·开发研究与工程应用· 文章编号: 1000-3428(2016)12-0314-07 文献标志码: A 中图分类号: TP37

多路高清 YUV 视频 GPU 实时拼接研究

王 震[1,2],许晓航[1,2],王 静[1,2],李 圣[1,2],郑 宏[1,2]

(1. 武汉大学 电子信息学院,武汉 430072; 2. 湖北省视觉感知与智能交通技术研发中心,武汉 430072)

摘 要: 为提高高清视频拼接的实时性能,提出一种基于 GPU 的多路高清 YUV 视频实时拼接方法,推导出 YUV422 图像拼接中的透视模型,并结合计算统一设备架构技术,实现透视变换、无缝融合等关键拼接步骤在 GPU 上的并行优化。在 4 路 1080p 高清视频上的实验结果表明,相比基于 RGB 颜色模型的拼接方法,该方法的实时拼接性能在不同 GPU 架构上有 20% ~40% 的提升,并且在 GRX780 上能达到 33 frame/s 的视频帧率。

关键词: 视频拼接; YUV 颜色模型; 透视模型; 图形处理器; 计算统一设备架构

中文引用格式: 王 震,许晓航,王 静,等. 多路高清 YUV 视频 GPU 实时拼接研究[J]. 计算机工程,2016,42(12):314-320.

英文引用格式: Wang Zhen, Xu Xiaohang, Wang Jing, et al. Research on GPU Real-time Stitching of Multi-channel High-definition YUV Video[J]. Computer Engineering, 2016, 42(12): 314-320.

Research on GPU Real-time Stitching of Multi-channel High-definition YUV Video

WANG Zhen[1,2], XU Xiaohang[1,2], WANG Jing[1,2], LI Sheng[1,2], ZHENG Hong[1,2]

(1. School of Electronic Information, Wuhan University, Wuhan 430072, China;
2. Hubei Research and Development Center of Vision Perception and Intelligent Transportation Technology, Wuhan 430072, China)

【Abstract】 To improve the real-time performance of high-definition video stitching, a stitching method of GPU-based real-time multi-channel high-definition YUV video is proposed. Firstly, the perspective model of YUV422 image stitching is deduced, and then parallel optimization of the stitching steps including perspective warping and image blending are realized and performed on GPU by Compute Unified Device Architecture(CUDA) technology. The experimental results show that, on the condition of implementing four-channel 1080p video stitching, the method in this paper achieves 20% ~ 40% improvement in real-time performance over RGB color model-based stitching method on different GPU, and video frame rate of 33 frame per second on GTX 780.

【Key words】 video stitching; YUV color model; perspective model; Graphics Processing Unit(GPU); Compute Unified Device Architecture(CUDA)

DOI: 10.3969/j.issn.1000-3428.2016.12.053

0 概述

在使用多相机监控大范围区域时,可以通过视频拼接来获得宽视野视频,这样不仅方便连续监控运动目标,而且降低了视频数据的冗余度。但高清视频的拼接需要实时处理大量的图像数据,使用 CPU 计算无法满足实时性要求。为解决此问题,将视频拼接算法移植到 GPU 上,利用 GPU 高带宽和强大的计算资源进行并行加速是一种可靠的解决方案。国内外与此相关的研究成果也较为普遍,如奥斯陆大学 Bagadus 研究组设计了一种应用于足球场的 GPU 实时拼接系统,并展开了一系列深入的研究工作[1-2];文献[3-5]分别在 OpenCl,CUDA 等并行编程平台上优化了拼接算法中的部分步骤。在上述文献中,拼接算法的并行加速均基于 RGB 颜色模型,但视频信号处理传输一般基于 YUV 颜色模型,意味着无论是实时拼接还是后续编解码等功能的扩展工作,都需要进行颜色空间的转换。更重要是,与 YUV 相比,RGB 视频数据冗余度更高,需要实时处理更多的数据,这些问题均对视频拼接的实时性能有所

基金项目: 国家"973"计划项目(2012CB719905)。

作者简介: 王 震(1990—),男,硕士研究生,主研方向为计算机视觉; 许晓航,博士; 王 静、李 圣,硕士; 郑 宏,教授。

收稿日期: 2015-11-03 修回日期: 2015-12-16 E-mail: wz0725@whu.edu.cn

影响。

本文在自动拼接技术[6]的基础上，对 YUV422 图像拼接中的透视模型进行研究，并结合 CUDA 技术，在 Nvidia GPU 上实现多路 YUV 视频的实时拼接。在透视变换、无缝融合等拼接算法的 CUDA 并行设计中，利用全局内存的合并访问[7]，合理配置存储器资源及减少线程分歧数量等优化策略，提高 YUV422 数据的访问和计算效率。

