计算机科学
COMPUTER SCIENCE

高阶密码算子在FPGA的编译优化与实现

裴雪, 魏帅, 邵阳雪, 于洪, 葛晨洋

引用本文

裴雪, 魏帅, 邵阳雪, 于洪, 葛晨洋. 高阶密码算子在FPGA的编译优化与实现[J]. 计算机科学, 2024, 51(11A): 231200184-11.

PEI Xue, WEI Shuai, SHAO Yangxue, YU Hong, GE Chenyang. Compilation Optimization and Implementation of High-order Cryptographic Operators on FPGA[J]. Computer Science, 2024, 51(11A): 231200184-11.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.231200184

高阶密码算子在FPGA的编译优化与实现

裴 雪[1] 魏 帅[1] 邵阳雪[2] 于 洪[1] 葛晨洋[1]
1 信息工程大学信息技术研究所 郑州 450002
2 嵩山实验室 郑州 450046

摘 要 针对密码算法的不同编译需求,提出一种不同粒度密码算子抽象方法,通过对不同粒度算子的编译优化及映射来解决高阶密码算子在FPGA上快速高效部署的问题。从密码算法中抽象出热点算子来构建算子库,使用多级编译优化对密码算法进行优化与部署,并通过数据张量化以及寄存器优化方法来提升高阶密码算子在VTA硬件架构的部署及运算效率。实验结果表明,采用张量化和寄存器优化方法,执行效率较原始编译部署方法提升了32倍,较OpenCL提升约34倍,且可以根据构建的算子库对密码算法进行快速开发与实现。

关键词:算子抽象;编译优化;代码生成;密码算法
中图分类号 TP311

Compilation Optimization and Implementation of High-order Cryptographic Operators on FPGA

PEI Xue[1],WEI Shuai[1],SHAO Yangxue[2],YU Hong[1] and GE Chenyang[1]
1 Institute of Information Technology,Information Engineering University,Zhengzhou 450002,China
2 Songshan Laboratory,Zhengzhou 450046,China

Abstract Aiming at the different compilation requirements of cryptographic algorithms,a method of abstracting cryptographic operators at different granularities is proposed. This method addresses the issue of rapid and efficient deployment of high-order cryptographic operators on FPGAs through compilation optimization and mapping of operators at different granularities. Hotspot operators are abstracted from cryptographic algorithms to construct an operator library. Multi-level compilation optimization is used to optimize and deploy cryptographic algorithms. Data tensorization and register optimization methods are employed to enhance the deployment and computation efficiency of high-order cryptographic operators on the VTA hardware architecture. Experimental results show that the execution efficiency using tensorization and register optimization methods is 32 times higher than the original compilation and deployment methods,and approximately 34 times higher than OpenCL. Additionally,the constructed operator library allows for the rapid development and implementation of cryptographic algorithms.

Keywords Operator abstraction,Compilation optimization,Code generation,Cryptographic algorithm

随着密码算法的飞速发展,早期的MD5,SHA1等密码算法已经不能满足安全性以及效率的需求。在此基础上发展出SHA-2,SHA-3系列算法,在提升算法安全性的同时,结构也愈加复杂,计算复杂度不断增加。密码算法要满足性能上的需求,不仅需要芯片算力的提升,也需要高效的编译技术。如何方便且高效地把上层应用通过编译部署到硬件上,提升运行效率,成为关键性问题。美国的Corelet[1]、PyNN[2]以及清华大学的天机芯的编译工具链都发挥了很大的作用,通过软硬件协同方法提升了计算性能。

硬件与上层应用相辅相成,硬件的多样性对于算法或人工智能(AI,Artificial Intelligence)模型部署既是机遇也是挑战。目前部署算法或模型的主流硬件平台包括FPGA,GPU,CPU,ASIC等。这些硬件平台在通用性、并行度、功率、推理速度等方面各有优劣。为实现硬件多样性,有效地将计算映射到硬件上以得到更高的性能,程序员必须熟悉目标硬件架构的特点,然后用领域专用语言改写程序代码或向代码中加入指导语。因为优化后的代码与目标硬件架构相关,所以源程序变得难以维护和移植,编译系统的易用性降低且软件开发时间变长。

为了弥补软件工具链的缺陷以减轻手动优化硬件的负担,工业界和学术界提出了几种流行的深度学习(Deep Learning,DL)编译器,将程序的优化转移到具有一定通用性的中间表示(Intermediate Representation,IR)层级,如TVM[3],Tensor Comprehension,Glow[5],nGraph[6],MLIR[7]。DL编译器的主要目的是将上层应用通过编译器的多级IR和特定的编译优化降低为硬件可识别的语言。深度学习编译器为密码算法提供了一种高效、自动化的解决方案,不仅提升了算法的性能,还简化了开发过程,增强了跨平台兼容性,并有助于充分利用现代硬件的计算能力。DL编译器相比传统编译器,在自动代码生成、分布式并行处理、针对

基金项目:国家重点研发计划重点专项(2022YFB4401401);嵩山实验室项目(纳入河南省重大科技专项管理体系)(221100211100-01)
This work was supported by the National Key Research and Development Program of China(2022YFB4401401) and Program of Songshan Laboratory(included in the management of Major Science and Technology Program of Henan Province)(221100211100-01).
通信作者:裴雪(peixue1949@163.com)

性优化、灵活性与性能平衡等方面具有优势。此外，现有的编译器还采用了通用编译器（如 LLVM[8]，OpenCL[9]）的成熟工具链，为各种硬件体系结构提供了更好的可移植性。现有编译器[10]系统通过加入多级 IR 将源程序的描述和面向硬件的优化解耦，在无须修改源程序描述的条件下提高编译效率，从而使得编译系统的易用性有了显著提升。引入 IR[11]层级优化可以在保证用户生产效率的情况下提高程序在硬件上的执行效率。

目前，基于 CPU+FPGA 异构计算平台进行程序设计及优化的工作很多。例如，Zheng[12]针对 OpenCL 的 CPU+FPGA 异构计算平台，通过多线程管理、命令队列管理以及负载均衡设计，使得 CPU 可以通过软件灵活使用哈希算法的通用加速器，提升加速器的计算效率，其中 SHA-256 算法比 CPU 获得 18.7 倍的性能提升，比 GPU 获得 2 倍性能提升。Zhang 等[13]提出了一个名为 FlexTensor 的自动优化框架，它使用修剪技术和机器学习技术为 CPU，GPU 和 FPGA 上的张量计算生成高性能调度，与 NVIDIA V100 GPU 上的 cuDNN 相比，平均加速提升 1.83 倍；与 Intel Xeon CPU 上的 MKL-DNN 相比，2D 卷积的平均加速度为 1.72 倍；与 VU9P FPGA 上的 OpenCL 基线相比，平均加速为 1.5 倍；与最先进的 NVIDIA V100 GPU 相比，平均加速速度为 2.21 倍。Zhao 等[14]提出一种新的环形平铺和融合结构，通过构建任意块形状和平铺融合策略，实现存储优化和内存层次优化，通过对神经网络、图像处理、稀疏矩阵计算和线性代数等众多领域提取 11 个基准测试，获得了 16%的性能提升。Li[15]针对数据安全 AES 算法，设计并实现向量化编程模型、核组间、核组内以及指令并行优化策略，充分挖掘处理器性能，并行优化后的 AES 算法在单个 SW26010 异构众核处理器上最大可获取 13.49GB/s 的吞吐量。以上相关工作有些是基于硬件架构或指令级别的方式进行优化来提高计算效率，有些是基于软件编译优化框架进行并行优化以提高计算效率，效率提升有限。且在这些硬件架构上开展加解密算法部署的相关工作也较少，将计算映射到硬件的效率较低。源程序的维护和移植还是较难，对用户端开发人员不够友好，开发人员需要较多的硬件底层知识的支撑，具体原因为以下几点：硬件底层知识的依赖性、性能调优的复杂性、跨平台兼容性的挑战、编程语言和工具的多样性、缺乏通用标准和接口等。本文将在研究工作中对现有工作的改进做相应补充。

在介绍本文工作前，首先解释一下文章中涉及的几个名词。“原子级算子”指一个简单的运算，例如加法运算；“函数级算子”指原子级计算的组合，例如在 MD5 算法中的一个 FF 函数运算；“算法级算子”指函数级和原子级算子的组合，例如一个 MD5 算法；“高阶算子”是一个复杂算子的统称，其包括函数级和算法级算子。

