XMAG:面向SW26010-Pro众核处理器的矩阵乘法代码自动生成系统

严愉程[1)] 马文静[1)] 刘芳芳[1)] 胡力娟[1)] 李 芳[2)]

[1)](中国科学院软件研究所并行软件与计算科学实验室 北京 100190)

[2)](国家并行计算机工程技术研究中心 北京 100190)

摘 要 稠密矩阵乘法在科学与工程计算、人工智能等诸多领域都有广泛应用,并且往往是决定应用性能的关键计算模块。在计算机处理器蓬勃发展的今天,异构众核处理器在高性能计算中占有越来越重要的位置。其中,有相当一部分芯片开始使用片上网络和分布式高速缓存。众核处理器上的矩阵乘法代码生成一直是研究热点,因此,对具有此类体系结构特征的众核处理器开发高效的矩阵乘法生成器成为一项迫切的需求。本文面向此类具有片上网络和分布式高速缓存的处理器,针对各种规模矩阵对访存和计算优化提出的挑战,结合此类众核处理器的特点,提出了一套矩阵乘法代码自动生成和自动调优系统框架:XMAG。要生成众核处理器上优化的矩阵乘法代码,首先,需要根据矩阵大小和硬件配置选择矩阵乘法在众核处理器上的任务划分方案及数据映射方案。其次,需要根据数据规模选择恰当的数据移动和数据交换方式。再次,为了得到更好的性能,在使用传统编译优化技术的基础上,必须设计合理高效的软件流水线。另外,需要编写深度优化的高性能计算核心代码,并根据数据规模进行调优。最后,为了保证正确性和性能,还需要对代码生成系统的一些实现细节进行优化。XMAG以国产申威SW26010-Pro众核处理器为目标平台,设计了swIR作为中间表示,根据SW26010-Pro处理器特点,能够生成不同计算模式的抽象语法树,并经过变换生成多种形式的软件流水线,从而为不同规模和形状的矩阵生成多种版本的高性能矩阵乘法代码。本文使用层次化方法对XMAG进行构建,整个代码生成系统包括计算方案生成,抽象语法树生成和变换,计算核心汇编代码生成,以及自动调优系统。计算方案包括任务划分与数据映射方法,以及数据移动策略。XMAG能够生成具有各种不同任务划分与数据映射方法及移动策略的计算方案。抽象语法树变换能够实现各种软件流水线,从而充分利用SW26010-Pro的体系结构特性,挖掘其性能潜力。而计算核心汇编代码由汇编代码生成器生成。在系统中,我们加入了自动调优机制,能够生成多种多样的矩阵乘法代码。这样,我们便构建了一套生成多种优化方案代码的系统性方法,适用于不同形状和规模的矩阵。通过自动调优,我们选出的代码得到了令人满意的性能。相比SW26010-Pro上手工优化的xMath2.0数学库,XMAG生成的DGEMM代码在不同规模的矩阵上获得了平均2.32倍的加速。我们的代码生成系统为SW26010-Pro处理器上的稠密矩阵计算优化开辟了一条方便有效的路径,也为异构处理器上代码生成系统的构建提供了初步的探索经验。

关键词 矩阵乘法;SW26010-Pro;代码生成;软件流水线;优化

中图法分类号 TP311 **DOI号** 10.11897/SP.J.1016.2026.01287

XMAG: A Matrix Multiplication Automatic Code Generator for SW26010-Pro Many-Core Processors

YAN Yu-Cheng[1)] MA Wen-Jing[1)] LIU Fang-Fang[1)] HU Li-Juan[1)] LI Fang[2)]

[1)](Laboratory of Parallel Software and Computational Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190)

收稿日期:2025-07-11;在线发布日期:2026-03-04。本课题得到国家重点研发计划项目(2023YFB3001703)资助. **严愉程**,硕士,工程师,主要研究领域为高性能计算、高性能代码生成。E-mail:yanyucheng21@mails.ucas.edu.cn。**马文静**(通信作者),博士,副研究员,中国计算机学会(CCF)会员,主要研究领域为高性能计算、高性能代码生成。E-mail:wenjing@iscas.ac.cn。**刘芳芳**(通信作者),博士,正高级工程师,中国计算机学会(CCF)会员,主要研究领域为高性能计算、异构平台并行计算、超级计算机评测软件。E-mail:fangfang@iscas.ac.cn。**胡力娟**,硕士,助理工程师,中国计算机学会(CCF)会员,主要研究领域为高性能计算、异构平台并行计算、BLAS库相关算法。**李 芳**(通信作者),博士,研究员,中国计算机学会(CCF)会员,主要研究方向为高性能计算应用及相关优化技术。E-mail:lifang56@163.com。

[2)](National Research Centre of Parallel Computer Engineering and Technology, Beijing 100190)

Abstract Dense matrix multiplication is widely used in scientific and engineering computation, artificial intelligence, as well as many other areas. And matrix multiplication is often the most performance critical component in the application. In modern days, as the research in processor design is thriving, heterogeneous many-core processors are playing a more and more important role. Among the new processors, quite a few are using on-chip network and distributed fast scratch-pad memory. Since code generation for matrix multiplication on many-core processors has been a hot research topic, generating matrix multiplication code for many-core processors with these architectural features is a highly desired functionality. Therefore, we propose XMAG, a code generation framework for GEMM on many-core processors with on-chip network and high speed distributed cache.

To achieve optimal performance for matrix multiplication on many-core processors, first, it is necessary to select appropriate task partitioning and data mapping strategies based on matrix size and hardware configuration. Second, suitable data movement and data exchange methods should be designed according to the size and shape of the matrices. Third, to achieve better performance, efficient and well-designed software pipelining must be implemented on top of traditional compiler optimization techniques. Additionally, highly optimized high-performance computing kernel code must be written and tuned according to matrix size. Finally, to ensure both correctness and performance, implementation details of the code generation system also require fine-tuning. Targeting the SW26010-Pro processor, we proposed an intermediate representation, swIR. With swIR, we are able to generate a large variety of ASTs according to the architectural features of SW26010-Pro, which can be transformed into many different software pipelines, leading to the generation of multiple versions of high performance code for matrix multiplication on matrices of various sizes and shapes.

We constructed XMAG using a hierarchical approach, and the entire code generation system encompasses computational scheme generation, abstract syntax tree (AST) generation and transformation, assembly code generation for computing kernels, and an automatic tuning system. The computational scheme includes task partitioning and data mapping methods, as well as data movement strategies. XMAG is capable of generating computational schemes with various task partitioning, data mapping, and data movement strategies. AST transformations enable various software pipelining techniques, fully leveraging the architectural features of the SW26010-Pro processor and unlocking its performance potential. The assembly code for computing kernels is generated by an assembly code generator. Within the system, we have integrated an automatic tuning mechanism capable of generating a wide variety of matrix multiplication codes. With the above methodology, we were able to establish a systematic approach to generate diverse code versions, adaptable to different matrix shapes and sizes.

Utilizing an auto-tuning mechanism, the GEMM code we select among the generated versions yields satisfactory performance, which even surpasses the performance of xMath2.0, the hand-optimized library for SW26010-Pro, in a number of tests. The XMAG generated DGEMM code got an average speedup of 2.32 over xMath2.0 on various types of irregular matrices. This system proposed a convenient and effective way to optimize dense matrix multiplication on SW26010-Pro processors, and provides hints on constructing code generation infra-structures on heterogeneous processors.

Keywords matrix multiplication; SW26010-Pro; code generation; software pipeline; optimize

1 引　言

稠密矩阵乘法（即BLAS等函数库中的GEMM函数）在许多高性能计算应用中都处于性能关键位置，广泛使用于科学与工程计算、人工智能等领域。因此，研究人员和工程技术人员一直致力于优化矩阵乘法，在充分利用体系结构特征，使矩阵乘法的性能尽量接近浮点计算理论峰值的基础上，对不同形状和规模的矩阵进行个性化优化。而这种做法，往往导致极大的代码空间，开发者需要耗费大量的时间和精力来对此函数进行开发和调优。因此，构建矩阵乘法代码生成系统逐渐成为研究热点，尤其是面向当今高性能计算中广泛使用的异构众核平台。在各类异构众核系统中，一个正在兴起的趋势是使用片上网络和分布式高速缓存。国产申威众核处理器SW26010-Pro便是一个典型的例子。该处理器的片上网络及特有的核间通信方式、分布式高速缓存设计等，给矩阵乘法优化带来了新的挑战和机遇。因此，面向SW26010-Pro，我们设计了一套矩阵乘法代码生成系统：XMAG。

要充分利用SW26010-Pro的体系结构特征，并搭建面向不同规模矩阵的代码生成系统，我们面临以下挑战：

首先，需要根据矩阵大小和硬件配置选择矩阵乘法在众核处理器上的任务划分方案及数据映射方案。其次，需要根据数据规模选择恰当的数据移动和数据交换方式。再次，为了得到更好的性能，在使用传统编译优化技术的基础上，必须设计合理高效的软件流水线。另外，需要编写深度优化的高性能计算核心代码，并根据数据规模进行调优。最后，为了保证正确性和性能，还需要对代码生成系统的一些实现细节进行优化。

为了应对这些挑战，我们使用层次化方法对XMAG进行构建，整个代码生成系统包括计算方案生成，抽象语法树生成和变换，计算核心汇编代码生成，以及自动调优系统。计算方案包括任务划分与数据映射方法，以及数据移动策略。抽象语法树变换能够实现各种软件流水线，从而充分利用SW26010-Pro的体系结构特性，挖掘其性能潜力。而计算核心汇编代码由汇编代码生成器生成。在系统中，我们加入了自动调优机制，能够生成多种多样的矩阵乘法代码。

本文的贡献如下：

（1）开发了一个完整的代码生成框架，能够为SW26010-Pro处理器上的矩阵乘法生成C语言代码，以及深度优化的计算核心汇编代码。使用此框架，我们能用比手工开发更少的时间，为不同形状的矩阵生成矩阵乘法代码。

（2）XMAG能够生成各种任务划分方式，数据映射方式，及数据移动方式的计算方案。在按照给定计算方案生成的抽象语法树基础上，我们提出了生成多种软件流水线的系统性方法，适用于不同矩阵形状和规模。

（3）在大规模双精度浮点数矩阵上（矩阵为16 384×16 384的方阵）XMAG生成的代码达到了SW26010-Pro单核组理论峰值性能的91.8%，非常接近xMath2.0中DGEMM函数的性能（xMath2.0是专门为SW26010-Pro优化的函数库）。通过自动调优，针对异形矩阵（即矩阵乘法的某一个或两个维度较小），XMAG生成的矩阵乘法代码相比xMath2.0实现了平均2.32倍的加速，最大加速比10.55。

2 背　景

2.1 SW26010-Pro体系结构及带有片上网络的众核处理器

片上网络在当今众核处理器设计中逐渐得到广泛应用。国产申威系列处理器中的SW26010-Pro处理器即是一个典型的例子。SW26010-Pro在继承上一代处理器SW26010基本架构的基础上，增加了许多新的特征。该处理器由6个核组构成。一个核组包括一个管理单元（即MPE，也称为“主核”）和一个由64个计算单元（CPE，也称为“从核”）组成的网格，如图1所示。管理单元是一个完整的处理器核心，具备包括资源管理、通信等在内的所有处理器功能。计算单元则用于执行计算任务。每个计算单元包含一个256 KB的可读写片上内存，称为LDM（本地数据内存），能够提供高速数据访问。LDM可以配置为两部分，一部分作为高速缓存（cache），一部分作为用户可读写的片上内存。例如，如果cache大小设为32 KB，那么用户可以将剩下的224 KB作为可读写片上内存。LDM和主存之间的数据移动可通过DMA通道完成，DMA传输最高能达到46 GB/s带宽。一个核组中的64个CPE能够提供2.304TFLOPS的理论峰值算力。