1　视频拼接方法

本文视频拼接方法分为两部分（如图 1 所示），分别是实时视频拼接和拼接模型的参数计算。实时视频拼接的具体步骤如下：首先通过 4 路 200 万高清网络相机获取实时视频，视频数据由网络传输到服务器后，SDI 视频采集卡负责解码数据并同时更新内存中 YUV422 数据的双缓冲队列，接着使用直接内存存取技术（Direct Memory Access，DMA）将 CPU 中的数据以 PCI-EXPRESS 3.0 的峰值带宽传送至 GPU 内存（分别对应图中的 $C_1 \sim C_4$），最后根据 YUV422 图像拼接模型的参数进行实时拼接，其中参数包括图 1 中的 4 路相机的 4 组单应矩阵（$\boldsymbol{H}_1 \sim \boldsymbol{H}_4$）与拼接掩模（$M_1 \sim M_4$）。此外，考虑到数据从 CPU 传输到 GPU 会消耗一定时间，采用异步传输[7]结合双缓冲队列的技术，使数据传输和拼接计算重叠，从而隐藏传输时间。

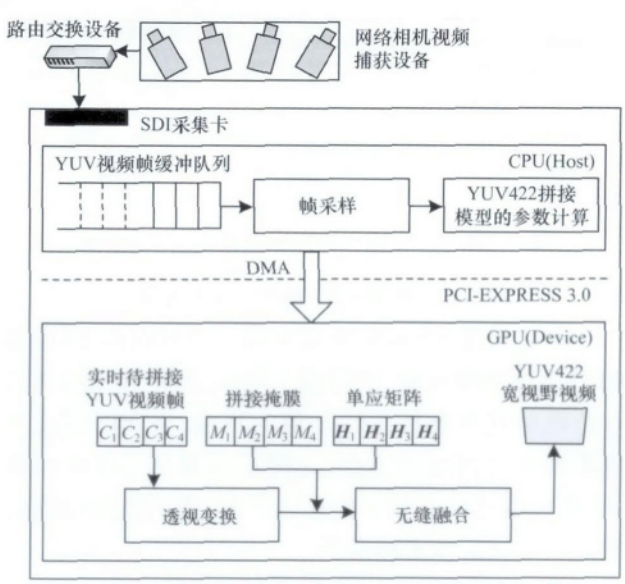

图 1　多路 YUV 视频实时拼接框架

拼接模型的参数计算属于离线部分。由于相机的相对位置和角度不发生变化，因此为避免重复计算拼接参数，只需抽取一组 4 路相机的样本帧，在自动拼接技术和图割法[8]的基础上，结合本文提出的 YUV422 拼接模型获得拼接所需参数并在实时拼接启动前传入 GPU。

2　基于 YUV 的拼接模型研究与 CUDA 实现

2.1　YUV 视频拼接中的内存瓶颈分析

YUV 是一种广泛运用于视频信号传输、采集、编码的色彩空间。与 RGB 不同的是，它将亮度分量 Y 与颜色分量 U，V 分离，并且编码方式使数据冗余度更小，因此，对 YUV 视频数据进行拼接减少了单位时间内处理视频数据的总量。此外，该方法不仅省去了拼接前的 YUV-RGB 空间转换的步骤，而且视频编码传输或存储一般都基于 YUV 格式，从而增强后续多媒体功能的可扩展性。

视频工业界使用的 YUV 编码格式种类繁多，不是每种都适合在 GPU 上存储和计算。常见的 YUV 格式有 YUV420，YUV422 等，它们的定义如下：

1) YUV420 是通过 2∶1 的水平采样和 2∶1 的垂直采样获得，相比 RGB 减少 50% 的数据量。

2) YUV422 是通过 2∶1 的水平采样获得，无垂直采样，相比 RGB 减少 33% 的数据量。

YUV420 的亮度与 2 个色度分量在数据量上无对称性，一般将 Y，U，V 这 3 个分量单独存储，称为平面（planar）类型，常见的编码格式有 YV12，NV12。相比而言，YUV422 相关格式有 YUV2，UYVY 等，它们以 YUYV 或 UYVY 作为一个像素单位进行存储，称为紧缩（packed）类型，这种线性结构与 RGB 类似，下文根据此特点推导出适用于 YUV422 的拼接模型。

由于紧缩类型的 YUV422 数据避免了对 Y，U，V 这 3 个分量的单独存储与计算，不仅减少了非连续内存的读取与操作，而且为 GPU 的内存优化提供了条件。如图 2 所示，GPU 中线程调度的基本单位为线程束（warp），它由 32 个线程组成，每个线程处理一 4 个字节的 YUYV 单位，那么每个线程束的起始访问地址为 128 的倍数，并依次以 4 个字节连续访问128 Byte的内存（图 2 中内存访问从128 Byte起至 256 Byte 结束），这样符合 GPU 中的合并访问模式[7]。由于合并访问将对某块内存的多次访问合并成一次 128 Byte 存储事务，因此提高了 GPU 设备中全局存储器的吞吐量。虽然 RGB 数据能够以 RGBA 格式存储来满足合并访问，但是增加了额外的存储空间与计算量。

