工科创 II – 领域专用加速器的设计和优化

Final Project: 部署一个小型神经网络

组员：N/A

摘要

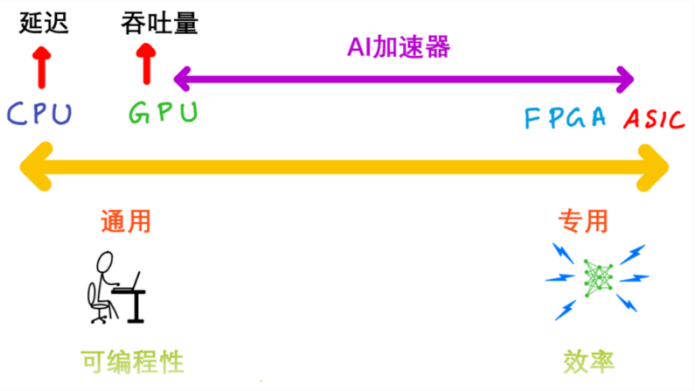
本项目旨在设计并实现一个硬件加速的矩阵乘法模块，并将其集成到 RISC-V 处理器中。我们使用 Chisel 硬件描述语言编写了带有状态机的矩阵乘法加速器模块，以支持任意大小的矩阵乘法。随后通过 RoCC（Rocket Custom Co-processor）接口，使用自定义 RISC-V 指令与矩阵乘法加速器通信，控制加速器进行矩阵乘法运算，RISC-V 指令可以通过 C 代码内联汇编调用。最后我们将一个原用 Python 实现的多层感知器（MLP）网络改写成 C 代码，并调用硬件加速的矩阵乘法模块进行前向传播计算。实验结果表明，硬件加速的矩阵乘法器返回了正确结果，且成功提升了 MLP 网络的计算效率。此次实验展示了硬件加速在复杂计算任务中的应用潜力和性能优势。

目录

引言

1.1 背景介绍

随着人工智能和机器学习的快速发展，矩阵运算在现代计算中扮演了越来越重要的角色。尤其在神经网络训练和推理过程中，大量的矩阵乘法运算对计算资源提出了极高的要求。传统的通用处理器在执行这些密集运算时往往难以达到所需的性能，因此硬件加速器成为解决这一问题的重要手段。硬件加速器能够通过并行计算和专用硬件逻辑显著提升矩阵运算的效率，减少计算时间和能耗。



1.2 相关工作

近年来，硬件加速器的研究和应用取得了显著进展。谷歌的 TPU（Tensor Processing Unit）和英伟达的 CUDA GPU 是两个典型的成功案例，这些专用硬件在深度学习训练和推理中展示了卓越的性能表现。此外，FPGA（Field-Programmable Gate Array）也被广泛应用于定制化的矩阵运算加速。与这些专用硬件相比，基于 RISC-V 指令集架构的处理器由于其开放性和可扩展性，成为学术界和工业界关注的热点。

