v-1} \end{pmatrix}$$

$$C \rightarrow \begin{pmatrix} C_{00} & C_{01} & \cdots & C_{0,v-1} \\ C_{10} & C_{11} & \cdots & C_{1,v-1} \\ \vdots & \vdots & \ddots & \vdots \\ C_{u-1,0} & C_{u-1,1} & \cdots & C_{u-1,v-1} \end{pmatrix}$$

图 0-1　矩阵乘法分块实现机理图

其中：

$$C_{i,j} = \sum_{p=0}^{w-1} A_{i,p} B_{p,j} \qquad \text{(公式 3-1)}$$

接下来将以上述矩阵乘法分块实现原理引入，在 KAZUSHIGE GOTO 等人提出的 GOTO 分块算法[4]的基础上，提出了 6 钟矩阵分块方案，并对其从计算量、存储空间和数据搬运三个角度进行了深入研究和分析。

1.1.1　GEMM 的分块实现方案

1. 分块术语说明

在矩阵乘法分块实现的各种方案中，根据矩阵规模 M、N 和 K 的大小，存在着业界通用的术语[4,11]，各种术语图解及释义如下：

表 0-1　GEMM 分块术语表

字母	形状	释义
GEXY		X 和 Y 分别象征矩阵 A 和矩阵 B 的形状
M	Matrix	行和列 都较大
P	Panel	行或列 有一者小的
B	Block	行和列 都较小
DOT		点积

m	n	k	图解	标签
large	large	large		GEMM
large	large	small		GEPP
large	small	large		GEMP
small	large	large		GEPM
small	large	small		GEBP
large	small	small		GEPB
small	small	large		GEPDOT
small	small	small		GEBB

图 0-2　GEMM 分块术语图

其中 large 和 small 并非描述矩阵规模的绝对量级，而是描述矩阵规模之间的相对量级。

2. 分块实现方案

根据上述术语阐述及矩阵乘法运算原理，矩阵乘法分块实现可从 GEPP、GEMP 和 GEPM[4]三个角度进行分析，并可再进一步分解为 GEBP、GEPB 和 GEPDOT 三个底层实现，如图：

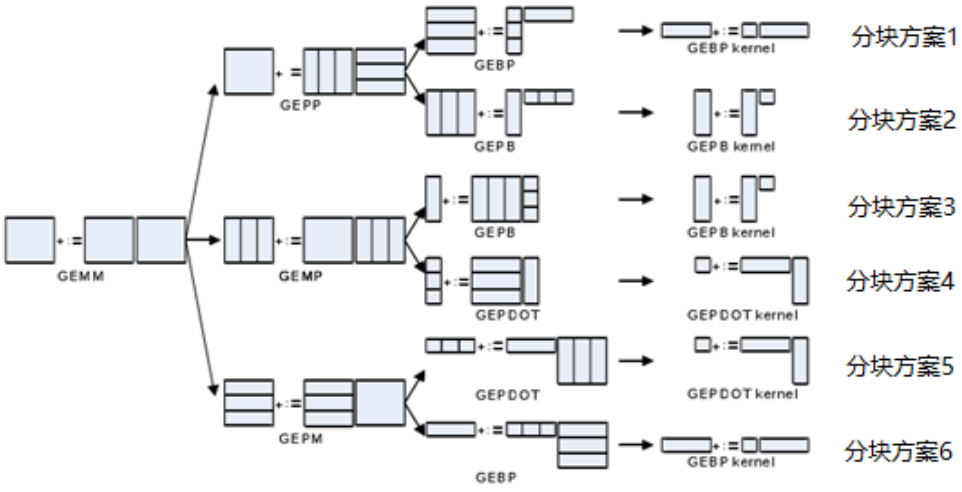

图 0-3　矩阵乘法分块实现方案图

其中基于 GEBP 的 GEPP 实现的图解如下：

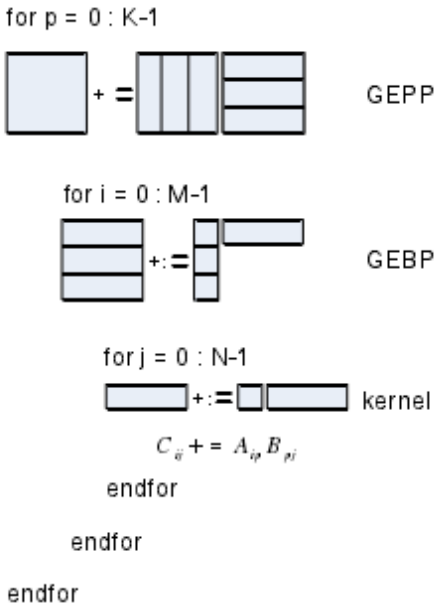

图 0-4　基于 GEBP 的 GEPP 的 GEMM 实现图

GEBP、GEPB 和 GEPDOT 三者作为矩阵乘法分块实现的底层循环实现，任何矩阵乘法分块实现在原则上均可映射为多个 GEBP、GEPB 和 GEPDOT 三者之一的底层实现。作为底层实现的 GEBP、GEPB 和 GEPDOT，与计算平台体系结构接触最密切，对于实现矩阵乘法分块实现有着决定性作用，因此称 GEBP、GEPB 和 GEPDOT 为核心循环。充分了解相应计算平台的体系结构，并将核心循环与相应计算平台体系结构及软硬件资源高效结合是实现高性能矩阵乘法分块实现的关键。

1.1.2　分块实现方案的分析与评价

1. 核心循环为 GEPDOT 的分块实现方案分析与评价

核心循环为 GEPDOT 的分块实现方案的是分块实现方案 4 和 5。在 3.2 节核心循环的分析与设计中，可知在计算平台的多级存储结构中，核心循环 GEPDOT 对于 L2Cache 搬移数据至处理器的带宽要求是其他核心循环的 2 倍，极大地降低了分块数据封装和搬移的效率，以至于限制了矩阵乘法分块实现的高性能，因此在本文中不考虑以 GEPDOT 为核心循环实现矩阵乘法，即在本文中不考虑分块实现方案 4 和 5。

2. 核心循环为 GEBP 的分块实现方案分析与评价

核心循环为 GEBP 的分块实现方案的是分块实现方案 1 和 6。分块实现方案 1 和 6 的不同之处在于对矩阵 A 和 B 的数据需求移动方向不同。如下图，分块实现

方案 1 在多个 GEBP 核心循环之间重复使用同一 B Panel，减少了数据在计算平台的多级存储结构之间的搬移次数，提高了矩阵乘法分块实现的效率。

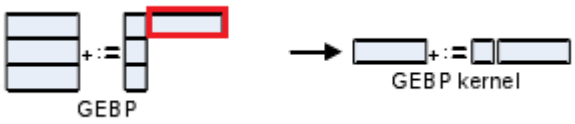

图 0-5

如下图，分块实现方案 6 在多个 GEBP 核心循环之间依次使用不同的 B Panel，增加了数据在计算平台的多级存储结构之间的换进换出次数，减少了 Cache 命中率，限制了矩阵乘法分块实现的效率。

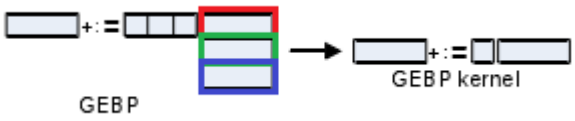

图 0-6

核心循环 GEBP 的实现如下图，图中将 B Panel 以高的长度远大于宽的长度切成若干微片进行分块计算，在本章第五节 B Panel 的数据封装与搬移中，更有利于列主序存储的矩阵减少寻址过程中产生的计算量，加快数据封装与搬移的效率。

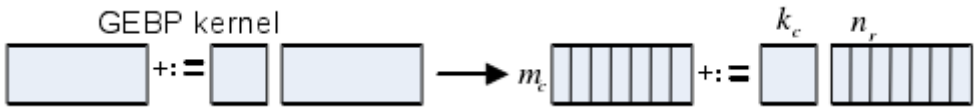

图 0-7　GEBP 中 B Panel 切片图

综上所述，基于 GEBP 的两种分块实现方案中，分块方案 1 比分块方案 6 在数据搬移上效率更高。列主序存储的矩阵乘法选择分块实现方案 1 有更高实现效率。

3. 核心循环为 GEPB 的分块实现方案分析与评价

核心循环为 GEPB 的分块实现方案的是分块实现方案 2 和 3。分块实现方案 2 和 3 的不同之处在于对矩阵 A 和 B 的数据需求移动方向不同。如下图，分块实现方案 2 在多个 GEPB 核心循环之间重复使用同一 A Panel，减少了数据在计算平台的多级存储结构之间的搬移次数，提高了矩阵乘法分块实现的效率。

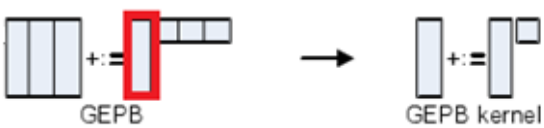

图 0-8

如下图，分块实现方案 3 在多个 GEBP 核心循环之间依次使用不同的 A Panel，

增加了数据在计算平台的多级存储结构之间的换进换出次数，减少了 Cache 命中率，限制了矩阵乘法分块实现的效率。

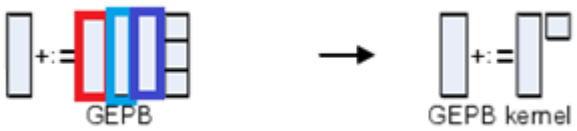

图 0-9

核心循环 GEPB 的实现如下图，图中将 A Panel 以宽的长度远大于高的长度切成若干微片进行分块计算，在本章第五节 A Panel 的数据封装与搬移中，更有利于行主序存储的矩阵减少寻址过程中产生的计算量，加快数据封装与搬移的效率。