CPE网格上的64个CPE可通过远程内存访问

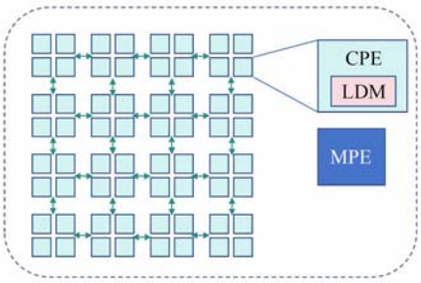

图1 SW26010-Pro处理器基本结构

(RMA)机制,利用片上网络进行通信。从核间通信可以通过广播方式或点对点方式进行。广播方式包括行广播、列广播,和全局广播。DMA和RMA操作可通过调用CRTS(通用运行时系统)库中的函数来实现。从核支持SW64指令集,能够执行512位向量指令。

除了SW26010-Pro,还有一些处理器也采用了此类设计。NVIDIA H100 GPU在现有计算层次的基础上增加了"thread block cluster"层级,部署SM间网络,从而使同一个cluster上的线程块能够共享块中共享缓存内的数据[1]。另外一个例子是Tesla的DOJO,拥有354个核心。它也使用了片上网络来进行核间通信,使得一个核心可以访问另一核心SRAM中的数据[2]。本文所讨论的方法和框架可为上述系统提供代码生成与优化方面的提示和初步探索经验。

2.2 SW26010-Pro处理器上的矩阵乘法

稠密矩阵乘法函数(GEMM)完成以下计算:

$$C=\beta*C+\alpha*A*B \tag{1}$$

这里A,B,C均为矩阵,α,β为标量。其实现可以表示为如算法1所示的嵌套循环。由于SW26010-Pro上的应用程序通常将一个进程映射到一个核组,因此本文讨论的代码生成工作集中于SW26010-Pro处理器上一个核组之内如何实现和优化GEMM函数计算。整个处理器(6个核组)上的矩阵乘法可在对计算任务进行切分的基础上,利用本文所述方法生成的代码实现[3-4]。此外,对使用SW26010-Pro的超级计算机而言,多结点(多进程)矩阵乘法,可利用现有的多进程矩阵乘法优化方法进行优化[5-7],而在进程内采用XMAG生成的高性能计算代码。

算法1. 矩阵乘法基本实现

```
1. for i from 0 to M step 1 do
2.     for j from 0 to N step 1 do
3.         C[i][j] *= β
4.         for k from 0 to K step 1 do
5.             C[i][j] += α*A[i][k] * B[k][j]
6.         end for
7.     end for
8. end for
```

SW26010-Pro上GEMM的优化包括对从核网格上的计算方案进行设计和优化,以及编写高性能计算核心代码[8-10]。而不同形状和规模的矩阵又对并行方案、数据移动方式、参数选择以及计算核心优化提出了更加复杂的要求[3]。

由于GEMM各维度的规模构成的空间极大,要为所有矩阵形状和规模手工编写精细优化的代码是极其繁重的工作。为了解决这个问题,我们提出了一套面向SW26010-Pro众核处理器的代码生成系统:XMAG。下文将详细介绍本系统。

3 相关工作

矩阵乘法的优化与代码生成一直是一个研究热点[11-14]。此类研究中,一个经常采用的方法是将任务调度与计算分离,这样既能充分利用手工优化的计算核心,又能自动生成大量不同的代码版本进行调优,Halide系统便是这一策略的典型案例[15]。还有许多研究工作基于Halide开发了搜索最优调度方案的系统性方法[16-18]。TVM是一种广泛使用的编译框架,能够为神经网络中的计算序列生成优化的代码[19]。但是TVM侧重于函数融合和任务调度,无法处理本文研究的新型处理器上的复杂数据交换和软件流水线。Ansor提供了基于现有计算核心的一种层次化解决方案[20]. Exocompilation[21]通过构建更加灵活的编译系统来解决代码生成问题,但它无法处理核间通信等操作,并且将数据移动等选项的决策任务交给用户。代码分析与生成的另外一个重要概念是多面体模型[22],它催生了许多编译优化技术,包括各种平台上的矩阵乘法代码生成[9,23-24]。但是对本文介绍的情形,该模型很难处理其复杂的数据映射和数据交换模式。

还有一些研究工作致力于生成高性能的计算核心汇编代码[25-26]。我们的汇编语言计算核心生成器中使用的优化方法,出发点与之类似,但因体系结构而异。

Tensile是为ADM GPU上的矩阵乘法生成高性能代码的一套工具,提供了一系列可调参数,构建各种并行策略和调度方式的代码[27]。TensorIR,

Graphene，Triton都能生成GPU上的高性能代码，尤其面向数据局部性进行了优化[28-30]。尽管这些工具都功能强大，但无法应用于本文面向的带有片上网络的体系结构。CUTLASS是NVIDIA开发的一个基于C++模型的框架[31]，在不同层级提供接口，可以实现灵活的函数融合，但是它的分块大小和优化策略需要用户指定。在CUTLASS基础上，有些矩阵乘法的优化专门为NVIDIA H100GPU做了优化[32-34]。这些工作也用到了片上网络和软件流水线，因此我们的代码生成系统具有与这些系统相结合的潜质。

与拥有片上网络的体系结构类似，分布式存储系统上的并行矩阵乘法也需要设计任务划分和数据通信方案[5-7]。但是，在分布式存储系统中，方案设计主要集中于通过计算与通信重叠来减少通信开销，或通过复制数据来减少通信量。而在拥有片上网络的众核处理器上，研究焦点在于设计多层次数据通信（对SW26010-Pro来说即DMA与RMA）与计算相重叠的软件流水线，且需要进行细粒度的优化。此外，我们的XMAG是一个代码生成框架，能提供更多的灵活性和可扩展性。

目前也有一些研究工作探索针对申威众核处理器的矩阵乘法代码实现。在早期的SW26010处理器上，swTVM提供了一个面向机器学习基本操作的代码生成框架，能够通过分析计算模式和分块大小确定缓冲区大小并生成恰当的DMA操作语句[35]。swATOP也构建了一个层次化的代码生成框架，将任务调度与计算核心分离[36]。但是，swTVM和swATOP都不能处理新一代申威众核平台SW26010-Pro处理器上面临的更复杂的计算模式和软件流水线。xMath2.0函数库对矩阵乘法实现进行了深度优化，其中包含了对若干异形矩阵的优化[3,8]。在XMAG的开发中，我们对标xMath2.0，对其中使用的优化方法进行了抽象、总结和扩展，从而能够通过系统性代码生成与调优，得到平均性能更高的代码。Tao Xiaohan等在PPCG的基础上开发了一个代码生成框架，能够使用多面体模型构造循环结构[9]。但是，此框架有很大的局限性。第一，它只能生成一种计算方案，并且只能使用一种固定的软件流水线。第二，因为此框架不包括汇编代码生成器，也没有自行开发汇编计算核心，因此只能调用xMath2.0中包含的计算核心。这就使得他们的分块选择必须与xMath2.0中的分块一致，所以此系统只能选择一种分块方案。这使得此系统无法适配多种异形矩阵。Xu Le等人开发了一套能够为SW26010-Pro上的矩阵乘法选择分块方案和循环顺序的框架。采用手写代码，先用coordination descent的方法对所选参数进行标注，然后用标注的实例训练出一个神经网络模型[37]。尽管此工作有详尽的分析并提供了多种循环形式，但他们实现的代码只能对计算任务进行8×8类型的切分，远不能覆盖本文所应对的诸多情况。

4　XMAG框架

XMAG的基本设计框架如图2所示，包括三个表示层次。第一层是“计算方案”，即图中的黄色方框。计算方案指的是某个函数（本文中为GEMM函数）在SW26010-Pro上的基本实现方案，包括任务划分，数据映射，及数据移动的方式。第二层是根据计算方案生成的抽象语法树（AST），即图中蓝色方框。AST可经优化遍历进一步进行变换。最后一层即由抽象语法树生成的C语言代码（图中粉色方框）。

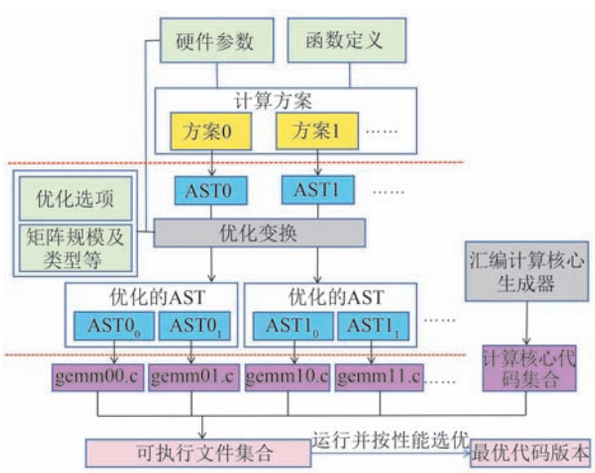

图2　XMAG代码生成流程

在生成代码时，我们首先基于函数定义和硬件配置创建一个计算方案集合。函数定义指的是函数的串行原始实现。由于本文仅讨论GEMM函数的代码生成，因此函数定义固定为矩阵乘法实现，即算法1。每个计算方案映射为一个AST。AST要进一步经过一系列变换的优化，而这些变换是根据矩阵规模特点和优化选项来确定的。根据优化选项的不同，每个基本AST可能会产生一个变换后的AST集合。最后，每个变换后的AST都会生成一份相应的C语言代码，可以直接在SW26010-Pro处理器上编译运行。这些生成的代码需要调用计算核

心汇编代码，我们将在下一章介绍汇编代码的生成。

4.1 用来表示抽象语法树的中间语言swIR

为了方便准确地表示SW26010-Pro上的各种操作，我们设计了一种中间语言：swIR。代码生成系统中的AST各节点由swIR语句来表示。swIR的基本数据结构是张量(Tensor)。swIR中的一个张量可以是一个矩阵或一个子矩阵，数据结构中包含矩阵大小、数据类型、数据位置(在LDM中还是主存中)等元数据。表1列出了矩阵乘法函数用到的swIR主要语句。使用这些语句，我们便可以精确描述一个计算方案，包括任务划分和数据映射(4.2节)，以及数据移动方案(4.3节)。为某个特定计算方案生成的AST可以经过优化变换(优化pass)生成多种AST(第5章)。

表1 swIR中的主要语句

语句	功能
DMA	发起一个DMA传输
RMA	发起一个RMA传输
MMA	调用单从核计算函数(汇编计算核心)
Wait	等待DMA或RMA传输完成
Sync	从核间同步
Padding	将数据块填充为指定大小

4.2 任务划分与数据映射

为了充分利用众核处理器的计算资源，我们需要使算法1中的嵌套循环并行地运行在从核网格上。此外，还应充分利用LDM进行数据缓存，并使用RMA数据交换来减少重复访存。因此，典型的SW26010-Pro上矩阵乘法实现如算法2所示(以$M-N-K$循环顺序为例)。对于矩阵乘法的一般情况，这种循环顺序能够最大限度地减少需要读写两种操作的C矩阵的重复访问，因此下文的讨论都以此循环顺序为基础。其他类型的循环顺序，有一些(例如$N-M-K$型)计算方案与之类似，有的则会在优化遍历的变换中实现。

算法2. SW26010-Pro一个核组上的矩阵乘法基本循环结构

```
1. for m0 from 0 to M step GM do
2.   for n0 from 0 to N step GN do
3.     for k0 from 0 to K step GK do
4.     if block_mode then
5.       DMA_GET(Ap and/or Bp) to LDM
6.     end if
7.     for m1 from 0 to M step pLoopm×PM do
8.       for n1 from 0 to N step pLoopn×PN do
9.         DMA_GET(Cp) to LDM
10.        for k1 from 0 to K step pLoopk×PK do
11.          if iterative_mode then
12.            DMA_GET(Ap and/or Bp) to LDM
13.          end if
14.          RMA_Broadcast(Ap and/or Bp)
15.          MMA; //计算Cp+=α×Ap×Bp
16.        end for
17.        if pLoopk≠1 then
18.          对Cp进行归约
19.        end if
20.        DMA_PUT(Cp) to 主存
21.      end for
22.    end for
23.  end for
24. end for
25. end for
```

