arXiv:2601.17221v1 [cs.DB] 23 Jan 2026

Vidformer：用于渲染视频原生查询结果的即插即用声明式优化

Dominik Winecki 俄亥俄州立大学 美国俄亥俄州哥伦布 winecki.1@osu.edu

Arnab Nandi 俄亥俄州立大学 美国俄亥俄州哥伦布 nandi.9@osu.edu

摘要

交互式探索视频数据时，原生视频查询需要将查询结果作为视频来使用，其中包括汇编提取的视频片段或添加数据叠加层等步骤。这类原生视频查询的瓶颈在于渲染，而非底层查询的执行。目前，渲染通常通过速度较慢的后处理脚本完成。这一步给交互式视频数据工作负载带来了关键阻碍：即使是短片段也包含数千个高清帧；传统的 OpenCV/Python 脚本必须先解码 → 转换 → 编码整个数据流，才能显示出第一个像素，让用户等待数秒、数分钟甚至数小时。

为解决这些问题，我们推出了 Vidformer，这是一款面向原生视频查询的即插即用渲染加速器，它能够：(i) 透明地将现有可视化代码提升 为声明式表示；(ii) 透明地优化并并行化渲染；(iii) 通过视频点播协议即时提供视频，并进行即时分段渲染。我们证明，Vidformer 能将各种标注工作负载的完整渲染时间缩短 2–3×，更重要的是，将开始播放时间降至 0.25–0.5 秒。这意味着性能提升了 400×，使片段长度与首帧播放延迟脱钩，并使亚秒级延迟的交互式原生视频查询成为可能。此外，我们还展示了该方法如何支持基于大语言模型的交互式原生视频对话式查询。

关键词

视频数据可视化、多模态查询、声明式视频编辑

1 引言

近年来，随着计算机视觉和多模态数据处理取得巨大进展，视频数据已无处不在。如今，特定领域的计算机视觉模型只需适度的计算资源，性能便已超越人类；与此同时，现代大语言模型（LLM）和视觉语言模型（VLM）本质上具有多模态能力，可实现与文本、图像、音频和视频的零样本交互。此外，视频数据库管理系统（VDBMS）在运行模型方面也取得了显著进展

处理视频 [26, 28]，在很大程度上解决了大规模推理问题。这些进展使视频数据管道中其他环节的局限性显现出来。我们聚焦于一个显著的瓶颈：渲染视频/数据可视化视频。

在计算机视觉中，为视频添加模型结果标注是调试和理解模型的一项关键技术。边界框和对象掩码有助于直观地了解检测和分割模型的优势与局限性。

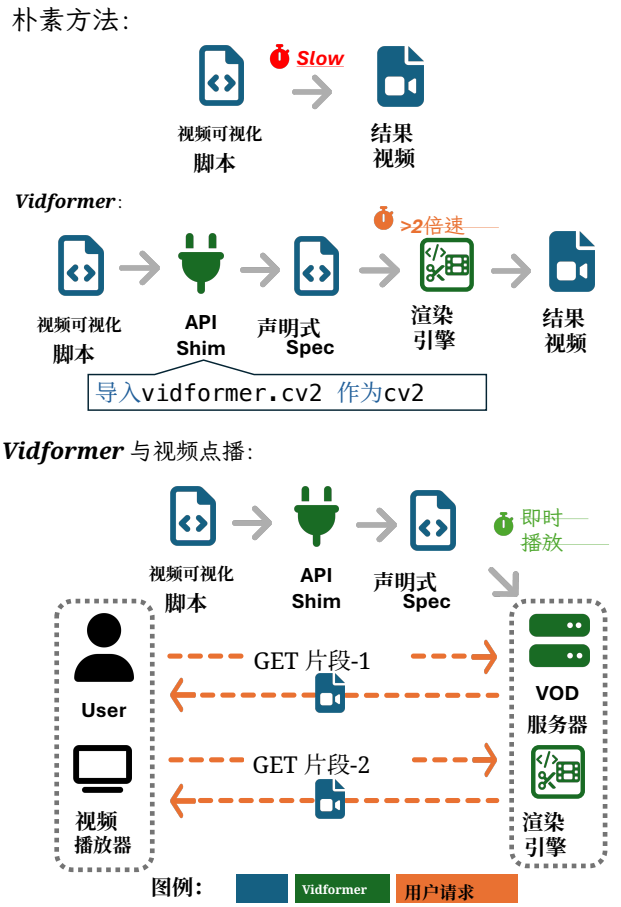

图 1：渲染可视化视频的朴素方法极其缓慢。*Vidformer* 使用即插即用的 API 垫片，将命令式代码转换为声明式规范，从而高效加速结果渲染。随后，Vidformer 的视频点播服务器会即时生成各个分段。

此外，注意力热图可以直观呈现各项特征的影响。类似地，视频数据库擅长回答有关视频数据集的查询，例如哪些视频或帧符合搜索条件，并返回可用于创建结果汇编视频的关系型/表格结果。许多其他应用也会创建类似的视频数据可视化，例如在无人机影像上叠加相关遥测信息，或将装配线上的制造传感器数据与视频流同步。这些用例都有一个共同目标：将视频流

与有关它们的传统数据结合起来，从而创建新颖且增强的视频可视化。这类视频数据可视化工作流广泛应用于科学、商业和工业领域。

我们发现，这些多种多样的任务几乎都有相同的实现方式：使用通用编程语言编写脚本来执行编辑，从而生成经过转换的可视化视频。这种方法与传统数据处理工具具有无可比拟的互操作性，同时还能充分发挥图灵完备脚本的无限表达能力。然而，这种渲染可视化视频的方法速度很慢；即使是快速的可视化脚本，导出视频也需要数秒甚至数分钟。这显著拖慢了迭代速度，并促使用户尽量少用可视化视频；即使使用，也只制作较短的视频。这种糟糕的体验主要源于工具的不足。在商用硬件上运行 Adobe Premiere Pro 或 Apple Final Cut 等传统视频编辑器，并实时观看编辑后的视频，显然说明了工具方面存在的差距。在本文中，我们详细介绍一种新颖的解决方案，可让用户即时查看由脚本定义的视频数据可视化。

要加速视频数据可视化的渲染，必须解决表达能力与控制流复杂性之间的冲突。对于简单任务，可以将功能硬编码到界面中，以实现合理的性能，例如在视频首次编码时将叠加层烘焙进去。然而，随着功能增加，实现复杂度会呈指数级增长，因此这种方法无法适应更复杂的任务。定制视频数据可视化脚本（例如图 2 所示的脚本）具有出色的表达能力，并能与数据、计算机视觉和多媒体生态系统良好互操作。然而，这些脚本会完整渲染输出视频，并将命令式控制流与无限复杂性紧密结合，从而排除了大多数透明优化的机会。同样，虽然可以将高效算法纳入这些脚本，但这会显著增加其复杂性。即使经过完美优化，这类硬件加速可视化脚本也无法达到所需速度，因为它们仍会完整渲染结果视频。

随着视频数据工作流不断增多，解决这些挑战至关重要。视频已成为众多科学、商业和消费类应用中的重要数据类型。尽管视频数据量不断增长，但与其他数据类型相比，视频在交互式界面方面的创新却相对滞后，导致视频数据的探索和分析速度缓慢。快速呈现视频数据可视化，是弥合这一差距的关键一步。除了能够立即加速现有工作负载之外，其次级效应还在于推动新交互范式的出现。长期以来，这一直是一个先有鸡还是先有蛋的问题：由于新功能和应用在实际使用中速度太慢，因此无人开发；而由于缺乏需要性能提升的功能或应用，提升速度也缺乏动力。解决视频数据可视化的性能挑战，便能在此基础上构建新颖的体验。具体而言，就是大语言模型驱动的视频原生查询，我们将在第 6.3 节详细介绍。

本文提出了 Vidformer，这是一种可直接集成的新方法，可加速视频数据可视化脚本。我们的方法将命令式视频数据可视化代码转换为声明式规范，然后高效地对其进行优化、并行化和执行，以提供经过转换的可视化视频。我们首先详细介绍如何通过符号计算将现有代码提升为声明式表示（第 4 节）。我们为最流行的此类库——OpenCV 的 Python 库——实现了一个 API 兼容层，只需一行导入语句 `import vidformer.cv2 as cv2`，即可转换现有代码。随后，我们详细介绍渲染引擎的设计（第 5 节）；该引擎执行视频的声明式变换，正如数据库管理系统执行 SQL 来变换关系数据。最后，我们提出一种利用视频点播（VOD）流式传输协议提供延迟生成的视频结果的方法（第 6 节）。较短的片段会按需在恰当时机惰性物化。

从用户的角度来看，视频可以立即使用；用户可以像以前一样播放视频并随意跳转。最重要的是，我们专注于打造一种灵活通用的解决方案；要让系统发挥作用，它不能只在典型情况下有效，而必须在所有情况下都有效，因此我们着重考虑最坏情况下的视频访问模式。通过设计通用解决方案，即使面对最复杂的任务，我们的系统在视频标注、转换和可视化方面也能像通用脚本语言一样灵活表达。