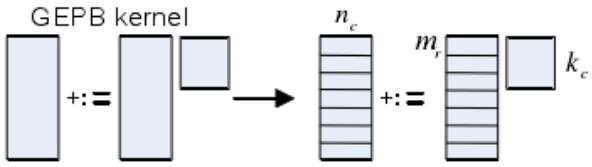

图 0-10　GEPB 中 A Panel 切片图

综上所述，基于 GEPB 的两种分块实现方案中，分块方案 2 比分块方案 3 在数据搬移上效率更高。行主序存储的矩阵乘法选择分块实现方案 2 有更高实现效率。

4. 结论

(1) 列主序存储的矩阵乘法选择分块实现方案 1 有更高实现效率；

(2) 行主序存储的矩阵乘法选择分块实现方案 2 有更高实现效率。

1.2　核心循环的分析与设计

如今的计算平台大都具有多核多线程和多级存储结构的特点，资源众多且相互之间联系复杂。核心循环是矩阵乘法分块实现最接近计算平台体系结构和硬件资源的部分。在计算平台复杂的体系结构和硬件资源支持下，如何高效利用相应的硬件资源成为了核心循环以及矩阵乘法分块高性能喜下实现的关键。

图 3-11 是计算平台的基于 Cache 的多级存储结构图，具有内存、Cache 和 Register 三个粗略的存储层次。本节从计算平台的多级存储结构角度入手，对三个核心循环 GEBP、GEPB 和 GEPDOT 从 Cache 和 Register 两个存储层次进行分析，最后根据计算平台具体的硬件资源数据完成核心循环分块参数的选取。在第二章中，已对计算平台的体系结构和相关的硬件资源作了详细介绍。

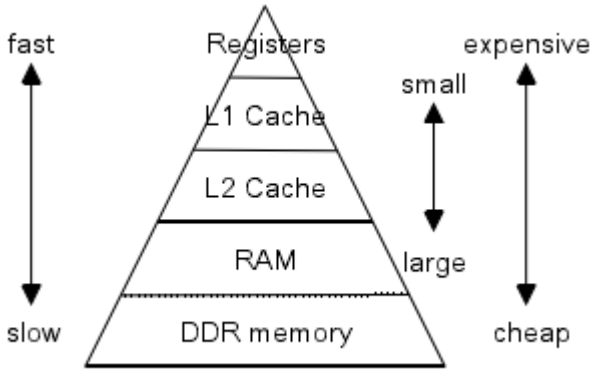

图 0-11　多级存储结构图

1.2.1　GEBP 的分析与设计

1. 基于 Cache 级的分块分析

本小节中，仅在 Cache 级分析核心循环 GEBP 的分块实现。假设 Block A 的规模为$M_c \times K_c$，Panel B 的规模为$K_c \times$ N，则 Panel C 的规模为$M_c \times$ N，如下图所示。

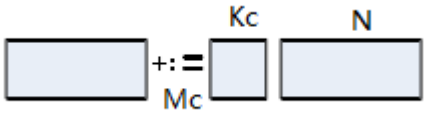

图 0-12

接下来将 Panel B 和 Panel C 以每N_r列拆分，每N_r列分别标记为B_j和C_j：

$$B_p = (B_0 \ B_1 \ B_2 \ \cdots \ B_j \ \cdots)$$

$$C_p = (C_0 \ C_1 \ C_2 \ \cdots \ C_j \ \cdots)$$

并将 Block A 标记为A_b，即$C_j = A_b B_j$，如下图所示。

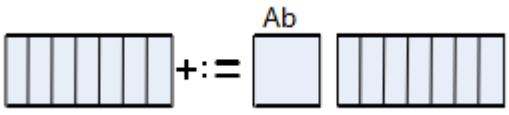

图 0-13

为了支撑下文分析，此处对基于 Cache 的多级存储模型中 L1Cache 与 Register 之间带宽和A_b、B_j和C_j的总体数据量作出相应假设：

假设 1：A_b、B_j和C_j的总体数据量足够全部存储至 L1Cache 中；

假设 2：若A_b、B_j和C_j均存储至 L1Cache，则在处理器中计算矩阵乘法$C_j = A_b B_j$时

能够接近处理器浮点计算峰值，即 L1Cache 与 Register 之间存在足够带宽以至于搬移A_b、B_j和C_j不能成为计算矩阵乘法$C_j = A_b B_j$的障碍。

下图为核心循环 GEBP 分块实现示意图。从图中可知，核心循环 GEBP 分块实现的访存量来源于四部分：将A_b从内存搬移至 Cache 的访存量$M_c K_c$、将所有B_j从内存搬移至 Cache 的访存量$K_c N$、将所有需要更新的C_j从内存搬移至 Cache 的访存量$M_c N$和将所有完成更新的C_j从 Cache 搬移至内存的访存量$M_c N$。核心循环 GEBP 分块实现的访存量总共为$M_c K_c + K_c N + 2M_c N$。核心循环 GEBP 分块实现的运算量来源于三部分：所有$A_b B_j$计算中的乘法计算量$M_c K_c N$、所有$A_b B_j$计算中的加法计算量$M_c(K_c - 1)N$和更新所有C_j的加法计算量$M_c N$，总运算量为$2M_c K_c N$。

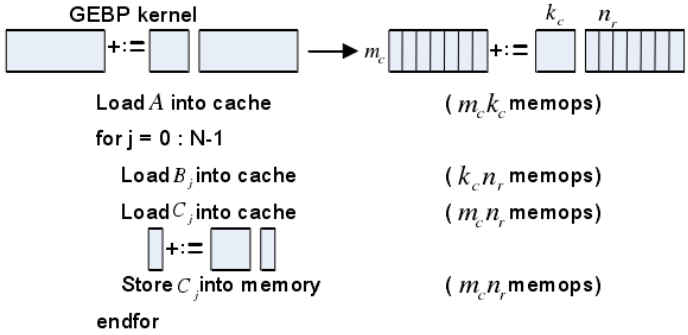

图 0-14　GEBP 分块实现示意图

那么核心循环 GEBP 分块实现的访存比为：

$$访存比 = \frac{2M_c K_c N}{M_c K_c + K_c N + 2M_c N} = \frac{2M_c K_c N}{M_c K_c + (K_c + 2M_c)N} \quad (公式\ 3\text{-}2)$$

当 N >> K_c且 N >> M_c时

$$访存比 \approx \frac{2M_c K_c}{K_c + 2M_c} = \frac{2}{1/M_c + 2/K_c} \quad (公式\ 3\text{-}3)$$

访存比决定着计算量与相应计算量所需访存量之间的比值。为了降低访存对计算的限制，突出计算的主导地位，则必须使访存比较大化。要使访存比较大化，则要在计算平台多级存储结构中 Cache 大小的限制下使得$M_c K_c$尽量的大。

另一方面，对$C_j = A_b B_j$运算量进行分析。其运算量来源于三部分：所有$A_b B_j$计算中的乘法计算量$M_c K_c N$、所有$A_b B_j$计算中的加法计算量$M_c(K_c - 1)N$和更新所有C_j的加法计算量$M_c N$。而更新C_j的运算量只占总运算量的$\frac{1}{2K_c}$。为了提高关键计算$A_b B_j$的计算量比重，降低更新C_j的运算量比重，K_c的选择也是越大越好。

由公式 3-3 以及大量 BLAS 中矩阵乘法分块实现的文献研究表明，M_c与K_c大致相等时效果最好，即A_b大致为方阵时矩阵乘法分块实现的效果最好。但在实际情况中，应结合其他各方面因素考虑M_c及K_c的参数选择。

由上述分析可知，M_c与K_c的参数选择越大矩阵乘法实现效果越好。但现代计算平台多级存储结构中 Cache 容量比较小，要将A_b、B_j和C_j全部存至 Cache 中不太现实，所以考虑将A_b移至 L2Cache 中存储，B_j和C_j依然存储至 L1Cache 中。

A_b移至 L2Cache 中存储之后，将A_b中的数据从 L2Cache 搬移至 Register 中需要一定的代价，即 L2Cache 与 Register 之间的带宽限制。在分块矩阵乘法$C_j = A_b B_j$中，计算量为$2M_c K_c N_r$，将A_b从 L2Cache 中搬移 Register 的访存量为$M_c K_c$。在假设二：L1Cache 与 Register 之间存在足够带宽以至于搬移C_j和B_j不能成为计算矩阵乘法$C_j = A_b B_j$障碍的前提下，假设计算平台的处理器从 Register 取得数据进行计算的速率为V_{comp}，L2Cache 与 Register 之间的带宽为V_{L2Load}，为保证将A_b从 L2Cache 中搬移 Register 满足计算的需求，应满足下列不等式：

$$\frac{2M_c K_c N_r}{V_{comp}} \geq \frac{M_c K_c}{V_{L2Load}} \quad \text{(公式 3-4)}$$