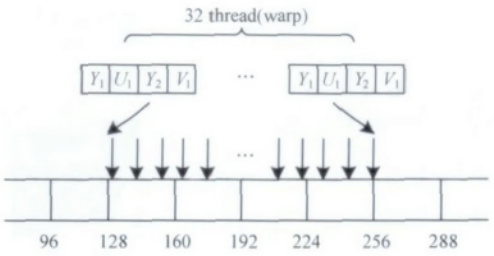

图 2　基于 YUV422 编码结构的合并访问优化

2.2 YUV422 图像拼接的透视模型研究

本文采用 2D 拼接中的平面投影模型[9]，通过一组单应矩阵将多路视频帧投影到同一平面，从而获得宽视野视频，这种变换称为透视变换[10]。下文在 RGB 图像平面投影模型的基础上，研究基于单应矩阵的 YUV422 图像拼接中的透视模型。

假设在 RGB 色彩空间下，$\boldsymbol{H}_i$ 对应第 i 路相机视频帧 $\boldsymbol{I}_i$ 映射到拼接投影面(宽视野视频帧) $\boldsymbol{I}_f$ 的单应矩阵，其定义如下：

$$\boldsymbol{H}_i=\begin{pmatrix}h_{11} & h_{12} & h_{13}\\ h_{21} & h_{22} & h_{23}\\ h_{31} & h_{32} & 1\end{pmatrix} \tag{1}$$

若 $\boldsymbol{p}_i=[x_i \quad y_i \quad 1]^{\mathrm{T}}$，$\boldsymbol{p}_f=[x_f \quad y_f \quad 1]^{\mathrm{T}}$ 分别为 $\boldsymbol{I}_i$ 与 $\boldsymbol{I}_f$ 上对应点的齐次坐标，则有：

$$\boldsymbol{p}_f=\boldsymbol{H}_i\boldsymbol{p}_i \tag{2}$$

x_f，y_f 可分别表示为：

$$x_f=\frac{h_{11}x_i+h_{12}y_i+h_{13}}{h_{31}x_i+h_{32}y_i+h_{33}} \tag{3}$$

$$x_f=\frac{h_{11}x_i+h_{12}y_i+h_{13}}{h_{31}x_i+h_{32}y_i+h_{33}} \tag{4}$$

如图 3 所示，YUV422 图像(图 3 中的深灰色)中一个 YUYV 单位可以表示水平方向上 2 个 RGB 的信息，即它的宽为 RGB 图像(图 3 中的浅灰色)的 1/2，高保持不变，可以通过缩放矩阵 $\boldsymbol{s}$ 来表示 YUV422 与 RGB 图像之间的线性关系，其中 $\boldsymbol{s}$ 的表达式如下：

$$\boldsymbol{s}=\begin{pmatrix}\lambda_1 & 0 & 0\\ 0 & \lambda_2 & 0\\ 0 & 0 & 1\end{pmatrix} \tag{5}$$

其中，$\lambda_1=0.5$，$\lambda_2=1$ 分别为 YUV422 图像与 RGB 图像的宽高之比。

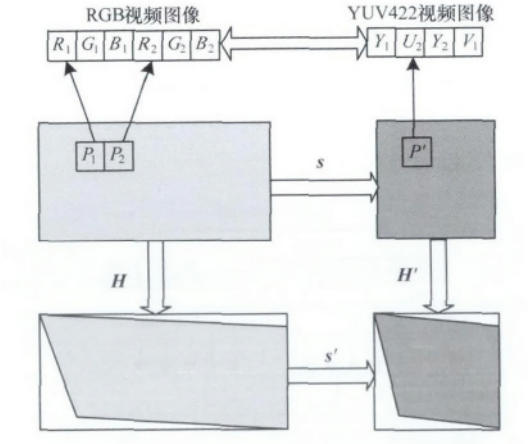

图 3 RGB 与 YUV422 拼接模型的转换关系

在 YUV422 色彩空间下，设 $\boldsymbol{p}_i'=[x_i' \quad y_i' \quad 1]^{\mathrm{T}}$，$\boldsymbol{p}_f'=[x_f' \quad y_f' \quad 1]^{\mathrm{T}}$ 分别为 $\boldsymbol{I}_i$ 与 $\boldsymbol{I}_f$ 上对应点的齐次坐标，那么 $\boldsymbol{I}_i$ 的齐次坐标的 YUV422-RGB 转换表达式为：

$$\boldsymbol{p}_i'=s\boldsymbol{p}_i \tag{6}$$

同理，$\boldsymbol{I}_f$ 的相关坐标转换的表达式为：

$$\boldsymbol{p}_f'=s\boldsymbol{p}_f \tag{7}$$

在 YUV422 色彩空间下，又设 $\boldsymbol{H}_i'$ 为 $\boldsymbol{I}_i$ 映射到 $\boldsymbol{I}_f$ 的单应矩阵(如图 3 深灰色形状变换所示)，$\boldsymbol{p}_i'$ 到 $\boldsymbol{p}_f'$ 的齐次坐标转换如下所示：

