

硕士学位论文

**一个基于MLIR针对定制DSP的编译方法**

学位申请人：丁顺顺

指导教师：赵银亮教授

学科名称：软件工程

2023年02月

摘 要

随着人工智能的发展，最近几年，AI呈现出了爆发趋势，使用通用处理器这个传统的方法已无法满足人工智能的各种应用对高并发和高计算能力的需求，因此，具有GPU、DSP、ASIC、 FPGA 或其它加速器（Accelerator）等高并行、高密集的计算能力的异构计算将成为未来的发展趋势。为了解决软件碎片化问题，改进异构计算的编译，显著降低构建特定领域编译器的成本，一种构建可重用和可扩展编译器基础结构的新方法MLIR编译器框架被提出[1]。本文将基于MLIR编译器框架，提出一种针对定制DSP的编译方法。

本文根据DSP的硬件架构，基于MLIR编译器框架构建了DSP Dialect，并且针对该DSP的向量计算模块和矩阵计算模块构建了两条从高层IR转换到DSP Dialect的下降管道。其中，DSP Dialect是一种指根据硬件架构和硬件指令的抽象描述，是生成特定硬件架构指令的桥梁。通过构建的两条下降管道，可以生成该DSP的向量计算模块和矩阵计算模块的LLVM IR,然后在该DSP上执行可以得到MLIR编译方式在DSP上的性能。本文完成的DSP编译器复用了MLIR提供的词法器和语法分析器，以及DSP的静态编译器。

经过实验测试，本文所实现的DSP 编译器编译出的可执行文件在该DSP的模拟器（Simulator）上可正常执行，并验证了程序运行结果的正确性。本文最后通过两个向量乘法和矩阵乘法与现有的手写高性能DSP算子库比较，我们发现该DSP编译器自动化生成的可执行文件没有现有算子库的复杂调用过程和VMEM分配流程，最终执行的性能与现有的手写高性能算子库的性能不相上下，甚至在向量长度很大的时候，其性能略高于现有的手写算子库。

**关 键 词：**异构计算；MLIR; DSP; 编译方法

ABSTRACT

With the development of artificial intelligence, in recent years, AI has shown an explosive trend. The traditional method of using general-purpose processors can no longer meet the needs of various applications of artificial intelligence for high concurrency and high computing power. Therefore, with GPU, Heterogeneous computing with high parallelism and high-intensity computing capabilities such as DSP, ASIC, FPGA or other accelerators (Accelerator) will become a future development trend. In order to solve the problem of software fragmentation, improve the compilation of heterogeneous computing, and significantly reduce the cost of building domain-specific compilers, a new method of building reusable and extensible compiler infrastructure MLIR compiler framework is proposed [1]. Based on the MLIR compiler framework, this paper proposes a compilation method for customizing specific DSP .

According to the hardware architecture of DSP, this paper builds DSP Dialect based on MLIR compiler framework, realizes the automatic allocation of DSP VMEM, and completes a small DSP compiler from front-end to back-end. Among them, DSP Dialect is an abstract description based on hardware architecture and hardware instructions, and is a bridge to generate specific hardware architecture instructions. The automatic allocation of DSP VMEM is an improvement and optimization of the system, because in the process of MLIR decline (Lower), it is extremely inconvenient to manually arrange the storage modules inside the DSP, and it is easy to cause read and write conflicts. The automatic distribution of DSP VMEM solves this problem. The DSP compiler completed in this paper reuses the lexer and syntax analyzer given in the MLIR tutorial, as well as the DSP back-end compiler.

After experimental testing, the executable file compiled by the DSP compiler implemented in this paper can be executed normally on the DSP simulator (Simulator), and the correctness of the program running results has been verified. At the end of this paper, by comparing the two vector multiplication and matrix multiplication with the existing handwritten high-performance DSP operator library, we found that the executable file automatically generated by the DSP compiler does not have the complicated calling process and VMEM allocation process of the existing operator library. The performance of the final execution is comparable to that of the existing handwritten high-performance operator library, and even when the vector length is large, its performance is slightly higher than that of the existing handwritten operator library.

**KEY WORDS**: heterogeneous computing; MLIR; DSP; compile method

目录

[摘 要 2](#_Toc10952)

[ABSTRACT 3](#_Toc19541)

[1 绪论 5](#_Toc12821)

[1.1 课题背景 5](#_Toc21629)

[1.1.1 特定硬件算子库的开发和维护愈发繁重 5](#_Toc22720)

[1.1.2 AI编译器和IR乱象 5](#_Toc5683)

[1.1.3 MLIR设计思想 7](#_Toc6417)

[1.1.4 DSP简介 8](#_Toc20739)

[1.2 国内外相关研究 8](#_Toc20732)

[1.2.1 基于MLIR生成GEMM算法高性能GPU代码 9](#_Toc23897)

[1.2.2 Buddy Compiler 10](#_Toc18520)

[1.3 研究目标和内容 10](#_Toc30791)

[1.4 论文组织结构 10](#_Toc29222)

# 绪论

## 课题背景

近年来，随着深度学习的火热，以及互联网发展所积累下的大量数据，使得传统的通用处理器已经无法满足对算力需求的指数级增长。因此，通用处理器加各种协处理器的异构计算将是挑起人工智能大梁的最优选择。处理器也将朝着多元化方向发展，如CPU偏向于整个SOC的控制，在异构系统系统中承担指挥统筹，控制核心的功能；GPU适用于大范围、多任务的简单运算；加速器（Accelerator）适用于特定领域的加速运算；DSP适用于大量复杂算法的实时性运算。而将深度学习算法和人工智能训练、部署到如此众多的异构架构的硬件上，并将硬件资源最大程度的利用，同时保证算法的可维护性和复用性，这无疑是一个巨大的挑战。

### 特定硬件算子库的开发和维护愈发繁重

在如今处理器百花齐放的时代，各大硬件厂商都在打造自己的硬件。通常在使用或推广自己的硬件加速器的时候，硬件厂商会有针对性的发布算子库，如英特尔推出的DNNL,是一款针对英特尔架构处理器和显卡的面向深度学习应用的高性能算子库。NVIDIA推出的cuDNN，是针对深度神经网络的GPU加速库。这些算子库有一个很明显的特点，那就是和当前的硬件深度绑定，各个算子库之间的优化无法进行互相融合、复用。每当有一个新的算子被提出时，也只能由各个厂商的算子开发工程师去实现，同时还要结合硬件特性进行手动调优测试，以尽可能的充分发挥硬件特性。而越来越多的新算子被提出，算子库的开发、维护、优化和测试工作量将呈指数上升。

另一方面，各种各样的专用加速器的爆发导致算子的可移植性成为一种刚需。大多数NPU使用ASIC,在神经网络场景对计算、存储和数据搬运做了特殊的指令优化，使得对AI相关的计算会提升性能，如NVIDIA虽然作为GPGPU,但是DGX系列提供专用的Tensor Code。不同的硬件厂商提供XPU的ISA(Instruction Set Architecture)各不相同，一般都缺乏如GCC/LLVM等编译工具链，使得针对CPU和GPU已有的优化算子库和针对语言优化的PASS很难在短期内移植到NPU上。

于是，一个算子开发和新硬件算子适配的组合爆炸的问题便出现了。一方面，随着深度学习和人工智能的发展，不断有新算子将会被提出，这些算子将需要在已有和未有的一些硬件上去开发实现；另一方面，各式各样的基于不同ISA的新硬件层出不穷，这些新硬件也将不断的开发已有的算子和新算子。造成这个局面的根本原因在于不同硬件之间的算子库不能互相融合和复用，已有的算子库在一个新硬件上的可移植性不强，算子库之间无法做到优化复用，只能重新开发。这无疑增加了很多不必要的成本。

### AI编译器和IR乱象

为了解决算子开发和新硬件算子适配的问题，研究者从编译角度提出了AI编译器。AI编译器并不是通过AI技术去替代传统编译器的某些步骤，而是一种给AI模型用的编译器。

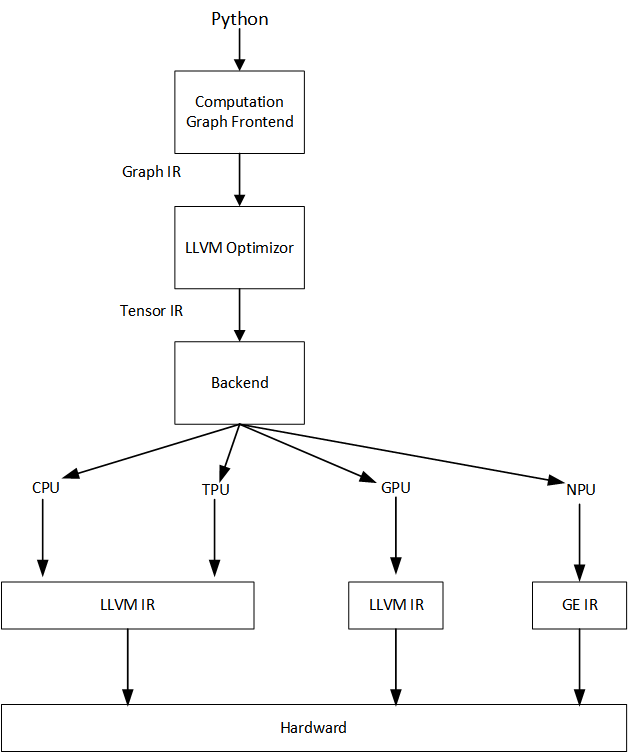


图1 AI编译器流程图

如图一，AI编译器是一种面向神经网络、深度学习特定优化的编译器。一般以Python为主的动态解释器语言作为前端，前端会将输入的深度学习模型转化为计算图 IR。中端的优化器一般使用多层IR设计，对不同层级的抽象做一些优化，如在计算图上做的自动微分，并行切分，剪枝量化，算子融合等。后端即CodeGen部分则需要具体到特定的硬件架构，针对具体的硬件特性，自动生成其高性能代码或IR，最后由传统编译器将生成的代码或IR编译成可执行程序执行。

为了便于优化，AI编译器会把输入的模型转换为IR（中间表示）,然后对IR进行优化和变换，最终生成特定硬件的高性能代码或IR。其中，不同的AI编译器都有自己的IR和优化方案，如TVM将各种深度学习训练框架的模型先转化为Relay IR，Relay的目标是可以接入各种前端给出的模型，然后做算子间的融合，最后由TVM生成低层的TIR；XLA将TensorFlow模型转换为HLO IR,将图层下发的子图中的算子全部打开成小算子，基于小算子组成的子图进行编译优化，然后使用LLVM编译生成LLVM IR；oneDNN将Graph IR转换为Tensor IR进行优化编译。尽管其中很多AI编译器做了多层IR设计，但是在众多IR之间的转化仍然是比较困难的。

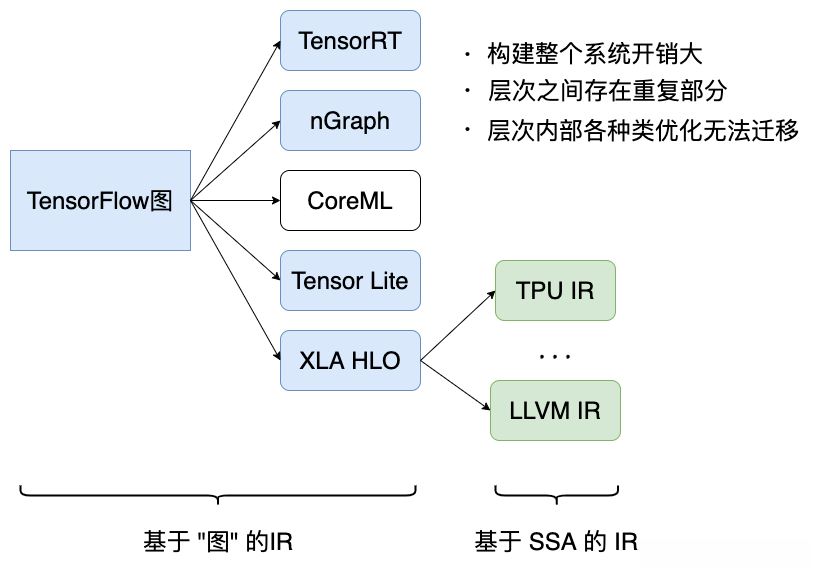


图2 TensorFlow编译流程

如图2，以TensorFlow为例，TensorFlow使用数据流图作为数据结构来进行各种数值计算，将TensorFlow模型转换为了XLA HLO后进行了一些High-Level的优化，接着就将HLO编译成了LLVM IR，使用LLVM编译到各种硬件的汇编语言。图2中左侧部分是基于图的IR,右侧是基于SSA(static single-assignment,简单静态单一赋值)的IR。这样的编译方式的缺点在于构建整个编译系统的开销大，每一层的IR设计实现会有重复部分，同一个层次的IR之间彼此相似，但无法做到相互转换，也就无法复用其优化内容，IR之间的转换效率和可迁移性不高，每种IR的优化方案可能都需要实现一遍，造成了软件碎片化和重复的开发工作。

