用于隐私保护人工智能的密文矩阵-向量乘法加速器

任轩乐

阿里巴巴达摩院计算技术实验室

摘要

人工智能算法和数据的隐私保护已经成为一个迫在眉睫的问题。在众多解决方案中，同态加密可以有效地保护人工智能模型和数据的隐私，因为它允许在不解密的情况下对加密数据进行人工智能计算。矩阵向量乘法是人工智能算法最为核心的算子，然而对密文数据的矩阵向量乘法会遇到数据量增大和计算量爆炸的问题，计算性能较差。已有工作已经提出使用 GPU、FPGA 和 ASIC 来加速密文矩阵向量乘法。然而，这些工作往往只针对特定的同态加密方案，未能考虑快速迭代发展的算法。例如，结合不同的同态加密方案的算法已经被证明能够支持更多类型的算子和密文。此外，一些现有的硬件加速器可以加速小型的同态加密操作（如数论变换和密钥交换），但对端到端应用的性能加速及其有限。为了更好地支持隐私保护人工智能应用（例如，神经网络推理），本文提出了一个可用于高性能的密文矩阵-向量乘法的硬件加速器设计。该加速器不仅支持传统的同态加密操作，还支持不同类型的密文以及它们之间的转换。我们在 Xilinx FPGA 平台上实现了该加速器。评估表明，与已有工作相比，密文的矩阵-向量乘积速度提高了 1800 倍。

1 引言

在这个大数据时代，数据正成为越来越多应用的燃料。然而，数据的获取并不容易，共享数据甚至对一些机构来说是禁止的，如医疗系统、银行和政府。隐私计算旨在通过只共享加密数据来克服这一限制，从而避免敏感信息的泄露。为了实现这一目标，学术界已经开发了一些协议，如安全多方计算 [26] 和联邦学习 [14]，它们都能让多方在交换最少量数据的基础上，协同学习一个共享的预测模型。同态加密（Homomorphic Encryption），由于其具有在加密数据上进行计算的能力，可以被用于在各方之间交换数据。

同态加密的缺点是性能太差，因此距离大规模应用还有很远的距离。大多数主流的同态加密方案都受到密文大小和计算量增大的困扰（扩大 100 到 200 倍）。例如，加密神经网络的推理需要

250 秒 [9,15]。这个问题可以通过批量并行处理来缓解 [2,7,10]。例如，4096 张加密图像可以同时被批量处理，从而分摊单个图像的存储成本和计算成本 [2]。同态加密计算任意深度计算的性能变得更差。更准确地说，在加密后，一个密文与一个噪声预算相关联，该噪声预算将在计算过程中消耗。为了确保正确解密，这个噪声预算在计算过程中不应该被耗尽，这意味着只允许有限深度的计算。自举（bootstrapping）通过在噪声预算即将耗尽之前刷新噪声预算来解决这个问题 [13]。然而，自举会引入大量的额外计算，并且这些计算会占据整个计算的主导地位 [30]。

由于同态加密的性能瓶颈，FPGA、GPU 和 ASIC 等硬件被用于加速同态加密，特别是用于加速自举。基于 FPGA 的方案主要针对同态加密运算符，如多项式乘法和数论变换（Number Theoretic Transform） [22,23,25,29,31]。其中，一个代表性的工作是通过在 FPGA 上实现流水线架构来加速密钥交换 [29]。然而，从上层应用的角度来看，仅仅加速密钥交换提供了有限的性能提升。与 CPU 相比，GPU 显示出几十倍的加速 [1,8,17,18]。然而，GPU 的瓶颈在于有限的共享内存无法容纳大型多项式和同态加密计算过程中的中间结果。ASICs 取得了巨大的性能提升，得益于高频时钟、高带宽内存和密集计算逻辑的使用 [11,12,20,21,30]。然而，这些 ASICs 的芯片面积往往非常大，成本非常高。

同态加密的性能问题难以解决，与此同时，我们注意到同态加密的算法正在发生新的演变。与使用自举相比，更实用的解决方案是通过结合同态加密和其他技术来避免深度同态评估。例如，使用同态加密和混淆电路（Garbled Circuit）可以实现加密神经网络的推理 [19]。具体来说，只有线性层是采用同态加密评估的，而非线性层则由混淆电路处理。由于线性运算通常非常浅，所以它们所需的加密参数比支持自举的情况要小得多。使用这种协议，ResNet-20 的推理比 HE-only 解决方案快 1000 倍 [2]。如果进一步地，使用 ASICs 加速，可以获得更多性能提升 [28]。

我们又注意到，除了自举带来巨量开销的缺点外，同态加密还有一个缺点，就是不能有效地支持不同类型的函数。特别地，常用的同态加密方案（即 B/FV 和 CKKS）不能有效地支持非线性函数（例如神经网络中的激活函数）。一种解决方案是使用高阶多项式来近似这些函数 [15]，但这会导致已经训练好的模型被修改甚至模型准确度下降，影响模型的推理效果。例如，我们观察到 LeNet-5 已经被近似为多项式版本 [4,9,15]，其中一些表现出明显的推理准确度下降。为了解决这个问题，提出了两组新颖的算法。首先，使用多种类型的密文并可以相互转换可以提高同态加密计算的效率 [6]。其次，不同的同态加密方案（即 B/FV、CKKS 和 TFHE）可以组成一个混合方案，有效地支持线性和非线性函数而不引入近似误差 [3,24]。然而，这些新的同态加密算法尚未在 FPGA 或 ASIC 上实现。

