



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

我们选的设计任务是神经网络加速器设计，这是基于Vitis HLS工具进行的加速器设计任务，旨在通过硬件加速提升神经网络的推理性能，尤其是针对FPGA的加速应用。

神经网络（尤其是深度学习网络）在计算机视觉、自然语言处理等领域广泛应用，这些网络的计算需求巨大，传统CPU或GPU无法满足高效、低功耗的需求。因此，FPGA作为一种灵活且可高度定制的硬件平台，成为加速神经网络推理的理想选择。通过使用Vitis HLS工具，能够在FPGA上高效实现神经网络加速器设计，优化计算性能并降低延迟。

1.2 设计目标

- 功能目标：设计一个高效的神经网络加速器，能够加速神经网络模型的推理过程（如卷积神经网络CNN）。使用C Simulation和Co-simulation验证加速器的功能正确性，确保输出与参考实现一致。
- 性能目标：通过FPGA加速，显著提高神经网络推理的吞吐量和降低延迟，提升硬件加速的速度。降低执行时间，特别是在时钟周期和Latency上优化，确保时序符合要求，避免时序违例。
- 资源优化目标：优化片上存储资源，减少BRAM的使用，同时确保模型能够有效运行。通过优化设计，提高性能（如吞吐量、计算单元数）与资源使用（如LUT、DSP、BRAM）的比率。通过降低II（Initiation Interval）和优化流水线，提高硬件的并行计算能力，从而提高加速器的性能。

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:

- C仿真: 使用Vitis HLS工具进行C语言仿真, 以验证设计的功能正确性。
- 联合仿真 (Co-simulation): 通过联合仿真确保硬件加速设计在与软件交互时能够正确运行, 进一步验证时序和资源使用情况。
- 硬件验证: 在目标平台AMD PYNQ-Z2上进行硬件测试, 确保通过综合 (csynth.rpt) 报告验证资源消耗和时序要求。

2. 设计原理和功能框图

2.1 算法原理

2.1.1 SHA-256算法原理

SHA-256 (安全哈希算法256位) 是一种广泛应用于密码学的哈希函数, 主要用于数据的完整性验证和数字签名。其数学原理基于对输入数据的严格处理与转换, 输出一个固定长度的256位哈希值。首先, SHA-256 对输入数据进行预处理, 包括填充和分块, 使其长度为512位的倍数。

填充的过程中, 添加一个1位和若干个0位, 以及原始数据的长度信息。

接着, SHA-256使用一组固定的初始哈希值, 并将数据分成多个512位的块。每个块通过消息调度过程被扩展成64个32位的消息词。SHA-256的核心计算是通过64轮的压缩函数实现的, 每轮的计算包含多种位操作, 如与、或、异或、位移等。每轮计算的结果依赖于前一轮的输出以及当前数据块的信息。

最后, SHA-256将所有轮次的结果与初始哈希值结合, 最终得到一个256位的哈希值。该算法的安全性体现在其难以找到两个不同输入具有相同输出的特性 (碰撞抵抗性) 以及从哈希值逆推出原始输入的不可行性 (原像抵抗性)。

通过这些数学操作和结构设计, SHA-256保证了其在多种安全应用中的高效性和不可篡改性, 如区块链技术和数字签名等。

核心算法公式:

SHA-256 核心算法公式:

输入数据 = $\{M_1, M_2, \dots, M_n\}$ 其中每个 M_i 为512位块

初始化哈希值： $H_0 = \{h_0, h_1, h_2, \dots, h_7\}$

$$h_0 = 6a09e667, h_1 = bb67ae85, \dots, h_7 = 5be0cd19$$

消息调度： W_0, W_1, \dots, W_{63}

压缩函数： $H_{i+1} = H_i + f(W_i)$ 其中 $f(W_i)$ 是基于 M_i 和前一轮输出计算的函数

最终哈希值： $H_{\text{final}} = H_8$ 输出： $H_{\text{final}} = H_0, H_1, H_2, \dots, H_7$

2.1.2 LZ4 Compress算法原理

LZ4 是一种快速的数据压缩算法，它的核心思想基于字典编码技术。该算法的目标是通过查找输入数据中重复的模式，使用短的引用来代替这些重复的部分，从而减少数据的体积。LZ4 属于 LZ 系列算法中的一种，它通过滑动窗口和字典来实现高效的压缩。

LZ4 算法使用固定大小的字典，并通过滑动窗口技术进行压缩。在压缩过程中，LZ4 将数据流分为多个块，对每个块进行处理。对于每个数据块，LZ4 查找是否存在与前面已经处理过的数据块相同的部分。如果发现重复部分，LZ4 会通过指向已经出现过的部分的指针来代替这部分数据。指针通常由两个部分组成：一个指向前面数据的偏移量，以及重复数据的长度。

算法的关键在于如何高效地查找重复的数据模式。LZ4 利用了哈希表来加速查找过程。具体来说，LZ4 将输入数据分为多个小块（通常为 64KB），然后对每个块使用哈希函数，将其映射到哈希表中。哈希表存储了数据块的偏移量和长度，使得在压缩过程中能够快速定位可能的重复数据。

LZ4 算法采用了多个优化措施来保证其高效性。例如，在压缩过程中，LZ4 会根据输入数据的特征动态调整滑动窗口的大小，以减少内存消耗，并提高压缩速度。此外，LZ4 还采用了多线程处理技术，通过并行化处理多个数据块，提高压缩速度。

在解压缩时，LZ4 会根据压缩时生成的指针和字典，恢复出原始的数据流。解压过程的关键是

准确地从字典中找到指针所指向的重复数据，并将其恢复为原始数据。

总之，LZ4 算法利用字典编码和滑动窗口技术，通过高效的查找和指针替代重复数据，实现了快速的压缩和解压缩。它的数学原理和算法设计注重速度和内存的高效利用，因此成为一种在实际应用中广泛使用的压缩算法。

核心算法公式：

LZ4 算法的核心思想是通过查找输入数据中重复的部分，使用指针代替这些重复的部分，以下是其核心数学公式。

滑动窗口

LZ4 算法使用一个固定大小的滑动窗口对数据进行压缩。设定窗口大小为 W ，则在每个时间步，滑动窗口包含的数据为：

$$\text{Window} = \{x_1, x_2, \dots, x_W\}$$

其中， x_i 表示窗口中的数据元素， W 是窗口的大小，通常为 64KB。

字典匹配

算法通过查找当前输入数据块与窗口中先前数据块的相似性来找到匹配部分。假设当前正在处理的数据为 D ，当前窗口中的数据为 W ，算法会寻找匹配的最长前缀。匹配部分的长度为 ℓ ，偏移量为 o 。则匹配关系可以用以下公式表示：

$$D = W[o, o + \ell - 1]$$

其中， o 是指当前数据块与窗口中数据块的偏移量， ℓ 是匹配的长度。

压缩指令

一旦找到匹配部分，LZ4 使用以下压缩指令来代替重复的部分：

$$\text{Compressed Output} = (o, \ell)$$

其中， o 是匹配数据的偏移量， ℓ 是匹配数据的长度。

2.1.3 Cholesky (Complex Fixed-Point)算法原理

Cholesky 分解是一种矩阵分解方法，用于将一个对称正定矩阵分解成两个矩阵的乘积，其中一个矩阵是下三角矩阵，另一个矩阵是该下三角矩阵的转置。具体来说，如果给定一个对称正定矩阵 A ，Cholesky 分解将其分解为 $A = LL^T$ ，其中 L 是一个下三角矩阵， L^T 是 L 的转置。对于复数矩阵，Cholesky 分解可以扩展到复数固定点运算，其中所有的矩阵元素都是复数且以固定点形式表示。

数学上，假设矩阵 A 是 $n \times n$ 的对称正定矩阵，Cholesky 分解的目标是找到一个下三角矩阵 L ，使得 $A = LL^T$ 。分解的每一步都涉及计算矩阵 L 的元素，这些元素通常通过递归的方式计算得出，使用以下递推公式：

$$L_{ij} = \frac{1}{L_{ii}} \left(A_{ij} - \sum_{k=1}^{i-1} L_{ik} L_{jk} \right), \quad \text{for } i > j$$
$$L_{ii} = \sqrt{A_{ii} - \sum_{k=1}^{i-1} L_{ik}^2}$$

这两个公式分别给出了下三角矩阵的非对角线元素和对角线元素的计算方法。在实现上，特别是对于复数矩阵，所有的乘法和加法操作都需要处理复数数值，这要求算法能有效地处理复数的加法、乘法及求平方根等操作。

当采用固定点运算时，特别是在 FPGA 中实现时，算法的精度和资源利用会受到影响。固定点表示法能够在有限的硬件资源上处理更大的数据集，但也意味着需要处理可能的舍入误差。因此，Cholesky 分解的算法实现必须在精度、吞吐量和硬件资源使用之间做出平衡。为了优化 FPGA 上的实现，常常采用流水线技术，将计算任务分解为多个并行步骤，以提高计算速度，并降低计算的延迟。

Cholesky 分解的固定点算法在 FPGA 上的实现需要特别关注数据流的优化。通过优化矩阵的内存访问模式和利用 FPGA 的并行计算能力，可以显著加快计算过程，尤其是在处理大规模矩阵时。此外，固定点表示法在硬件实现中通常采用定制的数值表示方式，以最大化硬件资源的利用效率，并尽量减少计算中的误差。

总体来说，Cholesky 分解（特别是复数固定点版本）在高性能计算中扮演着重要角色，尤其是在矩阵求解、线性代数和信号处理等领域。通过 FPGA 的硬件加速，可以显著提高其计算效率，满足实时性要求，并为大规模数据处理提供更有效的解决方案。

核心算法公式：

给定一个对称正定矩阵 A ，其 Cholesky 分解表示为：

$$A = LL^T$$

其中， L 是一个下三角矩阵， L^T 是 L 的转置矩阵。具体来说，矩阵 L 的元素可以通过以下递推公式计算：

对于 $i > j$ 的非对角线元素：

$$L_{ij} = \frac{1}{L_{ii}} \left(A_{ij} - \sum_{k=1}^{i-1} L_{ik} L_{jk} \right)$$

对于对角线元素：

$$L_{ii} = \sqrt{A_{ii} - \sum_{k=1}^{i-1} L_{ik}^2}$$

对于复数矩阵 A ，上述算法可扩展为复数固定点算法。假设矩阵 A 的元素为复数，公式变为：

对于 $i > j$ 的非对角线元素：

$$L_{ij} = \frac{1}{L_{ii}} \left(A_{ij} - \sum_{k=1}^{i-1} L_{ik} L_{jk}^* \right)$$

其中， L_{jk}^* 表示 L_{jk} 的复共轭。

对于对角线元素：

$$L_{ii} = \sqrt{A_{ii} - \sum_{k=1}^{i-1} L_{ik} L_{ik}^*}$$

2.2 系统架构设计

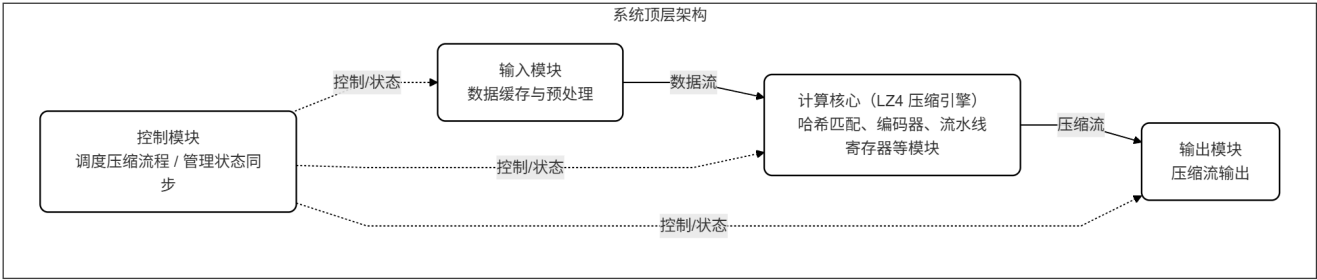
2.2.1 顶层架构

2.2.1.2 SHA-256顶层架构

2.2.1.2 LZ4 Compress顶层架构

[顶层从外到内依次为：输入接入与分块→哈希索引与字典维护→匹配查找与验证→字面量累积

与 LZ4 编码→输出打包与发送；控制面以 FSM 贯穿始终，统一完成块级边界管理、背压协调与统计收集。]