1.3 实验目的

本项目的主要目的是设计并实现一个基于 Chisel 和 RISC-V 的硬件加速矩阵乘法器，并将其应用于多层感知器（MLP）网络中。具体而言，我们的目标包括：

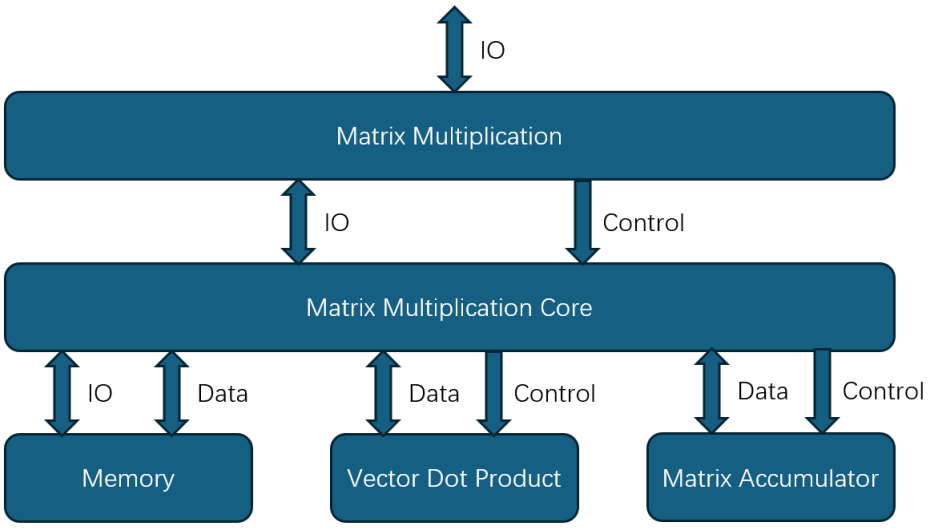
- 使用 Chisel 硬件描述语言设计并实现一个带有状态机的矩阵乘法加速器模块，支持任意大小的矩阵乘法运算。
- 利用 RoCC（Rocket Custom Co-processor）接口，将自定义的 RISC-V 指令与矩阵乘法加速器模块对接，实现指令级的控制和数据传输。
- 使用 C 语言编写矩阵乘法函数，通过内联汇编调用自定义的 RISC-V 指令，以验证硬件加速器的性能和正确性。
- 将原用 Python 编写的多层感知器（MLP）网络改写成 C 代码，并调用硬件加速的矩阵乘法模块进行计算，评估其在实际应用中的效果。
- 通过对比实验，测试硬件加速器的性能优势，并分析其在复杂计算任务中的潜力和应用前景。

通过以上实验步骤，我们期望能够展示硬件加速在提高计算效率方面的显著效果，并为进一步的研究和应用提供参考。

加速器模块的设计与实现

2.1 加速器模块的基本组成

加速器模块使用 Chisel 硬件描述语言编写，编译成 Verilog HDL 代码后映射到 FPGA 的布局布线。加速器模块由 Memory、Vector Dot Product、Matrix Accumulator、Matrix Multiplication Core 和 Matrix Multiplication 五个模块组成。关系如下图所示：



以下简略地描述各模块的功能特点。

2.2 Memory 模块

Memory 模块是使用 Chisel 的 SyncReadMem 实现的通用的存储器模块，它提供了对内存进行读写的功能，负责存储矩阵计算的输入数据和最终结果。主要特点如下：

- 支持字节(8 bits)、半字(16 bits)和字(32 bits)三种数据访问粒度。
- 写操作时根据不同的数据粒度，将数据分段写入对应的内存地址。
- 读操作时根据不同的数据粒度，从对应的内存地址读取数据并进行拼接。

由于本次测试中矩阵输入数据类型为 8 bits 整数，输出数据类型为 32 bits 整数，在 Memory 模块中支持了多种数据访问粒度的读写。这样可以防止存储输入数据时发生内存浪费，提高存储空间的使用效率。

2.3 Vector Dot Product 模块

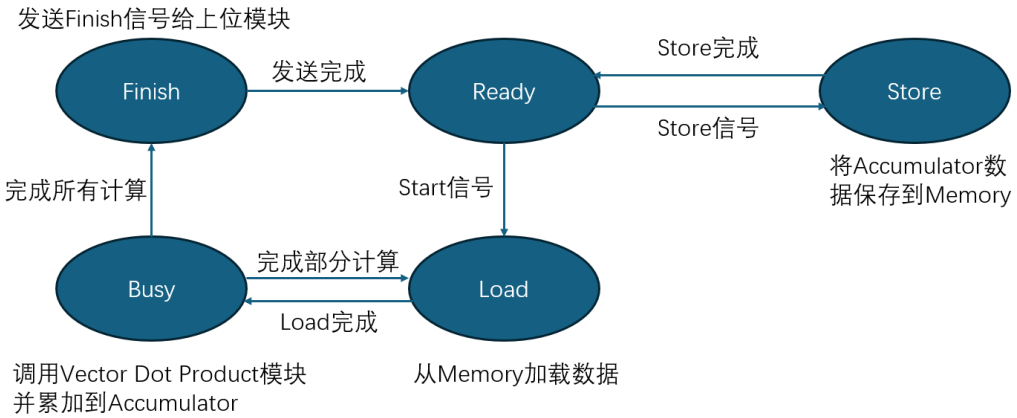
Vector Dot Product 模块实现了两个向量的点积计算功能，两个输入向量可以独立设置输入数据是否是有符号数。对于每个向量元素，先根据是否有符号进行数据类型转换，统一转换成有符号数，然后进行乘法运算，将所有乘法运算的结果进行累加，得到最终的点积结果。