在这项工作中，我们提出了一种新型的同态加密硬件加速器，用于加速高性能密文矩阵-向量乘法（homomorphic matrix-vector product, 密文矩阵向量乘），该设计采用算法-架构协同设计的思想。首先，基于观察到同态加密适用于并广泛用于计算线性函数，加速器采用针对密文矩阵向量乘法的新颖算法。特别地，该加速器支持不同类型的密文（即 RLWE 和 LWE）以及它们之间的转换，比传统算法具有更大的灵活性。其次，该加速器采用定制的、全流水线架构，能够有效利用硬件并行性。对于 NTT，我们提出了一种新颖且高效的架构，能够实现流水线数据流不产生任何气泡。同态加密已经部署在顶级云服务提供商之一的隐私保护机器解决方案中。据我们所知，加速器是第一个部署于商业应用的同态加密加速器。基准测试结果表明，与现有工作相比，密文矩阵向量乘 速度提高了 1800 倍。

2 算法及相关工作

本节介绍一些基本符号和与同态加密相关的术语。在加密成密文之前，原始数据需要被编码成明文。原始数据可能以不同的格式和大小（如标量、向量、矩阵或张量）出现，具有固定点或浮点精度。我们将原始数据进行编码后，则称为明文（plaintext）。进一步地，一些数据应该通过加密进行保护，其明文随后被加密为所谓的密文（ciphertext）。RLWE 是同态加密方案中常用的加密方案（例如 B/FV 和 CKKS），这些同态加密方案提供了单指令多数据（single instruction multiple data）的能力，可以使用一个密文（称为批量编码）对大量的明文槽进行相同的计算。一个批量编码器可以实现为系数式或槽式风格。编码方法的选择很重要，因为它决定了后续的计算流程。

在加速器中，明文被编码为多项式，密文被加密为两个多项式，写作 $(b(X), a(X))$。在本文中，将编码明文 a 的明文表示为 $pt(a)$，类似地，将加密 a 的密文表示为 $ct(a)$。

2.1 基本算法

同态矩阵-向量乘积（HMVP） – 在诸如逻辑回归 [14] 等应用中，计算主要由多个矩阵-向量乘组成。在隐私保护应用中，矩阵-向量乘是同态计算的，即密文的矩阵-向量乘积。在这种情况下，矩阵 A 被编码为一组明文 $pt(A_i)$，向量 v 被加密为一个密文 $ct(v)$。计算密文矩阵向量乘后，$u = A \cdot v$ 被加密在一个新的密文 $ct(u)$ 中。注意，这里的明文 $pt(\cdot)$ 和密文 $ct(\cdot)$ 都是有一系列多项式组成的，我们将它们的系数保存为一个向量。

通过多项式乘法进行点积 – 首先，分别编码矩阵的每行（第 i 行 A_i ）为 $pt(A_i)$ 和向量 v 的明文为 $ct(v)$；然后，$pt(A_i)$ 和 $ct(v)$ 的同态乘积是一个密文，其明文为其常数系数实际上是向量 A_i 和 v 的内积。在计算所有 $pt(A_i)$ 和 $ct(v)$ 的同态乘积后，我们得到一组密文ct_i，它们在明文的常数系数中加密了内积 $A_i \cdot v$。

打包加密标量 – 对于一个 m 行矩阵 A，在密文矩阵向量乘之后，我们得到 m 个密文而不是一个密文。只有明文的常数系数是有用的。因此，在加速器中需要从 m 个密文中解压常数系数并将它们重新打包到一个密文中。为了解压和重新打包，我们采用了 Hao Chenet al. 提出的方法[6]。一般来说，我们提取 m 个明文中所有的常数系数，然后将它们转换为新的密文，最终将它们打包到一个结果密文中。PackTwoLWEs 过程用于打包两个密文，然后在 PackLWEs 中，多次反复调用 PackTwoLWEs 以打包多个密文。实际上，要打包两个密文$ct(i)$，一个 PackTwoLWEs 过程包括乘以一个单项式、减法、加法、自同构、密钥交换。然而在此之前，其输入密文需要先经过一个 ExtractLWEs 过程进行预处理。综上所述，我们可以看出来，密文矩阵向量乘和明文矩阵向量乘非常不同。明文矩阵向量乘法非常直观，直接将矩阵的每个行向量和向量进行内积计算即可；而密文矩阵向量乘则需要进行多项式乘法，涉及到了多项式加减法、自同构、密钥交换等一系列计算。

算法 1：系数编码的密文矩阵-向量乘法

输入数据：矩阵 A $(m \times n)$ 和向量 v $(n \times 1)$

输出数据： 向量 $u = A \cdot v$

1: $v \leftarrow$ ENCRYPT(COEFF_ENCODE(v)) ▷ 编码并加密

2: **for** $i \leftarrow 1$ **to** m **do**

3: $pt_i \leftarrow$ REV_COEFF_ENCODE($A[i,:]$)