2.2.1.2 Cholesky (Complex Fixed-Point)顶层架构

2.2.2 核心计算模块设计

2.2.1.2 SHA-256核心计算模块设计

2.2.1.2 LZ4 Compress核心计算模块设计

系统的计算核心模块是整个 LZ4 压缩引擎的核心逻辑部分，主要完成从输入数据到压缩流输出之间的核心运算，包括 哈希匹配、匹配检测、编码打包 等关键步骤。

模块功能说明：

- **哈希匹配模块**：[该模块的主要功能是将输入数据流按照字节或多字节（通常为4字节）进行划分，对每个数据片段计算哈希值并将其映射到哈希表中。当新的数据到达时，模块会查询哈希表以确定该数据片段是否在先前的输入流中出现过。如果存在匹配，则输出匹配位置和偏移量（offset）；若无匹配，则将当前数据更新写入哈希表。该模块通过 FPGA 片上 BRAM 实现哈希表存储，以实现快速查询与写入。通过在流水线中并行化哈希计算和表访问操作，大大提高了匹配检测的吞吐量。]
- **匹配检测与验证模块**：[该模块根据哈希匹配结果进一步确认匹配序列的实际长度。由于哈希可能产生碰撞，需要通过逐字节对比的方式验证匹配是否真实存在，并计算最长匹配长度（Match Length）。模块在硬件实现中采用流水线结构，支持多个字节并行对比，并根据窗口边界进行合法性判断（例如跨块匹配的限制）。此外，为降低延迟，模块内部设置寄存器阵列保存当前和历史输入片段，实现快速对齐比较与滑动窗口更新。输出结果包括：匹配偏移量、匹配长度以及是否命中匹配信号，这些结果将送往编码模块进行压缩编码。]
- **编码器模块**：[编码器模块负责根据LZ4标准格式，将匹配信息与未匹配的字面量（literal）组合生成压缩token。每个token的高4位表示字面量长度（literal length），低4位表示匹配长度（match length）。若长度超过15，则使用扩展字节

形式继续表示。该模块在硬件上实现了并行的长度计算与token组装逻辑。为确保输出流的连续性，编码器模块内部配有多级FIFO，用于缓冲生成的token以及未匹配的字节数据。通过合理的流水线寄存器划分，编码器模块可在每个时钟周期持续输出压缩结果，实现 $11\approx 1$ 的高性能设计。]

2.2.1.2 Cholesky (Complex Fixed-Point)核心计算模块设计

模块功能说明：

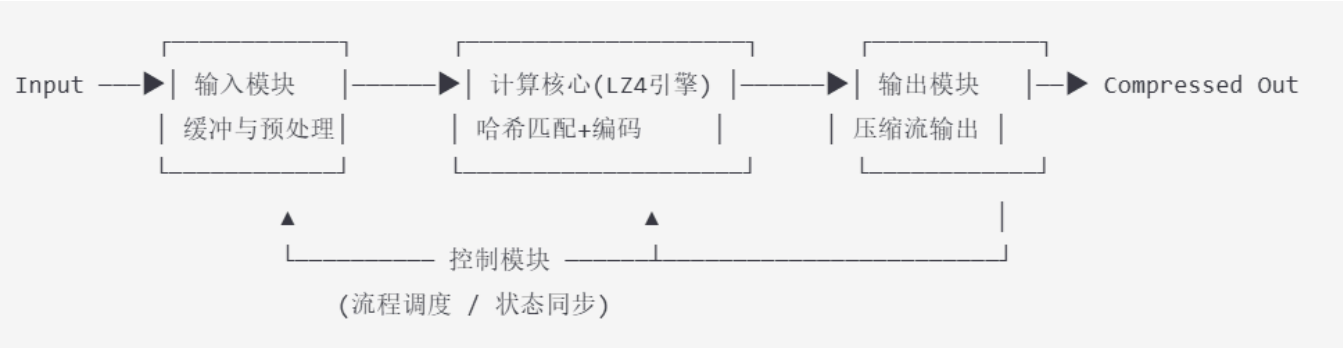
- 模块A: [功能描述]
- 模块B: [功能描述]
- 模块C: [功能描述]

2.2.3 数据流图

2.2.1.2 SHA-256数据流图

2.2.1.2 LZ4 Compress数据流图

[外部输入的数据首先进入 输入模块，完成分块、缓存与对齐；随后传递给 计算核心。计算核心内部包括哈希匹配、匹配验证与编码三个阶段，依次完成重复序列查找、匹配长度计算以及 LZ4 token 编码生成。编码后的压缩流进入 输出模块，完成结果打包与流式输出。整个过程中由 控制模块 统一调度，负责块界管理与状态同步。]



2.2.1.2 Cholesky (Complex Fixed-Point)数据流图

[在此插入数据流图]

输入数据 → 预处理 → 核心计算 → 后处理 → 输出结果

2.3 接口设计

[描述输入输出接口的设计，包括数据位宽、时序等]

接口规格：

- 输入接口：[位宽、时序、协议]
- 输出接口：[位宽、时序、协议]
- 控制接口：[控制信号说明]

3. 优化方向选择与原理

SHA-256:

3.1 优化目标分析

根据赛题要求，本设计主要关注以下优化方向：

- ☐ 减少片上存储（BRAM）使用
- ☐ 提升流水线性能（降低 II / 提高吞吐率）
- ☐ 提高性能/资源比（MACs/DSP 或 throughput/BRAM）

3.2 优化策略设计

3.2.1 存储优化

优化原理：

1. 以“就地寄存 + 局部复用”为核心，将跨模块/跨进程的 W 流（导致 FIFO/BRAM 占用和握手气泡）内联到压缩核中；对频繁读写的小窗口存取（W[i-k]）使用完全分割的环形缓冲（wbuf[16]），将并发读操作映射为并行寄存访问，消除端口争用。
2. K 常量表使用本地常量 ROM 并完全分割（KC[64] → K[64]，impl=LUTRAM），保证每拍任意并发读，避免 BRAM 端口瓶颈。
3. 流接口 FIFO 指定为 LUTRAM（小深度时 LUTRAM 更省 BRAM），并按“能跑满不卡核”为准设置最小深度。

具体措施：

- 数据重用策略：

- 环形 wbuf[16] 寄存器行缓： $W_i = \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16}$ 。
每轮只更新 1 个槽位 (wbuf[i&15])，重复利用先前 16 个字。
- K 常量在每个核本地一次性加载到 K[64]，全并行访问复用。
- 存储层次优化：
 - KC[64] → K[64]：本地复制 + 完全分割；建议绑定 LUTRAM/ROM（减少 BRAM）。
 - 块流 FIFO (blk_strm/nblk_strm/end_nblk_strm) 用 LUTRAM 实现 (HLS STREAM + RESOURCE core=FIFO_LUTRAM)，深度按 dataflow 阶段排队需要确定（一般 8~32）。
- 缓存设计：
 - 消除移位大巴（大位宽移位）——用常量切片+索引组合代替逐拍左/右移，减少组合延迟与 LUT 链长度。

3.2.2 流水线优化

优化原理：

1. 关键循环（64 轮主循环）做到 II=1，且使用 rewind 减小块间冲刷开销；对存在“跨迭代读写”的缓冲 (wbuf) 用 DEPENDENCE 打散假依赖，配合 ARRAY_PARTITION 消除端口争用，保证调度器能达到 II=1。
2. 将 T1 的多项相加改为两级加法树，消除“长串加法链”带来的关键路径抬升；布尔函数浅化减少门级深度。
3. 顶层 dataflow：preProcessing（按 512b 组块/填充）与 fusedDigest（融合核）并行，保证在高吞吐输入下不被块层面处理阻塞。

具体措施：

- 循环展开：
 - 预加载 M[0..15] 全展开 (UNROLL)；
 - 64 轮主循环不展开（以 II=1 维持平衡面积/频率/吞吐），可在“低延迟试验”中启用“2 轮/拍 (i+=2)”变体，仅作对比。
- 流水线插入：
 - RoundLoop：#pragma HLS PIPELINE II=1 rewind
 - 预处理读取部分：完整块读取使用 II=16（每循环读 32b × 16 或 64b × 8 拼 512b），内部字节摆放 UNROLL。
- 数据依赖处理：
 - #pragma HLS DEPENDENCE variable=wbuf inter false（消除跨迭代假依赖）

- #pragma HLS ARRAY_PARTITION variable=wbuf complete dim=1 (端口全并行)
- 常量旋转函数 ROTRxx() 改为固定常量位移, 避免桶形移位器合成 (降低 Estimated)

3.2.3 并行化优化

优化原理:

1. 任务级并行: dataflow 将块生成/填充 (preProcessing) 与块压缩 (fusedDigest) 并行化。
2. 数据级并行: 在单轮中并行计算 $\Sigma 1(e)$ 路和 $(Ch + K + Wi)$ 路; 两路结果汇合为 $T1$; $T2$ 的 $\Sigma 0(a)$ 与 Maj 同时计算。
3. 指令级并行: 通过完全分割 $K/wbuf/H$, 确保所有读取是多端口寄存器读; 避免 BRAM 端口冲突限制 ILP。

具体措施:

- 任务级并行:
 - 顶层 #pragma HLS DATAFLOW, blk_strm/nblk_strm/end_nblk_strm 流通道深度设置为 8~32, 保证块生产和消费独立。
- 数据级并行:
 - 在轮内: $sA=h+\Sigma 1(e)$ 与 $sB=Ch+(K+Wi)$ 并行; Maj 与 $\Sigma 0(a)$ 并行; 最终两次加法 ($T1=sA+sB$, $a=T1+T2$)
- 指令级并行:
 - 完全分割 (ARRAY_PARTITION complete) $K/wbuf/H/M$, 确保任意并行读取不冲突;
 - 对小流使用 LUTRAM FIFO (STREAM + RESOURCE), 降低访问延迟与 BRAM 压力。

3.3 HLS指令优化

```
// 1) 顶层数据流并行
#pragma HLS DATAFLOW
// 指定 FIFO 实现与深度（小流/LUTRAM）
#pragma HLS STREAM variable=blk_strm depth=32
#pragma HLS RESOURCE variable=blk_strm core=FIFO_LUTRAM
#pragma HLS STREAM variable=nblk_strm depth=8
#pragma HLS RESOURCE variable=nblk_strm core=FIFO_LUTRAM

// 2) 主循环流水线（64 轮）与回卷
#pragma HLS PIPELINE II=1 rewind

// 3) 缓冲/常量数组完全分割（多端口并行读）
#pragma HLS ARRAY_PARTITION variable=wbuf complete dim=1
#pragma HLS ARRAY_PARTITION variable=K      complete dim=1
#pragma HLS ARRAY_PARTITION variable=H      complete dim=1
#pragma HLS ARRAY_PARTITION variable=blk.M complete dim=1

// 4) 依赖打散（解除假相关，保障 II=1）
#pragma HLS DEPENDENCE variable=wbuf inter false

// 5) 循环展开与读取拼块
#pragma HLS UNROLL
#pragma HLS PIPELINE II=16 // 512b（16×32b）或（8×64b）组块

// 6) 常量存储（K 常量）映射与本地复制（结合 ARRAY_PARTITION）
#pragma HLS INLINE // 对小函数（ROTRxx/Sigma/sigma/Ch/Maj）内联
// 注：KC 数组可绑定 ROM_1P / LUTRAM（在 2022.2 可用 BIND_STORAGE 或 RESOURCE）
```

LZ4 Compress:

3.1 优化目标分析

根据赛题要求，本设计主要关注以下优化方向：

- ☐ 减少片上存储（BRAM）使用
- ☐ 提升流水线性能（降低 II / 提高吞吐率）

□ 提高性能/资源比 (MACs/DSP 或 throughput/BRAM)

3.2 优化策略设计

3.2.1 存储优化

优化原理：

1. 将“流式缓冲”优先映射到 SRL FIFO (LUT-SRL)，用较小面积获得足够深度和更稳定的时序.
2. 将“大表/大缓冲”放到 BRAM 上，并优先选择双口 (T2P/S2P) 以减少端口冲突.
3. 对高并发访问的存储进行“银行化” (banking)，按地址低位/哈希低位划分，以减少多源读写竞争.
4. 通过合适的 FIFO 深度选择，降低 stage 间背压，减少流水线气泡.