算法2包含两个层次：第一个层次是外层循环，第二个层次是从核网格上的并行任务。我们将这两个层次称为“外层计算”和“内层计算”。外层计算延续了串行$M-N-K$循环，每次迭代处理一个数据块(算法2中的第1-3行)。内层计算在从核网格上对此数据块进行处理，是SW26010-Pro上矩阵乘法计算的难点和复杂之处，因此是本文的研究重点。

在内层计算中，数据块同样以$M-N-K$顺序遍历，但以并行方式进行。为了完整描述从核组上的计算方案，我们用4个数组来组成一个并行方案描述符：$pLoop$，pA，pB，pC。其中，$pLoop$表示计算任务的划分，是一个三元组：

$pLoop=\{pLoop_m, pLoop_n, pLoop_k\}$；

pA，pB，pC分别表示三个矩阵数据在从核组上的划分，每个是一个二元组，即

$pA=\{pA_m, pA_k\}$，

$pB=\{pB_k, pB_n\}$，

$pC=\{pC_m, pC_n\}$。

图3展示了计算方案描述符中各个参数的作用。其中，$pLoop_m$和$pLoop_n$表示对C矩阵更新的计算在M方向和N方向分别划分的并行任务数，每个任务由一个CPE完成，图中$pLoop_k=1$表示在K方向不做并行处理。而pA、pB、pC分别表示三个矩阵被加载到LDM时数据在从核组上的划分。

设从核组是一个$cx\times cy$的网格，那么可以根据网格结构对任务进行划分，任务划分方案需满足约束

$pLoop_m\times pLoop_n\times pLoop_k\leqslant cx\times cy$。

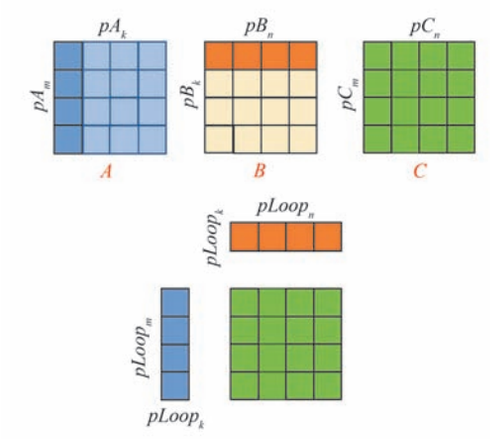

图3　计算任务划分与矩阵数据划分示意图

SW26010-Pro的从核为8×8网格，即cx=8，cy=8。如果使用整个网格，则$pLoop$可以是{8,8,1},{64,1,1},{1,64,1}等；如果仅使用部分从核，可能出现$pLoop_m \times pLoop_n \times pLoop_k < cx \times cy$的情况，例如，如果只使用从核组的上半部分，$pLoop$可以是{4,8,1}。本文暂不讨论此种情况。

一般来说，矩阵映射应与循环切分一致，也就是说，对一个矩阵某一维度进行切分的核数应等于参与此维度循环并行处理的核数。但有一种例外的情况：某个维度的循环没有进行并行切分，即此循环的并行处理核数为1，此时矩阵在此维度的切分可以大于1。下文将举例说明。在这种情况下，此维度的矩阵数据映射大于循环映射（循环映射为1）。因此，在所有情况下，都可以认为矩阵数据映射大于等于循环映射。

基于此种映射关系，内层循环处理的分块大小GM,GN,GK由以下公式得到（其中PM,PN,PK是单核计算核心的分块大小，也是DMA的分块大小）：

$$GM = \max(pA_m, pC_m) \times PM \tag{1}$$

$$GN = \max(pB_n, pC_n) \times PN \tag{2}$$

$$GK = \max(pA_k, pB_k) \times PK \tag{3}$$

综合以上规则，我们共选出了5种计算任务和数据划分方案，如表2所示。在下文中，我们用每种方案的代号来表示其对应的方案。代号中打头的字母表示在哪（几）个方向并行，之后数字表示在几个维度上并行。数字之后的字母表示在哪个方向串行迭代。

图4和图5展示了其中两种计算方案：MN2和M1N。图4是最常使用二维分块并行（表2第一行），即

表2　当前XMAG中能够生成的主要计算与数据划分方案

代号	方案描述	$pLoop$	pA	pB	pC
MN2	二维分块并行，K方向迭代	8,8,1	8,8	8,8	8,8
M1	M方向并行，K方向迭代	64,1,1	64,1	64,1	64,1
N1	N方向并行，K方向迭代	1,64,1	1,64	1,64	1,64
K1	K方向并行	1,1,64	1,64	64,1	1,1
M1N	M方向并行，N方向迭代	64,1,1	64,1	1,64	64,1

$pLoop$={cx,cy,1},

pA={cx,cy},

pB={cx,cy},

pC={cx,cy},

其中，cx=cy=8。对外层计算而言，每次更新一个$GM \times GN$的矩阵块，整个矩阵以这样的矩阵块为单位进行处理。在内层计算中，一个矩阵块被分成$cx \times cy$个小块，每个从核负责更新一个这样的小块。由于pA_k=pB_k=8，内层计算中引入了一个K方向上以PK为步长的循环。此循环的每次迭代中，各个从核沿K方向读入不同的$\boldsymbol{A}$、$\boldsymbol{B}$矩阵块进行计算，来更新本地$\boldsymbol{C}$矩阵块。图4中的深绿色平面表示K方向第一次迭代所做的更新，使用的是$\boldsymbol{A}$矩阵的第一个列块和$\boldsymbol{B}$矩阵的第一个行块。

图5展示的映射方式即上文提到的“例外”情况，即表2最后一行所示方案，适用于K很小而M和N较大的情况。此处

$pLoop$={$cx \times cy$,1,1}={64,1,1},

也就是说，M维度的循环完全并行在整个从核组上，而其他循环都保持串行。令pA={64,1},pC={64,1}，即$\boldsymbol{A}$和$\boldsymbol{C}$在M方向上切分到全部64个从核。由于$pLoop_k$=$pLoop_n$=1，$\boldsymbol{B}$的映射数组$p\boldsymbol{B}$可以是{64,1}或{1,64}。由于我们要处理的是K很小而N较大的$\boldsymbol{B}$矩阵，因此可以采用pB={1,64}的映射，这意味着$\boldsymbol{B}$矩阵在N方向上划分给64个从核，而内层计算有一个N方向上的循环，每次迭代由一个从核将$\boldsymbol{B}$矩阵的一个小块广播给所有从核，如图5所示。图中深绿色部分，是第二次迭代中更新的$\boldsymbol{C}$矩阵数据。

4.3　数据移动策略

从核组上的并行化引入了计算方案中一个新的参量，即如何将数据载入LDM。仍以图4所示的MN2计算方案为例。在此情况下，K方向每次迭代中的计算只需要$\boldsymbol{A}$矩阵的一个列块参与，按照矩阵映射方案，这一个列块由一列从核“负责”。这就意味着在K方向每次迭代中，并不需要所有从核参与

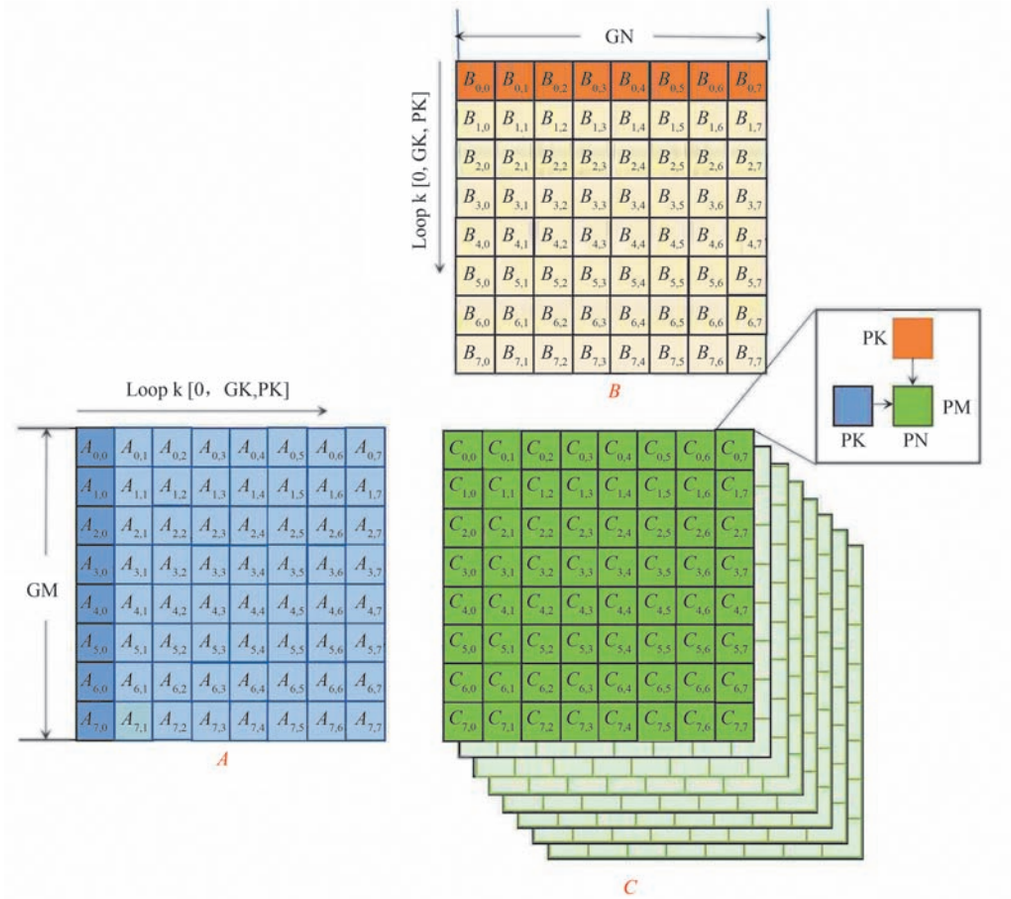

图4 计算方案MN2：$pLoop$={8，8，1}，$p\boldsymbol{A}$={8，8}，$p\boldsymbol{B}$={8，8}，$p\boldsymbol{C}$={8，8}

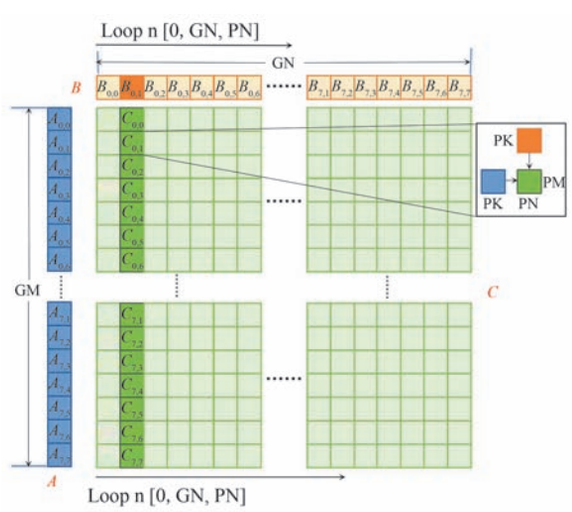

图5 计算方案M1N：$pLoop$={64，1，1}，$p\boldsymbol{A}$={64，1}，$p\boldsymbol{B}$={1，64}，$p\boldsymbol{C}$={64，1}

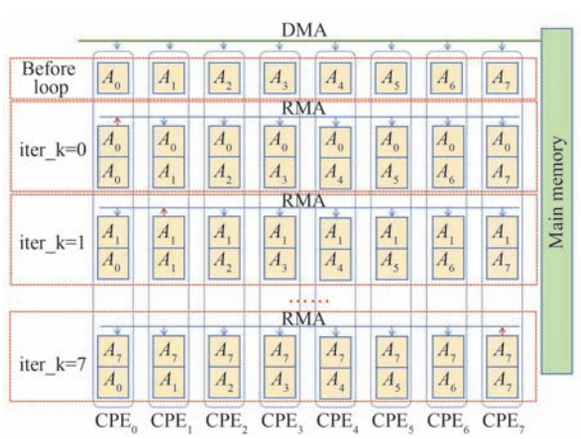

图6 $pLoop$={8，8，1}，$p\boldsymbol{A}$=$p\boldsymbol{B}$=$p\boldsymbol{C}$={8，8}的计算方案，采用块模式的数据交换示意图