Vidformer 将视频转换和渲染视为数据处理任务，在保持现有脚本语言完整表达能力的同时，以透明的方式扩展渲染控制流，从而优化并延迟可视化视频的物化。我们的评估表明，声明式提升和我们的渲染引擎可将渲染速度提升 $2-3\times$，并以快于 400× 倍的速度流式传输结果，使用户能够在不到一秒的时间内开始播放。

2 相关工作

计算机视觉模型输出的可视化。我们发现，Supervision [46] 是使用计算机视觉结果标注图像和视频的主流高层工具。模型结果主要来自 YOLO [45] 和 SAM [29] 的衍生模型，可以加载到一种通用检测格式中，该格式存储边界框、掩码、分类结果、置信度分数和目标跟踪 ID。随后可以使用不同的标注器，在底层帧上可视化检测结果。Supervision 内部基于 OpenCV 构建，我们在评估中使用了 Supervision 的标注器。在我们的综述中，几乎所有计算机视觉可视化都使用了 Supervision 或 OpenCV，唯一的例外是 PyTorch 的 [41] transforms 模块。这种普及性至关重要：大语言模型训练数据集中充斥着使用 OpenCV 和 Supervision 的代码，因此大语言模型能够熟练使用这些库。

返回视频的视频数据库。 许多视频数据库可以返回视频片段 [10, 13, 23, 26–28, 30, 51, 52], , 通常是裁剪后的分段，但只返回裁剪时间跨度的指针 ，而不返回实际视频。ApertureDB [1] 提供了用于从视频中提取片段的专用命令。此外，少数系统提供了有限的视频标注支持，但这几乎完全通过临时的、独立的 OpenCV [12]脚本实现。例如，EVA [26]和 Spatialyze [30] 都包含使用 OpenCV 绘制边界框的 Python 脚本。

声明式视频编辑。 将数据融入视频编辑的探索，始于对时间序列视频流 [18, 34–36], 进行视频标注；Mackay & Davenport 指出，“视频变成了一种信息流，一种可以被标记和编辑、分析和标注的数据类型” [36]。值得注意的是，DIVA [35] 提供了用于标注和重新排列视频流片段的“流代数”；据我们所知，它是第一个声明式视频编辑器（DVE）系统。在数据标注任务之外，DVE 和基于脚本的编辑也被用于传统的多媒体编辑 [9, 37]。随着 Web 的兴起，众多采用 DVE 概念的媒体平台应运而生。Orcc [49] 提供了基于图形数据流的编辑功能。NUBOMEDIA [20] 是一个基于 WebRTC 的媒体应用数据平面，而 Hop [47] 为此提供了一种统一语言和 Web 框架。类似地，Videostrates [31] 是一个协作式视频编辑工具包，提供声明式 HTML 和 CSS 界面来转换媒体。我们此前的工作 V2V [48], 提出了构建一种专用于从查询创建视频结果的 DVE 的构想。它将视频编辑建模为对帧应用滤镜的组合，并将视频表示为由领域特定语言（DSL）表示的（时间戳 × 帧表达式）对序列。Vidformer 在这一构想的基础上，将表达能力扩展到超越 DSL 的局限性，同时将渲染时间缩短到可用水平。

在开源领域，*editly* [17] 是一款基于 JSON 的领域特定语言，用于编辑视频；而 revideo [3], 是 remotion [4], 的开源替代方案，提供 TypeScript/React 界面。在业界，各家公司正通过应用程序接口提供声明式视频编辑服务，用于内容创作 [2, 4, 5], : 这些接口接收 JSON 规范并返回视频。VideoDB [6]支持通过 HLS 流式传输视频结果，以便创建汇编；但这项功能被硬编码在其应用程序接口中。这些工具都是为传统视频编辑任务设计的，缺乏完成哪怕简单标注所需的表达能力。

即插即用方案的必要性： 数十年来，声明式视频编辑一直时断时续地受到研究。尽管视频数据的规模和应用数量大幅增长，现有系统仍寥寥无几，也鲜有人使用。我们认为，这是因为研究者专注于创建声明式领域特定语言，而这类语言难以被采用。对于传统视频编辑任务，这类语言既无法与图形化编辑器（如 Adobe Premiere Pro、Apple Final Cut）的界面相竞争，也缺乏足够的表达能力，无法胜任数据标注等程序化任务。本文基于一种即插即用的声明式内部模型，利用其性能优势，同时像现有图形化视频编辑器那样对用户隐藏该模型。这种方法类似于 Modin [42], , 它将自身定位为可即插即用的替代方案，用于扩展 Pandas 代码。Vidformer 运用本文讨论的一系列技术扩展和优化视频管道，同时抽象掉实现细节，让用户能够毫无阻碍地采用数据库技术。

自动视频编辑。 自动编辑视频是一个成熟的问题 [11]。其中许多是常见的 精彩集锦风格剪辑 [9], , 由一段段短片组成的蒙太奇。此外，近期直接利用机器学习进行自动视频编辑的研究也大幅增加 [53], , 例如自动编辑风格迁移 [19]。这表明，人们有兴趣根据特定用户需求，在广泛的领域中制作实用视频。然而，这些研究只关注如何以逻辑方式编辑视频，而不涉及如何实际生成渲染后的结果。

符号数据处理库。 符号数据处理是计算密集型操作的一项基础技术。TensorFlow [7]和 PyTorch [41]等机器学习库会对张量执行符号运算，从而支持透明的梯度计算，并通过 GPU/TPU 加速。类似地，Apache Spark [50]允许对数据集进行符号变换，并在计算集群上执行。尽管这些系统采用了需要用户投入一定精力的专用接口，较新的方法则利用 API 兼容性，以极少的用户输入加速任务。Modin [42] 是一种即插即用的数据框处理加速器，支持多个后端。它采用 API 兼容层，使用户只需在 Python 中用一行代码 `import modin.pandas as pandas` 即可轻松采用。CuPy [39] 也以相同方式利用 GPU 加速 NumPy/SciPy 代码。与这些先前工作类似，我们通过单行 Python 导入语句提供 API 垫片，并使用符号化的 n 维数组引用，以声明式方式建模操作和变换。

渐进式与在线结果。 在非视频分析查询场景中，即时消费结果一直备受重视，因此催生了在线聚合技术：在查询扫描数据时，返回统计上有效的部分答案 [21, 22, 25]。因此，要实现真正交互式的视频原生查询，就需要新的算子流水线、优化方法和惰性物化策略，例如我们在本文中提出的方法。

3 问题定义

我们的目标是透明地加速视频数据可视化脚本的渲染。这些脚本使用通用编程语言 Python 编写代码，通过组合现有视频和其他数据来创建新视频。例如，在视频上绘制边界框以理解计算机视觉模型，或汇编与搜索查询匹配的视频片段。我们的任务是尽可能缩短向用户呈现此类脚本输出的开始播放延迟。

尽管应用范围广泛，但它们还存在一些共同的运行环境约束，必须予以考虑：

- **正确性：** 无论复杂程度如何，输出都必须与现有方法逐像素完全一致。
- **嵌入式：** 视频数据可视化存在于更大型的界面中。例如，计算机视觉工作通常在交互式笔记本中进行，而视频搜索结果则集成在更大型的 Web 应用程序中。
- **网络传输：** 应用程序通常运行在远程系统上，并通过（可能较慢的）网络提供服务。

这些限制确保最终系统能够在现有应用程序中实际使用。

4 视频数据可视化脚本的声明式提升

我们介绍了一种将现有的命令式视频处理代码提升为声明式规范的方法。我们以 Python OpenCV 库进行演示，因为它被广泛使用，每月下载量达 2600 万次。在 GitHub 上搜索发现，有 390 万个文件使用了该库，其中 9.98 万个文件创建了一个视频写入器

```
导入 cv2  #OpenCV导入 pandas  作为pd  # 数据框库

df = pd  . 读取_csv （"objects  .
    csv"） 捕获  =cv2. VideoCapture("in.mp4") 写
入器  =cv2. VideoWriter("out.mp4", ...)

i   = 0当为真时:ret , frame  =cap.read ()如果不是 ret: 中断
cv2. putText (frame , f"This is frame{i}", ...)对于 _, row 在 df[
df.frame  == i]. iterrows ():x1 , y1 , x2 , y2  =row["xyxy"]
cv2.rectangle(frame , (x1 , y1), (x2 , y2),...)writer . write(
frame)i   += 1cap.release ()writer. release ()
```

图 2：使用 OpenCV 编写的典型 Python 视频数据可视化脚本。每一帧都标注了帧编号，并在每帧上绘制来自标注文件的边界框。

```
vidformer . pixfmt(opencv. rectangle
(opencv. putText (vidformer .pixfmt (〈来自
in.mp4 的第 0 帧〉, from ="yuv420p", to ="
bgr24"),"这是第 0 帧",...) ,(10, 30) ,(200 ,
50),...) ,from ="bgr24", to ="yuv420p")
```