在本文中，面向杂凑类密码算法[16]构建高阶算子的需求，首先抽象出杂凑类密码算法基础热点算子，其次使用多级 IR 实现不同粒度热点算子，最后通过用户接口对多级中间表示算子的封装，构建高级语言算子调用接口来方便用户灵活使用，从而简化其开发和部署难度。通过多级 IR 编译映射、编译优化将算子映射到硬件[17]，以提高算法在硬件的计算效率。本文以 MD5 算法为例解决 3 个关键挑战：1）对 MD5 算法进行数据流分析，提取不同粒度算子通过编译优化为数据向量化提供支撑。2）计算特征信息提取，为支撑编译映射与硬件代码生成。3）实现基于多层 IR 算法特征到算子的匹配与重写，实现算法的高效编译映射。

本文的主要研究工作如下：

1）研究面向杂凑类密码算法的高阶算子，以 md5 为例子，对不同粒度算子抽象以及算子库的构建，方便多级编译优化框架构建领域专用语言的表示。

2）通过多级中间表示优化方法，实现不同粒度算子的编译映射以提高算子的灵活性，让用户可以根据需求灵活变动参数及算子组合。

3）实现高级语言编程接口，方便用户端使用，减少编程人员的工作量及后期维护成本。

4）生成硬件微内核并驱动整个系统高效执行计算任务。

5）对 md5 算法在编译映射过程中进行数据张量化，通过张量化数据重复使用、单指令多数据和单指令多线程减少寄存器文件带宽，提高计算效率，缩短硬件执行时间。

6）针对 md5 算法进行寄存器优化，进一步提升计算效率，提高寄存器访问效率。

1 研究现状及动机

FPGA 作为一种半定制电路，集成了 ASIC 硬件实现密码算法的优点，使用 FPGA 实现密码算法，同时满足计算性能、易于维护的需求。但使用 FPGA 需要大量硬件相关知识，并且熟悉数字电路设计，对开发人员的要求较高。Xilinx 和 Intel 都相继推出 FPGA 的高层次综合工具[18]（High-Level Synthesis，HLS），支持使用 C/C++等高级语言开发 FPGA，将复杂的硬件底层电路设计交于编译器负责。还进一步推出了支持 OpenCL 的开发套件 Xilinx Vitis 以及 Intel FPGA SDK for OpenCL[19]，为 FPGA 提供了便利。但是这些基本上都是针对传统编译器对密码算法支持，灵活性较差，对开发人员不友好，很难进行优化。不同硬件有不同的编译规则，需要一个较为通用的编译器来统一编译规则。现有的深度学习编译器可以作为软件和硬件之间的公共组件及桥梁，只需开发一次就能够为不同设备生成最优代码，可以方便地对循环进行各种变换以及参数调优，从而得到不同参数组合和硬件最佳算子实现。目前 TVM 编译框架生态完善、开源，可以导入各种深度学习模型，通过多级 IR 表示及中间代码优化映射到不同的硬件后端。本文参考 TVM 框架设计来支持密码领域的编译优化及硬件部署，以减轻应用的计算和内存负担，从而降低其开发和部署难度。主要动机和意义有如下：

1）深度学习编译器支持张量化运算，密码算法解密难度大，密钥空间大，解密时间长，通过编译框架进行硬件加速，用数据张量化来减少算法执行时间，提高密码算法在硬件上执行效率，减少硬件执行时间。

2）通过多级 IR 编译优化将源程序的描述和硬件优化进行解耦，在保证用户生产效率的前提下提高程序在硬件上的执行效率，提升密码算法的性能及可维护性。

3）通过对复杂密码算子的编译优化、数据张量化以减少寄存器文件带宽，提高计算效率。通过寄存器优化以提高访问效率，减少编程人员的工作量及后期维护成本，以满足密码算法的计算性能需求。

2　系统框架

高阶密码算子在 FPGA 的代码设计与实现采用“端-端”系统框架，将密码算子作为编译输入，通过多级编译优化框架进行算法部署。如图 1 所示，针对密码算法通过多级 IR 优化以及代码映射进行硬件部署，提高程序在硬件上执行效率。

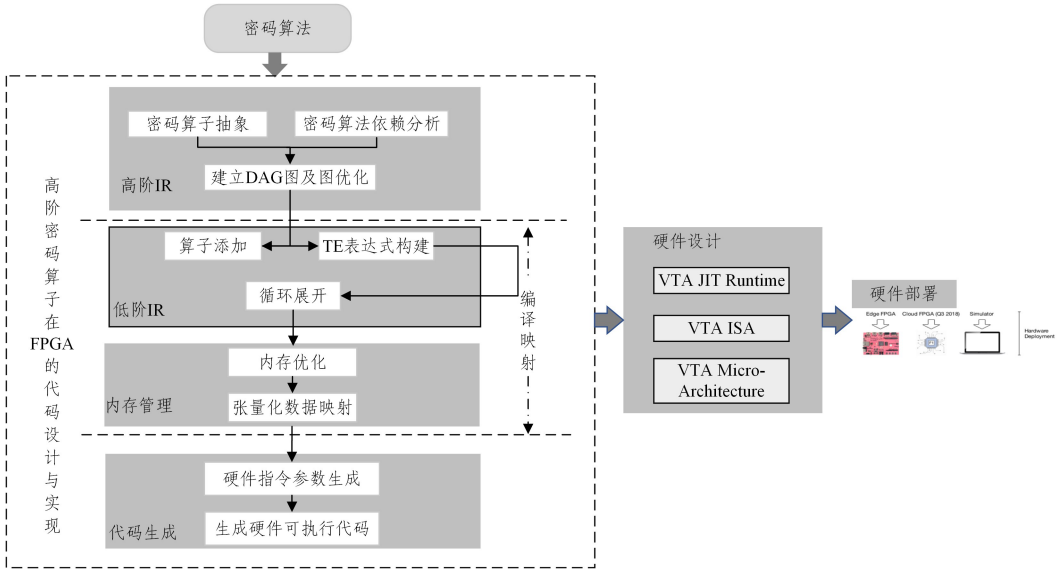

图 1　编译系统框架示意图

Fig. 1　Schematic diagram of the compilation system framework

编译系统框架由高阶 IR、低阶 IR、内存管理、代码生成、硬件设计以及硬件部署 6 个模块组成。其中，高阶 IR、低阶 IR、内存管理、代码生成 4 个模块是本文工作工作的核心。由于本文工作基于多级编译优化框架来实现，因此区分高阶 IR 和低阶 IR。高阶 IR 在编译器前端负责与硬件无关的转换和优化。低阶 IR 在编译器后端负责硬件的优化代码生成和编译。

1）高阶 IR 模块，对密码算法进行算法依赖分析，将其抽象为不同粒度的算子，通过领域专用语言生成有向无环图（Directed Acyclic Graph，DAG），进行计算图的优化，例如操作符融合、静态内存重用、数据布局转换等。

2）低阶 IR 模块，通过对张量表达式（Tensor Expression，TE）的构建，对 TE 循环展开进行边界信息抽取并转换为硬件相关的参数信息，在计算过程中对数据进行张量化来提高计算效率。

3）内存管理模块，对语句实例以及访存操作之间的映射关系进行分析，对数据的访存进行优化，减少 load 和 store 的次数以到达优化目的。

4）代码生成模块，通过算子抽象、算法依赖分析、图优化、算子添加、TE 表达式的构建、循环展开、内存优化、张量化数据映射之后，将 for 循环映射为硬件指令参数，再由多功能张量加速器（Versatile Tensor Accelerator，VTA）生成硬件可执行代码。

5）硬件设计模块，参考 VTA[20] 硬件架构进行硬件设计，主要包括取指模块、计算模块和存储模块。VTA 架构是可完全参数化架构，包括即时编译器、两级指令集架构（Instruction Set Architecure，ISA）。其中两级 ISA 指：（1）任务级 ISA，用来协调并行的计算与访存任务；（2）微码 ISA，实现了一系列的张量-张量计算算子。运行时系统包含 JIT 编译器，可在线生成代码，管理异构执行，提升 VTA 架构的使用效率。

