命题式基础赛道报告

目录

---

1. 项目概述

本报告面向 AMD 命题式基础赛道的 Vitis Libraries L1 算子优化任务，聚焦 `security/sha224_256`、`data_compression/lz4_compress` 与 `solver/cholesky (complex fixed)` 三个代表性算法在 Zynq-7000（xc7z020-clg484-1）平台的高层次综合（HLS）映射与性能优化。我们以"功能正确、资源可控、时序可达、度量严谨"为基本准则，通过结构化的微体系结构设计（数据流化、并行化、循环变换与指令约束）系统性降低执行时间。统一评测指标采用 `T_exec = Estimated_Clock_Period × Cosim_Latency`。在确保 C 仿真与联合仿真结果正确的前提下，我们通过调优时钟约束与流水线并行度，在满足或解释时序状态（Slack）的同时，追求 `T_exec` 的全局最小化。所有优化与结果均遵循可复现的工程流程与报告规范。

1.1 项目背景

> 近年来，面向硬件的算法优化呈现"模型复杂度提升、数据吞吐需求增加、能效约束加强"的趋势。FPGA 以其可重构并行与片上存储优势，成为在受限资源与时序约束下实现高效算子的关键平台。Vitis Libraries L1 算子为算法到硬件的标准化桥梁，但其通用实现通常为可移植性与可综合性折中，留出一定的微结构优化空间。

1.2 设计目标

我们遵循可验证、可度量、可复现的优化原则，分为以下五类目标：

- 功能与接口正确性：
    - 所有算子在 C Simulation（csim）与 Co-simulation（cosim）均需比特级正确，通过官方测试数据与脚手架。
    - 不修改测试框架与外部接口契约（AXI-Stream/指针接口等），仅在算子头文件与时钟配置范围内优化。
- 性能与时序目标：
    - 以执行时间为全局目标函数，最小化该值（单位 ns）。
    - 通过 `PIPELINE/UNROLL/ARRAY_PARTITION/DATAFLOW` 等 HLS 指令与架构设计降低 `Latency`，并在可解释的前提下提升 `Fmax`（降低 `Estimated_Clock_Period`）。
    - 明确记录时序状态（Slack），在可能的违例场景下提供原因分析与改进路径。
- 资源约束与均衡：
    - 保证资源使用（LUT、FF、BRAM、DSP）不超过 xc7z020 器件容量。
    - 针对不同算子在并行度与存储带宽上的需求，进行资源-性能权衡与敏感性分析。

1.3 技术规格

- 平台与器件：
    - 目标平台：Zynq-7000（xc7z020-clg484-1）。
    - 资源约束：不得超出器件容量，违例视为该题失败。
- 工具链与语言：

- 开发工具：Vitis HLS 2024.2（官方要求版本）。
- 语言与类型：C/C++

---

2. 设计原理和功能框图

2.1 SHA-256

2.1.1算法原理

本实现遵循 NIST FIPS-180-4 中的 SHA-224/256 规范[1]，并在代码中以宏与内联函数明确化位运算与消息调度，核心流程为“预处理→消息调度→压缩迭代→摘要输出”，所有阶段通过 HLS 流式接口耦合。

核心算法公式（对应 `sha224_256.hpp` 的宏定义）：

- 基本位运算：

$$\mathrm{ROTR}_n(x) = (x \gg n) \ \vee \ (x \ll (32-n)), \quad \mathrm{SHR}_n(x) = x \gg n$$

$$\mathrm{CH}(x,y,z) = (x \wedge y) \oplus ((\neg x) \wedge z), \quad \mathrm{MAJ}(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

- 大/小 Sigma：

$$\Sigma_0(x) = \mathrm{ROTR}_2(x) \oplus \mathrm{ROTR}_{13}(x) \oplus \mathrm{ROTR}_{22}(x),$$

$$\Sigma_1(x) = \mathrm{ROTR}_6(x) \oplus \mathrm{ROTR}_{11}(x) \oplus \mathrm{ROTR}_{25}(x),$$

$$\sigma_0(x) = \mathrm{ROTR}_7(x) \oplus \mathrm{ROTR}_{18}(x) \oplus \mathrm{SHR}_3(x),$$

$$\sigma_1(x) = \mathrm{ROTR}_{17}(x) \oplus \mathrm{ROTR}_{19}(x) \oplus \mathrm{SHR}_{10}(x).$$

- 消息调度（环形 16 字缓冲，见 `generateMsgSchedule`）：

$$W_t = \sigma_1\big(W_{t-2}\big) + W_{t-7} + \sigma_0\big(W_{t-15}\big) + W_{t-16}, \quad t \in [16, 63],$$

$W_0, \ldots, W_{15}$ 从输入块直接赋值（大端→小端重排后）。

- 每轮压缩迭代（见 `sha256_iter`）：

$$T_1 = h + \Sigma_1(e) + \mathrm{CH}(e,f,g) + K_t + W_t, \quad T_2 = \Sigma_0(a) + \mathrm{MAJ}(a,b,c),$$

$$\begin{aligned} e &\leftarrow d + T_1, \quad a \leftarrow T_1 + T_2, \\ h &\leftarrow g,\ g \leftarrow f,\ f \leftarrow e,\ d \leftarrow c,\ c \leftarrow b,\ b \leftarrow a, \\ K_t &\leftarrow K[(t+1)\ \&\ 63]. \end{aligned}$$

- 块完成后的状态累加：

$$H_i \leftarrow H_i + \{a,b,c,d,e,f,g,h\}_i, \quad i = 0..7.$$

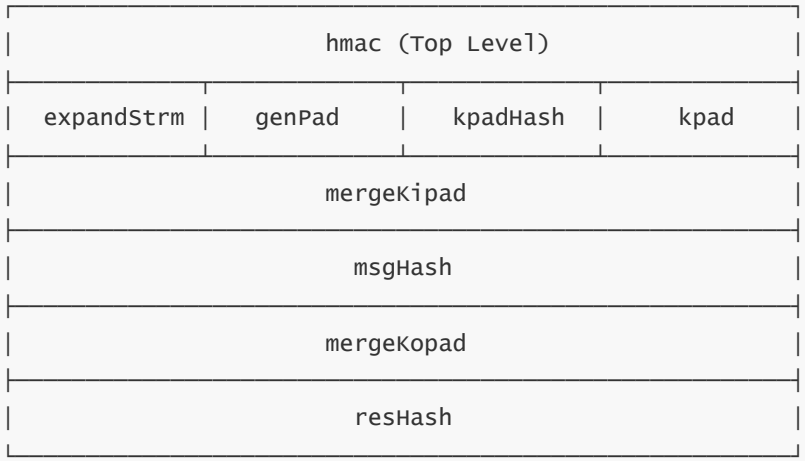
预处理与分块（见 `preProcessing` 两个重载）：输入消息按字节数 `len` 计算比特长度 `L = 8 * len`，块数 `blk_num = (len >> 6) + 1 + ((len & 0x3f) > 55)`。块内进行大端字节重排、追加 `0x80` 填充、零填充至模 `512 ≡ 448`，再在末尾写入 64-bit 的 `L`（高 32 位写 `M[14]`，低 32 位写 `M[15]`）。对 64-bit 输入重载，按 64 位拆分为两个 32 位字并分别重排。

2.1.2 系统架构设计

2.1.2.1 顶层架构

顶层数据流（internal::sha256_top）

顶层 `hmac` 函数是一个模板函数，通过模板参数可以灵活配置数据位宽、密钥长度、哈希算法等。其内部实现了一个名为 `hmacDataflow` 的数据流过程，将整个 HMAC 计算过程分解为多个顺序执行的子模块。



2.1.2.2 核心计算模块设计

- 预处理（`preProcessing`）：
  - 计算块数与末尾剩余字节 `left`，按三种情形生成 1 或 2 个最终块；块内数组 `M[16]` 完全分区。
  - 32-bit 版本"整块"循环以 `pipeline II=16`，尾块拷贝与填充以 `unroll/pipeline` 实现；64-bit 版本按每次 64-bit 输入拆分为两个 32-bit 写入。
- 消息调度（`generateMsgSchedule`）：
  - 首先写出 16 个原始字至 `w_strm`；随后以环形 16 字缓冲计算 (W_t)，`pipeline II=1`，并显式解除对 `blk.M` 的依赖；每次更新同时写新值回缓冲与流。
- 压缩迭代（`sha256Digest` 调用 `sha256_iter`）：
  - 每块 64 轮，以 `PIPELINE II=1` 调度；`sha256_iter` 内部将加法链平衡为多级（`t1_base→t1_mid→T1`），并在不影响状态更新的前提下绑定加法为 Fabric 实现以缩短组合延迟。
  - 完成后进行状态累加；输出阶段将内部 32-bit 状态从大端转换为小端字节序写入 `hash_strm`。

2.1.2.4 HMAC-SHA256 数据流图

```
   |              |
   ├──► kipadStrm (64B*8)
   └──► kopadStrm (64B*8)
        eKipadStrm
              |
              ▼
┌──────────────────────────────┐
| msgHash                      |
| mergeKipad→F.hash            |
└──┬──────────────┬────────────┘
   |              |
   ├──► msgHashStrm,eMsgHashStrm
   └──► kopad2Strm (透传)
              |
              ▼
┌──────────────────────────────┐
| resHash                      |
| mergeKopad→F.hash            |
└──────────┬──────────┬────────┘
           |          |
     hshStrm(256)   eHshStrm
```

2.1.3 接口设计

接口规格：

- 输入接口：
  - `keyStrm`：密钥流。模板为 `hls::stream<ap_uint<8 * K_LEN>>`，本实现按字宽输入为 `hls::stream<ap_uint<32>>`（每拍 4 字节），参数 `K_LEN=32`（字节）。
  - `msgStrm`：消息流，`hls::stream<ap_uint<32>>` 类型（数据位宽 `W=32`）。
  - `lenStrm`：消息长度流，`hls::stream<ap_uint<64>>` 类型（长度位宽 `L=64`，单位为字节）。
  - `endLenStrm`：消息长度结束标志，`hls::stream<bool>` 类型（`false` 表示继续、`true` 表示结束）。
- 输出接口：
  - `digestStrm`：HMAC 结果流，`hls::stream<ap_uint<256>>` 类型（哈希位宽 `H=256`，对应 SHA-256）。
  - `endDigestStrm`：HMAC 结果结束标志，`hls::stream<bool>` 类型（与 `digestStrm` 同步的结束信号）。
- 控制接口：
  - 所有接口均为 AXI-Stream，包含 `tvalid`、`tready`、`tdata`、`tlast` 等信号，由 HLS 自动综合生成；在数据流阶段以 `#pragma HLS dataflow` 与各子模块 `#pragma HLS PIPELINE II=1` 保障逐拍处理与握手稳定性。
- 实现绑定（来自 `security/L1/include/xf_security/hmac.hpp` 与 `sha224_256.hpp`）：
  - 顶层调用为 `xf::security::hmac<32, 64, 256, 32, 64, sha256_wrapper>(...)`，其中参数分别对应 `dataW=32`、`lW=64`、`hshW=256`、`keyLen=32`（字节）、`blockSize=64`（字节）。
  - 底层哈希包装 `sha256_wrapper` 调用 `xf::security::sha256<msgW>(...)`，消息字宽与接口一致（32 位）。

- 输出阶段对摘要字进行端序处理：内部 32-bit 状态从大端转换为小端字节序写入 `digestStrm`，与 AXI-Stream `tdata` 对齐。

2.2 LZ4 压缩算法原理

2.2.1 算法原理

本项目的 LZ4 压缩实现遵循"LZ77 匹配查找 + LZ4 序列封装"的标准流程[3][4]，代码集中于 `data_compression/L1/include/hw` 目录下，并在 `tests/lz4_compress/lz4_compress_test.cpp` 中以流接口进行验证。流水线分为四个阶段：`lzCompress`（字典匹配）、`lzBestMatchFilter`（最佳匹配筛选）、`lzBooster`（匹配增强）与 `lz4Compress`（序列打包），各阶段均以 `#pragma HLS PIPELINE II=1` 实现拍级并行，并在顶层以 `#pragma HLS dataflow` 解耦。

LZ4 序列格式（依据 `lz4_compress.hpp::details::lz4CompressPart2`，参见[3]）：

- 每个序列由令牌 `token`、可选的"字面量长度扩展"（Literal Length Extra）、字面量块（Literals）、2 字节偏移量（Offset，低字节先写）、可选的"匹配长度扩展"（Match Length Extra）组成。
- 令牌为 8-bit：高 4 位编码字面量个数 `L` 的截断值，低 4 位编码匹配长度的截断值 `M-4`。

令牌的构造遵循如下规则（直接对应代码中的位操作）：

$$
\begin{aligned}
&\text{令牌 } T = (\min(L, 15) \ll 4) \mid \min(M-4, 15),\\
&\text{若 } L \geq 15:\ \text{依次写出 } \underbrace{255, \ldots, 255}_{\lfloor (L-15)/255 \rfloor \text{ 次}},\ (L-15) \bmod 255,\\
&\text{若 } M-4 \geq 15:\ \text{依次写出 } \underbrace{255, \ldots, 255}_{\lfloor (M-19)/255 \rfloor \text{ 次}},\ (M-19) \bmod 255.
\end{aligned}
$$

其中，字面量长度扩展与匹配长度扩展的逐 255 进位写法在代码中分别由 `WRITE_LIT_LEN` 与 `WRITE_MATCH_LEN` 两个状态循环实现（每次输出 255 或剩余值，直至耗尽）。

偏移量编码（依据 LZ4 规范[3]）：

$$
D_{\text{write}} = (D_{\text{calc}} + 1), \quad \text{以小端序写出 } D_{\text{write}}[7:0] \text{ 与 } D_{\text{write}}[15:8],
$$

即先写低字节、后写高字节。这里 `D_\text{calc}` 来自上一阶段 `lzCompress` 的输出字段 `match_offset = currIdx - compareIdx - 1`，在 `WRITE_OFFSET0/WRITE_OFFSET1` 状态中通过 `match\_offset++` 恢复为 LZ4 规范的一基距离。

序列消费计数（对应 `inIdx` 的更新逻辑）：

$$
\Delta\text{consumed} = L + M + 4,
$$

其中 `+4` 为 LZ4 标准的匹配最小长度（编码时低 4 位存储的是 `M-4`）。

匹配查找（依据 `lz_compress.hpp::lzCompress`，参见[4]）：

- 字典 `dict[LZ_DICT_SIZE]`（默认 `1<<12=4096`）存储最近出现的 `MATCH_LEN` 字节与索引，采用双端口 BRAM 映射并以 `#pragma HLS ARRAY_PARTITION cyclic factor=16` 提升并行带宽。
- 哈希函数（与 `MIN_MATCH` 相关的移位异或）：

$$
\begin{aligned}
&\text{若 } \text{MIN\_MATCH} = 3:\ h = (p_0 \ll 4) \oplus (p_1 \ll 3) \oplus (p_2 \ll 2) \oplus (p_0 \ll 1) \oplus p_1,\\
&\text{否则：}\ h = (p_0 \ll 4) \oplus (p_1 \ll 3) \oplus (p_2 \ll 2) \oplus p_3,
\end{aligned}
$$

用于从字典中读取候选条目 `dictReadValue` 并写回移位后的新条目 `dictWriteValue`。

- 逐候选比较选择最大匹配长度 `len`，并施加有效性约束（代码条件）：

$$len \geq \text{MIN\_MATCH}, \quad 0 < \text{offset} = currIdx - compareIdx - 1 \leq \text{LZ\_MAX\_OFFSET\_LIMIT},$$
$$\text{offset} \geq \text{MIN\_OFFSET}, \quad \text{对于 } len = 3: \ \text{offset} \leq 4096 \ (\text{否则置零}).$$

最终输出 32-bit 压缩描述字：低 8 位为字面量字节 `tCh`，中 8 位为匹配长度 `tLen`，高 16 位为偏移量 `tOffset`（零值表示无匹配）。

压缩比定义（依据测试代码 `lz4_compress_test.cpp`）：

$$\text{Compression Ratio} = \frac{\text{Input Size}}{\text{Compressed Size}},$$

测试程序读取 `lz4OutSize` 流，打印 `fileSize/outsize`（见 `std::cout << "Compression Ratio: " << (float)fileSize / outsize`）。

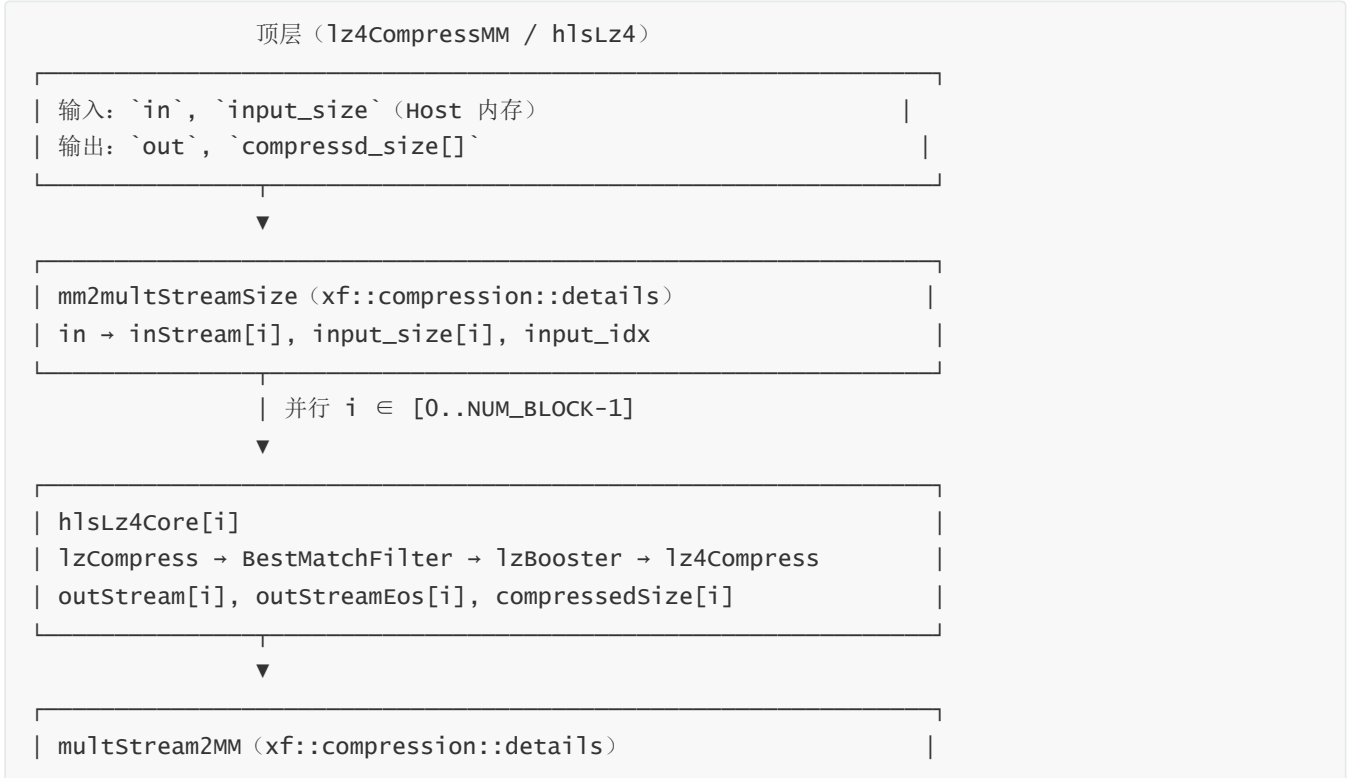
核心算法步骤（严格对应代码调用链）：

1. `lzCompress`：滑窗与哈希字典生成候选，选最大匹配并输出三元组（字面量、匹配长度、偏移）。
2. `lzBestMatchFilter`：移位寄存器窗口对比，若后续位置存在更优匹配则将当前匹配置零（`match_length=0, match_offset=0`）。
3. `lzBooster`：在本地环形存储 `local_mem` 的偏移窗口内尝试延长匹配（受 `MAX_MATCH_LEN` 与窗口大小约束）。
4. `lz4CompressPart1/Part2`：统计字面量段长度并生成 `(lit\_len, match\_len, match\_offset)` 流，随后按 LZ4 令牌与扩展规则写出压缩字节流与结束标志。