$\boldsymbol{A}$矩阵数据的搬运。因此，$\boldsymbol{A}$矩阵的加载可以两种方式完成：块模式和迭代模式。图6和图7展示了这两种模式的工作方式。

在块模式下，所有的数据都在最内层K循环之前通过DMA加载到各个从核的LDM中，如图6所示。之后，每次迭代所需的数据由RMA操作传输给各个从核。而在迭代模式下，每次迭代仅由一列从核加载本次迭代所需的$\boldsymbol{A}$矩阵块，再通过调用RMA操作来将数据广播到需要的从核。如图7所示，在第一次迭代中，矩阵块A_0由0号从核加载到LDM中，并广播到同行的其他从核中。两种模式在不同的场景中各有优劣。一般来说，块模式能够更

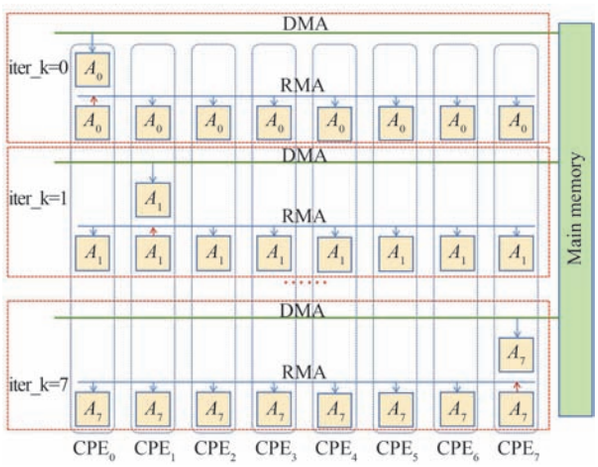

图7　$pLoop=\{8,8,1\}$，$pA=pB=pC=\{8,8\}$的计算方案，采用迭代模式的数据交换示意图

高效地利用DMA访存带宽；而迭代模式占用更少的LDM空间，并提供更多计算与访存重叠的机会。

在数据已经加载到从核的LDM中之后，我们需要确定在从核间进行数据交换所使用的RMA操作。例如，在前述MN2（图4）＋迭代模式的计算方案中，从核组中的一列在K方向循环每次迭代中读取一个列块，并采用RMA行广播将数据广播到同行的其他从核。类似地，读取B中一个行块的从核通过RMA列广播将数据广播到同列中其他从核。

4.4　生成基本抽象语法树

在确定了任务分配与数据移动方案之后，我们便可以生成抽象语法树（AST），其基本循环结构如算法2所示。在对AST进行初始化之后，先根据计算方案中的循环顺序生成外层M，N，K循环，循环步长分别是GM，GN，GK，对应算法2中的第1至3行。之后，按以下步骤生成内层循环，其中$\boldsymbol{A}_p$，$\boldsymbol{B}_p$，$\boldsymbol{C}_p$代表加载到一个从核的LDM中的矩阵块。

第一步，确定内层循环的循环形式，即算法2中第7、8、10行的for语句。这些循环仍然保持M—N—K的顺序，每个循环的步长分别为$pLoop_m\times PM$，$pLoop_n\times PN$，$pLoopk\times PK$，循环范围GM，GN，GK由公式（1），（2），（3）定义。在某些计算方案中，一些循环可能消失。例如，只有当$pLoop_n<pB_n$，循环n_1才会存在（即迭代次数大于1），也就是图5的情况；否则循环n_1便会消失。

第二步，在确定循环结构之后，如果采用块模式，我们就可以插入$\boldsymbol{A}$或$\boldsymbol{B}$矩阵的读取操作（DMA_get）。根据具体的计算方案，此处可能需要将$\boldsymbol{A}$或$\boldsymbol{B}$都读取，也可能只读取其中一个。

第三步，如果内层M和N方向循环迭代次数超过1，则插入这两个维度的循环（算法2的第7、8行）。并插入$\boldsymbol{C}$矩阵的读取操作（DMA_get）。

第四步，如果内层K方向循环迭代次数多于1（数据加载采用迭代模式的都属于此类），则生成此循环（第10行），并根据计算方案插入$\boldsymbol{A}$和$\boldsymbol{B}$的DMA_get语句（第12行）。

第五步，根据数据交换模式插入RMA语句（第14行）。

第六步，插入计算核心调用语句，即MMA（第15行）。

第七步，如果$pLoop_k>1$，说明循环k_1是并行化的，因此需要加入归约语句来对$\boldsymbol{C}$矩阵进行更新（第18行）。

第八步，将$\boldsymbol{C}$矩阵的写回操作（DMA_put）插入恰当的位置（第20行）。

算法3是一个AST的例子，展示了MN2方案（即$pLoop=\{8,8,1\}$，$pA=pB=pC=\{8,8\}$），数据移动使用块模式。其中迭代次数为1的循环已经移除。

一个需要提及的问题是缓冲区分配。在计算过程中，每个矩阵都关联到LDM中的一块缓冲区。但是一个矩阵可能需要多块缓冲区。例如，在块模式下，每个从核需要在LDM中为$\boldsymbol{A}$和$\boldsymbol{B}$矩阵另外开辟一块缓冲区，来接收每次迭代计算需要的数据。而实现软件流水线则需要更多的缓冲区来保证操作的并发执行不会造成数据读写冲突。

5　优化方法

本节介绍XMAG如何设计生成软件流水线，生成与手工优化性能相媲美的代码，以及其他相关优化技术。

5.1　软件流水线

在讨论软件流水线生成之前，我们先介绍XMAG系统中用到的两个传统编译优化技术。一是循环消除。如果某个循环只有一次迭代，那么此循环可以被消除。我们在算法2中已经根据计算方案消除了某些循环。在优化变换中，我们可以依据矩阵信息进行更多的循环消除。第二项优化技术是循环不变式外提。如果一个语句在循环的每次迭代中产生同样的输出，那么它可以被识别为一个循环不变式，因此可以被移到循环外，从而减少重复操作。例如，算法3展示了MN2＋块模式的计算方案。假设N和K都较小（$K\leqslant GK$，$N\leqslant GN$），那么就满足循环消除原则，循环n_0和k_0能够被消除。这样

就得到了算法4的代码。然后，通过循环不变式外提变换，找到循环不变式DMA_GET(B)，此语句可以被提到循环m_0之外。经过这两个变换之后，我们便得到了算法5所示的代码。在此代码基础上可进行针对M循环的软件流水线变换，而这在算法3上是无法应用的。

算法3. $pLoop = \{8, 8, 1\}$，$pA = pB = pC = \{8, 8\}$、块模式的计算方案

```
1. for m0 from 0 to M step GM do
2.   for n0 from 0 to N step GN do
3.     DMA_GET(Cp) to LDM
4.     for k0 from 0 to K step GK do
5.       DMA_GET(Ap) to LDM
6.       DMA_GET(Bp) to LDM
7.       for k1 from 0 to K step pLoopk×PK do
8.         RMA_Broadcast(Ap and/or Bp)
9.         RMA_Broadcast(Ap and/or Bp)
10.        MMA; //计算Cp+=α×Ap×Bp
11.      end for
12.      DMA_PUT(Cp) to 主存
13.    end for
14.  end for
15. end for
```

算法4. 对算法3进行循环消除之后的代码形式

```
1. for m0 from 0 to M step GM do
2.   DMA_GET(Cp) to LDM
3.   DMA_GET(Ap) to LDM
4.   DMA_GET(Bp) to LDM
5.   for k1 from 0 to K step pLoopk×PK do
6.     RMA_Broadcast(Ap)
7.     RMA_Broadcast(Bp)
8.     MMA; //计算Cp+=α×Ap×Bp
9.   end for
10.  DMA_PUT(Cp) to 主存
11. end for
```

算法5. 对算法4进行循环不变式外提之后的代码形式

```
1. DMA_GET(Ap and/or Bp) to LDM
2. for m0 from 0 to M step GM do
3.   DMA_GET(Cp) to LDM
4.   DMA_GET(Ap) to LDM
5.   for k1 from 0 to K step pLoopk×PK do
6.     RMA_Broadcast(Ap)
7.     RMA_Broadcast(Bp)
8.     MMA; //计算Cp+=α×Ap×Bp
9.   end for
10.  DMA_PUT(Cp) to 主存
11. end for
```

现在我们详细介绍软件流水线的设计和生成。我们用“stage”来定义一条软件流水线。对任何一个循环，循环体由一系列同步或异步操作组成。一个操作或一个操作序列可以被看作一个stage。只要一个stage包含异步操作，它就有可能与其他stage重叠，而从中获得性能收益。因此，一个循环便可以通过不同的操作重叠方式，形成不同的软件流水线。在XMAG中，我们按以下步骤生成一条软件流水线。

第一步，确定进行软件流水的维度。

在这一问题上，对矩阵乘法来说，我们主要考虑两种流水线。第一类流水线在K维度循环上进行操作重叠，这适用于K足够大的情况。第二类流水线包括在M或N方向进行操作重叠的流水线，适用于K维度较小，以至于在K循环上的流水线过短的情况。

第二步，确定各阶段间重叠的方式。

我们先来看第一类流水线。一个典型例子即MN2方案＋迭代模式的计算方案。如算法2所示，最内层循环包括以下操作：DMA_get($\boldsymbol{A}$,$\boldsymbol{B}$)，RMA_broadcast($\boldsymbol{A}$,$\boldsymbol{B}$)，以及MMA。这些操作可以按不同方式进行重叠。比如，我们可以将这几种操作全部重叠，即形成图8所示的流水线。另外，我们也可以将DMA_get和一个“计算”stage重叠。在不同的场景中，“计算”stage可以有不同的定义，它可以是RMA＋MMA，也可以仅是一个MMA操作。如果“计算”定义为仅包含一个MMA操作，此流水线适用于不使用RMA进行数据交换的计算方案（例如，M和N很小，K很大的情况）；如果“计算”定义为RMA＋MMA，则此流水线适用于DMA带宽较低的系统（例如SW26010）。

再来看第二类流水线。此类流水线适用于$\boldsymbol{A}$或$\boldsymbol{B}$矩阵较小的情况。我们以算法5中的抽象语法树为例（$\boldsymbol{B}$较小的情况）。这段代码中，DMA_get($\boldsymbol{B}$)已经作为循环不变式被移出循环，因此M循环包含三项操作：DMA_get($\boldsymbol{A}$，$\boldsymbol{C}$)、计算阶段（即RMA和MMA）、DMA_put($\boldsymbol{C}$)。如果将各个操作完全重叠，则软件流水线如图9所示。这种流水线的一个变种是将两个DMA_get操作及计算stage串行化，使它们合为一个stage，与DMA_put重叠。另一种变换是将计算stage与DMA_put串行化，使之与DMA_get重叠。

