中国科学院大学
《智能计算系统》实验报告

一、 实验内容

本实验旨在掌握深度学习处理器中矩阵运算器的设计原理，通过使用 Verilog 硬件描述语言实现内积运算器、矩阵乘向量运算器、矩阵乘法运算器的设计，并通过 Verilator 仿真工具进行功能验证。深度学习处理器（DLP）采用矩阵运算作为基本算子，相比于传统 CPU、GPU 采用的标量、向量运算，具有更高的运算密度，能够在同等访存能力下达到更强的运算能力。

实验主要包含以下任务：

1. **环境搭建：**在 Ubuntu 22.04 LTS 系统上安装配置 Verilator 和 GTKWave 仿真工具链。
2. **内积运算器设计：**实现 4 元素 16 位有符号整数的并行内积运算，采用两级流水线结构，包含输入寄存器、4 个并行乘法器和加法树。
3. **矩阵乘向量运算器设计：**基于内积运算器，通过实例化 4 个内积运算单元，实现 4×4 权重矩阵与 4 元素输入神经元激活值的矩阵乘向量运算。
4. **矩阵乘法运算器设计：**基于矩阵乘向量运算器，通过实例化 4 个矩阵乘向量运算单元，实现 4×4 矩阵与 4×4 矩阵的矩阵乘法运算。
5. **功能仿真与验证：**编写 C++ 测试文件，生成激励信号，验证三种运算器的功能正确性，并通过 GTKWave 观察波形进行时序分析。
6. **测试框架完善：**设计多组测试数据，编写一键测试脚本，对三种运算器进行全面的功能验证。

二、 实验步骤与设计分析

1 实验环境搭建

实验基于 Ubuntu 22.04 LTS 操作系统，使用 Verilator 作为 Verilog 仿真器，使用 GTKWave 作为波形查看工具。Verilator 是一款开源的 Verilog/SystemVerilog 仿真器和代码生成器，能够将 HDL 代码转换成高性能的 C++ 库，完成数字电路的功能仿真和验证。

首先安装 Verilator 编译所需的依赖包：

```
sudo apt-get install git help2man perl python3 make autoconf g++ flex bison ccache
sudo apt-get install libgoogle-perftools-dev numactl perl-doc
sudo apt-get install libfl2 libfl-dev zlibc zlib1g zlib1g-dev
```

然后下载并编译安装 Verilator：

```
git clone https://gitee.com/mirrors/Verilator.git
cd Verilator
autoconf
./configure
make -j `nproc`
sudo make install
```

最后安装 GTKWave 波形查看工具：

```
sudo apt-get install gtkwave
```

安装完成后，可通过 `verilator --version` 和 `gtkwave --version` 命令验证安装是否成功。

2 内积运算器设计

内积运算器是深度学习处理器矩阵运算单元的基础模块，其核心功能是实现两个向量的乘积累加运算。本实验实现的内积运算器接收 4 个 16 位有符号整数的激活值和 4 个 16 位有符号整数的权重，输出 32 位有符号整数的内积结果。采用两级流水线结构：第一级为输入采样阶段，在时钟上升沿将输入数据锁存到寄存器；第二级为计算输出阶段，4 个乘法器并行计算部分积，随后通过加法树累加得到最终结果。这种设计在保证每个时钟周期完成一次内积运算的同时，有效降低了关键路径延迟，提高了电路的最高工作频率。

使能信号（enable）控制寄存器的更新，当使能信号无效时寄存器保持原值，避免组合逻辑不必要的翻转，从而降低动态功耗。同步复位信号（reset）在高电平时将所有寄存器清零，确保电路初始状态的确定性。

```
module inner_product_4x16 (
    input wire clk,
    input wire reset,
    input wire enable,
    input wire signed [15:0] activations [0:3],
    input wire signed [15:0] weights [0:3],
    output reg signed [31:0] result
);
    // 输入寄存器
    reg signed [15:0] activations_reg [0:3];
    reg signed [15:0] weights_reg [0:3];

    // 组合逻辑：并行乘法与累加
    wire signed [31:0] dot_product =
        (activations_reg[0] * weights_reg[0]) +
        (activations_reg[1] * weights_reg[1]) +
        (activations_reg[2] * weights_reg[2]) +
        (activations_reg[3] * weights_reg[3]);

    // 寄存器更新逻辑
    integer i;
    always @(posedge clk) begin
        if (reset) begin
            for (i = 0; i < 4; i = i + 1) begin
                activations_reg[i] <= 16'd0;
                weights_reg[i] <= 16'd0;
            end
            result <= 32'd0;
        end else if (enable) begin
            for (i = 0; i < 4; i = i + 1) begin
                activations_reg[i] <= activations[i];
                weights_reg[i] <= weights[i];
            end
            result <= dot_product;
```

```
        end
    end
endmodule
```

在设计中，使用了 signed 关键字声明有符号数，确保乘法运算生成有符号数乘法器。

3 矩阵乘向量运算器设计

矩阵乘向量运算可以分解为多个内积运算：权重矩阵的每一行与激活值向量进行内积运算，产生输出向量的一个元素。因此，矩阵乘向量运算器通过并行实例化 4 个内积运算器实现。4 个内积运算器共享同一个激活值向量输入（通过广播机制），分别接收权重矩阵的不同行，并行产生 4 个输出结果。

