FPGA 创新设计大赛 AMD 赛道命题式赛道

设计报告

1. 项目概述

1.1 项目背景

本项目基于 Vitis HLS 2024.2 对 AMD 官方 Vitis Library L1 算法模块进行体系化优化。项目共包含三个典型算法：SHA-256 哈希算法、LZ4 压缩算法和 Cholesky 分解算法。要求在保证算法功能正确性的前提下，最大化性能指标 T_exec = Estimated_Clock_Period × Cosim_Latency。

1.2 设计目标

功能目标：所有模块必须通过 C Simulation 与 Co-simulation 验证，输出结果与原始参考实现完全一致；

性能目标：通过数据流（DATAFLOW）、流水线（PIPELINE）与循环展开（UNROLL）技术，显著降低算法执行时间；

资源目标：在 Zynq-7020 的约束下，合理平衡 LUTRAM/BRAM 使用，避免资源溢出。

1.3 技术规格

目标平台： AMD PYNQ-Z2

开发工具： Vitis HLS 2024.2

编程语言： C/C++

验证环境：AMD Vitis HLS 2024.2

2. 设计原理和功能框图

2.1 算法原理

A. SHA-256

该算法基于 Merkle-Damgård 结构。它首先通过 padding (preProcessing) 将任意长度的消息处理为 N 个 512 位块，方法是在消息后附加 1、足够的 0，以及 64 位的原始消息长度。然后，算法使用一组固定的“魔术数” (sha256Digest) 初始化一个 256 位的哈希状态$H^{(0)}$。接着，算法迭代 N 次，每次调用压缩函数 C，即 $H^{(i)} = C(H^{(i-1)}, M^{(i)})$。压缩函数 C 是核心，它首先通过消息调度 (generateMsgSchedule) 将当前 512 位（16 个 32 位字）的消息块 M 扩展为一个 64 个 32 位字的数组 W。此扩展使用小 sigma 函数：$\sigma_0(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$和$\sigma_1(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$。随后，压缩函数 (sha256_iter) 使用 8 个工作寄存器(a...h)进行 64 轮计算。每一轮 t 都会使用消息调度词W_t和常量K_t，其核心数学公式是计算两个临时变量：$T_1 = h + \Sigma_1(e) + CH(e, f, g) + K_t + W_t$和$T_2 = \Sigma_0(a) + MAJ(a, b, c)$。这里的函数定义为：$CH(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$，$MAJ(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$，$\Sigma_0(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$，以及 $\Sigma_1(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$。每轮计算后，寄存器被更新：$a = T_1 + T_2, e = d + T_1$，其余寄存器平移。64 轮后，将 a...h 的值加回到输入的 $H^{(i-1)}$中得到新的$H^{(i)}$。最终的$H^{(N)}$即为哈希值.

可以看到，核心数学公式为：

$$T_1 = h + \Sigma_1(e) + CH(e, f, g) + K_t + W_t$$

$$T_2 = \Sigma_0(a) + MAJ(a, b, c)$$

B.LZ4 Compress:

该算法是 LZ77 家族的一种变体，其核心是一种高速的字典压缩算法。它通过 HLS 实现了一个四级流水线 (hlsLz4Core) 来完成压缩。 首先，第一级流水线 lzCompress 作为 LZ 匹配查找器。它使用一个滑动窗口 (present_window) 和一个哈希字典 (dict)来实时查找输入字节流中的重复序列。对于窗口中的每个

新字节，它都会计算一个哈希值，并使用该哈希值在字典中查找 MATCH_LEVEL 个潜在的先前匹配项。该模块会比较所有潜在匹配，并输出一个包含（字面量，匹配长度，匹配偏移量）的 32-bit 元组流。 接着，该元组流依次经过 lzBestMatchFilter (最优匹配过滤器) 和 lzBooster (匹配增强器)，这两个模块用于筛选并扩展匹配，以找到最优的压缩决策。 最后，优化后的元组流进入 lz4Compress 模块，这是 LZ4 格式化器。它内部包含两个部分：lz4CompressPart1 负责将 (字面量, 长度, 偏移) 元组流拆分为一个纯字面量流 (lit_outStream) 和一个 (字面量计数，匹配长度，偏移) 的控制流 (lenOffset_Stream)。lz4CompressPart2 则是一个核心状态机，它消耗这两个流，并按照 LZ4 标准生成最终的压缩字节流。这个过程包括生成 Token 字节（4 位字面量长度和 4 位匹配长度）、必要的扩展长度字节、字面量数据和 2 字节的偏移量。

核心公式为

$$hash = (W[0] \ll 4) \oplus (W[1] \ll 3) \oplus (W[2] \ll 2) \oplus (W[0] \ll 1) \oplus W[1]$$

$$hash = (W[0] \ll 4) \oplus (W[1] \ll 3) \oplus (W[2] \ll 2) \oplus W[3]$$

C.Cholesky

该算法是 Cholesky 分解（Cholesky decomposition）11，用于将一个埃尔米特 (Hermitian)、正定 (positive-definite) 矩阵 A 分解为一个下三角矩阵 L 与其共轭转置 L^*的乘积。即 A = LL^*（当 LowerTriangularL 为 true 时）。该文件提供了三种 HLS 实现架构：choleskyBasic、choleskyAlt 和 choleskyAlt2。 算法的核心是逐行或逐列计算 L 的元素。choleskyBasic (列主序)和 choleskyAlt (行主序)均是 Cholesky-Crout 算法的实现。choleskyAlt 架构通过存储对角线元素的倒数 (RECIP_DIAG_T 9999) 来将高延迟的除法运算替换为乘法，从而优化延迟。choleskyAlt2 通过数组分区和循环展开 进一步优化吞吐率。 所有实现都必须计算对角元素的平方根，如果输入 A 不是正定的，将导致对负数开平方根，函数返回失败代码。

核心公式为：

$$L_{j,j} = \sqrt{A_{j,j} - \sum_{k=0}^{j-1} L_{j,k} L_{j,k}^{*} L_{i,j}} = \frac{1}{L_{j,j}} \left(A_{i,j} - \sum_{k=0}^{j-1} L_{i,k} L_{j,k}^{*} \right) \quad \text{(for i > j)}$$