6）硬件部署模块实现整个算法在硬件上的部署。编译框架中嵌入了 VTA 硬件提供的 API，编译并没有直接生成 VTA 指令，而是通过 VTA 的 API 控制 VTA 运行时系统，即 JIT（Just In Time，即时）编译，通过 PRC（Remote Procedure Call，远程过程调用）将编译后的文件上传到 FPGA 设备。之后生成 VTA 微内核，展开 VTA 指令，并驱动整个异构系统执行计算任务。在生成 VTA 指令之后，向 FPGA 设备的 PL（Programmable Logic，可编程逻辑）写入运行信号，VTA 的 HLS 加速器部分（FPGA 的 PL）开始以任务级并行执行方式执行计算任务，最终实现硬件部署。

3　高阶 IR 的构建

本章主要介绍面向杂凑类密码算法高阶算子的抽象，通过密码算法依赖建立 DAG 图并进行图优化，解决密码算法领域算子适配问题。同时提高密码算法在 FPGA[21] 上的高效计算，减少硬件执行时间。

3.1　密码算子抽象

为将计算有效地映射到各种不同硬件上，第一步是实现不同粒度的算子。首先分析密码算法，对其进行原子级抽象，抽取出一些共性算子和专用算子，实现原子级算子。然后通过函数级运算规则（比如，码算法需求及算子之间的数据流关系）对原子级算子进行抽象，实现函数级算子。再由函数级算子根据算法级规则构建算法级算子，即粗粒度算子，以满足不同应用需求或开发需求。最终实现多指令操作功能的算子。

图 2 所示为算子抽象原则。

本文设计了一种算子抽象原则，该原则主要分为算法级、函数级和原子级 3 部分，颗粒度逐级下降。原子级中的抽象原子是根据编译抽象过程中无法细分的运算操作而确定的，例如逻辑运算（与、或、非、异或）、算数运算（乘、加、赋值）、移位运算、替/置换操作等，原子级的算子代表着程序的最小运算单元。随着算子颗粒度的变大，函数级和原子级的算子多以多项式组合的形式呈现，以满足定制化算法的需求。函数

级算子作为抽象原则的中间部分，为上层算法级的专用算子提供下层原子级的算子组合模板。最上层的算法级算子直接对应用户的应用，提供一套独立的算法，例如 MD5。本文设计的算子抽象原则通过不同级别的算子，满足用户的定制化开发需求。

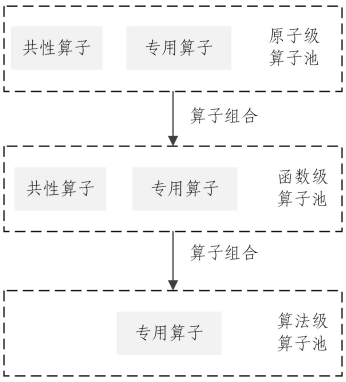

图 2 算子抽象原则

Fig. 2 Principles of operator abstraction

本文设计的抽象原则主要针对杂凑类密码算法，通过算子添加、编译优化来实现。主要支持 MD5 原子级、函数级和算法级算子，为用户定制化的密码算法应用构建提供支撑。下面给出了 3 个不同级别算子的用户端编程接口。如代码 1:“原子级”算子编程接口所示，原子级算子支持最细粒度的算子，灵活性最高。本文通过第 4 章(算子添加、TE 表达式构建)和第 5 章(内存优化、张量化数据映射)的方法，实现了杂凑类密码算法的原子级算子。用户可以根据原子级算子和特定规则定义任意算子。

代码 1 “原子级”算子编程接口

```
for i in range(64):# 64 轮运算
    if i< 16:
        bAndc=relay.bitwise_and(b,c)
        Notb=relay.bitwise_not(b)
        NotbAndd=relay.bitwise_and(Notb,d)
        f=relay.bitwise_or(bAndc,NotbAndd)
        Flag=i # 用于标识处于第几组信息
    elif i< 32:
        //用 relay 重构 f =(b&d) |(c&(-d))
        bAndd=relay.bitwise_and(b,d)
        Notd=relay.bitwise_not(d)
        NotdAndc=relay.bitwise_and(Notd,c)
        f=relay.bitwise_or(bAndd,NotdAndc)
        flag =(5 * i + 1)%16
    elif i< 48:
        //f=(b∧c∧d)
        bXorc=relay.bitwise_xor(b,c)
        f=relay.bitwise_xor(bXorc,d)
        flag =(3 * i+ 5) % 16
    else:
        //f=c∧(b|(-d))
        Notd=relay.bitwise_not(d)
        bNotd=relay.bitwise_or(b,Notd)
        f=relay.bitwise_xor(c,bNotd)
        flag =(7 * i) % 16
```

如代码 2:“函数级”算子编程接口所示，函数级算子，灵活性相对较高，可以随意组合，常量参数可以根据用户需求灵活变动。

代码 2 “函数级”算子编程接口

```
for i in range(64):# 64 轮运算
    if i< 16:
        f=relay.FF(b,c,d)
        flag=i # 用于标识处于第几组信息
    elif i < 32:
        //用 relay 亚构 f =(b&d)|(c&(-d))
        f= relay.GG(b,c,d)
        flag=(5*i+1)%16
    elif i < 48:
        //f=(b∧c∧d)
        f=relay.HH(b,c,d)
        flag=(3*i+5)%16
    else:
        //f=c∧(b|(~d))
        f=relay.II(b,c,d)
        flag =(7*i)%16
tmp=relay.add(a,f)
tmp=relay.add(tmp,L[i])
tmp_w=relay.const(w[flag],dtype='int64')
tmp=relay.add(tmp,tmp_w)
tmp_c=relay.const(0xffffffff,dtype="int64")
x=relay.bitwise_and(tmp.tmp_c)
```

如代码 3:“算法级”算子编程接口所示，算法级算子，灵活性最差，直接满足一些算法级用户的需求，不需要再定义任何计算，降低了用户端开发的难度。

代码 3 “算法级”算子编程接口

```
import mmpy
message = {"0":"1545sdsa","1":"dewe32ed",…,"15":"we3efdrtt4g"}
mmpy.md5(message)
```

本文以 MD5 算法为例详细介绍本文的算子抽象设计原则，表 1 列出了 MD5 算法的“原子级”抽象算子的说明。

表 1 算子运算描述

Table 1 Operator operation description

运算	说明
&	逻辑与
|	逻辑或
∧	逻辑异或
~	逻辑取非
$ROTL^n(x)$	循环左移，n 代表移动位数

3.2 密码算法依赖分析

下面对 MD5[21] 进行算法分析，并进行函数级算子抽象，MD5 一共有 64 轮迭代运算，这里把一个 step 作为一个 kernel。根据 MD5 算法内部的数据依赖关系，一共分为 4 个计算模块，如式(1)所示。其中输入的起始常数 b,c,d，对应的值分别为 0Xefcdab89，0X98badcfe，0X10325。

$$\begin{cases} F(b,c,d)=(b\&c)|(\sim b\&d) \\ G(b,c,d)=(b\&d)|(c\&(\sim d)) \\ H(b,c,d)=(b\wedge c\wedge d) \\ I(b,c,d)=(c\wedge(b|(\sim d))) \end{cases} \tag{1}$$

MD5 杂凑加密算法主要由式(2)所示的 FF,GG,HH,II 4 个函数计算完成，其算法分为 4 轮运算，每轮执行 4 个函数

中的一种 16 次，一种 64 次迭代运算。每次运算的输出结果作为下次运算的输入。

$$\begin{cases} FF(a,b,c,d,n,i)=b+ROTL^n(a+F(b,c,d)+W_i+T_i) \\ GG(a,b,c,d,n,i)=b+ROTL^n(a+G(b,c,d)+W_i+T_i) \\ HH(a,b,c,d,n,i)=b+ROTL^n(a+H(b,c,d)+W_i+T_i) \\ II(a,b,c,d,n,i)=b+ROTL^n(a+I(b,c,d)+W_i+T_i) \end{cases} \tag{2}$$

其中，常量 a 为 0X657452301，每次参与运算的加密消息为 512 bit，并将其分为 16 个 32bit 大小的数据块，表示为 W_i，T_i 是随机产生的常数数值，通过正弦函数产生，n 表示循环左移的位数。其中 $i\in[0,64]$。再将最后一轮输出的结果 a，b，c，d 级联起来，最终输出 128 bit 的加密结果。