$$\boldsymbol{p}_f'=\boldsymbol{H}_i'\boldsymbol{p}_i' \tag{8}$$

将式(2)、式(6)、式(7)代入等式(8)得到：

$$\boldsymbol{H}_i'=s\boldsymbol{H}_i\boldsymbol{s}^{-1} \tag{9}$$

通过式(8)即可推导出 YUV422 图像拼接中进行透视变换的单应矩阵 $\boldsymbol{H}_i'$。

图 4 为本文获取拼接参数的完整步骤，首先获取 4 路相机视频的样本帧，并将其转换为 RGB 图片，接着使用自动拼接技术，对这组图片进行特征点提取、特征点匹配、相机内外参数估计等步骤后，得到式(2)中的单应矩阵 $\boldsymbol{H}_i$($i=1,2,3,4$)，然后利用图割法生成包含最优缝合路径的拼接掩模，最后根据式(9)将 $\boldsymbol{H}_i$ 转换成适用于 YUV422 图像拼接的 $\boldsymbol{H}_i'$。

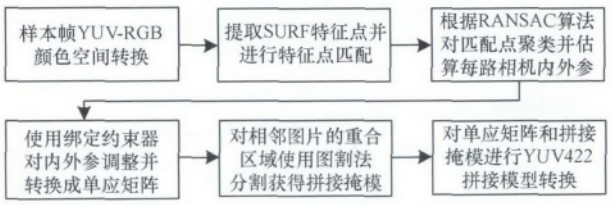

图 4 获取 YUV422 拼接模型参数的完整步骤

2.3 YUV422 图像透视变换的 CUDA 实现

本文使用 CUDA 技术实现了 YUV422 图像在 GPU 上的透视变换，如算法 1 所示，其中利用纹理内存拾取时具有硬件线性差值的特性，将 YUV422 图像数据绑定到纹理内存，这样不仅满足合并访问模式，还可以在映射变换坐标计算时，使用纹理的滤波模式自动完成像素的插值。

算法 1 YUV422 图像透视变换的 CUDA 实现

```
1. texture < unsigned char,2 >  tex//创建二维纹理
2. cudaBindTextureToArray ( tex, YUYV) //绑定 YUV422 //数据
3. dx = blockIdx. x* blockDim. y + threadIdx. x//输出图像 //列坐标索引
4. dy = blockIdx. y* blockDim. y + threadIdx. y//输出图像 //列坐标索引
```

```
5. norm = h31 * dx + h32 * dy + 1 //计算归一化系数
6. sx = (h11 * dx + h12 * dy + h13) /norm//待拼接图像列
//坐标索引
7. sy = (h21 * dx + h22 * dy + h23) /norm//待拼接图像行
//坐标索引
8. dst(dy,dx) = tex2D(tex,float(sx),float(sy)) //将像素
//值存入输出图像
9. cudaUnbindTexture(tex) //解绑纹理
```

3　YUV422 图像的无缝融合与 CUDA 实现

视频拼接中无缝融合相比静态图像拼接更重要,因为运动物体频繁穿越接缝会放大接缝之间色彩不连续、视差错误等问题带来的困扰。此外,融合算法的并行加速也是性能优化的关键。文献[4,5]在 GPU 上分别实现了动态接缝结合色彩纠正[11]、加权融合[12]等方法来解决接缝问题,但动态接缝会带来视频闪动,而色彩纠正、加权融和等方法难以同时消除相邻图像亮度和色彩的差异,并且 RGB 数据增加了额外的计算量。下文在使用图割法生成静态接缝的基础上,结合 GPU 架构的特点,使用 CUDA 技术实现了 YUV422 图像羽化融合、多波段融合等融合方法获得了良好的视觉效果,并具备较好的实时性能。

3.1　YUV422 图像羽化融合的 CUDA 实现

拼接中常见的融合算法有线性加权[5]和非线性加权 2 种[13]。下文使用的羽化融合[9]是一种结合拼接掩模的非线性加权融合方法,它的权值基于掩模中的前景点与背景的最近欧式距离。相比线性加权方法,它更好地解决了亮度差异带来的问题。而 YUV422 图像的亮度分量与颜色分量相互独立的特点使这一优势更加明显。

由图割法获得的拼接掩模为一幅二值图,像素值为 255 表示前景(融合时取此像素点),像素值为 0 表示背景(融合时丢弃此像素点),接缝为前景与背景的分界线。对此二值图做距离变换[14]获得一幅灰度图像,称为距离表,其中每个像素的灰度值表示该像素到背景部分(接缝)的最近欧式距离。设像素 p 为掩模中的某一点,在距离表中它对应的值为 $d_m(p,i)$。

$$d_m(p,i) = \begin{cases} 0, p = 0 \\ \min(distance(p,p_0)), p = 255 \end{cases} \tag{10}$$

其中,p_0 表示接缝上的点(即可能成为距离 p 最近的背景点)。

本文通过距离系数 α 和阈值 1 来设置融合的有效范围,同时获得对应第 i 路视频的融合权值图为 w_i,可以表示为:

$$w_i(p) = \begin{cases} 1, \alpha d_m(p,i) > 1 \\ \alpha d_m(p,i), \alpha d_m(p,i) \leqslant 1 \end{cases} \tag{11}$$