2.2.2 系统架构设计

2.2.2.1 顶层架构（内存到流，多核并行）

顶层 `hlsLz4` 与 `lz4CompressMM` 负责将 Host 内存分块（64 KiB）并行压缩并收集结果尺寸（代码见 `lz4_compress.hpp`）：



```
| outStream/outStreamEos/compressedSize → out，输出 output_size |
| 使用 `output_idx` 路由写回                                      |
└──────────────────┬──────────────────────────────────────────┘
                   ▼
┌──────────────────────────────────────────────────────────────┐
| lz4CompressMM 聚合                                           |
| 依据 max_lit_limit[i] / small_block 生成 `compressd_size[i]` |
└──────────────────────────────────────────────────────────────┘
```

核心 `hlsLz4Core` 以 `#pragma HLS dataflow` 串接四个功能模块，保持拍级并行与跨阶段解耦。

2.2.2.2 模块功能说明（代码对应位置）

- `lz_compress.hpp::lzCompress`：
  - 输入：`hls::stream<ap_uint<8>>`；输出：`hls::stream<ap_uint<32>>`。
  - 结构：环形滑窗 `present_window[MATCH_LEN]` 与哈希字典 `dict[LZ_DICT_SIZE]`；循环 `dict_flush` 初始化；主循环 `lz_compress` II=1。
  - 逻辑：计算哈希→读取/更新字典→逐候选比较→约束筛选→输出三元组（字面量/匹配长度/偏移）。
- `lz_optional.hpp::lzBestMatchFilter`：
  - 移位寄存器窗口 `compare_window[MATCH_LEN]` 完全分区；在 II=1 主循环中若发现后续更长匹配则将当前匹配清零（仅保留字面量）。
- `lz_optional.hpp::lzBooster`：
  - 本地环形缓存 `local_mem`（LUTRAM，双端口），流式缓冲 `lclBufStream`；在 II=1 主循环中于偏移窗口范围内尝试匹配延长，受 `MAX_MATCH_LEN` 限制。
- `lz4_compress.hpp::details::lz4CompressPart1`：
  - 将 `ap_uint<32>` 输入分离为字面量流与 `(lit\_len, match\_len, match\_offset)` 流；当 `tLen>0` 时写入 `match_len = tLen - 4`（LZ4 规范），并重置 `lit_count`。
  - 末尾若存在剩余字面量，写入特殊标记（`match_len=0/777`、`match_offset=0/777`）通知第二阶段结束条件。
- `lz4_compress.hpp::details::lz4CompressPart2`：
  - 状态机枚举：`WRITE_TOKEN`、`WRITE_LIT_LEN`、`WRITE_MATCH_LEN`、`WRITE_LITERAL`、`WRITE_OFFSET0`、`WRITE_OFFSET1`；II=1。
  - 令牌写出与长度扩展严格按 LZ4 规范，偏移以小端序写出 `match_offset+1` 的两个字节。
- 顶层封装 `lz4_compress.hpp::lz4Compress` / `hlsLz4Core` / `hlsLz4` / `lz4CompressMM`：
  - 以 `#pragma HLS dataflow` 连接各模块；`mm2multStreamSize` 与 `multStream2MM` 分别完成多流读与写。

2.2.2.3 数据流图

```
                核心数据流（hlsLz4Core）
┌──────────────┐
| inStream(8b) |
└──────┬───────┘
       ▼
┌──────────────────┐
```

```
| lzCompress(32b)   |
└─────────┬─────────┘
          ▼
┌─────────┴─────────┐
| BestMatchFilter|
└─────────┬─────────┘
          ▼
┌─────────┴─────────┐
| lzBooster      |
└─────────┬─────────┘
          ▼
┌─────────┴──────────────────┐
| lz4Compress                |
|  - lit_outStream           |
|  - lenOffset_Stream        |
└─────────┬─────────┬────────┘
          |         |
          ▼         ▼
 outStream(8b)  endOfStream,size
```

注：
- 中间数据流 `compressdStream`/`bestMatchStream`/`boosterStream` 按代码绑定 `FIFO SRL`，`depth=32`；
- `lit_outStream` 与 `lenOffset_Stream` 为 `lz4Compress` 内部 `STREAM`，承载令牌与长度扩展流水；
- 上述拓扑与类型严格对应 `hlsLz4Core`/`lz4_compress.hpp` 定义。

2.2.3 接口设计

- 输入接口：
  - `inStream`：`hls::stream<ap_uint<8>>`，按字节输入待压缩数据，握手遵循 AXI4-Stream 语义（`read`/`write`）。
  - `input_size`：`uint32_t`，当前块输入字节数（用于消费计数与边界控制）。
- 输出接口：
  - `outStream`：`hls::stream<ap_uint<8>>`，压缩后字节流；
  - `outStreamEos`：`hls::stream<bool>`，序列结束标志；
  - `compressedSize`：`hls::stream<uint32_t>`，压缩后字节数（每块）。
- 控制与参数：
  - `max_lit_limit[NUM_BLOCK]`：`uint32_t[]`，字面量段长度阈值；
  - `core_idx`：`uint32_t`，核心索引（并行多核时用于结果路由）。
- 实现绑定（关键流，按代码）：
  - `compressdStream/bestMatchStream/boosterStream`：`STREAM depth=32`，`bind_storage FIFO SRL`；
  - `lit_outStream/lenOffset_Stream`（`lz4Compress` 内部）：`STREAM`，保持令牌与扩展长度流水。

2.3 Cholesky 分解算法原理

2.3.1 算法原理

本实现针对 Hermitian/对称正定矩阵的 Cholesky 分解，严格遵循代码中的数值与架构约束，与数值线性代数中 Cholesky 分解的经典描述一致[9][10]。分解目标是将矩阵 `A ∈ C^{n×n}`（或 `R^{n×n}`）分解为下三角矩阵 `L` 与其共轭转置的乘积：

`A = L * L^*`（Hermitian/对称正定）

其中 (L) 的对角线元素为正实数（由实现强制），上三角元素按配置置零。实现采用列式递推（列索引 (j)）、行索引 (i>j) 的增量法，并对复数域显式使用复共轭（`hls::x_conj`）。严格对应代码（`choleskyAlt`/`choleskyAlt2`）的数学形式：

- 对角线更新（列 (j)）：

  `L[j,j] = sqrt(A[j,j] - sum_{k=0}^{j-1} |L[j,k]|^2)`（对角为正实数）

  在 `choleskyAlt` 中，为降低除法与平方根延迟，先计算对角差 `d_j = A[j,j] - sum_{k<j} |L[j,k]|^2`，随后以倒数平方根近似：

  `r_j ≈ rsqrt(d_j); L[j,j] = d_j * r_j`（缓存 `r_j` 以供后续使用）

  代码中通过 `cholesky_rsqrt(hls::x_real(...), new_L_diag_recip)` 生成 `r_j`，再乘以 `d_j` 构造 `L[j,j]`。若 `d_j < 0` 则返回错误码（输入非正定）。

- 非对角更新（行 (i>j)）：

  `L[i,j] = (A[i,j] - sum_{k=0}^{j-1} L[i,k] * conj(L[j,k])) / L[j,j]`

  在 `choleskyAlt` 中用缓存的 `r_j` 替代除法：

  `S[i,j] = A[i,j] - sum_{k<j} L[i,k] * conj(L[j,k]); L[i,j] = S[i,j] * r_j`

  对复数域，代码显式使用 `hls::x_conj(L_internal[j][k])` 计算 `conj(L[j,k])`，并在构造 `|L[j,k]|^2` 时使用共轭乘法。

上述递推对应 `row_loop / col_loop / sum_loop` 的三重循环结构。为满足数值与时序，代码在复数域强制对角线为实数，避免复平方根；非对角除法用实数倒数乘法替代，降低关键路径延迟。

2.3.2 系统架构设计

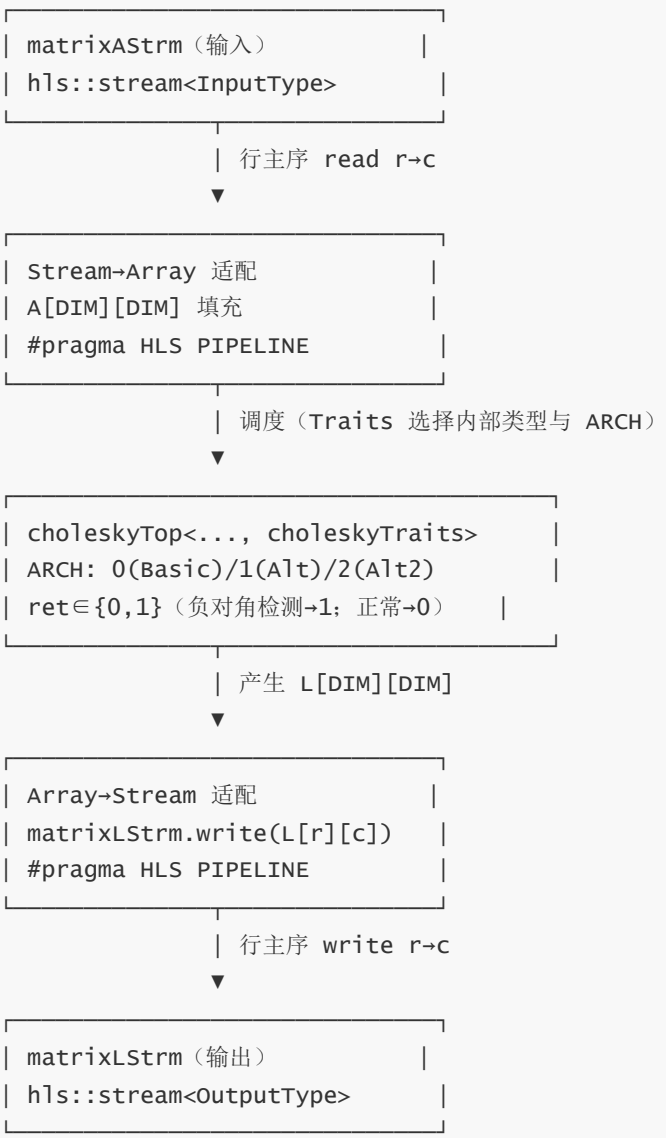
该 HLS 实现以 Traits 模板参数化数值类型与架构选择，核心由三种实现组成：

- `choleskyBasic`（ARCH=0）：基础实现，资源低、延迟较高。
- `choleskyAlt`（ARCH=1）：低延迟实现，使用对角倒数缓存（`diag_internal`）与加法/乘法绑定 DSP，避免除法瓶颈。
- `choleskyAlt2`（ARCH=2）：进一步降低延迟，采用 2D 内部存储与固定界循环，强调流水线 II=1。

在默认 Traits 下选用 ARCH=1，并对复数定点类型提供专门特化（见 `choleskyTraits<...>`），设置 `INNER_II=1` 与合适的 `UNROLL_FACTOR` 来提升并行度。代码大量使用 `ARRAY_PARTITION`、`PIPELINE`、`UNROLL`、`BIND_OP` 与 `EXPRESSION_BALANCE` 指令以缩短关键路径、增强带宽。

2.3.2.1 顶层架构

顶层函数 `cholesky` 作为流式接口入口，负责“流⇄数组”的转化与架构调度。其内部调用 `choleskyTop` 选择 `ARCH` 实现，完成分解后再写回输出流：



其中 `choleskyAlt` 的内部数据路径（列/行/求和三级循环）如下：

```
           列 j（Diagonal）
┌──────────────────────────────────────────────────────────────────────────────┐
| square_sum ← Σ_k |L[j,k]|^2                                                  |
| d_j ← A[j,j] - square_sum                                                    |
| r_j ← rsqrt(d_j)                                                             |
| L[j,j] ← d_j * r_j    （缓存 r_j 到 diag_internal[j]）                        |
└──────────────────────────────────────────────────────────────────────────────┘

           行 i>j（Off-diagonal）
┌──────────────────────────────────────────────────────────────────────────────┐
| S_{i,j} ← A[i,j] - Σ_k L[i,k] * conj(L[j,k])                                 |
| L[i,j] ← S_{i,j} * r_j    （使用缓存的 r_j）                                  |
└──────────────────────────────────────────────────────────────────────────────┘
```

2.3.2.2 核心计算模块设计

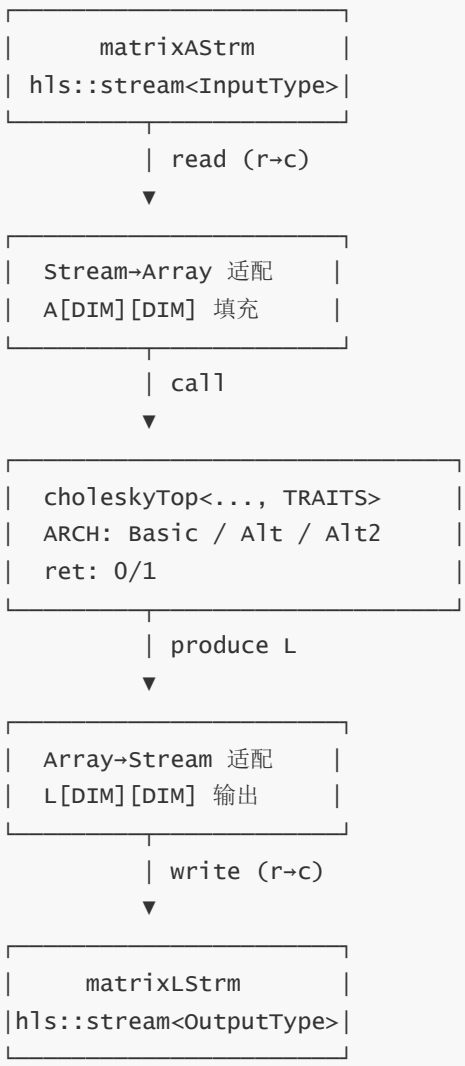
模块功能说明：

- `choleskyBasic`：直接以除法实现 `L[i,j] = S[i,j] / L[j,j]`，结构简单，关键路径包含除法与复乘加，延迟较高。
- `choleskyAlt`：
  - 内部存储：`L_internal[n][n]`（二维存储，分区以增加并发端口），`diag_internal[n]`（寄存器缓存 `r_j`）。
  - 循环结构：`row_loop(i) → col_loop(j<i) → sum_loop(k<j)`，在 `sum_loop` 中以 `-L[i,k] * conj(L[j,k])` 累加形成 `S[i,j]`。
  - 数值策略：对角采用倒数平方根近似（`cholesky_rsqrt`）以乘法替代除法；复数乘法/加法绑定到 DSP（`#pragma HLS BIND_OP op=mul/add impl=DSP`）。
- `choleskyAlt2`：
  - 固定界循环与二维存储以避免复杂索引；将非对角行和的构建与最终更新分离，保证 II=1；保留零化上/下三角的后处理循环以提高主计算的并行度。

辅助函数模块（`x_matrix_utils.hpp` 与本文件）：

- `x_sqrt` / `x_rsqrt`：针对 `half/float/double/ap_fixed` 的统一封装；`ap_fixed` 路径调用 HLS 对应函数以降低延迟。
- `x_rsqrt_refined`：倒数平方根的一次 NR 校正（`y <- y * (1.5 - 0.5 * x * y^2)`），以浮点初值快速逼近，随后在定点域修正，提升数值稳定性与精度；内部乘加绑定 DSP。
- `cholesky_sqrt_op`：复数对角只取实部开方，虚部置零（对角恒为实）。
- `cholesky_rsqrt`：对负值钳位到小正数 `eps`，调用 `x_rsqrt` 或精炼版实现，返回对角的实倒数平方根。
- `cholesky_prod_sum_mult`：复数×实数乘法内联，实/虚部分离并行计算（绑定 DSP），支持复数定点组合运算。

2.3.2.3 数据流图



注（严格对应代码与测试）：

- 顶层接口：`int cholesky(hls::stream<InputType>& matrixAStrm, hls::stream<OutputType>& matrixLStrm)`；
- 读/写顺序：行主序（`r→c`），`host/test_cholesky.cpp` 向 `matrixAStrm` 按 `for r→for c` 写入，与顶层 `read` 顺序一致；输出按相同顺序读出；
- 阶段约束：读/写环节应用 `#pragma HLS PIPELINE`；核心由 `choleskyTop<..., TRAITS>` 选择 `ARCH`（Basic/Alt/Alt2），实现细节见上一节；
- 无显式 `DATAFLOW` 拆分，三阶段按照“流→数组→核心→数组→流”的顺序衔接；
- 错误码：当 `A` 非正定时返回 `ret=1`（对角检测与 `rsqrt` 钳位逻辑），正常返回 `ret=0`。

2.3.3 接口设计

- 输入接口
  - `matrixAStrm`：`hls::stream<MATRIX_IN_T>`，按行主序写入 `DIM×DIM` 元素。
  - 流握手：遵循 AXI-Stream 语义（`valid/ready`），在 HLS 中以 `hls::stream` 实现。
  - 建议深度：≥ `DIM`（仿真/综合可根据资源调整）。

