Anatomy of High-Performance Matrix Multiplication

KAZUSHIGE GOTO
The University of Texas at Austin
and
ROBERT A. VAN DE GEIJN
The University of Texas at Austin

We present the basic principles which underlie the high-performance implementation of the matrix-matrix multiplication that is part of the widely used GotoBLAS library. Design decisions are justified by successively refining a model of architectures with multilevel memories. A simple but effective algorithm for executing this operation results. Implementations on a broad selection of architectures are shown to achieve near-peak performance.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: —*Efficiency*

General Terms: Algorithms;Performance

Additional Key Words and Phrases: linear algebra, matrix multiplication, basic linear algebra subprogrms

1. INTRODUCTION

Implementing matrix multiplication so that near-optimal performance is attained requires a thorough understanding of how the operation must be layered at the macro level in combination with careful engineering of high-performance kernels at the micro level. This paper primarily addresses the macro issues, namely how to exploit a high-performance "inner-kernel", more so than the the micro issues related to the design and engineering of that "inner-kernel".

In [Gunnels et al. 2001] a layered approach to the implementation of matrix multiplication was reported. The approach was shown to optimally amortize the required movement of data between two adjacent memory layers of an architecture with a complex multi-level memory. Like other work in the area [Agarwal et al. 1994; Whaley et al. 2001], that paper ([Gunnels et al. 2001]) casts computation in terms of an "inner-kernel" that computes $C := \tilde{A}B + C$ for some $m_c \times k_c$ matrix $\tilde{A}$ that is stored contiguously in some packed format and fits in cache memory. Unfortunately, the model for the memory hierarchy that was used is unrealistic in

Authors' addresses: Kazushige Goto, Texas Advanced Computing Center, The University of Texas at Austin, Austin, TX 78712, kgoto@tacc.utexas.edu. Robert A. van de Geijn, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, rvdg@cs.utexas.edu.

高性能矩阵乘法剖析

KAZUSHIGE GOTO 德克萨斯大学奥斯汀分校和 ROBERT A. VAN DE GEIJN 德克萨斯大学奥斯汀分校

我们提出了矩阵矩阵乘法的高性能实现基础的基本原理，该矩阵乘法是广泛使用的 GotoBLAS 库的一部分。通过连续完善具有多级存储器的架构模型来证明设计决策的合理性。执行此作结果的简单但有效的算法。在广泛的架构选择上实施可以实现接近峰值的性能。

类别和主题描述符：G.4 [数学软件]： —效率

一般术语：算法;性能附加关键词和短语：线性代数、矩阵乘法、基本线性代数子项目

1. 介绍

实现矩阵乘法以获得接近最佳的性能需要彻底了解如何在宏观层面上分层运算，并结合在微观层面仔细设计高性能内核。本文主要解决宏观问题，即如何利用高性能的“内部内核”，而不是与该“内部内核”的设计和工程相关的微观问题。

在 [Gunnels 等人，2001 年] 中，报道了一种实现矩阵乘法的分层方法。该方法被证明可以优化在具有复杂多级内存的架构的两个相邻内存层之间所需的数据移动。与该地区的其他工作一样 [Agarwal 等人，1994 年;Whaley 等人，2001 年]，该论文（[Gunnels 等人，2001 年]）根据“内核”进行计算，该内核计算 C ： = ˜AB + C 对于某些 m× kmatrix ˜
以某种打包格式连续存储并适合高速缓存的 A。不幸的是，使用的内存层次结构模型在

作者地址：Kazushige Goto，德克萨斯大学奥斯汀分校德克萨斯高级计算中心，德克萨斯州奥斯汀，德克萨斯州 78712，kgoto@tacc.utexas.edu。Robert A. van de Geijn，德克萨斯大学奥斯汀分校计算机科学系，德克萨斯州奥斯汀 78712，rvdg@cs.utexas.edu。

at least two ways:

—It assumes that this inner-kernel computes with a matrix $\tilde{A}$ that resides in the level-1 (L1) cache.

—It ignores issues related to the Translation Look-aside Buffer (TLB).

The current paper expands upon a related technical report [Goto and van de Geijn 2002] which makes the observations that

—The ratio between the rate at which floating point operations (flops) can be performed by the floating point unit(s) and the rate at which floating point numbers can be streamed from the level-2 (L2) cache to registers is typically relatively small. This means that matrix $\tilde{A}$ can be streamed from the L2 cache.

—It is often the amount of data that can be addressed by the TLB that is the limiting factor for the size of $\tilde{A}$. (Similar TLB issues were discussed in [Strazdins 1998].)

In addition, we now observe that

—There are in fact six inner-kernels that should be considered for building blocks for high-performance matrix multiplication. One of these is argued to be inherently superior over the others. (In [Gunnels et al. 2001; Gunnels et al. 2005] three of these six kernels were identified.)

Careful consideration of all these observations underlie the implementation of the DGEMM Basic Linear Algebra Subprograms (BLAS) routine that is part of the widely used GotoBLAS library [Goto 2005].

In Fig. 1 we preview the effectiveness of the techniques. In those graphs we report performance of our implementation as well as vendor implementations (Intel's MKL (8.1.1) and IBM's ESSL (4.2.0) libraries) and ATLAS [Whaley and Dongarra 1998] (3.7.11) on the Intel Pentium4 Prescott processor, the IBM Power 5 processor, and the Intel Itanium2 processor[1] It should be noted that the vendor implementations have adopted techniques very similar to those described in this paper. It is important not to judge the performance of matrix-matrix multiplication in isolation. It is typically a building block for other operations like the level-3 BLAS (matrix-matrix operations) [Dongarra et al. 1990; Kågström et al. 1998] and LAPACK [Anderson et al. 1999]. How the techniques described in this paper impact the implementation of level-3 BLAS is discussed in [Goto and van de Geijn 2006].

This paper attempts to describe the issues at a high level so as to make it accessible to a broad audience. Low level issues are introduced only as needed. In Section 2 we introduce notation that is used throughout the remainder of the paper. In Section 3 a layered approach to implementing matrix multiplication is introduced. High-performance implementation of the inner-kernels is discussed in Section 4. Practical algorithms for the most commonly encountered cases of matrix multiplication are given in Section 5. In Section 6 we give further details that are used in practice to determine parameters that must be tuned in order to optimize performance. Performance results attained with highly tuned implementations on

[1]All libraries that were timed use assembly-coded inner-kernels (including ATLAS). Compiler options `-fomit-frame-pointer -O3 -funroll-all-loops` were used.

至少两种方式：

—它假设该内部内核使用驻留在一级 （L1） 缓存中的矩阵 A 进行计算。

– 它忽略与转换后备缓冲区 （TLB） 相关的问题。

本文扩展了相关技术报告 [Goto 和 van de Geijn，2002 年]，该报告指出：

—浮点单元执行浮点运算（flops）的速率与浮点数从 2 级（L2）缓存流式传输到寄存器的速率之间的比率通常相对较小。这意味着矩阵 A 可以从 L2 缓存流式传输。—TLB 可以寻址的数据量通常是 A 大小的限制因素。（类似的 TLB 问题在 [Strazdins 1998] 中进行了讨论。

此外，我们现在观察到

—实际上，有六个内核应该被考虑用于高性能矩阵乘法的构建块。其中一个被认为本质上优于其他。（在 [Gunnels 等人，2001 年;Gunnels 等人，2005 年]，这六个内核中的三个被鉴定出来。

仔细考虑所有这些观察结果是实现 dgemm 基本线性代数子程序 （BLAS） 例程的基础，该例程是广泛使用的 GotoBLAS 库 [Goto 2005] 的一部分。

在图 1 中，我们预览了这些技术的有效性。在这些图表中，我们报告了我们的实现以及供应商实现（Intel 的 MKL （8.1.1） 和 IBM 的 ESSL （4.2.0） 库）和 ATLAS [Whaley 和 Dongarra 1998]（3.7.11） 在 Intel Pentium4 Prescott 处理器、IBM Power 5 处理器和 Intel Itanium2 处理器上的性能应该注意的是，供应商实现采用了与本文中描述的非常相似的技术。重要的是不要孤立地判断矩阵–矩阵乘法的性能。它通常是其他作的构建块，例如 3 级 BLAS（矩阵–矩阵作）[Dongarra 等人，1990 年;K˚agstr öm 等人，1998 年]和 LAPACK [Anderson 等人，1999 年]。本文中描述的技术如何影响 3 级 BLAS 的实施，在 [Goto 和 van de Geijn 2006] 中进行了讨论。

本文试图从高层次上描述这些问题，以便让广大读者能够理解。仅在需要时引入低级别问题。在第 2 节中，我们介绍了本文其余部分使用的符号。在第 3 节中，介绍了实现矩阵乘法的分层方法。第 4 节讨论了内部内核的高性能实现。第 5 节给出了最常见的矩阵乘法情况的实用算法。在第 6 节中，我们提供了更多细节，这些细节在实践中用于确定必须调整的参数以优化性能。通过高度调整的实施获得的性能结果

[1] 所有定时的库都使用汇编编码的内部内核（包括 ATLAS）。使用编译器选项 –fomit–frame–pointer –O3 –funroll–all–loops。

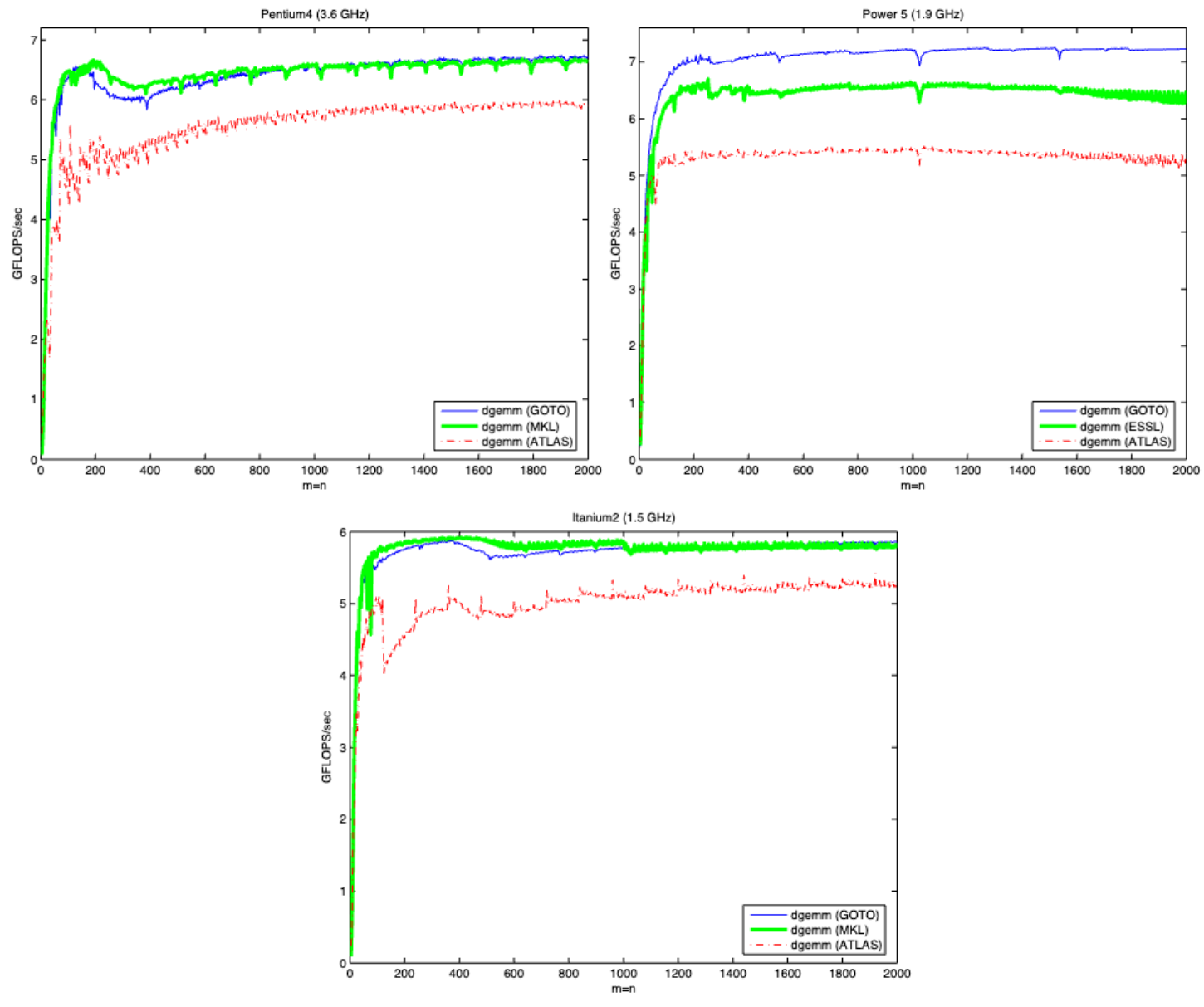


Fig. 1. Comparison of the matrix-matrix multiplication described in this paper with various other implementations. See Section 7 for details regarding the different architectures.

various architectures are given in Section 7. Concluding comments can be found in the final section.

2. NOTATION

The partitioning of matrices is fundamental to the description of matrix multiplication algorithms. Given an $m \times n$ matrix X, we will only consider partitionings of X into blocks of columns and blocks of rows:

$$X = \left(X_0 \middle| X_1 \middle| \cdots \middle| X_{N-1} \right) = \left(\begin{array}{c} \check{X}_0 \\ \hline \check{X}_1 \\ \hline \vdots \\ \hline \check{X}_{M-1} \end{array} \right),$$

where X_j has n_b columns and $\check{X}_i$ has m_b rows (except for X_{N-1} and $\check{X}_{M-1}$, which may have fewer columns and rows, respectively).

The implementations of matrix multiplication will be composed from multiplications with submatrices. We have given these computations special names, as tabulated in Figs. 2 and 3. We note that these special shapes are very frequently

高性能矩阵乘法剖析

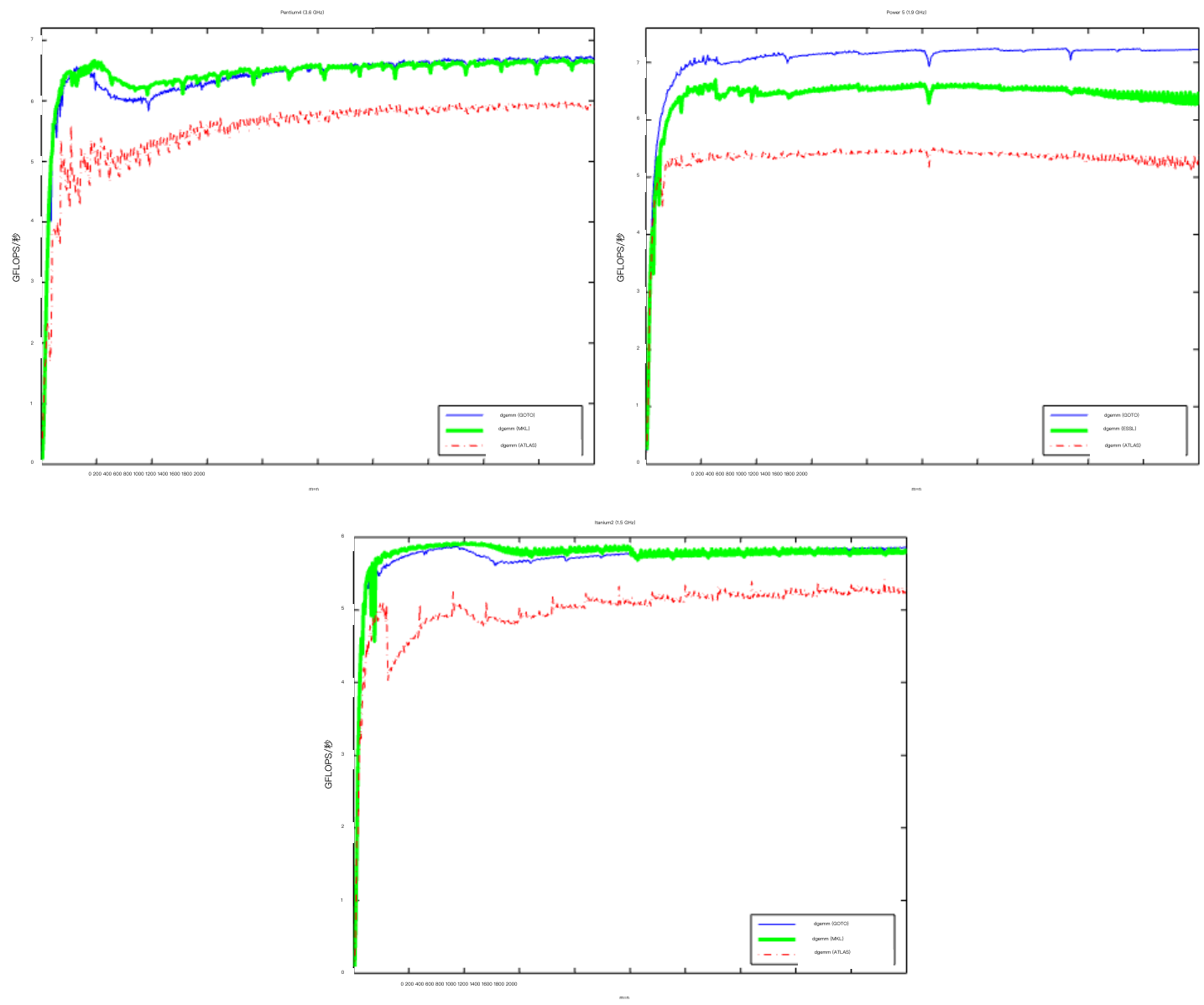


图 1.本文中描述的矩阵–矩阵乘法与其他各种实现的比较。有关不同架构的详细信息，请参见第 7 节。

第 7 节给出了各种架构。结论性评论可在最后一节中找到。

2. 表示法

矩阵的划分是矩阵乘法算法描述的基础。给定一个 m × n 个矩阵 X，我们只考虑将 X 划分为列块和行块：

$$X = \begin{pmatrix} X & X & \cdots & X \end{pmatrix} = \begin{pmatrix} \check{X} \\ \check{X} \\ \vdots \\ \check{X} \end{pmatrix},$$

其中 X 有 n 列和 $\check{X}$ 有 mrows（除了 X 和 $\check{X}$，它们可能分别具有较少的列和行）。
矩阵乘法的实现将由子矩阵的乘法组成。我们给这些计算起了特殊名称，如图 2 和图 3 所示。我们注意到这些特殊形状非常频繁

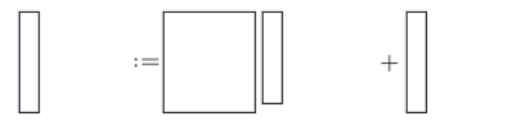
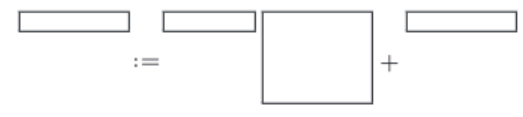
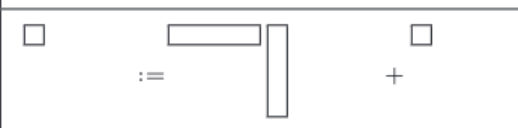
m	n	k	Illustration	Label
large	large	large		GEMM
large	large	small		GEPP
large	small	large		GEMP
small	large	large		GEPM
small	large	small		GEBP
large	small	small		GEPB
small	small	large		GEPDOT
small	small	small		GEBB

Fig. 2. Special shapes of GEMM $C := AB + C$. Here C, A, and B are $m \times n$, $m \times k$, and $k \times n$ matrices, respectively.

Letter	Shape	Description
M	Matrix	Both dimensions are large or unknown.
P	Panel	One of the dimensions is small.
B	Block	Both dimensions are small.

Fig. 3. The labels in Fig. 2 have the form GEXY where the letters chosen for X and Y indicate the shapes of matrices A and B, respectively, according to the above table. The exception to this convention is the GEPDOT operation, which is a generalization of the dot product.

encountered as part of algorithms for other linear algebra operations. For example, computation of various operations supported by LAPACK is cast mostly in terms of GEPP, GEMP, and GEPM. Even given a single dense linear algebra operation, multiple algorithms often exist where each of the algorithms casts the computation in terms of these different cases of GEMM multiplication [Bientinesi et al.].