下面对 MD5 进行算法级算子抽象，MD5 一共有 64 轮迭代运算，这里把一个完整 MD5 算法作为一个 kernel，根据 MD5 算法数据的之间依赖关系，分成 4 个计算模块，如式(3)所示：

$$\begin{cases} M(b,c,d)_0=(b\&c)|((\sim b)\&d) \\ M(b,c,d)_1=(b\&d)|(c\&(\sim d)) \\ M(b,c,d)_2=(b\wedge c\wedge d) \\ M(b,c,d)_3=(c\wedge(b|(\sim d))) \end{cases} \tag{3}$$

根据式(3)，MD5 算法的 64 轮迭代计算可以用式(4)来完成。

$$KK(a,b,c,d,n,k,i)=b+ROTL^n(a+M_k(b,c,d)+W_i+T_i) \tag{4}$$

其中，$a=$0X657452301，$b=$0Xefcdab89，$c=$0X98badcfe，$d=$0X10325476，MD5 的 64 轮计算通过 KK 函数的计算来完成，每一轮计算输出的结果作为下一轮计算的输入。W_i 表示输入的 512 比特 message，T_i 表示正弦函数产生的位随机数，n 表示循环左移的位数。其中 i 的取值范围为(0～63)，k 的取值范围为(0～3)。最后一轮输出的结果 a，b，c，d 将被级联起来，最终输出 128 bit 的加密结果。

整个 MD5 作为一个 kernel 的方法，内存连续，不需要频繁进行数据交换，内存利用率比一个 step 作为一个 kernel 的方法更高。

3.3　建立 DAG 图及图优化

编译框架从密码算法中接收模型的输入，通过对不同粒度算子的抽象，以及密码算法依赖分析，建立 DAG 图并进行图优化，最后生成指令完成到硬件的部署。不论是对指令的优化还是指令的生成，一个图的结构能清晰地描述数据流方向和操作符之间的依赖关系。

关于计算图的基本优化主要有算子融合、内存变换、数据布局转换等。通过建立 DAG 图来处理高阶 IR 层次的优化，再计算图中的节点表示对静态已知维数的张量操作，边表示张量操作之间的数据流依赖关系。例如算子融合操作，将多个操作符融合在一起是一种优化，可以大大减少执行时间，特别是在 GPU 和专门的加速器。其思想是将多个操作符组合在一个内核中，而不是将中间结果保存在全局内存中，进而减少执行所需要的时间。

4　低阶 IR

低阶 IR 编译映射对应图 1 中的低阶 IR 模块，本章具体描述密码算子添加、实现以及映射的过程。为使密码算子成功在 VTA 架构进行部署，实现了密码算法相关的算子及微码操作。算子的实现基于 TIR 进行开发，允许开发人员重用大部分现有的编译接口。抽象出杂凑类算法各阶段的算子粒度，通过多级 IR 的编译优化，最终将其映射到硬件中。大量频繁调用的高层级算子进行组合以模块化，并将对应的模块映射到硬件电路中，用以解决算子适配、性能优化及灵活性等问题。密码算法通过重用很大一部分现有编译架构，让编译开发人员主要关注算子编译映射部分，大幅减少开发人员的工作量。

4.1　算子添加

算子添加是根据编译映射规则添加密码算法需要的不同粒度算子，并描述如何将该算子降低为 TIR 进行代码生成的过程。表 2 是新增原子级操作，表 3 是新增函数级操作，以支持 MD5 等各种常见的算法实现，目的是扩展张量 ALU 支持的算子范围，以扩展算子种类，提高资源利用率。

表 2　原子级算子

Table 2　Atomic-level operators

新算子	操作	说明
AND(X,Y)	$R[X]=R[X]\&R[Y]$	逻辑与运算
OR(X,Y)	$R[X]=R[X]\|R[Y]$	逻辑或运算
NOT(X,Y)	$R[X]=\sim R[X]$	逻辑取非运算
ROL(X,Y)	$R[X]=R[X]\ll\_C$	循环左移运算
XOR(X,Y)	$R[X]=R[X]\wedge R[Y]$	逻辑异或运算

表 3　函数级算子

Table 3　Function-Level Operators

新算子	操作	说明
FF	$FF(x,y,z)=(x\&y)\|(\sim x\&z)$	对应 MD5 中的 $F(b,c,d)$
GG	$GG(x,y,z)=((x\&z)\|(y\&\sim z))$	对应 MD5 中的 $G(b,c,d)$
HH	$HH(x,y,z)=(x\wedge y\wedge z)$	对应 MD5 中的 $H(b,c,d)$
II	$II(x,y,z)=(y\wedge(x\|\sim z))$	对应 MD5 中的 $I(b,c,d)$

4.2　TE 表达式构建

根据定义的不同粒度算子，通过张量表达式来定义计算，张量表达式是基于 Halide 的算子描述和调度优化分离的思想实现的。张量表达式被转换为一种树状的中间表示形式，主要包含两种类型的节点：表达式节点和语句节点。表达式类型的节点主要由下列几个成员构成，变量和常量、数组、函数调用、算数和逻辑表达式等；另一种类型的节点主要由顺序、循环、分支、赋值等控制逻辑构成。

图 3 实例中定义了一个规模为(7,8,1,16)的“循环左移”运算的张量表达式。其中 2),3)表示定义的 A,B 两个变量；4),5)表示申请的内存空间地址；6)表示张量表达式的运算，A_buf 循环左移 B_buf 位；7)将 A_buf 和 B_buf 运算的结果以张量的形式存放到 C_buf 中。

```
1) o=7;m=8
2) A=te.placeholder((o,m,batch,block),dtype=env.acc_dtype)
3) B=te.placeholder((o,m,batch,block),dtype=env.acc_dtype)
4) A_buf=te.compute((o,m,batch,block),lambda *i:A(*i),*A_buf)
5) B_buf=te.compute((o,m,batch,block),lambda *i:B(*i),*B_buf)
6)C_buf=te.com,blockpute((o,m,batch,block),lambda
*i:A_buf(*i)<<B_buf(*i),name="C_buf")
7)C=te.com,blockpute((o,m,batch,block),lambda
*i:C_buf(*i).astype(env.inp_dtype),name="C")
```

图 3　“循环左移”运算张量表达式构建

Fig. 3　Construction of tensor expression for circular left shift

4.3　循环展开

在图 3 中，变量 A，B 规模为(7,8,1,16)的四维矩阵，矩

阵 ***C_buf*** 由矩阵 ***A_buf*** 和矩阵 ***B_buf*** 计算得到。图 3 中算子被翻译为如图 4 所示的 IR，图 4 中的 4 个 stage 组成一个语法树，其中每个 stage 都是树中的一个节点，通过对 for 循环进行分析以及边界条件抽取来进行循环展开。首先对 ***A_buf***，***B_buf***，***C_buf*** 进行内存空间分配，在 Stage1，Stage2 中是将 ***A_buf***，***B_buf*** 转换为 3 层 for 循环表示，Stage3 中是循环左移运算，Stage4 是将 Stage3 的计算结果保存到***C_buf*** 中。

```
PrimFunc([A, B, C]) attrs={"global_symbol": "main", "tir.noalias": (bool)1} {
// attr [A_buf] storage_scope = "global"                    内存空间分配
allocate A_buf[int32 * 896]
// attr [B_buf] storage_scope = "global"
allocate B_buf[int32 * 896]
// attr [C_buf] storage_scope = "global"
allocate C_buf[int32 * 896]
 for (i0, 0, 8) {                                           Stage1
  for (i1, 0, 7) {
   for (i3, 0, 16) {
    A_buf[((((i0*112) + (i1*16)) + i3)] = A[((((i0*112) + (i1*16)) + i3)]
   }
  }
 }
 for (i0, 0, 8) {                                           Stage2
  for (i1, 0, 7) {
   for (i3, 0, 16) {
    B_buf[((((i0*112) + (i1*16)) + i3)] = B[((((i0*112) + (i1*16)) + i3)]
   }
  }
 }
 for (i0, 0, 8) {                                           Stage3
  for (i1, 0, 7) {
   for (i3, 0, 16) {
    C_buf[((((i0*112) + (i1*16)) + i3)] = tir.shift_left(A_buf[((((i0*112) + (i1*16)) + i3)], B_buf[((((i0*112) + (i1*16)) + i3)])
   }
  }
 }
 for (i0, 0, 8) {                                           Stage4
  for (i1, 0, 7) {
   for (i3, 0, 16) {
    C[((((i0*112) + (i1*16)) + i3)] = C_buf[((((i0*112) + (i1*16)) + i3)]
   }
  }
 }
}
"""
```

