国防科技大学

BLISlab: 一个 GEMM 优化的实验环境

Jianyu Huang & Junbo Zhang

October 13, 2024

1 简介

Figure 1.1: European swallow.

矩阵乘法是一项对科学计算非常重要的基本运算，并且越来越多地用于机器学习。这是一个足够简单的概念，可以在典型的高中代数课程中引入，但在实践中足够重要，以至于它在计算机上的实现仍然是一个活跃的研究主题。本说明描述了一组使用此操作的练习，以说明如何在具有分层内存 (多个缓存) 的现代 CPU 上实现高性能。它通过公开一个模拟 BLIS 中实现的简化“沙箱”，以类似 BLAS 的库实例化软件 (BLIS) 框架为基础的见解来实现这一点。因此，它也成为 BLIS 优化的“众包”工具。我们将这组练习称为 BLISlab。

2 介绍

矩阵矩阵乘法 (Gemm) 经常用作一个简单的示例，通过它可以提高对如何在现代处理器上优化代码的认识。原因是该操作描述简单，难以完全优化，并且具有实际重要性。在本文档中，我们将向读者介绍当前最快的 CPU 架构实现背后的技术。

2.1 基本线性代数子程序 (BLAS)

基本线性代数子程序 (BLAS) 构成了一组线性代数运算的接口，在此基础上构建了更高级别的线性代数库，例如 LAPACK 和 libflame。其理念是，如果有人针对给定架构优化 BLAS ，那么根据对 BLAS 的调用编写的所有应用程序和库都将从此类优化中受益。

BLAS 分为三组：1 级 BLAS (向量-向量运算)、2 级 BLAS (矩阵-向量运算) 和 3 级 BLAS (矩阵-矩阵运算)。最后一组的好处是，如果所有矩阵操作数的大小都是 $n \times n$，则使用 O(n^2) 数据执行 O(n^3) 浮点运算，以便在内存层 (主内存、缓存和寄存器) 之间移动数据的成本可以通过多次计算分摊。因此，如果仔细实施这些操作，原则上可以实现高性能。

2.2 矩阵-矩阵乘法

特别是，BLAS 支持使用双精度浮点数的 Gemm 运算，可以通过适当选择 transa 和 transb 参数来实现（使用 Fortran 调用）。

$$dgemm(transa, transb, m, n, kalpha, A, lda, B, ldb, beta, C, ldc) \tag{2.1}$$

计算下式：

$$C := \alpha AB + \beta C; \quad C := \alpha A^T B + \beta C; \quad C := \alpha AB^T + \beta C; \quad \text{or} C := \alpha A^T B^T + \beta C.$$

这里 C 是 $m \times n$，k 是“第三维”。本文档稍后将介绍参数 *dla*、*dlb* 和 *dlc*。

$$C := AB + C$$

其中 C 是 $m \times n$，A 是 $m \times k$，B 是 $k \times n$。如果了解如何优化 dgemm 的这种特殊情况，那么可以轻松地将此知识扩展到所有 3 级 BLAS 功能。

2.2.1 高效实现

高性能实现是如此的复杂，以至于 BLAS 的实现，特别是 Gemm 的实现，通常是由为硬件供应商开发数字库的默默无闻的专家来完成的，例如作为 IBM 的 ESSL、英特尔的 MKL、Cray 的 LibSci 和 AMD 的 ACML 库的一部分。这些库通常 (至少是部分) 是用汇编代码编写的，并且高度专用于特定的处理器。

一篇关键的论文展示了“算法和体系结构”方法如何携手设计体系结构、编译器和算法，从而允许用 IBM Power 体系结构的高级语言 (Fortan) 编写 BLAS，并解释了在这些处理

器上实现高性能的复杂性。PHiPAC (Portable High Performance ANSI C) 项目随后提供了用 C 编写高性能代码的指导方针，并建议如何自动生成和调整以这种方式编写的 Gemm。自动调优的线性代数软件 (ATLAS) 建立在这些见解之上，并使自动调优和自动生成 BLAS 库成为主流。