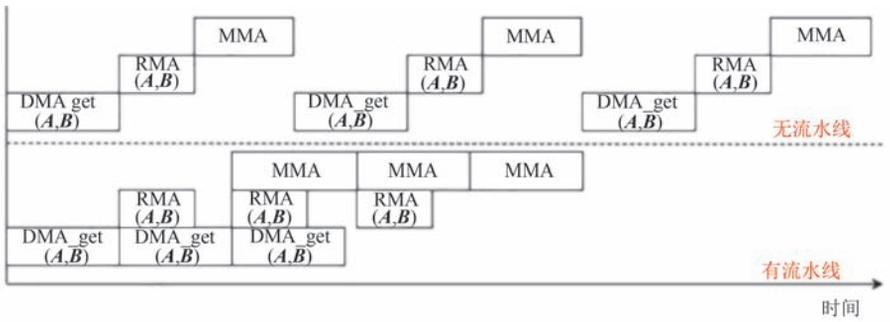

图8　DMA、RMA、MMA重叠的流水线

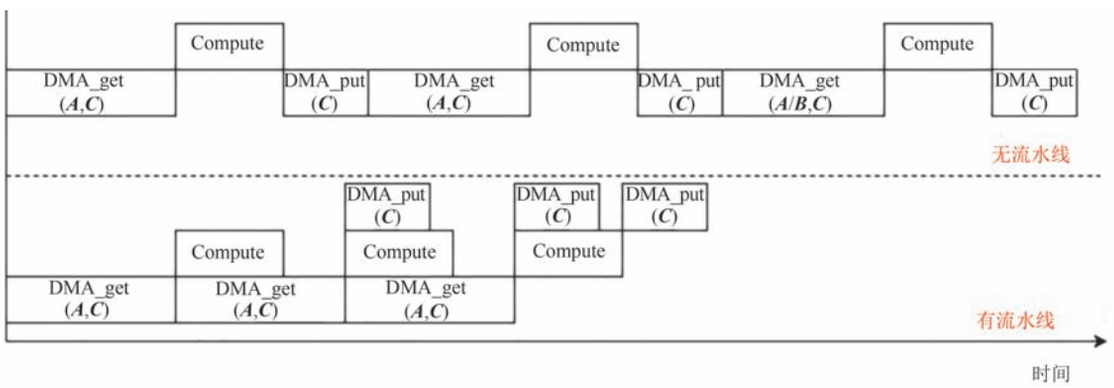

图9　DMA_get、DMA_put、计算重叠的流水线

第三步，将迭代剥离，完成整个计算。

给定一种重叠策略，我们根据各个阶段的定义及其顺序剥离出前序迭代和后序迭代，从而生成相应的AST，并进行缓冲区管理。此外，在迭代模式中，如果外层循环和内层循环是在同一维度上，则将两个循环合并。例如，如果循环m_1和n_1不存在，那么我们就将循环k_0和k_1合并，形成更长的不间断的流水线，从而减少前序迭代、后序迭代及反复更新C矩阵造成的开销。

第四步，生成嵌套流水线。

考虑图10所示的流水线：通过对k_1循环内的计算进行N方向上的进一步切分，可以得到RMA与MMA重叠的流水线。如果我们将图9与图10的流水线结合，就得到了“嵌套”流水线。外层流水线由M循环上的DMA_get($\boldsymbol{A},\boldsymbol{C}$)，计算stage（图9中的“compute”），以及DMA_put($\boldsymbol{C}$)三个stage重叠，而内层流水线重叠了计算stage内的RMA和MMA操作，也就是开辟了k_1循环内的一个新的N方向循环。这种嵌套流水线适用于算法5，也就是N和K较小，可以驻留在整个从核组的LDM中，但又没有小到可以全部放进单个从核的LDM中的情况。这种策略提供了更多的灵活性，使得不同的操作可以使用不同大小的缓冲区（DMA使用较大分块，RMA和MMA使用较小分块）。

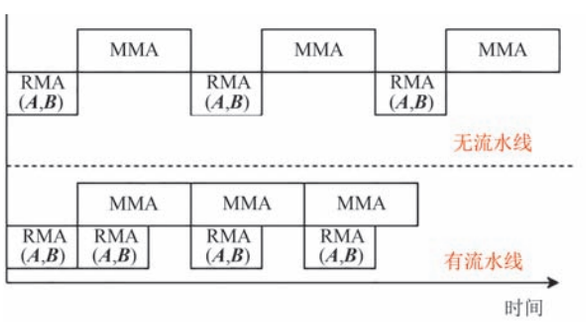

图10　RMA与MMA重叠的流水线

5.2　数据补齐相关问题

由于汇编计算核心要求数据符合一定的分块大小，所以在处理矩阵边缘数据时，经常碰到需要将数据块补齐的情况，因此产生了何时进行补齐操作的问题。如果计算过程需要RMA操作，那么这个问题就面临两种选择。一个方案是在数据通过DMA_get加载到LDM之后，立即进行补齐。另一个方案是在MMA之前，RMA之后进行补齐。第一种方案的好处是不会使MMA stage运行时间延长。第二种方案的优势在于能够减少DMA_get stage的开销。在XMAG中，我们将这两种选择作为代码生成的一个参数，通过自动调优为具体问题选择最优方案。

另外一个问题是确定数据块补齐时的实际大

小。这个问题的出现是流水线导致的。以图8为例，当数据通过DMA加载到LDM中时，我们便知道了数据块的实际大小。在K循环的下一次迭代中，数据块会通过RMA操作广播给其他从核。再下一次迭代，此块数据才会用于MMA计算。因此，在同一次迭代中，DMA、RMA、MMA三个stage需要分别确定所操作的数据块的大小。一个直观的解决方法是为一次迭代中的每个操作维护一个表示数据块规模的结构体。但这种方法会引入很多判断，增加条件分支造成的开销，并且也会增大代码生成的复杂度。我们采用了一种更为便捷的方法，如图11所示，当数据从主存中加载进来时，我们把每个数据块的实际大小作为元数据存放在这个数据块的开头。之后，不管数据块被发送到哪里，它的元数据都与它一起被传输，这样就保证对此数据块的任何操作都能获知其实际大小。图11(a)描述的是补齐发生在MMA之前的情况。加载数据块的从核CPE_i将它的大小(用pm和pn两个整数表示)写入元数据位置，然后发送给CPE_j。CPE_j即可根据此元数据将数据块补齐为目标大小($PM\times PN$)。图11(b)展示的是另一种情形，即数据块在进行RMA传输之前就进行打包，此时元数据也会被更新，从而避免数据块被CPE_j再次进行补齐。

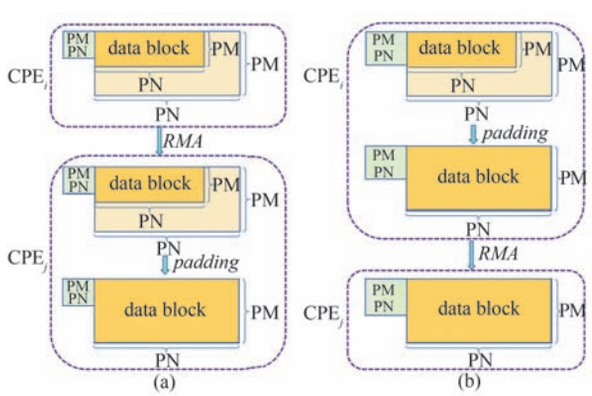

图11 带有元数据的数据块在从核间的传输及补齐(其中(a)是先RMA传输再补齐，(b)是先补齐再进行RMA传输)

5.3 汇编语言计算核心

如前所述，每个从核的计算是由一个精细调优的汇编计算核心完成的，而对不同的矩阵分块，我们需要一系列不同的计算核心。通常，在代码中使用宏定义表示一些计算模块，可以通过改变宏的值生成多个计算核心。但是这种方法的灵活度有限，因为有些计算核心不只是改变了某些参数，而是需要对代码本身进行变化。为了解决这个问题，我们还开发了汇编计算核心代码生成器。计算核心按$M-N-K$的循环顺序进行计算。最内层循环每次对一个$RM\times RN$的C矩阵块进行更新，如图12所示。

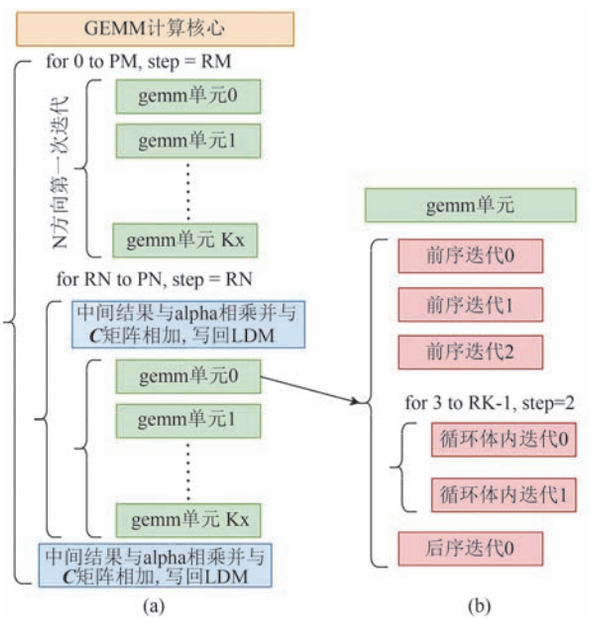

图12 汇编语言计算核心代码结构

RM、RN、RK的值是由SIMD指令的宽度和寄存器分配方案决定的。由于向量长度为8个双精度浮点数，按计算访存比最优的原则，缺省值选择RM=2，RN=4，RK=8，选择方法详见文献[3-4]。在XMAG的汇编代码生成器中，由于我们需要处理不同规模的最内层分块，因此对M方向较小的情况，选择RM=1，RN=8。代码生成器对内层循环和外层循环按一定的展开次数进行循环展开，以便更好地进行指令排布。为了充分利用每个从核上的硬件流水线以获得满意的指令级并行，我们为程序中的每个基本块维护两个指令队列，一个是SIMD浮点指令队列，另一个存放所有其他指令。在保证指令依赖不被破坏并考虑指令延迟的前提下，两个队列中的指令被交叉取出，放到要生成的计算核心代码，从而获得尽可能高的指令级并行。因此代码以如下所示序列的形式呈现：

浮点向量乘减(浮点计算指令)
浮点向量数据加载(访存指令)
浮点向量乘减(浮点计算指令)
浮点向量数据加载(访存指令)
浮点向量乘减(浮点计算指令)

指针移动(整型指令)

……

代码生成器提供了一系列的参数，来定义不同的代码形式。主要参数如下：

(1)循环展开次数：指定外层循环和内层循环的展开策略，包括前序迭代次数和后序迭代次数。后序迭代的最后一次迭代需要展开更多次，以方便插入数据预取，实现访存与计算的重叠。

(2)预取数量：这些参数指定在进入循环体之前要预取A矩阵和B矩阵的多少个元素。

(3)分块大小：分块大小即最内层循环每次迭代计算多少个矩阵元素。它决定了寄存器的使用，并影响双缓冲策略的选择。例如，当N方向的分块较大时，加载B矩阵就需要使用更多的寄存器。而使用双缓冲需要为相应的矩阵提供双倍的寄存器。因此，这样会造成寄存器使用数量超过实际寄存器数。所以，此种情况下，我们对B矩阵不使用双缓冲策略。

我们将采用不同参数组合生成的汇编计算核心分别编译为目标文件，与C语言代码生成的目标文件共同链接为可执行文件。

5.4　自动调优

与众多代码生成系统类似，XMAG也采用自动调优的方法为特定的输入选择最优代码。如图2所示，我们首先生成一个候选计算方案集合，然后为每个计算方案生成对应的AST。接下来，我们根据输入数据规模特点，生成优化选项组合的集合，对每个AST进行优化变换，生成多个优化的AST，形成最终的AST集合。最后，为集合中的每个AST生成适用于SW26010-Pro处理器的C语言代码。在进行自动调优时，我们首先列举出所有分块大小组合，形成分块大小集合BS。然后，对给定的输入规模，我们枚举所有可能用到的计算方案，再为每个计算方案设置可能的优化选项，形成方案集合SS。这两个集合的外积会形成一个较大的搜索空间。为了缩小搜索空间，我们选用了几种优化策略。第一种是根据矩阵规模进行剪枝。例如，如果生成代码所需缓冲区大小超过了硬件配置LDM中可用空间的大小，则此代码被抛弃。另外，我们会根据矩阵规模对包括数据移动策略在内的计算模式进行取舍。例如，任何规模的矩阵乘法，我们都会生成MN2+迭代模式的计算方案，因为这种计算方案能够充分利用效率最高的三缓冲流水线，即图4所示流水线，这也是xMath2.0中主要采用的实现方案和缺省实现方案[3-4]。而如果N和K维度都较小，使得B矩阵能够驻留于整个CPE网格，那么就还会生成MN2+块模式的代码。因为此种情况下，采用块模式能够进一步进行循环消除和循环不变式外提的优化，进而生成新的软件流水线。