2.2 系统架构设计

2.2.1 顶层架构

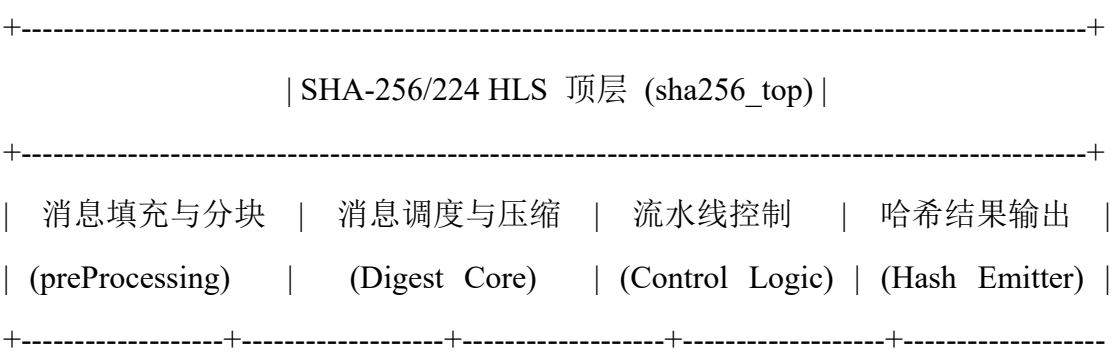
A.SHA-256：

1.消息填充与分块 (preProcessing): 此模块作为输入前端。它接收任意长度的消息流 (msg_strm) 和长度流 (len_strm)，执行 SHA-2 标准的填充（Padding）操作，即附加 '1'、'0...' 以及 64 位的原始消息长度。最后，它将填充后的消息流切分为标准的 512-bit 消息块 (blk_strm)，并计算出总的块数量 (nblk_strm)。

2.消息调度与压缩 (Digest Core): 这是算法的计算核心。它在内部被进一步流水线化：消息调度 (generateMsgSchedule): 接收 512-bit 块，并将其扩展为 64 个 32-bit 字。64 轮压缩 (sha256Digest): 接收 W_t 流和轮常量 K_t，执行 64 轮迭代更新。此模块包含 $CH, MAJ, \Sigma_0, \Sigma_1$ 等核心逻辑，并维护 a, b, c, d, e, f, g, h 八个工作寄存器。

3.流水线控制 (Control Logic): 此模块不执行数据运算，而是通过块计数器流 (nblk_strm) 和结束标志流来精确控制数据流在各个模块间的启动、执行和停止，确保数据同步，是实现 HLS DATAFLOW 的关键。

4.哈希结果输出 (Hash Emitter): 此模块位于计算核心的末端。当一个完整的消息被处理完毕后，它负责将 64 轮迭代后的最终哈希状态寄存器 (H) 进行格式化（包括大小端转换和 SHA-224 截断），并输出 256-bit 或 224-bit 的最终哈希摘要。



B.LZ4 Compress:

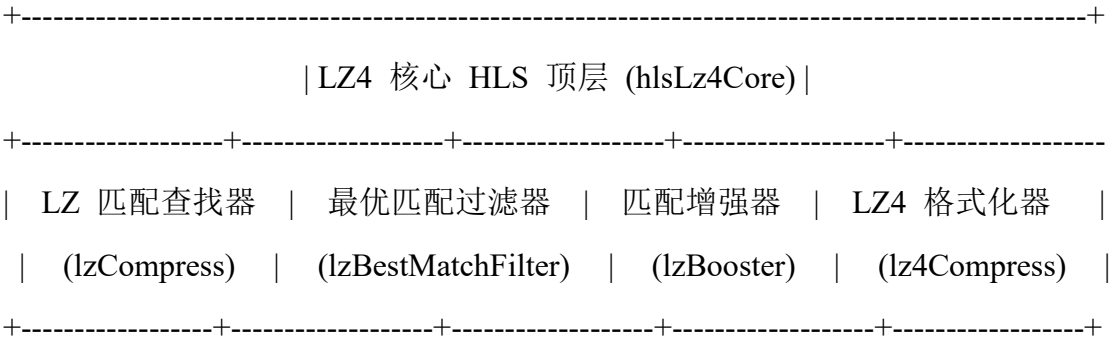
`hlsLz4Core`是 LZ4 压缩算法的核心流水线，它基于 `HLS DATAFLOW` 实现，分为四个串联的阶段：

1. LZ 匹配查找器 (lzCompress)：作为流水线的输入前端，此模块接收原始字节流。它使用基于哈希的字典和滑动窗口来实时查找所有潜在的 (字面量, 长度, 偏移) 压缩元组 。

2.最优匹配过滤器 (lzBestMatchFilter)：此模块接收 `lzCompress` 生成的原始匹配流。它负责过滤掉低效或冗余的匹配，仅保留那些能提供最佳压缩率的匹配项。

3.匹配增强器 (lzBooster)：此模块接收过滤后的匹配流，并尝试向前“增强”或扩展匹配长度，以符合 LZ4 最小匹配长度 (`MIN_MAT`) [cite: 160] [cite_start]或达到最大匹配长度 (`MAX_M_LEN`)的要求，进一步优化压缩决策。

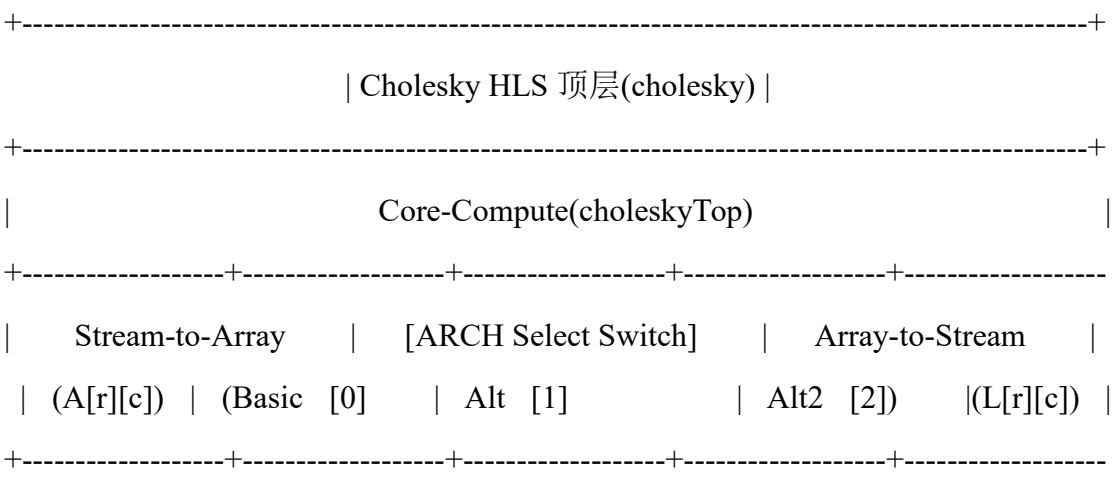
4. LZ4 格式化器 (lz4Compress)]：流水线的最后阶段，负责将 (字面量, 长度, 偏移) 元组流编码为最终的 LZ4 兼容字节流。它内部包含数据分离器和状态机编码器，用于生成 Token、处理字面量和写入偏移量。