即

$$N_r \geq \frac{V_{comp}}{2V_{L2Load}} \quad \text{(公式 3-5)}$$

综上所述，核心循环中A_b矩阵分块参数M_c、K_c的选取应越大越好，但受到多级存储结构中 Cache 大小的限制；在假设二：L1Cache 与 Register 之间存在足够带宽以至于搬移A_b和B_j不能成为计算矩阵乘法$C_j = A_b B_j$障碍的前提下，核心循环B_j矩阵分块参数N_r的选取受限于多级存储结构中 L2Cache 与 Register 之间的带宽大小。

2. 基于 Register 级的分块分析

在现代计算平台的多级存储结构中，Register 的容量和数量是极其有限的。在计算分块矩阵乘法$C_j = A_b B_j$中不可能将所需的分块矩阵数据A_b、B_j和C_j全部存储至 Register 中，考虑将A_b进一步以每M_r行拆分，每M_r行分别标记为A_{b_j}，

$$A_b = \begin{pmatrix} A_{b_0} \\ A_{b_1} \\ \vdots \\ A_{b_j} \\ \vdots \end{pmatrix}$$

因此矩阵乘法$A_b B_j$将进一步分块运算，以此更新C_j的$M_r \times N_r$子矩阵，如下图所示。

图 0-15

在上述进一步的分块运算中，有关C_j的$M_r \times N_r$子矩阵的运算量为$2M_r N_r K_c$，访存量来源于四部分：将A_{b_j}从内存搬移至 Cache 的访存量$M_r K_c$、将B_j从内存搬移至 Cache 的访存量$K_c N_r$、从内存中将C_j的$M_r \times N_r$子矩阵读取至 Cache 的访存量$M_r N_r$和将C_j的$M_r \times N_r$子矩阵从 Cache 更新至内存的访存量$M_r N_r$，总访存量为$M_r K_c + K_c N_r + 2M_r N_r$，其中有关$C_j$的$M_r \times N_r$子矩阵的访存量为$2M_r N_r$。访存比为：

$$\text{访存比} = \frac{2M_r K_c N_r}{M_r K_c + K_c N_r + 2M_r N_r} = \frac{2M_r N_r}{(M_r + N_r) + {2M_r N_r}/{K_c}} \approx \frac{2M_r N_r}{M_r + N_r} \qquad \text{(公式 3-6)}$$

当$K_c \gg M_r$时，有关C_j的$M_r \times N_r$子矩阵的总访存量$2M_r N_r$相对于运算量$2M_r N_r K_c$是非常小的，并且$2M_r N_r K_c$的运算量中有关C_j的$M_r \times N_r$子矩阵的也只有$M_r N_r$个加法。并且对C_j的$M_r \times N_r$子矩阵的访问术语流水式访问模式，即读入->计算->更新。考虑到 L1Cache 容量的紧缺性，C_j的$M_r \times N_r$子矩阵一直停留在 L1Cache 不是一个高效的做法，应将其以及C_j存储至下级 Cache 中。

综上所述，考虑到 Register 的容量及数量极其有限的原因，应将A_b分块为更小的矩阵A_{b_j}，同时由公式 3-5 可知，Register 确实是制约矩阵乘法分块高性能实现的一个重要原因。考虑到有关C_j的计算量与访存量在整个矩阵乘法分块计算中的低比重以及对C_j的流水式访问模式，应将C_j存储至 L1Cache 的下级 Cache 中。最后，考虑到将A_b分块后的各块的数据访问方向与矩阵在内存中的存储方向不一致的情况，将涉及到矩阵的数据封装与搬移的问题。矩阵的数据封装与搬移问题将在 3.3 节中

进行详细阐述。

3. 基于硬件资源的分块参数选择

在上述的分析中，涉及到 4 个参数：M_c、M_r、K_c和N_r。接下来，将结合相应计算平台的体系结构及硬件资源，对涉及的 4 个参数进行最优化选取配置。

C_j的子矩阵参数M_r和N_r的选取：

(1)应满足公式 3-5；

(2)考虑到多级存储结构中 Register 有限的个数，其中$1/2$用于存储C_j的$M_r \times N_r$子矩阵，剩余部分用于存储矩阵分块运算$A_b B_j$所需的数据；

(3)考虑到公式 3-6，在M_r和N_r尽可能大且$M_r \approx N_r$时访存比的值较大，性能较佳。

综上所述，结合相应计算平台的体系结构和硬件资源，最终确定$M_r = 4$，$N_r = 4$。

参数K_c的选取：

(1)考虑到公式 3-6，为了降低更新C_j的$M_r \times N_r$子矩阵所需的开销，K_c的选取应越大越好；

(2)大小为$K_c \times N_r$的矩阵B_j存储至 L1Cache 中，为了减少 L1Cache 中发生的冲突，增加 Cache 命中率，大小为$K_c \times N_r$的矩阵B_j在 L1Cache 中所占的空间应尽量低于 L1Cache 的$1/2$[19,20]；

(3)大小为$M_c \times K_C$的矩阵A_b存储至 L2Cache 中，为了减少 L2Cache 中发生的冲突，增加 Cache 命中率，大小为$M_c \times K_C$的矩阵A_b在 L2Cache 中所占的空间应尽量低于 L2Cache 的$1/2$[19,20]。

参数M_c的选取：

(1)考虑到公式 3-3，在M_c和K_c尽可能大且$M_c \approx K_c$时访存比的值较大，性能较佳；

(2)参数M_c的选取还受限于大小为$K_c \times N_r$的矩阵B_j和大小为$M_c \times K_C$的矩阵A_b的参数选取。

综上所述，结合相应计算平台的体系结构和硬件资源，最终确定$M_c = 256$，$K_c = 128$。

1.2.2 GEPB 的分析与设计

1. 基于 Cache 级的分块分析

本小节中，仅在 Cache 级分析核心循环 GEPB 的分块实现。假设 Block B 的规模为$K_c \times N_c$，Panel B 的规模为M × K_c，则 Panel C 的规模为M × K_c，如下图所示。

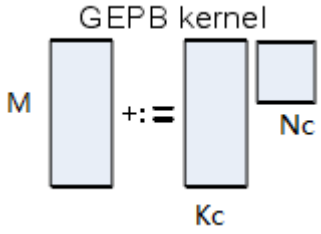

图 0-16

接下来将 Panel A 和 Panel C 以每M_r行拆分，每M_r行分别标记为A_j和C_j：

$$A_p = \begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_j \\ \vdots \end{pmatrix}, \quad C_p = \begin{pmatrix} C_0 \\ C_1 \\ \vdots \\ C_j \\ \vdots \end{pmatrix}$$

并将 Block B 标记为B_b，即$C_j = A_j B_b$，如下图所示。

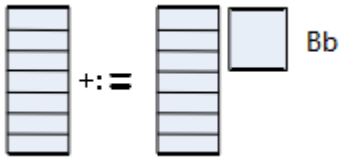

图 0-17

为了支撑下文分析，此处对基于 Cache 的多级存储模型中 L1Cache 与 Register 之间带宽和B_b、A_j和C_j的总体数据量作出相应假设：

假设 1：B_b、A_j和C_j的总体数据量足够全部存储至 L1Cache 中；

假设 2：若B_b、A_j和C_j均存储至 L1Cache，则在处理器中计算矩阵乘法$C_j = A_j B_b$时能够接近处理器浮点计算峰值，即 L1Cache 与 Register 之间存在足够带宽以至于搬移B_b、A_j和C_j不能成为计算矩阵乘法$C_j = A_j B_b$的障碍。

下图为核心循环 GEPB 分块实现示意图。从图中可知，核心循环 GEPB 分块实现的访存量来源于四部分：将B_b从内存搬移至 Cache 的访存量$K_c N_c$、将所有A_j从

内存搬移至 Cache 的访存量MK_c、将所有需要更新的C_j从内存搬移至 Cache 的访存量MN_c和将所有完成更新的C_j从 Cache 搬移至内存的访存量MN_c。核心循环 GEBP 分块实现的访存量总共为$K_cN_c + MK_c + 2MN_c$。核心循环 GEPB 分块实现的运算量来源于三部分：所有A_jB_b计算中的乘法计算量MK_cN_c、所有A_bB_j计算中的加法计算量$M(K_c - 1)N_c$和更新所有C_j的加法计算量MN_c，总运算量为$2MK_cN_C$。

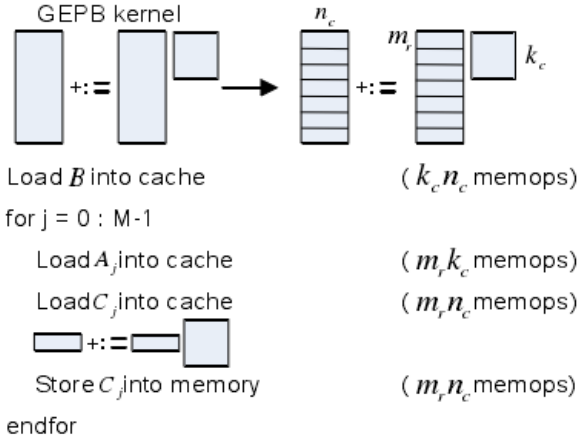

图 0-18　GEPB **分块实现示意图**

那么核心循环 GEPB 分块实现的访存比为：

$$访存比 = \frac{2MK_cN_C}{K_cN_c+MK_c+2MN_c} = \frac{2MK_cN_C}{K_cN_c+(K_c+2N_c)M} \quad \text{(公式 3-7)}$$