这种设计相比于分离的 4 个内积运算器具有更高的运算密度：激活值数据仅需读取一次即可参与 4 次内积运算，有效降低了对便笺存储器访问带宽的需求，在同等访存能力下可达到更高的算力。

```
module matrix_vector_mult_4x4x16 (
    input wire clk,
    input wire reset,
    input wire enable,
    input wire signed [15:0] activations [0:3],
    input wire signed [15:0] weights [0:3][0:3],
    output wire signed [31:0] results [0:3]
);

inner_product_4x16 ipu0 (
    .clk(clk),
    .reset(reset),
    .enable(enable),
    .activations(activations),
    .weights(weights[0]),
    .result(results[0])
);

inner_product_4x16 ipu1 (
    .clk(clk),
    .reset(reset),
    .enable(enable),
    .activations(activations),
    .weights(weights[1]),
    .result(results[1])
);

inner_product_4x16 ipu2 (
    .clk(clk),
    .reset(reset),
    .enable(enable),
    .activations(activations),
    .weights(weights[2]),
    .result(results[2])
);
```

```
inner_product_4x16 ipu3 (
    .clk(clk),
    .reset(reset),
    .enable(enable),
    .activations(activations),
    .weights(weights[3]),
    .result(results[3])
);

endmodule
```

在我的设计中，激活值向量通过广播机制同时传递给所有内积运算器，权重矩阵按行分配给各个内积运算器，`weights[i]` 表示第 i 行。输出结果为 `wire` 类型，直接连接到子模块的输出端口。这个运算器继承了内积运算器的两级流水线特性，输出结果延迟 2 拍产生。

4 矩阵乘法运算器设计

矩阵乘法运算可以分解为多个矩阵乘向量运算：激活值矩阵的每一行作为一个向量，与权重矩阵进行矩阵乘向量运算，产生输出矩阵的一行。因此，矩阵乘法运算器通过并行实例化 4 个矩阵乘向量运算器实现。4 个矩阵乘向量运算器共享同一个权重矩阵（通过广播机制），分别接收激活值矩阵的不同行，并行产生 4 行输出结果。

```
module matrix_mult_4x4x4x16 (
    input wire clk,
    input wire reset,
    input wire enable,
    input wire signed [15:0] activations [0:3][0:3],
    input wire signed [15:0] weights [0:3][0:3],
    output wire signed [31:0] results [0:3][0:3]
);

matrix_vector_mult_4x4x16 mxv0 (
    .clk(clk),
    .reset(reset),
    .enable(enable),
    .activations(activations[0]),
    .weights(weights),
    .results(results[0])
);

matrix_vector_mult_4x4x16 mxv1 (
    .clk(clk),
    .reset(reset),
    .enable(enable),
    .activations(activations[1]),
    .weights(weights),
    .results(results[1])
);

matrix_vector_mult_4x4x16 mxv2 (
    .clk(clk),
```

```
    .reset(reset),
    .enable(enable),
    .activations(activations[2]),
    .weights(weights),
    .results(results[2])
);

matrix_vector_mult_4x4x16 mxv3 (
    .clk(clk),
    .reset(reset),
    .enable(enable),
    .activations(activations[3]),
    .weights(weights),
    .results(results[3])
);

endmodule
```

权重矩阵通过广播机制同时传递给所有矩阵乘向量运算器，激活值矩阵按行分配给各个矩阵乘向量运算器，activations[i] 表示第 i 行。整个模块共包含 16 个内积运算器（4×4），可并行完成 16 次内积运算。

权重数据仅需读取一次即可参与 4 个批次的运算，使得运算密度进一步提高。然而，这种设计也存在工程实现上的挑战：随着运算器规模扩大，广播信号的扇出增大、布线距离变长，对电路的时序和功耗提出了更高要求。

5 编译与仿真环境配置

为了验证设计的正确性，需要编写 C++ 测试文件作为仿真顶层，并配置 Makefile 进行自动化编译。

Makefile 定义了三个编译目标，分别对应三种运算器，每个目标指定了需要编译的 Verilog 源文件和 C++ 测试文件：

```
MODULES = inner_product_4x16 matrix_vector_mult_4x4x16 matrix_mult_4x4x4x16
TEST_FILES = tb_inner_product.cpp tb_matrix_vector.cpp tb_matrix_mult.cpp

VERILATOR = verilator
VERILATOR_FLAGS = --cc --exe --build -Wall --trace -I$(SRC_DIR)

# 内积运算器编译
inner_product_4x16:
  $(VERILATOR) $(VERILATOR_FLAGS) --Mdir $(OBJ_DIR)/$@ \
     $(SRC_DIR)/$@.v \
     tb_inner_product.cpp \
     --top-module $@ \
     -CFLAGS "$(CXXFLAGS)"
```

测试文件负责读取数据文件、生成时钟信号、控制复位和使能信号、采集运算结果并与期望值比对。关键功能包括：

1. 从二进制格式的数据文件中读取激活值、权重和期望结果。
2. 生成周期性的时钟信号，控制复位和使能信号的时序。
3. 在适当的时刻将数据送入运算器，并在流水线延迟后采集结果。

4. 比对实际结果与期望结果，输出测试状态（PASS/FAIL）。

5. 可选生成 VCD 波形文件，用于 GTKWave 时序分析。

编译完成后，可通过以下命令运行测试：

```
cd sim
make clean
make all
./obj_dir/inner_product_4x16/Vinner_product_4x16 ../data
```