图 4 中间代码生成

Fig. 4 Intermediate code generation

5 内存管理

杂凑类密码算法是一种计算密集型和内存密集型的任务，为了更好地提升内存利用率及计算效率，通过内存优化及数据张量化方式产生硬件需要的目标码，来提升硬件计算效率。第 5.1 节将介绍 3 种内存优化方法。

5.1 内存优化

在 VTA 硬件架构中支持从“二元地址”到“四元地址”的内存表示。编译必须产生支持硬件指令集的目标码。本节主要描述了几种从“二元地址”到“四元地址”的内存优化方法。通过编译内存优化，产生硬件需要的目标码。

1）二元地址内存优化方法

图 5 中的数学表达式描述了两种张量计算规则。index0，index1 表示两个原索引地址内容，两个源地址内容运算完之后，将运算结果重新放回其中之一，这样可提高内存的使用效率。源数据和目的数据通过张量 ALU 运算把结果重新存放到目的地址中，目的是平衡资源利用率。因此张量和张量的操作是通过多个循环的向量和向量操作来执行的。但 VTA 的这种访存方式也限制了内存的利用率，导致 ALU 计算只适合做简单的运算。

本文密码算法都是简单的运算，并没有复杂的矩阵运算，因此使用该访存模式。当前编译优化规则不支持直接进行数据交换，导致密码算法在 ALU 上进行运算时，出现中间结果丢失，进而导致内存数据被篡改，运算结果出错的问题。在不进行内存优化之前，只能通过内存备份方式来达到数据交换的目的。

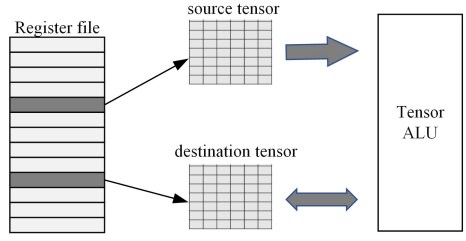

图 5 二元地址内存示意图

Fig. 5 Binary address memory diagram

2）三元地址内存优化方法

例如 MD5 算法，64 轮计算的中间结果都要进行数据交换完之后再传给下一轮使用。这导致设备端计算完的结果都需要把计算结果传回主机设备先进行数据交换，再把数据传回 VTA 设备进行下一轮的计算，如图 6 所示。

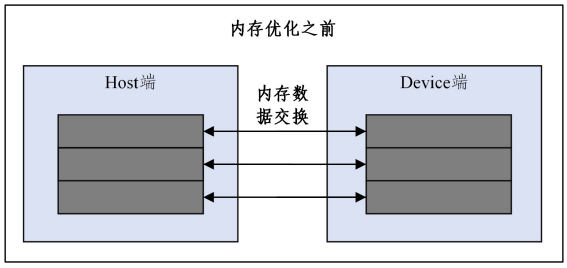

图 6 优化前的数据交换

Fig. 6 Data exchange before optimization

虽然通过这种方式解决了每轮数据复用过程中存的内存数据被篡改的问题，但主机端和设备端频繁的数据交换使得计算效率被降低，在整个算法的运算过程中，数据的交换占用了所有时间，数据参与运算的时间甚至可以忽略不计。

为解决内存数据被篡改的问题，每次的中间数据都需要备份多次。通过内存 buffer 的申请把原来的二元地址算法改为三元地址，保证了在计算过程中活跃的内存变量不会出现被覆盖的情况，如图 7 所示，其中数学表达式描述了 src0，src1 两个不同源地址内容运算完之后的结果存放到一个新的目标地址中。通过三元地址内存方式进行数据的访存，避免了内存覆盖的问题，而且减少了编程人员手动 buffer 备份，减少了内存的占用。这不仅降低了编程人员的工作负担，而且使得编译自动化。

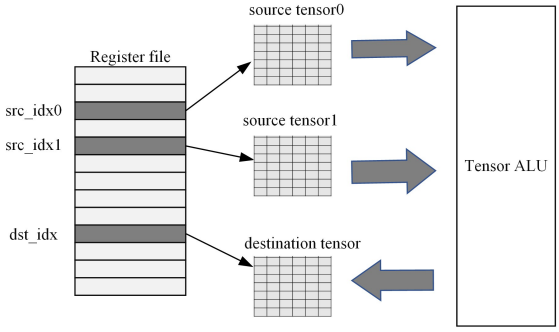

图 7　三元地址内存示意图

Fig. 7　Ternary address memory diagram

3）四元地址内存优化方法

函数级的算子都是 3 个输入 1 个输出，为四元的地址内存方式。为了更好地解决函数级算子输入的问题，把三元地址改为四元地址，在保证二元、三元地址内存方式的前提下，增加了四元地址内存访问方式。如图 8 所示的四元地址内存访问示意图，其中数学表达式描述了 src0，src1，src2 3 个不同源地址内容运算完之后的结果存放到一个新的目标地址 dst 中。通过四元地址内存方式进行数据的访存，达到与三元地址同样效果的优化。

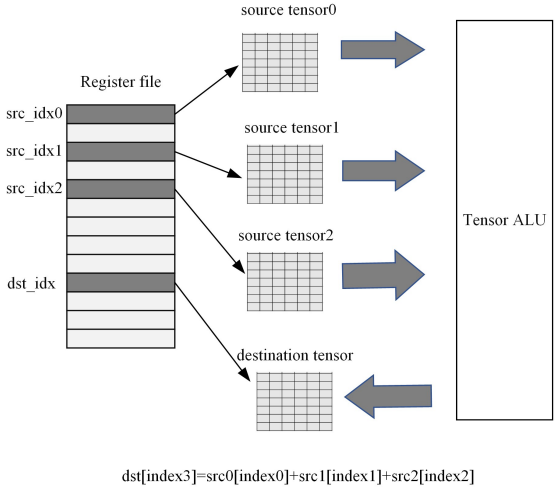

图 8　四元地址内存示意图

Fig. 8　Quaternary address memory diagram

通过内存 buffer 的申请把原来的二元地址算法改为三元地址、四元地址内存访问方式，保证了在计算过程中活跃的内存变量不会出现被覆盖的情况。为提高内存访问的效率，把计算的中间结果都存储在 VTA 设备中进行，以减少主机和设备之间的数据交互，提高计算效率。把计算以及中间数据的结果存储到 VTA 中可以提高计算效率。

如图 9 所示，以 MD5 算法为例，64 轮的计算全部放到设备端进行，这样就不需要把每轮的计算结果放到主机端交换之后再传回设备端，每轮的计算结果在设备端进行交换，只将最后结果传回主机端。这大大减少了主机端与设备端的数据交互，减少了数据的执行时间。

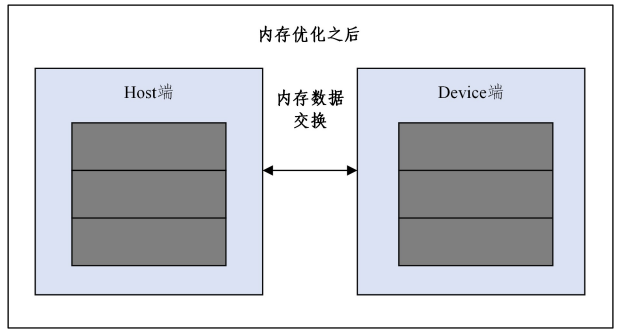

图 9　内存优化后的数据交换

Fig. 9　Data exchange after memory optimization

5.2　张量化数据映射

数据张量化是将低维数据映射为高维数据的过程。通过张量化得到的结果可以是二维矩阵、三维张量或者更高维张量。通过编译优化将程序映射到底层硬件的张量计算单元。张量化比向量化更具挑战性，张量化涉及多维数据、可变输入数据长度。

以 MD5 算法为例，图 10 展示张量化数据在重复使用情况下的数据映射过程。图 10 中 data0-data15 表示输入的 1 组报文，这组报文被分成 32 bit * 16 的数据，通过行存储方式进行存储访问，其中变量 i 表示地址索引，$i \in [1, 64]$。在 step1 中，当 $1 \leqslant i < 16$ 时，512 bit 报文按照顺序访问方式，进行数据的访存；在 step2 中，当 $16 \leqslant i < 32$ 时，512 bit 报文按照索引为 $(5 * i + 1) \% 16$ 的计算规则进行数据访问；在 step3 中，当 $32 \leqslant i < 48$ 时，512 bit 报文按照 $(3 * i + 5) \% 16$ 的方式进行数据访问；在 step4 中，当 $48 \leqslant i \leqslant 64$ 时，512 bit 报文按照 $(7 * i) \% 16$ 的方式进行数据访问；输入的张量指令被重复使用 4 次，这样可以减少所需的寄存器文件带宽，提高计算效率，减少硬件的执行时间。

