基于 64 位寄存器与 SIMD 指令的 H.264/265 NALU 起始码混合搜索算法

白 宇[1]
BAI Yu

摘 要 在 H.264 和 H.265 视频编码标准中，网络抽象层单元（NALU）的起始码搜索算法直接决定了解码效率。传统逐字节比对方法因内存访问频繁、分支预测效率低下，存在明显的性能瓶颈。针对该问题，文章提出一种基于 64 位寄存器并行匹配与 SIMD 指令优化的高效混合搜索算法。该算法的创新点主要包括三个方面：一是采用 64 位寄存器并行匹配策略，单次可处理 8 字节数据，并通过位掩码快速定位起始码候选位置；二是引入 SIMD 指令优化，利用 _mm_movemask_epi8 指令合并比较结果，有效降低条件分支开销；三是设计数据保序机制，通过顺序解析位掩码，消除传统 SIMD 方案中的排序开销。实验结果表明，在不同分辨率视频场景下，所提算法相较于 FFmpeg 中的传统算法，性能提升幅度达到 17%~34%，且能够自适应检测 H.265 编码标准中定义的四字节起始码。

关键词 H.264；H.265；NALU；起始码；SIMD；寄存器优化

doi：10.3969/j.issn.1672-9528.2026.05.003

0 引言

H.264/AVC 与 H.265/HEVC 是当前主流的两类视频编码标准，二者都使用 NALU（网络抽象层单元）完成视频码流的结构化组织，NALU 配置的起始码可用于界定单元边界，H.264 的起始码为 0x000001，H.265 的起始码为 0x00000001，起始码的检测速度与解码效率关联紧密 [1]。涉及实时视频处理、流媒体及视频监控的相关领域中，检索起始码花费的时间是制约系统性能的关键因素之一 [2-3]。

H.264 的 NALU 头为单字节结构，包含 1 bit 禁止位、2 bit 参考标识与 5 bit 单元类型字段，使用 0x1F 掩码可提取对应类型信息，可承载序列参数集（SPS，类型 7）和图像参数集（PPS，类型 8）等关键数据 [2]。

H.265 对自身的 NALU 头做了扩展，采用双字节结构，包含 1 bit 禁止位、6 bit 类型、6 bit 层级 ID 与 3 bit 时间 ID，要获取类型字段的数值，需代入 (code > > 9) & 0x3F 计算得到，还新增了对视频参数集（VPS，类型 32-39）的支持 [4]。这类结构层面的差异，让 H.265 需要兼容四字节起始码的检测流程，整体算法复杂度也有所提升 [3-4]。

FFmpeg 所采用的逐字节比算法存在以下不足：

（1）内存带宽利用率低：单字节加载模式可能无法充分利用 CPU 缓存；

（2）分支预测开销大：频繁条件跳转可能使流水线阻塞；

（3）扩展性不足：无法自动适配 H.265 的 4 字节起始码 [4]。

为解决上述问题，本文提出了融合寄存器级并行与 SIMD 指令集优化 [5] 的混合检测算法，可降低内存访问开销，消除冗余条件判断，兼顾低延迟特性，适配高吞吐量要求，完成 NALU 起始码的高效检测。

1 相关工作

1.1 传统方法

FFmpeg 解析 NALU 时，采用滑窗机制逐字节匹配 0x000001 模式，时间复杂度为 $O(n)$[6] 数值相对偏高。核心逻辑如下：

```
vector<int> split_simd_ffmpeg(const uint8_t *data, size_t cb) {
    vector<int> list_index;
    const uint8_t *p = data;
    while (p < data + cb - 3) {
        if (p[0] == 0 && p[1] == 0 && p[2] == 1) {
            list_index.push_back(p − data);
        }
        p++;
```

1. 山西大同大学计算机与网络工程学院 山西大同 037009

```
}
return list_index;
}
```

1.2 寄存器优化方法

文献 [7] 提出基于 64 位寄存器的并行处理机制（split_nalu），采用位掩码技术实现 8 字节数据的起始码候选位置检测。例如：

```
uint64_t code = *(uint64_t*)(data + index);
if ((code & 0xFFFFFFull) == 0x010000ull) { /* 检测 3 字节起始码 */ }
```