当 M >> K_c且 N >> N_c时

$$访存比 \approx \frac{2N_cK_c}{K_c+2N_c} = \frac{2}{{}^{1}/_{N_c}+{}^{2}/_{K_c}} \quad \text{(公式 3-8)}$$

访存比决定着计算量与相应计算量所需访存量之间的比值。为了降低访存对计算的限制，突出计算的主导地位，则必须使访存比较大化。要使访存比较大化，则要在计算平台多级存储结构中 Cache 大小的限制下使得N_cK_c尽量的大。

另一方面，对$C_j = A_jB_b$运算量进行分析。其运算量来源于三部分：所有A_jB_b计算中的乘法计算量MK_cN_c、所有A_jB_b计算中的加法计算量M$(K_c - 1)N_c$和更新所有C_j的加法计算量MN_c。而更新C_j的运算量只占总运算量的$\frac{1}{2K_c}$。为了提高关键计算A_jB_b的计算量比重，降低更新C_j的运算量比重，K_c的选择也是越大越好。

由公式 3-8 以及大量 BLAS 中矩阵乘法分块实现的文献研究表明，N_c与K_c大致相等时效果最好，即B_b大致为方阵时矩阵乘法分块实现的效果最好。但在实际情况中，应结合其他各方面因素考虑N_c及K_c的参数选择。

由上述分析可知，N_c与K_c的参数选择越大矩阵乘法实现效果越好。但现代计算平台多级存储结构中 Cache 容量比较小，要将A_j、B_b和C_j全部存至 Cache 中不太现实，所以考虑将B_b移至 L2Cache 中存储，A_j和C_j依然存储至 L1Cache 中。

B_b移至 L2Cache 中存储之后，将B_b中的数据从 L2Cache 搬移至 Register 中需要一定的代价，即 L2Cache 与 Register 之间的带宽限制。在分块矩阵乘法$C_j = A_j B_b$中，计算量为$2M_r K_c N_c$，将B_b从 L2Cache 中搬移 Register 的访存量为$K_c N_c$。在假设二：L1Cache 与 Register 之间存在足够带宽以至于搬移C_j和A_j不能成为计算矩阵乘法$C_j = A_j B_b$障碍的前提下，假设计算平台的处理器从 Register 取得数据进行计算的速率为V_{comp}，L2Cache 与 Register 之间的带宽为V_{L2Load}，为保证将B_b从 L2Cache 中搬移 Register 满足计算的需求，应满足下列不等式：

$$\frac{2M_r K_c N_c}{V_{comp}} \geq \frac{K_c N_c}{V_{L2Load}} \quad \text{(公式 3-9)}$$

即

$$M_r \geq \frac{V_{comp}}{2V_{L2Load}} \quad \text{(公式 3-10)}$$

综上所述，核心循环中A_b矩阵分块参数M_c、K_c的选取应越大越好，但受到多级存储结构中 Cache 大小的限制；在假设二：L1Cache 与 Register 之间存在足够带宽以至于搬移A_b和B_j不能成为计算矩阵乘法$C_j = A_j B_b$障碍的前提下，核心循环A_j矩阵分块参数M_r的选取受限于多级存储结构中 L2Cache 与 Register 之间的带宽大小。

2. 基于 Register 级的分块分析

在现代计算平台的多级存储结构中，Register 的容量和数量是极其有限的。在计算分块矩阵乘法$C_j = A_j B_b$中不可能将所需的分块矩阵数据B_b、A_j和C_j全部存储至 Register 中，考虑将B_b进一步以每N_r列拆分，每N_r列分别标记为B_{b_j}，

$$B_p = (B_{b_0}\ B_{b_1}\ B_{b_3}\ \cdots\ B_{b_j}\ \cdots)$$

因此矩阵乘法$A_j B_b$将进一步分块运算，以此更新C_j的$M_r \times N_r$子矩阵，如下图所示。

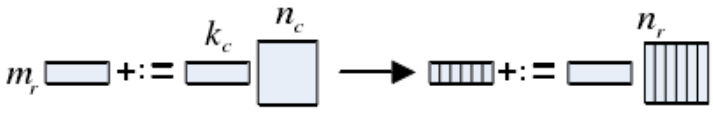

图 0-19　基于 Register 的 GEPB 示意图

在上述进一步的分块运算中，有关C_j的$M_r \times N_r$子矩阵的运算量为$2M_rN_rK_c$，访存量来源于四部分：将B_{b_j}从内存搬移至 Cache 的访存量K_cN_r、将A_j从内存搬移至 Cache 的访存量M_rK_c、从内存中将C_j的$M_r \times N_r$子矩阵读取至 Cache 的访存量M_rN_r和将C_j的$M_r \times N_r$子矩阵从 Cache 更新至内存的访存量M_rN_r，总访存量为$K_cN_r + M_rK_c + 2M_rN_r$，其中有关$C_j$的$M_r \times N_r$子矩阵的访存量为$2M_rN_r$。访存比为：

$$访存比 = \frac{2M_rK_cN_r}{K_cN_r+M_rK_c+2M_rN_r} = \frac{2M_rN_r}{(M_r+N_r)+{}^{2M_rN_r}/_{K_c}} \approx \frac{2M_rN_r}{M_r+N_r} \quad \text{(公式 3-11)}$$

当$K_c \gg M_r$时，有关C_j的$M_r \times N_r$子矩阵的总访存量$2M_rN_r$相对于运算量$2M_rN_rK_c$是非常小的，并且$2M_rN_rK_c$的运算量中有关C_j的$M_r \times N_r$子矩阵的也只有M_rN_r个加法。并且对C_j的$M_r \times N_r$子矩阵的访问术语流水式访问模式，即读入->计算->更新。考虑到 L1Cache 容量的紧缺性，C_j的$M_r \times N_r$子矩阵一直停留在 L1Cache 不是一个高效的做法，应将其以及C_j存储至下级 Cache 中。

综上所述，考虑到 Register 的容量及数量极其有限的原因，应将A_b分块为更小的矩阵A_{b_j}，同时由公式 3-11 可知，Register 确实是制约矩阵乘法分块高性能实现的一个重要原因。考虑到有关C_j的计算量与访存量在整个矩阵乘法分块计算中的低比重以及对C_j的流水式访问模式，应将C_j存储至 L1Cache 的下级 Cache 中。最后，考虑到将A_b分块后的各块的数据访问方向与矩阵在内存中的存储方向不一致的情况，将涉及到矩阵的数据封装与搬移的问题。矩阵的数据封装与搬移问题将在 3.3 节中进行详细阐述。

3. 基于硬件资源的分块参数选择

在上述的分析中，涉及到 4 个参数：N_c、N_r、K_c和M_r。接下来，将结合相应计算平台的体系结构及硬件资源，对涉及的 4 个参数进行最优化选取配置。

C_j的子矩阵参数M_r和N_r的选取：

(1)应满足公式 3-10；

(2)考虑到多级存储结构中 Register 有限的个数，其中${}^1/_2$用于存储C_j的$M_r \times N_r$子矩阵，剩余部分用于存储矩阵分块运算A_bB_j所需的数据；

(3)考虑到公式 3-11，在M_r和N_r尽可能大且$M_r \approx N_r$时访存比的值较大，性能较佳。

综上所述，结合相应计算平台的体系结构和硬件资源，最终确定$M_r = 4$，$N_r = 4$。

参数K_c的选取：

(1)考虑到公式 3-11，为了降低更新C_j的$M_r \times N_r$子矩阵所需的开销，K_c的选取应越大越好；

(2)大小为$M_r \times K_c$的矩阵A_j存储至 L1Cache 中，为了减少 L1Cache 中发生的冲突，增加 Cache 命中率，大小为$M_r \times K_c$的矩阵A_j在 L1Cache 中所占的空间应尽量低于 L1Cache 的$1/2$；

(3)大小为$K_c \times M_c$的矩阵B_b存储至 L2Cache 中，为了减少 L2Cache 中发生的冲突，增加 Cache 命中率，大小为$K_c \times M_c$的矩阵B_b在 L2Cache 中所占的空间应尽量低于 L2Cache 的$1/2$。

参数N_c的选取：

(1)考虑到公式 3-8，在K_c和N_c尽可能大且$K_c \approx N_c$时访存比的值较大，性能较佳；

(2)参数N_c的选取还受限于大小为$K_c \times M_r$的矩阵A_j和大小为$N_c \times K_c$的矩阵B_b的参数选取。

综上所述，结合相应计算平台的体系结构和硬件资源，最终确定$N_c = 256$，$K_c = 128$。

1.2.3 GEPDOT 的分析与设计

1. 基于 Cache 级的分块分析

本小节中，仅在 Cache 级分析核心循环 GEPDOT 的分块实现。假设 Block C 的规模为$M_c \times N_c$，Panel A 的规模为$M_c \times$ K，则 Panel B 的规模为K $\times N_c$，如下图所示：