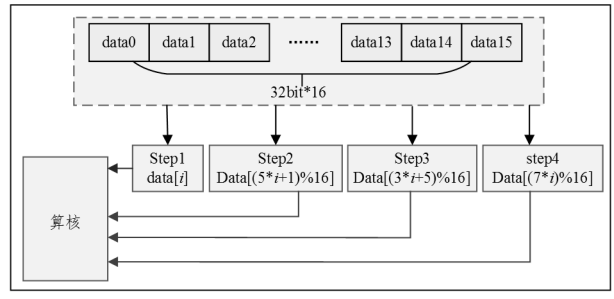

图 10　张量化数据重复使用

Fig. 10　Tensor data reuse

为进一步提高密码算法在 VTA 上的计算效率，减少硬件执行时间，图 11 展示张量化 16 的单指令多数据（Single Instruction Multiple Data，SIMD）内存访问方法。输入的报文是

16 组，每组报文被分成 32 bit * 16 数据，其中每行是输入的 1 组报文，即 32 bit * 16 的数据，列表示的是 16 组输入报文。例如，在 step1 中通过是行寻址方式来寻址，直到(32 bit * 16) * 16 的二维矩阵数据被访存完成。以这种方式实现 SIMD 的访存。SIMD 实现了在图 10 原有访存计算效率的基础之上提升 16 倍的计算效率。

在图 11 访存方式的基础之上，为进一步提高密码算法在 VTA 上的计算效率，减少硬件执行时间，图 12 展示张量化为单指令多线程(Single Instruction Multiple Thread，SIMT)实现方法。使用两个算核同时进行访存，输入的报文是(2 * 16)组，每组报文被分成 32 bit * 16 的数据。其中每个算核中的数据组织方式，每行表示输入 1 组报文，即 32 bit * 16 的数据，列表示 16 组输入报文。在两个维度上张变量化为 2 * (32 bit * 16) * 16，SIMT 在图 10 原有计算效率的基础上提升 32 倍的效率。

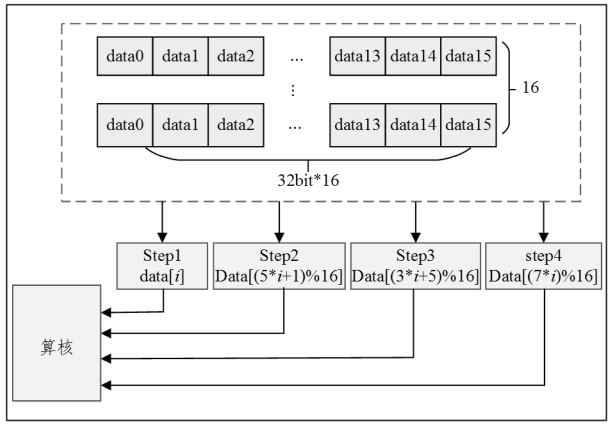

图 11 张量化为 16 单指令多线程

Fig. 11 Tensorization to 16 single instruction multiple threads

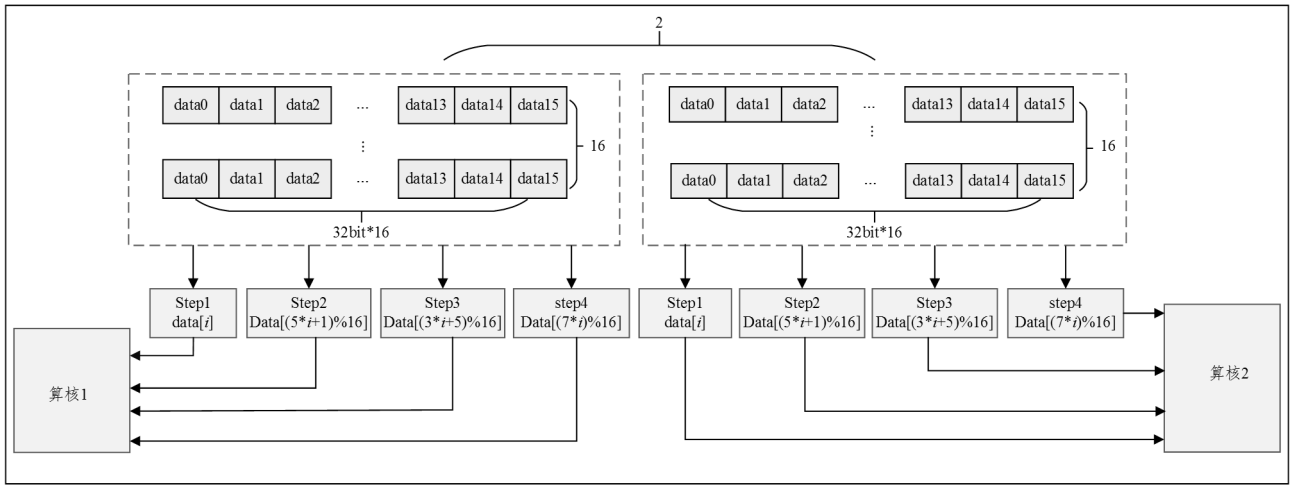

图 12 张量化为 2 * (32 bit * 16) * 16 单指令多线程

Fig. 12 Tensorization to 2 * (32 bit * 16) * 16 single instruction multiple threads

5.3 硬件指令参数生成

编译映射的过程，将 IR 降低为针对 VTA 硬件框架的低级实现。其主要思想是应用一组基本转换来增量地进行映射调度，以保持原始程序的逻辑等效性。在转换之后，降低的 VTA 调度直接调用 VTA 运行时，通过运行时编译加速器二进制文件。

本文以循环左移算子为例，把图 7 所示的 for 循环代码生成图 13 所示蓝色虚框中硬件所需的指令参数。

第一步：在 Step1 中，根据 Stage1，Stage2 中的 for 循环转换为 Step1 中所示微码操作。在 Step1 的绿色虚框中，其中 0，1 表示 A 和 B 两个源操作数的 SRAM 地址，3 表示目标 SRAM 地址。

第二步：在 Step2 中，根据 Stage1，Stage2 中的 for 循环变量索引 indices：[i0，i1，i3]，外层循环 extents：[7，8，16]，得到需要循环的次数，循环次数是由最外面两层循环 7 * 8 = 56 得到。

第三步：根据 loop_body. value. op. name = = "tir. loop_left_shift"得到对应的 alu_opcode 编码。

第四步：拿到循环左移算子中的左操作 A_buf[(((i0 * 112) + (i1 * 16)) + i3)]、右操作 B_buf[(((i0 * 112) + (i1 * 16)) + i3)]，目的地址 C_buf[(((i0 * 112) + (i1 * 16)) + i3)]。

第五步：通过第四步进行线性方程检测方式，得到 dst_coeff 目的地址中系数 dst_coeff = [112，16，1，0]，得到源地址中系数值 src_coeff = [112，16，1，0]，最外层循环变量 extents = [7，8，16]。

第六步：通过对循环系数的张量化，通过重写和规范化表达式方式，得到 src_coeff = [8，1，0]，dst_coeff = [8，1，0]，extents = [7，8]。

第七步：对 Step3 中最外面两层进行循环展开 extents = 7 * 8 = 56，取出第六步中 src_coeff 和 dst_ceff 列表中的后两个数，得到 src_coeff = [1，0] ，dst_coeff = [1，0]，dst_coeff [idx] = 1，src_coeff[idx] = 1 ，extents = [56]，由此得到 Step3 中的 uop_push 指令。

第八步：在 Step3 中，根据 Stage3 中的"tir. shift_left"导出操作码，在算子库中找到对应操作码的编码，其中 10 代表"循环左移"的微码操作，2 代表目的地址，其中第四位和第五位跟 Step1 中的绿色虚框含义一样。