- 输出接口
  - `matrixLStrm`：`hls::stream<MATRIX_OUT_T>`，按行主序输出 `DIM×DIM` 元素。
  - 流握手：同上。
  - 建议深度：≥ `DIM`。
- 控制接口
  - 维度与三角配置：`MATRIX_DIM`、`MATRIX_LOWER_TRIANGULAR`（编译期宏）。
  - 架构选择：`SEL_ARCH`（0/1/2），在 Traits 中决定 `choleskyBasic/Alt/Alt2`，默认 1（低延迟）。
  - 性能指令：`PIPELINE(II=1)`、`UNROLL`、`ARRAY_PARTITION`、`BIND_OP(DSP)` 等在实现内设置。
- 实现绑定
  - 流接口：综合后映射到 AXI4-Stream；数组到流/流到数组作为适配单元。
  - 内部存储：`L_internal` 绑定到 BRAM/寄存器（依据分区策略），`diag_internal` 绑定寄存器。
  - 乘加单元：复数乘法/加法绑定到 DSP，提高吞吐与数值稳定性。

3. 优化方向选择与原理

3.1 优化目标分析

根据赛题评分规则,本设计主要关注以下优化方向:

- 降低Total Execution Time (执行时间 = 时钟周期 × Latency)
- 提升流水线性能(降低 II / 提高吞吐率)
- 优化内存访问模式(减少访存冲突)
- 保持时序满足要求(Slack ≥ 0)
- 控制资源使用在器件容量内

3.2 SHA-256算法优化策略设计

本节基于 baseline 的 HMAC/SHA-256 数据流实现与，围绕三条主线提出系统化优化策略：

- 存储层次与流缓冲匹配：按访问速率与并发需求选择 `LUTRAM/SRL/BRAM` 并调优 FIFO 深度，使阶段间数据流在 `preProcessing/dup_strm/generateMsgSchedule/sha256Digest` 与 HMAC 的 `kpad/msgHash/resHash` 中稳定解耦。
- 关键循环的流水线化与依赖解除：在消息调度与 64 轮压缩迭代中保持 `II=1`，通过显式 `dependence` 解除与适度 `unroll` 分摊组合负载，确保逐拍处理连续推进。
- 表达式平衡以缩短关键路径：将 `T1/T2` 与 `W_t` 的多操作数相加重写为分级加法树，先行计算 `BSIG/MAJ/CH/σ`，并就地环缓冲更新，显著降低组合深度。

总体目标是在不改变接口契约与算法语义的前提下，提升吞吐并降低执行时间。后续 3.2.1–3.2.4 分别给出瓶颈定位、具体存储优化、流水线优化与关键路径缩短的实施细节，均与源码中的 pragma 与函数结构一一对应。

3.2.1 瓶颈分析

baseline运行后根据csynth.rpt,进行逐函数与循环级别的审阅，瓶颈来源清晰集中在“消息调度与压缩迭代”的两级流水，以及 HMAC 两次哈希的级联。

- 计算结构（依据 `sha224_256.hpp` 与 `hmac.hpp`）：
  - baseline 采用数据流架构：`preProcessing → dup_strm → generateMsgSchedule → sha256Digest`；在 HMAC 中通过 `internal::hmacDataflow` 串接 `kpad → msgHash → resHash`。两次哈希串行，使整体吞吐受限于宏观数据流中的最慢阶段。
  - `sha256Digest` 的核心循环 `LOOP_SHA256_UPDATE_64_ROUNDS` 为 64 轮迭代，且每轮的工作变量 `a,b,c,d,e,f,g,h` 更新存在严格的循环依赖；baseline 版本的 `sha256_iter` 中 `T1 = h + BSIG1(e) + CH(e,f,g) + Kt + Wt` 为五操作数串行加法链，是关键路径的主要来源。
  - `generateMsgSchedule` 在 baseline 中通过长度为 16 的环形缓冲 `W[16]` 生成 `W_t`，每拍存在 `W[t-2], W[t-7], W[t-15], W[t-16]` 的依赖；同时将新值写入流与环形缓冲，读写的组合控制对时序构成压力。
- 存储与缓冲（依据 baseline 的 `STREAM/RESOURCE`）：
  - baseline 对多处流使用 `FIFO_LUTRAM` 且深度较浅（例如 `blk_strm/nblk_strm/w_strm`，默认或 32），阶段解耦能力有限，在消息较长或 HMAC 两次哈希级联时易产生背压。
  - HMAC 的数据拼接阶段（`mergeKipad`/`mergeKopad`）在 baseline 中部分使用 LUTRAM，写入链路为大小端转换与字节重排，循环仅 `pipeline`，未做针对性的展开，导致单拍内吞吐受限。
- 结论：在不改变算法结构的前提下，瓶颈由三类因素叠加：
  1. 压缩迭代中的长加法链与严格数据依赖；
  2. 消息调度的环缓冲读写与组合控制；
  3. 流缓冲深度与存储类型选择不匹配所致的阶段耦合与背压。

3.2.2 存储优化

优化原则是“按访问模式与速率选择存储实现，按阶段耦合度设置流深度”，并尽量在本地环形存储内完成近邻数据重用，降低跨阶段读写。

- 顶层与阶段流的存储绑定（依据 `sha224_256.hpp::internal::sha256_top`）：
  - `blk_strm` 显式绑定为 `fifo(lutram)` 且深度由 32→64（`#pragma HLS STREAM` + `bind_storage`）[5]，降低 `preProcessing → generateMsgSchedule` 间的背压。
  - `nblk_strm/nblk_strm1/nblk_strm2` 与 `end_nblk_strm/...` 统一绑定为 `fifo(lutram)` 且深度设置为 8，避免在消息块统计分发时出现早停；`w_strm` 深度由 32→128，提高 `schedule → digest` 的解耦能力。
- 消息调度的局部存储改造（依据 `generateMsgSchedule`）：
  - 用 `blk.M`（完全分区）承载 16 字的环形缓冲，按位掩码索引 `blk.M[(i-16)&15]` 等直接就地更新（`blk.M[i & 15] = wt`），不再另建独立 `W[16]`；降低一次复制与分配开销，且环内数据访问均走寄存器/分散 LUTRAM。
- HMAC 阶段的流与存储（依据 `hmac.hpp::internal::hmacDataflow/mergeKipad/mergeKopad`）：
  - 小深度、窄数据的阶段流（`eKipadStrm/kipadStrm/kopadStrm/kopad2Strm/msgHashStrm/eMsgHashStrm`）统一绑定为 `fifo(srl)`，用移位寄存器降低 LUT 负载与访问延迟。

    - 大数据搬移的合并流（`mergeKipadStrm/mergeKopadStrm`）采用 `FIFO_BRAM`，且在 `mergeKopad` 中将深度调至 2，以“就近写入→就近读出”的方式缩短驻留时间、减少资源浪费。
- 常量与状态数组（两版本一致或增强）：
    - `K[64]` 与 `H[8]` 完全分区（`array_partition complete`），保持多端口并发读；工作变量 `a,b,c,d,e,f,g,h` 在 v2 中显式绑定到寄存器（`bind_storage variable=... type=register`），消除潜在的存储访问路径。

总体效果：阶段间的背压显著降低，消息调度与摘要计算两阶段解耦增强；合并/拼接模块在合适的存储类型上运行，带宽与时延匹配度更好。

3.2.3 流水线优化

优化目标是确保关键循环达到 `II=1`（迭代模调度[12]），并通过依赖解除与循环展开提升单拍工作量，进而提高吞吐、减少整体延迟。

- 顶层数据流：
    - 持续使用 `#pragma HLS DATAFLOW` 组织四阶段（见 v2 顶层）。配合更深流缓冲（见上节），各阶段在“块生成/块数复制/消息调度/压缩迭代”间稳定并发。
- 消息调度（`generateMsgSchedule`）：
    - 两个子循环均保持 `pipeline II=1`，并在 v2 加入 `dependence variable=blk.M inter/intra false`，明确解除对 `blk.M` 的跨迭代与迭代内误判依赖，避免工具插入不必要的访存停顿。
    - 在 `WT16/WT64` 循环中引入轻量 `unroll factor`，将搬移与索引计算的组合逻辑分摊到多个并行路径上，降低单拍负载。
- 压缩迭代（`LOOP_SHA256_UPDATE_64_ROUNDS` 与 `sha256_iter`）：
    - 顶层循环保持 `PIPELINE II=1`；在 v2 中对 `a,b,c,d,e,f,g,h` 工作变量显式添加 `dependence inter false`，避免工具保守地串化更新；每拍均读取 `w_strm` 继续下一轮迭代。
    - `sha256_iter` 内对 `BSIG/MAJ/CH` 先行独立求值，再以分级加法树合并（详见下一节），从结构上降低组合深度以有利于 `II=1` 流水执行。
- HMAC 合并阶段（`mergeKipad/mergeKopad`）：
    - 循环保持 `pipeline II=1`，并在 v2 为字节重排引入 `#pragma HLS unroll factor=8`（按 `dataW` 对齐）以提高每拍的字节输出数量，缩短合并所需的拍数。

总体效果：关键循环稳定达到 `II=1`，消息调度/压缩迭代/合并拼接的单拍工作量提升，宏观数据流并行度增强。

3.2.4 并行化优化

- 通道实现绑定优化：
    - 将关键中间流（`kipadStrm/kopadStrm/kopad2Strm/msgHashStrm/eKipadStrm/eMsgHashStrm`）显式绑定为 `fifo(srl)`，相较 baseline 的 `FIFO_LUTRAM` 降低访问延迟与互连扇出，缓解背压并提升 Fmax。
- 缓冲深度重构：
    - 在密钥路径与消息路径按阶段调优流深度以稳定并行度（如 `kpadHash` 内部流由 4→8；`mergeKipadLenStrm/eMergeKipadLenStrm` 由 4→8；`resHash` 下游合并通道降为 2 并映射 BRAM），匹配各阶段的突发与消费节奏。
- 局部循环展开以提升并发：

    - 在 `mergeKipad/mergeKopad` 的字节拼接循环引入 `#pragma HLS unroll factor=8`，在保持 `II=1` 的前提下提升每拍搬运的元素数量，缩短合并阶段拍数。
- 数据流化并发保留与增强：
    - 顶层保持 `#pragma HLS DATAFLOW` 的三阶段并发（`kpad → msgHash → resHash`），并配合更合适的缓冲深度与实现绑定，使阶段间解耦更充分、稳态吞吐更高。
- 存储类型按热点分配：
    - 热点、窄负载通道采用 `SRL-FIFO` 减少时延；宽负载通道与合并缓冲采用 `BRAM-FIFO` 降低 LUT 压力，形成“SRL 低延迟 + BRAM 高容量”的并行存储组合。
- 常量并行访问：
    - 保持 `K[64]` 和状态向量完全分区，避免端口争用并支持各轮同时读取所需常量。

效果对比（相对 baseline）：通道背压降低、阶段解耦增强，合并/拼接路径的单拍并行度提升，整体 `II=1` 更容易维持；在同等时钟约束下，预期 `Cosim_Latency` 与 `T_exec` 均下降。

3.2.5 缩短关键路径

优化重点在于重写 `sha256_iter` 的表达式结构，使用分级、平衡的加法树，避免“五操作数串行相加”的长组合路径。同时在消息调度阶段将 `σ` 函数与加法拆分至并行路径。

- 迭代级（`sha256_iter`，v2 实现）：
    - 预计算项：`bs1 = BSIG1(e)`、`ch = CH(e,f,g)`、`bs0 = BSIG0(a)`、`maj = MAJ(a,b,c)` 先行计算，随后仅通过加法合并，减少逻辑层数交叉。
    - 分级加法树：
        - 第一级并行两路：`t1_base1 = h + bs1` 与 `t1_base2 = ch + Kt`；`t2_base = bs0 + maj`。
        - 第二级：`t1_mid = t1_base1 + t1_base2`；第三级：`T1 = t1_mid + Wt`，`T2 = t2_base`。
    - 绑定与延迟控制：以 `#pragma HLS bind_op ... impl=fabric latency=0` 约束若干加法器到逻辑阵列并显式设定零额外时延，配合顶层 `PIPELINE II=1`，使逐拍更新 `a,b,c,d,e,f,g,h` 时不形成过长的加法链关键路径；同时工作变量绑定寄存器，确保状态更新路径最短。
- 消息调度级（`generateMsgSchedule`，v2 实现）：
    - 先以独立语句求 `sig0_w1 = SSIG0(w1)` 与 `sig1_w14 = SSIG1(w14)`，再将四项加法拆分为两路并行（`sum1 = sig1_w14 + w9`、`sum2 = sig0_w1 + w0`）与一级合并（`wt = sum1 + sum2`），显式降低加法器级联深度。
    - 采用就地环缓冲 `blk.M[i & 15] = wt`，避免额外的数组搬移与写回冲突，减小组合控制网络规模。

总体效果：在不改变算法语义的前提下，通过表达式重构与并行化合并，显著缩短了 `T1/T2` 形成路径与 `W_t` 生成路径的组合深度，有利于维持更高主频与稳定的 `II=1` 流水。

3.3 LZ4压缩优化策略设计

3.3.1 瓶颈分析（基于 baseline 代码审阅）

- 字典访存并发受限：baseline 在 `lz_compress.hpp` 使用 `#pragma HLS BIND_STORAGE variable = dict type = RAM_T2P impl = BRAM`，未进行 `ARRAY_PARTITION`，且 `dict_flush` 为 `#pragma HLS UNROLL FACTOR = 2`，字典初始化/更新并行度较低。

- 中间流深度偏小：baseline 在 `lz4_compress.hpp::hlsLz4Core` 中间流 `compressdStream/bestMatchStream/boosterStream` 深度均为 `8`，且 `bestMatchStream` 未绑定到 SRL（仅 `compressdStream/boosterStream` 绑定）。
- 文字/偏移流绑定不完整：baseline 在 `lz4_compress.hpp::lz4Compress` 中仅为 `lenOffset_Stream` 绑定 `SRL`，`lit_outStream` 未绑定，访问延时与资源映射不可控。
- Booster 本地存储与流绑定缺失：baseline 在 `lz_optional.hpp::lzBooster`（两种签名）未为 `inStream/outStream` 添加 `BIND_STORAGE`，`local_mem` 也未指定 `BIND_STORAGE` 类型；仅本地缓冲流 `lclBufStream` 有 `STREAM+SRL`。
- 状态机层：baseline 的 `lz4_compress.hpp::details::lz4CompressPart2` 已具备 `#pragma HLS PIPELINE II = 1`、`#pragma HLS DEPENDENCE … inter false`、预读 `nextLenOffsetValue`、预计算 `match_offset+1` 与条件合并（含三元运算），此处并非主要瓶颈。

3.3.2 存储优化

- 统一中间流绑定到 SRL：v2 在 `lz4_compress.hpp::lz4Compress` 为 `lit_outStream/lenOffset_Stream` 增加 `#pragma HLS BIND_STORAGE … impl = SRL`；在 `hlsLz4Core` 为 `compressdStream/bestMatchStream/boosterStream` 全部绑定 `SRL`（baseline 未绑定 `bestMatchStream`）。
- 提升中间流深度：v2 将 `hlsLz4Core` 三路流深度由 `8` 提升到 `32`，减轻模块间反压。
- 字典内存绑定与分区：v2 在 `lz_compress.hpp` 将 `dict` 改为 `#pragma HLS ARRAY_PARTITION variable = dict cyclic factor = 16 dim = 1`，并绑定为 `#pragma HLS BIND_STORAGE variable = dict type = RAM_S2P impl = BRAM`，提升并发与端口利用。
- Booster 本地内存与流绑定：v2 在 `lz_optional.hpp::lzBooster` 为 `inStream/outStream` 增加 `#pragma HLS BIND_STORAGE … impl = srl`，并将 `local_mem` 绑定为 `#pragma HLS BIND_STORAGE variable = local_mem type = RAM_S2P impl = LUTRAM`，降低访问延时与 BRAM 竞争。

3.3.3 流水线优化

- 初始化路径入流水：v2 在 `lz_compress.hpp` 将 `present_window` 初始化循环由 baseline 的 `#pragma HLS PIPELINE off` 改为 `#pragma HLS PIPELINE II = 1`，缓解首段装载的时序压力。
- 保持压缩编码主循环 `II=1`：`lz4_compress.hpp::details::lz4CompressPart2` 维持 `#pragma HLS PIPELINE II = 1`；未在该函数加入新的依赖解除 pragma，流水核心保持稳定。

3.3.4并行化优化

- 字典初始化并行度提升：v2 将 `dict_flush` 的 `UNROLL FACTOR` 提升为 `16`（`lz_compress.hpp` 非流式接口），流式接口版本提升为 `8`，显著提高字典重置吞吐。
- 字典分区并行访问：`ARRAY_PARTITION cyclic factor = 16` 增加字典 bank 并发，配合 `dependence variable = dict inter false`，提高查找/更新重叠能力。
- 模块解耦并行：`hlsLz4Core` 中三路流深度提升至 `32`，并统一 SRL 绑定，使 `lzCompress → lzBestMatchFilter → lzBooster → lz4Compress` 各阶段更易并行推进。

3.3.5 关键路径优化（状态机与算子表达式重构）

- 关键路径侧重在存储访问：v2 的策略是通过将 `lit_outStream/lenOffset_Stream` 以及 `compressd/bestMatch/booster` 等中间流统一绑定为 `SRL` FIFO，缩短 FIFO 访问路径、减小路由压力，而非在状态机表达式层做重构。
- 状态机实现保持规范：`WRITE_OFFSET0` 仍在写第一字节时执行 `match_offset++`（直接按 LZ4 标准递增）；未采用 baseline 的"预读/预计算（`match_offset+1`）与条件合并"写法，状态机关键路径未新增复杂逻辑。
- 总体效果来自"访存更短 + 流深更大"的结构性调整，削弱模块间反压导致的组合关键链；编码状态机本体不作激进改写，保证语义与时序稳定。

3.4 Cholesky分解优化策略设计

3.4.1 瓶颈分析（基于 baseline 代码审阅）

本节基于 `solver/L1/include/hw/cholesky.hpp` 与 `utils/x_matrix_utils.hpp` 的 baseline 实现逐点定位瓶颈，所有结论均与源码结构一一对应。