第二种是根据已有经验和测试结果对某些选项进行设定。例如，我们发现，对M方向的流水线，在M较大，N和K都较小时，图9所示的重叠方式是最佳的，那么我们就固定选择此种重叠方式。此外，在运行过程中，我们可以将各个代码版本的性能数据加入代码数据库中。后续的调优即可利用这些性能数据，对某些优化选项进行设定。例如，如果要对某个给定规模(M,N,K)的矩阵乘法进行调优，而数据库中已有相似规模(M',N',K')的性能数据，那么我们可以仅在(M',N',K')最优代码版本的基础上进行少量参数改变进行选优。

经过剪枝，我们为每种规模生成的代码集合包含代码版本数约在几十到几百个。每个规模的矩阵乘法代码集合运行时间约为10到30分钟。

在生成和运行所有候选代码版本之后，我们选出最优者。而代码数据库能够用于之后的调优，帮助减少调优时间。

6　实　验

为了检验我们的代码生成系统，我们进行了一系列实验。实验在SW26010-Pro处理器的一个核组上完成，对双精度实数矩阵乘法(DGEMM)进行如下测试。首先，我们测试了XMAG生成的代码在方阵上的性能。之后，我们展示生成的代码在异形矩阵上的性能。为了验证生成代码的性能，我们将XMAG所生成的代码的运行时间与xMath2.0[23]函数库中的DGEMM函数运行时间进行了比较。

本文的测试都是在A、B矩阵不转置的情况下做矩阵乘法计算。对A或B转置的情况，需要在进行DMA_get操作时对坐标计算进行相应变换。此外，如果A矩阵转置，则将5.2节所述打包操作改为转置操作，利用SW26010-Pro提供的读写表指令，可以高效实现此操作[3]。对于B矩阵转置的情况，只需在计算核心汇编代码生成器生成汇编代码时加以设置，就能生成B矩阵转置的计算核心代码。本节仅对不转置情况展示测试结果，以说明XMAG系统的有效性。

6.1 大规模矩阵上的矩阵乘法

xMath2.0在大规模方阵上的矩阵乘法能够获得接近峰值的浮点计算性能，我们首先检验XMAG生成的代码在这类矩阵上是否能达到与其相当的性能。图13展示了此类矩阵上的性能比较，可以看出，XMAG生成的代码性能与xMath2.0非常接近，证明了我们的代码生成系统能够为大规模矩阵生成高性能代码。

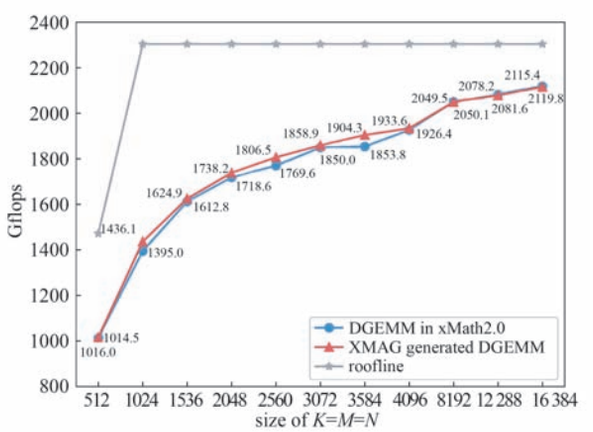

图13 大规模方阵上的矩阵乘法性能比较

6.2 异形矩阵上的矩阵乘法

XMAG的优势在于为不同大小的矩阵生成不同的代码。我们比较了异形矩阵上双精度浮点数矩阵乘(DGEMM)的性能，结果分别展示在图14至图19中。“异形矩阵”是指矩阵乘法中有一个维度较小，两个维度较大；或两个维度较小，一个维度较大。在这些测试中，我们将“较大”的维度大小设为8192。对只有一个维度较小的矩阵乘法，“较小”的维度大小在[16，128]范围内，以16为步长取值。对两个维度都较小的矩阵乘法，“较小”的维度大小除上述范围外，我们增加了[128，512]范围内的取值，以64为步长。我们测试了6种不同类型的异形矩阵乘法，与xMath2.0相比，平均加速比达到2.32，最高加速比达到10.55。

图14展示了N较小的矩阵乘法性能。当N较小时，我们对访存受限的情况选择M1+块模式，而对计算受限的情况选择MN2+迭代模式。当$N\leqslant 64$，XMAG生成的代码和xMath2.0都达到了接近roofline模型所预测的理论峰值的性能。但是当N在64和128之间时，xMath2.0不再使用此方案，而是采用了缺省版本，因此性能有比较明显的下降，而XMAG由于采用了系统性的调优，能够保持较平滑

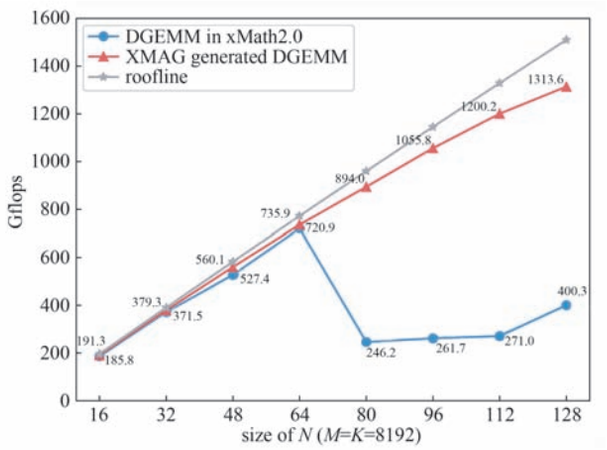

图14 N较小的矩阵乘法性能比较

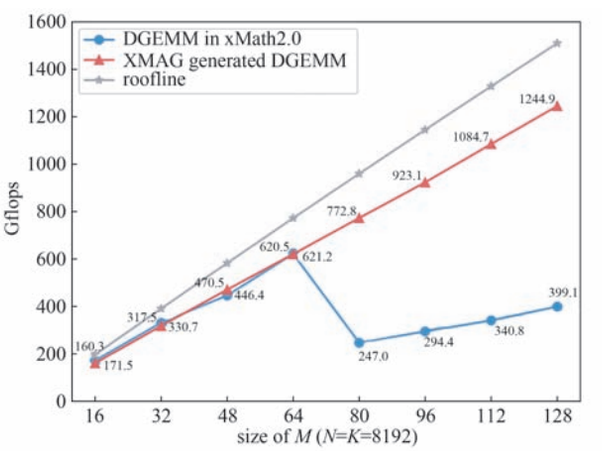

图15 M较小的矩阵乘法性能比较

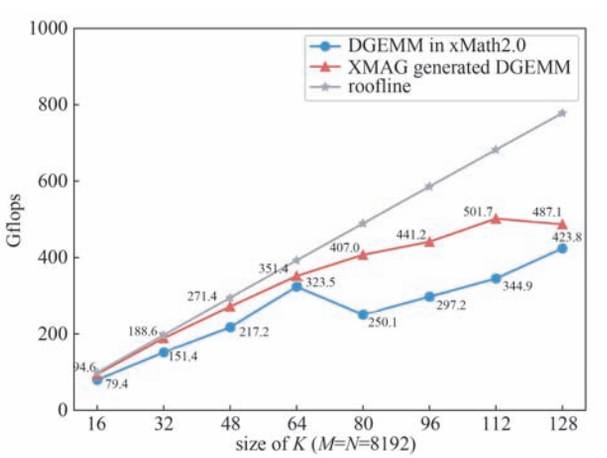

图16 K较小的矩阵乘法性能比较

的性能曲线。以$M=8192$，$K=8192$，$N=96$的情况为例，XMAG生成的方案中性能最优的是M1+块模式，$PM=64$，$PN=96$，$PK=48$，而xMath2.0使用的是缺省方案，即MN2+迭代模式，$PM=64$，$PN=64$，$PK=32$。表3(1)列出了这两份代码在浮点计算性能和访存性能上的差异。XMAG选出的最优

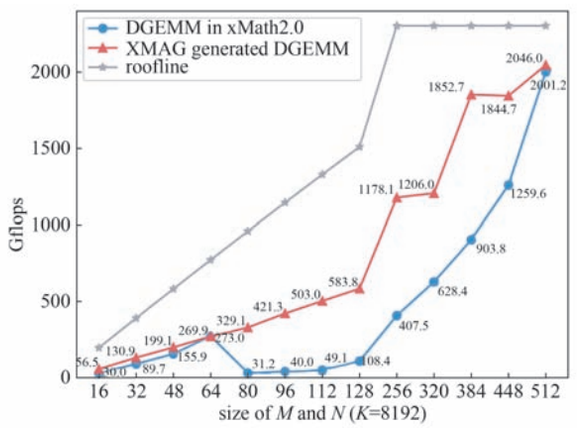

图17　M和N较小的矩阵乘法性能比较

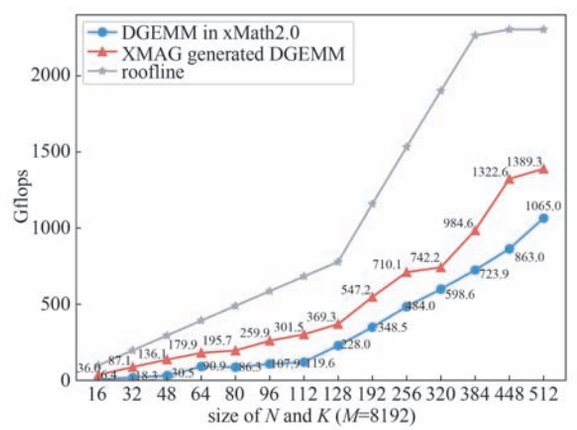

图18　N和K较小的矩阵乘法性能比较

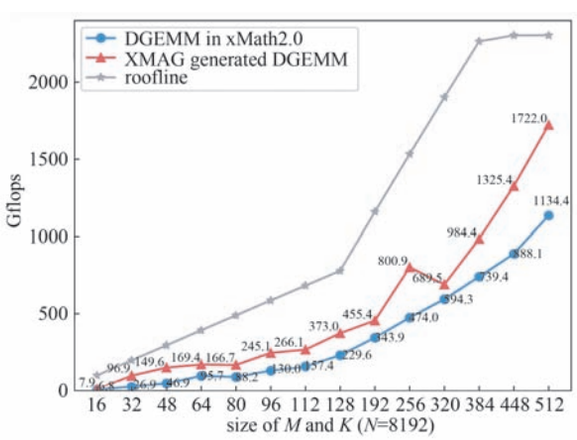

图19　M和K较小的矩阵乘法性能比较

方案采用一维分块并行，并且使PN与N大小相同，这样在M方向迭代次数减少，减少了B矩阵的重复读取。此外，xMath2.0使用的缺省方案采用二维分块并设定PM=64，PN=64，而本例中N=96，这就使得相当一部分CPE的计算时间浪费在对无效数据进行计算。因此XMAG为此类乘法选择的方案取得了显著的性能收益。

M较小的情况，其性能趋势与N较小的情况类似，性能比较见图15。XMAG选取的最优方案为N1+块模式，PM=112，PN=64，PK=64，而xMath2.0使用的是缺省方案，即MN2+迭代模式，PM=64，PN=64，PK=32。由表3(2)可以看出，XMAG取得性能优势的原因与N较小的情况类似，也是采用更合理的排布减少了资源闲置时间。

表3　XMAG生成代码与xMath2.0访存和浮点计算执行参数比较(表中数据采用申威平台上的swperf工具获得)

(1) M=8192，N=96，K=8192

	XMAG	xMath2.0
总访存量	read 199402704 B，write 22742868 B	read 272483796 B，write 22756148 B
执行向量浮点运算指令数	144 506 881	681 574 401

(2) M=112，N=8192，K=8192

	XMAG	xMath2.0
总访存量	read 198903664 B，write 23149036 B	read 278437838 B，write 23225836 B
执行向量浮点运算指令数	149 094 401	681 574 401

(3) M=8192，N=8192，K=80

	XMAG	xMath2.0
总访存量	read 213734632 B，write 205167316 B	read 267364490 B，write 205287944 B
执行向量浮点运算指令数	106 168 321	173 016 321