其中,α 设为 0.01,即默认相邻两路视频进行羽化融合的区域为 200 个像素单位。

由于距离变换中存在大量判断引起的分支循环,并不适合 GPU 的计算模型,而权值图只需计算一次,因此首先在 CPU 中计算出权值图,后将其传入 GPU,最后对 YUYV 的 4 个分量分别进行求和并归一化获得融合的像素值,其表达式如下:

$$I_f(p) = \frac{\sum_{i=1}^{N} w_i(p) I_i(p)}{\sum_{i=1}^{N} w_i(p)} \tag{12}$$

其中,$I_i(p)$ 为第 i 路相机的像素点 p 的值;$I_f(p)$ 为像素点 p 融合后的值。YUV422 图像加权与归一化的 CUDA 实现如算法 2 所示。

算法 2　YUV222 图像羽化中加权与归一化的 CUDA 实现

```
1. sx = blockIdx.x * blockDim.y + threadIdx.x//图像列
//坐标
2. sy = blockIdx.y * blockDim.y + threadIdx.y//图像行
//坐标
//对 YUV422 图像进行加权
3. Ii(p(sy,sx)Y1) += wi(p'(sy,sx)) //对 Y1 分量加权
4. Ii(p(sy,sx)U1) += wi(p'(sy,sx)) //对 U1 分量加权
5. Ii(p(sy,sx)Y2) += wi(p'(sy,sx)) //对 Y2 分量加权
6. Ii(p(sy,sx)V1) += wi(p'(sy,sx)) //对 V1 分量加权
//加权完毕后进行归一化操作
7. Ii(p(sy,sx)Y1) /= wi(p'(sy,sx)) //对 Y1 分量归一化
8. Ii(p(sy,sx)U1) /= wi(p'(sy,sx)) //对 U1 分量归一化
9. Ii(p(sy,sx)Y2) /= wi(p'(sy,sx)) //对 Y2 分量归一化
10. Ii(p(sy,sx)V1) /= wi(p'(sy,sx)) //对 V1 分量归一化
```

3.2　YUV422 图像多波段融合的 CUDA 实现

羽化融合虽然可以解决亮度差异所带来的影响,但从式(11)中可知,对应每个像素的权值 $w_i(p)$ 均小于 1,那么加权后会降低画面的对比度,同时也无法兼顾色彩信息的表达。针对此问题,文献[15]提出的多波段融合是一种有效的解决方案。该方案通过构造多层拉普拉斯金字塔,将原图像分解成不同分辨率(金字塔的不同层次)的图像,并对它们采用不同的规则融合,最后向上重构,将不同分辨率图像的细节特征融合在一起,从而提高融合效果。下文对该方法的 CUDA 实现进行研究。

YUV422 图像的拉普拉斯金字塔的分解和重构

与 RGB 一样，也是基于高斯金字塔来实现的，分解是对相邻两层的高斯金字塔差分获得，而重构是对分解的结果求和获得。在构造高斯金字塔中有 2 个重要的步骤，分别是 Reduce 操作和 Expand 操作，假设透视变换过的 YUV422 视频图像为高斯金字塔的第 0 层，前者表示对第 $L-1$ 层进行高斯滤波和向下采样获得第 L 层，后者表示对第 L 层进行高斯滤波和向上采样获得第 $L-1$ 层。根据以上定义，如果在 GPU 中构造 5 层拉普拉斯金字塔，4 路实时 YUV422 视频帧需要做 20 次 Reduce 和 20 次 Expand 操作，意味着通过 GPU 并行来提高 Reduce 和 Expand 效率是优化实时性能的重要步骤。

设第 L 层高斯金字塔图像为 G_L，G_{L-1} 通过 Reduce 运算获得 G_L 的等式，具体如下：

$$G_L(i,j)=\sum_{m=-2}^{2}\sum_{n=-2}^{2}w(m,n)\,G_{L-1}(2i+m,2j+n) \tag{13}$$

其中，i,j 分别表示第 L 层图像行和列的索引，$w(m,n)$ 是一个 5×5 线性可分的高斯核，它可以由 $\hat{w}(5)=\frac{1}{16}$[1 4 6 4 1]的外积来表示，那么利用此性质，可将二维的卷积运算拆分成行方向和列方向上的一维卷积，从而降低时间复杂度。在 GPU 硬件架构中，每个多流处理器都有一定大小的片上共享内存，它们相比全局内存能够提供 10:1 的内存存取加速比[7]。而参与行列卷积计算的图像数据和中间结果需要重复使用，可以将这些热点数据存入共享内存，从而提高内存的访问带宽。Reduce 的 CUDA 实现如算法 3 所示。

