基于CNN的深度学习硬件加速器

姓名：\*\* 学号：202130\*\*\*\*

**一、概论**

本实验的全部代码、更多文档请参考笔者的Github仓库：<https://github.com/ykykzq/CNN_Accelerator>。

随着登纳德缩放比例定律与摩尔定律的终结，其他提升微处理系统性价比与能效比的策略大多已经乏力。工程师们开始采用针对特定领域场景下的运算特性进行优化的方式，通过牺牲通用性来换取更高的算力。

在能效——通用性曲线上，比较有代表性的三种架构为CPU、GPU、ASIC或DSA。三种电路的通用性递减，而特定场景下的相对性能逐渐提高。

CPU是最为经典的体系结构，能完成非常复杂的计算类型，并承担非常复杂的处理任务。由于需要较高的通用性，CPU没有针对任何的特定领域进行针对性优化，这里当然也包括深度学习中的矩阵运算。CPU拥有自己的指令集架构，用于软件控制CPU执行不同操作。

GPU是与CPU截然不同的异构计算架构。GPU针对的是大规模并行浮点运算，能够执行从CPU卸载来的计算任务。GPU并非针对深度学习诞生，实际上其对于所有的简单重复运算都可以进行优化，特别是矩阵操作。

ASIC（Application Specific Integrated Circuit，专用集成电路）和DSA（Domain Specific Architecture，领域专用体系结构）是比较类似的概念，在这里不需要非常明确的区分。二者针对的是某类特定场景下的特定任务。在进行设计的过程中，设计师需要看到该场景的内在并行性，并进行针对性设计。例如，在深度学习场景下，进行卷积操作时，原始图像可以同时与同一个卷积核完成在不同位置的卷积运算，从而大大增加运行速度。

本次实验中，我们设计的电路非常特化，具体来说，只针对CNN这一种神经网络，只关心卷积、池化、全连接三种算子。在本设计中，我们不需要设计系统专用的指令集，只需要使用CPU来控制我们系统的具体行为。在本实验中，这是通过PYNQ库加载并执行我们设计的overlay实现的。在设计中，数据从CPU经由AXI总线发送到运算单元进行运算，因此内存访问的开销不在我们的设计与优化范畴之内。

**二、软硬件环境**

**1.硬件支撑**

**1.1 PC设备**

毫无疑问我们需要使用个人电脑，这里电脑主要在两个步骤起到作用。

一是用于HLS生成IP核和设计电路并综合生成比特流。

二是作为主机，控制向开发板中PYNQ镜像的写入、overlays的加载、运算的完成。

**1.2 基于ZYNQ的开发板**

ZYNQ系列是Xilinx公司推出的全可编程片上系统，包含PS（Processing System，处理器系统）和PL（Programmable Logic，可编程逻辑）两部分。

Zynq SoC整合了ARM双核处理器和Xilinx 7系列 FPGA 架构，实际上是一个片上系统，不仅拥有ASIC在能耗、性能和兼容性方面的优势，而且具有FPGA可编程性的优点。

根据官网，当前支持ZYNQ的开发板型号如下表所示。

我们使用的开发板为领航者ZYNQ 7010版（XC7Z010CLG100-1）。该开发板价格较低，内存也满足我们的要求，是比较适合作为开发硬件的设备。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Board | PYNQ-Z2 | PYNQ-Z1 | PYNQ-ZU | Kria KV260\* | Kria KR260\* | ZCU104 | RFSoC 2x2 | RFSoC 4x2 |
| SD card image | [v3.0.1](https://bit.ly/pynqz2_v3_0_1" \o "https://bit.ly/pynqz2_v3_0_1) | [v3.0.1](https://bit.ly/pynqz1_v3_0_1" \o "https://bit.ly/pynqz1_v3_0_1) | [v3.0.1](https://bit.ly/pynqzu_v3_0_1" \o "https://bit.ly/pynqzu_v3_0_1) | [Ubuntu 22.04](https://www.xilinx.com/products/som/kria/kv260-vision-starter-kit/kv260-getting-started-ubuntu/setting-up-the-sd-card-image.html" \o "https://www.xilinx.com/products/som/kria/kv260-vision-starter-kit/kv260-getting-started-ubuntu/setting-up-the-sd-card-image.html) | [Ubuntu 22.04](https://www.xilinx.com/products/som/kria/kr260-robotics-starter-kit/kr260-getting-started/setting-up-the-sd-card-image.html" \o "https://www.xilinx.com/products/som/kria/kr260-robotics-starter-kit/kr260-getting-started/setting-up-the-sd-card-image.html) | [v3.0.1](https://bit.ly/zcu104_v3_0_1" \o "https://bit.ly/zcu104_v3_0_1) | [v3.0.1](https://bit.ly/rfsoc2x2_v3_0_1" \o "https://bit.ly/rfsoc2x2_v3_0_1) | [v3.0.1](https://bit.ly/rfsoc4x2_v3_0_1" \o "https://bit.ly/rfsoc4x2_v3_0_1) |
| Previous versions | v2.7 v2.6 | v2.7 v2.6 | v2.7 v2.6 |  |  | v2.7 v2.6 | v2.7 v2.6 | [v2.7](https://bit.ly/rfsoc4x2_v2_7" \o "https://bit.ly/rfsoc4x2_v2_7) |