三、 实验结果及分析

1 初步运行结果

本小节展示使用原始数据目录（../data）对三种运算器进行的初步功能验证。数据文件以 16 位二进制格式存储，包含神经元激活值（neuron）、权重（weight）和期望结果（result）。

1.1 内积运算器测试结果

内积运算器测试输入为激活值向量 $[1, 2, 3, 4]$ 和权重向量 $[1, 0, 0, 0]$，期望输出结果为 1。图 1展示了终端运行结果，可以看到实际输出结果为 1，与期望结果一致，验证通过。

```
Chavapa@Chavapa-linux101:~/AICS-lab/AICS_dlp_ex/sim$ ./obj_dir/inner_product_4x16/Vinner_product_4x16

=== 内积运算器测试（单组数据）===
INFO: ../data/neuron 加载 16 个数据
INFO: ../data/weight 加载 16 个数据
INFO: 预期结果加载成功（十进制：1）
INFO: 权重文件包含16个数据，取前4个用于内积测试

=== 输入数据 ===
神经元数据（十进制）：1 2 3 4
权重数据（十进制）：  1 0 0 0
预期结果（十进制）：  1
INFO: 时钟上升沿 @ 时间 0
INFO: 时钟上升沿 @ 时间 2
INFO: 时钟上升沿 @ 时间 4
INFO: 释放复位 @ 时间 5
INFO: 时钟上升沿 @ 时间 6
INFO: 时钟上升沿 @ 时间 8
INFO: 加载数据 @ 时间 9
INFO: 时钟上升沿 @ 时间 10
INFO: 使能计算 @ 时间 11
INFO: 时钟上升沿 @ 时间 12
INFO: 时钟上升沿 @ 时间 14
INFO: 关闭使能 @ 时间 15
INFO: 时钟上升沿 @ 时间 16
INFO: 时钟上升沿 @ 时间 18
INFO: 时钟上升沿 @ 时间 20
INFO: 时钟上升沿 @ 时间 22
INFO: 时钟上升沿 @ 时间 24
INFO: 时钟上升沿 @ 时间 26
INFO: 在时间 26 捕获结果
INFO: 时钟上升沿 @ 时间 28
INFO: 时钟上升沿 @ 时间 30
INFO: 时钟上升沿 @ 时间 32
INFO: 时钟上升沿 @ 时间 34
INFO: 时钟上升沿 @ 时间 36
INFO: 时钟上升沿 @ 时间 38
INFO: 时钟上升沿 @ 时间 40
INFO: 时钟上升沿 @ 时间 42
INFO: 时钟上升沿 @ 时间 44
INFO: 时钟上升沿 @ 时间 46
INFO: 时钟上升沿 @ 时间 48
INFO: 时钟上升沿 @ 时间 50
INFO: 时钟上升沿 @ 时间 52
INFO: 时钟上升沿 @ 时间 54
INFO: 时钟上升沿 @ 时间 56
INFO: 时钟上升沿 @ 时间 58

=== 结果验证 ===
实际结果（十进制）：1
预期结果（十进制）：1
验证结果：通过
```

图 1: 内积运算器终端测试结果

测试过程详细记录了仿真的各个阶段：

- 时间 0-5：复位阶段，所有寄存器被清零。
- 时间 9：数据加载，将激活值和权重送入运算器输入端口。
- 时间 11：使能信号拉高，运算器开始工作。
- 时间 15：使能信号关闭。
- 时间 26：捕获结果，此时流水线延迟已过，输出结果稳定。

图 2展示了 GTKWave 波形图。从波形中可以清晰地观察到：

- `clk` 信号周期性翻转，提供同步时钟。
- `reset` 信号在初始阶段为高电平，随后拉低。
- `activations[0:3]` 和 `weights[0:3]` 在时间 9 加载数据。
- `activations_reg[0:3]` 和 `weights_reg[0:3]` 在使能信号有效后的第一个时钟上升沿锁存数据。
- `dot_product` 信号立即计算出内积结果（组合逻辑）。
- `result` 信号在下一个时钟上升沿输出最终结果（0x00000001，即十进制 1）。