3. A LAYERED APPROACH TO GEMM

In Fig. 4 we show how GEMM can be decomposed systematically into the special cases that were tabulated in Fig. 2. The general GEMM can be decomposed into multiple calls to GEPP, GEMP, or GEPM. These themselves can be decomposed into multiple calls to GEBP, GEPB, or GEPDOT kernels. The idea now is that if these three lowest level kernels attain high performance, then so will the other cases of GEMM. In Fig. 5 we relate the path through Fig. 4 that always take the top branch

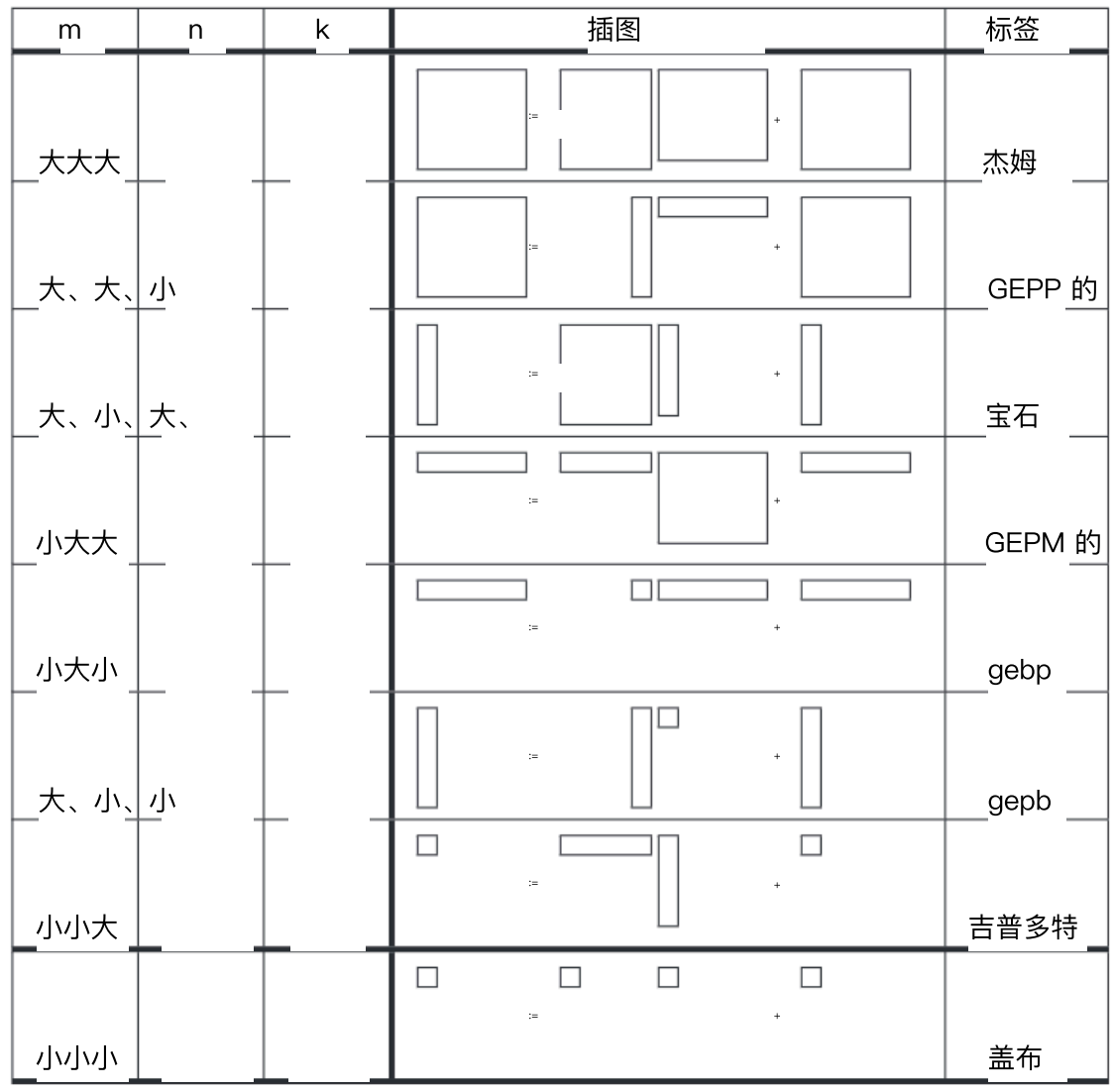


图 2.宝石 C 的特殊形状：= AB + C。这里 C、A 和 B 分别是 n 个矩阵× 个矩阵、 m × k 和 k × n 个矩阵。

字母形状说明	
m 矩阵	两个维度都很大或未知。
p 面板	其中一个尺寸很小。
b	块 两个尺寸都很小。

图 3.图 2 中的标签形式为 gexy，其中 x 和 y 选择的字母分别表示矩阵 A 和 B 的形状，如上表所示。此约定的例外是 gepdot 运算，它是点积的推广。

作为其他线性代数运算算法的一部分遇到。例如，LAPACK 支持的各种运算的计算主要以 gepp、gemp 和 gepm 的形式进行铸造。即使给定单个密集线性代数运算，也经常存在多种算法，其中每种算法都根据这些不同的 gemm 乘法情况进行计算 [Bientinesi 等人]。

3. GEMM 的分层方法

在图 4 中，我们展示了如何将 gemm 系统地分解为图 2 中列出的特殊情况。通用 gemm 可以分解为对 gepp、gemp 或 gepm 的多次调用。这些本身可以分解为对 gebp、gepb 或 gepdot 内核的多次调用。现在的想法是，如果这三个最低级别的内核达到高性能，那么 gemm 的其他情况也会如此。在图 5 中，我们关联了始终采用顶部分支的图 4 路径

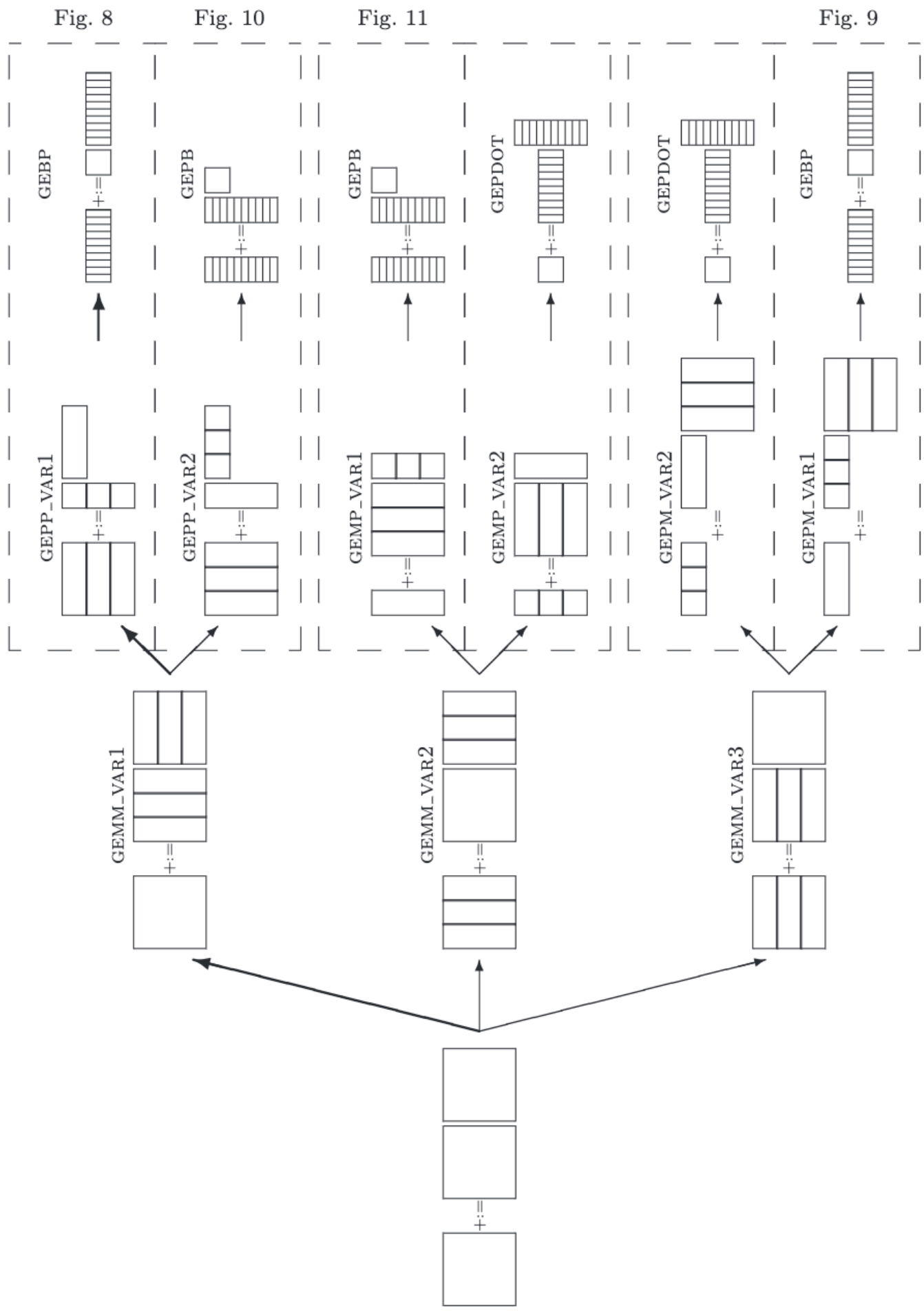


Fig. 4. Layered approach to implementing GEMM.

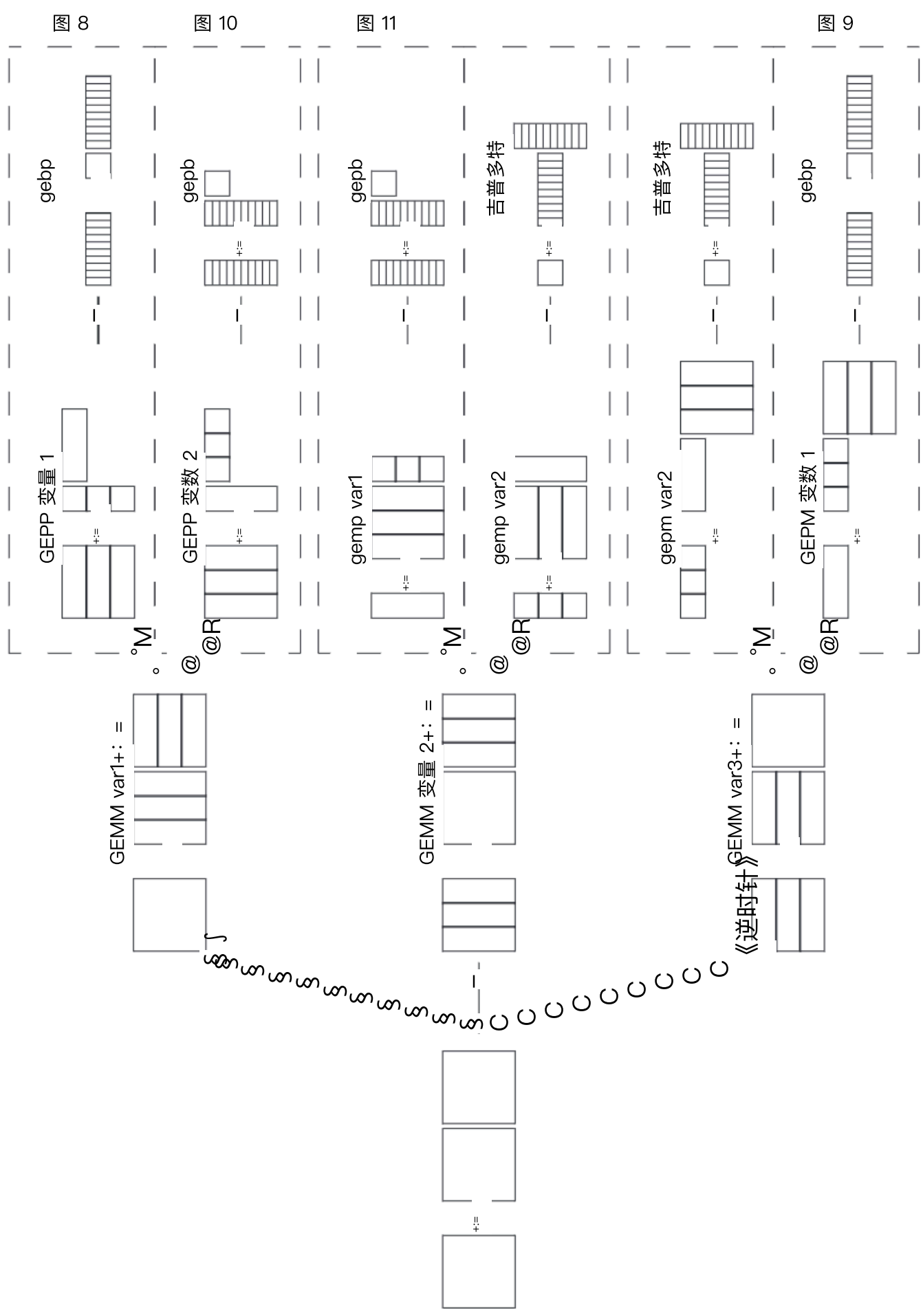


图 4.实现 gemm 的分层方法。

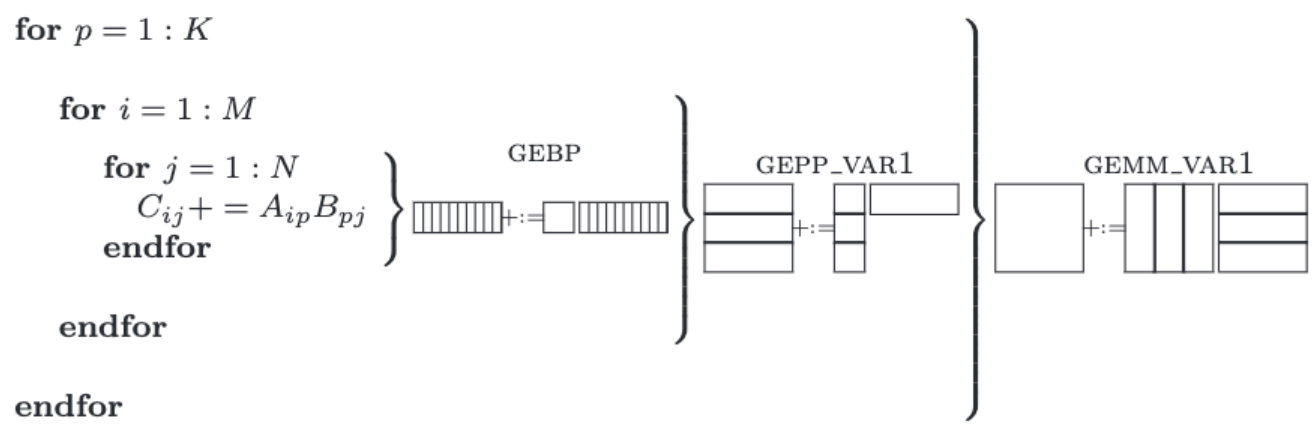


Fig. 5. The algorithm that corresponds to the path through Fig. 4 that always takes the top branch expressed as a triple-nested loop.

to a triple-nested loop. In that figure C, A, and B are partitioned into submatrices as

$$C = \begin{pmatrix} C_{11} & \cdots & C_{1N} \\ \vdots & & \vdots \\ C_{M1} & \cdots & C_{MN} \end{pmatrix}, A = \begin{pmatrix} A_{11} & \cdots & A_{1K} \\ \vdots & & \vdots \\ A_{M1} & \cdots & A_{MK} \end{pmatrix}, \text{ and } C = \begin{pmatrix} C_{11} & \cdots & C_{1N} \\ \vdots & & \vdots \\ C_{M1} & \cdots & C_{MN} \end{pmatrix},$$

where $C_{ij} \in \mathbb{R}^{m_c \times n_r}$, $A_{ip} \in \mathbb{R}^{m_c \times k_c}$, and $B_{pj} \in \mathbb{R}^{k_c \times n_r}$. The block sizes m_c, n_r, and k_c will be discussed in detail later in the paper.

A theory that supports an optimality claim regarding the general approach mentioned in this section can be found in [Gunnels et al. 2001]. In particular, that paper supports the observation that computation should be cast in terms of the decision tree given in Fig. 4 if data movement between memory layers is to be optimally amortized. However, the present paper is self-contained since we show that the approach amortizes such overhead well and thus optimality is not crucial to our discussion.

4. HIGH-PERFORMANCE GEBP, GEPB, AND GEPDOT

We now discuss techniques for the high-performance implementation of GEBP, GEPB, and GEPDOT. We do so by first analyzing the cost of moving data between memory layers with an admittedly naive model of the memory hierarchy. In Section 4.2 we add more practical details to the model. This then sets the stage for algorithms for GEPP, GEMP, and GEPM in Section 5.

4.1 Basics

In Fig. 6(left) we depict a very simple model of a multi-level memory. One layer of cache memory is inserted between the Random-Access Memory (RAM) and the registers. The top-level issues related to the high-performance implementation of GEBP, GEPB, and GEPDOT can be described using this simplified architecture.

Let us first concentrate on GEBP with $A \in \mathbb{R}^{m_c \times k_c}$, $B \in \mathbb{R}^{k_c \times n}$, and $C \in \mathbb{R}^{m_c \times n}$. Partition

$$B = \left(B_0 \middle| B_1 \middle| \cdots \middle| B_{N-1} \right) \quad \text{and} \quad C = \left(C_0 \middle| C_1 \middle| \cdots \middle| C_{N-1} \right)$$

and assume that

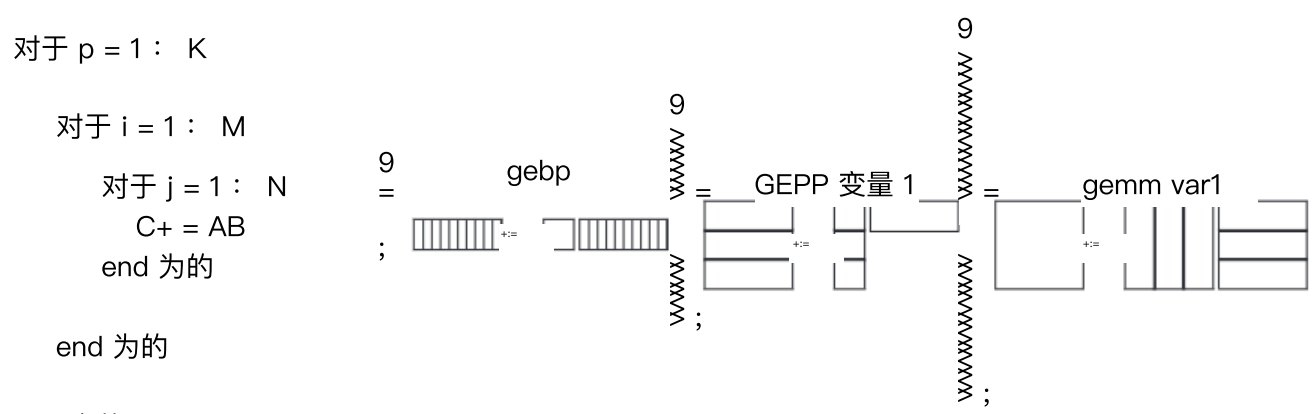


图 5.对应于通过图 4 的路径的算法，该路径始终采用表示为三重嵌套循环的顶部分支。

到三重嵌套循环。在该图中，C、A 和 B 被划分为子矩阵，如下所示

$$C = \begin{pmatrix} C_{0,0} & \cdots & C_{0,N-1} \\ \vdots & & \vdots \\ C_{M-1,0} & \cdots & C_{M-1,N-1} \end{pmatrix}, \quad A = \begin{pmatrix} A_{0,0} & \cdots & A_{0,K-1} \\ \vdots & & \vdots \\ A_{M-1,0} & \cdots & A_{M-1,K-1} \end{pmatrix}, \quad C = \begin{pmatrix} C_0 & \cdots & C_{N-1} \end{pmatrix}$$

其中 C∈ R、A∈ R 和 B∈ R。块大小 m、n 和 k 将在本文后面详细讨论。

支持本节中提到的一般方法的最优性主张的理论可以在 [Gunnels 等人，2001 年] 中找到。特别是，该论文支持这样的观察结果，即如果要对内存层之间的数据移动进行最佳摊销，则应根据图 4 中给出的决策树进行计算。然而，本文是独立的，因为我们表明该方法可以很好地摊销这种开销，因此最优性对我们的讨论并不重要。

4. 高性能 GEBP、GEPB 和 GEPDOT

我们现在讨论 gebp、gepb 和 gepdot 的高性能实现技术。为此，我们首先使用公认的朴素的内存层次结构模型分析在内存层之间移动数据的成本。在第 4.2 节中，我们为模型添加了更多实际细节。这为第 5 节中的 gepp、gemp 和 gepm 算法奠定了基础。

4.1 基础知识

在图 6（左）中，我们描绘了一个非常简单的多级存储器模型。在随机存取存储器（RAM） 和寄存器之间插入一层高速缓存。可以使用这种简化的架构来描述与 gebp、gepb 和 gepdot 的高性能实现相关的顶级问题。

让我们首先专注于 A ∈ R、B ∈ R 和 C ∈ R 的 gebp。分区

$$乙 = \begin{pmatrix} B_0 & \cdots & B_{N-1} \end{pmatrix} \quad 和 \ C = \begin{pmatrix} C_0 & \cdots & C_{N-1} \end{pmatrix}$$

并假设

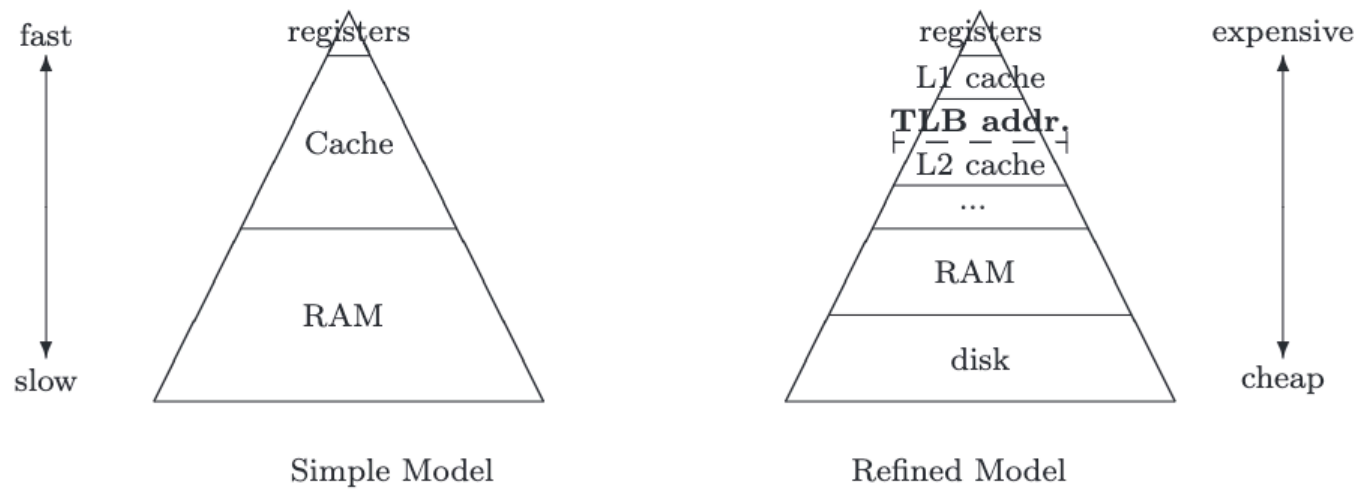
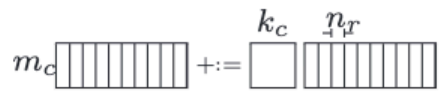


Fig. 6. The hierarchical memories viewed as a pyramid.

Algorithm: $C := \text{GEBP}(A, B, C)$



Load A into cache	($m_c k_c$ memops)
for $j = 0, \ldots, N-1$	
Load B_j into cache	($k_c n_r$ memops)
Load C_j into cache	($m_c n_r$ memops)
+:=	($C_j := AB_j + C_j$)
Store C_j into memory	($m_c n_r$ memops)
endfor	

Fig. 7. Basic implementation of GEBP.

Assumption (a). The dimensions m_c, k_c are small enough so that A and n_r columns from each of B and C (B_j and C_j, respectively) together fit in the cache.

Assumption (b). If A, C_j, and B_j are in the cache then $C_j := AB_j + C_j$ can be computed at the peak rate of the CPU.

Assumption (c). If A is in the cache it remains there until no longer needed.