2.4 Matrix Accumulator 模块

Matrix Accumulator 模块实现了一个矩阵累加器的功能，内部使用一个二维寄存器数组来存储矩阵元素的累加结果。当写使能有效时，根据行列坐标将输入数据累加到对应的元素上。当收到清零信号时，将 accumulator 中所有元素清零。另外还提供了矩阵元素的输出接口，可以根据行列坐标读取对应元素的累加结果。

2.5 Matrix Multiplication Core 模块

Matrix Multiplication Core 模块是一个大小固定的矩阵乘法器。由以上三个模块构成，并设计了状态机逻辑。状态转移如下：



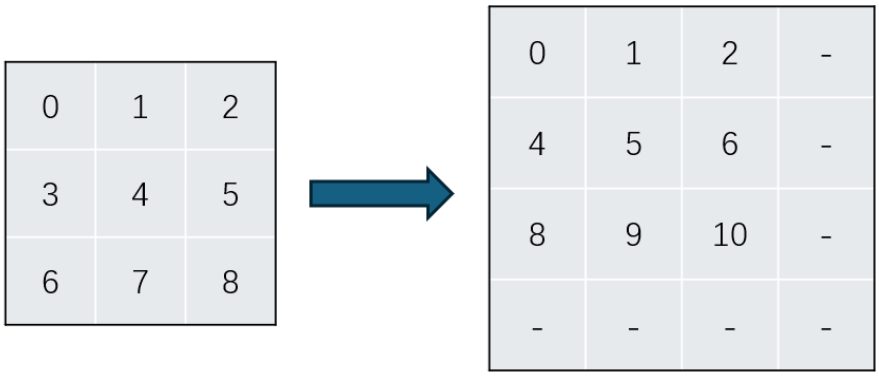
- Core 模块初始位于 Ready 状态
- 当模块接收到 start 信号时，初始化行列索引，开始遍历矩阵计算乘积。
- 模块首先根据当前的行列索引，通过内存模块读取对应的矩阵 A 的行向量和矩阵 B 的列向量，分别存入行缓冲区和列缓冲区，并转移到 Busy 状态。
- 在 Busy 状态时，模块调用向量点积模块计算行缓冲区和列缓冲区的点积，得到一个 32 位的中间结果。这个中间结果被送入矩阵累加器模块，根据当前的行列索引，累加到矩阵 C 对应的位置上。完成后，更新行列索引，如果还没有遍历完整个矩阵，就回到 Load 状态加载相应数据，继续计算下一个元素。一旦所有元素计算完成，状态机就会进入 Finish 状态。
- 在 Finish 状态时，Core 模块发送 finish 信号给上位模块。
- 另外，如果接收到 store 信号，状态机就会进入 Store 状态，将矩阵累加器模块的累加值写回到 memory 存储中。之后累加器中的数据会清零。

整体来看，该模块收到 start 信号时将相应矩阵相乘，并把结果累加到累加器中；收到 store 信号时将累加器中结果存入内存相应位置。Core 模块只支持固定大小的矩阵乘法计算，因此需要使用累加器模块，在上位模块进行分块乘法计算时保留中间累加结果。