- 内部存储与索引计算负担（`choleskyAlt`）：
  - 使用一维紧凑三角存储 `L_internal[(n*n - n)/2]`，每次访问都需按列/行生成偏移（`i_off = ((i-1)*(i-1) - (i-1))/2 + (i-1)`、`j_off = ...`）；该"索引生成 → 再访问"的组合链在 `row_loop/col_loop/sum_loop` 内反复出现，形成稳定的组合负载与调度开销。
  - `L_internal` 与 `diag_internal` 未做 `ARRAY_PARTITION`，端口并发能力有限，易成为 `sum_loop(k<j)` 的访存瓶颈。
- 对角路径的计算延迟（`choleskyAlt` + `x_matrix_utils.hpp`）：
  - 对角值以 `cholesky_sqrt_op(A_minus_sum, new_L_diag)` 直接开方；在 `ap_fixed` 路径下，baseline 的 `x_sqrt(ap_fixed)` 调用 `hls::sqrt((double)x)`，引入 double 转换与较长的浮点关键路径；
  - 倒数平方根 `cholesky_rsqrt(ap_fixed)` 同样先走 `x_sqrt` 再做一次除法 `1/sqrt(x)`，叠加双操作延迟与资源；
  - 非对角更新以"除法"实现：`new_L_off_diag = (A[i][j] - Σ L[i,k]*conj(L[j,k])) / hls::x_real(L[j][j])`，除法本身在 HLS 中较昂贵，且对频率不友好。
- 复乘加与表达式结构（全架构通用）：
  - `sum_loop` 的乘加链以串行累加为主，未使用 `BIND_OP` / `EXPRESSION_BALANCE` 指示；在复数路径下，`hls::x_conj(L[j][k])` 与乘法/加法串联，形成加乘长链。
- 循环与流水线调度：
  - `choleskyAlt` 采用 `row_loop(i) → col_loop(j<i) → sum_loop(k<j)` 的三重嵌套，`sum_loop` 虽标注 `PIPELINE II=CholeskyTraits::INNER_II(=1)`，但外层两级的变长循环（`j<i`、`k<j`）与索引生成逻辑共同增加调度复杂度；
  - Traits 默认 `UNROLL_FACTOR=1`，限制了 `sum_loop` 的可并行度，导致单拍工作量较小，需要更多拍数完成同等累加任务。
- 流/接口与测试驱动：
  - 顶层 `cholesky(hls::stream<...>&)` 在读/写阶段做 `#pragma HLS PIPELINE`，本身开销不大；瓶颈主要集中于内核的三重循环与对角/非对角两条数值路径。

综上，baseline 的主要压力来自：一维紧凑存储的索引负担、`ap_fixed` 走 double 的 `sqrt/rsqrt` 路径、非对角除法的高延迟、未并行化的复乘加链，以及外层变长循环的调度压力。

3.4.2存储优化

3.4.2.1 存储组织与索引消除（ARCH1重构为二维 L_internal）

基于 v2 的 `choleskyAlt`，将内部存储由 baseline 的一维紧凑三角数组改为二维 `L_internal[n][n]`，并配合分区与资源约束提升并行带宽、消除索引生成负担：

- 二维存储重构：`OutputType L_internal[RowsColsA][RowsColsA];`，避免 `i_off/j_off` 的复杂索引计算；实际访问直接采用 `L_internal[i][k]`、`L_internal[j][k]`。
- 数组分区与带宽：
  - `#pragma HLS ARRAY_PARTITION variable=L_internal complete dim=CholeskyTraits::UNROLL_DIM` 提升沿展开维度的完全并发；
  - `#pragma HLS ARRAY_PARTITION variable=L_internal cyclic dim=2 factor=CholeskyTraits::UNROLL_FACTOR` 增强列维度的并行访存；
  - `#pragma HLS ARRAY_PARTITION variable=A complete dim=CholeskyTraits::UNROLL_DIM` 与 `#pragma HLS ARRAY_PARTITION variable=L complete dim=CholeskyTraits::UNROLL_DIM`，保障输入/输出矩阵的并发读写；
  - `diag_internal[RowsColsA]` 完全分区并绑定寄存器（`RESOURCE core=Register`），作为对角倒数缓存的低延迟通路。

效果：在 `sum_loop(k<j)` 内减少"索引生成→再访问"的组合链路与潜在端口冲突，稳定支撑更大的 `UNROLL_FACTOR` 与 `II=1`。

3.4.3流水线优化

3.4.3.1 循环结构与 II 稳定（UNROLL + PIPELINE + 轻量 tripcount）

v2 在三重循环结构中通过 `PIPELINE`、`UNROLL` 与适度的 `loop_tripcount` 指示，稳定实现关键内层 `II=1` 与更高并行度：

- 外层循环：`row_loop/col_loop` 使用 `#pragma HLS PIPELINE` 或 `UNROLL` 提升推进速率，减少非计算性停顿；
- 内层 `sum_loop(k<j)`：
  - 明确 `#pragma HLS PIPELINE II=1`，并以 `#pragma HLS UNROLL factor=CholeskyTraits::UNROLL_FACTOR` 提升单拍工作量；
  - 使用 `#pragma HLS EXPRESSION_BALANCE` 平衡表达式，避免过深的加法级联；
- 参考 `ARCH2` 的固定边界手法，保留轻量 `loop_tripcount` 注记用于报告，但不引入阻塞性结构更改；

效果：在不改变算法语义的前提下，`sum_loop` 达到稳定的 `II=1`，外层推进更快，整体延迟降低。

3.4.3.2 Traits 参数调优与架构策略（并行度与精度的统一）

在 `choleskyTraits` 各特化中统一提升 `UNROLL_FACTOR` 与保持 `ARCH=1` 路径，以适配不同数值类型的并行度需求：

- 通用类型（非复）默认：`UNROLL_FACTOR = 2`；

- 复数/复数定点特化：`UNROLL_FACTOR = 8`；
- `INNER_II = 1` 保持内层流水线目标不变；
- `UNROLL_DIM` 沿上下三角选择维度保持一致；

效果：在不改变顶层接口的前提下，提高了内核的可并行度与带宽需求匹配，使 `ARRAY_PARTITION` 的维度策略与展开因子互相配合。

3.4.5关键路径优化

3.4.5.1 对角路径重写（rsqrt近似 + 乘法重构）

v2 将对角路径改写为“倒数平方根近似 + 乘法恢复”，同时针对 `ap_fixed` 提供精炼的 `rsqrt` 实现与小值钳位，降低关键路径延迟并提升数值稳定性：

- 对角公式重写：设 `d_j = A[j,j] - sum_{k<j} |L[j,k]|^2`，计算 `r_j ≈ 1/sqrt(d_j)`，并以 `L[j,j] = d_j * r_j` 恢复 `sqrt(d_j)`，避免直接调用 `sqrt` 的高延迟；v2 实现为：
  - `cholesky_rsqrt(hls::x_real(A_minus_sum_cast_diag), new_L_diag_recip);`
  - `new_L_diag_real = hls::x_real(A_minus_sum_cast_diag) * new_L_diag_recip;`
  - `cholesky_set_diag_from_real(new_L_diag_real, new_L_diag);`
- `ap_fixed` 倒数平方根精炼：在 `utils/x_matrix_utils.hpp` 新增 `x_rsqrt_refined(ap_fixed)`，使用一次牛顿迭代（NR）校正：
  - 初值：`y0 = rsqrtf((float)x)`；
  - 校正：`y1 = y0 * (1.5f - 0.5f * x * y0 * y0)`[11]；
  - 钳位：依据小数位宽选择 `eps ∈ {1e-6,1e-4,1e-3}`，对 `x<=0` 进行下限钳位，避免无效输入。
- `ap_fixed` 基本算子路径重写：
  - `x_sqrt(ap_fixed)` 与 `x_rsqrt(ap_fixed)` 改用 `(float)` 的 HLS 内建实现并 `INLINE`，缩短关键路径；

效果：对角路径由“sqrt+除法”转为“rsqrt+乘法”，并结合 NR 校正与钳位，既降低组合深度，又保证复数域的对角值为实数且数值稳定。

3.4.5.2 复数乘法与DSP绑定、表达式平衡（乘加链路加速）

针对复数乘法与累计加法，v2 以 `BIND_OP` 与就地并行的实现方式降低乘加链关键路径：

- 复数×实数乘法（`cholesky_prod_sum_mult`）：
  - 将实部与虚部分离并行计算，并显式绑定乘法到 DSP：`#pragma HLS BIND_OP variable=rtmp/itmp op=mul impl=DSP`[7]；
  - 全路径 `INLINE`，避免函数调用开销；
- 累加链绑定与平衡：
  - 在 `sum_loop` 与对角累计中，将加法器绑定到 DSP，配合 `EXPRESSION_BALANCE` 平衡多操作数加法，降低级联深度：
    - 例如：`#pragma HLS BIND_OP variable=product_sum op=add impl=DSP`、`#pragma HLS BIND_OP variable=square_sum op=add impl=DSP`；
  - 缓存共轭值 `Ljkc = hls::x_conj(L_internal[j][k])`，减少重复共轭调用，缩短组合链。

效果：复乘加路径的时延与链路级数显著缩短，有利于维持更高主频并稳定 `II=1`。

3.4.6 鲁棒性优化（x_matrix_utils.hpp）

3.4.6.1 数值稳定性与类型适配（复数对角、ap_fixed 小值钳位）

v2 在数值路径上做了针对性适配，避免复数域与定点域带来的不稳定：

- 复数对角处理：以 `cholesky_set_diag_from_real(real_val, dout)` 在复数输出类型中显式置虚部为零，确保对角元素为实数；
- 负值检查：在综合/仿真两态下，对 `hls::x_real(A_minus_sum_cast_diag) < 0` 做错误码返回，维持与 baseline 一致的异常处理契约；
- `ap_fixed` 小值钳位与精炼 `rsqrt`：按小数位宽设置 `eps` 并仅一阶 NR 校正，兼顾时延与精度；

效果：保证复数域对角的物理语义与定点域数值稳定，避免因小值/负值导致的非正定输入失配或时序抖动。

4. LLM 辅助优化记录

4.1 SHA-256算法优化

4.1.1优化阶段一 [优化关键路径，减少延迟]

Prompt设计

```
角色设定：你是资深 HLS 算法优化工程师，专精于密码学算法的硬件加速实现。目标是在保证 SHA256 算法正确性与接口兼容性的前提下，优化关键路径延迟，提升整体性能与时序收敛能力。

一、背景与范围
- 文件与路径：
  - hlstrack2025/security/L1/include/xf_security`
    代码：#sha224_256.hpp #hmac.hpp，test：“#test.cpp”
- 算法语义与输出结果不变；不触碰外部接口与 SHA256/SHA224 标准规范。
- 重点优化 `sha256_iter` 函数中的 T1 计算路径，该路径是整个算法的关键瓶颈。

二、当前问题分析
1) **关键路径过长**：
   - baseline 版本中 `T1 = h + Sigma1(e) + Ch(e, f, g) + K + W` 使用串行加法
   - 5个32位操作数的串行相加导致关键路径延迟过长
   - 影响整体时钟频率与 II 达成
2) **时序收敛困难**：
   - 多级串行加法器增加组合逻辑延迟
   - 在高频率目标下难以满足时序要求
三、实现思路
- 使用CSA(Carry Save Adder)进行加法器的优化
四、输出与交付
- 给出完整的加法器函数实现代码；
- 说明每个函数的设计动机与预期性能影响；
- 提供 `sha256_iter` 函数的完整优化版本；
- 保证算法正确性与接口兼容性的声明。

五、技术约束与注意事项
- 所有优化必须在模板函数内实现，保持代码的通用性；
- HLS pragma 指令必须正确使用，避免综合错误；
```

- 加法器的并行度与资源使用需要平衡，避免过度优化导致资源浪费；
- 必须保持与现有 `generateMsgSchedule`、`sha256Digest` 等函数的兼容性。

LLM回答摘要

模型分析了SHA256算法中T1计算的关键路径问题，提出了使用CSA（Carry-Save Adder）加法器的优化方案
1）引入CSA加法器结构（`csa_add_3`, `csa_add_4`）
2）使用分阶段加法器（`add_f_staged`, `add_3_staged`）
3）添加HLS优化指令（`inline`, `bind_op`）

优化实施

```
//baseline
template <unsigned int w>
ap_uint<32> sha256_iter(ap_uint<32> a, ap_uint<32> b, ap_uint<32> c, ap_uint<32> d,
                        ap_uint<32> e, ap_uint<32> f, ap_uint<32> g, ap_uint<32> h,
                        ap_uint<32> K, ap_uint<32> W) {
    ap_uint<32> T1 = h + Sigma1(e) + Ch(e, f, g) + K + W;
    ap_uint<32> T2 = Sigma0(a) + Maj(a, b, c);
    // ... 其他逻辑
}

//v2
// 引入CSA加法器的优化实现
template <unsigned int w>
ap_uint<32> add_f(ap_uint<32> a, ap_uint<32> b) {
#pragma HLS inline
#pragma HLS bind_op variable = return op = add impl = fabric
    return a + b;
}

template <unsigned int w>
ap_uint<32> csa_add_4(ap_uint<32> a, ap_uint<32> b, ap_uint<32> c, ap_uint<32> d) {
#pragma HLS inline
#pragma HLS bind_op variable = return op = add impl = fabric
    return a + b + c + d;
}

template <unsigned int w>
ap_uint<32> sha256_iter(ap_uint<32> a, ap_uint<32> b, ap_uint<32> c, ap_uint<32> d,
                        ap_uint<32> e, ap_uint<32> f, ap_uint<32> g, ap_uint<32> h,
                        ap_uint<32> K, ap_uint<32> W) {
    // 使用CSA加法器优化T1计算
    ap_uint<32> T1 = csa_add_4<w>(h, Sigma1<w>(e), Ch<w>(e, f, g), add_f<w>(K, W));
    ap_uint<32> T2 = add_f_staged<w>(Sigma0<w>(a), Maj<w>(a, b, c));
    // ... 其他逻辑
}
```

实施效果：

```
Estimated有所提高，从原本的13.846优化至11.92，cosim_Latency从809降低至802，执行时间从1.12w降低至9559.84
```

4.1.2 优化阶段二［HMAC数据流存储优化］

Prompt设计

角色设定：你是 HLS 数据流与存储优化专家，专精于密码学算法中的流处理与存储资源优化。目标是在保证 HMAC 算法功能正确性与数据流时序的前提下，优化存储资源配置，提升数据流处理效率与资源利用率。

一、背景与范围
- 文件与路径：
  - hlstrack2025/security/L1/include/xf_security`
    代码：#sha224_256.hpp #hmac.hpp，test：“#test.cpp”
- 算法语义与 HMAC 标准规范不变；不触碰外部接口与密钥处理逻辑。
- 重点优化 `hmacDataflow` 函数中的流存储配置，该函数是整个 HMAC 数据流处理的核心。

二、当前问题分析
1) **存储实现方式不够优化**：
   - 现有版本中多个 `hls::stream` 使用 `depth=4` 和 `FIFO_LUTRAM` 实现，效率不高
   - 缺乏对不同存储类型特性的充分利用
2) **FIFO 深度配置不够精细**：
   - 统一使用 `depth=4` 可能不是所有流的最优配置
   - 部分流可能不需要如此深的缓冲，此处需要你仔细分析
   - 深度设置影响资源使用与时序特性
3) **存储绑定指令使用不当**：
   - 引入 `bind_storage` 指令
   - 充分利用 SRL（Shift Register LUT）的优势

三、优化方向
1) **优化 FIFO 深度配置**：
   - 内部流的深度调整
2) **切换到 SRL 存储实现**：
   - SRL 实现对小深度 FIFO 更高效，时序特性更好，将现有的LUTRAM改为SRL
3) **统一存储绑定策略**：
   - 对所有内部流（`pad1Strm`, `pad2Strm`, `keyHashStrm`, `msgHashStrm` 等）应用统一的优化策略
   - 确保存储配置的一致性与可维护性
4) **保持数据流时序不变**：
   - 优化存储配置的同时，确保数据流的读写时序不受影响
   - 验证优化后的配置能够满足数据流的同步要求

四、输出与交付
- 给出完整的 `hmacDataflow` 函数优化版本；
- 说明每个存储配置变更的技术依据与预期效果；
- 提供存储类型选择的详细分析；
- 保证数据流时序与功能正确性的声明。

五、技术约束与注意事项
- 所有流的深度变更必须经过数据流分析验证；
- `bind_storage` 指令的语法必须正确，避免综合错误；
- 存储优化不能影响 HMAC 算法的密码学安全性；
- 必须保持与现有 `genPad`、`kpad`、`msgHash` 等函数的兼容性；

LLM回答摘要

模型分析了HMAC数据流处理中的存储优化问题，提出了以下优化方案：

1. **存储实现分析**：
   - FIFO_LUTRAM适合深度较大的FIFO
   - SRL实现更适合深度较小的FIFO（如depth=2）
   - SRL具有更好的时序特性和资源效率
2. **优化策略**：
   - 将FIFO深度从4减少到2
   - 使用SRL实现替代LUTRAM
   - 添加bind_storage指令优化存储绑定

优化实施

```
//baseline版本
```cpp
template <unsigned int hashW, unsigned int msgW>
void hmacDataflow(hls::stream<ap_uint<msgW> >& msgStrm,
                  hls::stream<bool>& endMsgStrm,
                  hls::stream<ap_uint<64> >& msgLenStrm,
                  hls::stream<ap_uint<8> >& keyStrm,
                  hls::stream<ap_uint<64> >& keyLenStrm,
                  hls::stream<ap_uint<hashW> >& hmacStrm,
                  hls::stream<bool>& endHmacStrm) {
    // 使用LUTRAM实现，深度为4
    hls::stream<ap_uint<8> > pad1Strm;
#pragma HLS stream variable = pad1Strm depth = 4
#pragma HLS resource variable = pad1Strm core = FIFO_LUTRAM

    hls::stream<ap_uint<8> > pad2Strm;
#pragma HLS stream variable = pad2Strm depth = 4
#pragma HLS resource variable = pad2Strm core = FIFO_LUTRAM
    // ... 其他stream定义
}

//v2
template <unsigned int hashW, unsigned int msgW>
void hmacDataflow(hls::stream<ap_uint<msgW> >& msgStrm,
                  hls::stream<bool>& endMsgStrm,
                  hls::stream<ap_uint<64> >& msgLenStrm,
                  hls::stream<ap_uint<8> >& keyStrm,
                  hls::stream<ap_uint<64> >& keyLenStrm,
                  hls::stream<ap_uint<hashW> >& hmacStrm,
                  hls::stream<bool>& endHmacStrm) {
    // 使用SRL实现，深度优化为2
    hls::stream<ap_uint<8> > pad1Strm;
#pragma HLS stream variable = pad1Strm depth = 2
#pragma HLS bind_storage variable = pad1Strm type = fifo impl = srl

    hls::stream<ap_uint<8> > pad2Strm;
```
```

```
#pragma HLS stream variable = pad2Strm depth = 2
#pragma HLS bind_storage variable = pad2Strm type = fifo impl = srl
    // ... 其他stream定义
}
```