图 3：对图 2 中脚本生成的第一个输出帧，其声明式规范的内部表示（在进行单行导入 vidformer.cv2 as cv2 的修改后）。OpenCV 函数与常量参数组合，底层帧则以声明式方式引用。

对象。这表明，大约有 2.6% 的 OpenCV 用户正在执行某种视频数据可视化、标注或编辑任务。图 2 展示了一个此类脚本示例。

4.1 声明式数据模型

我们通过存储每个输出帧的构造过程来表示视频变换。每个输出帧都表示为一个帧表达式，由滤波器函数、常量数据值和输入帧引用组合而成。例如，图 2 中的脚本生成的第一帧如图 3 所示。其中，“opencv.putText”、“opencv.rectangle”和“vidformer.pixfmt”是滤波器函数，〈来自 in.mp4 的第 0 帧〉是对 "in.mp4"中第一帧的输入帧引用，其余字符串、整数和元组则是常量数据值。我们将每个输出帧的帧表达式存储为一棵抽象语法树（AST）。由于帧表达式嵌套很深、冗长且重复，我们使用带驻留机制的扁平化 AST 数据结构存储帧表达式序列，以最大限度地减少内存使用。

我们还为帧应用类型语义，以便快速发现错误。帧的类型由其分辨率和像素格式组成；例如，在图 3 中，〈来自 in.mp4 的第 0 帧〉的类型为 <1280x720, yuv420p>。这是 "in.mp4" 视频文件的编码格式。随后，一个函数将其转换为 <1280x720, bgr24>，以应用 OpenCV 滤镜，然后再将其转换回 yuv420p 以进行最终编码。我们发现，包括 OpenCV 在内的许多系统默认使用 rgb24 或 bgr24 格式，尽管大多数视频采用 yuv* 格式。这种转换并非总是必要的，会导致不必要的像素格式转换并增加内存使用。通过允许滤镜使用不同的内存中像素格式，我们能够更高效地使用内存并进行渲染。

4.2 通过符号计算提升

我们提出了一种带有帧和滤镜符号引用的 OpenCV API垫片，用于组合声明式规范。从概念上讲，我们不再将帧表示为 n维数组，而是引入 Frame对象，它模拟虚拟的 n维数组，并跟踪各种变换以构建帧表达式。cv2.VideoCapture 对象为视频源中的帧提供迭代器；cv2.putText 等 OpenCV 函数映射为我们数据模型中的滤镜；而 cv2.VideoWriter 对象则收集渲染后的帧，以创建规范。由于没有视频帧被解码、转换或编码，这些符号代码的执行速度比对应的非符号代码快许多个数量级，同时内存使用也仅为其一小部分。

实例化 cv2.VideoCapture 对象时，会获取源视频的分辨率、像素格式和帧数。连续调用 read() 会创建并返回帧对象，其中存储了帧的分辨率、像素格式和构造过程。构造过程是由构建该帧所需的过滤函数调用组成的函数组合。帧对象提供面向用户且兼容 NumPy 的接口。我们还针对像素格式进行了额外优化。大多数视频以 yuv420p 或类似的像素格式存储，而 OpenCV 默认会在解码时将所有帧转换为 bgr24。因此，为避免不必要的像素格式转换，帧对象对外呈现为对应的 OpenCV 格式，但在内部保留真实格式，使其在应用需要进行像素格式转换的滤镜之前，始终保持为输入视频的原生格式。

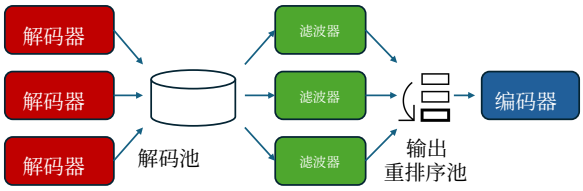

图 4：渲染引擎架构图。

每个受支持的 OpenCV 函数都以相同的函数签名实现，并会更新帧对象。OpenCV 函数与 Vidformer 兼容的 OpenCV 滤波器函数之间并非严格一一对应；许多 OpenCV/cv2 函数会原地修改帧，而 Vidformer 滤镜则是纯函数，始终返回单个帧；我们的 API 兼容层会屏蔽这些差异。少数不属于我们概念性数据模型的实用函数（例如 cv2.getTextSize()）会直接传递给实际的 cv2 库。转换后的帧会写入 cv2.VideoWriter，并转换为其声明式表示，随后附加到视频规范中。关闭此写入器后，视频规范即告完成。

*4.2.1 超越 Python/OpenCV 脚本。*我们主要关注使用 OpenCV 的 Python 脚本，因为这种组合在实践中十分普遍，但我们的方法也适用于其他前端。我们已验证，可以通过符号计算（例如使用 FFmpeg 滤镜）或转译（例如使用 DVEs）支持其他前端，包括第 2 节中提到的所有系统。许多更高级的工具（例如 Supervision [46],）都构建在 OpenCV 之上，并针对特定任务提供了更便捷的使用体验。我们只需付出极少的努力即可支持这些工具；我们的实现通过 `import vidformer.supervision as sv` 支持 Supervision，唯一需要做的就是移除一条 `assert frame is numpy.ndarray` 检查。

4.3 掩码、热力图及其他栅格数据

某些可视化类型会使用栅格数据，例如对象掩码或热力图。若将这类数据直接嵌入规范中作为常量参数，可能会带来挑战。为解决这一问题，我们将这些 n 维数据数组视为无损视频流中的帧，并用新创建的帧引用替换这些数据。在可视化对象分割检测器（例如 Segment Anything [29]）的结果时，我们会编码一个新视频，将每个对象掩码打包为单独的一帧，并采用单通道像素格式，即 gray8，然后使用轻量且无损的 FFV1 [38] 编解码器压缩该流。这样既能快速压缩并节省空间，尽管将对象掩码用作视频帧并非常规做法，但由于渲染引擎解码系统的设计（第 5.1 节），访问速度仍然很快。

5 声明式渲染结果视频

我们详细介绍了自己的声明式视频变换渲染引擎，它通过运行声明式规范来渲染视频。如图 4 所示，我们采用贪心的协同多进程方案。一组解码器线程负责解码所需帧，并将其放入解码帧池中。随后，一组滤镜线程使用解码池中的输入帧渲染输出帧。滤镜线程会将渲染后的输出帧插入输出重排序池。输出重排序池充当临时缓冲区，确保帧按正确顺序发送给编码器。编码器从输出重排序池中提取帧，并将生成的视频写入磁盘。每个输出帧都由一个帧表达式生成，该表达式是由多个过滤函数组成的依赖树。这样一来，滤镜和编码组件都相对简单；无论帧表达式有多复杂，也无论它们在连续输出帧之间变化幅度有多大，执行都可以并行化。然而，滤镜依赖于解码后的输入帧，要确保针对所有访问模式及时高效地加载帧，仍是一项重大的调度挑战。我们将此称为乱序帧访问问题。

5.1 乱序帧访问

为了达到现有方法的水平，必须支持所有 视频数据可视化/转换任务。大多数任务按顺序使用帧，但也有一些并非如此，例如每秒采样一帧、仅使用关键帧、反转视频、使用多个视频中的帧，或并排显示多帧。此外，当并行渲染大量输出帧时，即使是顺序帧访问模式也会变得不规则。再者，如第 4.3 节详述，视频中的数据同样经过压缩；例如，计算机视觉分割模型生成的对象掩码会作为独立帧存储在无损压缩视频中。渲染突出显示对象掩码的帧，需要解码所有对象掩码帧；每帧可能包含数十个对象，这些对象分散在对象掩码视频流中。此类任意帧访问模式与视频的编码和存储方式相冲突。视频编解码器会将帧分组成图像组（GOP），以利用视频的时间局部性。GOP 以关键帧/I 帧开头，随后是依赖型 P（渐进式）帧和 B（双向）帧；这些帧会引用 GOP 内先前已解码帧中的信息。较小的 GOP 能实现高效随机访问，因此传统视频编辑工作流倾向于使用全 I 帧（每一帧都是关键帧）压缩。较大的 GOP 可实现更高的压缩比，但代价是在解码期间占用更多内存，并且在流中定位新位置时需要额外计算。零星的访问模式会导致解码放大，其程度与 GOP 大小成正比，通常约为 30 – 300×。为避免这种不利的减速，我们设计渲染引擎，尽可能避免此类访问，并在无法避免时高效地并行处理。