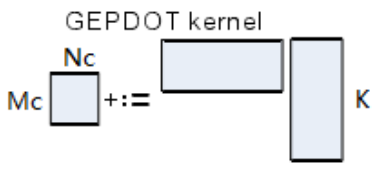

图 0-20

接下来将 Panel A 和 Panel B 以每K_r列拆分，每K_r列分别标记为A_j和B_j，如下所示：

$$A_p = \begin{pmatrix} A_0 & A_1 & A_2 & \cdots & A_j & \cdots \end{pmatrix}, B_p = \begin{pmatrix} B_0 \\ B_1 \\ \vdots \\ B_j \\ \vdots \end{pmatrix}$$

并将 Block C 标记为C_b，即$C_b = A_j B_j$，如下图所示：

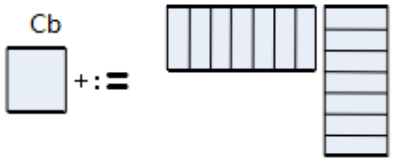

图 0-21

为了支撑下文分析，此处对基于 Cache 的多级存储模型中 L1Cache 与 Register 之间带宽和A_b、B_j和C_j的总体数据量作出相应假设：

假设 1：C_b、A_j和B_j的总体数据量足够全部存储至 L1Cache 中；

假设 2：若C_b、A_j和B_j均存储至 L1Cache，则在处理器中计算矩阵乘法$C_b = A_j B_j$时能够接近处理器浮点计算峰值，即 L1Cache 与 Register 之间存在足够带宽以至于搬移C_b、A_j和B_j不能成为计算矩阵乘法$C_b = A_j B_j$的障碍。

下图为核心循环 GEPDOT 分块实现示意图。从图中可知，核心循环 GEPDOT 分块实现的访存量来源于四部分：将所有A_j从内存搬移至 Cache 的访存量$M_c K$、将所有B_j从内存搬移至 Cache 的访存量$K N_c$、将需要更新的C_b从内存搬移至 Cache 的访存量$M_c N_c$和将所有完成更新的C_j从 Cache 搬移至内存的访存量$M_c N_c$。核心循环 GEPDOT 分块实现的访存量总共为$M_c K + K N_c + 2 M_c N_c$。核心循环 GEPDOT 分块实现的运算量来源于三部分：所有$A_j B_j$计算中的乘法计算量$M_c K N_c$、所有$A_j B_j$计算中的加法计算量$M_c (K-1) N_c$和更新C_b的加法计算量$M_c N_c$，总运算量为$2 M_c K N_c$。

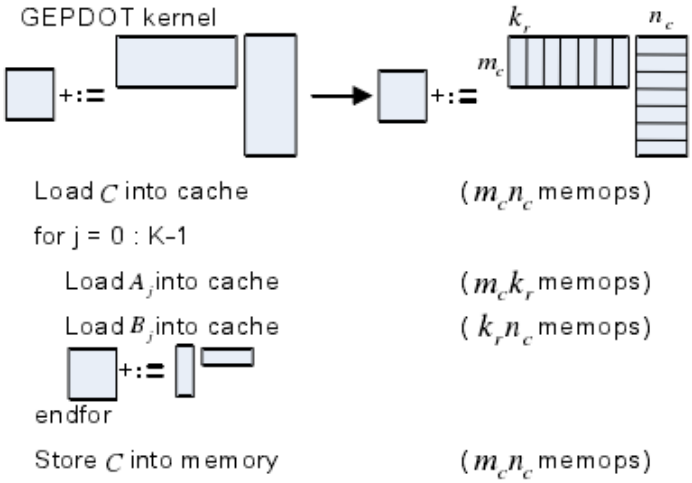

图 0-22　GEDOT 分块实现示意图

那么核心循环 GEPDOT 分块实现的访存比为：

$$访存比 = \frac{2M_cKN_c}{M_cK+KN_c+2M_cN_c} = \frac{2M_cKN_c}{(M_c+N_c)K+2M_cN_c} \quad \text{(公式 3-11)}$$

当 K >> N_c且 K >> N_c时

$$访存比 \approx \frac{2M_cN_c}{M_c+N_c} = \frac{2}{1/M_c+1/N_c} \quad \text{(公式 3-12)}$$

访存比决定着计算量与相应计算量所需访存量之间的比值。为了降低访存对计算的限制，突出计算的主导地位，则必须使访存比较大化。要使访存比较大化，则要在计算平台多级存储结构中 Cache 大小的限制下使得M_cN_c尽量的大。

由上述分析可知，M_c与N_c的参数选择越大矩阵乘法分块实现效果越好。但现代计算平台多级存储结构中 Cache 容量比较小，要将C_b、A_j和B_j全部存至 Cache 中不太现实，所以考虑将C_b移至 L2Cache 中存储，A_j和C_j依然存储至 L1Cache 中。

C_b移至 L2Cache 中存储之后，将C_b中的数据从 L2Cache 搬移至 Register 中需要一定的代价，即 L2Cache 与 Register 之间的带宽限制。在分块矩阵乘法$C_b = A_jB_j$中，计算量为$2M_cK_rN_c$，将C_b在 L2Cache 和 Register 之间搬移的访存量为 $2M_cN_c$。在假设二：L1Cache 与 Register 之间存在足够带宽以至于搬移A_j和B_j不能成为计算矩阵乘法$C_b = A_jB_j$障碍的前提下，假设计算平台的处理器从 Register 取得数据进行计算的速率为V_{comp}，L2Cache 与 Register 之间的带宽为V_{L2Load}，为保证将C_b从 L2Cache 中搬移 Register 满足计算的需求，应满足下列不等式：

$$\frac{2M_cK_rN_c}{V_{comp}} \geq \frac{2M_cN_c}{V_{L2Load}} \quad \text{(公式 3-13)}$$

即

$$N_r \geq \frac{V_{comp}}{V_{L2Load}} \quad \text{(公式 3-14)}$$

与核心循环 GEBP 和 GEPB 相比，GEPDOT 则需要 2 倍 L2Cache 与 Register 之间的带宽，这限制了基于核心循环 GEPDOT 的矩阵乘法分块实现的性能。由此，矩阵乘法分块实现不应基于核心循环 GEPDOT 实现。

1.3 GEMM 的单核实现

在上节中，对矩阵乘法分块实现中的三种核心循环进行了详细的分析，从 Cache 级和 Register 级对分块参数的影响因素给予了理论上的分析，并结合相应计算平台的体系结构和硬件资源对分块参数进行了选取配置，以达成在相应计算平台上矩阵乘法的分块高性能实现。

本节将从最简单的基于三级循环的 GEMM 实现开始，首先分析代码层面性能受限部分，加入避免使用乘法指令、使用 Register 变量、循环展开和显式 SIMD 四种代码级优化策略，在矩阵乘法问题规模较小时实现了较大的性能提升（由下节性能测试与分析可知）。为了解决矩阵乘法问题规模较大时性能提升较小的问题，接下来对矩阵乘法进行分块实现。由于本文中所采用的矩阵使用列主序存储，所以采用了上节中的基于核心循环 GEBP 的 GEPP 对矩阵乘法进行分块实现，在矩阵乘法问题规模较大时实现了较大的性能提升（由下节性能测试与分析可知）。最后对分块矩阵进行了数据封装和搬移，在一定程度上对矩阵乘法分块实现性能进行了提升（由下节性能测试与分析可知）。最终，实现了一个对代码层、算法层和数据层进行优化后的 GEMM 单核实现。

1.3.1 基于三级循环的 GEMM 实现

在线性代数中，矩阵乘法的本质即是一个乘加的三级循环。基于三级循环乘加的 GEMM 实现简单，结构清晰，并且无论矩阵问题规模如何，都可基于三级循环乘加对 GEMM 进行实现。基于三级循环乘加实现 GEMM 的缺点也比较明显：代码级编写不够高效、未考虑局部性原理乃至无法充分利用相应计算平台的硬件资源。这也给矩阵乘法实现留下了巨大的提升空间。

基于三级循环的 GEMM 算法实现伪代码如下：

Algorithm 3.1: 基于三级循环的 GEMM 算法

Input: Matrix A, Matrix B

Output: Matrix C

```
1.  for i = 0:M-1
2.      for j = 0:N-1
3.          for p = 0:K-1
4.              C_ij += A_ip B_pj
5.          end for
6.      end for
7.  end for
```

1.3.2 代码级优化技巧

1. 避免使用乘法指令

乘法指令因涉及复杂的控制逻辑和运算逻辑，在计算机的物理实现中，乘法指令的开销远大于加法指令的开销。与此同时，乘法指令本质即为多条加法指令的有序组合，因此在计算机中乘法指令可由多条加法指令和一些控制指令实现，这样可以降低执行乘法指令的开销。在一些特殊场景中，通过将乘法指令转化为加法指令来降低开销显得尤为重要。

在矩阵元素存取的过程中，获取元素内存地址需要执行一条乘加指令。若接下来要存取相邻矩阵元素的地址时，无需再执行一条乘加指令，只需用指针保存上一矩阵元素的地址，通过上一矩阵元素地址执行加减运算偏移即可。

例如在对列主序的矩阵 B 进行矩阵元素存取时，要获取矩阵元素$B_{i,j}$的内存地址需要执行一条乘加指令：

$$\text{j} \times \text{B 的行数} + \text{i}$$

并用指针保存下来。若在接下来要存取矩阵元素$B_{i+1,j}$，则需将指针加一即可，避免使用乘加指令，降低了计算开销。

2. 使用 Register 变量

在计算平台的多级存储结构中，Register 最接近处理器同时存取速度也最快。Register 变量[21]即为存储至 Register 中的变量。在 C 语言中，定义 Register 变量的指令较为简单，在普通的变量定义指令中数据类型前加上 register 即可，例如：

```
1. register int a;
```