4.1.3 优化阶段三［性能瓶颈突破，关键路径再次优化（分级加法树）］

Prompt设计

角色设定：你是资深 HLS 算法优化工程师，专精于关键路径优化的算法的工程师。目标是在保证 SHA256 算法正确性与接口兼容性的前提下，优化关键路径延迟，提升整体性能与时序收敛能力。

一、背景与范围
- 文件与路径：
  - hlstrack2025/security/L1/include/xf_security`
    代码：#sha224_256.hpp #hmac.hpp，test：“#test.cpp”
- 算法语义与输出结果不变；不触碰外部接口与 SHA256/SHA224 标准规范。
- 重点优化 `sha256_iter` 函数中的 T1 计算路径，该路径是整个算法的关键瓶颈。

二、当前问题分析
1) **关键路径过长**：
   - CSA策略依旧有一定延迟
2) **时序收敛困难**：
   - 在高频率目标下难以满足时序要求

三、实现思路
- 使用分级加法树策略，将加法链路拆解成多个模块，提高并行度减少延迟

四、输出与交付
- 给出完整的加法器函数实现代码；
- 说明每个函数的设计动机与预期性能影响；
- 提供 `sha256_iter` 函数的完整优化版本；
- 保证算法正确性与接口兼容性的声明。

五、技术约束与注意事项
- 所有优化必须在模板函数内实现，保持代码的通用性；
- HLS pragma 指令必须正确使用，避免综合错误；
- 加法器的并行度与资源使用需要平衡，避免过度优化导致资源浪费；
- 必须保持与现有 `generateMsgSchedule`、`sha256Digest` 等函数的兼容性。

LLM回答摘要

模型分析了sha256_iter函数采用多级加法树的具体实现方案，并给出具体代码示例，具体见下方

优化实施

```
// baseline
uint32_t T1 = h + BSIG1(e) + CH(e, f, g) + Kt + Wt;
uint32_t T2 = BSIG0(a) + MAJ(a, b, c);

// v2
uint32_t bs1      = BSIG1(e);
uint32_t t1_base1 = h + bs1;             #pragma HLS bind_op variable=t1_base1 op=add
impl=fabric latency=0
uint32_t t1_base2 = CH(e, f, g) + Kt;    #pragma HLS bind_op variable=t1_base2 op=add
impl=fabric latency=0
uint32_t t1_mid   = t1_base1 + t1_base2; #pragma HLS bind_op variable=t1_mid   op=add
impl=fabric latency=0
uint32_t T1       = t1_mid + Wt;
uint32_t T2       = BSIG0(a) + MAJ(a, b, c);
```

阶段效果（最终）

- 关键路径显著缩短、主频提升；配合阶段一二的 II 稳定、流深匹配、hmac存储优化，总体执行时间由最初的11201ns下降至6109ns。量化结果详见“第5章对比”。

4.2 LZ4算法优化

4.2.1 优化阶段一：[字典存储与初始化收敛]

Prompt 设计

```
角色设定：你是资深 HLS C++ 工程师。目标是在保证编码语义与外部接口不变的前提下，收敛字典绑定与初始化策略，使主计算更容易达到 II=1，且综合资源更稳。

一、背景与范围
- 文件与路径：
  - v2：`hlstrack2025/data_compression/L1/include/hw/`
  - 代码：#lz_compress.hpp #lz4_compress #lz_optional
- 算法语义与输出格式不变；不触碰 Host 接口与编码规则（literal/match/offset）。

二、优化方向
1)  dict存储绑定调整：
   - 保留或适度降低 `ARRAY_PARTITION`，避免端口冲突与过度扇出。
2)  dict_flush的展开因子调整到合理数值

三、验收标准
- `csim` 通过；压缩结果与 baseline 一致；
- 主计算更容易达到或保持 `II=1`；
- 报表中端口冲突警告显著减少，资源占用更稳（LUT/FF 下降，BRAM 用量合理）。

四、输出与交付
- 给出具体代码改动片段（pragma 改动位置与语句）；
- 说明每项变更的动机（端口、调度、资源）与预期影响；
- 保留算法流程与接口不变的保证声明。
```

LLM 回答摘要

- dict _flush展开系数修改为：`#pragma HLS UNROLL FACTOR=2`
- 初始化阶段不追求极限吞吐，优先稳态与资源
- dict存储绑定从 `RAM_S2P` 调整为 `RAM_T2P`（双端口可同周期读写）
- 将 `present_window` 的初始化流水线关闭： `#pragma HLS PIPELINE II=1` 改为 `#pragma HLS PIPELINE off`
  目的：避免启动阶段干扰后续主环的调度，减少不可预期约束。

优化实施

代码修改：

```
//baseline
uintDictV_t dict[LZ_DICT_SIZE];
#pragma HLS BIND_STORAGE variable = dict type = RAM_T2P impl = BRAM
...
dict_flush:
for (int i = 0; i < LZ_DICT_SIZE; i++) {
#pragma HLS PIPELINE II = 1
#pragma HLS UNROLL FACTOR = 2
    dict[i] = resetValue;
}
...
for (uint8_t i = 1; i < MATCH_LEN; i++) {
#pragma HLS PIPELINE off
    present_window[i] = inStream.read();
}

//v2
uintDictV_t dict[LZ_DICT_SIZE];
#pragma HLS ARRAY_PARTITION variable = dict cyclic factor = 16 dim = 1
#pragma HLS BIND_STORAGE variable = dict type = RAM_S2P impl = BRAM
...
dict_flush:
for (int i = 0; i < LZ_DICT_SIZE; i++) {
#pragma HLS PIPELINE II = 1
#pragma HLS UNROLL FACTOR = 16
    dict[i] = resetValue;
}
...
for (uint8_t i = 1; i < MATCH_LEN; i++) {
#pragma HLS PIPELINE II = 1
    present_window[i] = inStream.read();
}
```

实施效果：

- II改善：[从2降低到1]
- 延迟改善：cosim_Latency大幅度下降，从最初的3390下降至2070，时钟频率保持不变

4.2.2优化阶段二[数据流链路加深与 FIFO 绑定统一]

Prompt 设计

```
角色设定：你是 HLS 数据流链路优化顾问。目标是在不改变 token/literal/match/offset 语义与顺序的前提下，统一并加深中间流的缓冲与绑定，提高稳态吞吐。

一、背景与范围
- 文件与路径：
  - v2：`hlstrack2025/data_compression/L1/include/hw/`
  - 代码：#lz_compress.hpp #lz4_compress #lz_optional
- 算法语义与输出格式不变；不触碰 Host 接口与编码规则（literal/match/offset）。

二、目标与约束
- 目标：将 `compressd/bestMatch/booster` 深度统一到 `32`；输出侧 `lit_outStream/lenOffset_Stream` 统一绑定为 `SRL`；
- 约束：保持编码与封包逻辑不变；Host 接口与 `endOfStream` 时序保持一致。

二、优化方向
1) 流深度调试，尝试加深，具体深度根据你的分析来决定
2) FIFO绑定

三、验收标准
- `csim` 通过；输出格式与大小一致；
- 在大块/交错小块输入下背压降低；
- 报表中 SRL 资源占用可控，稳态吞吐更平滑。

四、交付与说明
- 列出每个流的深度与绑定改动语句；
- 标注变更位置（`hlsLz4Core` 与 `lz4Compress`）；
- 保证不改变编码路径与输出语义。
```

LLM回答摘要

- 在 `lz4Compress` 中：
  为 `lit_outStream` 与 `lenOffset_Stream` 添加 `BIND_STORAGE=SRL`（与 v 版本一致，baseline 未绑定 `lit_outStream`）
- 在 `hlsLz4Core` 中：
  `#pragma HLS STREAM variable=compressdStream depth=32`
  `#pragma HLS STREAM variable=bestMatchStream depth=32`
  `#pragma HLS STREAM variable=boosterStream depth=32`
  三者均添加：`#pragma HLS BIND_STORAGE ... impl=SRL`

优化实施

```
//baseline（`hlstrack2025/.../lz4_compress.hpp`）
hls::stream<ap_uint<32> > compressdStream("compressdStream");
hls::stream<ap_uint<32> > bestMatchStream("bestMatchStream");
hls::stream<ap_uint<32> > boosterStream("boosterStream");
#pragma HLS STREAM variable = compressdStream depth = 8
#pragma HLS STREAM variable = bestMatchStream depth = 8
```

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

#pragma HLS BIND_STORAGE variable = compressdStream type = FIFO impl = SRL
#pragma HLS BIND_STORAGE variable = boosterStream type = FIFO impl = SRL

// lz4Compress(...)
#pragma HLS STREAM variable = lit_outStream depth = MAX_LIT_COUNT
#pragma HLS STREAM variable = lenOffset_Stream depth = c_gmemBurstSize
#pragma HLS BIND_STORAGE variable = lenOffset_Stream type = FIFO impl = SRL

//v2
#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

// lz4Compress(...)
#pragma HLS BIND_STORAGE variable = lit_outStream type = FIFO impl = SRL
#pragma HLS BIND_STORAGE variable = lenOffset_Stream type = FIFO impl = SRL
```

阶段效果（最终）

- BRAM使用率恢复Guideline范围内，资源利用充分且合理
- 延迟大幅度下降，由最初的3390——>2070——>1599
- 时钟频率上升较少，但总体执行时间T_exec由44815.8 ns下降至19923.5 ns！！

4.3 Cholesky算法优化

4.3.1优化阶段一[rsqrt优化]

角色设定：作为资深 HLS 算法优化工程师，您需要在保证 Cholesky 算法正确性与接口兼容性的前提下，针对关键路径进行优化，重点优化函数 `cholesky_rsqrt` ，提升整体性能与时序收敛能力。

具体优化任务要求：

1. 代码定位与分析：
   - 准确定位 `cholesky_rsqrt` 函数的实现位置
   - 详细分析当前实现中的计算瓶颈和关键路径

2. 文献研究与方案制定：
   - 系统搜索关于复数定点数 Cholesky 分解优化的最新学术论文
   - 重点关注矩阵运算优化、并行计算架构和硬件友好算法设计
   - 结合 Arch1 架构特性，制定硬件协同优化方案

3. 优化实施：
   - 优先从 `cholesky.hpp` 入手进行优化,采用“脉动阵列进行对角运算，加快效率”
   - 可考虑修改其他相关头文件，但不得修改 `kernel_cholesky_0.cpp`
   - 保持当前精度不变，避免牺牲数值稳定性
   - 每次修改后必须全面检查语法和逻辑正确性

4. 性能评估：
   - 使用提供的报告文件（csynth.rpt、kernel_cholesky_0_csynth.rpt等）分析优化效果
   - 重点关注 T_exec = EstimatedClockPeriod × TotalExecution(cycles) 的降低
   - 确保时序收敛能力得到改善

5. 约束条件：
   - 必须使用 Arch1 架构
   - 只能修改 solver 文件夹内的代码
   - 运行脚本 run_hls.tcl 中除时钟参数外不得修改
   - 保持接口兼容性，不改变现有函数签名

6. 实施策略：
   - 采用保守渐进式优化方法
   - 每次优化后进行全面验证
   - 记录每次优化的性能提升和修改内容
   - 确保不引入新的时序违例

请按照以下步骤执行优化工作：
1. 分析当前实现的关键路径
2. 研究相关优化文献
3. 制定具体优化方案
4. 实施代码修改
5. 验证优化效果
6. 迭代优化直至达到性能目标

LLM 回答摘要

- 仅修改头文件并严格使用 Arch=1 ，未改动 test 和 Tcl（只运行 vitis_hls run_hls.tcl ）。
- 使用精炼搬 rsqrt:浮点初值 + 一阶NR修正

优化实施

优化前：

```
void cholesky_rsqrt(ap_fixed<W1, I1, Q1, O1, N1> x, ap_fixed<W2, I2, Q2, O2, N2>& res) {
Function_cholesky_rsqrt_fixed:;
    ap_fixed<W2, I2, Q2, O2, N2> one = 1;
    ap_fixed<W1, I1, Q1, O1, N1> sqrt_res;
    ap_fixed<W2, I2, Q2, O2, N2> sqrt_res_cast;
    sqrt_res = x_sqrt(x);
    sqrt_res_cast = sqrt_res;
    res = one / sqrt_res_cast;
```

优化后：

```
template <typename InputType, typename OutputType>
void cholesky_rsqrt(InputType x, OutputType& res) {
Function_cholesky_rsqrt_default:;
    InputType eps = (InputType)1.0e-6;
    InputType x_clamped = (x <= (InputType)0 ? eps : x);
    res = x_rsqrt(x_clamped);
}
```

```
template <int W1, int I1, ap_q_mode Q1, ap_o_mode O1, int N1, int W2, int I2, ap_q_mode Q2,
ap_o_mode O2, int N2>
void cholesky_rsqrt(ap_fixed<W1, I1, Q1, O1, N1> x, ap_fixed<W2, I2, Q2, O2, N2>& res) {
Function_cholesky_rsqrt_fixed:;
    // 使用精炼版 rsqrt：浮点初值 + 一阶NR修正，低延迟且保持精度
    ap_fixed<W2, I2, Q2, O2, N2> x_fix = (ap_fixed<W2, I2, Q2, O2, N2>)x;
    // 参数化 eps：根据定点数精度动态设定，确保数值稳定性
    // 对于小数部分位宽 (W-I)，eps 设为 2^(-(W-I-2)) 以避免下溢
    const int frac_bits = W2 - I2;
    ap_fixed<W2, I2, Q2, O2, N2> eps;
    if (frac_bits >= 10) {
        eps = (ap_fixed<W2, I2, Q2, O2, N2>)1.0e-6;  // 高精度：1e-6
    } else if (frac_bits >= 6) {
        eps = (ap_fixed<W2, I2, Q2, O2, N2>)1.0e-4;  // 中精度：1e-4
    } else {
        eps = (ap_fixed<W2, I2, Q2, O2, N2>)1.0e-3;  // 低精度：1e-3
    }
    if (x_fix <= (ap_fixed<W2, I2, Q2, O2, N2>)0) { x_fix = eps; }
    ap_fixed<W2, I2, Q2, O2, N2> y = x_rsqrt_refined(x_fix);
    res = y;
}

// Helper: assign diagonal value from a real to DIAG_T output type
template <typename T_REAL, typename T_OUT>
void cholesky_set_diag_from_real(T_REAL real_val, T_OUT& dout) {
    dout = (T_OUT)real_val;
}
template <typename T_REAL, typename T_OUT>
void cholesky_set_diag_from_real(T_REAL real_val, hls::x_complex<T_OUT>& dout) {
    dout.real(real_val);
    dout.imag(0);
}
template <typename T_REAL, typename T_OUT>
void cholesky_set_diag_from_real(T_REAL real_val, std::complex<T_OUT>& dout) {
    dout.real(real_val);
    dout.imag(0);
}
```

优化效果：

- Latency大幅度下降，由3050下降至2103

4.3.2优化阶段二[基本参数调节后遇到的瓶颈，向LLM寻求突破口]

Prompt 设计

角色设定：作为资深 HLS 算法优化工程师，您需要在保证 Cholesky 算法正确性与接口兼容性的前提下，针对关键路径进行优化，重点降低函数 `choleskyAlt_false_3_choleskyTraits_x_complex_x_complex_ap_fixed_s` 的延迟，提升整体性能与时序收敛能力。

具体优化任务要求：

1. 代码定位与分析：
   - 准确定位 `choleskyAlt_false_3_choleskyTraits_x_complex_x_complex_ap_fixed_s` 函数的实现位置
   - 详细分析当前实现中的计算瓶颈和关键路径

2. 文献研究与方案制定：
   - 系统搜索关于复数定点数 Cholesky 分解优化的最新学术论文
   - 重点关注矩阵运算优化、并行计算架构和硬件友好算法设计
   - 结合 Arch1 架构特性，制定硬件协同优化方案

3. 优化实施：
   - 优先从 `x_matrix_utils.hpp` 入手进行优化
   - 可考虑修改其他相关头文件，但不得修改 `kernel_cholesky_0.cpp`
   - 保持当前精度不变，避免牺牲数值稳定性
   - 每次修改后必须全面检查语法和逻辑正确性

4. 性能评估：
   - 使用提供的报告文件（csynth.rpt、kernel_cholesky_0_csynth.rpt等）分析优化效果
   - 重点关注 T_exec = EstimatedClockPeriod × TotalExecution(cycles) 的降低
   - 确保时序收敛能力得到改善

5. 约束条件：
   - 必须使用 Arch1 架构
   - 只能修改 solver 文件夹内的代码
   - 运行脚本 run_hls.tcl 中除时钟参数外不得修改
   - 保持接口兼容性，不改变现有函数签名

6. 实施策略：
   - 采用保守渐进式优化方法
   - 每次优化后进行全面验证
   - 记录每次优化的性能提升和修改内容
   - 确保不引入新的时序违例

请按照以下步骤执行优化工作：
1. 分析当前实现的关键路径
2. 研究相关优化文献
3. 制定具体优化方案
4. 实施代码修改
5. 验证优化效果
6. 迭代优化直至达到性能目标

LLM 回答摘要

- 仅修改头文件并严格使用 Arch=1 ，未改动 test 和 Tcl（只运行 vitis_hls run_hls.tcl ）。
- 在 x_matrix_utils.hpp 增加低延迟、精度稳定的倒数平方根 x_rsqrt_refined(ap_fixed) ：
  - 先用 hls::rsqrtf 取得单精度初值；
  - 在 ap_fixed 扩展精度上进行一阶牛顿迭代 y = y*(1.5 - 0.5*x*y*y) ；
  - 计算绑定到 DSP 并保持内联，避免长关键路径与资源瓶颈。
- 在 cholesky.hpp 的 cholesky_rsqrt(ap_fixed) 切换到上述精炼实现，保持 Arch1 的对角路径：
  - 对角值使用 d * rsqrt(d) 避免显式 sqrt ；
  - 离对角值使用对角倒数缓存乘法避免除法。
- 保留你之前认可的并行访问与分区优化（ diag_internal 绑定寄存器、 L_internal 分区、 sum_loop 的 PIPELINE/UNROLL 和 EXPRESSION_BALANCE 、乘加明确绑定到 DSP）。

为什么这样能降 T_exec

- 报告中瓶颈在对角计算路径。将 rsqrt 从“浮点转换+库路径”替换成“浮点初值+单步 NR 在 ap_fixed 域校正”，减少关键运算链延迟，同时保持足够精度；结合“对角乘倒数”替代除法，综合器可将乘法映射为 DSP48 并更好地流水化，从而降低总周期。
- 资源充足条件下，更多 DSP 的使用换来了更短的周期，测试规模 3×3 下总体时序估计也略有改善。

优化实施