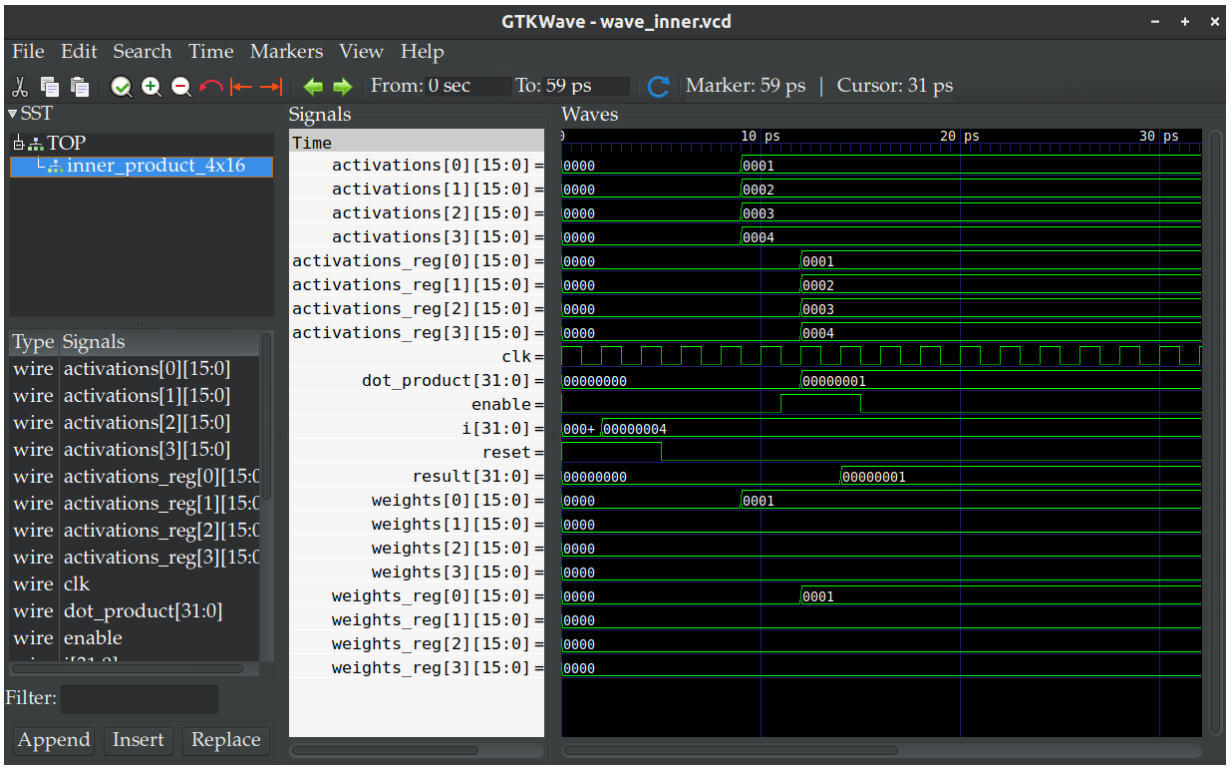

图 2: 内积运算器 GTKWave 波形图

波形验证了两级流水线的设计：输入寄存器在第一拍锁存数据，输出寄存器在第二拍输出结果，整个流水线延迟为 2 个时钟周期。

1.2 矩阵乘向量运算器测试结果

矩阵乘向量运算器测试输入为激活值向量 $[1, 2, 3, 4]$ 和单位权重矩阵：

$$W = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

期望输出结果为 $[1, 2, 3, 4]$。图 3展示了终端运行结果，实际输出的 4 个结果分别为 1、2、3、4，与期望结果完全一致，验证通过。

```
Chavapa@Chavapa-linux101:~/AICS-lab/AICS_dlp_ex/sim$ ./obj_dir/matrix_vector_mult_4x4x16/Vmatrix_vector_mult_4x4x16

=== 矩阵乘向量测试 ===
INFO: 向量维度 (16进制4 4 4 转换为10进制) = 4
INFO: ../data/neuron 加载 16 个数据
INFO: ../data/weight 加载 16 个数据
INFO: 结果文件加载 16 个结果
INFO: 激活值包含16个数据，取前4个
INFO: 结果包含16个数据，取前4个

=== 输入数据 ===
矩阵 (weights, 4x4) :
1       0       0       0
0       1       0       0
0       0       1       0
0       0       0       1
向量 (activations) : 1 2 3 4
=== 结果验证 ===
结果1: 实际=1, 预期=1 [通过]
结果2: 实际=2, 预期=2 [通过]
结果3: 实际=3, 预期=3 [通过]
结果4: 实际=4, 预期=4 [通过]
```

图 3: 矩阵乘向量运算器终端测试结果

图 4展示了 GTKWave 波形图。从波形中可以观察到：

- 左侧 SST 面板显示了模块层次结构:顶层模块 `matrix_vector_mult_4x4x16` 包含 4 个子模块 `ipu0-ipu3`，每个子模块都是一个 `inner_product_4x16` 实例。
- `activations[0:3]` 作为共享输入，广播到所有内积运算器。
- `weights[0:3][0:3]` 表示 4×4 权重矩阵，`weights_reg[i]` 显示各内积运算器锁存的权重行。
- `results[0:3]` 同时输出 4 个内积结果，分别为 0x00000001、0x00000002、0x00000003、0x00000004。