算法 3 YUV422 图像 Reduce 操作的 CUDA 实现

```
1. sx = blockIdx.x * blockDim.x + threadIdx.x //高斯金
//字塔第 L-1 层图像列坐标索引
2. sy = 2 * blockIdx.y //高斯金字塔第 L-1 层图像行坐标
//索引
3. dy = blockIdx.y //第 L 层图像行坐标索引
4. shared_mem[threadIdx.x] = ∑w(m) G_{L-1}(sy + m, sx)
//行卷积计算，将中间结果存入共享内存
5. __syncthreads() //线程块内线程同步，步骤 4 计算完毕
6. if threadIdx.x < blockDim.x/2 then
7. dx = (blockIdx.x * blockDim.x + 2 * threadIdx.x) /2
//第 L-1 层图像列坐标索引
8. G_L(dy, dx) = ∑w(m) shared_mem(2 * threadIdx.x + n)
//列卷积得到最后结果并存入全局内存
9. endif
```

Expand 相对 Reduce 操作是一个相反的过程，在 G_l 第 $k-1$ 次 Expand 结果的基础上进行第 k 次 Expand 的过程可由式(14)表示：

$$G_{l,k}(i,j)=4\sum_{m=-2}^{2}\sum_{n=-2}^{2}w(m,n)\,G_{l,k-1}\left(\frac{i-m}{2},\frac{j-n}{2}\right) \tag{14}$$

其中，为保证采样时所取像素点的行列索引为整数，计算时只取使 $i-m,j-n$ 能被 2 整除的行列坐标索引。Expand 的 CUDA 实现如算法 4 所示。

算法 4 YUV422 图像 Expand 操作的 CUDA 实现

```
1. dx = blockIdx.x * blockDim.x + threadIdx.x//第 k 次
//Expand 操作结果的图像列坐标索引
2. dy = blockIdx.y * blockDim.y + threadIdx.y//第 k 次
//Expand 操作结果的图像行坐标索引
3. shared_meml = G_{L,k-1}(dy/2, dx/2) //将第 k-1 次 Expand 操
//作结果的图像存入共享内存
4. __syncthreads() //线程块内部线程同步，确保上步赋值
//完毕
5. if threadIdx.y is even then
6. shared_mem2 = ∑w(m) shared_meml [threadIdx.y/2]
[threadId.x - n)/2] //列卷积运算
7. endif
8. __syncthreads() //线程块内部线程同步，步骤 6 计算完毕
9. G_{L,k}(dy, dx) = ∑w(m) shared_mem2(threadIdx.y + m,
threadIdx.x) //行卷积运算得到最后结果并存入全局内存
```

4 实验结果与分析

4.1 实验环境

本文实验在 3 块不同架构的 GPU 上进行，分别为 GTX480，GTX780，GTX750，它们的硬件参数如表 1所示。

表 1 实验使用的 GPU 设备相关参数

性能指标	GTX480	GTX780	GTX750
核心架构	Fermi	Kepler	Maxwell
时钟频率/MHz	1 401	836	1 020
计算核心数	480	2304	512
多流处理器数	15	12	4
常驻线程数	23 040	24 576	8 192

4.2 性能分析与总结

本文实验在 GPU 上实现了 4 路 200 万像素网络相机的实时视频拼接，并获得分辨率为 760 万的宽视野视频。在此条件下，下文对本文方法与基于 RGB 视频数据的实时拼接性能进行详细对比与性能分析。

4.2.1　透视变换的性能对比

表 2 为 RGB,RGBA,YUV422 图像数据在 3 种不同架构的 GPU 上进行透视变换的性能对比。测试结果表明,使用 YUV422 图像数据相比 RGB 性能至少提高了 40%,虽然 RGBA 可以通过合并访问来降低访问延时,但 YUV422 数据冗余度较低,相比之下性能也有至少 30% 的提高。

表 2　图像数据透视变换的实时性能比较　　ms

图像	GTX480	GTX780	GTX750
RGB	5.13	3.9	6.40
RGBA	3.87	2.72	4.82
YUV422	2.56	2.08	3.65

4.2.2　图像融合的性能对比和效果分析

表 3 为 RGB,RGBA,YUV422 图像数据在 3 种不同架构的 GPU 上进行无缝融合的性能对比。由于直接拼接和羽化融合的计算模型中的分支指令较少,它们均能达到极高的实时性能,而多波段融合消耗的时间为羽化融合的 3 倍左右。

表 3　图像融合的实时性能比较　　ms

图像	GTX480		GTX780		GTX750	
	羽化	多波段	羽化	多波段	羽化	多波段
RGB	7.83	36.60	7.00	32.72	13.30	51.00
RGBA	8.42	42.00	7.34	35.73	14.40	59.40
YUV422	5.39	30.00	5.08	27.38	7.05	39.10

如算法 3、算法 4 所示,尽管利用共享内存来提高热点数据的访问效率,但是片上共享内存和寄存器资源有限,无法满足所有并发线程的内存请求,导致部分线程处在阻塞状态。不仅如此,行列卷积在线程块内部存在同步操作,那么较快达到同步点的线程也会处在阻塞状态。这些问题均给多波段融合性能带来了一定影响,但是牺牲性能带来的好处是,当有运动物体穿过接缝时多波段融合的效果远好于另外 2 种方法。