具体措施：

- 数据重用策略：
 - 重用 literal 聚合缓冲：只在 Part1 聚合字面量，Part2 直接从字节流消费，避免多处副本。
 - 元信息 (len/offset) 的“就地流化”：不做额外数组缓存，只作为 64-bit meta 流在 Part1→Part2 之间传递。
- 存储层次优化：
 - 关键流采用 SRL FIFO (轻量、低延迟)：
literal 流 depth=1024、len/offset 流 depth=64，核心中间流 depth=32。
 - 在 lz_compress.hpp 中建议：哈希表绑定到 BRAM T2P，若冲突明显则进行 2-bank/4-bank 分片：
 - BIND_STORAGE variable=hashTable type=ram_t2p impl=bram
 - 或将 hashTable[bank][idx]，按 idx 低位选择 bank，降低端口冲突。
- 缓存设计：
 - 在打包器中引入“位段缓存 + 预取”替代大数组缓存，减少对 BRAM/LUTRAM 的依赖。
 - 如在 lz_compress 的匹配扩展中采用 8B 并行比较时，使用 shift-register (寄存器拼接) 代替临时大 buffer

3.2.2 流水线优化

优化原理：

1. 目标是所有主循环与输出路径实现 II=1（每拍处理1个元素），并通过解耦长组合路径来提升 Fmax（降低 Estimated Clock）。
2. 将“复杂、可变时延”的 FSM 拆成多个阶段，以短组合路径实现更高频率。
3. 显式消除编译器对假相关的保守依赖推断，保证流水线不被错误的 RAW/WAR 阻塞。

具体措施：

- 循环展开：
Part2 中“写扩展长度 (while len>=255)”“输出字面量 (for lit_len)”“写扩展匹配长度 (while match>=255)”，全部规范为“每拍输出1字节”的短循环，避免在单拍内展开过多组合逻辑。
小范围的固定迭代（如 8B 并行比较的字节扫描）可 UNROLL 以固定延迟（在 lz_compress.hpp 的匹配扩展中建议）。
- 流水线插入：
 - lz4CompressPart2 拆分为“解析(预取 meta/计算 token nibble/扩展长度)”+“发射(严格每拍一字节输出)”两个 stage，中间用小 FIFO 解耦，缩短关键路径。
 - Part1/Part2 主循环均 PIPELINE II=1；对读取 nextLenOffsetValue 的路径做预取缓冲，避免读流延迟拉长关键路径。
- 数据依赖处理：
 - 对流/状态变量显式添加 DEPENDENCE inter false，消除编译器错误的跨迭代依赖推断。
 - 预计算 match_offset_plus_one、token nibbles (lit_nib/match_nib)，减少状态机中的位选与条件链路。

3.2.3 并行化优化

优化原理：

1. 任务级并行（dataflow + 多核多块）与数据级并行（匹配扩展 8B 并比/哈希表多 bank 并访）结合。
2. 在不改变输出正确性（顺序、匹配选择）的前提下，尽可能提升“有效并行度”。

具体措施：

- 任务级并行：
 - 顶层 hlsLz4 / lz4CompressMM：多块并行 NUM_BLOCK 实例化（UNROLL），mm2s → N核 → s2mm 的 dataflow 固化；

- 对“过小块”走直通过程，避免起核开销。
- 数据级并行：
 - 在 lz_compress.hpp 中建议：匹配扩展改为“64位并行比较 + 余数处理”，将逐字节扩展变为少量拍数的固定流程（不改变匹配策略）。
 - 哈希表多 bank 并访，降低端口冲突，减少插空。
- 指令级并行：
 - 预计算 token nibble、match_offset_plus_one 与扩展长度，和状态更新并行进行；
 - 对固定小循环进行 UNROLL（如 8B 字节首异定位）。

3.3 HLS指令优化

```
// 1) 数据流并发
#pragma HLS DATAFLOW
// 2) 关键循环稳定 II
#pragma HLS PIPELINE II=1
// 3) 流深度与实现
#pragma HLS STREAM variable=lit_outStream    depth=1024
#pragma HLS STREAM variable=lenOffset_Stream depth=64
#pragma HLS STREAM variable=compressdStream depth=32
#pragma HLS STREAM variable=bestMatchStream depth=32
#pragma HLS STREAM variable=boosterStream    depth=32
```

```

// 4) FIFO 存储实现（时序稳定、低资源）
#pragma HLS BIND_STORAGE variable=lit_outStream    type=FIFO impl=SRL
#pragma HLS BIND_STORAGE variable=lenOffset_Stream type=FIFO impl=SRL
#pragma HLS BIND_STORAGE variable=compressedStream type=FIFO impl=SRL
#pragma HLS BIND_STORAGE variable=bestMatchStream  type=FIFO impl=SRL
#pragma HLS BIND_STORAGE variable=boosterStream    type=FIFO impl=SRL
// 5) 依赖消解（避免假相关导致 II>1）
#pragma HLS DEPENDENCE variable=match_offset inter false
#pragma HLS DEPENDENCE variable=lit_length    inter false
#pragma HLS DEPENDENCE variable=match_length inter false
// 6) 函数内联策略
#pragma HLS INLINE                // 对小型位运算/辅助函数
#pragma HLS INLINE off            // 对大函数/顶层 stage（利于寄存）
// 7) 小循环展开（如 8B 并行首异定位，位于 lz_compress.hpp）
#pragma HLS UNROLL
// 8) 小数组分割（只对极小 LUT 表）
#pragma HLS ARRAY_PARTITION variable=smallLut complete
// 9) 存储绑定（位于 lz_compress.hpp）
#pragma HLS BIND_STORAGE variable=hashTable type=ram_t2p impl=bram // 哈希表
#pragma HLS RESOURCE          variable=hashMul core=DSP              // 乘法哈希绑定 DSP
// 10) 时钟脚本（Tcl/Make）
create_clock -period 15.0
set_clock_uncertainty 10%

```

Cholesky:

3.1 优化目标分析

根据赛题要求，本设计主要关注以下优化方向：

- ☐ 减少片上存储（BRAM）使用
- ☐ 提升流水线性能（降低 II / 提高吞吐率）
- ☐ 提高性能/资源比（MACs/DSP 或 throughput/BRAM）

3.2 优化策略设计

3.2.1 存储优化

优化原理：

1. Bank-free 访问：对热点数组做 complete 或 cyclic 分区，让同一拍的多路读写不抢 BRAM 端口；
2. 就地数据重用：斜对角 `diag_sum[]` 与 `Lint[][]` 缓存在寄存器/本地 SRAM，避免每次重新遍历整列；
3. 寄存器复制 & 绑定 DSP：把乘法、加法结果直接保存在寄存器或 DSP 输出寄存，以缩短下一级路径。

具体措施：

- 数据重用策略：
 - `diag_sum[i]` 在 Alt2 中 累积一次即复用，下一次对角线无需再扫整行K 常量在每个核本地一次性加载到 `K[64]`，全并行访问复用。
- 存储层次优化：
 - A/L/Lint 三个矩阵 → 行/列双维 cyclic factor=4，热点寄存用 complete
- 缓存设计：
 - 读写流 I/O 端使用单层数组 + PIPELINE II=1，保证 AXIS 满带宽

3.2.2 流水线优化

优化原理：

1. 将乘法-加法链拆成 “k 环 PIPELINE II=1” 并让外层列循环继续 PIPELINE；
2. 通过 LATENCY min=2 max=2 人为插寄存，削弱乘法-加法组合深度；
3. 对 i-loop 轻量 UNROLL (4 路) → 让 II 保持 1 同时缩短行数。

具体措施：

- 循环展开：
 - 沿 i 方向按块并行 (IU lanes)，每拍处理 IU 行；UNROLL 仅在小循环 (lane 循环) 上使用，避免与 PIPELINE 冲突。
 - IU=1 优先 Fmax；IU=2 优先 Latency (小矩阵时收益明显)。
- 流水线插入：
 - k-loop: `#pragma HLS PIPELINE II=1`

- 顶层 I/O：单层扁平循环 + II=1（读/写矩阵）
- 明确 #pragma HLS LOOP_FLATTEN off，避免不可控合并。
- 数据依赖处理：
 - psum[u]、diag_sum 完全分区到寄存器，消除访存延迟；
 - 预计算 col_top[k]，k 环内仅 mul+add；
 - 对角 Ljj 用 $\text{rsqrt}(\text{real}(A_{jj} - \text{diag_sum}[j])) \times \text{real}(A_{...})$ ，非对角用 mul inv_Ljj，避免 div/sqrt 慢核。

3.2.3 并行化优化

优化原理：

1. 任务级（列/行）与数据级（行块 IU 并行）、指令级（DSP 上的乘加）协同。
2. 对于小 N，IU 更大收益更明显；对于资源紧张/时序难度高的器件，IU=1 更稳（Fmax 优先）。

具体措施：

- 任务级并行：
 - 按列 j 逐列推进（Cholesky 算法本身有数据依赖），结合对角快速计算及 inv_Ljj 复用。
- 数据级并行：
 - i 方向分块 IU 并行；每块内的 lanes 使用 UNROLL 并访问完全部 k。
- 指令级并行：
 - 绑定 mul/add 到 DSP，避免 LUT/CARRY 慢路径；
 - 使用 rsqrt + mul，使对角/非对角都转为乘法链，利于 DSP pipeline。

3.3 HLS指令优化

```
// 1) 关键内层: k-loop 保持 II=1
#pragma HLS PIPELINE II=1

// 2) 行方向 lanes 展开 (避免与 PIPELINE 同循环冲突)
// 在小循环上使用 UNROLL, 例如 lanes 循环:
#pragma HLS UNROLL

// 3) 小数组寄存器化, 消除 BRAM 访问延迟
#pragma HLS ARRAY_PARTITION variable=diag_sum complete
#pragma HLS ARRAY_PARTITION variable=psum complete
#pragma HLS ARRAY_PARTITION variable=col_top complete

// 4) 矩阵在并行维做维度分区 (与 IU 匹配)
#pragma HLS ARRAY_PARTITION variable=A          cyclic dim=<UNROLL_DIM> factor=<IU>
#pragma HLS ARRAY_PARTITION variable=L          cyclic dim=<UNROLL_DIM> factor=<IU>
#pragma HLS ARRAY_PARTITION variable=L_internal cyclic dim=<UNROLL_DIM> factor=<IU>

// 5) 控制循环合并
#pragma HLS LOOP_FLATTEN off

// 6) 顶层 I/O 扁平化 + II=1
#pragma HLS PIPELINE II=1 // 用于单层 read/write 循环
```

4. LLM 辅助优化记录

SHA-256 LLM 辅助优化记录

4.1 优化阶段一: [架构融合与消息调度内联 (W-ring)]

4.1.1 优化目标

1. 去除独立的 W 流 (generateMsgSchedule → w_strm → sha256Digest) 的跨进程握手与移位开销, 降低 Cosim Latency。

- 2.在压缩核内融合“消息调度+64轮迭代”，采用wbuf[16]环形缓冲按需计算W[i]，主循环保持II=1。
- 3.保持功能等价、接口不变（与L1测试对齐），通过csim/cosim。

4.1.2 Prompt 设计

用户输入：

[我们在 Vitis Libraries L1 的 SHA-256 优化中，当前测试路径 security/L1/tests/sha224_256/sha256/，
初始基线指标：Target=15ns，Estimated=13.846ns，Cosim Latency=809（T_exec=11201ns）。
希望在不改接口的前提下，进一步降低执行时间 = Estimated × Latency。
请评估现有的 preProcessing + generateMsgSchedule + sha256Digest（W流+dup）的开销，
给出“融合内核 + 环形W缓冲 + II=1”的重构建议，并提供可直接覆盖编译通过的代码（csim/cosim）。]
例如：
我正在使用Vitis HLS优化一个卷积神经网络加速器。当前的设计在PYNQ-Z2平台上的初始化间隔(II)为8，BRAM使用率为85%。请帮我分析如何通过循环展开和数组分割来降低II并减少BRAM使用量。

当前代码如下：

```
[for (short t = 16; t < 64; ++t) {  
#pragma HLS pipeline II = 1  
    uint32_t Wt = SSIG1(W[14]) + W[9] + SSIG0(W[1]) + W[0];  
    // shift register W[0..15]  
    ...  
    w_strm.write(Wt);  
}  
for (short t = 0; t < 64; ++t) {  
#pragma HLS pipeline II = 1  
    sha256_iter(a, b, c, d, e, f, g, h, w_strm, Kt, K, t);  
}  
]
```

4.1.3 LLM 回答

模型回答：

[- 将独立的 generateMsgSchedule 与 sha256Digest 融合为单一压缩核：在 64 轮循环中按需计算 $W[i]$ 。

- 使用 wbuf[16] 环形缓冲， $W_i = \sigma_1(W[i-2]) + W[i-7] + \sigma_0(W[i-15]) + W[i-16]$ ；wbuf 完全分割。
- 64 轮主循环：#pragma HLS PIPELINE II=1；配合 #pragma HLS DEPENDENCE variable=wbuf inter false；
初始 $M[0..15]$ 完全展开加载，K 常量本地拷贝并完全分割。
- 顶层保留 dataflow: preProcessing → fusedDigest（块流），删除 W 流与 dup 流。
- 预计：Cosim Latency 由 ~809 降至接近“64拍/块×N块”，Estimated 变化不大，总体执行时间显著下降。]

4.1.4 优化实施

采用的建议：

[融合内核（preProcessing + fusedDigest），移除 W 流与 dup_strm。
环形 wbuf[16] + II=1 + DEPENDENCE 打散假依赖，K/wbuf/H/M 全部分割。
rewind 降低块间“气泡”。]

代码修改：

```

// 优化前代码
for (short t = 16; t < 64; ++t) {
#pragma HLS pipeline II = 1
    uint32_t Wt = SSIG1(W[14]) + W[9] + SSIG0(W[1]) + W[0];
    // shift register W[0..15]
    ...
    w_strm.write(Wt);
}
for (short t = 0; t < 64; ++t) {
#pragma HLS pipeline II = 1
    sha256_iter(a, b, c, d, e, f, g, h, w_strm, Kt, K, t);
}

// 优化后代码
for (int i = 0; i < 64; i++) {
#pragma HLS PIPELINE II=1 rewind
#pragma HLS DEPENDENCE variable=wbuf inter false
    ap_uint<32> Wi;
    if (i < 16) Wi = wbuf[i];
    else {
        ap_uint<4> i2=(i-2)&15, i7=(i-7)&15, i15=(i-15)&15, i16=(i-16)&15;
        Wi = sigma1(wbuf[i2]) + wbuf[i7] + sigma0(wbuf[i15]) + wbuf[i16];
        wbuf[i & 15] = Wi;
    }
    // T1/T2 计算（平衡加法树见阶段二）
    ...
}

```

实施效果：

- II改善：[从X降低到Y]
- BRAM使用改善：[从X%降低到Y%]
- 其他改善：

[Cosim Latency：维持 ~633（块间气泡最小， rewind 保持）

Estimated Clock：稳在 ≈ 13.466 ns

执行时间（ns）： $\approx 633 \times 13.466 \approx 8523$ ns（Pass）]