续表

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Board | ZCU111 | ZCU208 | Ultra96V2 | Ultra96 (legacy) | ZUBoard 1CG | TySOM-3-ZU7EV | TySOM-3A-ZU19EG |
| SD card image | [v3.0.1](https://bit.ly/zcu111_v3_0_1" \o "https://bit.ly/zcu111_v3_0_1) | [v3.0.1](https://bit.ly/zcu208_v3_0_1" \o "https://bit.ly/zcu208_v3_0_1) | [v3.0.1](https://bit.ly/ultra96v2_v3_0_1" \o "https://bit.ly/ultra96v2_v3_0_1) | [v3.0.1](https://bit.ly/ultra96v1_v3_0_1" \o "https://bit.ly/ultra96v1_v3_0_1) | [v3.0.1](https://bit.ly/zuboard_v3_0_1" \o "https://bit.ly/zuboard_v3_0_1) | [v3.0.1](https://bit.ly/Tysom3_v3_0_1" \o "https://bit.ly/Tysom3_v3_0_1) | [v3.0.1](https://bit.ly/Tysom3a_v3_0_1" \o "https://bit.ly/Tysom3a_v3_0_1) |
| Previous versions | v2.7 v2.6 |  | v2.7 v2.6 | v2.7 v2.6 |  | [v2.7](https://bit.ly/tysom3_v2_7" \o "https://bit.ly/tysom3_v2_7) | [v2.7](https://bit.ly/tysom3a_v2_7" \o "https://bit.ly/tysom3a_v2_7) |

表1 支持PYNQ的硬件

**2.软件环境**

**2.1 Vivado软件**

软件与硬件环境需要适配。考虑到硬件支持的PYNQ版本，本设计中使用的Vivado软件版本为2018.3。

Vivado软件套件包括Vivado及Vitis HLS等。Vitis HLS软件负责生成卷积层IP核。基于HLS技术，软件可以基于我们编写的C++代码，生成特定功能的IP核。在Vivado中，我们可以利用自己编写的卷积层、池化层两个IP核，同时利用Xilinx官方提供的AXI等IP核完成电路的总体设计，并最终生成我们的overlay。

**2.2 Python环境**

为了将overlay加载到板子上，我们需要PYNQ、Jupyter两个Python库。我们这里使用的PYNQ版本为2.5.1。在这之后的版本删除了部分方法，为我们的实验造成困难。Jupyter lab可以使用1.1.3及以上版本，Python版本与之兼容即可。镜像中已经提前预置了对应版本的Python及库。

**2.3 Linux系统**

PYNQ目前只能在Linux下运行。本实验中，在镜像中已经烧录了Ubtuntu18.04。

**三、实验步骤**

实验步骤如下图所示。先编写C++代码，经过Vitis综合后生成IP核；然后再在Vivado中利用block design功能，并使用Xilinx官方提供的Floating Point、AXI4总线等IP核，把各种IP核组装起来，完成一系列操作之后，综合并最终生成overlay。