### MLIR设计思想

MLIR（Multi-Level Intermediate Representation，多级中间表示）是Chris Lattner在谷歌时提出的编译器框架，是一种用来构建可重用和可扩展编译基础设施的新方法。MLIR旨在解决软件碎片化，改进异构硬件的编译，显著减少构建特定领域编译器的成本，并帮助连接现有的编译器[1]。与其他推理引擎不同，MLIR的关注点在于统一众多的中间表示（IR）,推出一个可由用户自定义、扩展、复用转换IR的平台。如今是处理器百花齐放的时代，也是各种IR层出不穷的时代。从较为接近硬件汇编的IR到各种计算图的IR,多层IR的设计既满足了不同层级的抽象程度，又方便在不同层级上进行优化，所以多层IR设计是必要的。为了能够满足各种IR的需求，MLIR引入Dialect(方言)这一概念去描述IR,方言由Operations(操作)，Attributes(属性)和Types(类型)组成。

MLIR的方言也是多层级的，即每一种方言对应描述一种IR。如MLIR提供的接近各种硬件的方言:GPU Dialect,arm\_neon Dialect,x86vector Dialect...以及为了方便生成LLVM IR的LLVM Dialect；有方便操作控制流的scf Dialect,standard Dialect；有方便密集计算的affine Dialect,tensor Dialect,vector Dialect。MLIR的方言并不是一成不变的，可由用户按照方言的设计原则来新增和扩展，不同的方言对应不同的IR,也就对应了不同的抽象层级。从前端的计算图到后端的硬件架构，不同的抽象层级可以由不同的方言去描述，而相同的抽象层级也可以由不同的方言去描述，但是同层级的优化是相似的，基于MLIR的可扩展性和可复用性，这些优化完全是可复用、可扩展的，这些就是MLIR的优势所在。MLIR不但可以做到相同抽象层级IR优化的复用，也打通了不同抽象层级IR之间的转换：MLIR提供了一整套的基础设施可以让方言之间更加标准的转换，这个过程称之为Lower(下降)。Lower不一定是高层级抽象的方言向低层级方言的转换，准确来说任何层级的方言其Lower后的Operation是标记为合法的，都可以进行Lower。不仅如此，MLIR还支持混合方言的IR输出，也就是说从一个方言到另外一个方言的转换中，可以不让所有的Operations都进行Lower,可以标记某些Operations是合法的，在之后的Lower处理。MLIR通过方言来管理IR,这就意味着如今各种各样的IR都可以通过这种方式来快速的接入MLIR和LLVM生态，增加了对各类IR的兼容性。

### DSP简介

本文基于大疆创新自研DSP硬件及其模拟器、工具链等进行实验。该DSP主要为图像、视觉等应用提供一个高性能计算平台，它采用哈佛架构，即将程序指令存储和数据存储分开。DSP中包含一个ARV RISC-V核负责程序流控制核标量运算，ISA采用32bit编码，支持RISC-V 64 IMAF\_Zicsr\_Zifenci指令集，使用L1 Cache加快程序和数据的读取。除RISC-V主核外，DSP还有一个向量运算单元VEU,一个浮点向量计算单元VFPU,一个向量逻辑计算单元VALU,这三个向量运算相关的计算单元简称为VXU。除此之外DSP还有一个由脉动阵列组成的矩阵运算单元MMX,一个VMEM单元提供向量存储，一个运算加速器提供CORDIC和直方图运算，4个定时器和一个UART，一个专门用于数据和程序片内片外之间搬运的模块DMA。

DSP内运算单元和存储单元之间通过512比特的Crossbar互联，各个运算单元有独立的指令流水线执行并行运算，可以与VMEM存储系统之间的数据交互并行进行。DSP作为协处理器没有操作系统和文件系统，VXU和MMX只能处理DSP内部数据存储系统VMEM里的数据，VMEM由10块array组成，每块array 64kbytes，为了满足各种算法计算的需求，VMEM存储系统支持各种复杂存取数据的方式：线性取值、循环取值、自定义取值等。整个DSP由RISC-V主核控制、协调各模块之间运行。

## 国内外相关研究

自从2019年MLIR被提出之后，在学术界就被广泛关注，LLVM和MLIR社区也持续火热。各方主流硬件的方言逐渐加入完善，中科院软件所的张宏斌同学向社区提出了RISCVV方言尤为关注。同时MLIR社区给出了一个在GPU上使用MLIR编译器基础设施框架来优化GEMM算法的成功案例，该研究结果表明:使用MLIR编译器基础设施框架优化GEMM算法生成的高性能GPU代码，性能和cuBLAS持平。

### 基于MLIR生成GEMM算法高性能GPU代码

当前高性能深度学习的算子库通常由专业的程序员在low-level的级别进行手工优化和调优，为此付出了很大的努力。来自PolyMage Labs和印度理工学院的团队使用MLIR编译器基础架构针对NVIDIA GPU上的Tensor Core自动生成高性能代码。

该团队在MLIR Dialect中引入了Warp Matrix Multiply Accumulate（WMMA） Operation,针对NVIDIA Ampere架构的GPU Tensor Core单元，这个单元是用于矩阵乘累加（MMA）操作的专用单元，其吞吐量通常是普通CUDA核心的3-4倍。然后对WMMA Operation进行了一系列的Lower,其过程如下：

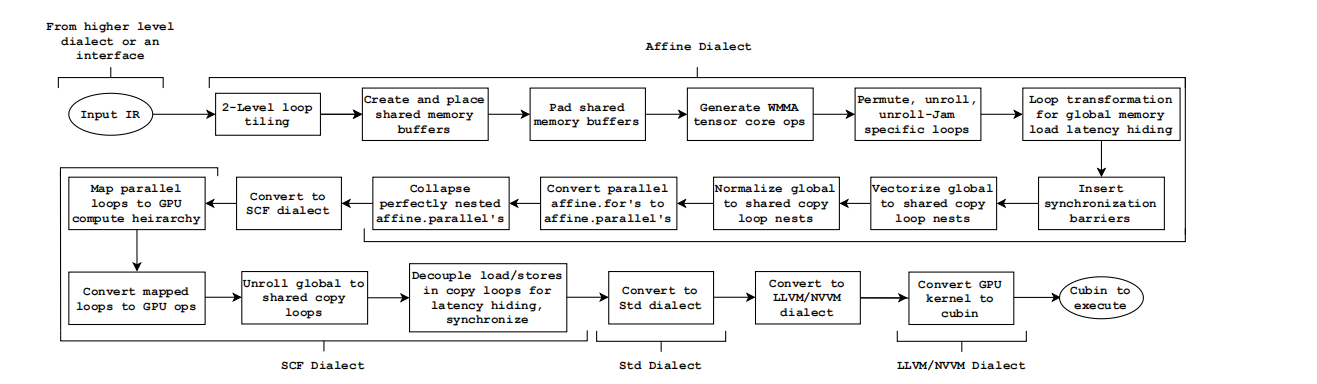


图3 WMMA Lower过程

1. 首先将higher level 的Dialect或矩阵乘Lower到Affine Dialect，因为生成的目标Kernel是仿射的，可以在Affine Dialect里使用很多MLIR已经提供好的优化模块，比如快速内存缓冲区的创建和放置、loop-tiling、unroll-jam、矢量化、并行循环的检测以及同步barriers的放置等。
2. 将Affine Dialect中组合优化后的IR转换到SCF Dialect。SCF Dialect开始后做的第一件事情就是将并行循环映射到GPU计算层次结构，因为MLIR中用于映射的现有实用程序和pass不支持将循环映射到单个warp,该团队扩展了实用程序和pass以添加对矩阵乘的支持。然后做了load和store解耦，完成了Global Memoey加载延迟隐藏。
3. 将host端的代码转换为std Dialect然后转换为LLVM Dialect。期间，来自GPU的Operation将通过MLIR CUDA运行时API函数将其Lower为LLVM IR，并生成目标代码。Device端的代码也被转换为std Dialect,然后转换为LLVM Dialect和NNVM Dialect的混合。然后由LLVM的NVPTX后端转换为PTX,使用NVIDIA的编译器将PTX转换为cubin。

该团队通过上述过程构建了针对Tensor Core的端到端matmul代码生成管道，初步的结果表明，在混合精度和半精度浮点上性能获得的性能非常接近CuBLAS 11.2，在一些较小的尺寸上，该性能略高于cuBLAS。

该文章基于MLIR做了一个关于矩阵乘法比较完整的代码生成工作，给MLIR的使用做了一个友好的示例。它证明了MLIR编译器基础设施框架在矩阵乘这种计算密集型算子上基于特定领域加速器的性能是可以做到与硬件厂商提供的高性能算子库持平的，在某些情况下甚至略高于cuBLAS库。由于该文章是针对GPU Tensor Core单元构建了一个矩阵乘法的Lower管道，而不是针对某一款GPU和算法做了完整的构建，所以构建的Lower管道的针对性比较高，不具备完整性和系统性。

### Buddy Compiler

随着特定领域的体系结构和语言的发展，对编译器的需求不断增长，编译器技术的研究进入了黄金时代。Buddy Compiler是由张宏斌发起的一个基于MLIR建立的编译器基础设施，Buddy Compiler的目标是提供一个框架来帮助用户快速实现特定领域编译器，包括DSL前端支持、IR级优化和DSA后端代码生成。

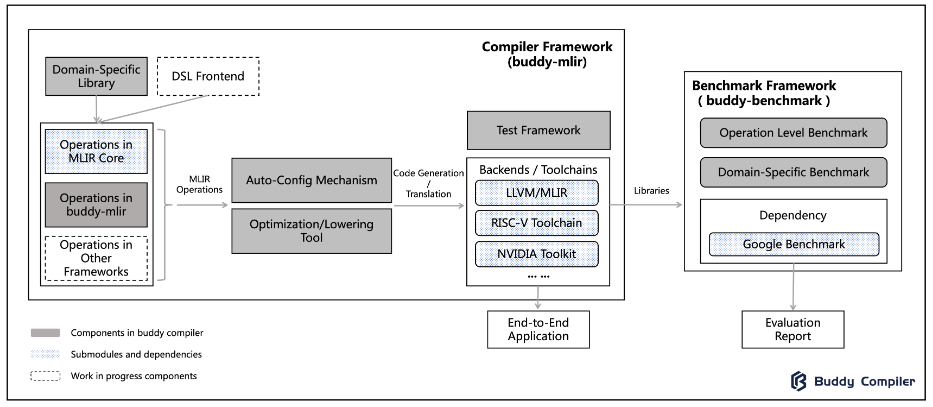


图4 Buddy compiler框架

Buddy compiler集成了LLVM项目，重用了MLIR基础设施和LLVM后端工具，其主要包含了buddy-mlir和buddy-benchmark两个模块。Buddy-mlir以MLIR为基石，在此基础上构建特定领域编译器。通过DSL前端将输入模型转换到MLIR、buddy-mlir和其他框架的Operations上，通过优化工具和自动配置工具对其进行优化Lower,生成对应后端的IR。Buddy-mlir还构建了Bud Dialect和DIP Dialect。Bud Dialect主要是为了测试和演示而设计，DIP Dialect是为了数字图像处理抽象而设计的。Buddy-benchmark提供了一些内置的测试用例来对比性能，可以进行Operation级别的对比，也可以进行领域特定计算库/框架级别的对比。同时支持自定义扩展，可以由用户自定义实现添加到benchmark中一起对比。Buddy-benchmark依赖Google Benchmark检测运行时间并生成性能报告。

Buddy-compiler甩掉LLVM/MLIR庞大的历史包袱，聚焦于特定领域编译器的快速构建，提供了数字图像处理抽象的方言和根据硬件自动装配优化PASS的工具，并且支持各种粒度的测试对比。目前Buddy-compiler还在原始的初期阶段，其DSL前端还在开发之中，接入的后端方言也只有RISCV Dialect,从前端到后端还没有系统性的集成，期待其后续发展。

## 研究目标和内容

基于现在构建一个从前端到后端特定领域编译器需要付出巨大努力的现状，本文将使用MLIR编译器基础设施框架来针对某公司自研DSP构建DSP Dialect,以及针对该DSP的向量计算模块VEU和矩阵计算模块MMX构建两条能自动生成向量加、向量减、向量乘以及矩阵乘的高性能代码的下降转换管道，为该公司的深度学习编译器接入DSP做准备。

本文首先基于某公司自研DSP的架构，构建了DSP Dialect。DSP Dialect是偏向于硬件的Dialect,主要作用是生成该类型DSP架构的LLVM IR,同时在MLIR中所有的Dialect最终都将Lower到最底层的LLVM Dialect，所以DSP Dialect在MLIR中起到了承上启下的作用。为了上层Dialect的Lower和使用，DSP Dialect的设计原则是模块化，原子化。它是根据DSP中的硬件模块划分的，将每个硬件模块的功能拆分成最小粒度的原子操作，然后模块化的将每个原子操作组装成Dialect的Operation。模块化的作用是为了更好的复用、使用这些Operation。所以DSP Dialect主要包括：声明Dialect支持的数据类型；构建DSP 的Intrinsic Operations；为VMEM存储系统实现线性取指模式、循环取指模式、自定义寻址模式的Operations；为DMA搬运数据模块实现dma\_transfer\_1d和dma\_transfer\_2d搬运的Operations；VXU模块向量加、向量减、向量乘、向量除以及配置向量长度和控制寄存器的Operations；MMX模块矩阵乘以及配置矩阵大小和控制寄存器的Operations。

基于已经构建的DSP Dialect，然后就可以很方便的得到该DSP架构的LLVM IR，为了得到基于DSP Dialect编译生成的高性能代码的性能，本文构建了两条针对向量计算模块VEU和矩阵计算模块MMX的自动生成高性能代码的下降（Lowering）管道。然后复用了MLIR提供的词法器和语法分析器，由前端的IR Lower到DSP Dialect，生成接近DSP硬件的LLVM IR,最后由该公司提供的DSP工具链中的llc静态编译器将LLVM IR编译生成DSP可执行程序。