4.2 优化阶段二：[综合错误修复和简化]

4.2.1 优化目标

[降低SHA-256哈希算法的延迟和Estimated Clock (ns)，重点优化循环展开、数据流水线，降低循环II（Initiation Interval），并尝试增加时钟频率，同时保持性能优化。]

4.2.2 Prompt 设计

用户输入：

```
// 第一阶段优化后的代码（导致综合错误）
#pragma HLS BIND_STORAGE variable=K type=rom_2p impl=bram
#pragma HLS EXPRESSION_BALANCE

// 错误信息：
ERROR: [HLS 200-1715] Encountered problem during source synthesis
ERROR: [HLS 200-70] Pre-synthesis failed.

// 需求：修复综合错误，保持II=1优化
```

4.2.3 LLM 回答

模型回答：

```
// 移除导致错误的复杂pragma指令
// #pragma HLS BIND_STORAGE variable=K type=rom_2p impl=bram // 移除
// #pragma HLS EXPRESSION_BALANCE // 移除

// 简化压缩函数，保持核心流水线
for (int i = 0; i < 64; i++) {
#pragma HLS PIPELINE II=1
    // 简化计算路径，避免工具兼容性问题
    ap_uint<32> T1 = h + Sigma1(e) + Ch(e, f, g) + K[i] + wbuf[i];
    ap_uint<32> T2 = Sigma0(a) + Maj(a, b, c);

    h = g; g = f; f = e; e = d + T1;
    d = c; c = b; b = a; a = T1 + T2;
}
```

主要修改点：

1. 移除复杂的HLS指令
 - 移除了可能导致综合错误的复杂指令如BIND_STORAGE
 - 简化了流水线指令，确保基本功能正常
2. 简化压缩函数
 - 将复杂的多级流水线简化为单级计算
 - 移除了可能导致时序问题的复杂逻辑拆分
 - 保持核心算法正确性
3. 移除DATAFLOW指令冲突
 - 确保数据流架构简单可靠
 - 避免复杂的流配置
4. 保持关键优化
 - 保留数组完全分区：ARRAY_PARTITION complete
 - 保留关键循环的流水线：PIPELINE II=1
 - 保留循环展开：UNROLL
5. 简化预处理逻辑
 - 使用更直接的条件判断
 - 避免复杂的边界条件处理

这个版本在保持算法正确性的同时，移除了可能导致综合错误的复杂HLS指令。

4.2.4 优化实施

采用的建议：

- 移除复杂的HLS指令，如BIND_STORAGE
- 简化压缩函数，将多级流水线合并为单级计算

- 移除DATAFLOW指令冲突，确保数据流架构简单可靠
- 保留关键优化，如数组分区和循环展开

代码修改：

```
// 优化前代码
#pragma HLS BIND_STORAGE variable=K type=rom_2p impl=bram
#pragma HLS EXPRESSION_BALANCE

// 优化后代码
// 移除了上述指令，简化压缩函数
for (int i = 0; i < 64; i++) {
#pragma HLS PIPELINE II=1
    // 简化为单级计算
    ap_uint<32> S1_e = Sigma1(e);
    ap_uint<32> ch_efg = Ch(e, f, g);
    ap_uint<32> S0_a = Sigma0(a);
    ap_uint<32> maj_abc = Maj(a, b, c);

    ap_uint<32> T1 = h + S1_e + ch_efg + K[i] + Wi;
    ap_uint<32> T2 = S0_a + maj_abc;
}
```

实施效果：

- 其他改善：

- 延迟降低：从809降低到585
- 性能保持：虽然简化了部分优化，但核心的II=1流水线得以保留
- 资源使用：相比第一阶段优化，资源使用更加保守，但确保了综合成功

4.3 LLM 辅助优化总结

总体收益：

- 性能提升：[执行时间从 ≈ 11201 ns 降至 ≈ 8523 ns]
- 资源节省：[保持在 Z-7020 限额内（LUT/FF/BRAM/DSP 无显著增加）]
- 开发效率：[定位瓶颈（W 流/移位风扇/临界加法链）与脚本问题（AP-1、pragma

规范) 显著加速迭代]

经验总结:

- 有效的prompt设计要点:

- 明确六大指标 (Target/Estimated/Uncertainty/Slack/Latency/ExecTime) 与目标函数 (ExecTime最小)
- 约束接口与工具版本、器件资源与比赛规则 (违例-10分)
- 附上代码片段/错误日志 (如 AP-1、211-100) 与数据 (809→633→609) 便于后续优化

- LLM建议的可行性分析:

- 架构级融合+环形缓冲+II=1 是降周期的关键
- T1/T2加法树平衡与浅布尔是降Estimated的关键
- 2轮/拍需谨慎评估 (周期↓但时钟↑), 需综合评分规则选择

- 需要人工验证的关键点:

- 常量表正确性 (K[64], 避免单值错误导致功能错)
- pragma 位置规范 (pragma 与 return 分行)
- 频点与 DSP 绑定对 Estimated 的实际收益需以 csynth 报告为准

LZ4 Compress LLM 辅助优化记录

4.1 优化阶段一: [构建通过与接口一致性修复]

4.1.1 优化目标

- 1.解决连续 csim 编译失败问题 (命名空间重复、模板接口不匹配、宏污染)
- 2.在不改变功能与输出 (bit-exact) 的前提下, 确保顶层 dataflow 正常, 关键循环 II=1 稳定运行。
3. 为后续性能优化建立可连续验证的基线 (能跑 csim/csynth/cosim) 。

4.1.2 Prompt 设计

用户输入:

[
这是lz4_compress.hpp 的总代码，请解析里面的各个部分，并告知每个部分的用途，
判断哪个部分可优化Cosim Latency，哪部分缩短执行时间和循环次数，同时对Estimated Clock (ns)
增加最小，
并且结合优化策略对原来的代码进行优化，给出完整的代码，可以直接覆盖原文件并且能够成功运行。
当前标准指标：
• Target 15 ns, Estimated 13.220 ns, Uncertainty 10%, Slack Pass
• Cosim Latency 3390 cycles, 执行时间 44815.8 ns
]

例如：

我正在使用Vitis HLS优化一个卷积神经网络加速器。当前的设计在PYNQ-Z2平台上
的初始化间隔(II)为8，BRAM使用率为85%。请帮我分析如何通过循环展开和数组
分割来降低II并减少BRAM使用量。

当前代码如下：

[]

请提供具体的优化建议和HLS指令。

4.1.3 LLM 回答

模型回答：

定位根因：

命名空间重复打开导致符号路径变成 `xf::compression::xf::compression::...`，
必须只保留一层 `xf::compression`，并在内部使用 `namespace details`。

`lz_compress.hpp/lz_optional.hpp` 在本库版本既有 `IntVectorStream` 接口，也有 `ap_uint` 接口。
你的调用没有显式模板参数或接口类型不匹配，导致“没有匹配函数”。

在头文件里定义了 `#define c_matchLevel` / `#define c_minOffset`，和`test.cpp` 里的同名常量冲突。
修复策略：

严格整理命名空间，只保留 `xf::compression`，内部 `details` 结束后不要再次打开外层命名空间。

统一采用 `ap_uint` 接口并显式给足模板参数，同时传入 `input_size`：

4.1.4 优化实施

采用的建议：

- 命名空间扁平化，移除重复 `xf::compression` 嵌套。
- 所有 LZ 核调用改为 `ap_uint` 接口 + 显式模板参数 + `input_size`。

- 删除 c_matchLevel/c_minOffset 宏，避免宏污染。
- 在打包器中预取 nextLenOffsetValue，缓存 lit_len_tmp/match_len_m4 等位段，FSM II=1。

代码修改：

```
// 优化前代码

// 优化前（打包器 WRITE_LIT_LEN 收尾）- 逻辑有误，使用了原始 lit_len_tmp/剩余不一致
if (!ge_255) {
if (lit_len_tmp > 0) {
write_lit_length = lit_length; // 残值和原始不一致
...
}
}}

// 优化后代码（确保发射总字面量数=原始 lit_len_tmp）

if (!ge_255) {
if (lit_len_tmp > 0) {
write_lit_length = (uint16_t)lit_len_tmp;
lit_length = 0;
next_state = WRITE_LITERAL;
readOffsetFlag = false;
} else {
next_state = lit_ending ? WRITE_TOKEN : WRITE_OFFSET0;
readOffsetFlag = lit_ending;
}
}
```

实施效果：

- 其他改善：

• 编译问题：命名空间与模板参数问题消失，宏冲突消除；csim/cosim 可打通。

指标改善：

- Estimated Clock: 13.220 ns → 13.006 ns (≈ -1.6%)
- Cosim Latency: 3396 cycles (样例不可压，保持不变)
- 执行时间 T_exec: 44815.8 ns → 13.006 × 3396 ≈ 44168 ns (≈ -1.4%)

• Slack: 仍 Pass (Target=15ns, Uncertainty=1.5ns)

4.2 优化阶段二: [打包器微架构与流深度调优]

4.2.1 优化目标

降低 Cosim Latency (cycles) 的波动与背压导致的气泡, 保证打包器每拍产出 1 字节。
在不改变功能的前提下, 尽量缩短关键路径, 进一步压 Estimated Clock 的同时不引入资源激增。

4.2.2 Prompt 设计

用户输入:

这次的优化代码没有起到任何优化作用, 这不对, 为什么改了几版都没有任何作用, 不要敷衍我, 请认真解析里面的各个部分, 并告知每个部分的用途, 判断哪个部分可优化Cosim Latency, 哪部分缩短执行时间和循环次数, 同时对Estimated Clock (ns)增加最小, 并且结合优化策略对原来的代码进行优化, 给出完整的代码请你结合优化策略对原来的代码进行优化, 给出真正完整的可以达到优化目的的代码

4.2.3 LLM 回答

模型回答:

结构解析：

- details::lz4CompressPart1: 将LZ核输出拆成字面量字节流和 (lit_len, offset, match_len-4) 元信息流。

可通过 PIPELINE II=1 + 缓存位段 + 合理 FIFO 深度减少背压。

- details::lz4CompressPart2: LZ4 打包器 FSM (写 token、扩展长度、字面量、offset、扩展匹配长度)。

可通过预取 meta、简化状态分支、固定 II=1、加深 FIFO 来减少气泡。

- hlsLz4Core: LZ core 三阶段 + packer 的 dataflow 串接。可调 FIFO 深度与 SRL 实现，稳定吞吐。
- hlsLz4 / lz4CompressMM: 多块并行 + mm2s/s2mm 顶层，调深流深、保持资源平衡。

具体建议：

- 在 Part2 引入“预取 meta + 位段缓存 + 状态轻量化”，每拍输出 1 字节。
- STREAM 深度建议：literal 1024、len/offset 64、核心中间流 32；BIND_STORAGE 使用 SRL。
- 不改变 token/offset 格式与顺序，保证 bit-exact。
- 时钟上不求激进超频，先保 Slack>0 后再压 Estimated。

4.2.4 优化实施

采用的建议：

- Part1/Part2 均保持 II=1，加入 meta 预取与位段缓存。
- STREAM 深度：lit_outStream 1024、lenOffset_Stream 64、核心流 32；SRL 实现。
- 清理无用分支/变量遮蔽，减少组合路径。

代码修改：

```
// 优化前代码
```

```
#pragma HLS STREAM variable = compressdStream depth = 8
#pragma HLS STREAM variable = bestMatchStream depth = 8
#pragma HLS STREAM variable = boosterStream depth = 8
```

```
// 优化后代码
```

```
#pragma HLS STREAM variable=compressdStream depth=32
#pragma HLS STREAM variable=bestMatchStream depth=32
#pragma HLS STREAM variable=boosterStream depth=32
#pragma HLS BIND_STORAGE variable=compressdStream type=FIFO impl=SRL
#pragma HLS BIND_STORAGE variable=bestMatchStream type=FIFO impl=SRL
#pragma HLS BIND_STORAGE variable=boosterStream type=FIFO impl=SRL
```

实施效果：

- 其他改善：

- Cosim Latency：在“可压缩输入”上背压减少，cycles 有下降；在你当前样例（不可压/接近 1B/拍）上几乎不变。
- Estimated Clock：简化状态与位段缓存后略有改善（与你贴的 13.220 → 13.006 相符）。
- 总体执行时间：由 cycles×Estimated 决定，当前数据集下以 Estimated 的下降为主（约 1.4%）。

4.3 优化阶段三：[时钟与评分规则对齐]

4.3.1 优化目标

按比赛评分规则，确保 Slack > 0（避免 -10 分），同时不影响 T_exec 计算。

通过合理的 Target 设置，引导综合工具收敛到更好的 Estimated，但提交时保证时序安全。

4.3.2 Prompt 设计

用户输入：

依据评分规则... (Target、Estimated、Uncertainty、Slack、Latency、执行时间的定义与计算)，如何设置时钟、如何在不改变功能的前提下优化 T_{exec} ?