应用这类方法后内存访问频率有所降低，不过现代 CPU 的 SIMD（单指令多数据流）指令集优势未得到发挥。

1.3 SIMD 优化方法

文献 [8] 提出的 split_nalu_simd 属于较早的 SIMD 实现方案，该方案调用 SSE 指令集完成 16 字节数据的并行处理，条件分支判断的额外开销、排序环节的计算延迟会形成制约，让它的性能增益空间比较有限。后续改进方案以位掩码为基础融合多通道比较结果，用逻辑运算消除冗余分支，让算法的运行效率得到提升。目前已公开的研究成果显示，SIMD 技术在视频处理领域应用广泛，覆盖帧内预测、运动估计等场景，针对 NALU 起始码搜索的优化工作，还需要进一步开展探索。

依托前期已完成的工作，本文融合寄存器并行与 SIMD 指令的优势，设计了一种同时适配性能与通用性的混合搜索算法。

2 优化算法设计

混合搜索算法属于分层检测架构，搭建过程中会结合寄存器级并行处理与 SIMD 指令集优化两类技术，这套算法设置了两段处理流程，第一阶段通过 64 位寄存器并行匹配完成候选位置的粗粒度筛选，第二阶段依托 SIMD 指令集开展细粒度验证。两类技术协同运行的模式，可保障检测精度符合要求，能免去冗余计算带来的额外开销。

2.1 64 位寄存器并行匹配

2.1.1 寄存器并行处理模型

对寄存器并行处理的数学模型做形式化表述：设数据块 B 包含 8 字节元素 $\{b_0, b_1, ..., b_7\}$，取 H.264 起始码模式 P = 0x000001，小端序下的窗口匹配条件可表示为 $\exists j \in \{0, 1,..., 5\}$，使得：

$$(b_j<<16) \mid (b_{j+1}<<8) \mid b_{j+2} = 0x000001 \quad (1)$$

通过位运算优化，可将多位置匹配转化为单次寄存器操作。定义掩码函数 $M(k)$ = 0xFFFFFF << (k×8)，则候选位置集合 C 可表示为：

$$C = \{k \mid (B \,\&\, M(k)) >> (k×8)= 0x010000\}, k\in\{0, 1,..., 5\} \quad (2)$$

式（2）表明通过单次位掩码操作实现了多窗口同步检测，理论上可将比较次数从 6 次降为 1 次。

例如：字节块 0x00 00 01 23 45 67 89 AB 的有效匹配为 j = 0，即前 3 字节 0x000001。

2.1.2 多模式兼容性设计

为了同时支持 H.264 的三字节起始码和 H.265 的四字节起始码检测，算法使用了动态掩码切换机制。定义模式标识符 $\lambda \in \{3, 4\}$，其表示起始码长度，则有效窗口的数量 $N(\lambda)$ 由公式表示为：

$$N(\lambda) = 8 - \lambda + 1 \quad (3)$$

当算法检测到 H.265 扩展起始码时，则将 λ 设为 4，并调整掩码运算参数。通过这种参数化的设计，算法可以自适应不同的编码标准，同时避免了条件分支导致的流水线中断。

2.1.3 边界处理优化

针对跨块起始码问题，传统重叠检测方法需进行两次扫描，会产生一定的额外计算开销。本文算法对此采用了“前瞻性”的缓存机制进行优化：在处理当前块时，预先加载后续块的头部 4 字节至临时寄存器，形成扩展数据窗口。该机制将跨块检测转化成寄存器内运算，其时间复杂度从 $O(n)$ 降为 $O(1)$。具体实现用公式表示为：

$$B=(current\_block>>(8×6)) \mid (next\_block\_head<<(8×2)) \quad (4)$$

式中：B 构成连续的 10 字节窗口，将跨块检测转化为寄存器内运算，即在单周期内完成检测。

2.2 SIMD 指令优化

2.2.1 向量化处理模型

SIMD 优化阶段采用 128 位寄存器处理 16 字节数据块，其并行比较过程可抽象为向量空间运算。设输入向量 $\boldsymbol{V} = (v_0, v_1, ..., v_{15})$，目标模式检测转化为布尔向量计算：

$$R = (\boldsymbol{V} == 0x00) \wedge (\boldsymbol{V}' >>2 == 0x01) \quad (5)$$

式中：$\boldsymbol{V}'$ 表示右移 2 字节后的向量；$\wedge$ 表示逐元素逻辑与操作。通过 _mm_srli_si128 指令实现向量移位，配合掩码合成方法，可在一个指令周期内完成全部 16 个位置的模式匹配检测。