5.2 ResultVideos 渲染引擎

我们设计了一种渲染流水线，以便在渲染过程中高效利用内存，如图 5 所示。为了在本文中更好地区分输入帧和输出帧，我们将每个输出帧索引称为生成或代。每个生成有三种状态之一：未规划、活动和完成。生成按顺序规划，并因此加入活动生成集合。调度器监视该集合，并指示解码器加载活动生成所需的所有输入帧。如果所需的所有输入帧都已存在于解码池中，则该生成处于“就绪”状态；否则处于“等待”状态。一旦某个活动生成变为“就绪”，调度器就会分配一个

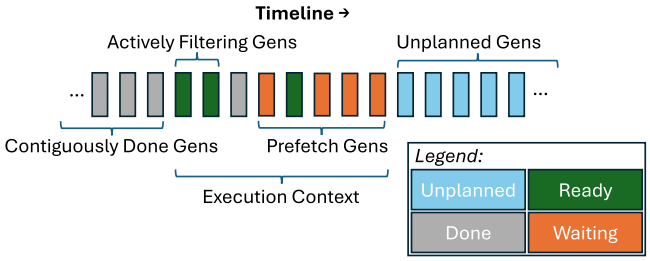

图 5：执行期间解码池的生成状态。源帧会在滤波之前预取。

过滤线程来渲染相应的输出帧，并将其插入输出重排序池。当编码器从输出重排序池中取出该输出帧后，该生成便标记为“完成”，并从活动生成集合中移除。

5.2.1 解码器。 视频片段的解码必须从 GOP 的开头开始。由于解码器会独立处理各个 GOP，我们将解码器建模为一个迭代器，用于迭代单个 GOP 中的帧。这与 Scanner [43] 用于视频批处理的技术相同。我们将每个解码器的状态表示为其目标 GOP 以及尚待解码的帧集合，并将其称为 FutureSet。它是一个集合，而非位置，因为我们预期解码器会以任意顺序输出帧。例如，一个包含三帧 1，2，3、类型为 I， B， P 的 GOP，会存储为 I， P， B，并按 1，3，2 的顺序解码。现有基于迭代器/流的系统 [16, 44] 会使用重排序缓冲区对帧重新排序，以提供连续的流。不同的是，我们会立即将帧放入全局解码池。与逐解码器重排序相比，这种方式的内存效率更高，因为它会从全局而非单个视频流的角度优化内存使用，并使后续生成能够比原本更早开始滤波。

5.2.2 调度与池管理。我们的声明式规范为每个输出帧提供帧表达式，并提取每个输出帧所需的输入帧集合。我们将这些所需帧集合的列表称为调度表。接下来，我们将 NeedSet 定义为活动生成所需的所有帧：

$$\text{NeedSet} = \bigcup_{g \in \text{ActiveGens}} \text{schedule}[g]$$

调度器规划各代生成，将其加入活动生成，直到 NeedSet 超出解码池容量，或活动生成的数量达到预取窗口大小。随后，调度器将 GOP 分配给解码器，以确保 NeedSet 中的所有项都存在于解码池中。为此，调度器会查找 NeedSet 中所有既不在解码池中、也不在任何现有解码器的未来集中的帧。在这些帧中，它选择尚未“完成”的、编号最低的生成所需的帧，并将包含该帧的 GOP 分配给解码器。调度器重复此过程，直到所有解码器线程都已分配。

每个解码器线程都会监视 NeedSet；只要其未来集中的某一帧也在 NeedSet 中，但尚未进入解码池，就会继续处理。如果某帧是尚未完成的任何生成所需的，就会将解码后的帧放入解码池。如果未来集中没有任何帧位于 NeedSet 中但不在解码池中，解码器就会停滞，直到这一条件发生变化。解码器线程完成对所分配 GOP 的解码后，会等待调度器分配新的任务。此外，如果满足以下所有条件，我们允许解码器提前放弃所分配的 GOP：

- 解码器停滞。
- 有更多关键帧需要解码：NeedSet 中存在不在解码池中的帧，而这些帧比解码器的 FutureSet 中所有帧都更早被某个生成所需要。
- 该解码器是需求最低的解码器：当前解码器中最早需要的未来帧，其需求时间晚于（或与）所有其他解码器中最早需要的未来帧。

该解码器 GOP 放弃策略使所有解码器线程保持忙碌，并防止系统发生死锁。

解码器线程会尝试将每个已解码帧插入帧解码池，该池的大小是固定的。由于 NeedSet 的大小永远不会超过池大小，因此它相当于池中预留的一部分，其余部分则用作缓存。如果该帧位于 NeedSet 中，则会强制插入，并在必要时触发驱逐。如果该帧不在 NeedSet 中，则会根据驱逐策略尝试插入。我们采用最优缓存驱逐策略，始终驱逐最不早被未完成生成所需的帧：

$$\text{下一个所需生成}(f) = \begin{cases} \min\{g \in \text{未完成的生成} \mid f \in \text{调度表}[g]\} & \text{如果 } \exists g,\ f \in \text{调度表}[g] \\ \infty & \text{否则} \end{cases}$$

我们使用解码帧池来掩盖时间局部性上的间隙，从而减轻乱序访问时解码所带来的负担。为了确保 Vidformer 能够加速即使是最复杂的视频变换，我们必须能够将帧池用作有效的缓存，而不受帧访问模式的影响。过滤线程会在执行上下文中处理较早的帧，而较晚的帧则会提前获取。如果调度器将计划安排得过于超前，解码池中就没有余量可用于缓存，从而降低效率。相反，如果预取不足，过滤线程就会在等待源帧到达时阻塞。我们通过限制活动生成的数量来控制预取窗口的大小，这会对解码器施加背压，使其停滞。概念上，这会在解码器的积极与惰性行为之间切换，以平衡效率和吞吐量。我们将在评估部分讨论如何配置这一限制。

5.2.3 实现说明。在前面的章节中，我们提出了一种多线程渲染引擎。为简洁起见，我们省略了对具体同步原语的讨论。完整细节可在我们的源代码中查阅。任何并发系统的实现都存在确保正确性的固有难题；我们使用 TLA+ [32] 检查了核心并发模型，并在随附的源代码中提供了 TLA+ 规范。

6 按需获取视频结果

我们的声明式视频变换渲染引擎通过扩展控制流来应用

优化并实现并行渲染。正如结果部分所述，这使多种标注工作负载的速度提升了 2 – 3× 。然而，这种方法仍受其组件局限性的制约：视频帧的解码、滤波和编码。若要实现真正交互式的数据探索体验，仅靠渲染优化不足以提供用户认为具有交互性的亚秒级响应时间 [33]。

因此，为了提供这样的体验，我们围绕部分物化来设计 Vidformer，避免渲染整个视频。虽然观看 生成的视频是最终目标，但传统上，对大多数实现而言，将整个视频写入磁盘这一缓慢过程通常是最省事的做法。我们不采用这种方式，而是在输出帧即将显示时才进行渲染；通过将渲染后的视频作为视频点播 （VOD）流提供，这种方式与现有视频播放器 兼容。Vidformer 与 VOD 相结合是一种务实的方案：由于视频点播标准广受媒体播放器支持，点播流媒体服务普遍采用这些标准；而且 VOD 协议在网络上的传输效率很高，这对于向边缘设备上的客户端交付视频结果至关重要。我们展示了 VOD 如何实现视频结果的近乎即时播放。

我们的实现采用 HTTP 实时流（HLS） [40]，但对于此用例而言，DASH [24] 在功能上与之等效。我们将帧划分为 2 秒的分段，并立即创建引用这些分段的流清单文件。当客户端请求某个分段时，Vidformer 会查找所需的帧，并将其声明式表示传递给我们的渲染引擎，由该引擎渲染并返回此分段。图 1 底部展示了这一过程。这样一来，渲染就成为一个常数时间操作；只需渲染一个短分段即可开始播放。用户可以在播放器中观看完整视频，并像播放已完整渲染的视频一样跳转到任意位置。

6.1 通过直播流媒体增量获取视频结果

在我们初步集成点播视频结果的工作中，我们发现渲染速度非常快，以至于可视化脚本的执行成了瓶颈。我们运行不含多媒体操作的脚本，其速度比实时快好几个数量级。然而，较长的视频可能导致脚本耗时数秒，这是 Python 解释器相对缓慢以及可视化脚本效率低下的结果（例如，图 2 中的脚本会针对每个输出帧对数据框 df 进行一次筛选）。这部分开销通常会分摊到整个渲染流水线中，因此可以忽略不计；但它却成了从发起请求到开始播放的延迟的主要来源。

为了持续将开始播放时间控制在一秒以内，我们会在可视化脚本执行时逐步显示视频。在我们的 Python/OpenCV 实现中，帧在写入 cv2.VideoWriter 时会被追加到规范中。由于输出视频的长度未知，我们使用视频点播直播流媒体来向视频播放器传达这一点。我们的视频点播服务器会将清单标记为未完成，并随着更多帧可用而逐步添加分段。cv2.VideoWriter 关闭后，清单会报告流已终止，从而使播放器切换到常规点播播放。这既不是一种视频点播流（其中整个视频都已可用），也不是一种直播流（会持续增长，但具有固定的回看窗口），而是二者的混合模式。视频流具有固定的起点，可以持续追加内容，并且所有分段始终可用；HLS 将其称为“事件流”。这种增量式直播流媒体方法使播放成为一种常数时间操作，即便中间查询和可视化脚本执行缓慢也是如此。