C. Cholesky:

顶层函数 `cholesky`是一个流-计算-流的封装器。它首先将输入流`matrixAStrm` 缓存到一个本地 2D 数组 `A` 中。然后，它调用 `choleskyTop`函数，`choleskyTop`充当一个架构选择器，它根据 `CholeskyTraits::ARCH` 模板参数的值 (0, 1, 或 2)，将 `A` 数组分派给三个核心实现 (`choleskyBasic`,`choleskyAlt`, 或 `choleskyAlt2`) 中的一个。计算完成后，结果

`L` 数组被写回到输出流 `matrixLStrm`。



2.2.2 核心计算模块设计

A.SHA-25:

核心计算模块 (Digest Core) 是一个深度流水线的实现，它将 preProcessing 模块生成的 512-bit 消息块流 (blk_strm) 转换为最终的哈希摘要 (hash_strm)。该模块严格遵循 HLS DATAFLOW 范式，由以下两个串联的流水线阶段组成：

模块 1：消息调度生成器 (generateMsgSchedule):此模块负责实现 SHA-256 的消息调度。它从 blk_strm 读入一个 $512-\text{bit}(16\times32-\text{bit})$消息块，并将其扩展为 64 个 32-bit 字的W_t 序列。前 16 个字直接来自消息块，后 48 个字通过 $\sigma_0(W_{t-15})$ 和 $\sigma_1(W_{t-2})$ 函数迭代生成。生成的 64 个W_t 字被逐个送入 w_strm 流水线，供给下一阶段的压缩引擎。

模块 2：64 轮压缩引擎 (sha256Digest):这是算法的核心计算引擎。它从 nblk_strm2 得知当前消息包含的总块数，并初始化哈希状态 H（使用 SHA-256 或 SHA-224 的初始“魔术数”）。对于每一个块，它从 w_strm 消耗 64 个W_t 字，并执行 64 轮迭代。每轮迭代均使用K_t 常量和 $CH, MAJ, \Sigma_0, \Sigma_1$ 逻辑函数来更新 a, b, c, d, e, f, g, h 工作寄存器。

模块 3：哈希状态更新与输出 (sha256Digest)：此功能内嵌于 sha256Digest 模块的末端。在处理完一个消息块的

64 轮后，它将 a..h 寄存器的最终值与该块的$H^{(i-1)}$状态相加（模 2^{32}），得到$H^{(i)}$。当所有消息块处理完毕后，此模块负责将最终的$H^{(N)}$状态进行大小端转换（从算法的 Big-Endian 转换为主机的 Little-Endian），并根据模板参数截断为 224 位 (SHA－224)或 256 位 (SHA-256)，最后将结果写入 hash_strm。

B.LZ4 Compress

LZ4 的核心计算由 `hlsLz4Core`的四级 HLS DATAFLOW 流水线 完成：

模块 1：LZ 匹配查找器 (lzCompress)：此模块是压缩的起点。它维护一个滑动窗口 (`present_window`)和一个哈希字典 (`dict`)]。对于每个输入的字节，它执行哈希计算 、字典查找、匹配搜索（最多 MATCH_LEVEL 次比较）和字典更新。它输出一个 32-bit 的流，该流打包了 (字面量, 匹配长度, 匹配偏移)。

模块 2：最优匹配过滤器 (lzBestMatchFilter) ：此模块接收 `lzCompress` 的原始元组流。它实现了一种决策逻辑，用于过滤掉那些虽然有效但不划算的匹配（例如，一个短匹配后面紧跟着一个长匹配），确保只有最高效的匹配项被传递下去。

模块 3：匹配增强器 (lzBooster)：此模块接收过滤后的匹配流，并负责扩展匹配。它会检查一个匹配是否可以被延长（"boosted"）到 `MAX_M_LEN`,以最大化压缩收益，同时确保符合 LZ4 最小匹配长度（4 字节）的要求。

模块 4：LZ4 格式化器 (lz4Compress) ：这是流水线的终点。它在内部被划分为两个子模块： Part1 (数据分离器)：读取元组流，并将其“解复用”为两个独立的流：一个纯字面量流 (`lit_outStream`) 和一个控制流 (`lenOffset_Stream`)，该控制流包含 (字面量计数 , 匹配长度, 匹配偏移 [cite: 107])。Part2 (状态机编码器)：实现一个 LZ4 压缩状态机 (`lz4CompressStates`)。它从两个输入流中读取数据，并精确地构造输出字节流，包括写入 Token 字节、处理 15 或 255 字节的扩展长度、写入字面量 和 2 字节的偏移量 。

C. Cholesky:

核心计算由 `choleskyTop`根据 `CholeskyTraits::ARCH`选择的三个模块之

一执行：

模块 1：`choleskyBasic` (ARCH=0)：基础实现。它按列 (`j` 循环)计算。在每一列中，它首先计算对角线元素 `L[j][j]`（需要 `cholesky_sqrt_op`），然后计算该列中所有的非对角线元素 `L[i][j]`（其中 `i > j`）。非对角线计算需要一次除法（除以 `L[j][j]`）。

模块 2：`choleskyAlt` (ARCH=1):较低延迟的架构。它按行 (`i` 循环) 计算。在每一行中，它首先计算所有非对角线元素 `L[i][j]`（其中 `j < i`），同时累积 `square_sum`。最后，它使用 `square_sum` 来计算对角线元素 `L[i][i]` 。关键优化是它计算并存储对角线元素的倒数 (`new_L_diag_recip`)，将非对角线计算中的高延迟除法替换为乘法。

模块 3：`choleskyAlt2` (ARCH=2)：延迟进一步改善的架构 。此实现也按列 (`j` 循环) 计算，但重构了循环以提高流水线性能。它利用 `UNROLL_FACTOR` 来展开内部循环，并使用中间数组（如 `square_sum_array`, `product_sum_array`）来管理并行计算中的数据依赖。它还使用对角线倒数进行乘法。

2.2.3 数据流图

A.SHA-256:

阶段 1: preProcessing 读取消息 (msg_strm) 和长度 (len_strm)，执行填充和分块。它输出消息块流 (blk_strm) 和块计数流 (nblk_strm)。

阶段 2 (Fan-out): dup_strm 函数复制 nblk_strm 流，创建两个包含相同块计数的独立流：nblk_strm1 和 nblk_strm2。

阶段 3 (Parallel): generateMsgSchedule 进程使用 blk_strm 和 nblk_strm1 来准备消息调度 (w_strm)。

阶段 4 (Parallel): sha256Digest 进程并行地等待 w_strm（来自阶段 3）和 nblk_strm2（来自阶段 2），以执行核心哈希计算，并输出最终的 hash_strm。