最后对DSP Dialect和构建的两条Lower 管道进行了功能测试和性能测试，通过与现有的工具链中的编译器生成的算子库做对比，证明了本文基于MLIR构建DSP Dialect的正确性和针对VEU以及MMX自动生成高性能代码的有效性。

## 论文组织结构

文章共包括六个章节，各章节内容安排如下：

1. 绪论，介绍课题研究背景、国内外相关研究、主要工作内容和论文组织结构。  
    第二章，相关理论与技术，介绍本文所使用到的理论与相关技术，主要有本文所用到的自研DSP的硬件架构、向量计算单元、矩阵计算单元、存储子系统、数据搬运模块，以及MLIR编译器基础设计框架、Dialect的组成和结构。

# 相关理论与技术

本章主要介绍本文所使用到的一些理论基础和相关技术，主要分为两部分进行介绍。首先是本文对DSP整体硬件架构、向量处理单元、存储单元VMEM以及负责数据搬运模块DMA进行介绍。之后对现有的MLIR编译器基础设施框架、Dialect、Dialect的组成和结构以及MLIR编译流程进行介绍。

## DSP硬件架构

本文基于某公司自研DSP进行Dialect的设计，自研DSP的整体架构图如图5所示

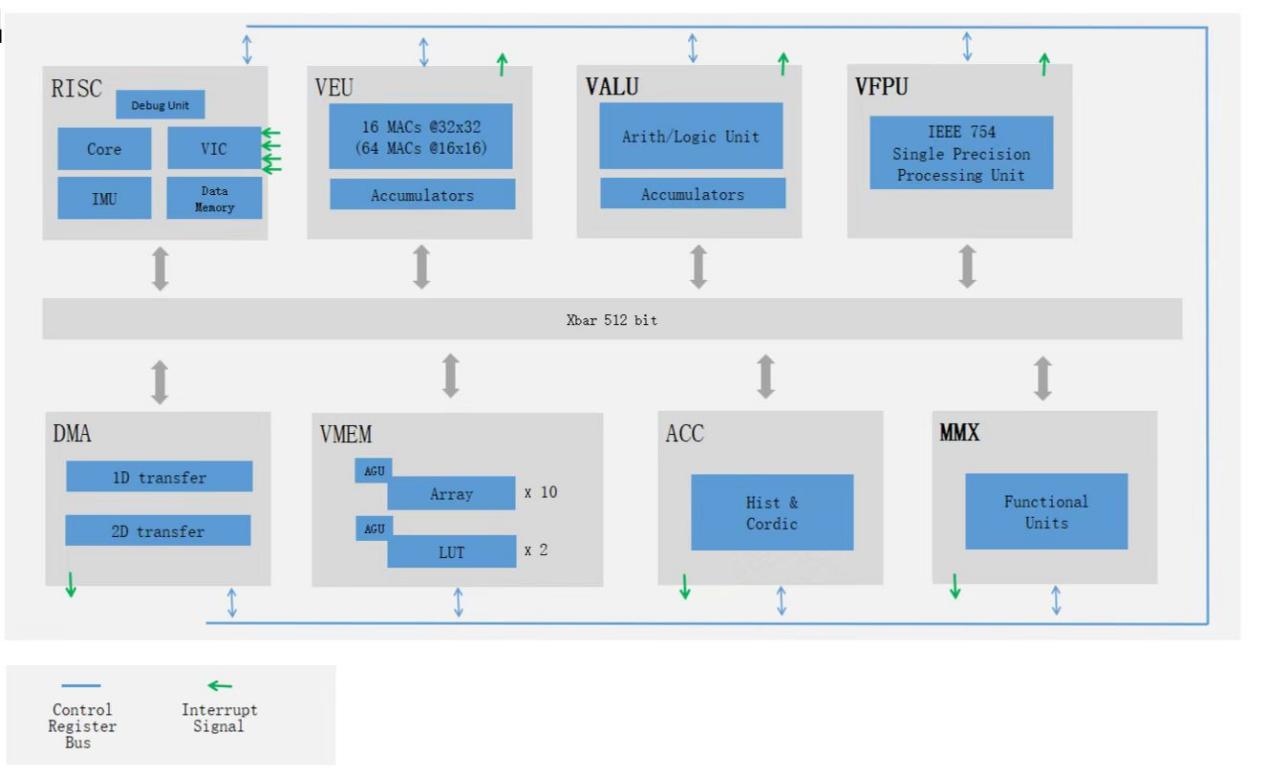


图5 DSP整体架构图

DSP中的RISC是一个RISCV核，是整个DSP的唯一主设备，其他模块都是从设备，由RISC负责整个DSP指令的获取和分发，RISC的执行是五级流水线。DSP内部各个组件之间既有数据互联也有控制互联，可分为指令总线、CRF总线、中断互联、Crossbar互联和组件间私有互联五类，这五种互联保证了DSP各个模块之间的指令控制和数据交互。

1. 指令总线：DSP由RISC获取指令，当RISC的IF级获取到RISC指令时，则让RISC去执行；当RISC获取到非RISC的指令时，会在EX阶段将它发送到指令总线上，其他连接到指令总线的执行单元根据指令编码的高3位获取归属于自己的指令并执行。
2. CRF总线：DSP中RISC作为唯一的主设备,通过CRF总线访问所有组件的控制寄存器，对各个组件的状态进行控制和获取。
3. 中断互联：DSP中各个组件通过中断信号与RISC进行同步或者上报错误。
4. Crossbar互联：除了RISC外的组件通过Crossbar与VMEM进行数据交换。
5. 组件间私有互联：DSP内部一些组件之间的私有互联，如RISC通过LSU访问VMEM等。

### Vector Execution Unit

VEU即Vector Execution Unit,是DSP系统中比较重要的向量运算单元之一。本文着重介绍VEU计算单元，向量逻辑运算单元VALU和浮点向量运算单元VFPU与之类似，不做重复介绍。

VEU只能计算存储在VMEM里的数据，其通过Crossbar与VMEM和DMA进行数据交换。VEU支持char,short,int数据类型的向量加，向量减，向量乘以及向量的累加运算。在使用VEU计算单元前，需要使用RISC对VEU的控制寄存器进行必要的配置。对于向量长度为N的运算，受限于有限的硬件资源，计算可能会持续多次才能完成，将每次运算的原子操作个数称为并行度，即op\_size。最大并行度是指指令一次可以进行的最多原子操作个数，它跟运算种类和数据类型都有关系，每运行一次op\_size个数据，将会花费1 cycle。op\_size位于控制寄存器CTRL0中，默认0即为最大并行度，在已知指令并行度的情况下，可以计算指令的运算耗时。一次完整的运算需要耗费m个cycles作为延迟,总cycles数为：

其中，iters是VEU的另外一个寄存器，代表当前任务的重复次数；vl也是VEU的一个控制寄存器，代表当前任务的数据个数，且不可配置为0。除此之外，在向量计算时有可能会超出当前的数据类型的大小，所以需要对计算前后的数据进行移位、饱和、进位等操作，这些都在控制寄存器CTRL0和CTRL1中进行配置。

由于VEU只能计算存储在VMEM里的数据，所以VEU支持的向量指令具有以下形式：

其中dst、src0、src1是VMEM里的array编号，dype、stype0和stype1分别是计算结果的数据类型和操作数的数据类型。VEU支持结果类型和操作数类型的混合计算，但不支持某一个操作数或运算结果向量里有不同的数据类型。

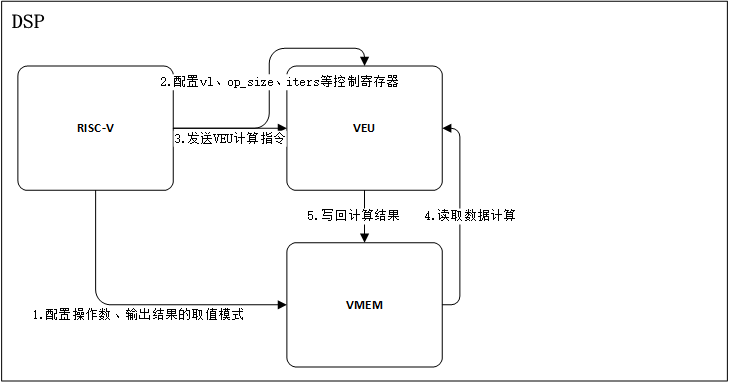


图6 VEU使用流程图

如图6所示，在使用VEU进行向量运算时，其工作流程如下：

1. 使用RISC向VMEM存储系统配置向量操作数和计算结果的取指模式，取指模式决定了后续VEU从VMEM中取出数据的顺序，并获取操作数和计算结果的array 编号。
2. 使用RISC配置VEU中的vl、op\_size、iters、CTRL0、CTRL1等控制寄存器，这些寄存器决定了要计算的向量长度，计算并行度，饱和移位等重要配置。
3. 使用RISC向VEU发送向量计算指令，且指令中的操作数和计算结果的编号是根据1）中获取的。
4. VEU将根据控制寄存器的配置和从RISC接收的指令，从VMEM对应的array编号里取出长度为op\_size的向量进行计算。
5. VEU将计算结果写回到对应的VMEM里。

### Matrix Multiplication Unix

MMX即Matrix Multiplication Unix,是DSP核心计算单元之一，采用脉动阵列结构，主要负责进行卷积、矩阵乘运算，对MMX的使用也是采用控制寄存器加指令的形式。MMX的控制寄存器比较多，使用前的配置也较为复杂。

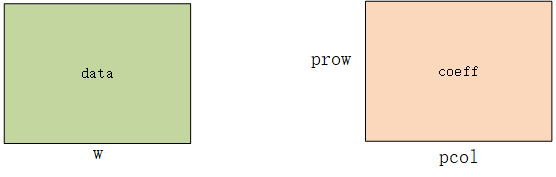


图7 MMX参数说明

如图7，不管是卷积还是矩阵乘都需要两个矩阵，第一个矩阵称之为data,第二个称之为coeff。其中，pcol代表coeff矩阵的列数，同时它与VL控制寄存器共同决定了每个cycle读取data/coeff矩阵的数据个数；prow代表coeff矩阵的行数；w代表data矩阵的列数，所以data矩阵的行数drow为

MMX一条矢量指令可以处理ITERS个矩阵乘法，对矩阵进行适当分块，可将高阶矩阵的运算转化为低价的矩阵运算，同时也使原矩阵的结构显得简单而清晰，从而能够大大简化运算步骤。MMX通过矩阵分块，行列拆分，分块相乘，分块累加，简化了运算，提升了效率。

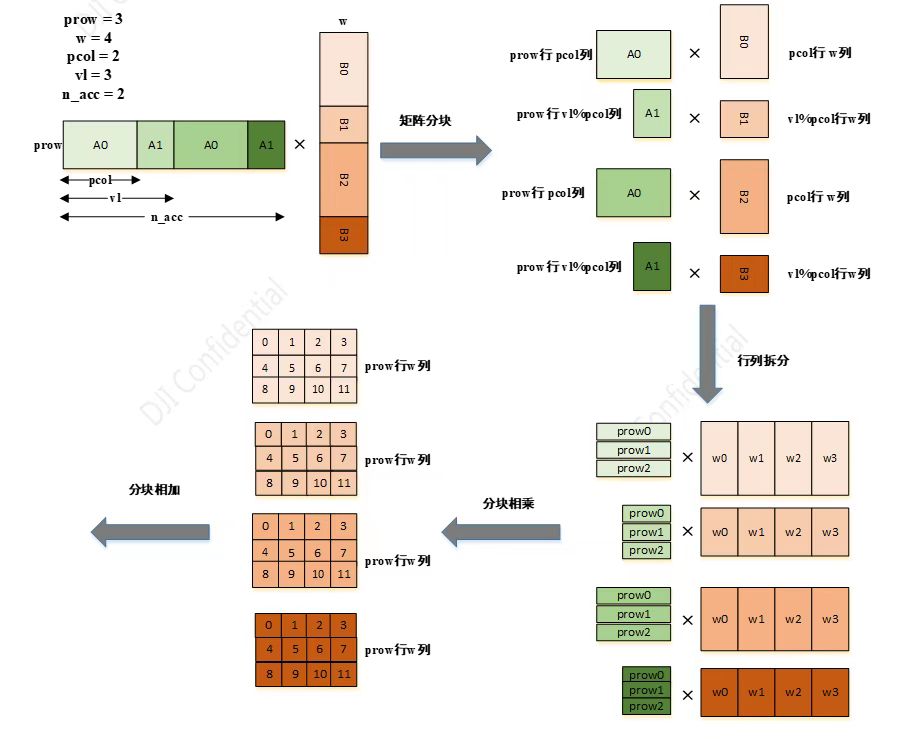


图8 矩阵分块

如图8，两个高阶矩阵可通过矩阵分块分为四个低阶矩阵进行计算，四个小矩阵分块相乘然后相加，可得到最终的结果。其中，n\_acc代表计算特征图每个点需要的累加次数，由于VL大于pcol，所以其op\_size为2，一次VL可划分为两个小矩阵，又因n\_acc等于2，所以则需要4个小矩阵相乘后相加得到最终特征图的每个点。n\_acc、prow、pcol、w均位于控制寄存器CTRL0中。除此之外，与VEU类似，VL位于VL控制寄存器，饱和、移位、进位等配置位于控制寄存器CTRL1中，ITERS代表了此次运算次数，位于ITERS控制寄存器。