第九步：在 Step4 中，dep_push(2，3)代表 uop 指令压栈的读写依赖关系，dep_pop(2，3)代表出栈的依赖关系。

第十步：在 Step5 中，表示存储指令，将计算的结果存储在 C 当中。其中 2 表示 SRAM 类型，4 表示目的数据在 DRAM 的地址。

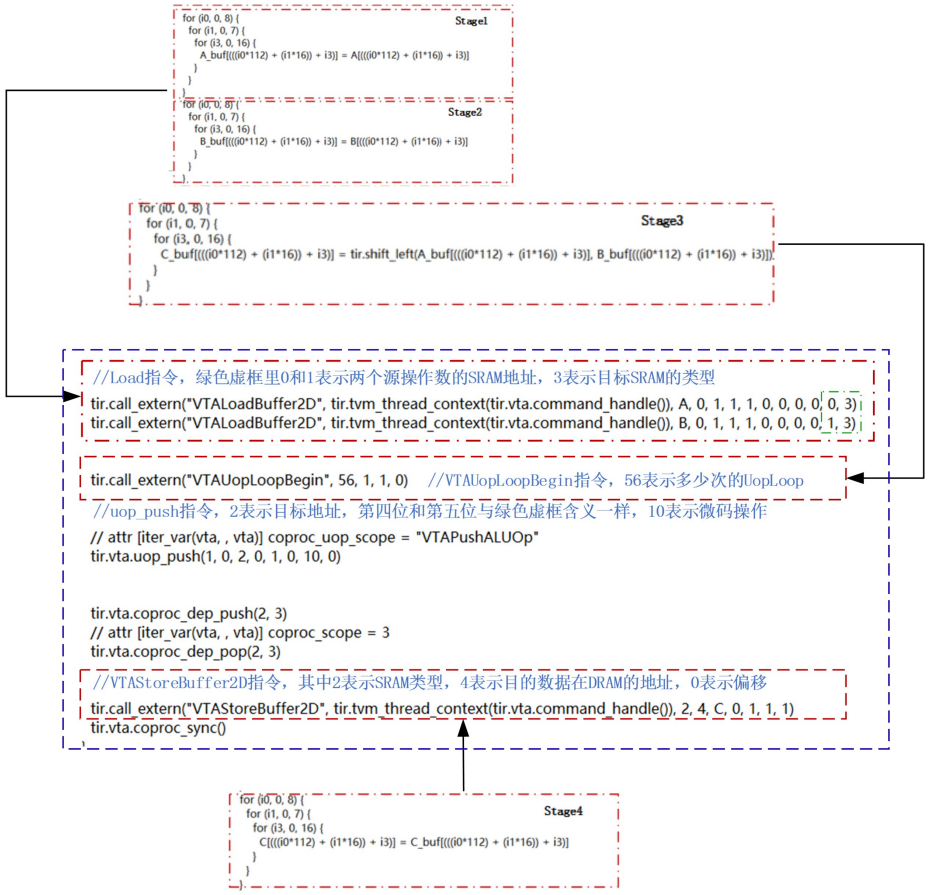

图 13　生成指令格式示意图

Fig. 13　Schematic diagram of the generated instruction format

6　实验结果

为验证高阶密码算子在 FPGA 的代码设计与实现方面的性能，我们使用 Xilinx Zynq-7000 可编程 SOC XC7Z045 FFG900-2 作为测试平台来验证我们的编译系统框架。XC7Z045 FFG900-2 包含一个由双核 ARM Cortex-A9 为核心构成的处理系统(Processing System，PS)和一个 FPGA 的可编程逻辑(Programmable Logic，PL)部分。以 Vitis HLS 2020.1 作为高层次综合工具，使用 SDAccel 提供系统平台集成并作为 OpenCL 工具链在 FPGA 上综合 OpenCL 内核，时钟频率为 100 MHz。OpenCL 是一个为不同计算设备组成的异构集群进行编程的工业标准，本文以 OpenCL 为实验的参照对编译映射张量化、寄存器优化以及执行效率进行评估。

6.1　实验设置及测试基准

以 MD5 算法为例，在 Zynq-7000 芯片中分别采用 OpenCL 和基于多级编译优化框架在 VTA 架构上进行实验对比。在第 5 章中讲到内存优化和张量化数据映射，本节以不同粒度算子为基准，进行性能的比较。

5.2 节介绍了密码算法张量化数据重复使用、单指令多数据 SIMD、单指令多线程 SIMT 实现方法。因此，我们在实验中对这 3 种情况展开测试。以 MD5 为例，在二元地址情况下，对密码算法张量化数据重复使用、单指令多数据、单指令多线程实现方法进行性能测试。以 OpenCL 为实验参考基准，以纳秒为单位，在本文的硬件平台环境下 OpenCL 测试的 MD5 运算时间为 177.49ns。

6.2　不同粒度算子的 MD5 性能测试

性能测试 1：根据图 10 张量化数据重复使用的优化方法，根据 3 种不同粒度算子编码 MD5 算法并进行性能比较。从表 4 中可以看出原子级算子、函数级算子分别在二元地址、三元地址下的执行效率都比 OpenCL 的差。在四元地址下算法级算子运算时间为 163 ns，相比 OpenCL 运行效率提升不明显。但张量化数据重复使用下不同粗粒度算子的 MD5 灵活度高，开发人员可以灵活定义算子计算规则。根据表 4 可知，以二元地址下原子级算子构建的 MD5 算法为基准，二元地址算法级算子构建的 MD5 算法的执行时间是二元地址下原子级算子执行时间的 10.4%。四元地址下算法级算子构建的 MD5 算法执行时间是二元地址下原子级算子执行时间的 10.8%，大幅提高了算法在 FPGA 上的执行效率。

表 4　数据重复使用下不同粒度算子的执行时间

Table 4　Execution time of different granularity operators under data reuse

(ns)

MD5 算子	二元地址	三元地址	四元地址
原子级算子	1902	1715	1496
函数级算子	1586	1438	1269
算法级算子	198	181	162

在张量化数据重复使用的情况下，为便于直观展示不同粒度以及不同寄存器优化方法下算法的执行效率，我们采用 3 种不同粒度的算子通过领域专用语言来快速构建 MD5 算法在 FPGA 上部署。在图 14 中可以直观地看到“算法级”的硬件执行时间比“原子级”和“函数级”构建的算法执行时间短得多。

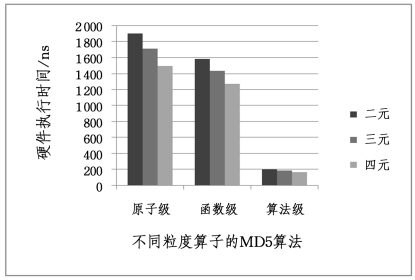

图 14 数据复用情况下 MD5 算法在硬件执行时间

Fig. 14 Hardware execution time of the MD5 algorithm under data reuse

性能测试 2:为进一步提高密码算法在 VTA 上的计算效率,减少硬件执行时间,根据图 11 展示的张量化为 16 的单指令多数据的实现方法,以不同的寄存器优化方法对原子级算子、函数级算子、算法级算子从 3 个不同粒度对 MD5 算法进行时间的比较。首先以 OpenCL 在本平台上的运算时间 177.49 ns 作为基准进行运算时间的比较,在二元地址的原子级算子情况下的运算时间是 118.75 ns,相比 OpenCL 算法执行时间减少至 66.9%。在四元地址的函数级算子情况下运算时间是 11.23 ns,相比 OpenCL 算法执行时间减少至 6.3%。

表 5 SIMD 下不同粒度算子执行时间

Table 5 Execution time of different granularity operators under SIMD

(ns)

MD5 算子	二元地址	三元地址	四元地址
原子级算子	118.75	108.58	93.12
函数级算子	98.75	89.86	74.23
算法级算子	18.86	12.12	11.23

在张量化为 16 的单指令多数据 SIMD 下,为便于展示不同粒度以及不同寄存器优化方法下的算法执行效率,在图 15 中可以直观地看出,同一粒度算子不同地址索引下构建的 MD5 算法,四元地址情况下“算法级”算子构建的 MD5 执行效率最高。在“单指令多数据”的方式下相比“张量化数据重复使用”的情况下在原子级、函数级、算法级粒度算子构建的 MD5 算法执行效率都提升 16 倍左右。