6.2 SourceVideo 输入/输出

此前，我们展示过如何使用视频的本地存储路径直接打开视频；然而，这种方式在分布式部署中行不通。取而代之的是，我们在视频所在位置直接访问视频，通常是通过底层存储服务（对象存储）进行访问。在我们的实现中，所有视频输入/输出都通过 Apache Open-DAL [14] 进行，以支持多种存储协议。与视频数据库集成时，脚本可以使用简单路径（例如，“<video primary key UUID>.mp4”），Vidformer 会将其映射到底层视频文件的位置。通过网络访问视频文件会带来额外的延迟问题，即便是在高速数据中心网络中也是如此。由于获取后续点播片段时会重复进行许多 I/O 访问，因此我们额外采用了共享 I/O 块缓存层，以消除反复打开和解析视频文件所产生的延迟开销。

6.3 基于大语言模型的视频查询

我们详细介绍 Vidformer 如何实现交互式的基于大语言模型的视频查询。这是 Vidformer 带来的一种令人振奋的新应用场景，也为我们详细介绍一套可用于生产环境的部署方案提供了契机。在流行文化中，能够发出命令并查看视觉媒体结果，是未来科技的一大经典特征。许多电视剧和电影（例如《星际迷航》和《头号玩家》）都着重描绘了角色通过语音发出命令，并即时收到视频或全息影像响应的场景。现代大语言模型能够将自然语言命令转换为代码，从而让这一切成为现实；然而，由于渲染生成的视频需要较高延迟，这种应用仍不切实际。Vidformer 与大语言模型相辅相成，采用图 6 所示的方法，实现交互速度的视频查询。

我们将 Vidformer 部署为一项 Web 服务，提供两个端点：一个用于其管理 API，另一个用于处理视频点播请求。当用户发出“给我展示……”这样的命令时，大语言模型智能体会生成数据库查询和 Python/cv2 可视化脚本，并向 Vidformer 服务注册一项新规范。Vidformer 服务会立即生成一个点播命名空间，并向用户返回点播流清单的链接。随后，浏览器会显示一个视频播放器，而点播客户端则反复轮询清单文件以获取分段。与此同时，大语言模型智能体会调用一个轻量级虚拟机，在安全环境中执行视频数据可视化脚本。脚本首先会经过增强，将 import cv2 替换为 import vidformer.cv2 as cv2，以应用 cv2 API 层。所有数据库查询结果都会复制到虚拟机中，并向虚拟机提供一个帧推送 API 端点；脚本将帧表达式写入 cv2.VideoWriter 对象时，Vidformer 库会逐步将这些表达式发送到该端点。Vidformer 服务接收来自虚拟机的这些推送请求，然后

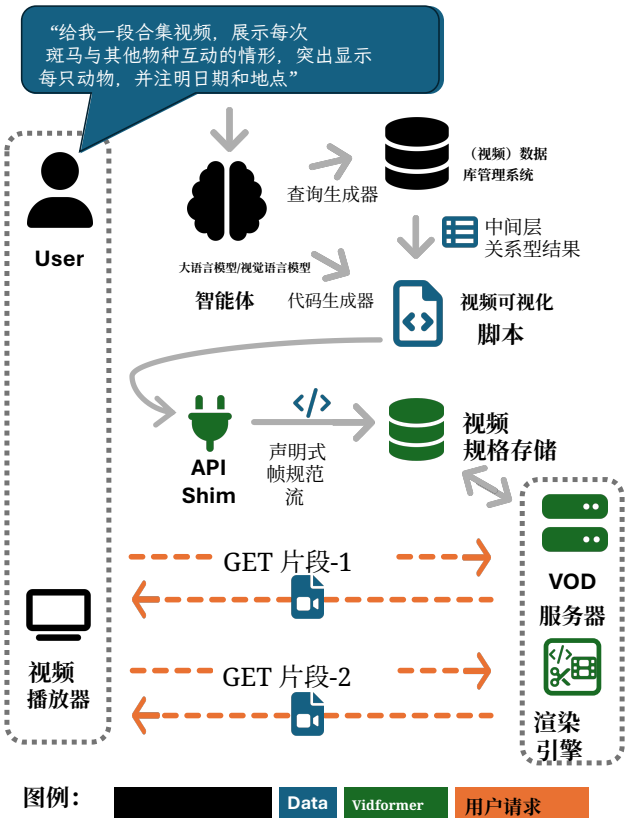

图 6：基于大语言模型的视频查询：当用户发出“给我展示……”查询时，大语言模型会查询数据库并编写视频数据可视化脚本。Vidformer 运行此脚本，从而在几秒内实现端到端的视频响应。

将它们插入视频规格存储库。推送端点还会执行类型检查并应用安全策略。类型检查可确保每个过滤函数都能接受所提供的参数，并确保帧的输出分辨率和像素格式正确。这样可以确保数据库中的视频规格能够正确渲染，并帮助可视化脚本快速发现错误。此外，该端点还会通过静态检查帧是否符合安全策略，防止出现不良规范；安全策略包括限制中间帧分辨率、内联数据值大小和规范树深度。每当点播客户端轮询清单时，Vidformer 服务都会统计已成功写入的帧数，以确定生成的清单中应包含多少个分段。例如，对于帧率为 30 FPS、每段时长为 2 秒的视频，脚本写入第 60 帧后，清单就会列出第一段。点播客户端随即请求该分段，并按需进行渲染。点播端点上的负载均衡器可并行渲染分段，将请求分发到渲染主机服务器或容器集群。负载均衡器还会缓存已渲染分段，因为点播客户端经常会清除并重新请求相同的分段，也可以借此让多个客户端使用同一数据流。可视化脚本执行完毕且 cv2.VideoWriter 关闭后，Vidformer API 适配层会最后调用一次 push，将视频规范标记为已终止，随后服务器会向清单添加流结束标记。这会使视频播放器显示视频的结束时间，并停止轮询清单。点播播放器可能会继续获取分段，直到会话结束；此时，清理例程会从规格存储库中移除视频规范。

6.4 局限性

6.4.1 范围。 本文提出的方法仅适用于用于可视化的视频转换，不会在可视化脚本中观察像素值或根据像素值进行分支。注释数据必须在外部计算。Vidformer 并非通用视频处理加速器；此类功能依赖外部系统或用户提供。

6.4.2 分段编码。 我们的方法固有的一项低效之处，是点播分段编码中增加了编码/解码开销。将渲染引擎生成的未压缩帧直接传入进程的帧缓冲区即可消除这项开销；然而，未压缩帧体积过大，无法通过网络传输，因此除非直接在客户端进行渲染，否则这种做法并不可行。实际上，客户端通常是网页浏览器，因此这种做法是可行的；但目前来看，它并非理想方案。边缘设备的性能通常远不及服务器，而较慢的网络会导致难以从外部视频源进行输入/输出。最重要的是，边缘设备操作系统和网页浏览器将快速、高效地播放点播流视为首要任务 [8]。偏离这一高度优化的流水线，会牺牲能效和可实现的最高分辨率，除非投入大量开发工作。

7 评估

我们使用 Rust 实现了 Vidformer 服务器和渲染引擎，利用 FFmpeg 的 libav* 库进行底层视频管理，并使用 OpenCV 实现大多数滤镜。我们使用 Python 和 OpenCV 的 cv2 库手写了基线。**基线和 Vidformer 都使用 FFmpeg 的 libav* 库进行解码/编码，并使用 OpenCV 实现滤镜，因此这是一次公平的比较。**

数据集。我们使用 Blender Open Movie 的《*Tears of Steel*》（ToS），这是一段常用于需要单个视频的任务的参考视频。它为 1280 × 720，帧率为 24 FPS，时长为 734 秒，编码后的文件在磁盘上占 56 MB。评估的大部分内容使用低分辨率视频，因为较高的分辨率可能会掩盖渲染引擎调度器的开销。我们在第 7.1.3 节使用 4K 原始视频的降采样编码版本。对于需要视频集合的任务，我们使用由美国公共广播电视公司《NOVA》和《FRONTLINE》系列的 100 部纪录片汇编而成的视频，这些纪录片均可在 YouTube 上公开观看。我们选择的来源为1920 × 1080，帧率为 29.97 FPS，每部时长约 55 分钟，总计 70 GB。所有视频均采用 H.264 编码，像素格式为 yuv420p。

环境。 我们使用了一台搭载 2×12 核（48 个硬件线程 vCPU）Intel Xeon Gold 6126 处理器、主频为 2.6 GHz、配备 384GB DDR4 内存的系统，运行 Debian 12。为准确衡量线程的影响，我们避免让 CPU 达到饱和；由于底层 FFmpeg libav* 库及其编解码器会在内部创建线程，因此我们将所有实验限制为总计 16 个解码和滤波线程。我们使用了 FFmpeg 7.0、OpenCV 4.9，