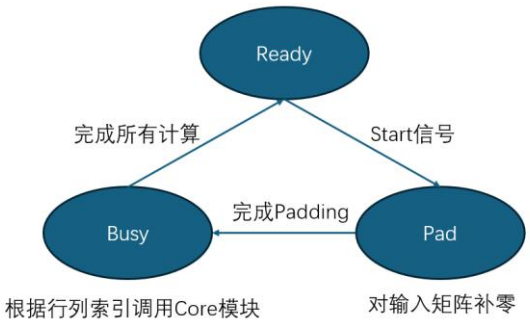
由于输入矩阵在内存中不是连续排列的，而是每行之间相隔一个偏移量，模块需要两个输入矩阵和输出矩阵的基地址和行间偏移量。模块在与 memory 模块传递数据时，读入数据格式是 8 bits 整数，而写回数据是 32 bits 整数，因此每个矩阵元素之间的地址差值也不尽相同。

2.6 Matrix Multiplication 模块

Matrix Multiplication 模块需要接收任意大小矩阵的乘法计算，因此要将矩阵分块。为了将矩阵变成 Core size 的整数倍，需要对输入的矩阵进行补零操作。由于存在补零操作，在输入输出时的矩阵元素地址会发生改变。因此对地址需要做译码操作，以 Core size = 4 为例，各元素的地址偏移量为：



该模块也设计了一个简单的状态机，设计如下：



在计算时，由于 Core Module 接收三个矩阵基地址的输入，因此只需要在遍历分块矩阵时修改基地址即可。Core Module 在计算时，该模块在 Busy 状态等待 finish 信号，计算结束后按照更新的索引计算下一个分块矩阵乘积。当计算完一组分块矩阵乘积时，发送 store 信号给 Core Module 将结果存入内存，并更新分块的行列索引。

模块向外提供内存的读写访问，读取默认为 32bits，写入默认为 8bits，经过译码的地址和数据通过 Core Module 最终转发给 Memory Module。实现外部 IO 直接读写内存。

硬件软件接口的设计实现

3.1 矩阵加速器模块的接口

上述加速器的设计中，输入输出接口主要包括：输入输出数据及其地址、控制信号以及输入矩阵的大小。其中前两者已经在 RoCC Shell 模块中实现，通过在指令中传递数组地址、片上内存地址和数组大小，驱动 DMA 模块搬运数据。为了在 C 代码和加速器模块之间传递数据矩阵大小这一参数，需要新设计一条指令。在 RoCC Shell 模块中译码后，通过寄存器传递给加速器模块。

3.2 RISC-V 指令设计与译码

在自定义指令中，共有 64 位的 rs1 和 32 位的 rs2 可用，因此可以将矩阵大小的三个参数以 32 位整数传递。为了判断指令的类型，设置指令的 funct 字段为与 load、store、compute 不同的数值即可。

```
void mat_info(uint32_t N, uint32_t K, uint32_t M)
{
    uint64_t rs1 = 0;
    rs1          = N;
    rs1          = (rs1 << 32) + K;

    uint32_t rs2 = 0;
    rs2          = M;

    ROCC_INSTRUCTION_SS(ROCC_OPCODE, rs1, rs2, OP_INFO);
}
```

在 chisel 代码中对指令进行译码，通过比较指令的 funct 字段判断指令类型，并将数据从 rs1 和 rs2 中提取出来送入加速器模块中即可。

```
when(io.cmd.fire && io.cmd.bits.inst.funct === cfg.infoISA) {
  gemm_matARows := io.cmd.bits.rs1(63, 32)
  gemm_matACols := io.cmd.bits.rs1(31, 0)
  gemm_matBRows := io.cmd.bits.rs1(31, 0)
  gemm_matBCols := io.cmd.bits.rs2(31, 0)
}
```