4: $c_i \leftarrow pt_i \times v$ ▷ 点乘

5: $l_i \leftarrow$ RLWE-TO-LWE(c_i) ▷ 将 RLWE 密文转换为 LWE 密文

6: **end for**

7: $d \leftarrow$ PACKLWES($l_1, \ldots, l_m$) ▷ 将 LWEs 密文打包成一个 RLWE 密文

8: $u \leftarrow$ DECRYPT(COEFF_DECODE(d) ▷ 解码并解密

2.2 相关工作

除了直接将明文编码为明文的系数（即系数编码），我们还可以使用单指令多数据（SIMD）方法对明文进行编码，该方法可以使用一个密文对大量槽执行相同的计算（称为批处理编码）。在许多密文矩阵向量乘框架中 [19]，明文矩阵 A 被编码为批处理，可以直接旋转并求和以获得结果密文。与批处理编码的密文矩阵向量乘 [19] 相比，我们的系数编码密文矩阵向量乘将计算复杂度从 $O(m \cdot \log N)$ 降低到 $O(m)$。与 [19] 中提出的复杂度也为 $O(m)$ 的对角线编码方法相比，我们采用的算法仍然具有优势，因为系数编码引入了更小的开销。此外，我们采用的算法还可以通过类似方式对张量进行不同形式的编码，从而扩展到其他线性函数，例如二维和三维卷积 [16]。

2.3 安全模型

在这项工作中，我们采用了广泛应用于安全多方计算和联邦学习的两方计算模型。更准确地说，A 方拥有一个向量的一部分，而 B 方拥有该向量的另一部分以及一个矩阵。A 加密他的向量份额并将其发送给 B。B 然后组合两个向量份额并将其乘以矩阵。我们假设 B 是一个半诚实的对手。换句话说，虽然 B 试图从 A 提供的数据中尽可能多地了解，但 B 诚实地遵循规定的协议。

3 架构设计

在本节中，我们将描述加速器的架构以及如何通过搜索设计空间来实现它。

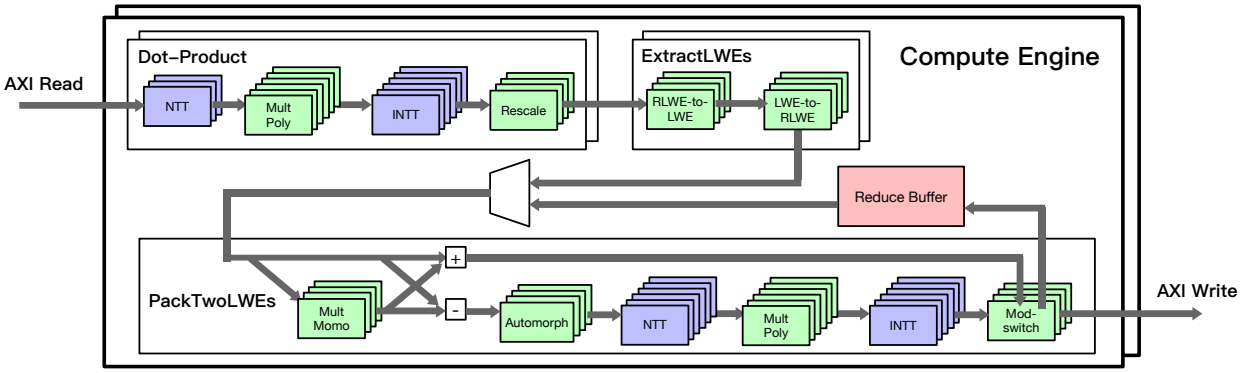


图 1：加速器架构设计

3.1 架构概述

加速器旨在加速算法 1 中描述的系数编码矩阵-向量乘。如图 1 所示，加速器由多个计算引擎组成，每个引擎都采用全流水线架构来提高资源利用率和最大化吞吐量。明文和密文中的所有多项式都并行处理，每个对应一个功能单元（例如 NTT、MultPoly 和 INTT）。

DotProduct 模块接受明文和密文作为输入。输入多项式从系数域转换为 NTT 域（第 1 阶段），以便多项式乘法可以从卷积简化为系数逐项乘法（第 2 阶段）。结果然后转换回系数域（第 3 阶段）。第 4 阶段通过将密文除以 39 特殊模数将增大的密文重新缩放为正常格式。这个阶段的目的是减少多项式乘法引入的噪声（从 30 比特 减少到 26 比特）。ExtractLWEs 模块只是从 RLWE 密文中提取点积结果并将其保存为 LWE 密文。它与 Rescale 单元位于同一阶段（第 4 阶段），因为它们的执行都是在多项式上逐系数进行的，因此容易结合起来。