### Vector Memory

VMEM即Vector Memory,是在DSP系统里面的存储单元，DSP中其他计算单元只能从VMEM中取出数据计算，并写回到VMEM。VMEM模块包含10块array,每块array的大小是64kbytes。除此之外VMEM还有10个地址产生器AGU，VMEM通过地址产生器读取config寄存器的配置信息，产生对应的地址，供其他模块读写,VMEM不支持跨array访问。

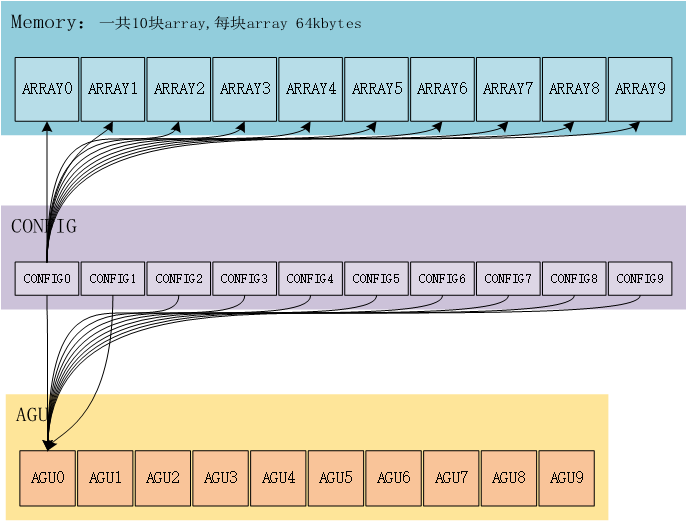


图9 VMEM

在使用VMEM之前，先由RISC向CONFIG控制寄存器配置对应array的取值模式，然后其他计算单元使用数据时，将通过AGU产生的地址找到其对应的数据。

### Direct Memory Access

DMA即Direct Memory Access,是DSP系统和外部进行数据传输的组件，DSP和外部存储器DDR之间通过AXI总线作为Master进行交互，和内部存储器通过Crossbar作为Master进行交互。DMA也是由控制寄存器驱动，通过RISC读写相应寄存器完成DMA数据传输任务的压入、执行、任务执行状态的查看等操作。

DMA有读任务、写任务两个任务队列，两个任务队列分别最多可缓存16个任务，超出的任务将会被丢弃。DMA支持读写通道并行传输，多任务间按优先级执行。DMA支持从外部存储器DDR和DMEM之间传输的数据类型有char、short、int，其任务描述符ID、搬运起始地址、搬运大小、搬进地址以及最大搬运并行度等重要参数都可由其控制寄存器进行配置。

## MLIR

编译器设计领域已经出现了很多成熟的技术平台，例如LLVM编译器框架，JAVA虚拟机等，这些平台已经实现了大规模的重用，但是这些主流的系统都会在前端构造自己的IR，而最终他们都将转换到较为低级的LLVM IR来生成可执行程序。如图10所示，传统编译过程如下。

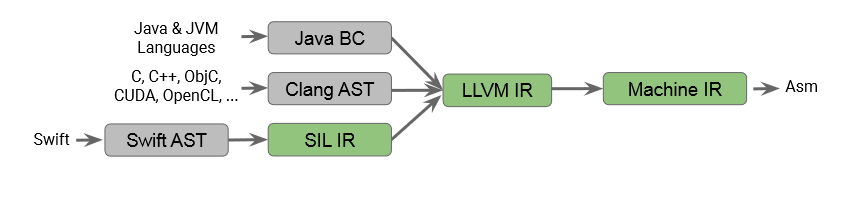


图10 传统编译过程

类似的，机器学习领域通常使用计算图来作为其前端的IR，而如今一些AI编译器的出现则让IR的种类更加层出不穷，而这些IR之间的转换和复用则是非常困难的。MLIR的其中一个目标则是为了有效的解决这一问题，Multi-Level Intermediate Representation(MLIR)旨在提供可复用的，可扩展的编译器基础设施，并降低构建特定领域编译器的成本。

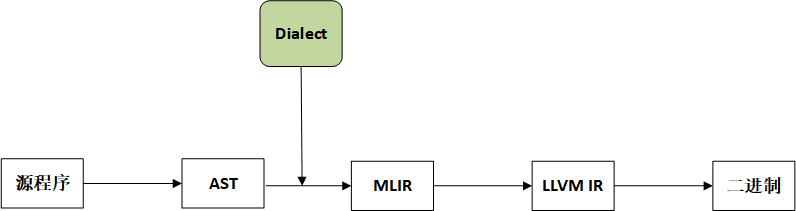
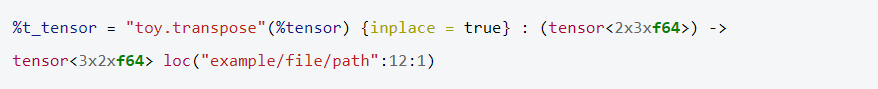


图11 MLIR编译过程

如图11，MLIR在编译的时候会接收前端生成的AST或一些较为高层的IR,如机器学习的计算图。然后通过Dialect定义的Operation，将接收到的AST或IR转换为MLIR,然后经过一系列的下降，最终将其转换到LLVM IR，由后端编译器将其编译生成二进制程序。下面介绍MLIR中一些重要的概念。  
 **Dialect**:MLIR使用Dialect来管理扩展性，Dialect是同一个命名空间下的一组Operations、Attributes、Types的集合。Dialect是一个逻辑组合，不引入实际的意义，通常情况下，一个Dialect或Dialect的混合对应一种IR。MLIR中的Dialect是多层设计的，并且提供了友好的OSD框架来方便用户定义新的Dialect或扩展已有的Dialect,除此之外MLIR还提供了一系列的组件和规则方便Dialect之间的互相转换。MLIR已经在不同的抽象层次上提供了许多实用的Dialect，下面介绍本文后续将会使用到的Dialect。

* Affine Dialect:Affine Dialect是MLIR最先提供的Dialect之一，Affine 提供了维度和符号两个概念对多面体编译的支持，使依赖分析和循环转换高效可靠。
* SCF Dialect:SCF Dialect包含了比控制流图分支更高层次上的控制流概念，例如for和while循环，条件分支跳转等。SCF Dialect是比较常用的Dialect之一。
* Standard Dialect:由SCF Dialect拆分出来的Dialect,提供了一些标量和向量的基本运算操作，如加减乘除，移位，取绝对值等操作。
* LLVM Dialect:LLVM Dialect是MLIR中最低的抽象层次，是为LLVM IR构建的Dialect，代码生成的最后阶段要生成LLVM IR，所以需要将构建的Dialect完全降低到LLVM Dialect。

MLIR中的Dialect并不是固定不变的，MLIR鼓励用户使用ODS框架来定义新的Dialect，理论上，可以为所有的IR构建对应的Dialect来接入MLIR。  
 **Operation**:Operation是MLIR中最基本的语义单元，也是Dialect中最重要的组成部分。一个Operation代表了一个操作，这个操作由Dialect的构造者实现，可以是一条指令，也可以是一个函数，一条加法运算。Operation由操作名称、操作结果、操作数、返回值等多个元素构成，一个基本的Operation如图12所示：

  
图12 operation

* %t\_tensor:此Operation执行后的结果，一个Operation可以定义零个或多个结果，并且它们是SSA值。
* toy.transpose:此Operation的名称，是在此命名空间中不可重复的字符串，Dialect的命名空间以“.”为前缀，这里可以理解为toy Dialect中的transpose操作。
* %tensor:零个或多个操作数的列表，它们是由其他Operation定义的或者是引用的Block参数。
* {inplace = true}:零个或多个Attribute的字典，它们是始终不变的特殊操作数。在这个例子中，这是一个“inplace”的布尔属性，且它的值为真。
* (tensor<2x3xf64>) -> tensor<3x2xf64>:指此Operation是函数形式，括号中的是参数类型和返回值类型。
* loc("example/file/path":12:1):指此Operation在源程序中的位置信息。

**Attribute**：在MLIR中，属性是指在编译时一些结构化的静态信息，通常属性是一组键值对，例如图12中{inplace = true}就是指此Operation有一个inplace的属性，其值为true。在一些Operation中，属性可保存一些变量的静态值，如一些常量值、字符串等。在Affine Dialect的循环中，属性还会保存控制循环的条件信息。属性也是可扩展的，允许直接引用外部的数据结构，这对于已经存在的系统集成来说是非常有用的。

**Type**:在MLIR中每一个SSA都有一个数据类型，这些SSA的数据类型由Operation的操作结果或者由块参数定义时决定，并且MLIR对数据类型的检查是非常严格的，在Operation的转换和SSA的引用时会严格的比较两边的数据类型是否一致。MLIR提供了一些基本的数据类型，包括整型，浮点类型，指针类型，多维的向量类型，tensor类型等。同时，数据类型在Dialect中也是可以扩展的，Dialect的数据类型可由用户选择和定义。

**Block**:在MLIR生成的IR的结构中，Block包含了一组Operation，除此之外Block可包含一组参数，称为块参数，且他们都是SSA值，被包含在这个Block的Operation都可以引用块参数作为操作数。在MLIR中，每个SSA都是一个Value类的实例，所以每个SSA都可以追溯到其引用者，这两者的关系被称之为def-use链。如图13所示，该Block一共包含了3个Operation和多个块参数，第一个Operation的操作数都来自块参数；第二个Operation的操作数一个来自块参数，一共来自第一个Operation的操作结果，前文讲述过，Operation的操作结果也是SSA值，所以可以被后续的Operation引用；第三个Operation的操作数都来自第二个Operaton的操作结果。Def-use链是MLIR中比较重要的数据结构，通过def-use链可以得到每个SSA值被引用的Operation。

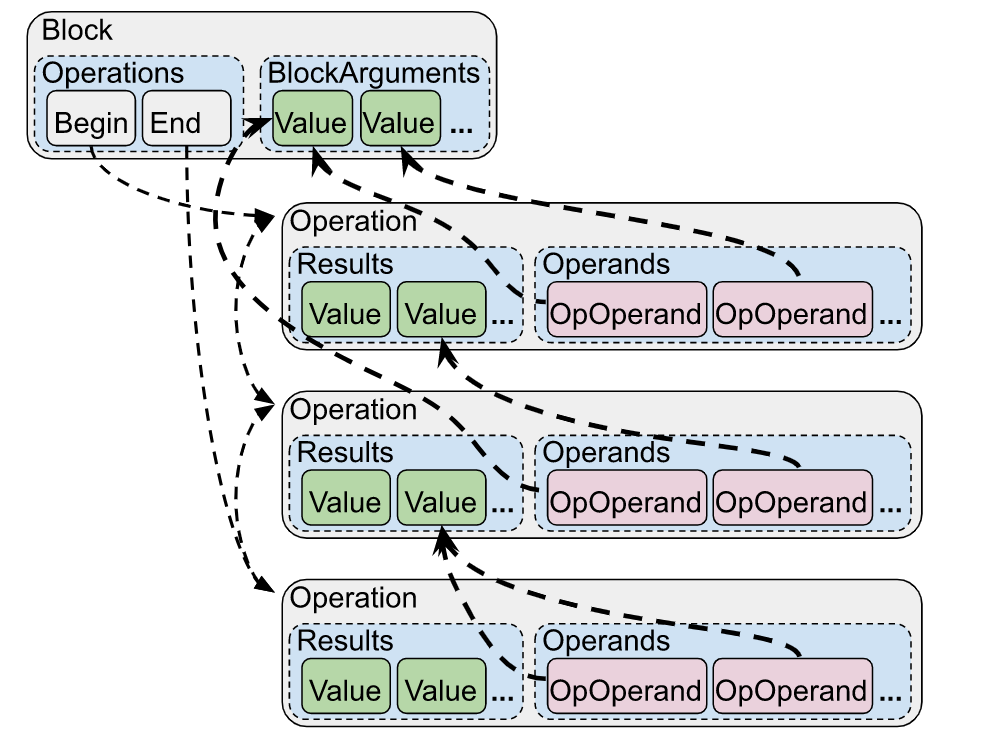


图13 def-use链

**Region**:如果说Block像是编程语言中一个函数的作用域，那么Region更像是包含了多个代码块的控制流图，一个Region只能包含一个或多个Block,与Block不同的是，Region没有自己的参数，Region的语义是由它所包含的Block定义的，例如，一个if-else分支语句包含了if的Block块和else的Block块。在MLIR中，一个Operation可包含一个或多个Region，一个Region可包含一个或多个Block,一个Block可包含零个或多个块参数和一个或多个Operation。Operation,Region,Block这三者之间是递归嵌套的，他们之间使用的SSA值是通过def-use链连接的。

## 本章小结

本章对本论文所用到的理论基础与相关技术进行了完整的介绍。本章首先介绍了所用到的DSP的整体架构,然后详细介绍了DSP的向量计算单元、矩阵计算单元、存储系统VMEM和数据搬运模块DMA。然后介绍了MLIR编译器基础设施框架使用时候的编译流程，以及MLIR中Dialect的概念、Dialect的组成和结构，给之后章节的DSP Dialect的构建提供了理论基础。