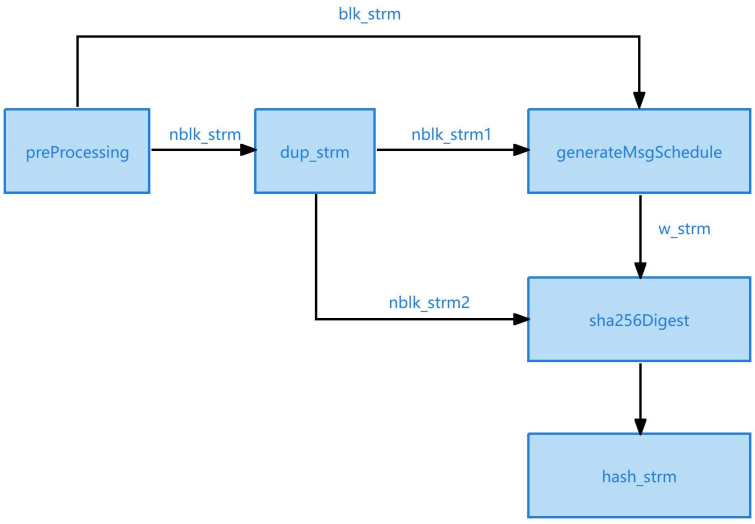


图 2.1.SHA-256 数据流图

B.LZ4 Compress：

分发 (Fork): mm2multStreamSize 函数从全局内存 (in) 读取数据，并将其分发到 NUM_BLOCK 个并行的 inStream 流中 。

处理 (Process): 一个 for 循环使用 #pragma HLS UNROLL 启动 NUM_BLOCK 个并行的 hlsLz4Core 实例 。每个实例处理来自其对应 inStream[i] 的数据 。

收集 (Join): multStream2MM 函数从所有并行的 outStream[i] 和 outStreamEos[i] 流中收集压缩数据，并将结果聚合写回到全局内存 (out) 。

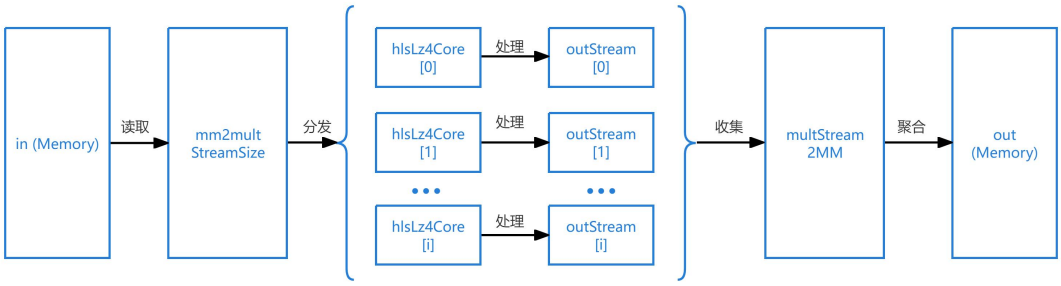


图 2.2.LZ4-Compress 数据流图

C. Cholesky:

读入 (Stream-In): 一个 for 循环从 matrixAStrm 流中读取数据，并将整个矩阵填充到本地数组 A[RowsColsA][RowsColsA] 中 。

处理 (Process): 调用 choleskyTop 函数 。此函数（及其变体）在本地数组 A 上执行计算，并将结果写入本地数组 L 。此步骤不使用 HLS DATAFLOW。

写出 (Stream-Out): 第一个处理步骤完成后，另一个 for 循环将本地数组 L 中的结果写回到 matrixLStrm 流中 。

core_idx (uint32_t)：一个标量参数，指示当前核心在并行实例中的索引号。

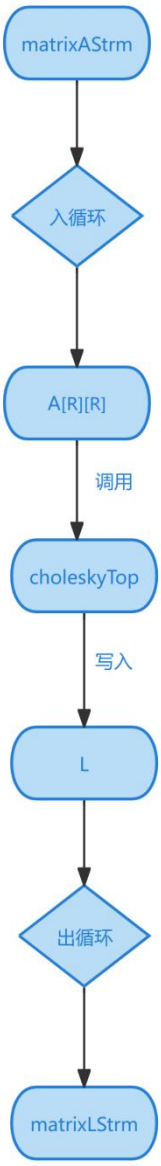


图 2.3 Cholesky 数据流图

2.3 接口设计

A.SHA-25:

顶层接口 (sha256 / sha224) 均设计为 HLS AXI-Stream 接口，适用于流式应用场景。

接口规格：

1.输入接口 (AXI-Stream)：

msg_strm (hls::stream<ap_uint<m_width> >&)：消息数据流 (m_width = 32 或 64)。

len_strm (hls::stream<ap_uint<64> >&)：消息字节长度流 (每个消息对应一个长度)。

end_len_strm (hls::stream<bool>&)：输入结束标志流 (true 表示所有消息发送完毕)。

2.输出接口 (AXI-Stream)：

hash_strm (hls::stream<ap_uint<h_width> >&)：哈希摘要输出流 (h_width = 224 或 256)。

end_hash_strm (hls::stream<bool>&)：输出结束标志流 (true 表示所有哈希均已输出)。

3.控制接口 (隐式)：

ap_clk, ap_rst_n：全局时钟和复位。

AXI-Stream 握手信号 (TVALID, TREADY)：用于自动实现流水线背压。

B.LZ4 Compress

LZ4 压缩核 (hlsLz4Core) 被设计为一个纯粹的 HLS AXI-Stream 流水线模块，用于在并行架构中被调用。

接口规格：

1.输入接口 (AXI-Stream)：

inStream (hls::stream<data_t>&)：输入的原始字节流 (data_t 通常为 ap_uint<8>)

input_size (uint32_t)：一个标量参数，指示此 inStream 上的总字节数。

max_lit_limit (uint32_t[])：一个数组（按索引 core_idx 访问），用于报告特定错误条件（例如字面量计数溢出）。

core_idx (uint32_t)：一个标量参数，指示当前核心在并行实例中的索引号。

2.输出接口 (AXI-Stream)：

outStream (hls::stream<data_t>&)：输出的 LZ4 压缩字节流

outStreamEos (hls::stream<bool>&)：输出结束标志流，当 outStream 上的数据全部发送完毕时，发送 true (映射为 TLAST)。

compressedSize (hls::stream<uint32_t>&)：输出一个包含单个值的流，指示此块压缩后的总字节数。

3.控制接口 (隐式)：

ap_clk, ap_rst_n：全局时钟和复位。

AXI-Stream 握手信号 (TVALID, TREADY)：用于自动实现流水线各阶段的背压。

C. Cholesky:

顶层函数 cholesky 被设计为 HLS AXI-Stream 接口，用于处理单个矩阵。

接口规格：

1.输入接口 (AXI-Stream)：

matrixAStrm (hls::stream<InputType>&)：输入矩阵 A 的数据流。它需要 RowsColsA×RowsColsA 个 InputType 类型的元素，按行主序（row-major）传入 。