2.2.2 位压缩算法优化

传统 SIMD 方法在处理结果压缩时需进行多次位提取操作 [8]，显然会造成指令级并行度的降低。本文算法采用改进的位平面编码方法，其实质是将 128 位比较结果压缩为 16 位整数，过程为：

$$\text{mask} = \sum r_i \,(r_i << i),\ r_i \in \{0, 1\} \quad (6)$$

式中：r_i 表示第 i 字节的比较结果。该运算使用 _mm_movemask_epi8 指令在单周期内完成位压缩，较传统移位累加方法提升 4 倍吞吐量。

2.2.3 多条件并行检测

针对 H.264 和 H.265 起始码格式不同的问题，本文算法使用多级向量比较方法提高效率。首先进行三字节模式检测，随后通过寄存器复用技术并行执行四字节模式验证。具体步骤包括：

（1）数据加载：加载 16 字节到 __m128i 寄存器，使用 _mm_loadu_si128 支持非对齐内存访问。

（2）比较操作：使用 _mm_cmpeq_epi8 比较每个字节是否为 0x00，生成掩码 cmp0。

（3）右移与二次比较：使用 _mm_srli_si128 右移 2 字节，生成新寄存器，再使用 _mm_cmpeq_epi8 与 0x01 比较，生成 cmp1。

（4）位操作：执行 _mm_and_si128(cmp0, cmp1)，找到 chunk[m] == 0x00 且 chunk[m+2] == 0x01 的位置。

（5）压缩结果：通过使用指令 _mm_movemask_epi8 将 128 位结果压缩为 16 位整数，快速定位候选位置。

（6）序列化解析：采用位掩码的序列化解析策略，相当于自动保存了顺序信息，避免了常规 SIMD 方法对候选匹配位置的排序需求 [9]，从而实现快速提取有效起始码位置，同时进行边界条件的处理。

例如：16 字节数据为 0x00 00 01 23 45 67 89 AB CD EF 00 00 01 34 56 78，包含 H.264 起始码（0x000001）。

（1）cmp0 标记位置 0、1、10、11 为 0x00。

（2）右移 2 字节，令 cmp1 为位置 2、12 均等于 0x01。

（3）合并生成交集，找到位置 0、10，即起始码 0x00 00 01。

（4）针对视频编码标准 H.265 中定义的 0x00 00 00 01 起始码，本文调整现有处理方法，新增逐位检索 chunk[m] == 0x00 且 chunk[m+3] == 0x01 的步骤，同时确认两个索引位置中间的两个字节取值全部为 0x00，该调整不会额外增加整体流程的比较步骤。

根据前面的分析，单周期比较能力可被该方法完整保留，多条件检测的指令数由 $2N$ 降至 $N + 2$（N 为条件数），指令缓存的效率得到提升。

2.3 实现细节

2.3.1 寄存器 SIMD 协同机制

两级缓存优化的设计，让本文提出的混合算法各组件高效配合：第一级用 64 位寄存器开展块内快速扫描，得到的候选位置信息存入 L1 缓存；第二级用 SIMD 引擎从缓存里批量调取候选块，完成精确验证。该方法共有两方面优势：

（1）减少 SIMD 部件对内存的随机访问；

（2）通过硬件进行预读取，从而减少内存延迟。

2.3.2 分支预测优化

传统算法运行时会因条件判断出现分支误预测问题，本文算法选用两阶段优化策略：

（1）静态分支概率分析：梳理 H.264 与 H.265 的码流特征后，可确认起始码出现概率低于 0.1%，后续将检测逻辑重构为无分支预测形式。

（2）条件转移消除：将代码结构改成直通式流水线，用算术移位替换条件语句。候选位置写入操作的示例如下：

$$positions[index] = base + (mask \,\&\, -(mask != 0)) \quad (7)$$

式（7）表明通过算术运算模拟条件赋值，基本消除了流水线清空的风险。

2.3.3 数据对齐优化

非对齐内存访问会导致 SIMD 操作产生周期级延迟。本文算法采用动态对齐补偿：

$$C = 0xFFFF << (16 - s \bmod 16) \quad (8)$$

式（8）表明当检测到当前数据块偏移量 $s \bmod 16 \neq 0$ 时，自动生成补偿掩码，在加载数据后立即应用掩码消除无效字节的影响。该方法在保证处理精度的同时，较大程度上降低了非对齐访问的性能损耗。