Under these assumptions, the approach to GEBP in Figure 7 amortizes the cost of moving data between the main memory and the cache as follows. The total cost of updating C is $m_c k_c + (2m_c + k_c)n$ memops for $2m_c k_c n$ flops. Then the ratio between computation and data movement is

$$\frac{2m_c k_c n}{m_c k_c + (2m_c + k_c)n} \frac{\text{flops}}{\text{memops}} \approx \frac{2m_c k_c n}{(2m_c + k_c)n} \frac{\text{flops}}{\text{memops}} \quad \text{when } k_c \ll n. \tag{1}$$

Thus

$$\frac{2m_c k_c}{(2m_c + k_c)} \tag{2}$$

should be maximized under the constraint that $m_c k_c$ floating point numbers fill most of the cache, and the constraints in Assumptions (a)–(c). In practice there are other issues that influence the choice of k_c, as we will see in Section 6.3. However, the bottom line is that under the simplified assumptions A should occupy as much

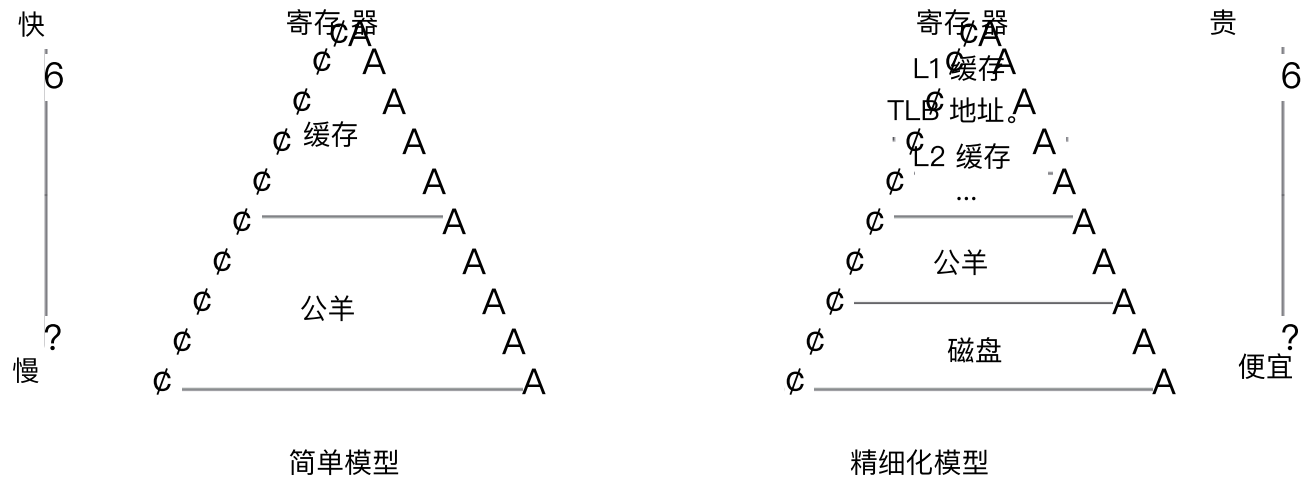


图 6.层次记忆被视为金字塔。

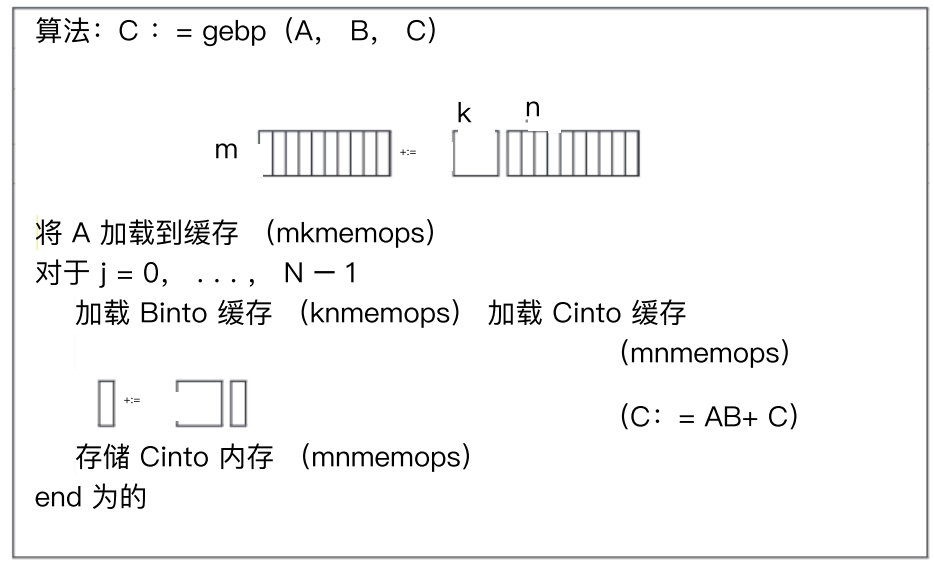


图 7.gebp 的基本实现。

假设（a）。维度 m， kas 足够小，以便 B 和 C 中的 A 和 n 列（分别为 Band C）一起放入缓存中。

假设（b）。如果缓存中的 A、C 和 Bare，则 C： = AB+ C 可以以 CPU 的峰值速率计算。

假设（c）。如果 A 在缓存中，它将一直保留在那里，直到不再需要为止。

在这些假设下，图 7 中的 gebp 方法分摊了在主内存和缓存之间移动数据的成本，如下所示。更新 C 的总成本为 mk+ （2m+ k）n 备忘录，用于 2mkn 翻牌。那么计算和数据移动之间的比率是

$$\frac{\text{2 毫克}}{\text{MK+（2M+K）N}} \frac{\text{失败}}{\text{备忘录}} \approx \frac{\text{2 毫克}}{\text{(2m+ k）n}} \frac{\text{失败}}{\text{备忘录}} \quad \text{当 k¿ n.} \quad (1)$$

因此

$$\frac{2mk}{(2m+ k)} \qquad (2)$$

应该在 mk 浮点数填充大部分缓存的约束和假设 （a）一（c） 中的约束下最大化。在实践中，还有其他问题会影响 k 的选择，我们将在第 6.3 节中看到。然而，底线是，在简化的假设下，A 应该占据同样多的

of the cache as possible and should be roughly square[2], while leaving room in the cache for at least B_j and C_j. If $m_c = k_c \approx n/100$ then even if memops are 10 times slower than flops, the memops add only about 10% overhead to the computation.

The related operations GEPB and GEPDOT can be analyzed similarly by keeping in mind the following pictures:

4.2 Refinements

In discussing practical considerations we will again focus on the high-performance implementation of GEBP. Throughout the remainder of the paper, we will assume that matrices are stored in column-major order.

4.2.1 *Choosing the cache layer.* A more accurate depiction of the memory hierarchy can be found in Fig. 6(right). This picture recognizes that there are typically multiple levels of cache memory.

The first question is in which layer of cache the $m_c \times k_c$ matrix A should reside. Equation (2) tells us that (under Assumptions (a)–(c)) the larger $m_c \times n_c$, the better the cost of moving data between RAM and the cache is amortized over computation. This suggests loading matrix A in the cache layer that is farthest from the registers (can hold the most data) subject to the constraint that Assumptions (a)–(c) are (roughly) met.

The L1 cache inherently has the property that if it were used for storing A, B_j and C_j, then Assumptions (a)–(c) are met. However, the L1 cache tends to be very small. Can A be stored in the $L2$ cache instead, allowing m_c and k_c to be much larger? Let R_{comp} and R_{load} equal the rate at which the CPU can perform floating point operations and the rate at which floating point number can be streamed from the L2 cache to the registers, respectively. Assume A resides in the L2 cache and B_j and C_j reside in the L1 cache. Assume further that there is "sufficient bandwidth" between the L1 cache and the registers, so that loading elements of B_j and C_j into the registers is not an issue. Computing $AB_j + C_j$ requires $2m_c k_c n_r$ flops and $m_c k_c$ elements of A to be loaded from the L2 cache to registers. To overlap the loading of elements of A into the registers with computation $2n_r/R_{\text{comp}} \geq 1/R_{\text{load}}$ must hold, or

$$n_r \geq \frac{R_{\text{comp}}}{2R_{\text{load}}}. \tag{3}$$

4.2.2 *TLB considerations.* A second architectural consideration relates to the page management system. For our discussion it suffices to consider that a typical modern architecture uses virtual memory so that the size of usable memory is not constrained by the size of the physical memory: Memory is partitioned into pages

[2]Note that optimizing the similar problem $m_c k_c/(2m_c + 2k_c)$ under the constraint that $m_c k_c \leq K$ is the problem of maximizing the area of a rectangle while minimizing the perimeter, the solution of which is $m_c = k_c$. We don't give an exact solution to the stated problem since there are practical issues that also influence m_c and k_c.

缓存的尽可能大致呈正方形，同时在缓存中至少为波段 C 留出空间。如果 m= k≈ n/100，那么即使备忘录比翻牌慢 10 倍，备忘录也只会给计算增加大约 10% 的开销。

通过记住以下图片，可以类似地分析相关的作 gepb 和 gepdot：

4.2 细化

在讨论实际考虑时，我们将再次关注 gebp 的高性能实现。在本文的其余部分，我们将假设矩阵按列主顺序存储。

选择缓存层。对内存层次结构的更准确描述可以在图 6（右）中找到。此图识别出通常存在多个级别的高速缓存。

第一个问题是 m× kmatrix A 应该驻留在哪个缓存层。

公式 （2） 告诉我们，（在假设 （a）— （c） 下）m×n 越大，在 RAM 和缓存之间移动数据的成本就越好。这表明将矩阵 A 加载到距离寄存器最远的缓存层（可以容纳最多数据）中，受制于（大致）满足假设 （a）— （c） 的约束。

L1 缓存本质上具有这样的特性：如果它用于存储 A、B 和 C，则满足假设 （a）— （c）。但是，L1 缓存往往非常小。是否可以将 A 存储在 L2 缓存中，从而允许 mand k 大得多？让 Rand 分别等于 CPU 可以执行浮点运算的速率和浮点数可以从 L2 缓存流式传输到寄存器的速率。假设 A 驻留在 L2 缓存中，B 和 Creside 驻留在 L1 缓存中。进一步假设 L1 缓存和寄存器之间有“足够的带宽”，因此将 Band Cin 的元素加载到寄存器中不是问题。计算 AB+ C 需要将 A 的 2mknflops 和 mk 元素从 L2 缓存加载到寄存器。要将 A 的元素加载到具有计算 2n/R 的寄存器中重叠≥1/R 必须保持，或

$$n \geqslant \frac{R}{2R \text{ 的}} . \tag{3}$$

TLB 注意事项。第二个架构考虑因素与页面管理系统有关。对于我们的讨论，只需考虑典型的现代架构使用虚拟内存，以便可用内存的大小不受物理内存大小的限制：内存被划分为多个页面

2 请注意，在 mk≤ K 的约束下优化类似问题 mk/（2m+2k） 是最大化矩形面积同时最小化周长的问题，其解为 m= k。我们没有给出上述问题的确切解决方案，因为有些实际问题也会影响 mand k。

of some (often fixed) prescribed size. A table, referred to as the *page table* maps virtual addresses to physical addresses and keeps track of whether a page is in memory or on disk. The problem is that this table itself is stored in memory, which adds additional memory access costs to perform virtual to physical translations. To overcome this, a smaller table, the Translation Look-aside Buffer (TLB), that stores information about the most recently used pages, is kept. Whenever a virtual address is found in the TLB, the translation is fast. Whenever it is not found (a TLB miss occurs), the page table is consulted and the resulting entry is moved from the page table to the TLB. In other words, the TLB is a cache for the page table. More recently, a level 2 TLB has been introduced into some architectures for reasons similar to those that motivated the introduction of an L2 cache.

The most significant difference between a cache miss and a TLB miss is that a cache miss does not necessarily stall the CPU. A small number of cache misses can be tolerated by using algorithmic prefetching techniques as long as the data can be read fast enough from the memory where it does exist and arrives at the CPU by the time it is needed for computation. A TLB miss, by contrast, causes the CPU to stall until the TLB has been updated with the new address. In other words, prefetching can mask a cache miss but not a TLB miss.

The existence of the TLB means that additional assumptions must be met:

Assumption (d). The dimensions m_c, k_c are small enough so that A, n_r columns from B (B_j) and n_r column from C (C_j) are simultaneously addressable by the TLB so that during the computation $C_j := AB_j + C_j$ no TLB misses occur.

Assumption (e). If A is addressed by the TLB, it remains so until no longer needed.

4.2.3 *Packing.* The fundamental problem now is that A is typically a submatrix of a larger matrix, and therefore is not contiguous in memory. This in turn means that addressing it requires many more than the minimal number of TLB entries. The solution is to pack A in a contiguous work array, $\tilde{A}$. Parameters m_c and k_c are then chosen so that $\tilde{A}$, B_j, and C_j all fit in the L2 cache *and* are addressable by the TLB.

Case 1: The TLB is the limiting factor. Let us assume that there are T TLB entries available, and let $T_{\tilde{A}}$, T_{B_j}, and T_{C_j} equal the number of TLB entries devoted to $\tilde{A}$, B_j, and C_j, respectively. Then

$$T_{\tilde{A}} + 2(T_{B_j} + T_{C_j}) \leq T.$$

The reason for the factor two is that when the next blocks of columns B_{j+1} and C_{j+1} are first addressed, the TLB entries that address them should replace those that were used for B_j and C_j. However, upon completion of $C_j := \tilde{A}B_j + C_j$ some TLB entries related to $\tilde{A}$ will be the least recently used, and will likely be replaced by those that address B_{j+1} and C_{j+1}. The factor two allows entries related to B_j and C_j to coexist with those for $\tilde{A}$, B_{j+1} and C_{j+1} and by the time B_{j+2} and C_{j+2} are first addressed, it will be the entries related to B_j and C_j that will be least recently used and therefore replaced.

The packing of A into $\tilde{A}$, if done carefully, needs not to create a substantial overhead beyond what is already exposed from the loading of A into the L2 cache

一些（通常是固定的）规定大小。表（称为页表）将虚拟地址映射到物理地址，并跟踪页面是在内存中还是在磁盘上。问题是这个表本身存储在内存中，这增加了执行虚拟到物理转换的额外内存访问成本。为了克服这个问题，保留了一个较小的表，即翻译后备缓冲区（TLB），用于存储有关最近使用的页面的信息。每当在 TLB 中找到虚拟地址时，转换速度就会很快。每当找不到它（发生 TLB 未命中）时，就会查阅页表，并将生成的条目从页表移动到 TLB。换句话说，TLB 是页表的缓存。最近，一些架构中引入了 2 级 TLB，其原因与引入 L2 缓存的原因类似。

缓存未命中和 TLB 未命中之间最显着的区别是，缓存未命中不一定会使 CPU 停滞。使用算法预取技术可以容忍少量缓存未命中，只要数据能够足够快地从内存中读取，并在计算需要时到达 CPU。相比之下，TLB 未命中会导致 CPU 停滞，直到 TLB 更新为新地址。换句话说，预取可以屏蔽缓存未命中，但不能屏蔽 TLB 未命中。

TLB 的存在意味着必须满足其他假设：

假设（d）。维度 m， kare 足够小，以便 A、来自 B （B） 的 ncolumns 和来自 C （C） 的 ncolumn 可同时被 TLB 寻址，以便在计算过程中 C：= AB+ Cno 发生 TLB 未命中。

假设（e）。如果 A 由 TLB 寻址，则它一直保持这种状态，直到不再需要为止。

包装2.3现在的根本问题是 A 通常是较大矩阵的子矩阵，因此在内存中不连续。这反过来意味着解决它需要的不仅仅是最小数量的 TLB 条目。解决方案是将 A 打包到一个连续的工作数组 ˜A 中。然后选择参数 mand k，以便 ˜A、B 和 Call 适合 L2 缓存并可由 TLB 寻址。

情况 1：TLB 是限制因素。让我们假设有 T TLB 条目可用，并让 T 、 T 和 Te 分别等于 A、B 和 C 的 TLB 条目数。然后

$$T + 2（T+ T） \leqslant T。$$

因素二的原因是，当 Band Care 首次处理下一个列块时，处理这些列的 TLB 条目应替换用于 Band C 的条目。然而，在完成 C：= ˜AB+ C 一些与 A 相关的 TLB 条目将是最近使用最少的，并且可能会被那些涉及 C 级的条目所取代。因素二允许与 B 和 C 相关的条目与 A、频段 C 的条目共存，当 C 带首次被处理时，与 C 带相关的条目将是最近使用最少并因此被替换的条目。

如果的包装，A 不需要产生超出˜将 A 加载到 L2 缓存中已经暴露的开销之外的大量开销

and TLB. The reason is as follows: The packing can be arranged so that upon completion $\tilde{A}$ resides in the L2 cache and is addressed by the TLB, ready for subsequent computation. The cost of accessing A to make this happen need not be substantially greater than the cost of moving A into the L2 cache, which is what would have been necessary even if A were not packed.

Operation GEBP is executed in the context of GEPP or GEPM. In the former case, the matrix B is reused for many separate GEBP calls. This means it is typically worthwhile to copy B into a contiguous work array, $\tilde{B}$, as well so that $T_{\tilde{B}_j}$ is reduced when $C := \tilde{A}\tilde{B} + C$ is computed.

Case 2: The size of the L2 cache is the limiting factor. A similar argument can be made for this case. Since the limiting factor is more typically the amount of memory that the TLB can address (e.g., the TLB on a current generation Pentium4 can address about 256Kbytes while the L2 cache can hold 2Mbytes), we do not elaborate on the details.

4.2.4 *Accessing data contiguously.* In order to move data most efficiently to the registers, it is important to organize the computation so that, as much as practical, data that is consecutive in memory is used in consecutive operations. One way to accomplish this is to not just pack A into work array $\tilde{A}$, but to arrange it carefully. We comment on this in Section 6.

From here on in this paper, "Pack A into $\tilde{A}$" and "$C := \tilde{A}B + C$" will denote any packing that makes it possible to compute $C := AB + C$ while accessing the data consecutively, as much as needed. Similarly, "Pack B into $\tilde{B}$" will denote a copying of B into a contiguous format.

4.2.5 *Implementation of* GEPB *and* GEPDOT. Analyses of implementations of GEPB and GEPDOT can be similarly refined.

5. PRACTICAL ALGORITHMS

Having analyzed the approaches at a relatively coarse level of detail, we now discuss practical algorithms for all six options in Fig. 4 while exposing additional architectural considerations.

5.1 Implementing GEPP with GEBP

The observations in Sections 4.2.1–4.2.4 are now summarized for the implementations of GEPP in terms of GEBP in Fig. 8. The packing and computation are arranged to maximize the size of $\tilde{A}$: by packing B into $\tilde{B}$ in GEPP_VAR1, B_j typically requires only one TLB entry. A second TLB entry is needed to bring in B_{j+1}. The use of C_{aux} means that only one TLB entry is needed for that buffer, as well as up to n_r TLB entries for C_j (n_r if the leading dimension of C_j is large). Thus, $T_{\tilde{A}}$ is bounded by $T - (n_r + 3)$. The fact that C_j is not contiguous in memory is not much of a concern, since that data is not reused as part of the computation of the GEPP operation.

Once B and A have been copied into $\tilde{B}$ and $\tilde{A}$, respectively, the loop in GEBP_OPT1 can execute at almost the peak of the floating point unit.

—The packing of B is a memory-to-memory copy. Its cost is proportional to $k_c \times n$ and is amortized over $2m \times n \times k_c$ so that $O(m)$ computations will be performed

和 TLB。原因如下：打包可以安排成，完成后 A 驻留在 L2 缓存中，并由 TLB 寻址，为后续计算做好准备。访问 A 以实现这一点的成本不必大大高于将 A 移动到 L2 缓存的成本，即使 A 没有打包，这也是必要的。

作 gebp 在 gepp 或 gepm 的上下文中执行。在前一种情况下，矩阵 B 被重用于许多单独的 gebp 调用。这意味着通常值得将 B 复制到连续的工作数组 ˜B 中，以便在计算 C 时减少 T：= A B + C。

案例 2：L2 缓存的大小是限制因素。类似的论点可以
是为这个案子而做的。由于限制因素更典型的是 TLB 可以寻址的内存量（例如，当前一代 Pentium4 上的 TLB 可以寻址大约 256KB，而 L2 缓存可以容纳 2MB），因此我们不详细说明细节。

连续访问数据。为了最有效地将数据移动到寄存器，组织计算非常重要，以便尽可能多地将内存中的连续数据用于连续作。实现这一目标的一种方法是不仅将 A 打包到工作数组 ˜A 中，而且要仔细排列它。

我们在第 6 节中对此进行了评论。
从本文的此处开始，“将 A 打包成 A˜”和“C：= AB + C”将表示任何打包，可以在连续访问数据时根据需要计算 C：= AB + C。同样，“将 B 打包到 ˜B”将表示将 B 复制成连续格式。

gepb 和 gepdot 的实现。对 gepb 和 gepdot 实现的分析也可以类似地细化。

5. 实用算法

在相对粗略的细节层面上分析了这些方法之后，我们现在讨论了图 4 中所有六个选项的实用算法，同时揭示了其他架构考虑因素。

5.1 使用 gebp 实现 gepp

现在，在图 8 中总结了第 4.2.1–4.2.4 节中关于 gepp 实现的 gepp 的观察结果。打包和计算的安排是最大化 ˜A 的大小：通过将 B 打包到 gepp var1 中的 ˜B 中，B 通常只需要一个 TLB 条目。需要第二个 TLB 条目才能引入 B。使用 C 意味着该缓冲区只需要一个 TLB 条目，以及 C 最多需要 nTLB 条目（n 如果 Cis 的前导维度很大）。因此，T 以 T — （n+ 3）为界。C 在内存中不连续的事实并不是什么大问题，因为该数据不会作为 gepp 作计算的一部分重用。

一旦 B 和 A 分别复制到 B 和 A˜ 中，gebp opt1 中的循环几乎可以在浮点单元的峰值处执行。

– B 的打包是内存到内存的副本。它的成本与 k× n 成正比，并在 2m × n × k 上摊销，因此将执行 O（m） 计算

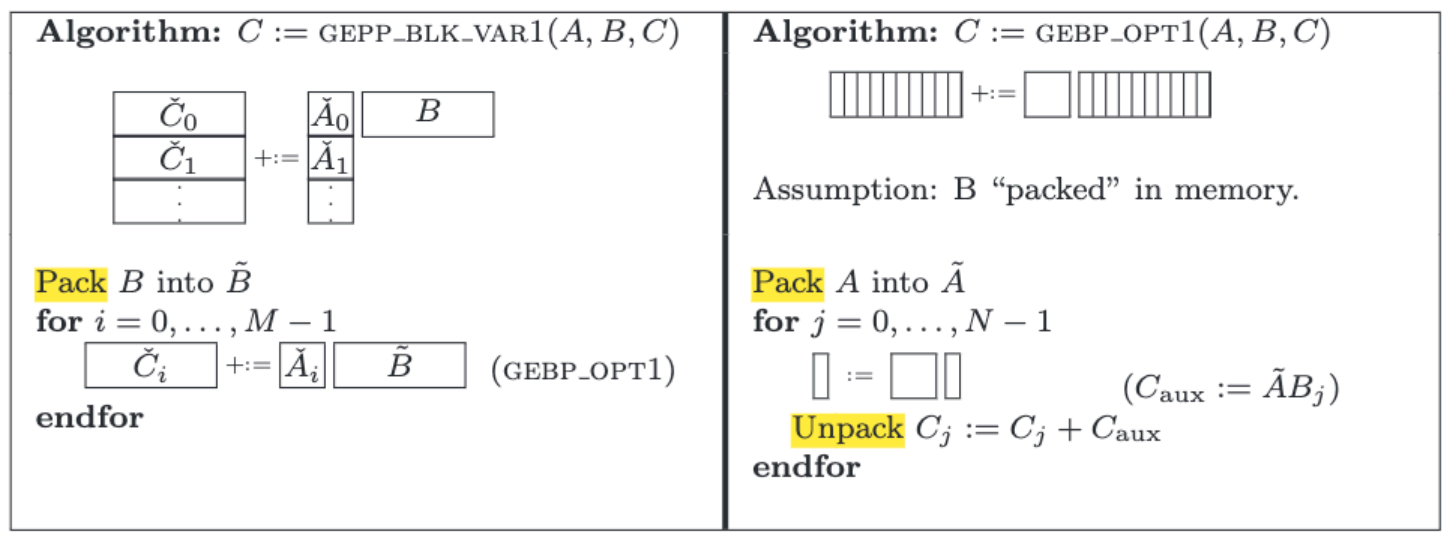


Fig. 8. Optimized implementation of GEPP (left) via calls to GEBP_OPT1 (right).