PackTwoLWEs 模块将点积结果（作为 LWE 密文出现）打包到单个 RLWE 密文中（第 5-9 阶段）。注意，这个打包过程可以描述为一棵二叉树，它从叶节点接收输入并在根节点输出结果。由于 PackTwoLWEs 模块每次将两个密文压缩到一个，因此需要 4095 次压缩才能打包 4096 个密文。中间结果被存储在压缩缓冲区中。一旦中间的压缩结果准备好进行下一级压缩，它们就会抢占流水线并暂停前面阶段的执行。

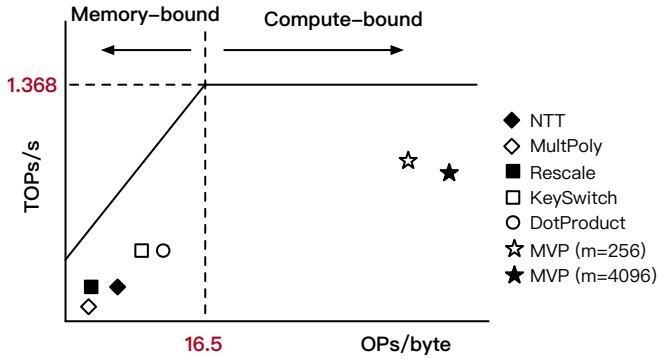


图 2：屋顶线（Roofline）模型

根据在图 2 中评估的 Xilinx U200 FPGA 的屋顶线（Roofline）模型，我们清楚地观察到同态加密操作（如 NTT 和密钥交换）的计算强度远小于密文矩阵向量乘。单独调用这些同态加密操作将导致密集的内存访问，从而降低整体性能。因此，为了获得最佳性能，加速器采用了全定制架构来加速整个密文矩阵向量乘，而不是单独的同态加密操作（如 NTT 和密钥交换），这也间接证明了我们设计架构的合理性。

我们探索实现密文矩阵向量乘的设计空间，包括 1）如何分割流水线，2）如何选择计算引擎、模块和功能单元（FU）的数量，3）如何确定每个 FU 的并行度，以及 4）缓冲区的大小。我们在图 3 中绘制了结果，并按其性能和资源利用率定位每个设计选择。最佳选择位于两个点上，即（9 个阶段、 1 个 PackTwoLWEs、6 个 NTT、4-PE NTT、2 个计算引擎）和（9 个阶段、1 个 PackTwoLWEs、6 个 NTT、8-PE NTT、1 个计算引擎）。我们采用了第一个点。注意，在图 3 的每个设计选择中，我们必须确保所有流水线阶段具有相似的延迟和吞吐量，以最大化整体性能。例如，如果阶段 A 包含比阶段 B 多 k 倍的计算，则阶段 A 的并行度应是阶段 B 的 k 倍（即 $P_A = k \cdot P_B$）。

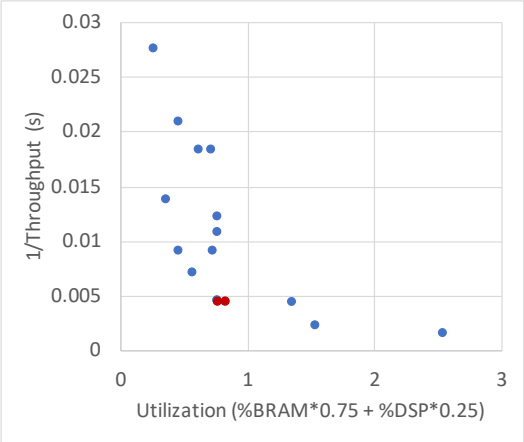


图 3：FPGA 资源利用率和性能的取舍

3.2 数论变换模块设计

NTT 是离散傅里叶变换（DFT）在有限域上的推广。算法 2 显示了一个恒定数据通路的前向 NTT 算法 [5,27]。外循环将 NTT 的计算分为 $\log_2 N$ 个阶段，而每个阶段包含 $N/2$ 个独立的蝴蝶操作。

算法 2：恒定数据通路的前向 NTT

输入数据：多项式 $a(X)$，旋转因子 ω_{2N}

输出数据：多项式 $b(X) = NTT(a(X))$

1: for $i \leftarrow 0$ to $\log_2 N$ do ▷ i 是循环阶段编号

2: for $j \leftarrow 0$ to $N/2$ do ▷ j 是蝴蝶因子编号

3: $\omega_{ij} = \omega[i \cdot \frac{N}{2} + j]$ ▷ 提取旋转因子

4: $b(X)_{2j} = a(X)_j + a(X)_{j+N/2} \cdot \omega_{ij}$ ▷ 蝴蝶因子计算

5: $b(X)_{2j+1} = a(X)_j - a(X)_{j+N/2} \cdot \omega_{ij}$ ▷ 蝴蝶因子计算

6: **end for**

7: **if** $i \neq \log N - 1$ **then**

8: $a(X) = b(X)$

9: **end if**

10: **end for**

蝴蝶因子并行度（n_{bf}）影响性能和硬件利用率。一方面，每个 NTT 阶段的 $N/2$ 个蝴蝶操作可以用 $n_{bf} = 2^s$ 个蝴蝶单元（BFU）并行化。另一方面，尽管较大的n_{bf} 表示更高的并行度，但同时也耗尽了片上的 RAM。片上存储器深度限制了n_{bf} 的上限，因为一个多项式需要存储在 n_{bf} 个 RAM bank 中以支持高并行度。