图 5 为 3 种方法的融合效果并用黑框标记了较为明显的接缝处,从穿过接缝的车尾可以发现相比羽化融合,多波段融合的结果几乎完全消除了色彩和亮度的差异(为保证可视的对比效果,图 5 中的拼接结果均为 YUV422 转化为 RGB 后的图像)。

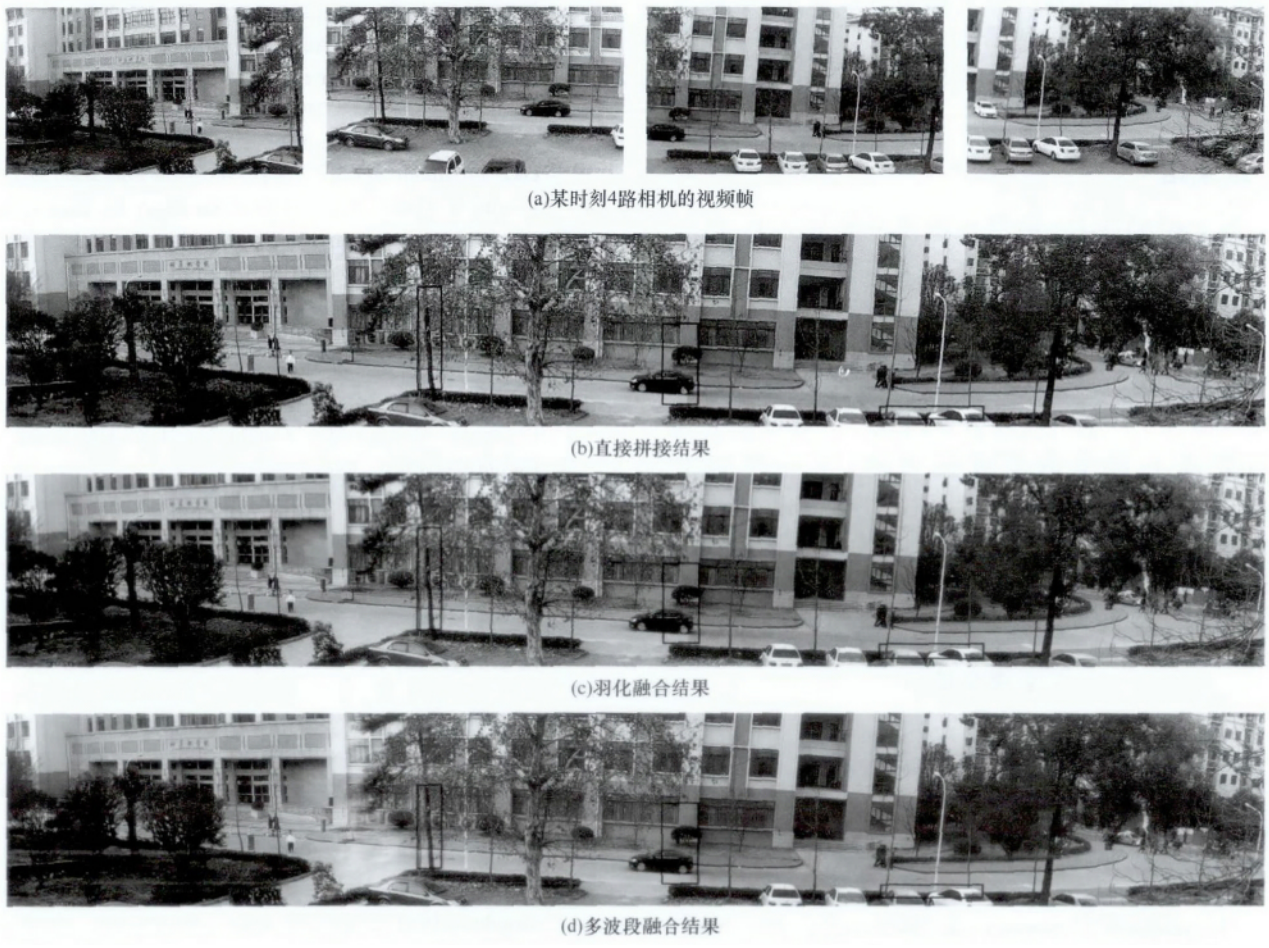

图 5　采用不同融合方法的拼接结果

4.2.3 实时视频拼接总体性能分析

表 4 为完整拼接流程在不同架构 GPU 上的总体性能对比以及本文方法所获得的加速比，实验数据表明在不同架构的 GPU 上，相比 RGB 视频拼接方法，本文方法总体的实时性能有 20% ~40% 的提高。