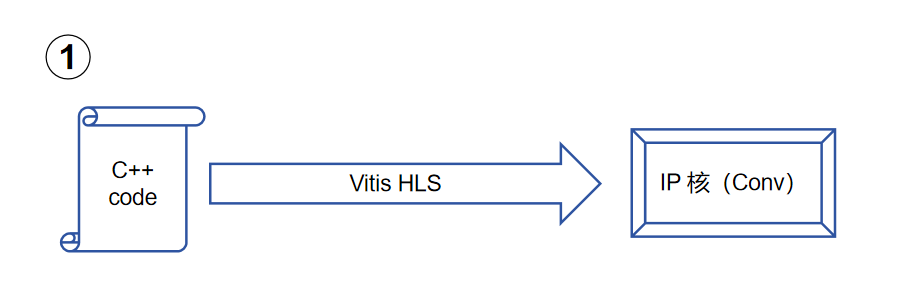


图1 使用Vitis HLS生成用于执行卷积的IP核

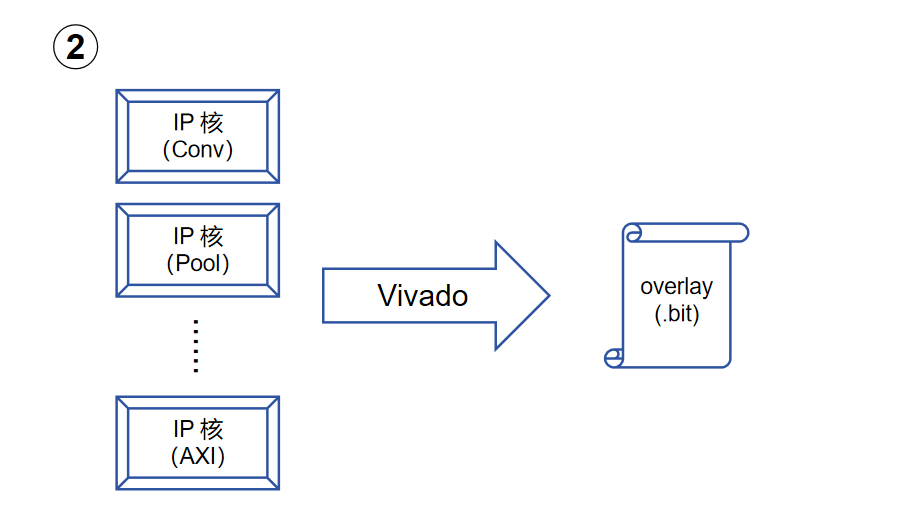


图2 基于Xilinx提供的IP核与Conv IP核完成总体设计

生成overlay之后，在Linux环境下配置好Python环境，并安装兼容版本的的Jupyter lab及PYNQ库。安装完成之后，在Jupyter Lab中连接PYNQ板，并把overlay加载到其中，基于PYNQ硬件运行CNN网络，并输出最终的检验结果以验证正确性。

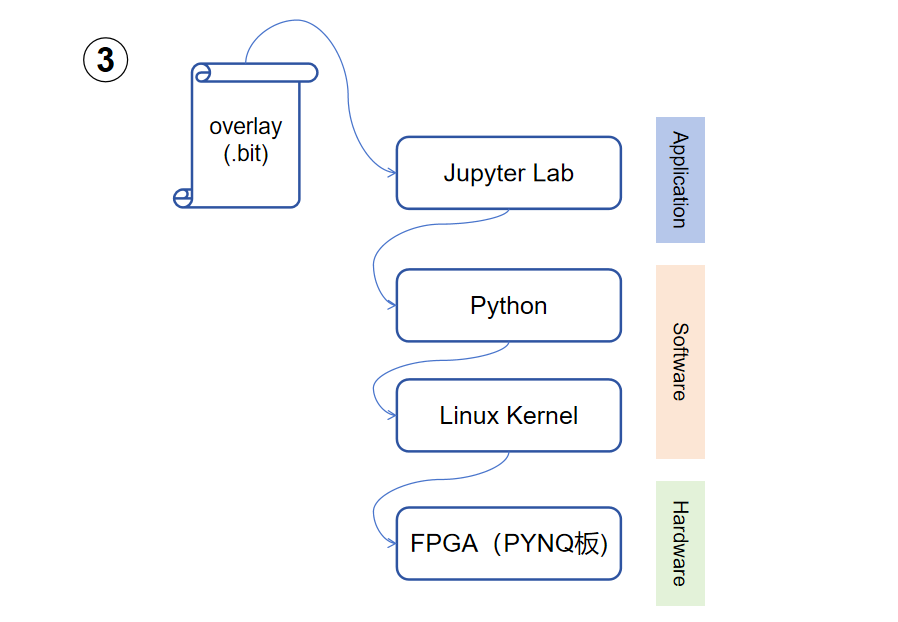


图3 加载overlay并执行CNN运算

下面给出设计并实现的步骤的简单解释，详细的步骤可以参考笔者的Github仓库。

**3.1 Vitis HLS生成IP核**

用于生成卷积神经网络的代码可在笔者的Github仓库中找到。首先需要创建工程。project name跟top function都填写为conv\_core即可。