注释	基线	VF	VF+视频点播
标签	116.09	41.32	0.28
框+标签	123.20	44.21	0.25
框角+标签	120.96	63.38	0.24
颜色+标签	141.15	47.39	0.30
遮罩+标签	144.00	51.34	0.50

表 1：以秒为单位的开始播放延迟。比较 Python 和 OpenCV 基线、Vidformer完整渲染以及 Vid-former+视频点播。

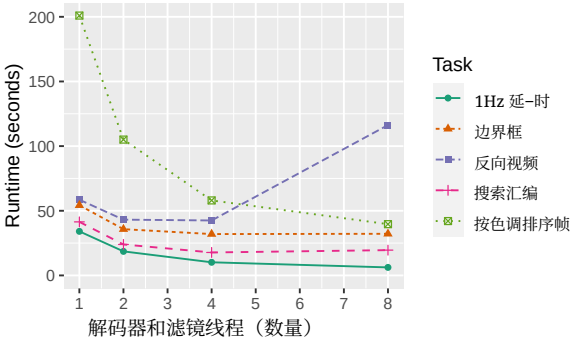

图 7：线程数与渲染运行时间。

Rust 1.77 和 Python 3.11。我们使用 Postgres 作为规范存储数据库，并使用 Valkey（一种开源 Redis 分支）作为源视频输入/输出缓存。

7.1 结果

我们首先评估简单的计算机视觉标注任务。我们使用 Supervision 的 [46] 标注器，它们在内部使用 OpenCV。“标签”任务对应于 supervision.LabelAnnotator。我们使用 YOLOv 8 [45] 推理的预计算检测结果，为 ToS 视频添加标注；掩码标注器使用 YOLOv8-Seg。我们使用了八个解码/过滤线程，解码池大小为 100。结果如表 1 所示。对于 Vidformer，基线和 VF 分别表示运行完整可视化脚本和完整渲染生成视频所需的时间。使用 VF+ 视频点播时，这表示开始播放视频所需的时间，此时只会渲染一个两秒片段。这些结果表明，Vidformer 渲染引擎可使现有工作流提速 2× 至 3× 倍。通过视频点播观看视频，可在半秒内开始播放，提速 ~400–500× 倍。

7.1.1 渲染并行性。为了评估 Vidformer 渲染引擎如何利用额外线程降低延迟，我们通过多种任务，从不同方面对渲染引擎进行压力测试。我们的所有任务都基于 ToS，除“搜索汇编视频”外，其余任务的含义都可从名称看出。“搜索汇编视频”会在 PBS 数据集的字幕中搜索某个口语词（我们使用“river”一词），制作汇编视频，并标注该词出现的次数、源视频以及源片段的时间戳。最终得到时长 1 分 42 秒的结果，其中包含来自 14 个不同源视频的 32 个片段。

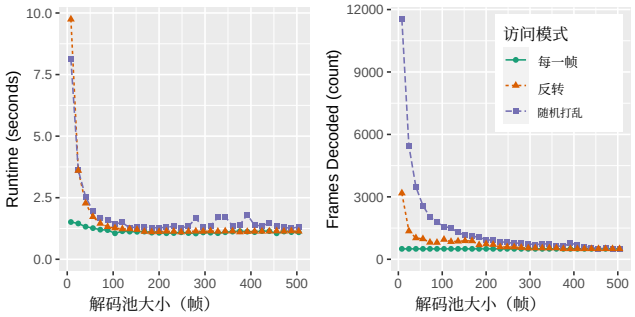

图 8：密集帧访问模式下，运行时间（左）和解码帧数（右）随解码池大小的变化。图中显示了解码 500 帧所需的时间。

我们将解码池大小设为 100 帧，预取窗口设为 80。在此配置下，解码池的内存使用量在 720p 时最高为 277 兆字节，在 1080p 时最高为 622 兆字节。图 7 中的结果表明，Vidformer 能够高效地在线程间分配工作负载，从而缩短运行时间。这一趋势有一个例外：“反向视频”任务在使用八个线程时的表现明显不如使用较少线程时。我们已确认，对于此任务，线程数更高时这一趋势还会进一步恶化。我们认为，这是解码器线程池中出现了突发的抖动行为所致。帧通常会按照接近其时间位置的顺序进行解码。因此，当需要 GOP 末尾的帧时，较早的帧会被丢弃，导致启动额外的解码器。当解码顺序类似于反向访问顺序时，就会出现此问题；反转视频时问题尤为严重。增大解码帧池，使其足以缓存一个 GOP，即可缓解这一问题。

7.1.2 帧访问模式。 我们评估了不同的帧访问模式，以考察我们的调度和帧缓存方法。我们通过访问每一帧、反转视频以及随机打乱帧的顺序来评估密集访问模式，从而涵盖高、中、低局部性访问模式。我们测量了在 ToS 的一个 500 帧片段中，不同解码池大小下所需的时间和解码帧数。结果如图 8 所示。正如预期，增大解码池大小会使我们每项任务的运行时间和解码帧数迅速下降。

稀疏访问帧很困难，因为这会不可避免地导致帧解码放大。在最坏情况下，解码器必须解码整个 GOP 才能取出一帧。在这种情况下，加快转换速度的唯一办法是增加解码器线程。我们在均匀的帧访问间隔下评估了这一行为，步长范围为 1 到 8192，总计访问 1000 帧。由于没有任何单个视频足够长，我们将 PBS 数据集中的所有视频拼接起来，构建了一个虚拟源视频。由此得到的虚拟视频包含 970 万帧。结果如图 9 所示，主要分为两类：小步长（1–8 帧）无论解码线程数量多少，处理速度都很快；而大步长（≥ 128）处理较慢，但执行速度与线程数量成正比。解码器数量从 1 个增加到 16 个，带来了超过 10× 的加速。

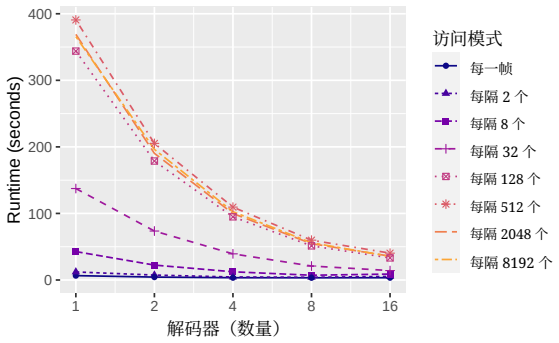

图 9：稀疏帧访问模式下，解码器线程数与运行时间的关系。图中显示提取 1000 帧所需的时间。

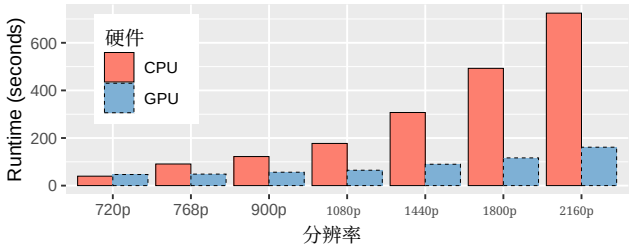

图 10：不同分辨率下 CPU 和 GPU 的渲染速度。

步长较小时，单流处理占主导地位，解码池已饱和，只能等待活动 GOP 中的帧。步长较大时，每个输出帧所需解码的 GOP 数会迅速接近最坏情况，即一个 GOP；不过，较高的 GOP/帧比意味着任意时刻都能在 NeedSet 中表示更多 GOP，从而有效利用额外的解码器线程。

7.1.3 GPU 加速。 图 10 展示了 Vidformer 的 GPU 加速性能。我们使用先前的 LabelAnnotator 任务，在常见的 16:9 分辨率下测试了 ToS 视频。我们使用搭载 H.264 NVDEC 和 NVENC、并采用“fast”预设的 NVIDIA Titan Xp。CPU 使用 FFmpeg 默认的编解码器 libx264，并采用“ultrafast”预设。我们将滤镜线程数增至 24，以缓解由基于 CPU 的滤波器造成的瓶颈。所有实验中，我们都在 CPU 上运行滤波器，因为 OpenCV 绘图方法仅支持 CPU。通过替换为 GPU 加速的同类方法（例如 CV-CUDA [15]），即可实现完整的端到端 GPU 加速。为保证公平，我们未将其纳入评估，因为它无法与基线实现像素级完全匹配。结果表明，GPU 能够在更高分辨率下始终实现快速渲染，在 4K（2160p）下最高可快 4.49× 倍。