表 4 YUV422 与 RGB 相关格式视频图像的完整拼接时间以及相对加速比

序号	性能指标	GTX480			GTX780			GTX750		
		直接拼接	羽化	多波段	直接拼接	羽化	多波段	直接拼接	羽化	多波段
1	RGB 拼接时间/ms	8.58	15.36	44.186	7.38	13.44	39.12	11.11	22.64	60.36
	YUV422 拼接时间/ms	4.90	9.87	34.45	4.67	9.44	30.94	7.02	15.18	45.21
	加速比/%	42.89	35.70	22.03	36.72	29.76	21.42	36.81	30.16	25.24
2	RGBA 拼接时间/ms	7.35	15.93	49.43	6.20	13.62	43.02	10.01	25.04	70.01
	YUV422 拼接时间/ms	4.90	9.87	34.45	4.67	9.44	30.94	7.02	15.18	45.21
	加速比/%	33.33	38.04	30.30	24.67	30.69	28.54	29.87	39.37	35.42

羽化融合在 3 块不同的显卡上都可以达到超过 100 frame/s 的拼接效率，而在硬件指标最高的 GTX780 上，采用多波段融合的方法在拼接帧率已经接近 33 frame/s，能够同时满足高质量视觉效果和高帧率的要求。但是值得注意的是，该方法在最新 Maxwell 架构的 GTX750 上的性能低于 15 frame/s，明显差于其他 2 块显卡，从表 1 可知 GTX750 的多流处理器仅为 4 个，能够并发的线程数量以及片上的共享内存总量大幅落后其他型号的显卡，因此在计算中隐藏访问内存延迟能力较弱，说明对实时性能具有一定要求的图像处理应用，多流处理器的数量是较重要的硬件指标。

5 结束语

本文在 YUV422 图像拼接模型的基础上，结合 CUDA 技术在 GPU 上实现了一种多路高清 YUV 视频实时拼接方法。相比 RGB 视频拼接方法，本文方法既提高了视频拼接的实时性能，又减少了色彩空间转换的步骤，为后续视频编码、传输等多媒体功能的拓展提供了便捷的接口数据格式。为获取更大视野与更高分辨率的宽视野视频，下一步将对 YUV422 图像数据在柱面、球面上的拼接投影模型进行研究。此外，将本文方法移植到 GPU 集群来满足更多路高清视频的实时拼接也将作为后续研究的重点。

参考文献

[1] Stensland H K, Gaddam V R, Tennoe M, et al. Bagadus: An Integrated Real-time System for Soccer Analytics [J]. ACM Transactions on Multimedia Computing, Communications, and Applications, 2014, 10(1s): 7.

[2] Kellerer L, Gaddam V R, Langseth R, et al. Real-time HDR Panorama Video [C]//Proceedings of ACM International Conference on Multimedia. New York, USA: ACM Press, 2014: 1205-1208.

[3] Liao W S, Hsieh T J, Chang Y L. GPU Parallel Computing of Spherical Panorama Video Stitching [C]//Proceedings of the 18th International Conference on Parallel and Distributed Systems. Washington D. C., USA: IEEE Press, 2012: 890-895.

[4] Meng X, Wang W, Leong B. SkyStitch: A Cooperative Multi-UAV-based Real-time Video Surveillance System with Stitching [C]//Proceedings of the 23rd Annual ACM Conference on Multimedia Conference. New York, USA: ACM Press, 2015: 261-270.

[5] 吴 健，兰时勇，黄飞虎. 一种多路快速视频拼接系统设计与实现[J]. 计算机工程，2014，40(2)：208-211，218.

[6] Brown M, Lowe D G. Automatic Panoramic Image Stitching Using Invariant Features [J]. International Journal of Computer Vision, 2007, 74(1): 59-73.

[7] Wilt N. The Cuda Handbook: A Comprehensive Guide to GPU Programming [M]. [S.l.]: Pearson Education, 2013.

[8] Kwatra V, Schödl A, Essa I, et al. Graphcut Textures: Image and Video Synthesis Using Graph Cuts [J]. ACM Transactions on Graphics, 2003, 22(3): 277-286.

[9] Szeliski R. Image Alignment and Stitching: A Tutorial [J]. Foundations and Trends in Computer Graphics and Vision, 2005, 2(11/12): 273-292.

[10] Hartley R, Zisserman A. Multiple View Geometry in Computer Vision [M]. Cambridge, UK: University of Cambridge Press, 2003.

[11] Xiong Y, Pulli K. Color Correction for Mobile Panorama Imaging [C]//Proceedings of the 1st International Conference on Internet Multimedia Computing and Service. New York, USA: ACM Press, 2009: 219-226.

[12] 宋宝森. 全景图像拼接方法研究与实现[D]. 哈尔滨：哈尔滨工程大学，2012.

[13] 王小强，陈临强，梁 旭. 实时全自动视频拼接方法[J]. 计算机工程，2011，37(5)：291-292.

[14] Shih F Y, Wu Y T. Fast Euclidean Distance Transformation in Two Scans Using a 3 × 3 Neighborhood [J]. Computer Vision and Image Understanding, 2004, 93(2): 195-205.

[15] Burt P J, Adelson E H. A Multiresolution Spline with Application to Image Mosaics [J]. ACM Transactions on Graphics, 1983, 2(4): 217-236.

编辑 陆燕菲