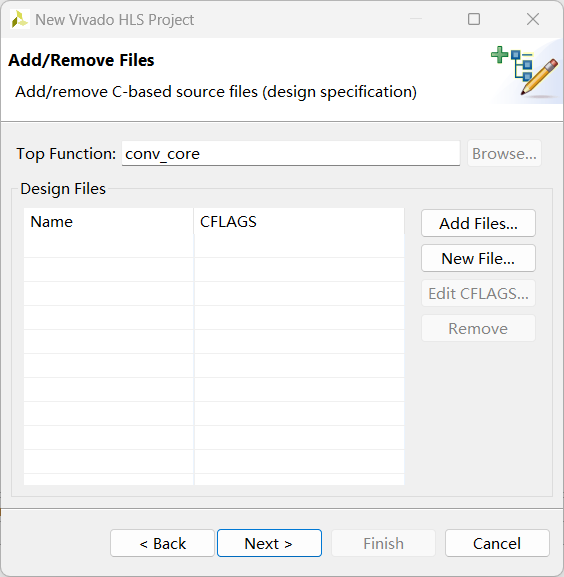


图4 选择top function

注意在下列界面选择FPGA的型号为xc7z010clg400-1。

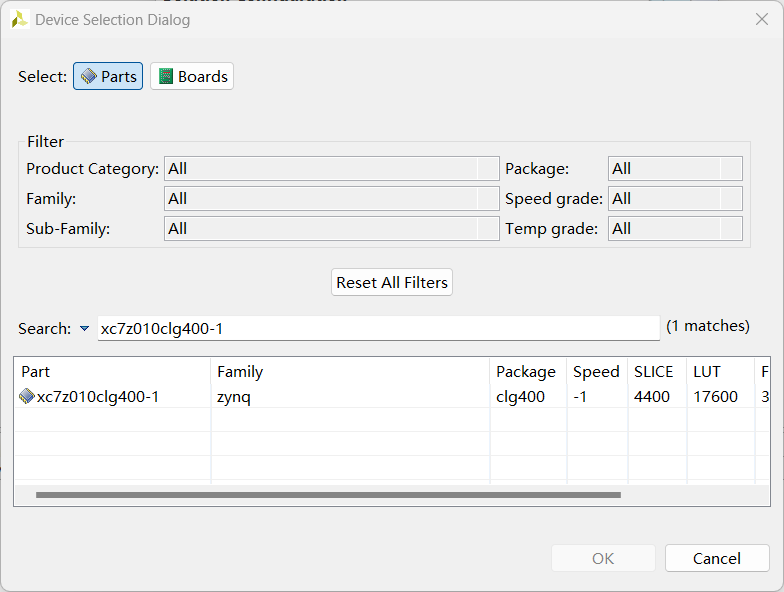


图5 选择FPGA芯片型号

然后把文件添加进去。把Conv\_core.cpp和Conv\_core.h添加到Source，把main.cpp添加到Test Bench里，如下所示。

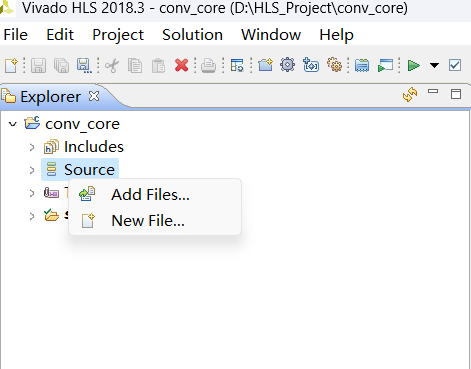


图6 添加CPP文件

然后可以启动RUN C Simulation验证功能正确性。在这之后，更改Project settings中Synthesis选项卡中的top function为Conv(conv\_core.cpp)。之后完成Synthesis，并导出Conv操作的IP核。

**3.2 Vivado实现并综合生成overlay**

本次实验中，我们不采用直接编写Verilog代码的方式完成设计，而是采用更高抽象层次的技术，利用我们以及官方的IP核，使用类似搭积木的方式完成总体设计。

总体上需要创建block design、导入conv IP核并设计、导入pool IP核并设计、分配内存、综合并生成比特流这些步骤。

**3.3 Jupyter Lab加载并运行CNN**

如何连线并启动板子的方法请参考笔者的Github仓库。从左到右的三条线为type-c串口、网线、电源线。左上角的炫光处为SD卡插槽，烧录好镜像的SD卡接触面向下插入其中。电源线旁边的蓝色按钮为开关。最下面的红色开关拨到全0，代表使用SD卡中的镜像启动。