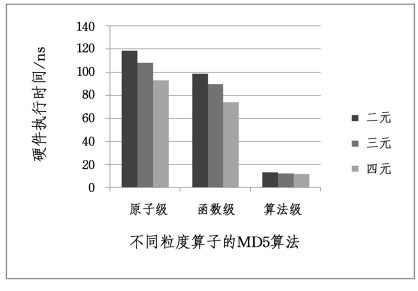

图 15 SIMD 下 MD5 的硬件执行时间

Fig. 15 Hardware execution time of MD5 under SIMD

性能测试 3:为进一步提高密码算法在 VTA 上的计算效率,减少硬件执行时间,图 14 展示张量化为 2 * (32 bit * 16 * 16)的单指令多线程实现方法。在 SIMT 的寄存器优化方法对原子级算子、函数级算子、算法级算子从 3 个不同粒度的 MD5 算法进行比较。首先以 OpenCL 在本平台上的运算时间 177.49 ns 作为基准,在二元地址的原子级算子情况下的运算时间是 58.96 ns,相比 OpenCL 算法执行时间减少至 33.2%。在四元地址的函数级算子情况下运算时间是 5.12 ns,相比 OpenCL 算法执行时间减少至 2.89%。

表 6 SIMT 下不同粒度算子的执行时间

Table 6 Execution time of different granularity operators under SIMT

(ns)

MD5 算子	二元地址	三元地址	四元地址
原子级算子	58.96	52.12	46.78
函数级算子	49.24	44.54	39.58
算法级算子	6.89	5.65	5.12

在张量化为 2 * (32 bit * 16 * 16)的 SIMT 下,为便于展示不同粒度以及不同寄存器优化方法下算法的执行效率,在图 16 中直观地可以看出,以同一粒度算子为参考,不同的内存优化方式在四元地址情况下效率都是最高的。以同一地址索引为参考,不同粒度的算子,粒度越大效率越高。在单指令多线程情况下相比单指令多数据的方式在原子级、函数级、算法级粒度算子构建的 MD5 算法执行效率都提升 16 倍左右。相比张量化数据重复使用的方式在原子级、函数级、算法级粒度算子构建的 MD5 算法执行效率都提升 32 倍左右。

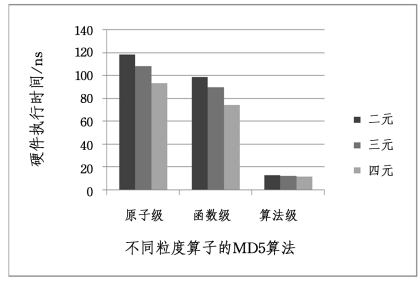

图 16 SIMT 下 MD5 的硬件执行时间

Fig. 16 Hardware execution time of MD5 under SIMT

由上面 3 组实验可以看出,以同一粒度算子为参考,不同的内存优化方式在不同粒度的算子下“四元地址”情况下效率都是最高的。以同一地址索引为参考,不同粒度的算子,粒度越大效率越高。这也充分证明在张量化 SIMT 下,把整个算法作为一个算子映射到硬件上,减少了内存的读写以及数据的交互,从而提升计算的效率。

结束语 为解决杂凑类密码算法在 FPGA 上快速高效的部署问题,本文设计实现了一个面向高阶密码算子在 FPGA 的代码设计与实现方法,系统采用端到端编译模式,基于高阶密码算子的实现,并在系统中利用多级中间表示进行算法依赖分析、调度变换、内存管理以及代码生成模块[22]。系统将 pthon 语言程序作为输入,抽象为多层中间表示形式,通过多级 IR 映射、内存管理以及代码生成模块,最终生成面向 VTA 架构的密码程序。最后,利用不同粒度算子编写的 MD5 算法进行测试,结果显示系统生成的不同粒度的算子编码的 MD5 执行效率相比 OpenCL 提升了 34 倍,且不同粒度算子也是为了提高算子的灵活性,用户可以根据支持的不同粒度算子任意定义计算规则。同时也给程序开发带来了便利性,可以更方便地根据用户的计算需求进行不同算法的开发,减少编程人员的工作量及后期维护成本。

目前我们在多级编译系统实现简单的不同粒度算子的密码算法的硬件映射策略,为了更好地验证本文系统的有效性,

我们下一步的工作将进一步的完善映射策略实现。本文以 MD5 为例做了相关实验的研究,但是基于杂凑类算法除了 MD5 以外还有 SHA1,SHA2 等相关算法都还没有做相关的验证,下一步将完善杂凑类算法的研究与实现。

参 考 文 献

[1] AMIR A,DATTA P,RISK W P,et al. Cognitive computing programming paradigm:A corelet language for composing networks of neurosynaptic cores[C]// International Joint Conference on Neural Networks(IJCNN). 2013:1-10.

[2] DAVISON A P,BRUDERLE D,EPPLER J,et al. PyNN:A Common Interface for Neuronal Network Simulators[J/OL]. Frontiers in Neuroinformatics, 2008: 2. http://doi. org/10. 3389/neuro. 11. 011. 2008.

[3] CHEN T Q,MOREAU T,JIANG Z H,et al. TVM:An automated end-to-end optimizing compiler for deep learning[C]// 13th Symposium on Operating Systems Design and Implementation. 2018:578-594.

[4] VASILACHE N,ZINENKO O,THEODORIDIS T,et al. Tensor comprehensions:Framework-agnostic high-performance machine learning abstractions[J]. arXiv:1802. 04730,2018.

[5] ROTEM N,FIX J,ABDULRASOOLl S,et al. Glow:Graph Lowering Compiler Techniques for Neural Networks[J]. arXiv:1805. 00907,2018.

[6] CYPHERS S,BANSAL A K,BHIWANDIWALLA A,et al. Intel ngraph:An intermediate representation,compiler,and executor for deep learning[J]. arXiv:1801. 08058,2018.

[7] ALEXANDER M,THIEN N. A MLIR Dialect for Quantum Assembly Languages[C]// 2021 IEEE International Conference on Quantum Computing and Engineering(QCE). 2021:255-264.

[8] LATTNER C,ADVE V. LLVM:A compilation framework for lifelong program analysis & transformation[C]// Proceedings of the international symposium on Code generation and optimization:feedback-directed and runtime optimization. 2004:75-86.

[9] CHEN C,LI K L,OUYANG A,et al. FlinkCL:AN OpenCL-based in memory computing architecture on heterogeneous CPU-GPU clusters for big date[J]. IEEE Transactions on Computers,2018,67(12):1765-1779.

[10] ASHOURI A H,KILLIAN W,CAVAZOS J,et al. A Survey on Compiler Autotuning using Machine Learning[J]. ACM Computing Surveys,2018,51(5):1-42.

[11] ROESCH J,LYUBOMIRSKY S,KIRISAME M,et al. Relay:A High-Level Compiler for Deep Learning[J]. arXiv:1904. 08368, 2019.

[12] ZHENG B W. Hash Algorithm Accelerator Based on FPGA [D]. Wuxi:Jiangnan University,2023.

[13] ZHENG S,LIANG Y,WANG S,et al. FlexTensor:An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System[C]// Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS'20). 2020:859-873.

[14] ZHAO J,DI P. Optimizing the Memory Hierarchy by Compositing Automatic Transformations on Computations and Data [C]// 53rd Annual IEEE/ACM International Symposium on Microarchitecture(MICRO) 2020. 2020:427-441.

[15] LI L D. Parallel Optimization of Data Intensive Computing on Sunway TaihuLight[D]. Beijing:Tsinghua University,2021.

[16] WANG X Y,YU H B. A review of cryptographic hash algorithms[J]. Information Security Research,2015,1(1):19-30.

[17] LI Y B,ZHAO R C,HAN L,et al. Parallelizing compliation framework for heterogeneous manycore processors[J]. Journal of Software,2019,30(4):98101001.

[18] CONG J,LIU B,NEUENDORFFER S,et al. High-level synthesis for FPGAs: From prototyping to deployment [J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,2011,30(4):473-491.

[19] EJJEH A,ADVE V S,RUTENBAR R A. Studying the Potential of Automatic Optimizations in the Intel FPGA SDK for OpenCL[C]// The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays(FPGA'20). 2020.

[20] MOREAU T,CHEN T,JIANG Z,et al. VTA:An Open Hardware-Software Stack for Deep Learning[J]. arXiv. 1807. 04188, 2018.

[21] LIN S Y. Optimized Fully-pipelined Architecture of SHA-2 and MD5 on FPGA[D]. Xiamen:Xiamen University,2020.

[22] TAO X H,ZHU Y,PANG J M,et al. Parallel Cod-e Generation for Sunway Heterogeneous Architecture[J]. Journal of Software,2023,34(4):1570-1593.

PEI Xue, born in 1992, master, assistant researcher. Her main research interest is software and hardware co-compilation.