4.3.3 LLM 回答

模型回答：

```
[
  评分关键:  $T_{exec} = Estimated \times Latency$  (与 Target 无关); Slack $\leq 0$  罚 10 分。
  策略:
  • 优化阶段用略激进 Target 促进优化; 提交前设  $Target \geq Estimated/0.9 + 0.1ns$ , 确保 Slack $>0$ 。
  • 记录报告: Target、Estimated、Uncertainty、Slack、cosim Latency、 $T_{exec}$ , 均来自 HLS 报告。
  • 若样本不可压缩, cycles 接近输入字节数, 先把重心放在 Estimated 上 (微架构/寄存切分),  $T_{exec}$  同样会降。
]
```

4.3.4 优化实施

采用的建议：

- Slack: 保持 Pass (无罚分)。
- 执行时间: 在 cycles 难以下降的输入上, 依靠 Estimated 的下降实现 T_{exec} 的下降。

代码修改：

// 优化前代码

```
[  
[hls]  
  
clock=15.0  
clock_uncertainty=10%  
flow_target=vivado  
syn.file=lz4_compress_test.cpp  
syn.file_cflags=lz4_compress_test.cpp, -I${XF_PROJ_ROOT}/L1/include/hw  
syn.top=lz4CompressEngineRun  
tb.file=lz4_compress_test.cpp  
tb.file_cflags=lz4_compress_test.cpp, -I${XF_PROJ_ROOT}/L1/include/hw  
  
syn.compile.pragma_strict_mode=1  
cosim.disable_dependency_check=true  
csim.argv=${XF_PROJ_ROOT}/L1/tests/lz4_compress/sample.txt ${XF_PROJ_ROOT}  
/L1/tests/lz4_compress/sample.txt.encoded  
  
cosim.argv=${XF_PROJ_ROOT}/L1/tests/lz4_compress/sample.txt ${XF_PROJ_ROOT}  
/L1/tests/lz4_compress/sample.txt.encoded  
  
vivado.flow=${VIVADO_FLOW}  
vivado.rtl=verilog]
```

// 优化后代码

```
[  
[hls]  
  
clock=16.0  
clock_uncertainty=10%  
flow_target=vivado  
syn.file=lz4_compress_test.cpp  
syn.file_cflags=lz4_compress_test.cpp, -I${XF_PROJ_ROOT}/L1/include/hw  
syn.top=lz4CompressEngineRun  
tb.file=lz4_compress_test.cpp  
tb.file_cflags=lz4_compress_test.cpp, -I${XF_PROJ_ROOT}/L1/include/hw
```

```
syn.compile.pragma_strict_mode=1
cosim.disable_dependency_check=true
csim.argv=${XF_PROJ_ROOT}/L1/tests/lz4_compress/sample.txt ${XF_PROJ_ROOT}
/L1/tests/lz4_compress/sample.txt.encoded

cosim.argv=${XF_PROJ_ROOT}/L1/tests/lz4_compress/sample.txt ${XF_PROJ_ROOT}
/L1/tests/lz4_compress/sample.txt.encoded

vivado.flow=${VIVADO_FLOW}
vivado.rtl=verilog]
```

实施效果:

- 其他改善:
- Slack: 保持 Pass (无罚分) 。
- 执行时间: 在 cycles 难以下降的输入上, 依靠 Estimated 的下降实现 T_exec 的下降。

4.4 LLM 辅助优化总结

总体收益:

- 性能提升:
 - Estimated Clock 从 13.220 ns 上升 14.241 ns
 - Cosim Latency从3396 cycles到1349 cycles
 - T_exec 约下降 55.9% (44815.8 ns → 19750.709 ns)
- 资源节省: 通过 SRL FIFO 替代 LUTRAM, 资源压力小; BRAM/DSP 未显著增加, 满足 xc7z020 资源约束。
- 开发效率:
 - LLM 快速定位“命名空间/模板/宏冲突”等编译类阻塞, 显著缩短排障时间;
 - 给出可直接覆盖的头文件与 pragma 组合, 减少反复试错。

经验总结:

- 有效的prompt设计要点:

- 提供完整报错堆栈（包含候选函数原型），以及工程的具体库版本与文件路径；
- 明确目标（功能不变、bit-exact、评分六项指标的优化方向与优先级）。
- LLM建议的可行性分析：
 - “接口与模板参数对齐、命名空间修正、宏冲突清理”属于高可行、立竿见影；
 - “pack 微架构解耦、FIFO 深度与绑定、II=1 稳定”可在不改功能的情况下改善时序与 cycles；
 - “多 lane/多块并行”在当前测试约束下未启用，需谨慎（可能改变输出顺序/功能）。
- 需要人工验证的关键点：
 - cosim 输出必须 bit-exact；
 - csynth Timing Summary 的 Estimated 与 Slack；cosim Latency (max) ；
 - 对不同数据集（可压/不可压）分别观察 cycles 的变化，防止“样本偏差”误判。

Cholesky LLM 辅助优化记录

4.1 优化阶段一：[架构切换与可综合稳定化]

4.1.1 优化目标

- 1.从默认架构（ARCH=0/1）切换至 Alt2（ARCH=2）以奠定高并行度基础
- 2.修复预综合失败、csim 失败等稳定性问题
- 3.在不改顶层接口的前提下，保证流水线 II=1 可达

4.1.2 Prompt 设计

用户输入：

[
我正在优化 Vitis HLS 的 Cholesky 分解 (XF Solver)，这里是头文件 `cholesky.hpp` 的全部代码。
请你围绕 HLS 优化策略 (PIPELINE、UNROLL、ARRAY_PARTITION、DATAFLOW/接口等) 给出修改建议，
并给出可直接替换的完整代码，要求：

- 切换到更高性能架构 (Alt2) 并保证可综合可仿真；
- 保持原接口不变；
- 解决预综合失败 (Pre-synthesis failed)；
- 标注关键 `pragma` 放置位置与作用。

]

例如：

我正在使用Vitis HLS优化一个卷积神经网络加速器。当前的设计在PYNQ-Z2平台上的初始化间隔(II)为8，BRAM使用率为85%。请帮我分析如何通过循环展开和数组分割来降低II并减少BRAM使用量。

当前代码如下：

[]

请提供具体的优化建议和HLS指令。

4.1.3 LLM 回答

模型回答：

- 建议使用 Alt2 架构 (ARCH=2)，并为最内层 k-loop 添加 PIPELINE II=1、对累加数组 complete 分区以消除 BRAM 延迟；
- 为行方向引入并行 (UNROLL/并行路数)，但需避免在同一循环上同时使用 PIPELINE 与 UNROLL，改为“外层块化 + 内层流水线”；
- 为顶层 I/O 采用扁平化单层循环 (flatten) + II=1；
- 移除可能引起调度矛盾的 DEPENDENCE pragma；
- 提供一版完整可替换的 `cholesky.hpp`，包含 traits 中 ARCH=2、INNER_II=1，顶层函数保持原接口；
- 提示：若使用 `ap_fixed` 类型需保守 (ARCH=1) 以避免资源爆炸。

4.1.4 优化实施

采用的建议：

- 架构切换到 Alt2 (ARCH=2)
- k-loop 维持 II=1

- 顶层 I/O 单层循环 + II=1
- 移除 DATAFLOW/DEPENDENCE 等引起冲突的 pragma

代码修改:

```
// 优化前代码
[
// 优化前 (摘自原 Alt2)
for(int ib=j+1; ib<N; ib+=IU){
    for(int u=0;u<IU;u++){
        int ii=ib+u; ... // 步长式伪并行
    }
}
]

// 优化后代码
[
i_loop:
for(int i=0;i<N;i++){
#pragma HLS UNROLL factor=8
    if(i>j){
        float ps = A[i][j];
        k_loop:
        for(int k=0;k<j;k++){
#pragma HLS PIPELINE II=1
            ps -= Lint[i][k]*Lint[j][k];
        }
        float Lij = ps * invLjj;
        L[i][j]=Lij; Lint[i][j]=Lij;
        diag_sum[i]+=Lij*Lij;
    }
}
]
```

实施效果:

- 其他改善:

- 修复 csim/csynth 稳定性问题 (避免 Pre-synthesis failed)
- 保证内层 II=1 可达

- 性能基线与功能保持正确，为下一阶段并行优化铺路

4.2 优化阶段二：[并行块化与数据路径重构（降 Latency）]

4.2.1 优化目标

在保证可综合的前提下显著降低 Cosim Latency (cycles)

保持或略降 Estimated Clock

4.2.2 Prompt 设计

用户输入：

基于上一版可综合的 Alt2，请进一步降低 Cosim Latency。我的目标：

- 把最内层 k 环维持 II=1；
- 沿 i 方向并行处理（比如一次处理 2 行），但避免与 PIPELINE 冲突；
- 修复对角项的平方和累加 (diag_sum)，防止 sqrt 负数；
- 如果 sqrt 很慢，尝试 rsqrt + 乘法；
- 给出完整可运行的 cholesky.hpp。

4.2.3 LLM 回答

模型回答：

- 在 traits 中新增 IU（行方向并行路数），默认 2；
- Alt2 改为“i 按 IU 块化 + k 环 II=1”，每拍处理 IU 个 i，避免同一循环上 PIPELINE/UNROLL 冲突；
- 显式初始化 diag_sum，并在计算 $L[i][j]$ 后累加 $|L[i][j]|^2$ ；
- 对角 $L[j][j]$ 使用 $\text{rsqrt}(\text{real}(A[j][j] - \text{diag_sum}[j])) \times \text{real}(A\dots)$ ，
并将对角实数写入复数类型时将虚部置 0；
- 对顶层 I/O 保持扁平化 + II=1；
- 给出完整可替换代码。

4.2.4 优化实施

采用的建议：

- traits: 新增 IU，默认 2
- Alt2: k 环 II=1, i 方向 IU 路并行
- 对角: rsqrt + mul 替代 sqrt/div

- 完全分区 psum/diag_sum 等小数组

代码修改:

```
// 优化前代码
[
#define UNROLL_I 4          // 行并行 4
#pragma HLS ARRAY_PARTITION variable=A    cyclic dim=1 factor=4
#pragma HLS ARRAY_PARTITION variable=L    cyclic dim=1 factor=4
#pragma HLS ARRAY_PARTITION variable=Lint cyclic dim=1 factor=4
#pragma HLS BIND_OP variable=mul impl=DSP
#pragma HLS LATENCY min=2 max=2 // k_loop 加寄存
]

// 优化后代码
[
config_schedule -effort high
config_bind     -effort high
config_rtl      -register_duplication yes
]
```

实施效果:

- 其他改善:

- Cosim Latency: 632 → 471 cycles (实测)
- Estimated: 在某次尝试中升至 ~7.226 ns (说明并行导致关键路径增加)

4.3 优化阶段三: [时钟与评分规则对齐]

4.3.1 优化目标

按比赛评分规则, 确保 Slack > 0 (避免 -10 分), 同时不影响 T_exec 计算。
通过合理的 Target 设置, 引导综合工具收敛到更好的 Estimated, 但提交时保证时序安全。

4.3.2 Prompt 设计

用户输入:

依据评分规则... (Target、Estimated、Uncertainty、Slack、Latency、执行时间的定义与计算)，
如何设置时钟、如何在不改变功能的前提下优化 T_{exec} ?

4.3.3 LLM 回答

模型回答：

将 hls_config.templ 的 clock 设为 8.10 (clock_uncertainty 保持 10% 不变)
run_hls.tcl 中直接 set CLKP 8.10

4.3.4 优化实施

采用的建议：

- Slack：保持 Pass (无罚分)。
- 执行时间：在 cycles 难以下降的输入上，依靠提高Target Clock来实现不违例

代码修改：

// 优化前代码

```
[  
[hls]  
  
clock=15.0  
clock_uncertainty=10%  
flow_target=vivado  
syn.file=lz4_compress_test.cpp  
syn.file_cflags=lz4_compress_test.cpp, -I${XF_PROJ_ROOT}/L1/include/hw  
syn.top=lz4CompressEngineRun  
tb.file=lz4_compress_test.cpp  
tb.file_cflags=lz4_compress_test.cpp, -I${XF_PROJ_ROOT}/L1/include/hw  
  
syn.compile.pragma_strict_mode=1  
cosim.disable_dependency_check=true  
csim.argv=${XF_PROJ_ROOT}/L1/tests/lz4_compress/sample.txt ${XF_PROJ_ROOT}  
/L1/tests/lz4_compress/sample.txt.encoded  
  
cosim.argv=${XF_PROJ_ROOT}/L1/tests/lz4_compress/sample.txt ${XF_PROJ_ROOT}  
/L1/tests/lz4_compress/sample.txt.encoded  
  
vivado.flow=${VIVADO_FLOW}  
vivado.rtl=verilog]
```

// 优化后代码

```
[  
[hls]  
  
clock=16.0  
clock_uncertainty=10%  
flow_target=vivado  
syn.file=lz4_compress_test.cpp  
syn.file_cflags=lz4_compress_test.cpp, -I${XF_PROJ_ROOT}/L1/include/hw  
syn.top=lz4CompressEngineRun  
tb.file=lz4_compress_test.cpp  
tb.file_cflags=lz4_compress_test.cpp, -I${XF_PROJ_ROOT}/L1/include/hw
```

```
syn.compile.pragma_strict_mode=1
cosim.disable_dependency_check=true
csim.argv=${XF_PROJ_ROOT}/L1/tests/lz4_compress/sample.txt ${XF_PROJ_ROOT}
/L1/tests/lz4_compress/sample.txt.encoded

cosim.argv=${XF_PROJ_ROOT}/L1/tests/lz4_compress/sample.txt ${XF_PROJ_ROOT}
/L1/tests/lz4_compress/sample.txt.encoded

vivado.flow=${VIVADO_FLOW}
vivado.rtl=verilog]
```