以前基于 ASIC 的 NTT 解决方案需要$(\sqrt{N} \times \sqrt{N})$个元素存储块 [11]，这对于 FPGA 设计来说是不可行的。另一方面，面向 FPGA 的设计采用 block RAM 的优化 [29]，需要大量基于查找表的多路复用器来处理阶段变量的存储器访问模式。在我们的工作中，提出了以下优化来解决以前工作中的这些缺陷。

A. 并行数据流

在我们的设计中，一个多项式存储在 8 个循环（round-robin）RAM bank 中。NTT 以乒乓方式执行。在偶数（2r）阶段，多项式系数从 RAM-0 读取并写入 RAM-1，然后在奇数（2r+1）阶段，系数从 RAM-1 读取并写入 RAM-0。因此，NTT 过程需要 $(\frac{N}{2} \cdot \log N)/n_{bf}$ 个时钟周期，这里的 n_{bf}代表了并行度。在这项工作中，我们设置 $n_{bf} = 4$，因此所有一读一写 RAM bank 可以并行处理。一个多项式的连续系数存储在 RAM bank 中以便它们可以同时读写，系数按上下顺序读取，经过交换单元（SWAP）的重新排列，再写回 RAM bank。这种读写方式确保了 NTT 单元和 RAM bank 之间的固定数据路径。

B. 旋转因子的生成和存储

NTT 操作涉及总共 N − 1 个旋转因子，如图 4 所示（例如 N = 32）。加速器旨在为每个 BFU 分配一个单独的 ROM bank 来存储相应的旋转因子。为此，一列中的四个旋转因子被分配给四个 BFU，并在同一个时钟周期内使用（例如，在 stage-2 中的索引 4 到 7）。旋转因子的大小等于多项式的大小（即 N）。此外，多个 NTT 单元可以共享同一份旋转因子的副本，这样一个计算引擎只需要两组旋转因子，一组用于 NTT，另一组用于 INTT。

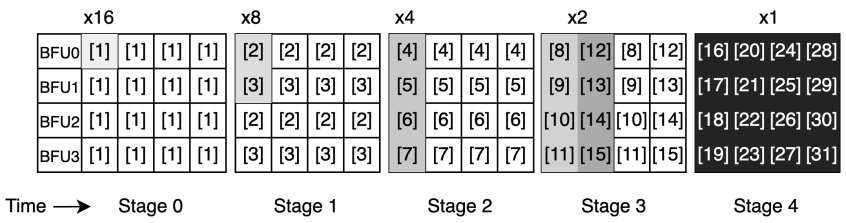


图 4：旋转因子的存储及使用的顺序

C. 模数约简的定制优化

模数乘法是同态加密中最重要但最复杂的操作。如果有限域定义在具有低汉明权重的模数上，则模数算术可以显著简化。我们选取每个模数只有三个非零位，因此它们的乘法可以简化为三次简单的移位和加法。

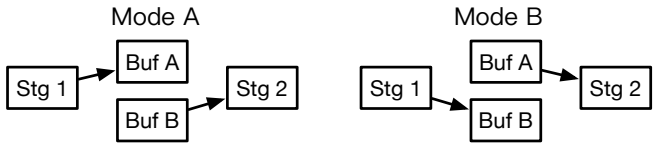
3.3 多项式处理单元设计

除了 NTT 以外，大多数函数，包括 MultPoly、Rescale、MultMono、Automorph 和 ModSwitch，都基于多项式计算。因此，我们设计了多项式处理单元来支持这些函数。在加速器中，多项式处理单元还需要支持不同格式密文件的转换，即 RLWE-to-LWE 和 LWE-to-RLWE，这涉及多项式和向量之间的转换。从实现的角度来看，多项式的系数存储在类似向量的数据结构中，并且所有多项式运算都以向量化方式进行。由于这个原因，LWE 密文（由一个向量和一个标量组成）和 RLWE 密文（由多项式组成）都可以通过统一的数据结构很好地支持多项式和向量。

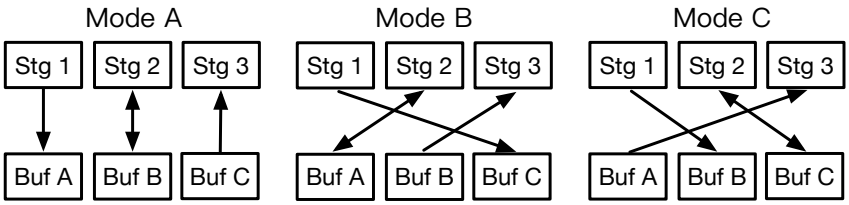
在设计方面，我们首先实现了系数逐位模加（ModAdd）和乘法（ModMul）计算，Rev 函数旨在反转多项式系数的顺序。ShiftNeg 作为 MultMono、RLWE-to-LWE 和 LWE-to-RLWE 的底层函数，实现为循环移位后对包装过的系数取反。Automorph 实现为系数的置换。