7.1.4 大语言模型生成的脚本。 我们要求大语言模型编写按色相对帧进行排序的代码。这个任务既简单，又具有明确的优化潜力，涵盖 OpenAI GPT-5（ChatGPT）、Anthropic Opus 4.1（Claude）和 GoogleGemini 2.5 Pro（Gemini）。我们分别在没有任何指导的情况下直接提出请求（Simple）、要求合理的内存使用量（LM）、要求在计算与内存之间取得平衡的高度优化方案（Smart），以及在提供了

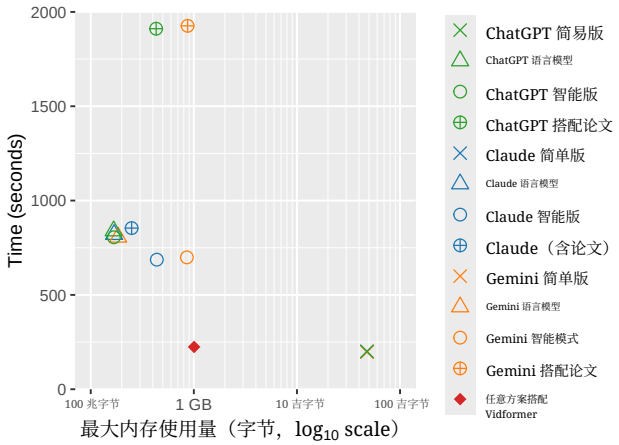

图 11：大语言模型生成的可视化脚本的内存使用量和运行时间。图中展示了不同模型和提示指导方式，以及使用 Vidformer 运行任意这些脚本时的性能。

第 5 节文本的情况下（附论文）。我们加入了 Vidformer 加速的替代方案；无论使用上述哪种脚本，该方案都具有相同的内存和计算特征。为保证公平，我们将 Vidformer 限制为单线程运行。需要注意的是，超过 700 兆字节的内存使用量属于 Vidformer 服务器开销，而非实际渲染引擎的使用量。图 11 中的结果表明，无论使用哪种模型或提示词，大语言模型生成的脚本都会在时间、内存或两者方面造成浪费。这说明，在规模更大、时长更长的视频数据集上，大语言模型生成的视频数据可视化脚本不仅速度过慢，还常常会因内存使用量过高而失败。应用 Vidformer 可通过提供一致的资源使用情况来解决这些问题，使大语言模型能够专注于以最直接的方式生成正确的脚本。

8 结论

我们提出了 Vidformer，这是一个能够实现视频原生查询结果即时播放的系统。Vidformer 将视频数据可视化脚本建模为对视频数据的转换。我们使用 API 垫片，仅需进行一行修改，即可将命令式脚本提升为声明式帧规范流。我们设计了一种渲染引擎，通过利用声明式规范来优化并行化渲染。此外，Vidformer 通过 VOD 协议提供视频结果，并按请求渲染短片段。

我们的评估表明，使用我们的渲染引擎可实现 2–3× 的加速，与 VOD 流媒体结合时可实现 400× 的加速。此外，我们发现，在各种任务中，我们的渲染引擎能够有效利用额外的线程和内存来降低延迟。即使在帧访问极为稀疏、只能受益于额外线程的最坏情况下，我们仍能观察到线性加速。我们还评估了 GPU 加速渲染，发现 GPU 能够以极小的额外延迟支持更高的分辨率。

我们展示了我们的方法不仅能够加速现有的视频数据可视化流水线和工作流，还能实现全新的应用。基于大语言模型的交互式自然语言查询中，“*show me ...*”命令可以即时生成视频结果；这不仅得益于 Vidformer 才成为可能，而且变得非常简单。通过将视频编辑器实时预览窗口的速度带入可视化脚本，Vidformer 实现了视频原生查询结果真正即时的播放。我们将 Vidformer 作为一个实用、易用的开源系统发布[1]。

致谢

本材料基于美国国家科学基金会第 1910356 号拨款及美国国家科学基金会计算与通信基金会 2118240 号影像组学研究所奖项资助的工作。图 1 和图 6 中的图标由 Font Awesome 提供（知识共享署名 4.0 国际）。

参考文献

[1] *ApertureDB*. 2024. https://github.com/aperture-data/aperturedb-python [2] 2024. Creatomate. https://creatomate.com/ [3] 2024. redotvideo/revideo. https://github.com/redotvideo/revideo [4] 2024. Remotion. https://www.remotion.dev/ [5] 2024. Shotstack. https://shotstack.io/ [6] 2024. VideoDB. https://videodb.io/ [7] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, San- jay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow：异构系统上的大规模机器学习。https://www.tensorflow.org/ 可从 tensorflow.org 获取软件。[8] Maha Abdallah, Carsten Griwodz, Kuan-Ta Chen, Gwendal Simon, Pin-Chun Wang, and Cheng-Hsin Hsu. 2018年。云端延迟敏感型视频计算：综述。*ACM* 多媒体计算、通信与应用汇刊14, 3s, 文章 54 2018 29 10 1145 3212804（6 月），页码。https://doi.org/ 。/ [9] Leif Andersen, Stephen Chang, and Matthias Felleisen. 2017. 用于制作电影的 Super 8 语言（函数式编程珍珠）。*ACM* 编程语言论文集 1, 2017 30 1 30 29 10 1145 3110274 ICFP（8 月），：–：。 [10] 2002 https://doi.org/ 。/ Umut Arslan. 。面向视频数据库的语义数据模型与查询语言。博士论文。ISBN：9798426865716 出版物标题：PQDT - Global。[11] Stefano Bocconi。2004。语义感知自动视频编辑。载于第 *12* 届 *ACM* 国际多媒体年度会议论文集。ACM，纽约州纽约市，971 972 10 1145 1027527 1027753 –。https://doi.org/ 。/。G. [12] 2000 Bradski. 。OpenCV 库。*Dr. Dobb's* 软件工具杂志（2000）。[13] T Catarci、ME Donderler、E Saykol、O Ulusoy 和 U Gudukbay。2003。BilVideo：视频数据库管理系统。*IEEE* 多媒体10, 1（2003），66–70。https://doi.org/10.1109/MMUL.2003.1167924 出版方：IEEE。[14] OpenDAL 贡献者。 [无日期]。*OpenDAL*。https://github.com/apache/opendal [15] 英伟达公司。2024年。CV-CUDA。https://developer.nvidia.com/cv-cuda。访问日期：2024-11-18。[16] FFmpeg 开发者。2024年。ffmpeg 工具。http://ffmpeg.org/ [17] Mikael Finstad。2024年。mifi/editly。https://github.com/mifi/editly 原始日期：2020-04-15。[18] Adam Fouse、Nadir Weibel、Edwin Hutchins 和 James D. Hollan。2011。ChronoViz：支持浏览时间编码数据的系统。载于*CHI '11* 人机交互系统中的因素会议扩展摘要（*CHI EA '11*）。计算机协会，纽约州纽约市，299–304。https: //doi.org/ 10 1145 1979742 1979706 。/ 。 [19] Nathan Frey、Peggy Chi、Weilong Yang 和 Irfan Essa。2021。自动非线性视频编辑迁移。https://doi.org/10.48550/arXiv.2105.06988 arXiv:2105.06988 [cs]。[20] Boni Garcia、Luis López、Francisco Gortázar、Micael Gallego 和 Giuseppe Antonio Carella。2017。NUBOMEDIA：首个开源 WebRTC 平台即服务。载于第 *25* 届 *ACM* 国际多媒体会议论文集（*MM '17*）。计算机协会，纽约州纽约市，1205–1208。https://doi.org/10.1145/3123266.3129392