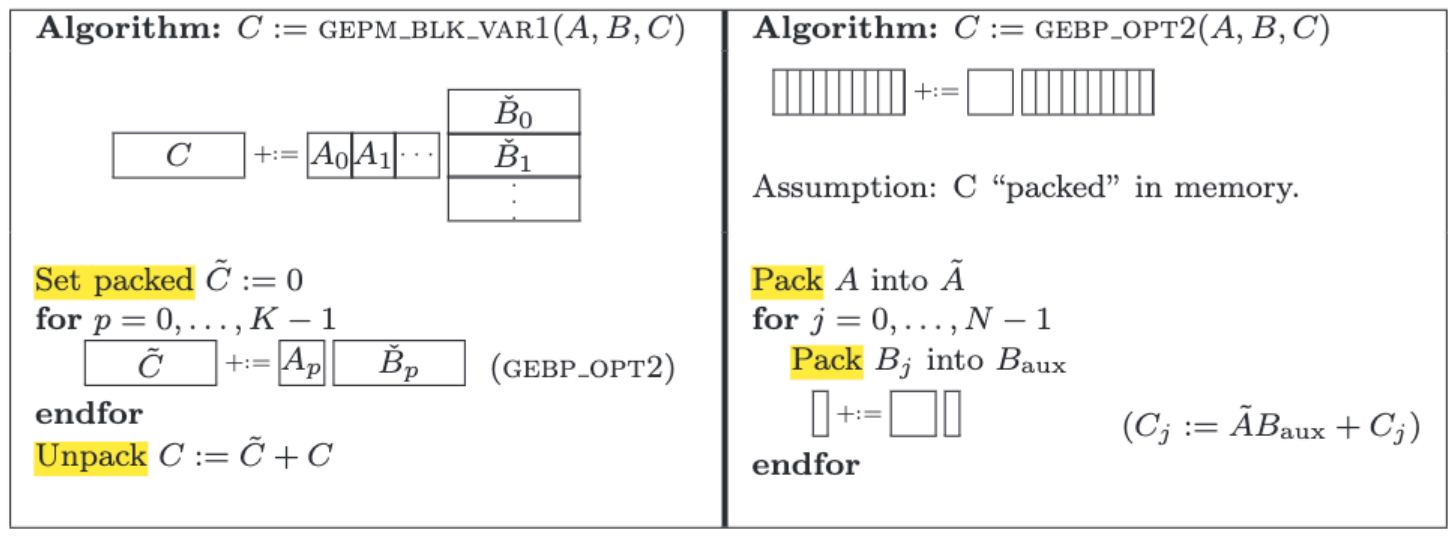


Fig. 9. Optimized implementation of GEPM (left) via calls to GEBP_OPT2 (right).

for every copied item. This packing operation disrupts the previous contents of the TLB.

—The packing of A to $\tilde{A}$ rearranges this data from memory to a buffer that will likely remain in the L2 cache and leaves the TLB loaded with useful entries, if carefully orchestrated. Its cost is proportional to $m_c \times k_c$ and is amortized over $2m_c \times k_c \times n$ computation so that $O(n)$ computations will be performed for every copied item. In practice, this copy is typically less expensive.

This approach is appropriate for GEMM if m and n are both large, and k is not too small.

5.2 Implementing GEPM with GEBP

In Fig. 9 a similar strategy is proposed for implementing GEPM in terms of GEBP. This time C is repeatedly updated so that it is worthwhile to accumulate $\tilde{C} = AB$ before adding the result to C. There is no reuse of $\check{B}_p$ and therefore it is not packed. Now at most n_r TLB entries are needed for B_j, and one each for B_{temp}, C_j and C_{j+1} so that again $T_{\tilde{A}}$ is bounded by $T - (n_r + 3)$.

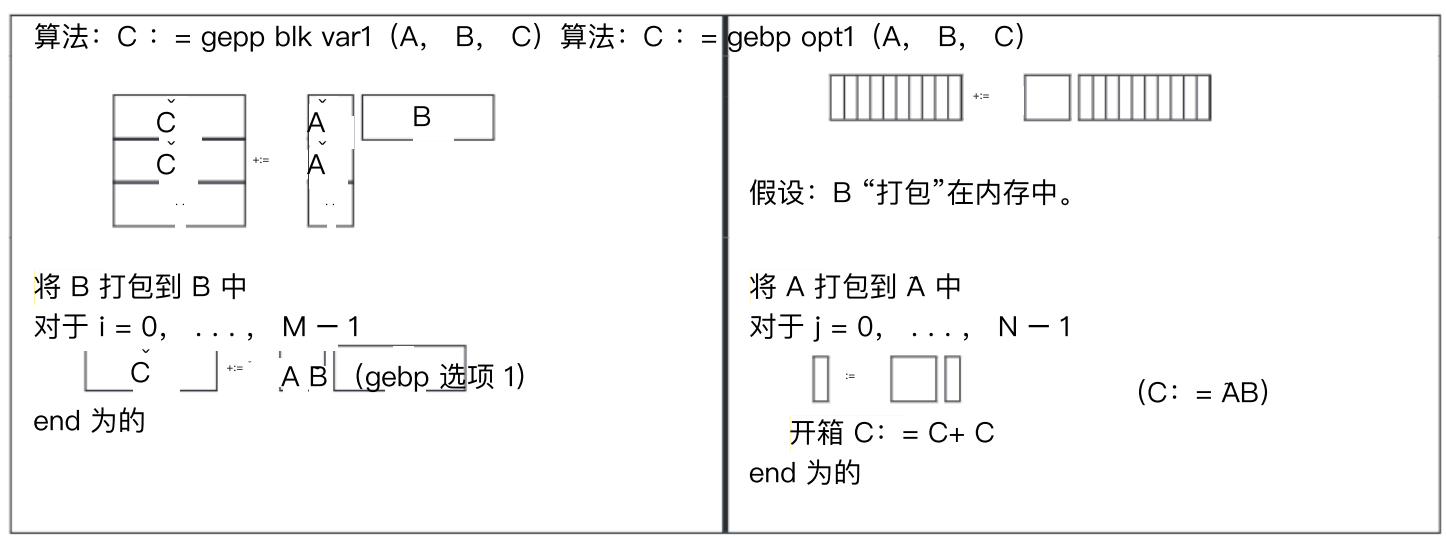


图 8.通过调用 gebp opt1（右）优化了 gepp（左）的实现。

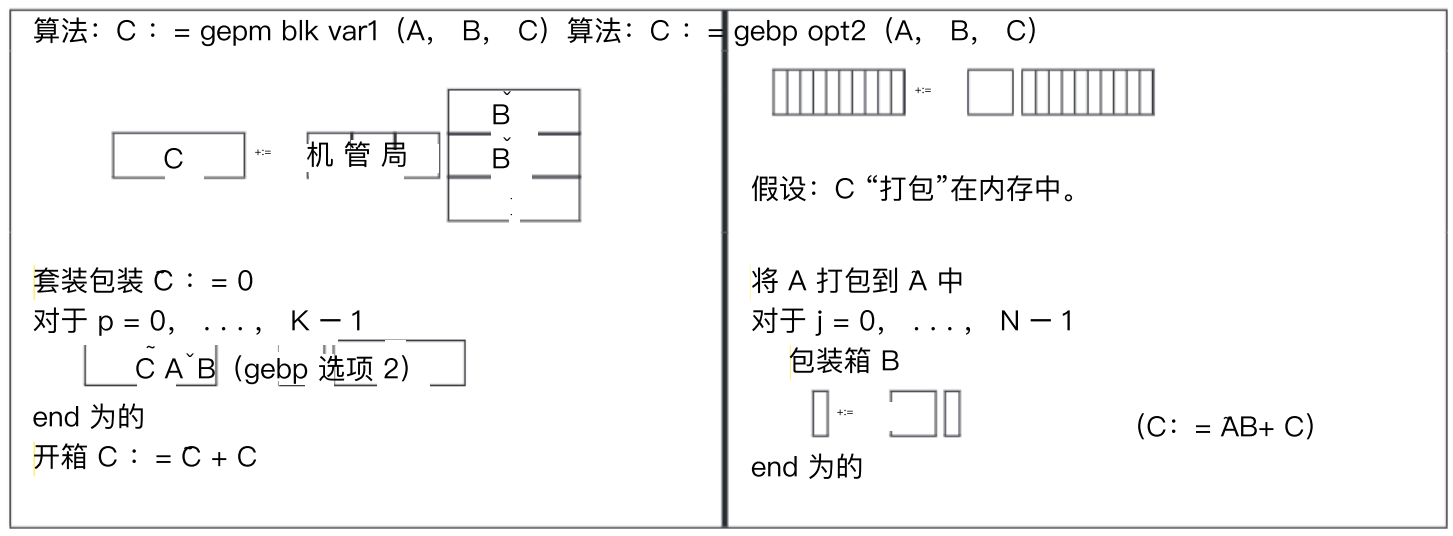


图 9.通过调用 gebp opt2（右）优化了 gepm（左）的实现。

对于每个复制的项目。此打包作会破坏 TLB 的先前内容。

—如果的包装心编排，A 会将此数据从内存重新排列到可能保留在 L2 缓存中的缓冲区，并让 TLB 加载有用的条目。它的成本与 m× kand 成正比×并且摊销为 2m k× n 次计算，因此将对每个复制的项目执行 O（n） 计算。实际上，此副本通常更便宜。

如果 m 和 n 都很大，并且 k 不太小，则此方法适用于 gemm。

5.2 使用 gebp 实现 gepm

在图 9 中，提出了一种类似的策略来实现 gebp。这一次 C 被反复更新，因此值得在将结果添加到 C 之前累积 C = AB。 Band 没有重复使用，因此它没有被打包。现在，B 最多需要一个 nTLB 条目，B、C 和 C 各需要一个条目，因此 T 再次以 T 一（n+ 3）为界。

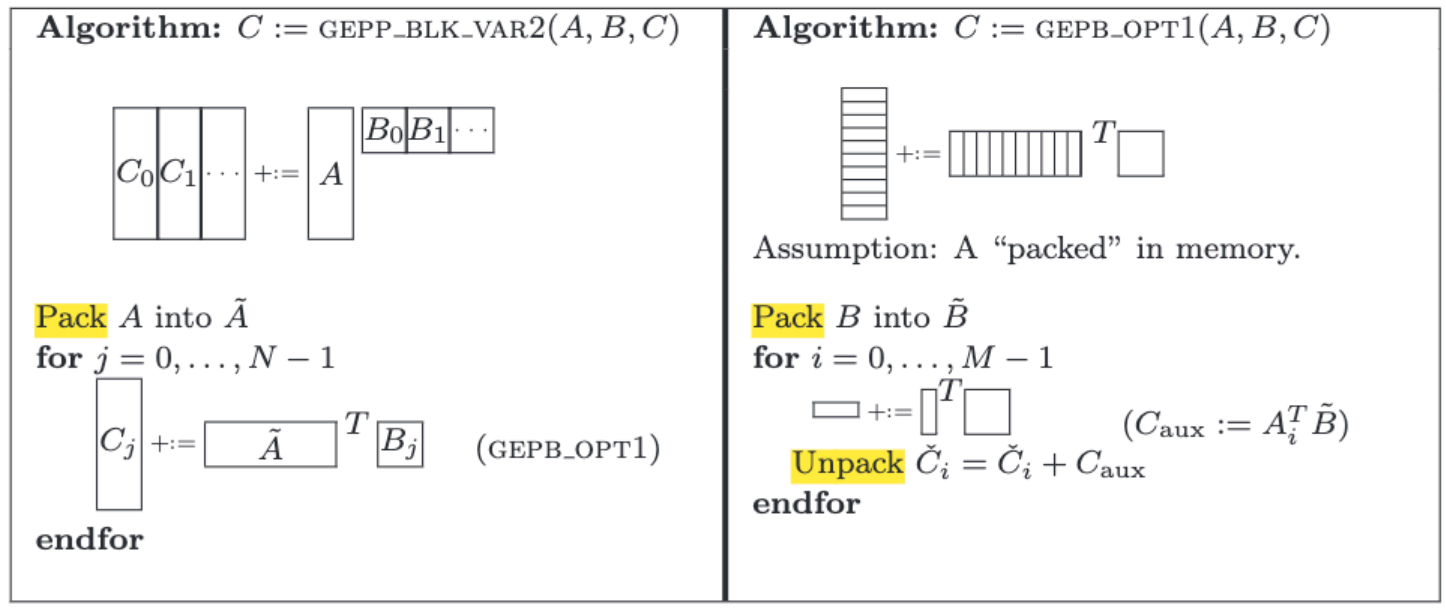


Fig. 10. Optimized implementation of GEPP (left) via calls to GEPB_OPT1 (right).

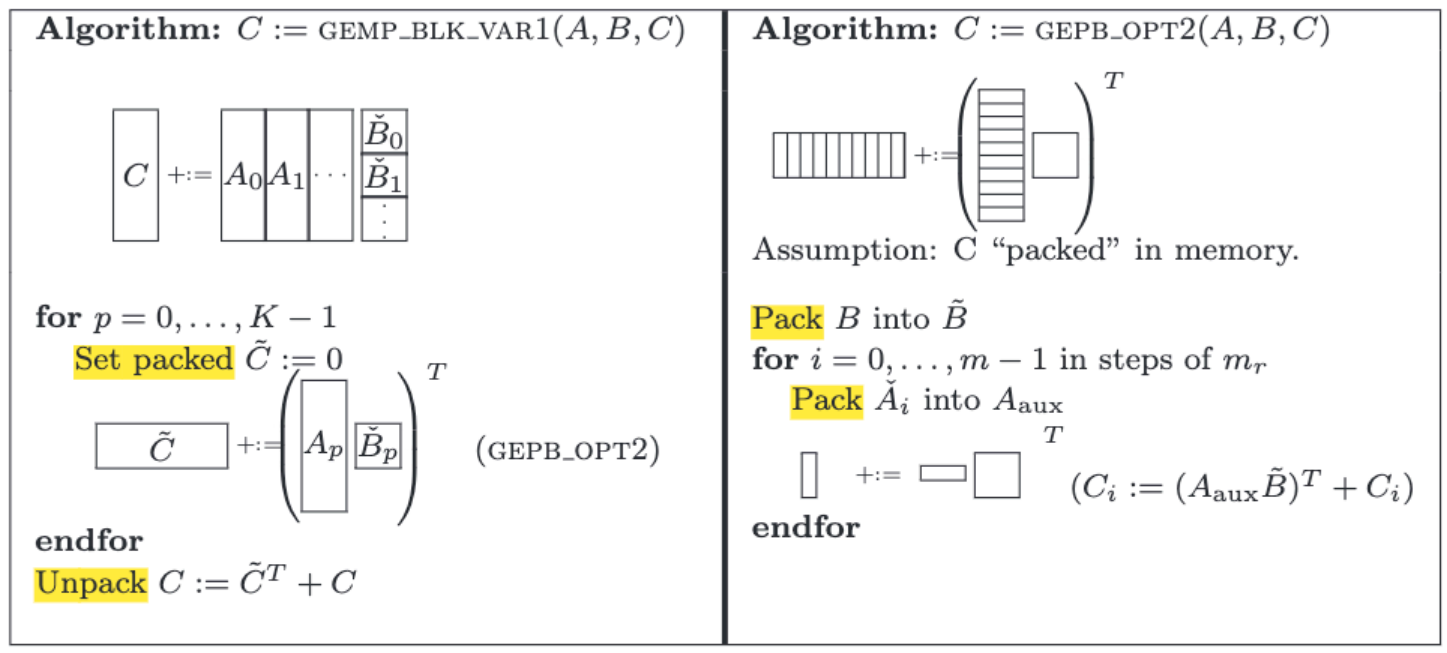


Fig. 11. Optimized implementation of GEMP (left) via calls to GEPB_OPT2 (right).

5.3 Implementing GEPP with GEPB

Fig. 10 shows how GEPP can be implemented in terms of GEPB. Now A is packed and transposed by GEPP to improve contiguous access to its elements. In GEPB B is packed and kept in the L2 cache, so that it is $T_{\tilde{B}}$ that we wish to maximize. While $\check{A}_i$, $\check{A}_{i+1}$, and C_{aux} each typically only require one TLB entry, C_i requires n_c if the leading dimension of C is large. Thus, $T_{\tilde{B}}$ is bounded by $T - (n_c + 3)$.

5.4 Implementing GEMP with GEPB

In Fig. 11 shows how GEMP can be implemented in terms of GEPB. This time a temporary $\tilde{C}$ is used to accumulate $\tilde{C} = (AB)^T$ and the L2 cache is mostly filled with a packed copy of $\tilde{B}$. Again, it is $T_{\tilde{B}}$ that we wish to maximize. While C_i, C_{i+1}, and A_{temp} each take up one TLB entry, $\check{A}_i$ requires up to m_c entries. Thus, $T_{\tilde{B}}$ is bounded by $T - (m_c + 3)$.

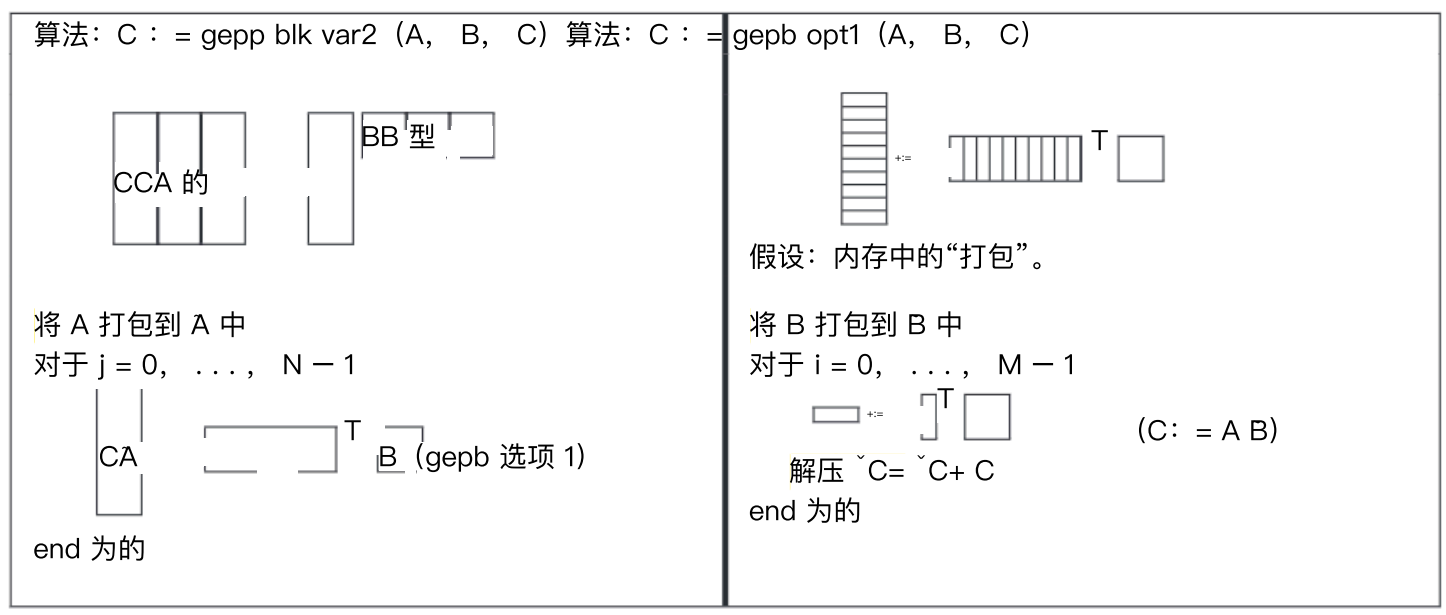


图 10.通过调用 gepb opt1（右）优化了 gepp（左）的实现。

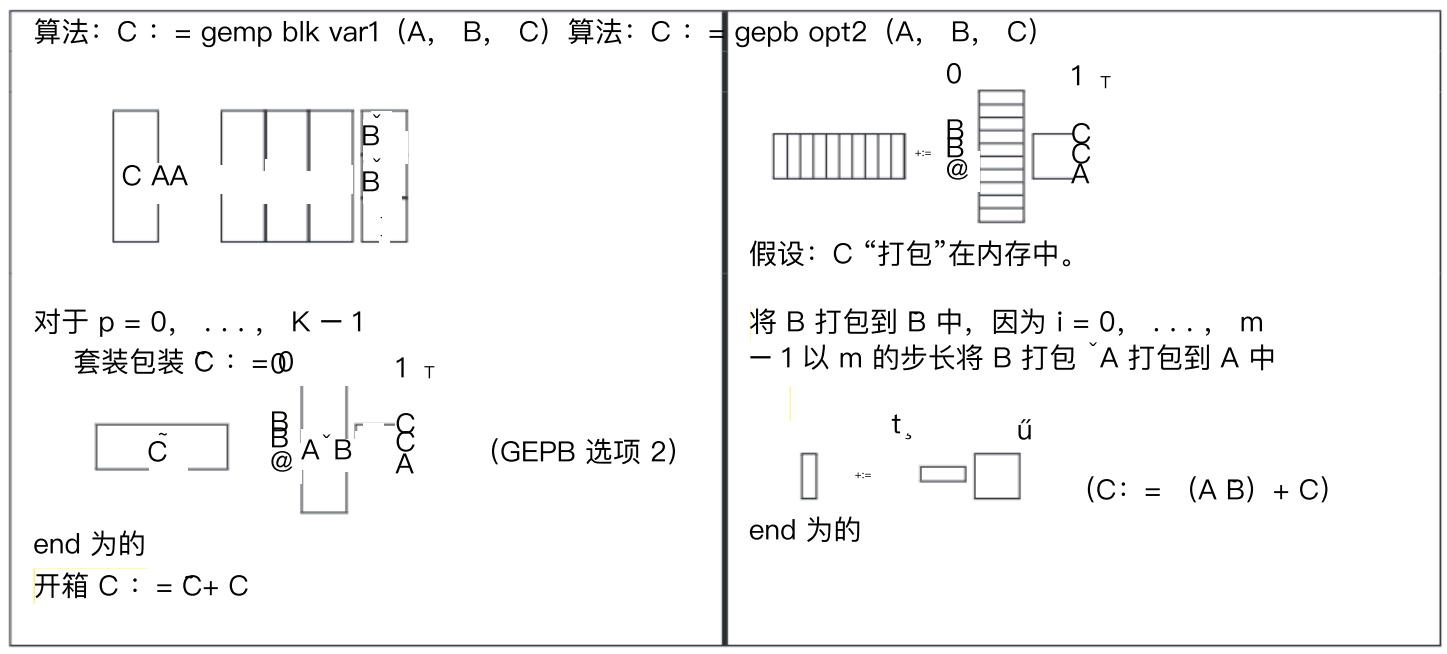


图 11.通过调用 gepb opt2（右）优化了 gimp（左）的实现。

5.3 使用 gepb 实现 gepp

图 10 显示了如何实现 gepp 在 gepb 方面。现在 A 被 gepp 打包和转置，以改善对其元素的连续访问。在 gepb 中，B 被打包并保存在 L2 缓存中，因此我们希望最大化的是 T。虽然 A、A 和 Ceach 通常只需要一个 TLB 条目，但如果 C 的前导维度很大，则 ˇC 需要 n。因此，T 以 T − （n+ 3） 为界。