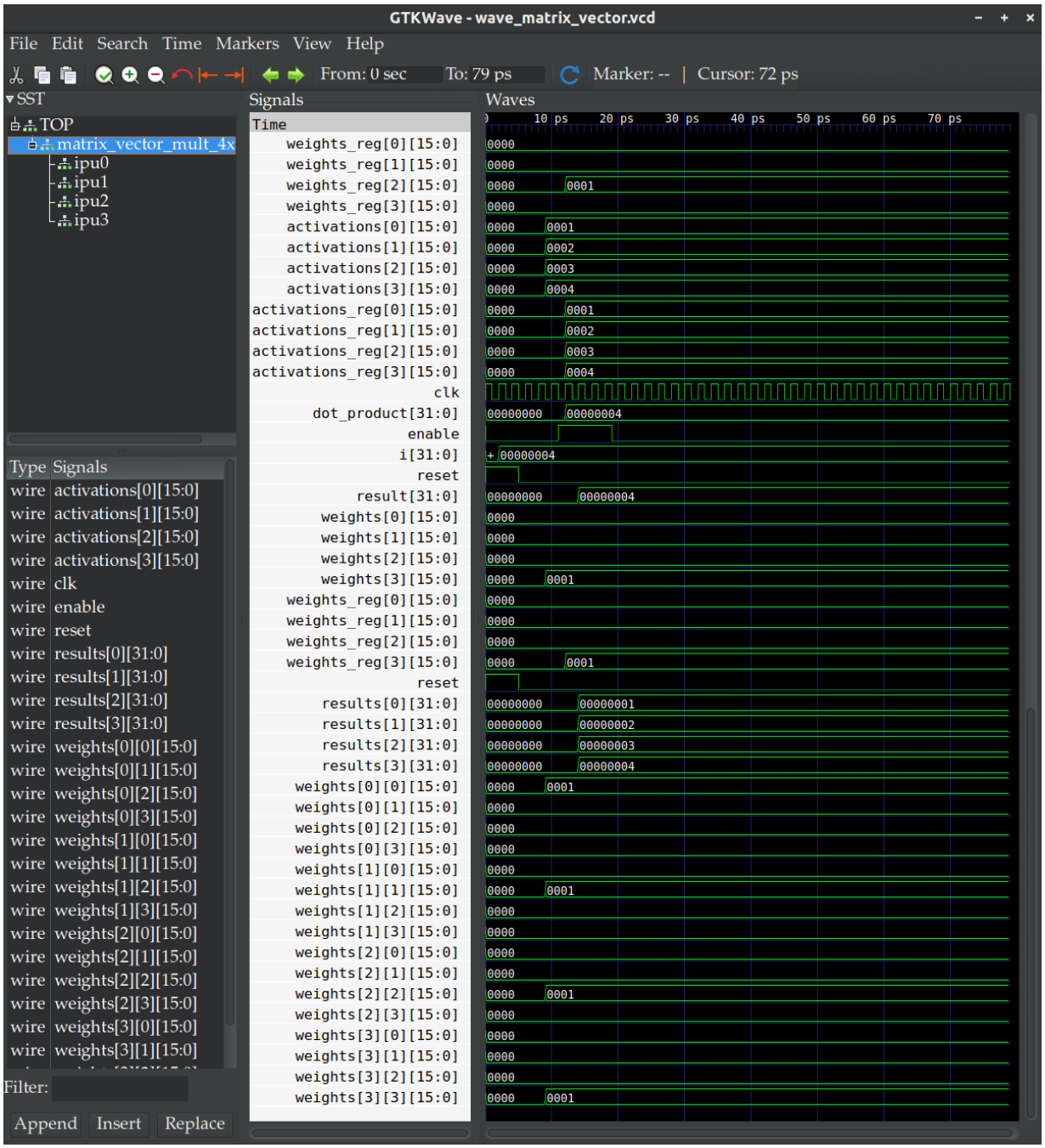

图 4: 矩阵乘向量运算器 GTKWave 波形图

波形验证了矩阵乘向量运算器的并行特性：4 个内积运算器同时工作，在同一个时钟周期产生 4 个输出结果。

1.3 矩阵乘法运算器测试结果

矩阵乘法运算器测试输入为激活值矩阵和权重矩阵均为单位矩阵，期望输出为单位矩阵。图 5展示了终端运行结果，实际输出的 16 个结果（4×4 矩阵）与期望结果完全一致，所有结果验证通过。

```
Chavapa@Chavapa-linux101:~/AICS-lab/AICS_dlp_ex/sim$ ./obj_dir/matrix_mult_4x4x4x16/Vmatrix_mult_4x4x4x16


=== 矩阵乘矩阵测试 ===
INFO: 矩阵维度（16进制转换为10进制）: A(4x4), B(4x4), 结果(4x4)
INFO: ../data/neuron 加载 16 个数据
INFO: ../data/weight 加载 16 个数据
INFO: 结果文件加载 16 个结果

=== 输入数据 ===
激活值矩阵（activations, 4x4）:
1       2       3       4
1       2       3       4
1       2       3       4
1       2       3       4

权重矩阵（weights, 4x4）:
1       0       0       0
0       1       0       0
0       0       1       0
0       0       0       1

=== 结果验证 ===
结果[0][0]: 实际=1, 预期=1 [通过]
结果[0][1]: 实际=2, 预期=2 [通过]
结果[0][2]: 实际=3, 预期=3 [通过]
结果[0][3]: 实际=4, 预期=4 [通过]
结果[1][0]: 实际=1, 预期=1 [通过]
结果[1][1]: 实际=2, 预期=2 [通过]
结果[1][2]: 实际=3, 预期=3 [通过]
结果[1][3]: 实际=4, 预期=4 [通过]
结果[2][0]: 实际=1, 预期=1 [通过]
结果[2][1]: 实际=2, 预期=2 [通过]
结果[2][2]: 实际=3, 预期=3 [通过]
结果[2][3]: 实际=4, 预期=4 [通过]
结果[3][0]: 实际=1, 预期=1 [通过]
结果[3][1]: 实际=2, 预期=2 [通过]
结果[3][2]: 实际=3, 预期=3 [通过]
结果[3][3]: 实际=4, 预期=4 [通过]

验证结果: 全部通过
```

图 5: 矩阵乘法运算器终端测试结果

图 6展示了 GTKWave 波形图。从波形中可以观察到：

- 左侧 SST 面板显示了三层模块层次结构：顶层 `matrix_mult_4x4x4x16` 包含 4 个 `mxv0-mxv3`（矩阵乘向量运算器），每个 mxv 又包含 4 个 `ipu0-ipu3`（内积运算器），总共 16 个内积运算器并行工作。
- `weights[0:3][0:3]` 作为共享输入，广播到所有矩阵乘向量运算器。
- `activations[0:3][0:3]` 表示 4×4 激活值矩阵，按行分配给不同的矩阵乘向量运算器。
- `results[0:3][0:3]` 输出 4×4 结果矩阵，每个元素为 32 位有符号整数。