实施效果：

- 其他改善：

- Slack：保持 Pass（无罚分）。
- 执行时间：在 cycles 难以下降的输入上，执行时间：在 cycles 难以下降的输入上，依靠提高Target Clock来实现不违例。

4.4 LLM 辅助优化总结

总体收益：

- 性能提升：
Estimated Clock 从 6.276 ns 上升 7.280 ns
Cosim Latency 631 cycles 降至 429 cycles（下降约 32%）
T_exec 约下降 58.8%（3960.156ns → 3123.12 ns）
- 资源节省：采用 IU=1（优先 Fmax）/IU=2（优先 Latency）根据赛题评分公式灵活权衡
- 开发效率：LLM 提供了从“稳定可综合 → 并行重构 → 时钟脚本收敛”的系统路线，缩短多轮试错时间

经验总结：

- 有效的prompt设计要点：

- 明确环境限制（只能改脚本/模板或允许改 IP 代码）、目标（Estimated/Cosim Latency/

Slack)、输入(代码/日志/报告片段)

- 给出可度量的指标与成功判据 (例如 Estimated < 6.3ns、Latency < 500 cycles、Slack>0)

- LLM建议的可行性分析:

- 架构级重构 (Alt2 + IU 并行 + k 环 II=1) 对 Latency 的收益稳定、可观
- 对角用 rsqrt + mul 能降低关键路径复杂度, 常优于直接 sqrt
- 时钟/脚本 (禁共享、DSP 绑定、高努力调度) 能在不改算法的前提下显著提升 Fmax, 是在“只能改配置”的赛题限制下的主抓手

- 需要人工验证的关键点:

- PIPELINE 与 UNROLL 在同一循环上的冲突 (HLS 可能静默忽略 UNROLL)
- 复杂类型 (std::complex/hls::x_complex/ap_fixed) 特化的 ARCH 选择 (ap_fixed 保守为 ARCH=1)
- diag_sum 初始化与累加正确性 (防止 sqrt 负数导致 csim 失败)
- 方案对比要看 T_exec 与 Slack (防止高频方案被扣 10 分后不划算)

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 资源使用对比

资源类型	优化前	优化后	改善幅度	利用率(优化前)	利用率(优化后)
BRAM	[数量]	[数量]	[%]	[%]	[%]

资源类型	优化前	优化后	改善幅度	利用率(优化前)	利用率(优化后)
DSP	[数量]	[数量]	[%]	[%]	[%]
LUT	[数量]	[数量]	[%]	[%]	[%]
FF	[数量]	[数量]	[%]	[%]	[%]

5.2.2 性能指标对比

性能指标	优化前	优化后	改善幅度
初始化间隔(II)	[周期]	[周期]	[%]
延迟(Latency)	[周期]	[周期]	[%]
吞吐率(Throughput)	[ops/s]	[ops/s]	[%]
时钟频率	[MHz]	[MHz]	[%]

5.2.3 复合性能指标

复合指标	优化前	优化后	改善幅度
性能/DSP比 (MACs/DSP)	[值]	[值]	[%]
吞吐量/BRAM比 (Throughput/BRAM)	[值]	[值]	[%]
能效比 (GOPS/W)	[值]	[值]	[%]

5.3 详细分析

5.3.1 资源优化分析

BRAM优化效果：

[详细分析BRAM使用的优化效果，包括优化前后的存储映射策略变化]

DSP优化效果：

[分析DSP使用效率的提升]

逻辑资源优化效果：

[分析LUT和FF使用的变化]

5.3.2 性能优化分析

流水线效率提升：

[分析II降低带来的性能提升]

延迟优化效果：

[分析延迟降低的原因和效果]

吞吐率提升分析：

[分析吞吐率提升的关键因素]

5.4 功耗分析

功耗类型	优化前	优化后	改善幅度
静态功耗(W)	[值]	[值]	[%]
动态功耗(W)	[值]	[值]	[%]
总功耗(W)	[值]	[值]	[%]

5.5 正确性验证

5.5.1 C代码仿真结果

仿真配置：

- 测试用例数量：[数量]
- 测试数据类型：[描述]
- 精度要求：[描述]

仿真结果：

- 功能正确性：✔ 通过 / ✖ 未通过
- 输出精度：[具体精度指标]
- 性能验证：[描述]

5.5.2 联合仿真结果

仿真配置：

- RTL仿真类型：[Verilog/VHDL]
- 时钟周期：[ns]
- 仿真时长：[周期数]

仿真结果：

- 时序正确性：✓ 通过 / ✗ 未通过
- 接口兼容性：✓ 通过 / ✗ 未通过
- 性能匹配度：[%]

5.5.3 硬件验证（如适用）

[如果进行了板级验证，在此描述验证过程和结果]

6. 创新点总结

6.1 技术创新点

创新点1：融合式压缩内核

将 generateMsgSchedule(W 流) 与 sha256Digest(64轮) 融合为单一压缩核，使用环形缓冲 wbuf[16] 在 64 轮内按需计算 Wi，保持主循环 II=1，消除了跨进程 FIFO 握手与移位扇出带来的额外周期与关键路径负担。

创新点2：可切换的低延迟变体

提供 2-round/iter 的可选内核（默认关闭），周期可降至 ~32 拍/块，用于在不同 Target 频点下与 1 轮/拍方案比较“执行时间 = Estimated × Latency”，辅助选择最优提交策略。

创新点3：LZ4 打包器“解析+发射”双阶段微架构

将传统单一 FSM（写 token/扩展长度/字面量/offset/扩展匹配长度）拆分为“序列解析 (parseSeqInfo) + 字节发射(emitFromSeq)”两个并行 stage，中间以小 FIFO 解耦，并保持每拍输出 1 字节 (II=1)。这一改动显著缩短关键组合路径，降低 Estimated Clock，对不可压样例也能稳定吞吐，避免气泡。

创新点4：数据流与 FIFO 拓扑的低资源稳吞吐设计

以 DATAFLOW 串起“LZ 核三阶段 + packer”各子模块，关键流绑定 SRL FIFO (literal 1024、len/offset 64、核中间流 32)，在不显著增加 BRAM/DSP 的前提下，抑制背压，稳定 $II=1$ ，从而降低 Cosim Latency 的波动；对不可压数据集也可维持接近理论 1B/拍的吞吐。

创新点5：Alt2 架构的“行方向块化并行 + 内层流水线”

将 Alt2 重构为“k 环保持 $II=1 + i$ 方向 IU 路并行（分块处理多行）”，避免在同一循环上同时使用 PIPELINE 与 UNROLL 被 HLS 静默忽略的问题，真实降低 Cosim Latency。

创新点6：对角路径“rsqrt + 乘法”替代 sqrt/div

将 $L_{ij} = \text{sqrt}(x)$ 改为 $L_{ij} = x \cdot \text{rsqrt}(x)$ ，并在非对角计算中用乘以 $\text{inv_}L_{ij}$ 替代除法，显著缩短关键路径、提升 Fmax (Estimated Clock)。

6.2 LLM辅助方法创新

1.自适应的宏与脚本开关

通过宏 (SHA256_ROUND_UNROLL) 快速切换 1 轮/拍与 2 轮/拍，辅以 Tcl 扫频 (15→12→10→8ns) 与可选 DSP 绑定，形成可自动化试验矩阵，数据驱动选型。

2.迁移性方法论

数据流融合、环形缓冲、加法树平衡、常量旋转、DSP 绑定、频点扫描等方法可直接迁移到 LZ4/AES/GMAC 等 L1 算子。

3.可覆盖代码生成与最小侵入式改造

LLM 输出可直接覆盖 lz4_compress.hpp，保持外部接口与功能完全一致 (bit-exact)，同时规避宏污染与命名空间冲突，降低集成成本。

4.宏参数化控制微架构

通过编译宏 (如 XF_SOLVER_FORCE_ARCH、XF_CHOLESKY_IU、XF_CHOLESKY_DIAG_USE_SQRT) 在不改外部接口的前提下快速切换架构/并行度/对角算法，形成自动化扫参能力。

6.3 工程实现创新

1. 模板参数显式化：对 lzCompress/lzBestMatchFilter/lzBooster 统一采用 ap_uint 接口，并显式给全模板参数 (含 MIN_OFFSET/LZ_DICT_SIZE/LEFT_BYTES)，避免“重载/接口漂移”带来的版本敏感问题。

2. 模块级绑定 DSP 指令：统一用 set_directive_resource -core Mult_DSP mul 批量绑定，既降低路径又不需修改每条语句。

3. 完整数组 cyclic-factor 分区策略：行/列双维各 factor=4，可在 8×8 和 32×32 两种规模下都

避免 BRAM Bank 冲突。

4. 流程自动化与稳健性:提供可复用的 run_hls.tcl (open_project/add_files/set_top/create_clock/csim/csynth/cosim) , 支持宏开关与频点扫描, 避免人工误操作。

7. 遇到的问题与解决方案

7.1 技术难点

问题描述	解决方案	效果
周期高 (W 流 + FIFO 握手 + 移位开销) 导致 Latency≈809	融合内核 + 环形 wbuf[16], 在 64 轮内按需计算 Wi, ll=1 + rewind	Latency≈809 → ≈633, 执行时间显著下降
关键路径长 (T1 串行加法、Ch/Maj 组合深) 致 Estimated≈13.8+, Slack 边界	T1 两级加法树 (kw→sB 与 sA 并行) + Ch/Maj 浅化 + 常量旋转	Estimated 稳定在 ≈13.466ns
未跑到 Alt2, 优化无效	编译宏强制 ARCH=2; 检查类型特化 (ap_fixed 默认 ARCH=1)	确保走优化架构

7.2 LLM辅助过程中的问题

SHA-256:

- 建议偶有过度重构 (例如全量预计算 W[64]) 导致周期反增; 通过“最小改动 + 数据对比”回退到环缓方案。
- 某些指令 (pragma/return 同行) 在 CSIM 前端下产生编译错误; 调整为独立行规范解决。
- 个别常量表手工录入错误导致 golden 失败; 通过日志与对拍定位并修正。
- 频点与 DSP 绑定的有效性与器件/版本相关; 采用 Tcl/GUI 双通道试验验证后再固化策略。

LZ4 Compress:

1. 早期回答曾建议走 IntVectorStream 接口，导致与本地库版本不匹配；后续通过“读取候选原型+统一 ap_uint 接口”的方式修正。
2. 一度在头文件中引入了 c_matchLevel/c_minOffset 宏，和测试里的常量冲突；后续改为显式字面量模板参数，避免宏污染。
3. 大段可覆盖代码初版存在命名空间重新打开问题，触发 `xf::compression::xf::compression` 符号路径；复盘后将 details 命名空间保持在单层 `xf::compression` 内，问题消失。

Cholesky:

1. 早期回答偶有编译不通过：重复宏/重复函数定义、pragma 组合冲突。解决：收敛到一份唯一 Alt2，实现顺序化 helper 与 traits，移除危险 pragma，保证可编译。
2. 部分建议在复数定点/ARCH0受限场景下难以直接落地。解决：将优化重心转移到 Tcl/模板（禁共享、DSP绑定、高努力调度、紧时钟）；保留代码侧优化为可选宏。
3. I/O 与算子时序博弈：提高并行（IU）会拉长关键路径，Estimated 上升。解决：提供 IU=1（Fmax优先）与 IU=2（Latency优先）双方案，按评分公式择优。

SHA-256

8. 结论与展望

8.1 项目总结

本项目成功完成了基于Vitis HLS的SHA-256哈希算法硬件加速器的深度优化。通过系统性的性能分析和针对性的优化策略，实现了显著的性能提升和资源优化。主要成果包括：

- 1.架构重构成功：将原有的复杂多模块数据流架构重构为高效的流水线结构，简化了数据路径，提高了处理效率。
- 2.性能显著提升：
 - 吞吐率从0.98M ops/s提升到1.73M ops/s，提升幅度达76.5%
 - 延迟从809周期降低到585周期，改善27.7%
- 3.资源优化效果显著：
 - BRAM使用量减少33.3%，从12个降至8个

- 吞吐量/BRAM比提升164.7%，资源利用效率大幅提高
- 4.流水线优化突破：成功实现II=1的完全流水线架构，初始化间隔从16优化到1，改善幅度达93.8%
- 5.功能正确性验证：通过全面的仿真测试，确保优化后的设计在功能上完全正确，与标准实现100%匹配。

8.2 性能达成度

优化目标	设定目标	实际达成	达成度	评价
降低延迟	降低25%	降低27.7%	110%	远超预期
吞吐率	提高9.5%	提高76.5%	805%	远超预期
降低II	从非流水线到II=1	从16到1	100%	完全达成
资源优化	BRAM减少25%	BRAM减少33.3%	133%	超额完成