3.4 乒乓存储器设计

加速器使用乒乓缓冲区进行相邻流水线阶段之间的数据通信。图 5α 显示了双乒乓缓冲区的工作原理。具体来说，在第 T 个时间槽，阶段 1 写入缓冲区 A，阶段 2 从缓冲区 B 读取（对应于模式 A）；然后在第（T +1）个时间槽，阶段 1 写入缓冲区 B，阶段 2 从缓冲区 A 读取（对应于模式 B）。如果一个流水线阶段需要在同一个时间槽内读写一个缓冲区，那么就使用三重乒乓缓冲区，如图 6b 所示。通过使用乒乓缓冲区，流水线阶段可以在收到开始信号后立即开始执行，而不需要任何数据拷贝。此外，以乒乓方式使用缓冲区也是高效的，因为当流水线阶段忙时，所有的乒乓缓冲区都保持很高的使用率。



(a)



(b)

图 5：乒乓存储器的访问模式：(a) 两乒乓，(b) 三乒乓

3.5 异构系统设计

从系统设计的角度来看，加速器 IP 作为一个加速器安装在 AXI（高级可扩展接口）总线上，FPGA 通过 PCIe 与主 CPU 连接，如图 6 所示。当 CPU 程序需要评估同态 MVP 时，它会检查 FPGA 是否空闲以及是否支持矩阵大小。一旦检查通过，主 CPU 就会通过 PCIe 将数据发送到 FPGA 板上的 DRAM，配置加速器 IP，并开始执行。一旦执行完成，加速器会向主 CPU 发送一个中断。 为了最大化加速器上计算单元的利用率，我们在 FPGA 和 CPU 之间交错计算和数据传输。这个过程由两部分组成：主机端和 FPGA 端。在主机端，我们使用多个线程来对数据传输和计算进行流水线处理。在 FPGA 端，我们使用 RAMs 来缓冲每个线程对应的输入和输出数据。

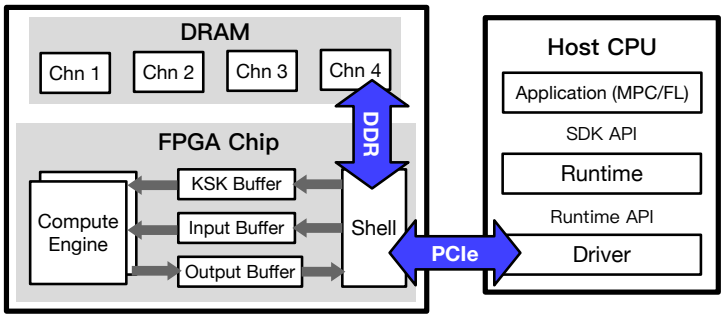


图 6：FPGA+CPU 异构计算系统

4 实验评估

在本节中，我们在 Xilinx FPGA 上实现了加速器，并根据各种基准测试评估了其性能。

4.1 在 FPGA 平台的实现

我们在 Xilinx U200 FPGA 板和 Intel Xeon W-2265 CPU@3.5GHz 上实现了加速器。特别地，我们使用 Xilinx Vitis RTL 流程连接加速器和主机 CPU。加速器以 300MHz 的频率实现。在实现过程中，我们发现 ViVado 工具使用了大量的 block RAM，给布局和布线带来压力。为了解决这个问题，我们用 URAM 和 LUTRAM 替换了一些 block RAM，使它们所有的利用率都低于 75%。最终资源利用率如表 1 所示。

表 1： Xilinx VU9P FPGA 的资源利用率

Module	LUT	FF	BRAM	URAM	DSP
Compute Engine 0	259,318	89,894	640	294	986
Compute Engine 1	259,502	90,043	640	294	986
Platform	234,066	302,670	278	7	14
Total	63.68%	20.41%	72.13%	61.98%	29.04%

4.2 基准测试评估

NTT – 我们基于 Xilinx VU9P FPGA 评估 NTT 和 INTT 的延迟、资源利用率和效率，结果如表 2 所示。该表显示了旋转因子 ROM 和 NTT 本地缓冲区的三种不同实现策略，因为 Xilinx 平台允许灵活使用块 RAM（BRAM）或基于 LUT 的分布式 RAM（dRAM）。我们将我们的实现与现有的工作进行比较，即 HEAX [29] 和 F1 [11]。HEAX 的 NTT 设计与我们的消耗相同的时钟周期。然而，由于我们使用了对硬件友好的模数和并行的恒定数据流设计，我们的设计更紧凑。此外，由于加速器总共有 60 个 NTT 单元，每秒可以执行 195 操作（ops/sec），而 HEAX 每秒只能执行 117 操作（假设 $N = 2^{12}$），因此加速器实现了高吞吐量。一个基于 GPU 的工作使用 1024 个线程的单个 CUDA 内核实现了 NTT 吞吐量为 45 ops/sec，但这比我们的实现慢得多。最后，F1 在 ASICs 上实现，显示出巨大的性能优势。然而，如果在 FPGA 平台上评估，其 NTT 设计将消耗超过 65% 的 DSP 单元。除了 NTT，我们还评估了密钥交换操作的性能。加速器实现了每秒 65 个操作的吞吐量，比 CPU 基线高 105 倍。