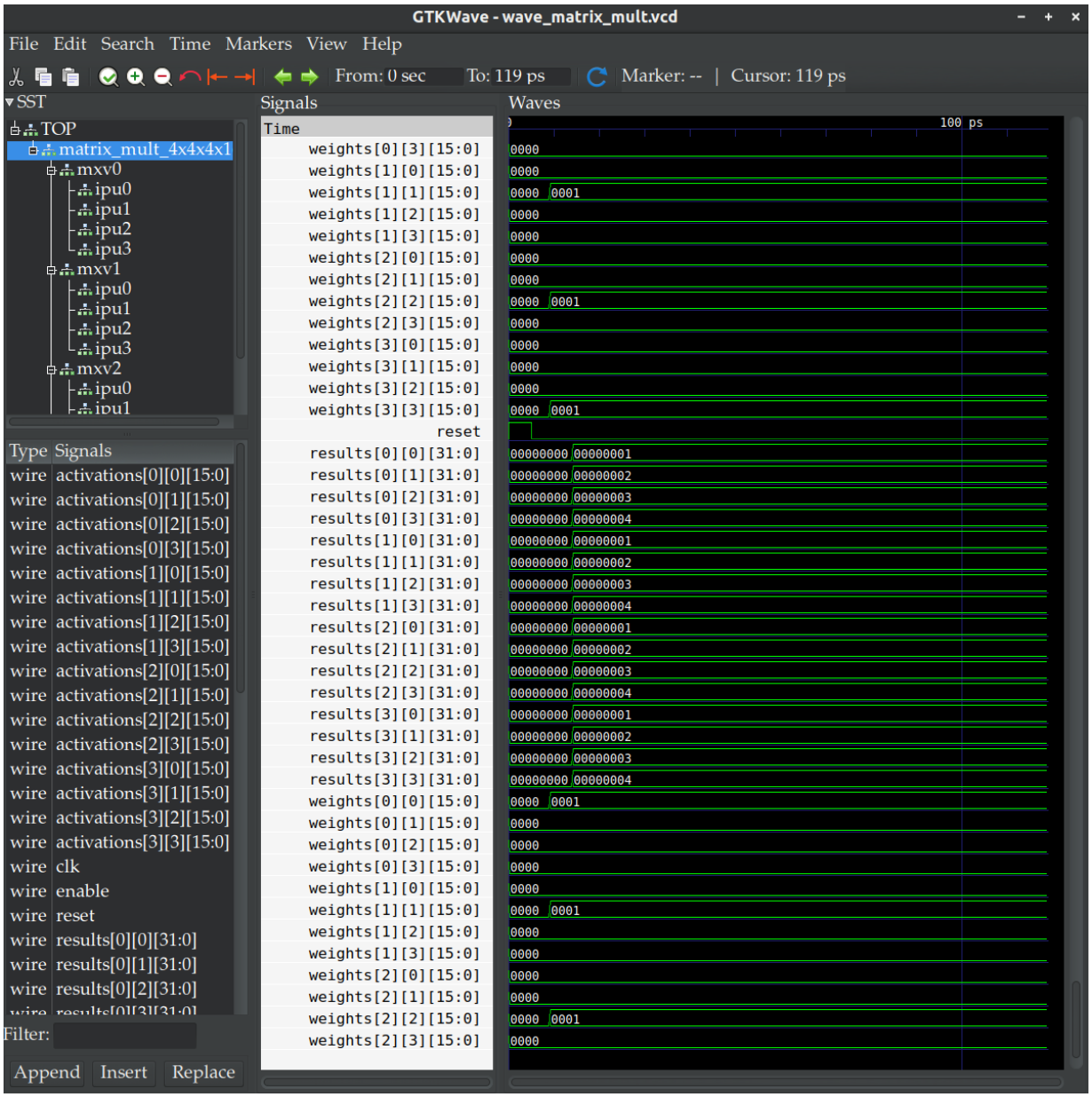

图 6: 矩阵乘法运算器 GTKWave 波形图

波形验证了矩阵乘法运算器的高度并行特性：16 个内积运算器同时工作，一个时钟周期完成 4×4 矩阵乘法的所有 16 次内积运算。

2 对更多不同测试数据的验证

为了更全面地验证三种运算器的功能正确性，我设计了 4 组不同的测试数据，并编写了 `run_all_tests.sh` 一键测试脚本。测试数据包括：

- **Test1：**单位矩阵测试，验证基本功能。
- **Test2：**全 1 矩阵测试，验证累加功能。
- **Test3：**对角缩放测试，验证不同位置的数据处理。
- **Test4：**非对称矩阵测试，验证一般情况下的计算正确性。