3.3 在 C 代码中调用硬件

调用原有的 ISA 中内联汇编以及新增的 mat_info 函数，可以设计硬件乘法的调用如下：

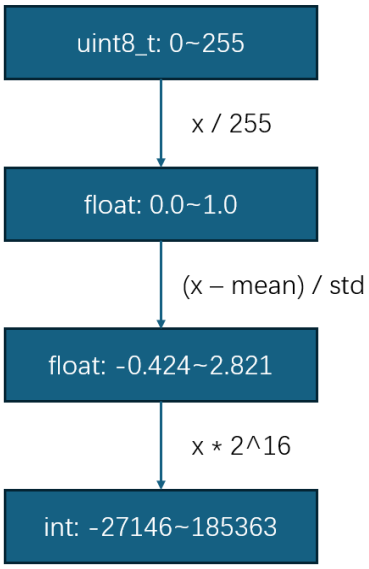
- 传递矩阵大小参数给加速器
- 将两个矩阵的数据基地址和片上地址传递给 RoCC，之后由 DMA 模块完成数据从 C 到硬件的搬运
- 传递 start 信号
- 用同样方法将片上数据的结果传递回结果数组地址

MLP 网络的实现

4.1 Python 代码分析

Python 代码主要完成了数据集的读取和处理，首先按照 MNIST 数据集格式解析数据，然后以尽量不损失精度的方式对数据进行 transform，最后调用 C 语言进行 MLP 的执行。最后将结果和标准答案比较，给出网络的准确度。

数据的 transform 方式保证了损失精度最小，具体实现如下：



4.2 数据集的生成

由于在 FPGA 中不支持文件的读写，需要将数据集以明文形式编译进 C 代码中。在标准 MNIST 数据集格式中，对于 image 文件，前四个大端序 4-byte 整数分别为文件标识符、图片数量、图片像素行数、图片像素列数，接下来为单字节组成的灰度像素串；对于 label 文件，前两个大端序 4-byte 整数为文件标识符和图片数量，接下来为单字节组成的 label 串，也即数字识别的答案。

为了减少片上数据 transform 的开销，根据以上格式直接将 transform 后的数据以明文形式硬编码写入 C 文件源码中，与其余部分一起编译。

4.3 C 代码的编写

由于数据集直接存储了 transform 后的结果，C 代码只需要调用 run_mlp 函数即可。设计主函数运行 10 个图片的 MLP 网络，并输出比较结果和答案。

为了调用硬件加速器，需要替换 nn_math 中的 mat_mul 函数。需要注意的是，C 代码中的数据以 int8_t 数组存储，而 DMA 读取时以 long 格式读取，因此需要使用 buffer 将数据拷贝到缓冲区中，计算完后再拷贝回结果。其余部分依照 3.3 节实现即可。

FPGA 实验结果

在 FPGA 上测试了两组数据结果如下：

```
RISC-V 64, Boot ROM V3.8
Sample 0: Predicted class: 7, Actual class: 7
Sample 1: Predicted class: 2, Actual class: 2
Sample 2: Predicted class: 1, Actual class: 1
Sample 3: Predicted class: 0, Actual class: 0
Sample 4: Predicted class: 4, Actual class: 4
Sample 5: Predicted class: 1, Actual class: 1
Sample 6: Predicted class: 4, Actual class: 4
Sample 7: Predicted class: 9, Actual class: 9
Sample 8: Predicted class: 6, Actual class: 5
Sample 9: Predicted class: 9, Actual class: 9
Correct predictions: 9/10
```

```
RISC-V 64, Boot ROM V3.8
Sample 1234: Predicted class: 8, Actual class: 8
Sample 1235: Predicted class: 5, Actual class: 5
Sample 1236: Predicted class: 1, Actual class: 1
Sample 1237: Predicted class: 2, Actual class: 2
Sample 1238: Predicted class: 1, Actual class: 1
Sample 1239: Predicted class: 3, Actual class: 3
Sample 1240: Predicted class: 1, Actual class: 1
Sample 1241: Predicted class: 7, Actual class: 7
Sample 1242: Predicted class: 9, Actual class: 4
Sample 1243: Predicted class: 5, Actual class: 5
Correct predictions: 9/10
```