作为本文档的一部分，我们讨论有关该主题的最新论文，包括介绍实现 Gemm 的 Goto 方法的论文和该方法的 BLIS 重构，以及其他更直接相关的论文。

THOUGHTS - NOTES: **Fusce varius orci ac magna dapibus porttitor. In tempor leo a neque bibendum sollicitudin. Nulla pretium fermentum nisi, eget sodales magna facilisis eu. Praesent aliquet nulla ut bibendum lacinia. Donec vel mauris vulputate, commodo ligula ut, egestas orci. Suspendisse commodo odio sed hendrerit lobortis. Donec finibus eros erat, vel ornare enim mattis et.**

3 基础知识

3.1 简单矩阵-矩阵乘法

在我们的讨论中，我们将考虑计算：

$$C := AB + C$$

其中 A、B 和 C 分别是 $m \times k$、$k \times n$、$m \times n$ 矩阵。让

$$A = \begin{pmatrix} a_{0,0} & \cdots & a_{0,k-1} \\ \vdots & & \vdots \\ a_{m-1,0} & \cdots & a_{m-1,k-1} \end{pmatrix}, B = \begin{pmatrix} b_{0,0} & \cdots & b_{0,n-1} \\ \vdots & & \vdots \\ b_{k-1,0} & \cdots & b_{k-1,n-1} \end{pmatrix}, \text{and} C = \begin{pmatrix} c_{0,0} & \cdots & c_{0,n-1} \\ \vdots & & \vdots \\ c_{m-1,0} & \cdots & c_{m-1,n-1} \end{pmatrix}$$

$C := AB + C$ 实际上就等价于计算：

$$c_{i,j} := \sum_{p=0}^{k-1} a_{i,p} b_{p,j} + c_{i,j}$$

如果 A、B 和 C 存储在二维数组 A、B 和 C 中，则使用以下伪代码计算 $C := AB + C$ 分别计算一个乘法和一个加法，计算需要 2mnk 次浮点运算（flops）。

```
for i=0:m-1
   for j=0:n-1
      for p=0:k-1
         C( i,j ) := A( i,p ) * B( p,j ) + C( i,j )
      endfor
   endfor
endfor
```

3.2 设置

为了让你高效地学习如何高效地计算，我们首先为您提供了大量的基础设施。我们将子目录“step1”进行了结构化，使其类似于一个实现了真实库的项目。对于我们的目的来说，这种结构可能过于复杂，但如何结构化软件项目是一项有用的技能。下面是文件介绍：README 是一个文件，用于描述目录的内容以及如何编译和执行代码。

sourceme.sh 是用于配置环境变量的文件。在该文件中：

BLISLAB USE INTEL，表明您使用的是 Intel 编译器（true）还是 GNU 编译器（false）。

BLISLAB USE BLAS 表示您的参考 dgemm 是否使用了外部 BLAS 库实现（如果在您的机器上安装了此类 BLAS 库，则为真），或者简单的三重循环实现（为假）。

COMPILER OPT LEVEL 为您的 GNU 或 Intel 编译器设置优化级别（O0、O1、O2、O3）。（请注意，例如，O3 由大写字母“O”和数字“3”组成。

OMP _NUM _THREADS 和 BLISLAB_IC _NT 设置用于代码并行版本的线程数。对于步骤 1，您将它们都设置为 1。

dgemm 是实现 dgemm 的例程所在的子目录。在这其中：

bl dgemm ref.c 包含例程 dgemm ref，它是 dgemm 的简单实现，您将使用它来检查实现的正确性。如果 BLISLAB USE BLAS = false。

my dgemm.c 包含例程 dgemm，它最初是 dgemm 的简单实现，您将作为掌握如何优化 gemm 的第一步进行优化。

bl dgemm util.c 包含实用程序例程，稍后将会派上用场。

include 此目录包含具有各种宏定义和其他标头信息的包含文件。

lib 此目录将包含由您实现的源文件（libblislab.so 和 libblislab.a）生成的库。您还可以在此目录中安装参考库（例如 OpenBLAS）以比较您的性能。

test 此目录包含各种实现的“test drivers”和正确性/性能检查脚本。

test bl dgemm.c 包含用于测试例程 bl dgemm 的“test driver”。

test bl dgemm.x 是测试 bl dgemm.c 的可执行文件。

run bl dgemm.sh 包含用于收集性能结果的 bash 脚本。

tacc run bl dgemm.sh 包含一个 SLURM 脚本，以便您 (可选地) 将作业提交到德克萨斯州高级计算中心 (TACC) 机器，如果您在那里有一个帐户。