即为定义一个 int 变量 a 并将其存储至 Register 中。注意 Register 变量的定义是建议型指令，定义了一个 Register 变量不代表会将变量存储至 Register 中。由于 Register 的容量及数量极其有限，不应随意将变量定义为 register 变量，应将计算频繁型变量定义为 register 变量，以高效利用多级存储结构中的 register 资源，提高计算效率。

3. 循环展开

循环开展（Loop Unrolling）[22]即通过增加每次迭代过程中的循环尺寸，减少迭代次数的一种代码级提高计算性能的方法。由于迭代次数的减少，迭代过程中分支预测失败的次数将会减少，即分支预测失败的开销将会降低。同时也会减少迭代过程中需要使用到的资源量及计算量，如计算循环索引的开销。由于扩展了每次迭代过程中的计算量，对于多核多线程的计算平台，可使用并行计算指令从代码级提高并行的可能性。

循环展开代码级优化技巧有两种使用方式：自动方式和手动方式。自动方式即在编译器选项中填写-funroll -loops 即可。手动方式进行循环展开需手动控制展开次数[23]。展开次数过低可能对于计算性能的提升作用不大，展开次数过高可能造成代码膨胀，降低迭代程序代码可读性。通常情况下，展开次数在 4 左右可在计算性能提升较大的同时不造成代码膨胀和程序代码可读性降低的负面效果。在某些特殊情况下，手动循环展开可能会误导编译器，对计算性能造成负面影响。

展开因子为 4 的循环展开后基于三级循环的 GEMM 算法实现伪代码如下：

Algorithm 3.2: 基于三级循环的 GEMM 算法

Input: Matrix A, Matrix B

Output: Matrix C

```
1.  for i = 0:M-1:4
2.      for j = 0:N-1:4
3.          for p = 0:K-1
```

4. $C_{ij} += A_{ip}B_{pj}$
5. end for
6. end for
7. end for

4. 显式 SIMD

按照 Flynn 分类法，并行计算机分为四类：SISD、SIMD、MISD 和 MIMD。其中 SIMD 即单指令流多数据流并行计算机，能够将一组数据打包至较大存储量的寄存器中，并对其所有数据进行相同指令对应的操作，这些操作具有相对独立、相同本质的特点。下图为 SIMD 并行计算机，PU 代表程序执行单元。

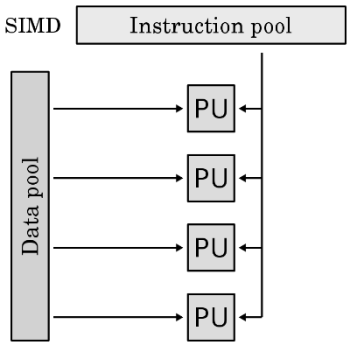

图 0-23 SIMD 并行计算机示意图

大部分处理器都有对应的 SIMD Intrinsic 指令集，如 Intel 平台的 SSE 指令集[24]、ARM 平台的 Neon 指令集。显式 SIMD 即通过在计算程序中手动插入相应计算平台的 SIMD Intrinsic 指令集中的指令达到对一组数据同时进行操作。显式 SIMD 由于能够对一组数据同时进行操作，进而可以从数据级提高程序的并行性。此外处理器对应的 SIMD Intrinsic 指令集在底层对于相应处理器体系结构进行了深度优化，能够充分利用对应处理器体系结构和硬件资源，从而提高程序的计算和访存效率。

SSE 从某种程度上如同一个矢量处理单元，矢量处理单元即专门用于对矢量进行处理。在 Intel 平台上，使用 SSE 矢量处理指令集有两种方式：

1. 直接在 C/C++程序中插入用于实现 SSE 指令集中函数的汇编语句；
2. 在 C/C++程序中使用 SSE 头文件封装好的矢量 SIMD Intrinsics 函数。

直接在 C/C++程序中插入用于实现 SSE 指令集中函数的汇编语句的方法为编译器省去了将 C/C++语句编译成汇编语句的过程，提高了程序执行性能，但要求程序编写人员掌握对 Intel 平台用于实现 SSE 指令集中函数的汇编语句，并且由于使用汇

编语句进行程序编写，代码可读性和程序维护性会大大降低。矢量 SIMD Intrinsics 函数是对实现 SSE 的汇编指令进行的一种封装，以函数接口的形式提供给程序编写人员，目的在于为程序编写人员提供一种高效简单地实现 SSE SIMD 矢量处理的方法。矢量 SIMD Intrinsics 函数不会产生函数调用带来的开销。在程序编译过程中，编译器已将 Intrinsics 函数编译为用于实现 SSE 的汇编指令，相当于直接在程序中顺序执行汇编指令，不同于传统意义上的函数调用。使用 SSE Intrinsics 函数的一个大前提是在 C/C++中包含相应的头文件：

```
1. #include <mmintrin.h>     //mmx
2. #include <xmmintrin.h>    //sse
3. #include <emmintrin.h>    //sse2
4. #include <pmmintrin.h>    //sse3
```

SSE 指令集中的指令包含两种类型：数据类型定义指令和数据操作指令。因为 SIMD 需同时处理一组数据，SSE 指令集中的数据类型定义指令定义的数据类型通常程度上比 C/C++中的数据类型定义指令定义的数据类型有更大的长度。数据操作指令通常程度上也比 C/C++中数据操作指令有着更大的数据处理长度。更大的数据处理长度意味着更高效的数据级并行性。SSE 指令集中的数据处理指令通常分为三种类型：数据搬移指令、内存处理指令、数据运算指令。接下来将讲解几个在 GEMM 的单核实现中用到的 SSE 指令。

在 GEMM 的单核实现中用到的数据类型定义 SSE 指令有__m128d：

```
1. typedef struct __declspec(align(16)) {
2.         double d[2];
3. } __m128d;
```

__m128d 本质占据 2 个 double 类型的空间，即占据 128 位的数据空间。

在 GEMM 的单核实现中用到的数据搬移 SSE 指令有_mm_load_pd 函数和_mm_loaddup_pd 函数：

```
__m128d _mm_load_pd(double const* men_addr);
__m128d _mm_loaddup_pd(double const* men_addr);
```

其中_mm_load_pd 函数的功能是将以 men_addr 为起始地址的 128 位数据作为函数的返回值。_mm_loaddup_pd 函数的功能是将以 men_addr 为起始地址的 64 位数据复制 1 份并将 2 份 64 位数据拼接为 128 位数据作为函数的返回值，以达到数据搬

移的目的。

在 GEMM 的单核实现中用到的内存处理 SSE 指令有_mm_setzero_pd 函数：

```
__m128d _mm_setzero_pd(void)
```

_mm_setzero_pd 函数的作用是返回一个数据值为 0 的__m128d 数据类型，用于变量的初始化操作。

在 GEMM 的单核实现中用到的数据运算 SSE 指令有同 C/C++中的数据运算指令相同，这里不再赘述。

1.3.3 GEMM 分块实现（基于 GEBP 的 GEPP 实现）

矩阵的存储形式为分为行主序和列主序。存储形式为行主序的矩阵内存布局如下图：

图 0-24 行主序的矩阵内存布局图

存储形式为列主序的矩阵内存布局如下图：

图 0-25 列主序的矩阵内存布局图

本文矩阵乘法中所采用矩阵的存储形式为列主序，在 3.1.2 小节中得出的结论可知：存储形式为列主序的矩阵乘法选择基于 GEBP 的 GEPP 实现的分块实现方案有更高的实现效率。所以下文将讨论基于 GEBP 的 GEPP 的矩阵乘法分块实现方案。

假设矩阵 A 的规模为$M \times K$，矩阵 B 的规模为$K \times N$，矩阵 C 的规模为$M \times N$。基于 GEBP 的 GEPP 的矩阵乘法分块实现方案的图解如下：

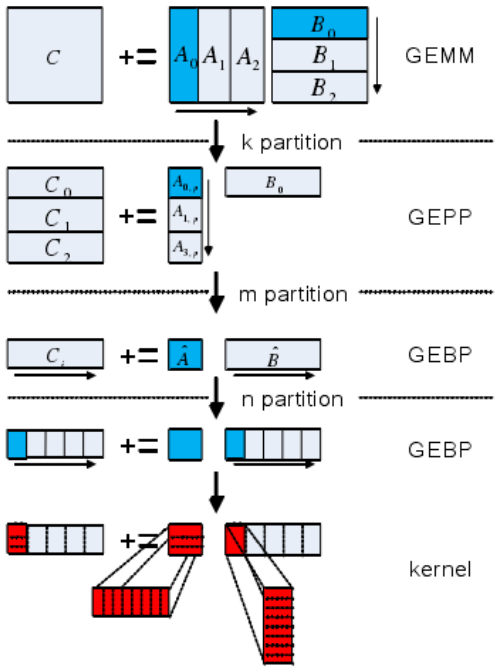

图 0-26　基于 GEBP 的 GEPP 的 GEMM 实现

图中分为五层，在第一层 GEMM 层中，将矩阵 A 以每K_c列进行分块，得到的 A 分块矩阵如下：

$$A = (A_0 \ A_1 \ A_2 \ \cdots \ A_{k-1} \ \cdots)$$

将矩阵 B 以每K_c行进行分块，得到的 B 分块矩阵如下：

$$\mathrm{B} = \begin{pmatrix} B_0 \\ B_1 \\ \vdots \\ B_{k-1} \\ \vdots \end{pmatrix}$$