# 构建DSP Dialect

由前文可以得知，虽然MLIR社区已经提供了GPU、ARM、RISCV等几款主流的处理器的Dialect,但对于某公司自研的DSP来说是不能正常使用的，长期以来，该公司的多款自研DSP的编译器开发和算子开发都需要大量的人力、财力投入，为了降低开发自研DSP编译器和高性能算子库的成本，本文为该公司某年的自研DSP构建Dialect。虽然目前该款自研DSP传统编译工具链已经具备，但是本文开发的源代码已经合入该公司的代码库，为该公司的深度学习编译器接入DSP做了准备。本章将针对该公司该款的自研DSP，依据MLIR构建Dialect的原则，为该DSP设计和实现Dialect。

## DSP Dialect的设计思想

DSP Dialect是偏向硬件的Dialect,其主要目的是为了将上层的IR经过一系列的转换后，将算子的实现下降到该DSP Dialect上，从而生成该DSP的可执行程序。

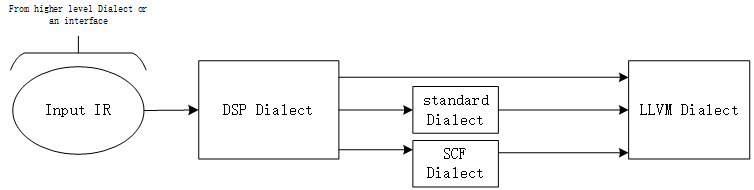


图14 DSP Dialect转换图

如图14所示，从较高层的Dialect转换到DSP Dialect后，DSP Dialect会转换成standard Dialect、SCF Dialect和LLVM Dialect的混合IR，而最终standard Dialect和SCF Dialect也都转换成LLVM Dialect，进而生成LLVM IR，由后端编译器将其编译成可执行程序。

为了方便上层的Dialect更好的使用DSP,同时保证该DSP功能的完整性，将DSP的性能发挥到最大，本文将分别详细介绍DSP内的每个子系统的Operation的设计和构造过程。同时为了保证DSP Dialect中Operation的模块化，本文将DSP内每个子系统的可编程寄存器一一整理出来，将它们的功能和作用做Operation级的封装，相同子系统单元的Operation以该子系统的名称缩写为前缀，以该Operation的功能名称为后缀，通过每个Operation的命名，上层的Dialect可以很容易的区分和使用这些Operation。而对于向量计算单元和矩阵计算单元，其向量指令一直以来都是以内联汇编的形式嵌入在LLVM IR中，由后续的汇编器将其翻译成可执行的二进制代码，故该DSP向量指令的Operation的目标是在下降到LLVM Dialect时生成相应的内联汇编，本文将其所支持的所有向量指令分类整理，将同类别的指令封装成一个Operation，并将它们的差异参数化，在使用时通过传递相应的参数来构造不同的向量指令。同样的，向量指令的Operation的名称也是以子系统名称缩写为前缀，以该类指令的功能名称为后缀命名的。

在DSP Dialect中的Operation实现过程中，MLIR编译器框架提供了Dialect下降转换的规则和方法，在编码中首先需要为对应的Operation构建其Lower类，然后实现其中的matchAndRewrite()方法，matchachAndRewrite()方法需要使用要下降的Dialect实现该Operation,然后将此Lower类加入PASS管理器中即可。

DSP Dialect的Attribute和Type将使用MLIR提供的默认集合，DSP中的RISC支持c语言中的基本数据类型，且支持其所有的相关的标量操作。RISC相关的所有标量指令，可直接下降转换到LLVM Dialect，由LLVM Dialect生成LLVM IR,后端编译器会将其翻译成RISC可执行的二进制代码。

## IntrinsicOp

IntrinsicOp是DSP Dialect中的基石，它是DSP Dialect中最基础的operation,它打通了DSP Dialect到LLVM Dialect到LLVM IR的整条通路。本文在构建IntrinsicOp之前需要先使用LLVM为DSP构建Intrinsic。

### Intrinsic与IntrinsicOp的关系

Intrinsic是编译器内建的函数，由编译器提供，类似于内联函数。与内联函数不同的是，编译器与硬件架构的联系紧密，会根据硬件的能力以最优的方式去提供这些功能，通常这些Intrinsic直接在调用的地方插入代码，避免了函数调用的额外开销，通常来说Intrinsic要比等效的内联汇编代码快。在LLVM中，Intrinsic函数一般是在IR级代码优化时引入的，也就是在生成LLVM IR的阶段产生，其函数的前缀一般都是保留字“llvm.”,LLVM后端会在代码生成阶段选择最高效的形式将Intrinsic转换给硬件执行，可以将Intrinsic函数拆分为一系列机器指令，也可以映射为单独一条机器指令，并直接调用相应的硬件功能。

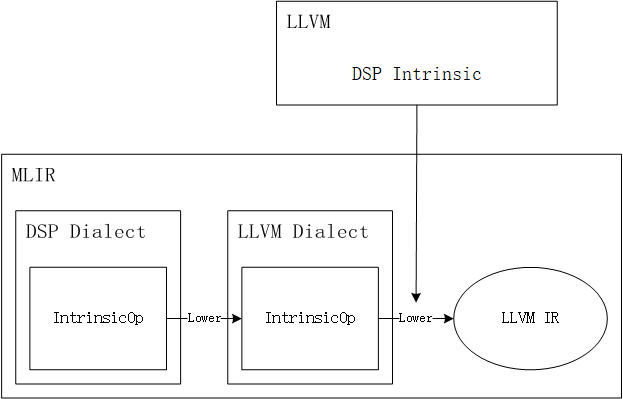


图15 Intrinsic与IntrinsicOp的关系

如图15所示，DSP Dialect在不断下降的过程中，IntrinsicOp会使用LLVM中DSP的intrinsic来生成LLVM IR。Intrinsic是LLVM中的概念，其主要目的是前端语言在生成LLVM IR的时候，生成编译器构建的内联函数。各个处理器的Intrinsic都不相同，但都是高效的且是高度可复用的，构建Intrinsic可以很方便的提升编译器的代码生成效率。而IntrinsicOp是MLIR中Dialect的Operation,虽然MLIR是LLVM中的一个子项目，且代码仓库和LLVM共用，但是两者是属于不同的概念。IntrinsicOp的主要作用是在MLIR Dialect层面对LLVM IR层面的Intrinsic做一层抽象，也就是说IntrinsicOp是LLVM中Intrinsic在MLIR的抽象形式，这是因为在MLIR中所有的IR都要看作一个Dialect,如果要通过LLVM Dialect去使用扩展的Intrinsic，就必须在LLVM Dialect层面构建相应的IntrinsicOp。

### 构建DSP Intrinsic

由前文叙述，DSP中的RISCV核主控制，其他向量计算单元、矩阵计算单元、存储模块等都是由RISC核配置控制寄存器，然后再由RISC核发送指定向量指令执行的，基于Intrinsic可复用性强、运行效率快等特性，本文构建了七条常用的Intrinsic。

|  |  |  |  |
| --- | --- | --- | --- |
| Intrinsic | 操作数类型 | 返回结果类型 | 作用 |
| llvm.riscv.get.cr | i64 | i64 | 获取该寄存器编号的状态值 |
| llvm.riscv.set.cr | i64 i64 | null | 设置某个寄存器的状态值 |
| llvm.riscv.wait.valu | null | null | 等待VALU单元向量指令执行结束 |
| llvm.riscv.wait.veu | null | null | 等待VEU单元向量指令执行结束 |
| llvm.riscv.wait.vfpu | null | null | 等待VFPU单元浮点向量指令执行结束 |
| llvm.riscv.light.sleep | null | null | 浅睡眠 |
| llvm.riscv.deep.sleep | null | null | 深睡眠 |

表1 DSP Intrinsic

如表1所示，DSP中的Intrinsic均以“llvm.riscv”为前缀，以功能名为中间名称，以执行单元为后缀命名的。其中，只有“llvm.riscv.get.cr”接收一个64位的int值作为其参数，并将该int值编号的状态以64位的int值返回。同样的，“llvm.riscv.set.cr”接收两个64位的int值，第一个值是要配置的寄存器编号，第二个值是配置到寄存器的Value。“llvm.riscv.set.cr”和“llvm.riscv.get.cr”是两条最常用的Intrinsic，是DSP进行向量运算的基石，通过这两条Intrinsic可以配置整个DSP中所有可编程的控制寄存器，包括RISC本身的寄存器。

### 构建DSP Dialect IntrinsicOp

IntrinsicOp是Intrinsic在MLIR中的抽象，由于LLVM Dialect是对LLVM IR的抽象，所以需要先在LLVM Dialect中扩充对应的IntrinsicOp，使其能够利用DSP Intrinsic能够生成对应的LLVM IR。然后在DSP Dialect层声明对应的Operation，将DSP Dialect中的Operation下降到LLVM Dialect中新扩充的IntrinsicOp，于是便可以通过DSP Dialect中的IntrinsicOp使用DSP的intrinsic。

|  |  |  |
| --- | --- | --- |
| DSP Dialect IntrinsicOp名称 | LLVM Dialect IntrinsicOp名称 | DSP Intrinsic |
| DSP\_GetCrOp | LLVM\_GetCrOp | llvm.riscv.get.cr |
| DSP\_SetCrOp | LLVM\_SetCrOp | llvm.riscv.set.cr |
| DSP\_WaitValuOp | LLVM\_WaitValuOp | llvm.riscv.wait.valu |
| DSP\_WaitVeuOp | LLVM\_WaitVeuOp | llvm.riscv.wait.veu |
| DSP\_WaitVfpuOp | LLVM\_WaitVfpuOp | llvm.riscv.wait.vfpu |
| DSP\_LightSleepOp | LLVM\_LightSleepOp | llvm.riscv.light.sleep |
| DSP\_DeepSleepOp | LLVM\_DeepSleepOp | llvm.riscv.deep.sleep |

表2 DSP Dialect IntrinsicOp

如表2所示，DSP Dialect IntrinsicOp、LLVM Dialect IntrinsicOp和DSP Intrinsic三者之间是一一对应的关系。虽然在LLVM Dialect扩展的IntrinsicOp就已经可以满足DSP Dialect中其他Operation的下降，但是为了保证Dialect的完整性和扩展性，依然需要在DSP Dialect层对LLVM Dialect扩展的IntrinsicOp做一次封装。并且为了降低Dialect之间的耦合度，在下降DSP Dialect中其他的Operation时应尽可能的使用LLVM Dialect的IntrinsicOp,而较为高层的IR在使用DSP实现算子的时候，应该尽可能的使用DSP Dialect中的IntrinsicOp，而不是直接使用LLVM Dialect中的。

由于“llvm.riscv.get.cr”和“llvm.riscv.set.cr”都需要接收64位的int值作为参数，所以在LLVM Dialect和DSP Dialect在构建IntrinsicOp时，需要两个Dialect都支持I64数据类型。LLVM Dialect是LLVM IR的抽象，几乎所有的基本数据类型都支持，而DSP Dialect则需要将I64的数据类型添加到其Dialect的数据类型集合里。MLIR对数据类型的检查是及其严格的，在Dialect的每一步转换时，MLIR都会对每个Operaton的操作数和操作结果与要转换的Operation的操作数和操作结果做比较，两边的数据类型必须是完全一致的，否则将会转换失败。所以当上层的IR在需要下降到DSP IntrinsicOp时，需要对其下降的Operation的操作数做一定的检查。

### 测试

在LLVM编译器开发过程中，针对特定平台必然要定义特定指令集及其指令格式，并对整个流程的各个阶段做相应修改，根据需求编写[测试用例](https://link.zhihu.com/?target=https://baike.baidu.com/item/%E6%B5%8B%E8%AF%95%E7%94%A8%E4%BE%8B" \t "https://zhuanlan.zhihu.com/p/_blank)的测试驱动开发是推动开发进行的有效方法，可以在出现问题时实现可回溯，有助于编写简洁可用和高质量的代码。LLVM测试框架是在LLVM编译器开发过程中实现测试驱动开发的有效手段，有很高的灵活性和健壮性，可保证加速开发过程稳步进行。在LLVM社区和实际的开发中，为每个新增的模块构建回归测试则是必要的，MLIR是LLVM中的一个子项目，可以很方便的使用LLVM测试框架对新开发的模块进行测试,由于本文的所有开发都是基于LLVM 14.0的代码库进行的，所以本章的回归测试也均是基于LLVM的测试框架下构建的。

1. DSP Intrinsic测试

LLVM测试框架提供了llvm-llt、FileCheck等诸多工具方便编写和运行测试用例，构造的七条Intrinsic是后续DSP Dialect的最基本的模块，本文首先为这七条Intrinsic编写测试文件。在llvm/test/CodeGen/RISCV/allspark/目录下新建一个Intrinsic.ll测试文件，其主要内容如下：

|  |
| --- |
| ; Tests mapping from get\_cr/set\_cr intrinsics to instructions  ; RUN: llc -mcpu=ask50-rv -o - %s | FileCheck %s  ...  ; Read from (constant) CSR address: use csrr instruction  ; CHECK-LABEL: csrr {{a[0-9]+}}, 1000  ; Write register to (constant) CSR address:  ; CHECK-LABEL: csrw 1001, {{a[0-9]+}}  ...  ; Write to maximum constant address (4095):  ; CHECK-LABEL: csrwi 4095,2  ...  ; wait valu  ; CHECK-LABEL: wait 2  ...  ; deep sleep  ; CHECK-LABEL: slp 3  define dso\_local signext i64 @test\_allspark\_builtins(i64 %p, i64 %val) local\_unnamed\_addr #0 {  entry:  %0 = call i64 @llvm.riscv.get.cr(i64 1000)  call void @llvm.riscv.set.cr(i64 1001, i64 %val)  ...  call void @llvm.riscv.set.cr(i64 4095, i64 2)  call void @llvm.riscv.wait.valu()  ...  call void @llvm.riscv.deep.sleep()  ret i64 %2  } |