3.3 开始！

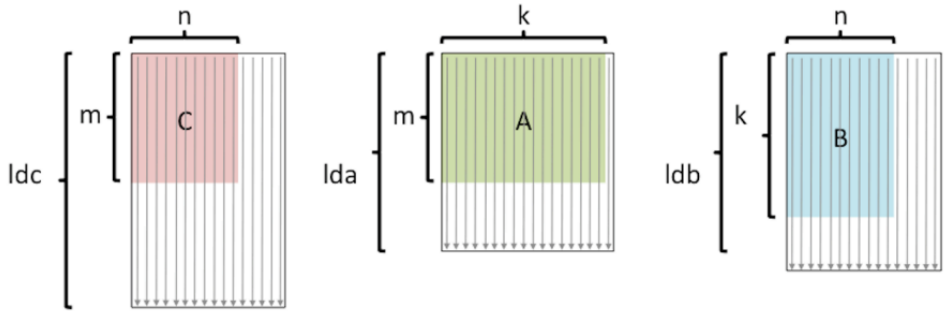
我们希望你从 my dgemm.c 中的实现开始，并通过应用各种标准优化技术对其进行优化。该文件中的初始实现是具有图 2 中给出的三个循环的直接实现。首先要注意的是二维数组是如何按照所谓的列主顺序映射到内存的。这样选择的原因是，原始 BLAS 假设数组的列存储为主，因为接口首先是面向 Fortran 用户的。检查下式：

$$C(i,j) += A(i,p) * B(p,j);$$

我们注意到，每个操作数都是一个宏。在文件的前面考虑:

$$\#define \quad C(i,j) \quad C[(j)*ldc+(i)]$$

地址 C 处的线性数组用于存储元素 $C_{i,j}$，因此 i,j 元素被映射到位置 j * ldc + i。查看该映射的方法是，C 的列都连续存储。然而，将矩阵 C 视为嵌入在具有 ldc 行的较大数组中，因此访问一行意味着以步长 ldc 遍历数组 C。二维数组 C 的前导维数通常指这个较大数组的行维数，即变量 ldc (C 的前导维数)。下面的图说明了这三个矩阵的情况:



3.3.1 配置默认实现

默认情况下，该练习编译并链接英特尔的 icc 编译器，该编译器将对代码应用编译器优化（O3 级）。您需要通过执行以下命令来设置环境变量：$ *source sourceme.sh*

在终端中，您将看到输出:

BLISLAB_USE_INTEL = true

COMPILER_OPT_LEVEL = O3

3.3.2 编译、执行和收集结果

如果您无法访问英特尔编译器（icc），请阅读 2.3.2 和 2.3.3 小节，然后继续阅读 2.3.5 小节。

你可以编译、执行代码并通过执行来收集性能结果:

make clean

make

cd test

./run_bl_dgemm.sh

在子目录 step1 中。您将看到性能结果输出：电脑上自己运行

可在 run bl dgemm.sh 中更改采样块大小。

3.3.3 绘制效果图

最后，您可以使用 MATLAB 使用我们的脚本绘制您的性能图表。在 test 子目录中，执行 *./collect_result_step*1 您将获得一个 MATLAB 文件 “step1 result.m”，其中包含性能结果。然后，您可以执行 *bl_dgemm_plot.m* 在 MATLAB 中，然后生成性能图。