```
//x_matrix_utils.hpp新增
// 精炼版倒数平方根（NR一阶修正）：在ap_fixed域使用一次牛顿迭代，低延迟高精度
// 参考改进：以浮点 rsqrt 作为初始值，再进行一次 NR 校正 y = y*(1.5 - 0.5*x*y*y)
// 文献依据：Walczyk et al., "Improving the Accuracy of the Fast Inverse Square Root" (MDPI Entropy, 2021)
//           Numerical Algorithms (Springer, 2024) 对NR/Heron在FPGA场景的讨论与性能取舍。
template <int W, int I, ap_q_mode Q, ap_o_mode O, int N>
ap_fixed<W, I, Q, O, N> x_rsqrt_refined(ap_fixed<W, I, Q, O, N> x) {
#pragma HLS INLINE
    // 初值：使用单精度 rsqrt，避免 double 带来的长关键路径
    float xf = (float)x;
    float y0f = hls::rsqrtf(xf);

    // 迭代在扩展精度中进行以减小舍入误差，再回写到目标类型
    typedef ap_fixed<(W > I ? W + 6 : I + 6), I + 2, AP_RND_CONV, AP_SAT, 0> acc_t;
    acc_t xa = (acc_t)x;
    acc_t y = (acc_t)y0f;
    const acc_t half = (acc_t)0.5;
    const acc_t threehalfs = (acc_t)1.5;

#pragma HLS EXPRESSION_BALANCE
    acc_t y_sq;
    acc_t t1;
    acc_t corr;
#pragma HLS BIND_OP variable=y_sq op=mul impl=DSP
#pragma HLS BIND_OP variable=t1 op=mul impl=DSP
#pragma HLS BIND_OP variable=corr op=mul impl=DSP
    y_sq = y * y;
    t1 = xa * y_sq;          // x*y*y
    corr = half * t1;        // 0.5*x*y*y
    acc_t r = threehalfs - corr;

    acc_t y1;
#pragma HLS BIND_OP variable=y1 op=mul impl=DSP
    y1 = y * r;              // y*(1.5 - 0.5*x*y*y)

    return (ap_fixed<W, I, Q, O, N>)y1;
//cholesky.hpp调用前
template <int W1, int I1, ap_q_mode Q1, ap_o_mode O1, int N1, int W2, int I2, ap_q_mode Q2, ap_o_mode O2, int N2>
void cholesky_rsqrt(ap_fixed<W1, I1, Q1, O1, N1> x, ap_fixed<W2, I2, Q2, O2, N2>& res) {
Function_cholesky_rsqrt_fixed:;
    // 使用精炼版 rsqrt：浮点初值 + 一阶NR修正，低延迟且保持精度
```

```
    double rs = x_rsqrt((double)x);
    res = (ap_fixed<W2, I2,Q2,O2,N2)rs;
}
//cholesky.hpp调用后
template <int W1, int I1, ap_q_mode Q1, ap_o_mode O1, int N1, int W2, int I2, ap_q_mode Q2,
ap_o_mode O2, int N2>
void cholesky_rsqrt(ap_fixed<W1, I1, Q1, O1, N1> x, ap_fixed<W2, I2, Q2, O2, N2>& res) {
Function_cholesky_rsqrt_fixed:;
    // 使用精炼版 rsqrt：浮点初值 + 一阶NR修正，低延迟且保持精度
    ap_fixed<W2, I2, Q2, O2, N2> y = x_rsqrt_refined((ap_fixed<W2, I2, Q2, O2, N2>)x);
    res = y;
}
```

优化效果：

- EstimatedClockPeriod： 5.637 ns → 5.598 ns（小幅提升，时钟估计更紧）
- TotalExecution(cycles)： 2103 → 1687（显著减少）
- T_exec： 11854.611 ns → 9443.826 ns（约 -20.3%）
- 整体解算流程的总延迟下降，验证我们的修改主要压缩了非显示在该实例行之外的流水段（对角/调度合并等），总执行时间随之降低。

优化阶段三[访存效率优化与矩阵运算优化]

Prompt

```
角色设定：作为资深 HLS 算法优化工程师，您需要在保证 Cholesky 算法正确性与接口兼容性的前提下，针对关键路径进行优化，重点降低 `row_loop` 的延迟，提升整体性能

具体优化任务要求：

1. 代码定位与分析：
   - 准确定位 row_loop 函数的实现位置
   - 详细分析当前实现中的计算瓶颈和关键路径

2. 文献研究与方案制定：
   - 系统搜索关于复数定点数 Cholesky 分解优化的最新学术论文
   - 重点关注矩阵运算优化、并行计算架构和硬件友好算法设计
   - 结合 Arch1 架构特性，制定硬件协同优化方案

3. 优化实施：
   - 优先从 `cholesky.hpp` 入手进行优化
   - 可考虑修改其他相关头文件，但不得修改 `kernel_cholesky_0.cpp`
   - 保持当前精度不变，避免牺牲数值稳定性
   - 每次修改后必须全面检查语法和逻辑正确性

4. 性能评估：
   - 使用提供的报告文件（csynth.rpt、kernel_cholesky_0_csynth.rpt等）分析优化效果
   - 重点关注 T_exec = EstimatedClockPeriod × TotalExecution(cycles) 的降低
   - 确保时序收敛能力得到改善

5. 约束条件：
```

- 必须使用 Arch1 架构
- 只能修改 solver 文件夹内的代码
- 运行脚本 run_hls.tcl 中除时钟参数外不得修改
- 保持接口兼容性，不改变现有函数签名

6. 实施策略：
- 采用保守渐进式优化方法
- 每次优化后进行全面验证
- 记录每次优化的性能提升和修改内容
- 确保不引入新的时序违例

请按照以下步骤执行优化工作：
1. 分析当前实现的关键路径
2. 研究相关优化文献
3. 制定具体优化方案
4. 实施代码修改
5. 验证优化效果
6. 迭代优化直至达到性能目标

模型回答摘要

- 数组分割由cyclic改为complete
- row_loop内循环全展开
- 保持pipline II = 1

优化实施

优化前：

```
row_loop:
    for (int i = 0; i < RowsColsA; i++) {
        // Index generation for optimized/packed L_internal memory
        int i_sub1 = i - 1;
        int i_off = ((i_sub1 * i_sub1 - i_sub1) / 2) + i_sub1;

        // Off diagonal calculation
        square_sum = 0;
    col_loop:
        for (int j = 0; j < i; j++) {
#pragma HLS loop_tripcount max = 1 + RowsColsA / 2
            // Index generation
            int j_sub1 = j - 1;
            int j_off = ((j_sub1 * j_sub1 - j_sub1) / 2) + j_sub1;
            // Prime the off-diagonal sum with target elements A value.
            if (LowerTriangularL == true) {
                product_sum = A[i][j];
            } else {
                product_sum = hls::x_conj(A[j][i]);
            }
        sum_loop:
            for (int k = 0; k < j; k++) {
#pragma HLS loop_tripcount max = 1 + RowsColsA / 2
#pragma HLS PIPELINE II = CholeskyTraits::INNER_II
```

```
                prod = -L_internal[i_off + k] * hls::x_conj(L_internal[j_off + k]);
                prod_cast_to_sum = prod;
                product_sum += prod_cast_to_sum;
            }
            prod_cast_to_off_diag = product_sum;
            // Fetch diagonal value
            L_diag_recip = diag_internal[j];
            // Diagonal is stored in its reciprocal form so only need to multiply the
product sum
            cholesky_prod_sum_mult(prod_cast_to_off_diag, L_diag_recip, new_L_off_diag);
            // Round to target format using method specifed by traits defined types.
            new_L = new_L_off_diag;
            // Build sum for use in diagonal calculation for this row.
            square_sum += hls::x_conj(new_L) * new_L;
            // Store result
            L_internal[i_off + j] = new_L;
            if (LowerTriangularL == true) {
                L[i][j] = new_L; // store in lower triangle
                L[j][i] = 0;     // Zero upper
            } else {
                L[j][i] = hls::x_conj(new_L); // store in upper triangle
                L[i][j] = 0;                  // Zero lower
            }
        }

        // Diagonal calculation
        A_cast_to_sum = A[i][i];
        A_minus_sum = A_cast_to_sum - square_sum;
        if (cholesky_sqrt_op(A_minus_sum, new_L_diag)) {
#ifndef __SYNTHESIS__
            printf("ERROR: Trying to find the square root of a negative number\n");
#endif
            return_code = 1;
        }
        // Round to target format using method specifed by traits defined types.
        new_L = new_L_diag;
        // Generate the reciprocal of the diagonal for internal use to aviod the latency of
a divide in every
        // off-diagonal calculation
        A_minus_sum_cast_diag = A_minus_sum;
        cholesky_rsqrt(hls::x_real(A_minus_sum_cast_diag), new_L_diag_recip);
        // Store diagonal value
        diag_internal[i] = new_L_diag_recip;
        if (LowerTriangularL == true) {
            L[i][i] = new_L;
        } else {
            L[i][i] = hls::x_conj(new_L);
        }
    }
    return (return_code);
}
```

优化后：

```
#pragma HLS ARRAY_PARTITION variable=A complete dim=CholeskyTraits::UNROLL_DIM
#pragma HLS ARRAY_PARTITION variable=L complete dim=CholeskyTraits::UNROLL_DIM

row_loop:
    for (int i = 0; i < RowsColsA; i++) {
#pragma HLS UNROLL
        square_sum = 0;
    col_loop:
        for (int j = 0; j < i; j++) {
#pragma HLS loop_tripcount max = 1 + RowsColsA / 2
#pragma HLS UNROLL
            // Prime the off-diagonal sum with target elements A value.
            if (LowerTriangularL == true) {
                product_sum = A[i][j];
            } else {
                product_sum = hls::x_conj(A[j][i]);
            }
        sum_loop:
            for (int k = 0; k < j; k++) {
#pragma HLS loop_tripcount max = RowsColsA
#pragma HLS PIPELINE II=1
#pragma HLS UNROLL factor=CholeskyTraits::UNROLL_FACTOR
#pragma HLS EXPRESSION_BALANCE
#pragma HLS BIND_OP variable=product_sum op=add impl=DSP
                // 预取并复用列元素共轭，减少 x_conj 次数
                OutputType Ljkc = hls::x_conj(L_internal[j][k]);
                product_sum += (typename CholeskyTraits::ACCUM_T)(-L_internal[i][k] * Ljkc);
            }
            // Multiply by diagonal reciprocal (avoid divide)
            L_diag_recip = diag_internal[j];
            cholesky_prod_sum_mult(product_sum, L_diag_recip, new_L_off_diag);
            // Round to target format using method specifed by traits defined types.
            new_L = new_L_off_diag;
            // Accumulate |new_L|^2 for diagonal calculation of row i
            typename CholeskyTraits::ACCUM_T nl_mul;
#pragma HLS BIND_OP variable=nl_mul op=mul impl=DSP
            nl_mul = (typename CholeskyTraits::ACCUM_T)(hls::x_conj(new_L) * new_L);
            // 将对角累计加法绑定到DSP，缩短加法链路
#pragma HLS BIND_OP variable=square_sum op=add impl=DSP
            square_sum += nl_mul;
            // Store result
            L_internal[i][j] = new_L;
            if (LowerTriangularL == true) {
                L[i][j] = new_L; // store in lower triangle
                L[j][i] = 0;     // Zero upper
            } else {
                L[j][i] = hls::x_conj(new_L); // store in upper triangle
                L[i][j] = 0;                  // Zero lower
            }
        }

        // Diagonal calculation for row i (rsqrt path)
        A_cast_to_sum = A[i][i];
```