该测试文件主要对七条Intrinsic做了功能测试，其中包括操作地址边界寄存器、大小常数入参，变量入参等用例。通过llvm-lit脚本工具运行测试测试文件：

|  |
| --- |
| $~/build/bin/llvm-lit Intrinsic.ll  -- Testing: 1 tests, 1 workers --  PASS: LLVM :: CodeGen/RISCV/allspark/Intrinsic.ll (1 of 1)  Testing Time: 0.04s  Passed: 1 |

1. DSP Dialect IntrinsicOp测试

同样的，需要为MLIR中DSP Dialect的七个IntrnsicOp编写测试用例，需要在mlir/test/Dialect目录下新建一个DSP目录，然后创建一个IntrinsicOp.mlir文件编写测试用例，其主要内容如下：

|  |
| --- |
| ; Check DSP Dialect IntrinsicOp ---> LLVM Intrinsic  ; RUN : DspCompiler %s -emit=llvm 2>&1 | FileCheck %s --check-prefix=CHECK  -LLVM  ; Check DSP Dialect IntrinsicOp ---> LLVM Dialect IntrinsicOp  ; RUN : DspCompiler %s -emit=mlir-llvm 2>&1 | FileCheck %s --check-prefix  =CHECK-MLIR  ; CHECK-LLVM: call void @llvm.riscv.light.sleep()  ...  ; CHECK-LLVM: call void @llvm.riscv.set.cr(i64 4095, i64 21)  ; CHECK-MLIR: llvm.riscv.light.sleep()  ...  ; CHECK-MLIR: llvm.riscv.set.cr(4095 : i64, 21) -> ()  func @main(%arg0 : i64, %arg1 : i64) {  dsp.light\_sleep()  ...  dsp.wait\_valu()  %0 = dsp.get\_cr(302) : (i64) -> i64  ...  dsp.set\_cr(4095,21) : (i64, i64) -> ()  dsp.return  } |

该测试文件包含了三种IR,两组测试用例，两组测试用例分别以CHECK-LLVM和CHECK-MLIR为前缀区分。@main中是DSP Dialect的IR形式，其中的Operation均以dsp为前缀，如果Operation有操作数和返回值，将会在Operation的“:”后给出。CHECK-MLIR是为了测试从DSP Dialect到LLVM Dialect的转换过程，其中包含了IntrinsicOp的功能性测试，边界测试和变量入参测试，CHECK-MLIR是LLVM Dialect的IR，其中的Operation均以llvm为前缀，且其操作数和返回值类型的显示格式与DSP Dialect的IR类似。CHECK-LLVM主要是为了测试从DSP Dialect转换到LLVM Dialect，再转换到LLVM IR的过程，其中的测试用例和CHECK-MLIR一致，CHECK-LLVM是LLVM IR，为了检查DSP Dialect是否生成了LLVM IR对应的Intrinsic。通过llvm-lit脚本工具运行测试测试文件：

|  |
| --- |
| $build/bin/llvm-lit IntrinsicOp.mlir  --Testing: 1 tests, 1 workers --  PASS: MLIR :: Dialect/DSP/IntrinsicOp.mlir (1 of 1)  Testing Time: 0.03s  Passed:1 |

## 向量计算单元

VEU是DSP的整数向量计算单元，主要负责计算整数向量加、向量减、向量乘，向量累加，向量绝对值等操作。向量逻辑运算单元VALU主要负责向量的逻辑运算，如向量左移、向量右移、向量或、向量与、向量异或等。浮点向量运算单元VFPU负责浮点向量运算，由于向量除法很容易产生浮点数，为了保证精度，在该DSP中所有的整数向量的除法运算都要先转换成浮点向量，然后使用浮点计算单元VFPU进行计算，最后将结果转换成整数向量，除此之外浮点运算单元VFPU也支持浮点向量加、浮点向量减、浮点向量乘等操作，与VEU类似。由于整数向量计算单元VEU、向量逻辑运算单元VALU、浮点计算单元VFPU三个有关向量计算的单元的功能和使用方法相似，三个模块在DSP Dialect中构建的Operation也是比较相似的，所以本文以整数向量单元VEU构建Operation为例，详细介绍向量计算单元Operation的设计和实现过程，向量逻辑运算单元VALU和浮点向量运算单元VFPU的Operation的构建过程本文不再赘述。

DSP中向量计算单元均是以控制寄存器和向量指令的形式来完成向量计算的。在使用相应的计算单元之前，需要使用RISC对该计算单元的控制寄存器进行必要的配置，然后再由RISC向其发送向量指令，向量计算单元接收到向量指令后，将根据控制寄存器相应的配置，完成向量计算，将计算结果写回DSP存储系统VMEM后将向RISC发送完成中断，RISC接收完成中断，向量运算完成。所以本文需要先为向量计算单元的控制寄存器构建相应的Operation。

### 构建向量计算单元寄存器相关的Operation

DSP的向量计算单元的控制寄存器均是32位的，通过控制寄存器总线与RISC链接，主要分为可编程寄存器和不可编程寄存器。可编程寄存器可由DSP的RISC通过csr指令读写，主要对接下来要执行的向量指令进行一些必要的配置。不可编程寄存器的功能主要是在向量计算单元在执行向量指令时对一些必要的中间数据进行保存，如累加器等。本文主要对可编程寄存器做DSP Dialect层面的封装，在一些Operation的下降实现时，有可能会读取一些不可编程寄存器的当前状态。

|  |  |  |  |
| --- | --- | --- | --- |
| DSP Dialect Operation | 操作数 | 下降转换到的LLVM Dialect Operation | 作用 |
| DSP\_VEUSetCtrl0Op | op\_size mode post\_shift pre\_shift | LLVM\_SetCrOp  LLVM\_ShlOp  LLVM\_OrOp  LLVM\_AndOp | 配置此次向量运算的最大并行度Op\_size和计算前后的数据shift大小 |
| DSP\_VEUSetCtrl1Op | post\_rnd  pre\_rnd | LLVM\_SetCrOp  LLVM\_ShlOp  LLVM\_OrOp  LLVM\_AndOp | 配置计算前后的数值舍去规则 |
| DSP\_VEUSetVlOp | vector\_length | LLVM\_SetCrOp | 配置向量运算的向量长度 |
| DSP\_VEUSetIterOp | iter | LLVM\_SetCrOp | 配置向量运算的次数 |

表3 向量计算单元控制寄存器相关Operation

如表3所示，为向量计算单元构造了四条控制寄存器相关的Operation,这四条Operation分别配置VEU单元的四个寄存器，配置向量长度、运算次数分别占用了两个寄存器，CTRL0主要配置向量并行度和数据的shift，CTRL1寄存器配置计算前后的数据取舍。四个Operation的实现是基于前文构造的LLVM Dialect中的LLVM\_SetCrOp，这个operation可以生成配置控制寄存器的Intrinsic。以DSP\_VEUSetCtrl0为例，该Operation下降转换为LLVM Dialect的伪代码为：

|  |
| --- |
| 方法名： matchAndRewrite()  输入：  要转换的Operation:DSP\_VEUSetCtrl0Op  步骤：  通过该Operation获取到其操作数。  声明一个整数变量result,用于保存最终操作的结果  for every operand do  按照寄存器的有效位置对result进行移位运算。  end for  使用result构造一个Value类型（在生成的LLVM IR中是一个SSA) vResult  LLVM\_setCR(CTRL0,vResult)  在该IR中删除DSP\_VEUSetCtrl0节点 |

表4 DSP\_VEUSetCtrl0Op转换

### 构建生成向量指令的Operation

向量计算单元通过接收RISC发过来的向量指令进行运算，VEU向量计算单元所支持的向量指令可分为两种，分别为基础向量指令和组合向量指令。基础向量指令是两个操作数的向量指令，例如两个向量相加，两个向量相乘等指令。组合向量指令是具有三个操作数的向量指令，例如前两个向量相加，再减去第三个向量。相同数据类型和向量长度下组合向量指令和基础向量指令花费的时间相当，也就是说一条组合向量指令比两条基础向量指令要快的多。

向量计算单元的向量指令在LLVM IR中具有以下形式：

|  |
| --- |
| Opcode [dst].dtype, [src0].stype0, [src1].stype1, ...  Opcode [dst].dtype, [src0].stype0, imm.stype1, ... |

其中，imm代表是一个常数。所以需要构造LLVM IR的内联汇编，幸运的是，LLVM Dialect提供了这样一个Operation，即LLVM\_InlineAsmOp。

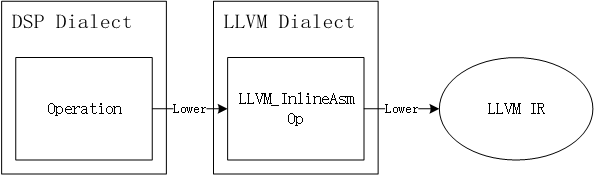


图16 向量指令Operation转换

如图16所示，在DSP Dialect中向量指令的Operation会被转换成LLVM Dialect的LLVM\_InlineAsmOp，然后在编译阶段将会生成对应的内联汇编。

由于VEU支持char,short,int数据类型，并且一条向量指令内的操作数和运算结果数据类型可以混合，所以导致VEU中向量指令非常多，因此构建的Operation也很多。为了覆盖到每一条向量指令，该组Operation的命名在原有的Operation命名规则之上增加了以下规则：

1. 加入该条指令的功能为前缀。例如一条向量加的向量指令，其Operation命名为DSP\_VEUAddxxxOp。一条向量加减的符合向量指令，其Operation为DSP\_VEUAddSubxxxOp。
2. 将此向量指令的操作数和操作结果数据类型的第一个缩写加入到Opeation的后缀中。例如一条操作数和结果都是int数据类型的向量加指令，其Operation的名称为DSP\_VEUAddiiiOp,前两个‘i’分别是第一个、第二个操作数的数据类型缩写，第三个‘i’是结果的数据类型缩写。

在具体实现时，MLIR提供了llvm-tblgen工具可以很方便构造这种同类型的Operation：在构造时可以将它们的差异参数化，而此时只需要在td文件中为这些Operation构造一个抽象类，然后可以通过定义该类的实例构造这些Operation。

### 测试

与IntrinsicOp的测试相似，向量计算单元的测试文件位于mlir/test/Di

alect目录下，文件名为veu.mlir,主要内容如下：

|  |
| --- |
| ; Check DSP Dialect VEU Operation ---> LLVM Dialect Operation  ; RUN : DspCompiler %s -emit=mlir-llvm 2>&1 | FileCheck %s  ; Check dsp.veu\_set\_ctrl0  ; CHECK: {{%[0-9]+}} = llvm.riscv.get.cr({{%[0-9]+}} : i64) : (i64) -> (i64)  ...  ; CHECK: llvm.inline\_asm has\_side\_effects "vmadd [0].char, [1].char, [2].char, [3].char", "" : () -> !llvm.void  func @test() {  dsp.veu\_set\_ctrl0(0,1,0,0)  ...  dsp.veu\_addiii(0,1,2)  ...  dsp.veu\_addccc(0,1,2)  ...  dsp.veu\_submulccc(0,1,2,3)  dsp.return  } |

该测试文件@test函数里是DSP Dialect的IR，CHECK注释里是LLVM Dialect的IR。首先构造了控制寄存器相关的Operation的用例，检查是否生成了对应的LLVM Dialect Operation。为了保证DSP Dialect的完整性，该文件对VEU所有的向量指令Operation做了全覆盖的测试，即所有的向量指令都将会被测试到，该测试文件将所有的向量指令都做了检查，如果有一条向量指令没有被正确生成，或者丢失，则此测试将会failed,通过llvm-lit脚本工具运行测试测试文件：

|  |
| --- |
| $build/bin/llvm-lit veu.mlir  --Testing: 1 tests, 1 workers --  PASS: MLIR :: Dialect/DSP/veu.mlir (1 of 1)  Testing Time: 0.63s  Passed:1 |

## 矩阵计算单元

矩阵计算单元MMX是CNN核心计算单元，主要负责进行卷积、矩阵乘运算，和向量计算单元类似，对MMX的控制也是采用控制寄存器加指令的形式。与向量计算单元不同的是，MMX仅支持有符号和无符号的char和short数据类型的计算，并且MMX支持的向量指令只有矩阵乘和卷积乘两类，所以与MMX有关的Operation比向量计算单元要少的多。首先需要为MMX的控制寄存器构建相应的Operation。