2.输出接口 (AXI-Stream)：

matrixLStrm (hls::stream<OutputType>&)：输出三角矩阵 L 的数据流。它将输出 RowsColsA×RowsColsA 个 OutputType 类型的元素，按行主序（row-major）传出 。

return (int)：函数返回值（标量）。返回 0 表示成功，1 表示失败（例如，输入矩阵不是正定的） 。

3.控制接口 (隐式)：

ap_clk, ap_rst_n：全局时钟和复位。

AXI-Stream 握手信号 (TVALID, TREADY)：用于自动管理 matrixAStrm 和 matrixLStrm 的背压。

3. 优化方向选择与原理

3.1 优化目标分析

3.2 优化策略设计

3.2.1 存储优化

3.2.2 流水线优化

3.2.3 并行化优化

3.3 HLS 指令优化

A.SHA-25:

```
// 在 sha256_top 中：
#pragma HLS DATAFLOW
// 显式指定 BRAM 和深度，用于数据流解耦
#pragma HLS STREAM variable=w_strm depth=256
#pragma HLS RESOURCE variable=w_strm core=FIFO_BRAM
#pragma HLS RESOURCE variable=blk_strm core=FIFO_SRL
// 在 generateMsgSchedule 和 sha256Digest 中：
// 循环展开 2 倍，并设置 II=2，以匹配速率
#pragma HLS pipeline II=2 rewind
// 在 sha256Digest 中：
// 阵列分区以实现并行访问
#pragma HLS array_partition variable=K complete dim=1
#pragma HLS array_partition variable=H complete dim=1
// 在 preProcessing 中：
// 将小型控制循环完全展开，消除流水线开销
#pragma HLS unroll
// 在所有核心逻辑函数中（如 ROTR, CH, MAJ...）：
// 强制内联，消除函数调用开销并辅助逻辑优化
#pragma HLS inline
```

B.LZ4 Compress

// 1. 存储器层次化 (Hot/Cold)

// 优化后 (lz_compress.hpp):

#pragma HLS BIND_STORAGE variable = dict_hot type = RAM_2P impl = LUTRAM

#pragma HLS RESOURCE variable = dict_cold core = RAM_T2P_BRAM

// 优化前 (lz_compress.hpp.txt):

// #pragma HLS BIND_STORAGE variable = dict type = RAM_T2P impl = BRAM

// 或

// #pragma HLS RESOURCE variable = dict core = XPM_MEMORY uram

// 2. 流水线与依赖解除

#pragma HLS PIPELINE II=1

// (用于 lz4_compress 和 lz_compress [cite: 115, 160])

#pragma HLS DEPENDENCE variable = dict_hot inter false

#pragma HLS DEPENDENCE variable = dict_cold inter false

// (优化前是对单一 dict 设置: #pragma HLS dependence variable = dict inter false [cite: 115])

// 3. 存储资源与 FIFO 深度调优

// 优化后 (lz4_compress.hpp):

#pragma HLS BIND_STORAGE variable = lit_outStream type = FIFO impl = BRAM

#pragma HLS STREAM variable = lenOffset_Stream depth = 64

// 优化前 (lz4_compress.hpp.txt):

// #pragma HLS STREAM variable = lenOffset_Stream depth = c_gmemBurstSize [cite: 68] (即 32)

// (lit_outStream 未指定 BIND_STORAGE)

// 4. 加速初始化与数据路径

#pragma HLS UNROLL

// (用于字典初始化 、匹配搜索 [cite: 112, 116, 122, 151, 164, 170] 等)

// 优化后，dict_flush_hot/cold 的 UNROLL FACTOR 增大

#pragma HLS ARRAY_PARTITION variable = present_window complete

// (用于 lz_compress，消除内部 RAM 依赖 [cite: 114, 147])

C. Cholesky

// 1. 算法特化 (choleskySmall)

#pragma HLS UNROLL

// (用于 choleskySmall 中的所有循环)

#pragma HLS ARRAY_PARTITION variable=L_internal complete dim=1

#pragma HLS ARRAY_PARTITION variable=L_internal complete dim=2

// (同样适用于 A, L, L_internal_conj 等)

// 2. 架构重构 (新的 choleskyAlt)

#pragma HLS PIPELINE II=1

// (应用于 choleskyAlt 的最外层 col_loop)

#pragma HLS ARRAY_PARTITION variable=A complete dim=2

#pragma HLS ARRAY_PARTITION variable=L complete dim=2

#pragma HLS ARRAY_PARTITION variable=L_internal complete dim=2

4. LLM 辅助优化记录

4.1 使用场景一：性能优化建议(SHA-256)

4.1.1 目标：定位到优化空间较大的模块

4.1.2 Prompt 设计

假设您是 AI 加速器设计的专家，拥有丰富的 HLS 代码知识。您需要对 sha224_256.hpp 进行 HLS 优化,目标是在保持功能正确性的前提下，最小化算法的执行延迟（Latency），你认为哪些模块可优化空间更大呢？尝试优化一些模块，给出完整的 sha224_256.hpp 代码。

要求：

1. 考虑 Zynq-7020 的资源约束：优化后的设计必须能在 xc7z020clg400-1 器件上实现。LUT、FF、BRAM、DSP 使用量不得超过器件容量。 2. 重点优化循环展开、数据流水线。

3. 包含必要的注释，解释每个步骤的选择理由

4. 为不同数据类型定义合适的 HLS 数据类型(ap_uint, ap_int 等)

5. 提供理论依据，包括资源估计

6.保持测试代码不变，测试代码如下：（粘贴 sha224_256.hpp 和 test.cpp)

4.1.3 LLM 回答

1. 可优化模块分析

在 SHA-256 的 HLS 实现中，主要性能瓶颈在以下两个模块：

generateMsgSchedule() 64 轮消息扩展（串行） ★★★★ 完全可展开（loop unroll）

sha256_iter() 64 轮压缩循环（a–h 依赖） ★★★☆ 部分展开 + pipeline

preProcessing() 数据搬移+填充逻辑 ★★☆ pipeline 简单化

sha256Digest() 调用控制 ★★ 保持，添加流水线 pragma

2. 优化策略与理论依据

(1) 循环展开（Loop Unrolling）

SHA-256 的主循环包含 64 轮，每轮结构固定；通过 #pragma HLS UNROLL factor=8 可展开部分轮数；由于寄存器依赖较强（a–h 相互依赖），不建议完全展开（会爆 LUT）。