5.4 使用 gepb 实现 gemp

在图 11 中，显示了如何根据 gepb 实现 gemp。这次使用临时 C 来累积 C = （AB），并且 L2 缓存大部分填充了 ˜B 的打包副本。同样，我们希望最大化的是 T。虽然 C、C 和 Aeach 占用一个 TLB 条目，但 ˇ A 最多需要 mentries。因此，T 以 T − （m+ 3） 为界。

5.5 Implementing GEPM and GEMP with GEPDOT

Similarly, GEPM and GEMP can be implemented in terms of the GEPDOT operation. This places a block of C in the L2 cache and updates it by multiplying a few columns of A times a few rows of B. Since we will argue below that this approach is likely inferior, we do not give details here.

5.6 Discussion

Fig. 4 suggests six different strategies for implementing a GEMM operation. Details for four of these options are given in Figs. 8–11. We now argue that if matrices are stored in column-major order, then the approach in Fig. 8, which corresponds to the path in Fig. 4 that always takes the top branch in the decision tree and is also given in Fig. 5, can in practice likely attain the best performance.

Our first concern is to attain the best possible bandwidth use from the L2 cache. Note that a GEPDOT-based implementation places a block of C in the L2 cache and reads *and writes* each of its elements as a few columns and rows of A and B are streamed from memory. This requires twice the bandwidth between the L2 cache and registers as do the GEBP and GEPBbased algorithms. Thus, we expect GEPDOT-based implementations to achieve worse performance than the other four options.

Comparing the pair of algorithms in Figs. 8 and 9 the primary difference is that the former packs B and streams elements of C from and to main memory, while the latter streams B from main memory, computes C in a temporary buffer, and finally unpacks by adding this temporary result to C. In practice, the algorithm in Fig. 8 can hide the cost of bringing elements of C from and to memory with computation while it exposes the packing of B as sheer overhead. The algorithm in Fig. 9 can hide the cost of bringing elements of B from memory, but exposes the cost of unpacking C as sheer overhead. The unpacking of C is a more complex operation and can therefore be expected to be more expensive than the packing of B, making the algorithm in Fig. 8 preferable over the one in Fig. 9. A similar argument can be used to rank the algorithm in Fig. 10 over the one in Fig. 11.

This leaves us with having to choose between the algorithms in Figs. 8 and 10, which on the surface appear to be symmetric in the sense that the roles of A and B are reversed. Note that the algorithms access C a few columns and rows at a time, respectively. If the matrices are stored in column-major order, then it is preferable to access a block of those matrices by columns. Thus the algorithm in Fig. 8 can be expected to be superior to all the other presented options.

Due to the level of effort that is required to implement kernels like GEBP, GEPB, and GEPDOT, we focus on the algorithm in Fig. 8 throughout the remainder of this paper.

We stress that the conclusions in this subsection are continguent on the observation that on essentially all current processors there is an advantage to blocking for the L2 cache. It is entirely possible that the insights will change if, for example, blocking for the L1 cache is preferred.

5.5 使用 gepdot 实现 gepm 和 gemp

同样，gepm 和 gemp 也可以根据 gepdot 作来实现。这会将 C 块放置在 L2 缓存中，并通过将 A 的几列乘以 B 的几行来更新它。由于我们将在下面论证这种方法可能较差，因此我们在这里不提供详细信息。

5.6 讨论

图 4 提出了实施 gemm 作的六种不同策略。图 8–11 给出了其中四个选项的详细信息。我们现在认为，如果矩阵以列优先顺序存储，那么图 8 中的方法（对应于图 4 中始终采用决策树中顶部分支的路径，并且也在图 5 中给出）在实践中可能会获得最佳性能。

我们首先关心的是从二级缓存中获得最佳带宽利用。
请注意，基于 gepdot 的实现将一个 C 块放置在 L2 缓存中，并在从内存中流式传输 A 和 B 的几列和行时读取和写入其每个元素。这需要 L2 缓存和寄存器之间的带宽是 gebp 和基于 gebb 的算法的两倍。因此，我们预计基于 gepdot 的实现将实现比其他四个选项更差的性能。

比较图 8 和图 9 中的一对算法，主要区别在于前者打包 B 并将 C 的元素从主存储器流式传输到主存储器，而后者从主存储器流式传输 B，在临时缓冲区中计算 C，最后通过将此临时结果添加到 C 中来解包。在实践中，图 8 中的算法可以通过计算隐藏将 C 元素从内存中引入和带入内存的成本，同时将 B 的打包暴露为纯粹的开销。图 9 中的算法可以隐藏从内存中引入 B 元素的成本，但将解包 C 的成本暴露为纯粹的开销。C 的解包是一个更复杂的作，因此可以预期比 B 的打包更昂贵，这使得图 8 中的算法比图 9 中的算法更可取。类似的论点可用于将图 10 中的算法与图 11 中的算法进行排名。

这让我们不得不在图 8 和图 10 中的算法之间做出选择，从表面上看，这些算法在 A 和 B 的作用是相反的意义上是对称的。请注意，算法一次分别访问 C 几列和几行。如果矩阵按列优先顺序存储，则最好按列访问这些矩阵的块。因此，可以预期图 8 中的算法优于所有其他提出的选项。

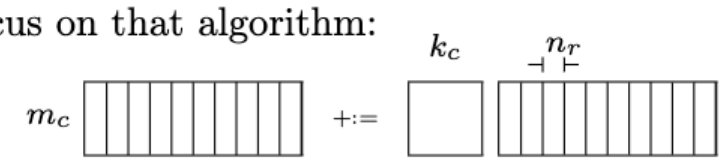
由于实现 gebp、gepb 和 gepdot 等内核所需的工作量，我们在本文的其余部分重点关注图 8 中的算法。

我们强调，本小节中的结论取决于以下观察结果：在基本上所有当前处理器上，阻塞 L2 缓存都有优势。例如，如果首选阻止 L1 缓存，则见解完全有可能发生变化。

6. MORE DETAILS YET

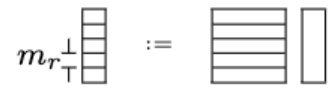
We now give some final insights into how registers are used by kernels like GEBP_OPT1, after which we comment on how parameters are chosen in practice.

Since it has been argued that the algorithm in Fig. 8 will likely attain the best performance, we focus on that algorithm:



6.1 Register blocking

Consider $C_{\text{aux}} := \tilde{A}B_j$ in Fig. 8 where $\tilde{A}$ and B_j are in the L2 and L1 caches, respectively. This operation is implemented by computing $m_r \times n_r$ submatrices of C_{aux} in the registers.



Notice that this means that during the computation of C_j it is not necessary that elements of that submatrix remain in the L1 or even the L2 cache: $2m_r n_r k_c$ flops are performed for the $m_r n_r$ memops that are needed to store the results from the registers to whatever memory layer. We will see that k_c is chosen to be relatively large.

The above figure allows us to discuss the packing of A into $\tilde{A}$ in more detail. In our implementation, $\tilde{A}$ is stored so that each $m_r \times k_c$ submatrix is stored contiguously in memory. Each such submatrix is itself stored in column-major order. This allows $C_{\text{aux}} := \tilde{A}B_j$ to be computed while accessing the elements of $\tilde{A}$ by striding strictly contiguously through memory. Implementations by others will often store $\tilde{A}$ as the transpose of A, which requires a slightly more complex pattern when accessing $\tilde{A}$.

6.2 Choosing $m_r \times n_r$

The following considerations affect the choice of $m_r \times n_r$:

—Typically half the available registers are used for the storing $m_r \times n_r$ submatrix of C. This leaves the remaining registers for prefetching elements of $\tilde{A}$ and $\tilde{B}$.

—It can be shown that amortizing the cost of loading the registers is optimal when $m_r \approx n_r$.

—As mentioned in Section 4.2.1, the fetching of an element of $\tilde{A}$ from the L2 cache into registers must take no longer than the computation with a previous element so that ideally $n_r \geq R_{\text{comp}}/(2R_{\text{load}})$ must hold. R_{comp} and R_{load} can be found under "flops/cycle" and "Sustained Bandwidth", respectively, in Fig. 12.

A shortage of registers will limit the performance that can be attained by GEBP_OPT1, since it will impair the ability to hide constraints related to the bandwidth to the L2 cache.

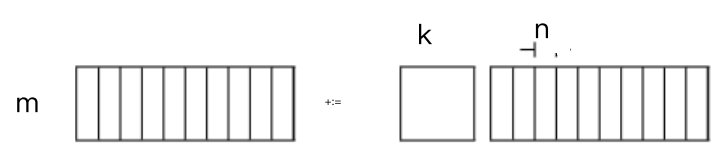
6.3 Choosing k_c

To amortize the cost of updating $m_r \times n_r$ elements of C_j the parameter k_c should be picked to be as large as possible.

The choice of k_c is limited by the following considerations:

6. 更多细节

现在，我们对 gebp opt1 等内核如何使用寄存器进行了一些最终的见解，之后我们评论了在实践中如何选择参数。
由于有人认为图 8 中的算法可能会获得最佳性能，因此我们关注该算法：



6.1 寄存器阻塞

考虑图 8 中的 C：= AB，其中 A 和 Bare 分别位于 L2 和 L1 缓存中。此作是通过计算 Cin 寄存器的 m× nsub 矩阵来实现的。

请注意，这意味着在计算 Cit 期间，该子矩阵的元素不必保留在 L1 甚至 L2 缓存中：对将结果从寄存器存储到任何内存层所需的 mnmemops 执行 2mnkflops。我们将看到 kis 选择相对较大。
上图让我们可以更详细地讨论将 A 打包到 Ā 中。在我们的实现中，Ā 被存储，以便每个 m× ksubmatrix 连续存储在内存中。每个这样的子矩阵本身都按列主顺序存储。这允许在访问 ˜A 的元素时通过严格连续地跨过内存来计算 C：= ˜AB。其他人的实现通常会将 ˜A 存储为 A 的转置，这在访问 ˜A 时需要稍微复杂的模式。

6.2 选择 m× n

以下考虑因素会影响 m×n 的选择：

—通常，一半的可用寄存器用于存储 C 的 m× nsub 矩阵。这留下了剩余的寄存器用于预取 ˜A 和 ˜B 的元素。—可以表明，当 m≈ n 时，摊销加载寄存器的成本是最佳的。

—如果从 L2 缓存到寄存器的时间不得超过使用前一个元素的计算时间，因此理想情况下 n≥ R/（2R） 必须成立。兰德 R 分别位于图 12 的“flops/cycle”和“Sustained Bandwidth”下。

寄存器的短缺将限制 gebp opt1 可以获得的性能，因为它会损害隐藏与 L2 缓存带宽相关的约束的能力。

6.3 选择 k

为了摊销更新 m 的成本× c 参数 k 的元素应该选择尽可能大。

k 的选择受到以下考虑因素的限制：

—Elements from B_j are reused many times, and therefore must remain in the L1 cache. In addition, the set associativity and cache replacement policy further limit how much of the L1 cache can be occupied by B_j. In practice, $k_c n_r$ floating point numbers should occupy less than half of the L1 cache so that elements of $\tilde{A}$ and C_{aux} do not evict elements of B_j.

—The footprint of $\tilde{A}$ ($m_c \times k_c$ floating point numbers) should occupy a considerable fraction of the L2 cache.

In our experience the optimal choice is such that k_c double precision floating point numbers occupy half of a page. This choice typically satisfies the other constraints as well as other architectural constraints that go beyond the scope of this paper.

6.4 Choosing m_c

It was already argued that $m_c \times k_c$ matrix $\tilde{A}$ should fill a considerable part of the smaller of (1) the memory addressable by the TLB and (2) the L2 cache. In fact, this is further constrained by the set-associativity and replacement policy of the L2 cache. In practice, m_c is typically chosen so that $\tilde{A}$ only occupies about half of the smaller of (1) and (2).

7. EXPERIMENTS

In this section we report performance attained by implementations of the DGEMM BLAS routine using the techniques described in the earlier sections. It is *not* the purpose of this section to show that our implementations attain better performance than those provided by vendors and other projects. (We do note, however, that they are highly competitive.) Rather, we attempt to demonstrate that the theoretical insights translate to practical implementations.

7.1 Algorithm chosen

Implementing all algorithms discussed in Section 5 on a cross-section of architectures would be a formidable task. Since it was argued that the approach in Fig. 8 is likely to yield the best overall performance, it is that variant which was implemented. The GEBP_opt1 algorithm was carefully assembly-coded for each of the architectures that were considered. The routines for packing A and B into $\tilde{A}$ and $\tilde{B}$, respectively, were coded in C, since compilers appear to be capable of optimizing these operations.

7.2 Parameters

In Fig. 12 we report the physical and algorithmic parameters for a cross-section of architectures. Not all the parameters are taken into account in the analysis in this paper. These are given for completeness.

The following parameters require extra comments:

Duplicate This parameter indicates whether elements of matrix B are duplicated as part of the packing of B. This is necessary in order to take advantage of SSE2 instructions on the Pentium4 (Northwood) and Opteron processors. Although the Core 2 Woodcrest has an SSE3 instruction set, instructions for duplication are issued by the multiply unit and the same techniques as for the Northwood architecture must be employed.

– 裸露中的元素被重复使用多次，因此必须保留在 L1 缓存中。此外，set associativity 和 cache replacement 策略进一步限制了 B 可以占用多少 L1 缓存。在实践中，浮点数应占用 L1 缓存的一半以下，以便 ˜A 和 C 的元素不会驱逐 B 的元素。—A （m×k 浮点数） 的占用空间应占 L2 缓存的相当大一部分。

根据我们的经验，最佳选择是 kdouble 精度浮点数占据半页。此选择通常满足其他约束以及超出本文范围的其他体系结构约束。

6.4 选择 m

已经填充 (1) mc×kc 子块内存和 (2) L2 缓存中较小者中的相当一部分。事实上，这进一步受到 L2 缓存的集合关联性和替换策略的约束。在实践中，通常选择 mis 以便 ˜A 仅占据 (1) 和 (2) 中较小的一半左右。

7. 实验

在本节中，我们报告了使用前面部分中描述的技术实现 dgemm BLAS 例程所获得的性能。本节的目的不是要表明我们的实现比供应商和其他项目提供的实现性能更好。（然而，我们确实注意到它们的竞争非常激烈。相反，我们试图证明理论见解可以转化为实际实施。

7.1 选择的算法

在架构的横截面上实现第 5 节中讨论的所有算法将是一项艰巨的任务。由于有人认为图 8 中的方法可能会产生最佳的整体性能，因此实施的是该变体。gebp opt1 算法针对所考虑的每个架构进行了仔细的汇编编码。将 A 和 B 打包为 ˜A 和 ˜的例程

B 分别是用 C 编码的，因为编译器似乎能够优化这些作。

7.2 参数

在图 12 中，我们报告了架构横截面的物理和算法参数。本文的分析中并未考虑所有参数。这些是为了完整起见而给出的。

以下参数需要额外的注释：

复制 此参数指示矩阵 B 的元素是否作为 B 打包的一部分进行复制。这是必要的，以便利用 Pentium4 （Northwood） 和 Opteron 处理器上的 SSE2 指令。尽管 Core 2 Woodcrest 具有 SSE3 指令集，但复制指令由乘法单元发出，并且必须采用与 Northwood 架构相同的技术。

Architecture		# of registers	flops/cycle	Duplicate	L1 cache				L2 cache				L3 cache				TLB				Block sizes		
Sub Architecture	Core				Size (Kbytes)	Line Size	Associativity	Sustained Bandwidth	Size (Kbytes)	Line Size	Associativity	Sustained Bandwidth	Size (Kbytes)	Line Size	Associativity	Sustained Bandwidth	Page Size (Kbytes)	L1 TLB	L2 TLB	Covered Area (Kbytes)	$\tilde{A}$ (Kbytes)	$m_c \times k_c$	$m_r \times n_r$
x86																							
Pentium3	Katmai	8	1	N	16	32	4	0.95	512	32	4	0.40					4	64	-	256		64×256	2×2
	Coppermine	8	1	N	16	32	4	0.95	256	32	4	0.53					4	64	-	256		64×256	2×2
Pentium4	Northwood	8[1]	2	Y	8	64	4	1.88	512	64	8	1.06					4	64	-	256	224	224×128	4×2
	Prescott	8[1]	2	N	16	64	4	1.92	2K	64	8	1.03					4	64	-	256	768	768×128	2×4
Opteron		8[1]	2	Y	64	64	2	2.00	1K	64	16	0.71					4	32	512	2K[2]	768	384×256	2×4
x86_64 (EM64T)																							
Pentium4	Prescott	16[1]	2	N	16	64	4	1.92	2K	64	8	1.03					4	64	-	256	$\approx$ 1K	696×192	4×4
Core 2	Woodcrest	16[1]	4	Y	32	64	8	2.00	4K	64	8	1.00					4	256	-	1K	1K	512×256	4×4
Opteron		16[1]	2	Y	64	64	2	2.00	1K	64	16	0.71					4	32	512	2K[2]	608	384×256	4×4
IA64																							
Itanium2		128	4	N	16	64	4	-	256	128	8	4.00	6K	128	24	2.00	16	-	128	2K[2]	1K	$128 \times 1K$	8×8
POWER																							
POWER3		32	4	N	64	128	4	2.00	1K	128	1	0.75					4	256	-	1K	512	256×256	4×4
POWER4		32	4	N	32	128	2	1.95	1.4K	128	4	0.93	128K	512	8	-	4	128[4]	1K[4]	4K[2]	288	144×256	4×4
PPC970		32	4	N	32	128	2	2.00	512	128	8	0.92					4	128[4]	1K[4]	4K[2]	320	160×256	4×4
POWER5		32	4	N	32	128	2	2.00	1.92K	128	10	0.93		128			4	128[4]	1K[4]	4K[2]	512	256×256	4×4
PPC440 FP2		32[1]	4	N	32	32	64	2.00	2	128	Full	0.75	4K	128	8	0.75	-[3]	-[3]	-[3]	-[3]	3K	$128 \times 3K$	8×4
Alpha																							
EV4		32	1	N	16	32	1	0.79	2K	32	1	0.15					8	32	-	256	14	32×56	4×4
EV5		32	2	N	8	32	1	1.58	96	64	3	1.58	2K	64	1	0.22	8	64	-	512	63	56×144	4×4
EV6		32	2	N	64	64	2	1.87	4K	64	1	0.62					8	128	-	1K	512	128×512	4×4
SPARC																							
IV		32	4	N	64	32	4	0.99	8K	128	2	0.12	8K	128	2	0.12	8	16	512	4K[2]	2K	512×512	4×4

[1] Registers hold 2 floating point numbers each. [2] indicates that the Covered Area is determined from the L2 TLB. [3] IBM has not not disclosed this information. [4] On these machines there is a D-ERAT (Data cache Effective Real to Address Translation [table]) that takes the place of the L1 TLB and a TLB that acts like an L2 TLB.

Fig. 12. Parameters for a sampling of current architectures.