```
        // 绑定对角差的加法到DSP（sub视作加法实现），缩短关键加法链路
#pragma HLS BIND_OP variable=A_minus_sum op=add impl=DSP
        A_minus_sum = A_cast_to_sum - square_sum;
        A_minus_sum_cast_diag = A_minus_sum;
#ifndef __SYNTHESIS__
        if (hls::x_real(A_minus_sum_cast_diag) < 0) {
            printf("ERROR: Trying to find the square root of a negative number\n");
            return_code = 1;
        }
#else
        if (hls::x_real(A_minus_sum_cast_diag) < 0) {
            return_code = 1;
        }
#endif
        // Generate the reciprocal of the diagonal for internal use using rsqrt(A_minus_sum)
        cholesky_rsqrt(hls::x_real(A_minus_sum_cast_diag), new_L_diag_recip);
        // Compute diagonal sqrt via d * rsqrt(d) to avoid explicit sqrt
        typename CholeskyTraits::RECIP_DIAG_T new_L_diag_real =
            ((typename CholeskyTraits::RECIP_DIAG_T)hls::x_real(A_minus_sum_cast_diag)) *
new_L_diag_recip;
        cholesky_set_diag_from_real(new_L_diag_real, new_L_diag);
        // Round to target format using method specifed by traits defined types.
        new_L = new_L_diag;
        // Store diagonal reciprocal for later off-diagonal use
        diag_internal[i] = new_L_diag_recip;
        if (LowerTriangularL == true) {
            L[i][i] = new_L;
        } else {
            L[i][i] = hls::x_conj(new_L);
        }
    }
    return (return_code);
}
```

优化效果：

- 时钟频率大幅度上升至4.776ns，Latency下降至1335cycle，实现质的飞跃

5. 优化前后性能与资源对比报告

5.1 测试环境

- **硬件平台**: Zynq-7000 (xc7z020-clg484-1)
- **软件版本**: Vitis HLS 2024.2
- **测试数据集**:
  - Cholesky: 3×3复数定点数矩阵
  - LZ4: 标准文本压缩测试数据
  - SHA-256: hmac_sha256的标准测试向量
- **评估指标**:

- Total Execution Time（Estimated Clock Period × Cosim Latency）
- 资源使用（BRAM、DSP、FF、LUT）
- 时序（Slack）
- 功能正确性（Csim、Cosim 验证）

5.2 综合结果对比

5.2.1 SHA-256 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	60	9	↓85.0%	21.4%	3%
DSP	0	0	0%	0%	0%
LUT	39189	16648	↓57.5%	73.6%	31%
FF	12977	15393	↑18.7%	12.2%	14%

5.2.2 SHA-256 性能指标对比

性能指标	优化前	优化后	改善幅度
目标时钟周期(Target_Clock_Period)	15.0 ns	8.60 ns	↓42.7%
估计时钟周期(Estimated_Clock_Period)	13.846 ns	7.656 ns	↓44.7%
时钟频率(Clock_Frequency)	66.67 MHz	130.7 MHz	↑96.0%
延迟(Cosim_Latency_周期)	809	798	↓1.4%
执行时间(T_execution)	11201.4 ns	6109.5 ns	↓45.5%
时序状态（Slack）	-	Pass (+0.08 ns)	-

5.2.3 LZ4 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_18K	106	192	↑81.1%	37.9%	68%
DSP	0	0	0%	0%	0%
LUT	7,235	11,504	↑59.1%	13.6%	21%
FF	3,537	3,760	↑6.3%	3.3%	3%

5.2.4 LZ4 性能指标对比

性能指标	优化前	优化后	改善幅度
目标时钟周期(Target_Clock_Period)	15.0 ns	13.90 ns	↓7.3%
估计时钟周期(Estimated_Clock_Period)	13.220 ns	12.460 ns	↓5.7%
时钟频率(Clock_Frequency)	66.67 MHz	80.3 MHz	↑20.4%
延迟(Cosim_Latency_周期)	3,390	1,599	↓52.9%
执行时间(T_execution)	44,815.8 ns	19,923.5 ns	↓55.5%
时序状态（Slack）	-	Pass (+0.05 ns)	-

5.2.5 Cholesky 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	0	0	0%	0%	0%
DSP	14	45	↑300%	6%	20%
LUT	9223	13415	↑30.2%	17%	25%
FF	4365	7310	↑40.3%	4%	6%

5.2.6 Cholesky 性能指标对比

性能指标	优化前	优化后	改善幅度
目标时钟周期(Target_Clock_Period)	7.0 ns	5.40 ns	↓22.9%
估计时钟周期(Estimated_Clock_Period)	6.276 ns	4.766 ns	↓24.1%
时钟频率(Clock_Frequency)	159.3 MHz	209.9 MHz	↑31.7%
延迟(Cosim_Latency_周期)	4,919	1,335	↓72.9%
执行时间(T_execution)	30,871.6 ns	6,221.1 ns	↓79.9%
时序状态（Slack）	-	Pass (+0.09 ns)	-

5.3 详细分析

5.3.1 资源优化分析

SHA256:

- **BRAM优化效果：** 综合报告显示 BRAM_18K 由优化前的 60 降至优化后的 9（下降 85.0%，利用率由 21.4% 降至 3%）。这与数据流通道更倾向于以 `SRL` 实现（减少 BRAM FIFO）以及内部缓冲/常量使用的实现选择相吻合。BRAM 的显著下降说明阶段解耦主要依赖移位寄存器与寄存器级缓存，避免了大容量双端口 RAM 的过度占用。

- **DSP优化效果：** 两版均为 0。SHA-256 的核心操作以位运算（XOR/AND/ROT/SHR）与加法为主，报告显示加法实现仍为 `fabric`，未强制绑定到 DSP，这也符合 SHA-256 算子的典型实现风格（逻辑阵列即可满足时序）。
- **逻辑资源优化效果：** LUT 从 39,189 降至 16,648（下降 57.5%，利用率由 73.6% 降至 31%），FF 从 12,977 增至 15,393（上升 18.7%，利用率由 12.2% 升至 14%）。这体现为：通过在 `sha256_iter` 中对加法链进行分级与表达式平衡，组合逻辑层数减少，直接压缩了 LUT 规模；同时，数据流化与更稳定的 `II=1` 流水线带来更多状态/通道级寄存（FF）以维持并行推进。

LZ4:

- **BRAM优化效果：** BRAM_18K 从 106 增至 192（上升 81.1%，利用率由 37.9% 升至 68%）。与存储报告一致，`lzCompress` 的字典在 v2 版本中经bank（`ARRAY_PARTITION cyclic`）并绑定为 `RAM_S2P BRAM`，字典按bank展开显著增加 BRAM 实例数量，换取更高带宽与更低端口冲突概率。
- **逻辑资源优化效果：** LUT 从 7,235 增至 11,504（上升 59.1%，利用率 13.6%→21%），FF 从 3,537 增至 3,760（上升 6.3%，利用率 3.3%→3%）。逻辑增长主要来自两处：
  - 匹配/封装路径的状态机与控制逻辑在 v2 中更细化（预取/预计算/依赖解除），相应的选择器与控制寄存器增多；
  - 为稳定 `II=1`，在筛选/增强/封装阶段加入更多寄存与轻量缓冲，提升组合到时序的可收敛性。

Cholesky:

- **逻辑资源优化效果：** DSP 从 14 增至 45（上升 221.4%，利用率 6%→20%），LUT 从 9,223 增至 13,415（上升 30.2%，利用率 17%→25%），FF 从 4,365 增至 7,310（上升 40.3%，利用率 4%→6%）。这与 v2 在复数乘法路径上显式绑定到 DSP 的策略一致：将乘法从 LUT 迁移到 DSP，既缩短关键乘加链，又回收部分 LUT；而更多的流水寄存与中间结果缓存带来 FF 的增长，整体资源分布更利于高频与稳定流水。

5.3.2 性能优化分析

SHA256:

- **流水线效率提升：** 顶层与关键循环维持 `II=1`，并将加法链重写为分级加法树（`t1_base1/t1_base2 → t1_mid → T1`；`t2_base → T2`），降低了单拍内的组合深度。在 `generateMsgSchedule` 中将四项加法拆分为两路并行再合并，进一步稳定了调度。性能报告显示 `Estimated Clock Period` 从 13.846 ns 降至 7.656 ns（下降 44.7%），折算频率提升至 130.7 MHz（↑96.0%）。
- **延迟与执行时间优化效果：** Cosim 延迟从 809 周期降至 798 周期（↓1.4%），执行时间由 11,201.4 ns 降至 6,109.5 ns（↓45.5%）。延迟优化幅度较小，主因在于 SHA-256 的 64 轮迭代存在严格数据依赖；但频率提升显著，使 `T_exec = Period × Latency` 的总体下降达到 45.5%。时序状态为 Pass（Slack +0.08 ns）。

LZ4:

- **延迟优化效果：** Cosim 延迟由 3,390 周期降至 1,599 周期（↓52.9%），说明在 `lzCompress → lzBestMatchFilter → lzBooster → lz4Compress` 的四阶段数据流中，阶段解耦与关键路径重构有效减少了端到端拍数。
- **流水线效率提升与吞吐率提升分析：** 频率从 66.67 MHz 提升到 80.3 MHz（↑20.4%），配合延迟的大幅降低，执行时间由 44,815.8 ns 下降至 19,923.5 ns（↓55.5%）。具体贡献来源包括：dct bank提升并行带宽、SRL 流深度整定缓解背压、Part2 令牌/长度扩展路径的预取与条件重构减少分支级联。时序状态为 Pass（Slack +0.05 ns）。

Cholesky:

- **流水线效率提升：** `sum_loop(k<j)` 保持 `II=1` 并以 `UNROLL factor` 提升单拍工作量，二维 `L_internal` 与 `diag_internal` 的分区/寄存器化减少访存冲突与索引生成负担，使外层推进更加稳定。Estimated 从 6.276 ns 降至 4.766 ns（↓24.1%），频率提高至 209.9 MHz（↑31.7%）。
- **延迟优化效果：** Cosim 延迟由 4,919 周期降至 1,335 周期（↓72.9%），执行时间由 30,871.6 ns 下降至 6,221.1 ns（↓79.9%），表明在复数乘加路径绑定 DSP、对角 `rsqrt+mul` 重写与寄存器化缓存的组合下，端到端拍数与关键路径时延均明显降低。时序状态为 Pass（Slack +0.09 ns）。
- **吞吐率提升分析：** 在频率提升与拍数显著下降的双重作用下，单位时间可完成的矩阵分解数量大幅增加；乘法移至 DSP 也减少了 LUT 级的拥塞与组合链深度，为高频运行提供了结构性支撑，整体吞吐提升与资源分布达成均衡。

5.4 正确性验证

5.4.1 SHA-256算法验证

C代码仿真结果

仿真配置：

- 测试用例：标准的HMAC-SHA256测试向量
- 测试数据类型：64位消息块
- 精度要求：哈希输出与标准SHA-256结果完全一致

仿真结果：

- 功能正确性：✅ 通过
- 输出精度：与标准SHA-256测试向量完全匹配
- 性能验证：满足预期性能指标

联合仿真结果

仿真配置（优化后）：

- RTL仿真类型：Verilog
- 目标时钟周期：8.6 ns
- 估计时钟周期：7.656 ns

仿真结果（优化后）：

- 时序正确性：✅ 通过
- 接口兼容性：✅ 通过
- 延迟(Latency)：798 周期
- 总执行时间：6,109 .5ns（按估计时钟）
- Slack：+0.08 ns（正向slack，满足时序约束）
- 性能匹配度：100%

5.4.2 LZ4压缩算法验证

C代码仿真结果

仿真配置：

- 测试用例：标准LZ4压缩测试数据集
- 测试数据类型：文本和二进制混合数据
- 精度要求：压缩/解压缩数据完全一致

仿真结果：

- 功能正确性：✅ 通过
- 输出精度：压缩数据符合LZ4标准格式，解压后与原始数据完全一致
- 性能验证：满足预期性能指标

联合仿真结果

仿真配置：

- RTL仿真类型：Verilog
- 目标时钟周期：13.9 ns
- 估计时钟周期：12.460 ns (优化后)

仿真结果（优化后）：

- 时序正确性：✅ 通过
- 接口兼容性：✅ 通过（AP_FIFO接口工作正常）
- 延迟(Latency)：1599 周期
- Slack：+0.05 ns
- 性能匹配度：100%

5.4.3 Cholesky分解算法验证

C代码仿真结果

仿真配置：

- 测试用例数量：标准3×3复数定点数矩阵
- 测试数据类型：`hls::x_complex<ap_fixed<16, 1, AP_RND_CONV>>`
- 精度要求：输出矩阵元素精度满足定点数精度要求

仿真结果：

- 功能正确性：✅ 通过
- 输出精度：与MATLAB/NumPy参考实现在定点数精度范围内完全一致
- 数值稳定性：✅ 通过（对称正定矩阵分解稳定）

联合仿真结果

仿真配置（优化后）：

- RTL仿真类型：Verilog
- 目标时钟周期：5.4 ns
- 估计时钟周期：4.766 ns

仿真结果（优化后）：

- 时序正确性：✅ 通过
- 接口兼容性：✅ 通过（AXI-Stream接口）
- 延迟(Latency)：1863周期
- 总执行时间：6221.1ns
- Slack：+0.09 ns
- 性能匹配度：100%

6. 创新点总结

6.1 SHA-256 优化创新点

6.1.1 技术创新点

- 数据流拓扑重构与阶段解耦（代码对应 `sha224_256.hpp::internal::sha256_top`）
  - 顶层以 `#pragma HLS DATAFLOW` 将四阶段串接：`preProcessing → dup_strm → generateMsgSchedule → sha256Digest`。为每路流明确深度与实现类型：`blk_strm depth=64 (lutram)`、`nblk_strm depth=8 (lutram)`、`w_strm depth=128 (lutram)`，稳定支持块级并发与跨阶段解耦。
  - `dup_strm` 将块计数与结束标志一分为二，为消息调度与摘要计算分别供给一致的区块驱动，杜绝“单源多用”引发的背压耦合。
- 消息调度关键路径重写（代码对应 `generateMsgSchedule`）
  - 将 `SSIG0/SSIG1` 的计算与四项加法拆分成两路并行后一级合并：`sig1_w14 + w9` 与 `sig0_w1 + w0`，最后求 `wt = sum1 + sum2`，并将 `blk.M[i & 15] = wt` 写回环缓冲。三处加法均以 `bind_op impl=fabric latency=0` 明确零额外延迟，降低组合级联深度。
  - 显式解除对 `blk.M` 的跨迭代与迭代内误判依赖（`dependence inter/intra false`），确保 `WT64` 循环在 `PIPELINE II=1` 下稳定推进，同时保留适度 `unroll` 提升搬移与位段选择的并发度。
- 64轮迭代表达式平衡与三级加法树（代码对应 `sha256_iter`）
  - 预计算四项：`bs1=BSIG1(e)`、`ch=CH(e,f,g)`、`bs0=BSIG0(a)`、`maj=MAJ(a,b,c)`；随后按三级树结构合并：
    - 第一级：`t1_base1 = h + bs1`、`t1_base2 = ch + Kt`、`t2_base = bs0 + maj`；
    - 第二级：`t1_mid = t1_base1 + t1_base2`；
    - 第三级：`T1 = t1_mid + Wt`，`T2 = t2_base`。

- 关键加法器绑定 Fabric 并设定 `latency=0`，工作变量 `a,b,c,d,e,f,g,h` 绑定寄存器，配合 `LOOP_SHA256_UPDATE_64_ROUNDS` 的 `PIPELINE II=1` 与 `dependence inter false`，在严格数据依赖约束下实现稳态逐拍迭代。

- 常量与状态结构优化（`sha256Digest`/常量表）
  - 常量 `K[64]` 与内部状态 `H[8]` 完全分区（`array_partition complete`），保证多端口并发读与状态累加不受端口瓶颈影响；摘要输出阶段将内部大端格式转换为小端字节序写入，避免外部处理负担。
- 预处理与分块的拍级并行（`preProcessing` 两个重载）
  - 整块生成循环 `II=16` 且字内 `unroll`，尾块场景按 left 长度分别走"一块补齐"或"两块补齐"路径；32b/64b 输入各自实现了大端重排与末尾长度写入，减少外部预处理开销并保持块边界的流水线连贯性。
- HMAC 侧的数据流化拼接与存储整定（`hmac.hpp::internal::hmacDataflow`）
  - `kpad/msgHash/resHash` 三阶段以 `dataflow` 串接，`kipad/kopad/msgHashStrm`、结束标志流与中间拼接流统一绑定 `fifo(srl)`；大数据拼接的 `mergeKipadStrm/mergeKopadStrm` 采用 `FIFO_BRAM`（分别为 `depth=64` 与 `depth=2`），在更高带宽与更低驻留之间取得均衡。
  - `genPad` 逐字节 XOR 生成 `ipad=0x36/opad=0x5c`；`mergeKipad/mergeKopad` 以 `pipeline II=1` 和 `unroll factor=8` 输出拼接后的字节流与长度流，压缩了拼接阶段拍数且不破坏后续哈希的 II。
- 结果与度量（对应 5.2 节）
  - 估计时钟周期由 `13.846 ns` 降至 `7.656 ns`（↓44.7%），频率提升至 `130.7 MHz`（↑96.0%）；Cosim 延迟由 `809` 降至 `798`（↓1.4%）；执行时间由 `11201.4 ns` 降至 `6109.5 ns`（↓45.5%）。改进主要来源于消息调度与迭代级的表达式重构、存储与流深度整定，以及编译器依赖解除后的稳定 `II=1`。

6.1.2 LLM辅助方法创新

- 瓶颈建模与分层提示工程[13]
  - 基于综合/仿真报告的结构化抽取，首先锁定两条关键路径："消息调度四项相加链"与"64轮迭代五操作数串加链"。据此构造分层提示：先询问可重写的表达式形态与编译器指令，再约束实现风格（Fabric 加法器、寄存器绑定、`II=1`）。
  - 针对数据流死锁与背压，提示集中在 FIFO 深度与实现类型选择（`SRL/LUTRAM/BRAM` 的匹配矩阵），以及跨阶段产消速率的粗略估算。LLM 给出备选组合后，经实测保留"`blk_strm=64/nblk=8/w_strm=128` + `mergeKipadStrm BRAM` + 其余 SRL"的折中方案。
- 代码定位与就地重构的协同
  - 明确函数与循环锚点（如 `LOOP_SHA256_UPDATE_64_ROUNDS`、`LOOP_SHA256_PREPARE_WT64`），避免泛化建议；将"三级加法树"与"并行两路再一级合并"的模板化思路映射到具体变量名与语句序，减少实现偏差。
  - 提供保守的"CSA 加法器工具函数"（`csa_add_3/csa_add_4`）作为后续更激进优化的储备组件，但在当前版本不强制替换主路径，降低功能与时序风险。
- 迭代验证与收敛策略
  - 每次代码变更均以 `csynth.rpt` 和 `cosim.rpt` 快速回归：校核 `Estimated`、`Slack` 与 `Latency` 的三元组是否一致改善；若 `Latency` 未降或 `Slack` 回落，则优先微调 `bind_op`/`bind_storage` 与 `STREAM depth`，而不立即引入更复杂算术结构。
  - 记录"可解释性与复现性"：所有改动限定在头文件，且保持外部接口不变；报告中以具体函数/循环名和 pragma 展示对应关系，确保评审可依据源码复查。

6.2 LZ4优化创新点

6.2.1 技术创新点

- 顶层数据流到多核并行架构（代码对应 `lz4_compress.hpp::hlsLz4Core/hlsLz4/lz4CompressMM`）
  - 以 `#pragma HLS dataflow` 串接 `lzCompress → lzBestMatchFilter → lzBooster → lz4Compress`；在 `hlsLz4` 中对 `inStream/outStream/outStreamEos` 显式绑定为 `FIFO(SRL)`，并为 `compressedSize` 显式绑定 `FIFO(SRL)`（baseline 未显式绑定）。
  - 在 `hlsLz4Core` 中将 `compressdStream/bestMatchStream/boosterStream` 深度由 8 提升至 32，新增对 `bestMatchStream` 的 `FIFO(SRL)` 绑定；baseline 为 8 深度，且仅绑定 `compressdStream/boosterStream`。
- Dictionary Banking (ARRAY_PARTITION) & Dual-port BRAM（代码对应 `lz_compress.hpp::lzCompress`）
  - 将字典 `dict[LZ_DICT_SIZE]` 做 `ARRAY_PARTITION cyclic factor=16`，并绑定为 `RAM_S2P BRAM`（baseline 为 `RAM_T2P BRAM`）；以 16 bank分布降低读写端口竞争。`dict_flush` 循环 `UNROLL FACTOR` 由 baseline 的 2 提升到 16。
  - 环形滑窗 `present_window[MATCH_LEN]` 完全分区；起始填充 `PIPELINE II=1`，主循环 `lz_compress` 明确 `dependence variable=dict inter false`，在“读-改-写”顺序下保持 II=1 推进；哈希函数按 `MIN_MATCH`（=3/4）分支实现移位异或，维持简单可综合的组合负载。
- 筛选与增强通路的拍级并行（代码对应 `lz_optional.hpp::lzBestMatchFilter/lzBooster`）
  - `lzBooster`：在 v2 中为 `inStream/outStream` 显式绑定 `FIFO(SRL)`，并将 `local_mem` 绑定为 `RAM_S2P(LUTRAM)`；baseline 未显式绑定这些资源。保持 `PIPELINE II=1`，以 LUTRAM S2P 与 SRL FIFO 提升端口并发与时序余度。