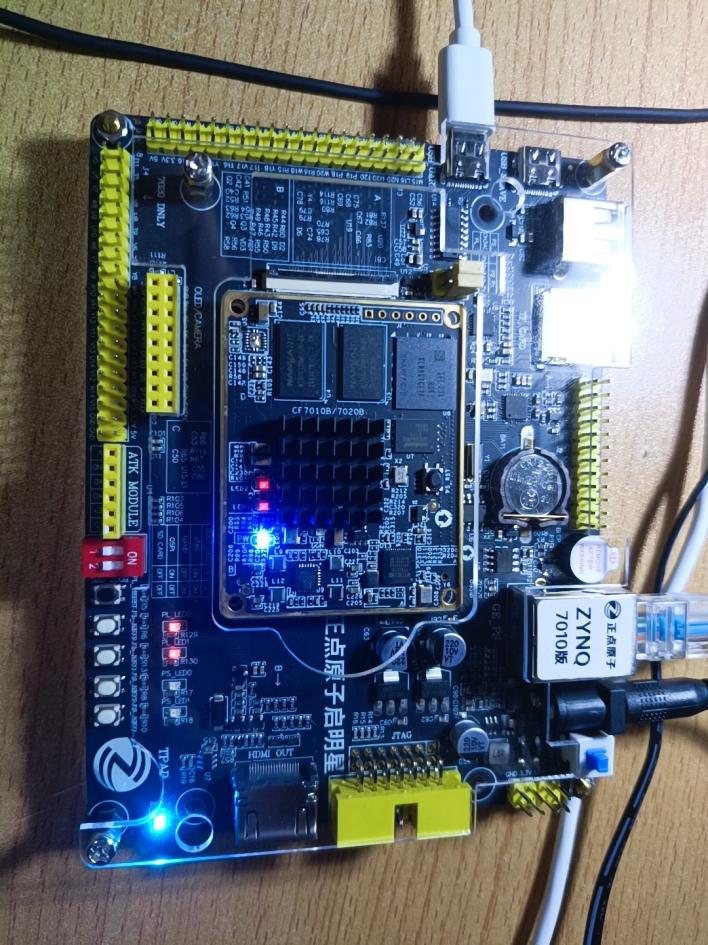


图7 工作中的PYNQ板子

相关.ipynb文件可在笔者Github仓库中找到。打开之后按顺序执行各个cell即可。

**四、设计架构**

在这一部分，我们分别展示我们使用Vivado完成的电路结构，以及在Jupyter Lab中完成的CNN网络。

电路结构如下所示。PYNQ单元使用AXI总线与两个IP核相联，同时存在一个用于产生50Hz信号与复位信号的单元，受PYNQ单元控制，复位信号用于控制其他单元的复位。