测试脚本实现：

```
#!/bin/bash

cd "$(dirname "$0")"

echo "========== Deep Learning Processor Test Report =========="
echo ""

# Compile
make clean > /dev/null 2>&1
make all > /dev/null 2>&1

# Test counters
total=0
passed=0

# Test Inner Product
echo "[1] Inner Product (4x16-bit):"
for i in 1 2 3 4; do
    total=$((total + 1))
    echo -n " Test$i: "
    ./obj_dir/inner_product_4x16/Vinner_product_4x16 ../data/test$i 2>/dev/null
    if [ $? -eq 0 ]; then
        passed=$((passed + 1))
    fi
done

# Test Matrix-Vector
echo "[2] Matrix-Vector Mult (4x4x16-bit):"
for i in 1 2 3 4; do
    total=$((total + 1))
    echo -n " Test$i: "
    ./obj_dir/matrix_vector_mult_4x4x16/Vmatrix_vector_mult_4x4x16 ../data/test$i 2>/dev/null
    if [ $? -eq 0 ]; then
        passed=$((passed + 1))
    fi
done

# Test Matrix-Matrix
echo "[3] Matrix-Matrix Mult (4x4x4x16-bit):"
for i in 1 2 3 4; do
    total=$((total + 1))
    echo -n " Test$i: "
    ./obj_dir/matrix_mult_4x4x4x16/Vmatrix_mult_4x4x4x16 ../data/test$i 2>/dev/null
    if [ $? -eq 0 ]; then
        passed=$((passed + 1))
    fi
done
```

```
# Summary
echo "============================================================"
echo "Result: $passed/$total tests passed"
if [ $passed -eq $total ]; then
    echo "Status: ALL PASS"
else
    echo "Status: SOME FAILED"
fi
```

脚本自动完成编译、对 3 种运算器分别运行 4 组测试数据、统计通过率并输出测试报告。

图 7展示了运行 `run_all_tests.sh` 脚本的完整输出。可以看到：

```
Chavapa@Chavapa-linux101:~/AICS-lab/AICS_dlp_ex/sim$ ./run_all_tests.sh
========== Deep Learning Processor Test Report ==========

[1] Inner Product (4x16-bit):
  Test1: A=[1,2,3,4] W=[1,0,0,0] Expected=1 Actual=1 [PASS]
  Test2: A=[1,1,1,1] W=[1,1,1,1] Expected=4 Actual=4 [PASS]
  Test3: A=[1,2,3,4] W=[2,0,0,0] Expected=2 Actual=2 [PASS]
  Test4: A=[2,3,1,4] W=[1,2,0,0] Expected=8 Actual=8 [PASS]

[2] Matrix-Vector Mult (4x4x16-bit):
  Test1: A=[1,2,3,4] Expected=[1,2,3,4] Actual=[1,2,3,4] [PASS]
  Test2: A=[1,1,1,1] Expected=[4,4,4,4] Actual=[4,4,4,4] [PASS]
  Test3: A=[1,2,3,4] Expected=[2,4,6,8] Actual=[2,4,6,8] [PASS]
  Test4: A=[2,3,1,4] Expected=[8,5,9,8] Actual=[8,5,9,8] [PASS]

[3] Matrix-Matrix Mult (4x4x4x16-bit):
  Test1: Matrix(4x4x4) Expected=[1,2,3,4,1,2,3,4,1,2,3,4,1,2,3,4] Actual=[1,2,3,4,1,2,3,4,1,2,3,4,1,2,3,4] [PASS]
  Test2: Matrix(4x4x4) Expected=[4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4] Actual=[4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4] [PASS]
  Test3: Matrix(4x4x4) Expected=[2,4,6,8,2,4,6,8,2,4,6,8,2,4,6,8] Actual=[2,4,6,8,2,4,6,8,2,4,6,8,2,4,6,8] [PASS]
  Test4: Matrix(4x4x4) Expected=[8,5,9,8,8,5,9,8,8,5,9,8,8,5,9,8] Actual=[8,5,9,8,8,5,9,8,8,5,9,8,8,5,9,8] [PASS]

============================================================
Result: 12/12 tests passed
Status: ALL PASS
```

图 7: 一键测试脚本运行结果

1. **内积运算器**通过了所有 4 组测试，包括：
   - Test1: $[1,2,3,4]\cdot[1,0,0,0]=1$
   - Test2: $[1,1,1,1]\cdot[1,1,1,1]=4$
   - Test3: $[1,2,3,4]\cdot[2,0,0,0]=2$
   - Test4: $[2,3,1,4]\cdot[1,2,0,0]=8$
2. **矩阵乘向量运算器**通过了所有 4 组测试，验证了与单位矩阵、全 1 矩阵、对角矩阵、非对称矩阵的乘法运算。
3. **矩阵乘法运算器**通过了所有 4 组测试，验证了 4×4 矩阵乘法的完整功能。
4. 总计 12 个测试全部通过（12/12），状态显示 `ALL PASS`。

测试结果表明，三种运算器的设计完全正确，能够处理各种不同的输入数据，满足设计要求。

四、 实验过程中遇到的问题

1 Verilator 编译环境配置问题

在 Ubuntu 系统上初次安装 Verilator 时，执行 `make` 命令时报错，提示缺少 `flex` 和 `bison` 等依赖库。发现问题后，我参照讲义第 8.3.1 节的安装指南，系统地安装了必需的依赖包。虽然在安装过程中出现少许报错，但

重新编译 Verilator 成功。

2 模块实例化端口连接问题

在实现矩阵乘向量运算器时，初次尝试实例化内积运算器，遇到端口连接错误。对于数组类型的端口（如 `weights[0:3][0:3]`），我不确定如何将二维数组的某一行连接到子模块。查阅资料后得知，Verilog 支持数组切片语法，可以直接使用 `weights[i]` 表示二维数组的第 i 行。正确的连接方式为：

```
inner_product_4x16 ipu0 (
    .activations(activations),
    .weights(weights[0]), // 连接权重矩阵的第0行
    .result(results[0])
);
```