建筑		# 寄存器	翻牌/循环	重复	L1 缓存				L2 缓存				三级缓存				TLB 的				块大小		
子建筑	核心				大小（千字节）	线大小	关联性	持续带宽	大小（千字节）	线大小	关联性	持续带宽	大小（千字节）	线大小	关联性	持续带宽	页面大小（千字节）	L1 TLB	L2 TLB	覆盖面积（千字节）	A（千字节）	$m_c \times k_c$	$m_r \times n_r$
x86 奔腾 3																							
	卡特迈	8	1	N	16	32	4	0.95	512	32	4	0.40					4	64	–	256		64 × 256	2 × 2
	铜矿	8	1	N	16	32	4	0.95	256	32	4	0.53					4	64	–	256		64 × 256	2 × 2
奔腾 4	诺斯伍德	8[1]	2	Y	8	64	4	1.88	512	64	8	1.06					4	64	–	256	224	224 × 128	4 × 2
	普雷斯科特	8[1]	2	N	16	64	4	1.92	2K	64	8	1.03					4	64	–	256	768	768 × 128	2 × 4
光龙		8[1]	2	Y	64	64	2	2.00	1K	64	16	0.71					4	32	512	2K[2]	768	384 × 256	2 × 4
x86 64 （EM64T）奔腾 4 普雷斯科特																							
		16[1]	2	N	16	64	4	1.92	2K	64	8	1.03					4	64	–	256	≈ 1K	696 × 192	4 × 4
核心 2	伍德克雷斯特	16[1]	4	Y	32	64	8	2.00	4K 的	64	8	1.00					4	256	–	1K	1K	512 × 256	4 × 4
光龙		16[1]	2	Y	64	64	2	2.00	1K	64	16	0.71					4	32	512	2K[2]	608	384 × 256	4 × 4
IA64 安腾 2																							
		128	4	N	16	64	4	–	256	128	8	4.00	6K	128	24	2.00	16	–	128	2K[2]	1K	128 × 1K	8 × 8
POWERPOWER3																							
		32	4	N	64	128	4	2.00	1K	128	1	0.75					4	256	–	1K	512	256 × 256	4 × 4
电源 4		32	4	N	32	128	2	1.95	1.4K	128	4	0.93	128K	512	8	–	4	128[4]	1K[4]	4K 的[2]	288	144 × 256	4 × 4
PPC970 型		32	4	N	32	128	2	2.00	512	128	8	0.92					4	128[4]	1K[4]	4K 的[2]	320	160 × 256	4 × 4
功率 5		32	4	N	32	128	2	2.00	1.92K	128	10	0.93		128			4	128[4]	1K[4]	4K 的[2]	512	256 × 256	4 × 4
PPC440 FP2		32[1]	4	N	32	32	64	2.00	2	128	满	0.75	4K 的	128	8	0.75	–	–	–	–	3K 的	128 × 3K 的	8 × 4
阿尔法 EV4																							
		32	1	N	16	32	1	0.79	2K	32	1	0.15					8	32	–	256	14	32 × 56	4 × 4
EV5		32	2	N	8	32	1	1.58	96	64	3	1.58	2K	64	1	0.22	8	64	–	512	63	56 × 144	4 × 4
EV6		32	2	N	64	64	2	1.87	4K 的	64	1	0.62					8	128	–	1K	512	128 × 512	4 × 4
斯帕西夫																							
		32	4	N	64	32	4	0.99	8K 的	128	2	0.12	8K 的	128	2	0.12	8	16	512	4K 的[2]	2K	512 × 512	4 × 4

[1] 寄存器每个寄存器包含 2 个浮点数。 [2] 表示覆盖区域是根据 L2 TLB 确定的。 [3] IBM 没有披露了这一信息。 [4] 在这些机器上有一个 D-ERAT（数据缓存有效实数到地址转换 [表]），它需要 L1 TLB 和充当 L2 TLB 的 TLB 的值置。

图 12.当前体系结构采样的参数。

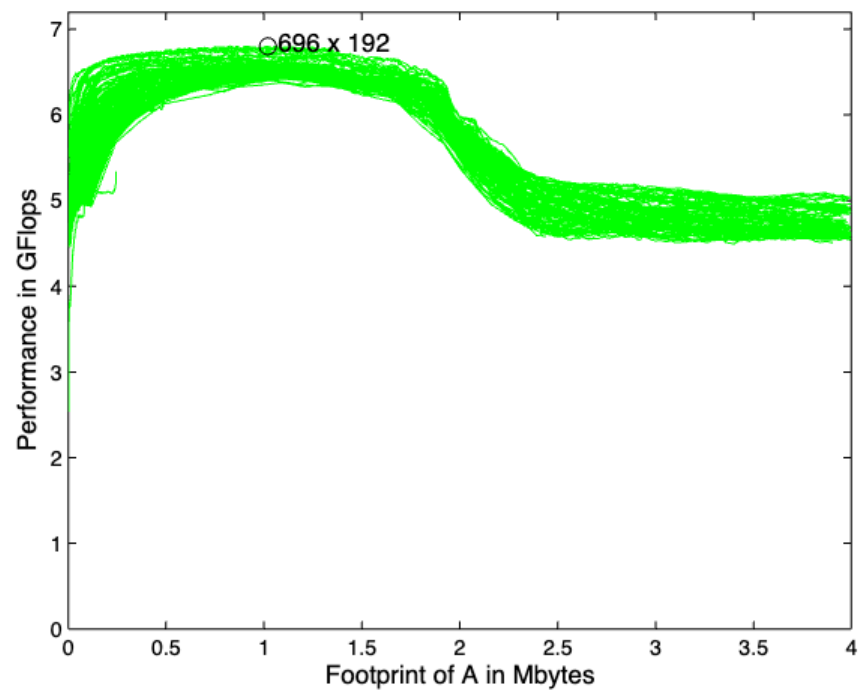


Fig. 13. Performance on the Pentium4 Prescott (3.6 GHz) of DGEMM for different choices of m_c and k_c, where m_c and k_c are varied from 8 to 2000 in steps of 8. This architecture has a 2Mbyte L2 cache, which explains the performance degradation when the footprint of $\breve{A}$ is about that size. The best performance is attained for $m_c \times k_c = 696 \times 196$, when the footprint is around 1 Mbyte.

Sustained Bandwidth This is the *observed* sustained bandwidth, in doubles/cycle, from the indicated memory layer to the registers.

Covered Area This is the size of the memory that can be addressed by the TLB. Some architectures have a (much slower) level 2 TLB that serves the same function relative to an L1 TLB as does an L2 cache relative to an L1 cache. Whether to limit the size of $\tilde{A}$ by the number of entries in L1 TLB or L2 TLB depends on the cost of packing into $\tilde{A}$ and $\tilde{B}$.

$\tilde{A}$ **(Kbytes)** This indicates how much memory is set aside for matrix $\tilde{A}$.

7.3 Focus on the Intel Pentium 4 Prescott processor (3.6 GHz, 64bit)

We discuss the implementation for the Intel Pentium 4 Prescott processor in greater detail. In Section 7.4 we more briefly discuss implementations on a few other architectures.

Equation (3) indicates that in order to hide the prefetching of elements of $\tilde{A}$ with computation parameter n_r must be chosen so that $n_r \geq R_{\text{comp}}/(2R_{\text{load}})$. Thus, for this architecture, $n_r \geq 2/(2 \times 1.03) \approx 0.97$. Also, EM64T architectures, of which this Pentium4 is a member, have 16 registers that can store two double precision floating point numbers each. Eight of these registers are used for storing entries of C: $m_r \times n_r = 4 \times 4$.

The choice of parameter k_c is complicated by the fact that updating the indexing in the loop that computes inner products of columns of $\tilde{A}$ and $\tilde{B}$ is best avoided on this architecture. As a result, that loop is completely unrolled, which means that storing the resulting code in the instruction cache becomes an issue, limiting k_c to 192. This is slightly smaller than the $k_c = 256$ that results from the limitation discussed in Section 6.3.

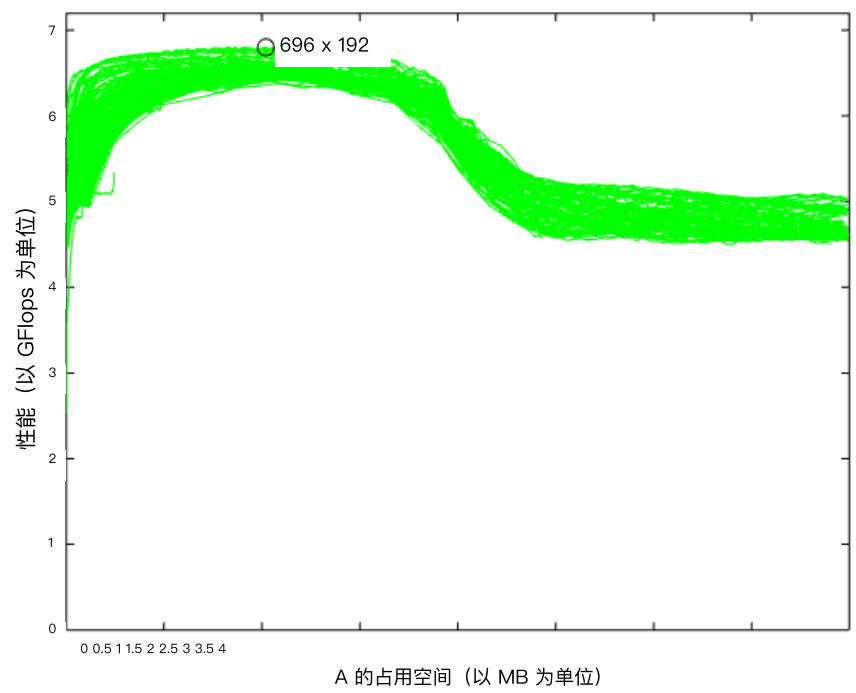


图 13.在 pentium4 Prescott （3.6 GHz） 的 dgemm 上针对不同 m 和 k 选择的性能，其中 mand kare 从 8 到 2000 不等，以 8 为步长。该架构具有 2 MB 的 L2 缓存，这解释了当 ˇA 的占用空间大约是该大小时性能下降的原因。当占用空间约为 1 MB 时，m× k= 696 × 196 可获得最佳性能。

持续带宽 这是观察到的从指定存储器层到寄存器的持续带宽，以双倍/周期为单位。

覆盖区域 这是 TLB 可以寻址的内存大小。
某些架构具有（慢得多的）2 级 TLB，其相对于 L1 TLB 的功能与 L2 缓存相对于 L1 缓存的功能相同。是否通过 L1 TLB 或 L2 TLB 中的条目数量来限制 A 的大小取决于打包到 ˜A 和 ˜B 的成本。

˜A （KB） 这表示为矩阵预留了多少内存 ˜一个。

7.3 专注于 Intel Pentium 4 Prescott 处理器（3.6 GHz，64 位）

我们更详细地讨论了英特尔奔腾 4 Prescott 处理器的实现。在第 7.4 节中，我们更简要地讨论了其他一些架构的实现。

必须选择具有 参数 nr 的隐藏元素 预取 R/（2R）。因此，对于这种架构，n≥ 2/（˜2 × 1.03） ≈ 0.97。此外，EM64T 架构（Pentium4 是其中的一员）有 16 个寄存器，每个寄存器可以存储两个双精度浮点数。其中八个寄存器用于存储 C：m×n= 4 × 4 的条目。

参数 ki 的选择变得复杂，因为在此架构上最好避免更新循环中的索引，该循环计算 ˜A 和 ˜B 列的内积。结果，该循环被完全展开，这意味着将生成的代码存储在指令缓存中成为一个问题，将 k 限制为 192。这略小于第 6.3 节中讨论的限制导致的 k= 256。

In Figure 13 we show the performance of DGEMM for different choices of m_c and k_c[3]. This architecture is the one exception to the rule that $\tilde{A}$ should be addressable by the TLB, since a TLB miss is less costly than on other architectures. When $\tilde{A}$ is chosen to fill half of the L2 cache the performance is slightly better than when it is chosen to fill half of the memory addressable by the TLB.

The Northwood version of the Pentium4 relies on SSE2 instructions to compute two flops per cycle. This instruction requires entries in B to be duplicated, a data movement that is incorporated into the packing into buffer $\tilde{B}$. The SSE3 instruction supported by the Prescott subarchitecture does not require this duplication when copying to $\tilde{B}$.

In Fig. 14 we show the performance attained by the approach on this Pentium 4 architecture. In this figure the top graph shows the case where all matrices are square while the bottom graph reports the case where $m = n = 2000$ and k is varied. We note that GEPP with k relatively small is perhaps the most commonly encountered special case of GEMM.

—The top curve, labeled "Kernel", corresponds to the performance of the kernel routine (GEBP_opt1).

—The next lower curve, labeled "dgemm", corresponds to the performance of the DGEMM routine implemented as a sequence of GEPP operations. The GEPP operation was implemented via the algorithms in Fig. 8.

—The bottom two curves correspond the percent of time incurred by routines that pack A and B into $\tilde{A}$ and $\tilde{B}$, respectively. (For these curves only the labeling along the right axis is relevant.)

The overhead caused by the packing operations accounts almost exactly for the degradation in performance from the kernel curve to the DGEMM curve.

The graphs in Fig. 15 investigate the performance of the implementation when m and n are varied. In the top graph m is varied while $n = k = 2000$. When m is small, as it would be for a GEPM operation, the packing of B into $\tilde{B}$ is not amortized over sufficient computation, yielding relatively poor performance. One solution would be to skip the packing of B. Another would be to implement the algorithm in Fig. 9. Similarly, in the bottom graph n is varied while $m = k = 2000$. When n is small, as it would be for a GEMP operation, the packing of A into $\tilde{A}$ is not amortized over sufficient computation, again yielding relatively poor performance. Again, one could contemplate skipping the packing of A (which would require the GEBP operation to be cast in terms of AXPY operations instead of inner-products). An alternative would be to implement the algorithm in Fig. 11.

7.4 Other architectures

For the remaining architectures we discuss briefly how parameters are selected and show performance graphs, in Figs. 16–20, that correspond to those for the Pentium 4 in Fig. 14.

[3]Ordinarily examining the performance for all possible combinations of m_c and k_c is not part of our optimization process. Rather, the issues discussed in this paper are used to identify a few possible combinations of parameters, and only combinations of m_c and k_c near these candidate parameters are tried. The graph is included to illustrate the effect of picking different parameters.

在图 13 中，我们显示了 dgemm 在不同选择 mand k 时的性能。此架构是 A 应可由 TLB 寻址的规则的一个例外，因为 TLB 未命中的成本低于其他架构。当选择 A 来填充一半的 L2 缓存时，性能比选择它来填充 TLB 可寻址内存的一半时略好。

Pentium4 的 Northwood 版本依赖于 SSE2 指令来计算每个周期的两次 flops。该指令要求复制 B 中的条目，这是一种数据移动，并入缓冲区 ˜B 中。Prescott 子架构支持的 SSE3 指令在复制到 ˜B 时不需要此复制。

在图 14 中，我们展示了该方法在 Pentium 4 架构上获得的性能。在此图中，上图显示了所有矩阵都是正方形的情况，而下图报告了 m = n = 2000 且 k 变化的情况。我们注意到，k 相对较小的 gepp 可能是 gemm 最常见的特例。

—顶部曲线标记为“内核”，对应于内核例程 （gebp opt1） 的性能。– 下一条标记为“dgemm”的下一条曲线对应于作为一系列 gepp 作实现的 dgemm 例程的性能。gepp 作通过图 8 中的算法实现。– 底部两条曲线对应于将 A 和 B 分别打包到 ˜A 和 ˜B 中的例程所花费的时间百分比。(对于这些曲线，只有沿右轴的标记是相关的。

打包作引起的开销几乎恰好解释了从内核曲线到 dgemm 曲线的性能下降。

图 15 中的图表研究了 m 和 n 变化时实现的性能。在上图中，m 是变化的，而 n = k = 2000。当 m 很小时，就像 gepm 运算一样，将 B 打包到 B 中不会在足够的计算中摊销，从而产生相对较差的性能。一种解决方案是跳过 B 的打包。另一种方法是实现图 9 中的算法。类似地，在底部图中，n 是变化的，而 m = k = 2000。当 n 很小时，就像 gemp 作一样，将 A 打包到 A 中不会在足够的计算中摊销，这再次产生相对较差的性能。同样，可以考虑跳过 A 的打包（这将需要根据 axpy 作而不是内积来强制转换 gebp 作）。

另一种方法是实现图 11 中的算法。

7.4 其他架构

对于其余架构，我们简要讨论了如何选择参数，并在图 16–20 中显示了与图 14 中 Pentium 4 的性能图相对应的性能图。

3 通常检查所有可能的 mand kis 组合的性能，不属于我们优化过程的一部分。相反，本文中讨论的问题用于确定一些可能的参数组合，并且仅尝试这些候选参数的组合。该图用于说明选择不同参数的效果。

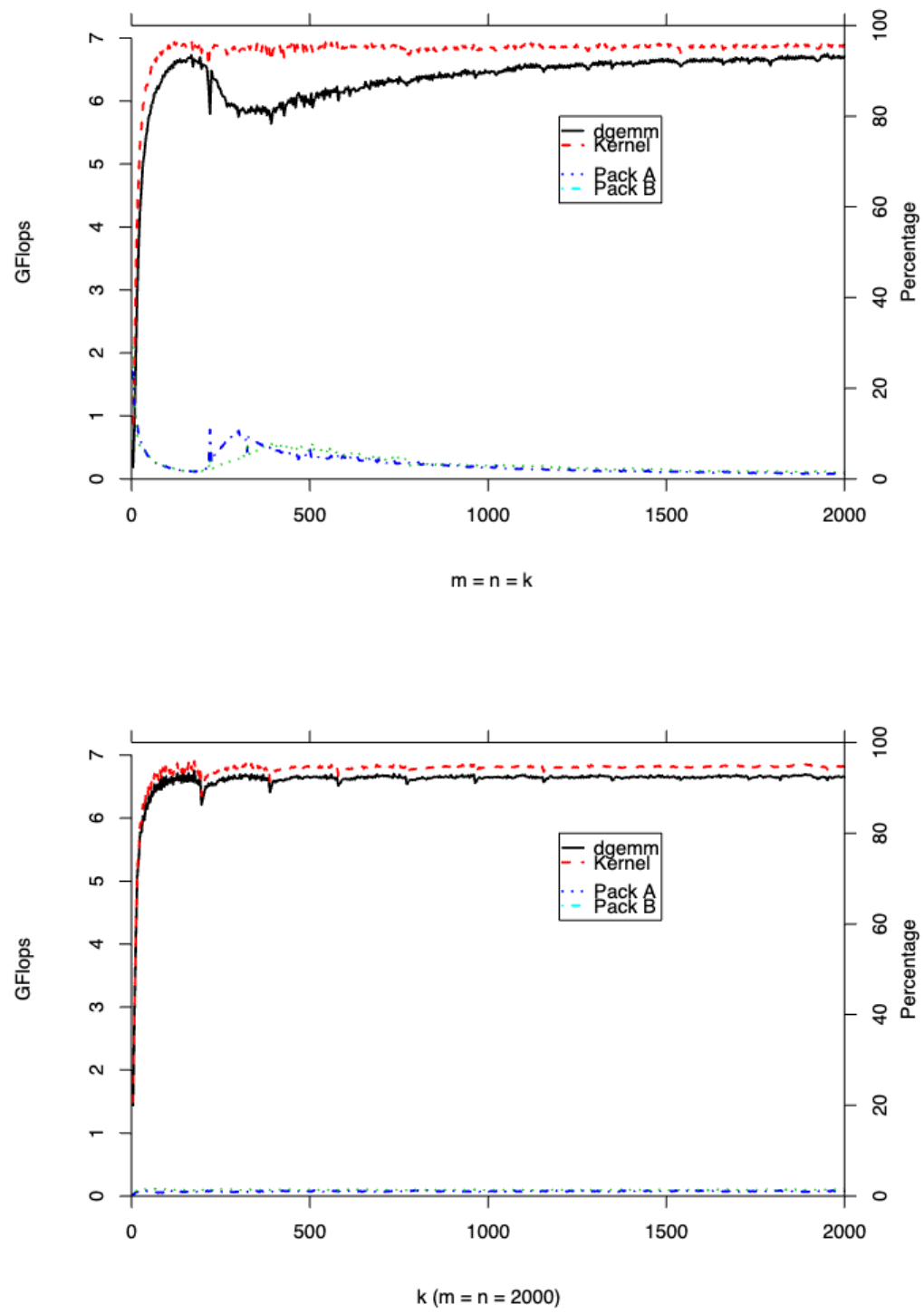


Fig. 14. Pentium4 Prescott (3.6 GHz).

AMD Opteron processor (2.2 GHz, 64bit)

For the Opteron architecture $n_r \geq R_{\text{comp}}/(2R_{\text{load}}) = 2/(2 \times 0.71) \approx 1.4$. The observed optimal choice for storing entries of C in registers is $m_r \times n_r = 4 \times 4$.

Unrolling of the inner loop that computes the inner-product of columns of $\tilde{A}$ and $\tilde{B}$ is not necessary like it was for the Pentium4, nor is the size of the L1 cache an issue. Thus, k_c is taken so that a column of $\tilde{B}$ fills half a page: $k_c = 256$. By taking $m_c \times k_c = 384 \times 256$ matrix $\tilde{A}$ fills roughly one third of the space addressable by the TLB.

The latest Opteron architectures support SSE3 instructions, we have noticed that duplicating elements of $\tilde{B}$ is still beneficial. This increases the cost of packing into $\tilde{B}$, decreasing performance by about 3%.