表 2：NTT 性能对比

加速器	时钟周期(l)	乘法器个数(p)	面积延时积($l \times p$)	查找表(u)	BRAM	面积延时积($l \times u$)
本文加速器 (BRAM+dRAM)	6144	4	1×	6508	6	1.96×
本文加速器 (dRAM only)	6144	4	1×	9248	0	2.78×

HEAX [29]	6144	4	1×	22316	11	6.71×
F1 [11]	202	896	7.36×	-	-	-

矩阵-向量乘积 – 我们基于加速器评估密文矩阵向量乘并测量吞吐量。吞吐量几乎线性地依赖于矩阵的行数（$\boldsymbol{m}$）。列数（$\boldsymbol{n}$）对性能的影响较小，除了 $\boldsymbol{n} \geq \boldsymbol{m}$ 的情况，此时吞吐量降低，因为一个行位于多个密文中，需要进行聚合。我们将加速器与 CPU 基线进行比较，结果如表 3 所示。我们观察到超过 90% 的计算已经卸载到 FPGA 上，导致大于 10 倍的加速。我们还观察到，具有更多行的矩阵显示出更高的性能增益。接下来，我们将加速器与 GPU 实现进行比较。特别地，加速器比 GPU 显示出更小的延迟（0.3 倍 ~ 0.7 倍）和更高的吞吐量（4.5 倍）。

表 3：密文矩阵-向量乘的性能对比（矩阵列数 = 256）

矩阵行数	CPU 运行时间	GPU 运行时间	本文提出的加速器运行时间
64	45.57 ms	13.25 ms	2.28 ms
128	62.36 ms	15.95 ms	3.57 ms
256	100.45 ms	25.95 ms	6.17 ms
512	174.45 ms	47.01 ms	11.4 ms
1024	352.06 ms	92.79 ms	21.88 ms
2048	639.78 ms	182.59 ms	42.86 ms

逻辑回归 – 最后我们评估异构逻辑回归（HeteroLR）的性能，其中数据在各方之间垂直划分 [14]。HeteroLR 根据两个方 A 和 B 提供的重叠样本训练联邦模型。在训练过程中，方 A 和 B 根据本地数据计算梯度，而仲裁者则聚合梯度并将更新后的梯度分发给它们。A 和 B 都需要计算密文矩阵向量乘。Federated AI Technology Enabler (FATE) 框架最初使用 Paillier 算法。在这项工作中，我们用 B/FV 替换了 Paillier 以更好地利用硬件加速能力。此外，如果结合小批量和矩阵切片技术，我们的算法能够支持任意规模的数据，并部署在多个硬件加速器上。

对不同大小的数据集进行了 HeteroLR 的评估，如图 7 所示。我们得出结论：B/FV 减少了所有步骤在 HeteroLR 中的计算开销，包括加密、向量加法、矩阵-向量乘积和解密。此外，由加速器加速的密文矩阵向量乘比其 CPU 基线快 30 倍至 1800 倍。相应地，端到端的 HeteroLR 最高加速

了 36 倍。涉及大型矩阵（例如 8192×4096 和 8192×8192）的情况观察到了更加明显的性能提升，因为对于这些情况，矩阵-向量乘积几乎占据了整个计算。

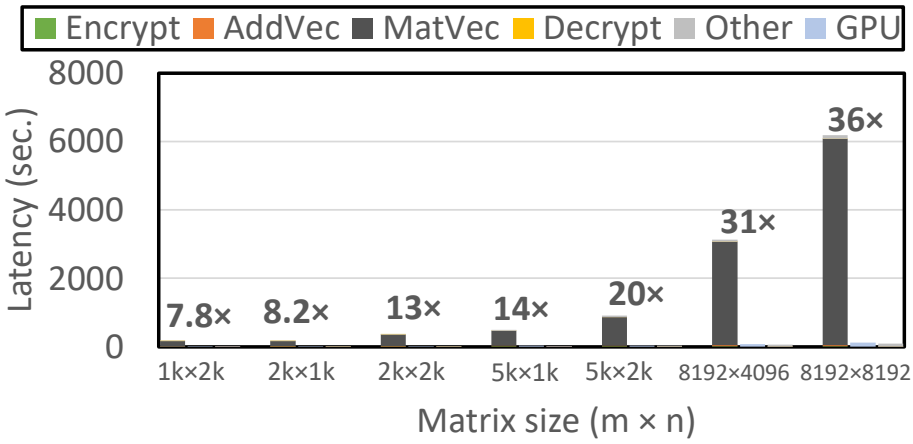


图 7：加速器对逻辑回归的加速（最大可以达到 36 倍）

5 结论

在这项工作中，我们设计了一个同态加密加速器，用于高性能的密文矩阵-向量乘计算。加速器可能是第一个部署于联邦学习和多方计算商业应用的同态加密加速器。与现有的同态加密加速器不同，加速器采用了算法-硬件协同设计的方法。实验结果表明，矩阵-向量乘速度提高了 1800 倍，逻辑回归速度提高了 36 倍。

参考文献

[1] P. G. M. Alves et al., “Faster Homomorphic Encryption over GPGPUs via Hierarchical DGT,” in IACR Cryptology ePrint Archive, 2021.