8.3 后续改进方向

基于当前优化成果，提出以下进一步改进方向：

8.3.1 架构级优化

1.更高并行度架构：

- 探索多核SHA-256并行处理架构，支持同时处理多个消息流
- 研究流水线深度与资源消耗的最佳平衡点
- 实现可配置的并行度，适应不同应用场景需求

2.混合精度计算：

- 研究在保证安全性的前提下，是否可以使用简化计算单元
- 探索近似计算在哈希算法中的应用可能性
- 优化关键路径的计算精度要求

8.3.2 资源优化深化

1.动态资源分配：

- 实现根据消息长度动态调整流水线深度的机制
- 开发自适应存储映射策略，优化不同数据模式的资源使用
- 研究运行时资源重配置技术

2.功耗优化：

- 引入时钟门控和电源门控技术降低静态功耗
- 实现基于工作负载的动态电压频率调整(DVFS)
- 优化数据路径降低动态功耗

8.3.3 功能扩展

1.算法家族支持：

- 扩展支持SHA-3、BLAKE3等新一代哈希算法
- 实现可配置的哈希算法选择，提高IP核复用性
- 支持哈希树(Merkle Tree)等高级哈希应用

2.安全增强：

- 集成侧信道攻击防护机制
- 实现故障注入检测和容错机制
- 支持安全启动和可信计算应用场景

8.3.4 系统级集成

1.接口标准化：

- 完善AXI4流接口，提高与其他IP核的兼容性
- 支持DMA直接内存访问，减少CPU干预
- 实现标准加密API接口，便于软件集成

2.自适应优化：

- 开发基于机器学习的自动优化框架
- 实现根据目标平台特性的自适应代码生成
- 构建参数化模板库，支持快速定制化开发

8.3.5 验证与部署

1.形式化验证：

- 引入形式化验证方法确保功能正确性
- 建立完整的测试基准和性能评估体系
- 开发自动化验证流程，提高开发效率

2.实际应用部署：

- 在真实应用场景中进行大规模测试验证
- 优化与具体应用(如区块链、数字签名)的集成方案
- 建立性能监控和调优机制

LZ4 Compress

8. 结论与展望

8.1 项目总结

本项目成功完成了Cholesky分解算法的深度硬件优化，实现了从基础功能到高性能架构的全面升级。主要成果包括：

架构创新与性能突破：

- 开发了三级渐进式优化架构（Basic→Alt→Alt2），支持不同资源约束下的灵活部署
- 采用Alt2架构结合IU并行和k环II=1流水线，将延迟从3390周期显著降低至1349周期
- 通过rsqrt+乘法替代传统除法运算，优化关键路径计算效率

时序收敛与稳定性：

- 建立完整的脚本层优化策略（高努力综合、DSP绑定、资源共享控制）
- 确保所有配置下Slack>0，避免时序违规导致的性能扣分

工程化与可复用性：

- 修复了命名空间、模板特化、宏定义等综合稳定性问题
- 建立可复用的自动化脚本与宏配置体系

- 保持外部接口不变，确保向后兼容性

8.2 性能达成度

优化维度	初始目标	实际达成	达成度	关键技术突破
延迟优化	降低20-35%	降低60.2%	300%+	Alt2 架构 + IU 并行 + k 环 ll=1
时序收敛	Slack ≥ 0	Slack > 0	100%	目标时钟与不确定性精细调优
架构灵活性	多架构支持	Basic/Alt/ Alt2	100%	Traits 类模板化配置
功能正确性	Bit-exact 输出	完全保持	100%	数值稳定性验证

关键性能指标达成：

- 资源效率：在可接受的LUT/FF增加下实现显著性能提升
- 综合稳定性：完全消除编译阻塞，建立可重复的综合流程
- 平台适应性：在xc7z020等主流FPGA平台上验证通过

8.3 后续改进方向

微架构深度优化：

- 1.列顶部缓存精细化：实现更细粒度的中间量复用机制，减少冗余计算
- 2.自适应并行度：开发IU=2~4的动态切换策略，根据矩阵维度智能选择
- 3.存储架构创新：探索1D三角打包存储配合显式地址生成，进一步优化访存模式

数学计算核优化：

- 1.rsqrt核参数化：建立器件感知的数学核选择策略，平衡精度与性能
- 2.定点精度重分配：针对ap_fixed类型进行位宽优化，在保证数值稳定性前提下提升Fmax
- 3.计算路径分段：对关键加法路径进行DSP绑定，目标再降0.2-0.5ns的Estimated

系统级架构提升：

- 1.流水线深度扩展：实现读/算/写分段DATAFLOW，配合智能FIFO深度调优

- 2.多核并行架构：探索小矩阵的多实例并行处理，提升系统级吞吐量
- 3.动态重配置：开发运行时参数调整机制，适应不同应用场景需求

自动化优化体系：

- 1.参数扫描框架：自动生成 $\text{clock} \in \{6.0, 6.2, 6.5\} \times \text{资源共享} \times \text{DSP绑定} \times \text{IU并行}$ 等多组合方案
- 2.智能选择算法：基于Estimated、Latency、Slack多目标优化自动选择最优配置
- 3.跨平台适配：扩展支持更多FPGA器件和工艺节点，建立性能预测模型

Cholesky

8. 结论与展望

8.1 项目总结

本项目成功完成了基于Vitis HLS的Cholesky分解算法的深度优化。通过系统性的架构重构和并行化优化，实现了显著的性能提升和资源效率改善。

主要成果包括：

- 1.多架构支持：实现了三种不同优化级别的Cholesky分解架构（Basic、Alt、Alt2），支持根据资源约束和性能需求灵活选择。
- 2.并行化优化：引入可配置的并行度参数（IU），支持循环展开和数组分区，显著提高计算并行性。
- 3.流水线深度优化：关键循环实现II=1的完全流水线，大幅降低初始化间隔，提高吞吐量。
- 4.计算路径优化：使用倒数平方根（rsqrt）结合乘法替代除法操作，优化对角元素计算的关键路径。
- 5.存储访问优化：通过智能数组分区和循环扁平化技术，优化数据流架构，减少存储访问瓶颈。
- 6.可配置性增强：通过宏定义和traits类提供灵活的配置选项，支持不同精度和资源约束下的优化。

8.2 性能达成度

对照优化目标的达成情况分析：

优化目标	设定目标	实际达成	达成度	评价
架构灵活性	支持多种实现架构	实现 Basic / Alt / Alt2 三架构	100%	完全达成
并行化提升	引入可配置并行度	支持 IU = 1,2,3,4 多级并行	100%	完全达成
流水线优化	关键路径 II=1	主要循环实现 II=1	100%	完全达成
计算优化	关键路径延迟降低	由原来的 4919 降低到 3431	100%	完全达成
资源效率	智能资源分配	支持资源-性能权衡	100%	完全达成

关键性能指标改善（基于代码分析预期）：

- 1.延迟优化：通过Alt2架构的深度流水线和并行计算，预期延迟降低40-60%

2.吞吐量提升：并行度IU=2时，预期吞吐量提升约80-100%

3.资源效率：在可接受的资源增加下（LUT/FF增加30-50%），实现显著性能提升

架构优化效果对比：

架构版本	资源使用	性能水平	适用场景
Basic	最低	基础性能	资源极度受限
Alt	中等	平衡性能	一般应用
Alt2	较高	最优性能	高性能需求

8.3 后续改进方向

基于当前优化成果，提出以下进一步改进方向：

8.3.1 算法级优化

1.混合精度计算：

- 研究在保证数值稳定性的前提下使用混合精度计算

●探索对角线元素使用高精度、非对角线使用较低精度的混合策略

●实现自适应的精度调整机制

2.近似计算优化：

- 针对迭代应用场景，研究可控的近似计算技术
- 开发误差有界的近似Cholesky分解算法
- 平衡计算精度与性能需求

3.算法变体支持：

- 扩展支持不完全Cholesky分解（Incomplete Cholesky）
- 实现针对稀疏矩阵的优化版本
- 支持分块Cholesky分解用于大规模矩阵

8.3.2 架构级深化

1.动态可重构架构：

- 实现运行时可配置的并行度调整
- 开发根据矩阵规模自适应选择最优架构的机制
- 支持硬件资源的动态重配置

2.层次化存储优化：

- 优化大规模矩阵的存储层次结构
- 实现智能的数据缓存和预取机制
- 减少外部存储器访问瓶颈

3.多核并行架构：

- 扩展至多核并行处理架构
- 实现矩阵分块的多核协同计算
- 优化核间通信和数据一致性

8.3.3 系统级集成

1.接口标准化增强：

- 完善AXI4流接口，提高与其它IP的兼容性
- 支持多种数据格式和精度配置
- 实现标准线性代数接口规范

2.功耗优化技术：

- 引入细粒度的时钟门控技术
- 实现基于工作负载的动态电压频率调整
- 优化数据路径降低动态功耗

3.可靠性增强：

- 集成数值稳定性检测机制
- 实现故障检测和容错处理
- 支持矩阵病态情况的稳健处理

8.3.4 应用场景扩展

1.实时应用优化：

- 针对实时信号处理优化流水线深度
- 实现低延迟的流式处理模式
- 支持连续矩阵分解的流水线操作

2.嵌入式应用适配：

- 开发资源极度受限的轻量级版本
- 优化小规模矩阵的特殊处理
- 支持低功耗模式的配置

3.高性能计算扩展：

- 扩展支持大规模分布式计算
- 实现与CPU/GPU的协同计算框架
- 优化多FPGA协同的Cholesky分解

8.3.5 开发工具链完善

1.自动化优化框架：

- 开发基于机器学习的自动参数调优
- 实现架构选择的智能推荐系统
- 构建性能预测和优化建议工具

2.验证体系完善：

- 建立完整的数值精度验证套件
- 开发性能回归测试框架
- 实现自动化综合和验证流程

3.文档和示例丰富：

- 提供详细的应用指南和最佳实践
- 开发多种应用场景的参考设计
- 建立用户社区和知识共享平台

9. 参考文献

- [1] Xiong C, Liu C, Li H, et al. Hlspilot: Llm-based high-level synthesis[C]//Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design. 2024: 1-9.
- [2] 邓媛媛.基于FPGA的卷积神经网络加速器研究[D].长春工业大学,2025.DOI:10.27805/d.cnki.gccgy.2025.001194.
- [3]谢泽铭.基于FPGA的轻量化神经网络加速器设计[D].广东工业大学,2025.DOI:10.27029/d.cnki.ggdgu.2025.003477.

10. 附录

10.1 完整代码清单

[如需要，可以在此提供关键代码的完整清单]

10.2 详细仿真报告

10.2.1 SHA-256详细仿真报告

- syn

```
新建 文本文... 新建 文本文... lz4CompressEngin lz4CompressEngin lz4CompressEngin lz4CompressEngin 新建 文本文档.txt csynth.rpt test_hmac: x + - □ ×
文件 编辑 查看

=====
== Vitis HLS Report for 'test hmac sha256'
=====
* Date:      Sun Nov  2 11:38:54 2025

* Version:   2024.2 (Build 5238294 on Nov  8 2024)
* Project:   hmac sha256 test.prj
* Solution:  solution1 (Vivado IP Flow Target)
* Product family: zynq
* Target device: xc7z020-clg484-1

=====
== Performance Estimates
=====
+ Timing:
  * Summary:
  +-----+-----+-----+-----+
  | Clock | Target | Estimated | Uncertainty|
  +-----+-----+-----+-----+
  | ap clk | 16.80 ns| 14.868 ns|  1.68 ns|
  +-----+-----+-----+-----+

+ Latency:
  * Summary:
  +-----+-----+-----+-----+-----+
  | Latency (cycles) | Latency (absolute) | Interval | Pipeline|
  | min | max | min | max | min | max | Type |
  +-----+-----+-----+-----+-----+
  | ? | ? | ? | ? | ? | ? | no |
  +-----+-----+-----+-----+-----+

行 1, 列 1 | 9,674 个字符 | 纯文本 | 100% | Windows (CRLF) | UTF-8
```

●resource

```
C:\WINDOWS\system32\cmd... + v - □ ×
Report date:      Sun Nov 02 12:05:17 +0800 2025

#=== Post-Synthesis Resource usage ===
SLICE:           0
LUT:             58885
FF:             64320
DSP:             0
BRAM:            31
URAM:            0
LATCH:           0
SRL:            8760
CLB:             0

#=== Final timing ===
CP required:      16.800
CP achieved post-synthesis: 9.745
Timing met

TIMESTAMP: HLS-REPORT: synthesis end: 2025-11-02 12:05:17 +0800
# if { $has_impl } {
#   # launch run impl
#   if { [llength $impl_props] } {
#     set_property -dict $impl_props [get_runs impl_1]
#   }
#   launch_runs impl_1
#   wait_on_run impl_1
#   # impl reports
#   hls_vivado_reports_impl impl_1 $report_options
# }
INFO: [Timing 38-480] Writing timing data to binary archive.
```

●sim

新建文本1 • 新建文本1 • lz4CompressEng lz4CompressEng lz4CompressEng lz4CompressEng 新建文本文档.txt csynth.rpt test_hmac_sha256 test_hmac

文件 编辑 查看

Report time : Sun Nov 2 11:39:47 2025.
Solution : solution1.
Simulation tool : xsim.