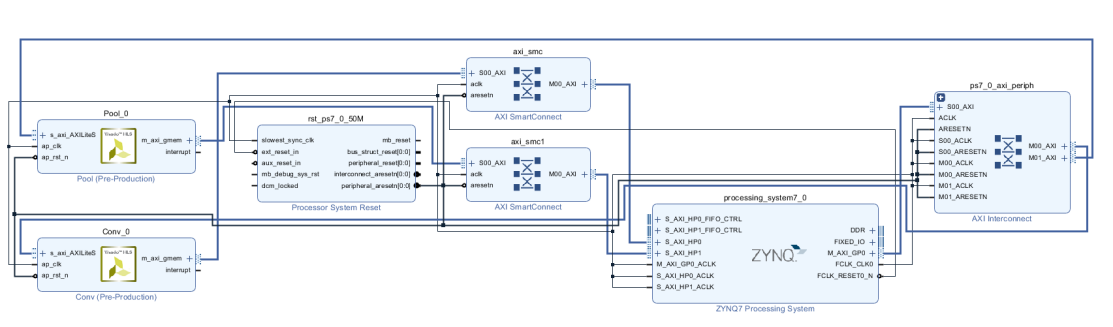


图8 Block Design概览

下面绘制我们使用的网络。实际上软件使用的是两个硬件资源，构成了一个虚拟的CNN网络。

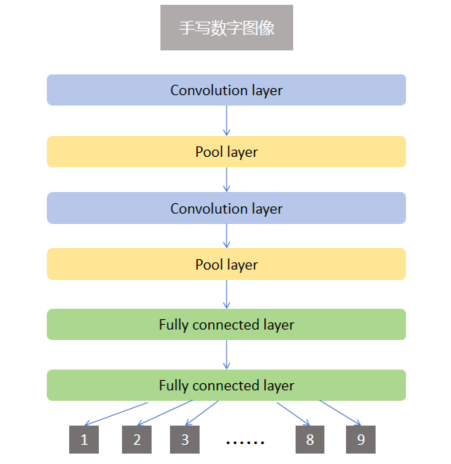


图9 展示CNN结构的DAG

**五、实验结果**

我们基于经典的MNIST数据集，验证我们用于手写数字识别的CNN网络的正确性。基于该系统，我们还通过实验验证了系统的精度与加速比。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 类型 | 平均推理时间 | 加速比 | 精度 | 能耗 |
| 硬件（PYNQ协同） | 1.706139s | 410.53 | 96.938% | 1.602W |
| 软件（纯CPU） | 700.424425s |

表2 实验结果

可以看到，在把卷积、池化、全连接运算卸载到硬件完成之后，可以实现420+的加速比，精度为96%以上。

如何测量能效。在Vivado-Synthesis-Open Synthesised Design-Report Power里可以设置对应的环境温度等参数，即可自动生成预测能耗数值。

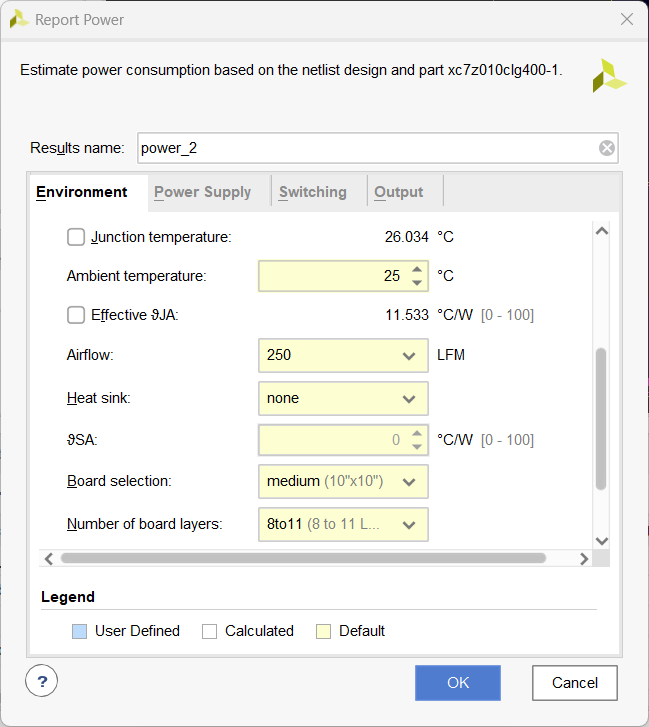


图10 能效参数设置

在完成设置之后，自动生成能效数值。

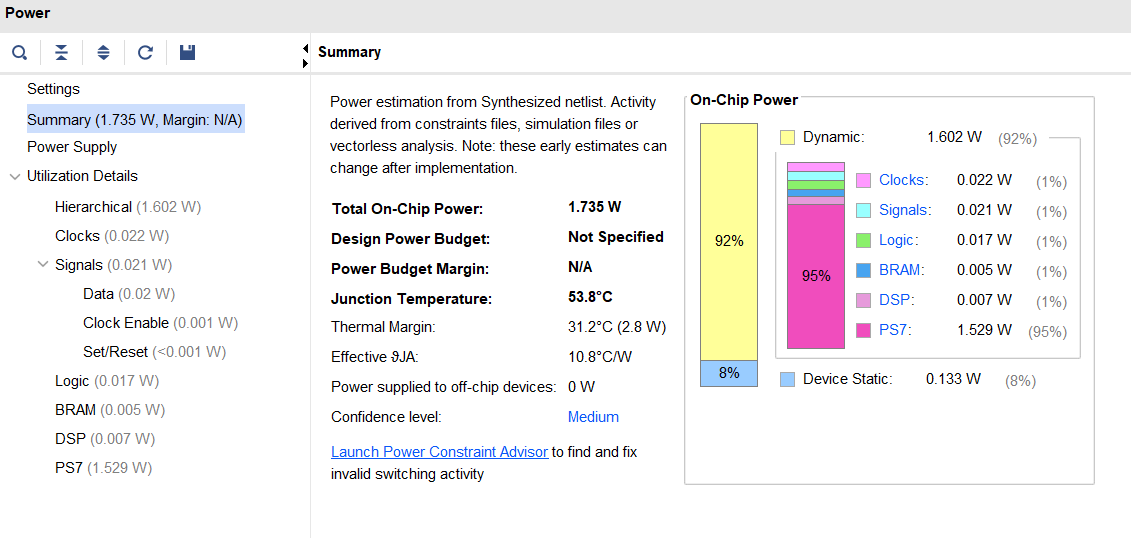


图11 对能耗的仿真预测

如何测量加速比。分别随机测量几次推理所需要的时间，以平均时间作为二者推理性能的指标，具体测量方法是基于time这一Python库，在推理前后分别测量一次时间，用二者的差值代表推理所消耗的时间。代码片段与结果如下所示。