[21] Peter J. Haas 和 Joseph M. Hellerstein。1999年。用于在线聚合的涟漪连接。载于《1999年ACM SIGMOD数据管理国际会议论文集》（美国宾夕法尼亚州费城）（SIGMOD '99）。计算机协会，纽约州纽约市，287–298。https: //doi.org/ 10 1145 304182 304208 . / . Joseph M. Hellerstein、Peter J. Haas 和 Helen J. [22] Wang。1997年。在线聚合。SIGMOD Rec. 26, 2 1997 171 182 10 1145 253262 253291 （6月），–。 [23] 1996 https://doi.org/ . / . Eenjun Hwang 和 V. S. Subrahmanian。. 查询视频库*。《视觉通信与图像表示期刊》7, 1（1996年3月），44–60。https: //doi.org/ 10 1006 1996 0005 . /jvci. . ISO/IEC - : . 信息技术——HTTP动态自适应流（ [24] 23009 1 2022 2022 DASH）。标准。国际标准化组织。[25] Chris Jermaine、Subramanian Arumugam、Abhijit Pol 和 Alin Dobra。2008年。利用DBO引擎实现可扩展的近似查询处理。《ACM数据库系统汇刊》33, 4，文章23（2008年12月），54页。https://doi.org/10.1145/1412331.1412335[26] Gaurav Tarlok Kakkar、Jiashen Cao、Pramod Chunduri、Zhuangdi Xu、Suryatej Reddy Vyalla、Prashanth Dintyala、Anirudh Prabakaran、Jaeho Bang、Aubhro Sengupta、Kaushik Ravichandran、Ishwarya Sivakumar、Aryan Rajoria、Ashmita Raju、Tushar Aggarwal、Abdullah Shah、Sanjana Garg、Shashank Suman、Myna Prasanna Kalluraya、Subrata Mitra、Ali Payani、Yao Lu、Umakishore Ramachandran 和 Joy Arulraj。2023年。EVA：端到端探索式视频分析系统。载于第七届端到端机器学习数据管理研讨会论文集。ACM，美国华盛顿州西雅图，1–5。https: //doi.org/ 10 1145 3595360 3595858 . / . Daniel Kang、Peter Bailis 和 [27] Matei Zaharia。2019。基于深度神经网络的视频分析中的挑战与机遇：BlazeIt视频查询引擎演示。载于CIDR。[28] Daniel Kang、Francisco Romero、Peter Bailis、Christos Kozyrakis 和 Matei Zaharia。2022年。VIVA：交互式视频分析端到端系统。CIDR。[29] Alexander Kirillov、Eric Mintun、Nikhila Ravi、Hanzi Mao、Chloe Rolland、Laura Gustafson、Tete Xiao、Spencer Whitehead、Alexander C. Berg、Wan-Yen Lo、Piotr Dollar 和 Ross Girshick。2023年。Segment Anything。载于IEEE/CVF国际计算机视觉会议（ICCV）论文集。4015–4026。[30] Chanwut Kittivorawong、Yongming Ge、Yousef Helal 和 Alvin Cheung。2023年。Spatialyze：具备空间感知优化的地理空间视频分析系统。http://arxiv.org/abs/2308.03276 arXiv:2308.03276 [cs]。[31] Clemens N. Klokmose、Christian Remy、Janus Bager Kristensen、Rolf Bagge、Michel Beaudouin-Lafon 和 Wendy Mackay。2019年。Videostrates：协作式、分布式和可编程的视频操控。载于第32届ACM用户界面软件与技术年度研讨会论文集（UIST '19）。计算机协会，纽约州纽约市，233–247。https: //doi.org/ 10 1145 3332165 3347912 . / . Leslie Lamport。. 使用 [32] 1999 TLA+规范并发系统。《计算系统设计》（1999年4月），183–247。 h t t p s:// www.microsoft.com/en-us/research/publication/specifying-concurrent-systems-tla/ [33] Zhicheng Liu 和 Jeffrey Heer。2014年。交互延迟对探索式可视化分析的影响。《IEEE可视化与计算机图形学汇刊》20, 12（2014），2122–2131。[34] W. E. Mackay。1989年。EVA：用于视频数据符号分析的实验性视频标注工具。ACM SIGCHI公报21, 2（1989年10月），68–71。https://doi.org/10. 1145/70609.70617[35] Wendy E. Mackay 和 Michel Beaudouin-Lafon。1998年。DIVA：借助多媒体流进行探索性数据分析。载于SIGCHI计算系统中的人为因素会议论文集（CHI '98）。ACM Press/Addison-Wesley Publishing Co.，美国， 416 423 10 1145 274644 274701 –。https://doi.org/ . / . [36] 1989 Wendy E. Mackay 和 Glorianna Davenport。. 交互式多媒体应用中的虚拟视频编辑。Commun. ACM 32, 7（1989年7月），802–810。https://doi.org/ 10 1145 65445 65447 . / . James Matthews、Peter Gloor 和 Fillia [37] Makedon。1993年。VideoScheme：用于自动化和媒体识别的可编程视频编辑系统。载于第一届ACM国际多媒体会议论文集（MULTIMEDIA '93）。计算机协会，纽约州纽约市，419–426。https://doi.org/ 10 1145 166266 168442 . / . Michael Niedermayer、Dave Rice 和 Jérôme Martinez。2021年。FFV1视频编码格式版本 [38] 0, 1, 3 9043 10 17487 9043 和。RFC。 [39] https://doi.org/ . /RFC Ryosuke Okuta、Yuya Unno、Daisuke Nishino、Shohei Hido 和 Crissman Loomis。2017年。CuPy：兼容NumPy的NVIDIA GPU计算库。载于神经信息处理系统第三十一届年会（NIPS）中的机器学习系统研讨会（LearningSys）论文集。http: //learningsys.org/nips17/assets/papers/paper_16.pdf [40] Roger Pantos 和 William May。2017年。HTTP 实时流。RFC 8216。https: //doi.org/10.17487/RFC8216[41] Adam Paszke、Sam Gross、Francisco Massa、Adam Lerer、James Bradbury、Gregory Chanan、Trevor Killeen、Zeming Lin、Natalia Gimelshein、Luca Antiga、Alban Desmaison、Andreas Köpf、Edward Yang、Zach DeVito、Martin Raison、Alykhan Tejani、Sasank Chilamkurthy、Benoit Steiner、Lu Fang、Junjie Bai 和 Soumith Chintala。2019。PyTorch：一种命令式风格的高性能深度学习框架

[1]https://github.com/ixlab/vidformer

库。arXiv:1912.01703 [cs.LG] https://arxiv.org/abs/1912.01703[42] Devin Petersohn、Stephen Macke、Doris Xin、William Ma、Doris Lee、Xiangxi Mo、Joseph E. Gonzalez、Joseph M. Hellerstein、Anthony D. Joseph 和 Aditya Parameswaran。2020年。迈向可扩展的数据框系统。《*VLDB*学报》13, 12（2020年8月），2033–2046。https://doi.org/10.14778/3407790. 3407807[43] Alex Poms、Will Crichton、Pat Hanrahan 和 Kayvon Fatahalian。2018年。Scanner：大规模高效视频分析。《*ACM*图形学汇刊》37, 4（2018年8月），1 13 10 1145 3197517 3201394 –。https://doi.org/ . / . GStreamer项目。[44] . GStreamer：开源多媒体框架。https://gstreamer.freedesktop.org/。[45] Joseph Redmon、Santosh Divvala、Ross Girshick 和 Ali Farhadi。2016年。You Only Look Once：统一的实时目标检测。载于*2016*年*IEEE*计算机视觉与模式识别会议（*CVPR*）。779–788。https://doi.org/10. 1109/CVPR.2016.91[46] Roboflow。 [无日期]。*Supervision*。https://github.com/roboflow/supervision [47] Manuel Serrano。2007年。使用Hop编写网络多媒体应用。载于第*15*届*ACM*国际多媒体会议论文集（*MM '07*）。计算机协会，纽约州纽约市，1001–1004。https://doi.org/10 1145 1291233 1291450 . / . Dominik Winecki 和 Arnab Nandi。[48] 2024年。V2V：为视频查询高效合成视频结果。载于*2024*年*IEEE*第*40*届数据工程国际会议（*ICDE*）。5614–5621。https://doi.org/10.1109/ICDE60146.2024.00449 [49] Herve Yviquel、Antoine Lorence、Khaled Jerbi、Gildas Cocherel、Alexandre Sanchez 和 Mickael Raulet。2013年。Orcc：让多媒体开发变得轻松。载于第*21*届*ACM*国际多媒体会议（*MM '13*）论文集。计算机协会，纽约，纽约州，美国，863–866。https://doi.org/10 1145 2502081 2502231 。/。Matei Zaharia、Reynold S. Xin、Patrick [50] Wendell、TathagataDas、Michael Armbrust、Ankur Dave、Xiangrui Meng、Josh Rosen、Shivaram Venkataraman、Michael J. Franklin、Ali Ghodsi、Joseph Gonzalez、Scott Shenker 和 Ion Stoica。2016年。Apache Spark：用于大数据处理的统一引擎。《*ACM*通讯》59卷，第11期，2016年，56–65，10 1145 2934664（10月），–。https://doi.org/。/[51] Enhao Zhang、Maureen Daum、Dong He、Magdalena Balazinska、Brandon Haynes 和 Ranjay Krishna。2023年。EQUI-VOCAL：基于有限用户交互合成组合式视频事件查询。 [技术报告]。*arXiv*预印本 *arXiv*: *2301 00929* 2023 10 48550 2301 00929 。（）。https://doi.org/。/arXiv.。Pengju Zhang、Chunmiao Yuan、Kunliang Liu、Yukuan Sun、Jiayu Liang、Guang-hao Jin 和 Jianming Wang。2019。基于自然语言查询的快速视频片段检索方法。载于《高级数据挖掘与应用》，Jianxin Li、Sen Wang、Shaowen Qin、Xue Li 和 Shuliang Wang（编）。施普林格国际出版公司，Cham, [52] 526 534 10 1007 978 3 030 35231 8_38 –。https://doi.org/。/ - - - - Xinrong Zhang、Yanghao Li、Yuxing Han 和 Jiangtao Wen。2022年。人工智能视频编辑：综述。*Preprints*（2022年2月）。https://doi.org/10.20944/preprints202201. 0016.v2 出版方：Preprints。[53]