### 构建矩阵计算单元寄存器相关的Operation

MMX的控制寄存器也都是32位的，且通过寄存器总线与RISC链接，其可编程寄存器主要是对要做运算的矩阵做一些必要的配置，prow和pcol分别代表coeff矩阵的行数和列数，w代表data矩阵的列数，n\_acc代表要计算最终结果图每个点要累加的次数，prow、pcol、w、n\_acc均位于MMX的CTRL0控制寄存器内。CTRL1控制寄存器主要配置计算的shift大小和取舍规则，CTRL3控制寄存器可配置两个矩阵做计算时子矩阵分块的大小，以及矩阵步长stride。当使用卷积指令时，还可以配置relu寄存器设置relu值。同样的，还有控制一次读取执行大小的vl寄存器，和配置指令执行次数的iter寄存器。本文根据可编程控制寄存器的功能，在DSP Dialect中为所有的可编程寄存器构造了相应的Operation。

|  |  |  |  |
| --- | --- | --- | --- |
| DSP Dialect Operation | 操作数 | 下降转换到的LLVM Dialect Operation | 作用 |
| DSP\_MMXSetCTRL0Op | pcol prow w n\_acc | LLVM\_SetCrOp  LLVM\_LshrOp  LLVM\_XorOp  LLVM\_ShlOp  LLVM\_AndOp | 配置coeff矩阵的行、列，data矩阵的列数，和最终每个点累加次数 |
| DSP\_MMXSetCTRL1Op | intrnd shift0 shift1 | LLVM\_SetCrOp  LLVM\_LshrOp  LLVM\_XorOp  LLVM\_AndOp  LLVM\_OrOp | 配置shift大小和取舍模式 |
| DSP\_MMXSetCTRL3Op | Divsion stride divied\_pcol | LLVM\_SetCrOp  LLVM\_ShlOp  LLVM\_XorOp  LLVM\_AndOp | 配置划分子矩阵大小和步长 |
| DSP\_MMXSetReluOp | relu\_val relu\_zero | LLVM\_AndOp  LLVM\_SetCrOp | 配置relu值 |
| DSP\_MMXSetVLOp | vector\_length | LLVM\_SetCrOp | 配置一次并发执行向量长度 |
| DSP\_MMXSetIterOp | iter | LLVM\_SetCrOp | 配置此向量指令执行次数 |

表3.4.1-1 MMX寄存器相关Operation

如表3.4.1-1所示，MMX可编程寄存器相关的Operation的操作数在MLIR中都是一个Value的示例，在LLVM IR中，所有的Value的实例都是一个SSA,所以在下降转换成LLVM Dialect的时候，需要对这些操作数做一系列的操作。以DSP\_MMXSetCTRL0Op为例：

|  |
| --- |
| 方法名： matchAndRewrite()  输入：  要转换的Operation:DSP\_MMXSetCTRL0Op  步骤：  通过该Operation获取到其操作数。  声明一个整数变量result,用于保存最终操作的结果  for every operand do  按照寄存器的有效位置对result进行移位运算。  end for  使用result构造一个Value类型（在生成的LLVM IR中是一个SSA) vResult  LLVM\_setCR(CTRL0,vResult)  在该IR中删除DSP\_MMXSetCTRL0Op节点 |

### 构建矩阵计算单元向量指令的Operation

MMX支持的向量指令只有矩阵乘和卷积两类，其中opcode为“mmc”的为矩阵乘，opcode为“mmcbias”的为卷积。支持的数据类型也只有有符号和无符号的char和short数据类型。同样的，MMX的向量指令在LLVM IR中是内联汇编的形式存在的，所以依然需要使用LLVM\_InlineAsmOp来实现MMX的向量指令Operation。Operation的命名规则同向量计算单元的命名规则一致，例如DSP\_MMXmmccccOp是一个矩阵相乘的Operation,高层的IR在使用时可能会在LLVM IR产生一条类似“mmc 0.char 1.char 2.char”矩阵相乘的内联汇编指令，其操作数和结果的数据类型都是char类型，并且data矩阵位于VMEM Block编号为1的array里，coeff矩阵位于VMEM Block编号为2的array里，运算后的结果保存在VMEM Block的编号为0的array里，具体的取值模式和数据地址由VMEM配置。

### 测试

矩阵计算单元的测试文件是mlir/test/Dialect/mmx.mlir，其主要内容如下：

|  |
| --- |
| ; Check DSP Dialect mmx Operation ---> LLVM Dialect Operation  ; RUN : DspCompiler %s -emit=mlir-llvm 2>&1 | FileCheck %s  ; Check dsp.mmx\_set\_ctrl0  CHECK: {{%[0-9]+}} = llvm.riscv.set.cr({{%[0-9]+}} : i64, {{%[0-9]+}} : i64) : (i64, i64) -> ()  ...  ; CHECK: llvm.inline\_asm has\_side\_effects "mmcbias [0].char, [1].char, [2].char, relu", "" : () -> !llvm.void  func @test\_mmx() {  dsp.mmx\_set\_ctrl0(3,3,3,1)  ...  dsp.mmx\_mmccccbias(0,1,2)  dsp.return  } |

该测试文件@test\_mmx函数里是DSP Dialect的IR，CHECK注释里是LLVM Dialect的IR。该测试文件首先将所有控制寄存器相关的Operation都做了测试，检查是否生成了对应的LLVM Dialect Operation。同时为了保证DSP Dialect的完整性，该文件对MMX所有的向量指令Operation也都做了全面测试,通过llvm-lit脚本工具运行测试测试文件：

|  |
| --- |
| $~/build/bin/llvm-lit mmx.ll  -- Testing: 1 tests, 1 workers --  PASS: LLVM :: CodeGen/RISCV/allspark/mmx.ll (1 of 1)  Testing Time: 0.36s  Passed: 1 |

## DSP数据存储模块

VMEM是DSP中的数据存储模块，DSP中除了RISC的栈和堆在program memory和data memory,其他计算单元需要的计算数据都只能存储在VMEM里。除了可以保存大量的数据，VMEM还有地址产生器，根据控制寄存器的配置可以自动产生数据地址，产生数据地址的规则由RISC配置相应的CONFIG实现,这便是取指模式。DSP中的向量计算模块和矩阵计算模块在执行向量指令之前，会先使用RISC将VMEM中对应的array块配置取值模式，然后向量计算模块和矩阵计算模块会根据地址产生器产生的地址取出对应array块中的数据进行计算，计算完成后，以同样的方式将结果写回到对应的array块中。所以在构建DSP Dialect时，VMEM相关的Operation就是要实现VMEM中的各种取指模式。

### 线性取指

线性取指模式是最简单取指模式，即地址产生器会根据所配置的基地址每次加1，从所配置起始地址开始往后读写数据。

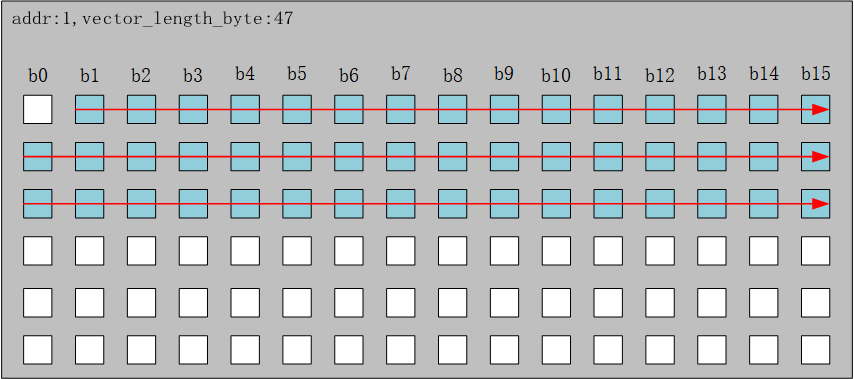


图20 线性取指

如图20所示，每一个方框代表可以存储一个byte,配置的基地址是1，需要取出47 bytes的数据，那么地址产生器就会从基地址1开始，在基地址的基础上依次累加，直到向量计算单元或矩阵计算单元取完所需的数据为止。所以线性取指模式适用于一些连续的数据计算，如向量间的计算等。线性取指在DSP Dialect中的Operation为“DSP\_VMEMSetLinearModeOp”,它需要接收两个Vlaue类型的参数，一个为基地址，一个为VMEM array对应的编号,其下降转换到LLVM Dialect的伪代码如下所示：

|  |
| --- |
| 方法名： matchAndRewrite()  输入：  要转换的Operation:DSP\_VMEMSetLinearModeOp  步骤：  通过该Operation获取到其操作数,addr\_base和arrayNum  While  检查该array里有无其他模块正在使用，若有，则等待；若没有，则跳出while循环  end while  LLVM\_setCR(CONFIG\_Mode,LINEAR | arrayNum)  LLVM\_setCR(CONFIG\_addr,addr\_base)  在该IR中删除DSP\_VMEMSetLinearModeOp节点 |

### 循环取指

线性取指在一定程度上可以满足向量间的计算，但是当大量的向量与某一个向量做运算时却不是很方便，于是就通过DSP Dialect的Operation来实现循环取指模式。循环取指就是指给定一定长度的向量，当它与其他很多向量做同样的运算时，它本身不是像线性取指那样继续往下加1，而是到达尾部后从第一个数据循环读取。

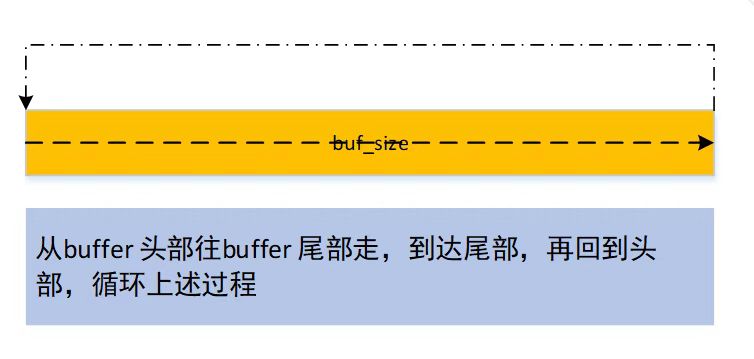


图21 循环取指

如图21 所示，循环取指就是给定buf\_size的大小和基地址后，地址产生器会从基地址开始增加，到达尾部之后会再次回到头部，循环这个取指过程。在DSP Dialect中循环取指的Operation为“DSP\_VMEMSetModuloModeOp”,它接收三个Value类型的参数，分别保存需要的基地址，VMEM array编号和buf\_size，其中buf\_size是指循环数据的字节个数。其下降转换到LLVM Dialect的伪代码如下所示：

|  |
| --- |
| 方法名： matchAndRewrite()  输入：  要转换的Operation:DSP\_VMEMSetModuloModeOp  步骤：  声明一个整数变量result,用于保存最终操作的结果  通过该Operation获取到其操作数,addr\_base,arrayNum和buf\_size  While  检查该array里有无其他模块正在使用，若有，则等待；若没有，则跳出while循环  end while  使用arrayNum和buf\_size构造一个Value类型vResult  LLVM\_setCR(CONFIG\_Mode,Modulo| vResult)  LLVM\_setCR(CONFIG\_addr,addr\_base)  在该IR中删除DSP\_VMEMSetModuloModeOp节点 |

### M1D取指模式

循环取指模式虽然解决了同一个向量与其他很多向量的计算时的取指方式，但是它无法解决一批向量与另外一批向量的取指问题，在大多数情况下，DSP要做的是一些大批量的，有规律的数据计算，所以在DSP Dialect中需要实现M1D取指模式。相较于循环取指，M1D在其基础上增加了一个inc0变量，作为下次一个buf\_size大小的数据从基地址取指的偏移，这个偏移是固定的，即每读取完buf\_size的数据后，地址产生器都会从上一次的基地址上增加inc0,然后再读取buf\_size个数据，以此类推。

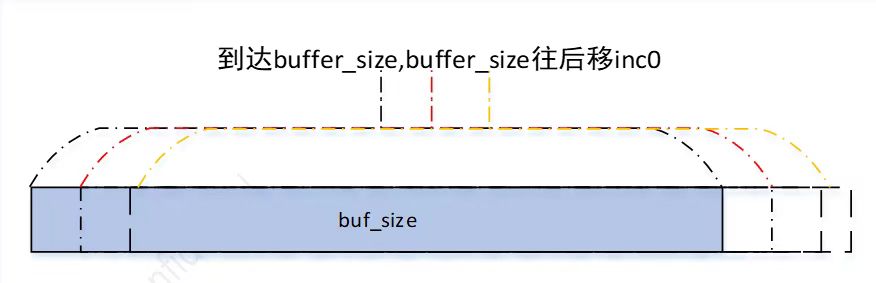


图22 M1D取指

如图22所示，M1D取指模式在读取完buf\_size数据后会往后固定偏移inc0地址读取数据，M1D取指模式在DSP Dialect的Operation为“DSP\_VMEMSetM1dModeOp”,它接收四个参数，其中三个参数与循环取指一样，inc0参数为每次偏移的大小，偏移的单位以字节为单位，其下降转换到LLVM Dialect的伪代码如下所示：

|  |
| --- |
| 方法名： matchAndRewrite()  输入：  要转换的Operation:DSP\_VMEMSetM1dModeOp  步骤：  声明一个整数变量result,用于保存最终操作的结果  通过该Operation获取到其操作数,addr\_base,arrayNum、buf\_size和inc0  While  检查该array里有无其他模块正在使用，若有，则等待；若没有，则跳出while循环  end while  使用arrayNum、buf\_size和inc0构造一个Value类型vResult  LLVM\_setCR(CONFIG\_Mode,M1D| vResult)  LLVM\_setCR(CONFIG\_addr,addr\_base)  在该IR中删除DSP\_VMEMSetM1dModeOp节点 |

### M2D取指模式

M2D取指模式在M1D的基础上，增加了inc1和repeat\_cnt0两个参数。repeat\_cnt0是指重复M1D取出buffer\_size数据的个数，取完repeat\_cnt0×buffer\_size个数据后，将在第一次基地址的基础上偏移inc1个字节，然后再按M1D取指模式取出repeat\_cnt0个buffer\_size数据，以此类推。