将矩阵 A、B 按上述分块方案进行分块后，按照矩阵乘法的分块计算机理，GEMM 被分解为若干个 GEPP 矩阵乘法相加：

$$C = A_0B_0 + A_1B_1 + \cdots + A_{k-1}B_{k-1} \qquad \text{(公式 3-15)}$$

在第二层 GEPP 层中，将矩阵A_p以每M_c行进行分块，得到的A_p分块矩阵如下：

$$A_p = \begin{pmatrix} A_{0,p} \\ A_{1,p} \\ \vdots \\ A_{m-1,p} \\ \vdots \end{pmatrix}$$

将A_p按上述分块方案进行分块后，GEPP 被分解为了若干个 GEBP 矩阵乘法相加：

$$A_pB_p = A_{0,p}B_p + A_{1,p}B_p + \cdots + A_{m-1,p}B_p \qquad \text{(公式 3-16)}$$

在第四层 GEBP 层中，将矩阵B_p以每N_r行进行分块。在第五层中，将矩阵$A_{i,p}$以每

M_r行进行分块，分析同上。在上述所有分块完成之后，最内层实现即为 GEMM MrxNr kernel。GEMM kernel 在实现的过程中运用了上文提到的避免使用乘法指令、使用 Register 变量、循环展开和显式 SIMD 四种代码级优化策略。

根据上述分块过程的 GEMM 分块实现（基于 GEBP 的 GEPP 实现）实现算法伪代码如下：

Algorithm 3.3: GEMM 分块实现（基于 GEBP 的 GEPP 实现）实现算法

Input: Matrix A, Matrix B

Output: Matrix C

```
1.  for i=0:Kc:K-1
2.      for j=0:Mc:M-1
3.          for k=0:Nr:N-1
4.              for p=0:Mr:Mc-1
5.                  GEMM MrxNr kernel
6.              end for
7.          end for
8.      end for
9.  end for
```

1.3.4 分块数据的封装与搬移

在上小节的分析中，在第二层 GEPP 层中将矩阵A_p以每M_c行进行分块，得到的A_p分块矩阵如下：

$$A_p = \begin{pmatrix} A_{0,p} \\ A_{1,p} \\ \vdots \\ A_{m-1,p} \\ \vdots \end{pmatrix}$$

在上述分块完成后，在第五层中，将矩阵$A_{i,p}$以每M_r行进行分块，得到的$A_{i,p}$分块矩阵如下：

$$A_{i,p} = \begin{pmatrix} A_{0,i,p} \\ A_{1,i,p} \\ \vdots \\ A_{m-1,i,p} \\ \vdots \end{pmatrix}$$

在矩阵$A_{i,p}$的使用中，数据移动方向为$A_{0,i,p}$、$A_{1,i,p}$、$A_{2,i,p}$、…。本文中矩阵乘法分块实现所使用矩阵的存储形式为列主序，根据内存块搬移的空间局部性原理[25]，将矩阵$A_{i,p}$中的数据是一列接一列的搬移至 Cache 中，这与矩阵$A_{i,p}$实际数据使用方向不一致，从而会降低 Cache 的命中率，从而限制矩阵乘法分块实现的性能。为了解决这一问题，采用了对矩阵$A_{i,p}$提前封装的方法，如下图，即将$A_{i,p}$以$A_{0,i,p}$、$A_{1,i,p}$、$A_{2,i,p}$、…的顺序在使用前提前搬移至一连续的内存空间中，以使得矩阵$A_{i,p}$实际数据使用方向与根据内存块局部性原理达成的内存与 Cache 之间搬移数据的方向一致。

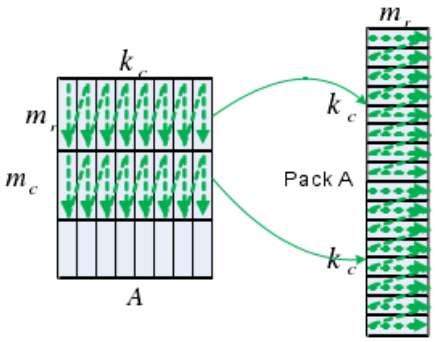

图 0-27　分块矩阵 A 数据封装与搬移图

在第四层 GEBP 层中，将矩阵B_p以每N_r行进行分块，得到的B_p分块矩阵如下：

$$B_p = (B_{0,p}\ \ B_{1,p}\ \ B_{2,p}\ \ \cdots\ \ B_{j,p}\ \ \cdots)$$

在矩阵B_p的使用中，数据移动方向为$B_{0,p}$、$B_{1,p}$、$B_{2,p}$、…。本文中矩阵乘法分块实现所使用矩阵的存储形式为列主序，根据内存块搬移的空间局部性原理，将矩阵B_p中的数据是一列接一列的搬移至 Cache 中，这与矩阵B_p实际数据使用方向不一致，从而会降低 Cache 的命中率，从而限制矩阵乘法分块实现的性能。为了解决这一问题，采用了对矩阵B_p提前封装的方法，如下图，即将B_p以$B_{0,p}$、$B_{1,p}$、$B_{2,p}$、…的顺序在使用前提前搬移至一连续的内存空间中，以使得矩阵B_p实际数据使用方向与根据内存块局部性原理达成的内存与 Cache 之间搬移数据的方向一致。

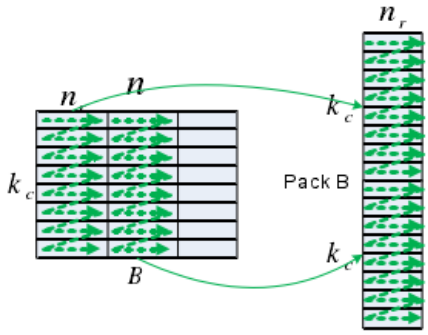

图 0-28　分块矩阵 B 数据封装与搬移图

在上小节的基础上加上分块数据的封装与搬移后的 GEMM 分块实现算法伪代码如下：

Algorithm 3.4: GEMM 分块实现（完善分块数据的封装与搬移）实现算法

Input: Matrix A, Matrix B

Output: Matrix C

```
for i=0:Kc:K-1
    PACK A
    for j=0:Mc:M-1
        PACK B
        for k=0:Nr:N-1
            for p=0:Mr:Mc-1
                GEMM kernel
            end for
        end for
    end for
end for
```

1.4　性能测试与分析

在上节中，对本文中 GEMM 单核实现的代码级优化技巧、分块乘法方案和分块数据封装和搬移技巧进行了分析和相应实现。

具备上节的基础后，本节首先将基于三级循环的 GEMM 实现与加入上文阐述

的四种代码级优化技巧后的基于三级循环的 GEMM 实现以 GFLOPS 为评价方式进行实现性能方面的比较，观察到在矩阵问题规模较小时加入代码级优化技巧后的基于三级循环的 GEMM 实现比无任何优化的基于三级循环的 GEMM 实现有较大的性能提升，但在矩阵问题规模较大时性能提升较小，还有充足的提升空间。为了解决加入代码级优化技巧后的基于三级循环的GEMM实现在矩阵问题规模较小大时性能提升较小的问题，将 GEMM 的实现方案更改为了基于 GEBP 的 GEPP 实现，代码级优化技巧仍然使用在实现当中，观察到与加入代码级优化技巧后的基于三级循环的 GEMM 实现相比，在矩阵问题规模较大时有了更多的性能提升。最后考虑到多级存储结构中内存与 Cache 之间内存块搬运的局部性原理，在基于 GEBP 的 GEPP 的 GEMM 实现的基础上，对分块矩阵进行了数据封装与搬移，增加了 Cache 的命中率，在整体上了有了相应的性能提升。最终，优化后的 GEMM 单核实现性能以 GFLOPS 为评价方式时相比最原始的基于三级循环的 GEMM 单核实现提升了 7 倍性能。

1.4.1 代码级优化后的 GEMM 性能测试与分析

基于三级循环的 GEMM 实现与加入上文阐述的四种代码级优化技巧后的基于三级循环的 GEMM 实现以 GFLOPS 为评价方式进行实现性能方面的比较的结果图如下：