图16比较了K较小的情况，经过调优，XMAG选取的计算方案是M1N+块模式。xMath2.0在K小于等于64时，也选取了此种方案。但XMAG生成的代码通过选择最优的分块大小和计算核心，能够获得高于xMath2.0的性能。而K>64时，xMath2.0仍然使用的是缺省版本，因此与XMAG相比性能有明显下降。以M=8192，N=8192，K=80的规模为例，XMAG生成的方案中性能最优的是M1N+块模式，PM=64，PN=32，PK=80。由表3(3)可以看出，与前两种情况类似，XMAG不但减少了冗余的浮点计算，总访存量也更少。这是由于此种方案对A和B两个矩阵的读取相当于“一次性DMA读入，再RMA分发”，因此减少了A、B矩阵的重复读取。

图17比较了M和N较小的情况。可以看出，当M和N在64和512之间时，XMAG生成的代码优于xMath2.0。在此区间xMath2.0采用了缺省实现(MN2+迭代模式)。而XMAG能够在更大的代码

和参数空间进行搜索。例如，当$M=N=112$时，XMAG选择了与xMath2.0相同的计算方案，但分块改为$PM=32$，$PN=112$，$PK=64$，由于分块与维度相同，因此减少了打包开销。而较大的PK又减少了C矩阵写回的开销。所以，生成的代码总体性能明显好于xMath2.0。当$M=N=512$时，xMath2.0和XMAG都采用MN2+迭代模式的方案和同样的分块大小，而此种方案在此规模时已经能够充分利用处理器核心和DMA带宽，因此都获得到了接近峰值的性能。当M和N在128和512之间时，由于XMAG生成代码时能够选择更优的分块大小，使得线程间负载更均衡，计算资源利用率更高，因此获得了更好的性能。

图18比较了N和K较小的情况。当N和K非常小时，我们可以通过采用循环消除等方法将B矩阵加载到每个从核的LDM，之后的计算便只有M方向的循环。当N和K变大时，如前文所述，会触发循环消除和循环不变式外提等优化变换，之后便可以采用5.1节所描述的嵌套流水线。因此，XMAG生成的代码具有与xMath2.0类似的性能曲线。而由于我们进行了更加全面的参数搜索，使用更加完备的汇编计算核心代码生成器，因此整体性能优于xMath2.0。在N和K稍大时，虽然使用嵌套流水线能在一定程度上进行更充分的资源利用，但是流水线的前序操作和后序操作仍占有较大比例，并且DMA读操作与写操作的重叠会造成一定程度的拥塞，因此XMAG生成代码及xMath2.0的代码性能都与理论峰值有一定差距。

图19比较了M和K较小的情况。由于有更精细的调优，XMAG生成的代码在所有用例中性能都超过了xMath2.0。尤其当M和K取值在[128，512]范围内时，xMath2.0仍使用缺省实现，而XMAG生成了xMath2.0中未实现的嵌套流水线，在N方向进行图9所示的流水，使A矩阵驻留在LDM中，并进一步对RMA和计算操作采用嵌套流水线，因而获得了明显高于xMath2.0的性能。与图18的情况类似，XMAG生成代码及xMath2.0的代码与理论峰值有一定差距。但与图18相比，较大矩阵（图19中的B矩阵、图18中的A矩阵）的访存在每次迭代中相对更连续，因此XMAG生成代码的性能与理论峰值差距略小。

6.3 XMAG生成矩阵乘法代码在大小维度比例在1到16之间的矩阵上的性能

除上述典型异形矩阵乘法外，我们还测试了一些更接近方阵的矩阵乘法（长宽比在1到16之间），其性能比较展示在图20～图23。其中，图20固定M大小为64，而将N和K的值设为128到1024之间。此类规模上XMAG生成代码略好于xMath2.0，但整体性能均偏低。图21固定N和K的大小为8192，M的取值在512和4096之间。在此区间的矩阵乘法，由于整体规模较大，因此XMAG生成的代码和xMath2.0采用了同样的计算方案和分块参数，且性能非常接近。

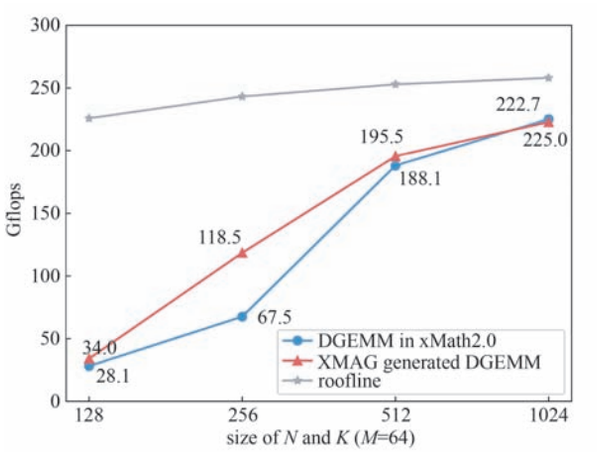

图20 M较小的矩阵乘法（大小维度比例在1～16之间，且整体规模较小）性能比较

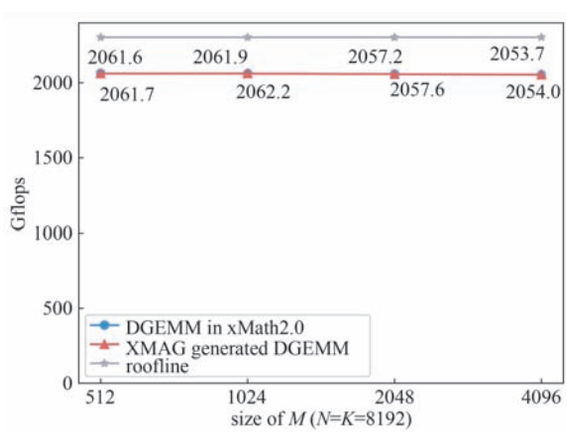

图21 M较小的矩阵乘法（大小维度比例在1～16之间，且整体规模较大）性能比较

图22和图23展示的是N和K较小的情况。图22中，N和K固定为64，M大小在128和1024之间。这几种情况也是矩阵规模较小，整体性能较低，XMAG生成的版本与xMath2.0性能接近。图23展示的是M固定为8192，N和K取值在512到4096之间的情况。当N和K在1024及以上时，XMAG生成的代码和xMath2.0采用了同样的计算方案和分

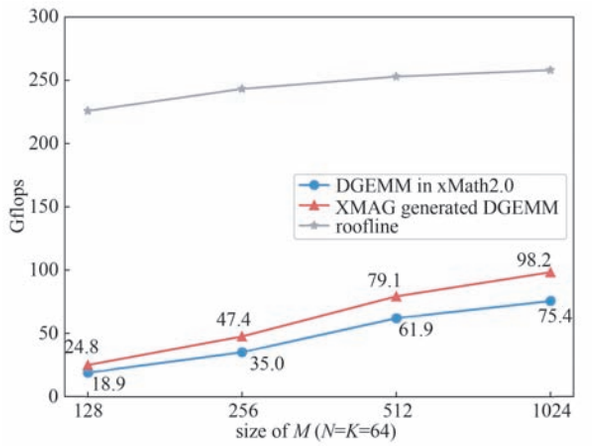

图22　*NK*较小的矩阵乘法（大小维度比例在1～16之间，且整体规模较小）性能比较

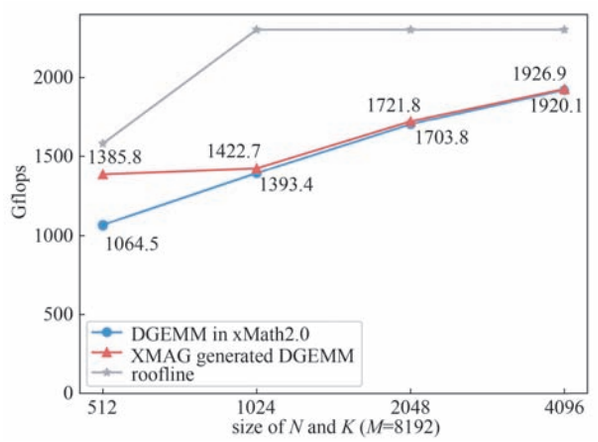

图23　*NK*较小的矩阵乘法（大小维度比例在1～16之间，且整体规模较大）性能比较

块参数，因此性能接近。而当$N=K=512$时，XMAG选择了更优的分块大小，获得了更高的性能。

6.4　不同优化方法对性能的影响

XMAG的主要用途是为给定规模和数据类型的矩阵乘法计算生成多种版本代码，从中选出最优者。为了展示XMAG代码生成系统的优势，我们在本节测试不同优化选项组合对性能的影响。以$M=N=9000$，$K=48$的DGEMM函数为例，我们从调优结果中，选取了表4所列的不同代码版本进行比较。

表中第一行是采用与xMath2.0相同的计算方案和分块大小，即MN2+迭代模式，$PM=64$，$PN=64$，$PK=32$，性能为101.5 Gflops。在此种计算方案中，通过改变分块大小，得到性能最优的版本为$PM=128$，$PN=32$，$PK=48$，性能达到158.7Gflops（表4第二行）。而XMAG对此规模还生成了M1N+块模式的计算方案。此种计算方案在$PM=128$，$PN=48$，$PK=48$时获得最佳性能，达到165.3Gflops（表4第三行）。而采用不同的补齐方式（在数据DMA读取之后立即打包）的版本，达到了最优性能：168.8Gflops（表4第四行）。

表4　不同优化技术对性能的影响（矩阵乘法维度：$M=9000$，$N=9000$，$K=48$）

计算方案+参数	性能（Gflops）
pLoop={8,8,1}, *pA*={8,8}, *pB*={8,8}, *pC*={8,8}, 迭代模式，*PM*=64, *PN*=64, *PK*=32, kernelPadding=1	101.5
pLoop={8,8,1}, *pA*={8,8}, *pB*={8,8}, *pC*={8,8}, 迭代模式，*PM*=128, *PN*=32, *PK*=48, kernelPadding=1	158.7
pLoop={64,1,1}, *pA*={64,1}, *pB*={1,64}, *pC*={64,1}, 块模式，*PM*=128, *PN*=48, *PK*=48, kernelPadding=1	165.3
pLoop={64,1,1}, *pA*={64,1}, *pB*={1,64}, *pC*={64,1}, 块模式，*PM*=128, *PN*=48, *PK*=48, kernelPadding=0	168.8

7　结　论

本文介绍了面向SW26010-Pro处理器的矩阵乘法生成框架XMAG，这是我们设计实现的一套全流程的代码生成系统。我们通过设定任务划分和数据移动方案，设计优化变换方法，生成汇编计算核心，形成了完备而高效的代码生成框架。使用XMAG，我们能够为SW26010-Pro处理器生成与手工优化性能相当甚至更好的矩阵乘法代码。

对未来工作，我们有如下计划：首先，我们希望支持更多维度的张量，例如深度学习和计算化学应用中所使用的张量[38-39]，支持更多数据类型，以及批量矩阵乘法。其次，我们准备对XMAG进行改进，使之能够生成更加复杂的融合函数，例如QR分解算法中用到的larfb函数[40]。此外，我们还准备使XMAG支持更多计算模式代码的生成，例如，可以通过引入解三角方程组的计算核心和更多的依赖控制，为trsm函数生成代码。我们还将研究更加精确和快速计算的性能模型，以便直接选择最优方案和参数或对搜索空间进行进一步裁剪。最后，我们将研究XMAG向其他处理器的扩展，使之具有更高的通用性和可扩展性。

参 考 文 献

[1] NVIDIA H100 Tensor Core GPU architecture overview. https://resources.nvidia.com/en-us-tensor-core. 2024

[2] Talpes E, Williams D, Sarma D D. DOJO: The microarchitecture of Tesla's Exa-scale computer//Proceddings of IEEE Hot Chips 34 Symposium (HCS). Cupertino, USA, 2022: 1-28

[3] Hu Yi. Performance Optimization Techniques of BLAS library on domestic SW26010P many-core processors. University of Chinese Academy of Sciences, Beijing, 2023 (in Chinese)
(胡怡. 面向国产 SW26010P 众核处理器的 BLAS 库性能优化技术研究. 中国科学院大学,北京,2023)