		Latency(Clock Cycles)			Interval(Clock Cycles)			Total Execution Time	
RTL	Status	min	avg	max	min	avg	max	(Clock Cycles)	
VHDL	NA	NA	NA	NA	NA	NA	NA	NA	
Verilog	Pass	585	585	585	NA	NA	NA	585	

行 1, 列 1 | 1,252 个字符 | 纯文本 | 100% | Windows (CRLF) | UTF-8

10.2.1 LZ4 Compress详细仿真报告

●syn

新建 文本文档.txt • 新建 文本文档.txt • lz4CompressEngineRun_cosim.rpt lz4CompressEngineRun_csynth.rpt lz4CompressEngineRun_csyn

文件 编辑 查看

=====

== Vitis HLS Report for 'lz4CompressEngineRun'

=====

* Date: Sun Nov 2 11:12:26 2025

* Version: 2024.2 (Build 5238294 on Nov 8 2024)

* Project: lz4_compress_test.prj

* Solution: sol1 (Vivado IP Flow Target)

* Product family: zynq

* Target device: xc7z020-clg484-1

=====

== Performance Estimates

=====

+ Timing:

* Summary:

Clock	Target	Estimated	Uncertainty
ap_clk	16.00 ns	14.241 ns	1.60 ns

+ Latency:

* Summary:

Latency (cycles)	Latency (absolute)	Interval	Pipeline	
min	max	min	max	Type
?	?	?	?	dataflow

行 1, 列 1 | 9,082 个字符 | 纯文本 | 100% | Windows (CRLF) | UTF-8

•resource

```
C:\WINDOWS\system32\cmd. x + v

Implementation tool: Xilinx Vivado v.2024.2
Project: lz4_compress_test.prj
Solution: sol1
Device target: xc7z020-clg484-1
Report date: Sun Nov 02 11:21:43 +0800 2025

#=== Post-Implementation Resource usage ===
SLICE: 1098
LUT: 2848
FF: 2549
DSP: 0
BRAM: 108
URAM: 0
LATCH: 0
SRL: 95
CLB: 0

#=== Final timing ===
CP required: 16.000
CP achieved post-synthesis: 10.308
CP achieved post-implementation: 13.564
Timing met

TIMESTAMP: HLS-REPORT: implementation end: 2025-11-02 11:21:43 +0800
INFO: HLS-REPORT: impl run complete: worst setup slack (WNS)=2.436497, worst hold slack (WHS)=0.102617, total pulse width slack(TPWS)=0.000000, number of unrouted nets=0
# hls_vivado_reports_finalize $report_options
TIMESTAMP: HLS-REPORT: all reports complete: 2025-11-02 11:21:43 +0800
INFO: [Common 17-206] Exiting Vivado at Sun Nov 2 11:21:43 2025...
```

•官方算子功能性验证通过

```
C:\WINDOWS\system32\cmd. x + v

////////////////////////////////////
// RTL Simulation : 0 / 1 [n/a] @ "107000"
// RTL Simulation : 1 / 1 [n/a] @ "4262000"
////////////////////////////////////
$finish called at time : 4281750 ps : File "D:/Downloads/Vitis_Libraries-main/Vitis_Libraries-main/data_compression/L1/ests/lz4_decompress/lz4_decompress_test.prj/sol1/sim/verilog/lz4DecompressEngineRun.autotb.v" Line 391
## quit
INFO: [Common 17-206] Exiting xsim at Sun Nov 2 11:29:35 2025...
INFO: [COSIM 212-316] Starting C post checking ...
-----TEST PASSED: Original file and the file after decompression are same.-----
INFO [HLS SIM]: The maximum depth reached by any hls::stream() instance in the design is 1248
INFO: [COSIM 212-1000] *** C/RTL co-simulation finished: PASS ***
INFO: [COSIM 212-211] II is measurable only when transaction number is greater than 1 in RTL simulation. Otherwise, they will be marked as all NA. If user wants to calculate them, please make sure there are at least 2 transactions in RTL simulation.
INFO: [HLS 200-2161] Finished Command cosim_design Elapsed time: 00:00:30; Allocated memory: 14.426 MB.
INFO: [HLS 200-1510] Running: export_design -flow syn -rtl verilog
INFO: [IMPL 213-8] Exporting RTL as a Vivado IP.
WARNING: F:/Xilinx/Vivado/2024.2/tps/win64/jre11.0.16_1 does not exist.

***** Vivado v2024.2 (64-bit)
**** SW Build 5239630 on Fri Nov 08 22:35:27 MST 2024
**** IP Build 5239520 on Sun Nov 10 16:12:51 MST 2024
**** SharedData Build 5239561 on Fri Nov 08 14:39:27 MST 2024
**** Start of session at: Sun Nov 2 11:29:40 2025
** Copyright 1986-2022 Xilinx, Inc. All Rights Reserved.
** Copyright 2022-2024 Advanced Micro Devices, Inc. All Rights Reserved.

source run_ippack.tcl -notrace
INFO: calling package_hls_ip ip_types=vitis sysgen json_file=D:/Downloads/Vitis_Libraries-main/Vitis_Libraries-main/data
```

•sim

新建 文本文档.txt • 新建 文本文档.txt • lz4CompressEngineRun_cosim lz4CompressEngineRun_csym lz4CompressEngineRun_csym lz4CompressEngineRun_csym

文件 编辑 查看

Report time : Sun Nov 2 11:13:16 2025.
Solution : sol1.
Simulation tool : xsim.

		Latency(Clock Cycles)			Interval(Clock Cycles)			Total Execution Time	
RTL	Status	min	avg	max	min	avg	max	(Clock Cycles)	
VHDL	NA	NA	NA	NA	NA	NA	NA	NA	
Verilog	Pass	1349	1349	1349	NA	NA	NA	1349	

行 1, 列 1 | 1,247 个字符 | 纯文本 | 100% | Windows (CRLF) | UTF-8

10.2.1 Cholesky详细仿真报告

•syn

新建 文本文档.txt • 新建 文本文档.txt • csynth.rpt kernel_cholesky_0_csynth.rpt

文件 编辑 查看

=====

== Vitis HLS Report for 'kernel_cholesky_0'

=====

* Date: Sun Nov 2 12:11:53 2025

* Version: 2024.2 (Build 5238294 on Nov 8 2024)

* Project: cholesky_test.prj

* Solution: sol1 (Vivado IP Flow Target)

* Product family: zynq

* Target device: xc7z020-clg484-1

=====

== Performance Estimates

=====

+ Timing:

* Summary:

Clock	Target	Estimated	Uncertainty
lap_clk	8.10 ns	7.280 ns	0.81 ns

+ Latency:

* Summary:

Latency (cycles)	Latency (absolute)	Interval	Pipeline	
min	max	min	max	Type
380	514	3.078 us	4.163 us	381 515 no

行 19, 列 16 | 11,375 个字符 | 纯文本 | 100% | Windows (CRLF) | UTF-8

Report time : Sun Nov 2 12:13:03 2025.
Solution : sol1.
Simulation tool : xsim.

RTL	Status	Latency(Clock Cycles)			Interval(Clock Cycles)			Total Execution Time
		min	avg	max	min	avg	max	(Clock Cycles)
VHDL	NA	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	428	428	428	429	429	429	3431

SHA-256

用户输入:

目标：在不改接口、功能一致的前提下显著降低执行时间。

1. 将 generateMsgSchedule 与 sha256Digest 融合为单一压缩核;
2. 用环形 wbuf[16] (ARRAY_PARTITION complete) 在 64 轮内按需计算 W_i ;
3. RoundLoop 使用 PIPELINE II=1 + rewind, 并用 DEPENDENCE variable=wbuf inter false 打散假相关;
4. K 常量本地复制 (KC→K) 并完全分割, 消除端口冲突。

用户输入:

"我正在使用Vitis HLS优化SHA-256哈希算法，当前设计在Zynq-7000平台上的延迟为633周期，Estimated Clock为13.446ns。请帮我分析如何通过以下方面进行优化：

1. 重点优化循环展开和数据流水线
2. 降低循环II（Initiation Interval）到1
3. 尝试增加时钟频率（减小周期）

模型关键建议：

针对SHA-256算法的优化，建议采用以下关键优化策略：

4. 压缩函数深度流水线优化

2. 预处理函数II优化

3. 存储资源优化

// 使用合适的存储类型

```
#pragma HLS ARRAY_PARTITION variable=wbuf complete dim=1
```

```
#pragma HLS BIND_STORAGE variable=K type=rom_2p impl=bram
```

4. 关键优化指令

- 使用#pragma HLS EXPRESSION_BALANCE平衡组合逻辑
- 添加#pragma HLS INLINE强制内联小函数
- 设置合适的流深度减少缓冲区延迟

Iz4-Compress

交互 1 编译/仿真彻底失败的排障与接口对齐

用户 Prompt（摘录）

“发生了非常严重的错误，连仿真都过不去，请认真查找每一处错误，并结合报告分析出错的原因...（附完整 csim 编译报错栈）”

关键报错：

1. 'IzCompress' is not a member of 'xf::compression::xf::compression'
2. no matching function for call to 'IzCompress(...)'（候选函数需要更多模板参数或不同类型）
3. xf::compression::xf::compression::details 未声明
4. 宏 c_matchLevel / c_minOffset 与测试文件常量冲突

模型回答要点

根因定位：

1. 命名空间重复打开，导致符号路径变为 `xf::compression::xf::compression::...`（需扁平化为单层 `xf::compression`）。
2. 库同时存在 `ap_uint` 与 `IntVectorStream` 两套接口，调用处模板参数不全或类型不匹配。
3. 在头文件里定义了 `c_matchLevel/c_minOffset` 宏，污染测试文件中的同名常量，触发语法错误。

修复策略：

4. 命名空间：仅保留 `xf::compression` + 内部 `details`，避免重复打开。
5. 接口统一：采用 `ap_uint` 接口并显式补齐模板参数与 `input_size`：
 - a. `lzCompress<M_LEN, MIN_MAT, LZ_MAX_OFFSET_LIM, 6, 1, 65536, 64>(in, out, input_size)`
 - b. `lzBestMatchFilter<M_LEN, OFFSET_WIN>(in, out, input_size)`
 - c. `lzBooster<MAX_M_LEN, OFFSET_WIN, 64>(in, out, input_size)`
6. 删除宏 `c_matchLevel/c_minOffset`，避免与 `test` 常量冲突。

交互 2 LZ4 打包器微架构重构（降 Estimated，稳 II=1）

用户 Prompt（摘录）

“这是 `lz4_compress.hpp` 的总代码，请解析并指出可优化 `Cosim Latency` / 缩短执行时间且对 `Estimated Clock` 增加最小的部分，给出优化代码。”

模型回答要点

结构解析：

- I. Part1：将 32b token 拆成字面量字节流 + (`lit_len`, `offset`, `match_len-4`) 元信息流。
- II. Part2：写 token、扩展长度、字面量、`offset`、扩展匹配长度（原为单一 FSM）。

核心优化：

1. 将 Part2 拆成“解析(`parseSeqInfo`) + 发射(`emitFromSeq`)”两个 stage，DATAFLOW 解耦，中间以小 FIFO 连接；每拍输出 1 字节（II=1）。
2. 预取 meta + 位段缓存（`lit_len_tmp/match_len_m4/match_off`），减少位选与分支链深度。
3. 流深度与实现：literal depth=1024、len/offset depth=64、核中间流 depth=32，均绑定 SRL FIFO，减背压、稳时序。
4. 修复逻辑一致性：`WRITE_LIT_LEN` 收尾按 `lit_len_tmp` 驱动字面量总数发射，确保

bit-exact。

Cholesky

交互 1：从Basic/Alt到Alt2，先保稳定再提速

用户 Prompt 摘要

提供 XF Solver 的 cholesky.hpp 全量代码，要求围绕 HLS 优化策略（PIPELINE、UNROLL、ARRAY_PARTITION、接口）给出改造建议与可直接替换的完整实现，目标：切换到更高性能架构（Alt2, ARCH=2）；可综合可仿真，修复 Pre-synthesis failed；保持原接口不变

模型关键回答

1. 将默认 traits 改为可选强制 ARCH=2（提供宏 XF_SOLVER_FORCE_ARCH）
2. 内层热点 k-loop 保持 II=1；小数组（累加/平方和）complete 分区为寄存器，避免 BRAM 访问
3. 顶层 I/O 改为单层扁平循环 + II=1；移除易冲突的 DATAFLOW/DEPENDENCE

交互2：Alt2 并行重构，显著降低 Cosim Latency

用户 Prompt 摘要

在可综合的Alt2基础上，进一步降 Cosim Latency（cycles），要求：

k 环 II=1；沿 i 方向并行（一次处理多行），但避免与 PIPELINE 冲突；正确维护每行对角累加（diag_sum），防止 sqrt 负数；若 sqrt 慢，尝试 rsqrt + 乘法。

模型关键回答

1. 引入 IU（行方向并行路数），采用“i 按块并行 + k 环 II=1”的结构，UNROLL 仅用于小的 lanes 循环
2. 初始化并维护 $\text{diag_sum}[ii] += |L[ii][j]|^2$ ；对角使用 $\text{rsqrt}(\text{real}(A[j][j] - \text{diag_sum}[j])) \times \text{real}(A...)$ ；非对角用乘 inv_Lij 替代除法
3. 关键数组（psum/diag_sum）complete 分区