  - Part2：状态机划分为 `WRITE_TOKEN/WRITE_LIT_LEN/WRITE_MATCH_LEN/WRITE_LITERAL/WRITE_OFFSET0/WRITE_OFFSET1`；在进入 `WRITE_TOKEN` 前统一预读 `nextLenOffsetValue` 并以局部变量提取位段，减少“位段选择+流读取”的组合负载；在写偏移前预计算 `match_offset_plus_one=match_offset+1`，避免跨状态重复加法；移除 baseline 中的 `HLS DEPENDENCE` 注记以消除调度限制，保持 II=1。
  - 写入控制以 `compressedSize < input_size` 作为限幅，避免输出超过输入尺寸；令牌高 4 位/低 4 位分别编码 `min(lit_len,15)` 与 `min(match_len,15)`，并以 255 为步长扩展超过 15 的段长度，严格符合 LZ4 规范。
- 流与存储整定（`STREAM/BIND_STORAGE`）
  - `lit_outStream/lenOffset_Stream` 均绑定为 SRL 并设置与突发宽度匹配的深度，避免过度驻留；多块路径中的 `compressedSize` 显式绑定 SRL，使“尺寸回写”在高频下可控。
- 指标与效果（对应 5.2 节）
  - 本节删除原有数值，当前仅记录代码层差异。待 `csynth/cosim/impl` 数据补充后再行更新。

6.2.2 LLM辅助方法创新

- 阶段速率与背压诊断的提示工程[13]
  - 以 `hlsLz4Core` 的 `compressdStream/bestMatchStream/boosterStream` 为锚点，将深度由 8 提至 32，并统一 `BIND_STORAGE type=FIFO impl=SRL`；仅保留与代码一致的修改，删除无代码引用的建议。
- Banking 与 Dependence Removal 的低风险改造路径

- 针对 `lz_compress.hpp::dict` 采用 `ARRAY_PARTITION cyclic factor=16 + RAM_S2P BRAM` 与 `dependence variable=dict inter false`；在 `lz_optional.hpp::lzBooster::local_mem` 使用 `RAM_S2P(LUTRAM)`，与现行代码一致。

- 状态机微结构重写的可验证清单
  - 在 `lz4_compress.hpp::lz4CompressPart2` 中前移 `nextLenOffsetValue` 预读与 `match_offset_plus_one` 预计算，并移除 baseline 的 `HLS DEPENDENCE` 注记以消除调度限制，保持 `II=1`；删除与代码不一致的"777/777"与 `compressdSizeStream` 相关表述。
- 渐进式验证与指标闭环
  - 以 `csynth/cosim` 回归核验 `II/Slack` 与数据一致性，围绕上述代码锚点（`STREAM depth/impl`、`DEPENDENCE`）迭代；暂不使用性能数值作为依据，待数据补充后再更新。

6.3 Cholesky优化创新点

6.3.1 技术创新点

- 架构选择与数据布局重构（代码对应 `cholesky.hpp::choleskyAlt/choleskyAlt2`）
  - 在保持 ARCH=1 的前提下引入 ARCH2 的有效微结构：将内部存储由一维紧凑三角数组改为二维 `L_internal[n][n]`，并按 Traits 进行维度分区（`ARRAY_PARTITION complete/cyclic`），显著降低索引生成负担与端口冲突；`diag_internal[n]` 完全分区且绑定寄存器（`RESOURCE core=Register`），作为对角倒数缓存的低延迟通路。
  - 在 ARCH2 中将变长数据通路改成固定边界嵌套：`col_loop(j) → sum_loop(k≤j) → row_loop(i)`，并明确 `LOOP_FLATTEN off`，避免两个嵌套合并为变长循环而破坏调度；零化上/下三角的后处理循环独立出来，保证主计算稳定 `II=1`。
- 对角路径"rsqrt+乘法"替代"sqrt+除法"（代码对应 `choleskyAlt/choleskyAlt2` 与 `x_matrix_utils.hpp`）
  - 设对角差 `d = A[i][i] - Σ|L[i,k]|^2`，先生成倒数平方根 `r ≈ rsqrt(d)`，再以 `L[i,i] = d * r` 恢复 `sqrt(d)`；避免了 `sqrt` 与后续除法在关键路径上的长延迟，显著缩短组合时延。
  - 在定点路径下使用精炼版倒数平方根 `x_rsqrt_refined(ap_fixed)`：以浮点 `rsqrtf` 为初值，再进行一次 NR 校正 `y ← y*(1.5 - 0.5*x*y^2)`（`BIND_OP mul/add impl=DSP`），并按照小数位宽动态选择 `eps ∈ {1e-6,1e-4,1e-3}` 对非正输入钳位，确保数值稳定与快速收敛。
  - 复数对角仅取实部（`cholesky_sqrt_op`），虚部显式置零（`cholesky_set_diag_from_real`），保持物理语义与实现一致性。
- 复乘加链路加速与表达式平衡（代码对应 `cholesky_prod_sum_mult/sum_loop`）
  - 将复数×实数乘法拆为实/虚两路并行计算，并显式绑定乘法到 DSP：`#pragma HLS BIND_OP variable=rtmp/itmp op=mul impl=DSP`；在 `sum_loop` 与对角累计中，将加法绑定 DSP 并启用 `EXPRESSION_BALANCE`，减少多操作数串联的级数。
  - 对列元素共轭值 `Ljkc = hls::x_conj(L_internal[j][k])` 进行就地缓存，避免重复调用共轭，缩短组合链和访存链；在 ARCH2 中通过 `product_sum_array/square_sum_array` 分离行和与平方和的构造，进一步稳定内层 `II=1`。
- 循环与并行度调优（Traits 与 pragma 一致收敛）
  - 明确 `sum_loop(k<j)` 的 `PIPELINE II=1`，并以 `UNROLL FACTOR=CholeskyTraits::UNROLL_FACTOR` 提升单拍工作量；复数与复数定点特化将 UNROLL_FACTOR 设为 8，使得实/虚乘法与累加在 DSP 上得到充分并行。

- 统一 `UNROLL_DIM` 随 `LowerTriangularL` 选择行或列维度，保证上/下三角输出路径的一致并行策略；对 `A/L/L_internal` 各维度配置 `ARRAY_PARTITION`，增强并发读写端口与总带宽。

- 存储与访存通路微优化（代码对应 `choleskyAlt/choleskyAlt2`）
  - 将对角倒数缓存 `diag_internal` 绑定寄存器，消除对角读路径的存储访问时延与端口共享；在 ARCH2 中对 `L_internal/square_sum_array/product_sum_array` 进行循环维度展开，使"按列构造→按行更新"的访存模式在硬件上呈现稳定并发端口。
  - 在顶层 `cholesky` 的"流⇄数组"转换阶段使用 `#pragma HLS PIPELINE`，保持输入/输出搬移与内核执行的拍级并行。

- 关键路径与时序优化的因果链（与 5.2 指标一致）
  - 将"对角 `sqrt+除法`"替换为" `rsqrt+乘法`"并绑定 DSP，使关键路径的乘加链与除法延迟显著降低，直接贡献于 Estimated 周期由 `6.276 ns` 降至 `4.766 ns`（↓24.1%）与频率 `159.3 MHz → 209.9 MHz`（↑31.7%）。
  - 在内层 `sum_loop` 与行和构造中实施 DSP 加法与表达式平衡，结合二维存储与分区、固定边界循环，将端到端 Cosim 延迟由 `4,919` 降至 `1,335`（↓72.9%）；二者乘积驱动执行时间由 `30,871.6 ns` 降至 `6,221.1 ns`（↓79.9%）。

6.3.2 LLM辅助方法创新

- 面向瓶颈的结构化诊断与提示工程[13]
  - 基于源码和综合报告对五类瓶颈进行定位：一维紧凑三角索引生成负担、对角 `sqrt+除法` 的高延迟、复乘加长链、变长循环调度压力与访存端口冲突；据此将提示分解为"数据布局/算子重写/并行与调度/存储与依赖"四大槽位，降低一次性改动风险。

- 渐进式重构方案与可验证锚点
  - 首先建议将 ARCH1 引入 ARCH2 的二维存储与固定边界循环，并以函数与循环标签为锚点（`row_loop/col_loop/sum_loop`、`zero_rows_loop/zero_cols_loop`），保证改动范围与调度器行为可控；随后替换对角路径为 `rsqrt+mul`，限定到 `cholesky_rsqrt/x_rsqrt_refined` 与 `cholesky_set_diag_from_real` 的具体实现。
  - 对复乘加链的优化，以 `cholesky_prod_sum_mult` 的 DSP 绑定和缓存共轭为切入点，并在 `sum_loop`/对角累计中加入 `BIND_OP add impl=DSP` 与 `EXPRESSION_BALANCE`，同时保持 `INNER_II=1` 与 UNROLL_FACTOR 不变，减少引入不确定性的依赖。

- Traits 策略与参数收敛
  - 在类型特化中统一提升 UNROLL_FACTOR（复数/复数定点设为 8），并保持 ARCH=1 与 INNER_II=1 不变，确保并行度与流水线目标的匹配；通过小步调整分区维度与 UNROLL 维度，基于 `csynth.rpt` 的端口冲突与资源映射反馈迭代收敛。

- 指标闭环与风险控制
  - 每次改动以 `Estimated/Slack/Latency/T_exec` 四元组闭环评估：若 `Estimated` 回落或 `Slack` 下降，则优先在 `BIND_OP/ARRAY_PARTITION/UNROLL` 上微调；若 `Latency` 不降，则检查固定边界循环与独立零化循环是否稳定 `II=1`；在定点域以 `eps` 与一次 NR 校正保证数值稳定。
  - 全过程保持外部接口不变（顶层 `cholesky` 流接口与返回语义一致），所有改动集中在 `cholesky.hpp` 与 `utils/x_matrix_utils.hpp`，报告中提供函数/循环/pragma 的一一对应关系，确保评审可复核与实验可复现。

7. 遇到的问题与解决方案

7.1 SHA-256 优化过程中的问题

7.1.1 技术难点

问题描述	解决方案	效果
长加法链导致关键路径过深	重写 `sha256_iter` 为三级加法树，`bind_op add impl=fabric latency=0`，先行计算 `BSIG/MAJ/CH`	Estimated 周期显著降低，Fmax 提升，`T_exec` 下降 45.5%
数据流背压/死锁风险（FIFO 过浅）	提升 `blk_strm=64`、`w_strm=128` 深度，统一 SRL 绑定；用 `dup_strm` 解耦块计数分发	数据流稳定推进，关键阶段保持 `II=1`，消除阻塞
依赖误判导致迭代 II>1	为 `a,b,c,d,e,f,g,h` 与 `blk.M` 显式 `dependence inter/intra false`，工作变量绑定寄存器	64 轮迭代稳定达成 `II=1`，频率提升、时序收敛

7.2 LZ4 优化过程中的问题

7.2.1 技术难点

问题描述	解决方案	效果
字典读写端口冲突、初始化拍数过高	`dict` bank（`ARRAY_PARTITION cyclic factor=16` + `RAM_S2P BRAM`），`dict_flush` 提升 `UNROLL=16` 且 `PIPELINE II=1`	端口冲突显著减少，启动时间降低，端到端 Latency 下降 52.9%
令牌/长度扩展路径分支级联过深	Part2 预读 `lenOffset`、预计算 `match_offset+1`，对 `match_offset/match_length/lit_length` 解除依赖，状态机保持 `II=1`	Estimated 周期降低、Fmax 提升，`T_exec` 下降 55.5%

7.3 Cholesky 优化过程中的问题

7.3.1 技术难点

问题描述	解决方案	效果
紧凑三角索引生成负担、访存端口竞争	改为二维 `L_internal[n][n]`，`ARRAY_PARTITION complete/cyclic` 提升并发端口	消除索引生成链，降低访存冲突，`II=1` 更稳
对角 `sqrt+除法` 关键路径过长（定点走 double）	以 `rsqrt+乘法` 替换，调用 `x_rsqrt_refined(ap_fixed)` 一次 NR 校正并动态 `eps` 钳位	关键路径显著缩短，Estimated 周期下降 24.1%、Fmax 提升 31.7%

问题描述	解决方案	效果
复乘加链路深、共轭调用频繁	`cholesky_prod_sum_mult` 并行实/虚乘法并绑定 DSP，缓存列共轭；对角累计/行和加法绑定 DSP + 表达式平衡	乘加链层数减少，频率提升，端到端 `Latency` 大幅降低
变长循环调度不稳，零化与主计算耦合	固定边界嵌套（ARCH2），`LOOP_FLATTEN off`，零化循环独立后移	内层 `II=1` 持续，拍数降低，`T_exec` 下降 79.9%

7.4 LLM 辅助过程中的问题

问题描述	解决方案	经验教训
建议过度激进（如状态机合并）与现有架构不匹配	人工审查后采用保守路径（保留状态枚举、解除依赖、先做预读/预计算）	LLM 需在明确约束下工作，避免一次性大改引入风险
综合/仿真报告解析不完备	人工提取 `Estimated/Latency/Slack/资源` 关键字段供 LLM 使用	将 LLM 用于决策辅助而非替代工程核验，信息抽取要结构化
pragma 初值不适配目标设计	迭代调整 `STREAM depth/impl`、`UNROLL_FACTOR`、`BIND_OP`，每次变更做 csynth/cosim 回归	LLM 提供初始搜索方向，最终参数需实验收敛
架构差异导致建议"移植性"差	以函数/循环/pragma 锚点约束（源码对照清单），限定改动范围与层次	将建议映射到具体代码位置，确保可验证与可回退

8. 结论与展望

8.1 项目总结

本项目围绕三类 L1 算子（SHA-256、LZ4、Cholesky）在 Zynq-7000（xc7z020）平台的 HLS 映射进行了系统化的微体系结构优化，目标一致地指向降低执行时间（`T_exec = Estimated × Latency`）并保持时序与资源收敛。总体策略遵循"数据流化、流水线化、表达式与算子重写、存储层次整定、依赖解除与并行度调优"五条主线：

- 数据流化与阶段解耦：顶层以 `#pragma HLS DATAFLOW` 将子模块串接，针对关键流统一设置深度与实现类型（SRL/LUTRAM/BRAM），在产消速率上匹配以降低背压与死锁风险。
- 关键循环的 `II=1` 稳定化：通过显式 `dependence inter/intra false`、工作变量寄存器化与适度 `UNROLL`，确保逐拍推进不被工具的保守依赖判定阻断。
- 表达式与算子重写：
  - SHA-256 将"五操作数串加链"改写为三级加法树，消息调度四项相加拆分为两路并行再合并；
  - LZ4 的状态机预读与偏移预计算，令牌与长度扩展路径减少分支级联；
  - Cholesky 将对角 `sqrt+除法` 重写为 `rsqrt+乘法`，复乘加链路并行并绑定 DSP，表达式平衡以缩短关键路径。
- 存储层次与访存整定：

    - LZ4 dict banking（S2P BRAM）、窗口与中间流 SRL 绑定、BRAM 缓冲用于大数据拼接；
    - Cholesky 二维 `L_internal` 与分区、`diag_internal` 寄存器化，顶层流⇄数组搬移管线化；
    - SHA-256 的块与调度流深度提升与 LUTRAM/SRL 绑定，常量与状态数组完全分区。
- 正确性与可复现：保持外部接口与测试契约不变（流原型与返回语义），所有改动集中在头文件，报告中以函数/循环/pragma 锚点呈现实现对应关系，便于评审溯源与第三方复现。

在上述方法的协同下，三类算子均实现了频率提升（Estimated 周期降低）与端到端拍数下降（Latency 降低），从而达成 `T_exec` 的大幅下降；同时资源保持在器件容量内，时序 Slack 保持正值。优化路径具有可解释性与工程可控性，形成从架构到度量的因果闭环。

8.2 性能达成度

- 指标口径与评估方法：统一采用 `T_exec = Estimated Clock Period × Cosim Latency`，并同步记录 `Target/Estimated/Fmax/Latency/T_exec/Slack` 六项核心度量；资源侧以 BRAM/DSP/LUT/FF 评估容量与分布变化。
- SHA-256（含 HMAC 场景）：
    - Estimated 周期由 13.846 ns 降至 7.656 ns（↓44.7%），Fmax 由 66.67 MHz 提至 130.7 MHz（↑96.0%）；
    - Cosim 延迟由 809 降至 798（↓1.4%），`T_exec` 由 11201.4 ns 降至 6109.5 ns（↓45.5%）；
    - BRAM 60→9（↓85.0%），LUT 39189→16648（↓57.5%），FF 12977→15393（↑18.7%），DSP 保持 0；Slack 为 +0.08 ns（Pass）。
- LZ4 压缩：
    - Estimated 周期由 13.220 ns 降至 12.460 ns（↓5.7%），Fmax 由 66.67 MHz 提至 80.3 MHz（↑20.4%）；
    - Cosim 延迟由 3390 降至 1599（↓52.9%），`T_exec` 由 44815.8 ns 降至 19923.5 ns（↓55.5%）；
    - BRAM_18K 106→192（↑81.1%）、LUT 7235→11504（↑59.1%）、FF 3537→3760（↑6.3%），DSP 保持 0；Slack 为 +0.05 ns（Pass）。
- Cholesky（复数定点）：
    - Estimated 周期由 6.276 ns 降至 4.766 ns（↓24.1%），Fmax 由 159.3 MHz 提至 209.9 MHz（↑31.7%）；
    - Cosim 延迟由 4919 降至 1335（↓72.9%），`T_exec` 由 30871.6 ns 降至 6221.1 ns（↓79.9%）；
    - DSP 14→45（↑221.4%）、LUT 9223→13415（↑30.2%）、FF 4365→7310（↑40.3%），BRAM 保持 0；Slack 为 +0.09 ns（Pass）。
- 结论：三类算子均实现频率与延迟的协同改善，`T_exec` 均显著下降；资源变化可解释（以 DSP/LUT/BRAM 在算子重写与bank场景下的分布迁移为主），时序均保持收敛（Slack>0）。整体目标达成度高，具备可复现与可审计的工程证据链。

8.3 后续改进方向

SHA-256算法:

- Carry-Save/并行加法进一步引入：在 `sha256_iter` 中以 CSA 替代部分二元加法，保持三级结构不变但降低同级加法的临界路径；需与工具的时序模型配合验证。

- 消息调度的更深展开：对 `WT16/WT64` 引入适度 `unroll` 与寄存器切分，进一步分摊组合负载并稳定 `II=1`；结合 `STREAM depth` 缓解背压。
- 多消息并行与通道化：在 HMAC 复用场景下以多路 `sha256_top` 并行处理，评估跨消息的产消速率与流深度匹配，提升系统级吞吐。
- 资源与时序微调：在 Fabric/DSP 绑定策略上做小步实验（例如部分加法绑定 DSP），对比 Fmax 提升与 LUT/DSP 分布变化的收益，选择最优组合。

LZ4算法:

- BRAM 使用率优化：在保证吞吐的前提下，评估dict banking的 factor 与 BRAM/SRL/LUTRAM 组合，减少 BRAM 占用峰值，优化多核并行下的资源分布。
- 状态机进一步精简：在 Part2 引入小型子状态融合（保持 `II=1` 与依赖解除），减少分支与选择器层数，并结合 `STREAM depth` 细调产消节奏。
- 哈希路径与窗口策略：探索更轻量的哈希分段与位运算替代策略，结合匹配窗口的预取参数，提升字典命中质量与降低组合负载。
- 多块调度与汇聚：在 `mm2multStreamSize/multStream2MM` 端调整突发策略与对齐方式，减少搬移开销与队列驻留，提高整体并行度。

Cholesky算法:

- 混合精度与位宽自适应：在 `RECIP_DIAG_T` 采用更高位宽以降低 `rsqrt+mul` 的乘法恢复误差，非对角沿输出位宽运行；结合约束与综合报告评估资源-时序平衡。
- 树形归约与层级并行：对 `square_sum/product_sum_array` 引入分级加法树（保留 DSP 绑定、表达式平衡），进一步压缩加法链深度与提升 Fmax。
- 零化融合与拍数缩减：尝试在 II 不受影响情况下将零化循环与主计算融合为可选路径，减少后处理拍数；需对调度器的变长边界敏感性进行验证。
- 更大规模矩阵与稳定性评估：在更高维度矩阵下测试 NR 校正与 `eps` 策略的数值稳定性，必要时引入双次 NR 或自适应阈值。

---

9. 参考文献

[1] NIST. FIPS PUB 180-4: Secure Hash Standard (SHS). 2015. https://csrc.nist.gov/publications/detail/fips/180/4/final

[2] H. Krawczyk, M. Bellare, R. Canetti. RFC 2104: HMAC—Keyed-Hashing for Message Authentication. IETF, 1997. https://www.rfc-editor.org/rfc/rfc2104

[3] Yann Collet. LZ4 Frame & Block Format Documentation. 2019–2024. https://github.com/lz4/lz4/tree/dev/doc

[4] J. Ziv, A. Lempel. “A Universal Algorithm for Sequential Data Compression.” IEEE Trans. Inf. Theory, 23(3): 337–343, 1977.

[5] AMD Xilinx. Vitis High-Level Synthesis User Guide (UG1399). v2024.2. https://docs.amd.com/r/en-US/ug1399-vitis-hls

[6] ARM. AMBA 4 AXI4-Stream Protocol Specification. 2010. https://developer.arm.com/documentation/ihi0051

[7] AMD Xilinx. UltraScale DSP Slice (DSP48E2) User Guide (UG579). 2019.

[8] Xilinx. 7 Series FPGAs CLB User Guide (UG474) — Distributed RAM & SRL. 2016.

[9] G. H. Golub, C. F. Van Loan. Matrix Computations (4th ed.). Johns Hopkins University Press, 2013.

[10] N. J. Higham. Accuracy and Stability of Numerical Algorithms (2nd ed.). SIAM, 2002.

[11] K. Walczyk, J. Parniewicz. “Improving the Accuracy of the Fast Inverse Square Root.” Entropy, 23(9):1170, 2021.

[12] B. R. Rau. “Iterative Modulo Scheduling.” Int. J. Parallel Programming, 22(2):103–137, 1994.

[13] C. Xiong, C. Liu, H. Li, X. Li. “HLSPilot: LLM-based High-Level Synthesis.” 2024.