3.3.4 对 *GNU* 编译器的更改

由于我们希望您明确了解哪些技巧可以带来高性能，并且由于您中的一些人可能无法访问英特尔编译器，因此您接下来应该改用 GNU C 编译器。为此，您必须编辑 sourceme.sh：$BLISLAB\_USE\_INTEL = false$ 然后，与默认设置类似，您需要通过执行来设置环境变量：

$ *source sourceme.sh*

3.3.5 关闭优化

接下来，我们希望你关闭编译器执行的优化。这有三个目的: 首先，这意味着你必须明确地执行优化，这将允许你了解架构和算法如何相互作用。其次，优化编译器很可能会试图“撤销”你明确想要完成的事情。第三，你在代码中设置的技巧越多，编译器就越难找出如何优化的方法。您需要先编辑 sourceme.sh:

$COMPILER\_OPT\_LEVEL = O0$

然后，与默认设置类似，您需要通过执行来设置环境变量：

$ *source sourceme.sh*

3.4 基本技术

在本小节中，我们将描述该行业的一些基本技巧。

3.4.1 使用指针

现在优化已经被关闭，矩阵元素存在的地址的计算被明确地暴露了出来。（一个优化编译器会消除这种开销。）你希望做的是修改 my_gemm.c 文件中的实现，使其改用指针。在你这么做之前，你可能需要备份原始的 my_gemm.c 文件，以防你需要从头开始。实际上，在每一步中，你可能都需要将之前的实现备份到单独的文件中。

以下是基本思路。假设我们想将矩阵 C 的所有元素设置为零。一个基本的循环，类似于你在 my_gemm.c 中找到的代码，可能如下所示：

```
for(i=0;i<m;i++){
    for(j=0;j<n;j++){
        c[i,j]=0.0;
    }
}
```

使用指针，我们可以将其实现为:

```
double *cp;

for(j=0; j<n; j++){
    cp = &C[j*idc];
    for(i=0; i<m; i++){
        *cp++ = 0.0;
```

```
    }
}
```

请注意，我们故意交换了循环的顺序，以便向前移动指针使我们沿着 C 的列向下移动。

3.4.2 循环展开

每次通过内层循环时更新循环索引 i 和指针 cp 会产生相当大的开销。因此，编译器将执行循环展开。使用展开因子 4，我们将 C 设置为 0 的简单循环变为

```
double* cp;
int j, i;

for (j = 0; j < n; j++) {
    cp = &C[j * ldc];
    for (i = 0; i < m; i += 4) {
        *(cp + 0) = 0.0;
        *(cp + 1) = 0.0;
        *(cp + 2) = 0.0;
        *(cp + 3) = 0.0;
        cp += 4;
    }
}
```

重要地:

- i 和 cp 现在每 4 次迭代只更新一次。
- *(cp+0) 使用一种称为间接寻址的机器指令，这比使用 *(cp+k) 计算要高效得多，其中 k 是一个变量。
- 当它将数据带入缓存时，它将以每次 64 字节的缓存行带入。这意味着，以 64 字节为单位访问连续数据，可以减少在内存层之间移动内存的开销。

请注意，展开时，如果 m 不是 4 的倍数，可能需要处理一个 “边缘”。在这个练习中，只要明智地选择采样块的大小，你无需担心这个边缘，这一点在 2.5 节中再次提到。

3.4.3 注册变量

注意，只有当数据存储在寄存器中时，计算才会发生。编译器将自动转换代码，以便插入将某些数据放入寄存器的中间步骤。我们可以给编译器一个提示，说明将某些数据保存在寄存器中是很好的，如下面的例子所示:

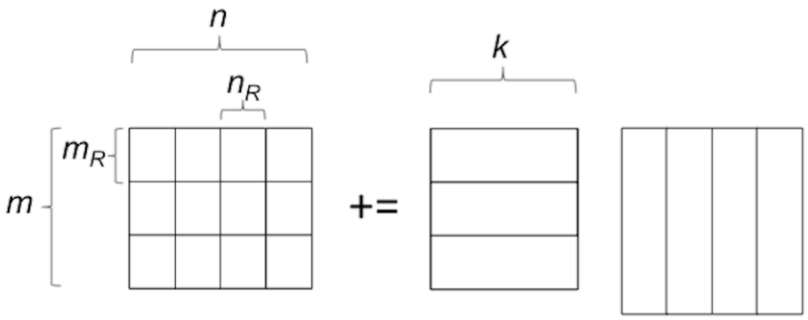
```
double* cp;

for (j = 0; j < n; j++) {
```

```
    cp = &C[j * ldc];
    for (i = 0; i < m; i += 4) {
        register double c0 = 0.0, c1 = 0.0, c2 = 0.0, c3 = 0.0;
        *(cp + 0) = c0;
        *(cp + 1) = c1;
        *(cp + 2) = c2;
        *(cp + 3) = c3;
        cp += 4;
    }
}
```

3.5 一个适度的首要目标

现在，我们要求您使用上面讨论的技术来优化 my_dgemm。现在，只需要担心尝试对较小的矩阵获得更好的性能。特别是下面这张图: 我们希望你做的是编写代码，使 C 的



$mR \times nR$ 块保存在寄存器中。你可以选择 mR 和 nR，但你需要更新文件 include/bl_config.h。这确保了测试驱动程序只尝试这些块大小的倍数的问题大小，因此您不必担心 “边缘”。您会注意到，即使是一个可以放入缓存内存的小矩阵，您的实现也比 MKL 或您可能安装的其他优化 BLAS 库的实现性能差 (很多)。原因是编译器没有使用浮点运算最快的指令。可以通过使用 vector 的内部函数来访问它们，这样就可以在 C 语言中显式地使用它们，也可以通过编写汇编代码来访问它们。现在，我们先不讨论这个。

4 阻塞

4.1 Poorman’s BLAS

本练习的第一步使你意识到，随着基于缓存的架构的出现，实现高效的 Gemm 运算需要对数据在内存层之间移动和与该数据相关的计算成本的摊销给予仔细的关注。为了使管理更加容易，你应该意识到，只需要对相对小的矩阵进行矩阵乘法运算的 “内核” 需要进行高度优化，因为对于较大的矩阵，可以将其分割开来，然后使用这样的内核而不会影响整体性能。这一洞察在文献 [9] 中得到了明确的阐述。

这有时被称作“穷人的 BLAS”，意思是如果只能负担得起优化矩阵-矩阵乘法（使用子矩阵），那么就可以构建 Gemm（以及其他被称为三级 BLAS 的重要矩阵-矩阵运算）。我们稍后会看到，实际上，这样做通常是一个好主意，既考虑到模块化，也考虑到性能。

在上一节中，您已经看到了一个阻塞的示例。

4.2 分块矩阵-矩阵乘法

分块 Gemm 以利用处理器的分层内存的关键是理解当这些矩阵被分块时如何计算 $C := AB + C$。

$$A = \begin{pmatrix} A_{0,0} & \cdots & A_{0,K-1} \\ \vdots & & \vdots \\ A_{M-1,0} & \cdots & A_{M-1,K-1} \end{pmatrix}, B = \begin{pmatrix} B_{0,0} & \cdots & B_{0,N-1} \\ \vdots & & \vdots \\ B_{K-1,0} & \cdots & B_{K-1,N-1} \end{pmatrix}, \text{ and } C = \begin{pmatrix} C_{0,0} & \cdots & C_{0,N-1} \\ \vdots & & \vdots \\ C_{M-1,0} & \cdots & C_{M-1,N-1} \end{pmatrix}.$$

where $C_{i,j}$ is $m_i \times n_j$, $A_{i,p}$ is $m_i \times k_p$, and $B_{p,j}$ is $k_p \times n_j$. Then

$$C_{i,j} := \sum_{p=0}^{K-1} A_{i,p} B_{p,j} + C_{i,j}.$$

4.3 任务

现在我们要求你在 my_dgemm 中实现分块矩阵-矩阵乘法。具体来说，对于小矩阵，你可以获得比大矩阵更好的性能，因为小矩阵可以放在缓存中。将矩阵分块成可以获得更高性能的子矩阵，你会发现即使矩阵更大，最终的实现也可以保持更好的性能。

5 阻塞多级缓存

5.1 实现 Gemm 的 Goto 方法

2000 年左右，Kazushige Goto 彻底改变了 Gemm 在当前 cpu 上的实现方式，他的技术首次发表在论文 [6] 中. 这种方法的进一步“重构”最近在 [16] 中进行了描述.