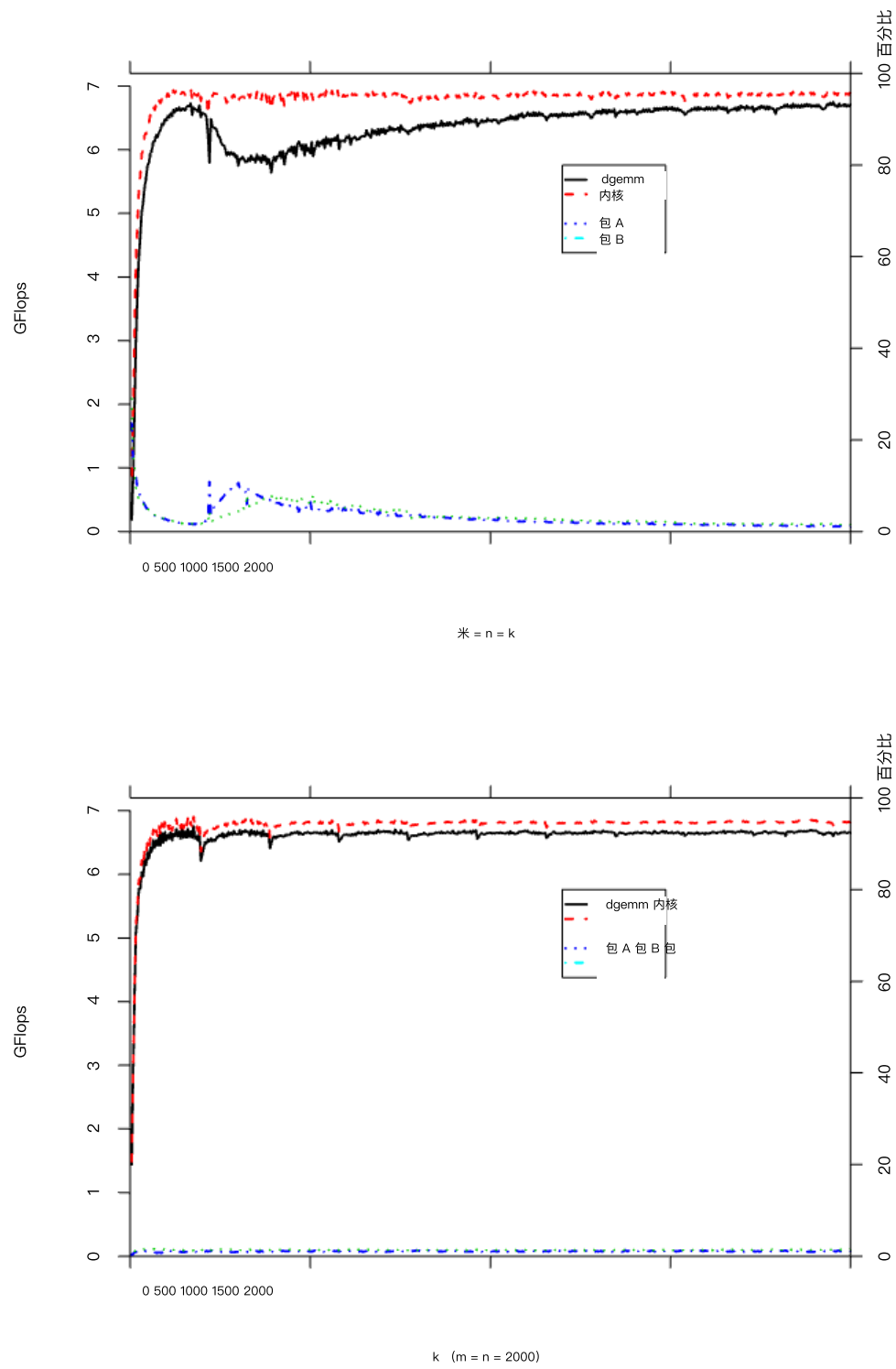


图 14.奔腾 4 Prescott （3.6 GHz）。

AMD 皓龙处理器（2.2 GHz，64 位）

对于 Opteron 架构 n≥ R/（2R） = 2/（2 × 0.71） ≈ 1.4。观察到的在寄存器中存储 C 条目的最佳选择是 m× n= 4 × 4。展开计算 ˜A 和 ˜B 列的内积的内部循环没有必要像 Pentium4 那样，L1 缓存的大小也不是问题。因此，采用 k，使得一列 B 填满半页：k= 256。通过取 m× k= 384 × 256 矩阵 A 填充了 TLB 可寻址空间的大约三分之一。

最新的 Opteron 架构支持 SSE3 指令，我们注意到复制 ˜B 的元素仍然是有益的。这增加了包装成本 ˜
B，性能下降约 3%。

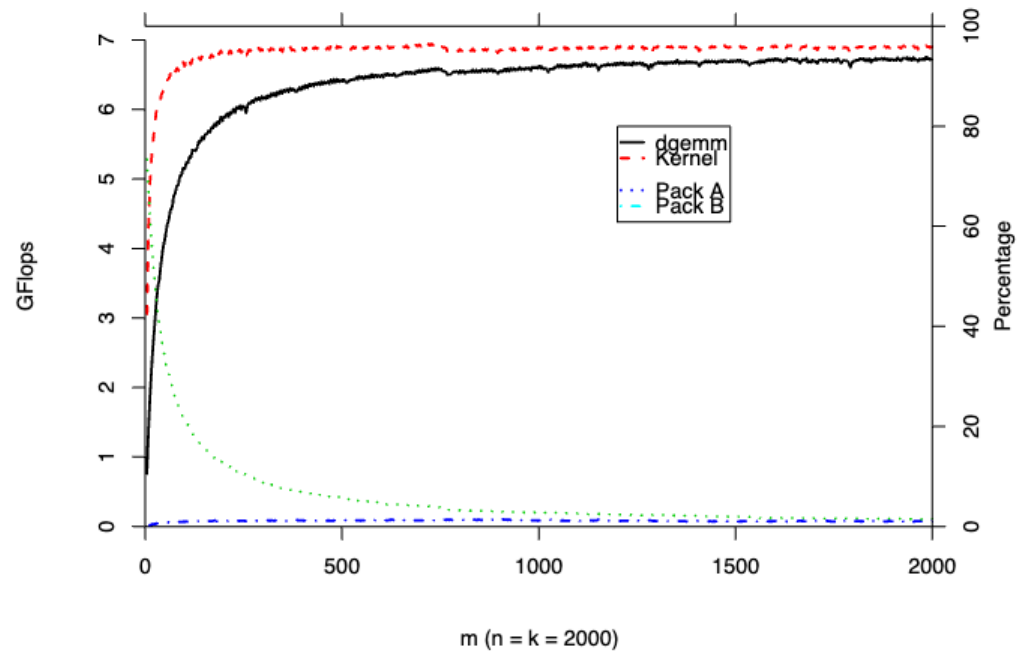


Fig. 15. Pentium4 Prescott(3.6 GHz).

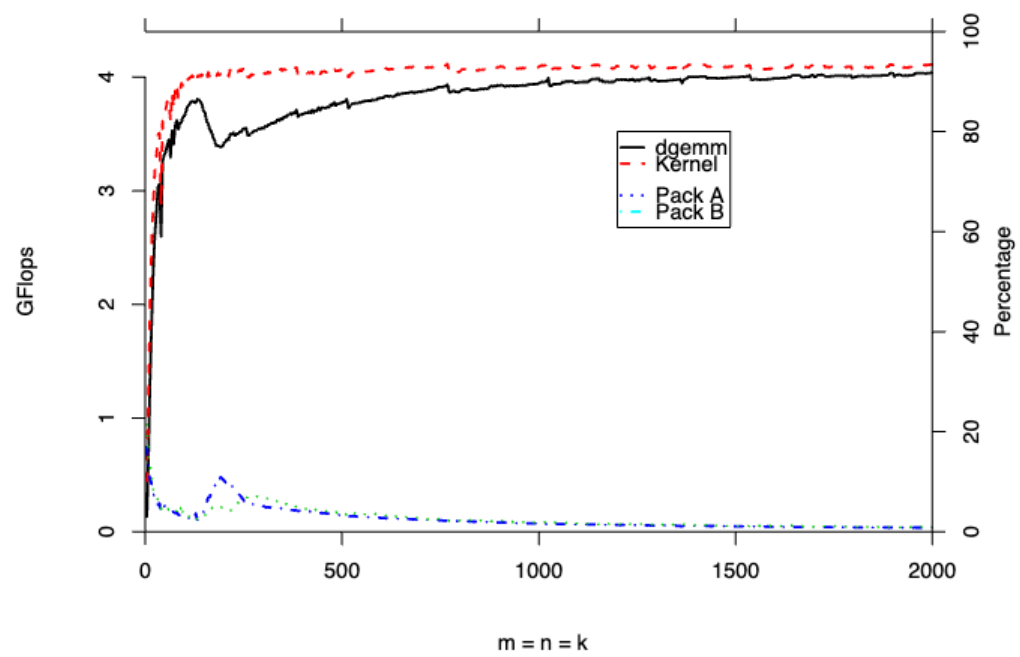


Fig. 16. Opteron (2.2 GHz).

Performance for this architecture is reported in Fig. 16.

Intel Itanium2 processor (1.5 GHz)

The L1 data cache and L1 TLB are inherently ignored by this architecture for floating point numbers. As a result, the Itanium2's L2 and L3 caches perform the role of the L1 and L2 caches of other architectures and only the L2 TLB is relevant. Thus $n_r \geq 4/(2 \times 2.0) = 1.0$. Since there are ample registers available, $m_r \times n_r = 8 \times 8$. While the optimal $k_c = 1\text{K}$ (1K doubles fill half of a page), in

图 15.奔腾 4 Prescott（3.6 GHz）。

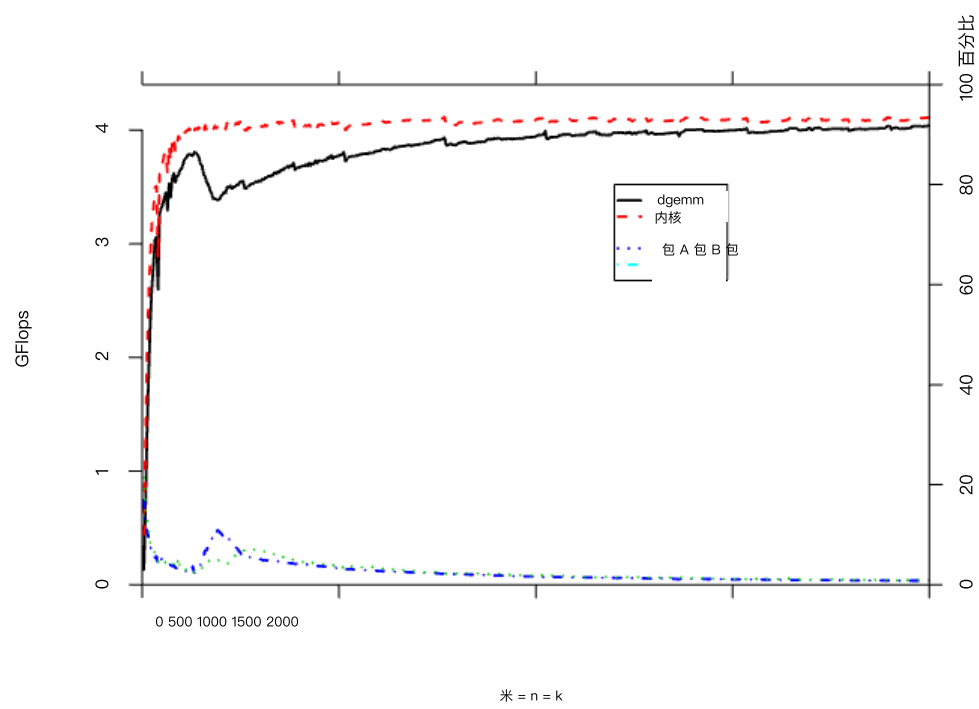


图 16.Opteron （2.2 GHz）。

该架构的性能如图 16 所示。

英特尔 Itanium2 处理器 （1.5 GHz）
对于浮点数，此架构本质上忽略了 L1 数据缓存和 L1 TLB。因此，Itanium2 的 L2 和 L3 缓存执行其他架构的 L1 和 L2 缓存的角色，只有 L2 TLB 是相关的。因此 n⩾ 4/（2 × 2.0）= 1.0。由于有足够的寄存器可用，m× n= 8 × 8。虽然最佳 k= 1K（1K 加倍填充一半页面），但在

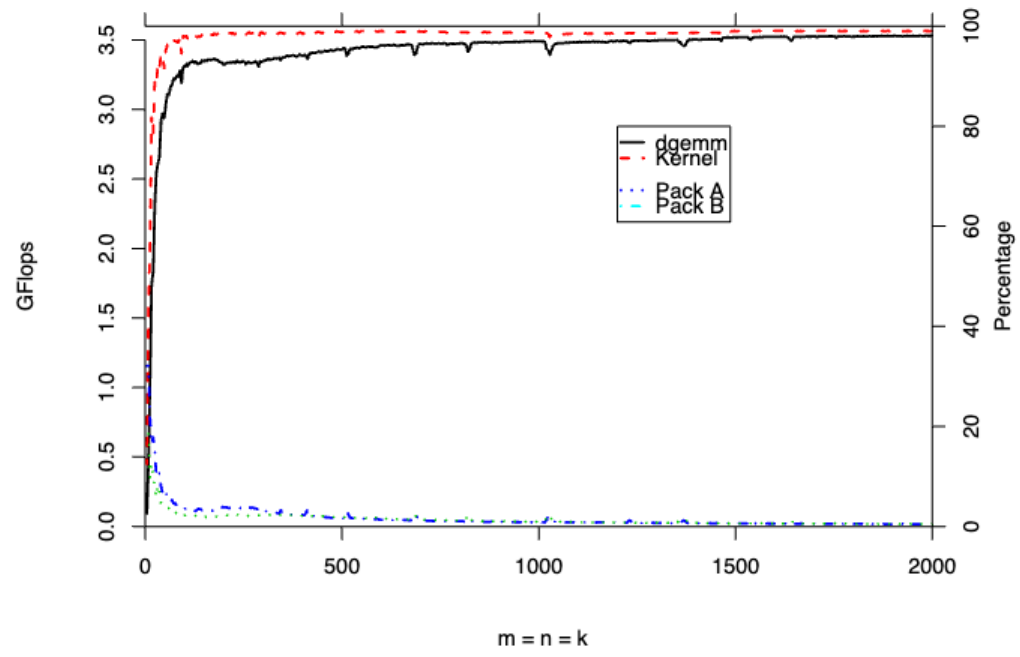


Fig. 17. Itanium2 (1.5 GHz).

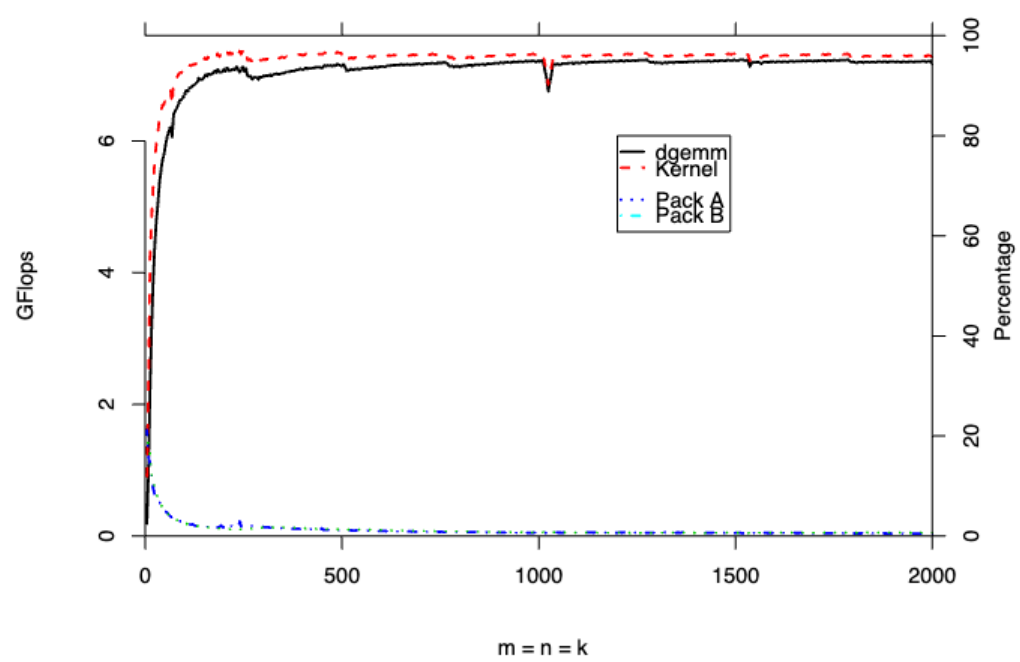


Fig. 18. POWER5 (1.9 GHz).

practice performance is almost as good when $k_c = 128$.

This architecture has many features that makes optimization easy: A very large number of registers , very good bandwidth between the caches and the registers, and an absence of out-of-order execution.

Performance for this architecture is reported in Fig. 17.

IBM POWER5 processor (1.9 GHz)

For this architecture, $n_r \geq 4/(2 \times 0.93) \approx 2.15$ and $m_r \times n_r = 4 \times 4$. This

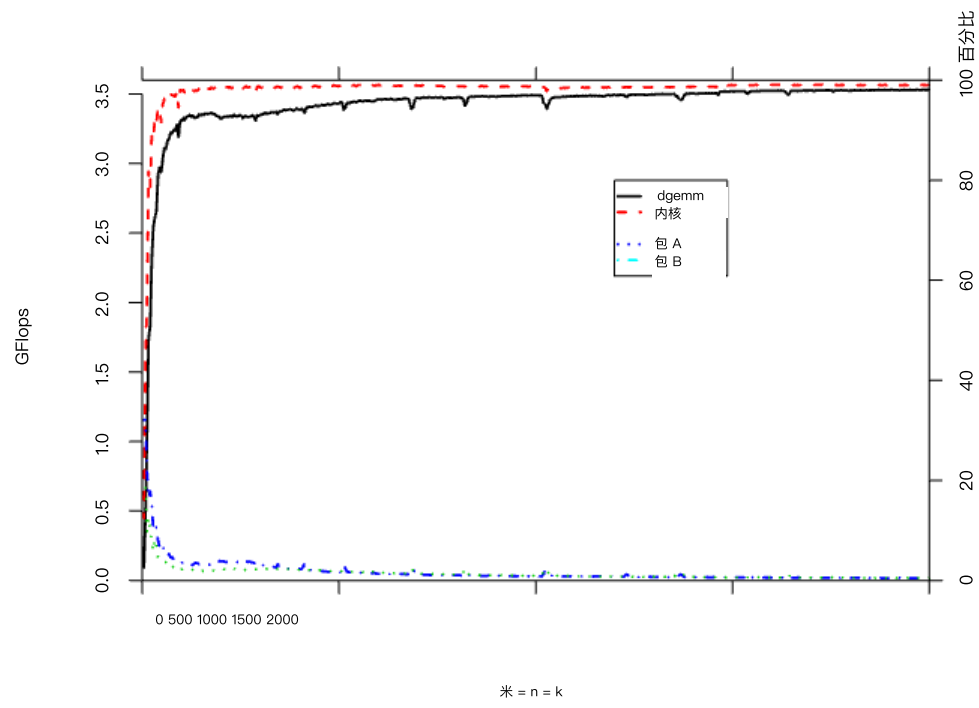


图 17.安腾 2 （1.5 GHz）。

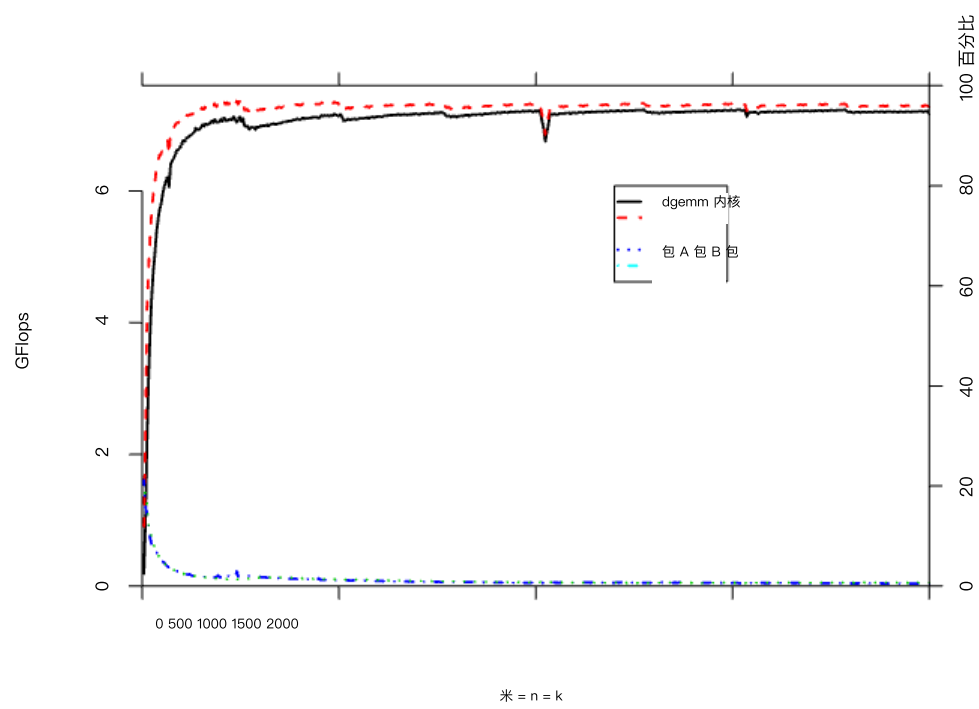


图 18.POWER5 （1.9 GHz）。

当 k= 128 时，练习性能几乎一样好。
该架构具有许多使优化变得容易的功能：非常多的寄存器，缓存和寄存器之间的带宽非常好，并且没有无序执行。

该架构的性能如图 17 所示。

IBM POWER5 处理器 （1.9 GHz）

对于此架构，n⩾ 4/（2 × 0.93） ≈ 2.15 和 m× n= 4 × 4。这

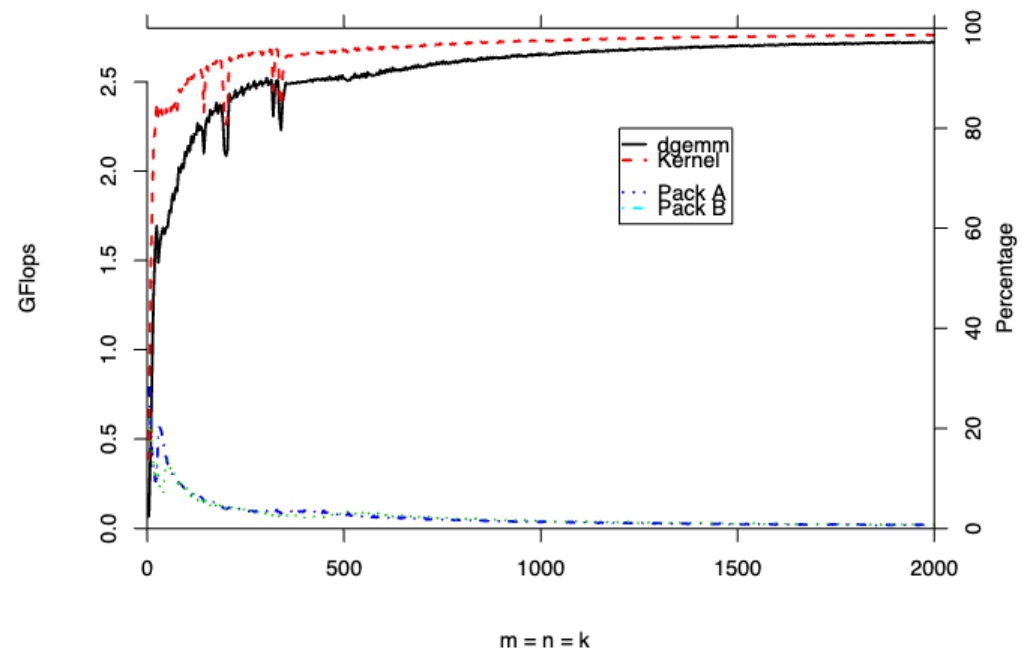


Fig. 19. PPC440 FP2 (700 MHz).

architectures has a D-ERAT (Data cache Effective Real to Address Translation [table]) that acts like an L1 TLB and a TLB that acts like an L2 TLB. Parameter $k_c = 256$ fills half of a page with a column of $\tilde{B}$. By choosing $m_c \times k_c = 256 \times 256$ matrix $\tilde{A}$ fills about a quarter of the memory addressable by the TLB. This is a compromise: The TLB is relatively slow. By keeping the footprint of $\tilde{A}$ at the size that is addressable by the D-ERAT, better performance has been observed.