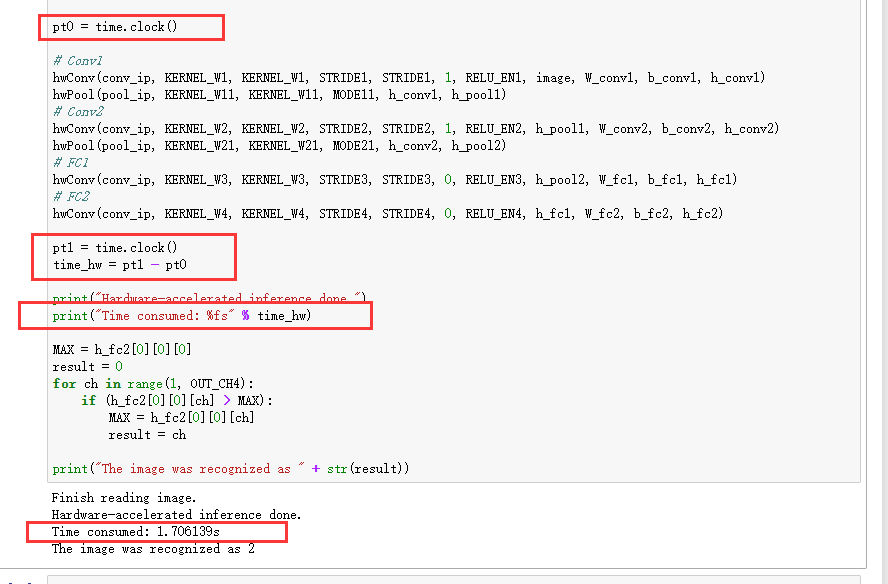


图12 测量加速比的方法

如何测量精度。由于软硬件实际上使用的是相同的权重，我们可以获知软硬件执行CNN网络的精度不会存在差别。我们使用MNIST数据集中的验证集。验证集大小为10000。考虑推理时间为1.7s，如果要验证所有验证集，需要的时间为约为17000s，大约为4.7小时，因此为了节省时间，我们只完成了一部分，共计5453个验证集数据的测量。