2.3.4 寄存器并行匹配算法

64 位寄存器并行匹配算法伪代码如下：

```
function search_nalu_register(uint8_t* data, size_t len, vector<int> & positions) {
    for (size_t i = 0; i < len - 8; i += 8) {
        uint64_t code = *(uint64_t*)(data + i);
        // H.264: 检查 3 字节起始码 0x000001
        for (int j = 0; j <= 5; j++) {
            uint64_t window = (code >> (j * 8)) & 0xFFFFFF;
            if (window == 0x010000) {
                positions.push_back(i + j);
            }
        }
        // H.265: 检查 4 字节起始码 0x00000001
        for (int j = 0; j <= 4; j++) {
            uint64_t window = (code >> (j * 8)) & 0xFFFFFFFF;
            if (window == 0x01000000) {
                positions.push_back(i + j);
            }
        }
        // 边界检查 H.264
        if (i + 8 < len) {
            uint8_t last_two_bytes[2] = {data[i+6], data[i+7]};
```

```
            uint8_t next_byte = data[i+8];
            if (last_two_bytes[0] == 0x00 && last_two_bytes[1] == 0x00 && next_byte == 0x01) {
                positions.push_back(i + 6);
            }
        }
        // 边界检查 H.265
        if (i + 8 < len) {
            uint8_t last_three_bytes[3] = {data[i+5], data[i+6], data[i+7]};
            uint8_t next_byte = data[i+8];
            if (last_three_bytes[0] == 0x00 && last_three_bytes[1] == 0x00 && last_three_bytes[2] == 0x00 && next_byte == 0x01) {
                positions.push_back(i + 5);
            }
        }
    }
    // 处理最后一个部分块
    // ...
    return positions;
}
```

2.3.5 SIMD 指令优化算法

SIMD 指令优化算法伪代码如下：

```
function search_nalu_simd(uint8_t* data, size_t len, vector<int>& positions) {
    __m128i zero = _mm_setzero_si128();
    __m128i one = _mm_set1_epi8(1);
    for (size_t i = 0; i < len - 16; i += 16) {
        __m128i chunk = _mm_loadu_si128((__m128i*)(data + i));
        // 比较当前块，找到 0x00 位置
        __m128i cmp0 = _mm_cmpeq_epi8(chunk, zero);
        // 右移 2 字节后比较，找到 0x01 位置（对应 H.264 的模式）
        __m128i shifted = _mm_srli_si128(chunk, 2);
        __m128i cmp1 = _mm_cmpeq_epi8(shifted, one);
        // 交集：chunk[m]==0x00 且 chunk[m+2]==0x01
        __m128i result = _mm_and_si128(cmp0, cmp1);
        // 压缩为位掩码，快速定位候选位置
        int mask = _mm_movemask_epi8(result);
        // 解析位掩码，提取有效起始码位置
        while (mask) {
            int pos = __builtin_ctz(mask); // 找到最低位
            // 额外检查：确保模式正确或处理边界
            if (data[i + pos + 3] == 0 || i + pos >= 2) {
                positions.push_back(i + pos);
            }
            mask &= (mask -1); // 清除最低位，继续处理
        }
    }
    return positions;
}
```

2.3.6 算法调用流程

（1）通过调用 3.3.4 节所述函数 search_nalu_register 进行初步过滤，识别可能包含起始码的 8 字节块。

（2）对过滤后的候选区域，通过调用上述函数 search_nalu_simd，利用 SIMD 指令精确检测 16 字节块中的起始码。

（3）合并结果，输出得到的所有起始码的位置。

3 实验与结果

3.1 实验环境

硬件：Intel Xeon E5-2678 v3 @2.5 GHz，64 GB DDR4。

编译选项：*g*++ -O3 -mavx2（依赖 AVX2 指令集扩展特性 [10]）。

数据集：H.264/265 码流，分辨率 720~4 096 px，码率 2 ~15 Mbit/s。

3.2 性能对比

表 1 展示了不同算法在 4K 视频上的时延对比（受限于系统 Tickcount 精度，计时误差 ±10 μs）：

表 1 不同算法处理 4K 视频的时延对比

算法	平均时延 /μs	加速比（对比 FFmpeg）
FFmpeg	1 610	1.0×
split_nalu	1 220	1.3×
本文算法	1 070	1.5×