五、 对思考题的回答

1 运算器资源对比与规模分析

题目

对比分析内积运算器、矩阵乘向量运算器和矩阵乘法运算器内部的乘法器数量、加法器数量和对外数据接口数量，计算并对比每种设计中这些元素的比例。如果增大三种运算器的规模（例如，向量长度从 4 增加到 16 乃至 128），这些设计的比例分别会发生什么样的变化？

解答

假设向量长度为 n（本实验中 $n=4$），三种运算器的乘法器数量分别为 n、n^2、n^3，加法器数量分别为 $n-1$、$n(n-1)$、$n^2(n-1)$。在数据接口方面，内积运算器需要 $2n$ 个 16 位输入和 1 个 32 位输出，总接口位宽为 $32n+32$ 位；矩阵乘向量运算器需要 $n+n^2$ 个 16 位输入和 n 个 32 位输出，总接口位宽为 $16n(n+3)$ 位；矩阵乘法运算器需要 $2n^2$ 个 16 位输入和 n^2 个 32 位输出，总接口位宽为 $64n^2$ 位。

定义运算密度 $\rho = \frac{\text{乘法器数量}}{\text{总接口位宽}}$ 来衡量单位数据接口的运算能力。在 $n=4$ 时，三种运算器的运算密度分别为 0.025、0.036、0.063，矩阵乘法运算器是内积运算器的 2.5 倍。通过数学推导可得，内积运算器的运算密度 $\rho_{inner} = \frac{n}{32(n+1)} \approx 1/32$（$n$ 较大时），矩阵乘向量运算器的运算密度 $\rho_{mv} = \frac{n}{16(n+3)} \approx 1/16$，而矩阵乘法运算器的运算密度 $\rho_{mm} = \frac{n}{64}$ 与 n 成正比。

当向量长度从 4 增加到 128 时，内积和矩阵乘向量运算器的运算密度几乎不变，而矩阵乘法运算器的运算密度从 0.063 提升到 2.0，增长了 32 倍。这意味着在大规模运算中，矩阵乘法运算器每位数据接口可支持 2 个乘法运算，充分体现了其在大规模神经网络中的巨大优势。实际工程中需要在运算密度和布线时序等挑战之间平衡，现代 DLP 芯片通常采用 16×16 到 32×32 的矩阵乘法单元。

2 深度学习处理器加速原理

题目

请说明深度学习处理器加速深度学习计算的基本原理是什么？

解答

深度学习处理器加速的核心在于采用矩阵运算作为基本算子。传统 CPU 和 GPU 使用标量或向量运算，执行矩阵运算时需要多次访存和发射指令，运算密度低。而 DLP 将矩阵乘向量、矩阵乘法作为基本算子，一个时钟

周期即可完成数十乃至数百次乘加操作，如本实验的矩阵乘法运算器一个周期完成 64 次乘法和 48 次加法。

数据复用是另一关键机制。在矩阵乘向量运算中，激活值向量被所有内积运算器共享，仅需读取一次；在矩阵乘法运算中，权重矩阵进一步被所有矩阵乘向量运算器共享，总体数据复用率可达 16 倍。这显著降低了对存储器访问带宽的需求，突破了深度学习运算中的“Memory Wall”瓶颈。DLP 还配备专用的便笺存储器暂存数据（如讲义中的图 8.1），通过 DMA 将主存访问与运算流水化。

此外，DLP 采用大规模并行计算（数百个运算单元同时工作）、低精度数据格式（INT8/INT16 降低面积和功耗）、以及针对深度学习算法特征的定制化设计，去除通用处理器中不必要的控制逻辑。通过这些综合设计，DLP 在深度学习任务上相比通用处理器实现了数十倍乃至数百倍的性能和能效优势，这正是 Google TPU、NVIDIA Tensor Core、华为昇腾等 AI 芯片的核心设计思想。

六、 实验总结与心得感想

通过实验，我深刻体会到深度学习处理器采用矩阵运算作为基本算子的优越性。相比于传统 CPU、GPU 采用的标量、向量运算，矩阵运算具有更高的运算密度。这种优势体现在两个方面：

首先是**数据复用**。在矩阵乘向量运算中，激活值向量被 4 个内积运算器共享，仅需读取一次即可参与 4 次内积运算；在矩阵乘法运算中，权重矩阵被 4 个矩阵乘向量运算器共享，数据复用程度进一步提高。这种设计显著降低了对便笺存储器访问带宽的需求，使得在同等访存能力下能够达到更高的算力。

其次是**并行度**。内积运算器包含 4 个并行乘法器，矩阵乘向量运算器包含 4 个并行内积运算器（16 个乘法器），矩阵乘法运算器包含 16 个并行内积运算器（64 个乘法器）。这种层次化并行结构使得大规模矩阵运算能够在极短时间内完成。以本实验实现的矩阵乘法运算器为例，每个时钟周期可完成 16 次内积运算，相当于 64 次乘法和 48 次加法，这在传统标量处理器上需要数十甚至上百个时钟周期才能完成。

展望未来，我希望能够在本实验的基础上进行更深入的探索。例如，可以尝试扩大运算器规模，从 4×4 扩展到 16×16 甚至更大；可以尝试实现更复杂的数据格式，如浮点数或低比特量化；可以尝试优化电路性能，如增加流水线级数、采用 Wallace 树加法器等。更进一步，可以将运算器部署到 FPGA 上进行实际测试，体验硬件加速的真实效果。

最后，衷心感谢讲义提供的详细指导和实验框架，也感谢老师和助教的悉心指导！