我们测量的方法如下所示。对于MNIST数据集中的每一个数据，每测量一个，测量数加一，如果测量正确，则测量正确数据数加一，二者的商就是精度。

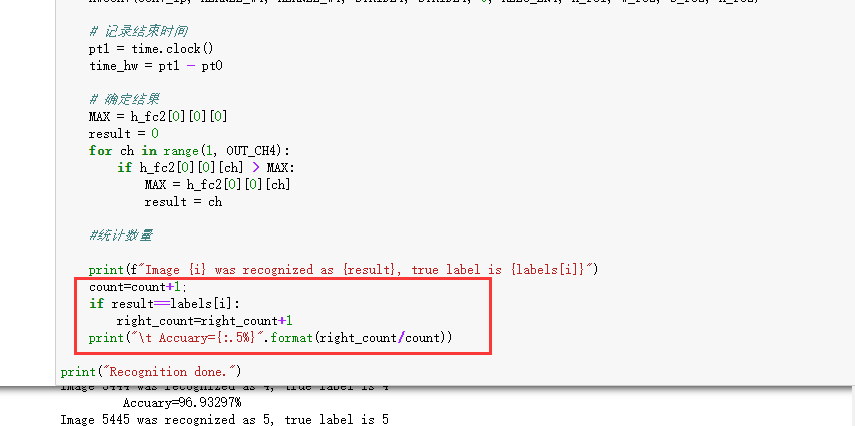


图13 测量CNN网络的精度

测量的结果如下所示。在测量了5453个测试数据之后，获得的精度为96.938%。

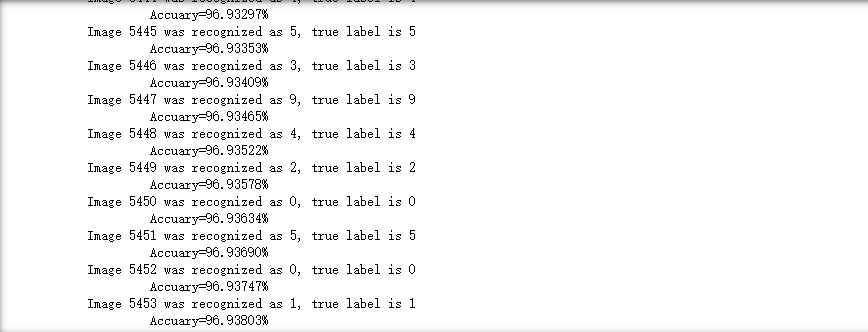


图14 在测试集上的识别精度

**六、总结**

历史车轮滚滚向前，新的技术不断涌现。对于我们计算机体系结构设计者来说，永恒不变的问题就是如何“trade off”，用最小的代价换取最高的收益，是我们的最高追求。

在陈云霁研究员所著的《智能计算系统》中，有这样一幅图，展示了当前的不同类型处理器的特征。通用性与能效存在着天然的内在矛盾，不存在一种既能能效高又能通用性好的处理器。深度学习加速器的设计理念，就是在X轴上向原点走一点，从而能在Y轴上向正方向多走一点点。

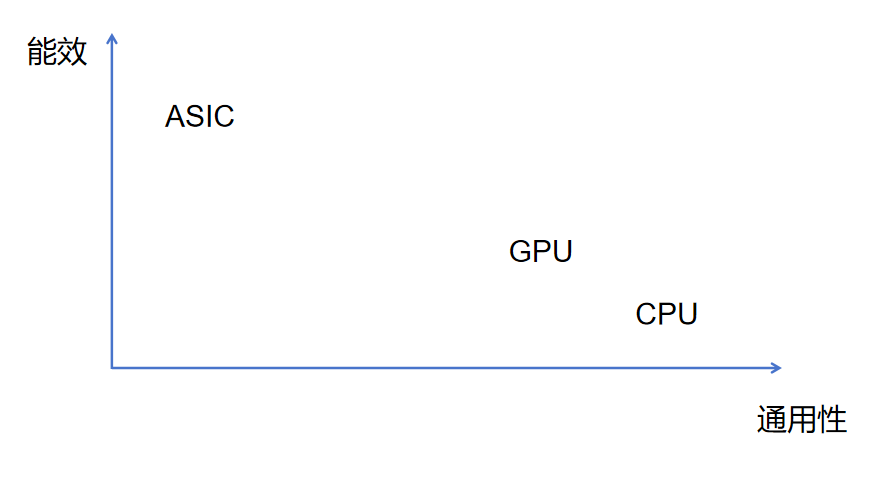


图15 不同类型处理器的能效及通用性

其实在我们的设计中，由于针对的仅仅是CNN网络这一场景，算子也仅仅只有池化、卷积、全连接几种，因此通用性是很差的。或者我们可以说，实际上我们在本次实验中设计的架构不能算是一款“深度学习加速器”，而是“卷积神经网络加速器”。

在我们预想的深度学习加速器中，算子是更加多样的，例如，在RNN、transformer等网络中还需要Recurrent Layer、Normalization Layer等层，也就需要循环算子与归一化算子。这也就要求我们在上面的曲线中稍稍向X轴正方向再走一点点，从而换取多一点点的通用性。

在讲了设计理念之后，我们可以再多谈一谈深度学习加速器的设计完整性。我们知道，领域专用处理器（一般称为device，设备端）一般不能脱离通用处理器（一般称为host，主机端）而存在，二者共同构成一个完整的异构计算系统。这也就引出了主机端如何控制设备端行为，以执行不同的算子操作。在我们的CNN加速器中，实际上使用的是“硬连线方案”，优点是更加特化、结构清晰明了，缺点是不能支持新的神经网络层。在一般的、不商业化的处理器系统中，这样简明的方案是有益的——“less is more”。如果追求可扩展性而放弃能效，反而违背我们trade off的哲学理念。

与之相对的是“指令集方案”，也就是仿照CPU的架构，为我们的处理器也设计一套ISA，利用不同的指令控制处理器的行为。实际上GPU正是这样的架构。GPU的CUDA扩展了C++语言，程序员可以通过编写CUDA程序，并经由MVCC翻译为GPU的指令，控制GPU完成计算任务。

本文的源代码、比特流文件、Jupyter文件、MNIST数据集等均可以在Github仓库中找到。请参考报告首页链接。