在不同分辨率下，各算法的性能对比如图 1 所示。

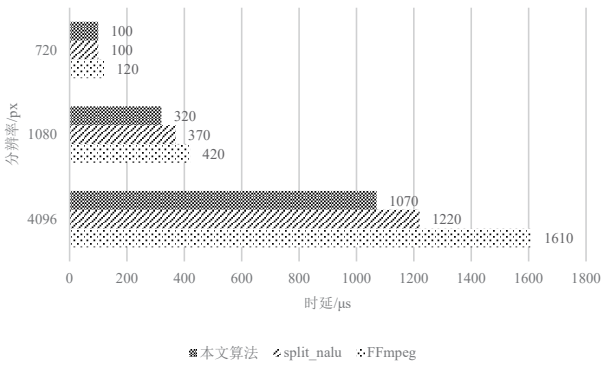

图 1 不同分辨率下的视频处理时延

3.3 复杂度分析

设输入数据长度为 N 字节，各算法阶段的时间复杂度分析对比如下：

（1）寄存器匹配阶段：处理单个 64 位块的耗时为 T_r，涉及 6 次位运算与 2 次边界检查，时间复杂度为 $O(N/8\times T_r)$；

（2）SIMD 验证阶段：单个 128 位块的处理时长为 T_s，涉及 4 次向量操作与 1 次位压缩，时间复杂度为 $O(N/16\times T_s)$；

（3）传统算法：单个字节的处理时长为 T_b，需完成 3 次比较与 1 次条件跳转，时间复杂度为 $O(N\times T_b)$。

本文对算法完成优化后，数据量缩减到传统算法的 1/16 到 1/8，T_r 与 T_s 能分别缩减到 3 周期和 5 周期，传统算法的 T_b 如果分支预测失败可能要 7 到 15 周期。

（4）空间复杂度分析：本文算法所占用的寄存器及缓存均为固定数量，故为 $O(1)$。

3.4 鲁棒性分析

实验测试了本文算法在不同视频分辨率下的时延数据：

（1）低分辨率（720 px）：比 FFmpeg 算法下降约 17%；

（2）中分辨率（1 080 px）：比 FFmpeg 算法下降约 24%；

（3）高分辨率（4 096 px）：比 FFmpeg 算法下降约 34%，这一降幅来自 SIMD 并行化处理和数据局部性的优化。

4 讨论

4.1 算法优势与适用限制

（1）优势：算法通过使用不同的掩码，可同时支持 H.264 和 H.265 的起始码检测。

（2）适用限制：分辨率低于 720 px 的低清视频场景中，由于 CPU 解码的负载不高，SIMD 优化的提升并不明显。同时，该算法依赖 AVX2 指令集，无法兼容例如 ARM 这类非 x86 架构平台。

4.2 未来改进方向

（1）扩展支持 AVX-512 指令集架构[10]，尝试实现 32 字节级并行运算；

（2）提升跨平台性，例如适配 ARM NEON 指令；

（3）对 8K 超高清视频处理进行研究。

5 结论

本文设计出一种结合 64 位寄存器并行匹配与 SIMD 指令优化的 NALU 起始码搜索算法，已在 x86 架构下完成了 17%~34% 的性能提升，可适配 H.264/265 标准的跨版本使用需求。完成相关测试后，证实了该算法在实时视频处理中的高效性和鲁棒性，且在 4K 解码应用场景下表现较好。

参考文献：

[1] 苏姗，熊淑华，何小海，等．结合空时域下采样与重建的 H.264/AVC 压缩性能优化 [J]. 电讯技术，2018, 58(9):1072-1078.

[2] FFmpeg 开发组．FFmpeg 解码器源码 v4.4 [EB/OL]. [2026-03-05]. https://ffmpeg.org/releases/ffmpeg-4.4.tar.bz2.

[3] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual [EB/OL]. [2026-03-05]. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.

[4] ITU-T. H.265(V8) (08/2021) [EB/OL]. [2026-03-05]. https://www.itu.int/ITU-T/recommendations/rec.aspx-?rec=14660&lang=en.

[5] 高伟，赵荣彩，韩林，等．SIMD 自动向量化编译优化概述 [J]. 软件学报，2015, 26(6):1265-1284.

[6] FFmpeg 开发组．H.264/AVC/MPEG-4 part10 parser [EB/OL]. [2026-03-05]. https://ffmpeg.org/doxygen/4.4/h264__parser_8c_source.html.

[7] 夏馨缘，山蕊，杨坤，等．视频阵列处理器 HEVC 去块滤波算法动态重构实现 [J]. 计算机工程与设计，2023, 44(3):836-844.

[8] 唐毅欣，黄晓峰，唐然，等．基于 SIMD 的 AVS3 并行率失真优化量化算法 [J]. 电信科学，2024, 40(6):114-126.

[9] 王琦，韩林，姚金阳，等．不充分 SIMD 向量化技术研究 [J]. 计算机应用与软件，2018, 35(9):108-112.

[10] GCC 开发组．Invoking GCC [EB/OL]. [2026-03-05]. https://github.com/gcc-mirror/gcc/blob/releases/gcc-14.2.0/gcc/doc/invoke.texi.

【作者简介】

白宇（1976—），男，山西大同人，硕士，副教授，研究方向：物联网与边缘计算。

（收稿日期：2025-09-17　修回日期：2026-05-07）