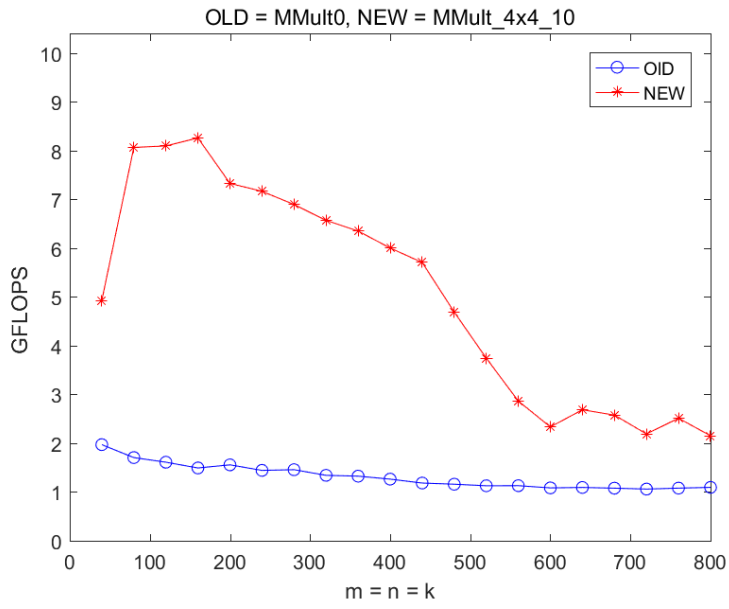

图 0-29 (OLD)基于三级循环的 GEMM 实现与(NEW)加入上文阐述的四种代码级优化技巧后的基于三级循环的 GEMM 实现性能比较图

从图中可见，加入上文阐述的四种代码级优化技巧后的基于三级循环的 GEMM 实现在矩阵规模较小时相对于基于三级循环的 GEMM 实现提升的性能比较大，从四种代码级优化策略分析性能提升的原因可知：避免使用乘法指令降低了指令运算复杂度，减少了运算器的时间开销；使用 Register 变量加速了相应计算平台多级存储结构之间数据搬移速度，降低了数据搬移的时间开销；循环展开降低了迭代过程中分支预测失败开销和计算循环索引有关的开销，同时为数据级并行计算提供了可能性；显式 SIMD 提高了 GEMM 实现过程中的数据并行性。

在矩阵规模变大后，加入上文阐述的四种代码级优化技巧后的基于三级循环的 GEMM 实现的性能提升效果明显降低，推测其原因，可能是：当矩阵规模较小时，Cache 中能够容纳大部分的矩阵元素数据，Cache 的命中率相对较高，Cache 与内存之间换进换出的次数相对较少。当矩阵规模变大后，由于矩阵数据元素存储的不连续性以及 Cache 空间的限制，Cache 的命中率逐渐变低，Cache 与内存之间换进换出的次数逐渐提高，以至于造成在矩阵规模变大后，性能提升效果明显降低的现象。为了解决这一问题，将 GEMM 计算方式从基于三级循环的实现改变为基于 GEBP 的 GEPP 实现。

1.4.2 基于 GEBP 的 GEPP 实现的 GEMM 性能测试与分析

基于 GEBP 的 GEPP 实现的 GEMM 与加入上文阐述的四种代码级优化技巧后的基于三级循环的 GEMM 实现以 GFLOPS 为评价方式进行实现性能方面的比较的结果图如下：

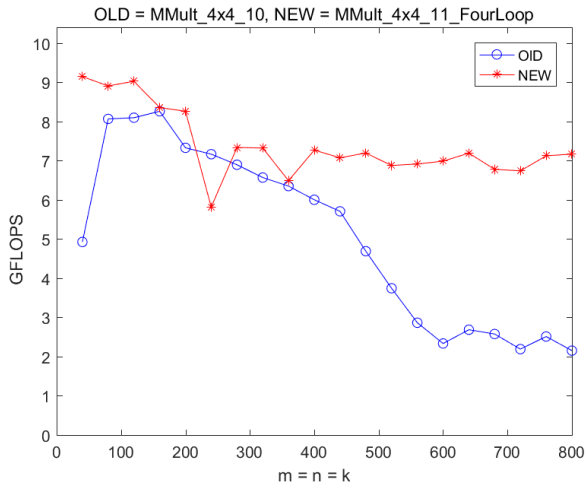

图 0-30 (NEW)基于 GEBP 的 GEPP 实现的 GEMM 与(OLD)加入上文阐述的四种代码级优化技巧后的基于三级循环的 GEMM 实现性能比较图

从图中可以看出，基于 GEBP 的 GEPP 实现的 GEMM 与加入上文阐述的四种代码级优化技巧后的基于三级循环的 GEMM 实现相比在矩阵规模较大时性能提升较大，这解决了加入上文阐述的四种代码级优化技巧后的基于三级循环的 GEMM 实现与基于三级循环的 GEMM 实现相比在矩阵规模较大时性能提升较小的瓶颈。可知，基于 GEBP 的 GEPP 实现的 GEMM 优化了矩阵乘法的分块方式，在矩阵规模较大时提高了 Cache 的命中率，降低了 Cache 与内存之间换进换出的次数。

1.4.3 封装和搬移分块数据后的 GEMM 性能测试与分析

加入分块数据封装与搬移的基于 GEBP 的 GEPP 的 GEMM 实现与无任何优化的基于 GEBP 的 GEPP 的 GEMM 实现以 GFLOPS 为评价方式进行实现性能方面的比较的结果图如下：

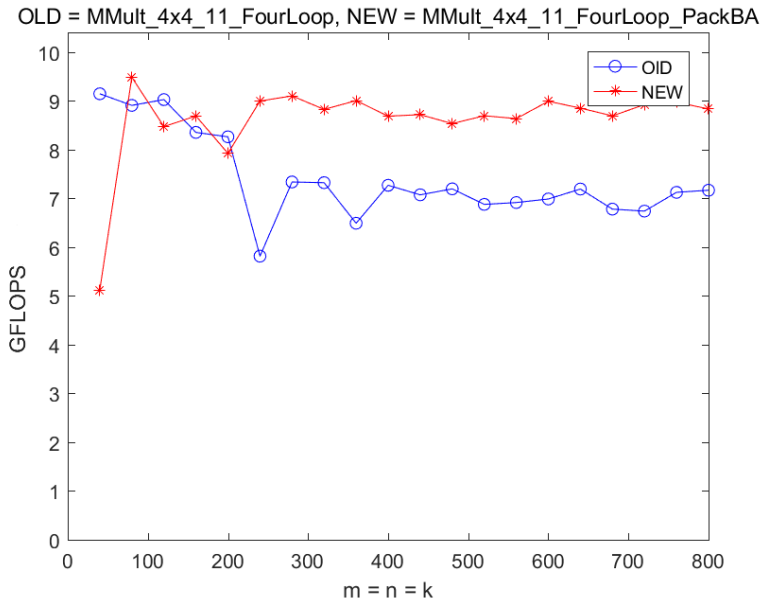

图 0-31 (NEW)加入分块数据封装与搬移的基于 GEBP 的 GEPP 的 GEMM 实现与(OLD)无任何优化的基于 GEBP 的 GEPP 的 GEMM 实现性能比较图

从图中可知，加入分块数据封装与搬移的基于 GEBP 的 GEPP 的 GEMM 实现在矩阵规模在 200 以下时性能几乎无任何提升，在矩阵规模为 200 以上的各级矩阵规模的性能提升相对平稳。

下图是加入分块数据封装与搬移的基于 GEBP 的 GEPP 的 GEMM 实现与无任何优化的基于三级循环的 GEMM 实现以 GFLOPS 为评价方式进行实现性能方面的比较图：

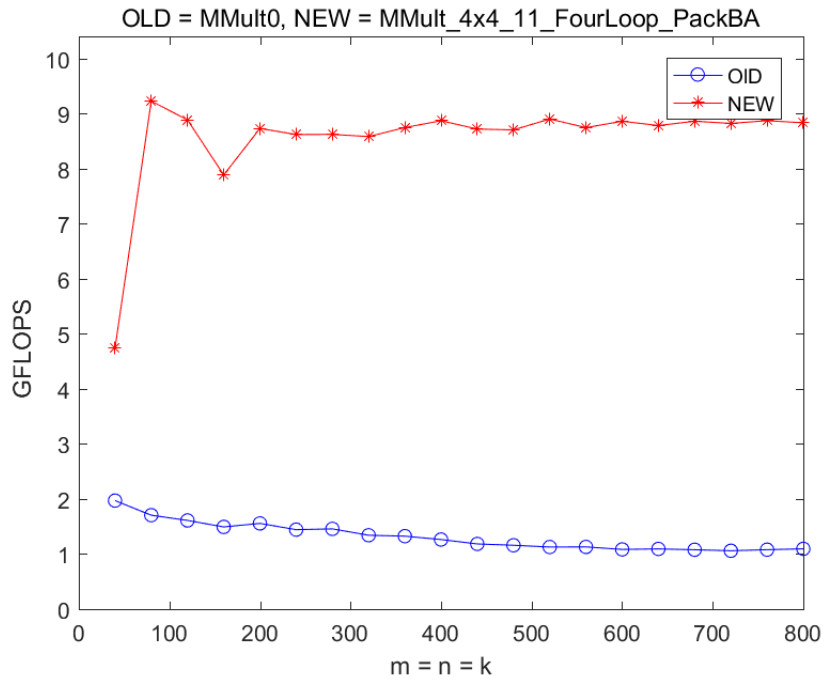

图 0-32 (NEW)加入分块数据封装与搬移的基于 GEBP 的 GEPP 的 GEMM 实现与(OLD)无任何优化的基于三级循环的 GEMM 实现性能比较图

从图中可知，加入分块数据封装与搬移的基于 GEBP 的 GEPP 的 GEMM 实现与无任何优化的基于三级循环的 GEMM 实现相比，性能提升了将近 8 倍，且在各级矩阵规模的提升都相对平稳。

推测限制分块数据封装与搬移的基于 GEBP 的 GEPP 的 GEMM 实现的性能提升的原因，是本章 GEMM 为单核实现，实现 GEMM 的处理器核心数和线程数不足，各级计算的并行度不足。在下章中将讨论 GEMM 的多核实现，测试和分析在多处理器和多线程环境中 GEMM 实现的性能结果。