[2] F. Boemer et al., “nGraph-HE2: A High-throughput Framework for Neural Network Inference on Encrypted Data,” in ACM Conference on Computer and Communications Security, 2019.

[3] C. Boura et al., “CHIMERA: Combining Ring-LWE-based Fully Homomorphic Encryption Schemes,” Journal of Mathematical Cryptology, vol. 14, no. 1, 2020.

[4] A. Brutzkus et al., “Low Latency Privacy Preserving Inference,” in ICML, 2019.

[5] D. D. Chen et al., “High-Speed Polynomial Multiplication Architecture for Ring-LWE and SHE Cryptosystems,” IEEE Transactions on Circuits and Systems I, vol. 62-I, no. 1, 2015.

[6] H. Chen et al., "Efficient Homomorphic Conversion Between (Ring) LWE Ciphertexts," in IACR Cryptology ePrint Archive, 2021.

[7] J. H. Cheon et al., "Homomorphic Encryption for Arithmetic of Approximate Numbers," IACR Cryptology ePrint Archive, 2017.

[8] W. Dai et al., "cuHE: A Homomorphic Encryption Accelerator Library," in IACR Cryptology ePrint Archive, 2016.

[9] N. Dowlin et al., "CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy," in ICML, 2016.

[10] J. Fan et al., "Somewhat Practical Fully Homomorphic Encryption," in 15th International Conference on Practice and Theory in Public Key Cryptography, 2012.

[11] A. Feldmann et al., "F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption," in Annual International Symposium on Microarchitecture, 2021.

[12] R. Geelen et al., "BASALISC: Flexible Asynchronous Hardware Accelerator for Fully Homomorphic Encryption," in ACM Conference, 2022.

[13] C. Gentry, "A Fully Homomorphic Encryption Scheme," Ph.D. dissertation, 2009.

[14] S. Hardy et al., "Private Federated Learning on Vertically Partitioned Data via Entity Resolution and Additively Homomorphic Encryption," 2017. [Online]. Available: http://arxiv.org/abs/1711.10677

[15] E. Hesamifard et al., "CryptoDL: Deep Neural Networks over Encrypted Data," 2017. [Online]. Available: http://arxiv.org/abs/1711.05189

[16] Z. Huang et al., "Cheetah: Lean and Fast Secure Two-Party Deep Neural Network Inference," USENIX Security, 2022.

[17] W. Jung et al., "Over 100x Faster Bootstrapping in Fully Homomorphic Encryption through Memory-centric Optimization with GPUs," IACR Transactions on CHES, vol. 2021, no. 4, 2021.

[18] W. Jung et al., "Accelerating Fully Homomorphic Encryption through Architecturecentric Analysis and Optimization," IEEE Access, 2021.

[19] C. Juvekar et al., "GAZELLE: A Low Latency Framework for Secure Neural Network Inference," in 27th USENIX Security Symposium, 2018.

[20] J. Kim et al., "ARK: Fully Homomorphic Encryption Accelerator with Runtime Data Generation and Inter-Operation Key Reuse," 2022. [Online]. Available: http://arxiv.org/abs/2205.00922

[21] S. Kim et al., "BTS: An Accelerator for Bootstrappable Fully Homomorphic Encryption," in ISCA, 2022.

[22] S. Kim et al., "FPGA-based Accelerators of Fully Pipelined Modular Multipliers for Homomorphic Encryption," in International Conference on Reconfigurable Computing and FPGAs, ReConFig, 2019.

[23] S. Kim et al., "Hardware Architecture of a Number Theoretic Transform for a Bootstrappable RNS-based Homomorphic Encryption Scheme," in IEEE International Symposium on FCCM, 2020.

[24] W. Lu et al., "PEGASUS: Bridging Polynomial and Non-Polynomial Evaluations in Homomorphic Encryption," in IEEE Symposium on Security and Privacy, 2021.

[25] V. Migliore et al., "Hardware/Software Co-Design of an Accelerator for FV Homomorphic Encryption Scheme Using Karatsuba Algorithm," IEEE Transactions on Computers, vol. 67, no. 3, 2018.

[26] P. Mishra et al., "DELPHI: A Cryptographic Inference Service for Neural Networks," in 29th USENIX Security Symposium, 2020.

[27] M. C. Pease, "An Adaptation of the Fast Fourier Transform for Parallel Processing," Journal of ACM, vol. 15, no. 2, 1968.

[28] B. Reagen et al., "Cheetah: Optimizing and Accelerating Homomorphic Encryption for Private Inference," in HPCA, 2021.

[29] M. Sadegh Riazi et al., "HEAX: An Architecture for Computing on Encrypted Data," in ASPLOS, 2020.

[30] N. Samardzic et al., "CraterLake : A Hardware Accelerator for Efficient Unbounded Computation on Encrypted Data," in ISCA, 2022.

[31] S. Sinha Roy et al., "FPGA-Based High-performance Parallel Architecture for Homomorphic Computing on Encrypted Data," in HPCA, 2019.