与本地测试结果相符：

```
Loaded 10000 samples.
Sample 0: Predicted class: 7, Actual class: 7
Sample 1: Predicted class: 2, Actual class: 2
Sample 2: Predicted class: 1, Actual class: 1
Sample 3: Predicted class: 0, Actual class: 0
Sample 4: Predicted class: 4, Actual class: 4
Sample 5: Predicted class: 1, Actual class: 1
Sample 6: Predicted class: 4, Actual class: 4
Sample 7: Predicted class: 9, Actual class: 9
Sample 8: Predicted class: 6, Actual class: 5
Sample 9: Predicted class: 9, Actual class: 9
Accuracy: 90.00%
```

```
Loaded 10000 samples.
Sample 1234: Predicted class: 8, Actual class: 8
Sample 1235: Predicted class: 5, Actual class: 5
Sample 1236: Predicted class: 1, Actual class: 1
Sample 1237: Predicted class: 2, Actual class: 2
Sample 1238: Predicted class: 1, Actual class: 1
Sample 1239: Predicted class: 3, Actual class: 3
Sample 1240: Predicted class: 1, Actual class: 1
Sample 1241: Predicted class: 7, Actual class: 7
Sample 1242: Predicted class: 9, Actual class: 4
Sample 1243: Predicted class: 5, Actual class: 5
Accuracy: 90.00%
```

总结与反思

本项目的主要目标是设计并实现一个基于 Chisel 和 RISC-V 架构的硬件加速矩阵乘法器，并将其应用于多层感知器（MLP）网络的推理计算中。通过本次实验，我们成功完成了以下几个步骤：

- 硬件设计与实现：使用 Chisel 硬件描述语言设计了一个带有状态机的矩阵乘法加速器模块，该模块支持任意大小的矩阵乘法运算。设计了 Memory 模块、Vector Dot Product 模块、Matrix Accumulator 模块、Matrix Multiplication Core 模块，以及 Matrix Multiplication 模块，并通过状态机进行协调控制。
- 硬件与软件接口设计：利用 RoCC 接口，将自定义的 RISC-V 指令与矩阵乘法加速器模块对接，实现了指令级的控制和数据传输。设计了自定义 RISC-V 指令，并在 C 代码中通过内联汇编调用这些指令，以验证硬件加速器的性能和正确性。
- MLP 网络的实现与测试：将原用 Python 编写的 MLP 网络改写成 C 代码，使用硬件加速的矩阵乘法模块进行前向传播计算。测试结果表明，硬件加速的矩阵乘法器返回了正确结果，并显著提升了 MLP 网络的计算效率。
- 实验验证：在 FPGA 上进行了仿真和实际测试，通过对比实验验证了硬件加速的效果，结果与本地测试结果一致，进一步确认了硬件设计的正确性和性能优势。

在本次项目中，我们取得了一些重要的成果，但也遇到了一些挑战和需要改进的地方：

- 设计与实现的复杂性：硬件设计的复杂性较高，在实际 Vivado 综合布线中花费了大量时间调试。未来可以进一步优化硬件描述语言的硬件友好性，以提高设计效率和可维护性。
- 性能优化：虽然硬件加速器显著提升了矩阵乘法的计算效率，但在硬件加速器和软件调用之间，数据传输速度仍然成为瓶颈。可以考虑进一步优化数据传输路径，或者引入更高效的存储管理策略。
- 扩展性和通用性：当前的设计主要针对特定大小的矩阵乘法运算，虽然支持任意大小的矩阵，但需要进行补零操作。未来可以考虑设计更通用的矩阵处理模块，减少补零操作带来的额外开销。另外也可以引入设计浮点计算单元，从而支持更多网络的计算。
- 实验验证与测试：本次实验验证主要集中在 MLP 网络的推理计算中，未来可以扩展到更多类型的神经网络和其他复杂计算任务，进一步验证硬件加速器的通用性和性能优势。

项目分工：

- N/A：代码实现、上板测试和报告撰写
- N/A：RoCC Shell 和 Python 代码的理解