Performance for this architecture is reported in Fig. 18.

PowerPC440 FP2 processor (700 MHz)

For this architecture, $n_r \geq 4/(2 \times 0.75) \approx 2.7$ and $m_r \times n_r = 8 \times 4$. An added complication for this architecture is that the combined bandwidth required for moving elements of $\tilde{B}$ and $\tilde{A}$ from the L1 and L2 caches to the registers saturates the total bandwidth. This means that the loading of elements of C into the registers cannot be overlapped with computation, which in turn means that k_c should be taken to be very large in order to amortize this exposed cost over as much computation as possible. The choice $m_c \times n_c = 128 \times 3K$ fills 3/4 of the L2 cache.

Optimizing for this architecture is made difficult by lack of bandwidth to the caches, an L1 cache that is FIFO (First-In-First-Out) and out-of-order execution of instructions. The addressable space of the TLB is large due to the large page size.

It should be noted that techniques similar to those discussed in this paper were used by IBM to implement their matrix multiply [Bachega et al. 2004] for this architecture.

Performance for this architecture is reported in Fig. 19.

Core 2 Woodcrest (2.66 GHz) processsor

At the time of the final revision of this paper, the Core 2 Woodcrest was recently released and thus performance numbers for this architecture are particularly interesting.

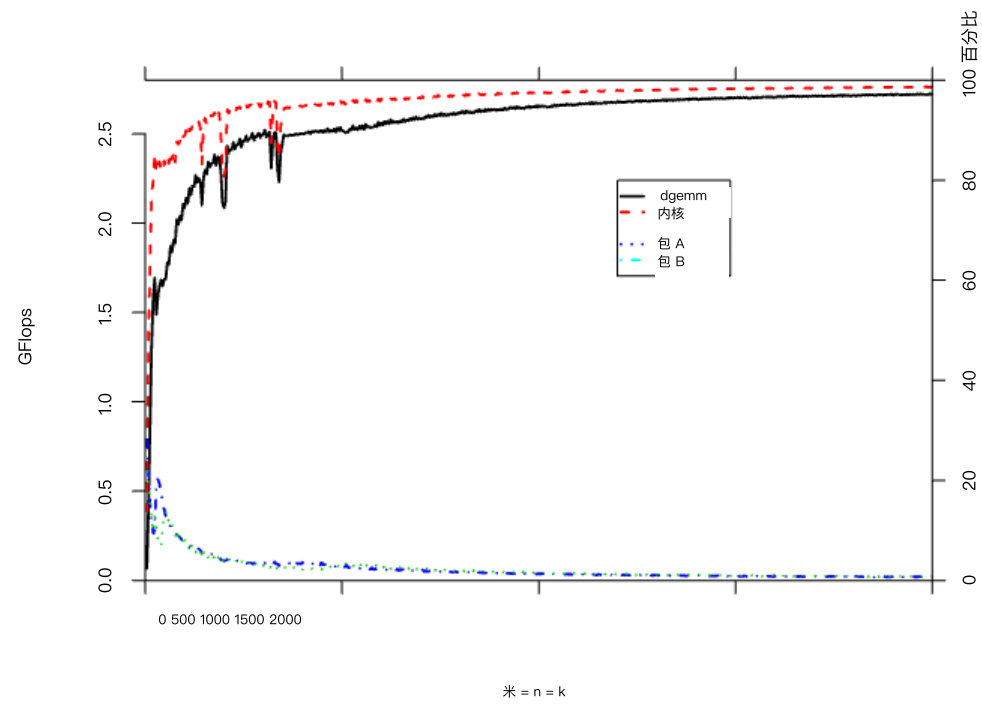


图 19.PPC440 FP2 （700 MHz）。

架构有一个 D-ERAT（数据缓存有效实数到地址转换 [表]），其作用类似于 L1 TLB 和一个 TLB 类似于 L2 TLB。参数 k= 256 用 B 列填充一半页面。通过选择 m× k= 256 × 256 矩阵 ˜A 填充了 TLB 可寻址内存的大约四分之一。这是一个折衷方案：TLB 相对较慢。通过将 ˜A 的封装保持在 D-ERAT 可寻址的大小，可以观察到更好的性能。

该架构的性能如图 18 所示。

PowerPC440 FP2 处理器 （700 MHz）
对于此体系结构，n≥ 4/（2×0.75） ≈ 2.7 和 m×n= 8×4。该架构的另一个复杂性是，将“B”和“A”元素从 L1 和 L2 缓存移动到寄存器所需的组合带宽使总带宽饱和。这意味着将 C 元素加载到寄存器中不能与计算重叠，这反过来又意味着 k 应该被认为非常大，以便在尽可能多的计算中摊销这种暴露的成本。选择 m× n= 128 × 3K 填充 3/4 的 L2 缓存。
由于缓存带宽不足、先进先出（先进先出）的 L1 缓存以及指令的无序执行，因此很难针对这种架构进行优化。由于页面大小较大，TLB 的可寻址空间很大。
应该注意的是，IBM 使用了类似于本文中讨论的技术来实现该架构的矩阵乘法 [Bachega 等人，2004 年]。

该架构的性能如图 19 所示。

Core 2 Woodcrest （2.66 GHz） 进程或
在本文最终修订时，Core 2 Woodcrest 最近发布，因此该架构的性能数据特别有趣。

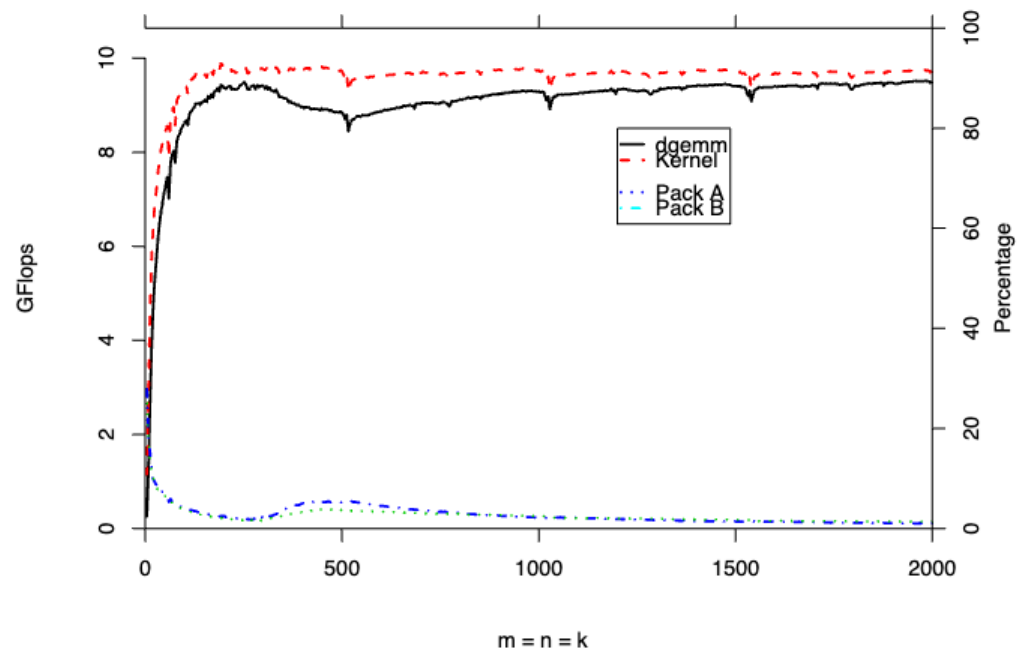


Fig. 20. Core 2 Woodcrest (2.66 GHz)

For this architecture, $n_r \geq 4/(2 \times 1.0) = 2$ and $m_r \times n_r = 4 \times 4$. As for the Prescott architecture, sixteen registers that can hold two double precision numbers each are available, half of which are used to store the $m_r \times n_r$ entries of C. The footprint of matrix $\tilde{A}$ equals that of the memory covered by the TLB.

Performance for this architecture is reported in Fig. 20.

8. CONCLUSION

We have given a systematic analysis of the high-level issues that affect the design of high-performance matrix multiplication. The insights were incorporated in an implementation that attains extremely high performance on a variety of architectures.

Almost all routines that are currently part of LAPACK [Anderson et al. 1999] perform the bulk of computation in GEPP, GEMP, or GEPM operations. Similarly, the important Basic Linear Algebra Subprograms (BLAS) kernels can be cast in terms of these three special cases of GEMM [Kågström et al. 1998]. Our recent research related to the FLAME project shows how for almost all of these routines there are algorithmic variants that cast the bulk of computation in terms of GEPP [Gunnels et al. 2001; Bientinesi et al. 005a; Bientinesi et al. 005b; Low et al. 2005; Quintana et al. 2001]. These alternative algorithmic variants will then attain very good performance when interfaced with matrix multiplication routines that are implemented based on the insights in this paper.

One operation that cannot be recast in terms of mostly GEPP is the QR factorization. For this factorization, about half the computation can be cast in terms of GEPP while the other half inherently requires either the GEMP or the GEPM operation. Moreover, the panel must inherently be narrow since the wider the panel, the more extra computation must be performed. This suggests that further research into the high-performance implementation of these special cases of GEMM is

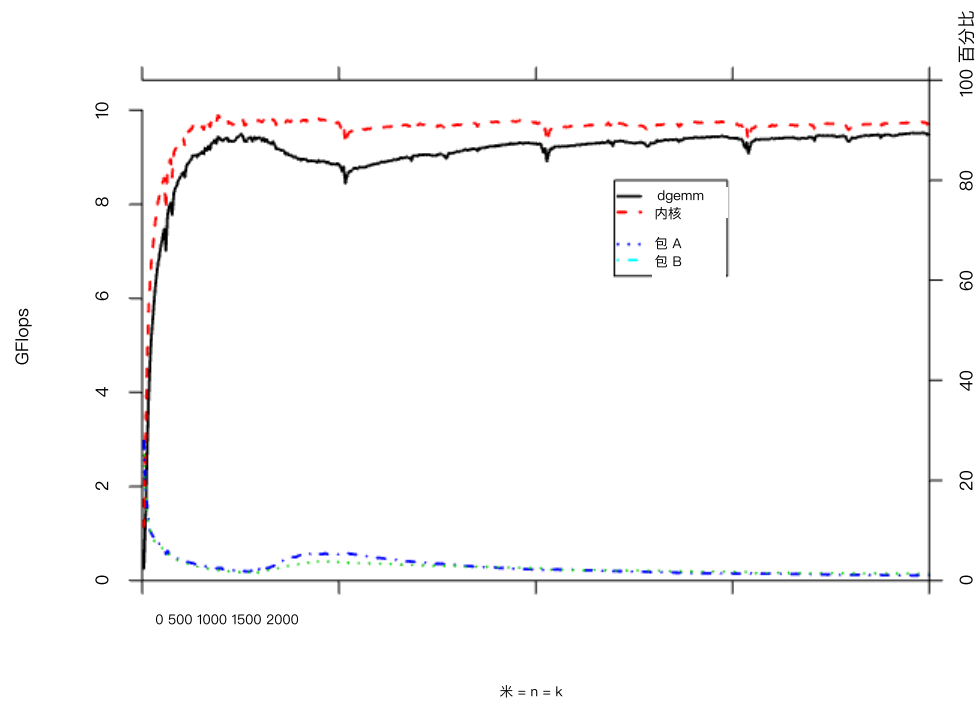


图 20.Core 2 Woodcrest （2.66 GHz）

对于此架构，n⩾ 4/（2 × 1.0） = 2 和 m× n= 4 × 4。至于 Prescott 架构，有 16 个寄存器可用，每个寄存器可以容纳两个双精度数字，其中一半用于存储 C 的 m× 个条目。矩阵 ˜A 的占用空间等于 TLB 覆盖的内存的占用空间。

该架构的性能如图 20 所示。

8. 结论

我们系统分析了影响高性能矩阵乘法设计的高层次问题。这些见解被纳入一个实现中，该实现在各种架构上实现了极高的性能。

目前属于 LAPACK [Anderson 等人，1999 年] 的几乎所有例程都在 gepp、gemp 或 gepm 作中执行大部分计算。同样，重要的基本线性代数子程序 （BLAS） 内核可以根据 gemm 的这三种特殊情况进行铸造 [K˚agstr öm et al. 1998]。我们最近与 FLAME 项目相关的研究表明，对于几乎所有这些例程，都有算法变体将大部分计算转换为 gepp [Gunnels 等人，2001 年;Bientinesi 等人，005a;Bientinesi 等人，005b;Low 等人，2005 年;Quintana 等人，2001 年]。当与基于本文见解实现的矩阵乘法例程交互时，这些替代算法变体将获得非常好的性能。

就大部分 gepp 而言，一个无法重铸的作是 QR 分解。对于这种因式分解，大约一半的计算可以用 gepp 进行转换，而另一半本质上需要 gemp 或 gepm 作。此外，面板本身必须很窄，因为面板越宽，必须执行的额外计算就越多。这表明对 gemm 这些特殊情况的高性能实现的进一步研究是

warranted.

Availability

The source code for the discussed implementations is available from
`http://www.tacc.utexas.edu/resources/software` .

Acknowledgments

We would like to thank Victor Eijkhout, John Gunnels, Gregorio Quintana, and Field Van Zee for comments on drafts of this paper, as well as the anonymous referees.

This research was sponsored in part by NSF grants ACI-0305163, CCF-0342369, and CCF-0540926, and by Lawrence Livermore National Laboratory project grant B546489. We gratefully acknowledge equipment donations by Dell, Linux Networx, Virginia Tech, and Hewlett-Packard. Access to additional equipment was arranged by the Texas Advanced Computing Center and Lawrence Livermore National Laboratory.

REFERENCES

Agarwal, R., Gustavson, F., and Zubair, M. 1994. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM Journal of Research and Development 38,* 5 (Sept.).

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Croz, J. D., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. 1999. *LAPACK Users' Guide*, Third Edition ed. SIAM Press.

Bachega, L., Chatterjee, S., Dockser, K. A., Gunnels, J. A., Gupta, M., Gustavson, F. G., Lapkowski, C. A., Liu, G. K., Mendell, M. P., Wait, C. D., and Ward, T. J. C. 2004. A high-performance simd floating point unit for bluegene/l: Architecture, compilation, and algorithm design. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, Washington, DC, USA, 85–96.

Bientinesi, P., Gunnels, J. A., Myers, M. E., Quintana-Ortí, E. S., and van de Geijn, R. A. 2005a. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Soft. 31,* 1 (March), 1–26.

Bientinesi, P., Gunter, B., and van de Geijn, R. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Soft.*. submitted.

Bientinesi, P., Quintana-Ortí, E. S., and van de Geijn, R. A. 2005b. Representing linear algebra algorithms in code: The FLAME APIs. *ACM Trans. Math. Soft. 31,* 1 (March), 27–59.

Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft. 16,* 1 (March), 1–17.

Goto, K. 2005. `www.tacc.utexas.edu/resources/software/`.

Goto, K. and van de Geijn, R. 2006. High-performance implementation of the level-3 BLAS. FLAME Working Note #20 TR-2006-23, The University of Texas at Austin, Department of Computer Sciences.

Goto, K. and van de Geijn, R. A. 2002. On reducing TLB misses in matrix multiplication. Tech. Rep. CS-TR-02-55, Department of Computer Sciences, The University of Texas at Austin.

Gunnels, J., Gustavson, F., Henry, G., and van de Geijn, R. A. 2005. *A Family of High-Performance Matrix Multiplication Algrorithms*. LNCS 3732. Springer-Verlag, 256–265.

Gunnels, J. A., Gustavson, F. G., Henry, G. M., and van de Geijn, R. A. 2001. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software 27,* 4 (December), 422–455.

必要。

可用性

所讨论的实现的源代码可从
http://www.tacc.utexas.edu/resources/software .

确认

我们要感谢 Victor Eijkhout、John Gunnels、Gregorio Quintana 和 Field Van Zee 对本文草稿的评论，以及匿名审稿人。
这项研究部分由 NSF 拨款 ACI-0305163、CCF-0342369 和 CCF-0540926 以及劳伦斯利弗莫尔国家实验室项目拨款 B546489 赞助。我们衷心感谢戴尔、Linux Networx、弗吉尼亚理工大学和惠普公司捐赠的设备。德克萨斯高级计算中心和劳伦斯利弗莫尔国家实验室安排了对额外设备的使用。

引用

Agarwal， R.、Gustavson， F. 和 Zubair， M. 1994。利用 POWER2 的函数并行性设计高性能数值算法。IBM 研究与发展杂志 38， 5（9 月）。

Anderson， E.、Bai， Z.、Bischof， C.、Blackford， S.、Demmel， J.、Dongarra， J.、Croz， JD、Greenbaum， A.、Hammarling， S.、McKenney， A. 和 Sorensen， D. 1999。LAPACK 用户指南，第三版，SIAM Press. Bachega， L.、Chatterjee， S.、Dockser， KA、Gunnels， JA、Gupta， M.、Gustavson， FG、Lapkowski， CA、Liu， GK、Mendell， MP、Wait， CD 和 Ward， TJC 2004。

一种用于 bluegene/l 的高性能 simd 浮点单元：架构、编译和算法设计.在 PACT '04：第 13 届并行架构和编译技术国际会议论文集。IEEE 计算机学会，美国华盛顿特区，85–96。

Bientinesi， P.、Gunnels， JA、Myers， ME、Quintana-Ort'ı， ES 和 van de Geijn， RA 2005 年 a。推导密集线性代数算法的科学。ACM Trans. Math. 软。31， 1 （三月）， 1–26.

Bientinesi， P.、Gunter， B. 和 van de Geijn， R. 与对称正定矩阵反演相关的算法系列。ACM Trans. Math. Soft..提交。

Bientinesi， P.、Quintana-Ort'ı， ES 和 van de Geijn， RA 2005b。在代码中表示线性代数算法：FLAME API。ACM Trans. Math. 软。31， 1（三月），27–59。

Dongarra， JJ、Du Croz， J.、Hammarling， S. 和 Duff， I. 1990。一组 3 级基本线性代数子程序。ACM Trans. Math. 软。16， 1 （三月）， 1–17.

后藤，K. 2005。www.tacc.utexas.edu/resources/software/。

Goto， K. 和 van de Geijn， R. 2006。3 级 BLAS 的高性能实现。FLAME 工作说明 #20 TR-2006-23，德克萨斯大学奥斯汀分校计算机科学系。

Goto， K. 和 van de Geijn， RA 2002。减少矩阵乘法中的 TLB 未命中。技术。Rep. CS-TR-02-55，德克萨斯大学奥斯汀分校计算机科学系。

Gunnels， J.、Gustavson， F.、Henry， G. 和 van de Geijn， RA 2005。高性能矩阵乘法算法系列。LNCS 3732。施普林格出版社，256–265。

Gunnels， JA、Gustavson， FG、Henry， GM 和 van de Geijn， RA 2001。火焰：为了-MAL 线性代数方法环境。ACM 数学软件汇刊 27， 4（12 月），422—455。

Gunnels, J. A., Henry, G. M., and van de Geijn, R. A. 2001. A family of high-performance matrix multiplication algorithms. In *Computational Science - ICCS 2001, Part I*, V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renner, and C. K. Tan, Eds. Lecture Notes in Computer Science 2073. Springer-Verlag, 51–60.

Kågström, B., Ling, P., and Loan, C. V. 1998. GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Soft. 24,* 3, 268–302.

Low, T. M., van de Geijn, R., and Zee, F. V. 2005. Extracting SMP parallelism for dense linear algebra algorithms from high-level specifications. In *Proceedings of PPoPP'05.*

Quintana, E. S., Quintana, G., Sun, X., and van de Geijn, R. 2001. A note on parallel matrix inversion. *SIAM J. Sci. Comput. 22,* 5, 1762–1771.

Strazdins, P. E. 1998. Transporting distributed BLAS to the Fujitsu AP3000 and VPP-300. In *Proceedings of the Eighth Parallel Computing Workshop (PCW'98).* 69–76.

Whaley, R. C. and Dongarra, J. J. 1998. Automatically tuned linear algebra software. In *Proceedings of SC'98.*

Whaley, R. C., Petitet, A., and Dongarra, J. J. 2001. Automated empirical optimization of software and the ATLAS project. *Parallel Computing 27,* 1–2, 3–35.

Received Month Year; revised Month Year; accepted Month Year

Gunnels， JA、Henry， GM 和 van de Geijn， RA 2001。高性能系列
矩阵乘法算法。在计算科学 – ICCS 2001，第一部分，VN Alexandrov、JJ Dongarra、BA Juliano、RS Renner 和 CK Tan 编辑计算机科学讲义 2073。施普林格–出版社，51–60。

K°agstr öm， B.、Ling， P. 和 Loan， CV 1998。基于 GEMM 的 3 级 BLAS：高性能模型实现和性能评估基准。ACM Trans. Math. 软。24, 3, 268—302.

Low， TM、van de Geijn， R. 和 Zee， FV 2005。从高级规范中提取密集线性代数算法的 SMP 并行性。在 PPoPP'05 的会议记录中。

Quintana， E. S.、Quintana， G.、Sun， X. 和 van de Geijn， R. 2001。关于平行矩阵反演的说明。暹罗 J. 科学计算。22, 5, 1762—1771.

斯特拉兹丁斯，PE 1998。将分布式 BLAS 传输到 Fujitsu AP3000 和 VPP–300。第八届并行计算研讨会（PCW'98） 论文集。69—76.

Whaley， RC 和 Dongarra， JJ 1998。自动调谐的线性代数软件。在 SC'98 的会议记录中。

Whaley， RC、Petitet， A. 和 Donggarra， JJ 2001。软件和 ATLAS 项目的自动经验优化。并行计算 27， 1—2，3—35。

收到月份年份;修订后的月份年份;接受的月份 年份