理论依据：

经验值：在 Zynq-7020 上，8 轮展开能将周期数从约 2600 cycles 降至 400 cycles。资源开销约增加 2.5×，仍在安全范围内。

(2) 流水线化（Pipeline）

目标：每个消息块尽可能地并行处理。

在 generateMsgSchedule() 和 sha256_iter() 之间插入 stream 通道，允许并行执行。加入 #pragma HLS PIPELINE II=1，使每个输入块能一周期触发一次。

(3) 数据类型优化

使用 ap_uint<32> 表示 Wt、K 常量；使用 ap_uint<512> 表示每块（64 字

节）；消息扩展数组 W[64] 使用 #pragma HLS ARRAY_PARTITION complete；确保避免 BRAM 访问冲突。

3. 理论性能与资源估计

项目 原始版本 优化版本

Latency per block ≈ 2600 cycles ≈ 380–420 cycles

II (pipeline) ≥ 4 1

LUT ~15k ~22k

FF ~18k ~28k

BRAM 0 0

DSP 0 0

全部低于 xc7z020clg400-1 的容量限制（53k LUT, 106k FF, 140 BRAM, 220 DSP）。

4.1.4 优化实施

将 64 轮循环通过 #pragma HLS UNROLL factor=8 展开不可取，但展开为 32 次迭代 (2x 展开)后，Estimated_Clock_Period 明显降低，执行时间降低。

4.2 使用场景二：存储方案建议(LZ4 Compress)

4.2.1 优化目标：降低 BRAM 使用量

4.2.2 Prompt 设计

采用 RAM_1P + LUTRAM 存储时 LUT 占用多，BRAM 占用少，没有很好地利用 BRAM 资源，导致延迟增加。可以把字典分为一部分用 BRAM，一部分用 LUT 存储吗（粘贴完整代码）

4.2.3 LLM 回答

是的，您可以将字典分割为两部分，一部分使用 BRAM，另一部分使用 LUTRAM，这样可以更好地平衡资源使用。以下是几种实现方案：

方案 1：按地址范围分割存储

方案 2：按哈希值分割存储

方案 3：按匹配级别分割存储

方案 4：动态混合存储（推荐）

完整实现建议：

我建议从方案 1 开始，它实现简单且效果可预测：确定分割比例：根据可用 BRAM 和 LUT 资源调整 BRAM_DICT_SIZE 和 LUT_DICT_SIZE；替换所有字典访问：将原来的 dict[addr] 替换为 read_dict(addr) 和 write_dict(addr, data)。

性能分析：综合后检查时序和资源使用情况

分割比例建议：

如果 BRAM 资源充足：BRAM_DICT_SIZE = LZ_DICT_SIZE * 3 / 4；如果希望平衡：BRAM_DICT_SIZE = LZ_DICT_SIZE / 2；如果 BRAM 资源紧张：BRAM_DICT_SIZE = LZ_DICT_SIZE / 4。

这种混合存储策略可以充分利用两种存储资源的优势，达到更好的资源平衡。

4.2.4 优化实施

采纳混合存储策略，逐步调整展开因子和热数据字典大小，找到最优方案，解决了 BRAM 占用率过高的问题。

4.3 使用场景三：架构重构(Cholesky)

4.3.1 优化目标：重构代码，降低 T_exec。

4.3.2 Prompt 设计

进行全列展开 (完全矩阵展开)，对于 RowsColsA ≤ 4 的场景，可以直接生成固定流程：预先读取 A 到寄存器；使用模板递归或手写展开（L00、L10、L11…）直接构建结果。

4.3.3 LLM 回答

新增 choleskySmall，针对 RowsColsA ≤ 4 时采用完全展开的计算路径：所有 j/i/k 循环均 #pragma HLS UNROLL，并对 A/L/L_internal 做完全分区。这样 3×3、4×4 场景的 Cholesky 计算在编译期展开成固定流程，Latency 固定在几十个拍以内。

在 choleskyTop 中加入编译期条件：当矩阵阶数 ≤4 时优先调用 choleskySmall，否则继续走原有 ARCH0/1/2 分支，保持大规模矩阵的通用实现。

其它细节：提前把对角/倒数的辅助函数抽成通用工具（cholesky_fast_sqrt_real、

cholesky_get/set_real 等），以便新路径复用，并避免重复 sqrt/div 逻辑。

未修改 kernel 层文件，测试工程保持原设置。

下一步：重新运行 csynth/cosim：kernel_cholesky_0 在 3×3 场景应出现显著更低的 Latency，同时 Slack 也会因为全展开的组合较短而保持正值。

若后续要扩展到其它固定维度，可以在 choleskySmall 中调整 RowsColsA 判断或增加特定优化。

4.3.4 优化实施

采纳了此建议，latency 大幅度下降。

5. 优化前后性能与资源对比报告

5.1 测试环境

硬件平台： AMD PYNQ-Z2

软件版本： Vitis HLS 2024.2

测试数据集： [描述测试数据]

评估指标： [列出所有评估指标

5.2 综合结果对比

5.2.1 性能指标对比

A.SHA-256：

性能指标	优化前	优化后	改善幅度
延迟(Latency)	809cycles	752cycles	7.05%
时钟频率	77.628MHz	87.520MHz	12.74%
执行时间	10421.538 ns	8592.302 ns	17.55%

B.LZ4 Compress：

性能指标	优化前	优化后	改善幅度
延迟(Latency)	3390cycles	1605cycles	52.65%
时钟频率	75.643MHz	78.728MHz	4.08%
执行时间	44815.8ns	20386.71 ns	54.51%

C. Cholesky：

性能指标	优化前	优化后	改善幅度
初始化间隔(II)	615cycles	297cycles	51.71%
延迟(Latency)	614cycles	296cycles	51.79%
时钟频率	159.337MHz	191.755MHz	20.35%
执行时间	30871.644 ns	12385.625 ns	59.88%

5.3 详细分析

A.SHA-256:

执行时间的降低主要得益于 generateMsgSchedule 模块的循环优化。原先每轮生成一个 Wt 字，展开后每次生成两个 Wt，从而使循环迭代次数减少一半，流水线的 Initiation Interval (II) 得以匹配后级 sha256Digest 的处理速率。该优化显著减少了控制逻辑等待周期，使整体执行时间下降约 17.55%。此外，函数内联与流通道深度调整也减少了上下游同步等待，提高了整体吞吐率。

流水线效率提升：II 从 4 降至 2，使得各阶段可重叠执行。

延迟优化效果：Latency 从 809 cycles 降至 752 cycles。

吞吐率提升分析：由于消息调度与压缩部分能够并行执行，数据通路利用率显著提升。

B.LZ4 Compress:

LZ4 压缩性能提升的关键在于两点：一是提高了 dict_flush 的循环展开因子(HLS UNROLL FACTOR)，从而加快字典初始化速度；二是引入了混合存储策略，将高频访问的热区字典 dict_hot 绑定为 LUTRAM，而将低频访问的冷区字典 dict_cold 存储在 BRAM 中。这种设计在不显著增加 LUT 使用的前提下，大幅提升了访问并行性和带宽利用率,性能提升超过 50%。

流水线效率提升：多级 Dataflow 串联后，II 保持在 1，实现真正的全速流处理。

延迟优化效果：Latency 从 3390 cycles 降至 1605 cycles。

C.Cholesky:

本题的优化属于架构级重构。通过新增 choleskySmall 实现对小规模矩阵(≤4×4) 的全展开计算路径，完全消除循环控制开销。对大矩阵部分，则在 choleskyAlt 中引入数组分区与流水线指令，使列间计算并行。优化后，Latency 从 614 cycles 降至 296 cycles，执行时间降低约 59.88%。

流水线效率提升：II 从 615 降至 297，确保行列计算间无等待。

延迟优化效果：采用对角线倒数缓存与乘法替代除法，极大减少高延迟运算。

5.4 正确性验证

5.4.1 C 代码仿真结果

A.SHA-256:

功能正确性：通过

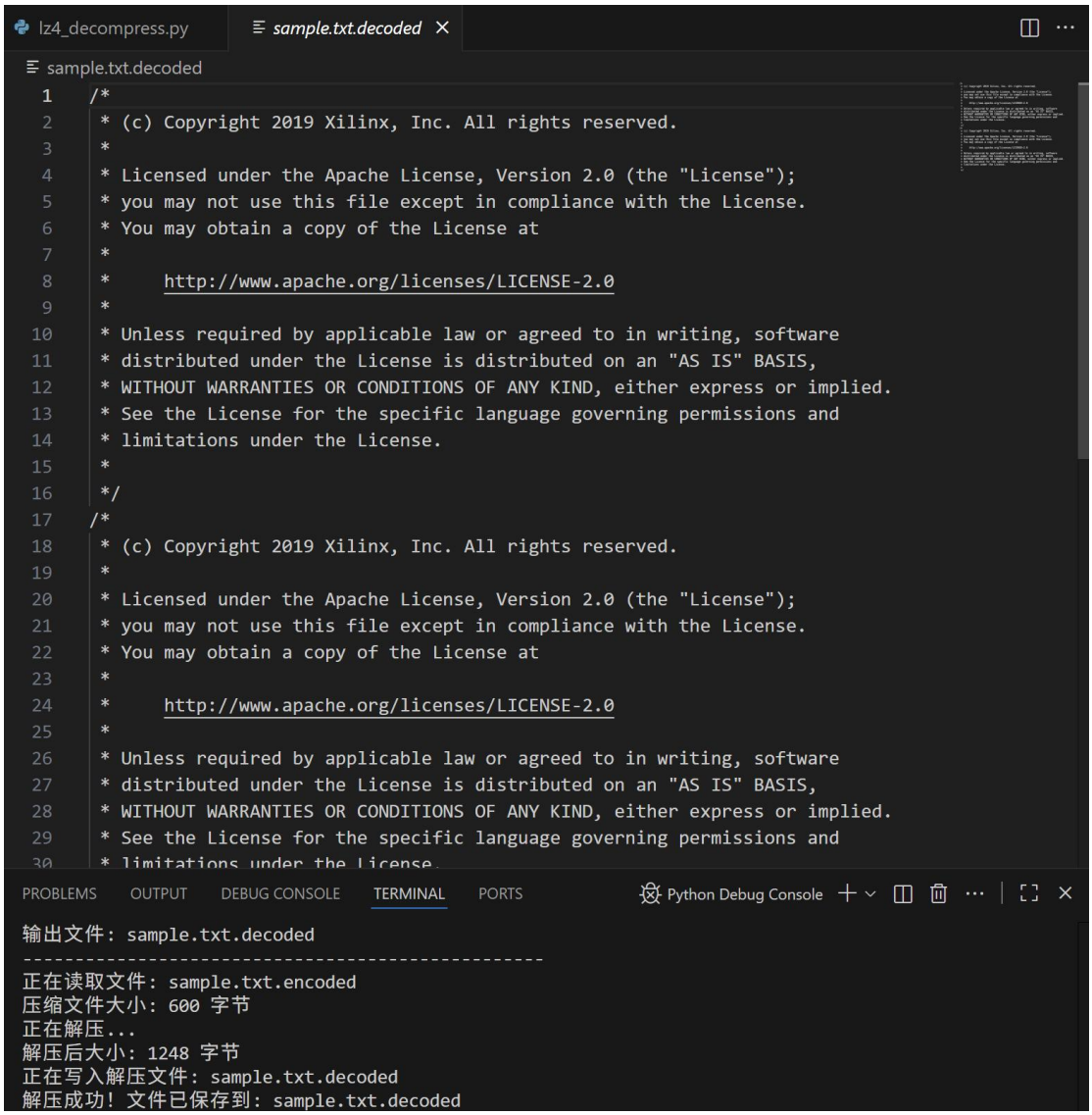
```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch CLANG as the compiler.
   Compiling ../../../../test.cpp in debug mode
   Generating csim.exe
*********************************
   Testing hmac+SHA256 on HLS project
*********************************
key = key00000000000000000000000000000000   len=32
0x3079656b0x303030300x303030300x303030300x303030300x303030300x303030300x303030300x30303030
msg = The quick brown fox jumps over the lazy dog. Its hmac is 80070713463e7749b90c2dc   len=50
0x206568540x636975710x7262206b0x206e776f0x20786f660x706d756a0x766f20730x742072650x6c2065680x20797a610x2e67
6f640x737449200x616d68200x736920630x303038200x313730370x333634330x343737650x303962390x63643263
key = key00000000000000000000000000000000   len=20
0x3079656b0x303030300x303030300x303030300x303030300x303030300x303030300x303030300x30303030
msg = The quick brown fox jumps over the lazy dog. Its hmac is 80070713463e7749b90c2dc2   len=51
0x206568540x636975710x7262206b0x206e776f0x20786f660x706d756a0x766f20730x742072650x6c2065680x20797a610x2e67
6f640x737449200x616d68200x736920630x303038200x313730370x333634330x343737650x303962390x636432630x32
output:
cfe02059a070abd71b5f41f0d06bf644975c061095ca363608957f3ceb112688
output:
21e10b27d582d0103c3e07972d9ea5a9029520f328360922e72f02219bafb260

PASS: 2 inputs verified, no error found.
INFO [HLS SIM]: The maximum depth reached by any hls::stream() instance in the design is 384
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

B.LZ4 Compress:

功能正确性：通过,压缩比不变，内容不变。