[4] Hu Yi, Chen Dao-kun, Yang Chao, Ma Wen-jing, Liu Fang-fang, Song Chao-bo, Sun Qiang, Shi Jun-da. Many-core parallel optimization of level-3 BLAS functions on domestic SW26010-Pro processors. Journal of Software, 2024, 35(3): 1569-1584 (in Chinese)
(胡怡, 陈道琨, 杨超, 马文静, 刘芳芳, 宋超博, 孙强, 史俊达. 国产SW26010-Pro处理器上3级BLAS函数众核并行优化. 软件学报, 2024, 35(3): 1569-1584. http://www.jos.org.cn/1000-9825/6811.htm)

[5] Agarwal R C, Gustavson F G, Zubair M. A high-performance matrix-multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. IBM Journal of Research and Development, 1994,38,(6): 673-681

[6] Choi Jaeyoung, Dongarra Jack J, Walker David W. LAPACK Working Note 57: PUMMA: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. Technical Report. USA. 1993

[7] Kwasniewski Grzegorz, Kabić Marko, Besta Maciej, VandeVondele Joost, Solcà Raffaele, Hoefler Torsten. Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. Denver, Colorado, 2019: 1-22

[8] Liu Fangfang, Ma Wenjing, Zhao Yuwen, Chen Daokun, Hu Yi, Lu Qinglin, Yin WanWang, Yuan Xinhui, Jiang Lijuan, Yan Hao, Li Min, Wang Hongsen, Wang Xinyu, Yang Chao. xMath2.0: A high-performance extended math library for SW26010-Pro many-core processor. CCF Transactions on High Performance Computing, 2023,5(1): 56-71

[9] Tao X, Zhu Y, Wang B, Xu J, Pang J, Zhao J. Automatically generating high-performance matrix multiplication kernels on the latest Sunway processor//Proceedings of the 51st International Conference on Parallel Processing. Bordeaux. France. 2022: 1-12

[10] Xu Le, An Hong, Chen Junshi, Zhang Pengfei. High-performance matrix multiplication on the new generation Shenwei processor//Proceedings of IEEE 24th International Conference on High Performance Computing & Communications; 8th International Conference on Data Science & Systems; 20th International Conference on Smart City; 8th Int Conference on Dependability in Sensor, Cloud & Big Data Systems & Application. Hainan, China, 2022: 894-903

[11] OpenBLAS: An optimized BLAS library. https://www.openblas.net/. 2023

[12] Goto Kazushige and Van De Geijn Robert A. 2008. Anatomy of high-performance matrix multiplication. ACM Transactions On Mathematical Software, 2008,34(3): 1-25

[13] Van Zee F G, Van De Geijn R A. BLIS: A framework for rapidly instantiating BLAS functionality. ACM Transactions On Mathematical Software, 2015,41(3): 1-33

[14] Wei Cunyang, Jia Haipeng, Zhang Yunquan, Yao Jianyu, Li Chendi, Cao Wenxuan. IrGEMM: An input-aware tuning framework for irregular GEMM on ARM and X86 CPUs. IEEE Transactions on Parallel and Distributed Systems, 2024, 35(9): 1672-1689

[15] Ragan-Kelley Jonathan, Barnes Connelly, Adams Andrew, Paris Sylvain, Durand Frédo, Amarasinghe Saman. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines//Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. Seattle, USA, 2013: 519-530

[16] Adams Andrew, Ma Karima, Anderson Luke, Baghdadi Riyadh, Li Tzu-Mao, Gharbi Michaël, Steiner Benoit, Johnson Steven, Fatahalian Kayvon, Durand Frédo, Ragan-Kelley Jonathan. 2019. Learning to optimize Halide with tree search and random programs. ACM Transactions on Graphics , 2019,38(4):121:1-121:12

[17] Li Tzu-Mao, Gharbi Michaël, Adams Andrew, Durand Frédo, Ragan-Kelley Jonathan. Differentiable programming for image processing and deep learning in Halide. ACM Transactions on Graphics ,2018,37(4):1-13

[18] Mullapudi Ravi Teja, Adams Andrew, Sharlet Dillon, Ragan-Kelley Jonathan, Fatahalian Kayvon. Automatically scheduling Halide image processing pipelines. ACM Transactions on Graphics,2016, 35(4):1-11

[19] Chen Tianqi, Moreau Thierry, Jiang Ziheng, Zheng Lianmin, Yan Eddie, Cowan Meghan, Shen Haichen, Wang Leyuan, Hu Yuwei, LuisCeze, CarlosGuestrin, Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning//Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation USA, 2018: 579-594

[20] Zheng L, Jia C, Sun M, Wu Z, Yu H, Haj-Ali A, Wang Y, Yang J, Zhuo D, Sen K, Gonzalez J E, Stoica I. Ansor: generating high-performance tensor programs for deep learning//Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation. online, 2020: 863-879

[21] Ikarashi Yuka, Bernstein Gilbert Louis, Reinking Alex, Genc Hasan, Ragan-Kelley Jonathan. Exocompilation for productive programming of hardware accelerators//Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. San Diego, USA, 2022: 703-718

[22] Grosser Tobias, Verdoolaege Sven, Cohen Albert. Polyhedral AST generation is more than scanning polyhedra. ACM

Transactions on Programming Languages and Systems, 2015, 37(4):1-50

[23] Baghdadi R, Ray J, Romdhane M B, Sozzo E D, Akkas A, Zhang Y, Suriana P, Kamil S, Amarasinghe S. Tiramisu: A Polyhedral compiler for expressing fast and portable code. https://doi.org/10.48550/arXiv.1804.10694 arXiv:1804.10694. 2018

[24] Verdoolaege S, Carlos Juega J, Cohen A, Gómez J I, Tenllado C, Catthoor F. Polyhedral parallel code generation for CUDA. ACM Transactions on Architecture and Code Optimization, 2013, 9(4):1-23

[25] Su Xing, Liao Xiangke, Xue Jingling. Automatic generation of fast BLAS3-GEMM: a portable compiler approach// Proceedings of the 2017 International Symposium on Code Generation and Optimization . Austin, USA, 2017: 122-133

[26] Wang Qian, Zhang Xianyi, Zhang Yunquan, Yi Qing. AUGEM: Automatically generate high performance dense linear algebra kernels on X86 CPUs//Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. Denver, USA, 2013: 1-12

[27] Tanner. Tensile: Auto-Tuning GEMM GPU Assembly for All Problem Sizes//Proceedings of IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). Vancouver, Canada, 2018:1066-1075

[28] Feng Siyuan, Hou Bohan, Jin Hongyi, Lin Wuwei, Shao Junru, Lai Ruihang, Ye Zihao, Zheng Lianmin, Yu Cody Hao, Yu Yong, Chen Tianqi. TensorIR: An abstraction for automatic tensorized program optimization//Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. Vancouver, Canada, 2023: 804-817

[29] Hagedorn Bastian, Fan Bin, Chen Hanfeng , Cecka Cris, Garland Michael, Grover Vinod. Graphene: An IR for optimized tensor Computations on GPUs//Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3. Vancouver, Canada, 2023: 302-313

[30] Tillet P, Kung H T, Cox D. Triton: An intermediate language and compiler for tiled neural network computations// Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages. Phoenix, USA, 2019: 10-19

[31] Kerr Andrew, Merrill Duane, Demouth Julien, Tran John. CUTLASS: Fast linear algebra in CUDA C++ . https://developer.nvidia.com/blog/cutlasslinear-algebra-cuda/. 2017

[32] Developing CUDA kernels for accelerated matrix multiplication on NVIDIA Hopper architecture using the CUTLASS library. https://research.colfax-intl.com/wp-content/uploads/2023/12/colfax-gemm-kernels-hopper.pdf. 2023

[33] H100_GEMM. https://github. com/Faraz9877/H100_GEMM. 2024

[34] Outperforming cuBLAS on H100: a Worklog. https://cudaforfun. substack. com/p/outperforming-cublas-on-h100-a-worklog. 2024

[35] Liu Changxi, Yang Hailong, Sun Rujun, Luan Zhongzhi, Gan Lin, Yang Guangwen, Qian Depei. swTVM: Exploring the automated compilation for deep learning on Sunway architecture. ArXiv abs/1904.07404,2019

[36] Gao Wei, Fang Jiarui, Zhao Wenlai, Yang Jinzhe, Wang Long, Gan Lin, Fu Haohuan, Yang Guangwen. swATOP: Automatically optimizing deep learning operators on SW26010 many-core processor// Proceedings of the 48th International Conference on Parallel Processing. Kyoto Japan, 2019:1-10

[37] Xu Le, An Hong, Chen Junshi, Zhang Pengfei. High-performance matrix multiplication on the new generation Shenwei processor//Proceedings of IEEE 24th International Conference on High Performance Computing & Communications; 8th International Conference on Data Science & Systems; 20th International Conference on Smart City; 8th International Conference on Dependability in Sensor, Cloud & Big Data Systems & Application. Hainan, China, 2022: 894-903

[38] Feng Siyuan, Hou Bohan, Jin Hongyi, Lin Wuwei, Shao Junru, Lai Ruihang, Ye Zihao, Zheng Lianmin, Yu Cody Hao, Yu Yong, Chen Tianqi. TensorIR: An abstraction for automatic tensorized program optimization//Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. Vancouver, Canada, 2023: 804-817

[39] Ma Wenjing, Krishnamoorthy Sriram, Villa Oreste, Kowalski Karol, Agrawal Gagan. Optimizing tensor contraction expressions for hybrid CPU-GPU execution. Cluster Computing. 2013,16(1):131-155

[40] Ma Wenjing, Liu Fangfang, Chen Daokun, Lu Qinglin, Hu Yi, Wang Hongsen, Yuan Xinhui. An optimized framework for matrix factorization on the new Sunway many-core platform. ACM Transactions on Architecture and Code Optimization. 2023,20:1-24

YAN Yu-Cheng, M. S., engineer. His research interests include high-performance computing, and high performance code generation.

MA Wen-Jing, Ph. D., associate professor. Her research interests include high-performance computing and high performance code generation.

LIU Fang-Fang, Ph. D., senior advanced engineer. Her research interests include high-performance computing, parallel computation on heterogeneous platforms, and supercomputer evaluation software.

HU Li-Juan, M. S., assistant engineer. Her research interests include high-performance computing, parallel computation on heterogeneous platforms, and BLAS library related algorithm optimization.

LI Fang, Ph. D., professor. Her research interests include high performance computing applications and related optimization technique.

Background

Dense matrix multiplication is a hot spot of research, not only because its performance is critical to many applications in science and engineering computing, as well as artificial intelligence, but also because its optimization is closely related to hardware features and the optimization varies according to matrix shape and size. Therefore, a tempting solution is using code generator to generate a number of codes and select the most favorable one for a certain input. This technique raises challenge especially on heterogeneous many-core platforms, Among which a new trend is using on-chip network and distributed scratch-pad memory.

The SW26010-Pro processor is a typical platform with this feature. To obtain satisfactory performance on GEMM with various shapes, it is inevitable to figure out a way to construct a code generation system for this processor. The existing frameworks, which target either CPUs or GPUs, cannot handle the complex DMA and RMA operations with flexible software pipelines. The only two code generator that works for SW26010-Pro both suffer greatly from their limitation of implementation, and cannot generate the deeply optimized code for GEMM with various shapes and sizes.

Therefore, we propose XMAG, a code generation framework for GEMM on SW26010-Pro. We designed the whole system in a hierarchical fashion, with the higher level defining the computation schemes, the middle level generating and transforming the ASTs, and the last level generating C code that can compile and run on the CPE mesh of the processor. Based on the analysis of optimization techniques used in the current xMath2.0 library and other implementation, we incorporate loop mapping, matrix mapping, and data exchange pattern in the computation schemes, and provide a variety of software pipeline generation strategies to ensure the coverage of the optimization suitable for matrices of various sizes and shapes. We also developed a code generator for the assembly computation kernels, which further provides flexibility to the code. Together with an auto-tuning framework, we are able to generate and find the best solution for GEMM with a large variety of matrix sizes and shapes, achieving an average speedup of 2.32 over xMath2.0, which is a library deeply optimized for SW26010-Pro.

This work has been supported by the National Key R&D Program of China (2023YFB3001703)。