BLIS 框架的优点是，它将必须高度优化的内核 (可能使用向量内联函数或汇编代码) 简化为微内核。在本节中，我们简要描述该方法的要点。然而，我们强烈建议读者自己熟悉上述两篇论文。

图 3(左) 说明了 Goto 方法为三层缓存 (L1、L2 和 L3) 构建阻塞结构的方式。在 BLIS 框架中，实现正是以这种方式构造的，因此只有底层的微内核需要针对给定的体系结构进行高度优化和定制。在原来的 GotoBLAS 实现 (现在维护为 OpenBLAS[11]) 中，从围绕微内核的第二个循环开始的操作是自定义的。为了获得最佳性能，它帮助所有数据都是连续访问的，这就是为什么在到达微内核之前的某个时刻，数据按箭头所示的顺序打包:

现在，注意图片中 A 的每个块中的每一列都与 B 的相应块中的每一行的每个元素相乘。（我们将这些 A 和 B 的块称为微面板。）这意味着，将 A 的微面板中的一个元素从 L2 缓存中调入所需的时间（即 L2 缓存延迟）可以分摊到 $2n_R$ 次浮点运算上。因此，我们可

以组织计算，使得 A 的微面板通常驻留在 L2 缓存中。实际上，我们还可以做得更好：当使用 A 和 B 的微面板中的一列进行秩 1 更新时，下一列的 A 的微面板可以被加载到寄存器中，从而使计算掩盖了这部分数据移动的成本。由于我们希望将 B 的微面板保留在 L1 缓存中（因为它将被许多 A 的微面板重复使用），这限制了阻塞参数 k_C。

有了这些见解，剩下的部分应该会变得更加清晰。围绕微内核的第一层循环处理的是一个已经被打包并驻留在 L2 缓存中的 A 的块，记作 $\tilde{A}_i$（这是由于计算顺序的设计）。这限制了阻塞参数 m_C。该 A 的块与一个同样被打包以驻留在 L3 缓存中的 B 的块 $\widetilde{B}_p$ 相乘（如果处理器有 L3 缓存的话）。请注意，$\tilde{A}_i$ 的打包成本可以在所有与 $\widetilde{B}_p$ 相关的计算中摊销，而 $\widetilde{B}_p$ 的打包成本则可以在与许多 A 的块 A_i 的计算中摊销。最外层的循环将 B 划分为多个部分，以便每个块 $\widetilde{B}_p$ 都能适应 L3 缓存，或者如果处理器没有 L3 缓存，则限制用于打包 $\widetilde{B}_p$ 的工作空间需求。这限制了阻塞参数 n_C。

有人可能会问，上述方案是否是最佳方案。在 [7] 中，给出了一个理论，表明在理想化的模型下，上述是局部最优的 (在这种意义上，假设数据在层次结构中的某个内存层中，在该层次上提出的阻塞以最优的方式摊销数据移动的成本与下一个内存层)。在 [13] 中给出了一个指导各种阻塞参数选择的理论。

5.2 设置

图 4 演示了子目录 step3 的目录结构。与步骤 1 相比，我们修改/添加了以下目录/文件:
kernels 这个目录包含了适用于各种架构的微内核实现。

- ‘bl_dgemm_ukr.c‘ 提供了一个朴素的 C 语言实现。

- ‘bl_dgemm_int_kernel.c‘ 提供了一个针对 Haswell 架构的 AVX/AVX2 内联汇编（intrinsics）微内核实现。

- ‘bl_dgemm_asm_kernel.c‘ 提供了一个针对 Haswell 架构的 AVX/AVX2 汇编语言微内核实现。

5.3 先进的技术

Intel Intrinsics Guide 以及 Intel ISA Extensions