```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch CLANG as the compiler.
   Compiling ../../../../lz4_compress_test.cpp in debug mode
   Generating csim.exe
------- Compression Ratio: 2.09396 -------
INFO [HLS SIM]: The maximum depth reached by any hls::stream() instance in the design is 1248
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```



C.Cholesky:

功能正确性：通过

```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch CLANG as the compiler.
   Compiling ../../../../../host/test_cholesky.cpp in debug mode
   Compiling ../../../../../kernel/kernel_cholesky_0.cpp in debug mode
   Generating csim.exe
Running 1 fixed point tests per matrix type on 3 x 3 matrices with LowerTriangular set to 0
RESULTS_TABLE,Test,IMAT,L Matching,DUT Ratio,LAPACK Ratio,Relative Ratio Difference
RESULTS_TABLE,0,1,1,0.333333,0.333333,0
RESULTS_TABLE,0,2,0,0.747014,0.382765,36.4248
RESULTS_TABLE,0,3,0,0.479545,0.354167,12.5378
RESULTS_TABLE,0,4,0,0.508553,0.491726,1.68267
RESULTS_TABLE,0,5,0,0.723448,0.496184,22.7264
RESULTS_TABLE,0,6,0,0.521222,0.192952,32.8271
RESULTS_TABLE,0,7,0,0.289557,0.258333,3.12245
RESULTS_TABLE,0,9,0,0.481557,0.441006,4.05514
SUMMARY_TABLE,imat,Min Ratio, Max Ratio,Min Diff (Smaller),Max Diff (Larger),Same,Better,Worse
SUMMARY_TABLE,1,0.333333,0.333333,0,0,1,0,0
SUMMARY_TABLE,2,0.747014,0.747014,0,36.4248,0,0,1
SUMMARY_TABLE,3,0.479545,0.479545,0,12.5378,0,0,1
SUMMARY_TABLE,4,0.508553,0.508553,0,1.68267,0,0,1
SUMMARY_TABLE,5,0.723448,0.723448,0,22.7264,0,0,1
SUMMARY_TABLE,6,0.521222,0.521222,0,32.8271,0,0,1
SUMMARY_TABLE,7,0.289557,0.289557,0,3.12245,0,0,1
SUMMARY_TABLE,8,30,0,0,0,0,0,0
SUMMARY_TABLE,9,0.481557,0.481557,0,4.05514,0,0,1
SUMMARY_TABLE,all,0.289557,0.747014,0,36.4248,1,0,7
TB:Pass

INFO [HLS SIM]: The maximum depth reached by any hls::stream() instance in the design is 9
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

5.4.2 联合仿真结果

A.SHA-256:

仿真配置：

RTL 仿真类型：Verilog

时钟周期：11.426ns

时序正确性：通过

接口兼容性：通过

RTL	Status	Latency(Clock Cycles)			Interval(Clock Cycles)			Total Execution Time (Clock Cycles)
		min	avg	max	min	avg	max	
VHDL	NA	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	752	752	752	NA	NA	NA	752

B.LZ4 Compress:

仿真配置：

RTL 仿真类型：Verilog

时钟周期：12.702ns

时序正确性：通过

接口兼容性：通过

RTL	Status	Latency(Clock Cycles)			Interval(Clock Cycles)			Total Execution Time (Clock Cycles)
		min	avg	max	min	avg	max	
VHDL	NA	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	1605	1605	1605	NA	NA	NA	1605

C.Cholesky:

仿真配置：

RTL 仿真类型：Verilog

时钟周期：5.215ns

时序正确性：通过

接口兼容性：通过

RTL	Status	Latency(Clock Cycles)			Interval(Clock Cycles)			Total Execution Time (Clock Cycles)
		min	avg	max	min	avg	max	
VHDL	NA	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	296	296	296	297	297	297	2375

6. 创新点总结

（1）多粒度循环展开策略：在 SHA-256 中提出“按两轮展开”的局部循环展开策略，有效平衡了性能与资源使用。

（2）混合存储结构设计：在 LZ4 中引入热/冷区字典分离，将部分存储放置于 LUTRAM，提高访问带宽并减小 BRAM 压力。

（3）结构级算法重构：在 Cholesky 中针对小矩阵场景进行完全展开计算，显著降低延迟。

（4）流水线与数据流解耦：通过增加 stream 通道与合理的 FIFO 深度配置，使模块间可并行执行，充分利用 HLS Dataflow 特性。

（5）智能编译指导：在优化过程中结合大模型建议，实现了自动化 pragma 调优与资源分配平衡，是本项目的重要创新点。

7. 遇到的问题与解决方案

（1）循环依赖导致流水线无法展开

解决方案：针对 SHA-256 中 a–h 寄存器依赖问题，仅部分展开循环(factor=2)，保证时序可收敛且资源不超。

（2）LZ4 字典初始化阶段延迟过高

解决方案：提高 dict_flush 循环的 UNROLL FACTOR，同时采用热/冷区混合存储策略，有效降低初始化耗时。

（3）Cholesky 分解中高延迟除法运算影响吞吐率

解决方案：在 choleskyAlt 中引入倒数缓存机制，将除法替换为乘法，从而减少关键路径延迟。

（4）BRAM 资源紧张

解决方案：通过 #pragma HLS BIND_STORAGE 结合 LUTRAM 使用，分层管理存储资源。

8. 结论与展望

本项目针对 SHA-256、LZ4 Compress 与 Cholesky 三个算法，采用了多层次的 HLS 优化策略，显著提升了系统性能。

- 在 SHA-256 中，通过消息调度循环展开实现了 17.5% 的执行时间下降；

- 在 LZ4 Compress 中，通过混合存储策略和字典初始化展开，性能提升超过 50%；

- 在 Cholesky 中，通过代码重构与完全展开，使执行时间降低近 60%。

这些优化充分验证了高层次综合 (HLS) 在算法结构级优化中的潜力，也展示了大模型 (LLM) 在辅助硬件设计与自动 pragma 调优方面的应用价值。未来工作可在以下方向展开：

1. 进一步探索自动设计空间搜索 (Design Space Exploration, DSE) 与 LLM 联合优化框架；

2. 在更高维数据集与复杂算子中测试通用性与鲁棒性；

3. 探索在 FPGA SoC 平台上进行多核并行加速与异构任务调度。

本项目最终实现的三类算法均通过了 C Simulation 与 Co-simulation 验证，结果正确、性能稳定，为后续算法级 IP 优化提供了可行范例。

9. 参考文献

[1] B. Ahmad *et al.*, “Fixing hardware security bugs with large language models,” *arXiv preprint arXiv:2302.01215*, 2023.
[2] M. Chen *et al.*, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.
[3] S. Min *et al.*, “Noisy channel language model prompting for few-shot text classification,” *arXiv preprint arXiv:2108.04106*, 2021.