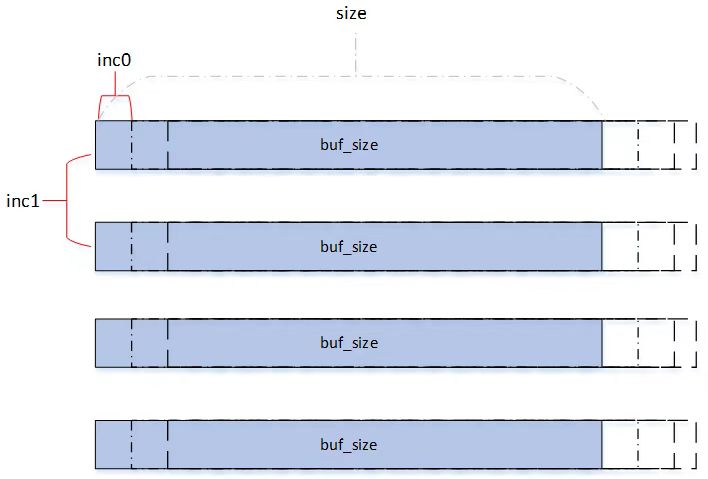


图23 M2D取指模式

如图23所示，M2D取指模式在M1D的基础上增加了inc1和repeat\_cnt0两个参数，第一行在按照M1D取指模式取出repeat\_cnt0个buf\_size之后，将会按照第一次基地址偏移inc1个字节，然后再按照M1D的取指模式读取repeat\_cnt0个数据。在DSP Dialect中，M2D取指模式的Operation的名称为“DSP\_VMEMSetM2dModeOp”,它接收六个参数，其下降转换到LLVM Dialect的伪代码如下所示：

|  |
| --- |
| 方法名： matchAndRewrite()  输入：  要转换的Operation:DSP\_VMEMSetM2dModeOp  步骤：  声明一个整数变量result,用于保存最终操作的结果  通过该Operation获取到其操作数,addr\_base,arrayNum、buf\_size、inc0、repeat\_cnt0和inc1  While  检查该array里有无其他模块正在使用，若有，则等待；若没有，则跳出while循环  end while  使用arrayNum、inc0、repeat\_cnt0构造一个Value类型vResult  LLVM\_setCR(CONFIG\_CTRL0,inc1)  LLVM\_setCR(CONFIG\_size,buf\_size)  LLVM\_setCR(CONFIG\_Mode,M2D| vResult)  LLVM\_setCR(CONFIG\_addr,addr\_base)  在该IR中删除DSP\_VMEMSetM2dModeOp节点 |

### 测试

DSP存储模块VMEM的测试文件是mlir/test/Dialect/vmem.mlir，其主要内容如下：

|  |
| --- |
| ; Check DSP Dialect vmem Operation ---> LLVM Dialect Operation  ; RUN : DspCompiler %s -emit=mlir-llvm 2>&1 | FileCheck %s  ; Check dsp.vmem\_set\_linear\_mode  CHECK: {{%[0-9]+}} = llvm.riscv.set.cr({{%[0-9]+}} : i64, {{%[0-9]+}} : i64) : (i64, i64) -> ()  ...  ; CHECK: llvm.cond.br {{%[0-9]+}}  func @test\_vmem() {  dsp.vmem\_set\_linear\_mode(0x8000,0)  ...  dsp.vmem\_set\_2d\_mode(0x16000,1,200,10,100)  dsp.return  } |

该测试文件@test\_vmem函数里是DSP Dialect的IR，CHECK注释里是LLVM Dialect的IR。该测试文件首先将DSP VMEM存储系统的Opertaion都调用做了测试，检查是否生成了对应的LLVM Dialect Operation，为了保证DSP Dialect的完整性，该文件对VMEM所有的Operation也都做了覆盖测试,通过llvm-lit脚本工具运行测试测试文件：

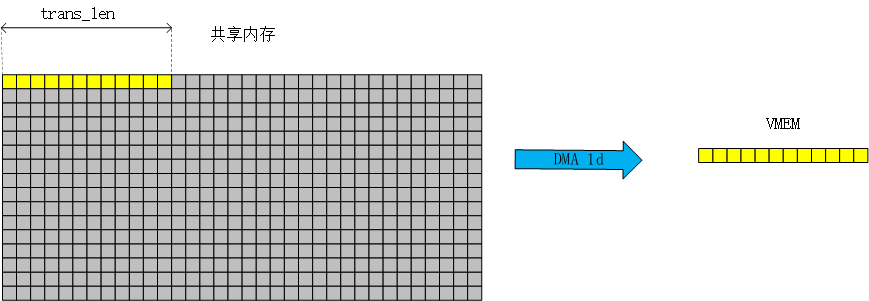
|  |
| --- |
| $~/build/bin/llvm-lit vmem.ll  -- Testing: 1 tests, 1 workers --  PASS: LLVM :: CodeGen/RISCV/allspark/vmem.ll (1 of 1)  Testing Time: 0.08s  Passed: 1 |

## DSP数据搬运模块

在整个SOC中DSP一般作为辅处理器，来处理算法中大量数据运算，于是主处理器与DSP之间需要共享内存来完成数据之间的交互。DSP中没有文件系统，DMA是负责DSP数据搬运的模块，在使用DSP的时候需要先使用DMA将所需要计算的数据从共享内存搬运到DSP的存储模块VMEM里，完成运算之后同样需要将计算结果从DSP内的存储模块VMEM搬运到共享内存的指定位置。DSP中使用DMA来完成主处理器与DSP的数据交互是因为DMA的搬运速度非常快，非常适合大批量的数据搬运。DMA有读任务、写任务两个任务队列，并且支持读写通道并行传输，支持读写任务保序执行，支持多任务间按优先级排队处理，而在传输模式上，DMA主要分为1D传输和2D传输两种。

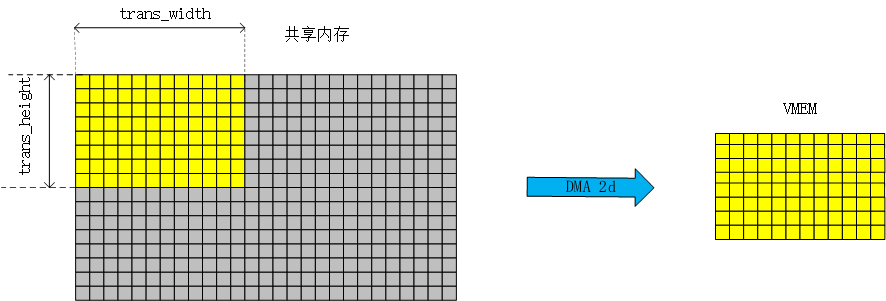
### 功能描述

1. DMA 1D 传输



如图所示，是DMA从DSP片外的共享内存搬运到DSP内部存储模块VMEM的示意图，从VMEM搬出到共享内存与之类似。DMA 1D传输是在一维上对连续数据的搬运过程，它需要接收传输两端的起始地址、传输数据的大小以及传输数据的类型等参数，然后会按照配置的优先级将搬运过程分别压入读队列和写队列中，按照队列的优先级完成读写任务。DMA 1D传输支持无符号和有符号char、short、int数据类型，同时支持不等待任务完成，由外部等待的情况。

1. DMA 2D 传输



如图所示，相较于DMA 1D传输，DMA 2D传输是在二位上对连续数据的搬运过程，它不需要传入传输数据的大小，而是增加了 trans\_width作为传输宽度，trans\_height作为传输高度，传输数据的大小可有trans\_width×trans\_height得出，其他参数配置同DMA 1D传输。

### DMA Operation实现

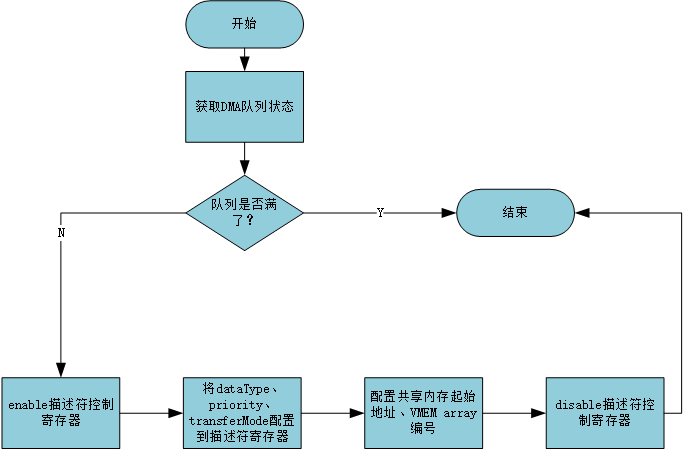
DMA是DSP中的功能模块，按照DSP Dialect Opertaion模块化、原子化的原则，在设计实现DMA的Operation时，需要先将DMA传输按照其传输模式划分成两类Operation，即DMA 1D传输和 DMA 2D传输，然后这两类传输模式按照其功能可详细划分两种Operation,即一种负责从共享内存搬进DSP VMEM的Operaton，和一种负责从VMEM搬出到共享内存的Operation。按照DMA的功能划分Operation的优势是方便上层Dialect下降转化时使用，使其转换的代码更具有可读性和易维护性，例如Affine Dialect是较为上层的Dialect,其关注点在于多面体循环优化，其声明的AffineDmaStartOp和AffineDmaWaitOp是一种在上层IR的抽象，目的是加入了DMA传输类型的描述，而不关注其具体的实现。而DSP Dialect是一种偏向于硬件的Dialect,是提供硬件功能的一种IR描述,它需要提供不同的搬运功能来使上层Dialect Operation的转换更具有可读性，如果使用Affine Dialect下降转换到DSP运行，那么则需要具体关注DSP上DMA是否是搬进还是搬出。

以DMA 1D传输为例，在MLIR中实现DMA 1D传输时，根据MLIR提供的ODS框架，首先需要在DSP的Ops.td中添加DSPDma1DTransferInOp和DSPDma1DTransferOutOp的声明。

|  |
| --- |
| def DSPDma1DTransferInOp : DSP\_Op<"dma\_1d\_transfer\_in">,Arguments<(ins I8Attr:$priority,I8Attr:$dataType,I64Attr:$ddr\_addr,I64Attr:$trans\_len,I8Attr:$vmem\_array\_num,I8Attr:$is\_wait)>,Results<(outs I1Attr:$status)> {  ...  } |

如上表是DSPDMA1DTransferInOp的声明，由于DMA的任务队列在任务满的时候会将新任务丢弃掉，从而会导致搬运失败，为了方便调试和Debug,需要将搬运任务是否成功压入任务队列作为返回值返回，所以DMA的Operation都具有一个i1类型的返回值，如果返回1则任务压入队列成功，为0则失败。其次因为该DSP的RISC核和共享内存的地址都是64位的，所以该Operation的参数中共享内存的地址是以I64Attr的形式保存的。

然后需要在mlir/lib/Dialect/DSP/Tranforms/LegalizeForLLVMExport文件实现声明的Operation的转换，需要将声明的每一个Operation都构建一个Lowering类，实现其中的匹配重写方法，其逻辑如下：



首先需要使用LLVM\_getCrOp获取到当前DMA任务队列的状态，如果当前任务队列已经满了，则返回0结束；如果没有满，则需要将描述符控制寄存器使能，使能描述符寄存器才能将DMA的任务压入任务队列里中，然后将Operation的参数按照一定的顺序去配置到相应的寄存器上，最后关闭描述符控制器，返回1。其中配置参数到相应寄存器的顺序是由硬件决定的，在使能描述符寄存器之后，硬件会按照固定的顺序将读取各个寄存器，然后将任务压入任务队列中。由于整个转换的过程中不能出现其他Dialect的Operation，所以DMA相关的Operation下降中参数之间的移位、与运算、或运算都只能使用LLVM Dialect所提供的相应Operation，才能将DSP Dialect的Operation完全下降到LLVM Dialect。最后，在配置DSP Dialect转换到LLVM Dialect的pattern列表中加入实现的Lowering类。

### 测试

DSP搬运模块VMEM的测试文件是mlir/test/Dialect/dma.mlir，其主要内容如下：

|  |
| --- |
| ; Check DSP Dialect dmaOperation ---> LLVM Dialect Operation  ; RUN : DspCompiler %s -emit=mlir-llvm 2>&1 | FileCheck %s  ; Check dsp.vmem\_set\_linear\_mode  ; CHECK: llvm.riscv.set.cr({{%[0-9]+}} : i64, {{%[0-9]+}} : i64) : (i64, i64) -> ()  ...  ; CHECK: {{%[0-9]+}} = llvm.riscv.get.cr({{%[0-9]+}} : i64) : (i64) -> (i64)  func @test\_dma() {  dsp.dma\_1d\_transfer\_in(0,1,0x2000,100,0,1)  ...  dsp.dma\_2d\_transfer\_out(0,1,0x2000,100,20,0,1)  dsp.return  } |

该测试文件@test\_dma函数里是DSP Dialect的mlir，CHECK注释里是LLVM Dialect的mlir。该测试文件首先将DSP DMA搬运模块的Opertaion都调用做了测试，为了保证DSP Dialect的完整性，将DMA相关的Operation即DSPDma1DTransferInOp、DSPDma1DTransferOutOp、DSPDma2DTransferInOp、DSPDma2DTransferOutOp四个Operation都做了覆盖性测试，检查其是否正确的生成了对应的LLVM Dialect的MLIR,通过llvm-lit脚本工具运行测试测试文件：

|  |
| --- |
| $~/build/bin/llvm-lit dma.ll  -- Testing: 1 tests, 1 workers --  PASS: LLVM :: CodeGen/RISCV/allspark/dma.ll (1 of 1)  Testing Time: 0.05s  Passed: 1 |